Reinforcement Learning in Infinite Mario

Shiwali Mohan

University of Michigan, Ann Arbor

Abstract

Many modern computer games have continuous, enormous state-action spaces and are characterized by complex relationships between their components. Without applying abstractions, reinforcement learning learning in a such games becomes infeasible. In this work, we investigate using different forms of abstraction to make learning in a computer game - Infinite Mario, tractable in limited scenarios. We show that imposing hierarchies on the actions and the tasks constricts the state space as a result of which, learning is faster. We further demonstrate that a relational representation allows for the use of structural information like the presence of objects and relationships between them in the description of value functions. Such representations facilitate the inclusion of background knowledge that qualitatively describes a state and can be used to design agents that demonstrate learning behavior in domains with large state and actions spaces like such as computer games. This formulation introduces partial observability in the domain which makes learning hard in specific situations.

1 Introduction

The current generation of computer and video games offers interesting test-beds for artificial intelligence and machine learning research. They are real-time and very dynamic, encouraging fast and intelligent decisions. Traditionally (Nareyek, 2004), artificial intelligence has been used to control the non-player characters in various games to make the game more interesting to a human player, which is challenging problem on its own right. However, lately, much research in being invested in developing agents that when placed in a new game with limited or no background knowledge, learn various nuances of the game and in the process become more efficient at playing the game. Modern computer games share some characteristics with the real world environment such as having have large state and action spaces. Studying such agents may offer insights into creating agents that can learn in complex environments.

Various real-time strategy games (RTS) such as Battle of Survival (Ponsen et al., 2006), Neverwinter Nights (Spronck et al., 2006) etc. particularly have been studied by the machine learning and AI community. Learning to play these games involve a search in the strategy space to devise a good strategy based on some heuristics. Another genre of popular video games - action games, have been popular with AI research community and have been studied in great detail using rule based systems (Laird and VanLent, 2001) and static scripting (Spronck et al., 2006). In this work, we are interested in looking at AI agents that learn online, to play a variant of a popular action game Super Mario, using a common reinforcement learning algorithm. These games differ from RTS games in that an agent learning to play an action game will need to learn various skills involving sensory-motor primitives concurrent to learning a strategy to navigate through the game environment.

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

The rest of the paper is organized as follows. We discuss some motivations for the current work in Section 2. Section 3 reviews some related work. In Section 4 we give a brief background in reinforcement learning and in Section 5, we talk about specifications of the domain - Infinite Mario used in experiments and discuss the challenges it presents for a learning agent. In Section 7 we talk about reinforcement learning agents and their implementation details and we end with concluding remarks in Section 8 and with a brief discussion of future work in Section 9. Appendix A gives details about the domain.

2 Motivation

In reinforcement learning problems, an agent interacts with its environment and iteratively learns a policy. Policies are often represented in a tabular format, where each cell includes a state or state-action value representing, respectively, the desirability of being in a state or the desirability of choosing an action in a state. Several simple but powerful algorithms like SARSA, Q-Learning have been developed to iteratively and quantitavely compute the desirability of choosing an action in a state. These algorithms are robust and have been successfully employed in variety of deterministic and stochastic domains. Most of the work has focused on the algorithmic aspect, i.e. various ways of computing value functions and policies. Usually the representational aspects are limited to the use of attribute-value or propositional languages to describe states, actions etc. These learning algorithms and propositional representations have proved to be feasible in toy domains such as Grid World that are characterized by limited state and action spaces. In contrast, in more complex domains the number of states grows exponentially, resulting in an intractable learning problem. Modern computer games are typical examples of such complex domains. They offer a unique set of artificial intelligence challenges, such as dealing with huge state and action spaces and real-time decision making in stochastic and partially observable worlds.

Recent attempts to curb the curse of dimensionality have turned to exploiting temporal abstractions where instead of taking a decision at each step, execution of temporally-extended activities are invoked. The temporally-extended activities or *macro-actions* follow their own policies till termination. The idea of temporal abstraction is not new in artificial intelligence research. Researchers have addressed the need for large-scale planning and problem solving by introducing various forms of abstraction into problem solving and planning systems. Some examples include: applying the concept of searching for macro-operators in large problems (Korf, 1985); abstract actions in robot planning (Fikes et al., 1972); and ABSTRIPS, a system that automatically constructs abstract hierarchies for planning (Sacerdott, 1973). What these techniques have in common is the intuition that a complex problem can be solved by decomposing it into a collection of smaller problems. Such decomposition helps in speeding up learning in three different ways. One, it allows us to limit the choices available to the agent, two, it allows us to specify local goals for certain parts of the policy and three, it allows the sharing of common sub-goals so when the agent learns one task, the knowledge can be transferred to a new task that is composed of similar sub-goals.

Representations based on first-order logic are used in much of the literature on artificial intelligence (Russell and Norvig, 2003) as well as intelligent agents (Woolridge and Wooldridge, 2001). The use of first-order logic enables powerful abstractions to be used in order to solve and reason over complex problem domains. Much of the planning literature assumes a first-order logic in which domains can be described in terms of objects and relations. The use of relational representations in reinforcement learning contexts offers many potential advantages. An important advantage is generalization across objects and possibly transfer of learned knowledge to different tasks in similar environments. For example, a policy learned for a grid world domain with 25 squares will often quite naturally generalize to a domain containing more squares if the policy can be representated based on relative positions of objects and square instead of absolute positions. Furthermore, the use of relational representations enables the use of background (prior) knowledge in a natural way, facts like *if there is wall on the left of a square, then it is unreachable from a square in its left* can be easily encoded and transferred to similar situations

Through this work, we explore how these two techniques - hierarchical reinforcement learning and relational representations, shown to handle the complexity in various domains in prior research, can be combined to develop agents that demonstrate learning behavior while operating in a complex, game domain - Infinite Mario. The game is composed of several objects with different properties and certain interactions with these objects lead to positive reward. We are interested in investigating how the complex task of learning to play the games can be divided into smaller sub-goals of learning various interactions with different objects, how

relational representations can be leveraged to build action hierarchies, and if good policies can be learnt by imposing these restrictions in a reinforcement learning framework.

3 Related Work

Reinforcement learning has been widely explored for its applicability in computer games domain. Ponsen et al. (2006) looked at hierarchical reinforcement learning applied to a learning task in a real time-strategy computer game. They employ a deictic state representation that reduces the complexity as compared to a propositional representation and allows the adaptive agent to learn a generalized policy and it is capable of transferring knowledge to unseen task instances. They show that hierarchical reinforcement learning significantly outperforms flat learning. Marthi et al. (2005) applied hierarchical RL to scale to a complex environment. They learned navigational policies for agents in a limited real-time strategy computer game domain. Their action space consisted of partial programs, essentially high-level pre-programmed behaviors with a number of choice points that were learned using Q-learning. Driessens (2001) combined RL with regression algorithms to generalize in the policy space. He evaluated his relational RL approach in two RTS and RPG (role playing games) computer games. Our work significantly differs from these works in that we we are looking at object oriented, relational representations and their integration in a hierarchical reinforcement learning framework and if such a formulation helps an agent learn reactive skills and navigational strategy in a different genre - action games.

In the domain of action games, in recent work, Diuk et al. (2008) introduce Object-Oriented Markov Decision Processes, a representation that is based on objects in the environment and their relationships with each other. They claim that such representations are a natural way of describing many real-life domains and show that such description of the world enables and agent to learn efficient *models* of the action game, *Pitfall*. Wintermute (2010) has explored agents that learn to play various Atari action games using predictive features generated by *spatial imagery models* in reinforcement learning and report improvements over RL agents using typical algorithms and state features. However, these works involve use of models either learned or provided as background knowledge, whereas our work is a model-free approach.

Infinite Mario has recently become of popular domain for AI research with the advent Mario AI competition (RL-Competition, 2009a). Many agents have been written that are efficient at playing the game. Most of them are based on A* search for path finding and use detailed models of physics and minimal learning. Our work is significantly different from previous works on the game in that we are looking at learning agents that start with little or no background knowledge about the physics or the terrain of the game.

4 Background

4.1 Reinforcement Learning

Reinforcement Learning (RL) is a computational approach to automating goal-directed learning and decision making. It encompasses a broad range of methods for determining optimal ways for behaving in complex, uncertain and stochastic environments. Most current RL research is based on the theoretical framework of Markov Decision Processes (MDPs) (Sutton and Barto, 1998). MDPs are a standard, very general formalism for studying stochastic, sequential decision problems.

A MDP is defined by its state and action sets and by the dynamics of the environment. On each time step t, the agent observes the state of its environment, s_t, contained in a finite discrete set S, and decides on an action, a_t, from a finite action set A. One time step later, the agent receives a reward r_{t+1} and the environment transitions to a next state, s_{t+1}. The framework is assumed to have *Markov Property*, that it, that environments response at $t+1$ depends only on the state and action representations at t and not on any past values of states and actions. Thus, given any state and action , s and a, at time t the *transition* probability to possible next state, s', at time $t+1$ is:

$$P_{ss'}{}^{a} = Pr\{s_{t+1}|s_t = s, a_t = a\}$$

Similarly, given any state and action pair, s and a, at time t along with the next state, s', at time $t+1$ the

expected value of the reward is given by:

$$R_{ss'}{}^{a} = E\{r_{t+1}|s_t = s, a_t = a, s_(t+1) = s'\}$$

A RL problem is finding a way of behaving in an environment such that the reward accumulated is maximized. The way of behaving, or policy, is defined as probability distribution for picking actions in each state: $\pi : S \times A \rightarrow [0, 1]$. The goal of the agent is to find a policy that maximizes the total reward received over time. For any policy π and any state $s \epsilon S$, the value of taking action a in state s under policy π, denoted $Q^{\pi}(s, a)$, is the expected discounted future reward starting in s, taking a, and henceforth following π:

$$Q^{\pi}(s, a) = E_{\pi}\{r_{t+1} + \gamma r_{t+2} + ...|s_t = s, a_t = a\}$$

The *optimal* action-value function is:

$$Q^{*}(s, a) = \max_{\pi} Q^{\pi}(s, a)$$

In an MDP, there exists a unique optimal value function, $Q^{*}(s, a)$, and at least one optimal policy, π^{*}, corresponding to this value function:

$$\pi^{*}(s, a) > 0 \iff a \epsilon \arg\max_{a} Q^{*}(s, a')$$

Many popular reinforcement learning algorithms aim to compute Q (and thus implicitly π) based on the observed interaction between the agent and the environment. One of the most widely-used is probably SARSA introduced by(Rummery and Niranjan, 1994), which allows learning Q directly from interaction with the environment.

SARSA is an is an on-policy temporal difference learning algorithm for Markov decision processes. It is based on the following update equation:

$$Q(s, a) \leftarrow (1 - \alpha)Q(s, a) + \alpha[r + \gamma Q(s', a')]$$

where $0 \leq \alpha \leq 1$ is a learning rate parameter and $0 \leq \gamma < 1$ is the discount factor for future rewards. The update equation states that for a given state-action pair $(s, a) \epsilon S \times A$ the new state-action value is obtained by adding a small (depending on γ) correction to the old value. The correction is the difference between the immediate reward r increased by the discounted future state-action value $\gamma Q(s', a')$ and the old state-action value $\gamma Q(s, a)$. SARSA (s, a, r, s', a') is an on-policy learning algorithm in the sense that it estimates the value of the same policy that it is using for control. QLearning (Watkins and Dayan, 1992) constitutes an off-policy alternative to SARSA and replaces the term $\gamma Q(s', a')$ by $\gamma max_{a' \epsilon A(s')} Q(s', a')$ in the above equation. This allows for separating the policy being evaluated from the policy used for control.

4.2 Hierarchical Reinforcement Learning

The main reason for introducing hierarchical architectures in RL is to bridge the gap between theoretical considerations to machine intelligence and practical application to real-world problems. Hierarchical RL is especially interesting for large-scale problems where the performance of flat RL is too poor with regard to learning speed and computational effort.

Semi-Markov decision processes (SMDPs) serve as theoretical basis for many hierarchical RL approaches developed during the last decade. In these hierarchical approaches temporally-extended and abstract actions need to be modeled. SMDPs may be considered as generalization of MDPs. Sutton et al. (1999) define a SMDP as a continuous-time decision problem that is treated as a discrete-time system, where the system makes discrete jumps from one time at which it has to make a decision to the next.

Formally an SMDP is a tuple $< S, B, T, R >$, where S is a set of states, B is a set of temporally-abstract actions, $T : S \times B \times S \times \mathbb{R} \rightarrow [0, 1]$ is a transition function (including duration of execution), and $R : S \times B \times \mathbb{R} \rightarrow [0, 1]$ is a reward function:

$$T(s', k|s, B) = P(B_t \; terminates \; in \; s' \; at \; time \; t + k|s_t = s, B_t = B)$$

$$R(r|s, B) = P(r_t = r|s_t = s, B_t = B)$$

T and R both obey Markov Property.

A policy is a mapping $\pi : S \rightarrow B$ from states to behaviors. Executing a behavior results in a sequence on primitive actions being performed. The value of the behavior is equal to the value of that sequence. Thus if behavior B is initiated in state s_t and terminates sometime later in state s_{t+k} then the SMDP reward value r is equal to the accumulation of the one-step rewards received while executing B: $r = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + ... + \gamma^{k-1} r_{t+k-1}$ which gives identical state-value function as in MDPs. Since the value measure V^π for a behavior based policy π is identical to the value measure V^π for a primitive policy we know that π^* yields the optimal primitive policy over the limited set of policies that a hierarchy allows. Learning internal policies of behaviors can be expressed along the same lines.

Various techniques for hierarchically decomposing a task have been proposed in the literature that are based on SMDP formulation of the problem. The options framework (Precup et al., 1998) relies on reducing the entire problem as a single SMDP. Another technique that relies on the theory of SMDPs is MAXQ (Dietterich, 2000), tha creates a strict hierarchy of SMDPs whose solutions can be simultaneously learnt. We employ hierarchical technique which is very similar to MAXQ, where a problem in hierarchically decomposed into several sub-goals whose solutions are simultaneously learnt.

5 Domain description - Infinite Mario

Infinite Mario is a reinforcement learning domain developed for Reinforcement Learning Competition 2009. It is 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. Figure 1 shows a typical visual scene in Infinite Mario.

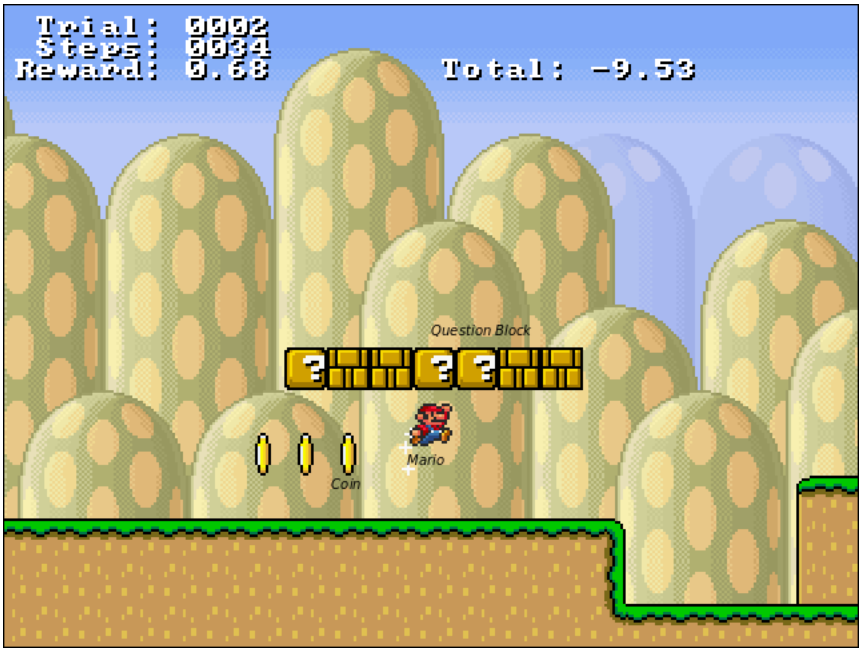

Figure 1: Typical visual scene from Infinite Mario

Infinite Mario has been implemented on RL-Glue developed by Tanner and White (2009); a standard interface that allows connecting reinforcement learning agents, environments, and experiment programs together. The agent's perception of the visual scene is comprised of various numeric and character arrays. It can perform actions in the environment by setting the values of action integer array.

- *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 matrix generated by the environment corresponding to the given scene. Each tile (element in the input matrix) can have one of the 13 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. For every monster visible, the agent is provided with the type of the monster, its current location, and its speed in both horizontal and vertical direction. Mario is one of the monsters in the observation.

- *Actions*
  The actions available to the agent are same as those available to a human player through game-pad 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 the action array.

- *Reward*
  The agent earns a huge positive reward when it reaches the finish line. Every step the agent takes in the environment earns it a small negative reward. The agent gets some negative reward if Mario 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.

- *Sample Agent*
  The Infinite Mario package also contains a sample learning agent. The sample agent does not perform reinforcement learning, however, it learns through memorization in that it stores the sequence of actions (decided using some heuristics) it takes as it moves through the episode. If the game terminates because Mario dies by running into a monster, on its next trial through the episode the agent repeats all but the last 7 stored actions, after which it switches to taking random actions. If the game terminates because Mario finishes the episode by reaching the finish line, the agent just repeats the stored sequence of actions. As we will see in the following sections, such design despite its simplicity, is difficult to beat in performance.

- *Game Levels and Difficulty*
  The domain is capable of generating several instances of the game with great variability in difficulty, i.e. as the agent moves to a higher difficulty, making good decisions gets harder. This can be due to close interaction with many objects at the same time or having more constraints on how the agent can move. The lowest difficulty, which is the easiest is difficulty 0. The game also has different levels which correspond to different kinds of terrains. For game instances of similar difficulty, making good decisions remains equally hard across levels.

More details about the domain can be viewed in Appendix A.

Following are some interesting properties of this domain:

- *Continuous, High Dimensional State Space*: The domain, inherently, is continuous. Although, the visual scene is divided into discrete tiles, the motion is continuous described by real valued speed and absolute positions. True continuous state representations would lead to infinitely large state space. Even on discretization, the state space is enormous. Without applying suitable generalization to different states, learning in this domain is a computationally expensive problem because of the high dimensionality of the state space. An example episode that is 300 tiles long has around 5000 tiles, which can be of 13 different types. It also will have many monsters that can be on any tile in the episode at a given time. Without abstracting this information to extract useful features, learning in such domains is extremely difficult. The discretization may lead to loss of important information preventing the agent from taking good decisions.

- *Dynamic*: The environment is highly dynamic, and there is a high degree of relative motion between objects. Although the input from the environment consists of speed, and exact locations of the monsters, predicting the next position of objects is hard because different objects have different behavior which can be learned only through experience. High degree of motion in the domain necessitates representations that are not dependent on the absolute positions of the objects for efficient learning.

- *Deterministic*: The motion of the objects is completely deterministic, and theoretically, it is possible to build perfect models of motion. However, since the domain is continuous, building correct models based only on experience is challenging.
- *Partially observable*: An episode can be of arbitrary length and structure and at a given instant an agent has access to information about only a part of the episode - the visual scene. This and the fact that the agent is free to move in both right and left directions introduces an element of partial observability in the domain. Also, some stationary objects might have different behavior in different instances, for example a 'pipe' may or may not contain a monster which adds to partial observability problem.
- *Relational, Object-Oriented*: Since the domain consists of various objects that affect each other, the domain can be described in a relational, object oriented language.
- *Generalization*: The game consists of many levels with increasing difficulty, therefore representations and learning that facilitate transfer of knowledge across levels are necessary. The agent should able to learn fast and it should be able to learn general policies from episodes it plays that can be applied to unseen episodes with similar elements.
- *Incremental Learning*: The domain contains many objects that may affect the agent's decision in various ways. It is hard to predict ahead of time how many or what objects will need to be considered while taking an action. Hence, the design of the agent should able to augment the state-action space incrementally, as the game progresses to provide for the new combinations of objects the agent might be sensing.

6 Research Question

We are interested in exploring relational, object-oriented representations and their effects on reinforcement learning in Infinite Mario. More specifically,

1. Does desciribing the state using relational representations
2. Can the task of playing the game be divided into a hierarchy of functional sub-goals related to different objects in the game leading to a good score?
    - Can policies for achieving these functional goals be learned by using a reinforcement learning algorithm - SARSA? Learning policies for achieving object related, functional goals are akin to learning various skills like killing an enemy, climbing obstacles in action games.
    - At different times in the game, the agent can encounter many objects at once and will be forced to make a choice between different sub-goals. Can this selection knowledge be acquired from experience in the game using SARSA? These selection policies can be thought of as simple, strategies that are used by players in an action game.
3. We are also interested in investigating if a relational language allows for easy inclusion of domain specific facts such as spatial reasoning knowledge and internal rewards and how these facts affect agents' learning behavior.

We also present a very simple, Flat RL agent with propositional state representation that is based on direct observations from the environment and is similar to representations used in grid-world like domains.

7 Reinforcement Learning Agents

The agents described in this section have been instantiated in the Soar cognitive architecture. Soar supports two different reinforcement learning algorithms - Q-learning and SARSA. All the results presented are averaged over 10 trials. We have used SARSA algorithm as introduced by Rummery and Niranjan (1994) to update value function of the state-action pairs, with learning rate α of 0.3 and discount rate γ of 0.9. The

agent uses an epsilon-greedy policy with ϵ of 0.1 and reduction rate of 0.9999.
We enumerate the agents that we created and the game levels they were tested on, here and summarize the results. The following, corresponding sections describe the agents in detail.

Experiment 1 Agent with tile-based propositional representation, similar to those commonly used in grid worlds.

- Was tested on difficulty 0. Does not converge to a policy in 5000 runs.

Experiment 2 Agent with tile-based propositional representation, similar to those commonly used in grid worlds with linear value function approximation.

- Was tested on difficulty 0. Does not converge to a policy.

Experiment 3 Agent with object oriented representation, flat learning

- Was tested on difficulty 0. Does not converge to a policy in 5000 runs

Experiment 4 Agent with object oriented representation, hierarchical, functional task decomposition, programmed selection knowledge

- Was tested on difficulty 0 and 1. It converges to a policy that earns high positive reward at difficulty 0, but fails to converge to a policy at higher difficulties.

Experiment 5 Agent with object oriented representation, hierarchical, functional task decomposition, acquired selection knowledge

- Was tested on difficulty 0 and 1. It converges to a policy that earns high positive reward (policy converges faster than in Experiment 4) at difficulty 0, but fails to converge to a policy at higher difficulties .

Experiment 6 Agent with object oriented representation and task decomposition, background spatial facts.1

- Was tested on difficulty 0 and 1. It converges to a policy that earns high positive reward at difficulty 0, earning a higher reward than the agent in 3. The policy does not converge at higher difficulties.

Experiment 7 Agent with object oriented representation and task decomposition and internal rewards.

- Was tested on level type 0, difficulty 0 and 1. It converges to a policy that earns high positive reward at difficulty 0, faster that agent in 3, but fails to converge to a policy at higher difficulties.

Experiment 8 Agent with object oriented representation and task decomposition with linear value function approximation.

- Was tested on difficulty 0 and 1. It converges to a policy that earns low positive reward at difficulty 1.

In our runs, the sample agent that learns by memorizing the sequence of actions taken in the previous episodes earned a total reward of 128.48 averaged over 10 runs, on converging.

7.1 Agent with Propositional Representation, Flat Learning

Using propositional representations is common in RL literature (Sutton and Barto, 1998) and has been used successfully in grid world type domains. Given that the input consists of the tile-by-tile information about the visual scene, designing an agent with propositional representation is easy. However, it is clear that the state space with such a representation is enormous and intractable. The visual scene has $16 * 22$, 352 tiles, each tile can be of 13 different kinds. This gives us a space of 13^{352} distinct, *possible* states; including types and positions of monsters that can appear on the visual scene will cause the space to explode even further. Although, most of these states will never occur in any instance of the game, there is no way of knowing ahead

of time which part of this state space is relevant. The enormity of the state space is an important issue for any practical system.

A common observation(John and Vera, 1992) is that while playing Mario, a human player rarely looks at regions on the screen that are far away from the Mario figure. The contents of visual scene around the Mario figure are more relevant to deciding what the next action should be. Relying on this reasoning, we designed agent that minited sensing to $5 * 3$ tile-space around the Mario figure. The tiles can be of 13 different types and even after this modification, the state space is still huge (close to 13^{15} distinct, possible states). We also included the position of monsters that were within range of the tile-space in the state representation and their location on the visual scene was discretized, for example, a monster at *4.8 tiles* on horizontal dimension was assumed to at a location of *5 tiles*.

Expected returns were calculated and maintained for every state-action pair, where the action is an element of action set composed of all the atomic, primitive actions available to the agent and states are described by the position and the type of tiles around Mario. If the discretized position of a monster corresponded with a tile, the tile was considered to be occupied by the monster regardless of its type. The state of the propositional agent can be described by the tuple: $< tile-type_0, tile-type_2, tile-type_2, ...tile-type_{14} >$.

Since, many different configurations and arrangement of tiles and monsters could occur and creating tables for all possible configurations is wasteful, we augmented the value function as new states were observed.

Results: Experiment 1

As can be observed in figure 3, even after 2000 trials through the same episode, the agent demonstrates any limited learning at best. We also ran the experiment up to $10,000$ trials; the agent did not show any additional learning. The sample agent, despite of its relatively simple design converges to good policy making it to the finish line every time towards the later stages of the trial. The Soar agent learned q-values for 3,400 state action pairs.

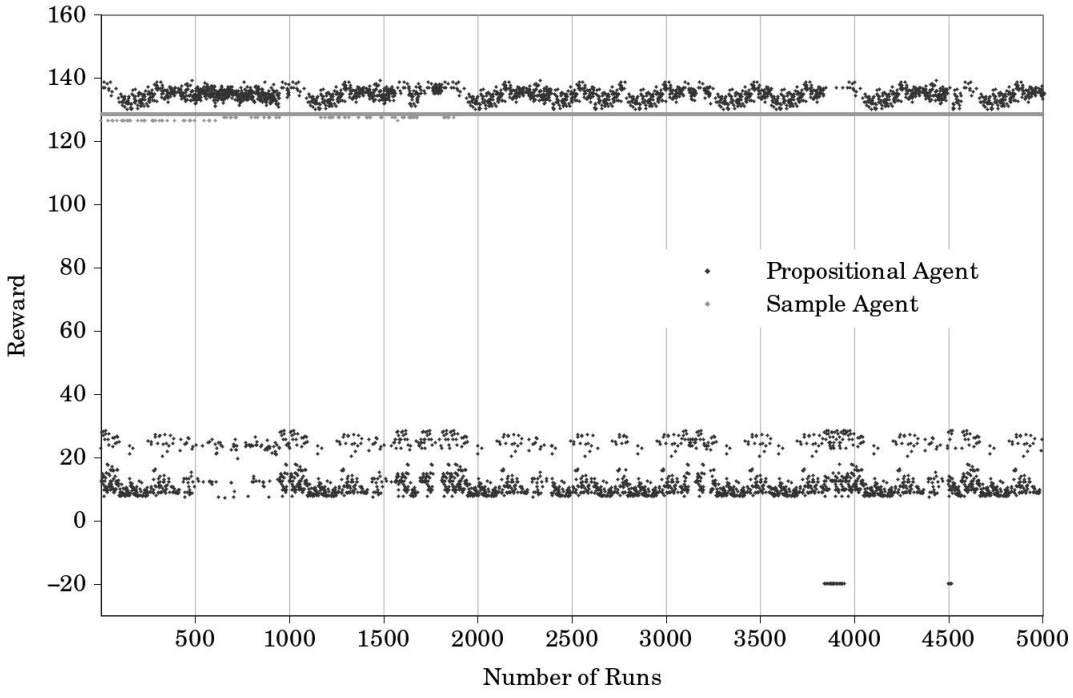

Figure 2: Performance of Propositional Soar-RL agent and Sample Agent in level type 0, difficulty 0

7.1.1 Agent with Propositional Representation, Linear Value Function Approximation

Despite the fact that the agent described previously looked at only a small portion of the visual space, the state space still is huge and hence, the agent learns slowly (or not at all). An interesting observation from the game is that various features of the state space repeat multiple times in a single game instance. For example, the agent may observe *the exists a monster of type goomba at tile (2,1)*, multiple times in a single game in different states. The presence of a particular monster at a particular tile affects agent's behavior in a certain way. This is true for other objects such as coins, blocks etc. in the domain as well. The *goodness* of an action in a state is dependent on the occupants of different tiles in the state space. A *move-right* action will always lead to a negative reward if the occupant of the next tile is a monster, regardless of other tiles in the space.

This fact can be leveraged to reduce the enormous state space problem to a certain extent. Value function approximations using coarse coding have been widely used in high dimensional state-space domains to approximate value functions for new states based on previously observed features. We used a particular tile, its occupant object and its attributes (such as its relative speed) as a feature. Each state was expressed as a coarse code based on applicable features. A table of q-values for each feature and action was maintained. Reward update received in a state for applying an action was divided indiscriminately between all the features that were applicable in the state. The value of of a state-action pair was estimated by averaging the q-values of the applicable features and the particular action. Such a formulation leads to a state space of $13 * 15$ states.

Results: Experiment 2
Figure 3 shows the result of a propositional, SARSA agent with value function approximation based on coarse coding on level type 0, difficulty 0.

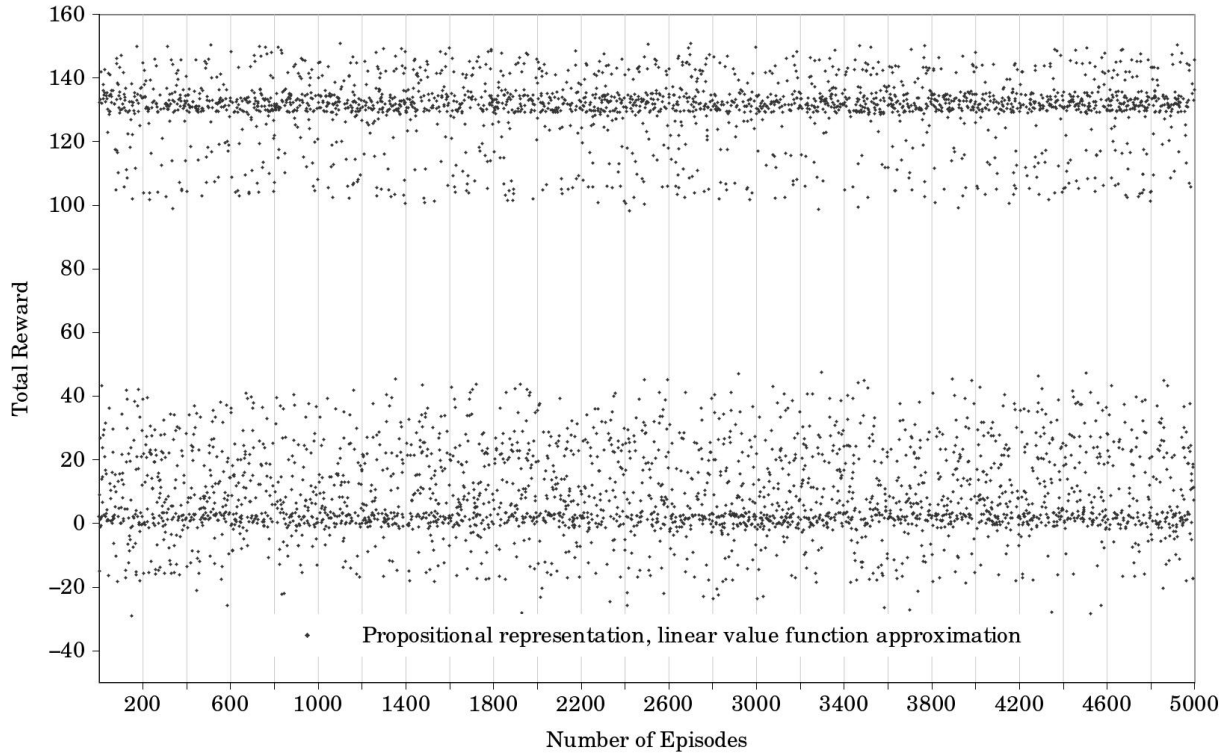

Figure 3: Performance of Propositional Soar-RL agent and Sample Agent in level type 0, difficulty 0

The agent shows little to no learning with such description of the state. We believe that this might be because of the environmental reward (+1 for killing a monster) can be attributed to a single object and hence, to a single feature in the current design. However the agent has no way of associating rewards to

different features of its current state, q-values associated with all features are updated even if only one of those features lead to the current reward. This leads to spurious learning.

7.2 Agent with Object-Oriented Task Decomposition and Hierarchical Learning

Relational representations have been used in several RL domains such as robotics as a means of introducing abstractions in a domain with high dimensional state space. We investigate if object oriented, relational representations fit naturally with hierarchical reinforcement learning framework and task decomposition similar to MAXQ . We then investigate whether such representations also facilitate inclusion of background knowledge, that may be hard to learn from experience alone.

Object-Oriented view of the World

The agent perceives the visual scene as a collection of tiles and monsters that are in view through its input link. We identified common, regular structures in the environment and extracted those structures to include them in the state description. For example, a visual scene could contain tiles that had *coins*, these tiles are important because the agent gains points on collecting coins; therefore we augmented the state with an attribute *coin*. Similarly, augmentations for *blocks*, *monsters*, *pit*, *pipes*, *platform* and other objects that were thought of to be important for the agent to navigate in the environment, were added to the state. Through this step, we moved from a tile-by-tile input data to a more abstract, object oriented view of the world as shown in Figure 4. A very similar representation was used by Diuk et al. (2008) to derive the model of another video-game, $Pitfall$.

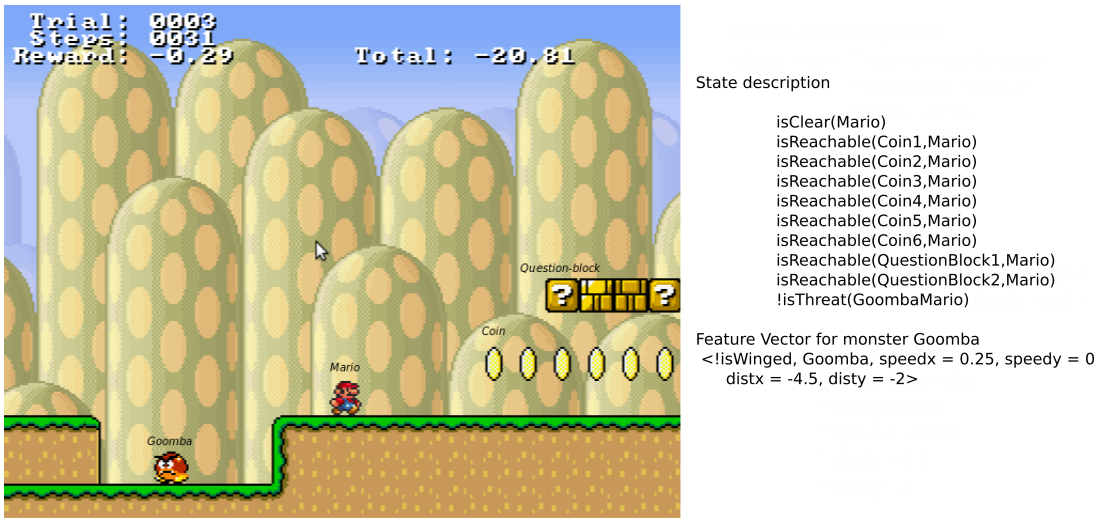

Figure 4: Object-Oriented State Representation

Each object also has a set of features associated with it that are extracted from the input data. For example, an instance of a *monster* can have features that describe its type, and if it had wings. Instances of objects were also attributed with features such as speed and distance that described their motion relative to Mario. An instance of a *monster* would have a feature vector - $< type, iswinged, relativespeed, relative - distance >$ associated with it.

First order relations such is *isreeachable*, *on(x)*, *isthreat* were also derived for instances of objects. These attributes were used to control learning, as will be explained in the next sections.

Results: Experiment 3

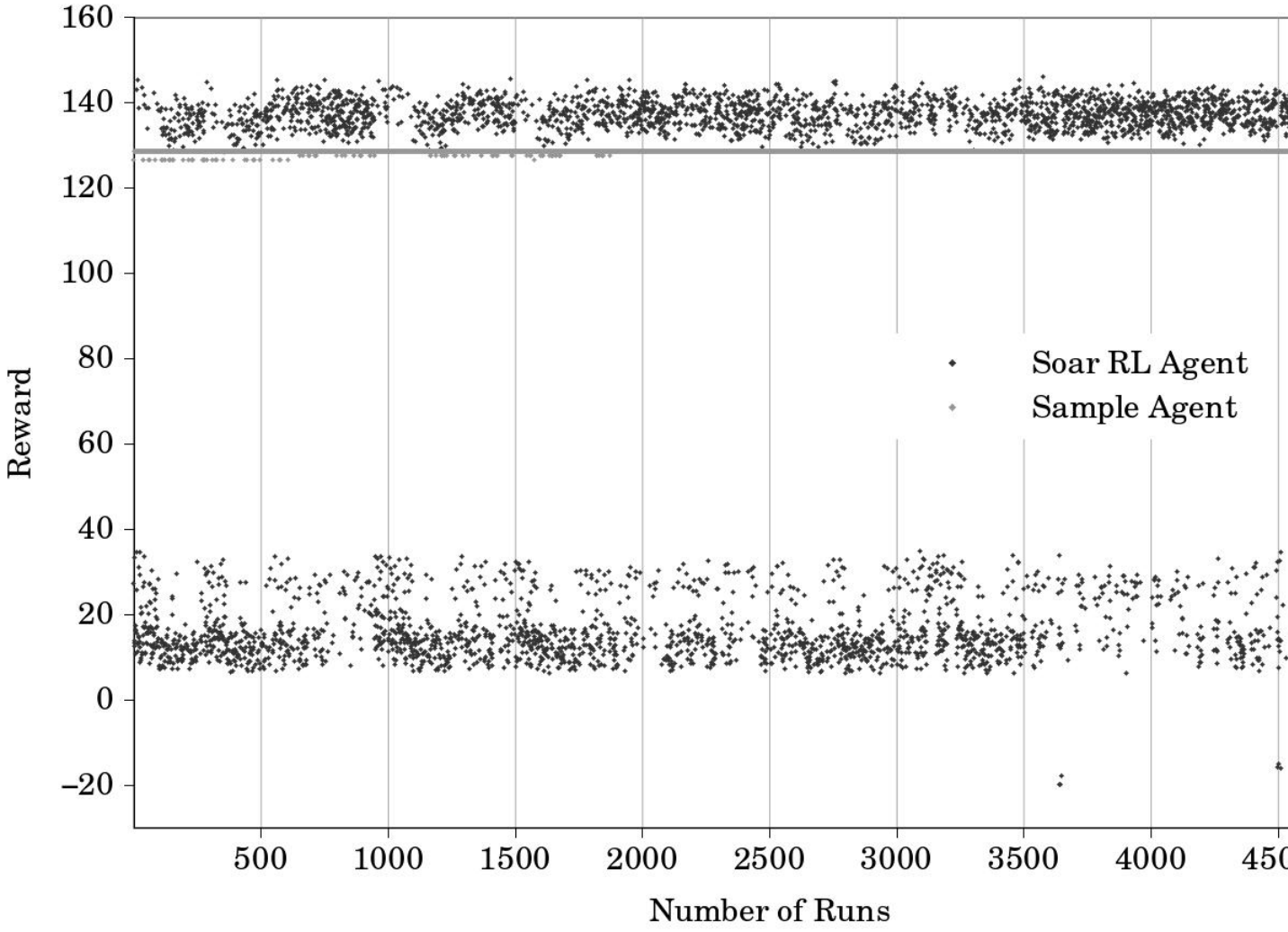

Figure 5: Performance of Soar-RL agent (internal rewards) and Sample Agent in level type 0, difficulty 0

Hierarchy of Operators

A GOMS (Goals, Operators, Methods and Selection rules) analysis of Nintendo's Super Mario Brothers conducted in 1992 by (John and Vera, 1992) demonstrated that the behavior of a Soar agent that used simple, hand coded heuristics formulated by taking knowledge explicit in the instruction booklet and by reasoning about the task, was predictive of a human expert playing the game. The analyses was carried out a two levels: the *functional-level* where operators are at the level of searching a block or killing a monster, and at the *keystroke-level*, where operators are at the level of primitive actions on the game's control panel. Such formulation of the experiment is interesting in that it decomposes the a long, complicated decision process into smaller and more constrained decision tasks that can be potentially be learned. Note that this analyses used hand-coded rules for selection between various FLOs. We show that this knowledge can be learned by expereince.

The task decomposition hierarchy we implemented for Infinite Mario is shown in Figure 6. We defined *Keystroke-level operators*, KLOs to be primitive, atomic-actions that the agent can perform like moving right or left, jumping, and shooting. As shown in Figure 6, a *functional-level operator* FLO (macro-action), is a collection of KLOs that when performed in succession, perform a specific task related to an object, such as killing a monster, or grabbing a coin. Every kind of object has a unique FLO and a goal associated with it, apart from which non-object specific FLOs like *move-right* can also be defined by specifying the applicability

conditions.

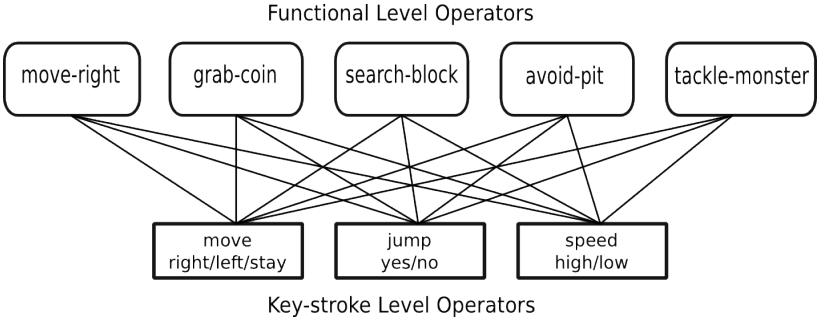

Figure 6: Operator Hierarchy

Applicability of an FLO depends on which relations hold. If a monster exists, i.e. it is a part of the sensory data and *isathreat* relation exists, then *tackle-monster* is applicable. The goals of a FLO can also be described using the relational properties of the domain; the goal of *tackle-monster* is to either avoid the monster which if successful, causes the *isathreat* relation to become false or to kill a monster, in which case the monster is not the part of the observations. Note that the state representation changes as the agent switch between two levels of hierarchy. At the higher FLO level, the state is represented using first order relations thus grouping many propositional states together. At a lower level, the state representation has much more detail and depends of which FLO was selected.

As the agent begins acting in the world, it encounters various objects. As its get close enough to an object such that it should start affecting agent's actions, an object-specific FLO is proposed. For example (shown in Figure 7), if a monster in *Mario's* vicinity, i.e. the attribute *isthreat* for that particular instance of the monster is true, operator *tackle-monster* is proposed. The agent then creates a substate with a goal to avoid that particular monster by moving away from it or by killing it. Once in the subgoal, the agent executes a sequence of KLOs to reach a desirable result, i.e avoid the monster. If it is successful, it moves out of that substate and resumes the default FLO, *move-right*. Throughout the progression in the game, the agent proceeds to the finish line and interacts with objects by creating substates with specific goals.

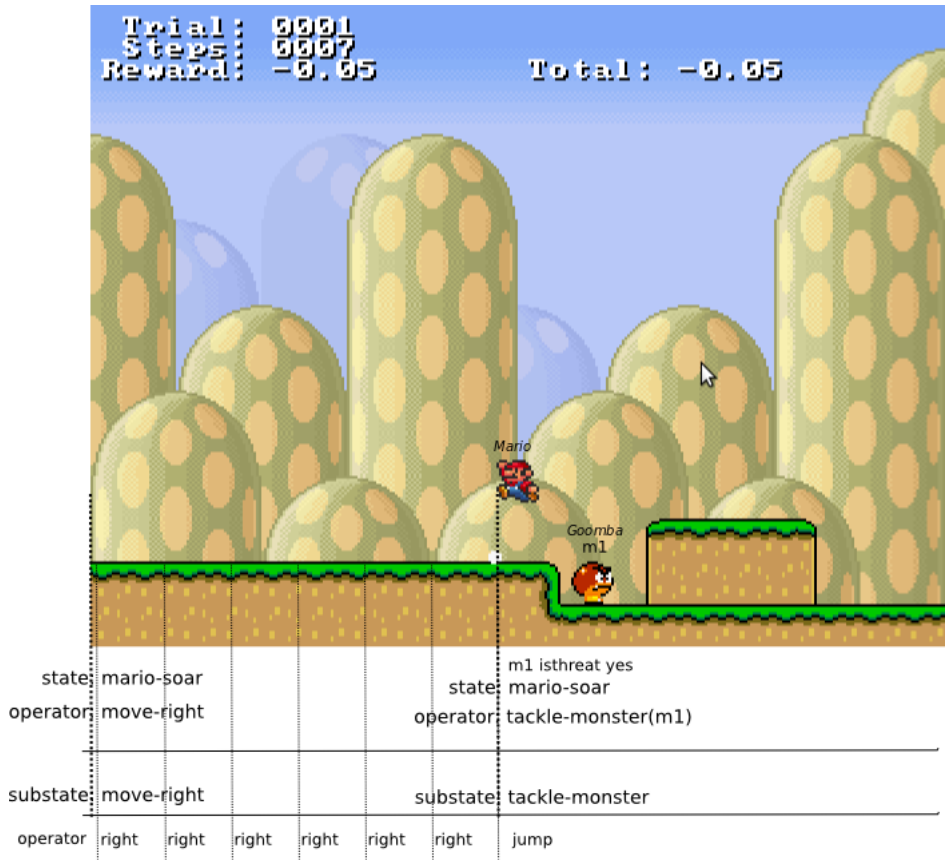

Figure 7: Substates and Objects

Such a formulation of the problem divides the game into several sub-goals of collecting a *coin*, killing a *monster* etc. As the agent learns of solve these sub-goals, it acquires reactive skills which are an important aspect of learning to play a computer game. FLOs proposed in a given state allow the agent to interact with corresponding objects in the environment. The agent selects between the FLOs based on which interaction is most important. For example if a *coin* and a *monster* are in close vicinity of Mario, the agent learns to choose to interact with the nearby *monster* first, because a failure to do so would result in the termination of the game. Learning to choose between FLOs is equivalent to devising a good strategy, which again is an important aspect of learning an action game.

Results: Experiment 4

In the first set of experiments with hierarchical scheme, the preference order of FLOs was decided using hand-coded heuristics specific to the game. FLO *move-to-goal* is proposed when the agent is initialized and is never retracted. It is given the lowest 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 Figure 8. The policy converges with an average total reward of 146.91 (averaged over last 100 runs).

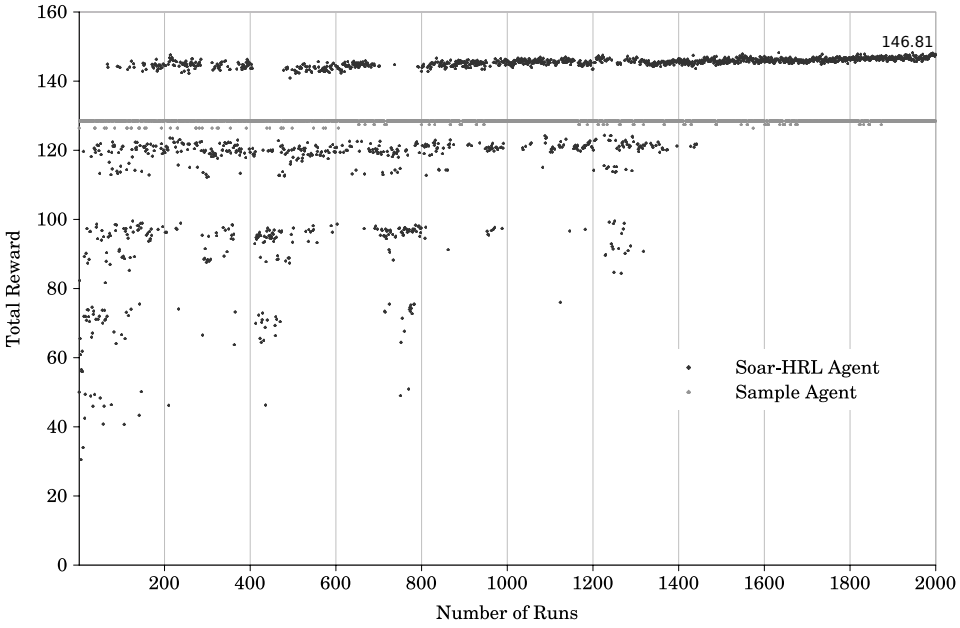

Figure 8: Performance of Soar-HRL agent (pre-programmed preferences) and Sample Agent in level type 0, difficulty 0

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 propositional agents in Section 7.1, that fail to converge to a policy even after training on 2000 runs through the episode. Three horizontal bands that seem to appear in the graph are because of the presence of three monsters in the game that kill Mario. The corresponding score on the vertical axes is the average reward earned by the agent before dying. The score corresponding to the topmost band is the average score earned by the agent when it finishes the level.

Results: Experiment 5

In the next set of experiments we investigated whether an agent can learn the selection knowledge about FLOs (the abstract, macro operators) through experience using RL. Results are summarized in Figure 9. The policy converges with a average total reward of 145.95 (averaged over last 100 runs).

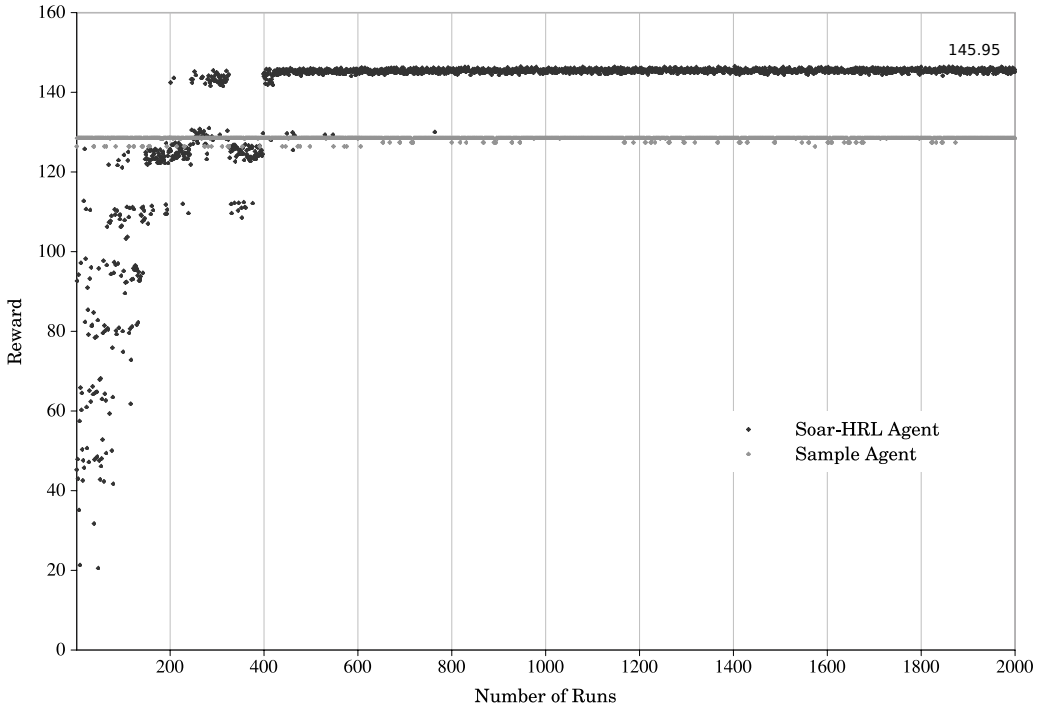

Figure 9: Performance of Soar RL agent (background knowledge) and Sample Agent in level type 0, difficulty 0

The performance of HRL agent is better than the agent with selection knowledge about FLOs described in *Experiment 1*. It converges to a good policy after only 600 runs where as the agent with pre-programmed FLO preferences takes about 1500 runs to converge to a good policy. When symbolic preferences are pre-programmed into the design of an agent, its choices are severely restricted such that it has to apply those symbolic preferences in every scenario. There could be situations where the pre-programmed preferences do not result into the best choice. Consider the scenario where a coin and a monster are within range, i.e. the attribute *isreachable* for the coin and *isthreat* for the monster is set. When symbolic preferences are hard-coded into the agent, a programmer might encode *if attribute 'isreachable' for coin is set and attribute 'isthreat' for a monster is set, then 'tackle-monster' is of higher priority*, limiting the choices an agent has. However, an agent that is learning FLO preferences can be designed to extract knowledge "*if attribute 'isreachable' for coin is set and a coin is st distance x and attribute 'isthreat' for a monster is set and monster is a distance y, then 'grab-coin' is of higher priority*". An agent that learns these preferences from experience learn a more specific policy and converges faster for a single episode. It is harder to hand-code specific symbolic preferences because it requires the programmer to know the exact distances, which are difficult to determine. This is an example where learning from experience allows an agent to acquire more specific knowledge than what can be hand-coded by a programmer.

7.2.1 Agent with Background Facts

Use of relational, state descriptions allow for the addition of more facts and background knowledge (procedural, semantic or logical) in the agent through hand programming. Rules like *'if there is a pit ahead, then jump while moving right*, or *'if a platform is reachable and there is a coin on it, then the coin is reachable too'* can be easily encoded. This allows the agent to learn to grab coins that that are on a platform and cannot be grabbed by jumping, by learning to climb the platform. The agent can benefit from the human spatial reasoning through such rules, and probably does not need to learn these fact from experience. The agents performance was similar to the agent described in Section 7.2 *Experiment 2*, however the agent could learn to grab extra coins and question-blocks result in a higher total reward (148.56 as compared to 145.95

in *Experiment 2*) on convergence.

Results: Experiment 6

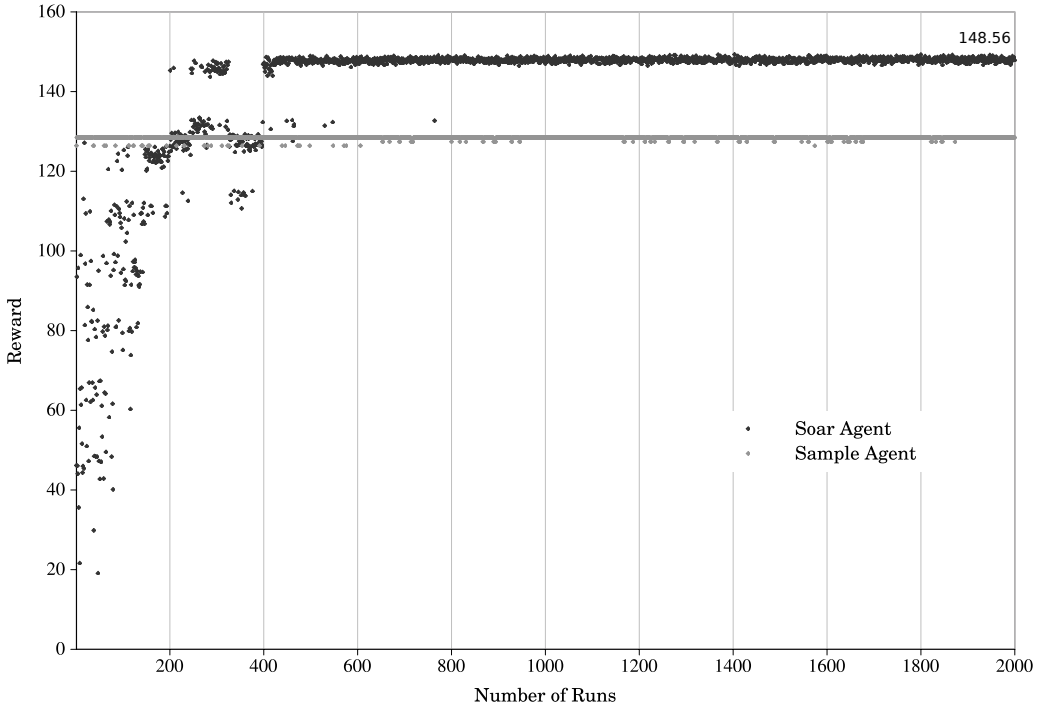

Figure 10: Performance of Soar-HRL agent (learned preferences) and Sample Agent in level type 0, difficulty 0

7.2.2 Agent with Internal Rewards

In the agent designs previously discussed, both levels of the hierarchy received only environmental reward which leads to incorrect updates in the value function in some situations. For example, an action of jumping may cause the agent to grab a coin and collect a positive reward while it is trying to learn to kill a monster.

To remedy this incorrect update, we introduced internal rewards in the design of the agent such that the agent had an explicit representation of the goal. In contrast to this, the previous designs had goals that were implicit in the reward received and next set of observations from the environment. Explicit representations of the goals were used by the agent to generate internal rewards if the goal conditions were met. Under this design, the agent received different rewards at different levels of the hierarchy. At the FLO level the value function was updated with the actual reward from the environment, however while executing the FLO, positive reward was generated by the agent only if goal conditions were met.

Including internal rewards at the lower level of the hierarchy helps learn actions that are associated with the correct goal of the FLO. Rewards received by the agent on performing other actions that earn a positive reward but are not associated with the correct goal are not used for updating the policy of the FLO currently being executed. This curbs the spurious rewards the agent receives while learning to perform a subtask.

Results: Experiment 7 Results are summarized in Figure 11. The policy converges with an average total reward of 145.98 (averaged over last 100 runs).

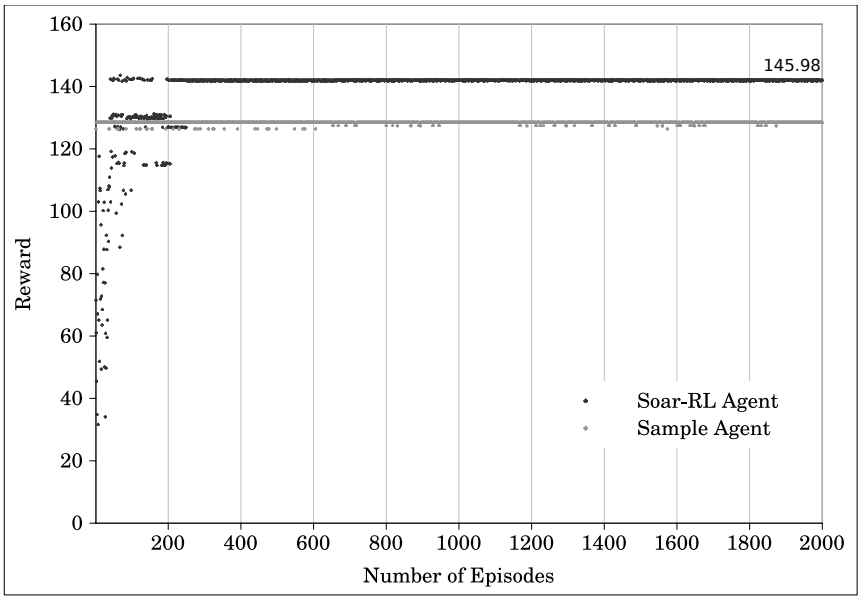

Figure 11: Performance of Soar-RL agent (internal rewards) and Sample Agent in level type 0, difficulty 0

The agent learns faster (200 runs through the episode) as compared to the agent previously discussed in Section 7.2, *Experiment 2*(took 400 runs to converge to a policy). Such formulation however can not alleviate the problem of partial observability which prevents the policy from converging to a high score in higher difficulties of the game. A specific example is discussed below.
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 there is a clear preference order. 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 there is no clear preference order amongst multiple objects, the agent fails to learn the best 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 12, 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 12: Example Scenario

Given the current design, the agent will make a selection between *Red Koopa* 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 suboptimal move in both the policies.

Partial Observability
In previous works, abstractions based on relational representations have been shown to introduce partial observability in domains, which is also true for formulations presented in this work. When the agent makes a selection between objects and learns to execute the associated FLO, it disregards the part of state-space that is not related to the selected object. While learning to execute tackle-monster, the agent disregards the other objects that might be near-by and might contribute to the reward. In specific cases where the object density around Mario is high, this leads to problems in learning a good policy because of incorrect updates to the value function. The problem of spurious rewards was slightly remedied by providing correct pseudo-rewards that are programmed into the design of agent itself, which lead to faster learning in specific instances of the game. However, at higher levels learning was severely limited.

Representing complete state while learning a policy for FLOs
Representing a state completely would involve including all interfering objects while learning to execute a a subgoal of killing a monster. Consider the example shown in Figure 12, a new FLO *tackle-two-monsters* will be proposed when two monsters are in vicinity and the state description would include *Red Koopa at distance x AND Green Koopa at distance y*.

However, this state space grows exponentially with the number of monsters, making learning intractable. Also, knowledge gained while interacting with one monster is essentially lost when interaction with two monsters in required, because dealing with two monsters is a new state for the reinforcement learning algorithm.

7.2.3 Agent with Object Based Coarse Coding of State and Linear Value function Approximation

Presence of objects in the vicinity of Mario affects its behavior and rewards. The value of executing an action in a state is dependent on which objects are included in the state. As we have seen in the earlier discussion, selecting an object based on prior experience and then learning about it leads to the problem of

partially observable state in a RL setting. Inclusion of all objects in the state description while learning an FLO leads to an exponential state space. A middle ground can be reached by coarse coding a state based on what objects are in vicinity and arriving at an approximate value function through linear combination of values functions from individual objects. The state space in which case is linear in the number of objects. Performance of the agent in level 1 is depicted in Figure 13. The policy converges with an average total reward of 61.93 (averaged over last 100 runs).

Results: Experiment 8

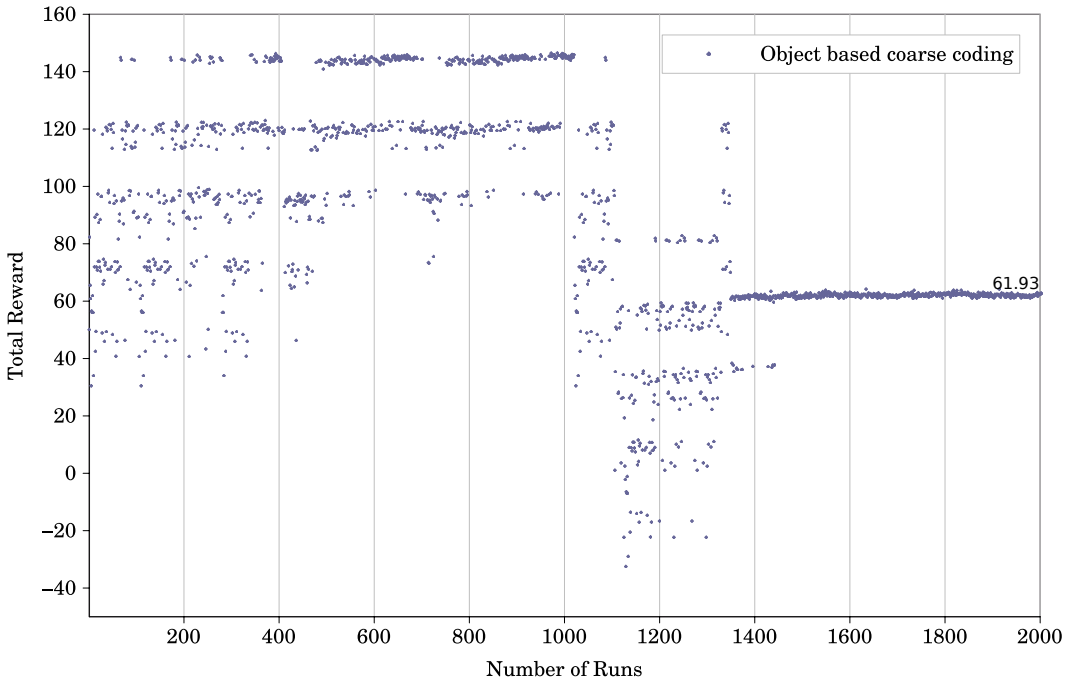

Figure 13: Performance of RL agent with object based sparse coding level type 0, difficulty 1

The value function converges to a small positive total reward, which is much smaller than the upper bound on the total reward that can be earned in the particular instance of the game.

Structural Credit Assignment Problem

The reason for such behavior arises from structural credit assignment problem. Consider Figure 12, where in the current setting, the value of any state action pair will be a linear combination of individual value functions of different monsters. If the agent chooses to *move-right*, the agent receives a high negative reward because it collides with the *Green Koopa* on the right. During update, all value functions receive a negative update, which then interferes with the policy previously learned. So if the agent encounters the *flying-koopa* is a different setting in future, its behavior will sub-optimal even if the value function for it had converged to an optimal policy during prior experience.

8 Conclusions

Through this work we have studied reinforcement learning in an action game Infinite Mario which has high dimensional state and spaces, and in characterized by complex relationships between its components. We showed that with a simple, typical reinforcement learning algorithm - SARSA, it is possible to put constraints on the state space based on first order relation amongst various objects in the domain such that when combined with hierarchical task decomposition and hierarchical learning, the agent learn faster and earns better total reward that with simplistic, propositional state representation. The agent was able to

acquire certain skills, and was able to apply learned but limited strategy while navigating in the domain. Object oriented representations when applied to relational domains naturally fit with task decomposition and hierarchical learning and can improve performance by putting reasonable constraints on state and action spaces.

However, as discussed in the text such designs may introduce partial observability in the learning framework, leading to limited or no learning. We investigated some changes in the design that could mitigate this problem, which leads to an exponential state space explosion and reduces transfer of learned knowledge. Linear value function approximation with coarse coding based on presence or absence of nearby objects cannot approximate the value function in this domain.

9 Future Work

In this domain, collisions are the most interesting events. A collision between Mario and a monster may result in different circumstances, some collisions resulting in a high positive reward and some in negative rewards. Similarly, collisions with other objects lead to different outcomes. Good models should be able to predict collisions and predictive features derived from these models can be used for reinforcement learning. As the domain is continuous, learning good models for the domain by experience alone may be a hard problem. Some knowledge about the physics of the game may be beneficial. We are currently looking at model-based learning methods where instance based models learned by experience can be used to reject actions that would lead to bad outcomes.

Wintermute (2010) showed that simple spatial imagery models can be used derive abstract states from agents perceptions which leads to good generalizations in games domains, and eventually into efficient learning. It is not clear if spatial reasoning is absolutely necessary in this domain, however since it has been shown to improve learning in agent operating in various games that share characteristics with Infinite Mario, this approach is worth exploring.

References

Dietterich, T. G. (2000). Hierarchical reinforcement learning with the maxq value function decomposition. *Journal of Artificial Intelligence Research*, 13(1):227303.

Diuk, C., Cohen, A., and Littman, M. L. (2008). An Object-Oriented Representation for Efficient Reinforcement Learning. In *Proceedings of the 25th international conference on Machine learning*, page 240247.

Driessens, K. (2001). Relational reinforcement learning. *Multi-Agent Systems and Applications*, page 271280.

Fikes, R. E., Hart, P. E., and Nilsson, N. J. (1972). Learning and executing generalized robot plas the research reported.

John, B. E. and Vera, A. H. (1992). A GOMS Analysis of a Graphic Machine-paced, Highly Interactive Task. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, page 251258.

Korf, R. E. (1985). *Learning to solve problems by searching for macro-operators*. Pitman Publishing, Inc., Marshfield, MA, USA.

Laird, J. and VanLent, M. (2001). Human-Level AI's Killer Application: Interactive Computer Games. *AI magazine*, 22(2):15.

Marthi, B., Russell, S., Latham, D., and Guestrin, C. (2005). Concurrent Hierarchical Reinforcement Learning. In *Proceedings of the National Conference on Artificial Intelligence*, volume 20, page 1652.

Nareyek, A. (2004). Ai in computer games. *Queue*, 1(10):58–65.

Ponsen, M., Spronck, P., and Tuyls, K. (2006). Hierarchical reinforcement learning with deictic representation in a computer game. In *Proceedings of the BNAIC*.

Precup, D., Sutton, R. S., and Singh, S. (1998). Theoretical results on reinforcement learning with temporally abstract options. In *in: Proc. 10th European Conference on Machine Learning*. Springer.

RL-Competition (2009a). Mario AI competition 2009. `http://julian.togelius.com/mariocompetition2009/`.

RL-Competition (2009b). RL competition 2009. `http://2009.rl-competition.org/`.

Rummery, G. and Niranjan, M. (1994). On-line Q-learning using Connectionist Systems. Technical report, Cambridge Univeristy Engineering Department, England.

Russell, S. and Norvig, P. (2003). *Artificial Intelligence: A Modern Approach*. Prentice-Hall, Englewood Cliffs, NJ, 2nd edition edition.

Sacerdott, E. D. (1973). Planning in a hierarchy of abstraction spaces. In *IJCAI'73: Proceedings of the 3rd international joint conference on Artificial intelligence*, pages 412–422, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

Spronck, P., Ponsen, M., Sprinkhuizen-Kuyper, I., and Postma, E. (2006). Adaptive Game AI with Dynamic Scripting. *Machine Learning*, 63(3):217248.

Sutton, R. and Barto, A. (1998). *Reinforcement Learning:An Introdution*. MIT Press, Cambridge, MA.

Sutton, R., Precup, D., and Singh, S. (1999). Between MDPs and semi-MDPs: A Framework for Temporal Abstraction in Reinforcement Learning. *Artificial intelligence*, 112(1):181–211.

Tanner, B. and White, A. (2009). Rl-Glue: Language-Independent Software for Reinforcement-Learning Experiments. *Journal of Machine Learning Research*, 10:21332136.

Watkins, C. J. and Dayan, P. (1992). Q-Learning. *Machine learning*, 8(3):279292.

Wintermute, S. (2010). Using imagery to simplify perceptual abstraction in reinforcement learning agents. *Ann Arbor*.

Woolridge, M. and Wooldridge, M. J. (2001). *Introduction to Multiagent Systems*. John Wiley & Sons, Inc., New York, NY, USA.

Appendix A - Infinite Mario Specifications

i. Actions

The action integers are an array of length 3: [-1,1], [0,1], [0,1]. They correspond with the buttons on a Nintendo controller, direction pad, A, B

- The first refers to the direction Mario is heading, with -1 for left, 0 for neither and 1 for right.
- The second refers to not jumping (0) or jumping (1).
- The third refers to the speed button being off (0) or on (1).

ii. Observations

The state space has no set specification, as its intArray and doubleArray sizes can grow depending on how many monsters are visible at any given time. The charArray is a constant size, representing the layout of the visible tiles.

intArray[0] is the x coordinate of the left-most visible tile, used to align the tiles represented in the charArray to the world.

For every monster, two values are appended to the intArray and 4 values are appended to the doubleArray. The values appended to the intArray are [type, winged]. The values appended to the double array are [x, y, x speed, y speed].

x and y are aligned with the tiles, but can have in-between values. x and y speed are in tiles per step.

Value	Type of Monster
0	Mario
1	Red Koopa
2	Green Koopa
3	Goomba
4	Spikey
5	Piranha Plant
6	Mushroom
7	Fire Flower
8	Fireball
9	Shell
10	Big Mario
11	Fiery Mario

If an enemy is winged, it bounces.

The charArray is a set of 16 rows separated by carriage returns, each with 21 chars. Thus, there are 16*(21+1)=352 elements in the charArray. Each element represents a tile in the game. The row a tile appears in determines its y position, the last row being y=0, and y increasing as the row number decreases. The x position is determined by the column + the offset found in intArray[0]. If the charArray is printed to a terminal, it will look like the game, without the monsters.

The tiles can be any of the following:

Value	Type of Tile
0-7	a bit vector indication which side of the tile is solid
b	brick
?	a question-block
$	a coin
—	a pipe
!	finish line
M	tile Mario is standing on

Tiles with x position < 0 are considered always solid.