Learning to play Mario

Shiwali Mohan, John E. Laird
A.I. Laboratory, Computer Science and Engineering, University of Michigan, Ann Arbor
{shiwali, laird}@umich.edu

Abstract

Computer Games are interesting test beds for research in Artificial Intelligence and Machine Learning. Games usually have continuous state spaces, large action spaces and are characterized by complex relationships between components. Without applying abstraction and generalizations, learning in computer games domain becomes infeasible. Through this work, we investigate some designs that facilitate tractable reinforcement learning in symbolic agents developed using Soar architecture operating in a complex domain, Infinite Mario. Object oriented representations of the environment greatly simplify otherwise complex state spaces. We also demonstrate that imposing hierarchy in problem structure greatly reduces the complexity of the tasks and aids in learning generalized policies that can be transferred across similar tasks.

1 Introduction

Infinite Mario [1] is a Reinforcement Learning domain developed for Reinforcement Learning Competition 2009. It is a variant of Nintendo Super Mario [2], a complete side-scrolling video game with destructible blocks, enemies, fireballs, coins, chasms, and platforms. It requires the player to move towards right to reach the finish line, earning points and powers along the way by collection coins, mushrooms, fire flowers and killing monsters. The game is interesting because it is requires agent to reason and learn at several levels; from modeling sensory-motor primitives to path-planning and devising strategies to deal with various components of the environment.

Soar [3] is a symbolic cognitive architecture that has been extensively used in designing intelligent agents across various domains. The central theme of the Soar project is developing systems that exhibit intelligent cognitive behavior. Many animals and human beings learn from trial and error experimentation and use corresponding positive/negative reward for reinforcement. Soar-RL [4] is an attempt to capture this behavior. A naive agent should capable of developing its own strategies and tactics to deal with the environment through experimentation and exploration. The reward could be environment driven (positive points for successful completion of the task) or intrinsic to the agent (emotion or affect). Through this work we aim to assess a Soar-RL agent's capabilities in a difficult reinforcement learning tasks. Mario is a sufficiently complex game that requires skill and strategy to accomplish a high score. It has complicated state space that necessitates abstractions at many levels.

One of the important challenges involved in devising learning agents that play computer games is defining representations that allow transfer of knowledge learned on one episode to other similar episodes or similar games. An agent that has experienced one level should be better at learning the next level than an agent that has never experienced any learning.

The rest of the paper is organized as follows. Section 2 reviews research work done on reinforcement learning in computer games. In sections 3, 4, 5 and 6, we give brief introduction to Reinforcement Learning, Hierarchical Reinforcement Learning, Object Oriented Representations in RL and Soar RL. Domain specifications are given in section 7. Section 8 describes the approaches taken in this project and presents few empirical results. We end with observing limitations of the current design in Section 9, future work in Section 10, some observations on Soar – RL mechanism in Section 11 and concluding remarks in section 12.

2 Related Work

Game playing is a major area of interest for research in machine learning. Traditionally, the research has concentrated on learning in strategy games such as tic-tac-toe, checkers, backgammon, chess or bridge. However, recently computer and video games have received increased attention [5,6] because they present challenges which are close to real world problems like that of the enormity of information in a highly self-contained and circumscribed environment. Reinforcement learning, of late, has emerged as an attractive approach to look at learning in Computer Games. Much of the previous work in application of RL in computer games addresses limited game scenarios. Spronck et al. [6] and Ponsen et al. [7] have implemented RL techniques in several computer games and have reported good learning performances in considerably reduced state and action space. However, little work has been done in abstraction of large spaces to facilitate good learning behavior. A work that comes close to our research has been done by Ponsen et al. [8] on relational hierarchical learning in computer games.

3 Reinforcement Learning

Reinforcement learning (RL) defines an approach to solving problems rather than specifying all the intricacies involved in solving the problem through to implementation. RL problems are defined in terms of an agent interacting with an environment. The agent's experience of the environment is encapsulated in a q function, which spans the different state-action pairs possible in the environment and associates an accumulative value with each state-action pair. This q function is updated upon receiving feedback, defined by the reward function, from the environment. This reward function is statically declared at the outset of a problem and is outside of the influence of the agent, and therefore steers the development of the q function. It is important to note that rewards can be either negative or positive, discouraging or encouraging the agent accordingly. The agent follows a policy that maps states to actions, and collaborates with the q function in dictating the behavior of the agent [9].

Most RL research is based on the framework of Markov decision processes (MDPs). MDPs are sequential decision making problems for fully observable worlds with a Markovian transition model. The goal of the agent is to maximize long term cumulative reward. Its initial behavior is purely trial and error driven, but as the agent starts to form an impression about the states, and their relative merits, it becomes increasingly important for it to strike a balance between the exploration of new states which may provide maximum reward, and the exploitation of existing knowledge.

At the core of every RL problem is the Q function. In its most simple form this would be a table containing a q-value associated with every state. These values are an indication of the long term reward associated with a particular state. When we leave a state we adjust its current value towards the value of the state we are entering. The α factor determines the extent to which future rewards affect the current value estimation. This approach of "backing up" the values is an example of temporal-difference learning and is a way of propagating information about future rewards backwards through the value function.

The agent can have many different policies. With a purely greedy policy, the agent will always select the state transition believed to offer the greatest long-term reward. For reinforcement learning to discover the optimal policy, it is necessary that the agent sometimes choose an action that does not have the maximum predicted value. Such exploration is necessary because actions may be undervalued. This situation can occur both during the initial learning of a task and as a result of change in the dynamics or reward structure of a task. With an epsilon-greedy method the agent will select the best state transition the majority of the time and take exploratory moves on all the other state transitions. The frequency of these exploratory moves is determined by the value of epsilon ε utilized by the policy. It is possible to vary ε, in order to have an initially open minded agent that gains confidence in its value function as its experience increases over time. One problem inherent in the epsilon-greedy approach is that the agent explores indiscriminately and is as likely to explore an obviously unattractive avenue as it is to explore a promising one. The softmax policy associates a probability of selection with every state transition that increases with the predicted value of the destination state.

As the size of state space increases, the number of parameters to be learned grows exponentially. Recent attempts to combat this problem have been in the direction of exploiting the hierarchical structure of task at hand. The agent is not required to get to a decision at each step; it can invoke execution of temporally extended activities that follow their own policies until termination.

4 Hierarchical Reinforcement Learning

Hierarchical RL (HRL) [10] is an interesting and intuitive approach to scale up RL to more complex problems. In HRL, a complex task is decomposed into a set of simpler subtasks that can be solved independently. Each subtask in the hierarchy is modeled as a single MDP and allows appropriate state, action and reward abstractions to augment learning compared to a flat representation of the problem. Additionally, learning in a hierarchical setting can facilitate generalization, e.g., knowledge learned by a subtask can be transferred to other subtasks. HRL relies on the theory of Semi-Markov decision processes (SMDPs). SMDPs differ from MDPs in that actions in SMDPs can last multiple time steps. Therefore, in SMDPs actions can either be primitive actions or temporally extended actions.

5 Object Oriented Reinforcement Learning

Object oriented representations were introduced in [12] as a way that provides a natural way of modeling domains while enabling generalizations. The environment is described in terms of objects – entities that affect agent's behavior. The state of the agent is represented as current relationships between different objects in the environment and their interactions guide the decision making process.

6 Soar RL

Soar (States, Operators and Results) is a production-rule based architecture [3] that enables the design of cognitive agents. Soar agents solve problems based on a cycle of state-perception, proposal of actions, action selection, and action execution. Soar has been augmented with Reinforcement Learning [4] to capture statistical-based learning. The RL mechanism automatically learns value functions for working memory state- proposed operator pair as a Soar agent executes. Optimal behavior of the agent is defined by the reward and a discount factor. The agent executes such that it maximizes the expected reward value. The reinforcement learning algorithm captures agent's experience into the Q-function. The function Q(s,a) maps the state-operator pair to its expected sum of future rewards (Q-value). As the agent repeatedly works its way through the environment, the RL algorithm learns successively closer approximations of the true Q-value. These values are stored in productions so that if multiple operators are proposed for a single state, the agent can act optimally by selecting the operator with the highest Q-value. Soar is capable of performing both Q learning and sarsa. A soar-agent can execute epsilon-greedy and Boltzmann exploration policies. This project uses sarsa and epsilon-greedy policy, performance of soar-agents with Boltzmann policy on this task still remains to be evaluated.

Soar-HRL is built upon operator no-change impasse. When multiple operators are proposed for the same state and if no preference order exists, soar creates a subgoal to solve the impasse. In HRL, each impassed operator is a temporally extended action that achieves some immediate subgoal.

7 Infinite Mario – Formal Specifications [RL Competition 2009]

Infinite Mario has been implemented on RL-Glue; a standard interface that allows connecting reinforcement learning agents, environments, and experiment programs together. The agent is provided information about the current visual scene through arrays. It can perform actions in the environment by setting the values of action integer array.

7.1 State Observations

The state space has no set specification. The visual scene is divided into a two-dimensional [16 x 22] matrix of tiles. At any given time-step the agent has access to a char array generated by the environment corresponding to the given scene. Each tile (element in the char array) can have one of the many values that can be used by the agent to determine it the corresponding tile in the scene is a brick, or contains a coin etc. The details of valid values the tiles can take are given in Appendix A.
For every monster visible, the agent is provided with the type of the monster, its current location, and its speed in both x and y direction. Mario is one of the monsters in the observation. Details can be viewed at [1].

7.2 Actions

The actions available to the agent are same as those available to a human player through gamepad in a game of Mario. The agent can choose to move right or left or can choose to stay still. It can jump while

moving or standing. It can move at two different speeds. All these actions can be accomplished by setting the values of action array.

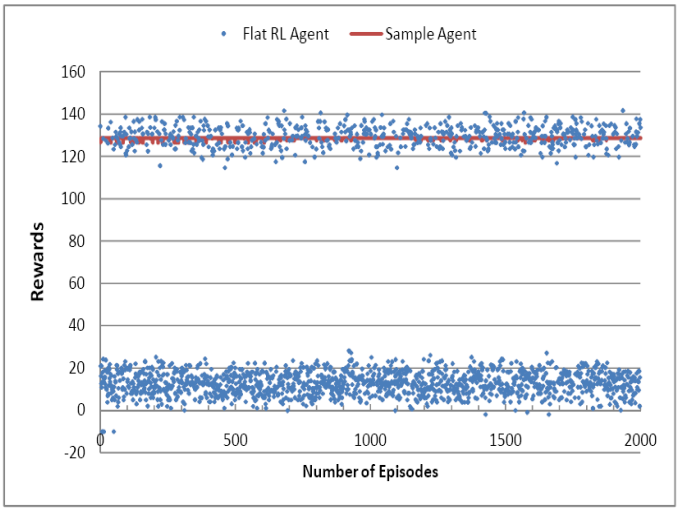
7.3 Reward

The reward structure varies across different MDPs. The agent earns a huge positive reward when it reaches the finish line. Every step the agent takes in the environment, it gets a small negative reward. The agent gets some negative reward if it dies before reaching the finish line and gets some positive reward by collecting coins, mushrooms and killing monsters. The goal is to reach the finish line which is at a fixed distance from the starting point such that the total reward earned in an episode is maximized.

8 Approaches and Results

8.1 Flat RL

We started with trying to implement a flat-RL agent. If the complete visual-space is considered as a state, the state space becomes enormous. There are 16 x 22 tiles and each tile can take 13 different values. This counts up to 13352 states. To reduce the state space, we considered the 5 x 5 tile space around Mario agent as a single state. The agent designed in such a fashion failed to show any significant learning. The results are shown in Graph 1. At best the agent is as good as a random agent. When it finishes one episode successfully, the high score earned by it is more that the sample agent but even after training on 2000 episodes the agent shows little improvement. This is in line with the results presents by Ponsen et. al. [8], where they show flat RL agents take a long time to reach to an optimal policy in Computer Games domain.



Graph 1: The average performance of RL agent and sample agent over 5 experiments of 2000 episodes each in Infinite Mario Level Type 0, seed 121. The x axis denotes the number agent of training episodes and the y axis denotes the reward earned in an episode

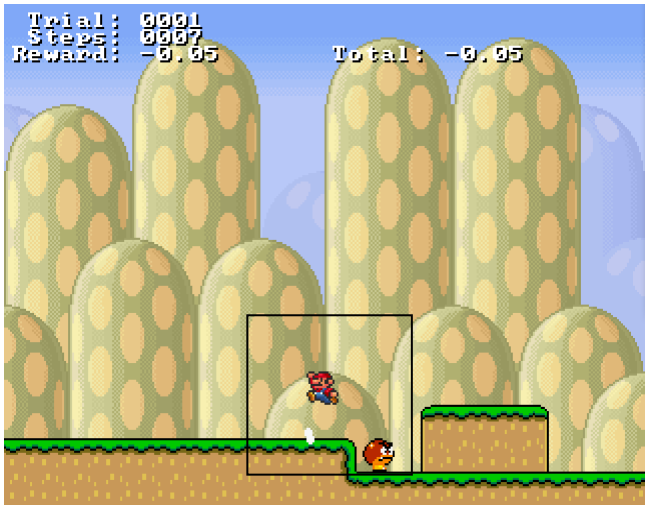


Figure 1: State space in Flat RL

8.2 Abstractions

To facilitate any form of learning in a complex state space like that of Mario, abstractions and generalizations are absolutely necessary. We present some of the abstractions we made to achieve good results in the task.

State Abstraction:

The key idea here is to move from the low level tile by tile representation of the visual space to a view that is composed of objects and their relationships with the agent and each other. Thus, the state representation changes from the form "the tile at position *(x,y)* is of type *t*" to "there exists a pit at a distance of three tiles". In Soar, these types of elaborations can be easily done by using simple heuristics. These heuristics can be accounted for as background knowledge humans have about continuity of space and properties of certain objects. All the objects present in the environment are indentified by the agent through specific symbols. The agent can then use these symbols to reason about and plan its way through the environment. This symbolic representation also allows for addition of more facts and background knowledge in the agent. Rules like "if there is a pit ahead, then jump while moving right" can be easily encoded. A typical Mario episode contains many objects. Most common amongst them are monsters, coins, question blocks, pipes (which may or may not contain a monster), raised platforms and the finish line. Once these object have been identified from the visual scene, each object is also assigned its relative position from Mario in both x and y dimensions. As this information is recorded, Mario agent can reason about if any object is close enough to initiate some action.

Operator Abstraction:

In 1990, a GOMS [11] analysis was used to predict behavior of a human expert on the task on playing Mario. The analysis was implemented on Soar and the authors demonstrated that using some very simple heuristics Soar could predict the behavior of the human expert. The analysis makes a distinction between key-stroke level and functional level operators. Key-stroke level operators (KLOs) are the primitive actions available to the human user, the action that can be performed using a keyboard or a joystick. For a task of Mario, these would include moving right or left, jumping and shooting which can be input to the game using a controller. On the other hand, functional level operators (FLOs) are a collection of keystroke level operators that when performed in succession, perform a specific task like moving towards goal, or grabbing a coin.

The game of Mario contains limited number of functional level operators, as do most of the computer games. For completing most episodes successfully, a few FLOs like move-to-goal, tacklemonster, grab coin, search-question and avoid-pit seem to be enough. The important observation in the GOMS analysis was that a human expert would make a selection between these FLOs depending on some very local conditions. If nothing was in the way, the human agent would continue to move to right towards the goal, however if there was some danger from a monster nearby, tackling the monster would gain preference. The authors identified these rules and implemented them on Soar such that the agent gets maximum points over the complete course. For this project we defined attributes like isthreat for monsters and isreachable for coins and question blocks which if true signified immediate action needed to be taken.

A key point to note here is, once a FLO is selected the agent/expert has an immediate goal to attain. When tackle-monster is selected, the immediate goal is to either kill the monster, or to avoid it. Learning in Flat

RL was hard because the goal state was too far away in the future and it took many trials for the reward earned at the goal state to propagate back.

Each episode in Mario takes around 500 primitive steps. When an agent is playing, it executes many sub-optimal steps, some of which may cause the agent to lose points, however q values of all state action pairs are updated indiscriminately. For domains where the total number of steps taken in completing a task is small, the q values do converge after running a large number of trials as the effects even out. For domains like this, q value may take more than millions of steps before they converge to some point. However, now that the agent has an immediate goal, the reward earned by succeeding or failing at the goal can be propagated to the actions taken by the agent while particular FLOs was selected. This places a bound the state action pairs that need to be updated whenever a reward is earned. This seems correct intuitively too, if the agent is performing tackle-monster and fails and earns a negative reward, only the actions that participated in tackle-monster need to be updated, rest all actions taken were probably good. Learning is highly improved because the goals are well defined.

Figure 1 shows the hierarchy of operators used in this task. FLOs on level 2 are proposed for each instance of the object they pertain to. For example, if there are two coins in the visual scene that are within agent's reach, two grab-coin operators will be proposed, one for each coin.

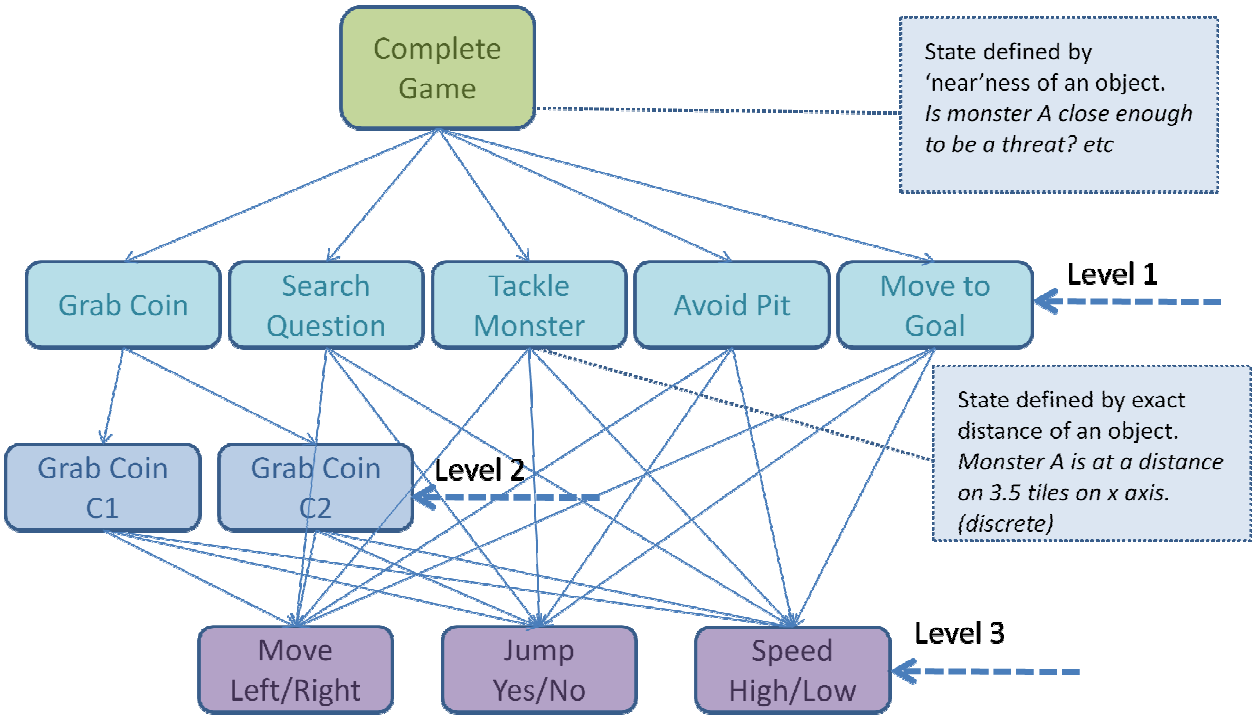


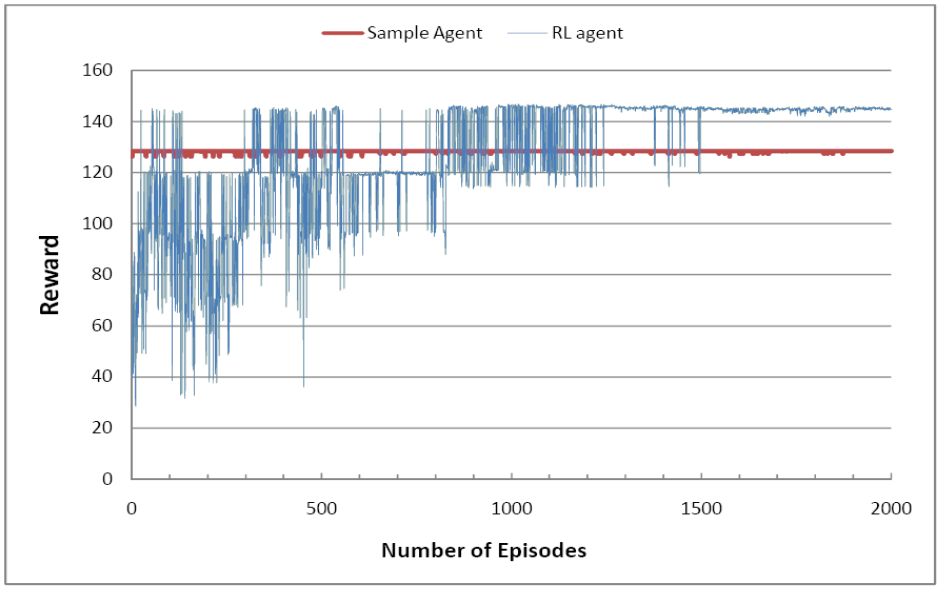
Figure 2 : Operator hierarchy in Infinite Mario task

The problem now reduces to learning the preference of operators in different states. We implemented two different learning scenarios, details of which are discussed in following sections.

8.2.1 Agent with hard coded FLO preferences

In the first set of experiments with hierarchical scheme, the preference order of FLOs was decided using heuristics specific to Mario game. move-to-goal is proposed when the agent is initialized and is never retracted. It is given the least preference. If a monster is close enough to be threat for Mario, tackle-monster is assigned the highest preference, higher than any other operator proposed. In general any operator that prevents game from ending before Mario reached finish line is given highest preference. If there is no threat to Mario, search-block had higher preference than grab-coin.

The task was to learn a set of primitive actions that constituted FLOs such that the reward earned was maximized. The results are summarized in Graph 2.



Graph 2: The average performance of RL agent with hard coded preference order in FLOs and sample agent over 5 experiments of 2000 episodes each in Infinite Mario Level Type 0, seed 121. The x axis denotes the number of training episodes and the y axis denotes the reward earned in an episode.

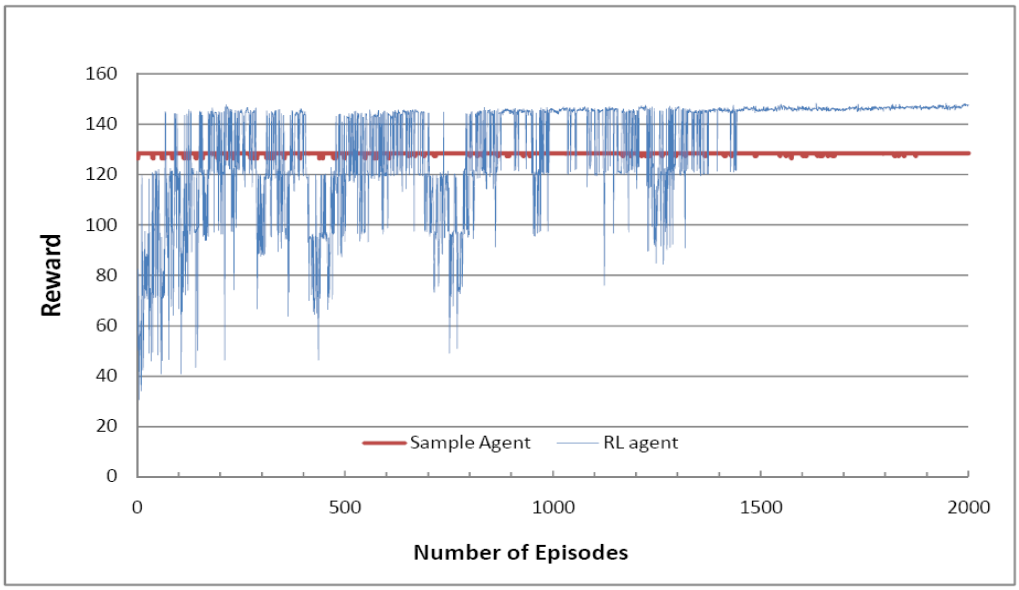
The agent shows significant improvement in performing the task after training. After training on approximately 1500 episodes, the agent always finishes the episode and on an average earns more reward than the sample agent. This is a considerable improvement over a flat RL agent (graph 1), that fails to converge to a policy even after training on 2000 runs through the episode.

8.2.2 Agent with learned FLO preferences

The next obvious question to ask is, if an agent can learn the preference order of FLOs autonomously. This question is close to Hierarchical Reinforcement Learning, where a larger task is divided into smaller subtasks. A subtask is accomplished through a executing a temporally extended action, analogous to FLOs

previously described. A temporally extended action lasts multiple time steps and is essentially composed of a set of primitive actions.

The agent is required to learn policies at multiple levels. Learning at levels 2 and 3 (form figure 1) has been presented earlier in Section 8.2.1. Additionally the agent also learns the preferences of FLOs in different states. The agent structure mostly remains the same, the reward obtained be executing the subgoal is used to update the q values of the primitive actions that accomplished the subgoal and a discounted reward is used to update the q value of the FLO applied. The results are summarized in Graph 3.



Graph 3: The average performance of HRL agent and sample agent over 5 experiments of 2000 episodes each in Infinite Mario Level Type 0, seed 121. The x axis denotes the number of training episodes and the y axis denotes the reward earned in an episode.

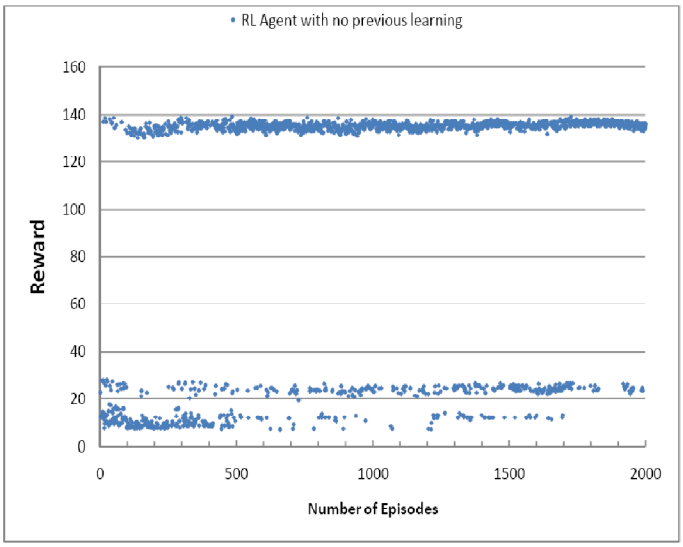
The performance of HRL agent is similar to the agent with background knowledge about FLOs. By the time the agent learns the primitive actions that constitute the FLOs, it also manages to learn their preference order. It is because the reward received is immediate; if move-to-goal operator is selected instead of tackle-monster when monster is a threat, the agent dies earning a big negative reward. This feature of the game helps the agent to quickly converge to optimal q-values of FLOs.

8.3 Transfer of acquired knowledge across tasks

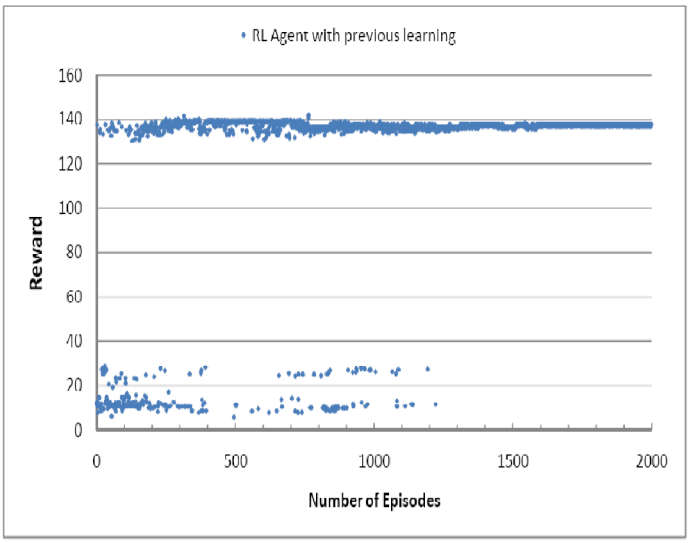
In the approach described in Section 8.2, the agent learns policies that are specific to subtasks rather than policies that fit the training episodes, unlike most RL approaches. These subtask-policies can be applied to other tasks that have similar sub-components. This property of learning in this representation was tested by training the agent in level type 0 and comparing its performance at level type 1 with a naive agent that has never experienced any learning.

The results are presented in Graphs 4 and 5. Graph 4 summarizes performance of a naïve agent on Level Type 1. The agent does not converge to an optimal policy in 2000 runs through the episode. In contrast to

this, from Graph 5, an agent that has been previously trained at Level type 0 manages to converge to an optimal policy in Level Type 1 in about 1200 runs.



Graph 4: The average performance of HRL agent with no background learning over 5 experiments of 2000 episodes each in Infinite Mario Level Type 1, seed 0. The x axis denotes the number of training episodes and the y axis denotes the reward earned in an episode.



Graph 5: The average performance of HRL agent with background learning over 5 experiments of 2000 episodes each in Infinite Mario Level Type 1, seed 0. The x axis denotes the number of training episodes and the y axis denotes the reward earned in an episode.

This experiment partially explores inductive transfer of learned knowledge. The knowledge acquired while learning to play Level Type 0, helped the agent in learning to play Level Type 1. Level Type 0 and 1 are almost similar; they have same type of monsters and similar reward structure. The agent could apply knowledge accrued across different tasks having similar structure. In future we wish to perform rigorous experiments on this aspect of learning.

9 Limitations of the current design

The current, object oriented learning design, performs well in situations where -
1) Only single object affects agent behavior. *Example:* only a coin, or a monster in close vicinity
2) Multiple objects affect behavior but their preference is intuitively clear. *Example a*: a coin and a monster in close vicinity, tackling the monster has a preference over collecting the coin. *Example b*: two coins are present, collecting the coin that is closer is preferred over a coin further away.

However, in situations where multiple objects have similar intuitive preference, the agent fails to learn a policy. The current design forces the agent to make a selection between objects and then deal with them in order of their preference. In the example scenario in Figure 4, both flying red koopa and green koopa have to be considered while taking the next step. We assume that individual policies for tackling red koopa and green koopa have converged through prior experience of game playing.

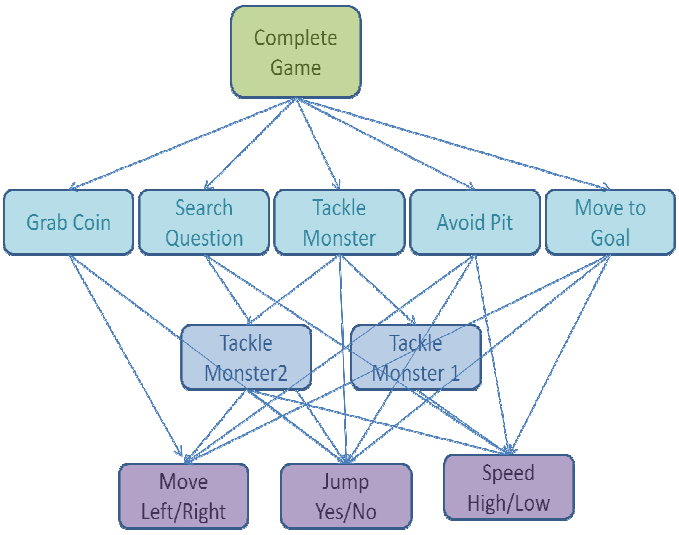


Figure 3: Operator hierarchy when two FLOs conflict

Figure 4: Example Scenario

Given the current design, the agent will make a selection between red and green koopa, and perform a series of actions to tackle the selected monster. If it selects red koopa, the converged policy dictates that the agent take a step towards right because in doing so it can avoid red koopa and move a step closer to the goal. However, this results in the agent colliding with green koopa moving to the left and the game ends with a high negative reward. If green koopa is selected, the agent should execute a right-jump step that eliminates green koopa. However, in the current setup the agent collides with red koopa in the air and earns a huge negative reward. Moving left intuitively seems a good move in the present case, but it is a sub-optimal move in both the policies.

10 Future Work

In the case presented in section 9, KLOs right and jump-right have conflicting effects with the two monsters. Moving right is good step to take while dealing with red koopa; it ends up in agent earning a negative reward. Similarly jump-right causes the agent to earn a high negative reward despite of it being an apparent fruitful way of dealing with green koopa.

When different objects propose primitive action operators with conflicting preferences, a good move would be to execute an action that is not "too bad" for any of the objects being considered. This action might be sub-optimal in all individual policies.

One of the solutions to this problem is incorporating mechanisms in the architecture that facilitate autonomous creation of more specialized rules from generic ones when a conflicting situation is perceived.

A different solution is to formulate this problem as one that encompasses other aspects of human cognition, like episodic memory and spatial reasoning as well as reinforcement learning.

Another important question is, what is a conflict between preferences? We as humans have a qualitative sense associated with moves in a game. Certain moves are 'good' while others are 'bad' in a setup. In Soar reinforcement learning agents, the quality of a move is related to its numerical preference. The move with highest numerical preference is the 'best'. However, how large should the difference between numerical preferences of operators be so as they can be safely assumed to be conflicting? We wish to investigate this issue further and look at how different heuristics and game structure can be employed to associate this quality with numbers.

11 Observations on Soar Architecture

The architecture has certain features that lead to a relatively complex design of agents for this task. Assume the agent is in state S_a and the only operator applicable is O_a and the agent begins the execution of operator O_a. Due to application of O_a the state changes to S_{ab} where O_a and O_b are applicable. As long as the proposal rule for O_a is same for both S_a and S_{ab}, the agent continues with the execution of O_a and does not make a selection between O_a and O_b.

A more specific example from this project:
move-to-goal is a FLO that is proposed as soon as agent is initialized, and does not retract till the agent is alive. Other FLOs like tackle-monster are proposed only if the agent is in some particular state; i.e. if monster is a threat in this case. The task is to learn the preference order of move-to-goal and tackle-monster if the monster is a threat. Now since move-to-goal was executing when the state changed and its proposal rule is same in the new state, the agent does not make a selection between move-to-goal and tackle-monster and continues with move-to-goal. We wrote different operator proposal rules for different states to get around this. It remains to be discussed if this behavior is desirable.

12 Conclusion

We investigated the performance of Soar-RL agents in a challenging and complicated task and demonstrated that symbolic, relational representation of state space allows for a significant reduction in the number of states; making learning tractable. Simple heuristics can used to easily extract features of the environment. A symbolic representation also allows for providing the agent with background information about the task which is taken for granted in humans. We also demonstrated that agents so designed can attain a high score on Level 1 of the game. Similar agents can be designed which can attain a high score on more difficult levels. It was further established that imposing hierarchies on actions and tasks further reduces the complexity of the task at hand and facilitates better learning in a reinforcement learning agent. Additionally generalized policies were learnt that could be scaled to new but similar task instances. We discovered potential areas on the cusp of reinforcement learning and cognitive architectures, further research on which may lead to a more comprehensive theory on learning in intelligent agents.

References

[1] Reinforcement Learning 2009 competition (http://2009.rl-competition.org/mario.php)

[2] Super Mario Bros (http://en.wikipedia.org/wiki/Super_Mario_Bros.)

[3] Laird, J. (2008). “Extending the Soar Cognitive Architecture”. *Artificial General Intelligence Conference*, Memphis, TN.

[4] Nason, S., and Laird, J. (2005): Soar-RL, integrating reinforcement learning with Soar”. *Cognitive Systems Research 6*

[5] Laird, J. and van Lent, M. (2001): “Human-level AI’s killler application: Interactive computer games.” *AI Magazine.*

[6] Spronck, P., Ponsen, M, Sprinkhuizen-Kuyper, I., and Postma E. O. (2006) “ Adaptive game AI with dynamic scripting andMachine Learning”. *Special Issue on Machine Learning in Games.*

[7] Ponsen, M. and Spronck, P. (2004): “Improving Adaptive game AI with evolutionary learning”. Artificial *Intelligence, Design and Education.*

[8] Ponsen, M., Spronck, P., and Tuyls, K.(2006). Towards Relational Hierarchical Reinforcement Learning in Computer Games. *Proceedings of the 18th Benelux Conference on Artificial Intelligence*, Belgium.

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

[10] Barto A. G. and Mahadevan S. (2003): “Recent Advances in hierarchical reinforcement learning”. *Discrete Event Dynamic Systems: Theory and Application.*

[11] John, B.E., Vera, A. H., and Newell, A. (1990): “Towards Real-time GOMS. *Carnegie Mellon University, School of Computer Science Technical Report. CMU-SCS-90-95*

[12] Diuk, Cohen and Littman (2008). An object-oriented representation for efficient reinforcement learning. *Proceedings of 25th International Conference on Machine Learning*, Finland