Implementing Checkers using Alpha-Beta Pruning and Reinforcement Learning*

Varun Raval, Ashutosh Ghildiyal
College of Computer and Information Science, Northeastern University
360 Huntington Ave, Boston, MA 02115
raval.v@husky.neu.edu, ghildiyal.a@husky.neu.edu

1 Introduction

Checkers is a century-old game that Computer Scientists have studied since the 1950s. The game is extremely complex, with roughly 5×1020 possible positions, about a million times more complex than Connect Four. Our project's focus on the game of checkers is to create an artificial intelligence that a user can play against. The main task of this artificial intelligence is to gather the available moves, select the strongest action. In checkers, this is more complicated than what may seem at first glance, since every action could have ramifications several turns in advance, and strategy in taking actions is crucial to a good game-play.

For our experiments, we used two player checkers game with 8x8 checkerboard. Each player has 12 pieces initially. Normal piece can move forward in diagonal directions. A king can move forward and backward. All the pieces can only move single step at a time. During attack moves, a piece can capture multiple pieces of opponent if there is a sequence of attacks. More specific rules are described later.

Even though the game is very complex, the actions in any state are very simple. This makes checkers an ideal game to study Artificially Intelligent algorithms. Because of the simplicity of the action space, it becomes easy to train the reinforcement learning agents and compare them. This was our main motivation to apply artificial intelligence to checkers game.

This paper analyzes two kinds of approaches to play checkers. A non-learning approach and a learning approach. For non-learning approach, we have implemented minimax agent with alpha-beta pruning. For learning approach, we have implemented Q learning and SARSA learning. Both the agent were able to play checkers well against human. As the trend, reinforcement learning agents were not able to play as well as alpha-beta agents, but they did learn nice moves from alpha-beta agent.

2 Related Work

The history of creating artificial intelligence for checkers is nice and fascinating. It started with Arthur Samuel in the 1950's, who used rote learning and minimax to create a competent AI, despite poor computation power in that era. Interestingly, Samuel was one of the first computer scientists to implement alpha-beta pruning, a technique we implemented. However, his evaluation heuristic was simpler than ours, and for at least part of his research, he kept his weights entirely constant, preferring to optimize his search in other ways (including rote learning).

The most successful checkers program is Chinook, the program the University of Alberta created that plays the game optimally. The basic approach was to create an endgame database with a definite outcome, and prove that every state in the game would be reduced to one of these endgames. Since this approach's goal is to completely solve the game of checkers, it is entirely different from our framework, which cannot search deep enough. However, this program is a great oracle that we can compare our agents with, even though it's impossible to beat and very, very difficult to draw.

3 Approach

We have considered checkers game with following specific rules:

1. There is no flying king. This means that a king, similar to a normal piece can only take one step in one move. Though, a king can make a move in forward or backward direction.
2. Pieces has to attack an opponent's piece, if there is an attack move.

The main reason for the above rules is that they restrict the number of moves in a state which helps in training reinforcement learning models. Also, less number of moves help in comparing performances of different algorithms since the features or evaluation function required for them are not needed to be much complicated.

Yet, the state space of the checkers game is considerably large. Hence, it is challenging to craft good evaluation function for algorithms like minimax and also challenging to perform reinforcement learning. Considering the action space, assuming each piece is a king in a given state, every piece can have 4 actions (assuming all the actions are valid), 8x8 checkers board has 48 actions (each player contain 12 pieces). This is the upper bound on the actions in each state.

We have built agents for checkers game using following two approaches:

*These match the formatting instructions of IJCAI-07. The support of IJCAI, Inc. is acknowledged.

1. Minimax Algorithm with Alpha-Beta Pruning
2. Reinforcement Learning

3.1 Minimax Algorithm with Alpha-Beta Pruning

A common application of minimax algorithm is zero sum games. Checkers is a zero sum game and we have built an agent using minimax algorithm which we will refer to as minimax agent.

Minimax agent tries to search for the best action from the given state assuming that the opponent will play optimally. Given a state s, minimax agent will find successor states s' for all the possible actions a. Given successor state s' in which it is the turn of opposite player, minimax agent will find all the successor states of state s' using all the actions a'. Minimax agent will keep on doing so until the game ends. Then in some state s'', if it was the turn of the opponent, minimax agent will choose the action that is best for the the opponent and if it was the turn of the agent itself in s'' then agent will choose the action that is best for itself. In this manner it propagates upwards using the score obtained at the end of the game and chooses the action in the current state.

As mentioned earlier, each state in checkers can have 48 actions. An average game in checkers lasts for 50 moves. If minimax agent searches for the correct action in the next state enumerating all the states till the end of game, it has to explore 48^{50} leaves. This kind of future view in practically intractable. Hence, instead of going to the end of game, evaluation function is used after certain depth to measure how good is a state for minimax agent. Higher the value of the evaluation function for a state, higher are the chances for the minimax agent to win from that state. Thus, instead of going to the end of the game, we can keep evaluation function at a certain depth. We can control the amount of time the minimax agent takes to think for a move by controlling the depth at which it calculates the evaluation function.

Many times, minimax agent searches within a branch even if the final answer is from the branch that is already visited. This kind of unnecessary searching can be reduced using Alpha-Beta pruning.

For implementing minimax agent for checkers game, we used alpha-beta pruning to prune the branches to decrease unnecessary search. We have used evaluation function at depth 3 for our experiments. The reason for such choice is that depth 4 would cause to explore $48^4 = 5,308,416$ leaves which will take long time on each step while depth 3 takes reasonable time. Evaluation function at lesser depth would have made the agent nearsighted.

We also considered implementing expectimax agent instead of minimax agent. The main reason for not using expectimax agent is that we wanted to train the reinforcement learning agent using the non-learning agent that we create. Because minimax agent thinks that opponent plays optimally, reinforcement learning agent can learn faster from that behavior.

Evaluation Function

Following is considered for a state s, for minimax agent

- If minimax agent wins, $+500$
- If minimax agent loses, -500
- If none of the above happens, evaluation function is summation of following values
  - + (1) times the number of minimax agent's pawns
  - + (2) times the number of minimax agent's kings
  - + (-1) times the number of opponent's pawns
  - + (-2) times the number of opponent's kings

We chose above values because higher is the value of an evaluation function for a state, better is that state for the minimax agent.

Game Rules

We also placed a limit on the number of moves a game can continue. It was to stop the game turning into infinite loop in case of testing with reinforcement learning agent. More details on it are provided later. In such case, to give penalty to both agent, we considered both the agents as lost and hence, minimax agent in that case will evaluate state as -500. We have kept the maximum number of moves to 200 which is 4 times the average number of moves in a checkers game. Also, we analyzed that after 150 moves in worst case, both agents come to a condition where both have only one piece left.

3.2 Reinforcement Learning

A standard approach in reinforcement learning is to learn by remembering the states that are already visited and its corresponding actions. The state space of checkers game is very huge. Hence, it is infeasible to keep a dictionary of the state action pairs. Instead, we can represent the state action pairs using the set of features. This technique is called reinforcement learning using function approximation [Sutton and Barto, 1998].

In most RL approaches, Q value is learned which is then used to find the optimal policy. Since Q value uses state action pair, it is necessary to effectively represent the state action pairs using features.

Features to represent state-action pair

Assume that agent is in state s and has many choices for action a, and for each choice of action, environment takes agent to a state s'. For our reinforcement learning agent, we used following features to represent state action (s, a) pairs:

1. Number of agent's pawns in s
2. Number of agent's kings in s
3. Number of opponent's pawns in s
4. Number of opponent's kings in s
5. Difference between number of opponent's pawns in state s' and s
6. Difference between number of opponent's kings in state s' and s
7. Difference between total number of opponent's pieces in state s' and s
8. Number of agent's pieces being attacked by opponent in state s'

We are capturing the state information using number of agent's pawns and kings and number of opponent's pawns and kings. We are capturing the information about action using difference between opponent's pawns and kings in next state and current state. We are also calculating number of pieces of the agent being attacked by the opponent if an agent is taking an action and helping the agent to consider action using that parameter. All the above features are made keeping in mind how a human player chooses an action from a state.

Reward function

For the reinforcement learning agent to learn, environment needs to provide feedback to the agent which indicates how good the move is. The agent takes action a in state s and moves to state s'. The opponent takes action in state s' and moves to state s''. The environment will then given the reward to the agent for the action a in state s. Thus reward function depends on current state of the agent s, the action it took a and the state in which it will take next action s''.

The reward function is of the form $R(s,a,s'')$. We created following reward function for this purpose:

- Reward function is sum of the following:
  - + $(-0.4)\times$ (number of agent's pawns in s $-$ number of agent's pawns in s'')
  - + $(-0.5)\times$ (number of agent's kings in s $-$ number of agent's kings in s'')
  - + $(+0.2)\times$ (number of opponent's pawns in s $-$ number of opponent's pawns in s'')
  - + $(+0.3)\times$ (number of opponent's kings in s $-$ number of opponent's kings in s'')
- If agent wins in state s'' or s', reward of $(+500)$
- If agent loses in state s'' or s', reward of (-500)
- If all of above is 0, a living reward of (-0.1)

The reward function is intuitive. Agent is taught to value keeping its own pieces safe more than taking the opponent's pieces. This is reflected by the weights in front of each reward in the first step.

Function Approximation

Using features, we can approximate the Q values using following equation:

$$Q(s,a) = w_1 \times f_1(s,a) + w_2 \times f_2(s,a) + \dots w_8 \times f_8(s,a)$$

$f_1, f_2, \dots, f_8$ are the 8 features described above.

We have to initialize the weights. We can either initialize all of them to zero or to some random value. In our experiments, we initialized the weights to 0

Q learning

Q learning is the off policy learning technique. Instead of learning Q values directly, we learn the weights using the equation below:

$$w_i = w_i + \alpha\big(R(s,a,s'') + \gamma \max_{a''} Q(s'',a'') - Q(s,a)\big) \times f_i$$

$i \in [1,8]$ for our experiments. α is the learning rate and γ is the discount factor. The way in which values were chosen is given in the Experiments section below.

SARSA

SARSA is the on policy learning technique. According to the literature, off policy technique in way described above does not work well with function approximation. Hence, we implemented on policy learning technique. We also got the similar results which are described in the Evaluation/Results section. Weight values are updated in the following manner:

$$w_i = w_i + \alpha\big(R(s,a,s'') + \gamma Q(s'',a'') - Q(s,a)\big) \times f_i$$

where agent will take action a'' in the state s''.

Policy Function

The agent needs a policy function to perform learning. We implemented following two policies:

1. Epsilon Greedy: With small ϵ probability choose random action and with $1-\epsilon$ probability choose optimal action.
2. Softmax Greedy: Choose actions using probability given by
$$\pi(s,a) = \frac{e^{Q(s,a)/t}}{\sum_{a'} e^{Q(s,a')/t}}$$

The main advantage of softmax is that for example among 5 actions if two actions have very high q values then softmax will give high probability to both of them while epsilon greedy will give high probability to only one of them, treating every other action of equal probability.

3.3 Experiments

To teach the Q learning agent and SARSA agent, we made both of them play games against Alpha-Beta agent. Once we figured out that off policy technique is not learning quickly, we stopped learning it and continued with only on policy learning. We let play SARSA agent 35,000 games against Alpha-Beta agent. The details on how we figured off policy is not learning quickly and results of learning with SARSA are given in Results section.

We implemented both policy functions described above. For softmax greedy policy, win and loss reward values were made to $(+10)$ and (-10) because $(+500)$ and (-500) were causing exponential overflow.

Choosing Parameter Values

To choose the value of α, we started with 0.01. We tried higher values of alpha but they caused the learning process to diverge very early causing the weight values to overflow.

Starting with $\alpha = 0.01$, we decreased the value of alpha by dividing by 2 every 10,000 games.

We picked initial value of ϵ as 0.5, and decreased it by dividing by 2 at every 10,000 games. Higher values of epsilon initially would have caused too much exploration which was not necessary because we have played many games against alpha-beta agent.

We picked initial value of temperature t of the softmax greedy policy function as 1 and decreased it by dividing by 2 every 10,000 games.

4 Evaluation/Results

4.1 Alpha-Beta agent

We evaluated Alpha-Beta agent by playing against human. We played many games against Alpha-Beta agent. Because we are novice players, Alpha-Beta agent was able to beat us almost all the times. Also, Alpha-Beta agent was able to trick us by sacrificing its own piece and taking two of our pieces.

4.2 Reinforcement Learning Agent

As mentioned earlier, we started training both Q learning and SARSA agents. After 7000 games we found that Q learning agent is not learning as fast as SARSA agent.

Figure 1. is the graph of performance of Q learning agent during training.

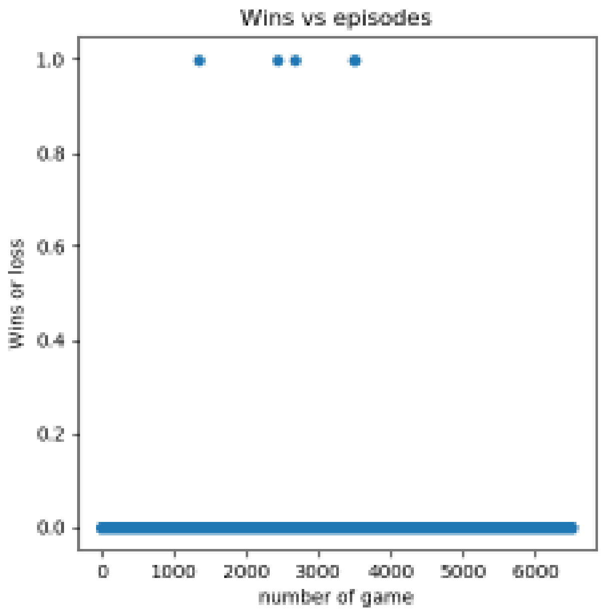

Figure 1: Number of game is shown on the x-axis and win or loss is shown on the y-axis for Q learning agent.

Figure 2. is the graph of performance of SARSA learning agent during training.

As seen form figure 1, Q learning agent wins very few number of times in the first 6000 games. As seen from figure 2, SARSA agent wins many times in the first 6000 games. Hence, it can be inferred that SARSA agent is learning faster than Q learning agent.

We performed evaluation of SARSA agent in following ways:

- We played many games against SARSA agent. SARSA agent was able to beat us almost all the times. Also, SARSA agent learned from Alpha-Beta agent to trick us by sacrificing its own piece and taking two of our pieces in return.
- During training, to see how well and fast agent is training, we made tests of 100 games after training of every 100 games. For example, after training 200 games,

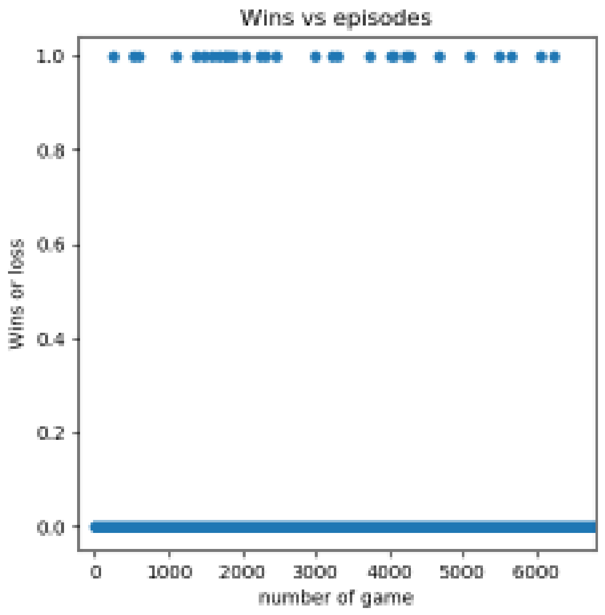

Figure 2: Number of game is shown on the x-axis and win or loss is shown on the y-axis for SARSA agent.

we performed 100 tests by setting α and ϵ to 0. Similarly, after training 300 games, we performed 100 tests by setting α and ϵ to 0. Initially, the agent lost against Alpha-Beta agent such that there were many pieces of the Alpha-Beta agent in final state. But, as SARSA agent kept learning, after many games of training, SARSA agent was able to bring the game to state such that only single piece of Alpha-Beta agent is left. But ultimately, Alpha-Beta agent won against SARSA agent.

Even though SARSA agent did not win against Alpha-Beta agent, it was able to learn from it gradually and bring the Alpha-Beta agent to the final state where only single piece of Alpha-Beta agent is left.

Figure 3 shows the graph of learning progress of SARSA over 35,000 games. After every game(episode), we measured the maximum q value of the start state among all the actions. As shown in graph, after 10,000 games, the variation in q values decreases. This means that after 10,000 games, SARSA has become more focused on the better q values. In other words SARSA has reduced the exploration and made steps towards obtaining correct q value. This behavior is seen because we have reduced alpha and epsilon by half after 10,000 games, enabling SARSA to concentrate on better actions. Again after 20,000 games we reduced alpha and epsilon by half and after few games, variance in q values decreased. Again after 30,000 games we reduced alpha and epsilon by half and variance in q values decreased.

5 Conclusion

Even though SARSA agent was not able to learn to win against Alpha-Beta agent, it was able to learn to beat a hu-

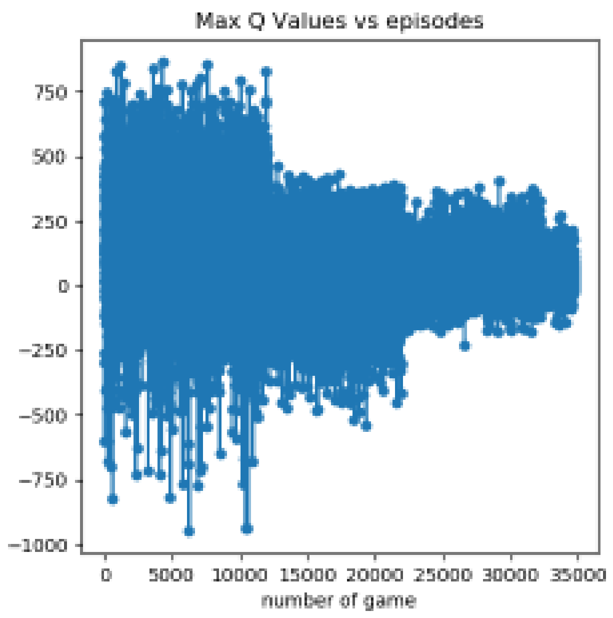

Figure 3: Number of game is shown on the x-axis and max q value of the start state is shown on the y-axis.

man player. We think it is possible to overcome that behavior and teach SARSA agent to beat alpha-beta agent with the use of a dictionary of good end moves.

Acknowledgments

We would like to thank Sam Ragusa for providing his implementation of Checkers board[1].

The code for Alpha-Beta agent is written by Ashutosh Ghildiyal. The code for Reinforcement Learning agent is written by Varun Raval. The design of checkers features, reward function and evaluation function for Alpha-Beta agent was done by both of us together. We designed the classes Game and GameState together. The design was inspired by pacman assignments.

Also, we would like to express our gratitude to Professor Stacy Marsella for his advice on our project work.

References

[Sutton and Barto, 1998] Richard S. Sutton and Andrew G. Barto. *Introduction to Reinforcement Learning*. MIT Press, Cambridge, MA, USA, 1st edition, 1998.

[1]https://github.com/SamRagusa/Checkers-Reinforcement-Learning/blob/master/Board.py