Polar Tic-Tac-Toe: an examination of artificial intelligence techniques through gaming

Joshua Leger, Fred Kneeland

December 10, 2014

Abstract

This paper examines several methods for the creation of an artificial intelligence system to play the game Polar Tic-Tac-Toe. The main player uses a minimax search that include a couple options. The first option is the pruning algorithm, which includes both an alpha-beta prune, and a nearest neighbour prune. The other option is the game state evaluation heuristic. Three heuristics were created, a basic test of free rows, a decision tree, and a neural network. The results were largely optimistic, with many combinations playing a respectable game.

1 Introduction

The problem to be solved for this project was to test different artificial intelligence methods for the game of Polar Tic-Tac-Toe. While the specific game for this project is Polar Tic-Tac-Toe, the main goal of the project was the experimentation of the various strategies, which can be applied to a large range of games. There has been much research in artificial intelligence put toward game theory, with several success such as Deep Blue, the first program to beat a grand master at chess, and TD-Gammon, the first program to beat a grand master at backgammon. There are many games though that have not been attempted and many artificial intelligence methods not tried extensively. The goal of this project is to try applying a range of techniques in a game not experimented much on.

Understanding the concepts that make game artificial intelligence perform well will help in applying the most efficient strategy to every game, and while this is a fun goal, it is also very useful to artificial intelligence as a whole because games give us a very controlled and structured environment to test solving techniques that would be hard to implement and get good data back on in the real world.

The main difficulty with games and with real life is the number of possibilities there are. In a game of chess, there are approximately 10^{47} possible states [2]. The sheer number of states a game can have makes it hard to enumerate an answer that is just a lookup table with the states and the recommended move. The most common solution is to use a game tree. Game trees represent games with a node representing a game state and the children of that node are the possible legal moves. Game trees can be optimized to not search the entire game every turn. They are used in many aspects of artificial intelligence, including real life applications. By optimizing tree searching in games, when applied in real life, computer scientists can already have a good idea of what algorithms work in what situations instead of testing every possible algorithm.

One way to help with tree searching are heuristics which estimate game states. There has been a lot of research on finding good heuristics, with one of the most famous used in the TD-Gammon algorithm. TD-Gammon plays backgammon on a level with grand masters. The heuristic used was created using a self learning algorithm called temporal difference neural networks. In this, the computer played games against itself many thousands or millions of times, learning as it went[3]. The result was a heuristic which was very good at predicting game states. Others have tried a similar approach with varying success. One heuristic used in the program tries this strategy.

In section 2, the different algorithms created are described. In section 3, the method for testing each algorithm is described and section 4 displays the results of each experiment. The final section analyses the results of the previous section.

2 Problem Statement

Polar Tic-Tac-Toe is a game with similar of Tic-Tac-Toe. Each player takes turns placing Xs and Os, trying to get four in a row. Rows can happen vertically, horizontally, and diagonally. The main difference between Polar

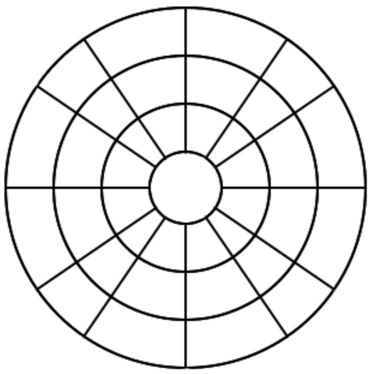

Figure 1: empty Polar Tic-Tac-Toe board

Tic-Tac-Toe and Tic-Tac-Toe is the board, which is shown in figure 1. The Polar Tic-Tac-Toe board has a height of four and a width of twelve, with the ends meeting up to create a circular board. The other main rule difference than with Tic-Tac-Toe is that moves, except for the first one, can only be made adjacent to a move previously made. The reason Polar Tic-Tac-Toe was chosen over Tic-Tac-Toe is because of the search space. Tic-Tac-Toe solvers are easy to implement because the search space is so small, it can be completely searched through in the time it takes to make a turn. Polar Tic-Tac-Toe on the other hand has a much larger search space and so heuristics and pruning must be utilized to create a player that plays well and makes turns in a reasonable amount of time.

Our experiments in Polar Tic-Tac-Toe revolve around the minimax search. We use three different heuristic types and two different pruning algorithms to try to improve both speed and accuracy. As mentioned before, games can be represented in tree structures with the possible moves for each state the children of the state node. The goal of the pruning algorithms is to reduce how much of the tree needs to be looked at. If entire subsections of the tree can be skipped, the algorithm can run faster and search deeper. Some can go deeper, but prune the wrong nodes, so a couple of different methods were attempted. The heuristics are needed because the tree cannot always search all the way down to a leaf node or terminal state. There needs to be a function it can call to get an approximation for how good or bad an intermediate state is, and the heuristic function fulfils that goal. Heuristics can be hard to come up with, so several methods were attempted.

2.1 Traditional Minimax Search

In lay terms, minimax is looking ahead a certain number of moves and picking which move to take for each round based on the assumption that your opponent will respond with the best move for them. We implemented min-max search recursively. To start it we would call the min-max function on all legal moves. The minimax function would call itself on all legal moves given the current move. It would iterate down until it reached the specified search depth. At this point it would call the given heuristic function for the game board with the added moves. This would return to its parent which if the parent was a max node it would return the highest value and if it was min-node it would return the lowest value. It would continue to bubble up until it returned a score for each move from the current position. The minimax player would then choose the move with the highest score.

2.2 Minimax with Alpha-Beta Pruning

Alpha-Beta seeks to reduce the extremely large search time of minimax. The idea behind Alpha-Beta is that if we know that a move is bad there is no need to find out just how bad it is. Alpha-Beta is min-max with a very simple addition. If we are at a max node and our current score is greater than the parents score, which is a min-node, then the parent will not choose us as its move regardless of what us we discover. Therefore instead of exploring the potentials of the rest of our children we can exit out. Additionally if we are at a min-node and our score is less than the parents score, then the parent will never choose us. This means that exploring the rest of our children will not have any impact on our parents decision, allowing us to stop searching without any information loss.

To implement alpha-beta we used the same recursive solution as minimax. However, we added another parameter to the function which was the parents score. When the function was iterating over its children it would use the parents score to check if it was able to prune. If so it stopped exploring its children and returned its current score. We added another optimization with the win checker. Before we explored our children we would check to see if either team won. If a team won then we returned either 999 if we won or -999 if the opponent won. This prevented problems with the heuristic where the AI would go for the win in two rounds before blocking the opponent from winning this round. It also cut down the search time significantly at higher search depths whenever the opponent was one move away from winning.

2.3 Alternate Pruning

In addition to alpha-beta we also experimented with a nearest neighbour pruning algorithm. To do this, we utilized the same recursive strategy as in min-max and alpha-beta. However instead of searching all of the nodes children we searched them based on a probability function from calling the heuristic function on them. If we were a max node we favored high scoring nodes and as a min node we favored lower scoring nodes. This allow nearest neighbour to search far deeper in the same amount of time as alpha-beta. Additionally we incorporated alpha-beta pruning into the nearest neighbour algorithm.

2.4 Heuristic Functions

The heuristic function is based on the one describe in the book[1]. The basic structure of it is to find rows(vertical, horizontal, and diagonal) which are either only occupied by X or only occupied by O, sum the total for each, and return the difference between the two. If the team querying is X, the result is X - O and if the querier is O, O - X is returned.

The singly owned rows are found iteratively, with each type of row checked separately. First, the twelve possible vertical are checked. For each vertical row with only X, the total for X's free rows increased by one. The same happened for O in a separate variable. The same occurred for diagonal, both arcing left and right, and finally horizontal. Every possible horizontal was checked, so a single X on a row would increase the score by four points. The reasoning being that horizontal wins had to be blocked earlier, when only two in a row, so were considered greater threats. Another reason to check every possibility is that skipping every four could have the heuristic miss some possible wins since it would appear the other team had blocked the way, such as in the situation of O-XX-O. Unless the heuristic matched perfectly with -XX-, it would fail to see the potential win.

2.5 Win Checking with Resolution

There were six different statements used for the win checker. The first two had to do with the identity of the location. isX(a,b) was true if the piece at the location (a,b) was an X. Similarly, isO(a,b) was true if the piece at location (a,b) was an O. The next four statements all had to do with the relationship between the points (a,b) and (c,d). vertical(a,b,c,d) was true if the points were directly on top of each other. horizontal(a,b,c,d) checked if the points were directly next to each other. diagonalLeft(a,b,c,d) checked for two points being directly next to each other in the shape of a backslash while diagonalRight(a,b,c,d) was true if the two points directly made a forward slash.

There were eight queries we attempted to prove in order to prove a win. The first four are the four ways X can win. The equations for O look identical except all the isX's are isO's. The negated versions of the X checks look as follows:

$$\begin{gathered}\neg vertical(m,0,m,1) \vee \neg vertical(m,1,m,2) \vee \\ \neg vertical(m,2,m,3) \vee \neg isX(m,0) \vee \neg isX(m,1) \vee \\ \neg isX(m,2) \vee \neg isX(m,3)\end{gathered}$$

$$\begin{gathered}\neg horizontal(n,m,o,m) \vee \neg horizontal(o,m,p,m) \vee \\ \neg horizontal(p,m,q,m) \vee \neg isX(n,m) \vee \neg isX(o,m) \vee \\ \neg isX(p,m) \vee \neg isX(q,m)\end{gathered}$$

$$\begin{gathered}\neg diagonalRight(m,0,n,1) \vee \neg diagonalRight(n,1,o,2) \vee \\ \neg diagonalRight(o,2,p,3) \vee \neg isX(m,0) \vee \neg isX(n,1) \vee \\ \neg isX(o,2) \vee \neg isX(p,3)\end{gathered}$$

$$\begin{gathered}\neg diagonalLeft(m,0,n,1) \vee \neg diagonalLeft(n,1,o,2) \vee \\ \neg diagonalLeft(o,2,p,3) \vee \neg isX(m,0) \vee \neg isX(n,1) \vee \\ \neg isX(o,2) \vee \neg isX(p,3)\end{gathered}$$

The algorithm starts by The algorithm looping through the game board, creating the knowledge base. The knowledge base consists of separated, simple statements with constants, such as isX(0,1) or vertical(0,1,0,2). Once the knowledge base is defined, the algorithm tries to prove each of the eight previous statements separately, moving on to the next if the previous one fails. As soon as one of the queries is proven, the winner is returned.

The resolution algorithm was optimized to help save on time, so is not the full resolution algorithm, but instead a derivative that still works because of the nature of the set up of the algorithm. Resolution works as follows. A set is created that initially holds only the query clause. The program loops until it either gets an empty clause or no new clauses. Inside the loop, each statement in the knowledge base is resolved with the query set, which includes unifying the variables. Once all the resolutions are attempted and stored in a temporary set, they are checked. If there are no sets returned, the algorithm knows the query cannot be proven, so moves to the next query. If there is an empty statement returned, then the query is known to be true so the algorithm returns the answer. Otherwise, the query set is replaced with the temporary set and the loop is started again. When resolving two statements, the algorithm attempts to find a negated version of the knowledge base in the query. If found, the negated statement is removed, unification is run, and the new query statement is returned. Otherwise, nothing happens.

The unification algorithm is also shortened to increase speed. The unification algorithm simply looks at the knowledge base query and the negative of it in the query statement. Each variable is checked for differences in the variables. If the one in the knowledge base is a constant and the one in the query is a variable, then the variable is replaced with the constant throughout the query statement.

There were several optimizations in both algorithms which were used to increase speed. First, in the resolution algorithm, the knowledge base is only resolved with the query. All the statements in the knowledge base are positive and singular and all the query statements are negative, which means that trying to resolve knowledge base with knowledge base or query with query would be a waste of computing power.

The next optimization involves what is in the query set during each loop, which is only what was resolved from the last loop, instead of everything. Because the knowledge base only ever has one statement and can only resolve itself away in a query statement leaving another completely negative query statement, there is no reason to keep queries from previous groups. Leaving previous queries would only cause work to be repeated several times. By only resolving the latest, the algorithm was sped up significantly.

The final optimization was in unification. All the statements were very

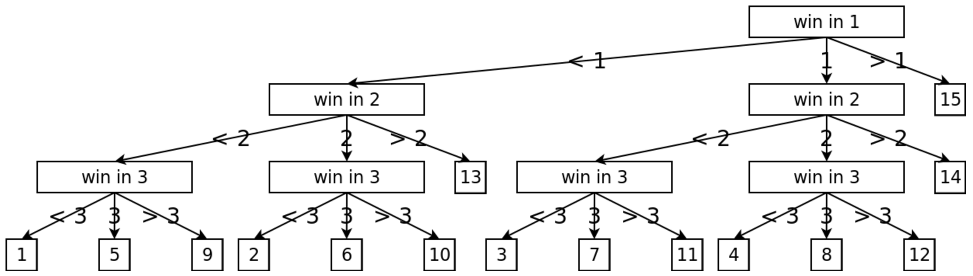

Figure 2: decision tree for classifier

simple, just functions with either variables or constants inside and possibly connected with or statements so there was no need for all the other functionality that unification usually has to deal with. Overall, all the optimizations made the win checker far faster than the original with the full algorithm implemented.

2.6 Game Evaluation with Classifiers

The classifier that was chosen was a decision tree as shown in figure 2. The tree is evaluated on each team and a rank is returned from one to fifteen. Fifteen is the best and one is the worst. The data used to determine the tree structure was taken from situations in the hopes of more accuracy since how good a state is can be unclear.

The first situation was that there are at least two ways to win with one move, meaning the player will win in their next turn. Next, if there are at least three ways a player can win in two moves, they have a greater chance of winning and finally if there are more than three ways a player can win in three moves, they have a decent chance of winning.

Next, if there is one way the player will win in one, two ways a player will win in two, or three ways a player will win in three, they are doing moderately well, but not a good. Anything else and they are in poor condition. That structure will look for what rank the player is in based on the previous states. The tree is applied to both players and then the return is the current teams rank minus the other teams rank.

2.7 Reinforcement Learning and Games

The neural network used was three layers. The input layer has 48 nodes, each representing a location on the game board. An X is represented with a 1, and O with a -1, and nothing is represented with 0. The middle layer is composed of 40 nodes. 40 was chosen because it was guessed that it would be a good balance between speed of computation and accuracy of the answer. The output layer consists of two nodes. The first represents, on a scale of 0-1 of how good a state is for X and the second represents the same for O. All nodes in one layer are connected to all nodes in the next layer and the function used in the middle and output layers is

$$R(j) = \frac{1}{1 + e^{-x}}$$

where x represents the sum of the inputs and R(j) is the output of the node j.

The algorithm used in evaluating a game state starts by filling in the input layer based on the given game board. Next, for each node in the inner layer, all the values of the input layer are summed, factoring the weights for each, and fed through the above function. Finally, the same happens for the output layer, it takes in the weighted sum of the inner layer nodes and applies the function to get an output. Then, depending on which player is asking the question, the relevant state evaluation is returned.

To learn the neural net weights to use in the algorithm, a learning environment was set up. The neural net was given an initially random set of weights. The environment would loop through 5 million games, having the neural net play against itself. Inside a game, there was another loop, which was run once a turn. In a turn, a player would first evaluate the current game state, then loop through each legal move, running the evaluation function again to find the next best move. When the best move was chosen, an update rule would update the neural net using a back propagation algorithm. Back propagation was also used at the end of each game. In the end, the final neural net was output to be used by the main algorithm in the program.

The back propagation algorithm was used to update the neural net between turns and after each game. The algorithm was the same for both in game and after, the difference was the inputs. During the game, the input was the current game state and the expected output was the output of the next game state. This served the purpose of tying a sequence of moves together. Then, at the end of the game, another run was done which used the final game state as the input and the correct output as the expected output, so 1,0 if X won and 0,1 if O won. Because of the in game updating, the final change at the end of the game would propagate through the entire sequence, so that instead of just knowing that the end was good, the net could determine good states earlier.

The back propagation is similar to in regular neural net updating with one exception. The algorithm is only run once, instead of over a series of examples several times. That is accomplished through the learning algorithm with turns being the different examples. The algorithm first find the output given the input. It compares that output to the expected output, with the difference being the error. Then, the error is propagated back through the net, all the way to the input layer. The weights are updated based on the equation

$$w_{i,j} += \alpha * R(i) * E(j);$$

where $w_{i,j}$ is the weight, α is the learning rate, R(i) is the output of that node, and E(j) is the error of the node at j.

While in the main game, we use minimax searching, we only used one round look ahead for the learning for a couple of reasons. The first was speed. By not searching a huge number of states, only a few are looked at, making the training very fast, running the 5 million matches in only one hour. We could skip the search because we are training the neural net to be a heuristic, so any node needs to be evaluated and whether it is higher up or lower down does not matter. The other reason was for the update rule. If we implemented minimax, the expected output in game would not have made as much sense, possibly weakening the net. Since we can use any search depth, we may have had a net that only worked well at one of them.

3 Experimental Methods

As we were developing the various AIs we did a lot of human versus AI testing to validate that it was picking optimally and could identify potential wins. However we choose not to collect data on this testing as human error frequently resulted in easy wins for the AI. Additionally by the end it became difficult even with multiple humans working together to present the AI with a challenge worthy of it. Therefore our experimental data was taken with the AI playing itself. Against the players who had random attributes such as the random player as well as the nearest neighbour (which choose the nodes to search based on a probability distribution) we ran a hundred matches for our data. The other AIs had no random attributes and therefore would play exactly the same game, so we only ran one official test game for each combination of search methods, heuristics, and search depths for both X and O.

We also ran a test to determine the best neural net. By varying the in game learning rate and post-game learning rate, we were able to get a total of 42 neural nets to try and compare to each other. Each neural net was given the same random seed in order to only test the different learning rates.

4 Results

There were 42 combinations of learning rate used to test the neural nets. For the in game rate, the rates .01, .02, .03, .04, .05, and .1 were attempted and for the post-game rates, .01, .03, .05, .07. .1, .12, and .15 were used. These numbers were chosen somewhat arbitrarily, but it was assumed that the post-game rate should be higher than in game. There are two graphs summarizing the results of running the different learning rates. The first, shown in figure 3 shows the average win rate compared to the other nets given each learning rate, averaging over all the post-game rates. The second graph, shown in figure 4 shows the opposite, the average win rate compared to other nets for each post-game rate where the values are averaged over the in game rate.

Another test we ran was to run each heuristic against the random player for 100 games with different search depths. This produced several results, including general heuristic effectiveness, improvement of agent given larger search depths, and average time taken to compute turns given a search depth. Figures 5, 6, and 7 show the results against random for the basic heuristic, classifier, and neural net respectively. The timing results are shown in figure 8. The times are taken from running minimax with no pruning.

The last test checked different times for minimax and minimax with alpha beta. The results are shown in figure 9 and show results for minimax and minimax with alpha beta.

Finally, the results of running the heuristics are as follows: the base heuristic beats the classifier and loses to the neural net and the neural net also beats the classifier. These results are taken using a search depth of five.

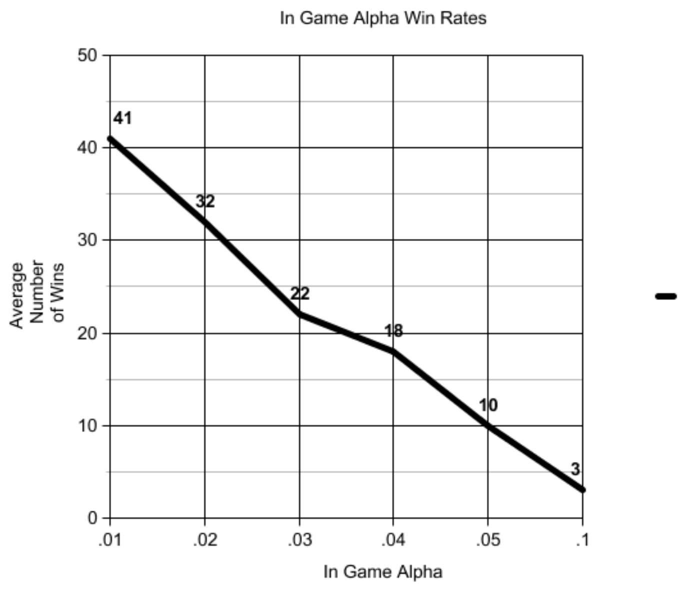

Figure 3: average win rates by in game learning rate

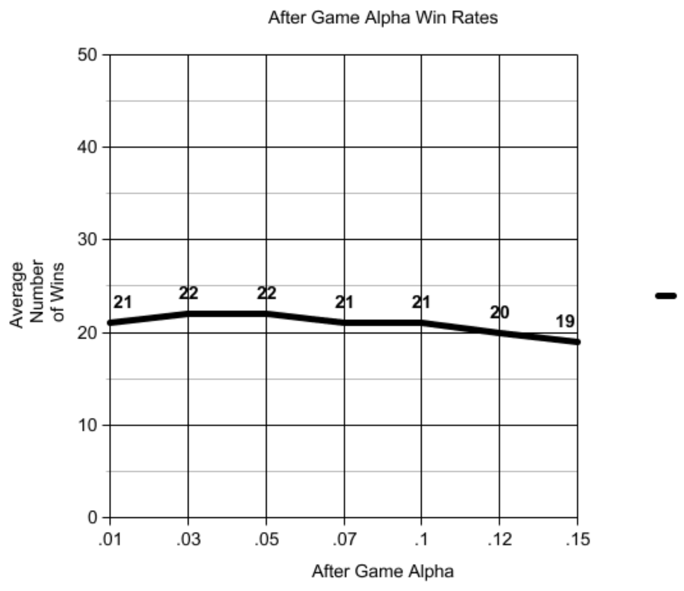

Figure 4: average win rates by post game learning rate

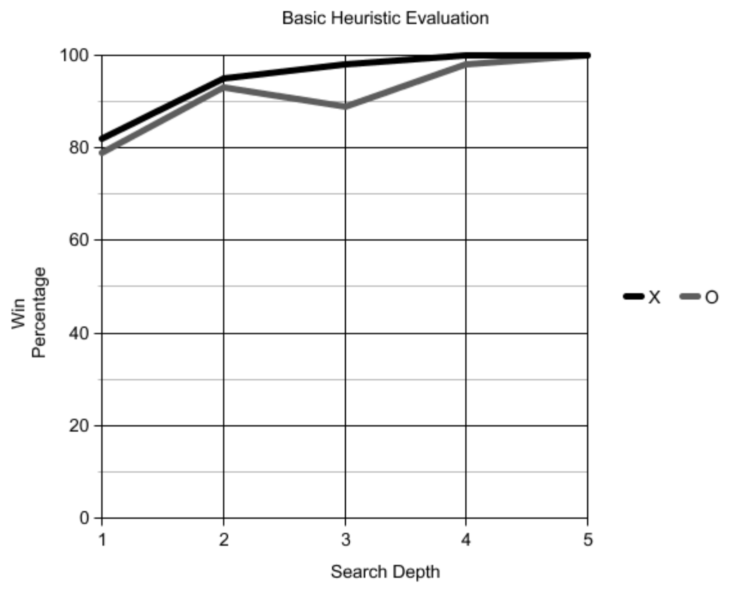

Figure 5: average win rates for the basic heuristic

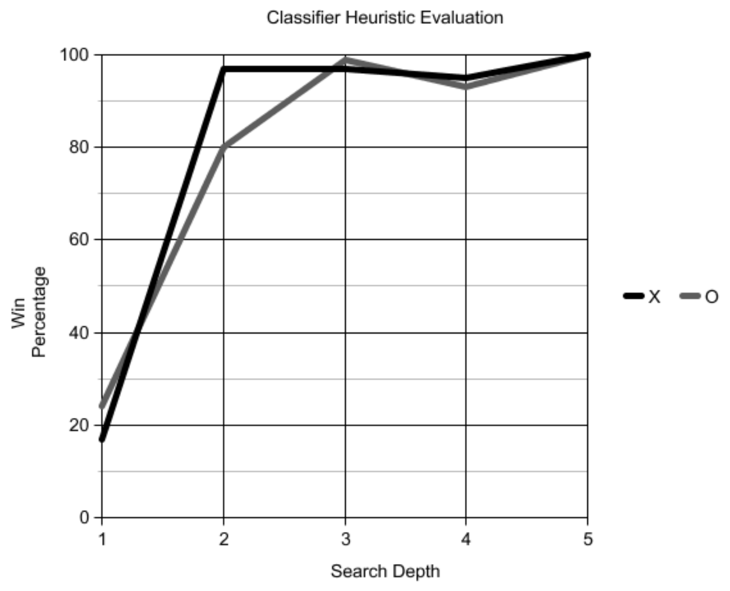

Figure 6: average win rates for the classifier heuristic

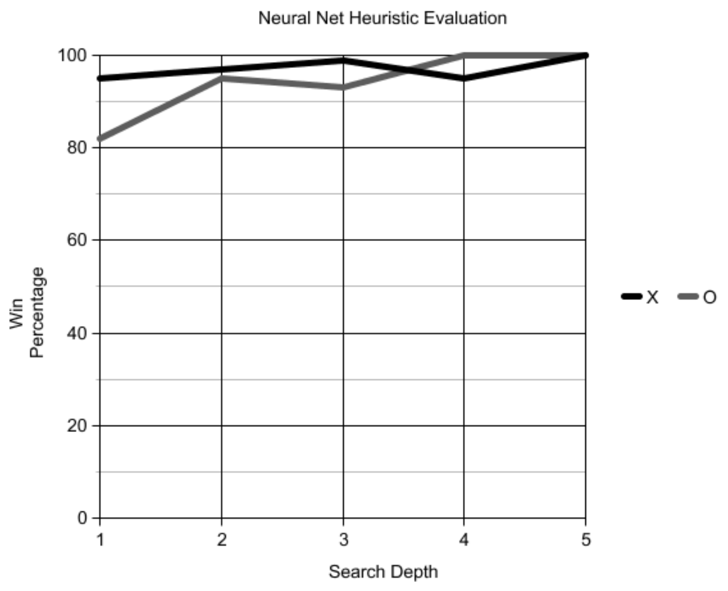

Figure 7: average win rates for the neural net heuristic

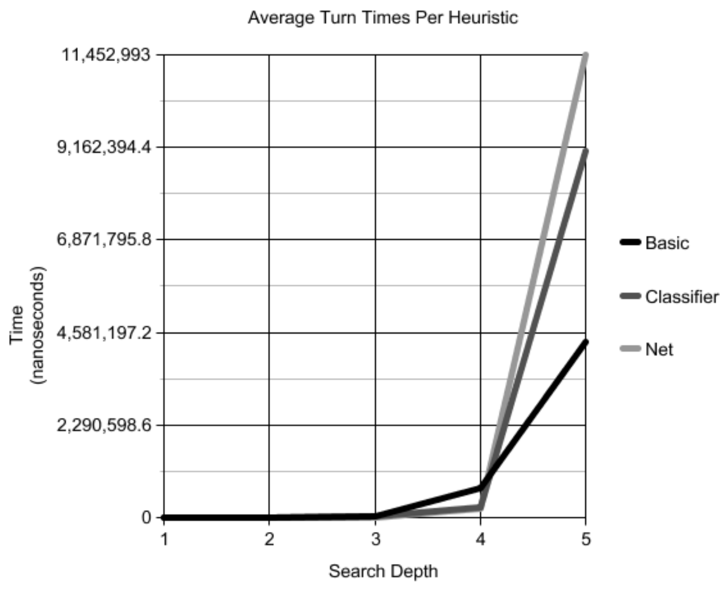

Figure 8: average move time for each heuristic

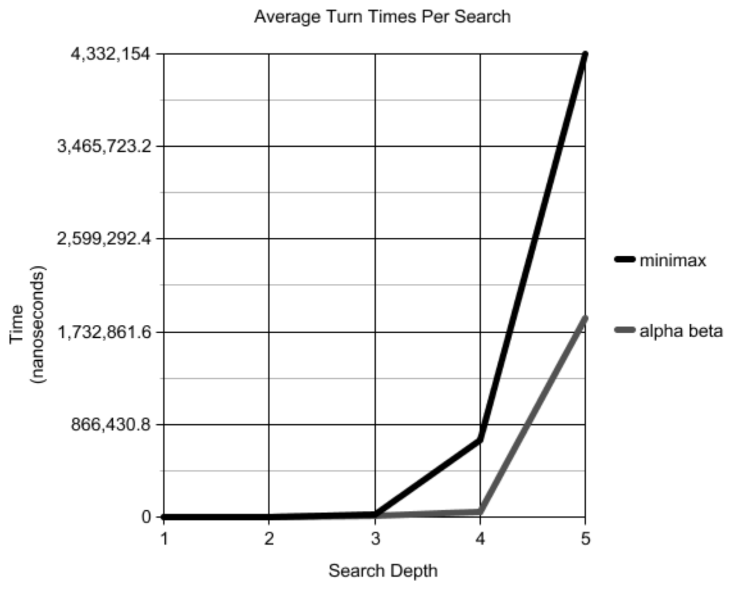

Figure 9: average time for a turn using the basic heuristic

5 Discussion

The first and most obvious result is how much the search depth affects the system. As seen in figures 5, 6, and 7, increasing to just a search depth of two made a drastic difference for the classifier and noticeable differences for the other two. The search depth of one did show some other useful results too.

The pruning also makes a huge difference. As seen in figure 9, you can see that initially, the two are very similar in times but as the search depth increases, the pruning takes out a huge chunk of time. Alpha beta is essential in order to increase search depth.

Our original idea for the classifier seemed to have backfired. When run against random on a search depth of one, it actually did worse than random. Because the classifier is based on later game data, it would make sense that at the beginning of the game it does very poorly. Searching even to a depth of two helped a large amount, which seems to also suggest that the classifier is better as a late game heuristic, not an early one.

Another interesting result is in the neural net training. When looking at the learning rate post game, the results are not definitive enough to make any useful conjectures, but the in game learning rate shows a definite trend. It seems to suggest that if the learning rate was lower it could conceivable do even better. A maximum does not seem to be obvious. The neural net that did best had an in game learning rate of .01 and a post game learning rate of .1. This leads us to further believe that the post game learning rate is much less important than the in game one since .1 was not a maximum for the post game rates.

Based on empirical data from playing the final result, I would say that the neural net plays the best game, especially as X, which is backed up by our results. This was a very interesting result because getting a good neural net the first time did not seem extremely likely. It was under the least guidance and I believe because of that it was able to really evaluate the problem without the error we had.

6 Conclusions

We learned many new techniques in doing this project. At the beginning almost nothing was known about artificial intelligence or how it can be used for game playing. As the semester progressed and as more of the project was completed, more and more knowledge was accumulated until we created an AI system which can actually play a respectable game, however basic theory was not the only thing we learned.

One of the largest lessons for this project had to do with efficiency. A couple lines of code can save seconds. A faster algorithm can save minutes. In many classes, speed is not much a concern, just the algorithm. For artificial intelligence systems, speed is one of the main considerations. You can play a perfect game, but if your program needs to think about it for days, then it is not a good solution. There are many considerations to be made for an algorithm and several improvements that are now obvious.

One of the main things to work more on is the neural network. We tested different learning rates, which helped to make a better net, but there were other parameters we did not have time to play with, such as the number of nodes in the inner layer, testing two independent nets against each other, and adding more, intelligent, input parameters. In future work, playing with all these changes could yield a much better neural net.

Overall, the system behaved better than expected and provided us with the tools to improve our current system or apply our newly formed skills on a whole range of situations. While the specific results may not greatly impact the artificial intelligence community now, the experiments taken can be used as the basis for more meaningful and impacting work.

References

[1] Stuart Russel and Peter Norvig. *Artificial Intelligence A Modern Approach.* Pearson Education Inc.

[2] Claude E. Shannon. Programming a computer for playing chess. *Philosophical Magazine*, 41(314):1–18, 1950.

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