IEEE 2005 Symposium on Computational Intelligence and Games

CIG'05

April 4-6 2005

Essex University, Colchester, Essex, UK

Graham Kendall and Simon Lucas (editors)

Contents

Poster Presentations

Preface

Welcome to the inaugural 2005 IEEE Symposium on Computational Intelligence and Games. This symposium marks a milestone in the development of machine learning, particularly using methods such as neural, fuzzy, and evolutionary computing. Let me start by thanking Dr. Simon Lucas and Dr. Graham Kendall for inviting me to write this preface. It's an honor to have this opportunity.

Games are a very general way of describing the interaction between agents acting in an environment. Although we usually think of games in terms of competition, games do not have to be competitive: Players can be cooperative, neutral, or even unaware that they are playing the same game. The broad framework of games encompasses many familiar favorites, such as chess, checkers (draughts), tic-tac-toe (naughts and crosses), go, reversi, backgammon, awari, poker, blackjack, arcade and video games, and so forth. It also encompasses economic, social, and evolutionary games, such as hawk-dove, the prisoner's dilemma, and the minority game. Any time one or more players must allocate resources to achieve a goal in light of an environment, those players are playing a game.

Artificial intelligence (AI) researchers have used games as test beds for their approaches for decades. Many of the seminal contributions to artificial intelligence stem from the early work of Alan Turing, Claude Shannon, Arthur Samuel, and others, who tackled the challenge of programming computers to play familiar games such as chess and checkers. The fundamental concepts of minimax, reinforcement learning, tree search, evaluation functions, each have roots in these early works.

In the 1940s and 1950s, when computer science and engineering was in its infancy, the prospects of successfully programming a computer to defeat a human master at any significant game of skill were dim, even if hopes were high. More recently, seeing a computer defeat even a human grand master at chess or checkers, or many other familiar games, is not quite commonplace, but not as awe-inspiring as it was only a decade ago. Computers are now so fast and programming environments are so easy to work with that brute force methods of traditional AI are sufficient to compete with or even defeat the best human players in the world at all but a few of our common board games.

Although it might be controversial, I believe that the success of Deep Blue (the chess program that defeated Garry Kasparov at chess), Chinook (the checkers program that earned the title of world champion in the mid-1990s), and other similar programs mark the end of a long journey - but not the journey started by Turing, Shannon, and Samuel - but rather a different journey.

The laudatory success of these traditional AI programs has once again pointed to the limitations of these programs. Everything they "know" is preprogrammed. They do not adapt to new players, they assume their opponent examines a position in a similar way as they do, they assume the other player will always seek to maximize damage, and most importantly, they do not teach themselves how to improve beyond some rudimentary learning that might be exhibited in completing a bigger lookup table of best moves or winning conditions. This is not what Samuel and others had in mind

when asking how we might make a computer do something without telling it how to do it, that is, to learn to do it for itself. Deep Blue, Chinook, and other superlative programs have closed the door on one era of AI. As one door closes, another opens.

Computational intelligence methods offer the possibility to open this new door. We have already seen examples of how neural, fuzzy, and evolutionary computing methods can allow a computer to learn how to play a game at a very high level of competency while relying initially on little more than primitive knowledge about the game. Some of those examples include my own efforts with Blondie24 and Blondie25 in checkers and chess, respectively, and perhaps that is in part why I was asked to contribute this preface, but there are many other examples to reflect on, and now many more examples that the reader can find in these proceedings. No doubt there will be many more in future proceedings.

Computational intelligence methods offer diverse advantages. One is the ability for a computer to teach itself how to play complex games using self-play. Another is the relatively easy manner in which these methods may be hybridized with human knowledge or other traditional AI methods, to leapfrog over what any one approach can do alone. Yet another is the ability to examine the emergent properties of evolutionary systems under diverse rules of engagement. It is possible to examine the conditions that are necessary to foster cooperation in different otherwise competitive situations, to foster maximum utilization of resources when they are limited, and when players might simply opt out of playing a game altogether. Computational intelligence offers a versatile suite of tools that will take us further on the journey to making machines intelligent.

If you can imagine the excitement that filled the minds of the people exploring AI and games in the 1940s and 1950s, I truly believe what we are doing now is even more exciting. We all play games, every day. We decide how to allocate our time or other assets to achieve our objectives. Life itself is a game, and the contributors to this symposium are players, just as are you, the reader. Not all games are fun to play, but this one is, and if you aren't already playing, I wholeheartedly encourage you to get in the game. I hope you'll find it as rewarding as I have, and I hope to see you at the next CIG symposium. There is a long journey ahead and it is ours to create.

David B. Fogel
Chief Executive Officer
Natural Selection, Inc.
La Jolla, CA, USA

Acknowledgements

This symposium could not have taken place without the help of a great many people and organisations.

We would like to thank the IEEE Computational Intelligence Society for supporting this symposium. They have provided both their knowledge and experience as well as providing the budgetary mechanisms to allow us to establish this inaugural event.

Smita Desai was one of our main contacts at IEEE and she guided us through the financial decisions that we had to make. As this was our first IEEE symposium as co-chairs, we found her help, advice and experience invaluable.

The UK Engineering and Physical Sciences Research Council (EPSRC) provided financial support. Without their support we would only have had one plenary speaker and their assistance has added significantly to the profile and quality of the symposium.

The on-line review system was developed by Tomasz Cholewo. This system made the organisation of the conference a lot easier than it might otherwise had been.

We would like to thank the four plenary speakers (Jordan B. Pollack, Risto Miikkulainen, Martin Mueller, Jaap van den Herik). Their willingness to present at this inaugural event has greatly helped raised the profile of the symposium.

We are deeply grateful to the three tutorial speakers (Andries P. Engelbrecht, Evan J. Hughes, Thomas P. Runarsson) who all gave up part of their weekend to allow others to learn from their knowledge and experience.

The program committee (see next page) did an excellent job in reviewing all the submitted papers. The legacy of a symposium is largely judged by the people represented on the program committee and we realise that we have been fortunate to have internationally recognised figures in computational intelligence and games represented at this symposium.

Lastly, but by no means least, we would like to acknowledge the contribution of David Fogel. We would to thank him for having the confidence in us to realise his idea for this symposium. David has also been a constant source of advice and support which we have found invaluable.

Graham Kendall and Simon Lucas
The University of Nottingham and the University of Essex

Program Committee

- Dan Ashlock, University of Guelph, Canada
- Ian Badcoe, ProFactum Software, UK
- Luigi Barone, The University of Western Australia, Australia
- Alan Blair, University of New South Wales, Australia
- Bruno Bouzy, Universite Rene Descartes, France
- Michael Buro, University of Alberta, Canada
- Murray Campbell, IBM T.J. Watson Research Center, USA
- Darryl Charles, University of Ulster, UK
- Ke Chen, University of Manchester, UK
- Sung-Bae Cho, Yonsei University, Korea
- Paul Darwen, Protagonist Pty Ltd, Australia
- Abdennour El Rhalibi, Liverpool John Moores University, UK
- Andries Engelbrecht, University of Pretoria, South Africa
- Thomas English, The Tom English Project, USA
- Maria Fasli, University of Essex, UK
- David Fogel, Natural Selection, Inc., USA
- Tim Hays, Natural Selection, Inc., USA
- Jaap van den Herik, Universiteit Maastricht, The Netherlands
- Phil Hingston, Edith Cowan University, Australia
- Howard A. Landman, Nanon, USA
- Huosheng Hu, University of Essex, UK
- Evan J. Hughes, Cranfield University, UK
- Doran Jim, University of Essex, UK
- Graham Kendall, University of Nottingham, UK (co-chair)
- Thiemo Krink, University of Aarhus, Denmark
- Sushil J. Louis , University of Nevada, Reno
- Simon Lucas, University of Essex, UK (co-chair)
- Stephen McGlinchey, University of Paisley, UK
- Risto Miikkulainen, University of Texas at Austin, USA
- Martin Muller, University of Alberta, Canada
- Jordan Pollack, Brandeis University, USA
- Thomas Philip Runarsson, University of Iceland, Iceland
- Jonathan Schaeffer, University of Alberta, Canada
- Lee Spector, Hampshire College, USA
- Rene Thomsen, University of Aarhus, Denmark
- Nikos Vlassis, Univ. of Amsterdam, The Netherlands
- Lyndon While, University of Western Australia, Australia
- Xin Yao, University of Birmingham, UK

Plenary Presentations

Is Progress Possible?

Jordan Pollack
DEMO Laboratory
Brandeis University
Waltham MA 02454
pollack@cs.brandeis.edu

For the past decade my students and I have worked on coevolutionary learning, both in theory, and in practice. Coevolution tries to formalize a computational "arms race" which would lead to the emergence of sophisticated design WITHOUT an intelligent designer or his fingerprints[1] left in the choice of data representations and fitness function.

Coevolution thus strives to get maximal learning from as little bias as possible, and often takes the shape of a game tournament where the players who do well replicate (with mutation) faster than the losers. The fitness function, rather than being absolute, is thus relative to the current population. We have had many successes, such as in learning game strategies in Tic Tac Toe (Angeline & Pollack 1993) or Backgammon (Pollack & Blair, 1998), in solving problems like sorting networks and CA rules (Juille & Pollack 1998, 2000), and in co-designing Robot morphology and control (Funes & Pollack, 1998, Lipson & Pollack 2000, Hornby & Pollack, 2002).

But we find that often, the competitive structure of coevolution leads to maladies like winner-take-all equilibria, boom and bust cycles of memory loss, and mediocre stable states (Ficici & Pollack, 1998) where an oligarchy arises which survives by excluding innovation rather than embracing it.

One of the new instruments which has emerged is the use of adaptive agents themselves to measure and reveal their own incentive infrastructure, rather than assuming it is what we hoped (Ficic, Melnik, & Pollack, 2000). We have elaborated these learning dynamics using a particularly simple class of model called the "numbers games" (Watson & Pollack, 2001), and have been looking at underlying issues and methods for treating the maladies, including Pareto coevolution (Ficici & Pollack 2001), emergent dimensional analysis (Bucci & Pollack 2002, 2003), methods for preserving information which can drive learning (Dejong & Pollack, 2004) and idealized memory for what has been learned (Ficici & Pollack, 2003).

Nevertheless, something is wrong if after many years of believing that competition is the central organizing principle of Nature, we have yet to have in hand a convincing mathematical or computational demonstration that competition between self-interested players– without a central government - can lead to sustained innovation.

Is there a missing principle, a different mechanism design under which self-interested players can optimize their own utility, yet as a whole the population keeps improving at the game? If so, and if we discover this "principle of progress" in the realm of computational games, would it transfer it to human social organization?

I will describe one candidate we have been working on, a game metaphor called "The Teacher's Dilemma", which can explain why peer learners in many situations are actually motivated to stop learning. The Teacher's Dilemma also suggests how to design learning communities motivated to produce and maintain their own gradient.

[1] Fingerprints refers to intentional inductive bias, the gradients engineered into the learning environment and representations of evolutionary and neural learning systems.

References

Angeline, P. J. & Pollack, J. B. (1993) Competitive environments evolve better solutions to complex problems. Fifth International Conference on Genetic Algorithms. 264-270.

Bucci, A. and Pollack, J.B. (2002). Order-theoretic Analysis of Coevolution Problems: Coevolutionary Statics. 2002 Genetic and Evolutionary Computation Conference Workshop: Understanding Coevolution.

Bucci, A. and Pollack, J.B. (2003). *A Mathematical Framework for the Study of Coevolution*. FOGA 7: Proceedings of the Foundations of Genetic Algorithms Workshop.

De Jong, E.D. and J.B. Pollack (2004). Ideal Evaluation from Coevolution. *Evolutionary Computation*, Vol. 12, Issue 2, pp. 159-192

Ficici, S. & Pollack, J. (1998) "Challenges in Coevolutionary Learning: Arms-Race Dynamics, Open-Endedness, and Mediocre Stable States." Proceedings of the Sixth International Conference on Artificial Life. Adami, Belew, Kitano, Taylor, eds. Cambridge: MIT Press

Ficici, Sevan G. and Pollack, Jordan B. (2001). Pareto Optimality in Coevolutionary Learning. Advances in Artificial Life: 6th European Conference (ECAL 2001) J. Kelemen, P. Sosik (eds.), Springer, 2001.

Ficici, S, & Pollack, J. B. (2003) A Game-Theoretic Memory Mechanism for Coevolution. Proceedings of the 2003 Genetic and Evolutionary Computation Conference, Springer Verlag, 2003

Funes, P & Pollack, J. B (1998). Evolution of modular structures: Steps towards adaptive robot bodies. Artificial Life, 4, 336-357.

Hornby, G. S. and Pollack, J. B. (2002) Creating High-level Components with Generative Representation for Body-Brain Evolution. Artificial Life 8,3. 223-246

Juille, H. and Pollack, J. B. (1998) Coevolving the ``Ideal" Trainer: Application to the Discovery of Cellular Automata Rules. Proceedings of the Third Annual Genetic Programming Conference (GP-98), Madison, Wisconsin, USA, July 22-25, 1998.

Hugues Juille and Jordan B. Pollack, (2000) ``Coevolutionary Learning and the Design of Complex Systems" in Advances in Complex Systems, pp.371-393.

Pollack. J B. & Blair A. (1998). Co-Evolution in the Successful Learning of Backgammon Strategy. Machine Learning, 32, 225-240.

Lipson H., Pollack J. B., 2000, “Automatic design and Manufacture of Robotic Lifeforms”, Nature 406, pp. 974-978.

Watson RA & Pollack JB, 2001, "Coevolutionary Dynamics in a Minimal Substrate", in GECCO-2001: Proceedings of the Genetic and Evolutionary Computation Conference. Spector, L, et al, editors. Morgan Kaufmann, 2001

Creating Intelligent Agents through Neuroevolution

Risto Miikkulainen
The University of Texas at Austin
risto@cs.utexas.edu

Abstract- The main difficulty in creating artificial agents is that intelligent behavior is hard to describe. Rules and automata can be used to specify only the most basic behaviors, and feedback for learning is sparse and nonspecific. Intelligent behavior will therefore need to be discovered through interaction with the environment, often through coevolution with other agents. Neuroevolution, i.e. constructing neural network agents through evolutionary methods, has recently shown much promise in such learning tasks. Based on sparse feedback, complex behaviors can be discovered for single agents and for teams of agents, even in real time. In this talk I will review the recent advances in neuroevolution methods and their applications to various game domains such as othello, go, robotic soccer, car racing, and video games.

Challenges in Computer Go

Martin Müller
Department of Computing Science
University of Alberta
mmueller@cs.ualberta.ca

Abstract- Computer Go has been described as the "final frontier" of research in classical board games. The game is difficult for computers since no satisfactory evaluation function has been found yet. Go shares this aspect with many real-life decision making problems, and is therefore an ideal domain to study such difficult domains. This talk discusses the challenges of Computer Go on three levels: 1. incremental work that can be done to improve current Go programs, 2. strategies for the next decade, and 3. long term perspectives.

Opponent Modelling and Commercial Games

H.J. van den Herik, H.H.L.M. Donkers, P.H.M. Spronck
Department of Computer Science, Institute for Knowledge and Agent Technology, Universiteit Maastricht.
P.O.Box 616, 6200 MD, Maastricht, The Netherlands.
Email: herik,donkers,p.spronck@cs.unimaas.nl

Abstract- To play a game well a player needs to understand the game. To defeat an opponent, it may be sufficient to understand the opponent's weak spots and to be able to exploit them. In human practice, both elements (knowing the game and knowing the opponent) play an important role. This article focuses on opponent modelling independent of any game. So, the domain of interest is a collection of two-person games, multi-person games, and commercial games. The emphasis is on types and roles of opponent models, such as speculation, tutoring, training, and mimicking characters. Various implementations are given. Suggestions for learning the opponent models are described and their realization is illustrated by opponent models in game-tree search. We then transfer these techniques to commercial games. Here it is crucial for a successful opponent model that the changes of the opponent's reactions over time are adequately dealt with. This is done by dynamic scripting, an improvised online learning technique for games. Our conclusions are (1) that opponent modelling has a wealth of techniques that are waiting for implementation in actual commercial games, but (2) that the games' publishers are reluctant to incorporate these techniques since they have no definitive opinion on the successes of a program that is outclassing human beings in strength and creativity, and (3) that game AI has an entertainment factor that is too multifaceted to grasp in reasonable time.

1 Introduction

Ever since humans play games they desire to master the game played. Obviously, gauging the intricacies of a game completely is a difficult task; understanding some parts is most of the time the best a player can aim at. The latter means solving some sub-domains of a game. However, in a competitive game it may be sufficient to understand more of the game than the opponent does in order to win a combat. Remarkably, here a shift of attention may take place, since playing better than the opponent may happen (1) by the player's more extensive knowledge of the game or (2) by the player's knowledge of the oddities of the opponent. In human practice, a combination of (1) and (2) is part of the preparation of a top grandmaster in Chess, Checkers or Shogi. Opponent modelling is an intriguing part of a player's match preparation, since the preparing player tries to understand the preferences, strategies, skill, and weak spots of his[1] opponent.

In the following we distinguish between the player and the opponent if a two-person game is discussed. In multi-person games and in commercial games we will speak of agents. Opponent modelling is a research topic that was envisaged already a long time ago. For instance, in the 1970s chess programs incorporated a contempt factor, meaning that against a stronger opponent a draw was accepted even if the player was +0.5 ahead, and a draw was declined against a weaker opponent even when the player had a minus score. In the 1990s serious research in the domain of opponent modelling started [5, 19]. Nowadays opponent modelling also plays a part in multi-person games (collaboration, conspiracy, opposition) and in commercial games. Here we see a shift from opponent modelling towards subject modelling and even environmental entertainment modelling.

The course of the article is as follows. Section 2 defines types and roles of opponent models. In section 3 we provide a brief overview of the development of opponent models currently in use in Roshambo, the Iterated Prisoner's Dilemma, and Poker. We extrapolate the development to commercial Games. Section 4 lists six possible implementations of the opponent models. A main question is dealt with in section 5, viz. how to learn opponent models. We describe two methods, refer to a third one, and leave the others undiscussed. Section 6 focuses on the three implementations in game-tree search: OM search, PrOM search, and symmetric opponent modelling. Section 7 presents dynamic scripting as a technique for online adaptive game AI in commercial games. Finally section 8 contains our conclusions.

2 Roles of Opponent Models

In general, an opponent model is an abstracted description of a player or a player's behaviour in a game. There are many different types. For instance, a model can describe a player's preferences, his strategy, skill, capabilities, weaknesses, knowledge, and so on.

For each type we may distinguish two different roles in a game program. The first role is to model a (human or computer) opponent in such way that it informs the player appropriately in classical two-person games. Such opponent model can be *implicit* in the program's strategy or made *explicit* in some internal description. The task of such an opponent model is to understand and mimic the opponent's behaviour, in an attempt either to beat the opponent (see section 2.1) or to assist the opponent (section 2.2).

The second role is to provide an artificial opponent agent for the own agent (program or human player) using the program (see section 2.3), or an artificial agent that participates in an online multi-person game (section 2.4). Iteratively, such an opponent agent could bear in itself an opponent model of its own opponents. In most cases, the task of an

[1] In this article we use 'he' ('his') if both 'he' and 'she' are possible.

opponent model in this second role is to manifest an interesting and entertaining opponent to human players.

Regardless of its internal representation, an opponent model may range from statically defined in the program to dynamically adaptable. Opponent models that are dynamically adapted (or adapt themselves) to the opponent and other elements of the environment are to be preferred.

Below we will detail the four appearances in which opponent models are of use.

2.1 Speculation in heuristic search

The classical approach in Artificial Intelligence to board games, such as Chess, Checkers and Shogi, is heuristic search. It is based on the Minimax procedure for zero-sum perfect-information games as described by Von Neumann and Morgenstern [41]. However, the complexity of board games makes Minimax infeasible to be applied directly to the game tree. Therefore, the game tree is reduced in its depth by using a static heuristic evaluation, and quite frequently also in its breadth by using selective search. Moreover, during the detection of the best move to play next, much of the reduced game tree is disregarded by using $\alpha\beta$ pruning and other search enhancements. Actual game playing in this approach consists of solving a sequence of reduced games. Altogether, the classical approach has proven to be successful in Chess, Checkers, and a range of other board games.

In the classical approach, reasoning is based on defending against the worst case and attempting to achieve the best case. However, because heuristic search is used, it is not certain that the worst case and the best case are truly known. It means that it might be worthwhile to use additional knowledge during heuristic search in order to increase the chance to win, for instance, knowledge of the opponent. It is clear that humans use their knowledge of the opponent during game playing.

There are numerous ways in which knowledge of the (human) opponent can be used to improve play by heuristic search. One can use knowledge of the opponent's preferences or skills to force the game into positions that are considered to be less favourable to the opponent than to oneself. In the case that a player is facing a weak position, the player may try to speculate on positions in which the opponent is more likely to make mistakes. If available, a player may use the opponent's evaluation function to speculate (or even calculate) the next move an opponent will make and thus adopt its strategy to find the optimal countermoves. We will concentrate on the last approach in section 5.

2.2 Tutoring and Training

An opponent model can be used to assist the human player. We discuss two different usages: tutoring and training. Commercial board game programs (can) increase their attractiveness by offering such functionality.

In a tutoring system [20], the program can use the model of the human opponent to teach the player some aspects of the game in a personalized manner, depending on the type of knowledge present in the opponent model. If the model includes the player's general weaknesses or skills, it can be used to lead apprentices during a game to positions that help them to learn from mistakes. When the model includes the strategy or preferences of the player, then this knowledge can be employed to provide explicit feedback to the user during play, either by tricking the player into positions in which a certain mistake will be made and explicitly corrected by the program, or by providing verbal advice such as: "you should play less defensive in this stage of the game".

A quite different way to aid the apprentice is to provide preset opponent types. Many game programs offer an option to set the playing strength of the program. Often, this is arranged by limiting the resources (e.g., time, search depth) available to the program. Sometimes, the preferences of a program can be adjusted to allow a defensive or aggressive playing style. An explicit opponent model could assist even the experienced players to prepare themselves for a game against a specific opponent. In order to be useful, the program should in this case be able to learn a model of a specific player. In Chess, some programs (e.g., CHESS ASSISTANT[2]) offer the possibility to adjust the opening book to a given opponent, on the basis of previously stored game records.

2.3 Non-player Characters

The main goal in commercial computer games is not to play as strong as possible but to provide entertainment. Most commercial computer games, such as computer roleplaying games (CRPGs) and strategy games, situate the human player in a virtual world that is populated by computer-controlled agents, which are called "non-player characters" (NPCs). These agents may fulfil three roles: (i) as a companion, (ii) as an opponent, and (iii) as a neutral, background character. In the first two roles, an opponent model of the human player is needed. In practice, for most (if not all) commercial games this model is implemented in an implicit way. The third role, however commercially interesting, is not relevant in the subject area of opponent modelling, and thus it is not discussed below.

In the companion role, the agent must behave according to the expectations of the human player. For instance, when the human player prefers a stealthy approach to dealing with opponents agents, he will not be pleased when the computer-controlled companions immediately attack every opponent agent that is near. If the companions fail to predict with a high degree of success what the human player desires, they will annoy the human player, which is detrimental for the entertainment value of the game. Nowadays, companion agents in commercial games use an implicit model of the human player, which the human player can tune by setting a few parameters that control the behaviour of the companion (such as "only attack when I do too" or "only use ranged weapons").

In the opponent role, the agent must be able to match the

[2] See: http://store.convekta.com.

playing skills of the human player. If the opponent agent plays too weak a game against the human player, the human player loses interest in the game [34]. In contrast, if the opponent agent plays too strong a game against the human player, the human player gets stuck in the game and will quit playing too [25]. Nowadays, commercial games provide a 'difficulty setting' which the human player can use to set the physical attributes of opponent agents to an appropriate value (often even during gameplay). However, a difficulty setting does not resolve problems when the quality of the *tactics* employed by opponent agents is not appropriate for the skills of the human player.

The behaviour of opponent agents in commercial games is designed during game development, and does not change after the game has been released, i.e., it is static. The game developers use (perhaps unconsciously) a model of the human player, and a program behaviour for the opponent agents appropriate for this model. As a consequence, the model of the human player is implicit in the programmed agent behaviour. Since the agent behaviour is static, the model is static. In reality, of course, human players may be very different, and thus it is to be expected that for most games a static model is not ideal. A solution to this problem would be that the model, and thus the behaviour of the opponent agent, is dynamic. However, games' publishers are reluctant to release games where the behaviour of the opponent agents is dynamic, since they fear that the agents may learn undesirable behaviour after the game's release.

The result is that, in general, the behaviour of opponent agents is unsatisfying to human players. Human players prefer to play against their own kind, which is a partial explanation for the popularity of multi-person games [33].

2.4 Multi-person games

In multi-person games, opponent models can be used to provide NPCs as well. Clearly, the problem mentioned in the previous subsection is also present here, only in a much harder form for the opponent agents, since they have to deal with many human players with many different levels of skills in parallel.

Yet another role of opponent models comes into sight in multi-person games. There are situations in which a player is not able or willing to continue playing, but the character representing the player remains 'alive' inside the game. Such a situation could arise from (i) a connection interrupt in an online game, (ii) a 'real-world' interruption of the human player, or (iii) a human player wanting to enter multiple copies of himself in the game. An opponent model could be used in those instances to take over control of the human's alter-ego in the game, while mimicking the human player's behaviour. Of course, such a model should be adaptable to the player's characteristics.

3 Towards Commercial Games

Below we deal with three actual implementations of opponent models (3.1), viz. in Roshambo, the Iterated Prisoner's Dilemma, and Poker. From here we extrapolate the development to commercial games (3.2) with an emphasis on adaptive game AI.

3.1 Opponent models used Now

Many of the usages of opponent models as presented in the previous section are still subject of current and future research. However, in a number of games, adaptive opponent models are an essential part of successful approaches. It is especially the case in iterated games. These are mostly small games that are played a number of times in sequence; the goal is to win the most games on average. Two famous examples of iterated games are Roshambo (Rock-Paper-Scissors) and the Iterated Prisoner's Dilemma (IPD). Both games consist of one simultaneous move after which the score is determined. Roshambo has three options for each move and zero-sum scores, IPD has only two options, but has nonzero-sum scores. Both games are currently played by computers in tournaments.

In Roshambo, the optimal strategy in an infinitely repeated game is to play randomly. However, in an actual competition with a finite number of repetitions, a random player will end up in the middle of the pack and will not win the competition. Strong programs, such as IOCAINE POWDER [12] apply opponent modelling in order to predict the opponent's strategy, while at the same time they attempt to be as unpredictable as possible.

Although IPD seems not so different from Roshambo, the opponent model must take another element into account: the willingness of the opponent to cooperate. In IPD, the players receive the highest payoff if both players cooperate. Since the first IPD competition by Axelrod in 1979 [2], the simple strategy 'Tit-for-Tat' has won most of the competitions [23]. However, the 2004 competition was won by a team that used multiple entries and recognition codes to 'cheat'. Although this is not strictly opponent modelling, the incident caused the birth of a new IPD competition at CIG'05 in which multiple entries and recognition codes are allowed. IPD illustrates an aspect of opponent modelling that will play a role, in particular, in multi-person games, viz., how to measure the willingness to cooperate and how to tell friendly from hostile opponents?

A more complex iterated game is Poker. The game offers more moves than Roshambo and IPD, involves more players in one game and has imperfect information. However, the game does not need heuristic search to be played. Although many Poker-playing programs exist that do not use any opponent model, the strong and commercially available program POKI ([3]) is fully based on opponent-modelling. Schaeffer states: "No poker strategy is complete without a good opponent-modelling system. A strong poker player must develop a dynamically changing (adaptive) model of each opponent, to identify potential weaknesses." Opponent modelling is used with two distinct goals: to predict the next move of each opponent and to estimate the strength of each opponent's hand.

3.2 The Future is in Commercial Games

The answer to the question "Are adaptive opponent models really necessary?" is that adaptive opponent models are sorely needed to deal with the complexities of state-of-the-art commercial games.

Over the years commercial games have become increasingly complex, offering realistic worlds, a high degree of freedom, and a great variety of possibilities. The technique of choice used by game developers for dealing with a game's complexities is rule-based game AI, usually in the form of scripts [29, 40]. The advantage of the use of scripts is that scripts are (1) understandable, (2) predictable, (3) tuneable to specific circumstances, (4) easy to implement, (5) easily extendable, and (6) useable by non-programmers [40, 39]. However, as a consequence of game complexity, scripts tend to be quite long and complex [4]. Manually-developed complex scripts are likely to contain design flaws and programming mistakes [29].

Adaptive game AI changes the tactics used by the computer to match or exceed the playing skills of the particular human player it is pitted against, i.e., adaptive game AI changes its implicit model of the human player to be more successful. Adaptive game AI can ensure that the impact of the mistakes mentioned above is limited to only a few situations encountered by the player, after which their occurrence will have become unlikely. Consequently, it is safe to say that the more complex a game is, the greater the need for adaptive game AI [13, 24, 16]. In the near future game complexity will only increase. As long as the best approach to game AI is to design it manually, the need for adaptive game AI, and thus for opponent modelling, will increase accordingly.

4 How to Model Opponents

The internal representation of an opponent model depends on the type of knowledge that it should contain and the task that the opponent model should perform. Artificial Intelligence offers a range of techniques to build such models

4.1 Evaluation functions

In the context of heuristic search, an opponent model can concentrate on the player's preferences. These preferences are usually encoded in a static heuristic evaluation function that provides a score for every board position. An opponent model can consist of a specific evaluation function. The evaluation function can either be hand-built on the basis of explicit knowledge or machine-learned on basis of game records.

4.2 Neural networks

The preferences of an opponent can also be represented by a neural network or any other machine-learned function approximator. Such a network can be learned from game records or actual play. However, neural networks can also be used to represent other aspects of the opponent's behaviour. They could represent the difficulty of positions for a specific opponent [28], or the move ordering preferred. The Poker program POKI also uses neural networks to represent the opponent model.

4.3 Rule-based models

A rule-based model consists of a series of production rules, that couple actions to conditions. It is a reactive system, that tests environment features to generate a response. A rule-based model is easily implemented. It is also fairly easy to be maintained and analysed.

4.4 Finite-State Machine

A finite-state machine model consists of a collection of states, which represent situations in which the model can exist, with defined state transitions that allow the model to go into a new state. The state transitions are usually defined as conditions. The model's behaviour is defined separately for each state.

4.5 Probabilistic models

The finite-state machine model can be augmented by probabilistic transitions. It results in a probabilistic opponent model. This kind of model is especially useful in games with imperfect information, such as Poker, and most commercial games.

A second probabilistic opponent model consists of a mixture of other models (opponent types). In these models, the strategy of the opponent is determined by first generating a random number (which may be biased by certain events) and then on the basis of the outcome selecting one type out of a set of predefined opponent types.

4.6 Case-based models

A case-based model consists of a case base with samples of situations and actions. By querying the case base, the cases corresponding with the current situation are retrieved, and an appropriate action is selected by examining the actions belonging to the selected cases. An advantage of a case-based model, is that the model can be easily updated and expanded by allowing it to collect automatically new cases while being used.

5 Learning Opponent Models

A compelling question is: can a program learn an opponent model? Below we describe some research efforts made in this domain. They consist of learning evaluation functions (5.1), learning probabilistic models (5.2), and learning opponent behaviour (5.3).

5.1 Learning evaluation functions

There are two basic approaches to the learning of opponent models for heuristic search: (1) to learn an evaluation function, a move ordering, the search depth and other search preferences used by a given opponent, and (2) to learn the

opponent's strategy, which means to learn directly the move that the opponent selects at every position.

The first approach has been studied in computer chess, especially the learning of evaluation functions. Although the goal often is to obtain a good evaluation function for $\alpha\beta$search, similar techniques can be used for obtaining the evaluation function of an opponent type. For instance, Anantharaman [1] describes a method to learn or tune an evaluation function with the aid of a large set of positions and the moves selected at those positions by master-level human players. The core of the approach is to adapt weights in an evaluation function by using a linear discriminant method in such a way that a certain score of the evaluation function is maximized. The evaluation function is assumed to have the following form: $V(h) = \sum_i W_i\, C_i(h)$. The components $C_i(h)$ are kept constant, only the weights W_i are tuned. The method was used to tune an evaluation function for the program DEEP THOUGHT, a predecessor of DEEP BLUE. Although the method obtained a better function than the hand-tuned evaluation function of the program, the author admits that it is difficult to avoid local maxima. Fürnkranz [15] gives an overview of machine learning in computer chess, including several other methods to obtain evaluation functions from move databases.

5.2 Learning probabilistic models

The learning of opponent-type probabilities during a game is limited since the number of observations is low. It can, however, be useful to adapt probabilities that were achieved earlier, for instance by offline learning. Two types of online learning can be distinguished: a *fast* one in which only the best move of every opponent type is used, and a *slow* one in which the search value of all moves is computed for all opponent types.

Fast online learning happens straightforwardly as follows: start with the prior obtained probabilities. At every move of the opponent do the following: for all opponent types detect whether their best move is equal to the actually selected move. If so, reward that opponent type with a small increase of the probability. If not, punish the opponent type. The size of the reward or punishment should not be too large because this type of learning will lead to the false supremacy of one of the opponent types. This type of incremental learning is also applied in the prediction of user actions [8].

Slow online learning would be an application of the naive Bayesian learner (see [9]). A similar approach is used in learning probabilistic user profiles [30]. Slow online learning works as follows. For all opponent types ω_i, the sub-game values $v_{\omega_i}(h + m_j)$ of all possible moves m_j at position h are computed. These values are transformed into conditional probabilities $\Pr(m_j|\omega_i)$, that indicate the "willingness" of the opponent type to select that move. This transformation can be done in a number of ways. An example is the method by Reibman and Ballard [31]: first determine the rank $r(m_j)$ of the moves according to $v_{\omega_i}(h+m_j)$ and then assign probabilities:

$$\Pr(m_j|\omega_i) = \frac{(1-P_s)^{r(m_j)-1} \cdot P_s}{\sum_k (1-P_s)^{r(m_k)-1} \cdot P_s} \qquad (1)$$

P_s ($\in (0,1]$) can be interpreted as the likeliness of the opponent type not to deviate from the best move: the higher P_s, the higher the probability on the best move. It is however also possible to use the actual values of $v_{\omega_i}(h + m_j)$. Now Bayes' rule is used to compute the opponent-type probabilities given the observed move of the opponent.

$$\Pr(\omega_i|m_\Omega(h)) = \frac{\Pr(m_\Omega(h)|\omega_i)\Pr(\omega_i)_t}{\sum_k \Pr(m_\Omega(h)|\omega_k)\Pr(\omega_k)_t} \qquad (2)$$

These a-posteriori probabilities are used to update the opponent-type probabilities.

$$\Pr(\omega_i)_{t+1} = (1-\gamma)\Pr(\omega_i)_t + \gamma \Pr(\omega_i|m_\Omega(h)) \quad (3)$$

In this formula, parameter γ ($\in [0,1]$) is the learning factor: the higher γ, the more influence the observations have on the opponent-type probabilities. The approach is called *naive* Bayesian learning, because the last formula assumes that the observations at the subsequent positions in the game are independent.

5.3 Learning opponent behaviour

Direct learning of opponent strategies is studied extensively on iterated games [14]. For learning opponent strategies in Roshambo we refer to Egnor [12]. General learning in repeated games is studied, for example, by Carmel and Markovitch [7].

6 Opponent Models in Game-Tree Search

Junghanns [22] gave an overview of eight problematic issues when using $\alpha\beta$ in game-tree search. He also listed alternative algorithms that aimed at overcoming one or more of these problems. The four most prominent problems with $\alpha\beta$ are: (1) the *heuristic-error* problem (heuristic values are used instead of real game values), (2) the *scalar-value* problem (only a single scalar value is used to express the value of an arbitrarily complex game position), (3) the *value-backup* problem (lines leading to several good positions are preferable to a line that leads to a single good position), and (4) the *opponent* problem (knowledge of the opponent is not taken into account).

The first attempt to use rudimentary knowledge of the opponent in heuristic search is the approach by Slagle and Dixon [35] in 1970. At the base of their M & N-search method lies the observation that it is wise to favour positions in which there are several moves with good values over positions in which there is only one move with a good value. In 1983, Reibman and Ballard [31] assume that the opponent sometimes is fallible: there is a chance in each position that the opponent selects a non-rational move. In their model, the probability that the opponent selects a specific move depends on the value of that move and on the degree of fallibility of the opponent.

Below we will discuss three further approaches of dealing with Junghanns's fourth problem; viz. Opponent-Model (OM) search, Probabilistic OM (PrOM) search, and symmetric opponent modelling.

6.1 OM search

The main assumption of OM search is that the opponent (called MIN) uses a Minimax algorithm (or equivalent) with an evaluation function that is known to the first player (called MAX). Also the depth of the opponent's search tree and the opponent's move order are assumed to be known. This knowledge is used to construct a derivative of Minimax in which MAX maximizes at max nodes, but selects at min nodes the moves that MIN will select, according to MAX' knowledge of MIN's evaluation function.

For a search tree with even branching factor w and fixed depth d, OM search needs $n = w^{\lceil d/2 \rceil}$ evaluations for MAX to determine the search-tree value: at every min node, only one max child has to be investigated but at every max node, all w children must be investigated. Because the search-tree value of OM search is defined as the maximum over all these n values for MAX, none of these values can be missed. This means that the efficiency of OM search depends on how efficient the values for MIN can be obtained.

A straightforward and efficient way to implement OM search is by applying $\alpha\beta$ *probing*: at a min node perform $\alpha\beta$ search with the opponent's evaluation function (the *probe*), and perform OM search with the move that $\alpha\beta$ search returns; at a max node, maximize over all child nodes. The probes can in fact be implemented using any enhanced minimax search algorithm available, such as MTD(f). Because a separate probe is performed for every min node, many nodes are visited during multiple probes. (For example, every min node P_j on the principal variation of a node P will be probed at least twice.) Therefore, the use of transposition tables leads to a major reduction of the search tree. The search method can be improved further by a mechanism called β-pruning (see Figure 1).

The assumptions that form the basis of OM search give rise to two types of risk. The first type of risk is caused by a player's imperfect knowledge of the opponent. When MIN uses an evaluation function different from the one assumed by MAX (or uses a different search depth or even a different move ordering), MIN might select another move than the move that MAX expects. This type of risk has been described in detail and thoroughly analyzed in [18, 21]. The second type of risk arises when the *quality* of the evaluation functions used is too low. The main risk appears to occur when the MAX player's evaluation function overestimates a position that is selected by MIN. This position may then act as an attractor for many variations, resulting in a bad performance. To protect the OM search against such a performance the notion of *admissible* pairs of evaluation functions is needed: (1) MAX's function is a better profitability estimator than MIN's, and (2) MAX's function never overestimates a position that MIN's does not overestimate likewise [11].

```
OmSearchBPb(h, β)
1    if (h ∈ E) return (V_0(h), null)
2    if (p(h) = MAX)
3        L ← m(h) ; m ← firstMove(L)
4        m* ← m ; v*_0 ← −∞
5        while (m ≠ null)
6            (v_0, mm) ← OmSearchBPb(h + m, β)
7            if (v_0 > v*_0) v*_0 ← v_0 ; m* ← m
8            m ← nextMove(L)
9    if (p(h) = MIN)
10       (v*_op, m*) ← αβ-Search(h, −∞, β, V_op(·))
11       (v*_0, mm) ← OmSearchBPb(h + m*, v*_op + 1)
12   return (v*_0, m*)
```

Figure 1: β-pruning OM search with $\alpha\beta$ probing.

6.2 PrOM search

In contrast to OM search that assumes a fixed evaluation function of the opponent, PrOM search [10] uses a model of the opponent that includes uncertainty. The model consists of a set of evaluation functions, called *opponent types*, together with a probability distribution over these functions. More precisely, PrOM search is based on the following four assumptions:

(1) MAX has knowledge of n different *opponent types* $\omega_0 \ldots \omega_{n-1}$. Each opponent type ω_i is a minimax player that is characterized by an evaluation function V_{ω_i}. MAX is using evaluation function V_0. For convenience, one opponent type (ω_0) is assumed to use the same evaluation function as MAX uses ($V_{\omega_0} \equiv V_0$).

(2) All opponent types are assumed to use the same search-tree depth and the same move ordering as MAX.

(3) MAX has subjective probabilities $\Pr(\omega_i)$ on the range of opponents, such that $\sum_i \Pr(\omega_i) = 1$.

(4) MIN is using a *mixed strategy* which consists of the n opponent-type minimax strategies. At every move node, MIN is supposed to pick randomly one strategy according to the opponent-type probabilities $\Pr(\omega_i)$.

The fourth assumption is a crucial one because it determines the semantics of the opponent model: the mixed strategy acts as an approximation of opponent's real strategy. The subjective probability of every opponent type acts as the amount of MAX's belief that this opponent type resembles the opponent's real behaviour.

The applicability of $\alpha\beta$ probing in PrOM search is clear (see Figure 2). The values of $v_{\omega_i}(P)$ and the best move P_j for opponent type ω_i at min node P, can safely be obtained by performing $\alpha\beta$ search at node P, using evaluation function $V_{\omega_i}(\cdot)$. Notice that an $\alpha\beta$ probe has to be performed for every opponent type separately. These $\alpha\beta$ probes can be improved by a number of search enhancements. If transposition tables are used, then a separate table is needed per opponent type. The transposition table for an opponent type must not be cleared at the beginning of each probe, but only

```
PromSearchBPb(h, β̄)
1    if (h ∈ E) return (V_0(h), null)
2    if p(h) = MAX
3        L ← m(h) ; m ← firstMove(L) ; m*_0 ← m
4        v*_0 ← −∞
5        while (m ≠ null)
6            (v_0, mm) ← PromSearchBPb(h + m, β̄)
7            if (v_0 > v*_0) v*_0 ← v_0 ; m*_0 ← m
8            m ← nextMove(L)
9    if p(h) = MIN
10       L ← ∅
11       for i ∈ {0, ..., n − 1}
12           (v̄*_i, m̄*_i) ← αβ-Search(h, −∞, β̄_i, V_i(·))
13           L ← L ∪ {m̄*_i}
14       v*_0 ← 0; m*_0 ← null ; m ← firstMove(L)
15       while (m ≠ null)
16           for i ∈ {0, ..., n − 1}
17               if (m = m̄*_i) β̄_i ← v̄*_i + 1 else β̄_i ← ∞
18           (v_0, mm) ← PromSearchBPb(h + m, β̄)
19           for i ∈ {0, ..., n − 1}
20               if (m = m̄*_i) v*_0 ← v*_0 + Pr(ω_i) v_0
21           m ← nextMove(L)
22   return (v*_0, m*_0)
```

Figure 2: β-pruning PrOM search with $\alpha\beta$ probing.

at the start of the PrOM search so that knowledge of the search tree is shared between the subsequent probes for the same opponent type.

Because of the usage of multiple opponent models, the computational efforts for PrOM search are larger than those needed for OM search. However, the risk while using PrOM search is lower than while using OM search, when MAX uses the own evaluation functions as one of the opponent types. Experimental results indicate that when computational efforts are disregarded, PrOM search performs better than OM search with the same amount of knowledge of the opponent and with the same search depth.

6.3 Symmetric Opponent Modelling

Instead of the asymmetric opponent models in OM search and PrOM search, it might be more natural to assume that both players use an opponent model of each other of which they are mutually aware. In the context of heuristic search it means that both players agree that they have different (i.e., non-opposite) evaluation values for positions. The key concept is common interest. Evaluation values are based on many factors of a position. Some of these factors are pure competitive, such as the number of black pieces on a chess board, other factors are of interest of both players. Carmel and Markovitch [6] give an example for the game of checkers. Another example is the degree to which a chess position is 'open' or 'closed'. An open position (in which many pieces can move freely) is favoured by many players over closed positions. Therefore, the openness of a position is a common interest of both players.

Assume that the competitive factors of a position count S and the common-interest factors count C, then the value for the first player would be $C+S$. In the standard zero-sum approach, the opponent would be assumed to use the value $-(C+S)$ for the same position, which would mean that the opponent would award the common interest of the position with $-C$. However, it seems more natural that the second player uses the value $C - S$ for the position. In the model of Carmel and Markovitch [6], only one of the players is assumed to be aware of this fact. However, why should we not assume knowledge symmetry and let both players agree on the size of C and S? When the two players receive different pay-offs (e.g., $C + S$ and $C - S$) and these pay-offs are common knowledge, we achieve a nonzero-sum game of perfect information. In such a game there is both opponent modelling and knowledge symmetry, leading to symmetric opponent modelling. It should be noted that in any nonzero-sum game, it is possible to describe the pay-offs in terms of competitive and common-interest factors. If the first player receives A and the second player B, the common interest C is equal to $(A + B)/2$ and the competitive part S is equal to $(A - B)/2$.

The use of a nonzero-sum game as a means of symmetric opponent model introduces two challenges: (1) how to select the best equilibrium and (2) how to search efficiently. In contrast to zero-sum games in which all equilibria have the same value, in nonzero-sum games equilibria can co-exist with different values. Although all equilibria of a nonzero-sum game of perfect information can be found easily by backward induction (similar to Minimax, see Figure 3), the selection of the best one among them is hard. Moreover, the basic backward induction procedure is not feasible for large game trees, so an $\alpha\beta$-like pruning mechanism and other enhancements are asked for.

```
BackInd(h)
1    if (h ∈ E) return (V_1(h), V_2(h), null)
2    v* ← −∞, L ← ∅
3    for m ∈ m(h)
4        (·, v_1, v_2) ← BackInd(h + m)
5        if (v_{p(h)} > v*)   L ← {(m, v_1, v_2)}
6                v* ← v_{p(h)}
7        if (v_{p(h)} = v*)   L ← L ∪ {(m, v_1, v_2)}
8    select (m, v_1, v_2) ∈ L
9    return (m, v_1, v_2)
```

Figure 3: Backward Induction.

Both tasks can be helped by restricting ourselves to games with bounded common interest. These are nonzero-sum games where the value of C is bounded to an interval $[-B/2, B/2]$ around zero and where B is (much) smaller than the largest absolute value of S in any pay-off. The profit of using this bound is that it allows for pruning during game-tree search since the difference between the value for player 1 and 2 in each equilibrium is restricted to B. Moreover, the range of values of those equilibria is restricted, as we will show below. We will call this types of games: *BCI*

games (Bounded Common Interest games). It can be proven that the bound B on the common interest puts a bound on the values that the equilibria can take. For trees of depth d, the range is $v^* \pm B(d-1)$ for Player 1 and $-v^* \pm Bd$ for Player 2. These ranges indicate the 'damage' that has to be feared when selecting a suboptimal equilibrium. The ranges can also be used to rule out moves that cannot lead to any equilibrium.

The bound on common interest, B, also allows for pruning in an $\alpha\beta$-like manner. This pruning is based on the fact that in case of bounded common interest, the difference between the values for Player 1 and Player 2 is also bounded at any position in the tree. So, the value for one player can be used to predict the value for the other player, and bounds on the value for one player can be used to bound the value for the other player. In this way, shallow and deep pruning is possible, but the amount of pruning depends on the value of B and on the depth of the tree. With every additional level of depth, the bounds on the values are widened by B, leading to less and less pruning.

Two-player nonzero-sum games of perfect information can be used for symmetric opponent modelling. A fundamental difference with the standard zero-sum games is that several equilibria can exist in one game and that selecting a good equilibrium is very hard. We proved that when bounded common interest is assumed, the range of values that equilibria can take on is also bounded. Furthermore, BCI games allow pruning during the determination of the equilibria in a game tree. BCI games offer an alternative to Minimax-based algorithms and to Opponent-Model Search in heuristic search, but experimental evidence has to be collected on the practical usability and effectiveness of the approach. The BCI game also offers an opportunity to apply a range of search techniques from Artificial Intelligence to a class of games that is of interest to a broader audience than the traditional one.

7 Opponent Models with Dynamic Scripting

In this section we present dynamic scripting as a technique that is designed for the implementation of online adaptive game AI in commercial games. Dynamic scripting uses a probabilistic search to update an implicit opponent model of a human player, to be able to generate game AI that is appropriate for the player. Those interested in a more detailed exposition of dynamic scripting are referred to [37].

Dynamic scripting is an unsupervised online learning technique for games. It maintains several rulebases, one for each class of computer-controlled agents in the game. The rules in the rulebases are manually designed using domain-specific knowledge. Every time a new agent of a particular class is generated, the rules that comprise the script controlling the agent are extracted from the corresponding rulebase. The probability that a rule is selected for a script is proportional to the weight value that is associated with the rule. The rulebase adapts by changing the weight values to reflect the success or failure rate of the associated rules in scripts. A priority mechanism can be used to let certain rules take precedence over other rules. Dynamic scripting has been demonstrated to be fast, effective, robust, and efficient. The dynamic scripting process is illustrated in Figure 4 in the context of a game.

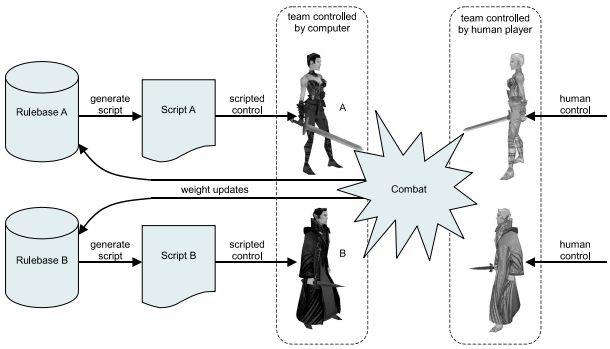


Figure 4: Dynamic scripting.

The learning mechanism in the dynamic-scripting technique is inspired by reinforcement learning techniques [38, 32]. 'Regular' reinforcement learning techniques, such as TD-learning, in general need large amounts of trials, and so are usually not sufficiently efficient to be used in games [27, 26]. Reinforcement learning is suitable to be applied to games if the trials occur in a short timespan (as in the work by [17], where fight movements in a fighting game are learned). However, for the learning of complete tactics, such as scripts, a trial consists of observing the performance of a tactic over a fairly long period of time. Therefore, for the online learning of tactics in a game, reinforcement learning will take too long to be particularly suitable. In contrast, dynamic scripting has been designed to learn from a few trails only.

In the dynamic-scripting approach, learning proceeds as follows. Upon completion of an encounter (i.e., a fight), the weights of the rules employed during the encounter are adapted depending on their contribution to the outcome. Rules that lead to success are rewarded with a weight increase, whereas rules that lead to failure are punished with a weight decrease. The remaining rules are updated so that the total of all weights in the rulebase remains unchanged.

Weight values are bounded by a range $[W_{min}, W_{max}]$. The size of the weight change depends on how well, or how badly, a computer-controlled agent behaved during an encounter with the human player. It is determined by a fitness function that rates an agent's performance as a number in the range $[0, 1]$. The fitness function is composed of four indicators of playing strength, namely (1) whether the team to which the agent belongs won or lost, (2) whether the agent died or survived, (3) the agent's remaining health, and (4) the amount of damage done to the agent's enemies. The new weight value is calculated as $W + \triangle W$, where W is the original weight value, and the weight adjustment $\triangle W$ is expressed by the following formula:

$$\triangle W = \begin{cases} -\lfloor P_{max} \dfrac{b - F}{b} \rfloor & \{F < b\} \\ \lfloor R_{max} \dfrac{F - b}{1 - b} \rfloor & \{F \geq b\} \end{cases} \quad (4)$$

In equation 4, $R_{max} \in \mathbb{N}$ and $P_{max} \in \mathbb{N}$ are the maximum reward and maximum penalty respectively, F is the agent fitness, and $b \in \langle 0, 1 \rangle$ is the break-even value. At the break-even point the weights remain unchanged.

In its pure form, dynamic scripting does not try to match the human player's skill, but tries to play as strongly as possible against the human player. That, however, is in conflict with the goal of commercial games, namely providing entertainment.

A variation on dynamic scripting allows it to adapt to meet the level of skill of the human player. This variation uses a fitness scaling technique that ensures that the game AI enforces an 'even game', i.e., a game where the chance to win is equal for both players. The domain knowledge stored in the rulebases used by dynamic scripting has been designed to generate effective behaviour at all times. Therefore, even when enhanced with a fitness-scaling technique, against a mediocre player dynamic scripting does not exhibit stupid behaviour interchanged with smart behaviour to enforce an even game, but it exhibits mediocre behaviour at all times.

We called the difficulty-scaling technique that was the most successful 'top culling'. Top culling works as follows.

In dynamic scripting, during the weight updates, the maximum weight value W_{max} determines the maximum level of optimisation that a learned strategy can achieve. A high value for W_{max} allows the weights to grow to large values, so that after a while the most effective rules will almost always be selected. This will result in scripts that are close to a presumed optimum. With top culling activated, weights are allowed to grow beyond the value of W_{max}. However, rules with weights higher than W_{max} will be excluded from the script generation process. If the value of W_{max} is low, effective rules will be quickly excluded from scripts, and the behaviour exhibited by the agent will be inferior (though not ineffective).

To determine the value of W_{max} that is needed to generate behaviour at exactly the level of skill of the human player, top culling automatically changes the value of W_{max}, with the intent to enforce an even game. It aims at having a low value for W_{max} when the computer wins often, and a high value for W_{max} when the computer loses often. The implementation is as follows. After the computer has won a fight, W_{max} is decreased by W_{dec} per cent (with a lower limit equal to the initial weight value W_{init}). After the computer has lost a fight, W_{max} is increased by W_{inc} per cent.

To evaluate the effect of top culling to dynamic scripting, we employed a simulation of an encounter of two teams in a complex computer roleplaying game, closely resembling the popular BALDUR'S GATE games. We used this environment in earlier research to demonstrate the efficiency of dynamic scripting [37]. Our evaluation experiments aimed at assessing the performance of a team controlled by the dynamic scripting technique using top culling, against a team controlled by static scripts. In the simulation, we pitted the dynamic team against a static team that uses one of five, manually designed, basic strategies (named 'offensive', 'disabling', 'cursing', 'defensive', and 'novice'), or one of three composite strategies (named 'random team', 'random agent' and 'consecutive').

Of the eight static team's strategies the most interesting in the present context is the 'novice' strategy. This strategy resembles the playing style of a novice BALDUR'S GATE player. While the 'novice' strategy normally will not be defeated by arbitrarily picking rules from the rulebase, many different strategies exist that can be employed to defeat it, which the dynamic team will quickly discover. Without difficulty-scaling, the dynamic team's number of wins will greatly exceed its losses. Details of the experiment are found in [36].

For each of the static strategies, we ran 100 tests without top culling, and 100 tests with top culling. We recorded the number of wins of the dynamic team for the last 100 encounters. Histograms for the tests with the 'novice' strategy are displayed in Figure 5. From the histogram it is immediately clear that top culling ensures that dynamic scripting plays an even game (the number of wins of the dynamic player is close to 50 out of 100), with a very low variance. The same pattern was observed against all the other investigated tactics. We can therefore conclude that dynamic scripting, enhanced with top culling, is successful in automatically discovering a well-working implicit model of the human player. As a perk, this model will be automatically updated when the human player learns new behaviour.

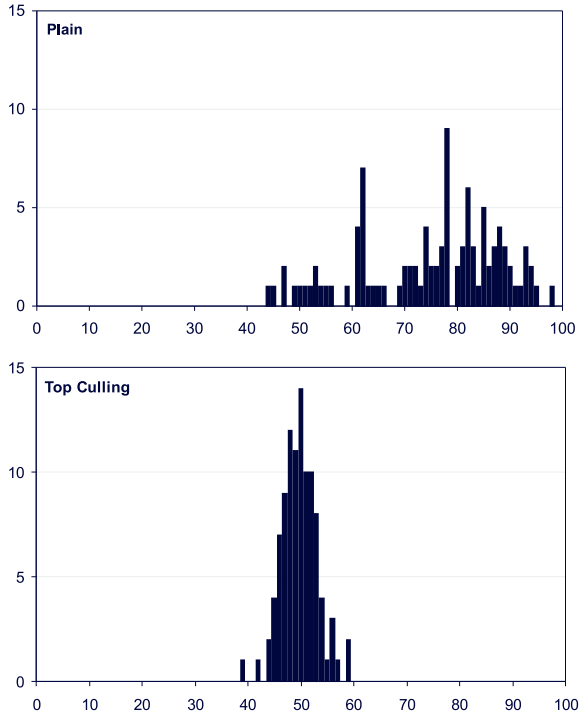


Figure 5: Histograms of 100 tests of the achieved number of wins in 100 fights, against the 'novice' strategy. The top graph is without difficulty scaling, the bottom graph with the application of top culling.

8 Conclusions

For human beings, opponent modelling is an essential and intriguing part of a player's match preparation. In this contribution we have discussed how opponent models can be implemented in computer programs. We investigated the full collection of games, ranging from classical two-person games via multi-person games to commercial games. Although opponent modelling is on the research table almost from the beginning of computer game research, serious implementation started in 1993 and the realization of most ideas is still in its infancy. There are three successful instances of actual implementation, viz. in Roshambo, the Iterated Prisoner's Dilemma, and Poker. Yet we may conclude that there is a wealth of techniques that are waiting for implementation in actual games.

In the contribution we have discussed OM search, PrOM search, and symmetric opponent modelling for classical games, and dynamic scripting for commercial games. In the last application (i.e., dynamic scripting and in particular in top culling) we see a shift in the goal to be reached. In classical games opponent modelling is used to raise the playing strength, in commercial games opponent modelling has as its main goal raising the entertainment factor. Currently, it is not clear to what extent both goals (i.e., raising the playing strength and raising the entertainment factor) are interchangeable. This is a topic of future research. However, at this moment it leads us to the conclusion that the undecided research question has as consequence that commercial games' publishers are reluctant to incorporate these techniques since they do not know whether a program that is outclassing human beings in strength and creativity will also raise the level of entertainment. From the research performed so far in this new area we may conclude that game AI (our current research tool for raising entertainment) has an entertainment factor that is too multifactored to grasp in reasonable time. Hence, new ideas should be developed that bring us a new classification of entertainment factors (types and roles) and will shed new light on the trade-off between issues on raising the playing strength and raising the entertainment.

Bibliography

[1] Anantharaman, T. (1997). Evaluation tuning for computer chess: Linear discriminant methods. *ICCA Journal*, Vol. 20, No. 4, pp. 224–242.

[2] Axelrod, R.M. (1984). *The Evolution of Cooperation.* BASIC Books, New York.

[3] Billings, D., Davidson, A., Schaeffer, J., and Szafron, S. (2000). The challenge of poker. *Artificial Intelligence*, Vol. 134, No. 1–2, pp. 201–240.

[4] Brockington, M. and Darrah, M. (2002). How *Not* to implement a basic scripting language. *AI Game Programming Wisdom* (ed. S. Rabin), pp. 548–554, Charles River Media, Inc., Hingham, MA.

[5] Carmel, D. and Markovitch, S. (1993). Learning models of opponent's strategies in game playing. *Proceedings AAAI Fall Symposion on Games: Planning and Learning*, pp. 140–147, Raleigh, NC.

[6] Carmel, D. and Markovitch, S. (1998). Pruning algorithms for multi-model adversary search. *Artificial Intelligence*, Vol. 99, No. 2, pp. 325–355.

[7] Carmel, D. and Markovitch, S. (1999). Exploration strategies for model-based learning. *Autonomous Agents and Multi-Agent Systems*, Vol. 2, No. 2, pp. 141–272.

[8] Davison, B.D. and Hirsh, H. (1998). Predicting sequences of user actions. *Predicting the Future: AI Approaches to Time-Series Problems*, pp. 5–12, AAAI Press, Madison, WI. Proceedings of AAAI-98/ICML-98 Workshop, published as Technical Report WS-98-07.

[9] Domingos, P. and Pazzani, M. (1997). On the optimality of the simple bayesian classifier under zero-one loss. *Machine Learning*, Vol. 29, pp. 103–130.

[10] Donkers, H.H.L.M., Uiterwijk, J.W.H.M., and Herik, H.J. van den (2001). Probabilistic opponent-model search. *Information Sciences*, Vol. 135, No. 3–4, pp. 123–149.

[11] Donkers, H.H.L.M., Uiterwijk, J.W.H.M., and Herik, H.J. van den (2003). Admissibility in opponent-model search. *Information Sciences*, Vol. 154, No. 3-4, pp. 119–140.

[12] Egnor, D. (2000). Iocaine powder. *ICGA Journal*, Vol. 23, No. 1, pp. 33–35.

[13] Fairclough, C., Fagan, M., MacNamee, B., and Cunningham, P. (2001). Research directions for AI in computer games. *12th Irish Conference on Artificial Intelligence & Cognitive Science (AICS 2001)* (ed. D. O'Donoghue), pp. 333–344.

[14] Fudenberg, D. and Levine, D.K. (1998). *The Theory of Learning in Games.* MIT Press, Cambridge, MA.

[15] Fürnkranz, J. (1996). Machine learning in computer chess: The next generation. *ICCA Journal*, Vol. 19, No. 3, pp. 147–161.

[16] Fyfe, C. (2004). Independent component analysis against camouflage. *Computer Games: Artificial Intelligence, Design and Education (CGAIDE 2004)* (eds. Q. Mehdi, N.E. Gough, S. Natkin, and D. Al-Dabass), pp. 259–262, University of Wolverhampton, Wolverhampton, UK.

[17] Graepel, T., Herbrich, R., and Gold, J. (2004). Learning to fight. *Computer Games: Artificial Intelligence, Design and Education (CGAIDE 2004)* (eds. Q. Mehdi, N.E. Gough, S. Natkin, and

D. Al-Dabass), pp. 193–200, University of Wolverhampton, Wolverhampton, UK.

[18] Iida, H., Uiterwijk, J.W.H.M., and Herik, H.J. van den (1993a). Opponent-model search. Technical Report CS 93-03, Universiteit Maastricht, Maastricht, The Netherlands.

[19] Iida, H., Uiterwijk, J.W.H.M., Herik, H.J. van den, and Herschberg, I.S. (1993b). Potential applications of opponent-model search. Part 1: the domain of applicability. *ICCA Journal*, Vol. 16, No. 4, pp. 201–208.

[20] Iida, H., Handa, K-i, and Uiterwijk, J. (1995). Tutoring strategies in game-tree search. *ICCA Journal*, Vol. 18, No. 4, pp. 191–204.

[21] Iida, H., Kotani, I., Uiterwijk, J.W.H.M., and Herik, H.J. van den (1997). Gains and risks of om search. *Advances in Computer Chess 8* (eds. H.J. van den Herik and J.W.H.M. Uiterwijk), pp. 153–165, Universiteit Maastricht, Maastricht, The Netherlands.

[22] Junghanns, A. (1998). Are there practical alternatives to alpha-beta? *ICCA Journal*, Vol. 21, No. 1, pp. 14–32.

[23] Kendall, G. (2005). Iterated prisoner's dilemma competition. http://www.prisoners-dilemma.com/.

[24] Laird, J.E. and Lent, M. van (2001). Human-level's AI killer application: Interactive computer games. *Artificial Intelligence Magazine*, Vol. 22, No. 2, pp. 15–26.

[25] Livingstone, D. and Charles, D. (2004). Intelligent interfaces for digital games. *Proceedings of the AAAI-04 Workshop on Challenges in Game Artificial Intelligence* (eds. D. Fu, S. Henke, and J. Orkin), pp. 6–10, AAAI Press, Menlo Park, CA.

[26] Madeira, C., Corruble, V., Ramalho, G., and Ratitch, B. (2004). Bootstrapping the learning process for the semi-automated design of challenging game ai. *Proceedings of the AAAI-04 Workshop on Challenges in Game Artificial Intelligence* (eds. D. Fu, S. Henke, and J. Orkin), pp. 72–76, AAAI Press, Menlo Park, CA.

[27] Manslow, J. (2002). Learning and adaptation. *AI Game Programming Wisdom* (ed. S. Rabin), pp. 557–566, Charles River Media, Inc., Hingham, MA.

[28] Markovitch, S. (2003). Learning and exploiting relative weakness of opponent agents. NWO-SIKS Workshop on Opponent Models in Games, Maastricht, the Netherlands.

[29] Nareyek, A. (2002). Intelligent agents for computer games. *Computers and Games, Second International Conference, CG 2000* (eds. T.A. Marsland and I. Frank), Vol. 2063 of *Lecture Notes in Computer Science*, pp. 414–422, Springer-Verlag, Heidelberg, Germany.

[30] Pazzani, M. and Billsus, D. (1997). Learning and revising user profiles: The identification of interesting web sites. *Machine Learning*, Vol. 27, pp. 313–331.

[31] Reibman, A.L. and Ballard, B.W. (1983). Non-minimax search strategies for use against fallible opponents. *AAAI'83*, pp. 338–342, Morgan Kaufmann Publ., San Mateo, CA.

[32] Russell, S. and Norvig, P. (2003). *Artificial Intelligence: A Modern Approach*. Prentice Hall, Pearson Education, Upper Saddle River, NJ, second edition edition.

[33] Schaeffer, J. (2001). A gamut of games. *Artificial Intelligence Magazine*, Vol. 22, No. 3, pp. 29–46.

[34] Scott, B. (2002). The illusion of intelligence. *AI Game Programming Wisdom* (ed. S. Rabin), pp. 16–20, Charles River Media, Inc., Hingham, MA.

[35] Slagle, J.R. and Dixon, J.K. (1970). Experiments with the M & N tree-searching program. *Communications of the ACM*, Vol. 13, No. 3, pp. 147–154.

[36] Spronck, P.H.M., Sprinkhuizen-Kuyper, I.G., and Postma, E.O. (2004a). Difficulty scaling of game AI. *GAME-ON 2004 5th International Conference on Intelligent Games and Simulation* (eds. A. El Rhalibi and D. Van Welden), pp. 33–37, EUROSIS, Ghent, Belgium.

[37] Spronck, P.H.M., Sprinkhuizen-Kuyper, I.G., and Postma, E.O. (2004b). Online adaptation of game opponent AI with dynamic scripting. *International Journal of Intelligent Games and Simulation*, Vol. 3, No. 1, pp. 45–53.

[38] Sutton, R.S. and Barto, A.G. (1998). *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, MA.

[39] Tomlinson, S.L. (2003). Working at thinking about playing or a year in the life of a games AI programmer. *Proceedings of the 4th International Conference on Intelligent Games and Simulation (GAME-ON 2003)* (eds. Q. Mehdi, N. Gough, and S. Natkin), pp. 5–12, EUROSIS, Ghent, Belgium.

[40] Tozour, P. (2002). The perils of AI scripting. *AI Game Programming Wisdom* (ed. S. Rabin), pp. 541–547, Charles River Media, Inc., Hingham, MA.

[41] von Neumann, J. and Morgenstern, O. (1944). *Theory of Games and Economic Behavior*. Princeton University Press, Princeton, NJ.

Oral Presentations

Utile Coordination: Learning interdependencies among cooperative agents

Jelle R. Kok[§] jellekok@science.uva.nl
Pieter Jan 't Hoen[*] hoen@cwi.nl
Bram Bakker[§] bram@science.uva.nl
Nikos Vlassis[§] vlassis@science.uva.nl

[§] Informatics Institute, Faculty of Science, University of Amsterdam, The Netherlands
[*] CWI, Centre for Mathematics and Computer Science, Amsterdam, The Netherlands

Abstract-

We describe Utile Coordination, an algorithm that allows a multiagent system to learn where and how to coordinate. The method starts with uncoordinated learners and maintains statistics on expected returns. Coordination dependencies are dynamically added if the statistics indicate a statistically significant benefit. This results in a compact state representation because only necessary coordination is modeled. We apply our method within the framework of coordination graphs in which value rules represent the coordination dependencies between the agents for a specific context. The algorithm is first applied on a small illustrative problem, and next on a large predator-prey problem in which two predators have to capture a single prey.

1 Introduction

A multiagent system (MAS) consists of a group of interacting autonomous agents [Stone and Veloso, 2000, Vlassis, 2003]. Modeling a problem as a MAS can have several benefits with respect to scalability, robustness and reusability. Furthermore, some problems are inherently distributed and can only be tackled with multiple agents that observe and act from different locations simultaneously.

This paper is concerned with fully cooperative MASs in which multiple agents work on a common task and must learn to optimize a global performance measure. Examples are a team of soccer playing robots or a team of robots which together must build a house. One of the key problems in such systems is *coordination*: how to ensure that the individual decisions of the agents result in jointly optimal decisions for the group.

Reinforcement learning (RL) techniques have been successfully applied in many single-agent domains to learn the behavior of an agent [Sutton and Barto, 1998] . In principle, we can treat a MAS as a 'large' single agent and apply the same techniques by modeling all possible joint actions as single actions. However, the action space scales exponentially with the number of agents, rendering this approach infeasible for all but the simplest problems. Alternatively, we can let each agent learn its policy independently of the other agents, but then the transition and reward models depend on the policy of the other learning agents, which may result in suboptimal or oscillatory behavior.

Recent work (e.g., [Guestrin et al., 2002a, Kok and Vlassis, 2004]) addresses the intermediate case, where the agents coordinate only some of their actions. These 'coordination dependencies' are context-specific. It depends on the state whether an agent can act independently or has to coordinate with some of the other agents. This results in large savings in the state-action representation and as a consequence in the learning time. However, in that work the coordination dependencies had to be specified in advance.

This paper proposes a method to learn these dependencies automatically. Our approach is to start with independent learners and maintain statistics on expected returns based on the action(s) of the other agents. If the statistics indicate that it is beneficial to coordinate, a coordination dependency is added dynamically. This method is inspired by 'Utile Distinction' methods from single-agent RL [Chapman and Kaelbling, 1991, McCallum, 1997] that augment the state space when this distinction helps the agent predict reward. Hence, our method is called the Utile Coordination algorithm.

As in [Guestrin et al., 2002b, Kok and Vlassis, 2004], we use a coordination graph to represent the context-specific coordination dependencies of the agents compactly. Such a graph can be regarded as a sparse representation of the complete state-action space and allows for factored RL updates. Our method learns how to extend the initial coordination graph and represent the necessary coordination dependencies between the agents using derived statistical measures.

The outline of this paper is as follows. In section 2 we review the class of problems and solution methods that we take into consideration. In section 3 we describe the concept of a coordination graph which is used extensively in the remainder of the paper as our representation framework. In section 4 the specific contribution of this paper, the Utile Coordination method, is explained. Experiments are presented in section 5.1 and section 5.2 which illustrate our new method on respectively a small coordination problem and a much larger predator-prey problem. In this popular multiagent problem a number of predators have to coordinate their actions to capture a prey. We show that our method outperforms the non-coordinated individual learners and learns a policy comparable to the method that learns in the complete joint-action space. We conclude in section 6 with some general conclusions and future work.

2 Collaborative multiagent MDPs

In this section we discuss several multiagent RL methods using the *collaborative multiagent MDP* (CMMDP) framework [Guestrin, 2003], which extends the single agent Markov Decision Process (MDP) framework to multiple cooperating agents. Formally, a CMMDP is defined as a tuple $\langle n, S, \mathcal{A}, T, R \rangle$ where n is the number of agents, S is a fi-

nite set of world states, $\mathcal{A} = \times_{i=1}^{n} \mathcal{A}_i$ are all possible joint actions defined over the set of individual actions of agent i, $T : S \times \mathcal{A} \times S \rightarrow [0, 1]$ is the Markovian[1] transition function that describes the probability $p(s'|s, a)$ that the system will move from state s to s' after performing the joint action $a \in \mathcal{A}$, and $R_i : S \times \mathcal{A} \rightarrow \mathbb{R}$ is the reward function that returns the reward $R_i(s, a)$ for agent i after the joint action a is taken in state s. A policy is defined as a mapping $\pi : S \rightarrow \mathcal{A}$. The objective is to find an optimal policy π^* that maximizes the expected discounted future cumulative reward, or expected return

$$Q^*(s, a) = \max_{\pi} Q^{\pi}(s, a) = \max_{\pi} E\left[\sum_{t=0}^{\infty} \gamma^t R(s_t, \pi(s_t))|\pi, s_0 = s, a_0 = a\right] \quad (1)$$

for each state s. The expectation operator $E[\cdot]$ averages over reward and stochastic transitions and $\gamma \in [0, 1)$ is the discount factor. Note that the agents try to maximize global returns based on global expected reward $R(s, a) = \sum_{i=1}^{n} R_i(s, a)$ which is the sum of all individual rewards. This is in contrast with stochastic games [Shapley, 1953] where each agent tries to maximize its *own* payoff. If this framework is constrained such that each agent receives the same reward, it corresponds exactly to the MMDP (multiagent MDP) framework of [Boutilier, 1996].

Fig. 1 depicts a small example problem of a collaborative multiagent MDP (and MMDP) with two agents and seven states. In each state every agent selects an individual action from the action set $\mathcal{A}_1 = \mathcal{A}_2 = \{c, d, e\}$, and based on the resulting joint action the agents move to a new state. The next state (and the subsequent reward) depends on the *joint* action only in state s_0. When either of the agents chooses action d, they move to s_1, and after any of the possible joint actions (indicated by $(*, *)$) both agents receive a reward of 0.5 in state s_4. For joint action (e, e) the agents will eventually receive a reward of 3, while for the remaining three joint actions in s_0, the agents will receive a large negative reward of -15. It is difficult for the agents to learn to reach the state s_5 if they learn individually. We discuss this further in section 5.1.

Reinforcement learning (RL) [Sutton and Barto, 1998] can be applied to learn the optimal policy in MDPs. In this paper we consider the case where the transition and reward model are not available, but an agent observes the complete state information. We focus on Q-learning, a well-known learning method for this setting. Q-learning starts with an initial estimate $Q(s, a)$ of the expected discounted future reward for each state-action pair. When an action a is taken in state s, reward r is received and next state s' is observed, the corresponding Q-value is updated by

$$Q(s, a) := Q(s, a) + \alpha[R(s, a) + \gamma \max_{a'} Q(s', a') - Q(s, a)] \quad (2)$$

[1]The Markov property implies that the state at time t provides a complete description of the history before time t.

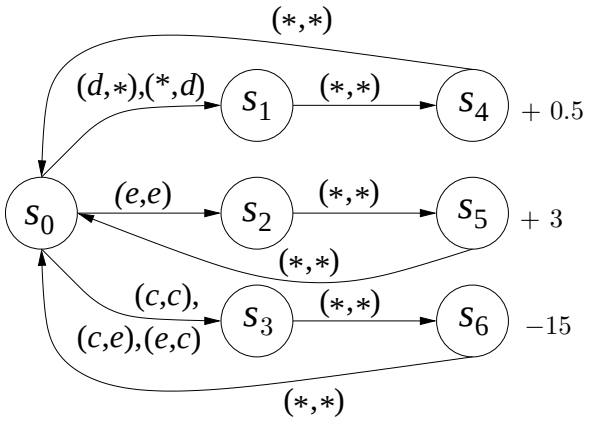


Figure 1: Simple coordination problem with seven states. Only in state s_0 does the joint action has an influence on the next state. The digits on the right represent the given reward to the agents in the corresponding state.

where $\alpha \in (0, 1)$ is an appropriate learning rate. Q-learning converges to the optimal Q-function $Q^*(s, a)$ when all state-action pairs are visited infinitely often by means of an appropriate exploration strategy. One of the most common strategies is $\epsilon-$greedy exploration in which at every step the greedy action $a^* = \arg\max_a Q(s, a)$ is selected with probability $1 - \epsilon$ and a (random) non-greedy action is selected with probability ϵ. In the above description of RL for MDPs, we assumed a tabular representation of the Q-table in which all state-action pairs are explicitly enumerated. Next, we will discuss three methods to apply RL to a CMMDP, which has multiple agents and joint actions.

At one extreme, we can represent the system as one large agent in which each joint action is modeled as a single action, and then apply single agent Q-learning. In order to apply such a *joint action MDP* (JAMDP) learner a central controller represents the complete JAMDP Q-function and informs each agent of its individual action, or all agents represent the complete Q-function separately, and execute their own individual action[2]. This approach leads to the optimal policy, but is infeasible for large problems since the joint action space, which is exponential in the number of individual actions, becomes intractable both in terms of storage, as well as in terms of exploration[3]. In the example of Fig. 1, this approach stores a Q-value for each of the nine joint actions in state s_i.

At the other extreme, we have *independent learners* (IL) [Claus and Boutilier, 1998] who ignore the actions and rewards of the other agents and learn their policies independently. This results in a large reduction in the state-action representation. However, the standard convergence proof

[2]The problem of determining the joint (possible exploration) action can be solved by assuming that all agents are using the same random number generator and the same seed, and that these facts are common knowledge among the agents [Vlassis, 2003].

[3]Note that function approximations techniques can also be used to deal with large state-action spaces. However, they are more often applied to large state spaces, instead of large action spaces because of the difficulty of generalizing over different actions.

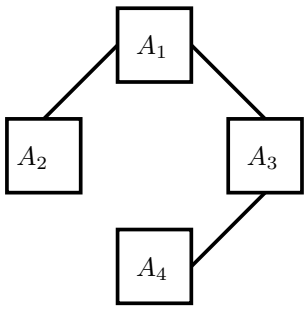


Figure 2: An example coordination graph for a 4-agent problem. Each node represents an edge, while the edges define the coordination dependencies.

for single agent Q-learning does not hold in this case, since the transition model for each agent depends on the unknown policy of the other learning agents. This can result in oscillatory behavior or convergence to a suboptimal policy. As we will see in section 5.1, independent learners converge to the suboptimal policy (d, d) for state s_0 in the example problem of Fig. 1, since the penalty for incorrect coordination has a large negative influence on the individual Q-values for actions c and e.

The next section is devoted to an intermediate approach, introduced in [Kok and Vlassis, 2004], in which the agents only coordinate their actions in certain predefined states. In section 4 we extend this method to learn the states in which coordination is needed.

3 Coordination Graphs

In this section, we will describe context-specific coordination graphs (CGs) [Guestrin et al., 2002b] which can be used to specify the coordination dependencies for subsets of agents. In a CG each node represents an agent, while an edge defines an action dependency between two agents. For example, the graph in Fig. 2 shows a CG for a 4-agent problem in which agent A_3 and A_4 have to coordinate, and A_1 has to coordinate with both A_2 and A_3. Since only connected agents have to coordinate their actions, the global coordination problem is decomposed into a number of local problems. The dependencies between the agents are specified using *value rules* of the form $\langle \rho; c : v \rangle$, where c is defined as the context (defined over possible state-action combinations) and the payoff $\rho(c) = v$ is a (local) contribution to the global payoff. This is a much richer representation than the IL or JAMDP variants, since it allows us to represent all possible dependencies between the agents in a context-specific manner. In 'coordinated' states, where actions of the agents depend on each other, the rules are based on joint actions, while for 'uncoordinated' states they are based on the individual actions of an agent. In our running example of Fig. 1, value rules for all possible joint actions, e.g., $\langle \rho; s_0 \wedge a_1 = e \wedge a_2 = e : v \rangle$ are needed in state s_0, while value rules based on individual actions, e.g., $\langle \rho; s_1 \wedge a_1 = a : v \rangle$, are a rich enough representation for all other states. The rules in a CG can be regarded as a sparse

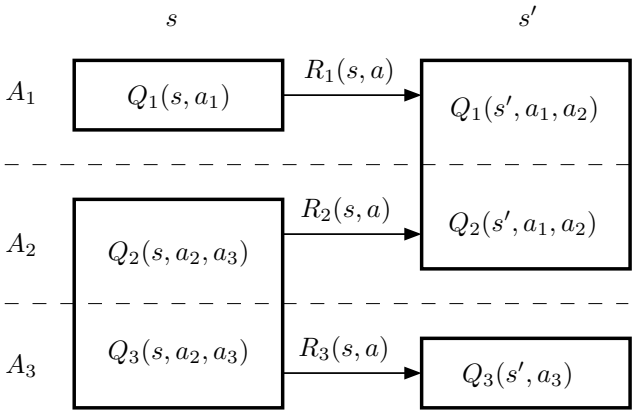


Figure 3: Example representation of the Q components of three agents for a transition from state s to state s'. In state s agent 2 and 3 have to coordinate their actions, while in state s' agent 1 and 2 have to coordinate their actions.

representation of the complete state-action space since they are defined over subsets of all state and action variables.

In order to compute the joint action with maximum total payoff, the agents first condition on the context and eliminate all rules that are inconsistent with the current state. Then a *variable elimination* algorithm is applied in which each agent first solves a local maximization problem (which depends only on its neighbors in the graph) and then communicates the resulting conditional strategy to one of its neighbors. After this, the communicating agent is eliminated from the graph. This procedure continues until only one agent remains, which then determines its contribution to the optimal joint action based on the conditional strategies of all agents. Thereafter, a pass in the reverse order is performed in which all eliminated agents fix their strategies based on the selected actions of their neighbors. After completion of the algorithm, the selected joint action corresponds to the optimal joint action that maximizes the sum of the payoff of the applicable value rules for the current state. Although the elimination order does not have an effect on the outcome of the algorithm, it does have an effect on the needed computation time. We refer to [Guestrin et al., 2002b] for details.

We applied coordination graphs successfully in our RoboCup simulation team by manually specifying both the coordination dependencies and the associated payoffs using value rules [Kok et al., 2004]. This resulted in the world champion title in the RoboCup-2003 soccer simulation league, illustrating that such a representation can capture very complex and effective policies.

In [Kok and Vlassis, 2004] coordinated behavior is learned using the concept of CGs and variations of Q-learning. We will refer to this method as 'Sparse Cooperative Q-learning'. In that work a predefined set of value rules is specified that captures the coordination dependencies of the system. At each time step the global Q-value equals the sum of the local Q-values of all n agents. The local Q-value, $Q_i(s, a)$ of an agent i depends on the payoff of the value rules in which agent i is involved and that is

consistent with the given state-action pair (s, a):

$$Q_i(s,a) = \sum_j \frac{\rho_j^i(s,a)}{n_j}, \quad (3)$$

where each payoff is divided proportionally over the n_j involved agents. Such a representation of $Q_i(s,a)$ can be regarded as a linear expansion into a set of basis functions ρ_j^i, each of them peaked on a specific state-action context which may potentially involve many agents. In the sparse cooperative Q-learning method, the 'weights' of these basis functions (the values of the rules) are updated as follows:

$$\rho_j(s,a) := \rho_j(s,a) + \alpha \sum_{i=1}^{n_j} [R_i(s,a) + \gamma Q_i(s',a^*) - Q_i(s,a)]. \quad (4)$$

Note that each rule is updated based on their local contribution for the global optimal joint action. In order to compute this joint action $a^* = \arg\max_a Q(s,a)$ that maximizes the sum of the (local) payoffs for state s, the variable elimination algorithm is applied. From this, the agents can determine their contribution $Q_i(s',a^*)$ to the total payoff. A rule is updated by adding the individual reward and individual expected future reward of each agent involved in the rule, similar to Eq. (2). Effectively, each agent learns to coordinate with its neighbors, in a context-specific manner.

As an example, assume we have the following set of value rules[4]:

$$\begin{aligned}
&\langle \rho_1 \,;\, a_1 \wedge s &&: v_1 \rangle \\
&\langle \rho_2 \,;\, \overline{a}_1 \wedge a_2 \wedge s' &&: v_2 \rangle \\
&\langle \rho_3 \,;\, a_1 \wedge \overline{a}_2 \wedge s' &&: v_3 \rangle \\
&\langle \rho_4 \,;\, a_1 \wedge \overline{a}_2 \wedge s &&: v_4 \rangle \\
&\langle \rho_5 \,;\, a_2 \wedge a_3 \wedge s &&: v_5 \rangle \\
&\langle \rho_6 \,;\, \overline{a}_3 \wedge s' &&: v_6 \rangle
\end{aligned}$$

Furthermore, assume that $a = \{a_1, a_2, a_3\}$ is the performed joint action in state s and $a^* = \{a_1, \overline{a}_2, \overline{a}_3\}$ is the optimal joint action found with the variable elimination algorithm in state s'. After conditioning on the context, the rules ρ_1 and ρ_5 apply in state s, whereas the rules ρ_3 and ρ_6 apply in state s'. This is graphically depicted in Fig. 3. Next, we use Eq. (4) to update the value rules ρ_1 and ρ_5 in state s as follows:

$$\begin{aligned}
\rho_1(s,a) &= v_1 + \alpha[R_1(s,a) + \gamma \frac{v_3}{2} - \frac{v_1}{1}] \\
\rho_5(s,a) &= v_5 \; + \; \alpha[R_2(s,a) + \gamma \frac{v_3}{2} - \frac{v_5}{2} + \\
&\qquad R_3(s,a) + \gamma \frac{v_6}{1} - \frac{v_5}{2}].
\end{aligned}$$

Note that in order to update ρ_5 we have used the (discounted) Q-values of $Q_2(s',a^*) = v_3/2$ and $Q_3(s',a^*) = v_6/1$. Furthermore, the component Q_2 in state s' is based on a coordinated action of agent A_2 with agent A_1 (rule ρ_3), whereas in state s agent A_2 has to coordinate with agent A_3 (rule ρ_5).

[4] Action a_1 corresponds to $a_1 = true$ and action $\overline{a}_1$ to $a_1 = false$.

In the above description, we assume that the coordination dependencies among the agents are specified beforehand. In the next section we describe how these dependencies can be *learned* automatically by starting with an initial set of (individual) rules based on individual actions and dynamically adding rules for those states where coordination is found to be necessary.

4 Utile Coordination

Previous work using coordination graphs assumes a known CG topology. In this section we describe our method to learn the coordination dependencies among the agents automatically. Our approach builds on the ideas of Chapman & Kaelbling's [Chapman and Kaelbling, 1991] and McCallum's [McCallum, 1997] adaptive resolution RL methods for the single agent case. These methods construct a partitioning of an agent's state space based on finding so-called 'Utile Distinctions' [McCallum, 1997] in the state representation. These are detected through statistics of the expected returns maintained for hypothesized distinctions. For every state, this method stores the future discounted reward received after leaving this state and relates it to an incoming transition (the previous state). When a state is Markovian with respect to return, the return values on all incoming transitions should be similar. However, if the statistics indicate that the returns are significantly different, the state is split to help the agent predict the future reward better. This approach allows a single agent to build an appropriate representation of the state space.

In our Utile Coordination algorithm, we take a similar approach. The main difference is that, instead of keeping statistics on the expected return based on incoming transitions, we keep statistics based on the performed actions of the other agents. The general idea is as follows. The algorithm starts with independent uncoordinated learners[5], but over time learns, based on acquired statistics, where the independent learners need to coordinate. If the statistics indicate there is a benefit in coordinating the actions of independent agents in a particular state, that state becomes a coordinated state. In the CG framework, this means new coordinated value rules are added to the coordinated graph.

Statistics of the expected return are maintained to determine the possible benefit of coordination for each state. That is, in each state s where coordination between two (or more) agents in a set I is considered, a sample of the 'combined return', $\hat{Q}_I(s, a_I)$, is maintained after a joint action a is performed. The combined return is an approximation of the expected return that can be obtained by the involved agents in I and equals the sum of their received individual reward and their individual contribution $Q_i(s',a^*)$ to the maximal global Q-value of the next state as in Eq. (4):

$$\hat{Q}_I(s,a_I) = \sum_{i \in I} \hat{Q}_i(s,a) = \sum_{i \in I} [R_i(s,a) + \gamma Q_i(s',a^*)]. \quad (5)$$

[5] Note that it is also possible to start with an initial CG incorporating coordination dependencies that are based on prior domain-specifi c knowledge.

These samples are stored with the performed action a_I. For each of these joint actions, the expected combined return can be estimated by computing the mean, $\bar{Q}_I(s, a_I)$, of the last M samples[6]. These statistics are never used to change the agent's state-action values, but are stored to perform a statistical test at the end of an m-length trial to measure whether the largest expected combined return for a state s, $\max_{a_I} \bar{Q}_I(s, a_I)$ (with variance $\sigma^2_{\max}$), differs significantly from the expected combined return $\bar{Q}_I(s, a^*_I)$ (with variance σ^2_*). The latter is the return obtained when performing the greedy joint action a^*_I in the state s (and thus corresponds to the actually learned policy). This greedy joint action can be found using the variable elimination algorithm. Note that initially, when all states are uncoordinated, a^*_I corresponds to the vector of individually greedy actions: $a^*_i = \arg\max_{a_i} Q_i(s, a_i)$.

We use the t-test [Stevens, 1990] as the statistical test to compare the two values:

$$t = \frac{\max_{a_I} \bar{Q}_I(s, a_I) - \bar{Q}_I(s, a^*_I)}{\sqrt{[(2/M)((M-1)\sigma^2_{\max} + (M-1)\sigma^2_*)/(2M-2)]}} \quad (6)$$

with $(2M - 2)$ degrees of freedom. From this value the level of significance, p, is computed indicating the probability of rejecting the null hypothesis (the two groups are equal) when it is true.[7]

An additional statistical *effect size measure* d determines whether the observed difference is not only statistically significant, but also sufficiently large. In this paper d is similar to standard effect size measures [Stevens, 1990], and it relates the difference in means to the observed maximum and minimum reward available in the task:

$$d = \frac{\max_{a_I} \bar{Q}_I(s, a_I) - \bar{Q}_I(s, a^*_I)}{r_{max} - r_{min}}. \quad (7)$$

If there is a statistically significant difference ($p < P$) with sufficient effect size ($d > D$), there is a significant benefit of coordinating the agents' actions in this state: apparently the current CG leads to significantly lower returns than the possible returns when the actions are coordinated. This can occur in the situation where one specific joint action will produce a high return but all other joint actions will get a substantially lower return (see the example at the end of section 2). Since the agents select their actions individually they will only occasionally gather the high return. However, when the stored statistics (based on joint actions) are compared with the current policy, this produces a statistical difference indicating that it is beneficial to change this state into a coordinated state. In our CG framework, the value rules based on individual actions are replaced by value rules based on joint actions for this particular state. The value of each new rule $\rho(s, a_I)$ is initialized with the learned value $\bar{Q}_I(s, a_I)$.

As coordination rules are added, the samples of $\hat{Q}_I(s, a_I)$ may correspond to *joint* actions of two agents that now coordinate in a particular state, such that now coordination between three or more agents can be learned. Alternatively, 3D, 4D, etc., tables can be constructed for three or more independent learners to test for coordination benefits when coordination between only 2 agents is not beneficial. In any case, the statistical test always looks at only two estimates of expected combined return: $\max_{a_I} \bar{Q}_I(s, a_I)$ and $\bar{Q}_I(s, a^*_I)$.

In very large state-action spaces, memory and computation limitations will make it infeasible to maintain these statistics for each state. In fact, those are the most interesting cases, because there learning sparse coordination is most useful, as opposed to the full coordination done by JAMDP learners. It then makes sense to use a heuristic 'initial filter' which detects potential states where coordination might be beneficial. The full statistics on combined returns are then only maintained for the potential interesting states detected by the initial filter. In this way, large savings in computation and memory can be obtained while still being able to learn the required coordination. One useful heuristic for filtering may be to compare $Q_i(s, a)$ to the *mode* of the last M samples of $\hat{Q}_i(s, a)$ stored in a small histogram. If they are sufficiently different, this indicates multi-modality of expected returns for this agent i, which may mean a potentially dependence on other agents' actions. In this paper, the emphasis is on showing the validity of the Utile Coordination algorithm and its learning efficiency compared to independent learners and JAMDP learners. Therefore, in the experiments reported below no heuristic initial filter is used and statistics are stored for every state.

5 Experiments

In this section, we apply the Utile Coordination algorithm to two problems: the example of section 2 and to the much larger predator-prey domain.

5.1 Small Coordination problem

In this section, we apply our algorithm to the simple intuitive problem depicted in Fig. 1 and compare it to the two Q-learning methods mentioned in section 2, the JAMDP learners and the Independent Learners (ILs). The latter only keep Q-values for their individual actions and therefore $42 (= 2 \cdot 7 \cdot 3)$ Q-values are stored in total. The JAMDP learners model the joint action for every state resulting in $63 (= 7 \cdot 3^2)$ Q-values. Just as with the ILs, our Utile Coordination approach starts with value rules based on individual actions; but it checks, after $m = 1000$ steps, for every state whether the action of the other agent should be incorporated. We use an ϵ-greedy exploration step[8] of 0.3, a learning rate $\alpha = 0.25$, and a discount factor $\gamma = 0.9$. For

[6] In our experiments, we used $M = 10$.

[7] Other statistical tests that compare two groups are possible. In particular, nonparametric tests may be used, because assumptions of normality and homogeneity of variance may be violated. However, the t-test is fairly robust to such violations when group sizes are equal (as in our case) [Stevens, 1990].

[8] Note that for fair comparison of the results, independent learners explore jointly in all experiments. That is, with probability ϵ both agents select a random action, which is more conservative than each agent independently choosing a random action with probability ϵ.

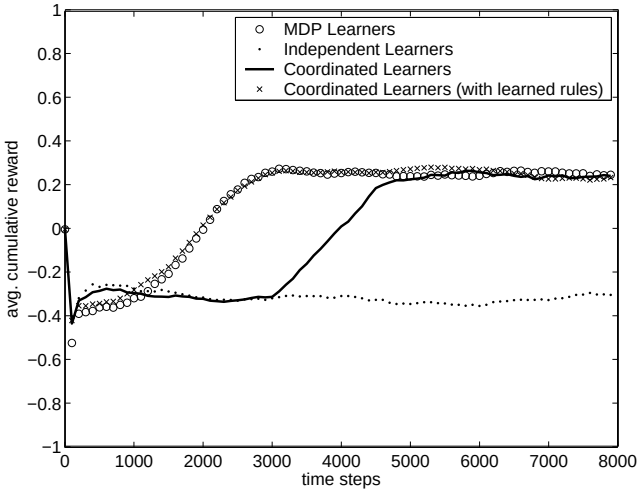


Figure 4: Running average of total cumulative reward of the previous 1500 time steps (including exploration) for the different Q-learners in the problem from section 2. Results are averaged over 30 runs.

the parameters in our Utile Coordination approach we use a significance level $P = 0.05$ and an effect size $D = 0.01$.

Fig. 4 shows the running average of the cumulative reward (including exploration) for the three different Q-learning approaches. The independent learners do not converge to the optimal policy since the actions resulting in a low reward have a large negative impact on the Q-values corresponding to the individual actions of the optimal joint action. The JAMDP learners do not have this problem, since they model each joint action and quickly learn to converge to the optimal policy. Since our Utile Coordination approach starts with individual value rules, the learning curve resembles that of the independent learners in the beginning. However, after the third trial (3000 time steps), appropriate coordination is added for state s_0 in all 30 runs, and thereafter the system converges to the optimal policy. Fig. 4 also shows that simulation runs that start with the learned coordination dependencies found with the Utile Coordination approach produce identical results as the JAMDP learners. Although the learned representation of the Utile Coordination approach uses a sparser representation, both methods quickly converge to the optimal policy.

5.2 Predator-prey problem

In this section, we apply our Utile Coordination algorithm to the predator-prey problem. We concentrate on a problem where two predators have to coordinate their actions in order to capture a prey in a 10×10 toroidal grid. Each agent can either move to one of its adjacent cells or remain on its current position. In total this yields $242,550$ (joint) state-action pairs. All agents are initialized at random positions at the beginning of an episode. An episode ends when the prey is captured. This occurs when both predators are located in cells adjacent to the prey and only one of the two agents moves to the location of the prey and the other remains on its current position. Fig. 5 shows an example grid in which the predators will capture the prey when either the predator north of the prey, or the prey east of the prey will move

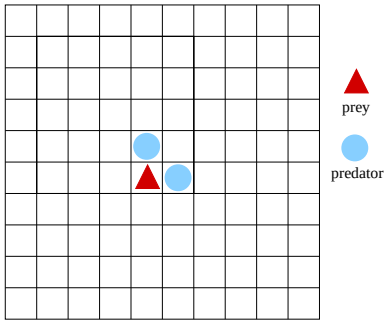


Figure 5: Graphical representation of a 10×10 grid with two agents and one prey. This situation shows a possible capture position for two predators. The prey is only captured when one of the two agents moves to the prey position and the other remains on its current position.

to the prey position and the other predator will remain on its current position. A predator is penalized and placed on a random position on the field when it moves to the prey position without coordinating with the other predator, or moves to the same cell as the other predator. The predators thus have to coordinate their actions in all states in which they are close to each other or when they are close to the prey. In all other states, the agents can act individually. The prey behavior is fixed: it remains on its current position with a probability of 0.2 and otherwise moves moves to one of its free adjacent cells with uniform probability.

Just as with the small coordination problem, we will apply our method and compare it with the two other Q-learning approaches. Each predator i receives an (individual) reward $R_i = 37.5$ when it helps to capture the prey, a reward of -25.0 when it moves to the prey without support, a reward of -10 when it collides with another predator, and a reward of -0.5 in all other situations to motivate the predators to capture the prey as quickly as possible. We use an ϵ-greedy exploration step of 0.3, a learning rate $\alpha = 0.25$, and a discount factor $\gamma = 0.9$. Again, we

use a significance level $P = 0.05$ and and an effect size $D = 0.01$ for the Utile Coordination approach. Statistical tests to determine coordination are performed after every $m = 20,000$ episodes.

Fig. 6 shows the capture times for the learned policy during the first $400,000$ episodes for the different methods (running average of the capture times of the last 300 episodes is shown) and includes the exploration steps taken by the agents. The results are averaged over 10 runs. The IL approach does not converge to a stable policy but keeps oscillating; the Q-values for the individual actions for capturing the prey are decreased substantially when an action is performed that results in an illegal movement to the prey. The JAMDP learners model these dependencies explicitly in every state which results in convergence to the optimal policy. Our Utile Coordination approach initially does not take these dependencies into account and follows the curve of the independent learners. However, after the end of the first trial (episode $20,000$), the agents add coordinated value rules for the states in which the gathered statistics indicate that coordination is beneficial, and immediately the capture times decrease as is visible in Fig. 6. Thereafter, the average capture times keep decreasing slowly as more fine-grained coordination dependencies are added and the agents learn in the updated coordination graph structure. At the end, while new value rules are added, the found policy is similar to the policy found by the JAMDP Learners.

Table 1 shows the final capture times and the number of Q-values needed to represent the state-action space for each method. For the Utile Coordination approach on average $457.90\,(\pm 53.4)$ out of 9702 states were found to be statistically significant and added as coordinated states. This is in contrast with the $1,248$ manually specified states in [Kok and Vlassis, 2004], where coordinated rules were added for all states in which the predators where within two cells of each other or both within two cells of the prey. This difference is caused by the fact that for many states where a collision is possible and the agents have to coordinate their actions, the agents are able to learn how to avoid the collision independently and no specific coordination rule is needed.

When the learned coordination dependencies of the Utile Coordination approach are used to learn the policy of the agents, the learning curve is similar to that of the JAMDP learners. However, the latter needs a larger representation to store the Q-values. In this experiment, this does not result in a negative influence on the learning curve because of the relative small joint action-size. However, for lager problems with more agents (and more agents dependencies), this will be more severe.

Both the Utile Coordination approach and the approach based on the learned rules converge to a slightly higher capture time than that of the JAMDP Learners, indicating that coordinating in some states, not statistically significant for the Utile Coordination approach, has a very small positive influence on the final result.

6 Conclusion and future work

This paper introduced the Utile Coordination algorithm, which starts with independent, non-coordinating agents and learns automatically where and how to coordinate. The method is based on maintaining statistics on expected returns for hypothesized coordinated states, and a statistical test that determines whether the expected return increases when actions are explicitly coordinated, compared to when they are not. We implemented this method within the framework of coordination graphs, because of its attractive properties of representing compactly and efficiently the agents' state-action space, values, RL updates, and context-specific coordination dependencies. In this context, the method can be understood as testing, for a given CG, the standard CG assumption that the overall return $Q(s, a)$ is simply the sum of the individual components $Q_i(s, a)$. If this assumption is violated, the algorithm adds appropriate value rules to make the resulting CG adhere to the assumption.

There are many avenues for future work. As described before, maintaining the complete statistics for all states is not computationally feasible for large CMMDPs. Heuristic initial filters should be investigated in detail, such that the Utile Coordination algorithm can be applied to such large problems. In particular, tasks with many interacting agents should be investigated, as these are the tasks where the problem of maintaining full statistics is most obvious, and where at the same time the advantage of Utile Coordination over JAMDP learners, in terms of space and learning time, will be more pronounced.

Heuristic initial filters are not the only way to deal with large state spaces. An equally important, orthogonal possibility is a variation of the Utile Coordination algorithm based on more sophisticated, e.g., factorial or relational, state representations. This should combine well with coordination graphs, because they were explicitly designed for such state representations. An individual agent would then be able to represent only its own individual view of the environment state. Furthermore, it could for instance learn to coordinate with another agent 'when the other agent is near', rather than having to represent explicitly all environment states when it is near the other agent and learn to coordinate separately for all those states.

Finally, we note that it should be possible to use the same statistical tests to allow *pruning* of coordination rules if they turn out to be of little use. Also, a user may inject some coordination rules into the algorithm based on a priori knowledge, and the system can subsequently learn additional rules or prune superfluous user-inserted rules. In this way, a priori knowledge and learning can be combined fruitfully.

Bibliography

[Boutilier, 1996] Boutilier, C. (1996). Planning, learning and coordination in multiagent decision processes. In *Proc. Conf. on Theoretical Aspects of Rationality and Knowledge*.

[Chalkiadakis and Boutilier, 2003] Chalkiadakis, G. and Boutilier, C. (2003). Coordination in multiagent reinforcement learning: A bayesian approach. In *Proc. of the 2nd*

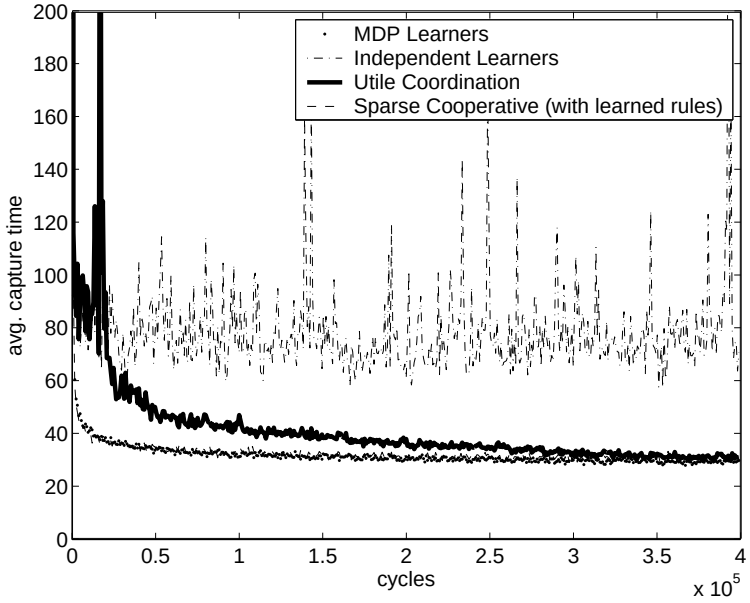


Figure 6: Running average of the capture times (over the last 300 episodes) for the learned policy of the four different methods during the first $400,000$ episodes. Results are averaged over 10 runs. Note that the learning curve of the JAMDP and the curve based on the learned representation (bottom curve) overlap and are almost similar.

Table 1: Average capture time after learning (averaged over the last 1,000 episodes) and the number of state-action pairs for the different methods.

Method	**avg. time**	**#Q-values**
Independent learners	68.77	97,020
JAMDP Learners	29.71	242,550
Utile Coordination	30.57	105,868
Sparse Cooperative (using learned rules)	30.93	105,868

Int. Joint Conf. on Autonomous agents and multiagent systems, pages 709–716, Melbourne, Australia. ACM Press.

[Chapman and Kaelbling, 1991] Chapman, D. and Kaelbling, L. P. (1991). Input generalization in delayed reinforcement learning: An algorithm and performance comparisons. In Mylopoulos, J. and Reiter., R., editors, *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence (IJCAI-91)*, pages 726–731, San Mateo, Ca. Morgan Kaufmann.

[Claus and Boutilier, 1998] Claus, C. and Boutilier, C. (1998). The dynamics of reinforcement learning in cooperative multiagent systems. In *Proc. 15th Nation. Conf. on Artificial Intelligence*, Madison, WI.

[Guestrin, 2003] Guestrin, C. (2003). *Planning Under Uncertainty in Complex Structured Environments*. PhD thesis, Computer Science Department, Stanford University.

[Guestrin et al., 2002a] Guestrin, C., Lagoudakis, M., and Parr, R. (2002a). Coordinated reinforcement learning. In *Proceedings of the Nineteenth International Conference on Machine Learning*.

[Guestrin et al., 2002b] Guestrin, C., Venkataraman, S., and Koller, D. (2002b). Context-specifi c multiagent coordination and planning with factored MDPs. In *Proc. 8th Nation. Conf. on Artificial Intelligence*, Edmonton, Canada.

[Kok et al., 2004] Kok, J. R., Spaan, M. T. J., and Vlassis, N. (2004). Noncommunicative multi-robot coordination in dynamic environments. *Robotics and Autonomous Systems*. In press.

[Kok and Vlassis, 2004] Kok, J. R. and Vlassis, N. (2004). Sparse Cooperative Q-learning. In Greiner, R. and Schuurmans, D., editors, *Proc. of the 21st Int. Conf. on Machine Learning*, pages 481–488, Banff, Canada. ACM.

[McCallum, 1997] McCallum, R. A. (1997). *Reinforcement Learning with Selective Perception and Hidden State*. PhD thesis, University of Rochester, Computer Science Department.

[Shapley, 1953] Shapley, L. (1953). Stochastic games. *Proceedings of the National Academy of Sciences*, 39:1095–1100.

[Stevens, 1990] Stevens, J. P. (1990). *Intermediate statistics: A modern approach*. Lawrence Erlbaum.

[Stone and Veloso, 2000] Stone, P. and Veloso, M. (2000). Multiagent systems: a survey from a machine learning perspective. *Autonomous Robots*, 8(3).

[Sutton and Barto, 1998] Sutton, R. S. and Barto, A. G. (1998). *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, MA.

[Vlassis, 2003] Vlassis, N. (2003). A concise introduction to multiagent systems and distributed AI. Informatics Institute, University of Amsterdam. http://www.science.uva.nl/˜vlassis/cimasdai.

Forcing neurocontrollers to exploit sensory symmetry through hard-wired modularity in the game of Cellz

Julian Togelius
Department of Computer Science
University of Essex
Colchester, Essex, CO4 3SQ
julian@togelius.com

Simon M. Lucas
Department of Computer Science
University of Essex
Colchester, Essex, CO4 3SQ
sml@essex.ac.uk

Abstract- Several attempts have been made in the past to construct encoding schemes that allow modularity to emerge in evolving systems, but success is limited. We believe that in order to create successful and scalable encodings for emerging modularity, we first need to explore the benefits of different types of modularity by hard-wiring these into evolvable systems. In this paper we explore different ways of exploiting sensory symmetry inherent in the agent in the simple game Cellz by evolving symmetrically identical modules. It is concluded that significant increases in both speed of evolution and final fitness can be achieved relative to monolithic controllers. Furthermore, we show that simple function approximation task that exhibits sensory symmetry can be used as a quick approximate measure of the utility of an encoding scheme for the more complex game-playing task.

1 Background

The current interest in exploring and exploiting modularity in evolutionary robotics can be understood in several ways: as a way of studying modularity in biological evolved systems, as a way of making evolution produce systems which are easier (or at least possible) for humans to understand and thus to incorporate into other human-made systems, and as a means of scaling up evolutionary robotics beyond the simple behaviours which have been evolved until now. In our opinion, these perspectives are complementary rather than exclusive.

For those interested in evolving autonomous agents for computer games, certainly the two latter perspectives are the most important. Agents will need to be able to perform complex tasks, such as serving as opponents to human players, in environments constructed for human players, and their internal structure should ideally be amenable to changes or enhancements from game constructors.

The ways in which modularity can help scaling up and make evolved solutions comprehensible is by improving network updating speed, reducing search dimensionality, allowing for reusability, and diminishing neural interference.

When a neural network is divided up into modules, the number of connections for the same number of neurons can be significantly reduced compared to a non-modular, i.e. a fully connected network. As propagating an activation value along a connection is the most frequent operation performed when updating a neural network, the time needed for updating the network can be likewise significantly reduced. This not only allows the controller to be used in time-critical operations, like real-time games, but it also speeds up evolution.

However, even if a modular network has the same number of connections as its modular counterpart, as is the case with the architectures presented in this paper, evolution can be sped up by modularity. In most encodings of neural networks, the length of the genome is directly proportional to the number of connections, but when several modules share the same specifications, the genome for a modular network might be significantly smaller than for a non-modular network with the same number of connections. This reduces the dimensionality of the space the evolutionary algorithm searches for the solution in, which can improve the speed of evolution immensely.

Neural interference (Calabretta et al. 2003) refers to the phenomenon that the interconnection of unrelated parts of a neural network in itself can hamper evolution, because any mutation is likely to set that interconnection to a non-zero value, which means that activity in these non-related parts of the network interfere with each other. A good modularisation alleviates this problem.

Finally, many problems arising in computer games and elsewhere have the property that parts of their solutions can be reused in other problems arising in similar context. An evolutionary algorithm that could reuse neural modules in the development of new solutions could cut evolution time in such circumstances.

The flipside to all this is that not every architecture is a modular architecture, and constraining your network to modular topologies means running the risk of ruling out the best architectural solutions; constraining your network to weight-sharing (reusable) modules means even more constraints, as this is optimal only when there is indeed some repeating problem structure to exploit.

Many attempts have in the past been made to achieve the benefits outlined above while minimizing the negative effects of topological constraints. Several of these attempts try to allow for the division of the neurocontroller into modules to emerge during evolution, instead of explicitly specifying the modules. For example, Cellular Encoding (Gruau 1994), inspired by L-systems (Lindenmayer 1968) grows neural networks according to information specified in graphs, allowing segments of the network to be repeated several times. Gruau's architecture

has been put to use and expanded by other researchers, such as Hornby et al. (2001) and Kodjabachian and Meyer (1995). An alternative modular encoding, called Automatically Defined Functions (Koza 1994) is used in Genetic Programming. Bongard (2003) has recently devised an encoding based on gene regulation, which is capable of producing modular designs; a good overview of approaches such as those mentioned above is given in (Stanley & Miikkulainen 2003).

However, even in these encoding schemes, human design choices arguably influence the course of evolution; some forms of modularity are more likely to evolve than others. For example, a given encoding scheme might be better suited for evolving modules that connect to each other in a parallel fashion than for evolving modules that connect together in a hierarchic fashion. At the same time, we usually don't have a theory of what sort of modularity would best benefit a particular combination of task, environment and agent. This could be why these encodings, though mathematically elegant, have failed to scale up beyond very simple tasks, at least in neural network-based approaches. Furthermore, these encodings seem very poor at expressing re-usable modules compared to languages used for expressing hardware or software designs, such as VHDL or Java respectively. To properly express modular designs, it is necessary to allow specification not only of the details of a module, but also how modules may be sensibly interconnected together, and how new module designs may be constructed from existing ones via delegation and inheritance. The concepts of evolving objects (Keijzer et al, 2001), or object-oriented genetic programming (Lucas, 2004) suggest some promising directions, but more work is needed in these areas.

We believe that the complementary approach of explicitly defining and hard-wiring modules and their interrelations could be useful in investigating what sorts of modularity are best suited to any particular problem, or problem class; knowledge which would be useful when developing new encoding schemes allowing for emergent modularity. We also believe that explicit modular definition will scale up better than any other method in use today.

Raffaele Calabretta and his colleagues have reported increased evolvability from hard-coded modularity (with non-identical modules) in different contexts, such as robotic can-collecting (Calabretta et al. 2000) and a model of the *what and where* pathways of the primate visual system (Calabretta et al. 2003), but note the conflicting findings of Bullinaria (2002).

Of special interest in our approach are cases where aspects of the problem or agent show some form of symmetry, so that identical modules can be evolved and replicated in several positions in the controller, using different inputs. Little work seems to have been done on this, but note Vaughan (2003) who evolves a segmented robot arm with identical modules, and Schraudolph et al. (1993) use tiled neural networks that take advantage of the symmetry inherent in the game Go. However the problem of playing Go is very different than playing most computer games. They also use temporal difference learning rather than evolution.

In this paper, we are comparing the results and dynamics of evolving monolithic networks (standard multi-layer perceptrons) of different sizes with those of evolving modular architectures with identical modules that exploit sensory symmetry. The evolved neural networks are compared both in terms of maximum fitness, fitness growth, and behaviour of the resulting controllers.

As a test bed, we have used the game Cellz, which has the benefit that the agent has 8-way radial symmetry. While Cellz was developed especially for testing evolutionary algorithms, the computational expense can still be prohibitive for exploring large parameter spaces. Therefore, we have constructed a simple function approximation task to have similar difficulty and demands on network architecture as the Cellz control task, but which evaluates much faster. Experiments with different network architectures were carried out first using the function approximation task, and then using Cellz, and the qualitative similarity of results using these two tasks were investigated.

2 Methods

2.1 Cellz

The game of Cellz (Lucas 2004) has been designed as a test bed for evolutionary algorithms. The game was run as a competition for the GECCO 2004 conference, and the source code is available on the web. The elements of the game are a number of cellz and a number of food particles, and the objective of the game is for the cellz to eat as many food particles as possible. A cell eats a food particle by moving over it, which increases its mass; when its mass increases over a threshold it splits into two. The food particle, upon being eaten, vanishes and reappears somewhere else on the game area. A cell moves by applying a force vector to itself, which trades some of its mass for changing its speed – the problem of movement is not trivial, having to take momentum and friction into account. Neither is the problem of deciding which food particles to go for, which in the case of only one cell is an instance of the travelling salesman problem, but quickly becomes more complex as other cells are added. A major problem is not to go for a food particle that another cell will get to first. Furthermore, each game starts with the cells and the food in random locations, and each new piece of food is added in a random location, which means that evolution should aim to acquire general good behaviours rather than those that just happen to work well for a particular game configuration. Figure 1 shows the trace of a part of a game run using an evolved perceptron controller (from Lucas 2004), and illustrates how the cells (thick lines) move in chaotic patterns in their attempts to eat food (dots) and divide.

Each cell is equipped with eight cell sensors and eight food sensors spread evenly around its body; (Figure 2) each sensor measures the distance to and concentration of other cellz or food in its 45 degree angle. The sensor

arrays are used as inputs to the controllers, and their outputs are used to generate the force vectors.

The total mass of all cells was used as fitness value for each game, which was run for 1000 time steps, and the fitness value for each individual in each generation was computed as the mean of ten such games in order to reduce noise.

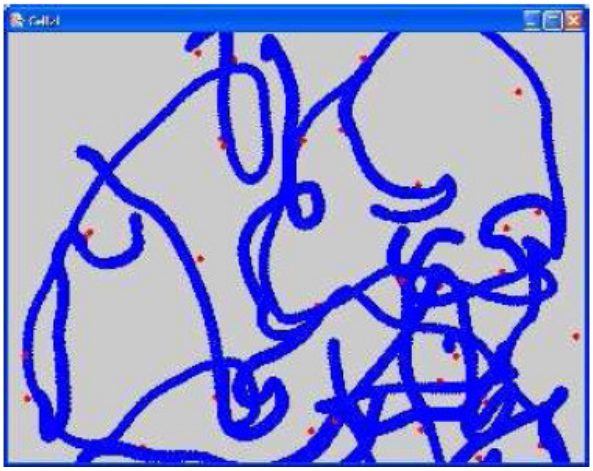


Figure 1: A sample run of Cellz with an evolved perceptron controller (from Lucas 2004).

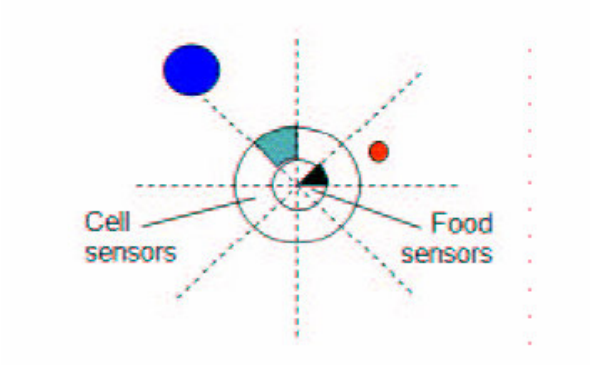


Figure 2: The wrap around input sensors. From Lucas (2004).

2.2 Neural networks

Four different neural architectures were tested and compared. In all of them, each neuron implemented a *tanh* activation function, and the synapse weights were constrained to be in the range [-1..1], as were inputs and outputs.

The first two architectures were standard multi-layer perceptrons (MLPs). The first MLP consisted of an input layer of 16 neurons, an 8 neuron hidden layer and an output layer of two neurons. The second MLP had two hidden neuron layers of 16 neurons each. In both architectures, positions 0-7 received inputs from the ”food” input vector of the Cellz agent, positions 8-15 received inputs from the ”cells” input vector, and the two outputs from the network were used to create the force vector of the cell.

The other two “convoluted” architectures consist of eight separate but identical neural network modules - they share the same genome. Each module can be thought of as assigned to its own pair of sensors, and thus being at the same angle r relative to the x axis as those sensors. The outputs from the each module’s two output units is rotated –r degrees, and then added to the summed force vector output of the controller.

In the convoluted architectures, each module gets the full range of sixteen inputs, but they are displaced according to the position of the module (e.g. module number 3 gets food inputs 3, 4, 5, 6, 7, 0, 1, 2, in that order, while the input array to module 7 starts with sensor 7; Figure 3). In the first convoluted architecture the modules lack hidden layer, but in the second convoluted architecture, each has a hidden layer of two neurons.

It is interesting to compare the number of synapses used in these architectures, as that number determines the network updating speed and the dimensionality of the search space. The MLP with 8 hidden neurons has 144 synapses, while the MLP with two hidden layers totals 544 synapses. The perceptron-style convoluted controller has 32 synapses per module, which sums to 256 synapses, and the hidden-layer convoluted controller has 36 synapses per module, which sums to 288 synapses. It should be noted that while the convoluted controllers have little or no advantage over the MLPs when it comes to updating speed, they present the evolutionary algorithm with a much smaller search space, as only 32 or 36 synapses are specified in the genome.

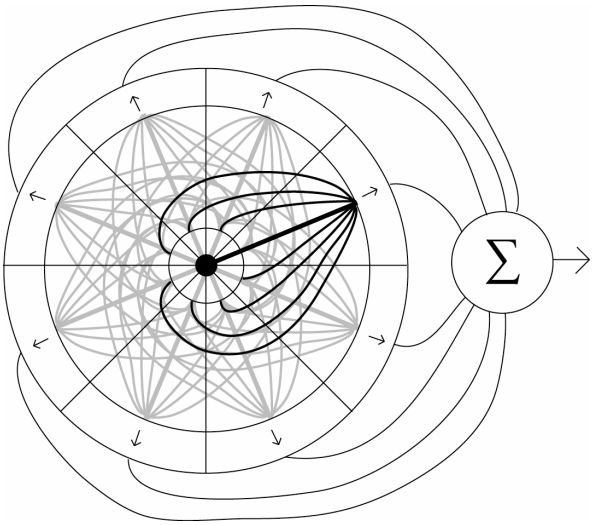


Figure 3: Simplified illustration of the convoluted architectures, taking only one type of sensor into account. The connections in black are the connections from all sensors to one module; this structure is repeated (grey lines) for each module.

2.3 Function approximation task

Like the Cellz task, the function approximation task requires the network to have 16 inputs and 2 outputs. The input array is divided into two consecutive arrays of 8 positions; each position has an associated angle in the same manner as the Cellz controller. Each time a network is evaluated, a random position on an imaginary circle, i.e. a random number in the range [0..2π], is produced. The network inputs receive activations corresponding in a nonlinear fashion to their associated angles' distance to the target position. The function to be approximated by the outputs of the network is the sine and cosine of the target position, and the fitness function is the mean absolute summed difference between these values and the actual network outputs. The time it takes to evaluate a neural network on the function approximation task is on the order of a thousand times less than the time taken to evaluate the same network as a neurocontroller for Cellz.

2.4 Evolutionary algorithm

Controllers for the agents were evolved using an evolutionary algorithm with a population size of 30. Truncation selection was used, and elitism of 5; at each generation, the population was sorted on fitness, the worse half was replaced with clones of the better half, and all controllers except the top 5 were mutated. Mutation consisted of perturbing all synaptic weights by a random value, obeying a Gaussian distribution with mean 0 and standard deviation 0.1.

3 Results

In all the graphs presented in this section, the dark line tracks the fitness of the best controller in each generation, while the other line represents the mean population fitness.

3.1 Evolving function approximators

For the function approximation problem, we define fitness to be the negative of the mean error – which gives a best possible fitness of zero. Each figure in this sub-section depicts the mean of ten evolutionary runs. Both monolithic (MLP) architectures were evolved for 100 generations. (Figures 4 and 5) They eventually arrived at solutions of similar quality, though the dual-layer MLP took longer time to get there.

The perceptron-style convoluted network reached fitness similar to that of the monolithic networks, but somewhat faster (Figure 6). The real difference, though, was with the convoluted network with one hidden layer; it achieved much higher fitness than any of the three other architectures, and did so with fewer fitness evaluations (Figure 7). In this case, we are observing a very clear benefit of the enforced modular structure. Next we investigated how well this function approximation task serves as a test-bed for the real game, which has a significant degree of noise, together with complex dynamics. While the network weights to solve each task are likely to be very different, the overall connection topologies, and the constraints on those topologies are likely to be rather similar, based on our construction of the function approximation task. Hence, while the evolved weights cannot be transferred from the function approximation task to the game, it is still possible that the both the task and the game measure similar qualities of the encoding schemes.

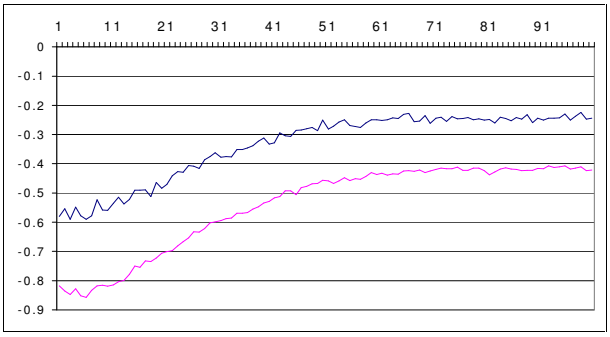


Figure 4: MLP with one hidden layer on the function approximation task.

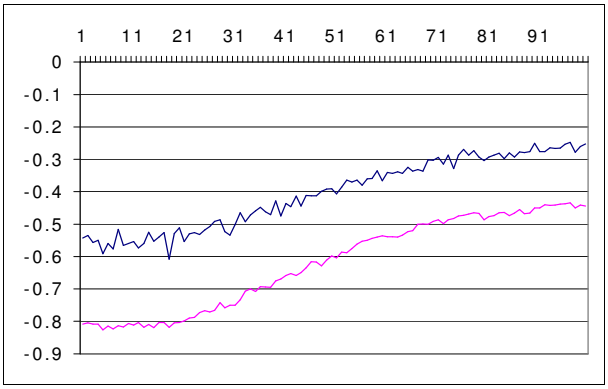


Figure 5: MLP with two hidden layers on the function approximation task.

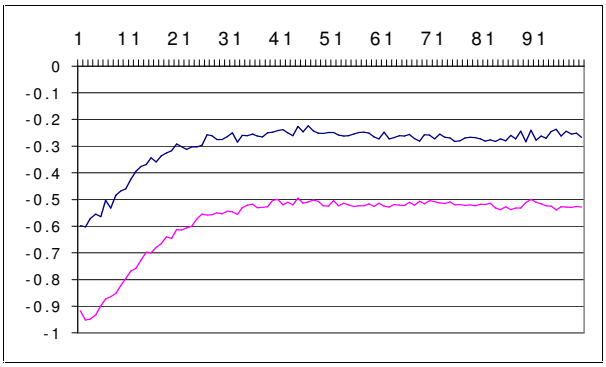


Figure 6: Perceptron-style convoluted network on the function approximation task.

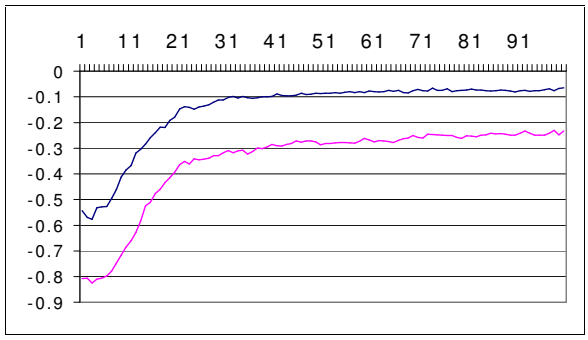


Figure 7: Convoluted network with one hidden layer on the function approximation task.

3.2 Evolving Cellz controllers

The two MLP architectures were evolved for 200 generations. Each figure now shows a single run, but each experiment was repeated several times, and the graphs shown are representative. The one-layer MLP evolved somewhat faster and reached a higher final fitness. Both evolutionary runs produced good controllers, whose agents generally head straight for the food, even though they fairly often fail to take their own momentum into consideration when approaching the food, overshoot the food particle and have to turn back. (Figures 8 and 9).

Finally, the two convoluted controllers were evolved for 100 generations, and quickly generated very good solutions. The convoluted controller with a hidden layer narrowly outperformed the one without. Not only did good solutions evolve considerably faster than in the cases of the MLPs, but the best evolved convoluted controllers actually outperform the best evolved MLP controllers with a significant margin. As the computational capabilities of any of the convoluted controllers is a strict subset of the capabilities of the MLP with two hidden layers, this is slightly surprising, but can be explained with the extravagantly multidimensional search problems the MLPs present evolution with – even if a better solution exists it is improbable that it would be found in reasonable time.

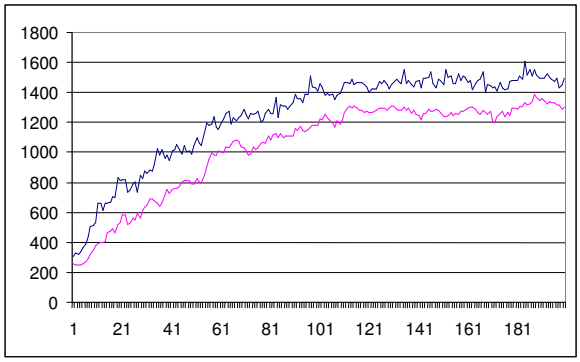


Figure 8: Evolving an MLP with one hidden layer for the Cellz game.

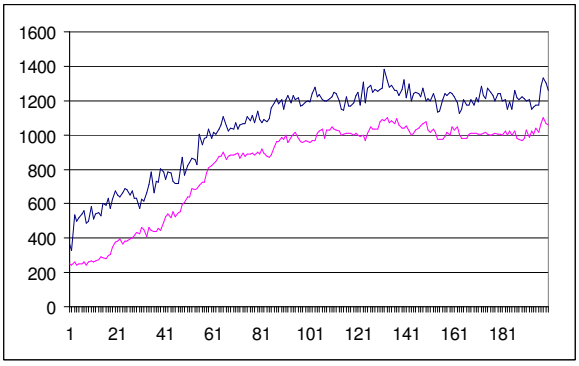


Figure 9: Evolving an MLP with two hidden layers for the Cellz game.

It is also interesting to note that the length of the neural path from sensor to actuator in the robots (that is, the number of hidden layers) seems to be of relatively small importance. (Figures 10 and 11) A comparison between controllers evolved in this paper, the winner of the GECCO 2004 Cellz contest, and the hand designed controllers mentioned in the original Cellz paper (Lucas 2004) is presented in Table 1. Note that the differences between the best three controllers are not statistically significant. The convoluted controller with one hidden layer is one of the best controllers found so far (though the convolutional aspect of the controller was hand-designed). Note that the winner of the GECCO 2004 contest was also partially hand-designed, as a neural implementation of the hand-coded sensor controller[1]. So far the purely evolved neural networks have been unable to compete with the networks that have been partially hand-crafted.

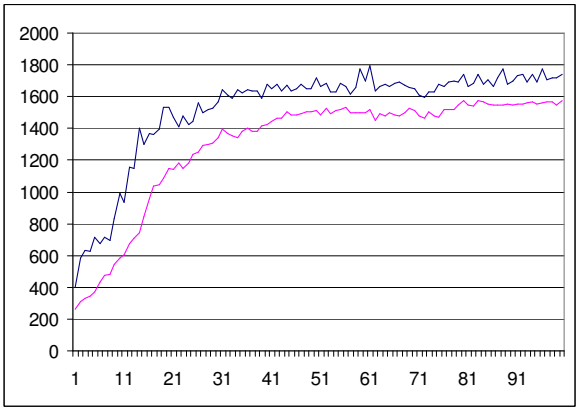


Figure 10: Evolving a perceptron-style convoluted controller for the Cellz game.

[1] See: http://cswww.essex.ac.uk/staff/sml/gecco/results/cellz/CellzResults.html

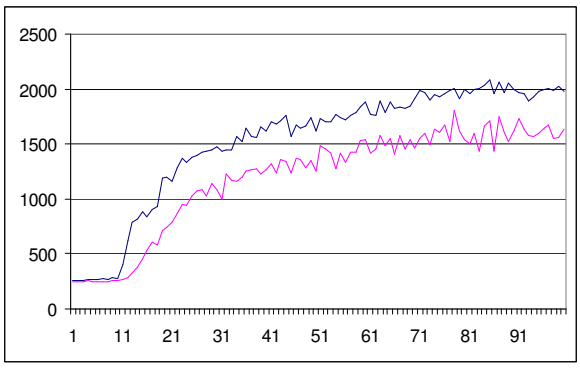


Figure 11: Evolving a convoluted MLP controller with one hidden layer for the Cellz game.

Controller	Fitness	S. E.
JB_Smart_Function_v1.1 (Winner of GECCO 2004 Cellz contest)	1966	20
Convoluted controller with one hidden layer	1934	18
Hand-coded sensor controller	1920	26
MLP with one hidden layer	1460	13
Hand-coded greedy controller	1327	24
MLP with two hidden layers	1225	14

Table 1: Mean fitness and standard errors over 100 game runs for controllers mentioned in this paper in comparison to other noteworthy Cellz controllers.

4 Conclusions

Our results clearly show that hard-coding modularity into neurocontrollers can increase evolvability significantly, at least when agents show symmetry, as they do in many computer games and robotics applications. They also show that certain types of modularity perform better than others, depending on the task at hand. Adding hidden neural layers might either increase or decrease evolvability, depending on the task. As neural encodings that intend to let modularity emerge always have certain biases, these results need to be taken into account when designing such an encoding.

We have also seen that the performance of the different architectures on the fitness approximation task are qualitatively comparable to the results of the same networks on the full Cellz task, e.g. the MLP with two hidden layers evolves more slowly than the MLP with only one, and the convoluted networks outperform monolithic networks. This suggests that this much simpler task can be used to further investigate the merits of different network architectures for Cellz; in particular, it might be possible to evolve network architectures for this simpler task, and later re-evolve the connection strengths of the evolved architectures for the Cellz task. The advantage of using the simpler task as a test-bed is that it is around 1,000 times faster to compute. This method can probably be used for other games as well.

The research described here is part of the first author's doctoral project investigating the role of modularity in artificial evolution; previous work on evolving layered structures was reported in Togelius (2004). In the future, we plan to extend this approach to more complicated tasks and input representations, such as first-person games with visual input. Eventually, we aim towards using the results of those studies as a requirements specification in the creation of new representations with which to evolve modular systems.

Bibliography

Bongard, J. C. (2003): *Incremental Approaches to the Combined Evolution of a Robot's Body and Brain*. PhD dissertation, University of Zurich.

Bullinaria, J. A. (2002): *To Modularize or Not To Modularize?* Proceedings of the 2002 U.K Workshop on Computational Intelligence: UKCI-02.

Calabretta, R., Nolfi, S., Parisi, D., Wagner, G. P. (2000): *Duplication of modules facilitates functional specialization.* Artificial Life, 6(1), 69-84.

Calabretta, R, Di Ferdinando, A. D., Wagner, G. P., Parisi, D. (2003): *What does it to take to evolve behaviorally complex organisms?* Biosystems 69(2-3): 245-62.

Gruau, F. (1993): *Genetic Synthesis of Modular Neural Networks*. In Forrest, S. (Ed.): Proceedings of Fifth International Conference on Genetic Algorithms. Morgan Kaufmann, 318-325.

Hornby, G. S., Lipson, H., Pollack, J. B. (2001): *Evolution of Generative Design Systems for Modular Physical Robots.* IEEE International Conference of Robotics and Automation.

M. Keijzer, J. J. Merelo, G. Romero, and M. Schoenauer (2001) *Evolving Objects: a general purpose evolutionary computation library*, Proceedings of Evolution Artificielle. Lecture Notes in Computer Science 2310, 231-244.

Kodjabachian, J. and Meyer, J. A. (1995): *Evolution and development of control architectures in animats.* Robotics and Autonomous Systems, 16, 161-182.

Koza, J. R. (1994): *Genetic Programming II: Automatic Discovery of Reusable Programs.* Bradford Books.

Lindenmayer, A. (1968): *Mathematical models for cellular interaction in development: Parts I and II.* Journal of Theoretical Biology, 18, 280-299, 300-315.

Lucas, S. M. (2004): *Cellz: A Simple Dynamic Game for Testing Evolutionary Algorithms.* Proceedings of the Congress on Evolutionary Computation, 1007-1014.

Lucas, S.M. (2004), *Exploiting Reflection in Object Oriented Genetic Programming*, Proceedings of the European Conference on Genetic Programming, 369-378.

Schraudolph, N. N., Dayan, P., Sejnowski, T. J. (1993): *Temporal Difference Learning of Position Evaluation in the Game of Go.* In Cowan et al. (eds): Advances in Neural Information Processing 6. San Mateo, CA: Morgan Kaufmann, 817 – 824.

Stanley, K. O. and Miikkulainen, R. (2003): *A Taxonomy for Artificial Embryogeny*. Artificial Life, 9(2), 93-138.

Togelius, J. (2004): *Evolution of a subsumption architecture neurocontroller.* Journal of Intelligent and Fuzzy Systems 15(1), 15-20.

Vaughan, E. (2003): *Bilaterally symmetric segmented neural networks for multi-jointed arm articulation.* Unpublished paper, University of Sussex. Available at: http://www.droidlogic.com/sussex/adaptivesystems/Arm.pdf

Designing and Implementing e-Market Games

Maria Fasli
University of Essex
Wivenhoe Park
Colchester CO4 3SQ, UK
mfasli@essex.ac.uk

Michael Michalakopoulos
University of Essex
Wivenhoe Park
Colchester CO4 3SQ, UK
mmichag@essex.ac.uk

Abstract- Trading in electronic markets has been the focus of intense research over the last few years within the areas of Artificial Intelligence and Economics. This paper discusses the need for tools to support the design and implementation of electronic market games. Such games simulating real life problems can be used in order to conduct research on mechanism design, markets and strategic behaviour. To this end we present the e-Game platform which was developed to support the design, implementation and execution of market game scenarios involving auctions. How game development is aided is demonstrate with an example game.

1 Introduction

Trading in electronic markets has been the focus of intense research over the last few years. Within Artificial Intelligence there has been a mounting interest in deploying agent-based and multi-agent systems to automate commerce. Unlike "traditional" software, agents are personalised, semi-autonomous and continuously running entities. Software agents can be particularly useful in automating commercial transactions, negotiating for goods and services on behalf of their users reflecting their preferences and perhaps negotiation strategies.

One of the most popular ways of negotiating for goods and services is via auctions [6, 9]. Auctions constitute a general class of negotiation protocols in which price is determined dynamically by the auction participants. With the advent of the world wide web auctions have been particularly popular for both B2B and C2C negotiations. There are hundreds of websites that provide facilities for hosting on-line auctions for C2C commerce the most prominent being ebay [2]. Auctions have been used extensively to trade products and services ranging from holidays to computer spare parts. Constructing trading agents to take part in auctions for a single good is relatively simple and although software trading agents are not commonplace in online auctions today, there has been some effort to automate the bidding process. eBay for example have introduced proxy bidding to make this process more convenient and less time-consuming for buyers. A user enters the maximum amount she would be willing to pay for an item which is kept confidential from the other bidders and the seller. The eBay system then compares that maximum bid to those of the other bidders. The system places bids on behalf of the bidder by proxy by using only as much of their bid as is necessary to maintain their high bid position (or to meet the reserve price). The system will only bid up to the maximum amount entered by a bidder. Obviously, if another bidder has a higher maximum, then the bidder by proxy would be outbid. But if no other bidder submitted a higher maximum, then this bidder would win the item. Although simple, this is an attempt to use a simple agent to bid on behalf of the user. The advantages are that the user does not have to constantly be on-line checking the status of the auctions and her bid. However, individual customers or businesses may have to negotiate for a bundle of perhaps interrelated goods being auctioned in different auctions following different rules. Developing agents that can participate in simultaneous auctions offering complementary and substitutable goods is a complex task. The successful performance of a trading agent does not only depend on its strategy but on the strategy of the other agents as well. Designing efficient and effective bidding strategies is difficult since real world data about trading agents is difficult to obtain. This paper discusses the need for tools to support the design and implementation of electronic market games. Such games simulating real life problems can be used in order to conduct research on mechanism design, markets and strategic behaviour. To this end we present the e-Game platform which was developed to support the design, implementation and execution of market game scenarios involving auctions.

2 Motivation

Despite their increased popularity, the huge potential of agents and multi-agent systems in e-commerce hasn't been fully realised [13]. One of the difficulties is in testing trading agents and strategies in real life complex marketplaces. This is impractical and carries high risks. One approach to this problem is to run simulated market scenarios and test one's bidding strategies. However, the results of such experiments conducted by a single person may lack the touch of realism, since the bidding strategies explored may not necessarily reflect diversity in reality. This is the central idea and major motivation behind the International Trading Agent Competition (TAC) [5] which features artificial trading agents competing against each other in market-based scenarios. Currently two scenarios are available and run as part of the competition: the Classic Travel Agent Scenario game and the latest Supply Chain Management game. Ideally, we would like open-source platforms where market-based scenarios could be implemented and researchers have the opportunity to build and test their trading agents against those of others.

Designing and implementing electronic markets is a complex and intricate process [11]. Not all negotiation protocols may be appropriate for a given situation. Again simulated market games may offer the only way to actually approach this problem in a systematic way. Guided first by theory and following a detailed analysis of a given domain one can proceed to choose appropriate negotiation protocols and then design and implement a marketplace game and finally experiment to verify the appropriateness of the protocols used.

However, platforms or tools that would allow researchers and developers to design and implement their own marketplaces are currently lacking. The remainder of the paper discusses such a platform that provides the facilities for market games to be developed that involve a variety of auction protocols. The paper is structured as follows: The next section provides an introduction to auctions as negotiation protocols and discusses their characteristics and basic types. Next the main features of the e-Game platform are presented. Game development is discussed next and an example game is presented. The paper closes with the conclusions.

3 Auctions

Auctions constitute a general class of negotiation protocols in which price is determined dynamically by the auction participants. In auctions goods of undetermined quality can be sold. They can be used for the sale of a single item, or for multiple units of a homogeneous item as well as the sale of interrelated goods [1]. There are two main (self-) interested parties in an auction: The auctioneer who wants to sell goods at the highest possible price (or subcontract out tasks at the lowest possible price) who may be the owner of the good or service or a representative of the actual seller, and the bidders who want to buy goods at the lowest possible price (or get contracts at the highest possible price). Bidders bid in order to acquire an item for personal use, for resale or commercial use. The task of determining the value of a commodity is transferred from the vendor to the market, thus leading to a fairer allocation of resources based on who values them most. The process of an auction involves the following typical stages:

• Registration. Buyers and sellers register with an auction house.

• Bidding Phase. The participants bid according to the rules of the particular auction protocol used. A bid indicates a bound on the bidders' willingness to buy or to sell a good.

• Bid Processing. The auctioneer checks the validity of a bid according to the rules of the auction used and updates its database (manual or electronic).

• Price quote generation. The auction house via the auctioneer or by other means may provide information about the status of the bids. A *bid quote* is the highest outstanding effective offer to buy, that is the current price that a bidder has to exceed in order to acquire the good. An *ask quote* is the lowest outstanding effective offer to sell, that is if a seller wants to trade, it will have to be willing to accept less than that price.

• Clearance (or Matching). Through clearance, buyers and sellers are matched and the transaction price is set. What is termed clearing price is the final transaction price.

• Transaction Phase. The transaction actually takes place: the buyer bidder pays and receives the good/service.

The core three activities that characterise and differentiate auction protocols are bidding, price quote generation and clearance [15].

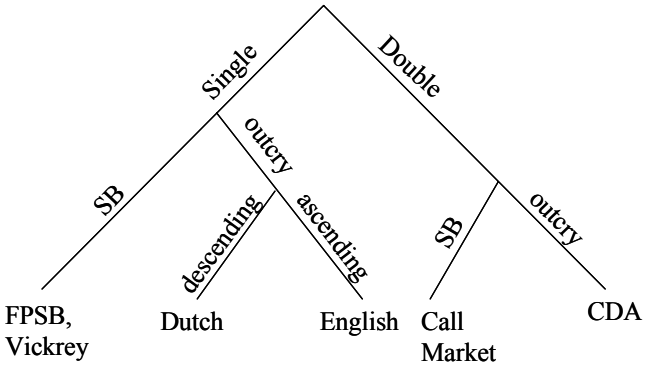


Figure 1: Classic auction classification [15].

3.1 Auction Classification

Traditional auction protocols are generally classified depending on whether multiple buyers and sellers are allowed (single/double), the bidders have information on each other's bids (open/closed), the flow of prices (ascending/descending) and how the bidders form their valuations (private/common/correlated value) [15], Figure 1. Among the most well known auctions are the English, Dutch, FPSB, Vickrey and CDA. The English auction is an open-outcry and ascending-price auction which begins with the auctioneer announcing the lowest possible price (which can be a reserved price). Bidders are free to raise their bid and the auction proceeds to successively higher bids. When there are no more raises the winner of the auction is the bidder of the highest bid. The Dutch auction is an open and descending-price auction. The auctioneer announces a very high opening bid and then keeps lowering the price until a bidder accepts it. The FPSB (First-Price Sealed-Bid) and Vickrey (also known as uniform second-price sealed-bid) auctions have two distinctive phases: the bidding phase in which participants submit their bids, and the resolution phase in which the bids are opened and the winner is determined. The bid that each bidder submits in both auctions is sealed. In the FPSB auction the highest bidder wins and pays the amount of its bid, while in the Vickrey auction the higher bidder wins, but pays the second-highest bid. Finally, the CDA (Continuous Double Auction) is a general auction mechanism that is used in commodity and stock markets [1]. Multiple buyers and sellers can participate in such auctions and a number of clearing mechanisms can be used. For instance in the stock market clearing takes place as soon as bids are matched.

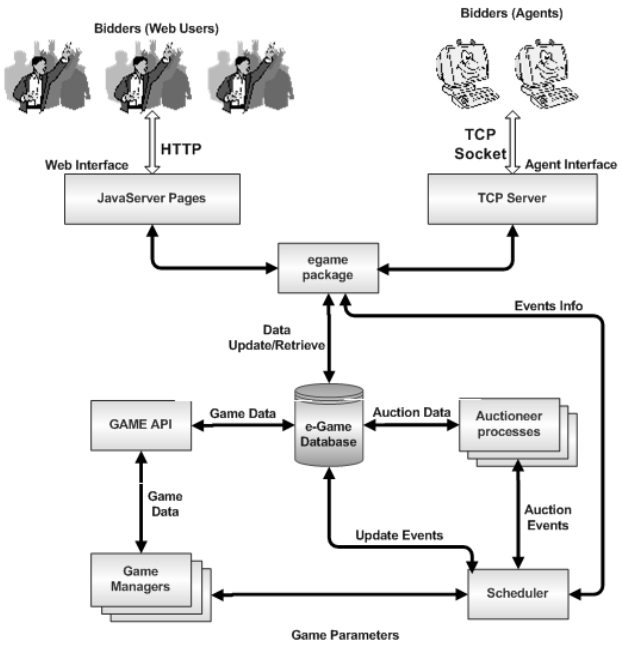


Figure 2: The architecture of e-Game.

4 The e-Game Platform

e-Game (**e**lectronic **G**eneric **a**uction **m**arketplac**e**) is a configurable auction server in which participants may be humans or software agents. It is a generic platform that can be extended to support many types of auctions. Additionally, e-Game is designed to support auction games analogous to those found in the Trading Agent Competition [5] which can be designed and developed by third parties.

e-Game is based on a modular architecture separating the interface layer from the data processing modules and the database (Figure 2). The database is used to model the auction space (parameters such as auction type, startup,closing time, clearing rules), as well as the user data (registration, user bids, auctions the user created). It also stores information regarding the game (which game started which auctions, ids of players, scores). It provides fault tolerance, in that the events queue can be reconstructed if there is a problem with the auctioneers or the scheduler. Its modularity and expandability are further strengthened by adopting the object-oriented paradigm for its components. The platform has been developed in JAVA [4] in order to gain the benefits of platform independence and transparency in networking communication and object handling. Some of the main components and features of the e-Game platform are described next.

4.1 Interfaces

e-Game is designed to allow for both web users and software agents to participate and as such has two interfaces to the outer world: the *Web* and the *Agent* interfaces similarly to [14]. The *Web* interface provides a series of functions such as registration, creation of new auctions using a variety of parameters (auction type, information revealing, clearing and closing mechanism), bid submission, search facilities according to a series of attributes (auctioned item, status of auction, user's participation in an auction) and viewing the user's profile (change personal information, view previous auction information).

The *Agent* interface provides agent developers the same functionality with the *Web* interface with respect to auctions plus an additional set of commands for participating in market games. Agents connect to the *Agent* Interface using the TCP protocol and submit their commands using a simple string format which complies with the syntax of the HTTP query string: (command: parameter1=value1\&..parameterN=valueN). This facilitates participation in individual auctions as well as auction-based market exercises. Such commands allow agents for instance, to retrieve information about an auction such as start up and closing time, submit bids to auctions or obtain information about the state of a submitted bid. The full set of commands is available for agent developers. For every command an agent submits, the e-Game platform returns an appropriate result together with a command status value that indicates the outcome of the command.

The majority of the functions that the web users and the agents perform are identical and are implemented in a single package which is shared between the two interfaces. The advantage of such an approach is that both components can be easily extended in the future with the minimum effort in terms of implementation and testing.

4.2 Scheduler

The *Scheduler* runs in parallel with the *Web* and the *Agent* interfaces and is responsible for starting up auctions, games and passing bids to the appropriate *Auctioneer* processes. The auctions can be scheduled by web users or by a specific *GameManager* handler according to a market scenario's rules, whereas games are scheduled by web users. The *Scheduler* provides a fault-tolerant behaviour, since if it is brought off-line for some reason, when it is restarted it can reconstruct its lists of events by looking into the database for pending events.

4.3 Auction Support

Currently e-Game supports a wide range of auction types (Table 1). These basic types of auctions can be further refined by allowing the user to define additional parameters that include [14]:

• Price quote calculation: Upon arrival of new bids, at fixed periods or after a certain period of buy/sell inactivity.

• Auction closure: At a designated date and time or after a period of buy/sell inactivity.

• Intermediate clearing: Upon arrival of new bids, at fixed periods or after a certain period of buy/sell inactivity.

• Information revelation: Whether the price quotes and the closing time are revealed or not.

Such a parameterization can have an effect on the users' or agents' behaviour, since they modify the basic functionality of an auction as well as change the amount of informa-

Auction Type	Sellers	Units	Buyers
English	1	1	Many
Vickrey	1	1	Many
FPSB	1	1	Many
Dutch	1	1	Many
Mth Price	1	Many	Many
Continuous Single	1	Many	Many
Double Auction	Many	Many	Many

Table 1: Auctions supported by e-Game.

tion revealed regarding bids and closing time.

The actual auctions are carried out by the *Auctioneer* classes. Each *Auctioneer* is started by the *Scheduler* and implements a specific auction protocol. Therefore, there are as many *Auctioneer* classes, as the number of auction protocols supported by e-Game. Since all the *Auctioneer*s perform alike actions, they inherit the basic behaviour and state from a *BasicAuctioneer* class. Once an auction starts, the *BasicAuctioneer* class is responsible for generating the auction's events based on the choices the user made during the setup. At a given moment in time there may be a number of similar or different classes of active *Auctioneers* within the e-Game platform, each one handling a different auction.

The implementation of all auctions is based on the Mth and M+1st price clearing rules. The clearing rules for the auctions are just special cases of the application of the Mth and M+1st clearing rules. However, much depends on the initial set up performed by the user. For instance to achieve a chronological order clearing in an auction one can set the intermediate clearing periods to be as soon as a bid is received and thus the parameterized auctioneer will attempt to perform intermediate clearings upon the arrival of a new bid. As in classical chronological matching, if a portion of the bid cannot transact, then it remains as a standing offer to buy (or sell). The implementation of the Mth and M+1st clearing rules uses ordered lists.

5 Game Development

Apart from the support for agents and web users regarding auction operations, e-Game's main feature is that it supports the design, development and execution of different market scenarios or games that involve auctions analogous to [5]. Since the e-Game platform allows for web users as well as agents to participate in auctions, it is feasible for everyone to participate in a market game as well. This may sometimes be desirable when comparisons between humans and software agents need to be drawn with respect to laying out a strategy to maximize utility.

Prior to designing a new game, a developer should have a good understanding of the auction types supported by e-Game. The web site offers a glossary that describes the different types of auctions and their setup parameters. If the participants of a game are web users, the web site already provides the functionality to participate in the auctions involved in the game. However in most cases, one is interested in designing a game that will be played by software agents. When designing a new game, the developer should have in mind the following key points:

• Realism: The more realistic the auction scenario is, the more comprehensible it will be to the people who wish to develop agents to participate in it. Ideally, the auction-based market scenario should describe a realistic situation, which may be encountered as part of everyday life (holiday planning, buying interrelated goods, scheduling of resources).

• Strategic challenge: the game should present difficult strategic challenges that artificial trading agents may have to face.

• Efficient use of e-Game: Ideally, the game should not be based on a single type of auction: the game developer has a wide range of auctions to choose from. The usage of different types of auctions raises more challenges for the participants, who will have to lay out different strategies depending on the auction type and the amount of information that is revealed at run time.

• Fairness with regards to specifications: In an auction game different agents with most probably different strategies will try to accomplish a certain goal. Though the goal will be similar in concept for every agent (for example, assemble one computer by combining different resources), the achievement of individual goals will give different utilities for each agent. The developer should consider specifications that give everyone the opportunity to come up with the maximum utility.

• Fairness with regards to agent design: Agents with a more appropriate bidding strategy should manage to achieve a better outcome, than agents with "naive" and "simplistic" strategies. When "naive" strategies actually represent corresponding actions in real life (for which people can actually reason), this prerequisite does not hold. However, the purpose of introducing this key point is that there should not be any flaws in the market game that would allow non-realistic strategies to outperform rational ones.

• Computational efficiency: The handler of an auction game should have the ability to operate in a real-time manner. Normally, there should not be any algorithms that take a long time to run, especially in critical time points. It is acceptable to have such algorithms at the end of the game when an optimal allocation may be required and scores have to be calculated.

It is the developer's responsibility to ensure that a game is realistic, fair and non-trivial. Transparency in e-Game is enforced by publishing the logfiles and outcome of auctions.

5.1 Game Implementation

In terms of implementation, game development is simplified by extending the *GenericGame* class as provided by e-Game. As a consequence, developers are expected to be familiar with the Java programming language and provide the implementation for a number of processes in the form of methods that are part of a game. In general a game involves the following phases:

• Set up. In this initial phase of the game the developer sets up the game environment and specifies the auctions that

are to be run and the various parameters for these. The *startupGame* method needs to be implemented to provide this information.

• Generation of parameters. Since a marketplace is simulated in a game, agents that will be participating in this will have some objective such as for instance to acquire goods on behalf of some clients while at the same time maximizing their profit. Client preferences, initial endowments, other information regarding the state of the market, scheduled auctions, availability of goods and other resources and any other data the developer wants to communicate to each agent in the beginning of the game can be done through the *generateParameters* method. The agents are responsible for retrieving this information and then using it in their strategy.

• Execution. In this phase the actual auctions that are part of the game run, agents are allowed to submit bids according to the auctions rules and information is revealed according to the parameters specified in the *startupGame* method. e-Game runs the auctions and handles the messages that need to be exchanged with the agents. The agents are responsible for retrieving information on the status of their bids and making decisions accordingly during this phase of the game.

• Score calculation. An agent's success is usually measured in terms of a utility function. This obviously depends on the strategy that an agent is using, however the utility function that measures how successful an agent is needs to be provided by the developer. The *calculateScores* method allows the developer to look into the auctions' results and calculate the utility for each individual agent.

• Close down. This is the final phase of a game and it is automatically called at the end. This also allows for any necessary clean up code to be performed.

Game developers do not have the ability to directly call methods from the other platform's components or gain direct access to the e-Game database. They can only use the methods that the *GenericGame* class offers them which allow them to perform a wide range of tasks such as:

• Retrieve the IDs of the participants in the current game.

• Schedule any type of auction according to the rules of the game (for example, set a reserved price or choose not to reveal the closing time).

• Associate certain generated parameters with an agent.

• Retrieve information for the auctions associated with this game, such as their IDs, closing time or current winning bids (quantity and price) and their owners.

• Associate a score with a certain agent.

• Submit bids for a certain auction, for example when there is a Descending Price auction and the *GameManager* is actually the Seller.

The developer can also provide two more classes when developing a new game:

• A sample agent that other users can extend according to their needs and that will also be used as an agent that will fill one of the available player slots when starting a game.

• An applet that can be loaded at runtime, so users can monitor the progress of a game.

The decision whether to design and implement a sample Agent and an applet is left to the developer. Once the game and the applet have been developed, it is very easy to integrate it with the e-Game platform. The developer simply needs to provide the class file, together with an XML file that describes general properties of the game such as the name of the game, the name of the implemented classes (game, applet), a short description for the game, its duration in seconds, the number of allowed participants and the resources (items) that this game will use.

Information is parsed by the *GameImporter* application and is stored in the database. Web users can browse the web site and by following the link "Agent Games" they can view all the installed types of games, together with appropriate links to schedule new instances of games, monitor the progress of a current game and view previous scores. When a new game is scheduled, the *Scheduler* receives the corresponding event and at the appropriate time, loads the user defined *GameManager* class. Since the name of the user defined class is not known until runtime, the *Scheduler* uses Java's Reflection mechanism to dynamically load it. At the end of a game, users can view their score and the resources, they managed to obtain, as well as the initial parameters.

5.2 The Computer Market Game

To provide the reader with a feeling of the capabilities of the e-Game platform regarding game development, this section discusses the implementation of a simple game. In the Computer Market Game (CMG), which lasts nine minutes, each of the six agents participating is a supplier agent whose task is to assemble PCs for its five clients. For simplicity, there are only three types of parts that are necessary to make up a properly working PC: a motherboard, a case and a monitor. There are three types of motherboards, each one bundled with CPUs of 1.0 GHz (MB1), 1.5 GHz (MB2) and 2.0 GHz (MB3). Cases come in two different types, one packed with a DVD player (C1) and another one with a DVD/RW (C2) combo drive. For the purpose of this game there is only one monitor type.

At the beginning of the game the agents receive their clients' preferences in terms of a bonus value for upgrading to a better motherboard (MB2 or MB3) and a better case (C2). So for instance, in a particular game instance an agent participating in the game may receive the preferences illustrated in Table 2. e-Game generates values in the following ranges for these preferences MB2 = [100..150], MB3=[150..200], C2=[200..300]. Hence, in the above example client 2 of agent 1 has a bonus of 130 for obtaining a 1.5 GHz motherboard and a bonus of 200 for upgrading to a 2.0 GHz motherboard. For upgrading to the better case with the DVD/RW drive the customer is offering 300.

Each component is available in a limited quantity and is traded in different auctions. Table 3 summarizes the quantities of the items and the respective auctions. Figure 3 illustrates how the auctions for the various items are scheduled during the nine minutes of the game. The dotted part of the lines indicate that the auctions may close anytime in that period, but the exact closing time is not revealed to the agents.

An agent's success in this game depends on the satisfac-

Agent	Client	MB2	MB3	C2
1	1	110	150	220
1	2	130	200	300
1	3	120	170	250
1	4	150	184	296
1	5	100	156	201

Table 2: Example of client preferences.

Component	Quantity	Auction
MB1	17	Mth Price
MB2	8	Mth Price
MB3	5	Mth Price
C1	20	Mth Price
C2	10	Mth Price
M	30	Continuous single seller

Table 3: Component availability and auctions.

tion of its clients. An individual client's utility (*CU*) is:

CU = 1000 + Motherboard Bonus + Case Bonus

For each client that has been allocated a completed PC the agent gets 1000 monetary units plus any bonus for upgrading to a better motherboard or case. If no completed PC is allocated to a client, then this utility is 0. An agent's utility function (*AU*) is the sum of all the individual client utilities minus the expenses incurred:

AU = (CU1+CU2+CU3+CU4+CU5) - Expenses

The agent's strategy should be focused on providing a PC to each one of its clients or to as many as possible, while at the same time trying to minimize costs. There are obvious interdependencies between goods, as a fully assembled PC requires three components. In creating a strategy for this game, one has to take into account that the availability of goods is limited, prices in auctions may vary, auctions close at different times and therefore one may have to switch to a different auction if they fail to acquire a certain good, and customers give different bonuses for upgrading to a better specification. Moreover, an agent's success, does not only depend on its own strategy, but that of the other agents too. However, an agent does not have any means of knowing the identity of the other players or monitoring their bids.

In order to implement the above scenario the developer would simply provide an XML file describing the general properties of the game (names of game and applet classes, number of players, duration of the game, auctioned resources), together with the appropriate classes. The specific

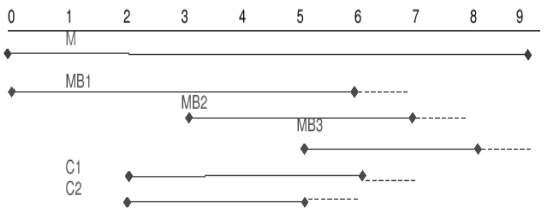


Figure 3: Auction schedule in CMG.

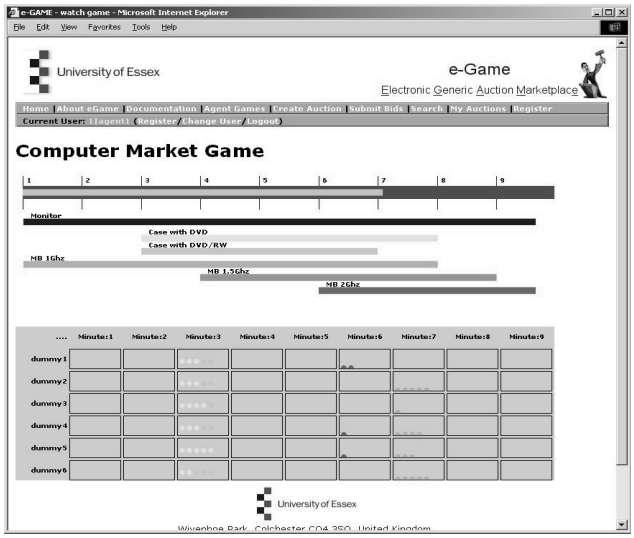


Figure 4: The CMG applet.

GameManager would simply schedule the auctions at the beginning of the game by implementing the method *startupGame* and using the *scheduleAuction* methods. The next step would be to generate the parameters that each agent receives by implementing the method *generateParameters*. The utility of each agent would be calculated at the end of the game by examining the winning bids of each auction. Finally, any language resources that the developer used would be freed-up in the *closeGame* method. In addition, the developer may choose to write an applet so that web users can view the progress of their agent. Such an applet for this simple game is illustrated in Figure 4.

6 Conclusions

The deployment of intelligent agents in commerce would bring significant advantages: it would eliminate the need for continuous user involvement, reduce the negotiation time and transaction costs and potentially provide more efficient allocations for all parties involved. This paper discussed the need for tools to support the design and development of electronic market games that could be used to conduct research on mechanism design, negotiation protocols and strategies. To this end we presented the e-Game platform. e-Game is a generic auction platform that allows web users as well as software agents to participate in electronic auctions and auction-based market games. The most important feature of e-Game and what distinguishes it from other efforts such as [12, 8, 14, 5, 3] is that it provides independent developers the infrastructure for developing complex market-based scenarios and conducting experiments on bidding strategies. Unlike [3] for instance, e-Game does not simply provide the facilities for scheduling, running or conducting experiments on the use of strategies in standalone auctions. Individual users/developers can write their own *GameManager* and define complete market scenarios together with accompanying applets and servlets that can be inserted and run on top of the e-Game infrastruc-

ture. The development of these modules is completely separate from the rest of the system. By making use of existing classes and documentation, users can develop new games quickly. The process of integrating user-developed scenarios is fully automated. This is feasible by introducing appropriate built-in classes for each module separately. By making use of Java's Reflection it is then possible to look into a newly introduced class and access its methods and data members.

Moreover, e-Game provides the facility of having humans play against agents. This may be useful in designing strategies and examining strategy efficiency and complexity. One could study the benefits from using agents in complex markets in a systematic way. For example, in a simple game-scenario, a human may do better than an agent, but it would be interesting to pit humans against software agents and study what happens when the scenario gets more complicated (penalty points, allocation problems etc.).

Apart from using the platform for designing and running complex market experiments, it can also be used in teaching issues in negotiation protocols, auctions and strategies. Students can have a hands-on experience with a number of negotiation protocols and put into practice the principles taught.

Possible future extensions include the introduction of more complicated auction formats such as multi-attribute and combinatorial auctions [7].

Another possible future extension includes replacing the communication and bidding languages with a standard ACL language (FIPA ACL) similarly to [16]. We believe that such an action contributes to agents' standardization and establishment of automated electronic markets. e-Game could also be extended so that the *Agent Interface* provides the same functionality as the *Web Interface*. Therefore, software agents could potentially set up auctions, and search for auctions. For instance, an agent could search for auctions selling specific items and then decide on whether or not to participate as well as its strategy. Although this may not be necessary at the current stage it is certainly feasible. Finally, we are considering the possibility of introducing a rule-based specification such as [10] in order to describe complex market-based scenarios.

Bibliography

[1] Auctions. http://www.agorics.com/Library/auctions.html.

[2] Ebay. http://www.ebay.com/.

[3] JASA: Java Auction Simulator API. http://www.csc.liv.ac.uk/∼sphelps/jasa/.

[4] Java technology. http://www.java.sun.com.

[5] Trading Agent Competition. http://www.sics.se/tac/.

[6] Y. Bakos. The emerging role of electronic marketplaces on the internet. *Communications of the ACM*, 41(8):35–42, 1998.

[7] M. Bichler. A roadmap to auction-based negotiation protocols for electronic commerce. In *Hawaii International Conference on Systems Sciences (HICSS-33)*, 2000.

[8] A. Chavez and P. Maes. Kasbah: An agent marketplace for buying and selling goods. In *First International Conference on the Practical Application of Intelligent Agents and Multi-Agent Technology (PAAM'96)*, pages 75–90, 1996.

[9] M. Kumar and S. Feldman. Internet auctions. In *Proceedings of the 3rd USENIX Workshop on Electronic Commerce*, pages 49–60, 1998.

[10] K.M. Lochner and M.P. Wellman. Rule-based specifications of auction mechanisms. In *Third International Joint Conference on Autonomous Agents and Multiagent Systems Conference (AAMAS)*, pages 818–825, 2004.

[11] D. Neumann and C. Weinhardt. Domain-independent enegotiation design: Prospects, methods, and challenges. In *13th International Workshop on Database and Expert Systems Applications (DEXA'02)*, pages 680–686, 2002.

[12] M. Tsvetovatyy, M. Gini, B. Mobasher, and Z. Wieckowski. Magma: An agent-based virtual market for electronic commerce. *Journal of Applied Artificial Intelligence*, 6, 1997.

[13] N. Vulkan. Economic implications of agent technology and e-commerce. *The Economic Journal*, 453:67–90, 1999.

[14] P. Wurman, M. Wellman, and W. Walsh. The Michigan Internet AuctionBot: A configurable auction server for human and software agents. In *Proceedings of the Autonomous Agents Conference*, pages 301–308, 1998.

[15] P. R. Wurman, M. P. Wellman, and W. E. Walsh. A parameterization of the auction design space. *Games and Economic Behavior*, 35(1):304–338, 2001.

[16] Y. Zou, T. Finin, L. Ding, H. Chen, and Pan R. Taga: Trading agent competition in agentcities. In *Workshop on Trading Agent Design and Analysis held in conjunction with the Eighteenth International Joint Conference on Artificial Intelligence (IJCAI)*, 2003.

Dealing with parameterized actions in behavior testing of commercial computer games

Jörg Denzinger, Kevin Loose
Department of Computer Science
University of Calgary
Calgary, Canada
{denzinge, kjl}@cpsc.ucalgary.ca

Darryl Gates, John Buchanan
Electronic Arts Inc.
4330 Sanderson Way
Burnaby, Canada
{dgates, juancho}@ea.com

Abstract- We present a method that enhances evolutionary behavior testing of commercial computer games, as introduced in [CD+04], to deal with parameterized actions. The basic idea is to use a layered approach. On one layer, we evolve good parameter value combinations for the parameters of a parameterized action. On the higher layer, we evolve at the same time good action sequences that make use of the value combinations and that try to bring the game in a wanted (or unwanted) state. We used this approach to test cornerkicks in the FIFA-99 game. We were able to evolve many parameter-value-action sequence combinations that scored a goal or resulted in a penalty shot, some of which are very short or represent a rather unintelligent behavior of the players guided by the computer opponent.

1 Introduction

In recent years, many AI researchers have turned to using commercial computer games as "AI friendly" testbeds for AI concepts, like improved path finding algorithms and concepts for learning. Often, the guiding idea is to make the used game smarter to beat the human players, which naturally shows off the capabilities of a particular AI concept and scores points in AI's constant struggle to develop intelligent programs. But we see games incorporating such AI concepts rather seldom, in fact the current commercial computer games have to be seen as rather behind what AI researchers have accomplished with available source code of older games (see, for example, [vR03]). The reason for this lies in the priorities game companies have: their games are supposed to keep a human player or a group of human players entertained, which requires them to walk a very fine line between making it too easy and too difficult to win. Therefore, techniques that make games less predictable (for the game designer) have to be seen as a risk, especially given the current state of testing methods for games. Unfortunately, the aim of many AI methods is exactly that: to make games less predictable and more resembling what humans would do.

But we think that AI itself can come to the rescue: by using AI methods to improve the testing of commercial computer games we can not only improve the current situation with regard to testing games within the short time spans the industry can afford, we can also open the way for incorporating more AI methods into games, allowing for more complex game behavior that can even include aspects of learning and adaptation to the human player. We see as a key method for improving game testing the use of concepts from learning of (cooperative) behavior for single agents or agent groups, like [DF96], to find game player behavior that brings the game into unwanted game states or results in unwanted game behavior in general. Learning of behavior can also be used to find various game player action sequences that bring the game into states that match conditions a game designer has for certain special game behaviors, which also helps improving the testing of such ideas of the designers.

In [CD+04], we presented an evolutionary learning approach that allowed us to evolve action sequences, which lead with more than a certain high probability, to scoring a goal in the game FIFA-99 when executed. Evolutionary learning with its ability to achieve something similar to human intuition due to the mixture of randomness with exploitation of knowledge about individuals is very well suited to imitate game player behavior and the methods some players use to find *sweet spots* in games. In contrast to a search using more conventional methods like and-or-trees or -graphs, an evolutionary search can deal with the indeterminism that many commercial computer games rely on to offer different game playing experiences every time the game is played in a natural manner.

Our approach of [CD+04] used a fixed set of actions on which the action sequences to be evolved were built. Most sport games have a rather limited number of actions that are "activated" by pressing buttons and that are interpreted by the game in a manner appropriate to the situation in which the action is activated. But in many games there are at least some actions that require the game player to provide some parameter-values for the action (again, using buttons in a special manner). Examples of such actions are a pitch in a baseball game or a cornerkick in a soccer game. [CD+04] did not cover such parameterized actions.

In this paper we present an extension of the method of [CD+04] that allows to deal with special parameterized actions occurring at specific points in an action sequence. The basic idea (inspired by [St00]'s idea of layered learning) is to use a two-layered evolutionary algorithm, one layer evolving action sequences and one layer evolving parameter combinations for the parameterized action. Naturally, the fitness of a parameter layer individual has to depend on the fitness that this individual produces when applied in an individual of the action sequence layer, in fact it depends on the fitness of many individuals on the sequence layer and vice versa. We were also inspired by concepts from

co-evolving cooperative agents, see [PJ00] and [WS03], although we make additionally use of the fact that the fitness on the action sequence layer is based on an evaluation of the game state after each action and therefore we can emphasize also the evaluation of the outcome of the special action.

We evaluated our extension to special actions within the FIFA-99 game, again. The special action we targeted was the cornerkick and our experiments show that the old soccer saying *a corner is half a goal*, while not true anymore in todays professional soccer, is definitely true within FIFA-99. Many of the behaviors we found (with regard to scoring) are rather acceptable, but we also found some very short sequences that always scored and some rather "stupid" opponent behavior consistently leading to penalty shots awarded to the attackers.

This paper is organized as follows: After this introduction, we re-introduce the basic ideas of [CD+04], namely our abstract view on the interaction of human players with a game in Section 2 and the evolutionary learning of action sequences for game testing in Section 3. In Section 4, we introduce our extension of the learning approach to allow for special parameterized actions. In Section 5, we present our case study within FIFA-99 and in Section 6 we conclude with some remarks on future work.

2 An interaction behavior based view on commercial computer games

On an abstract level, the interaction of a human game player with a commercial computer game can be seen as a stream of inputs by the player to the game. These inputs have some influence on the state of the game, although often there are other influences from inside and sometimes outside of the game. If we have several human players, then the other players are such an outside influence, since the game combines all the input streams and thus produces a sequence of game states that describe how a particular game run was played.

More precisely, we see the inputs by a game player as a sequence of action commands out of a set Act of possible actions. Often it is possible to identify subsets of Act that describe the same general action inside the game, only called by the game with different parameter values (like a pitch in baseball, where the parameters determine where the player wants to aim the ball at, or a penalty shot in soccer). We call such a subset a parameterized action (described by the general action and the parameter values, obviously) and all interactions of the player with the game that are needed to provide general action and parameter values are replaced by the parameterized action in our representation of the interaction as action sequence from now on. By $PAct$ we refer to the set of parameterized actions.

The game is always in a game state out of a set S of possible states. Then playing the game means producing an action sequence

$$a_1, ..., a_m;\ a_i \in Act \cup PAct$$

and this action sequence produces a game state sequence

$$s_1, ..., s_m;\ s_i \in S.$$

Naturally, the game has to be in a state s_0 already, before the action sequence starts. This start state can be the general start state of the game, but many games allow to start from a saved state, also.

What state the game is in has some influence on the actions that a player can perform. For a situation s, by $Act(s) \subseteq Act \cup PAct$ we define the set of actions that are possible to be performed in s (are recognized by the game in s). Especially the parameterized actions in most games are only possible to be performed in a limited number of "special" states, since providing the parameter values for such an action is often rather awkward (what usually is *not* done is to allow a user to enter a numerical parameter value directly; instead some graphical interface has to be used that often requires quite some skill to handle). The "special" states recognizing parameterized actions often allow only for one particular parameterized action.

For the learning of action sequences that bring the game into a state fulfilling some wanted (or, for testing purposes, unwanted) condition –we call this also a wanted (unwanted) behavior– the fact that not every action can be executed in every state poses a certain problem. But the fact that $Act(s)$ is finite for each s (if we count a parameterized action, regardless of the used parameter values, as just one action) allows us to order the elements in $Act(s)$ and to refer to an action by its index in the ordering (which is a natural number). And if we use the index number of an action to indicate it in an action sequence, we can guarantee that in every game state there will be an action associated with any index number. If such a number ind is smaller than $|Act(s)|$ in situation s, then obviously it refers to a legal action (although to different actions in different states). If $ind > |Act(s)|$, then we simply execute the action indicated by ind modulo $|Act(s)|$ which is a legal action, again.

If we look at action sequences and the game state sequences they produce, then in most games the same action sequence does not result in the same state sequence all the time (especially if we use indexes to indicate actions, as described in the last paragraph). This is due to the fact that the successor state s' of a game state s is not solely based on the action chosen by the game player and s. In addition to other players, random effects are incorporated by game designers in order to keep games interesting and these random effects influence what s' will be. For example, wind is often realized as a random effect that influences a pitch or shot in a sports game or the aim of an archer in a fantasy role playing game.

It has to be noted that everything we have observed so far with regard to keeping a human player entertained makes games less predictable and hence more difficult to test. We have to run action sequences several times to see if they have an intended effect, for example. But fortunately there are also "shortcomings" of human players that help with the testing effort. For example, producing an action sequence that totally solves or wins a game, i.e. starting from the game start and going on until the winning condition is fulfilled, is very difficult, since we are talking about a sequence that might have more than tens of thousands of actions. But

fortunately human players also cannot produce such a sequence in one try, respectively without periods in which they can relax. Therefore commercial computer games allow the user to save game states, as already mentioned, and structure the game into subgoals, like minor quests in role playing games or one offensive in a soccer or hockey game. This means that for testing purposes smaller action sequences can be considered, although it might be necessary to look at several ways how these sequences are enabled.

3 Evolutionary behavior testing without parameterized actions

In [CD+04], we used the interaction behavior based view on computer games from the last section to develop an approach to testing of computer games that aims at evolving action sequences that bring the game into a state fulfilling a given condition. This condition might either be a condition that the game designer does not want to occur at all, a condition that should occur only in certain circumstances and the designer wants to make sure that there is no action sequence that brings us there without producing the circumstances, or a condition that describes a game state from which the game designer or tester wants to launch particular targeted tests. The approach in [CD+04] did not deal with parameterized actions. In the following, we will briefly describe this original approach, on which we will base the extension that we describe in the next section.

The basic idea of [CD+04] was to use a Genetic Algorithm working on action sequences as individuals to produce a game behavior fulfilling a condition $\mathcal{G}_{unwanted}$. Due to the indeterminism incorporated in most games, this condition has to be fulfilled in more than a predefined percentage of evaluation runs of an action sequence. The crucial part of this idea is the definition of a fitness function that guides the evolutionary process towards action sequences that fulfill $\mathcal{G}_{unwanted}$. Following [DF96], the fitness of an individual is based on evaluating each game state produced during an evaluation run of the action sequence (as input to the target game), summing these evaluations up and performing several evaluation runs (starting from the same start state) and summing up the evaluations of all these runs.

The basic idea of an evolutionary algorithm is to work on a set of solution candidates (called individuals) and use so-called Genetic Operators to create out of the current individuals new individuals. After several new individuals are created, the original set and the new individuals are combined and the worst individuals are deleted to get a new set (a new *generation*) the size of the old set. This is repeated until a solution is found. The initial generation is created randomly. What individuals are selected to act as "parents" for generating the new individuals is based on evaluating the *fitness* of the individuals (with some random factors also involved). The idea is that fitter individuals should have a higher chance to act as parents than not so fit ones. The fitness is also used to determine what are the worst individuals.

This general concept was instantiated in [CD+04] as follows. The set of possible individuals $\mathcal{F}$ is the set of sequences of indexes for actions, i.e.

$$\mathcal{F} = \{(a_1,...,a_m) \mid a_i \in \{1,..., \max\{|\mathcal{A}ct(s_i)|\}\}.$$

The fitness of an individual is based on k evaluation runs with this individual serving as input to the game from a given start game state s_0 (k is a parameter chosen by the tester of the game). A low fitness-value means a good individual, while a high fitness-value is considered bad.

A single run of an individual $(a_1,...,a_m)$ produces a state sequence $(s_0,...,s_m)$ and we define the single run fitness $single\_fit$ of $(s_0,...,s_m)$ as follows:

$$single\_fit((s_0,...,s_m)) = \begin{cases} j, \\ \quad \text{if } \mathcal{G}_{unwanted}((s_0,...,s_j)) = \text{true} \\ \quad \text{and } \mathcal{G}_{unwanted}((s_0,...,s_i)) \\ \quad = \text{false for all } i < j \\ \sum_{i=1}^{m} near\_goal((s_0,...,s_i)), \\ \quad \text{else.} \end{cases}$$

Remember that $\mathcal{G}_{unwanted}$ is the condition on state sequences that we are looking for in the test. Note that we might not always require $\mathcal{G}_{unwanted}$ to take the whole sequence into account. If we only look for a property of a single state, then we have to test s_j only. By using as fitness of a run fulfilling $\mathcal{G}_{unwanted}$ the number of actions needed to get to the first game state fulfilling $\mathcal{G}_{unwanted}$, we try to evolve short sequences revealing the unwanted behavior to make analyzing the reasons for the behavior easier.

Within the function $near\_goal$ we have to represent the key knowledge of the game designer about the state or state sequence he/she is interested in. $near\_goal$ has to evaluate state sequences that do not fulfill $\mathcal{G}_{unwanted}$ and we need it to measure how near these sequences come to fulfilling $\mathcal{G}_{unwanted}$. $near\_goal$ depends on the particular game to test and the particular condition we want to fulfill.

$single\_fit$ sums up the evaluations by $near\_goal$ for all subsequences of $s_0,...,s_m$ starting with the subsequence s_0,s_1. On the one hand, this assures in most cases that a state sequence not fulfilling $\mathcal{G}_{unwanted}$ has a $single\_fit$-value much higher (usually magnitudes higher) than a sequence that does fulfill $\mathcal{G}_{unwanted}$. On the other hand, we award sequences that come near to fulfilling $\mathcal{G}_{unwanted}$ and stay near to fulfilling it. And finally, we are able to identify actions (resp. indexes in the individual) that move us away from the behavior or result we are looking for (after coming near to it).

To define the fitness fit of an individual $(a_1,...,a_m)$, let $(s_0,s_1^1,...,s_m^1)$,..., $(s_0,s_1^k,...,s_m^k)$ be the game state sequences produced by the k evaluation runs. Then

$$fit((a_1,...,a_m)) = \sum_{i=1}^{k} single\_fit((s_0,s_1^i,...,s_m^i))$$

As Genetic Operators, the standard operators Crossover and Mutation on strings can be employed. Given two action sequences $(a_1,...,a_m)$ and $(b_1,...,b_m)$, Crossover selects randomly a number i between 1 and $m-1$ and produces $(a_1,...,a_i,b_{i+1},...,b_m)$ as new individual. Mutation also selects randomly an i and a random action a out of $Act - \{a_i\}$ to produce $(a_1,...,a_{i-1},a,a_{i+1},...,a_m)$ as new individual.

[CD+04] also introduced so-called *targeted* variants of Crossover and Mutation that make use of the fact that

with $near\_goal$ we can judge the consequences of individual actions within sequences and can identify actions within a sequence that lead away from game states fulfilling $\mathcal{G}_{unwanted}$. The precise definition of Targeted Crossover and Targeted Mutation require that when evaluating an evaluation run of an action sequence (a_1,...,a_m), we remember the first position s_j such that

$$near\_goal((s_0,...,s_j)) \geq near\_goal((s_0,...,s_{j-1}))+ lose\_goal,$$

for a parameter $lose\_goal$. *Targeted Crossover* between (a_1,...,a_m) and (b_1,...,b_m) first selects one of the reported positions in (a_1,...,a_m), say a_j, then selects a position in (a_1,...,a_m) between $a_{j-COtarg}$ and a_{j-1} and performs a Crossover at this selected position. The same modification leads us to a *Targeted Mutation* using $Mtarg$ to define the interval to choose from.

4 Layered evolutionary behavior testing for parameterized actions

There are several possible ways by which handling parameterized actions could be integrated into the method defined in the last section. One rather obvious way is to treat them like ordinary actions, start them off with random parameter values and add a Genetic Operator that mutates the parameters of parameterized actions. While this allows for having parameterized actions everywhere in an action sequence, it is not easy to balance the mutation of the parameters with the other Genetic Operators, the search spaces for action sequences and parameter value combinations multiply each other and there is a big danger that targeted Genetic Operators target primarily parameterized actions in early stages of the search, since the parameter value combinations are not evolved enough. Additionally, the general quality of a parameter value combination cannot be expressed, since we have only a fitness for a whole action sequence.

Since in many games parameterized actions only occur under special circumstances, we decided to use a different approach that separates the evolution of action sequences and parameter value combinations a little bit more and allows for evolving action sequences that "work" with several value combinations and value combinations that are producing good results with several action sequences. Additionally, we made the assumption that parameterized actions are a good point to break action sequences into parts, i.e. being able to perform a particular parameterized action is often a good subgoal in a game. This then means that the parameterized action is the first action of an action sequence after the subgoal is achieved (like a cornerkick or penalty kick in soccer or a pitch in baseball). And naturally reaching the subgoal should be used as good point to save the game, so that we are provided with a clearly defined start state for the sequence starting with the parameterized action.

Under these assumptions, integrating parameterized actions can be achieved by having a second evolutionary learning process that tries to evolve good parameter value combinations, in a kind of lower layer of the whole process (similar to the layered learning idea of [St00] for reinforcement learning). The crucial problem that has to be solved is how the fitness of such a parameter value combination is calculated, once again. In fact, also the question of how to evaluate the fitness of an action sequence has to be raised, again. Our solution is to evaluate the fitness of a particular parameter value combination by employing it in several action sequences as the parameter values of the parameterized action the sequence starts with. Part of the fitness is then the combination of the fitnesses of the evaluation runs of the sequences with the value combination. But we also incorporate the immediate result of the parameterized action by measuring the resulting game state after performing it. As a consequence, we also modified the fitness of an action sequence by not only using the evaluation run(s) for one value combination, but the runs of all value combinations with the action sequence.

More formally, if a_{param} is the parameterized action with which we start an action sequence and if it requires p parameters out of the value sets V_1,...,V_p, then an individual on the parameter value level has the form (v_1,...,v_p) with $v_i \in V_i$. On the action sequence level, as before, an individual has the form (a_1,...,a_m). The fitnesses are then evaluated as follows.

On the action sequence level, we evaluate each individual (a_1,...,a_m) with a selection from the current population of the parameter value level, i.e. with individuals (v_1^1,...,v_p^1),...,(v_1^l,...,v_p^l). As before a single run of (a_1,...,a_m) starts from a start state s_0, performs action $a_{param}(v_1^i,...,v_p^i)$ for an i to get to state s_1^i and then uses a_1,...,a_m to produce states s_2^i,...,s_{m+1}^i. We then compute $single\_fit$((s_0,s_1^i,...,s_{m+1}^i)) and sum up over k runs for each (v_1^i,...,v_p^i), as before.

On the parameter value level, we use the evaluation runs done for the action sequence level. This means that for an individual (v_1,...,v_p) we have game state sequences (s_0,s_1^j,...,s_{m+1}^j) produced by running the action sequence $a_{param}(v_1,...,v_p)$,a_1^j,...,s_m^j (again, we might use the same action sequence k times, but j should range from 1 to $k \times o$ to have more than one action sequence, namely o, used and to achieve an even distribution of the evaluation runs among the individuals on the parameter value level). For the fitness $fitp$ of (v_1,...,v_p), we sum up the $single\_fit$-values for the produced state sequences, but we also consider especially the "quality" of the outcome of $a_{param}(v_1,...,v_p)$ by computing a weighted sum of the evaluation of this quality and the fitness of the action sequences. More precisely, if w_{seqfit} and w_{nexst} indicate the weights, then

$$\begin{aligned} fitp((v_1,...,v_p),(s_0,s_1^1,...,s_{m+1}^1),...,(s_0,s_1^{k\times l},...,s_{m+1}^{k\times l})) \\ = w_{seqfit} \times \sum_{i=1}^{k\times o} single\_fit((s_0,s_1^i,...,s_{m+1}^i)) \\ + w_{nexst} \times \sum_{i=1}^{k\times o} near\_goal((s_0,s_1^i)) \end{aligned}$$

With regard to Genetic Operators, on the action sequence level we use the operators introduced in Section 3, includ-

ing the targeted operators. On the parameter value level, we use the rather well-known operators for lists of numerical values, i.e. crossover similar to the one on the action sequence level (but not targeted) and a mutation that varies the value of a parameter by adding/subtracting a randomly chosen value within a mutation range r_{V_i} for the i-th parameter (the r_{V_i} are parameters of the evolutionary learning algorithm).

The two layers of the learning system proceed at the same pace, i.e. new generations for both levels are created, the individuals in both levels are evaluated and based on this evaluation the next generation is created.

5 Case study: FIFA-99

We instantiated the general methods described in Sections 3 and 4 for the Electronic Arts' FIFA-99 game. In this section, we will first give a brief description of FIFA-99, then we will provide the necessary instantiations of the general method, and finally we will report on our experiments and their results.

5.1 FIFA-99

FIFA-99 is a typical example of a team sports games. The team sport played in FIFA-99 (in fact, in all games of the FIFA series) is soccer. So, two teams of eleven soccer players square off against each other and the players of each team try to kick a ball into the opposing team's goal (*scoring a goal*). In the computer game, the human player is assigned one of the teams and at each point in time controls one of the team's players (if the team has the ball, usually the human player controls the player that has the ball). In addition to actions that move a player around, the human user can let a player also handle the ball in different ways, including various ways of passing the ball and shooting it. As parameterized actions on the offensive side, we have, for example, cornerkicks and penalty kicks. On the defensive side the actions include various tackles and even different kinds of fouls.

The goal of the human user/player is to score more goals than the opponent during the game. FIFA-99 allows the human player to choose his/her opponent from a wide variety of teams that try to employ different tactics in playing the game. In addition, Electronic Arts has included into FIFA-99 an AI programming interface allowing outside programs to control NPC soccer players (i.e. the soccer players controlled by the computer). For our experiments, we extended this interface to allow for our evolutionary testing system to access various additional information, set the game to a cornerkick situation and to feed the action sequences to the game.

5.2 Behavior testing for cornerkicks

A human player of FIFA-99 (and other team sport games) essentially controls one soccer player at each point in time, allowing this player to move, pass and shoot and there is also the possibility for the human player to switch his/her

Figure 1: Cornerkick start situation

control to another of his/her players. But this switch is also performed automatically by the game if another soccer player of the human player is nearer to the ball (and switch goes through a predefined sequence of the players in the team to accommodate the way the controls for the human player work). The set of actions without parameters is

- NOOP
- PASS
- SHOOT
- SWITCH
- UP
- DOWN
- RIGHT
- LEFT
- MOVEUPLEFT
- MOVEUPRIGHT
- MOVEDOWNLEFT
- MOVEDOWNRIGHT

There are several other, parameterized actions that all are associated with special situations in the game. For our experiments, we concentrated on the action

CORNER(x,z,angle)

that performs a cornerkick. Here, the x parameter gives the x-coordinate, a value between 0 and the width of the field. The z parameter describes the z-coordinate (at least in FIFA-99 it is called z), a value between 0 and the length of the field. Finally, angle provides the angle to the field plane that the kick is aimed at (within 90 degrees).

The unwanted behavior we are looking for in our test is either scoring a goal or getting a penalty shot. So, $\mathcal{G}_{unwanted}((s_0,...,s_i))$ = true, if FIFA-99 reports a goal scored in one of the states s_0,...,s_i or a penalty shot is awarded in one of these states. In [CD+04], s_0 was the kick-off, while in our experiments in the next subsection, we start from the cornerkick situation depicted in Figure 1.

The fitness functions of both layers are based on the function $near\_goal$. We use the same definition for this function as in [CD+04]. This definition is based on dividing the playing field into four zones:

Zone 1 : from the opponent goal to the penalty box

Zone 2 : 1/3 of the field length from the opponent goal

Zone 3 : the half of the field with the opponent goal in it

Zone 4 : the whole field.

We associate with each zone a penalty value (pen_1 to pen_4) and decide which value to apply based on the position of the player with the ball (resp. if none of the own players has the ball, then we look at the position of the player that had the ball in the last state). If the position is in zone 1, we apply pen_1 as penalty, if the position is not in zone 1, but in zone 2, we apply pen_2 and so on. By $dist(s_i)$ we denote the distance of the player position from above to the opponent's goal. Then for a state sequence (s_0,...,s_i), we define $near\_goal$ as follows:

$$near\_goal((s_0,...,s_i)) = \begin{cases} dist(s_i) \times \text{penalty}, \\ \quad \text{if the own players had the} \\ \quad \text{ball in } s_{i-1} \text{ or } s_i \\ max\_penalty \\ \quad \text{else.} \end{cases}$$

The parameter $max\_penalty$ is chosen to be greater than the maximal possible distance of a player with the ball from the opponent's goal multiplied by the maximal zone penalty, so that losing the ball to the opponent results in large $near\_goal$-values and a very bad fitness. For the targeted operators we set $lose\_goal$ to $max\_penalty$ and did not consider $near\_goal$((s_0,...,s_{j-1})) at all, so that we target those actions that result in losing the ball. As in [CD+04], we used $Mtarg = COtarg = 1$.

For the parameters guiding the computation of $fitp$, we set w_{nexst} to 20 and w_{seqfit} to 1. Since a lower fitness means a better individual, the influence of the state directly after the cornerkick on $fitp$ is substantial, so that parameter value combinations that result in the own team not being in possession of the ball have a very bad fitness.

5.3 Experimental evaluation

We used the instantiation of our general method from the last subsection to run several test series. As described in [CD+04], we flagged action sequences that fulfilled $\mathcal{G}_{unwanted}$ in one evaluation run and then did 100 additional runs with this action sequence-parameter value combination. And we only consider sequence-value combinations that were successful in 80 of these additional runs for reporting to the developer/tester. The population size in the parameter value level was 20 and an individual on the action sequence level was evaluated using each of the individuals of the current value level once (i.e. $l = 20$, $k = 1$). On the action sequence level, the population size was 10.

In our experiments, every run of our test system quickly produced one or several action sequences that were flagged for further testing and from every run we got at least one parameter value-action sequence combination that passed the additional test (and scored a goal; we usually got several combinations that produced a penalty shot). Many of these

Figure 2: First sequence, ball in air

Figure 3: First sequence, ball under control

combinations are quite acceptable, i.e. they present a game behavior that could be observed in a real soccer game, except for the fact that the game does allow for the scoring by this combination so often (or even all the time).

In the following, we will present a few rather short action sequences that scored in our experiments all the time, with screenshots for the shortest sequence we found. We will then also present some screenshots showing a not very intelligent game behavior consistently leading to a penalty shot. This is the kind of game behavior that a developer will want to change, respectively allow to be exploited only once or twice. In all figures, the light-colored soccer players are the attackers controlled by the action sequences produced by our system and the dark-colored players are the opponents.

Since a cornerkick is a dangerous situation for defenders in a soccer game, it can be expected in a computer soccer game that there are many opportunities for scoring goals. Here are the 3 shortest action sequences we have found using our testing method, so far:

Figure 4: First sequence, shoot

Figure 5: First sequence, goal!

Figure 6: Second sequence, ball in air

Figure 7: Second sequence, ball under control?

- CORNER(-1954,-620,28.1), LEFT, SHOOT
- CORNER(-1954,-620,28.1), MOVEDOWNRIGHT, UP, SHOOT
- CORNER(-2097,-880,27.5), LEFT, DOWN, MOVEDOWNRIGHT, SHOOT

The first two were produced by the same run, after 3:15, resp. 3:13 minutes of search, while the third was found after 7:22 minutes (in a different run). The first two are a good example for how the fitness function fit aims at producing shorter sequences.

Figure 2 shows the effect of the first value-sequence combination, resp. the effect of the cornerkick parameter values. In Figure 3, the attacker has the ball under control and moves to the left to get some distance to the goalie that will come out of its goal soon. Figure 4 shows the situation after the goalie has come out and the attacker has shot towards the goal. And Figure 5 shows the goal.

When we added being awarded a penalty shot as a successful attack action sequence, we were not aiming at cornerkicks (this was more of interest for the sequences starting from the kickoff, in our opinion). But surprisingly, our system evolving action sequences found more sequences leading to penalty shots than it found leading to a goal. These sequences are longer (but found earlier by the evolutionary search) and while some of the situations could happen in a real soccer game (although you do not see many penalty shots developing out of corners), there are quite a few sequences that we would consider unrealistic, due to the foul committed being rather unnecessary. The following sequence is an example for such a sequence:

- CORNER(-1954,-775,28.1), NOOP, DOWN, DOWN, RIGHT, SWITCH, MOVEUPLEFT, MOVEDOWNRIGHT, LEFT, LEFT, SWITCH

Figure 6 shows the ball in the air. Figure 7 shows the attacker trying to get the ball under control. In Figure 8, the attacker is still trying to get control, while the action sequence now switches to a different attacker, essentially leaving the player with the ball with the built-in control, which

Figure 8: Second sequence, still trying to get control

Figure 9: Second sequence, stupid foul!

drives it nearly out of the penalty box. In Figure 9, the control switches back just when the attacker is brought down from behind just a little inside of the penalty box. This is a rather stupid move by the defender and as consequence, Figure 10 shows the referee indicating a penalty shot. We had several other sequences leading to a penalty shot, where actively moving control away from the key attacker to let the game take over its behavior resulted in a penalty. While this naturally is not a behavior expected by a human player (and therefore not intensively tested) it shows an advantage of our testing method.

6 Conclusion and Future Work

We presented an extension to the behavior testing method proposed in [CD+04] that allows us to deal with actions that have parameters. By adding a lower evolutionary layer for evolving good parameter value combinations, we are able to co-evolve parameter value combinations and action sequences resulting in value-sequence combinations that produce unwanted behavior in games. Our tests with the FIFA-99 game produced many value-sequence combinations scoring goals and resulting in penalty shots. Several of the penalty shot situations are not very realistic both from the attacker and the defender side, so that we revealed a weakness of the AI controlling the defender that should have been avoided.

Figure 10: Second sequence, penalty kick!

Our future work will focus on developing fitness functions for unwanted behavior that do not focus on scoring goals. In newer versions of FIFA, we have special players with special moves and special abilities (modeled after real, and well-known human soccer players) for which the game designers have special expectations. Bringing the game into situations where these expectations can be observed (and reaching such situations in different ways) is not easy to achieve using human testers, but an obvious application for our method.

Bibliography

[CD+04] B. Chan, J. Denzinger, D. Gates, K. Loose and J. Buchanan. Evolutionary behavior testing of commercial computer games, Proc. CEC 2004, Portland, 2004, pp. 125–132.

[DF96] J. Denzinger and M. Fuchs. Experiments in Learning Prototypical Situations for Variants of the Pursuit Game, Proc. ICMAS'96, Kyoto, 1996, pp. 48–55.

[PJ00] M.A. Potter and K.A. De Jong. Cooperative coevolution: An architecture for evolving coadapted subcomponents, Evolutionary Computation 8, 2000, pp. 1–29.

[St00] P. Stone. Layered Learning in Multi-Agent Systems: A Winning Approach to Robotic Soccer, MIT Press, 2000.

[vR03] J. van Rijswijck. Learning Goals in Sports Games, Game Developers Conference, San Jose, 2003. http://www.cs.ualberta.ca/ javhar/research/ LearningGoals.doc

[WS03] S. Whiteson and P. Stone. Concurrent Layered Learning, Proc. AAMAS-03, Melbourne, 2003, pp. 193–200.

Board Evaluation For The Virus Game

Peter Cowling
Modelling Optimisation Scheduling And Intelligent Control (MOSAIC) Research Centre
Department of Computing
University of Bradford
Bradford BD7 1DP
UK
P.I.Cowling@bradford.ac.uk
http://mosaic.ac

Abstract- The Virus Game (or simply Virus) is a turn-based two player perfect information game which is based on the growth and spread of competing viruses. This paper describes a CPU efficient and easy to use architecture for developing and testing AI for Virus and similar games and for running a tournament between AI players. We investigate move generation, board representation and tree search for the Virus Game and discuss a range of parameters for evaluating the likely winner from a given position. We describe the use of our architecture as a tool for teaching AI, and describe some of the AI players developed by students using the architecture. We discuss the relative performance of these players and an effective, generalisable scheme for ranking players based on similar ideas to the Google PageRank method.

1 Introduction

For two player games of perfect information with a reasonably low number of moves in any given position, such as chess, draughts and Othello, strong AI players to date have principally used a combination of fast, limited-depth minimax tree search using alphabeta pruning (Knuth and Moore 1975) and a board evaluation function to approximate the probability of each player winning from a given position. Tree search has been enhanced by techniques such as iterative deepening, using on-chip hardware to conduct the search, maintaining hash tables of previously evaluated positions (Campbell et al 2002) and heuristic pruning techniques (Buro 2002).

Creating a machine to "learn" game strategy has been an important goal of AI research since the pioneering work on checkers/draughts of (Samuel 1959). Research to date has shown little advantage for learning approaches applied to the tree search for two player perfect information games. However, many of the strongest AI players in existence now "learn" board evaluation functions. Logistello (Buro 2002) uses statistical regression to learn parameter weights for over a million piece configurations based on a very large database of self-play games. Blondie24, which has been popularised by the highly readable book (Fogel 2002), evolves weights for an artificial neural network to evaluate positions in the game of draughts. (Kendall and Whitwell 2001) uses an evolutionary scheme to tune the parameter weights for an evaluation function of a chess position. Their evolutionary scheme is notable in that evolution occurs after every match between two AI players, which gives faster convergence, since running a single game can take several CPU seconds. (Abdelbar et al 2003) applies particle swarm optimisation to evolve a position evaluation function for the Egyptian board game Seega. (Daoud et al 2004) evolve a position evaluation function for Ayo (more comonly known as Awari). Their evolutionary scheme is interesting in that a "test set" is chosen and the result of matches against this test set determines the fitness of an individual in the population. The "test set" was chosen at random. Later in this paper we will suggest how this idea may be taken further using a ranking scheme based on the principal eigenvector of the results matrix. In the work of (Ferrer and Martin 1995) the parameters used to measure board features for the ancient Egyptian board game Senet are not given in advance, but are evolved using Genetic Programming. While Senet is not a game of perfect information, this is also an interesting angle of attack for perfect information games.

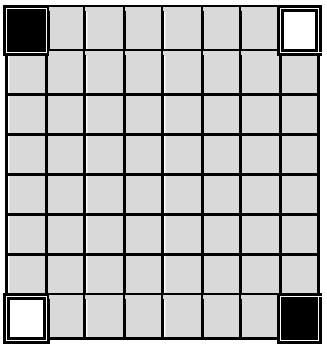


Fig.1. The Virus game starting position.

In this paper we will explore the two-player perfect information game of Virus. The earliest appearance of which we are aware of the Virus Game was in the Trilobyte

game 7th Guest (Matthews 2000). The Virus game is a two player game of perfect information, played on an *nxn* board (in this paper we will use $n = 8$ as in chess, draughts/checkers or Othello/reversi). There are two players, black and white, who play alternately, starting with black. Initially the board is set up as in Fig. 1.

Two types of moves are available at each turn. The first type of play involves *growing* a new piece adjacent to an existing piece of the same colour (Fig. 2). The second type of play involves *moving* a piece to another square a distance exactly 2 squares away (via an empty square) (Fig. 3). Note that squares are considered adjacent if they share an edge or corner. In either case all opposing pieces next to the moved piece change colour.

Play continues until neither player can move, or until one player has no pieces left, when the player with the most pieces wins the game.

Virus has a higher branching factor than all of chess, draughts/checkers or Othello/reversi, but in common with all of those games there are a large number of moves from any given position which would immediately be discounted as ridiculous by a reasonably intelligent player. Hence tree pruning based on alphabeta search is effective for Virus, as we will see later.

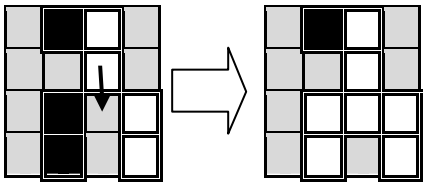


Fig. 2. The white piece grows.

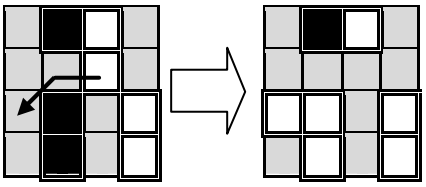


Fig. 3. The white piece moves.

We have used Virus as a testbed since it has very simple rules and yet the tactics and strategy of the game appear somewhat difficult. In particular, since the board changes a very great deal after only a few moves, it is arguably a difficult game for a human player to play well (much the same might be argued for Othello (Buro 1997)). However, after playing the game several times strategic and tactical ideas start to appear, and we will discuss these later. We have developed an Application Programming Interface (API) which greatly simplifies the implementation of AI ideas amd which ensures that they use highly efficient code.

We do not know of any published work about the game, so Virus proved very useful as a tool for teaching AI to undergraduate and Masters students. Students were given a library containing the API, search code and a client for visualising Virus games (Fig. 7), and had to devise and refine a board evaluation function using positional ideas and evolution of parameter weights. The students produced 43 Virus board evaluation functions of varying sophistication and effectiveness. We will describe briefly some of the ideas used in section 5 below.

This paper looks at how we may represent the Virus board in section 2, investigates the nature of the search tree for Virus in section 3, discusses the API for developing AI for Virus (and other board games) as well as the Virus client and server in section 4, discusses a method for tournament ranking and its wider possibilities in section 5, explains several parameters which may be used to evaluate Virus positions in section 6, finishing with conclusions in section 7.

2 Representing the Virus Board

To greatly speed up computations based on the Virus board, we represent the board as a pair (B,W) of 64-bit unsigned integers, where each bit in the first (second) integer is set to one if and only if there is a black (white) piece in the corresponding square. We use the convention that in any board representation, it is always the black player to move (by reversing the colour of all pieces if necessary) since the game is symmetric with respect to black and white. It is possible to represent all positions of the Virus board using $\lceil \log_2(3^{64}) \rceil = 102$ bits, but any such representation would be much more difficult, and slower, to manipulate. We may then use fast bit manipulation routines such as those at (Anderson 2004).

A Virus move is represented by a single 64 bit unsigned integer M where

$$(B,W) \mathbin{®}_M ((W \,\&\, M) \,\mathbf{XOR}\, W, B \,\mathbf{XOR}\, M)$$

Where **&** is the binary AND operator and **XOR** is the binary exclusive-or operator. This move representation can be used for any game with alternating turns where (for black about to move):

1. No new white pieces may be added.
2. Black squares may become empty.
3. Empty squares may become black.
4. White squares may become black or remain white, but may not be emptied.

Hence this representation is immediately applicable to Othello. With the minor modification

$$(B,W) \mathbin{®}_M ((W \,\&\, M) \,\mathbf{XOR}\, W, B \,\mathbf{XOR}\, (M \,\mathbf{XOR}\, W))$$

we can change restriction 4. to

4. White squares may become empty or remain white, but may not become black.

when we can use this representation for games such as draughts.

The power of this representation becomes evident when we consider how succinctly we may write useful board manipulation functions using binary arithmetic. For example, one of the most commonly used functions in move generation and board evaluation is to find all squares adjacent to a set *C* of squares, **Adj**(*C*). Let **N**, **S**, **E**, **W**, **NE**, **NW**, **SE**, **SW** be the 64 bit integers which

0	0	0	0	0	0	0	0
0	1	1	1	1	1	1	1
0	1	1	1	1	1	1	1
0	1	1	1	1	1	1	1
0	1	1	1	1	1	1	1
0	1	1	1	1	1	1	1
0	1	1	1	1	1	1	1
0	1	1	1	1	1	1	1

NW

1	1	1	1	1	1	1	0
1	1	1	1	1	1	1	0
1	1	1	1	1	1	1	0
1	1	1	1	1	1	1	0
1	1	1	1	1	1	1	0
1	1	1	1	1	1	1	0
1	1	1	1	1	1	1	0
0	0	0	0	0	0	0	0

SE

Fig.4. **NW** and **SE** unsigned long integers.

consist of all ones except for a row of zeroes in the top row (**N**), bottom row (**S**), left column (**W**) or right column (**E**) as illustrated in Fig. 4, where the least significant bit is in the bottom right hand corner, the bit to the left of the 2^i bit is the 2^{i+1} bit and the bit above the 2^i bit is the 2^{i+8} bit.

Then, if Ú represents the binary OR operator, Ø the NOT operator and $^{®i}$, $^{¬j}$ the right- and left-shift operators, respectively,

$$\mathbf{Adj}(C) = ØC \,\&\, [(C \,\&\, \mathbf{N})^{¬8} Ú\ (C \,\&\, \mathbf{NE})^{¬7} Ú\ (C \,\&\, \mathbf{E})^{®1} Ú\ (C \,\&\, \mathbf{SE})^{®9} Ú\ (C \,\&\, \mathbf{S})^{®8} Ú\ (C \,\&\, \mathbf{SW})^{®7} Ú\ (C \,\&\, \mathbf{W})^{¬1} Ú\ (C \,\&\, \mathbf{NW})^{¬9}]$$

which can be computed using only 25 primitive binary operators, corresponding to 25 machine instructions at chip level. Other operators such as the iterators **FirstSquare**(*C*) which finds the rightmost square in the lowest possible row of *C*, and **NextSquare**(*C*,*d*) which finds the next square in *C* after square *d*, may be calculated as follows:

$$\mathbf{FirstSquare}(C) = C \ \mathbf{XOR}\ (C \,\&\, (C\text{-}1))$$

$$\mathbf{NextSquare}(C,d) = \mathbf{FirstSquare}(C \,\&\, (Ø((d^{¬1})\text{-}1))$$

Generation of all moves from a given position becomes simple and fast, using this representation, and pseudocode is given in Fig. 5.

Another fundamental function for a Virus board, is to count the number of squares in a subset. A naïve implementation which looks at each bit would be far too slow. In this case we use a **Count** operator which uses 24 machine instructions to count the number of bits in a 64-bit integer as follows:

$S \leftarrow (S \,\&\, 01..01) + (S^{®1} \,\&\, 01..01)$
$S \leftarrow (S \,\&\, 0011..0011) + (S^{®2} \,\&\, 0011..0011)$
...
$\mathbf{Count}(S) = (S \,\&\, 0..01..1) + (S^{®32} \,\&\, 0..01..1)$

where after the first step each pair of bits is replaced by a count of the number of bits in the pair, after the second step each group of 4 bits (nybble) is replaced by the number of bits in the nybble, and so on. A slightly faster implementation based on table lookup is possible, but would consume large amounts of memory which could be a problem when using a large number of AI players on a server.

```
// Generate all moves from position (B,W)
Moves = {}

// Empty squares in position (B,W)
E = Ø(B Ú W)

// First 1-step moves (to c1 from any adjacent square)
C1 = Adj(B) & E
c1 = FirstSquare(C1)
while (c1 ≠ 0)
    Moves.Add(c1 Ú (Adj(c1) & W))
    c1 ← NextSquare(C1,c1)

// Now 2-step moves (from c2 to d2)
c2 = FirstSquare(B)
while (c2 ≠ 0)
    D2 = Adj(Adj(c2) & E) & E
    d2 = FirstSquare(D2)
    while (d2 ≠ 0)
        Moves.Add(c2 Ú d2 Ú (Adj(d2) & W))
        d2 ← NextSquare(D2,d2)
    c2 ← NextSquare(B,c2)
```

Fig. 5. Pseudocode for Virus move generation.

We will see further illustration of the efficiency of the compiled code resulting from this representation and the succinctness with which tactical ideas may be expressed when we talk about board parameters later.

3 Searching the Virus Game Tree

We search for the best move from a given position by searching the Virus game tree to a fixed depth (ply) using negamax search. This negamax search is further enhanced by using alphabeta pruning and ordering moves based on the very simple evaluation function (number of black pieces) – (number of white pieces). The difference between the basic minimax search and the search with these two enhancements is illustrated in Fig. 6 where we plot log (|leaf nodes|) against search depth for a mid-game position. The figure clearly illustrates the very large advantages of alphabeta pruning and move ordering. Using regression on these three plots, we estimate that the branching factor per ply is 86.6 for pure minimax search, 14.3 for Minimax with alphabeta pruning and 8.5 for alphabeta pruning with a simple ordering heuristic. More complex move ordering heuristics and transposition tables might also be used to drive the branching factor down a little further.

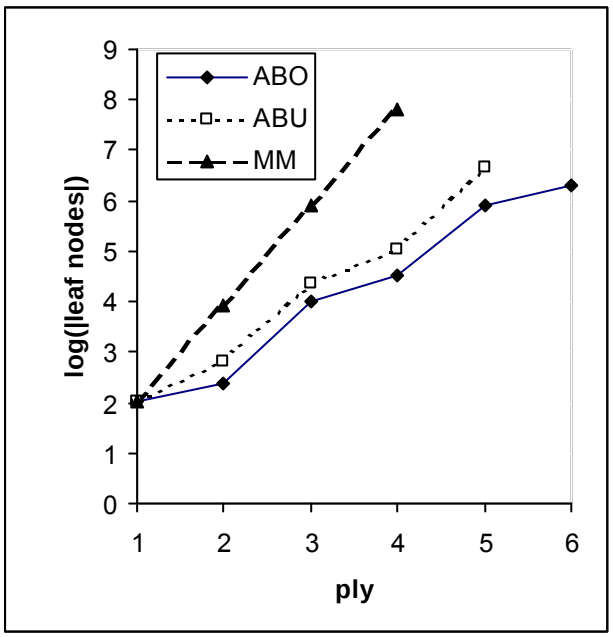


Fig.6. Plot of $\log_{10}$(|leaf nodes|) against ply for Alphabeta ordered (ABO), alphabeta unordered (ABU) and minimax without alphabeta (MM) search strategies.

4 The Virus Client and Server

A graphical Virus client, illustrated in Fig. 7 allows visualisation of games. This client allows the user to watch a game being played as well as offering the possibility to save and load games and to step forwards and backwards through saved games.

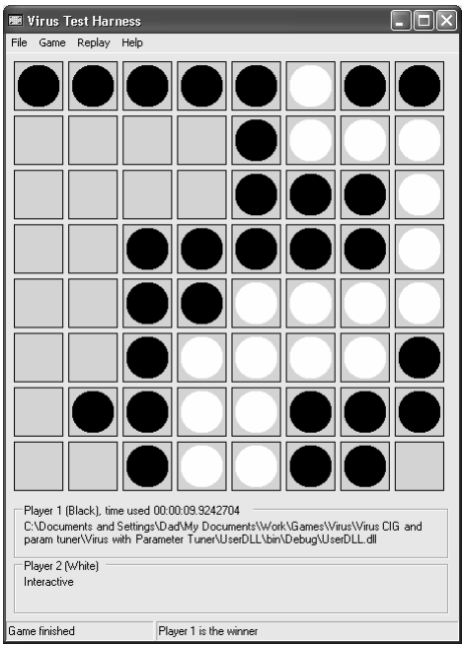


Fig. 7. The Virus graphical interface.

In order to present a user-friendly Application Programming Interface (API) for Virus and yet still benefit from the speed of the binary representation as given in section 2, we use the expressiveness of C# and its ability to hide complex functions and expose a straightforward programming interface. We have built a SquareSet struct which may be visualized as a set of squares by the AI programmer, and which presents highly efficient (and relatively complex) binary manipulation to the user as a suite of simple variables and functions. For example, SquareSet.Count is one of the most common operations on a SquareSet, as described earlier. "Pretending" that this is a variable of the SquareSet hides the complexity behind that function. Another fundamental operator which is hidden by an efficient binary function is the **GetConnectedGroup()** function which gets a connected group of squares. As we can see below using our API this becomes straightforward:

```
GetConnectedGroup(S)
    G ← GetFirstSquare(S)
    while (Adj(G) & S ≠∅)
        G ← G Ú (Adj(G) & S)
    return G
```

Fig. 8. Pseudocode for **GetConnectedGroup()**

The operators on a square set (**&**, Ú, **Ø**, **XOR**) are documented in terms of squares (rather than bits) to facilitate their understanding in a game context. We define the natural operators + as a synonym for Ú and – where $A - B = A$ **&** ($\varnothing B$).

A Board consists of two SquareSets (B,W), one for black pieces and one for white. Again it implements a number of fast methods for board manipulation while hiding the details. All access functions accept the player as a parameter (so that separate board evaluation functions do not have to be written for white and black). In particular a Board exposes the sets B_i and W_i. B_0 (W_0) is the set of black (white) squares. B_i (W_i) is the set of empty squares at distance i from a black (white) square. We also define $B_{\le m} = \bigcup_{i=0,1,..m} B_i$ and $W_{\le m} = \bigcup_{i=0,1,..m} W_i$. Then, for example, B_1 is the set of squares which black can move to via a 1-step move, B_2 is the set of 2-step moves for black etc. Finding these sets using our API is equally straighforward. We illustrate the calculation of B_i and $B_{\le i}$ in Fig. 9.

The client and API is generic across all games on an 8×8 board with pieces played on squares and two piece colours (including Othello, draughts and Seega). Only the GetMoves() function needs to be changed in order for the API to work for a different game. Of course the leaf node evaluation would be very different for each of these games.

To use Virus as a teaching tool and to allow competition between Virus AI players submitted via the Internet, Black Marble, a software development house developed a web server which ran games between

different AI players and reported the results and league tables. It was found that allowing 10 CPU seconds per game for each AI (on a powerful twin 3.06 GHz Xeon processor server machine with 2GB RAM) for a 3ply search was the best compromise between AI search, board evaluation and getting a large number of results. A league was played using all of the currently loaded AI players. As new players were uploaded (by ftp) they jumped to the head of the queue and quickly got games, in order to encourage submission to the server. Over 600,000 games were played on the server, submitted by 45 different players over a 4 week period. Following this tournament the AI players had to be ranked. We will discuss an effective ranking mechanism in the next section.

```
// Empty squares in position (B,W)
E = Ø(B Ú W)

// Current distance under consideration
d = 0

B0 ← B
B≤0← B

while (Bd ≠ ∅)
    B≤d+1← B≤d Ú (Adj(Bd) & E)
    Bd+1← B≤d+1 XOR B≤d
    d ← d + 1
```

Fig. 9. Pseudocode for finding B_d and $B_{\leq d}$

5 Tournament Ranking and Results

Suppose that we have played a tournament with n players, where each pair has played both as black/white and as white/black. Then we have an $n \times n$ matrix M of tournament results. A player scores 64 points for a won game, plus the difference in the number of squares, 32 points for a drawn game, and 0 points for a loss. Then M_{ij} is the score for player j when i and j played. If we normalise this matrix so that row sums are one, then we have a stochastic matrix, and the entries in this matrix can be regarded as the transition probabilities in a Markov chain, where the probabilities represent the probability of jumping from a losing player (state) to a winning player (state). If we work out the steady state distribution of this transition matrix then this represents the probability that a given player wins a match averaged over an infinite series of games. This is similar to the method that the Google PageRank algorithm uses for ranking web pages (Brin and Page 2000). Hence the steady state of this Markov chain gives a very fair reflection of the relative performance of each player.

In order to find the steady state we must find a vector $\boldsymbol{x}$ satisfying

$$\boldsymbol{x}M = \boldsymbol{x}$$

i.e. we must find the principal eignevector of matrix M. By the Perron-Frobenius theorem (Grimmett and Stirzaker 1987) we know that such an $\boldsymbol{x}$ exists and is unique. Moreover, we know that for any vector $\boldsymbol{y}$ which is not orthogonal to $\boldsymbol{x}$ we know that

$$\lim_{k\to\infty} \boldsymbol{y}M^k = \boldsymbol{x}$$

so that we may calculate $\boldsymbol{x}$ iteratively starting from, say, $\boldsymbol{1}$ (the vector of all 1s) and iteratively multiplying by M until convergence occurs (typically after only five or six matrix multiplications).

The use of ranking methods based on the principal eigenvector has wide potential in the evolution of board evaluation methods, since it gives us a very precise idea of the relative effectiveness of players, which can be used, for example, in roulette wheel selection for evolutionary algorithms, or for choosing elite populations. An incomplete set of matches between players can be used to get a good idea of their ranking. Hence rather than, for example, choosing a test set as in (Daoud et al 2004) we may find the results of a small percentage of the matches and use the stationary distribution/eigenvector method to "fill in the gaps". The fact that these values are already normalised so that a principal eigenvector value twice as large corresponds to a player who is twice as strong makes them particularly useful. We are currently investigating a range of board evolution methods based upon the eigenrank.

The ideas present in the 45 AI players submitted to the server represent a wide cross-section of tactical (and to a lesser extent strategic ideas). These ideas include:

- Consideration of "degree of safety" for a piece and square
- Mobility
- Pattern matching (notably to count the number of times the disadvantageous pattern consisting of three pieces of one colour and one empty square occurs.
- Assigning positional scores for owning different board squares such as recognising that corner squares are good in the opening and assigning a different score to each square.
- Development of an opening book. Note that similarly to the game of Go (Müller 2002) the opening stages of Virus are very difficult to analyse, even on an 8×8 board.
- Ratio of surface area to volume (using a biological analogy)
- Tuning parameter weights dependent on game stage (as measured by the number of empty squares left).

The sophistication of the ideas submitted to the Virus server gives strong support to the idea of games as an effective way to teach and promote understanding of

advanced AI concepts. In addition, many of the submitting players developed add-on tools and used evolutionary parameter tuning schemes.

6 Evaluation Functions for the Virus Game

There are several parameters for Virus which capture the strategic and tactical ideas behind the game. In this section we will illustrate how succinctly these tactical and strategic ideas may be presented using the notation (and API) described above.

The simplest measure is the number of black and of white counters in the board position:

- $|B_0|$ and $|W_0|$

and indeed a 3-ply search using only $|B_0| - |W_0|$ as an evaluation function is a challenging opponent for a human virus player.

Other ideas capture the common notion of mobility (ie. the number of moves in a given position:

- $|B_1|, |B_2|, |W_1|$ and $|W_2|$

We may further refine these measures by considering only moves which result in captures for one player or another.

It is also interesting to consider the size of the largest move of each type for each player (for $i = 1,2$):

- $\max_{s \in Bi} |\mathbf{Adj}(s)\ \&\ W_0|, \max_{t \in Wi} |\mathbf{Adj}(t)\ \&\ B_0|$

We may also consider the total number of the opponent's pieces which are vulnerable to capture

- $|(\mathrm{Adj}(B_1) \cup \mathrm{Adj}(B_2))\ \&\ W_0|$,
  $|(\mathrm{Adj}(W_1) \cup \mathrm{Adj}(W_2))\ \&\ B_0|$

An important strategic idea which emerges after several games is the notion of "encirclement". Here we wish to limit the range of squares which may count for the opponent at game end by encircling the opponent's pieces into a small area of the board, and thus capturing all of the empty space remaining at our leisure at game end (since there is no way for the oponent to move into the space). Hence we have the idea of "totally safe squares". For example, all of the empty squares in Fig. 7 are "totally safe squares" for black since they are surrounded by black pieces which cannot be captured (i.e. the black pieces are not adjacent to any "unsafe" empty squares which would allow them to be captured. The algorithm for calculating this parameter is quite complicated without our API notions, but with the API we get the algorithm as shown in Fig. 10.

While the idea of totally safe squares captures the immediate tactical consequences of enclosure, in order to address the strategic issue of how we can work towards enclosure of the opponent, we can consider the set of empty squares which are k closer to one player than the other, e.g. for the black player:

- $\bigcup_{i=1,2,\ldots} (B_i - W_{\leq i+k})$

Other strategic and tactial ideas for Virus (and indeed other games) can be easily and efficiently implemented using our API. The above set of parameters are arguably a set which, given proper tuning of parameter weights, could give rise to a very strong player.

```
// Find the set T_B of totally safe squares for black from position (B,W)
T_B = ∅

// Squares not reachable from W_0
W¥ = Ø(W_0 Ú W_1 Ú ...)

// The candidate groups (together with their boundary of black squares)
C = W¥ Ú Adj(W¥)

// Empty squares in position (B,W)
E = Ø(B Ú W)

// Go through connected group by connected group checking for safety along the boundary.
while (C ≠ 0)
    G = GetConnectedGroup(C)
    while (G ≠ 0)
        if (Adj(G) & E == ∅)
            T_B ← T_B Ú (G & E)
        C ← C - G
```

Fig. 10. Pseudocode for a fast algorithm to find totally safe squares for the black player.

7 Conclusion

We have presented the board game Virus and a generalisable API for Virus which allows effective AI to be developed quickly and with relatively little experience. We have analysed the search tree for Virus as well as presenting a range of tactical and strategic ideas. We have used these ideas to show how an easy-to-use API can facilitate the development of AI for research and teaching purposes. We have discussed a server architecture for Virus that allows AI players to be submitted across the Internet and a generally applicable ranking method based on treating match results as transition probabilities in a Markov chain.

We are currently working on a generalisable evolutionary scheme for tuning the parameters of the evaluation function for any board game which may be represented using the Virus API.

The Virus API, client and server are publically available for non-commercial use. If you would like to make use of them send an email to P.I.Cowling@bradford.ac.uk.

Acknowledgments

The work described here was partially funded by Microsoft UK Ltd., and I am particularly grateful to Gavin

King of Microsoft for his support and advice. I would like to thank Steve Foster who first introduced me to the Virus Game. I am also grateful to Robert Hogg, Richard Fennell and Nick Sephton of Black Marble Ltd. who created and continue to maintain the client and server for Virus. I would particularly like to thank the 43 students on the "AI for Games" module who acted as guinea pigs for this learning experiment. I am grateful to Professor Simon Shepherd who reminded me of the possibilities of the principal eigenvector and pointed out the link with Google. Finally, Naveed Hussain gave me some additional references and is continuing this work.

Bibliography

Abdelbar, A.M., Ragab, S., Mitri, S., "Applying co-evolutionary particle swarm optimisation to the Egyptian board game Seega", in Proceedings of the First Asia Workshop on Genetic Programming (part of CEC 2003) 9-15.

Anderson, S.A., "Bit Twiddling Hacks", http://graphics.stanford.edu/~seander/bithacks.html.

Brin, S., Page, L., "The anatomy of a large-scale hypertextual Web search engine", Computer Networks and ISDN Systems, vol. 30 (1-7) (1998) 107-117.

Buro, M., "Improved Heuristic Mini-Max Search by Supervised Learning", Artificial Intelligence, Vol. 134 (1-2) (2002) 85-99.

Buro, M., "The Othello match of the year: Takeshi Murakami vs. Logistello", ICCA J. 20 (3) (1997) 189-193.

Campbell, M., Hoane, A.J. Jr., Hsu, F-h., "Deep Blue", Artificial Intelligence 134 (2002) 57-83.

Daoud, M., Kharma, N., Haidar, A., Popoola, J., "Ayo, the awari player, or how better representation trumps deeper search" in Proceedings of the 2004 IEEE Congress on Evolutionary Computation (CEC 2004) 1001-1006.

Ferrer, G.J., Martin, W.N., "Using genetic programming to evolve board evaluation functions" in Proceedings of the 1995 IEEE Congress on Evolutionary Computation (CEC95) 747-752.

Fogel, D.B., "Blondie24: Playing at the edge of AI", Morgan Kaufmann, 2002.

Grimmett, G.R., Stirzaker, D.R., "Probability and Random Processes", Oxford Science Publications 1987.

Kendall, G., Whitwell, G., "An evolutionary approach for the tuning of a chess evaluation function using population dynamics", in Proceedings of the 2003 IEEE Congress on Evolutionary Computation (CEC 2003) 995-1002.

Knuth, D.E., Moore, R.W., "An analysis of alpha-beta pruning", Artificial Intelligence 6(4) (1975) 293-326.

Matthews, J., "Virus Game Project", http://www.generation5.org/content/2000/virus.asp.

Müller, M., "Computer Go", Artificial Intelligence, Vol. 134 (2002) 145-179.

Samuel, A., "Some studies in machine learning using the game of checkers", IBM J. Res. Develop. 3 (1959) 210-229.

An Evolutionary Approach to Strategies for the Game of Monopoly®

Colin M. Frayn
CERCIA
School of Computer Science
University of Birmingham,
Edgbaston, Birmingham, UK
B15 2TT
mailto:cmf@cercia.ac.uk

Abstract- The game of Monopoly® is a turn-based game of chance with a substantial element of skill. Though much of the outcome of any single game is determined by the rolling of dice, an effective trading strategy can make all the difference between an early exit or an overflowing property portfolio. Here I apply the techniques of evolutionary computation in order to evolve the most efficient strategy for property valuation and portfolio management.

1 Introduction

Monopoly® is primarily a game of skill, though the short-term ups and downs of an individual player's net worth are highly dependent on the roll of a pair of dice. As such, it separates itself from completely skill-controlled games such as chess and Go, where no direct element of unpredictability is involved except that of guessing the opponent's next move. Despite the element of change, a strong strategy for which properties to purchase, which to develop and which to trade, can vastly increase the expected results of a skilful player over less knowledgeable opponents.

There are many parallels here with real life, where a wise property investor, though largely subject to the whims of the property market, can increase his or her expected gains by a process of shrewd strategic dealing. Much of the skill is involved with appraising the true value of a certain property, which is always a function of the expected net financial gain, the rate of that gain and the certainty of that gain.

The game considered in this work is as faithful to the original rules as possible. Players take it in turns to roll two dice, the combined total determining the number of squares over which they move. Players may acquire new assets either by purchasing available properties on which they randomly land, or else by trading with other players for a mutually agreeable exchange price. Rent is charged when a player lands on a property owned by another player, varying based on the level of development of that particular property. In this study, we use the property names of the standard English edition of the game.

Much anecdotal evidence exists concerning the supposed "best strategies", though very few careful studies have been performed in order to gain any quantitative knowledge of this problem. Because of the inherently stochastic nature of the game, "best strategies" are often described without a sufficient statistical foundation to support them.

In 1972, Ash & Bishop performed a statistical analysis of the game of Monopoly® using the method of Markov chain analysis. They evaluated all the squares on the board in order to determine the probability of an individual player landing on each square in any one turn. Furthermore, they gave an expected financial gain for every roll of the dice, given the property ownership situation. The results of this study showed that the most commonly visited group of properties was the orange street consisting of Bond Street, Marlborough Street and Vine Street.

This analysis gave some insights into suggested strategies for the game. For example, encouraging the acquisition of regularly-visited properties. A simple strategy built on a re-evaluation of property value based on the expected gain per turn could well pay dividends. For example, in the standard rules, it takes on average 1400 opponent turns to pay for the purchase cost of Old Kent Road (if unimproved), but only 300 to pay for Mayfair (Ash & Bishop, 1972). One might argue therefore that Mayfair is nearly 5 times under-priced compared to Old Kent Road.

However, there is much more to this than a simple statistical evaluation. For example, when improved to a hotel, both the properties mentioned above (Old Kent Road and Mayfair) take approximately 25 opponent turns to repay their own development costs. So we must find a fair value for these two properties which considers not only their expected (pessimistic) time to repay development costs, but also the potential gain should we be able to purchase all members of the same colour group, and the potential cost of then developing those properties to the required levels. It should also consider other factors such as the mortgage value for each property, strategies for paying to leave jail, when to develop, when to mortgage (and un-mortgage) and how to handle bidding wars.

Clearly we need a more advanced method of obtaining fair prices for all the properties on the board, based on one's own state and that of the opponents.

In this work, I investigate an evolutionary approach to the game of Monopoly®. I propose a scheme for representing a candidate strategy (section 2), and present the results a considerable number of games using both a single- (section 3) and multiple-population (section 4) approach. I conclude with the lessons learned from this study (section 5) and the scope for future investigation (section 6).

2 Evolutionary Approach

Evolutionary computation can be applied to the problem of strategy design in the game of Monopoly®. It allows the simultaneous optimisation of a very large number of interdependent variables, which is exactly what is required in order to develop coherent fair-price strategies for such a complex environment.

In the case of Monopoly®, each individual in the population represents a different set of strategies which can be used to make the various decisions required in the game. The representation used in this work consists of four distinct elements;

- Valuations for each property on the board.
- Valuations for each property when held as a member of a street.
- Valuations for the extra value of a property based on the number of houses built on it.
- Extra game-related heuristic parameters.

The first three elements are self-explanatory, though the fourth requires some elaboration. In order to generate a list of required parameters, it was necessary to consider all the decisions made by a human player during the course of a game, and to decide how to encode those decisions as parameter values. The final list was as follows:

- Parameters concerned with whether or not to pay to exit jail. This was modelled as a linear combination of the maximum and average estimated opponent net worth and house counts.
- Parameters concerned with the valuation penalty applied to mortgaged properties, depending on whether they are members of complete streets or not.
- Parameters governing a desired minimum cash position, based on average and maximum opponent net worth and number of houses.

All parameters were stored as floating-point values, and were initialised with random perturbations about the following defaults:

- Properties are worth 1.5 times their face value, but 4 times if members of a street.
- House values are twice their development cost.
- Stay in jail if (maximum opponent worth) + (average opponent worth) + (10*number of houses on the board) is greater than 10000. Otherwise, pay to exit.
- Keep a minimum of 200 pounds in cash, plus 1% of the total and average opponent net worth, plus 5% of the number of houses or hotels.

The exact choice of default values here made no difference to the outcome of the simulation, except that outrageously unsuitable values would cause the evolution process to take longer to settle down to a stable end state.

A detailed interface was also designed, incorporating all the rules of Monopoly®. A few slight alterations were made in order to make the game easier to deal with.

- When a player becomes bankrupt, his or her properties are all returned to the bank, instead of auctioning them (which tends to reward the players who happen, by chance, to have a lot of spare cash at that particular time.) In later work we shall use a standard auction at this point instead, to check if this affects the behaviour. It is possible that, by using auctions, we might instead encourage strategies involving more prudent use of resources so that such events might be exploited more effectively.
- Chance and Community Chest cards were picked randomly with replacement, instead of remembering a random initial card order and cycling through these. 'Get out of jail' cards were tracked, and not replaced until used.
- There was no maximum on the number of houses or hotels allowed on the board simultaneously. It is not clear if this affected the strategies, avoiding the need for property strategies dealing with housing shortages.
- Games were limited to 500 turns. If there was no clear winner at this point, then the players were ranked by total net worth.
- Properties not purchased immediately were not auctioned, but remained unsold. See section 6 below for a discussion on this point.

Calculations were run on a 3GHz Pentium-IV machine. For simplicity, all games were started with four players. After careful optimisation, games could be simulated at the rate of approximately 400 per second. This project involved a total of over 377 million games of Monopoly®, one quarter of which (87 million) are included in the final results.

3 Single Population Results

As an initial test of the evolutionary algorithm approach, I generated a single population of individuals, and ran a

standard evolutionary algorithm, using a population size of 1000 and 24 hours of CPU time. This resulted in a total of 1420 generations completed. In each generation, 100 iterations were played. For each iteration, the 1000 individuals were selected off randomly into groups of four, and each group played one single game of Monopoly®. The game ended when there was only one player left, or after 500 turns. Points were awarded to the individuals based on their position in each game. First place was awarded 4 points, second place was awarded 2 points, third place 1 point and last place 0 points.

The fitness function, therefore, consisted of the sum of the points gathered by an individual over the 100 iterations in each generation.

At the end of each generation, the top three individuals survived by right as elites. 300 survivors were selected using a size 2 tournament selection algorithm with replacement. A further 300 individuals were selected in the same manner to continue to the next generation after undergoing random mutations. During a mutation, ten values from each of the property prices, street prices and house prices were randomly mutated using a Gaussian kernel of standard deviation 10% of the variable's value. All other values were mutated by the same amount with a probability of 50%.

The remainder of the next generation (397 individuals) were generated using a crossover between two tournament-selected parents from the current generation. During crossover, the child acquired each parameter randomly from either parent with equal weighting.

Previous experience with evolutionary algorithms has taught us that the solutions derived in most applications are not very sensitive to these values, and that the above values lie within sensible ranges. Alternative values for the elitism fraction, mutant fraction, crossover fraction etc. were not tested.

At the end of each generation, the best individual was tested in 1000 games against randomly generated opponents, with the same scoring system as detailed above. This was used as a check to ensure that the algorithm was indeed moving towards greater fitness. These testing figures are shown in figure 1. They show a very sharp rise over the first 20-30 generations, from an initial score of 2522. From generation 30-40 up to approximately generation 200, the test scores then slowly declined, before levelling off around 2930. The plots in this paper show the result from a single trial. However, multiple trials were performed during testing, with subtly different algorithmic details, and very similar results were achieved each time.

A speculative interpretation of this behaviour is linked to the manner in which strategies evolve within such a complex evolutionary environment. The population rapidly learned some strong, simple strategies which could be used to good effect against simple, randomly generated opponents. However, after the first few dozen generations, individuals began to develop counter-strategies which partly refuted these 'easy wins'. Because the opposition from other individuals in the population was now much higher, the simple strategies had to be abandoned, leading to worsening performance against random opponents, but better performance against other members of the population, against whom the fitness function was measured.

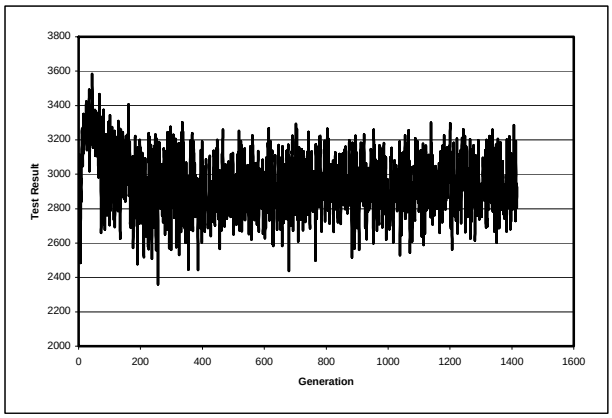


Figure 1 : Test score of best individual as a function of generation (single trial)

This style of learning is common to a wide variety of games. For example, a novice chess player might quickly learn some clever tricks (such as "fool's mate") by which he or she can defeat rather inexperienced players. By using such tricks, the novice begins to win a significant number of games because his or her opponents fall for these simple traps.

However, soon the opponents wise up to this strategy, and the novice player can no longer use the same tactics. In fact, these simple 'trap' openings often prove rather weak if the opponent knows how to deal effectively with them. If the novice then finds himself facing an unknown opponent then he will not use these same tricks any more, instead using more advanced opening strategies with which he may be far less confident. Against a good player, this will be a better strategy, but against a true novice, using a trick strategy might have given a better chance of winning.

After the end of the run, the best individual from each generation was studied in order to evaluate the degree of learning that had occurred. It was possible to examine how the estimated values of the individual properties and the various other numerical values used in the winning strategies had evolved over time.

Figure 2 shows the change in the estimated property value for "Old Kent Road", the square immediately after "Go" and the least valuable square on the board, according to face price. In Monopoly®, Old Kent Road is valued at £60, and the individual houses cost £50 to build once a player also owns Whitechapel Road, two squares further along.

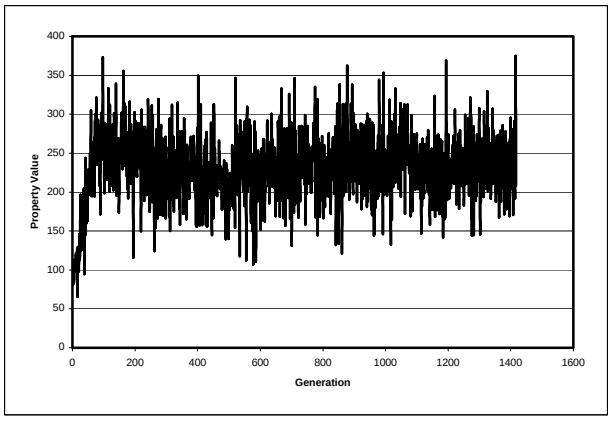


Figure 2 : Perceived value of Old Kent Road as a function of generation

By the end of the run, the average estimated worth of Old Kent Road, averaged over the last 100 generations, was £238, meaning that the evolutionary algorithm valued this property at a mark-up of approximately 297%. Whitechapel Road was valued at £290, or a mark-up of 383%. This property was valued slightly higher because it offers substantially more rental income, especially once developed. Based on individual rent prices, these properties were therefore valued on a forward per-earnings multiple of 119 times and 72.5 times respectively. Clearly, these prices would therefore only be worth paying if the player could expect to own the entire street and develop it with houses, or else prevent an opponent from doing the same.

Figure 3 shows a summary of the average property value for the ownable properties, versus their nominal face value. If a property is valued at less than its face value then the individual player will never purchase that property directly (though it might buy it in a trade from another player for a lesser amount). Note that every property along the lower and left-hand sides of the board appears to be undervalued, and almost all of the remaining properties appear to be overvalued, often enormously so. The only properties that the computer player would buy after "Free Parking" are the two stations (Fenchurch Street and Liverpool Street), the water works and the dark blue properties (Mayfair & Park Lane).

Figures 4 and 5 show the rapid reduction in perceived values of Strand (red property) and Bond Street (green property) over the simulation run. After 1420 generations, the algorithm values these two properties at £90 and £65, at a net discount to their face values of 59% and 80% respectively. During testing with smaller populations, or subtly different selection procedures and fitness functions, I obtained an extremely similar result every time.

For some reason the end result appears to be that the evolved Monopoly® players dislike the red, yellow and green streets. They will never buy a new property on any of these streets. The reason for this is difficult to discern, but it is such a pronounced effect that I suspect that it is due to the house cost for the upper and right-hand sides of the board. When houses cost £150 or £200 each then it gets very difficult to develop the red, yellow and green streets unless you are already winning by quite a considerable margin. And if you're already winning then you needn't bother developing new streets.

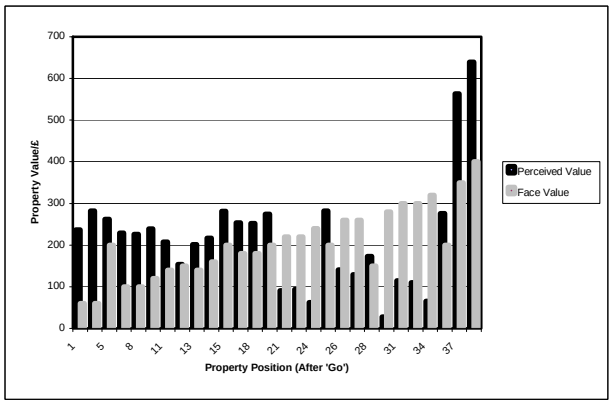


Figure 3 : Perceived value versus face value for all purchasable properties

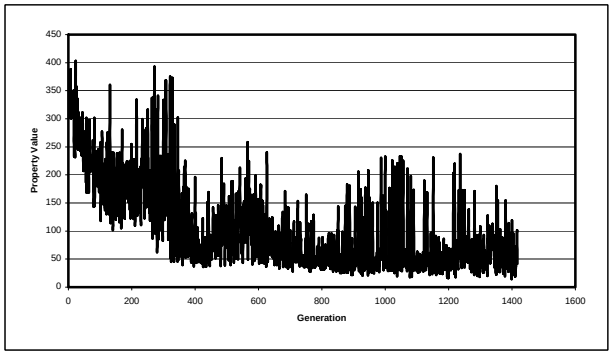


Figure 4 : Perceived value of Strand as a function of generation

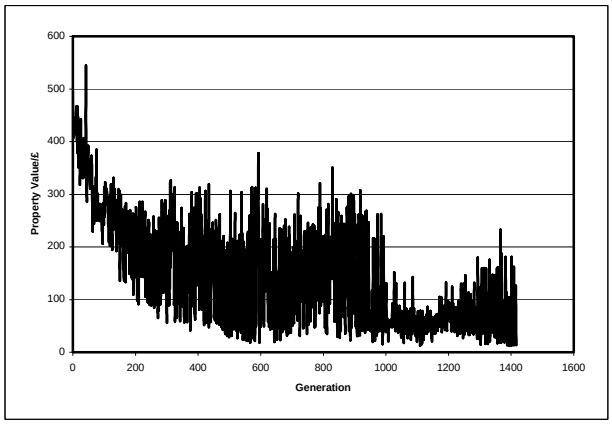


Figure 5 : Perceived value of Bond Street as a function of generation

So the conclusions drawn from this single population experiment seem to show that the best strategy is to gather

the lower value streets as quickly as possible, develop them rapidly and aim to win quickly.

4 Twin Population Results

In order to test these results, I implemented a multiple population approach to test to see whether the results from a single population strategy held up when two or more distinct populations evolved separately with only a very small trickle of individuals exchanging between them.

For this section, I implemented a two population approach with a migration rate of 0.5% at the end of every generation. Each population was set up exactly as the single population above (1,000 individuals, randomly seeded, fitness function and breeding as above). The simulation was run for 800 generations, and the results compared both between the two populations, and also back to the original single population.

The variations between the two parallel populations at the end of the simulation were found to be extremely minimal. When compared to the single population, the variations were slightly larger, but still the results were largely the same. Figure 6 shows the difference between the property valuations in the single population and multiple population runs. The differences between the two populations in the multiple-population simulation were so small that I have just plotted the first population results here.

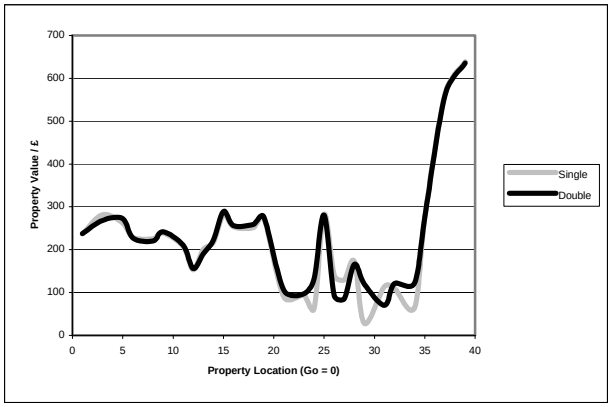


Figure 6 : Comparing final property prices between single population and multiple population simulations

The first thing to note about figure 6 is that the two simulations gave exactly the same results for all of the streets whose perceived value was greater than their face value. Figure 7 shows this feature.

In figure 7, the x-axis represents the ratio of perceived property value divided by the face value. Properties that were perceived to be undervalued on their face value are therefore towards the right in this diagram. Properties which were deemed less valuable than their face price (and therefore would not be bought) are at values less than one. The y-axis represents the logarithm of the disagreement between the perceived property value derived by the single-pop and 2-pop simulations, as a percentage of the single-pop perceived value.

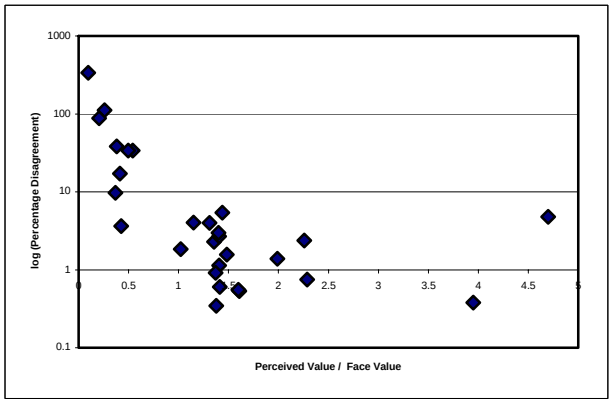


Figure 7 : Disagreement between single-pop and 2-pop simulations as a function of perceived property premium

This plot shows that the largest disagreements by far are caused by those properties whose perceived value was less than their face value. That is to say, those properties which the computer would never buy when it landed on them. In these cases, the perceived property value is useful only for bartering and dealing between players, and the likelihood of any player owning a street of this colour property would be very small. Thus, the variation in the perceived values was occasionally very high. However, for the properties whose perceived value was greater than the face value, the two simulations converged to remarkably similar estimates.

The other result to note is that the evolutionary algorithm, as expected, values the most expensive member of each colour group slightly higher than the other members of that colour group. The perceived value of station properties, all with a face value of 200 pounds, was also slightly variable. Marylebone was, as expected, the most valuable of the four, with an estimated value of £285 taken as the average of the single-pop and 2-pop simulation valuations. This is because there is a chance card moving players to Marylebone without choice. Next came Fenchurch St. At £281.40, King's Cross at £276.50 and finally Liverpool St. At £276.10. However, the variation between the prices here was not large, and there is insufficient evidence to suggest that these prices vary at all from a universal valuation of between £275 - £280.

5 Other strategies

Together with estimated house values, the genome for an individual also contained estimated price premiums for owning a property as a member of an entire street, and developing it with houses. Figure 8 shows the premium, that is the multiple of the basic perceived value, for owning a property as part of a street instead of singly.

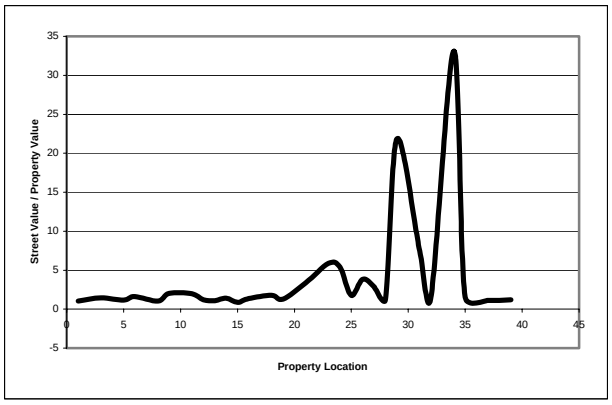


Figure 8 : Street ownership premium for all properties

As is clear from figure 8, all properties up to and including 19 (the lower and left-hand sides of the board) operate at very modest premiums to their individual value. However, the properties of higher face value (that is the red, yellow and green streets, though not the dark blue street, the stations or the water works) have a much higher street value compared to their individual value. This is particularly striking for the most expensive properties in these three zones, namely Trafalgar Square, Piccadilly and Bond Street, which operate at 5, 21 and 33 times their individual values respectively.

This inflated value for these properties as a member of streets reflects their apparently low value as single properties. The evolutionary algorithm grows to dislike these properties, though obviously once two are acquired, it becomes favourable to gain the third. As the chance of the algorithm acquiring two of the properties is very low, this adaptation is probably more of a defensive measure rather than anything else – stopping opponents from gaining the streets rather than aiming to build on those properties itself.

The final set of values used in the genome concern particular financial strategies necessary for accurate play. These values varied enormously between the runs, tending to hint that they were largely irrelevant to the overall performance of any one individual. However, a few rather general conclusions could be drawn.

(1) It is wise to retain approximately 110-150 pounds in your bank account as a bare minimum. Add to this approximately 5% of the net worth of the strongest opponent.
(2) The value affecting the minimum amount of money to retain is much more strongly linked to the net worth of the strongest opponent than to the average or total net worth of the players on the board.
(3) It is almost always a wise idea to pay to get out of jail. If you get to the point when you are staying in jail to avoid paying rent then you've probably lost anyway! However, there are also times when staying in jail can allow you to collect considerable rent from opponents landing on your properties, without the risk of you yourself being fined.
(4) Avoid accepting a mortgaged property for a trade unless it makes up a new complete street.

6 Conclusions

This genetic algorithm approach to playing Monopoly® has given a variety of insights into the game. Much of what the simulations discovered has been known for some time, though it is always reassuring to confirm this. However, some strategies are completely new.

In most games, landing on *any* property with a hotel will cause a considerable dent in a player's net worth. Doing this twice will probably spell the end of the game. Therefore, it makes sense to concentrate on the properties that are cheapest to develop, so that you can reach a high level of rent-gathering as rapidly as possible.

For example, for the red properties, reaching the level where you can charge an average rent of nearly £300 would cost £1580 (purchasing all three properties, plus two houses on each – average rent £267) . For the brown properties, this only costs £620 (buying two properties and a hotel on each – average rent £350).

With the orange properties – which are the most frequently visited on the board – £1460 can buy you all three properties, plus three houses on each – charging an average rent of £567 pounds. Not only is this £120 cheaper than developing the red properties as above, but it also gives a rent of well over twice the amount. Moreover, these properties are more frequently visited, therefore making the developed orange properties a vastly superior investment. A fine of 567 pounds would considerably dent all but the strongest of opponents.

In addition to the property valuation strategy, three further tips arose from the best evolved strategies.

Firstly, always retain a small reserve of cash to stop you from mortgaging properties. Mortgaging can be useful, but ultimately you are stifling a revenue source, which tends to drop you further back in the game. The penalties derived for mortgaging were very steep – with mortgaged properties sometimes worth as little as 2% - 3% of their perceived un-mortgaged value.

Secondly, don't be afraid to make bids for opponent properties. Human beings often vastly undervalue the cheaper streets – giving an astute player a certain strategic advantage if he or she can initiate a favourable trade. Single, expensive properties can be very useful indeed if they are traded for less expensive properties, even at a substantial concession to their face value.

Thirdly, don't be a coward and stay in jail – fortune favours the bold! Saving 50 pounds in the short term could well cost you the opportunity to pick up on a vital deal later on.

One potential extension to this study is to vary the maximum game length. Setting this well below the

expected survival time for the three losing players (say, 50 turns) would encourage strategies which accumulated wealth very rapidly, but perhaps not in a stable way. This is well worth investigating. Figure 9 shows that most games are either complete by approximately turn 200, or last the full 500 turns. Any game surviving several hundred turns is likely to be in one of two states: either (1) oscillations in power between two or more players, so that the eventual winner is largely random or (2) a stalemate where no player owns any streets nor wants to sell any. It is likely that reducing the maximum game length to, say, 250 turns will not greatly affect the results. However, as the overwhelming majority of games are completed by this stage, the saving in CPU time will be barely noticeable.

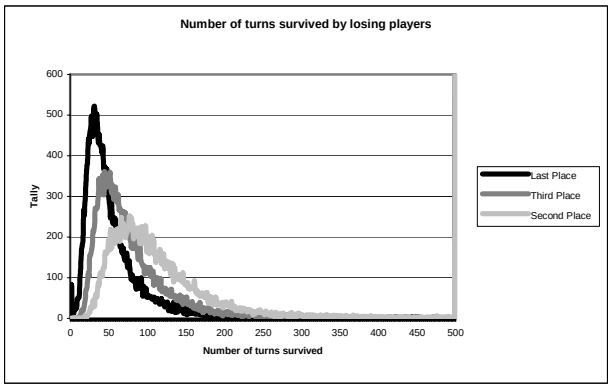


Figure 9 : Average survival period for the losing players in one generation. Note the large spike at 500 turns for games which lasted the full maximum duration.

As an extension, I am investigating the effects of completing the implementation of the full realistic rules. During this study, I used a simplified subset of the rules as I believed that any mild affect on the strategies developed would be more than generously offset by the reduced programming complexity and the increased number of generations that could be run.

Subsequent study has tentatively suggested that the introduction of a more realistic rule set might affect the rules more strongly than I had predicted. Most importantly, the introduction of a full auction system appears partially to prevent the perceived reduction in value for the more expensive properties, and also tends to shift all perceived values upwards. However, such an implementation slows the game speed down considerably, and reduces the learning rate, so a considerable amount more processing time is required in order to investigate this effect more thoroughly. After careful optimisation, and using most of the full rules, the games are running five times slower than reported in the present work. I hope to release a follow-up paper in the future investigating the effects of these changes.

Acknowledgments

I acknowledge Hasbro, Inc. who own the trademarks and publishing rights for the game of Monopoly®. I also acknowledge Advantage West Midlands, who supplied the funding for my research position and for the formation of the Centre of Excellence for Research in Computational Intelligence and Applications.

Bibliography

Ash, Robert B., and Bishop, Richard L. 1972. "Monopoly® as a Markov process." *Mathematics Magazine* 45:26-29.

Stewart, Ian. 1996. "Monopoly® revisited." *Scientific American* 275:116-119

Stewart, Ian. 1996. "How fair is Monopoly®?" *Scientific American* 274:104-105

Further Evolution of a Self-Learning Chess Program

David B. Fogel
Timothy J. Hays
Sarah L. Hahn
James Quon
Natural Selection, Inc.
3333 N. Torrey Pines Ct., Suite 200
La Jolla, CA 92037 USA
dfogel@natural-selection.com

Abstract- Previous research on the use of coevolution to improve a baseline chess program demonstrated a performance rating of 2550 against *Pocket Fritz 2.0* (PF2). A series of 12 games (6 white, 6 black) was played against PF2 using the best chess program that resulted from 50 generations of variation and selection in self-play. The results yielded 9 wins, 2 losses, and 1 draw for the evolved program. This paper reports on further evolution of the best-evolved chess program, executed through 7462 generations. Results show that the outcome of this subsequent evolution was statistically significantly better than the prior player from 50 generations. A 16-game series against PF2, which plays with the rating of a high-level master, resulted in 13 wins, 0 losses, and 3 draws, yielding a performance rating of approximately 2650.

1 Introduction and Background

As noted in [1], chess has served as a testing ground for efforts in artificial intelligence, both in terms of computers playing against other computers, and computers playing against humans for more than 50 years [2-9]. There has been steady progress in the measured performance ratings of chess programs. This progress, however, has not in the main arisen because of any real improvements in anything that might be described as "artificial intelligence." Instead, progress has come most directly from the increase in the speed of computer hardware [10], and also straightforward software optimization.

Deep Blue, which defeated Kasparov in 1997, evaluated 200 million alternative positions per second. In contrast, the computer that executed *Belle*, the first program to earn the title of U.S. master in 1983, searched up to 180,000 positions per second. Faster computing and optimized programming allows a chess program to evaluate chessboard positions further into the prospective future. Such a program can then select moves that are expected to lead to better outcomes, which might not be seen by a program running on a slower computer or with inefficient programming.

Standard chess programs rely on a database of opening moves and endgame positions, and generally use a polynomial function to evaluate intermediate positions. This function usually comprises features regarding the values assigned to individual pieces (material strength), mobility, tempo, and king safety, as well as tables that are used to assign values to pieces based on their position (positional values) on the chessboard. The parameters for these features are set by human experts, but can be improved upon by using an evolutionary algorithm. Furthermore, an evolutionary algorithm can be employed to discover features that lead to improved play.

Research presented in [1] accomplished this using an evolutionary program to optimize material and positional values, supplemented by three artificial neural networks that evaluated the worth of alternative potential positions in sections of the chessboard (front, back, middle), as shown in Figure 1. Following similar work in [10], the procedure started with a population of alternative simulated players, each initialized to rely on standard material and positional values taken from open source chess programs, supplemented with the three neural networks. The simulated players then competed against each other for survival and the right to generate "offspring" through a process of random variation applied to the standard parameters and neural connection weights.

Survival was determined by the quality of play in a series of chess games played against opponents from the same population. Over successive generations of variation and selection, the surviving players extracted information from the game and improved their performance. At the suggestion of Kasparov [11], the best-evolved player after 50 generations was tested in simulated tournament conditions in 12 games (6 as black, 6 as white) against *Pocket Fritz 2.0*. This is a standard chess program that plays at a rating of 2300-2350 (high-level master) [11, and also as assessed by nationally ranked master and co-author Quon]. The evolved player won this contest with nine wins, two losses, and one draw.

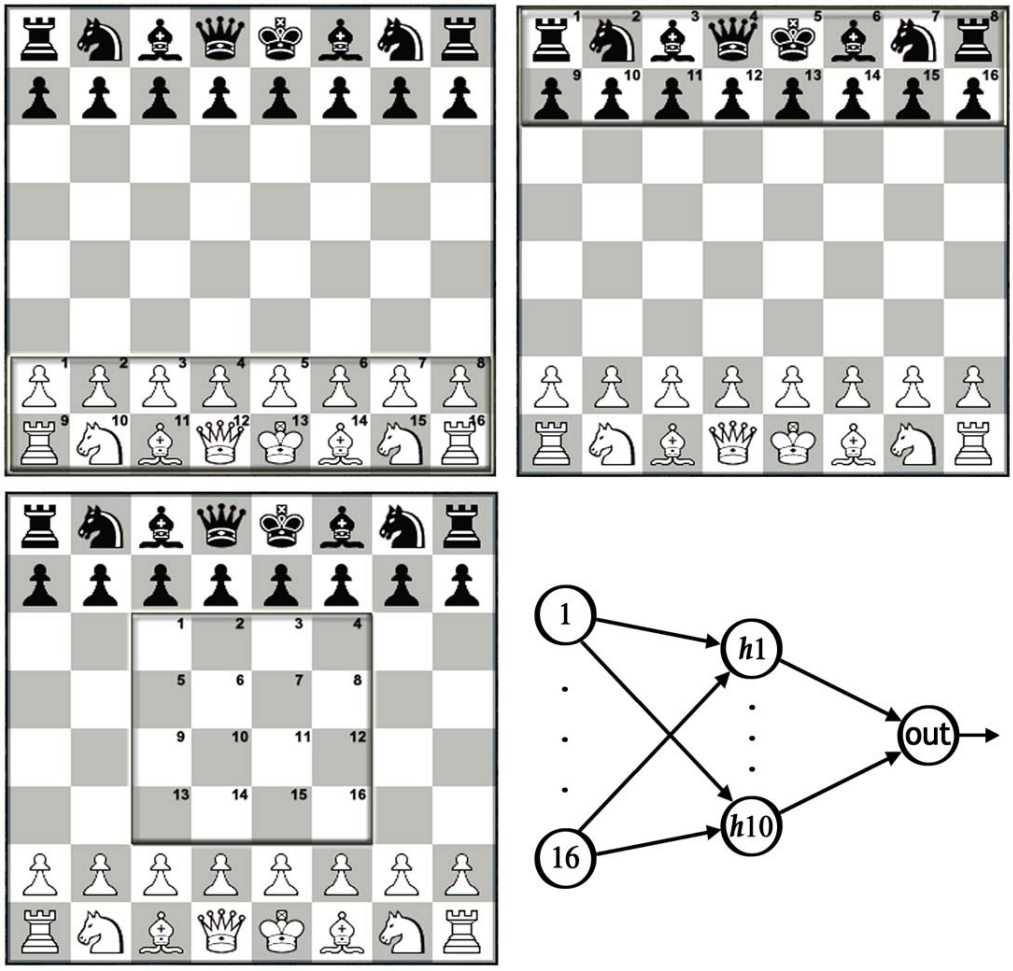


Figure 1. The three chessboards indicate the areas (front, back, middle) in which the neural networks focused attention, respectively. The upper-left panel highlights the player's front two rows. The 16 squares as numbered were used for inputs to a neural network. The upper-right panel highlights the back two rows, and the contents were similarly used as input for a neural network. The lower-left panel highlights the center of the chessboard, which was again used as input for a neural network. Each network was designed as shown in the lower-right panel, with 16 inputs (as numbered for each of the sections), 10 hidden nodes (h1-h10), and a single output node. The bias terms on the hidden and output are not shown. Neural networks that focus on particular items or regions in a scene are described as *object neural networks*.

Over a period of nearly six months, additional evolution was applied starting with the best-evolved chess player from [1]. After 7462 generations (evolution interrupted by a power failure), further testing was conducted on the new best-evolved player. The results of playing against a nonevolved baseline player and also against *Pocket Fritz 2.0* are reported here. The next section summarizes (and at times repeats) the methods of [1], and readers who would like additional details should refer to [1] directly.

2 Method

A chess engine was provided by Digenetics, Inc. and extended for the current and prior experiments of [1]. The baseline chess program functioned as follows. Each chessboard position was represented by a vector of length 64, with each component in the vector corresponding to an available position on the board. Components in the vector could take on values from {$-K$, $-Q$, $-R$, $-B$, $-N$, $-P$, 0, $+P$, $+N$, $+B$, $+R$, $+Q$, $+K$}, where 0 represented an empty square and the variables P, N, B, R, Q, and K represented material values for pawns, knights, bishops, rooks, and the queen and king, respectively. The chess engine assigned a material value to kings even though the king could not actually be captured during a match. The sign of the value indicated whether or not the piece in question belonged to the player (positive) or the opponent (negative).

A player's move was determined by evaluating the presumed quality of potential future positions. An evaluation function was structured as a linear combination of (1) the sum of the material values attributed to each player, (2) values derived from tables that indicated the worth of having specific pieces in specific locations on the board, termed "positional value tables" (PVTs), and (3) three neural networks, each associated with specific areas of the chessboard. Each piece type other than a king had a corresponding PVT that assigned a real value to each of the 64 squares, which indicated the presumptive value of having a piece of that type in that position on the chessboard. For kings, each had three PVTs: one for the case before a king had castled, and the others for the cases of the king having already castled on the kingside or queenside. The PVTs for the opponent were the mirror image of the player's PVTs (i.e., rotated 180 degrees). The entries in the PVTs could be positive and negative, thereby encouraging and discouraging the player from moving pieces to selected positions on the chessboard. The nominal (i.e., not considering the inclusion of neural networks) final evaluation of any position was the sum of all material values plus the values taken from the PVTs for each of the player's own pieces (as well as typically minor contributions from other tables that were used to assess piece mobility and king safety for both sides). The opponent's values from the PVTs were not used in evaluating the quality of any prospective position.

Games were played using an alpha-beta minimax search of the associated game tree for each board position looking a selected number of moves into the future (with the exception of moves made from opening and endgame databases). The depth of the search was set to four ply to allow for reasonable execution times in the evolutionary computing experiments (as reported in [1], 50 generations on a 2.2 GHz Celeron with 128MB RAM required 36 hours).

The search depth was extended in particular situations as determined by a quiescence routine that checked for any possible piece captures, putting a king in check, and passed pawns that had reached at least the sixth rank on the board (anticipating pawn promotion), in which case the ply depth was extended by two. The best move to make was chosen by iteratively minimizing or maximizing over the leaves of the game tree at each ply according to whether or not that ply corresponded to the opponent's move or the player's. The games were executed until one player suffered checkmate, upon which the victor was assigned a win and the loser was assigned a loss, or until a position was obtained that was a known draw (e.g., one king versus one king) or the same position was obtained three times in one game (i.e., a three-move rule draw), or if 50 total moves were exceeded for both players. (This should not be confused with the so-called 50-move rule for declaring a draw in competitive play.) Points were accumulated, with players receiving +1 for a win, 0 for a draw, and –1 for a loss.

2.1 Initialization

The evolutionary experiment in [1] was initialized with a population of 20 computer players (10 parents and 10 offspring in subsequent generations) each having nominal material values and entries in their PVTs, and randomized neural networks. The initial material values for P, N, B, R, Q, and K were 1, 3, 3, 5, 9, and 10000, respectively. The king value was not mutable. The initial entries in the PVTs were in the range of –50 to +40 for kings, –40 to +80 for queens and rooks, –10 to +30 for bishops and knights, and –3 to +5 for pawns, and followed values gleaned from other open source chess programs.

Three object neural networks (front, back, middle, see Figure 1) were included, each being fully connected feedforward networks with 16 inputs, 10 hidden nodes, and a single output node. The choice of 10 hidden nodes was arbitrary. The hidden nodes used standard sigmoid transfer functions $f(x) = 1/(1 + \exp(-x))$, where x was the dot product of the incoming activations from the chessboard and the associated weights between the input and hidden nodes, offset by each hidden node's bias term. The output nodes also used the standard sigmoid function but were scaled in the range of [–50, 50], on par with elements of the PVTs. The outputs of the three neural networks were added to the material and PVT values to

come to an overall assessment of each alternative board position. All weights and biases were initially distributed randomly in accordance with a uniform random variable $U(-0.025, 0.025)$ and initial strategy parameters were distributed $U(0.05)$.

Candidate strategies for the game were thus represented in the population as the material values, the PVT values, the weights and bias terms of the three neural networks, and associated self-adaptive strategy parameters for each of these parameters (3,159 parameters in total), explained as follows.

2.2 Variation

One offspring was created from each surviving parent by mutating all (each one of) the parental material, PVT values, and weights and biases of all three neural networks. Mutation was implemented on material and positional values, and the weights of the neural networks, according to standard Gaussian mutation with self-adaptation using a single scaling value $\tau = 1/\text{sqrt}(2n)$, where there were n evolvable parameters (see [1]). The material value and PVT strategy parameters were set initially in [1] to random samples from $U(0, 0.05)$, and were initialized in the new experiments reported here to be the values of the best-evolved player from [1]. In the case where a mutated material value took on a negative number, it was reset to zero.

2.3 Selection

Competition for survival was conducted by having each player play 10 games (5 as white and 5 as black) against randomly selected opponents from the population (with replacement, not including itself). The outcome of each game was recorded and points summed for all players in all games. After all 20 players completed their games, the 10 best players according to their accumulated point totals were retained to become parents of the next generation.

2.4 Experimental Design

A series of 10 independent trials was conducted in [1], each for 50 generations using 10 parents and 10 offspring. The best result of each trial was tested in 200 games against the nonevolved baseline player. All ten trials favored the evolved player over the nonevolved player (sign-test favoring the evolved player, $P < 0.05$), indicating a replicable result. The complete win, loss, and draw proportions over the 2000 games were 0.3825, 0.2390, and 0.3785, respectively. Thus the win-loss ratio was about 1.6, with the proportion of wins in games decided by a win or loss being 0.6154. The best player from the eighth trial (126 wins, 45 losses, 29 draws) was tested in tournament conditions against *Pocket Fritz 2.0* (rated 2300-2350, high-level master) and in 12 games (6 white, 6 black) scored 9 wins, 2 losses, and 1 draw. This corresponded to a performance rating of about 2550, which is commensurate with a grandmaster.[1]

3 Results of Further Evolution

For a period of six months, the evolutionary program was allowed to continue iterating its variation and selection algorithm, until a power outage halted the experiment after 7462 generations. Ten players were selected in an ad hoc manner from the last 20 generations of evolution and were tested in 200 games each against the original nonevolved player. The results are shown in Table 1.

Table 1. Results of 200 games played with each of 10 sampled players from the last 20 generations of the subsequent evolution against the nonevolved player.

Wins	Losses	Draws
89	37	74
92	45	63
109	49	42
127	32	41
126	34	40
122	46	32
108	52	40
82	46	72
78	48	74
79	52	69

The total win, loss, and draw proportions were 0.506, 0.2205, and 0.2735, respectively. The proportion of wins in games that ended in a decision was 0.6964. A proportion test comparing this result to the prior result of 0.6154 shows statistically significant evidence ($P << 0.05$) that these players improved over the results from 50 generations in 10 trials.

Following the prior suggestion of Kasparov [11] the best-evolved program from the ad hoc sample (trial #4) was tested (using an Athlon 2400+/256MB) against *Pocket Fritz 2.0* under simulated tournament conditions, which provide 120 minutes for the first 40 moves, 60 minutes for the next 20 moves, and an additional 30 minutes for all remaining moves. Unlike *Pocket Fritz 2.0* and other standard chess programs, the evolved player does not treat the time per move dynamically. The time per move was prorated evenly across the first 40 moves after leaving the opening book, with 3 minutes per move allocated to subsequent moves. *Pocket Fritz 2.0* was executed on a "pocket PC" running at 206MHz/64MB RAM, with all computational options set to their maximum strength, generating an average base ply depth of about 11.

A series of 16 games was played, with the evolved program playing 8 as black and 8 as white. The evolved program won 13, lost none, and drew 3. The results

[1] Earning a title of grandmaster requires competing against other qualified grandmasters in tournaments.

provide evidence to estimate a so-called "performance rating" of the evolved player under tournament settings at approximately 2650, about 325 points higher than *Pocket Fritz 2.0*, and improves on the performance rating of about 2550 earned in [1]. For additional comparison, a series of 12 games with the nonevolved baseline chess program against *Pocket Fritz 2.0* in the same tournament conditions yielded 4 wins, 3 losses, and 5 draws for a performance rating that is on par with *Pocket Fritz 2.0*.

4 Conclusions

The approach adopted in this research, following [10], relies on accumulating payoffs over a series of games in each generation. Selection is based only on the overall point score earned by each simulated player, not on the result of any single game. Indeed, the players do not have any concept of which games were won, lost, or drawn.

In 1961 [12], Allen Newell was quoted offering that there is insufficient information in "win, lose, or draw" when referred to an entire game of chess or checkers to "provide any feedback for learning at all over available time scales." Research presented in [1], [10], [13], [14], and now here, shows conclusively that not only was this early conjecture false, but it is possible to learn how to play these games at a very high level of play even without knowing which of a series of games were won, lost, or drawn, let alone which individual moves were associated with good or bad outcomes.

In addition, the approach utilizes only a simple form of evolutionary algorithm with a small population, Gaussian mutation, and no sophisticated variation operations or representation. The use of the neural networks to focus on subsections of the board, coupled with positional value tables, and opening and endgame databases, provides more upfront expertise than was afforded in prior Blondie24 checkers research [10]; however, when compared to the level of human chess expertise that is relied on in constructing typical chess programs, the amount of knowledge that is preprogrammed here is relatively minor.

All performance ratings that are based on a relatively small sample of games have an associated high variance. (Note that programs rated at [15] have a typical variation of plus or minus 25 points when testing in about 1000 games.) Yet, the performance rating of the best-evolved chess player based on 16 games against *Pocket Fritz 2.0* is sufficiently encouraging to both continue further evolution and also seek to measure the program's performance against another program of world-class quality (e.g., as rated on [15]). In addition, the level of play may be improved by including additional opportunities for evolution to learn how to assess areas of the chessboard or the interaction between pieces in formations.

Acknowledgments

The authors thank Digenetics, Inc. for use of its chess game engine, Garry Kasparov for comments on our earlier research, and the anonymous reviewers for helpful criticisms. Portions of this paper were reprinted or revised from [1] in accordance with IEEE Copyright procedures. This work was sponsored in part by NSF Grants DMI-0232124 and DMI-0349604. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation (NSF).

Bibliography

1. Fogel, D.B., Hays, T.J., Hahn, S.L., and Quon, J. (2004) "An Evolutionary Self-Learning Chess Program," *Proceedings of the IEEE*, December, pp. 1947-1954.
2. Shannon, C.E. (1950) "Programming a Computer for Playing Chess," *Philosophical Magazine*, Vol. 41**,** pp. 256-275.
3. Turing, A.M. (1953) "Digital Computers Applied to Games," in *Faster than Thought*, B.V. Bowden, Ed., London: Pittman, pp. 286-310.
4. Newell, A, Shaw, J.C., and Simon, H.A. (1958) "Chess-Playing Programs and the Problem of Complexity," *IBM J. Res. Dev*., Vol. 2, pp. 320-325.
5. Levy, D.N.L. and Newborn, M. (1991) *How Computers Play Chess*, New York: Computer Science Press, pp. 28-29, 35-39.
6. Cipra, B. (1996) "Will a Computer Checkmate a Chess Champion at Last?" *Science*, Vol. 271, p. 599.
7. McCarthy, J. (1997) "AI as Sport," *Science*, Vol. 276, pp. 1518-1519.
8. Markman, A.B. (2000) "If You Build It, Will It Know?" *Science*, Vol. 288, pp. 624-625.
9. Holden, C. (2002) "Draw in Bahrain," *Science*, Vol. 298, p. 959.
10. Fogel, D.B. (2002) *Blondie24: Playing at the Edge of AI*, Morgan Kaufmann, San Francisco.
11. Kasparov, G. (2004) personal communication.
12. Minsky, M. (1961) "Steps Toward Artificial Intelligence," *Proc. IRE*, Vol. 49, pp. 8-30.
13. Chellapilla, K. and Fogel, D.B. (1999) "Evolution, Neural Networks, Games, and Intelligence," *Proc. IEEE*, Vol. 87, pp. 1471-1496.
14. Chellapilla, K. and Fogel, D.B. (2001) "Evolving an Expert Checkers Playing Program Without Using Human Expertise," *IEEE Transactions on Evolutionary Computation*, Vol. 5, pp. 422-428.
15. The Swedish Chess Computer Association publishes ratings of the top 50 computer programs at http://w1.859.telia.com/~u85924109/ssdf/list.htm

Combining coaching and learning to create cooperative character behavior

Jörg Denzinger
Department of Computer Science
University of Calgary
denzinge@cpsc.ucalgary.ca

Chris Winder
Department of Computer Science
University of Calgary
winder@cpsc.ucalgary.ca

Abstract- We present a concept for developing cooperative characters (agents) for computer games that combines coaching by a human with evolutionary learning. The basic idea is to use prototypical situation-action pairs and the nearest-neighbor rule as agent architecture and to let the human coach provide key situations and his/her wishes for an associated action for the different characters. This skeleton strategy for characters (and teams) is then fleshed out by the evolutionary learner to produce the desired behavior. Our experimental evaluation with variants of Pursuit Games shows that already a rather small skeleton –that alone is not a complete strategy– can help solve examples that learning alone has big problems with.

1 Introduction

From the first computer games on, there have been games that confront the game player with not just one opponent (i.e. "the computer") but several entities in the game that, according to design, act as opponents (although from the game implementation perspective, there often was only one opponent with many "bodies"). Later, the human player has also become responsible for several "characters", be it the players of a team sports game or a party of adventurers in computer versions of role-playing games. Naturally, under sole human control, the characters do not act together as well as a computer controlled opponent could direct them.

Nowadays, computer games offer a user several fixed character control scripts from which he or she can choose and naturally each character can also be taken over by the user for a more fine-tuned strategy. But both computer controlled opponents and side-kicks for a user-controlled character are far away from acting on a human-like level, in fact, often they are acting less intelligent than a pet and require from a human player a lot of skill in jumping the control between the characters of the team to execute the strategy the player has developed.

In this paper, we present an approach that combines techniques for learning cooperative behavior for agents with the possibility for a user to develop (and tell the agent) some basic ideas for solving a problem. This combination allows us to overcome the basic drawbacks of the two individual ideas, namely the problem that players are not programmers, so that we cannot expect them to write their own character control scripts, and the fact that automated learning of behavior requires providing the learner with a lot of experiences, too many experiences for most human players to "endure". In addition, we see our combined approach also as a good method for game developers to develop more complex, and possibly more human-like, scripts for their non-player characters.

Our approach is based on the evolutionary learning approach for cooperative behavior of agents presented in [DF96], [DE02], [DE03a], and [DE03b]. The basic agent architecture used in these papers are sets of prototypical situation-action pairs and the nearest-neighbor rule for action selection. In any given situation, an agent uses a similarity measure on situations to determine the situation-action pair in its set that is most similar to the situation it faces and it then performs the action indicated by this pair. This resembles the drawing board a coach uses to explain to his/her players game moves he/she wants them to perform. The obvious advantages of such an architecture are that there will always be an action indicated to an agent and that there is quite some resistance to "noise" in the architecture. If an agent's position is a little bit away from a particular position this will in most cases be tolerated by the use of a similarity measure (if being a bit away is not crucial). Additionally, situation-action-pair sets have proven to be learnable by evolutionary algorithms.

The basic idea of our approach is to let the user define basic and key situations and the necessary or wanted actions for all characters involved on his/her side. Then we use the evolutionary learning approach to add to this skeleton of a strategy additional situation-action pairs that are needed to make the user's idea work. Our experimental evaluation shows that rather small and obvious skeletons combined with learning can solve problems that learning alone is incapable to solve (or at least has big problems with). And the used skeletons alone were not able to solve these examples.

2 Basic definitions

The characters of a computer game interact within the game or specific parts of it. As such, the characters can be seen as agents of a multi-agent system with the game being the environment in which these agents are acting. The area of multi-agent systems has established itself over the last two decades and we will be using its terminology to present our ideas. Obviously, the key component of a multi-agent system are agents. There is no agreed-upon definition of what an agent is, since there are a lot of properties of agents that different people are interested in and consequently want them to be covered by their agent definitions.

On a very abstract level, an agent $\mathcal{A}g$ can be described by three sets Sit, Act, Dat and a function $Dat \rightarrow Act$. The set Sit describes the set of situations $\mathcal{A}g$

can be in (according to $\mathcal{A}g$'s perceptions), Act is the set of actions $\mathcal{A}g$ can perform and Dat is the set of all possible values of $\mathcal{A}g$'s internal data areas. Internal data areas is our term for all variables and data structures representing $\mathcal{A}g$'s knowledge. This knowledge can include $\mathcal{A}g$'s goals, $\mathcal{A}g$'s history and all other kinds of data that $\mathcal{A}g$'s designer wants it to store. The function $f_{\mathcal{A}g}$, $\mathcal{A}g$'s *decision function*, takes a situation and a value of its internal data areas and produces the action that $\mathcal{A}g$ will take under these circumstances. It should be noted that $f_{\mathcal{A}g}$ can be rather primitive, looking up actions in a table based on situations only, for example, but also very complex involving a lot of complex inferences based on the current value from Dat, for example, to form complex plans.

In a multi-agent setting, usually the elements of the three sets from above can be more structured to reflect the fact that several agents share the environment. Each situation description in Sit will usually have a part dealing with other agents and a part dealing with the rest of the environment. The set Act will contain a subset of actions aimed at communicating and coordinating with other agents and a subset of actions not related to other agents (like the agent moving around). If an agent can perform several actions at the same time, often such "combined" actions have a communication part and a part not related to other agents (often there will also be a part manipulating the internal data areas). Finally, due to the unpredictability introduced by having several agents, the knowledge in the internal data areas is often divided into sure knowledge about other agents, assumptions about other agents and, naturally, the knowledge the agent has about itself (with the latter often being structured much more detailed).

The particular agent architecture we are using for our work are prototypical (extended) situation-action pairs (SAPs) together with a similarity measure on situations applied within the Nearest-Neighbor Rule (NNR) that realizes our decision function. More precisely, an element d of the set Dat of our agents consists of a part d_{SAP} and a part $d_{Rest} \in Dat_{Rest}$ (i.e. $d = d_{SAP}d_{Rest}$), where d_{SAP} is a set of pairs (s',a). Here s' denotes an extended situation and $a \in Act$. An extended situation s' consists of a situation $s \in Sit$ and a value $d_{Rest} \in Dat_{Rest}$ (this allows us to bring the current value of Dat into the decision process; note that not a full "state" from Dat_{Rest} is required, often only some parts of a d_{Rest} are used or no part of such an element at all). So, by changing the d_{SAP}-value of an agent, we can influence its future behavior, and even more, the agent can also change its d_{SAP}-value itself.

For determining what action to perform in a situation s with Dat-value $d_{SAP}d_{Rest}$, we need a similarity measure sim that measures the similarity between sd_{Rest} and all the extended situations s'_1,...,s'_m in d_{SAP}. If $sim(sd_{Rest},s'_i)$ is maximal, then a_i will be performed (if several extended situations in d_{SAP} have a maximal similarity, then the one with lowest index is chosen; in contrast to rule-based systems, no other conflict management is necessary). Naturally, it depends on the particular application area how these general concepts are instantiated.

If we have an agent $\mathcal{A}g$ based on a set of (extended) SAPs, then we can define this agent's behavior $\mathcal{B}$ as follows: if $s_0d_{Rest_0}$ is the extended situation from which the agent starts, then $\mathcal{B}(\mathcal{A}g,s_0d_{Rest_0})$ = $s_0d_{Rest_0}$,sap$_1$,$s_1d_{Rest_1}$,sap$_2$, ... ,$s_{i-1}d_{Rest_{i-1}}$,sap$_i$,$s_id_{Rest_i}$,..., where sap$_j$ is an element in $d_{SAP_{j-1}}$, i.e. the set of SAPs that guided the agent at the time it made the decision. Naturally, $s_jd_{Rest_j}$ is the extended situation that is the result of $\mathcal{A}g$ applying the action associated with sap$_j$ and *of the actions of all other agents after s_{j-1} was observed by $\mathcal{A}g$*. The SAPs in the behavior of an agent can be seen as the justifications for its actions. Note that for other agent architectures we might use different justifications, but describing the behavior of an agent by sequences of (extended) situations that are influenced by others is a rather common method.

3 Evolutionary learning with SAPs

As stated in the previous section, SAPs together with NNR provide a good basis for learning agents, since the strategy of an agent can be easily manipulated by deleting and/or adding SAPs. Note that one SAP in a set of SAPs can cover a lot of (extended) situations. We found the use of an evolutionary algorithm, more precisely a Genetic Algorithm for individuals that consist of a set of elements, a very good way to perform learning of a good strategy for a given task (see [DF96]).

The general idea of evolutionary learning of strategies for agents in our case is to start with random strategies, evaluate them by employing them on the task to solve (or a simulation of it) and then to breed the better strategies in order to create even better ones until a strategy is found that performs the given task. It is also possible to continue the evolutionary process to hopefully get even better strategies and so to find the optimal strategy for a task or at least a very good one (since evolutionary algorithms usually cannot guarantee to find the optimal solution to a problem). The crucial points of this general scheme are how a strategy is represented and how the performance of an agent or an agent team is measured. Genetic Operators and their control depend on these two points.

Following [DF96], we have chosen to have an individual of our Genetic Algorithm representing the strategies of all agents in a team that are supposed to be learning agents. More formally, an individual $\mathcal{I}$ has the form $\mathcal{I}$ = ({sap$_{11}$,...,sap$_{1m_1}$},...,{sap$_{n1}$,...,sap$_{nm_n}$}) for a team with n learning agents $\mathcal{A}g_1$,...,$\mathcal{A}g_n$. Then {sap$_{j1}$,...,sap$_{jm_j}$} will be used as the d_{SAP}-value of agent $\mathcal{A}g_j$.

This representation of an individual already implies that the performance evaluation has to be on the agent team level. For this evaluation, i.e. the fitness of an individual, we want to measure the success of the team in every step of its application to the task to solve. More precisely, due to the dependence on the particular application task, we need a function $\delta : Sit \rightarrow \mathbb{N}$ that measures how far from success a particular situation is. To define the fitness fit_run of an evaluation run for an individual $\mathcal{I}$, we sum up the δ-value of each situation encountered by the agent team. More precisely, if $\mathcal{A}g_j$ is one of the learning agents in the team

and $\mathcal{B}(\mathcal{A}g_j,s_0d_{Rest_0})$ = $s_0d_{Rest_0}$,sap$_1$,...,sap$_k$,$s_kd_{Rest_k}$ is the behavior of this particular agent in the evaluation run, then

$$fit\_run(\mathcal{I},\mathcal{B}(\mathcal{A}g_j,s_0d_{Rest_0})) = \sum_{i=1}^{k}\delta(s_i)$$

If we assume that all our agents have the same perception of the situations the team encounters, then just using $\mathcal{B}(\mathcal{A}g_j,s_0d_{Rest_0})$ for one agent is enough for defining fit_run. If the success of a team contains some requirements on the value of the internal data areas of the agents (i.e. on the value from their Dat_{Rest} sets), then we can extend fit_run to take this into account by extending δ (that then has to look at some kind of "extended" extended situations, i.e. extended situations for all agents) and naturally by looking at the behavior of all agents. This can be of a certain interest in the context of using the evolutionary learning together with the coaching we present in Section 4 to develop characters for computer games, since a developer naturally is interested in the internal data areas of a character/agent and wants to use the values of these areas.

If the task to solve does not involve any indeterminism, that might be due to other agents interfering with the agent team or other random outside influences, then the fit_run-value of a single evaluation run can already be used as the fitness-value for the individual $\mathcal{I}$ representing the team of agents. But if there is some indeterminism involved, then our fitness function fit uses the accumulated fit_run-values of several runs. The number r of runs used to compute fit is usually a parameter of the algorithm. Let $\mathcal{B}_1(\mathcal{A}g_j,s_0d_{Rest_0})$,...,$\mathcal{B}_r(\mathcal{A}g_j,s_0d_{Rest_0})$ be the r behaviors of $\mathcal{A}g_j$ in those r runs, then

$$fit(\mathcal{I},\mathcal{B}_1(\mathcal{A}g_j,s_0d_{Rest_0}),...,\mathcal{B}_r(\mathcal{A}g_j,s_0d_{Rest_0})) = \sum_{i=1}^{r} fit\_run(\mathcal{I},\mathcal{B}_i(\mathcal{A}g_j,s_0d_{Rest_0}))$$

As Genetic Operators, we use the rather standard variants of Crossover and Mutation for sets. If we look at just one agent $\mathcal{A}g_j$ and two individuals $\mathcal{I}_1$ and $\mathcal{I}_2$ that have as SAP-sets for $\mathcal{A}g_j$ $\{\text{sap}^1_{j1},...,\text{sap}^1_{jm_j}\}$ and $\{\text{sap}^2_{j1},...,\text{sap}^2_{jm_j}\}$, then Crossover picks out of $\{\text{sap}^1_{j1},...,\text{sap}^1_{jm_j}\} \cup \{\text{sap}^2_{j1},...,\text{sap}^2_{jm_j}\}$ randomly SAPs (up to a certain given limit) to create a new strategy for $\mathcal{A}g_j$. Mutation either deletes an SAP in $\{\text{sap}^1_{j1},...,\text{sap}^1_{jm_j}\}$, or adds a randomly generated SAP to $\{\text{sap}^1_{j1},...,\text{sap}^1_{jm_j}\}$, or exchanges an element in $\{\text{sap}^1_{j1},...,\text{sap}^1_{jm_j}\}$ by a new randomly generated SAP to create a new strategy for $\mathcal{A}g_j$. We can then have Crossover and Mutation on the level of individuals by either just creating a new strategy for one agent (as described above, with the agent chosen randomly) or by creating new strategies for a selection of agents. In our experiments, just modifying one agent in the case of Mutation and modifying one agent but choosing the strategies for the other agents from both parents (randomly) was already sufficient.

4 Coaching characters

SAPs together with NNR are not only a good basis for learning agents, this agent architecture also is very similar to one human method for coordinating the behavior of several persons, a method we call the "coach's drawing board". In many team sports, the coaches can draw little pictures indicating the players in the team and the opponent players and their particular (spatial) relations to each other (similar to a situation) and also indicate for each player in the team an action or action sequence. This is repeated until a complete behavior, covering the most likely alternatives in detail, is presented. And the coach assumes that the individual players will recognize the situations similar to the current one and will select the best of the drawn situations and the indicated action, similar to the SAPs with NNR architecture we described before. Sometimes, additionally some initial signals are exchanged before a play is initiated, which can be seen as producing an extended situation.

So, many humans seem to be rather familiar with the concept of prototypical situations and associated actions and therefore we think that it is relatively easy for a human being to understand agents that are based on SAPs and NNR. And even more, we think that humans can help in coming up with good SAP sets for agents. Given a good interface that allows to observe solution attempts by the learner (resp. the agents using learned strategies) and to input suggestions for SAPs easily (preferably in a graphical manner), "programming" in SAPs should be much easier than programming behavior scripts, and SAPs with NNR (using an "obvious" similarity measure sim) should be understandable by all kinds of game players (and a good development tool for game designers, too). Even more, since the previous work has shown that the evolutionary learning approach for SAPs can solve non-trivial tasks already without help from humans, it is possible to let the human developer or coach concentrate on only the key ideas for a character or a team (meaning key SAPs) for a particular task and let the evolutionary learning develop such a skeleton strategy into a working strategy for the task. If the skeleton strategy is not enough help, it can be extended after observing what the learning accomplishes and where there are still problems. The latter also allows to correct some behaviors of agents, which can be used to produce flaws or weaknesses in some agents (which might be necessary to allow a human game player to win a game).

More formally, we modify the concepts presented in Section 3 in the following manner. If we have n "coached" learning agents $\mathcal{A}g_1$,...,$\mathcal{A}g_n$, then the d_{SAP}-value of agent $\mathcal{A}g_j$ consists of a set of "coached" SAPs $\{\text{sap}^{co}_{j1},...,\text{sap}^{co}_{j,m(co)_j}\}$ and a set of learned (or evolved) SAPs $\{\text{sap}^{ev}_{j1},...,\text{sap}^{ev}_{j,m(ev)_j}\}$ that together are used in $\mathcal{A}g_j$'s decision making. If we supply the coached SAPs at the beginning and want to learn good strategies for the n agents without changing the coached SAPs, then the learning method of Section 3 can be used as described there, except that an individual $\mathcal{I}$ now has the form $\mathcal{I}$ = $(\{\text{sap}^{ev}_{11},...,\text{sap}^{ev}_{1,m(ev)_1}\},...,\{\text{sap}^{ev}_{n1},...,\text{sap}^{ev}_{n,m(ev)_n}\})$ and that

the coached SAPs are added to the evolved SAPs of an agent before the evaluation runs are performed. Note that we add the coached SAPs before the evolved ones, so that in case of having a coached SAP and an evolved one of the same (minimal) similarity to the current situation the coached one will be preferred.

If we want to change $\{sap^{co}_{j1},...,sap^{co}_{j,m(co)_j}\}$ during learning –for example, we might take a look at the evaluation runs of the best individual found after some number of generations and suggest additional coaching SAPs or remove previously used ones– then there are several ways how this can be incorporated into the learning process. [DK00] presented several possibilities, how strategies evolved under slightly different conditions can be merged and a change of the set of coached SAPs for even only one agent definitely produces slightly changed conditions. The most obvious alternative is to throw away the current population of individuals and essentially restart the whole learning process with the new sets of coached SAPs. But this means that we totally lose the experience accumulated so far by the learner.

Another alternative is to just re-evaluate the current generation of individuals (using the new sets of coached SAPs). There is a good chance that the new coached SAPs will boost the evaluation of some of the individuals (if the "coach" knows what he/she is doing), perhaps even allow an individual to fulfill the given task, which naturally is much better than starting from scratch. There are also combinations possible, where some of the evolved individuals "survive" to be re-evaluated and other members of the new population are randomly generated.

5 Experimental evaluation

We have tested our method for coaching characters combined with evolutionary learning within the OLEMAS system (see [DK00]), a testbed for evaluating concepts for learning of cooperative behavior. OLEMAS uses Pursuit Games as application. In the following, we will first take a closer look at Pursuit Games and the many variants that are covered in literature and then describe OLEMAS and especially how the general concepts and methods from Sections 2 and 3 are instantiated. Finally, we will present some case studies regarding the coaching of characters in this environment.

5.1 Pursuit Games

The term Pursuit Games is used in multi-agent systems to describe games where some predator agents hunt one or several prey agents, a scenario that is not only rather common in Nature, but was also used in games like Pacman. For multi-agent systems, the use of Pursuit Games as testbed was first suggested in [BJD85] and since then it has seen at least as many variants in research literature as there are variants of Pacman out there.

The version presented in [BJD85] had 4 dot-shaped hunters going after one dot-shaped prey on an infinite grid world without any obstacles or other agents, having all agents perform all actions with the same speed and in a turn-based manner. This description already shows off many of the features of Pursuit Games that can be modified, like numbers of hunter and prey agents, possible moves and their speed, having a more or less complex world by using obstacles and agents acting as bystanders, providing agents with shapes and so on. Even the condition to fulfill for winning can be varied: in [BJD85], the goal of the hunters was to immobilize the prey, while the ghosts in Pacman definitely were out to kill by occupying the same square as the prey. A rather extensive description of many features of Pursuit Games and possible feature values can be found in either [DF96] or [DS04].

From a learning point of view, Pursuit Games are of interest because of the many variants that really cry out for automatically developing winning strategies for the hunter agents. Naturally, many variants require cooperation between hunters to win the game. They also provide the possibility to study co-evolution of agents in various scenarios. From the computer game point of view, Pursuit Games provide a rather abstract testbed that nevertheless allows for scenarios that capture the essence of character interaction in many games.

5.2 The OLEMAS System

The OLEMAS system (**O**n-**L**ine **E**volution of **M**ulti-**A**gent **S**ystems; but the "O" can also be interpreted as **O**ff) was primarily developed to provide a testbed to evaluate evolutionary learning of cooperative behavior of agents based on SAPs and NNR. But naturally, other kinds of agents and other kinds of learning can also be integrated to work with the part of OLEMAS that simulates a wide variety of Pursuit Games. In the current version of OLEMAS, we have a variety of hard coded decision functions for agents, with quite a selection of such functions for prey agents and a few primitive functions for hunters and other agents.

At the core of OLEMAS is, as mentioned, a simulator for Pursuit Games. The particular game that is simulated is determined by providing values for the various features of Pursuit Games mentioned in the last subsection. This is usually done using a configuration file. Part of these features include a collection of agents, i.e. hunters, preys and bystander agents. Bystander agents are also used to define obstacles in the world. Each agent is defined by its own configuration file that contains the shape of an agent, its possible actions (with the number of game turns each action needs to be executed) and the decision function used by the agent. In Figure 1, we see a screenshot of the graphical interface for OLEMAS (called XOLEMAS; OLEMAS can also run without this interface which usually speeds up learning runs quite a lot). The window in the upper right corner shows the messages by the simulator. In this screenshot, we see reports on the loaded configuration files of agents.

What happens during a simulation is reported by the two windows on the left of Figure 1. The lower window displays the game. The upper window provides more detailed information on a simulation run, like the current step (or turn) number and the agents involved.

OLEMAS also has a component for learning the behav-

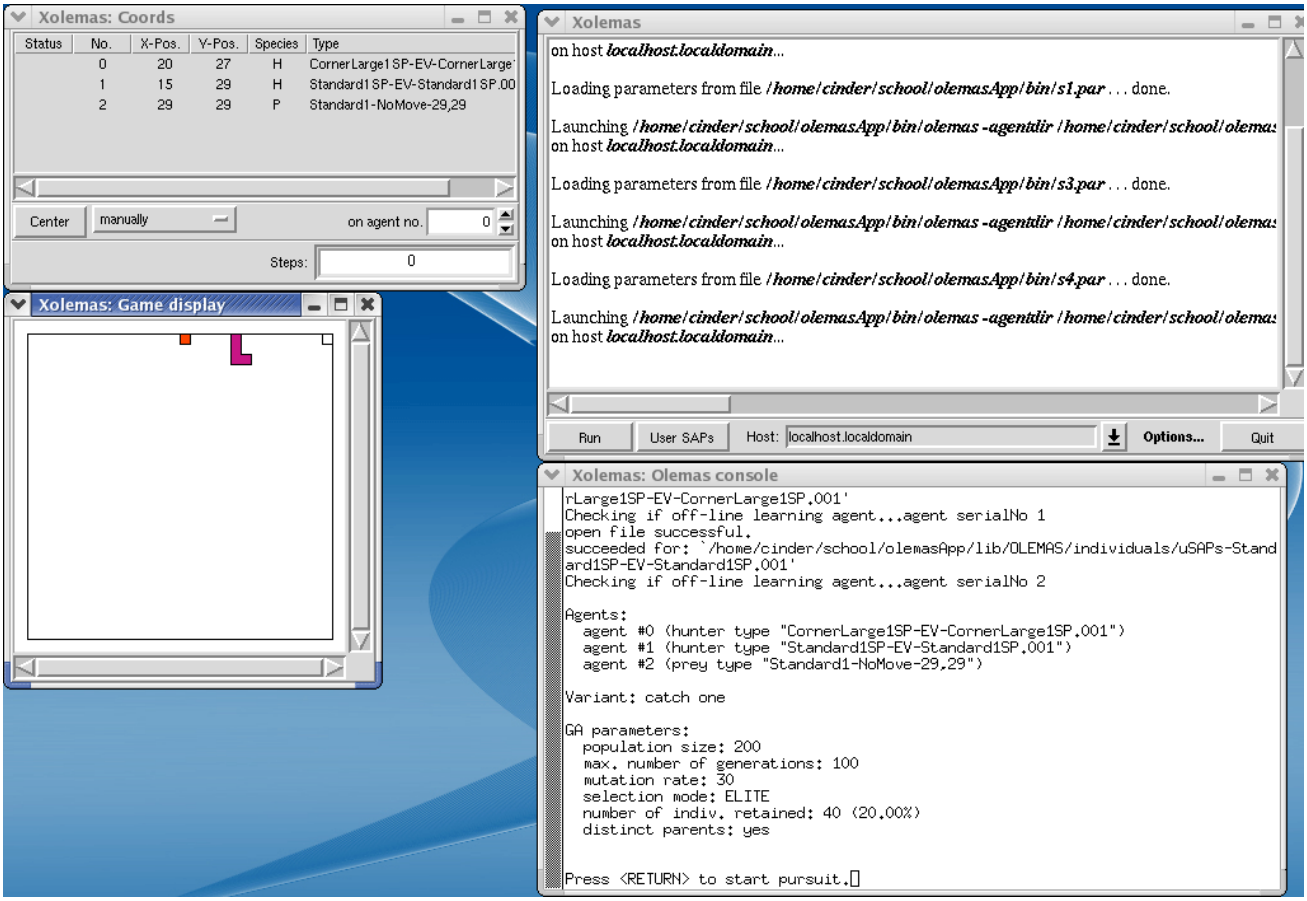


Figure 1: The interface of OLEMAS

ior of agents that implements the evolutionary learning as described in Section 3. The lower window to the right in Figure 1 displays the initial information for a learning run with details about game variant and parameter values for the genetic algorithm. A situation in OLEMAS is a vector that contains for each agent the relative coordinates, the general orientation of it in the world, the agent role (hunter, prey, or obstacle), and the agent type. Relative coordinates mean relative to the agent that is observing the situation. The last line of the text window in Figure 2 presents a situation-action pair (with the starting 0 being the number indicating the action to take).

The similarity measure sim uses only the coordinates and the orientation information and it is defined for two situations s_j and s_k with relative coordinates of agents $\mathcal{A}g_i$ being x_{ij} and y_{ij}, resp. x_{ik} and y_{ik}, and orientation of the agent being o_{ij}, resp. o_{ik} as

$$sim(s_j, s_k) = \sum_{i=1}^{n}((x_{ij} - x_{ik})^2 + (y_{ij} - y_{ik})^2 + ((o_{ij} - o_{ik})^2 \mathrm{mod} 8)).$$

Note that we do not have extended situations in OLEMAS, yet.

For the fitness of an individual, we need the function δ that measures how good a particular situation is. While we have done some work on developing measures that incorporate additional knowledge into this measure (see [DS04]), for this paper we use the rather intuitive idea of measuring how far the hunters are from the prey agents (it is important for coaching to have an intuitive measure). For the distance between two agents we use the Manhattan distance and for δ we sum up the distances between all hunter agents to all prey agents.

For incorporating the idea of coaching, we added a special interface to OLEMAS that is depicted in Figure 2. It allows to enter, resp. delete, the coached SAPs for the individual agents. In the upper left corner, we have the graphical representation of a situation. We can select agents to move them around in this presentation, using the buttons below the situation representation. On the right side, we can select a particular learning agent for which we want to edit the coached SAPs, can look at the coached SAPs, can choose the action that we want to add as SAP and we can remove a coached SAP. Note that we did not put very much work into this interface, an interface for a game developer or a game player would have to be much more sophisticated. This interface can be activated before any learning takes place or at any time the learning performed by OLEMAS is interrupted, which is determined from the main interface. If the user makes changes to the set of coached SAPs, then the current population of individuals is re-evaluated using the new coached SAPs and the learning continues.

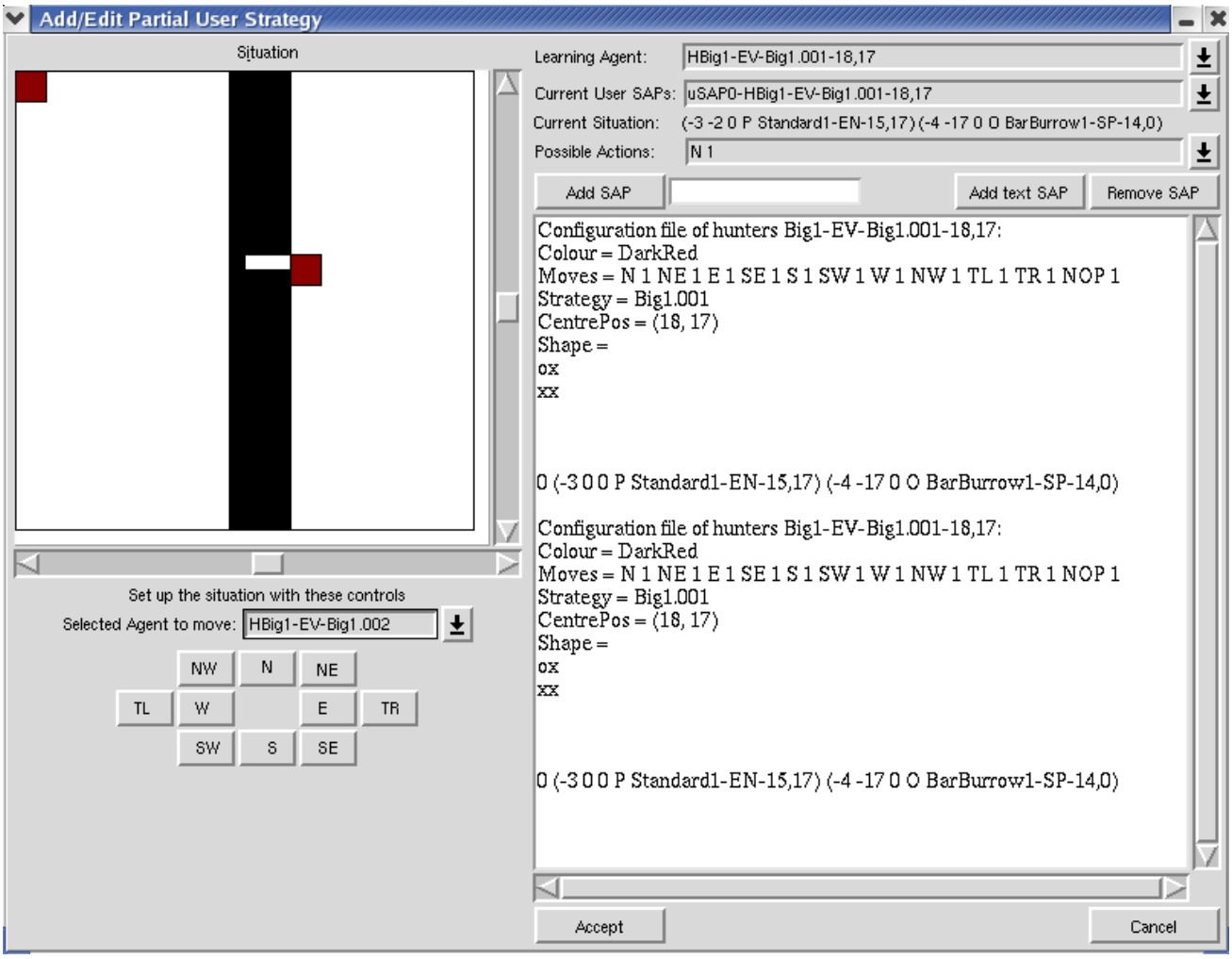


Figure 2: The coaching interface of OLEMAS

5.3 Experiments

We performed several experiments with various variants of Pursuit Games within OLEMAS. Due to lack of space, we can not present all of them and therefore selected 3 examples that show the advantages of combining coaching and learning, but also give insight into problems that the combination has so far. In order to provide some idea regarding the effort necessary for learning, we do not develop the coached SAPs interactively, but report on runs where the coached SAPs are given to OLEMAS at the beginning.

Figure 3 presents the start situations for the 3 examples. In all examples, all agents can move in all 8 directions (but no rotations) and each action takes 1 turn (step) to be completed. The prey agents (light colors) try to stay as far away as possible from the nearest hunter agent (hunters are depicted with dark colors). In all examples, the hunters goal is to "kill" the prey by moving on the same square with it.

In the first example, we have the prey agent hiding within the obstacle. Due to the used fitness measure, once the hunter reaches the obstacle, strategies where the hunter moves to the left of the prey are not favored because the hunter would be moving "away" from the prey, so the prey does not move. As Table 1 shows, the learner is not able to come up with a successful strategy (within our limit of 100 generations). Figure 4 presents the two coached SAPs that are needed to help the hunter. By sending the hunter left to the prey, the prey is flushed out of its hole and then the learner can take care of the rest. Table 1 shows that the remaining task for the learner is still difficult (one of our 10 runs was not successful within 100 generations), but the coached SAPs make a big difference. The coached SAPs alone are not able to catch the prey.

The second example presents a cooperative variant of the first example. Again, the prey is in a hole and has to be flushed out. The right hunter cannot do this alone, its colleague has to get near to the prey, while the right hunter is away, so that the prey will run away. Here learning alone can be successful (remember, we are using evolutionary learning and the random effects involved produce different learning runs each time), but the 100 generations can be too short. Using the coached SAPs from Figure 5 (the top 4 are for the right hunter, the bottom 2 are sufficient for the left one) together with learning we are always successful in catching the prey. The coached SAPs are far away from a successful strategy, they just achieve (by chance) that the learned SAPs are needed for the catch. The right hunter

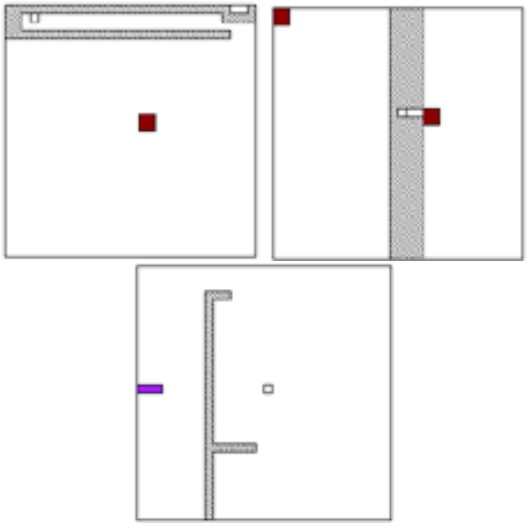


Figure 3: The start situations

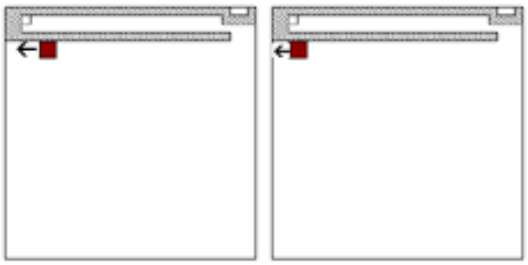


Figure 4: Coached SAPs for first example

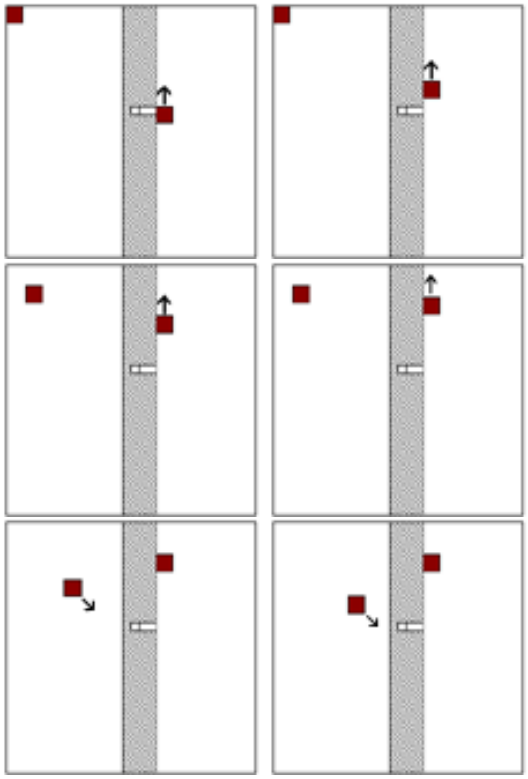


Figure 5: Coached SAPs for second example

also needs learned SAPs to get in a position to do its job.

It should be noted that the average number of steps needed for catching the prey in the successful runs of the learner alone is smaller than the average number for learning together with the coached SAPs. We search only for the first strategy that is successful. The particular coached SAPs we used here are so helpful that many sets of learned SAPs are successful together with them, so that we had a few rather curious strategies as results of the learning runs. In addition, the successful runs where the learner succeeds alone are different from those of the combined approach with the learner and coached SAPs. In these runs, the hunter on the right did not move up at all and the prey simply ran directly into it whereas in the runs with learning and coaching the prey was caught using the desired approach to the problem. This shows that not only can the coached SAPs help to solve the task but can direct the learning in the direction of a specific desired solution.

If we look at the coached SAPs for the hunter on the right in example 2, we can observe that they express the same idea: get away from the prey! Naturally, the question is, why are 4 SAPs needed. For an explanation, look at the third example and the coached SAPs that guaranteed success of the learner (see Figure 6). Navigating large obstacles has been identified as a weakness of the evolutionary learning method we use in [DK00], due to the fitness function that does not consider obstacles in the way. With coached SAPs this problem can be overcome, but several SAPs are needed to reinforce the idea of moving around the obstacle. Runs with less SAPs giving the right direction were not successful and we observed the agents going through a ping-pong effect: the learner often generated SAPs that negated the effects of the coached SAPs. But in interactive mode, this can be easily fixed.

All in all, the combination of learning and coaching is quite successful, allowing the learner to overcome it's weaknesses while still making use of it's strengths. We think that this can be a powerful tool for the development of character behavior. It would allow the human "behavior designer" to simply express the key idea for the behavior, while the learner takes care of making the idea work.

6 Related Work

There are many papers concerned with improving learning by integrating more knowledge. For our work, the following works have some relevance. In [AR02], reinforcement learning was used to learn parameter values of function skeletons, that a user defined to solve a cooperative task. We not only use a different learning method, we use a different agent architecture that does not require programming skills (at least, if combined with learning and if applied to coordination and movement problems).

Improving scripting is an important issue in commercial computer games. But most of the work focuses on making scripting easier by introducing higher level concepts (see [MC+04]). The use of learning techniques, especially evolutionary methods, has been suggested in [ML+04] and

Exp.	Learning only			Coached SAPs only		Learning With Coaching		
	Succ. Rate	Avg. Steps	Avg. Learn Time	Succ. Rate	Avg. Steps	Succ. Rate	Avg. Steps	Avg. Learn Time
1	0%	–	305.7s	0%	–	90%	55.6	73.6s
2	40%	24.8	434.5s	0%	–	100%	61.4	159.4s
3	0%	–	450.9s	0%	–	100%	66.4	54.5s

Table 1: Learning vs. pure coaching vs. learning and coaching

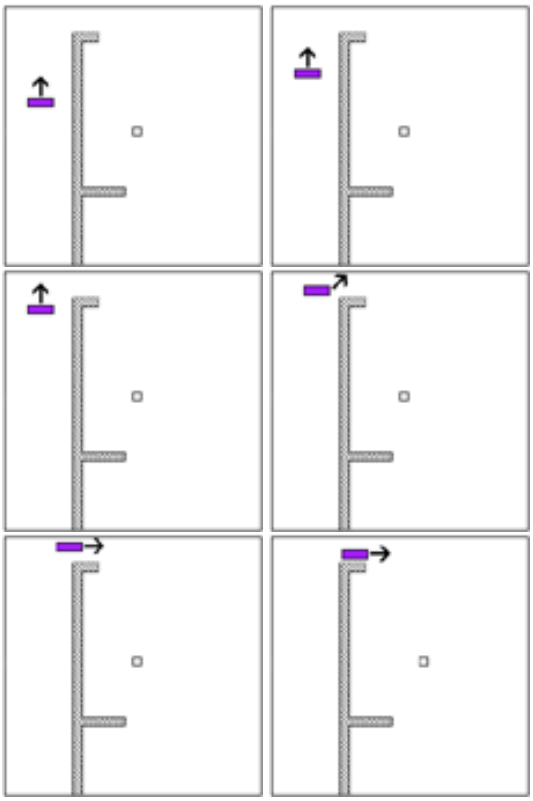


Figure 6: Coached SAPs for third example

[FHJ04] for generating better behavior of non-player characters, but the learning was on the level of parameters for existing functions/scripts, which allows less flexibility for the learner since the level of influence is on a higher level.

7 Conclusion and Future Work

We presented a method that combines learning of cooperative behavior with human coaching for agents that use prototypical SAPs and NNR as agent architecture. Providing the key behavior idea through the use of coached SAPs is very well enhanced by the evolutionary learning approach that "fills in" the other needed SAPs to produce a successful strategy. Our experiments show that the combined approach is very successful for scenarios in which the learner alone has problems. Our method is aimed at helping both game developers that have to create non-player characters and game users that want to create cooperative behavior of their characters that goes beyond the use of built-in scripts without having to write programs.

In the future, we want to address the problem where sometimes the learner tries to undo the effects of the coached SAPs. It seems that more serious changes of the evolutionary learning approach might be needed. We also want to research the effects that different fitness functions and similarity measures for situations have with regard to the ease of coaching.

Acknowledgements

This work was supported by IRIS within the IACCG project.

Bibliography

[AR02] D. Andre and S.J. Russell. State Abstraction for Programmable Reinforcement Learning Agents, Proc. AAAI-02, Edmonton, AAAI Press, 2002, pp. 119–125.

[BJD85] M. Benda, V. Jagannathan and R. Dodhiawalla. An Optimal Cooperation of Knowledge Sources, Technical Report BCS-G201e-28, Boeing AI Center, 1985.

[DE02] J. Denzinger and S. Ennis. Being the new guy in an experienced team - enhancing training on the job, Proc. AAMAS-02, Bologna, ACM Press, 2002, pp. 1246–1253.

[DE03a] J. Denzinger and S. Ennis. Improving Evolutionary Learning of Cooperative Behavior by Including Accountability of Strategy Components, Proc. MATES 2003, Erfurt, Springer LNAI 2831, 2003, pp. 205–216.

[DE03b] J. Denzinger and S. Ennis. Dealing with new guys in experienced teams - the old guys might also have to adapt, Proc. 7th IASTED ASC, Banff, ACTA Press, 2003, pp. 138-143.

[DF96] J. Denzinger and M. Fuchs. Experiments in Learning Prototypical Situations for Variants of the Pursuit Game, Proc. ICMAS'96, Kyoto, 1996, pp. 48–55.

[DK00] J. Denzinger and M. Kordt. Evolutionary On-line Learning of Cooperative Behavior with Situation-Action-Pairs, Proc. ICMAS-2000, Boston, IEEE Press, 2000, pp. 103–110.

[DS04] J. Denzinger, and A. Schur. On Customizing Evolutionary Learning of Agent Behavior, Proc. AI 2004, London, Springer, 2004, pp. 146–160.

[FHJ04] D.B. Fogel, T. Hays and D. Johnson. A Platform for Evolving Characters in Competitive Games, Proc. CEC2004, Portland, IEEE Press, 2004, pp. 1420–1426.

[MC+04] M. McNaughton, M. Cutumisu, D. Szafron, J. Schaeffer, J. Redford and D. Parker. ScriptEase: Generative Design Patterns for Computer Role-Playing Games, Proc. ASE-2004, Linz, 2004, pp. 88–99.

[ML+04] C. Miles, S. Louis, N. Cole and J. McDonnell. Learning to Play Like a Human: Case Injected Genetic Algorithms for Strategic Computer Games, Proc. CEC2004, Portland, IEEE Press, 2004, pp. 1441–1448.

Evolving Reactive NPCs for the Real-Time Simulation Game

JinHyuk Hong
Dept. of Computer Science, Yonsei University
134 Sinchon-dong, Sudaemoon-ku
Seoul 120-749, Korea
hjinh@sclab.yonsei.ac.kr

Sung-Bae Cho
Dept. of Computer Science, Yonsei University
134 Sinchon-dong, Sudaemoon-ku
Seoul 120-749, Korea
sbcho@cs.yonsei.ac.kr

Abstract- AI in computer games has been highlighted in recent, but manual works for designing the AI cost a great deal. An evolutionary algorithm has developed strategies without using features that are based on the developer. Since the real-time reactive selection of behaviors for NPCs is required for better playing, a reactive behavior system consisting neural networks is presented. Using only the raw information on games, the evolutionary algorithm optimizes the reactive behavior system based on a co-evolutionary method. For demonstration of the proposed method, we have developed a real-time simulation game called 'Build & Build'. As the results, we have obtained emergent and interesting behaviors that are adaptive to the environment, and confirmed the applicability of evolutionary approach to designing NPCs' behaviors without relying on human expertise.

1 Introduction

Game basically refers to a simulation that works entirely or partially on the basis of a game player's decisions (Angelides, 1993). Resources are allocated to the player, and he makes a decision to move for acquiring scores or a victory (Cehllapilla, 1999). Various strategies and environments make a game more interesting, and many associated strategies are developed together with the development of it. Especially computer games are very popular in these days because of the development of hardware and computer graphics (Johnson, 2001). Different with board games such as chess and checkers, computer games provide more complex and dynamic environments so as to force players to decide diverse actions. Therefore, computer games give many chances to device fantastic strategies.

Making a game more interesting requires to construct various strategies. Non-player characters (NPC) should have various patterns of behaviors within an environment. In recent, many artificial intelligence (AI) technologies are applied to design NPC's behaviors. Since computer games offer inexpensive and flexible environments, it is challengeable for many researchers to apply AI to control characters (Laird, 2001). Finite state machines and rule-based systems are the most popular techniques in designing the movement of characters, while neural networks, Bayesian network, and artificial life are recently adopted for flexible behaviors. 'The Sims' and 'Black and White' are two very successful games, in which AI has been featured. The role of AI in games might be important to produce more complex and realistic games (Laird, 2001).

It is very hard for game developers to develop many strategies. Not enough human expertise on games makes a strategy plain and tedious. Even if there is a developer with perfect expertise, it is limited in costs and time. After all, it will result in the increment of costs to develop the strategies of a game.

Evolution has been recognized as a promising approach to generate strategies of games, and applied for board games (Chellapilla, 1999). The difficulty for designing handcraft strategies makes the evolution be much outstanding, since it generates useful strategies automatically. Moreover, many works have contributed that strategies generated by the evolution are excellent enough to compete with human. Because of the applicability, however, it is rare to apply evolution to computer games such as role playing, strategy, action, etc.

In this paper, we propose a strategy generation method by evolution for simulation games. For the real-time reactivity, a reactive behavior system composed of neural networks is presented, and the system is optimized by co-evolution. A real-time computer game is developed for the demonstration of the proposed method, and several simulations are conducted to verify the strategy generation method and the proposed reactive behavior system. This paper explores the overview of AI in games for character behaviors, co-evolution, and the reactive behavior system in section 2. Section 3 describes the proposed simulation game named "Build & Build." The automatic generation of strategies by co-evolution is presented at section 4, and simulations and results will be shown in section 5. This paper concludes with a discussion on the proposed method and simulations.

2 Related Work

2.1 AI in games: Overview

The early efforts on researching computer performance at games were limited to board games such as chess, checker, etc. Developing game-playing programs is a major research area of artificial intelligence, and hence most promising artificial intelligence techniques have been applied to AI of board games (Schaeffer, 2002). With the great development of computer games in a

number of recent years, the interest of applying AI in specific computer games such as action, simulation, and role playing has been encouraged. The main research on controlling behaviors has been conducted in the field of robot and agent research, and those techniques have been slowly adopted to control NPCs in games. AI in games requires high reactivity to environments and a player's actions rather than that of robotics. A foolish behavior is apt to make a game player tedious so as to decrease the fun of the game. In this section some of the major AI techniques used in games are explored, the manner in which they been used in specific games and some of the inherent strengths and weaknesses. Conventional AI techniques in games are purely 'rules based', while in these days advanced AI techniques used in robot research are attempted for learning or adapting to environments or players' behaviors.

Rule based approach

AI of many computer games is designed with rules based techniques such as finite state machines (FSMs) or fuzzy logic. Even if they are simple rather than the other methods, they are feasible to manage behaviors well enough and familiar to game developers, since they are easy to test, modify and customize (Gough, 2000).

FSMs have a finite set Q of possible internal states, and a finite set of inputs I and outputs O. For a given internal state and inputs the automaton undergoes deterministically a transition to a new internal state and generates associated outputs. FSMs are formally defined by a transition function r.

$$r: Q \times I \rightarrow Q \times O$$

A character modeled with FSMs has a distinct set of behaviors (outputs) for a given internal state. With a specific input the associated transition works to change its internal state and to output the associated behavior. They have often been used to control enemies in first person shooters (FPS) (*e.g.*, Doom, Quake). FSMs have a weak point of its stiffness; however, the movement of a character is apt to be unrealistic.

Being against the limitation of FSMs, there is a trend towards fuzzy state machine (FuSM) that is a finite state machine given a fuzzy logic. Many computer games aim to make a character more cognitive when it makes a decision. Highly rational decisions are not always required in those games, since they sometimes decrease the reality of behaviors. Fuzzy releases the tightness so as to be more flexible in making decisions. FuSMs were applied to the FPS game 'Unreal' to make enemies appear intelligent. Fuzzy logic estimates the battle situation to decide to make a specific action. 'S.W.A.T. 2' and 'Civilisation: Call to Power' are other games using FuSMs (Johnson, 2001).

Adaptation and learning: NNs, EAs, and Artificial life

So far many game developers have let adaptation and learning in computer games unchallenged. Nevertheless, it is expected that the adaptation and learning in games will be one of the most major issues making games more interesting and realistic. Since players are apt to be tedious and easy to win because of the static strategies of NPCs, adaptation will change the strategies dynamically based on how to play. This might force the player to continually try new strategies rather than a perfect strategy so as to find interest rising.

Even though many games do not adopt adaptation, because of the difficulty of applying adaptation to games, it might offer a number of benefits to game developers. In many cases, it is very difficult to find a strategy appropriate against an opponent in a specific battle map, while learning algorithm is manageable to discover an associated strategy with minimal human knowledge. Codemasters' Colin McRae Rally 2.0 is a case that learns the movement by a neural network. While adaptation and learning can be applied to most genres of games with a degree of modeling, especially reactive games are suitable so that there are some works on applying learning to fighting games or shooting games (Johnson, 2001). Neural network, evolutionary algorithms, and artificial life are promising artificial intelligence techniques for learning in computer games.

Neural network is good at updating the AI as the player progresses through a game. The network improves continuously, so that the game is manageable to change its strategies against a game player. There are several successful games, in which neural networks are applied, such as 'Battlecruiser 3000AD', 'Dirt Track Racing', and so on. However, it has some limitations of learning and adjusting. It requires the clear specification of inputs and outputs, and sometimes this task can be very difficult for game developers. Moreover, once the neural network is badly trained, there are any other methods only to reset the network (Johnson, 2001).

Some game developers have started investigating evolutionary algorithms in computer games. Despite of cruel remarks from the game industrial world (Rabin, 2002), the evolutionary approach has great potentialities AI in games. The genetic algorithm, one of popular evolutionary algorithms, is based on the evolution theory suggested by John Holland in the beginning of 1970s. It imitates the mechanism of nature's evolution such as crossover, mutation, and the survival of the fittest, and applies to many problems of searching optimum solutions. But in the field of games, many developers have argued that the genetic algorithm required too many computations and were too slow to produce useful results. Nevertheless, evolutionary approaches might provide the adaptation of characters' behaviors and generate emergent behaviors (Jin, 2004). The difficulties in designing many strategies are feasible to be dealt by the evolutionary algorithm. There are many works on applying evolution to generate useful strategies of board games or IPD games (Fogel, 2002). Reynolds applied genetic programming to the game of tag, and obtained interesting results (Reynolds, 1994). In computer games, 'Creatures' and 'Cloak, dagger, and DNA' are the representative games using the genetic algorithm (Woodcock, 1999).

Artificial life is the study of synthetic systems that appear a natural living life, but only recently it is looked by game developers to build better game AI. It tries to

develop high-level functionalities as results of the interaction between low-level of reactions, and hence exhibit emergent behaviors. Flocking is a standard example of artificial life, while it is now applied to many fields such as movies, games, etc (Carlson, 2000). 'Creature', 'Beasts', 'Ultima online', 'Half-life', and 'Unreal' are games using artificial life to control NPCs' movements (Woodcock, 1999; Johnson, 2001).

A new AI approach

Generally recognizing that computer games are challenging applications of artificial intelligence, many advanced AI approaches have been investigated as AI in games. Bayesian network, which is used for rational reasoning with incomplete and uncertain information, was adopted for the real-time reactive selection of behaviors for an agent playing a first person shooter game (Hy, 2004). Behavior-based approaches (Arkin, 1998) are often studied to realize efficient and realistic NPCs, and applied to a popular first-person shooter game 'Half-life' (Khoo, 2002). Hybrid models of neural networks and evolutionary algorithms have been tried in the field of game research (Nelson, 2004). The hybrid model has been actively investigated for generating strategies of board games (Fogel, 2002).

2.2 Co-evolution

Co-evolution is an evolving methodology to enrich the performance of evolutionary algorithms. By simultaneously evolving two or more species with coupled fitness, it allows the maintenance of sufficient diversity so as to improve the whole performance of individuals. Ehrlick and Raven mentioned co-evolution by describing the association between species of butterflies and their host plants (Delgado, 2004). Competitive co-evolutionary models, often called host-parasite models, are a very general approach in co-evolution, in which two different species interact. Each species evolves to increase the efficiency competing to the other, and all over both of them improve their ability for surviving. Even though competitive models are limited in their narrow range of applicability, they have been widely applied to evolving the behaviors of robots or agents, the strategies of board games, and many optimization solutions (Reynolds, 1994; Fogel, 2002).

Computer games provide a good platform for the competitive co-evolution, since there are many competitions between strategies. Superior strategies for an environment have been discovered by co-evolutionary approaches. Fogel et al. generated a checker program that plays to the level of a human expert by using co-evolutionary neural networks without relying on expert knowledge (Fogel, 2002). Othello strategies based on evolved neural networks were learned at the work of Chong (Chong, 2003), while Tic Tac Toe strategies were developed by the competitive co-evolution at Angeline's work (Angeline, 1993). Shi and Krohling attempted to solve min-max problems with co-evolutionary particle swarm optimization (Shi, 2002). Reynolds presented interesting results by applying the competitive co-evolution to the game of tag (Reynolds, 1994).

2.3 Reactive behavior

Reactive model is a representative behavior selection model of intelligent robots or agents. Agents using reactive model observe the information from sensors and decide behaviors reactively. For a situation hard to be described, it performs effectively since it considers the current situation only. The inputs and outputs are directly connected with simple weights, so it is feasible to react quickly on an input. As compensation, however, it does not guarantee an optimized behavior to perform a task and shows a drop in efficiency for complex tasks or environments. Subsumption model is a standard reactive model preventing a selection of multiple behaviors.

Some genres of computer games request NPCs to reactively conduct a behavior. It keeps the tension of the game so as to increase the amusement of game players. Neural networks and behavior-based approaches are recently used for the reactive behavior of NPCs keeping the reality of behaviors.

3 The game: Build & Build

'Build & Build' developed in this research is a real-time strategic simulation game, in which two nations expand their own territory and take away the other. Each nation has soldiers who individually build towns and fight against the enemies, while a town continually produces soldiers for a given period. The motivation of this game is to observe multi-agents' behaviors and to demonstrate AIs in computer games, especially the usefulness of evolutionary approaches. Fig. 1 shows the brief view of 'Build & Build'.

Fig. 1. The brief view of 'Build & Build'

3.1 Designing the game environment

The game starts two competitive units in a restricted land with an initial fund, and ends when a nation gets to be eliminated. There are two types of the land such as normal and rock, while units are able to take some actions at the normal land but not at the rock land. A unit can build a town when the nation has enough money, while

towns produce units using some money. The nation collects taxes from towns in a proportion of the possession. This structure is originally motivated from the SCV of the 'StarCraft.'

For increasing the reality of the game, various interaction rules are designed and summarized in Table 1. Those rules are applied every unit period, and each rule has the effect on action and reaction for not causing a super strategy. Scores are calculated by several measures such as buildingTowns scores, attacking scores, producingUnits scores, and the summation of them. Since there are diverse measures to evaluate a strategy, various types of strategies might be explored by evolutionary approaches.

Table 1. Various rules and information used in this game

Basic game rule	
Gain resources per unit period	*Resource* += *Possession* × *ObtainRate*
Possession per unit period	*Possession* += *PossessionRate*, *until Possession* = 500
Build a town	Build time: 1000 frames *Cost*: 500 *Energy*: 30
Produce a unit	*Produce time*: 180 frames *Cost*: 500 *Energy*: 30
Unit period	30 frames
ObtainRate	0.01
PossessionRate	1.0
Unit size	1×1
Town size	2×2
Unit interaction rule	
Attack a unit	*Damage* = *Attacker's energy* × *DamageRate* Lost energy: Damage
Attack a town	*Damage* = *Attacker's energy* × *DamageRate* *Lost energy*: *Damage*
DamageRate	0.1

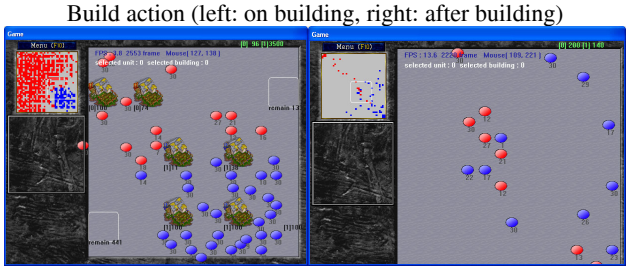
Table 2. variiables and basic actions of the NPC

Variable	
Unit ID	The distinct number of the NPC
Energy	The energy of the NPC
Location	The coordinates of the NPC (X,Y)
Delay	The frame left to make an action
Action	
Left move	Move to the left direction
Right move	Move to the right direction
Up move	Move to the up direction
Down move	Move to the down direction
Build a town	Build a town at the NPC's location
Attack a town	Attack the opponent's town
Attack a unit	Attack the opponent's NPC
Merge	Combine into one with other NPCs

3.2 Designing NPCs

Even though a type of NPC is proposed in this game, there are various actions so as to construct many strategies. The NPC called as the soldier moves based on its energy. It can move by 4 directions as well as build towns, attack units or towns, and merge with other NPCs. When the land is not a rock, the move action is possible, while the attack actions are automatically executed when an opponent locates beside the NPC. The merging is a special ability that causes the synergism of interactions, but it has a side effect on reducing the parallel activity by multi agents. It is also automatically conducted when two NPCs cross each other. Table 2 presents the types of actions and some information of a NPC, and Fig. 2 shows some representative actions of the NPC.

Build action (left: on building, right: after building)



Attack actions (left: attack buildings, right: attck NPCs)

Merge action (left: before merging, right: after merging)

Fig. 2. The basic actions of the NPC

3.3 Discussion

We have developed 'Build & Build', a real-time computer game with dynamics so as to be used as a test bed for studying various AI techniques in games, especially co-evolving reactive behavior systems of game characters in this work. Since it provides a multi-agent architecture, emergent cooperation and competition can be also investigated. Many maps can be designed using the rock land. As a future work, we will expend the game that has various types of characters and landforms. In section 4, we will present a co-evolutionary method for evolving the reactive behavior system of NPC based on 'Build & Build'

4 Evolving Reactive Behavior

Conventional approaches to construct a character are almost handcrafted, so the developer should design the behavior of the character. Even though the manual design has some benefits, it is apt to make the character static and simple. In the field of autonomous robotics, it is very important to behave flexibly in dynamic and complex

environments, since it gives the realty of the robot's behavior. Being the same as that, computer games become vary dynamic and complex, so flexible and reactive behaviors are promising to construct a character. Neural networks or behavior-based architectures are newly used in computer games for that.

4.1 Basic behavior model

The NPC has five actions excluding two attack actions and an emerge action (since they are executed automatically), and five neural networks are used to decide whether the associating action executes or not. Preventing the selection of multiple actions, the behavior model uses the subsumption model that is popular in the reactive system. The sequence of the neural networks is changeable so that an important action can have more chances to be evaluated. Fig. 3 shows the basic behavior model of the proposed method. Since the reactive behavior model considers only local information, it might be easily converged at a point. In order to actively seek a dynamic situation, the model selects a random action with a probability (in this paper, a = 0.2) in advance.

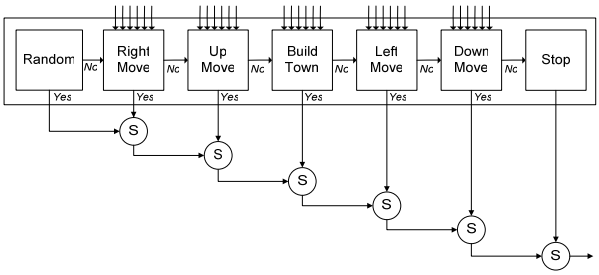


Fig. 3. The reactive behavior model of the NPC

Implementing the reality of behaviors, the neural network gets the local information of the NPC as inputs, while it results yes or no as outputs. The situation besides the NPC is encoded into a string as shown in Fig. 4, and the coding scheme is described at Table 3. The code is composed of two parts: land and NPC, and hence the code is the double-size of local blocks. In this paper, two different grid scales are used for the input of the neural network such as 5×5 and 11×11.

Table 3. Coding scheme for the input of neural networks

Land State	Value	NPC State	Value
Normal land	0	Opponent's NPC greater than the NPC	-2
Opponent's town	-1	Opponent's NPC less than the NPC	-1
Own town	1	Our forces	1
Rock land	1		

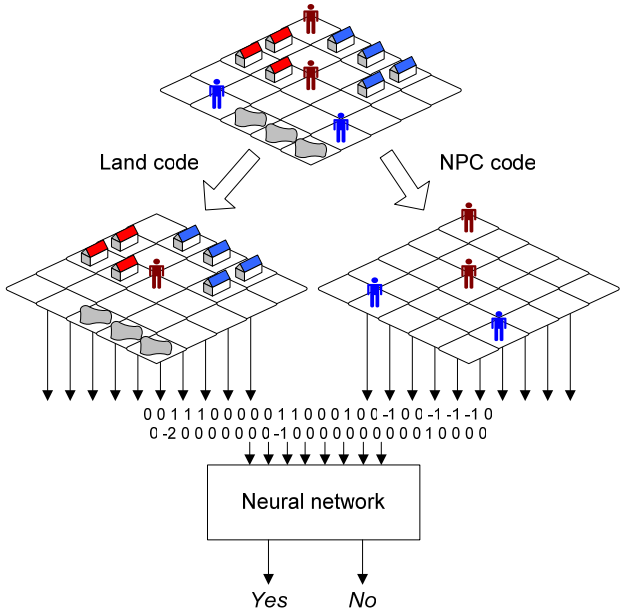


Fig. 4. The input-encoding for the neural network

4.2 Co-evolutionary behavior generation

Competition arises in nature when several species share the same limited resource, and it makes a species to adapt itself to the environment. Various strategies are necessary to survive in diverse environments, but the development of the strategies is very difficult in designing computer games. Even with different environment, we can meet a game that uses the same strategy and decreases the pleasure of playing.

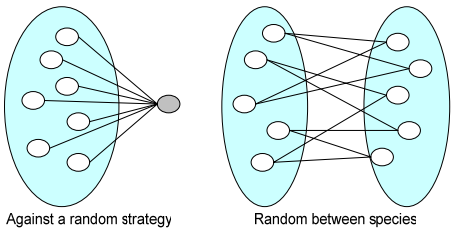


Fig. 5. Two pair-wise competition patterns adopted in this research

Co-evolutionary approaches have been investigated in recent, which effectively utilize the appearance of competition in games. In this paper, we adopt the co-evolutionary method using the genetic algorithm to generate behavior systems that are accommodated to several environments. Since $(N^2\text{-}N)/2$ competitions are necessary for a single species population of N individuals when using a simple competition pattern, two pair-wise competition patterns are adopted to effectively calculate the fitness of an individual as shown in Fig. 5. The first one has a target opponent, and especially in this paper a random strategy is used as the target. There are N competitions at each generation. The seond one divides the population into two species to compete between them by randomly selecting M opponents among the other species. This requires $N \times M/2$ competitions. When a battle

map is asymmetric, this might be expected to produce two asymmetric strategies.

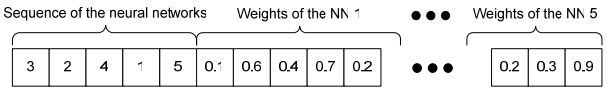


Fig. 6. The coding scheme of the chromosome

The genetic algorithm is applied to evolve the reactive behavior systems, especially two objects: the sequence of the neural networks and the weights of each neural network. The chromosome is encoded as a string that represents a reactive behavior system as shown in Fig. 6. The crossover and mutation operations and the roulette wheel selection are used to reproduce a new population. As state above, the fitness is estimated based on two pair-wise competition patterns.

The fitness of an individual is measured by the scores against randomly selected *M* opponents. Since there are several score matrixes, a weighted summation of them is used to calculate the fitness of the individual as the following formula. For efficient evolution, the total frame of a battle is included in estimating the total score. The parameters of the genetic algorithm are set based on several pre-tests as shown in Table 4.

$$TotalScore = \frac{w_1 \times BuildScore + w_2 \times AttackScore + w_3 \times ProduceScore}{\text{Total frame of the game}}$$

$$Fitness = \sum_{i=0}^{M} \frac{Own\ TotalScore\ against\ Opponent_i}{Opponent_i\text{'s}\ TotalScore}$$

Table 4. Parameters of the genetic algorithm in this paper

Parameter	Value
Selection type	Roulette wheel
Selection rate	0.7
Crossover rate	0.7
Mutation rate	0.2
Population #	200
Maximum generation #	1000

5 Experiment and Results

5.1 Simulation environments

Four different battle maps are designed as shown in Fig. 7 in order to demonstrate the proposed method in generating strategies adaptive to each environment: A plain, two symmetric lands with rocks, and an asymmetric land. The size of the maps is 20 × 20. The starting points of two teams are fixed at each corner of maps, and a battle is finished when a team is completely destroyed or it takes a time limit. At the latter case, the higher scoring team is decided to win the games. Table 5 shows the experimental environments.

Fig. 7. Battle maps used in this paper: Plain (1), two symmetric (2,3), one asymmetric (4)

Table 5. Experiment environments

Input block size	Co-evolutionary pattern	Map type
5×5, 11×11	Against a random strategy, random between species	Plain, two symmetric lands, two asymmetric lands

5.2 Experimental results

In the first experiment, which estimates the performance based on the input block size of the neural network against a random strategy, the case with 11×11 shows more diverse behaviors than that with 5×5, since it observes information on a more large area. As shown in Table 6, the behavior system with 5×5 obtains lower winning averages for complex environment, while it performs better when the environment is rather simple.

Table 6. Winning averages by input block size against a random strategy

Input block size							
5×5				11×11			
Map type				Map type			
1	2	3	4	1	2	3	4
0.86	0.64	0.17	0.35	0.82	0.65	0.36	0.41

Two input block size behavior systems have been co-evolved by random between species. On map type 3, we made them compete each other and Fig. 8 shows the result. At the early stage of evolution, the smaller shows better optimization performance than the larger, while the larger beats the smaller at the latter of evolution.

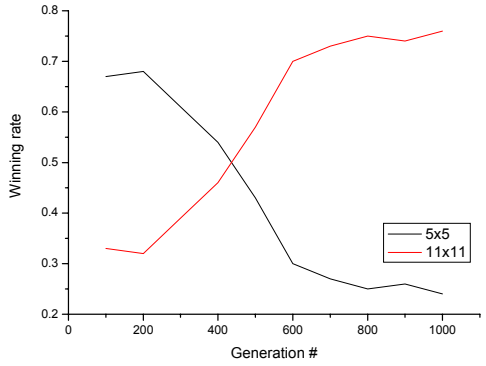


Fig. 8. Winning rate between 5×5 behavior and 11×11 behavior at each generation on map type

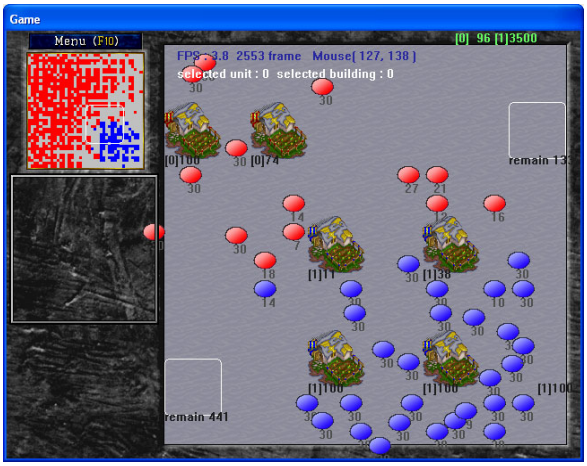


Fig. 9. A strategy obtained for the 5×5 behavior system on the plain map

We have analyzed strategies obtained for each map type against a random strategy. For the plain map, 5×5 behavior system shows a simple strategy that tries to build a town as much as possible. Building a town leads to generate many NPCs so as to slowly encroach on the battle map as showns in Fig. 9. When the starting point is the left-top position of the map, the NPCs move right or down, and build a town, but it does not show intentional movements to emerge.

The result on the map type 3 (the second symetric map) shows the limitation of the reactive behavior systems with small size of the input block. The obtained strategy works well at the early stage of a battle, but after a middle point of the battle it does not act intelligently. That kind of map requests more than two strategies for the win as descripbed in Fig. 10, but the reactive system is hard to get them at the same time. The 11×11 shows the better performance than the 5×5, since it considers more various input conditions so as to generate diverse actions.

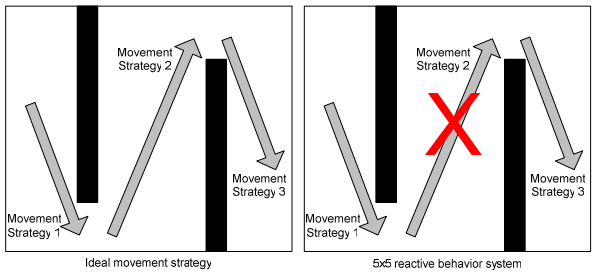


Fig. 10. Multiple movement strategies for wining battles

Asymetric maps restrict the flexibility of NPCs' movements so as to make it difficult to generate excellent winning strategies.

5.3 Discussion

We have explored evolving ractive behavior systems for various environments. The reactive system shows good performance on simple environments like the plain map, but it does not work well for complex environments. The experiments also shows that the amount of input information is important for the reactive system when the environment is not simple.

As the future work, we will extend the experiments to more various environments such as various size of input blocks, map types, and co-evolving pattens. The hybrid behavior system also will be researched, since NPCs might be more intelligent when planning fuctions are hybrided into the reactive system.

6 Conclusions

In this paper, a new evolutionary computer game environment was described and the experiment results on the reactive behavior system of the NPC were presented. The game 'Build & Build' was developed to address several current issues in the field of AIs in games, and a reactive behavior system was presented for the flexible and reactive behavior of the NPC. Co-evolutionary approaches have shown the potentialities of the automatic generation of excellent strategies corresponding to a specific environment.

This work will be extended by applying a deliberative behavior system to the game 'Build & Build', it might be beneficial to select an action to achieve a high-level goal. The hybrid model will provide the player much fun with clever strategies to win a battle. And then, multiple species and various characters and actions will be exploited for a more realistic computer game. Enhanced coding schemes for the input of neural networks will be also taken into consideration.

Acknowledgments

This paper was supported by Brain Science and Engineering Research Program sponsored by Korean Ministry of Science and Technology.

Bibliography

Angelides, M. and Paul, R. (1993) "Developing an intelligent tutoring system for a business simulation game," Simulation Practice and Theory, 1(3), 109-135.

Arkin, R. (1998) Behavior-based Robotics, MIT Press. Cambridge, MA.

Carlson, S. (2000) "Artificial life: Boids of a feather flock together," Scientific American, 283(5), 112-114.

Chellaphilla, K. and Fogel, D. (1999) "Evolution, neural networks, games, and intelligence," Proceedings of the IEEE, 87(9), 1471-1496.

Chong, S. et al. (2003) "Evolved neural networks learning othello strategies," Congress on Evolutionary Computation, 2222-2229.

Delgado, M. et al. (2004) "Coevolutionary genetic fuzzy systems: a hierarchical collaborative approach," Fuzzy Sets and Systems, 141, 89-106.

Fogel, D. and Chellapilla, K. (2002) "Verifying anaconda's expert rating by competing against Chinook: experiments in co-evolving a neural checkers player," Neurocomputing, 42(1-4), 69-86.

Gough, N. et al. (2000) "Fuzzy state machine modeling of agents and their environments for games," Proc. 1st SCS Int. Conf. GAME-ON 2000 Intelligent Games and Simulation.

Hong, J. (2004) "Evolution of emergent behaviors for shooting game characters in Robocode," Congress on Evolutionary Computation, 1, 634-638.

Hy, R. et al. (2004) "Teaching Bayesian behaviours to video game characters," Robotics and Autonomous Systems, 47, 177-185.

Johnson, D., and Wiles, J. (2001). "Computer games with intelligence," Australian Journal of Intelligent Information Processing Systems, 7, 61-68.

Khoo, A. et al. (2002) "Efficient, realistic NPC control systems using behavior-based techniques," AAAI Spring Symposium on Artificial Intelligence and Interactive Entertainment.

Laird, J. (2001) "Using a computer game to develop advanced AI," Computer, 34(7), 70-75.

Nelson, A. et al. (2004) "Evolution of neural controllers for competitive game playing with teams of mobile robots," Robotics and Autonomous Systems, 46, 135-150.

Schaeffer, J. and Herik, H. (2002) "Games, computers, and artificial intelligence," Artificial Intelligence, 134, 1-7.

Shi, Y. and Krohling, R. (2002) "Co-evolutionary particle swarm optimization to solve min-max problems," Congress on Evolutionary Computation, 1682-1687.

Rabin, S. (2002) AI game programming wisdom, Charles River Media, INC.

Reynolds, C. (1994) "Competition, coevolution and the game of tag," Proc. of Artificial Life, 4, 59-69.

Woodcock, (1999) S. "Game AI - the state of the industry," Game Developer Magazine.

A Generic Approach for Generating Interesting Interactive Pac-Man Opponents

Georgios N. Yannakakis
Centre for Intelligent Systems
and their Applications
The University of Edinburgh
AT, Crichton Street, EH8 9LE
g.yannakakis@sms.ed.ac.uk

John Hallam
Mærsk Mc-Kinney Møller Institute
for Production Technology
University of Southern Denmark
Campusvej 55, DK-5230, Odense M
john@mip.sdu.dk

Abstract- This paper follows on from our previous work focused on formulating an efficient generic measure of user's satisfaction ('interest') when playing predator/prey games. Viewing the game from the predators' (i.e. opponents') perspective, a robust on-line neuro-evolution learning mechanism has been presented capable of increasing — independently of the initial behavior and playing strategy — the well known Pac-Man game's interest as well as keeping that interest at high levels while the game is being played. This mechanism has also demonstrated high adaptability to changing *Pac-Man* playing strategies in a relatively simple playing stage. In the work presented here, we attempt to test the on-line learning mechanism over more complex stages and to explore the relation between the interest measure and the topology of the stage. Results show that the interest measure proposed is independent of the stage's complexity and topology, which demonstrates the approach's generality for this game.

1 Introduction

Over the last 25 years there have been major steps forward in computer games' graphics technology: from abstract 2D designs to complex realistic virtual worlds combined with advanced physics engines; from simple shape character representations to advanced human-like characters. Meanwhile, artificial intelligence (AI) techniques (e.g. machine learning) in computer games are nowadays still in their very early stages, since computer games continue to use simple rule-based finite and fuzzy state machines for nearly all their AI needs [1], [2]. These statements are supported by the fact that we still meet newly released games with the same 20-year old concept in brand new graphics engines.

From another viewpoint, the explosion of multi-player on-line gaming over the last years indicates the increasing human need for more intelligent opponents. This fact also reveals that interactive opponents can generate interesting games, or else increase the perceived satisfaction of the player. Moreover, machine learning techniques are able to produce characters with intelligent capabilities useful to any game's concept. Therefore, conceptually, the absolute necessity of artificial intelligence techniques and particularly machine learning and on-line interaction in game development stems from the human need for playing against intelligent opponents. These techniques will create the illusion of intelligence up to the level that is demanded by humans [3]. Unfortunately, instead of designing intelligent opponents to play against, game developers mainly concentrate and invest in the graphical presentation of the game. We believe that players' demand for more interesting games will pressure towards an 'AI revolution' in computer games in the years to come.

Predator/prey games is a very popular category of computer games and among its best representatives is the classical Pac-Man released by Namco (Japan) in 1980. Even though Pac-Man's basic concept — the player's (*PacMan's*) goal is to eat all the pellets appearing in a maze-shaped stage while avoiding being killed by four opponent characters named '*Ghosts*'— and graphics are very simple, the game still keeps players interested after so many years, and its basic ideas are still found in many newly released games. There are some examples, in the Pac-Man domain literature, of researchers attempting to teach a controller to drive *Pac-Man* in order to acquire as many pellets as possible and to avoid being eaten by *Ghosts* [4].

On the other hand, there are many researchers who use predator/prey domains in order to obtain efficient emergent teamwork of either homogeneous or heterogeneous groups of predators. For example, Luke and Spector [5], among others, have designed an environment similar to the Pac-Man game (the Serengeti world) in order to examine different breeding strategies and coordination mechanisms for the predators. Finally, there are examples of work in which both the predators' and the prey's strategies are co-evolved in continuous or grid-based environments [6], [7].

Recently, there have been attempts to mimic human behavior off-line, from samples of human playing, in a specific virtual environment. In [8], among others, human-like opponent behaviors are emerged through supervised learning techniques in *Quake*. Even though complex opponent behaviors emerge, there is no further analysis of whether these behaviors contribute to the satisfaction of the player (i.e. interest of game). In other words, researchers hypothesize — by looking at the vast number of multi-player on-line games played daily on the web — that by generating human-like opponents they enable the player to gain more satisfaction from the game. This hypothesis might be true up to a point; however, since there is no explicit notion of interest defined, there is no evidence that a specific opponent behavior generates more or less interesting games. Such a hypothesis

is the core of Iida's work on board games. He proposed a general metric of entertainment for variants of chess games depending on average game length and possible moves [9].

Similar to [5], we view Pac-Man from the *Ghosts'* perspective and we attempt to off-line emerge effective teamwork hunting behaviors based on evolutionary computation techniques, applied to homogeneous neural controlled [10] *Ghosts*. However, playing a prey/predator computer game like Pac-Man against optimal hunters cannot be interesting because of the fact that you are consistently and effectively killed. To this end, we believe that the interest of any computer game is directly related to the interest generated by the opponents' behavior rather than to the graphics or even the player's behavior. Thus, when 'interesting game' is mentioned we mainly refer to interesting opponents to play against.

In [11], we introduced an efficient generic measure of interest of predator/prey games. We also presented a robust on-line (i.e. while the game is played) neuro-evolution learning approach capable of increasing — independently of the initial behavior and *PacMan*'s playing strategy — the game's interest as well as keeping that interest at high levels while the game is being played. This mechanism demonstrated high robustness and adaptability to changing types of *PacMan* player (i.e. playing strategies) in a relatively simple playing stage. In the work presented here, we attempt to test the on-line learning mechanism over more complex stages and furthermore to explore the relation between the interest measure and the topology of the stage. Results show that the interest measure introduced in [11] is independent of the stage's design which demonstrates the approach's generality for this game.

The arcade version of Pac-Man uses a handful of very simple rules and scripted sequences of actions combined with some random decision-making to make the *Ghosts'* behavior less predictable. The game's interest decreases at the point where *Ghosts* are too fast to beat [12]. In our Pac-Man version we require *Ghosts* to keep learning and constantly adapting to the player's strategy instead of being opponents with fixed strategies. In addition, we explore learning procedures that achieve good real-time performance (i.e. low computational effort while playing).

2 The Pac-Man World

The computer game test-bed studied is a modified version of the original Pac-Man computer game released by Namco. The player's (*PacMan's*) goal is to eat all the pellets appearing in a maze-shaped stage while avoiding being killed by the four *Ghosts*. The game is over when either all pellets in the stage are eaten by *PacMan* or *Ghosts* manage to kill *PacMan*. In that case, the game restarts from the same initial positions for all five characters. Compared to commercial versions of the game a number of features (e.g. power-pills) are omitted for simplicity; these features do not qualitatively alter the nature of 'interesting' in games of low interest.

As stressed before, the Pac-Man game is investigated from the viewpoint of *Ghosts* and more specifically how *Ghosts'* emergent adaptive behaviors can contribute to the interest of the game. Pac-Man — as a computer game domain for emerging adaptive behaviors — is a two-dimensional, multi-agent, grid-motion, predator/prey game. The game field (i.e. stage) consists of corridors and walls. Both the stage's dimensions and its maze structure are predefined. For the experiments presented in this paper we use a 19×29 grid maze-stage where corridors are 1 grid-cell wide. The snapshot of the Pac-Man game illustrated in Figure 1 constitutes one of the four different stages used for our experiments. Information about the selected stages' design and the criteria for their selection are presented in Section 2.1.

The characters visualized in the Pac-Man game (as illustrated in Figure 1) are a white circle that represents *PacMan* and 4 ghost-like characters representing the *Ghosts*. Additionally, there are black squares that represent the pellets and dark grey blocks of walls.

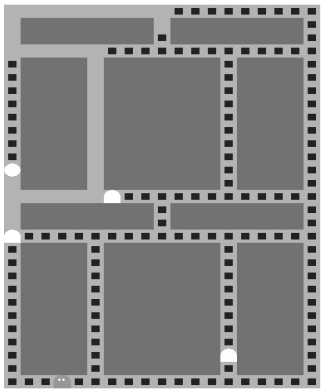


Figure 1: Snapshot of the Pac-Man game

PacMan moves at double the *Ghosts'* speed and since there are no dead ends, it is impossible for a single *Ghost* to complete the task of killing it. Since *PacMan* moves faster than a *Ghost*, the only effective way to kill *PacMan* is for a group of *Ghosts* to hunt cooperatively. It is worth mentioning that one of *Ghosts'* properties is permeability. In other words, two or more *Ghosts* can simultaneously occupy the same cell of the game grid.

The simulation procedure of the Pac-Man game is as follows. *PacMan* and *Ghosts* are placed in the game field (initial positions) so that there is a suitably large distance between them. Then, the following occur at each simulation step:

1. Both *PacMan* and *Ghosts* gather information from their environment.
2. *PacMan* and *Ghosts* take a movement decision every simulation step and every second simulation step respectively. (That is how *PacMan* achieves double the *Ghost's* speed.)
3. If the game is over (i.e. all pellets are eaten, *PacMan* is killed, or the simulation step is greater than a predetermined large number), then a new game starts from the same initial positions.

4. Statistical data such as number of pellets eaten, simulation steps to kill *PacMan* as well as the total *Ghosts'* visits to each cell of the game grid are recorded.

2.1 Stages

As previously mentioned, in this paper we attempt to test the on-line learning mechanism's ability to generate interesting games (as presented in [11]) over more complex stages and, furthermore, over stages of different topology.

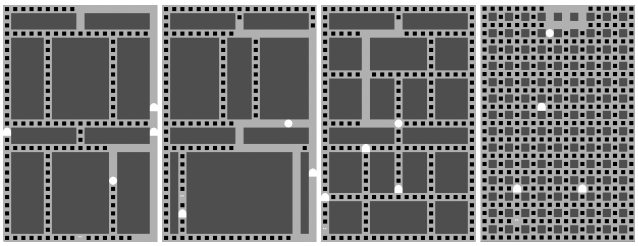


Figure 2: The 4 different stages of the game. Increasing complexity from left to right: Easy (A and B), Normal and Hard.

2.1.1 Complexity

In order to distinguish between stages of different complexity, we require an appropriate measure to quantify this feature of the stage. This measure is

$$C = 1/E\{L\} \tag{1}$$

where C is the complexity measure and $E\{L\}$ is the average corridor length of the stage.

According to (1), complexity is inversely proportional to the average corridor length of the stage. That is, the longer the average corridor length, the easier for the *Ghosts* to block *PacMan* and, therefore, the less complex the stage.

Figure 2 illustrates the four different stages used for the experiments presented here. Complexity measure values for the Easy A, Easy B, Normal and Hard stages are 0.16, 0.16, 0.22 and 0.98 respectively. Easy A stage is the test-bed used in [11]. Furthermore, given that a) blocks of walls should be included b) corridors should be 1 grid-square wide and c) dead ends should be absent, Hard stage is the most complex Pac-Man stage for the *Ghosts* to play.

2.1.2 Topology

Stages of the same complexity, measured by (1), can differ in topology (i.e. layout of blocks on the stage). Thus, in the case of Easy A and Easy B (see Figure 2), stages have the same complexity value but are topologically different.

The choice of these four stages is made so as to examine the on-line learning approach's ability to emerge interesting opponents in stages of different complexity or equally complex stages of different topology. Results presented in Section 6 show that the mechanism's efficiency is independent of both the stage complexity and stage topology and, furthermore, illustrate the approach's generality for the game.

2.2 PacMan

Both the difficulty and, to a lesser degree, the interest of the game are directly affected by the intelligence of the *PacMan* player. We chose three fixed *Ghost*-avoidance and pellet-eating strategies for the *PacMan* player, differing in complexity and effectiveness. Each strategy is based on decision making applying a cost or probability approximation to the player's 4 neighbor cells (i.e. up, down, left and right). Even though the initial positions are constant, the non-deterministic motion of *PacMan* provides lots of diversity within games.

- Cost-Based (CB) *PacMan*: The CB *PacMan* moves towards its neighbor cell of minimal cost. Cell costs are assigned as follows: $c_p = 0$, $c_e = 10$, $c_{ng} = 50$, $c_g = 100$, where c_p: cost of a cell with a pellet (pellet cell); c_e: cost of an empty cell; c_g: cost of a cell occupied by a *Ghost* (*Ghost* cell); c_{ng}: cost of a *Ghost's* 4 neighbor cells. Wall cells are not assigned any cost and are ignored by *PacMan*. In case of equal minimal neighbor cell costs (e.g. two neighbor cells with pellets), the CB *PacMan* makes a random decision with equal probabilities among these cells. In other words, the CB *PacMan* moves towards a cost minimization path that produces effective *Ghost*-avoidance and (to a lesser degree) pellet-eating behaviors but only in the local neighbor cell area.
- Rule-Based (RB) *PacMan*: The RB *PacMan* is a CB *PacMan* plus an additional rule for more effective and global pellet-eating behavior. This rule can be described as follows. If all *PacMan's* neighbor cells are empty ($c = 10$), then the probability of moving towards each one of the available directions (i.e. not towards wall cells) is inversely proportional to the distance (measured in grid-cells) to the closest pellet on that direction.
- Advanced (ADV) *PacMan*: The ADV *PacMan* checks in every non-occluded direction for *Ghosts*. If there is at least one *Ghost* in sight, then the probability of moving towards each one of the available directions is directly proportional to the distance to a *Ghost* in that direction. If there is no *Ghost* in sight, then the ADV *PacMan* behaves like a RB *PacMan*. The ADV moving strategy is expected to produce a more global *Ghost*-avoidance behavior built upon the RB *PacMan's* good pellet-eating strategy.

2.3 Neural Controlled Ghosts

A multi-layered fully connected feedforward neural controller, where the sigmoid function is employed at each neuron, manages the *Ghosts'* motion. Using their sensors, *Ghosts* inspect the environment from their own point of view and decide their next action. Each *Ghost's* perceived input consists of the relative coordinates of *PacMan* and the closest *Ghost*. We deliberately exclude from consideration

any global sensing, e.g. information about the dispersion of the *Ghosts* as a whole, because we are interested specifically in the minimal sensing scenario. The neural network's output is a four-dimensional vector with respective values from 0 to 1 that represents the *Ghost*'s four movement options (up, down, left and right respectively). Each *Ghost* moves towards the available — unobstructed by walls — direction represented by the highest output value. Available movements include the *Ghost*'s previous cell position.

2.4 Fixed Strategy Ghosts

Apart from the neural controlled *Ghosts*, three additional fixed non-evolving strategies have been tested for controlling the *Ghost's* motion. These strategies are used as baseline behaviors for comparison with any neural controller emerged behavior.

- Random (R): *Ghosts* that randomly decide their next available movement. Available movements have equal probabilities to be picked.
- Followers (F): *Ghosts* designed to follow *PacMan* constantly. Their strategy is based on moving so as to reduce the greatest of their relative distances $(\Delta_{x,P}, \Delta_{y,P})$ from *PacMan*.
- Near-Optimal (O): A *Ghost* strategy designed to produce attractive forces between *Ghosts* and *PacMan* as well as repulsive forces among the *Ghosts*. For each *Ghost* X and Y values are calculated as follows.

$$\begin{aligned} X &= \text{sign}[\Delta_{x,P}]h(\Delta_{x,P}, L_x, 0.25) \\ &- \text{sign}[\Delta_{x,C}]h(\Delta_{x,C} - 1, L_x, 10) \quad (2) \\ Y &= \text{sign}[\Delta_{y,P}]h(\Delta_{y,P}, L_y, 0.25) \\ &- \text{sign}[\Delta_{y,C}]h(\Delta_{y,C} - 1, L_y, 10) \quad (3) \end{aligned}$$

where $\text{sign}[z]=z/|z|$ and $h(z, z_m, p) = [1 - (|z|/z_m)]^p$. X and Y values represent the axis on which the near-optimal *Ghost* will move. Hence, the axis is picked from the maximum of $|X|$ and $|Y|$ whereas, the direction is decided from this value's sign. That is, if $|X| > |Y|$, then go right if $\text{sign}[X] > 0$ or go left if $\text{sign}[X] < 0$; if $|Y| > |X|$, then go up if $\text{sign}[Y] > 0$ or go down if $\text{sign}[Y] < 0$.

3 Interesting Behavior

In order to find an objective (as possible) measure of interest in the Pac-Man computer game we first need to define the criteria that make a game interesting. Then, second, we need to quantify and combine all these criteria in a mathematical formula — as introduced in [11]. The game should then be tested by human players to have this formulation of interest cross-validated against the interest the game produces in real conditions. This last part of our investigation constitutes a crucial phase of future work and it is discussed in Section 7.

To simplify this procedure we will ignore the graphics' and the sound effects' contributions to the interest of the game and we will concentrate on the opponents' behaviors. That is because, we believe, the computer-guided opponent character contributes the vast majority of features that make a computer game interesting.

By being as objective and generic as possible, we believe that the criteria that collectively define interest on the Pac-Man game are as follows.

1. *When the game is neither too hard nor too easy*. In other words, the game is interesting when *Ghosts* manage to kill *PacMan* sometimes but not always. In that sense, optimal behaviors are not interesting behaviors and *vice versa*.
2. *When there is diversity in opponents' behavior over the games*. That is, when *Ghosts* are able to find different ways of hunting and killing *PacMan* in each game so that their strategy is less predictable.
3. *When opponents' behavior is aggressive rather than static*. That is, *Ghosts* that move towards killing *PacMan* but meanwhile, move constantly all over the game field instead of simply following it. This behavior gives player the impression of an intelligent strategic *Ghosts'* plan which increases the game interest.

In order to estimate and quantify each of the aforementioned criteria of the game's interest, we let the examined group of *Ghosts* play the game N times and we record the simulation steps t_k taken to kill *PacMan* as well as the total number of *Ghosts'* visits v_{ik} at each cell i of the grid game field for each game k. Each game is played for a sufficiently large evaluation period of t_{max} simulation steps which corresponds to the minimum simulation period required by the RB *PacMan* (best pellet-eater) to clear the stage of pellets — in the experiments presented here t_{max} is 300 for the Easy stage, 320 for the Normal stage and 466 for the Hard stage.

Given these, the quantifications of the Pac-Man game's three interest criteria can be presented as follows.

1. According to the first criterion, an estimate of how interesting the behavior is, is given by T in (4).

$$T = [1 - (E\{t_k\}/\max\{t_k\})]^{p_1} \quad (4)$$

where $E\{t_k\}$ is the average number of simulation steps taken to kill *PacMan* over the N games; $\max\{t_k\}$ is the maximum t_k over the N games; p_1 is a weighting parameter (for the experiments presented here $p_1 = 0.5$);

The T estimate of interest demonstrates that the greater the difference between the average number of steps taken to kill *PacMan* and the maximum number of steps taken to kill *PacMan*, the higher the interest of the game. Given (4), both poor-killing ('*too easy*') and near-optimal ('*too hard*') behaviors get low interest estimate values (i.e. $E\{t_k\} \simeq \max\{t_k\}$).

2. The interest estimate for the second criterion is given by S in (5).

$$S = (\sigma/\sigma_{max})^{p_2} \quad (5)$$

where

$$\sigma_{max} = \frac{1}{2}\sqrt{\frac{N}{(N-1)}}(t_{max} - t_{min}) \qquad (6)$$

and σ is the standard deviation of t_k over the N games; σ_{max} is an estimate of the maximum value of σ; p_2 is a weighting parameter (for the experiments presented here $p_2 = 1$); t_{min} is the minimum number of simulation steps required for the fixed strategy Near-Optimal *Ghosts* to kill *PacMan* ($t_{min} \leq t_k$). In this paper, t_{min} is 33 simulation steps for the Easy stage; 35 for the Normal stage and 63 for the Hard stage.

The S estimate of interest demonstrates that the greater the standard deviation of the steps taken to kill *PacMan* over N games, the higher the interest of the behavior. Therefore, by using (5) we promote *Ghosts* that produce high diversity in the time taken to kill *PacMan*.

3. A good measure for quantifying the third interest criterion is through entropy of the *Ghosts'* cell visits in a game, which quantifies the completeness and uniformity with which the *Ghosts* cover the stage. Hence, for each game, the cell visits' entropy is calculated and normalized into $[0, 1]$ via (7).

$$H_n = \left[-\frac{1}{logV_n}\sum_i \frac{v_{in}}{V_n} log\left(\frac{v_{in}}{V_n}\right)\right]^{p_3} \qquad (7)$$

where V_n is the total number of visits of all visited cells (i.e. $V_n = \sum_i v_{in}$) and p_3 is a weighting parameter (for the experiments presented here $p_3 = 4$).

Given the normalized entropy values H_n for all N games, the interest estimate for the third criterion can be represented by their average value $E\{H_n\}$ over the N games. This implies that the higher the average entropy value, the more interesting the game becomes.

All three criteria are combined linearly (8)

$$I = \frac{\gamma T + \delta S + \epsilon E\{H_n\}}{\gamma + \delta + \epsilon} \qquad (8)$$

where I is the interest value of the Pac-Man game; γ, δ and ϵ are criterion weight parameters (for the experiments presented here $\gamma = 1, \delta = 2, \epsilon = 3$).

The measure of the Pac-Man game's interest introduced in (8) can be effectively applied to any predator/prey computer game because it is based on generic features of this category of games. These features include the time required to kill the prey as well as the predators' entropy throughout the game field. We therefore believe that (8) — or a similar measure of the same concepts — constitutes a generic interest approximation of predator/prey computer games (see also [13] for a successful application on a dissimilar prey/predator game). Moreover, given the two first interest criteria previously defined, the approach's generality is expanded to all computer games. Indeed, no player likes any computer game that is too hard or too easy to play and, furthermore, any player would like diversity throughout the play of any game. The third interest criterion is applicable to games where spatial diversity is important which, apart from prey/predator games, may also include action, strategy and team sports games according to the computer game genre classification of Laird and van Lent [14].

4 Off-line learning

We use an off-line evolutionary learning approach in order to produce some 'good' (i.e. in terms of performance) initial behaviors. An additional aim of this algorithm is to emerge dissimilar behaviors of high fitness — varying from blocking to aggressive (see Section 6) — offering diverse seeds for the on-line learning mechanism in its attempt to generate emergent *Ghost* behaviors that make the game interesting.

The neural networks that determine the behavior of the *Ghosts* are themselves evolved with the evolving process limited to the connection weights of the neural network. Each *Ghost* has a genome that encodes the connection weights of its neural network. A population of 80 neural networks (*Ghosts*) is initialized randomly with initial uniformly distributed random connection weights that lie within [-5, 5]. Then, at each generation:

- Every *Ghost* in the population is cloned 4 times. These 4 clones are placed in the Pac-Man game field and play ten games of t_{max} simulation steps each. The outcome of these games is to ascertain the time taken to kill *PacMan* t_k for each game.
- Each *Ghost* is evaluated via (9) for each game and its fitness value is given by $E\{f\}$ over the N_t games.

$$f = [1 - (t_k/t_{max})]^{\frac{1}{4}} \qquad (9)$$

By the use of (9) we promote *Ghost* behaviors capable of achieving high performance on killing *PacMan*.

- A pure elitism selection method is used where only the 10% fittest solutions are able to breed and, therefore, determine the members of the intermediate population. Each parent clones an equal number of offspring in order to replace the non-picked solutions from elitism.
- Mutation occurs in each gene (connection weight) of each offspring's genome with a small probability p_m (e.g. 0.02). A uniform random distribution is used again to define the mutated value of the connection weight.

The algorithm is terminated when a predetermined number of generations g is completed (e.g. $g = 1000$) and the fittest *Ghost's* connection weights are saved.

5 On-line learning (OLL)

This learning approach is based on the idea of *Ghosts* that learn while they are playing against *PacMan*. In other words, *Ghosts* that are reactive to any player's behavior and

learn from its strategy instead of being the predictable and, therefore, uninteresting characters that exist in all versions of this game today. Furthermore, this approach's additional objective is to keep the game's interest at high levels as long as it is being played. This mechanism is first introduced in [15] for an abstract prey-predator game called "Dead-End" and in [11] for the Pac-Man game. In this paper, we give a short description of OLL.

Beginning from any initial group of homogeneous off-line trained (OLT) *Ghosts*, OLL attempts to transform them into a group of heterogeneous *Ghosts* that are interesting to play against as follows. An OLT *Ghost* is cloned 4 times and its clones are placed in the Pac-Man game field to play against a selected *PacMan* type of player in a selected stage. Then, at each generation:

Step 1: Each *Ghost* is evaluated every t simulation steps via (10), while the game is played — $t = 50$ simulations steps in this paper.

$$f' = \sum_{i=1}^{t/2} \left\{ d_{P,2i} - d_{P,(2i-1)} \right\} \tag{10}$$

where $d_{P,i}$ is the distance between the *Ghost* and *Pac-Man* at the i simulation step. This fitness function promotes *Ghosts* that move towards *PacMan* within an evaluation period of t simulation steps.

Step 2: A pure elitism selection method is used where only the fittest solution is able to breed. The fittest parent clones an offspring with a probability p_c that is inversely proportional to the normalized cell visits' entropy (i.e. $p_c = 1 - H_n$) given by (7). In other words, the higher the cell visits' entropy of the *Ghosts*, the lower the probability of breeding new solutions. If there is no cloning, then go back to Step 1, else continue to Step 3.

Step 3: Mutation occurs in each gene (connection weight) of each offspring's genome with a small probability p_m (e.g. 0.02). A gaussian random distribution is used to define the mutated value of the connection weight. The mutated value is obtained from (11).

$$w_m = \mathcal{N}(w, 1 - H_n) \tag{11}$$

where w_m is the mutated connection weight value and w is the connection weight value to be mutated. The gaussian mutation, presented in (11), suggests that the higher the normalized entropy of a group of *Ghosts*, the smaller the variance of the gaussian distribution and therefore, the less disruptive the mutation process as well as the finer the precision of the GA.

Step 4: The cloned offspring is evaluated briefly via (10) in off-line mode, that is, by replacing the worst-fit member of the population and playing an off-line (i.e. no visualization of the actions) short game of t simulation steps. The fitness values of the mutated offspring and the worst-fit *Ghost* are compared and the better one is kept for the next generation. This pre-evaluation procedure for the mutated offspring attempts to minimize the probability of group behavior disruption by low-performance mutants. The fact that each mutant's behavior is not tested in a single-agent environment but within a group of heterogeneous *Ghosts* helps more towards this direction. If the worst-fit *Ghost* is replaced, then the mutated offspring takes its position in the game field as well.

The algorithm is terminated when a predetermined number of games has been played or a game of high interest (e.g. $I \geq 0.7$) is found.

We mainly use short simulation periods ($t = 50$) in order to evaluate *Ghosts* in OLL aiming to the acceleration of the on-line evolutionary process. The same period is used for the evaluation of mutated offspring; this is based on two primary objectives: 1) to apply a fair comparison between the mutated offspring and the least-fit *Ghost* (i.e. same evaluation period) and 2) to avoid undesired high computational effort in on-line mode (i.e. while playing). However, the evaluation function (10) constitutes an approximation of the examined *Ghost*'s overall performance for large simulation periods. Keeping the right balance between computational effort and performance approximation is one of the key features of this approach. In the experiments presented here, we use minimal evaluation periods capable of achieving good estimation of the *Ghosts'* performance.

6 Results

Off-line trained (OLT) emergent solutions are the OLL mechanisms' initial points in the search for more interesting games. OLT obtained behaviors are classified into the following categories:

- Blocking (B): These are OLT *Ghosts* that tend to wait for *PacMan* to enter into a specific area that is easy for them to block and kill. Their average normalized cell visit's entropy value $E\{H_n\}$ lies between 0.55 and 0.65
- Aggressive (A): These are OLT *Ghosts* that tend to follow *PacMan* all over the stage in order to kill it ($E\{H_n\} \geq 0.65$).
- Hybrid (H): These are OLT *Ghosts* that tend to behave as a Blocking-Aggressive hybrid which proves to be ineffective at killing *PacMan* ($E\{H_n\} < 0.55$).

6.1 OLL experiment

In order to portray the OLL impact on player's entertainment, the following experiment is conducted. a) Pick nine different emerged *Ghosts'* behaviors produced from off-line learning experiments — Blocking (B), Aggressive (A) and Hybrid (H) behaviors emerged by playing against each of 3 *PacMan* types — for each one of the three stages; b) starting from each OLT behavior, apply the OLL mechanism by playing against the same type of *PacMan* player and in the same stage the *Ghosts* have been trained in off-line. Initial behaviors for the Easy B stage are OLT behaviors emerged

from the Easy A stage. This experiment intends to demonstrate the effect of the topology of a stage in the interest of the game; c) calculate the interest of the game every 100 games during each OLL attempt.

Interest is calculated by letting the *Ghosts* play 100 non-evolution games in the same stage against the *PacMan* type they were playing against during OLL. In order to minimize the non-deterministic effect of the *PacMan*'s strategy on the *Ghost*'s performance and interest values as well as to draw a clear picture of these averages' distribution, we apply the following bootstrapping procedure. Using a uniform random distribution we pick 10 different 50-tuples out of the 100 above-mentioned games. These 10 samples of data, of 50 games each, are used to determine the games' average as well as confidence interval values of interest. The outcome of the OLL experiment is presented Figure 3 and Figure 4.

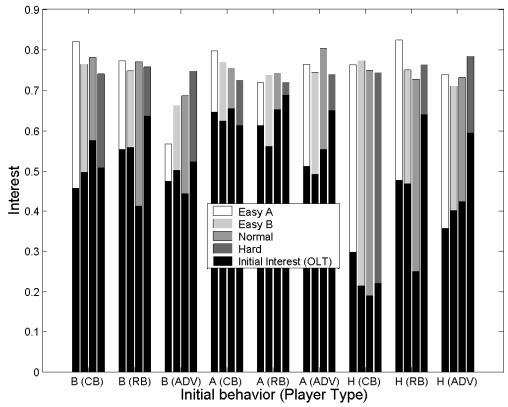


Figure 3: On-line learning effect on the interest of the game. Best interest values achieved from on-line learning on *Ghosts* trained off-line (B, A, H). Experiment Parameters: $t = 50$ simulation steps, $p_m = 0.02$, 5-hidden neurons controller.

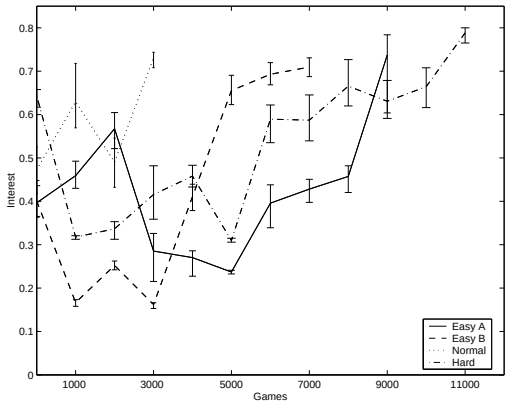


Figure 4: On-line learning effect on interest of ADV Hybrid initial behavior in all four stages. For reasons of computational effort, the OLL procedure is terminated when a game of high interest ($I \geq 0.7$) is found.

Since there are 3 types of players, 3 initial OLT behaviors and 4 stages, the total number of different OLL experiments is 36. These experiments illustrate the overall picture

			Play Against		
		Stage	CB	RB	ADV
Fixed Behaviors	R	Easy A	0.5862	0.6054	0.5201
		Easy B	0.5831	0.5607	0.4604
		Normal	0.5468	0.5865	0.5231
		Hard	0.3907	0.3906	0.3884
	F	Easy A	0.7846	0.7756	0.7759
		Easy B	0.7072	0.6958	0.6822
		Normal	0.7848	0.8016	0.7727
		Hard	0.7727	0.7548	0.7627
	O	Easy A	0.6836	0.7198	0.6783
		Easy B	0.6491	0.6725	0.6337
		Normal	0.7297	0.7490	0.6855
		Hard	0.6922	0.7113	0.4927

Table 1: Fixed strategy *Ghosts'* (R, F, O) interest values. Values are obtained by averaging 10 samples of 50 games each.

of the mechanism's effectiveness over the complexity and the topology of the stage as well as the *PacMan* type and the initial behavior (see Figure 3). Due to space considerations we present only 4 (see Figure 4) out of the 36 experiments in detail here, where the evolution of interest over the OLL games (starting from the hybrid behavior emerged by playing against the ADV *PacMan* player) on each stage is illustrated.

As seen from Figure 4, the OLL mechanism manages to find ways of increasing the interest of the game regardless the stage complexity or topology. It is clear that the OLL approach constitutes a robust mechanism that, starting from suboptimal OLT *Ghosts*, manages to emerge interesting games (i.e. interesting *Ghosts*) in all 36 cases. It is worth mentioning that in 15 out of 36 different OLL attempts the best interest value is greater than the respective Follower's value (see Table 1). Furthermore, in nearly all cases, the interest measure is kept at the same level independently of stage complexity or — in the case of Easy A and B stages — stage topology. Given the confidence intervals (± 0.05 maximum, ± 0.03 on average) of the best interest values, it is revealed that the emergent interest is not significantly different from stage to stage.

However, a number in the scale of 10^3 constitutes an unrealistic number of games for a human player to play. On that basis, it is very unlikely for a human to play so many games in order to notice the game's interest increasing. The reason for the OLL process being that slow is a matter of keeping the right balance between the process' speed and its 'smoothness' (by 'smoothness' we define the interest's magnitude of change over the games). A solution to this problem is to consider the initial long period of disruption as an off-line learning procedure and start playing as soon as the game's interest is increased.

7 Conclusion & Discussion

Predator strategies in prey/predator computer games are still nowadays based on simple rules which make the game

rather predictable and, therefore, uninteresting (by the time the player gains more experience and playing skills). A computer game becomes interesting primarily when there is an on-line interaction between the player and its opponents who demonstrate interesting behaviors.

Given some objective criteria for defining interest in predator/prey games, in [11] we introduced a generic method for measuring interest in such games. We saw that by using the proposed on-line learning mechanism, maximization of the individual simple distance measure (see (10)) coincides with maximization of the game's interest. Apart from being fairly robust, the proposed mechanism demonstrates high adaptability to changing types of player (i.e. playing strategies).

Moreover, in this paper, we showed that interesting games can be emerged independently of initial opponent behavior, playing strategy, stage complexity and stage topology. Independence from these four factors portrays the mechanism's generality and provides more evidence that such a mechanism will be able to produce interesting interactive opponents (i.e. games) against even the most complex human playing strategy.

As already mentioned, an important future step of this research is to discover whether the interest value computed by (8) for a game correlates with human judgement of interest. Preliminary results from a survey based on on-line questionnaires with a statistically significant sample of human subjects show that human players' notions of interest of the Pac-Man game correlate highly with the proposed measure of interest. More comprehensively, subjects are asked to determine the most interesting of several pairs of games, while their opponents are selected so as to produce significantly different interest values. Subsequently, a statistical analysis is carried out which is based on the correlation between observed human judgement of interest of these games and their respective interest values. Obtained results reveal that the interest metric (8) is consistent with the judgement of human players and will be part of a technical paper to be published shortly.

Bibliography

[1] Steven Woodcock. Game AI: The State of the Industry 2000-2001: It's not Just Art, It's Engineering. Game Developer magazine, August 2001.

[2] S. Cass. Mind games. *IEEE Spectrum*, pages 40–44, 2002.

[3] Alex J. Champandard. *AI Game Development*. New Riders Publishing, 2004.

[4] J. Koza. *Genetic Programming: On the programming of computers by means of natural selection*. MIT Press, 1992.

[5] Sean Luke and Lee Spector. Evolving teamwork and coordination with genetic programming. In John R. Koza, David E. Goldberg, David B. Fogel, and Rick L. Riolo, editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 150–156, Stanford University, CA, USA, 1996. MIT Press.

[6] Thomas Haynes and Sandip Sen. Evolving behavioral strategies in predators and prey. In Sandip Sen, editor, *IJCAI-95 Workshop on Adaptation and Learning in Multiagent Systems*, pages 32–37, Montreal, Quebec, Canada, 1995. Morgan Kaufmann.

[7] G. Miller and D. Cliff. Protean behavior in dynamic games: Arguments for the co-evolution of pursuit-evasion tactics. In D. Cliff, P. Husbands, J.-A. Meyer, and S. W. Wilson, editors, *From Animals to Animats 3: Proceedings of the Third International Conference on Simulation of Adaptive Behavior (SAB-94)*, pages 411–420, Cambridge, MA, 1994. MIT Press.

[8] Christian Thurau, Christian Bauckhage, and Gerhard Sagerer. Learning human-like Movement Behavior for Computer Games. In S. Schaal, A. Ijspeert, A. Billard, Sethu Vijayakumar, J. Hallam, and J.-A. Meyer, editors, *From Animals to Animats 8: Proceedings of the* 8^{th} *International Conference on Simulation of Adaptive Behavior*, pages 315–323, 2004. The MIT Press.

[9] Hiroyuki Iida, N. Takeshita, and J. Yoshimura. A metric for entertainment of boardgames: its implication for evolution of chess variants. In R. Nakatsu and J. Hoshino, editors, *IWEC2002 Proceedings*, pages 65–72. Kluwer, 2003.

[10] X. Yao. Evolving artificial neural networks. In *Proceedings of the IEEE*, volume 87, pages 1423–1447, 1999.

[11] Georgios N. Yannakakis and John Hallam. Evolving Opponents for Interesting Interactive Computer Games. In S. Schaal, A. Ijspeert, A. Billard, Sethu Vijayakumar, J. Hallam, and J.-A. Meyer, editors, *From Animals to Animats 8: Proceedings of the* 8^{th} *International Conference on Simulation of Adaptive Behavior*, pages 499–508, 2004. The MIT Press.

[12] Steve Rabin. *AI Game Programming Wisdom*. Charles River Media, Inc, 2002.

[13] Georgios N. Yannakakis and John Hallam. Interactive Opponents Generate Interesting Games. In *Proceedings of the International Conference on Computer Games: Artificial Intelligence, Design and Education*, pages 240–247, 2004.

[14] John E. Laird and Michael van Lent. Human-level AI's killer application: Interactive computer games. In *National Conference on Artificial Intelligence (AAAI)*, 2000.

[15] Georgios N. Yannakakis, John Levine, and John Hallam. An Evolutionary Approach for Interactive Computer Games. In *Proceedings of the Congress on Evolutionary Computation (CEC-04)*, pages 986–993, June 2004.

Building Reactive Characters for Dynamic Gaming Environments

Peter Blackburn and Barry O'Sullivan
Cork Constraint Computation Center
Department of Computer Science, University College Cork, Ireland
{p.blackburn|b.osullivan}@cs.ucc.ie

Abstract- Interactive computer games are widely seen as a killer application domain for Artificial Intelligence (AI) [8]. Quite apart from the significant size of the games market in terms of revenue [3], computer games provide complex, dynamic, uncertain and competitive environments that are perfect for developing, testing and deploying AI technologies. While many researchers currently focus on enhancing games with sophisticated AI, most overlook the role that AI has to play in the development of the games themselves. In this paper we present an approach to building non-player characters (NPCs) for a well-known computer game, Unreal Tournament[1]. Specifically, we use decision trees induced from human player strategies to define how an NPC in the game performs in a highly dynamic environment. The benefits of this approach are twofold. Firstly, it provides a basis for building competitive AI-based NPCs for interactive computer games. Secondly, this approach eases the development overhead of such characters. Our empirical evaluation demonstrates that the NPCs we create are very competitive against hand-crafted ones in a number of simulated gaming sessions.

1 Introduction

While the variety of genres of interactive computer games is ever increasing, it is still the case that one of the most popular is the "first person shooter". The reason for this is simple. The first person shooter is effectively the only game genre in which the human player is emersed in a virtual world and experiences this through the eyes of the game character. However, in the past these games typically provide human players with opponents that simply acted as "cannon fodder" [6], and consequently little effort was placed in developing complex behaviours for them.

However, recently the games industry has recognised that game players have grown bored of this genre due to the lack of sophistication of the opponents in many of these games. Consequently, games companies have revitalised this genre with a number of titles that attempt to overcome this issue. These games boast opponents that have sophisticated behaviours, largely due to the use of AI in their development. Some examples of such games are: Half-Life[2], No One Lives Forever 2[3] and HALO[4]. Amongst the complex behaviours exhibited by the characters in these games are the ability to evade attacks from other players, the ability to plan retreats in order to obtain support or to find a better tactical vantage point.

As a consequence of this increasing demand for more sophisticated AI in interactive computer games [20], game developers are always searching for techniques to build intelligent opponents that display human-like decision making capabilities. It is well known that sophisticated AI is a factor in ensuring a high replay rate for games [12]. Unfortunately, developing games with a high AI content often involves a significant number of very complex and challenging software development issues.

In this paper we attempt to address these issues. We present a approach that enables developers to incorporate AI in games with minimal development overhead. The method that we present demonstrates how one can build a purely reactive character that has skills based upon the strategies a human player would use if they were competing in the game. Our approach involves decomposing the traditional finite-state machine (FSM) architecture, typically used in the types of games we are interested in here, by separating the implementation of the various states that a character needs to reach and the conditions that determine the transitions from one state to another. Specifically, the approach uses a decision tree, induced from the strategies that human players would employ in various gaming scenarios as a basis for defining the conditions under which each state is reached. All the game developer needs to do is provide implementations of each state, which is typically a much easier task than implementing a traditional FSM. The decision tree defines the complex relationships between each state giving rise to behaviour by re-evaluating the state (classification) suggested by the decision tree during game play. Building game opponents in this way yields natural reactive behaviour. We can relax the unnatural behaviours that many game developers use in order to provide the illusion of intelligence such as giving game players the ability to see through walls or around corners, or to access internal data structures describing the locations of advantageous positions [11].

Therefore, the contributions we make in this paper are as follows:

1. We propose an approach to building complex, but realistic, behaviour into game characters through the use of decision trees induced from human strategies;
2. The proposed approach also reduces the software development burden required to build complex behaviours into computer games.

The remainder of this paper is organised as follows. Section 2 gives a brief overview of the literature in this area. Section 3 gives an overview of decision tree learning. Section 4 presents Unreal Tournament, the game used in this

[1] http://www.unrealtournament.com
[2] http://www.valvesoftware.com
[3] http://www.lith.com
[4] http://www.bungie.net

research, and describes the approach we used to integrate with it. Section 5 presents an evaluation of our approach, focusing specifically of the performance of the characters that are built using our approach. Section 6 presents some possible extensions and makes a number of concluding remarks.

2 Background

Incorporating AI in games is receiving significant attention from the mainstream AI community [5]. However, Nareyek [12] highlights that most academic research in the field of AI for computer games never makes it into commercial releases. This is largely due to the heavy burden that state-of-the-art research approaches place on game developers. Instead one of the most common AI techniques that one finds in games is A^*, based on Dijkstra's shortest-path algorithm [2], which is used in path-finding.

There have been many attempts to build more sophisticated AI into computer games, such as strategic planning [17], spatial reasoning [4] and case-based reasoning [16]. However, we discuss two particular pieces of work in more detail since they are quite related to the approach we present here.

Bain and Sammut [14, 1] have proposed an approach known as "behavioural cloning" that can be used to capture human sub-cognitive skills and incorporate them into a computer program. The approach uses logs of the actions performed by human subjects in performing some task as input to a learning program. The learning program produces a set of rules that can reproduce the skilled behaviour. Behavioural cloning has been used to train a computer system to fly an aircraft along a fixed route within a simulated environment using decision trees [15].

Geisler [6] demonstrated quite convincingly that Artificial Neural Networks (ANN) could be used to train an NPC to learn a set of basic commands from an experienced player: accelerate or decelerate, move forward or backward, face forward or backward and jump or do not jump. The result being the development of an NPC that could play and manoeuver successfully in a modified version of Soldier Of Fortune 2[5].

The common element employed in both of the latter techniques is the idea of training an NPC by monitoring actual game play over a long period of time. In this paper, inspired by these techniques, we show how one can use learning to control an NPC in a computer video game such as Unreal Tournament, but with significantly lower training data requirements. We demonstrate how decision trees can be used successfully for building competitive NPCs based on human-provided responses to a number of gaming scenarios. Such responses can be gathered very cheaply in an offline fashion, without game play and the need for developing sophisticated monitoring software for recording the actions of players in a live context.

[5]http://www.ravensoft.com

3 Decision Tree Learning

Decision tree learning is a standard machine learning technique for approximating discrete-valued functions [13, 10]. Decision trees make classifications by sorting the features of an instance through the tree from the root to some leaf node. Each leaf node provides the classification of the instance. At each node a test on some variable is performed, and each branch descending from that node corresponds to a possible outcome for that test. Upon reaching a leaf-node a classification is made.

A decision tree is constructed in a top-down fashion by adding a node that performs a test on a feature of the problem that best classifies the set of training examples. Based on the outcome of the test at the current node, the set of training examples is partitioned across all possible outcomes. The test for the next node on each path is determined recursively until all training examples on some branch of the tree are members of the same class. At that point we can simply place a classification node (a leaf on the tree) representing this class.

In this paper we construct decision trees using *ITI*, or *Incremental Tree Inducer* [19]. Contrary to the name, we use this software in a non-incremental or *direct metric* mode. However, we can easily use the algorithm in incremental mode in order to learn over time.

An example is presented below involving a decision tree that predicts whether we should go outside or remain inside based upon a set of example scenarios shown in Table 1, that forms the set of training examples.

Table 1: Weather experiences.

Case	Outlook	Humidity	Temp	Choice
1	Sunny	Normal	High	Go Outdoors
2	Rain	Normal	Low	Stay Indoors
3	Rain	Normal	Medium	Stay Indoors
4	Sunny	Normal	High	Go Outdoors
5	Overcast	Normal	Low	Stay Indoors
6	Overcast	High	High	Stay Indoors
7	Sunny	Normal	Low	Stay Indoors
8	Rain	Normal	Medium	Stay Indoors
9	Sunny	High	High	Go Outdoors
10	Sunny	Normal	High	Go Outdoors
11	Normal	Rain	Low	Stay Indoors
12	Sunny	Normal	Low	Stay Indoors
13	Sunny	Normal	High	Go Outdoors
14	Overcast	Normal	Low	Stay Indoors
15	Overcast	High	High	Stay Indoors

For each example in Table 1 we are presented with the values of various features that characterise the weather on a particular day, i.e. outlook, humidity and temperature. For each example we have an associated choice (classification) that specifies what we prefer to do on each day. Using the set of training examples presented in Table 1, ITI builds the decision tree presented in Figure 1.

Given the example $\langle Sunny, Normal, High \rangle$ we can determine the classification by following the appropriate

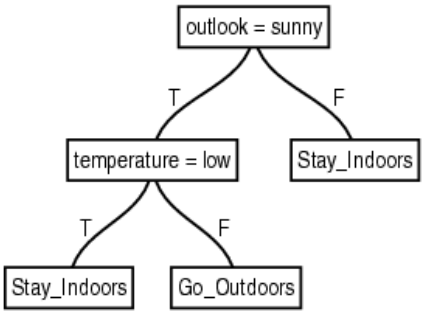


Figure 1: The decision tree induced from the weather experiences data in Table 1.

path through the decision tree. In this case, the root node checks if the outlook is sunny, since this is true in our example, we follow the branch labelled T (left hand side) to the test on the temperature feature. The tree checks if the temperature is low, but since in our example it is false, we follow the branch labelled F to reach a classification node (leaf) indicating that we should go outdoors (the classification of our example).

In the next section we present how we integrate decision tree learning with the Unreal Tournament game.

4 Integration with Unreal Tournament

Unreal Tournament is a first person shooter game developed by Epic Games, Inc [7]. Within this game there are various different game modes: death match, domination and capture the flag, to mention but a few. For the purposes of this paper we will be focusing on the death match mode of this game. A screen-shot from this game mode is presented in Figure 2.

The objective in the death match game mode is simple. Several characters compete to achieve a target number of "frags" within a specified time-frame (typically 30 minutes). A frag is a score computed as the number of opponents that are eliminated by the character less the number of times that the character eliminated himself by using a weapon inappropriately.

Figure 2: The death match mode of Unreal Tournament.

In Section 4.1 we present the underlying architecture used to interface with Unreal Tournament.

4.1 Architecture for Controlling an NPC

The programming environment for controlling our NPCs uses the client-server architecture that was first developed at the University Of Southern California's Information Science Institute called GameBot [18]. This is an architecture that is becoming very popular within the AI community in order to evaluate and test new techniques for games.

The server-side provides sensory information about the environment within the game as well as any events that are occurring. For example we may wish to access the current state of health of our NPC, the location of other opponents that are currently visible, the positions of obstacles, etc. Furthermore, the client-side provides the capability to send messages to the server, in response to the sensory data that is returned, thus specifying what action an NPC should take, for example move, turn, shoot, etc.

Within the GameBot architecture two types of message are defined that are sent from the server to the client. The first type of message are *asynchronous messages* that are dispatched immediately once an event has occurred, such as an NPC being eliminated or one character seeing another. The second type of messages are *synchronous messages* that are dispatched at predefined time intervals. These messages can contain various pieces of information ranging from current scores for a character within a game, to specific details about it such as its current health level, ammunition level, weapons equipped, full 3D location for the character, as well as yaw, pitch, roll values for the character's orientation, etc.

These messages can be parsed using any programming language since all server-client and client-server messages are sent as ASCII text strings. We chose to use a prebuilt package called JavaBot [9] which is built on top of the GameBot infrastructure and thus allows us to create and control NPCs easily using this interface without having to worry about the underlying network protocols. Essentially JavaBot enables the games developer to easily extend the JavaBot application and provide a set of classes written using the Java programming language that defines how an NPC should behave within the environment.

In Section 4.2 we discuss how we can use the JavaBot infrastructure to define how an NPC is controlled using a decision tree.

4.2 Using Decision Trees to Control NPCs

As mentioned above we use the JavaBot infrastructure to develop and evaluate our approach. This required that we extend the application in order to implement our own NPC. The extensions that we developed involved changing the standard structure of the JavaBot code. Typically JavaBot, in its distributed state, implements a simple state changing method to control an NPC. We extended this to a more sophisticated decision tree-based architecture. In this architecture we built several small simple independent states that

would be triggered by classifications obtained from the decision tree. These states provide very basic behaviours. For example, our explore state simply tells the NPC to move around the level using the in-game way-point node system, while head-on attack simply instructs the NPC to attack the enemy head-on while shooting.

In order to define the manner in which the basic states were to be configured together we built a decision tree from examples of how a human player would act in different scenarios in the game. This decision tree was then built into the game to provide our NPC with human-like behaviour and skill. We describe each phase in more detail below.

Acquiring strategies from a human player. In order to acquire the behaviours that our NPC should exhibit, we captured the approach that a skilled human player would adopt in various game scenarios. We used a form of user interview by creating a questionnaire-based system which randomly generates a series of scenarios that could occur within the game, along with various statistics which we may have, e.g. the level of health of the character, the state of its armour, etc. The human selects one of several behavioural actions in response to each scenario, e.g. the human user may choose that attack, retreat, seek health are the best actions to follow. The choice of actions that the user has available is based on the set of basic behaviours that we can build into our NPC. Typically, many scenarios are proposed to the user in order to achieve a sufficiently large training set. An example of one such questionnaire is displayed in Figure 3.

From experience we have found that even responding to over 100 scenarios is not overly burdensome on the user. Typically the time taken to answer this number of question is approximately 15 minutes, which is an extremely short time in terms of the overall amount of time involved in game development, especially given that this process is all that is required to complete the behaviour of an NPC.

Inducing the decision tree. Once the human user has responded to the set of scenarios, the scenarios themselves and the user's response forms a training set from which a decision tree can be induced. As mentioned earlier, the decision tree algorithm used for this purpose is ITI [19]. An example of one such decision tree for Unreal Tournament that models a human player's responses to various game scenarios is presented in Figure 4.

Exploiting the Decision Tree. The final phase of the integration with Unreal Tournament involved using the decision tree built above to guide the behaviour of an NPC during the game. This phase of the integration is described below.

As mentioned earlier, the game developer decides what behaviours the NPC will have at its disposal, e.g. attack, retreat, seek health, etc., and these are used in the process of interviewing the human in order to ensure that only valid behaviour states are specified in the decision tree. The process through which a behaviour is determined for the NPC during game play is as follows.

JavaBot provides real-time information from the game

Scenario 1 of 100. In this scenario what would you do? These are your statistics and your possible choices of behavior.

Health=Medium
Armour=Medium
Weapon=Long
Ammo=High
Enemy Present=True
Enemy State=Sees you
Enemy Distance=Near
Weapon Stronger=False

Your options are as follows.
Please enter the value of your choice:

0 : Head on Attack
1 : Turn & Run
2 : Retreat Shooting
3 : Run Pass them
4 : Retreat
5 : Seek Health
6 : Seek Weapons
7 : Seek Ammo
8 : Seek Armor
9 : Explore
10 : Switch Long Distance Weapon
11 : Switch Short Distance Weapon
12 : Switch to Weapon with most Ammo

Figure 3: Sample Questionnaire.

environment in the form of messages. These messages describe the state of our NPC, the presence and location of opponents that are visible, the presence and location of obstacles, etc. The majority of this data is in the form of continuous numerical values. Since decision trees do not work on numerical values directly, a preprocessor is required to discretise this data so that it can be readily processed by the decision tree. This is a standard thing to do, and involves grouping data into a set of sub-ranges that can be given a discrete label. For example, for health points a value of 0-40 might be considered "Low", 41-75 might be "Medium" and greater than 75 might be considered to be "High". This is done for each sensory variable in a meaningful way, i.e. ammunition obviously is gauged differently in that a value of 40 for ammunition might be considered to be "High". It should be noted that Bain and Sammut [1] have pointed out that this sub-ranging can lead to dramatic changes in behaviour. Therefore, these values usually need to be considered carefully for each particular game that this technique is being applied too. For our purposes the ranges mentioned above, in the context of our evaluation we found the results to be very satisfactory.

After we have preprocessed the data from JavaBot, it can be used to determine a classification from the decision tree. This is easy to do, as outlined earlier in this paper. Once

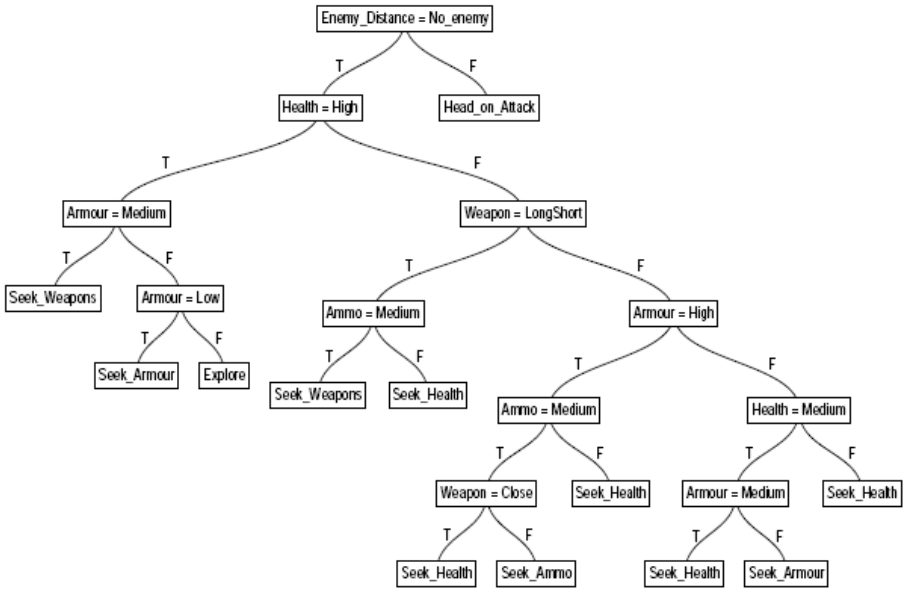


Figure 4: An example decision tree for controlling an Unreal Tournament NPC.

we have reached a leaf node in the decision tree we can conclude what action the character should take. This process is then repeated at regular intervals in order to keep the character as reactive to its environment as possible. It should be noted that this "refresh" time period can also affect the NPC's performance quite dramatically. For example, if this time period is very long (e.g. once every two seconds) then the character will obviously be very slow to respond to changes in its environment and will invariably make decisions too late. Conversely, if this time period is extremely short (ten times a second) then we are wasting resources re-evaluating the decision tree unnecessarily since it is unlikely that a different conclusion will be derived. For the purposes of our work we found that seeking a reclassification from the decision tree twice per second was adequate. However, this time again will vary depending on the game context to which this technique is being applied too. For example in simulator games like Civilization or SimCity, where the concept of time is condensed so that seconds can represent days or weeks, this polling rate may need to be increased.

5 Evaluation

To evaluate our approach to developing reactive NPCs for dynamic gaming environments we set up a series of games where we had a character built using decision trees compete against a collection of nine other standard Unreal Tournament NPCs in a death-match context. In these experiments we varied the number of questions used to build the decision trees used in each case, selecting values of 10, 50, 75 and 100, giving us different sized training sets. Furthermore, we varied the skill level of the opponent characters, comparing against both novice and average skill levels in Unreal Tournament.

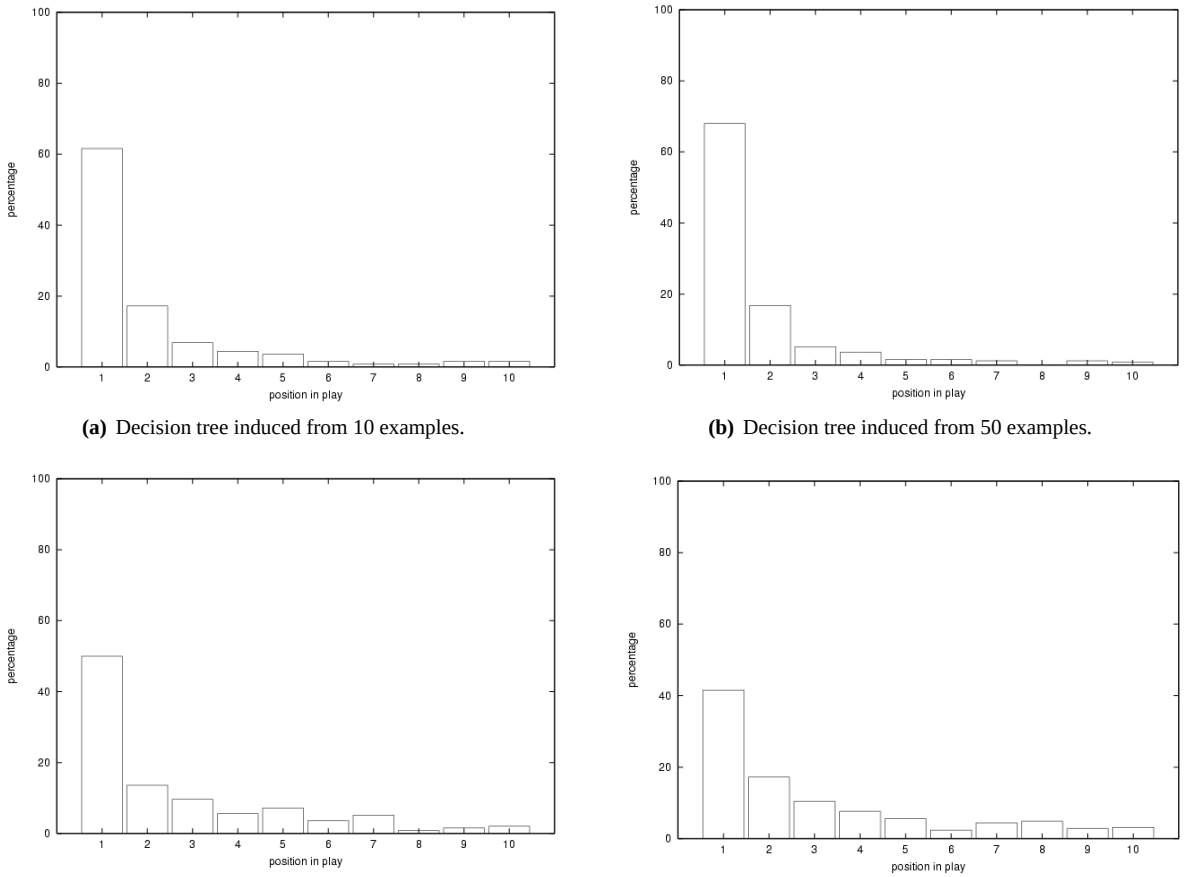
For each combination of training set size and opponent skill level, we built 5 different decision trees for each training set size and used each in 50 game sessions. Each game session had a maximum time limit of 30 minutes and a target frag count of 20. For each game we recorded the score and position of the decision tree-driven character. We present a summary of the averaged results in Table 2 in which for each game configuration we present the percentage of times, based on a weighted average over all decision trees and game sessions in each case, that the decision tree-driven NPC was ranked 5th or higher as well as the average position of the character. Figures 5 and 6 present more detailed positional information for each configuration showing the percentage of times that our NPC was placed in each position.

Table 2: Performance of the decision tree NPC.

	% times at least 5th		Average position	
#Examples	Novice	Average	Novice	Average
10	94%	93%	2.04	1.98
50	95%	93%	1.76	1.98
75	86%	88%	2.63	2.67
100	82%	82%	3.06	3.23

We can immediately see from these results that the approach we propose is extremely successful. The NPC that we produce is very competitive, consistently performing better than both the standard novice and average skilled characters in Unreal Tournament. Considering Table 2, we can see that our character is, on the average, placed in the top 3 positions in any configuration.

It is interesting to see that increasing the number of examples used to build the decision tree does not always translate into better performance. In fact, if we consider the average position of the character, performance seems to degrade once the number of examples exceeds 50. Considering the



(a) Decision tree induced from 10 examples.

(b) Decision tree induced from 50 examples.

(c) Decision tree induced from 75 examples.

(d) Decision tree induced from 100 examples.

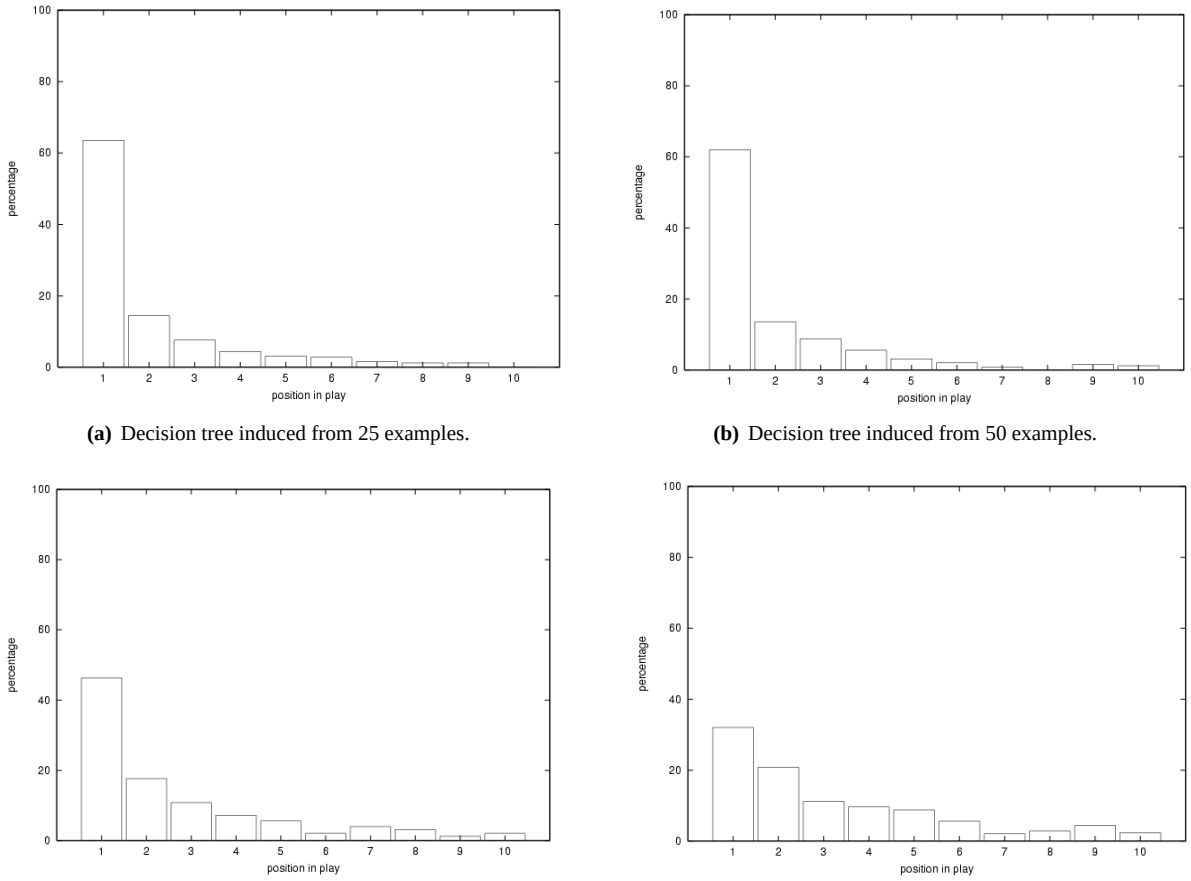
Figure 5: Performance of the NPC developed using a decision tree against novice skilled characters.

more detailed positional results presented in Figures 5 and 6, we can see that as the number of examples goes from 10 to 50, the character seems to do at least as well, if not a little better. Once we increase the number of examples to 75 we see a marked decrease in the number of first positions, however the dominance of the decision tree-driven character is never compromised.

What causes this seemingly counter-intuitive drop in performance as the number of examples increases? The answer is easy. In a death match the primary objective is to eliminate as many opponents as possible. Therefore, in order to score well, players must tradeoff the risks associated with staying in the centre of the battle with the risks of being killed or killing one's self inadvertently. As we increase the number of examples we are more likely to be presented with a larger number of examples that do not involve attack, but rather seeking health, armour, ammunitions, and opponents. While these are necessary activities for survival, in the case of seeking out health, armour and ammunitions, these are activities that remove the character from the battle zone where scores can be picked up.

From inspection of the decision trees generated by each number of examples, we noticed that the performance of the character was very sensitive to the number of non-attack classifications in the decision tree. This raises a very interesting issue. The issue here is that there is a successful strategy that one can adopt to perform well in a death match like the one we considered here: one should attack as much as possible and only leave battle when it is absolutely necessary.

We believe that it is possible to also learn good game strategy with a decision tree in an automated fashion, such as the one mentioned above. One speculative way of doing this is to inform the decision tree induction process with feedback about the performance of the resultant tree after a particular size of training set has been used. One way of providing this feedback loop is to begin by generating a large set of training data by interviewing a human user. Based on this data, we can begin to induce a decision trees based on some small number of examples, simulate the NPC in a gaming context to record its performance, and then incrementally extend the decision tree by adding new training examples to the training set. As the number of examples is increased slowly we may reach a point at which the performance of the characters begins to fall-off. We then build NPCs based on what we have found to be a good sized training set from the perspective of performance. Note that the ITI tool that we used can support incremental decision tree learning. Decision trees induced from a particular number of training examples, for a particular game, giving some average level of performance, not only propose good actions for an NPC to follow, they can also be regarded as making



(a) Decision tree induced from 25 examples.

(b) Decision tree induced from 50 examples.

(c) Decision tree induced from 75 examples.

(d) Decision tree induced from 100 examples.

Figure 6: Performance of the NPC developed using a decision tree against average skilled characters.

choices at an appropriate level of detail, depending on the number of nodes in the decision tree. This is a direction that we plan to consider as part of our future work in this area.

In terms of the effect of opponent skill level on the performance of the decision tree character, we can see from our results that there is a slight degradation in performance as we encounter more skilled opponents. This is most notably the case if we consider the percentage of times that our character is placed 1st in Figures 5 and 6. However, this degradation in performance is by no means dramatic. From Table 2 we see that our NPC is ranked at least 5th in the majority, and a similar number, of cases in both cases.

6 Conclusions

In this paper we have described how NPCs can be created in a logical and easy manner for a complex interactive computer game. This approach regards the choice of action that an NPC makes as a classification problem and, therefore, uses decision trees rather that traditional finite state machines. Decision trees have many advantages. Firstly, they are easy to build and apply, particularly since many fine tools for generating them are available to use in an off-the-shelf fashion. Secondly, they allow the game developer to shift his focus from developing sophisticated state transitions to only building a number of implementations of simple actions that a game character needs to perform. Decision tree classification can be used to link these states together in a way that we have shown results in competitive games characters. This latter point has the potential to save a game developer significant amounts of debugging, that goes with the territory of building finite state machines. Therefore, we have outlined a method of creating game character AI that is easy to incorporate as well as being extremely lightweight on resources, therefore having the potential to be applied in real world reactive gaming development contexts.

This method also provides for a form of character "personality". This essentially means that a developer can have some control over the personality that a character can have. For example, if a developer would like to develop a very aggressive character, i.e. one that always attacks regardless of its health or other statistics, this can be assisted by providing very aggressive responses to the scenarios presented on the questionnaire described earlier. Conversely, if the developer wanted to create a smarter more cautious character they might prioritize to keep health and armour statistics high. This situation could arise in a team-based game where there is a "medic" character who visits the other players on the team providing health upgrades, etc. It is important that this character survives for as long as possible to aid the others, so they can focus on achieving the objective of their mission.

It should also be mentioned that this system is easily extendable in that to add new behaviours to the system the games programmer simply has to write new behaviour states, for example a retreat tactic. This new tactic can then be included in the questionnaire used to generate the training set for the decision tree, through which it is available for inclusion in the NPCs generated using our approach. It is expected that with a larger number of states available than the ones which we created (simply for experimental purposes) much more complex behaviours can be achieved.

We would like to investigate how our approach might be extended to the task of controlling a team of characters. The key idea is that each character would behave similarly to the one outlined in this paper except all characters would be governed by a "commander" decision tree which would determine how they behave as a group. For example, the team's task might be to "capture the flag" of an opponent at a specified location. The commander tree initially informs the characters that they must capture the flag (their goal). However, the commander may also determine roles for the individual characters so that they can achieve this goal efficiently. For example, it may tell one character to flank the enemy flag, another character to run towards the enemy to create a distraction while a third character advances unnoticed to capture the flag. Inevitably some of these characters may encounter resistance, therefore they might communicate this to the commander who might then determine from its decision tree that the next course of action might be for all characters to retreat or that others should come to the aid of some other individual.

Finally, the ultimate goal is to build these techniques into a high level visualization tool and development environment that developers can use to build and modify decision tree-based NPCs. We believe that the techniques we have presented here are applicable to various other genre of games, beyond the one focused on in this paper.

7 Acknowledgements

This research is funded by Enterprise Ireland under their Basic Research Grant Scheme (Grant Number SC/02/289). We would also like to thank Ger Lawlor of Kapooki Games, Ireland, for providing us with useful feedback and comments.

Bibliography

[1] M. Bain and C. Sammut. A Framework for Behavioural Cloning. *Machine Intelligence*, 5, 1995.

[2] E.W. Dijkstra. A note on two problems in connection with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[3] ESA. Entertainment software association. http://www.theesa.com.

[4] K. Forbus, J. Mahoney, and K. Dill. How qualitative reasoning can improve strategy game AIs. *IEEE Intelligent Systems*, 17(4):25–30, 2002.

[5] D. Fu, S. Henke, and J. Orkin, editors. *Challenges in Game Artificial Intelligence*. AAAI Press, July 2004. Papers from the AAAI Workshop.

[6] B. Geisler. Integrated machine learning for behaviour modeling in video games. In Dan Fu, Stottler Henke, and Jeff Orkin, editors, *Proceedings of the AAAI-2004 Workshop on Challenges in Game Artificial Intelligence*, pages 54–62. AAAI Press, 2004.

[7] Epic Games Inc. Unreal Tournament, 1999. http://www.unrealtournament.com.

[8] J. Laird and M. van Lent. Interactive Computer Games: Human-Level AI's Killer Application. *AI Magazine*, 22(2):15–25, 2001.

[9] A. Marshall, R. Rozich, J. Sims, and J. Vaglia. Unreal Tournament Java Bot. http://sourceforge.net/projects/utBot/.

[10] T. Mitchell. Decision tree learning. In *Machine Learning*, chapter 3, pages 52–80. McGraw Hill, 1997.

[11] C. Moyer. How Intelligent is a Game Bot, Anyway? http://www.tcnj.edu/ games/papers/Moyer.html (online paper).

[12] A. Nareyek. AI in computer games. *ACM Queue*, 1(10):58–65, 2004.

[13] J.R. Quinlan. Induction of decision trees. *Machine Learning*, 1(1):81–106, 1986.

[14] C. Sammut. Acquiring expert knowledge by learning from recorded behaviours. In *Japanese Knowledge Aquisition Workshop*, pages 129–144, 1992.

[15] C. Sammut. Automatic construction of reactive control systems using symbolic machine learning. *The Knowledge Engineering Review*, 11(1):27–42, 1996.

[16] D. Sinclair. Using example-based reasoning for selective move generation in two player adversarial games. In *Proceedings of EWCBR*, LNAI 1488, pages 126–135, 1998.

[17] S.J.J. Smith, D.S. Nau, and T. Throop. Success in spades: Using AI planning techniques to win the world championship of computer bridge. In *Proceedings of AAAI/IAAI*, pages 1079–1086, 1998.

[18] ISI University Of Southern California. Gamebot API. http://www.planetunreal.com/gamebots/index.html.

[19] P.E. Utgoff, N.C. Berkman, and J.A. Clouse. Decision tree induction based on efficient tree restructuring. *Machine Learning*, 29:5–44, 1997.

[20] M. van Lent and J. Laird. Developing an artificial intelligence engine. In *Game Developers Conference*, pages 577–588, 1999.

Adaptive Strategies of MTD-f for Actual Games

Kazutomo SHIBAHARA Nobuo INUI Yoshiyuki KOTANI
Tokyo University of Agriculture and Technology
k-shiba@fairy.ei.tuat.ac.jp nobu@cc.tuat.ac.jp kotani@cc.tuat.ac.jp

Abstract- MTD algorithm developed by Plaat is a variation of SSS* which uses the depth-first strategy to resolve the storage problem coming from the best-first strategy. Since MTD algorithm is based on the zero window search algorithm, the initial range of the searching windows plays an important role in the performance. In this paper, we show some basic experimental results of MTD algorithm. From considerations of the results, the performance of MTD algorithm is expected to be improved by the reduction of the number of times that zero window procedure is called. Then we propose two variations of MTD algorithm. Our purpose is the reduction of searching nodes by the setting of initial window ranges and the way of narrowing the range in the search. Experimental results show 6 percents improvement of searching nodes against MTD-f algorithm.

1. Introduction

In the current state of art, the fast game-tree searching algorithm is essential for the development of strong game-playing systems. Though, efficient algorithms like PN searching algorithm were developed for ending games, there are not efficient algorithms over alpha-beta method essentially for middle games. The best-first searching algorithms like SSS* are theoretically efficient, though the storage problem remains. The storage problem comes from the selection of an optimal successor among all nodes on the current searching tree. MTD algorithm [1][4] was developed for overcoming this problem. MTD algorithm uses zero window searching method [10] repeatedly. Concretely, MTD algorithm goes to find the optimal solution from a range of evaluation values specified before the execution of the algorithm. Zero window search is an alpha-beta searching algorithm which windows, defined by the alpha value and the beta value, is narrow enough to get the evaluation value with the reasonably short time. MTD algorithm was experimented for several games like checker, othello and chess, and showed better results than the traditional algorithms such as Nega-Scout method.

In this paper, we apply the MTD algorithm to game trees which are randomly generated or extracted from SHOGI games. The section 2 describes several variations of MTD algorithm. In the section 3, we argue the performance of MTD algorithms for random-generated game-trees. We propose the new variations of MTD algorithm, MTD-f-step and MTD-f-alpha-beta, which are developed for the reduction of total searching nodes, in the section 4 and show some experimental results in the section 5. We make a discussion of our method in the section 6 and make a conclusion in the section 7.

2. Various Variations of MTD

MTD algorithm extends “zero window alpha-beta algorithm” named by Pearl to finding optimal solutions. Fundamentally, zero window algorithm judges whether the minimax value of a subtree is in the window specified by the alpha and the beta value or not. The zero window searching algorithm is usually called NULL window searching algorithm. The purpose of game tree searching is usually to find the minimax value of a game-tree to evaluate the current position. MTD algorithm goes to find the minimax value of a whole game tree by narrowing an existing range of the true mini-max value. While searching by MTD algorithm, there are many subtrees which are a part of a whole game-tree and are possible to be duplicated each other. To avoid generating the same subtrees, a transposition table is used to judge whether a node is checked or not previously.

There are two parameters in MTD, which affect the performance:

- A searching range which is set initially
  We expect that the true minimax value exists in this range.
- A way of re-setting a searching range after an execution of zero window searching

The paper [1] has shown the several possibilities as

described below:

(a) MTD+infinity

In MTD+infinity, an initial searching range is [-infinity, +infinity]. The upper bound of the range is used to search zero window algorithm. After the execution of the zero windows algorithm, the upper bound of the range is rewritten by returned value of zero window algorithm until the true minimax value is found. MTD+infinity is equivalent to SSS* [13].

(b) MTD-infinity

In MTD-infinity, an initial searching region is [-infinity, +infinity] as same as MTD+infinity. The different point of MTD-infinity against MTD+infinity is that the lower bound of the range is rewritten after the execution of zero window algorithm. By this, the behavior of MTD-infinity is quite different from MTD+infinity. Simply speaking, MTD-infinity expands all MIN nodes (nodes for the opponent player) and a MAX node (self nodes) included in a game tree against MTD+infinity expands all MAX nodes and a MIN node. So when the search depth is odd, there are few nodes expanded by MTD-infinity.

(c) MTD-f

In MTD-f, an initial searching range is [-infinity, f] or [f, +infinity], where "f" approximates the true minimax value. MTD-f algorithm changes the region by the result of zero window algorithm. When the true minimax value is over "f", the upper bound is rewritten as same as MTD+infinity and when the true minimax value is under "f", the lower bound is rewritten as same as MTD-infinity. The range is expected to be set near the true minimax value in both cases.

(d) Other MTDs

Other models like MTD-bi, MTD-step and MTD-best have been proposed in [1]. A re-set value is an average value of the upper bound and the lower bound in MTD-bi. The true minimax value is searched in a binary search method in this case. In MTD-step, an initial range is [-infinity, +infinity] and the upper bound is rewritten by the value calculated by the subtraction of the fix value from the upper bound. In this case, the range is expected to be changed more rapid than other methods. MTD-best only explores the lower bound, not the true minimax value. MTD-best is possible to find the lower bound faster than the other method, since the subtree generated while the execution become small. But there is a risk to fail to find the true minimax value.

MTD-f is shown as the best method among the variation of MTD algorithm in [1]. So we try to improve MTD-f in this paper.

3. Preliminary Experiments of MTD-f for a Random Game Tree

Before considering improvement of MTD-f, we investigated the performance of it.

3.1. Experiment Set-up

In our preliminary experiment, we compare MTD-f algorithm with NegaScout algorithm [12] and alpha-beta algorithm for 500 randomly generated trees of the fixed depth. We made the tree that is similar to actual game tree. Concretely, random numbers are first allocated to all nodes. Then, the evaluation value of the leaf node allocates the total of random numbers of ancestor's nodes. As a result, the tendency to the evaluation value of the leaves becomes congruent with that of actual game trees. The depth (D) is varied from 3 to 6 and neither the search extension nor the iterative deepening is introduced to the experimental systems. The initial value "f" of MTD-f is determined by the result of the shallow alpha-beta search (the depth of searching is D-2).

3.2. Experimental Results

The fig. 1 shows the relation of the search depth and the total number of search nodes.

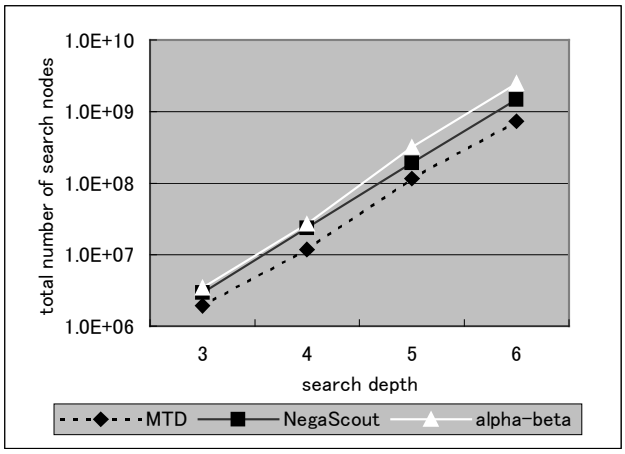


Fig.1 The relation between tree depth and the number of search nodes

MTD-f showed higher performance than Nega-Scout

and alpha-beta algorithm. Moreover, in the execution time in the depth 6, MTD was 1.8 times as fast as Nega-Scout. Since the cost of evaluating positions cannot be ignored in actual games, MTD-f method would show the performance advantage clearly.

The number of times that zero window algorithm is called is 4 in average. Since, calling zero window algorithm is required twice at first, our experimental results means that MTD-f is very efficient for the random tree searching.

The minimum tree is defined as a game tree such that all the first successor is the best among brother successors. In this case, alpha-beta algorithm works well. The table 1 shows the number of nodes searched for the minimum trees in this experiment.

Table 1. The Number of Searched Nodes the Minimum Tree

depth	MTD's nodes	NS's nodes	MTD's time(s)	NS's time(s)
5	135201	135196	0.11	0.093
6	385189	385182	0.313	0.281
7	6761030	6759788	2.532	2.343
8	19280065	19253820	12.875	12.641
9	338319068	337771768	123.813	117.922

We cannot find the difference between MTD-f and Nega-Scout from the table 1. This is because MTD-f algorithm only searches the minimum trees to find the solutions. The number of nodes searched for the minimum trees are increased for MTD-f algorithm, since the duplicate nodes must be searched at the first two searches to judge whether the true minimax value is over "f" or under "f". MTD-f is the best method in another experiment that the best successor is placed the last among all successors. From these experimental results, we conclude that the number of nodes searched by MTD-f is not larger than the other method. So to improve the performance of MTD-f is only by the reduction of the times of calling zero window algorithm.

4. Improvement MTD-f

In actual game tree search, especially SHOGI, the number of times of calling zero window algorithm is expected to be increased, because of the difference among evaluation values for each depth of a game tree. This means the shallow search used for determining an initial region is not so correct. If an approximate value "f" is precious, MTD-f can find true minimax value as same as in random-generated trees. Though game trees can be designed as the minimum trees, when advanced evaluation functions are available [3], it is not realistic in the current state of art. So the faster searching algorithms for searching nodes and depths are necessary.

MTD-step mentioned in the section 2 is proposed for finding the true minimax value rapidly. However, the performance is almost same as MTD+infinity since the initial range is set as [-infinity, +infinity]. To improve the performance of MTD-f, the combination of MTD-f and MTD-step is a natural way for the improvement. In this section, we propose a new type of MTD algorithm, which we name "MTD-f-step". In addition, we try the combination of MTD-f and the window search algorithm, called MTD-f-alpha-beta.

4.1 MTD-f-step

MTD-step uses the initial range [-infinity, +infinity]. Against this, our proposed MTD-f-step uses the initial range determined by the first solution of MTD-f. We expect the behavior of MTD-f-step is similar to MTD-f by this initial range. To use MTD-step, the step size should be determined in the next.

In the MTD-f-step, the step size is varied by the history of searching. This method is based on the binary search algorithm. Initially, we set the step size as 100 and fix it while the true minimax value is in the range. When the true minimax value is in the outside of the range, the previous narrowing of a range is cancelled and the step size is made half (50 in the first time). MTD-step algorithm works in MTD-f-step until the step size becomes 0. When the step size is 0, MTD-f-step algorithm changes the searching strategy to MTD-infinity or MTD+infinity. The pseudo-code of MTD-f-step is shown in Fig 2.

Though MTD-f-step mentioned above is not useful when the initial range is estimated preciously, we consider that MTD-f-step is efficient for the actual game trees.

4.2 MTD-f-alpha-beta

In this section, we describe MTD-f-alpha-beta algorithm which is the combination of MTD-f-step and the window search [11]. Window search algorithms are used in various SHOGI programs [2]. In the same way as MTD-f-step with the step size 100, the range is narrowed while the true minimax value is in the range. The algorithm stops when the true minimax value is in the outside of range. Then this causes the MTD-f-alpha-beta start the window search, which is alpha-beta algorithm with non-infinite alpha and

```
struct move MTD-f-step(){
   int UpperBound = +infinity;
   int LowerBound = -infinity;
   int f,r,step;
   struct res Res;
   // Res.V has minimax value
   // Res.M has best move;

   step = (stepsize);
   f = (previous search result or search result with depth-2);

   while(UpperBound != LowerBound){
      if(f==LowerBound) r=f+1; else r=f;
      Res = ZeroWindowSearch(r-1,r);
      if(Res.V<r) UpperBound = Res.V;
      else LowerBound = Res.V;

      if(UpperBound != +infinity &&
           LowerBound != -infinity){
            step = step / 2;
            if(step<10) step = 0;
      }

      if(UpperBound == Res.V) f = Res.V - step;
      else f = Res.v + step;
      if(UpperBound < f) f=UpperBound;
      if(LowerBound > f) f=LowerBound;
   }
   return Res.M;
}
```

Figure 2. Pseudo-code of MTD-f-step

beta values, ranged by the previous range of MTD-f-step. Since MTD-f-step algorithm guarantees the existence of the true minimax value in the range, the window search correct finds the true minimax value. A pseudo-code of MTD-f-alpha-beta is shown in Fig 3.

4.3 Consideration for Transposition Table

In the same way as MTD-bi and MTD-step, the two algorithms we proposed above are efficient, if a transposition table holds the lower and the upper bounds of positions. We implement our transposition table with one bound for the technical reason. Actually, the true minimax value is introduced if our transposition table holds a bound under the little sacrifice of the performance. A true value of either the upper limit or the lower limit is included in the transposition table. Moreover, the result of the newest

```
struct move MTD-f-alpha-beta(){
   int UpperBound = +infinity;
   int LowerBound = -infinity;
   int f,step;
   struct res Res;
   // Res.V has minimax value
   // Res.M has best move;

   step = (stepsize);
   f = (previous search result or search result with depth-2);

   while(UpperBound == +infinity ||
            LowerBound == -infinity){
      Res = ZeroWindowSearch(f-1,f);
      if(Res.V<f) UpperBound = Res.V;
      else LowerBound = Res.V;

      if(UpperBound == Res.V) f = Res.V - step;
      else f = Res.v + step;
   }
   Res = alphabeta(LowerBound,UpperBound);
   return Res.M;
}
```

Figure 3. Pseudo-code of MTD-f-alpha-beta

search is put until a true value is obtained. The Fig 4 and 5 show the illustrations of changing “f” values in both algorithms.

5. Experiments

5.1 Experimental Environment

In this experimentation, we apply the variations of MTD algorithm proposed the previous section to SHOGI. This section describes the experimental environment.

The SHOGI system "MATTARI YUCHAN" which we have been developing for a few past years is used for evaluating our proposed method. The system searches game trees, using the alpha-beta method with an iterative deepening method and a move ordering, a probabilistic extension of the depth of game trees and a singular extension method for middle games. But we used the old version in this experiment, so the probabilistic extension and iterative deepening method were not mounted in the system. Opening book is used for the beginning of games and PN search is used for endgames. The evaluation functions which evaluate positions of the middle games have several factors like the value of pieces and the positional values of each piece. To find the effect of our

proposed method, we fundamentally use the evaluation function and the search extension which controls the depth of game tree by the kind of move like taking pieces, check move and so on.

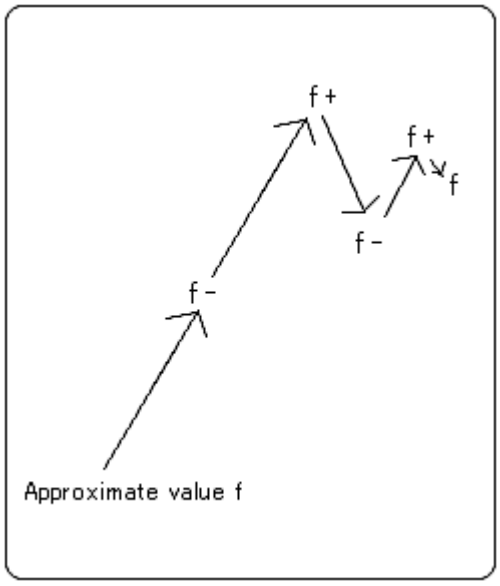


Figure 4. Convergence transitions of MTD-f-step

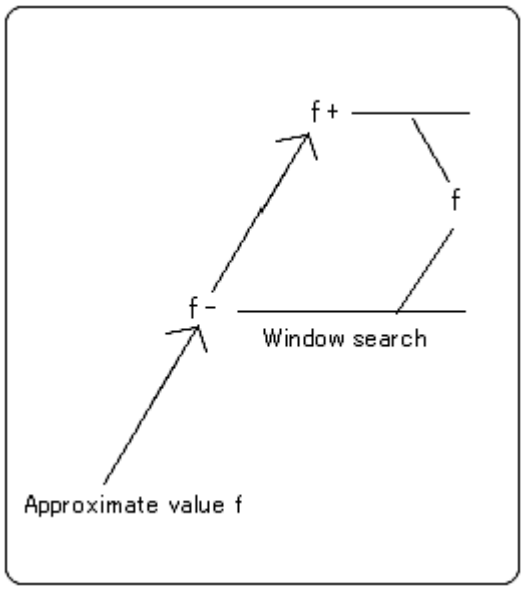


Figure 5. Convergence transitions of MTD-f- α β

MTD algorithms yield the best moves for given positions in the meaning of alpha-beta method. So the selected moves have to be the same exactly. However, we observed the different moves between the variations of MTD algorithms and alpha-beta method. This is because of the information held on the transposition table. Usually the size of game tree generated by MTD method is smaller than that generated by alpha-beta method. In this case, the positional information on the transposition table generated by alpha-beta method is informative. This phenomenon is caused by the probabilistic extension.

In our experiment, we compare the variation of MTD method with Nega-Scout method by the number of search nodes, search time, and the number of times of calling zero window algorithm. In addition, we evaluate these methods by games trying 255 games. Each game is done within 10 minutes. The searching depth is limited to 15 for the normal positions and 25 for the capturing positions. Forward pruning that limits the number of candidate hands according to depth is used for the search. Moreover, important moves such as the checkmate are adjusted to search deeply.

MTD+infinity, MTD-infinity, MTD-f, MTD-f-step, and MTD-f-alpha-beta were evaluated. The evaluation value of a best move in previous search result was used as an approximate value "f". Moreover, MTD-f-alpha-beta used Nega-Scout search as the window search. An error distribution with an approximate value and a mini-max value is shown in Fig. 6.

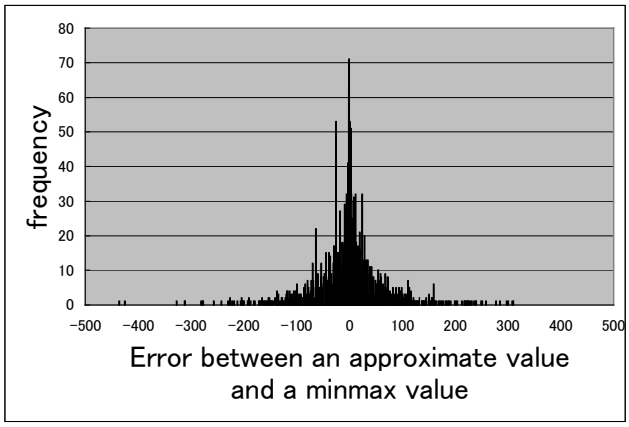


Figure 6. The error distribution of the approximate value and the mini-max value

However, the information concerned with infinitely is excluded from this figure. The number of root position is 2282. The average value of this error distribution is 0.2, and standard deviation is 65.0. Here, the value of a pawn in the evaluation function is about 20. MTD-f-step is expected for an evaluation function with the wide distribution of errors, since the next upper or lower bound is re-set over a return value of zero window searching. SHOGI is a game such that the design of evaluation

function is difficult. If it is possible to use an evaluation function with the narrow distribution of errors in SHOGI, MTD-f solves SHOGI positions within several times of calling zero window algorithm [3]. However, such an evaluation function generally is not available in SHOGI.

For MTD-f-step and MTD-f-alpha-beta, we also investigated the effect of the step size. We investigated the effect of the initial "f" in MTD-f-step. For the MTD-f-alpha-beta we try various step sizes like 100, 50, and 25. Since the difference of evaluation values between each depth is not ignored in SHOGI endgames, the number of times of calling zero window algorithm is expected to increase. So we limited the number of zero window algorithm to 1000.

5.3 Result of Games

The data obtained from games are shown in the table 2. Since NegaScout algorithm is the most efficient algorithm used in state of the art game tree searching, it is meaningful to compare our method with it. The evaluation is done on the number of search nodes and the search time for actual SHOGI positions.

Table 2. Performance comparison with MTD+infinity, MTD-infinity, MTD-f, MTD-f-step, and MTD-f-alpha-beta

	When NegaScout's is 1		NS	MTD		Average Calls zero window
method	AN	Search time	Nodes /s	Nodes /s	DBM	
mtd+infinity	1.482	0.849	84165	146990	40	43.32
mtd-infinity	1.274	0.806	86487	136761	42	32.97
mtd_f	0.900	0.646	85161	118700	43	12.00
mtd_s_100	0.914	0.662	84720	117008	33	6.98
mtd_s_50	0.878	0.634	83069	114937	40	6.50
mtd_a_100	0.911	0.670	85753	116658	30	2.11
mtd_a_50	0.872	0.637	83441	114166	48	2.35
mtd_a_25	0.852	0.629	84135	113983	42	2.81

In the table 2, "AN" is the number of search nodes, and "DBM" is a number which best move of the alpha-beta method disagrees with the variations of the MTD method.

The average of number of nodes in NegaScout is about 34000. MTD-f never returns the same result as the $\alpha\ \beta$ search as noted above. DBM is the frequency. Moreover, this trial number of times is about 30000. And "mtd_s" is MTD-f-step and "mtd_a" means MTD-f-alpha-beta. The number of the right-hand side is the step size.

We find that the performances of the variations of MTD are higher than Nega-Scout. Some methods can search more nodes than Nega-Scout. Since MTD searches the same node two or more times, this is caused by calling position values from the transposition table.

For MTD-f-step and MTD-f-alpha-beta, the performances differ depending on the step size. As shown the table 2, the narrow step size causes better results. MTD-f-alpha-beta with the 25 step size shows the best performance among experimental algorithms.

6. Discussions

Depending on the step size, MTD-f-step and MTD-f-alpha-beta search game-trees faster than MTD-f. This is considered to be related to the number of times of calling zero window algorithm. Although the number of the times in MTD-f is 12 times on an average, MTD-f-step calls it 7 times. Therefore, MTD-f-step and MTD-f-alpha-beta uses the values of "f" effectively.

For MTD-f-alpha-beta, when the number of times of calling zero window algorithm increases, the better performance is obtained, though the too much calls of zero window algorithm is not better. It is expected to be optimal from this discussion that the number of calling zero window algorithm is three.

Experimental results show that our methods improve the performance of MTD-f in SHOGI positions included little errors of estimating the true minimax value, as shown in the figure 5. We expect the drastic results, when the evaluation function is not useful for estimating values of future positions.

When we experimented deeper depth search shown in the preliminary experiment, there were no difference of performance between MTD-f and Nega-Scout. This can be the same for MTD-f-$\alpha\ \beta$. As the reason why the speed of search got worse, we start with an assumption that the overhead by the increase of the number of times of MT becomes severe because of deeper search.

The table 3 shows the total number of search nodes and the total search time at the depth 6 and 7 random trees with forward pruning setup of SHOGI system. In the depth 6, the search speed of MTD is about 1.6 times for the Nega-Scout. On the other hand, it is about 1.8 times of the search speed when the forward pruning method is not introduced. From this, the forward pruning made the performance of MTD worse. In addition, since search extension is used in the SHOGI system, the number of successors decreases nodes in the deep depth of game trees.

So we consider that MTD method is not suitable for a system with strong forward pruning strategies.

7. Conclusions

We proposed in this paper the new variation of MTD, MTD-f-step and MTD-f-alpha-beta, which are for the reduction of the searching nodes by the setting of the initial range and the way of re-set ranges after zero window algorithm. Our experiment is done for random-generated game trees and actual SHOGI game trees. We confirmed the improvement of our proposed method against the original MTD-f method.

As future works, the design of a transposition table for MTD-step is remained.

Table 3. The total number of search nodes and the total search time for the random tree with forward pruning

depth	Total number of search nodes		
	MTD	NegaScout	$\alpha\ \beta$
6	343739537	628103273	1069207656
7	1816564793	2806926235	5916587602
depth	Total search time(s)		
	MTD	NegaScout	$\alpha\ \beta$
6	325.6	528.57	829.17
7	1489.67	2319.56	4371.32

Bibliography

[1] Aske Plaat, Jonathan Schaeffer, Wim Pijls, Arie de Bruin :A New Paradigm for Minimax Search, Technical Report TR-CS-94-18, 1994

[2] Hitoshi Matsubara: Advances of Computer SHOGI vol.3 (in Japanese) , kyoritsu publisher, 2000

[3] Makoto Miwa, Daisaku Yokoyama, Kenziro Taura, Takashi Chikayama: Massive Parallelization of Game Tree Search (in Japanese), The 65th national conferences of Information Processing Society of Japan, 2003

[4] Aske Plaat, Jonathan Schaeffer, Wim Pijls and Arie De Bruin :Best-first and depth-first minimax search, Artificial Intelligence 87(1-2):255-293, 1996

[5] Aske Plaat, Jonathan Schaeffer, Wim Pijls and Arie De Bruin :SSS* = α - β + TT, Technical Report TR-CS-94-17, Department of Computing Science, University of Alberta Edmonton, AB, Canada, December 1994

[6] Aske Plaat: Research Re: search & Re-search, PhD Thesis, Tinbergen Institute and Department of Computer Science, Erasmus University Rotterdam, Thesis Publishers, Amsterdam, The Netherlands, June 20, 1996

[7] Aske Plaat, Jonathan Schaeffer, Wim Pijls and Arie De Bruin :A minimax algorithm better than Alpha-Beta? No and Yes, Technical Report 95-15, University of Alberta, Department of Computing Science, Edmonton, AB, Canada, June 1995.

[8] Jonathan Schaeffer and Aske Plaat :New Advances in Alpha-Beta Searching, Proceedings of the 24th ACM Computer Science Conference, pp. 124-130, 1996.

[9] Aske Plaat, Jonathan Schaeffer, Wim Pijls, Arie de Bruin : An Algorithm Faster than NegaScout and SSS* in Practice, at the Computer Strategy Game Programming Workshop at the World Computer Chess Championship, May 26, 1995

[10] J. Pearl, Scout: A simple Game-Searching Algorithm with Proven Optimal Properties, First Annual National Conference on Artificial Intelligence, Stanford, 1980

[11] G. Baudet, The Design and Analysis of Algorithms for Asynchronous Multiprocessors, PH.D. thesis, Department of Computer Science, Carnegie-Mellon University, 1878

[12] Alexander Reinefeld. Spielbaum Suchverfahren. Informatik-Fachberichte 200. Springer Verlag, 1989.

[13] George C. Stockman. A minimax algorithm better than alpha-beta? Artificial Intelligence, 12(2):179-196, 1979.

[14] J. Mandziuk and D. Osman. Alpha-Beta Search Enhancements with a Real-Value Game-State Evaluation Function, ICGA Journal Vol. 27, No.1, 2004

Monte Carlo Planning in RTS Games

Michael Chung, Michael Buro, and Jonathan Schaeffer
Department of Computing Science, University of Alberta
Edmonton, Alberta, Canada T6G 2E8
{mchung,mburo,jonathan}@cs.ualberta.ca

Abstract- Monte Carlo simulations have been successfully used in classic turn–based games such as backgammon, bridge, poker, and Scrabble. In this paper, we apply the ideas to the problem of planning in games with imperfect information, stochasticity, and simultaneous moves. The domain we consider is real–time strategy games. We present a framework — MCPlan — for Monte Carlo planning, identify its performance parameters, and analyze the results of an implementation in a capture–the–flag game.

1 Introduction

Real–time strategy (RTS) games are popular commercial computer games involving a fight for domination between opposing armies. In these games, there is no notion of whose turn it is to move. Both players move at their own pace, even simultaneously; delays in moving will be quickly punished. Each side tries to acquire resources, use them to gain information and armaments, engage the enemy, and battle for victory. The games are typically fast–paced and involve both short–term and long–term strategies. The games are well–suited to Internet play. Many players prefer to play against human opponents over the Internet, rather than play against the usually limited abilities of the computer artificial intelligence (AI). Popular examples of RTS games include WarCraft [1] and Age of Empires [2].

The AI in RTS games is usually achieved using scripting. Over the past few years, scripting has become the most popular representation used for expressing character behaviours. Scripting, however, has serious limitations. It requires human experts to define, write, and test the scripts comprised of 10s, even 100s, of thousands of lines of code. Further, the AI can only do what it is scripted to do, resulting in predictable and inflexible play. The general level of play of RTS AI players is weak. To enable the AI to be competitive, game designers often give AI access to information that it should not have or increase its resource flow.

Success in RTS games revolves around planning in various areas such as resource allocation, force deployment, and battle tactics. The planning tasks in an RTS game can be divided into three areas, representing different levels of abstraction:

1. **Unit control (unit micromanagement)**. At the lowest level is the individual unit. It has a default behaviour, but the player can override it. For example, a player may micromanage units to improve their performance in battle by focusing fire to kill off individual enemy units.

2. **Tactical planning (mid–level combat planning).** At this level, the player decides how to conduct an attack on an enemy position. For example, it may be possible to gain an advantage by splitting up into two groups and simultaneously attacking from two sides.

3. **Strategic planning (high–level planning).** This includes common high–level decisions such as when to build up the army, what units to build, when to attack, what to upgrade, and how to expand into areas with more resources.

In addition, there are other non–strategic planning issues that need to be addressed, such as pathfinding.

Unit control problems can often be handled by simple reactive systems implemented as list of rules, finite state machines, neural networks, etc. Tactical and strategic planning problems are more complicated. They are real–time planning problems with many states to consider in the absence of perfect information. It is apparent that current commercial RTS games deal with this in a simple manner. All of the AI's strategies in the major RTS games are scripted. While the scripts can be quite complex, with many random events and conditional statements, all the strategies are still predefined beforehand. This limitation results in AI players that are predictable and thus easily beaten. For casual players, this might be fun at first, but there is no re–playability. It is just no fun to beat an AI player the same way over and over again.

In RTS games, there are often hundreds of units that can all move at the same time. RTS games are fast–paced, and the computer player must be able to make decisions at the same speed as a human player. At any point in time, there are many possible actions that can be taken. Human players are able to quickly decide which actions are reasonable, but current state–of–the–art AI players cannot. In addition, players are faced with imperfect information, i.e. partial observability of the game state. For instance, the location of enemy forces is initially unknown. It is up to the players to scout to gather intelligence, and act accordingly based on their available information. This is unlike the classical games such as chess, where the state is always completely known to both players. For these reasons, heuristic search by itself is not enough to reason effectively in an RTS game. For planning purposes, it is simply infeasible for the AI to think in terms of individual actions. Is there a better way?

Monte Carlo simulations have the advantage of simplicity, reducing the amount of expert knowledge required to achieve high performance. They have been successfully used in games with imperfect information and/or stochastic elements such as backgammon [14], bridge [9], poker [5], and Scrabble [11]. Recently, this approach has been tried in two-player perfect-information games with some success (Go [6]). A framework for using simulations in a game–playing program is discussed in [10], and the subtleties of getting the best results with the smallest sample sizes is discussed in [12].

Can Monte Carlo simulations be used for planning in RTS games? If so, then the advantages are obvious. Using simulations would reduce the reliance on scripting, resulting in substantial savings in program development time. As well, the simulations will have no or limited expert bias, allowing the simulations to explore possibilities not covered by expert scripting. The result could be a stronger AI for RTS games and a richer gaming experience.

The contributions in this work are as follows:

1. Design of a Monte Carlo search engine for planning (MCPlan) in domains with imperfect information, stochasticity, and simultaneous moves.
2. Implementation of the MCPlan algorithm for decision making in a real–time capture–the–flag game.
3. Characterization of MCPlan performance parameters.

Section 2 describes the MCPlan algorithm and the parameters that influence its performance. Section 3 discusses the implementation of MCPlan in a real–time strategy game built on top of the free ORTS RTS game engine [7]. Section 4 presents experimental results. We finish the paper by conclusions and remarks on future work in this area.

2 Monte Carlo Planning

Adversarial planning in imperfect information games with a large number of move alternatives, stochasticity, and many hidden state attributes is very challenging. Further complicating the issue is that many games are played with more than two players. As a result, applying traditional game–tree search algorithms designed for perfect information games that act on the raw state representation is infeasible. One way to make look–ahead search work is to abstract the state space. An approach to deal with imperfect information scenarios is sampling. The technique we present here combines both ideas.

Monte Carlo sampling has been effective in stochastic and imperfect information games with alternating moves, such as bridge, poker, and Scrabble. Here, we want to apply this technique to the problem of high–level strategic planning in RTS games. Applying it to lower level planning is possible as well. The impact of individual moves — such as a unit moving one square — requires a very deep search to see the consequences of the moves. Doing the search at a higher level of abstraction, where the execution of plan becomes a single "move", allows the program to envision the consequences of actions much further into the future (see Section 2.2).

Monte Carlo planning (MCPlan) does a stochastic sample of the possible plans for a player and selects the plan to execute that has the highest statistical outcome. The advantage of this approach is that it reduces the amount of expert–defined knowledge required. For example, Full Spectrum Command [3] requires extensive military–strategist–defined plans that the program uses — essentially forming an expert system. Each plan has to be fully specified, including identifying the scenarios when the plan is applicable, anticipating all possible opponent reactions, and the consequences of those reactions. It is difficult to get an expert's time to define the plans in precise detail, and more difficult to invest the time to analyze them to identify weaknesses, omissions, exceptions, etc. MCPlan assumes the existence of a few basic plans (e.g. explore, attack, move towards a goal) which are application dependent, and then uses sampling to evaluate them. The search can sample the plans with different parameters (e.g. where to attack, where to explore) and sequences of plans—for both sides. In this section, we describe MCPlan in an application–independent manner, leaving the application–dependent nuances of the algorithm to Section 3.

2.1 Top–Level Search

The basic high–level view of MCPlan is as follows, with a more formal description given in Figure 1:

1. Randomly generate a plan for the AI player.
2. Simulate randomly–generated plans for both players and execute them, evaluate the game state at the end of the sequence, and compute how well the selected plan seems to be doing (`evaluate_plan`, Section 2.3).
3. Record the result of executing the plan for the AI player.
4. Repeat the above as often as possible given the resource constraints.
5. Choose the plan for the AI player that has the best statistical result.

The variables and routines used in Figure 1 are described in subsequent subsections.

The top–level of the algorithm is a search through the generated plans, looking for the one with the highest evaluation. The problem then becomes how best to generate and evaluate the plans.

2.2 Abstraction

Abstraction is necessary to produce a useful result and maintain an acceptable run–time. Although this work is discussed in the context of high–level plans, the implementor is free to choose an appropriate level of abstraction, even at the level of unit control, if desired. However, since MCPlan relies on the power of statistical sampling, many data points are usually needed to get a valid result. For best performance, it is important that the abstraction level be chosen to make the searches fast and useful.

In Figure 1, `State` represents an abstraction of the current game state. The level of abstraction is arbitrary, and in simple domains it may even be the full state.

2.3 Evaluation Function

As in traditional game–playing algorithms, at the end of a move sequence an evaluation function is called to assess how good or bad the state is for the side to move. This typically requires expert knowledge although the weight or

```
// Plan: contains details about the plan
// For example, a list of actions to take
class Plan {
  // returns true if no actions remaining in the plan
  bool is_completed();
  // [...] (domain specific)
};

// State: AI's knowledge of the state of the world
class State {
  // return evaluation of the current state
  // (domain specific implementation)
  float eval();
  // [...] (domain specific)
};

// MCPlan Top-Level
Plan MCPlan(
  State state,   // current state of the world
  int num_plans, // number of plans to evaluate
  int num_sims,  // simulations per evaluation
  int max_t)     // max time steps per simulation
{
  float best_val = -infinity;
  Plan best_plan;

  for (int i = 0; i < num_plans; i++) {
    // generate plan using (domain-specific) plan generator
    Plan plan = generate_plan(state);
    // evaluate using the number of simulations specified
    float val = evaluate_plan(plan, state, num_sims, max_t);
    // keep plan with the best evaluation
    if (val > best_val) {
      best_plan = plan;
      best_val = val;
    }
  }
  return best_plan;
}
```

Figure 1: MCPlan: top–level search

importance of each piece of expert knowledge can be evaluated automatically, for example by using temporal difference learning [13]. For most application domains, including RTS games, there is no easy way around this dependence on an expert. Note that, unlike scripted AI which requires a precise specification and extensive testing to identify omissions, evaluation functions need only give a heuristic value.

2.4 Plan Evaluation

Before we describe the search algorithm in more detail, let us define the key search parameters. These are variables that may be adjusted to modify the quality of the search, as well as the run–time required. The meaning of these parameters will become more clear as the search algorithm is described.

1. `max_t`: the maximum time, in steps or moves, to look ahead when performing the simulation–based evaluation.
2. `num_plans`: the total number of plans to randomly generate and evaluate at the top–level.
3. `num_sims`: the number of move sequences to be considered for each plan.

The `evaluate_plan()` function is shown in Figure 2. Each plan is evaluated `num_sims` times. A plan is evaluated using `simulate_plan()` by executing a series of plans for both sides and then using an evaluation function to assess the resulting state. In the pseudo–code given, the value of a plan is the minimum of the sample values (a pessimistic assessment). Other metrics are possible, such as

```
// Evaluate Plan Function. Takes minimum of num_sims
// plan simulations (pessimistic)
float evaluate_plan(Plan plan, State state,
                    int num_sims, int max_t)
{
  float min = infinity;
  for (int i = 0; i < num_sims; i++) {
    float val = simulate_plan(plan, state, max_t);
    if (val < min) min = val;
  }
  return min;
}
```

Figure 2: MCPlan: plan evaluation

```
// Simulate Plan. Perform a single simulation with the given
// plan and return the resulting state's evaluation.
float simulate_plan(Plan plan, State state, int max_t)
{
  State bd_think = state;
  Plan plan_think = plan;

  // generate a plan for the opponent (domain specific)
  Plan opponent_plan = generate_opponent_plan(state);

  while (true) {
    // simulate a single time step in the world
    // (domain specific)
    simulate_plan_step(plan_think, opponent_plan, bd_think);

    // check if maximum time steps has been simulated
    if (--max_t <= 0) break;

    // check if plan has been completed
    if (plan_think.is_completed()) break;

    // check if the opponent's plan has been completed
    if (opponent_plan.is_completed()) {
      // if so, generate a new opponent plan
      opponent_plan = generate_opponent_plan(bd_think);
    }
  }
  return bd_think.eval();
}
```

Figure 3: MCPlan: plan simulation

taking the maximum over all samples, the average of the samples, or a function of the distribution of values. Also, in the presented formulation of MCPlan information about the plan chosen by the player is implicitly leaking to the opponent. This turns a possible imperfect information scenario into one of perfect information leading to known problems [8]. We will address this problem in future work. Here, we restrict ourselves to a simple form which nevertheless may be adequate for many applications. Each data point for a plan evaluation is done using `simulate_plan()`. Both sides select a plan and then executes it. This is repeated until time runs out. The resulting state of the game is assessed using the evaluation function. Note that opponent plans can cause interaction; how this is handled is application dependent and it is discussed in Section 3. The `evaluate_plan()` function calls `simulate_plan()` `num_sims` times, and takes the minimum value. Figure 3 shows the `simulate_plan()` function.

2.5 Comments

MCPlan is similar to the stochastic sampling techniques used for other games. The fundamental difference — besides obvious semantic ones such as not requiring players to alternate moves — is that the "moves" can be executed at an abstract level. Abstraction is key to getting the depths of search needed to have long–range vision in RTS games.

MCPlan lessens the dependence on expert–defined knowledge and scripts. Expert knowledge is needed in two places:

1. Plan definitions. A plan can be as simple or as detailed as one wants. In our experience, using plan *building blocks* is an effective technique. Detailed plans are usually composed of a series of repeated high–level actions. By giving MCPlan these actions and allowing it to combine them in random ways, the program can exhibit subtle and creative behaviour.

2. Evaluation function. Constructing accurate evaluation functions for non–trivial domains requires expert knowledge. In the presence of look–ahead search, however, the quality requirements can often be lessened by considering the well–known trade–off between search and knowledge. A good example is chess evaluation functions, which — combined with deep search — lead to World–class performance, in spite of the fact that the used features have been created by programmers rather than chess grandmasters. Because RTS games have much in common with classical games, we expect a similar relationship between evaluation quality and search effort in this domain, thus mitigating the dependency on domain experts.

3 Capture the Flag

Commercial RTS games are complex. There are many different variations, some involving many RTS game elements such as resource gathering, technology trees, and more. To more thoroughly evaluate our RTS planners, we limit our tests to a single RTS scenario, capture–the–flag (CTF). Our CTF game takes place on a symmetric map, with vertical and horizontal walls. The two forces start at opposing ends of the map. Initially the enemy locations are unknown. The enemy flag's location is known — otherwise much initial exploration would be required. This is consistent with most commercial RTS games, where the same maps are used repeatedly, and the possible enemy locations are known in advance.

The rules of our CTF game are relatively simple. Each side starts with a small fixed number of units, located near a home base (post), and a flag. Units have a range in which they can attack an opponent. A successful attack reduces the nearby enemy unit's hit–points. When a unit's hit–points drops to zero, the unit is "dead" and removed from the game.

The objective of CTF is to capture the opponent's flag. Each unit has the ability to pick–up or drop the enemy flag. To win the game, the flag must be picked up, carried, and dropped at the friendly home base. If a unit is killed while carrying the flag, the flag is dropped at the unit's location, and can later be picked up by another unit. A unit cannot pick up its own side's flag at any time.

Terrain is very important to CTF. For most of our tests we keep it simple and symmetric to avoid bias towards either side. However, even with more complex terrains, while there may be a bias towards one side, it is expected that planners that perform better on symmetric maps will also perform better on complex maps. While CTF does not capture all the elements involved in a full RTS game — such as economy and army–building — it is a good scenario for testing planning algorithms. Many of the features of full RTS games are present in CTF — including scouting and base defense.

Before we discuss how we applied MCPlan to a CTF game we first describe the simulation software we used.

3.1 ORTS

ORTS (Open RTS) is a free software RTS game engine which is being developed at the University of Alberta and licensed under the GNU General Public License. The goal of the project is to provide AI researchers and hobbyists with an RTS game engine that simplifies the development of AI systems in the popular commercial RTS game domain. ORTS implements a server–client architecture that makes it immune to map–revealing client hacks which are a widespread plague in commercial RTS games. ORTS allows users to connect *whatever* client software they wish — ranging from distributed RTS game AI to feature–rich graphics clients. The CTF game which we use for studying MCPlan performance has been implemented within the ORTS framework. For more information on the status and development of ORTS we refer readers to [4][7].

3.2 CTF Game State Abstraction

In the state representation, the map is broken up into tiles (representing a set of possible unit locations). Units are located on these tiles, and their positions are reasoned about in terms of tiles, rather than exact game coordinates. The state also contains information about the units' hit–points, as well as locations of walls and flags.

3.3 Evaluation Function

We tried to keep our evaluation function simple and obvious, without relying on a lot of expert knowledge. The evaluation function for our CTF AI has three primary components: material, exploration/visibility, and capture/safety. The first two components are standard to any RTS game. The third component is specific to our CTF scenario. Without it, the AI would have no way to know that it was actually playing a CTF game, and it would behave as if it was a regular battle. In each component the difference of the values for both players is computed. In the following we briefly give details of the evaluation function.

Material

The most important part of any RTS game is material. In most cases, the side with the most resources — including military units, buildings, etc. — is the victor. Thus, maximizing material advantage is a good sub–goal for any planning AI. This material can later be converted into a decisive advantage such as having a big enough army to eliminate the enemy base. There is a question of how to compare healthy units to those with low hit–points. For example, while it may be clear that two units each with 50% health are better than one unit with 100% health, which would be better, one unit with 100% health, or two units with 25% health? While the two units could provide more firepower, they could also

be more quickly killed by the enemy. There are different situations where the values of these units may be different. For our tests, we provide a simple solution: each unit provides a bonus of $\sqrt{0.01 \times \text{hp}}$. The maximum hp (hit–point) value is 100. Thus, each live unit has a value of between 0.1 and 1. The value for friendly units is added to our evaluation, and enemy units values are subtracted. Taking the square root prefers states which — for a constant hit–point total — have a more balanced hit–point distribution.

Exploration and Visibility
When not doing something of immediate importance, such as fighting, exploring the map is very important. The side with more information has a definite advantage. Keeping tabs on the enemy, finding out the lay of the land, and discovering the location of obstacles are all important. The planner cannot accurately evaluate its plans unless it has a good knowledge of the terrain and of enemy forces and their locations. The value of information is reflected by these evaluation function sub–components:

- Exploration bonus: $0.001\times$ # of explored tiles, and
- Vision bonus: $0.0001\times$ # visible tiles.

Note that the bonus values can be changed or even learned.

Flag Capture and Safety
To win a CTF game, the opponent's flag has to be captured. It is important to encourage the program to go after the enemy's flag, while at the same time ensuring that the program's flag remains safe:

- Bonus for being close to enemy flag: +0.1 per tile,
- Bonus for possession of enemy flag: +1.0,
- Bonus for bringing enemy flag closer to our base: +0.2 per tile, and
- Similar penalties apply if the enemy meets these conditions.

Note that all these heuristic values have been manually tuned. Machine learning would be a way to more reliably set these values.

Combining the Components
The simplest thing to do, and what we do right now, is have constant factors for adding the three components together. There are exceptions where this is not the best approach. For example, if we are really close to capturing the enemy flag, we may choose to ignore the other components, such as exploration. Such enhancements are left as future work. In our experiments we give each component equal weight.

Evaluation Function Quality
We can perform experiments to test the effectiveness of our evaluation function. For example, we could measure the time it takes to capture the flag if there are no enemy units. This removes all tactical situations and focuses on testing that the evaluation function is correctly geared towards capturing the enemy flag. Playing the MCPlan AI against a completely random AI also provides a good initial test of the evaluation function. A random evaluation function would perform on the same level as the random AI, whereas a better evaluation function would win more often.

3.4 Plan Generation

There are two types of plan generation used in this project: random and scripted. The random plans are simple and are described below. The scripted plans are slightly more sophisticated, but still quite simple. Only the random plans are used in this implementation, as we do not have many scripted plans implemented.

Random Plans
A random plan consists of assigning a random nearby destination for each unit to move to. That is, for each unit, a nearby unoccupied destination tile is selected. The maximum distance to the destination is determined by the `max_dist` variable. The A* pathfinding algorithm is then used to find a path for each unit. Note that collisions are possible between the units, but are ignored for planning purposes. We did not implement any group–based pathfinding, although it is a possible enhancement.

Scripted Plans
We have implemented a small number of action scripts which provide test opponents for the MCPlan algorithm. As previously mentioned, scripted plans have many disadvantages — most notably, the need to have an expert define, refine and test them. However, there is the possibility that given a set of scripted plans, applying the search and simulation algorithms described in this paper can result in a stronger player.

3.5 Plan Step Simulation

Simulation must be used because when the planner evaluates an action, the result of that action cannot be perfectly determined because of hidden enemy units, unknown enemy actions, randomized action effects, etc. Also, as our simulation acts on an abstracted state description, the computation should be much faster. The plan step simulation function takes the given plans for the friend and enemy sides and executes one–tile moves for each side. Unit attacks are then simulated by selecting the nearest opposing unit for each unit, and reducing its hit–points. The attacks may not match what would happen in the actual game, due to many reasons. For example, units may seem to be in range but actually they are not, due to the abstracted distances. Also, in some games, the attack damage is random, so the damage results may not be exactly the same as what will happen in the game. However, it is expected that with a large enough value of `num_evals`, the final result should be more statistically accurate.

3.6 Other Issues

In this subsection we discuss some implementation issues related to developing and testing a search/simulation based RTS planning algorithm such as MCPlan.

Map Generation
It is clear that in performing the tests, map generation is a hard problem. To produce an unbiased map, the map should be completely symmetric. A more complex asymmetric map could favor one side. In addition, it is possible that

different types of maps could favour different AI's. For our tests we use a simple symmetric map, to avoid most of these issues. It is expected — and to be confirmed — that on more complex and on randomly generated maps, the conclusions we draw from our experiments should still hold.

Server Synchronization
The tests should be run with server synchronization turned on. This option tells the ORTS server to wait for replies from both clients before continuing on to the next turn. In the default mode with synchronization off, the first player to connect may possibly have an advantage, due to being able to move while the second player's process is still initializing its GUI, etc. The server synchronization option eliminates this possible source of bias, as well as reducing the randomness caused by random network lag.

Interactions and Replanning
As players interact previous planning may quickly become irrelevant. In many cases, replanning must occur. Not every interaction should result in replanning. This would result in too frequent replanning, which would slow down the computation while perhaps not improving the decision quality much. Instead, only important interactions should result in replanning. Possible such interactions are: "a unit is destroyed," "a unit is discovered," or "a flag is picked up." Note that attacks, while important, happen too frequently and thus should not trigger replanning.

4 Experiments

In this section, we investigate the performance issues of MCPlan on our CTF game.

4.1 Experimental Design

Each experimental data point consisted of a series of games between two CTF programs. The experiments were performed on 1.2 GHz PCs with 1 GB of RAM. Note that because the experiments were synchronized by the ORTS server the speed of the computer does not affect the results. Each data point is based on the results of matching two programs against each other for 200 games. For a given map, two games are played with the programs playing each side once. A game ends when a flag is captured, or one side has all their men eliminated. A point is awarded to the winning side. Draws are handled depending on the type of draw. If the game times out and there is no winner, then neither side gets a point. If both sides achieve victory at exactly the same time, then both sides get a point. The reported win percentage is one side's points divided by the total points awarded in that match. In a match with no draws the total points is equal to the number of games (200).

Maps
Figure 4 shows the maps that have been used in the experiments. Their dimensions are 20 by 20 tiles. By default each side starts with five men.

Search Parameters
The `max_dist` parameter is the maximum distance that a unit can move from its current position in a randomly generated plan. In all these experiments, the `max_dist` parameter is set to 6 tiles, unless otherwise stated. The unit's sight radius is set to 10 tiles, and unit's attack range is set to 5 tiles. To reduce the number of experiments needed, the number of simulations (`num_sims`) is set to be equal to the number of plans (`num_plans`). This makes sense as the number of simulations is also the number of opponent plans considered.

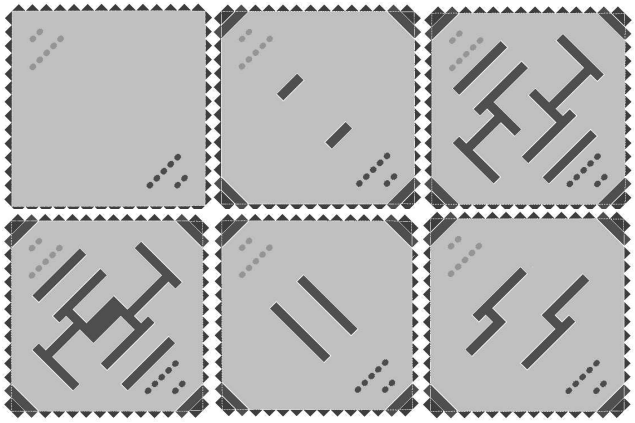


Figure 4: Maps and unit starting positions used in the experiments: **map 1** (upper left): empty terrain (this is the default), **map 2**: simple terrain with a couple of walls, **map 3**: complex terrain, **map 4**: complex terrain with dead–ends, **map 5**: simple terrain with a bottleneck, **map 6**: intermediate complexity.

Players
There are two opponents tested in these experiments other than the MCPlan player: Random and Rush–the–Flag. Random is equivalent to MCPlan running with `num_plans` = 1. It simply generates and executes a random plan, using the same plan generator as the MCPlan player. Rush–the–Flag is a scripted opponent which behaves as follows:

1. If the enemy flag is not yet captured, send all units towards the enemy flag and attempt to capture it.
2. If the enemy flag is captured, have the flag carrier return home. All other units follow the flag carrier.

While simple in design, the Rush–the–Flag opponent proves to be a strong adversary.

4.2 Results

We now investigate the performance of MCPlan against a variety of opponents and using different combinations of search parameters.

Increasing Number of Plans
In Figure 5, the performance of the MCPlan algorithm on the default map is evaluated as a function of the number of plans considered. Each data point represents the result of a player considering p plans playing against one that considers $2p$ plans. This results show that the program's play improves as the number of plans increases, but with diminishing returns. Eventually, the sample size is large enough that adding more plans results in marginal performance improvements, as expected.

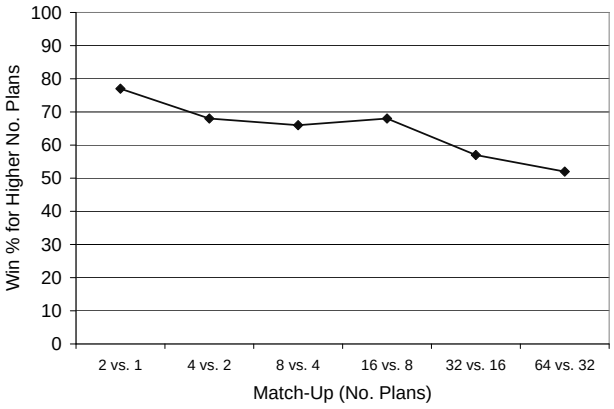


Figure 5: Increasing Number of Plans

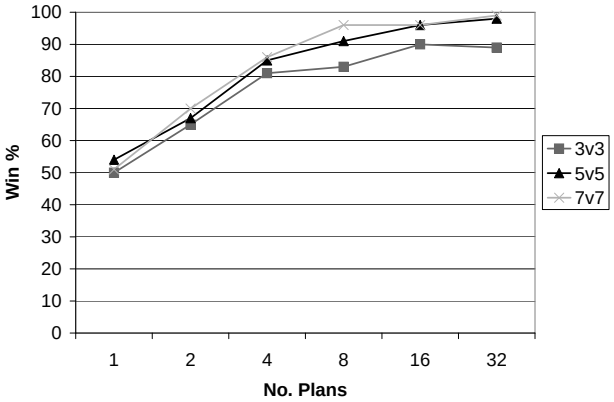


Figure 6: Different Number of Units. MCPlan vs. Random

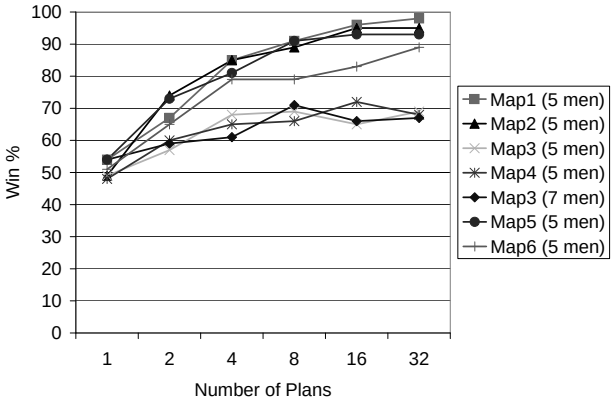


Figure 7: Different Maps. MCPlan vs. Random

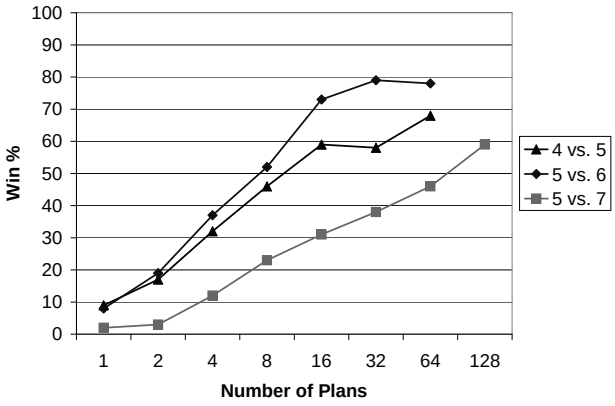


Figure 8: Less Men and Stronger AI vs. Random

Number of Units

Figure 6 shows the results when the number of units is varied. The results in the figure are for MCPlan against Random on the default map. As expected, regardless of the number of units aside, increasing the number of plans improves the performance of the MCPlan player. With a larger number of units per side, MCPlan wins more often. This is reasonable, as the number of decisions increases with the number of units, and there is more opportunity to make "smarter" moves.

Different Maps

The previous results were obtained using the same map. Do the results change significantly with different terrain? In this experiment, we repeat the previous matches using a variety of maps. Figure 7 shows the results. Note that one map has 7 men aside. The results indicate that MCPlan is a consistent winner, but the winning percentage depends on the map. The more complex the map, the better the random player performs. This is reasonable, since with more walls, there is more uncertainty as to where enemy units are located. This reduces the accuracy of the MCPlan player's simulations. In the tests using the map with a bottleneck (map 5), the performance was similar to the tests with simple maps without the bottleneck. This shows that the simulation is capable of dealing with bottlenecks, at least in simple cases.

Unbalanced Number of Units

Figure 8 illustrates the relative performance between MCPlan and Random when Random is given more men. The results show that given a sufficient number of plans to evaluate, MCPlan with less men and better AI can overcome Random with more men but a poorer AI. The results suggest that using MCPlan is strong enough to overcome a significant material advantage possessed by the weaker AI (Random). The figure shows the impressive result that 5 units with smart AI defeat 7 units with dumb AI 60% of the time when choosing between 128 plans.

Optimizing Max–Dist

A higher `max_dist` value results in longer plans, which allows more look–ahead, as well as a higher number of possible plans. The higher number of possible plans may increase the number of plans required to find a good plan.

More look–ahead should help performance. However, with too much look–ahead, noise may become a problem. The noise is due to errors in the simulation — which uses an abstracted game state — and incorrect predictions of the opponent plan. The longer we need to guess what the opponent will do, the more likely we are to make an error. So, more simulations are required to have a good chance of predicting the opponent's plan or something close enough to it.

In this experiment we vary the `max_dist` parameter to optimize the win percentage against the Random opponent on map 1 and the Rush–the–Flag opponent on map 2 (see Figure 9). The planner playing against random achieves its best performance of 94% at dist=6. Note that although one may expect MCPlan to score 100% against Random, in

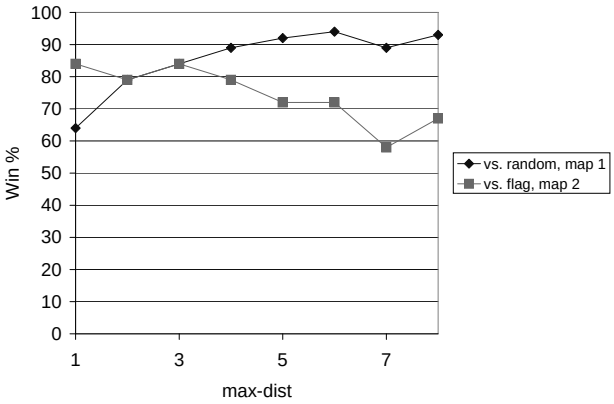


Figure 9: Optimizing Max–Dist Parameter

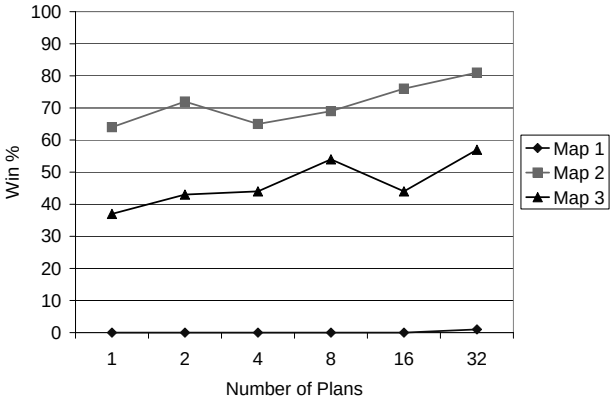


Figure 10: MCPlan vs. Rush–the–Flag Opponent

practice this will not happen. A lone unit may unexpectedly encounter a group of enemy units. Once engaged in a losing battle, it is difficult to retreat, since all units move at the same speed.

Rush–the–Flag Opponent

Figure 10 shows MCPlan playing against Rush–the–Flag. The playing strength of Rush–the–Flag is very map dependent, as it has a fixed strategy. On the first map, Rush–the–Flag wins nearly every game. Rushing is a near–optimal strategy on an empty map. On map 2, where the direct path to the other side is blocked, Rush–the–Flag is much weaker. MCPlan wins more than 60% of the time even with `num_plans`=1. With `num_plans`=32, MCPlan wins more than 80% of the time. However, on map 3, where the map is more complex and all paths to the other side are long, Rush–the–Flag again becomes a challenging opponent. However, with `num_plans`=32, MCPlan wins more than 55% of the games.

Run–Time for Experiments

In order to get more statistically valid results, the experiments were not run in real–time. Rather, they were run much faster than real–time, about 100 times faster.

While the run–time depends on the parameters, using typical parameters (map 1, 16 plans, 5 men per side) a 200–game match runs in about 80 minutes on our test machines. The average time per game is less than 30 seconds. As the planner re–plans hundreds of times per game, this results in planning times of a fraction of a second.

5 Conclusions and Future Work

This paper has presented preliminary work in the area of sampling–based planning in RTS games. We have described a plan selection algorithm – MCPlan – which is based on Monte Carlo sampling, simulations, and replanning. Applied to simple CTF scenarios MCPlan has shown promising initial results. To gauge the true potential of MCPlan we need to compare it against a highly tuned scripted AI, which was not available at the time of writing. We intend to extend MCPlan in various dimensions and apply it to more complex RTS games. For instance, it is natural to add knowledge about opponents in form of plans that can be incorporated in the simulation process to exploit possible weaknesses. Also, the top–level move decision routine of MCPlan should be enhanced to generate move distributions rather than single moves which is especially important in imperfect information games. Lastly, applying MCPlan to bigger RTS game scenarios requires us to consider more efficient sampling and abstraction methods.

Acknowledgments

We thank Markus Enzenberger and Nathan Sturtevant for valuable feedback on this paper. Financial support was provided by the Natural Sciences and Engineering Research Council of Canada (NSERC) and Alberta's Informatics Circle of Research Excellence (iCORE).

Bibliography

[1] `http://www.blizzard.com`.

[2] `http://www.ensemblestudios.com`.

[3] `http://www.ict.usc.edu/disp.php?bd=proj_games_fsc1`.

[4] `http://www.cs.ualberta.ca/~mburo/orts`.

[5] D. Billings, L. Pena, J. Schaeffer, and D. Szafron. Using probabilistic knowledge and simulation to play poker. In *AAAI National Conference*, pages 697–703, 1999.

[6] B. Bouzy and B. Helmstetter. Monte Carlo go developments. In *Advances in Computer Games X*, pages 159–174. Kluwer Academic Press, 2003.

[7] M. Buro and T. Furtak. RTS games and real-time AI research. In *Proceedings of the Behavior Representation in Modeling and Simulation Conference (BRIMS), Arlington VA 2004*, pages 51–58, 2004.

[8] I. Frank and D.A. Basin. Search in games with incomplete information: A case study using bridge card play. *AI Journal*, 100(1-2):87–123, 1998.

[9] M. Ginsberg. GIB: Steps toward an expert-level bridge-playing program. In *International Joint Conference on Artificial Intelligence*, pages 584–589, 1999.

[10] Jonathan Schaeffer, Darse Billings, Lourdes Peña, and Duane Szafron. Learning to Play Strong Poker. In J. Fürnkranz and M. Kubat, editors, *Machines That Learn To Play Games*, pages 225–242. Nova Science Publishers, 2001.

[11] B. Sheppard. *Towards Perfect Play in Scrabble*. PhD thesis, 2002.

[12] B. Sheppard. Efficient control of selective simulations. *Journal of the international Computer Games Association*, 27(2):67–80, 2004.

[13] R. Sutton and A. Barto. *Reinforcement Learning*. MIT Press, 1998.

[14] G. Tesauro. Temporal difference learning and TD-Gammon. *Communications of the ACM*, 38(3):58–68, 1995.

Fringe Search: Beating A* at Pathfinding on Game Maps

Yngvi Björnsson
School of Computer Science
Reykjavik University
Reykjavik, Iceland IS-103
yngvi@ru.is

Markus Enzenberger, Robert C. Holte and Jonathan Schaeffer
Department of Computing Science
University of Alberta
Edmonton, Alberta, Canada T6G 2E8
{emarkus,holte,jonathan}@cs.ualberta.ca

Abstract- The A* algorithm is the *de facto* standard used for pathfinding search. IDA* is a space-efficient version of A*, but it suffers from cycles in the search space (the price for using no storage), repeated visits to states (the overhead of iterative deepening), and a simplistic left-to-right traversal of the search tree. In this paper, the *Fringe Search* algorithm is introduced, a new algorithm inspired by the problem of eliminating the inefficiencies with IDA*. At one extreme, the Fringe Search algorithm expands frontier nodes in the exact same order as IDA*. At the other extreme, it can be made to expand them in the exact same order as A*. Experimental results show that Fringe Search runs roughly 10-40% faster than highly-optimized A* in our application domain of pathfinding on a grid.

1 Introduction

Pathfinding is a core component of many intelligent agent applications, ranging in diversity from commercial computer games to robotics. The ability to have autonomous agents maneuver effectively across physical/virtual worlds is a key part of creating intelligent behavior. However, for many applications, especially those with tight real-time constraints, the computer cycles devoted to pathfinding can represent a large percentage of the CPU resources. Hence, more efficient ways for addressing this problem are needed.

Our application of interest is grid-based pathfinding. An agent has to traverse through a two-dimensional world. Moving from one location in the grid to another has a cost associated with it. The search objective is to travel from a $start$ location in the grid to a $goal$ location, and do so with the minimum cost. In many commercial computer games, for example, pathfinding is an essential requirement for computer agents (NPCs; non-player characters) [Stout, 1996]. For real-time strategy games, there may be hundreds of agents interacting at a time, each of which may have pathfinding needs. Grid-based pathfinding is also at the heart of many robotics applications, where the real-time nature of the applications requires expensive pathfinding planning and re-planning [Stentz, 1995].

A* [Hart et al., 1968] and IDA* [Korf, 1985] (and their variants) are the algorithms of choice for single-agent optimization search problems. A* does a best-first search; IDA* is depth-first. A* builds smaller search trees than IDA* because it benefits from using storage (the Open and Closed Lists), while IDA* uses storage which is only linear in the length of the shortest path length. A*'s best-first search does not come for free; it is expensive to maintain the Open List in sorted order. IDA*'s low storage solution also does not come for free; the algorithm ends up re-visiting nodes many times. On the other hand, IDA* is simple to implement, whereas fast versions of A* require a lot of effort to implement.

IDA* pays a huge price for the lack of storage; for a search space that contains cycles or repeated states (such as a grid), depth-first search ends up exploring all distinct paths to that node. Is there any hope for the simplicity of a depth-first search for these application domains?

A* has three advantages over IDA* that allows it to build smaller search trees:

1. IDA* cannot detect repeated states, whereas A* benefits from the information contained in the Open and Closed Lists to avoid repeating search effort.

2. IDA*'s iterative deepening results in the search repeatedly visiting states as it reconstructs the *frontier* (leaf nodes) of the search. A*'s Open List maintains the search frontier.

3. IDA* uses a left-to-right traversal of the search frontier. A* maintains the frontier in sorted order, expanding nodes in a best-first manner.

The first problem has been addressed by a transposition table [Reinefeld and Marsland, 1994]. This fixed-size data structure, typically implemented as a hash table, is used to store search results. Before searching a node, the table can be queried to see if further search at this node is needed.

The second and third problems are addressed by introducing the *Fringe Search* algorithm as an alternative to IDA* and A*. IDA* uses depth-first search to construct the set of leaf nodes (the frontier) to be considered in each iteration. By keeping track of these nodes, the overhead of iterative deepening can be eliminated. Further, the set of frontier nodes do not have to be kept in sorted order. At one extreme, the Fringe Search algorithm expands frontier nodes in the exact same order as IDA* (keeping the frontier in a left-to-right order). At the other extreme, it can be made to expand them in the exact same order as A* (through sorting).

This paper makes the following contributions to our understanding of pathfinding search:

1. Fringe Search is introduced, a new algorithm that spans the space/time trade-off between A* and IDA*.

2. Experimental results evaluating A*, memory-enhanced IDA*, and Fringe Search. In our test domain, pathfinding in computer games, Fringe Search is shown to run significantly faster than a

highly optimized version of A*, even though it examines considerably more nodes.

3. Insights into experimental methodologies for comparing search algorithm performance. There is a huge gap in performance between "textbook" A* and optimized A*, and a poor A* implementation can lead to misleading experimental conclusions.

4. Fringe Search is generalized into a search framework that encompasses many of the important single-agent search algorithms.

Section 2 motivates and discusses the Fringe Search algorithm. Section 3 presents experimental results comparing A*, memory-enhanced IDA*, and Fringe Search. Section 4 illustrates the generality of the Fringe Search idea by showing how it can be modified to produce other well-known single-agent search algorithms. Section 5 presents future work and the conclusions.

2 The Fringe Search Algorithm

We use the standard single-agent notation: g is the cost of the search path from the $start$ node to the current node; h is the heuristic estimate of the path cost from the current node to the $goal$ node; $f = g + h$; and h^* is the real path cost to the $goal$.

Consider how IDA* works. There is a starting threshold ($h(root)$). The algorithm does a recursive left-to-right depth-first search, where the recursion stops when either a goal is found or a node is reached that has an f value bigger than the threshold. If the search with the given threshold does not find a goal node, then the threshold is increased and another search is performed (the algorithm iterates on the threshold).

IDA* has three sources of search inefficiency when compared to A*. Each of these is discussed in turn.

2.1 Repeated states

When pathfinding on a grid, where there are multiple paths (possibly non-optimal) to a node, IDA* flounders [Korf and Zhang, 2000]. The repeated states problem can be solved using a transposition table [Reinefeld and Marsland, 1994] as a cache of visited states. The table is usually implemented as a (large) hash table to minimize the cost of doing state look-ups. Each visit to a state results in a table look-up that may result in further search for the current sub-tree being proven unnecessary.

A transposition table entry can be used to store the minimal g value for this state and the backed-up f value obtained from searching this state. The g values can be used to eliminate provably non-optimal paths from the search. The f values can be used to show that additional search at the node for the current iteration threshold is unnecessary. In this paper, IDA* supplemented with a transposition table is called Memory-Enhanced IDA* (ME-IDA*).

2.2 Iterating

Each IDA* iteration repeats *all* the work of the previous iteration. This is necessary because IDA* uses essentially no storage.

Consider Figure 1: each branch is labeled with a path cost (1 or 2) and the heuristic function h is the number of moves required to reach the bottom of the tree (each move has an admissible cost of 1). The left column illustrates how IDA* works. IDA* starts out with a threshold of $h(start) = 4$. Two nodes are expanded (black circles) and two nodes are visited (gray circles) before the algorithm proves that no solution is possible with a cost of 4. An expanded node has its children generated. A visited node is one where no search is performed because the f value exceeds the threshold. The f threshold is increased to 5, and the search starts over. Each iteration builds a depth-first search, starting at $start$, recursing until the threshold is exceeded or $goal$ is found. As the figure illustrates, all the work of the previous iteration i must be repeated to reconstruct the $frontier$ of the search tree where iteration $i + 1$ can begin to explore new nodes. For domains with a small branching factor, the cost of the iterating can dominate the overall execution time. In this example, a total of 17 nodes have to be expanded ($start$ gets expanded 3 times!) and 27 nodes are visited.

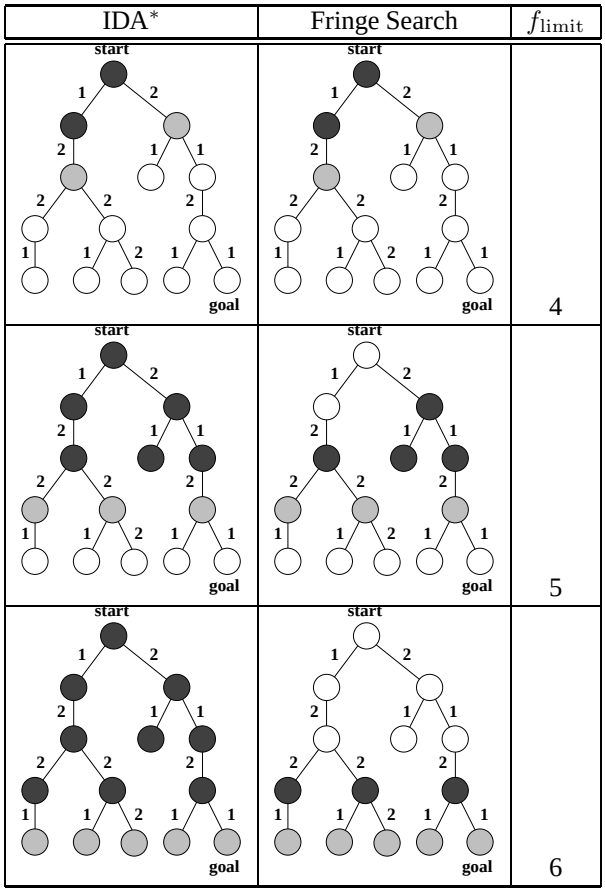


Figure 1: Comparison of IDA* and Fringe Search on an example graph. Visited nodes (gray) and expanded nodes (black) are given for each iteration

There exist IDA* variants, such as IDA*_CR [Sarkar et al., 1991], that can reduce the number of iterations. However, IDA* could be further improved if the repeated states overhead of iterating could be eliminated all together. This can be done by saving the leaf nodes (the $frontier$) of the iteration i search tree and use it as the starting basis for iteration $i + 1$.

The middle column of Figure 1 illustrates how this algorithm works. It starts with $start$ and expands it exactly as in IDA*. The two leaf nodes of the first iteration are saved, and are then used as the starting point for the second iteration. The second iteration has 3 leaf nodes that are used for the third iteration. For the last iteration, IDA* has to visit every node in the tree; the new algorithm only visits the parts that have not yet been explored. In this example, a total of 9 nodes are expanded and 19 nodes are visited. Given that expanded nodes are considerably more expensive than visited nodes, this represents a substantial improvement over IDA*.

This new algorithm is called the Fringe Search, since the algorithm iterates over the fringe (frontier) of the search tree[1]. The data structure used by Fringe Search can be thought of as two lists: one for the current iteration (now) and one for the next iteration ($later$). Initially the now list starts off with the root node and the $later$ list is empty. The algorithm repeatedly does the following. The node at the head of the now list ($head$) is examined and one of the following actions is taken:

1. If $f(head)$ is greater than the threshold then $head$ is removed from now and placed at the end of $later$. In other words, we do not need to consider $head$ on this iteration (we only visited $head$), so we save it for consideration in the next iteration.

2. If $f(head)$ is less or equal than the threshold then we need to consider $head$'s children (expand $head$). Add the children of $head$ to the front of now. Node $head$ is discarded.

When an iteration completes and a goal has not been found, then the search threshold is increased, the $later$ linked list becomes the now list, and $later$ is set to empty.

When a node is expanded, the children can be added to the now list in any order. However, if they are inserted at the front of the list and the left-to-right ordering of the children is preserved (the left-most child ends up at the front of the list), then the Fringe Search expands nodes *in the exact same order as IDA**. The children can be added in different ways, giving rise to different algorithms (see the following section).

2.3 Ordering

IDA* uses a left-to-right traversal of the search frontier. A* maintains the frontier in sorted order, expanding nodes in a best-first manner.

[1]Note that Frontier Search would be a better name, but that name has already been used [Korf and Zhang, 2000].

The algorithm given above does no sorting — a node is either in the current iteration (now) or the next ($later$). The Fringe Search can be modified to do sorting by having multiple $later$ buckets. In the extreme, with a bucket for every possible f value, the Fringe will result in the same node expansion order as A*. The Fringe Search algorithm does not require that its now list be ordered. At one extreme one gets the IDA* left-to-right ordering (no sorting) and at the other extreme one gets A*'s best-first ordering (sorting). In between, one could use a few buckets and get partial ordering that would get most of the best-first search benefits but without the expensive overhead of A* sorting.

2.4 Discussion

The Fringe Search algorithm essentially maintains an Open List (the concatenation of now and $later$ in the previous discussion). This list does not have to be kept in sorted order, a big win when compared to A* where maintaining the Open List in sorted order can dominate the execution cost of the search. The price that the Fringe Search pays is that it has to re-visit nodes. Each iteration requires traversing the entire now list. This list may contain nodes whose f values are such that they do not have to be considered until much later (higher thresholds). A* solves this by placing nodes with bad f values at/near the end of the Open List. Thus, the major performance differences in the two algorithms can be reduced to three factors:

1. Fringe Search may visit nodes that are irrelevant for the current iteration (the cost for each such node is a small constant),

2. A* must insert nodes into a sorted Open List (the cost of the insertion can be logarithmic in the length of the list), and

3. A*'s ordering means that it is more likely to find a goal state sooner than the Fringe Search.

The relative costs of these differences dictates which algorithm will have the better performance.

2.5 Implementation

The pseudo-code for the Fringe Search algorithm is shown in Figure 2. Several enhancements have been done to make the algorithm run as fast as possible (also done for A* and ME-IDA*). The now and $later$ lists are implemented as a single double-linked list, where nodes in the list before the current node under consideration are the $later$ list, and the rest is the now list. An array of pre-allocated list nodes for each node in the grid is used, allowing constant access time to nodes that are known to be in the list. An additional marker array is used for constant time look-up to determine whether some node is in the list. The g value and iteration number (for ME-IDA*) cache is implemented as a perfect hash table. An additional marker array is used for constant time look-up to determine whether a node has already been visited and for checking whether an entry in the cache is

```
Initialize:
    Fringe F ← (s)
    Cache C[start] ← (0, null),
    C[n] ← null for n ≠ start
    f_limit ← h(start)
    found ← false
Repeat until found = true or F empty
    f_min ← ∞
    Iterate over nodes n ∈ F from left to right:
        (g, parent) ← C[n]
        f ← g + h(n)
        If f > f_limit
            f_min ← min(f, f_min)
            continue
        If n = goal
            found ← true
            break
        Iterate over s ∈ successors(n) from right to left:
            g_s ← g + cost(n, s)
            If C[s] ≠ null
                (g', parent) ← C[s]
                If g_s ≥ g'
                    continue
            If F contains s
                Remove s from F
            Insert s into F after n
            C[s] ← (g_s, n)
        Remove n from F
    f_limit ← f_min
If found = true
    Construct path from parent nodes in cache
```

Figure 2: Pseudo-code for Fringe Search

valid. As for ME-IDA* (see below), the marker arrays are implemented with a constant time clear operation.

If the successors of a node n are considered from right-to-left, then they get added to the Fringe list F such that the left-most one ends up immediately after n. This will give the same order of node expansions as IDA*.

3 Experiments

In this section we provide an empirical evaluation of the Fringe Search algorithm, as well as comparing its performance against that of both Memory-Enhanced IDA* (ME-IDA*) and A*. The test domain is pathfinding in computer-game worlds.

3.1 Algorithm Implementation Details

For a fair running-time comparison we need to use the "best" implementation of each algorithm. Consequently, we invested a considerable effort into optimizing the algorithms the best we could, in particular, by use of efficient data structures. For example, the state spaces for game and robotics grids are generally small enough to comfortably fit into the computer's main memory. Our implementations take advantage of this by using a lookup table that provides a constant-time access to all state information. Additional algorithm-dependent implementation/optimization details are listed below. It is worth mentioning that the most natural data structures for implementing both Fringe Search and ME-IDA* are inherently simple and efficient, whereas optimizing A* for maximum efficiency is a far more involved task.

3.1.1 A* Implementation.

The Open List in A^* is implemented as a balanced binary tree sorted on f values, with tie-breaking in favor of higher g values. This tie-breaking mechanism results in the $goal$ state being found on average earlier in the last f-value pass. In addition to the standard Open/Closed Lists, marker arrays are used for answering (in constant time) whether a state is in the Open or Closed List. We use a "lazy-clearing" scheme to avoid having to clear the marker arrays at the beginning of each search. Each pathfinding search is assigned a unique (increasing) *id* that is then used to label array entries relevant for the current search. The above optimizations provide an order of magnitude performance improvement over a standard "textbook" A^* implementation.

3.1.2 ME-IDA* Implementation.

Memory-Enhanced IDA* uses a transposition table that is large enough to store information about all states. The table keeps track of the length of the shortest path found so far to each state (g value) and the backed-up f-value. There are three advantages to storing this information. First, a node is re-expanded only if entered via a shorter path ($g(s) < g_{cached}(s)$). Second, by storing the minimum backed-up f-value in the table (that is, the minimum f-value of all leaves in the sub-tree), the algorithm can detect earlier when following a non-optimal path (e.g., paths that lead to a dead-end). Combining the two caching strategies can drastically reduce the number of nodes expanded/visited in each iteration.

There is also a third benefit that, to the best of our knowledge, has not been reported in the literature before. When using non-uniform edge costs it is possible that ME-IDA* reduces the number of iterations by backing up a larger f-limit bound for the next iteration. We show an example of this in Figure 3. The current iteration is using a limit of 10. In the tree to the left transpositions are not detected (A and A' are the same node). Nodes with a f-value of 10 and less are expanded. The minimum f-value of the children is then backed up as the limit for the next iteration, in this case $min(12, 14)$ or 12. In the tree to the right, however, we detect that A' is a transposition into A via an inferior path, and we can thus safely ignore it (back up a f-value of $\inf$ to B). The f-limit that propagates up for use in the next iteration will now be 14.

3.2 Testing Environment

We extracted 120 game-world maps from the popular fantasy role-playing game *Baldur's Gate II* by *Bioware Inc.*

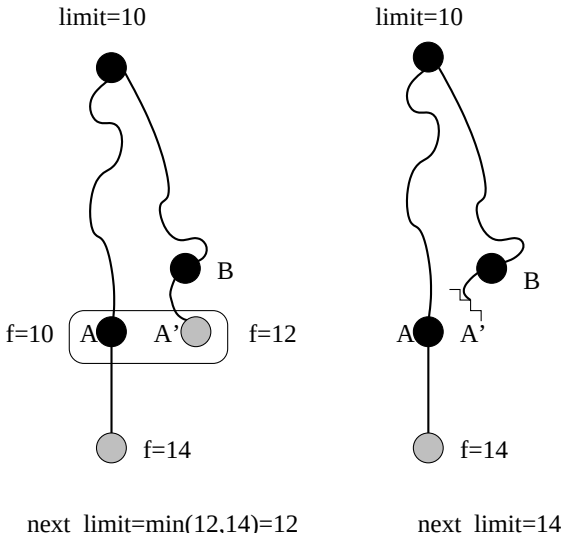


Figure 3: Caching reducing the number of iterations

Rectangular grids were superimposed over the maps to form discrete state spaces ranging in size from 50×50 to 320×320 cells, depending on the size of the game worlds (with an average of approximately 110×110 cells). A typical map is shown in Figure 4.

For the experiments we use two different grid-movement models: *tiles*, where the agent movement is restricted to the four orthogonal directions ($move\ cost = 100$), and *octiles*, where the agent can additionally move diagonally ($move\ cost = 150$). To better simulate game worlds that use variable terrain costs we also experiment with two different obstacle models: one where obstacles are impassable, and the other where they can be traversed, although at threefold the usual cost. As a heuristic function we used the minimum distance as if traveling on an obstacle-free map (e.g. Manhattan-distance for tiles). The heuristic is both admissible and consistent.

On each of the 120 maps we did 400 independent pathfinding searches between randomly generated start and goal locations, resulting in a total of 48,000 data points for each algorithm/model. We ran the experiments on a 1GHz Pentium III computer (a typical minimum required hardware-platform for newly released computer games), using a recently developed framework for testing pathfinding algorithms [Björnsson et al., 2003].

3.3 Fringe Search vs. ME-IDA*

The main motivation for the Fringe Search algorithm was to eliminate IDA*'s overhead of re-expanding the internal nodes in each and every iteration. Figure 5 shows the number of nodes expanded and visited by Fringe Search relative to that of ME-IDA* (note that the IDA* results are not shown; most runs did not complete). The graphs are plotted against the initial heuristic estimate error, that is, the difference between the actual and the estimated cost from *start* to *goal* ($h^*(start) - h(start)$). In general, the error increases as the game maps get more sophisticated (larger and/or more obstacles). We can see that as the heuristic

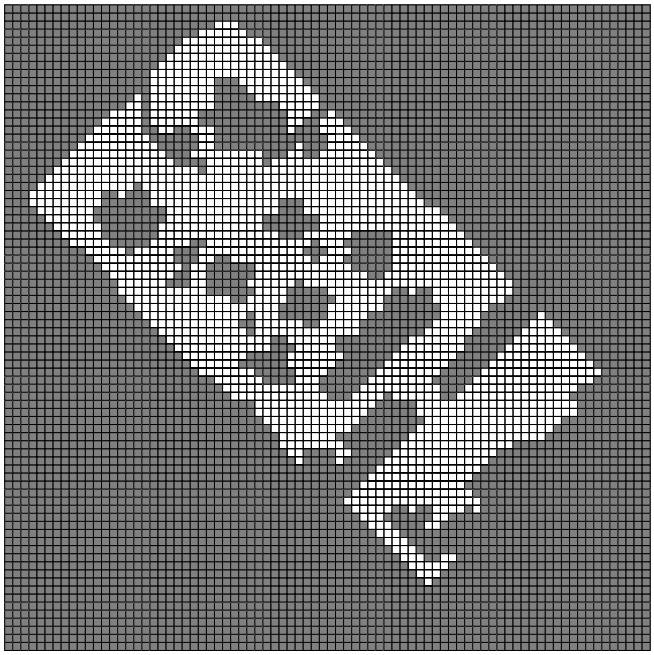


Figure 4: Example map

error increases, the better the Fringe Search algorithm performs relative to ME-IDA*. This can be explained by the observation that as the error increases, so will the number of iterations that *IDA** does. The data presented in Tables 1 and 2 allows us to compare the performance of the algorithms under the different pathfinding models we tested. They are based on pathfinding data on maps with impassable and passable obstacles, respectively. The tables give the CPU time (in milliseconds), iterations (number of times that the search threshold changed), visited (number of nodes visited), visited-last (number of nodes visited on the last iteration), expanded (number of nodes expanded), expanded-last (number of nodes expanded on the last iteration), path cost (the cost of the optimal path found), and path length (the number of nodes along the optimal path).

The data shows that Fringe Search is substantially faster than ME-IDA* under all models (by roughly an order of magnitude). The savings come from the huge reduction in visited and expanded nodes.

3.4 Fringe Search vs. A*

The A* algorithm is the *de facto* standard used for pathfinding search. We compared the performance of our new algorithm with that of a well-optimized version of A^*. As we can seen from Tables 1 and 2, both algorithms expand comparable number of nodes (the only difference is that because of its g-value ordering A^* finds the target a little earlier in the last iteration). Fringe Search, on the other hand, visits many more nodes than A*. Visiting a node in Fringe Search is an inexpensive operation, because the algorithm iterates over the node-list in a linear fashion. In contrast A* requires far more overhead per node because of the extra work needed to maintain a sorted order. Time-wise the Fringe Search algorithm outperforms A* by a significant

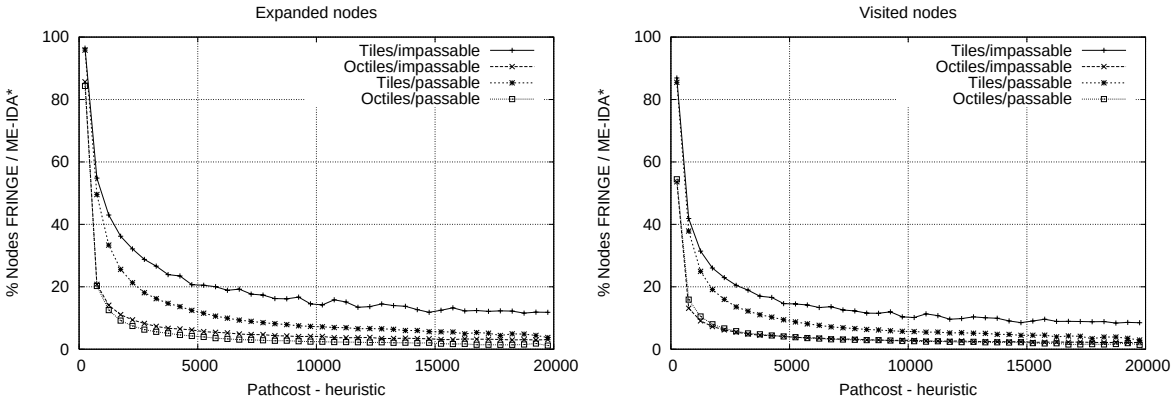


Figure 5: Comparison of nodes expanded/visited by Fringe Search vs. ME-IDA*

Table 1: Pathfinding statistics for impassable obstacles

	Octiles			Tiles		
	A*	Fringe	ME-IDA*	A*	Fringe	ME-IDA*
CPU/msec	1.7	1.3	33.7	1.2	0.8	5.6
Iterations	25.8	25.8	26.9	9.2	9.2	9.2
N-visited	583.4	2490.7	91702.9	607.0	1155.3	11551.3
N-visited-last	27.7	79.5	620.1	54.5	103.8	276.9
N-expanded	582.4	586.5	14139.1	606.0	613.2	4327.6
N-expanded-last	26.7	30.7	115.9	53.5	60.7	127.8
P-Cost	5637.7	5637.7	5637.7	6758.6	6758.6	6758.6
P-Length	46.1	46.1	46.1	68.6	68.6	68.6

margin, running 25%-40% faster on average depending on the model.

Note that under the passable obstacle model, there is a small difference in the path lengths found by A* and Fringe/ME-IDA*. This is not a concern as long as the costs are the same (a length of a path is the number of grid cells on the path, but because under this model the cells can have different costs it is possible that two or more different length paths are both optimal cost-wise).

Our implementation of Fringe Search is using the IDA* order of expanding nodes. Using buckets, Fringe Search could do partial or even full sorting, reducing or eliminating A*'s best-first search advantage. The expanded-last row in Tables 1 and 2 shows that on the last iteration, Fringe Search expands more nodes (as expected). However, the difference is small, meaning that for this application domain, the advantages of best-first search are insignificant.

The ratio of nodes visited by Fringe Search versus A* is different for each model used. For example, in the impassable and passable obstacles model these ratios are approximately 4 and 6, respectively. It is of interest to note that a higher ratio does not necessarily translate into worse relative performance for Fringe Search; in both cases the relative performance gain is the same, or approximately 25%. The reason is that there is a "hidden" cost in A* not reflected in the above statistics, namely as the Open List gets larger so will the cost of maintaining it in a sorted order.

4 Related Algorithms

The Fringe Search algorithm can be seen either as a variant of A* or as a variant of IDA*.

Regarded as a variant of A*, the key idea in Fringe Search is that the Open List does not need to be fully sorted. The essential property that guarantees optimal solutions are found is that a state with an f-value exceeding the largest f-value expanded so far must not be expanded unless there is no state in the Open List with a smaller f-value. This observation was made in [Bagchi and Mahanti, 1983], which introduced a family of A*-like algorithms based on this principle, maintaining a record of the largest f-value of nodes expanded to date. This value, which we will call *Bound*, is exactly analogous to the cost bound used in iterative-deepening (f-limit); it plays the same role as and is updated in an identical manner.

Analogous to the FOCAL technique (pp. 88-89, [Pearl, 1984]) it is useful to distinguish the nodes in the Open List that have a value less than or equal to *Bound* from those that do not. The former are candidates for selection to be expanded in the present iteration, the latter are not.

This is a family of algorithms, not an individual algorithm, because it does not specify how to choose among the candidates for expansion, and different A*-like algorithms can be created by changing the selection criterion. For example, A* selects the candidate with the minimum f-value. By contrast algorithm C [Bagchi and Mahanti, 1983] selects the candidate with the minimum g-value.

Table 2: Pathfinding statistics for passable obstacles

	Octiles			Tiles		
	A*	Fringe	ME-IDA*	A*	Fringe	ME-IDA*
CPU/msec	2.5	1.9	52.2	1.9	1.1	11.6
Iterations	14.2	14.2	14.2	5.0	5.0	5.0
N-visited	741.7	4728.3	132831.0	800.5	1828.3	23796.2
N-visited-last	27.0	119.7	1464.2	54.8	143.2	817.0
N-expanded	740.7	748.4	18990.1	799.5	816.3	7948.1
N-expanded-last	26.0	33.7	226.1	53.8	70.6	290.1
P-Cost	5000.6	5000.6	5000.6	5920.1	5920.1	5920.1
P-Length	39.5	39.5	39.5	56.3	56.5	56.5

If the heuristic being used is admissible, but not consistent, A* can do an exponential number of node expansions in the worst case, even if ties are broken as favorably as possible [Martelli, 1977]. By contrast, C can do at most a quadratic number of expansions, provided that ties in its selection criterion are broken in favor of the goal (but otherwise any tie-breaking rule will do)[2]. If the heuristic is not admissible, C maintains this worst-case speed advantage and in some circumstances finds superior solutions to A*.

Fringe Search is also a member of this family. It chooses the candidate that was most recently added. This gives it a depth-first behaviour mimicking IDA*'s order of node generation.

Among the variants of IDA*, Fringe Search is most closely related to ITS [Ghosh et al., 1994]. ITS is one of several "limited memory" search algorithms which aim to span the gap between A* and IDA* by using whatever memory is available to reduce the duplicated node generations that make IDA* inefficient. As [Ghosh et al., 1994] points out, ITS is the only limited memory algorithm which is not "best first". SMA* [Russell, 1992], which is typical of the others, chooses for expansion the "deepest, least-f-cost node". ITS, by contrast, is left-to-right depth first, just like IDA* and Fringe Search. New nodes are inserted into a data structure representing the search tree, and the node chosen for expansion is the deepest, left-most node whose f-value does not exceed the current cost bound. ITS requires the whole tree structure in order to retract nodes if it runs out of memory. Because Fringe Search assumes enough memory is available, it does not need the tree data structure to support its search, it needs only a linked list containing the leaf (frontier) nodes and a compact representation of the closed nodes.

5 Conclusions

Large memories are ubiquitous, and the amount of memory available will only increase. The class of single-agent search applications that need fast memory-resident solutions will only increase. As this paper shows, in this case, A* and IDA* are not the best choices for some applications. For example, Fringe Search out-performs optimized versions of A* and ME-IDA* by significant margins when pathfinding on grids typical of computer-game worlds. In comparison to ME-IDA*, the benefits come from avoiding repeatedly expanding interior nodes; compared to A*, Fringe Search avoids the overhead of maintaining a sorted open list. Although visiting more nodes than A* does, the low overhead per node visit in the Fringe Search algorithm results in an overall improved running time.

Since we ran the above experiments we have spent substantial time optimizing our already highly-optimized A* implementation even further. Despite of all that effort A* is still not competitive to our initial Fringe implementation, although the gap has closed somewhat (the speedup is ca. 10% for octiles and 20% for tiles). This is a testimony of one of Fringe search greatest strengths, its simplicity.

As for future work, one can possibly do better than Fringe Search. Although the algorithm is asymptotically optimal with respect to the size of the tree explored (since it can mimic A*), as this paper shows there is much to be gained by a well-crafted implementation. Although our implementation strove to be cache conscious, there still may be performance gains to be had with more cache-friendly data structures (e.g., [Niewiadomski et al., 2003]). Also, our current implementation of the Fringe Search algorithm traverses the fringe in exactly the same left-to-right order as IDA* does. Fringe Search could be modified to traverse the fringe in a different order, for example, by using buckets to partially sort the fringe. Although our experimental data suggests that this particular application domain is unlikely to benefit much from such an enhancement, other domains might.

Acknowledgments

This research was supported by grants from the Natural Sciences and Engineering Council of Canada (NSERC) and Alberta's Informatics Center of Research Excellence (iCORE). This work benefited from the discussions of the University of Alberta Pathfinding Research Group, including Adi Botea and Peter Yap. The research was inspired by the needs of our industrial partners Electronic Arts (Canada) and Bioware, Inc.

[2] "Exponential" and "quadratic" are in terms of the parameter N defined in [Bagchi and Mahanti, 1983].

Bibliography

[Bagchi and Mahanti, 1983] Bagchi, A. and Mahanti, A. (1983). Search algorithms under different kinds of heuristics – a comparative study. *Journal of the Association of Computing Machinery*, 30(1):1–21.

[Björnsson et al., 2003] Björnsson, Y., Enzenberger, M., Holte, R., Schaeffer, J., and Yap., P. (2003). Comparison of different abstractions for pathfinding on maps. In *International Joint Conference on Artificial Intelligence (IJCAI'03)*, pages 1511–1512.

[Ghosh et al., 1994] Ghosh, S., Mahanti, A., and Nau, D. S. (1994). ITS: An efficient limited-memory heuristic tree search algorithm. In *National Conference on Artificial Intelligence (AAAI'94)*, pages 1353–1358.

[Hart et al., 1968] Hart, P., Nilsson, N., and Raphael, B. (1968). A formal basis for the heuristic determination of minimum cost paths. *IEEE Trans. Syst. Sci. Cybernet.*, 4(2):100–107.

[Korf, 1985] Korf, R. (1985). Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence*, 27:97–109.

[Korf and Zhang, 2000] Korf, R. and Zhang, W. (2000). Divide-and-conquer frontier search applied to optimal sequence alignment. In *National Conference on Artificial Intelligence (AAAI'02)*, pages 910–916.

[Martelli, 1977] Martelli, A. (1977). On the complexity of admissible search algorithms. *Artificial Intelligence*, 8:1–13.

[Niewiadomski et al., 2003] Niewiadomski, R., Amaral, N., and Holte, R. (2003). Crafting data structures: A study of reference locality in refinement-based path finding. In *International Conference on High Performance Computing (HiPC)*, number 2913 in Lecture Notes in Computer Science, pages 438–448. Springer.

[Pearl, 1984] Pearl, J. (1984). *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison & Wesley.

[Reinefeld and Marsland, 1994] Reinefeld, A. and Marsland, T. (1994). Enhanced iterative-deepening search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 16:701–710.

[Russell, 1992] Russell, S. J. (1992). Efficient memory-bounded search methods. In *European Conference on Artificial Intelligence (ECAI'92)*, pages 1–5, Vienna, Austria. Wiley.

[Sarkar et al., 1991] Sarkar, U., Chakrabarti, P., Ghose, S., and Sarkar, S. D. (1991). Reducing reexpansions in iterative-deepening search by controlling cutoff bounds. *Artificial Intelligence*, 50:207–221.

[Stentz, 1995] Stentz, A. (1995). The focussed D* algorithm for real-time replanning. In *International Joint Conference on Artificial Intelligence (IJCAI'95)*, pages 1652–1659.

[Stout, 1996] Stout, B. (1996). Smart moves: Intelligent path-finding. *Game Developer Magazine*, (October):28–35.

Adapting Reinforcement Learning for Computer Games: Using Group Utility Functions

Jay Bradley **Gillian Hayes**

Institute of Perception, Action and Behaviour
School of Informatics, University of Edinburgh
James Clerk Maxwell Building, King's Buildings
Edinburgh, EH 9 3JZ, United Kingdom

jay.bradley@ed.ac.uk gmh@inf.ed.ac.uk

Abstract-

Group utility **functions are an extension of the common** *team utility* **function for providing multiple agents with a common reinforcement learning signal for learning cooperative behaviour. In this paper we describe what group utility functions are and suggest using them to provide non-player computer game character behaviours. As yet, reinforcement learning techniques have very rarely been used for computer game character specification.**

Here we show the results of using a group utility function to learn an equilibrium between two computer game characters and compare this against the performance of the two agents learning independently. We also explain how group utility functions could be applied to learn equilibria between groups of agents.

We highlight some implementation issues arising from using a commercial computer game engine for multi-agent reinforcement learning experiments.

1 Introduction

Multi-agent reinforcement learning has not often been used for computer game non-player character development. Why is this? There are several possible reasons including the perceived (and often real) complexity involved with learning to use any new technology, and the inflexibility and lack of control over the final behaviours learned. One major problem is the wide ranging complaint that "academic" artificial intelligence techniques are only concerned with optimality of behaviour and ignore problems where the solution is a mixture of aesthetics and performance. Here we introduce methods for using reinforcement learning to allow agents to learn to play computer games not just optimally but to the ability level required and in a manner aligned with the game structure.

Behaviours for non-player game characters are usually specified using rule based systems [29]. Commonly, finite state machines are used with fuzzy logic. Even with clever modularisation and combination of behaviour specification rules it takes a lot of human effort to design each character. Hence, behaviour specification is often copied across several characters and this is often very obvious to human game players. Despite this, hand coded rules are currently the chosen way for game producers to specify behaviour for non-player characters. With the recent popularisation of massively multiplayer games the number of non-player game characters required for a typical game is increasing. Thus the proliferation of non-player characters showing remarkably similar behaviours is set to become more of a problem. Reinforcement learning is another possible way to specify behaviours for multiple characters. Reinforcement learning is more reusable than hand coding finite state machines. That is, the reward function and state representation can be reused for training many characters. The characters behaviour will depend on their abilities and their experiences whilst learning. Thus we can get more unique behaviours for less human design time. The reinforcement learning somewhat automates the behaviour production process. Of course, human time still needs to be spent designing the characters' state representations, the reward function(s) and finding appropriate learning parameters.

So reinforcement learning in general and multi-agent reinforcement learning in particular, should be useful tools for computer game producers but so far only single agent reinforcement learning has featured in any commercial games and then, only in a few commercial games. The only game of real note is Black & White [15] in which one non-player character exhibited a small amount of learning [6]. This was described as a "gamble" by the game's designer, Peter Molyneux [17]. We believe that by introducing multi-agent reinforcement learning methods suitable for and applicable to computer games, we can provide technology that could be used for a massively multiplayer computer game in the foreseeable future.

The main inspiration behind this work is the group nature of computer game character societies. Characters are usually largely defined by which group they belong to. For example are they a goody or a baddy? What type of baddy are they? What species are they? There are usually groups of characters and also groups

within groups. Which group or groups an agent belongs to usually determines its behaviour to a large extent. The relationship between these groups provides the structure of many games, obvious examples being good and evil groups, sports teams and groups within sports teams (such as defence). For this reason, we have incorporated a group structure into standard multi-agent reinforcement learning methods. The only existing simple group learning mechanism for multi-agent reinforcement learning is the well known team utility function (see [27, 26], for discussion and criticism).

A team utility function rewards each agent in a group with the sum of all the group's agents' rewards (we use the term *modified reward* to refer to the reward that an agent receives after any processing). In this way a notion of teamwork is achieved with agents hopefully learning that what is good for the team is good for them. There exist a number of unexplored group utility functions other than the team utility function, that provide a modified reward to individual agents that is derived from the agents' performances. The way in which the individual agents' rewards are combined determines the relationship they have with each other. Team utility sums all the agents' rewards which implies that each agent is on the same team (hence the name) with the same goals, whereas other utility functions introduced here use different reward combining functions to specify different relationships between agents. For example later we introduce a combining function (note: for team utility the combining function is summation) that gives the negative standard deviation of the agents' rewards. By using this combining function we can make agents in a group learn to perform approximately only as well as each other. This could be useful in a computer game if we need several agents to compete and draw at some task. For example, we could use this combining function to provide non-player driving characters in a race game if we wanted the result to be close.

For team utility we have a flat structure. That is, all the agents rewards are summed by one team utility function and given back to them as modified reward. Using group utility functions we can create a hierarchy of groups, each with their own utility function. Later we show that it is possible to have groups within groups in a hierarchical structure (Note that this is entirely unrelated to hierarchical reinforcement learning methods). For example, using one balance inducing combining function (i.e. the negative standard deviation) with two team utility functions "beneath" it, it is possible to train two teams of agents to perform equally as well. (Also, there is no reason why agents cannot belong to more than one group at a time. One can imagine a crooked character, or group of characters, whose allegiances can change. This is left for future work.)

We believe that this novel use of combining functions to produce different group utility functions allows us to have more control over agents' learnt behaviours. The system is shown abstractly in Figure 1. In Figure 1, agents 1 and 2 have the same group utility function which has a combining function encouraging collaboration or team work. Agent 3 is in its own single member group. Both the collaborative group and agent 3 "belong to" a group utility function that encourages equal or balanced performance among its children groups. So, with this set up we should be able to make the three agents learn to perform as two teams that will perform equally as well as each other. In essence we are gaining more control over multi-agent reinforcement learning by modifying the reinforcement signal of each agent based on the group(s) it belongs to. We believe this to be novel research that is a natural extension of the team utility function that, for some reason, has not been investigated before.

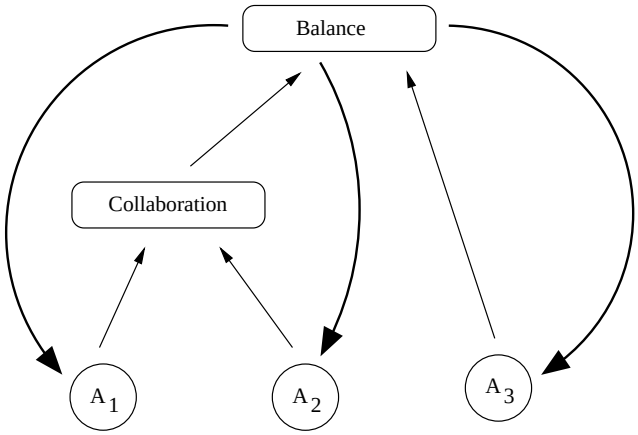


Figure 1: The agents individually collected reward being sent up through the group utility functions is shown with thin arrows. The modified reward is shown as thick lines coming from the topmost group utility function. With this setup we can make the three agents learn to perform as two teams that will perform equally as well as each other.

2 Background

Game theory allows us to describe computer games as stochastic extensive-form games with (semi-)situated agents. One of the goals of our research is to present reinforcement learning in a simple way. Thus we do not follow fully the strictest definitions from game theory. Instead preferring to use a more rudimentary approach [1, 3, 24]. In detail, a stochastic multi-agent game is represented by a tuple: $(n, S, A_{1...n}, T, R_{1...n})$, where:

- n is the number of agents (agents and characters are interchangeable terminology here),
- S is a set of states.

- A_i is the set of actions available to agent i (and A is the joint action space $A_1 \quad ... \quad A_n$). In this work we use probabilistic action-selection unless purposefully selecting an exploratory action. That is, the actions with the largest utility will most likely be executed, and similarly the action with the second largest utility will be the second most likely action to be executed, and so on. Our reasons for using probabilistic action selection is given later in Section 5.

- T is the transition function $S \quad A \quad S \rightarrow [0, 1]$, that is, the state will change according to the joint actions of all the agents. Extensive-form means that agents do not act simultaneously. A lot of reinforcement learning research simplifies agent movement by synchronising agents' time steps. This cannot be done in computer games as the resulting multi-character behaviour looks ridiculous. This is an aesthetic issue not normally addressed. Our solution to it, is shown in Section 5.

- and R_i is a reward function for the ith agent, $S \quad A \rightarrow \Re$. In our changes to this model shown later (Section 3) we do not actually change the reward function directly but insert one or more functions in between the reward function and the receiving agent(s). These functions are the combining functions associated with the group utility functions.

Above we say "semi-situated" agents, as although computer game characters can be thought of as embodied agents, acting in real-time in a well defined environment, there are several restrictions we do not have to deal with that affect truly situated agents (see [12] for a good explanation of what it means for an agent to be truly situated). There is negligible cost for communication between agents and centralised communication is easily possible. Also, whereas truly situated agents can have a strict time limit for learning, game agents can be run in simulated games for as long is necessary. Also, if we wanted to we could have perfect knowledge. However, we do not utilise or propose the use of perfect knowledge in this work for two reasons: the high processing requirements and to maintain perceptual honesty [10]. Firstly games, remember, are supposed to be implemented on home machines, not academic super computers. The processing power of home games machines, whilst increasing fast, is still limited [28]. Secondly, adversarial characters in computer games are often accused of "cheating" by human players because they clearly have use of more information than the human player does. By not gathering perfect knowledge we deny our agents one way of becoming cheats.

The group utility function introduced below is based on achieving balanced performance which can then be modified by weightings. What do we mean by "balanced performance?" As a definition for this work, we take a balanced performance to mean that each agent, or group of agents, will achieve the same, or similar, amount of reward over a period of time. For instance, if no agent makes any move for the length of an episode then the performance of each agent can be said to be balanced as each agent will presumably receive the same reward. However, this is a degenerate and useless case as no agent does anything at all! We will show more useful balancing later. A more precise definition of balance cames from game theory which gives us the Nash equilibrium [18] (and several associated best response points such as correlated-Q learning [8], Nash-Q [9] and minimax-Q [16]). Simply (and ignoring a large amount of game theory), a multi-agent system can be said to be in equilibrium when each agent can do no better given the actions of the other agents. Our work in this paper could be surmised as practically attaining an approximate equilibrium through simple reinforcement learning. Also of interest when considering balanced performance between agents is work on homo egualis societies [20]. The homo egualis approach works on the same problem as we address in this paper and is described as applying to, "multi-agent problems where local competition between agents arises, but where there is also the necessity of global cooperation." The homo egualis society method affects the action sets of agents so that if they are doing comparatively well then they are denied the use of certain actions until balance is attained. By affecting the action sets for periods of time, agents of a homo egualis society can be said to be using a *periodic policy*. In this paper agents select actions probabilistically and therefore learn a *mixed policy*.

Methods of achieving an equilibrium between groups of agents have before been classed as naive and sophisticated [24]. Naive methods usually ignore the fact that there are other agents in the system. Sophisticated methods often model the other agents explicitly. We are placing our approach at the naive end of the spectrum. We include other agents in an agent's state but model their actions in the world only through the modified (by group utility functions) reinforcement signal and the joint state space. This is why we refer to group utility functions as a *simple* multi-agent reinforcement learning method.

Game theory provides a theoretical framework for expressing multi-agent learning. However, often real-world problems are not easily described strictly in game theory. If we call truly game theoretic based algorithms that explicitly learn equilibria, equilibria learners, then we can call other approaches (and our approach) best response learners. That is, by learning a best response to the environment, including other agents in the environment and their disruption to the environment, then when multiple agents converge due to learning they do so to an equilibrium [1]. Best response learners can be thought of as a more casual form of equilibria learners

that are often easier to implement for complex, real-world, temporally dependent tasks, such as computer games. (It has been noted before that computer games are usually sufficiently complex enough to make them difficult to model in game theory [24]).

2.1 Multi-agent Reinforcement Learning in Computer Games

To the best of our knowledge, research on multi-agent reinforcement learning for computer games characters has not been presented before. As noted in the introduction, we believe only Black & White [15] has used reinforcement learning and only in a limited capacity for one character. Our development of a reinforcement learning system for computer games has stumbled many times because of unforeseen practicalities that are not well documented because of a lack of existing research material.

In a wider sense, research using computer games is now quite common (see [14, 13] for a call to arms and [7, 19] for an overview). There are several other researchers and departments pushing the use of computer games as research tools. Use of artificial intelligence techniques in commercial computer games is growing very quickly [21, 28, 4, 30, 11, 5], however the techniques are usually limited to finite state machines. However, the technology used in computer games is advancing quickly so this will not remain the case for long.

3 Group Utility Functions

It is easiest to think of group utility functions as an extension of the well known team utility function. It is implicit when using a team utility function that all agents share the same goal. However, this is not the case with computer games that have many groups of agents that, whilst sharing the overall goal of providing the game playing experience, have differing goals at differing levels of abstraction.

We are making the assumption throughout our work that a balanced game is a good game. That is, it is the aim of this part of our work to be able to create agents and groups of agents that can be made to compete equally well at tasks (even though the groups may differ in their actual capabilities or resources). After that we can place agents in more than one group and use group weightings to affect the behaviour of agents and groups of agents more radically). So we are skipping the troubling notion of what exactly makes a good computer game. We will leave this decision to the experts in game design. We believe that, given the ability to train agents and groups of agents to give a certain balanced performance and given that we can then affect this balance later by weighting the groups, this is sufficient technology for game designers wishing to use reinforcement learning to create behaviour specification. As a practical example, using group utilities we may hope to train two armies to fight each other so that the battle is very close and either army could win. Maybe this scenario is what is needed for some part of a game. We could use three group utilities, two with sum combining functions and one with a balance inducing combining function. All of the agents (soldiers) of one army would have their reward modified by one of the summative utility functions. The agents in the other army would "belong to" the other summative utility function. The two summative combining functions should be "placed under" the balancing group utility function. These group utilities modify an agents reward signal. How an agent's modified reward is calculated is explained below in Section 3.2.

To allow an element of balance, an agent's state representation must contain information indicating their performance, or their group's performance, relative to other agents or groups. An agent must know if they are currently performing too well or not well enough. It is not possible to learn an equilibrium without this information. Without being able to distinguish between winning or losing the agent is unable to make sense of the reward signal changes that occur when an imbalance occurs. Other equilibrium learners use relative performance indicators in the state representation also [2, 3, 20].

3.1 Combining Functions

The team utility function works by summing all the agents' rewards for the previous time step and then giving each agent the total group reward. We call the reward the agent gets after the summation the modified reward (notation, r'_i is the modified reward for agent i). We call the summation function a combining function. The summation function lets us express a teammate relationship. Section 3.2 describes how the agents rewards' are related to the their modified rewards. Here we consider a few combining functions:

- Individual. Each agent receives the reward it should receive as a result of its actions. If an agent somehow collects 10 reward units then that is the reward that is passed on to the agent. This just implements naive single agent learning. We include it here to show how group utility functions can implement this.
- Sum.
$$r'_i = \sum_{i=1}^{n} r_i$$
This implements team utility. As above it is included to show how group utility functions can achieve team work.
- Average. Similar to sum but not often considered. It is different from the sum combining function because the modified reward is scaled down. This can interact with the actual magnitude of rewards received by the agents.

$$r'_i = \frac{\sum_{i=1}^{n} r_i}{n}$$

- Negative standard deviation. If we take the standard deviation at one time step of all the agents' rewards then we have a measure of how well balanced the performance of the agents is at that time step. We take the negative result because then, the closer we come to no deviation, the higher the modified reward will be.

$$r'_i = \sqrt{\frac{\sum(r_i \quad avg(r_1...r_n))^2}{n}}$$

- Negative range. Again here we are looking at how well balanced the agents' performances are. We can further investigate here by looking at the inter-quartile range. This combining function has not been used in experimentation yet but is here to signify that there are several suitable functions to measure balance.

$$r'_i = \quad range + \sum_{i=1}^{n} r_i$$

$$= \quad (max(r_1...r_n) \quad min(r_1...r_n)) + \sum_{i=1}^{n} r_i$$

- There are any number of other possibly interesting functions to consider.

3.2 Calculating Agents' Modified Rewards

In the results below we only show the case with two agents under one group utility. We introduce a group affiliation parameter that scales the group utility. An agent's reward is the sum of its individually received reward and its group utility value scaled by the group affiliation value (if it only has one group utility, as it odes in our initial experiments presented below). The scaling is very important and has been difficult for us to set properly for the negative standard deviation group utility function. For our experiments (section 5), agents receive a reward of minus one for each action not leading to a reward or, zero when a reward is picked up. If the agents are in a group with the negative standard deviation combining function then the negative standard deviation needs to be scaled to interact appropriately with their typical rewards. So if two agents in a group with the negative standard deviation combining function have both failed to pick up any reward in the episode so far they will get a modified reward of 1 or, $1 + (\sqrt{\frac{\sum(r_i \quad avg(r_1...r_n))^2}{n}})$. If one of the agents picks up a reward we need to reflect that it was good to pick up a reward but bad to unbalance the agents' performance. The negative standard deviation is taken of the number of rewards picked up by each agent. After one reward is picked up the negative standard deviation will be 0.5. We discovered that a value of 0.5 is not enough to discourage unbalanced performance. Therefore we could set the group affiliation to be 2.5 or 3. This gives the group utility with a negative standard deviation combining function more presence in the agents modified reward.

We have found that for the negative standard deviation (and presumably for other combining functions) the interaction between the group affiliation value and (is the learning rate, for sarsa(λ), see below for an explanation of the learning algorithm) is critically important. We need the group affiliation to be set so that the negative punishment for becoming unbalanced does not overcome the positive reward for picking up a reward that makes the performance only slightly unbalanced. However, if the imbalance becomes large we do want the negative punishment to be larger in magnitude than any positive reward for picking up rewards. Also, we need the agents to be a little short sighted so that they will pick up reward without realising that they will eventually be punished for unbalancing the system. We do this by setting a low discount rate.

3.2.1 The Learning Algorithm

The learning algorithm is quite standard sarsa(λ), as taken from [23].

- $\forall$ agents
  - Place agent randomly
  - Set initial- , initial- , and λ
  - $\forall\ s \in S, a \in A, \quad Q(s,a) = 0$
- Place rewards randomly.

- $\forall$ agents
  - = initial- $\left[\frac{\frac{\text{number of episodes}}{2} \quad \text{episode}}{\frac{\text{number of episodes}}{2}}\right]$
  - = initial- $\left[\frac{\frac{\text{number of episodes}}{2} \quad \text{episode}}{\frac{\text{number of episodes}}{2}}\right]$
  - $r_i{}^t \quad R_i$ (Remove any rewards if picked up and replace with another reward)
  - $a_i{}^{t+1} \quad {}_i{}^t$ using -greedy selection (with the actions chosen probabilistically according to their utility values when not choosing to perform a random action)
  - $r'_i{}^t$ = agents own reward +(first group above's reward group affiliation) + (second group above's reward group affiliation^2) + ...
  - $\delta = r'_i{}^t + \quad Q(s_{t+1}, a_{t+1}) \quad Q(s_t, a_t)$
  - $e(s_t, a_t) = 1$ (Replacing trees)
  - $\forall\ e(s,a)$
    - $Q(s_t, a_t) = Q(s_t, a_t) + \quad \delta e(s_t, a_t)$
    - $e(s_t, a_t) = \quad \lambda e(s_t, a_t)$

4 Asynchronous Time-steps in a Multi-agent Reinforcement Learning System Using Group Utility Functions

Team utility assigns the same reward to each agents at each time step. It does this by summing each agents' privately collected reward for the current time step and giving the total to each agent as its modified reward. A subtle change is introduced if the agents' actions are asynchronous. The problem is that it is possible for an agent to complete more than one of its own time steps during only one time step of another agent. In this work for the purposes of group utility we will take an agent's last reported privately collected reward as its assumed last time step reward. This is shown in Figure 2. Existing literature presents team utility being used with synchronised actions or, whether the actions are synchronised is not made clear.

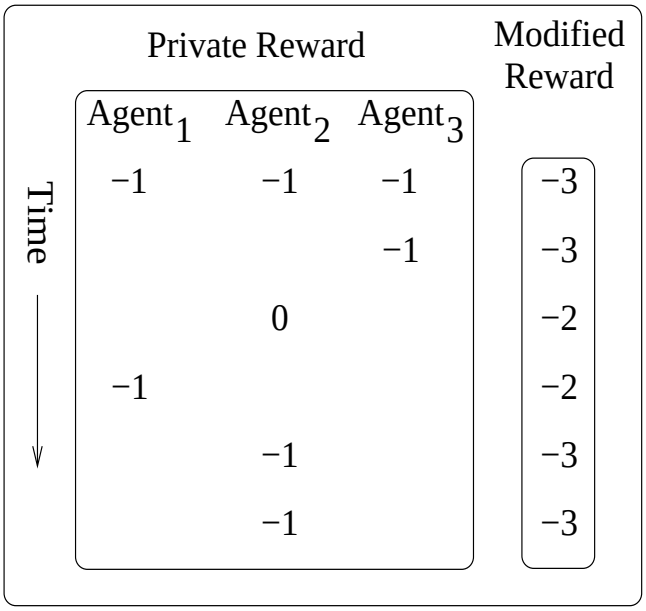


Figure 2: Shows the relationship between privately collected reward (what the agent actually collects in the environment) and modified reward (what the agent receives from its group utility functions) for the sum combining function (essentially the team utility. Remember that the team utility is the sum of all the agents' rewards). Here all agents start with an assumed reward value of -1 as in all our experiments.

5 Experiments

We have used a foraging task for all our experiments so far. The agents are placed randomly as are three rewards. When an agent picks up a reward another reward is placed randomly until thirty rewards have been picked up. An agent receives a reward of minus one for each action not leading to picking up a reward and zero if a reward is picked up (this is before modification by the group utility function). The agents have a total of five hundred action steps shared between them to complete the task. An episode ends when all thirty rewards have been picked up or five hundred action steps are reached. Each experiment ran for four thousand episodes. The Torque [25] game engine was used. It is our intention to use only computer game engines for our learning experiments. This brings up some running time issues for large experiments. We ran the Torque game engine as a dedicated server on a linux box. A four thousand episode experiment takes approximately one and a half hours with some wide variation. We were able to distribute many experiments over many machines to gather large number results of the same experiment for averaging. All results in this paper are averaged over one hundred runs.

The state representation contains the distance, direction and nearest agent to the two nearest rewards. There are four discrete distances and eight directions. The state also contains a variable stating whether the agent is winning, losing or drawing with other agents or whether the agent's group is winning, losing or drawing with the other groups. By organising the rewards by distance and taking only the nearest two we are learning a simpler task than we are actually achieving. This state representation is inspired by work on robot football [22] and we think this is the best method of presenting potentially complex and especially dynamic computer game worlds to agents in a concise way.

Actions available to the agents are: move towards a reward, move away from a reward, circle left relative to a reward and circle right relative to a reward. Computer games allow us to use these high level actions and make the agents actions appear much more realistic. Originally we used the traditional north, south, east and west actions in our experiment but aesthetically this is very poor as the game characters' movements looked very unrealistic. By using high level actions we can allow game designers to still control the behaviour of the characters to a certain degree.

Through somewhat ad hoc search, for our results shown below we used sarsa(λ) [23] with the following learning parameters: initial $= 0.5$, $= 0.6$, $\lambda = 0.9$ and group affiliation $= 3$. -greedy action selection was used with an initial of 0.6. and were degraded linearly from their initial values at the first episode to zero at half-way through the number of episodes (as shown in the learning algorithm - Section 3.2.1).

In may be clear to the reader why we need to choose actions probabilistically, but if not: If we choose actions deterministically then an agent will always pick the action with the highest Q-value. This method of action-selection does not have enough subtlety to differentiate between two actions that are very nearly as useful as each other in the same situation. Therefore it is not possible for an agent to learn that it should pick up reward but at the same time be wary of picking up too much in comparison to other agents. This requires that the action-selection method can choose actions probabilistically so that it takes into account

exactly how worthy an action is of being executed.

6 Results

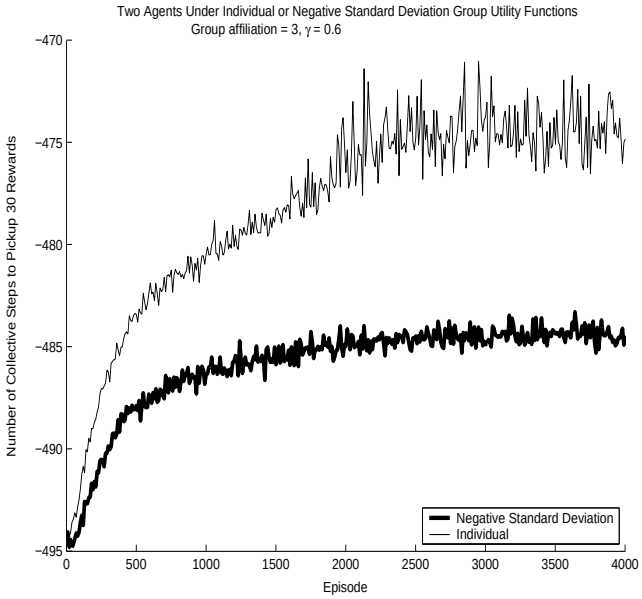


Figure 3: Compares the performance of two agents under an individual group utility function and under a negative standard deviation group utility function. The drop in final performance for balanced behaviours is approximately only 10 action steps extra.

6.1 Evaluation

As the reader can see from Figures 3 and 4, when the agents belong to the group utility with a negative standard deviation combining function, they pick up approximately the same number of rewards each episode. In can be said that the agents have an inequality aversion [20]. On the other hand when the two agents belong to the group utility with an individual combining function (i.e. their individually received rewards are left untouched), the amount of reward each agent picks up per episode has a high variance.

Group utility functions with the sum and average combining functions are not shown on the graph. They have almost exactly the same performance as the individual combining function.

7 Conclusions

We can see from Figures 3 and 4 that we are able to maintain a balance between the performance of two agents whilst learning a task and that the final policies maintain that balance. This is in comparison to two agents acting individually. The results show that what you can gain in balance you have to lose in performance. This makes intuitive sense. With randomly placed agents and rewards, a certain amount of extra travelling will be needed by the agents to maintain an equal number of collected rewards.

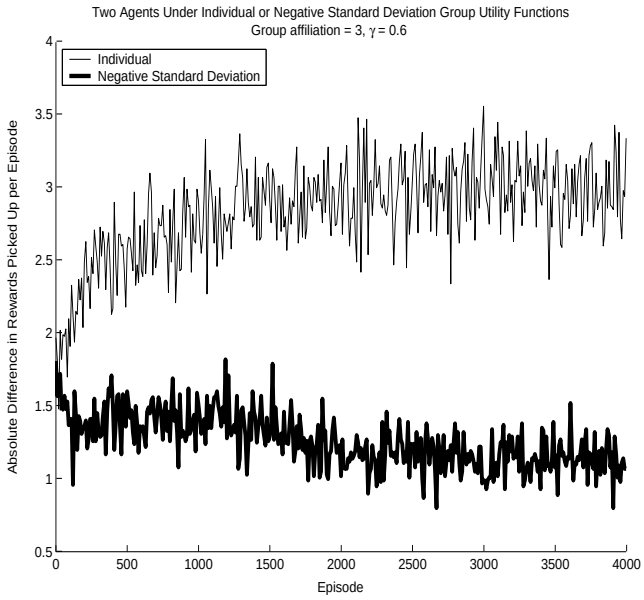


Figure 4: Shows that the negative standard deviation group utility function can keep the two agents close in performance. The lines show the absolute difference between the number of rewards each agent has picked up. Both lines start near zero because neither agent has learnt to pick up rewards yet. As the agents' ability to pick up rewards improves, the lines diverge.

Besides the main result of achieving a simple equilibrium through the use of a group utility function, we have also discovered much about how to implement reinforcement learning in a computer game. We have shown that an agent centered, feature based state representation and feature related actions can generalise learning experience in a dynamic environment. We have also shown that group utility functions can be used with asynchronous actions.

For future work we are immediately turning to achieving similar results but on a grander scale. Most equilibrium learners have been described working at the agent level but group utility functions should scale to learning equilibriums between groups as well as agents. Indeed, in more recent research we have used group utility functions to learn an equilibrium between a team of two agents and a single agent. We found that each agent needed an affiliation parameter for each group it belonged to. An individual group affiliation parameter allows us to easily scale the group utility values so that we can indicate to the agents, via their modified reward signal, the relative importance of the groups they belong to.

Bibliography

[1] Bowling, M., and Veloso, M. Existence of multiagent equilibria with limited agents. Technical report CMU-CS-02-104, Computer Science Department, Carnegie Mellon University, 2002.

[2] Bowling, M., and Veloso, M. Multiagent learning using a variable learning rate. *Artificial Intelligence* (2002).

[3] Bowling, M., and Veloso, M. Simultaneous adversarial multi-robot learning. In *Proceedings of IJCAI'03* (2003).

[4] Dybsand, E. Game developers conference 2001: An AI perspective. www.gamasutra.com/ features/20010423/dybsand_01.htm, 2001.

[5] Dybsand, E. AI roundtable moderator's report. www.gameai.com/cgdc04notes.dybsand.html, 2004.

[6] Evans, R., and Lamb, T. B. Gdc 2002: Social activities: Implementing wittgenstein. www.gamasutra.com/features/20020424/ evans_01.htm, 2002.

[7] Fairclough, C., Fagan, M., Namee, B. M., and Cunningham, P. Research directions for AI in computer games. In *Proceedings of the Twelfth Irish Conference on Artificial Intelligence and Cognitive Science* (2001).

[8] Greenwald, A., and Hall, K. Correlated q-learning. In *Proceedings of the Twentieth International Conference on Machine Learning* (2003).

[9] Hu, J., and Wellman, M. Nash q-learning for general-sum stochastic games. *Journal of Machine Learning Research*, 4 (2003).

[10] Isla, D., Burke, R. C., Downie, M., and Blumberg, B. A layered brain architecture for synthetic creatures. In *IJCAI* (2001), pp. 1051–1058.

[11] Kirby, N. AI roundtable moderator's report 2004. www.gameai.com/cgdc04notes.kirby.html.

[12] Konidaris, G. D., and Hayes, G. M. An architecture for behavior-based reinforcement learning. *To appear in the Journal of Adaptive Behavior* (2003).

[13] Laird, J. E. Research in human-level AI using computer games. Communications of the ACM, 2002.

[14] Laird, J. E., and van Lent, M. Human-level AI's killer application: Interactive computer games, 2000. ai.eecs.umich.edu/people/laird/papers/AAAI-00.pdf.

[15] Lionhead Studios. Black and White. blackandwhite.ea.com, 2001.

[16] Littman, M. L. Friend-or-foe q-learning in general-sum games. In *Proceedings of the Eighteenth International Conference on Machine Learning* (2001).

[17] Molyneux, P. Postmortem: Lionhead Studios' Black & White. www.gamasutra.com /features/20010613/molyneux_01.htm, 2001.

[18] Nash, J. F. Equilibrium points in *n-person games. PNAS, 1950.*

[19] Niederberger, C., and Gross, M. H. *Towards a game agent. Tech. Rep. 377, Institute of Scientific Computing, ETH Zürich, 2002.*

[20] Nowé, A., Verbeeck, K., and Lenaerts, T. *Learning agents in a homo egualis society. In* Proceedings of Learning Agents Workshop at Agents 2001 Conference *(2001).*

[21] Pottinger, D. C., and Laird, J. E. *Game AI: The state of the industry, part two, 2000. www.gamasutra.com/features/20001108/ laird_01.htm.*

[22] Stone, P., and Sutton, R. S. *Scaling reinforcement learning toward robocup soccer. In* Proc. 18th International Conf. on Machine Learning *(2001).*

[23] Sutton, R. S., and Barto, A. G. Reinforcement Learning: An Introduction. *MIT Press, 1998.*

[24] Tesauro, G. *Multi-agent learning mini-tutorial. Multi-Agent Learning: Theory and Practice Workshop at NIPS2002, 2002.*

[25] Torque game engine. www.garagegames.com.

[26] Wolpert, D., Sill, J., and Tumer, K. *Reinforcement learning in distributed domains: Beyond team games. In* Proceedings of the Seventeenth National Conference on Artificial Intelligence *(2001), pp. 819–824.*

[27] Wolpert, D. H., and Lawson, J. W. *Designing agent collectives for systems with markovian dynamics. In* Proceedings of the First International Joint Conference on Autonomous Agents and Multiagent Systems *(2002), C. Castelfranchi and W. L. Johnson, Eds., vol. 3, pp. 1066–1073.*

[28] Woodcock, S. *Game AI: The state of the industry, 2000. www.gamasutra.com /features/20001101/woodcock.htm.*

[29] Woodcock, S. *Recognizing strategic dispositions: Engaging the enemy. In* AI Game Programming Wisdom. *Charles River Media, 2002.*

[30] Woodcock, S. *AI roundtable moderator's report. www.gameai.com, 2003.*

Academic AI and Video games: a case study of incorporating innovative academic research into a video game prototype

Aliza Gold
Digital Media Collaboratory, IC^2 Institute
University of Texas at Austin
2815 San Gabriel Street
Austin, TX 78705
aliza@icc.utexas.edu

Abstract- Artificial intelligence research and video games are a natural match, and academia is a fertile place to blend game production and academic research. Game development tools and processes are valuable for applied AI research projects, and university departments can create opportunities for student-led, team-based project work that draws on students' interest in video games. The Digital Media Collaboratory at the University of Texas at Austin has developed a project in which academic AI research was incorporated into a video game production process that is repeatable in other universities. This process has yielded results that advance the field of machine learning as well as the state of the art in video games. This is a case study of the process and the project that originated it, outlining methods, results, and benefits in order to encourage the use of the model elsewhere.

1 Introduction

Video game technology can provide a rich platform for validating and advancing theoretical AI research (Adobbati et al. 2001, Laird 2001, Isla & Blumberg 2002, Fogel 2003). Therefore, projects that can bring the areas of artificial intelligence research and video game technology together can uniquely benefit academia. These benefits can sometimes extend to industry as well. A research and development team at the Digital Media Collaboratory (DMC) in the IC^2 Institute at the University of Texas at Austin has developed a process for utilizing video game production techniques to both apply and advance academic AI research. The process has yielded successful results and demonstrates one way to bring the two domains together. In the NeuroEvolving Robotic Operatives (NERO) project, a team of student artists and programmers has created a prototype game based on a Ph.D. candidate's novel neuroevolution (i.e. the evolution of neural networks with a genetic algorithm) method. Although this ongoing project has not been funded by an industry group, it has incorporated industry needs into its goals. It is possible to blend game development and research, and building 'killer apps' for industry is within reach for academia, without sacrificing research results. In fact, this project produced dissertation-level research that could not have been accomplished without the use of an off-the-shelf game engine as a test-bed.

Projects like NERO are repeatable. Currently there is a unique confluence of the ideas, people, tools, and technology required to implement similar projects. Many AI researchers today have grown up playing video games and are motivated to improve them, and at the same time many university departments are beginning to see video games as a legitimate medium, worthy of study and research (Laird & van Lent 2000). These schools are bringing people from the game industry to their programs so that game industry veterans and academic research students can work alongside each other. Simultaneously, some game development tool companies have forged new business models that allow them to price their products in ways that university programs can afford. And finally, exponential increases in processor power have enabled advanced AI to run in real time on personal computers.

Thus there is an opportunity to create projects in academia that yield potential benefits for both research and commercial interests. Current university projects reflect this interest in using computer games as a test-bed for AI research (Adobbati et al. 2001, Geisler 2002, McLean 2003). These articles report results in applied AI research, however few articles document the development processes used in obtaining the results. Discussing project development processes can aid groups in other universities to conduct applied artificial intelligence research projects with video game test beds successfully. This article outlines the process the DMC has employed to develop such a project. It describes the approaches used, challenges encountered, and makes suggestions for implementing projects such as these in other academic environments.

2 Background

When initiating NERO, the project team was motivated by simultaneous goals: 1) achieve tangible academic research results, and 2) create an application that could demonstrate and advance the state of the art of artificial intelligence in the video game industry. The project leaders hypothesized

that video game production techniques combined with academic research methods could achieve these goals.

Computer and video game entertainment software originally evolved from both action-oriented digital games played in arcades and from fantasy role-playing gamers who used the computer as a digital storytelling medium (Garriott 2002, King 2003). These games were small and could be programmed by one person or a team of a few people. Over the past 20 years, computer and video games have gained enough popularity to be considered a mainstream entertainment medium provided for by an industry with revenues of 7 billion US dollars in 2003 (ESA 2004). The most popular video games today are complex systems, and developing them requires large project teams of up to 100 people (Bethke 2003) with the associated project management infrastructure. Large teams producing complex projects require management processes in order to reliably produce results within the constraints of a schedule and budget. Formal software engineering processes have been applied as useful paradigms for projects in the commercial sector, including video game development.

In the NERO project, the team attempted to use standard software development models utilized in video game production with varying success as discussed in Section 3.2.2. The examples defined here provide clarity in the discussion of NERO:

- Waterfall method. This model was first introduced by Royce (1970) as software projects grew in size and complexity. It stipulates that projects follow distinct phases of requirements gathering, preliminary program design, analysis, coding, and testing, with documentation produced at the end of each phase. Each phase is completed before the next phase begins.
- Spiral model. Boehm (1988) developed the spiral model as a way to manage the risk that arises from uncertainties in a project. The spiral begins with well-defined exploratory phases in which risks are assessed and mitigations determined. As more definition for the software is achieved, a series of prototypes may be developed iteratively or it may be appropriate to transition to a waterfall or other model.
- Incremental method. This is a parallel process method used primarily on large projects. Rather than an entire project team following the waterfall method, the coding of separate segments of an application's system are begun according to when requirements can be defined for them.

Additional concepts from management science for the commercial sector are also useful in understanding the circumstances in which the NERO project was developed:

- Interdisciplinary teams. Also known as cross-functional teams, these are comprised of members with expertise from a variety of domains relevant to the project.
- Skunkworks. This term refers to a process engineering method innovation pioneered in the 1940s at Lockheed Martin (Brown 2001). It is characterised by the use of a small group completing a prototype or project from beginning to end (Wolff 1987). The team is given its own independent space and autonomy from the bureaucratic norms of the larger organization.

The waterfall method has been a useful tool for the game industry as game projects coordinate the efforts of designers, programmers, and artists to create products that are both functional and engaging for game-playing audiences. All groups contribute to the design of the product and group understanding of the final design is crucial to the success of the project. However, a video game can consist of numerous sub-systems within the larger product, and risk management strategies like those in the spiral method are sometimes utilized, as well as incremental strategies allowing different systems modules to develop within their own time cycles inside the larger project (Gattis & Waldrip 2004). The NERO project leadership experimented with all of these software development models. Some proved useful, while others presented unique challenges in an academic environment.

3 Project NERO

3.1 From NEAT to NERO

Laird (2000) listed three reasons that the closer ties and working relationships between the video game industry and the artificial intelligence research community that many parties identify as mutually beneficial has recently seen an increase in activity: 1) advancing technology with better processor power, 2) student gamers now in universities are doing AI research, and 3) players are demanding better AI in video games.

This change was the impetus for a workshop held by the DMC in August of 2003 on the topic of AI in Video Games, where University of Texas at Austin researchers and game industry developers presented their research in a single conference track. One of the researchers was then Ph.D. candidate Kenneth O. Stanley (completed August, 2004), who presented his method for evolving neural networks called NeuroEvolution of Augmenting Topologies (NEAT; Stanley & Miikkulainen 2002). Mat Buckland, in his 2002 book *AI Techniques for Game Programming* posited the potential for NEAT to improve AI in video games. By the time of the workshop Stanley had conceived of a game that would require machine learning capabilities that intelligent agents trained with the NEAT method could provide. In breakout sessions for game design he proposed the game idea and then presented it to the conference. After the conference, the DMC decided to create a prototype of Stanley's game idea, instantiating an applied research project that would build on Stanley's basic NEAT research and create a real-time implementation of it (Stanley et al. 2004). This project came to be called NERO, based on the game concept of training and battling robotic armies: NeuroEvolving Robotic Operatives.

Video game prototypes or demos often function as a demonstration of the fully developed game and as a proof

Figure 1: A screenshot from a training scenario in the NERO prototype in which AI agents learn to find their way around obstacles to a goal

of concept for the technology behind the game (Bethke 2003). The NERO project was envisioned similarly, and by conducting the implementation of the AI method into a game software environment, the project generated tangible research, real-time NEAT (rtNEAT), a new derivative of NEAT which was reported in Stanley's dissertation (Stanley 2004).

3.2 Development Methods

3.2.1 Environment/Context

The skunkworks construct (Section 2) has many similarities to the project circumstances and environment at the DMC. The physical location of the project has always been the DMC lab, where team members meet and work individually and collaboratively. The DMC is part of a university research unit, not a department in a college, and thus does not need to follow departmental norms. The DMC director has given few directives to the project staff except for requesting a target milestone date to have a functional prototype completed. The core leadership of the NERO team has been the same since the inception, as in Wolff (1987), described in Section 2. The team's work has been design and prototyping.

3.2.2 Process

The DMC team's original hypothesis was that applying game development team structures and typical game software development methods (Bethke 2003) to an applied AI research project would yield constructive results. Thus the leadership began with the waterfall method, as most video game industry projects do, to confirm requirements and create a preliminary design. However, the team immediately encountered difficulty with the waterfall method, since it was unclear who would be the ultimate customer of this technology, and whether the DMC would release NERO as a game eventually. Similar problems arose when attempting high-level game design because at the beginning the team did not have enough knowledge about how NEAT would work in the game engine, or if it could be made to work in real time. Thus starting with the typical waterfall phases of requirements and preliminary design, as game projects normally would, was not appropriate for the NERO project.

As noted by McCormack and Verganti (2003) in their discussion of uncertainty in projects and how they affect the development process, "different types of projects carried out in different environments are likely to require quite different development processes if they are to be successful." This is echoed by game developers who acknowledge that video games, for which custom software systems are often built, frequently require a custom mix of development approaches (Gattis & Waldrip 2004).

The NERO leadership team evolved an approach suited to the research and development challenges the project has faced. The team adapted the spiral method, defining specific intermediate goals for research and production, and choosing target completion dates. Since the university semester schedule imposes long breaks and team turnover among student participants, semesters provide natural milestones for the project in which the team can evaluate progress, identify the next round of goals and tasks, and evaluate which tasks to tackle immediately and which to postpone based on current resources.

Implementing the NEAT AI technique in a 3D game engine required crafting a series of experiments that would both validate and advance the capabilities of NEAT. At the same time, choosing the commercial engine Torque

(Garage Games) as a development platform and research test-bed required many modifications to accommodate rtNEAT and implement the NERO game design around it. NERO has incorporated academic research goals, alongside concurrent production involving engine coding and art creation. The team leadership also has adapted an incremental approach to the project by having some programmers work individually or in small groups on different parts of the prototype, some conducting research experiments, while others work on the user interface or other game-engine related tasks. The research aspect of the project has taken a large percentage of the total effort and often drives the other development tasks. As the project progressed, the application increasingly took on the appearance of a game while research simultaneously advanced (Figure 1).

3.2.3 Team Structure

The game industry relies on interdisciplinary teams (Section 2) to develop complex entertainment software products. The NERO project has used team structure and development methods borrowed from the game industry, but on a much smaller scale and in an academic environment. Thus for NERO a team-based structure has been employed, with subteams and associated team leads for programming, art, and design. Almost all of the team members have been undergraduate student volunteers with no professional experience in industry. Of those who are not volunteers, the project producer and the lead designer are staff members of the DMC, and the AI researchers Kenneth Stanley and Bobby D. Bryant, Ph.D. candidate, are both on fellowships. The project was not prohibitively costly to implement since volunteers and staff members completed much of the work.

For the art team and programming team leadership positions, which sometimes change, the producer tapped willing volunteers. They usually had only classroom experience in project work and leadership. However the result of giving undergraduates positions of responsibility has been that they perform beyond expectations, carrying out research and solving difficult problems (Kanter 1985).

The programming and art leads make up half of the project leadership team. Stanley and Bryant, the producer, and the designer make up the rest of the leadership team, who meet weekly to make decisions and coordinate all aspects of the project. These meetings are crucial for directing the production work, staying abreast of the research progress, and continually evaluating how the advancing research impacts the design and production. The DMC lab has been the center of operations for all meetings and most of the project work.

3.2.4 Design Considerations

For the NERO project, there have been two distinct but related areas of design – the game design aspect of the prototype and the research design. The game designer has created a context for Stanley's game concept, which included background fiction, a visual design concept, and some game design elements. This vision is a motivator for the volunteer team, who are excited to work toward a real game. The two AI researchers have had the responsibility of designing the series of experiments that document and validate the success or lack thereof of implementing NEAT into the Torque game environment and exhibiting the desired capabilities (Figure 2 shows an example of an experiment series). The game design and the research design have been intertwined throughout development because what has been possible in the game depended on the capabilities of the artificial intelligence. As research and production have continued, new possibilities have presented themselves, based on what has been accomplished in the research. Sometimes new research directions have appeared, which implied accompanying shifts in the game design. This has contributed to the usefulness of the spiral development method for this project, where design choices are made iteratively, prototypes are produced to reflect the new choices, and risks are evaluated at each turn (Boehm 1998).

Experiments -
E1 – Learn to move to approach some stationary target.
For convenience, just use an enemy for the target.
E2 – Learn to acquire and shoot to hit a stationary target.
May not require a factory.
b – Ditto, except with a mobile target.
E3 – Learn to approach a stationary turret that is firing in an oscillating pattern, without getting killed.
b – Ditto, except against two turrets.
E4 – Learn basic assault tactics: move across the map into firing range and destroy a stationary turret that is firing in an oscillating pattern, without getting killed.
b – Ditto, except against two turrets.
E5 – Deploy two teams trained as in E4b in a combat area, and show that they display appropriate behavior.
No evolution involved.
Use teams trained to different behaviors, e.g. "move a lot" vs. "don't move much", "disperse a lot" vs. "stay bunched up", or "approach the enemy" vs. "avoid the enemy".

Figure 2: An excerpt from the first plan of experiments

3.2.5 Results

The NERO project has resulted in a playable video game prototype, now its second version. An exciting result is that the novel aspect of rtNEAT has yielded novel gameplay in the prototype (NERO will be presented in the Experimental Gameplay Workshop at the 2005 Game Developers Conference).

In NERO, a player trains a group of ignorant robot soldiers by setting learning objectives for the group through an interface. After the objective is set, the robots learn in real time to achieve their goal. The player can incrementally increase the challenge for the robot soldiers though the learning objective interface and by customizing the training arena, thus creating increasingly sophisticated

behavior in the group. Once the real-time training is completed to the player's satisfaction, the player can save the team. Then the player can battle the team against the computer's or a human opponent's team, observing the results of the training in a battle situation, where learning is no longer taking place. The real-time training in NERO, powered by rtNEAT, is a type of gameplay that currently does not exist in commercial games. This new style of gameplay could lead to new video game genres.

NERO has been a successful project because it has resulted in innovations in terms of both research and what is possible in games. NEAT first had to be reengineered to work in real time, producing one of the first genetic algorithm systems to do so. Stanley created rtNEAT, an innovation that he included in his doctoral dissertation (Stanley 2004), and with the undergraduate lead programmer implemented the method into the Torque 3D engine. Sensors had to be engineered to enable the agents to perceive elements of the environment such as enemies or walls. The agents perceive the world differently than typical non-player characters in a video game that are fed information about the environment through the game system. NERO agents sense the environment egocentrically, more like robots in the physical world, and thus had to be equipped with egocentric sensors.

Another essential element of the training function of the game is the interface where players, from their perspective, assign objectives to the agents during training, such as approaching the on-field enemy or avoiding getting hit by fire. This interface gives players access, without their explicit knowledge, to the fitness functions of the agents, allowing them to interact with the evolution of the networks in real time, another innovation enabled by rtNEAT (Stanley et al. 2004).

Additionally, art students have created robot models to embody the agents, and game level environments to more aptly demonstrate the agents' capabilities. Thus at this point NERO is a successful demonstration of NEAT, and as such has generated considerable interest from both companies in the game industry and groups in the military. A provisional patent has been filed on rtNEAT, an SDK is being developed, and licensing options are being investigated. NERO has been an experiment on multiple levels: artificial intelligence research, process management, and video game design. At each level interesting results have emerged.

4 Discussion

4.1 Project-Based Considerations

Over the yearlong course of developing the NERO project, there has been no road map to tell the leadership how to direct a project of this kind. Some of the solutions the leadership has developed may be generalizable to other projects.

4.1.1 Industry Input

Other projects of this kind would benefit from getting input from industry early in the project. Whether through casual conversations or formal presentations with industry professionals, input concerning what would be desirable or innovative is essential if a goal is to eventually commercialize the research. It would be undesirable for academic groups who are unfamiliar with video games to attempt to solve problems that are considered already solved in the game industry. Posing a question such as "If our group had a technology that could do [X technology functionality], what would you expect it to be capable of in a game environment?" would yield valuable information that a team could use to guide the development of the project by comparing industry interests to the group's own research interests.

4.1.2 The Research and Production Mix

In a project where the goal is to create a technology that interests industry while at the same time advancing research in an academic field, maintaining awareness of those sometimes opposing goals is important. In the NERO project the leadership team has experienced the tension between those goals, and that has been an occasional source of conflict. In any project phase, there is a limit to the resources available to apply to project tasks, and choices must be made and priority given to some tasks over others. Transition points between phases are the best time to evaluate the meta-goals of the project for the upcoming phase.

4.1.3 Technology Considerations

Using an existing game engine (Torque) as a platform for applying new artificial intelligence methods has been very helpful for the NERO team. It has allowed the team to develop a functional prototype fairly quickly after the effort of making NEAT workable in Torque. Additionally, using a commercial engine gives our undergraduate programmers the experience of working with an industry quality engine, which is valuable for industry hopefuls. However, it is also important to be aware of the limitations of the chosen engine or platform. The Torque engine is designed for the first-person shooter game genre, a very different kind of game than NERO is intended to be; thus some functionality that would ideally already exist does not. For instance, the training aspect of NERO requires the player to build training environments, and an engine similar to the one used in the video game SimCity would have that functionality built in. Currently there is not an affordable engine for every existing game genre, and thus much functionality may need to be built from scratch.

Alongside functionality considerations are other technical concerns that should be understood when evaluating engine platforms. In Torque, the STL (Standard Template Library) is not supported, which required that the NEAT code be rewritten so as not to use that library. This was a surmountable challenge, but non-trivial, and has kept some researchers from wanting to use the engine.

4.1.4 Process

When dealing with research experiments it is much more difficult to create a reliable schedule based on the estimated level of effort, as would be done with a commercial project. Encountering unexpected difficulties when conducting experiments can extend the time to complete them far beyond what a team lead might imagine or plan. The NERO leadership has found it more productive to create a milestone list of experiments and production work based on the semester, without assigning specific completion dates. Together the leads create an agreed-upon list of goals and track the completion of them throughout the semester.

4.1.5 Team

The primary difficulties encountered when working with an undergraduate student volunteer team are inexperience and turnover. The NERO project leadership has developed processes and techniques for bringing on new students and retaining them:

1. Recruiting process. It is most efficient to bring on new programmers and train them as a group rather than individually. That also gives them camaraderie as new members of the team.
2. High expectations from the outset. Too many uncommitted people on the team are an administrative overload for the lead. By broadcasting the need for commitment to the project, people know what to expect and the lead can limit participation to members who are likely to stay on the project and contribute significantly (Botkin 1985).
3. Weekly meetings. These are important for each subteam to keep people on track and help them understand their contributions in a group context. Meetings are especially important for team members who lack experience working in a group toward a common goal.
4. Mentorship. Create an environment where the more senior members can advise newer ones; with new members joining the team each semester, it can take some of the burden off the lead to establish times where most members are in the work space, thus enabling mentoring opportunities.
5. Communication. Project leaders improve morale by communicating the meta-goals for the project phase and gathering team members' input. As implementers, team members have valuable insights and perspectives of which the project leadership needs to be aware.
6. Fun. Project leaders can demonstrate their appreciation to the team by hosting game nights, bringing in industry speakers, and recognizing accomplishments whenever possible.

While some might be concerned that having volunteers on the project is exploitative, it is apparent that the experience on the project has proved beneficial for undergraduate team members. Game projects appeal to undergraduates, and many students hope to find employment in the game industry, which is notoriously difficult to enter. Working on a game engine with a team provides valuable experience toward that goal. Other students who are less driven by game industry goals value the opportunity to work with Ph.D.-candidate-level graduate students doing innovative research, which many undergraduates find inspiring. Undergraduates are a frequently overlooked group; however, they have proved to be crucial contributors to the NERO project team. NERO project veterans have gone on to create their own lines of research based on the project or, following their graduation, have found employment with local game development studios. The lessons learned with NERO are serving the project team well as it goes forward into a second year of development, reducing the time spent on administrative necessities such as student recruiting, and allowing more time for research and game production work.

4.2 Institutional Considerations

Accompanying the project-based considerations in conducting a project like NERO, there are also issues at the institutional level. There is an increasing movement in academia, in the US, Europe, and elsewhere, to commercialize university research, a movement that has created controversy in the academic community. Proponents cite the important benefits for society, such as bringing technology improvements to the public, job creation, and funding for universities. However, there are serious concerns among academic leaders that market-driven influences will corrupt academic research processes with conflicts of interest and manipulated research agendas (see McMaster 2002, Bok 2003). Nevertheless, NERO has shown that it is possible to create university/industry collaborations that can provide educational benefits to students as well as financial benefits to departments, without compromising basic research.

Dr. George Kozmetsky, the founder of the University of Texas at Austin's IC^2 Institute, promoted an agenda of capitalism for the public good. The Institute has a long history of technology commercialization with groups under its umbrella such as the Austin Technology Incubator and the Masters of Science in Science and Technology Commercialization degree program. The DMC is the newest organization within the IC^2 Institute, created in part to apply basic research with the University of Texas' faculty and students and bring it to the public/commercial sector. The DMC is an obvious organization to implement projects of this kind, and the NERO project has been an initial implementation of Dr. Kozmetsky's vision.

The DMC has provided a fertile ground for implementing NERO as an academic applied research project that supplies some of the project's research goals by observing industry needs. University departments anywhere can develop similar projects by creating the necessary infrastructure to implement them. Departments can create opportunities for applied research projects outside of students' normal classwork, supplementing their knowledge acquisition with collaborative, project-based educational experiences. Creating this infrastructure

does not require a large investment of departmental resources.

The purpose of having a structure in which projects occur outside of classes is to provide an avenue for the creation of projects that develop for longer than a semester. It also allows students to be driven by motivations other than external class-related measures, such as receiving grades.

In order to create such an infrastructure, university departments could institute processes such as these:

- Create a process to invite applied research project ideas, requiring a group implementation, from graduate students and faculty, based on their research.
- Have a committee that includes faculty and industry members to choose the best ideas based on industry interests and advancements in the relevant academic field.
- Choose a project manager to create teams based on the chosen project ideas. This coordinator could be a student in the department, a faculty member, or a student from another major such as business. This role requires commitment, deep interest in the research area, and organizational and communication skills. It may be advisable to assign a departmental staff member the job of supervising these project coordinators to provide mentorship and guidance.
- Give students autonomy in how they develop their projects by allowing project leaders to set goals in collaboration with the other team leads.
- Have a review process at appropriate milestones with team members, faculty members, and industry members in order to assess project progress.

To broaden the participation of students from other disciplines, invite students from departments such as communications, art, business, design, and information sciences. All of these departments represent roles in game industry projects, which can increase the real world impact of the project.

5 Implications and Future Work

The NERO project is an example of the advantages of leveraging existing technology (e.g. off-the-shelf game engine applications) for advancing academic research. Such technologies, which have become increasingly more affordable for university department budgets, can supply one researcher or a team with the foundation to relatively quickly implement research technologies such as machine learning, robotics, databases, or other areas. These technologies provide platforms for visualization and validation, and because they support high-quality demonstrations, these platforms can be a bridge to commercialization of research technologies.

After implementing rtNEAT in the NERO game, other potential applications have become apparent, even within the realm of video game technology. The rtNEAT method could be implemented in a persistent online 3D environment. Massively multiplayer online games (MMOGs) are one possibility for such an implementation. In a multiplayer persistent virtual world, the agents, as non-player characters, could evolve new behaviors by reacting to players' choices. Non-player-characters with true adaptive, not scripted, behavior are very desirable for game developers and players alike (Ilsa & Blumberg 2002, Fogel 2003). Non-player-characters in current MMOs have predictable behaviors with a limited ability to provide players with new challenges. To address game designer needs, a back-end interface to the evolution similar to the NERO interface would ensure that designers had the necessary control over the agents' behavior, which is an important industry consideration.

There are also implications for non-game environments. By modeling the agents as animals or other biological entities and giving them appropriate animations, the adaptation ability shown in NERO might result in artificial life agents who could evolve for weeks, months, or perhaps years, based on the concept of open-ended evolution (Taylor 1999). This sort of artificial ecology experiment would be an exciting next step that could provide a test-bed for exploring the limits of rtNEAT's complexification characteristics.

Another conclusion of this case study is that closer collaboration with the video game industry could bring significant benefits to the academic community. One of these benefits is in the area of applied research problems. Most AI machine learning researchers who have worked with games until recently have used board games such as checkers or Go (Furnkranz 2000) as research areas. In contrast, the video game industry offers complex and novel application challenges, which have "real world" status that can motivate student teams who are interested in advancing the state of the art in game technology.

Game companies might be persuaded to provide funding to departments engaged in projects that could produce usable technologies if they understood the benefit of having highly specialized academic research applied to the video game domain on problems they need solved. Again, departments must always weigh industry interests against the interests of advancing the research in the relevant academic field, and attempt to find areas of commonality.

6 Conclusions

Video game technology has given the rtNEAT method a platform to demonstrate its capabilities, which has proved invaluable for introducing it to the world. Projects such as NERO can be repeated in other universities where there is support for innovative ways of conducting applied research in computer science or interdisciplinary departments. Universities can use video game technology to accelerate research advancement as well as reveal commercialization opportunities. These projects provide motivating challenges for students who are highly interested in doing innovative work, even as undergraduates.

University departments can draw on students' intense interest in video games. Research conducted by the DMC in 2003 shows that, among middle school students in Texas, 75% of boys and 50% of girls were found to play video games (Gold et al. 2005). With so many university students engaged as users of this medium, there is a large pool of potential team members for game-related projects.

Ultimately, as innovative artificial intelligence techniques are implemented in commercial video games and AI students and professors leverage video game technology as applied research platforms, everyone benefits.

Acknowledgments

The author thanks Risto Miikkulainen for his crucial initial support and helpful suggestions for the analysis. Bobby D. Bryant, former NERO lead programmer, Kenneth O. Stanley, and Damon Waldrip provided insightful comments on the manuscript and editorial assistance. The success of the project has depended on the contributions of Michael Chrien, NERO lead programmer, and the entire NERO team, past and present. The project has only been possible with the support of the DMC staff: Alex Cavalli, Melinda Jackson, Aaron Thibault, and Matt Patterson, NERO designer.

Bibliography

Adobbati, R., Marshall, A., Scholer A., Tejada, S., Kaminka, G., Schaffer, S., and Sollitto, C. (2001). Gamebots: A 3D virtual world test-bed for multi-agent research. *Proceedings of the 2nd Workshop on Infrastructure for Agents, MAS, and Scalable MAS.* New York, NY: ACM Press. □

Botkin, J.W. (1985) Transforming creativity into innovation: Processes, prospects, and problems. In *Frontiers in Creative and Innovative Management,* Kuhn, R. Ed. Cambridge, MA: Ballinger Publishing Company.

Bethke, E. (2003). *Game Development and Production*. Plano, TX: Worldware Publishing.

Boehm, B. W. (1988) A spiral model of software development and enhancement. *IEEE Computer*: 21.

Bok, D. (2003). *Universities in the Marketplace*. Princeton, NJ: Princeton University Press.

Brown, T. (2001). Skunk works: A sign of failure, a sign of hope? For the *Future of Innovation Studies Conference*. The Netherlands.

Buckland, Mat. (2002) AI techniques for game programming. Boston, MA: Premier Press.

Entertainment Software Association. (2004). *Essential Facts about the Computer and Video game Industry.*

Fogel, D. (2003). Evolutionary entertainment with intelligent agents. *IEEE Computer:* 36.

Furnkranz, J. (2000). Machine learning in games: A survey. Austrian Research Institute for Artificial Intelligence. Technical Report OEFAI-TR-2000-31.

Garriott, R. (2002). The origins of Ultima. Presentation at the *Digital Media Collaboratory GameDev 2002 Workshop.*

Gattis, G., and Waldrip, D. Interviewed by author, October 10, 2004.

Geisler, B. (2002). An empirical study of machine learning algorithms applied to modeling player behavior in a "first-person shooter" video game. Masters thesis, Department of Computer Science, University of Wisconsin-Madison.

Gold, A., Durden, E. Alluah, S., Kase, M. Career aspirations and their influences among central texas middle school youth. To be presented at the *American Educational Research Association 2005 Conference.*

Isla, D. & Blumberg, B., (2002). New challenges for character-based AI for games. *AAAI Spring Symposium on AI and Interactive Entertainment*, 2002

Kanter, R.M. (1985). Change-Master skills: What it takes to be creative. In *Handbook for Creative and Innovative Managers.* Kuhn, R. Ed. New York, NY: McGraw-Hill.

King, B. and Borland, J. (2003). *Dungeons and Dreamers: The Rise of Video Game Culture*. Emeryville, CA: Mcgraw-Hill/ Osborne.

Laird, J. E. (2000). Better AI development: Bridging the gap. *Game Developer Magazine*. August 2000.

Laird, J. E. (2001). Using a computer game to develop advanced AI. *IEEE Computer:* 34.

Laird, J. E., and van Lent, M. (2000). Human-level AI's killer application: Interactive computer games. In *Proceedings of the 17th National Conference on Artificial Intelligence.* Cambridge, MA: MIT Press.

MacCormack, A., and Verganti, R. (2003). Managing the sources of uncertainty: Matching process and context in software development. *Journal of Product Innovation Management, 20: 217-232.*

McLean, S. (2003). Multi-agent cooperation using trickle-down utility. Ph.D. thesis, Division of Informatics, University of Edinburgh.

McMaster, G. (2002) The balancing act of commercializing research. *Atlanta Express News,* February 8.

Royce, W. W. (1970). Managing the development of large software systems. *Proceedings, IEEE WESCON,* August: 1-9.

Stanley K. O. (2004). Efficient evolution of neural networks through complexification. Department of Computer Sciences, University of Texas at Austin. Technical Report AI-TR-04-39.

Stanley, K. O., Bryant, B. D., and Miikkulainen, R. (2004). The NERO real-time video game. To appear in: *IEEE Transactions on Evolutionary Computation*, special issue on *Evolutionary Computation and Games.*

Stanley, K. O. and Miikkulainen, R. (2002). Evolving neural networks through augmenting topologies. *Evolutionary Computation,* 10(2).

Taylor, T. (1999) From Artificial Evolution to Artificial Life. Ph.D. thesis, Division of Informatics, University of Edinburgh.

Case-Injection Improves Response Time for a Real-Time Strategy Game

Chris Miles
Evolutionary Computing Systems LAB (ECSL)
Department of Computer Science and Engineering
University of Nevada, Reno
miles@cs.unr.edu

Sushil J. Louis
Evolutionary Computing Systems LAB (ECSL)
Department of Computer Science and Engineering
University of Nevada, Reno
sushil@cs.unr.edu

Abstract- We present a case-injected genetic algorithm player for Strike Ops, a real-time strategy game. Such strategy games are fundamentally resource allocation optimization problems and our previous work showed that genetic algorithms can play such games by solving the underlying resource allocation problem. This paper shows how we can learn to better respond to opponent actions (moves) by using case-injected genetic algorithms. Case-injected genetic algorithms were designed to learn to improve performance in solving sequences of similar problems and thus provide a good fit for responding to opponent actions in Strike Ops which result in a sequence of similar resource allocation problems. Our results show that a case-injected genetic algorithm player learns from previously encountered problems in the sequence to provide better quality solutions in less time for the underlying resource allocation problem thus improving response time by the genetic algorithm player. This improves the responsiveness of the game and the quality of the overall playing experience.

1 Introduction

The computer gaming industry now has more games sales than movie ticket sales and both gaming and entertainment drive research in graphics, modeling and many other areas. Although AI research has in the past been interested in games like checkers and chess [1, 2, 3, 4, 5], popular computer games like Starcraft and Counter-Strike are very different and have not yet received much attention from evolutionary computing researchers. These games are situated in a virtual world, involve both long-term and reactive planning, and provide an immersive, fun experience. At the same time, we can pose many training, planning, and scientific problems as games where player decisions determine the final solution. This paper uses a case-injected genetic algorithm to learn how to play Strike Ops a Real-Time Strategy (RTS) computer game [6].

Developers of computer players (game AI) for popular First Person Shooters (FPS) and RTS games tend to use finite state machines, rule-based systems, or other such knowledge intensive approaches. These approaches work well - at least until a human player learns their habits and weaknesses. The difficulty of the problem means that competent Game AI development requires significant player and developer resources. Development of game AI therefore suffers from the knowledge acquisition bottleneck well known to AI researchers.

Since genetic algorithms are not knowledge intensive and were designed to solve poorly understood problems they seem well suited to RTS games which are fundamentally resource allocation games. However, genetic algorithms tend to be slow, requiring many evaluations of candidate solutions in order to produce satisfactory allocation strategies.

To increase responsiveness and to speed up gameplay, this paper applies a case-injected genetic algorithm that combines genetic algorithms with case-based reasoning to provide player decision support in the context of domains modeled by computer strategy games. In a case-injected genetic algorithm, every member of the genetic algorithm's evolving population is a potential case and can be stored into a case-base. Whenever confronted with a new resource allocation problem (P_i), the case-injected genetic algorithm learns to increase performance (playing faster and better) by periodically injecting appropriate cases from previously attempted similar problems ($P_0, P_1, \ldots, P_{i-1}$) into the genetic algorithm's evolving population on P_i. Case-injected genetic algorithms acquire domain knowledge through game play and our previous work describes how to choose appropriate cases, how to define similarity, and how often to inject chosen cases to maximize performance [6].

Players begin Strike Ops with a starting scenario and the genetic algorithm player attempts to solve P_0, the underlying resource allocation problem. During game play, an opponent's action changes the scenario creating a new resource allocation problem P_1 and triggering the genetic algorithm to attempt to solve P_1. Combined with case-injection, the genetic algorithm player uses past experience at solving the problem sequence generated by opponent actions, $P_1, P_2, \ldots, P_{i-1}$, to increase performance on current and subsequent problems, $P_i, P_{i+1}, \ldots, P_{i+n}$. Preliminary results show that our case-injected genetic algorithm player indeed learns to play faster and better.

The next section introduces the strike force planning game and case-injected genetic algorithms. We then describe previous work in this area. Section 4 describes the specific strike scenarios used for testing, the evaluation computation, our system's architecture, and our encoding. The subsequent two sections explain our test setup and our results with using case-injected genetic algorithm to increase our performance at playing the game. The last section provides conclusions and directions for future research.

2 Strike Ops

A Strike Ops player wins by destroying opponents' installations while minimizing damage to the player's own installations and assets. An attacking player allocates a collection of strike assets on flying platforms to a set of opponent targets and threats on the ground. The problem is dynamic; weather and other environmental factors affect asset performance, unknown threats can popup and become new targets to be destroyed. These complications as well as the varying effectiveness of assets on targets make the game interesting and the underlying resource allocation problems difficult and thus suitable for genetic and evolutionary computing approaches. Figure 1 shows a screenshot from the game.

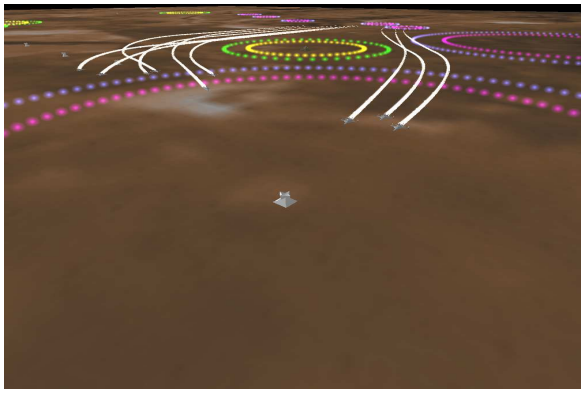


Figure 1: Game Screen-shot

Our game involves two sides: Blue and Red, both seeking to allocate their respective resources to minimize damage received while maximizing the effectiveness of their assets. Blue plays by allocating a set of assets on aircraft (platforms), to attack Red's buildings (targets) and defensive installations (threats). Blue determines which targets to attack, which weapons (assets) to use on them, as well as how to route each platform to the targets, trying to minimize risk presented while maximizing weapon effectiveness.

Red has defensive installations (threats) which protect its targets by attacking enemy platforms that come within range. Red plays by placing these threats to best protect targets. Potential threats and targets can also "pop-up" on Red's command in the middle of a mission, allowing a range of strategic options. By cleverly locating threats Red can feign vulnerability and lure Blue into a deviously located popup trap, or keep Blue from exploiting such a weakness out of fear of a trap. The many possible offensive and defensive strategies make the game exciting and dynamic.

In this paper, a human plays Red while a Genetic Algorithm Player (GAP) plays Blue. The fitness of an individual in GAP's population solving the underlying allocation problem is evaluated by running the game. GAP develops strategies for the attacking strike force, including flight plans and weapon targeting for all available aircraft. When confronted with popups or other changes in situation, GAP responds by replanning with the case-injected genetic algorithm in order to produce a new plan of action that responds to changes.

2.1 Case-Injected Genetic Algorithms for RTS games

The idea behind a case-injected genetic algorithm is that as the genetic algorithm component iterates over a problem it selects members of its population and caches them (in memory) for future storage into a case base [6]. Cases are therefore members of the genetic algorithm's population and represent an encoded candidate strategy for the current scenario. Periodically, the system injects appropriate cases from the case base, containing cases from previous attempts at *other, similar* problems, into the evolving population replacing low fitness population members. When done with the current problem, the system stores the cached population members into the case base for retrieval and use on new problems.

A case-injected genetic algorithm works differently than a typical genetic algorithm. A genetic algorithm randomly initializes its starting population so that it can proceed from an unbiased sample of the search space. We believe that it makes less sense to start a problem solving search attempt from scratch when previous search attempts (on similar problems) may have yielded useful information about the search space. Instead, periodically injecting a genetic algorithm's population with *relevant* solutions or partial solutions to similar previously solved problems can provide information (a search bias) that reduces the time taken to find a quality solution. Our approach borrows ideas from case-based reasoning (CBR) in which old problem and solution information, stored as cases in a case-base, helps solve a new problem [7, 8, 9]. In our system, the data-base, or case-base, of problems and their solutions supplies the genetic problem solver with a long term memory. The system does not require a case-base to start with and can bootstrap itself by learning new cases from the genetic algorithm's attempts at solving a problem.

The case-base does what it is best at – memory organization; the genetic algorithm handles what it is best at – adaptation. The resulting combination takes advantage of both paradigms; the genetic algorithm component delivers robustness and adaptive learning while the case-based component speeds up the system.

The Case-Injected Genetic Algorithm (CIGAR) used in this paper operates on the basis of solution similarity. CIGAR **periodically injects** a small number of *solutions* similar to the current best member of the GA population into the *current* population, replacing the worst members. The GA continues searching with this combined population. The idea is to cycle through the following steps. Let the GA make some progress. Next, find solutions in the case base that are similar to the current best solution in the population and inject these solutions into the population. If injected solutions contain useful cross problem information, the GA's performance will be significantly boosted. Figure 2 shows this situation for CIGAR when it is solving a sequence of problems, $P_i, 0 < i \leq n$, each of which undergoes periodic injection of cases.

We have described one particular implementation of such a system. Other less elitist, approaches for choosing population members to replace are possible, as are different

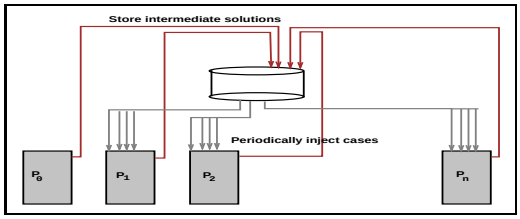


Figure 2: Solving problems in sequence with CIGAR. Note the muliple periodic injections in the population as CIGAR attempts problem $P_i, 0 < i \leq n$.

strategies for choosing individuals from the case base. We can also vary the injection percentage; the fraction of the population replaced by chosen injected cases.

Note that CIGAR *periodically* injects cases into the evolving population. We **have to** periodically inject solutions based on the makeup of the current population because we do not know which previously solved problems are similar to the current one. That is, we do not have a problem similarity metric. However, the hamming distance between binary encoded chromosomes provides us a simple and remarkably effective solution similarity metric. By finding and injecting cases in the case-base that are similar (hamming distance) to the current best individual in the population, we are assuming that similar solutions must have come from similar problems and that these similar solutions retreived from the case base contain useful information to guide genetic search. Results on design, scheduling, and allocation problems show the efficacy of this similarity metric and therefore of CIGAR [6].

An advantage of using solution similarity arises from the string representations typically used by genetic algorithms. A chromosome is, after all, a string of symbols. String similarity metrics are relatively easy to come by, and furthermore, are domain independent. For example, in this paper we use hamming distance between binary encoded chromosomes for the similarity metric.

What happens if our similarity measure is noisy and/or leads to unsuitable retrieved cases? By definition, unsuitable cases will have low fitness and will quickly be eliminated from the GA's population. CIGAR may suffer from a slight performance hit in this situation but will not break or fail – the genetic search component will continue making progress towards a solution. In addition, note that diversity in the population - "the grist for the mill of genetic search [10]" can be supplied by the genetic operators **and** by injection from the case-base. Even if the injected cases are unsuitable, variation is still injected. CIGAR is robust.

The system that we have described injects individuals in the case-base that are deterministically closest, in hamming distance, to the current best individual in the population. We can also choose schemes other than injecting the closest to the best. For example, we have experimented with injecting cases that are the furthest (in the case-base) from the current worst member of the population. Probabilistic versions of both have also proven effective.

Reusing old solutions has been a traditional performance improvement procedure. This work differs in that 1) we attack a set of tasks, 2) store and reuse **intermediate** candidate solutions, and 3) do not depend on the existence of a problem similarity metric. More details about CIGAR are provided in [6].

Genetic algorithms can be used to robustly search for effective strategies inside our game. A genetic algorithm can generate an initial resource allocation (a plan) to start the game. No plan survives contact with the enemy, however, as the dynamic nature of the game requires re-planning in response to opponent decisions (moves) and the ever changing game-state. Replanning must be done fast enough to keep the game lively and responsive, while plans produced must be effective enough to produce a competent opponent. Can genetic algorithms satisfy these speed and quality constraints?

We have shown that case-injected genetic algorithms learn to increase performance with experience at solving similar problems [6, 11, 12, 13, 14]. For the case-injected genetic algorithm, re-planning is simply solving a similar planning problem arising from opponent actions and this paper shows that when used for re-planning in Strike Ops, a case-injected genetic algorithm quickly produces better new plans in response to changing game dynamics. In our game, aircraft break off to attack newly discovered targets, reroute to avoid new threats, and re-prioritize to deal with changes to the game state.

2.2 Previous Work

Previous work in strike force asset allocation has been done in optimizing the allocation of assets to targets, the majority of it focusing on static pre-mission planning. Griggs [15] formulated a mixed-integer problem (MIP) to allocate platforms and assets for each objective. The MIP is augmented with a decision tree that determines the best plan based upon weather data. Li [16] converts a nonlinear programming formulation into a MIP problem. Yost [17] provides a survey of the work that has been conducted to address the optimization of strike allocation assets. Louis [18] applied case injected genetic algorithms to strike force asset allocation.

From the computer gaming side, a large body of work exists in which evolutionary methods have been applied to games [2, 19, 4, 20, 3]. However the majority of this work has been applied to board, card, and other well defined games. Such games have many differences from popular real time strategy (RTS) games such as Starcraft, Total Annihilation, and Homeworld[21, 22, 23]. Chess, checkers and many others use entities (pieces) that have a limited space of positions (such as on a board) and restricted sets of actions (defined moves). Players in these games also have well defined roles and the domain of knowledge available to each player is well identified. These characteristics make the game state easier to specify and analyze. In contrast, entities in our game exist and interact over time in continuous three dimensional space. Entities are not directly controlled by players but instead sets of parametrized algorithms control them in order to meet goals outlined by players. This adds a level of abstraction not found in more traditional

games. In most such computer games, players have incomplete knowledge of the game state and even the domain of this incomplete knowledge is difficult to determine. John Laird [24, 25, 26] surveys the state of research in using Artificial Intelligence (AI) techniques in interactive computers games. He describes the importance of such research and provides a taxonomy of games. Several military simulations share some of our game's properties [27, 28, 29], however these attempt to model reality while ours is designed to provide a platform for research in strategic planning, knowledge acquisition and re-use, and to have fun. The next section describes the scenario (mission) being played.

3 Missions

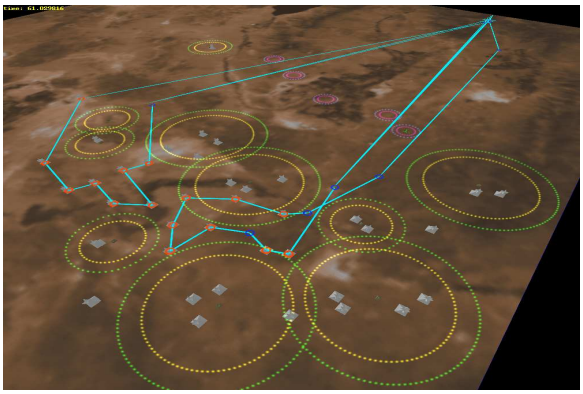


Figure 3: The Scenario

Figure 3 shows one of the missions being played by our GA. This mission takes place in north-west Nevada, Lake Tahoe is just off the lower left hand side of the screen. The rings represent the radii of the various threats, the darker colored rings represent threats that are currently inactive - they are waiting to surprise attackers. The square looking objects are targets and there are a number of them. The blue lines represent routes in a plan designed by a human playing blue. The human player has organized platforms into two wings and plans to thread through gaps in defense coverage to attack undefended targets. Evaluating a plan, or computing a plan's fitness is dependent on the representation of entities' states inside the game, and our way of computing fitness and representing this state is described next.

3.1 Fitness

We evaluate the fitness of an individual in GAP's population by running the game and checking the game outcome. Blue's goals are to maximize damage done to red targets, while minimizing damage done to its platforms. Shorter simpler routes are also desirable, so we include a penalty in the fitness function based on the total distance traveled. This gives the fitness calculated as shown in Equation 1

$$fit(plan) = Damage(Red) - Damage(Blue) - d*c \quad (1)$$

d is the total distance traveled by Blue's platforms and c is chosen such that $d * c$ has a 10-20% effect on the fitness ($fit(plan)$). Total damage done is calculated below.

$$Damage(Player) = \sum_{E \in F} E_v * (1 - E_s)$$

E is an entity in the game and F is the set of all forces belonging to that side. E_v is the value of E, while E_s is the probability of survival for entity E. We use probabilistic health metrics to evaluate entity damage.

3.2 Probabilistic Health Metrics

In many games, entities (platforms, threats, and targets in our game) posses hit-points which represents their ability to take damage. Each attack removes a number of hit-points and when reduced to zero hit-points the entity is destroyed and cannot participate further. However in reality, weapons have a more hit or miss effect, destroying entities or leaving them functional. A single attack may be effective while multiple attacks may have no effect. Although more realistic, this introduces a large degree of stochastic error into the game.

To mitigate stochastic errors, we use probabilistic health metrics [30, 31]. Instead of monitoring whether or not an object has been destroyed we monitor the probability of its survival. Being attacked no longer destroys objects and removes them from the game, it just reduces their probability of survival according to Equation 2 below.

$$S(E) = S_{t_0}(E) * (1 - D(E)) \quad (2)$$

E is the entity being considered: a platform, target, or threat. $S(E)$ is the probability of survival of entity E after the attack. $S_{t_0}(E)$ is probability of survival of E up until the attack and $D(E)$ is the probability of that platform being destroyed by the attack and is given by equation 3 below.

$$D(E) = S(A) * E(W) \quad (3)$$

Here, $S(A)$ is the attackers probability of survival up until the time of the attack and $E(W)$ is the effectiveness of the attackers weapon as given in the weapon-entity effectiveness matrix. This method gives us the expected values of survival for all entities in the game within one run of the game, thereby producing a representative evaluation of the value of a plan. As a side effect, we also gain a smoother gradient for the GA to search as well as consistently reproducible evaluations.

3.3 Encoding

To play the game, players get the starting scenario and have some initialization time to prepare strategy. GAP applies the case injected genetic algorithm to the underlying resource allocation and routing problem and chooses the best plan to play against Red. The game then begins. During the game, Red can activate popup threats detectable upon activation by GAP. GAP then runs the case-injected genetic algorithm producing a new plan of action, and so on.

GAP must produce routing data for each of Blue's platforms. We use the A* algorithm [32] to build routes between locations platforms wish to visit. A* finds the cheapest route, where cost is a function of route length and route risk.

We parameterize the routing algorithm in order to produce routes with specific characteristics and allow GAP to tune this parameter. For example, we have shown that to avoid traps, GAP must be able to specify that it wants to avoid areas of potential danger. In our game, traps are most effective in areas confined by other threats. If we artificially inflate threat radii, threats expand to fill in potential trap corridors and A* produces routes that go around these expanded threats. We thus introduce a parameter, rc that increase threats' effective radii. Larger rc's expand threats and fill in confined areas, smaller rc's lead to more direct routes. rc is currently limited to the range $[0, 3]$ and encoded with eight (8) bits at the end of our chromosome. We encoded a single rc for each plan but are investigating the encoding of rc's for each section of a route.

Most of the encoding specifies the asset to target allocation with rc encoded at the end as described above. Figure 4 shows how we represent the allocation data as an enumeration of assets to targets. The scenario involves two platforms (P1, P2), each with a pair of assets, attacking four targets. The left box illustrates the allocation of asset A1 on platform P1 to target T3, asset A2 to target T1 and so on. Tabulating the asset to target allocation gives the table in the center. Letting the position denote the asset and reducing the target id to binary then produces a binary string representation for the allocation.

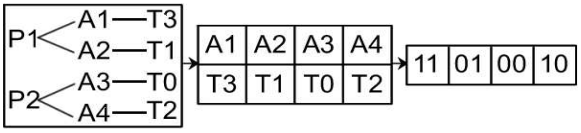


Figure 4: Allocation Encoding

4 Learning from Experience

Previous work has taught us that we can play the game with a GA [30, 31]. Our goal has since become to play the game faster and better. Even on a computer cluster the time required to respond to a change in state is slow compared to the fast pace of the game. We achieve our goal of playing faster and better through case injection.

Case-injected GA's have been shown to increase performance at similar problems as they gain experience. In strategic games such situations occur often. Every opponent action and situational change is a change in game state. However, these changes are often minor as few opponent actions are worth redeveloping your entire strategy. Case injection maintains some information from previous strategy development, allowing us to keep our current strategy in mind when developing new ones in response to changes in game state. Case injection thus allows the GA to adapt old strategies to new situations, maintaining past knowledge that is still applicable, and redeveloping new strategies as unforeseen situations arise.

5 Results

In this work we explore how we can improve genetic algorithm performance through case injection. We analyze how case-injection impacts re-planning with respect to game complexity and re-planning scope. Except for the large mission (below), we used a steady-state population size of 20 run for 20 generations with a single-point crossover probability of 0.9 and a mutation probability of 0.1. We injected two individuals every generation chosen using a probabilistic closest to the best strategy where the probability of being picked for injection from the case-base was directly proportional to hamming similarity. The large mission used a population size of 40 run for 40 generations. The algorithm stopped either when it exceeded the set number of iterations (20 or 40) or when there was no improvement for over ten generations or when f_{max}/f_{avg} was less than 1.01. Here f_{max} is the population maximum fitness and f_{avg} is the population average fitness.

5.1 Game Complexity

Game complexity refers to the complexity of the individual mission being played. Increasing the resources available for each side to allocate increases the strategic search space presented to each player. How does this increase in search space alter the effect of case injection on the genetic search?

To test this, we first constructed 3 missions of increasing complexity. More complex missions have more attacking aircraft loaded with more weapons to attack more targets. The mission's general character does not change. The defending player follows a script activating popup threats early in the mission leading the attacking player (GAP) to replan attacking strategy. Scripting keeps the analysis straightforward and repeatable. We then let the GA play this game multiple times, with and without case injection and analyze the GA's response. We provide mission profiles below

- Simple Mission
  - 4 platforms
  - 8 assets
  - 8 targets
  - 30 bits per chromosome
- Medium Mission
  - 6 platforms
  - 12 assets
  - 20 targets
  - 66 bits per chromosome
- Large Mission (population size 40)
  - 10 platforms

- 20 assets
- 40 targets
- 114 bits per chromosome

We run a GA with and without case injection on the three missions ten time with different random seeds and plot the average number of evaluations made in Figure 5. Note the **C**ase-**I**njected **G**enetic **A**lgo**R**ithm (CIGAR) greatly reduces the number of evaluations (time) to converge outperforming the non-injected GA, especially as the mission becomes more complex. On more complicated missions CIGAR retains more information, giving it a larger advantage over the GA.

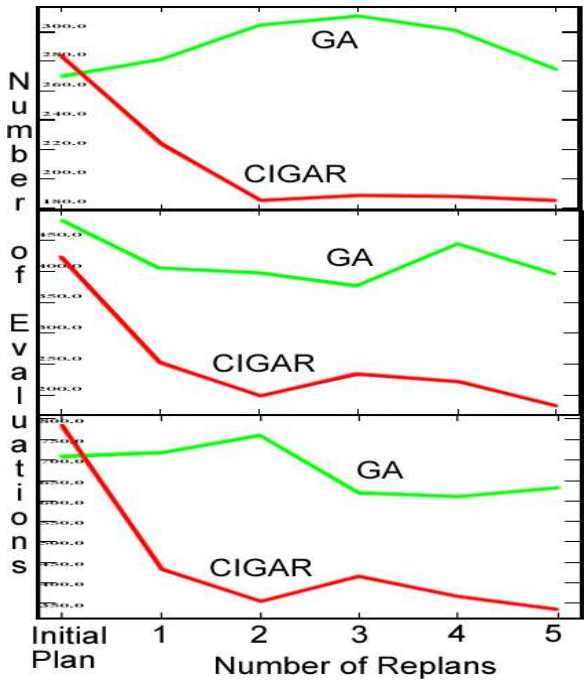


Figure 5: Mission Complexity — Evaluations Made

We can also see that although CIGAR takes less time to converge, the quality of solutions produced by CIGAR does not suffer. Figure 6 plots the average of the maximum fitness found by the GA or CIGAR over ten runs and shows that CIGAR seems to produce better quality solutions; although we need to have more runs to prove the improvement is statistical significant.

5.2 Replanning Scope

GAP re-plans whenever the situation changes. These changes range from minor events like discovering a poorly valued target to big events like highly time-critical and important targets appearing. Case-injection exploits information gained in previous searches, and the scope of the situation change determines how much of that previous knowledge is pertinent to the current situation. How does case-injection work under these different kinds of changes?

We again construct three missions, this time with different defending layouts and scripted actions for the defending player. In each mission the defending player makes five

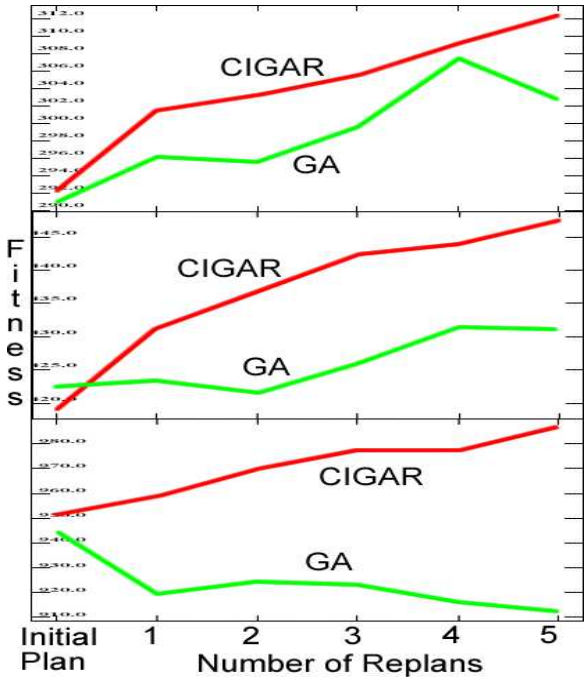


Figure 6: Mission Complexity — Fitness of Solution

changes, these changes having an increasing impact on the attacking players strategy. We summarize the missions and GAP's response below.

- Simple Replan - Small threats popup on the way to targets (same as previous 3 missions).
  - GA reroutes to avoid new threats
  - Minor changes to weapon-target allocation
- Moderate Replan - Medium threats popup around a handful of targets
  - Moderate changes in allocation
  - Avoids newly protected low value threats
  - Redirects additional attackers to newly protected high value targets
  - Significant routing changes to avoid new threats and reach new targets
- Complex Replan - Large popups occur defending a large cluster of targets
  - Large changes of allocations
  - Wings (groups) of aircraft diverted from new hot zones
  - Focusing of aircraft towards the most highly valued targets
  - Rerouting of most aircraft for each replan

Figure 7 shows the number of evaluations required as a function of the number of re-plans for the above missions. As the scope of the replan increases, case-injection's advantage decreases. In other words, case-injection focuses

search towards previously successful solutions, as the new solution moves further from the old solution the advantage provided by case-injection decreases. Figure 8 shows once again that CIGAR's speed advantage does not come at the expense of lower quality solutions. On the contrary the figure shows that CIGAR produces better quality plans.

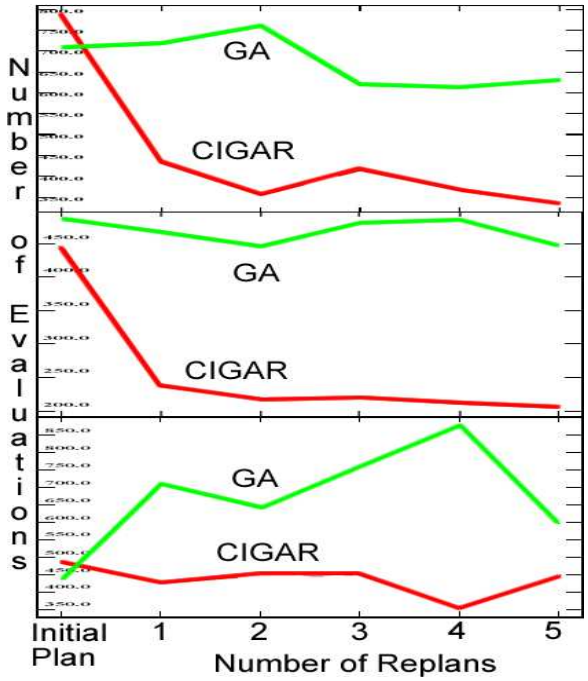


Figure 7: Replanning Size — Evaluations Taken

5.3 Statistical Significance

CIGAR's speedup over the GA **is** statistically significant. Using CIGAR, we can expect significant speedup with no loss in quality of solution produced. In fact, our results lead us to believe that we should also expect better quality solutions. These results fit well with previous work that shows case-injected genetic algorithms more quickly delivering better quality solutions [6].

6 Conclusions and Future Work

We have shown that a genetic algorithm can play computer strategy games by solving the sequence of underlying resource allocation problems. Case-injection statistically significantly improved the speed with which our case-injected genetic algorithm player (GAP) responds to opponent actions and other changes, while improving the fitness of solutions produced. We explored the effects of mission complexity and replanning scope, showing that the advantage provided by case injection increases as the mission becomes more complicated, and decreases as the difference between new and old situations grows. Note that case injection still provides a significant improvement even when the game situation changes drastically. Playing RTS games with a GA presents a good application of case injection, and we have explored how case-injection impacts the dynamics of the

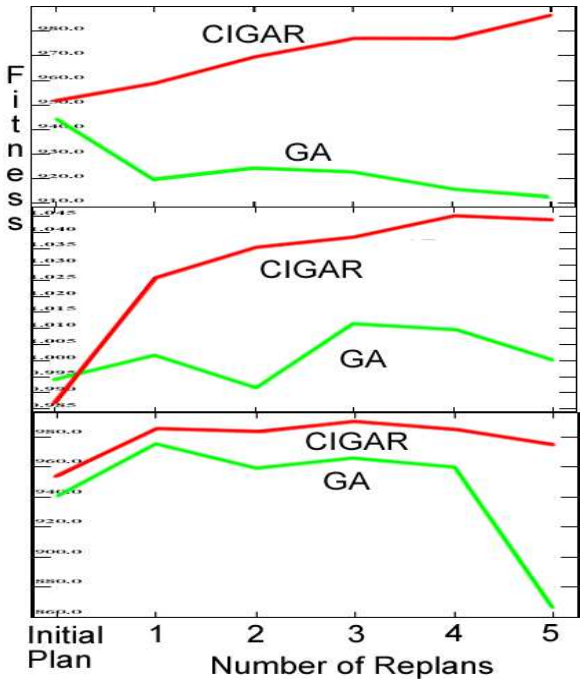


Figure 8: Replanning Size — Fitness of Solution

game, showing significant improvement in response time as well as some improvement in response quality.

We are exploring several avenues for further improving response times. One possibility is to use a reduced surrogate to quickly approximate fitness evaluations. Finally, the game itself is several stages from being human playable and we are currently working on making the game much more playable.

Acknowledgments

This material is based upon work supported by the Office of Naval Research under contract number N00014-03-1-0104.

Bibliography

[1] Angeline, P.J., Pollack, J.B.: Competitive environments evolve better solutions for complex tasks. In: Proceedings of the 5th International Conference on Genetic Algorithms (GA-93). (1993) 264–270

[2] Fogel, D.B.: Blondie24: Playing at the Edge of AI. Morgan Kauffman (2001)

[3] Samuel, A.L.: Some studies in machine learning using the game of checkers. IBM Journal of Research and Development **3** (1959) 210–229

[4] Pollack, J.B., Blair, A.D., Land, M.: Coevolution of a backgammon player. In Langton, C.G., Shimohara, K., eds.: Artificial Life V: Proc. of the Fifth Int. Workshop on the Synthesis and Simulation of Living Systems, Cambridge, MA, The MIT Press (1997) 92–98

[5] Tesauro, G.: Temporal difference learning and td-gammon. Communications of the ACM **38** (1995)

[6] Louis, S.J., McDonnell, J.: Learning with case injected genetic algorithms. IEEE Transactions on Evolutionary Computation (To Appear in 2004)

[7] Riesbeck, C.K., Schank, R.C.: Inside Case-Based Reasoning. Lawrence Erlbaum Associates, Cambridge, MA (1989)

[8] Schank, R.C.: Dynamic Memory: A Theory of Reminding and Learning in Computers and People. Cambridge University Press, Cambridge, MA (1982)

[9] Leake, D.B.: Case-Based Reasoning: Experiences, Lessons, and Future Directions. AAAI/MIT Press, Menlo Park, CA (1996)

[10] Goldberg, D.E.: Genetic Algorithms in Search, Optimization, and Machine Learning. Addison-Wesley (1989)

[11] Louis, S.J.: Evolutionary learning from experience. Journal of Engineering Optimization (To Appear in 2004)

[12] Louis, S.J.: Genetic learning for combinational logic design. Journal of Soft Computing (To Appear in 2004)

[13] Louis, S.J.: Learning from experience: Case injected genetic algorithm design of combinational logic circuits. In: Proceedings of the Fifth International Conference on Adaptive Computing in Design and Manufacturing, Springer-Verlag (2002) to appear

[14] Louis, S.J., Johnson, J.: Solving similar problems using genetic algorithms and case-based memory. In: Proceedings of the Seventh International Conference on Genetic Algorithms, Morgan Kauffman, San Mateo, CA (1997) 283–290

[15] Griggs, B.J., Parnell, G.S., Lemkuhl, L.J.: An air mission planning algorithm using decision analysis and mixed integer programming. Operations Research **45** (Sep-Oct 1997) 662–676

[16] Li, V.C.W., Curry, G.L., Boyd, E.A.: Strike force allocation with defender suppression. Technical report, Industrial Engineering Department, Texas A&M University (1997)

[17] Yost, K.A.: A survey and description of usaf conventional munitions allocation models. Technical report, Office of Aerospace Studies, Kirtland AFB (Feb 1995)

[18] Louis, S.J., McDonnell, J., Gizzi, N.: Dynamic strike force asset allocation using genetic algorithms and case-based reasoning. In: Proceedings of the Sixth Conference on Systemics, Cybernetics, and Informatics. Orlando. (2002) 855–861

[19] Rosin, C.D., Belew, R.K.: Methods for competitive co-evolution: Finding opponents worth beating. In Eshelman, L., ed.: Proceedings of the Sixth International Conference on Genetic Algorithms, San Francisco, CA, Morgan Kaufmann (1995) 373–380

[20] Kendall, G., Willdig, M.: An investigation of an adaptive poker player. In: Australian Joint Conference on Artificial Intelligence. (2001) 189–200

[21] Blizzard: Starcraft (1998, www.blizzard.com/starcraft)

[22] Cavedog: Total annihilation (1997, www.cavedog.com/totala)

[23] Inc., R.E.: Homeworld (1999, homeworld.sierra.com/hw)

[24] Laird, J.E.: Research in human-level ai using computer games. Communications of the ACM **45** (2002) 32–35

[25] Laird, J.E., van Lent, M.: The role of ai in computer game genres (2000)

[26] Laird, J.E., van Lent, M.: Human-level ai's killer application: Interactive computer games (2000)

[27] Tidhar, G., Heinze, C., Selvestrel, M.C.: Flying together: Modelling air mission teams. Applied Intelligence **8** (1998) 195–218

[28] Serena, G.M.: The challenge of whole air mission modeling (1995)

[29] McIlroy, D., Heinze, C.: Air combat tactics implementation in the smart whole air mission model. In: Proceedings of the First Internation SimTecT Conference, Melbourne, Australia, 1996. (1996)

[30] Miles, C., Louis, S.J., Drewes, R.: Trap avoidance in strategic computer game playing with case injected genetic algorithms. In: Proceedings of the 2004 Genetic and Evolutionary Computing Conference (GECCO 2004), Seattle, WA. (2004) 1365–1376

[31] Miles, C., Louis, S.J., Cole, N., McDonnell, J.: Learning to play like a human: Case injected genetic algorithms for strategic computer gaming. In: Proceedings of the International Congress on Evolutionary Computation, Portland, Oregon, IEEE Press (2004) 1441–1448

[32] Stout, B.: The basics of a* for path planning. In: Game Programming Gems, Charles River media (2000) 254–262

A Hybrid AI System for Agent Adaptation in a First Person Shooter

Michael Burkey, Abdennour El Rhalibi
School of Computing and Mathematical Sciences
Liverpool John Moores University,
Liverpool
a.elrhalibi@livjm.ac.uk

Abstract- *The aim of developing an agent that is able to adapt its actions in response to their effectiveness within the game provides the basis for the research presented in this paper. It investigates how adaptation can be applied through the use of a hybrid of AI technologies. The system developed uses the pre-defined behaviours of a finite state machine and fuzzy logic system combined with the learning capabilities of a neural network. The system adapts specific behaviours that are central to the performance of the bot in the game, with the main focus being on the weapon selection behaviour; selecting the best weapon for the current situation. As a development platform, the project makes use of the Quake 3 Arena engine, modifying the original bot AI to integrate the adaptive technologies.*

1 Introduction

With graphics at an almost photo-realistic level and complex physics systems becoming commonplace, AI is becoming more important in providing realism in games. In the past, game AI has used techniques that are suited to the restricted computational power available to it, but which still produced believable, but limited, non-player characters (NPC's) – AI technologies such as Finite State Machines (FSM) and Rule Based Systems (RBS). These techniques were also used due to their relative simplicity which did not require much development time to implement and were easy to debug, especially as the programmers generally didn't specialise in AI.

With the increase in computational power available for AI, more complex techniques can be incorporated into games creating more complex behaviours for NPC's. The increasing importance of AI in games has meant that specialised AI programmers are becoming part of development teams bringing techniques from academia [Laird, 2000]. One of the areas of AI which has gathered interest is that of using machine learning techniques to create more complex NPC behaviours.

Most players develop styles of play that take advantage of certain weaknesses inherent in the NPC AI that become apparent as they become more proficient at the game. Once discovered, these deficiencies in the pre-programmed AI mean the competitive edge is lost making the player lose interest in the now all too easy game. If the NPC developed new tactics, adapting to the players style, uncovered their hiding places or even discovered tactics that exploited weaknesses in the players' play then this would add immeasurably to the enjoyment and prolong the life of the game [Palmer, 2003].

This paper describes a method of implementing a first person shooter (FPS) bot, a computer controlled player that imitates a human adversary, which uses machine learning to adapt its behaviour to the playing style of its opponent. It uses a combination of small, focussed, artificial neural networks and pre-defined behaviours that allow the bot to exhibit changes in those behaviours to compensate for different player styles. For the purposes of this paper, only a single behaviour is focused on, that of weapon selection, although it could be used for other behaviours and even other genres of game.

2 Foundation Technologies

2.1 Fuzzy Logic

Whereas traditional logic describes concepts in terms of 'true' or 'false', fuzzy logic provides a way of describing values by the degree with which they correspond to a certain category within the concept, called the *degree of membership* in a *set*. *Linguistic variables* are collections of sets that represent real concepts, for example the variable *health* could be made up of the sets Near Death, Good and Excellent, as shown in **Figure 1** [Zarozinski, 2001].

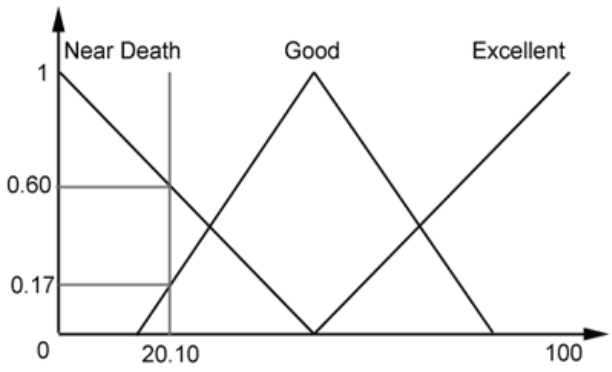


Figure 1: Sets Defining the Linguistic Variable 'Health'

Fuzzy logic provides a way of combining more than one variable to give a single output value, making decisions based on multiple criteria. For example, the aggression of a game character based on its health and the distance to the enemy.

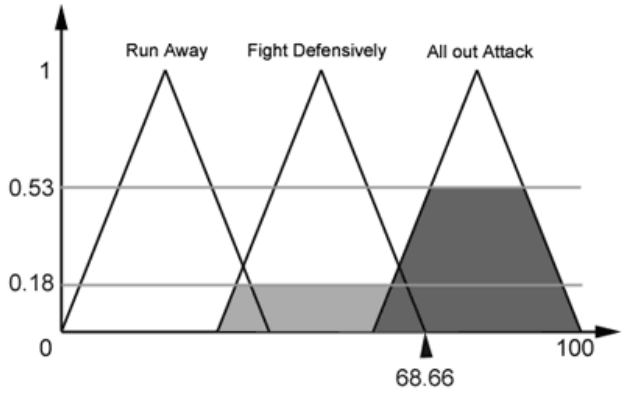


Figure 2: Sets Defining the Output Variable 'Aggressiveness'

Using fuzzy logic to derive decisions based on the input values for a number of variables requires that a sequence of steps be carried out.

1. Selection of sets that comprise the linguistic variables for the inputs and output. As with the input variables, the output variable consists of a number of sets defined by a range of values (see Figure 2). The difference is in the way they are used to calculate the final out value (see step 6, *Defuzzification*).
2. Creation of fuzzy rules corresponding to the different combinations of inputs. The rules determine the output set for the different combinations of inputs. Using the previous example, if health is 'Good' and distance is 'Close', the rule could be 'Fight Defensively'.
3. *Fuzzification* of the crisp inputs into fuzzy values giving the degree of membership for the inputs sets. Figure 1 shows the fuzzification of the crisp value 20.1 resulting in the DOM for each set of Near Death=0.6, Good=0.17 and Excellent=0.0.
4. Use *inference* to evaluate which rules are active based on the degree of membership of the input sets that make up that rule. Each combination of sets (for each input variable) is compared with the rulebase to determine which output sets are active. The DOM of the output set is determined, in this case, using the lowest DOM of the inputs (there are a number of methods for calculating the DOM). This results in a number of possible DOMs for each set of the ouput variable.
5. Combine the multiple DOMs for each rule into the output sets using *composition*. This results in a single DOM for each of the output sets, as shown in Figure 2.
6. *Defuzzification* of the output sets to give a single crisp value. This is done by calculating the centre of the area under the graph defined by the degree of membership in each set. Figure 2 shows an example for the output variable 'Aggressiveness' with DOMs of 0.18 for 'Fight Defensively' and 0.53 for 'All out Attack'. There are a number of methods, of varying complexity and accuracy, for determining the output value. One of the least expensive, in terms of computation, is the *mean of maximum* method, the equation for which is shown in **Equation 1**.

$$(RA_{max} * RA_{dom} + FD_{max} * FD_{dom} + AA_{max} * AA_{dom}) / (RA_{dom} + FD_{dom} + AA_{dom})$$

where,

- RA_{max} – crisp value for centre of 'Run Away' set (where fuzzy value =1).
- RA_{dom} – fuzzy value for degree of membership for Run Away set.
- FD_{max} and FD_{dom} – as above for 'Fight Defensively' set.
- AA_{max} and AA_{dom} – as above for 'All-out Attack' set.

Equation 1: Calculation of Mean of Maximum

2.1.1 Combs Method

In traditional fuzzy logic a rule needs to be defined for every combination of set for all the input variables. This can result in combinatorial explosion as the number of rules required grows exponentially according to the number of fuzzy sets for each linguistic variable, e.g. 2 variables each with 5 sets = 5^2 = 25 rules and 5 variables with 5 sets = 5^5 = 3,125 rules. This can make large systems slow, confusing and difficult to maintain which, particularly in games, can make fuzzy logic impractical [Zarozinski, 2001].

The main difference between Combs method and the traditional method is in the way the rule-set is defined. It builds rules based on each individual set's relationship to the output, considering one variable at a time, rather than creating rules for every combination of set for all the variables. This reduces the exponential growth of the number of rules into a linear growth, so that a system with 10 variables and 5 sets per variable would have 50 rules as opposed to 9,765,625 with the traditional system.

2.2 Artificial Neural Networks

There are many forms of artificial neural nets (ANN) of varying complexity which attempt to mimic the biological operation of the brain artificially by modelling the inter-connected cells that enable the brain to process information. The simplest form of ANN, the one used here, is the Perceptron which is modelled as a single neuron with a set of weighted inputs mapping to a single output [Champandard, 2004].

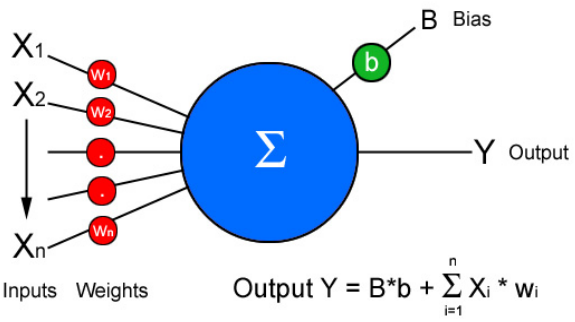


Figure 3: Architecture of a Perceptron

The inputs (X_1 to X_n) to the perceptron can vary in number and value (binary or real numbers) depending on the application. Each input is multiplied by its corresponding weight (W_1 to W_n) and the weighted inputs are then added together, along with the bias, giving the output value. The bias represents a constant offset and can be treated as another input with a constant value of 1. By adjusting its weights the perceptron can be trained to recognise specific combinations of inputs and generalise for similar inputs.

2.2.1 Training the Perceptron

Initially, the perceptrons use a default value for all of their weights. This, in effect, means that the perceptrons will not have any influence over the effectiveness rating for the weapons, only the characteristics and fuzzy logic will affect the value. Once adaptation has begun, occurring every time there is feedback, the following training procedure is performed.

The training of perceptrons described here uses an incremental approach, computing the adjustments to the weights by way of the steepest descent technique [Champandard, 2004]. The *delta rule* algorithm calculates the change required Δw_i for each weight w_i by taking the difference between the actual y and the desired t output and multiplying it by the input value x_i for that weight and by a, typically small, learning rate η (see Equation 2).

$$\Delta w_i = \eta(t-y)x_i$$

Equation 2: Computation of Required Adjustment for Each Weight

The new weight for each input can then be found using the *steepest descent technique*, as shown in Equation 3, changing the weights as result of feedback.

$$w_i \leftarrow w_i + \Delta w_i$$

Equation 3: Computation of Adjusted Weight Value

The incremental nature of the algorithm means that it can be performed as the game is being played using feedback from actions performed.

2.3 Quake 3 Arena

In order to implement the adaptable AI a suitable environment was required that provided all the features of a FPS so that the capabilities of the bot can be tested. The Quake 3 Arena (Q3A) game engine [Id Software, 1999] provided the framework for the development of the bot AI. The new AI was integrated with the original AI, reusing many of its features. For more information regarding the Q3A engine, specifically in relation to the interface between the AI and the game engine, [van Waveren, 2003] provides the most comprehensive documentation.

Using the original bot AI provided the opportunity to be able to define the characteristics of the bot using text files that determine the style of play of the bots within the game. This proved helpful in the evaluation of the new AI, which was carried out in matches against the 'standard' Q3A bots. By defining specific characteristics, situations could be set up that required the bot to adapt its behaviour.

3 System Design

A number of features are required of the adaptable AI system in order to achieve the aim of a bot that is able to adapt to the play of an adversary:

- Be able to play competitively from the first game (out of the box).
- Adapt its behaviour as the game is being played (online).
- Be computationally inexpensive.

The system makes use of the indirect adaptation technique, using a conventional AI layer to control the bot, with the adaptation AI modifying the behaviours of the bot in response to feedback according to its actions. This enables the bot to be competitive immediately by giving it *a priori* knowledge, as recommended by [Manslow, 2001].

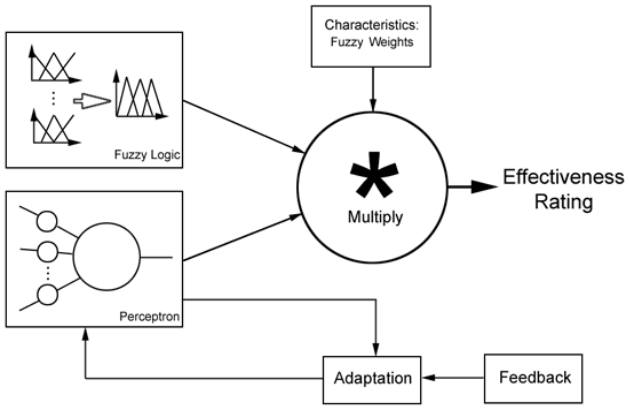


Figure 4: Adaptation System Overview

The adaptation system incorporates a number of components that combine to rate the effectiveness of a choice within a behaviour and adapt the value to reflect how well the chosen action performs in the game. Figure

4 shows how the separate elements are linked together to calculate the rating and allow adaptation to occur.

The system utilises a hybrid of 2 AI technologies – fuzzy logic and perceptrons. The fuzzy logic acts as the prior knowledge enabling the bot to perform in the game at a competitive level. The perceptron is used to facilitate the adaptation, acting as a form of memory enabling the bot to 'remember' the effectiveness of actions in certain situations, altering its weights based on the feedback it receives from game. By using perceptrons, rather than more complex multi-layer networks, the computational requirements are kept as low as possible whilst retaining the basic features of a neural net.

The system is composed of 2 main mechanisms:

- The Effectiveness Rating Mechanism – used to determine how effective a certain choice is according to the input values.
- The Adaptation Mechanism – used to change the effectiveness rating according to feedback from the game on how effective it was.

The effectiveness of a choice is predicted using a combination of the characteristics of the bot, defined in the characteristic files, a fuzzy logic component and a perceptron component. This system is used for each of the choices within a behaviour. The effectiveness is calculated by multiplying the outputs from the fuzzy logic component and the perceptron together with the characteristic for the choice

The adaptation mechanism uses feedback from the game to determine how successful the choice was compared to the perceptron's predicted effectiveness of the choice. The feedback and output of the perceptron are then used to train the perceptron, increasing or decreasing the weight values according to the delta rule training algorithm discussed in section 2.2.1. Adjusting the weights of the perceptron changes its output impacting on the effectiveness rating for the action, thus making it more or less likely to be used.

3.1 Adaptation of Weapon Selection Behaviour

Modern FPS games, such as Quake 3 Arena, make use of complex 3D environments for their game worlds which, in turn, means that the NPC's that inhabit them must have complex AI to interact with them, and the player, convincingly. Bots must be able to exhibit a number of behaviours, specialising in particular actions or strategies that contribute to the overall aim of winning the game. Due to the nature of the game, the aim being to kill the opponent more times than they kill you, the behaviours that would benefit most from adaptation are those that relate to combat with opponents, either directly or indirectly. One such behaviour is that of selecting the most effective weapon for the current situation. The rest of the paper will focus on this behaviour to demonstrate how the system can be applied.

The aim of adapting the selection of weapons is to enable the bot to change its weapon preferences depending on its success in particular situations. By changing the 'effectiveness' or 'fitness' of each weapon, by way of changing the perceptron weights according to the input values, different play styles can be adapted to.

The selection of information used as inputs for the system components is vital to their efficiency at performing actions in the game. The following sections detail the inputs for the fuzzy logic and perceptron components.

3.1.1 Fuzzy Logic for Weapon Selection

Each of the weapons have a set of data defined for the variables (inputs) that represent the range of values that are significant to that weapon. The variables used for the fuzzy logic component are:

- Distance to the enemy – each of the weapons available is better or worse at different distances. For example, the Lightning Gun has a maximum range of 768 and the Rocket Launcher risks splash damage when used at close distance. The distance needs to be broken down into fuzzy sets defining the effectiveness of each weapon for the distance range represented by that set.
- Ammunition amount for each weapon – each of the weapons have different firing rates. For example, the Machine Gun fires a shot every $1/10^{th}$ of a second whilst the Railgun can only fire a shot every 1 ½ seconds. Running out of ammo in a fight means changing to another weapon, which takes time, reducing the damage that can be inflicted on the enemy. The ammo level needs to be represented as a number of fuzzy sets spanning the maximum amount of ammo (200). Each weapon requires a unique collection of set data defining the relative ranges of ammo depending on their rates of fire – 10 ammo for the Railgun is different to the same amount for the Machine Gun.

3.1.2 Perceptron for Weapon Selection

Each weapon is represented by a perceptron, each having a unique set of weight values for that weapon. The inputs to the perceptron are the same for each of the weapons, although weapon specific inputs, i.e. the amount of ammo, will result in certain inputs having slightly different values. Some of the variables investigated are:

- Distance to the enemy – by adapting the distance at which the weapon should be used the weapons will increase/decrease the range at which they are used. An example of a use for this is if the enemy is very aggressive and continues attacking when low on health. Normally the Rocket Launcher may not be used at close range due to the danger of splash damage, selecting a less damaging, and less successful, weapon instead. The system could adapt the lower range of the Rocket Launcher so that it is selected over the less useful weapon, incurring damage to the bot but also killing the opponent with one shot.
- Ammo – the amount of ammo for each weapon can be adapted to make use of weapons that the opponent is more susceptible to being damaged by.

Used by the fuzzy logic component, it has a large influence on the selection of weapons and by adjusting the ranges the bot will be more likely to stick with a successful weapon even though the ammo is running low in the hope of killing them before the weapon needs to be switched.

- Visibility of enemy – it would be useful to adapt the weapon selection based on the visibility of the enemy so that areas that contain obstacles, creating cover for the enemy to hide behind, can influence the selection to favour weapons that have splash damage enabling the weapon to inflict damage around corners.
- Height difference – like the visibility of the enemy, the height difference between the bot and its opponent could be used to influence the use of weapons that have splash damage. If the opponent is below the bot it can aim at the floor near to the enemy, hitting with radial damage. If the opponent is higher, making it difficult to hit them with splash damage then Grenades can be launch onto the higher area or more precise weapons used. Adapting the relative strengths of weapons when there is a height difference will select the most effective weapons in those situations.

3.2 Feedback for Perceptron Training

The feedback that is used to train the perceptrons for the weapon selection behaviour is focused on the criteria of causing as much damage as possible whilst avoiding inflicting damage to oneself. This means that it must account for a combination of health lost by the enemy and by the bot itself as a result of its own attack (not damage sustained from enemy attack). A timed aspect is required to allow for the different characteristics of each of the weapons (firing rate and damage per shot) and enable the performance of the weapons to be compared. To reward weapons that have the capability of 'finishing off' enemies (for example Railguns are very good at one-shot kills) a bonus is also required when the opponent is killed by the current weapon. This increases the overall feedback value thus increasing the weight values when training.

3.3 Categorisation of Perceptron Inputs

Due to the linear nature of perceptrons (they are unable to handle non-linear problems) difficulties arise with inputs that can be effective at high and/or low values. One problem is that higher input values will always output higher ratings and so if the lower values are better they will not be able to represent it. Another problem is if the weapon is more effective with an input value that is in the middle range, such as the Grenade Launcher that can cause splash damage close up but has a limited range. This is compounded by the training mechanism that changes the weight of the input depending on the input value. This means that high values will always be penalised more than low values.

To allow adaptation to occur independently for different levels of the same input, its range of values needs to be categorised into ranges. The fuzzy logic component can be utilised to achieve this. It is able to take a single value and assign a degree of membership for each of the categories by fuzzifying the input value. Each of the categories represents an input into the perceptron, splitting the single input value into the number of sets that represents that input, as shown in Figure 5. The advantage of this approach is that it will categorise the input into continuous values for each set, rather than the imprecise method of just determining whether the value is in a category or not. It also uses functionality that is already within the system so no new component needs to be developed.

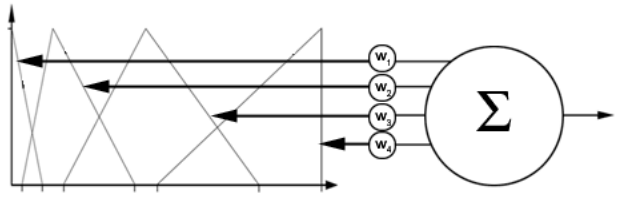


Figure 5: Categorisation of perceptron inputs

One of the main advantages with using fuzzy logic to categorise the input value is that the fuzzy values will represent the degree of membership for the set. This means that the low category can have a high input value and the high a low value – 0 (100% membership of the low category) could input a 1. When training the perceptron this will be useful in correctly rewarding or punishing the value range responsible for the action selected. Another advantage is that the maximum membership of a set is 100%, in effect normalising the input values for each set to a value between 0 and 1. Although the input value can be in multiple sets, the combined fuzzy values will approximate 1 (fuzzy values needn't add up to 1 but are usually near to this value, depending on the set data).

4 Evaluation

For the purposes of testing the adaptation system, the fuzzy logic component was designed to mimic the selections made by the original Q3A AI as closely as possible. This was done so that the changes in behaviour of the bot due to adaptation during a match could be directly compared with the behaviour exhibited by the original AI.

4.1 Adaptation of Weapon Selection

In order to test whether the bot is able to change weapon preferences within the game its preferences were set up so that it had a high preference for a certain weapon but also low accuracy. By assigning another weapon a high accuracy but normal preference, the system's ability to change preferences was tested. The adaptable bot's preferences and accuracy levels were set up as follows:

- **Plasma Gun**: Accuracy=0.1, Preference=300.
- **Rocket Launcher**: Accuracy=0.9, Preference=200.

- **Grenade Launcher**: Accuracy=0.8, Preference=100.
- **Shotgun**: Accuracy=0.7, Preference=150.

Figure 6 shows the output of the weapon selection choices, comparing the Q3A AI with the adaptable AI. The graph shows how the adaptable AI and Q3A AI make very similar selections at the start of the match, with only slight variations in the choice of weapon. Towards the end of the match the differences of choice become more evident with regards to the Plasma Gun (weapon 8) in particular; seen clearly in Figure 7 which shows a close up of the last part of the match.

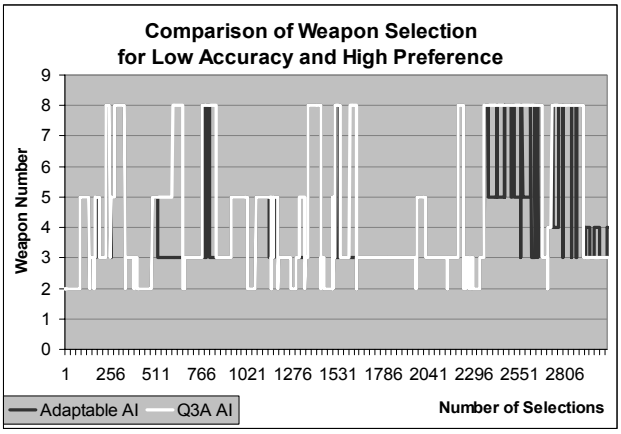


Figure 6: Graph of Weapon Selection Comparison between Adaptable AI and Q3A AI for Low Accuracy and High Preference of Plasma Gun

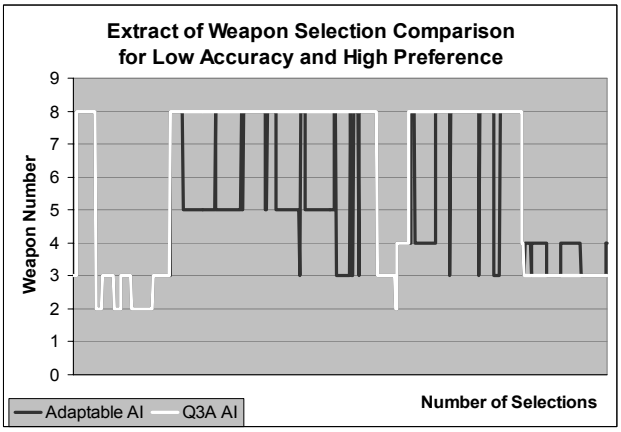


Figure 7: Extract of Comparison of Weapon Selection for Low Accuracy and High Preference, showing Difference Due to Adaptation

This clearly demonstrates the adaptation occurring on the Plasma Gun's use as, due to its very low accuracy, the negative feedback lowers its effectiveness rating over the course of the match until the other weapons effectiveness scores make them a better choice. The rating of the Plasma Gun falls so low that the Grenade Launcher (4), with only a third the preference rating of the Plasma Gun, is preferred over it in some situations. The Rocket Launcher (5) is shown to be preferred to the Plasma Gun in almost all situations, and those times when it is not can be accounted for by the Rocket Launcher running out of ammo.

The graphs showing the adaptation of the perceptrons for the Plasma Gun (Figure 8) and the Rocket Launcher (Figure 9) illustrate how rating of the Plasma Gun drops and the Rocket Launcher rises to a point were the preferences change for the weapons. Whereas, the Plasma Gun's *medium* range drops to around 0.2, the Rocket Launcher's rises, albeit only slightly, to 0.55. This is enough of a variation to cause the change in weapon selection to occur.

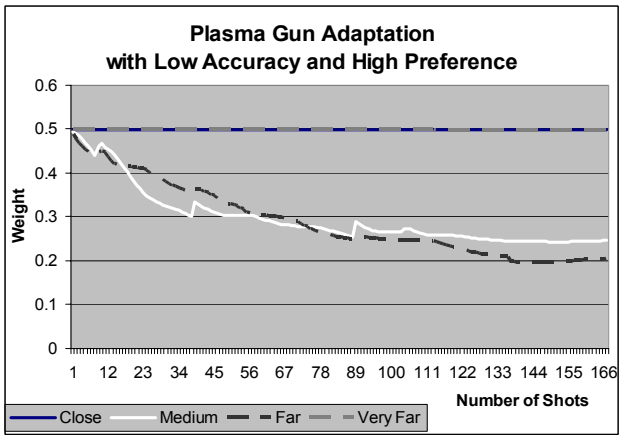


Figure 8: Adaptation of Plasma Gun Distance Due to Low Accuracy and High Preference

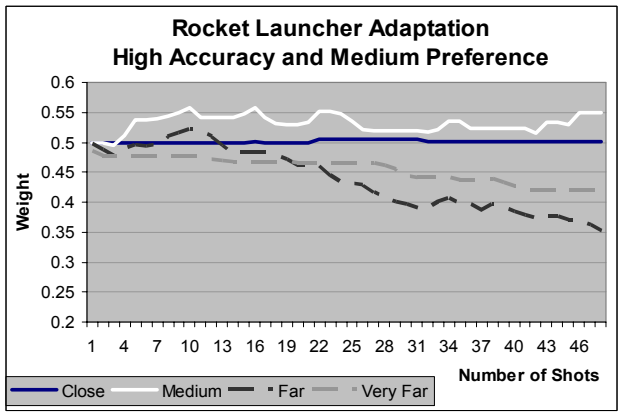


Figure 9: Adaptation of Rocket Launcher Distance Due to High Accuracy and Medium Preference

4.2 Validity of Input Choices for Perceptron

The inputs chosen for the perceptron resulted in varying degrees of success in their ability to affect the selection of weapons due to adaptation. The distance input was successful in reflecting the feedback of the weapon's strengths and weaknesses in the adaptation of its weights. Trends can be identified from the adjustments made during training that relate to the performance of the weapon in the game.

Another input that demonstrated an effect on the selection process was the height difference, although not to the extent of the distance input. It showed a higher

effectiveness for when the enemy is below the bot and lower for enemies above.

The ammo input showed little influence over determining the correct weapons by adapting its values. This is caused by there being no direct link between the effectiveness of the weapon and the amount of ammunition, therefore the feedback could not influence the ammunition training. The only direct influence of the ammo on the weapon selection came when the level fell to 0, causing the weapon to be changed to another. All other levels had no bearing on how the weapon performed, indicating that categorisation was not required. Possibly, restricting the ammo input to a 'low ammo' input would better serve the selection of weapons.

The evaluation of the inputs shows that certain types of input lead to better performance of the adaptation of the perceptron while others contribute little. Generally, the most effective inputs:

- Directly influence the behaviour – the ammo input had no direct influence over the effectiveness of the weapon, whereas the distance changed how well it performed.
- Are reflected in the feedback – there was no feedback that reflected the effect of ammo at levels other than 0, when the weapon needed to change. The amount of damage that could be inflicted was not affected by larger or smaller amounts of ammunition.

5 Conclusions

This paper has shown how, by combining traditional AI techniques, a system can be developed that enables the choices made by a conventional AI layer to be altered in response to feedback from the actions selected. Although the development was limited to just the weapon selection behaviour, so limiting the effect on the game that is visible from testing, evidence was found that the system is capable of adapting to feedback by a significant enough amount to change the actions that prove unsuccessful to those that are successful. The results showed interesting trends that indicate that, with more development and testing to determine optimum settings, the system developed could form the basis of a useable adaptable AI system.

References

[Laird, 2000] Laird, J. E., "Game AI: The State of the Industry 2000, part two" *Game Developer* magazine, August 2000.

[Palmer, 2003] Palmer, N., "Machine Learning in Games Development", 2003, http://ai-depot.com/GameAI/Learning.html.

[Champandard, 2004] Champandard, A.J., "AI Game Development: Synthetic Creatures with Learning and Reactive Behaviours", New Riders Publishing, Indianapolis, Indiana, 2004.

[Manslow, 2001] Manslow, J., "Learning and Adaptation", *AI Game Programming Wisdom*, Charles River Media, 2002, pp.557-566.

[Id Software, 1999] Quake 3 Arena, http://www.idsoftware.com, Id Software Inc., USA, 1999.

[Zarozinski, 2001] Zarozinski, M., "Imploding Combinatorial Explosion in a Fuzzy System", *Game Programming Gems 2*, Charles River Media, 2001, pp. 342.

[van Waveren, 2003] van Waveren, J.M.P., 'The Quake III Arena Bot', University of Technology Delft, Faculty ITS, Holland, 2003.

Dynamic Decomposition Search: A Divide and Conquer Approach and its Application to the One-Eye Problem in Go

Akihiro Kishimoto
Department of Computing Science
University of Alberta
Edmonton, Canada, T6G 2E8
kishi@cs.ualberta.ca

Martin Müller
Department of Computing Science
University of Alberta
Edmonton, Canada, T6G 2E8
mmueller@cs.ualberta.ca

Abstract- Decomposition search is a divide and conquer approach that splits a game position into sub-positions and computes the global outcome by combining results of local searches. This approach has been shown to be successful to play endgames in the game of Go. This paper introduces *dynamic decomposition search* as a way of splitting a problem dynamically during search. Our results in solving one-eye problems in the game of Go show the promise of this approach. Additionally, we propose *relaxed decomposition*, a more ambitious way of splitting positions.

1 Introduction

In two-player games with perfect information, programs typically employ lookahead search to determine which move to play. Traditional brute-force search algorithms explore all possible moves as deeply as possible in order to improve the strength of a program. Such approaches have been very successful, enough to reach the strength of the best human players in games such as chess and checkers [2, 8]. However, the game of Go has been resistant to a pure search-based approach because of its difficult position evaluation and large search space. Current computer Go programs use a combination of exact and heuristic rules instead of brute-force search.

One clear defect of heuristic approximations is that they sometimes fail. As a result, all current computer Go programs show weaknesses in assessing the life and death status of groups, the so-called *tsume-Go* problem. In general, search is the only way to reliably determine the life and death status of stones. Therefore we need to find effective ways to tackle the large branching factor of Go.

One approach to overcome the large search space is to divide a position into independent subpositions and combine the outcome of local searches. This way, we can not only achieve a large reduction of the search space, but also guarantee correctness. *Decomposition search* is such a divide and conquer approach for Go endgames [5]. Using this approach, programs can solve a much larger class of endgame problems than with classical minimax-based solvers. However, in basic decomposition search a problem is split into subproblems only at the root node of a search. There are no further splits during the search, so the method fails for example when there is just a single large undivided area in the beginning. In applications such as tsume-Go, it seems necessary to do decomposition *dynamically* within the search tree.

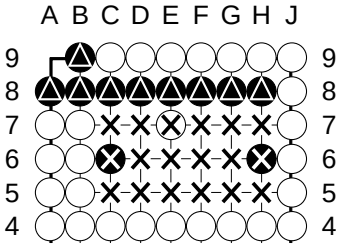


Figure 1: Example of a one-eye problem (Black to live).

This paper presents a dynamic divide and conquer approach to the problem of making *one eye* in an enclosed region. Experimental results show that our approach achieves better performance on average. Also, we introduce the notion of *relaxed decomposition*, which achieves a more fine-grained division into sub-problems.

The structure of the paper is as follows: Section 2 briefly describes the one-eye problem in Go. Section 3 reviews previous work. Section 4 presents the dynamic decomposition algorithm. Section 5 discusses empirical results. Section 6 introduces relaxed decomposition search, and Section 7 presents some conclusions and discusses future work.

2 The One-Eye Problem in Tsume-Go

The one-eye problem in Go is a special case of Life and Death (tsume-Go). It addresses the question of whether a player can create an eye connected to the player's stones in a given region. A problem can be investigated for either player moving first. Although this problem is simpler than full tsume-Go, which is concerned with making two eyes, there are many similarities. For example, every tsume-Go problem in which the group under attack has already surrounded one eye in some region reduces to the one-eye problem on the rest of the board.

A one-eye problem in a given Go position is defined by the following input:

- The two players, called the *defender* and the *attacker*. The defender tries to make an eye and the attacker tries to prevent it.
- The *region*, a subset of the board. At each turn, a player must either make a legal move within the region or pass.

- One or more blocks of *crucial stones* of the defender. The defender wins a one-eye problem by creating an eye connected to all the crucial stones inside the region. The attacker can win by either capturing at least one crucial stone, or by preventing the defender from creating a connected eye in the region.
- *Safe attacker stones*, which surround the region together with crucial defender stones.

In the remainder of this paper, whenever we briefly say "make one eye" we mean "create an eye connected to all the crucial stones inside the region", as above.

Figure 1 shows an example of a one-eye problem. Black is the defender and White is the attacker. Crucial stones are marked by triangles and the region is marked by crosses. Black must make an eye inside the region, while White tries to prevent that. There are unsafe stones at **C6**, **E7**, and **H6**. If these stones are captured, a player might play at such a point later, so they are part of the region.

3 Related Work

3.1 Decomposition Search

In decomposition search, a position is split into subpositions, which are surrounded by safe stones [5]. Local searches, based on combinatorial game theory [1], are then used to analyze each subposition. If all subproblems have loop-free values, then the combinatorial game values of the subproblems can be combined to achieve globally optimal play. As mentioned above, all decomposition happens at the root and there is no further decomposition during search.

3.2 Previous Work on Our One-Eye Solver

The previous version of our one-eye solver is described in [3]. It uses a modified version of Nagai's depth-first proof-number (df-pn) search algorithm [7] which can deal with ko repetitions. The solver checks each point in a given region to find all *potential eye points*. If a complete eye connected to crucial stones is found, the defender wins. If no potential eye point exists, the attacker wins. In all other cases, df-pn search is performed. The basic solver generates all legal moves including a pass. The solver therefore guarantees correctness for enclosed positions. An improved version adds some simple and correct Go-specific knowledge, such as detecting connections to safe stones and forced moves that safely prune other moves. Despite a relatively small amount of Go-specific knowledge, this solver succeeded in solving hard problems that are not solved by the best general tsume-Go solvers in a reasonable time.

3.3 Related Work on Tsume-Go

Wolf's *GoTools* is currently the best tsume-Go solver that specializes in solving completely enclosed positions [10]. GoTools contains a sophisticated evaluation function that includes look-ahead aspects, powerful rules for life and death recognition, and learning dynamic move ordering from the search [11]. Most competitive Go programs also contain a tsume-Go module. The commercial database *Tsume-Go Goliath* uses a proof-number search engine to check the user's inputs. Vilà and Cazenave presented a static approach to detect large eye shapes [9]. Such eye shapes guarantee life by either dividing it into two eyes or living in seki.

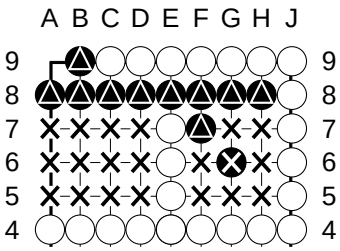


Figure 2: A position to which a divide and conquer approach is applicable. (Black to play.)

4 Dynamic Decomposition Search for the One-Eye Problem

4.1 Basic Idea

The basic idea of our divide and conquer approach is simple. For example, assume that Black needs to make the second eye in Figure 2. A naive algorithm would generate moves at all marked points in its search. This is clearly inefficient, since the marked region is already split into two separate areas. With the exception of *ko* fights, no move played in one area can affect the result of whether there is an eye in the other area. Instead of performing a global search, a divide and conquer approach performs two local searches that can be combined into a global result. This approach can reduce the branching factor and depth of the search by a large margin.

However, if ko fights are involved in a local solution, this approach can change the ko status because formerly local ko threats become non-local. Figure 3 presents such a case. In this example, White can capture the ko first but Black has a local ko threat at 2. White needs one external ko threat to win. However, if the region is divided into two parts, and the solver looks only for eyes, it will miss the ko threat at 2 in the other region, and the ko becomes one where Black needs an external ko threat.

To fix this problem, the solver would need to be extended to search for ko threats in all subregions whenever the result of the one eye search is a ko. This is currently not implemented. There are further complications, for example if two or more subregions end up as some kind of complex multi-step or multi-stage ko. For simplicity, in this paper we concentrate only on eyes for which the defender does not need to fight ko. Our solver correctly deals with double ko etc. as long as all ko are in the same region.

4.2 The Dynamic Decomposition Search Algorithm

Let R be the region at the root position, and r_0 be the *working region* that the algorithm is currently searching in. At

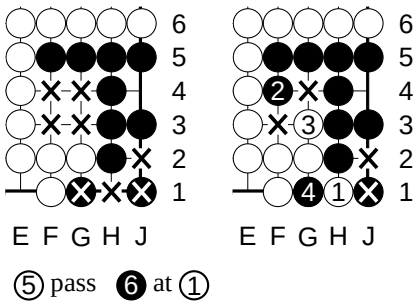


Figure 3: Interacting regions in ko with the divide and conquer approach.

the start, set $R_w = R$. Then *dynamic decomposition search* (DDS) works as follows:

1. If there is an eye in R_w, the defender wins.
2. Otherwise, if no eye space remains in R_w, the attacker wins in R_w.
3. Before generating moves, the region R_w is tested for a possible split into subpositions. Both safe attacker stones and crucial defender stones are used for splitting. This check is done at every newly encountered nonterminal position.
4. Suppose that a position is already partitioned into several subpositions $R_1 \cdots R_k$. If the defender is to play and a move in R_i is chosen by the search control (see the next subsection), then the working region R_w is restricted to R_i. Below this position, moves are generated only in R_i, or an even smaller region when further decompositions occur. This reduces the number of possible moves and the search depth to reach terminal positions. If the defender finds an eye in one of the subregions $R_1 \cdots R_n$, the defender wins. If no eye is found in any subregion, the attacker wins.
5. If the attacker is to play, all moves in R_w are tried.

4.3 Search Control

If there are several subregions, the defender must select one to expand the search in. Df-pn with DDS selects the move to play based on proof and disproof numbers. This dynamically selects a most promising working region at each step. Figure 4 shows an example. Let the defender be a player to prove a position, and the numbers on the board be proof numbers. In this figure, Black plays at **B6**, because it has the smallest proof number. The working region is narrowed to the left side. White answers only in the left subregion after Black's **B6**. Assume that the proof number at **B6** is changed to 5 after exploring positions below Black's play at **B6**. Then, Black plays at **G6** because it becomes the smallest proof number. The working region switches to the right subregion.

4.4 Using the Transposition Table in DDS

In a normal transposition table, Zobrist hashing [12] maps a full board position to its hash key. However, in DDS we must distinguish between different working regions. For example, if a position contains two subregions A and B, there can be three cases for move generation: only in A, only in B, and in both A and B. In order to differentiate these cases, we encode the working region into the hash key as well.

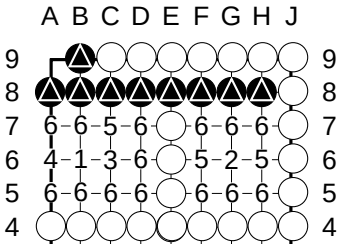


Figure 4: Example of using proof numbers to select the working region.

5 Experimental Results

5.1 Setup of Experiments

Even though there is a large amount of literature on tsume-Go, there are almost no specialized collections of one-eye problems. Landman [4] has a collection of small examples. We created our own test collection with more challenging instances, available at `http://www.cs.ualberta.ca/~games/go/oneeye/`. Each test problem can be solved for either color moving first. Some problems are only interesting if one particular player goes first, and are very easy for the other.

We used two test suites in the experiments. The first test suite, the *toy problem collection* (TOY), contains 13 test positions (26 problems) that are already completely or mostly split into independent problems at the root. Figure 5 illustrates an example of this kind of problem. These problems were used mainly to verify that the decomposition approach works. The second test suite is the *standard problem collection* (STANDARD), an extension of the test collection used in [3]. It contains 81 test positions (162 problems). Some problems are hard for our previous one-eye solver. Figure 6 shows a representative example.

We compare two versions of our solver, with and without dynamic decomposition search (DDS). The version without

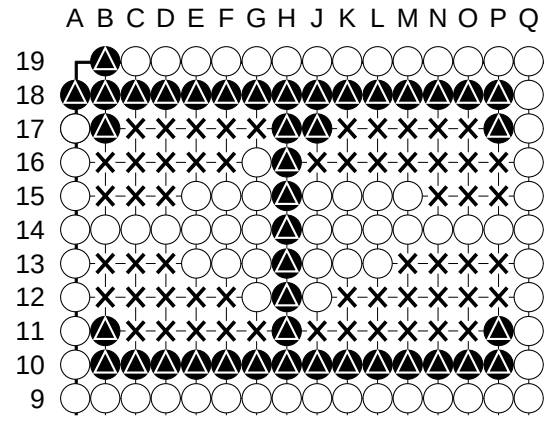


Figure 5: Example of a one-eye problem that is already split (divide-conquer.12.sgf, Black to live by playing at **O12**).

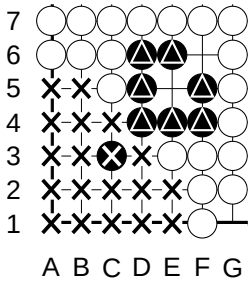


Figure 6: Example of a hard problem (oneeyee.1.sgf, Black lives with **B2**).

DDS, no-DDS, is based on the solver described in [3]. It is improved by adding heuristic initialization of proof and disproof numbers at leaf nodes. All experiments were performed on an Athlon XP 2800+ with a 300 MB transposition table. The time limit was 5 minutes per problem.

5.2 Results

Tables 1 and 2 compare the solving abilities of DDS and no-DDS. More positions are solved by using DDS in TOY. Moreover, all problems solved by no-DDS were also solved by DDS. On the other hand, both versions solved the same subset of problems in STANDARD. The improvement achieved by DDS is a factor of 66 in TOY and 1.2 in STANDARD in total execution time. This indicates that DDS surpasses the abilities of our previous df-pn solver.

On average, DDS is about 13% slower in terms of node expansions per second (see Table 2). However, sometimes DDS is faster, especially in TOY (see Table 1). In particular, with decompositions at or near the root, DDS can concentrate on a smaller region, which speeds up basic operations such as detecting potential eye points and move generation.

Table 1: Performance comparison for DDS and no-DDS in TOY. All statistics are computed for 23 problems solved by both program versions.

	Number of problems solved	Total time (sec)	Total nodes expanded	Nodes expanded per second
No-DDS	23	52	2,169,239	41,636
DDS	26	0.79	49,433	62,573
Total Problems	26	-	-	-

Table 2: Performance comparison for DDS and no-DDS in STANDARD. All statistics are computed for 157 problems solved by both program versions.

	Number of problems solved	Total time (sec)	Total nodes expanded	Nodes expanded per second
No-DDS	157	1,975	84,084,752	42,573
DDS	157	1,645	62,129,738	37,774
Total Problems	162	-	-	-

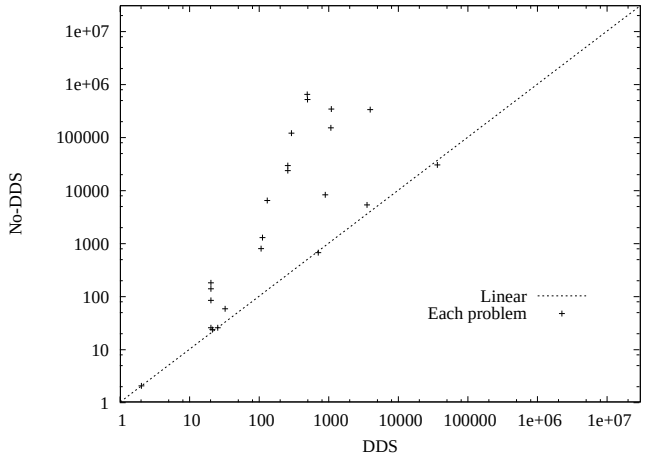


Figure 7: Node expansions for toy problems solved by both versions.

Figure 7 compares node expansions of both solvers for each problem in TOY. The number of nodes explored by DDS is plotted on the X-axis against no-DDS on the Y-axis on logarithmic scales. In points above the diagonal DDS performed better. Except for one problem DDS expanded at most as many nodes as no-DDS, and often dramatically less. The performance of DDS scales exponentially better in the size of problems. For example, DDS solved the position in Figure 5 in 1,075 nodes, while no-DDS needed 334,718 nodes. This is not surprising, since all positions in this set are ideal for DDS, while no-DDS suffers from combinational explosion.

Figures 8 and 9 present the results for STANDARD. None of these problems were designed with decomposition in mind. In contrast to Figure 7, there are more problems where DDS was slower. However, on average DDS explores less nodes and needs less execution time. This is especially true for the larger problems, so DDS seems to scale better. DDS sometimes improves the performance by a large margin. For example, DDS needed 360,163 nodes in 7.5 seconds for the position in Figure 6, whereas no-DDS explored 1,732,845 nodes in 35.6 seconds. In this position, decompositions triggered by black tactical (one and two) safe stones reaching the borders of the board seem to occur

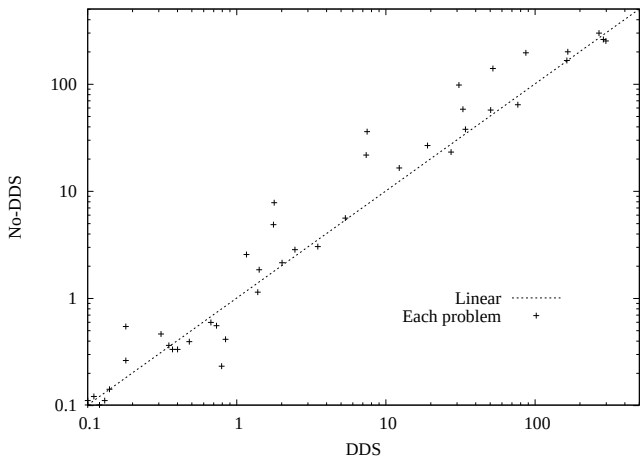


Figure 8: Execution time for standard problems solved by both versions.

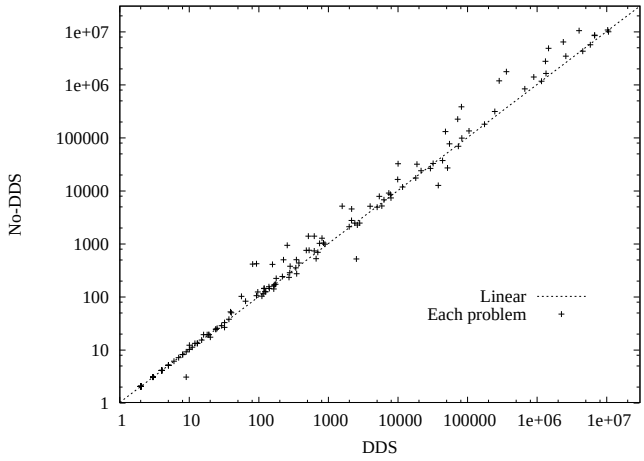


Figure 9: Node expansion for standard problems solved by both versions.

frequently.

In the hard problems of this set, the percentage of nodes in which decompositions are possible varies from 16% to 50%. In Figure 6, DDS detected 127,479 (35.3%) decompositions.

6 A Relaxed Decomposition Model

DDS is limited in the way that splits are recognized. The only points used to split positions are those occupied by safe attacker and crucial defender stones. In this section we introduce a less rigid decomposition that uses "almost safe" attacker stones as well. Figure 10 shows an example. If we assume that the two white stones marked by squares are safe, then we can split the area into two subregions, a left subregion marked by small grey squares and a right region marked by crosses. The attacker can always make the two marked white stones safe by following a simple *miai* connection strategy: Whenever the defender plays either *connection point* A or B, reply on the other point. In our *relaxed decomposition* model, we use such stones for splitting a position. However, during the search we must handle the case where a miai connection is attacked by the defender.

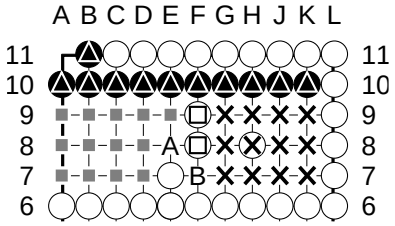


Figure 10: Relaxed decomposition.

In this case, our approach extends the search to the union of the affected subregions. We explain the algorithm in detail with the help of Figure 10. In this figure, let region R_1 consist of the empty point A and all points marked by filled squares. Region R_2 contains B and all points marked by crosses. The connection points A and B together form a miai connection from safe attacker stones to an almost safe attacker block. Without loss of generality, assume that the defender starts playing in R_1. Then *Relaxed Decomposition Search* (RDS) defines the strategies of both players as follows:

1. If both A and B are empty, both players are restricted to play moves in R_1.

2. If either A or B contains a stone of the attacker, the relaxed decomposition has changed into a normal decomposition. Both players keep playing in R_1.

3. Otherwise, if at least one of A and B contains a stone of the defender, and the other point is empty or also occupied by the defender, the region is extended and both players continue playing in $R_1 \cup R_2$.

In RDS, as long as the defender does not play at A, both players stick to play in R_1. However, if the defender plays at A and the attacker does not respond at B, then the defender can invade R_2. As in DDS, the defender can switch between trying moves in R_1 and R_2 at the root. This process is controlled by proof numbers.

The following lemma is needed to prove correctness of RDS.

Lemma 6.1 *Assume the following:*

1. *Position P contains region R which is split into two subregions R_1 and R_2 by relaxed decomposition.*

2. *The attacker is to play in P.*

3. *Two connection points A in R_1 and B in R_2 are empty.*

4. *If the attacker plays at A in P, the defender can still create an eye unconditionally (without ko) in R_1.*

Then, the defender can create an eye in R_1 unconditionally from the position after the attacker plays at A.

We give a proof sketch for the case of a DAG. Let P_1 be the node after the attacker plays a move at A for P, and P_2 be the node after the attacker plays a move in R_2 for P. We prove that the defender can make an eye for P_2 by following the winning strategy for P_1. The lemma is proven by induction on the maximum depth d of a terminal node in the proof graph of P_1.

Case 0: $d = 0$ If P_1 is a terminal position, the same eye exists in both P_1 and P_2.

Case 1: $d = 1$ The defender can create an eye by making move $m \neq A$ in R_1 for P_1. Since P_1 is identical to P_2 within $R_1 - \{A\}$, m is legal in P_2 and also creates an eye there.

Case 2: induction step Assume that Lemma 6.1 holds for all $d \leq k$. We prove that the lemma also holds for $d = k + 2$. Let $m \in R_1 - \{A\}$ be the winning defender move in P_1, Q_1 be P_1's child after playing m for P_1, and $n_1, n_2, \cdots, n_l$ be Q_1's children. Let n_1 be the node after the attacker passes for Q_1. Since $m \in R_1 - \{A\}$, m is legal for P_2. Let Q_2 be P_2's child by playing m, o_1 be Q_2's child after the attacker plays at A, $o_2 \cdots o_p$ be Q_2's children after moves in R_2, and $o_{p+1} \cdots o_{p+q}$ be Q_2's children from moves in R_1. The defender can make an eye for $o_{p+1} \cdots o_{p+q}$ by the assumption of Lemma 6.1. R_1 is completely separated from R_2 in o_1 and n_1. Moreover, since o_1 and n_1 are identical positions in R_1, the defender can create an eye in o_1. By induction, since o_1 has depth $d \leq k$, the defender can create an eye for $o_2 \cdots o_p$. Thus, the lemma is proven for the case of $k + 2$.

The following theorem guarantees the correctness of RDS in the case that an eye is found.

Theorem 6.1 *Assume that R is split into two subregions R_1 and R_2 by RDS. If the result of RDS shows that the defender can create an eye unconditionally (without ko) in either R_1 or R_2, then that eye can always be made against any attacker strategy in $R_1 \cup R_2$.*

In the following, we call the proof graph created by RDS the *RDS proof graph*, and a proof graph for the whole region $R_1 \cup R_2$ an *original proof graph*.

We present a proof sketch which shows that each RDS proof graph can be converted into an original proof graph, for the case where the RDS proof graph is a DAG. We use induction on depth of a terminal node in the RDS proof graph. We only explain the first case of RDS with the help of Figure 10. The other two cases are trivial, because searching either with completely separated subregions or with the whole region is performed in those cases.

Assume without loss of generality that the first defender move is in R_1. As above, if the RDS graph contains only a terminal node, an eye already exists and the RDS proof graph also works as an original proof graph.

Otherwise, let n be the root of the RDS proof graph. Assume that by induction n's descendants in the RDS graph have been converted to original proof graphs.

- If n is an OR node, n's move m leading to n's child n_c in the RDS proof graph is also legal for searching in $R_1 \cup R_2$. n_c's RDS proof graph can be converted to n_c's original proof graph by the induction assumption. Hence, we can construct n's original proof graph by adding a branch m from n to n_c's original proof graph.

- If n is an AND node, assume that n's children $n_{c_1}, \cdots, n_{c_k}$ in R_1 have proof graphs. Let n_{c_1} be n's child after the attacker plays a move at A, $n_{c_{k+1}} \cdots n_{c_l}$ be n's children by playing in R_2. We need to prove that $n_{c_{k+1}} \cdots n_{c_l}$ have original proof graphs. n_{c_1} guarantees that an eye can be made in R_1, since R_1 is completely separated from R_2. By applying Lemma 6.1 to $n_{c_{k+1}} \cdots n_{c_l}$ based on n_{c_1}'s proof graph, the defender can make an eye for $n_{c_{k+1}} \cdots n_{c_l}$. Hence, $n_{c_{k+1}} \cdots n_{c_l}$ have original proof graphs.

We believe that our lemma and theorem also hold for cyclic graphs in the case where the eye can be made unconditionally. However, we need a different approach to prove our conjecture for cyclic graphs, because we use a property of DAGs in proving the theorem by induction. The induction in our proof uses the fact that children have a depth that is at least 1 smaller than their parents. This property does not hold for cycles.

RDS can split positions more frequently than DDS. The approach can be generalized to more than two relaxed split subregions, as long as all the miai connections to safe attacker stones are disjoint. However, the completeness of the relaxed decomposition algorithm is not known yet. To prove that no eye is possible, the worst case scenario might require re-searches in the whole region. In this case, we must devise an efficient way for re-searches.

7 Conclusions and Future Work

In this paper, we presented a method that dynamically decomposes a position into sub-positions during search. The results of this dynamic decomposition search are encouraging. In many problems, DDS is able to reduce the search space by a large margin, thereby enabling the one-eye solver to solve harder problems more quickly. However, there are still a lot of unexplored topics such as:

- The current version of DDS is limited to dealing with ko fights only in the same region. To overcome this problem, we need to find a detailed ko status and ko threat status of each divided region, and combine them.

- Investigating relaxed decomposition search is a challenging topic from both the theoretical and practical point of view. Furthermore, splitting a position in a more aggressive way such as by using divider patterns [6] is an interesting extension of this research topic.

- Applying the ideas to other parts of Go, such as connections, territories, and tsume-Go, or other games

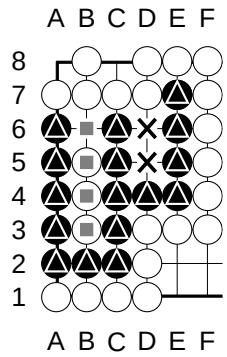


Figure 11: Decomposition for tsume-Go.

such as Hex will be a challenging topic. In particular, tsume-Go is an interesting domain for further investigations. In tsume-Go, the decomposition will be more complicated. Suppose that a region is split into two completely separated rooms A and B. There are several possibilities to be considered, such as making (1) two eyes at A, (2) one eye at A and the other eye at B, (3) two eyes at B, or (4) one eye either at A or at B if one eye already exists. Local searches must distinguish between *sente* and *gote*. For example, the position in Figure 11 has two subregions, a left subregion R_1 marked by small grey squares and a right subregion R_2 marked by crosses. Move **B6** in R_1 makes one and a half eye and move **D6** in R_2 makes half an eye. Black can live with **B6**, since White cannot play both **B4** and **D6**. To solve such a problem with two separate searches in R_1 and R_2, return values such as "1.5 eyes" or "0.5 eyes" [4] must be recognized by the search.

- Incorporating DDS into a complete Go-playing program is an important topic. We will probably have to heuristically decompose positions, since many positions in the real games are not closed off by safe attacker stones.

Acknowledgments

We would like to thank Adi Botea for reading drafts of the paper and giving valuable comments. Financial support was provided by the Natural Sciences and Engineering Research Council of Canada (NSERC) and the Alberta Informatics Circle of Research Excellence (iCORE).

Bibliography

[1] E. Berlekamp, J. Conway, and R. Guy. *Winning Ways*. Academic Press, 1982.

[2] M. Campbell, A. Joseph Hoane Jr., and F.-h. Hsu. Deep Blue. *Artificial Intelligence*, 134(1-2):57–83, 2002.

[3] A. Kishimoto and M. Müller. Df-pn in Go: Application to the one-eye problem. In *Advances in Computer Games. Many Games, Many Challenges*, pages 125–141. Kluwer Academic Publishers, 2003.

[4] H. Landman. Eyespace values in Go. In R. Nowakowski, editor, *Games of No Chance*, pages 227–257. Cambridge University Press, 1996.

[5] M. Müller. Decomposition search: A combinatorial games approach to game tree search, with applications to solving go endgames. In *Proceedings of International Joint Conference on Artificial Intelligence (IJCAI'99)*, volume 1, pages 578–583, 1999.

[6] M. Müller. Computer Go. *Artificial Intelligence*, 134:145–179, 2002.

[7] Ayumu Nagai. *Df-pn Algorithm for Searching AND/OR Trees and Its Applications*. PhD thesis, Department of Information Science, University of Tokyo, 2002.

[8] J. Schaeffer. *One Jump Ahead: Challenging Human Supremacy in Checkers*. Springer-Verlag, 1997.

[9] R. Vilà and T. Cazenave. When one eye is sufficient: A static approach classification. In *Advances in Computer Games. Many Games, Many Challenges*, pages 109–124, 2003.

[10] T. Wolf. The program GoTools and its computer-generated tsume Go database. In Hitoshi Matsubara, editor, *Game Programming Workshop (GPW)*, pages 84–96. Computer Shogi Association, 1994.

[11] T. Wolf. Forwarded pruning and other heuristic search techniques in tsume go. *Information Sciences*, 122(1):59–76, 2000.

[12] A. L. Zobrist. A new hashing method with applications for game playing. Technical report, Department of Computer Science, University of Wisconsin, Madison, 1970.

Combining Tactical Search and Monte-Carlo in the Game of Go

Tristan Cazenave
Labo IA, Université Paris 8
2 rue de la liberté
93526, St-Denis, France
cazenave@ai.univ-paris8.fr

Bernard Helmstetter
Labo IA, Université Paris 8
2 rue de la liberté
93526, St-Denis, France
bh@ai.univ-paris8.fr

Abstract- We present a way to integrate search and Monte-Carlo methods in the game of Go. Our program uses search to find the status of tactical goals, builds groups, selects interesting goals, and computes statistics on the realization of tactical goals during the random games. The mean score of the random games where a selected tactical goal has been reached and the mean score of the random games where it has failed are computed. They are used to evaluate the selected goals. Experimental results attest that combining search and Monte-Carlo significantly improves the playing level.

1 Introduction

Monte-Carlo Go has been invented in 1993 [1]; it is a simple way to program a decent computer Go program using very little knowledge. It has been recently the subject of renewed interest [2, 3, 4]. The combination of Monte-Carlo with traditional Go programming techniques is promising and gives good results, as can be seen from recent computer Go events. In this paper we show that an original combination of Monte-Carlo methods with tactical search outperforms Monte-Carlo alone. The resulting program is about 50 points above standard Monte-Carlo on the 9x9 board, and 26 points above the previous version of Golois.

The program starts with performing searches for each possible tactical goal and for each color starting first, in order to find unsettled problems. Examples of tactical goals are capturing a string, connecting two strings or making an eye. In a second phase, the program selects interesting goals related to unsettled problems. In a third phase, it computes statistics on the selected goals.

In standard Monte-Carlo Go, the means of the random games where an intersection has been played first by a player are computed for each intersection. What is done for the intersections can also be done for tactical goals. Therefore, we define the following unification of the notion of a goal: a goal can be either related to an empty intersection (in which case the success of the goal depends only on who has played first on the intersection), or it can be related to a tactical goal. We handle these two different classes of goals in a similar way: we compute the mean of the results of the random games where the goal has been reached and the mean of the results of the random games where it has failed. The value of the goal is the difference between the two means. We choose the goal of highest value. If this goal is an intersection goal we play at the intersection; if it is a tactical goal we play a move that reaches the goal. In case several moves reach the goal, we choose the one having the highest intersection value.

The second section presents Monte-Carlo methods for games; the third section details the different search algorithms used in our program; the fourth section presents the statistics our program computes in the random games; the fifth section deals with the combination of Monte-Carlo and search; the sixth section presents experimental results.

2 Monte-Carlo methods and Games

Monte-Carlo methods in games use statistics on more or less random games in order to find the best move. The first application of Monte-Carlo methods to Go was written by B. Bruegmann [1]. Recently, other Go programs have started using it, and improved it, simplifying the method and proposing basic improvements [2] or combining it with a knowledge based program that selects a few number of moves that are later evaluated by the Monte-Carlo method [3].

There are several slightly different ways to write a Monte-Carlo Go program [2]. In this paper, we call standard Monte-Carlo Go the following algorithm. The program plays a large number (usually 1,000 to 10,000) of random games starting at the current position. The moves of the random games are chosen almost randomly among the legal moves, except that they must not fill the player's eyes. A player passes in a random game when his only legal moves are on his own eyes. The game ends when both players pass. In the end of each random game, the score of the game is computed using Chinese rules (in our case, it consists in counting one point for each stone and each eye of the player's color, and subtracting the player's count to its opponent count). The program computes, for each intersection, the mean results of the random games where it has been played first by one player, and the mean for the other player. The value of a move is the difference between the two means. The program plays the move with the highest value.

Monte-Carlo simulations have also been used in other games such as Bridge [5], Poker [6], Tarok [7] and Scrabble [8] for example. In the games of Bridge and Tarok, a combination of Monte-Carlo and search is usual. Statistics are performed on open deals that are solved by search. However, in our approach, searches are performed only once, independently of the random games, and the tactical problems that are solved by search are used to choose the statistics that will be computed during the random games.

Our approach of combining search and Monte-Carlo is new and orthogonal to the previous approaches used in Go: it is very likely that it also improves their performances.

3 Search

We use search algorithms to solve tactical problems such as capturing/saving a string or connecting/disconnecting two strings. In this section, we present five tactical search algorithms that are used in our program. The first one is a capture search, the second one is a connection search, we follow with search for the connection between a string and an empty intersection, eye search and life and death search.

3.1 Capture Search

For each string on the board a capture search is tried. If the capture search fails, no other search is performed. If a capturing move is found, a search that tries to save the string, by playing first a move of the color of the string, is performed. If none of the possibly saving moves works, the string is captured even if the color of the string plays first, and capture searches for this string are stopped. On the contrary, if a saving move exists, other searches are performed. The program tries to find all the possible capturing moves, and all the possible saving moves.

At the end of the process, for each string on the board, its capture status and its save status are known, and for strings that can both be captured by one player and saved by the other, multiple capturing and saving moves are found when possible.

3.2 Connection Search

For each string on the board, the program looks for the strings that can be connected to the first string by playing at most four moves in a row. Then, for each pair of strings, it searches for a connection. When a connecting move is found, it also searches for a disconnection. When a disconnecting move is also found, it searches for all the connecting and disconnecting moves.

After connection search, the program has a list of possible connections, and for each connection the connection status and the disconnection status, as well as multiple connecting and disconnecting moves for strings that can both be connected by one player and disconnected by the other.

3.3 Empty Connection Search

The empty connection goal is based on the connection goal. It involves a string and an empty intersection. The goal of the game is to connect the empty intersection to the string. In practice, in order to find unsettled empty connection problems, the program plays a move of the color of the string on the empty intersection, then the disconnection search is called for the string containing the played empty intersection and the string to connect to. If they cannot be disconnected the empty connection problem is unsettled and the associated move consists in playing on the empty intersection.

3.4 Eye Search

For each intersection on the board, it searches if an eye can be made on the intersection or on any of the direct neighbors.

3.5 Life and Death Search

Life and death search uses Generalized Widening [9] for non enclosed groups. Life and death search is called for each group of strings. Groups are built using the results of the connection search between strings.

4 Statistics on random games

In this section, we describe the different statistics that are computed during random games. Statistics are related to goals. We compute the mean of the results of the random games where a goal has been reached and the mean of the results of the random games where it has failed.

Among the unsettled goals found by search, the program chooses some interesting ones and computes statistics on them.

The different types of goals we use for the statistics are :

- The goal of playing first on an intersection. It is the only goal used in the standard Monte-Carlo approach.
- The goal of owning an intersection at the end of a game.
- The goal of capturing a string. The goal is considered to be lost as soon as the string has more than four liberties.
- The goal of connecting two strings. It is possible that the two strings are captured after being connected in a random game, but we still consider this as a success for the connection.
- The goal of connecting a string and an empty intersection.
- The goal of making an eye on an intersection or on any of its neighbors.

5 Combining Search and Monte-Carlo

There are usually different problems in the initial position, i.e. capture, connection, empty connection, eyes and life and death problems. In this section we detail the combination of search and Monte-Carlo. We start explaining why all the problems are not taken into account and how we select the problems that will be used to compute statistics on. We follow with the statistics collected during the random games. We describe how a move is chosen. Then, we discuss on the usefulness of collecting statistics on unsettled problems, and we define positive and negative goals.

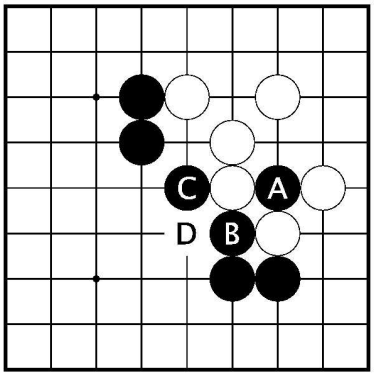


Figure 1: Choosing the simplest connection

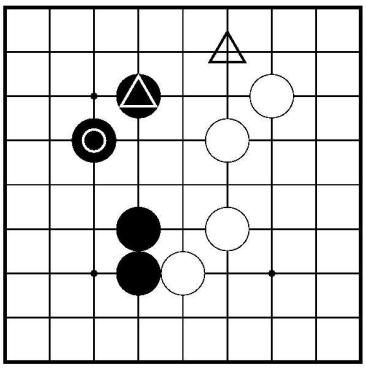


Figure 2: Empty connection

5.1 Selecting problems

In order to avoid playing bad moves and overestimating the importance of a problem, the program needs selecting among the unsettled problems the ones that will be used to compute statistics.

Strings that cannot be disconnected are amalgamated in groups. Groups are used to select the unsettled connection problems that will be used to gather statistics.

It can happen that a string A is connected with a complex search to string B, that string B is connected with a complex search to string C, and that A and C are connected but that the search to find it is too complex. In this case the program can think there is an unsettled connection between A and C and add a useless move. In order to avoid this behavior, complex unsettled connection problems between two strings of the same group are not taken into account.

Furthermore, when there are multiple connection problems that connect the same groups, only the simplest connection problem is retained. The simplicity of a connection problem is measured by the number of moves in a row that are needed to connect the two string in the initial position. For example, in the figure 1 the strings B and C belong to the same group. The connection between A and B can be prevented by White, capturing A for example. The connection between C and A can also be prevented by capturing A, but the white move at D also prevents the connection. However, D is not the kind of move that we want the program to consider to disconnect the black group from A. Here the connection between A and B is simpler than the connection between A and C, so the program will only retain the connection between A and B for gathering statistics. The problem of only retaining the simplest connection is not only a problem of avoiding moves like D, it is also a matter of correctly evaluating the value of the connection between two groups. The mean of the games where A and C have been connected is larger than the mean of the games where A and B have been connected. So keeping only the simplest connection avoids overestimating connections.

Empty connections between strings and empty intersections follow a similar pattern. In the figure 2, the triangle empty intersection can connect to the triangles string. However, it can also connect to the circled string. The mean of the games containing the connection to the circled string is higher than the mean of the games with the connection to the triangles string. In order to avoid overestimating empty connections, the program only selects the simplest empty connection when there are multiple empty connections of the same color to the same empty intersection.

Empty connections are also used to find long distance connections between groups that are too complex to be found by the connection search. If an empty intersection is connected to two different groups, and that there is no unsettled connection problem between these two groups, then a new unsettled connection problem is created that joins the two groups, playing on the common connected empty intersection. There are some exceptions to this rule that can be found using a search for transitivity of connection [10].

5.2 Gathering statistics on selected problems

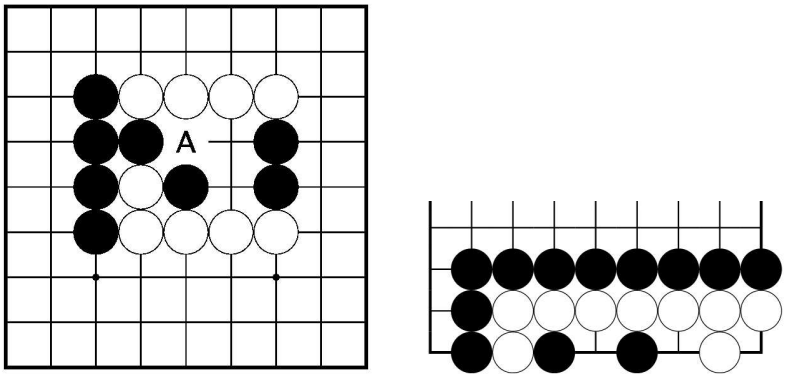
After the program has selected the interesting goals, it plays many random games and for each selected goal, it computes the mean score of the random games where the goal has succeeded, and the mean score of the random games where it has failed.

We call *raw mean* of a move the value of the intersection goal associated to the move.

The statistics on the final color of an intersection are used to evaluate the importance of playing moves related to life and death. The mean score of a life problem related to a string is the mean score of the games which ended with an intersection of the string keeping its original color. The mean score of the associated death problem is the mean score of the games which ended with the intersection of the other color.

5.3 Choosing a move

For the selected goals, we compute the value of the goal which is the difference between the mean score of the games where it succeeds and the mean score of the games where it fails. The program chooses to play the goal with the highest value. Among all the moves associated to the selected goal, the program chooses the one with the highest raw mean.



useful connection useless connection

Figure 3: connections

5.4 Why are statistics on unsettled problems useful?

It is useful to compute statistics in the random games on goals related to unsettled problems. The raw mean of a move is different from the mean of the games where the goal associated to the move has been reached. Figure 3 explains the difference between the two means for the connection goal. We see in the *useful connection* diagram that the move at A connects the two strings. In the random games where a black move at A has been played first, the two strings will only be connected three times out of four. But the program knows, with the connection search, that it is possible to always connect the two strings. The mean of the results of the games where the two strings have been connected give an evaluation of the move at A, which is better than the raw mean of the move at A because it takes into account the fact that the two strings are connected after A.

Another example is given in the *useless connection* diagram of figure 3. This time the games where the two black strings have been connected have a mean which is less than the mean of all the games since the connection move is always useless. Therefore the mean associated to the connection is less than the mean of the best move, and the program does not play the connection. It finds out by itself that it is a useless connection.

Figure 4 shows the symmetric example. The two white strings are disconnected three times out of eight, and the disconnecting move results in a disconnection of the two strings only three times out of four in the random games. On the contrary, the search tells us that they can always be disconnected. To evaluate the disconnecting move, it is more accurate to use the mean of the games where the two strings have been disconnected, than to use the mean of the games where the disconnecting move has been played first.

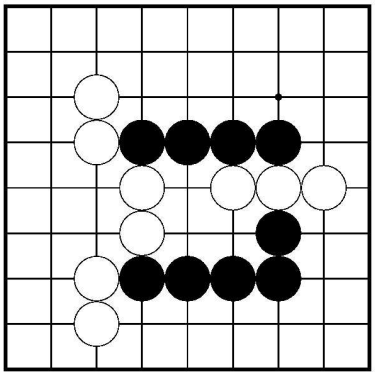


Figure 4: disconnection

5.5 Positive and negative goals

We introduce the notion of positive and negative goals. Positive goals are goals we have confidence they can be reached if the search algorithm returns so. For example, when the search algorithm finds a capturing move, we have confidence that the string can be captured. Examples of positive goals are capture, connection and life. On the contrary negative goals are not sure. For example, when the capture search finds a move that saves a string, we are not assured that the string cannot be captured latter: the string might have gained five liberties but may be completely surrounded by alive enemy groups and therefore be bound to capture anyway. Example of negative goals are saving a string, disconnecting two strings or killing a group.

For negative goals the statistics tend to over-estimate the interest of playing in the related problems. For example, statistics on disconnection measure the mean of the games where two strings have not been connected. It means the program does not count the games where the strings have been disconnected according to the search at a time in the game, but have been connected later in the game because of the capture of a common adjacent string that had temporarily gained five liberties. In order to avoid this behavior we have used the connection game evaluation function to detect the disconnection found by the search at any time in the random games. This way we count for the disconnection mean the games were the evaluation function sends back lost in the game even if at the end of the game the two strings end being connected. Similarly, for the save goal, our program counts all the games where the string had more than four liberties during the game.

6 Experimental results

In order to test our combination of tactical search with Monte-Carlo, we have played 9x9 games between two programs. The first program is the standard Monte-Carlo algorithm. The second program combines tactical search and Monte-Carlo. They both play 10,000 random games before choosing a move. Each program played twenty 9x9 games against the other: ten games with black and ten games with white. The games were scored using Chinese rule.

The capture, connection and eye search use Generalized Threats Search [11]. The threat used is the (6,3,2,0) threat. Life and death search uses Generalized Widening [9] for non enclosed groups.

The program that combines search and Monte-Carlo wins on average by 52.1 points against the standard Monte-Carlo method on the 9x9 board, the standard deviation being 34.2 points. On a Celeron 1.8 GHz, the standard Monte-Carlo algorithm plays a move in five seconds on average for 10,000 9x9 random games. The combination of search and Monte-Carlo plays a move in ten seconds for 10,000 9x9 random games.

In order to compensate for the additional time used by the combination of Monte-Carlo and search, we played the combination program with 1,000 random games against the standard Monte-Carlo with 10,000 games. The combination is then twice as fast as the standard Monte-Carlo. Still, the combination program wins on average by 24.6 points on the 9x9 board, the standard deviation being 40 points.

The combination of Monte-Carlo and search with 10,000 random games has also been tested against Golois. Golois uses exactly the same tactical search algorithm as the program based on Monte-Carlo and search. Golois uses a depth one global search and hand tuned heuristics to evaluate the strength of groups and the moves. Forty 9x9 games have been played between the two programs and the combination of search and Monte-Carlo wins on average by 26 points. Given that the two programs have the same tactical Go knowledge, it appears that the use of Monte-Carlo to assess the importance of tactical goals is a promising alternative to hand tuned evaluation knowledge.

7 Conclusion and Future Work

We have presented a way to integrate search with the Monte-Carlo method in Go. Our program computes statistics during the random games on the goals searched in the initial position, in order to improve the accuracy of the evaluation of the moves related to the goal. The resulting program improves the average result of Monte-Carlo methods against a standard Monte-Carlo Go program by more than 50 points in 9x9 games.

Future work includes using search and statistics on other goals and combinations of goals. It may be possible to better handle the different results of the search for negative goals, and to improve the evaluation of these goals. We could also improve the confidence in the statistics on the random games, by taking into account the moves that threaten the won tactical goals, and replying them inside the random games so as to keep the important tactical goals won. This would prevent the program to overestimate the value of threats and it would result in a better evaluation of the position.

Bibliography

[1] Bruegmann, B.: Monte Carlo Go. ftp://ftp-igs.joyjoy.net/go/computer/mcgo.tex.z (1993)

[2] Bouzy, B., Helmstetter, B.: Monte Carlo Go developments. In: Advances in computer games 10, Kluwer (2003) 159–174

[3] Bouzy, B.: Associating domain-dependent knowledge and monte carlo approaches within a go program. In: Joint Conference on Information Sciences, Cary (2003) 505–508

[4] Bouzy, B.: Associating shallow and selective global tree search with monte carlo for 9x9 go. In: 4th Computer and Games Conference, Ramat-Gan (2004)

[5] Ginsberg, M.L.: GIB: Steps toward an expert-level bridge-playing program. In: IJCAI-99, Stockholm, Sweden (1999) 584–589

[6] Billings, D., Davidson, A., Schaeffer, J., Szafron, D.: The challenge of poker. Artificial Intelligence **134** (2002) 210–240

[7] Lustrek, M., Gams, M., Bratko, I.: A program for playing tarok. ICGA Journal **26** (2003) 190–197

[8] Sheppard, B.: Efficient control of selective simulations. ICGA Journal **27** (2004) 67–80

[9] Cazenave, T.: Generalized widening. In: ECAI 2004, Valencia, Spain, IOS Press (2004) 156–160

[10] Cazenave, T., Helmstetter, B.: Search for transitive connections. Information Sciences (2005)

[11] Cazenave, T.: A Generalized Threats Search Algorithm. In: Computers and Games 2002. Volume 2883 of Lecture Notes in Computer Science., Edmonton, Canada, Springer (2002) 75–87

Bayesian generation and integration of K-nearest-neighbor patterns for 19x19 go

Bruno Bouzy
Université Paris 5, C.R.I.P.5
45, rue des Saints-Pères 75270 Paris Cedex 06 France
tél: (33) (0)1 44 55 35 58, fax: (33) (0)1 44 55 35 35
email: bouzy@math-info.univ-paris5.fr

Guillaume Chaslot
Ecole Centrale de Lille
Cité Scientifique - BP 48, 59651 Villeneuve d'Ascq Cedex
email: chaslot.guillaume@ec-lille.fr

Abstract- This paper describes the generation and utilisation of a pattern database for 19x19 go with the K-nearest-neighbor representation. Patterns are generated by browsing recorded games of professional players. Meanwhile, their matching and playing probabilities are estimated. The database created is then integrated into an existing go program, INDIGO, either as an opening book or as an enrichment of other pre-existing hand-crafted databases used by INDIGO move generator. The improvement brought about by the use of this pattern database is estimated at 15 points on average, which is significant on go standards.

1 Introduction

Because the branching factor and the game length forbid global tree search in go, and because evaluating non terminal go positions is hard [16], computer go remains a difficult task for computer science [17, 15]. In addition, computer go is an appropriate testbed for AI methods [8]. INDIGO [7] is made up of the Monte Carlo (MC) module and the knowledge module. The MC module has been described recently [9, 4], and the knowledge module was described before 2003 [8, 5, 6]. To briefly present the current INDIGO move decision process, the knowledge module provides the MC module with ns moves, and, in order to select the best move, the MC module plays out a lot of complete random games starting with these moves and computes mean values.

The knowledge module includes various pattern databases built manually. Hand-crafted databases have many downsides: they contain errors, they have holes, and they cannot be easily updated. Furthermore, the various pattern bases in INDIGO do not share the same format: the first one (FORME_M) includes domain-dependent features used by the conceptual evaluation function, the second one (FORME_3X3) contains 3x3 patterns optimized for fast simulations, and the last one is dedicated to large patterns useful at the beginning of the game (FORME_B and FORME_C). Due to the success of the MC module within INDIGO, we aimed at using statistics in the knowledge module too. Thus, it was the right time to test the automatic creation of a new pattern database and observe its positive effects within the INDIGO architecture. The automatic creation of patterns avoids errors and holes in the database. The automatic creation is performed by browsing recorded professional games to create patterns and to estimate both their matching probabilities and their playing probabilities when matching. In other words, the approach we adopted is a bayesian approach.

In order to avoid any limitation due to the size of patterns, particularly at the beginning of the game, we used the K-nearest-neighbor representation in which the relevant neighbors are the occupied intersections and the edges. For this reason, this database is named FORME_K.

Section 2 is a summary of works related to the current paper. Section 3 defines the K-nearest-neighbor representation used. Then, section 4 describes the creation of patterns and their probabilistic features. Section 5 underlines the experiments performed to integrate this work within INDIGO, and assesses the improvements. Before conclusion, some interesting perspectives are highlighted by section 6.

2 Related work

Despite of its importance within go programs, the litterature about pattern acquisition, local move generation or recorded professional games is not very abundant. [2] by Mark Boon was the first paper to describe a pattern-matcher in great details: the 5x5 window pattern-matching algorithm of Goliath, best program in 1990. But this paper did not deal with the pattern acquisition. Recently, Erik van der Werf described a neural network approach using professional recorded games to generate local moves [19], predict life and death [21], or score final positions [20]. Since it also browses recorded games to produce local moves, the current work is similar to Erik van der Werf's approach, but it is less sophisticated because it uses the K-nearest-neighbor representation instead of a neural network. Moreover, it is not intended to predict life and death or to score positions. Tristan Cazenave worked on automatic acquisition of tactical patterns for eyes or connections [11], even including liberties [12]. The current work is similar to Cazenave's work because it consists in automatic acquisition of patterns but it is quite different because Cazenave's patterns were generated in a specific tactical context: connecting or making eyes, by using explanation-based learning. Finally, [3] was an attempt to generate 4x4 patterns by retrograde analysis. Although it dealt with automatic acquisition of patterns, this work was completely different from the present work because it was limited to small boards. Furthermore, it did not use the bayesian approach but retrograde analysis. Finally, the approach used by Franck de Groot [14] to build his game analyser software is not far from our approach because it also consists in browsing professional games.

However, it is different because the patterns representation remains window-based, and because the number of games used by his approach is significantly higher (50,000 instead of 2,000).

3 K-nearest-neighbor representation

The K-nearest-neighbor representation is common in pattern recognition [1]. This section defines the K-nearest-neighbor representation used in this work.

3.1 K-nearest-neighbor patterns

The picture below shows an example a K-nearest-neighbor pattern.

```
  +@+
 +++++
 ++*+O
 +++++
  ++@
```

The pattern always advises to move *in its center* marked by a '*'. '+' represents an empty intersection. 'O' represents a white stone. '@' represents a black stone. '+' is an *unimportant* fact in this representation. 'e' represents an empty intersection located on the edge of the board. A black or white stone, an edge or a corner are *important* facts. A pattern contains a number of important facts, named K. In the example above, K=3.

The center of the pattern being given, we assume the neighboring intersections are ordered according to a distance. Moreover, we assume that this pre-defined order avoids ties between intersections situated at the same distance from the center of the pattern. With such assumption, the pattern matching principle remains simple and can be programed efficiently.

The upside of this representation lies in the lack of limitation on the size of the patterns. In go, many moves are played in the neighborhood of stones and edges. To simplify the work we constrained the patterns to advise one move in its center only, and not elsewhere.

Other go pattern representations usually contain "don't care" points [2], i. e. points called '#' that can be either '+', 'O', '@' or 'e'. The K-nearest-neighbor representation does not explicitly contain such points. However, the points situated far from the center of a pattern are "don't care" points. Thus, the K-nearest-neighbor representation implicitly contains such points. Besides, not managing these points explicitly simplifies the pattern matching algorithm. Moreover, because replacing a pattern containing a "don't care" point by four patterns containing one of the four explicit values (black, white, empty, edge) is still possible, this representation does not lose generality provided that the memory space is sufficient.

Pattern-matching must deal with the symmetries, rotations and black and white inversions of board pieces in a way or another. Upon the 16 patterns that belong the same equivalence class when considering the symmetries, rotations, and black and white inversions, a first approach to match a given pattern with a piece of board consists in storing one pattern only in memory, and let the pattern-matching algorithm compare the actual piece of board with the 16 patterns equivalent to the given pattern. The other approach consists in storing explicitly the 16 patterns equivalent to a pattern, and lightens the pattern-matcher algorithm with the symmetries, rotations and black and white inversions. In our first release, not yet concerned with memory limitation but with fast development, we have chosen the second approach.

3.2 Creating patterns

For a given set of games, the creating process is straightforward. It corresponds to the following pseudo-code:

```
createPatterns()     {
  For k = 1 up to Kmax
    For each game
      For each move i of the game
        createPattern(k,     i);
}
```

If the pattern does not exist yet, the function $createPattern(k, i)$ creates the pattern centered on i with k neighbors following the predefined order between intersections. The patterns are stored in a tree whose nodes have four children: the node "if empty", the node "if black", the node "if white" and the node "if edge". Thanks to such a tree pattern-matching is efficient.

4 Bayesian generation

This section describes the bayesian aspect of the work, classical in classification tasks [1]. First, we define and name the relevant probabilities with the bayesian properties of a pattern. We show how we compute the pattern probabilities. Finally, we discuss the way our system eliminates bad patterns.

4.1 Definitions

P names a probablility. i names either an intersection or a move being played on it. p names a pattern. $P(p)$ is the probability that pattern p matches on an arbitrary intersection. $P(i)$ is the probability that the move is being played on i. $P(i, p)$ is the probability that the move is being played on i and that pattern p matches on i. $P(i|p)$ is the probability that the move is being played on i given that pattern p matches on i. Finally, $P(p|i)$ is the probability that pattern p matches on i given that the move is being played on i. $P(i)$ and $P(p)$ are prior probabilities. $P(i|p)$ and $P(p|i)$ are posterior probabilities.

At playing-time, the underlying idea remains to perform pattern-matching on every i intersection of the board, and to use $P(i|p)$ as an estimation of the urgency of the move played on i. At building-time, we adopt a frequentist approach, a probability that an event arises is approximated

by the number of times that the event arises divided by the number of tests performed. We say that a pattern is *frequent* when $P(p)$ is high, *good* when $P(i|p)$ is high, and *useful* when $P(p|i)$ is high. Therefore, we defined a class FORME_K whose bayesian properties are specified below. The term "static" is a C++ keyword which refers to a feature of the whole class.

```
class Forme_k {
  static int n_test;
  static int n_play;
  int n_match;                    // p.n_match
  int n_play;                     // p.n_play
  static float p_play;            // P(i)
  float p_match;                  // P(p)
  float p_play_given_match;       // P(i|p)
  float p_match_given_play;       // P(p|i)
  ...
};
```

The formula to approximate the probabilities by counting the events are:

```
P(i)   = n_play/n_test;
P(p)   = p.n_match/n_test;
P(i|p) = p.n_play/p.n_match;
P(p|i) = p.n_play/n_play;
```

With such definitions, the Bayes formula remains valid.

4.2 Computing the pattern probabilities

For a given set of games and a given set of patterns, the bayesian process corresponds to the following pseudo-code:

```
computeProbabilities()      {
  n_play = n_test = 0;
  For each pattern p,
    p.n_match = p.n_play = 0;
  For each game {
    For each move of the game {
      n_play++;
      For each intersection i,
        test(i);
    }
  }
  For each pattern p,
    p.p_play = p.n_play/p.n_match;
}

test(i) {
  n_test++;
  patternMatching();
  For each matched pattern p on i {
    p.n_match++;
    if move played on i then
      p.n_play++;
  }
}
```

A test on an i intersection on a given position of a given game answers the two questions: is the move played on i, and which patterns are matching on i ? On 19x19 boards, 200 or 300 tests are performed by position and a game lasts approximately 200 moves, thus 50,000 tests are performed during one game. With the 2,000 professional games we currently have, we reach about 100,000,000 tests.

4.3 Eliminating bad patterns

The underlying idea of this subsection consists in eliminating the patterns which are not good enough or computed with too low a confidence level. First, because the low playing probability patterns are less interesting than the high playing probability patterns, the extracting process only kept p patterns such as $P(i|p) > 0.01$. Second, we can estimate the confidence on the computed probabilities P (being $P(i|p)$) computed at building-time. Basic statistics [13] yield $\sigma = \sqrt{P(1-P)}$. For most patterns we have $P << 1$, thus $\sigma \approx \sqrt{P}$. The relevant quantity to assess the confidence level is $s(i|p) = \sigma/\sqrt{N_{match}} = \sqrt{N_{play}}/N_{match}$

Then, the system may eliminate p patterns such as $P(i|p) < A \times s(i|p)$. However, in practice, we decided to apply this rule only when our set of games is larger. With such pragmatic decision, our system extracted K-dependent databases, K being the maximal number of neighbors considered during generation. Table 1 provides the number of patterns generated for some values of K.

K	6	9	15
patterns encountered	200,000	700,000	2,400,000
patterns kept	8,000	27,000	85,000

Table 1: Number of generated patterns for $K = 6, 9, 15$.

In the following experiments, the value of K is set to 15. Because, at this stage we only have 2,000 professional games, the number of patterns kept is linear in the number of games.

5 Experiments

There are two possible ways of using FORME_K: direct play as an opening book without MC verification (subsection 5.1), and integration with MC verification (subsection 5.2).

For each way, we set up experiments to assess the effect of FORME_K. One experiment consists in a 100-game match between the program to be assessed, KATIA, and the experiment reference program, the 2004 release of INDIGO that attended the 2004 Computer Olympiads, each program playing 50 games with Black. The result of one experiment is a set of relative scores provided by a table assuming that KATIA is the max player. A positive number in a cell corresponds to a successful integration. Given that the standard deviation of 19x19 games played by our programs is roughly 75 points, 100 games enable our experiments to lower σ down to 7.5 points (only) and to obtain a 95% confidence interval of which the radius equals 2σ,

i.e., 15 points. We have used 2.4 GHz computers. INDIGO and KATIA both use the handcrafted databases FORME_B, FORME_C, FORME_M and FORME_3X3. Besides, KATIA uses FORME_K.

5.1 Using Forme_K as an opening book without MC verification

The initial idea was to replace FORME_B and FORME_C by FORME_K. FORME_B and FORME_C are parts of the MC preprocessor, implying that their moves are verified by the MC module. However, to assess the effect of the FORME_K more frankly and quickly, we decided that KATIA will directly play the move advised by FORME_K, using it as an opening book without MC verification. Figure 1 yields the first 40 moves of a go opening self-played by KATIA in such a way. This opening is played with a very good style indeed, and may appear as very smart by human go players. One should believe that this opening is produced by strong human players. In fact, this appearance can be misleading. Against weak opponents that do not play with professional style, KATIA would not be able to confirm the good behavior shown by Figure 1. As shown in the following, the result of this book approach decreases rapidly after move forty. However, this good opening reflects the strength of a bayesian approach on a K-nearest-neighbor representation in go.

After this qualitative assessment, it is now important to assess FORME_K in terms of quantitative results. Table 2 shows the results between KATIA(K, BEGIN) and INDIGO. During the first $begin$ moves, the move played by KATIA is the move advised by FORME_K. After the opening stage, KATIA keeps using the same move selection as INDIGO.

	6	9	15
20	-9	-2	-5
30	-9	-8	+3
40	-30	-10	-6

Table 2: Average result of KATIA(K, BEGIN) against INDIGO for k = 6, 9, 15 and begin = 20, 30, 40.

As expected, the result is improved when K increases, but until $K = 15$ the results remain negative. At this point, KATIA(K=15, BEGIN=30) is the only positive result. We wondered why the results were not satisfactory. In fact, replacing FORME_B and FORME_C by FORME_K was misleading. Actually, FORME_B and FORME_C do have importance in INDIGO, and it was better to keep them in KATIA as well.

Thus, giving up the initial idea to replace FORME_B and FORME_C by FORME_K, we have added FORME_K in KATIA and we have kept both FORME_B and FORME_C. This addition gave better results provided by Table 3. Moreover, the correct value of $begin$ remained to be determined. First, this result shows that KATIA was not weaker than INDIGO. Second, because KATIA(BEGIN=40) plays instantly during the first forty moves of the game, she saves about 30% of the thinking process throughout one game. Therefore, at this point, the integration already showed a positive effect in terms of both playing level and time.

0	10	20	30	40	50
	+1.0	+5.4	+0.6	+3.5	-11.1

Table 3: Average result of KATIA(BEGIN) also using FORME_B and FORME_C against INDIGO for begin = 0, 10, 20, 30, 40, and 50.

5.2 Integrating Forme_K with MC verification

Within INDIGO, the knowledge-based preprocessor uses several databases along with the conceptual evaluation function to select ns moves for the MC module. The idea developed by this subsection is then to integrate FORME_K within the MC preprocessor.

We name KATIA(NK) the release of KATIA that selects nk moves with FORME_K, and selects $ns - nk$ moves with the existing preprocessor, finally providing them to the MC module for verification. If both FORME_K and the existing pre-processor select the same moves, extra moves are taken from the existing preprocessor. Moves selected by FORME_K are filtered by heuristics using the tactical results available in the preprocessor. In 2004, INDIGO used $ns = 7$. Because life and death knowledge is necessary to a go program, and because FORME_K does not contain life and death knowledge, we expected results for nk to vary from 0 up to 4, to keep at least 3 moves concerning life and death. Table 4 shows these results.

	0	10	20	30	40	50
0		+1.0	+5.4	+0.6	+3.5	-11.1
1	-1.3	+4.9	+1.8	+3.6	+2.1	-2.9
2	+9.6	+15.8	+10.0	+6.1	+5.8	-13.0
3	+3.9	+8.6	-0.7	-1.5	-6.7	-20.1
4	+5.1	-2.5	+6.7	-4.2	+1.0	-16.9

Table 4: Average result of KATIA(BEGIN, NK) against INDIGO for begin = 0, 10, 20, 30, 40 and 50 and for nk = 0, 1, 2, 3, 4.

Some of these results are then clearly positive. KATIA(BEGIN=10,NK=2) averages about fifteen points better than INDIGO. It is interesting to comment upon the KATIA(NK=2) results. First, the KATIA(BEGIN=0,NK=2) result shows the effect of integrating FORME_K with MC verification independently of using FORME_K as an opening book. It is interesting to point out the 10 point improvement resulting from the insertion of 2 FORME_K moves within the 7 moves selected by the pre-processor. This fact reflects the lack of patterns within the hand-crafted databases, and the presence of these patterns within FORME_K. Second, the KATIA(BEGIN=10,NK=2) result is also amazing. It shows that KATIA improves by 5 points on average by playing the first 5 moves by using FORME_K as an opening book. The first 5 moves corresponds to the very early beginning in go standards. This result shows that appropriate first 5 moves can already show a positive effect. Third, the

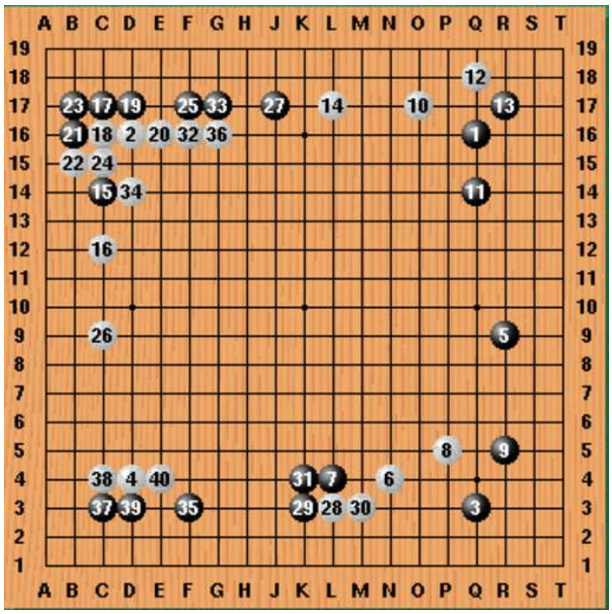


Figure 1: Katia first 40 moves during a self-play opening

KATIA(BEGIN=20,NK=2) result corresponds to a compromise between time and average playing level. The playing level is about the same than KATIA(BEGIN=0,NK=2), but about 20% of thinking time is saved by playing the first 10 moves instantly. Finally, KATIA(BEGIN=30 OR 40,NK=2) can be considered as possible compromises between time and playing level, but due to its very negative result KATIA(BEGIN=50,NK=2) cannot be. Besides, the results can be commented by column. The best results are obtained for $nk = 2$. FORME_K does not include any life and death information, while the conceptual evaluation function does. It is then normal to observe that a module including life and death is more useful (it provides 5 moves upon 7) than FORME_K which provides 2 moves upon 7.

Finally, by copying the appropriate release of KATIA into INDIGO, may be KATIA(BEGIN=20,NK=2), we can conclude that FORME_K can be successfully integrated into INDIGO. However, we have tested KATIA(BEGIN=20,NK=2) against GNU Go 3.2 [10] and no improvement was observed.

6 Perspectives

We plan to re-generate the FORME_K database with a number of games greater than 2,000. For instance, the GoGod CDROM contains about 30,000 professional games, and has the appropriate size for the next assessment. This regeneration would refine the probabilities estimation, and consequently the move urgencies at playing time. An improvement should be observed. Although numerous 9x9 and 13x13 professional games are not massively available, checking the non-regression results of KATIA on 9x9 or 13x13 boards is a mandatory task. Taking the symmetries, rotations and black and white inversions into account within the pattern matching and probability estimations is also an important perspective that will enhance the confidence level of probability estimations. Besides, we also plan to extend the patterns by allowing moves being played not only in the center of the pattern but also on the intersections situated near the center.

Moreover, we have two other interesting perspectives: first, integrating a relevant subset of FORME_K within the conceptual evaluation function module to replace FORME_M, and, second, integrating another appropriate subset of FORME_K within the MC engine to replace FORME_3X3. The first integration should be a difficult knowledge engineering task because FORME_M is used by the conceptual evaluation function in a very intricated way. The second one should be possible provided the patterns are limited to a pre-defined neighborhood of the center of the pattern, because speed considerations are crucial within the MC engine. Another interesting challenge of this second integration is the off-line computation of move urgencies. This can be done either by using a function of the probabilities computed by browsing recorded games, or by reinforcement learning technique [18].

7 Conclusion

We have suggested a method to extract patterns automatically from professional recorded games. This method uses basic probability estimations, and does not assume any domain-dependent knowledge. To this extent, it is a good continuation of a MC go program. The representation used is the K-nearest-neighbor representation. The bayesian generation of K-nearest-neighbor patterns gives an opening book that produces very good openings indeed. This work experimentally demonstrates that the strength of this method lies in the K-nearest-neighbor representation adapted to the game of go. Its weakness lies in its lack of life and death understanding, life and death being the cornerstone of any strong go program. Thus, this approach cannot

be used as such, and must be combined with other existing approaches.

We have integrated the database built along such a method into the go playing program INDIGO. The results are positive. Adding the database within the preprocessor of the MC module enables INDIGO to improve by 15 points on 19x19 boards on average, which is significant in go standards. Furthermore, in the opening of games, the quality of the twenty or thirty first moves provided by the database allows INDIGO to play these moves directly without MC verification. Consequently, 20% of the thinking time of INDIGO can be saved, allowing room for other future improvements.

Bibliography

[1] C. Bishop. *Neural networks and pattern recognition*. Oxford University Press, 1995.

[2] M. Boon. A pattern matcher for Goliath. *Computer Go*, 13:13–23, 1990.

[3] B. Bouzy. Go patterns generated by retrograde analysis. In *Computer Olympiad Workshop*, Maastricht, 2001.

[4] B. Bouzy. Associating knowledge and Monte Carlo approaches within a go program. In *7th Joint Conference on Information Sciences*, pages 505–508, Raleigh, 2003.

[5] B. Bouzy. Mathematical morphology applied to computer go. *International Journal of Pattern Recognition and Artificial Intelligence*, 17(2):257–268, March 2003.

[6] B. Bouzy. The move decision process of Indigo. *International Computer Game Association Journal*, 26(1):14–27, March 2003.

[7] B. Bouzy. Indigo home page. www.math-info.univ-paris5.fr/∼bouzy/INDIGO.html, 2004.

[8] B. Bouzy and T. Cazenave. Computer go: an AI oriented survey. *Artificial Intelligence*, 132:39–103, 2001.

[9] B. Bouzy and B. Helmstetter. Monte Carlo go developments. In Ernst A. Heinz H. Jaap van den Herik, Hiroyuki Iida, editor, *10th Advances in Computer Games*, pages 159–174, Graz, 2003. Kluwer Academic Publishers.

[10] D. Bump. Gnugo home page. www.gnu.org/software/gnugo/devel.html, 2003.

[11] T. Cazenave. Automatic acquisition of tactical go rules. In *3rd Game Programming Workshop in Japan*, pages 10–19, Hakone, 1996.

[12] T. Cazenave. Generation of patterns with external conditions for the game of go. In B. Monien H.J. van den Herik, editor, *Advances in Computer Games*, volume 9, University of Limburg, Maastricht, 2001.

[13] CISIA CERESTA, editor. *Aide-mémoire statistique*. 1999.

[14] F. de Groot. Moyo go studio. www.moyogo.com, 2004.

[15] M. Müller. Computer go. *Artificial Intelligence*, 134:145–179, 2002.

[16] M. Müller. Position evaluation in computer go. *ICGA Journal*, 25(4):219–228, December 2002.

[17] J. Schaeffer and J. van den Herik. Games, Computers, and Artificial Intelligence. *Artificial Intelligence*, 134:1–7, 2002.

[18] R. Sutton and A. Barto. *Reinforcement Learning: an introduction*. MIT Press, 1998.

[19] E. van der Werf, J. Uiterwijk, E. Postma, and J. van den Herik. Local move prediction in Go. In Yngvi Björnsson J. Schaeffer, M. Müller, editor, *Computers and Games*, volume 2883 of *Lecture Notes in Computer Science*, pages 393–412. Springer, 2002.

[20] E. van der Werf, J. Uiterwijk, and J. van den Herik. Learning to score final positions in the game of go. In H. Jaap van den Herik, Hiroyuki Iida, and Ernst A. Heinz, editors, *Advances in Computer Games, Many Games, Many Challenges*, volume 10, pages 143–158. Kluwer Academic Publishers, 2003.

[21] E. van der Werf, M. Winands, J. van den Herik, and J. Uiterwijk. Learning to predict life and death from go game records. In *7th Joint Conference on Information Sciences*, pages 501–504, Raleigh, 2003.

Evolving Neural Network Agents in the NERO Video Game

Kenneth O. Stanley
Department of Computer Sciences
The University of Texas at Austin
Austin, TX 78712 USA
kstanley@cs.utexas.edu

Bobby D. Bryant
Department of Computer Sciences
The University of Texas at Austin
Austin, TX 78712 USA
bdbryant@cs.utexas.edu

Risto Miikkulainen
Department of Computer Sciences
The University of Texas at Austin
Austin, TX 78712 USA
risto@cs.utexas.edu

Abstract- In most modern video games, character behavior is scripted; no matter how many times the player exploits a weakness, that weakness is never repaired. Yet if game characters could learn through interacting with the player, behavior could improve during gameplay, keeping it interesting. This paper introduces the real-time NeuroEvolution of Augmenting Topologies (rtNEAT) method for evolving increasingly complex artificial neural networks in *real time*, as a game is being played. The rtNEAT method allows agents to change and improve during the game. In fact, rtNEAT makes possible a new genre of video games in which the player *teaches* a team of agents through a series of customized training exercises. In order to demonstrate this concept in the NeuroEvolving Robotic Operatives (NERO) game, the player trains a team of robots for combat. This paper describes results from this novel application of machine learning, and also demonstrates how multiple agents can evolve and adapt in video games like NERO in real time using rtNEAT. In the future, rtNEAT may allow new kinds of educational and training applications that adapt online as the user gains new skills.

1 Introduction

The world video game market in 2002 was between $15 billion and $20 billion, larger than even that of Hollywood (Thurrott 2002). Video games have become a facet of many people's lives and the market continues to expand. Because there are millions of players and because video games carry perhaps the least risk to human life of any real-world application, they make an excellent testbed for techniques in artificial intelligence and machine learning (Laird and van Lent 2000). In fact, Fogel et al. (2004) argue that such techniques can potentially both increase the longevity of video games and decrease their production costs.

One of the most compelling yet least exploited technologies in the video game industry is machine learning. Thus, there is an unexplored opportunity to make video games more interesting and realistic, and to build entirely new genres. Such enhancements may have applications in education and training as well, changing the way people interact with their computers.

In the video game industry, the term *Non-player-character* (NPC) refers to an autonomous computer-controlled agent in the game. This paper focuses on training NPCs as intelligent agents, and the standard AI term *agents* is therefore used to refer to them. The behavior of agents in current games is often repetitive and predictable. In most video games, scripts cannot learn or adapt to control the agents: Opponents will always make the same moves and the game quickly becomes boring. Machine learning could potentially keep video games interesting by allowing agents to change and adapt. However, a major problem with learning in video games is that if behavior is allowed to change, the game content becomes unpredictable. Agents might learn idiosyncratic behaviors or even not learn at all, making the gaming experience unsatisfying. One way to avoid this problem is to train agents offline, and then freeze the results into the final game. However, if behaviors are frozen before the game is released, agents cannot adapt and change in response to the tactics of particular players.

If agents are to adapt and change in real-time, a powerful and reliable machine learning method is needed. This paper describes a novel game built around a real-time enhancement of the NeuroEvolution of Augmenting Topologies method (NEAT; Stanley and Miikkulainen 2002b, 2004). NEAT evolves increasingly complex neural networks, i.e. it *complexifies*. Real-time NEAT (rtNEAT) is able to complexify neural networks *as the game is played*, making it possible for agents to evolve increasingly sophisticated behaviors in real time. Thus, agent behavior improves visibly during gameplay. The aim is to show that machine learning is indispensable for some kinds of video games to work, and to show how rtNEAT makes such an application possible.

In order to demonstrate the potential of rtNEAT, the Digital Media Collaboratory (DMC) at the University of Texas at Austin initiated the NeuroEvolving Robotic Operatives (NERO) project in October of 2003 (`http://dev.ic2.org/nero_public`). This project is based on a proposal for a game based on rtNEAT developed at the *2nd Annual Game Development Workshop on Artificial Intelligence, Interactivity, and Immersive Environments* in Austin, TX (presentation by Kenneth Stanley, 2003). The idea was to create a game in which learning is *indispensable*, in other words, without learning NERO could not exist as a game. In NERO, the player takes the role of a trainer, teaching skills to a set of intelligent agents controlled by rtNEAT. Thus, NERO is a powerful demonstration of how machine learning can open up new possibilities in gaming and allow agents to adapt.

This paper describes rtNEAT and NERO, and reviews results from the first year of this ongoing project. The next section briefly reviews learning methods for games. NEAT is then described, including how it was enhanced to create rtNEAT. The last sections describe NERO and the current status and performance of the game.

2 Background

This section reviews several machine learning techniques that can be used in games, and explains why *neuroevolution* (NE), i.e. the artificial evolution neural networks using a genetic algorithm, is the ideal method for real-time learning in NERO. Because agents in NERO need to learn online as the game is played, predetermined training targets are usually not available, ruling out supervised techniques such as backpropagation (Rumelhart et al. 1986) and decision tree learning (Utgoff 1989).

Traditional reinforcement learning (RL) techniques such as Q-Learning (Watkins and Dayan 1992) and Sarsa(λ) with a Case-Based function approximator (SARSA-CABA; Santamaria et al. 1998) can adapt in domains with sparse feedback (Kaelbling et al. 1996; Sutton and Barto 1998) and thus can be applied to video games as well. These techniques learn to predict the long-term reward for taking actions in different states by exploring the state space and keeping track of the results. However, video games have several properties that pose significant challenges to traditional RL:

1. **Large state/action space**. Since games usually have several different types of objects and characters, and many different possible actions, the state/action space that RL must explore is high-dimensional. Not only does this pose the usual problem of encoding a high-dimensional space (Sutton and Barto 1998), but in a real-time game there is the additional challenge of checking the value of every possible action on every game tick for every agent in the game.
2. **Diverse behaviors**. Agents learning simultaneously should not all converge to the same behavior because a homogeneous population would make the game boring. Yet since RL techniques are based on convergence guarantees and do not explicitly maintain diversity, such an outcome is likely.
3. **Consistent individual behaviors**. RL depends on occasionally taking a random action in order to explore new behaviors. While this strategy works well in offline learning, players do not want to see an individual agent periodically making inexplicable and idiosyncratic moves relative to its usual behavior.
4. **Fast adaptation**. Players do not want to wait hours for agents to adapt. Yet a complex state/action representation can take a long time to learn. On the other hand, a simple representation would limit the ability to learn sophisticated behaviors. Thus, choosing the right representation is difficult.
5. **Memory of past states**. If agents remember past events, they can react more convincingly to the present situation. However, such memory requires keeping track of more than the current state, ruling out traditional Markovian methods.

While these properties make applying traditional RL techniques difficult, NE is an alternative RL technique that can meet each requirement: (1) NE works well in high-dimensional state spaces (Gomez and Miikkulainen 2003), and only produces a single requested action without checking the values of multiple actions. (2) Diverse populations can be explicitly maintained (Stanley and Miikkulainen 2002b). (3) The behavior of an individual during its lifetime does not change. (4) A *representation* of the solution can be evolved, allowing simple behaviors to be discovered quickly in the beginning and later complexified (Stanley and Miikkulainen 2004). (5) Recurrent neural networks can be evolved that utilize memory (Gomez and Miikkulainen 1999). Thus, NE is a good match for video games.

The current challenge is to achieve evolution in *real time*, as the game is played. If agents could be evolved in a smooth cycle of replacement, the player could interact with evolution during the game and the many benefits of NE would be available to the video gaming community. This paper introduces such a real-time NE technique, rtNEAT, which is applied to the NERO multi-agent continuous-state video game. In NERO, agents must master both motor control and higher-level strategy to win the game. The player acts as a trainer, teaching a team of robots the skills they need to survive. The next section reviews the NEAT neuroevolution method, and how it can be enhanced to produce rtNEAT.

3 Real-time NeuroEvolution of Augmenting Topologies (rtNEAT)

The rtNEAT method is based on NEAT, a technique for evolving neural networks for complex reinforcement learning tasks using a genetic algorithm (GA). NEAT combines the usual search for the appropriate network weights with *complexification* of the network structure, allowing the behavior of evolved neural networks to become increasingly sophisticated over generations. This approach is highly effective: NEAT outperforms other neuroevolution (NE) methods e.g. on the benchmark double pole balancing task (Stanley and Miikkulainen 2002a,b). In addition, because NEAT starts with simple networks and expands the search space only when beneficial, it is able to find significantly more complex controllers than fixed-topology evolution, as demonstrated in a robotic strategy-learning domain (Stanley and Miikkulainen 2004). These properties make NEAT an attractive method for evolving neural networks in complex tasks such as video games.

Like most GAs, NEAT was originally designed to run *offline*. Individuals are evaluated one or two at a time, and after the whole population has been evaluated, a new population is created to form the next generation. In other words, in a normal GA it is not possible for a human to interact with the multiple evolving agents *while they are evolving*. This section first briefly reviews the original offline NEAT method, and then describes how it can be modified to make it possible for players to interact with evolving agents in real time. See e.g. Stanley and Miikkulainen (2002a,b, 2004) for a complete description of NEAT.

NEAT is based on three key ideas. First, evolving network structure requires a flexible genetic encoding. Each genome includes a list of *connection genes*, each of which refers to two *node genes* being connected. Each connection gene specifies the in-node, the out-node, the connection

weight, whether or not the connection gene is expressed (an enable bit), and an *innovation number*, which allows finding corresponding genes during crossover. Mutation can change both connection weights and network structures. Connection weights mutate as in any NE system, with each connection either perturbed or not. Structural mutations, which allow complexity to increase, either add a new connection or a new node to the network. Through mutation, genomes of varying sizes are created, sometimes with completely different connections specified at the same positions.

Each unique gene in the population is assigned a unique innovation number, and the numbers are inherited during crossover. Innovation numbers allow NEAT to perform crossover without the need for expensive topological analysis. Genomes of different organizations and sizes stay compatible throughout evolution, and the problem of matching different topologies (Radcliffe 1993) is essentially avoided.

Second, NEAT speciates the population, so that individuals compete primarily within their own niches instead of with the population at large. This way, topological innovations are protected and have time to optimize their structure before competing with other niches in the population. The reproduction mechanism for NEAT is *explicit fitness sharing* (Goldberg and Richardson 1987), where organisms in the same species must share the fitness of their niche, preventing any one species from taking over the population.

Third, unlike other systems that evolve network topologies and weights (Gruau et al. 1996; Yao 1999) NEAT begins with a uniform population of simple networks with no hidden nodes. New structure is introduced incrementally as structural mutations occur, and only those structures survive that are found to be useful through fitness evaluations. This way, NEAT searches through a minimal number of weight dimensions and finds the appropriate complexity level for the problem.

In previous work, each of the three main components of NEAT (i.e. historical markings, speciation, and starting from minimal structure) were experimentally ablated in order to demonstrate how they contribute to performance (Stanley and Miikkulainen 2002b). The ablation study demonstrated that all three components are interdependent and necessary to make NEAT work. The next section explains how NEAT can be enhanced to work in real time.

3.1 Running NEAT in Real Time

In NEAT, the population is replaced at each generation. However, in real time, replacing the entire population together on each generation would look incongruous since everyone's behavior would change at once. In addition, behaviors would remain static during the large gaps between generations. Instead, in rtNEAT, a single individual is replaced every few game ticks (as in e.g. (μ,1)-ES; Beyer and Paul Schwefel 2002). One of the worst individuals is removed and replaced with a child of parents chosen from among the best. This cycle of removal and replacement happens continually throughout the game (figure 1). The challenge is to preserve the usual dynamics of NEAT, namely protection of innovation through speciation and complexification.

The main loop in rtNEAT works as follows. Let f_i be

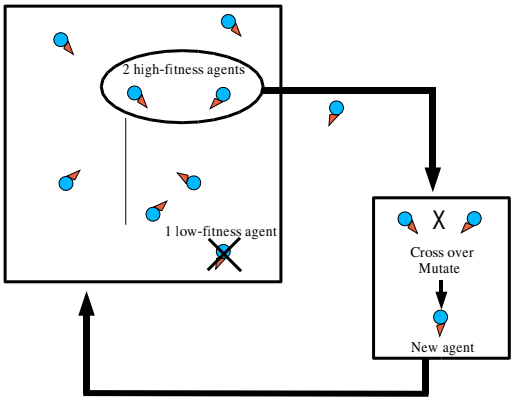


Figure 1: **The main replacement cycle in rtNEAT.** Robot game agents (represented as small circles) are depicted playing a game in the large box. Every few ticks, two high-fitness robots are selected to produce an offspring that replaces another of lower fitness. This cycle of replacement operates continually throughout the game, creating a constant turnover of new behaviors.

the fitness of individual i. Fitness sharing adjusts it to $\frac{f_i}{|S|}$, where $|S|$ is the number of individuals in the species. In other words, fitness is reduced proportionally to the size of the species. This adjustment is important because selection in rtNEAT must be based on adjusted fitness rather than original fitness in order to maintain the same dynamics as NEAT. In addition, because the number of offspring assigned to a species in NEAT is based on its average fitness $\overline{F}$, this average must always be kept up-to-date. Thus, after every n ticks of the game clock, rtNEAT performs the following operations:

1. Remove the agent with the worst *adjusted* fitness from the population assuming one has been alive sufficiently long so that it has been properly evaluated.
2. Re-estimate $\overline{F}$ for all species
3. Choose a parent species to create the new offspring
4. Adjust compatibility threshold C_t dynamically and *reassign* all agents to species
5. Place the new agent in the world

Each of these steps is discussed in more detail below.

3.1.1 Step 1: Removing the worst agent

The goal of this step is to remove a poorly performing agent from the game, hopefully to be replaced by something better. The agent must be chosen carefully to preserve speciation dynamics. If the agent with the worst *unadjusted* fitness were chosen, fitness sharing could no longer protect innovation because new topologies would be removed as soon as they appear. Thus, the agent with the worst *adjusted* fitness should be removed, since adjusted fitness takes into account species size, so that new smaller species are not removed as soon as they appear.

It is also important not to remove agents that are too young. In original NEAT, *age* is not considered since networks are generally all evaluated for the same amount of

time. However, in rtNEAT, new agents are constantly being born, meaning different agents have been around for different lengths of time. It would be dangerous to remove agents that are too young because they have not played for long enough to accurately assess their fitness. Therefore, rtNEAT only removes agents who have played for more than the minimum amount of time m.

3.1.2 Step 2: Re-estimating $\overline{F}$

Assuming there was an agent old enough to be removed, its species now has one less member and therefore its average fitness $\overline{F}$ has likely changed. It is important to keep $\overline{F}$ up-to-date because $\overline{F}$ is used in choosing the parent species in the next step. Therefore, rtNEAT needs to re-estimate $\overline{F}$.

3.1.3 Step 3: Choosing the parent species

In original NEAT the number of offspring n_k assigned to species k is $\frac{\overline{F_k}}{\overline{F}_{\text{tot}}}|P|$, where $\overline{F_k}$ is the average fitness of species k, $\overline{F}_{\text{tot}}$ is the sum of all the average species' fitnesses, and $|P|$ is the population size.

This behavior needs to be approximated in rtNEAT even though n_k cannot be assigned explicitly (since only one offspring is created at a time). Given that n_k is proportional to $\overline{F}$, the parent species can be chosen probabilistically using the same relationship:

$$Pr(S_k) = \frac{\overline{F_k}}{\overline{F}_{\text{tot}}}. \quad (1)$$

The probability of choosing a given parent species is proportional to its average fitness compared to the total of all species' average fitnesses. Thus, over the long run, the expected number of offspring for each species is proportional to n_k, preserving the speciation dynamics of original NEAT.

3.1.4 Step 4: Dynamic Compatibility Thresholding

Networks are placed into a species in original NEAT if they are compatible with a representative member of the species. rtNEAT attempts to keep the number of species constant by adjusting a threshold, C_t, that determines whether an individual is compatible with a species' representative. When there are too many species, C_t is increased to make species more inclusive; when there are too few, C_t is decreased to be stricter. An advantage of this kind of *dynamic compatibility thresholding* is that it keeps the number of species relatively stable.

In rtNEAT changing C_t alone cannot immediately affect the number of species because most of the population simply remains where they are. Just changing a variable does not cause anything to move to a different species. Therefore, after changing C_t in rtNEAT, the entire population must be reassigned to the existing species based on the new C_t. As in original NEAT, if a network does not belong in any species a new species is created with that network as its representative.[1]

[1]Depending on the specific game, C_t does not necessarily need to be adjusted and species reorganized as often as every replacement. The number of ticks between adjustments is chosen by the game designer.

3.1.5 Step 5: Replacing the old agent with the new one

Since an individual was removed in step 1, the new offspring needs to replace it. How agents are replaced depends on the game. In some games, the neural network can be removed from a body and replaced without doing anything to the body. In others, the body may have died and need to be replaced as well. rtNEAT can work with any of these schemes as long as an old neural network gets replaced with a new one.

Step 5 concludes the steps necessary to approximate original NEAT in real-time. However, there is one remaining issue. The entire loop should be performed at regular intervals, every n ticks: How should n be chosen?

3.1.6 Determining Ticks Between Replacements

If agents are replaced too frequently, they do not live long enough to reach the minimum time m to be evaluated. For example, imagine that it takes 100 ticks to obtain an accurate performance evaluation, but that an individual is replaced in a population of 50 on every tick. No one ever lives long enough to be evaluated and the population always consists of only new agents. On the other hand, if agents are replaced too infrequently, evolution slows down to a pace that the player no longer enjoys.

Interestingly, the appropriate frequency can be determined through a principled approach. Let I be the fraction of the population that is too young and therefore cannot be replaced. As before, n is the ticks between replacements, m is the minimum time alive, and $|P|$ is the population size. A *law of eligibility* can be formulated that specifies what fraction of the population can be expected to be ineligible once evolution reaches a steady state (i.e. after the first few time steps when no one is eligible):

$$I = \frac{m}{|P|n}. \quad (2)$$

According to Equation 2, the larger the population and the more time between replacements, the lower the fraction of ineligible agents. Based on the law, rtNEAT can decide on its own how many ticks n should lapse between replacements for a preferred level of ineligibility, specific population size, and minimum time between replacements:

$$n = \frac{m}{|P|I}. \quad (3)$$

It is best to let the user choose I because in general it is most critical to performance; if too much of the population is ineligible at one time, the mating pool is not sufficiently large. Equation 3 allows rtNEAT to determine the correct number of ticks between replacements n to maintain a desired eligibility level. In NERO, 50% of the population remains eligible using this technique.

By performing the right operations every n ticks, choosing the right individual to replace and replacing it with an offspring of a carefully chosen species, rtNEAT is able to replicate the dynamics of NEAT in real-time. Thus, it is now possible to deploy NEAT in a real-time game and interact with complexifying agents as they evolve. The next section describes such a game.

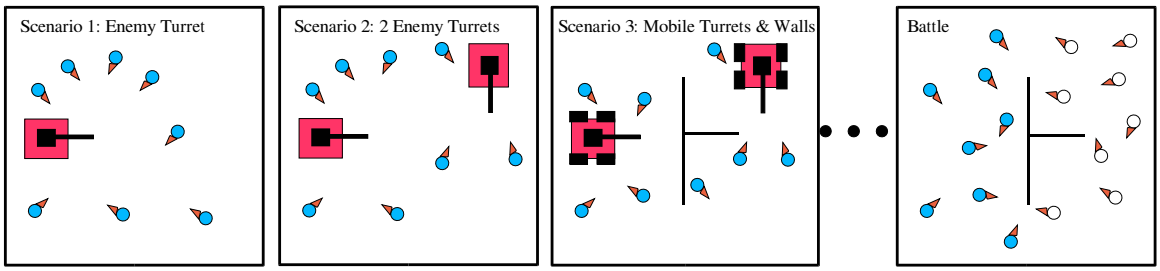


Figure 2: **A turret training sequence.** The figure depicts a sequence of increasingly difficult and complicated training exercises in which the agents attempt to attack turrets without getting hit. In the first exercise there is only a single turret but more turrets are added by the player as the team improves. Eventually walls are added and the turrets are given wheels so they can move. Finally, after the team has mastered the hardest exercise, it is deployed in a real battle against another team.

4 NeuroEvolving Robotic Operatives (NERO)

NERO is representative of a new genre that is only possible through machine learning. The idea is to put the player in the role of a *trainer* or a *drill instructor* who teaches a team of agents by designing a curriculum.

In NERO, the learning agents are simulated robots, and the goal is to train a team of robots for military combat. The robots begin the game with no skills and only the ability to learn. In order to prepare for combat, the player must design a sequence of training exercises and goals specified with a set of sliders. Ideally, the exercises are increasingly difficult so that the team can begin by learning a foundation of basic skills and then gradually building on them (figure 2). When the player is satisfied that the team is prepared, the team is deployed in a battle against another team trained by another player (possibly on the internet), making for a captivating and exciting culmination of training. The challenge is to anticipate the kinds of skills that might be necessary for battle and build training exercises to hone those skills. The next two sections explain how the agents are trained in NERO and how they fight an opposing team in battle.

4.1 Training Mode

The player sets up training exercises by placing objects on the field and specifying goals through several sliders (figure 3). The objects include static enemies, enemy turrets, rovers (i.e. turrets that move), and walls. To the player, the sliders serve as an interface for describing ideal behavior. To rtNEAT, they represent coefficients for fitness components. For example, the sliders specify how much to reward or punish approaching enemies, hitting targets, getting hit, following friends, dispersing, etc. Each individual fitness component is normalized to a Z-score so that each fitness component is measured on the same scale. Fitness is computed as the sum of all these normalized components multiplied by their slider levels. Thus, the player has a natural interface for setting up a training exercise and specifying desired behavior.

Robots have several types of sensors. Although NERO programmers frequently experiment with new sensor configurations, the standard sensors include enemy radars, an "on target" sensor, object rangefinders, and line-of-fire sensors. Figure 4 shows a neural network with the standard set of sensors and outputs. Several enemy radar sensors divide

Figure 3: **Setting up training scenarios.** This screenshot shows items the player can place on the field and sliders used to control behavior. The robot is a stationary enemy turret that turns back and forth as it shoots repetitively. Behind the turret is a wall. The player can place turrets, other kinds of enemies, and walls anywhere on the training field. On the right is the box containing slider controls. These sliders specify the player's preference for the behavior the team should try to optimize. For example the "E" icon means "approach enemy," and the descending bar above it specifies that the player wants to punish robots that approach the enemy. The crosshair icon represents "hit target," which is being rewarded. The sliders represent fitness components that are used by rtNEAT. The value of the slider is used by rtNEAT as the coefficient of the corresponding fitness component. Through placing items on the field and setting sliders, the player creates training scenarios where learning takes place.

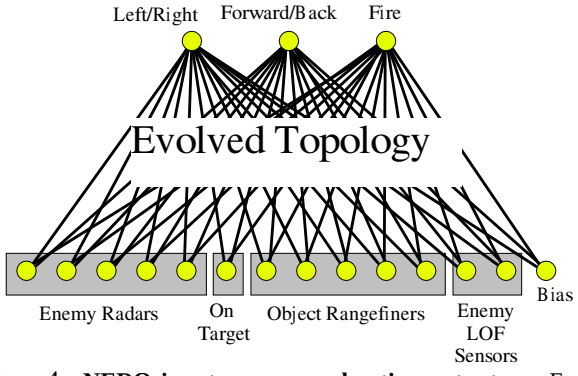


Figure 4: **NERO input sensors and action outputs.** Each NERO robot can see enemies, determine whether an enemy is currently in its line of fire, detect objects and walls, and see the direction the enemy is firing. Its outputs specify the direction of movement and whether or not to fire. This configuration has been used to evolve varied and complex behaviors; other variations work as well and the standard set of sensors can easily be changed.

the 360 degrees around the robot into slices. Each slice activates a sensor in proportion to how close an enemy is within that slice. Rangefinders project rays at several angles from the robot. The distance the ray travels before it hits an object is returned as the value of the sensor. The on-target sensor returns full activation only if a ray projected along the front heading of the robot hits an enemy. The line of fire sensors detect where a bullet stream from the closest enemy is heading. Thus, these sensors can be used to avoid fire. Robots can also be trained with friend radar sensors that allows them to see what each other are doing. The complete sensor set supplies robots with sufficient information to make intelligent tactical decisions.

Training mode is designed to allow the player to set up a training scenario on the field where the robots can continually be evaluated while the worst robot's neural network is replaced every few ticks. Thus, training must provide a standard way for robots to appear on the field in such a way that every robot has an equal chance to prove its worth. To meet this goal, the robots spawn from a designated area of the field called the *factory*. Each robot is allowed a limited time on the field during which its fitness is assessed. When their time on the field expires, robots are transported back to the factory, where they begin another evaluation. Neural networks are only replaced in robots that have been put back in the factory. The factory ensures that a new neural network cannot get lucky by appearing in a robot that happens to be standing in an advantageous position: All evaluations begin consistently in the factory. In addition, the fitness of robots that survive more than one deployment on the field is updated through a diminishing average that gradually forgets deployments from the distant past.

Training begins by deploying 50 robots on the field. Each robot is controlled by a neural network with random connection weights and no hidden nodes, as is the usual starting configuration for NEAT. As the neural networks are replaced in real-time, behavior improves dramatically, and robots eventually learn to perform the task the player sets up. When the player decides that performance has reached a satisfactory level, he or she can save the team in a file. Saved teams can be reloaded for further training in different scenarios, or they can be loaded into battle mode. In battle, they face off against teams trained by an opponent player, as will be described next.

4.2 Battle Mode

In battle mode, the player discovers how training paid off. A battle team of 20 robots is assembled from as many different training teams as desired. For example, perhaps some robots were trained for close combat while others were trained to stay far away and avoid fire. A player can compose a heterogeneous team from both training sessions.

Battle mode is designed to run over a server so that two players can watch the battle from separate terminals on the internet. The battle begins with the two teams arrayed on opposite sides of the field. When one player presses a "go" button, the neural networks obtain control of their robots and perform according to their training. Unlike in training, where being shot does not lead to a robot body being damaged, the robots are actually destroyed after being shot several times in battle. The battle ends when action ceases either because one team is completely eliminated, or because the remaining robots will not fight. The winner is the team with the most robots left standing.

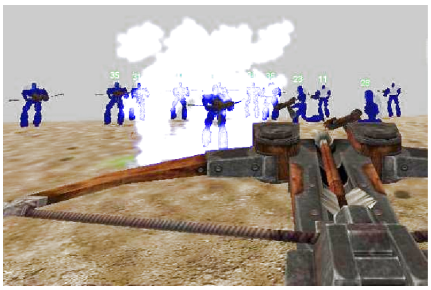


Figure 5: **Running away backwards.** This training screenshot shows several robots backed up against the wall after running backwards and shooting at the enemy, which is being controlled from a first-person perspective by a human trainer using a joystick. Robots learned to run away from the enemy backwards during avoidance training because that way they can shoot as they flee. Running away backwards is an example of evolution's ability to find novel and effective behaviors.

The basic battlefield configuration is an empty pen surrounded by four bounding walls, although it is possible to compete on a more complex field, with walls or other obstacles. Players train their robots and assemble teams for the particular battlefield configuration on which they intend to play. In the experiments described in this chapter, the battlefield was the basic pen.

The next section gives examples of actual NERO training and battle sessions.

5 Playing NERO

Behavior can be evolved very quickly in NERO, fast enough so that the player can be watching and interacting with the system in real time. The game engine Torque, licensed from GarageGames (`http://www.garagegames.com/`), drives NERO's simulated physics and graphics. An important property of the Torque engine is that its physics simulation is slightly nondeterministic, so that the same game is never played twice.

The first playable version of NERO was completed in May of 2004. At that time, several NERO programmers trained their own teams and held a tournament. As examples of what is possible in NERO, this section outlines the behaviors evolved for the tournament, the resulting battles, and the real-time performance of NERO and rtNEAT.

NERO can evolve behaviors very quickly in real-time. The most basic battle tactic is to aggressively seek the enemy and fire. To train for this tactic, a single static enemy was placed on the training field, and robots were rewarded for approaching the enemy. This training required robots to learn to run towards a target, which is difficult since robots start out in the factory facing in random directions. Starting from random neural networks, it takes on average 99.7 seconds for 90% of the robots on the field learn to approach the enemy successfully (10 runs, $sd = 44.5$s).

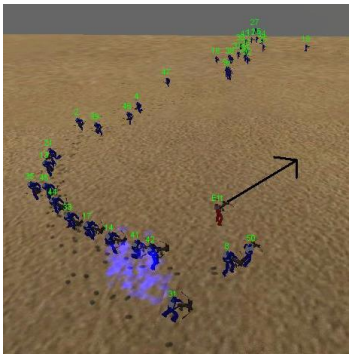


Figure 6: **Avoiding turret fire.** The arrow points in the current direction of the turret fire (the arrow is not part of the NERO display and is only added for illustration). Robots in training learn to run safely around the enemy's line of fire in order to attack. Notice how they loop around the back of the turret and attack from behind. When the turret moves, the robots change their attack trajectory accordingly. Learning to avoid fire is an important battle skill. The conclusion is that rtNEAT was able to evolve sophisticated, nontrivial behavior in real time.

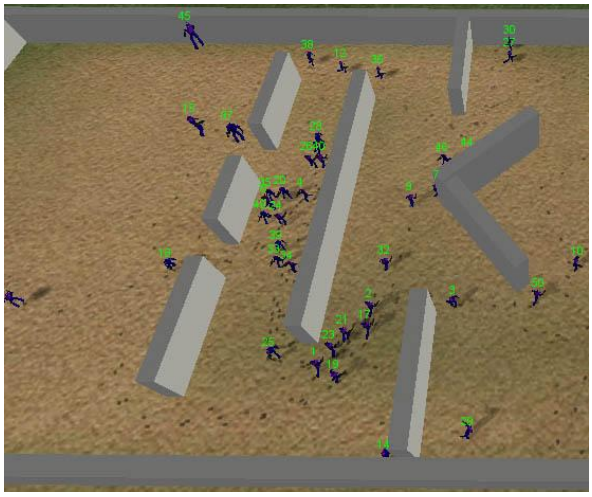


Figure 7: **Navigating a maze.** Incremental training on increasingly complex wall configurations produced robots that could navigate this maze to find the enemy. The robots spawn from the left side of the maze and proceed to an enemy at the right.

Robots were also trained to avoid the enemy. In fact, rtNEAT was flexible enough to *devolve* a population that had converged on seeking behavior into a completely opposite, avoidance, behavior. For avoidance training, players controlled an enemy robot with a joystick and ran it towards robots on the field. The robots learned to back away in order to avoid being penalized for being too near the enemy. Interestingly, robots preferred to run away from the enemy backwards because that way they could still shoot the enemy (figure 5).

By placing a turret on the field and asking robots to approach the turret without getting hit, robots were able to learn to avoid enemy fire (figure 6).

Other interesting behaviors were evolved to test the limits of rtNEAT rather than specifically prepare the troops for battle. For example, robots were trained to run around walls to approach the enemy. As performance improved, players incrementally added more walls until the robots could navigate an entire maze without any path-planning (figure 7)!

In a powerful demonstration of real-time adaptation with implications beyond NERO, robots that were trained to approach a designated location (marked by a flag) through a hallway were then attacked by an enemy controlled by the player (figure 8). After two minutes, the robots learned to take an alternative path through an adjacent hallway in order to avoid the enemy's fire. While such training is used in NERO to prepare robots for battle, the same kind of adaptation could be used in any interactive game to make it more realistic and interesting. Such fast strategic adjustment demonstrates that rtNEAT can be used in existing video game genres as well as in NERO.

In battle, some teams that were trained differently were nevertheless evenly matched, while some training types consistently prevailed against others For example, an aggressive seeking team from the tournament had only a slight advantage over an avoidant team, winning six out of ten battles, losing three, and tying one. The avoidant team runs in a pack to a corner of the field's enclosing wall. Sometimes, if they make it to the corner and assemble fast enough, the aggressive team runs into an ambush and is obliterated. However, slightly more often the aggressive team gets a few shots in before the avoidant team can gather in the corner. In that case, the aggressive team traps the avoidant team with greater surviving numbers. The conclusion is that seeking and running away are fairly well-balanced tactics, neither providing a significant advantage over the other. The interesting challenge of NERO is to conceive strategies that are clearly dominant over others.

One of the best teams was trained by observing a phenomenon that happened consistently in battle. Chases among robots from opposing teams frequently caused robots to eventually reach the field's bounding walls. Particularly for robots trained to avoid turret fire by attacking from behind (figure 6), enemies standing against the wall present a serious problem since it is not possible to go around them. Thus, training a team against a turret with its back against the wall, it was possible to familiarize robots with attacking enemies against a wall. This team learned to hover near the turret and fire when it turned away, but back off quickly when it turned towards them. The wall-based team won the first NERO tournament by using this strategy. The wall-trained team wins 100% of the time against the aggressive seeking team. Thus, it is possible to learn sophisticated tactics that dominate over simpler ones like seek or avoid.

6 Discussion

Participants in the first NERO tournament agreed that the game was engrossing and entertaining. Battles were exciting for all the participants, evoking plentiful clapping and cheering. Players spent hours honing behaviors and assembling teams with just the right combination of tactics.

The success of the first NERO prototype suggests that the rtNEAT technology has immediate potential commercial applications in modern games. Any game in which agent behavior is repetitive and boring can be improved by allowing rtNEAT to at least partially modify tactics in real-

(a) Robots approach flag (b) Player attacks on left (c) Robots learn new approach

Figure 8: **Video game characters adapt to player's actions.** The robots in these screenshots spawn from the top of the screen and must approach the flag (circled) at the bottom left. White arrows point in the direction of mass motion. (a) The robots first learn to take the left hallway since it is the shortest path to the flag. (b) A human player (identified by a square) attacks inside the left hallway and decimates the robots. (c) Even though the left hallway is the shortest path to the flag, the robots learn that they can avoid the human enemy by taking the right hallway, which is protected from the human's fire by a wall. rtNEAT allows the robots to adapt in this way to the player's tactics in real time.

time. Especially in persistent video games such as Massive Multiplayer Online Games (MMOGs) that last for months or years, the potential for rtNEAT to continually adapt and optimize agent behavior may permanently alter the gaming experience for millions of players around the world.

7 Conclusion

A real-time version of NEAT (rtNEAT) was developed to allow users to interact with evolving agents. In rtNEAT, an entire population is simultaneously and asynchronously evaluated as it evolves. Using this method, it was possible to build an entirely new kind of video game, NERO, where the characters adapt in real time in response to the player's actions. In NERO, the player takes the role of a trainer and constructs training scenarios for a team of simulated robots. The rtNEAT technique can form the basis for other similar interactive learning applications in the future, and eventually even make it possible to use gaming as a method for training people in sophisticated tasks.

Acknowledgments

Special thanks are due to Aaron Thibault and Alex Cavalli of the IC^2 and DMC for supporting the NERO project. The entire NERO team deserves recognition for their contribution to this project including the NERO leads Aliza Gold (Producer), Philip Flesher (Lead Programmer), Jonathan Perry (Lead Artist), Matt Patterson (Lead Designer), and Brad Walls (Lead Programmer). We are grateful also to the original volunteer programming team Ryan Cornelius and Michael Chrien (Lead Programmer as of Fall 2004), and newer programmers Ben Fitch, Justin Larrabee, Trung Ngo, and Dustin Stewart-Silverman, and to our volunteer artists Bobby Bird, Brian Frank, Corey Hollins, Blake Lowry, Ian Minshill, and Mike Ward. This research was supported in part by the Digital Media Collaboratory (DMC) Laboratory at the IC^2 Institute (`http://dmc.ic2.org/`), in part by the National Science Foundation under grant IIS-0083776, and in part by the Texas Higher Education Coordinating Board under grant ARP-003658-476-2001. NERO physics is controlled by the Torque Engine, which is licensed from GarageGames (`http://www.garagegames.com/`). NERO's official public website is `http://dmc.ic2.org/nero_public`, and the project's official email address is `nero@cs.utexas.edu`.

Bibliography

Beyer, H.-G., and Paul Schwefel, H. (2002). Evolution strategies – A comprehensive introduction. *Natural Computing*, 1(1):3–52.

Fogel, D. B., Hays, T. J., and Johnson, D. R. (2004). A platform for evolving characters in competitive games. In *Proceedings of 2004 Congress on Evolutionary Computation*, 1420–1426. Piscataway, NJ: IEEE Press.

Goldberg, D. E., and Richardson, J. (1987). Genetic algorithms with sharing for multimodal function optimization. In Grefenstette, J. J., editor, *Proceedings of the Second International Conference on Genetic Algorithms*, 148–154. San Francisco: Kaufmann.

Gomez, F., and Miikkulainen, R. (1999). Solving non-Markovian control tasks with neuroevolution. In *Proceedings of the 16th International Joint Conference on Artificial Intelligence*. San Francisco: Kaufmann.

Gomez, F. J., and Miikkulainen, R. (2003). Active guidance for a finless rocket through neuroevolution. In *Proc. of the Genetic and Evolutionary Computation Conf. (GECCO-2003)*. Berlin: Springer Verlag.

Gruau, F., Whitley, D., and Pyeatt, L. (1996). A comparison between cellular encoding and direct encoding for genetic neural networks. In Koza, J. R., Goldberg, D. E., Fogel, D. B., and Riolo, R. L., editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, 81–89. Cambridge, MA: MIT Press.

Kaelbling, L. P., Littman, M., and Moore, A. W. (1996). Reinforcement learning: A survey. *Journal of Artificial Intelligence*, 4:237–285.

Laird, J. E., and van Lent, M. (2000). Human-level AI's killer application: Interactive computer games. In *Proc. of the 17th Nat. Conf. on Artificial Intelligence and the 12th Annual Conference on Innovative Applications of Artificial Intelligence*. Menlo Park, CA: AAAI Press.

Radcliffe, N. J. (1993). Genetic set recombination and its application to neural network topology optimization. *Neural computing and applications*, 1(1):67–90.

Rumelhart, D. E., McClelland, J. L., and the PDP Research Group (1986). *Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Vol. 1: Foundations*. Cambridge, MA: MIT Press.

Santamaria, J. C., Sutton, R. S., and Ram, A. (1998). Experiments with reinforcement learning in problems with continuous state and action spaces. *Adaptive Behavior*, 6(2):163–218.

Stanley, K. O., and Miikkulainen, R. (2002a). Efficient reinforcement learning through evolving neural network topologies. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2002)*. San Francisco: Kaufmann.

Stanley, K. O., and Miikkulainen, R. (2002b). Evolving neural networks through augmenting topologies. *Evolutionary Computation*, 10(2).

Stanley, K. O., and Miikkulainen, R. (2004). Competitive coevolution through evolutionary complexification. *Journal of Artificial Intelligence Research*, 21:63–100.

Sutton, R. S., and Barto, A. G. (1998). *Reinforcement Learning: An Introduction*. Cambridge, MA: MIT Press.

Thurrott, P. (2002). Top stories of 2001, #9: Expanding video-game market brings Microsoft home for the holidays. *Windows & .NET Magazine Network*.

Utgoff, P. E. (1989). Incremental induction of decision trees. *Machine Learning*, 4(2):161–186.

Watkins, C. J. C. H., and Dayan, P. (1992). Q-learning. *Machine Learning*, 8(3):279–292.

Yao, X. (1999). Evolving artificial neural networks. *Proceedings of the IEEE*, 87(9):1423–1447.

Coevolution in Hierarchical AI for Strategy Games

Daniel Livingstone
School of Computing
University of Paisley
High St., Paisley,
Scotland
daniel.livingstone@paisley.ac.uk

Abstract. Real-Time Strategy games present an interesting problem domain for Artificial Intelligence research. We review current approaches to developing AI systems for such games, noting the frequent decomposition into hierarchies similar to those found in real-world armies. We also note the rarity of any form of learning in this domain – and find limitations in the work that does use learning. Such work tends to enable learning at only one level of the AI hierarchy. We argue, using examples from real-world wars and from research on coevolution in evolutionary computation, that learning in AI hierarchies should occur concurrently at the different strategic and tactical levels present. We then present a framework for conducting research on coevolving the AI

1 Introduction

In recent years advances in computing power have enabled the development of simulations and models in which (very) large numbers of individual agents are explicitly modelled – allowing individual based models to be used alongside more traditional mathematical modelling techniques in a variety of scientific disciplines (see, for example, the range of papers presented in Adami et al. 1998). Other uses have been found in the entertainment industries – for computer generated mobs, masses and armies in films (recently used to great effect in the Lord of the Rings film trilogy), and to create interactive mobs, masses and armies in computer and video games (such as the hugely successful Total War series of games).

However, where traditional approaches to agent based modelling usually require that agent activity, communication and coordination is done within an autonomous framework, for example (Rebollo et al. 2003), commercial games generally have no such requirements. The goal of such Artificial Intelligence (AI) is to entertain, and it is of no concern if principles of autonomy or embodiment are broken, as noted by (Buro 2004).

However, ignoring the demand for the AI to be entertaining, there are some interesting research problems in developing AI for Real-Time Strategy (RTS) games – including planning in an uncertain world with incomplete information; learning; opponent modelling and spatial and temporal reasoning (Buro 2004).

In a similar vein, (Corruble et al. 2002) discuss the different AI problems that exist in computer-based war-games, and generally consider the difficulty inherent in designing AI systems for such games. Particular problems, they argue, are caused by the large search spaces – environments consisting of many thousands of possible positions for each of hundreds, possibly thousands, of units – and the parallel nature of the problem – unlike traditional games, any number of moves may be made simultaneously.

In this paper we review some of the current approaches to building AI systems for RTS games, and propose a framework for developing learning-AI for RTS games in which adaptation occurs at different tactical and strategic levels, which coevolve over time.

2 Hierarchical AI in RTS Games

Hierarchical approaches to developing AI systems are not new. For example, Brooks' famous subsumption architecture uses a hierarchy of behaviours within a single agent (Brooks 1991). The application of hierarchies for military organisation is considerably older. Throughout human history armies have long been organised along very strictly defined hierarchical principles – the ancient Roman armies providing but one good example.

Hierarchies are also very natural control structures for use in RTS games, allowing the development of separate AI systems for controlling high level strategic behaviours of the army as a whole and for low level tactical behaviours which can be given to individual units (Kent 2004). Hierarchal AI can also be used more generally in other games in which very large organisations are modelled (Wallace 2004), and in games which might only model small groups of units but in which there is some form of group command (van der Sterren 2002b).

In an RTS AI hierarchy, the uppermost level will be some form of command agent with overall strategic control. The lowest level will be a – potentially very simple – agent AI responsible for the control of an individual soldier (tank, horse, etc.), and there may be any number of intermediary layers of AI representing command over groups of individual soldiers (such as squads) and progressively larger groupings of combat units.

Within this problem domain we will not currently concern ourselves with problems such as path planning or terrain evaluation, which remain problems of interest. Instead we will concentrate on the problems raised by, and on the benefits of, attempting to build adaptive learning AI systems which may exist at multiple levels of the command hierarchy, and which coevolve over time.

3 Current Hierarchical AI in Video Games

A number of hierarchical AI architectures for use in video games have already been proposed and documented.

For control of small squads, (van der Sterren 2002b) considers a de-centralized approach whereby squad agents communicate and form emergent strategies, such as for fire-and-move behaviours, illustrating how team tactics can emerge from interactions between a number of agents. The advantages and disadvantages of the decentralized approach are illustrated using an example of having the squad wait in ambush. The key limitation appears to be that there is no single AI responsible for the strategic deployment of the squad: a solution to this is presented in the following paper, (van der Sterren 2002a), which modifies the architecture from a self-organising one into a simple hierarchical one. This presents two distinct command styles: Authoritarian, where the member agents always perform the actions they are commanded to do; and Coach, where a higher level squad AI (not an explicit agent commanding the squad, but some strategic controller AI) sends requests to the agents, which then evaluate them in light of their current circumstances before deciding which action to perform. It is suggested that a third style, which combines elements of the first two might be the most successful.

This idea of using command hierarchies for controlling units of agents is also used by (Reynolds 2002), again for the control of small squads of agents. For controlling larger numbers of agents, (Kent 2004) provides an architecture for a non-adaptive hierarchical strategic/tactical AI to control armies in a RTS game. The principal behind the architecture is to mirror the hierarchical structure of real armies in the simulated one.

At the top level is a human or AI game player, which makes the highest level decisions; below this are AIs for individual armies, divisions, brigades, companies, platoons and squads as required by the sizes of the forces involved and the complexity of the game. Finally, at the bottom level of the hierarchy are AIs to control individual soldiers.

(Ramsey 2004) presents a similar approach, less specifically focussed on military organisation, which he calls the 'Multi-Tiered AI Framework', MTAIF. This contains four levels of intelligence: strategic, operational, tactical and individual unit. Within this more generalised framework, the AI approach is similar to that advocated by Kent, but the tasks to be carried out by the different units can vary more, including resource gathering and city and empire building. This paper, like that of Kent (2004), devotes much discussion to solving the problems of message passing and coordination with large hierarchies.

One notable aspect of these AI architectures is the absence of any form of learning. One of the rare occurrences of learning in wargames is in an online report, (Sweetser et al. 2003) which presents work on developing command agents for strategy simulations using Cognitive Work Analysis and Machine Learning approaches – but here the learning is only to aid the decisions of the overall commander, not for the subordinate units.

A hierarchical learning example is presented by (Madeira et al. 2004), who focus on the problem of partially-automating the design of AI for strategy games, proposing a system in which the AI learns how to play the game during development, rather than having the AI manually programmed. While their work uses reinforcement learning in a strategy game environment, they note that in principal a wider range of machine learning techniques may be used.

It is noted that strategy games currently primarily use rule-based systems. Advantages and disadvantages of such an approach are discussed, before reinforcement learning and machine learning are introduced as alternatives.

Again, the problem is decomposed using the natural hierarchical decomposition of the command-structure, allowing decision making to be carried out on several different levels. They repeat the observation that this allows higher levels to focus on more abstract strategic reasoning, leaving fine-grained tactical reasoning to units involved in combat.

They train a reinforcement learning AI player against a rule-based AI player, allowing learning to take place over thousands of games.

Madeira et al. note that enabling learning at different levels simultaneously can be problematic, with a reference to the difficulty of concurrent layered learning (Whiteson and Stone 2003), and instead train a single level of their AI hierarchy at a time. Accordingly, they initialise their learning AI with Rule-Based System controllers for each level of the AI hierarchy and train the top level only. Once training of the top-level is complete, its AI is fixed and training of the AI for the next level down is started – and so on down to the soldier level. While this work is still in progress, with learning yet to be extended to levels below the top level player, we can note that by limiting learning to a single layer at a time, (Madeira et al. 2004) limit the search space available to their learning AI – this, and the potential benefits of enabling simultaneous adaptation at multiple levels is what we consider next.

4 Adaptation at Multiple Levels in Battle Strategies

It is actually quite easy to find real-world examples that highlight the limitations of the approach proposed by (Madeira et al. 2004). Whenever new technology is introduced into the battlefield, or new formations and methods of organising units at a low level are developed, high level strategies must adapt to make good use of the new opportunities. New formations may be ineffective without a new strategy for using them, and new strategies might simply not be available without the new formations or technologies they depend on – as the following two examples demonstrate.

World War I featured the first large scale use of machine guns – yet tactics for infantry assault remained unchanged. These typically entailed sending large lines of men marching steadily towards enemy positions – an ideal target for opposing machine gunners. It was only towards the end of the war that General Oskar von Hutier of

Germany introduced new tactics for his attacking units – and a strategy ('Infiltration') for using them: having loose formations of lightly encumbered soldiers (Storm Troopers) rapidly advance over and beyond the enemy lines, bypassing then attacking strong-points from the rear. This finally allowed infantry to effectively attack enemy lines despite the presence of the deadly machine guns, enabling the Central Powers to briefly make significant advances against the Allies (Makepeace and Padfield 2004).

In the middle ages, spearmen were often used defensively against cavalry. The schiltrom was a particularly effective defensive formation – presenting bristling spear points in every direction, a deadly barrier for oncoming cavalry. At Bannockburn, Robert the Bruce trained his spearmen to charge as a phalanx, and to change formation between schiltrom and phalanx at command (Bingham 1998, p. 222), and the strategic use of these new tactics aided his decisive victory at that battle.

This clearly implies that we should make use of a coevolutionary approach. However, the advantages of using a coevolutionary method are not necessarily clear cut – and there are some potential disadvantages that we need to be aware of.

5 Problems with Coevolution

It has been known for some time that coevolution can lead to the development of sub-optimal solutions (van Valen 1973, Cliff and Miller 1995), and that these conflict with the advantages of coevolutionary processes. (Watson and Pollack 2001) provide several concrete examples of coevolution leading to suboptimal solutions. One particular cause of this is that in typical coevolution scenarios, where the coevolving populations and individuals are competing for resources, the fitness of agents is *relative*. Rather than having an external objective measure of fitness (as typically used in evolutionary computation), fitness is found by determining which of a number of competitors is best.

One consequence of this is that the best solution may drift into worse solutions, as long as it remains the best of the solutions available – and if a chance mutation does improve the best strategy, it may not be selected for as it will not actually improve its fitness. One solution to this problem is to keep examples of previous best solutions in the population (Cliff and Miller 1995).

Another problem investigated by (Watson and Pollack 2001) is that caused by *Intransitive Superiority*. Intransitive Superiority exists in any situation where for three (or more) strategies or players no single strategy or player is the best; instead, and rather like scissors-paper-stone, some form of circular superiority relationship exists between them. E.g. Strategy A beats B and B beats C but C beats A. In such a situation it is may be troublesome even to find the fittest individuals in populations of coevolving strategies.

Of particular note, (Watson and Pollack 2001) show that coevolution with intransitive superiority can not only allow best solutions to drift to poorer ones, but can in some cases *drive* the evolution towards poorer general performance. This is of particular concern to the domain of RTS games, where intransitive superiority is often deliberately designed into the game (Rollings and Morris 1999).

A variety of algorithms and techniques are able to overcome problems caused by intransitive superiority, and the possibility of cycling that can exist even where superiority is transitive (de Jong 2004b). Solutions generally involve the addition of some form of memory or archive to the underlying evolutionary computational technique used.

We are particularly interested in the application of Artificial Immune Systems (AIS) (Dasgupta 2000), a biologically inspired approach in which memory of previous good solutions is a core feature. One recent work has demonstrated the successful application of AIS to the problem of unit selection in RTS, where an intransitive superiority relationship exists between the available unit types (Fyfe 2004).

6 Coevolution in AI Hierarchies

It should be noted that all of the problems described assume that the coevolution occurs between *competing* players or populations. In our case we are actually interested in a quite different situation – where the coevolving strategies are not opponents, but partners operating at different levels. This is somewhat similar to the problem of concurrent layered learning in individual agents, explored by (Whiteson and Stone 2003). While the difficulties of such an approach were cited by (Madeira et al. 2004) as a reason not to coevolve the different levels in an AI hierarchy (Section 3), the point made in (Whiteson and Stone 2003) is that concurrent layered learning is beneficial in certain situations and can outperform traditional, one layer at a time, layered learning (as we argue it may be in Section 4).

In our case we can potentially avoid some of the problems of coevolving opponent AI by pitting our evolving hierarchical AI against fixed, non-evolving opponents. In doing so we may lose some of the advantages of co-evolving against another hierarchical AI, such as having an opponent that presents a 'hittable' target, or allowing for more open-ended evolution instead of having evolution merely drift once the fixed AI has been beaten (Watson and Pollack 2001).

But issues remain. Let A be the set of possible high level strategies and B the set of low level tactics. Further, let *a1* and *a2* be two high level strategies and *b1* and *b2* be two low level strategies. We can easily imagine a situation where, against a fixed opponent, the fitness, *f*, of these strategies is such that $f(a2,b2) > f(a1,b1)$. Now what if, as suggested in Section 4, the fitness advantages of *a2* and *b2* only exists when these strategies are used together. In other terms, there exists a linkage between the variables (de Jong 2004a). Then we may also have $f(a1,b1) > f(a1,b2)$ and $f(a1,b1) > f(a2,b1)$. The fitness landscape presented by coevolving hierarchical AI is likely to be high-dimensional and feature many sub-optimal maxima

and may feature hard-to-find global maxima, presenting a significant challenge.

(de Jong 2004a) note that if the order of linkage – the maximum number of linked variables – is small and some exploitable structure exists, a Hierarchical Genetic Algorithm is most likely able to find the globally optimal solution, given that such a solution exists.

7 Proposal for Coevolutionary Hierarchical AI for RTS Games

We are currently developing a simple RTS environment for testing the coevolution of strategies within a hierarchical AI. Given the concerns and issues mentioned within this review, we plan to proceed according to the following experimental framework.

Our initial experiments will use only two levels of learning. For the first experiments the two coevolving strategies will control, at the lower level, the individual soldier units and, at the higher level, formations and flocks of soldiers. The formation controller will be able to select formations and set formation parameters – such as preferred spacing. Alternative two-level coevolution could focus on high level strategies, such as unit selection and deployment, and formation control – with no evolution at the individual soldier level. We intend to use AIS for both levels in the initial experiments – particularly to exploit the benefit of memory. The experience here will be useful in guiding later work.

In evolving the hierarchical AI, we will have it compete against a small set of fixed AI opponents of differing difficulty. In this way we hope to always present a hittable target for improvement. A population of hierarchical AI will be so tested, allowing relative evolution of the hierarchical AIs against one another alongside the objective measure of their performance against the fixed AI opponents. This will hopefully also enable some open ended evolution beyond the capabilities of the fixed AI as from the population a hall of fame can also be kept – with performance against both the fixed and evolving AI being the criteria for membership.

Evolving and testing the AI players will involve playing thousands of games between fixed and evolving AI players, and accordingly the RTS testbed will allow games to be run faster than real-time, without graphical updates during game play. Further, as well as developing our own RTS test bed for our experimental work, we are currently evaluating third-party open-source RTS environments, such as ORTS (Buro 2004).

8 Conclusions

RTS games provide a rich environment for artificial evolution and other AI approaches to problem solving. Good players, human or AI, need to reason at multiple levels of detail – from overall grand strategies down to highly localised decisions. RTS games also possess very natural and intuitive levels of detail, allowing the problem to be quite neatly decomposed into a number of smaller problems – although these are not independent of one another.

While current approaches to building AI systems for RTS games already use a hierarchical decomposition, these generally do not include learning or adaptation. Those systems that exist that do enable learning, and those that have been proposed, enable learning in single layers only in order to simplify the search involved. One proposal to extend this is to enable learning at successively lower levels after each higher level of learning is complete, in turn.

We note that such a solution may fail to exhaustively search the space of possible solutions, as some good solutions may depend on combinations of adaptations at multiple levels. Accordingly, we propose to develop a system in which the different levels of the hierarchical AI coevolve. We have noted a number of issues regarding this and further proposed a basic framework for future progress on this.

Acknowledgements

Daniel Livingstone would like to thank the Carnegie Trust for the Universities of Scotland for supporting this work. He would also like to thank Donald McDonald and Colin Fyfe for their useful comments and suggestions.

Bibliography

Adami, C., R. K. Belew, H. Kitano and C. E. Taylor (1998). Artificial Life VI. Artificial Life VI, UCLA, MIT Press.

Bingham, C. (1998). Robert The Bruce. London, Constable and Company.

Brooks, R. A. (1991). "Intelligence Without Representation." Artificial Intelligence Journal **47**: 139–159.

Buro, M. (2004). Call for AI Research in RTS Games. AAAI workshop on Challenges in Game AI, D. Fu and J. Orkin (eds.), July 25th-26th 2004, San Jose, 139-141.

Cliff, D. and G. F. Miller (1995). Tracking the Red Queen: Measurements of adaptive progress in co-evolutionary simulations. Third European conference on Artificial Life, Springer-Verlag, pp 200-218.

Corruble, V., C. Madeira and G. Ramalho (2002). Steps Towards Building a Good AI for Complex Wargame-Type Simulation Games. Game-On 2002, The Third International Conference on Intelligent Games and Simulation, Q. Mehdi, N. Gough and M. Cavazza (eds.), London, 155-159.

Dasgupta, D. (2000). Artificial Immune Systems and their Applications, Springer-Verlag.

de Jong, E. D. (2004a). Hierarchical Genetic Algorithms. 8th International Conference on Parallel Problem Solving from Nature, Sept. 18th-22nd, Birmingham, UK, 232-241.

de Jong, E. D. (2004b). Intransitivity in coevolution. 8th International Conference on Parallel Problem

Solving from Nature, Sept. 18th-22nd, Birmingham, UK, 843-851.
Fyfe, C. (2004). "Dynamic Strategy Creation and Selection using Artificial Immune Systems." International Journal of Intelligent Games & Simulation **3**(1): 1-6.
Kent, T. (2004). Multi-tiered AI layers and terrain analysis for RTS games. AI Game Programming Wisdom 2. S. Rabin. Hingham, MA, Charles River Media, Inc.
Madeira, C., V. Corruble, G. Ramalho and B. Ratitch (2004). Bootstrapping the Learning Process for the Semi-automated Design of a Challenging Game AI. AAAI workshop on Challenges in Game AI, D. Fu and J. Orkin (eds.), July 25th-26th 2004, San Jose, 72-76.
Makepeace, D. and A. Padfield (2004). "Trench Hell and the role of the 'Stosstruppen'." wargamesjournal.com: http://www.wargamesjournal.com/prewwii/trench_hell.asp.
Ramsey, M. (2004). Designing a Multi-Tiered AI Framework. AI Game Programming Wisdom 2. S. Rabin. Hingham, MA, Charles River Media, Inc.
Rebollo, M., V. Botti and E. Onaindia (2003). Formal Modeling of Dynamic Environments for Real-Time Agents. CEEMAS 2003, V. Marík, J. Müller and M. Pchouek (eds.), June 16-18, Prague, Springer (Lecture Notes in AI, 2691), 474-484.
Reynolds, J. (2002). Tactical Team AI Using a Command Hierarchy. AI Game Programming Wisdom. S. Rabin. Hingham, MA, Charles River Media, Inc.
Rollings, A. and D. Morris (1999). Game Architecture and Design. Scottsdale, AZ., Coriolis.
Sweetser, P., M. Watson and J. Wiles (2003). Intelligent Command Agents for Command and Control Simulations using Cognitive Work Analysis and Machine Learning, (http://www.itee.uq.edu.au/~penny/). The University of Queensland, Brisbane, Australia.
van der Sterren, W. (2002a). Squad Tactics: Planned Maneuvers. AI Game Programming Wisdom. S. Rabin. Hingham, MA, Charles River Media, Inc.
van der Sterren, W. (2002b). Squad Tactics: Team AI and Emergent Maneuvers. AI Game Programming Wisdom. S. Rabin. Hingham, MA, Charles River Media, Inc.
van Valen, L. (1973) A New Evolutionary Law, Evolutionary Theory 1, p. 1-30.
Wallace, N. (2004). Hierarchical Planning in Dynamic Worlds. AI Game Programming Wisdom 2. S. Rabin. Hingham, MA, Charles River Media, Inc.
Watson, R. A. and J. B. Pollack (2001). Coevolutionary Dynamics in a Minimal Substrate. 2001 Genetic and Evolutionary Computation Conference (GECCO 2001), L. Spector and et al. (eds.), Morgan Kaufmann.
Whiteson, S. and P. Stone (2003). Concurrent Layered Learning. Second International Joint Conference on Autonomous Agents and Multi-Agent Systems, Melbourne, ACM Press, 193-200.

Coevolving Probabilistic Game Playing Agents Using Particle Swarm Optimization Algorithms

Evangelos Papacostantis
Department of Computer Science
University of Pretoria
epapa@cs.up.ac.za

Andries P. Engelbrecht
Department of Computer Science
University of Pretoria
engel@cs.up.ac.za

Nelis Franken
Department of Computer Science
University of Pretoria
nfranken@cs.up.ac.za

Abstract- Coevolutionary techniques in combination with particle swarm optimization algorithms and neural networks have shown to be very successful in finding strong game playing agents for a number of deterministic games. This paper investigates the applicability of a PSO coevolutionary approach to probabilistic games. For the purposes of this paper, a probabilistic variation of the tic-tac-toe game is used. Initially, the technique is applied to a simple deterministic game (tic-tac-toe), proving its effectiveness with such games. The technique is then applied to a probabilistic 4x4x4 tic-tac-toe game, illustrating scalability to more complex, probabilistic games. The performance of the probabilistic game agent is compared against agents that move randomly. To determine how these game agents compete against strong non-random game playing agents, coevolved solutions are also compared against agents that utilize a strong hand-crafted static evaluation function. Particle swarm optimization parameters/topologies and neural network architectures are experimentally optimized for the probabilistic tic-tac-toe game.

1 Introduction

The work on computer games has been one of the most successful and visible results of artificial intelligence research [17]. This is due to the fact that games provide challenging "puzzles" that require a great level of sophistication in order to be played/solved. Games can be considered as problems, which operate by following strict rules within a game environment. The manner in which moves may be executed and games may be won or drawn are strictly defined and may not be deviated from. Game environments are an ideal domain to investigate the effectiveness of an array of different AI techniques. Pioneers of AI research used games as problem domains in their research, for example, Arthur Samuel [16], Claude Shannon [18] and Alan Turning [20].

This paper investigates the effectiveness of using a coevolutionary technique in combination with particle swarm optimization algorithms and neural networks to find game playing agents for probabilistic games from pure self-play. This means that no game information or game strategies are provided to the learning algorithm and the agent learns its own playing strategy by competing against other players. A similar approach has been used by Messerschmidt and Engelbrecht [13] and Franken *et al* [2, 3, 4, 5] for the deterministic games of tic-tac-toe [13, 5, 2], checkers [3, 2] and the iterated prisoners dilemma [4, 2]. The PSO coevolutionary approach used is an adaptation of the evolutionary algorithm approach developed by Chellapilla and Fogel [1].

Probabilistic games contain hidden game information and players compete against each other based on elements of chance, which are beyond their control. Games that are played with dice or cards are usually probabilistic games, which include backgammon, poker, bridge and scrabble. Deterministic games on the other hand are games that provide perfect information to all players at all times. There are no hidden elements and the players can execute moves in any manner they wish to, within the rules of the game, without any probabilistic elements affecting their game decisions. Such games include tic-tac-toe, checkers, chess and go.

Probabilistic games can not be solved, meaning that it is not possible to play a probabilistic game in such a way that you are always guaranteed to either win or draw. This is due to the probabilistic element that may favor any player during the course of the game. This is the reason why games of this nature are similar to real world problems. With real world problems it is very difficult for one to define constraints and such problems almost never contain perfect information. Successful techniques applied to probabilistic games may therefore be more scalable to real world problems in comparison to techniques that are successful when applied to deterministic games.

Game trees have contributed a tremendous amount to game learning, providing the ability to see favorable or non-favorable moves in the future of a game. A number of game tree construction methods exist, with the most popular being minmax [6], alpha-beta [12] and NegaScout [15]. To elaborate a bit more, game trees are constructed by adding every possible move that can be played by each player, alternating the player each time the game tree depth increases. For probabilistic games the construction of a tree becomes impractical, since the tree has to represent all possible outcomes of the probabilistic elements used by the game. This usually causes the tree to become extremely large, hampered by time consuming efforts for its construction and use. Game trees can be constructed based on the probabilities of certain moves that can be executed. Moves that have higher probabilities in being selected are used to construct the tree, while less probable options are excluded. Game trees are not extensively used in this report, with only simple minmax trees, expanded to a depth of 1.

Coevolutionary techniques in combination with other learning algorithms have successfully been applied to probabilistic games, specifically in backgammon[19, 14]. Tem-

poral difference learning is one of these, which is based on Samuel's machine learning research [16]. This learning algorithm has been successfully applied to backgammon by Gerry Tesauro [19]. TD-Gammon, the name given by Tesauro to the program, is a neural network that takes a board state as an input and returns the score of the board state as an output. The weights of the neural network were optimized using temporal difference. TD-Gammon is regarded to be as good as the best human backgammon players in the world, and possibly even better [17]. Another successful learning technique in combination with coevolution has been applied by Blair and Pollack, which managed to use a simple hill-climbing learning algorithm to coevolve competitive backgammon agents [14]. All the examples given above illustrate that coevolution is a very powerful learning machine, which when combined with even the simplest learning algorithms, it may return sound results.

Simulation-based techniques have also shown to be very effective [7, 8]. These techniques allow the simulation of games from their current state to completion, by using different randomly selected possible outcomes to replace the probabilistic elements. By doing so, statistical information about the game is gathered for each possible move. Based on the probability of a win, draw or loss for each move, a decision is made on which move to select. There have been some notable applications of this technique: The University of Alberta developed a poker-playing program called Loki that utilizes simulation-based techniques [7]. Loki is the first serious academic effort to build a strong poker playing program and at best is rated as a strong intermediate-level poker player [17]. Mathew Ginsberg's bridge playing program GIB [8] also makes use of simulation-based techniques. GIB forced the frequent bridge world champion Zia Mahmood to withdraw in 1999 a prize award of £1000000 for any program that manages to beat him, after narrowly beating GIB in an exhibition match [17]. GIB also utilizes another technique frequently applied to probabilistic games, the Monte Carlo simulation method, where a representative sample of all possible moves is chosen to give a statistical profile of an outcome.

2 Particle Swarm Optimization

The particle swarm optimization algorithm is a population-based algorithm that enables a number of individual solutions, called particles, to move through a hyper dimensional search space in a methodical way. The movement of the particles is done in such a way that it enables them to incrementally find better solutions. The algorithm is based on the simulation of the social behavior of birds within a flock and was first described by Kennedy and Eberhart [10]. What mainly drives a PSO algorithm, is the social interaction between its particles. Particles within a swarm share their knowledge with each other, specifically with regards to the quality of the solutions they have found at specific points in the search space. The best solution discovered by a specific particle is referred to as a personal best solution. Particles then move towards other personal best solutions using certain velocities, in an attempt to discover improved solutions.

It is obvious that the pattern of communication between the particles will ultimately affect the manner by which the particles move within the search space. Different information sharing patterns/structures will enable the search space to be explored in different ways. Topology is a term that refers to a pattern by which particles communicate with each other. The following topologies are most commonly used:

- Global Best: All particles communicate with each other, forming a fully interconnected social network. With this topology all particle movements are affected by their own personal best solution and a global best solution. The global best solution forms the best solution of the entire swarm.
- Local Best: A neighborhood size is defined for this topology, which determines the number of particles with which each particle can communicate and share information with. If a neighborhood size is 3, for example, neighborhoods of 3 particles are formed by selecting the two adjacent neighbors of each particle in variable space. With this topology all particle movements are affected by their own personal best solution and a local best solution. The local best solution forms the best solution within the neighborhood the particle belongs to.
- Von Neuman: This topology is very similar to the local best topology, which allows each particle to form a neighborhood with its immediate top, bottom, left and right particles in variable space [11].

A particle swarm optimization algorithm has a number of parameters which allow it to be fine-tuned for better performance. The swarm size and the topology form two parameters, which have already been discussed. Furthermore, four other parameters determine the behavior of the particle movement. Two acceleration constants determine the degree by which a personal best and neighborhood best solution affects a particle's movement. c_1 forms the personal acceleration constant and c_2 the global acceleration constant. The inertia weight variable ϕ determines how much previous particle velocities influence new particle velocities. Finally, V_{max} is a value that sets an upper limit for velocities in all dimensions, which limits particles from moving too rapidly.

3 Coevolution

Coevolution is a competitive process between a number of species that enables the species to continuously evolve in order to overcome and outperform each other. Consider the example of a lion and a buck, where the two are competing in a survival "game". The lions' survival depends on capturing the buck for food, while the bucks' survival on the other hand depends on outwitting the lion so it never gets caught. The buck can initially run faster than the lion, avoiding its capture. The lion fails in numerous attempts, but in the process strengthens its leg muscles, enabling it to run faster and eventually to capture the buck. The buck then develops

a technique that allows it to continuously dodge the lion, since it can not run any faster and gets caught. In return the lion manages to increase its stamina in the process of trying to keep up with the buck, allowing it to follow the buck until it gets exhausted from its dodging maneuvers and then capturing it. The two continuously discover different ways that will enable them to survive in turn. This pattern is similar to the one seen in the case of arms races, where a number of countries compete against each other to produce more destructive and technologically advanced weapons. Another excellent example of coevolution is described by Holland [9].

This continuous coevolution allows each of the species competing to incrementally become stronger, with a tit-for-tat relationship fueling the process. The example given above is a demonstration of predator-pray coevolution, where there is an inverse fitness between the two species. A win for the one means ultimately a lose for the other, with loosing species improving in order to challenge winning species. A different form of coevolution exists, called symbiotic coevolution. In this case species do not compete against each other, but rather cooperate for the general good of all the other species that are coevolving. A success for one of the species, means the improved survival fitness of all other species too.

For the purposes of this report, a combination of symbiotic and predator-prey coevolution has been chosen. The actual algorithm is described in Section 6. This coevolutionary approach is based on the coevolution of two separate populations of game playing agents that compete against each other. A score scheme is used that enables the awarding of points to game playing agents that are successful in winning and drawing games, while loosing agents are penalized. Agents within a population cooperate in an attempt to improve the overall fitness of the population. A PSO algorithm is applied to each population separately to adapt agents.

The size of each population and the scoring scheme used have an influence on the performance of the coevolutionary process.

4 Tic-Tac-Toe Variation

The variation introduced below in section 4.2, extends the original tic-tac-toe game by adding and modifying rules that make the game more complex and probabilistic.

4.1 Tic-Tac-Toe

The original game is a deterministic 2 player game that is played on a 3x3 grid, which initially contains empty spaces. The player who competes first must place an X piece in one of the 9 spaces of the grid with the second player following by doing the same with an O piece. The players may not place a piece in an already occupied space and they may not skip a turn. The objective of the game is for a player to complete a full row, column or diagonal with his own pieces in sequence, with the win going to the player that manages to do so. Both players compete until a player successfully completes the objective or until no more empty spaces exist. In the last case, this implies a draw between the two players.

Table 1 shows the probabilities of a win, draw and loss between two players when playing tic-tac-toe. These probabilities were calculated using two random playing agents competing against each other. $Player^{1st}$ plays first while $Player^{2nd}$ plays second for a total of 100000 games.

	Games	%
$Player^{1st}$	58277	58.277
$Player^{2nd}$	28968	28.968
$Draw$	12755	12.755

Table 1: Tic-tac-toe: probabilities.

The table clearly shows an advantage for $Player^{1st}$. This advantage is due to two facts. The first being the priority of $Player^{1st}$ to capture the center empty space of the 3x3 grid, giving him a significant advantage over $Player^{2nd}$. This is because the center space forms a part of one row, one column and two diagonals, and by placing a mark there, $Player^{1st}$ already has secured 4 winning options to his favor while denying 4 winning options for $Player^{2nd}$. No other space gives such an advantage. The second advantage is that $Player^{1st}$ will have the opportunity to place more pieces on the board, since the board initially consists of an odd number of empty spaces.

4.2 Probabilistic 4x4x4

The tic-tac-toe variation described here is a probabilistic 2 player game, played on 4 layers consisting of 4x4 grids. Another way to visualize the game board is by seeing it as a 3 dimensional cube, consisting of 64 smaller separate cube spaces which make up the positions in which a player can place a piece. The figure below shows this.

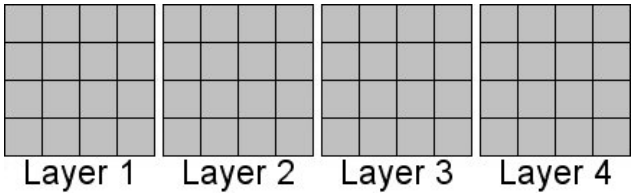


Figure 1: A probabilistic 4x4x4 game board.

The complexity of the game is increased by introducing the new dimension and increasing the board size by 1. The first player, $Player^{1st}$, will no longer have a large advantage over $Player^{2nd}$ as in the original tic-tac-toe game. This is because a center space does not exist any longer due to the even sized board edge. Furthermore, the total number of spaces is an even number too, allowing $Player^{1st}$ and $Player^{2nd}$ to have an equal number of pieces on the board.

The game is played similar to the original tic-tac-toe game, with $Player^{1st}$ and $Player^{2nd}$ alternating turns and respectively placing an X or O piece on one of the board layers. A player does not have the freedom though of placing a piece in any of the 4 available layers. The layer in which a player has to make a move is determined by a "4 sided dice". Just before executing a move, the player

rolls this dice to determine the level to play. If a player has to play on a layer where all spaces are occupied by pieces, he misses that round and the game moves on to the next player. The game only ends when there are no more empty spaces to place a piece. When the board is full, each player counts the number of rows, columns and diagonals he has completed and gets a point awarded for each successful 4 pieces placed in sequence. The player with the most points wins the game. If the players have an equal score, the game is a draw. Figure 2 shows different combinations in which a player can score points. All three dimensions can be used and any 4 pieces lined up in sequence can score a point.

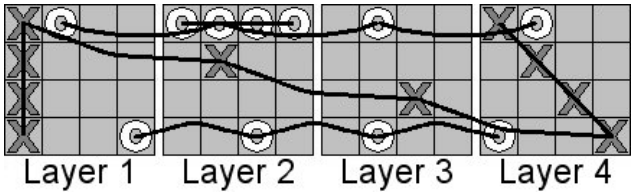


Figure 2:Probabilistic 4x4x4 point combinations.

Table 2 shows the probability of a win for both $Player^{1st}$ and $Player^{2nd}$, and of a draw for a total of 100000 games that are played randomly by both players.

	Games	%
$Player^{1st}$	50776	50.776
$Player^{2nd}$	44367	44.367
$Draw$	4857	4.857

Table 2: Probabilistic 4x4x4: probabilities.

The advantage of $Player^{1st}$ over $Player^{2nd}$ has been considerably reduced compared to the original tic-tac-toe game. Only a 6.4% winning advantage seperates $Player^{1st}$ from $Player^{2nd}$ in the probabilistic variation.

5 The Game Playing Agents

The game playing agents are represented by standard 3-layer feed forward neural networks, consisting of summation units. The size of the input layer is dependant on the size of the game board. The input layer therefore consists of 9 neurons for a standard 3x3 tic-tac-toe game, while 64 neurons are required for a probabilistic 4x4x4 tic-tac-toe game. The size of the hidden layer varies, depending on the complexity of the game. Only one neuron is used in the output layer. The architecture explained above excludes all bias units. The sigmoid activation function is used for each neuron in the hidden and output layers, with the steepness value λ set to 1. The weights of the neural network are randomly initialized between the range $[\frac{-1}{\sqrt{fanin}}], [\frac{1}{\sqrt{fanin}}]$, where $fanin$ represents the number of incoming weights to the neuron.

The neural network is used to evaluate a given state by accepting the actual state as an input and returning as an output a value that represents how advantageous the state is, with states returning higher values preferred.

Assume that $Player^x$ plays against $Player^y$ and that $Player^x$ needs to plan a new move. Let $State^{current}$ denote the current game state. The following steps are used to determine the next state.

1. Build a game tree with a depth of N, using $State^{current}$ as the root node and by adding all possible moves for $Player^x$ for all odd depths and $Player^y$ for all even depths.

2. Evaluate all leaf nodes by using the neural network as an evaluation function in the following manner:

   i. For all valid positions on the board assign a value of 0.5 for every $Player^x$ piece on the board, a value of -0.5 for every $Player^y$ piece and a value of 0 if there is no piece on a specific position.

   ii. Supply these values as inputs to the neural network and perform a feed forward process to determine the output.

   iii. Assign the value of the neural network output as the evaluation value of the node.

3. Using the minmax [6] algorithm, determine the most beneficial state to execute the next move.

Instead of using minmax, the alpha-beta [12] or NegaScout [15] algorithms can be used to optimize the game tree. Only a single depth for the game tree has been considered throughout this paper.

The input representation scheme used in the first point of step 2.i, results in identical board states to be inversely represented depending on whether the player played first or not [3].

Since a neural network is used to evaluate how good a state is, the objective is to find a set of weights which can differentiate between good states and bad states. Usually, supervised training would be used to adjust the neural network weights. With supervised training there exists a training set consisting of both inputs and the associated desired outputs. The most popular supervised training algorithm is back propagation [22]. With back propagation, each pattern of the training set is used and the difference between the actual output and the target output is used to adjust the weights. After repeating this process for a number of times for the full training set, the neural network eventually fits a curve over the training set to relate inputs with desired outputs. In the case of game learning one does not have a training set and the weights can therefore not be adjusted using back propagation or any other supervised training technique. This problem is overcome with the use of coevolution and particle swarm optimization algorithms.

6 The Game Training Process

This section explains exactly how the coevolution and particle swarm optimization algorithms are combined to train game playing agents. Initially, two populations of game playing agents are instantiated by generating a number of neural networks with randomly initialized weights. These

neural networks form possible game playing agents that will coevolve, growing stronger for each new generation. The agents represent particles in a swarm. Each particle represents the weights of one neural network. Each agent has the ability to store two sets of weights: its current weights ($Weights^{current}$) and its best weights ($Weights^{best}$) it has encountered during the training process. Each agent in the first population competes against all other agents in the second population, and vice-versa. A scoring scheme is then used to evaluate the agents. Agents are rewarded/penalized based on whether a game has been lost, won or drawn. Points are accumulated for each agent over all games played by the agent. Higher scoring agents are considered better than lower scoring agents. These scores are used to determine personal, local and global best solutions, as needed for the PSO algorithm. To discover the overall best agent, the two best agents of each population compete against a random player for a total of 10000 games. Based on the score scheme used, the agent with the highest score is regarded as the overall best agent and stored in a hall of fame. In no way does this agent affect training and the sole reason the agent is stored is to preserve the overall best agent during training. The evaluation against a random player may be time consuming, but the diversity of game playing strategies that they offer is valuable, making them appropriate to be used for evaluation purposes. A detailed step-by-step algorithm is given in the following subsection.

6.1 Step-by-Step

1. Instantiate two new populations of agents. Each agent is initialized in the following way:
   - The $Weights^{current}$ are initialized as explained in Section 5.
   - The $Weights^{best}$ are set equal to the $Weights^{current}$ for the first generation.
2. Agents compete against agents in the opposing population, as explained in Section 5. Agents use both their $Weights^{current}$ and $Weights^{best}$ to compete. Competing agents use a preselected score scheme, based upon which each agent receives a specific score. The scheme adopted by this paper awards the following: 3 points for a win, 1 for a draw and 0 for a lose. The weights of each agent are used as follows to compete against all other agents in the other population:
   - All $Weights^{current}$ of the one population compete against all $Weights^{current}$ of the other population by playing both first and second. Based on each $Weights^{current}$ wins/losses/draws, a score is assigned to each $Weights^{current}$.
   - All $Weights^{best}$ of the one population compete against all $Weights^{current}$ of the other population by playing both first and second. Based on each $Weights^{best}$ wins/losses/draws, a score is assigned to each $Weights^{best}$.
3. The scores of all $Weights^{current}$ and $Weights^{best}$ for each agent in the two separate populations are compared. If the score of an agents' $Weights^{current}$ is larger than the score its $Weights^{best}$, then its $Weights^{current}$ becomes the new $Weights^{best}$ and therefore its new personal best.
4. All the scores of the $Weights^{best}$ in each separate population are compared. The agent with the highest score for its $Weights^{best}$, becomes the local/global best of the population.
5. The $Weights^{best}$ of both agents with the highest scores that exist in both populations compete against a random playing agent for a total of 10000 games, 5000 of which the agent plays first and the remaining 5000 the agent playing second. A score for the $Weights^{best}$ for both best agents that belong to the two populations is determined. If a score is found that is the highest score observed thus far during training, the weights are stored as the best weights encountered during training. This set of weights is called $Weights^{supreme}$.
6. Update all $Weights^{current}$ based on the PSO algorithm used for both populations.
7. If the algorithm has not converged at a specific solution, go to step 2 and repeat the process.

7 Results

The following section reports the results of the coevolutionary technique, as applied to the two tic-tac-toe games.

Each simulation was executed 30 times, with the $Weights^{supreme}$ stored for each. The evaluation of two agents is determined by using a sample of 100000 games. Each agent competes for 50000 games by playing first while the remaining 50000 games the agent competes second. The percentage of wins, loses and draws is given in each case, together with F, its Franken performance measure [2].

$Player^{static}$ refers to players utilizing a hand-crafted static evaluation function, $Player^{ran}$ refers to players playing randomly and $Player^{supreme}$ to players that utilize the $Weights^{supreme}$ that were found with the coevolutionary PSO method. In the case of $Player^{supreme}$, the average of all 30 best solutions is shown in the given tables.

The hand-crafted evaluation function used by $Player^{x}$ to compete against $Player^{y}$ for both games is defined as:

$$\sum_{k=1}^{n_{Player^x}} pieces_k - \sum_{k=1}^{n_{Player^y}} pieces_k \qquad (1)$$

where n_{Player^x} is the total number of rows, columns and diagonals (in all dimensions) of the game that only contain pieces belonging to $Player^{x}$, and $pieces^{k}$ is the total number of pieces in that specific row, column or diagonal. In the case where $pieces^{k} = max$, then $pieces^{k} = +\infty$. The value of max represents the maximum number of pieces that can be placed in sequence. The value of 3 is used

for tic-tac-toe and 4 is used for probabilistic 4x4x4 tic-tac-toe. The reason this assignment is done is to allow agents using equation (1) to immediately take opportunities that will allow them to complete full sequences which enables them to win games/score points. Higher values returned by the hand-crafted evaluation function represents better board states.

7.1 Tic-Tac-Toe

7.1.1 Hand-Crafted Evaluation Results

Table 3 shows the results of how the hand-crafted function performs against $Player^{ran}$.

	%	F
$Player^{static}$	93.55	
$Player^{ran}$	1.78	95.88
$Draw$	4.67	

Table 3: Tic-tac-toe: hand-crafted evaluation.

It is clear that the function works very well, since it only looses 1.78% in total. One could argue that this is not satisfactory, since the game is very simple and could be played in such a way that games are only won or drawn. This is where the depth of the minmax tree comes into play. In order for the agents to achieve perfect play for the tic-tac-toe game using the provided hand-crafted static evaluation function, it is required for them to be able to construct deeper trees that will enable them to explore possible future moves. When a depth of 4 (ply-depth of 2) is used, perfect play is achieved. In order to keep the complexity of the learning algorithms as low as possible, depths of only 1 are used, keeping in mind that increased depth sizes may return improved results.

7.1.2 Coevolutionary PSO Results

The initial configurations of the coevolutionary and particle swarm optimization algorithms are taken from [5], specifically the configuration that returned the best results was chosen. The Von Neuman topology was selected, with c_1, c_2 and ϕ all initialized to the value of 1. No V_{max} value was selected, meaning that the velocity of the particles are not restricted in any way. The swarm size for each population was set to 10 (20 particles are used in total), with each particle having 7 neurons in the hidden layer. A score scheme that awarded 3 points for a win, 1 point for a draw and 0 points for a loss was used. Table 4 reveals the results of this parameter configuration.

	%	F	Variance
$Player^{supreme}$	72.07		
$Player^{ran}$	22.83	74.61	$\pm$2.66
$Draw$	5.1		

Table 4: Tic-tac-toe: initial setup.

The coevolutionary PSO algorithm does not manage to produce agents that perform very well, with a mediocre improvement when compared to Table 1. The $Player^{supreme}$ under performs in comparison to $Player^{static}$. Figure 3 indicates the performance of the two best set of weights in each population and the performance of the overall best set of weights ($Weights^{supreme}$) over the generations for one of the executed simulations. The performance measure used in the graph is the score when competing against $Player^{ran}$ for 10000 games. The covered white part of the graph conveys the performance of $Weights^{best1}$, which belongs to the first population, while the black part conveys the performance of $Weights^{best2}$ which belongs to the second population. The two gray lines are sixth degree polynomials fitted through $Weights^{best1}$ and $Weights^{best2}$.

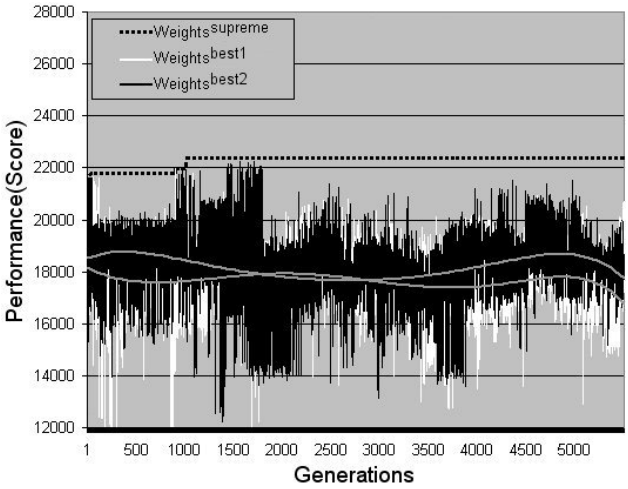


Figure 3: Best agents performance: initial setup.

The graph clearly indicates a premature convergence taking place. The best agents in both populations are clearly struggling to find better solutions and do not improve at all in the later stages during training. The performance of the weights belonging to both agents remain constant throughout training, with no arms race pattern being visible. By investigating the velocity values during training, it was noticed that these grew considerably large in all dimensions. The maximum velocity value V_{max} was therefore set to 1, a very small value, to investigate how this affects training. The results are given in Table 5, clearly showing an improvement.

	%	F	Variance
$Player^{supreme}$	80.34		
$Player^{ran}$	13.76	83.48	$\pm$3.65
$Draw$	5.9		

Table 5: Tic-tac-toe: $Vmax$ = 1.

Figure 4 shows how this change has affected the performance of the best agents in each population. Both agents are now alternating and continously finding newer weights with improved solutions during training, clearly revealing an "arms race" effect, as described in Section 3. Sixth degree polynomials have been fitted through the performance of both sets of weights, making this more apparent.

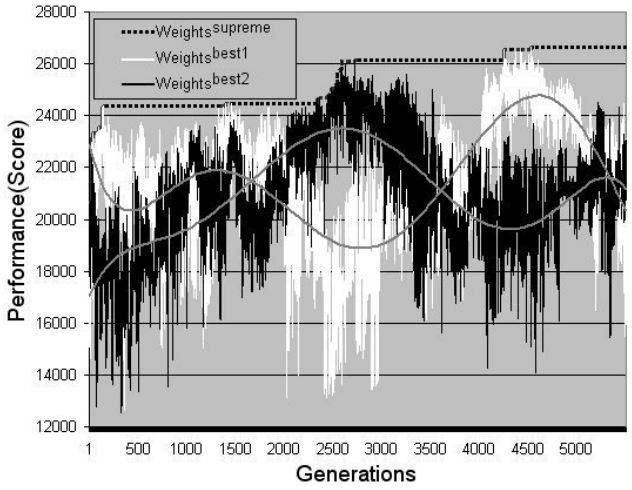


Figure 4: Best agents performance: $Vmax$ = 1.

7.2 Probabilistic 4x4x4 Tic-Tac-Toe

7.2.1 Hand-Crafted Evaluation

The results of the hand-crafted static evaluation function for the probabilistic 4x4x4 tic-tac-toe game are shown in table 6.

	%	F
$Player^{static}$	99.43	99.55
$Player^{ran}$	0.32	
$Draw$	0.25	

Table 6: Probabilistic 4x4x4: hand-crafted function.

The results indicate that the hand-crafted evaluation function for this game is extremely good, but not perfect. This is expected though, since the game is probabilistic, making it impossible for a player to constantly win or draw games, since the probabilistic element can not always favor $Player^{static}$.

7.2.2 Coevolutionary PSO Results

Using the exact same setup that proved successful for tic-tac-toe, the results for probabilistic 4x4x4 tic-tac-toe is shown in Table 7.

	%	F	Variance
$Player^{supreme}$	86.19	87.49	$\pm$2.95
$Player^{ran}$	11.21		
$Draw$	2.6		

Table 7: Probabilistic 4x4x4: Initial setup.

The results look very promising, with 86.19% of the games won. A series of other simulations were done, investigating different topologies, neural network architectures and swarm sizes. Table 8 shows the results when the Global Best and Local Best topologies were used, indicating that the Von Neuman topology performs marginally better (Table 7). The Von Neuman topology had managed to succeeded in finding solutions that win an average of 1.17% more than the solutions found by Global Best and an average of 0.62% more than solutions found by Local Best.

Topology		%	F	Variance
GBest	$Player^{supreme}$	85.02	86.29	$\pm$3.23
	$Player^{ran}$	12.42		
	$Draw$	2.56		
LBest	$Player^{supreme}$	85.57	86.9	$\pm$2.93
	$Player^{ran}$	11.77		
	$Draw$	2.66		

Table 8: Probabilistic 4x4x4: Different topologies.

Hidden		%	F	Variance
10	$Player^{supreme}$	85.22	86.50	$\pm$2.54
	$Player^{ran}$	12.19		
	$Draw$	2.59		
15	$Player^{supreme}$	84.65	86.02	$\pm$3.06
	$Player^{ran}$	12.61		
	$Draw$	2.74		

Table 9: Probabilistic 4x4x4: Different hidden layer sizes.

Size		%	F	Variance
15	$Player^{supreme}$	92.57	93.32	$\pm$2.83
	$Player^{ran}$	5.93		
	$Draw$	1.5		
20	$Player^{supreme}$	95.06	95.615	$\pm$2.48
	$Player^{ran}$	3.85		
	$Draw$	1.09		
25	$Player^{supreme}$	96.55	96.8	$\pm$2.12
	$Player^{ran}$	2.94		
	$Draw$	0.51		

Table 10: Probabilistic 4x4x4: Different swarm sizes.

Table 9 indicates that no performance improvement was gained with an increase in the number of the hidden neurons. Therefore the hidden layer size is not increased and remained as a value of 7. Different population sizes were investigated, with the results shown in Table 10. Population sizes of up to 25 were used, which included sizes of 15, 20 and 25 agents. The results clearly show that there is a direct relation on the improvement of agents as the population sizes increase. Larger swarm sizes offer a larger diversity of solutions, enabling the PSO algorithm to discover better solutions. One must not forget though the increase in complexity of the coevolution process, since a larger population requires more games to be played to evaluate agents. Even though a population size of 25 produced a 1.49% improved winning result over a 20 sized population, the complexity increase does not make this worth while. The population size of 20 would therefore be more favorable.

8 Conclusions and Future Work

The coevolutionary technique in combination with a particle swarm optimization algorithm has shown to be very successful in finding strong agents for the probabilistic 4x4x4 tic-tac-toe game. The most optimal setup presented in this paper was capable in producing a network that could almost match the performance of the hand-crafted evaluation function. Future work includes a more detailed study

with regards to the PSO parameters and the application of the technique to more complex probabilistic games such as backgammon and poker. One important coevolutionary aspect was not approached in this report, which forms the next step in improving the technique. This is concerning the selection of agents within populations which are used for fitness sampling. Different selection strategies should be examined which might prove more efficient than the selection of the entire population which is adopted in this paper. In addition to that, a more in depth investigation must be done to examine how different scoring schemes affect training. Scoring schemes that award equal points for winning and drawing would encourage defensive strategies to be found, while strategies that only award wins with points would encourage aggressive strategies to be found.

Bibliography

[1] Chellapilla K, Fogel D. (1999) Evolving neural networks to play checkers without expert knowledge. IEEE transactions on neural networks, 10(6):1382-1391.

[2] Franken N. (2004) PSO-based coevolutionary game learning. MSc thesis, Department of Computer Science, University of Pretoria, South Africa.

[3] Franken N, Engelbrecht AP. (2003) Comparing PSO structures to learn the game of checkers from zero knowledge. In proceedings of the IEEE congress on evolutionary computation (CEC2003), Canberra, Australia.

[4] Franken N, Engelbrecht AP. (2004) PSO approaches to co-evolve IPD strategies. In proceedings of the IEEE congress on evolutionary computation (CEC2004), Portland, USA.

[5] Franken N, Engelbrecht AP. (2004) Evolving intelligent game playing agents. South African Computer Journal.

[6] Brudno A. (1963) Bounds and valuations for abridging the search for estimates. Problem of cybernetics, 10:225-241. Translation in Problemy Kibernetiki, 10:141-150.

[7] Billings D, Pena L, Schaeffer J, Szafrom D. (1999) Using probabilistic knowledge and simulation to play poker. In AAAI national conference, pages 697-703.

[8] Ginsberg M. (1999) GIB: Steps toward an expert-level bridge-playing program. In international joint conference on artificial intelligence, pages 584-589.

[9] Holland JH. (1990) ECHO: explorations of evolution in a miniature world. (eds) Farmer JD, Doyne J. Proceedings of the second conference on artificial life, Addison-Wesely.

[10] Kennedy J, Eberhart RC. (1995) The particle swarm: social adaptation in information-processing systems. (eds) Corne D, Dorigo M, Glover F. New ideas in optimization, McGraw-Hill, pages 379-387.

[11] Kennedy J, Mendes R. (2002) Population structure and particle swarm performance. In proceedings of congress on evolutionary computation (CEC 2002), Honolulu, Hawaii, USA.

[12] Knuth D, Moore R. (1975) An analysis of alpha-beta pruning. Artificial intelligence, 6(4):293-326.

[13] Messerschmidt L, Engelbrecht AP. (2002) Learning to play games using a pso-based competitive learning approach. In proceedings of the 4th Asia-Pacific conference on simulated evolution and learning, Singapore.

[14] Pollack B, Blair AD. (1998) Co-evolution in the successful learning of backgammon strategy. Machine learning, 32(3),pages 225-240.

[15] Reinfeld A. (1983) An improvement of the scout tree-search algorithm. Journal of the international computer chess association, 6(4):4-14.

[16] Samuel A. (1959,1967) Some studies in machine learning using the game of checkers. IBM journal of research and development.

[17] Schaeffer J. (2001) The games computers (and people) play. Academic Press, Vol.50, pages 189-266. (eds) Zelkowitz MV.

[18] Shannon C. (1950) Programming a computer for playing chess. Philosophical magazine, 41:256-275.

[19] Teasuro G. (1995) Temporal difference learning and TD-Gammon. Communications of the ACM, 38(3):58-68.

[20] Turing A. (1953) Digital computers applied to games. (eds) B Bowden. Faster than thought, pages 286-295, Pitman.

[21] van den Bergh F. (2002) An analysis of particle swarm optimizers. PhD thesis, Department of Computer Science, University of Pretoria, South Africa.

[22] Werbos PJ. (1974) Beyond regression: new tools for prediction and analysis in the behavioral sciences. PhD thesis, Harvard University, Boston, USA.

Evolving a Neural Network Location Evaluator to Play Ms. Pac-Man

Simon M. Lucas
Department of Computer Science
University of Essex
Colchester, UK
sml@essex.ac.uk

Abstract

Ms. Pac-Man is a challenging, classic arcade game with a certain cult status. This paper reports attempts to evolve a Pac-Man player, where the control algorithm uses a neural network to evaluate the possible next moves. The evolved neural network takes a handcrafted feature vector based on a candidate maze location as input, and produces a score for that location as output. Results are reported on two simulated versions of the game: deterministic and non-deterministic. The results show that useful behaviours can be evolved that are frequently capable of clearing the first level, but are still susceptible to making poor decisions. Currently, the best evolved players play at the level of a reasonable human novice.

1 Introduction

Games have long been used in the study of computational intelligence and machine learning. Much of the early focus was on strategy games such as chess and checkers. There have been noteworthy successes for machine learning methods in Backgammon using reinforcement learning [12], or co-evolution [9], and also using co-evolution to find good position evaluation functions for checkers [3].

More recently, with the rise in popularity of real-time arcade and console games, there has been growing interest in applying computational intelligence to such games [7]. This paper investigates the use of Pac-Man style games, in particular Ms. Pac-Man as a challenge for computational intelligence. One of the motivations for this work is to gain insight into the kind of approaches are effective in this domain, where the large state space precludes the direct use of game-tree search. While this paper only evolves the Pac-Man controller, it provides a foundation for also evolving ghost behaviours, which might lead to Pac-Man variants that are even more fun to play than the original. Furthermore, the aim is to eventually gain insight into how hard these games are to play: are good human players simply reacting quickly in ways that would be obvious to some rudimentary form of computational intelligence, or does Ms. Pac-Man have chess-like complexity?

Pac-Man is a classic arcade game originally developed by Toru Iwatani for the Namco Company in 1980. The game rapidly achieved cult status, and various other versions followed. The best known of these is Ms. Pac-Man, released in 1981, which many see as being a significant improvement over the original. In the original game, the ghosts behaved deterministically, and a player could exploit this behaviour by using set patterns or routes. Provided the successful route was executed faithfully (in some routes precise pauses were necessary at certain points on the route), then the level could be cleared while achieving maximum points in the process. From a machine learning perspective, the learning of such routes is less interesting than the learning of more general intelligent behaviour, able to cope with novel circumstances. In Ms. Pac-Man, the ghosts have similar hunting abilities to the original game, but are non-deterministic, which makes the game much harder and more interesting. Ms. Pac-Man also uses four different mazes, which are revealed as the player progresses through levels, though in this paper we only use the first level. The rules and game setup are reasonably simple, at least compared to more modern arcade and console games, though close investigation reveals many interesting subtleties that might go unnoticed by the casual player. Many of these finer points are mentioned in sections 2 and 3 below.

From a computational intelligence perspective, there are many aspects of the game worthy of study. These include evolving Pac-Man playing agents and evolving ghost behaviours. The latter is a problem in multi-agent systems, since the ghosts should cooperate with each other to be maximally effective, though the game designer can impose various constraints on the ghosts, such as disallowing direct ghost communication. Both the Pac-Man and Ms. Pac-Man ghosts behave far from optimally, and this may add to the appeal of the game. However, it would also be interesting to

experience playing against evolved 'optimal' ghosts, even if it turns out to be less enjoyable than playing against the standard sub-optimal variety.

1.1 Previous Research

There have been several previous efforts to evolve Pac-Man players. One of the earliest was that of Koza [6]. Koza used his own implementation of Pac-Man, based on the first screen of Ms. Pac-Man. He defined 15 functions: two conditionals and 13 primitives for controlling the Pac-Man. These included output controls such as "move towards to nearest pill along shortest path". Koza's implementation appears to be significantly different from the real game, and the ghost behaviours make the game quite easy for the Pac-Man. In his version, all four ghosts follow the same behaviour: pursue Pac-Man when he is in sight for twenty out of twenty-five time steps, and randomly change direction for five out of twenty-five time-steps. This is very different from the original Ms. Pac-Man ghosts, who each had distinctively different behaviours, would pursue the Pac-Man more aggressively, and would reverse direction much less often (more like once in two hundred time steps on average), and less predictably. More frequent ghost direction reversals make the game much easier for the Pac-Man, since each reversal can act as a lifesaver in an otherwise hopeless situation.

More recently, Gallagher and Ryan [5] evolved a Pac-Man player based on a finite-state-machine plus rule-set. In their approach they evolved the parameters of the rule set (85 in total), where the individual rules were hand-specified. However, the game simulator they used was a greatly simplified version of the original. Although the maze was a faithful reproduction of the original Pac-Man maze, only one ghost was used (instead of the usual 4), and no power pills were used, which misses one of the main scoring opportunities of the game.

De Bonet and Stouffer [2] describe a reinforcement learning approach to simple mazes that only involved one other ghost, and a 10×10 input window for the control algorithm centred on the Pac-Man's current location. They were able to learn basic pill pursuit and ghost avoidance behaviours. There has also been some work on learning routes for Pac-Man, but as explained above, this approach is not viable for Ms. Pac-Man.

2 Ms. Pac-Man Game Play

The player starts with three lives, and a single extra life is awarded at $10,000$ points. While it is never a good idea to sacrifice a life, it may be better to take more risks when there are lives to spare. There are 220 food pills, each worth 10 points. There are 4 Power Pills, each worth 50 points. The score for eating each ghost in succession immediately after a power pill starts at 200 and doubles each time. So, an optimally consumed power pill is worth 3050 $(= 50 + 200 + 400 + 800 + 1600)$. Note that if a second power pill is consumed while some ghosts remain edible from the first power pill consumption, then the ghost score is reset to 200.

Additionally, various types of fruit appear during the levels, with the value of the fruit increasing with each level. The fruit on level one is worth only 100 points, but this increases to many thousands of points in higher levels. It is not necessary to eat any fruit in order to clear a level.

While good Pac-Man players often make use of fixed routes, the non-determinism of Ms. Pac-Man makes this approach ineffective. Instead, short term planning and reactive skill is more important. An appreciation of ghost behaviour also plays a significant part, since although the ghosts are non-deterministic, they are still at least partially predictable. The reactive skill lies in judging the distance between the Pac-Man and the ghosts, and working out relatively safe routes to the pills and the power-pills. Appropriate consumption of the power pills is of critical importance both for gaining high-scores, and for clearing the levels. Expert players will often get the ghosts to form a closely-knit group, which chases the Pac-Man to the power pill, only to be eaten after consumption of the power pill. Indeed, expert players generally have a good understanding of how the ghosts will behave in any circumstances. Billy Mitchell, the world Pac-Man (but not Ms. Pac-Man) champion said the following:

> "I understand the behavior of the ghosts and am able to manipulate the ghosts into any corner of the board I choose. This allows me to clear the screen with no patterns. This was a more difficult method for the initial 18 screens. I chose to do it this way because I wanted to demonstrate the depths of my abilities. I wanted to raise the bar higher - to a level that no one else could match."

In both Pac-Man and Ms. Pac-Man, the ghost behaviour is far from optimal in several ways, in the sense that the ghosts do not try and eat the Pac-Man as quickly as possible. On leaving the nest, each ghost typically meanders for a while before pursuing the Pac-Man more actively. They then adopt different behaviours. The red ghost is the most predatory one, and is quite direct in seeking out the Pac-Man. The ghosts get less aggressive in order, through pink and blue down to the orange ghost, which behaves in a somewhat random way - and will often let Pac-Man escape from situations where the right moves would secure a certain kill. The game also appears to have a bug, in that sometimes the Pac-Man is able to escape what looks like a certain death situation by passing straight through one of the ghosts. Whether this is a bug (which can occur all

too easily in multi-threaded systems due to synchronisation problems), or a deliberate feature is not clear. Either way, it does not detract from the quality of the game play, and perhaps even enhances it. For the current paper, only the opening maze of Ms. Pac-Man has been implemented, as the evolved players are still at an early stage.

3 Experimental Setup

While the long-term aim of this work is to evolve controllers for the original Ms. Pac-Man game, there are many other worthy objectives as well, such as evolving new ghost behaviours, and experimenting with the effects of various changes to the rules. For these purposes, and also to enable much faster fitness function evaluation, we implemented our own Ms. Pac-Man simulator. The implementation was designed to be a reasonable approximation of the original game, at least at the functional if not cosmetic level, but note that there are several important differences:

- The speed of both the Pac-Man and the ghosts are identical, and the Pac-Man does not slow down to eat pills.
- Our Pac-Man cannot cut corners, and so has no speed advantage over the ghosts when turning a corner.
- The ghost behaviours are different (ours are arguably more aggressive, apart from the random one).
- Our ghosts do not slow down in the tunnels. This comes as a bit of a shock when playing the game!
- In our maze, there is no fruit. The main aim at present is to learn the basic game. Fruit only plays a minor part until the higher levels when it becomes more valuable, and can add significantly to the score.
- No additional life at $10,000$ points. This would have been easy to implement, but has little bearing on evolved strategies.
- All ghosts start immediately at the centre of the Maze, and are immediately in play: there is no nest for them to rest in.
- A ghost once eaten returns instantly to the centre of the maze, and resumes play immediately.
- Currently, our ghosts do not consider the locations of the other ghosts; they could improve their hunting if they utilised such information to 'spread-out' more when hunting the Pac-Man.
- Our game lacks the nice cosmetic finishes of the original. Game play (for a human player) is surprisingly diminished as a result of being chased by a silent rectangle, instead of a wailing ghost with watchful eyes!

Some of these are due to lack of time available in implementing the game, but replicating the exact ghost behaviours is hard. While it is possible that the underlying behaviours may be reducible to a few simple rules, it is difficult to determine these from observation alone, though clearly the expert human players have done a good job of this, at least implicitly. Indeed, learning the ghost behaviours by observing game-play would be a challenging machine learning project in itself.

3.1 The Implementation

The implementation is written in object-oriented style in Java and is reasonably clean and simple. The maze is modelled as a graph, with each node in the graph being connected to its immediate neighbours. Each node has two, three or four neighbours depending on whether it is in a corridor, a T-junction, or a crossroads. Each graph node is separated by two screen pixels - this gives very smooth movement. After the maze has been created, a simple efficient algorithm is run to compute the shortest-path distance between every node in the maze and every other node in the maze. These distances are stored in a look-up-table, and allow fast computation of the various controller-algorithm input features listed below. During evolution, each game lasts for between a few hundred and a few thousand time-steps, depending on luck and on the quality of the controller. Typically, around 33 games per second can be played (on a 2.4ghz PC) when evolving a perceptron. This slows to around 24 games per second for an MLP with 20 hidden units. The cost of simulating the game and computing the feature vector for each node under consideration is greater than the cost of simulating the neural networks.

For a given maze, game play is defined by the ghost behaviours and a few other parameters, listed in Table 1. The ghosts operate by scoring each option and choosing the best one, when more than one option exists, which is only the case at junction nodes (recall that ghosts are not allowed to turn back on themselves, except during a reversal event). Ghost 1 pursues the most aggressive strategy, always taking the shortest path to the Pac-Man. Ghosts 2 and 3 operate under less effective distance measures, while ghost 4 makes random choices, except when playing deterministically, in which case it chooses the option that minimises the y distance to the Pac-Man. All ghosts have small pseudo-random noise added to their evaluations in order to break ties randomly.

The author tested the implementation to assess the difficulty of the game. For a human player, of course, this is influenced very significantly by the speed at which the game is played. When played at approximately the same speed as the original Ms. Pac-Man, the game seemed harder than the original, with the author's best score after ten games be-

Feature	Description
Ghost 1	Minimise shortest-path dist. to Pac-Man
Ghost 2	Minimise Euclidean dist. to Pac-Man
Ghost 3	Minimise Manhattan dist. to Pac-Man
Ghost 4	Random (Or minimise y-dist. if non-rand.)
Edible	Ghosts are edible for 100 time steps
Reversal	Ghosts reverse every 250 time-steps
Ed. Spd	Ghosts move at half-speed when edible

Table 1. *The Simulated Game Setup.*

ing only $12,890$, compared to over $30,000$ on the original Ms. Pac-Man. When slowed down to $1/3$ speed, however, a score of over $25,000$ was achieved before boredom set in. Since playing the original game at $1/3$ speed was not possible, it is hard to make a direct comparison beyond this.

3.2 Location Features

Our Pac-Man control algorithm works by evaluating each possible next location (node), given the current node. Note that each node is two screen pixels away from it's neighbour, which leads to smooth movement, similar to the arcade game, but also makes the maze-graph rather large, with $1,302$ nodes in total.

The control algorithm chooses the move direction that will move it to the best-rated location. From an evolutionary design viewpoint, the most satisfying approach would be to use all the information available in the state of the game as inputs to the node evaluation neural network. This would include the maze layout, the position of each pill and power pill, and the position of each ghost (and whether the ghost was currently edible, and how much longer it was due to remain edible for). This would involve processing a large amount of information, however. For the current experiments, a less satisfactory, but more tractable approach has been taken. Features were chosen on the basis of their being potentially useful, and efficient to calculate. These are listed in Table 2. These features are unlikely to be optimal, and almost certainly place significant limitations on the ultimate performance that such a controller can achieve. Whether the networks evolved for this paper have reached these limits is another question. Note that the use of maze shortest-path distances in the features naturally helps the Pac-Man avoid getting stuck in corridor corners; something that could otherwise happen if the inputs were based on geometric distance measures (such as Euclidean). Note that two of the ghosts do use geometric distance measures, but since ghosts are not normally allowed to turn back on themselves, they cannot become stuck in this way.

Input	Description
$g_1 \dots g_4$	distance to each predatory ghost
$g_1 \dots g_4$	distance to each edible ghost
x, y	location of current node
pill	distance to nearest power pill
junction	distance to nearest junction

Table 2. *The feature vector. Each input is the shortest-path distance to the specified object, or set to the maximum possible distance if that object does not currently exist in the maze.*

3.3 Evolutionary Algorithm

Our Evolutionary Algorithm (EA) is an $(N + N)$ evolutionary strategy [1]. Experiments were done with $N = 1$ and $N = 10$. The case of $N = 1$ results in a simple random mutation hill-climber. These hill-climbers can outperform more complex EAs in some cases [8], but they have to be applied carefully in this domain, due to the high-levels of noise present in making fitness evaluations. For this reason, we investigate the effects of playing multiple games per fitness evaluation, which reduces noise. The ES was also run with $N = 10$, and this was found to give better evolutionary progress, with the larger population also quenching the effects of noise. Using an ES removes the need to define a crossover operator. This would not be a problem for the single-layer perceptrons, but naively defined crossover operators are typically destructive when evolving MLPs, due to the problem of competing conventions [11].

The overall objective was to evolve the best player possible, where quality of play is measured by average score over a significant number of games (e.g. 100). The fitness function used by the EA was the average score over a number of games, where the number of games was either 1, 5, or 50.

3.4 Neural Networks to be Evolved

The neural networks were implemented in object-oriented style with the basic class being a Layer, consisting of a weight matrix mapping the inputs (plus bias unit) to the output units, and a *tanh* non-linearity being applied to the net activation of each output. A single layer perceptron then consisted of just one such layer, a multi-layer perceptron consisted of two or more such layers, though experiments in this paper were limited to a maximum of two layers (hence one hidden layer in standard terminology).

For initial members of the population, all weights were drawn from a Gaussian distribution with zero mean and standard deviation equal to the reciprocal of the square root of the fan-in of the corresponding node. A mutated copy

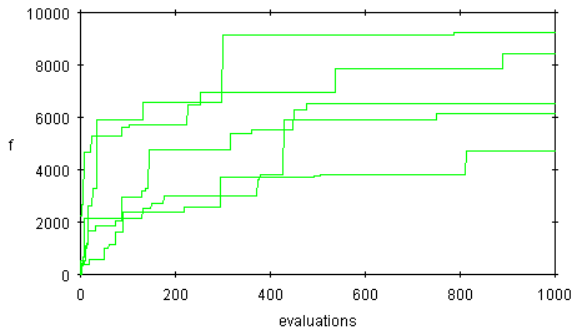


Figure 1. *Fitness evolution of a perceptron for the deterministic game.*

of a network was made by first creating an identical copy of the network and then perturbing the weights with noise from the same distribution. Weights to be mutated were selected in one of four ways as follows (probabilities in parentheses): all the weights in the network (0.1), all the weights in a randomly selected layer (0.27), all the weights going into a randomly selected node (0.27), or a single randomly selected weight (0.36). While these parameters could have been adapted (or indeed, the noise distribution for each weight could have been adapted), these adaptations can also have detrimental effects on the convergence of the algorithm when high levels of noise are present, as is the case here.

4 Results

Results are presented for evolving neural network location evaluators, first for the deterministic game, then for the non-deterministic game.

4.1 Evolving Players for Deterministic Play

As might be expected, evolving strategies against deterministic ghosts is easier than when the ghosts exhibit a small amount of non-determinism. Effective ghosts chase aggressively most of the time, while occasionally making a surprise move in order to disrupt any patterns. To test this we experimented with evolving players against deterministic ghosts. Figure 1 shows the fitness evolution of a perceptron against deterministic ghosts. Since, for a given neural network, every run of the game is identical, only one game run per fitness evaluation is required. The maximum fitness (score) attained was 9,200. Note that when played in the non-deterministic game, this player did not have especially high fitness, with an average score (over 100 games) of only 1,915 (Table 3).

4.2 Simple Ghost Avoidance

Before evolving neural networks for the non-deterministic game, tests were run to see how well a simple ghost avoidance strategy would perform. To this end, a hard coded controller was implemented, which scored a node as the shortest path distance to the closest ghost, and then chose the node that maximised that score. This strategy performed poorly, however, as shown in Table 3 (Simp-Avoid), with an average score of only 980. This was thought to be a worthwhile test in order to demonstrate that the neural networks are learning something more than this simple, if somewhat flawed, ghost avoidance technique.

4.3 Evolving for Non-Deterministic Play

In the non-deterministic game, the aim is to learn general behaviours, but the noise makes progress difficult. Increasing the number of games per fitness evaluation improves the reliability of the fitness estimate, but means that fewer fitness evaluations can be made within the same time.

Figure 2 shows the effects that the games per fitness evaluation has on evolving an MLP with twenty hidden units, using a $(1 + 1)$ ES. The graph shows plots for 1, 5 and 50

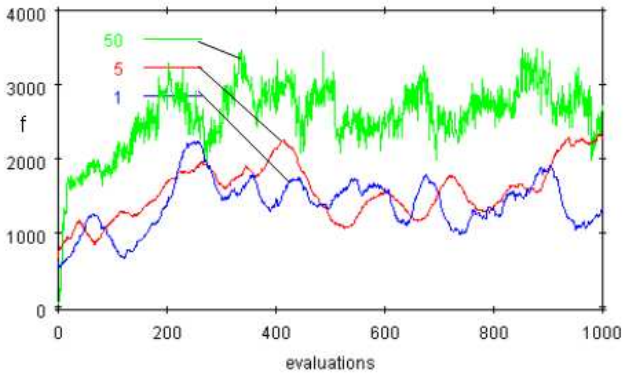


Figure 2. *Evolving an MLP (20 hidden units) with a $1 + 1$ ES, using 1, 5, and 50 games per fitness evaluation.*

games per fitness evaluation, and indicates that a large number is needed in order for evolution to make progress, with best performance being obtained when we run 50 games per evaluation. Note that the graphs for 1 and 5 games have been averaged using a smoothing window of 25 samples either side of the centre. This is to present a better picture of the average fitness at each point on the graph. We ran the same $(1 + 1)$ experiments for a perceptron, and found that this was able to significantly outperform the MLP, but

that the overall trend was the same, in that making many fitness evaluations per game (e.g. 50) was essential in order to make good evolutionary progress.

An alternative that seems to work better, is to increase the population size instead. Figure 3 plots average fitness versus generation when using a $10 + 10$ ES to evolve an MLP with 20 hidden units. Error bars are shown at $(+/-)$ one standard error from the mean, to give an idea of the range of performance present in each generation. Experiments were also made with 5 and 10 hidden units, but 20 seemed to perform best, though this was not thoroughly analysed. For this graph, 5 games per fitness evaluation were used, and therefore the total number of games played was identical to the 50 game plot of Figure 2. Unlike the $1 + 1$ ES, the larger population is able to smooth the effects of noise. A similar graph is observed when evolving a single layer perceptron using the $10 + 10$ ES (not shown), but it was consistently found that the best evolved MLP out-performed the best evolved perceptron. While each evolutionary run produces neural networks that have different behaviours, and even a particular network can behave rather differently on each game run, there are some interesting trends that can nonetheless be observed, and these are further discussed in Section 4.4.

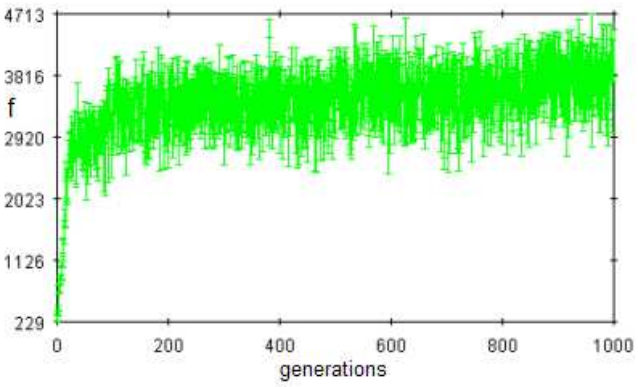


Figure 3. *Evolving an MLP (20 hidden unit) with a* $10 + 10$ *ES, using* 5 *games per fitness evaluation.*

Table 3 shows the performance statistics for three evolved controllers and one hand-designed controller. MLP-20 (20 hidden units) is one of the best MLPs found with the $10 + 10$ ES. This is significantly better than the evolved perceptron (Percep-LC) - where LC stands for Level-Clearer - as explained below. Next we have the perceptron evolved on the deterministic game, but tested on the non-deterministic game (Percep-Det), and finally the hand-designed weights, and Simp-Avoid, both of which perform very poorly.

Controller	min	max	Mean	s.e.
MLP-20	2590	9200	4781	116
Percep-LC	1670	8870	4376	154
Percep-Det	440	5140	1915	90
Hand-Coded	400	3540	1030	58
Simp-Avoid	480	1320	980	82

Table 3. *Controller performance statistics (measured over 100 runs of the non-deterministic game).*

4.4 Evolved Behaviours

One of the most surprising behaviours that sometimes evolves is the ghost chaser strategy. This exploits the fact that when a ghost reaches a junction, it is not normally allowed to retrace its path (unless all ghosts are reversing). Therefore, especially when the ghosts are bunched together, the Pac-Man can safely chase them for a while - until all the ghosts suddenly reverse, at which point, the Pac-Man should make a hasty retreat, but this particular evolved strategy fails to do this and causes the Pac-Man to be immediately eaten instead.

A more successful strategy has been dubbed the Level-Clearer (Percep-LC), owing to its ability to regularly clear the first level of the game, although it usually loses a few lives in the process, and therefore does not usually clear the second level (even though all levels are currently identical). Also, it is rather poor at eating ghosts, and so does not achieve very high scores relative to its progress through the game. The best evolved controller, MLP-20, on the other hand is not as good at clearing a level, but much better at eating ghosts.

4.5 Interpreting the Evolved Neural Network

Recall that the evolved perceptron assigns a value to each possible neighbour of the Pac-Man's current location. To gain more insight into the kind of judgements it is making we can plot the score for every node in the network, which is independent of the current Pac-Man location. The intensity gradients then show the immediate route the Pac-Man will follow, and show the current ratings of the rest of the maze locations. Note that in many cases, all ratings fall into a small range, but only the order of the nodes matters, not their absolute values. For display purposes, we map the range of scores into the full range of grey levels from black to white.

Figure 4 shows these scores during a sample game run for an evolved perceptron controller. The Pac-Man is following a gradient towards some pills and an edible ghost. In this game the Pac-Man next proceeds to eat the two nearest edible ghosts, but is then eaten by the red ghost (which has

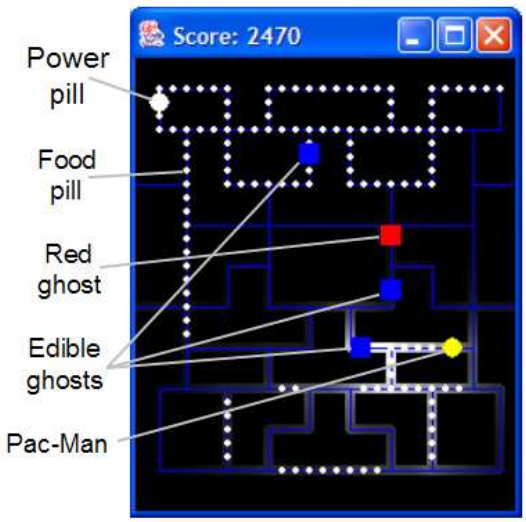


Figure 4. *The node scores overlaid on the Maze as background intensity.*

just itself been eaten by the Pac-Man, and is on the chase again having been reset to the centre of the maze).

Since the single layer perceptron is a linear weighting of the input features, we can inspect these to gain insight into what the system has learned. Table 4 shows the weights for the evolved level clearing perceptron, together with a set of hand-designed weights. One of the most surprising observations is that the evolved controller actually makes Pac-Man *attracted* to the strongest ghost (g_1), as well as the random ghost (g_4). The strongest attraction is towards pills (-15), while unexpectedly, the Pac-Man is mildly repelled by junctions. This was unexpected, since junctions have more escape routes than corridors and are therefore safer places to be. Note that the Pac-Man is attracted to only three out of four edible ghosts - again a surprise. The evolved controllers do vary from run to run, but many display these counter-intuitive features. When observing the game-play, it became clear why there is sometimes an advantage in being attracted to ghosts - it enables the Pac-Man to lure the ghost to a power-pill - and subsequently eat the ghost. This is a dangerous policy, however, and inevitably leads the Pac-Man to unnecessary death on many other occasions.

With these considerations in mind, a hand-designed controller was constructed, with the weights also shown in Table 4. This controller performed poorly, however, as can be seen from the statistics in Table 3.

Parameter	Designed	Evolved
g_1 (red)	1.0	-0.29875
g_2 (pink)	0.8	2.88190
g_3 (blue)	0.6	2.60610
g_4 (orange)	0.4	-0.31166
e_1	-1.0	-1.72596
e_2	-1.0	-0.66969
e_3	-1.0	-1.36305
e_4	-1.0	0.21379
x	0.0	-0.35274
y	0.0	3.48056
pill	-0.1	-15.18330
power	-1.0	-3.95439
junction	-1.0	1.31463

Table 4. *Designed and evolved perceptron weights (see Table 2 for description of parameters).*

5 Discussion and Future Directions

Perhaps the least satisfactory aspect of the current work is the fact that the input vector has been hand-designed. This was done in order to enable fast evolution of behaviours, but also places limits on how good the evolved controllers ultimately become. One possibility would be to feed the raw 2D array of pixels to the network, and see what it can learn from that, although the answer may well be 'not very much'. Note however, that this approach has been applied with some success in [4], where they evolved a car controller given the visual scene generated by a 3D car simulator as input to their evolved active vision neural network. In that case, the main thing it learned was to follow the edge of the pavement. Learning a good Pac-Man strategy, however, is a rather different kind of challenge.

A promising direction currently being investigated is using the connectivity of the nodes in the maze as the basis for a modular recurrent neural network. The idea is that the current location of the objects (pills, power-pills, ghosts and Pac-Man) will propagate a signal-vector through the neural maze-model, where the features of the vector and its propagation characteristics would be evolved.

Another interesting issue is that of the best space for evolution to work in. It is not clear that neural networks are the most appropriate structures for evolving good behaviours. It may be that expression trees or function graphs involving logical, comparator and arithmetic functions are a better space to work in, since much of the reasoning behind location evaluation might be reducible to making distance comparisons between various objects in the maze and acting accordingly on the basis of those comparisons.

There is currently a strong trend in evolving neural network game players where fitness is a function of game per-

formance. By contrast, reinforcement learning techniques such as temporal difference learning utilise information available during the game to update the network parameters, which can ideally lead to faster and more effective learning than evolution [10].

Regarding ghost behaviours, the current ghosts tend to crowd together too much, making the game too easy for a human player, at least when slowed down. A future version will use ghosts that repel each other, in order to spread out more. Also, a very natural development would be to co-evolve ghost strategies, where the fitness is assigned to a team of ghosts whose objective is to minimise the score that the Pac-Man can obtain.

6 Conclusions

This paper described an approach to evolving Pac-Man playing agents based on evaluating a feature vector for each possible next location given the current location of the Pac-Man. The simulation of the game retains most of the features of the original game, and where there are differences, our version is in some ways harder (such as the ghosts traversing the tunnels at full speed).

As expected, whether the ghosts behave deterministically or not has a significant effect on the type of player that evolves, and on the typical score that players achieve. The non-deterministic version is much harder, since a strategy must be learned which is generally good under lots of different circumstances.

On the other hand, one of the features of simulated evolution is its ability to exploiting any loopholes in the way a system has been configured. Evolution does indeed exploit the deterministic game in this way. The neural network weights, together with the maze and the ghost behaviours, dictate in a complex and indirect way the exact route that the Pac-Man will take, and hence the ultimate score obtained. While it is impossible to foresee the effects that even minor changes in neural network weights will have on the way the game unfolds, without actually simulating it, the fitness function cares only about the final score, and exploits any changes that happen to be advantageous.

While it is harder to evolve controllers for the non-deterministic game, it is a more interesting domain to study, since evolved controllers should now encode generally good strategies, rather than particular routes. However, the non-determinism causes high-levels of noise, which makes evolutionary progress more problematic. This would be less of a problem if the game consisted only of eating pills, but eating multiple ghosts can cause major differences in the score that could be due either to differences in strategy, or just luck.

Ms. Pac-Man is an interesting and challenging game to evolve controllers for, and much work remains to be done in this area. This paper explored the use of a purely reactive neural network for controlling the Pac-Man, but there are many other possible approaches, and it would also be interesting to investigate how game-tree search techniques could be adapted to cope with the large state space of the game.

Acknowledgements

I thank the anonymous reviewers and the members of the Natural and Evolutionary Computation group at the University of Essex, UK, for their helpful comments on this work.

References

[1] H.-G. Beyer. Toward a theory of evolution strategies: The (mu, lambda)-theory. *Evolutionary Computation*, 2(4):381–407, 1994.

[2] J. S. D. Bonet and C. P. Stauffer. Learning to play Pac-Man using incremental reinforcement learning., 1999. `http://www.ai.mit.edu/people/stauffer/Projects/PacMan/`.

[3] K. Chellapilla and D. Fogel. Evolving an expert checkers playing program without using human expertise. *IEEE Transactions on Evolutionary Computation*, 5:422 – 428, (2001).

[4] D. Floreano, T. Kato, D. Marocco, and E. Sauser. Coevolution of active vision and feature selection. *Biological Cybernetics*, 90:218 – 228, 2004.

[5] M. Gallagher and A. Ryan. When will a genetic algorithm outperform hill climbing? In *Proceedings of IEEE Congress on Evolutionary Computation*, pages 2462 – 2469. 2003.

[6] J. Koza. *Genetic Programming: on the programming of computers by means of natural selection*. MIT Press, Cambridge, MA, (1992).

[7] J. Laird. Using a computer game to develop advanced AI. *Computer*, 34:70 – 75, 2001.

[8] M. Mitchell, J. Holland, and S. Forrest. When will a genetic algorithm outperform hill climbing? In J. Cowan, G. Tesauro, and J. Alspector, editors, *Advances in Neural Information Processing Systems 6*, pages 51 – 58. Morgan Kaufman, San Mateo, CA, 1994.

[9] J. B. Pollack and A. D. Blair. Co-evolution in the successful learning of backgammon strategy. *Machine Learning*, 32:225–240, 1998.

[10] T. Runarsson and S. Lucas. Co-evolution versus self-play temporal difference learning for acquiring position evaluation in small-board Go. *IEEE Transactions on Evolutionary Computation*, page Submitted August 2004.

[11] J. Schaffer, D. Whitley, and L. Eshelman. Combinations of genetic algorithms and neural networks: a survey of the state of the art. In *Proceedings of COGANN-92 – IEEE International Workshop on Combinations of Genetic Algorithms and Neural Networks*, pages 1 – 37. IEEE, Baltimore, (1992).

[12] G. Tesauro. Temporal difference learning and td-gammon. *Communications of the ACM*, 38(3):58–68, 1995.

Co-evolutionary Strategies for an Alternating-Offer Bargaining Problem

Nanlin Jin
Department of Computer Science
University of Essex
Colchester CO4 3SQ, UK
njin@essex.ac.uk

Edward Tsang
Department of Computer Science
University of Essex
Colchester CO4 3SQ, UK
edward@essex.ac.uk

Abstract- In this paper, we apply an Evolutionary Algorithm (EA) to solve the Rubinstein's Basic Alternating-Offer Bargaining Problem, and compare our experimental results with its analytic game-theoretic solution. The application of EA employs an alternative set of assumptions on the players' behaviors. Experimental outcomes suggest that the applied co-evolutionary algorithm, one of Evolutionary Algorithms, is able to generate convincing approximations of the theoretic solutions. The major advantages of EA over the game-theoretic analysis are its flexibility and ease of application to variants of Rubinstein Bargaining Problems and complicated bargaining situations for which theoretic solutions are unavailable.

1 Introduction

There are several methodologies applied for studying games. One of them is to evolve players' strategies in a way that simulates the natural evolution. Evolutionary Algorithm (EA) have proven effective for a wide variety of problems.

The purpose of our research is to apply Evolutionary Algorithms, specifically Co-evolutionary Algorithms, to solve an Alternating-Offer Bargaining Problem. To this problem, the assumptions of the game-theoretic solutions Subgame Perfect Equilibrium (SPE) are relaxed by equipping players with imperfect abilities of game theoretical reasoning. Delays and any possible divisions of a cake, not limited to SPE, are therefore possible. Having used a co-evolution adaptive system in which bargaining players learn "how to bargain from experiences" while competing against each others, we study experimentally how the discount factors affect the bargaining outcomes and compare those with the theoretical SPE. The findings reveal that the evolutionary computation approach is a convincingly complementary and approximating tool potentially able to tackle bargaining problems which would require excessive efforts if approached by traditional game-theoretic methods.

We start with brief descriptions to these methods and then focus on applying an evolutionary algorithm to an infinite extensive-form game, Alternating-offers bargaining problems.

1.1 Game theory

Game theory is a branch of mathematics that uses models to study interactions with formalized incentive structures ("games"). Game theory is important to many fields, including economics, biology, politics and computer science.

1.2 Analytical method

Von Neumann and Morgenstern first formalizes two-person zero-sum games and presents their theoretical optimal solutions for ideally rational players by equilibrating through mathematical reasoning ([Von Neumann & Morgenstern 1944]).

Game theorists solve games, assuming that every involver has "Perfect" rationality as an 'economic man' who typically has complete information relevant to problems, full computing capacity and well-defined and stable system of preferences. Rational players know all involvers are rational and know the rules of the game ([Simon 1955]).

[Nash 1950] formulates Nash Equilibrium for multi-player games. Complex problems probably have multiple Nash Equilibriums. How to select equilibrium becomes a problem. Theorists have proposed different definitions of rationality to eliminate some equilibrium in order to refine the Nash equilibrium.

Game-theoretic analysis normally requires substantial time and costs. Theorists spend years to present a particular equilibrium for a particular situation. Substantial efforts may be required to find equilibriums for slightly modified situations.

1.3 Behavioral method

Some psychologists, along with social scientists and experimental economists collect data from human answers to questionnaires and competitions ([Simon 1982], [Barkow et al 1992], [Kagel & Roth 1995]). They observe and analyze *actual* human behaviors and try to explain why in some (simple) cases, people learn to perform better, *as if* they know theoretical equilibriums. In some other (often complicated) situations, people give intuitively reasonable responses, not using rational choices.

1.4 Evolutionary game theory

Maynard Smith and Price (1973) initiated *Evolutionarily Stable Strategy* (ESS) which is the most influential work since the Nash Equilibrium in Game Theory. Unfortunately, like traditional game-analytical methods, ESS does not explain *how* a population adapts to such a stable strategy [Weibull 1995]. Using ESS theory, one can check whether a strategy is robust to continually evolutionary pressures.

In the real world however, an ESS may not dominate in a population during a certain period of time, due to strong stochastic components emerging in evolutionary process. [Fogel et al. 1997] and [Fogel et al. 1998] show that "even in simple games, ESSs may not be stable under conditions that are pertinent in the real-world, such as finite population size and culling selection. Under proportional selection, large finite populations may tend to vary around an ESS, but large can be on the order of 5000 or more individuals in a population."

1.5 Evolutionary Algorithm Simulations

Evolutionary Algorithms refer to a class of algorithms which are inspired by natural evolution. Related methods to this work are Genetic Programming (GP) [Koza 1992] [Langdon & Poli 2001] and Genetic Algorithms (GA) [Holland 1975]. Evolution Algorithms are different from ESS, although both are rooted in evolutionary biology. To use an EA approach to solve games can be regarded as a way of simulation, which do not necessarily converge to game-theoretic equilibriums or ESS.

[Axelrod 1987] studies normal-form repeated games. His GA experimental results and [Miller 1996]'s results coincide with some reciprocity phenomena shown in human entries of Iterated Prisoners Dilemma (IPD) competitions. [Koza 1992] investigates a finite, extensive-form and no-repeated games with complete and perfect information, for which he finds Subgame Perfect Equilibrium using Genetic Programming (GP).

In this work, we attempt to employ Evolutionary Algorithm to study an infinite extensive-form two-person game with complete and perfect information: *Basic Alternating-Offer Bargaining Problem* (BAOBP), or Rubinstein Bargaining Problem [Rubinstein 1982] whose *Subgame Perfect Equilibrium* (SPE) is known. We start by introducing the BAOBP and its SPE. EA framework and a co-evolving system for BAOBP are developed, after which experimental outcomes are analyzed. Conclusions and future work will be given in the end.

2 Bargaining Problems

Bargaining problems study a class of situations where participants have common interests but conflict over how to divide the interest among them. Participants try to achieve agreements through negotiation. [Nash 1950] formulates the Nash Bargaining Problem and [Rubinstein 1982] models and solves the Basic Alternating-Offer Bargaining Problem. Based on these, other researchers study more complex bargaining situations.

2.1 Alternating-Offer Bargaining Problem

BAOBP describes a bargaining scenario in which the participant A makes an offer or count-offers to the player B on dividing a cake $\pi = 1$ at time of 0, 2, 4, 6, ... B makes a counter offer at time 1, 3, 5, 7, ... The bargaining process ends once an offer or a counter-offer is immediately accepted by the other player. A proposal on division by the player i is x_i for himself and $x_j = 1 - x_i$ for the other j. Player i's discount factor δ_i is his bargaining cost per time interval, $\delta_i \equiv e^{-r_i}$ where r_i is the player i's discount rate. The payoff gained by player i who has a share of x_i from the agreement, reached at time t is determined by the payoff function: $x_i \delta_i^t$.

2.2 Assumptions and Subgame Perfect Equilibrium

[Muthoo 1999] characterizes solutions to BAOBP problem by satisfying two properties: "no delay" and "stationarity". No delay means that "whenever a player has to make an offer, her equilibrium offer is accepted by the other player". Stationarity requires "in equilibrium, a player makes the same offer whenever she has to make an offer". Theorists mathematically analyze Subgame Perfect Equilibrium under such strong assumptions, in which players should offer nothing other than the perfect equilibrium partition and for sure will be accepted at time 0. Partitions are guaranteed before a bargain even starts, given the discounts factors.

The unique equilibrium taken as the Game-theoretic formula solution of this game is a Subgame Perfect Equilibrium in which the first player A obtains:

$$x_A^* = \frac{1 - \delta_B}{1 - \delta_A \delta_B}$$

and the second player B gets:

$$x_B^* = 1 - x_A^*$$

Technical treatments and proofs are available in [Rubinstein 1982], [Muthoo 1999], and [Bierman 1998].

3 Evolutionary Algorithms

Evolutionary algorithms are a population-based improvement mechanism. Individuals are selected based on their performance (fitness). Better individuals have higher probability to be selected as "raw material" to breed new offspring for the forthcoming generation. The offspring are created by the genetic operators (crossover and mutation) on the "raw" genetic material. Evolution pushes individuals (more specifically, the genetic materials) to continue improving their adaptation to the environments or objectives in order to survive. The improvement of individuals illustrates the process of acquiring behavior patterns adaptive to the environments.

In many applications, Evolutionary Algorithms are used as stochastic search methods, which are proposed to produce near-optimal solutions to a given problem. Given the size of the search space (depending on how strategies are represented), exhaustive search is normally impractical. An efficient approach able to search acceptable strategies within a reasonable time is therefore needed. EA are chosen not only because they have succeeded in many other applications, but also because they are expected that the same mechanism is applicable to slightly modified scenarios.

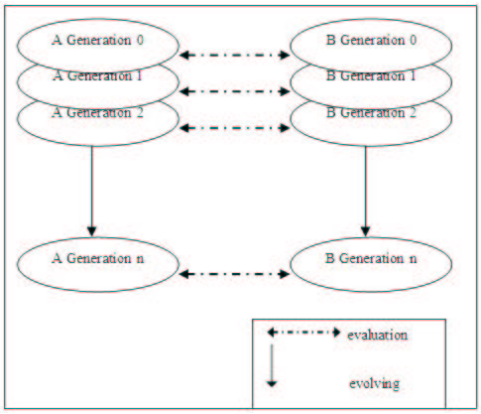


Figure 1: The diagram of Co-evolution

3.1 Co-evolution

Natural co-evolution is the mutual evolutionary influences between two species dependent on each another. The survival skills from co-evolving species in nature inspire scientists to borrow co-evolution principles to solve problems in which two elements are greatly interacting with each other. Figure 1 shows in an idealized two species situation, the species A and B are coexisting. One species's fitness is its current adaptation to the other species that is evolving simultaneously. Computer scientists have modeled [Schmitt 2004] and provided experimental outcomes to substantiate that co-evolution is practically good in some games. [Tsang & Li 2002] successfully developed a co-evolutionary system EDDIE/FGP which aids investors to seek dealing rules in financial markets.

4 Experiments

In this classical model of alternating-offers bargaining proposed by Rubinstein, the discount factors, specifying the respective costs subject to bargaining time for the two players, are the only elements that determine their bargaining powers. In this model, the theoretical SPE is unique and can be expressed analytically. However the assumptions of this model are too idealized hence other equilibriums and possibilities, which would arise in more realistic assumptions, are ruled out.

4.1 Assumptions to Players

Subsection 2.2 has provided the game-theoretic solution. The unique point prediction P.E.P analyzed by game theoretical method is a compelling one because it is difficult to see why perfectly rational players knowing all involvers are rational and knowing the rules of the game will do anything else. However, it is the assumptions that make applicators of the theory unlikely to believe provided that realistic players often lack the perfect rationality assumed. So, more realistic settings of assumptions have to be made in order to see whether strategies of players with Bounded Rationality converge to SPE. We hypothesize there are reciprocal interactions between players' behaviors, and players learn through trial-and-error experiences.

In our experiments, strategies have dynamic behaviors and can propose any division within the size of a cake rather than "stationarity"; bargaining procedure can last at most 10 time intervals (due to computational resources) instead of no delay; and players are allowed to use various strategies. Instead of assuming certain rationality of players, we hypothesize that

- Players in bargaining problems have a very low level of intelligence and are incapable of game-theoretic reasoning;
- The only goal of a player is to maximize his overall payoffs;
- Players learn through trial-and-error experiences over generations;
- There are reciprocal interactions between bargaining players' behaviors, like co-adapting organisms;
- A strategy has no ability of identifying opponents' strategies;
- A strategy has no memory of historical behaviors of the opponent's strategy in the undergoing bargaining;
- A strategy is unable to adjust its behaviors during a bargaining procedure. In other words, a strategy is a function without any parameters responding to its opponent's actions.

Using this set of assumptions avoids the difficulties of defining rationality.

4.2 Experimental Set-up

We build a two-population co-evolutionary system implemented by Genetic Programming (GP). Each player has his own population: a strategy pool consisting of candidate solutions. The strategies of the two players evolve at the same time. In this system, the objective of the strategies is to maximize the payoff from bargaining agreements. Well-performed strategies are chosen under the guideline of selection and undergo progressive modifications to be more competitive in forthcoming bargaining.

Genetic parameters are stated in Table 1.

Game parameters: 10 pairs of discount factors are chosen, which are shown in the Table 2. In theory, $0 < \delta_i < 1$. $\delta_i = 0.1$ means that a cake with a size 1 will shrink to be 0.1 after one time interval for the player i and $\delta_i = 0.9$ means that the cake will be 0.9 after one time interval. There are two examples of a low and a high discount factors.

Representation: An individual $g_i \in I$ is a genetic program in player i's population I. Its corresponding strategy

Table 1: GP parameters

Parameter	Value
Population Size	100
Number of Generations	300
Function Set	{ +, -, ×, ÷ (Protected) }
Terminal Set	{1, -1, δ_A, δ_B }
Initial Max Depth	5
Initialization Method	Grow
Selection Method	Tournament
Crossover rate	0 to 0.1
Mutation rate	0.01 to 0.5
Maximum nodes of a GP program	50

is $s(g_i)$. In order to make the search space smaller, currently we evolve only g_i. A time-dependent strategy of player i is $s(g_i) = g_i \times (1 - r_i)^t$, where t is time, an non-negative integer.

Fitness Function: A strategy's performance highly depends on other strategies whom it meets. The design of using a group of fixed representative strategies as the fitness assessment has a risk that evolution may exploit the weaknesses of the pre-defined representatives, but perform poorly against others. So the fitness of a strategy should be based on its performance against the opponent's co-evolving strategies at the same evolutionary time. In other words, for this bargaining problem, the relative fitness [Koza 1992] assessment is a fair choice. Game Fitness of a strategy $s(g_i)$, denoted by $GF(s(g_i))$ is defined as the average payoff of $s(g_i)$ gained from agreements with strategies in the opponent's population J which has a set of n number of bargaining strategies, $j \in J$:

$$GF(s(g_i)) = \frac{\sum_{j \in J} p_{s(g_i) \to s(g_j)}}{n}$$

where $p_{s(g_i) \to s(g_j)}$ is the payoff gained by $s(g_i)$ from an agreement with $s(g_j)$ which receives $p_{s(g_j) \to s(g_i)}$. An Incentive Method to handle constraints is used in defining the fitness function for all the individuals in the populations. Detailed designs of the fitness function are described in [Tsang & Jin 2004].

4.3 Observations

We have executed 100 runs with different random seeds for each game's setting chosen. For each game's setting, the average of shares x_A from final agreements made by the best-of-generation individuals from both populations at the 300^{th} generation is shown in the table 2. We observe that after 300 generations, the 100 x_As cluster around SPE, having minority of exceptions found. To our hypothesis that SPE is the same as the mean of our experimental shares, a t-test shows the t Critical value two-tail is 2.2621, larger than the t statistics value 1.3011. So our hypothesis is accepted at the 95% confidence level. Many experiments,

Table 2: The means of the shares x_As obtained by the best-of-generation individual in the population A at the 300^{th} generation

Discount Factors	SPE x_A^*	Experimental Average x_A	x_As' Deviation
(0.1, 0.4)	0.6250	0.9101	0.0117
(0.4, 0.1)	0.9375	0.9991	0.0054
(0.4, 0.4)	0.7143	0.8973	0.0247
(0.4, 0.6)	0.5263	0.5090	0.0096
(0.4, 0.9)	0.1563	0.1469	0.1467
(0.5, 0.5)	0.6667	0.6745	0.0271
(0.9, 0.4)	0.9375	0.9107	0.0106
(0.9, 0.6)	0.8696	0.8000	0.1419
(0.9, 0.9)	0.5263	0.5065	0.1097
(0.9, 0.99)	0.0917	0.1474	0.1023

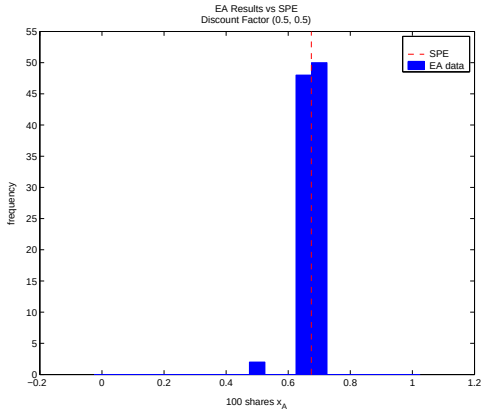


Figure 2: The distribution of 100 runs x_A at 300^{th} generations: δ_A = 0.5 and δ_B = 0.5. The vertical line x = 0.6667. is the SPE x_A^*.

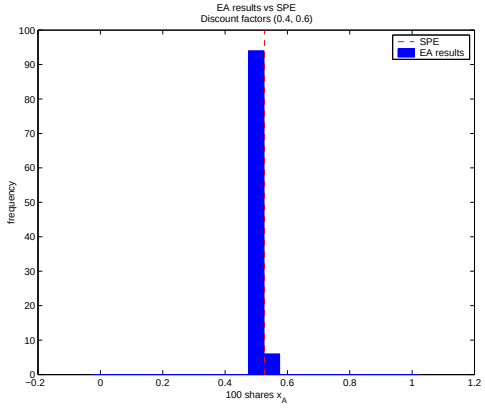


Figure 3: The distribution of 100 runs x_A at 300^{th} generations: δ_A = 0.4 and δ_B = 0.6. The vertical line x = 0.5263. is the SPE x_A^*.

Table 3: Bargaining Time for agreements made by the best-of-generation individuals from the two populations at the 300^{th} generation

Discount Factors	Average Time
(0.1, 0.4)	0.0000
(0.4, 0.1)	0.0000
(0.4, 0.4)	0.0000
(0.4, 0.6)	0.0000
(0.4, 0.9)	0.2121
(0.5, 0.5)	0.0000
(0.9, 0.4)	0.0100
(0.9, 0.6)	0.4700
(0.9, 0.9)	3.8500
(0.9, 0.99)	5.6100

the SPEs are within the distributions of experimental outcomes for example in Figure 2 and Figure 3. Therefore it is very likely that co-evolutionary might generate exact solutions as the theoretical ones, at a certain degree of precision. For game settings with extreme low or high discount factors, experimental results are far from the SPE predictions. In our experiments, extreme bargaining parameters refer to the sets of discount factors: (0.1, 0.4), (0.4, 0.4) and (0.9, 0.99). [Bragt et al. 2002] simulates the bargaining by a multi-agent evolving system that is implemented by real number-coded Genetic Algorithms. They have found similar results although their experiments only test the situations when the $\delta_A = 0.6$ or $\delta_A = 0.3$.

All experimental results clearly demonstrate the influence of discount factors upon bargaining powers: the player with higher discount factor, comparative to his opponent, attains a larger portion of cake. If both players have the same discount factors, the first player receives a larger part of division. This discovering is consistent with the analysis by bargaining theory [Muthoo 1999].

The discount factors also determine the negotiation time required for settlements (Table 3). Not all bargains reach an agreement at the time $t = 0$. Delays ($t > 0$) emerge as a consequence of players' preferences to higher payoff and expectations that higher payoff will obtain in future, by (mainly) those players who have high discount factor; i.e. relatively patient players. Impatient players, on the other hand, are eager to agree as soon as possible to avoid such delays due to relatively higher costs per time interval they should pay than that of patient players should. Any delay ($t > 0$) is costly to both of the two players. Thus the total sum of payoff they get is less than the size of cake, which means the cake is not divided efficiently.

Moreover, the EA approach opens a window to show the process of artificial evolving over time, which neither game-theoretic solution nor ESS can provide. In Figure 4 and 5, two players' discount factors are (δ_A, δ_B) = (0.9, 0.4). The horizontal line of 0.9375 and the horizontal line of 0.0625 are SPE for player A and B respectively. The values of g_i (labeled as GPT), the shares from agreements

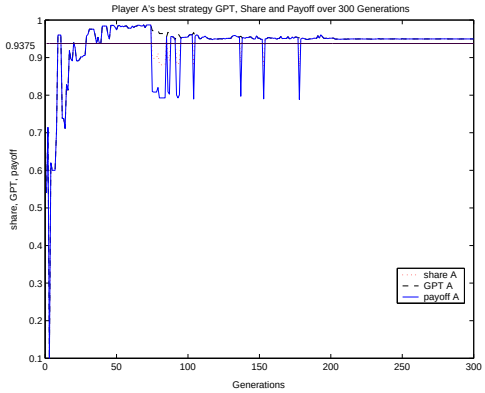


Figure 4: The best-of-generation individual of Population A over 300 generations. (δ_A =0.9 and δ_B = 0.4)

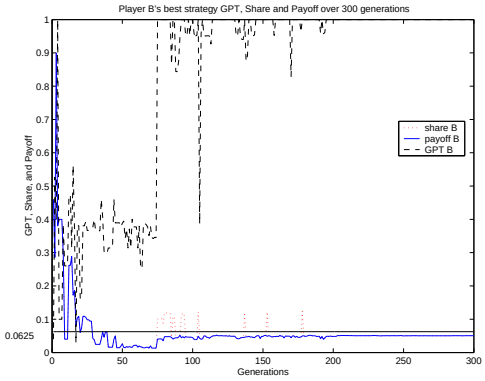


Figure 5: The best-of-generation individual of Population B over 300 generations. (δ_A =0.9 and δ_B = 0.4)

and the payoffs of the corresponding shares of the best-of-generation strategies of every generation are displayed. Strategies continually update themselves over time to co-adapt each other and come up with using relatively stable strategies, measured by fitness. We comment now on modifications of strategies' behaviors in this typical run. In the initial population strategies are generated randomly and for both players this means offering a deal of roughly 50% of the cake in average. Soon after, the player A learns that he can obtain more, because for him, delay is less costly as he has a higher discount factor than his opponent. He changes his first offers as much as he can in order to maximize his payoff. Finally, he approaches a value relatively close to the theoretical perfect equilibrium. Player B learns that she has to secure an agreement as soon as possible because it is not worthy for her to wait. Thus players finally reach at the point where both of them are willing to agree at time 0 when no bargaining costs occur. This fits nicely with the theoretical explanation.

5 Conclusions

Our objective was to compare the EA results with the Subgame Perfect Equilibrium solutions. The experimental observations show that co-evolutionary algorithms ideally provides approximate SPE solutions, excluding the game's set-

tings having very high and/or very low discount factors. Although co-evolution has not always evolved identical solutions to SPE in our experiments, it has given approximate alternatives. Moreover, this approach has provided interesting information concerning the influences of discount factors on bargaining time, the divisions of the cake and the evolving process. Compared with other research methods mentioned in the section 1, an EA approach has particular advantages: on much lower costs over human subjects' experiments; on less human intelligence over economics theorists and on its reusability and modifiability for complicated bargaining situations without knowledge of game-theoretic solutions such as variants of basic alternating-offer bargaining problems and Incomplete information bargaining problems.

In this work, we have only studied one bargaining problem, and emphasize that considerably more work will be required to determine the general utility of EAs in the problem domain. In future, we also plan to compare the EA results with actual behaviors by human-subject experiments.

Acknowledgement

This research is partly sponsored by University of Essex Research Fund. We benefited greatly from conversations with Abhinay Muthoo and Riccardo Poli. Alberto Moraglio, Yossi Borenstein and Vikentia Provizionatou have given important feedbacks on the draft of this paper. Mathias Kern helped with the use of LaTeX. Anonymous reviewer's comments are very helpful for the revision. Tim Gosling helped improve the quality of the final submission.

Bibliography

[Axelrod 1987] Axelrod, R. (1987) The evolution of strategies in the iterated prisoner's dilemma. In Lawrence Davis (Ed.), Genetic Algorithms and Simulated Annealing. Morgan Kaufmann.

[Barkow et al 1992] Barkow, J., Cosmides,L., Tooby, J. edited (1992) The Adapted mind : evolutionary psychology and the generation of culture, Oxford University Press.

[Bierman 1998] Bierman, H. and Fernandes, L. (1998) Game Theory with Economic Applications, Addison-Wesley, New York.

[Bragt et al. 2002] van Bragt, D., Gerding,E., and La Poutré, J. (2002) Equilibrium Selection in Alternating-Offers Bargaining Models: The Evolutionary Computing Approach, Electronic Journal of Evolutionary Modeling and Economic Dynamics (e-JEMED) 2.

[Fogel et al. 1997] Fogel, D., Fogel, G., Andrews, P., (1997) On the instability of evolutionary stable strategies, BioSystems 44, 135-152.

[Fogel et al. 1998] Fogel, G., Andrews, P., Fogel, D. (1998) On the instability of evolutionary stable strategies in small populations, Ecological Modelling 109, 283-294.

[Gibbons 1992] Gibbons, R. (1992) A Primer in Game Theory, Harvester Wheatsheaf.

[Holland 1975] Holland, J. (1992) Adaptation in Natural and Artificial Systems, the second edition, MIT Press.

[Koza 1992] Koza, J. (1992) Genetic Programming: On the Programming of Computers by Means of Natural Selection. The MIT Press.

[Langdon & Poli 2001] Langdon, W.B. and Poli, R. (2001) Foundations of genetic programming, Springer.

[Miller 1996] Miller, J. (1996) The Co-evolution of Automata in the Repeated Prisoner's Dilemma. Journal of Economic Behavior and Organization, 29(1), 87-112.

[Muthoo 1999] Muthoo, A. (1999) Bargaining Theory with Applications, Cambridge University Press.

[Nash 1950] Nash, J. (1950) The Bargaining Problem, Econometrica, 18: 155-162

[Rubinstein 1982] Rubinstein, A. (1982) Perfect Equilibrium in a Bargaining Model, Econometrica, 50: 97-110.

[Schmitt 2004] Schmitt, L. (2004) Classification with Scaled Genetic Algorithms in a Coevolutionary Setting. GECCO (2): 138-149

[Simon 1955] Simon, H. (1955) A Behavioral Model of Rational Choice, The Quarterly Journal of Economics, Vol. 69, NO.1, 99-118

[Kagel & Roth 1995] Kagel, J. and Roth, A editors (1995) The Handbook of Experimental Economics, Princeton University Press.

[Maynard Smith 1982] Maynard Smith, J. (1982) Evolution and the theory of games, Cambridge University Press.

[Nawa 2003] Nawa,N., Shimohara,K., and Katai, O. (2003) Diversity in Time Preferences and Fairness in Bargaining Games Played by Evolutionary Agents, The Second International Conference on Computational Intelligence, Robotics and Autonomous Systems.

[Simon 1982] Simon, H. (1982): Models of bounded rationality, Cambridge, Mass, MIT Press.

[Tsang & Jin 2004] Tsang, E., Jin, N. (2004) Incentive Method for Constraint Handling used with Evolutionary Algorithms, Technical report CSM-417, Department of Computer Science, University of Essex.

[Tsang & Li 2002] Tsang, E., Li, J. (2002) EDDIE for financial forecasting, in S-H. Chen (ed.), Genetic Algorithms and Programming in Computational Finance, Kluwer Series in Computational Finance.

[Von Neumann & Morgenstern 1944] Neumann, J. and Morgenstern, O. (1944) Theory of games and economic behavior, Princeton University Press.

[Weibull 1995] Weibull, J. (1995) Evolutionary game theory, Cambridge, Mass. : MIT Press.

[Wikipedia] Wikipedia, the free encyclopedia on line: http://en.wikipedia.org/

A New Framework to Analyze Evolutionary 2×2 Symmetric Games

Umberto Cerruti
Mathematics Department
University of Torino
Torino, Italy
umberto.cerruti@unito.it

Mario Giacobini
Information Systems Department
University of Lausanne
Laussane, Switzerland
mario.giacobini@unil.ch

Ugo Merlone
Statistics and Applied Mathematics Department
University of Torino
Torino, Italy
merlone@econ.unito.it

***Abstract*— In this paper we present a new framework to analyze the behavior of evolutionary 2×2 symmetric games. The proposed approach enanbles us to predict the dynamics of the system using the parameters of the game matrix above, without dealing with the concepts of Nash equilibria and evolutionary stable strategies. The predictions are in complete accordance with those that can be made with these latter concepts. Simulations have been performed on populations with spatial structures, and show a good agreement with the model's predictions. We also analyze the dynamics of a particular system, showing how effectively the framework applies to it.**

I. Introduction

Harrald [1] used genetic algorithms as evolutionary dynamics and gives a representation of players with limited memory in repeated games. His approach is based on binary representation of mixed strategy players and is extended in order to use deterministic finite automata for these games. Starting from this approach we introduce a different and, in some sense, more natural representation for players and are able to give an elegant analysis of the game evolution. Finally, we implement this framework using both Matlab and C++ and compare the results. The structure of the paper is the following: in Section II we recall some fundamental notions of evolutionary game theory and present Harrald's contribution; in Section III we discuss some of Harrald's assumptions and present our approach. In Section IV we introduce a spatially structured evolutionary algorithm and give a formal description of the evolutionary system. Section V is devoted to the simulation analysis and conclusions are given in Section VI. The Appendix concerns some consequences of floating point arithmetic error we encountered in our implementations.

II. Probabilistic Players

Let's consider the general form of a 2×2 symmetric game where the two players always choose from the same action set, say $\{X, Y\}$, with the payoff matrix as depicted in Table I.

	X	Y
X	e	g
Y	h	f

TABLE I
The general payoff matrix for a symmetric 2×2 game.

In [1], Paul Harrald proposed an approach based on probabilistic strategies. A player's strategy is no longer deterministic, and becomes a probability of playing action X, regardless of the past actions played by both players. With the exception of the cases where the strategy probability is either 0 (i.e., always play action Y) or 1 (i.e., always play action X), the resulting strategies are mixed. A strategy is represented as a binary chromosome of fixed length L: the binary string is decoded into an integer value that is then divided by $2^L - 1$, so to obtain the actual value of the strategy. In a panmictic (i.e., not spatially distributed) population, each agent in the population plays against each other agent in a repeated game for a fixed number of iterations, obtaining a total payoff representing his fitness. During a game, each player determines his moves randomly choosing between the two actions X and Y with the probability encoded in his chromosome. By tournament selection, couples of parents are selected according to their fitness values. Two offspring are obtained from each couple of parents using one-point crossover, and each bit of their chromosome is mutated by standard binary mutation. The obtained offspring population is then considered as the new population of the next generation.

We consider symmetric bimatrix games $G(I, S, \pi)$, where $I = \{1, 2\}$ is the player set, consisting of two players, S is the pure strategies space and π is the combined payoff function fully represented by the associated payoff matrix pair (R, C), where $C = R^{\mathrm{T}}$ (see [2] for details). As usual, the set of mixed strategies for player i is denoted Δ_i and, since we restrict our attention to symmetric games, it holds $\Delta := \Delta_1 = \Delta_2$

In this class of games we define the *symmetric Nash equilibrium* as any strategy pair $(x, y) \in \Delta^2$ such that $x \in \beta(y)$ and $y \in \beta(x)$ where $\beta(\cdot)$ is the best reply correspondence, which maps each mixed strategy to the face of Δ which is spanned by the pure best reply to '$\cdot$'. Finally an *evolutionary stable strategy* (ESS) is a strategy $x \in \Delta$ such that for every strategy $y \neq x$ there exists some $\bar{\epsilon}_y \in (0, 1)$ such that for all $\epsilon \in (0, \bar{\epsilon}_y)$ it holds:

$$x \cdot R(\epsilon y + (1 - \epsilon) x) > y \cdot R(\epsilon y + (1 - \epsilon) x)$$

III. Probabilistic Payoffs

Two main criticisms can be raised to Harrald's evolutionary machinery:

- Since a player has no memory of the previous moves in a game, there is no need to make each couple of opponents play all the iterations of a game. In fact, given a big enough number of play iterations, if we denote the probabilities of the two players A and B with p_A and p_B respectively, then we can approximate the expected gain of player A according to the game described by the matrix in Table I by the expression:

$$ep_Ap_B+gp_A(1-p_B)+h(1-p_A)p_B+f(1-p_A)(1-p_B). \quad (1)$$

- The binary representation of the probability could not be the most suitable one (for a complete discussion on the representation choice see, for example, [3]). Other possible representation could be better suited, such as the real number one, as suggested by the author himself in the article.

Concerning the first criticism, let's consider a population $P(t)$ of N probabilistic players at generation t and denote p_i, the agent i's probability of playing action X. The expected gain of player i, when playing with agent j, follows from expression (1):

$$\begin{aligned} G(i,j,t) &= ep_i(t)p_j(t) + gp_i(t)(1-p_j(t)) + \\ &\quad + h(1-p_i(t))p_j(t) + f(1-p_i(t))(1-p_j(t)), \end{aligned}$$

therefore, the fitness (i.e., the sum of his payoffs against all other agents in the population) $f(i,t)$ of agent i at generation t is

$$f(i,t) = \sum_{j\neq i} G(i,j,t). \quad (2)$$

If we denote the sum of the probabilities of all players in the population at generation t with $U(t)$, equation (2) becomes

$$\begin{aligned} f(i,t) &= ep_i(t)(U(t)-p_i(t)) + \\ &\quad + gp_i(t)(N-1-U(t)+p_j(t)) + \\ &\quad + h(1-p_i(t))(U(t)-p_i(t)) + \\ &\quad + f(1-p_i(t))(N-1-U(t)+p_j(t)). \quad (3) \end{aligned}$$

Equation (3) gives an effective way of calculating the fitness of an agent of a given population without having to perform the actual games between the agent and all the other agents in the population.

It is well known (see for example [2]) that every 2×2 symmetric game can be normalized, and is equivalent to a doubly symmetric game, where the payoff matrix is symmetric. While this equivalence is proven in game theoretical context, it remains to be analyzed when considering dynamical evolutions. The new game, that is called reduced, has the payoff matrix displayed in Table II, where $a=e-h$ and $b=f-g$.

We decided to focus our attention on doubly symmetric games, given their relevance in evolutionary game theory. In

	X	Y
X	a	0
Y	0	b

TABLE II

The general payoff matrix for a reduced symmetric 2×2 game.

the case of a reduced symmetric game, the expected gain of player i playing against agent j at generation t is

$$G(i,j,t) = ap_i(t)p_j(t) + b(1-p_i(t))(1-p_j(t)),$$

thus, following the same reasoning done for equation (3), the fitness $f(i,t)$ of agent i at generation t is given by

$$\begin{aligned} f(i,t) &= ap_i(t)(U(t)-p_i(t) + \\ &\quad + b(1-p_i(t))(N-1-U(t)+p_i(t)). \quad (4) \end{aligned}$$

At generation t, let's define the mean agent $\bar{p}(t)$ of population $P(t)$ as the mean of the probabilities of the N agents of the population, i.e., $\bar{p}(t)=U(t)/N$. If we replace in equation (4) the value of $p_i(t)$ with the value of the mean agent $\bar{p}(t)$, and we divide by the constant factor $N-1$, we obtain the expression

$$F(\bar{p},t) = (a+b)\bar{p}^2(t) - 2b\bar{p}(t) + b, \quad (5)$$

proportional to the fitness of the mean agent at generation t. This equation determines a parabola with vertex V at abscissa $b/(a+b)$. Note that this value coincides with the value of the game.

The selection pressure of an evolutionary algorithm evolving this kind of agents' strategy will drive the mean agent of the population towards higher values on the parabola described by equation (5).

According to the possible values of the matrix parameters a and b in a reduced symmetric game we have the following four cases:

1) both a and b are positive: the parabola is concave and the evolution will depend from the mean agent $\bar{p}(0)$ of the initial population $P(0)$; if $\bar{p}(0) < b/(a+b)$ (the vertex of the parabola, i.e., the value of the game), then the evolution will be driven toward action Y; otherwise the evolution will be driven toward action X (see Figure 1(a));
2) a is negative and b is positive: the parabola is decreasing in the interval $[0,1]$. Therefore, whatever the initial population is, the evolution will be driven toward action Y (see Figure 1(b));
3) both a and b are negative: the parabola is convex, since its vertex is inside the interval $[0,1]$. Thus, whatever the initial population is, the evolution will be driven toward the vertex, i.e. the value of the game (see Figure 1(c));
4) a is positive and b is negative: the parabola is increasing in the interval $[0,1]$. Therefore whatever the initial population is, the evolution will be driven toward action X (see Figure 1(d)).

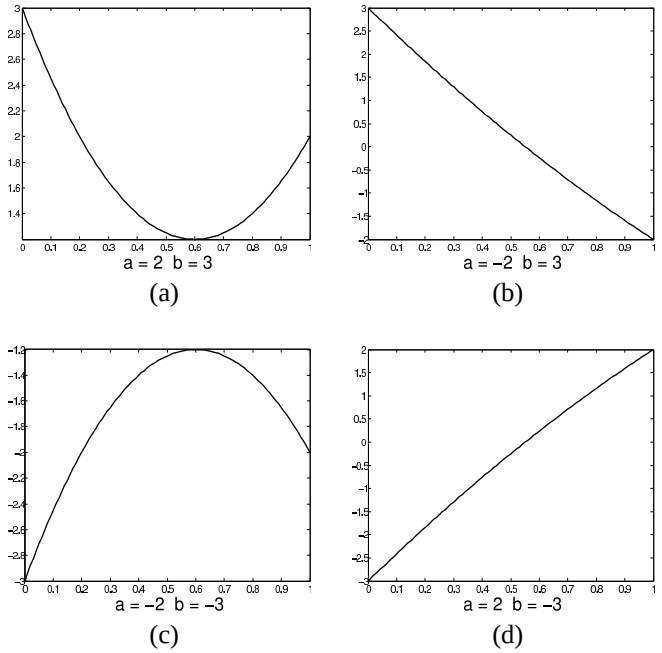


Fig. 1. The parabola defined by equation (5) when $a > 0$ and $b > 0$ (a), $a < 0$ and $b > 0$ (b), $a < 0$ and $b < 0$ (c), and $a > 0$ and $b < 0$ (d).

The results of this analysis completely agree with classical results of the evolutionary theory of 2×2 symmetric games (see [2] and [4]) obtained using the concepts of Nash equilibria and evolutionary stable strategies. Moreover, the model implies that the spatial structure of the population does not influence the mean behaviors of the evolved populations, as we will see in the next section.

IV. Spatial Artificial Evolution

The framework described in the previous section is independent of the spatial structure of the evolved population. To test whether its predictions are good when spatial constraints are introduced, we have decided to evolve populations on two-dimensional regular lattices: each agent is placed on a vertex of a rectangular grid with periodic boundary conditions (i.e., a toroidal structure), and is connected with the eight closest agents, thus defining a Moore neighborhood. While other neighborhoods are possible, the results remain qualitatively the same. Furthermore, one of the authors is running an experiment on human subjets; among the results, it is evident that individuals choose a partner in their physical Moore neighborhood. In further research we will consider the dynamical evolution on different networks such as small-world networks, fragmented networks and random networks.

The evolution is performed synchronously: at each generation, each agent selects the fittest agent in his neighborhood and produces an offspring whose associated probability is obtained by intermediate crossover (also known as arithmetical or guaranteed average crossover [3]) between the two probabilities associated with the agent itself and the selected agent. No mutation is used in this process, and the produced offspring replaces the considered agent in his location in the structure. The dynamical system is completely deterministic: different attractors can be found for each system, but, as we will show in the simulations in Section V, the mean agent will always tend to 0, 1 or the value of the game (the vertex of the parabola) depending on the cases described in the previous section.

Even if the algorithm is quite simple, we have noticed that using two different implementations in Matlab and C++ we obtained results qualitatively comparable but numerically different. This is probably due to the different internal representations of real numbers: for more details see the implementation note in the Appendix. While different approaches have been suggested for handling errors in floating point representations (e.g., interval arithmetic, see [5]), given the finite state structure of our model, we decided to use integer representation for state s. This approach is similar, in a certain sense, to that used by Harrald [1].

To each agent a_i is thus associated an integer state $s_i \in \{0, 1, \ldots, M\}$: the agent will then play action X with probability $p_i = s_i/M$. To calculate the gain (the fitness) of agent a_i, this probability is used in equation (4): if we denote with $W(t)$ the sum of the states of the agents of the population at time t ($W(t) = \sum_{i=1}^{N} s_i$), we have $U(t) = W(t)/N$. Multiplying by M^2 and simplifying, we obtain the following form of equation (4):

$$\begin{aligned} F(i,t) &= -(a+b)s_i^2(t) + \\ &\quad + ((a+b)W(t) + Mb(2-N))s_i(t) + \\ &\quad + Mb(M(N-1) - W(t)). \end{aligned} \tag{6}$$

This evolutionary system can be described in a more formal way: let's consider a discrete time t and a population $P(t)$ of N agents. To each agent are associated a state and a location: $P(t) = \{a_1(t), a_2(t), \ldots, a_N(t)\}$, with $a_i = \langle s_i(t), l_i \rangle$, where $s_i(t) \in S = \{s_1, s_2, \ldots, s_m\}$, the set of the possible states of the agents, and $l_i \in L = \{l_1, l_2, \ldots, l_N\}$, the set of the locations of the agents in the structure of the population. If we denote with T the product space between the space of the possible states and the space of the possible locations of the agents ($T = S \times L$), a population of N agents is an element of T^N.

A fitness function $F : T^N \rightarrow R^N$ (where R is the set of real numbers) is given, such that each population $P(t) = \{a_1(t), a_2(t), \ldots, a_N(t)\}$ is associated to a vector $f(t) = F(P(t)) = \langle f_1(t), f_2(t), \ldots, f_N(t) \rangle$ with $f_i(t)$ being the fitness value of agent $a_i(t)$.

The selection mechanism is described by a function $Sel : T^N \times R^N \rightarrow S^N$ such that $\langle s'_1(t), s'_2(t), \ldots, s'_N(t) \rangle = Sel(P(t), F(P(t)))$. For each agent in the population it selects the state of the agent in his neighborhood with the highest fitness value. Note that only the state of an agent is selected, since the location of the selected agent doesn't influence the successive crossover. On the contrary the location of the selecting agent influences the function Sel, since it determines the selection pool for each location in the structure. The topology of the structure thus affects the selection function, but not the successive reproduction operators.

The state of the agent in the considered location is then combined with the selected state by a function $Op : S \times S \rightarrow S$, producing the state of the agent in the next generation for the considered location. If only a crossover operator is used, as it is the case in our evolutionary algorithm, the function Op can be represented in the form of an $N \times N$ matrix of elements of S.

Given the topology of the structure, the set L of the possible locations of the agents, the set S of the possible states of the agents, the fitness function F, the selection function Sel, the recombination function Op, and the population $P(t)$, the population $P(t+1) = \{a_1(t+1), a_2(t+1), \ldots, a_N(t+1)\}$ at the next generation is formed by agents $a_i(t+1) = \langle s_i(t+1), l_i \rangle$ such that $s_i(t+1) = Op(s_i(t), s'_i(t))$.

V. Simulations Analysis

Two groups of simulation have been performed to test the exactness of the models' predictions: the first time, we let the system evolve starting from random populations. Then we created a particular initial population and the system dynamical behavior is observed, so as to show how the prediction of the model actually works.

At first, we let evolve a population of 2500 agents at 21 possible states disposed on a 50×50 toroidal grid with a Moore neighborhood (each agent's neighborhood is composed by the agent itself and the 8 agents directly surrounding him). The agents face a game whose matrix is the one depicted in Table III).

	X	Y
X	-2	0
Y	0	-3

TABLE III

THE PAYOFF MATRIX FOR THE SIMULATIONS.

Such a matrix falls under case 3) of Section II since both a and b are negative. The model in this case predicts that, whatever the initial population is, the mean agent will tend to the value of the game, which in this case is 0.6. This prediction is confirmed by the simulation results: in Figure 2 the evolution of the mean player over a generation is shown, when starting with a random population composed by 20% of agents playing action X with probability 1, and 80% playing action Y with probability 1.

The evolution over the generations of the fitness of the mean agent is shown in figure 3(a): it can be noticed how, even though the mean agent value oscillates between two different states, its fitness value (its payoff against all other members of the population) stabilizes. The fitness of the mean agent is clearly linked to the mean fitness of the population: the artificial evolution tends to populations of different agents who have very similar fitness values. In fact the difference between the maximal and the minimal fitnesses of the population through generations tends to 0, as it is shown in figure 3(b).

As we have previously pointed out, the model can predict the behavior of the mean agent, without taking into account

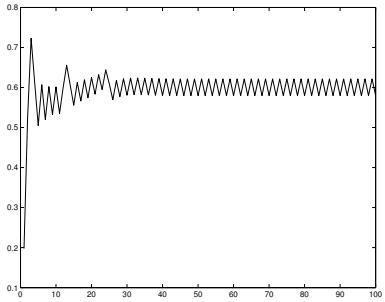
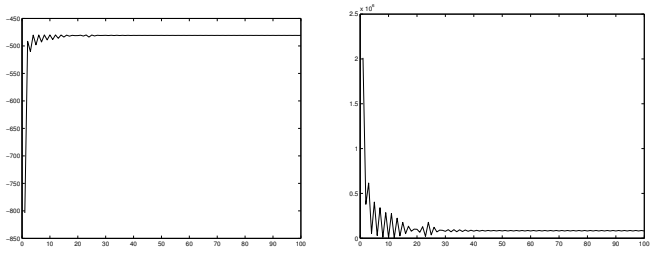


Fig. 2. Evolution of the value of the mean player over time of a population of 2500 agents at 21 possible states disposed on a 50×50 toroidal grid with a Moore neighborhood.



(a) (b)

Fig. 3. Evolution over the generations of the fitness of the mean agent (a) and of the difference between the maximal and the minimal fitness of a population of 2500 agents at 21 possible states disposed on a 50×50 toroidal grid with a Moore neighborhood.

the structure of the population and the initial disposition of the agents. In fact, starting with different populations, we will observe different attractors for the evolutionary process. For this simulation a period 2 attractor can be observed (see figure 4, where the two populations are shown): darker agents correspond to probabilities closer to 0 of playing action X, with black agent corresponding to probability 0, and white agents to probability 1.

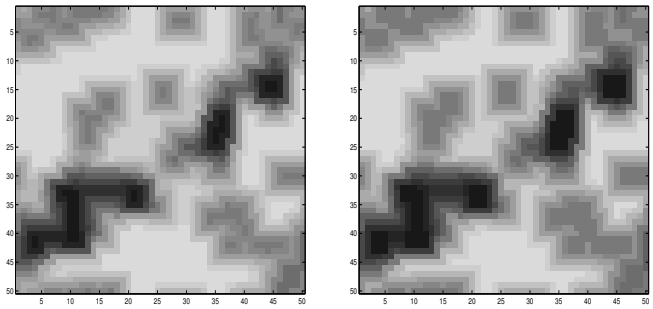


Fig. 4. Final populations of 2500 agents at 21 possible states disposed on a 50×50 toroidal grid with a Moore neighborhood. The two populations form a period 2 cyclic attractor of the evolutionary system.

If the payoff matrix is changed to the one depicted in Table IV, the game falls under case 1) of Section II since both a and b are positive.

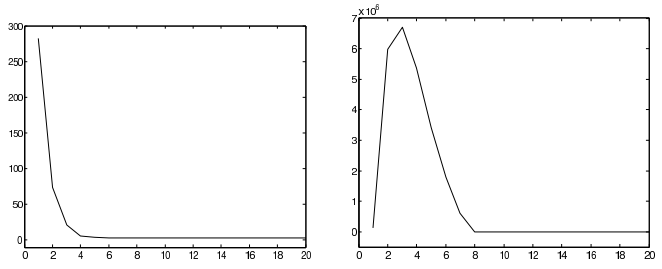
The model predicts that the evolution will depend on the mean agent $\bar{p}(0)$ of the initial population $P(0)$; if $\bar{p}(0) < 0.6$ (the value of the game), then the evolution will be driven to-

	X	Y
X	2	0
Y	0	3

TABLE IV

THE PAYOFF MATRIX FOR THE SIMULATIONS.

wards action Y; otherwise the evolution will be driven towards action X. The prediction is fully confirmed by the simulations shown in Figures 5 and 6, where the time evolution of the mean agent and of the difference between the maximal and the minimal population fitnesses are shown, in the case of initial random populations with $\bar{p}(0) = 0.5939$ and $\bar{p}(0) = 0.6047$ respectively. Note how the difference between the maximal and the minimal fitnesses in the population rapidly grows at the beginning of the evolution (the agents split towards opposite strategies), and then tends to 0.



(a) (b)

Fig. 5. Evolution over the generations of the mean agent (a) and of the difference between the maximal and the minimal fitnesses of a population of 2500 agents at 21 possible states disposed on a 50×50 toroidal grid with a Moore neighborhood. The initial population has a mean agent with associated probability $\bar{p}(0) = 0.5939 < 0.6$, the value of the game.

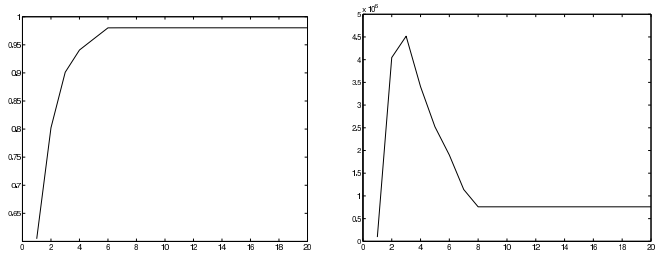


Fig. 6. Evolution over the generations of the mean agent (a) and of the difference between the maximal and the minimal fitnesses of a population of 2500 agents at 21 possible states disposed on a 50×50 toroidal grid with a Moore neighborhood. The initial population has a mean agent with associated probability $\bar{p}(0) = 0.6047 > 0.6$, the value of the game.

In the second part of the simulations we consider a small population of 121 agents disposed on a 11×11 toroidal grid. Each agent's neighborhood is composed by the agent itself and the 8 agents directly surrounding him (thus forming a Moore neighborhood). The payoff matrix of the game is the same as for the first group of simulations (see Table III).

The set of an agent's possible states is composed of 7 elements ($S = \{0, 1, \ldots, 6\}$). The probabilities of playing action X associated to the 7 states are respectively: 0, 0.1667, 0.3333, 0.5, 0.6667, 0.8333, and 1. To draw the populations during the evolution, we have associated to each state a color on a grey scale (see Figure 7).

Fig. 7. Color scale for 7 state agents: from state 0 (black) we pass through states corresponding to probabilities 0.1667, 0.3333, 0.5, 0.6667, 0.8333, to finally reach state 6 (white) that corresponds to probability 1 of playing action X.

The intermediate crossover between the integer states is performed according to the crossover matrix of Table V: recombining a state i agent with a state j agent, the state of the offspring agent will be the one at the intersection of row i and column j of the matrix.

	0	1	2	3	4	5	6
0	0	0	1	1	2	2	3
1	0	1	1	2	2	3	3
2	1	1	2	2	3	3	4
3	1	2	2	3	3	4	4
4	2	2	3	3	4	4	5
5	2	3	3	4	4	5	5
6	3	3	4	4	5	5	6

TABLE V

THE CROSSOVER MATRIX FOR AGENTS WITH 7 POSSIBLE STATES.

Figure 8 shows the evolution of the system starting from an initial population solely of all agents playing action X with probability 1 (agents' states 6), with the exception of the central individual who plays action Y with probability 1 (agent state 0). For each generation ($t = 0, 1, \ldots, 7$) the population is plotted on the left, and the parabola associated to the population is drawn on the right: the probabilities associated with the 7 possible states of the agents are on the x axis, and the corresponding fitness values, function of the sum $U(t)$ of the probabilities associated to the agents in the population, are on the y axis.

At time $t = 0$ the single agent at state 0 has the highest fitness value (the parabola is decreasing in the interval $[0, 1]$), and therefore it will be selected by its surrounding neighbors for recombination. Applying the crossover matrix (see Table V), at the next generation ($t = 1$) the population will be formed by one agent at state 0 surrounded by 8 agents at state 3, and all other agents at state 6. The behavior of the population from time 1 to time 4 is analogous to that of time 0: since smaller associated probabilities have higher fitness values, the agents will recombine with the agent in their neighborhood with smaller states. Note that the agents at state 6 disappear, because they have always the smallest fitness value. Since the crossover matrix allows the production of state 6 agents only when both the parents have state 6, that state will never appear once lost in the population.

At time $t = 5$ the parabola becomes increasing in the interval $[0, 1]$. The agents at state 0 have the lowest fitness

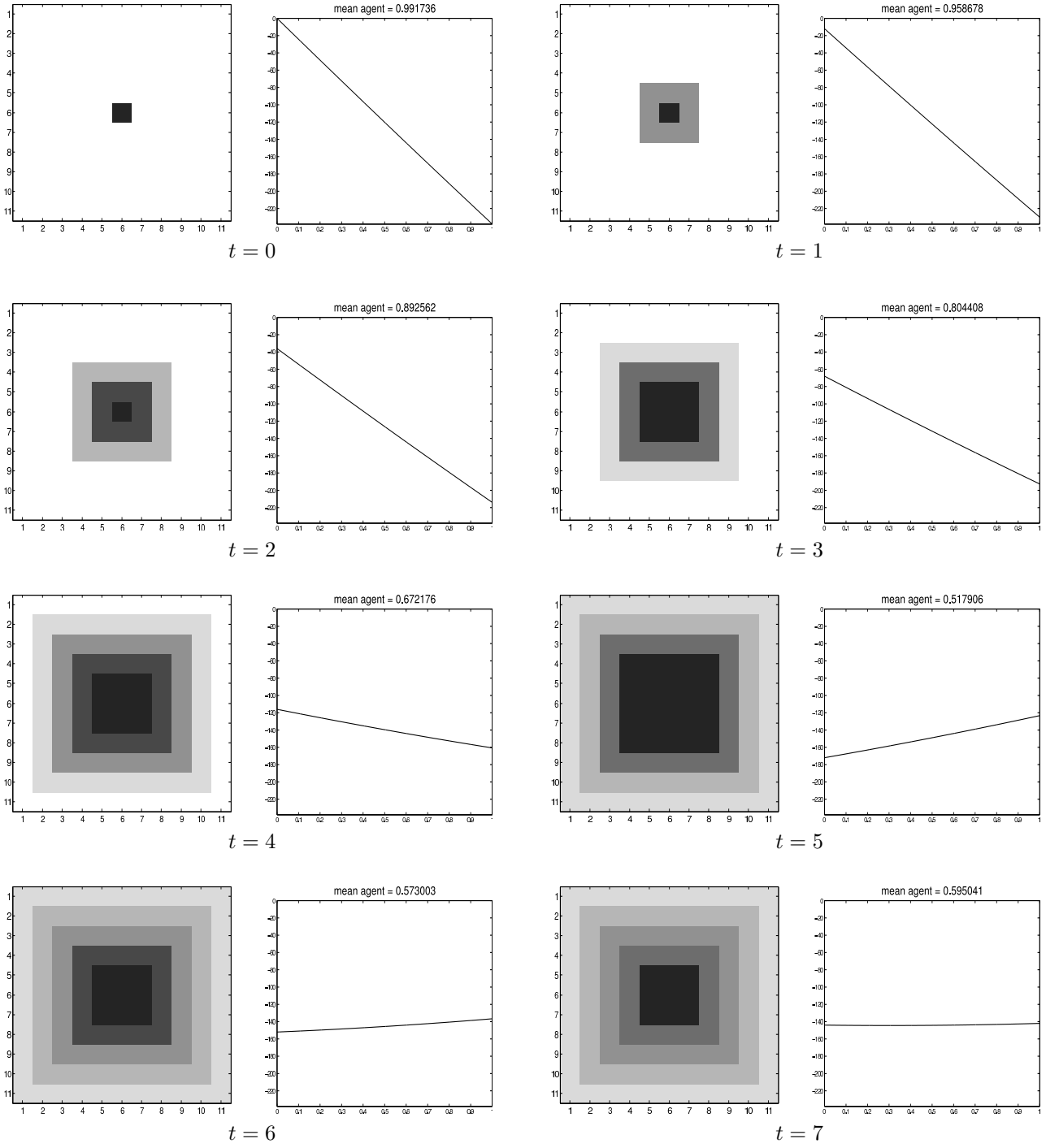


Fig. 8. Evolution of a population of 11×11 agents that can assume 7 possible states: the initial population is composed by all agents playing action X with probability 1 (agents' states 6), except the central individual that plays action Y with probability 1 (agent state 0). For each time step, the population (left) and the corresponding parabola (right) are shown. At time $t = 8$ the population will be the same as at time $t = 6$, resulting in an attractor of period 2 for the dynamic of the system.

value: those at the border of the region will select state-2 agents for recombination (since they have higher fitness), producing state-1 offspring agents. Agents at state 2 will select, for the same reason, agents at state 4, producing state-3 offspring agents. All other agents don't change state, since the crossover operator will produce offsprings with the same state (see Table V).

At time $t = 6$ (see the enlargement in Figure 9 left) the parabola is still increasing in the interval $[0, 1]$: only agents at state 1 will change state, since the only crossover producing

offsprings with a different state is the one between agents at state 1 and agents at state 3. At time $t = 7$ (see the enlargement in figure 9 right), parabola is such that state-0 agents have a higher fitness than state-1, -2, and -3 agents, and therefore only agents at state 2 selecting agents at state 0 will produce an offspring at a different state (1). A population equal to that of generation 6 is produced, and the system enters in a period-2 attractor oscillating between the two configurations. The system oscillates between states in which the mean agent has strategies 0.573003 and 0.595041. This result completely agrees with the model's prediction: since both a and b are negative, the mean agent shall tend to the value of the game, which in this case is 0.6.

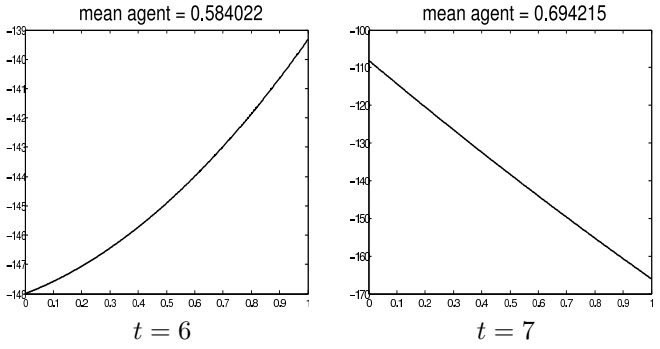


Fig. 9. Enlargement, enlarging y-axis, of the parabola of figure 8 at generations 6 and 7: from increasing in the probability interval $[0, 1]$ at generation 6, it becomes decreasing at generation 7.

VI. Conclusions and Future Work

We have introduced a new framework to analyze and predict the behavior of evolutionary 2×2 symmetric games. This approach only uses the parameters of the payoff matrix of the game, and leads to behavior predictions that are in perfect agreement with classical evolutionary theory, without dealing with Nash equilibria or evolutionary stables strategies. The proposed model is not influenced by the spatial structure of the evolving population of agents.

The evolutionary algorithm used to experimentally validate the model is then described, introducing a new formalism for the evolution of spatially structured populations. The experiments fully confirm the predicted behaviors, and a complete analysis of a simple dynamical system is presented, in order to exemplify the model.

In the future we intend to investigate the different dynamics induced by different crossover matrices, and the possibility of the co-evolution of strategies and crossover operators. We also want to study and model the introduction of random mutation in the evolutionary algorithm.

Acknowledgments

Mario Giacobini gratefully acknowledge financial support by the Fonds National Suisse pour la Recherche Scientifique under contract 200021-103732/1.

Appendix

The consequences of floating points arithmetic error are well known in the simulation literature (see for instance [6]). In order to avoid this common pitfall we decided to implement our framework using both Matlab and C++. With continuous states the results obtained by two implementations were qualitatively the same even if numerically different.

Since exact replication of the experiments is obviously desirable we decided to have quantized states in order to obtain crossover results that were consistent between the two implementations.

While usually such effects are thought to be arising from accumulated floating point errors, in our case we found such discrepancies almost immediately.

In fact even with the seven-state simulation described in Section V our Matlab implementation incurred in some problems, when performing the arithmetical crossover.

For example, consider crossover between two agents with probabilities $1/2$ (state 3) and 0 (state 0). With the arithmetic crossover the offspring is $1/4$ and, in deciding the state of the offspring, two quantities must be compared: $1/4 - 1/6$ and $2/6 - 1/4$. Note that obviously in this case the two quantities are identical and a rule should be implemented for deciding the state of the offspring. The problem we encountered is that Matlab considers $1/4 - 1/6$ greater than $2/6 - 1/4$. With the C++ implementation we used long double i.e., floating-point data type with 80 bits of precision for our variables and the problem did not occur. Yet, since the accumulated floating point errors could not be ruled out completely, we decided to consider integer representation for the states.

Nevertheless a further step was in order. Since, due to the internal representation, the same fitness could be considered different depending on the implementation, we resolved to consider integer values for the fitness as well. This, to the best of our knowledge, solved our problems with the only drawback of imposing an upper bound on the number of states to be considered.

References

[1] Paul Harrald, "Evolving behaviour in repeated 2-player games," in *Practical Handbook of Genetic algorithms: Applications, Volume I*, Lance Chambers, Ed., pp. 459–496. CRC Press, Boca Raton, FL, 1995.
[2] J. Weibull, *Evolutionary Game Theory*, MIT Press, 1997.
[3] Z. Michalewicz, *Genetic Algorithms + Data Structures = Evolution Programs*, Springer-Verlag, Heidelberg, third edition, 1996.
[4] J. Maynard Smith, *Evolution and the Theory of Games*, Cambridge University Press, 1982.
[5] G. Alefeld and J. Herzberger, *Introduction to interval computation*, Academic Press, New York, New Jersey, 1983.
[6] J.G. Polhill, L.R. Izquierdo, and N.M. Gotts, "The ghost in the model (and other effects of floating point arithmetic)," *Journal of Artificial Societies and Social Simulation*, 2004, (to appear).

Synchronous and Asynchronous Network Evolution in a Population of Stubborn Prisoners

Leslie Luthi, Mario Giacobini, Marco Tomassini
Information Systems Department
University of Lausanne, Switzerland
{leslie.luthi,mario.giacobini,marco.tomassini}@unil.ch

***Abstract*— We study by computer simulation a population of individuals playing the prisoner's dilemma game. Each player has an invariable strategy (cooperate or defect) but the network of relationships between players is allowed to change over time following simple rules based on players' degree of satisfaction. The population almost always reaches a stable state and we observe that, in the long run, cooperators tend to cluster together in order to maintain a high average payoff and to protect themselves from exploiting defectors. Thus network topology plays an important role even though strategies are not allowed to evolve. We investigated both synchronous and asynchronous network dynamics, observing that asynchronous update, in addition of being more reasonable in a social setting, induces system stability more often than the synchronous one.**

I. Introduction and Previous Work

Game theory is an attempt to model and analyze conflicting situations such as those that arise in the economy, in biology, and in society in general [1]. In this context, the well-known game called *Prisoner's Dilemma* (PD) has played an extremely important role and has received a lot of attention, including public computer tournaments (for a survey, see Axelrod's book [2]). The prisoner's dilemma has fascinated researchers because it is an interaction where the individual rational pursuit of self-interest, one of the pillars of game theory, produces a collective result that is self-defeating. The following payoff matrix represents the prisoner's dilemma game as a two-person game in normal form:

	C	D
C	(R,R)	(S,T)
D	(T,S)	(P,P)

In this matrix, C stands for cooperation and D for defection. R stands for the *reward* the two players receive if they both cooperate, P is the *punishment* for bilateral defection, and T is the *temptation*, i.e. the payoff that a player receives if she defects, while the other cooperates. In this latter case, the cooperator gets the *sucker's* payoff S. These names are traditional, but there is nothing special about them, of course. The payoff values are ordered numerically in the following way: $T > R > P > S$ and $R > (T + S)/2$.

This game has a unique Nash equilibrium, (P,P), in which both players defect, i.e. they both choose the strategy D. Thus, in a one-shot play of the game the rational outcome is for both players to play D. However, both players would be better off cooperating, with a payoff (R,R). But the outcome (R,R), although it is preferable, is not a Nash equilibrium, and thus two rational players will always play D instead.

If the PD game is iterated a known finite number of times, the result doesn't change, and steady defection of the two players is the rational outcome of each encounter in the sequence. However, when the game is iterated an *indefinite* number of times, strategies that allow cooperation to emerge and persist are possible, as described by Axelrod [2]. This is an interesting result since it could lend some justification to the commonly observed fact that cooperation does appear in society, in spite of individual greed. There is a large amount of literature on the iterated PD, see for instance [2] and [3]. We will only deal with the one-shot case in the rest of the paper.

Considering now not just two players but rather a population of N players, *evolutionary game theory* [4], [5] prescribes that defection is the evolutionarily stable strategy of the population, given memoryless players. However, in 1992, Nowak and May [6] showed evidence, by using computer simulations, that cooperation in the population is sustainable under certain conditions even in the one-shot game, provided that the population of players has a *spatial structure*. A spatial structure means that the population members are disposed on some kind of lattice, and each player has a local small neighborhood of other players with which he plays a number of two-person PD games in succession. Nowak and May originally used a square grid with five or nine neighbors per cell, but the results are robust with respect to changes of the neighborhood shape and size, if it remains small and localized [7].

The fact that cooperation may be stable in a structured context even when players do not have memory of past encounters is a remarkable result. However, many real conflicting situations in society are not well described by a fixed geographical disposition of the players. In economy, for instance, markets and relations between firms are not limited by geographical distance in this era of fast global communication. The same can be said of many social and political interactions where relationships may change over time. Thus, it becomes of interest to study how the topology of the interactions influences the global outcome, and how this same relational structure network may evolve under the pressure of the player's strategic interactions. The PD is an excellent way of studying such an evolution in a simplified

and understandable environment.

Recently, Zimmermann *et al.* [8] have published a study in which both the strategies of players – C or D – and the network of players' relationships may change and adapt during time. More specifically, they use the same rule as Nowak and May to play successive two-person PD games between a given player and all the players that are linked to it in the network. Each player then adopts the strategy – C or D – of the neighbor with the best accumulated payoff. Players are *satisfied* if their payoff is the highest among the neighbors, otherwise they are *unsatisfied*. Only Unsatisfied D-players can then break a link to another D-player with a certain probability and rewire it randomly. The evolution is synchronous, i.e. all the players update their strategies and possibly the neighborhood simultaneously. Using computer simulations, Zimmermann *et al.* find that, for some value of the parameters, the network of players self-organizes to stable cooperative states. Moreover, by weighting the rewiring with a probability that favors D players that are closer in the network – in the sense of the path length between them – they also show that the resulting steady-state networks are of the small-world type [9].

Here we follow similar ideas but we assume a population of players each of which has a fixed unchanging strategy C or D. Thus, our goal cannot be the study of the evolution of the proportion of C and D strategies in the population, since this proportion is fixed. Instead, we concentrate on the purely topological aspects i.e., the evolution of the network of relationships among the players. With respect to [8], our network update rules are different, as explained in the following section. As well, we study both synchronous and asynchronous update policies and compare the results.

Although the assumption of unchanging strategy may seem unrealistic, this is not necessarily so. Indeed, there are many situations in which the player has little or no choice of alternative strategies, either because of insufficient knowledge or because of external social pressure. Think for instance of religious or social beliefs and behaviors in homogeneous human societies, or of market agents that do not know how to adapt their strategies to market changes and stick to fixed strategies for long periods of time. Another example would be always voting for a given political party irrespective of other considerations, a commonly observed situation. Thus, fixing the strategies and allowing the relational network to evolve is a useful first step toward an understanding of the interplay of topology and dynamics in social interactions as represented by the PD.

II. The Model

The game considered consists of a population of N individuals all playing the prisoner's dilemma. The population can be subdivided into the subset of the cooperators, denoted E_C, and the one comprising the defectors, denoted E_D. Initially, there are no links between the N players in the population. At a given time t, an individual i interacts exclusively with a subset of the entire population known as its *neighbors* or *neighborhood* and denoted by $V_i(t)$ with the condition $i \notin V_i(t)\ \forall t$. An individual i does not necessarily have neighbors in which case $V_i = \emptyset$. An interaction of an individual i with one of its neighbors j is represented by an undirected link so that $i \in V_j(t) \Rightarrow j \in V_i(t)\ \forall t$. The different values of $S = 0$, $P = 0.1$, $R = 1$, $T = 1.39$ have been chosen to be in accordance with the $T > R > P > S$ and $R > (T + S)/2$ relationships of the PD mentioned in the introduction. Furthermore, $T = 1.39$ was chosen due to the interesting results obtained in previous works as to the persistance of C and D together for values of T in between 1.2 and 1.6 [7]. This leads us to the following payoff matrix for a 2-agents one shot PD game:

	C	D
C	(1,1)	(0,1.39)
D	(1.39,0)	(0.1,0.1)

TABLE I

PAYOFF MATRIX FOR THE TWO-PERSON PD GAME USED IN THE SIMULATIONS.

A. Normalized Average Payoff and Notion of Satisfaction

Let s_i be the strategy of an individual i, with $s_i = 0$ for a defector (D) and $s_i = 1$ for a cooperator (C). Furthermore, let us denote $\Pi_i(t)$ the normalized average payoff of the individual i at a given time step t. This gives us the following equation:

$$\Pi_i(t) = \begin{cases} \frac{-1}{s_i R + (1-s_i)T} & \text{if } V_i = \emptyset, \\ \frac{s_i(\gamma_i(t)R + \delta_i(t)S) + (1-s_i)(\gamma_i(t)T + \delta_i(t)P)}{|V_i(t)|\ (s_i R + (1-s_i)T)} & \text{otherwise.} \end{cases} \quad (1)$$

where $\gamma_i(t)$ (resp. $\delta_i(t)$) is the number of cooperators (resp. defectors) $\in V_i$ at the given time step t. $\Pi_i(t)$ has a negative value when player i is isolated to distinguish this case, where the player has to randomly choose another player in the population to be its neighbor, from the case where the player has a 0 payoff and must thus rewire one of its links. Once $\Pi_i(t)$ is defined, we can introduce the *satisfaction threshold* Θ_i of an individual i as:

$$\Theta_i = \frac{s_i(S + \sigma(R - S)) + (1 - s_i)(P + \sigma(T - P))}{s_i R + (1 - s_i)T} \quad (2)$$

where $0 \leq \sigma \leq 1$ is called the *satisfaction degree*.
An individual i is said to be *satisfied* iff $\Pi_i \geq \Theta_i$. In all other cases, i is said to be *unsatisfied*. The *satisfaction degree* characterizes somewhat the minimum percentage of cooperators an individual should have among its neighbors in order to be satisfied. Notice however that σ does not indicate precisely this minimum percentage since D-neighbors equally increase a player's payoff, even though this contribution is generally insignificant. The two limit cases are $\sigma = 0$, which means that an individual i will always be satisfied except if it has no neighbors ($V_i = \emptyset$), and $\sigma = 1$ implying that i will be satisfied iff its neighborhood is composed solely of cooperators

(i.e. $\Pi_i = \Theta_i = 1$). It is to be noted that the notion of a satisfied individual voluntarily differs from the one defined by Zimmermann *et al.* [8] where a player is satisfied only if it has the highest payoff among its neighbors. In our opinion, in a real-world situation, one doesn't necessarily desire to be "the best". Take for example an employee working for a company. She need not earn the highest salary among her colleagues to be pleased with her pay. As a matter of fact, it is likely that the employee even ignores the salary of his or her fellow workers. This is comparable to our notion of satisfaction where an individual is not obliged to know the neighbors' payoff to be contented. Furthermore, we find it important to work with an average payoff as opposed to an accumulated one considered by Zimmermann *et al.* [8]. Indeed, in the case of an accumulated payoff, there could be illogical situations, due to the fact that this type of payoff privileges quantity over quality, meaning that an individual with many neighbors has a good chance of having a better payoff than an agent with just a few "good" ones. A good example of such a situation is a C-player surrounded by two dozen defectors having a better payoff than one who has exactly one or two C-neighbors and no D-neighbors.

B. The Rules

The interactions between the participants of the game evolve in time according to the following basic rules:

Let i be a player and t_n the n_{th} time step.

- *if* $V_i(t_n) = \emptyset$:
  i is unsatisfied and must choose an individual uniformly at random j in the population to be i's new neighbor, i.e. $j \in V_i(t_{n+1})$.
- *else if* $\Pi_i(t_n) < \Theta_i$:
  i is unsatified and must hence pick randomly one of its D-neighbors and replace it with a randomly chosen individual j satisfying $j \notin V_i(t_n)$.
  If such a j does not exist, nothing is done and i must try to bear with its unsatisfaction.
  Notice that an unsatisfied individual i with $V_i \neq \emptyset$ necessarily has a D-neighbor since C-neighbors only contribute to i's satisfaction.
- *otherwise*:
  i is satisfied of its situation and will thus continue to play against exactly the same individuals at time step t_{n+1} unless, independent of i, one of its neighbors decides to cut off its link with i or an outsider inserts itself into i's neighborhood.

Finally, let us stress the fact that unlike the model in [8], defectors, *as well as cooperators*, have the possibility to attempt a change in their neighborhood.

C. The Updates

The changes made to a population between two succesive time steps t_n and t_{n+1} will be done in two different ways:

1) Synchronous: In the *synchronous* update, every individual i starts by playing the PD game with its neighbors and uses the outcome to calculate its normalized average payoff $\Pi_i(t_n)$. Next, all the unsatisfied players simultaneously modify their neighborhood according to the previously mentioned rules (II-B). Possible collisions are resolved by using a temporary network data structure. The synchronization lies in the fact that each player considers the necessary changes to make in its neighborhood as it sees it at time step t_n unaware of what the other players are planning to do.

2) Asynchronous: For the *asynchronous* case we use *independent random ordering* [10] of updates in time. This is a close approximation of a Poisson process. The time t needed to update the whole population is subdivided into a sequence $(u_1, u_2, \ldots, u_N)$ of *update steps*. During an update step u_k individual i is randomly picked, $\Pi_i(u_k)$ is calculated, and the appropriate rules (II-B) are used to immediately adapt its neighborhood if unsatisfied. Note that in the rules II-B, $V_i(t_{n+1})$ becomes $V_i(u_{k+1}), \quad k = 1, 2, \ldots, N$. This process is iterated N times (where N is the population size) with replacement. When the N update steps are over, only then do we change time step to t_{n+1}. Note that this is only one of the many possible sequential update policies [11], [10], but it is a reasonable one in our case.

Despite the fact that most work in the field has been done using synchronous simulation, there are good reasons for introducing an asynchronous update policy in spatial games. Indeed, the hypothesis of a global clock, which is a necessary requirement for synchronous dynamics, could be unrealistic in a social setting, since information travels at finite speed. Thus, having the whole population update its state all at once is only an idealization. Hubermann and Glance [12] elaborate on this point, arguing that asynchronous update policies correspond better to the dynamics of social interactions. A different point of view is discussed in [7] where the authors state that there is not much difference at the macroscopic, population level. Here we have decided to study both update models, in order to gain further insight on the corresponding dynamical processes.

III. Simulations Analysis

All the experiments in the following section refer to two subsets E_C and E_D of the population of equal size. The case of unequal proportions of cooperators and defectors is discussed in section III-B.

A. General Results

1) $0.0 \leq \sigma \leq 0.9$*:* In order to analyze the influence of the satisfaction degree on the network of players and compare the results between the synchronous and asynchronous updates, we varied σ from 0.0 to 0.9 by steps of 0.1. For each of these values, two series of 20 runs of 400 time steps each were executed on a population of 1600 players, one series per update policy.

Among the several quantities that were observed at the end of each run, we studied in particular the *mean normalized*

average payoff $\overline{\Pi}_C(t)$ and $\overline{\Pi}_D(t)$ of cooperators and defectors respectively, defined as:

$$\overline{\Pi}_C(t) = \frac{\sum_{i \in E_C} \Pi_i(t)}{|E_C|} \quad (3a)$$

$$\overline{\Pi}_D(t) = \frac{\sum_{i \in E_D} \Pi_i(t)}{|E_D|} \quad (3b)$$

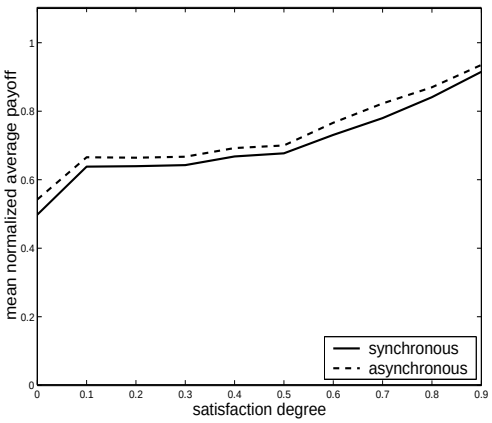


Fig. 1. The mean normalized average payoff $\overline{\Pi}_C$ after the final time step, averaged over 20 runs. Synchronous versus asynchronous updating.

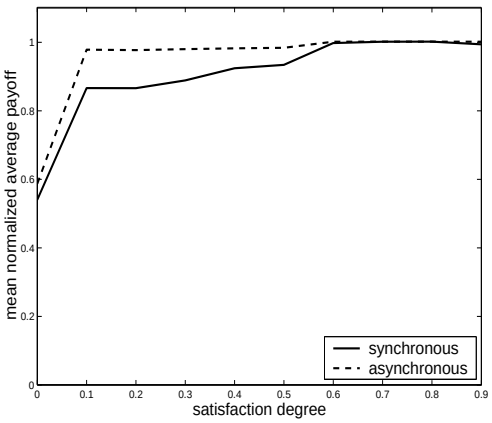


Fig. 2. The mean normalized average payoff $\overline{\Pi}_D$ after the final time step, averaged over 20 runs. Synchronous versus asynchronous updating.

As the satisfaction degree increases, the mean average payoff of the cooperators seems to clearly tend to the C maximum payoff R (Fig. 1). Moreover, both synchronous and asynchronous lead to the same behavior. Unlike $\overline{\Pi}_C$, $\overline{\Pi}_D$ differs a little from synchronous to asynchronous. Fig. 2 shows that when using an asynchronous update, the defectors globally attain a payoff close to the their maximum possible ($\overline{\Pi}_D > 0.9$) for already very small values of σ ($\sigma = 0.1$) whereas in the synchronous case, $\overline{\Pi}_D$ greater than 0.9 are reached only for $\sigma > 0.5$. Nevertheless, if we ignore transients, the long-term trend is identical for the two types of updates. The correlation between the increase of σ and that of $\overline{\Pi}_C$ and $\overline{\Pi}_D$ is explained easily by the fact that the higher the satisfaction degree, the more demanding the players become of their neighbors. For example, $\sigma = 0.8$ implies that in order for an agent to be satisfied, its neighborhood must be composed of about 80% of cooperators.

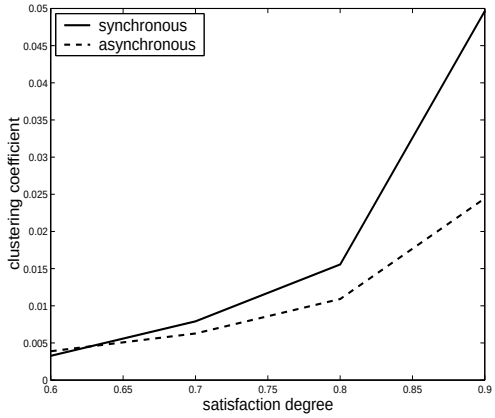


Fig. 3. Cooperator clustering coefficient $\mathcal{C}$ at system stability, averaged over 20 runs. Synchronous vs asynchronous updating. Take notice that the x-axis starts at $\sigma = 0.6$. For $\sigma \leq 0.5$, the main connected subset of cooperators has an insufficient size to make statistically significant measures of $\mathcal{C}_n$.

To highlight the phenomenon of σ pressuring the cooperators to stick together, one can also study the *Clustering Coefficient* ($\mathcal{C}$) of the connected subgraph of cooperators where the *Clustering Coefficient* of a graph is defined as follows [9]:

Consider a particular node n in a graph G, and let us assume that it has k edges connecting it to its k neighboring nodes. If all k vertices in the neighborhood were completely connected, then the number of edges would be equal to $k(k-1)/2$. The clustering coefficient $\mathcal{C}_n$ is defined as the ratio between the e edges that actually exist between the k neighbors and the number of possible edges between these nodes, which give us:

$$\mathcal{C}_n = \frac{2e}{k(k-1)}$$

The clustering coefficient of the whole network with N vertices is the average of all $\mathcal{C}_n$: $\mathcal{C} = \frac{1}{N}\sum_{n \in G} \mathcal{C}_n$.

The $\mathcal{C}$ of a random graph is simply $\langle k \rangle / N$, where $\langle k \rangle$ is the average degree of the graph and N its total number of vertices. The $\mathcal{C}$ of a complete graph, is clearly equal to 1.

The increasing clustering coefficient of the main connected subset of cooperators (Fig. 3) indicates, by definition, that more and more C-neighbors of a given C-player are neighbors of each other. The difference between the two curves of Fig. 3 is caused by a higher averge degree of the overall network (particularly of the main connected C-subcomponent) in the synchronous case, which is in turn due to a "collision" problem not present when using an asynchronous update. This problem is explained later in the section.

2) $\sigma = 1.0$: Do the tendencies mentioned under III-A.1 hold true for the limit case of $\sigma = 1$ where all the players, whether they are cooperators or defectors, are unsatisified

as long as a defector is found among its neighbors? To answer this question, and predicting a large number of links as well as abundant rewiring requiring a lot of time, the runs were increased to 1000 time steps each and the size of the population was reduced to 40 individuals (20 C and 20 D). This reduction was essentially done for obvious computational reasons, but also enables us to have a visual representation of the networks and the mechanisms at hand (Fig. 4).

Let us take a look at the time series of the mean normalized average payoffs $\overline{\Pi}_C$ and $\overline{\Pi}_D$ (Fig. 5).

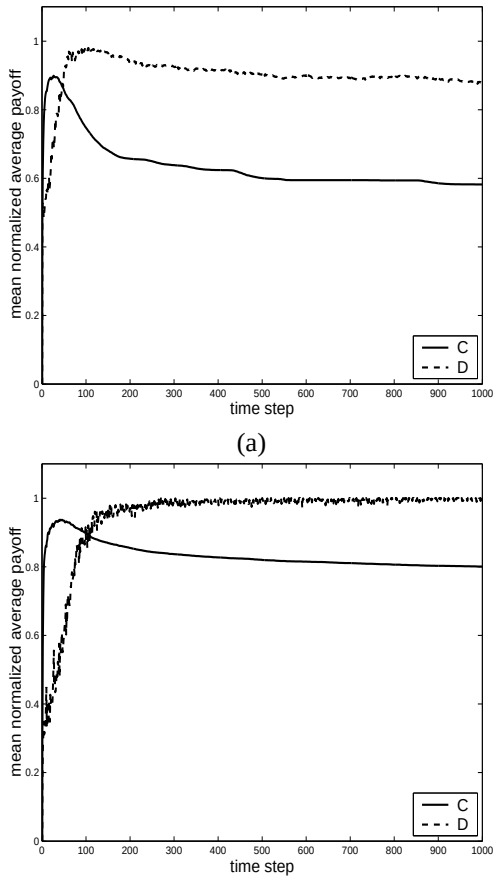


Fig. 5. Evolution of $\overline{\Pi}_C$ and $\overline{\Pi}_D$ averaged over 20 runs; (a) using synchronous update, (b) using asynchronous update.

An interesting point to notice is that when using an asynchronous update, all the D-players are able to reach and maintain their maximum payoff. The small fluctations that occur after system stability (generally reached around time step 150) are due to defectors getting totally disconnected from their neighbors (which were of course all unsatisfied cooperators). However, the lone defectors reattain their maximum payoff by finding a new C-neighbor usually in a matter of one to two time steps. In the synchronous case, the defectors on average seem to tend to their maximum payoff in the first 100 time steps, $\overline{\Pi}_D$ gently decreases. This is also true concerning the payoff of the cooperators which diminishes at the the same speed as $\overline{\Pi}_D$ does.

Fig. 5 unfortunately does not give a detailed picture of the rewiring at hand and the underlying organization, since once again it only shows an average of the 20 runs. In order to have a better understanding, we must have a look at a few particular runs and study not only $\overline{\Pi}_C$ and $\overline{\Pi}_D$, but also the evolution of the different types of degrees of the network.

The first thing to be observed is the cooperators forming a complete graph whether the update is synchronous or asynchronous (see Figs. 6 and 7, where, after a transient period, the average degree of cooperators becomes 19, which is the maximum possible degree given that there are 20 cooperators in the population). This is a direct consequence of the pressure $\sigma = 1$ exerts on the C-players forcing them to continue changing neighbors for as long as they are connected to defectors. Since defectors, on the other hand, continously seek to have interactions with them, the C-players inevitably end up forming a huge cluster where they are all linked to one another (Fig. 4).

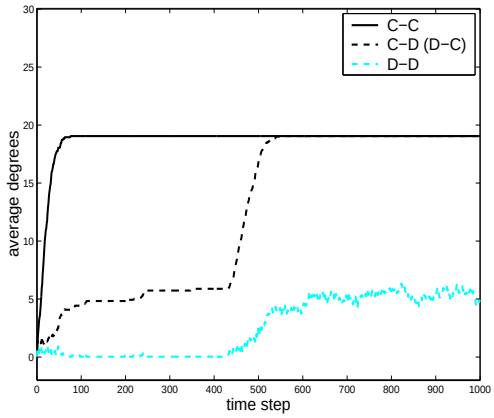


Fig. 6. Network degrees of a particular run; update: synchronous, population size: 40, $\sigma = 1$. C-D and D-C links are the same since there are as many cooperators as defectors.

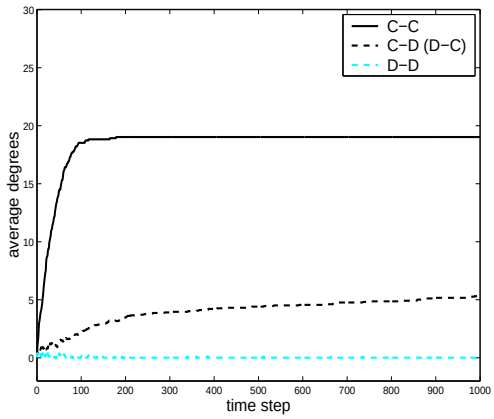


Fig. 7. Network degrees of a particular run; update: asynchronous, population size: 40, $\sigma = 1$. C-D and D-C links are the same since there are as many cooperators as defectors.

Secondly, as mentioned above, when using an asynchronous

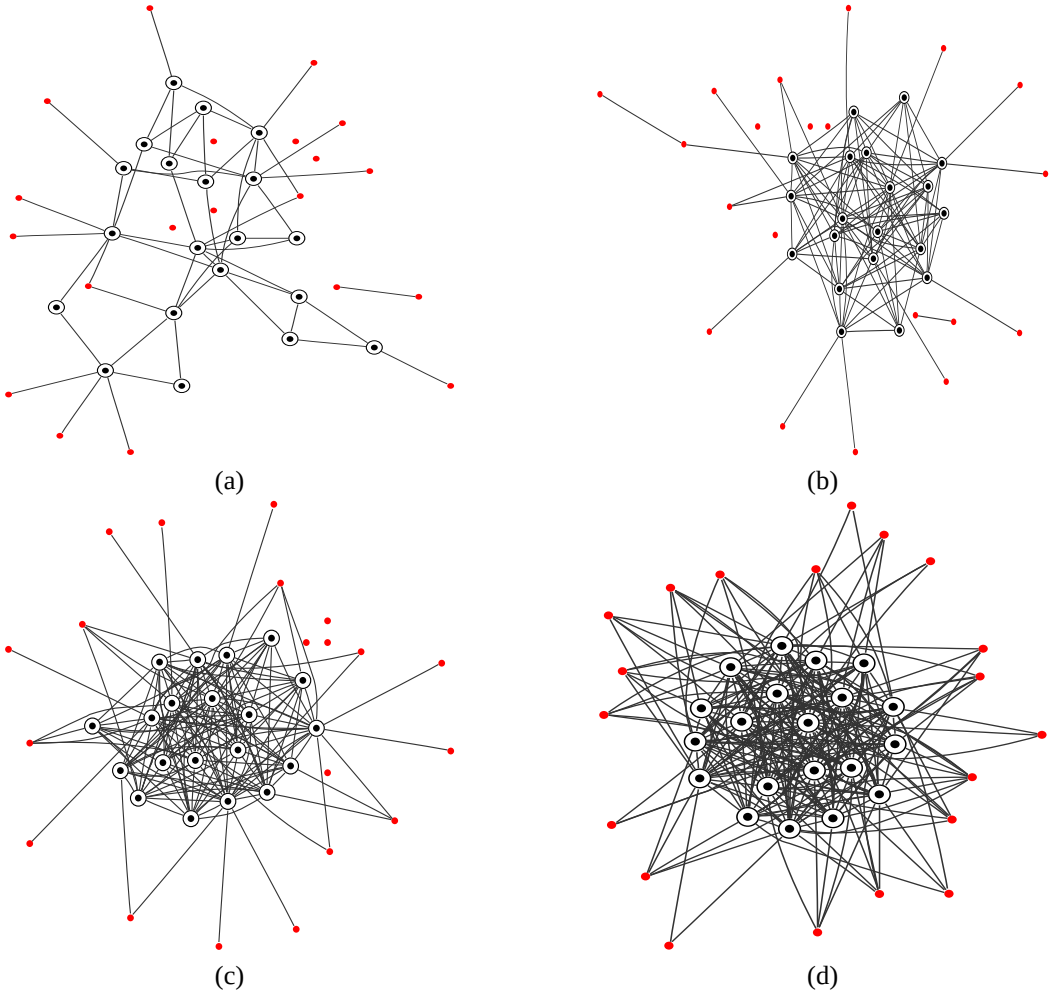


Fig. 4. A network of 40 players (20 C-players and 20 D-players) at different time steps with $\sigma = 1$ and using asynchronous update; (a) time step 10, (b) time step 30, (c) time step 50, (d) time step 1000 (end of run). Cooperators are represented by circled dots; defectors by simple dots.

update, all the defectors reach their maximum payoff and maintain it (Fig. 5(b)). Now, looking at Fig. 9 together with Fig. 10, showing respectively for the same run, the mean normalized average payoffs and the number of connected subcomponents the network comprises, we see that the local drops of $\overline{\Pi}_D$ are due to defectors finding themselves cut off from the cluster of cooperators and having thus a momentarily negative normalized average payoff (Eq. 1). These defectors will then randomly create a new link, rewiring it until it eventually reconnects them to a cooperator. Only then will the system reach once again a stable state. This effect is apparent in Fig. 10 where the number of components oscillates between one and two, in the limit of long simulation times. When there are two components, they are constituted, respectively, by a single defector and all the remaining players. Notice however that once the subgraph of cooperators is complete, these new links that end up connecting a defector with a cooperator will cause $\overline{\Pi}_C$ to gradually decrease since there is no possibility for a C-individual to rewire a link to another cooperator. Note that, although we have analyzed a single run, all the runs follow the same qualitative trend.

When using a synchronous update, things are not as simple. Fig. 6 and Fig. 8 show that although the system reaches a stable state, the latter is quite fragile and is not safe from collapsing to attain another type of stable state, as can be seen in Fig. 8. There are two mechanisms at work: The first one, responsible for the fall of $\overline{\Pi}_C$ and $\overline{\Pi}_D$ and, conversely, the abrupt increase of C-D and D-D connections is the same one discussed for the local drops of $\overline{\Pi}_D$ in the asynchronous case. The second mechanism occurs only after the first one has taken place. If the new link created at time step t_n by an isolated D-individual A connects it to another D-agent B, causing the latter to be unsatisfied, it will necessarily be rewired at time step t_{n+1} *by both players*. This "double rewiring" will generate, at time step t_{n+2}, two new links l_{A-X} and l_{B-Y} respectively connecting A with an agent X and B with an agent Y. For the sake of the argument, we will suppose that X and Y are two different players who were either lone players at t_{n+1} or were satisfied before being forced to interact with their corresponding new partner. Depending on X's strategy

s_X and Y's strategy s_Y, different outcomes are possible:

- $s_X = s_Y = 0$ (both are D):
  We have $\overline{\Pi}_D(t_{n+2}) < \overline{\Pi}_D(t_{n+1})$, but more importantly, A and X (resp. B and Y) will want to rewire l_{A-X} (resp. l_{B-Y}) elsewhere, with the risk of iterating this process several times generating one to two new links at each step. The consequence is a strong increase of the number of links in the network, and hence an increase of the clustering coefficient.
- $s_X = 0$, $s_Y = 1$ *or* $s_X = 1$, $s_Y = 0$:
  We find ourselves in the same situation encountered at time step t_{n+1}. Notice also that $\overline{\Pi}_C(t_{n+2})$ is slightly lower than $\overline{\Pi}_C(t_{n+1})$.
- $s_X = s_Y = 1$ (both are C):
  $\overline{\Pi}_D(t_{n+2}) > \overline{\Pi}_D(t_{n+1}) > \overline{\Pi}_D(t_n)$ and the system recovers its stability with a lower $\overline{\Pi}_C$.

At each step of the above process, there is about 75% chance for one of the first two cases to occur as long as the third case is not encountered. That is why, when using a synchronous update, approximately half the runs present sooner or later a rise of the different average degrees accompanied by a drop of both $\overline{\Pi}_C$ and $\overline{\Pi}_D$. The system usually reattains stability once the $C-D$ average degree reaches $\frac{N}{2}-1$ where N is the size of the population (see Fig. 6).

On the same figure, the fact that the D-D average degree does not increase past a certain level — about six in the figure — is due to the fact that the higher the D-D average degree, the higher the probability of two unsatisfied D agents breaking a link with other D players, and trying both to connect to one another. This results in two D-D links disappearing for only one new D-D link created.

To ascertain that the results obtained with 40 individuals are also valid for bigger size populations, a few very long runs (12000 time steps) were executed on the initial size of 1600 players. These runs — not shown here to save space — indeed confirm all the quantities measured with the small size population, such as the complete graph of cooperators and the fragile stable state induced by a synchronous update.

B. Different proportions of cooperators and defectors

What happens if the population is not composed of equal size subsets of cooperators and defectors? Do some aspects drastically change or are they all similar to the $50\% - 50\%$ case?

To answer these questions, we studied two different proportions of cooperators and defectors: 40% C - 60% D and 60% C - 40% D. Here we briefly report on the main results. For σ values between 0.0 and 0.9 the results are globally the same as the 50% C - 50% D proportion. However, in the 40% C - 60% D case, when using a synchronous update, we obtain higher average degree values, clearly due to the higher number of defectors increasing the probability of two connected D-players mutually deciding to break up and thus adding a new link to the network. For the same reason, defectors can never maintain the average maximum payoff T when $\sigma = 1$. As for the 60% C - 40% D case, the average degrees have, for the exactly symmetrical reasons, lower values than the 50% C -

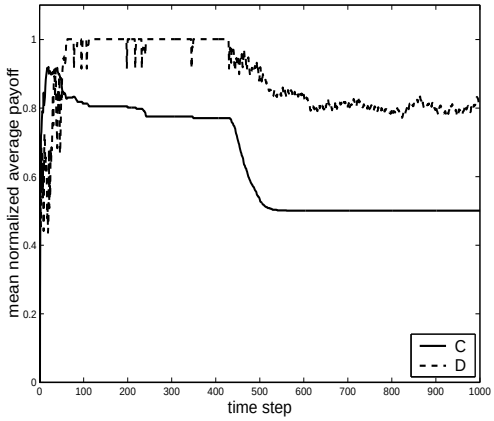


Fig. 8. Time evolution of the mean averaged payoffs $\overline{\Pi}_C$ and $\overline{\Pi}_D$ during a particular run; update: synchronous, population size: 40, $\sigma = 1$. The population recovers a few times from isolation of a D player but ends up falling into another network state with lower mean payoff.

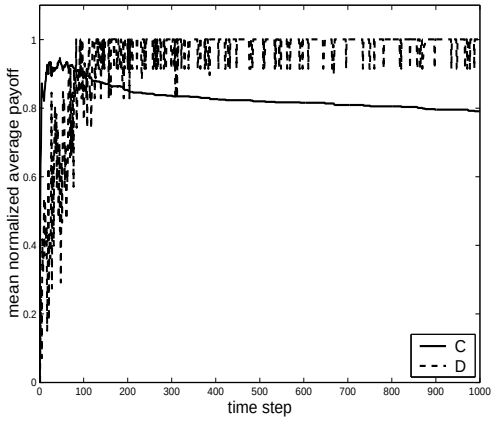


Fig. 9. Time volution of the mean averaged payoffs $\overline{\Pi}_C$ and $\overline{\Pi}_D$ during a particular run; update: asynchronous, population size: 40, $\sigma = 1$. With respect to the previous figure, the population always recovers after isolation of a single defector, thus keeping the average payoffs higher.

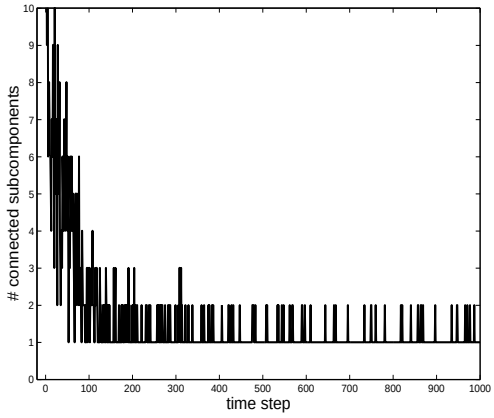


Fig. 10. The number of connected subcomponents of a particular run; update: asynchronous, population size: 40, $\sigma = 1$.

50% D case and the system always attains a stable state with the defectors at a maximum average payoff of T.

IV. Conclusions and Future Work

In this work we have studied by computer simulation the dynamical features of a population of individuals playing the one-shot prisoner's dilemma game. In spite of the fact that the players are not allowed to change their strategy, which is fixed at the outset, the network of relationships between players self-organizes towards a situation were the cooperators tend to cluster together and are surrounded by defectors.

The elementary rules that allow the adaptation of the topology are local and extremely simple: they are based on the satisfaction threshold of an individual, which is a function of its current average payoff at a given time step. Unsatisfied players break a random link with a defector neighbor and randomly choose another neighbor in the population. Clustering of cooperators as a way of sustaining cooperation, is well-known in spatial PD games. Here, however, it emerges in the absence of the possibility for a player to imitate another agent's strategy. Since the topology of the network plays an important role in social interactions, it is important to understand how and why this topology changes globally under the effect of simple local rules that represent likely actions of single agents that do not possess knowledge other than of their immediate surroundings.

Most spatial games simulations are of the synchronous type. However, especially for social interactions, some kind of asynchronicity in the players' actions seems to be more realistic. We have studied both fully synchronous and a simple asynchronous model. Although the long-term trend of the dynamical evolution of the population is similar in several respects for both update policies, the asynchronous model gives rise to more stable patterns and avoids some artificial bookeeping operations that are needed to simulate simultaneous update of the agents. In fact, the synchronous system dynamics can be sometimes subject to sudden instabilities that modify the system state in a non-trivial manner. Overall, asynchronous update seem to be preferable in situations such as those represented here.

In future work we would like to investigate the stability of the long-term steady states of the population in the face of random perturbations of the system. As well, the behavior of the system with respect to variations in the temptation parameter T should be investigated. Further steps should include studying the time evolution of populations in which players can locally change their strategy or their neighborhood, but not both. And, finally, we would like to study populations in which both the network topology and the strategies can change.

Acknowledgements

Mario Giacobini gratefully acknowledges financial support by the Fonds National Suisse pour la Recherche Scientifique under contract 200021-103732/1.

References

[1] R. D. Luce and H. Raiffa, *Games and Decisions*, John Wiley and Sons, New York, 1957.

[2] R. Axelrod, *The Evolution of Cooperation*, Basic Books, Inc., New-York, 1984.

[3] K. Lindgren and M. G. Nordahl, "Evolutionary dynamics of spatial games," *Physica D*, vol. 75, pp. 292–309, 1994.

[4] J. Maynard Smith, *Evolution and the Theory of Games*, Cambridge University Press, 1982.

[5] J. W. Weibull, *Evolutionary Game Theory*, MIT Press, Boston, MA, 1995.

[6] M. A. Nowak and R. M. May, "Evolutionary games and spatial chaos," *Nature*, vol. 359, pp. 826–829, October 1992.

[7] M. A. Nowak, S. Bonhoeffer, and R. M. May, "Spatial games and the maintenance of cooperation," *Proceedings of the National Academy of Sciences USA*, vol. 91, pp. 4877–4881, May 1994.

[8] M. G. Zimmermann, V. M. Eguíluz, and M. San Miguel, "Coevolution of dynamical states and interactions in dynamic networks," *Physical Review E*, vol. 69, pp. 065102(R), 2004.

[9] R. Albert and A.-L. Barabasi, "Statistical mechanics of complex networks," *Reviews of Modern Physics*, vol. 74, pp. 47–97, 2002.

[10] B. Schönfisch and A. de Roos, "Synchronous and asynchronous updating in cellular automata," *BioSystems*, vol. 51, pp. 123–143, 1999.

[11] M. Sipper, M. Tomassini, and M. S. Capcarrère, "Evolving asynchronous and scalable non-uniform cellular automata," in *Proceedings of International Conference on Artificial Neural Networks and Genetic Algorithms (ICANNGA97)*, G. D. Smith, N. C. Steele, and R. F. Albrecht, Eds. 1997, pp. 67–71, Springer-Verlag, Vienna.

[12] B. A. Huberman and N. S. Glance, "Evolutionary games and computer simulations," *Proceedings of the National Academy of Sciences USA*, vol. 90, pp. 7716–7718, August 1993.

Poster Presentations

Pared-down Poker: Cutting to the Core of Command and Control

Kevin Burns
The MITRE Corporation
202 Burlington Road
Bedford, MA 01730-1420 USA
kburns@mitre.org

Abstract- Poker poses cognitive challenges like those of warfare, business and other real world domains. This makes poker a good test bed for basic research on how people make Command and Control decisions and for applied research on how systems might help people make better decisions. In this paper, I compare the cognitive challenges of poker and warfare, and present a new suite of "Pared-down Poker" games that cut to the core of Command and Control. Compared to full-scale poker, Pared-down Poker is more tractable to normative analyses in the lab and more relevant to cognitive challenges in the world. The games have been programmed in Java along with various "animal archetypes" that simulate poker personalities. One game has been used to study the computational effectiveness of cognitive style against normative skill, and the findings from this study highlight questions for further research.

1 Introduction

Useful research on computational intelligence requires both *rigor* and *relevance*. Poker offers the promise of both, because poker is relatively tractable to mathematical computations (for rigor) and arguably applicable to practical situations (for relevance).

I say "relatively tractable" because standard poker is complicated by the many combinations of 5-card hands that can be made from a 52-card deck (Epstein 1977). I say "arguably applicable" because the link between poker and warfare or business or other domains has, to the best of my knowledge, been established only informally (McDonald 1950).

My interest is computational intelligence as applied to *Command and Control* of military missions. My approach begins with a basic model of Command and Control, which I relate to the game of poker. I then pare-down full-scale poker in a way that strengthens the connection between poker and warfare.

The result is a suite of *Pared-down Poker* games that are, compared to standard poker, more tractable to computational investigations (for rigor) and more formally related to Command and Control (for relevance). These games have been programmed in Java, along with *animal archetypes* (software robots) that simulate poker personalities. In this paper, I describe the games and present the results of a pilot study on style and skill.

How does this work differ from previous work on poker in Game Theory and Artificial Intelligence? One difference is that Pared-down Poker strikes a novel balance between the stripped-down (extremely simple) pokers studied in Game Theory (Kuhn 1950) and the full-scale (extremely complex) pokers studied in Artificial Intelligence (Billings et al. 2003). Pared-down Poker is a suite of four games ranging from simple to complex in order to facilitate a progressive research program where findings from the simpler games can be used to inform studies of more complex games. Another difference is that Pared-down Poker is designed for research on human strategies (also see Findler 1977), as opposed to machine strategies for optimal play in stripped-down poker or near-optimal play in full-scale poker. Of course, a thorough study of cognitive strategies must include models of normative or near-normative strategies in order to benchmark *how well* people perform. And in this respect, Pared-down Poker is more useful than full-scale poker because it is more tractable to normative analyses of optimal solutions.

2 The Four Steps of Command and Control

In military missions, the term *Command and Control* has a specific meaning that refers to:

"a process... [that] begins with assessing the battlefield situation from available information. Following this assessment, the *commander* decides on a course of action. The commander then implements this decision by directing and *controlling* available forces. The final step… is evaluating the impact of the action on both friendly and opposing forces. This evaluation then serves as an input into an updated assessment of the situation, and the process continues." (Mandeles et al. 1996, pg. 4).

Based on this definition, Command and Control can be boiled down to four steps: (1) *inference* – in assessing a situation, (2) *investment* – in deciding on a course of action, (3) *implementation* – in acting to carry out the decision, and (4) *iteration* – in adapting to new situations. Of course, the same four steps are involved in many decision making tasks, and any information processing task can be characterized as having a beginning, middle, end and sequel – so *Command and Control* also occurs in business, medicine and other non-military domains. However, below I use the term in its more narrow sense at it applies to warfare (and poker).

At a functional (strategic) level, poker is often compared to business and warfare (McDonald 1950). Here, to make the connection more explicit, I note the following: At ***step 1*** of Command and Control, a poker player must make *inferences* about the relative strength of his own forces (cards and chips) relative to his opponents' forces. And, he must do so with only partial information about his opponents' forces (from exposed cards and previous bets) under changing conditions (as cards are dealt and drawn and as chips are bet and won) in the course of a battle (hand). At ***step 2*** of Command and Control, a poker player must make *investments* (of chips) in order to achieve a desired outcome, where his desire may be to win the current battle (pot) or to set up wins in future battles. At ***step 3*** of Command and Control, *implementation* by a poker player must be in a manner that does not give away information about his forces or strategies, and perhaps even in a manner intended to convey deceptive information. Finally, at ***step 4*** of Command and Control, *iteration* by a poker player must account for changes in available forces (cards and chips), both his own and his opponents'. A poker player must also adapt his strategies based on what he has discovered about his opponents in previous battles (hands) and what he expects of his opponents in future battles.

In the real world, Command and Control is accomplished by "individuals working in specific offices within larger organizations" (Mandeles et al. 1996, pg. 5), and modern warfare poses many challenges of communication and coordination between individuals and organizations. These challenges are typically addressed by operating procedures and computerized *systems* and *subsystems*, like those aboard the US Air Force's Airborne Warning and Control System. Here, at the systemic (tactical) level, poker bears less resemblance to Command and Control than at the strategic (functional) level because a poker player is an individual sitting at a card table rather than an organization of individuals sitting at war consoles (computers). However, poker can be played on computers, and computer poker can be played on teams in settings where teammates can communicate via systems. And even without formal teams, poker can give rise to temporary coalitions acting in *co-opetition* (Brandenburger and Nalebuff 1996), where weaker players cooperate to a limited extent against stronger players, which also occurs in warfare.

Moreover, tactical Command and Control systems are typically distributed among individuals and organizations because they *must* be for physical reasons and not because they *should* be for optimal outcomes (Mandeles et al. 1996, Snook 2000). Therefore, a poker perspective that focuses on strategic functions may be useful for improving the integration of tactical systems across individuals and organizations. Integration is a critical component of next generation Command and Control systems, like the US Air Force's Multisensor Command and Control Aircraft (Tirpak 2002). As such, poker has relevance to both strategic functions and tactical systems in modern warfare.

In short, when Command and Control is cut to the core, there are four steps that map to four skills of poker playing. This makes computer-based poker a useful test bed for investigating how (and how well) people perform in Command and Control, and how (and how well) systems might support people in dealing with the cognitive challenges of modern warfare. With this background and purpose, I now take a closer look at the cognitive challenges of poker (and warfare).

3 The Four Skills of Poker Playing

As an example, consider a simplified (pared-down) game of poker between two players, Player A and Player B, illustrated in Figure 1. Assume that each player has anted 1 chip to the pot and Player A makes a bet of 2 chips (so pot=1+1+2=4 chips). Player B is then faced with the choice between fold, call or raise. Below I dissect this decision following the four steps of *inference, investment, implementation and iteration.*

3.1 Inference

Before he does anything else, Player B must first make an *inference* about the strength of Player A's hand, i.e., he must compare the expected strength of A's hand to the actual strength of his own hand. To make this *inference*, Player B gets only partial information from the cards (e.g., Player A cannot have any cards that are in Player B's hand) and from Player A's bet. But did Player A bet because he has a strong hand and he wants to build the pot? Or did Player A bet as a bluff, hoping the bet would cause Player B to fold so Player A could take the pot with a weak hand? Or did Player A not really think much at all when he made his bet?

To answer these questions, Player B needs a *causal* model of Player A that can be used to estimate $P(A_{bet}|A_h)$, e.g., to estimate the probability that Player A would make a bet (as he did in this example) given a possible hand A_h that would beat Player B's hand. Without such a model, Player B has no basis for inferring the relative strength of Player A's hand. However, if Player B has a causal model of Player A, then he can do some more thinking.

In fact, a normative (optimal) Player B needs to estimate $P(A_{bet}|A_h)$, which is the *likelihood* that Player A would bet given a hand A_h, along with the *prior* probability $P(A_h)$ that Player A would have been dealt a hand A_h. With these probabilities, Player B can then compute the probability that Player A has a hand A_h given the bet, $P(A_h|A_{bet})$, by a process know as *Bayesian inference* using the following equation:

$$P(A_h|A_{bet}) = P(A_h)*P(A_{bet}|A_h) / P(A_{bet}), \text{ where}$$

$$P(A_{bet}) = P(A_h)*P(A_{bet}|A_h) + \Sigma P(A_i)*P(A_{bet}|A_i)$$

where the subscript i refers to other hands in the set of possibilities that are not A_h. This *posterior* probability, $P(A_h|A_{bet})$, is what Player B needs in order to judge the chance that Player A has a hand A_h that is better than the hand that Player B holds himself. As such, Bayesian *inference* or some process that approximates it is first and foremost in poker (and warfare), even if poker players (or Commanders and Controllers) do not consciously recognize it as such (also see Korb et al. 1999).

Notice that a causal model (see above) is needed to estimate the likelihood, $P(A_{bet}|A_h)$, but this likelihood itself is not the *inference* of interest because what Player B really needs to know is the posterior $P(A_h|A_{bet})$. The two are different but related by the prior $P(A_h)$ via Bayes Rule (see above), and this is why Bayesian *inference* is critical to poker (and warfare). Then later, as further evidence is provided by another bet or raise in the same hand, the posterior $P(A_h|A_{bet})$ becomes the prior for a subsequent Bayesian *inference* using another likelihood from an appropriate causal model to get an updated posterior, etc.

Now to continue the example, assume that Player B's *inference* puts the odds at 2:1 (probability=2/3=67%) that he (Player B) has a better hand than Player A. The question now is: Should he fold, call or raise?

3.2 Investment

Player B's choice involves an *investment* of chips. Thus, to make the choice Player B needs to know both his win:loss odds (see *inference*, above) as well as the cost:pot stakes. In the example here, the pot contains 4 chips when it is Player B's turn to act after both players have anted 1 chip and Player A has bet 2 more chips. The cost for Player B to call the bet is 2 chips, so the cost:pot stakes for Player B are 2:4.

With cost:pot stakes of 2:4 and win:loss odds of 2:1, the expected utility for a call by Player B is 6*(2/3)+0*(1/3)=4 chips, since a win will pay him 6 chips and a loss will pay him 0 chips. The expected utility of a fold by Player B is 2 chips, because he will give up the pot (with probability 1) but retain the 2 chips it would cost him to call. Since the expected utility of a call is 4 chips and the expected utility of a fold is only 2 chips, Player B should call rather than fold. But, should he make an even larger *investment* and raise rather than call?

To make this choice, Player B must be able to predict what Player A will do in response to a raise. Is the raise likely to make A fold his hand, in which case B should raise with almost any hand? Or is Player A likely to call because Player B's raise feeds into a setup that A had been planning all along, ever since A's initial bet? Like the case of *inference* above, answers to these *investment* questions depend on having a causal model of Player A.

With such a model, Player B can estimate the chances that A will call or fold in response to a raise, $P(A_{call}|B_{raise})$ and $P(A_{fold}|B_{raise})$. Player B can then go further to select the option (call or raise) that will maximize his expected utility. Without such a model, Player B must use some other (sub-optimal) strategy to make a choice.

3.3 Implementation

Once he makes an *investment* decision, Player B is faced with the challenge of *implementation*. Here he must take chips (resources) from his stack and put them in the pot (battlefield) in a manner that does not give Player A any additional information about his hand strength or strategy. In fact, the problem of *implementation* arises even before chips are put in the pot, since Player B may convey information about his hand strength or strategy while he ponders his *inference* and *investment* decisions.

In poker jargon, the unintentional conveying of information by mannerisms in *implementation* is called a *tell*. In some cases, a tell can provide as much or more information about an opponent's hand as a bet or raise. Of course, in computer-based poker, some tells like trembling fingers and dilated pupils (both of which usually signify a good hand) cannot be observed and hence do not play a role. However, other mannerisms like how long it takes a player to make a decision, or table chat ("acting") via text messages can provide players with information about their opponents' hand strengths and strategies. Here, as in warfare, the key to winning poker is to outwit opponents in a battle of "he thinks that I think that he thinks…". And, as in modern warfare, computer-based poker provides less direct information than hand-to-hand combat, which makes it that much more important to extract the maximum certainty from available information.

3.4 Iteration

For a specific decision, like the call or raise choice of Player B in the above example, the cognitive challenges of poker boil down to the 1-2-3 of *inference*, *investment* and *implementation*. But poker, like warfare, is not a one shot deal. Rather, poker involves multiple hands (deals) in a temporal sequence, and even within a given hand there are usually several rounds of betting as the game state changes when cards are dealt or drawn, or when chips are bet or raised, or when other players fold.

Thus, the 1-2-3 steps of *inference*, *investment* and *implementation* are repeated in *iteration* as players adjust to changes in the game state (from round to round) and to data they get on opponents' behavior (from hand to hand). The latter is particularly important, and the use of such data is actually a higher-level process of Bayesian *inference* in which a player must update causal models based on opponents' behavior.

Recall that, to make an *inference* about the relative strength of his hand, a player needs a causal model of his opponent that he can use to estimate likelihoods of the form P(bet|hand, model), where "hand" is the cause of the opponent's "bet" in a "model". The problem in *iteration* is that there are many possible models, so a player must make a second-order *inference* about P(model|data), which he can do in Bayesian fashion if he has a meta-model that provides an estimate of the *prior* P(model) as well as the *likelihood* P(data|model). So here again, but now at a higher level, we see the importance of Bayesian *inference* in poker (and warfare).

But the challenge of *iteration* goes even beyond these higher-level Bayesian *inferences* that are needed to update causal models of opponent behaviors. That is, a player must also assess the previous (and project the upcoming) effectiveness of his strategies. For example, a player's *investment* strategies must consider the accuracy of his *inferences* (and underlying causal models) as well as his ability to estimate probabilities and to anticipate possibilities. Similarly, a player's *implementation* strategies must consider *tells* and *tilt* (emotional response) by himself and his opponents. These are practical and critical aspects of human poker and warfare, which are typically ignored in studies of machine poker and which motivate my study of human and machine performance in Pared-down Poker.

4 Full-scale Poker

Poker games differ primarily in their dealing rules and betting limits. The dealing rules are usually one of three major types. In *Draw Poker*, all cards are dealt face down and some cards may be exchanged via discarding and drawing. In *Stud Poker*, some cards are dealt face up to each player. In other pokers, like *Texas Hold'em*, some cards are dealt face up and shared by all players.

The betting limits are also one of three major types. In *fixed limit* games, the size of each bet and raise is a fixed amount. In *spread limit* games, the bets and raises can be any amount between certain limits. In *no limit* games (often played with dealing rules of Texas Hold'em), the size of a bet or raise is limited only by a player's bankroll.

Given some betting limits (see above), the betting rules for most poker games are as follows: Each hand starts with a pre-deal payment of chips to the pot, either as an *ante* (by all players) or a *blind bet* (by only some players, rotating with the deal). This is followed by one or more rounds of betting where the game state changes via dealing or drawing cards (per dealing rules, see above) between each betting round. Within a given betting round, the betting rules are as follows: starting with one player (usually left of the dealer), the player can either *check or bet*. [A check is like a "pass" in which no chips are added to the pot but the player stays in the hand. In some games, checking is not allowed and the player is forced to *bet or fold*.] Once one player bets, the options for subsequent players are *fold, call or raise*. [There is a limit on the number of raises per round.]

Besides some dealing rules and betting limits, a poker game needs a ranking structure to determine which hand will win a showdown. In full-scale poker, where the deck contains 52 cards and each hand is made of 5 cards, the hands are ranked by classes such as Royal Flush, Straight Flush, Four-of-a-Kind, Full House, etc. These ranks are based on relative rareness, but then there are further sub-ranks that are not based on rareness. For example, Four Queens beats Four Jacks even though Queens are not more rare than Jacks. [And in some games the object is get the lowest rank.]

From a research perspective, if not a player's perspective, the problem with full-scale poker is that the optimal decision in a given situation is intractable to closed-form analytical solution (Epstein 1977). In fact, even brute-force numerical solutions are beyond the current state of the art (Billings et al. 2003, Koller and Pfeffer 1997) because, unlike chess and other games of perfect information, poker is a game of *imperfect information* where there are millions of possible game states to consider at each step.

Although the combinatorial problems of 5-card hands dealt from a 52-card deck may be mathematically interesting, they are not particularly interesting from either a cognitive perspective or a practical perspective. From a cognitive perspective, the interesting issues are the big four of *inference, investment, implementation* and *iteration*, and these issues can be studied with more *rigor* in the context of simpler games that use fewer cards in the deck and fewer cards in each hand.

From a practical perspective, poker played with fewer cards is also more *relevant* to real world problems. This is because the types of probabilistic problems faced in the real world are rarely like the combinatorial problems posed by 5-card hands dealt from 52-card decks – i.e., poker dealings result from *random* sampling (of a known deck) while real world events arise from *causal* factors (often unknown). As such, the probabilistic problems that are most similar between poker and warfare are those that deal with Bayesian *inferences* (see above) and the associated causal models of opponents' intentions. By reducing the combinatorial problems of full-scale poker, with smaller hands dealt from a smaller deck, pared-down poker allows players and scientists to focus on the more *relevant* problems of causal models and Bayesian *inference.*

In short, pared-down poker offers advantages of both *rigor* and *relevance* over full-scale poker for the study computational intelligence. And it is on this basis that I designed a suite of *Pared-down Pok*er games for research on the big four issues of *inference, investment, implementation* and *iteration*.

5 Pared-down Poker

Pared-down Poker is a suite of four games ranging from simple to complex for progressive research on the big four issues. While all of the games are pared-down in comparison to full-scale poker, the most complex game of Pared-down Poker is similar to the full-scale pokers studied in the field of Artificial Intelligence. At the other end of the spectrum, the simplest game of Pared-down Poker is similar to the stripped-down pokers studied in the field of Game Theory. As such, the suite of Pared-down Poker games can help bridge a gap left by previous research in computer science and mathematics, especially for those with an interest in cognitive (not just normative) performance – as in the new field of *Behavioral Game Theory* (Camerer 2003).

The four games of Pared-down Poker are designed to be easy to learn and easy to play but hard to play well (depending on the opponents). The games are also designed to be fun to play, and they get more fun as they move from simple to complex. Here I would note that fun and games are not trivial; rather they are critical because the games pose cognitive challenges like those in the real world and because the fun keeps people engaged in experiments.

5.1 Four Games

There are four games of Pared-down Poker, each of which can be played by 2-4 players. The four games, from simple to complex, are named as follows:

One Card High
Two Card High
Pairs and Straights
Get a Clue

In *One Card High*, each player gets one card from a deck of 11 cards. The betting structure of this game is illustrated by the game tree in Figure 1. In *Two Card High*, each player gets two cards (one per round, in two rounds) from a deck of 22 cards. *Pairs and Straights*, which is also played with two cards in each hand, is a cross between pre-flop Texas Hold'em and Draw Poker (for readers familiar with these games). *Get a Clue* is an even more complex game played with two cards in each hand, where the backs of the cards are marked (with their suit) to give players clues to the fronts of the cards.

5.2 The Rules

In ***Two Card High,*** each player gets two cards and the player with the highest card wins in a showdown. The game can be played with fixed limits or no limits on bets and raises.

The deal is from a deck of 22 Cards. There are two suits and any two will do (although our Java games use a custom-designed card deck with Red and Blue suits). Each suit has 11 cards numbered Zero (Joker) to Ten. Zero (Joker) is the lowest card and Ten is the highest card. Each hand is made from two cards and a hand is ranked by its high card. For example, a hand of 8 and 3 beats a hand of 7 and 6 because 8 > 7. A hand of 8 and 3 beats a hand of 8 and 2 because 3 > 2.

There are two rounds of dealing and betting. To start, each player makes an ante to the pot and is dealt one card. This is followed by a round of betting, which starts to the left of the dealer. The first player must bet or fold (no checking is allowed). The next player can either raise, call or fold. The next player can either re-raise, call or fold, but there is a limit of two raises (one re-raise) per round. Players who do not fold are dealt another card, and this is followed by another round of betting.

A player wins the pot when he has the highest hand in a showdown or when all other players have folded.

5.3 More Rules

In a simpler game, called ***One Card High***, each player is dealt only one card from a half deck (one suit, 11 cards). There is only one round of betting and only one raise allowed (see Figure 1).

In a harder game, called ***Pairs and Straights***, each hand has two cards but the hands are ranked as either a *Pair* like (8, 8), a *Straight* like (8, 7) or a *Mutt* like (8, 2). A *Pair* beats a *Straight* and a *Straight* beats a *Mutt*. Within each class, the higher the better. For example: a *Pair* (8, 8) beats a *Pair* (7, 7); a *Straight* (8, 7) beats a *Straight* (7, 6); a *Mutt* (8, 2) beats a *Mutt* (7, 5). For *Straights*, (10, 9) is the highest and (0, 10) is the lowest. The deal is two cards to each player, then a round of betting, then the option to discard and draw a card, then another round of betting.

In the hardest game, called ***Get a Clue***, the hands are dealt and ranked like *Pairs and Straights* but the cards have colored backs (clues) that match the suits on the fronts of the cards. This gives you a clue to your opponents' hands (e.g., two Red-backed cards cannot be a *Pair*), and it gives them a clue to your hand.

5.4 Software

The four games of Pared-down Poker (above) have been programmed in Java to support online experiments. In the Java versions (Burns 2005), play is between 2-4 players and each player can be either a human being (connected on the web) or a software robot. The default version is one human playing against *animal archetypes* (software robots) that simulate the "style and skill" of poker personalities (see below).

The Java versions are designed to be played online so that cognitive experiments can be performed anywhere and anytime that people have a web-connected computer. A pilot version is available for demonstration on the world wide web (Burns 2005). A screen shot from this version is shown in Figure 2.

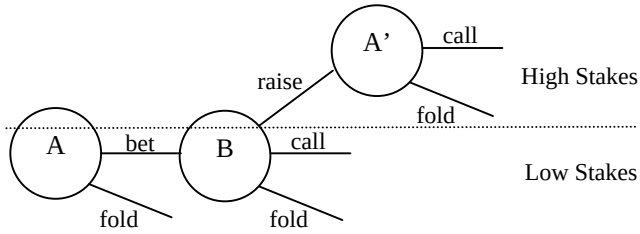


Figure 1. The game tree for a simple Pared-down Poker played "heads-up" (two players) in one round with no checking and one raise. To start, Player A must bet or fold. If Player A bets then Player B must raise, call or fold. If Player B raises then Player A (denoted A') must call or fold. In an example game (discussed in the text), betting amounts are fixed limits where the ante is 1 chip, a bet is 2 chips and a raise is 2 more chips.

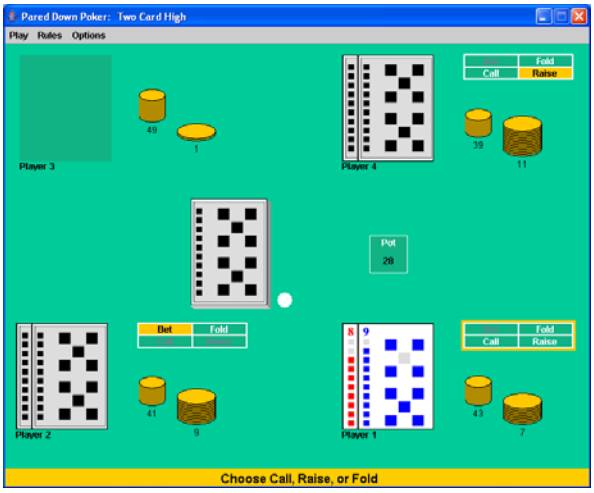


Figure 2. Screen shot of Pared-down Poker, as programmed in Java and playable on the web. A human (lower right) is playing the game of *Two Card High* against three software robots (one has folded).

On the upper left of the game screen is a menu that allows players to choose a game (one of four, see above) and read the rules. The menu also includes options, which are expanded to *display options*, *player options* and *betting options*. The display options include options to display additional information about other players' cards (e.g., flip cards face up when a player folds). The player options include options to change the number of players (from 2-4) as well as the skill (e.g., Novice or Expert) and style (e.g., Tight or Loose and Passive or Aggressive) of robot players. The betting options allow any of the four games to be played with fixed limits (adjustable to any amount) or no limits.

These options are designed to provide flexibility for experimental purposes. For example, in human testing, the display features can be changed to give more or less information and thereby study how well people exploit the extra information. Similarly, the player features can be changed to study how well people adapt to different styles of opponents and different numbers of opponents. Finally, the betting features can be changed to study how well people adapt their betting strategies as the stakes are raised to high limits (or no limits).

6 The Four Styles of Poker Playing

As discussed above, the four skills of poker playing can be characterized in normative terms, e.g., for the skill of *inference* the normative strategy is Bayesian *inference*.

The question now is: *How* and *how well* do people play poker (Command and Control)? That is: What are the cognitive means by which people perform and how do they stack up to normative skills? Answers to this question are important for a cognitive-scientific understanding of poker and for cognitive-engineering applications in warfare.

There are many "how-to" books that shed light on the question of how people play poker. Most of these books offer only street-smart folklore rather than formal models, but since the folklore is street-smart (i.e., somewhat tested in table practice) it provides a starting point for the development of more formal models.

For example, poker writers often draw a distinction between *style* and *skill* but do not really specify how the two are distinguished. In an effort to formalize the difference, I proposed the following as a working definition (Burns 2004): **Style is a functional basis for making decisions in the absence of a rational basis for making decisions, i.e., when the expected utility for each option is the same.**

By this definition, skill is the ability to make normative *inferences* (i.e., about Bayesian probability) and *investments* (i.e., for maximum utility). The problem, of course, is that people have mental limits of *precision* in estimating odds and stakes, as well as mental limits of *projection* in estimating future events. Bound by these mental limits, many choices look like a toss up and when faced with these choices people rely on personal preferences that are commonly referred to as *style* (see Burns 2004).

As a formal basis for different styles, I proposed a binary distinction between *conservative* and *non-conservative* preferences as well as a contextual distinction between *low stakes* and *high stakes* consequences. The binary distinction between conservative and non-conservative captures the major styles in poker playing, i.e., *Tight versus Loose* and *Passive versus Aggressive*. The contextual distinction captures the difference between the *low stakes* (Tight-Loose) dimension of style and the *high stakes* (Passive-Aggressive) dimension of style, since poker players are often non-conservative at low stakes but become conservative as the stakes are raised, or vice versa. Thus, there are four major styles of poker personalities as illustrated in Figure 3 (also see Schoonmaker 2000).

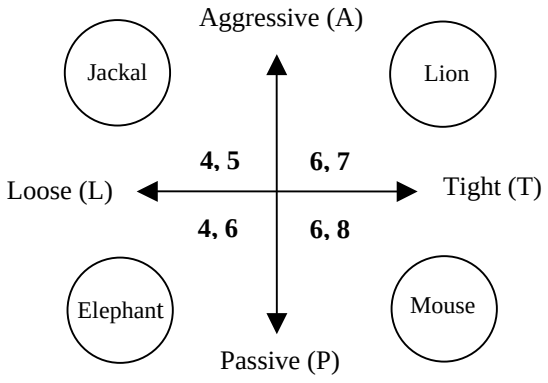


Figure 3. The four styles of poker playing are: Tight-Passive (Mouse), Tight-Aggressive (Lion), Loose-Passive (Elephant) and Loose-Aggressive (Jackal). The noted numbers are minimum bet and raise cards in the game of *One Card High* (see text for details).

The four styles illustrated in Figure 3 are so common (and useful) in the poker folklore that they are often labeled as *animal archetypes* (personalities). The animals in Figure 3 are adopted from Hellmuth (2003), who distinguishes between four animals with different styles at Novice skill. Hellmuth also defines a fifth animal (Eagle) at a higher level of Expert skill.

7 Style and Skill in Pared-down Poker

Pared-down Poker can be used to perform computational investigations of *how* people make decisions as well as *how well* people make decisions As an example of these uses, I performed a pilot study of *style* and *skill* in Pared-down Poker (Burns 2004).

7.1 Assumptions

My analysis was motivated by both practical and theoretical matters. As a practical matter, poker writers (Sklansky 1987) note that Novice players (who play with different styles, see Figure 3) typically make their betting decisions with little or no regard for what other players do. That is, they follow rules like: If I have a hand of X or better then I will bet, otherwise I will fold. As a theoretical matter, such simple rules are far from the optimal strategies, which involve detailed calculations of probabilities and utilities for complex scenarios. Thus, the question is: How well do different Novice styles (in Figure 3, using simple rules) perform against one another and against Expert skill (as defined by normative strategies)?

To answer this question, I developed models of Novice styles and Expert skill in the game of *One Card High* (see Figure 1). The Novice styles are defined by two numbers, (X, Y), which are the minimum cards at which a player will bet or call a bet (X) and raise or call a raise (Y).

7.2 Calculations

The assumed values of (X, Y) for each style are noted in Figure 3. Using these values as simple thresholds (betting rules), I held computer *face-offs* in *heads-up* (pairwise) play between the different Novice styles. For example, in a face-off, a Mouse will always bet or call a bet if he has a 6 or higher card, and he will always raise or call a raise if he has an 8 or higher card. Each face-off was a set of 220 games that captured all 110 combinations of cards that the two players could be dealt from a deck of 11 cards, with each combination played in each order (i.e., each player in role A and role B, see Figure 1).

I also held computer face-offs between each Novice style and Expert skill. Here I assumed that the Expert knew his opponent's style and used this information along with normative equations to compute expected utilities and optimize his decisions. For example, at node A (see Figure 1) the expected utility of a bet by Player A is: $U_{A,bet} = P_{B,fold}*[b+(b+2a)] + P_{B,call}*[b+P_{A,win|B,call}*(2b+2a)] + P_{B,raise}*[P_{A',call|B,raise}* P_{A',win|B,raise}* (4b+2a) + P_{A',fold|B,raise}* (b)]$, where a=ante=1 chip and b=bet=2 chips for the fixed limit game analyzed in my face-offs (see Burns 2004).

Figure 4 shows the results of the face-offs between each pair of Novice styles, and between each Novice style and Expert skill. With respect to the question of how well different styles perform against each other, Figure 4 shows that the Mouse beats all other styles. However, the results are nonlinear: the Mouse beats both the Lion and the Jackal, yet the Lion beats the Jackal by more chips than the Mouse beats the Jackal. These results provide formal support for the informal (street-smart) notion that it is important to adapt one's style to an opponent's style. For example, based on Figure 4, one should play like a Mouse when facing a Lion, but one should play like a Lion when facing a Jackal.

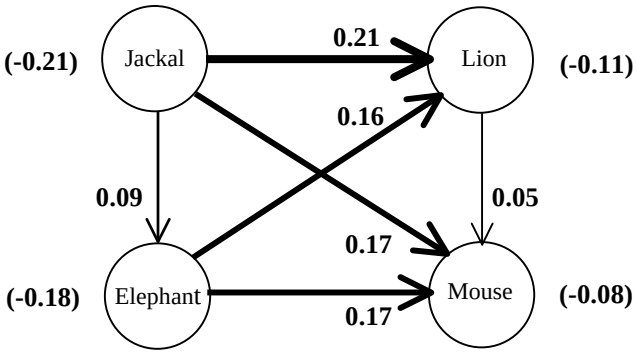


Figure 4. Results of pair-wise face-offs between each Novice style. Arrowheads point to winning style. Numbers are earnings (chips/game). Line thickness reflects earnings. Parenthetic numbers are losses against an Eagle (Expert).

With respect to the question of how well Novice styles perform against Expert skill, Figure 4 shows that style is remarkably effective against skill. That is, while each Novice style loses to Expert skill (by parenthetical numbers in Figure 4), the losses are in the same ballpark (0.1-0.2 chips/game) as the losses of Novice styles to each other. And in some cases, like the Lion against the Jackal, a Novice does as well as the Expert. This shows that cognitive strategies can be very effective, even when they are very simplistic.

7.3 Questions

Based on these findings, the questions are: *Why* are these simplistic strategies so effective? *When* do simple styles break down such that poker players should develop more complex skills? *How* do poker players acquire styles and skills, and *how well* do they adapt to time-changing conditions and style-changing opponents? As a practical matter, answers to these questions are needed to guide the design of *adaptive* support systems that leave people alone in cases where their cognitive strategies work well and help people perform in cases where their cognitive strategies do not work well.

With respect to the question of why the Novice strategies perform so well in *One Card High*, there are two reasons (see Burns 2004). First, winning poker (pared-down or full-scale) takes a lot of luck and not that much skill when it is played with low limits on bets and raises. In fact, this is the reason that poker pros prefer high limit or no limit games, where advanced strategies like bluffing become important. Bluffing allows a player to win chips even when he is not dealt a good hand – but bluffing does not work for low limit games because your opponents can call your bluffs without much cost.

The other reason that a Novice like the Mouse does so well is that his simple strategy (style) is functionally equivalent to a much more complex strategy (skill). In particular, the Mouse's style, which requires a better than average card to bet and a significantly higher card to raise, implicitly captures two features of strategic skill, namely: (1) one should bet and raise only when one has a decent chance of winning a showdown and/or a decent chance of causing one's opponent to fold, and (2) one's chance of winning a showdown and chance of an opponent folding are typically less after an opponent has raised if the opponent is also considering (1).

7.4 Extensions

Based on this pilot study of style and skill, further analyses and experiments are planned to explore the questions outlined above. The test bed for these investigations will be the Java versions of Pared-down Poker, which allow online experiments with human players competing against each other and/or against software robots.

Motivated by the dual need for *rigor* and *relevance*, this research will focus on the four skills of poker playing that are also the four steps of Command and Control, namely: ***inference, investment, implementation and iteration.***

The ultimate objective of this research is to improve the design of computer systems that advise and automate cognitive functions in real world Command and Control. Towards this end, Pared-down Poker provides a test bed for understanding cognitive strengths and limits. It also provides a test bed for evaluating support systems that might be designed and employed to improve cognitive performance.

8 Conclusion

Motivated by the need to balance rigor (tractability) and relevance (applicability), I proposed a new suite of Pared-down Poker games that cut to the core of Command and Control. In designing these games, I highlighted the cognitive connections between poker and warfare. I also formalized the difference between style and skill, and analyzed a simple game of Pared-down Poker to set the stage for further research.

Acknowledgments

This research was supported by the MITRE Technology Program. Thanks to Craig Bonaceto and Eric Kappotis for software development and helpful discussions.

Bibliography

Billings, D., Burch, N., Davidson, A., Holte, R., Schaeffer, J., Schauenberg, T., and Szafron, D. (2003) Approximating Game-Theoretic Optimal Strategies for Full-Scale Poker. *Proceedings of the International Joint Conference on Artificial Intelligence*.

Brandenburger, A. M., and Nalebuff, B. J. (1996) *Co-opetition*. New York, NY: Doubleday.

Burns, K. (2005) *Mental Models in Naturalistic Decision Making*, http://mentalmodels.mitre.org.

Burns, K. (2004) Heads-Up Face-Off: On Style and Skill in the Game of Poker. *Proceedings of the AAAI Fall Symposium on Style and Meaning in Language, Art, Music and Design*. Washington, DC, October 21-24.

Camerer, C. F. (2003) *Behavioral Game Theory: Experiments in Strategic Interaction*. Princeton, NJ: Princeton University Press.

Epstein, R. A. (1977) *The Theory of Gambling and Statistical Logic*. San Diego, CA: Academic Press.

Findler, N. V. (1977) Studies in Machine Cognition Using the Game of Poker. *Communications of the ACM* 20(4):230-245.

Hellmuth, P. (2003) *Play Poker Like the Pros*. New York, NY: Harper-Collins.

Koller, D., and Pfeffer, A. (1997) Representations and Solutions for Game-Theoretic Problems. *Artificial Intelligence* 94:167-215.

Korb, K., Nicholson, A., and Jitnah, N. (1999) Bayesian Poker. *Proceedings of the 15th Annual Conference on Uncertainty in Artificial Intelligence.*

Kuhn, H. W. (1950) A Simplified Two-Person Poker. *Contributions to the Theory of Games*, Vol. 1. Princeton, NJ: Princeton University Press, pp 97-103.

Mandeles, M. D., Hone, T. C., and Terry, S. S. (1996) *Managing Command and Control in the Persian Gulf War*. Westport, CT: Praeger.

McDonald, J. (1950) *Strategy in Poker, Business and War*. New York, NY: Norton.

Schoonmaker, A. N. (2000) *The Psychology of Poker*. Las Vegas, NV: Two Plus Two Publishing.

Sklansky, D. (1987) *The Theory of Poker*. Las Vegas, NV: Two Plus Two Publishing.

Snook, S. A. (2000) *Friendly Fire: The Accidental Shootdown of U.S. Black Hawks Over Northern Iraq*. Princeton, NJ: Princeton University Press.

Tirpak, J. A. (2002) Seeking a Triple-Threat Sensor. *Journal of the Air Force Association* 85(11), http://www.afa.org/magazine/Nov2002/1102threat.asp

On TRACS: Dealing with a Deck of Double-sided Cards

Kevin Burns
The MITRE Corporation
202 Burlington Road
Bedford, MA 01730-1420 USA
kburns@mitre.org

Abstract- TRACS (Tool for Research on Adaptive Cognitive Strategies) is a new suite of card games played with a special deck, where the back of each card is a clue to the front of the card. This design simulates the clue/truth structure of real world domains like medicine and warfare, where truths (fronts of cards) must be diagnosed from clues (backs of cards) in order to make decisions (cards to choose, chips to bet, etc.). Here I present the cards and rules of TRACS. I also discuss how the games have been used for computational investigations of memory limits and Bayesian inference. The methods for these studies include human experiments and agent simulations, both of which are facilitated by the unique features of TRACS. The products of TRACS research include a computational model of memory limits and a decision support system for Bayesian inference.

1 Introduction

How do people make decisions and how can systems help them make better decisions in the context of real world domains like medicine, business and warfare? This is the question that motivates my research on computational intelligence, with a focus on cognitive strategies.

Previous research on *Judgment and Decision Making* (JDM, see Connolly et al. 2000) has typically pursued computational studies in the lab. Conversely, previous research on *Naturalistic Decision Making* (NDM, see Zsambok and Klein 1997) has typically pursued ecological studies in the field. While each camp strives for both *rigor* and *relevance*, the reality is that JDM has produced mostly rigorous findings that are lacking in ecological relevance while NDM has produced mostly relevant findings that are lacking in computational rigor.

I think that both rigor and relevance are needed for useful research and that *mind games* can help bridge the gap between JDM and NDM. However, to fulfill this promise, the mind games must be both tractable to computational analysis in the lab (for rigor) and prototypical of psychological challenges in the world (for relevance).

This paper presents a new suite of card games called **TRACS**: **T**ool for **R**esearch on **A**daptive **C**ognitive **S**trategies (Burns 2001, Burns 2005). TRACS is unique in that it uses a special deck of double-sided cards to play games that are, compared to standard card games, more tractable to computational analysis and more typical of practical situations. In this paper, I present the TRACS cards and rules and discuss how TRACS games have been used for basic research on mental models and applied research on support systems.

1.1 Card Games

Card games have two features that make them attractive as mind games for research on cognitive strategies, namely they are *flexible* (for experiments) and they are *familiar* (for participants). Flexibility is desirable so that game tasks can be modified as a research program evolves. Card games are especially valuable in this regard since is it easy to design variations on a theme, as evidenced by new games that have been developed for both recreation and research purposes (Gardner 2001, Abbott 1963). Familiarity to a general population is desirable because a domain-specific mind game (e.g., military or business or medical) requires recruiting and/or training of experts with domain-specific knowledge. Moreover, domain-specific research findings often do not generalize outside the domain. The tokens and rules of card games are abstract analogs of many domains (McDonald 1950), and this allows research findings to be applied across domains.

Besides being flexible and familiar, card games also offer advantages of *rigor* and *relevance*. With respect to *rigor*, the cards in a standard deck present a relatively small set of features (e.g., suits and pips) to constrain the possibilities that players must consider as they reason about probabilistic game states in a dynamic context. With respect to *relevance*, card games are played with imperfect information about face down cards, which simulates real world conditions better than board games like checkers and chess that are played with perfect information about the game state. With respect to *both* rigor and relevance, card games are better than 3-D graphic and virtual adventure games for research on human judgment and decision making – because the simple images (cards) and discrete sequences (moves) allow players and researchers to focus more directly on cognitive strategies rather than sensing and motor skills.

1.2 New Games

Along with the advantages noted above, card games also have some disadvantages. With respect to computational *rigor*, the optimal decisions in most card games are not amenable to analytical solution (Epstein 1977, Koller and Pfeffer 1997), except for trivial versions (Kuhn 1950, Nash and Shapely 1950) that are not very *relevant* to practical situations. The combinatorial complexities of most cards games played with a standard deck of 52-cards make it extremely difficult to establish normative performance, which is needed to benchmark cognitive strategies. And, with respect to *relevance*, standard playing cards provide information on only one side of the cards, which does not reflect the basic clue/truth structure of many real world problems like diagnosing a medical disease or military target (truth) from an X-ray image or radar return (clue).

The TRACS cards and rules are designed to overcome both of these limitations found in standard card games. For practical *relevance*, TRACS uses double-sided cards where the backs give clues to the fronts, and TRACS poses diagnostic and decision making challenges that simulate real world dilemmas. For computational *rigor*, TRACS is a suite of games ranging from simple (and tractable) to complex (less tractable), which facilitates progressive research on cognitive strategies.

2 The Cards

Standard playing cards present information on only one side, in the form of shape-color *suits* (Club, Diamond, Heart, Spade) and numerical *pips* (A, 2, 3, …, J, Q, K). The TRACS cards (Figure 1) are different because they present information on both sides to better reflect the clue/truth structure of imperfect information in real world domains. This novel feature of TRACS provides research advantages over standard card games, but also presents a design challenge in helping players to internalize the structure of the double-sided (unfamiliar) deck.

The design of the deck is based on the notion of *tracks* and *treads*, where the back of each card is a track that gives a clue to the tread on the front. The analogy is that of a track (shape) left by the tread of a shoe or a tire. Each tread is set of shapes and there are two treads in TRACS: a Red tread (Figure 1, upper) and a Blue tread (Figure 1, lower).

Each track, on the back of a card, is a single black shape (triangle, circle or square) that gives a clue to the tread on the front of the card. The clue comes from the structure of the deck, in which there are different numbers of each track (shape) in each tread (set), as shown in Figure 1 and as illustrated by the Red and Blue sets of shapes in the center on the front of the cards.

2.1 Rationale

Why are there only two treads (Red and Blue)? Two treads are used to keep the games as simple possible but still interesting. If there were only one tread the player would have nothing to diagnose.

Why are there three tracks? Again, it is to keep the games as simple as possible but still interesting. Three is the minimum number of track types needed to capture the basic difference between what is *likely*, *unlikely* and *ambiguous* (50-50) in probabilistic diagnoses.

How is this structure relevant to real world domains? A fundamental dilemma in virtually every domain is to diagnose the likely truth from a given clue. For example, in medical diagnosis one must infer the most likely state of a tissue (healthy or diseased?) from the clue given by an X-ray image. Similarly, in military intelligence one must infer the most likely identity of a possible target (friendly or enemy?) from the clue given by a radar return. TRACS *cards* reflect the essential structure of this task, because players must infer the likely truths (treads) from clues (tracks). TRACS *rules* reflect the relevant context in which people must deal with clues and truths in the real world, which is both probabilistic and dynamic.

Along with this basic clue/truth structure, a related feature of the TRACS deck is that it contains multiple copies of each track/tread card type, i.e., either 2, 4 or 6 of each card type (see Figure 1). This design has both theoretical advantages and practical advantages over standard playing cards. A theoretical advantage is that TRACS can be used to study the memory processes that people use to count “carbon copies” (multiple instances), as opposed to unique objects like the cards in a standard deck. A practical advantage is that the deck can be scaled (halved, doubled, sampled, etc.) to change the number and/or distribution of card types. For example, the deck used in solitaire games can be doubled for a two-player game to preserve the number of cards per player.

2.2 Example

As an example of the basic problem that arises in TRACS (and the real world), consider a card that is dealt from a full deck with tread down and track up. Assume this card shows a triangle track.

Based on the distribution of track/tread cards in a full deck (Figure 1), the Red:Blue odds for this triangle are 6:2 [P(Red)=6/8=75%]. Similarly, the Red:Blue odds for a square track are 2:6 [P(Red)=2/8=25%]. Thus, when faced with a decision like “choose the Red card”, you would do better to pick a triangle track than a square track.

Now consider a situation where some triangle tracks have been turned over in play but then removed from play and taken out of sight. How well can you remember which cards you have seen so you can update the track/tread odds for a triangle track that is dealt later in the game?

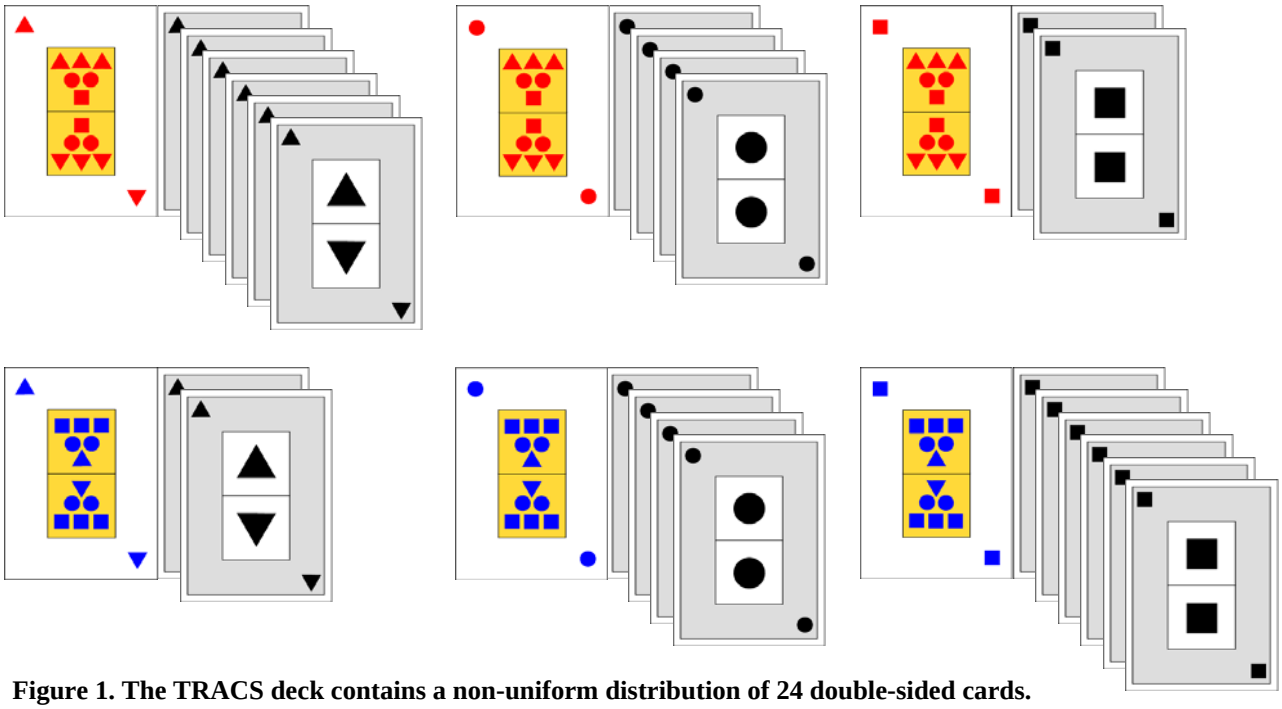


Figure 1. The TRACS deck contains a non-uniform distribution of 24 double-sided cards.

Next consider a situation where you get additional information from a *spy* who, like all spies, is of limited reliability. For example, assume a triangle track is dealt at the start of the game and the spy says, "That track is Blue". Also assume that the reliability of the spy is known (e.g., 90% correct in reporting Red or Blue). How do you use this *likelihood* information along with your *prior* knowledge of the Red and Blue probabilities (based on the deck distribution) to estimate the *posterior* (after spy) probability that the triangle track is Red?

Finally, consider a situation where a triangle track appears in the hand of an opponent, along with two square tracks. Also assume that the game is a 3-card poker where the best hands are flushes (all three Red cards or all three Blue cards) and that your opponent, who holds the triangle and two squares, has just made a big bet. Based on the distribution of cards in the deck, his two squares are probably Blue and his one triangle is probably Red. But he just made a big bet and this behavior provides some evidence to support the hypothesis that his cards are either all Red or all Blue. How well can you fuse the evidence from his tracks and bets to make inferences about what he has in his hand, so you can make the best choice about whether to call his bet, raise his bet or fold your hand?

In fact, these three examples (above) highlight the cognitive challenges of three TRACS games, called *Straight TRACS*, *Spy TRACS* and *Poker TRACS*, respectively. In the following sections I discuss these and other TRACS games.

3 The Games

TRACS is a suite of games that can be played with real cards or online (Burns 2001, Burns 2005). The online games are programmed in Java, and computer versions have also been written in MATLAB for agent simulations and human experiments.

The TRACS deck can be used to play many games, both solitaire and multiplayer, only some of which have been programmed in computer versions. One class of online games, the ***TRACS Arcade,*** includes three solitaire games called *Straight TRACS*, *Wild TRACS* and *Booty TRACS*. *Straight TRACS* is a game of forced choice on each turn, played in a series of turns. *Wild TRACS* gives the player some options on each turn, and *Booty TRACS* gives the player even more options on each turn.

The other class of online games, in the ***TRACS Casino***, includes gambling games for one or more players. These games are similar to familiar casino games but with various twists that are made possible by the unique design of TRACS. The games include *Slot TRACS* (like a slot machine), *Black TRACS* (like Blackjack) and *Poker TRACS*.

Below I discuss the games in the *TRACS Arcade* with a focus on *Straight TRACS*, which is the simplest game and which has been used to perform normative analyses and cognitive experiments. I also discuss the games in the *TRACS Casino* with a focus on *Poker TRACS*, which is the most complex game and is therefore the most computationally interesting. The details of all games, as well as online play, are available at www.tracsgame.com and http://mentalmodels.mitre.org.

4 TRACS Arcade

The *TRACS Arcade* is a suite of solitaire games. The simplest game, called ***Straight TRACS***, is a matching game played in a series of turns as the deck is depleted.

Figure 2 illustrates a typical turn. At the start of the turn, three cards are dealt from the deck (on left) to a *field* (on right) as one *tread* flanked by two *tracks* (Figure 2a). The diagnostic challenge is to judge the likely color of each track. The decision challenge is to choose the track, either left or right, that is most likely to match the color of the tread in the middle. The chosen track is turned (Figure 2b) and the turn is scored as a *save* (if colors match) or a *strike* (if colors mismatch, as in Figure 2b). The pair of treads (save or strike) is then removed from play and stored in a stack (tread down, not shown in Figure 2), using one stack for saves and one for strikes.

The remaining track on the field is turned to reveal its tread (Figure 2c) and this tread is moved to the center of the field where it becomes the color to match on the next turn (Figure 2d). Two more tracks are dealt from the deck to the field (left and right) and the sequence continues until all cards have been paired as either a match (save) or mismatch (strike). The last pair does not count because the player has no choice.

The object of the game is to make saves (matches) and avoid strikes (mismatches). The challenge of the game is to count the cards and update odds as the deck is depleted, in order to make the best choice on each turn.

Straight TRACS is a good game for learning TRACS, but it is not too fun for participants and not too deep for experiments. This is because every turn is a forced choice (left or right) with a fixed outcome (strike or save), whereas choices in real life often involve several options and scalable outcomes. Thus, the *TRACS Arcade* also includes two variants of *Straight TRACS* that are designed to capture these features of real world decisions.

In one game, called ***Wild TRACS***, the player gets four wild cards that can be used as tickets to exercise options. Each wild card can be played in one of two ways, either as a *spare* (which is like passing a turn) or a *dare* (which is like betting double-or-nothing). Each wild card can be played only once, as either a spare or a dare, and the player's challenge is to optimize use of this limited resource and finish a trip through the deck with no strikes. In another game, called ***Booty TRACS***, every *save* (match) becomes *booty* that the player can use to bet double-or-nothing on later turns. In this way, a player must bet booty to get booty, and the player can scale the stakes (booty) beyond the one point of a save or strike in *Straight TRACS* and the additional point of a dare in *Wild TRACS*. The object is to get the most booty on a trip through the deck.

The simplest game of Straight TRACS has been used to perform human experiments and agent simulations, and the findings are discussed in Section 6. Section 5 (below) outlines other games in the TRACS Casino.

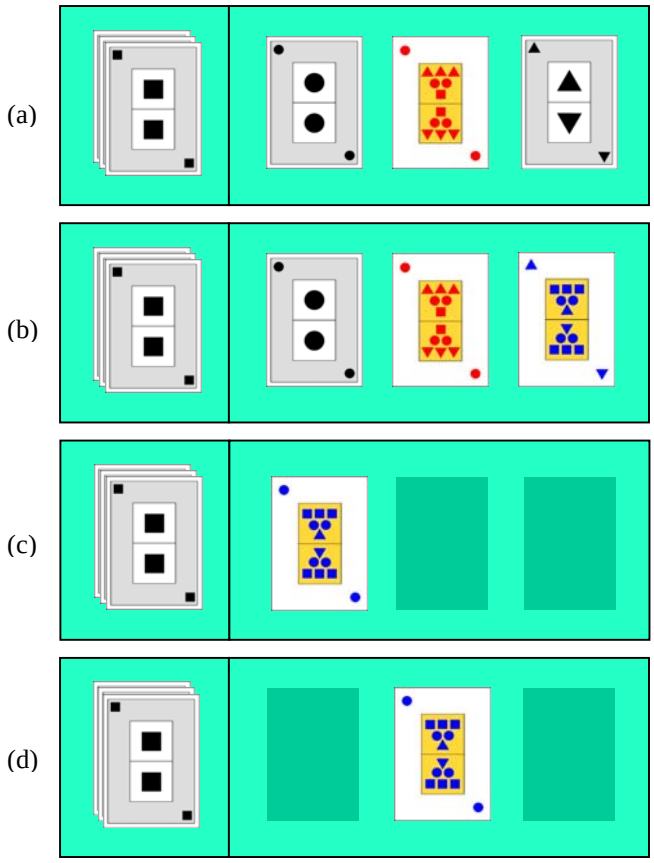


Figure 2. A turn in *Straight TRACS*. This turn is scored a *strike* (colors mismatch).

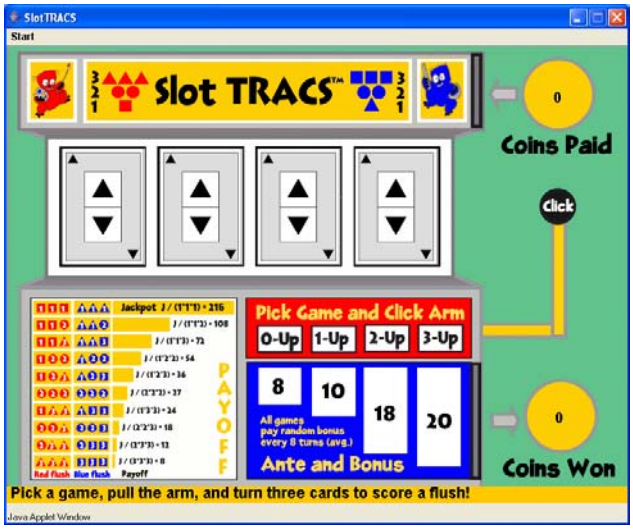


Figure 3. Screen shot of *Slot TRACS*.

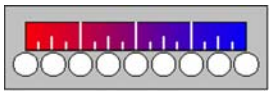


Figure 4. Colored ruler used as a probe in human testing on *Straight TRACS*.

5 TRACS Casino

The *TRACS Casino* is a suite of gambling games, similar to familiar games but with a TRACS twist. ***Slot TRACS*** is the simplest game, and is similar to slot machines except the player must make a few decisions other than whether or not he wants to pull the arm, which is the only decision in most slot machines. *Slot TRACS* (Figure 3) has four windows (slots), and behind each is a simulated wheel with a full TRACS deck (see Figure 1). The wheels are spun and when they stop a single card appears behind each window. The player must select three cards in three of the four windows, trying to make a flush of all three Red or all three Blue. The selected cards are clicked to turn them face up and show their colors. The player gets a payoff if he makes a flush, and the payoff gets higher as the flush gets rarer, where rareness is determined by the distribution of cards in the deck. For example, a flush of three Red squares or three Blue triangles is the most rare and so it pays the Jackpot. Other flushes pay less in proportion to their rareness.

Slot TRACS is interesting because the player must make two kinds of decisions. First, he must decide how many cards he wants to be spun face up: 0, 1, 2 or 3. The ante (cost to play) increases as more cards are spun face up and the player must balance the cost of this extra information with its benefit in making rare flushes.

The second decision in playing *Slot TRACS* is which track(s) to turn face up after the spin. This choice is interesting from a cognitive perspective because it involves a tradeoff between a high chance of getting a low payoff versus a low chance of getting a high payoff. The payoff schedule in *Slot TRACS* is purposely designed such that all choices are normatively equivalent, yet players tend to exhibit individual preferences that reflect risk seeking or risk averse tendencies.

Black TRACS is another game in the *TRACS Casino*, and it is similar to Blackjack but with a twist. Like Blackjack, *Black TRACS* is played against a dealer who must make his choices according to a pre-set strategy while the player can adjust his strategy to account for changes in the number of each card type remaining in the deck. Unlike Blackjack, it is harder to keep an effective count in *Black TRACS* because the player must count six types of track/tread cards at once. In the standard card-counting strategy for Blackjack (Lewis 2002), the player need only count only one thing, namely the net number of high cards that have been seen (i.e., high cards seen minus low cards seen).

Poker TRACS is the most complex and challenging game in the *TRACS Casino*. Unlike standard poker, a hand is made of three cards (not five), and the ranking system is the same as the payoff structure of *Slot TRACS* (see above). That is, a flush (three cards of the same color) ranks higher than a non-flush, and rarer flushes are better, where rareness is determined by the number of each track/tread card type in the full deck.

More formally, the possible flushes are ranked by their conditional probability P(flush|tracks) in the full deck, and the ranks are computed as follows: First, each card is assigned a number 1, 2 or 3 proportional to the number of its track (shape) in the tread (set) as illustrated by the two tread designs (Red and Blue) on the fronts of the cards. That is: Red triangle = 3, Red circle = 2, Red-square = 1; Blue square = 3, Blue circle = 2, Blue triangle = 1. Then, for a flush of 3 cards {X, Y, Z}, all Red or Blue, the numbers are multiplied to get N=X*Y*Z. The 10 possible N are {1, 2, 3, 4, 6, 8, 9, 12, 18, or 27}, where smaller N are rarer. [N=1 is possible only with a double deck.]

This ranking system preserves the basic structure of standard poker but eliminates the need to memorize a ranking schedule (like Flush beats Straight) and reduces the number of possible hand strengths by orders of magnitude. That is, while standard poker also has 10 ranks based on rareness (i.e., Royal Flush, Straight Flush, Four-of-a-Kind, Full House, Flush, Straight, Three-of-a-Kind, Two Pair, One Pair, No Pair), it makes further distinctions within each rank (e.g., King beats Queen even though each is equally rare). *Poker TRACS* reduces the number of possibilities to 10 flushes (plus non-flushes), and the flush ranks makes it easy for players to compute the odds of hands that their opponent may hold.

For example, in standard poker (with 5-card hands dealt from a 52-card deck), the odds that your opponent was dealt a Straight versus a Flush are about 2:1 (Straight:Flush). But this is a complex calculation that poker players cannot do in their heads, and many players do not even know that the approximate answer is 2:1. In *Poker TRACS*, you can easily estimate the odds of possible hands that your opponent was dealt because these odds are given by the tracks in his hand and the ranking system (above). For example, if your opponent's tracks are {triangle, square, square}, then either he holds a Red flush of N=3*1*1=3, or a Blue flush of N=1*3*3=9, or a non-flush. Furthermore, it is approximately three times less likely (3:9) that he holds the Red flush with N=3 than the Blue flush with N=9.

I say "approximately" because the ranking structure reflects the baseline (full deck) odds and the actual odds of flushes in opponents' hands change for two reasons. First, the fronts of some cards may be observed in play (i.e., in your own hand). For example, if you hold two Red squares, and there are only two in the whole deck, then any squares in your opponent's hand must be Blue. Second, you get additional information about your opponent's hand from his bets and raises. In the above example, if your opponent makes a big bet or raise then he probably has the stronger flush rather than the weaker flush or a non-flush, unless he is bluffing. And, of course, this is the essence of poker (and business and warfare, see McDonald 1950), in which players must use imperfect information from cards and bets along with models of their adversaries to make inferences about one's chances of winning and to make investments of one's chips to the pot.

6 Human Testing

Agent-based simulations have been performed in order to establish normative strategies in *Straight TRACS* and other solitaire games, and in order to develop cognitive surrogates (robot players) for multiplayer games like *Poker TRACS*.

Human-based experiments have been performed on the simplest game of *Straight TRACS* and a variant called *Spy TRACS* in order to measure cognitive performance against normative standards. The methods and findings from these experiments are reviewed below, to show how TRACS can be used to develop valuable insight into how well people reason about probabilistic information in dynamic situations.

6.1 Experiment 1

Experiment 1 was designed for two purposes. One purpose was to test how well people could count cards and update odds in *Straight TRACS*. Another purpose was to develop a computer model of cognitive limits.

Straight TRACS requires probabilistic estimation of track/tread odds under dynamic conditions as the deck is depleted, where the key skills are basic memory (like card counting in Blackjack) and revising probabilities. The hard part is that, in TRACS, the player must keep a count of all six track/tread cards types (see Figure 1) in order to update the odds. For example, the Red:Blue odds for circle tracks start out at 4:4, and to update the odds after some circles have been seen the player must know both how many Red circles have been seen and how many Blue circles have been seen.

The probe for Experiment 1 was a colored ruler with buttons (Figure 4), which appeared on the computer screen below each track. The ruler ranges from 100% Red on one end, to 50-50 in the middle, to 100% Blue on the other end. The player's task was to click the button that matched the chances (player judgment) that the track will turn out Red or Blue. The experiment also measured which track the player chose to match the color of the tread in the middle (see Figure 2). Here I focus on the data for players' judgments (of % Red or Blue) rather than on players' choices (of track to turn). Further details on both judgments and choices in the experiment are provided elsewhere (Burns 2002, 2003).

The major finding from Experiment 1 is that people are extremely limited in their ability to count cards and update odds. Approximately 100 participants were tested, each playing 10-20 games of *Straight TRACS*, where each game involved 11 turns. All participants played several practice games beforehand, and some participants played many practice games. The cognitive performance of participants was seen to exhibit a pattern of *anchoring and adjustment* (Burns 2002, 2003), in which players were anchored to the baseline (full deck) odds and made only minor adjustments thereto – typically much less than the optimal adjustments – as a result of memory limitations.

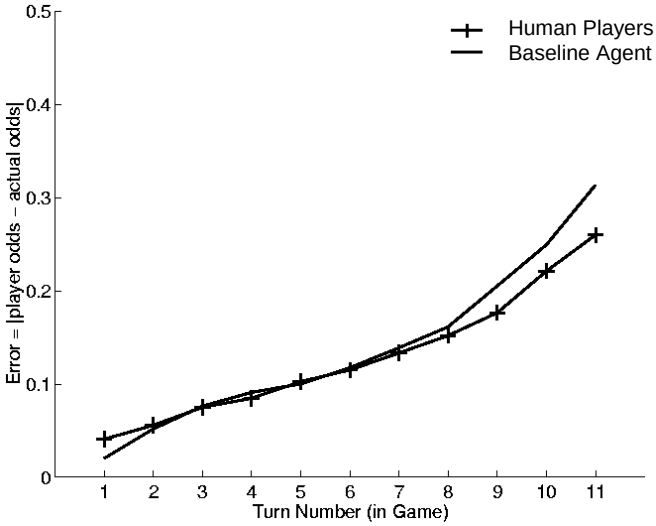


Figure 5. Error versus turn in *Straight TRACS*, showing a baseline bias.

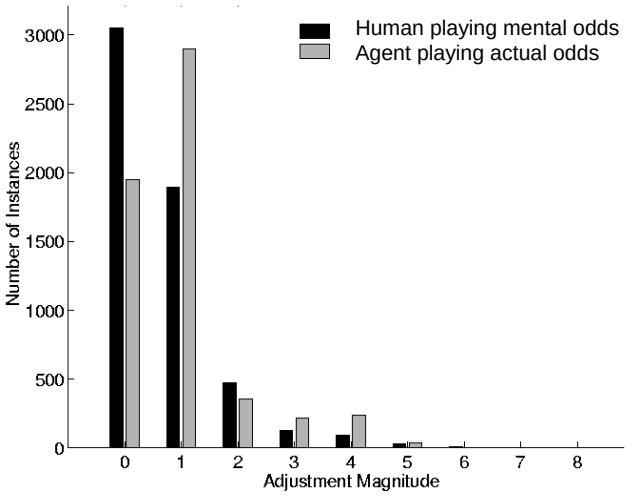


Figure 6. Turn-to-turn adjustments in odds, showing anchoring (adjustment magnitude 0).

Figure 5 shows the average performance for human players and for a simulated baseline agent, who is a player that never updates odds. A perfect player who correctly updates odds at each turn would have zero error always. This figure illustrates the *baseline bias* that people exhibit in playing *Straight TRACS*, i.e., the average error increases as the deck is depleted in play similar to the behavior of a baseline agent.

Figure 6 shows the number of turn-to-turn adjustments in odds made by human players and perfect agents. Here the magnitude of adjustment is measured as a span of buttons on the colored ruler (see Figure 4). Figure 6 illustrates *anchoring* behavior where human players make too many *non-adjustments* (adjustment magnitude zero).

Besides these results, I also measured the hits and misses made by players in counting all cards of a certain class, where the six card types (Figure 1) are grouped into three classes (2-count, 4-count, 6-count) depending on how many of that card type there are in the deck. A hit occurred when all of the Red or all of the Blue cards of a given track type (triangle, circle or square) had been turned over in play and the player correctly clicked the rightmost (Blue) or leftmost (Red) button on the colored ruler. A miss occurred when all of the Red or all of the Blue cards of a given track type had been turned over and the player did not click the rightmost (Blue) or leftmost (Red) button. The results show a hit rate of ~50% for counting up to two of each card type (while counting all six types at once), but the hit rate for counting up to four or six was only ~10%.

6.2 Cognitive Model

Based on these findings, I developed a computational model to explain and predict human memory in playing *Straight TRACS* (Burns 2003). The model employs *fuzzy functions*, which are like fuzzy logic in that they specify a fuzzy mapping from a physical event (turning a card) to a cognitive belief (update of odds). The model is based on an *accumulator analogy* where discrete bins are filled in a stepwise fashion and the filling of one bin after a lower bin is governed by a fuzzy function (leaky filling).

Figure 8 shows how well the computer model compares to the cognitive data for all three track types (triangle, circle, square) in one game. The black plots the mean (line) and standard deviation (bars) of all data from humans. The gray plots the mean and standard deviation for the model, which exhibits variability due to its fuzzy functions. The dotted line plots the performance of a perfect agent who counts cards and updates odds with no error. The comparison shows that the model (gray) does a good job of capturing the mean and spread of cognitive performance (black), especially relative to normative performance (dotted). Results were similar for other games as reported elsewhere (Burns 2003).

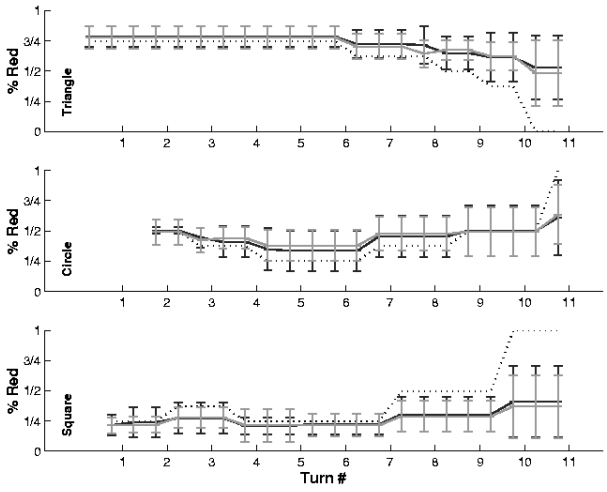


Figure 7. Model (gray) versus data (black).

These experimental findings and the computational model demonstrate how TRACS can be used to gain insight into human thinking. Both descriptive data and predictive models are needed to understand the strengths and limits of cognitive competence, and Experiment 1 shows that TRACS can be used as a test bed for getting such data and building such models.

While memory limitations are important in many practical applications, computer systems that can support such limits are straightforward (e.g., a card counter in *Straight TRACS*). Thus, further experiments (discussed below) were designed and performed to study another task where cognitive competence is also limited but where a support system is not so obvious.

Experiment 2 used a game called *Spy TRACS* to study *Bayesian inference*, which is a critical and challenging task that arises in many real world domains. Like *Straight TRACS*, the basic problem in *Spy TRACS* is to update odds in light of additional information. But unlike *Straight TRACS*, the additional information in *Spy TRACS* comes from a "teammate" (spy) – who like all teammates is of limited reliability

7 System Design

To test cognitive competence in a task of Bayesian inference, *Spy TRACS* is played like *Straight TRACS* but with two twists. One twist is that the player does not have to count cards because he is given the *deck odds* based on the current card count by the computer. The other twist is that the player is also given a spy report with associated reliability for each track. The player's task is to combine the deck odds with the spy odds to get the *fused odds* in order to make the best choice of a track (left or right) on each turn. A colored ruler like that of Figure 4, but with numerical % instead of buttons, was used for presenting odds (deck and spy) to players and for measuring odds (fused) from players.

The results (Burns in press) confirmed previous findings of cognitive conservatism (Edwards 1982) in Bayesian inference. That is, people extracted far less certainty than they should have from the data they were given. For example, given a deck probability of 80% Red and a spy probability of 67% Red, most people reported a fused probability of less than 80% Red, and some people reported <67% Red. The Bayesian answer is actually 89% Red.

The findings from *Spy TRACS* were used to build and test a support system that could help people in tasks of Bayesian inference. The support system, called *Bayesian Boxes* (Burns in press, 2004) works like a colored calculator (see Figure 8) where the user dials in a *prior* (deck odds, at bottom) and a *likelihood* (spy odds, on the left) and reads off the *posterior* (fused odds, at the top). Further experiments were performed with *Spy TRACS* to evaluate the benefits of *Bayesian Boxes*.

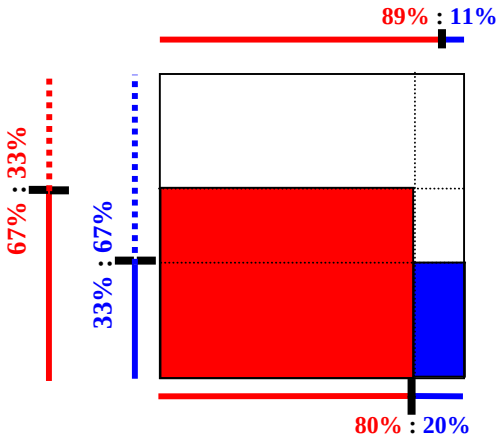


Figure 8. Screen shot from a colored calculator called *Bayesian Boxes*.

Here the intent was not to see if people could dial numbers in and read numbers off the colored calculator. Rather, the intent was to see if practice with *Bayesian Boxes* improved people's intuitive understanding of Bayesian inference. So, the experimental procedure was as follows.

After several games of *Spy TRACS* to get the *before* data (which motivated the development of *Bayesian Boxes*, see above), players were given the colored calculator along with a few sample problems to solve with it (along the lines of the problems they had faced in *Spy TRACS*). After about 5 minutes of practice, the colored calculator was taken away and players were tested on several more games of *Spy TRACS* to collect *after* data. The *before* data was compared to the *after* data and the results showed a significant increase (about doubling) in the fraction of Bayesian responses (Burns in press). Encouraged by these results, *Bayesian Boxes* has been further enhanced and applied to real world problems of Bayesian inference like the forensic assessment of disputed authorship (Burns 2004).

8 Conclusion

TRACS is a suite of games played with a special deck of double-sided cards. In this paper I discussed how TRACS can provide a unique blend of rigor and relevance for research on cognitive strategies. I also reviewed how TRACS has been used to perform cognitive experiments and computer simulations, and how TRACS can be used to develop designs for support systems in practical applications.

Acknowledgments

This research was supported by the MITRE Technology Program. Thanks to the many MITRE volunteers who participated in TRACS experiments.

Bibliography

Abbott, R. (1963) Eleusis. In Abbott, R. *Abbott's New Card Games.* New York, NY: Funk & Wagnals, pp. 73-92.

Burns, K. (in press) Dealing with Probabilities: On Improving Inferences with Bayesian Boxes. In Hoffman, R., (Ed.) *Expertise Out of Context*. Mahwah, NJ: Lawrence Erlbaum.

Burns, K. (2005) *Mental Models in Naturalistic Decision Making*, http://mentalmodels.mitre.org.

Burns, K. (2004) A Bayesian Approach to Detecting a Hoax. *Proceedings of the International Conference on Knowledge Engineering and Decision Support*, Porto, Portugal, July 21-23, pp. 305-312.

Burns, K. (2003) 1, 2, 3, More: An Accumulator Architecture for Anchoring and Adjustment. *Proceedings of the 7th Joint Conference on Information Sciences*, Cary, NC, September 26-30, pp 1673-1676.

Burns, K. (2002) On Straight TRACS: A Baseline Bias from Mental Models. *Proceedings of the 24th Conference of the Cognitive Science Society*, pp. 154-159.

Burns, K. (2001) *TRACS: A Tool for Research on Adaptive Cognitive Strategies*, www.tracsgame.com.

Connolly, T., Arkes, H., and Hammond, K. (2000) *Judgment and Decision Making: An Interdisciplinary Reader*. Cambridge, UK: Cambridge University Press.

Edwards, W. (1982) Conservatism in Human Information Processing. In Kahenman, D., Slovic, P., and Tversky, A. (Eds) *Judgment under Uncertainty: Heuristics and Biases.* New York, NY: Cambridge University Press, pp. 359-369.

Epstein, R. A. (1977) *The Theory of Gambling and Statistical Logic*. San Diego, CA: Academic Press.

Gardner, M. (2001) The New Eleusis. In Gardner, M., *The Colossal Book of Mathematics*. New York, NY: Norton, pp. 504-514.

Koller, D., and Pfeffer, A. (1997). Representations and Solutions for Game-Theoretic Problems. *Artificial Intelligence* 94:167-215.

Kuhn, H. W. (1950) A Simplified Two-Person Poker. *Contributions to the Theory of Games*, Vol. 1. Princeton, NJ: Princeton University Press, pp 97-103.

Lewis, K. (2002) How to Count Cards and Beat Vegas. In Mezrich, B., *Bringing Down the House: The Inside Story of Six MIT Students Who Took Vegas for Millions*. New York, NY: Free Press, pp 252-257.

McDonald, J. (1950) *Strategy in Poker, Business and War*. New York, NY: Norton.

Nash, J. F., and Shapley, L. S. (1950) A Simple Three-Person Poker Game. *Contributions to the Theory of Games*, Vol. 1. Princeton, NJ: Princeton University Press, pp 105-116.

Zsambok, C. E., and Klein, G., (1997) *Naturalistic Decision Making*. Mahwah, NJ: Lawrence Erlbaum.

A Study of Machine Learning Methods using the Game of Fox and Geese

Kenneth J. Chisholm & Donald Fleming

School of Computing,
Napier University,
10 Colinton Road,
Edinburgh EH10 5DT.
Scotland, U.K.
k.chisholm@napier.ac.uk

Abstract: The game Fox and Geese is solved using retrograde analysis. A neural network trained using a co-evolutionary genetic algorithm with the help of the expert knowledge database was found to be a very capable Fox and Geese player after training, and quickly learned to beat training opponents.

Key-Words: Game theory, rote-learning, neural networks, genetic algorithms, co-evolution.

1 Introduction

1.1 The Game of Fox and Geese

Fox and Geese is a derivative of draughts (checkers) and is played on a standard 8 by 8 draughts or chess board. The black player has four pieces (the Geese) which are initially placed on the four dark squares at the top of the board. The white player has a single piece (the Fox), which is normally placed either at the bottom of the board, on the second dark square from the left (figure 1) or on any free dark square on the board, chosen by the white player.

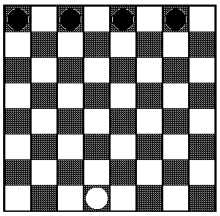


Fig. 1. The standard starting position for Fox and Geese

Black pieces (Geese) can move one square, but can only move down the board, so their options are limited to the 2 squares diagonally ahead. White's Fox piece can move diagonally one square in *any* direction. There is no taking or jumping in Fox and Geese, so an occupied square is blocked to both players.

The object of Fox and Geese, for the Fox, is to break past the line of Geese and reach one of the four dark squares at the top of the board, where the Geese are initially placed. The aim for the Geese is to hem the Fox in so that it can no longer make a legal move (There are no drawn games in Fox and Geese. If the Fox fails to break through the line of Geese, it will eventually be pinned to the bottom of the board, and will lose the game (see Perham, 1998 or Berlekamp, Conway and Guy, 1982 for full details [2][9]).

2 A Simple Rote-Learning Player

A rote-learning algorithm, closely based on the technique used by Samuel [11], was used to improve the play of a basic AI Fox and Geese program. Previously encountered board positions are recalled from a database of moves in order to increase the look-ahead ability of a mini-max search tree.

Samuel's rote-learning method can quite easily be adapted for the game Fox and Geese. As the rules of Fox and Geese are relatively similar to draughts, and the playing board is the same, the AI mini-max algorithm is easily adapted to this new implementation. The only part of the AI programming that needs to be really specific to Fox and Geese is the design of the board evaluation function [16].

2.1 A Board Evaluation Function for Fox and Geese

For this implementation a range of values between -100 and 100 was chosen, thus allowing a score to be efficiently stored in one byte of memory. A score of -100 represents an overwhelming advantage for the Geese, and a score of 100 means the same for the Fox.

As the AI algorithm would eventually be supplanted to some extent by the accumulated knowledge from previous games, there was no need to create a highly complex board evaluation function which can perform as well as a master player. The design of the evaluation function reflects these simple needs, and values are based on only a few features of the board. First, the board evaluation function checks for a winning position and returns either 100 or –100 if it finds one. Otherwise, the score is calculated by setting an initial value of -50 and then adding two points for every dark square reachable by the white piece. To this total a bonus of two points is added for every row the Fox has advanced from the bottom of the board.

The other important feature in the evaluation function, that of increasing the score as the Fox advances up the board, is included to encourage the Fox to move forward as much as possible.

A simple mini-max AI player using this board evaluation function with a search limit of 6 ply can play a fairly competent game of Fox and Geese. The algorithm does, however, perform better for the Geese than it does for the Fox. When the AI program takes both players' roles, the Geese usually win, although at lower search depths (below 6 ply) the advantage the Geese have is diminished considerably, and the Fox can sometimes win. The Geese can also defeat most human opponents, whereas a fairly competent human player can easily beat an AI Fox. The temptation to tinker further with the evaluation function (and improve the Fox's performance) has been resisted as it is felt to be adequate for the machine learning experiments to come.

3 A Complete Solution of Fox and Geese

A simple form of retrograde analysis was used to construct a database of valid boards and moves for Fox and Geese [7][13]. First, in a relatively straightforward manner, all possible boards were simulated and stored. These moves were then edited down to two smaller databases consisting of boards with legal white moves and legal black moves.

3.1 Assigning Values to the Expert Database

Having now collected and ordered all playing positions in Fox and Geese, it only remained to assign a game-theoretic value to each position. A simple implementation, similar to the retrograde analysis that employed by Schaeffer et al. for the *Chinook* checkers program [7][13], was used for this purpose. In this case however, a *forward* moving analysis is found to be most suitable and each position is resolved by calculating its successors. Initially a first pass through the databases resolves all known winning positions (those where the Fox is either trapped, or has achieved the top row of the board). Subsequent iterations through the database (from each legal board position) simulate all succeeding moves from any positions which as yet have unknown values.

From these studies it was thus determined that the value for the game Fox and Geese is a ***win*** for the ***Geese*** and the game is thus strongly solved, as defined by Allis [1]. This significant result is believed by the authors to be the first such complete solution of the game Fox and Geese.

4 A Neural Network Player: F&G-NN/BP

An artificial neural network was trained using back propagation to perform the role of a board evaluation function in a standard mini-max search algorithm. The database of perfect moves provided expert knowledge for training.

Each time the neural network is called upon to give a value to a board at the leaves of a search tree, the raw output of the network (between 0 and 1) is compared to the value found in the perfect moves database for the relevant board. The result from the database is allotted a value of 1 for Fox wins and 0 for Geese wins, so that this result can be directly compared with the network's output. If the difference between the network output and the database value falls outside of a tolerance of ±0.1, the networks weights are updated by back propagation using the network output and database value as actual and target values. The learning rate (α) is set to 0.5, and no momentum or other optimisation measures are used.

4.1 F&G-NN/BP Experiments

The neural network was trained by playing 200 games against the simple mini-max AI opponent using a look-ahead of 6 ply. For each game the winner is recorded, as is the proportion of plays made by the network player which are valued as an eventual win in the perfect moves database. Also recorded are the number of times back propagation is used to adjust the network weights during each game and the average error for each game. The error for each network activation is expressed as the difference between the target and actual network output (a value between 1 and 0). The average error is simply the total network error for one game divided by the total number of network activations.

This experiment, and all other experiments, were conducted 10 times in order to reduce the potential for misleading results caused by the random nature of network weights initialisation. The roles of the AI player and the neural network were then switched and the experiments were performed a further 10 times.

4.2 F&G-NN/BP Results

Results show that the neural network player was able to quickly supplant the simple mini-max AI player. When playing as both the Fox and the Geese, the neural network is able to beat the simple AI player within a few games of training commencing. After the first ten games the neural network was able to win more than half as either player.

The neural network plays better as the Fox. The average number of wins (out of ten) starts above eight, and varies throughout training between eight and ten. By the end of training the neural network playing as the fox wins almost every game against the opposing AI player.

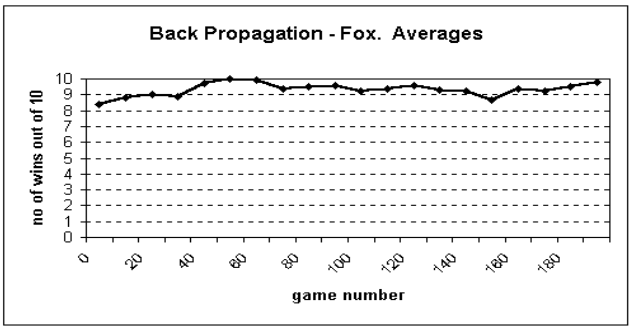


Fig. 2. Back propagation. Average no. of wins per 10 games for *Fox*. Number of wins (out of 10) for every 10 games of the 200 game training run. (This graph represents the average from the 10 runs.)

The neural network playing the Geese performs less strongly, only winning an average of around six of its first ten games. After the initial ten games performance

fluctuates, but the last sixty games show a steady improvement towards an average of more than eight wins out of ten.

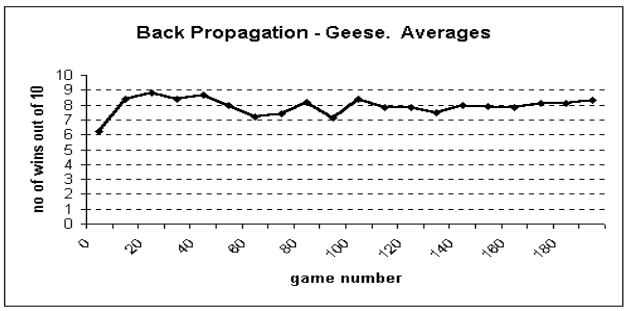


Fig. 3. Back propagation. Average no. of wins per 10 games for *Geese*. (This graph represents the average from 10 training runs.)

The slightly erratic nature of the learning process expressed in terms of game wins and losses is in contrast to the results gained from testing the average network output error for each game. Here, results show a smooth decrease towards an extremely low error rate.

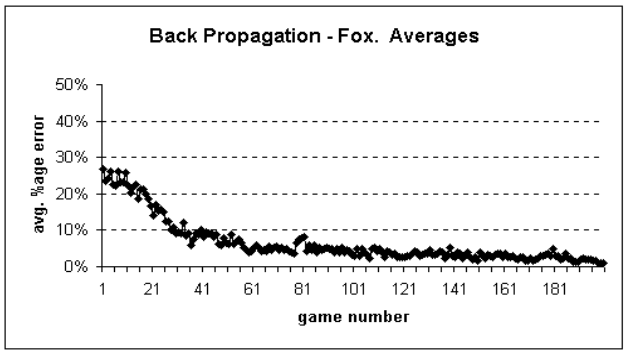


Fig. 4. Back propagation. Average percentage error for *Fox*. Percentage error is calculated by summing the errors from the network (target output – actual output) and dividing the result by the total number of network activations**.** (A similar graph is obtained for the Geese.)

This disparity is probably caused by the neural network's non-static evaluation of board positions at different points in a mini-max search tree. Although the accuracy of the network outputs is steadily increasing, the adjustment of the network during mini-max searches leads to erratic search results. Once the fluctuation of the network weights has reduced to a background level, the game-playing performance of the neural network player stabilises towards more consistent results.

5 Machine Learning using GAs

There recently been a number of board game implementations using genetic algorithms in some way [6]. Chisholm and Bradbeer have used a genetic algorithm to control and optimise the board evaluation function of a draughts program [5]. The algorithm uses crossover and selection to develop optimal weights for board evaluation. Each weight represents the importance assigned to a feature on the board, such as number of pieces left, number of kings on the board [5].

Some researchers such as Carling have used genetic algorithms to train neural networks to play board games[3]. Richards, Moriarty, McQuesten and Mükkulainen have experimented with this approach [10]. Rather than evolving whole networks by adapting the weights between nodes, Richards et al. have developed a system where promising neurones are bred and combined in new networks every generation. Their system, which they have called SANE, has been applied to the task of playing Go with some considerable success.

An alternative solution to this problem of training neural networks is described in Chellapilla and Fogel [4]. Here, instead of using a reinforcement method such as back propagation or temporal difference learning, Chellapilla and Fogel create a set of network weights by co-evolution [4].

6 A Neural Net/GA Player: F&G-NN/GA

A genetic algorithm was used to breed a set of optimal weights for an artificial neural network. The neural network comprises a standard multi-layer feed-forward network consisting of 33 input units (32 concerning the state of each square on the board and one for who is to move), a hidden layer of 20 units and a single output node. A pool of ten randomly created network players competes against each other in a tournament using a co-evolutionary strategy. Poorer players are replaced with offspring bred from two successful players, as well as new randomly initialised players being introduced.

6.1 Implementation

The genetic algorithm used in this implementation has the task of breeding optimal weights for the artificial neural network. The network performs the role of a board evaluation function, used by a mini-max search tree to give values to board positions encountered during a search [8]-[12].

A binary system of encoding was not used in this system. Instead the network weights themselves each form a gene of the GA chromosome. The chromosome or string for this implementation is simply the entire collection of network weights. New offspring are bred from two parent sets of weights. Crossover points are chosen randomly at intervals anywhere between one and five weights along a string. This method of crossover provides the reason that weights are grouped into sub-strings by the network nodes that they feed into, rather than the nodes they emanate from. As the output layer of the neural network consists of only a single node, strings formed by the weights between the 20 node output layer and the hidden layer are grouped into one string of 20 weights, rather than 20 strings of one weight each.

During crossover, for each weight the offspring receives from a parent, there is a 0.1% chance of mutation. Mutated weights are created by randomly re-initialising the weight to a random value between –1 and +1.

6.2 The F&G-NN/GA Fitness Function

The solution chosen here is to use the perfect moves database (see section 3) to supply the level of fitness based on the number of winning moves the players make during the course of a game.

A pool of ten players is used, randomly initialised having each of the network weights set to a random value between –1 and +1. Every generation, each player plays one game against an AI mini-max opponent. The total fitness of each player is calculated by counting the number of correct moves (moves which will *definitely* lead to an eventual win), the player makes. The initial fitness value given to each player is simply the proportion of correct moves to total moves made expressed as a percentage. To this total is added either a winning bonus of 150 points, or for a losing player 3 points for every move made in the game. The bonus added for losing players is designed to promote long games and prevent the players' learning stalling at local minima. A player who plays four moves, and only makes one (fatal) mistake scores a fitness of 75, and is unlikely to improve further if the fitness function only reflects the proportion of winning moves. The bonuses encourage longer games, and more generalised good play. The winning bonus is introduced to ensure that winning play, as the main goal of the board game problem, is rewarded above all other fitness criteria.

The rules for breeding new players are as follows. The four best players are kept on for the next generation. The rest of the players are replaced, four by offspring and two by new randomly initialised players. The only difference is the pool members chosen for breeding. The existing players are replaced by offspring in reverse order to that of their fitness ranking. This is done in order to include the soon to be replaced players in the fifth, sixth and seventh position in the breeding. For each replacement, offspring are bred from two parents randomly chosen from all players above the new offspring in the fitness ranking. This allows players which do not qualify in the top four a limited opportunity to breed before they are replaced. The further up the fitness table these players rank, the more offspring they qualify as parents for, before they are replaced. The top four players are, of course, candidates for breeding all four offspring.

The limited inclusion of lower ranking players in the breeding process is intended to decrease the risk of the GA settling in local minima [6]. If the breeding network weights are too similar, the resulting offspring may well be carbon copies of the original. Although mutation may eventually reintroduce diversity, mixing the pedigree of new offspring helps to increase the mix of weights in the pool, and consequently, any local minima will eventually be surpassed by a new combination of weights.

6.3 The F&G-NN/GA Experiments

Each generation, a pool of ten players play one game each against an AI mini-max type opponent. After each generation, the fitness function ranks players based on their performance, and breeding takes place based on these results. Each experiment lasts 200 generations, and was repeated ten times to reduce the chance of anomalous results. A further ten runs were then performed with the GAs and the AI opponent switching roles. The number of games won by the GA was recorded, as was the percentage of winning moves played in each game.

6.4 F&G-NN/GA Results

The genetic algorithm breeds players able to beat the AI player. As the Fox player, the GA is particularly strong, and is very quickly able to beat the AI player (see figure 5).

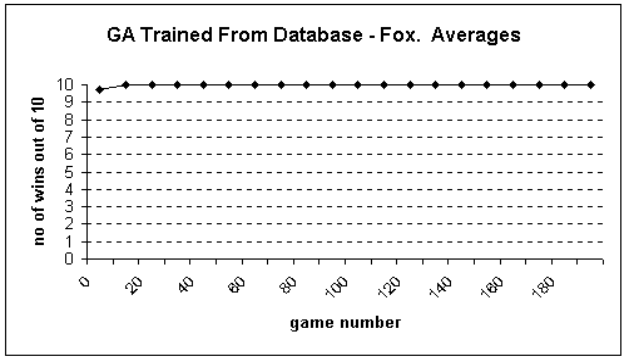


Fig. 5. Neural Network/GA. Average no. of wins out of 10 for the ***Fox***. This graph represents the number of wins out of 10 for every 10 epochs of the 200 generation training run. The results are taken from the highest ranking player in a pool of ten. (Averaged over ten runs.)

It takes the GA considerably more time to breed network weights capable of defeating a Fox AI player. Two out of the ten runs did not win any games at all during the training period. On average, results show a smooth learning curve (figure 6) and a consistent improvement in performance which results in a player strong enough to win eight games out of ten.

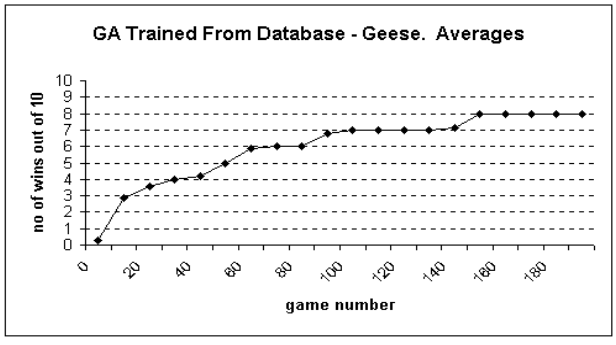


Fig. 6. Neural Network/GA. Average no. of wins out of 10 for the ***Geese*** (Results are based on the average from 10 training runs.)

The results showing percentage of winning moves made per game also show a steady increase in performance throughout training (see figures 7 and 8). This is

unsurprising, as the percentage of winning moves is one of the main features of the GA fitness function. Nevertheless, these results give a good indication that the quality of play increases fairly steadily throughout training.

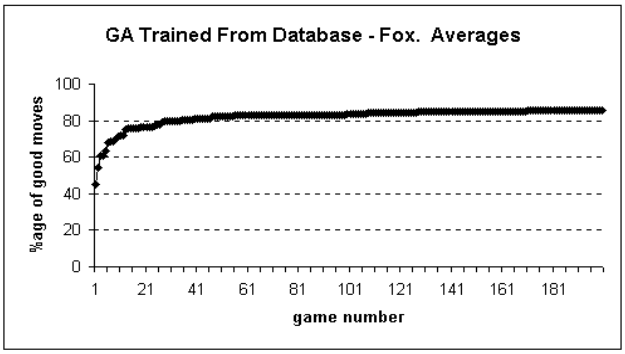


Fig. 7. Neural Network/GA. Average percentage of winning moves for ***Fox*** This graph shows the percentage of moves in each game which are held in the perfect moves database as eventual winning positions. Results are an average from 10 training runs.

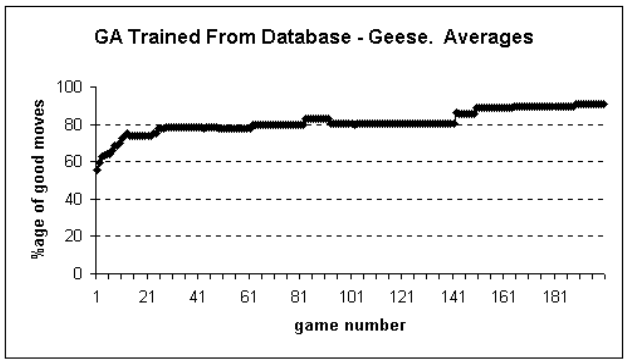


Fig. 8. Neural Network/GA. Average percent of winning moves - ***Geese***

7 Conclusions

The three learning algorithms implemented, rote learning, neural network/BP and neural network/GA, have been able to improve their ability to play the board game Fox and Geese. Table 1 shows the total number of games won by each algorithm during training, and the results show a clear measure of success for both learning techniques. The neural network/GA shows the best aptitude for learning as both Fox and Geese. The rote-learning algorithm displays a poorer performance, especially for those games where it played as the Geese.

TABLE 1. Total (average) number of wins during 200 training games with each starting. (Final total out of 400)

Training Method	Average no of Wins (out of 400)		
	Fox	Geese	Total
Rote learning	165.3	36.1	201.4
Neural Network/BP	186.5	158.1	344.6
Neural Network/GA	199.7	119.8	319.5

As may be noticed from table 1 above, there were a very high number of wins for the neural network/GA Fox. This is due to the fact that the simple AI Geese did not play well at the relatively low ply search used in these experiments (due to cpu-time constraints). Fortunately, as stated in section 4.2 and shown in figure 4, winning was not the sole fitness criteria used in the training process.

The rote-learning algorithm clearly experiences problems overcoming the native deficiencies of its AI board evaluation function, and has shown a general lack of flexibility when approaching the learning task. Considering that, unlike the neural network/BP and neural network/GA, the rote-learning system has a pre-programmed board evaluation function and does not have to learn the game from scratch, learning (although clearly demonstrated during the Fox runs, at least) could be said to be incremental at best.

Although some game knowledge is pre-existent in the rote-learning algorithm within the board evaluation function, this system is only one of the methods which presents unsupervised learning. The neural network/BP and the neural network/GA are helped to learn the game by having access to the contents of a perfect moves database. In this respect, the rote-learning results are more significant than they at first appear.

The genetic algorithm is clearly established as an effective technique for the training of a neural network. Excellent results are achieved from training runs as both the Fox and the Geese. The GA shows results far better than rote learning, even though playing ability was learned from scratch. Perhaps detracting a little from these results though, is the supervised nature of the learning.

The back propagation neural network applies itself to the problem of playing Fox and Geese with great success. Showing the most wins of all, the neural network is able to easily supplant its simple AI competitor whether playing as the Fox or the Geese.

7.1 Machine Learning Behaviours

It is interesting to note that all three of the machine-learning techniques perform much better when playing as the Fox (see Table 2). This at first may seem at odds with the remarks in section 2.1, which suggest that for experienced Fox and Geese players, the Geese have the advantage, but the following discussion clarifies these results.

TABLE 2. Proportion of the total wins achieved as Fox and Geese.

Training Method	% of Total Wins	
	Fox	Geese
Rote learning	82.1%	17.9 %
Neural Network/BP	54.1%	45.9%
Neural Network/GA	62.5%	37.5%

Table 2 above shows the spread of all the games won by the machine learning algorithms during training runs of 200 games playing the Fox, and 200 games as the Geese. Results represent a percentage of the total number of

victories from the 400 games (and are averaged over 10 training runs).

Although all three algorithms show ability to play the game, and can usually defeat a simple AI opponent, neither of them can really be described as expert players. During a game of Fox and Geese, the onus is on the Geese to preserve a defensive line of pieces. For a Fox, the tactics involved in playing against an expert are much more complex than those used against a less experienced player. If the Geese player is prone to occasional mistakes, it is not overly difficult to capitalise on one of these errors, and win the game. This is what appears to be happening here. The various machine learning players are competent enough to play well as the Fox, adopting the simple tactics of pushing against the line of Geese and waiting for an opportunity to slip through. Taking the more complex role of the Geese is more problematic for the learning algorithms. The task of keeping a tight formation of pieces is more complex, and each mistake can potentially lead to a quick defeat.

The assertion made in section 2.1 that the simple AI player performs better as the Geese is qualified by observing that lower search depths may expose flaws in the AI player's game. Both machine-learning implementations are trained at relatively low search depths (4 ply for the neural network/GA system and 6 ply for rote learning), so the AI Geese player may well be prone to making mistakes which the opposition can capitalise on.

References

[1] Allis, L.V. 1994. *Searching for Solutions in Games and Artificial Intelligence.* Ph.D. thesis, Department of Computer Science, University of Limburg.

[2] Berlekamp, E. R., Conway, J. H. and Guy, R. K. 1982. *Winning Ways For Your Mathematical Plays*. London: Academic Press.

[3] Carling, A. 1992. *Introducing Neural Networks*. London: John Wiley & Sons.

[4] Chellapilla, K. and Fogel, D. B. 1999. "Co-evolving checkers playing programs using only win, lose, or draw." In *Proceedings of SPIE's AeroSense'99: Applications and Science of Computational Intelligence II*.

[5] Chisholm, K. J. and Bradbeer, P.V.G., 1997, "Machine Learning Using a Genetic Algorithm to Optimise a Draughts Program Board Evaluation Function", *Proceedings of IEEE International Conference on Evolutionary Computation, (ICEC'97), Indianapolis, USA, 1997.*

[6] Goldberg, D. 1989. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Reading, MA: Addison-Wesley.

[7] Lake, R. Schaeffer, J. and Lu, P. 1994. Solving Large Retrograde Analysis Problems Using a Network of Workstations. In H.J. van den Herik, I.S. Herschberg and J.W.H.M. Uiterwijk (eds.). *Advances in Computer Chess* VII , Maastricht: University of Limburg.

[8] Levy, D. and Newborn, M. 1991. *How Computers Play Chess.* New York: W. H. Freeman.

[9] Perham, M. (Ed.). 1998. *The Encyclopedia of Games.* London: Aurum Press.

[10] Richards, N., Moriarty, D., McQuesten, P. and Miikkulainen, R. 1997. "Evolving neural networks to play Go." In *Proceedings of the 7th International Conference on Genetic Algorithms*. East Lansing, MI.

[11] Samuel, A.L. 1959. "Some Studies in Machine Learning Using the Game of Checkers." *IBM Journal of Research and Development*, vol. 3, no. 3.

[12] Shannan, C. E. 1950. Programming a Computer for Playing Chess. *Philosophical Magazine,* 41(4), 256-275.

[13] Shaeffer, J. and Lake, R. 1996. Solving the Game of Checkers. In Richard J. Nowakowski (ed.). *Games of No Chance*. Cambridge: Cambridge University Press.

[14] Sutton, R. S. 1988. Learning to Predict by the Methods of Temporal Differences. *Machine Learning*, 3, 9-44.

[15] Tesauro, G. and Sejnowski, T. J. 1989. A parallel network that learns to play backgammon. *Artificial Intelligence*, 39 ,357-390.

[16] Turing, A. M., Strachey, C., Bates, M. A.and Bowden, B.V. 1953. Digital Computers applied to games. In Bowden, B.V. (Ed.). *Faster Than Thought*, London: Pitman.

Teams of cognitive agents with leader: how to let them some autonomy.

Damien Devigne
LIFL - CRNS UMR 8022
Université de Lille 1
Villeneuve d'Ascq, F-59655 cedex
devigne@lifl.fr

Philippe Mathieu
LIFL - CRNS UMR 8022
Université de Lille 1
Villeneuve d'Ascq, F-59655 cedex
mathieu@lifl.fr

Jean-Christophe Routier
LIFL - CRNS UMR 8022
Université de Lille 1
Villeneuve d'Ascq, F-59655 cedex
routier@lifl.fr

Abstract- Most often situated multi-agent simulations, of which platform-games are an example, uses reactive agents. This approach has limitations as soon as complex behaviours are desired. For these reasons we propose an approach using cognitive agents. They have knowledge, objectives and are able to build plans in order to achieve their goals and then execute them.

In this paper we particularly address the problem of teams of cognitive agents. We chose to build teams directed by a leader. One major problem is the building of the team plan and in particular one difficulty is to find the means in order to let autonomy to the team members. This can be done if the leader builds abstract plans. We present in this article a solution to this problem.

1 Introduction

Our work aims at producing agent-based simulations where the agents, situated in a geographical environment, behave "rationally". An application of such a work can be simulation platforms of which computer games are an instance, movies is another like "*The Lord of the Rings*" and the MASSIVE application illustrate it[1].

The main characteristics that we consider for these simulations are: first, the environment is defined by a geography, then the notions of positions or coordinates have a meaning, this is a critical feature since relative positions must be considered before executing an action and consequently moves must be performed; second, the world is dynamic (agents can appear or disappear), and then heavily non monotonic (ie. values, once known, can change); third, agents are different: they can perform and suffer actions that are different from one to the other; and last, the agents are embodied: they are situated in the environment, and have a partial perception of it, from this it follows that the agent's knowledge is incomplete, moreover because of the non monotonic nature of the environment, this knowledge can be wrong.

According to J. Laird, computer games constitute the "killer application" for human-level AI [LvL00]. The characters involved in video games, like FPS or role-playing games, have indeed to be perceived as autonomous entities with increasing realistic behaviours. They have to be *convincing*, thus their behaviour must comply with the rational expectations of their partner or opponent human players. They also need to adapt to new situations, acquire additional abilities throughout the game, etc. In addition, team strategies are also often useful. Some research has been done concerning agents and games [Nar00], and most of them concern reactive agents [Nar98].

But reactive agents, while effective in several cases, offer limited behaviours. Indeed their behaviours are "short term directed" and not "goal oriented". Their ability to perform some tasks depends on the immediate surroundings and is not the result of wilful acts. In current commercial games, too often the character's behaviours are reactive ones, coded using scripts based on trigger/action sets. This approach has limitations [Toz02]. First the obtained behaviours are rather limited and it is difficult to get deliberate group behaviours unless they hard-coded them. This leads to a second major problem: the software design concern. It appears to be very difficult to reuse parts of AI from one game to another: scripts are too much tied to game design.

Our proposition aims at offering cognitive, driven by goals, proactive agents. To use cognitive agents allows to obtain more abstract reasoning. From one simulation/game to another the cognitive behavioural engine stays the same even if the context changes, and the behavioural components, the *interactions*, can be at least partially reused. Our approach uses declarative knowledge and thus favours the separation between the game logic and code. This promotes reusability and should ease the development.

In a first part we describe how we design simulated worlds or game environments: the geography, the laws that rule the world (the interactions) and the cognitive agents and their behavioural engine. Then we discuss teams and present our proposition to obtain team plans.

2 Simulations with cognitive agents

We define a simulation as follows:

$$Simulation = E \times I \times A$$

where E is the topologic *environment*, I is the set of *interactions* that rule the simulated world and A is the set of *situated agents* involved in the simulation. In the following subsections we will quickly define these three points.

2.1 Environment

The environment describes the geography of the simulated "world". We represent the environment by a graph where nodes are *places* and vertices denote *path* from one place to another (see Table 1). A place is an elementary geographical area. The granularity of a place depends on the simulation, the only constraint is that inside a place there is no restriction neither for moves, nor for perception (restrictions due

[1] see http://www.massivesoftware.com/news.html

to the other agents, like collision problems, are exempt). A place can represent a room, a town or any other part of the environment, and inside a place the position of an agent can be handled discretely or continuously depending on needs.

$$\begin{aligned} \textit{Environment} &= \{\textit{place*},\textit{path*}\} \\ \textit{path} &= (\textit{place}_{origin}, \textit{place}_{dest}, \textit{condition}) \end{aligned}$$

Table 1: Definition of environment

A path denotes an oriented transition between two places. It is defined by the places that it links, and a condition that must be satisfied if an agent wants to use this path (usually most of the conditions are simply *true*). The paths are oriented and the condition to go from some place a to a place b is not necessarily the same than the one to go from b to a. This formalism allows to describe, for example, that a door between two rooms must be opened if we want to go from one room to the other, or that an agent must be able to swim to cross a river between two fields. In this last case, our approach allows, depending on needs, to choose to represent or not the river with a place. It depends on whether ot not the crossing of the river has a meaning in the simulation (see figure 1).

Figure 1: Left: river is modelled. Paths are: (a,r,*"agent can swim"*), (r,a,*true*), (b,r,*"agent can swim"*), (r,b,*true*) Right: river is not modelled. The paths are: (a,b,*"agent can swim"*),(b,a,*"agent can swim"*).

2.2 Interactions

Knowledge representation is based on the notion of what we call "*interactions*"[MPR03]. Since the objective is to achieve simulations (like games are), it is necessary to represent the laws that rule the simulated world and to allow the agents to manipulate these as knowledge elements. We introduce our interactions in this goal.

Interactions are the backbone of our simulation model. They are at the basis of the knowledge representation in the simulation. Some agents (the actors) can perform interactions and others (possibly the same) can suffer from them (the targets).

An interaction is defined by a name and three parts:

- a *condition*, it tests the current context of execution of the interaction and consists mainly of tests on values of target or actor properties.
- a *guard*, it checks general conditions for the interaction applicability, typically it defines that to be fired an interaction requires that the distance between the target and the actor must be less than some given value.
  The guard is separated from the condition since it corresponds to the knowledge due to the geographically situated feature of the simulations. In a non situated context, one would have only the condition and action parts. The guard will be at the origin of the moves in the plan, moves that are indeed specific to situated problems. We do not express explicitly in an interaction that the agent has a move to do in order to fire it, we want the agent to plan it when required.
- an *action*, it describes the consequence of the interaction, it can be a change in the state of the actor and/or of the target (ie. a change of the value of a property), and/or the activation of an environment action (like the creation of an agent).

By example, to *open* an object (door, chest, window, etc.) makes it changing from *closed* state to *opened* one. The nature of the target is of no importance here (insofar as it can suffer *open*), this knowledge can then be represented in a "universal" way by the interaction (see below). In this sense, interactions are declarative knowledge: they describe an action and not how to solve/use it. Let us remark that there is no mention of moves to be performed to fire the interaction, the knowledge due to the situated property is mentioned in the guard. An interaction is a piece of *abstract* knowledge where the situated point of view is taken into account in the guard.

$$\textit{open:} \begin{cases} \texttt{condition} = \text{"}\texttt{target.opened} = \texttt{false}\text{"} \\ \texttt{guard} = \text{"}\texttt{distance(actor, target)} < 1\text{"} \\ \texttt{action} = \text{"}\texttt{target.opened} = \texttt{true}\text{"} \end{cases}$$

One advantage in using such interactions is that this approach favours a good software engineering design. Since interactions are not tied to a particular agent nor to a simulated world, they can be reused from one to another. This is clearly the case with the above *open* interaction. Reusability is of course an important concern in software engineering design and in particular in AI game design where it is reputed to be not applied although wished.

To increase the generic nature of our interactions we propose a way to specialize them. This is not the object of this paper to detail it but let us say that the aim is to keep the declarative and abstract nature while taking into account the fact that to solve a given abstract action can require different conditions. To solve that we use something like inheritance of interactions. Using an example should help to present it shortly: again consider the *open* interaction, we said that it can be applied to different types of targets and used in a plan such that "*to fetch an apple in the next room I must* open *this door*". Now let us assume that this door is a lockable one (and is indeed locked). From an abstract point of view the plan is still valid, but the *open* interaction must be understood as "make the door change from *closed* to *opened* state *when it is unlocked*". The problem is then how the same abstract plan (open the door to fetch the apple) can receive different solutions (just "open" or "unlock and then open") depending on the target (whether it is lockable or not). Our solution is to allow to specialize interactions by adding extra conditions, thus you create another *open* interaction that "inherits" the previous one and to which you add the condition `target.isLocked=true`. This is this version that is given as *can-suffer* interaction to the lockable agents.

2.3 Agents

Our agents are embodied agents that are situated in their environment, they are influenced by it and more precisely by where they are in it.

We distinguish two kinds of agents: inanimate and animate agents (not to mistake them with mobile/non-mobile agents). Both are defined by properties and can suffer interactions but the latter can also perform interactions and have a behavioural engine (see Table 2). The animate agents are cognitive and proactive agents. They are responsible for the dynamics of the simulations. The interactions that an agent can perform correspond to its abilities.

Agents	=	*Animate* | *Inanimate*
Inanimate	=	{*Properties**, can-suffer}
Properties	=	*(name, value)*
can-suffer	=	*Interaction-name**
Animate	=	*Inanimate* ∪ can-perform ∪ *brain* ∪ *goals*
can-perform	=	*Interaction-name**
brain	=	*planning engine* ∪ *memory*
goals	=	*interaction-goal* | *condition-goal*

Table 2: Agent's definition

The cognitive agents The structure of the animate agent's "mind" is presented in Figure 2. Agents have a memory that can be seen like a "degraded environment". This one represents the knowledge base for all the information gathered by the agent concerning the environment: the topology of the environment, the other agents (their position and state). This information is used by the planning engine to determine the action that the agent must try to execute in the environment to fulfil its goal. A perception module is used to pick up information in the environment and to update the memory, this perception is local. Updates are performed by a separate module that has an influence on the planning engine in order to adapt the currently computed plan to the new perceived situation. This last module is a kind of "short term memory". From this it results that the knowledge of an agent is not complete, then an agent may have to search for unknown information, and can be wrong, but the agent is supposed to behave with respect to its knowledge. This is due to the dynamic and non monotonic nature of the environment. Knowing that its environment is non monotonic must be taken into account by the agent.

Goals Animate agents have goals. The satisfaction of these goals leads the agents to behave according to a computed plan. There exist two kind of goals. First, the *interaction-goals*, they correspond to an (inter)action that the agent has to execute. The target of this interaction can be less or more precisely given: from a named agent to any agent that can be the target of the interaction, as shown in the next table:

goal	type of target
`eat(apple_12)`	a given named apple
`eat(an apple)`	any apple
`eat(*)`	any eatable (ie. "who can-suffer eat") agent

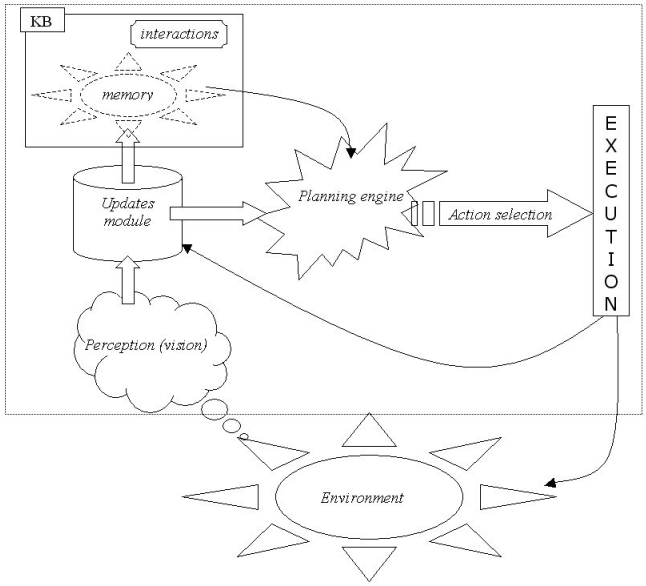


Figure 2: Different elements of agent's mind.

Second, the *premiss-goals* (or *condition-goals*), they correspond to a condition that the agent wants to become true. For example:

`actor.energy > 100:`
"having his energy being greater than 100"

Planning engine In order to achieve their behaviour the animate agents have an engine that computes plans in order to satisfy the goals given to them. The plan is produced by a backward chaining on the *can-perform* interactions of the agents. The plan building depends on the information stated in the memory (ie. the beliefs base) of the agent. In a rather classical way, the plan can be viewed as a tree (see Figure 3). The nodes are made of the different goals and subgoals encountered during the resolution. Some are *interaction-goals*, others are *premiss-goals*. Thus the tree is an alternation of condition and interaction nodes and corresponds to an AND-OR tree. AND-nodes correspond to condition-nodes (for condition-goals) and OR-nodes to interaction-nodes (for interaction-goals).

The condition and interaction nodes are classical case. The sons of a condition node are interaction-nodes built from the interactions whose action part offers a way to satisfy the condition (or help to satisfy it). The interaction-node's sons are built from the conditions that can be found in the condition and guard parts of the interaction: from these, condition-nodes are built. This is classical in backward chaining.

We want to underline a point that introduces differences in comparison with planning in non situated context. Indeed, since we consider embodied agents situated in a geographical environment and since we want to simulate their behaviour in such an environment, agents must perform moves. Typically an agent must move next to a target to interact with it. It is here that the guards that we have introduced in our interactions play their roles. To be allowed

to fire the action part of the interaction, the agent must satisfy the conditions and the guards. But guards, since they imply moves that are a crucial side-effect in situated simulations, require specific consideration. We have discussed this problem in [DMR04] where we show in which ways the situated context has an influence on "planning while executing". Indeed, considering only conditions and actions leads to *abstract plans* that are valid independently of any situated context: "to open a door only requires that it is closed and makes it opened". However, simulations in situated environment implies that in the *execution plan* moves must be done, and then "*to open a door*" requires also the actor being next to the door as expressed in the guard of the *open* interaction. This guard must be considered while building the plan. The backward chaining on the guard produces the moves and these moves can require specific planning in order to be achieved. The conditions that must be satisfied in order to be able to perform a move are the conditions that exist between the places in the computed path. Then the same *abstract plan* can lead to several *execution plans* each depending on the execution context where the agent is situated.

Let us just add that the plan is not rebuild at each step but partial replanning is done according to the new perceived information given by the updates module.

A small example To illustrate the different points described in the previous paragraph we will consider a very small and simple example. We consider a world with two places/rooms separated by a door d, the path between these two rooms has the condition "*d.locked=false*". Four interactions define the laws: *unlock*, *take*, *move*, *push* (see Table 3). In the world are an animate agent a that can perform these 4 interactions, and three inanimate agents, the door d that can suffer *open*, a key k that can suffer *take* (and can be used to unlock d) and a button b that can suffer *push*. The goal of the agent is to push on b. The figure 3 presents the planning in two different situations.

unlock:	*condition*	=	"`target.locked = true`"
	guard	=	"`distance(actor,target) < 1`"
	action	=	"`target.locked = false`"
take:	*condition*	=	`true`
	guard	=	"`distance(actor,target) < 1`"
	action	=	"`actor.own(target) = true`"
push:	*condition*	=	`true`
	guard	=	"`distance(actor,target) < 1`"
	action	=	"`target.pushed = true`"
move:	*condition*	=	`conditionsfoundinpath`
	guard	=	
	action	=	"`distance(actor,target) < 1`"

Table 3: Definitions of the interactions (adapted - but without distortion - to shorten the example)

3 Teams

In the previous section we have briefly described our approach to model the simulated world and the agents. In particular we present how we obtain individual agent behaviour

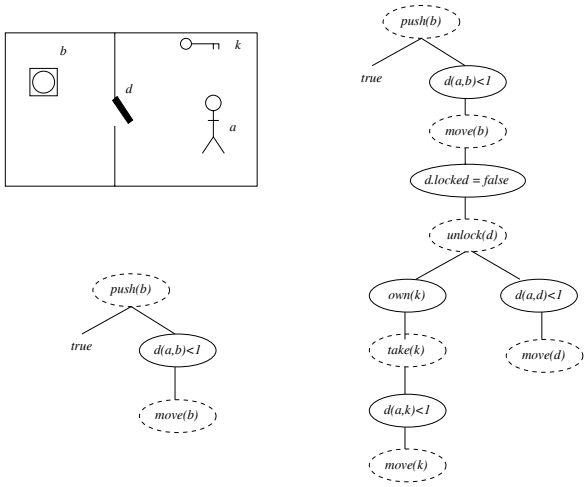


Figure 3: An animate agent a is situated in an environment where are also 3 inanimate agents, a door d, a button b and a key k. The goal of a is to push b. a must build a plan to achieve it. A plan can be drawn as a tree, nodes due interaction-goals are drawn with dashed lines and nodes due to condition-goals with solid lines. Depending on the execution context, different plans can be obtained. Left is the tree obtained when d is not locked and right is the case where d is locked. a must adapt its plan to the context.

that allows agents to achieve tasks. However individual behaviours are not enough, sometimes tasks must be done by groups, or teams, of agents. In computer games the need of teams is important: teams of fighters in FSP games, groups of units in strategy games, teams of characters in role-player games, etc.

Several situations can require the use of teams. First, getting a number of agents to do a job can speed up its achievement, this corresponds to task parallelization when several agents have the same abilities and perform similar tasks simultaneously, one agent could have done it alone but it would have taken more time. Second, in some cases, one agent is not sufficient to perform a task, and several must cooperate *simultaneously* to do it, by example this is the case when a heavy load must be carried and two or more agents are required in order to lift it, of course they must do this simultaneously. Third, complex tasks require a lot of different abilities and it is not often the case that one agent alone has all of them, then several "specialist agents" that together gather these abilities must cooperate to achieve the task. Of course, these three cases can mix. In this paper we mainly address this last case.

3.1 Teams of cognitive agents with leader

Making teams of agents work has been the subject of several approaches. Emergence is a solution to obtain a team behaviour [CGGG03]. But in this case we think that the notion of team work is only "apparent" since the team behaviour is a collateral effect of the sum of the individual behaviours and is not deliberate. We mean that the agents are not conscious that they work in a team and no team strategy is explicitly

planned.

Our objective is to make our cognitive agents work in a team and being aware of it. To perform this we make some choices in this paper:

1. the team is assumed to be already created, that means that we are not concerned here with the problem of recruiting an able agent or constituting the team before doing the job.

2. the team is directed by a leader which is an agent that plays a particular role in the team. It is known at the beginning.

3. the leader is in charge of the team strategy and coordination, therefore our work does not consider the cases where agents negotiates to cooperate and to find an agreement on a plan. For example, plan merging [KMS98, AFH$^+$97] is not our interest here.

Having a leader that builds the team plan can be seen as a restriction since this implies that control is partially centralized. However one can see by oneself that this is often the case in real life: firemen in a squadron or workers in a building site obey to the orders of their leader. The point is that in such teams, even if the leader gives orders, team members still have their autonomy. They must behave according to the leader plan but have to use their knowledge and abilities to achieve these orders.

Indeed, an important feature is the granularity of the leader orders. A site foreman does not order a bricklayer "*take this red brick, bring it there and put it on the foundation, then take this second one, bring it there and put it next to the first one, then take this third one...*". His order is simply "*build this brick wall here*". How the wall is built is the responsibility and the competence field of the bricklayer. As we see here, the leader gives rather high level orders and is not concerned with details. Moreover these orders can be abstract insofar as they are not necessarily tied to a particular situated context: the foreman can use the plan of construction in his office to show the bricklayer the walls to be built, this one is in charge to do it according to the site constraints and situation. The worker is autonomous once the order has been given, probably he only must report when he succeeds or even informs his leader when a problem he cannot solve occurs.

Therefore, the leader is in charge to build the plan that solves the team goal but this plan does not describe the solution in full details.

In the following (see paragraph 3.3) we propose a solution to reproduce this behaviour: the leader has the knowledge about its team members abilities, it builds an abstract plan to solve the team goal and it distributes orders to the members. The members autonomously resolve their tasks and report to the leader.

3.2 Description of a team

Describing a team simply as a group of agents is not sufficient. Our proposition consists in describing the team structure independently of any concrete agent and then to instantiate it with the members.

Since we are interested in teams that gather several complementary specialists, we design the team structure in term of roles. A role corresponds to a set of abilities (ie. interactions) required to play it (see Table 4). We add a cardinality to each role, this allows to precise when several agents of the same type are required in a team. This information is for example useful to handle dynamic reorganization of the team, but we will no more use it in the following of this paper.

To instantiate a team consists in selecting existing able agents and to attribute them some role in the team. Of course to be able to play a role an agent must have all the interactions that describe it in its *can-perform* property. Then a team is given by a team-structure and a mapping from the role in the structure and the agents members of the team.

TeamStructure	=	*(Role, Cardinality)**
Role	=	*Interaction**
Cardinality	=	*Natural..Natural*
Team	=	*TeamStructure* $\times$ *(Role,Agent)**

Table 4: Definition of team

The knowledge concerning the team is given to the team leader. In this paper we are not interesting in how the leader recruits its team-mates.

3.3 Our proposition

Giving autonomy to the team members is an important point. First, as said before, this is more realistic and simulates what happens in real life. Second, this avoids to obtain stupid behaviours, in particular because we consider situated agents in dynamic environments like game worlds are. One must not forget that, as we say earlier, agent's knowledge, and by way of consequence the leader's knowledge, is not necessarily correct. Then, in the case where the leader builds the plan in every detail and gives very precise orders to the team members, those having no right to modify them, it is more than probable that members will be confronted with unexpected situations and will not be able (nor authorized actually) to handle them. As an example, such situations can be due to objects that are not where they are supposed to be. Then a precise order like "*go to a given precise location and take the brick*" can not be solved by a non autonomous agent if the brick has been moved. This is not the case with the more abstract order "*take the brick*" given to an autonomous agent that can decide and plan how to find the brick according to the environment it is confronted with. As a third advantage this provides an easy way to consider team of teams, we will discuss this later in the paper.

As it has been described earlier our agents are cognitive ones and are able to build plan according to their knowledge. Insofar as individual and team plans are of the same nature, there is no reason that the individual planning strategy could not be applied to team planning.

So, the problem that arises is: *How to adapt the individ-*

ual planning to team work in order to let some team members autonomy. Here follows our proposition.

As we have seen the plan can be viewed as a tree where root is the goal and leaves are actions to be executed in order to solve the goal. In a team plan, leaves are then the orders that the leader gives to the members. To be able to build this plan, the leader must have some knowledge concerning the members abilities, that is about their *can-perform* interactions.

Let us consider that the leader knows all the *can-perform* interactions of the members. Then exactly like in the individual planning, he could build a plan to solve the goal. But in this case he would build a full plan and team members will no more have any planning autonomy! So the solution is to not let the leader plans "until the end", but to limit the tree depth. But this can not be done arbitrarily. There is no reason to decide that the leader unfolds the tree until it has a depth of 3 rather than 5 or 10. The appropriate depth will depend on the goal and the members abilities, it will be different at every time. In order to compute the depth's limit the leader would have to compute the full plan before to cut it at a relevant depth. It is a nonsense to compute the full plan, then to forget it and ask the members to re-build it!

Therefore this approach is not correct. We must not forget that we want the plan built by the leader to be abstract. And then the leader does not need to have explicitly all the knowledge about the *can-perform* of its team members. Actually, the leader only has to know what high-level tasks the members are able to do. It even does not need to know by itself how to perform these tasks, like a foreman does not necessarily have to know how to build the wall, it suffices that he knows that the bricklayer is able to do it. Indeed, the leader has to know what things must be done but not necessarily how to do them.

To achieve this we propose to hide some knowledge to the leader: the guards and conditions in the *can-perform* of the members are hidden to the leader. Thus the leader can know which of the member's interactions can be used in its plan since, knowing their action parts, it can use them during its backward chaining. However, since the leader knows no condition (nor guard) for these, it considers they are satisfied and stops the chaining. Then these interactions become necessarily leaves of the leader tree plan and can be distributed to the members as goals. Those members, having full knowledge, are able to make the appropriate plan.

Let us take an example. We have one agent named a_0 who can perform some interaction I_0 (see Table 5). Two other agents, named a_1 and a_2 can respectively perform interactions $\{I_1, I_3, I_4\}$ and $\{I_2, I_5\}$ (see Table 6).

a_0 will be the leader of the team. The team structure is made of two roles r_1 and r_2 that are defined respectively by interactions $\{I_1, I_3\}$ and $\{I_2\}$. This implies that the "high level" tasks the leader can ask to the members are to satisfy p_1, p_2 or p_3. The conditions and guards of these interactions are hidden to a_0 (see Table 5).

As we can see, agents a_1 and a_2 can play roles r_1 and r_2 respectively, they are chosen as team members.

Now, the team is given the goal p_0. The leader, a_0 builds

	leader.can-perform	team.can-perform		
name	I_0	I_1	I_2	I_3
conditions	p_1, p_2	–	–	–
guard	true	–	–	–
actions	p_0	p_1	p_2	p_3

Table 5: Leader's knowledge, conditions and guards are hidden for the interactions of the team roles. They are considered as *true*.

name	I_1	I_2	I_3	I_4	I_5
conditions	p_3, p_4	p_5	true	true	true
guard	G_1	G_2	G_3	G_4	G_5
actions	p_1	p_2	p_3	p_4	p_5

Table 6: Definitions of *can-perform* interactions of team members.

the plan for it. According to its knowledge, the backward chaining leads to use interaction I_1[2] that requires p_1 and p_2 to be solved. These lead respectively to the use of I_1 and I_2 (in this plan I_3 does not interfere from the leader point of view). For the leader, I_1 and I_2 have their conditions and guards satisfied (since they are hidden and seem to have none), then it stops its planning here (see Figure 4).

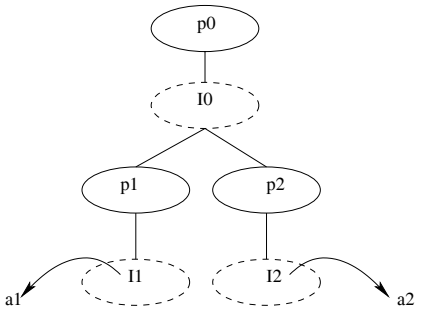


Figure 4: Tree representing the plan built by the leader. It is developed until the abstract team's interactions are reached, goals can then be given to the team members.

Now the leader can distribute the tasks to its team members according to their role in the team, indeed the leader is not able to perform the task itself since I_1 and I_2 are not in its *can-perform*. Then it gives p_1 to a_1 as goal and p_2 to a_2. Now each agent autonomously solves its goal according to its knowledge and builds the appropriate plan (see Figure 5). Then it can determine which action to fire to solve its goal and can inform its leader when it succeeds or if it fails.

The importance of the autonomy of the agent is increased while considering the influence of the surrounding environment and specially with the situated aspect. This is expressed within the guards whose resolution has not been detailed in above trees. Indeed, as we have seen since guards express that the actor must be near the target in order to fire the interaction, they requires planning to be solved. But this is highly context dependant. And it would have been

[2] In the following we assume condition p_i not to be satisfied.

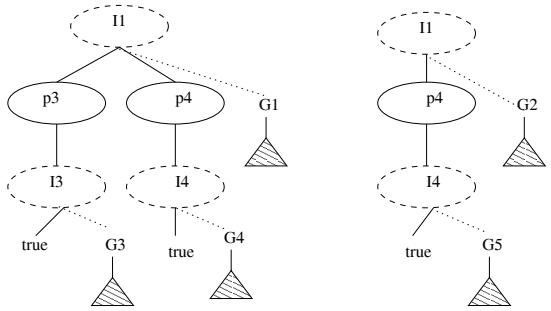


Figure 5: Trees representing the individual plans built autonomously by the 2 team members (guard's plan are not detailed, they depend on the context).

particularly irrelevant for the leader to plan it instead of the members.

3.4 Team of teams

With our above described approach it is easy to build "team of teams", or team built as a hierarchy of agents, where one "big leader" orders to subleaders that order to and so on, until "basic members". In fact it applies immediately to such cases without change!

Indeed, each level of the hierarchy corresponds to a level of decision with its type of orders. The higher in the hierarchy an agent is, the more high-level or abstract its orders are. For example, a works foreman can order a bricklayer leader to have walls built and the carpenter leader to have windows installed. Each of them orders to his team-mates to do the appropriate work.

With our approach the leader's planning stops with the abstract orders that can be given to team members and become a goal for them. If a team member agent is a leader itself, it applies the same procedure: starting from the goal that he has received, he builds a plan that stops when the abstract knowledge of its team (ie. interactions where conditions are hidden) is used. Then it can give orders to its team members. As we note nothing particular has to be done in order to consider hierarchies of teams: building the teams with their knowledge suffices.

4 Conclusion

In this paper we propose a mean to handle teams of cognitive situated agents directed by a leader. The presented solution let the team members some autonomy in the way they contribute to the team plan achievement. Indeed, the leader uses abstract knowledge on team's abilities to build an abstract plan and then distributes high-level orders to its team-mates.

Several problems have not been addressed in this paper and require complementary works, among them let us cite:

- how is the team built, that is how the leader recruits its team-mates?
- how to dynamically reorganize a team when an agent leave it (the cardinality information that are just mentioned in the paper can be used here)?
- how the information are exchanged inside the team?
- how to proceed in the case of teams with no leader?

They are, with others, the subjects of future works.

Thanks Many thanks to the anonymous referee that corrects so many of our language mistakes. Thanks to all referees for their remarks too.

Bibliography

[AFH+97] R. Alami, S. Fleury, M. Herrb, F. Ingrand, and S. Qutub. Operating a large fleet of mobile robots using the plan-merging paradigm. In *Proceedings of IEEE ICRA 97*, 1997.

[CGGG03] D. Capera, J-P. Georgé, M-P. Gleizes, and P. Glize. The amas theory for complex problem solving based onself-organizing cooperative agents. In *Proceedings of WETICE'03*, pages 383–388, 2003.

[DMR04] D. Devigne, P. Mathieu, and J-C. Routier. Planning for spatially situated agents in simulations. In *Proceedings of the 2004 IEEE/WIC/ACM International Joint Conference IAT'04*, 2004.

[KMS98] Subbarao Kambhampati, Amol Mali, and Biplav Srivastava. Hybrid planning for partially hierarchical domains. In *Proceedings of the 15th national/10th conference on Artificial Intelligence/Innovative applications of artificial intelligence*, pages 882–888. AAAI, 1998.

[LvL00] John E. Laird and Michael van Lent. Human-level AI's Killer Application: Interactive Computer Games. 2000.

[MPR03] P. Mathieu, S. Picault, and J-C. Routier. Simulation de comportements pour agents rationnels situés. In *Actes des Secondes Journées Francophones MFI'03*, pages 277–282, mai 2003.

[Nar98] A. Nareyek. Specification and development of reactive systems. In *1998 AIPS Workshop*, pages 7–14, Menlo Park California, 1998. AAAI Press.

[Nar00] A. Nareyek. Intelligent agents for computer games. In *Computers and Games, Second International Conference, CG 2000. LNCS 2063.*, pages 414–422, 2000.

[Toz02] Paul Tozour. The perils of ai scripting. In *AI Game Programming Wisdom*, pages 541–547, 2002.

Incrementally Learned Subjectivist Probabilities in Games

Colin Fyfe
Applied Computational Intelligence Research Unit,
The University of Paisley, Scotland.
Colin.fyfe@paisley.ac.uk

Abstract
In this paper, we show how our AI opponents learn internal representations of probabilities. We use a Bayesian interpretation of such subjectivist probabilities but do not implement full Bayesian methods of parameter estimation since we wish the AIs to be as human-like as possible. Thus the parameters of the subjectivist probabilities are learned incrementally.

1. Introduction

Following the seminal work of Axelrod [Axelrod 1984], we have previously investigated games of incomplete information using such tools as Evolutionary Algorithms [Wang and Fyfe 2004a], Population Based Incremental Learning [Fyfe and Wang 2004], Artificial Immune Systems [Wang and Fyfe 2004b, Fyfe 2004b] and Artificial Neural Networks [Fyfe 2004a]. Yet such investigations have left us somewhat dissatisfied in that the mechanisms whereby the games are resolved are often remote from those which we humans actually use to play games or solve paradoxes. Additionally, such modern artificial intelligences are essentially distributed in that we are rarely in a position to, for example, isolate *the* specific individual connection within an artificial neural network or identify *the* specific interaction in an evolutionary simulation or isolate the effect of a specific interaction between an artificial antibody and an artificial antigen which led to the development of a solution within the game.

Yet there are advantages in the above set of Artificial Intelligence (AI) techniques: we turned to them because we were dissatisfied with the current intelligences exhibited by the "AI" in contemporary computer games. These typically form intelligence with the use of standard techniques such as expert systems, A* searches and so on. Such intelligence is fast to implement but often lacks robustness: the intelligence is static and can usually be outwitted by a human opponent who learns the AI's technique and circumvents it. For example, the AI in computer games is often very good at managing micro-resources and will often launch steady waves of attacks against human-controlled opponents; the human response is usually to defend his resources against such (generally predictable) attacks while simultaneously building a bank of sufficient resources to launch an overwhelming attack against the AI's resources. The AI is essentially beaten because of its lack of flexibility. We wish to retain the flexible responses which the above technologies incorporate while making the responses which our AI's exhibit both more humanlike and more transparent. To do so, we turn to probabilities which we update incrementally during the game as more information becomes available.

We may view the application of intelligence building within computer games as part of the greater game of creating machines which can think, something which many people consider to be impossible. However, we can do no better than turn to the authority of Von Neumann who is quoted in [Jaynes 2003] as saying:

> "You insist that there is something a machine cannot do. If you will tell me precisely what it is that a machine cannot do, then I can always make a machine which will do just that!"

We will demonstrate that using probabilities allows us to explicitly respond to any objections about thought and intelligence with our game AIs.

Note that we take the view that we are not interested in creating the most intelligent machine since, as we have seen with deterministic games, we can build a machine which is capable of beating any human. Nor do we wish to build an intelligent machine and then plumb in "artificial stupidity", a tactic which is easily identified as false by a human opponent. Rather our aim is build a system which truly mimics human game play. We tend to think of this approach as artificial humanity and will demonstrate initial attempts to follow this route.

2. Subjectivist Probabilities

The use of probabilities clearly enables us to explicitly formulate responses within a computer game in a "white-box" manner; however, we also wish that our implementation of probabilities to be such that a human observer can resonate with the logic of the AI's responses. We also wish to ensure that we have robustness and learning built into the system of probabilities.

Most engineers and many scientists still view probabilities within the frequentist paradigm: a probability is (loosely) a limit taken over an infinite

number of trials of the relative frequency of an event. This is challenged by the subjectivist or Bayesian paradigm [Jaynes 2003] which views probabilities as having no meaning outwith the agent who is assigning these probabilities. In other words, each person has a subjective view of the probability of any event being realised and this view may be quantised by a real number which should correspond in some degree to common sense. There are other slightly more esoteric 'desiderata' that must be taken into account but these are no more than the obvious constraints of consistency, conformity and so on.

There is a great deal of controversy associated with the use of Bayesian concepts in the context of game theory with many authors taking a very strong stance against their use: for example, Binmore ([Binmore 1992, page 487]) considers "that Bayesianism does not call for much mental prowess on the part of the players". Or even stronger, from [Gintis 2000, page 289], "the "belief" concept involves all sorts of philosophical nonsense". Yet, even if full-blown use of Bayesian methods is not usual, Bayesian updating of probabilities is widespread e.g. there is a chapter "Learning who your friends are: Bayes' Rule and Private Information" in [Gintis 200] which is devoted to probability updates according to Bayes' Rule. The same theme emerges in [Dixit and Skeath 1999] though these authors also stress the role of Bayes' Rule in two player, simultaneous play games. However using Bayes' Rule in this way does not necessarily constitute employing the whole Bayesian paradigm.

2.1 A Full Bayesian Game

Consider the game of Chicken: the two protagonists each prefer that the other gives way, but failing that, will themselves give way (Macho is better than Chicken but Chicken is better than Death). A full Bayesian treatment of such a game – the game will be non-fatal so it can be repeatable - would involve having a prior probability that one's opponent would be Chicken which will be updated as evidence appears. Examples of prior Beta probability (see Appendix) density functions are shown in Figure 1.

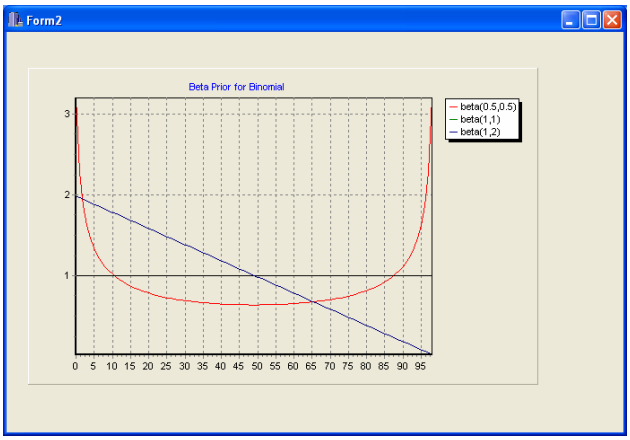


Figure 1. Three different prior beta distributions, (see Appendix) horizontal axis from 0 to 1 in hundredths.

One advantage of a Beta distribution is that it is so malleable (can be unimodal, bimodal, symmetric or not etc.) For this simulation we chose Beta(0.5,0.5) as a prior which says that we are giving most probability mass to our prior conviction that the opponent will be very macho or very chickenish. Now we sample from the binomial distribution in which the opponent will play macho with probability 1/8 and update the posterior probabilities accordingly (Figure 2).

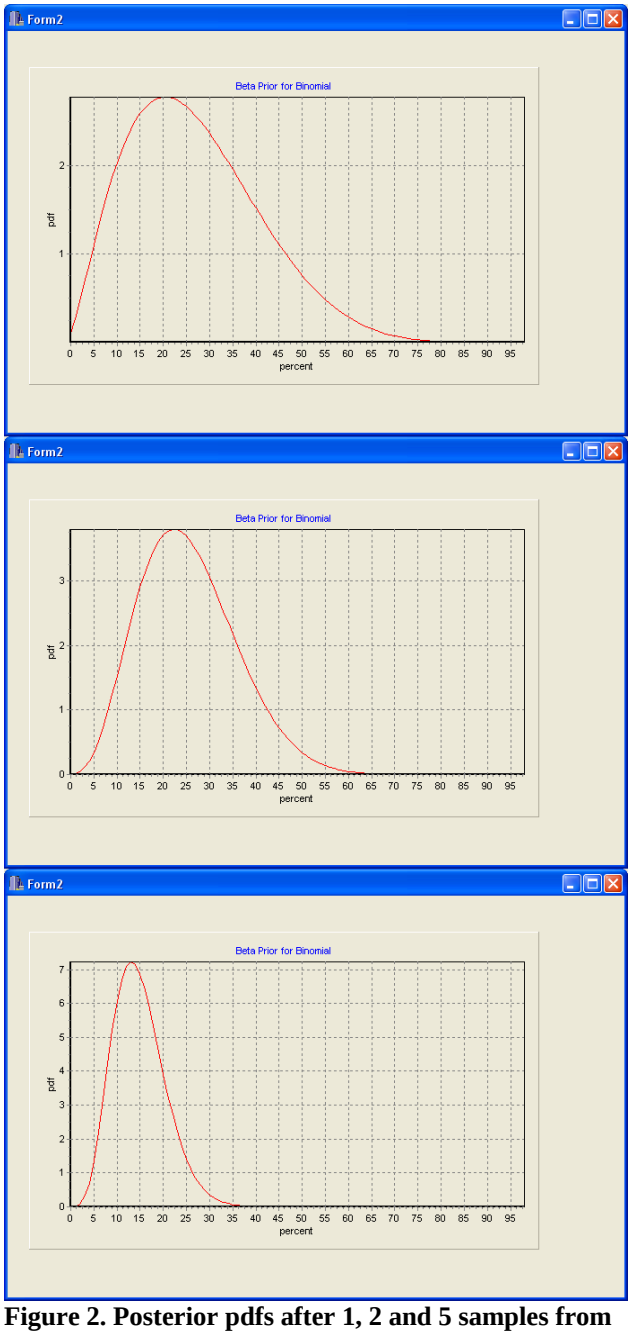


Figure 2. Posterior pdfs after 1, 2 and 5 samples from a chicken opponent.

We first note that the chicken end of the distribution quickly acquires most of the probability mass and that the variance of the distribution decreases in time (the AI becomes more confident of its beliefs). We also see that we have much more information than we really wish: we really only wish to identify whether we are playing

against a hawk or a chicken but we actually have a full pdf at each turn. Also there is plenty of evidence that humans are not actually very good at implementing Bayes' Rule when it comes to evaluating evidence. Thus in the following, we have adopted an incremental updating of subjectively-based probabilities rather than go down a full-scale Bayesian implementation.

3. Simulations

In this section, we will discuss simulations of a simultaneous game of complete information, and then a sequential game of incomplete information before returning to our game of chicken.

3.1 A simultaneous game of complete information

We first examine a coordination game to investigate whether our AIs can learn to cooperate with each other. To do this, we create a typical game situation: we have three AIs each with slightly different interests. We create a payoff function discussed in (Fudenberg and Tirole, 1991, page 15) in a game which we have previously investigated with Artificial Immune Systems. Let us have three AIs labelled 1, 2 and 3 and let the possible strategies be labelled A, B and C. The game consists of each of the AIs independently and simultaneously choosing a strategy and the strategy with the highest number of votes is adopted. If no strategy has a higher number of votes than any other, there is no payoff. The payoff functions are

$\mu_1(A)= \mu_2(B)= \mu_3(C)=2$
$\mu_1(B)= \mu_2(C)= \mu_3(A)=1$
$\mu_1(C)= \mu_2(A)= \mu_3(B)=0$

where we are using $\mu_i(X)$ as the payoff for AI_i when strategy X is adopted.

We will use this very simple model in a didactic role to illustrate how AIs can develop intelligent cooperation.

3.1.1 A Maximum Utility Model

Maximum utility models try to find the mode of a distribution and ignore what is happening in other parts of the distribution. In this section, our AIs select which of the three strategies they will use in a deterministic manner. However, we also have a probabilistic element in this section, in that each AI holds a mental model of the probabilities that each of its two opponents will opt for a particular strategy. For example, AI_1's model of probabilities is shown in Table 1.

	AI_1	AI_2	AI_3
A	*	0	$1-\beta_1$
B	*	α_1	0
C	*	$1-\alpha_1$	β_1

Table 1The first AI's estimate of the probabilities that the second and third will opt for strategies A, B or C.

AI_1 has no probability estimates of his own choices (the *s) but maintains a model of his estimates of his two co-players' actions. He considers that they are rational and so each has nothing to gain by opting for the strategy which will gain him 0. Therefore his estimate of the probability that AI_2 will opt for strategy A is 0. His estimate of the probability that AI_2 will opt for strategy B is α_1 and so his estimate that the probability that AI_2 will opt for strategy C is $1-\alpha_1$. We may write this as $P_1(2,A)=0$, $P_1(2,B)=\alpha_1$, $P_1(2,C)=1-\alpha_1$. Note that we need the subscripts on the probabilities such as α_1 since $P_3(2,B)=\alpha_3$ which need not equal $P_1(2,B)=\alpha_1$ since the probabilities are specifically defined as internal to the individual.

So AI_1 wishes to maximise his payoff. He can expect a maximum payoff of Max($P_1(2,B)$. $\mu_1(B)$, $P_1(3,A)$. $\mu_1(A)$) $=\text{Max}(\alpha_1, (1-\beta_1)*2)$. Note that, within the rules of this game, he can get only one or other of these, not both. Now AI_1 is himself rational and so he will base his strategy selection on which of these gives him the greatest expected return. Thus if $\alpha_1 > (1-\beta_1)*2$, AI_1 will select strategy B; otherwise he will select strategy A. He will never select strategy C since this gains him 0. The other AIs are reasoning similarly.

Now the game is started. Initially each AI's estimate of α and β is $0.5+\varepsilon$, where ε is a random number from a uniform distribution between -0.1 and +0.1. As the game progresses, each AI updates its probabilities in a rational manner: if AI_2 is seen to be choosing strategy B during the game, AI_1 updates $P_1(2,B)$ to $\alpha_1+\eta(1-\alpha_1)$ and changes $P_1(2,C)$ to $1-\alpha_1-\eta(1-\alpha_1)$. On the other hand, if AI_2 is seen to be choosing strategy C during the game, AI_1 updates $P_1(2,B)$ to $\alpha_1-\eta\alpha_1$ and changes $P_1(2,C)$ to $1-\alpha_1+\eta\alpha_1$. For this game, we give all AIs the same value of $\eta=0.05$. The first 25 rounds of a typical game are shown in Figure 3.

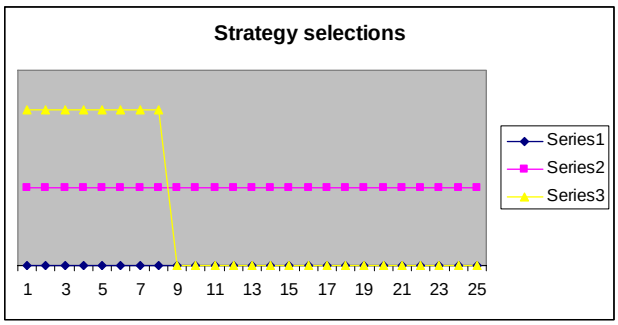


Figure 3 The AIs' selections during the first 25 rounds of the game.

The vertical axis of that figure has Strategy A at the lowest level, Strategy B in the middle and Strategy C at the top. We see that for the first 8 rounds each AI opts for its own best individual strategy. At round 9, AI_3 concedes to AI_1 by switching to strategy A and so is gaining a reward of 1 to AI_1's reward of 2. It would now pay AI_2 to switch to Strategy C so that it would share success with AI_3 (though AI_2 would be rewarded only 1 compared to AI_3's reward of 2) and it does make occasional switches

to Strategy C, however by now the important probabilities are very close to 1 or 0 and max(P_3(A,1), 2*(1-P_3(B,2))) stays at P_3(A,1). This is illustrated in Figure 4: we see a slight wobble in AI_3's belief in AI_2's intentions but it is never sufficient to overcome its conviction that AI_1 will select strategy A, thereby gaining it a reward of 1.

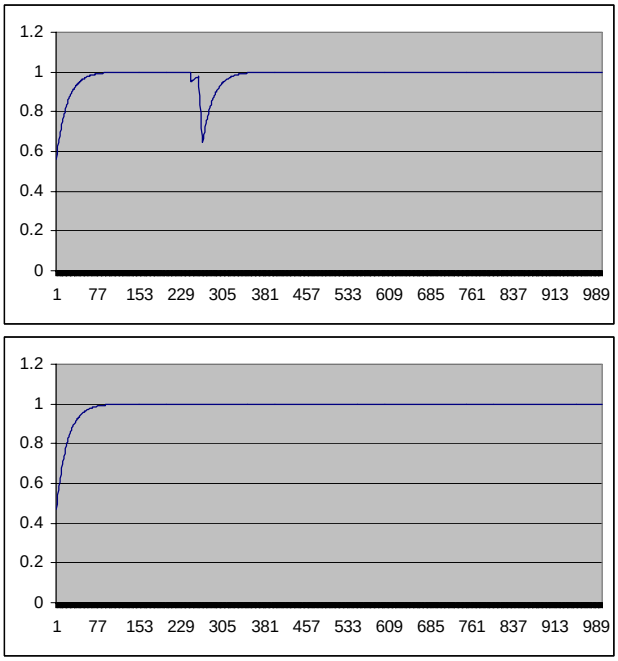


Figure 4. Top AI_3's estimate ofAI_2's probability of selecting Strategy C. Bottom: AI_3's estimate of AI_1's probability of selecting Strategy A.

This is somewhat typical of Maximum Likelihood (ML) methods. [Mackay 2004, page 306] notes (in the context of ML clustering) "This is known as overfitting. The reason we are not interested in these solutions with enormous likelihood is … the density is large over only a very small volume of parameter space". Also, as we have seen, it may be very hard work to escape from this region.

Note that we chose to increment the probabilities rather than go straight to the mode of the distribution. We do this in order to mimic human learning in a game situation: our aim is to make our AIs' behaviour as human-like as possible not to make the AIs play the game as well as possible. Thus rather than move straight to the modal probabilities, we incrementally update the probabilities just as a human would.

Note also that we have not built in to this system the requirement of consistency (e.g. P_1(2,B)≠ P_3(2,B)) since this is ubiquitous in human existence.

3.1.2 Using SoftMax Probabilities

However, we are dissatisfied that Player 2 did not play with human insight by switching his vote. We can alleviate this situation if we use parameters a_i and b_i which underlie the α_i and β_i in that α_i=exp(a_i)/Σexp(a_i) and β_i=exp(b_i)/Σexp(b_i). We then update the parameters using a_i directly: if AI_2 is seen to be choosing strategy B during the game, AI_1 updates a_i to a_1+ηa_1. Similarly with the b parameters. Now we have selections such as shown in Figure 5

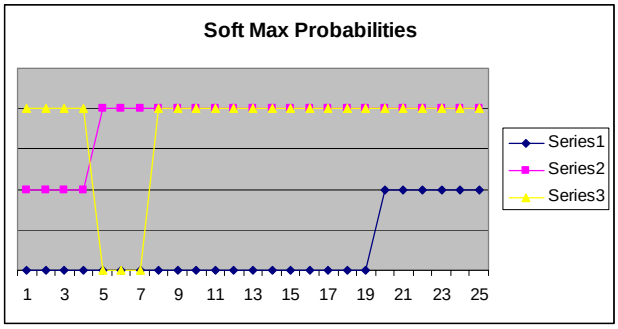


Figure 5. The selections made by the AIs when softmax probabilities are used.

We see that after 4 games in which each AI chooses its optimal strategy, both AI_1 and AI_2 simultaneously change to their respective second best strategies. However AI_2 is not immediately rewarded since AI_3 has changed. But after three more games AI_3 moves back to strategy C and AI_2 and AI_3 are both positively rewarded. AI_1 perseveres with strategy A till round 20 when it switches to strategy B. This situation continues till round 60 when AI_2 switches to strategy B so that now AI_1 and AI_2 are being positively rewarded. At round 193, AI_3 switches to its second option – strategy A - but this does not evoke the change in AI_1's behaviour till round 633 when it switches to A. We see that while switching behaviour does happen, it takes longer and longer to manifest itself with this simple model.

The longer time it takes for subsequent changes to players' votes is not a problem: in this paper, we keep the value of η constant but there is nothing to stop this increasing in time. We are implicitly then taking account of player's experience to be more confident of the changes he makes in his subjective probabilities.

3.2 A sequential game

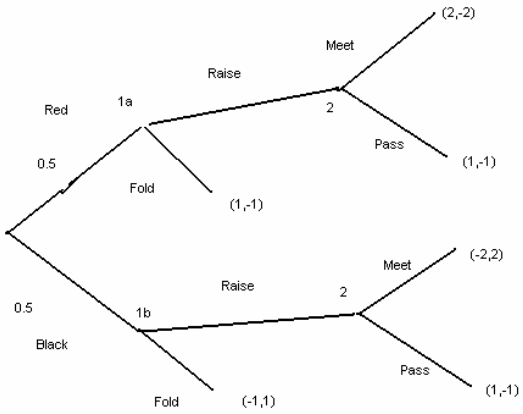


Figure 6 A sequential game

We have illustrated a simple sequential game in Figure 6. The game is between two players. Both players initially put £1 into a pot. Player 1 then draws a card from a deck containing an equal number of red and black cards but does not show it to Player 2. Player 1 may either fold or raise (put another £1 into the pot). If he folds and the card is red, he gets the pot; if he folds when the card is black, Player 2 gets the pot. If he raises, Player 2 must decide whether to meet the pot or pass. If he passes, Player 1 gets the pot. If Player 2 posts another £1, he gets the pot if the card is black but Player 1 gets the pot if the card is red.

This game is specifically used in [Myerson 1997] to illustrate Bayesian games. As with most textbooks (e.g. [Gintis 2000, Dixit and Skeath 1999]) on such sequential games of incomplete information, the discussion in [Myerson 1997] is based on Harsanyi's method. This envisages a prior move by chance which selects whether the card is black or red. Then Player 1's action must be discussed under two headings depending on whether the card is red or black. Player 2, on the other hand, does not require this treatment since, when he plays, he does not know the colour of the card. Thus we have a node 1a and 1b in the Figure but only a single node 2.

We consider Harsanyi's methodology to be rather unintuitive and revert to first principles i.e. we view the card drawn as a latent variable with its own set of probabilities. Let $P_i(X)$ be the subjective probability that Player *i* has that X will occur. Then we are interested in

- P_2(C=Black|1=Raise), Player 2's estimate of the probability that the colour of the card is Black given that Player 1's action was raise. This depends via Bayes' Rule on
- P_2(1=Raise|Card=Black)
- P_1(2=Pass|1=Raise), Player 1's subjective probability that Player 2 will pass when he sees Player 1's raise. This depends on
- P_1(Raise|C=Black), the probability that Player 1 will raise on a Black card

Then these probabilities can be used to determine the actions of the players: Player 1, being rational, will always raise on a red card; he will also raise if his expected payoff from a raise is positive:
$P_1(2=Pass|1=Raise).2 > P_1(2=Meet|1=Raise).(-1)$
i.e. $P_1(2=Pass|1=Raise).2 > (1- P_1(2=Pass|1=Raise)).(-1)$
i.e. $P_1(2=Pass|1=Raise) > 1/3$
If Player 1 raises, Player 2 requires to estimate
$P_2(C=Black|1=Raise)$
$= P_2(C=Black,1=Raise)/ P_2(1=Raise)$
$= P_2(1=Raise|Card=Black)P(Card=Black)/$
$\{P_2(1=Raise|Card=Black)P(Card=Black)+$
$P_2(1=Raise|Card=Red)P(Card=Red)\}$
$= P_2(1=Raise|Card=Black)/\{1+ P_2(1=Raise|Card=Black)\}$
since $P(Card=Black)= P(Card=Red)=0.5$.

Again we can increment these subjective probabilities using, if the card is black and Player 1 folded,
$P_2(1=Raise|Card=Black) = P_2(1=Raise|Card=Black)*(1-\eta)$
If the card is Black, and Player 1 raised, or the card is red,
$P_2(1=Raise|Card=Black) = P_2(1=Raise|Card=Black)*(1-\eta) + \eta$.
Note that $P_2(1=Raise|Card=Black)$ can rise even when the card is red since this is Player 2's subjective probability and, when he plays, he does not know the colour of the card.

If card is black, and Player 1 raises, Player 2 meets,
$P_1(2=Pass|1=Raise) = P_1(2=Pass|1=Raise)*(1-\eta)$
$P_1(Raise|C=Black)= P_1(Raise|C=Black)* (1+\eta)$

If Player 1 raised but Player 2 passed,
$P_1(2=Pass|1=Raise) = P_1(2=Pass|1=Raise)*(1-\eta) +\eta$
$P_1(Raise|C=Black)= P_1(Raise|C=Black)* (1-\eta) + \eta$

With these rules and η=0.05, within 20 games,
$P_1(Raise|C=Black)=1/3$ and $P_1(2=Pass|1=Raise) =1/3$ where they remain stably throughout the simulation.

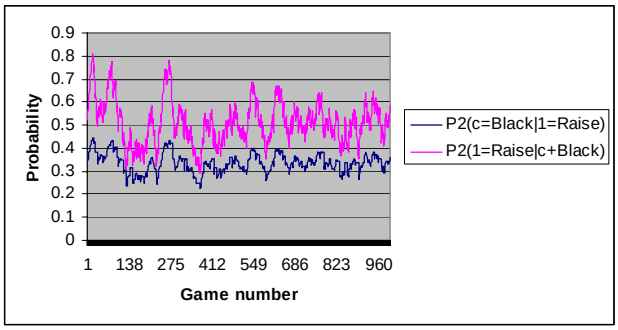


Figure 7. P2's changing subjective probabilities.

On the other hand, Player 2's probabilities are much more volatile (see Figure 7): $P_2(1=Raise|Card=Black)$ does hover around the correct figure (1/3), but is not fixed at this value.

In setting out this game, we have assumed that it is common knowledge (everyone knows, every one knows that everyone knows etc.) that Player 1 will always raise on a red card. Again this is done for the strictly subjective reason that this is exactly what happens in real life. Note that we have only modelled first order beliefs (about the state of the card) and second order beliefs (about the response the opponent will make given the player's actions) but not gone beyond that since humans are not capable of infinite recursions of beliefs.

We said that the colour of the card chosen is a latent variable, at least to Player 2 who plays without seeing the card. A more interesting game would be to have different numbers of red and black cards in the game and have the players attempt to estimate that. This type of latent variable estimation is discussed when we return to the game of chicken in the next section.

3.3 Return to Chicken

Finally, let us return to the game of Chicken which we discussed in a full Bayesian setting. Let us have two probability distributions determining the time when each of the two protagonists will chicken out of the game. Let t_i be the time that Player i will chicken out of the game and let t_i be drawn from a normal distribution of mean a_i and standard deviation σ, $t_i \sim N(a_i,\sigma)$, $i=1,2$. Each player will have a probability density function associated with his belief that his partner will chicken out, $P_i(j=Chick, j\neq i)$. Initially both of these are ½, signifying that we have no prior information. But as the game develops we gain information: at each time step, $P_i(j=Chick, j\neq i) = P_i(j=Chick, j\neq i) + \eta$ for i= argmax $\{t_1,t_2\}$ and $P_i(j=Chick, j\neq i) = P_i(j=Chick, j\neq i) - \eta$ for the other. Now with this simplified game, the player with the belief, $P_i(j=Chick, j\neq i) < 1/2$ will chicken out while the other will believe he has the upper hand. The outcomes of a simulation in which $a_1 = 0.3$, $a_2 = 0.4$, $\sigma = 0.2$ is shown in Figure 8.

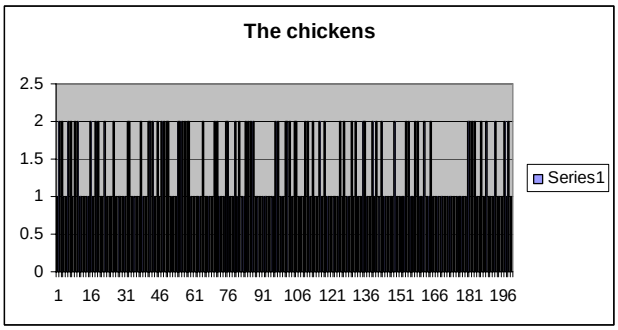


Figure 8. Mostly Player 1 was the chicken but occasionally 2 chickened out.

The probabilities, $P_i(j=Chick, j\neq i)$, were learned and are shown in Figure 9. We see that very soon, each Player has good grounds for playing chicken or not.

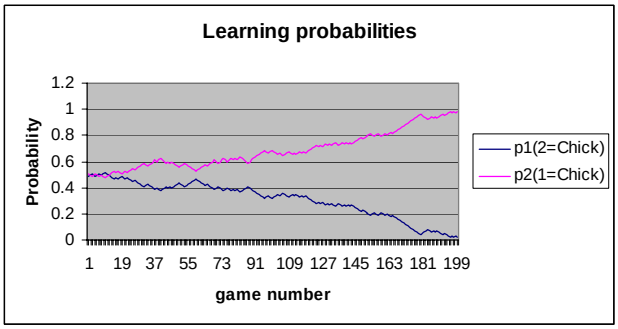


Figure 9. The probabilities learned by the two players.

However, in a full Bayesian treatment, the underlying factors would also be the subject of the investigation. Now while we do not wish to undergo a full Bayesian analysis, we recognise that Chicken is a signalling game: it is in each player's best interest to signal strongly to the other if he is not a chicken and so in a repeated human contest, one player will quickly learn the role of chicken while the other will learn the role of macho. This can easily be built into the system by adapting the latent variables (the means of the visible variables) using $a_i = a_i + \eta$ for the winner and $a_i = a_i - \eta$ for the loser. Figure 9 shows the result of such a simulation. Clearly changing the centres of the distributions gives far fewer games in which the first player wins. Also the probabilities are learned far more quickly.

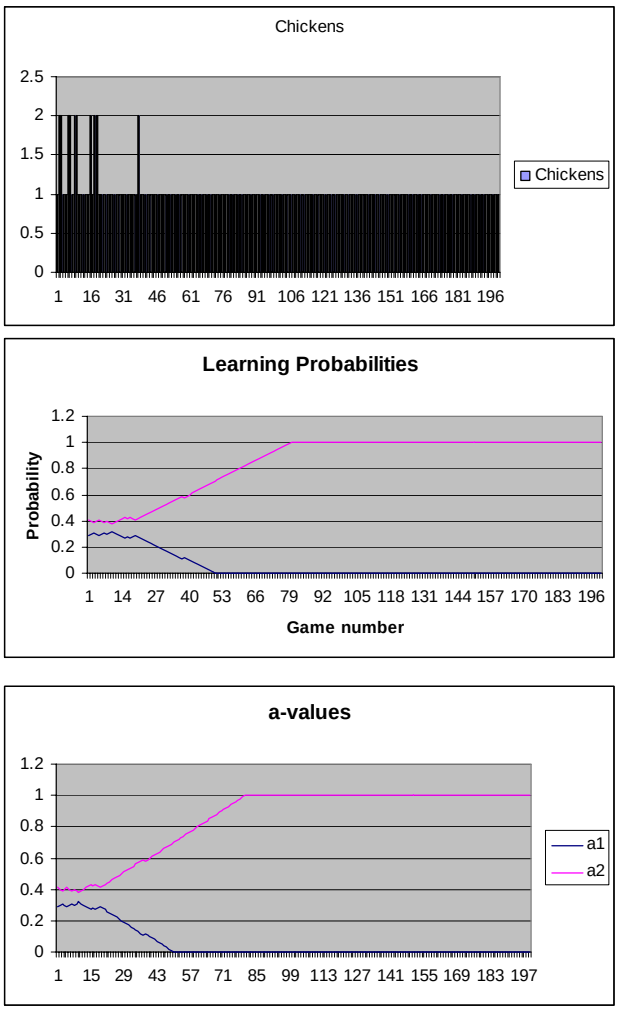


Figure 10. Top: the samples, Middle: probabilities. Bottom: the learned means of the distributions.

4. Conclusion

We stated at the outset that we consider that the search for Artificial Intelligence may not be optimal in the context of computer games. It is often easy to create a game which an AI can play better than a human: the real task is to create an AI and a game in which the AI can perform in a human-like manner. In creating our AI players, we have adopted the subjectivist probability paradigm which states that all players associate some prior probabilities to the events within the game. However, we have not gone down the full Bayesian route which would require that we update our whole probability distribution as more evidence becomes available. This may indeed be optimal for generating intelligence but we question whether such intelligence is really human-like, particularly in the context of computer games.

We have, however, incrementally changed the beliefs of our players in the computer games. We have, in this paper, used fixed values for these increments; we could argue that a sigmoidal shaped convergence of values might be more appropriate: we change our probabilities a little from the first few pieces of evidence (these may, after all, be subject to random noise), then a great deal from the next group of evidences then finally little as the remaining evidence comes in (we may wish to leave a little doubt, since many of the most enjoyable games contain a stochastic element).

Future work will extend the above latent variable models. We are particulary interested in modelling opponents' behaviour when it is affected by our own responses/behaviour which suggests that a Hidden Markov Model may be appropriate for many games.

References

- Axelrod, R. The Evolution of Co-operation, Penguin books, 1984.
- Binmore, K. Fun and Games, a Text on Game Theory, D.C. Heath and Co., 1992.
- Dixit, A. and Skeath, S. Games of Strategy, W. W. Norton and Co, 1999.
- Fyfe, C. Independent Component Analysis against Camouflage, CGAIDE 2004a.
- Fyfe, C. Dynamic Strategy Selection and Creation using Artificial Immune Systems, *International Journal of Intelligent Games & Simulation*, 3(1) 2004b.
- Fyfe, C. and Wang, T.-Z. Cooperative Population Based Incremental Learning, CGAIDE 2004.
- Fudenberg, D. and Tirole, J. Game Theory, MIT Press, 1991.
- Gintis, H. Game theory evolving, Princeton University Press, 2000.
- Jaynes, E. T. Probability Theory, the Logic of Science, Cambridge University Press, 2003.
- MacKay, D. J. C. Information Theory, Inference and Learning Algorithms, Cambridge University Press, 2003.
- Myerson, R.B. Game Theory, Analysis of Conflict, Harvard University Press, 1997.
- Wang, T.-Z. and Fyfe, C. Traffic Jams: an evolutionary investigation, IEEE/WIC/ACM International Joint Conference on Intelligent Agent Technology, IAT-2004a.
- Wang, T.-Z. and Fyfe, C. Achieving Cooperation using Artificial Immune Systems, (submitted), 2004b.

Appendix

If we have an unknown quantity, θ, which determines the actual values of n observations, $X=\{X_1,X_2,\ldots, X_n\}$ then the dependence of X on θ can be detemined by the conditional probability density function (pdf) p(X|θ). If we have a prior belief about the pdf of θ, p(θ), then we may use Bayes' Theorem to calculate the posterior distribution

$$p(\theta \mid X) \propto p(\theta) p(X \mid \theta)$$

If, in addition, p(θ|X) is a member of the same family of distributions as p(θ), this family of distributions is said to form a conjugate family. For the binomial trials being performed for the game of chicken, the unknown quantity or parameter is the probability, θ, of the opponent being chicken. For n independent trials, the number of successes is x~B(n,θ), a binomial distribution and so

$$p(x \mid \theta) \propto \theta^x (1-\theta)^{n-x}$$

If θ~Be(α,β), a Beta distribution so that

$$p(\theta) \propto \theta^{\alpha-1} (1-\theta)^{\beta-1}$$

then

$$p(\theta \mid x) \propto \theta^{\alpha+x-1} (1-\theta)^{\beta+n-x-1}$$

i.e. is of the same form as our prior distribution and so we may continually gain new evidence from repeated observations while continuing to work within the same family of distributions.

Therefore, in practice, for the simulations reported in Section 2.1, we chose to sample 8 times from the unknown distribution (i.e. n=8) and counted the number (x) of successes which we used to update our estimates of α and β in the above.

Training an AI player to play Pong using a GTM

Gayle Leen
Applied Computational Intelligence Research Unit.
School of Information & Communication Technologies
University of Paisley, Scotland
gayle.leen@paisley.ac.uk

Colin Fyfe
Applied Computational Intelligence Research Unit.
School of Information & Communication Technologies
University of Paisley, Scotland
colin.fyfe@paisley.ac.uk

Abstract- We extend the work of [McGlinchey 2003], in which the author trained an AI player to play Pong from game observation data recorded from games played by humans. The data trained a Self Organising Map (SOM) , and it was found that the AI player played Pong with a human style of play. However one of the drawback of using the SOM was that the movement of the bat was jerky, due to quantisation of vectors. The author had to applying smoothing to the AI player's bat to make the movement more realistic. It was also found that the AI players were easy to beat. In this paper we train the AI player using Generative Topographic Mapping (GTM) and we show that using the mean of the conditional density to estimate the bat's position is better than using the mode.

1 Introduction

We wish the AI player to play in a human like fashion, which will create a more enjoyable and convincing gaming experience for a human opponent. The question arises as to how to make the AI human-like. We have [Leen and Fyfe 2004a, 2004b] previously investigated this using a variety of games and a variety of computational intelligence techniques. In this paper, we use a generative probablistic method on a single game, Pong which was popular in the early days of computer games.

Previous research on this topic using the game of Pong was described in [McGlinchey 2003]. In this paper, McGlinchey used a Self-organising Map (Section 3) to learn to copy the moves of a human player in a game of pong. One difficulty with this system is that the Self-organising Map quantises the data so that the AI will respond to a Voronoi region of similar data points with exactly the same bat position; also the data includes quantities such as ball position and velocity which are continuous variables but as a variable moves from one Voronoi region to an adjacent region, the quantised data moves from one output node to the next. If each node is associated with a particular bat position, the bat will jerk from one position to the next rather than smoothly change along a real line. This is far from human-like behaviour.

It is possible to get round this difficulty by sharing responsibility for a particular set of data points (measuring ball velocity, position etc) around a set of nodes and then letting the set of nodes together deterimine the position of the bat. However, [Bishop et al, 1996] have recently developed a mapping similar to the Self-organising map known as the Generative Topographic Mapping (section 4) which automatically allocates responsibility for each data point to a number of nodes. This is inherent in the training process and requires no extra ad hoc arrangements. As a bonus, the Generative Topographic Mapping can be described as a "principled alternative to the Self-organising Map" in that it is derived from explicit probabilistic axioms and comes with an innate error model.

The remainder of the paper is as follows: in section 2, we discuss the game of pong and the characteristics which we would hope to see in an artificial pong player. In section 3, we review the Self-organising Map and in section 4, the Generative Topographic Mapping. In section 5, we show results using the GTM and give our conclusions in section 6.

2 Playing Pong

2.1 The game of Pong

The game of Pong is a simple two player game which was made popular by early games consoles. The game is a minimalist table tennis (or Ping Pong) simulator, in which each player is represented as a bat which can deflect a bouncing ball.

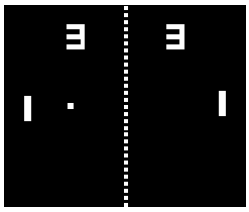


Figure 1: A screenshot from the game of Pong

Since the game is real time, and incorporates human like reactions, styles of play, and simple tactics, it is an ideal environment in which to create an AI player which exhibits human like behaviour.

2.2 Creating an AI Pong player

An ideal AI opponent for the game of Pong would be one that plays at a level similar to its human opponent, but exhibits 'human like' behaviour in its style of play. Obviously it is possible to create an AI opponent that can easily match the ability of its human opponent; the AI's bat could be constantly updated so that its centre is equal to the vertical position of the ball. However, this creates an opponent which never makes any errors and consequently would not be fun to play against, and the game would feel unfair to play from the human player's perspective. Also, the motion of the AI player's bat would not be realistic: it may move too smoothly or react too quickly to its opponent's moves. The difficulty lies in incorporating human like traits of play in the AI opponent's behaviour.

2.2.1 The issue of learning

[McGlinchey 2003] relies on Game Observation Capture (GoCap) to achieve a convincing level of human like behaviour. GoCap is the process of recording data from a live user during the execution of a game, with a view to using machine learning to train an AI player. Unsupervised learning is used to train a neural network which self organises based on the statistics of the training data.

[McGlinchey 2003] captured a data vector consisting of the ball's position, speed and direction, and the vertical position of one player's bat every frame with the game running at 60 frames / second. He then trained a Self Organised Map on the recorded data. Once the SOM had been sufficiently trained, the AI player can then play Pong by constructing an input vector based on the ball's speed, position, and direction, and inputting it into the network, and looking up the corresponding bat position from the correct component of the winning node's weight vector.

[McGlinchey 2003] found that using a SOM to train the AI resulted in erratic and jerky bat movement due to quantisation, but solved this problem by using multiple winners and interpolating between their corresponding bat positions. It was found that the AI played Pong with a human style of play but its level of play did not challenge experienced players, perhaps because the position of the AI's bat is completely deterministic based on the bat's speed, position, and direction.

We propose to extend the work of [McGlinchey 2003] by using the recorded data to train a Generative Topographic Mapping (GTM). Since this is a continuous mapping model, we expect that the motion of the trained AI player's bat should be smoother.

2.3 The difficulty of modelling Pong data

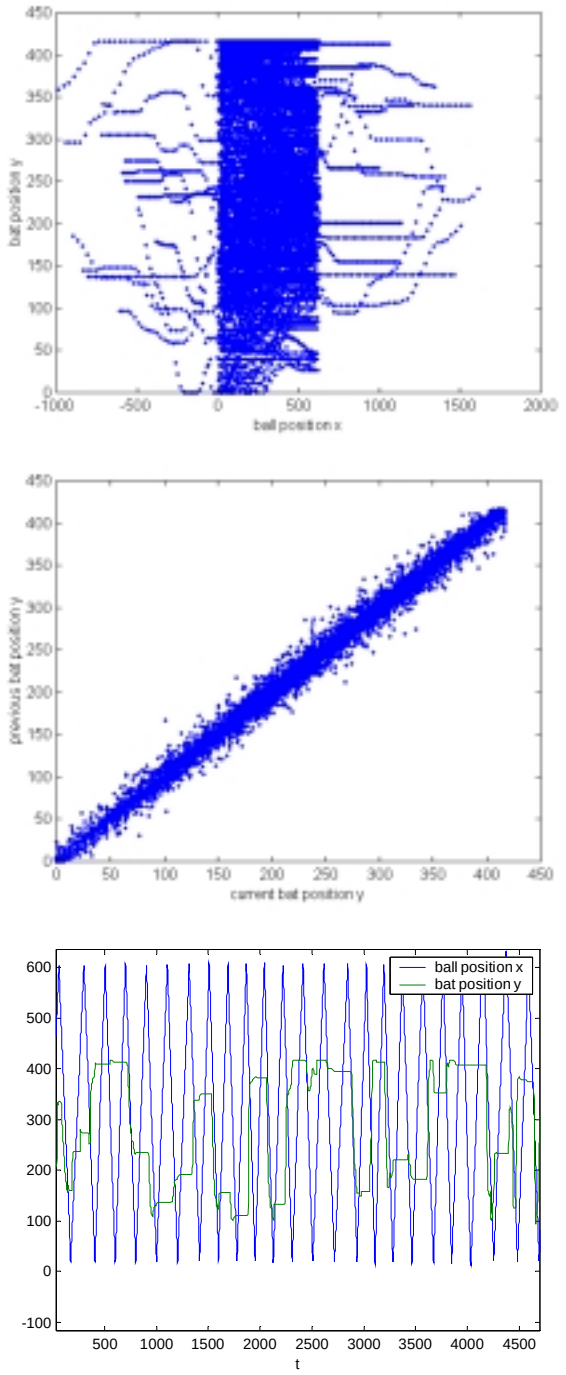


Figure 2: Examples of the recorded data set. Top: bat position against ball position, Middle: The previous bat position against current bat position, Bottom: ball and bat position against time

The problem with modelling the data set from a game such as Pong is that we have many data variables, between which there are complex mappings. For instance, for the LHS player, we would expect that at the LHS of the board (ball position x = 0) where the player returns the ball, for the player's movement to be more restricted, as opposed to when the ball is in another portion of the board; in other words we expect there to be a one-to-one mapping

between ball position x and bat position y when x is small; and a many-to-one mapping otherwise.

We find that the current bat position is always close to the previous bat position. This shows that the bat position changes very smoothly. This linear relationship can be seen in the other data variables which suggests that points in the data space which are close together will also be close in time. This suggests that the data may have an intrinsic dimensionality of 1. We can use the GTM to model the data with a one dimensional latent space.

3 The Self Organising Map

Teuvo Kohonen developed the Self-Organising Map (SOM) in 1982 as a visualisation tool for high dimensional data on a low dimensional display. A SOM is composed of a discrete array of *L* nodes arranged on a *N*-dimensional lattice and it maps these nodes into *D*-dimensional data space while maintaining their ordering. The dimensionality, *N*, of the lattice is normally less than that of the data. The SOM can be viewed as a non-linear extension of Principal Component Analysis (PCA), where the map manifold is a globally non-linear representation of the training data. As with local PCA, the data space is partitioned with each node of the map capturing a different partition. However, with the SOM, all data in a partition is quantised to a single point, and the combined effect of all of the vector-quantising nodes is to give a globally non-linear representation of the data set.

Typically, the array of nodes is one or two-dimensional, with all nodes connected to the *N* inputs by an *N*-dimensional weight vector. The self-organisation process is commonly implemented as an iterative on-line algorithm, although a batch version also exists. An input vector $\mathbf{x}$ is presented to the network and a winning node *c* is chosen whose weight vector $\mathbf{w}_c$ has the smallest Euclidean distance from the input.

$$c = \arg\min_i \left(\left\| \mathbf{x} - \mathbf{w}_i \right\| \right) \qquad \textbf{(1)}$$

So the SOM is a vector quantiser, and data vectors are quantised to the reference vector in the map that is closest to the input vector. The weights of the winning node and the nodes close to it are then updated to move closer to the input vector. The neighbourhood of node i is the set of nodes denoted by $N^{(i)}$ that are close enough to be influenced by the node i whenever it is the winner. Therefore, if the winner is c, then the weights of the nodes $i \in N^{(c)}$ will be updated during training. It may be that every node of the map is included in this set, but there can be significant savings in computational cost if a localised neighbourhood is used, especially in large maps. The amount by which the neighbours are updated is determined by the neighbourhood function, h_{ci}, which is a function of the Euclidean distance between the winner (c) and the other nodes in its neighbourhood (i). This function is normally a Gaussian or difference of Gaussians ("Mexican hat"). There is also a learning rate parameter, η, that is usually decreased as the training process progresses. The weight update rule is:

$$\mathbf{w}_i(t+1) = \mathbf{w}_i(t) + \eta(t) h_{ci}(t) \left[\mathbf{x}(t) - \mathbf{w}_i(t) \right], \forall i \in N^{(c)} \qquad \textbf{(2)}$$

When this algorithm is iterated sufficiently, the map self-organises to produce a topology-preserving mapping of the lattice of weight vectors to the input space based on the statistics of the training data. Each weight vector lies approximately at the centre of its Voronoi region, which holds the subset of points in the data space that are closer to this vector than any other in the map. Another interesting property is that in certain circumstances the mapping is approximately equiprobabilistic i.e. when a vector is chosen at random from the training set, each node has an equal probability of winning. This is true if the mapping gives a faithful representation of the data, i.e. the weight density is similar to the density of the data set. So for areas of the input space that have a high density of data, there will be more weight vectors found in that region than other regions with lower data densities. However, this does not always hold true, since there are some data distributions that may cause part of the map to become "stretched" over regions of the data space with a very low density of data, and the equiprobability is then greatly dependent on sufficient narrowing of the neighbourhood function. This finding is empirical, and it is not an objective of the SOM algorithm. Some work has been done recently where equiprobability is used as a prior, and an objective function optimises the network parameters to achieve equiprobability.

4 The Generative Topographic Mapping

The Generative Topographic Mapping [Bishop et al. 1996] is a non linear latent variable model, for which the parameters can be determined by using the EM algorithm.

4.1 Latent Variable Models

We can capture the correlation between the variables of a data set $\mathbf{t} = (t_1, \ldots, t_D)$ by modelling the data's distribution in terms of some latent variables $\mathbf{x} = (x_1, \ldots, x_L)$. We are interested in the case when the dimensionality L of the latent space is smaller than the dimensionality D of the data space, since this shows that the data has an intrinsic dimensionality which is smaller than D.

We define a joint distribution over the data and the latent variables, which can be decomposed into the product of the marginal distribution of the latent variables $p(\mathbf{x})$, and the conditional distribution $p(\mathbf{t} \mid \mathbf{x})$ of the data variables given the latent variables.

$$p(\mathbf{t}, \mathbf{x}) = p(\mathbf{x})\, p(\mathbf{t} \mid \mathbf{x}) = p(\mathbf{x}) \prod_{i=1}^{d} p(t_i \mid \mathbf{x}) \qquad \textbf{(3)}$$

We express the conditional distribution $p(\mathbf{t} \mid \mathbf{x})$ in terms of a mapping from latent space to data space $\mathbf{y}(\mathbf{x}; \mathbf{W})$. Geometrically, the function $\mathbf{y}(\mathbf{x}; \mathbf{W})$ maps the latent space into an L dimensional manifold embedded within the data space.

We complete the definition of the latent variable model by defining the marginal distribution $p(\mathbf{x})$. We can then obtain the distribution $p(\mathbf{t})$ of the data by marginalising over the latent vairables.

$$p(\mathbf{t}) = \int p(\mathbf{t} \mid \mathbf{x})\, p(\mathbf{x}) d\mathbf{x} \qquad \textbf{(4)}$$

In general, this integration will be analytically intractable except for specific forms of $p(\mathbf{t} \mid \mathbf{x})$ and $p(\mathbf{x})$.

4.2 The GTM algorithm

The Generative Topographic Mapping (GTM) uses a non linear mapping function $\mathbf{y}(\mathbf{x}; \mathbf{W})$ which is chosen to be given by a generalised linear regression model of the form:

$$\mathbf{y}(\mathbf{x}; \mathbf{W}) = \mathbf{W} \phi(\mathbf{x}) \qquad \textbf{(5)}$$

where the elements of $\phi(\mathbf{x})$ consist of M fixed non-linear basis functions.

We define the marginal distribution $p(\mathbf{x})$ as a sum of delta functions centred on the nodes of a regular grid (which is analagous to the nodes of the SOM) in latent space:

$$p(\mathbf{x}) = \frac{1}{K} \sum_{i=1}^{K} \delta(\mathbf{x} - \mathbf{x}_i) \qquad \textbf{(6)}$$

which allows the integral in (4) to be performed analytically.

We choose the distribution of $\mathbf{t}$ for given $\mathbf{x}$ and $\mathbf{W}$ to be a radially symmetric Gaussian centred on $\mathbf{y}(\mathbf{x}; \mathbf{W})$, having inverse variance β:

$$p(\mathbf{t} \mid \mathbf{x}, \mathbf{W}, \beta) = \left(\frac{\beta}{2\pi} \right)^{D/2} \exp\left\{ -\frac{\beta}{2} \left\| \mathbf{y}(\mathbf{x}; \mathbf{W}) - \mathbf{t} \right\|^2 \right\} \qquad \textbf{(7)}$$

The distribution in data space then takes the form:

$$p(\mathbf{t} \mid \mathbf{W}, \beta) = \frac{1}{K} \sum_{i=1}^{K} p(\mathbf{t} \mid \mathbf{x}_i, \mathbf{W}, \beta) \qquad \textbf{(8)}$$

with log likelihood function:

$$L(\mathbf{W}, \beta) = \sum_{n=1}^{N} \ln \left\{ \frac{1}{K} \sum_{i=1}^{K} p(\mathbf{t}_n \mid \mathbf{x}_i, \mathbf{W}, \beta) \right\} \qquad \textbf{(9)}$$

We can then find the weight matrix $\mathbf{W}$ and inverse variance β that maximise the log likelihood function, by the EM (Expectation – Maximisation) algorithm.

We can evaluate the posterior probabilities, or responsibilities, of each latent sample for every data point using Bayes' theorem:

$$R_{in}(\mathbf{W}, \beta) = p(\mathbf{x}_i \mid \mathbf{t}_n, \mathbf{W}, \beta) = \frac{p(\mathbf{t}_n \mid \mathbf{x}_i, \mathbf{W}, \beta)}{\sum_{j=1}^{K} p(\mathbf{t}_n \mid \mathbf{x}_j, \mathbf{W}, \beta)} \qquad \textbf{(10)}$$

The posterior distribution is a sum of delta functions centred at the latent grid points, with coefficients given by the responsibilities.

When considering the whole set of data points, it is convenient to summarise the posterior distribution by its mean, given for each data point $\mathbf{t}_n$ by:

$$\left\langle \mathbf{x} \mid \mathbf{t}_n, \mathbf{W}, \beta \right\rangle = \int p(\mathbf{x} \mid \mathbf{t}_n, \mathbf{W}, \beta) \mathbf{x} d\mathbf{x} = \sum_{i=1}^{K} R_{in} \mathbf{x}_i \qquad \textbf{(11)}$$

or alternatively, by the mode of the distribution:

$$i_{\max} = \arg\max_{\{i\}} R_{in} \qquad \textbf{(12)}$$

which only uses the latent point which has most responsibility for the current data point. We will investigate using both of these representations in the next section.

5 Method and experimental results

5.1 Training

The data set which we used to train the GTM was the data captured from a two player human – human game, in each frame, with the game running at 60 frames / second. Each data vector is 6 dimensional and consists of the ball's position (2 coordinates) and velocity (2 values) and speed

and the vertical position of one player's bat. The nth vector of the data set is:

$$t_n = (\text{ball pos x, ball pos y, ball vel x, ball vel y, ball speed, bat pos})$$

The data was normalised so that all fields ranged from -1 to 1.

We trained the GTM using a 1-dimensional latent space, with the assumption that the underlying variable to the data set is time. During training, the model parameters are adjusted so that the data becoming increasingly likely given the model (as seen in Figure 3). We also notice the variance decreasing as the model becomes more confident of the fit it is producing. The table below shows the joint log likelihood for all the data variables (calculated as in equation 9) per data point for different values of M, the number of Gaussian basis function, and K, the number of latent variable samples.

K (no of latent samples)	M (no of basis fns)	Beta (inverse variance)	Log likelihood / data point
20	5	24.0157	-0.4361
50	25	49.5366	0.5142
100	49	46.6166	-0.4926
400	81	54.3403	0.6905

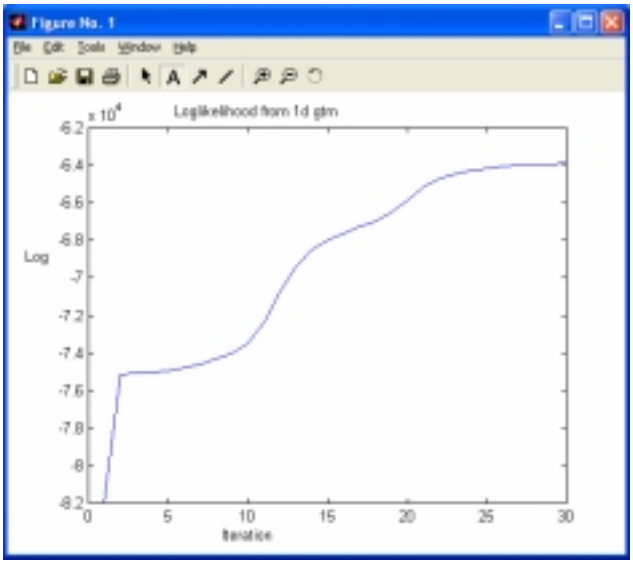


Figure 3. The log likelihood of the data according to the 1 dimensional gtm model.

We find that results are similar for different values of M and K.

5.2 Playing

5.2.1 Estimating the conditional density of the bat position given the five input variables

After training we wish to use the trained gtm to play the game of pong. To do this, it must react to the first 5 variables on the data list in order to generate an estimate of the correct value of the sixth (the bat's position). During training, we found a model for the joint density of the data variables, and now we estimate the conditional density of the bat position given the five other inputs. This can be treated as a missing value problem. We denote the nth data point as

$$\mathrm{t}_n = \left(t_n^o, t_n^m\right)$$

with t_n^m denoting the missing data variable (the bat position) and t_n^o the nth set of observed data variables (the other five inputs such as ball position x and y etc)
We wish to find the conditional density:

$$p(t_n^m \mid t_n^o, \mathbf{W}, \beta) = \int p(t_n^m \mid \mathbf{x}, \mathbf{W}, \beta)\, p(\mathbf{x} \mid t_n^o) d\mathbf{x} = \frac{1}{K}\sum_{i=1}^{K} R_{in}\, p(\mathbf{t}_n^m \mid \mathbf{x}_i, \mathbf{W}, \beta) \quad \textbf{(13)}$$

where

$$p(\mathbf{x} \mid \mathbf{t}_n^o) = \frac{1}{K}\sum_{i=1}^{K} R_{in}\,\delta(\mathbf{x} - \mathbf{x}_i)$$

(and R_{in} is the responsibility of latent point i for the nth data point's observed variables t_n^o)

For the nth data point, the density of the 'missing' data variable in data space becomes

$$p(\mathbf{t}_n^m \mid \mathbf{t}_n^o, \mathbf{W}, \beta) = \frac{1}{K}\sum_{i=1}^{K} R_{in}\left\{\left(\frac{\beta}{2\pi}\right)^{D/2} \exp\left\{-\frac{\beta}{2}\left\|\mathbf{y}_i^m - \mathbf{t}_n^m\right\|^2\right\}\right\} \quad \textbf{(14)}$$

where $\mathbf{y}_i^m$ is the missing variable's dimension of the Gaussian centre in data space generated by the ith latent point

5.2.2 Estimating the bat position

Once we have the conditional density of the bat position given the other five input variables, we have to find a way to estimate the bat position from this density.

We can calculate the bat position by finding the mean of the conditional density for each set of observed variables. We find the expected value of t_n^m given the nth observed data point by:

$$E\left\{\mathbf{t}_n^m \mid \mathbf{t}_n^o\right\} = \int \mathbf{t}_n^m\, p(\mathbf{t}_n^m \mid \mathbf{t}_n^o, \mathbf{W}, \beta) d\mathbf{t}_n^m$$

or

$$E\left\{\frac{1}{K}\sum_{i=1}^{K} R_{in}\left\{\left(\frac{\beta}{2\pi}\right)^{D/2}\exp\left\{-\frac{\beta}{2}\left\|\mathbf{y}_i^m-\mathbf{t}_n^m\right\|^2\right\}\right\}\right\}$$
$$=\frac{1}{K}\sum_{i=1}^{K} R_{in}E\left\{\left(\frac{\beta}{2\pi}\right)^{D/2}\exp\left\{-\frac{\beta}{2}\left\|\mathbf{y}_i^m-\mathbf{t}_n^m\right\|^2\right\}\right\} \quad \textbf{(15)}$$
$$=\frac{1}{K}\sum_{i=1}^{K} R_{in}\mathbf{y}_i^m$$

Alternatively, we can use the mode of the conditional density.

$$\max_{\mathbf{t}_n^m} p(\mathbf{t}_n^m \mid \mathbf{t}_n^o, \mathbf{W}, \beta) \quad \textbf{(16)}$$

We can approximate this by finding the latent point which has most responsibility for the observed variables, and assume that this latent point is most likely to have generated the complete data point. We then map this latent point into data space along the dimension of the missing variable, to find the most likely bat position:

$$E\left\{\mathbf{t}_n^m \mid \mathbf{t}_n^o\right\} = \max_i\left\{R_{in}\right\}\mathbf{y}_i^m \quad \textbf{(17)}$$

Figure 4 shows the estimated bat position from the mean of the conditional density of the bat position given the input variables. In this simulation, we used K=50, M=25. As can be seen, the overall shape of the estimated bat movement is similar to the actual bat movement; however the trained GTM does not capture those sections where the player is 'waiting' for the ball. As this is an important part of a player's style, in general the artificial AI player is dissimilar to the real player (due to the difference on a local level).

This difference in behaviour could be due to having a multimodal distribution over bat positions; as we estimate the bat position by selecting the mean of the distribution, we average between more than one possible bat position which differs from the actual bat position.

We could try and address this issue by taking the mode of the conditional density; however we would expect the bat transitions to be jerky (similar to the results gained with the SOM), as can be seen in Figure 5.

The mean of the conditional density gives a more smoothly varying representation than that found using a single point.

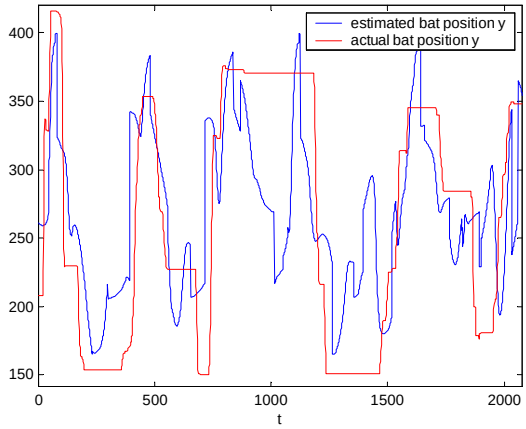


Figure 4: Estimating the bat position using the mean of the conditional density of bat position given input variables

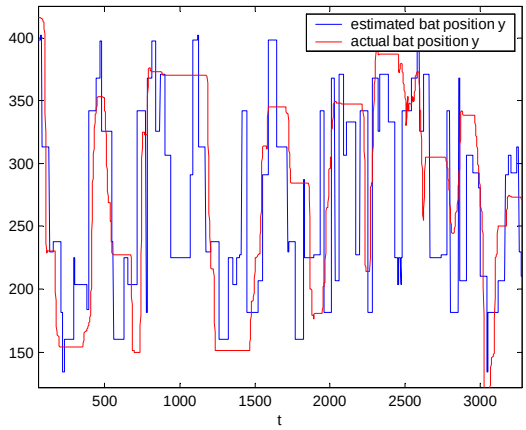


Figure 5: Estimating the bat position from the mode of the conditional density of the bat position given the input variables

6 Discussion

Our motivation for beginning this study was to try to create a more smoothly varying mapping than that achieved by the SOM. The GTM can certainly do this: it automatically allocates responsibility for data points to points in a latent space. However our results also suggest that such topology preserving methods may have other faults when used in this way.

It is known that the SOM does not always provide a quantisation of a data set which captures the underlying probabilties in data set. Some regions of the data space will be allocated more reference vectors than their probabilty mass suggests they deserve while inevitably other regions may be under-represented.

The creators of the GTM suggest that using the most responsible latent node as well as the weighted sum of latent nodes give alternative representations of a data set. We have shown that, in the context of quantising a game space, the weighted sum of latent nodes gives a far better spread of estimates (modelling of the joint density) in the data space.

The simulations presented in this paper demonstrate the difficulty in generating a probabilistic model in the context of game playing. We found that the model adapted to the statistics of the training data, from which we could calculate a conditional density that could estimate the bat position trajectories on a 'global' level, but not on a local level. This may be due to the fact that we are estimating a conditional density from a joint density, and we are modelling unnecessarily modelling correlations between the input variables, rather than the relationship between input and output. A possible solution to this would be to estimate the conditional density directly.

Other work will investigate a competition between AIs trained with the SOM algorithm and AIs trained with the GTM algorithm, both in terms of which is the most likely to win and, more importantly, in terms of which is most difficult to distinguish from a human player [Livingstone and McGlinchey 2004]. We also wish to investigate whether other refinements of the GTM specific to game playing might be developed.

Acknowledgments

The authors would like to thank Stephen McGlinchey for providing them with the data set which he used with the SOM and for insightful comments on the application of the GTM to this data set.

Bibliography

Bishop, C. M., Svenson, M., and Williams, C. (1996). *GTM: The Generative Topographic Mapping.* Neural Computation. **10**. 1. pp 215--235.

Kohonen, T. (1982). *Self Organising Maps*. Berlin: Springer-Verlag

Leen, G. and Fyfe, C. *Agent Wars with Artificial Immune Systems,* 3rd International Conference on Entertainment Computing, ICEC2004, 2004a.

Leen, G. and Fyfe, C. An investigation of alternative planning algorithms: Genetic algorithms, artificial immune systems and ant colony optimisation, Conference on Computer Games: Design, AI and Education, CGAIDE2004, 2004b.

Livingstone, D. and McGlinchey, S. *What believability testing can tell us,* CGAIDE 2004.

McGlinchey, S. J., (2003). *Learning of AI players from Game Observation data*. Proceedings of Fourth International Conference on Intelligent games and Simulation, 2003, p106-110 ISBN: 90-77381-05-8

Nannon™: A Nano Backgammon for Machine Learning Research†

Jordan B. Pollack
Computer Science Department
Brandeis University
Waltham, MA 02454
pollack@cs.brandeis.edu
http://demo.cs.brandeis.edu

Abstract- A newly designed game is introduced, which feels like Backgammon, but has a simplified rule set. Unlike earlier attempts at simplifying the game, Nannon maintains enough features and dynamics of the game to be a good model for studying why certain machine learning systems worked so well on Backgammon. As a model, it should illuminate the relationship between different methods of learning, both symbolic and numeric, including techniques such as inductive inference, neural networks, genetic programming, co-evolutionary learning, and reinforcement learning based on value function approximation. It is also fun to play.

1 Introduction

Backgammon is an ancient game which is still popular in many parts of the world. Although it is based on a lucky device - the roll of dice to limit each player's moves - humans have discovered a wide range of strategies and skills, filling up many books with acquired backgammon knowledge, both folk and mathematical (Jacoby 1970, Magriel 1976). Its popularity soared in the US with clubs and pub tournaments in the late 70's, and it is growing in popularity again, online.

Backgammon has also become an object of study for computational gaming, as a stochastic rather than deterministic game like chess. However, the difficulty of coding all the arcane rules, especially regarding forced moves and bearing off - makes computer logic for the game run to several pages of impenetrable logic which is difficult to fully debug. Also, the breadth of the game tree prohibits deep look ahead, because rolling doubles, which allow 4 checkers to move, causes combinatorial explosion.

Nevertheless by the mid seventies it was possible to write backgammon programs on that era's IBM 360 computers. Such a player could make reasonably proficient moves. It comprised a legal move generator, a set of measurement and position testing functions, and parameter based methods to rank positions based on rough heuristics for determining game phase.

One of the earliest published computer players was built by Hans Berliner (1977). His player was similarly based on a set of hand-built polynomials over measurements of positions, as well as logical functions to determine which "phase" of a game the player was in; However, Berliner went further to include smoothing mechanisms after noticing that the computer player could be exploited as it wavered between strategic boundaries. With further work, his BKG became a respectable computer player for humans to train against.

Backgammon next became a domain for scaling up neural network learning, e.g. back Propagation (Rumelhart Hinton & Williams, 1986). Gerald Tesauro wrote a series of influential papers on training back-propagation networks to become value estimators for backgammon positions. A player can be made by combining a value estimator with a greedy algorithm which looks at all possible moves for a given dice roll, and picks the highest scoring position for the current player. His early Neurogammon approach used encyclopedic tables drawn from human tournaments. Later, it was extended with contrast-enhancing techniques (Tesauro 1987, 1989). Then, in 1992, using large scale computing power provided by IBM Yorktown Heights, he published a breakthrough paper on learning backgammon via self-play using the method of temporal differences. (Sutton 1989, Tesauro 1992). After manually increasing the set of primitive features, and using multi-ply search, TD-gammon was recognized as one of the top players in the world. (Tesauro 1995). The success of TD-gammon stimulated a lot of research in Reinforcement Learning for the rest of the decade, as well as drove acceptance of commercial programs providing analysis and challenge for professional gambling and tournament play, such as Jellyfish and Snowie.

Our work on co-evolutionary algorithms began in the early 90's (Angeline & Pollack 1993) as part of a search for clear evidence that software could be a medium for the kind of open-ended evolution of complexity seen in the

† Nannon is a copyrighted game, but may be used for research and academic purposes. Nannon is a trademark of Nannon Technology corp., which provided permission to publish the rules and board in this paper.

"arms-race" phenomena in Nature. In co-evolution, learners face a dynamically changing environment, usually composed of other learners, such that as some improve, the challenges for others would automatically increase. Besides Hillis's work on Sorting networks, Axelrod's IPD GA experiment, and Ray's Tierra model, we considered Tesauro's TD-Gammon to be indicative of successful Co-evolution, since it improved by essentially increasing the difficulty of the learning environment as it progressed. However, the fact that it used a population of 1 caused some cognitive dissonance because most co-evolutionary systems based on Genetic Algorithms or Genetic Programming used populations in the 100's.

In 1998, Alan Blair and I did a small experiment based on a validated backgammon legal move generator provided by Mark Land. We used 1+1 hill-climbing on a neural network as a value estimator. Using the current network as Champion, we added random noise to the weights and had it compete against the champion. Despite the simplicity of this algorithm, we substantially replicated the co-evolutionary learning effect of TD-gammon, although our player was not as good as the ones derived by Tesauro.

In that and subsequent work, we started asking the question: what is it about backgammon, which makes complex learning possible? Learning in the backgammon domain has far exceeded success in other games which seem much easier to learn, such as TicTacToe and Othello. The Backgammon success has not been replicated in harder games like Chess and Go, although Fogel (2002) reports intriguing results in checkers.

One approach is to try to change other tasks to be more like backgammon in order to achieve better learning, for example, adding randomness to chess. Another approach is to find a simpler problem to study. A new kind of very simple game, called the Numbers game, has been valuable in illustrating co-evolutionary dynamics (Watson & Pollack, 2001; DeJong & Pollack 2002). However, the numbers game doesn't lead to the acquisition of any knowledge or strategy.

What we realized would be needed is a simpler version of Backgammon. Tesauro started the work in TD-Gammon by simply learning to bear off from an end game position. However, learning this subgame doesn't transfer much knowledge to the full game. There are other hopeful variants of Backgammon, such as Trouble, where children race in the same direction using 4 pieces but no blocking, and Hypergammon, using 3 pieces but the full rule set, however these simplifications of the game basically turn into luck-driven races with little strategic content or the volatility we think of as *turnaround dynamics*.

Backgammon, besides the balance between luck and skill, is different from games with random elements like Monopoly or Risk, which early advantages lead to winner-take-all. In Backgammon, specific dice rolls can quickly turn a game from favoring one player to the other. It is also "mixed motive" in that Humans develop symbolic strategies involving recognizing whether to play offensively or defensively, balancing competing goals to block, contain, hit, and run.

Our hope for a small game would be one which maintains all the elements of Backgammon including:

- A random element
- Turnabout Dynamics
- Occasional forfeited and forced moves
- No Draw or Stalemate possible
- Complex strategy with mixed motives
- No first player advantage.

Such a game should have an easy-to-write legal move generator, should allow researchers to compare various machine learning techniques, should allow the development of some notions of optimal play against which to measure success.[1] A simpler game should require less computer resources for study, broadening the number of researchers involved, leading to a deeper understanding of why certain kinds of learning work. In particular, we are interested in the relationship between co-evolution, reinforcement, and dynamic programming, as well as the historic division between knowledge-based symbolic learning and numeric-based control of behavior.

2 Introducing Nannon

Nannon is a new game that was invented to meet these goals. Its rules and conditions were chosen to minimize complexity, maximize strategic choice, maintain volatility, and remove any first player advantage. First, consider using only one random number (instead of two), providing only 2, 3, or 4 checkers (instead of 15) per player and using a board from 4 to 12 spaces long (instead of 24).

Because of the availability of 6-sided dice, I settled on a 6-point board, with 3 checkers per side, although the game admits a whole family of games of related sizes and different dynamics. Like backgammon, players move in opposite directions, with a goal of getting all checkers off the board and out of play, while hitting their opponents back to the beginning to start over.

The game starts in an initial position, then each player takes a turn by rolling a die and if possible, moving one of their checkers the number of steps shown on the die, or off

[1] Hypergammon admits a 200 Megabyte table of positions calculated by GNUBG which allows for value function to be approximated in several days of CPU time.

the board to safety. The initial position was chosen to increase strategic interaction.

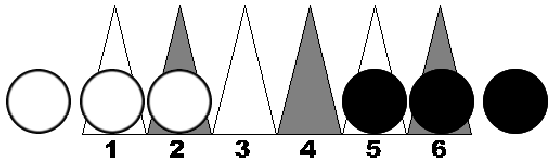


Figure 1: The initial position where White is moving right, Black is moving left.

The goal is to get all one's checkers across the board and out of play ("to safety"), but like in backgmmon, intermediate goals are hitting and blocking your opponent, overcoming the luck of the die with strategic choices. Consider if black rolls a 2, and moves the piece from the 5 to the 3 position:

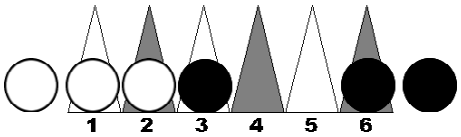


Figure 2: Black rolls a 2 and makes a bad move, exposing two men instead of preserving a prime.

Hitting means landing a checker on an opponent checker and sending it back to the beginning ("home").[2] If white rolls a 3, the player can move onto the board, hitting back the black piece (to the "7" position).

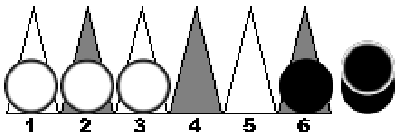


Figure 3: White rolled a 3 and hit Black back to the 7.

With only 3 checkers, our core realization was that since a "point" in Backgammon requires two checkers on a space, and blocking requires several or even 6 points in a row, any reduced checker game with the full rules cannot maintain blocking, which is a core strategic element of Backgammon.

So how can blocking be brought back into the reduced game? The answer we arrived at is to use *adjacency* to create a block. If a player can locate two or three checkers next to each other, we declare the other player cannot land on or hit those checkers. Therefore, the 3 white checkers above protect each other from getting hit, and block black from moving on certain rolls.

Three checkers blocking one checker would cause a forfeited turn only 50% of the time[3]. In the position of Figure 3, Black would forfeit 33% of the time, upon rolling a 4 or 5. (Black's checker on the 6 can move to safety with a 6.)

We made a second important rule decision which simplified the board representation. We decided that only one checker ever allowed on a space - e.g. no stacking of checkers at all! But what happens if the dice would allow one checker to land on another? There are 3 alternatives rules: It cannot move, it skips forward (which accelerates the game), or you are forced to hit yourself (which is quite odd!). We chose the simplest idea; a checker cannot land on another checker of the same color. Thus, in the current position of Figure 3, White rolling a 2 cannot stack the checker from the 1 to the 3 position, but must move from the 2 or 3 point.

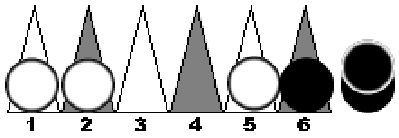


Figure 4: White moved 2. If Black rolls a two and hits the White checker on the 5-point, the game cycles back to the initial position.

This no-stacking rule simultaneously increased the effectiveness and importance of blocking, created forced bad rolls which break up blocks, and made the legal move generator extremely simple, as seen in the Matlab and Java examples given in the Appendix.

To find legal moves for a player, we first compute which of the 6 board positions are blocked by either a player's own checkers, or by opponent checkers which are adjacent. Then we simply calculate which of the player's 3 checkers still in play can land on a non-blocked space or escape off the board.

Of course, developing over thousands of years, Backgammon has many rules which control the emergent issues that arise during the game. For example, you need to have all the pieces off the bar in order to move any other piece, you need have all checkers in the Home quadrant before bearing off, and you have to move the highest number if you have a choice of forced moves between two dice. These rules are unnecessary or lead to stalemate in Nannon.

2.1 Starting Position and First Player Advantage

We represent a position as two sorted triples, of the locations of each player's checkers. The board positions are 1-6, and we use 0 to represent player 1's home and

[2] In backgammon home would be called "the bar", and no other pieces can move when any piece is on the bar. This rule doesn't make sense for Nannon.

[3] However, if we considered a rule to make a 3-point prime completely block the other player, we would end up with stalemates, which are undesirable

player 2's goal, and 7 to represent player 2's home and player 1's safety. Switching viewpoints consists of reversing the vector and subtracting it from 7. An alternative computer representation is to represent each player as a bit string using 6 bits for the location of the checkers on the board, and two or three bits to count the number of checkers which are off the bar.

We found that the default home position [000 777] was not satisfactory as it gave an overwhelming (60%) advantage for the first mover, and many games with no strategic interaction.

So the final issue in designing the game was reducing this first player advantage and increasing interaction. We looked at a variety of opening positions and rules to balance the game. We found that a starting position of [012 567] increased interaction.

3 Analysis of the game

Using both random play and the expert play after value function approximation, we now show that the goals for a reduced backgammon like game are satisfied.

3.1 Size of the game

The number of possible board states is given by the following equation, where n is the number of spaces on the board, and k is the number of checkers per player:

$$\sum_{i=0}^{k}\sum_{j=0}^{k}\binom{n}{i}\binom{n-i}{j}(k+1-i)(k+1-j)$$

Consider placing i=3 checkers of player 1, and j=1 checker of player 2 on a 6 point board, leaving 2 of player 2 checkers off the board. There are $\binom{6}{3}$ ways of placing the first 3 checkers, $\binom{6-3}{1}$ ways to place the 1 checker of the second player, and 3 ways to allocate the two remaining player 2 checkers to either home or safety.

For the 6-position, 3-checker game, this works out to 2530 states, although in practice the state where both players have 3 checkers to safety cannot be reached.

By comparison, using 3 checkers on an 8, 10 or 12-point game have 9784, 31426, and 86148 states respectively. Using 6 checkers each on a 12-point board creates a rare stalemate possibility within its 4,203,123 states. Nannon is really a parameterized set of backgammon like games.

3.2 No First Player advantage

Even though the raw starting position has 57% equity for player one, rolling a 4 sided die (no 5 or 6 on opening) drops the advantage to 53%. Subsequently we found a new initial roll: Both players roll their dice, and the winner gets a first roll based on the difference between the dice (e.g. 6-4=2). This lowers the retries from 1/3rd to 1/6th of the time and is fair to both players. The initial roll is biased in that 1/3rd of the time it results in a "bad" 1, and 1/15th of the time it gets a "good" 5.

Although it is a short game, and each dice roll is meaningful, the initial position and dice roll makes it so it so that the first player has no significant advantage, at 51.5%.

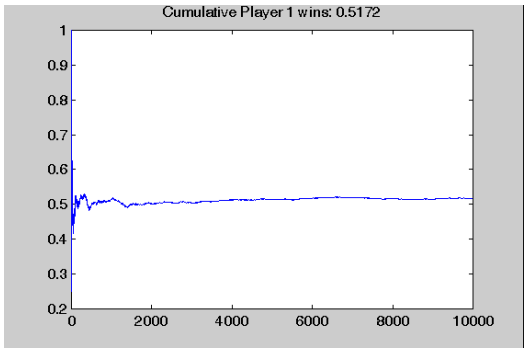


Figure 5: In 10,000 games between optimized players, Player 1 wins about 51% of the time.

3.3 Turnabout Dynamics maintained

One of the critical issues in reduced backgammon games is the loss of the volatility, or turnabout dynamics; this unpredictability about which player is going to win is essential to the popularity of the game, as it is to sports like Basketball and Soccer. Nannon allows games to reverse almost until the final few rolls. This can be seen in the following analysis of 10,000 games. We calculated the equity of player 1 at every move and count the times per game the first player equity crosses zero. Only 20% of the time does an initial lead carry through.

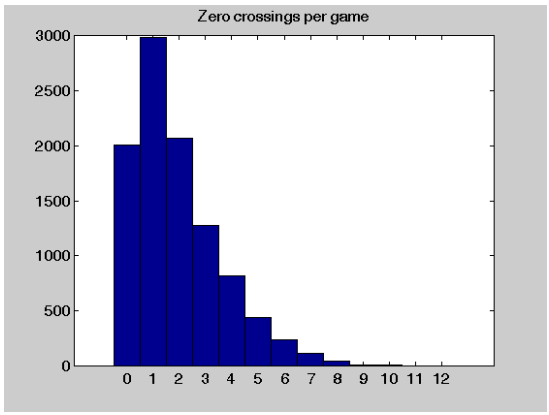


Figure 6: Volatility is shown by the number of times the expected winner changes across a game. Calculated in 10,000 games with optimized players. X-axis is the number of flip-flops per game; Y-axis is the number of games out of 10,000.

3.4 Length of game

Nannon is a fast game, with a mean of 13 rolls to completion, although long games up to 38 rolls have been observed. This enables 10's of games per second to be evaluated in a high level language like Matlab or Lisp, and 1000's in a compiled language like C or Java. Figure 7 shows the length of games out of 10,000.

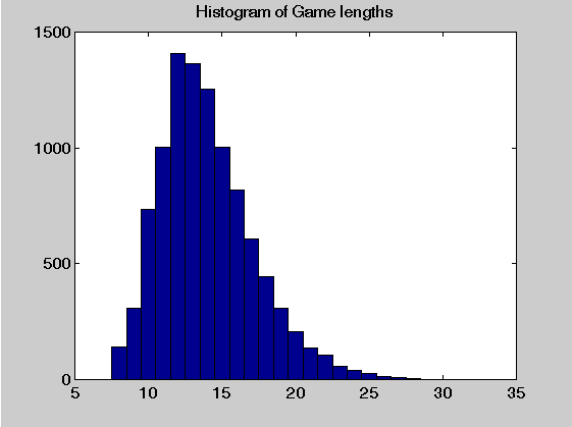


Figure 7: Histogram of game length.

3.5 Balance between Luck and Skill

Over a number of games, we calculate how many times each player forfeits a move, is forced by the die to make a specific move, or has 2 or 3-way choice. Just under 50% of the moves involve choice, as shown in the pie chart below.

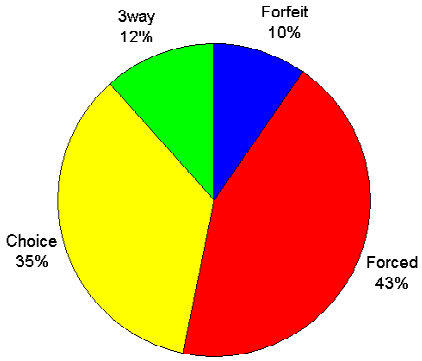


Figure 8: Almost 50% of the time, players have a strategic choice between two and three checkers. Forfeited rolls occur when an opponent has adjacent checkers (a prime). Forced moves occur mostly when a player has only one checker left.

3.6 Learnable using value function approximation

The game falls under the Bellman (1957) equation, which means there is theoretically an optimal sequential control policy based on a converged expected value for each state. The value of any state is the utility (or equity in backgammon terms) based on fair dice and future optimal play by both players. Each position can be assigned a value, and a strategy for play is simply the greedy algorithm, which looks at all moves enabled by the roll of the die and chooses the one with maximum likely reward for the current player. This is the same way that a neural network value estimator like TD-Gammon is turned into a player. Learning the symbolic rules for a game remains a hard problem.

Calculating the value function is given for ending positions – E.g. 0 or loss and 1 for win is trivial[4]. For position in a racing game, after no more contact or hitting is possible, calculating the value functions is a simple recursive application of dynamic programming. However there is a large set of positions that enable hitting to form cycles, which lead to a large system of unknowns. In many real world applications, the number of possible states is too high, but for Nannon (with a 6 point board and 3 checkers each), there are only 2530 possible positions making value function approximation eminently practical. For each state of the game, either it is an end state or we update its value by looking ahead under all dice roles for the opponent's optimal response, and multiply it by the probability of the die roll (e.g. $1/6^{th}$).

Starting with the end game positions labeled as 0 or 1, and with values exactly solved for the racing states where no further hitting is possible, in 15 passes across the 2530 states, the sum of the square of difference between values before and after each iteration rapidly dropped to 10^{-7}.

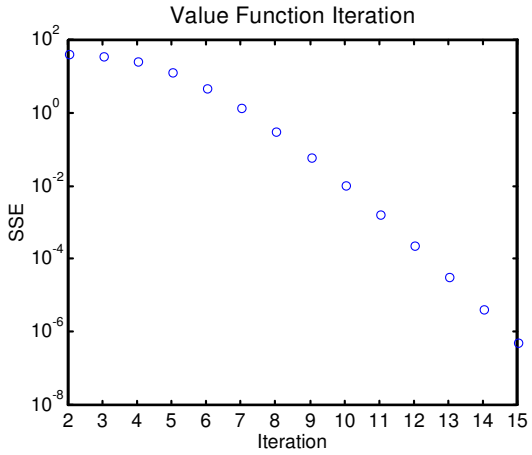


Figure 9: Convergence of VFA in Nannon leads to an optimized player.

[4] In actual play, the use of doubling, "gammons", and tournament rules, complicates the value calculation.

4 Conclusions

Although we have not yet done a wide range of machine learning experiments on the Nannon game besides value function approximation and simple heuristics based on Maslow's "Hierarchy of Needs," (like Always go to safety, then Always Hit, then Always keep block) there are many more experiments and comparisons which can be done across learning methods using this game as a model.

For example, the game can be subject to genetic programming, co-evolutionary learning, neural networks, TD learning and other reinforcement methods related to dynamic programming, as well as symbolic techniques such as Inductive inference or Inductive Logic Programming.

Backgammon, in this simpler form of Nannon is a perfectly sized test problem which ultimately could shed light on the old computational intelligence issue of whether cognition is analog and numeric based on associationism and control theory, or digital and symbolic based on universal computation.

Certainly as humans play such a game, they discuss symbolic strategies regarding when to hit, when to run, when to keep a prime versus losing tempo and so on. As expertise develops, the symbolic is infused with more statistical and numeric models to aid decision-making. Yet, according to the theory of sequential choice developed by Bellman, a greedy policy based on the converged value function should be the top player in the world (assuming fair dice).

Perhaps as our understanding of consciousness has evolved to realize that the narrative is just a story our mind constructs to explain our complex behavior based on diffuse and physical complex processes of our brains (Dennett 1991), perhaps the symbolic rules of a game is also just a story we tell as our biological organs adapt to optimize utility.

Acknowledgements

Dylan Pollack and Brad Rosenberg helped play the first few games. Anthony Bucci supplied the Java legal move code. Michael Daitzman provided much moral support and user testing. Thanks especially to Michael Littman for a discussion on VFA one evening.

5 References

Angeline, P. J. & Pollack, J. B. (1993) Competitive environments evolve better solutions to complex problems. Fifth International Conference on Genetic Algorithms. 264-270.

Robert M. Axelrod (1987) The evolution of strategies in the iterated prisoner's dilemma. In Genetic Algorithms and Simulated Annealing, chapter 3, pages 32--41. Morgan Kaufmann.

Hans J. Berliner (1977) Experiences in Evaluation with BKG - A Program that Plays Backgammon. IJCAI 428-433

De Jong, E.D. and J.B. Pollack (2004) Ideal Evaluation in Coevolution, Evolutionary Computation, Vol. 12, Issue 2, pp. 159-192

Dennett, D.C. (1991) Consciousness Explained. Boston: Little, Brown.

Fogel, D. B (2002) BLONDIE24: Playing at the edge of AI. San Francisco: Morgan Kaufmann.

Hillis, D. (1992). Co-evolving parasites improve simulated evolution as an optimization procedure. In Alife II: Proceedings of the 2nd International Conference on Artificial Life. Addison-Wesley.

Paul Magriel. (1976) BACKGAMMON, New York: Times Books

Oswald Jacoby & John R. Crawford. (1970) The Backgammon Book. New York: Viking.

Pollack. J B. & Blair A. (1998). Co-Evolution in the Successful Learning of Backgammon Strategy. Machine Learning, 32, 225-240.

Ray, T. S. (1991), An approach to the synthesis of life. In : Langton, C., C. Taylor, J. D. Farmer, & S. Rasmussen [eds], Artificial Life II, Santa Fe Institute Studies in the Sciences of Complexity, vol. XI, 371-408. Redwood City, CA: Addison-Wesley.

Rumelhart, DE, Hinton, GE, and Williams, RJ (1986) Learning representations by back-propagating errors. Nature, 323, 533-536.

Sutton, R.S. (1988) Learning to predict by the methods of temporal differences. Mach. Learning 3, 9-44.

Tesauro, Gerald (1992), Practical Issues in Temporal Difference Learning, Machine Learning 8, 257-277.

Tesauro, Gerald (1995) Temporal Difference Learning and TD-Gammon, Communications of the ACM, March 1995, 38(3):58-68.

Tesauro (1989): "Connectionist learning of expert preferences by comparison training", Advances in NIPS 1, 99-106.

Backgammon Varieties (2004) http://www.bkgm.com/rgb/rgb.cgi?view+846

Watson RA & Pollack JB, (2001), "Coevolutionary Dynamics in a Minimal Substrate", in GECCO-2001: Proceedings of the Genetic and Evolutionary Computation Conference. Spector, L, et al, editors. Morgan Kaufmann, 2001.

Appendices

5.1 Legal move generator in MATLAB

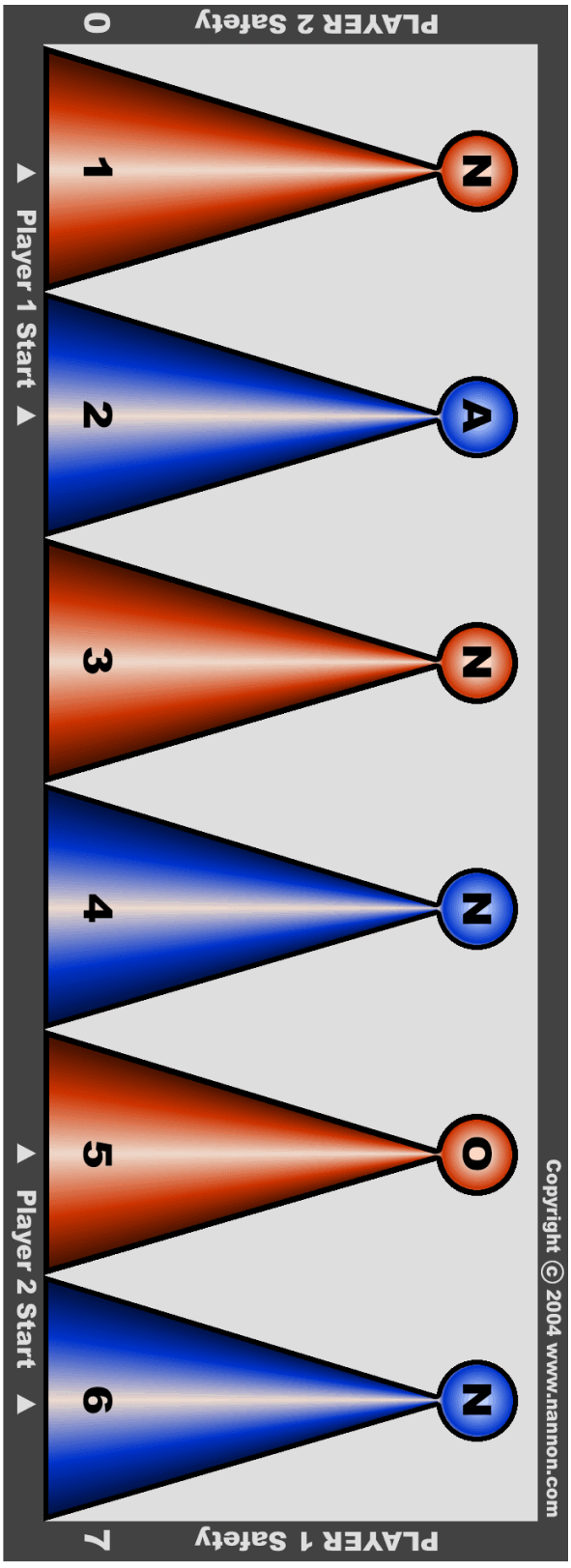
```
function moveable=legmove(pos,die)
% pos is a sixtuple [p1 p1 p1 p2 p2 p2]
% each from 0 to 7, each triple sorted
% output is a bitvector for moving
pos(1:3)
% assumes player 1 to move

moveable=zeros(1,3);
blocked=zeros(1,7);%blocked(7) is
always 0

%block adjacent opponents
%remember that pos(4 5 6) are sorted
if pos(4)<6 & pos(4)>0 &
pos(4)+1==pos(5) blocked(pos([4
5]))=1;end;
if pos(5)<6 & pos(5)>0 &
pos(5)+1==pos(6) blocked(pos([5
6]))=1;end;

%block my own checkers on the board
for i=1:3
   if mod(pos(i),7)
blocked(pos(i))=1;end
end
%Calculate unique unblocked moves
for j=1:3
   if pos(j) ~= 7 % once in safety
don't move
      if j==3 | pos(j) ~= pos(j+1)
%stop duplicate 0 choices here
         if
~(blocked(min(7,pos(j)+die)))
moveable(j)=1;
         end
      end
   end
end
```

5.2 Printable Board



5.3 Legal move generator in Java

```
/*
   we store the board in two ints,
   m_black and m_red which look like
this
   (take careful note of the indexing;
red is indexed
   backwards w.r.t. black);

   b b b 0 B B B B B B 0
   0 1 2 3 4 5 6 7 8 9 10

   b is the home
   B is the board
   0 are for efficient legal move calc.

   we'll index this in two ways: with
pos and with idx
   (position and index, resp).  idx
indexes the bits in
   the int, so starts from 0 and runs
to NUM_PIECES + BOARD_SIZE + 2 - 1
   (2 for the pads).  pos indexes the
board, starting from 0
   and running to BOARD_SIZE - 1.
negative positions indicate the
   bar; -1 is the 0, -2 is the bar, -3
is the bar, etc.*/

public class Board {
    // handy constants
    public static final int NO_ONE = -
1;
    public static final int BLACK = 0;
    public static final int RED = 1;
    public static final int NUM_PIECES
= 3;
    public static final int BOARD_SIZE
= 6;

    // board state
    int m_black;
    int m_red;
    int m_whoseTurn;
    int m_nMoves;

    public boolean isLegal(int
nFromPos, int nDie) {
        int me = m_whoseTurn == BLACK ?
m_black : m_red;
        int opp = m_whoseTurn == BLACK
? m_red : m_black;
        int nFromIdx = nFromPos +
NUM_PIECES + 1;
        int nToIdx = nFromPos < 0 ?
nDie + NUM_PIECES : nDie + nFromIdx;
        int nToPos = nToIdx -
NUM_PIECES - 1;

        if(nFromPos < 0) {
            for(int i = NUM_PIECES-1 ;
i >= 0 ; i--) {
                if( (me & (1<<i)) != 0
) {
                    nFromIdx = i;
                    nFromPos = i -
NUM_PIECES - 1;
                    break;
                }
            }
        }

        if(nFromPos >= BOARD_SIZE)
return false;

        if( (me & (1<<nFromIdx)) == 0 )
return false;

       if(nToPos >= BOARD_SIZE) return
true;

        if( (me & (1<<nToIdx)) != 0 )
return false;

        int nOppToIdx = BOARD_SIZE +
2*NUM_PIECES + 1 - nToIdx;
        if( (opp & (1<<nOppToIdx)) != 0
&& ( (opp & (1<<(nOppToIdx+1))) != 0 ||
(opp & (1<<(nOppToIdx-1))) != 0 ) )
return false;
        }



    public int[] getLegalMoves(int
nDie) {
        Vector v = new Vector();
        for(int pos = -1 ; pos <
BOARD_SIZE ; pos++) {
            if( isLegal(pos,nDie) )
v.add(new int[]{pos});
        }

        int[] ret = new int[v.size()];
        for(int i = 0 ; i < v.size() ;
i++) {
            ret[i] =
((int[])v.elementAt(i))[0];
        }

        return ret;
    }
}
```

Similarity-based Opponent Modelling using Imperfect Domain Theories

Timo Steffens
Institute of Cognitive Science
Kolpingstr. 7
49080 Osnabrück, Germany
tsteffen@uos.de

Abstract- This paper proposes a similarity-based approach for opponent modelling in multi-agent games. The classification accuracy is increased by adding derived attributes from imperfect domain theories to the similarity measure. The main contributions are to show how different forms of domain knowledge can be incorporated into similarity measures for opponent modelling, and to show that the situation space of the opponent modelling approach is not required to be the same as the situation space of the opponent players. Our approach has been implemented and evaluated in the domain of simulated soccer.

1 Introduction

Opponent modelling is an essential part of playing well in a game, as it allows to predict future actions of the opponent and adapt one's own behavior accordingly. Case-based reasoning (CBR) is a common method for opponent modelling in multi-agent games (e. g., [3, 32, 9]): From a CBR perspective, predicting the opponent's action in a given situation S is the classification goal. A CBR system compares S to a case-base of previously observed situations. The situation S' that is most similar to S will be selected, and the action in S' is returned.

The classification- or prediction-accuracy of CBR is largely determined by the quality of the similarity measure. Unfortunately, implementing a similarity measure is not trivial, since similarity is not an absolute quantity: What should be regarded as similar depends on the context. The similarity measure must be adapted to the game situation and role of the agent whose action is to be predicted. Consider situations in a soccer game: The positions of team B's defenders will be rather irrelevant if the classification goal is the action of team A's goalie, but rather relevant if the classification goal is the action of team A's forwards. For these two classification goals, the similarity measure must weight attributes (i. e., player positions) differently. Other CBR approaches in simulated soccer dealt with this problem by introducing a focus: Cases contain only positions of those players that are close to the ball [3]. Another method is to partition the known cases into defensive, transitional, and offensive sets [16]. Yet, such methods are domain specific ad-hoc solutions.

The next section gives an overview of the evaluation-domain, simulated soccer. Section 3 defines which types of knowledge can be used. Then it will be described how similarity measures are extended to incorporate domain knowledge. Section 5 will show how the knowledge-rich similarity measure can be used for attribute- as well as multi-agent matching. Section 6 reports the evaluation results, and the last section concludes and outlines future work.

2 An example Multi-Agent Game: RoboCup

The RoboCup domain is a typical multi-agent game where opponent modelling is crucial for successfully counteracting adversary agents [14, 3]. Two teams of autonomous agents connect to a server and play simulated soccer against each other. Each player is an autonomous process. This is a challenge for opponent-modelling, since the behavior of each opponent player has to be approximated individually.

Decision making is done in real-time, more precisely, in discrete time steps: Every 100ms the agents can execute a primitive action and the world-state changes based on the actions of all players. Basically, the action primitives are *dash, turn, kick*, which must be combined in consecutive time steps in order to form high-level actions like passes or marking. The agents act on incomplete and uncertain information: Their visual input consists of noisy information about objects in their limited field of vision. There is an additional privileged agent, the online coach, which receives noise-free and complete visual input of the playing field. The online coach is almost exclusively used for opponent modelling purposes. Every 100 ms it receives information about the position and velocity of all objects on the playing field (22 players and the ball). The agents' actions cannot be observed directly, but can be inferred from the differences between consecutive world-states. E. g., in our implementation the coach assumes that the player controlling the ball executed a kick, if the ball's velocity increases.

Cases for our CBR system are generated from the observations of the coach. A case is represented in two parts: 46 attributes (23 positions and 23 velocities) specifying the situation, and 22 attributes storing the actions. In a prediction task, only the situation is known and one of the actions serves as the classification goal; the other actions are ignored.

RoboCup is an ideal domain for evaluating our approach, because the same case-base can be used for different classification goals: The action of each player is handled as a single prediction task with its own classification goal. Since the coach receives information about all situation attributes, cases can be stored without further analysis. Generalization wrt. the specific classification goal is deferred until classification time, a property of lazy learning [2].

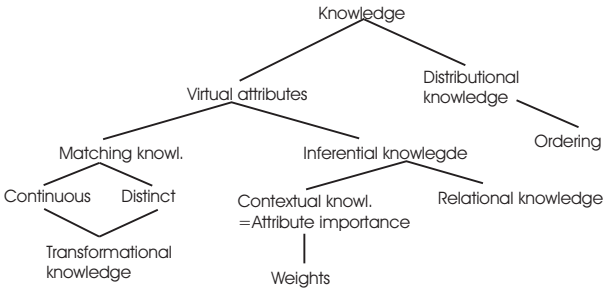


Figure 1: Hierarchy of knowledge types.

3 Types of domain knowledge

This section discusses which types of knowledge are useful for similarity-based opponent modelling. Previous similarity-based approaches used domain knowledge implicitly or in an ad-hoc kind of way (e. g., [9, 16, 3]). Other works incorporated isolated knowledge types. A systematic analysis of which types of knowledge are useful will also provide insights into which information should be learned from the instances if the knowledge is not explicitly given. For each knowledge type we will refer to CBR systems that employed such knowledge.

For the examples, we use the following notation: $C_1, C_2, C_3 \in \mathbb{R}$ are continuous attributes. $D_1, D_2 \in \mathbb{Z}$ are discrete attributes. $P(x)$ is a binary concept applicable to instance x. $C_i(x)$ or $D_i(x)$ denote the value of instance x for attribute C_i or D_i. $w \in \mathbb{R}$ is a weight.

We categorize the relevant types of knowledge into a hierarchy (see figure 1). At the most general level, we distinguish *virtual attributes* [19] (or *derived attributes*) from *distributional knowledge*. The latter includes knowledge about the range and distribution of attributes and their values. As is commonly used, knowledge about the range of an attribute can be used to normalize the attribute similarity to (0..1). Since this type of knowledge is commonly used in CBR, we focus on the less researched type of knowledge that can be formalized as virtual attributes.

Virtual attributes are attributes that are not directly represented in the cases but can be inferred from other attributes [19]. They are already quite common in database research. In CBR, virtual attributes are useful if the classification goal does not depend on the represented attributes themselves, but on relations between them. For example, if C_1 is the position of player A and C_2 is the position of player B, then a virtual attribute $C_3(x) = C_1(x) - C_2(x)$ could be the distance between A and B.

We further distinguish between *matching knowledge* and *inferential knowledge*. Discrete matching knowledge states that two values of an attribute are equivalent. The famous PROTOS system made extensive use of this type of knowledge [18]. Also taxonomies are instantiations of matching knowledge. They were also used in CBR [4]. Continuous matching knowledge defines regions in the instance space. Examples:

- $C_1(x) > 30 \ \wedge \ C_1(x) < 50$ (continuous)
- $D_1(x) \equiv D_1(y)$ (discrete)

Matching knowledge can be used to match syntactically different attributes that are semantically equivalent. For example, in our opponent modelling approach two different players will be treated as equivalent if they have the same role (such as defender).

Transformational knowledge is a special form of matching knowledge where usually some arithmetic or operations are involved in order to map a point in the instance-space to another point. For example, transformational knowledge has been used to establish identity despite geometric rotation (e. g., [23]). Example: $C_1(x) = rotate(C_1(y), 30)$ In our RoboCup implementation, transformational knowledge is used to match local scenes from one wing to the other.

Inferential knowledge specifies the value of an attribute that is inferrable from some other attributes' values. This type of knowledge has been used in explanation-based CBR (e. g., [1]). Example: $P(x) \leftarrow C_1(x) > 30 \wedge C_1(x) < 50$ Note that the condition part makes use of matching knowledge. A typical example from simulated soccer is to define offside as a virtual attribute over the directly represented player and ball positions.

Contextual knowledge is a special form of inferential knowledge. It states that some feature is important given some other features. For an overview over contextual features, refer to [31]. Example: $important(P(x)) \leftarrow C_1(x) > 30 \wedge C_1(x) < 50$ We use contextual knowledge to code that a team's defenders' positions are irrelevant if the team's forward handles the ball close to the opponent goal.

In our hierarchy, *weights* are a special form of contextual knowledge. They allow to express the importance of a feature on a continuous scale. Thus, we can express feature weights in a global way $important(P(x), w) \leftarrow TRUE$, or in a local way $important(P(x), w) \leftarrow C_1(x) > 30 \wedge C_1(x) < 50$.

In other words, contextual knowledge and weights can be called "attribute importance" knowledge.

Relations are special forms of inferential knowledge. The condition part uses at least two different attributes. Relational knowledge for similarity is prominent in computational modelling of human categorization [17]. Example: $P(x) \leftarrow C_1(x) > C_2(x)$. Note that relations usually make use of matching knowledge in the condition part, as they define regions in which the relation holds. Relations are for example necessary to code wether a certain opponent is between the ball and the goal.

Ordering of nominal feature values is a special form of distribution knowledge. It establishes a dimension of the instance space. In [30] it was shown that using knowledge of the ordering of discrete feature values can increase classification accuracy.

In previous work [26] we used goal-dependency networks (GDNs) as proposed in [28]. In this framework, GDNs are a combination of relational knowledge about the subgoal-relation and contextual knowledge (a property is important if a certain subgoal is active).

4 The similarity measure

The non-extended similarity measure is defined in the following way:

$$sim(S_1, S_2) = \sum_{i=1}^{22} [\omega_i * \Delta(p(i, S_1), p(i, S_2)) + \omega'_i * \Delta(v(i, S_1), v(i, S_2))] + \omega_0 * \Delta(bp(S_1), bp(S_2)) + \omega'_0 * \Delta(bv(S_1), bv(S_2)) \quad (1)$$

where S_1 and S_2 are the two situations in comparison, $p(i, S_j)$ and $v(i, S_j)$ are the position and velocity of player i in situation S_j, respectively, $bp(S_j)$ and $bv(S_j)$ are the ball-position and ball-velocity in S_j, respectively. $\Delta(A, B)$ is the Euclidean distance between A and B, and ω_k and ω'_k with $\sum_{k=0}^{22}(\omega_k + \omega'_k) = 1$ are weights for positions and velocities, respectively. Semantically, weights denote the relevance of attributes.

Note that the non-extended similarity measure uses some domain knowledge, that is, it makes use of distributional knowledge, as it normalizes the ball and player positions and velocities.

In comparison, the extended similarity measure is defined as follows: In line with the well-known local-global principle [5], we compute the similarity between two situations as the weighted average aggregation of the attributes' local similarities:

$$sim(S_1, S_2) = \sum_{i=1}^{n}(\omega_i * s_i)$$

where s_i are the local similarity values (i. e., s_i is the similarity for attribute i), and the ω_i are the corresponding weights.

$$\begin{aligned} sim(S_1, S_2) = {} & \omega_1 * 1(role(S_1), role(S_2)) + \\ & \omega_2 * 1(region(S_1), region(S_2)) + \\ & \omega_3 * 1(offside(S_1), offside(S_2)) + \\ & \omega_4 * 1(pressing(S_1), pressing(S_2)) + \\ & \omega_5 * \sum_{i=1}^{22} 1(free(S_1, i), free(S_2, i)) + \\ & \omega_6 * 1(ahead(S_1), ahead(S_2)) + \\ & \omega_7 * \Delta(positions(S_1), positions(S_2)) + \\ & \omega_8 * \Delta(velocities(S_1), velocities(S_2)) \end{aligned} \quad (2)$$

where $1(X, Y) = 1$ iff $X = Y$, and 0 otherwise.

The attributes $role$ and $region$ make use of matching knowledge as they define hyper-planes in the instance-space. $role(S) \in \{forward, defender, midfielder\}$ denotes the role of the ball owner. $region(S) \in \{inFrontOfGoal, penaltyArea, corner, wing, midfield\}$ denotes the region the ball is in. Note that no distinction is made between left and right wing, and between the four corners, which is achieved by transformational knowledge about mirroring and rotating.

The attributes $offside$, $pressing$, and $free$ make use of inferential knowledge. $offside(S)$ is a binary predicate that checks whether the situation is offside. $pressing(S)$ checks whether pressing is performed in the situation, that is, whether the opponent attacks the ball owner with two or more players. $free(S, i)$ checks whether player i stands free (i. e., no player of the opponent is within a certain distance).

The predicate $ahead(S)$ makes use of relational knowledge. It denotes the number of opponents that are between the ball and the goal.

$positions(S)$ and $velocities(S)$ are examples for contextual knowledge. They denote the positions and velocities of the players that are relevant in the given situation. Relevance of a player is computed by its role and the role of the ball owner. If the ball owner is a forward, its own defenders and the opponent's forwards are deemed irrelevant. If the ball owner is a defender, its own forwards and the opponent's defenders are deemed irrelevant.

We applied the RELIEF method [13] (with extensions for kNN with $k > 1$ and for non-binary target classes proposed in [15]) for learning the attribute weights.

5 Multi-agent matching

In most CBR applications, matching attributes is straightforward, as equally named attributes or attributes at the same position of a vector are matched. However, when applying CBR to opponent modelling in multi-agent games, matching of attributes is not trivial. For example, the positions and velocities stored in the cases are linked to specific players. Since it is rather common to swap positions in RoboCup (e. g., to move a tired player from an exhausting to a slower position), comparing positions of situation S_1 to situation S_2 must take into account that it is not necessarily the case that the position of player number 3 in situation S_1 must be compared to the position of player number 3 in situation S_2. Instead, it might be the case that in S_2 players 3 and 9 swapped positions. In that case, the desired concept of similarity can only be achieved by comparing the positions of players 3 and 9.

Hence, before two situations can be compared the agents of the two situations have to be matched. Traditional multi-agent matching usually requires a 1 to 1 matching [29]. However, when doing multi-agent matching for attribute matching, this requirement must be lifted. Consider the example soccer situation in figure 2 (top). The players A,B,C belong to one team and x,y,z to the other team. The situation on the right differs from the one on the left only in that player x has been moved. An optimal 1 to 1 matching (minimizing the summed distances) would match each player from the left situation to itself in the right situation. However, the relevance of player x is different in both situations. On the left, player x marks player C so that A cannot safely pass the ball to C. On the right, player x is basically equivalent to player z and does not mark player C. Hence, we propose to match both players x and z on the right to player z on the left, and to leave player x on the left unmatched, as no player on the right corresponds to its

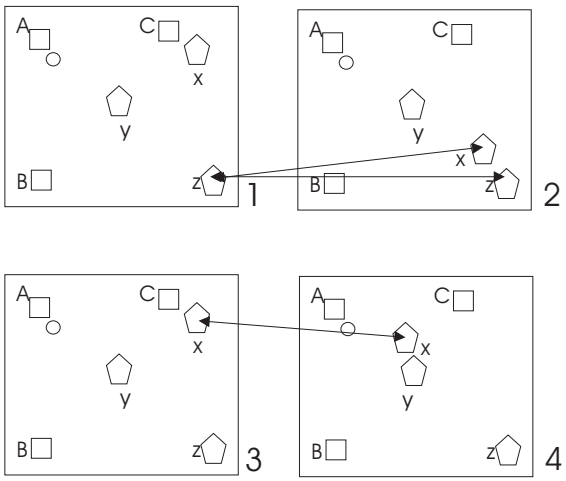


Figure 2: Player matching in soccer situations. Players A,B,C belong to one team, player x,y,z to the other. Unless otherwise depicted by arrows, players from the left are matched to themselves on the right. Top: Situation 1 and 2 are compared. Bottom: Situation 3 and 4 are compared.

situation-specific role of marking.

Therefore, in our implementation multi-agent matching is done by matching those players of the same team that are most similar with respect to the similarity-measure. Thus, the knowledge-rich virtual attributes are also used for multi-agent matching. This is different than previous work in multi-agent matching in RoboCup where players were matched based on their spatial distance only [29].

To illustrate why virtual attributes are also useful for multi-agent matching, consider the example soccer situation in figure 2 (bottom). The left bottom situation is the same one as the left top one. It differs from the right bottom one only in the position of player x. A matching algorithm that computes player similarity only based on spatial distance would assign player x and y from situation 4 to player y in situation 3. However, in situation 4 player x is not equivalent to player y since it may intercept a pass from player A to player C, just as player x in situation 3. Thus, a virtual attribute $betweenBallAndPlayer(X, .)$ is useful which is true for player x in both situations. If it is weighted great enough so that it outweighs the spatial distances, player x from situation 1 will be matched to player x in situation 2, which is consistent with the player's situation-specific roles.

Additionally, in our matching algorithm, the ball-owners of two situations are always matched, and of course players are only matched to players of their own team.

Since some contextual attributes in the similarity measure specify that some players are irrelevant (for example, team A's defenders are deemed irrelevant if team A's forward has the ball) in certain situations, the matching algorithm does not match these players in the corresponding situations.

Table 1: Mean prediction accuracies of the extended and the non-extended similarity measures. All attributes were weighted equally. p is the significance level of a paired, two-tailed t-test. N is the number of logfiles.

Non-extended	Extended	p	N
86.3	86.4	0.838	20

6 Experiments

The following experiments tested whether the prediction accuracy for player actions increases if the similarity measure is extended with imperfect domain knowledge. Both the unextended and the extended similarity measures are tested on the same case-base and test cases. The test domain is RoboCup. We used 20 publicly available logfiles of recorded games between 19 different teams. Logfiles contain the same data that the online coach (which was introduced in section 2) would receive. For each game, the first 2500 cycles of the match were recorded into the case-base. A complete game lasts 6000 time steps. The test cases were drawn from the remaining time steps at fixed intervals of 50 time steps. The classification goal was the action of the ball owner.

In the first experiment, all attributes (including the additional ones) were weighted equally. In the second experiment, the attribute weights were learned with RELIEF. For each game the weights were relearned using the case-base as training data.

6.1 Focus on high-level actions

In a complex domain such as RoboCup it is infeasible to predict an agent's behavior in terms of primitive actions. For individual skills (e. g., dribbling), primitive actions are often combined by neural networks. These are trained using amounts of training instances that are typically one or two levels of magnitude greater than the amount of observations available for opponent modelling. Hence, it is infeasible to predict an agent's primitive actions. Rather, in our experiments we predicted the high-level action *shoot on goal*. We assume that for taking countermeasures it is sufficient to anticipate high-level actions within a certain time window. For example, if a defender knows that within the next 20 time steps an opponent will shoot on the goal, it can position itself accordingly (and maybe even inhibit the shot by doing so. Therefore, in our prediction experiments the agents do not use the predictive information in order not to interfere with the prediction accuracy.) For both the static and the adaptable similarity measures, the prediction of an action is counted as correct if the action occurs within 20 time steps after the prediction.

6.2 Results

The mean prediction accuracy of both similarity measures of the first experiment are shown in table 1.

Unfortunately, adding virtual attributes alone does not

Table 2: Mean prediction accuracies of the extended and the non-extended similarity measures. All attribute weights were learned with RELIEF. p is the significance level of a paired, two-tailed t-test. N is the number of logfiles.

Non-extended	Extended	p	N
86.4 %	88.2 %	0.009	20

improve prediction accuracy. Still, the accuracy is not decreased either, although it is known that irrelevant attributes negatively impact accuracy [10]. This is consistent with results in other domains which showed that virtual attributes are only beneficial if their weights are learned [27]. The second experiment weighs attributes based on their relevance as learned by RELIEF. The results are shown in table 2.

The extended similarity measure predicts significantly better than the non-extended similarity measure if the attribute weights are learned. In a two-tailed t-test, the significance level is 0.009. However, the accuracy difference between the extended and non-extended measure is small. Our analysis suggest that the small impact of the similarity measure is due to the fact that any increase of prediction accuracy is difficult, since the behaviors of the player agents are implemented by many different methods. Player implementations range from simple decision trees [6], over probabilistic approaches [8] to neural networks [20]. Particularly if behaviors are learned, the partitioning of the situation space can be highly irregular and complex. Furthermore, it is very unlikely that the opponent players used the same domain knowledge. Hence, their situation space will be different from the situation space of our case-base. Considering this, we believe that an accuracy increase of 1.8 percent points is substantial.

7 Related work

Apart from the similarity-based approach, there are several different approaches to opponent modelling. However, not all of them are well-suited for a continuous, real-time multi-agent game such as RoboCup.

In game theory there are approaches to learn opponent models from action sequences [7]. Usually a payoff-matrix is necessary, but for predicting the opponent's actions this requirement does not hold [22]. Unfortunately, these learning techniques assume that the opponent strategy can be described by a deterministic finite automaton, which might not always be the case in a complex domain. Most importantly, game theory can describe game states only as history of actions, which is infeasible in complex games such as RoboCup, where subsequent game states are not only determined by player actions but also by the game physics.

Predicting opponent actions can also be done via plan-recognition [12, 11]. Predefined plan libraries are needed, however, and reactive agents cannot be modelled. Similarly, classifying opponents into classes and selecting appropriate counter-strategies [25, 21] requires pre-defined model libraries. In contrast, CBR only requires a set of observations, tuning of the similarity measure is optional. However, CBR as used in this paper does not provide an appropriate counter-action. Up to now it only predicts the opponent's actions.

An approach that avoids predicting the opponent's actions is for example to adapt probability weights of action rules by reinforcement [24]. Instead of choosing actions that counter-act the opponent's next move, the own behavior is adapted based on rewards and penalties.

8 Conclusions

We presented an approach that enriches similarity based opponent modelling in multi-agent games with imperfect domain knowledge. A taxonomy for different types of domain knowledge was proposed and it was shown how each type can be incorporated into similarity measures. The prediction accuracy of the knowledge-rich measure was compared to a knowledge-poor measure in the domain of simulated soccer. The results suggest that similarity-based opponent modelling can benefit from domain knowledge even if it is not known whether the opponent uses the same domain knowledge.

Bibliography

[1] Agnar Aamodt. Explanation-driven case-based reasoning. In Stefan Wess, Klaus-Dieter Althoff, and Michael M. Richter, editors, *Topics in Case-Based Reasoning*, pages 274–288. Springer, 1994.

[2] David Aha. Editorial for the special issue: lazy learning. *Artificial Intelligence Review*, 11:7–10, 1997.

[3] Mazda Ahmadi, Abolfazl Keighobadi-Lamjiri, Mayssam M. Nevisi, Jafar Habibi, and Kamiz Badie. Using a two-layered case-based reasoning for prediction in soccer coach. In Hamid R. Arabnia and Elena B. Kozerenko, editors, *Proceedings of the International Conference of Machine Learning; Models, Technologies and Applications (MLMTA'03)*, pages 181–185. CSREA Press, 2003.

[4] Ralph Bergmann. On the use of taxonomies for representing case features and local similarity measures. In Lothar Gierl and Mario Lenz, editors, *Proceedings of the Sixth German Workshop on CBR*, pages 23–32, 1998.

[5] Ralph Bergmann. *Experience Management*. Springer, Berlin, 2002.

[6] Sean Buttinger, Marco Diedrich, Leonhard Hennig, Angelika Hoenemann, Philipp Huegelmeyer, Andreas Nie, Andres Pegam, Collin Rogowski, Claus Rollinger, Timo Steffens, and Wilfried Teiken. Orca project report. Technical report, University of Osnabrueck, 2001.

[7] David Carmel and Shaul Markovich. Learning models of intelligent agents. In Howard Shrobe and Ted Senator, editors, *Proceedings of the Thirteenth National Conference on Artificial Intelligence and the Eighth Innovative Applications of Artificial Intelligence Conference, Vol. 2*, pages 62–67, Menlo Park, California, 1996. AAAI Press.

[8] Remco de Boer, Jelle Kok, and Frans C. A. Groen. Uva trilearn 2001 team description. In Andreas Birk, Silvia Coradeschi, and Satoshi Tadokoro, editors, *RoboCup 2001: Robot Soccer World Cup V. Lecture Notes in Computer Science 2377*, pages 551–554, Berlin, 2002. Springer.

[9] Joerg Denzinger and Jasmine Hamdan. Improving modeling of other agents using stereotypes and compactification of observations. In *Proceedings of AAMAS 2004*. ACM, 2004.

[10] Anthony D. Griffiths and Derek G. Bridge. A yardstick for the evaluation of case-based classifiers. In Ian D. Watson, editor, *Proceedings of Second UK Workshop on Case-Based Reasoning*, 1996.

[11] Gal A. Kaminka and Dorit Avrahami. Symbolic behavior-recognition. In Mathias Bauer, Piotr Gmytrasiewicz, Gal A. Kaminka, and David V. Pynadath, editors, *Workshop on Modeling Other Agents from Observations at AAMAS 2004*, pages 73–80, 2004.

[12] Henry Kautz. A formal theory of plan recognition and its implementation. In J. Allen, H. Kautz, R. Pelavin, and J. Tenenberg, editors, *Reasoning about Plans*, pages 69–125. Morgan Kaufman, San Mateo, CA, 1991.

[13] Kenji Kira and Larry A. Rendell. A practical approach to feature selection. In Derek H. Sleeman and Peter Edwards, editors, *Proceedings of the Ninth International Workshop on Machine Learning*, pages 249–256. Morgan Kaufmann Publishers Inc., 1992.

[14] Hiroaki Kitano, Milind Tambe, Peter Stone, Manuela Veloso, Silvia Coradeschi, Eiichi Osawa, Hitoshi Matsubara, Itsuki Noda, and Minoru Asada. The robocup synthetic agent challenge,97. In *International Joint Conference on Artificial Intelligence (IJCAI97)*, pages 24–29, San Francisco, CA, 1997. Morgan Kaufmann.

[15] Igor Kononenko. Estimating attributes: Analysis and extensions of RELIEF. In F. Bergadano and L. de Raedt, editors, *Proceedings of the European Conference on Machine Learning*, pages 171–182, Berlin, 1994. Springer.

[16] Cynthia Marling, Mark Tomko, Matthew Gillen, David Alexander, and David Chelberg. Case-based reasoning for planning and world modeling in the robocup small sized league. In Ubbo Visser, editor, *IJCAI Workshop on Issues in Designing Physical Agents for Dynamic Real-Time Environments*, 2003.

[17] Douglas L. Medin, Robert L. Goldstone, and Dedre Gentner. Respects for similarity. *Psychological Review*, 100(2):254–278, 1993.

[18] Bruce W. Porter, Ray Bareiss, and Robert C. Holte. Concept learning and heuristic classification in weak-theory domains. *Artificial Intelligence*, 45(1-2):229–263, 1990.

[19] Michael M. Richter. Fallbasiertes Schliessen. *Informatik Spektrum*, 3(26):180–190, 2003.

[20] Martin Riedmiller, Arthur Merke, W. Nowak, M. Nickschas, and Daniel Withopf. Brainstormers 2003 - team description. In Daniel Polani, Andrea Bonarini, Brett Browning, and Kazuo Yoshida, editors, *Pre-Proceedings of RoboCup 2003*, 2003.

[21] Patrick Riley and Manuela Veloso. On behavior classification in adversarial environments. In Lynne E. Parker, George Bekey, and Jacob Barhen, editors, *Distributed Autonomous Robotic Systems 4*, pages 371–380. Springer-Verlag, 2000.

[22] Collin Rogowski. Model-based opponent-modelling in domains beyond the prisoner's dilemma. In Mathias Bauer, Piotr Gmytrasiewicz, Gal A. Kaminka, and David V. Pynadath, editors, *Workshop on Modeling Other Agents from Observations at AAMAS 2004*, pages 41–48, 2004.

[23] Joerg W. Schaaf. Detecting gestalts in cad-plans to be used as indices. In Angi Voss, editor, *FABEL - Similarity concepts and retrieval methods*. GMD, Sankt Augustin, 1994.

[24] Pieter Spronck, Ida Sprinkhuizen-Kuyper, and Eric Postma. Online adaptation of game opponent ai in simulation and in practice. In Quasim Mehdi and Norman Gough, editors, *Proceedings of the 4th International Conference on Intelligent Games and Simulation (GAME-ON 2003)*, pages 93–100, Belgium, 2003. EUROSIS.

[25] Timo Steffens. Feature-based declarative opponent-modelling in multi-agent systems. Master's thesis, Institute of Cognitive Science Osnabrueck, 2002.

[26] Timo Steffens. Adapting similarity-measures to agent-types in opponent-modelling. In Mathias Bauer, Piotr Gmytrasiewicz, Gal A. Kaminka, and David V. Pynadath, editors, *Workshop on Modeling Other Agents from Observations at AAMAS 2004*, pages 125–128, 2004.

[27] Timo Steffens. Virtual attributes from imperfect domain theories. In Brian Lees, editor, *Proceedings of the 9th UK Workshop on Case-Based Reasoning at AI-2004*, pages 21–29, 2004.

[28] Robert E. Stepp and Ryszard S. Michalski. Conceptual clustering: Inventing goal-oriented classifications of structured objects. In Ryszard S. Michalski, Jaime G.

Carbonell, and Tom M. Mitchell, editors, *Machine Learning: An Artificial Intelligence Approach*, volume II. Morgan Kaufman Publishers, Inc., Los Altos, CA, 1986.

[29] Frieder Stolzenburg, Jan Murray, and Karsten Sturm. Multiagent matching algorithms with and without coach. In Michael Schillo, Matthias Klusch, Jrg Mller, and Huaglory Tianfield, editors, *Proceedings of the 1st German Conference on Multiagent System Technologies*, pages 192–204, Berlin, 2003. Springer.

[30] Jerzy Surma. Enhancing similarity measure with domain specific knowledge. In *Proceedings of the Second European Conference on Case-Based Reasoning*, pages 365–371, Paris, 1994. AcknoSoft Press.

[31] Peter Turney. The management of context-sensitive features: A review of strategies. In *Proceedings of the Workshop on Leaning in Context-sensitive Domains at the 13th International Conference on Machine Learning*, pages 60–65, 1996.

[32] Jan Wendler. Recognizing and predicting agent behavior with case based reasoning. In Daniel Polani, Andrea Bonarini, Brett Browning, and Kazuo Yoshida, editors, *RoboCup 2003: Robot Soccer World Cup VII*, *Lecture Notes in Artificial Intelligence*, Berlin, Heidelberg, New York, 2004. Springer.

A Survey on Multiagent Reinforcement Learning Towards Multi-Robot Systems

Erfu Yang
University of Essex
Wivenhoe Park, Colchester
CO4 3SQ, Essex, United Kingdom
eyang@essex.ac.uk

Dongbing Gu
University of Essex
Wivenhoe Park, Colchester
CO4 3SQ, Essex, United Kingdom
dgu@essex.ac.uk

Abstract- Multiagent reinforcement learning for multi-robot systems is a challenging issue in both robotics and artificial intelligence. With the ever increasing interests in theoretical research and practical applications, currently there have been a lot of efforts towards providing some solutions to this challenge. However, there are still many difficulties in scaling up multiagent reinforcement learning to multi-robot systems. The main objective of this paper is to provide a survey on multiagent reinforcement learning in multi-robot systems, based on the literature the authors collected. After reviewing some important advances in this field, some challenging problems are analyzed. A concluding remark is made from the perspectives of the authors.

1 Introduction

Multi-Robot Systems (MRSs) can often be used to fulfil tasks that are difficult to be accomplished by an individual robot, especially in the presence of uncertainties, incomplete information, distributed control, and asynchronous computation, etc. During the last decade MRSs have received considerable attention [1–16].

When designing MRSs, it is impossible to predict all the potential situations robots may encounter and specify all robot behaviours optimally in advance. Robots in MRSs have to learn from, and adapt to their operating environment and their counterparts. Thus, learning becomes one of the important and challenging problems in MRSs.

Over the last decade there has been increasing interest in extending individual reinforcement learning (RL) to multiagent systems, particularly MRSs [16–35]. From a theoretic viewpoint, this is a very attractive research field since it will expand the range of RL from the realm of simple single-agent to the realm of complex multiagents where there are agents learning simultaneously.

There have been some advances in RL for both multiagent systems and MRSs. The objective of this paper is to review these existing works and analyze some challenging issues from the viewpoint of multiagent RL in MRSs.

2 Preliminaries

2.1 Markov Decision Process

Markov Decision Processes (MDPs) are the mathematical foundation for single agent RL and defined as [26, 36]:

Definition 1 (MDP) *A Markov Decision Process is a tuple* $< \boldsymbol{\mathcal{S}}, \boldsymbol{\mathcal{A}}, \boldsymbol{\mathcal{T}}, \boldsymbol{\mathcal{R}}, \gamma >$*, where* $\boldsymbol{\mathcal{S}}$ *is a finite discrete set of environment states,* $\boldsymbol{\mathcal{A}}$ *is a finite discrete set of actions available to the agent,* γ *(*$0 \leq \gamma < 1$*) is a discount factor,* $\boldsymbol{\mathcal{T}} : \boldsymbol{\mathcal{S}} \times \boldsymbol{\mathcal{A}} \rightarrow \Pi(\boldsymbol{\mathcal{S}})$ *is a transition function giving for each state and action, a probability distribution over states,* $\boldsymbol{\mathcal{R}} : \boldsymbol{\mathcal{S}} \times \boldsymbol{\mathcal{A}} \rightarrow \mathbb{R}$ *is a reward function of the agent, giving the expected immediate reward received by the agent under each actions in each state.*

Each MDP has a deterministic stationary optimal policy [26]. In an MDP, the agent acts in a way such as to maximize the long-run value it can expect to gain. Under the discounted objective the factor γ controls how much effect future rewards have on the decisions at each moment. It is noted that the reward can be probabilistic in some cases.

2.2 Reinforcement Learning

The objective of RL is to learn how to act in a dynamic environment from experience by maximizing some payoff functions. In RL, the state dynamics and reinforcement function are at least partially unknown. Thus the learning occurs iteratively and is performed only through trial-and-error methods and reinforcement signals, based on the experience of interactions between the agent and its environment.

2.3 Q-Learning

Q-learning [37] is a value learning version of RL that learns utility values (Q values) of state and action pairs. It is a form of model-free RL and provides a simple way for agents to learn how to act optimally in controlled Markovian domains. It also can be viewed as a method of asynchronous dynamic programming. In essence Q-learning is a temporal-difference learning method. The objective of Q-learning is to estimate Q values for an optimal policy. During the learning an agent uses its experience to improve its estimate by blending new information into its prior experience. Although there may be more than one optimal policy, the Q^* values are unique [37]. The individual Q-learning in discrete cases has been proved to converge to optimal values with probability one if state action pairs are visited infinite times and learning rate declines [37].

2.4 Matrix Games

Matrix games are the most elementary type of many player, particularly two-player games [38]. In matrix games players select actions from their available action space and receive rewards that depend on all the other player's actions.

Definition 2 (Matrix Games) *A matrix game is given by a tuple* $< n, \boldsymbol{\mathcal{A}}_1, \cdots, \boldsymbol{\mathcal{A}}_n, R_1, \cdots, R_n >$*, where* n *is the number of players,* $\boldsymbol{\mathcal{A}}_i$ *and* R_i $(i = 1, \cdots, n)$ *are the finite action set and payoff function respectively for player* i*.*

2.5 Stochastic Games

Currently multiagent learning has focused on the theoretic framework of Stochastic Games (SGs) or Markov Games (MGs). SGs extend one-state MG to multi-state cases by modeling state transitions with MDP. Each state in a SG can be viewed as a MG and a SG with one player can be viewed as an MDP. A great deal of research on multiagent learning has borrowed the theoretic frameworks and notions from SGs [17–21, 23, 25–29, 31–34, 39–42]. SGs have been well studied in the field of RL and appear to be a natural and powerful extension of MDPs to multi-agent domains.

3 Theoretic Frameworks for Multiagent RL

3.1 SG-Based Frameworks

A RL framework of SGs is given as follows [17, 26, 34, 43]:
Definition 3 (Framework of SGs) *A learning framework of SGs is described by a tuple* $< \boldsymbol{\mathcal{S}}, \boldsymbol{\mathcal{A}}_1, \cdots, \boldsymbol{\mathcal{A}}_n, T, R_1, \cdots, R_n, \gamma >$*, where*

- $\boldsymbol{\mathcal{S}}$ *is a finite state space;*
- $\boldsymbol{\mathcal{A}}_1, \cdots, \boldsymbol{\mathcal{A}}_n$ *are the corresponding finite sets of actions available to each agent.*
- $T : \boldsymbol{\mathcal{S}} \times \boldsymbol{\mathcal{A}}_1 \times \cdots \times \boldsymbol{\mathcal{A}}_n \rightarrow \Pi(\boldsymbol{\mathcal{S}})$ *is a state transition function, given each state and one action from each agent. Here* $\Pi(\boldsymbol{\mathcal{S}})$ *is a probability distribution over the state space* $\boldsymbol{\mathcal{S}}$*.*
- $R_i : \boldsymbol{\mathcal{S}} \times \boldsymbol{\mathcal{A}}_1 \times \cdots \times \boldsymbol{\mathcal{A}}_n \rightarrow \mathbb{R} (i = 1, \cdots, n)$ *represents a reward function for each agent.*
- $0 \leq \gamma < 1$ *is the discount factor.*

In such a learning framework of SGs, learning agents attempt to maximize their expected sum of discounted rewards. Correspondingly a set of Q-functions for agent i $(i = 1, \cdots, n)$ can be defined according to their stationary policies $\pi_1, \cdots, \pi_n$. Unlike a single-agent system, in multi-agent systems the joint actions determine the next state and rewards to each agent. After selecting actions, the agents are transitioned to the next state and receive their rewards.

3.2 Fictitious Play Framework

In a known SG, the framework of fictitious play can be used as a technique to find equilibria. For a learning paradigm, fictitious play can also be applied to form a theoretical framework [44]. It provides a quite simple learning model. In the framework of fictitious play, the algorithm maintains information about the average estimated sum of future discounted rewards. According to the Q-functions of agents, the fictitious play method deterministically chooses the actions for each agent that would have done the best in the past. For computing the estimated sum of future discounted rewards, a simple temporal difference backup may be used.

Compared with the framework of SGs, the main merit of fictitious play is that it is capable of finding equilibria in both zero-sum games and some classes of general-sum games [44]. One obvious disadvantage of this framework is that fictitious play merely adopts deterministic policies and cannot play stochastic strategies. Hence it is hard to apply in zero-sum games because it can only find an equilibrium policy but does not actually play according to that policy [44]. In addition learning stability is another serious problem. Since the fictitious play framework is of inherent discontinuity, a small change in data could lead to an abrupt change in behaviour [33]. To overcome this unstable problem, many variants of fictitious play have been developed, see [33] as well as the literature therein.

3.3 Bayesian Framework

The multiagent RL algorithms developed from the SG framework, such as Minimax-Q, Nash-Q, etc., always require to converge to desirable equilibria. Thus, sufficient exploration of strategy space is needed before convergence can be established. Solutions to multiagent RL problems are usually based on equilibrium. In order to obtain an optimal policy, agents have to find and even identify the equilibria before the policy is used at the current state.

A Bayesian framework for exploration in multiagent RL systems was proposed in [31, 33]. It is a model-based RL model. In this framework a learning agent can use priors to reason about how its action will influence the behaviours of other agents. Thus, some prior density over possible dynamics and reward distribution have to be known by a learning agent in advance.

A basic assumption in the Bayesian framework is that the learning agent is able to observe the actions taken by all agents, the resulting game state, and rewards received by other agents. Of course, this assumption will have no problem for the coordination of multiagents, but it will restrict its applications in other settings where opponent agents generally will not broadcast their information to the others.

To establish the belief, a learning agent under the Bayesian framework has some priors, such as probability distribution over the state space as well as the possible strategy space. The belief is then updated during the learning by observing the results of its actions and action choices of other agents. In order to accurately predict the actions of other agents, the learning agent has to record and maintain appropriate observable history. In [31, 33] it is assumed that the learning agent can keep track of sufficient history to make such predictions. Besides the aforementioned assumptions, in [31, 33] there are two extra assumptions on the belief. First, the priors over models can be factored into independent local models for both rewards and transitions. Second, it needs to be assumed that the belief about opponent strategies also can be factored and represented in some convenient form [31, 33].

3.4 Policy Iteration Framework

Unlike the value iteration frameworks, the policy iteration framework can provide a direct way to find the optimal strategy in the policy space. Under the policy iteration framework Bowling and Veloso [45] proposed a WoLF-PHC al-

gorithm if the other agents are assumed to be playing stationary policies. Other works following the thinking lines of [45] can be found in [46,47]. It seems to have no other reported works on the algorithms under this framework when the other agents in the system are considered to learn simultaneously. Compared with the aforementioned frameworks, many researches for policy iteration RL in multiagent systems still need to be done in the future. Fortunately, there already have been many research results on policy iteration algorithms in single-agent RL systems, for instance, one may refer to [48] as well as the literature therein. Thus one possible way is to extend the existing policy iteration algorithms in single-agent systems to the field of multiagent systems.

4 Multiagent RL Algorithms

The difference between single-agent and multiagent system can be seen as a difference in the properties of the agent's environment. In multiagent systems other adapting agents make the environment no longer stationary, violating the Markov property that traditional single agent behavior learning relies upon.

For individual robot learning, the traditional Q-learning has been successfully applied to many paradigms. Some researchers also apply Q-learning in a straightforward fashion to each agent in a multiagent system. However, the aforementioned fact that the environment is no longer stationary in multiagent system is usually neglected. Over the last decade many researchers have made efforts to use RL methodology, particularly the Q-learning framework as an alternative approach to the learning of MRSs. As pointed out early, the basic assumption for traditional Q-learning working is violated in the case of MRSs.

4.1 Minimax-Q Learning Algorithm

Under SG framework, Littman [17] proposed a *Minimax-Q* learning algorithm for zero-sum games in which the learning player maximizes its payoffs in the worst situation. The players' interests in the game are opposite. Essentially the Minimax-Q learning is a value-function reinforcement learning algorithm. In the Minimax-Q learning the player always try to maximize its expected value in the face of the worst-possible action choice of the opponent. Hence the player would become more cautious after learning. To calculate the probability distribution or the optimal policy of the player, Littman [17] simply used linear programming.

The Minimax-Q learning algorithm was firstly given in [17], which just included empirical results on a simple zero-sum SG game version of soccer. A complete convergence proof was provided in the works thereafter [18, 23, 26], which can be summarized in the following theorem:

Theorem 1 *In a two-player zero-sum multiagent SG environment, an agent following the Minimax-Q learning algorithm will converge to the optimal Q-function with probability one. Furthermore, an agent using a GLIE (Greedy in the Limit with infinite exploration) policy will converge in behaviour with probability one if the limit equilibrium is unique.*

The Minimax-Q learning algorithm may provide a safe policy in that it can be performed regardless of the existence of its opponent [26]. The policy used in the Minimax-Q learning algorithm can guarantee that it receives the largest value possible in the absence of knowledge of the opponent's policy. Although the Minimax-Q learning algorithm manifests many advantages in the domain of two-player zero-sum multiagent SG environment, an explicit drawback of this algorithm is that it is very slow to learn since in each episode and in each state a linear programming is needed. The use of linear programming significantly increases the computation cost before the system reaches convergence.

4.2 Nash-Q Learning Algorithm

Hu and Wellman [21, 34] extended the zero-sum game framework of Littman [17] to general-sum games and developed a *Nash-Q* learning algorithm for multiagent RL. To extend Q-learning to the multiagent learning domain, the joint actions of participating agents rather than merely individual actions are needed to be taken into account. Considering this important difference between single-agent and multiagent RL, the Nash-Q learning algorithm needs to maintain Q values for both the learner itself and other players. The idea is to find Nash equilibria at each state in order to obtain Nash equilibrium policies for Q value updating.

To apply the Nash-Q learning algorithm, one has to define the Nash Q-value. A Nash Q-value is defined as the expected sum of discounted rewards when all agents follow specified Nash equilibrium strategies from the next period on.

Hu and Wellman [21, 34] have shown that the Nash-Q learning algorithm in multi-player environment converges to Nash equilibrium policies with probability one under some conditions and additional assumptions to the payoff structures. More formally, the main results can be summarized in the following theorem [21, 26, 34]:

Theorem 2 *In a multiagent SG environment, an agent following the Nash-Q learning algorithm will converge to the optimal Q-function with probability one as long as all Q-functions encountered have coordination equilibria and these are used in the update rule. Furthermore, the agent using a GLIE policy will converge in behaviour with probability one if the limit equilibrium is unique.*

To guarantee the convergence, the Nash-Q learning algorithm needs to know that a Nash equilibrium is either unique or has the same value as all others. Hu and Wellman used quadratic programming to find Nash equilibrium in the Nash-Q learning algorithm for general-sum games. Littman [26] has argued the applicability of Theorem 2 and pointed out that it is hard to apply since the strict conditions are difficult to verify in advance. To tackle this difficulty, Littman [28] thereafter proposed a so-called Friend-or-Foe Q-learning (FFQ) algorithm, see the following subsection.

4.3 Friend-or-Foe Q-learning (FFQ) Algorithm

Motivated by the conditions of Theorem 2 on the convergence of Nash-Q learning, Littman [28] developed a Friend-

or-Foe Q-learning (FFQ) algorithm for RL in general-sum SGs. The main idea is that each agent in the system is identified as being either "friend" or "foe". Thus, the equilibria can be classified as either coordination or adversarial. Compared with the Nash-Q learning, the FFQ-learning can provide a stronger convergence guarantee.

Littman [28] has presented the following results to prove the convergence of the FFQ-learning algorithm:

Theorem 3 *Foe-Q learns values for a Nash equilibrium policy if there is an adversarial equilibrium; Friend-Q learns values for a Nash equilibrium policy if the game has a coordination equilibrium. This is true regardless of opponent behaviour.*

Theorem 4 *Foe-Q learns a Q-function whose corresponding policy will achieve at least the learned values regardless of the policy selected by the opponent.*

Although the convergence property of FFQ-learning has been improved over that of Nash-Q learning algorithm, a complete treatment of general-sum stochastic games using Friend-or-Foe concepts is still lacking [28]. In comparison to the Nash-Q learning algorithm, the FFQ-learning does not require learning estimates to the Q-functions of opponents. However, the FFQ-learning still require a very strong condition for application, that is the agent must know how many equilibria there are in game and an equilibrium is known to be either coordinating or adversarial in advance. The FFQ-learning itself does not provide a way to find a Nash equilibrium or identify a Nash equilibrium as being either a coordination or an adversarial one. Like Nash-Q learning, FFQ-learning also cannot apply to the system where neither coordination nor adversarial equilibrium exists.

4.4 rQ-learning Algorithm

Morales [49] developed a so-called rQ-learning algorithm for dealing with large search space problems. In this algorithm a *r-state* and an *r-action* set need to be defined in advance. A *r-state* is defined by a set of first-order relations, such as *goal_in_front, team_robot_to_the_left, opponent_robot_with_ball*, etc. An *r-action* is described by a set of pre-conditions, a generalized action, and possibly a set of post-conditions. For an *r-action* to be defined properly, the following condition must be satisfied: if an *r-action* is applicable to a particular instance of a *r-state*, then it should be applicable to all the instances of that *r-state*.

Although the rQ-learning algorithm seems to be useful for dealing with large search space problem, it may be very difficult to define a *r-state* and a *r-action* set properly, particularly in the case with incomplete knowledge on the concerned MRS. Furthermore, in the *r-state* space there is no guarantee that the defined *r-actions* are adequate to find an optimal sequence of primitive actions and sub-optimal policies can be produced [49].

4.5 Fictitious Play Algorithm

Since Nash-equilibrium-based learning has difficulty in finding Nash equilibria, the fictitious play may provide another method to deal with multiagent RL under SG framework. In the fictitious play algorithm, the beliefs of other players' policies are represented by empirical distribution of their past play [1, 32]. Hence, the players only need to maintain their own Q values, which are related to joint actions and are weighted by their belief distribution.

For stationary policies of other players, the fictitious play algorithm becomes a variant of individual Q-learning. For non-stationary policies of other players, these fictitious-play-based approaches have been empirically used in either competitive games where the players can model their adversarial opponents - called opponent modelling, or collaborative games where the players learn Q values of their joint actions - the player is called Joint Action Learner (JAL) [1].

For the fictitious-play-based approaches, the algorithms will converge to a Nash equilibrium in games that are iterated dominance solvable if all players are playing fictitious play [50]. Although the fictitious-play based learning eliminate the necessity of finding equilibria, learning agents have to model others and the learning convergence has to depend on some heuristic rules [45].

4.6 Multiagent SARSA Learning Algorithm

The Minimax-Q and Nash-Q learning algorithms are actually off-policy RL since they replace the *max* operator of individual Q-learning algorithm with their best response (Nash equilibrium policy). In RL, an off-policy learning algorithm always tries to converge to optimal Q values of optimal policy regardless of what policy is being executed.

SARSA algorithm [36] is an on-policy RL algorithm that tries to converge to optimal Q values of policy currently being executed. Considering the disadvantages of the Minimax-Q and Nash-Q learning algorithms, a SARSA-based multi-agent algorithm called as EXORL (Extended Optimal Response Learning) was developed in [32]. In [32] the fact that the opponents may take stationary policies is taken into account, rather than Nash equilibrium policies. Once opponents take stationary policies, there is no need to find Nash equilibria at all during learning. So, the learning updating can be simplified by eliminating the necessity of finding Nash equilibria if the opponents take stationary policies. In addition some heuristic rules were also employed to switch the algorithm between the Nash-equilibrium-based learning and the fictitious-play-based learning.

The basic idea of the EXORL algorithm is that the agent should learn a policy which is an optimal response to the opponent's policy, but it tries to reach a Nash equilibrium when the opponent is adaptable. Like Nash-Q leaning algorithm, the EXORL algorithm will have a difficulty when there exist multiple equilibria. Another obvious shortcoming of the EXORL algorithm is that one agent is assumed to be capable of observing the opponent's action and rewards. In some cases this will be a very serious restriction since all the agents may learn their strategies simultaneously and one agent cannot obtain the actions of the opponent at all in advance. Moreover, the opponent also may take stochastic strategy instead of deterministic policies. Observing rewards obtained by the opponent will be more difficult since

the rewards are only available after the policies are put into action practically.

Only some empirical results were given in [32] for the EXORL algorithm, there still lacks a theoretic foundation. Hence, a complete proof for convergence will be expected.

4.7 Policy Hill Climbing (PHC) Algorithm

The PHC algorithm updates Q values in the same way as the fictitious play algorithm, but it maintains a mixed policy (or stochastic policy) by performing hill-climbing in the space of mixed policies. Bowling and Veloso [44, 45] proposed a WoLF-PHC algorithm by adopting an idea of Win or Learn Fast (WoLF) and using a variable learning rate. The WoLF principle can result in the agent learning quickly when it is doing poorly and cautiously when it is performing well. The change in such a way for the learning rates will be helpful for convergence by not overfitting to the other agents' changing policies. At this point the WoLF-PHC algorithm seems to be attractive. Although many examples from MGs to zero-sum and general-sum SGs were given in [44, 45], a complete proof for the convergence properties has not been provided so far.

Rigorously speaking, the WoLF-PHC algorithm is still not a multiagent version of PHC algorithm since the learning factors of other agents in the non-Markovian environment are not taken into account at all. Thus, it is only rational and reasonable if the other agents are playing stationary strategies. In addition, the convergence may become very slow when the WoLF principle is applied [32].

4.8 Other Algorithms

Sen et al. [51] studied multiagent coordination with learning classifier systems. Action policies mapping from perceptions to actions were used by multiple agents to learn coordination strategies without shared information. The experimental results provided in [51] indicated that classifier systems can be more effective than the more widely used Q-learning scheme for multiagent coordination.

In multiagent systems, a learning agent may learn faster and establish some new rules for its own utility under future unseen situations if the experiences and knowledge from other agents are available to it. Considering this fact and the possible benefits gained from extracting proper rules out of the other agents' knowledge, a weighted strategy sharing (WSS) method was proposed in [30] for coordination learning by using the expertness of RL. In this method, each agent measures the expertness of other agents in a team and assigns a weight to their knowledge and learns from them accordingly. Moreover, the Q-table of one of the cooperative agents is changed randomly.

In tackling the coordination of multiagent systems, Boutilier [39] proposed a sequential method by allowing agents to reason explicitly about specific coordination mechanisms. In this method an extension of value iteration in which the state space is augmented with the state of the adopted coordination mechanism needs to be defined. This method allows agents to reason about the prospects for coordination, and make decisions to engage or avoid coordination problems based on expected value [39].

5 Scaling RL to Multi-Robot Systems

Multi-robot learning is a challenge for learning to act in a non-Markovian environment which contains other robots. Robots in MRSs have to interact with and adapt to their environment, as well as learn from and adapt to their counterparts, rather than only take stationary policies.

The tasks arising from MRSs have continuous state and/or action spaces. As a result, there will be difficulties in directly applying the aforementioned results on multiagent RL with finite states and actions to MRSs.

State and action abstraction approaches claim that extracting features from a large learning space is effective. The approaches include condition and behaviour extraction [2], teammate internal modelling, relationship-state estimation [5], and state vector quantisation [9]. However, all these approaches can be viewed as variants of individual Q-learning algorithms since they model other robots either as parts of environment or as stationary-policy holders.

One research on scaling reinforcement learning toward RoboCup soccer has been reported by Stone and Sutton [24]. The RoboCup soccer can be viewed as a special class of MRS and is often used as a good test-bed for developing AI techniques in both single-agent and multiagent systems. In [24], an approach using episodic SMDP SARSA(λ) with linear tile-coding function approximation and variable λ was designed to learn higher-level decisions in a keepaway subtask of RobotCup soccer. Since the general theory of RL with function approximation has not yet been well understood, the linear SARSA(λ) which could be the best understood among current methods [24] was used in the scaling of RL to RoboCup soccer. Moreover, they also claimed that it has advantages over off-policy methods such as Q-learning, which can be unstable with linear and other kinds of function approximation. However, they did not answer the open question of whether SARSA(λ) fails to converge as well.

To study the cooperation problems in learning many behaviours using RL, a subtask of RoboCup soccer, i.e., keep away was also investigated in [52] by combining SARSA(λ) and linear tile coding function approximation. However, only single-agent RL techniques, including SARSA(λ) with eligibility traces, tile coding function approximation, were directly applied to a multiagent domain. As pointed out previously, such a straightforward application of single-agent RL techniques to multiagent systems has no sound theoretic foundation. Kostiadis and Hu [53] used Kanerva coding technique [36] to produce a decision-making module for football possession in RoboCup soccer. In this application Kanerva coding was used as a generalisation method to form a feature vector from raw sensory reading while the RL uses this feature vector to learn an optimal policy. Although the results provided in [53] demonstrated that the learning approach outperformed a number of benchmark policies including a hand-coded one, there lacked a theoretic analysis on how a series of single-agent

RL techniques can work very well in multiagent systems.

The work in [2] presented a formulation of RL that enables learning in concurrent multi-robot domains. The methodology adopted in that study makes use of behaviours and conditions to minimize the learning space. The credit assignment problem was dealt with through shaped reinforcement in the form of heterogeneous reinforcement functions and progress estimators.

Morales [49] proposed an approach to the RL in robotics based on a relational representation. With this relational representation, this method can be applied over large search spaces and domain knowledge also can be incorporated. The main idea behind this approach is to represent states as sets of properties to characterize a particular state which may be common to other states. Since both states and actions are represented in terms of first order relations in the proposed framework of [49], policies are learned over such generalized representation.

To deal with the state space growing exponentially in the number of team members, Touzet [8] studied the robot awareness in cooperative mobile robot learning and proposed a method which requires a less cooperative mechanism, i.e., various levels of awareness rather than communication. The results illustrated in [8] with applications to the cooperative multi-robot observation of multiple moving targets shows some better performance than a purely collective learned behaviour.

In [11] a variety of methods were reviewed and used to demonstrate for learning in multi-robot domain. In that study behaviours were thought as the underlying control representation for handling scaling in learning policies and models, as well as learning from other agents. Touzet [16] proposed a pessimistic algorithm-based distributed lazy Q-learning for cooperative mobile robots. The pessimistic algorithm was used to compute a lower bound of the utility of executing an action in a given situation for each robot in a team. Although Q-learning with lazy learning was used, the author also neglected the important fact for the applicability of Q-learning, that is in multi-agent systems the environment is not stationary.

Park et al. [54] studied modular Q-learning based multiagent cooperation for robot soccer, where modular Q-learning was used to assign a proper action to an agent in multiagent systems. In this approach the architecture of modular Q-learning consists of learning modules and a mediator module. The function of the mediator is to select a proper action for the learning agent based on the Q-value obtained from each learning module.

Although there have been a variety of RL techniques that are developed for multiagent learning systems, very few of these techniques scale well to MRSs. On the one hand, the theory itself on multiagent RL systems in the finite discrete domains are still underway and have not been well-established. On the other hand, it is essentially very difficult to solve MRSs in general case because of the continuous and large state space as well as action space.

To deal with the problem of continuous state and action space, particularly there has been an increasing effort to apply fuzzy logic to the RL of both single and multiple agent/robot systems in recent years. Fuzzy Logic Controllers (FLCs) can be used to generalize Q-learning over continuous state spaces. The combination of FLCs with Q-learning has been proposed as Fuzzy Q-Learning (FQL) for many single robot applications [55–57].

In [58] a modular-fuzzy cooperative algorithm for multiagent systems was presented by taking advantage of modular architecture, internal model of other agent, and fuzzy logic in multiagent systems. In this algorithm, the internal model is used to estimate the agent's own action and evaluate other agents' actions. To overcome the problem of huge dimension of state space, fuzzy logic was used to map from input fuzzy sets representing the state space of each learning module to output fuzzy sets denoting action space. A fuzzy rule base of each learning module was built through the Q-learning, but without providing any convergence proof.

Kilic and Arslan [59] developed a Minimax fuzzy Q-learning for cooperative multi-agent systems. In this method, the learning agent always needs to observe the actions other agents take and uses the Minimax Q-learning to update fuzzy Q-values by using fuzzy state and fuzzy goal representation. It should be noted that the Minimax Q-learning in [59] is from the sense of fuzzy operators (i.e., *max* and *min*) and it is totally different with the Minimax-Q learning of Littman [17]. Similarly to [58], there was no proof to guarantee the optimal convergence of the Minimax fuzzy Q-learning.

The convergence proof appears to be very difficult, particularly for the multiagent RL with fuzzy generalizations. More recently, a convergence proof for single agent fuzzy RL (FRL) was provided in [60]. However, one can find that the example presented in [60] does not reflect the theoretic work of that study at all. An obvious fact is that the triangular membership functions were used in the experiment instead of the Gaussian membership functions which are the basis of the theoretic work in [60]. Therefore one can find that the proof techniques and outcomes will be very difficult to be extended to the domains of multiagent RL with fuzzy logic generalizations. Furthermore, Watkins [37] early pointed out that Q-learning may not converge correctly for other representations rather than a look-up table representation for the Q-function.

6 Main Challenges

MRSs often have all of the challenges for multiagent learning systems, such as continuous state and action spaces, uncertainties, and nonstationary environment. Since the aforementioned algorithms in Section 4 require the enumeration of states either for policies or value functions, we expect a major limitation for scaling up the established multiagent RL outcomes to MRSs.

In matrix games the joint actions correspond to particular entries in the payoff matrices. For the RL purpose, agents play the same matrix game repeatedly. For applying repeated matrix game theory to multiagent RL, the payoff structure must be given explicitly. This requirement seems to be a restriction for applying matrix games to the domains

of MRSs where payoff structure is often difficult to define in advance. Additionally, Most SGs studied in multiagent RL are of simple agent-based background where players execute perfect actions, observe complete states (or partially observed), and have full knowledge of other players' actions, states and rewards. This is not true for most MRSs. It is unfeasible for robots to completely obtain other players' information, especially for competitive games since opponents do not actively broadcast their information to share with the other players.

Moreover, adversarial opponents may not act rationally. Accordingly, it is difficult to find Nash equilibria for the Nash-equilibrium-based approaches or model their dynamics for the fictitious-play-based approaches.

Taking into account the state of the art for multiagent learning system, there is particular difficulty in scaling up the established (or partially recognized at least) multiagent RL algorithms, such as Minimax-Q leaning, Nash-Q learning, etc., to MRSs with large and continuous state and action spaces. On the one hand, most theoretic works on multiagent systems merely focus on the domains with small finite state and action sets. On the other hand there is still lacking of sound theoretic grounds which can be used to guide the scaling up the multiagent RL algorithms to MRSs. As a result, the learning performance (such as convergence, efficiency, and stability, etc.) cannot be guaranteed when approximation and generalization techniques are applied.

One important fact is that most of the multiagent RL algorithms, such as Minimax-Q leaning, Nash-Q learning is value-function based iteration method. Thus, for applying these technique to a continuous system the value-function has to be approximated by either using discretization or general approximators (such as neural networks,polynomial functions, fuzzy logic, etc.). However, some researchers has pointed out that the combination of DP methods with function approximators may produce unstable or divergent results even when applied to some very simple problems, see [61] as the references therein.

Incomplete information, large learning space, and uncertainty are major obstacles for learning in MRSs. Learning in Behaviour-Based Robotics (BBR) can effectively reduce the search space in size and dimension and handle uncertainties locally. The action space will be transformed from continuous space of control inputs into some limited discrete sets. However, the convergence proof for the algorithms using the behaviour-based strategies of MRSs will also be a very challenging problem.

When the state and action spaces of the system are small and finite discrete, the lookup table method is generally feasible. However, in MRSs, the state and action spaces are often very huge or continuous, thus the lookup table method seems inappropriate. To solve this problem, besides the state and action abstraction, function approximation and generalization appears to be another feasible solution. For learning in a partially observable and nonstationary environment in the area of multiagent systems, Abul et al. [62] presented two multiagent based domain independent coordination mechanisms,i.e. perceptual coordination mechanism and observing coordination mechanism. The advantage of their approach is that multiple agents do not require explicit communication among themselves to learn coordinated behaviors. To cope with the huge state space, function approximation and generalization techniques were used in their work. Unfortunately, the proof of convergence with function approximation and generalization techniques was not provided at all in [62]. Currently, a generic theoretic framework for proving the optimal convergence of function approximation implementation of the popular RL algorithms (such as Q-learning) has not been established yet. Interestingly, there is an increasing effort in this direction in either single-agent or multi-agent systems. For the single-agent Temporal-Difference learning with linear function approximation, Tadić [63] studied the convergence and analyzed its asymptotic properties. Under mild conditions, the almost sure convergence of Temporal-Difference learning with linear function approximation was given and the upper error bound also can be determined.

7 Concluding Remarks

Recently there have been growing interests in scaling up multiagent RL to MRSs. Although RL seems to be a good option for learning in multiagent systems, the continuous state and action spaces often hamper its applicability in MRSs. Fuzzy logic methodology seems to be a popular candidate for dealing with the approximation and generalization issues in the RL of multiagent systems. However, this scaling approach still remains open. Particularly there is a lack of theoretical grounds which can be used for proving the convergence and predicting the performance of fuzzy logic-based multiagent RL (such as fuzzy multiagent Q-learning).

For cooperative robots systems, although some research outcomes in some special cases have been available now, there are also some difficulties (such as multiple equilibrium and selecting payoff structure, etc) for directly applying them to a practical MRS, e.g., robotic soccer system.

This paper gave a survey on multiagent RL towards MRSs. The main objective of this work is to review some important advances in this field, though still not completely. Some challenging problems were discussed. Although this paper cannot provide a more complete and exhaustive survey of multiagent RL in MRSs, we still believe that it will help us more clearly understand the existing works and challenging issues in this ongoing research field.

Acknowledgment

This research is funded by the Engineering and Physical Sciences Research Council (EPSRC) under grant GR/S45508/01 (2003-2005). The authors also thank Huosheng Hu of the University of Essex for his support and helpful suggestions.

Bibliography

[1] Y. U. Cao, A. S. Fukunaga, and A. B. Kahng, "Cooperative mobile robotics: antecedents and directions," *Auton. Robot.*, vol. 4, pp. 1–23, 1997.

[2] M. J. Matarić, "Reinforcement learning in the multi-robot domain," *Auton. Robots*, vol. 4, pp. 73–83, 1997.

[3] F. Michaud and M. Matarić, "Learning from history for behavior-based mobile robots in non-stationary conditions," *Auton. Robots*, vol. 5, pp. 335–354, 1998.

[4] T. Balch and R. C. Arkin, "Behavior-based formation control for multirobot teams," *IEEE Trans. Robot. Automat.*, vol. 14, no. 6, pp. 926–939, 1998.

[5] M. Asada, E. Uchibe, and K. Hosoda, "Co-operative behaviour acquisition for mobile robots in dynamically changing real worlds via vision-based reinforcement learning and development," *Art. Intel.*, vol. 110, pp. 275–292, 1999.

[6] M. Wiering, R. Salustowicz, and J. Schmidhuber, "Reinforcement learning soccer teams with incomplete world models," *Auton. Robots*, vol. 7, pp. 77–88, 1999.

[7] S. V. Zwaan, J. A. A. Moreira, and P. U. Lima, "Cooperative learning and planning for multiple robots," 2000. [Online]. Available: citeseer.nj.nec.com/299778.html

[8] C. F. Touzet, "Robot awareness in cooperative mobile robot learning," *Auton. Robots*, vol. 8, pp. 87–97, 2000.

[9] F. Fernandez and L. E. Parker, "Learning in large co-operative multi-robot domains," *Int. J. Robot. Automat.*, vol. 16, no. 4, pp. 217–226, 2001.

[10] J. Liu and J. Wu, *Multi-Agent Robotic Systems*. CRC Press, 2001.

[11] M. J. Matarić, "Learning in behavior-based multi-robot systems: policies, models, and other agents," *J. Cogn. Syst. Res.*, vol. 2, pp. 81–93, 2001.

[12] M. Bowling and M. Veloso, "Simultaneous adversarial multi-robot learning," in *Proc. 8th Int. Joint Conf. Artificial Intelligence*, August 2003.

[13] I. H. Elhajj, A. Goradia, N. Xi, and et al, "Design and analysis of internet-based tele-coordinated multi-robot systems," *Auton. Robots*, vol. 15, pp. 237–254, 2003.

[14] L. Iocchi, D. Nardi, M. Piaggio, and et al, "Distributed coordination in heterogeneous multi-robot systems," *Auton. Robots*, vol. 15, pp. 155–168, 2003.

[15] M. J. Mataric, G. S. Sukhatme, and E. H. Østergaard, "Multi-robot task allocation in uncertain environments," *Auton. Robots*, vol. 14, pp. 255–263, 2003.

[16] C. F. Touzet, "Distributed lazy Q-learning for cooperative mobile robots," *Int. J. Advanced Robot. Syst.*, vol. 1, no. 1, pp. 5–13, 2004.

[17] M. L. Littman, "Markov games as a framework for multi-agent learning," in *Proc. 11th Int. Conf. Machine Learning*, San Francisco, 1994, pp. 157–163.

[18] M. L. Littman and C. Szepesvári, "A generalized reinforcement-learning model: convergence and applications," in *Proc. 13th Int. Conf. Machine Learning*, Bari, Italy, July 3-6 1996, pp. 310–318.

[19] C. Szepesvári and M. L. Littman, "Generalized markov decision processes: dynamic-programming and reinforcement-learning algorithms," Department of Computer Science, Brown University, Technical report CS-96-11, 1996.

[20] C. Claus and C. Boutilier, "The dynamics of reinforcement learning in coopertive multiagent systems," in *Proc. 15th National Conf. Artificial Intelligence*, Madison, WI, 1998, pp. 746–752.

[21] J. Hu and M. P. Wellman, "Multiagent reinforcement learning in stochastic games," 1999. [Online]. Available: citeseer.ist.psu.edu/hu99multiagent.html

[22] R. Sun and D. Qi, "Rationality assumptions and optimality of co-learning," Lecture Notes in Computer Science, vol. 1881. Springer, 2000, pp. 61–75.

[23] C. Szepesvári and M. L. Littman, "A unified analysis of value-function-based reinforcement learning algorithms," *Neur. Comput.*, vol. 11, no. 8, pp. 2017–2059, 1999.

[24] P. Stone and M. Veloso, "Multiagent systems: a survey from a machine learning perspective," *Auton. Robots*, vol. 8, pp. 345–383, 2000.

[25] J. Hu and M. P. Wellman, "Learning about other agents in a dynamic multiagent system," *J. Cogn. Syst. Res.*, vol. 2, pp. 67–79, 2001.

[26] M. L. Littman, "Value-function reinforcement learning in markov games," *J. Cogn. Syst. Res.*, vol. 2, pp. 55–66, 2001.

[27] M. L. Littman and P. Stone, "Leading best-response strategies in repeated games," in *17th Ann. Int. Joint Conf. Artificial Intelligence Work. Econ. Agents, Models, and Mechanism*, 2001.

[28] M. L. Littman, "Friend-or-foe Q-learning in general-sum games," in *Proc. 18th Int. Conf. Machine Learning*, Morgan Kaufman, 2001, pp. 322–328.

[29] F. A. Dahl, "The lagging anchor algorithm: reinforcement learning in two-player zero-sum games with imperfect information," *Mach. Learn.*, vol. 49, pp. 5–37, 2002.

[30] M. N. Ahmadabali and M. Asadpour, "Expertness based cooperative Q-learning," *IEEE Trans. Syst., Man, and Cyber.-Part B: Cybernetics*, vol. 32, no. 1, pp. 66–76, February 2002.

[31] G. Chalkiadakis, "Multiagent reinforcement learning: stochastic games with multiple learning players," Department of Computer Science, University of Toronto," Technical report, 2003.

[32] N. Suematsu and A. Hayashi, "A multiagent reinforcement learning algorithm using extended optimal response," in *Proc. 1st Int. Joint Conf. Auton. Agents & Multiagent Syst.*, Bologna, Italy, July 15-19 2002, pp. 370–377.

[33] G. Chalkiadakis and C. Boutilier, "Multiagent reinforcement learning: theoretical framework and an algorithm," in *2nd Int. Joint Conf. Auton. Agents & Multiagent Syst.*, Melbourne, Australia, July 14-18 2003, pp. 709–716.

[34] J. Hu and M. P. Wellman, "Nash Q-learning for general-sum stochastic games," *J. Mach. Learn. Res.*, vol. 4, pp. 1039–1069, 2003.

[35] M. Wahab, "Reinforcement learning in multiagent systems," 2003. [Online]. Available: http://www.cs.mcgill.ca/~mwahab/RL%20in%20MAS.pdf

[36] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. Cambridge, Massachusetts: MIT Press, 1998.

[37] C. J. Watkins and P. Dayan, "Q-learning," *Mach. Learn.*, vol. 8, pp. 279–292, 1992.

[38] T. Basar and G. J. Olsder, *Dynamic Noncooperative Game Theory*. London: Academic Press, 1982.

[39] C. Boutilier, "Sequential optimality and coordination in multiagent systems," in *Proc. 16th Int. Joint Conf. Artificial Intelligence*, 1999, pp. 478–485.

[40] M. G. Lagoudakis and R. Parr, "Value function approximation in zero-sum markov games," 2002. [Online]. Available: http://www.cs.duke.edu/~mgl/papers/PDF/uai2002.pdf

[41] X. Li, "Refining basis functions in least-square approximation of zero-sum markov games," 2003. [Online]. Available: http://www.xiaolei.org/research/li03basis.pdf

[42] Y. Shoham and R. Powers, "Multi-agent reinforcement learning: a critical survey," 2003. [Online]. Available: http://www.stanford.edu/~grenager/MALearning˙ACriticalSurvey˙2003˙0516.%pdf

[43] J. Hu and M. P. Wellman, "Multiagent reinforcement learning: theoretical framework and an algorithm," in *Proc. the 15th Int. Conf. Machine Learning*, San Francisco, California, 1998, pp. 242–250.

[44] M. Bowling, "Multiagent learning in the presence of agents with limitations," Ph.D. dissertation, School of Computer Science, Carnegie Mellon University, Pittsburgh, May 2003.

[45] M. H. Bowling and M. M. Veloso, "Multiagent learning using a variable learning rate," *Art. Intell.*, vol. 136, no. 2, pp. 215–250, 2002.

[46] B. Banerjee and J. Peng, "Convergent gradient ascent in general-sum games," in *Proc. 13th Europ. Conf. Mach. Learn.*, August 13-19 2002, pp. 686–692.

[47] ——, "Adaptive policy gradient in multiagent learning," in *Proc. 2nd int. joint conf. Auton. agents and multiagent systems*. ACM Press, 2003, pp. 686–692.

[48] R. S. Sutton, D. McAllester, S. Singh, and Y. Mansour, "Policy gradient methods for reinforcement learning with function approximation," in *Advanc. Neur. Inf. Proc. Syst.*. MIT Press, 12, pp. 1057–1063.

[49] E. F. Morales, "Scaling up reinforcement learning with a relational representation," in *Workshop Adaptabil. Multi-Agent Syst.*, Sydney, 2003.

[50] D. Fudenberg and D. K. Levine, *The Theory of Learning in Games*. Cambridge, Massachusetts: MIT Press, 1999.

[51] S. Sen and M. Sekaran, "Multiagent coordination with learning classifier systems," [Online]. Available: citeseer.ist.psu.edu/sen96multiagent.html

[52] S. Valluri and S. Babu, "Reinforcement learning for keepaway soccer problem," 2002. [Online]. Available: http://www.cis.ksu.edu/~babu/final/html/ProjectReport.htm

[53] K. Kostiadis and H. Hu, "KaBaGe-RL: kanerva-based generalisation and reinforcement learning for possession football," in *Proc. IEEE/RSJ Int. Conf. Intelligent Robots and Systems*, Hawaii, 2001.

[54] K.-H. Park, Y.-J. Kim, and J.-H. Kim, "Modular Q-learning based multi-agent cooperation for robot soccer," *Robot. Auton. Syst.*, vol. 35, pp. 109–122, 2001.

[55] H. R. Berenji and D. A. Vengerov, "Cooperation and coordination between fuzzy reinforcement learning agents in continuous-state partially observable markov decision processes," in *Proc. 8th IEEE Int. Conf. Fuzzy Systems* , 2000.

[56] ——, "Advantages of cooperation between reinforcement learning agents in difficult stochastic problems," in *Proc. 9th IEEE Int. Conf. Fuzzy Systems* , 2000.

[57] H. R. Berenji and D. Vengerov, "Generalized markov decision processes: dynamic-programming and reinforcement-learning algorithms," [Online]. Available: http://www.iiscorp.com/projects/multi-agent/tech˙rep˙iis˙00˙10.pdf

[58] I. Gültekin and A. Arslan, "Modular-fuzzy cooperative algorithm for multi-agent systems," Lecture Notes in Computer Science, vol. 2457. Springer, 2002, pp. 255–263.

[59] A. Kilic and A. Arslan, "Minimax fuzzy Q-learning in cooperative multi-agent systems," Lecture Notes in Computer Science, vol. 2457. Springer, 2002, pp. 264–272.

[60] H. R. Berenji and D. Vengerov, "A convergent Actor-Critic-based FRL algorithm with application to power management of wireless transmitters," *IEEE Trans. Fuzz. Syst.*, vol. 11, no. 4, pp. 478–485, August 2003.

[61] R. Munos, "A study of reinforcement learning in the continuous case by the means of viscosity solutions," *Mach. Learn.*, vol. 40, pp. 265–299, 2000.

[62] O. Abul, F. Polat, and R. Alhajj, "Multiagent reinforcement learning using function approximation," *IEEE Trans. Syst., Man, and Cybern.-Part C: Appli.n and Rev.*, vol. 30, no. 4, pp. 485–497, 2000.

[63] V. Tadić, "On the convergence of temporal-difference learning with linear function approximation," *Mach. Learn.*, vol. 42, pp. 241–267, 2001.

How to Protect Peer-to-Peer Online Games from Cheats

Haruhiro Yoshimoto[†] Rie Shigetomi[†] Hideki Imai[†]

[†]University of Tokyo
Imai Laboratory, Institute of Industrial Science,
The University of Tokyo, 4-6-1 Komaba,
Meguro-ku, Tokyo 153-8505 Japan
{ypon, sigetomi}@imailab.iis.u-tokyo.ac.jp , imai@iis.u-tokyo.ac.jp

Abstract- Recently, P2P (peer-to-peer) online game systems have attracted a great deal of public attention. They work without central servers, thus, the maintenance and organization costs have been drastically reduced. However, in P2P systems, it is difficult for game creators to prevent cheats by malicious players, due to the lack of trusted servers. In order to solve the problem, we propose a practical and secure protocol based on public key cryptography, suitable for such P2P online game systems. Our scheme guarantees that players can immediately detect when cheating by other malicious players happens, and that honest players can prove their innocence, on condition that more than half of the participants are honest. We categorized cheating into four groups, *Crack-The-Game-Software attack*, *Change-The-Input-After-Communication attack*, *Forge-The-Result attack* and *Be-Offline-When-Losing attack*, and proved that our system is secure against each attack. Moreover, we showed that our scheme is general and directly applicable to almost all of existing online games including simulation games, fighting games, and MMORPG.

1 Introduction

Recently, a P2P (peer-to-peer) online game, which is a game that players communicate each other over P2P network and play online game, has attracted a great deal of public attention. The maintenance and organization cost is reduced by the absence of the central server. Since there is no server to calculate and maintain the state of a game, all calculation is done in parallel on each player's computer by using the distributed computation technique. Any group of people can start playing a game anytime as there's no need to set up and maintain a dedicated server. A P2P online game has potential to become an ideal game platform in the near future.

One thing that has to be kept in mind is that not all the players are honest. Some malicious players try to *cheat* the game in various ways to win the game easily. They can crack the game software and alter the behavior of the game software to make it easy for them to win the game. This is a serious problem in online games. A large number of people can be affected by this, unlike offline games where only local participants are affected. The effect is not limited to virtual world only because sometimes items in game worlds are traded by players in real world with real money. At present, the most general way to prevent cheating is to watch the behavior of all players and find out which one is cheating. This is often done by watching the connection to the server. However it is not possible to apply this scheme to P2P online game because the server does not exist. A player can not know which player is honest and which player is not.

The lack of the server allows malicious players to cheat in another ways. The attacks like *Change-The-Input-After-Communication attack*, *Forge-The-Result attack* and *Be-Offline-When-Losing attack* become possible. The detail of these attack will be explained later.

In this paper, we present a protocol to make it possible to detect a cheating on P2P online games. The protocol further allows you to prove a session is played corretly when more than half players are honest. Our scheme is based on public key cryptography and we use digital signature to prove a game is played correctly.

Our protocol has strong relation to other cryptographic schemes, especially to fair coin flipping, mental poker, multi-party computation, and fair exchange. However, our protocol differs from those schemes in the configuration and the goal. This will be discussed later at section 3.

This paper is organized as follows. First, we define our security model in section 2. Then, we introduce related works and compare them to our security model in section 3. The section 4 explains cryptographic tools used in our scheme. Our scheme is described in section 5. We discuss the security of our scheme in section 6. Finally, we conclude in section 7.

2 Security Model

In this section, we describe the setting of the game, define what an adversary can do, and define what the security is.

We consider that these definitions are proper to the current P2P online game.

2.1 Settings

game session The game can be separated into small sessions. we call it *game session*

number of players On a P2P online game, not all players participate in one game session. The game such as fighting game is usually playable for only 2 players for one session, so we define as follows.

Definition 1 : (*m,n*)-P2P online game *The P2P online game which m players participate in one game session, and the number of players who consist the P2P network is n, is called (m,n)-P2P online game.*

The fighting game is usually *(2,n)-P2P online game*.

input and the output Firstly, we define the *output* of the game software as follows.

Definition 2 : Output Data *The output data which is needed to play next game session.*

Informally so-called Savedata or Highscore is the *output*. Then, we do not call the display status, or other output of the software as *output*. Note that the *output* of one game session is different for each player and each player holds their own *output*.

Next, we define the *input*. Usually the input of the game software is the input from the device such as mouse, keyboard and joypad . In this paper we define the *input* as follows.

Definition 3 : Output O *For output O, the data which is needed to output O is called input.*

This means that not only the input from the mouse, keyboard and joypad but also the random seed, previous output, communication from other players, current time and so on, is the *input*.

2.2 Adversary Model

The attacks which an adversary will do is as follows. Our scheme can cope with these attacks.

Attack 1 : ***Crack-The-Game-Software attack*** The attacker cracks the software and alter the behavior of the software. Altering the *output* of the software or communication data of the software is classified to this attack. Getting huge amount of money by using cheat codes is a kind of this attack.

Attack 2 : ***Change-The-Input-After-Communication attack*** Watch the communication from other player and after that he decide the input to the game software so that attacker can win the game. This is done because the communication is not fair but the communication of the game have to be done simultaneously.

Attack 3 : ***Forge-The-Result attack*** The attacker broadcasts the *output* to the all players as if he won the game. This can also be done by acting as if he is the another player.

Attack 4 : ***Be-Offline-When-Losing attack*** The attacker becomes offline when he is losing, he tries to act like the game session did not happen. In the real world, this must be treated as a give-up. Also malicious player can claim that the opponent gave up, even if the opponent is not yet given up.

These attacks includes almost all the attacks which is so-called "Cheating". Preventing from these cheats is enough for general online games.

2.3 Definition of Security

We define that our system is secure for cheating when following condition holds.

Definition 4 : Secure System *That the system is secure is anyone without the software creator is not possible to success any four attacks: Crack-The-Game-Software attack, Change-The-Input-After-Communication attack, Forge-The-Result attack, and, Be-Offline-When-Losing attack.*

2.4 Examples

We show that the adversary model defined in subsection 2.2 is reasonable. We introduce two examples, chess and Rock-Paper-And-Scissors, and show how to cheat these games. Then we show that those cheats are classificated to the attacks define in subsection 2.2. Note that **Attack 1** is a very strong attack, and can be done not only to P2P online game but can be done to offline game. The **Attack 3** can be done to any game that saves the result of the game to their computer, so we do not describe the **Attack 3**. Also, the **Attack 4** can be done to any game which becoming offline during the game means give-up, so we do not describe the **Attack 4**.

2.4.1 Example : chess

When playing chess on P2P network game, the example of **Attack 1** is as follows

- Add the time left for player.
- Move the piece on anywhere on the board.
- Add any piece on the board.

In chess, player can know whether the opponent is cheating or not because all the data of chess is open to each player, so he can claim that the opponent is cheating, but the cheater can claim that the opponent is cheating too. Therefore, the player have to prove which player is playing to others. In our scheme, even if the player cannot know whether the opponent is cheating or not, the player cannot cheat.

The **Attack 2** cannot be done in chess because the move of the chess is not done simultaneously.

2.4.2 Example : Rock-Paper-And-Scissors

When playing Rock-Paper-And-Scissors on P2P network game, the example of **Attack 1** is as follows

- Make hand that can win to any hand, like joker in the card game.

The **Attack 2** can be done in Rock-Paper-And-Scissors . Sending the hand to each player have to be done simultaneously because cheater can change the hand into the stronger one after getting the opponent's hand.

3 Related Works

In this section, we introduce the related works. These schemes are popular in cryptography and well studied. The settings of these schemes are similar to our setting, and have a strong relationship to our work. We will explain what different parts between previous scheme and our scheme.

3.1 Multi-Party Computation

Let f denote a given n-ary function, and suppose parties P_1, ... , P_n each hold an input value x_1, ... , x_n, respectively. A secure multiparty computation for f is a joint protocol between parties P_1, ... , P_n for computing $y = f(x_1, ..., x_n)$ securely. That is, even when a certain fraction of the parties is corrupted, (i) each party obtains the correct output value y and (ii) no information leaks on the input values of the honest parties beyond what is implied logically by the value of y and the values of the inputs of the corrupted parties.

There are various applications for some situations such as multi-party computation, electronic voting, fair coin flipping[Blum82], mental poker[SRA81][Crepeau87].

Secure two party computation was first investigated by Yao[Yao82][Yao86] and was later generalized to multi-party computation. The seminal paper by Goldreich et al.[GMW87] proves the existence of a secure solution for any functionality.

The difference to our situation One difference between Multi-Party Computation and our situation is that we do not have to make the input secret. When playing poker, knowing the input of other player after the game might be useful because he can know the strategy of other players, and that information is valuable. On the other hand, for general online game, knowing the strategy of other players might be not so useful.

One reason is that the software is not perfectly one-way. It means that watching the communication from other player, they can guess the input. Consider fighting game, they can see what other player is doing, and can guess the input.

Other informations about input is kept secret but knowing it after the game does not matter. Another reason is that the game which strategy is very important, usually the input is open to public. Consider chess, the game of famous player is open to public, but they can win even their strategy is known. What we need is a scheme that the players can not change the input after the communication, or game.

We can easily understand that, making a scheme that all player can hide their inputs to others is useful, but our scheme can not do this. Another difference between Multi-Party Computation and our situation is that the output is not same for each players.

3.2 Fair Exchange

The setting of fair exchange is as follows. Suppose that Alice buys a book (we call this b) from Bob. Alice has to pay the price for the book (we call this a) to Bob. If Alice sends a to Bob first and Bob is dishonest, then Bob would not send b to Alice, so Alice loses. If Bob sends b first and Alice is dishonest, then Alice would not send a to Bob, so Bob loses. To trivial solution to this problem is using TTP. First, Alice sends a to TTP, and Bob sends b to TTP. Then TTP sends b to Alice, and TTP sends a to Bob. Micali[Micali03] proposed a scheme such that TTP is used only when either player cheated. Formal security model of fair exchange is as follows.

Definition of security The outcome of the fair exchange must be either of the following.

- Alice gets "b" and Bob gets "a"
- Alice gets nothing and Bob gets nothing

The difference to our situation Suppose that Bob is dishonest and did not send b to Alice even after getting a. Even though Bob is dishonest, getting b has no meaning to Alice, because not sending b means that Bob gives up and Alice wins. Alice only have to claim to other players that Bob gave up. So our scheme do not need TTP. What we need is that Alice to "prove" to other player that Bob gave up.

4 Preliminaries

In this section, we explain the cryptographic tools that we use in our scheme. These tools are well implemented and distributed in many platforms.

4.1 Hash Function

Hash function is a function that makes a message digest. Usually Hash function $H()$ takes an arbitrary length of string as an input and outputs constant length string (such as 128 bit). Security requirements for hash function $H()$ are follows.

One-wayness Given a message X, it is should be "hard" to find a message M satisfying X = H(M).

Collision resistance It should be "hard" to find two messages $M1 \neq M2$ such that $H(M1) = H(M2)$.

SHA-1[RFC3174] and MD5[RFC1321] are well known and widely used hash algorithm. these algorithm are very fast and the security is well considered. However, the collision was found in MD5[WFLY04] so our recommend of hash function is SHA-1.

4.2 Digital Signature

Firstly, we explain about public key cryptography. Consider Alice and Bob. Bob wants Alice to be able to send secure

secret messages to him. To permit this, Bob generates a key pair consisting of two related "keys". One key is called the public key, and the other, the secret or private key. In the most useful of these algorithms, it is impractical, even for a well-founded organization, to compute the private key from the public key (this is the private key / public key property). Bob keeps his private key secret, and publishes his public key.

Alice retrieves Bob's public key and, using it to control the appropriate encryption algorithm, scrambles or encrypts the message. For quality algorithms (at least in the belief of those well-informed on the subject), once encrypted using Bob's public key, the ciphertext cannot be descrambled or decrypted without the private key. Thus, no one who intercepts the ciphertext will be able to read it, even knowing Bob's public key. When Bob receives the message, he decrypts it using his private key (kept secret since generation time and so known only to him). Therefore, the message will be secure against the unauthorized, and Bob and Alice do not need a "secure channel" to exchange a shared key.

Next we explain about digital signature. Bob wants to send a message to Alice and wants to be able to prove it came from him (but does not care whether anybody else reads it). In this case, Bob sends a cleartext copy of the message to Alice, along with a copy of the message encrypted with his private key (not the public one). Alice (or any other recipient) can then check whether the message really came from Bob by decrypting the ciphertext version of the message with Bob's public key and comparing it with the cleartext version. If they match, the message was really from Bob, because the private key was needed to create the signature and no one but Bob has it. The ciphertext version is Bob's digital signature for the message because anyone can use Bob's public key to verify that Bob created it.

More usually, Bob applies a cryptographic hash function to the message and encrypts the resulting message digest using his private key. This makes the signature much shorter and thus saves both time (since hashing is generally much faster than public-key encryption), and space (since even an enciphered message digest is much shorter than the ciphertext version of the entire plaintext).

In this paper, what we call "signature" is using a hash function to digest the message. Note that the player cannot get any information of message from the signature, because of the onewayness of the hash function.

The practical signature scheme is presented in [BR96][Schnorr91]

4.3 Commitment Scheme

A commitment scheme is a cryptographic protocol between two parties, a sender and a receiver. The protocol consists of two-phases.

Commit Phase A sender sends commitment of message M to a receiver.

Opening Phase A sender sends message M to a receiver.

The security requirements for Commitment scheme is as follows.

Concealing A receiver cannot get any information of M from the commitment of M.

Binding A sender cannot change the message M after the Commit Phase.

We will introduce an example. Alice wants to commit a message M to Bob. At the commit phase, Alice sends her signature $Sign_{SK_A}(M)$ to Bob as a commitment. Bob cannot know what the message is because he only has the signature of the message so it is concealing. At the unveil phase Alice sends the message M to Bob. If Alice sends message different from M, Bob can know that by verifying the signature. so it is binding. We use this scheme as a commitment scheme. Also, this is not a perfect bit commitment because the security of the scheme relies on the onewayness of the signature scheme. Other commitment scheme is shown in [Blum82]

4.4 Notation

We notate as follows.

$a|b$: concatenation of a and b
P_1, P_2, ... , P_i, ... , P_n : the players with player ID i
SK_{P_1}, SK_{P_2}, ... , SK_{P_n} : the secret key for the each player
$Sign_i(M)$: the signature of the message M using the secret key of P_i
$input_i$: *input* of P_i
$output_i$: *output* of P_i

5 Our Scheme

In this section we describe our scheme and show that this scheme is tolerant to attacks defined in section 2. Our scheme uses *Game Simulator* to cope with Attack 1 and Attack 3, and use *fair communication* to cope with Attack 2.

5.1 Game Simulator

First we define a *Game Simulator*. *Game Simulator* is defined as follows.

Definition 5 : Game Simulator *The Game Simulator is a software which outputs the same output as original game software for any input, and also outputs the communication data.*

Game Simulator is used to check whether the input-output behavior is as same as original game software. If the input-output behavior does not match with *Game Simulator* then the some kind of cheat was done.

Claim 1 : The Game Simulator works much faster than the original game software

Reason : This is because the simulator can skip the process

of waiting the player's input, and also displaying the game to the display. Usually these processes are bottleneck of the software and almost all the execution time of the software is exhausted by this process.

Although the implementation of this *Game Simulator* is not easy, this will be a very useful tool against cheating. The general and easy way to implement this *Game Simulator* is an open problem.

5.2 Fair Communication

We define *fair communication* as follows.

Definition 6 : Fair Communication *The communication which both players commits their message before communication, we call this fair communication.*

Actually when P_i wants to send $data_i$ to P_j, and P_j wants to send $data_j$ to P_i in a *fair communication*, P_i first sends $Sign_i(data_i)$ to P_j as a commitment, and P_j sends $Sign_j(data_j)$ to P_i as a commitment.

Claim 2 : Consider when P_i wants to send $data_i$ to P_j, and P_j wants to send $data_j$ to P_i, and the order of the communication is not decided in the game rule. If altering the $data_j$ after knowing $data_i$ can make advantage to P_j, and altering the $data_i$ after knowing $data_j$ can make advantage to P_i, then that communication must be fair communication

Note that not all the communication must be fair. One reason is because the communication data is not always useful datas to play game for others. Which communication must be fair and which do not have to be fair is decided by the people who implements the software. To do this decision automatically is an open problem.

5.3 Protocol Details

The protocol of our scheme for (m,n)-P2P online game is as follows. Each game session consists of one or more numbers of communication phases. Note that Phase 0-1 and 0-2 is not done on the first game session.

Phase 0-1 All players who join the game session sends previous *output* and all the signature acquired at Phase 10 to all the other players who are joining to the game session.

Phase 0-2 All players who join the game session verifies the signature which was send by other players, and if the number of the correct signature is more than $n/2$ then the output sent from other player is a "*output* with no cheat". If verif

Phase 1 All players share same random session ID *SID*.

Phase 2 Suppose that P_i wants to send communication data $data_k$ to P_j. This is the beginning of the communication phase k.

Phase 3 P_i chooses random number R_k, and sends $s_{ik} = Sign_i(SID|R_k|i|j|data_k)$ to P_j , s_{ik} is the commitment of $data_k$.

Phase 4 P_j sends $s_{jik} = Sign_j(s_{ik})$ to P_i , P_i verifies the signature. s_{jik} is the proof that P_j received the s_{ik}.

Phase 5 If this communication phase must be a *fair communication*, then P_i waits until he gets $s_{jk'} = Sign_j(SID|R_{k'}|j|i|data_{k'})$ from P_j, and P_i sends $s_{ijk'} = Sign_i(s_{jk'})$ to P_j

Phase 6 P_i sends $data_k$ and R_k to P_j.

Phase 7 If this communication phase must be a *fair communication*, then P_i gets $data_{k'}$ and $R_{k'}$ from P_j, so P_i can check whether $s_{jk'}$ is a valid signature.

Phase 8 Increment k, Repeat Phase 2 till Phase 7 in parallel for each players until all the communication ends.

Phase 9 Each player sends *SID*, all the signature he received and sent, all the R,i,j and $input_i$, to all players including the players those who are not joining to this game session.

Phase 10 P_m verifies the signature he received, run the *Game Simulator* with the $input_i$ and get the output $output_i$ and communication data. If the communication data is right, he sends $s_m = Sign_m(output_i)$ to P_i.

Note that $input_i$ implies all $data_k$, so it is possible to verify the signature at Phase 10.

Table 1: Protocol Detail

Phase	P_i		P_j	The other players(P_m)
0-1		$output\prime_i$, $Sign_*(output\prime_i)$ →		
0-2		← $output\prime_j$, $Sign_*(output\prime_j)$		
	...	...	...	
3	Unfair Communication Phase	$s_{ik_1} = Sign_i(SID\|R_{k_1}\|i\|j\|data_{k_1})$ →		
4		← $s_{jik_1} = Sign_j(s_{ik_1})$		
6		$data_{k_1}$, R_{k_1} →		
	...	...	...	
3	Unfair Communication Phase	$s_{ik_2} = Sign_i(SID\|R_{k_2}\|i\|j\|data_{k_2})$ →		
4		← $s_{jik_2} = Sign_j(s_{ik_2})$		
6		$data_{k_2}$, R_{k_2} →		
	...	...	...	
3	Fair Communication Phase	s_{ik_3} →		
5		← s_{jk_4}		
4		s_{ijk_4} →		
5		← s_{jik_3}		
6		$data_{k_3}$, R_{k_3} →		
7		← $data_{k_4}$, R_{k_4}		
	...	...	...	
9		i, j , $input_i$, SID, s_{ik_*}, s_{jk_*}, s_{jik_*}, s_{ijk_*} , R_{k_*} →		Run *game simulator* with
10		← $Sign_m(output_i)$		$input_i$, and get $output_i$
9		i, j , $input_i$, SID , s_*, R_{k_*} →	Run *game simulator* with	
10		← $Sign_j(output_i)$	$input_i$ and get $output_i$	
9			j, i , $input_j$, SID , s_*, R_{k_*} →	Run *game simulator* with
10			← $Sign_m(output_j)$	$input_j$, and get $output_j$
9	Run *game simulator* with	← j, i , $input_j$, SID , s_*, R_{k_*}		
10	$input_j$ and get $output_j$	$Sign_i(output_j)$ →		

$*$ is a wildcard character, and it means send all the datas that matches to it. (example. R_{k_*} : send R_{k_1} , R_{k_2} , ... , R_{k_n})

5.4 Protocol Analysis

Tolerance to Attack 1 Because all the player have the *Game Simulator*, even the attacker changes the behavior of the game software, he cannot change the *output* of the *Game Simulator*, and if attacker alters the communication data they can detect it. Even if the communication data is the original one,while the cheater alters the $output_i$ to $output\prime_i$, the signature he can get is signature for $output_i$, so he cannot use the $output\prime_i$ in the Phase 0-1 of next game session. Thus, the Attack 1 does not succeed.

Tolerance to Attack 2 Because using *fair communication*, Attack 2 does not succeed. In detail, after sending the communication data $data_k$ in Phase 4, the player cannot change the communication data afterwards. If the player cheats and send different data such as $data\prime_k$ in Phase 6, verifying the s_{ik} fails.

Tolerance to *Attack 3* The Attack 3 is to cheat Phase 9, Phase 10. However, because P_i can forge s_{ik} but can not forge s_{jik}, the attack does not succeed. For explanation, assume that there is no s_{jik}. With the lack of the signature, if the player P_i wants to cheat, player P_i can make different communication data $data\prime_k$ and sign it with his secret key and make new $s\prime_{ik}$. This is a valid sinagure, so it passes the check in Phase 10 and get the signature of $output\prime$ which is different from the original $output_i$. On the other hand, if we use s_{jik}, the player cannot alter $data_k$, that is because the P_i does not know the secret key of P_j, he cannot create valid s_{jik}.

Tolerance to Attack 4 When Attack 4 occurs, because of the existence of s_{jik}, P_i can prove that he really sent $data_k$ to P_j and P_j received it. Therefore, the player can prove all the conduct of the communication. The other players can know the progress of the game, and decide the $output$ and sign it. Let us consider RPG. All the conduct of the communication is signed, so the other players can know the item player got, the place where he gone, and so on. Therefore they can "sign to the result" even if the game is not yet done. Without the signature, the other players have to discard the result because of the possibility of the cheat.

6 Discussion

The penalty of the cheat What shall player be done when he cheats? We do not present the answer for this because this depends on the type of the game. For fighting game, the penalty might be that cheater loses once. Note that the Attack 4 might occur accidentally for reason like traffic trouble, so the penalty must not be so heavy.

How to improve the efficiency If we only have to cope with *Attack 1*, *Attack 2* and *Attack 3*, we do not have to sign each communication data separately , except the communication data which is used in fair communication. Thus, we can concatenate those communication data, and sign it. The signature will not be sent every communication phase, but several times each communication phase. However, note that doing this weakens the tolerance to *Attack 4* because doing this, it makes player impossible to prove the occurrence of all the communication when the opponent player goes offline. However, the player can prove the occurrence of the communication that is done before the last signature has been send, so concatinating the communication data that will be send in some period (about 1sec or so) , and signing it whold be practical.

How to Authenticate the keys Consider the situation that Alice got a Bob's public key of from Bob himself. Is this really the public key of Bob? Someone might be acting as Bob. We have to make sure that the public key is Bob's. This is called authentication of public key. If TTP such as PKI's certification authority[PKI] is available, the solution is easy. TTP creates all the public key and secret key pair and send it to players. If player wants to authenticate the public key, he just asks TTP who is the owner of the public key. However, our scheme is on P2P network, so we can not rely on TTP. All the pair of public key and private key is created by each players. One solution for this problem is that using the authentication scheme that is used by PGP. All the player evaluate the other players and set the level of trust. This is called "web of trust".

Discard-the-Result attack Although player cannot alter his *output*, but he can discard it. If he "lose" or some bad thing has happened at the game session, he can discard that *output* and use previous *output* instead. How can we deal with this problem? One answer is that every player shares the hash value of the other player's *output*.

Bot attack In this paper, we did not consider of *bot attack*, this attack is to use software such that plays game automatically. Some times this software is called macro. To prevent this, we have to decide whether human or the computer is playing the game. Using CAPTCHA[ABL02] might be useful to cope with this attack. CAPTCHA is a scheme to separate the human and the computer, using an problem that hard to solve for computer. However, when using CAPTHA in the game, we have to take care not to bother the play.

7 Conclusions

We suggested a new solution to detect cheating on P2P online game. The protocol is suitable for popular P2P online games, such as fighting and RPG games. Our scheme uses digital signature as a commitment scheme. With the commitment scheme, you can verify a game is played fairly. The *Game Simulator* is used to check nobody cheated in a game. Implementing generic *Game Simulator* and deciding which communication must be *fair communication* is an open problem and is a future work.

Bibliography

[RFC3174] RFC3174 - US Secure Hash Algorithm 1

[RFC1321] RFC1321 - The MD5 Message-Digest Algorithm

[ABL02] L. von Ahn, M. Blum, and J. Langford."Telling humans and computers apart (automatically)". *CMU Tech Report CMUCS- 02-117*, February 2002.

[BR96] M. Bellare and P. Rogaway. "The exact security of digital signatures: How to sign with RSA and Rabin," *In Proceedings of EUROCRYPT'96*, volume 1070 of Lecture Notes in Computer Science. Springer-Verlag, 1996.

[Blum82] M. Blum, "Coin Flipping by Telephone", *Proc. 24th IEEE Compcon*, 1982, pp. 133-137.

[Crepeau87] C. Crépeau, "A zero-knowledge poker protocol that achieves confidentiality of the players' strategy or how to achieve an electronic poker face," Proceedings in CRYPTO 86, Lecture Notes in Computer Sciences 263, Springer, 1987 , pp.239-247

[GMW87] O. Goldreich, S. Micali, and A. Wigderson. "How to play any mental game - a completeness theorem for protocols with honest majority". *In 19th ACM Symposium on the Theory of Computing*, 1987, pages 218-229.

[Micali03] S. Micali. "Simple and fast optimistic protocols for fair electronic exchange". *2003 ACM Symposium on Principles of Distributed Computing*, pages 12-19, 2003.

[PKI] NICT PKI Home, http://csrc.nist.gov/pki/

[Schnorr91] C. P. Schnorr. Efficient signature generation by smart cards. Journal of Cryptology, 4:161-174, 1991.

[SRA81] A.Shamir, R. L. Rivest, L. M. Adleman, "Mental Poker," *The Mathematical Gardner*, Wadsworth, 1981, pp.37-43

[WFLY04] X. Wang, D. Feng, X. Lai, H. Yu. "Collisions for Hash Functions MD4, MD5, HAVAL-128 and RIPEMD," *Cryptology ePrint Archive*, http://eprint.iacr.org/2004/199/

[Yao82] A. C. Yao. "Protocols for secure computations". *In Proc. 23rd IEEE Symposium on Foundations of Computer Science (FOCS '82)*, pages 160-164. IEEE Computer Society, 1982.

[Yao86] A. C. Yao. "How to generate and exchange secrets". *In Proceedings of the 27th IEEE Symposium on Foundations of Computer Science*. IEEE, 1986, pages 162-167.

Author Index