Universiteit Leiden
The Netherlands

Opleiding Informatica

Agents for the Strategy Game Onitama

Robert Arntzenius

Supervisors:
dr. W.A. Kosters & dr. J.M. de Graaf

BACHELOR THESIS

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

July 14, 2021

Abstract

Onitama is a two player abstract strategy game based on Japanese martial arts. This thesis will discuss the implementation and behaviour of two Monte Carlo AI-strategies to Onitama: Pure Monte Carlo and Monte Carlo Tree Search. The implementation of MCTS is optimised by the application of a pruning algorithm, which enhances both its general and runtime performance. MCTS performs significantly better than MC, with the gap in performance increasing with higher computation times. From looking at the behaviour of MCTS we notice patterns that are independent of starting configuration, which can be applied as strategies.

Contents

1 Introduction

Onitama is a two-player abstract strategy game with perfect information. It is produced in 2014 by Arcane Wonders [Wona] and is thematically inspired by Japanese martial arts. Because of its perfect information and simple rule set, the game is perfect for the application of strategic agents. In this thesis, the strategic agents *Pure Monte Carlo* and *Monte Carlo Tree Search* will be applied to Onitama.

The game is played on a 5×5 board. At the start of the game both players have one master pawn and four student pawns. The players move their pawns by using movement cards, that cycle around the board between both players. While the game has perfect information, the cards function as a random element as they get distributed randomly at the start of the game. The goal of the game is to either get your master pawn across the game board to your opponent's so-called temple arch or to capture your opponent's master pawn.

Figure 1 shows an example of a game state with both players (shown in red and blue respectively) having only their master pawn (represented as a diamond) and one student pawn (represented as a circle) left. In this example we see the red player using a movement card to move and capture blue's master pawn, which wins them the game.

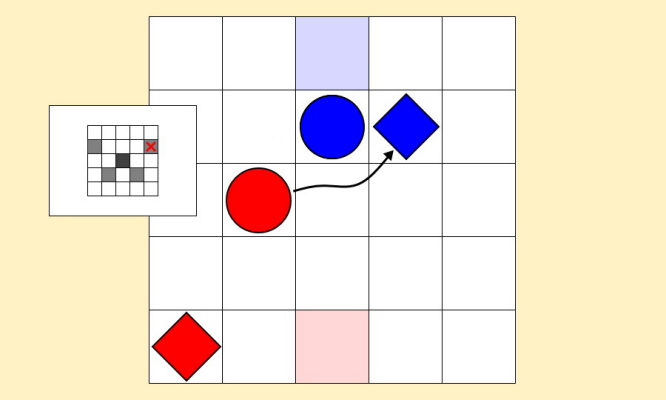


Figure 1: Example of a game state and a winning move.

The strategic agents that will be discussed use the Pure Monte Carlo strategy (MC) and the Monte Carlo Tree Search strategy (MCTS; see [Mag15, BPW+12]). These strategies are both heuristic algorithms often applied to similar strategy games. MCTS is most notably applied to the game Go with programs like AlphaGo [SHM+16] and AlphaGoZero [HPA+18]. Both algorithms play a large number of games randomly from specific possible board states and then compare the number of times the player has won these games. The strategies are often used to approximate complicated deterministic problems in a relatively easy and computationally cheap way. Another aspect that makes the strategies appealing is that they can be applied to many different types of problems.

The goal of this thesis is to apply and compare the different strategic agents and to find patterns in their behaviour to find strategies. Notably, we will see that the MCTS agent's optimal exploration parameter differs strongly from the theoretical optimum and that MCTS performs similar behaviour regardless of the starting configuration.

1.1 Thesis overview

The thesis will begin with an explanation of the game's rule set in Section 2. Section 3 contains a number of works related to this thesis and the AI-strategies applied. In Section 4 the different strategic agents will be explained in detail. Section 5 contains the experiments performed and their results. Section 6 will follow this up with an analysis of the behaviour of the optimised agents. Finally, Section 7 concludes the research and discusses potential further research.

This bachelor thesis was written at the Leiden Institute of Advanced Computer Science (LIACS) and supervised by Walter Kosters and Jeannette de Graaf.

2 The Rules of Onitama

This section discusses the game ONITAMA and its rules. ONITAMA is played on a 5×5 board by two players. Each player begins the game with 5 pawns of a single colour (red or blue), consisting of 4 students and 1 master; see Figure 2. The base game has 16 movement cards. Each card is named after an animal with each describing up to four movement options, as seen in Figure 3. The colour used for a card indicates whether the card has symmetrical (green), primarily left (blue) or primarily right (red) moves. Rule explanation is largely based on the official rule book of the game [Wonb].

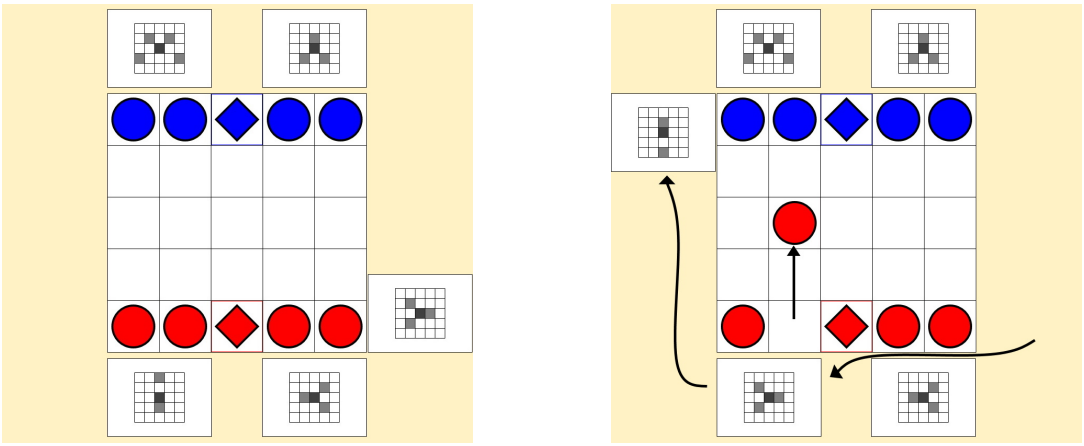
Figure 2: Components.

Figure 3: Cards.

Set-up Both players place their master pawn on the temple arch of their respective colour (being the central square on their side of the board) and their 4 student pawns on the 2 squares on either side of the master. The move cards are shuffled and 2 cards are given to each player—these become the players' starting hands. Finally a last card is turned over. The colour of the stamp in the bottom-right corner of this card determines the starting player. After determining the starting player, the card is placed to the right side of the board respective to and facing that player. The rest of the cards can be put away. The resulting board should look like Figure 4. In this case the starting player is red.

Figure 4: Board setup.

Movement Every turn a player gets to move a single piece according to one of the two cards in their hand. A move card shows a number of movement options described by the coloured squares' relative positions to the black square in the centre, as shown in Figure 5. Note that every move can be applied to all pawns, including the master and that movement cards have a specific orientation and can't be moved around. Movement across the board is not blocked by other pawns, but moves that would otherwise cause a pawn to move off the board or onto the same square as another one of the same colour are not allowed.



(a) Before (b) After

Figure 5: Movement example.

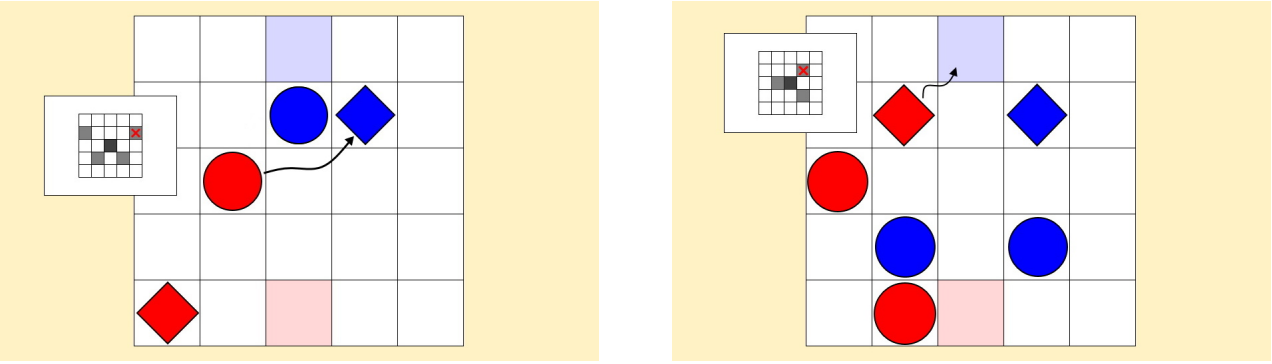
Capturing If a player's pawn moves onto a square that is occupied by an opponent's pawn, the opponent's pawn is captured and removed from the game. Movement through a square does not result in capturing a pawn.

Card exchange After moving a pawn, the player takes the move card used, and places it to the left side of the board facing the opponent. Finally, the player takes the card to their right and adds it to their hand, which ends the players turn.

Victory conditions There are two ways to win the game: the Way of the Stone and the Way of the Stream:

- To win in the *Way of the Stone*, a player has to capture the opponent's master pawn as shown in Figure 6a.
- To win in the *Way of the Stream*, a player has to move their own master pawn onto their opponent's temple arch square as shown in Figure 6b.

Note that is also possible to satisfy both victory conditions at once if the opponent's master pawn is captured by the player's own master pawn on the opponent's temple arch square, this is seen as a normal victory.



(a) Red player wins by Way of the Stone.

(b) Red player wins by Way of the Stream.

Figure 6: Victory conditions.

Special conditions. A player must execute a valid move every turn, even if they do not want to. However, it is possible—although not very likely—that a player has no legal moves in hand. In this case, the player exchanges a move card of their choosing as usual without moving any pawn and their turn ends.

3 Related Work

Since MCTS is an AI strategy commonly applied to strategy games, there have been many articles and papers written on similar strategic games. As mentioned in Section 1 one of these games is Go. As of December 2017 the current state-of-the-art Go AI is AlphaZero designed by Google's DeepMind Technologies [SHS+18]. This is a generalised variant of AlphaGo Zero [HPA+18] published in October 2017. This means that the AI can be applied to not only Go, but any strategy game. The paper shows us impressive results in both chess and shogi to show how the agent's capabilities surpass Go.

AlphaZero uses an advanced implementation of MCTS that uses a deep neural network instead of a handcrafted evaluation function and move-ordering heuristics.

> "AlphaZero replaces the handcrafted knowledge and domain-specific augmentations used in traditional game-playing programs with deep neural networks, a general-purpose reinforcement learning algorithm, and a general-purpose tree search algorithm."

This means that their advanced implementation replaces the selection formula that will be discussed in Section 4.3 with one determined by neural networks. This kind of implementation would not be as beneficial in our case, because of the randomised card distribution, which complicates the use of information from previous games.

Another game that MCTS has been applied to is the game KATARENGA. In the thesis of Tayfun Aygün [Tay21], he optimises his implementations of MC and MCTS and analyses the behaviour of the agents applied to this similar abstract strategy game. In his research, he optimises his implementations by finding the optimal exploration parameter and including depth in his scoring system.

4 Strategic Agents

In this section we will discuss the different AI strategies applied and the resulting strategic agents. All agents will be described from a theoretical perspective as well as a more detailed explanation of the specific implementation for ONITAMA [Arn21].

4.1 Random agent

The random player is simply an agent that bases all of its actions on the result of a random number generation. It finds all the possible moves from a given game state and randomly selects one of all possible options with all options having equal probability. This agent is not very interesting by itself; however it gives us everything we need to apply the other AI strategies.

4.2 Pure Monte Carlo agent

The Pure Monte Carlo strategy (MC) uses randomness to determine which from the player's possible moves has a statistical advantage over the others. It works as follows. The agent first identifies all possible moves. For every possible move it creates a copy of the board and plays that move on the copy. From that game state, it can use the implemented random player to play randomly for both players until a player wins the game; this process is referred to as a playout. For each of the possible moves it performs the same number of playouts and computes a score value based on the number of times it won its playouts.

Whenever a new score has been computed, it is compared with the maximum score that has been found up to that point. If a new maximum is found, the move is assigned to be the new best move. If multiple moves result in the same score, earlier moves will be considered better. Although the game normally does not result in a tie, the MC agent does cut off playouts which take too many moves to finish, this is regulated by a maximum length variable. If a playout is cut off, instead of counting it as a loss or win, a different constant value will be added to the total score of the move. To discourage MC from causing ties through infinite loops, this constant value has been set to a small negative value (in this particular implementation it has been set to -0.05).

4.3 Monte Carlo Tree Search

The final strategic agent we discuss is the Monte Carlo Tree Search agent (MCTS; see [Mag15, BPW+12]). It is an altered strategy of MC. The difference being that MCTS tries to favour moves that have shown to be potentially good by spending more computation time on these moves. It also differs from MC in the way it stores its information; where MC does not store anything but the best move and the highest number of wins, MCTS stores a complete tree structure with every node being a certain game state after a specific series of moves and with the current state as the root. MCTS uses this structure to not only look at its immediate moves, but look a number of moves ahead.

The method can be divided into four phases: *selection*, *expansion*, *simulation* and *back-propagation*. These four phases are visualised in Figure 7.

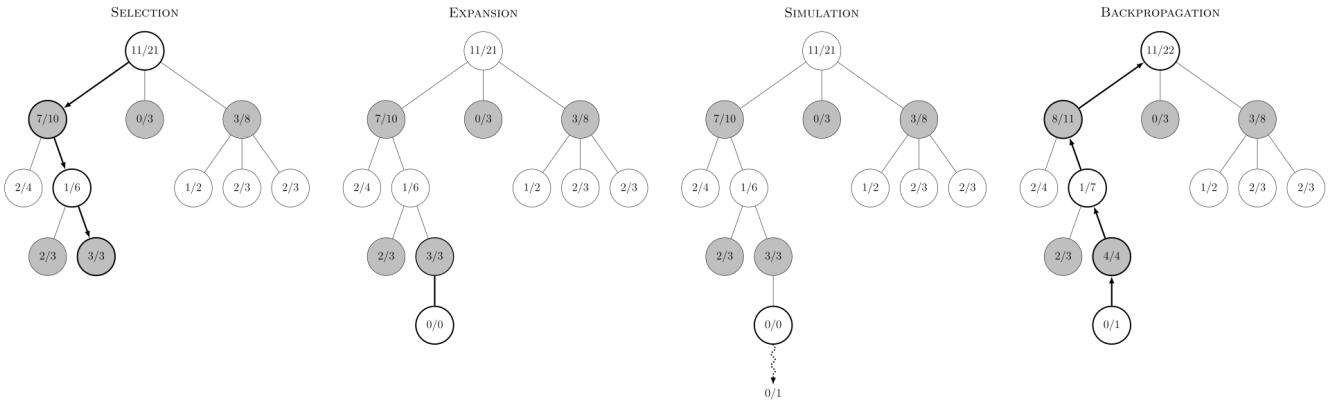


Figure 7: Visualisation of MCTS (image taken from [Wik]).

Selection During the selection phase MCTS performs a tree walk from the root to a leaf. Starting at the root of the tree, a child node is selected according to the selection formula:

$$\frac{w_i}{n_i} + c\sqrt{\frac{\ln N_i}{n_i}} \tag{1}$$

Here:

w_i denotes the number of wins for the node considered after the i-th move,

n_i the number of simulations for the node considered after the i-th move,

N_i the total number of simulations after the i-th move run by the parent node, and

c the exploration parameter—theoretically equal to $\sqrt{2}$.

For all child nodes of the node considered, a value will be computed according to this formula. The child with the highest value is selected and will be considered next. The selection phase ends as soon as the considered node is a leaf node.

Expansion After the selection phase, MCTS continues with the expansion phase from a leaf. If the leaf represents a finished game-state, MCTS skips to the back-propagation step. Otherwise a new child node is created from the leaf, this child node represents the board state after a specific move is played from the board state of the original leaf.

Simulation The simulation phase continues by simulating one or more random playouts from the newly created leaf similar to MC.

Back-propagation Finally in the back-propagation phase, the information in the nodes from the new leaf up to the root gets updated. Note that the wins and losses are updated according to the active player in each node.

MCTS continuously cycles through these phases according to the number of playouts, we refer to one cycle as a node cycle. After all playouts have been performed, the move with the highest amount of playouts will be the resulting move. Note that the highest amount of playouts does not directly imply the highest number of playouts won.

4.3.1 Pruning

MCTS can be optimised in a number of ways, some to increase accuracy, others to decrease computation time. One additional implementation to optimise accuracy is option pruning. After a large portion of computation time is spent, a pruning function is called to remove or prune all subtrees that are unable to become the most visited subtree of the parent. This is done in order to prevent distributing valuable computation time to known sub-optimal options, which in turn causes the algorithm to distribute its computation time more efficiently.

Since the computational intensity should be minimal, we use the simple expression below to determine whether a subtree is able to become the most visited:

$$playouts + v_i < v_{max} \tag{2}$$

Here:

$playouts$ denotes the number of playouts left to distribute,

v_i the number of visits of the i-th child,

v_{max} the number of visits of the most visited sibling node.

The expression states that an option can be pruned if its theoretical maximum (the number of visits it would have if it would be the only node considered from that point) is lower than the current maximum.

Time optimisation In addition to pruning the tree, another simple optimisation to decrease computation time is to cut off MCTS if, immediately after pruning, there is only a single option subtree remaining. This prevents the algorithm from spending computation time on unnecessary playouts whenever there is a clear outcome.

Since the pruning algorithm can be performed in $\mathcal{O}(n)$ time, it is a relatively harmless optimisation in terms of runtime complexity, however it is not useful nor efficient enough to prune after each nodecycle. We can state that no options can be pruned before at least half of the playouts have been distributed. This is because Expression 2 with $playouts \geq \frac{1}{2} \cdot total$ and $v_{max} \leq \frac{1}{2} \cdot total$ is false regardless of the value of v_i.

A more efficient way of pruning can be achieved by finding an optimal pruning interval such that the time won from the optimisation minus the time loss of applying the pruning algorithm is maximised. Finding such an interval will be further discussed in Subsection 5.3.2.

5 Experiments

This section will discuss a number of experiments performed during the research. The first experiments revolve around the comparison of the implemented strategic agents. After that, the experiments explore the effectiveness of the applied optimisations with special regards to parameter tuning. Throughout all experiments the starting player is picked at random.

5.1 Agents vs random

In Figure 8 we see the agents' wins against the random player for a certain number of playouts of both Pure Monte Carlo and Monte Carlo Tree Search. It is immediately apparent that both MC and MCTS outperform the random player with MC having a higher win rate at lower numbers of playouts and MCTS winning more with number of playouts larger than 1,000. The most important thing we see in the graph is that the percentage of wins at 1000 and 10,000 playouts for both agents exceeds 99%.

We can also see that MCTS with a single playout has won less than half of the matches against the random agent, this seems strange but there are multiple reasons that this is the case. The first reason is that random games that are not won by either player are cut off and counted as a loss for MCTS and MC respectively; during these experiments, the maximum duration in number of turns was set to 100 which resulted in 251 ties. The other explanation is that MCTS—unlike MC—performs the move with the highest number of *playouts* and not the highest number of *wins*. This means that, since there is only a single playout, MCTS with only a single playout always plays the first move available.

Runtime comparison To compare the agents further we need to look at their runtime performances. Figure 9 shows the runtimes of both agents in seconds from 1000 matches against the random agent. The random opponent has a negligible impact on the runtime, especially at higher numbers of playouts. Here we see the runtime having linear growth throughout the whole graph (since both axes are in the same logarithmic scale). The growth is not perfectly linear because

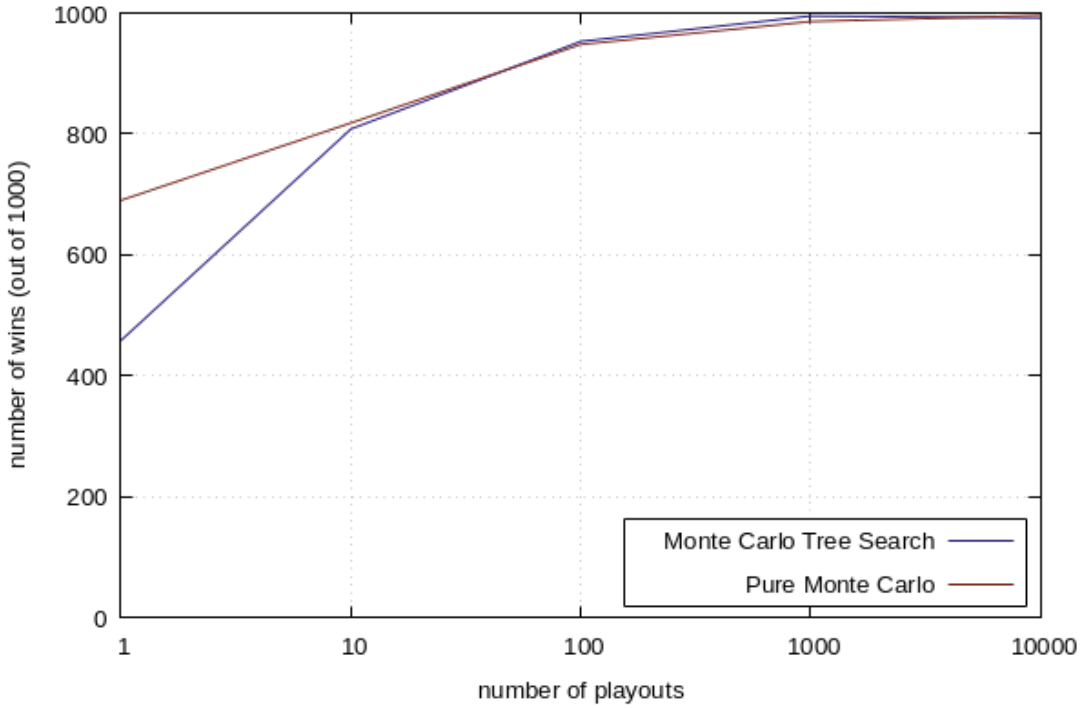


Figure 8: Performance of agents against random opponent. Horizontal axis is in logarithmic scale.

the average length of a game also decreases, with MCTS decreasing from 34.650 to 12.626 and MC from 24.981 to 9.336. Both agents have similar runtimes; with 100,000 playouts, MCTS takes about 1 hour and 18 minutes and MC about 1 hour and 27 minutes. MCTS is overall faster having runtimes of somewhere around 90% of MC's runtimes even with the average game duration being higher for MCTS.

5.2 MCTS vs MC

Figure 10 shows the number of games won by MCTS in games against MC plotted against the number of playouts allowed for both agents. Note that for both agents the number of wins is shown and that this does not have to add up to 1000 because of possible ties. Here we see that MCTS starts winning more than half of the games at 10 playouts. With numbers of playouts at around 100,000, MCTS wins over 90% of all matches. Since the number of wins of MCTS rises throughout the whole graph, we expect the rise to continue at higher numbers of playouts. The graph would most likely begin stagnating in the high 90 percents, since the MCTS algorithm allows it to make actual use of every playout.

Higher numbers of playouts allow MCTS to take more advantage of the tree structure and reach higher depths, while MC always stays in the same depth, regardless of the number of playouts. Since MC simulates from only the current depth, it will keep a pretty surface level understanding of a game state. MCTS however, can use all playouts to their full potential by performing them at higher depths and it keeps building on that information when it updates the data in the tree structure.

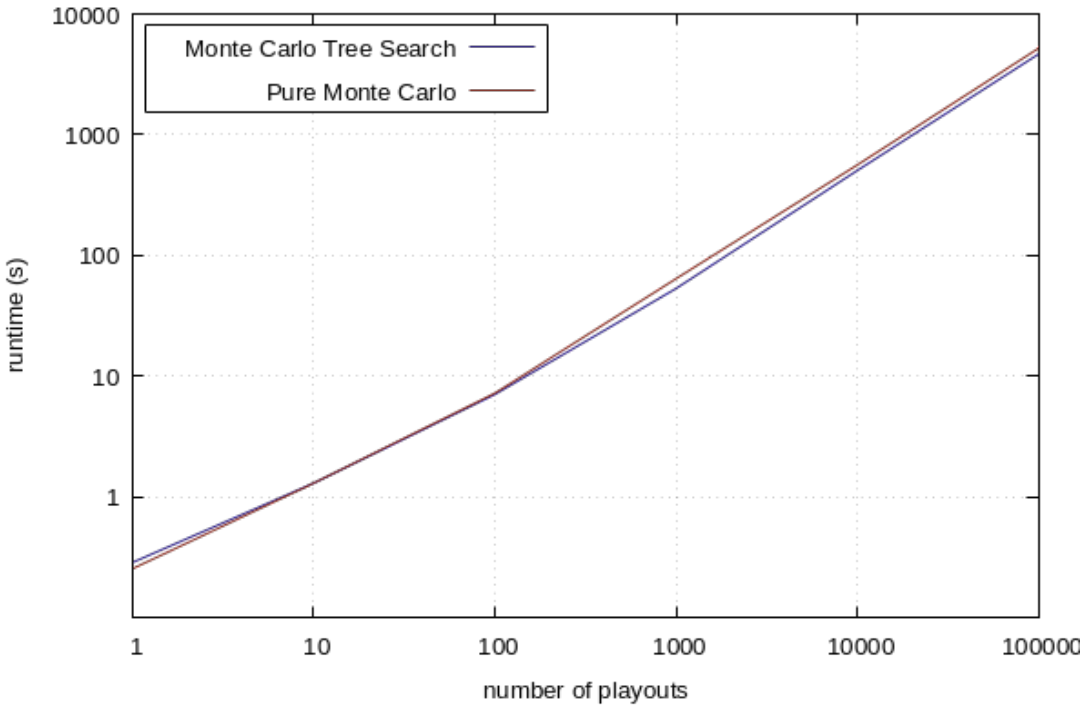


Figure 9: Runtime of agents against random opponent. Both axes are in logarithmic scale.

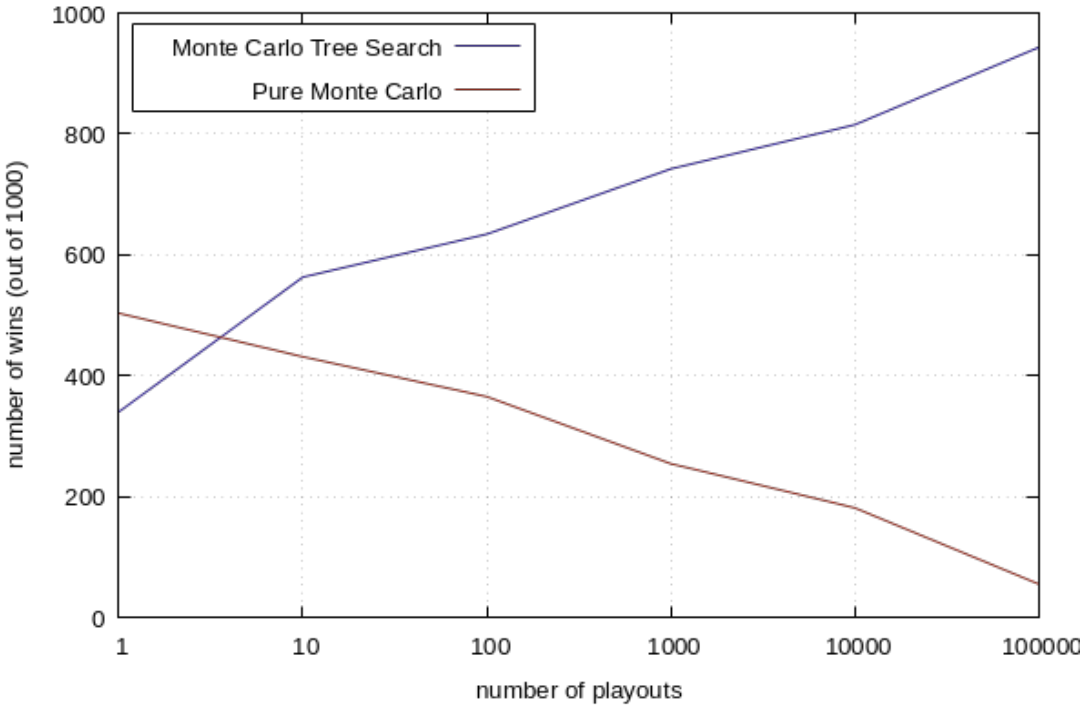


Figure 10: Performance MCTS agent against MC opponent. Horizontal axis is in logarithmic scale.

5.3 Optimisations

The following experiments revolve around how MCTS was optimised by parameter tweaking and the additional implementations that have been previously discussed in Subsection 4.3.1.

5.3.1 Exploration parameter

The most important constant in the MCTS algorithm is the exploration parameter. This is the constant value c in the selection formula 1 that controls where MCTS spends its computation time. The lower the parameter is, the more time it spends on moves that have already shown to be promising. The higher the exploration parameter is, the more equally the computation time is divided between moves. The goal of an optimal exploration parameter value is to find a balance between these, so the algorithm gives every option a fair chance, while spending a significant portion of its available computation on the best moves.

Figure 11 shows the number of wins of MCTS wile playing against MC at 1000 playouts plotted against the exploration parameter. We can see that although the theoretical optimal value for the exploration parameter is equal to $\sqrt{2}$, in our case the optimum lies significantly lower at around 0.4. This implies that MCTS has a greater accuracy when it spends a significant amount of computation time on moves that have impressive results. All other experiments are performed with this value for the exploration parameter c.

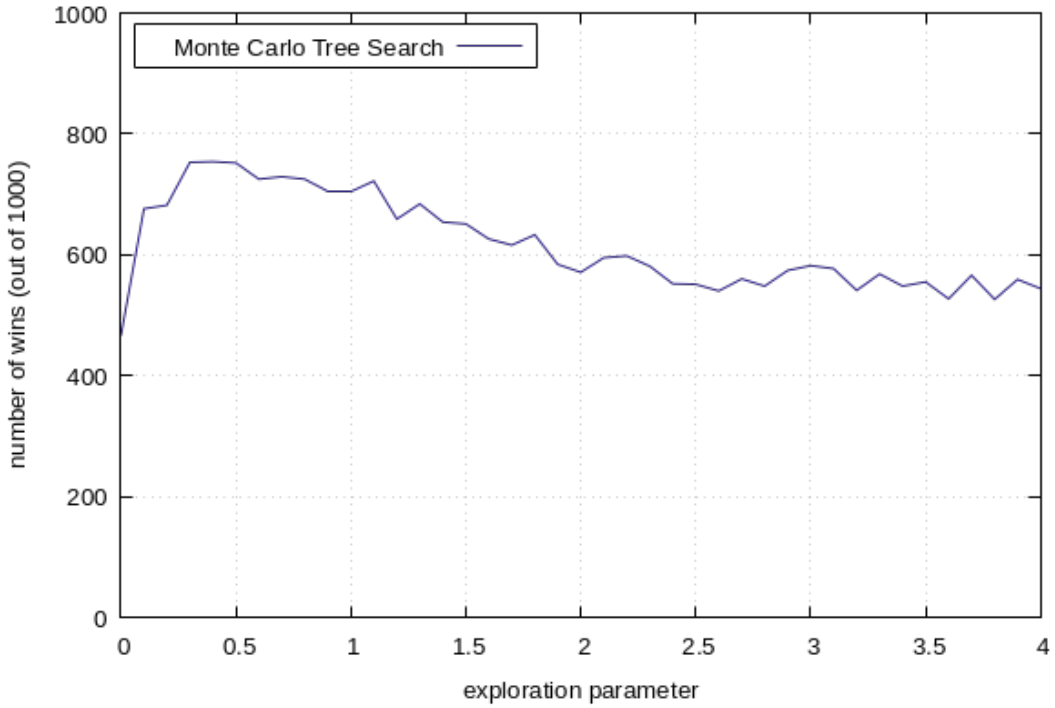


Figure 11: Performance of MCTS at variable exploration parameter values.

5.3.2 Option pruning

The implementation of pruning influences both how the algorithm distributes its computation time in the tree as well as the total runtime. The following experiments are about pruning and the impact it has on the accuracy and efficiency of MCTS.

Pruning optimally To figure out the optimal moments for MCTS to prune its subtrees, we need to know when pruning can be performed. In Figure 12 we see a distribution that shows when pruning was performed by MCTS with the pruning algorithm being called each node cycle. It covers the pruning data of 1000 games with 100,000 playouts between two MCTS agents. We can see that the distribution is relatively constant except for a peak around the interval $[0, 1000]$ and an enormous peak at the interval $[49000, 50000]$.

The latter of the two peaks is the first interval where pruning can occur and it occurs only when there is a large difference between the quality of the available moves. The other peak interval ($[0, 1000]$) is the last interval where pruning can occur. It occurs when there are more than one equally performing moves. An example is two winning moves. In such a case, the final move can only be pruned when there are very little playouts left. Since the distribution shows that pruning occurs at any point after half of the playouts have been distributed, it is important that pruning is done regularly throughout the second half of the playouts.

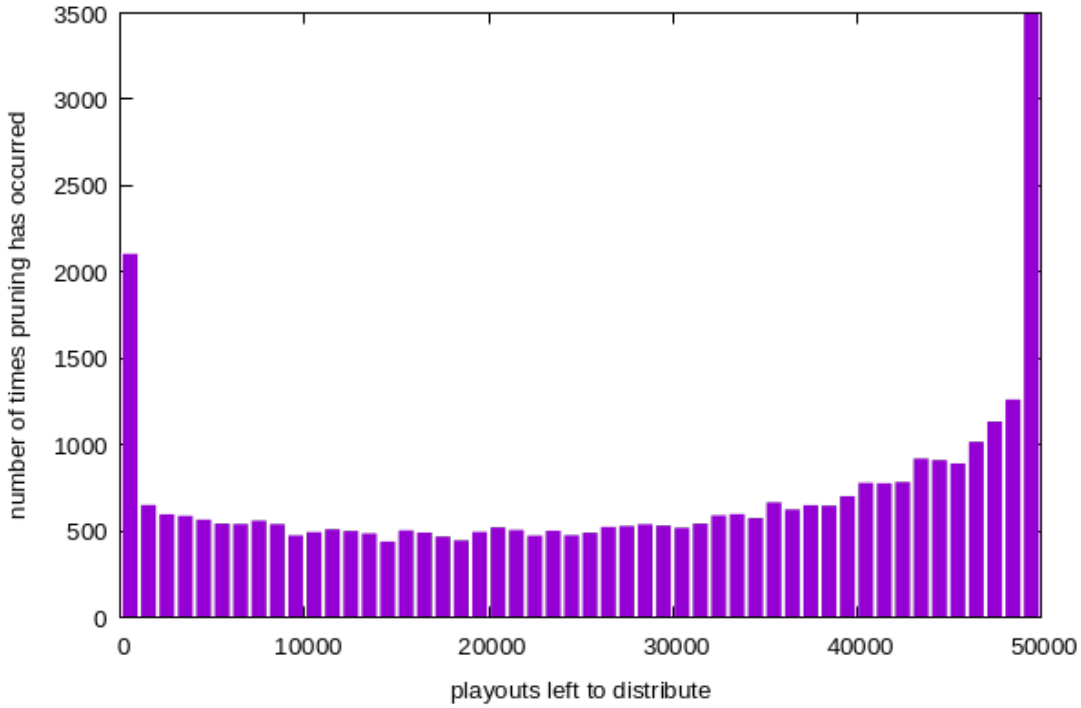


Figure 12: Pruning distribution over 1000 games of MCTS at 100,000 playouts.

Win-rate comparison Since pruning causes computation time to be distributed more efficiently, it could impact the accuracy of MCTS. Figure 13 shows the number of games won by MCTS with

pruning every cycle against MCTS without pruning plotted against the number of playouts allowed for each agent. We can see that MCTS with pruning active in general wins more than MCTS without pruning, however, the difference is not significant enough to conclude an actual increase in accuracy.

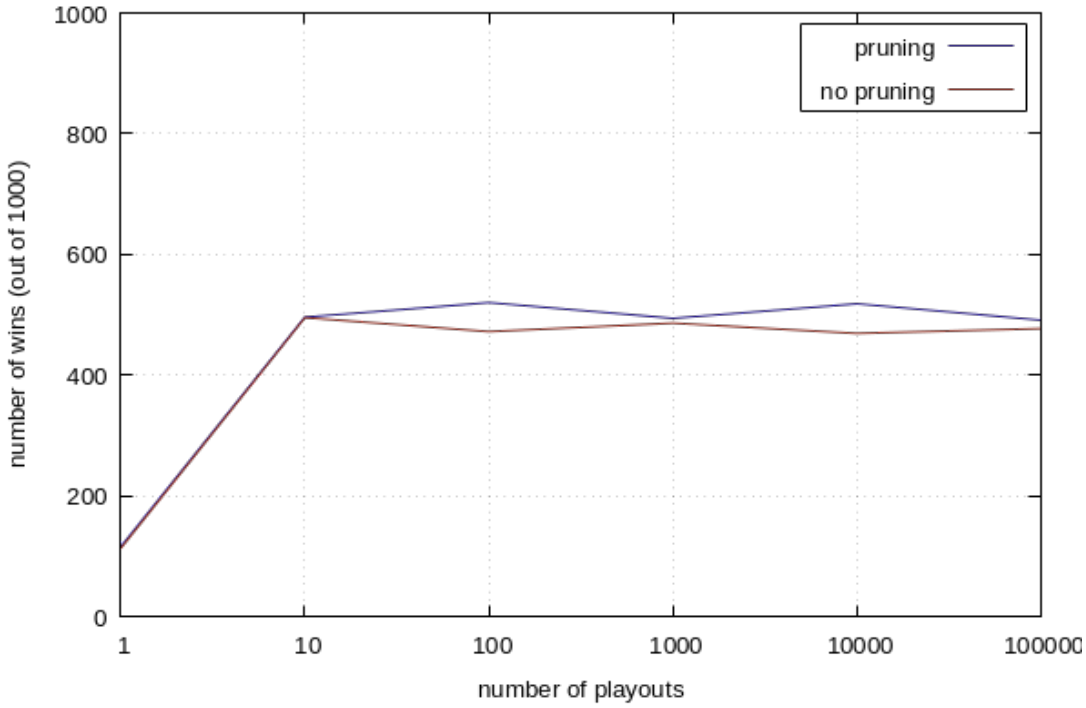


Figure 13: Performance of MCTS with pruning against MCTS without pruning. Horizontal axis is in logarithmic scale.

Cycle reduction As explained in Subsection 4.3.1, pruning can reduce the number of cycles/playouts by cutting the algorithm off early if there is only a single option left after pruning. Figure 14 shows the reduction in performed cycles at different pruning intervals. With a high number of playouts, pruning has a proportionally larger effect on reducing the number of cycles performed. At 100,000 playouts, the number of cycles has already decreased from 100,000 cycles on average to just over 70,000. This is a 30% reduction

The minimal pruning interval (1) means that after each cycle there will be pruned. This functions as a reference and performance measure for higher intervals. We can see that both the other pruning intervals (10 and 100) come very close to the minimal interval. Both start overlapping where the pruning interval is $^1/_{00}$ times the number of playouts. Since at these points the reduction is indistinguishable from the minimal interval, these higher intervals become more efficient since the pruning algorithm itself also increases the runtime complexity.

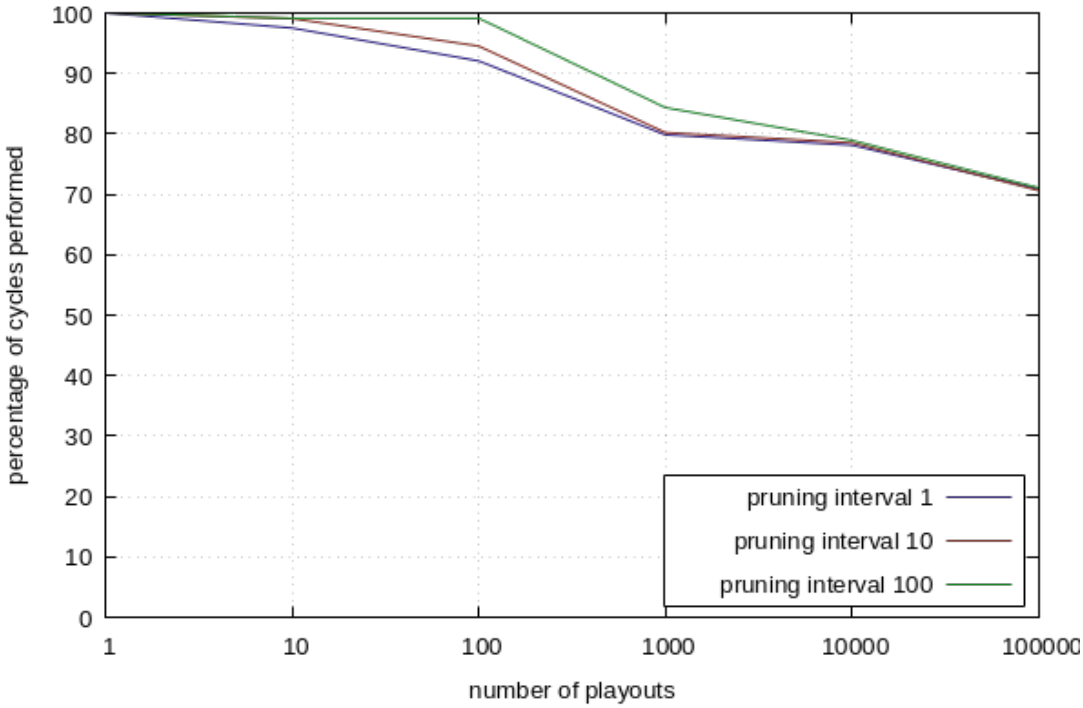


Figure 14: Cycle reduction. Horizontal axis is in logarithmic scale.

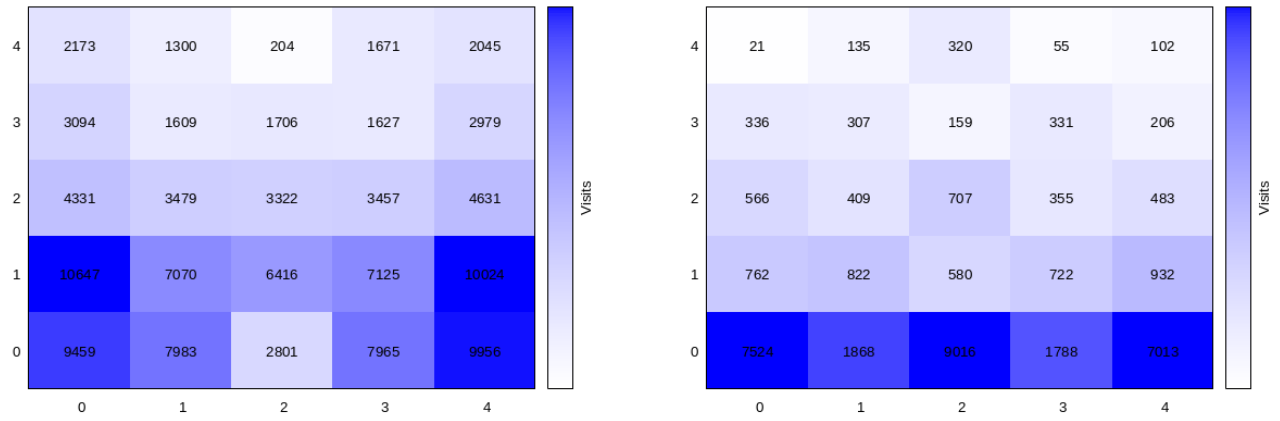
6 Behaviour analysis

This section will discuss strategies and patterns found in the behaviour of the fully optimised agents. This helps gain an understanding of how high level play of the game looks. The reason we focus on behaviour patterns and not concrete strategies is due to the game having a randomised set of cards; there are 16 unique cards to choose from of which 5 are randomly picked with two sets of two that are unordered, so in total $\binom{16}{2} \cdot \binom{14}{2} \cdot 12 = 131{,}040$ possibilities. This number of unique starting configurations complicates the search for concrete strategies. Since Section 5 showed that MCTS wins most games from MC, all following analyses are played between two MCTS agents, both with pruning active with intervals equal to $1/100$ times the number of playouts.

6.1 Heatmaps

To gain a better understanding of the strategies applied by MCTS, we can look at the behaviour differences between the different pawn types. In Figure 15 we see two heatmaps showing position data of the winning player of 1000 games with 100,000 playouts. Figure 15a shows the position data of the student pawns and Figure 15b that of the master pawns. The heatmaps show us the number of turns one of the respective pawns stood on all board positions throughout these 1000 games with darker colours representing more visits and lighter colours representing less visits.

[1]Note that Figure 15a has a color range of $[0, 10000]$ and Figure 15b one of $[0, 2000]$; this means that the master heat map has squares that are visited significantly more often than the color suggests.



(a) Students

(b) Master

Figure 15: Heatmaps that show the number of visits of students and master respectively[1]

6.1.1 Student heatmap

The first thing we notice in the student heatmap is that the students' starting positions are all visited many times, this is obviously because the students start here and it often takes a number of turns to get all of the students from their starting positions onto the center of the board. However, these positions are not the most visited positions for students; the two positions on either side of the board on the second rank both show a number of visits exceeding 10,000 and with that all positions on the first rank.

In general, we can see that the student pawns tend to surround the center of the board and when pushing further, they stay close to the edges of the board. The first two ranks are favorable being very defensive positions especially with the right movement cards in hand. Also students rarely find themselves on either of the temple arches. This is because if a student pawn is occupying the opponent's arch, the master pawn won't be able to move to the arch, so the student becomes an obstacle for the way of the stream. The reason that students avoid their own temple arch is mostly because the master pawn is already occupying this space, since it's an otherwise good defensive position.

6.1.2 Master heatmap

The master heatmap shows the immobility of the master pawn. The primary goal of the master pawn is to prevent itself from being captured. This is why the pawn wants to be protected at all times and occupy defensive positions. The most defensive positions are the corner squares on the first rank. These are occupied the most except for the starting position. The starting position is occupied the most because the master cannot move to the safer corner squares without the student pawns being moved first.

Although the master can stay in the corner safely through most of a match, it sometimes has to make itself vulnerable by moving into the center of the board. It does this mostly in a later stage of the game when some or all of the opponent's student pawns have been captured to try and win by way of the stream.

6.2 Game duration

One aspect of high level play that indicates the actual complexity of a game is the duration of matches between well performing agents. Figure 16 shows the distribution of the duration of games between two MCTS agents from 1000 games. The distribution has a positive skew with the average game taking 35.784 turns and median being 31 turns. Since the maximum game duration was set to 100 turns, the distribution has a large peak at 100 of games that took 100 turns or longer.

One interesting thing we notice in the graph is the minimal duration that has occurred. Three times, the agents have played a game that lasted five turns. This is curious since such a game comes very close to the theoretical shortest game; this would be a game that lasts three turns. A game of five turns would require very poor judgement from one of the agents or a very unbalanced start configuration.

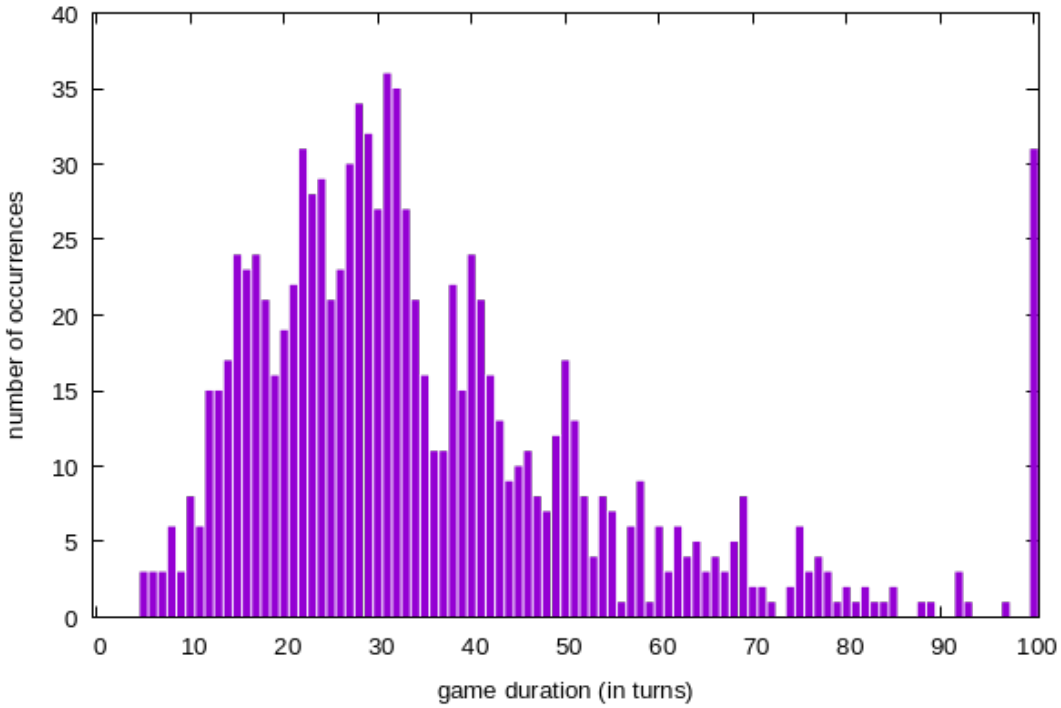


Figure 16: Distribution of game duration over 1000 games.

Since MCTS relies on randomness, it is vulnerable for unlucky random seeds. This is amplified by the c-parameter being lowered, since this increases the chance that a good move will be overshadowed by other seemingly good moves. This is also more likely since the start of the game has the most moves to be considered, which is one of the reasons why the start of the game is often considered as one of the weaknesses of all Monte Carlo strategies.

The median game duration of 31 turns makes the game nowhere near the complexity of Go which has a median of upwards of 200 moves. The average number of turns that a game lasts also increases as the agents get more computation time, meaning that the 31 turn median is most likely lower than the actual game duration of even higher level players.

6.3 Victory conditions

As mentioned in Section 2, the game has two victory conditions that can be satisfied with the first being the way of the stone (capturing the opponent's master pawn) and the other the way of the stream (moving your own master pawn onto the opponent's temple arch). Figure 17 shows the number of times each win condition has been met over 1000 games plotted against the number of playouts for both MCTS agents.

The graph shows us that regardless of the number of playouts, the way of the stone as a means of winning occurs more than the way of the stream. At lower numbers of playouts this seems logical, since an agent with random behaviour would have at most five pawns with which it can win by capturing the opponent's master pawn and only a single pawn (the master) that can win by moving onto the opponent's temple arch.

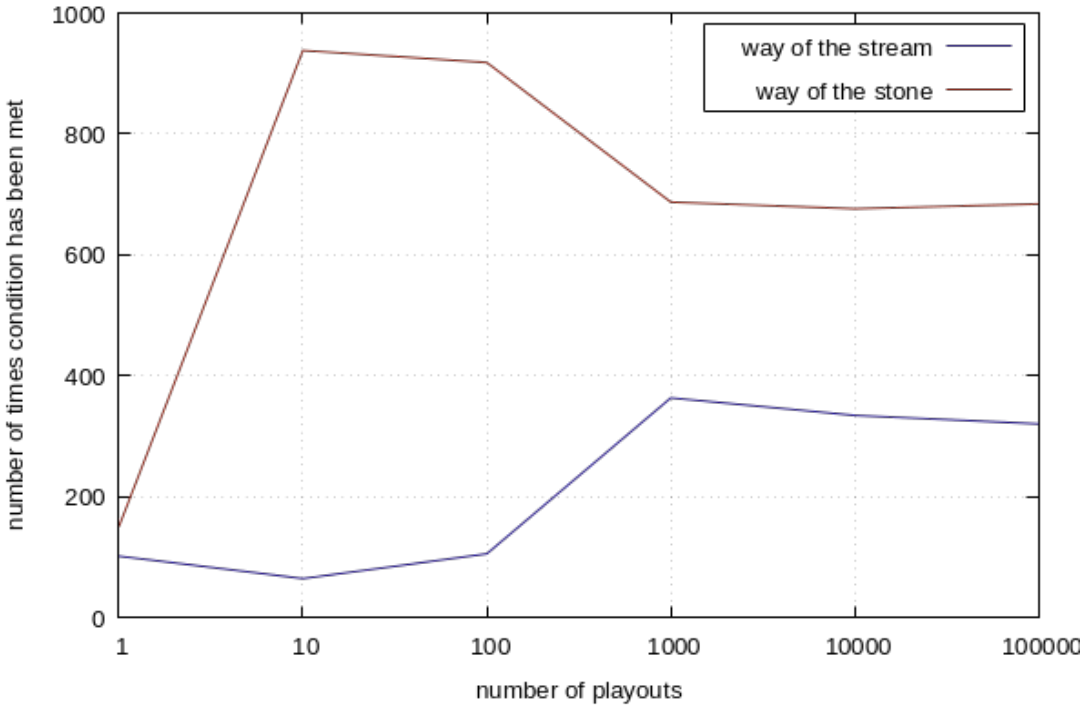


Figure 17: Victory conditions over 1000 games. Horizontal axis is in logarithmic scale.

We can also explain the curious behaviour with a single playout; the reason it's proportionally more likely to win by way of the stream is because it does not act randomly but it always chooses the first available move as explained in Subsection 5.1. This results in a lot of ties and since the implementation has the master pawn as the first pawn, it is very likely that most moves will be performed using the master pawns of both players, which makes both victory conditions almost equally likely to occur.

Lastly after the numbers of playouts have reached a reasonably high amount, the ratio seemingly stagnates around $320 : 684$ or $80 : 171 \approx 7 : 15$. This seems to indicate that although it is still more common to end games with the way of the stone, the way of the stream is a very valid strategy, especially when (almost) all of the student pawns have been captured.

6.4 Card usage

Arguably, the most interesting mechanic of ONITAMA is its movement mechanic, involving the movement cards. As mentioned in Section 2 there are 16 cards in the game all named after animals. Figure 18 shows the percentage of turns each of these cards were used over the scope of 1000 games. Each card was manually selected for 1000 games between two MCTS agents with 10,000 playouts each; all the other cards randomly distributed. Besides the 16 cards from the game, the figure also shows the data of two extreme cards with one having only a single movement option of staying in place and the other one having all movement options within a range of two spaces horizontal and vertical.

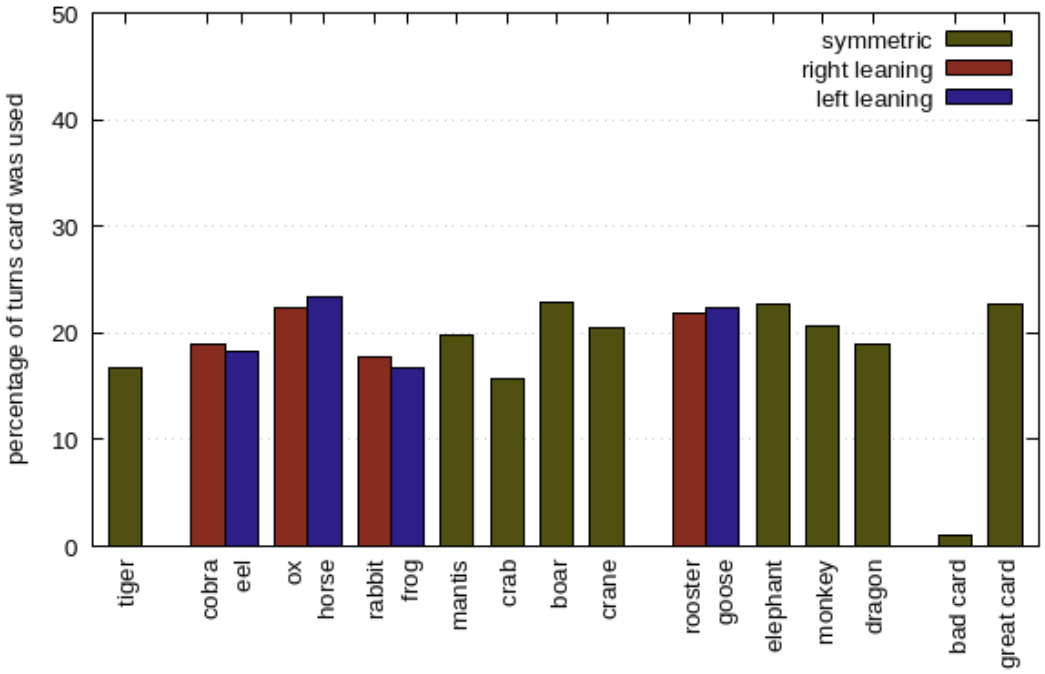


Figure 18: Usage of movement cards.

In the figure we see all opposite cards (cards that are the mirrored equivalent of each other) matched together, since the game as a whole is symmetrical with each non-symmetrical card having an opposite, these bars should be about equal; this is the case with the biggest difference being approximately 1.1%.

The cards in the figure are ordered by the number of moves that the card allows, with the card *tiger* being the only card with only two moves, the cards *cobra* through *crane* all having three moves and the cards *rooster* through *dragon* allowing four moves. Although the differences between cards with the same number of moves is noticeable, there does seem to be a trend of cards with a higher number of movement options generally being used more often than cards with lower numbers of moves.

6.4.1 More than two cards

Although the game is regularly played with two cards in hand, this number can be adjusted without changing any of the games rules. Figure 19 shows us how the average duration of a game over 1000 games between two MCTS agents with 10,000 playouts changes when we adjust the number of cards. We see the duration gradually decrease when more cards are given to each player. This could be because when the players get more cards, they consequently have more movement options so they can use their cards and moves more efficiently and win is less turns.

A different explanation could be that the decrease in game duration is because, as stated in Subsection 6.2, MCTS performs worse when it has more movement options. Which could cause the agents to make more mistakes and therefore have shorter games on average.

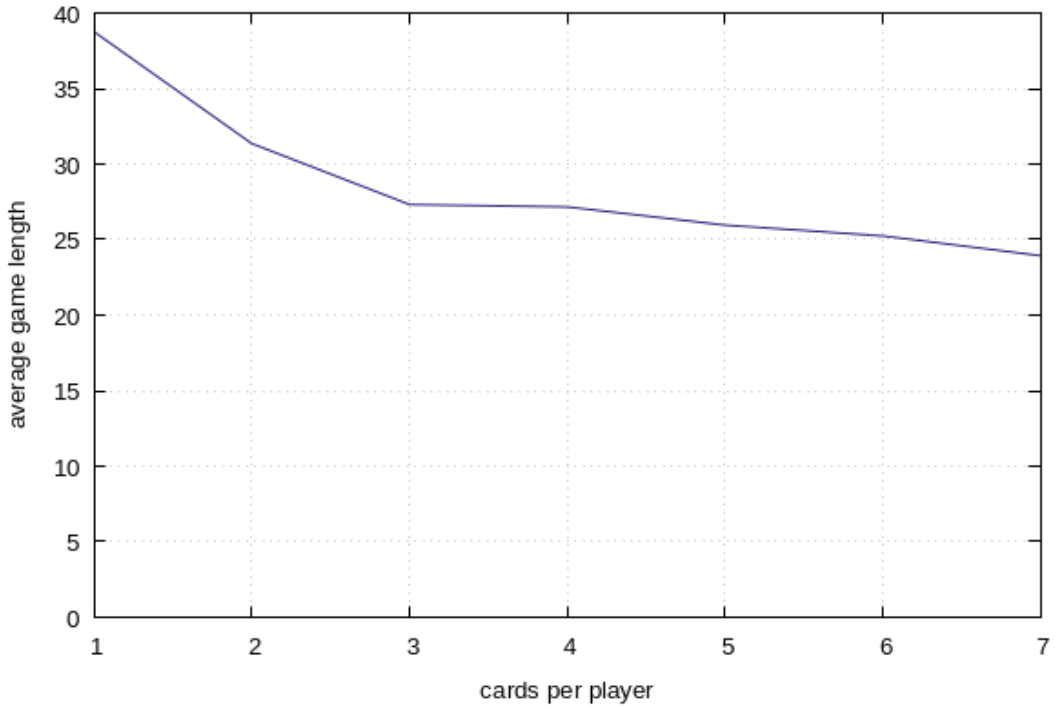


Figure 19: Average game duration at different numbers of cards.

6.4.2 Using cards to keep tension

Since the graph shows us how often a card was used, we might want to conclude that being used more implies that a card is useful. However, this conclusion is not completely true. Cards can be extremely useful for a player when held in hand. An example is shown in Figure 20. Here we see a situation where it's the red player's turn with the blue player coming close to a win by way of the stream. If allowed, the blue player can use its cards to get to the opponent's temple arch by first moving to the marked square (so moving diagonally to the bottom right twice).

The red player right now has one piece protecting the square: it's student on the bottom rank. This pawn protects this square when holding the *dragon* movement card (the left card of the red player).

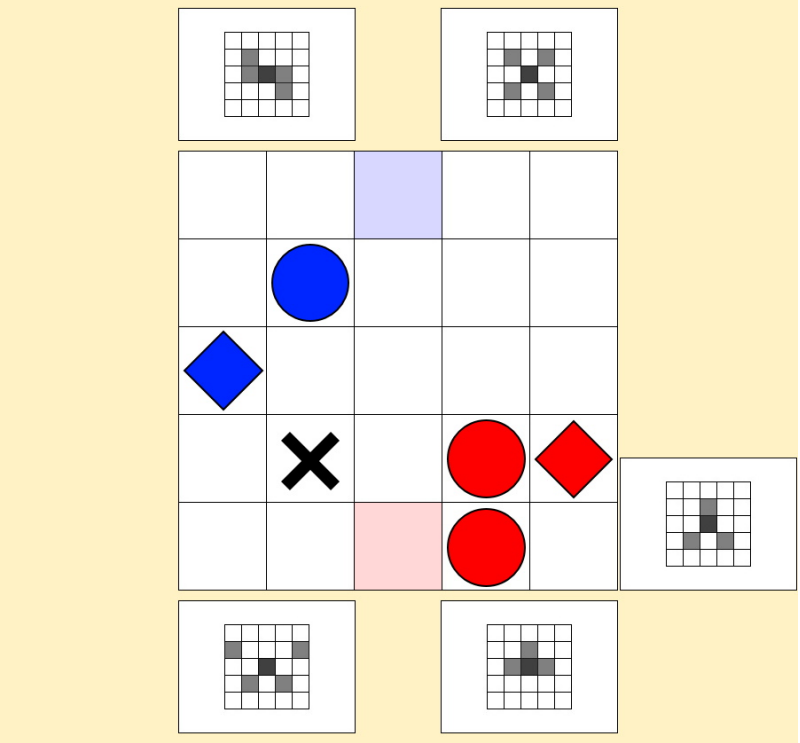


Figure 20: Example of situation where holding cards is useful.

The only way to prevent the blue player from winning the game is to keep the *dragon* and keeping its student on the same square [2]. The optimal move for red to make is to move their master pawn forward one square with the *boar* card, from there a win for red by way of the stream is inescapable.

7 Conclusions and Further Research

In this thesis we applied two Monte Carlo strategies, Pure Monte Carlo and Monte Carlo Tree Search, to the abstract strategy game Onitama. MCTS was optimised by applying a pruning algorithm and adjusting the exploration parameter to a more optimal value on the basis of its performances in the experiments. Finally, the behaviour of MCTS was analysed from which patterns and possible strategies were found and discussed.

The experiments of Section 5 have shown us that MCTS and MC have a significant performance difference when given the same number of playouts. While both are fit to win against the random opponent, MCTS can win over 90% of the matches with 100,000 playouts for both agents. This difference is due to the tree structure providing better insight at higher depths and it having an overall better distribution of its computation time.

When optimising MCTS, we found that the optimal exploitation parameter was significantly lower than the theoretical optimum. This makes the algorithm prioritise moves with promising results

[2]Note that although the marked square can be protected by the master pawn by moving to the most central square, this leaves the master pawn available for capture by the blue student pawn.

rather than moves that have not been given as much computation time—more exploitation and less exploration. Applying a pruning algorithm optimised runtime in terms of playouts by up to 30%, with little time loss from the algorithm itself. It also seemed to have increased performance; MCTS with pruning active won slightly over half of the games at multiple numbers of playouts. However, the performance boost is very minimal.

In Section 6, by using heatmaps, we found general strategies that are applied at high level play by the agents regardless of the starting configuration of the movement cards. This general behaviour takes place mostly during the opening phase of the game. In almost every game, both players try to move their student pawns to their second rank respectively and move their master pawn into either corner on their side of their side of the board. Although the master pawn is less mobile on a corner square, it can easily be defended by the student pawns on the second rank.

7.1 Further research

To improve and extend the research from this thesis, Onitama could be programmed more efficiently and be run on more powerful hardware to be able to simulate even higher numbers of playouts. MCTS could also be further optimised in a similar manner as Tayfun Aygün has done in his research [Tay21], where he included depth in the scoring system, to make playouts performed at lower depths more valuable than those at higher depths. While this thesis focused on Monte Carlo agents, further research could also extend this by implementing a Negamax agent with alpha-beta pruning. This would be a great agent to compare to MCTS, but it would require a heuristic function to evaluate game states.

References

[Arn21] Robert Arntzenius. Github repository Onitama. https://github.com/robertarntzenius/Onitama, 2021. Last accessed: 07-07-2021.

[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.

[HPA+18] Sean D. Holcomb, William K. Porter, Shaun V. Ault, Guifen Mao, and Jin Wang. Overview on deepmind and its alphago zero ai. In *Proceedings of the 2018 International Conference on Big Data and Education*, ICBDE '18, page 67–71, New York, NY, USA, 2018. Association for Computing Machinery.

[Mag15] Max Magnuson. Monte Carlo Tree Search and its applications. *Horizons: University of Minnesota, Morris Undergraduate Journal*, Vol. 2(Iss. 2, Article 4), 2015. http://digitalcommons.morris.umn.edu/horizons/vol2/iss2/4.

[SHM+16] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, Jan 2016.

[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.

[Tay21] Tayfun Aygün. Optimizing Monte Carlo agents for the game Katarenga. Bachelor thesis LIACS, Leiden University, 2021.

[Wik] Wikipedia. Monte Carlo tree search. https://en.wikipedia.org/wiki/Monte_Carlo_tree_search. Last accessed: 18-06-2021.

[Wona] Arcane Wonders. Onitama — An elegant and simple game of martial tactics. https://www.arcanewonders.com/game/onitama/. Last accessed: 29-05-2021.

[Wonb] Arcane Wonders. Onitama-rulebook. https://www.arcanewonders.com/wp-content/uploads/2017/06/Onitama-Rulebook.pdf. Last accessed: 02-06-2021.