UIS3901S: Independent Study Module

Group Report

Mastering Onitama using Traditional Artificial Intelligence

Student Names: | *Student Number:*

Arijit Dasgupta | **A0182766R**

Chong Yu Quan | **A0136286Y**

Theodore Vito | **A0187147W**

Student Emails:

arijit.dasgupta@u.nus.edu

e0321496@u.nus.edu

theodore.v@u.nus.edu

November 13, 2020

Contents

List of Figures

Acknowledgements

We would like to express our deepest gratitude to Professor Guillaume Sartoretti for his guidance throughout this project. The team have gained many meaningful insights with regards to traditional artificial intelligence algorithms through his mentoring. We thoroughly enjoyed the project and are proud to present our work in this report. The team is also highly motivated to further explore the field of artificial intelligence in the future through this positive experience.

Chapter 1

Introduction

Board games are known to man for thousands of years. Although different periods and territories each have their own board games, some of them are still played even today. Chess is believed to originate around the 10^{th} century, with even older forms believed to exist back in the 3^{rd} century. The earliest written record for Go dates back to the 4^{th} century BC. These board games test skills such as problem solving, improvisation, and strategy, all of which are indicative of human intelligence. Of course, there are other board games which include elements that are beyond human control, such as luck, but most board games that include high degrees of luck still require some degree of strategy to win. As time passes and technology advances, human intelligence is being challenged by artificial intelligence (AI).

In recent decades, the world of AI has seen major milestones, such as Deep Blue's victory against world chess champion Garry Kasparov in 1997 and AlphaGo's victory against 18-time world Go champion Lee Sedol in 2016. At the surface, these two examples may look very similar. Both Deep Blue and AlphaGo are computers designed to play their specific games and both of them defeated arguably the best humans at those games. However, if we examine the methods employed by Deep Blue and AlphaGo, they are very different. Deep Blue 'plays' chess using brute force; it simulates all possible moves for the next several turns and chooses the move that will bring it closest to victory. On the other hand, AlphaGo 'learns' how to play Go, it evaluates records of Go games between highly rated human players and then supplements what it has learned from humans by playing against itself. Both of these methods have their own merits and limitations, and AlphaGo's method, deep learning, is newer than Deep Blue's brute force method. This study will focus on Deep Blue's method, known as the minimax algorithm and the more advanced Monte Carlo Tree Search. Both of these algorithms are search algorithms, like the one employed by Deep Blue, instead of learning algorithms employed by AlphaGo.

For the purposes of this study, the board game, Onitama, will be used. Onitama was chosen as it has a reasonable number of possible moves and perfect information on the state of the game. A reasonable number of possible moves is important due to computing limitations. Despite the efforts to hasten computing time through heuristics like alpha-beta pruning and multi-processing, brute force search algorithms would inevitably take a significantly longer time for the computer to do computations for games with a large number of possible moves or possible game states, like in chess or Go. Onitama uses a relatively small game board, 5-by-5, compared to chess' 8-by-8 and Go's 19-by-19. Unlike chess with its many different rules for the pieces (pawns, bishops, rooks, etc.), Onitama only has rules for two pieces: the master and the pawns, and these two pieces move in the exact same way, with the only difference being that the master is tied directly to the victory conditions to be further elaborated in later sections. These two factors contribute to the ability for the computer to be able to play at human level performance while not taking an unreasonable amount of time doing so. Perfect information is necessary because it allows the decision making process to be coded in a straightforward and deterministic manner without the need to consider probabilities and expectation values. This is important as it will highlight the play strength of the computer, which may become less apparent should uncontrollable factors such as luck come to play in influencing the outcome of a game.

Chapter 2

Onitama

2.1 Rules

The rules of Onitama are elaborated in detail in the appendix, while this sections highlights the win conditions for the game. There are two possible ways to win in Onitama, which this study will refer to as conquer and destroy. Conquer will refer to either the red master moving into the blue temple or the blue master moving into the red temple. Destroy will refer to either the red team (pawn or master) capturing the blue master or the blue team capturing the red master. Due to how the game is coded currently, if both conquer and destroy is achieved at the same time (e.g. the blue master killing the red master who is standing in the red temple), destroy will take priority over conquer.

2.2 Code Representation

In this study, the board game Onitama is represented using a Python class containing the necessary attributes and functions covering all the gameplay mechanics as well as the implementation of the minimax and Monte Carlo Tree Search (MCTS) algorithms. The minimax and MCTS algorithms are also Python classes that interact with the Onitama class for the implementation of each algorithm. The entire game interacts with the human user via a generated text interface. For Onitama, the board is represented using a nested list representing a 5 by 5 array. A Cartesian coordinate system in the form of a tuple (i, j), where i is the row number and j is the column number, is used to represent each square on the board, with the origin (0, 0) on the top left as illustrated in Figure 2.1 shown below.

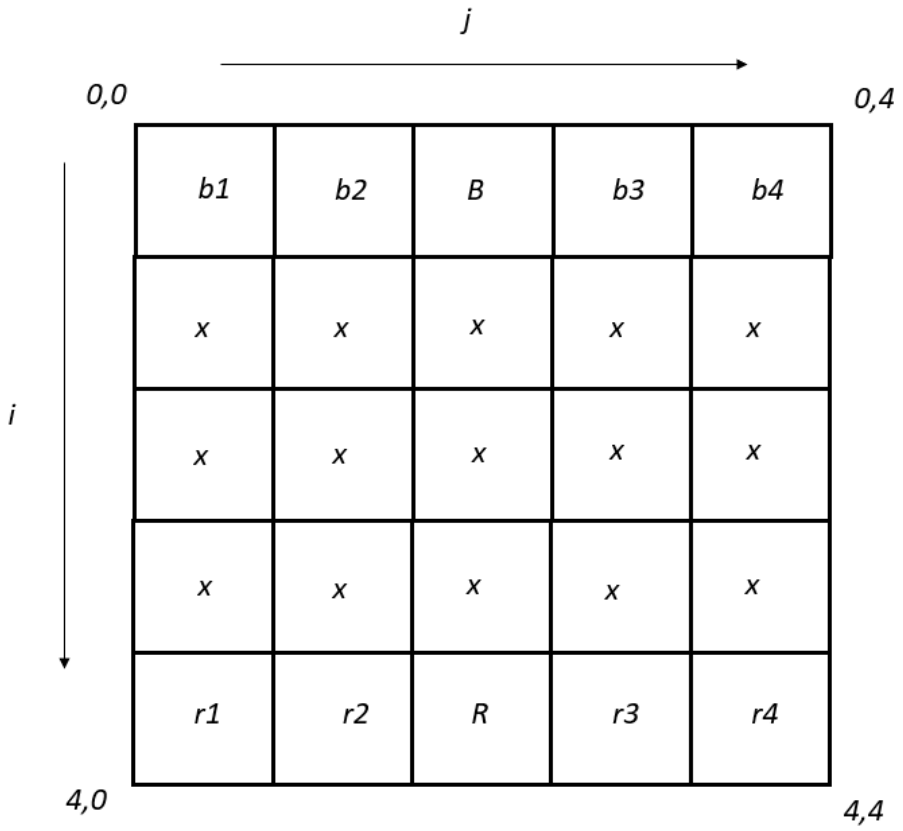


FIGURE 2.1: Coordinate convention of the Onitama board

Each piece will also be named separately to make identification of the pieces (especially the pawns) easier for human players. B represents the blue master while b1, b2, b3, b4 represent the four blue pawns. The same convention goes for the red pieces. This naming convention comes at the cost of increasing the number of permutations when executing the minimax algorithm. This is because with the code as it is, b1(0, 0) and b2(0, 1) is different from b1(0, 1) and b2(0, 0) while ideally both conditions should be considered the same as both b1 and b2 are blue pawns and other than their positions they are identical pieces. The state of each piece is saved in the 'piece_state' dictionary, with the name of the piece (b1, B, r1, etc.) as the key. Each key will correspond to either the coordinates of the piece (for alive pieces) or '-1' (for dead pieces).

The 16 different cards are coded in the 'cards' dictionary, containing the name of the card (tiger, ox, etc.) with each name being a key to a dictionary containing the possible moves of each card. The moves are named A, B, C, D and are represented in the dictionary as changes in coordinates. For example, move 'A' for tiger is (2, 0), meaning that the piece will move 2 steps in the i-direction (down) and 0 step in the j-direction. There is another dictionary containing the name of the cards as a key

to the colour of the card for the purposes of determining the first player to move. To make a move, the player or AI must provide 3 inputs: a piece name, a card name, and a move name (e.g. b1 ox A). Checks are then run to ensure the selections are valid and once all the validity checks are passed, the moves are executed and the game state is updated (position changes, pieces killed, the movements of the cards, etc.).

As stated before, there are two win conditions: conquest and destroy. The code representation of 'conquest' is either 'B' moving into (4, 2) resulting in blue winning or 'R' moving into (0, 2) resulting in red winning. If at the end of the turn either 'R' or 'B' is dead (represented as '-1' in the piece_state dictionary), the team with their master piece still alive wins the game by 'destroy'.

Chapter 3

Algorithms

3.1 Minimax

In the implementation of AI into the code, the algorithm 'minimax' is used. Minimax is an exhaustive search algorithm that allows the computer to choose its best next move by simulating all possible (legal) moves for the future n-turns, where n is the depth of the minimax algorithm; how many turns into the future the computer is looking through. For its next move, the computer will then select the move that will give the best outcome after the n-turns have passed, according to its simulation. In this simulation, the computer plays both sides, with the red player as the maximising player, meaning that it will attempt to gain the highest number of points, and the blue player as the minimising player, which aims to get the lowest number of points (a large negative value). The value of any given game state is calculated using 3 factors (in decreasing importance): winning, having pieces alive and killing enemy pieces, and distance between own master and the enemy's temple. The purpose of having minimising and maximising players is to simulate the opponent (human or computer) only making the best moves possible, so if the opponent makes a less than ideal move, it can be capitalised upon. However, being an exhaustive search function, the computation time of minimax will increase exponentially as its depth increases. This is a significant problem in the games with a large number of possible moves such as chess and Go as the computation time becomes unreasonably long for a depth that is necessary to make 'good' moves. To counteract this problem, alpha-beta pruning is used.

Alpha-beta pruning is done by adding two variables, alpha (α) and beta (β), to the minimax algorithm. Alpha and beta are variables that store the best scores found in the search so far for each player, alpha represents the maximising player while beta represents the minimising player. This means that the value of alpha can only be changed during the maximising player's turn (in Onitama's

case, red) while beta can only be changed during the minimising player's turn (blue). The purpose of alpha and beta is to determine whether or not another branch of the search provides a better score compared to the current branch, hence making it unnecessary to continue searching on the current branch. A simple example on the details of how minimax with and without alpha-beta pruning can be found in the appendix.

3.2 Monte Carlo Tree Search

Instead of an exhaustive search like minimax, MCTS builds a statistical decision tree and uses a random rollout sampling to aid it in searching large state spaces. MCTS has 4 main components: selection, expansion, simulation, and back-propagation. In the selection stage, MCTS makes a selection from the parent node, which represents the current game state, to a child node, representing the game state after a specific sequence of legal actions has been executed, based on the value of the child node. The value of the child node is determined by the following equation 3.1, where v is the value of the child node, w_i is the number of wins after the i^{th} move, n_i is the number of simulations after the i^{th} move, c is exploration parameter and t is the total number of simulations for the parent node.

$$v = \frac{w_i}{n_i} + c * \sqrt{\frac{2 * log(t)}{n_i}} \tag{3.1}$$

Using equation 3.1, the fine balance of exploitation (choosing the nodes with the best ratio of number of wins to number of simulations) and exploration (choosing the nodes that are least explored) is maintained. This selection process is repeated until a child node is selected that no other child nodes are attached to it. From that child node, the expansion step occurs, where a new node will be added to the statistical decision tree. After selection and expansion, assuming that the child node is not a terminal state, the simulation phase occurs, where the game is played using a random policy till termination with a corresponding outcome of win (1) or loss (0). After simulation, back propagation occurs, where the all the nodes attached to the child node back to the parent node are updated in 2 aspects: the number of win and the number of simulation. This process of selection, expansion, simulation and back propagation then repeats again starting from the parent node, for a fixed number of iteration or a fixed amount of time, from which the the next move is selected from the child node directly attached to the parent node with the most number of simulations (not the number of wins or the ratio of the number of wins to number of simulations). This whole process then repeats again after a move has been made using the selected move.

Chapter 4

Experiments

4.1 Methodology

To test the capabilities of the Minimax and MCTS, we will simulate the gameplay among numerous different combinations of the algorithms. We aim to answer the following 9 questions by pitting an AI representing the red player and an AI representing the blue player.

1. If both red and blue players use **Minimax** of the **same** depth, but **only the blue player** gets to start first all the time, will the blue player always win?

2. If both red and blue players use **Minimax** of the **same** depth, and both players get to start first equally, will they match evenly in average performance?

3. If both the players use **Minimax**, but the red player uses a **higher** depth, will the red player always win?

4. If both red and blue players use **MCTS** of the **same** depth, but **only the blue player** gets to start first all the time, will the blue player always win?

5. If both red and blue players use **MCTS** of the **same** depth, and both players get to start first equally, will they match evenly in average performance?

6. If both the players use **MCTS**, but the red player uses a **higher** depth, will the red player always win?

7. If the red player uses **Minimax** and the blue player uses **MCTS**, how do the two algorithms match? In other words, how many iterations does the blue player need to run to match the performance of the red player at different depths?

8. When AIs of greater capability face off, do they last more turns as compared to AIs of lesser capabilities? (Capability refers to the depth in Minimax and number of iterations in MCTS)

9. How do the AIs of different capabilities and types (Minimax or MCTS) prefer to defeat their opponent? By destroying or conquering?

One may notice that questions 1-3 match 4-6, with the difference being that the former only considers Minimax and the latter MCTS. To directly answer the first 7 questions, we have devised an experimental plan accordingly shown in Table 4.1 below.

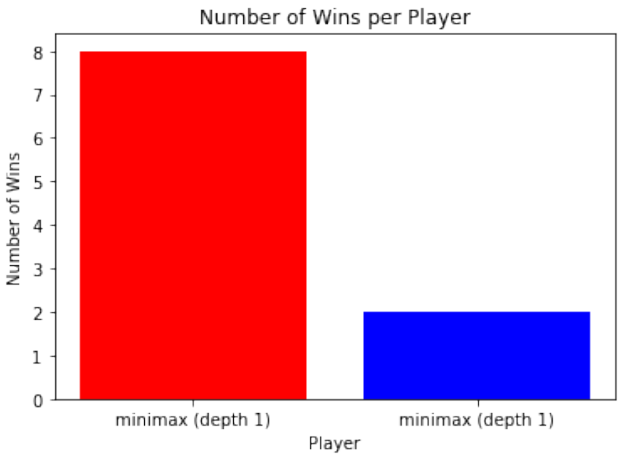
TABLE 4.1: Summary of the first 7 experiments for performance of Minimax and MCTS

Expt.	Red Player	Blue Player	First Move	Performance Comparison	Remarks
1	Minimax	Minimax	Blue Always	Both have the same depth	Depth varied from 1 to 6
2	Minimax	Minimax	Both get equal chances	Both have the same depth	Depth varied from 1 to 6
3	Minimax	Minimax	Both get equal chances	Red has a higher depth (either +1 or +2)	The following comparisons are made: 2v3, 3v4, 4v5, 5v6, 2v4, 3v5, 4v6
4	MCTS	MCTS	Blue Always	Both have the same number of iterations	Following iterations tested: 100, 500, 1000, 5000
5	MCTS	MCTS	Both get equal chances	Both have the same number of iterations	Following iterations tested: 100, 500, 1000, 5000
6	MCTS	MCTS	Both get equal chances	Red has a higher number of iterations	The following comparisons are made: 100v500, 500v1000 & 1000v5000
7	Minimax	MCTS	Both get equal chances	Varied for both players	Depth varied from 2 to 6 and Iterations varied from 100, 500, 1000, 5000, 10000

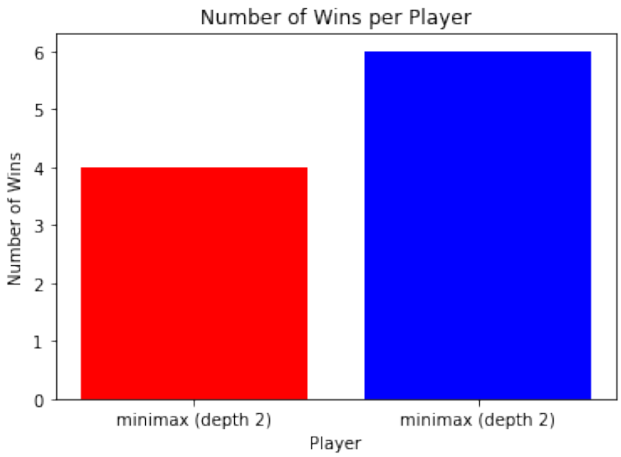
For each experiment, multiple cases are considered. For instance, in experiment 2, both red and blue players are pitted against each other for depths of 1-6. In each one of these cases, a total of 10 gameplays is simulated. 10 iterations is sufficiently high to determine if both algorithms are performing equally or one is clearly beating the other. The number of iterations is limited to 10 in consideration for the high computation cost and time required, should we increase the number. This leads to a total of 55 different cases, leading to a full simulation of 550 games by the AI models. Note that the starting conditions of the gameplay (i.e. the cards assigned) are random for every game. Although some games like Minimax depth 2 vs Minimax depth 2 could finish in less than a second, more computationally expensive games like Minimax depth 6 v MCTS (10000 iterations) could take up to several minutes. To answer questions 8 & 9, we will analyse through all the simulations to identify the trends in number of turns taken for the game to end, and the manner in which the winner won.

4.2 Results and Discussion

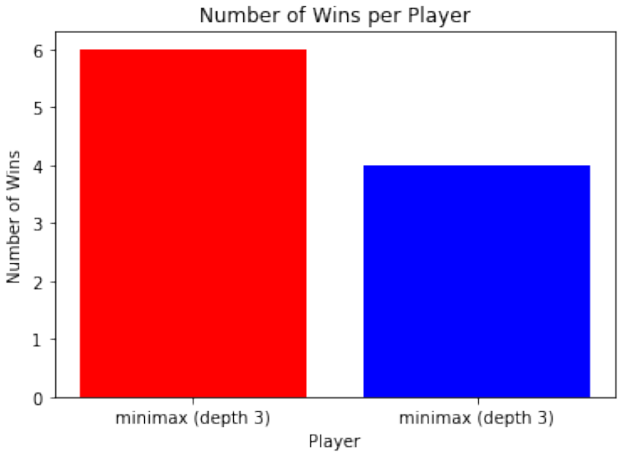
Figure 4.1 below shows the performance between two minimax AIs of same depths with the blue player going first only (experiment 1). The idea behind experiment 1 is that as both minimax have the same level of look-ahead, the starting player always has the advantage. However this assumption is proven wrong as the red player often outperforms the blue player, specifically in depths 1, 3 & 4. Usually, a player has an advantage if it has the right pieces and cards at the right time. Minimax of lower depths are not able to capitalise on those long term opportunities as it can only look fewer turns ahead. However, at depth 6, the blue player has a clear mandate over the red player after getting to go first. This may be because it is able to look far ahead enough to reduce the luck factor of both players. Hence, it is possible that the starting player has the advantage only if it has a deep look-ahead.



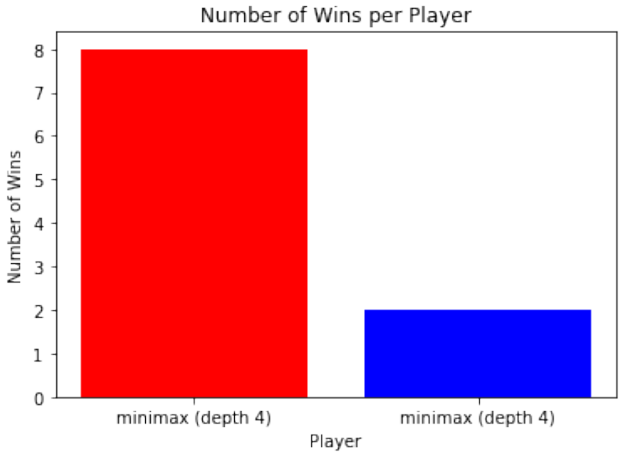
(A) Depth 1 vs Depth 1



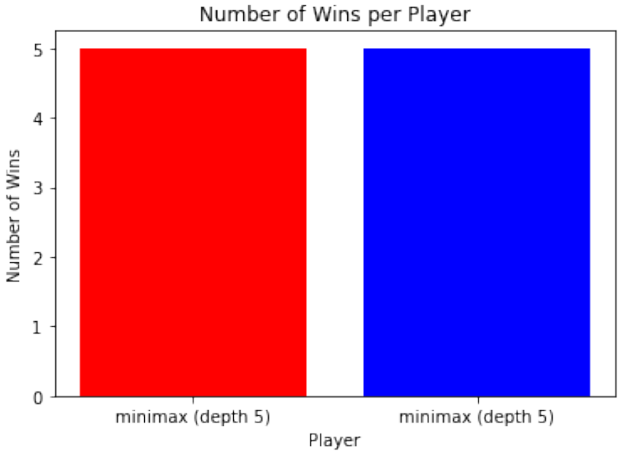
(B) Depth 2 vs Depth 2



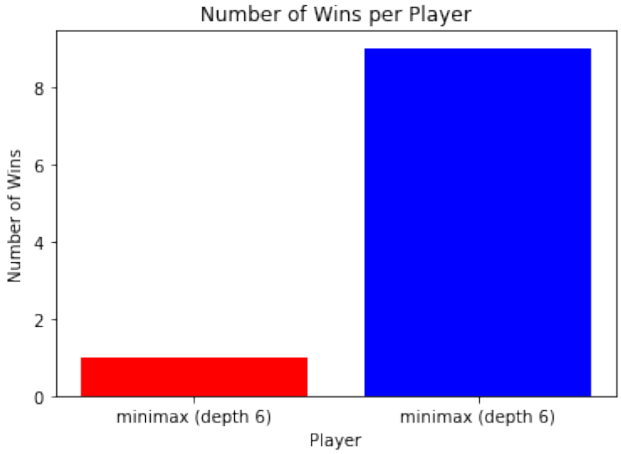
(c) Depth 3 vs Depth 3



(d) Depth 4 vs Depth 4

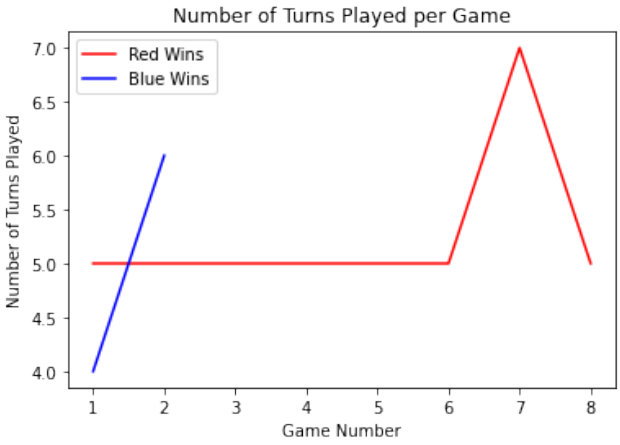


(e) Depth 5 vs Depth 5

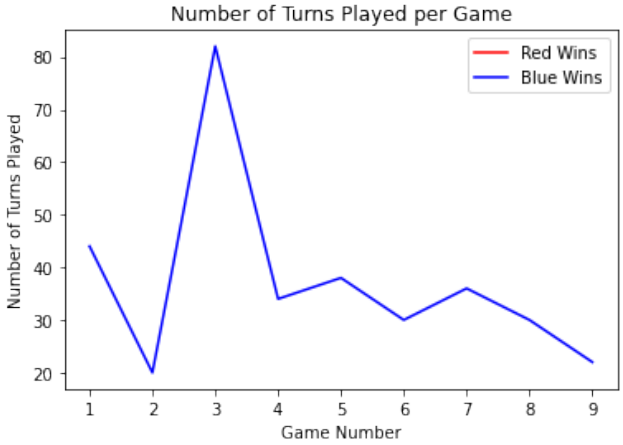


(f) Depth 6 vs Depth 6

Figure 4.1: Results for experiment 1



(a) Depth 1 vs Depth 1

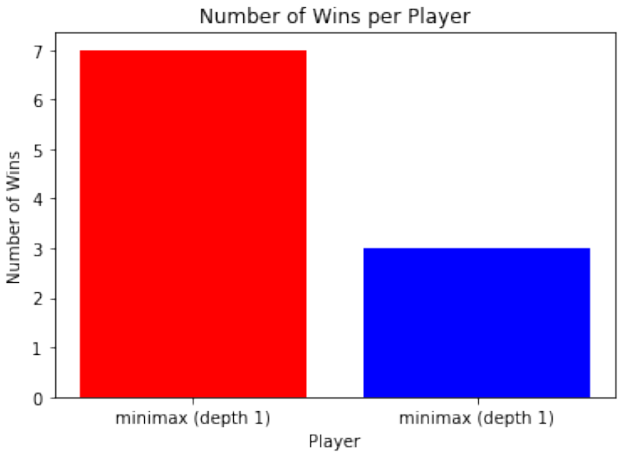


(b) Depth 6 vs Depth 6

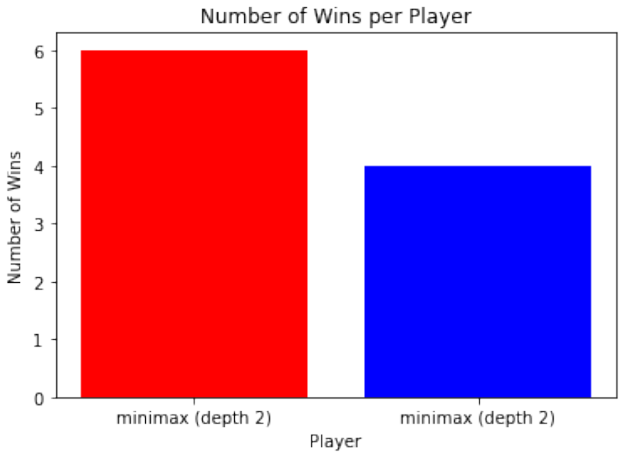
Figure 4.2: The different number of turns played in different depths

Figure 4.2 above compares the number of turns played in a match when blue or red won at depths of 1 and 6. Naturally, games with higher depths take longer to finish as AIs of lower depth go for

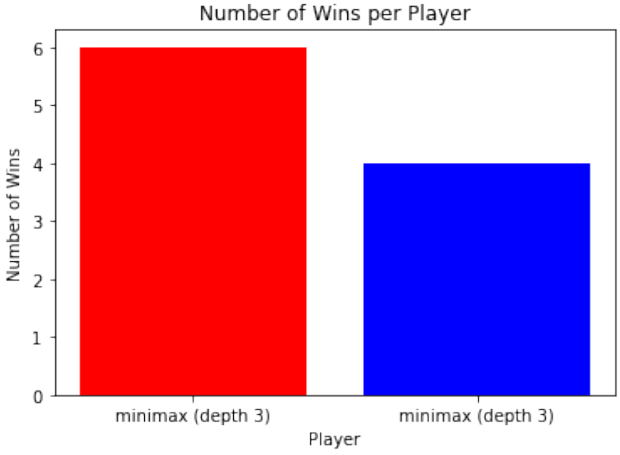
immediate rewards as they can't consider counter-moves their opponent can make in subsequent turns. However, AIs of high depths can see the many ways their opponents can trap them and are hence likely to be more cautious. This is quite reminiscent of top Chess players.



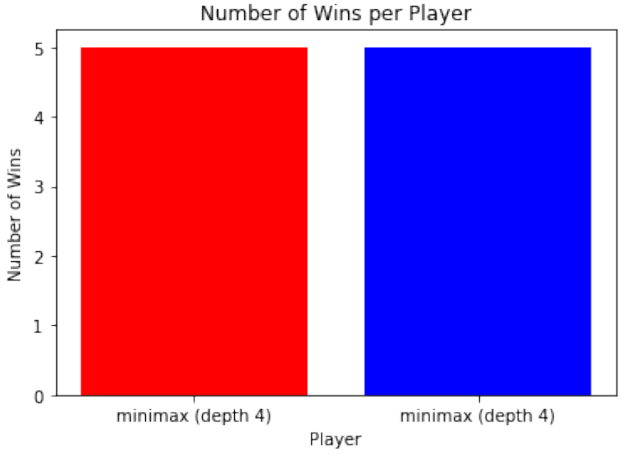
(A) Depth 1 vs Depth 1



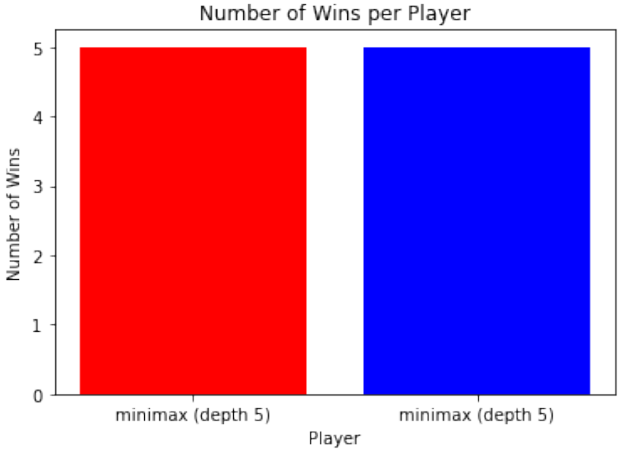
(B) Depth 2 vs Depth 2



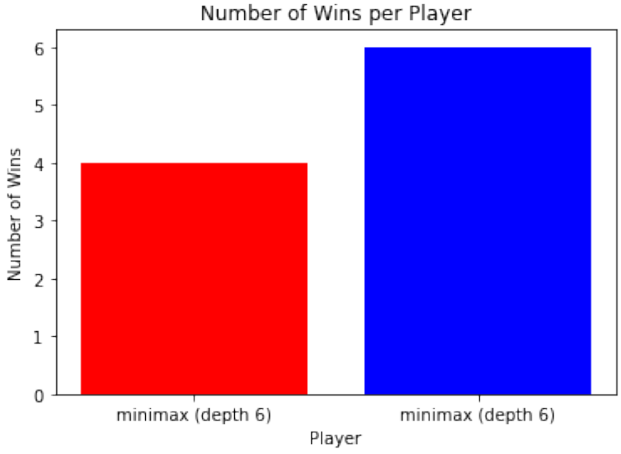
(C) Depth 3 vs Depth 3



(D) Depth 4 vs Depth 4



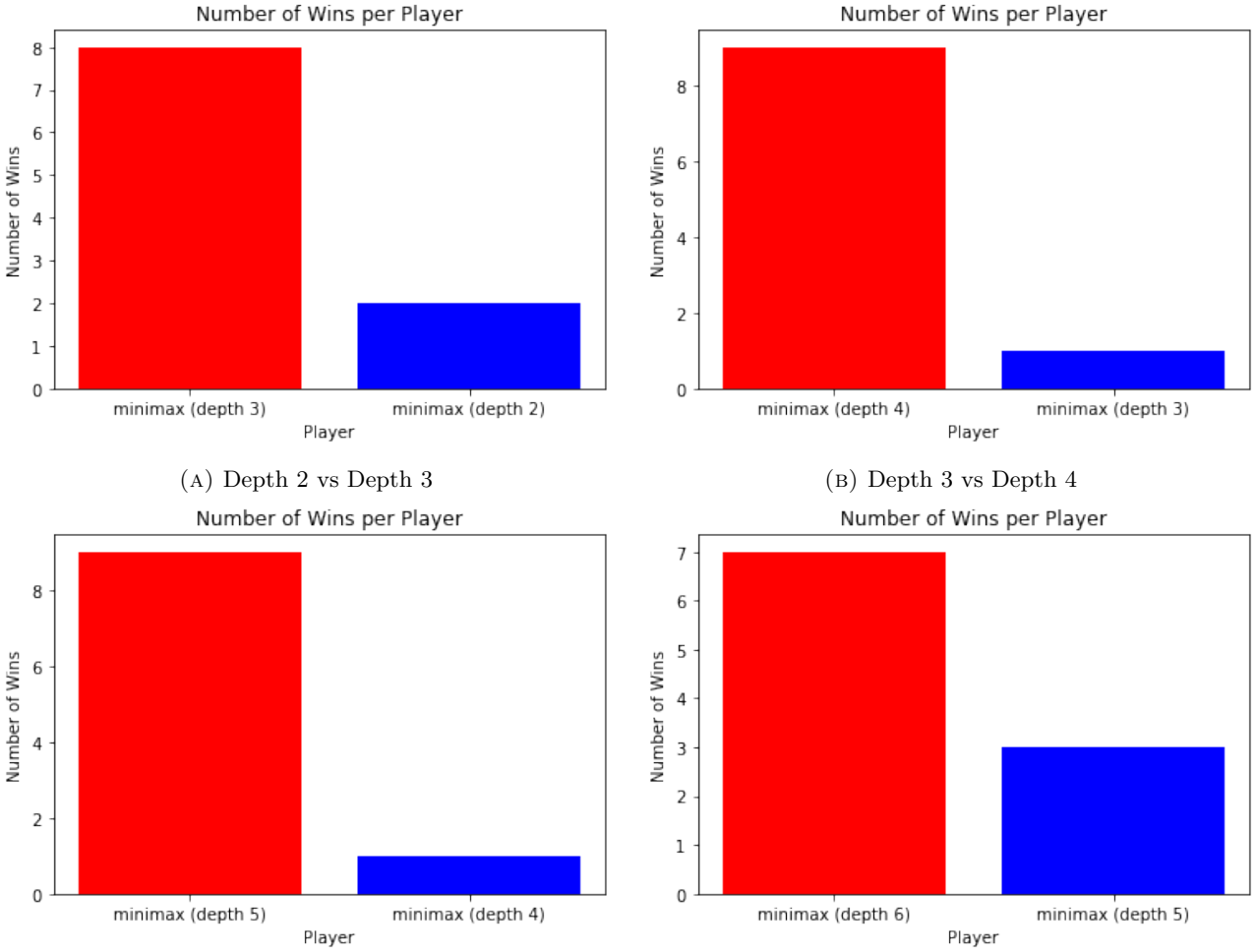
(E) Depth 5 vs Depth 5



(F) Depth 6 vs Depth 6

FIGURE 4.3: Results for experiment 2

In experiment 2, both the red and blue players start 5 times each. The results are shown in Figure 4.3 above. At all depths, there is no clear indication of one player outperforming the other. Any small differences could arise by chance due to the random advantage of randomly assigned cards every game. Hence, we conclude that neither Minimax player has an advantage when they each get to start first equally. This is expected as the parameters of play for both blue and red are the same. But what if the red player is given a slight edge? Experiment 3 explores this by giving the red player a depth of 1 higher than the blue player with results shown in Figure 4.4 below.



(A) Depth 2 vs Depth 3

(B) Depth 3 vs Depth 4

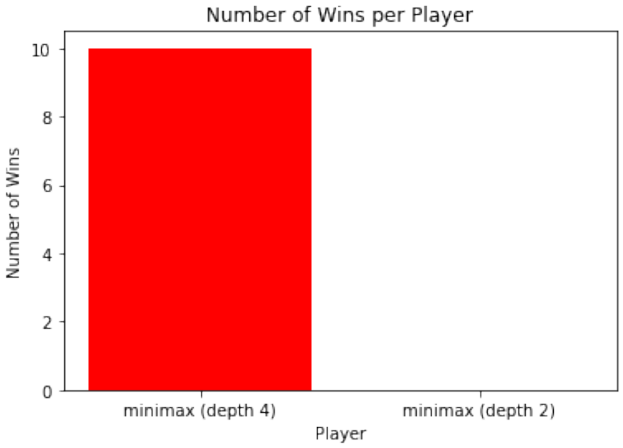
(C) Depth 4 vs Depth 5

(D) Depth 5 vs Depth 6

FIGURE 4.4: Results for experiment 3 with difference in depth of 1

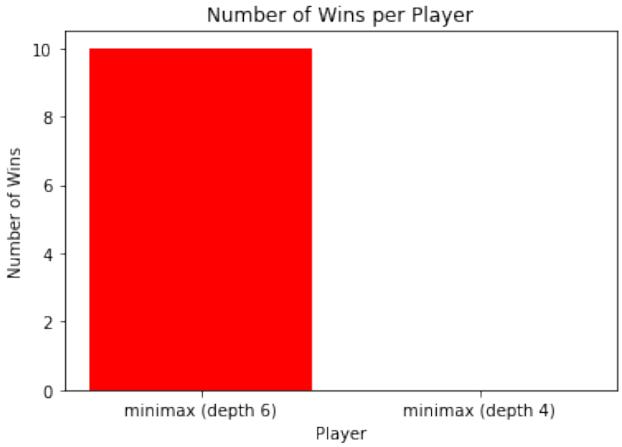
As expected, the red player outperforms the blue player in all cases as the red player is able to look ahead one extra move, which could shape a decisive move that the blue player cannot reverse as it was not able to "consider" it. An interesting observation made is that the red player did not completely annihilate the blue player. In all of the cases, the blue player manages to win at least one game from the red player, suggesting that the luck factor of the initial cards could still be at play.

Hence, a higher depth difference is required to win all 10 games. The red player is now tested with a depth that is higher than the depth of blue by 2. The results are shown in Figure 4.5 below, where it is apparent that a depth difference of 2 was enough to give the red player a win in every single game.



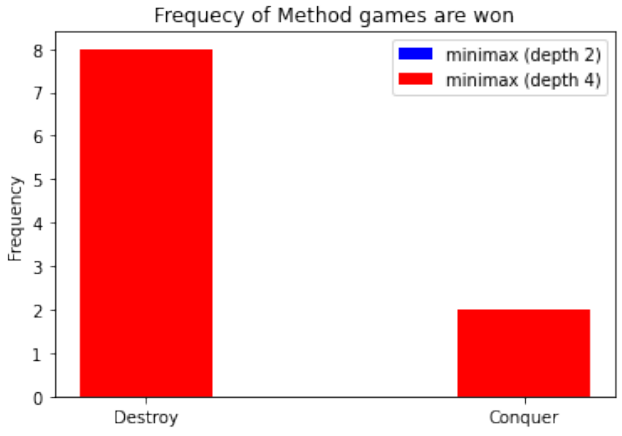
(A) Depth 2 vs Depth 4

(B) Depth 3 vs Depth 5

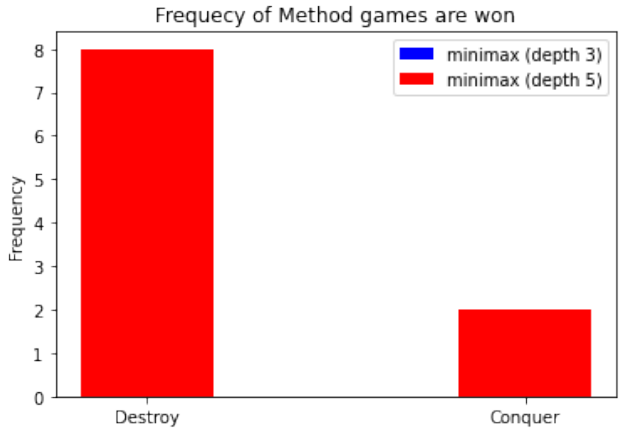


(C) Depth 4 vs Depth 6

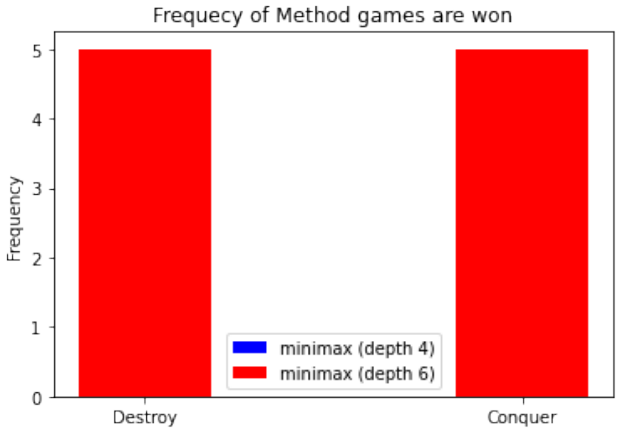
FIGURE 4.5: Results for experiment 3 with difference in depth of 2



(A) Depth 2 vs Depth 4



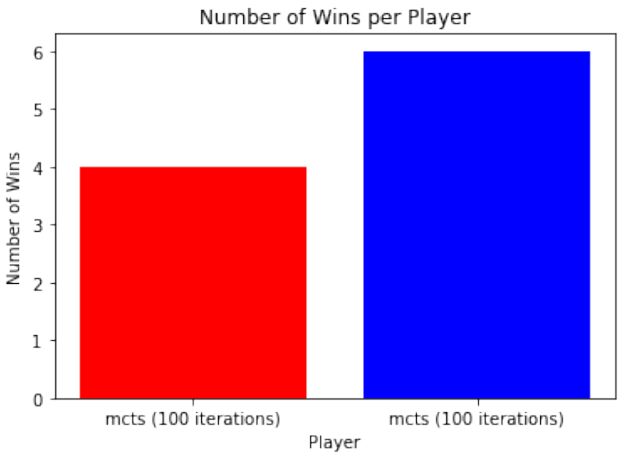
(B) Depth 3 vs Depth 5



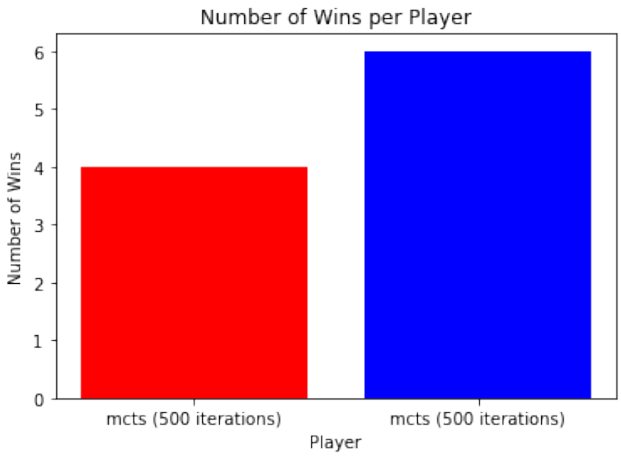
(C) Depth 4 vs Depth 6

FIGURE 4.6: Results for experiment 3 of victory methods with difference in depth of 2

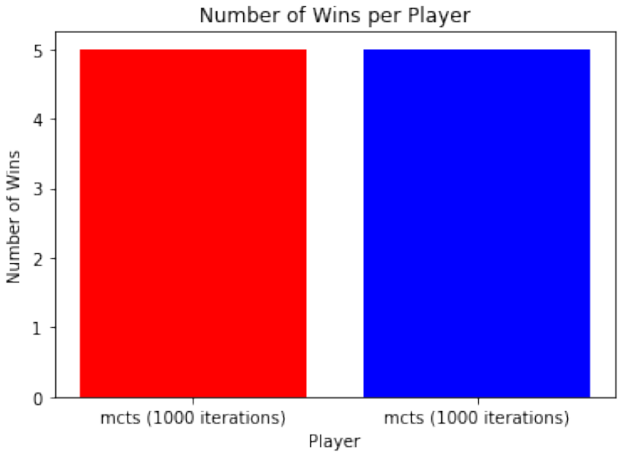
Figure 4.6 shows the manner in which the red player wins all the matches. At depths of 4 & 5, the red player preferred to win by capturing the opponent's master piece, while at a higher depth of 6, it chose to conquer half the time. A possible explanation could be due to the fact that winning by "conquering" is generally a more unlikely and harder method to win by relative to "destroying" as it may require a stricter set of conditions. Nevertheless, there is insufficient data points to support the claim and we are not experts at Onitama as well. In the next experiment, we adopt the MCTS into both the red and blue players and pit them against each other. Like we did for the minimax, we give both players the same number of iterations while only letting the blue player start every time. Figure 4.7 below shows the results for experiment 4.



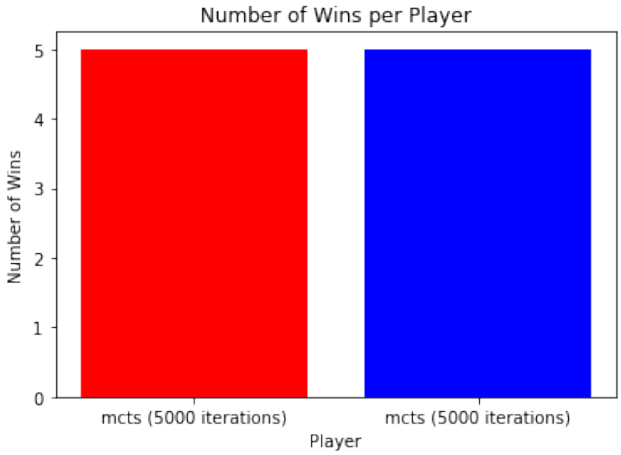
(A) 100 iterations vs 100 iterations



(B) 500 iterations vs 500 iterations



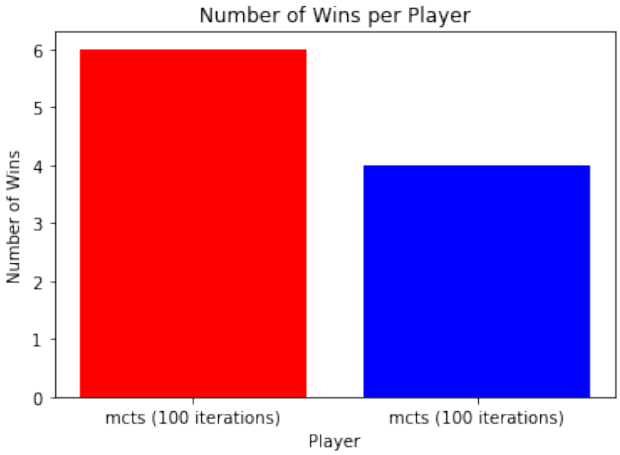
(C) 1000 iterations vs 1000 iterations



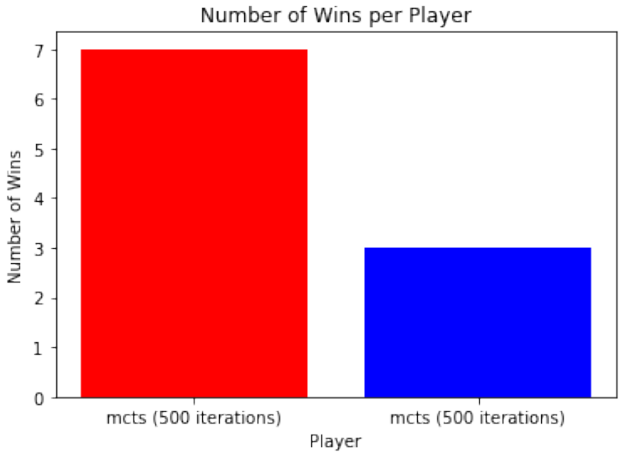
(D) 5000 iterations vs 5000 iterations

FIGURE 4.7: Results for experiment 4

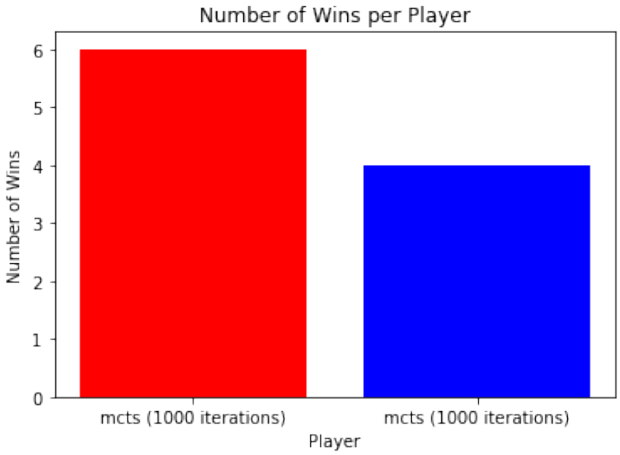
Even though the blue player gets to always start first, the results show that both players of the same number of iterations perform relatively equally. Hence, it does not matter which player starts first. At this point, it is to be reminded that unlike the minimax algorithm, the MCTS is not deterministic by nature. Despite there being perfect information and deterministic state transitions in Onitama, the MCTS uses a random rollout policy. Hence, even if we give the same state, the MCTS may compute slightly different outcomes in all of its iterations. Nonetheless, with a high enough number of iterations, the optimal move should usually be the same. This could partially explain why the MCTS results have some variability.



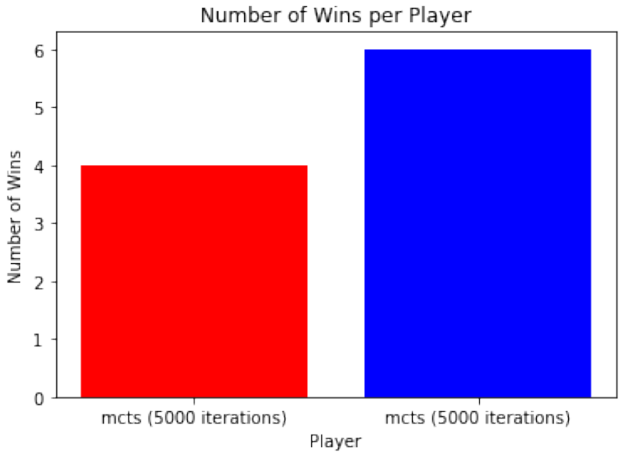
(A) 100 iterations vs 100 iterations



(B) 500 iterations vs 500 iterations



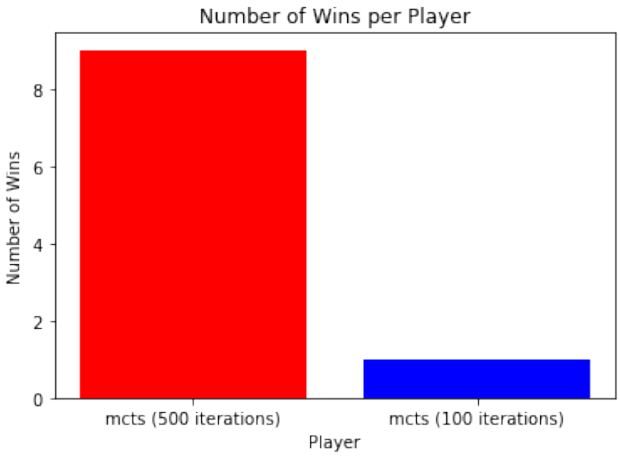
(c) 1000 iterations vs 1000 iterations



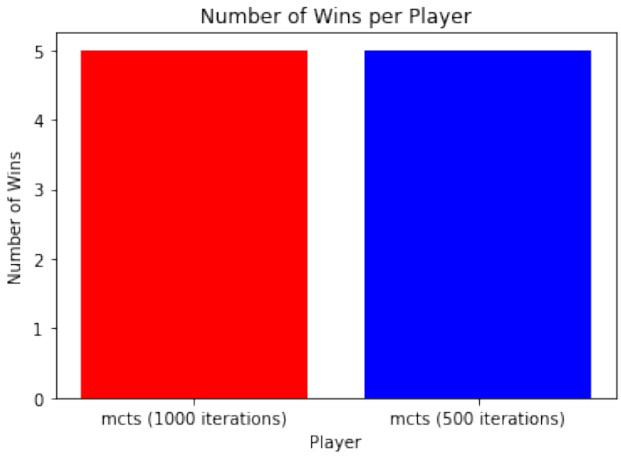
(d) 5000 iterations vs 5000 iterations

FIGURE 4.8: Results for experiment 5

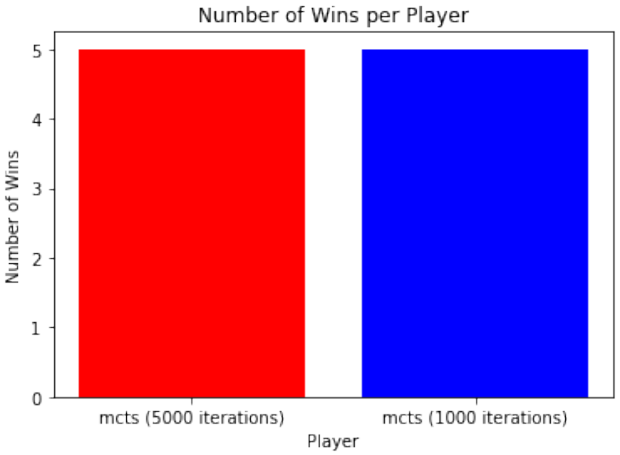
Likewise, when we let both players have equal chances to start first, they perform equally in general as well. As shown in the results of experiment 5 in Figure 4.8, neither player outperforms the other player heavily. Any small differences in performance could arise by chance given the random nature of the random rollout of MCTS during simulation. Now just like we did to minimax, what if we let the red player simulate more iterations per turn? Figure 4.9 below summarises the results of experiment 6.



(a) 500 iterations vs 100 iterations



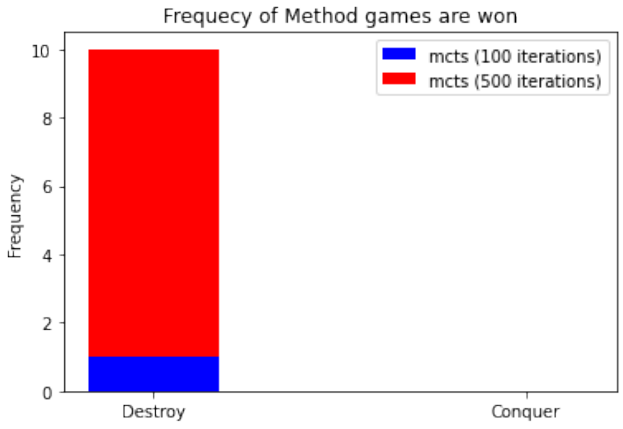
(b) 1000 iterations vs 500 iterations



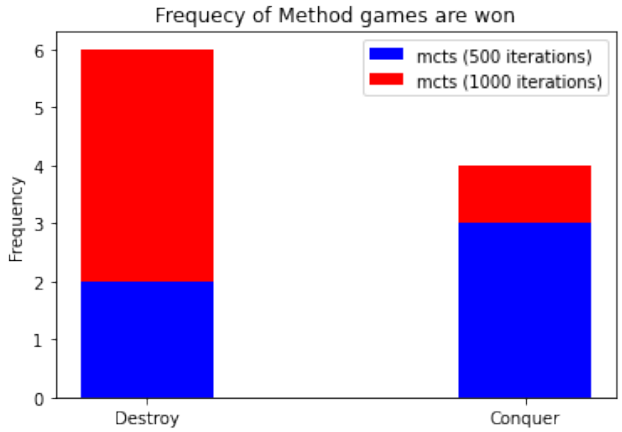
(c) 5000 iterations vs 1000 iterations

FIGURE 4.9: Results for experiment 6

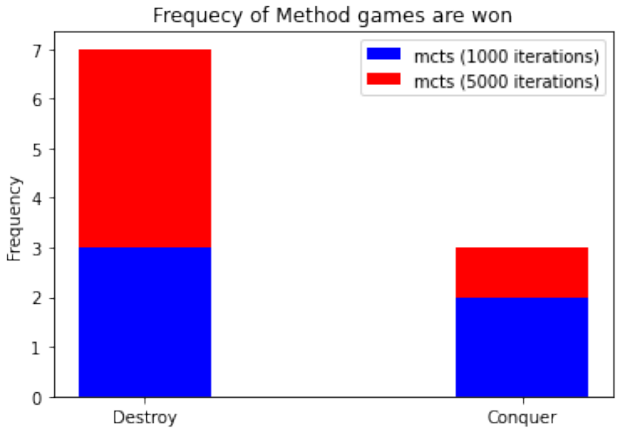
The results from this are rather surprising. In the case of 100 iterations v 500 iterations, the red player heavily outperformed the blue player as one could naturally expect. However, for the other 2 cases, both the players performed equally. It is hard to understand why an MCTS AI that performs 5 times as many simulations as the other can perform on equal terms. One possibility may be that the range of 500 to 5000 iterations would perform in a similar manner as there may not have been enough iterations to look ahead enough to find winning moves. At every depth of the game look-ahead tree, the number of total possible states exponentially increases , hence 5000 iterations is not exploring 10 times the depth of 500 iterations, it is most likely to be significantly less. Therefore the improvement from 500 iterations to 5000 may not be as pronounced as one would hope for. We now take a closer look at the manner in which the red and blue players won the games as shown below in Figure 4.10.



(a) 500 iterations vs 100 iterations



(b) 1000 iterations vs 500 iterations



(c) 5000 iterations vs 1000 iterations

FIGURE 4.10: Results for experiment 6 of victory methods

The main takeaway from Figure 4.10 is that the player with the higher number of iterations (red player) generally prefers to win by the Destroy method. The blue player however often seeks to win by Conquering. Perhaps it is possible that the player with less iterations takes a less aggressive strategy on the opponent, which ultimately gives it an advantage in conquering the opponent instead. Usually in the game of Onitama, a player often has to make a choice between a strategy that favours 'Destroy' than 'Conquer' and vice versa. This however, is merely a guess for it is very difficult to substantiate its possibility.

The final experiment conducts a thorough comparison between Minimax and MCTS of different depths and iterations respectively. We simulated the gameplay between them at all combinations of Minimax depths 2-6 and MCTS from 100, 500, 1000, 5000, 10000. This gives rise to a total of 25 different cases and 250 games simulated. Instead of representing the results with the plots, a heatmap is as shown in Figure 4.11 to visualise the overall results. For the performance of the players, a +1 score is used to represent a win by Minimax per game while a -1 score is used to represent a win by MCTS per game. Hence if MCTS wins all games, the score in the cell will be $(+1)*0+(-1)*10 = -10$. If Minimax wins 6 times and MCTS wins 4 times, the score will be $(+1) * 6 + (-1) * 4 = +2$ and so on.

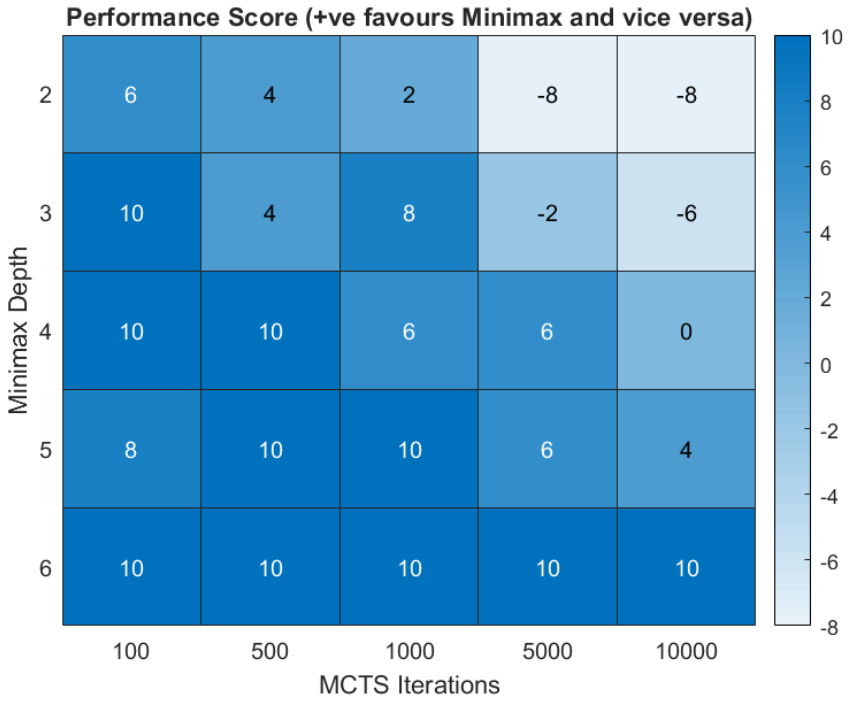


FIGURE 4.11: Results of experiment 7 of performance scores

The heatmap shows the trend of how Minimax and MCTS perform at different capabilities. As a low minimax depth of 2, Minimax is able to effectively perform better than MCTS at iterations of up to 1000. This tells us that even 1000 rollout simulations may not be enough to beat a low depth of 2 more often than losing. We consider the Depth of 2 to have a similar performance to 1000 MCTS iterations. However at MCTS iterations of 5000 and 10000, it completely dominates depth 2. Still, the story changes slightly at depth 3, where 5000 iterations still performs slightly better, but not by much, winning 6 out of 10 games. 10000 iterations still moderately outperforms depth 3. In this case, we consider the performance of depth 3 to be equal to 5000 iterations. In the case of depth 4, the Minimax always beat MCTS with the exception of 10000 iterations where both AIs won 5 games each. This is to clearly say that depth 4 performance for Minimax is equal to 10000 iterations in MCTS. In the cases for Minimax depths 5 & 6, the MCTS was constantly beaten. In fact Minimax depth 6 beat MCTS 10000 iterations constantly in every game. This signals to us that we may potentially need significantly more iterations than 10000 to stand a chance of winning a game against a powerful Minimax AI of depth 6. In the next analysis, we look at the heatmap in Figure 4.12 for the average number of turns taken for all the games in each case to finish.

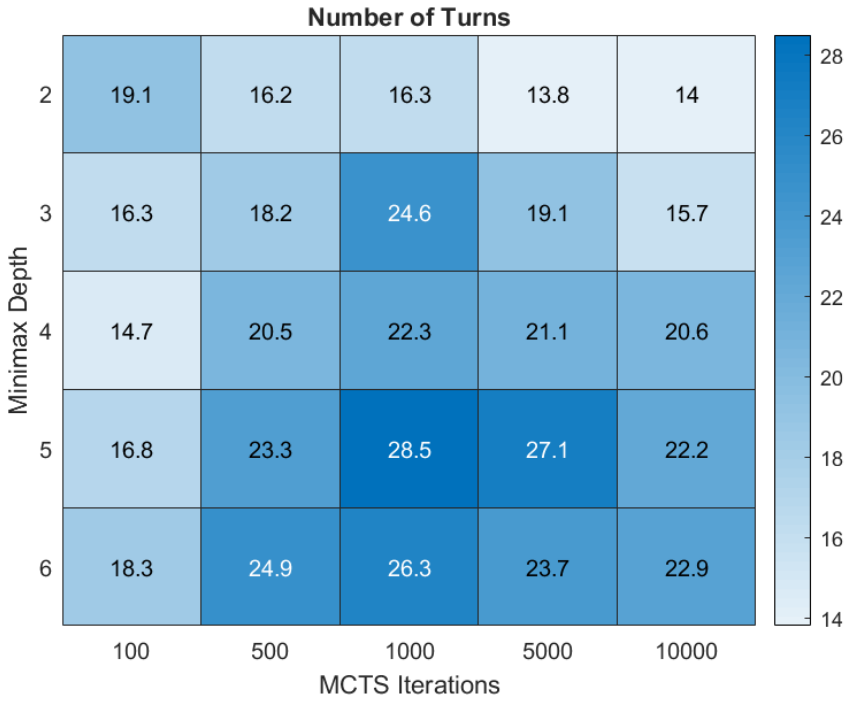


FIGURE 4.12: Results of experiment 7 of average number of turns in games

The aim of this is to answer question 8 from the experimental methodology section, to see if stronger AIs take longer to finish a game. Visually, the top left region of the heatmap represents weaker AIs for both Minimax and MCTS while the bottom right represent the strong ones. The heatmap does indeed show an increase in the average moves taken to finish the game when moving from weak to strong AIs, however this increase is not by much. The number of turns taken on average for Minimax depth 2 v MCTS 100 iterations is 19.1, while the highest average is 28.5, which is not that significant. Another interesting trend we see is that when stronger AIs play against weaker AIs, the number of turns also decreases. Along the top row of the heatmap, the number of turns reduces as the MCTS gets more capable against a Minimax of depth 2, from 19.1 to 14. This is also not surprising as a dominant AI can make better moves to end the game quickly. The final heatmap to analyse would be the manner in which the winners of the games in each case won. A 'Destroy' is given a score of +1 while 'Conquer' is given a score of -1. The heatmap is shown in Figure 4.13.

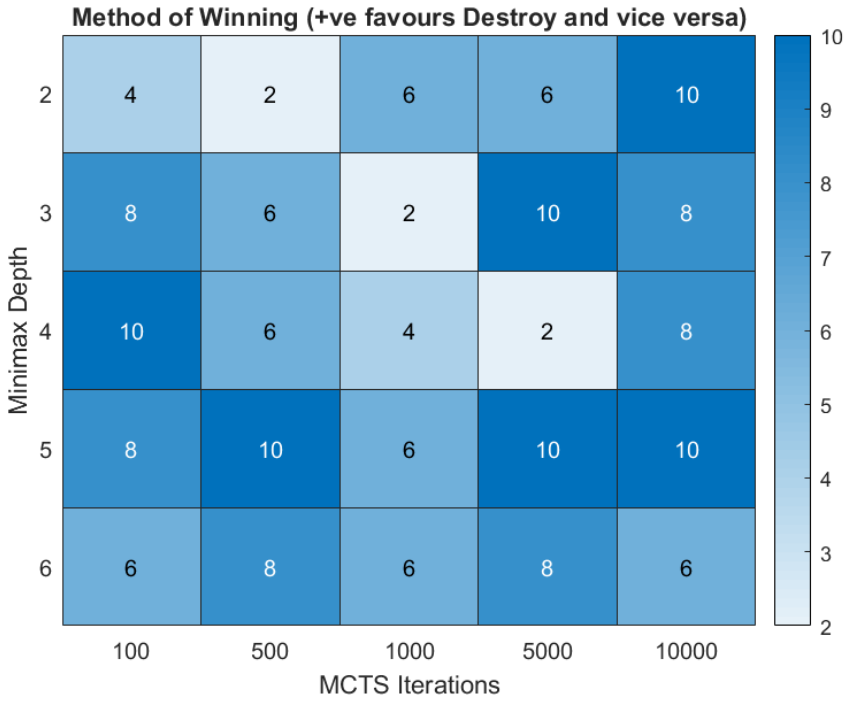


FIGURE 4.13: Results of experiment 7 of preferred methods of winning in games

This heatmap aims to explore question 9 of the experimental methodology, to identify any trends in the manner the AI wins at different capabilities. The heatmap tells us that there is none that we can conclude on. There is no clear trend or pattern to see if there is a manner in which the AI changes strategy. It is possible that this choosing of strategy is a lot more subtle, and hence requires a more fine grain analysis, where we increase the number of cells in the heatmap above. Another possible change to make could be increasing the number of iterations in each cell, if enough computational time is available. This concludes the section on the results from the experiments run in this project.

Chapter 5

Conclusion

With regards to further improvements of the project, we would like to be able to run the experiments with a greater number of games than 10 should time be not a constraint. This would allow the inferences drawn to hold greater weight especially for the case of MCTS. In addition, further experiments can be implemented to the MCTS with a greater number of iterations (beyond 10000) to find out the point for which the MCTS matches the performance of the minimax algorithm for each depth. While many tests have been conducted between the algorithms with varying parameters, one key question with regards to the algorithms would be their performance against real humans. Through a few tests with friends and between the group, we are happy to conclude that the algorithm is definitely capable of human level performance and beyond. For example, the minimax algorithm running with a depth of 4 would usually win against any human players and any depth beyond 4 would have utterly decimated any human players. However, as with the results of the tests, the general consensus between us as well as the participants that played against both algorithms was that the minimax algorithm is in general a much better algorithm than MCTS for Onitama. This is due to fact the play strength of the MCTS stems from the amount of sampling (through larger number of iterations etc.), which we limited for a reasonable amount of response time during play.

While the random rollout sampling of MCTS can be its weakness in the context of the Onitama game, we recognize that the samping nature of MCTS a key strength of the algorithm in the context of games with significantly more complex board states (e.g. Go). While minimax works sufficiently well for a reasonable number of depth, it suffers from the curse of dimensionality with increasing number of possible legal actions at any given state as it is a deterministic search algorithm. Given that the only way to get the algorithm to work is to significantly reduce the number of depth, the corresponding sharp drop in performance makes the minimax algorithm implausible for tackling problems like Go.

However, the sampling based MCTS position itself in a better position in addressing increasing high dimensional and complex problems beyond Onitama. Nevertheless, the minimax algorithm remains the best algorithm for Onitama for traditional algorithms. Our team hopes to extend this project by implementing the relatively more modern deep reinforcement learning methods and pitting them against the traditional algorithms in this project.

Appendix A

Detailed Rules of Onitama

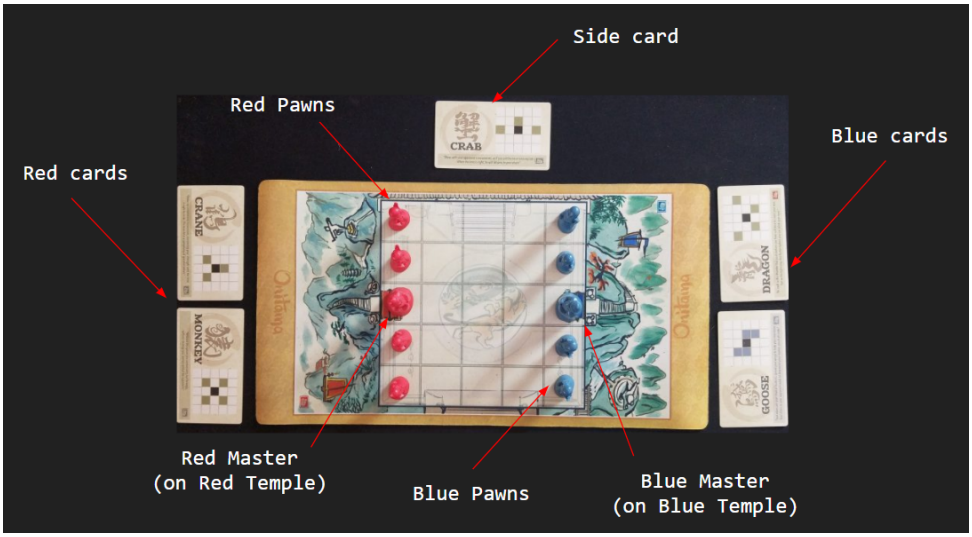


FIGURE A.1: Entire Onitama game state

Figure A.1 highlights the entire board state for the players with the red arrows indicating the components that would be described in detail below.

- The board: The onitama board is a 5x5 game board with a red temple on one side and the blue temple on the opposite side. The master pieces start the game in the temple of their respective colours, flanked by two of their pawns on either side.
- The pieces: Each side has a master piece and four pawn (student) pieces. These pieces move in the same exact rule, determined by the cards.
- The cards: There are 16 cards inside the Onitama deck as shown in Figure A.2 below.

FIGURE A.2: All 16 cards in Onitama

Each card is given a name, a moveset, and a colour. Some flavour text is also written on each card but these are not relevant to the game. Each card has between two and four moves in their moveset. These moves determine how each piece can move for the current turn. In the moveset of the card, the black square represents the current position of the piece, and the other squares squares (colour does not matter) represents where that piece can move with respect to its current position. For example, if the blue player chooses his master and the card 'Ox' (shown in the Figure A.3 below), his master can move one square forwards, backwards, or to the right. All the directions are with respect to the current player.

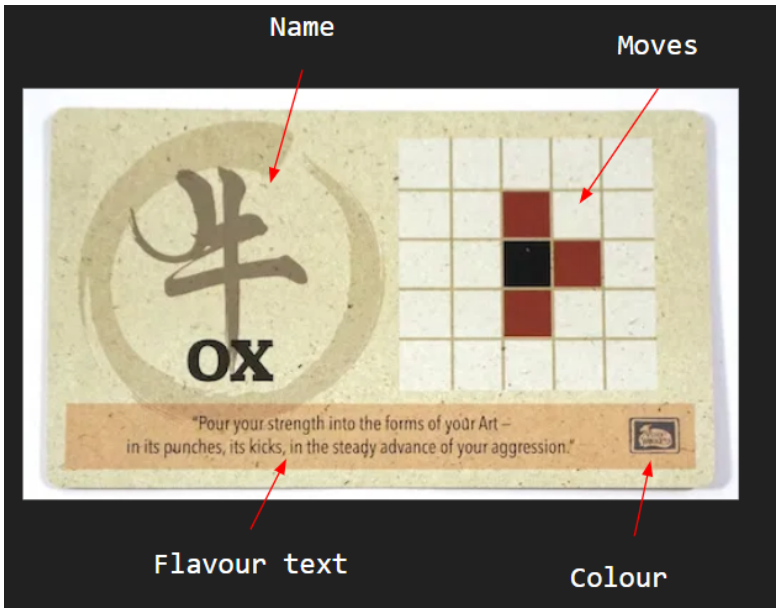


FIGURE A.3: Details of the 'Ox' card

The win conditions of Onitama are as follows:

- Conquer: Winning by conquer happens when the master piece of one player moves into the temple of the opponent (where the opponent's master starts). The master can move into the temple even when an opposite piece is defending the temple, in which case the opposite piece will be removed from the game.
- Destroy: A victory by destroy is achieved when one of the masters is killed by an opponent's piece. Any piece (pawn or master) can kill the master by moving to the square it currently is occupying. The player who killed his opponent's master piece wins the game by destroy.

The following describes the gameplay of Onitama:

1. At the start of the game, the board is set up as shown in the figure above (the masters on their respective temples, flanked by their pawns). Then, the deck is shuffled and five cards are drawn at random. Each player will receive two of these five cards, placed face up in front of each player, with the remaining card placed by the side of the board (as shown in the figure above). It is important that the cards be placed upright to its respective player, meaning that opponents will always see each other's card in an upside down manner. The player who will make the first move is determined by the colour of the side card, which must always be upright with respect to the current player (if it is the red player's turn, the side card must be upright with respect to him, and vice versa).
2. In a turn of gameplay, the player will choose a piece, one of his cards, and a move on that card. A piece cannot land outside the board or on another piece of the same colour. If a piece lands on another piece of a different colour, the piece it lands on is removed from the game. The card that the player selected to make the move is then placed on the side, and the player takes the side card and places it in front of him (to replace the card he just used). Then, it is the opponent's turn to move.
3. The players will keep alternating turns until a win condition is reached (explained above). In this game, a draw is not possible.

Appendix B

Worked example for minimax

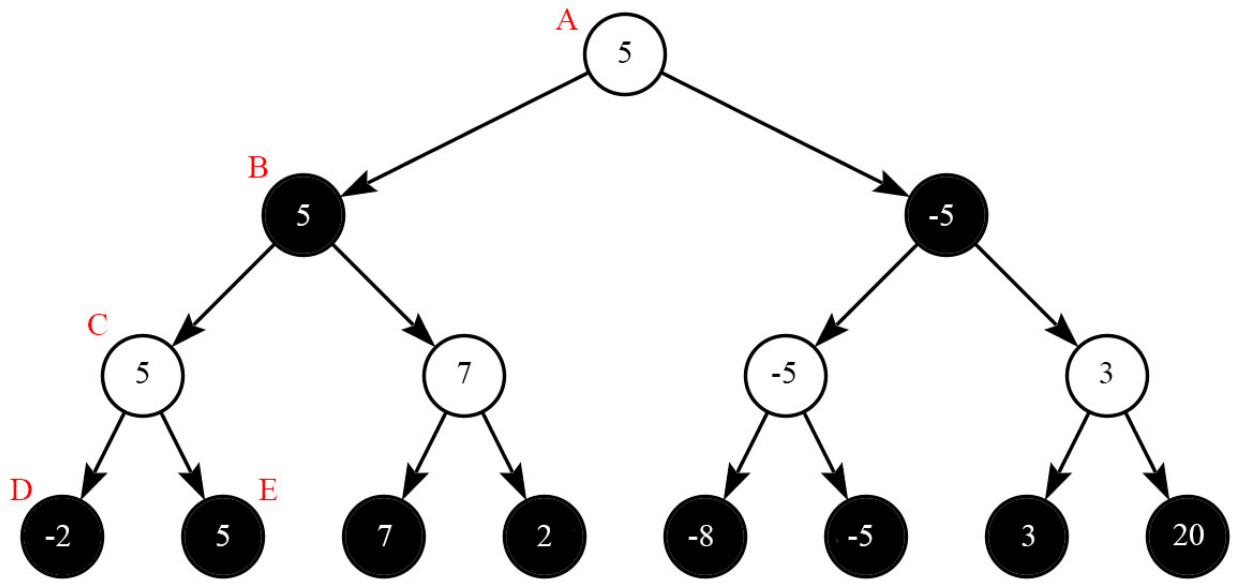


FIGURE B.1: Minimax tree diagram

A simplified example of minimax can be explained with in Figure B.1 shown above. The white circles denote the player (maximiser) and the black circles denote the opponent (minimiser). The numbers in the circle denotes the value of the choice taken, the more positive the value, the better for the player and vice versa. In this diagram, the minimax has a depth of 3. Starting from circle A, the player will start with the first move, the left branch. To calculate the value of this move, the minimax function is called again (recursively) but this time from the opponent's perspective, towards circle B. The opponent also chooses the left branch, circle C. This continues until the specified depth (depth = 3, circle D) or a terminal state (either the player or the opponent winning) is reached. At this state, a score is calculated with some predetermined method (in this research, as stated before the points are determined by the pieces still alive and the distance between the master and the opponent's temple).

In the diagram, a score of -2 is received from circle D, and hence it is passed up to the depth above it (circle C). However, from circle C, there is still one branch not explored (circle E). So minimax will once again be called to resolve this and circle E returned a score of 4. Now there are two choices of scores, -2 and 5. Since the player is the maximiser, he will choose the larger of those two, 5, hence why in circle C it is 5, not -2. The values will be passed up to circle B, which has a choice between 5 and 7. Since it is the opponent's (minimiser) choice, he will choose 5. In the end, this will result in circle A having the score of 5. This means that at the current point (circle A), the player's next move is to go to the left, since even if the opponent chooses the best moves, the player will receive at least 5 points.

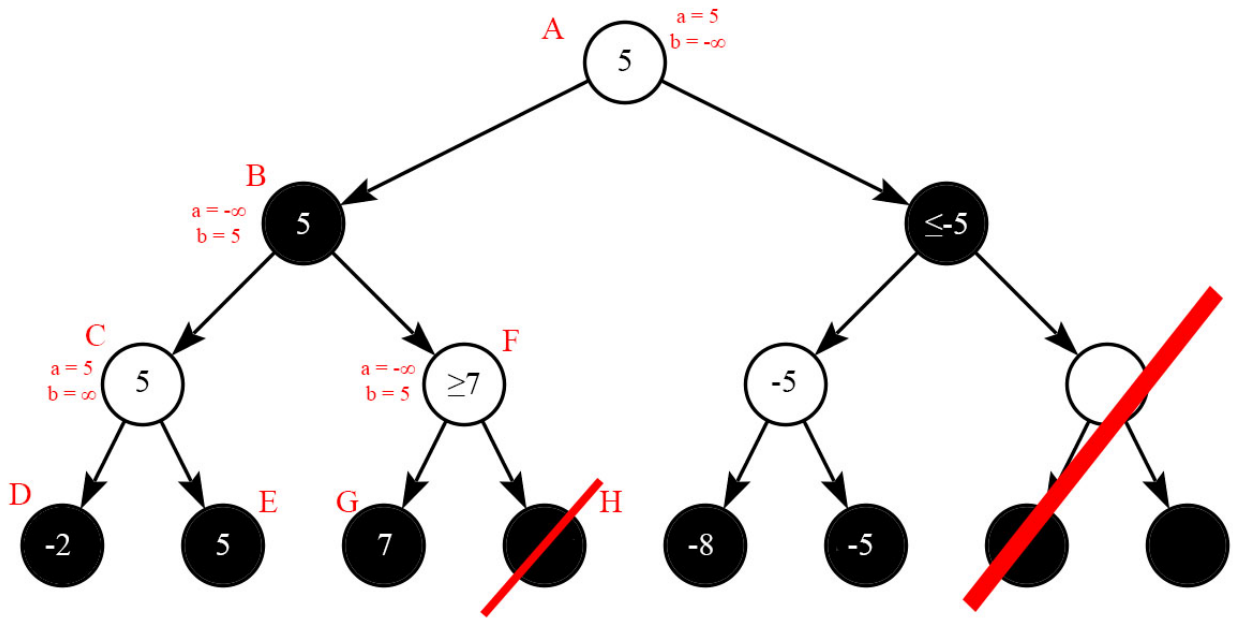


FIGURE B.2: Minimax tree diagram after alpha-beta pruning

For minimax with alpha-beta pruning, we can start in the same way as minimax without alpha-beta pruning. Pruning does not necessarily always happen in every branch, it only happens on two conditions: score $> \beta$ when maximising or score $< \alpha$ when minimising. So, in the figure above we go all the way to circle C, as explained before, and determine the score there from circles D and E, and we get a score of 5. However, now we compare that score with the value of a, initialized at $-\infty$. Since score $> \alpha$, now α = score = 5. Then, these values are passed up to circle B, and since we only have the score from circle C, 5, this is the best alternative for the minimising player, so $\beta = 5$ while the value of a is initialized back to $-\infty$. Then we pass down these values to circle F, which gets its score from circle G, so score = 7. Now, we see that score $> \beta$. This means that the minimising player has a better choice to make earlier on, since by going through the left-most branch it will get a score

of 5 instead of 7 from the current branch. At this point, we can prune circle H (its score does not need to be computed) because there are two scenarios. In the first scenario, the score for circle H is greater or equal to the score of circle F, which means that the minimising player will not choose this path, preferring the path through circle C since that path gives him a better score. In the second scenario, the score for circle H is smaller than the score of circle G, which means that the maximising player will always choose circle G instead of circle H. In any case, the score of circle H can be ignored and it can be pruned. The same process can be repeated to the right side of the diagram and this pruning-enhanced algorithm will give the same result as without pruning, but less computation is needed, saving time and computing power.