Universiteit Leiden
The Netherlands

Opleiding Informatica

Optimizing Monte Carlo Agents
for the Game Katarenga

Tayfun Aygün

Supervisors:
W. A. Kosters and J. M. de Graaf

BACHELOR THESIS

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

July 6, 2021

Abstract

We will examine the Pure Monte Carlo and the Monte Carlo Tree Search agent for the board game KATARENGA. In order to optimize the two agents, multiple experiments are conducted. The goal is to optimize the agents to be as effective as possible and try to keep the resources used minimal to make faster decisions. We will experiment with various parameters, scores, board sizes and board lay-outs. The Pure Monte Carlo agent without any modifications performs worse than the Monte Carlo Tree Search agent. By analyzing the results, the Pure Monte Carlo agent has more options to optimize in contrast to the Monte Carlo Tree Search agent. As for the final test, both agents are optimized and put against each other for a hundred games twice. The losing Pure Monte Carlo agent now goes even against the Monte Carlo Tree Search agent.

Contents

1 Introduction

When going out to get a drink, do you want a cup of coffee or a cup of tea? Making this decision should not be that difficult. However, the decision is much more difficult when buying a car, because you get many more options to choose from, for example the car brand, model, chassis and color. But what happens when you have to make a decision when you have more than one million options? Evaluating every available option would take too long, if not be impossible. A way to tackle that problem is to create an artificial intelligence agent which would make the decision for you. This agent analyzes its environment and takes actions that maximize the chance of success. The agents we will primarily focus on are the Pure Monte Carlo agent and the Monte Carlo Tree Search agent. These agents rely on random sampling to get results. We let these agents play the board game Katarenga.

Artificial Intelligence (AI) is the ability of a computer to perform tasks which are commonly associated with human intelligence. These AI systems learn to make decisions which would normally require a human level of expertise. Using sensors, digital data or remote inputs, an AI agent combines information and analyzes the material instantly and acts on the insights derived from those data [Wes].

The board game Katarenga is played by two players. These players could either be the random player, the Pure Monte Carlo player or the Monte Carlo Tree Search player. The goal of the research is to optimize the Monte Carlo methods. The optimizations are done by making "better" decisions to ultimately win more games and also executing the algorithm in less time and winning faster. We will examine the Pure Monte Carlo and the Monte Carlo Tree Search agents for the game Katarenga.

1.1 Thesis overview

In Section 3 the rules of the game will be thoroughly explained. This includes the starting positions, how to move the pawns, how to score points, when the game is finished and the different properties of the tiles on the board. Section 4 explains how the Pure Monte Carlo and the Monte Carlo Tree Search agents work. The agents build a game tree for a given game state in which random play outs give scores to the nodes in the tree. Nodes with a higher score, maximize the chance of success. After finishing the game tree, the node with the highest score will be chosen as the best move to play. Afterwards, we will go over the main subjects of the experiments in Section 4.4 and quickly follow it up with the experiments in Section 5. Finally, Section 6 concludes.

This research was conducted as a bachelor thesis at LIACS, supervised by: W.A. Kosters and J.M. de Graaf.

2 Related work

The Monte Carlo method is a successful method of performing repeated random sampling in order to obtain numerical results, which would otherwise be highly complicated to calculate without repeated random sampling. The Monte Carlo method depends on the power of computers to simulate large numbers of events. These simulations are based upon random sampling and probability distributions.

A game very similar to KATARENGA is CHESS. One of the strongest artificial intelligence agent created for CHESS is AlphaZero, created by DeepMind [SHS+18]. AlphaZero makes use of a general reinforcement learning algorithm. The algorithm learns by deep convolutional neural networks, trained solely by reinforcement learning from self-played games. The AlphaZero agents also makes use of the Monte Carlo Tree Search algorithm, which we will go through in Section 4.3.
A similar research to this bachelor's thesis is done by Robert Arntzenius for his own bachelor's thesis [Arn21], in which he experiments with the Pure Monte Carlo agent and the Monte Carlo Tree Search agent in the board game ONITAMA [Sat].

3 Game rules

In this section the game KATARENGA will be explained. For our research of the game, we will take a slightly modified version of the original game. This modified version will be explained in Section 3.1.

3.1 Game components and starting position

This board game is a slightly modified variant of the board game KATARENGA [Par]. This modified game of KATARENGA is a 2-player game, played on a board with n by n tiles, together with n white pawns and n black pawns. The players play against each other and start on opposite sides of the board. The white pawns belong to player 1, who shall be referred to as player White and the black pawns belong to player 2, who shall be referred to as player Black. Each player puts their own pawns on the first row of their side of the board, as seen in Figure 1. The tiles of the board have different properties, which are indicated by a letter. These tiles are randomly generated every game. The white player is first to move when the game starts.

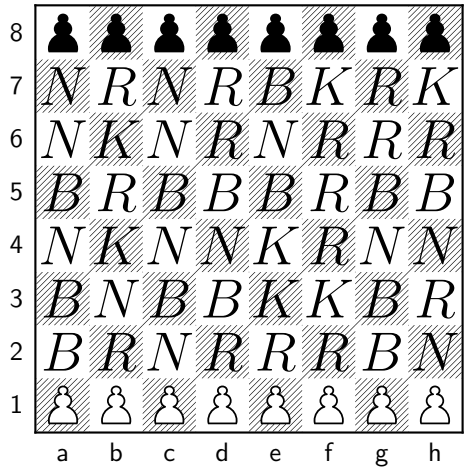

Figure 1: Starting position of the game KATARENGA.

3.2 Tile properties

The tile where the pawn is placed at determines the moves the pawn can take. There are four types of tiles in total. Every tile has a letter on it indicating what type of tile it is. Similar to Chess, the four types of tiles are:

- (K)ing tile: The pawn can move to any adjacent tile, horizontal, vertical or diagonal.
- (R)ook tile: The pawn can move up to and including three tiles horizontally or vertically.
- (B)ishop tile: The pawn can move up to and including three tiles in any diagonal.
- K(N)ight tile: The pawn can move two tiles horizontally and one tile vertically, or two tiles vertically and one tile horizontally. The knight's movement has the shape of an L.

Unlike Chess, the rooks and bishops can only move up to three tiles in their respective directions. In Figure 2 the moves of each tile are visually shown.

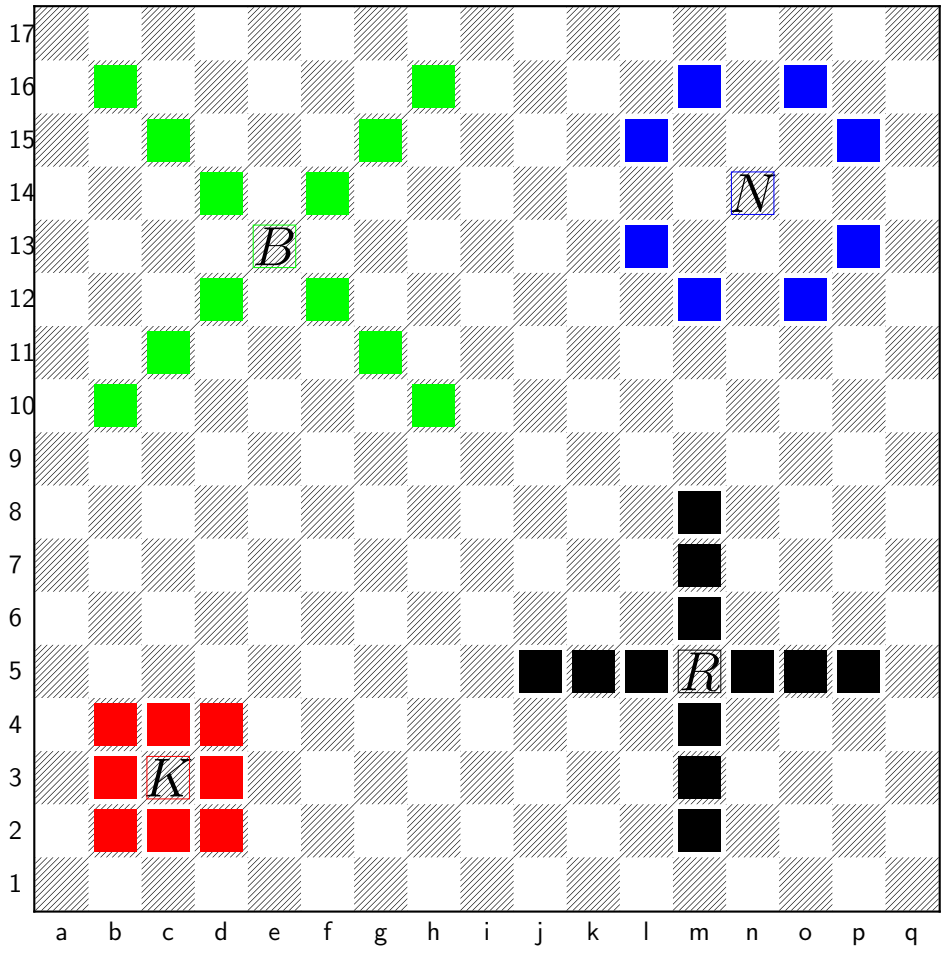

Figure 2: Tiles B, K, R and N with their possible moves.

3.3 Legal and illegal moves

In addition to the tile properties, there are a few more rules in order to make a move legal:

- A pawn can not move to a tile where there is another pawn of the same color.
- A pawn can move to a tile where there is another pawn of the opposite color, hereby taking the opposite pawn.
- A pawn can not move outside the board, unless the pawn gets taken or the pawn scores a point, see below.
- The rook and bishop can not "step over" pawns. This means that a rook or bishop can not move behind other pawns on their respective moves.
- The knight, on the other hand, can "step over" pawns.
- The turn switches to the opposite player after every legal move that is played.

3.4 Wins, draws and scoring points

The only way to win a game is to score two points. A point can be scored when one's pawn reaches the other side, the row where the opponent starts. The turn after one's pawn reaches this row, the player can take the pawn off the board to score one point. Scoring a point is not mandatory, so the player can also play other moves if he/she wishes to. The first player to score two points, wins the game.
If a player has no pawns left on the board, the other player scores one point for every pawn on the board. After the points are scored, if the other player has scored two or more points, that player wins.
Finally, a draw occurs when there are no more pawns left on the board and neither player has scored two points.

4 The three agents

For conducting the experiments we will develop three agents: the random agent, the Pure Monte Carlo agent and the Monte Carlo Tree Search agent.

4.1 The random agent

The random agent, as the name suggests, plays completely random. The random agent does not keep any sort of score or state evaluation. Out of all possible moves, the agent will randomly choose one of the moves and play it. All possible moves have the same probability to be chosen.

4.2 The Pure Monte Carlo agent

The Pure Monte Carlo agent is an algorithm for finding good decisions. It is a much faster algorithm than the Monte Carlo Tree Search algorithm. However, the algorithm loses accuracy for its speed. The algorithm works by keeping scores based on random play-outs. Given a game state, random sampling will be performed on each possible move
Random sampling is done by playing out a fixed number X of games for each possible move, in which both players play randomly, the so-called *play-outs*. For every possible move a score is kept. The score is the number of wins from the X randomly played games.
After the random sampling has finished, the algorithm picks the move with the highest score as the best move. In case of a tie between the scores of multiple moves, a move will be randomly selected among these moves.

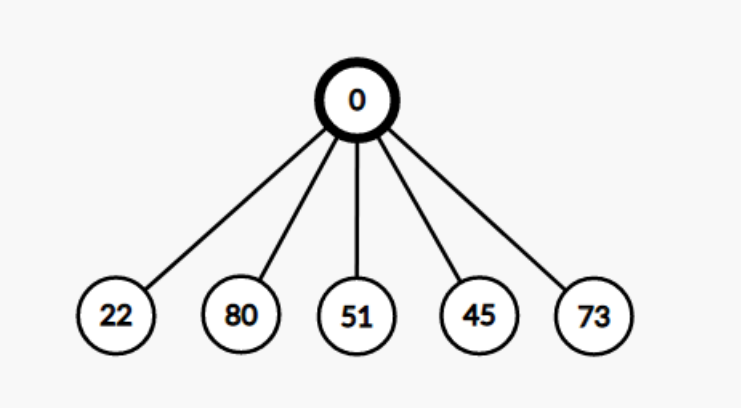

Figure 3: The Pure Monte Carlo agent, with one hundred play-outs.

Figure 3 shows the Pure Monte Carlo agent with one hundred play-outs per possible move. In this case, the move with a score of 80 will be chosen as the best move.

4.3 The Monte Carlo Tree Search agent

The Monte Carlo Tree Search (MCTS) algorithm [Cou06, BPW+12] is a heuristic search algorithm for finding good decisions. The Monte Carlo Tree Search algorithm is more accurate than the Pure Monte Carlo algorithm, however it gives up speed for accuracy.
The Monte Carlo Tree Search algorithm builds a game tree in which all of its nodes are game states. The root of the game tree is the current game state. The algorithm makes use of four different steps to find the "optimal" move.

The four steps are:

- Selection
- Expansion
- Simulation

- Backpropagation

The purpose of the four different steps, shown in Figure 4 will be explained below.

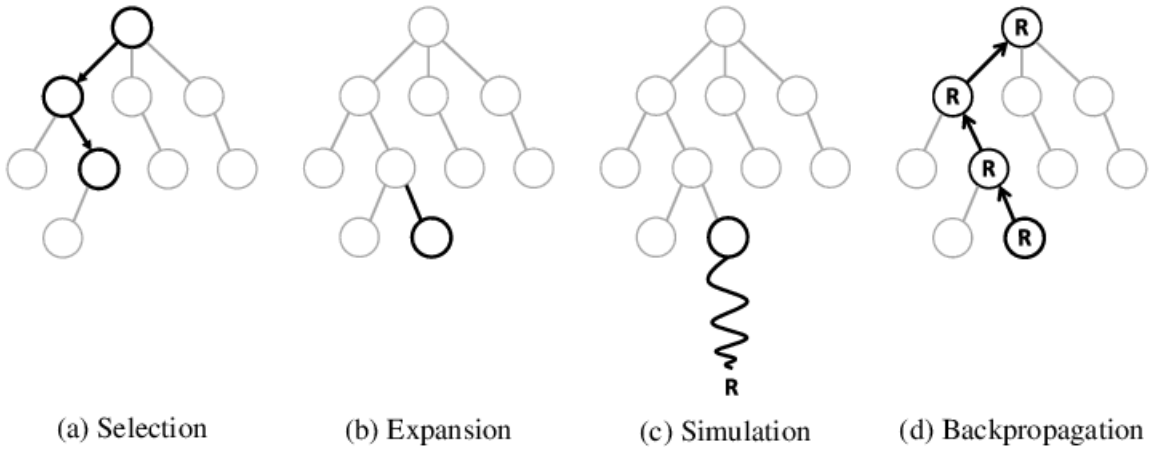

Figure 4: The steps of the Monte Carlo Tree Search agent [JKR17].

Selection

In the selection process, a node in the game tree is selected which is the most promising node at that moment, see Figure 4. Selection is based upon the UCT score [KS06], which stands for Upper Confidence bounds applied to Trees. The UCT Score is calculated by the formula: $w_i/n_i + c\sqrt{\ln(N_i)/n_i}$, where

- w_i: number of wins in node i.
- n_i: number of games played in node i.
- N_i: number of games played in the parent of node i.

The first component of the formula (w_i/n_i) captures the exploitation of the game tree, the second component of the formula $(c\sqrt{\ln(N_i)/n_i})$ captures the exploration of the game tree. The constant c is a parameter that we will be experimenting with, and we will look at the values where c gives the best results.

Exploration and exploitation

Finding the current most promising node in the game tree requires a good balance between exploration and exploitation,

- Exploitation (w_i/n_i): To exploit nodes which currently have a high win ratio.
- Exploration $(\sqrt{\ln(N_i)/n_i})$: To explore nodes which have none to few simulations.

The Monte Carlo Tree Search agent gets allocated a limited number of play-outs for every turn. A good balance would be to not only allocate these resources to exploit nodes with a high win ratio, but also to explore nodes which have none to few simulations done. Unexplored nodes may potentially contain good or winning moves, so nodes should not be neglected. With a balanced management of resources, the current most promising node is chosen and built further upon in the following steps.

Expansion

When a node is selected, the expansion step first checks whether the state of this node is a terminal node. If this is the case, the UCT score of the node gets updated and the simulation process will be skipped.
If the selected node is not in a finished game state, the node gets expanded. All of the current nodes' child nodes get created, and one of the child nodes gets selected randomly. Then this node is sent to the simulation process.

Simulation

In the simulation step from the node which has been selected, a random play-out of the game will be started. At the end of the play-out, the game ends in a win, loss or a draw. The UCT score will be updated with the new win ratio. This node will then be sent to the last step.

Backpropagation

After the simulation step has finished, the node's UCT scores gets updated, however the score of all ancestor nodes also need to be updated. Backpropagation updates the scores of the node and all of its ancestor nodes until the root. The root is not updated, as it can not be a legal move.

Repeat

These four processes are repeated a number of times. The more simulations are done, the more accurate and slower the algorithm gets.

The best move

After the algorithm is finished, the child node with the most simulations, will be chosen as the best move to play. This node may not always have the highest UCT score, however it is the most consistent best move.
When there is a tie between multiple nodes with the most simulations, one of these nodes will be randomly chosen, with all of the nodes having the same probability to be chosen.

4.4 Aspects

There are five main aspects we will focus on:

- Different board sizes and tile usage

- The number of simulations
- The c parameter from MCTS
- The depth of play-outs
- The heatmaps

The experiments are done by letting the three different agents, random player, Pure Monte Carlo player and the Monte Carlo Tree Search player, play against each other, with different board sizes and parameters.
Different playing styles for different board sizes are expected. A small board means each player only has a few pawns to play with. For example, on a 4×4 board, each player has 4 pawns to play with. This makes it more attractive to capture enemy pawns because they are so valuable. The player may win the game by capturing the opponent's pawns, instead of scoring points. On the other hand, if one plays on big board sizes, for example a 20×20 board, each player has 20 pawns. This makes the individual pawns less valuable, because capturing one pawn out of four is much more gainful than capturing one pawn out of twenty. On top of that, scoring a point on small boards by sacrificing a pawn could be more harmful than beneficial since the pawns are so valuable. On a big board, it is expected to win more often by scoring two points, than by capturing all of one's opponent pawns. Capturing twenty pawns instead of four is more difficult and also takes more time.
The number of simulations should lead to more consistent results, the more simulations have been done. This goes at the expense of the execution time. The c parameter in the MCTS agent decides the exploration or exploitation. The higher the c parameter, the more exploration will occur. More exploration goes at the expense of less exploitation and vice versa.
The depth of the play-outs also impacts the efficiency of the agents. It is better to win a game in five turns, than in fifty turns. So for making better decisions, the agents also have to consider the depth of the play-outs.
Finally, we will create heatmaps of the board. These heatmaps show the amount of moves played from each tile in the board. Every tile has a counter for every time a move has been played from that tile and this counter is shown as a color. The darker the color of a tile, the more often a move is played from that tile. The heatmaps show the locations where pawns have been played and we will use this information to better understand the strategies of the agents.

Different board sizes and tile usage

We will analyze the behaviour of the agents, by looking at which tiles the agents play. Since the board is random for each game, we are interested in the number of moves done per tile. We do not take much interest in the win percentages for this experiment and put the primary focus on the tile usage for this experiment.
We count the moves for each of the four tiles, king, rook, bishop and knight tiles, at various board sizes to analyze the strategies for both agents.

The number of simulations

We express the time the algorithm runs in the number of simulations the agent has done. Within these simulations, we do not count the moves in the play-outs. A simulation is done every time a

move has been played and the game tree is updated. The number of simulations will play a big part in creating an efficient agent.

MCTS c parameter

The c constant is there to adjust the number of explorations. A high c value encourages to select nodes which have not been visited in a while. This is to not neglect nodes with a relatively low number of simulations. A node with a low number of simulations is inaccurate and does not reflect the actual likeliness to win the game through this node. Finding a value for the parameter c in order to balance exploration and exploitation should result in better resource management and hopefully better results.
The c parameter was introduced by [KS06]. The article proposed a value of $\sqrt{2}$ for the c. We will start our experiments with this c value as the baseline, then compare it with experiments with modified c values.

Depth of play-outs

The depth of play-outs counts the simulations done for a play-out and is saved for each possible move. When considering which move to play, it is also important to consider the depth of the play-out, as it is better to win in five turns than in fifty turns. The scores for the moves will be penalized according to the depth of the play-outs. The bigger the depth, the bigger penalty the move receives.

Win percentage

The game Katarenga has not been solved yet, which means that there is no solution for the game which will always result in a win or a tie. However, because the game tree of possible moves is finite, the game is solvable.
Take for example the game tic-tac-toe. Tic-tac-toe is a solved game because of the very small finite search space. Because of the small possible move pool, optimal strategies have been found for tic-tac-toe.
This also applies to a more similar game to Katarenga, the board game Checkers, also known as Draughts. The Checkers board game has been solved recently [Sch07]. Assuming neither player will make a mistake, the game will always end in a draw.
As for Katarenga, much like the game Chess, there could still be optimal strategies which just have not been found yet. The game tree of possible moves for this game is so enormous, there currently is no existing computer which has the computing power to solve this game.
In order to create an efficient agent, we strive for an agent who is also winning. To begin, we have to define what efficiently winning is. When looking at playing one match it is simple to conclude when an agent is winning. The game can finish in three states: win, lose or draw. However when looking at, say, a hundred games, when do we call the agent winning?
Do we call the agent winning if it wins 51 of the hundred matches? Or is it winning starting at seventy wins? Or do we call the agent winning, only if he wins all hundred games?
To address this problem, we will have to analyze the results and define efficiently winning ourselves. The goal is to have a high win percentage, with the fewest possible simulations.

Heatmaps

On top of analyzing the tile usage, we will also analyze a heatmap for each agent. For this experiment the tiles on the board will be randomized once and fixed for fifty games for each agent.
If the board was randomized every game, the heatmap would not be of much use since the heatmap would show different strategies for each board and show it as one whole strategy.

5 Experiments

The experiments on the six topics will be conducted in this section, as well as two extra experiments: If there are any beneficial boards and the final test of the optimized Pure Monte Carlo against the optimized Monte Carlo Tree Search agent.

5.1 Efficiency of the Pure Monte Carlo agent

We start with experimenting with the Pure Monte Carlo player as player White. We will examine the change in win percentages by the number of play-outs per move for the Pure Monte Carlo player. We will let the Pure Monte Carlo agent (with different play-outs) play against the three standard agents:

- Random player
- Pure Monte Carlo player (with 600 play-outs per move)
- Monte Carlo Tree Search (with $c = \sqrt{2}$, 1600 play-outs per move)

The standard agents are chosen in such way that they perform quite well, without taking too much time running fifty games on a 8×8 board. The number of play-outs was chosen by looking at the amount of simulations done, then choosing for a high number of simulations for the Pure Monte Carlo agent and a low number of simulations for the Monte Carlo Tree Search.

Play-outs

We will express the efficiency in terms of the number of simulations done, because the number of play-outs is not consistent. For example, the Pure Monte Carlo agent with 600 play-outs per move will do approximately 50.000 simulations on a 4×4 board and approximately $17M$ simulations on a 8×8 board.
Figure 5 shows the win percentages of the Pure Monte Carlo agent with play outs starting at 10 up to 1200. The random agent has been left out of the graph, because the Pure Monte Carlo quickly won with a win percentage of 100% at a very low number simulations.
We can see that the win percentage increases more slowly, the more simulations are done. Two notable data points, with the total number of simulations, are:

- Pure Monte Carlo (17M) vs Pure Monte Carlo (17M): $\sim$50% win percentage
- Pure Monte Carlo (2M) vs Monte Carlo Tree Search (2M): $\sim$40% win percentage

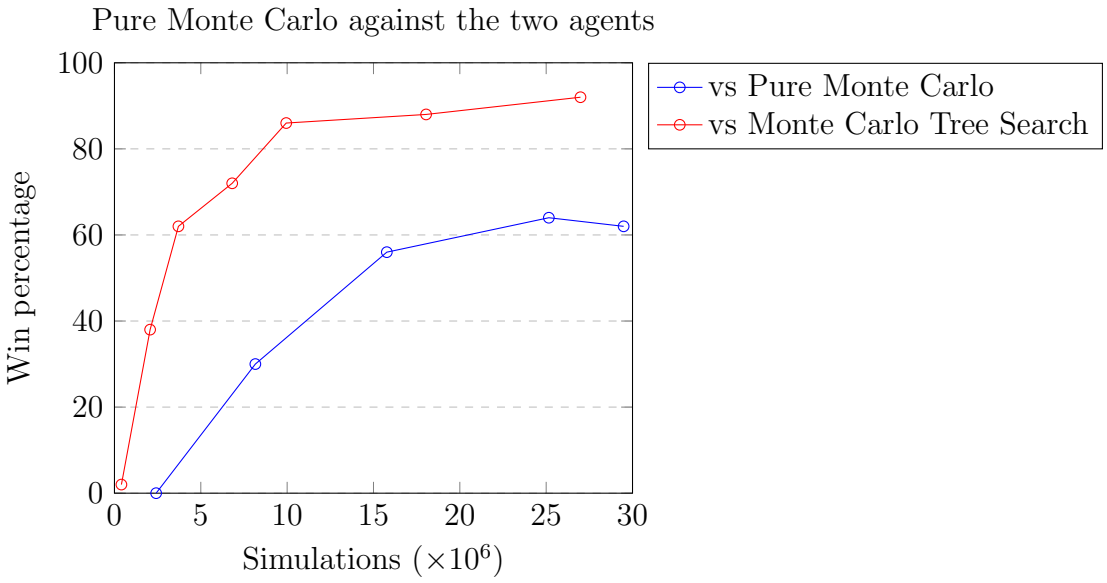

Figure 5: Win percentages of different simulations with Pure Monte Carlo.

The first data point suggests that there is no advantage of playing as the white or black player, as the win percentage is equal for both sides with equally smart agents.
The second data point suggests that the Pure Monte Carlo agent loses against the Monte Carlo Tree Search agent, when the agents allocate the same amount of resources. So at $2M$ simulations, the Monte Carlo Tree Search agent is performing better than the Pure Monte Carlo agent.

When is the agent efficient?

In order to examine when the agent is efficient, we have to analyze the gains the agent gets with increasing the number of simulations. This will be done by extracting the following data:

- p: Number of play-outs in range [10–1200].
- $G_{\text{win(p)}}$: Win percentage gain (%), calculated by: (Win percentage with p play-outs) - (Win percentage with $0.5p$ play-outs).
- $G_{\text{sim(p)}}$: Simulations gain, calculated by: (simulations with p play-outs) $-$ (simulations with $0.5p$ play-outs).
- G_p: Percentage point gain per million simulations, calculated by: $G_{\text{win(p)}}/(G_{\text{sim(p)}} \times 10^{-6})$.
- R_p: The ratio of simulations between player 1 and 2, calculated by: (simulations player 1 with p play-outs) / (simulations player 2 with p play-outs).

We will need to analyze the G_p at different R_p. Between each data point in Figure 5 these data will be calculated and put in the following figures.
In Figure 6, the Pure Monte Carlo plays against an equally smart agent, of the same type. The graph shows a decreasing trend, which means the G_p decreases the more simulations are being done.

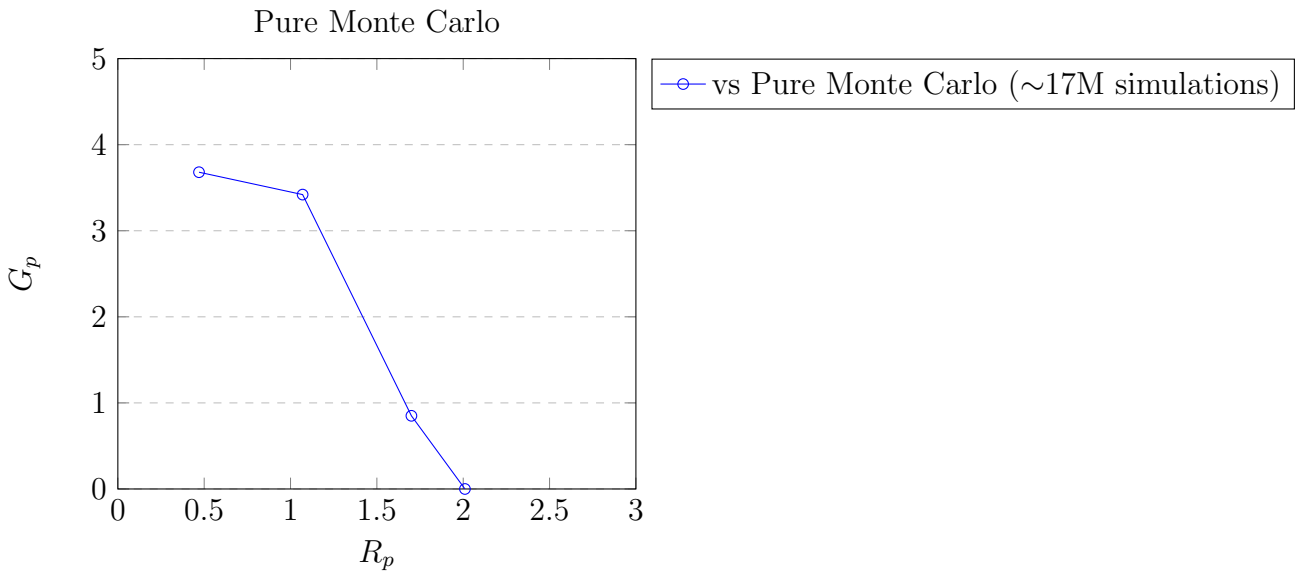

Figure 6: Win percentage gain at various simulation ratios.

A G_p of x means for every million extra simulations, you would gain a win percentage of $x\%$. When dropping below a G_p value of 1, you gain less than a percentage point per million simulations. For continuing the research, we assume our "efficiency" stops at $G_p = 1$.
Against an exactly equally smart agent, the G_p drops below 1 at $R_p \approx 1.65$.

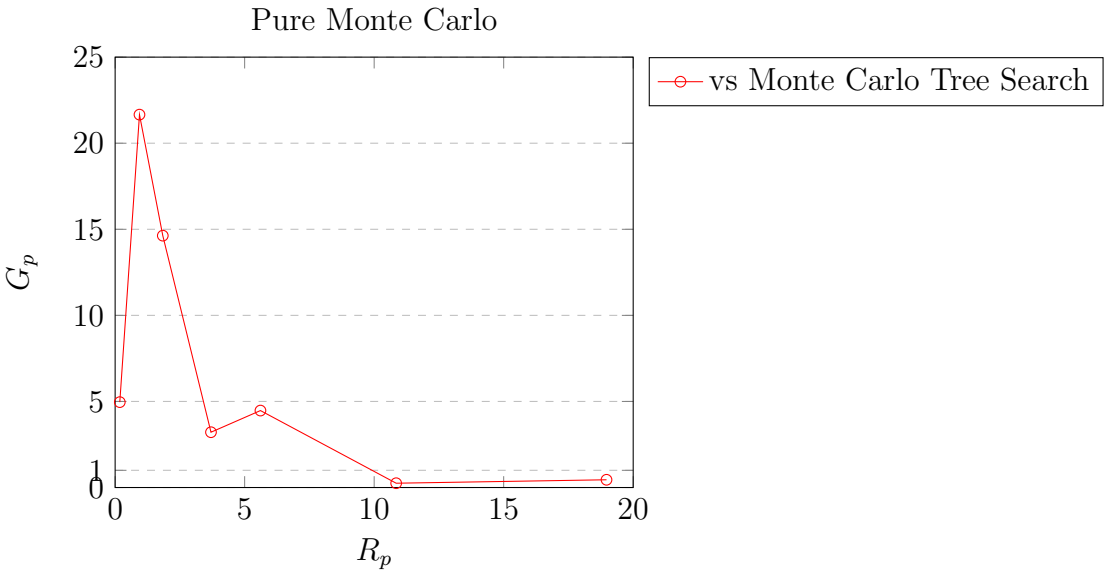

Figure 7: Efficiency against Monte Carlo Tree Search (∼1.8M simulations).

Figure 7 shows the results of the Pure Monte Carlo agent against the Monte Carlo Tree Search agent. Figure 5 had shown that the Pure Monte Carlo agent lost against the Monte Carlo Tree Search agent, when both agents had approximately $2M$ simulations done. So there is a difference

in the cleverness of the agents.
This time, we take an agent at a much smaller number of simulations (2M where the previous agent had 17M).
We can see the line spike at a much higher value than the previous graph. The reason for this might be that the Monte Carlo Tree Search agent has not learned nearly as much as the Pure Monte Carlo agent with 17M simulations.
We assume that the Pure Monte Carlo agent stops being efficient at $G_p = 1$. This results in a R_p of 10. So the less the opposing agent has been developed, the longer the agent will be efficient. This also explains why the agent in Figure 5 had higher win percentages against the Monte Carlo Tree Search agent, than the Pure Monte Carlo agent.

5.2 Efficiency of the Monte Carlo Tree Search agent

Now the same experiments are done on the same three standard agents, however this time for the Monte Carlo Tree Search agent as player White.

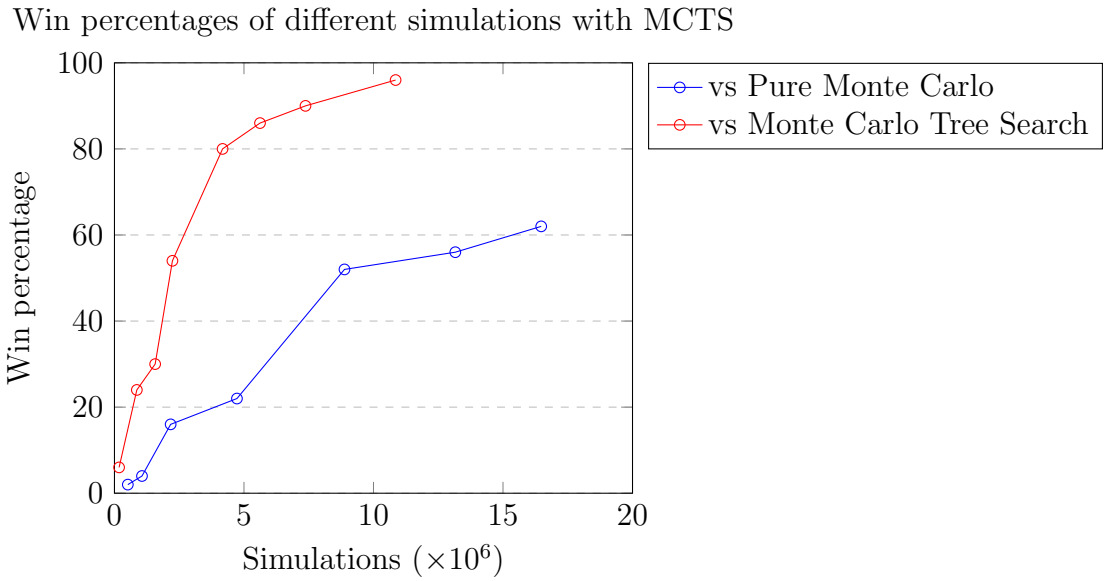

Figure 8: Win percentages of different simulations with Monte Carlo Tree Search.

Again, Figure 8 seems to be following the same trend of lines, which are the lines that increase more slowly over time. Compared with Figure 5, we see that the Monte Carlo Tree Search agent requires a much smaller number of simulations for the same win percentages. Notable data points are:

- Monte Carlo Tree Search (17M) vs Pure Monte Carlo (17M): ∼60% win percentage
- Monte Carlo Tree Search (2M) vs Monte Carlo Tree Search (2M): ∼50% win percentage

The first data points suggests that the Monte Carlo Tree Search wins against the Pure Monte Carlo agent, when the agents allocate the same amount of resources. So at $17M$ simulations, the Monte Carlo Tree Search agent is performing better than the Pure Monte Carlo again.

Win percentages comparison			
AI Agents	Win percentage (%)	Simulations ($\times 10^6$)	Speedup
Pure MC vs Pure MC	60	20	1.00
MCTS vs Pure MC	60	15	1.33
Pure MC vs MCTS	80	8	1.00
MCTS vs MCTS	80	4	2.00

Table 1: Performance ratios with the Pure Monte Carlo agent as baseline.

The second data point suggests that there is no advantage of playing as the white of black player, as the win percentage is equal for both sides with equally smart agents.
We can see that the Monte Carlo Tree Search agent makes better use of its resources, as shown in Table 1. The Monte Carlo Tree Search agent requires fewer simulations for the same win percentage as the Pure Monte Carlo agent.

When is the agent efficient?

We analyze the data for the Monte Carlo Tree Search agent in the same way we did for the Pure Monte Carlo agent.

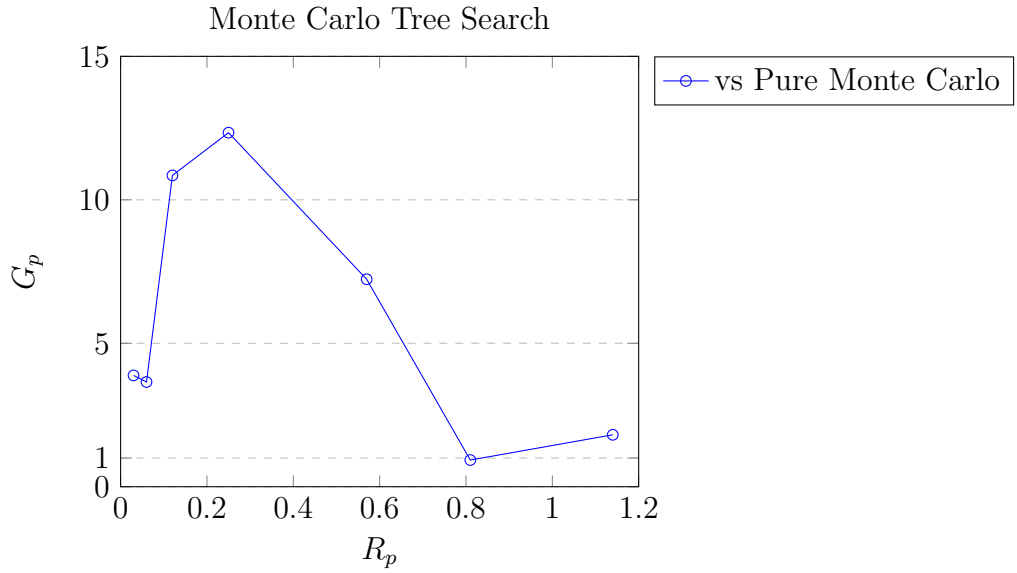

Figure 9: Efficiency against Pure Monte Carlo (∼17M simulations).

Against the relatively smart Pure Monte Carlo agent, with around 17 million simulations done, the G_p value drops below 1 at around R_p of 0.8. So the Monte Carlo Tree Search agent wants to focus on reducing the number of simulations done, rather than doing more simulations than the opponent for higher win percentages. According to the value of R_p, which is 0.8, the Monte Carlo Tree Search agents wants to stop simulating at about 80% of the opponent's total number of simulations.

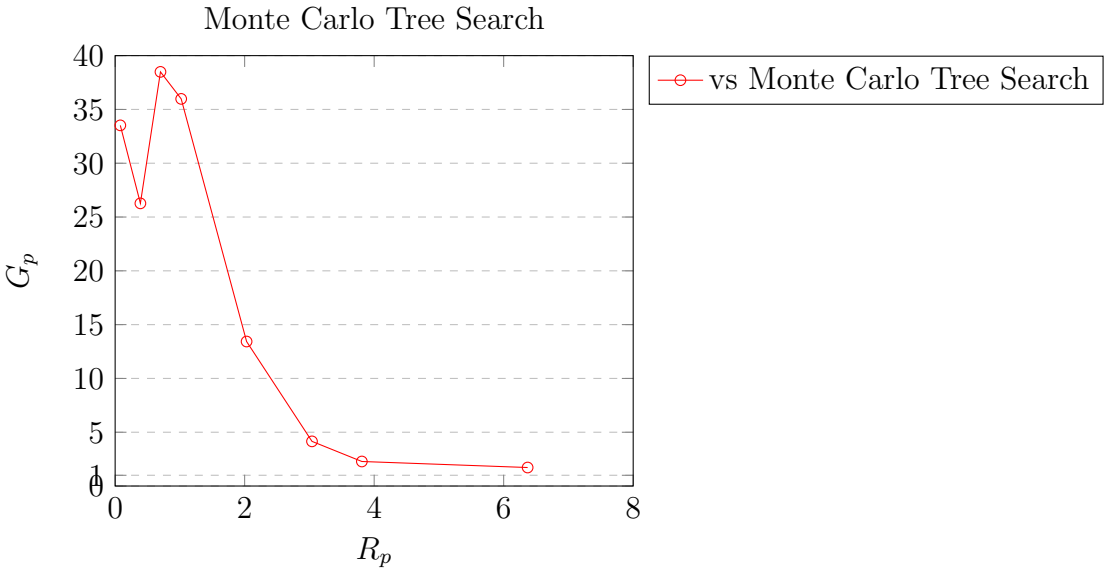

Figure 10: Efficiency against Monte Carlo Tree Search (∼2M simulations).

Following the line of Figure 10, the line does not drop below $G_p = 1$ yet. When looking at the win percentages in Figure 8, the line looks very promising, almost reaching a win percentage of 100%. Since we do not have enough data points, we can assume the line would follow the same trend and be smaller than 1 at some point. Obviously, whenever the total win percentage reaches 100%, the line would immediately drop to 0, as it is not possible to have a win percentage above 100%.
This could also mean that the agent could reach a win percentage of 100%, while never dropping below $G_p = 1$.

5.3 The c parameter

For this next experiment, we will be using different values for the MCTS c parameter. This value determines the agent's exploration. A high value for c means higher exploration.
The agents will all have a total number of simulations of ∼2M. Fifty games are played for each data point.
The c parameter seems to spike at a value of one, as seen in Figure 11. The results show a decline in win percentages when increasing the value of the c parameter beyond one. For increasing the efficiency of the Monte Carlo Tree Search agent, we will therefor be using a c value of one.

5.4 Different board sizes & tile usage

By counting how many times pawns move to one of the four different tiles, we can evaluate the effectiveness of the tile. On top of that, we will be creating a heat map of the board to analyze the

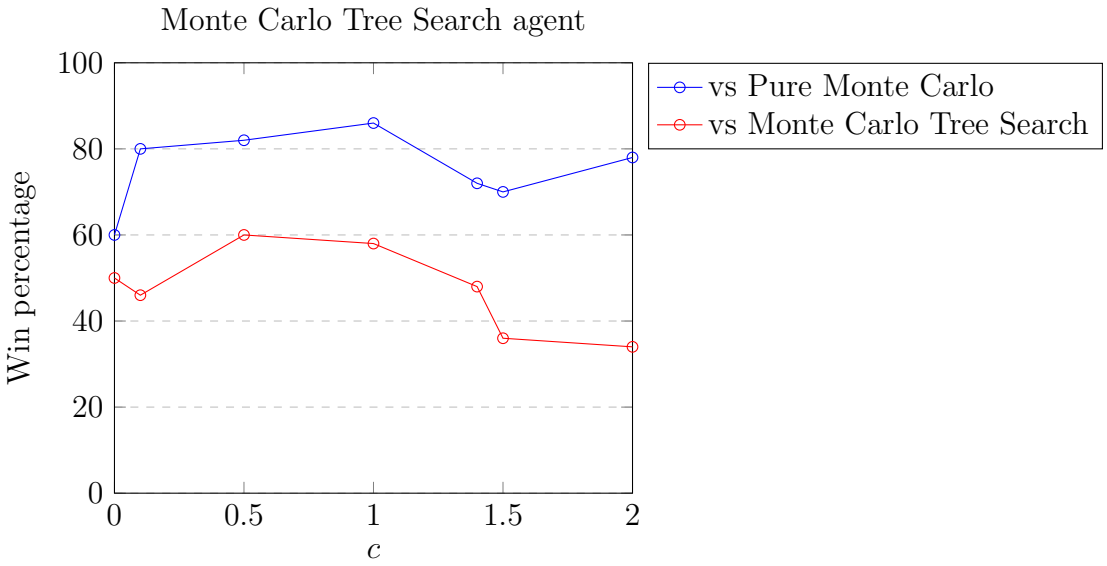

Figure 11: Win percentages with varying c values.

strategies the agents use. In order to create a heat map, we will have to make the tiles on the board static. The board will be created randomly and will not change for the duration of the experiment, which is fifty games. The Pure Monte Carlo agent will be playing as the white player, against the Monte Carlo Tree Search agent, with the optimized c parameter, as the black player. The tile usage is counted for both of the agents. Sections 5.1 and 5.2 had shown that the white and black players both have equal chance of winning.
In Table 2 we can see distribution of the usage of the different tiles. The counts of tile usage are the sum of all fifty games. The score does not reflect the actual performance of the agents correctly, because the agents have not the same number of allocated resources, for example: for the 12×12 board, the win percentage for both agents is 50%. However, when comparing the number of simulations done, the Pure Monte Carlo had double the number of simulations than the Monte Carlo Tree Search agent.
For this experiment we are more interested in the strategies played by the two Monte Carlo agents. As seen in the table, all four tiles are played relatively equal. Even though the differences in usage is very slim, both agents prefer to play king and rook pawns, over bishop and knight pawns.

5.5 Heatmaps

In the case of Katarenga, each new game randomizes the tiles on the board. Randomizing the board makes every new game unique and every new game requires different strategies. As such, creating a heatmap does not contribute much. This would only result in many different strategies crammed into one heatmap. So we use a fixed board.

The board

For experimenting purposes, we let the Monte Carlo Tree Search and the Pure Monte Carlo play fifty games against each other, with a fixed board. This fixed board is generated randomly and

Tile usages						
AI Agents	Board	King	Rook	Bishop	Knight	W/L/D
Pure MC vs MCTS	4×4	64	40	49	38	10/40/0
Pure MC vs MCTS	6×6	138	149	169	139	8/42/0
Pure MC vs MCTS	8×8	380	356	313	286	15/35/0
Pure MC vs MCTS	10×10	524	619	616	640	14/36/0
Pure MC vs MCTS	12×12	1066	1078	960	947	24/26/0
MCTS vs Pure MC	4×4	54	52	51	55	40/10/0
MCTS vs Pure MC	6×6	153	191	132	124	42/8/0
MCTS vs Pure MC	8×8	338	327	340	335	35/15/0
MCTS vs Pure MC	10×10	616	649	587	555	36/14/0
MCTS vs Pure MC	12×12	1019	1087	1018	946	26/24/0

Table 2: Tile usage count on different board sizes.

will remain the same for all fifty games. The fixed board is show in Figure 12 below. The starting position of the pawns are not included in the heatmap, only the moves played by the agents.

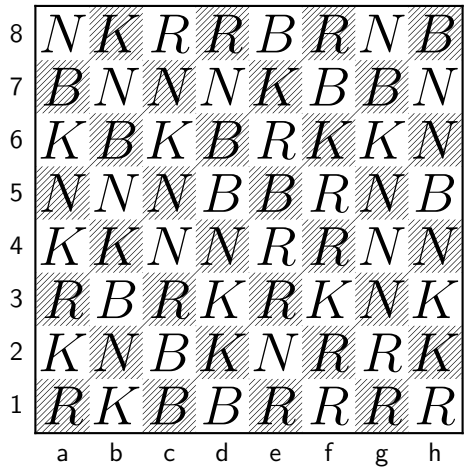

Figure 12: The fixed board used for the heatmaps.

The two agents as the white player

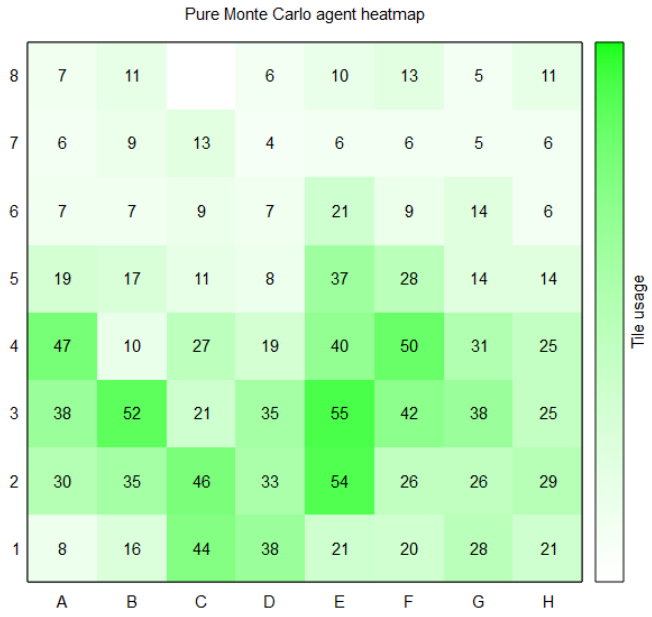

Figure 13: Heatmap from fifty games, played by the Pure Monte Carlo agent as white.

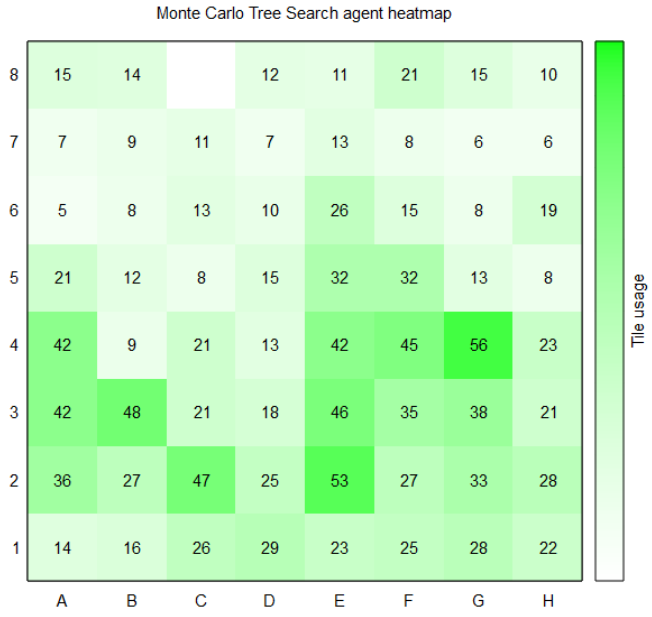

Figure 14: Heatmap from fifty games, played by the Monte Carlo Tree Search agent as white.

Both Figure 13 and Figure 14 show a similar heatmap. This means both agents had approximately the same strategy as far as the heatmap is concerned. Both agents formed a diagonal defense, or

a “V” shaped defense, which were connected with multiple pawns to strengthen the defense. For example the two diagonals: D1, C2 and B3 are bishop tiles and E3, E4 and F4 are rook tiles.
There is also a tile which has not been visited once by the agents, the C8 tile. The C8 tile is very hard to access, the only way to get to this tile is via the B8 king tile or D8 rook tile. From both the B8 and D8 tile the player can score a point, so there is no reason to move to the C8 tile.
When looking at the opponent side of the board, from row five to eight, the most used tiles are the rook, bishop and knight tiles. These tiles can move multiple tiles in a direction which allows the pawns to attack from a distance. The further the pawn is from the player’s side of the board, the easier it is for the opponent to capture that pawn, because it is harder to defend a pawn when there are more enemy pawns than allied pawns in the area. On the other hand, when looking at the player’s side of the board, the king tile is much more commonly used while the knight tile is used less.
So the rook and bishop tiles are excellent for attacking and defending, while the knight tile is better at attacking and the king tile is better at defending.

The two agents as the black player

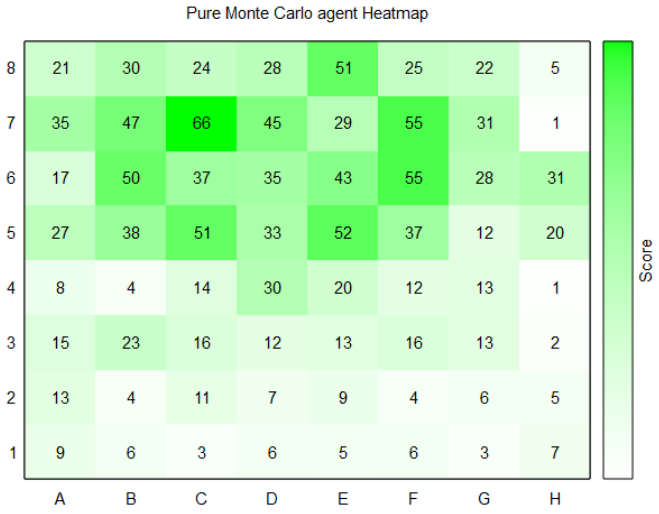

Figure 15: Heatmap from fifty games, played by the Pure Monte Carlo agent as black.

Again, the two Monte Carlo agents have a pretty similar heatmap, as seen in Figure 15 and Figure 16. The agents are focusing their pawns more to the middle of the board. The knight and bishop tiles are very frequently used. One thing to point out is that the top half of the board, the black player’s side, only has very few rook tiles, which might be the reason the agents are not utilizing the rook tile, like the white side agents.

5.6 Depth of play-outs

In order to create an efficient agent, the agent must try to win in as few turns as possible. For the previous experiments, winning in five turns and fifty turns received the same score, so it was

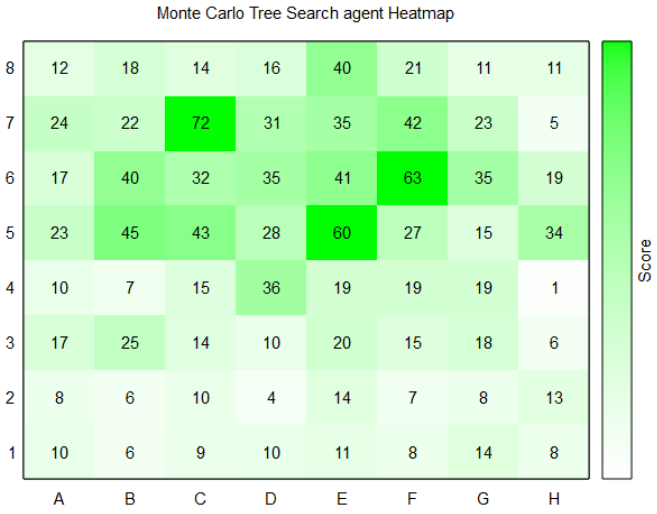

Figure 16: Heatmap from fifty games, played by the Monte Carlo Tree Search agent as black.

considered that both games were equally good. In order to save time, the agent must choose to win in fewer turns.
This is done by tracking the depth of the play-outs. After the play-out is done, the depth will be added as a penalty for the score. Which means that the bigger the depth of the play-out is, the bigger penalty the score will receive. The depth of the play-out is donated by D. The formulas for giving penalties are:

The Pure Monte Carlo agent

- Loss: $-100 - D$
- Draw: $-D$
- Win: $100 - D$

The Monte Carlo Tree Search agent

- UCT Score: $w_i/n_i + c\sqrt{\ln(N_i)/n_i} - (D/300)$

In this experiment, we let the two agents play against themselves. The first time without taking the depth of the play-outs into account and the second time with the depth of the play-outs included. Each match consists of one hundred games.
The results of including the depth of the play-outs is shown in Table 3. For the Pure Monte Carlo agent, the inclusion of the depth of the play-outs significantly increased the win percentage of the agent. By looking at some more statistics, the Pure Monte Carlo agent with the depth of play-outs included moved the knight tile the most and the king tile least (K:381, R:465, B:449 and N:526). The Pure Monte Carlo agent without the inclusion of the depth of the play-outs had a very balanced distribution of the tile usage (K:466, R:466, B:457, N:436).

Depth of play-outs		
AI Agents	Include depth of play-outs	Score
Pure MC vs Pure MC	No	48/52/0
Pure MC vs Pure MC	Yes	71/29/0
MCTS vs MCTS	No	49/51/0
MCTS vs MCTS	Yes	19/81/0

Table 3: The two Monte Carlo agents with and without depth of play-outs.

On the other hand, the Monte Carlo Tree Search agent with the inclusion of the depth of the play-outs, significantly decreases the win percentage of the white playing agent. Looking at further statistics, both Monte Carlo Tree Search agents have very similar tile usages. The addition of the depth of the play-outs to the Monte Carlo Tree Search agent, made the agent lose performance.
In the case of the Monte Carlo Tree Search agent, the depth of play-outs punishes exploitation. Exploitation wants to build further upon nodes which have a high win ratio, while each time increasing the node's depth. So nodes with high win ratio get punished more than nodes with smaller win ratio's. This also increases the exploration of the game tree, as nodes who have been visited a low number of times have a smaller depth and thus receives a smaller penalty. So the depth of the play-outs conflict with the exploitation and exploration part of the UCT score.

5.7 Beneficial boards

Finally, we will examine if there are beneficial boards. Beneficial board are board layouts which give one of the players an advantage over the other player. We will conduct two different experiments on an 8×8 board.

First experiment

The first experiment is to generate a board layout, and keep this layout for fifty games. We let two equal agents play on this board and see if one of the players wins more often. This will be done on one not-fixed random board and three different fixed board layouts.
The experiment begins with a playing fifty games on fifty different layouts of the board, to compare the results with. This is followed by selecting three randomly chosen layouts of the board and letting the agents play fifty games on each fixed board.
We can see some differences in the scores of the game, as seen in Table 4, however these results alone are not enough to conclude if beneficial boards exists. The samples are not big enough and it is still based on random sampling, so the results could also be the result of chance.

Beneficial layout first experiment		
AI Agents	Randomized board	Score
Pure MC vs Pure MC	Yes	25/25/0
Pure MC vs Pure MC	No	29/21/0
Pure MC vs Pure MC	No	25/25/0
Pure MC vs Pure MC	No	23/27/0

Table 4: Fifty games on the same board.

Second experiment

For the second experiment, we will split the board in half, one being the white player's side of the board and the other the black player's side of the board.
The black player's side of the board will remain like the usual, being randomized every game. The white player's side of the board, on the other hand, will consist of only one type of tile. We will start by only king tiles, followed by the rook, bishop and finally the knight tile. Again, both agents will be the same, so they should be equally smart.

Beneficial board second experiment		
AI Agents	White player's side of the board	Score
Pure MC vs Pure MC	Random	25/25/0
Pure MC vs Pure MC	K	32/18/0
Pure MC vs Pure MC	R	24/26/0
Pure MC vs Pure MC	B	24/26/0
Pure MC vs Pure MC	N	27/23/0

Table 5: Fifty games with different layouts for each half of the board.

The final results are shown in Table 5. The only board layout which stands out is the white player's side of the board filled with only king tiles. We have seen in Section 5.5 that the king tile is good for defending. When filling the player's own side of the board with good defending pawns, it is very hard for the opponent to capture points or pawns. Although the average turns for a single game doubled, from approximately forty to approximately one hundred, we can see that beneficial boards do exist.
As for the other tiles, the results remained balanced at about 50%.

5.8 Final test

To conclude, the final experiment is to let the two agents play against each other for one last time. This time, both agents have been optimized with the knowledge gained from previous experiments. The first hundred games will be played with the Pure Monte Carlo agent as the white player, against the Monte Carlo Tree Search agents as the black player. The next hundred games the two agents switch from sides.
The Pure Monte Carlo agent has been optimized with the depth of play-outs included and the Monte Carlo Tree Search agent has been optimized with a new c value.

Optimized Monte Carlo agents final test			
AI Agents	Simulations player White	Simulations player Black	Score
Pure MC vs MCTS	2.4M	3.0M	50/50/0
MCTS vs Pure MC	2.9M	3.0M	55/45/0

Table 6: The Monte Carlo agents playing against each other, as player White and player Black.

In Table 6 the results of the final experiment is shown. We see that the Pure Monte Carlo agent has improved quite well and is able to even out the score when playing with as the white player. The agents seem the be approximately equally smart with the changes, with the Pure Monte Carlo agents as the most improved agent.

6 Conclusion and further research

We examined the Pure Monte Carlo and the Monte Carlo Tree Search agents for the board game Katarenga. We quickly found out that both agents perform significantly better than the random agent, as both agents have never lost a single game against the random agent.
The first few experiments showed that the Monte Carlo Tree Search agent performed better than the Pure Monte Carlo agent. We experimented with different numbers of simulations done and played games on many different board sizes, in which the Monte Carlo Tree Search agent won more often.
By analyzing the results for both agents, the Monte Carlo Tree Search agent barely had improvements to make. The algorithm is already efficient and the only optimization we could find is to change the c parameter value from $\sqrt{2}$ to 1.
The Pure Monte Carlo agent, on the other hand, has improved significantly by changing the score values and adding the depth of the play-outs as an extra factor for the score. These changes to the Pure Monte Carlo agent made it even catch up with the Monte Carlo Tree Search agent. Playing one hundred games twice, first as the white player and secondly as the black player, the outcome was fairly similar. The agents performed quite equally.

In the future, we would like to add a secondary score evaluation system to both the agents. This system would evaluate every game state and give a score based on a couple of simple parameters.

Examples for some parameters are: the difference in white and black pawns, if it is possible to score a point, if the player controls the center of the board and whether pawns are under attack or defended.

References

[Arn21] Robert Arntzenius. Agents for the strategy game Onitama. Bachelor's thesis, Leiden University, 2021.

[BPW+12] Cameron Browne, Edward Jack Powley, Daniel Whitehouse, Simon M. Lucas, Peter I. Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez Liebana, Spyridon Samothrakis, and Simon Colton. A survey of Monte Carlo Tree Search methods. *IEEE Trans. Comput. Intell. AI Games*, 4(1):1–43, 2012.

[Cou06] Rémi Coulom. Efficient selectivity and backup operators in Monte-Carlo Tree Search. In H. Jaap van den Herik, Paolo Ciancarini, and H. H. L. M. Donkers, editors, *Computers and Games, 5th International Conference, CG 2006*, volume 4630 of *Lecture Notes in Computer Science*, pages 72–83. Springer, 2006.

[JKR17] Steven D. James, George Dimitri Konidaris, and Benjamin Rosman. An analysis of Monte Carlo Tree Search. In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence*, pages 3576–3582, 2017.

[KS06] Levente Kocsis and Csaba Szepesvári. In *17th European Conference on Machine Learning: ECML 2006*, volume 4212 of *Lecture Notes in Computer Science*, pages 282–293. Springer, 2006.

[Par] David Parlett. A chess-flavoured game for two players. https://www.parlettgames.uk/katarenga/. Accessed July 4, 2021.

[Sat] Sato, Shimpei. Onitama — An elegant and simple game of martial tactics. https://www.arcanewonders.com/game/onitama/. Accessed July 4, 2021.

[Sch07] Jonathan Schaeffer. Game over: Black to play and draw in checkers. *J. Int. Comput. Games Assoc.*, 30(4):187–197, 2007.

[SHS+18] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, and Demis Hassabis. A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play. *Science*, 362(6419):1140–1144, 2018.

[Wes] D.M. West. What is artificial intelligence? https://www.brookings.edu/research/what-is-artificial-intelligence/. Accessed July 4, 2021.