Studying the Impact of Domain Ergodicity on Efficiency of Reinforcement Learning and Training through Self-Play

Reza Mahjourian
Department of Computer Science
University of Texas at Austin
reza@cs.utexas.edu

Prateek Maheshwari
Department of Computer Science
University of Texas at Austin
prateek@cs.utexas.edu

1. INTRODUCTION

In the paper titled ”Why did TD-Gammon Work?” [6] Pollack attributes the success of TD-Gammon to “the setup of co-evolutionary self-play biased by the dynamics of backgammon.” Pollack used a hill climbing solution to achieve some of the success of TD-Gammon. Tesauro wrote a follow-up [11] and argued that the difference between the performance of TD-Gammon against the Pubeval program (56% win) and that of Pollack’s solution (40-45% win) is significant and compares to the difference between a world class player and an average player. Nonetheless, the success of TD-Gammon [10] has not been repeated in other domains.

In his paper, Pollack does not discuss exactly what aspects of the dynamics of backgammon are making self-play and reinforcement learning work. Our hypothesis is that in part it might have something to do with the partially ergodic nature of backgammon. In this project, we attempt to evaluate this hypothesis.

If the MDP is observed for an infinitely long period of time, a *recurrent* state is one that gets visited infinitely many times. On the other hand, a *transient* state is one which will be visited a finite number of times in an infinite viewing of the MDP. An *ergodic* MDP is an MDP in which every state is recurrent. In other words, in an ergodic MDP every state can be reached from every other state through some transitions. Therefore, if an agent is making decisions in an ergodic MDP, no current choices can cause it to be limited to a restricted region of the state space in the MDP.

Backgammon is not a completely ergodic game, as there are transient states in the game, for example in the endgame. But, we can also use ergodicity in a looser sense and consider a *degree of ergodicity* by comparing the number of recurrent states to the number of transient ones. One can also consider a *degree of recurrency* for a state, based on the expected likelihood of visitations to that state. With these notions, one can say that backgammon exhibits a higher degree of ergodicity and has states that have higher degrees of recurrency, compared to some other games.

As an example, in the game of chess, once the first piece is captured, none of the board configurations which include all the pieces on the board can be visited anymore. In chess, most middle-game states are highly transient. Therefore we could say that the game is less ergodic.

The game of backgammon consists of two phases. In the first phase, the players try to move all their checkers past the opponent’s checkers. Since the checkers for the two players move in opposite directions, there is a high likelihood that the players hit one another’s checkers. If a checker is hit, it has to enter the board again. Once a player has moved all their checkers to the last quadrant on the board, the second phase of the game–the race–begins, where the player can take their checkers off the board. The first player to finish wins the game.

Since no checker exits the game in the first phase, all 15 checkers from each player are always going to be in play. This makes most middle-game positions in backgammon recurrent states, as they can be repeatedly visited in the same game.

One interesting fact in that may support this hypothesis is that TD-Gammon was pretty strong in middle-game positions and quite weak in the endgame, which does not have recurrent states. Our hypothesis is that the higher degree of ergodicity helps TD-Gammon in two ways:

1. It helps with self-play, because in any given game the players are likely to visit many middle-game positions. This is caused by the constant hitting and re-entering. Consider a solution that uses an evolutionary method to evolve a backgammon agent using self-play. When a challenger is matched up against a champion, the challenger will need to have a broad strength on many middle-game positions in order to beat the champion. If the challenger makes slightly better moves in some positions and really bad moves in some others, it is going to have a harder time beating the champion in backgammon. This is because the ergodic nature of the game makes it more likely that those weak positions are visited in any single game.

   Moreover, no matter what opening moves are taken, most middle-game positions can still be encountered.

So, this rules out a common problem with self-play and co-evolutionary setups, which is developing strategies that cover only a restricted part of the state space. Usually, such suboptimal strategies can be developed due to some bias created early on for choosing a specific series of opening moves.

2. It helps with reinforcement learning, because the high degree of recurrency in the states makes them roughly equally likely to be encountered in the games in the long run. So, the amount of experience that the agent gathers for updating the state values would be balanced more evenly among the game states.

2. PROBLEM DOMAINS

To study the effect of ergodicity on learning, we designed and used two problem domains that can potentially show varying degrees of ergodicity.

2.1 mini-gammon

We designed mini-gammon in an attempt to create a simplified version of backgammon that would be easier to study. The board of mini-gammon is shown in Figure 1.

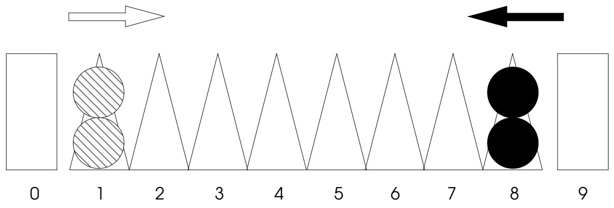

Figure 1: The mini-gammon Board

The board consists of one row with 8 cells (points in backgammon speak.) Each player has two checkers starting at the shown positions. The players take turns and move the checkers in the shown directions. During each turn, one player tosses a coin whose two sides read 1 and 2. The player then picks one of the two checkers and moves it ahead 1 or 2 cells, depending on the outcome of the coin toss.

Each point can be occupied by one or two checkers belonging to the same player. When a player has two checkers in the same cell, they form a block. Checker's in a block can not be hit by the opponent. This is while if the player has a single checker in a cell, the opponent can hit that checker by moving a checker of their own to the same cell. Once a checker has been hit, it is taken off the board and placed in the rectangular area on the player's side. This area is called the bar in backgammon.

If a player has a checker on the bar, they have to play that checker on the next turn. Depending on the outcome of the coin toss, the hit checker will enter on the first or second cell on the player's side. If the player has another checker in the target cell, they will form a block. If the opponent has a single checker in the target cell, the player hits the opponent's checker by entering the board. If the opponent has a block in the target cell, the player can not enter the checker and has to forfeit their turn. If a player has no legal moves, the turn will change and the other player continues by tossing the coin.

Once a player has both checkers on the second half of the board (on the opponent's side,) they can start taking checkers off the board by moving them past the last cell on the opponent's side. This is called bearing checkers off. The player who takes both checkers off first wins the game.

The board and the rules have been designed to limit the number of possible states so that a tabular temporal difference (TD) learning method can be used for a baseline reinforcement learning agent.

Requiring that the hit checkers enter from the end of the board, as in backgammon, increases the chance for conflicts on the board and the likelihood that configurations are repeatedly visited during games. This corresponds to a higher degree of ergodicity in the domain. To vary the ergodicity of the domain, we considered moving the designated reentry position forward towards the opponent. For example, considering a reentry offset of 1 would mean that by tossing a 1 the player can place their checker on the 2-point and by tossing a 2 they can place it on the 3-point. If the reentry offset is higher, then the checkers that reenter the board would be less likely to have the opportunity to engage the opponent checkers.

We also considered using two designated reentry offsets of 0 and 4 and using a probabilistic process to determine one of these two reentry points on each reentry. A reentry offset of 0 corresponds to entering from the end of the board and a reentry offset of 4 corresponds to entering from the middle point of the board. A parameter p that stays constant during the whole game controls the likelihood that either of the reentry offsets are selected. If $p = 1.0$, then the checkers always reenter from the end of the board and if $p = 0.0$ then the checkers always reenter from the middle of the board. Choosing other values for p would give us a spectrum of possibilities. Higher values of p should correspond to a higher degree of ergodicity in the game, as it increases the degree of recurrency of those states that include checkers on the first half of the board.

2.2 Nannon

The second problem domain that we use is the backgammon-like game called Nannon$^{\text{TM}}$ designed by Pollack [5]. The board of Nannon is shown in Figure 2.

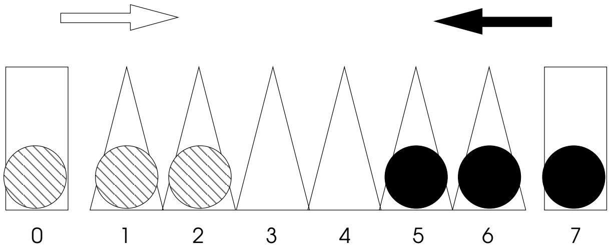

Figure 2: The Nannon Board

In Nannon, a single six-sided die is used to control checker

Offset	Avg. # of plies per game	# of unique states visited		Avg. # of visits to individual states		Var. # of visits to states
	over 1000 games	over 1000 games	per 1000 plies	over 1000 games	per 1000 plies	over 1000 games
0	59.26	1809	30.53	32.76	0.55	2104.44
1	40.30	1487	36.90	27.10	0.67	2006.41
2	30.55	1208	39.54	25.29	0.83	2304.85
3	24.77	986	39.80	25.12	1.01	2135.88
4	20.48	911	44.48	22.48	1.10	2210.65

Table 1: Effects of varying the reentry offset on dynamics of the game in mini-gammon

movements. Each player has 3 checkers and they start at the positions shown in Figure 2. Two checkers start on the player's 1-point and 2-point and the third checker is on the bar. Each point can be occupied by at most one checker at any given time. To simulate blocking, Nannon has the rule that any two checkers that are adjacent to each other on the board (not including the bar) form a block. Checkers in a block can not be hit by the opponent. Nannon's rules are a little more relaxed compared to backgammon. For example a player does not have to play a checker that is hit on the next turn. Also, players can bear off checkers at any time regardless of the positions of the other checkers.

To reduce the advantage for the player who starts the game in Nannon, the outcome of the first roll is decided by subtracting the outcome of two die rolls. More specifically, this makes rolling a 1, which would be a bad roll more likely than rolling a 5, which would be a good roll to start the game with.

We considered using the same two parameters of reentry offset and p that were introduced for mini-gammon to vary the dynamics of Nannon with the intention of controlling the degree of ergodicity in the game.

2.3 Some Statistics on the Problem Domains

Before implementing the learning algorithms, we collected some statistics on these two problem domains to gain some insight on the impact of varying the parameter values on the behavior of the domains.

For each problem domain, we ran 1000 games between two agents which selected moves randomly. Tables 1 and 2 show some statistics on varying the reentry offset in mini-gammon and Nannon, respectively. As the numbers indicate, lower values for offset lead to longer games on average in both domains. Lower values for offset also increase the total number of board configurations visited in these 1000 games.

One particular piece of statistics that was of interest to us is the variance of the number of visitations to individual states. We thought a lower variance would be desirable since it would help balance the amount of experience that a TD-based learner would gain on different configurations in the game. The data in the tables suggest that there is not a great amount of difference in the variance values for different reentry offsets.

Offset	Avg. # of plies per game	# of unique states visited		Avg. # of visits to individual states		Var. # of visits to states
	over 1000 games	over 1000 games	per 1000 plies	over 1000 games	per 1000 plies	over 1000 games
0	13.05	1742	133.49	7.49	0.57	695.79
1	11.47	1670	145.60	6.87	0.60	922.54
2	10.43	1613	154.69	6.46	0.62	935.35
3	9.29	1493	160.75	6.22	0.67	755.60
4	8.55	1246	145.75	6.86	0.80	948.50

Table 2: Effects of varying the reentry offset on dynamics of the game in Nannon

We also generated similar statistics with different value for p ranging from 0.0 to 1.0 in both domains. We leave out those results for brevity in this report. However, the statistics from varying p suggested that medium values of p would lower the visitation variance in mini-gammon.

Also, we realized that by varying p we would be varying the degree of stochasticity in the domains. In one of his initial papers [9] on TD-Gammon, Tesauro mentions the high degree of stochasticity in backgammon as one of the reasons why learning backgammon is a challenging task. However, in his later paper [10] he mentions the stochasticity in backgammon as one of the potential reasons why TD-Gammon works so well. The high degree of stochasticity facilitates exploration by the agents. This is specially of importance in TD methods which have to manually balance exploitation with exploration. Also in self-play learning, the stochastic outcome of moves can help the learner get out of local minima.

For this reason, we mostly counted on experimenting with various reentry offsets to find a link between ergodicity and learning efficiency. However, we also performed experiments with varying p with the hope that it could help us observe the impact of changing the domain's dynamics in different ways.

Figures 3 and 4 show the state discovery rates corresponding to different reentry offsets in mini-gammon and Nannon, respectively. Regardless of the eventual number of states discovered for each value, it seems that with a lower offset the plots plateau more quickly. This might translate to an increased efficiency in exploration. Figures 5 and 6 show the state discovery rates with varying values of p in mini-gammon and Nannon, respectively.

In general, the main challenge with designing the problem domain is to vary the characteristic of the game that we are interested in without changing much else. For example, the data in Tables 1 and 2 show that lowering the value of the offset increases the total number of states discovered per 1000 games. But it is not clear whether discovering more states indicates that we would be doing better at exploration, or if it is just due to the games taking longer to complete.

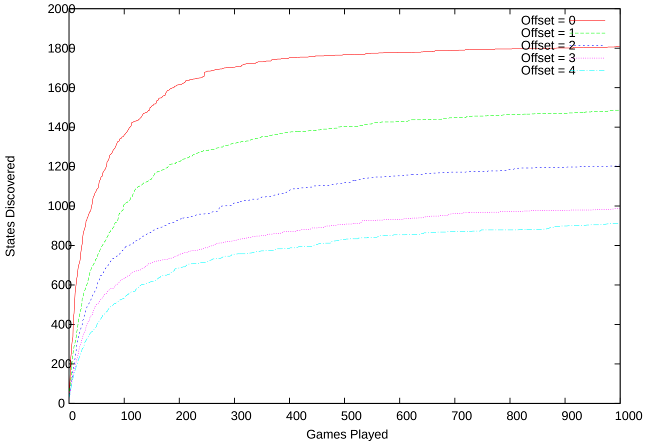

Figure 3: Total number of states discovered with different values of reentry offset in mini-gammon

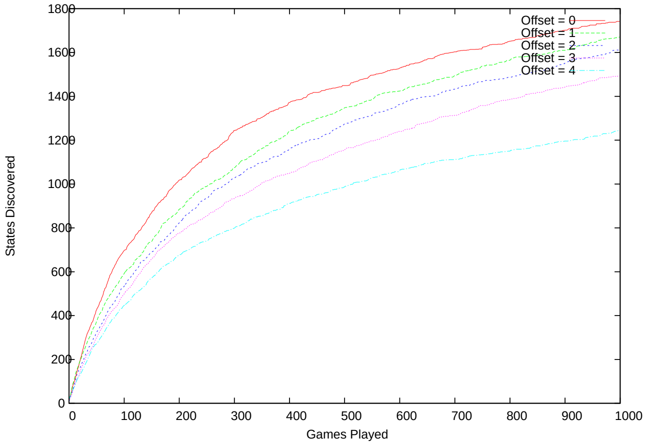

Figure 4: Total number of states discovered with different values of reentry offset in Nannon

Tables 7 and 8 show the distribution of visits to individual states in mini-gammon for offset = 0, and offset = 4 over 1000 games. The states have been order based on the earliest ply number when they were visited. With offset = 0, there is an increased rate of visitation to early-game states. This might be because when the checkers are hit and reenter, then enter in the same positions that they would occupy by moving forward in the beginning of the game. Since with TD methods, the state updates values that propagate back to the initial states decay in magnitude, an increase in number of visitations to early-game positions might speed up the learning process. Some of the higher rate of visitation to the early-game states must be attributed to the reversed funneling effect due to opening the game from the same starting state every time. It would be interesting to compare this plot with a similar plot on the game of chess and see if the downward slope on the visitation counts to early-game positions would be different in chess.

3. LEARNING AGENTS

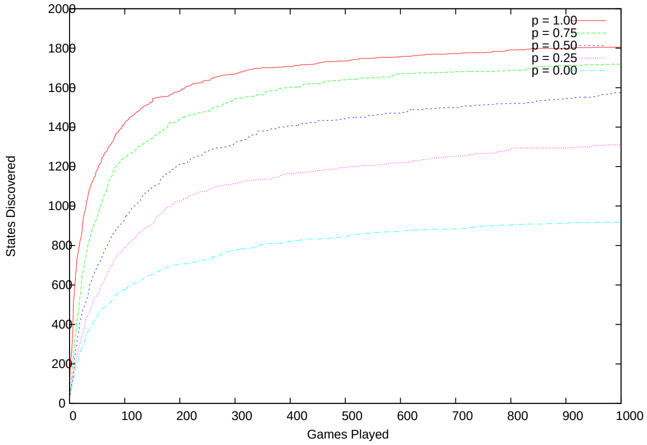

Figure 5: Total number of states discovered with different values of p in mini-gammon

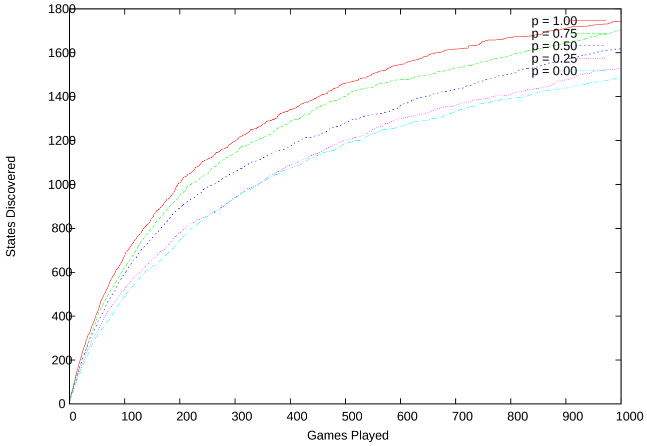

Figure 6: Total number of states discovered with different values of p in Nannon

In this section, we discuss the three types of learning agents that we developed for this project.

3.1 A Tabular Sarsa(λ) Agent

In order to develop an idea on how well one could learn to play mini-gammon and Nannon, we started by implementing a reinforcement learning agent using tabular Sarsa(λ) [8]. Since we didn't have access to any existing opponent which we could play against, we settled on training and evaluating our learner against an opponent which selected moves randomly.

The tabular Sarsa(λ) learner maintains a Q table on state-action pairs. The state is constructed by encoding the current board configuration, which includes the positions of the checkers, the current roll of the die, and the player who has the turn.

After making each move, the Q values are updated according to this equation [8]:

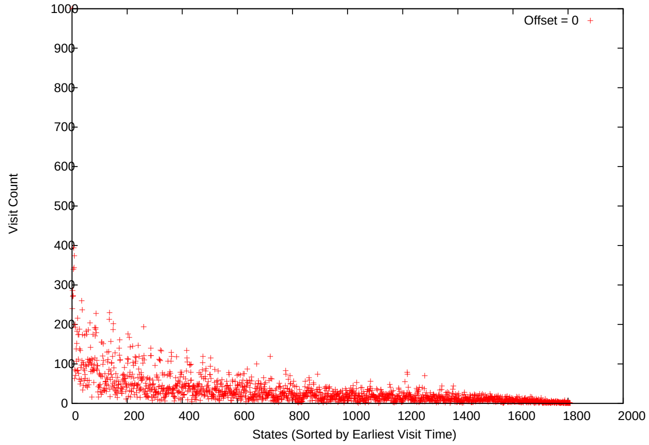

Figure 7: Distribution of number of visits to individual states in mini-gammon for offset = 0

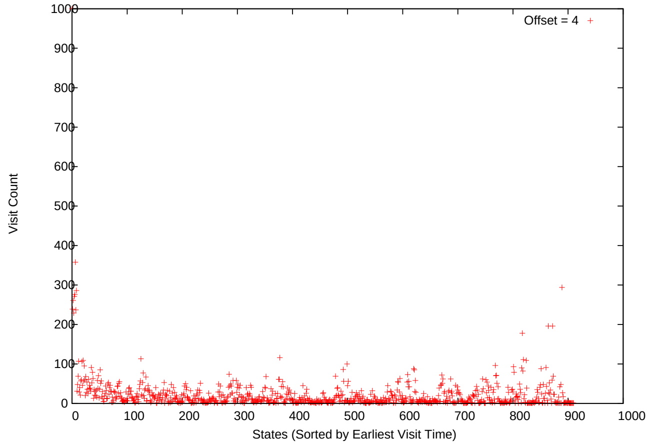

Figure 8: Distribution of number of visits to individual states in mini-gammon for offset = 4

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

where

$$\delta_t = r_{t+1} + \gamma Q_t(s_{t+1}, a_{t+1}) - Q_t(s_t, a_t)$$

The term r_{t+1} stands for the reward received on transitioning from state s_t to the state s_{t+1}, and δ_t is the TD update that is applied to each state. In backgammon, the are no intermediate rewards in the game and only at the end of the game the winner and the loser receive a reward value of 1 and 0, respectively. $e_t(s,a)$ stands for eligibility traces for past visited states. After making every moves, the eligibility traces for the past states are decayed according to the λ parameter, which we set to 0.90.

Since there is no discounting factor in backgammon, we set γ equal to 1.0. We also used optimistic initialization for entries in the Q table to facilitate exploration. For α, the learning rate parameter, we initially used a fixed value of 0.1, but later implemented an annealing schedule that set α equal to 1.0 divided by the number of visitations to the state whose value is being updated. The learner used an ϵ-greedy move selection method with ϵ equal to 0.05 during the entire training.

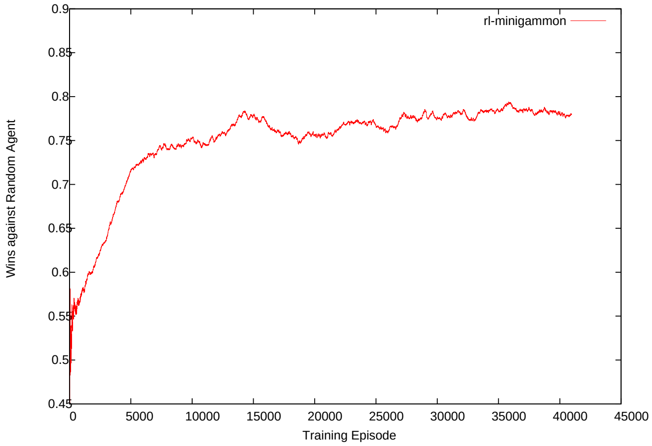

Figure 9: Learning performance of the Sarsa(λ) agent in mini-gammon

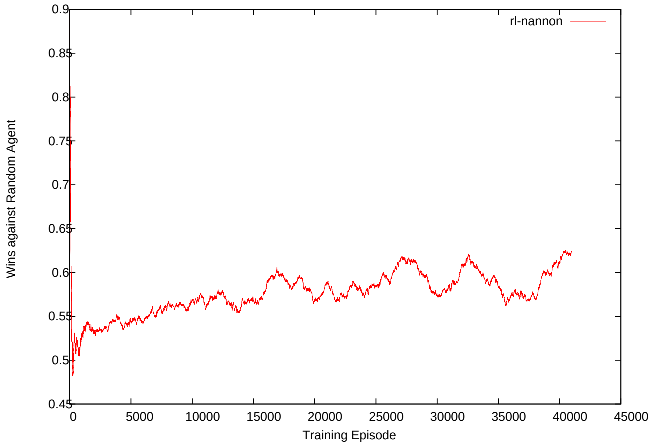

Figure 10: Learning performance of the Sarsa(λ) agent in Nannon

Figures 9 and 10 show the learning performance in mini-gammon and Nannon, respectively. Here, the offset parameter is set to its natural value of 0, corresponding to the standard versions of the games. In these plots, the win ratios have been averaged over the last 3000 episodes.

Even though our learner is not perfect and it is possible that we could improve the results by tuning the learning parameters, the plots suggest that the mini-gammon domain admits more to developing strategies, at least against a random opponent. The learner could reach a win ratio of 82% in our trials for mini-gammon. This is while for Nannon, the agent couldn't win more than 63% of the games against the random opponent.

3.2 The Hill-Climbing Agent

To let ourselves evaluate the impact of ergodicity on efficiency of learning through self-play, we implemented a Hill-Climbing (HC) agent following the method used by Pollack for HC-Gammon [6]. The motivation for developing this agent was to observe if a higher degree of ergodicity would make self-play more efficient.

The evolutionary setup used to evolve the hill climbing agent is as follows. Following Pollack's method, we use a population of size one. The agent uses a neural network as the state value predictor for board configurations. The board configuration is fed to the neural network using a unary encoding of the number of checkers from each player on each cell on the board. For example, in mini-gammon, there are two input nodes per player per board cell. If the player playing white has one checker on a cell, the first input is turned on. If they have two checkers on the same cell, both inputs are turned on. (We also had the option of turning on only the second input if two checkers were present, but I believe this is what Tesauro and Pollack did, even though I couldn't find a reference in their papers.) The network input encoding also includes two inputs for encoding the player who has the turn.

The networks used by Tesauro and Pollack used additional inputs. Pollack included an input node which was turned on during the endgame, where all the checkers from the two players have moved past each other. Tesauro also included input nodes which provided the network with additional information, such as the probability of getting hit on the next dice roll.

Following Pollack's method, our network has a single output node, which predicts the value of the state for the white player. Tesauro used two output nodes predicting values for the white and black player separately (He actually used four output nodes to take care of winning gammons, which would receive a double reward.)

When selecting moves, the agent considers the outcome of playing each possible checker. The resulting configurations are fed into the network and the predicted state values are extracted. Then the agent selects the configuration that has the highest value for itself, depending on the checker color that it is playing. If the agent is playing white checkers, it selects the configuration that has the highest predicted value for white and if it is playing black checkers, it selects the configuration that has the lowest predicted value for white.

We used a total of 10 hidden units in our neural network. This is lower than what Pollack and Tesauro used, but considering that our problem domain is simpler, we thought 10 hidden units might be enough. Using 10 hidden units and bias units brings the number of connection weights in our neural networks to 441 for mini-gammon and 281 for Nannon. The network used by Pollack had 20 hidden units and 3980 total weights.

At the start of each generation, the connection weights of the only member of the population (the champion) are mutated to create a challenger. The champion and the challenger play a number of challenge games. If the challenger wins a set number of the challenge games, the champion's weights are moved in the direction of the challenger's weights. If the challenger loses, a new challenger is created and the challenge games are repeated. We required winning 7 out of 8 games in our experiment. Pollack started with requiring winning 3 out of 4 games and increased the ratio of wins required after tens of thousands of generations. However, since our game is simpler, we thought requiring more wins from the start would be beneficial.

If the challenger beats the champion, the champion's weights are moved toward the challenger by 5%. That is, the new champion's weights are calculated as 0.95 × champion's weights + 0.05 × challenger's weights. This is what Pollack suggests to prevent losing a good champion when a challenger manages to beat the champion by luck.

Following Pollack's solution, we initialize all the network weights to 0.0. Pollack mutated the champion's weights by adding some Gaussian noise. He mentions in his paper that the noise was set up such that the RMS distance between the weights would be equal to 0.05. Based on our calculations, this amounts to a standard deviation of about 0.22. This is what we used for generating the Gaussian noise. It is possible that some other numbers would be more suitable for a network of smaller size that we use. But we settled on using the same numbers.

Our initial attempts at evolving the HC agent were not successful. At the end of each generation, we evaluated the champion against the random opponent for about 1000 games. The agent's win ratio would just fluctuate around 50-55%.

Then we realized that during our challenge games, we are always having the champion play the white checkers and the challenger play the black checkers. This is while during the evaluation games, the champion played as white and the random agent played as black. We expected that if the challenger beating the champion is an indication that the weights in the challenger's network were more accurate in predicting state values for white, then those weights could be used by a white player as well. However, for some reason, this was not the case.

In retrospect, one potential reason why this was happening could be due to the weights on the two network inputs that encoded who had the turn to move. When black evaluates the outcome of moving checkers, it creates a copy of the current state, applies the move, and encodes the resulting state to feed it to the network. Since the state automatically alternates the player who has the turn upon making a move, the board configurations that the black player evaluates would always have the input corresponding to "white's turn to move" on. So, the black player never uses the connection weights coming from the input corresponding to "black's turn to move" in its evaluations. Then if we have that challenger play as white, it would be using a different set of weights in its evaluations. Those would be the weights that are not used and challenged during the challenge games.

We modified the setup of the games so that the players would alternate playing black and white checkers on every other game. We were already alternating the player who started the game at the beginning of every game. In order to increase the fairness in games, we have also been controlling the random seeds [2]. We already used the same random seed for each pair of games. Adding the requirement for alternating checker colors increased the number of trials for every game (corresponding to an initial random seed) to four. The players would get to play the same game as white and black, and as the first and second player. At this point we started using this setup for all our game sets, during training and evaluation likewise.

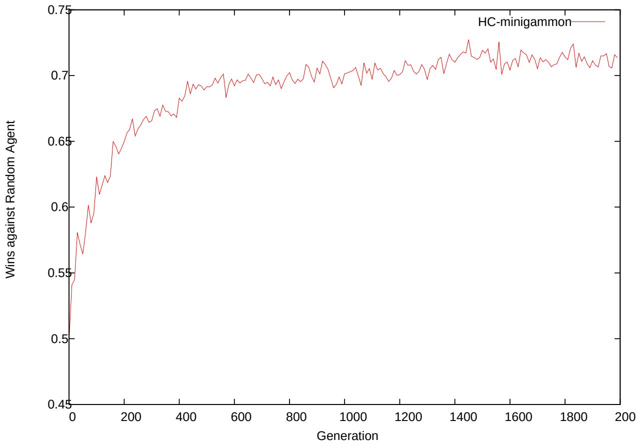

Figure 11: Learning performance of the HC agent in mini-gammon

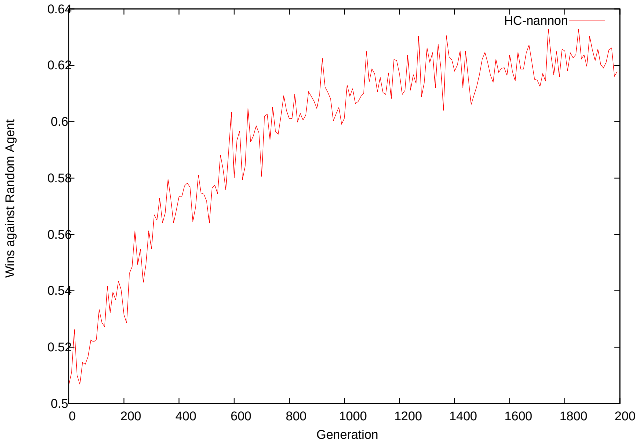

Figure 12: Learning performance of the HC agent in Nannon

Implementing this change greatly improved the performance of the HC agent. Figures 11 and 12 show the learning progress of the HC against in mini-gammon and Nannon, respectively. For these plots, the parameters controlling reentry offset and p were not used. At the beginning of each generation, we evaluated the champion against the random agent for 1024 games. The checker colors and the starting players were alternated during the evaluation games as well. The data for plots were computed by averaging results from 10 trials.

As the plots show, the agents were able to achieve win ratios comparable to what was achieved by the tabular Sarsa(λ) agent. A considerable portion of the time used in the evolutionary algorithm was spent on finding successful challengers. A single run of evolving the HC agent took about 12 hours to finish on the condor cluster in the department. Figures 13 shows the number of challengers mutated per generation until a successful one was found for mini-gammon. As the plot shows, in some generations, it could take up to 180 8-game challenges to find an agent that could beat the champion.

We have used a total of 2000 generations. It is possible that continuing the evolution for more generation could improvement the results. The number of weights in the neural networks that we used are about a tenth of the number of weights that were used for HC-Gammon.

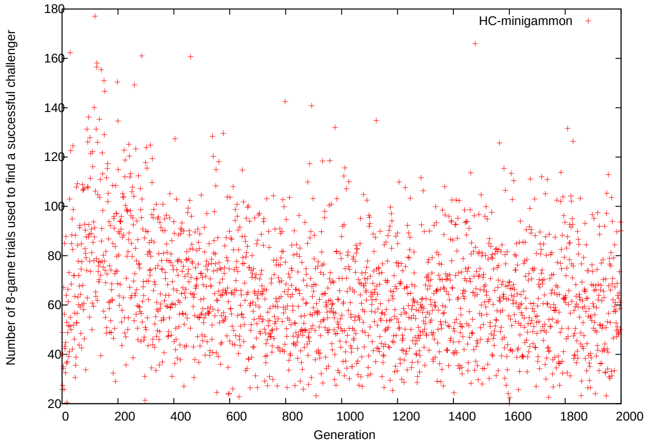

Figure 13: Number of HC challengers mutated per generation for mini-gammon

We also ran experiments with the hill-climbing agent by varying the parameters p and the reentry offset in mini-gammon and Nannon domains. The results are shown in Figures 14, 15, 16, and 17. All the plots have been generated based on data averaged over 10 trials. We have included the plots here for improving the presentation of the report. These results are discussed in Section 4.

3.3 The Neural TD Agent

The last agent that we implemented was a reinforcement learning agent that used a neural network and trained the weights in the network using a TD learning method, namely Sarsa(λ). The motivation for developing this agent was to see if higher degrees of ergodicity would help with reinforcement learning.

This agent used configurations similar to what was used by Tesauro for TD-Gammon. We used the same network architecture that we used for the HC agents, with the difference that we used two output nodes predicting state values for

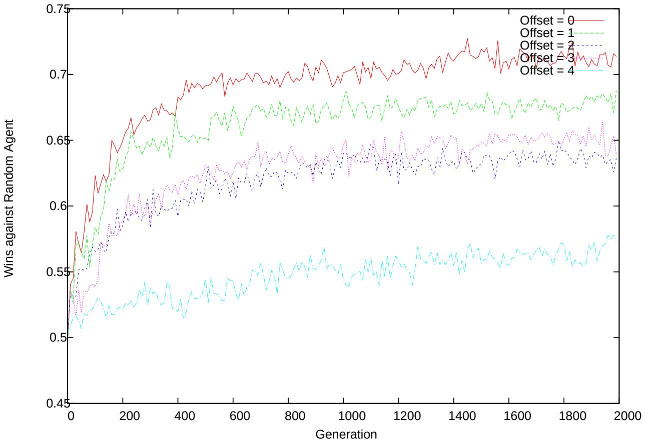

Figure 14: Learning performance of the HC agent in mini-gammon with different reentry offsets

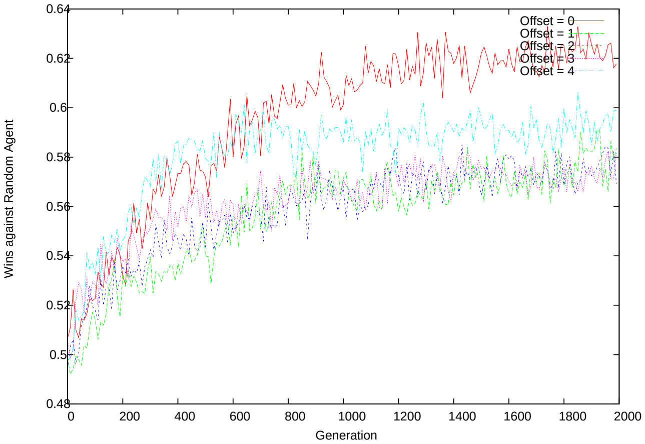

Figure 15: Learning performance of the HC agent in Nannon with different reentry offsets

white and black separately. This would be in line with the architecture of the output layer in TD-Gammon. We didn't provide the network with any additional inputs besides encoding the positions of the checkers and player who has the turn.

Following Tesauro's recipe, we had the neural TD agent play both sides of the board during the training games. The agent's move selection algorithm was similar to what we used for the HC agent. The agent would consider the outcome of moving every possible checker and feed them to the network. It would then select the move that would result in a configuration with the highest value for itself. Since the network provided values both for white and black, we computed the difference between the two values available at the output nodes and used the difference to determine the relative values of the states. It is not clear from Tesauro's papers whether he used this same approach for interpreting the state values or not.

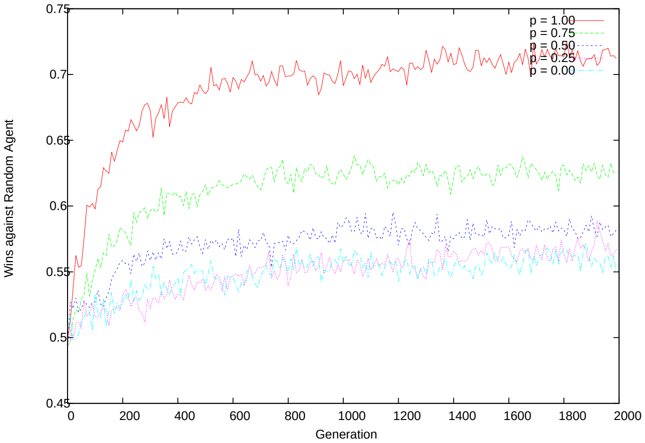

Figure 16: Learning performance of the HC agent in mini-gammon with different values of p

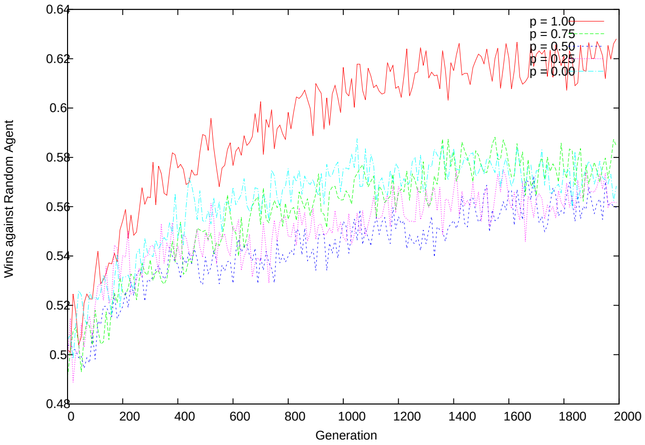

Figure 17: Learning performance of the HC agent in Nannon with different values of p

The neural TD agent used a TD learning algorithm. Tesauro provides the following equation for updating the weights in the network after observing every transition:

$$w_{t+1} - w_t = \alpha(Y_{t+1} - Y_t)\sum_{k=1}^{t}\lambda^{t-k}\nabla_w Y_k \tag{2}$$

The w vector represents the weights in the network. The term Y_t represents the network outputs corresponding to the state visited at time t. The term $\nabla_w Y_k$ is the gradient of the network output with respect to the weights. The term λ^{t-k} determines the eligibility of each of the past visited states for receiving an update from temporal difference observed at the current state $(Y_{t+1} - Y_t.)$

The components in this equation directly correspond to the components in Equation 1 that was used for calculating the TD updates for the Sarsa(λ). So, instead of modifying the network weights directly according to the equation used by Tesauro, we applied the updates by computing the targets for network inputs matching the required TD updates and used backpropagation to update the weights in the network. More specifically, after every transition, we computed the size of the updates on all eligible past states and created a training dataset that mapped the network inputs corresponding to those states to new output values which were calculated by adding the TD update to the current output of the network.

We initially used the same learning parameters that were used by our Sarsa(λ) agent for computing the TD updates. More specifically, we set λ to 0.9, and used an annealing schedule to compute the value of α in Equation 1 which controlled by how much the current state value should be moved in the direction of the update.

For α in Equation 2 which controlled the learning rate of the neural network, we used a value of 0.1. The network was trained on the dataset containing the TD updates for exactly one epoch.

This implementation was quite slow, as it required training the network after making almost every move. So we implemented some approximations to speed up the process. More specifically, we delayed applying the TD updates until the very end of every episode. During the episode, we stored the requested TD updates for every state. Then at the end of the episode, for every state all the TD updates requested during the episode were summed up and applied at once in one training dataset. Note that these approximations would lower the accuracy of our implementation, as the state value updates that should be applied during an episode would not become visible to the other states until the end of the episode. This could specially have an impact if states are revisited during the same episode. However, this approximation made the algorithm about 20 times faster.

Since we are not modifying the network until the end of the episode, we also implemented some caching mechanisms for avoiding hitting the network for getting state values too often. We also cached the network input encodings for board configuration so that they won't need to be recomputed.

Our initial attempts at training the neural TD agent were not successful. We tried increasing and decreasing the value of the learning rate parameter for backpropagation, α, and also the eligibility decay rate, λ, but they didn't help.

Later we realized that the source of the problem was the α parameter in Equation 1. This parameter is designed to limit the fluctuations in state values when using a tabular value approximator. In a domain were the outcome of the actions are stochastic, we wouldn't want an occasional observation to distort the state values too much. However, the α in Equation 2 used for computing updates to the network weights is already limiting the amount of change to the network's predicted values by moving the weights only a fraction of the way toward the requested targets. This is especially true since we were applying backpropagation only for one epoch. Since we used an annealing schedule to lower the value of α in Equation 1 for computing the TD updates, then after a number of episodes, the TD updates would shrink considerably and the network weights would practically become fixed.

Realizing that the learning rate used for backpropagation is already damping the impact of the updates, we set the value of α for computing the TD updates all the way up to 1.0. This improved the results significantly.

Figures 18 and 19 show the learning performance of the neural TD agent in mini-gammon and Nannon. The plots have been generated by averaging the data from 10 trials. During each iteration, the agent was trained against itself for 16 games. Then it was evaluated against the random agent and its win ratio was recorded. The agent's learning mechanism was paused during the evaluation games and resumed before playing the next training game set.

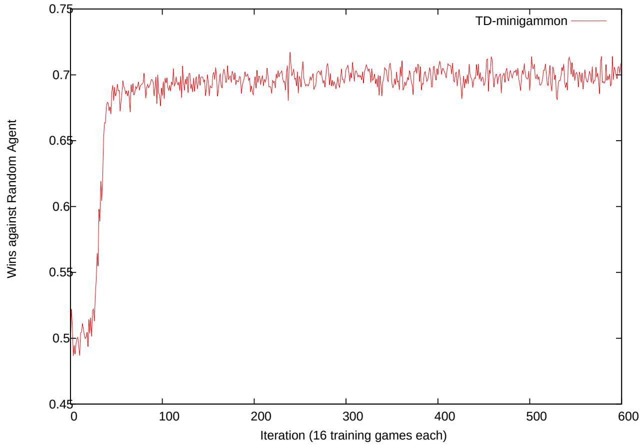

Figure 18: Learning performance of the neural TD agent in mini-gammon

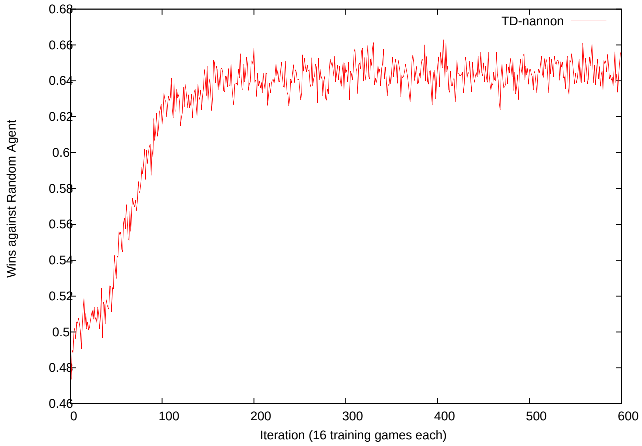

Figure 19: Learning performance of the neural TD agent in Nannon

The performance of the TD agent is comparable to the HC agent. However, for mini-gammon, it doesn't reach the win ratios achieved by the tabular Sarsa(λ) agent. We don't know the reason behind this difference. It is possible that increasing the number of hidden units could help. Also, it is possible that we are suffering the "continual learning problem" that is discussed by Whiteson in [12].

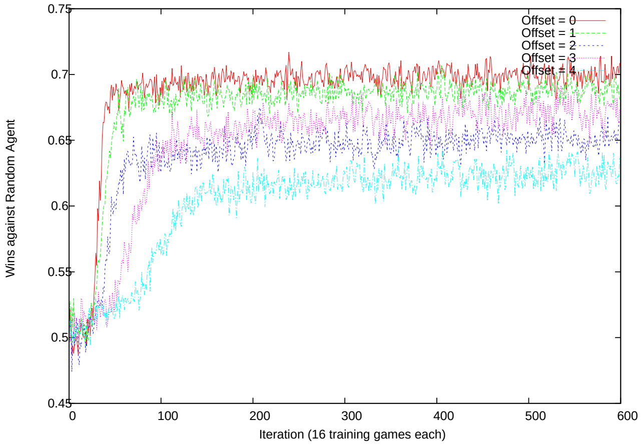

Figure 20: Learning performance of the neural TD agent in mini-gammon with different reentry offsets

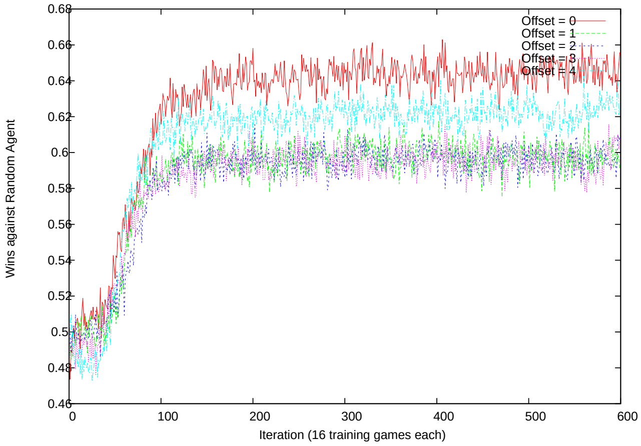

Figure 21: Learning performance of the neural TD agent in Nannon with different reentry offsets

For training a TD agent through self-play, we had two options. We could either have a single agent play both sides of the board, or we could use two instances of the agent and train them by playing against each other. We used a single instance, following the approach used by Tesauro. We haven't verified this, but we believe using a single instance would work better than using two instances of the same agent. If an agent only plays one side of a game, say white, then it only observes the transitions between states where white gets to play. It is true that even though the agent doesn't observe the intermediate states where black gets to move, given enough sample trajectories, it can still develop a good approximation of the state values. However, observing the intermediate board configuration can speed up the propagation of state values, since the agent is also developing approximations on the values of the opponent's states.

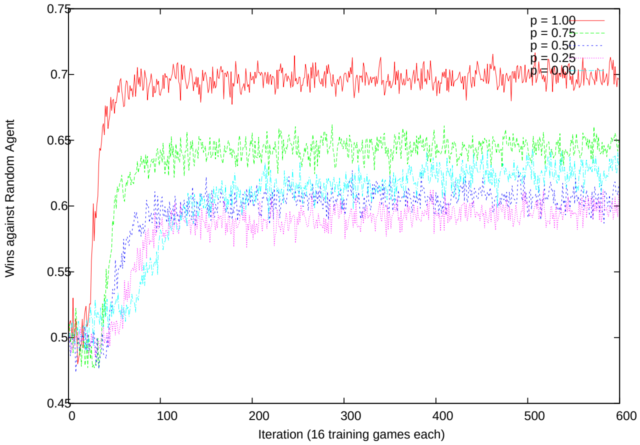

Figure 22: Learning performance of the neural TD agent in mini-gammon with different values of p

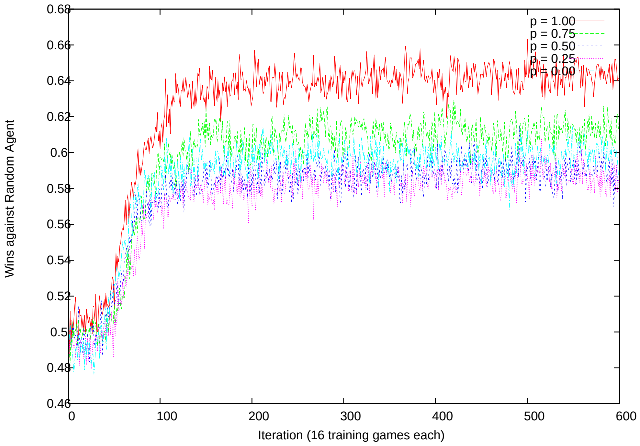

Figure 23: Learning performance of the neural TD agent in Nannon with different values of p

If the agent playing white applies an update on one of the black's states, it can later on immediately use that estimate upon seeing a transition from one of its own states to that black state.

Figures 20, 21, 22, and 23 show the results of running the neural TD agent on mini-gammon and Nannon domains with different values of reentry offset and p. The following section discusses these plots.

4. EXPERIMENTS WITH ERGODICITY

In this section we discuss the results of running experiments involving the hill-climbing agent and the neural TD agent by varying the parameters p and the reentry offset in the mini-gammon and Nannon domains.

The plots in Figures 14, 15, 16, and 17 show the results for the HC agent. It can be seen in the plots that with lower values of reentry offset and higher values of p the HC agent can achieve higher win ratios against the random opponent.

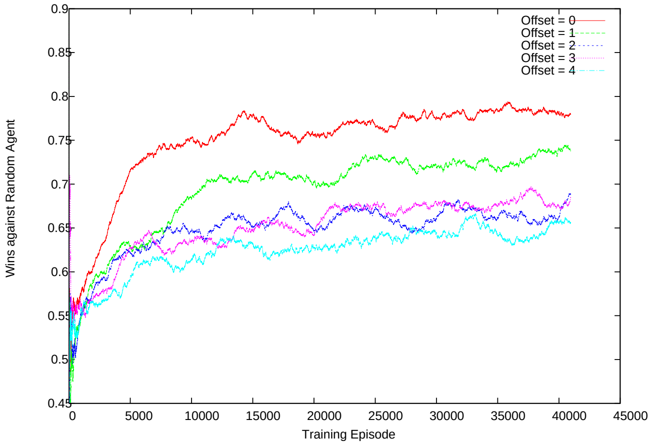

Figure 24: Learning performance of the Sarsa(λ) agent in mini-gammon with different reentry offsets

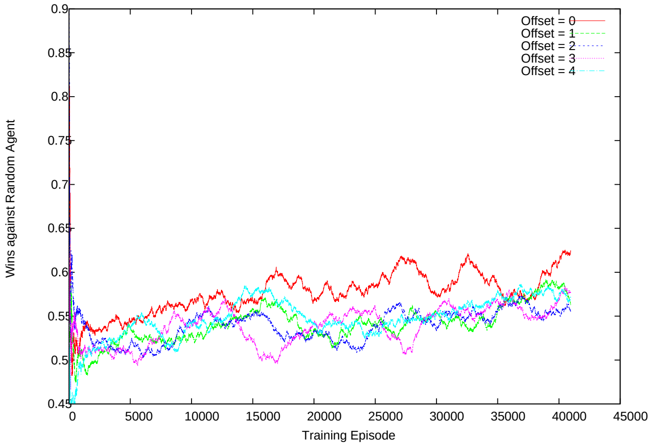

Figure 25: Learning performance of the Sarsa(λ) agent in Nannon with different reentry offsets

However, having a higher win ratio can not be attributed to the presumed higher degree of ergodicity associated with these parameter values.

In fact, it might be the case that increasing the reentry offset and lowering the value of p changes the dynamics of the games to the point that achieving the same win ratio is not possible any more. This is not hard to imagine, since by reducing the chance for engaging the opponent checkers, the amount of leverage an intelligent agent has on a random opponent would also decrease.

To be able to better evaluate the performance of our HC and TD agents under different parameter values, we also ran the tabular Sarsa(λ) agent against the random opponent with the same ranges of parameter values.

Figures 24, 25, 26, and 27 show the performance of the Sarsa(λ) in the two domains. It is evident that the Sarsa(λ) agent can also achieve higher win ratios with those particular parameter values.

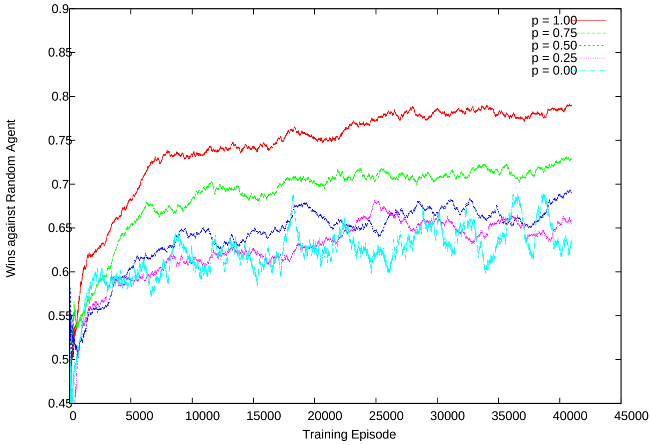

Figure 26: Learning performance of the Sarsa(λ) agent in mini-gammon with different values of p

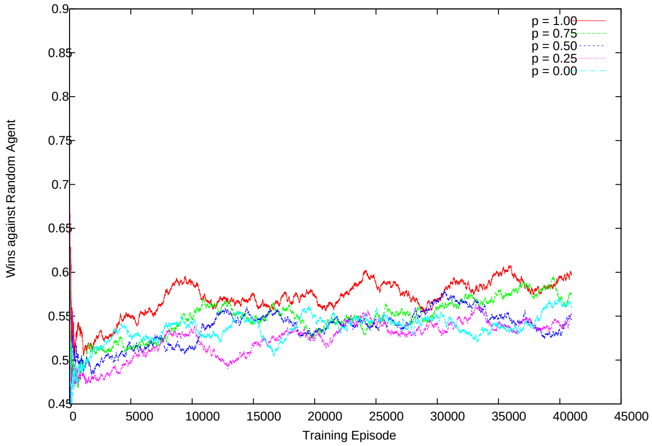

Figure 27: Learning performance of the Sarsa(λ) agent in Nannon with different values of p

On the other hand, looking at the highest possible win ratios is not the only way to assess the learning performance of a learner. We can also consider the speed at which the agents reach the eventual win ratios, as a metric for evaluating the efficiency of a learner.

Figures 20, 21, 22, and 23 the results from the TD agent in both domains. Figure 20 suggests that with a lower value for the reentry offset, the neural TD agent is able to reach its maximum win ratio more quickly, compared to higher offset values.

We haven't had enough time to study these plots more carefully, but it's evident that the parameters that we have designed the presumed ergodicity of the domain are affecting other aspects of the dynamics of the domain. It's not clear whether one could ever only change the degree of ergodicity in a domain without affecting much else. But we would at least need to be able to isolate the effect of our changes on other aspects of the domain's behavior.

5. RELATED WORK

Legg and Hutter [4] prove that ergodic MDP are self-admitting to optimization. That is, given unbounded time, an optimal policy can always be developed for ergodic MDPs using self-optimization. While backgammon is not fully ergodic, it can be said to have a high degree of ergodicity, and this result is encouraging. At the same time, this work does not discuss the efficiency of learning in terms of the time needed for finding an optimal policy.

We couldn't find other related papers discussing the impact of ergodicity on learning, but there is considerable amount of work done on using reinforcement learning and evolutionary computing for playing board games.

Wiering et al [13] use temporal difference learning to learn a game position evaluation function. They attribute the effectiveness of self-play in backgammon to the smoothness of the evaluation function. They discuss that since a a particular position's value is averaged over the possible dice rolls, unlike chess and go, nearby positions do not differ significantly in their values, and the evaluation function is smoother.

Runarsson and Lucas [7] study and compare the performance of temporal differencs learning using self-play and co-evolution for learning a position evaluation function for the game of go. They report that temporal difference learning learns faster, but under the right set of parameters, co-evolution can learn to win more games than temporal difference methods.

Darwen [1] showed that co-evolution could evolve a linear backgammon position evaluator that outperformed the linear version of Pubeval, which uses temporal difference learning. However, co-evolution had difficulty evolving a nonliner position evaluator that performed well, as it required a prohibitively large number of games per generation.

Kotnik and Kalita [3] compare temporal difference learning and co-evolution for the game of rummy. They report that co-evolutionary agents in general outperform TD agents and exhibit a more balanced strategy.

6. CONCLUSIONS AND FUTURE WORK

We have implemented three different types of learners that can successfully learn to play the two games discussed in this report. Even though the results from our experiments with varying ergodicity are not conclusive at all, they seem promising. We have the ingredients in place to try our approach on other domains or with different configurable parameters. For example, it is not difficult to increase the number of checkers or increase the size of the board in minigammon.

We would also need to develop some concrete metric for measuring the degree of ergodicity in a domain. One of the methods that we have considered is to analyze the games and count the number of backward transitions in every game. Suppose we sort all the states in the game by their earliest visit times. A transition $s \rightarrow t$ would be backward if we have $earliest\_visit\_time(t) < earliest\_visit\_time(s)$. We can also consider the length of a backward transition by calculating the difference between the earliest visit times to the two states. Having more and longer backward transitions should correspond to a higher degree of ergodicity.

7. REFERENCES

[1] P. Darwen. Why co-evolution beats temporal difference learning at backgammon for a linear architecture, but not a non-linear architecture. In *Evolutionary Computation, 2001. Proceedings of the 2001 Congress on*, volume 2, pages 1003–1010. IEEE, 2001.

[2] J. Kiefer and J. Wolfowitz. Stochastic estimation of the maximum of a regression function. *The Annals of Mathematical Statistics*, 23(3):462–466, 1952.

[3] C. Kotnik and J. Kalita. The significance of temporal-difference learning in self-play training td-rummy versus evo-rummy. In *MACHINE LEARNING-INTERNATIONAL WORKSHOP THEN CONFERENCE-*, volume 20, page 369, 2003.

[4] S. Legg and M. Hutter. Ergodic mdps admit self-optimising policies. Technical report, Technical Report IDSIA-21-04, IDSIA, 2004.

[5] J. Pollack. Nannon: A nano backgammon for machine learning research. In *Proc. 2005 IntâĂŹl Conf. Computational Intelligence in Games*, pages 277–284, 2005.

[6] J. Pollack and A. Blair. Why did td-gammon work? In *Advances in Neural Information Processing Systems*, pages 10–16. Citeseer, 1997.

[7] T. Runarsson and S. Lucas. Coevolution versus self-play temporal difference learning for acquiring position evaluation in small-board go. *Evolutionary Computation, IEEE Transactions on*, 9(6):628–640, 2005.

[8] S. Singh and R. Sutton. Reinforcement learning with replacing eligibility traces. *Recent Advances in Reinforcement Learning*, pages 123–158, 1996.

[9] G. Tesauro. Practical issues in temporal difference learning. *Machine learning*, 8(3):257–277, 1992.

[10] G. Tesauro. Temporal difference learning and td-gammon. *Communications of the ACM*, 38(3):58–68, 1995.

[11] G. Tesauro. Comments on "co-evolution in the successful learning of backgammon strategy". *Mach. Learn.*, 32:241–243, September 1998.

[12] S. Whiteson and P. Stone. Evolutionary function approximation for reinforcement learning. *The Journal of Machine Learning Research*, 7:877–917, 2006.

[13] M. Wiering, J. Patist, and H. Mannen. Learning to play board games using temporal difference methods. *UU-CS*, (2005-048), 2005.