Technical Report AI08-9, The University of Texas at Austin, Department of Computer Sciences.

Competition Between Reinforcement Learning Methods in a Predator-Prey Grid World

Jacob Schrum (schrum2@cs.utexas.edu)
Department of Computer Sciences
University of Texas at Austin
Austin, TX 78712 USA

Abstract

Tabular and linear function approximation based variants of Monte Carlo, temporal difference, and eligibility trace based learning methods are compared in a simple predator-prey grid world from which the prey is able to escape. These methods are compared both in terms of how well they lead a prey agent to escape randomly moving predators, and in terms of how well they do in competition with each other when one agent controls the prey and each of the predators is controlled by a different type of agent. Results show that tabular methods, which must use a partial state representation due to the size of the full state space, actually do surprisingly well against linear function approximation methods, which can make use of a full state representation and generalize their behavior across states.

1 Introduction

The most popular reinforcement learning algorithms are designed to function in single agent systems, which is to say that these algorithms treat other agents that may be present simply as part of the environment.

In cooperative multiagent environments, this is not necessarily a problem. Different agents working together gradually learn to adjust to changes in their fellow agents, and as all agents become more skilled at cooperating, they can depend on fairly consistent, gradually improving behavior from their fellows. For this reason it is acceptable to treat these agents as part of the environment. In fact, it is often ok to have multiple cooperating agents that share a policy (Tan 1993).

Competitive environments pose a problem however, since in many competitive domains one can maximize its reward by drastically changing strategies in a way that the opponent will not expect. Such considerations have led to many algorithms that focus on finding a Nash equilibrium between competitors (Greenwald and Hall 2003), distinguishing between friends and foes (Littman 2001), and determining *best-response* policies (Weinberg and Rosenschein 2004).

The work described in this paper takes a step back from such considerations and presents a direct competitive comparison of classical reinforcement learning methods. The objective is not to find behavior that is optimal in a competitive environment, but rather to compare these classical methods to see if the manners in which they learn interact with each other in a way that allows certain learning methods to overcome others, be it due to differences in learning rate, representational power, or state space representation used.

Many different reinforcement learning methods are put in competition with each other in a predator-prey grid world domain. Descriptions of the reinforcement learning methods used in this paper are presented first, followed by a detailed description of the domain, experimental approach, results, discussion and conclusion.

2 Reinforcement Learning Methods

Reinforcement learning allows an agent to learn a task online by using rewards provided by the environment. At every discrete time step the learning agent chooses one of a discrete set of actions to perform. The available actions depend on the state the agent is in. Every action sends the agent to another (possibly the same) state and provides it with a (possibly zero) reward. These transitions may be stochastic.

Most reinforcement learners use the rewards provided by the environment to update a value function, $Q(s, a)$, which numerically expresses the utility of a given action a in a given state s. In the simplest case, Q is represented by a table. From the value function, an agent derives a policy $\pi(s)$, which indicates which action to take in state s. This policy can be derived in several ways, but in order to be effective, the policy must favor actions that are known to be good (*exploitation*) while still trying other, less certain actions (*exploration*). The action selection method used by all agents in this paper is ϵ-greedy, in which the agent normally chooses what it perceives to be the best action (maximum utility for the given state: $\mathrm{argmax}_a\, Q(s, a)$), but for ϵ portion of the actions it will choose a random action. The ϵ value used in all experiments in this paper was 0.1.

If the reinforcement learning method updates the value function only according to the actions it takes, including random ones, then it is an *on-policy* method. If updates are performed that do not always reflect the action taken, then the method is *off-policy*. Off-policy methods have an easier time learning an optimal policy because the policy they learn does not always reflect their actions, which may be suboptimal with respect to the current values function. However, on-policy methods may actually perform better during online evaluation because the policies they learn take into account the possibility that they may be forced to take a random action at any point. This makes them slightly more cautious.

The utility values that a reinforcement learner learns represent the expected long term sum of rewards (*expected return*) that will be accumulated by taking the given action in the given state, and then continuing to follow the currently

known policy, which incorporates the chance of random actions for an on-policy method. Therefore, an accurate value function allows an agent to attain optimal global behavior by choosing locally greedy actions. It is common to decrease the value of rewards that are received further in the future by multiplying them by γ^k, where k is the time step on which the reward was received and $0 \leq \gamma \leq 1$ is a parameter called the *discount rate*. Therefore the return R_t received from time t onward is defined as:

$$R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}$$

where r_t is the reward at time t. Discounting is generally required in continuous tasks, but is optional in episodic tasks such as the one used in this paper. The above equation still holds for episodic tasks if we define r_t for all t after the end of the episode to be zero. For the experiments presented in this paper a γ of 0.9 was used.

Several different reinforcement learning methods are applied in this paper, each of which learns its value function in a different way. Each method has its own strengths and weaknesses, which depend on the nature of the environment, the way the environment is represented, and nature of other agents in the environment. A summary of all methods used in this paper follows. For a more in depth look at these methods, see Sutton and Barto 1998.

2.1 Monte Carlo Methods

Monte Carlo methods are based on averaging sample returns. At the end each episode the utility of every state/action pair that was visited in the course of the episode is updated by

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha(R_t - Q(s_t, a_t))$$

where R_t is defined above, $0 < \alpha < 1$ is the *learning rate parameter*, and t refers to the first time that the given state/action pair was visited in the course of the episode. This update method is known as *first visit Monte Carlo*. Because updates are performed on all state/action pairs visited in the episode regardless of whether or not random moves were made, this is an on-policy control method.

Monte Carlo methods typically use a value of $\alpha = 1/N_{(s,a)}$, where $N_{(s,a)}$ is the number of times that the given state/action pair has been visited so far. This makes the resulting value function utilities a true average of all previously seen returns. Annealing α in general helps convergence towards the optimal policy. However, Monte Carlo also works with a fixed α parameter, and using the same α allows for fairer comparison with the other reinforcement learning methods used in this paper. Furthermore, since these agents will learn in a multiagent scenario, it makes more sense to never lower the learning rate, since agents may need to relearn to adjust to changing opponents. The α value used for this and all other methods in this paper was 0.01.

Because Monte Carlo updates are based purely on sample returns, the utilities they learn will tend to accurately reflect these samples, even if these samples do not accurately reflect the dynamics of the environment. This can easily be the case if state/action pairs that are rarely visited have high variance in utility. However, this reliance on samples can also be a boon if the state representation available to the agent is non-Markov.

A state representation that has the *Markov property* is one in which the probability of a given response from the environment depends only on the current state and action. A non-Markov state representation can depend on states and actions preceding the current state and action. It is therefore difficult to solve a non-Markov problem with reinforcement learning methods, which are primarily designed to solve Markov Decision Processes. When the real dynamics of the system are uncertain, it makes sense to only base utility estimates on the final outcomes of many samples, since no assumptions can be made about the environment in the middle of an episode. This is one advantage that Monte Carlo has of temporal difference methods, which are presented next.

2.2 Temporal Difference Methods

Temporal difference methods are based on bootstrapping of utility value estimates based on previous estimates. They have an advantage over Monte Carlo methods in that they can learn during the episode instead of just at the end. They are particularly well suited to situations where the Markov property holds, because they tend to find estimates that are accurate for the maximum-likelihood model of the Markov process presumed to be controlling the environment's dynamics.

A popular method for on-policy temporal difference learning is Sarsa, and a similarly popular method for off-policy temporal difference learning is Q-Learning.

Sarsa The Sarsa algorithm is so named because after every action it decides what its next action will be and then updates its utility function based on the values of s, a, r, s' and a', where s and a are the initial state and action, r is the reward received as a result of the action, s' is the state transitioned to, and a' is the action chosen in that state. The exact form of the update, in terms of time t, is:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha(r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t))$$

This is an on-policy method because the update is based on the action taken at the following time step. This means that Sarsa is only guaranteed to converge to an optimal policy if ϵ is annealed towards zero throughout the course of training, which is not the case in this paper. However, Sarsa can perform better than off-policy methods like Q-Learning because the risk of exploration is incorporated into the learned value function.

Q-Learning Q-Learning is an action selection method similar to Sarsa, except that instead of using the value of the next action it picks as part of its update target, it uses the value of the best action it could possibly pick from that state. This will usually be the same as the action that the agent actually performs, but because an ϵ-greedy action selection method is used, suboptimal actions will sometimes be chosen. This is why Q-Learning is an off-policy method. The exact form of its update rule is:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha(r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t))$$

By selecting the best action that could be selected, Q-Learning is updating its value function towards the optimal policy, regardless of what policy it actually follows. We would therefore expect the policy learned by Q-Learning to be better than the policy learned by Sarsa, but this does not mean that a Q-Learning agent performs better during training than a Sarsa agent. In fact, a Q-Learning agent may often make risky moves because it does not care if a subsequent exploratory move forces it to receive a low reward. From the agent's perspective, that exploratory move was not its fault, and has nothing to do with the policy it is learning.

2.3 Intermediate Methods

There also exist methods that blend aspects of Monte Carlo and temporal difference methods with the use of eligibility traces. Eligibility traces are a mechanism that allows rewards to be scattered across all states visited so far in an episode. This means that earlier states receive a portion of the reward for helping lead the agent to its current state. The result is that many utility values are updated on every action, which greatly speeds learning.

These methods make use of an additional parameter $0 \leq \lambda \leq 1$, where higher values of λ indicate that larger portions of the reward should be given to more distant state/action pairs, and smaller lambda values indicate the opposite. In fact, with a λ value of 0, these methods degenerate into pure temporal difference methods. Similarly, a λ value of 1 results in the same updates as Monte Carlo, with the fundamental difference that updates can now be performed during the episode as opposed to only at the end. The λ value used for the experiments in this paper was 0.8.

In order to implement eligibility traces, we associate with each state/action pair an additional value $e(s, a)$, which is the eligibility of the state/action pair. At the start of an episode, all eligibility values are set to zero. Whenever a given state/action pair is visited, its eligibility value is set to 1. This type of trace is a *replacing trace*, which contrasts to straight forward *accumulating traces*, which add 1 to the previous eligibility whenever a state is visited. Replacing traces are further complicated in that the eligibility of other actions available at the state are set to zero when the agent revisits a previously visited state. Both methods treat unvisited states the same way. The eligibility of all unvisited states is decayed by the term $\gamma\lambda$. These conditions result in the following update rule for eligibility traces:

$$e_t(s, a) = \begin{cases} 1, & \text{if } s = s_t \text{ and } a = a_t \\ 0, & \text{if } s = s_t \text{ and } a \neq a_t \\ \gamma\lambda e_{t-1}(s, a), & \text{if } s \neq s_t \end{cases}$$

Replacing traces were used instead of accumulating traces in this paper because the domain (described below) allows for frequent revisiting of the same state, which is generally a bad idea since rewards are discounted. However, accumulating traces would assign high eligibility to revisited state/action pairs, which would artificially heighten their utility values due to the manner in which eligibility traces are used to update the value function. The agent would think that going in circles was a good idea.

To use eligibility traces, one simply multiplies the normal update adjustment by the value of the eligibility trace, and then performs an update on all state/action pairs. Eligibility traces can be combined with any temporal difference method. When combined with Sarsa, the resulting update rule is:

$$Q_{t+1}(s, a) = Q_t(s, a) + \alpha\delta_t e_t(s, a), \text{ for all } s, a$$

where $\delta_t = (r_{t+1} + \gamma Q_t(s_{t+1}, a_{t+1}) - Q(s_t, a_t))$. This is known as Sarsa(λ).

For use with Q-Learning, the rule is the same, but $\delta_t = (r_{t+1} + \gamma \max_a Q_t(s_{t+1}, a) - Q(s_t, a_t))$. This presents a problem however, because it means Q-Learning is no longer an off-policy method. If this type of update is performed after a point when a random exploratory action has been made, then some of the updates are based on the exploratory action. Therefore, in order to maintain the off-policy nature of Q-Learning, the eligibility of all state/action pairs is reset to zero any time an action is chosen that does not have the same utility as the best action. Note that we need not reset the eligibilities simply because an exploratory action was made, since sometimes exploratory actions will randomly pick the best action anyway. This method is known as Watkin's Q(λ). It results in the learning agent being able to learn less per action on average than Sarsa(λ), but it still learns quicker than regular Q-Learning.

2.4 Linear Function Approximation

In their pure forms, all of the above methods use tables to represent their value functions, with one entry for every state/action pair. This is problematic when the state space is very large, because states must be visited many times before their utility values become accurate. This is especially the case in stochastic domains. What is needed is a way to generalize between states such that information learned in one state will improve the agent's performance in states that are similar but different.

This can be done by replacing the tabular value function with any one of many possible function approximators such as neural networks, decision trees, or linear function approximators. This paper uses linear function approximators to represent the value function in cases where the state space representation is too large for tabular methods to handle.

A linear function approximator is represented by a vector of randomly initialized weight parameters $\hat{\theta}$, one for each input in the input vector $\hat{\phi}$. The output of the linear function approximator is simply the weighted sum of the inputs:

$$\sum_{i=1}^{n} \theta(i)\phi(i)$$

where n is the number of inputs.

In order to use linear function approximators to represent the value function, we must first be able to derive a feature vector from any given state. How this is done in the domain used in this paper is described below. Next, because the value function determines the utility of state/action pairs, a separate linear function approximator is needed for each possible action. Therefore the number of weight parameters that needs to be learned is $n|\mathcal{A}|$, where $\mathcal{A}$ is the set of possible actions. This number can be large, but will generally be much smaller than the state space. Having set up the linear function approximators in this way, the utility of a given state is calculated as:

$$Q(s,a) = \sum_{i=1}^{n} \theta_a(i)\phi_s(i)$$

Given this method for representing the value function, any of the above reinforcement learning methods can be applied, except that instead of learning by updating table entries, every update must improve the effectiveness of the value function approximator by modifying its weights. This can be done using gradient descent, which attempts to reduce the error between the actual output and the desired output. In the case of linear function approximators, this is particularly easy because the gradient is equal to the input vector $\hat{\phi}$. Therefore, the update rule used with linear function approximators is:

$$\hat{\theta}_{(a,t+1)} = \hat{\theta}_{(a,t)} + \alpha(v_t - Q_t(s_t,a_t))\hat{\phi}_s$$

where the definition of the update target v_t depends on which reinforcement learning method is used. If Monte Carlo is used, then R_t is used for v_t. If Sarsa is used, then the target is $r_{t+1} + \gamma Q_t(s_{t+1}, a_{t+1})$. For Q-Learning, the target is $r_{t+1} + \gamma \max_a Q_t(s_{t+1}, a)$. This gives us linear approximation versions of Monte Carlo, Sarsa and Q-Learning.

However, it should be noted that there is a risk in using Q-Learning with a linear function approximator. When an off-policy method like Q-Learning is used in conjunction with a function approximator, there is a chance that the estimations of the function approximator will diverge away from the optimal value function. Despite this, it is possible for Q-Learning to work with a linear approximator, and therefore such a learner is implemented in this paper.

Incorporating Eligibility Traces Linear function approximators can also be combined with intermediate methods that make use of eligibility traces. For this to work, each eligibility trace is associated with a weight parameter instead of a particular state. The eligibility of each weight is then updated based on the gradient with respect to that weight, which for linear function approximators is simply the corresponding input value.

$$e_{(a,t)}(i) = \begin{cases} \phi(i), & \text{if } \phi(i) \neq 0 \\ \gamma\lambda e_{(a,t)}(i), & \text{if } \phi(i) = 0 \end{cases}$$

We then modify the update rule to be:

$$\hat{\theta}_{(a,t+1)} = \hat{\theta}_{(a,t)} + \alpha(v_t - Q_t(s_t,a_t))\hat{e}_{(a,t)}$$

where the value of v_t is defined as above for Sarsa and Q-Learning. When combining these methods in this way, the results are *gradient descent Sarsa(λ)* and *gradient descent Q(λ)*, though they will simply be referred to as linear Sarsa(λ) and linear Q(λ) in this paper. As with Watkin's Q(λ) above, the gradient descent version of Q-Learning with eligibility traces must also take special precautions to remain an off-policy method. Therefore gradient descent Q(λ) resets $\hat{e}$ to $\hat{0}$ whenever it chooses an action that is not optimal with respect to its current value function.

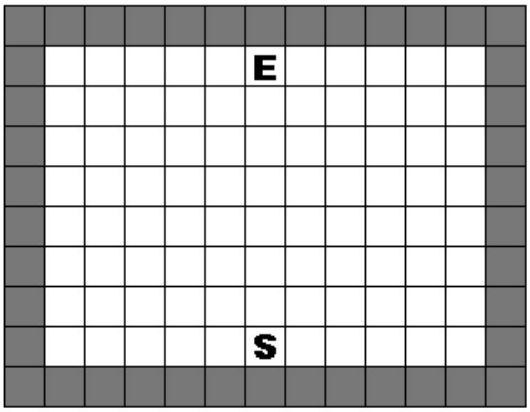

Figure 1: **Grid 0.** A generic grid surrounded by walls. There is a straight shot from start to exit provided predators do not get in the way, and there is plenty of room to maneuver, both of which make this grid fairly easy to escape.

Having fully described the reinforcement learning methods used in this paper, we now move on to a description of the predator-prey domain used.

3 Predator-Prey

The domain used in this paper is a simple set of predator-prey grid worlds. Many variations of the predator-prey domain have been implemented both with and without reinforcement learning. For a survey of many issues concerning how to define the predator-prey domain, and a collection of methods that have been previously applied to it, see Stone and Veloso 2000.

This paper uses three grids, each consisting of walls, starting points, and one exit. There is one prey that starts each episode at a random starting point, and there are three predators that start each episode in a random grid cell that is not a starting point. The goal of the prey is to reach the exit as quickly as possible without being eaten by the predators, and the goal of the predators is for any of them to eat the prey before it escapes. Only the prey is allowed to occupy the exit space and thus escape. A predator eats prey by occupying the same space as the prey after all agents have acted on the given time step. It is also allowable for two predators to occupy the same space.

Agents move simultaneously. After all agents have moved, the prey checks to see if it has escaped and each predator checks to see if it shares a space with the prey and can thus eat it. This allows for some risky maneuvers on the part of the prey. If the prey is one cell below a predator and moves up just as the predator moves down, then the two will switch

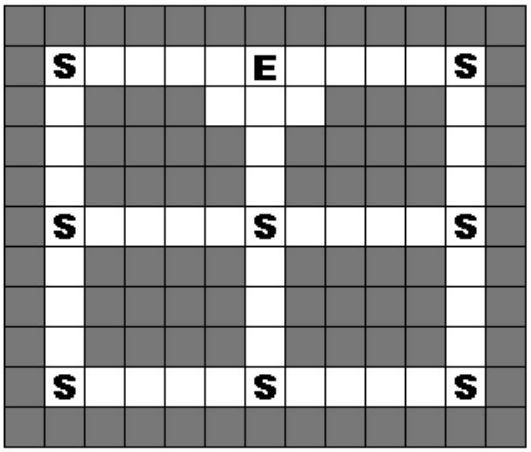

Figure 2: **Grid 1.** A maze with few open paths, but multiple starting points. Because of the narrow hallways, it is difficult, and in some cases impossible for the prey to escape. However, some starting points make it very easy to reach the goal.

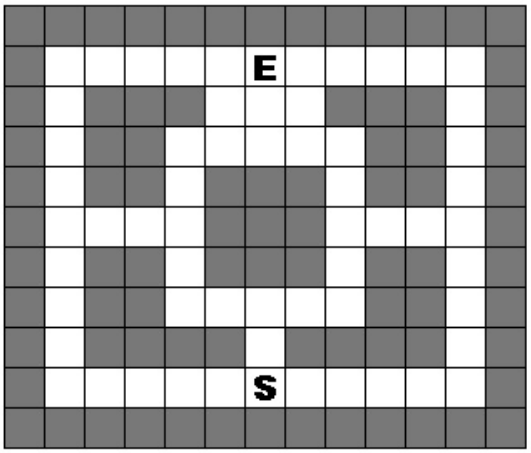

Figure 3: **Grid 2.** A more challenging maze with only one starting point. This maze is particularly difficult because there is not much room to maneuver, and the only starting point is distant from the exit.

places without the prey being eaten. No-op actions are not allowed. All agents must move up, down, left or right on each time step. However, if an agent attempts to move into a wall it will fail to move anywhere, as if it had executed a no-op action.

The rewards are as follows: +1 to the prey for escaping, -1 to the prey for being eaten, +1 to all predators if any of them eats the prey, -1 to all predators if the prey escapes. Because the task is discounted, it is in the best interest of the prey to escape quickly, and in the best interest of the predators to capture the prey as quickly as possible. The discounting also makes negative rewards less damaging if they occur at a later time step, which is to say that the prey's penalty for being eaten is less if it avoids being eaten for longer. Likewise, the predators suffer a lesser penalty if they prevent the prey from escaping for a long time than they would if the prey escaped quickly.

Of the grids used in this experiment, grid 0 (Figure 1) is very easy to escape from, because there is plenty of space to maneuver in, and because the path from start to exit is a straight shot. This world most directly resembles the type of world typically used in predator-prey studies. The maze worlds are more complicated and considerably more difficult to escape because at all but one of the hallway intersections the number of paths available is less than or equal to the number of predators. There exist situations in which it is impossible for the prey to escape because it is completely surrounded by predators.

In grid 1 (Figure 2), the prey's chances are improved by the variety of starting positions, some of which are very close to the goal. In grid 2 (Figure 3), the chances of escape are always slim. Still, despite an unfair world, the prey will still try to learn how to maximize its reward as best as it can, even though the odds of escape may be very slim.

Although the state and action spaces of this problem are discrete, they are very large. A naive state space representation consisting of the x and y coordinates of each agent in the world results in devastatingly slow learning, such that no learning seems to be occurring at all. This representation was provided to several tabular reinforcement learners acting as the prey against randomly moving predators in grid 1, and even after 100,000 episodes these agents were still encountering states which they had never encountered before.

Therefore better state space representations and value function estimation methods are needed in order to perform well in this domain.

4 State Space Representation

Two different state space representations are used in this paper. One is an partial state representation intended for tabular agents, and the second is a full representation intended for agents using linear function approximators as their value function.

4.1 Partial State Representation

Because a full representation of the state space results in a number of states that makes learning prohibitively slow for tabular agents, they are provided with a simple partial state representation (Figure 4). Partial state representations are not uncommon in predator-prey domains (Gomez and Miikkulainen 1997; Tan 1993), so we can expect that agents using this state space representation will perform competently despite being unaware of the full state.

An agent using the partial state representation is completely aware of predators, prey, starting points, the exit and walls within a square extending two cells from its location in all directions.

A matter of interest for this state representation is that agents do not sense themselves. In this scheme, the location of the sensing agent will never change, and thus does not need to be part of the state representation. As a result, predators are capable of sensing when another predator occupies the same space they do. Technically, prey can sense this as well, but

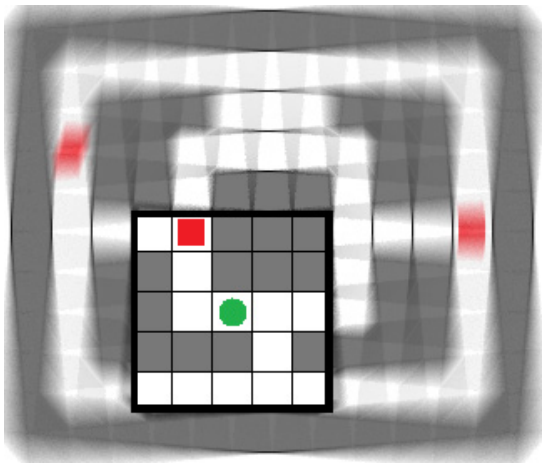

Figure 4: **Partial State Representation.** An example of partial state in grid 2. The prey (circle) can sense the walls and the predator that are within two cells of it, as indicated by the box surrounding it. The area outside this box is distorted to indicate that the prey does not sense it at all.

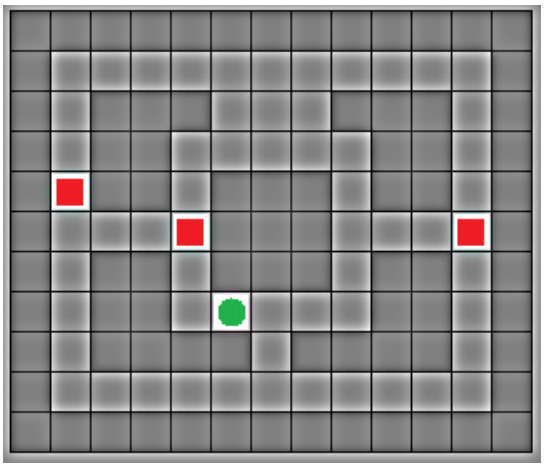

Figure 5: **Full State Representation.** An example of full state in grid 2. The full state ignores walls because their positions never change, and thus do not need to be incorporated into the state representation. Only predators and prey are included.

there is not much point since such a sensation would indicate that the episode was over.

When this state representation is applied to the domain, the domain becomes a Partially Observable Markov Decision Process (POMDP). This is a special form of non-Markov task in which the state representation is non-Markov but the underlying dynamics of the system are Markov. Since the state representation is non-Markov, we would expect pure temporal difference methods to have a hard time with this task.

4.2 Full State Representation

If more information about the environment is available, it is preferable to use it, especially if this allows the problem to have the Markov property. Given a state representation that has the Markov property, we will never forget pertinent information required to solve the problem optimally. The aim of the full state representation is to provide the Markov property at the cost of resorting to function approximators to represent the value function.

The feature vector used to represent the state consists of three values for each cell in the grid world. The first value is 1 if there is a predator present in the cell and 0 otherwise. The second value is 1 if the prey is present in that cell and 0 otherwise. The third value is 1 if the sensing agent occupies that cell and 0 otherwise. Because there is only one prey, this third set of values is really only needed by the predators so that they can distinguish between each other. However, the prey include this redundant information in their state representation as well.

Since there are three predators and only one prey, no more than five inputs will be active at a time (one for each agent, and one for the agent to sense itself), and therefore no more than five weights will ever be updated at once.

It should be noted that this representation completely ignores walls, starting points and the exit. All of these objects have the same position in all states, thus there is no need to include information about them in the state representation.

One more point worth being aware of is that despite its name, this state representation is technically neither full nor Markov due to the fact that multiple predators can occupy the same cell. If two predators are sharing a cell, then only two of the three predators will be sensed. In such a case it is impossible to tell which of the two sensed predator positions actually has two predators in it without looking at the history of past states. However, such a factor should not have a strong influence on the ability of agents to learn. Since the state representation is kept simpler by excluding this extra information, we are willing to accept a slightly non-Markov state representation.

5 Experimental Approach

Before having the reinforcement learning methods defined above compete in the described predator-prey domain, they are first tested against randomly moving agents to attain a baseline for the level of performance that can be expected of each method in this domain.

These methods are then placed in competition with each other in the predator-prey domain with one type of agent representing the prey and another type of agent representing each of the predators.

Detailed experiments in which the predators learn, but the prey does not, are not presented in this paper. Although this is the approach commonly used in most versions of the predator-prey domain, these versions usually do not allow the prey to escape. The goal of the predators is simply to catch the prey as quickly as possible. Although it is possible for the predators to learn to catch the prey more quickly, they are still almost guaranteed to catch the prey by chance eventually if it uses a random policy. There is no sense of failure for the predators.

In the domain used in this paper, the predators have failed if they allow the prey to escape. Although they can improve their overall reward by catching the prey quicker, they are

still considered to have done well if they catch the prey at all. By this measure, the predators are very likely to succeed because a randomly moving prey has little chance of finding the exit. Brief experiments were performed with all of the reinforcement learning methods used in this paper in which the predators learned and the prey moved randomly. In just 100 episodes, predators of all types were already catching the prey 86% of the time or more. Most methods were catching the prey around 95% of the time.

For the domain used in this paper, it is therefore trivially easy for the predators to succeed at capturing randomly moving prey. Hence forth this paper focuses instead on the cases of learning prey vs. randomly moving predators and learning prey vs. learning predators.

5.1 Learning to Escape Random Movers

All of the reinforcement learning methods mentioned above are evaluated in each grid world against randomly moving predators. For each method, there is one version that uses the partial state representation and represents its value function with a table, and there is another version that uses the full state representation and represents its value function with a set of linear function approximators. These learning agents are designated as tabular Monte Carlo (M), tabular Sarsa (S), tabular Q-Learning (Q), tabular Sarsa(λ) (SL), tabular Watkin's Q(λ) (QL), linear Monte Carlo (LM), linear Sarsa (LS), linear Q-Learning (LQ), linear Sarsa(λ) (LSL) and linear Watkin's Q(λ) (LQL).

Each method was evaluated as the prey once in each of the three grid worlds for 100,000 episodes against three randomly moving predators. As previously mentioned, the parameters used to control learning for all methods were $\alpha = 0.01$, $\gamma = 0.9$ and $\epsilon = 0.1$. For methods that used eligibility traces, $\lambda = 0.8$ was used.

5.2 Learning to Escape Other Learning Agents

Each of the ten types of learning agents were also pitted against each other in the predator-prey domain. One type of agent was used to control the prey and another type was used to control each of the predators. Many possible pairings exist between these ten types of agents, so we chose to focus only on the most pertinent ones.

The first set of comparisons is between tabular methods using a partial state representation and their counterparts using linear function approximators with a full state representation. This set of comparisons, with the prey type listed first in capital letters and the predator type listed second in lowercase, is made up of: M vs. lm, LM vs. m, Q vs. lq, LQ vs. q, S vs. ls, LS vs. s, QL vs. lql, LQL vs. ql, SL vs. lsl and LSL vs. sl.

The second set of tests is designed to compare Monte Carlo methods against temporal difference methods. These methods are split up so that tabular methods are compared against other tabular methods, and linear function approximation methods are compared against other linear methods. The first set of comparisons is made up of: M vs. s, S vs. m, M vs. q, Q vs. m, M vs. sl, SL vs. m, M vs. ql and QL vs. m. The second batch is made up of LM vs. ls, LS vs. lm, LM vs. lq, LQ vs. lm, LM vs. lsl, LSL vs. lm, LM vs. lql and LQL vs. lm.

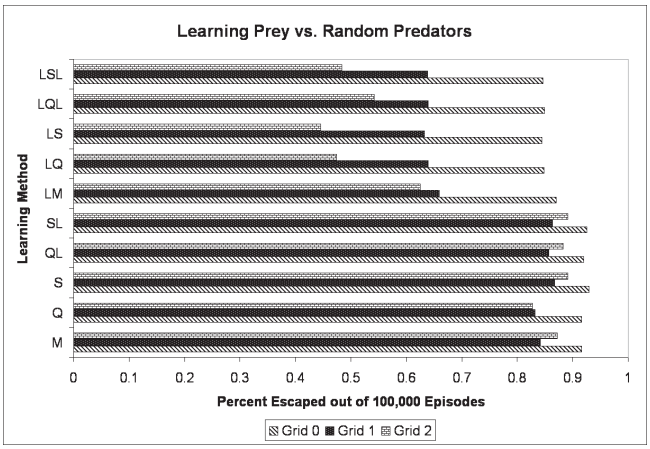

Figure 6: **Learning Prey vs. Random Predators.** This bar graph summarizes information about how often each method escaped the randomly moving predators.

As with the first set of experiments, each of these evaluations ran for 100,000 episodes with the same parameters listed above for learning against random movers.

6 Results

Results for prey learning against random opponents is presented first, followed by results for various learning methods in competition with each other. Each experiment is evaluated in terms of what percentage of the time the prey escaped and the average returns of the prey throughout the course of learning.

6.1 Learning to Escape Random Movers

Figure 6 summarizes results on how often each learning method escaped the randomly moving predators. The results for each grid are analyzed in detail under the headings below.

It is interesting that the tabular methods do better than the linear function approximation methods, especially on grids 1 and 2. These grid worlds have more intricate structures that allow the agents using the partial state representation to better identify their location despite having only partial information. This helps them reach the exit quickly despite the fact that the cramped quarters makes these grid worlds more difficult to escape.

The reason that the linear function approximation methods do not do as well on these grid worlds is likely that the limited number of weight parameters available to them to represent their value functions is not enough to generalize properly to these more difficult grid worlds. In grid 0, a good general strategy is simply to move up all the time. This will succeed in many cases, but sometimes it leads straight into a predator. In grid 1 the winning strategy is a bit more complicated because of the different starting points, but all shortest routes to the exit involve at most one direction switch, provided predators do not get in the way. This is less general that grid 0 however, so we see a drop in escape percentage. Meeting a predator is also more likely in this grid. The toughest grid is grid 2, in which all successful escape strategies require mul-

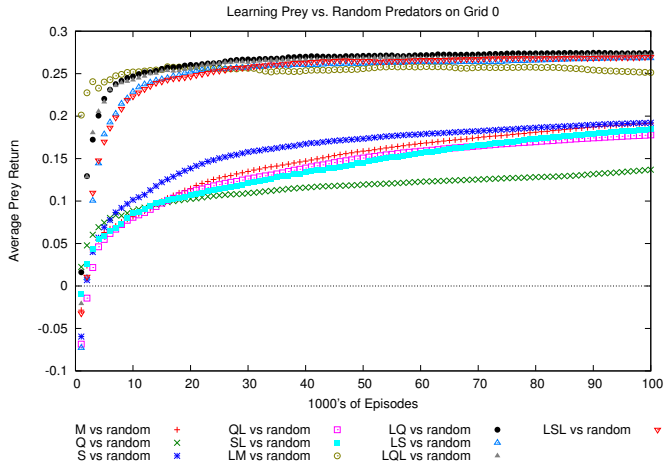

Figure 7: **Learning Prey vs. Random Predators in Grid 0.** All methods perform well in this grid, but some better than others. The average returns of the tabular methods are less than those achieved by the linear approximation methods, despite the fact that the tabular methods escaped from the grid a higher percentage of the time.

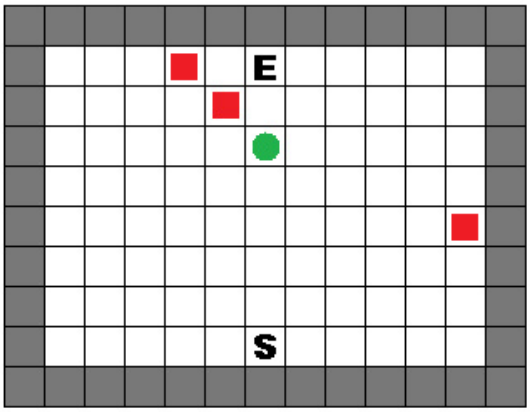

Figure 8: **Risky Policy in Grid 0.** If the prey moves up, there is a risk of being eaten, but an even higher chance of being able to exit soon. Despite the risk, the linear Sarsa agent chooses to move up.

tiple direction changes. The function approximation methods seem unable to generalize in this grid, though the linear Monte Carlo method does slightly better than the rest.

Escape percentage is not the only metric of success. In fact, the goal of the learning agents is actually to maximize their episode returns. We now analyze the performance of these methods on each grid in terms of average return throughout the course of training.

Grid 0 Figure 7 shows the average return of each agent in grid 0 over the course of 100,000 episodes. Because the exit is seven steps away from the starting point, the maximum attainable return in this world is $\gamma^6 \approx 0.53$.

Note that the linear methods achieve higher average returns than the tabular methods even though their escape percentages are lower. An examination of several sample runs from the end of training reveals why. All linear temporal difference methods, including the λ-based methods, which together are the set of methods that attain the highest average returns from this grid, tend to learn a simple policy that advises them to always move up, regardless of the dangers. This maximizes the return when it is successful, and against randomly moving opponents in this simple grid world, such a policy usually is successful. However, blindly moving forward has risks, and sometimes causes these linear temporal difference agents to be captured when a little bit of caution would have allowed them to escape.

To see an example of risky behavior on the part of a linear temporal difference agent, see figure 8. A linear Sarsa agent (circle) must choose an action. If it moves up there is a 25% chance the closest predator (square) will move right and capture it. However, it has a 75% chance of being right next to the exit for its next move. This move is risky not only because of the chance of being eaten on the first move, but also due to the risk of a random move following it. Since Sarsa is an on-policy method, it must take this risk into account. In this situation, the linear Sarsa agent trained in these experiments chooses to move up, which turns out to be the move that maximizes average return in the long run. Rather than avoid the risk in order to guarantee a more discounted reward, the agent takes a risk in hopes of getting the maximum reward possible.

Incidentally, the linear Monte Carlo agent learned a similar strategy, except that it seems slightly more inclined to make sideward moves. This is why its average returns are slightly lower than the other linear methods.

The tabular methods also use a strategy that leads them upwards as quickly as possible, but they are more willing to avoid predators. This is likely due to the fact that once they are out of sensing range of the walls, they have no way of knowing how far they are from the exit, since they use a partial state representation. From their perspective, the center region is a very unpredictable place. All they know is that if they keep going up, the exit will appear on the north edge with some low probability. In the meantime they avoid predators if they need to. Sometimes this can lead them astray, which is a big problem if they reach the north wall but do not see the exit. They could just as easily be left of the exit as right of it. If their value functions encourage them to go the wrong way, then they will likely waste a lot of time going in circles, even if they do eventually manage to escape. Such wasted time results in a highly discounted return.

Notice that the performance of tabular Q-Learning is slightly beneath that of tabular Sarsa, but runs parallel to it. This makes sense because Q-Learning is following a more dangerous policy than the one it is learning, since it is an off-policy method.

Grid 1 The average returns in grid 1 (Figure 9) are more in line with the escape percentages for this grid. Tabular Sarsa(λ), tabular Q(λ) and tabular Monte Carlo all end up tying for best at the end. Tabular Sarsa lags slightly behind, and tabular Q-Learning stays below Sarsa for most episodes, as in grid 0.

At a level of performance lower than these methods are bunched all of the linear function approximation methods.

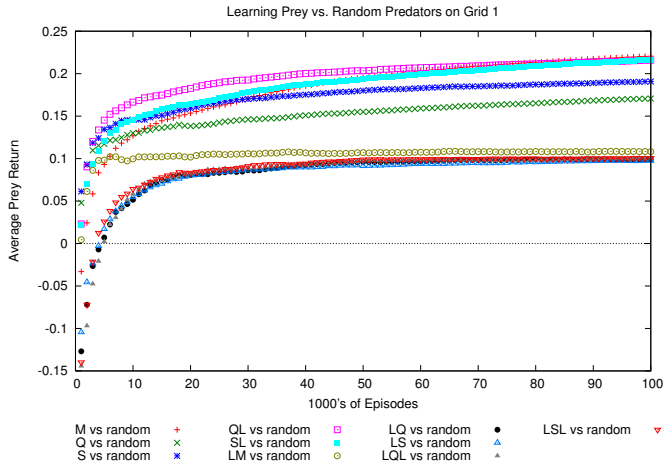

Figure 9: **Learning Prey vs. Random Predators in Grid 1.** The tabular methods perform exceptionally well, especially the λ-based methods and Monte Carlo. All tabular methods beat the methods using linear function approximation, which is in line with the escape percentages of each of these methods from this grid. Their performance is merely adequate, not exceptional.

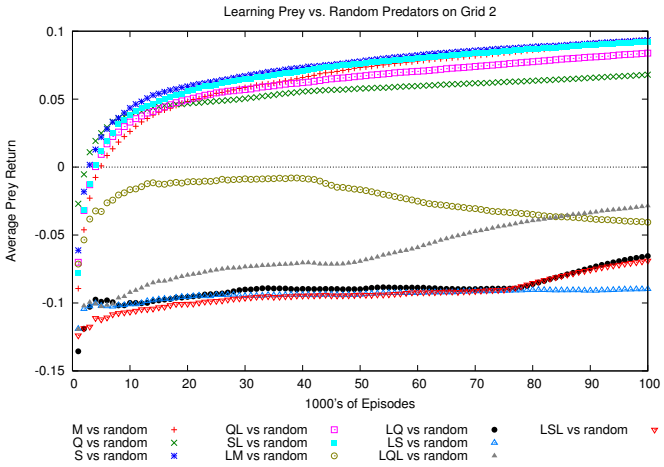

Figure 10: **Learning Prey vs. Random Predators in Grid 2.** Tabular methods do well as usual, but the linear approximation methods do quite poorly. They can only attain negative average returns.

Only the linear Monte Carlo method distinguishes itself as being slightly better than the other linear methods.

Looking at example episodes once again reveals that the linear agents favor risky policies. These methods take them directly to the exit in many cases, but the additional walls in this grid world lower the chances of the prey slipping by in many cases. The reason that the linear function approximation methods favor such risky policies is that such policies are more general, and therefore easier to represent using its limited weight parameters.

As before, the tabular methods show more caution. Example runs show that the tabular methods will actually wait two spaces away from intersections if a predator is lurking there. Once the predator is gone, they continue seeking the exit.

Grid 2 This grid proves particularly challenging for all agents (Figure 10). Only the tabular methods are able to achieve positive average returns, and these returns are small. However, the best return possible on this grid world is less that in the previous two. The most direct path from start to exit consists of 12 steps, making for a maximum possible return of $\gamma^{11} \approx 0.31$

The most shocking result in this grid world is the poor performance of all linear function approximation methods. However, the performance of some of them actually jumps near the end, which gives hope that they might attain positive average returns in the near future. In strange contrast, linear Monte Carlo, which starts out doing better than the other linear methods, experiences a sharp drop in performance. Results from other experiments done with these methods (not shown) on this grid show that the average return curves can be very different, likely due to random initialization of the starting weights and different exploratory moves executed during training.

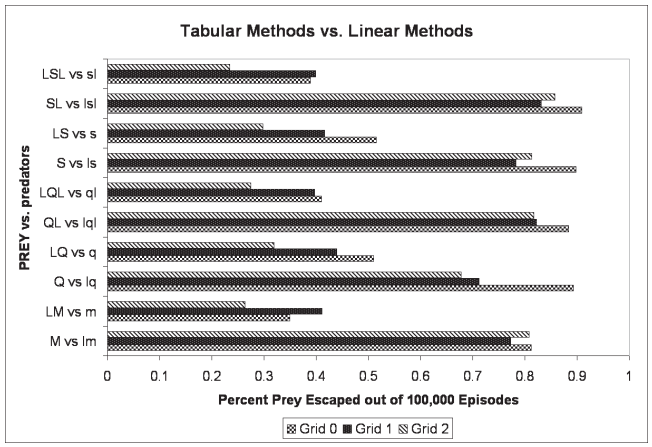

Figure 11: **Tabular Methods vs. Linear Methods.** Percent of the time that the learning prey (capital letters) escaped three learning predators (lower case). Comparisons are between tabular methods and their corresponding linear counterpart.

6.2 Learning to Escape Other Learning Agents

Having measured each learning method's performance against randomly moving predators, we now put each learning method in competition with sets of predators that also learn. Due to the unpredictable chance ways in which differing learning methods interact, it should be emphasized that these results are not conclusive. There are indications that the qualitative nature of the results may very well be a matter of chance in particular cases.

Figure 11 summarizes the escape percentages when comparing tabular methods using a partial state representation with their corresponding linear methods using the full state representation. Tabular methods do better against their linear counterparts in all grids. As prey they attain higher escape percentages than the corresponding linear method does against tabular predators.

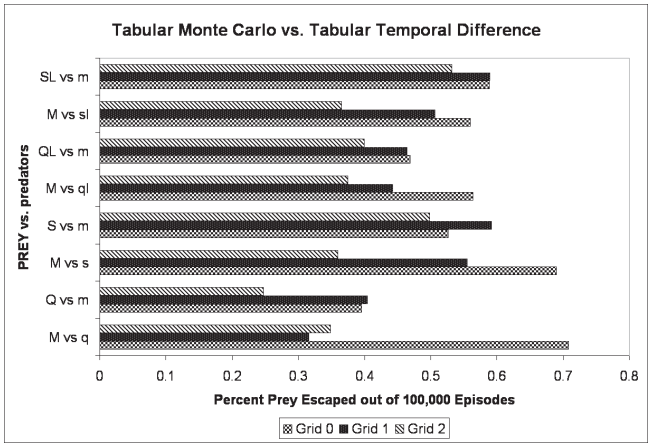

Figure 12: **Tabular Monte Carlo vs. Tabular Temporal Difference.** Tabular Monte Carlo is both the prey (M) and a set of predators (m) against the tabular versions of Sarsa, Q-Learning, Sarsa(λ) and Q(λ).

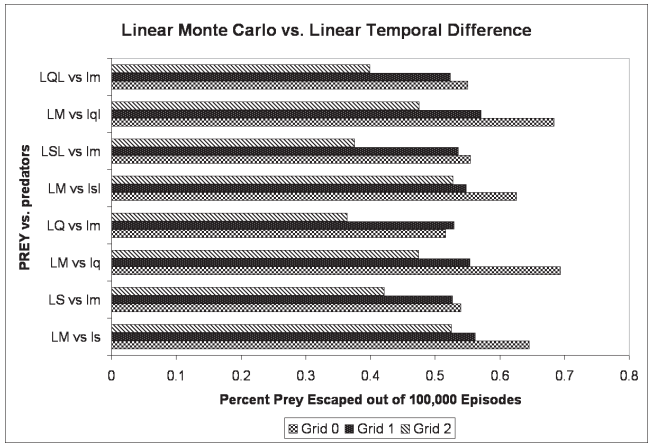

Figure 13: **Linear Monte Carlo vs. Linear Temporal Difference.** Linear Monte Carlo as both prey (capital) and predator (lowercase) against each temporal difference method. Monte Carlo tends to do slightly better than temporal difference in all cases.

The next table (Figure 12) shows how tabular Monte Carlo compares against tabular temporal difference methods. This set of results is particularly interesting because it compares the methods that did the best in previous experiments against each other.

Tabular Sarsa(λ) prey consistently escapes just over half the time from tabular Monte Carlo predators in all three grid worlds. This is especially impressive for grid 2. Monte Carlo prey only performs competently against tabular λ-based methods. Its ability to escape degrades with the difficulty of the grid.

When Monte Carlo is the prey against plain Sarsa and Q-Learning predators, it does well in the open spaces of grid 0, but has considerable trouble in the cramped corridors of grids 1 and 2. As prey against Monte Carlo predators, Sarsa performs moderately well, but Q-Learning is mediocre, likely due to its off-policy nature.

The final set of comparisons is between linear Monte Carlo and linear temporal difference methods (Figure 13). These results show that linear Monte Carlo does consistently better as prey against linear temporal difference methods than they do against it, but not by much. The margin is small, but present against all tested opponents.

We now turn to examine the results of each grid world in more detail.

Grid 0 Figure14 shows how tabular methods do against linear methods in grid 0 in terms of average return. Tabular methods are the clear victors in all comparisons.

The downward curves all correspond to a linear method as the prey vs. its better performing tabular counterpart. Linear Monte Carlo in particular is overwhelmed by Monte Carlo based predators. Downward curves like this indicate that the amount of information learned by the predators per episode far outstrips the amount of information learned per episode by the prey. The tabular agents also perform well as prey vs. their linear counterparts for the same reason. They simply learn more in a shorter period of time.

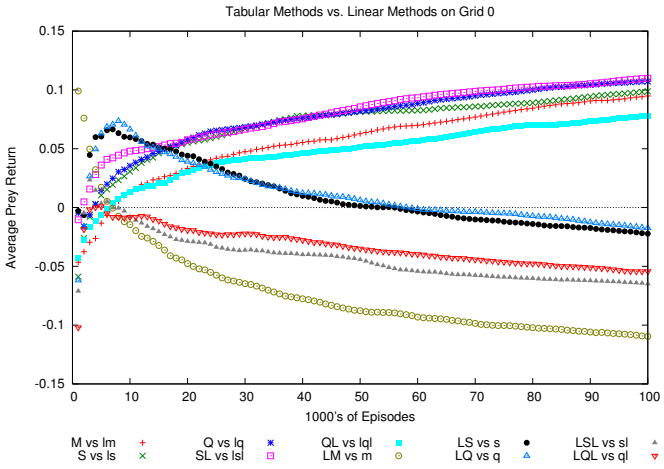

Figure 14: **Tabular Methods vs. Linear Methods in Grid 0.** Each line plots the average return of a given prey agent (capital letters) against the corresponding group of three predator agents (lowercase).

Inspection of actual episodes in this grid reveals an interesting predator strategy that sometimes develops among the tabular agents. Sometimes one of the three predators learned to stay near the exit if it found it. This predator would guard the exit while the other agents tried to find the prey. This makes sense because tabular agents only had access to a partial state representation, which makes them uncertain as to the location of the prey. However, the predators learned that the prey has an inclination towards going to the exit, so if they find that, then they might as well wait there for the prey. This strategy works because all predators are awarded equally if any of them catch the prey. Therefore one of the agents does not mind being on guard duty. It would be interesting to see what would happen if predators were only rewarded for their own captures.

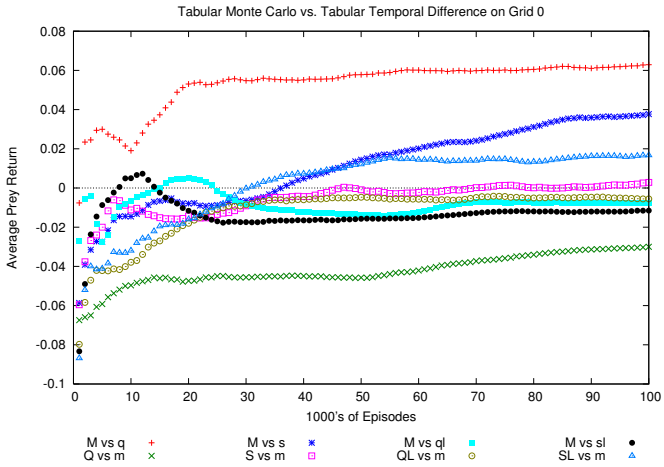

Figure 15: **Tabular Monte Carlo vs. Tabular Temporal Difference in Grid 0.** Tabular Monte Carlo is compared to all other tabular methods, including the λ-based ones, which are its major competitors in the previous experiments.

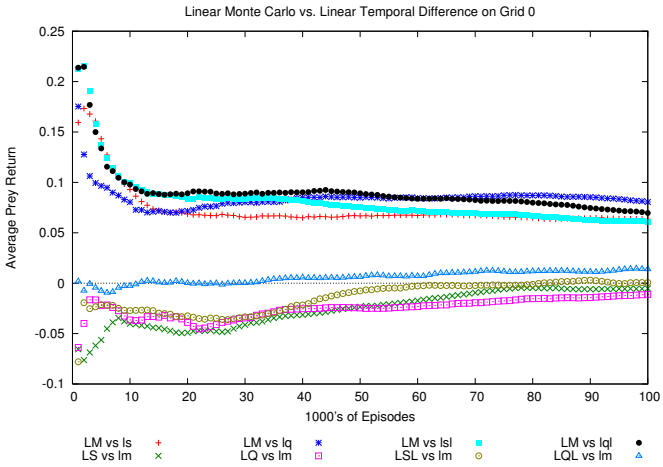

Figure 16: **Linear Monte Carlo vs. Linear Temporal Difference in Grid 0.** Monte Carlo quickly learns to seize high returns as the prey, but the temporal difference methods catch on quickly and bring its average returns lower, though they stay positive.

Next we take a closer look at how tabular Monte Carlo fared vs. tabular temporal difference in grid 0 (Figure 15). Recall that in terms of escape percentages, Monte Carlo had an advantage in grid 0, especially against the plain temporal difference methods. This is evident in the curves of the graph. The only agent that Monte Carlo does not have an advantage over in grid 0 is tabular Sarsa(λ), which actually achieves positive average returns as prey against Monte Carlo predators.

Note that as prey, Monte Carlo gets large returns against Q-Learning, and as predators against Q-Learning they prevent the prey from getting good returns. Both of these results are likely due to Q-Learning's off-policy nature. However, even though these curves represent the high and low points on the graph, its scales is still only from -0.1 to 0.08. This is a very small range, and it indicates that the various tabular methods do a fairly good job of canceling each other out. This is further evidenced by the fact that all of the curves in this graph not yet mentioned seem to converge to zero.

Next we compare the linear versions of Monte Carlo and temporal difference. Figure 16 shows the pertinent data.

Monte Carlo prey quickly learns how to exploit the environment early on, but soon afterwards each of the temporal difference methods finds a way to drop Monte Carlo's average returns back down, but not enough to make them negative. Linear Monte Carlo is able to maintain reasonably good average returns against all opponents.

When the temporal difference methods are the prey, they have difficulty against Monte Carlo. Only the λ-based methods are able to break into positive average returns, but not by very much. Still, it makes sense that λ-based methods would learn faster, since on average they are likely updating more weight values at once.

Having analyzed the results from grid 0, we no move on to grid 1.

Grid 1 Figure 17 compares tabular and linear methods in grid 1. These results tell the same story as the results for

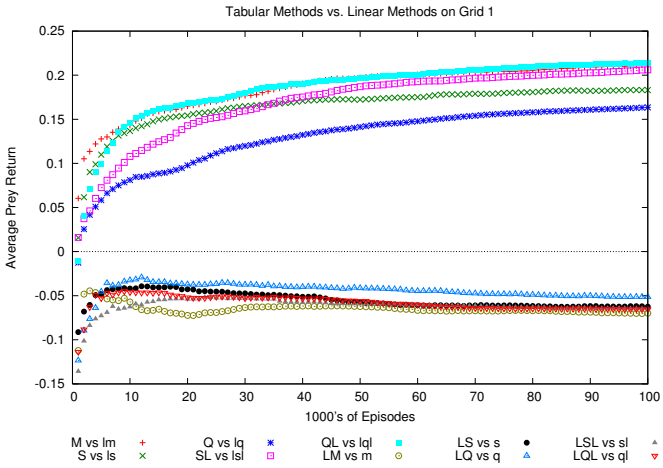

Figure 17: **Tabular Methods vs. Linear Methods in Grid 1.** Comparison of each tabular method using a partial state representation with its corresponding linear approximation method using a full state representation in grid 1.

grid 0, namely that tabular methods beat linear methods, except that now the differences in performance are more pronounced.

The reason that the tabular methods do even better than before in this environment is likely that the structure of the walls provides more landmarks by which an agent using the partial state representation can actually tell its location. The layout of walls in grid 1 is actually such that an agent using the partial state representation can always tell where it is.

The next figure (Figure 18) shows the results of comparing tabular Monte Carlo and temporal difference methods in grid 1. Some of the results on this grid are quite different from what happened on grid 0.

On this grid, all of the temporal difference methods main-

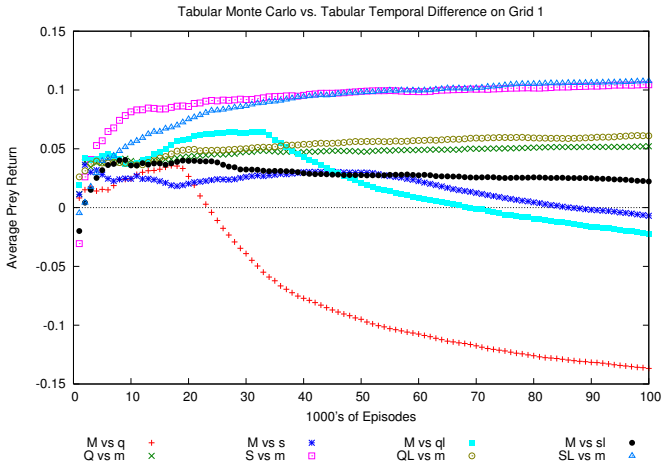

Figure 18: **Tabular Monte Carlo vs. Tabular Temporal Difference in Grid 1.** The comparisons between tabular Monte Carlo and temporal difference in grid 1 differ in important ways from the same graph for grid 0.

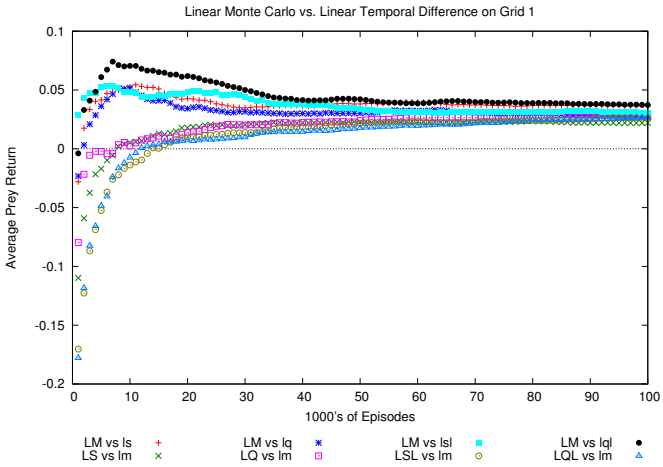

Figure 19: **Linear Monte Carlo vs. Linear Temporal Difference in Grid 1.** All average returns converge to the same point.

tain positive average returns against Monte Carlo predators. This was not the case in grid 0, and this is likely due to the relative degree to which these grids are non-Markov. Given the partial state representation, any agent will forget about any agents that leave its field of view, regardless of which grid is used. However, grid 0 is additionally complicated by the fact that an agent will forget is location in the world when it leaves sight of all walls. In contrast, the wall layout of grid 1 makes it possible for an agent using the partial state representation to uniquely identify all positions in that grid world. In this sense, grid 0 is *less Markov* than grid 1. The temporal difference methods slightly depend on the environment being Markov, whereas Monte Carlo does not depend much on this assumption at all. Therefore, Monte Carlo was able to do better as predators in the less Markov environment (grid 0) than in the more Markov environment (grid 1). Another way to look at it is that the temporal difference methods were able to take advantage of the slightly Markovian environment of grid 1.

It is interesting that Monte Carlo prey starts off doing fairly well against all types of temporal difference method predators, but that against each one there is a point where the average returns suddenly drop. The only exception is Sarsa(λ), though there does not seem to be any obvious reason for this. It seems that in this grid, all temporal difference agents have a reasonable shot at learning the trick that enables them to defeat Monte Carlo prey, but that for this series of episodes, Sarsa(λ) just has not succeeded at doing so yet.

The final set of results for this grid show how linear Monte Carlo compares with the other linear methods (Figure 19). On this grid, all average prey returns converge to the same small positive value, regardless of whether the prey was the Monte Carlo agent or a temporal difference agent.

The only big difference between these curves is whether they curve up or down. For all of the curves where linear Monte Carlo is the prey, the curve starts high and slowly goes down. When a temporal difference agent is the prey, the curves start in the negative and then rise up. This once again implies that linear Monte Carlo is able to quickly learn a strategy that is fairly useful until the temporal difference predators learn how to thwart it.

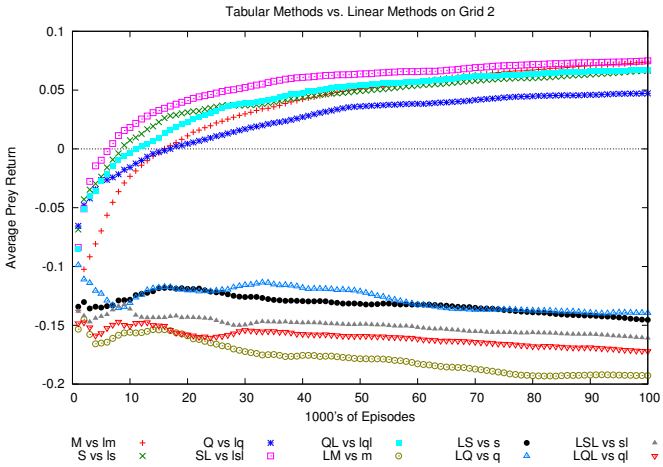

Figure 20: **Tabular Methods vs. Linear Methods in Grid 2.** Results similar to those for grid 1.

Grid 2 The results comparing tabular methods and their linear counterparts in grid 2 (Figure 20) are similar to the results in grid 1, except that all plots are shifted downward to reflect the extra challenge of this grid.

In this grid, the linear methods actually have noticeable downward slopes in their average returns, which indicate that not only are they not doing well, but they are getting worse.

The next figure (Figure 21) shows the comparison of between tabular Monte Carlo and tabular temporal difference on this grid. No type of prey is able to obtain a positive average return, and the qualitative form of the plots is very different from some of the plots in earlier grids.

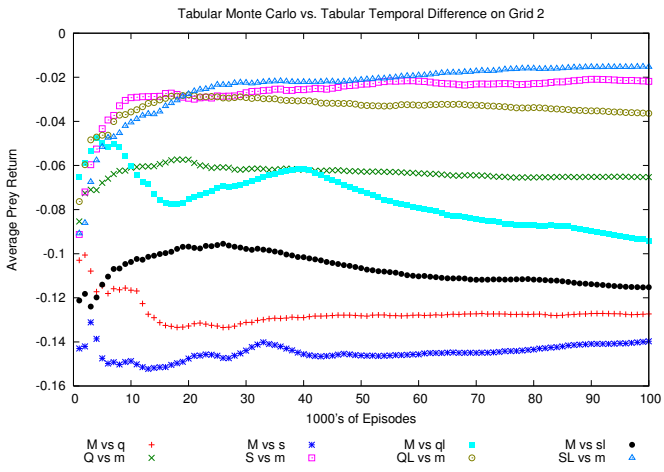

Figure 21: **Tabular Monte Carlo vs. Tabular Temporal Difference in Grid 2.** None of the prey in these experiments were able to attain positive average returns.

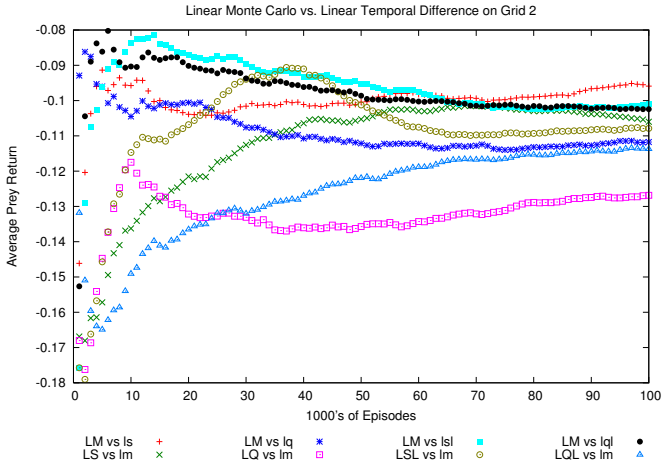

Figure 22: **Linear Monte Carlo vs. Linear Temporal Difference in Grid 2.** As with the tabular comparisons, none of the prey are able to achieve positive average returns.

The plot for tabular Monte Carlo prey against tabular Q(λ) predators is very strange. First it spikes upward in favor of Monte Carlo, and then it spikes down and back up again before settling into a steady drop. Several other curves in this graph tend to zigzag as well.

Although all average returns are negative, it is still clear that the temporal difference methods are doing better than Monte Carlo.

The final figure 22 shows how linear Monte Carlo compares with linear temporal difference. As with the tabular comparisons, none of the prey achieve positive average returns, which enhances the likelihood that such low performance is more due to the difficulty of this grid rather than any weakness in the learning methods used.

These curves also zigzag, as with the tabular results, which indicates that there must be an unusual space of strategies in this grid that tend to gain rewards quickly, but that tend to be easy for predators to learn how to defend against.

Although these curves do not converge nicely as in grid 1, there is still a general trend of linear Monte Carlo prey starting high and dropping slightly, and temporal difference methods starting low and rising up.

7 Discussion

The major victors in most of the experiments in this paper are the tabular methods, which make use of a partial state representation. It is interesting that they tend to do well in all environments, and also that in grid 0 they do not do as well as linear function approximation methods using a full state representation.

As has already been mentioned, when the partial state representation is used, grids 1 and 2 are more Markov than grid 0 since it is at least possible for an agent to know its absolute location in grids 1 and 2. Use of Monte Carlo and λ-based methods helps deal with partial state to some extent, but it is still better to have a state that is more Markov.

However, the linear function approximation methods had access to the full state in all grids, yet did not perform as well in grids 1 and 2 as the best tabular methods. Therefore the shortcomings of these methods must have come from their inability to represent good policies with their limited weight parameters. This suggests that when the complexity of an environment is large enough that a full state representation would overwhelm a tabular agent, it is not necessarily the best idea to switch to a function approximation based agent. It may actually be better to switch the state representation than to switch the value function estimation method. Being able to acurrately represent the value function for a small partial state representation can be better than inaccurately representing the value function for a larger, more informative state space.

However, in hindsight it is not clear that a linear function approximation based agent cannot do as well as a tabular agent in this domain, because there is actually a shortcoming in the state representation used by the linear function approximators.

To understand what this shortcoming is, we first consider the issue of why the linear methods tend to learn such risky policies. First consider the fact that in grid 0 the linear methods learned to go up all of the time. This makes sense because there is a weight associated with the agent being at every position from the start to the exit. As the agent gets rewards for exiting, these weight values increase, regardless of the positions of the predators in the grid. The prey can follow the same trajectory multiple times, and the same set of weights corresponding to its position will be updated each time, even if for each trajectory the predators are at completely different positions each time. This allows the agent to generalize its decisions based on its own location, regardless of where the predators are.

The problem with this is that an agent should not go up if there is an predator in its path. However, given the full state representation defined for this paper, it is impossible for the agent to realize that the cell corresponding to its position has a relation to the cell containing the nearby predator. The inputs it receives are linearly independent, and thus cannot be

combined into the needed representation by a linear function approximator.

This could be fixed by adding a set of input features that are activated when any two of the current features are present. This makes for a lot of combinations, which complicates the function approximator by greatly increasing the number of weights it has to learn. Another option would be to use a neural network or some other method as the function approximator.

However, even if these methods worked, they might not do much better than the tabular methods using a partial state representation. This indicates that it might sometimes be a good idea to give up some state and attack a problem with a tabular method before jumping straight to more complicated function approximators to estimate the value function. Tabular methods are easy to apply, so there is little effort in at least trying a tabular method first before resorting to function approximation, which can always be done later if the tabular method fails.

Of course, the predator-prey domain of this experiment is merely a toy domain, which may be the real reason that the tabular methods outperformed the linear function approximation based methods. This approach would need to be tried on a more complex problem before it could be taken seriously.

8 Conclusion

In this paper, several reinforcement learning methods were compared in a predator-prey domain. Methods were compared both in terms of how well they could escape randomly moving predators when they controlled a prey agent, and in terms of how well they did against predators that also learned. The tabular methods utilized a partial state representation, and the linear function approximation methods utilized a full state representation.

Both methods represent a trade-off. The tabular methods more accurately estimate the values of states they can recognize, but they cannot distinguish between all states because of their partial state representation. The linear function approximation methods have access to the full state of the world, but they also treat multiple states in a similar fashion because they do not have enough weight parameters to encode information about individual states. They generalize, which is both an advantage and a disadvantage.

For the predator-prey domain presented in this experiment, tabular methods using a partial state representation performed the best. Among these it is not entirely clear which was the best, though the λ-based methods have obvious advantages over both plain temporal difference and Monte Carlo, since they combine the best features of both.

Although predator-prey is a toy domain, the results of this paper suggest that in some situations it may be worthwhile to attempt solving a complex problem with a simple method before trying a complex method. If the state space can be simplified in a way that still contains enough pertinent information, then a tabular method is more reliable than a function approximation based method, and is also easier to implement.

References

Gomez, F., and Miikkulainen, R. (1997). Incremental Evolution of Complex General Behavior. *Adaptive Behavior*, 5:317–342. Also Available from http://nn.cs.utexas.edu/.

Greenwald, A., and Hall, K. (2003). Correlated-q learning. In *Proceedings of the Twentieth International Conference on Machine Learning*, 242–249.

Littman, M. L. (2001). Friend-or-foe Q-learning in general-sum games. In *Proc. 18th International Conf. on Machine Learning*, 322–328. Morgan Kaufmann, San Francisco, CA.

Stone, P., and Veloso, M. M. (2000). Multiagent systems: A survey from a machine learning perspective. *Autonomous Robots*, 8(3):345–383.

Sutton, R. S., and Barto, A. G. (1998). *Reinforcement Learning: An Introduction (Adaptive Computation and Machine Learning)*. The MIT Press.

Tan, M. (1993). Multi-agent reinforcement learning: Independent vs. cooperative agents.

Weinberg, M., and Rosenschein, J. S. (2004). Best-response multiagent learning in non-stationary environments. In *AAMAS '04: Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems*, 506–513. Washington, DC, USA: IEEE Computer Society.