

Evolution-based Discovery of Hierarchical Behaviors

Justinian P. Rosca and Dana H. Ballard

Computer Science Department
University of Rochester
Rochester, NY 14627
E-mail: {rosca,dana}@cs.rochester.edu

Abstract

Procedural representations of control policies have two advantages when facing the scale-up problem in learning tasks. First they are implicit, with potential for inductive generalization over a very large set of situations. Second they facilitate modularization. In this paper we compare several randomized algorithms for learning modular procedural representations. The main algorithm, called Adaptive Representation through Learning (ARL) is a genetic programming extension that relies on the discovery of subroutines. ARL is suitable for learning hierarchies of subroutines and for constructing policies to complex tasks. ARL was successfully tested on a typical reinforcement learning problem of controlling an agent in a dynamic and nondeterministic environment where the discovered subroutines correspond to agent behaviors.

Introduction

The interaction of a learning system with a complex environment represents an opportunity to discover features and invariant properties of the problem that enable it to tune its representations and optimize its behavior. This discovery of modularity while learning or solving a problem can considerably speed up the task, as the time needed for the system to “evolve” based on its modular subsystems is much shorter than if the system evolves from its elementary parts (Simon 1973). Thus machine learning, or machine discovery approaches that attempt to cope with non-trivial problems should provide some hierarchical mechanisms for creating and exploiting such modularity.

An approach that incorporates modularization mechanisms is genetic programming (GP) with automatically defined functions (ADF) (Koza 1994). In ADF-GP computer programs are modularized through the explicit use of subroutines. One shortcoming of this approach is the need to design an appropriate *architecture* for programs, i.e. set in advance the number of subroutines and arguments, as well as the nature of references among subroutines.

A biologically inspired approach to architecture discovery introduced in (Koza 1995) is based on new operations for duplicating parts of the genome. Code

duplication transformations seem to work well in combination with crossover, as duplication protects code against the destructive effects of crossover. Duplication operations are performed such that they preserve the semantics of the resulting programs.

This paper presents an alternative, *heuristic*, solution to the problem of architecture discovery and modularization, called Adaptive Representation through Learning (ARL). ARL adopts a search perspective of the genetic programming process. It searches for good solutions (representations) and simultaneously adapts the architecture (representation system). In GP, the representation system or *vocabulary* is given by the primitive terminals and functions. By adapting the vocabulary through subroutine discovery, ARL biases the search process in the language determined by the problem primitives.

The paper outline is as follows. The next section describes the task used throughout the paper: controlling an agent in a dynamic and nondeterministic environment, specifically the Pac-Man game. Section 3 estimates a measure of the complexity of the task by evaluating the performance of randomly generated procedural solutions, their iterative improvement through an annealing technique and hand-coded solutions. Section 4 presents details of the ARL approach. Its performance and a comparison of results with other GP approaches are described in the following two sections. This work is placed into a broader perspective in the related work section, before concluding remarks.

The Application Task

We consider the problem of controlling an agent in a dynamic environment, similar to the well known Pac-Man game described in more detail in (Koza 1992). An agent, called Pac-Man, can be controlled to act in a maze of corridors. Up to four monsters chase Pac-Man most of the time. Food pellets, energizers and fruit objects result in rewards of 10, 50 and 2000 points respectively when reached by Pac-Man. After each capture of an energizer (also called “pill”), Pac-Man can chase monsters in its turn, for a limited time (while monsters are “blue”), for rewards of 500 points

for capturing the first monster, 1000 points for the next etc. The environment is nondeterministic as monsters and fruits move randomly 20% of the time.

The problem is to learn a controller to drive the Pac-Man agent in order to acquire as many points as possible and to survive monsters. The agent’s movements rely on the current sensor readings, possibly past sensor readings and internal state or memory. Pac-Man knows when monsters are blue and has smell-like perceptions to sense the distance to the closest food pellet, pill, fruit and closest or second closest monster. Overt action primitives move the agent along the maze corridors towards or backwards from the nearest object of a given type.

Representation Approaches

The Pac-Man problem is a typical reinforcement learning (RL) task. In response to actions taken, the agent receives rewards. Often rewards are delayed. The task in reinforcement learning is to learn a policy maximizing the expected future rewards.

Formally, an agent policy is a mapping

$$p : \mathcal{P} \times \mathcal{S} \rightarrow \mathcal{A}$$

where \mathcal{P} is the set of perceptions, \mathcal{A} the set of actions and \mathcal{S} the agent state space. When searching for a policy, the size of the hypotheses space is the number of such mappings i.e. $|\mathcal{A}|^{|\mathcal{P}| \cdot |\mathcal{S}|}$. The number of required examples for PAC-learnability is proportional to the logarithm of the hypothesis space size. This outlines two major problems. First, explicitly representing the state space is undesirable from a learnability perspective. Second, the large number of perceptions given by various distance values is critical. In contrast to explicit representations, implicit representations such as programs have the potential of better generalization with a smaller number of training examples. This makes GP a candidate approach to learn policies.¹

Besides generalization, an implicit representation of the agent policy would improve on two important aspects: compressibility and modularity. Compressibility means that a solution is representable in a concise form. Also, small solutions may have increased generality, according to Ockham’s razor principle. Representation modularity is important from a scale-up perspective. Ideally, a modular representation organizes the knowledge and competences of the agent such that local changes, improvements or tuning do not affect the functioning of most other components. Researchers in “behavior-based” artificial intelligence (Maes 1993) talk about integrated competences or behaviors as given decompositions of the problem. We are interested in discovering decompositions that naturally emerge from the interaction agent-environment.

¹For the same reason, parameterized function approximators have been used to replace table lookup in reinforcement learning.

A general implicit representation that can be made modular is a procedural representation. Candidate solutions are programs. Modules naturally correspond to functions or subroutines.

Pac-Man Procedural Representations

Programs that define agent controllers will be built based on perception, action and program control primitives. The agent perception primitives return the Manhattan distance to the closest food pellet, pill, fruit and the closest or second closest monster (SENSE-DIS-FOOD etc.) The “sense blue” perception is combined with an if-then-else control primitive into the if-blue (IFB) lazy evaluation function which executes its first argument if monsters are blue, otherwise executes its second argument. The action primitives advance/retreat the agent with respect to the closest object of a given type and return the distance² between the agent and the corresponding object (ACT-A-PILL/ACT-R-PILL, etc.)

The above primitives were combined to form programs in two ways. In the first alternative (*A*) we used the if-less-than-or-equal (IFLTE) lazy function which compares its first argument to its second argument. For a “less-than” result the third argument is executed. For a “greater-or-equal” result the fourth argument is executed. Representation *A* can develop an intricate form of state due to the side effects of actions that appear in the condition part of the IFLTE function.

In the second alternative (*B*) primitives are *typed*. We used the if-then-else IFTE lazy evaluation function that takes a BOOL type as first argument and two more ACT type arguments. All actions and control operators have type ACT, all logical expressions have type BOOL, and distances have type DIST. Relational operators (<, =, ≥, ≠) and logical operators (AND, OR, NOT) are used to generate complex logical expressions. The programs that can be written in representation *B* are equivalent to decision trees, which makes them easy to understand and analyze.

While playing the game the learner remembers mistakes that led the game to a premature end (agent eaten by a monster). Equally useful is to *improve* the learner’s control skills that were acquired in previous runs. Similarly, a machine learning technique aims to generate better and better programs that control the agent by borrowing fragments from good previously learned programs. To facilitate modular learning we defined the modular alternatives of representations *A* and *B* called *M-A* and *M-B*. In these cases programs were made up of three fragments: two subroutines of two arguments each and a main program. Each fragment had access to the entire problem vocabulary. In addition, the second subroutine could also invoke the first one, and the main program could invoke either subroutine.

²If the shortest path/closest monster/food are not uniquely defined, then a random choice from the valid ones is returned by the corresponding function.

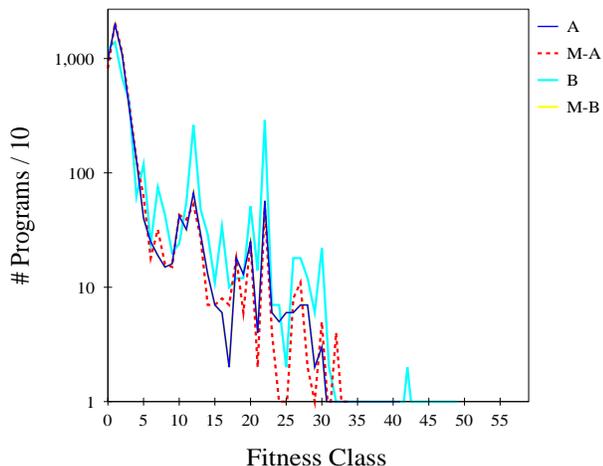


Figure 1: Distribution of fitness values over 50000 random programs generated using the ramped-half-and-half method from [Koza, 92]. Each fitness class covers an interval of 100 game points. The fitness of each program is the average number of game points on three independent runs.

Programs are simulated and are assigned a *fitness* equal to the number of points acquired by Pac-Man on average in several simulated games, which are the fitness cases. Solution generality was estimated by testing the learned programs on new cases.

Next we explore several methods for designing programs to represent procedural agent policies.

First Solutions

Random Search

The parse trees of random programs are created recursively in a top-down manner. First a function is chosen as the label of the program root node and then for each formal argument of the function new subprograms are generated recursively. (Koza 1992) describes a method called *ramped-half-and-half* for generating very diverse random tree structures. No particular structure is favored due to both randomly choosing node labels and randomly varying the tree depth and balance.

In order to test the Pac-Man representations we generated both simple and modular random programs. Figure 1 shows the distribution of fitness values obtained in the four alternative representations. The best programs were obtained with problem representation *B*, followed by the modular versions *M-A* and *M-B*.

Simulated Annealing

A simple technique for iterative improvement is simulated annealing (SA) (Kirkpatrick, Gelatt, & Vecchi 1983). Although SA performs well in continuous spaces, it has also been applied to combinatorial optimization problems in search for optima of functions of discrete variables. For example, a similar search technique, GSAT (Selman & Kautz 1993), offers the

best known performance for hard satisfiability problems. The space of programs is also non-continuous. SA has been previously tested on program discovery problems (O’Reilly 1995). The SA implementation for program search by O’Reilly used a mutation operation that modifies subtrees through an insertion, deletion or substitution sub-operation trying to distort the subtree only slightly. In contrast, we define a primitive mutation of a program p that replaces a randomly chosen subtree of with a new randomly generated subtree. The SA-based algorithm for iterative improvement of *programs* will be called PSA from now on.

The cooling schedule is defined by the following parameters: the initial temperature T_0 , the final temperature T_f , the length of the Markov chain at a fixed temperature L and the number of iterations G (Laarhoven 1988). PSA used a simple rule to set these parameters: “accept the replacement of a parent with a 100-point worse successor with a probability of 0.125 at the initial temperature and a probability of 0.001 at the final temperature.”³ No attempts have been made to optimize these parameters other than these initial choices.

Hand-Coding

We carefully designed modular programs for both representations *A* and *B*. This was not as easy as it might appear. The best programs were found after a couple of hours of code twiddling. Contrary to intuition, simpler programs proved to be better than the most complex programs we designed. The performance results of these first attempts to learn or design a Pac-Man controller are summarized in Table 1. The best result was obtained with PSA and representation *M-A*.

Table 1: Performance, in average number of game points, of best evolved programs and carefully hand-coded programs. The maximum depth of randomly generated programs was 8. PSA was seeded with the best solution from 500 randomly generated programs. Training was done on three cases and testing on 100 cases.

Representation	A	M-A	B	M-B
Random	4110.0	3420.0	4916.7	4916.7
PSA	5436.7	7646.7	5790	5663.3
Hand-coding	-	7460	-	5910

Architecture Discovery in ARL

In the standard genetic programming algorithm (SGP) an initial population of randomly generated programs are transformed through crossover, occasional mutation, and fitness proportionate reproduction opera-

³We obtained $T_0 = 48, T_f = 14$. We also chose $L = 100$ and $G = 25000$. The value of G is justified by the desire to make similar the overall computational effort (the total number of programs evaluated) for PSA and the GP techniques to be described next. These parameters determine an exponential cooling parameter of: $e^{\frac{L}{G} \ln \frac{T_f}{T_0}}$.

tions. SGP relies on a hypothesis analogous to the genetic algorithm (GA) building block hypothesis (Holland 1992), which states that a GA achieves its search capabilities by means of “building block” processing. Building blocks are relevant pieces of partial solutions that can be assembled together in order to generate better partial solutions.

Our modular representations are modeled after the automatically defined functions (ADF) approach (Koza 1994). ADF is an extension of GP where individuals are represented by a fixed number of components or *branches* to be evolved: a predefined number of function branches and a main program branch. Each function branch (consider for instance three such branches called ADF_0 , ADF_1 , and ADF_2) has a predefined number of arguments. The main program branch (*Program-Body*) produces the result. Each branch is a piece of LISP code built out of a specific vocabulary and is subject to genetic operations. The set of function-defining branches, the number of arguments that each of the function possesses and the vocabulary of each branch define the *architecture* of a program. The architecture imposes the possible hierarchical references between branches. For instance, if we order the branches in the sequence $ADF_0, ADF_1, ADF_2, Program-Body$ then a branch may invoke any component to its left.

(Rosca 1995) analyzed how this preimposed hierarchical ordering biases the way ADF searches the space of programs. In the “*bottom-up evolution hypothesis*” he conjectured that ADF representations become *stable* in a bottom-up fashion. Early in the process changes are focused towards the evolution of low level functions. Later, changes are focused towards higher levels in the hierarchy of functions (see also (Rosca & Ballard 1995)). ARL will consider a bottom-up approach to subroutine discovery as the default.

The ARL Algorithm

The nature of GP is that programs that contain useful code tend to have a higher fitness and thus their offspring tend to dominate the population. ARL uses heuristics which anticipate this trend to focus search.

It searches for good individuals (representations) while adapting the architecture (representation system) through subroutine invention to facilitate the creation of better representations. These two activities are performed on two distinct tiers (see Figure 2). GP search acts at the bottom tier. The fitness proportionate selection mechanism of GP favors more fit program structures to pass their substructures to offspring. At the top tier, the subroutine discovery algorithm selects, generalizes, and preserves good substructures. Discovered subroutines are reflected back in programs from the memory (current population) and thus adapt the architecture of the population of programs.

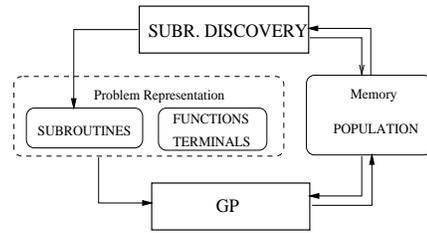


Figure 2: Two-tier architecture of the ARL algorithm.

Discovery of Subroutines

The vocabulary of ARL at generation t is given by the union of the terminal set \mathcal{T} , the function set \mathcal{F} and the set of evolved subroutines \mathcal{S}_t (initially empty). $\mathcal{T} \cup \mathcal{F}$ represents the set of primitives which is fixed throughout the evolutionary process. In contrast, \mathcal{S}_t is a set of subroutines whose composition may vary from one generation to the next. \mathcal{S}_t extends the representation vocabulary in an adaptive manner. New subroutines are discovered and the “least useful” ones die out. \mathcal{S}_t is used as a pool of additional problem primitives, besides \mathcal{T} and \mathcal{F} for randomly generating some individuals in the next generation, $t + 1$.

The subroutine discovery tier of the ARL architecture attempts to automatically discover useful subroutines and adapt the set \mathcal{S}_t . New subroutines are created using blocks of genetic material from the population pool. The major issue is the detection of “useful” blocks of code. The notion of usefulness is defined by two concepts, *differential fitness*, and *block activation* which are defined next. The subroutine discovery algorithm is presented in Figure 3.

Differential Fitness The concept of differential fitness is a heuristic which focuses block selection on programs that have the biggest fitness improvement over their least fit parent. Large differences in fitness are presumably created by useful combinations of pieces of code appearing in the structure of an individual. This is exactly what the algorithm should discover. Let i be a program from the current population having raw fitness $F(i)$. Its differential fitness is defined as:

$$\text{DiffFitness}(i) = F(i) - \min_{p \in \text{Parents}(i)} \{F(p)\} \quad (1)$$

Blocks are selected from those programs satisfying the following property:

$$\max_i \{\text{DiffFitness}(i)\} > 0 \quad (2)$$

Figure 4 shows the histogram of the differential fitness defined above for a run of ARL on the Pac-Man problem. Each slice of the plot for a fixed generation represents the number of individuals (in a population of size 500) vs. differential fitness values. The figure shows that only a small number of individuals improve on the fitness on their parents. ARL will focus on such individuals in order to discover salient blocks of code.

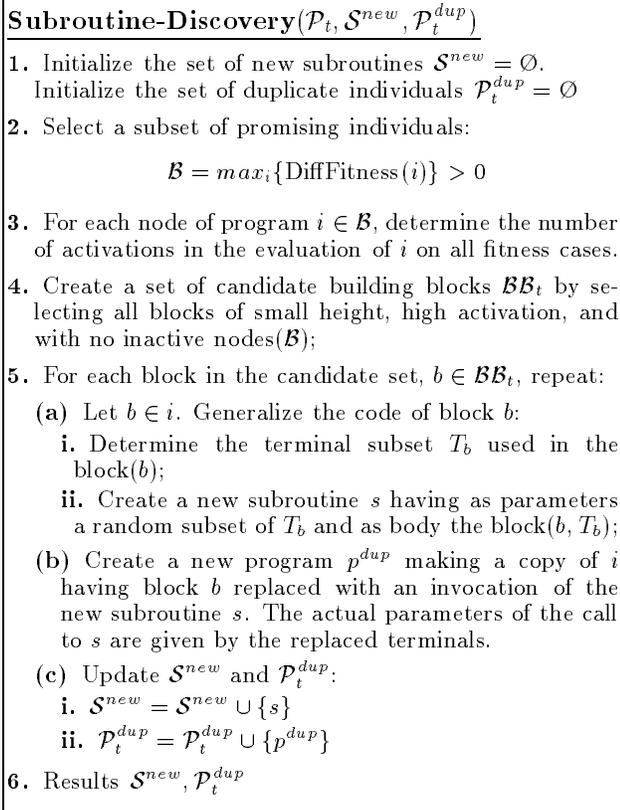


Figure 3: ARL extension to GP: the subroutine discovery algorithm for adapting the problem representation. \mathcal{S}^{new} is the set additions to \mathcal{S}_t . Duplicate individuals \mathcal{P}_t^{dup} are added to the population before a next GP selection step.

Block Activation During repeated program evaluation, some blocks of code are executed more often than others. The more active blocks become “candidate” blocks. *Block activation* is defined as the number of times the root node of the block is executed. Salient blocks are active blocks of code from individuals with the highest differential fitness. In addition, we require that all nodes of the block be activated at least once or a minimum percentage of the total number of activations of the root node.⁴

Generalization of Blocks The final step is to formalize the active block as a new subroutine and add it to the set \mathcal{S}_t . Blocks are generalized by replacing some random subset of terminals in the block with variables (see Step 5a in Figure 3). Variables become formal arguments of the created subroutine.⁵

⁴This condition is imposed in order to eliminate from consideration blocks containing introns and hitch-hiking phenomena (Tackett 1994). It is represented by the pruning step (4) in Figure 3.

⁵In the typed implementation block generalization additionally assigns a signature to each subroutine created. The subroutine signature is defined by the type of the function that labels the root of the block and the types of the ter-

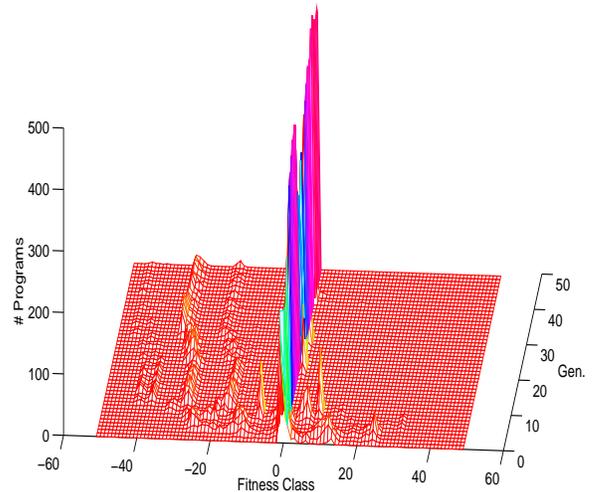


Figure 4: Differential fitness distributions over a run of ARL with representation A on the Pac-Man problem. At each generation, only a small fraction of the population has $\text{DiffFitness} > 0$.

Subroutine Utility

ARL expands the set of subroutines \mathcal{S}_t whenever it discovers new subroutine candidates. All subroutines in \mathcal{S}_t are assigned utility values which are updated every generation. A subroutine’s utility is estimated by observing the outcome of using it. This is done by accumulating, as *reward*, the average fitness values of all programs that have invoked s over a fixed time window of W generations, directly or indirectly.

The set of subroutines co-evolves with the main population of solutions through creation and deletion operations. New subroutines are automatically created based on active blocks as described before. Low utility subroutines are deleted in order to keep the total number of subroutines below a given number.⁶

ARL at Work

ARL inherited the specific GP parameters.⁷ In addition, ARL used our implementation of typed GP for runs with representation B . It was run only for a maximum of 50 generations. The ARL-specific parameters are the time window for updating subroutine utilities (10) and the maximum number of subroutines (20). Next we describe a typical trace of ARL on a run with representation B , which obtained the best overall results, and present statistical results and comparisons among SGP, ADF, PSA, and random generation of programs.

minals selected to be substituted by variables.

⁶In order to preserve the functionality of those programs invoking a deleted subroutine, calls to the deleted subroutine are substituted with the actual body of the subroutine.

⁷For SGP and ADF population size = 500, the number fitness cases = 3, the crossover rate = 90% (20% on leaves), reproduction rate = 10%, number of generations = 100.

<p>Generation 3.</p> <ul style="list-style-type: none"> • S1663 (ACT 1 (BOOL)) (LAMBDA (A0) (IFTE (NOTEQUAL (SENSE-DIS-PILL) A0) (ACT-A-FOOD) (ACT-A-PILL)))) <p>Generation 10.</p> <ul style="list-style-type: none"> • S1749 (ACT 0) (LAMBDA () (IFTE (< (SENSE-DIS-FRUIT) 50) (ACT-A-FRUIT) (ACT-A-MON-1)))) <p>Generation 17.</p> <ul style="list-style-type: none"> • S1765 (ACT 1 (DIS)) (LAMBDA (A0) (IFTE (< (SENSE-DIS-FRUIT) A0) (ACT-A-FRUIT) (ACT-R-PILL)))) • S1766 (BOOL 1 (DIS)) (LAMBDA (A0) (< (SENSE-DIS-FRUIT) A0)) <p>Generation 30.</p> <ul style="list-style-type: none"> • S1997 (ACT 0) (LAMBDA () (IFTE (< (SENSE-DIS-FOOD) (SENSE-DIS-PILL)) (ACT-A-FOOD) (S1765 19)))) <p>Generation 33. Best-of-run individual:</p> <ul style="list-style-type: none"> • (IFB (IFTE (S1766 21) (ACT-A-FRUIT) (IFB (S1997) (S1663 21)))) (IFB (IFTE (S1766 22) (ACT-A-FRUIT) (ACT-A-FOOD)) (IFTE (S1766 22) (ACT-A-PILL) (IFTE (< (SENSE-DIS-FOOD) (SENSE-DIS-PILL)) (ACT-A-FOOD) (S1749))))))
--

Table 2: Evolutionary trace of an ARL run on the Pac-Man problem (representation *B*). For each discovered subroutine the table shows the signature (type, number of arguments and type of arguments if any), and the subroutine definition.

Programs evolved by ARL are modular. ARL usually evolves tens of subroutines in one run, only twenty of which are preserved in \mathcal{S}_t at any given time t . Subroutines have small sizes due to the explicit bias towards small blocks of code. The hierarchy of evolved subroutines allows a program to grow in effective size (i.e. in expanded structural complexity, see (Rosca & Ballard 1994)) if this offers an evolutionary advantage.

For instance, the best-of-generation program evolved by ARL in one run with problem representation *B* is extremely modular. ARL discovered 86 subroutines during the 50 generations while it ran. Only 5 subroutines were invoked by the best-of-run program which was discovered in generation 33. These useful subroutines form a three-layer hierarchy on top of the primitive functions. Each of the five subroutines is effective in guiding Pac-Man for certain periods of time. A trace of this run is given in Table 2.

The five subroutines define interesting “behaviors.” For example, S1749 is successfully used for attracting monsters. S1765, invoked with the actual parameter value of 19 defines a fruit-chasing behavior for blue periods. The other subroutines are: an applicability predicate for testing if a fruit exists (S1766), a food-hunting behavior (S1997), and a pill-hunting behavior (S1663). The main program decides when to invoke and how to combine the effects of these behaviors, in response to state changes.

Comparison of Results

In order to detect differences in performance and qualitative behavior from PSA, the current SGP experi-

Table 3: Comparison of generalization performance of different Pac-Man implementations: average fitness of evolved solutions over 100 random test cases.

Rep.	Rand	PSA	SGP	ADF	ARL	Hand
A	1503	2940	2906	1569	3611	2424
B	2321	4058	3370	3875	4439	2701

ments used crossover as its only genetic operation and a zero mutation rate.

Most importantly, we are interested in the generality of the best solutions obtained with random, PSA, SGP, ADF and ARL approaches (see Table 3). For the random and PSA cases we took the results of the runs with representations M-A and M-B which were the best. We tested all solutions on the same 100 random test cases. ARL achieved the best results. Hand solutions were improved from the initial ones reported in Section 3. We also determined the 95% confidence interval in the average number of points of a solution. For example, the ARL solution for representation *B* has an interval of ± 90 points, i.e. the true average is within this interval relative to the average with 95% confidence. From the modularity perspective solutions, ADF modularity is confined by the fixed initial architecture. ARL modularity emerges during a run as subroutines are created or deleted. SGP solutions are not explicitly modular.

Other Related Work

Tackett studied, under the name “gene banking,” ways in which programs constructed by genetic search can be mined off-line for subexpressions that represent salient problem traits (Tackett 1994). He hypothesized that traits which display the same fitness and frequency characteristics are salient. Unfortunately, many subexpressions are in a hierarchical “part-of” relation. Thus it may be hard to distinguish “hitchhikers” from true salient expressions. Heuristic reasoning was used to interpret the results, so that the method cannot be automated in a direct manner. In contrast, in ARL salient blocks have to be detected efficiently, on-line. This is possible because candidate blocks are only searched for among the blocks of small height present in individuals with the highest differential fitness.

Functional programming languages, such as LISP, treat code and data equivalently. ARL takes advantage of this feature to analyze the behavior of the code it constructs and to decide when and how subroutines can be created. More generally, pure functional languages such as the ML language treat functions and values according to a formal set of rules. As a consequence, the process of formal reasoning applied to program control structures can be automated. One recent example of such an attempt is ADATE (Olsson 1995). ADATE iteratively transforms programs in a top-down manner, searching the space of programs written in a subset of ML for a program that explains a set of

initial training cases. ADATE creates new predicates by abstraction transformations. Algorithms that use predicate invention are called constructive induction algorithms. Predicate invention is also a fundamental operation in inductive logic programming where it helps to reduce the structural complexity of induced structures due to reuse. More importantly, invented predicates may generalize over the search space thus compensating for missing background knowledge. A difficult problem is evaluating the quality of new predicates (Stahl 1993).

The predicate invention problem is related to the more general problem of *bias* in machine learning. In GP, the use of subroutines biases the search for good programs besides offering the possibility to reuse code. An adaptive learning system selects its bias automatically. An overview of current efforts in this active research area appeared recently in (Gordon & DesJardins 1995).

Conclusions

Although the Pac-Man is a typical reinforcement learning task it was successfully approached using GP. GP worked well for the task because it used an implicit representation of the agent state space. Therefore, GP solutions acquire generality and can be modularized. This last feature was particularly exploited in ARL. Programs, as structures on which GP operates, are symbolically expressed as compositions of functions. By applying the differential fitness and block activation heuristics, the ARL algorithm manages to discover subroutines and evolve the architecture of solutions which increase its chances of creating better solutions.

Evolved subroutines form a hierarchy of “behaviors.” On average, ARL programs perform better than the ones obtained using other techniques. A comparison among solutions obtained using the PSA, GP, ADF and ARL algorithms with respect to search efficiency and generalization is ongoing on the Pac-Man domain as well as other problems. Additionally, a comparison with memoryless and state-maintaining reinforcement learning algorithms is also worth further investigation. ARL can be studied as an example of a system where procedural bias interacts with representational bias (Gordon & DesJardins 1995). This may shed additional light on how GP exploits structures and constructs solutions.

References

Gordon, D. F., and DesJardins, M. 1995. Evaluation and selection of biases in machine learning. *Machine Learning* 20:5–22.

Holland, J. H. 1992. *Adaptation in Natural and Artificial Systems, An Introductory Analysis with Applications to Biology, Control and Artificial Intelligence*. Cambridge, MA: MIT Press. Second edition (First edition, 1975).

Kirkpatrick, S.; Gelatt, C.; and Vecchi, M. 1983. Optimization by simulated annealing. *Science* 220:671–680.

Koza, J. R. 1992. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press.

Koza, J. R. 1994. *Genetic Programming II*. MIT Press.

Koza, J. R. 1995. Gene duplication to enable genetic programming to concurrently evolve both the architecture and work-performing steps of a computer program. In Mellish, C. S., ed., *IJCAI*, volume 1, 734–740. Morgan Kaufmann.

Laarhoven, v. P. J. M. 1988. *Theoretical and computational aspects of simulated annealing*. Netherlands: Centrum voor Wiskunde en Informatica.

Maes, P. 1993. Behavior-based artificial intelligence. In *SAB-2*. MIT Press.

Olsson, R. 1995. Inductive functional programming using incremental program transformation. *Artificial Intelligence* 74:55–81.

O’Reilly, U.-M. 1995. *An Analysis of Genetic Programming*. Ph.D. Dissertation, Ottawa-Carleton Institute for Computer Science.

Rosca, J. P., and Ballard, D. H. 1994. Hierarchical self-organization in genetic programming. In *11th ICML*, 251–258. Morgan Kaufmann.

Rosca, J. P., and Ballard, D. H. 1995. Causality in genetic programming. In Eshelman, L., ed., *ICGA 95*, 256–263. San Francisco, CA., USA: Morgan Kaufmann.

Rosca, J. P. 1995. Genetic programming exploratory power and the discovery of functions. In McDonnell, J. R.; Reynolds, R. G.; and Fogel, D. B., eds., *Evolutionary Programming IV Proceedings of the Fourth Annual Conference on Evolutionary Programming*, 719–736. San Diego, CA, USA: MIT Press.

Russell, S. J., and Norvig, P. 1995. *Artificial Intelligence: A Modern Approach*. Englewood Cliffs, New Jersey: Prentice Hall.

Selman, B., and Kautz, H. A. 1993. An empirical study of greedy local search for satisfiability testing. In *AAAI*. AAAI Press/The MIT Press. 46–51.

Simon, H. A. 1973. The organization of complex systems. In Howard H. Pattee, G. B., ed., *Hierarchy Theory; The Challenge of Complex Systems*. New York. 3–27.

Stahl, I. 1993. Predicate invention in ILP - an overview. In Brazdil, P. B., ed., *ECML*, 313–322. Springer-Verlag.

Tackett, W. A. 1994. *Recombination, Selection and the Genetic Construction of Computer Programs*. Ph.D. Dissertation, University of Southern California.