Mario, State Abstractions and the Wonderful World of Options

Richard Gibson and Nick Abou Risk

April 9, 2009

1 Introduction

For this study, the “Infinite Mario” domain from the RL 2009 Competition was chosen. This is a challenging new domain, added to the RL Competition, based on the “Mario Bros.” video games. The main character, Mario, is the agent in this episodic task ($\gamma = 1$) and attempts to accumulate as much reward as possible before the level terminates (via death, finish line, or exceeding a time step threshold).

The state space in this domain is extremely large and ever-changing. Thus, state abstraction techniques were used to reduce the state space considerably by choosing features which captured the essence of the environment. These were chosen based on both expert knowledge and knowledge of the static reward system of the environment. Egocentric representation was used to aggregate many similar states to a single abstract state (e.g. measuring the relative position between Mario and another object instead of a conjunction of their exact positions).

Several, on- and off-policy, learning methods were used in this study. Function approximation, mostly through tile coding, was also used extensively. The learning algorithms included: several Sarsa variants (with and without eligibility traces), Watkins's Q(λ)-learning, and finally option learning. Options turned out to be quite powerful as there are many intrinsic subtasks in this domain. Trying to develop a single agent with a state abstraction strong enough to do well in many levels with differing reward systems was nearly impossible. Instead, we employed the idea of having several specialists that a Master Agent (analogous to a coach) could choose from depending on the situation. This resulted in a much more robust agent that generally did well all throughout a run (typically 1000 episodes).

2 Infinite Mario Domain

2.1 Actions

The action space for this domain consists of three independent and discrete dimensions corresponding to a Nintendo controller:

1. Direction (direction pad): -1 for left, 1 for right, 0 otherwise
2. Jumping (A button): 1 for jumping, 0 otherwise
3. Running (B button): 1 for running, 0 otherwise

This produces a total of 12 ($= 3 \cdot 2 \cdot 2$) discrete actions.

2.2 Observations

The representation of the state made available to our agent is called an observation. Each observation contains information about tiles and monsters.

The game screen is comprised of a matrix of tiles (16 rows by 21 columns). This matrix is provided as an array of characters, with each element representing the type of tile: brick, question block, coin, pipe, finish line, Mario, and different types of solids.

There are 9 types of monsters: Mario, Red Koopa, Green Koopa, Goomba, Spikey, Piranha Plant, Mushroom, Fire Flower, and Fireball. However, not all monsters are Mario's enemies (monster is a misnomer presented in the domain description of "Infinite Mario" on the RL 2009 Competition website). Of these monsters only Goombas, Koopas, Spikeys, and Piranha Plants are enemies. For each monster on-screen, the following information is provided by the observation:

- x- and y-positions (continuous)
- x- and y-velocities (continuous)
- winged/unwinged (discrete)
- type (discrete)

2.3 Levels

Levels are automatically generated in this domain through four specified numbers: seed, type, difficulty, and instance. The seed and difficulty change how the terrain and monsters are placed on the level map. The type specifies whether the level is indoors or outdoors. Finally, the instance determines the rewards, physics, and map length. Unless otherwise specified, all tests in this study were performed with the default level seed (121), type (0 for outdoors), difficulty (0 for least difficult), and instance (0). The reward system corresponding to each instance is static. Here is the reward system corresponding to the default instance, 0:

- +100 : reaching the finish line
- +1 : collecting a coin
- +1 : hitting a question block
- +1 : killing an enemy
- -0.01 : every passing time step
- -10 : dying (via pits, enemies, or exceeding 1000 steps per episode)

3 State Abstraction

The observation space described previously is effectively infinite in size (due to the continuous positions and velocities). Thus, we must create a state abstraction that encapsulates the essence of the given observation. Ultimately, we decided to extract features from the observation (through parsing and processing) that seemed important. More specifically, we extracted features directly related to the reward system of the default instance.

3.1 Egocentric Representation

Often in the Mario domain, the exact coordinates – either global or local – of Mario and nearby objects are not necessary. What is important, however, is the relative distance between Mario and these objects. For example, consider the scenario where the only monsters on-screen are Mario and a Goomba and the terrain is flat (so they have the same y-coordinate). In this example, if we abstracted the state to simply be the conjunction of Mario's x-coordinate, x_M, with the Goomba's x-coordinate, x_G, then $s_1 = (x_M, x_G) = (1, 5) \neq s_2 = (2, 6)$. In this abstraction, we would have to learn the value of all states $(x_M, x_M + 4)$ even though they are equivalent, for all intents and purposes. Thus, we should aggregate all states (x_M, x_G) such that $x_G - x_M = d$ to a new state $s' = d$. By reducing the number of abstract states in this fashion, we will use less memory (to store the value function) and will speed up both efficiency and the learning rate.

This so-called egocentric representation was employed in this study. Figure 1 demonstrates this concept diagrammatically. All relative positions are measured with respect to Mario, and can be either positive or negative. Also, one of Mario's goals is often to reach the finish line which is always located on the right-hand side of the map in this domain. So, some of the most important features extracted to fit this representation include: distance to the nearest enemy, distance to the finish line, distance to the nearest coin, distance to the nearest question block, and distance to the nearest pit (a hole or crevasse in the terrain). These features were chosen because each object listed is directly related to the aforementioned reward system.

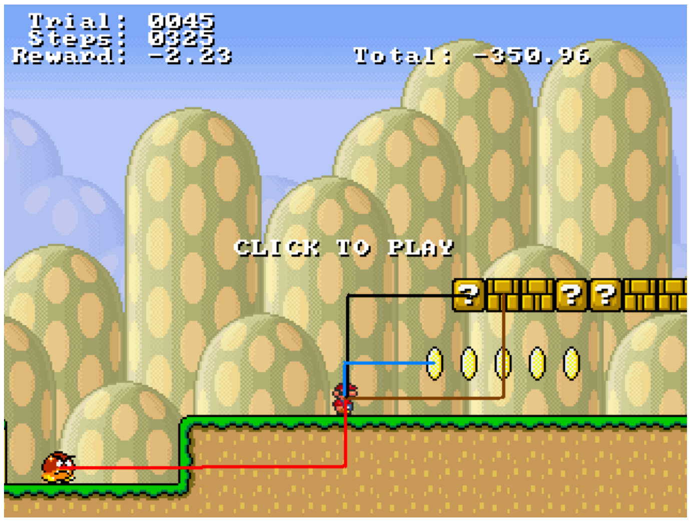


Figure 1: Screen shot of Mario and the relative position measurements to the present features: coin, question block, brick, and enemy.

Only the nearest object of each type is included as a feature in the state abstraction. This choice was made to simplify the state space whilst maintaining a high amount of useful information. However, for much higher difficulty levels (which include clusters of closely-packed enemies), a better abstraction for enemies would be required. New features such as the distance to the second-nearest enemy or the centroid of all on-screen enemies could be useful in such a situation.

Some other useful yet non-egocentric features were extracted from the observation. Some of these include Mario's size (small, big, or fiery) and the positions, velocities, and directions of on-screen monsters.

3.2 Tile Coding

Tile coding has been used extensively in this study in several different ways. Naturally, the continuous positions and velocities were tiled; they were often conveniently tiled to match the size of the on-screen tiles. That is, relative positions were stored in terms of how many unit tiles Mario was from an object. Similarly, the velocities were often stored in terms of how many unit tiles Mario would move in each direction in one time step.

Grid tiling – conjunction of all present features – and striping – generalization over many feature dimensions – were utilized. In general, striping outperformed grid tiling; striping learned very quickly and played fairly competently within relatively few episodes. However, given sufficient training episodes and removing the condition of max steps per episode, grid tiling would likely do better in the long run. The agents using grid tiling would really maximize reward locally but spend too much time accumulating reward and often not reach the finish line in time.

One might think that the specifics of the terrain are very important for Mario's success. However, it turns out that, in general, this is not the case. Aside from knowledge of pits (introduced in Section 6), the abstractions used were completely terrain-independent. As a result, Mario learned to jump more often than a typical human would (and likely an AI agent with terrain knowledge, for that matter) in order to surpass obstacles. The frequencies of actions taken over many episodes were recorded. Perhaps not surprisingly, the most frequent action taken was (right, jump, run). Thankfully, this is consistent with expert knowledge.

4 Implementation

Several learning algorithms and action selection policies were used throughout this study.

4.1 Learning Algorithms

The learning algorithms used were Sarsa(0) with grid tiling, Sarsa(λ) with grid tiling, Sarsa(λ) with striped tilings, Watkins's Q(λ)-learning, and option learning. Figure 2 compares the Sarsa methods just mentioned. Sarsa(λ) with striped tilings performs the best while Sarsa(0) performs the worst. Note that combinations of grid tiling and striped tilings (i.e. having a conjunction of multiple stripes) were tested. Natural combinations such as putting the x- and y-distances together and generalizing along all other dimensions were attempted. To our disappointment, however, these did not perform better than the striped tilings. These combinations ended up receiving about the same total reward over a 1000 episode run, but learned at a slower rate.

Replacing traces (with zeroing non-selected actions) were used for the methods compared in Figure 2 with a value of $\lambda = 0.9$ and all throughout this study. Accumulating traces and non-zeroing replacing traces were also recently tested. As can be seen in Figure 3, accumulating traces outperformed both replacing trace methods (with the zeroing method doing better than the non-zeroing method). However, accumulating traces did not do significantly better and, given more time, should be averaged over many 1000 episode runs.

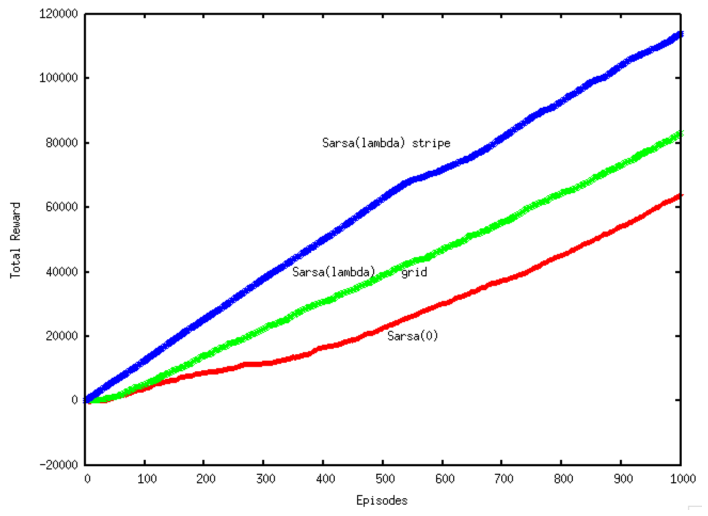


Figure 2: Comparison of three different Sarsa learning methods. The top blue curve corresponds to Sarsa(λ) with striped tilings, the middle green curve corresponds to Sarsa(λ) with grid tiling, and the bottom red curve corresponds to Sarsa(0) with grid tiling. Replacing traces and values of $\alpha = 0.025$ and $\lambda = 0.9$ were used to produce this data. Each data set was computed from a single run of 1000 episodes.

4.2 Action Selection

The policy method used in this study was almost exclusively ϵ-greedy action selection. Each non-greedy action is selected with a probability of $\frac{\epsilon}{12}$ where $|A(s)| = |A| = 12$ for all states, s, since our action space is independent of our state.

Another action selection method, Gibbs-Boltzmann softmax, was recently implemented. Figure 4 shows a comparison in the total reward earned in 1000 episodes for both softmax and ϵ-greedy methods, with softmax doing significantly better.

In retrospect, softmax should have been implemented earlier and included in the tests as it seems to be better suited for this domain. For example, when near an enemy, ϵ-greedy action selection makes the non-greedy actions that result in Mario's death equiprobable to all other non-greedy actions. However, such a risky (and bad) action would be taken with a much lower probability using softmax action selection. This is due to the probability of taking an action with softmax being proportional to $e^{Q_t(a)/\tau}$. Also, softmax gives a near-greedy action a much higher probability of being chosen than ϵ-greedy action selection would. For example, the actions (right, jump, run) and (right, no jump, run) are likely to be close in value in many states in this domain. One fallback of using softmax is that choosing τ requires knowledge of the reward system, and thankfully we do have such knowledge for this problem in the default instance.

5 Creating a Team of Agents Via Options

Our learning agents described so far perform quite well on a number of levels and instances. They do a good job of killing monsters and collecting coins that are on-screen, and can eventually learn to finish levels when a large reward is received for doing so. On the other hand, the example

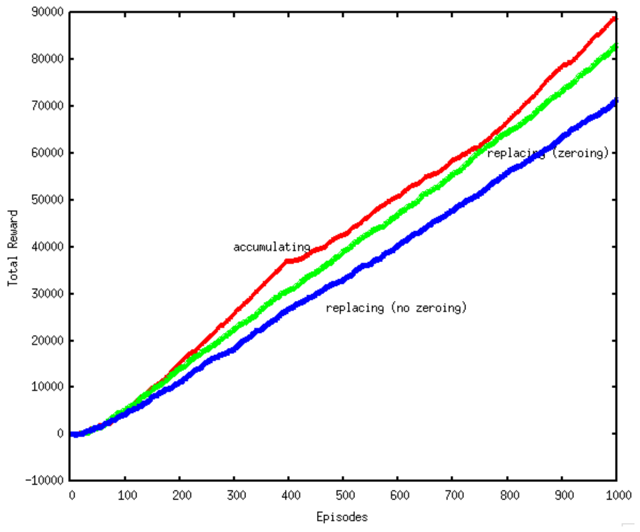


Figure 3: Comparison of three different eligibility trace update methods. The top red curve corresponds to accumulating traces, the middle green curve corresponds to replacing traces (zeroing non-selected actions), and the bottom blue curve corresponds to replacing traces with no zeroing. Values of $\alpha = 0.025$ and $\lambda = 0.9$ were used to produce this data. Each data set was computed from a single run of 1000 episodes.

agent in the Mario domain package, ExMarioAgent, is mostly a rule-based agent that excels in completing levels quickly and collecting coins on Mario's path to the finish line. It has very good heuristics for jumping when enemies are nearby or when coins are within Mario's grasp, and repeats all but its last seven actions from the previous episode. An agent that could behave at times like our learning agents and at other times like ExMarioAgent would likely perform better than either agent alone over a broad set of levels and instances. In this section, we explore this idea.

In general, consider a collection of n agents, labeled 1 through n, where agent i has its own state-action value function estimate $Q^{(i)}(s,a)$, policy $\pi^{(i)}(s,a)$ and state-action features $\phi^{(i)}(s,a)$. In state s_t, we wish to choose the agent that we believe will allow us to achieve the best cumulative reward following state s_t, given that we follow the agent's policy at time t. So, consider a coach or a "Master Agent" which instead of selecting actions such as "run to the right and jump," selects options that indicate which of the n agents to poll for an action. Thus the Master Agent has a state-option value function $Q(s,o)$ and hierarchical policy $\mu(s,o)$, and the collection of agents become options of the Master Agent. Note that the Master Agent itself also needs state-option space features $\phi(s,o)$ that suit the Master Agent's needs. We suggest setting $\phi(s,o)$ to the concatenation of the feature vectors $\phi^{(i)}(s)$ (removing the action features) of all of the options combined with the option o (thus including the option through conjunction rather than its own stripe tiling). Since the Master Agent should know about the features that its options know about, we feel that is in a sense the minimum feature set for the Master Agent, and is what we use throughout the rest of this paper. We give a basic diagram of this approach below in Figure 5.

We keep our implementation of option learning as simple as possible. The Master Agent's

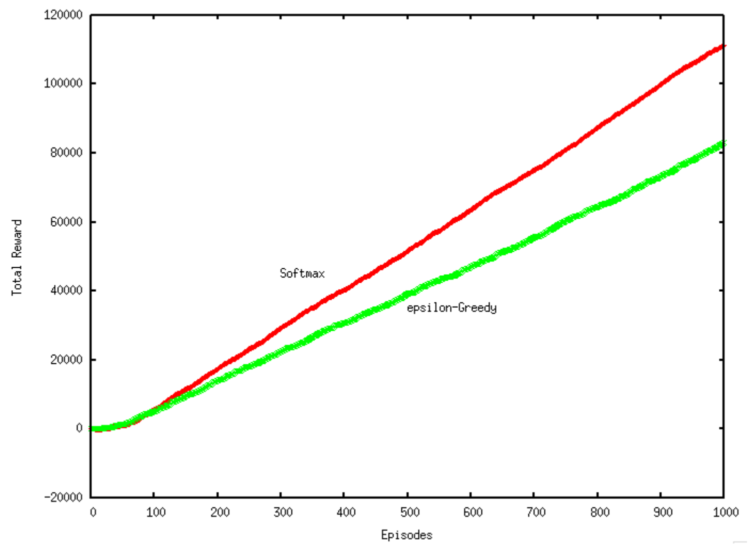


Figure 4: Comparison of two action selection methods: Gibbs-Boltzmann softmax ($\tau = 0.05$) and ϵ-greedy ($\epsilon = 0.02$). Softmax is the top red curve and ϵ-greedy is the bottom green curve. Sarsa(λ) with grid tiling, $\lambda = 0.9$, and $\alpha = 0.025$ was used. Each data set was computed from a single run of 1000 episodes.

state-option value function $Q(s, o)$ is estimated via linear gradient-descent Sarsa(λ), while intra-option learning of each of the $Q^{(i)}(s, a)$ is done using linear gradient-descent Watkins's $Q(\lambda)$ (each using replacing traces). We assume that every option terminates after a single time step, which allows us to avoid worrying about interrupts. For now, we also assume that every option is available in every state (we remove this assumption in Section 6). Thus given the 6-tuple $(s_t, o_t, a_t, r_{t+1}, s_{t+1}, o_{t+1})$ at time step $t + 1$ (where a_t is the action given by option o_t), we update $Q(s, o) \approx \theta^T \phi(s, o)$ via

$$\Delta\theta_{t+1} = \alpha \left(r_{t+1} + \gamma \theta_t^T \phi(s_{t+1}, o_{t+1}) - \theta_t^T \phi(s_t, o_t) \right) \vec{e}.$$

We note here that not only is the eligibility trace set to 1 for (s_{t+1}, o_{t+1}) after the above update, but also for all (s_{t+1}, o) where option o would have delivered the same action as o_{t+1} in s_{t+1}. Then, for each $Q^{(i)}(s, a)$, we first reset all other eligibility traces to zero if option i would not have taken action a_t in s_t, and then perform the update

$$\Delta\theta_{t+1}^{(i)} = \alpha^{(i)} \left(r_{t+1} + \gamma^{(i)} \max_a \theta_t^{(i)T} \phi^{(i)}(s_{t+1}, a) - \theta_t^{(i)T} \phi^{(i)}(s_t, a_t) \right) \vec{e}^{(i)}. \tag{1}$$

The corresponding policies are then improved using ϵ-greedy option/action selection.

Our experiments in this section consist of a Master Agent with just two options. The first option, which we call "ExMario", is a rule-based option which uses the same heuristics as ExMarioAgent, but does not repeat actions from the previous episode. No learning is done within this option. The second option is taken from one of our previous agents that use 6 tilings to abstract the state-action space. We name this option "Tile Coder." Figure 6 below displays this setup. The parameters used for the Master Agent were $\epsilon = 0.025$, $\gamma = 1$, $\alpha = \frac{0.1}{6}$ and $\lambda = 0.9$, while the parameters used for the Tile Coder option were $\epsilon^{(2)} = 0.02$, $\gamma^{(2)} = 1$, $\alpha^{(2)} = \frac{0.1}{6}$ and $\lambda^{(2)} = 0.9$.

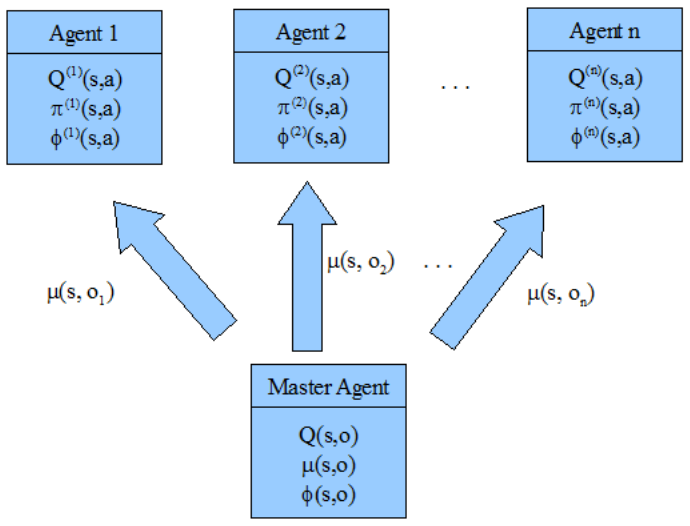


Figure 5: A team of agents used as options in the Master Agent.

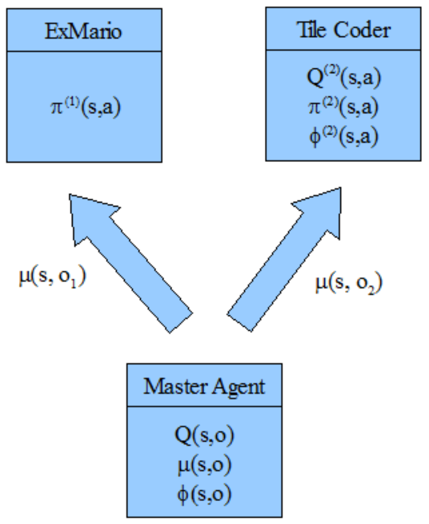


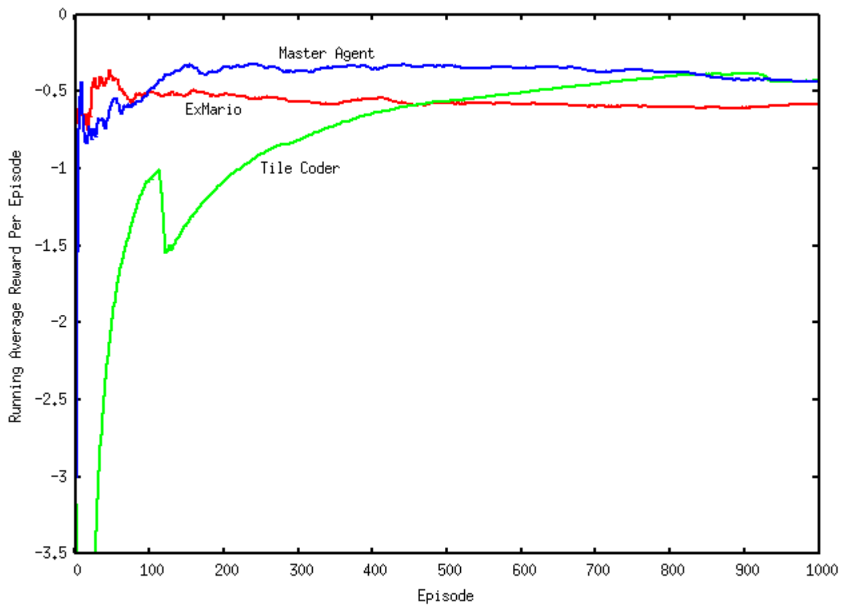
Figure 6: A Master Agent with just two options, one derived from ExMarioAgent (which does not improve its internal policy) and the other derived from one of our previous learning agents.

Two different instances were used to compare the quality of the Master Agent against the qualities of each of its options running alone. In the first instance (instance 1 in the Mario domain package), the only rewards are +1 for killing a monster and −0.01 at every time step. In the second (instance 3 in the domain package), the only rewards are +1 for collecting a coin and −0.01 at every time step. Figure 7 below shows the results of our experiments. In the first instance, the Tile Coder takes time to learn how to kill monsters for reward, but eventually is able to do a better job than the ExMario option. In the second instance, the Tile Coder does well initially as there are a lot of coins to collect (and random actions work fairly well in doing so), but then its performance drops off well below that of the ExMario option. We believe this is due to both its inability to complete levels (since no positive reward is gained for doing so), and its belief that getting killed by a monster might actually be a good thing when coins are still out of sight (as death results in no direct penalty as well as terminates the small negative rewards each step). However, the Master Agent performs well in both instances and by the end of 1000 episodes, its overall performance is on par with the better of the two options for that particular instance. Thus if we do not know which instance we are running beforehand, as in the RL Competition, we would prefer to use the Master Agent over either of its two options alone.

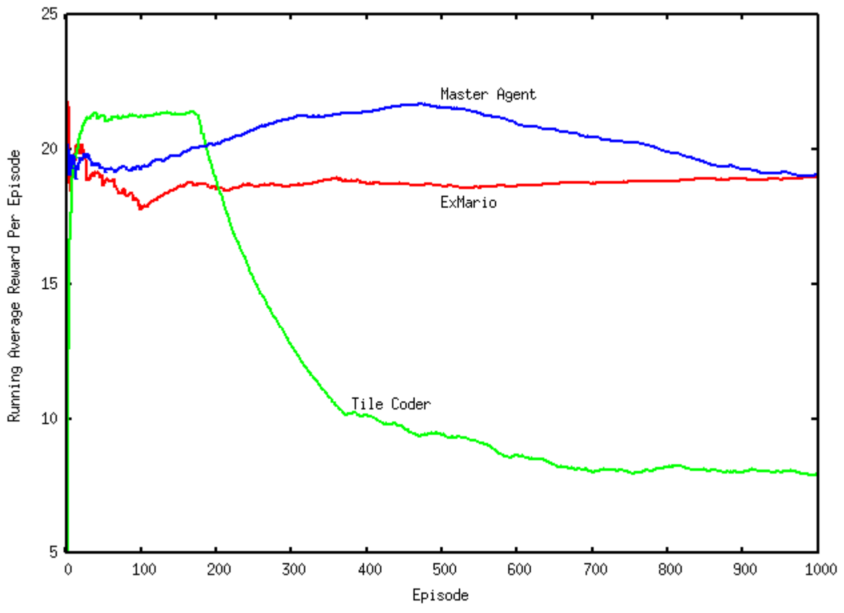
6 Addressing Weaknesses in Agents Via Options

One frustrating aspect of working in the Infinite Mario domain is that ExMarioAgent performs quite well on very difficult levels. For example, ExMarioAgent can achieve over +10,000 reward on difficulty 8 of 10 in less than 1000 episodes. For these tough levels, our learning agents do poorly as more complicated state abstractions are necessary for dealing with things such as multiple monsters of differing types. However, ExMarioAgent is not perfect; for example, he has a lot of trouble navigating across tricky pits with stairs on either side (see Figure 8 below). If ExMarioAgent could get past these types of pits, then he may be able to actually complete these difficult levels and thus achieve even greater reward (+100 each time the goal is reached). Can we learn to improve individual weaknesses, such as avoiding pits, without decreasing performance through the rest of the level?

We take an approach to this problem similar to the approach described in the last section. Here, we create three options for our master agent, which we will call "Anti-Pit Agent." First, we derive two options based on the behavior of ExMarioAgent. One of these options simply repeats the best sequence of actions taken in an episode thus far, where best is determined by total reward for the episode. This option is only available at the beginning of an episode until another option is taken, and like ExMarioAgent we automatically terminate the option before the last seven actions in the sequence. The nice thing here is that because the Mario domain is "nearly" deterministic, Anti-Pit Agent has a very good estimate of the value of taking or continuing this option, namely the remaining cumulative reward received in the best episode thus far (so rather than learning this option's value, we use this instead). There is one more catch to this option: We only remember "safe" sequences. If we repeat an action sequence and the episode ends before the sequence terminates, we deem the previous sequence not safe and erase it from memory. If the domain were completely deterministic, this would be unnecessary; however, we have found cases where Mario jumps when very close to a monster which kills him in some episodes but not in others. This is not good because in many instances, being killed results in a negative reward.



(a) Mario domain instance 1



(b) Mario domain instance 3

Figure 7: The running average reward per episode for the ExMario option alone, the Tile Coder option alone and the Master Agent on two different instances of level 121, type 0 and difficulty 0. Each data set was computed from a single run of 1000 episodes.

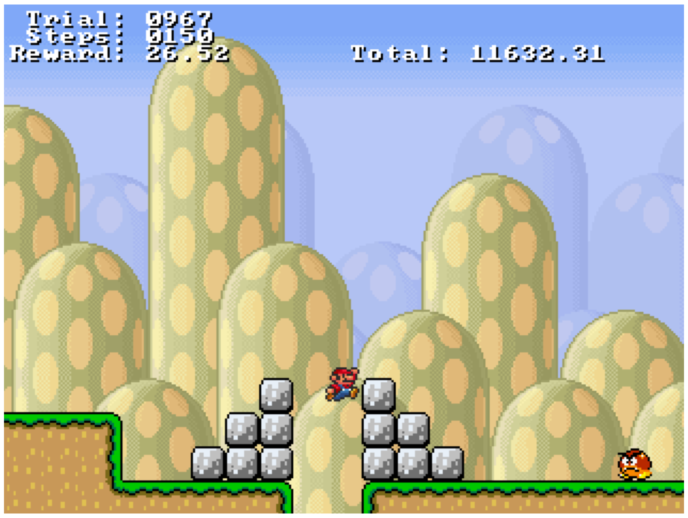


Figure 8: ExMarioAgent falling into a tricky pit on level 105, type 0, difficulty 8 and instance 0.

The second of Anti-Pit Agent's options is the same as the ExMario option from the previous section. The third option is the "Pit Specialist" and is only concerned with pits. The state abstraction used here is a grid tiling containing the relative position of Mario from the left edge of the pit, the width of the pit, the heights of the three tiles immediately preceding the left edge of the pit and the velocity of Mario. This option is only available to Anti-Pit Agent when a pit is at most 4 tiles in front of Mario. Finally, we include an extra set of 4 overlapping tilings within Anti-Pit Agent for the position of Mario as we found that our initial evaluations of the ExMario option were poor without these. A basic diagram of this setup is shown in Figure 9 below.

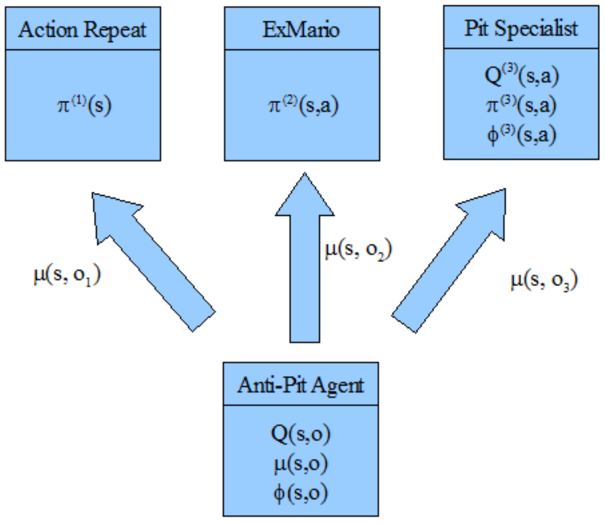


Figure 9: The options for Anti-Pit Agent.

Because the Pit Specialist is not available in every state, its learning rule differs slightly to (1). Rather than updating at every time step, it only performs backups and eligibility trace updates at states where it is available. So, if s_{t+k} is the first state since s_t in which the Pit Specialist is available, we update our estimate of $Q^{(3)}(s, a)$ via (omitting the superscripts)

$$\Delta\theta = \alpha \left(\sum_{m=1}^{k} \gamma^{m-1} r_{t+m} + \gamma^k \mathrm{max}_a \theta^T \phi(s_{t+k}, a) - \theta^T \phi(s_t, a_t) \right) \vec{e}.$$

Note that when $k = 1$, this is equivalent to (1).

For the Pit Specialist, we consider two different reward signals. Firstly, we simply use the reward from the environment just like in the previous section. However, this may result in a poor learning rate since the environment does not immediately reward the agent for successfully jumping over a pit. For instance, if Anti-Pit Agent clears a pit and then is quickly killed by a monster on the other side, the Pit Specialist will interpret this death as equivalent to simply jumping into the pit. To perhaps compensate for this, we can attempt to detect pit clearances or pit deaths and reward or penalize the Pit Specialist appropriately with our own custom rewards. In this scenario, we reset the eligibility traces to 0 when the option becomes unavailable so that the Pit Specialist does not "watch" what happens after the pit is cleared.

Figure 10 below compares ExMarioAgent and Anti-Pit Agent with and without custom rewards for the Pit Specialist on a level of difficulty 8. Anti-Pit Agent uses linear gradient-descent Sarsa(λ) with parameters $\epsilon = 0.05$, $\gamma = 1$, $\alpha = 0.02$ and $\lambda = 0.95$. We also introduce two new parameters to Anti-Pit Agent that somewhat resemble using interrupts. The first is $\omega = 0.8$ for the probability of choosing an option different from the previous option o_t when $Q(s_{t+1}, o_t) < \mathrm{max}_{o'} Q(s_{t+1}, o')$. The other is $\epsilon_2 = 0.0001$ for the probability of randomly choosing a new option. The Pit Specialist uses linear gradient-descent Watkins's$Q(\lambda)$ with parameters $\gamma = 1$, $\epsilon = 0.025$, $\alpha = 0.1$ and $\lambda = 0.95$. For the custom rewards case, the Pit Specialist receives -0.5 reward for falling into the pit, -1 reward for running away from the pit, 0.1 reward for clearing the pit and -0.01 reward on other time steps.

On average, Anti-Pit Agent was able to reach the goal after about 700 episodes, whereas ExMarioAgent failed to reach the goal within 1500 episodes. Once the goal has been reached, the Action Repeat option becomes highly valuable and Anti-Pit Agent is able to repeatedly complete the level for a lot of positive reward. Contrary to our prediction, the custom rewards variant did not perform the best of the three. However, the results are only over 2 runs so we cannot conclude for certain that using custom rewards was a failure. Unfortunately, runs of 1500 episodes take a long time to complete and thus we were unable to produce more data. Also, ExMario is the more dominating option and much of the performance relies on how quickly it can find a way past all of the monsters, so the Pit Specialist is not always to blame for a bad run.

7 Conclusion

A major fact that we take away from this project is that state abstractions require a lot of thought and must be managed with care. We created numerous different tiling schemes before finally creating abstractions that were good enough to match ExMarioAgent's performance on just the easiest difficulties. All of our tilings use uniformly-sized increments along each dimension. Since things really close to Mario are more important to detail than objects further away, tilings which

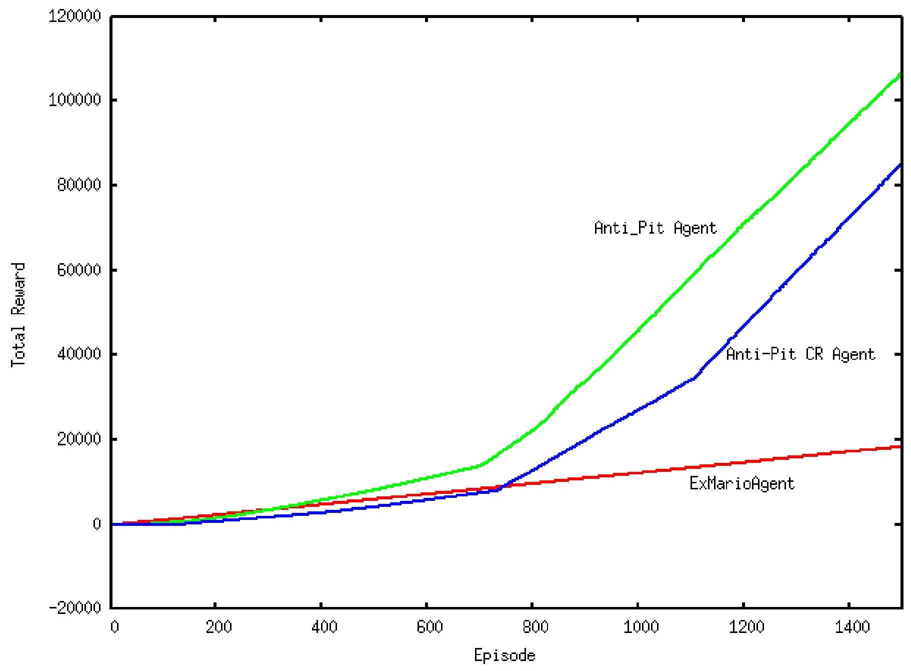


Figure 10: Total reward earned by ExMarioAgent, Anti-Pit Agent, and Anti-Pit CR Agent (custom rewards) after 1500 episodes on level 105, type 0, difficulty 8 and instance 0. The results are averaged over two separate runs.

use logarithmic widths might be a good idea in this domain. This is just one of the items on our long state abstraction to-do list that we unfortunately did not find time to test.

Options are very powerful, both as a way of combining different agents and tweaking an existing agent without entirely disrupting its general behavior. A significant source of this power, which may have gone unmentioned, is the ability to partition the state space so that each part is abstracted independent of the rest of the space. For instance, the Pit Specialist looks only at pits and ignores other important features of the observation. This allows us to include more specific features in our state abstraction that we may wish to omit in a tile coding agent without options. Along with numerous tweaks which we will not go into here, we want to expand Anti-Pit Agent to include more options, perhaps one for dealing with monsters, and another for detecting coins.

The workload was divided as follows: Nick worked on the general state abstraction, including the grid and stripe tilings, as well as soft-max action selection and accumulating/replacing trace tests. Richard worked on integrating options and built the Pit Specialist.