Department of Computer Science


Submitted in part fulfilment for the degree of BEng

# AN EVALUATION OF CLASSICAL AND MONTE CARLO GAME TREE SEARCH ALGORITHMS IN THE MULTIPLAYER DOMAIN OF CHINESE CHECKERS

Jake Phillips

07 / 05 / 2020

Supervisor: Peter Nightingale

Word count: 9739

# Acknowledgements

I would like to express my sincere thanks to my supervisor, Peter Nightingale, for his guidance and support throughout this project.

# Statement of Ethics

This project was written in accordance with the IEEE code of ethics and IEEE code of conduct. However, given the independent and impersonal nature of this project, there are no substantial ethical considerations.

# Table of Contents

# Table of Figures

# Table of Tables

# Executive summary

Traditional game tree search algorithms have been well studied and thoroughly evaluated in literature, primarily addressing Minimax [21] based methods which have dominated traditional zero-sum board games of perfect information such as chess [1] and Monte Carlo approaches [6] which have succeeded in mastering Go [2]. However, when exploring the domain of multiplayer games, the optimal approach still does not appear clear.

Various multiple adaptations have been proposed to address this problem, namely the, greedy [20], maxN [7], and paranoid [9] algorithms. Alternatively, the Monte Carlo Tree Search (MCTS) algorithm [6] has gained a lot of popularity in recent years for its ability to perform well in games without any specific domain knowledge. Despite this research [11], the optimal algorithm choice for specific game domains have not been well established or thoroughly examined.

In this paper, I compare the effectiveness of the three previously mentioned algorithms as a replacement for minimax in the multiplayer domain of Chinese Checkers. Additionally, I explore the impact that increasing the computational budget of the chosen algorithms has on their play quality, and examine the tradeoff between search time and performance. Lastly, I further the research of MCTS' adaptability to multiplayer games through applying the general monte carlo framework to Chinese checkers.

Chinese checkers [13] proposes an interesting domain to explore this problem due to two key characteristics: the fact that pieces are not removed from the board, and the availability of backward movement. Due to these two factors, Chinese checkers presents a prohibitively large search tree, which, when combined with the large board and number of players, makes it an ideal game to study.

In order to study this, I programmed the rules of Chinese checkers in Java, following an abstracted game state interface model [17]. In this model, an individual game state (the board) is represented, alongside methods which can move between these states (moves),

and finally a method to identify when a terminal state has been reached (a win condition). Notably, I decided to implement this game with a reduced board size in order to perform a greater number of tests in the limited time.

Within this model, an environment for performing tests was established. Firstly, I played each algorithm against itself in "self-play" at varying search depths to establish a rough idea of the average moves taken to win, and to get an initial idea of the effect increasing the depth of search has on the quality of play. Secondly, the algorithms were played against each other in a fair environment and different computational budget intervals in order to evaluate which algorithm performs best with a greater search depth.

From this, it was found that MCTS did not perform well in this domain, and failed to produce moves which were even remotely intelligent. As such, it was required to remove MCTS from the comparative play tests in order to not bias the results of the other algorithms.

From the self play tests, it was found that maxN's play quality did slightly improve with an increased search depth, with a lower average distance from winning, and a higher win rate at greater depths. However, possibly due to the fact the algorithm could not reach the second round of maximising nodes with a depth 3 search, the improvement was minute. Conversely, the paranoid algorithm's performance deteriorated with an increased search depth, possibly due to the fact that it played overly defensively. Similarly, the comparative play test results proved that maxN greatly out performed both paranoid and greedy algorithms at all depths.

Thus, the outcomes of the project are clear: MaxN and Paranoid both successfully offer a replacement for minimax in multiplayer games, MaxN vastly outperforms the paranoid algorithm, search depth has little impact on the performance of maxN, and an unoptimised MCTS does not compare to any of the other examined approaches.

# 1. Introductory Material

## 1.1. Abstract

As computing power has exponentially expanded in recent years, so has the corresponding capability and potential of computers to find novel and efficient solutions to previously perceived challenging problems involving a large search space. In this paper, I will explore and attempt to expand upon research into effective search techniques for zero-sum, deterministic, perfect information board games. In order to complete this research, the performance, computation time and play quality in relation to search depth was compared for the maxN, paranoid and Monte-Carlo tree search (MCTS) algorithms in the domain of Chinese checkers. It was found that whilst the MCTS algorithm did converge to a win, it performed extremely poorly in relation to the traditional algorithms. Additionally, It was found that MaxN vastly outperformed both Paranoid and greedy algorithms.

## 1.2. Background

Perhaps the most famous application of this area of research is IBM's Deep Blue chess computer [1] which used a minimax search algorithm with alpha-beta pruning alongside a comprehensive evaluation function [2] to succeed in championing the world chess champion, Garry Kasparov. Despite the victory, one of the developers of Deep Blue denies that this was artificial intelligence at all, but simply an optimised brute force method [3]. This type of algorithm is typically labeled as "Good Old-Fashioned Artificial Intelligence" (GOFAI) and covers the majority of classical search algorithms that do not utilise machine learning approaches [4].

Conversely, the Chinese game of Go offers a much more interesting problem given that it's prohibitively large search trees have large branching factors which render brute force approaches to be much less effective [5]. In 1993, Brugmann presented a fascinating application of the statistical Monte Carlo method for modeling an alternative algorithmic approach named the Monte Carlo Tree

Search (MCTS) [6]. This model is one which has extensively been built upon in the modern AlphaGo [5], resulting in a combination of MCTS and reinforcement learning to produce a program capable of defeating the top professional human Go players in 2017.

Despite the apparent success of the two aforementioned projects for mastering two-player games such as Chess and Go, when applying these algorithmic solutions to N-player games they tend to perform rather poorly. In 1986, Luckhardt and Irani proposed an extended minimax algorithm named maxN [7]. This approach assumes that each player is simply trying to maximise their perceived return without minimising the opposing players, and assumes the other players are doing the same. However, MaxN approaches still present a problem within large search spaces given that pruning a MaxN tree remains rather ineffective [8].

Sturtevant and Korf [9] proposed a solution to this hurdle of pruning through a reduction to a paranoid two player game by making the paranoid assumption that all other players have created a coalition against the root player and are playing with the intention to hinder their play. With this reduction, standard alpha-beta pruning is possible to optimise the search of the game tree. Despite this, Paranoid approaches typically do not guarantee an increase in play quality, as the algorithm tends to play overly defensively.

A more recent approach to multiplayer games is Schadd and Winands' best reply search algorithm [10], which assumes only one opponent is trying to minimise the root player's score, and plays to counter them. To this day, BRS remains one of the optimal traditional approaches to multiplayer games [11].

Conversely, as identified in [11], it appears that whilst traditional GOFAI algorithms are reaching their computational ceiling for multiplayer games of large branching factors, alternative branches of search methods such as the Monte Carlo tree search (MCTS) [5] still present much potential and opportunity for further optimisation. MCTS presents a modern, unique approach which does not depend on prerequisite game knowledge and is adaptable to N-players.

# 2. Literature Review

## 2.1. Chinese checkers

Originating in Germany in 1892 as a simplified version of the American board game, “Halma” [11], Chinese Checkers (Chequers) is a tactical board game which can be played by two, three, four, or six players. Each player is assigned ten pieces on pegs within their starting triangle with the objective of moving all ten pieces across the hexagram shaped board into the home section of the opposite player’s starting triangle. Players can move their pieces directly with a “step” onto any adjacent empty space, with a “jump” directly over any peg on the board, or with a chain of jumps over multiple spaced out pegs.

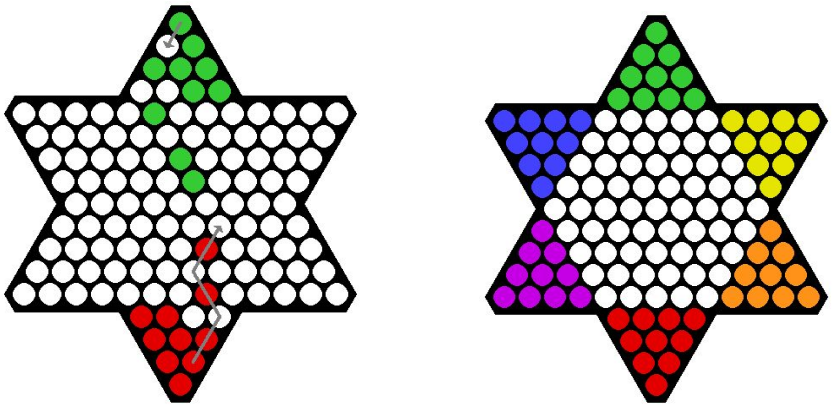

Figure 1: A Chinese Checkers board showcasing “Step” moves in green and a chain of “Jump” moves in red.

A game of Chinese Checkers can be broken into three distinct phases [12]:

1. The “gambit” phase, where the teams advance towards one another but do not interact.
2. The “melee” phase, where the teams interact with each other's pieces and eventually pass through one another.
3. The “packing” phase, where the teams separate and attempt to fill the target areas as fast as possible.

Unlike the English game of Checkers (Draughts), pieces are not removed from the board when jumped over. As identified by

Fonseca in [13] this results in an extremely large branching factor within the playout of the game given the large number of potential moves available for any given piece. Also, as a result of pieces being able to move both backwards and forwards, the search tree has an infinite depth. Due to these two factors, Chinese Checkers becomes a particularly interesting game to study algorithmic solutions.

## 2.2. Board representation

Chinese checkers is played on a hexagram shaped board made from a grid of hexagonal tiles which is somewhat more complex to represent than a square grid. The various different approaches to representing such grids in code is documented by Patel in [15]. Most simply, the board can be represented with “offset coordinates” where every row or column is offsetted so that the horizontal and vertical hexagons each have two variants.

Alternatively, the board can be represented as a slice along the diagonal plane at $x + y + z = 0$ on a cubic grid. This creates a rather complicated representation which is hard to visualize, however Patel [15] argues that it makes hexagonal grid algorithms much simpler due to the ease of implementing standard cartesian operations.

Another method for representing a hexagonal board is through doubled coordinates in which either the horizontal or vertical step size is doubled so that $(x + y) \% 2 == 0$. This has the added benefit of being able to be placed within an intuitive square grid which can be stored easily within an array.

Whichever method of representation is chosen, there exist algorithms to convert between each representation. As such, Patel [15] claims that if no complex board manipulation is required, one should opt for either doubled or offsetted coordinates, and to convert to cubic in order to perform calculations.

## 2.3. Game tree search

In game theory, a game tree is a directed graph whose nodes represent certain positions in a game (and typically the heuristic

value of that position), and whose edges represent the moves between positions [16].

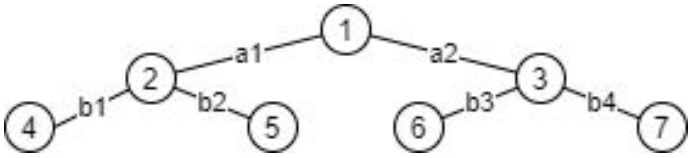


*Figure 2: An example game tree.*

In order to traverse such a game tree, certain information is required [17]: an initial board state, a set of move operators dictating how to transition between board states, an heuristic evaluation function to calculate the value of each board state, and a terminal test to define a win or lose condition.

## 2.4. Heuristic Evaluation Function

The play quality of any game tree search algorithm strongly depends on how effectively it can evaluate the value of any given board state. This is typically named the evaluation function, heuristic or utility payoff [18]. For board games, this evaluation function is generally composed of a linear combination of multiple features with corresponding weights such as: the sum of the distance of each piece to the goal, the sum of the values of the remaining pieces, or the sum of the value of the positioned pieces on the board.

$$e(p, w) = \sum_i w_i * f_i(p)$$

*Figure 3: Example general evaluation function to sum up the feature value, $f_i$, of position, $p$, multiplied by its corresponding weight, $w$.*

One proposed evaluation function for Chinese checkers by Clune in [19] is as follows :

$$E = 1.201 * f_1 - 3.000 * f_2 + 80$$

$$f_1 = fraction\ of\ marbles\ in\ goal, f_2 = total\ distance\ of\ marbles\ from\ goal$$

*Figure 4:* Example Chinese Checkers evaluation function.

### 2.5. Greedy Search

A Greedy search algorithm [20] is an extremely simple and intuitive approach for exploring a game tree that always exploits the most obvious and immediate best move. This results in an algorithm which thrives in situations where choosing the locally optimal moves also leads to the optimal global solution. Greedy approaches do not consider the moves of the other player and, as a result, tend to perform poorly against other algorithms.

### 2.6. Minimax Search

Minimax [21] is a well studied recursive depth-first algorithm used in two player games that produces a competent computer player by minimising the worst-case potential loss. The algorithm considers all the best opponent responses up to a certain depth, and selects the strategy such that the opponents best strategy results in a payoff as large as possible.

This is achieved by traversing the game tree up to a certain depth in order to evaluate the utility value of the board several moves in the future. This value is then back-propagated up the tree to the parent node where it is compared to the current utility of the parent. If the parent node is maximising, then the terminal node utility value replaces the utility of the parent node if it is greater than the current value, or has not yet been assigned a utility value, and vice versa should the parent node be minimising.

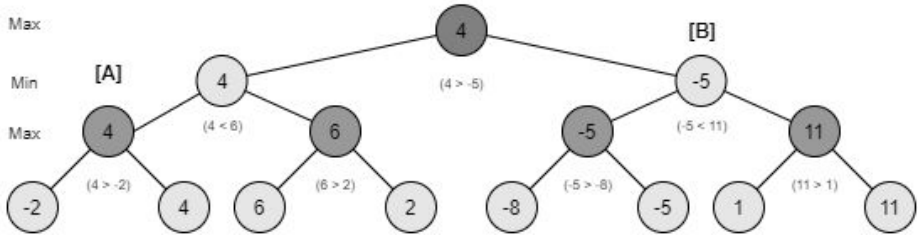


*Figure 5: Example completed Minimax game tree.*

As minimax is a depth first search, it begins it's evaluation at the leftmost node which is not a leaf node. This can be seen in figure 5 at the node labeled "A". Here, the minimax algorithm will be

maximising the value (4 > -2) of it's two children and propagating that value up the tree. This is repeated for all nodes at this depth until all the propagated values have been established. Next, the maximising player assumes that the minimising player will select the optimal move and choose the minimum value at this depth, this can be seen at the node labelled "B" where -5 < 11. This process is completed until the value of the root node is found, and the optimal move for the root player is identified.

### 2.6.1. Minimax Alpha-Beta Pruning

This process can be made more efficient with a technique called Alpha-beta pruning [22], which limits the number of nodes searched by pruning certain branches from the tree which are known to have no impact on the final evaluation of the search. Two new variables are added named Alpha and Beta which hold the corresponding best values of the maximising and minimising player. When transersing the tree, branches can be cut off which are known to have values worse than the current alpha or beta values. The benefit to pruning these branches is a greatly reduced computation time as less nodes are explored.

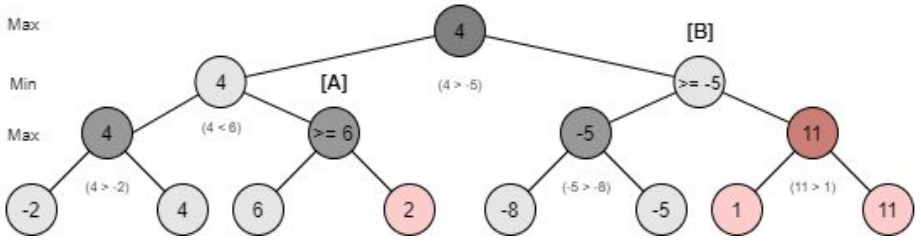


*Figure 6: Minimax game tree with Alpha-Beta pruning.*

This pruning process can be seen in figure 6. The algorithm begins as normal and propagates the value of the left most branch. However, at node A, once the value of 6 is explored, it can be established that the value of this node will be at least 6, given that the maximising player will choose the highest value of the children. As 6 is greater than the previously propagated value of 4, the minimising player does not need any more information, as they can safely propagate the value of four without the need of searching through all of [A]'s children. A similar process can be seen at node B.

## 2.7. MaxN Search

MaxN search (abbreviated from $Max^N$) is an algorithm presented by Luckhardt and Irani [7] which offers a method of applying Minimax search to games with more than two players. Rather than having a minimising and maximising player, each player simply attempts to maximise their potential return without minimising the other players, and assumes that all other players are doing the same.

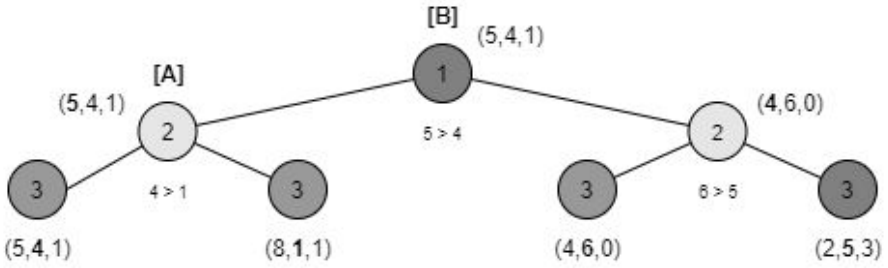


*Figure 7: Example MaxN Tree*

At each leaf node, an n-tuple is generated of the players utility values at that node, where n is the number of players. In order to transverse the tree to the root node, the corresponding player propagates up the child node which has the best personal payoff, and ignores the payoffs of the other players. As seen in figure 7, at node [A], player two chooses the child node (5,4,1) to be propagated as $4 > 1$, and at node [B], player one chooses (5,4,1) as $5 > 4$.

### 2.7.1 MaxN Pruning

The possibilities of pruning a MaxN tree were first explored in Korf's paper [8] which compared immediate, shallow and deep pruning on a maxn tree, and concluded that despite some slight optimisation from shallow pruning, "alpha-beta pruning is found to be effective only in the special case of two-player games". This is due to alpha-beta pruning techniques not being applicable to a MaxN as whilst deeper nodes do not represent the maxN value of the root, they can still indirectly affect it and thus cannot be safely pruned. Despite this, immediate and shallow pruning techniques still prove somewhat effective, as proved by Strtevant in [9]. Immediate pruning can occur when the chosen player's utility is equal or greater than the sum of all player's utilities.

Shallow pruning requires identifying the lower bounds (MinP) and upper bounds (MaxP) for each individual player's utility, and an upper bound for the sum of each player's score (MaxSum). From this, an upper bound of each remaining component at any particular node can be calculated based on the MaxSum - MinP.

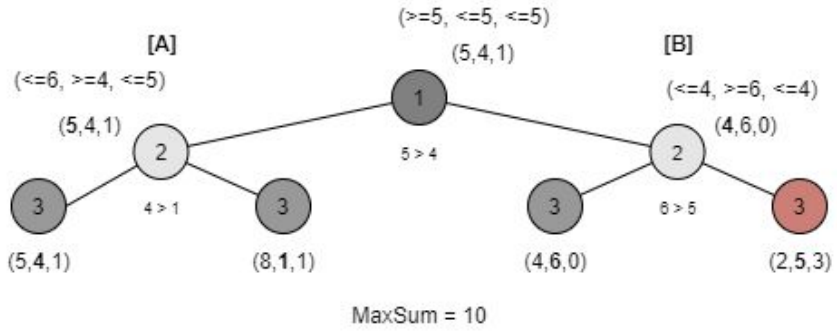


*Figure 8: Shallow Pruning of a MaxN tree.*

This can be used to prune child nodes in the case that the upper bound of the appropriate component is less than or equal to the lower bound of its parent node. In figure 8 at node [B], player one's score of 5 is greater than the parent's bound of 4 and thus there is no need to continue the search to player three. This is because player two can do no worse than 6, and as such, player one can do no better than 4. As a general rule, one can only prune when the value of player one, *a,* is greater than or equal to the MaxSum minus the current value of player two, *b*, ( *MaxSum - b <= a* ).

### 2.8. Paranoid Search

One proposed approach which solves the problem of pruning is to reduce a multiplayer game into a two player game as suggested by Sturtevant and Korf in [9]. This requires making the paranoid assumption that all other players are creating a coalition against the root player and, as such, are trying to minimise their score.

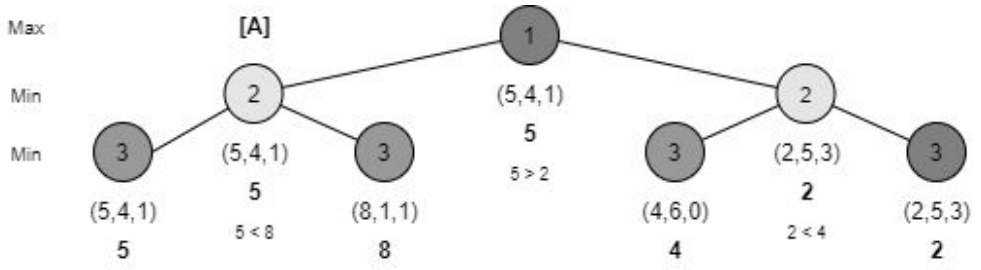


*Figure 9: Example reduction to a paranoid search tree of Figure 7.*

As seen in Figure 9, at node [A] player two (and three) is focusing on minimising player one's score and ignoring their own, whilst player one is trying to maximise their score. Through reducing the game tree to a simpler form of standard two player game, the tree becomes similar to a standard minimax format and can be optimised with appropriate methods such as alpha-beta pruning.

However, despite the obvious improvement in search efficiency proposed by this reduction, the rudimentary paranoid assumption that every player is actively playing against the root player is claimed by Sturtevant in [23] to result in overly defensive play which does not seize offensive opportunities and thus performs rather poorly against other algorithms such as MaxN. However, it is possible that this reduction in play quality is counteracted by the ability to search the tree to a greater depth and thus play more tactically.

### 2.9. Monte Carlo Tree Search

Monte Carlo Tree Search [5] (MCTS) presents an entirely different approach to selecting moves from a game tree which utilises reinforcement learning from random simulations rather than an evaluation function to guide the search. These simulations are combined to form estimated probabilities of winning for each child node, and the node with highest probability of winning at the end of the MCTS process is chosen. Thus, instead of brute forcing through millions of possible nodes, MCTS simply chooses the best current possible move from the current state of the game tree. This process can be broken down into four key stages [24]: Selection, Expansion, Simulation, and Backpropagation.

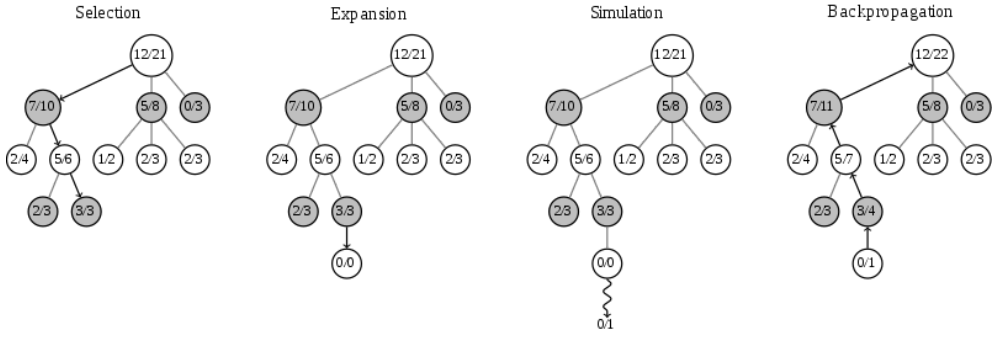


*Figure 10: Visual representation of each stage of the Monte Carlo Tree Search algorithm.*

Within the selection phase, a child selection is recursively applied in order to transverse from the root node until the most urgent leaf node is found which no simulation has been run from. Many different policy's exist for this selection which try to balance between the exploitation of paths with a high average win rate and the exploration of moves with very few simulations. This is typically described as the multi-armed bandit problem [25].

The most common approach of balancing these two aspects is with a method utilising *Upper Confidence Bounds* (UCB1) [26] from which the *Upper Confidence Bound applied to trees* (UCT) policy was formed [26]. This policy chooses a node of the game tree for which the value of the formula in figure 11 is greatest.

$$\frac{w_i}{n_i} + c\sqrt{\frac{\ln N_i}{n_i}}$$

$w_i$ , the number of wins for the child node after the i-th move.
$n_i$ , the number of simulations for the child node after the i-th move.
$N_i$ , the number of simulations for the parent after the i-th move.
$c_i$ , the exploration parameter.

*Figure 11: UCT formula.*

Once a node has been selected which hasn't been fully expanded, the expansion phase begins by adding a random child node to the tree. From here, the simulation phase begins from the newly added node and plays out a simulation of the game until completion. The moves chosen within this playout are dictated by the default policy which is typically a uniformly random move selection.

Lastly, the result of the playout is propagated up the tree with each node's statistics updated to incorporate the simulation along the way. From here, the process can be iteratively repeated to develop a sense of which paths present the best probabilities of winning.

As stated by Winands in [24], the main advantage of using MCTS is that the algorithm can be terminated at any point and still produce relatively strong move selections. Additionally, general MCTS methods do not require any specific game knowledge in order to produce strong results as they do not rely on evaluation functions to

evaluate the strength of each board position. Despite this, due to the random aspect of the simulation, the algorithm may ignore specific branches of the game tree which could be highly important in the later game. Similarly, the algorithm faces the issue of pursuing initially promising moves which result in a loss due to the lack of knowledge of the game. These flaws however, can all be mitigated through the incorporation of various enhancements to MCTS to make it more specific to the relevant application.

### 2.9.1. MCTS Improvements

Certain improvements, such as those evaluated by Browne and Powley in [27] or Banks in [28] can be made to the MCTS algorithm in order to shorten the required search time through the optimisation of node selection and improved accuracy of simulations.

Perhaps the biggest flaw of the general MCTS algorithm is it's slow exploration of game states when challenged with new domains. MCTS takes many rounds of random move selection in its exploration phase before beginning to find promising strategies. This can be mitigated through the use of the Rapid Action Value Estimation (RAVE) algorithm [29] which shares the value of actions across each subtree in order to form rough estimates of the value of each move. This approach assumes that it does not matter when a certain action is played in the game and as such, is less accurate.

Alternatively, a whole range of MCTS improvements come from incorporating specific game knowledge into the algorithm. One such example would be implementing a heuristic function to initialise the values of new positions in the search tree and increase bias towards selecting optimal moves.

In 2011, Nijssen and Winands proposed an enhancement to MCTS [30] which did exactly this. This playout search technique was found to significantly increase the play quality in games such as Chinese checkers [31], at the cost of slowing down the speed of playouts.

# 3. Aims and Motivation

## 3.1. Aims and Objectives

In this project, I aim to:

- Explore the effectiveness of different multiplayer game tree search algorithms as a replacement for minimax and to further research into the characteristics of each algorithm in the domain of Chinese Checkers.

- Explore the impact that increasing the computational budget of the chosen algorithms has on their play quality and to examine the tradeoff between search time and performance.

- Compare the performance of the chosen algorithms against each other in a range of board configurations to establish which algorithms perform the strongest at specified computational budget intervals.

- Further the research of Monte Carlo algorithms in the search domain of Chinese Checkers by comparing them against traditional algorithms and exploring Monte Carlo's adaptability to multiplayer games with a large branching factor.

## 3.2. Motivation

There have been numerous modern successes in developing algorithms that excel at traditional zero-sum board games of perfect information such as chess [1] and Go [2], yet little research has been completed into the performance of search techniques on multiplayer games with a large branching factor such as Chinese Checkers [13]. Whilst research has mostly moved on from exploring single player minimax based approaches to multiplayer games [7, 10, 11], there exists little research [23] into how these algorithms perform against each other with varying computational power.

Chinese Checkers presents two interesting aspects [14] that differ from traditional games: first, the checkers are not removed from the board and, second, the availability of backward movement and repetitive cycles of moves. These features, combined with the large board, lead to a prohibitively large search tree and a limited available search depth. In this paper, I wish to explore the impact that increasing search depth has on play quality in order to examine if a greater computational budget is necessary and if so, to what extent.

Additionally, as MCTS presents an interesting new and alternative approach to game tree search problems, I wish to explore the success of the MCTS algorithm in the domain of Chinese checkers to establish whether it offers an alternative to traditional approaches.

### 3.3. Expectations

Based on my findings in the literature review by [11, 23] I am expecting that in games where maxN is compelled to do a brute force search, the paranoid algorithm will be more effective. As Chinese Checkers has a large search space, I would expect maxN to struggle pruning in comparison to the paranoid algorithm. This assumption aligns with the findings of [23] which show that "paranoid widely outperforms maxN in Chinese checkers".

However, due to the fact that individual moves have very little impact on the final result of the game and that strategies to winning are not immediately obvious, I am expecting the winner of each game to be largely impacted by a random factor of luck in how the other players align their pieces. As such, I am expecting the search depth of the algorithms to have little impact on the overall play quality, as looking ahead more moves does not realistically provide much information.

Additionally, As discussed in [31], MCTS performs much better in Chinese Checkers with optimisations to incorporate domain specific knowledge. As this project does not explore these optimisations, I am expecting MCTS to perform rather poorly.

# 4. Methodology

Chinese Checkers was implemented in Java following an abstracted game state interface model [17]. In this model, an individual game state (the board) is represented, alongside methods which can move between these states (moves). Lastly, a method which can evaluate the board at any given state, and decide when a terminal state (a winning state) has been reached.

## 4.1. Board Representation

It was decided that working with a smaller board size (Figure 13), as suggested by [31], would decrease the overall playtime of each game which would enable more tests to be completed in an equivalent time frame. Additionally, with the smaller board only having six pieces for each player, the branching factor of each state is vastly reduced which means that the algorithms are able to lookahead to a greater depth.

The board was stored inside a two dimensional array using the “doubled width” hexagonal grid representation where large (17 x 25), and small (13 x 19) board layout arrays are stored in a local variable.


*Figure 12 : Doubled width horizontal board layout.* [15]

In this representation, the integer 9 represents an unplayable space, 0 represents an empty playable space, and 1-6 represents the players. The inclusion of an unplayable space allowed the board to be padded as to create the required doubled horizontal step size. Whilst this does waste a small amount of memory space, it should

not be a concern for the small scale of this project but presents a viable area of improvement for the future.

This layout was then mapped to a board object with an array of cells by instantiating the corresponding Cell objects and their respective Piece objects to the same positions as in the predefined layout. From this, any board state can be displayed with the toString method as a basic GUI.

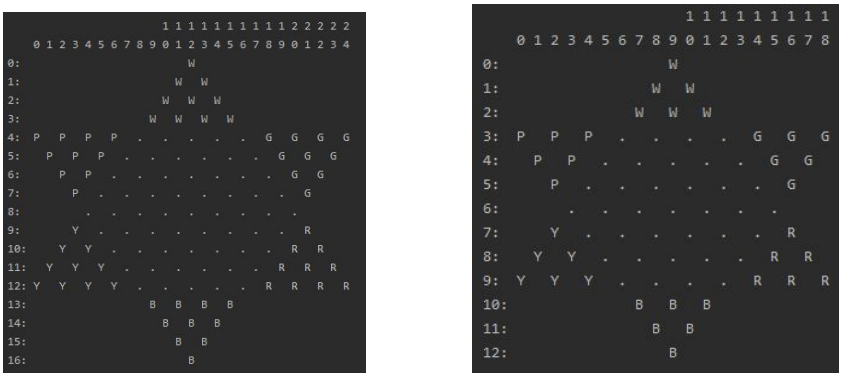

*Figure 13: Example console GUI outputs of the large / small six player boards.*

### 4.2. Move operators

First, a Move object class was established to store information about specific moves. Then, methods which take these Move objects as a parameter were established for moving pieces on the board, and also undoing a move on the board. This enabled the board to transition between states easily and established the basic framework of the game.

Following this, methods which can generate a valid Move Object from an initial Cell to a Target cell were established for both Steps and Jumps. Next, a method which attempts to move a piece from a specific cell to all neighbouring cells was established which returns a list of all the valid step and jump moves. This jump method was called recursively up to depth five to generate a list of all the possible available jumps.

After combining the previous two methods to generate a method which returns all the available moves from a specific cell, a method was created to locate all the pieces of a certain colour on the board.

With this, a final method could be created which gets all the available moves for every piece of specific colour on the board. With this information, it is up to the various implemented AIs to choose an optimal move from the returned list of all available moves to playout. By abstracting the specific game related logic of move generation, the AI class is able to be implemented generally without any specific knowledge of the game. This ensured robustness as the algorithms could be easily transferred to similar games.

### 4.3. Terminal test

Testing for an end game state simply requires a method which checks whether all of a specific player's coloured pieces are in the opposite player's base. These winning states are hard-coded into a variable and are compared against the board state after every turn.

### 4.4. Evaluation function

A simple evaluation function which considers: the number of pieces in the goal, the sum of the distance of each piece to the goal and the spread of the pieces on the board. The weightings of these features were manually tuned through play testing to that seen in figure 14.

$$e(p,w) = \sum_i w_i * f_i(p)$$

$$where: \ f_1(p) = Sum\ of\ Pieces\ in\ Goal, \ w_1 = 5$$

$$f_2(p) = Sum\ of\ Pieces\ in\ Base, \ w_2 = 5$$

$$f_3(p) = Sum\ of\ the\ distance\ of\ Pieces\ to\ the\ Goal, \ w_3 = 3$$

*Figure 14: Chosen Evaluation function.*

Calculating the distance between the goal point and a given piece on the board proved complicated with a doubled width board representation. However, as described in [15], a method was created which converts any coordinate to a 3D hex coordinate. From this, a method for cartesian operations to calculate the distance between two points could be established. Additionally, the sum of pieces in the goal / base could be counted by checking whether the distance to the goal / base was less than the size of the base.

### 4.5. Greedy Algorithm implementation

Implementing the Greedy algorithm was as simple as calculating the evaluation of the board after each potential move, and returning a list of all the moves with the best value. From this, a random move could be selected from the array of best moves to be played out.

### 4.6. Minimax implementation

The Minimax Algorithm was implemented as detailed by Russell and Norvig in [32] using the pseudocode shown in figure 15.

```
function ALPHA-BETA-SEARCH(state) returns an action
  v ← MAX-VALUE(state, −∞, +∞)
  return the action in ACTIONS(state) with value v

function MAX-VALUE(state, α, β) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← −∞
  for each a in ACTIONS(state) do
    v ← MAX(v, MIN-VALUE(RESULT(s,a), α, β))
    if v ≥ β then return v
    α ← MAX(α, v)
  return v

function MIN-VALUE(state, α, β) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← +∞
  for each a in ACTIONS(state) do
    v ← MIN(v, MAX-VALUE(RESULT(s,a) , α, β))
    if v ≤ α then return v
    β ← MIN(β, v)
  return v
```

*Figure 15: Minimax Pseudocode* [32]

Minimax is a recursive algorithm which transverses the game tree by simulating all the possible moves from the root node and ranking them respectively. A base case is required to terminate the recursive loop which is called when a terminal state is reached, or the depth of the search has reached 0. This base case returns the respective evaluation of the current board state.

Next, the algorithm loops through all the possible moves from the current position up to the defined depth by recursively calling the minimax function and backward propagating the evaluation of future board states up the search tree to the root node. This is achieved by traversing the tree through recursive calls up until a leaf node is found (the base case).

From here, the evaluation of the board state is compared against the current best value. Should the new evaluation present an optimal value, it is assigned to the respective alpha variable if maximising, or the beta variable if minimising. This simulated move is then undone in order not to disrupt the game board from the players perspective. Should the alpha variable be greater than or equal to the beta variable at this stage, the loop is broken in order to “prune” the respective branch of the tree. Finally, the algorithm returns the respective alpha or beta value representing the move with the optimal value.

Now that a method for evaluating a board state with the minimax function was established, a method for generating an array of optimal moves from the minimax evaluation was required. This was achieved by iterating through all the available moves for a specific player’s pieces and calling the minimax function to assign a value to value to each move. This method then sorts through the list and returns an array containing all the moves with the highest or lowest move value. From this, the AI can simply select a move at random from the list of optimal moves to be played out.

### 4.7. MaxN implementation

MaxN was implemented as described by [7]. I could not find any documented pseudocode to support this description, however my interpretation is as shown in figure 16.

```
function MaxN(state, player) {
        If TERMINAL-TEST(state) then return UTILITY(state)

        bestChild ← MaxN(first child, next player)

        for each action in ACTIONS(state) do:
                currentChild = MaxN(child, next player)
                If (currentChild[Player] > bestChild[Player] then
                        bestChild ← currentChild

        return BestChild
}
```

*Figure 16: MaxN Pseudocode*

MaxN's is essentially a stripped back version of minimax which can be extended to N-players. As such, the implementation is very similar to minimax but rather than switching between minimising and maximising players, every player maximises their own evaluation stored in an N-tuple.

As discussed in the literature review, pruning a MaxN tree is rather difficult as alpha-beta pruning is unavailable. Shallow pruning is possible in many cases, however is dependent on the evaluation function of the game states being constant-sum (where the sum of all scores at every node is constant). As the evaluation function in this implementation calculates the distance between the goal point and every piece on the board, it cannot be constant-sum and thus no MaxN pruning optimisations can be implemented.

### 4.8. Paranoid implementation

Similarly, a paranoid algorithm is a simplified version of MaxN which utilises pruning techniques from minimax. This was implemented according to [9] by following the same structure as figure 13. In this implementation, one maximizing player was chosen, and the rest were minimising. For the base case, the method returns the evaluation function for N players at index N-1. Every other aspect is exactly the same as the minimax implementation.

### 4.9. Monte Carlo implementation

The Monte Carlo algorithm was implemented as traditionally described by [6] and modernly by [33] in figure 17. Before the MCTS algorithm could be implemented, a game tree data structure needed to be established. In order to achieve this, a Node class was devised which stores: the move taken to reach this state, the parent node, a list of child nodes, the number of visits and wins from this state, and a list of untried moves from this state.

As not to interfere with the active game state, a method was devised to create a cloned copy of any given state in order to perform the monte carlo evaluation on. From here, MCTS could be broken down into four distinct phases, each with a corresponding method:

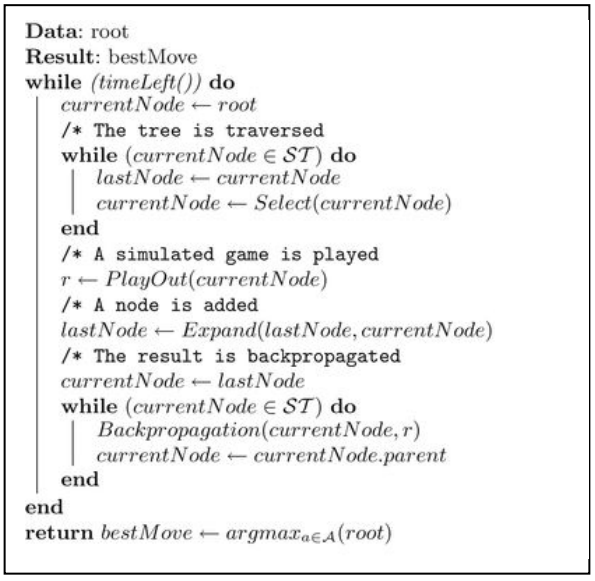


*Figure 17: Monte Carlo Tree Search Pseudocode* [33]

The selection method is called recursively in order to traverse the game tree until a node which has not been fully expanded is found. This selection process is guided by evaluating the UCB1 value for each child node and selecting the node with the greatest UCB1 value. Once a child node has been selected, a random move is selected to be expanded via a new method, and a child node containing this move information is added to the game tree.

Next, the algorithm needed to simulate a complete playout of the game until a terminal state is reached. Traditional MCTS performs this playout randomly, however due to the large branching factor of Chinese checkers, random playouts were found to provide no useful information as they would never converge to a winning position. Thus, a simple evaluation function was added to the simulation which prioritizes selecting moves which move towards the goal. This ensured the simulations reflected real play and provided meaningful information. Finally, a recursive method to backpropagate the results of the simulation up the game tree was added.

Now, a loop containing the four previously established methods could be built, which will perform one iteration of the MCTS algorithm. This loop can be called any number of times in order to build the game tree and further the algorithm's knowledge of optimal moves. With the MCTS algorithm completed, a method which

selects the best action from the root node was required. This iterates through each potential move from a state (the child nodes) and returns the node with the best win ratio to be played.

### 4.10. Experiment environment

An experiment object was established which plays out one game until completion and stores the relevant metrics. Firstly, it recorded the winning player and their moves taken to win, alongside the other players estimated number of moves away from winning. Most importantly, the average computation time for each move generation was also calculated. With these three metrics, a thorough analysis of the algorithms performance could be undertaken. Lastly, these results were exported to a .csv file in order to interpret the data using python's numpy and matplotlib libraries.

### 4.11. Comparison of algorithms

In order to make an accurate analysis of each algorithm's performance, multiple factors needed to be considered. Firstly, the algorithms were played against themselves at varying search depths and the impact on play quality of a greater search was explored. Additionally, these tests recorded the average number of moves required to win for each algorithm. This roughly reflects the play quality of an algorithm, as winning in fewer moves implies a stronger algorithm. With this information, a baseline was established for each algorithm's performance in a controlled environment.

It was found that whilst testing the MCTS algorithm against itself that it was incapable of playing intelligibly. Whilst the algorithm did eventually converge to a terminal state, it took an unfeasible amount of moves to do so. Because the performance of MCTS was so poor, it was found to skew the tests of the other algorithms by unfairly hindering them. As such, it was decided not to include MCTS in the comparative tests. With this established, multiple variations of the three other algorithms were played against each other on a 3-player board with a range of configurations and search depths.

# 4. Results

## 4.1. Self play-results

The first round of testing involved playing each individual algorithm against themselves with different computational budgets in order to establish a baseline of play quality at each search depth. The average moves required to reach a winning state (calculated by summing the average moves taken and the average moves from winning) for each algorithm is compared in relation to the depth of the search.

### 4.1.1. Minimax self-play

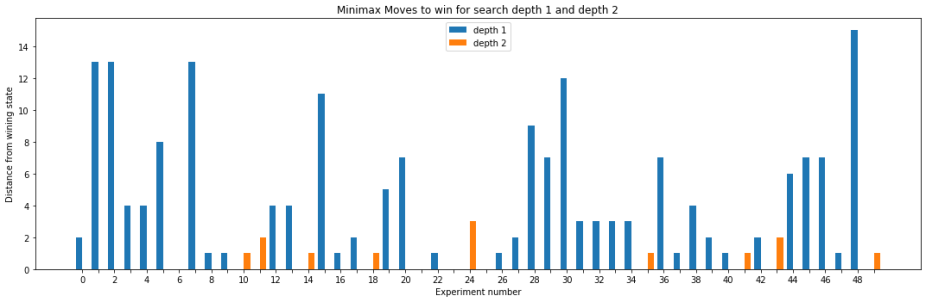


*Figure 18: Minimax depth 1 vs depth 2 bar chart*

As displayed in figure 18, the strength of the minimax algorithm clearly improves with search depth. The depth 1 algorithm very clearly takes many extra moves to win.

| Search Depth | Average moves from winning | Average moves | Average thinking time | Win rate |
|---|---|---|---|---|
| Depth 1 | 3.80 | 26.6 | 0.003 | 0.18 |
| Depth 2 | 0.26 | 22.24 | 0.101 | 0.82 |
| Depth 1 | 2.94 | 24.28 | 0.002 | 0.32 |
| Depth 3 | 1.54 | 22.20 | 2.060 | 0.68 |
| Depth 2 | 1.12 | 22.64 | 0.044 | 0.28 |
| Depth 3 | 0.36 | 21.16 | 1.625 | 0.72 |

*Table 1: Minimax Depths 1-3 results*

Based on observing the play of the minimax algorithms, it became apparent that minimax excelled in lining up pieces on the board to enable future chains of jumps. This meant that the pieces tended to be moved in small groups which could be jumped over in order to maximise the distance moved towards the goal with each jump.

### 4.1.2. MCTS self-play

MCTS was found to perform extremely poorly. Whilst the algorithm did converge to a win, it was not strong enough to compete with the other algorithms in chinese checkers due to it taking over double the number of moves before reaching a winning state. However, increasing the computational budget was found to significantly increase play quality. As such, it is feasible to assume that a much larger computation time would allow MCTS to compete with the other algorithms. Unfortunately, this was not possible to test due to hardware limitations.

| Algorithm | Iter max | Average distance | Average moves | Average thinking time | Win rate |
|---|---|---|---|---|---|
| **MCTS** | 100 | 5.46 | 68.90 | 0.475 | 0.04 |
| | 250 | 0.34 | 63.86 | 1.107 | 0.96 |

*Table 2: MCTS self play results*

Based on examining the play of MCTS, it appears that the algorithm plays poorly due to the fact playing non optimal moves can still result in a win from the simulations as it is difficult to predict the impact one move has on the final result of the game.

### 4.1.3. Greedy self-play

As expected, the greedy algorithm did not play with any intelligence or planning, and tended to move one piece toward the goal at a time. This resulted in the games taking slightly longer than the other algorithms, with an average number of moves of 28.71. However, whilst the greedy algorithm did not manage to intentionally line up its own pieces for consecutive jumping, it managed to utilise the other players pieces rather effectively. Due to this, there appeared to be a large luck factor required for the greedy algorithm to play effectively.

### 4.1.4. MaxN self-play

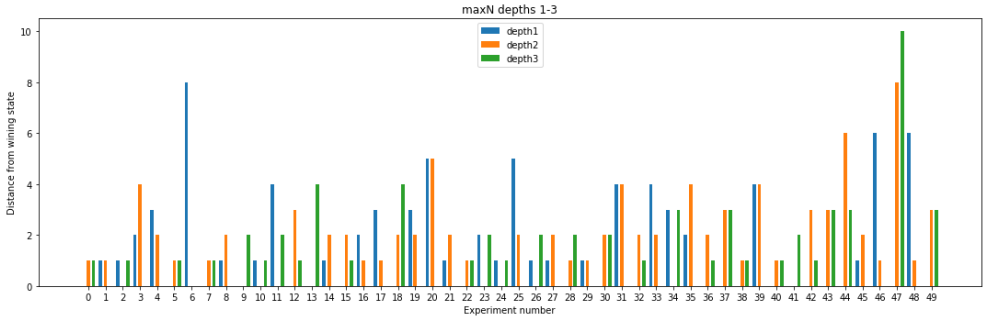


*Figure 19: MaxN depth 1,2,3 self play bar chart*

From analysing the data shown in figure 19, it was found that whilst a very minor improvement is given to the algorithm playing with a greater search depth, this improvement is very minor. This can be seen in the graph, as there appears to be no clear trend in the results. At depth 1, the algorithm won an average of 0.42 games, compared to 0.14 and 0.42 (totaling 0.56) of depth 2 and 3. The depth 3 algorithm had the lowest average distance from winning, with 1.22, compared to 1.54 and 1.82 of depth 1 and 2.

From observing the play of the maxN algorithm, no large noticeable differences between the three players were found. However, the algorithm with depth 3 appeared to play more akin to minimax with it's alignment of pieces to maximise the number of consecutive jumps. Although the other two depth algorithms did not seem to use this style of play, they still benefited from consecutive jumps by utilising the opposing players pieces by taking a greedy approach.

### 4.1.5. Paranoid self-play

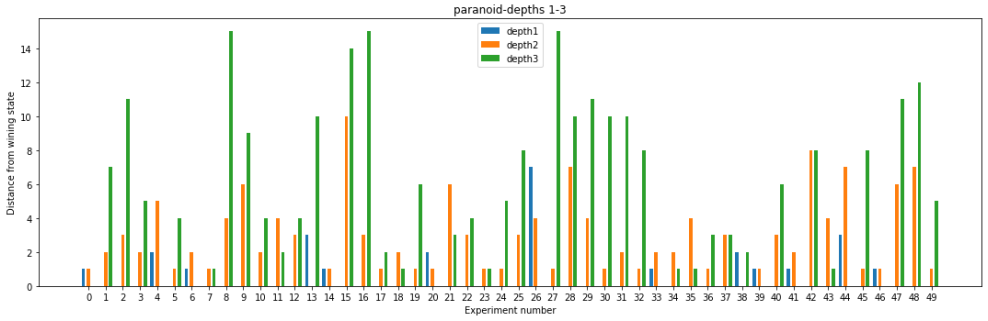


*Figure 20: Paranoid depth 1,2,3 self play bar chart*

The paranoid results present a peculiar trend, with a greater search depth actually hindering the play of the algorithm. At depth 1, the paranoid player had an average distance from winning of just 0.52, compared to 2.84 and 5.12 of the depth 2 and 3 algorithms. This appeared to be due to the large variance in the results of the depth 3 algorithm, where it tended to perform either rather well or extremely poorly. One possible reason for this could be due to the algorithm playing overly defensively and thus not seizing opportunities to make large jumps across the board.

### 4.1.6. Self Play conclusions

| Algorithm | Depth | Average distance from win | Average moves | Average thinking time | Win rate |
|---|---|---|---|---|---|
| **MaxN** | 1 | 1.54 | 23.12 | 0.004 | 0.44 |
| | 2 | 1.82 | 22.84 | 0.139 | 0.14 |
| | 3 | 1.22 | 22.38 | 5.002 | 0.42 |
| **Paranoid** | 1 | 0.52 | 25.32 | 0.003 | 0.74 |
| | 2 | 2.84 | 27.38 | 0.096 | 0.04 |
| | 3 | 5.12 | 29.70 | 3.038 | 0.22 |
| **Greedy** | n/a | n/a | 28.71 | 0.000….. | n/a |

*Table 3: MaxN, Paranoid and Greedy self play results.*

Based on these rudimentary results, it appears that MaxN is the strongest Chinese checkers player with the lowest average moves required to win of roughly 23. It does not appear that the maxN algorithm benefits from a greater search depth, and therefore I would argue that the exponential increase in computation time is not worth the very slight increase in play quality. The same can be argued for the paranoid algorithm, whose play quality actually deteriorates with an increased search depth.

Based on observing both of the algorithms self-play, neither seem to play to the same standard as a human player, and I was personally

able to outplay them without much skill. Both algorithms tended to struggle during the packing phase of the game, as they failed to find an optimal way to fit their pieces into the goal. This often resulted in many moves being wasted near the end of the game as they reshuffled the final pieces unnecessarily. I believe this may have slightly skewed the results regarding their performance, as had I solely measured the initial opening moves, the algorithms with a greater search depth clearly outperformed their opponents.

### 4.2. Comparative play results

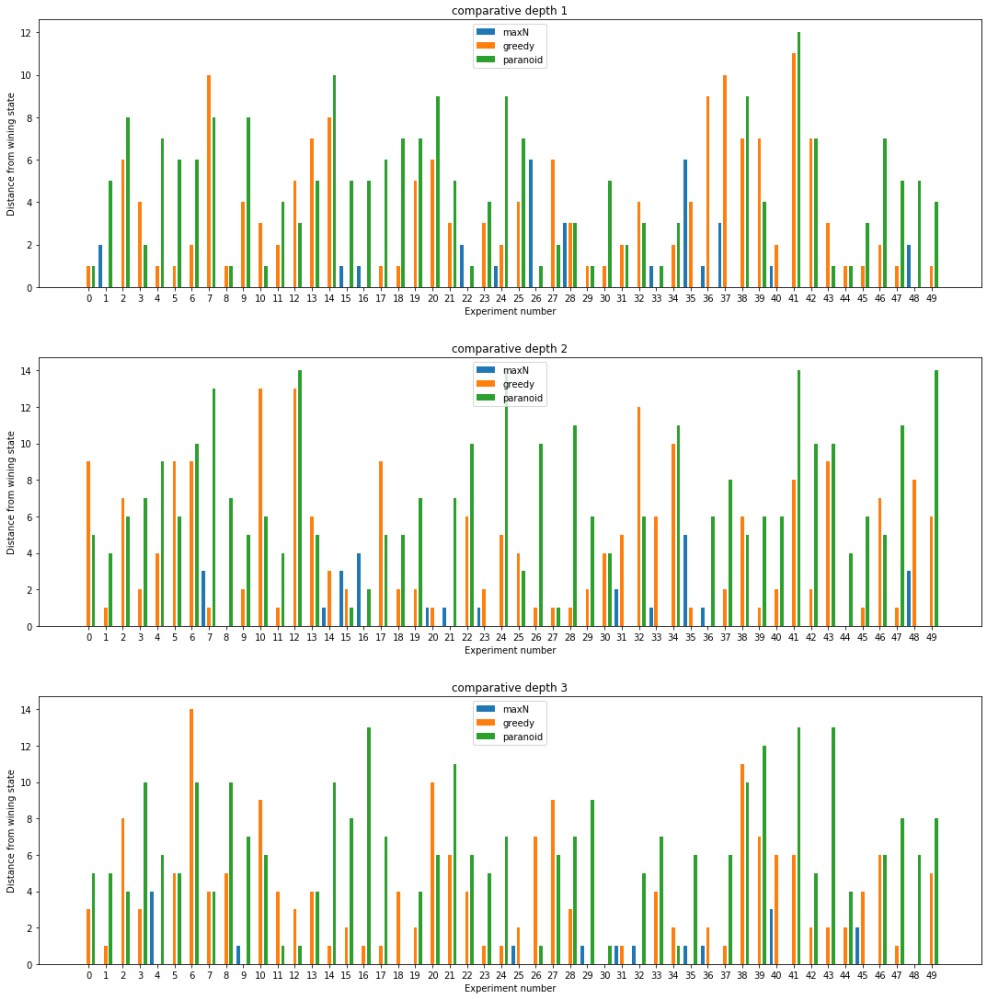


*Figure 21: MaxN vs Paranoid vs Greedy at Depth 1,2,3 bar chart.*

As displayed in the above graphs, the MaxN algorithm clearly outperforms paranoid at all search depths, whilst paranoid slightly outperformed the greedy algorithm at all depths. Increasing the

depth of search appears to have little impact on the results of the games.

| Depth | Algorithm | Average distance from win | Average moves | Average thinking time | Win rate |
|---|---|---|---|---|---|
| **Depth 1** | MaxN | 0.6 | 24.68 | 0.003 | 0.74 |
| | Paranoid | 3.3 | 27.72 | 0.003 | 0.14 |
| | Greedy | 4.38 | 28.94 | 0.000... | 0.12 |
| **Depth 2** | MaxN | 0.52 | 23.74 | 0.114 | 0.76 |
| | Paranoid | 4.18 | 27.16 | 0.110 | 0.14 |
| | Greedy | 6.18 | 29.26 | 0.000... | 0.10 |
| **Depth 3** | MaxN | 0.32 | 23.58 | 4.124 | 0.8 |
| | Paranoid | 3.58 | 26.64 | 4.007 | 0.1 |
| | Greedy | 5.78 | 28.94 | 0.000... | 0.1 |

*Table 4: MaxN, Paranoid and Greedy comparative play results*

Table 4 highlights these findings and showcases the superiority of the maxN algorithm at all examined search depths. It can be clearly seen that increasing the depth of search furthers maxN's dominance over the other two algorithms, with the average number of moves taken to win decreasing, and the win rate increasing as the search depth increases. However, it is likely that this increase in performance is due to the fact the paranoid algorithm actually performed worse with a greater search depth, resulting in maxN appearing to perform better. That being said, paranoid still clearly outplays the baseline performance of the greedy algorithm and as such, clearly still offers a viable alternative to minimax.

# 5. Evaluation

## 5.1. Summary

Based on the findings of the previously detailed experiments, it can be concluded that maxN performs significantly better than the paranoid algorithm in the domain of Chinese checkers. Additionally, it does not appear that increasing the search depth provides much substantial improvement to the maxN algorithm, possibly because the algorithm does not reach the second round of maximising nodes in its search. Conversely, the paranoid algorithm's performance deteriorated with an increased search depth, possibly because the algorithm played increasingly defensively and did not seize optimal offensive strategies. These results contradict those found in [23], where the paranoid algorithm "widely outperforms" maxN. However, they align with the previously documented expectation that search depth would have little impact in the game of Chinese Checkers.

Additionally, it can be concluded that an unoptimised MCTS algorithm does not produce strong play under a similar computational budget to generic GOFAI algorithms such as Minimax. It is clear that in order for MCTS to be effective in this domain, vast improvements must be made to mitigate the problem of the large branching factor. This would require incorporating specific game knowledge to improve the algorithm's search, and adding optimisations to prune the search tree and improve selection.

As such, the motivations for the project have certainly been met: thorough research into various alternatives for minimax at varying computational budgets has been completed and the small gap in research identified in the literature review has been filled.

## 5.2. Outcomes

- It has been successfully proven that MaxN and paranoid algorithms play effectively as a replacement for minimax, both out performing the greedy algorithm.

- It has been found that increasing the computational budget of MaxN slightly increases play quality. Conversely, it has been found that the Paranoid algorithm performs worse with an increasing depth. As such, increasing the depth is not worth the corresponding increase in search time.

- It has been proven that MaxN vastly outperforms paranoid and greedy algorithms in comparative play, whilst paranoid appears to play at a very similar level to the greedy algorithm.

- It has been established that an unoptimised Monte Carlo algorithm does not perform to any reasonable standard in the game of Chinese checkers.

### 5.3. Improvements

This project could have been greatly improved by increasing the number of tests run for each configuration in order to minimise the impact of the random aspect of luck in the results. Additionally, with more computational resources and time, the impact of a greater search depth could have been examined. In order to add more aspects to the project, further pruning methods could have been examined, alongside repeating the tests in different game domains.

### 5.4. Reflection

In my opinion, the project was a success overall yet was limited in a few areas. I spent a very long time attempting to get the MCTS algorithm to perform well, before coming to the realisation that the problem was not with my code, but rather that it simply did not fit this game domain. This resulted in the tests being rushed towards the end of the project and a rather limited amount being produced. This, combined with the lack of access to a server or a computer lab, meant that the tests took days of running on my personal laptop which was unable to perform evaluation of depths greater than 3. Despite this, the algorithms were all successfully implemented, and a range of characteristics and features were measured. Furthermore, a clear set of results was synthesised with a set of clear outcomes. As such, I do believe this project to be a success and I am proud of the work I have produced.

# 6. Bibliography


1. IBM Research, *Deep Blue* [Online]. Available: https://www.ibm.com/ibm/history/ibm100/us/en/icons/deepblue/. [Accessed: Jan. 25, 2020].

2. F. H. Hsu, *Behind Deep Blue: Building the Computer that Defeated the World Chess Champion.* Princeton University Press, 2004.

3. C. Krauthammer, *Be Afraid*. The Weekly Standard [Online]. Available: https://www.washingtonexaminer.com/weekly-standard/be-afraid-9802 [Accessed: Jan 25, 2020].

4. J. Haugeland, *Artificial Intelligence: The Very Idea*. MIT Press, 1986.

5. DeepMind, *AlphaGo* [Online]. Available: https://deepmind.com/research/case-studies/alphago-the-story-so-far [Accessed: Jan 25, 2020].

6. B. Brugmann, *Monte Carlo Go.* White Paper, 1993.

7. C. A. Luckhardt, K. B. Irani, "*An Algorithmic Solution of N-Person Games.*" *AAAI*, vol. 86, pp. 158-162, 1986.

8. R. Korf, "*Multi-player alpha-beta pruning*." *Artificial Intelligence,* vol. 48, pp. 99-111, 1991.

9. N. R. Sturtevant, R. Korf, *"On Pruning Techniques for Multi-Player Games." AAAI / IAAI*, vol. 49, pp. 201–207, 2000.

10. M. Schadd, M. Winands, *"Best Reply Search for Multiplayer Games*." *IEEE Transactions on Computational Intelligence and AI in Games 3*, pp. 57-66, 2011.

11. J. Nijssen, M. Winands, *"An Overview of Search Techniques in Multi-Player Games." Computer Games Workshop at ECAI*, pp. 50-61, 2010.

12. The Editors of Encyclopaedia Britannica, *Halma (Chinese checkers).* Encyclopaedia Britannica [Online]. Available: https://www.britannica.com/topic/Halma-game [Accessed

Jan 28, 2020].

13. G. I. Bell, *"The Shortest Game of Chinese Checkers and Related Problems." Integers: Electronic Journal of Combinatorial Number Theory*, vol 9, pp. 17-39, 2009.

14. N. Fonseca, *Optimizing a Game of Chinese Checkers.* BSU Honors Program Theses and Projects [Online]. Available: https://vc.bridgew.edu/honors_proj/129 [Accessed Feb 1, 2020].

15. A. Patel, *Hexagonal Grids,* Red Blob Games [Online]. Available: https://www.redblobgames.com/grids/hexagons/ [Accessed Feb 5, 2020].

16. W. Pijls, A. Bruin, "*Game tree algorithms and solution trees." Theoretical Computer Science,* vol 242, pp. 197-215, 2001.

17. A. Elnaggar, et al.*, "A Comparative Study of Game Tree Searching Methods." International Journal of Advanced Computer Science and Applications,* vol 5, pp. 68-77, 2014.

18. R. Korf, "*Heuristic evaluation functions in artificial intelligence search algorithms.*" *Minds and Machines,* vol 5, pp. 489-498, 1995.

19. J. Clune, "*Heuristic evaluation functions for general game playing.*" *AAAI*, vol 7, pp. 1134-1139, 2007.

20. T. Cormen, et al., *Introduction to Algorithms,* MIT press, 2009.

21. R. Luce, H. Raiffa, *Games and decisions: Introduction and critical survey,* Courier Corporation, 1989.

22. S. Fuller, et al., *Analysis of the alpha-beta pruning algorithm*. Department of Computer Science, Carnegie-Mellon University, 1973.

23. N. Sturtevant, "*A Comparison of Algorithms for Multi-Player Games." International Conference on Computers and Games,* pp. 109-122, Springer, 2002.

24. M. Winands, "*Monte-Carlo Tree Search in Board Games." Handbook of Digital Games and Entertainment Technologies*, pp. 47-76, 2015.

25. P. Auer, et al., “*Finite-time Analysis of the Multiarmed Bandit Problem” Machine Learning,* vol 47, pp. 235-256, 2002.

26. L. Kocsis, C. Szepesvari, “*Bandit based Monte-Carlo Planning*.” *European conference on machine learning,* pp. 282-293, 2006.

27. C. Browne, et al., "*A survey of monte carlo tree search methods.*" I*EEE Transactions on Computational Intelligence and AI in games,* vol 4, pp. 1-43, 2012.

28. T. Banks, “Comparing Different Monte Carlo Tree Search Extensions in the Game of Chinese Checkers”, Department of Computer Science, University of York, 2019.

29. A. Rimmel, et al., "*Biasing Monte-Carlo simulations through RAVE values.*" *International Conference on Computers and Games. pp . 59-68, 2010.*

30. J. Nijssen, M. Winnads, “*Playout Search for Monte Carlo Tree Search in Multi-Player Games*.” A*dvances in Computer Games*, pp. 72-83, 2011.

31. J. Nijssen, M. Winnads. *“Enhancements for Multi-Player Monte-Carlo Tree Search.” International Conference on Computers and Games. pp . 238-249, 2010.*

32. S. Russell, P. Norvig, *Artificial intelligence: a modern approach (3rd edition).* Prentice Hall, 2009.

33. M. Winnans, "*Monte-Carlo tree search in board games.*" *Handbook of Digital Games and Entertainment Technologies,* pp. 46-76, 2017.