1.5em 0pt

Opleiding Informatica

How to win in Checkers?

for a Thesis

Kwan Lie (s2652617)

Supervisors:
Prof.dr. Aske Plaat

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
www.liacs.leidenuniv.nl

dd/mm/yyyy

Abstract

Checkers is a board game played by two players, and the objective of the game is to capture all the opponent's pieces. This thesis will compare two algorithms, minimax and Monte Carlo Tree Search (MCTS), and see which algorithm is able to play the game better. This thesis contains information about the game Checkers itself as well as the algorithms. This also contains a short explanation of how the code, for running the experiments, is constructed. In order to determine which one of the algorithms performs the best, both algorithms will be optimized first. The minimax will be optimized based on the king value relatively to a normal piece and the positional structure. Also, a comparison of function calls between the regular minimax and minimax $\alpha\beta$. An attempt has been made to optimize the efficiency of the minimax $\alpha\beta$. For the MCTS, the constant value C in the UCB1 formula has been optimized to the value 0.3. Lastly, the two algorithms play against each other by having the MCTS algorithm having 100-500 playouts play against minimax $\alpha\beta$ with a depth 2-4. MCTS resulted in drawing and losing many games against the minimax player. This is probably due to the game setup or the amount of playouts the MCTS algorithm has. So for further research, the suggestion is to test the comparison on a variation of Checkers or rewrite/optimize the code in order to make the MCTS play with more playouts.

Contents

1 Introduction 1
1.1 Thesis overview 1

2 Related Work 2
2.1 How does a Checkers program look like? 2
2.1.1 Rules 2
2.1.2 History of Checkers programs 3
2.1.3 Chinook 4
2.2 Minimax algorithm 4
2.2.1 Alpha-Beta pruning 7
2.3 Monte Carlo Tree Search 9

3 Implementation and Agents 11
3.1 Setup game 11
3.2 Random Player 12
3.3 Minimax Player 12
3.3.1 Minimax Alpha-Beta Player 13
3.4 MCTS algorithm 13

4 Experiments 14
4.1 Optimizing Minimax 14
4.2 Comparing Minimax and Alpha-Beta pruning 18
4.3 Optimizing MCTS 20
4.4 Minimax vs MCTS 20

5 Conclusions and Further Research 22

References 24

1 Introduction

There are many types and genres of games to play such as online games, party games, strategy games and many more. Some games are affected by some sort of luck, for example games which contain a die. And some games are only based on a player's skill. Such games usually have very little to none aspects of luck which influences the outcome of the game. A game which is played with perfect information is called a combinatorial game. A combinatorial game does not rely on any form of luck or chance, and the state of the game together with the set of available moves are always known by the players[1]. An example of a combinatorial game is the game Checkers.

Checkers is a board game played by 2 players and is a slight deviation of the game Draughts. The game is played on an 8×8 board, with each side having 12 pieces. The objective of the game is to remove all your opponent's pieces by taking them using your own pieces. Many people know the general rules of each game, but of course everyone has a different level of skill. How do you determine which move is the best during the game? Although this can be quite a difficult task for a person, computers are able to determine these good moves by computing the position using different algorithms. They are able to go through a large search space to find the best possible move within the current state of the game, and this is exactly the theme of this thesis.

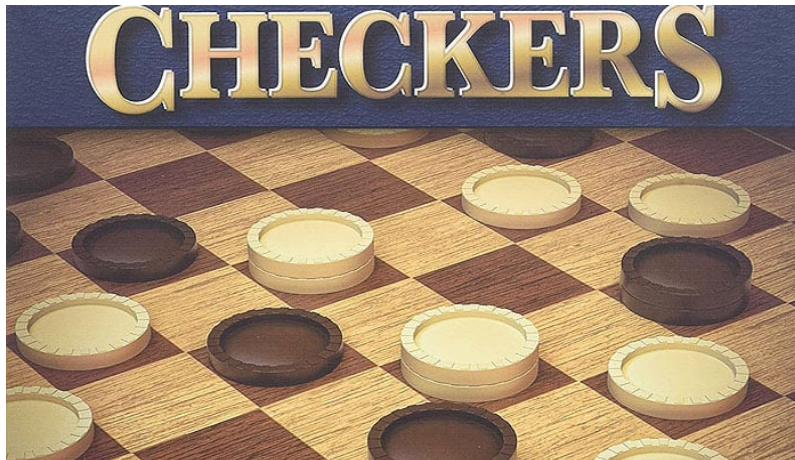

Figure 1: Visual reference Checkers

In this thesis, the MiniMax-algorithm and the Monte Carlo Tree Search (MCTS) will be compared to play the Checkers. The algorithms will be tested with different depths and play outs to see which one of the two performs better. It is speculated that the MCTS-algorithm performs better. Therefore, the research question of the thesis: "Does the MCTS-algorithm outperform the MiniMax-algorithm in the game Checkers?".

1.1 Thesis overview

This thesis, including the introduction section 1, contains of 5 sections. Section 2 discusses the related work as well as the description of the used algorithms. Section 3 goes over the written code

used for this thesis. Section 4 describes the experiments and their outcome. The last section 5 will conclude and discusses the potential further research.

2 Related Work

2.1 How does a Checkers program look like?

2.1.1 Rules

There are many variations to play the game of Checkers. Depending on the region the game is played, the rules of the game may vary. In this thesis, the Anglo-American standard Checkers will be played, which is also the most commonly played Checkers variant. The rules are as follows:

The game is played on an 8x8 board, with each player having 12 pieces. One player has dark pieces, usually the color black, and the other player has light pieces, most often white. The player who plays the dark pieces move first.

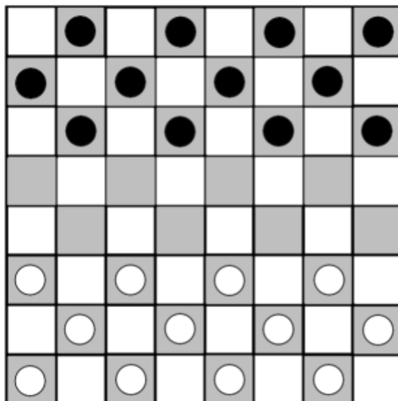

Figure 2: Starting position

Pieces move diagonally over the board, single pieces are only allowed to make forward moves. A piece making a non-capturing move may only one square.

The objective of the game is to capture all your opponent's pieces. To capture a piece of the opponent, the player's piece leaps over the opponent's piece and lands in a straight diagonal line on the other side. The opponent's piece has to be adjacent to the player's piece, and the landing square must be empty.

When a piece is captured, the piece is removed from the game. Only a single piece may be captured in a single jump, but multiple jumps are allowed within a single turn. If a player is able to capture a piece, the player is forced to capture that piece. If more than one capture is available, then the player decides if he prefers this or not. Single pieces are only allowed to make forward captures.

When a piece reaches the furthest row, the piece itself is promoted to a king. The differences between kings and single pieces is that kings can move and capture forward as well as backwards.

A player wins the game when the opponent is not able to make a move. This typically happens when the opponent does not have any pieces left, however this can also happen because the opponent's pieces are blocked in.

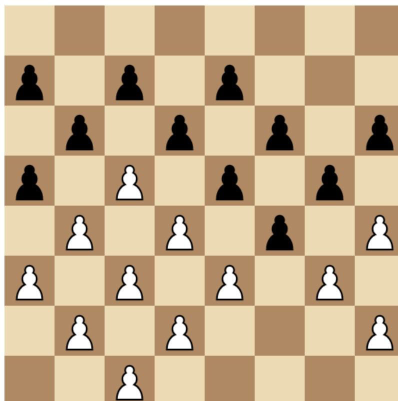

Figure 3: Example of a blocked position. Whoever turn it is right now loses

2.1.2 History of Checkers programs

The game Checkers has a long history and is originally played over the board. However, since computers became more compatible, people started to create programs to play Checkers with computers. The first program ever was written by Christopher Strachey, National Research Development Corporation, London, in the early 1950s. This program was run on a Pilot ACE which was one of the first computers built in the United Kingdom. It exhausted the computer's memory, so the program soon ported to a better computer, Ferranti Mark 1. His program is written in the programming language CPL.

The second person who created a Checkers program was named Arthur Samuel. Arthur Samuel started developing his Checkers program in the 1950s. In 1962, the program played a game against Robert Nealey, who was a strong Checkers player, and won. This single win was at the time an historic event and gained a lot of attention in the media. In the field of artificial intelligence research, this was a big milestone, and even after 30 years this work has been cited quite frequently. Many people thought that this was the point of time that computers were able to play the game better than humans and in a result thought that the game was solved. Scientists moved to other games, such as chess. However, this assumption turned out to be far from the truth.

A rematch was organized the following year. The match started in fall and lasted more than five months. In this match, Nealey played a total of six games against the computer. Playing at home, Nealey analysed and studied positions on each of the six games. When he knew the move he wanted to play, he typed the moves on the postcard and mailed them to the IBM Watson Research Center

in Yorktown Heights, N.Y., where the computer is located. Depending on the position, the machine probed to a depth of at least six and at most 20 moves ahead.

The final result was a win for Nealey, where he scored 1 win and 5 draws. He commented: "By sticking to its programmed instructions, it may find an extraordinary move that a man who is gifted imaginatively may never find. It knows so much and carries its analysis to such depths that it sometimes, by the beauty of its mathematics, comes up with a truly brilliant move. This is difficult to express, but I think the machine's complete lack of imagination is its most formidable strength!"

In 1966, Samuel took his program to the world championship match between Walter Hellman and Derek Oldbury. IBM was the sponsor of the event with the condition that the participants had to play a few games against the program. Each player played 4 games against the program, whereas the program lost all the 8 games relatively easily. Therefore, the conception that the game of Checkers was solved was completely wrong [5] [7]!

2.1.3 Chinook

The Checkers program Chinook began its development in 1989 with the goal of beating the human World Checkers champion. The program won for the first time against a champion in 1990, but lost soon after in 1992. It then again won in 1994 and in 1996 it was clearly to see that the program was indeed more capable to play Checkers than humans.

Chinook uses a large database of the opening and ending phase. This database has been created by many of the best Checkers players in the world. In the middle phase, Chinook uses a deep search algorithm and a good move evaluation function.

On April 27th, 2007, Chinook announced that they have solved the game of Checkers. From the starting position, the dark pieces are guaranteed to draw with perfect play. The light pieces are also guaranteed to draw with perfect play, regardless of which move the dark pieces starts with. A draw means that pieces remain on the board but are unable to capture the opponent without getting caught themselves first. That is why the player would rather run away from the other player, which could end up forever. Checkers has a search space of 5×10^{20} and is therefore the largest game that has been solved to date [9].

2.2 Minimax algorithm

A common algorithm used for analysing a game with perfect information such as Checkers is the minimax algorithm. In this algorithm, there are usually two players, which will be called MIN and MAX. Players take turns moving until the game is over. A game can be formally defined with the following elements:

- S_0: The initial state, which specifies how the game is set up at the start of the algorithm
- To-move(s): The player whose turn it is to move in state s.
- Actions(s): The set of legal moves in state s.

- Result(s, a): The transition model, which defines the state resulting taking action a in state s.
- Is-Terminal(s): A terminal test, which is true when the game is over and false otherwise.
- Utility(s,p): An utility function which defines the numeric value to player p when the game ends in terminal state s, this can also be seen as an evaluation function.

The initial state, Actions function, and Result function define the state space graph. A state space graph is a graph where the vertices are states, edges are moves and a state might be reached by multiple graphs. The complete game tree is a search tree which follows every sequence of moves from a specific state all the way until to a terminal state. A tree may be infinite if the amount of actions is unbounded or that the rules of the game allow an infinite repetition of moves.

In figure 4, an example of a game tree of the game tic-tac-toe can be seen.

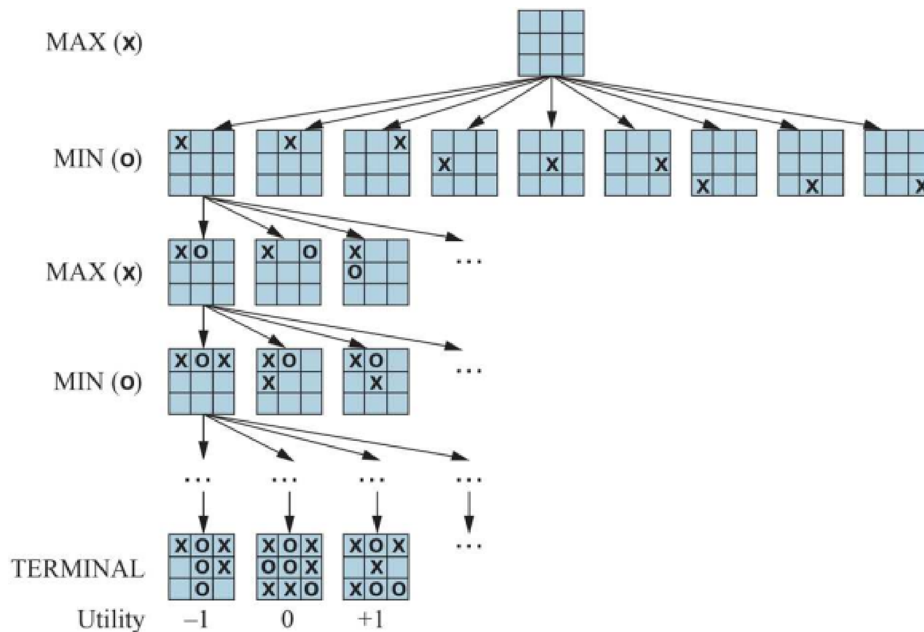

Figure 4: Game tree of tic-tac-toe

From the initial state, the player x has 9 possible moves. From each of those moves 8 other moves can be filled in by player 0 and this goes back and forth until one of the players win or a draw is concluded. Once a terminal state is reached, which is a leaf of the tree, the utility function returns a number. High values are good for MAX and bad for min, vice versa.

So a MAX player aims for a score as high as possible and a MIN player aims for a score as low as possible, so how will the tree look like exactly? Take the following game tree as an example:

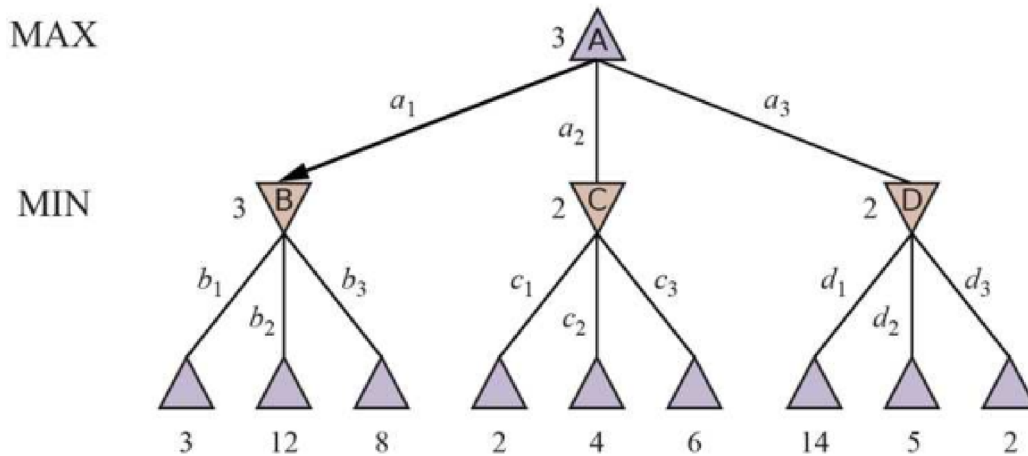

Figure 5: Example game tree minimax

In order to pick the best move, firstly a tree with the possible states is composed. After that, scores can be assigned to each leaf within the node and the best move corresponding to each state is picked from the bottom-up. The possible moves of player MAX are a_1, a_2 and a_3. The possible moves for player MIN are $b_1, b_2, b_3, c_1, etc....$ Each layer represents whose turn it is. In state B, it is the turn of player MIN and the player chooses the lowest score so 3. In state C the value 2 and in state D. In state A, the MAX player has 3 options, 3, 2 and 2, and picks the highest value 3 done by the action a_1.

The minimax algorithm is a recursive algorithm that goes all the way down to the leaves and backs up the minimax values as the recursion unwinds. If 5 is taken for example, the algorithm will first compute the leaves after state B, then backs up the value 3 to state B. After this it goes to the leaves after state C etc. until all the moves have been done for the MIN player. At the end, the algorithm picks the value 3 for state A.

Algorithm 1 Minimax algorithm: minimax(child, bestmove)

```
if depth== 0 or end state is reached then
    RETURN evalution of the state
end if
if MAX player's turn then
    maxValue = -infinity
    for all the possible moves do
        eval = minimax(depth-1, bestmove)
        if eval > maxValue then
            maxEva = eval
            Remember move i
        end if
    end for
    RETURN maxValue
else
    minValue = infinity
    for all the possible moves do
        eval = minimax(depth-1, bestmove)
        if eval < minValue then
            maxEva = eval
            Remember move i
        end if
    end for
    RETURN minValue
```

When the algorithm has finished, the action "bestmove" can be done on the board. For games with a small search space, it is possible to plot out the whole game tree. However, for more complex games like Checkers, plotting out a whole game tree will be unfeasible since the search space will be extremely large. That is why the algorithm keeps track of a depth and only creates a tree with that many levels at max or when it reaches an end state. The minimax algorithm performs a complete depth-first exploration of the created game tree. Take the variable m for the maximum depth of the tree and b for the amount of possible moves in each state. The complexity of the minimax algorithm is $O(b^m)$. The space complexity is O(bm) for an algorithm that generates all actions at once, and the space complexity is O(m) for an algorithm that generates actions one at a time[8].

2.2.1 Alpha-Beta pruning

The number of states is exponential in the depth of the tree. However, it is possible to compute the correct minimax decision without going over every state. This is done by pruning large parts of the tree and saves quite a lot of calculations. This particular technique is called alpha-beta pruning. Take again the example of figure 5.

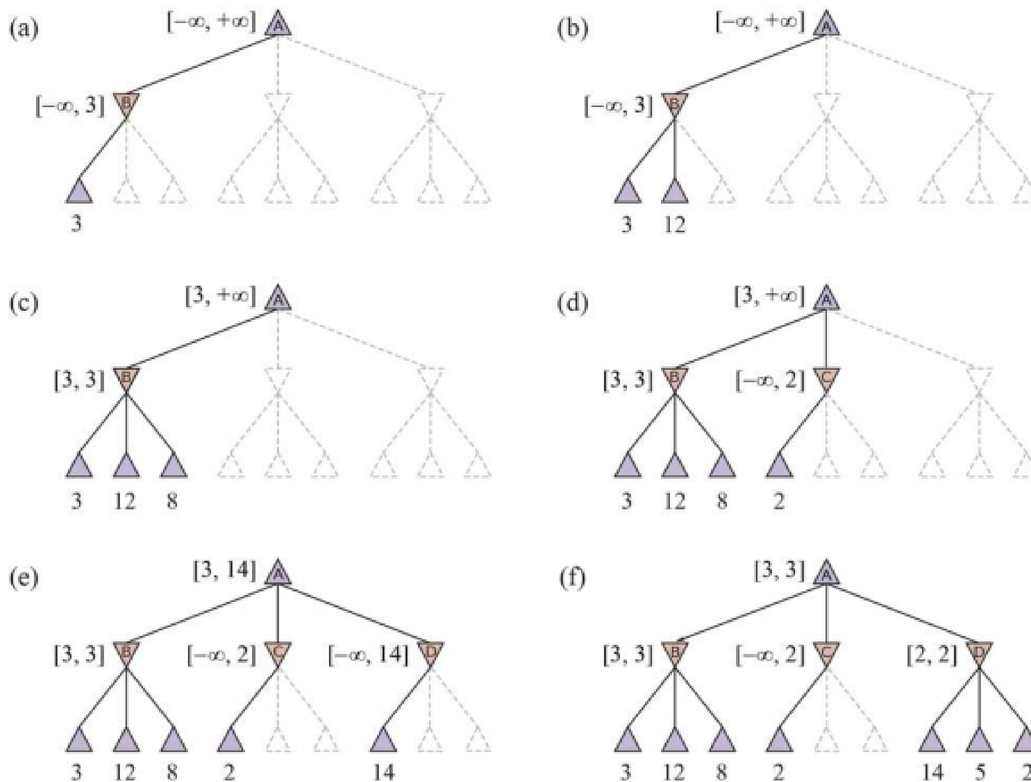

Figure 6: Example game tree minimax

In stages a, b and c, the algorithm compares all the leaves of node B. This node B is a MIN node, therefore the algorithm picks the value 3. In this case, it can also be inferred that the node A picks a node with at least the value 3. d) Next, the first leaf of node C has the value 2. Node C is a MIN node, hence it will pick a value at most of a value 2. It turned out earlier that node B has a value 3 so node A will never pick node C and so it is unnecessary to look further into node C and so the tree is pruned. e) The first leaf of node D has the value 14 so node D is at most worth 14. This is more than the current highest value 3 and so more leaves of node D will be compared. In step f, it turns out that node D has a value of at most 2 so in the end the node B will be chosen by A with a value of 3.

Alpha-beta pruning can be applied to trees of any depth, and with this method it is also possible to prune whole subtrees instead of only leaves. The general principle is as follows: consider node n somewhere in the tree, whereas a player has the option to move to n. If the player has a better choice either in the same level or earlier in the tree, the player will never move to node n. So once there is information that there are better moves leading to n, the node n can be pruned.

Minimax search is depth-first, so only nodes along a single path of a tree need to be considered. Alpha-beta pruning gets its name from the two extra parameters α and β. Whereas α holds the value for the current best choice that is found for the MAX player, and β holds the value for the current best value of the MIN player.

It is important to note that the effectiveness of Alpha-beta pruning is highly dependent on the

order that the moves are searched trough. The earlier good moves are found, the more pruning this algorithm can do. However, in this thesis, there will be no extra emphasis placed on the move ordering [8].

2.3 Monte Carlo Tree Search

The second algorithm that will be looked at is the Monte Carlo Tree Search, MCTS in short. The basic MCTS strategy does not rely on a heuristic function. Instead, the value of each state calculated by the average utility over certain amount of simulations of complete games starting from each state. A simulation is also called a playout or rollout. Starting from a state, a game will be played until a terminal state.

The MCTS uses a selection policy that focuses the computational resources on the important parts of the game tree. In order to do so, two factors are important to take into consideration: exploration of states that have a relatively low amount of playouts and exploitation of states that have been done well in past playouts, to get a more accurate estimate of their value. MCTS does this by maintaining a search tree and expands this tree every iteration of the following four steps:

- Selection: Starting at the root of the node, pick a move guided based on the selection policy, leading to a successor node. Repeat this process on the successor node until a leaf has been reached.
- Expansion: Expand the search tree by creating a new child of the selected node.
- Simulation: Perform a playout on the newly created child. The moves within this playout are determined by the playout policy. The moves themselves are not recorded in the tree, it is only the result that matters.
- Back-propagation: The result of the simulation is used to update all the nodes going all the way to the root.

An example of an MCTS search tree can be seen in Figure 7. In this tree, white starts by making a move. The root 37/100 means that white won 37 games out of the 100 playouts. The second level stands for the amount of wins for black. The third level goes back to white etc. a) The path to the node is based on the selection policy UCT. The path is 37/100, 60/79, 16/53 and ends in the leaf 27/35. b) Arrived at the node, a new child is created and a playout is performed. The result is that black wins, so the value of the node is 0/1. c) At last the backpropagation is performed where the values of the previous visited nodes are updated to 28/36, 16/54, 61/80, 37/101.

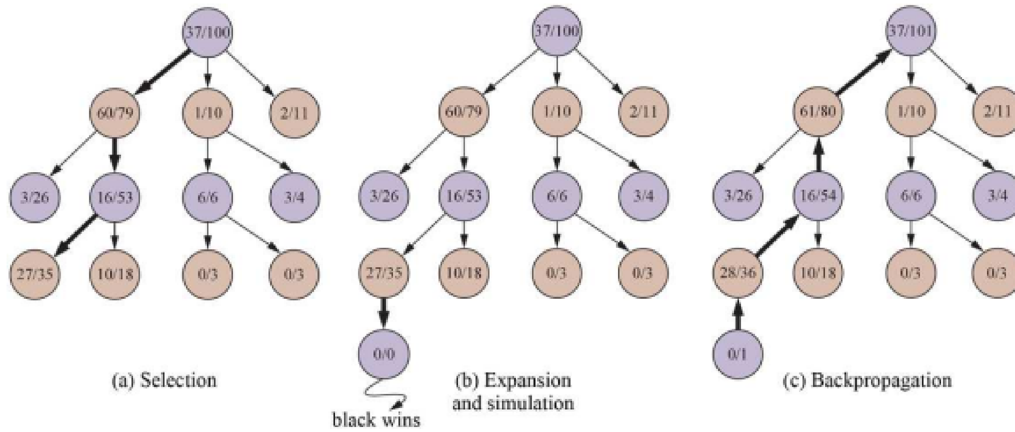

Figure 7: Example game tree minimax

One commonly used and effective selection policy is called "upper confidence bounds applied to trees" or UCT. The policy ranks each possible move based on an upper confidence bound formula called UCB1. For node n, the formula is:

$$UCB1(n) = \frac{U(n)}{N(n)} + C \times \sqrt{\frac{logN(Parent(n))}{N(n)}} \tag{1}$$

- U(n): total utility of all playouts that went through node n
- N(n): number of playouts through node n
- Parent(n): parent node of n in the tree

The first division in the formula looks at the quality/utility of the node and is therefore the exploitation term. The term with the square root is the exploration term. the count N(n) makes it, so the term will be high for nodes that have been explored only a few times. The numerator contains the log of the number of times that the parent n has been explored. This gives the preference to the node which has been relatively little. Note that a division through zero is not possible, and therefore some extra implementation is needed.

The variable C is a constant value which is used to balance the exploitation and exploration. There exists an argument that C should be $\sqrt{2}$, but in practice, programmers test different values and pick the best one depending on the setting or game which the algorithm is applied to. The UCB1 formula ensures that the move with the highest win percentage in considered to be the best move. This is because the selection process favours the win percentage more as the amount of playouts go up.

Algorithm 2 Monte Carlo Tree search (MCTS) algorithm

```
tree ← node(state)
while Amount of nodes < n do
    leaf ← SELECT(tree)
    child ← EXPAND(leaf)
    result ← SIMULATE(child)
    BACK-PROPAGATE(result, child)
end while
RETURN the move in ACTIONS(state) whose node has highest number of playouts
```

The time to compute a playout is linear, not exponential, in the depth of the tree, since only one move is taken at each point. This makes it possible to have multiple playouts. The big advantage of MCTS towards minimax alpha-beta is that MCTS is more applicable to games where they have a huge branching factor or when it is difficult to determine an evaluation function for the game. A single mistake in the evaluation function for minimax alpha beta can have detrimental effects on the performance of the game. MCTS can be applied to new games, in which there is no information for the evaluation function. MCTS has a disadvantage when a single move can change the course of the game, since of its stochastic nature it might fail to consider that move [8].

3 Implementation and Agents

This section will go through how the code is implemented for running the experiments. The general setup of the game and the algorithms will be explained. The code's overall structure is inspired by the third assignment of the course Kunstmatige Intelligentie (Artificial Intelligence) 2021 from Universiteit Leiden and the thesis "Monte Carlo Tree Search for Dots-and-Boxes"[4] by Said Krebbers. The code is written in C++.

3.1 Setup game

The board is set up as a 2D char array. An element in the array can either be a player or empty. The player with the light pieces are represented as a 'w' for a normal piece or 'W' for a king piece. The player with the dark pieces are represented as 'b' or 'B'. An empty square is represented as 'o'. According to the rules, the black player starts. An important rule to note is that if a player can take a piece, the player will have to capture a piece. That is why, before the player performs an action, it is checked beforehand if a piece can be captured.

The data for doing a turn consists of at least 4 values. The x and y coordinate of a piece and the x and y coordinates of where the piece wants to move. For both algorithms, minimax and MCTS, all the possible moves that can be done on the current state of the board need to be computed. These moves are saved in a `std::vector<std::vector<int>>`, where the vector within the vector are the coordinates of the move and the vector itself is the possible available moves. Obviously, the moves are always checked before added to the vector. Checks are for example: is there a free space available if an opponent's piece is taken or if the piece doesn't walk out of bounds.

When multiple captures are possible, a depth first search will be done to find all the possible moves. The program will recursively find all the possible captures and adds them to the current vector. A turn with 2 captures has 8 elements, 3 captures 12 elements, etc. Take for example 9 where the white piece is a king piece. This shows the order of the capture moves found in the program. Note that there are 5 different moves that can be played this turn 1, 1-2, 1-2-3, 1-2-4, 1-2-4-5.

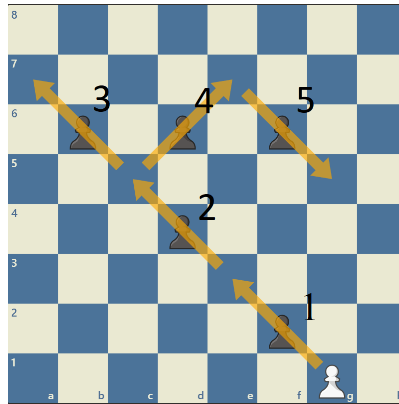

Figure 8: The order of capture moves found

3.2 Random Player

The random player simply finds all the possible moves that can be played within the position and chooses one of these moves completely randomly. The randomness is determined with `srand(seed)` with the seed given when the program is started.

3.3 Minimax Player

As mentioned before, the minimax algorithm can be formally defined with 6 elements. Based on the rules and the setup of the program, these are as follows:

- S_0: The current position of pieces on the board when the algorithm is called
- To-move(s): Either the light or the dark player's turn. This is kept up with a global boolean variable.
- Actions(s): All the possible moves in the current position, so basically the possible moves vector
- Result(s,a): The state of the board after a certain move has been done
- Is-Terminal(s): A check whether the game is over or not. Either one of the players does not have a move left or the game ended with a draw. A draw is concluded when the maximum amount of moves have been reached.

- Utility(s,p): A function that evaluates the current state of the board. Think of the amount of pieces and the amount of kings.

The algorithm is implemented accordingly as stated in 2.2. Normally, a copy of the board is created, and a move is done on the copy of the board. However, to save computational power, instead of making a copy, the move is undone by resetting the board to its position before the move was made.

The algorithm holds 2 parameters, depth and bestmove. Depth is the depth of the tree that the algorithm will look into. Bestmove is the best n move that can be done in the current position.

3.3.1 Minimax Alpha-Beta Player

Almost the same as the regular minimax Player, however it holds two extra parameters with are needed for the alpha-beta pruning. The way that the possible capture moves are found shows that the move found at last is the move with the most captures. With that in mind and with the assumption that more captures in one move would most likely be the best move, this move will be explored first.

3.4 MCTS algorithm

The MCTS algorithm is built up with two classes. The first class is the 'Node' class. This class is responsible for structuring the nodes and building up the tree properly. The second class is the 'MCTS' class, which holds the 4 iterative steps and does a move on the board accordingly.

To do a MCTS move, the algorithm starts off by having a root node and a copy board the current board. The first step is the selection, the algorithm selects the best child based on the selection method UCB. The variable C is set to 1.4 and in order to prevent dividing by 0, when a node does not have any playouts, $N(n)$ is set to 0.5. The algorithm will keep selecting nodes going down the tree until it has reached a leaf node.

Next step is the expansion, when arrived to a leaf node and the amount of playouts is not 0, new children will be created for this specific node. The amount of children created is the same amount of possible moves within that position.

Then, the first child is chosen, and this move is done on the copy of the board. Next, in the simulation step, the next a playout will be done on the node where the algorithm currently finds itself in. This playout, in other words the playout policy, will be done only with random moves.

Lastly, in the back propagation step, the playout returns a value. Based on which player, light or dark, the MCTS player is, a value is given to the node and the tree is updated. These steps are repeated for a certain amount of simulations that can be set when starting the program.

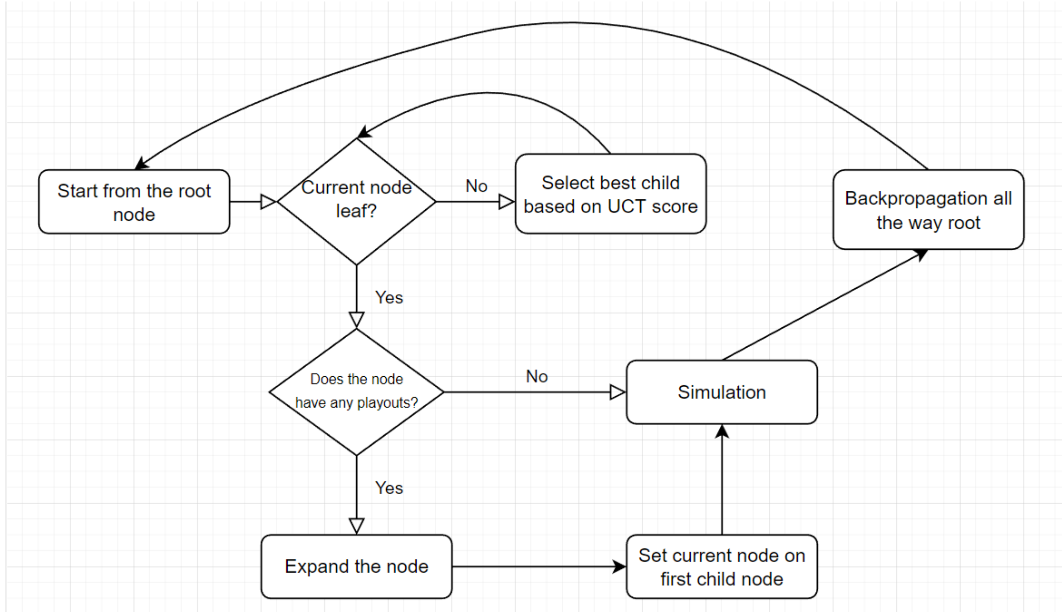

Figure 9: Flowchart of the MCTS algorithm

4 Experiments

For the experiments, two different algorithms will play against each other and their results will be noted. In each experiment, 100 games will be played where each player plays 50 games each side. In order to create variety between the 100 games, the first 2 moves of a game will always be random. The seed for the experiments is 123. The processor which the program is executed on is: Intel(R) Core(TM) i5-6500 CPU @ 3.20GHz.
Before the minimax player and the MCTS player play against each other, the heuristic function will be optimized of the minimax player. The MCTS does not have such function, but will be optimized on its constant C value in the UCB1 formula.

4.1 Optimizing Minimax

In the first experiment, the goal is to optimize the minimax algorithm as much as possible. It is known that the quality of the minimax algorithm heavily relies on its heuristic function, therefore it is important that certain aspects of the game hold the optimal correct value. The 3 aspects that will be optimized are:

- Value of a normal piece
- Value of a king piece
- Positional structure of the pieces

First, each piece of a player will be scored 5 points. The first step is to try to value the king piece as accurately as possible. The positional structure will not be taken account with yet. The x-axis shows how many points extra a king piece is worth compared to a normal piece. The depth of the algorithm will be 4 with the seed 123.

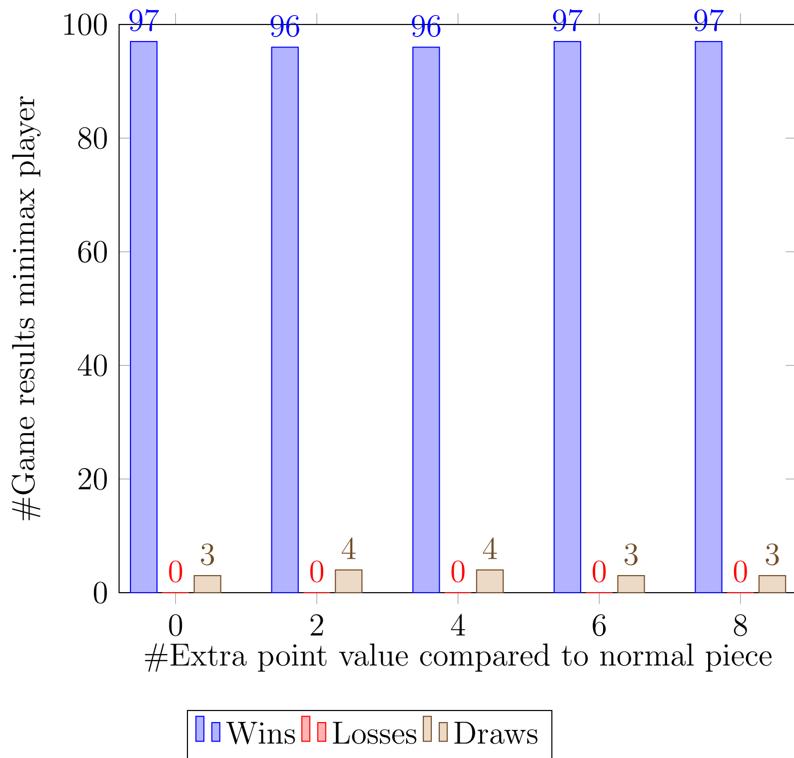

Figure 10: Minimax Player vs Random Player : king piece value

It appears to be that giving extra value to a king piece does not result into more wins for the minimax player when it is playing against a random player. The amount of wins does not necessarily go up or peak at a certain value for a king piece. A king piece is objectively a stronger piece than a normal one, so this result is quite surprising. However, this might be the cause that a random player is not a representative opponent to test the heuristic function. That is why this time two minimax players will play against each other where one puts emphasis on the king pieces and the other one does not.

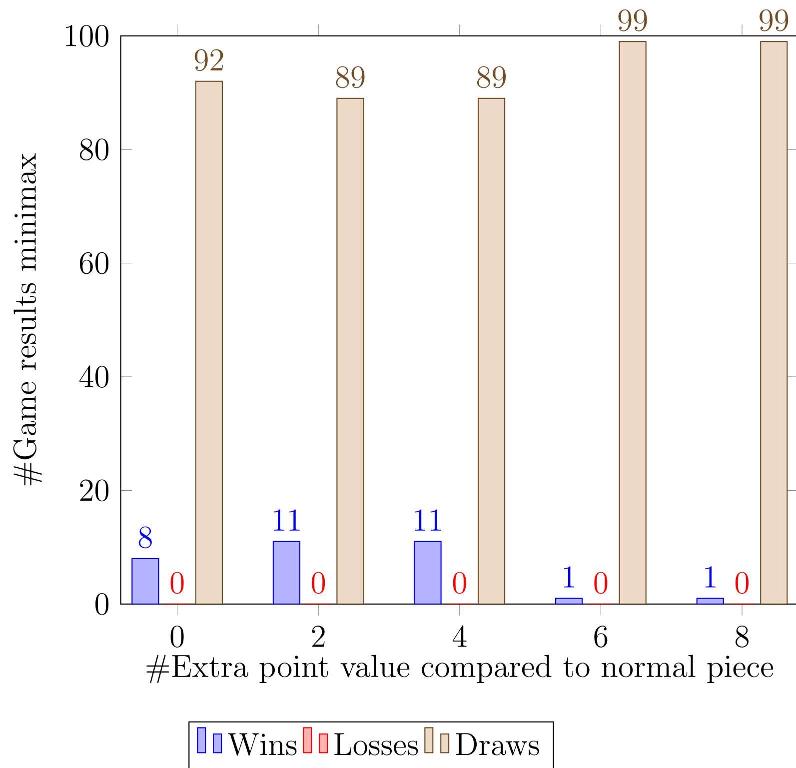

Figure 11: Minimax Player with emphasis vs normal Minimax Player: King piece value; The optimal value falls between 2-4

Apparently, when two minimax players play against each other, the importance of the king piece becomes more clear. Even though the difference in the won games is not much, which is around 10 percent, it can be seen that the minimax player who values the king piece with a value between 2 and 4, wins more games compared to other values. 3 extra points assigned for a king piece, the compared minimax player will now have this feature and the next experiment continues.

There are two aspects that will be looked at in a positional structure within this experiment. The first aspect is to take control of the center. Apparently it is advantageous for a player if their pieces are positioned in the center of the board since that's the place where the most actions occur. The second aspect is to keep the home row pieces as long as possible, since this rejects the opponent from promoting their piece. These aspects are based of the first 3 websites that pop up when searching in Google: "Tips for Checkers" [3] [2] [6].

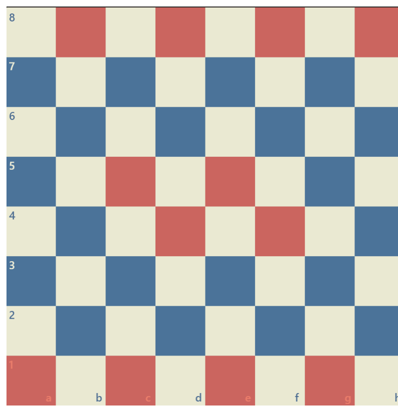

Figure 12: Home row and centre sqaures

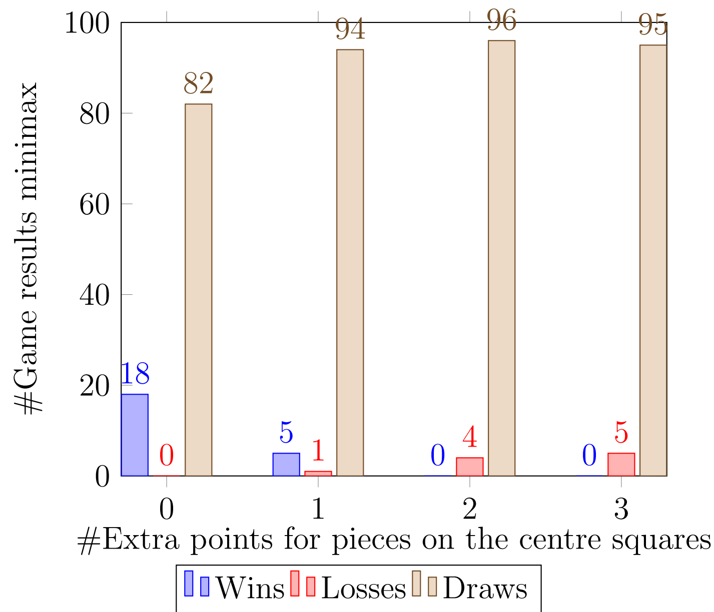

Figure 13: Minimax Player with emphasis vs Minimax Player: center pieces value; Performance worsens when the center pieces are rewarded a higher value

It appears to be that when there is extra score awarded for pieces within the centre, the game results in less wins and more draws or even more losses. It seems to have a negative effect on the game, so this aspect will be neglected.
In the results can be seen that when regarding positional aspects of a game, the minimax does not necessarily improve. In both cases, the algorithm is better off not using them, with getting more wins and fewer draws and loses. The following experiments will therefore use the minimax version, where a king piece is awarded 3 extra points compared to a normal piece, which is worth 5 points.

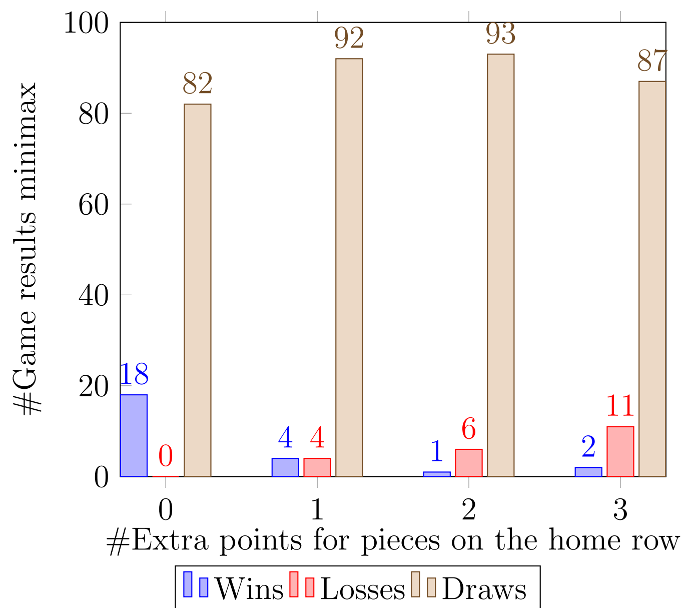

Figure 14: Minimax Player with emphasis vs Minimax Player : Extra piece value for home row pieces; erformance worsens when the home row pieces are rewarded a higher value

4.2 Comparing Minimax and Alpha-Beta pruning

The next experiment will revolve around the comparison between the regular minimax algorithm and the minimax algorithm with alpha-beta pruning based on the amount of nodes within the tree. The nodes of the tree are simply the amount of times the function has been called. The results are as follows:

Depth	Minimax	Minimax $\alpha\beta$	Runtime 100 games
1	570	570	<1 sec
2	3396	1867	<1 sec
3	15915	6008	0m5.973s
4	83875	16636	0m24.013s
5	506329	56151	2m41.158s
6	2400124	141112	12m43.326s
7	N/A	N/A	N/A

Table 1: Comparison amount of function calls and duration of 100 games played: Minimax vs Minimax $\alpha\beta$

The amount of function calls grow exponentially when the depth increases in both algorithms. However, with alpha-beta pruning, this is greatly reduced. The results show that even in depth 6 minimax $\alpha\beta$ only holding as much function calls as minimax in depth 4. In each depth, the regular minimax multiplies by approximately 5 whereas with pruning the amount multiplies approximately 3.

As seen in figure 9, the way the moves are loaded into the possible moves vector, the move which takes the most pieces is located at the end of the vector. Normally, the vector is read from beginning to end, but when using alpha-beta pruning, it might be more conventional to read the vector from end to beginning. The follow experiment will be the same as the last one, however this time the possible moves vector will be read from beginning to end to see if this is really more conventional.

Depth	Minimax	Minimax $\alpha\beta$	Runtime 100 games
1	577	577	<1 sec
2	3356	1948	<1 sec
3	15470	5945	0m5.973s
4	109556	21862	0m32.757s
5	583709	69652	2m33.5s
6	2985870	173232	12m49.294s
7	N/A	N/A	N/A

Table 2: Comparison amount of function calls and duration of 100 games played: Minimax vs Minimax $\alpha\beta$. Variation: possible moves vector is read backwards.

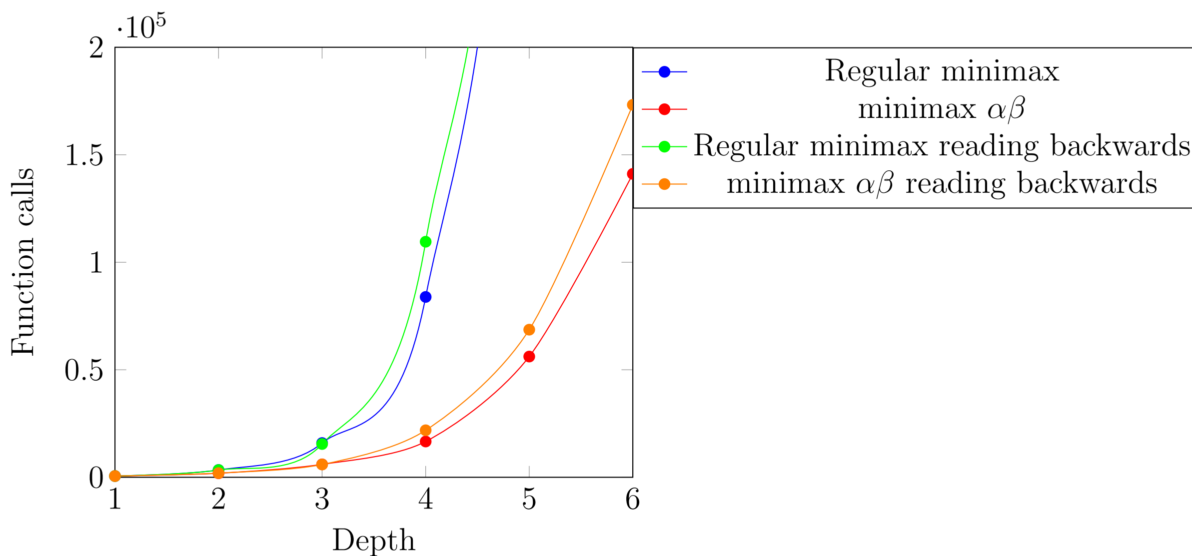

Figure 15: Number of function calls of all minimax variations

It appears to be that reading the possible moves vector from end to beginning does not result in less function calls. It actually increases the amount of function calls and therefore this feature is not used. Since the minimax $\alpha\beta$ does the same moves as the regular minimax algorithm with the only difference that the amount of function calls are lower, the minimax $\alpha\beta$ will be used for the upcoming experiments.

4.3 Optimizing MCTS

Before the MCTS player will play the minimax player, the MCTS player will be optimized first. Unlike minimax, MCTS does not have a heuristic function. However, the way how MCTS will be optimized is by optimizing the constant value C within the UCB1 formula. The MCTS player with 200 playouts will play against a random player and the results will be noted:

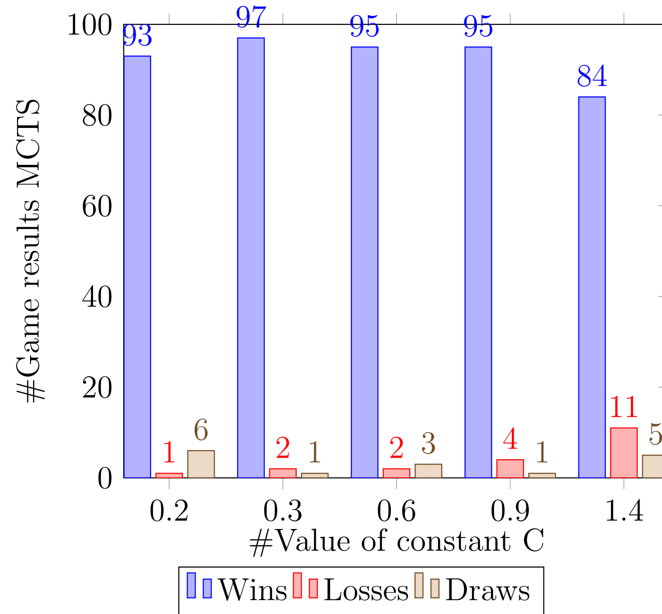

Figure 16: MCTS player vs Random Player

The differences between that amount of wins are very small, but it seems that the value constant value 0.3 works the best. Using this value will result in the most wins compared to the other constant values. Also, the value 1.4, which is close to $\sqrt{2}$, does not perform well at all. The value 0.3 will be used for the upcoming experiments.

4.4 Minimax vs MCTS

The final experiment will compare the minimax player and the MCTS player. In order to determine which player performs the best, the MCTS player will play with different amount of simulations against the minimax player with different depth. Each time, the MCTS player will play against a minimax player starting from 2 and going up, and will have playouts varying from 100 to 500. A winner will be determined if it takes over 10 minutes to run the experiment or that the results become stale.

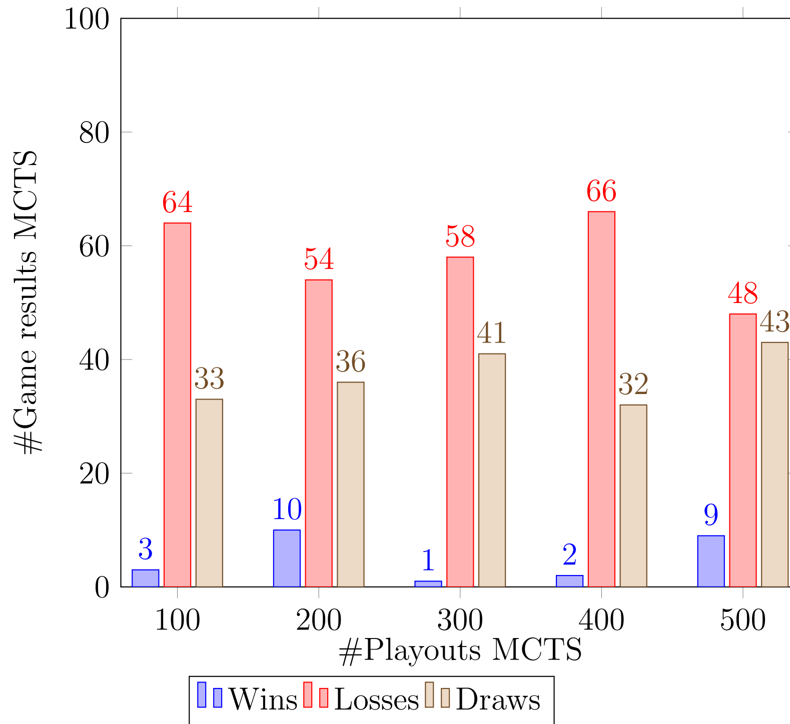

Figure 17: MCTS player vs Minimax Player (depth 2)

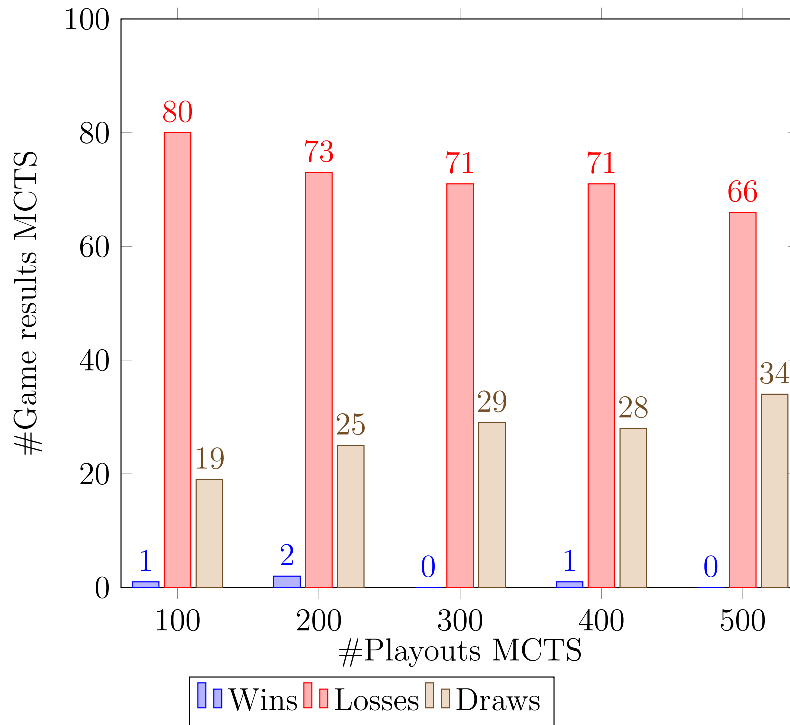

Figure 18: MCTS player vs Minimax Player (depth 3)

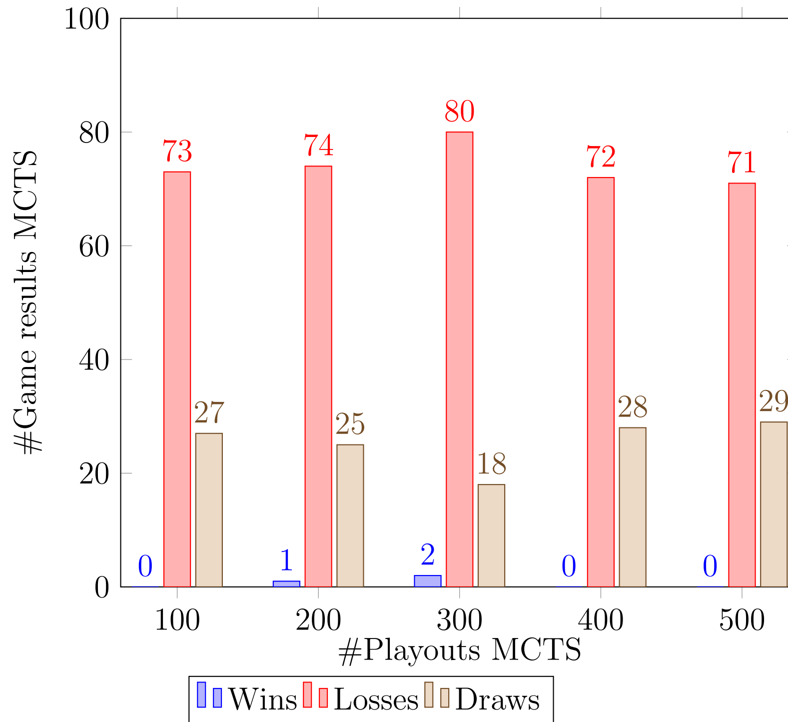

Figure 19: MCTS player vs Minimax Player (depth 4)

When looking at the results, it is clearly to see that the MCTS player is unable to beat the minimax player. Most of the games play result in a win for the minimax player. When playing against a minimax player with a depth of 2, the results are very inconsistent. The MCTS player wins by a lot at 200 playouts but loses relatively much at 400 playouts, but holds the amount of losses fairly well in 500 playouts. When playing a minimax player with a depth of 3, it is clearly to see that the strength of a MCTS player increases when it has more playouts. The amount of loses decreases when the amount of playout increases. Lastly, when playing against a minimax player with a depth of 4, this increase of strength is also visible. The amount of loses decreases in playouts 100, 400 and 500. In playouts 200 and 300, the amount of loses increase however, the MCTS player is able to get 1 or 2 wins.

Note that running these 100 games against the 2 players takes quite a while. This is mainly because of the MCTS player, since an estimation of the runtime of the minimax$\alpha\beta$ is already known. For a MCTS player with 100 playouts takes the 100 games around 3 minutes and for 500 playouts this duration will go up to 8 minutes.

5 Conclusions and Further Research

So to answer the research question: " Does the MCTS-algorithm outperform the MiniMax-algorithm in the game Checkers?", the answer for this research is no. The MCTS-algorithm with playouts varying from 100 to 500 fails to win against a minimax $\alpha\beta$ algorithm with a depth of 2 or higher.

To come to this answer, firstly, the minimax algorithm has been optimized. The king piece value is worth 3 points more than a regular 5 point piece. However, when awarding point for pieces standing on a specific spot, this does not have a positive effect on the minimax player. This might be because the known tips for Checkers are handy for humans, whereas with the minimax algorithm, these problems are already foreseen by the algorithm, and therefore it is not needed to have extra precautions. Secondly, the constant value C of the UCB1 formula of the MCTS player has been optimized. The best value turned out to be 0.3 and the usual value $\sqrt{2}$ was relatively weak.

At the end, the MCTS player and the minimax player played against each other. The minimax player was undoubtedly the better player of the two by getting way more wins. At some experiments, the increasing strength with more playouts for the MCTS player is more visible than others. The first two moves of each game were random, so there is a chance that the first two moves would result in a significant result to one player, and that could explain the sometimes inconsistent results.

The reason why the MCTS player is losing against a minimax player might be because the algorithm is not always able to find the optimal move and therefore ends up in a losing position and which would result in losing the game. In addition, the runtime of the MCTS algorithm is also way longer than the minimax algorithm.

There are many variations for the game Checkers, hence it might be interesting to see how the comparison works with a Checkers variation with a bigger search space like Draughts. Minimax has the advantage with games where the amount of moves is limited and where the position can be easily evaluated. However, with bigger games, searching the whole game tree and evaluating this might be more difficult and the MCTS might gain the upper hand.

It is remarkable that the MCTS performs this badly compared to the minimax algorithm, since it is known that the MCTS algorithm is considered to be quite a strong algorithm for playing games. The outcome of this thesis might be heavily dependent on the amount of playouts the MCTS algorithm has. The MCTS algorithm is most often written in Python instead of C++, so MCTS might be more efficient when run in Python. The other option would be to make the current code efficient for more playouts. Either way, these options have the goal to greater the amount of playouts for the MCTS algorithm, so more tests can be done for the comparison between the two algorithms.

References

[1] Michael H. Albert, Richard J. Nowakowski, and David Wolfe. *Lessons in Play, An Introduction to Combinatorial Game Theory*. CRC Press, 2nd edition, 2019.

[2] Howard Allen. Checkers strategy and tactics: How to win every time. https://hobbylark.com/board-games/Checkers-Strategy-Tactics-How-To-Win, HobbyLark, 2022. Accessed: 2022-07-07.

[3] Seth Brown. Basic strategies for winning at checkers. https://www.thesprucecrafts.com/how-to-win-at-checkers-411170, the spruce crafts, 2020. Accessed: 2022-07-07.

[4] Said Krebbers. Monte carlo tree search for dots-and-boxes. Bachelor's thesis, Universiteit Leiden, Leiden, Netherlands, 2021.

[5] Brent Gorda Brent Knight Robert Lake Paul Lu Jonathan Schaeffer Nathan Sturtevant Steve Sutphen Duane Szafron Ken Thompson Norman Treloar Martin Bryant, Joe Culberson. Chinook. https://webdocs.cs.ualberta.ca/~chinook/project/. Accessed: 2022-06-27.

[6] (n.d). Tips to win checkers. https://www.ultraboardgames.com/checkers/tips.php. Accessed: 2022-07-07.

[7] Richard J. Nowakowski. *Games of No Chance*. Cambridge University Press, 1994.

[8] Stuart J. Russell and Peter Norvig. *Artificial intelligence : a modern approach*. Pearson, 4th edition, 2020.

[9] Jonathan Schaeffer, Robert Lake, Paul Lu, and Martin Bryant. Chinook the world man-machine checkers champion. *AI Magazine*, 17(1):21, Mar. 1996.