Playing Chinese Checkers with Monte Carlo and Minimax Search

Jenny Huang and Josh Klukas

May 4, 2021

Abstract

A python program was developed to play two-player Chinese checkers using Minimax and Monte Carlo search. Minimax with a weighted sum heuristic was found to be the most effective algorithm, whereas Monte Carlo guided by the same heuristic was found to be effective only when the playout policy of the opponent was known.

1 Introduction

Finding the optimal move in zero-sum games can be achieved through the use of search algorithms that explore the consequent states of a given move. Chess and checkers are two games that have been extensively studied in this respect and pose significant challenges due to the enormous (functionally infinite) size of their state spaces.

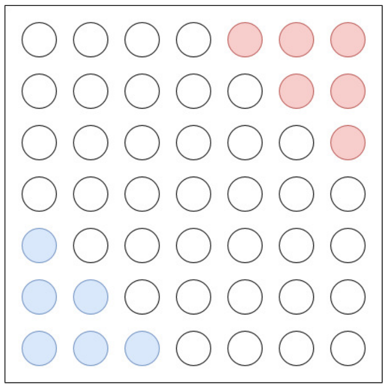

Figure 1: *Two-player Chinese checkers setup*

Chinese checkers is a similar zero-sum, perfect information game. However, it is unique in that there is no concept of capturing, and the goal of the game is to move all of one's pieces to the other side of the board. It is therefore not feasible to apply Monte Carlo search with randomly chosen moves. In this study, an attempt was made to compare the performance of a pure Monte Carlo search guided by a non-random playout policy to Minimax search. Due to time constraints and limited computational resources, the simple two-player case was considered as shown in figure 1.

2 Background

The literature on the use of search algorithms to solve Chinese checkers is limited compared to that on checkers and chess. The prevailing search strategy in the current literature is Minimax search with alpha-beta pruning, where each node of the search tree corresponds to a possible game state [8]. Enhancements to this algorithm through different heuristics have been explored with varying levels of success.

2.1 Minimax and Alpha-Beta Pruning

The Minimax search algorithm is useful for adversarial gameplay in which players will choose the move that is best for only them and not their opponents. This is done by assigning a value to each state using an evaluation function, which players base their next move on. In two player games, Minimax assigns each player the role of either "Max" or "Min". The Max player makes decisions that will maximize the evaluation function, while the Min player makes decisions that will minimize the evaluation function [7].

Alpha-Beta is a modified implementation of the Minimax algorithm that prunes away moves that do not give more information. For each node in the search tree, an upper bound, alpha(which starts at -infinity),and a lower bound, beta(which starts at +infinity), are maintained. The search is pruned if the value of the child of a max node is greater than or equal to beta or if the value of the child of a min node is less than or equal to alpha[7]. A common variation of minimax, called negamax, takes some heuristic to be the max value and the negative of the heuristic to be the min value[9].

Much of the literature discussed here concerning Minimax search and alpha-beta pruning are centered either on the strictly two-player game of checkers or two-player iterations of Chinese checkers. In multiplayer scenarios, Minimax is limited in its success because it cannot effectively handle the concept of coalitions or other scenarios in which some players may not behave in a purely adversarial manner. Alpha-beta pruning is also less effective in reducing the search space in multiplayer scenarios, since pruning is much more difficult when considering more than two players.

One method of managing the search space size discussed in the literature is the use of iterative deepening in conjunction with Minimax and alpha-beta as discussed by Zhao et al [9]. This was applied to the game of checkers where the heuristic was a sum of four factors, piece type(king or pawn), piece locations, potential to attack, and surrounding pieces. Iterative deepening was implemented by choosing a fixed time and searching while the amount of time spent in search was less than this fixed time. This is essentially an implementation of type A, where all nodes are considered to a certain depth. In type B evaluation, only some number of the best nodes are considered to the maximum depth. One way to implement the latter is by establishing an aspiration window at each level of the tree which establishes an upper and lower bound on searchable nodes. Papadopoulo et al.[1] describe a window which is linearly decreasing with the depth of the tree:

$$windowsize = (treedepth - currentdepth) * constant. \tag{1}$$

When cutting off the search, it is important to ensure that the heuristic function is not evaluated at a state where there is a pending move which could drastically shift the tide of the game. In the game of abalone, for example, attacking moves have high values but may put the attacker's pieces at risk. To avoid this, the search depth can be extended by some fixed amount[1].

Another way to save on time is to maintain transposition tables of past node move sequences. This helps avoid cycles and eliminates the need to recalculate the evaluation function for states that have already been reached [7]. This can be implemented using a hash table and Zobrist Hashing, in which one assigns a random 64-bit number to each piece at each position and XORs these numbers to obtain a unique key for each state [1].

Normally, the time complexity of Alpha-Beta pruning is $O(b^{\frac{3m}{4}})$, where b is the branching factor and m is the maximum depth of the search tree. However, if child nodes are sorted, a time complexity of $O(b^{\frac{m}{2}})$ can be obtained[7]. This is the aim of combined move ordering as described by Papadopoulo et al.[1]. The most useful sorting occurs in the first few levels of the search tree, where it is computationally cheap to calculate the evaluation function for each child node and sort them.

2.2 Monte Carlo

The Monte Carlo search algorithm, in which moves are evaluated by the relative success (number of wins versus losses) that proceeds from it, is useful for multiplayer games and other problems with large search spaces or in which evaluation functions (quantifying how "good" a move is) are difficult to define [7]. The typical implementation of a Monte Carlo tree search in game play involves selection of a game state (node), expansion of the tree with the addition of that node, repeated play-out of the game starting from that state and storing of results (wins for each player), and finally back-propagation of the results up to the root of the search tree [6]. Play-out from the selected node is done such that subsequent moves are chosen randomly; this has the effect of reducing computational demands. This is repeated a given number of times and the total number of wins s is taken to be the evaluation function score such that the more wins that arise from this move, the "better" it is considered to be.

Modifications can be made to this basic algorithm to increase the precision of the evaluation function. Progressive history is one such modification that takes into account the number of times a move has been played throughout the search by favoring less explored nodes [6]. Another variation of Monte Carlo involves using a probability function, P, to determine which node is selected as opposed to always selecting the node with the greatest score:

$$P = \frac{N^{\frac{1}{1}}}{\sum_k N_k^{\frac{1}{t}}} \tag{2}$$

where N is the number of times the kth node has been visited, and $t > 1$ is determined experimentally. Overall, Monte Carlo is a simple method of solving complex search problems and can be made more precise by the incorporation of other heuristic information.

2.3 Other Heuristics

Aside from Monte Carlo, the most common heuristics mentioned in the literature are variations of the distance of the farthest piece to the destination corner. For example, Liu et al.[5] use the simple heuristic

$$h(A) = (row_i - col_i) - (row_f - col_f) \tag{3}$$

where i, f, and A denote initial positions, final positions, and the two-tuple, (row, col), respectively. The downside of this heuristic is that it cannot tell if a bridge has been reached and will therefore ignore initially sub-optimal solutions that could potentially result in a "serpent formation", which is the optimal leapfrogging pattern in Chinese checkers[2].

Generally, the more information conveyed in the evaluation function, the more useful and precise the evaluation function is in determining the optimal move. He et al. [3] discuss the use of three heuristics within a single evaluation function as opposed to just one. These heuristics are weighted in the calculation of the evaluation function such that the two-player version is calculated as follows:

$$v = w_A(A_2 - A_1) + w_B(B_2 - B_1) + w_C(C_1 - C_2) \tag{4}$$

where A_i, B_i, and C_i are different heuristics for player i (in the two-player version, $i = 1$ or 2), and w is the corresponding weight value for a heuristic. These weights were tuned experimentally using function training. Several heuristics were tested in the evaluation function but the three most useful ones were the squared sum of vertical distances from the destination corner for a player's pieces (A); the squared sum of the horizontal distances from the vertical central line for a player's pieces (B); and the sum of maximum vertical advance for a player's pieces (C). A was found to be the most influential heuristic (with the highest weight value), while C was found to be the least important, suggesting that horizontal variance of pieces is less critical in influencing the outcome of the game [3].

Several improvements to Alpha-Beta pruning were discussed including iterative deepening, aspirations windows, transposition tables, and combined move ordering. The Monte Carlo heuristic can be applied in the absence of an explicit formula by running many random simulations and picking nodes which result in the most wins. Explicit heuristics should encourage bridge building by giving higher scores when pieces are closer together as in Equation 3.

3 Methods

A program was designed in python to carry out game play between several agents[4]. It uses three main classes which are described here.

3.1 Board Class

The game state is stored in a Board class which contains a two-dimensional matrix with zeros representing empty spaces and 1's or 2's representing player pieces. The Board also contains an array of Piece objects, which store the piece positions and which player they belong to. Children representing all possible moves for the current player are generated and stored in an array according to the following pseudo code.

Algorithm 1

```
function GENERATE CHILDREN(row, col, piece, board)
    for (newRow, newCol) surrounding piece do
        if (row, col) is unoccupied, (newRow, newCol) is occupied then
            newRow = row + (row-piece.row)
            newCol = col + (col-piece.col)
            if (newRow, newCol) is unoccupied then
                newBoard = board.update(player,piece,newRow,newCol)
                board.children.append(newBoard)
            end if
        else
            if (newRow, newCol) is unoccupied then
                newBoard = board.update(player,piece,newRow,newCol)
                board.children.append(newBoard)
            else
                newRow = row + (row-piece.row)
                newCol = col + (col-piece.col)
                if (newRow, newCol) is unoccupied then
                    newBoard = board.update(player,piece,newRow,newCol)
                    board.children.append(newBoard)
                    GENERATE CHILDREN(newRow,newCol,piece,board)
                end if
            end if
        end if
    end for
```

All children are generated at once, and each Board has a parent field which is updated for each child to allow for forward and backward traversal of the search tree. The main program allows for any two agents, including a human, to play against each other. To allow for quick testing, a random agent was created in the main python file, which randomly picks a child of the current game state.

3.2 Minimax Class

The Minimax class has parameters to store the root node, the best move, and the search depth in the tree. Its first method expands all nodes up to the search depth. There are several other methods that carry out alpha-beta pruning and minimax with two different heuristics. The first heuristic, h1, is the one shown in equation 4 in the background section. The weights for depths 1 and 2 were taken from the source literature[3] to be

$$(w_A, w_B, w_C) = (0.911, 0.140, 0.388) = (w_A, 0.15w_A, 0.42w_A). \tag{5}$$

An attempt was later made to verify these weights using the following weight tuning algorithm[3], where ϵ is some number close to zero, α is the learning rate, and the feature vector is

$$(A_2 - A_1, B_2 - B_1, C_1 - C_2) \tag{6}$$

Algorithm 2

```
1: function TUNE WEIGHTS($w$)
2:     while Norm(w-w$_{new}$) $>\epsilon$ do
3:         Play a game with both players following minimax, for each move:
4:         Store the feature vector in the matrix $\Psi$
5:         Store the minimax score in the vector, f
6:         $w_{new} = LeastSquare(\Psi, f)$
7:         $w = w + \alpha(w - w_{new})$
```

The second heuristic, h2, shown in equation 3, is used as a benchmark for testing of the algorithms. The reason for this choice is discussed in the experiment and results section.

3.3 Monte Carlo Class

The Monte Carlo class has parameters to store the root node, the player, the depth of each simulation, and the number of simulations. It has a method to implement pure Monte Carlo search with full playout. In full playout, one simulation is carried out for each child of the root node and a move is taken in the direction of the path with the least number of nodes that results in a win. One simulation is used, because gameplay is guided by one of the heuristics described in the previous section, so repeated simulations will yield the same result.

4 Experiment and Results

4.1 Greedy h2 as a Benchmark

An attempt was made to use the random agent as a benchmark for the other algorithms, but it was discovered that the random agent often failed to exit its own goal, resulting in draws. Therefore, following [3], h2 was used as a benchmark for the performance of Monte Carlo and minimax. In order to measure the performance of these algorithms after each game, h2 was also used to determine the number of moves necessary for the loser to reach its goal so that a larger number of moves implied better performance for the winning player. Figure 2 shows a histogram of this for 100 games of minimax with h2 and a search depth of 1 versus the random agent.

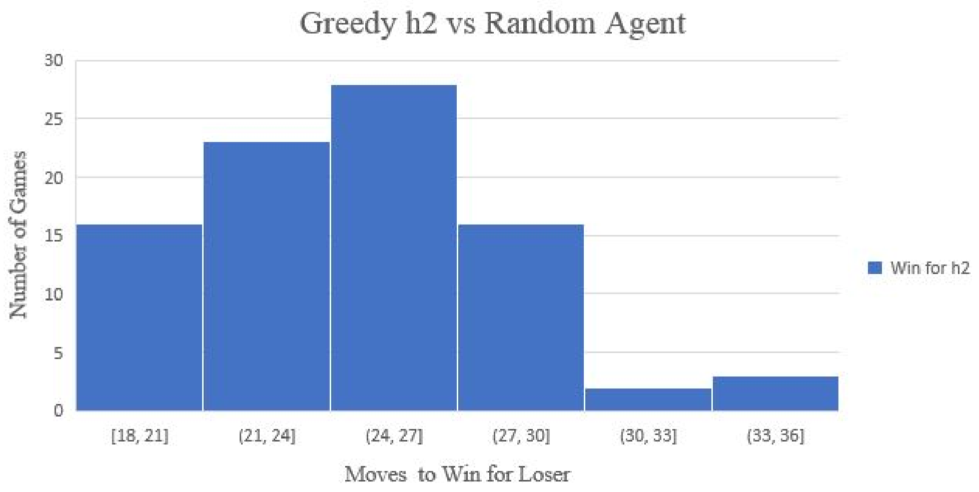

Figure 2: *Result of 100 games of h2 minimax with depth = 1 versus the random agent. The random agent lost every game that did not end in a tie.*

h2 was only used with a search depth of 1, because in 20 tests with a search depth of 2 against the random agent, h2 did not achieve a win in 200 moves. The reason for this is discussed in the discussion section.

4.2 Weight Tuning

The weight tuning algorithm shown in section 3.2 was applied to h1 with a search depth of 2 for 40 iterations. Figure 3 shows the convergence of the algorithm, $Norm(w - w_{new})$, with respect to the number of iterations. The weights with the best convergence were found to be

$$(w_A, w_B, w_C) = (0.244, 0.054, -0.988) = (w_A, 0.22w_A, -4.04w_A) \quad (7)$$

On comparison with the weights in equation 5, it can be seen that w_A and w_B are relatively close to those found in [3]. The reason for the discrepancy in w_C is that firstly, C was calculated as the maximum vertical displacement as opposed to the sum. Secondly, $C_1 - C_2$ was used to calculate h_1 as opposed $C_2 - C_1$, which explains the sign difference. These mistakes were only realized after testing had been done, therefore the h1 heuristic used in this study should be interpreted as only being comprised of the A and B components. That is, the contribution of the C component to the heuristic was negligible compared to A and B.

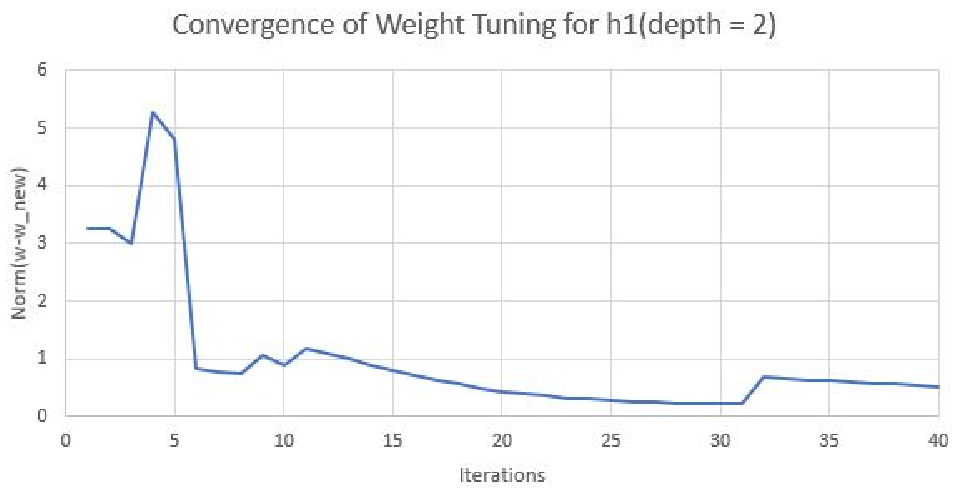

Figure 3: *Convergence of weight tuning algorithm for h1 with a search depth of 2. Each iteration denotes a game carried out to a win state.*

4.3 Performance of Minimax with h1 Against h2

h1 was tested against the benchmark for search depths of 1, 2, and 3. Due to time constraints and the excessive move evaluation time for h1 with a depth of 3, weights were not derived for this case, so the weights in equation 5 were used. Figures 4 and 5 show the results of the games played for depths of 1 and 2. It can be seen that h1 with a search depth of 2 performed much better than the greedy agent.

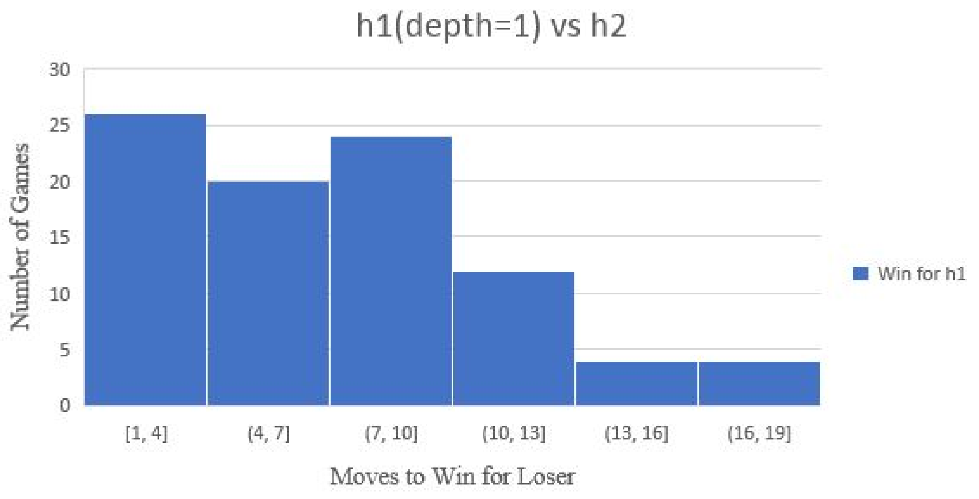

Figure 4: *Wins and losing moves in h1(depth=1) vs. h2 minimax playout.*

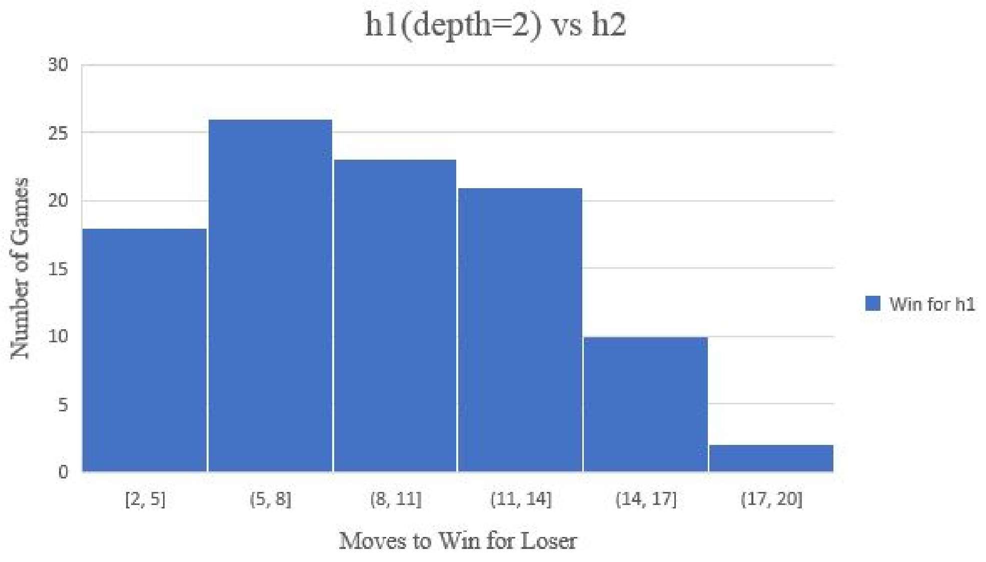

Figure 5: *Wins and losing moves in h1(depth=2) vs. h2 minimax playout.*

Unsurprisingly, in 20 tests of h1 with a depth of 3 against h2, h1 lost every time because the weights were not tuned. Interestingly, h1 lost by one move in each case.

4.4 Performance of Monte Carlo Against h2

Monte Carlo with full playout was applied, and simulations were guided by h1 for the max player and h2 for the min player(opponent). Due to the long move evaluation time, only 20 games were carried out against the benchmark. This is shown in figure 6.

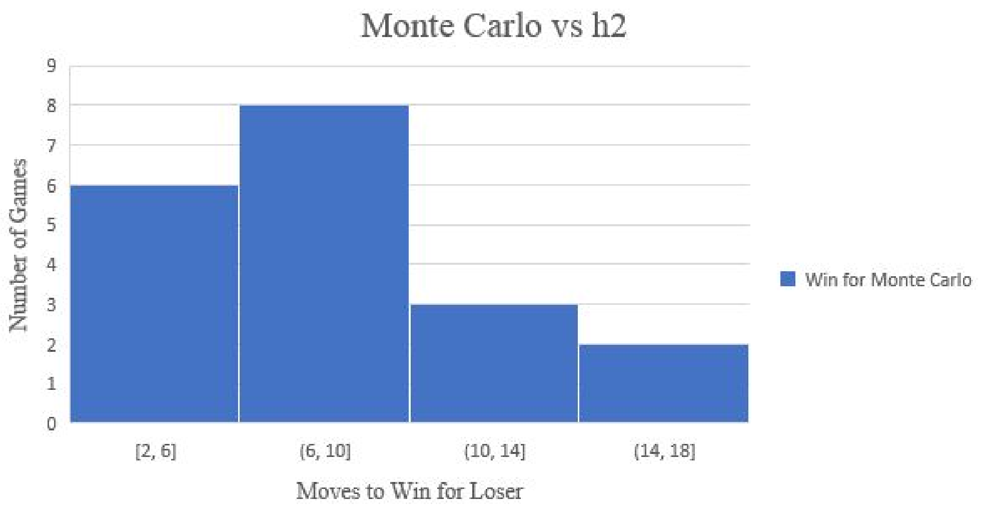

Figure 6: *Wins and losing moves in monte carlo vs. h2 minimax playout.*

A game was also carried out between Monte Carlo and h1 with a depth of 1, which resulted in a win for Monte Carlo. It is important to note that this was after changing the playout policy for the opponent to h1. If the playout policy was not changed, the Monte Carlo algorithm lost. Only one game was carried out, because subsequent games would have been the same as a result of the heuristics not changing. h2 is a caveat to this, because multiple moves often have the same heuristic value, so there are multiple possible games that h2 can play against any given agent.

5 Discussion

As mentioned in section 4.1, the h2 heuristic with search depths greater than 1 did not achieve a win against the random agent. This is likely because the heuristic only takes into account a single feature of the board which often results in multiple moves having the same heuristic value. As a consequence, when the agent picks a move because it sees it will later be able to perform a jump, it will not necessarily be able to identify this jumping move on the next turn. It follows that good heuristics must at least be capable of distinguishing between moves based on their evaluation functions.

A major limitation in carrying out tests was the move evaluation time of the algorithms. The suspicion arose that alpha-beta did not run any faster than minimax, despite evaluating fewer nodes. To confirm this, the move evaluation time was plotted with respect to the branching factor for the two algorithms as shown in figure 7. Tables 1 and 2 also show the time for node generation and search for the first move of the game.

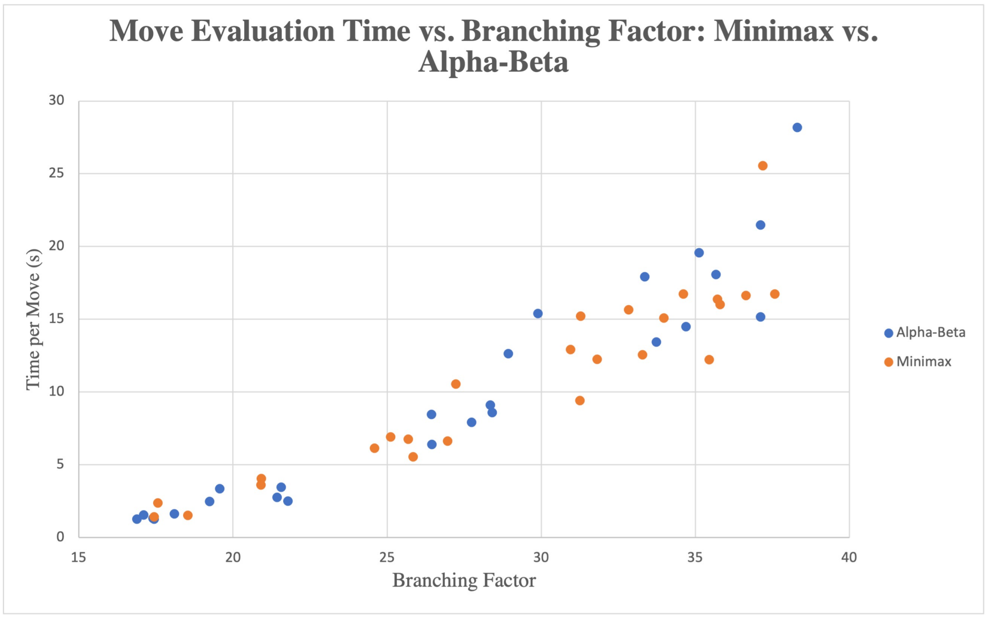

Figure 7: *Move evaluation time vs average branching factor of Alpha Beta compared to minimax for a search depth of 3*

Table 1: *Number of nodes generated and time of move evaluation for node generation and search(minimax)*

minimax			
depth	nodes explored	node generation time [s]	search time [s]
2	111	0.03	>0.001
3	1851	0.63	0.05
4	32127	10.54	0.88

Table 2: *Number of nodes generated and time of move evaluation for node generation and search(alpha-beta)*

alpha-beta			
depth	nodes explored	node generation time [s]	search time [s]
2	61	0.03	>0.001
3	465	0.63	0.01
4	2952	10.53	0.08

It can be seen that the majority of the time is spent generating nodes and that both algorithms spend the same amount of time in this respect. This is because all nodes are generated down to the search depth before the search is conducted. Although the alpha beta algorithm conducts its search up to 10 times faster than minimax, it saves little time overall as node generation is much more costly than evaluating the heuristic.

The efficacy of minimax and monte carlo search was shown against the benchmark, h2, however there was not enough time to do extensive testing of these algorithms against human players. Out of the small number of tests performed with humans, minimax with h1 and a search depth of just 2 showed great promise. Monte carlo did not perform so well against human players however. As shown in section 4.4, the monte carlo algorithm performs very well, assuming it knows the playout policy of the opponent. When the playout policy used in simulations does not approximate the policy of the opponent, monte carlo falls behind and eventually cannot find a win state. Because the monte carlo search in this study is so dependent on knowledge of the opponent's strategy, it might be better used in conjunction with a neural network as in [5].

6 Conclusion

A program was written in python to successfully play two-player Chinese checkers using the minimax and monte carlo search algorithms. Both algorithms were successful against a naive heuristic, h2, with minimax even showing great promise against human opponents. Because children could not be generated and pruned individually, the benefits of alpha beta were not observed, and experiments involving minimax and monte carlo were limited. Future work on the program should therefore be focused on allowing for children to be generated one at a time. This would allow for weight tuning of h1 at deeper depths and more extensive testing. It would also be valuable to test these algorithms extensively against human players to ensure that the algorithms are not overfitted as in the case of monte carlo. To make monte carlo more capable of playing against human players, it would be necessary to implement something like a neural network in order to learn the playout strategy.

References

[1] Antonios Chrysopoulos Athanasios Papadopoulos, Konstantinos Toumpas and Pericles A. Mitkas. Exploring optimization strategies in board game abalone for alpha-beta search. *2012 IEEE Conference on Computational Intelligence and Games*, 2012.

[2] Arthur T. Benjamin Auslander, Joel and Daniel S. Wilkerson. Optimal leapfrogging. *Mathematics Magazine, Vol. 66, No. 1,pp.14-19*, February, 1993.

[3] Sijun He, Wenjie Hu, and Hao Yin. Playing chinese checkers with reinforcement learning. 2016.

[4] Jenny Huang and Josh Klukas. Chinese checkers python code. `https://github.com/kluka029/Chinese-Checkers`.

[5] Ziyu Liu, Meng Zhou, Weiqing Cao, Qiang Qu, Henry Wing Fung Yeung, and Vera Yuk Ying Chungus. Towards understanding chinese checkers with heuristics, monte carlo tree search, and deep reinforcement learning. 2006.

[6] J. (Pim) A.M. Nijssen and Mark H.M. Winands. Enhancements for multi-player monte-carlo tree search. *Computers and Games*, pages 238–249, 2010.

[7] Stuart J. Russell, Peter Norvig, and Ernest Davis. *Artificial Intelligence: A Modern Approach, 3rd ed.* Prentice Hall, 2010.

[8] Qi Wang, Chang Liu, Yue Wang, and Danyang Chen. Move generation and search strategy research for computer game of checkers. *27th Chinese Control and Decision Conference*, 2015.

[9] Zicheng Zhao, Songze Wu, Jiao Liang, Fuyu Lv, and Changlong Yu. The game method of checkers based on alpha-beta search strategy with iterative deepening. *26th Chinese Control and Decision Conference*, 2014.