Min-Max versus Monte Carlo Tree Search for Quoridor

Zagesh Parshotam
School of Computer Science and Applied Mathematics
University of Witwatersrand
Johannesburg, South Africa
zagesh2000@gmail.com / 1845347@students.wits.ac.za

***Abstract*—There is great significance of artificial intelligence in game theory, optimizing decision-making processes and tactics in competitive environments. This paper focuses on two widely used algorithms, Min-Max and Monte Carlo Tree Search (MCTS). The objective of this paper is to compare and analyse the strengths and limitations of Min-Max and MCTS, assessing their performance and time complexity using the strategy board game Quoridor. By providing valuable insights into these algorithms, the study aims to contribute to the advancement of game theory and potentially facilitate their application in real-world scenarios.**

***Index Terms*—Min-Max, Monte Carlo Tree Search, Quoridor, AI agents**

I. Introduction

In order to comprehend decision-making processes and optimize tactics in competitive situations, game theory has become a crucial aspect. With these developments in game theory comes the development of artificially intelligent algorithms that make those decision processes easier. Two of these algorithms, Min-Max and Monte Carlo Tree Search (MCTS), have become very popular choices and have shown to be very successful.

Min-Max has been used effectively for classical games such as *chess, checkers* and *tic-tac-toe* [1]. It is a classical algorithm and has been influential in the development of game theory. Many advancements over the years have contributed to this algorithm to make it more efficient and more effective. Some of these innovations are discussed in [2] and [3]. Min-Max will always be an effective tool in building Artificial Intelligence for strategy board games.

MCTS is a slightly more modern approach to searching game trees. It has been shown to be very successful for complex games such as *Go* and *Connect-four* [4]. The game tree is built dynamically in this approach as it runs simulations of the games to help choose the optimal outcome. It is a very useful algorithm when it comes to games with a large search space. For this reason, it has shown to be beneficial in solving real-world problems.

In this paper we compared and analysed the features and downfalls of Min-Max and MCTS. We tested their performance against each other. This was done using a two-player strategy board game called *Quoridor*. The two algorithms competed against each other until a winner was found. Thus, determining the superior approach. In conjunction with the results of the matches, the time taken for each algorithm to select a move will also be investigated.

The purpose of this paper is to provide valuable insight into these two algorithms and therefore contribute to game theory developments. It is shown in this paper that Min-Max is superior to MCTS in *Quoridor*. These developments may perhaps also be useful in using these algorithms for real world situations.

Section II aims to provide background knowledge for this paper. We explain the game of Quoridor and its rules. We then discuss the concepts of both Min-Max and MCTS. In Section III we discuss how these algorithms have been used previously for similar types of games. In Section IV we explain how our algorithms were implemented. Section V discusses how we implemented the experiment. The results of these experiments are discussed in Section VI. Finally in Section VII, we will discuss what the results mean to the current field and how we can further develop this work to obtain more conclusive results.

II. Background

In this chapter, we present the different techniques used to create an artificially intelligent tree search algorithm and the background knowledge needed to build them. Firstly, in Section II-A, we provide a basic explanation on how "Quoridor" works and all its rules. Section II-B discusses Min-Max algorithm, and its subsequent subsection discusses a variation. That being Alpha-Beta Pruning in Section II-B1. Next, Section II-C we discuss Monte Carlo Tree Search and its implementation.

A. Quoridor Rules

Quoridor [5] is a two/four player strategy board game. However, for this paper we are only interested in the two player version. The following section explains the rules and terminology as described by [6].

Quoridor is played on a board made of 81 squares (9x9). A player is represented by a pawn which is placed on the centre space on said player's side of the board. The two players play from opposite sides as seen in Figure 1. The goal of the game

is to get your pawn to the opposite side of the board. Players make moves one after the other.

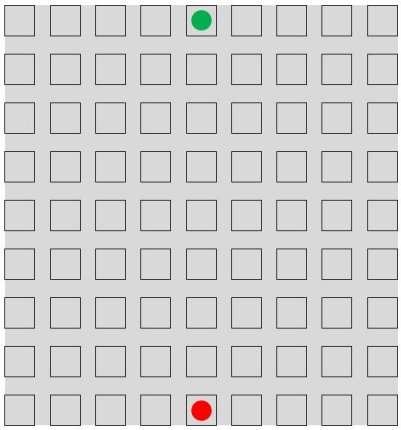

Fig. 1. Initial Quoridor setup

The distinguishing feature of the game are the walls. Walls are two square-spaces wide pieces that can be placed in the space between the square spaces. The walls block the path of the pawns, and the pawns must go around them. Each player receives 10 walls.

On a player's turn, they may either choose to place a wall or to move their pawn. The pawn may only move up, down, left or right. If the pawns face each other on adjacent squares, not separated by a fence, the player whose turn is it can jump over the opponent's pawn therefore advancing an extra square. If that square is not available (off the board or blocked by a wall), the player may play their pawn adjacent to the opponents. Walls may never be jumped. As shown in Figure 2.

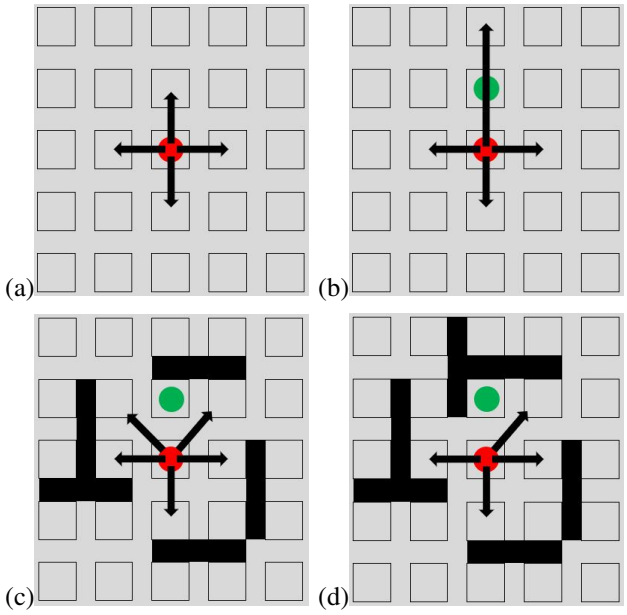

Fig. 2. Moves allowed for the red pawn

Walls may be placed between two spaces not already occupied by a wall. A wall may not be placed in any space that may block the only remaining path of any pawn to its opposite side as seen in Figure 3.

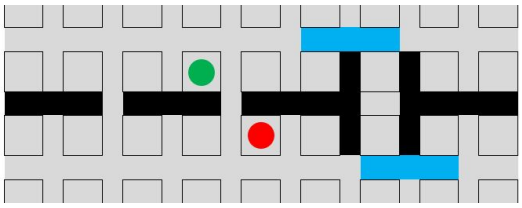

Fig. 3. Blue walls indicate illegal moves.

The first player to reach one of the nine squares on their opponent's side is the winner.

B. Min-Max Tree Search

Min-Max is most commonly used for two-person perfect information games between players which we call Min and Max [7, Chapter 5]. The idea is for the player to choose the optimal move for themselves at each step. The game starts with Max making the first move after which they alternate moves.

As explained by [2], the game defines a tree of moves that could be made for any two player perfect information game. The tree is split into two subsets of moves called *Min* and *Max* depending on which player's turn it is. For each move a set of further moves which are all the children of the parent move. The player must then select from this children set to make a move. The opponent will then select a move from the children of the newly selected move. Moves with no children are called terminal moves and are the leaves. The game ends if one of these leaves are reached.

Each leaf has a score associated with it. The value being the benefit of the move in *Max's* perspective [3] [8]. By recursion, we determine the score of a move by moving up or "minimaxing" the value of the leaf.

If it is *Max's* turn, the player will choose the maximum score and Min will choose the minimum score when it is their turn. Figure 4 from [2] shows an example game tree where square nodes represent *Max* moves and circle nodes represent *Min* moves. The bold arrows show the optimal path.

On smaller trees, Min-Max is optimal. However, variations of Min-Max are used to optimise game play. We will now discuss a variation.

1) Alpha-Beta Pruning: As discussed in [3], alpha-beta reduces the cost of searching a tree by pruning sub-trees that do not affect the final value of the root. The basic idea being that when we know the current score will not be affected by a child node, we prune that branch. Alpha (α) being the best value that Max can guarantee and Beta (β) being the best value that Min can guarantee [7, Chapter 5].

[9] explains it as follows. Two players are defined F and G, where F tries to maximise and G tries to minimise. The position on the tree is defined by p and its subsequent positions are defined by $(p_1, \ldots, p_d)$, where $d > 1$. The value of the move is represented by $f(p)$. The best value that F can achieve is $F(p)$ and $G(p)$ for G.

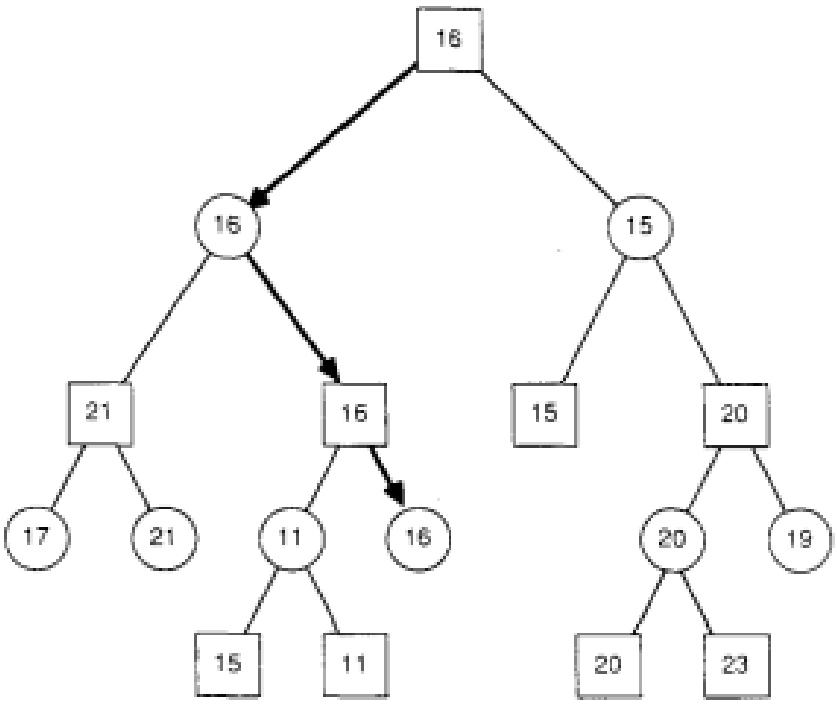

Fig. 4. A small game tree.

$$F(p) = \begin{cases} f(p) & \text{if } d = 0, \\ max(G(p_1), \ldots, G(p_d)) & \text{if } d > 0 \end{cases}$$

$$G(p) = \begin{cases} g(p) & \text{if } d = 0, \\ max(F(p_1), \ldots, F(p_d)) & \text{if } d > 0 \end{cases}$$

It can be proven by induction that $G(p) = -F(p)$.

[9] defines the algorithm as follows. The algorithm being called F2.

Algorithm 1 Alpha-Beta Pruning Algorithm

```
1:  function F2(position p, α, β)
2:      m, i, d, t
3:      determine successor states p_1, ..., p_d
4:      if d = 0 then
5:          return f(p)
6:      else
7:          m ← α
8:          for i = 1 until d do
9:              t ← −F2(p_i, −β, −m)
10:             if t > m then
11:                 m ← t
12:             end if
13:             if m ≥ β then
14:                 return m
15:             end if
16:         end for
17:         return m
18:     end if
19: end function
```

C. Monte Carlo Tree Search

The Monte Carlo Tree Search (MCTS) is different in that we build the tree as we iterate. The main idea is that the initial tree is empty and it begins from the root. The tree is expanded and a child is selected. If that node is not a leaf node, expand the tree until one is reached. If a leaf node is found, simulate the rest of the game and get a score. The score

is then backpropagated up the tree to the visited nodes all the way to the root. [10] [11]

There are four phases of MCTS: Selection, Expansion, Simulation and Backpropagation. Each node contains 2 values: the average of the results of previous simulations that have visited this node and the total number of visits. A common strategy used to to determine this average of results is called UCT which stands for Upper Confidence bound applied to Trees. The idea being to choose the node i which gets the highest UCT [12].

$$v_i + C\sqrt{\frac{\ln N}{n_i}}$$

v_i is the current value of the node. n_i is how many times said node has been visited. N is how many times the parent node has been visited. C is a coefficient that can be selected or set experimentally. C may be adjusted as to either explore more or to exploit more. The higher C is the more we try to expand new nodes. [13]

Selection is the phase where one of the children of the current node is chosen. The main idea being to select the node that finds the best result. We choose the move that will maximise the UCT.

The Expansion phase where we decide if a leaf node will be expanded. The children of this node are stored. The first node that has not been evaluated is the first node stored.

The Simulation phase plays random moves until the end of a game is reached. Random moves may not be optimal and a strategy for this phase might prove advantageous.

Back-propagation phase is where the result of the simulation phase is taken back up the tree. The score is moved up to the root and every node's new v_i is calculated.

The move selected is the child of the root with the greatest number of visits.

III. Related Work

In this section we discuss how these algorithms have been implemented in past works.

A. Min-Max Tree Search

This is a commonly used algorithm and has been used extensively to search game trees for two player strategy games.

In the case of [1], Min-Max was used to for *tic-tac-toe*. The Min-Max model on a serial machine was compared to using the model on a parallel computing machine. The idea being to different processors to search branches of the tree in parallel. The paper concluded that the parallel method was more efficient.

[2] used an alternate version of Min-Max where the mean values between the minimum and maximum values were used to make approximations. This algorithm was used on games of *Connect-four* showing encouraging results.

B. Alpha-Beta Pruning

Alpha-beta algorithm was used on games of *Checkers* by [14]. This paper investigated using this algorithm with iterative deepening. Iterative deepening is commonly in conjunction with *Depth-first search*. The idea being to run the algorithm repeatedly while increasing the depth limit of the search.

C. Monte Carlo Tree Search

MCTS has been used frequently in more recent times and to very promising results. [13] and [6] used this algorithm for *Backgammon* and *Quoridor* respectively. In [13], the results show that the algorithm works but it was not the most efficient due to the chance factor in *Backgammon*. However, [6] used the algorithm efficiently to create several different models, each doing different tasks in the game.

D. Combining Min-Max Tree Search and Monte Carlo Tree Search

Combining these two algorithms has been used to great effect in increasing efficiency when searching game trees.

[4] used three different methods on games of *Connect-four* and *Breakthrough*. The first being using Min-Max in the simulation phase, the second in the selection/expansion phases and the third in the backpropagation phase. The results were mixed as different methods worked better for different games. The paper concluded that the best method depended on how the game tree was built.

[15] used the combination on games with chance factors. The games being *Pig, EinStein Würfelt Nicht!, Can't Stop,* and *Ra*. A variation of Min-Max was used for chance nodes.

IV. Methodology

We aim to compare two algorithms, Min-Max and Monte Carlo Tree Search (MCTS). This will be done on the board game Quoridor.

A. Min-Max

First we need to be able to represent the game state. For this we create a *Board* class. In this class we represent the board as a dictionary where every square on the board is represented by a *key* and all the squares it is connected to are its *value pairs*. We have created an undirected graph with each square connected to its neighbours. This class also holds the side to play the next move, the positions of both players as well as the number of fences they have remaining.

Each square is represented as a coordinate. First its column represented by an alphabet [A to I] and then its row represented by number [1 to 9].

We then have to remove the connections on the graph that are blocked by a fence. This is shown in Figure 5

We have two types of moves a player can make, either a possible *piece move* or a possible *fence move*. These are saved in a list called *possible moves*.

For the possible *piece move*, we use the square of the player to move and find all the squares that it is connected to. If the

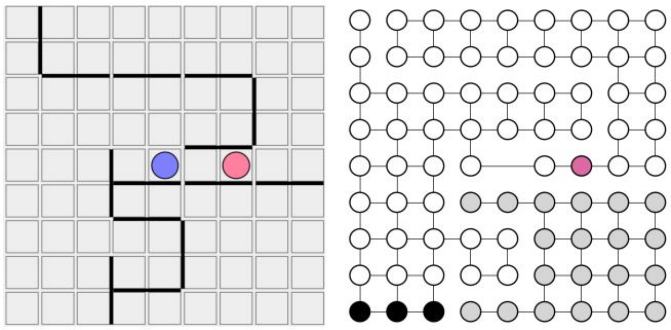

Fig. 5. Undirected graph representation of board

opposition player is next to the player we have to remove that square and add the surrounding squares we can move to.

For possible *fence move*, we find all the possible fence positions that do not currently have fences and do not overlap other fences. Further adjustments are done so that possible fences may not block either players route to the goal completely. A path to goal for both players needs to always be possible.

We need to be able to execute these moves and make changes the game state. For a *piece move*, we change the position of the player that made the move and we update the side to play to the opponent. For a *fence move*, we first add the fence to the board so that the new board has a disconnect at the relevant squares. We then update the number of fences remaining of the player that made the move and we update the side to play to the opponent.

Now that we have a system that can play the game, we can add the agent. The Min-Max agent agent makes use of Alpha-Beta pruning to select the best possible move. The function is built as follows:

Algorithm 2 Min-Max Algorithm

```
1:  function MINMAX(board, fences, depth, α, β)
2:      if isGameOver(board) or depth ≤ 0 then
3:          return evaluate(board)
4:      end if
5:      moves = possibleMoves(board, fences)
6:      for move in moves do
7:          board, fences = makeMove(move, board, fences)
8:          score = -MINMAX(board, fences, depth-1, −β, −α)
9:          if score ≥ β then
10:             return β
11:         end if
12:         if score > α then
13:             α = score
14:         end if
15:     end for
16:     return α
17: end function
```

In the *isGameOver* function we check if the game has reached a conclusion.

In the *evaluate* function, we determine the *raw score*. If player one has reached hes goal, the *raw score* is -10000. If player two has reached hes goal, the *raw score* is 10000. If no player is at their goal, the *raw score* is the steps to the goal for player one minus the steps to the goal for player two. This is done using Breadth-First Search. The *raw score* is from player two's point of view and needs to therefore be multiplied by -1 if it is player one's turn to play.

In line 8 we get the score from the min player's view. If this score is greater than beta, we prune the remaining branches of the node and return beta. If the score is greater than alpha, than alpha gets updated.

The returned score is the max score that the player can receive.

We select the move that has the highest score. This done by running the following algorithm.

Algorithm 3 Min-Max Algorithm

```
1:  function GET BEST MOVE(board, fences, depth, α, β)
2:      best move = None
3:      value = any large negative number
4:      moves = possibleMoves(board, fences)
5:      for move in moves do
6:          board, fences = makeMove(move, board, fences)
7:          score = -MINMAX(board, fences, depth-1, −β, −α)
8:          if score > value then
9:              value = score
10:             best move = move
11:         end if
12:     end for
13:     return best move
14: end function
```

Each move is executed in this function and the score of the resulting board is found. The move with the highest score is deemed the best move and this is the output of the algorithm.

B. Monte Carlo Tree Search

The class *Monte Carlo Tree Search (MCTS)*, holds the search tree and is used to determine where to expand and where to exploit. Each node is represented by the board state, the current fences, the side to play, the player positions, the fences remaining for each player, the parent node, the action the parent took to get to this node, the children, how many times the node ha been visited, the results of these visits, and all the possible actions that have yet to be expanded.

In this class, there is a *best action* function. This function carries out the four phases of MCTS.

Algorithm 4 MCTS Algorithm

```
1: function BEST ACTION(self)
2:     choose numberOfSimulations
3:     for i from 0 to numberOfSimulations do
4:         node = selectNode(self)
5:         reward = node.rollout(self)
6:         node.backpropogate(reward)
7:     end for
8:     return self.bestChild(C = 0.1)
9: end function
```

We first choose the number of simulations to run. Next, in line four, we either expand the current node if it is not fully expanded or we select the child with the highest UCT (as discussed in Section II-C) score to expand. We use a $C = 2$ as to expand more than we exploit.

In line 5, we get the reward of the chosen node by the simulation/roll-out phase. The roll-out policy selects moves at random until the game was completed or a maximum number of moves has been reached.

In line 6, this reward is then backpropagated up the tree to all the nodes visited along the path to the root.

Once the simulations are completed, we choose the root node's best child, giving us the optimal move.

As in Min-Max, each node's state is represented by a dictionary where every square on the board is represented by a *key* and all the squares it is connected to are its *value pairs*. If there is a fence blocking two squares the connections will be removed in the dictionary.

V. Experiments

To run each agent we need to create a game string to represent the game. This is represented as follows:

<player1 position/player2 position> <fence positions> <player to move> <fences remaining1/fences remaining2>

As shown in these examples:

E1/E9 None 1 10/10
E1/E9 E3h/G6v/D8v 1 8/9

We then use a *Game Manager* (GM) to send game strings as input to the agents and receive their selected moves. The GM also has a board representation similar to the agents so that it can keep track of game states. The GM made use a the *subprocess* library in python to run the agents. Input and output was sent and received using this process

The GM firsts passes a game string to an agent. The GM then receives the selected move by the agent as well as the time taken to execute this move. The GM executes received moves on the game board and passes the new game string to the next agent. This continues until the game reaches a conclusion and a winner is found.

The GM saves the game-play from start to finish as well as the time taken for the agents to execute moves in text files.

```
GM -> MCTS: H1/E6 None 1 10/10
MCTS.py -> I1
GM -> MinMax: I1/E6 None 2 10/10
MinMax.py -> H1h
GM -> MCTS: I1/E6 H1h 1 10/9
MCTS.py -> H1
GM -> MinMax: H1/E6 H1h 2 10/9
MinMax.py -> F1h
GM -> MCTS: H1/E6 H1h/F1h 1 10/8
MCTS.py -> I1
GM -> MinMax: I1/E6 H1h/F1h 2 10/8
MinMax.py -> E5
GM -> MCTS: I1/E5 H1h/F1h 1 10/8
MCTS.py -> H1
GM -> MinMax: H1/E5 H1h/F1h 2 10/8
MinMax.py -> E4
GM -> MCTS: H1/E4 H1h/F1h 1 10/8
MCTS.py -> I1
GM -> MinMax: I1/E4 H1h/F1h 2 10/8
MinMax.py -> E3
GM -> MCTS: I1/E3 H1h/F1h 1 10/8
MCTS.py -> H1
GM -> MinMax: H1/E3 H1h/F1h 2 10/8
MinMax.py -> E2
GM -> MCTS: H1/E2 H1h/F1h 1 10/8
MCTS.py -> I1
GM -> MinMax: I1/E2 H1h/F1h 2 10/8
MinMax.py -> E1
MinMax wins
```

Fig. 6. Example game simulation

Six separate game simulations were run. These were run with Min-Max searching at a depth of two, while MCTS ran a simulation number that was two times the number of the root node's possible actions.

The resulting winner of each game simulation is saved to show the superior agent. The times the agents took to execute each move in the game simulation is also saved. This will be able to tell us the total time an agent took to play the game as well as the average time an agent took to make each move. We use this to determine which agent has the better time complexity as well as time complexity of selecting a move at different points in the game.

A snippet of an example game play simulation is shown in Figure 6.

For this simulation Min-Max searched to a depth of 1 while MCTS ran 2 simulations.

VI. Results

In our results we aim to show evidence of one of the algorithms being superior to the other.

As mentioned previously, we ran six game simulations and collected the game results and the times taken for each move.

A. Game Winners

In Table 1 we have the resulting winner of each game and the number of moves each agent had to play.

This clearly shows that Min-Max is the superior agent. It has outperformed MCTS in every game while also using a minimum number of moves to achieve these results.

Game Number	Winner	Number of moves
1	Min-Max	14
2	Min-Max	15
3	Min-Max	18
4	Min-Max	14
5	Min-Max	20
6	Min-Max	16

TABLE I
GAME SIMULATION WINNERS

Min-Max searched to a depth of two only and still managed to choose the optimal moves. The use of Alpha-Beta pruning proved successful in removing less than optimal moves.

MCTS performing so poorly could be the result of having too few game simulations. This means that not enough nodes have been expanded and as a result the best move may not have even been explored. Secondly, the roll-out policy chose moves at random. This may have not been optimal. Since we have limited the number of moves the agent makes in the roll-out phase, not every node is getting the correct reward. This could have also hindered performance. Lastly, our UCT function may not have been optimal for *Quoridor*. We used a C-parameter or 2 thus preferring to expand rather than to exploit.

B. Time Complexity

In Figure 7 we show the total time taken to select moves by each agent in each specific game. It can clearly be seen than MCTS took much longer to select moves in each game.

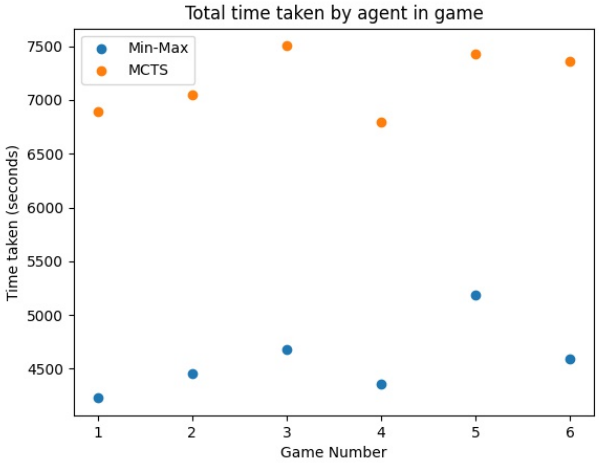

Fig. 7. Total time taken in Each game

From this we can see that Min-Max has a much better time complexity for selecting moves in *Quoridor*. This could be due to the fact that Alpha-Beta pruning removed inferior moves and thus the agent did not need to explore these.

MCTS taking much longer to choose a move could be due to there being too many simulations run. *Quoridor* being a game that has a tree with a large branching factor means that the agent has many nodes to expand and explore. This took much longer than we expected.

In Figure 8, we look at the average time taken for the agents to choose a move at different points in the game. This shows us how long the agents took to choose each move. Since each game took a different number of moves to complete, we selected the times of the first 14 moves. Figure 8 shows that MCTS took longer than Min-Max to choose moves.

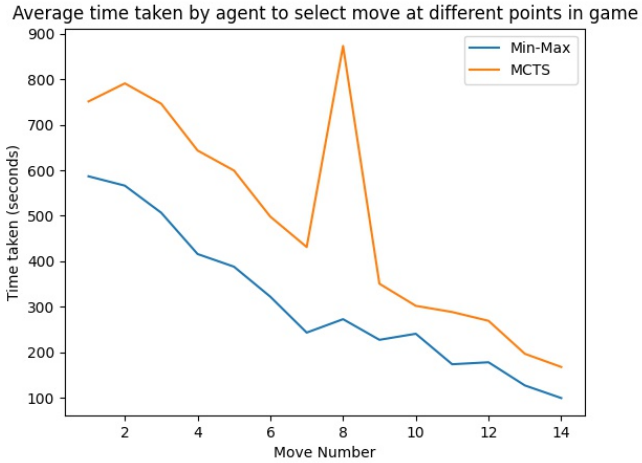

Fig. 8. Average time per move

Both agents are shown to take much longer to choose a move at the start of the game than towards the end. This is down to there being fewer available moves to the agent as the game progresses.

Min-Max, once again, proved to be superior as it took less time to choose a move at all points in the game.

MCTS has to explore every possible move that can be made while Min-Max is able to prune the inferior nodes. At all points in the game this was the case.

However, MCTS has shown that it reduces its time to choose a move much better than Min-Max as the game progresses. This is shown by the slightly steeper slope of MCTS in Figure 8. This is because, as the game progresses, MCTS has fewer moves to make in the roll-out phase. Therefore, reducing the time much further.

Move number 8 is the anomaly in this assessment. This is due to MCTS taking extremely long to choose a move in one of the games.

A further evaluation that could have been made would have been to assess the number of nodes in each agent's game tree. This will tell us about the different space complexities of the agents and which one is better.

The unexpected results from MCTS is similar to the results seen by [13]. In both implementations were not ideal due to the random selection of nodes in the roll-out phase.

The helpfulness of Alpha-Beta pruning to Min-Max as explained by [3] [7, Chapter 5] proved to be extremely useful.

Code for both the agents and the GM can also be found at: https://github.com/Zagesh/MinMax-vs-MCTS.

VII. Conclusion

Artificial intelligence has proved to be very important to game theory. The development of AI algorithms has vastly improved the way we think about game theory and how we approach it. Min-Max and Monte Carlo Tree Search are very important to this idea. Both algorithms have been proven successful for various games as well as in real life situations. They have been thoroughly investigated and many significant improvements have been made over the years.

This paper has shown that Min-Max is superior to MCTS in games like *Quoridor*. Min-Max performs much better at evaluating the best possible moves to make at all points of the game. It has also been shown to have a much better time complexity than MCTS.

What this paper shows is that even though Min-Max has previously been found to be inferior for games with large search spaces, it still may be useful when many of the branches are irrelevant to the current game space. This result may be used in the research of other board games that are similar to the type of search tree built by *Quoridor*.

Future work will include trying to get MCTS to expand and exploit more efficiently. This could be done with further testing the UCT function and adjusting it to make exploitation more valuable.

Using a heuristic in the evaluation function of Min-Max and in the roll-out policy of MCTS could be found useful. As explained in [16], this could include using the expected outcomes of a game as a useful heuristic.

References

[1] P. Borovska and M. Lazarova, "Efficiency of parallel minimax algorithm for game tree search," in *Proceedings of the 2007 international conference on Computer systems and technologies*, pp. 1–6, 2007.

[2] R. L. Rivest, "Game tree searching by min/max approximation," *Artificial Intelligence*, vol. 34, no. 1, pp. 77–96, 1987.

[3] M. S. Campbell and T. A. Marsland, "A comparison of minimax tree search algorithms," *Artificial Intelligence*, vol. 20, no. 4, pp. 347–367, 1983.

[4] H. Baier and M. H. Winands, "Monte-carlo tree search and minimax hybrids," in *2013 IEEE Conference on Computational Inteligence in Games (CIG)*, pp. 1–8, IEEE, 2013.

[5] M. Marchesi, "Quoridor. family games," 1997.

[6] V. M. Respall, J. A. Brown, and H. Aslam, "Monte carlo tree search for quoridor," in *19th International Conference on Intelligent Games and Simulation, GAME-ON*, pp. 5–9, 2018.

[7] S. J. Russell and P. Norvig, *Artificial intelligence a modern approach*. London, 2010.

[8] G. C. Stockman, "A minimax algorithm better than alpha-beta?," *Artificial Intelligence*, vol. 12, no. 2, pp. 179–196, 1979.

[9] D. E. Knuth and R. W. Moore, "An analysis of alpha-beta pruning," *Artificial intelligence*, vol. 6, no. 4, pp. 293–326, 1975.

[10] S. Samothrakis, D. Robles, and S. Lucas, "Fast approximate max-n monte carlo tree search for ms pac-man," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 3, no. 2, pp. 142–154, 2011.

[11] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton, "A survey of monte carlo tree search methods," *IEEE Transactions on Computational Intelligence and AI in games*, vol. 4, no. 1, pp. 1–43, 2012.

[12] L. Kocsis and C. Szepesvári, "Bandit based monte-carlo planning," in *European conference on machine learning*, pp. 282–293, Springer, 2006.

[13] F. Van Lishout, G. Chaslot, and J. W. Uiterwijk, "Monte-carlo tree search in backgammon," in *Computer Games Workshop*, 2007.

[14] Z. Zhao, S. Wu, J. Liang, F. Lv, and C. Yu, "The game method of checkers based on alpha-beta search strategy with iterative deepening," in *The 26th Chinese Control and Decision Conference (2014 CCDC)*, pp. 3371–3374, IEEE, 2014.

[15] M. Lanctot, A. Saffidine, J. Veness, C. Archibald, and M. H. Winands, "Monte carlo*-minimax search," *arXiv preprint arXiv:1304.6057*, 2013.

[16] B. Abramson and R. E. Korf, "A model of two-player evaluation functions.," in *AAAI*, pp. 90–94, 1987.