UNIVERSITY OF HERTFORDSHIRE

School of Computer Science

MSc in Artificial Intelligence with Robotics

Artificial Intelligence with Robotics Master Project

7COM1036

13/01/2020

Heuristic Analysis and Comparison of the Game Searching Algorithms

Vural Erdogan

16082775

Supervisor: Prof. Daniel Polani

Table of Contents

Chapter 1 Introduction

1. Introduction

As a human, we always would like to reduce uncertainty as much as possible in our lives. This instinct triggers us to explore undiscovered sources such as a planet or a Wikipedia page. Even on social media, we mostly like to have full information of the people's lives that are usually interacted with us closely; therefore, we try to have more knowledge about them as much as we can, that is also called "stalking" nowadays. The limited information about a source is always an obstacle for people.

The uncertainty problem is also discussed systematically in many fields. For example, in economics, statistics, engineering, decision theory and information science, it plays a significant role. Especially in decision theory, that considers the utility of all possible outcomes and recommends the course of action that optimizes the expected utility, the area of choice under uncertainty represents the heart of the theory. In this project, I will be discussing "heuristic strategies" for the games, which play a huge role to make judgements under uncertainty.

Heuristics are problem-solving strategies that are usually used to solve complex problems in a quick way with the given time constraints and under uncertainty. In real life, humans, animals or machines apply heuristic methods so often. When we drive our cars, if an accident or threat is recognised, we don't actually calculate how much we will press the brake pedal. However, under time pressure, we apply a heuristic strategy to solve that problem intuitively. The heuristic solutions may not be necessarily optimal or best but should bring to the table good-enough solutions.

In this project, for heuristic analysis, *Minimax with Depth-cut* and *Monte Carlo Tree Search* algorithms have been decided to compare as they explore the possible moves vertical and horizontal way [1]. Minimax and Monte Carlo Tree Search are popular decision-making algorithms that can be formulated for two-player zero-sum games. Minimax algorithm, also referred to as "maximin", finds the optimal move for a player supposes that your opponent also plays optimally. Maximiser and minimiser are the two players in this algorithm; maximiser plays to get the highest score possible while the minimiser tries to get the lowest score possible. While exploring the possible moves, the board state returns with a value that shows how much the move does worth to play. On the other hand, the Monte Carlo Tree Search follows a statistic way to decide next move. Gaining popularity of Monte Carlo Tree Search after Alpha-Go Zero's achievement against Lee' Sedol increased research areas on this topic [2]. This project compares and analyses heuristics of these two algorithms for the Pawn Game that is a discrete two players zero-sum game similar to chess game but played with only pawns.

Before comparing the heuristics, the pawn game has been coded in Python and tested. First, the minimax algorithm applied and tested with a unit test. After, alpha-beta pruning and depth-cut methods implemented with a simple heuristic, and whether a more complex heuristic method is necessary or not discussed. Further, the Monte Carlo Tree Search coded for the experiments. The increasing number of rollout's effects have been observed and compared if the algorithm plays more intelligently or not. Finally, the heuristic experiments conducted between two players selected among these algorithms; Monte Carlo Tree Search with UCB1, depth-cut with simplified heuristic, depth-cut with enhanced heuristic and random move creator.

Throughout the thesis, these questions tried to be answered:

· Can minimax and alpha-beta applied for the Pawn Game to compute entire decision trees for winning move?

· Can we add heuristic search to alpha-beta pruning method?

· Does board size influence the players' scores?

· Is a player's turn important on the Pawn Game?

· How the heuristic is effective on the scores against a random player or other action recommender players? And can we enhance it?

· Can MCTS successfully be applied to the Pawn Game? How the rollout is effective on the scores against a random player or other action recommender players?

· Is MCTS better player than Minimax with depth-cut on the Pawn Game?

Chapter 2
Background

2. Background

The minimax theorem was proved by John von Neumann in 1928 [3]. Before that, according to some sources, the origin of the algorithm dates back to the 1700s that discussed between Francis Waldegrave and his uncles for the card game le Her [4]. With this theorem, a significant theory of the economy, Game Theory, has been initialised.

Minimax Theorem has been elaborated by not only John von Neumann but also many other scientists such as J. Ville, A. Wald, S. Karlin, H. Kneser, K. Fan [5]. As it is one of the key theorems of Game Theory, it has a substantial role in economic, political, military and other social conflicts. Even the opponents or enemies change their strategy, they can't improve their gains due to this specific case of Nash equilibrium [6].

The Minimax algorithm has been designed as a finite-horizon search for zero-sum games, in which the sum of the payoffs for each outcome is zero, by Claude Shannon in 1950 published a paper entitled "Programming a Computer for a Playing Chess" [7] using John von Neumann's mathematical formulas. The main idea of this algorithm was very similar to the theorem that a player considers all of the best opponent responses to its own strategy and chooses the best of the best possible outcomes.

However, Shannon realised that computing the entire decision tree is almost impossible considering his own century's technology. Therefore, in the paper, he suggested that the searching algorithm should explore until a depth level then it should stop. After stopping, the evaluation function should estimate positional factors and decide which move is worth to play for winning by considering the dominance of the teams. In this way, the first time, the heuristic function (depth-cut strategy) has been implemented in theory. For complex game trees, this algorithm reduced the computation problem tremendously.

Another crucial implementation for reducing computation power was that; Alpha-Beta pruning which has enhanced Minimax's decision speed significantly eliminating branches of the search tree. With this strategy, the depth-cut algorithm could search deeper layers for the next move. It was first mentioned in a proposal by J. McCarthy in 1958; however, according to Allen Newell and Hertbert A.simon, alpha-beta has been reinvented several times [8]. Independently invention of the algorithm in the United States by Michael Levin, Richards, Timotyh Hart is a supportive argument for that alpha-beta or similar ideas proposed in the past by different people on different times [9].

Further, NegaMax algorithm has been explored by D. Knuth and R. Moore in 1975 [10]. The algorithm is a variant of Minimax but instead of applying min and max equations, it only operates max function. In other words, while evaluating opponent moves, instead of finding the opponents' best moves (min player) separately, the algorithm assumes that the opponent is itself (max player), therefore, it only maximises the gain on each step.

Alpha-Beta pruning is another way to improve NegaMax in similar manner Minimax. In this way, memoizing the values are gaining significant using the hash tables. As a result, Dennis Breuker, in his thesis, proposes transposition tables [11]. Essentially, a given board position can be observed more than once during playing. In order to skip duplicate nodes or paths, he suggests transposition tables that saves previously computed values of the nodes. However, if the depth constraints are applied, that brings some other issues as the nodes actually don't have their real values since heuristic is applied.

Another method is iterative deepening searching strategy that has dynamic move ordering technique depending on heuristic history considering allocated time [12]. If the time is exhausted for while searching for the move, the algorithm always has the option to go back using the last iteration's decision. In this way, a partial search can be accepted for the results. The algorithm can also be modified with transposition tables or alpha-beta method. The success of this algorithm is as efficient as alpha-beta and its enhancements surprisingly.

Moreover, Alexnder Reinefeld proposed "NegaScout" algorithm at the beginning of 1983s in his books by providing proof of correctness [13] by improving Judea Pearl's Scout [14]. This directional searching algorithm can work faster than alpha-beta pruning as it relies on a strategy that adjusts nodes in an accurate order to capitalise on its advantage. The algorithm assumes that the previously explored node is better than future ones and uses a scout window to explore moves considering alpha and beta values. If the moves are ordered randomly, that can affect the algorithm's efficiency negatively, and the algorithm might work worse than alpha-beta.

Another alternative to Alpha-Beta method, SSS* (state-space search) algorithm has been developed by George C. Stockman in 1979 [15]. The algorithm follows best-first fashion. Instead of searching in a dept-first way, it explores the most promising nodes first. Additionally, Stockman proved that SSS* would never evaluate more leaf nodes than alpha-beta. According to other experimental results, SSS* has shown success over alpha-beta but did not beat NegaScout [16]. However, storing and sorting the saved values is a problem and limited in this algorithm due to best-first search nature that makes algorithm slower and memory inefficient. On the other hand, the study [16] showed that null-window alpha-beta's sequence visits the same nodes in the same order just as SSS*'s when alpha-beta is endorsed with transition tables.

Another searching algorithm is that Principal Variance Search (PVS) has been introduced by T. Marsland and M. Campbell in 1983 [17]. Both algorithms have been introduced independently about the same years. PVS works similar to NegaScout but the difference is the way how they operate re-searches. While alpha-beta values are passed directly in the PSV method, in the Negascout, the scores that returned by the scout window search are passed instead of alpha value. Apart from window and PV searching differences, both algorithms are identical but formulated differently that is also supported by other researchers such as Yngvi Bjornnson [18].

When it comes to the other heuristic methods, Monte Carlo Search algorithm has been introduced in around 1940s. The researchers were looking for a domain-independent method that does not need a game background. The first time, Bruce Abramson merged minimax search with an expected-outcome model instead of the heuristic evaluation function in his thesis, in 1987 [19]. But the tremendous development of this method has been proved itself in 2006 that was used for AlphaGo's algorithm applying Monte Carlo Tree Search [20] on Go game. In contrast to minimax or alpha-beta type of methods, the algorithm decides next moves without an explicit evaluation function. However, finding promising paths or moves is a challenging problem for the algorithm while searching, as it has limited rollouts to discover unknown paths and potential rewards. As a result, the algorithm has to balance visits for unexplored nodes and promising nodes, seen by previous rollouts. That problem is also called exploration and exploitation dilemma.

After MCTS's development, in order to solve exploration and exploitation dilemma, Upper Confidence Bound1 has been introduced by L. Kocsis and C, Szepesvâri [21]. The policy follows a way that explores in the beginning, then after having exponentially growing regret, it starts to use the promising paths that discovered during rollouts. However, it does not give up for the undiscovered nodes and still explores them though not often as the in beginning. In this way, the algorithm gains an adaptiveness feature as well. If more rollouts are used, the moves likely become more intelligent in the future rollouts. Moreover, the algorithm can be interrupted at any time giving the most promising move till last experimented rollout.

Although MCTS with UCT brings many features together such as accurateness, preciseness, adaptiveness, efficient decision process for complex decision trees and domain-independency, it is still an important research question to say which algorithm is better since each frame, environment, problems have different settings.

Chapter 3
Model and Implementation

3. Model and Implementation

3.1. Pawn Game

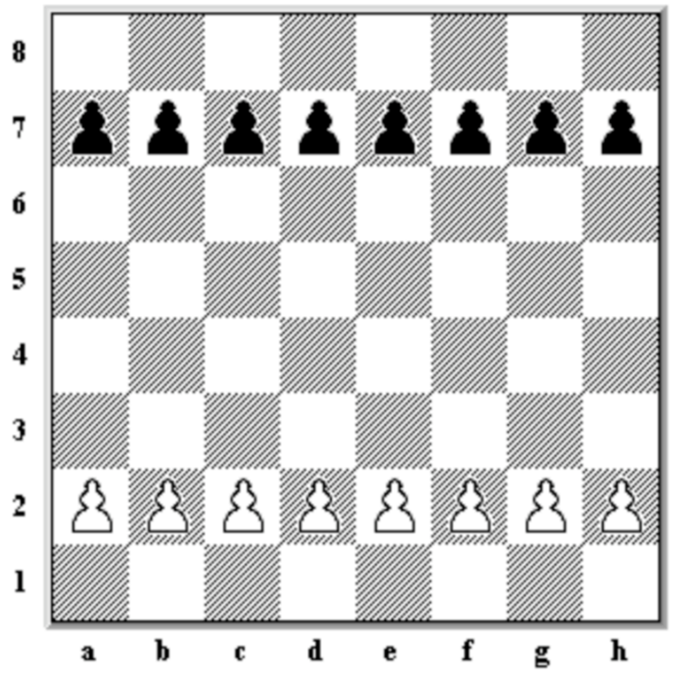


Fig. 1: The Pawn Game [22].

The Pawn Game is a simple version of the chess game that is played with only pawns (Fig. 1). The game has 5 simple rules [1]:

• 1vs1 game played in turn (White, black, white, so on).

• White plays first. That can be 1 or 2 squares forward as it is in the beginning position same as a normal chess game, and the pawns capture diagonally.

• Win position: If you reach the end of the board with a pawn.

• Win position: If you take all the opponent's pawns.

• Win position: if the opponent cannot make a move.

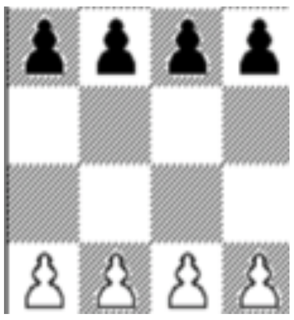


Fig. 2: Modified Pawn Game for Implementation

In this project, we changed the board size and made experiments with 4x4 sizes (Fig. 2) by removing the last edge lines behind the pawns. Also, we removed 2 squares forward move to make the game simpler by changing the second rule. Therefore, the experiments conducted with less computation power due to the small size board that helped to make more experiments in a limited time. Consequently, the second rule is replaced with the following line;

• Black plays first. The pawns capture diagonally and move 1 square forward.

3.2. Minimax

Minimax is a recursive searching algorithm which can be applied to turn-based zero-sum adversarial games such as Checkers and Chess. The algorithm has perfect knowledge about the game as it computes all the possible future positions. The name, “Minimax”, comes from min and max players in the algorithm. The max player aims to maximise the gain that obtained from action results, the min player tries to be against the max player by minimising the utility value that finds best move can be played as an opponent. The algorithm follows a depth-first search by going down of the entire tree until it reaches a terminal node (e.g. a win or a lose).

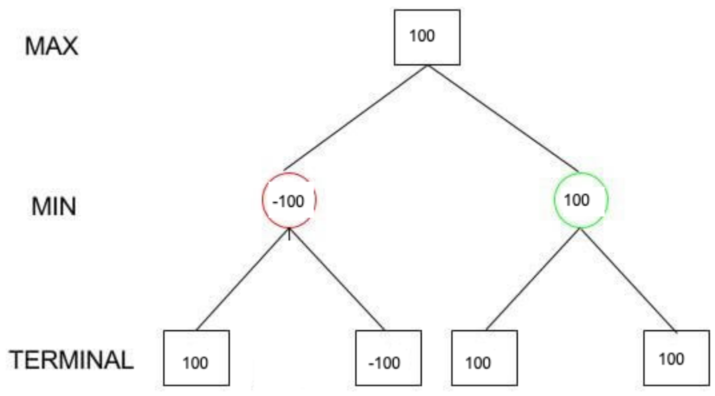


Fig. 3: The Minimax Algorithm with a Simple Decision Tree [23].

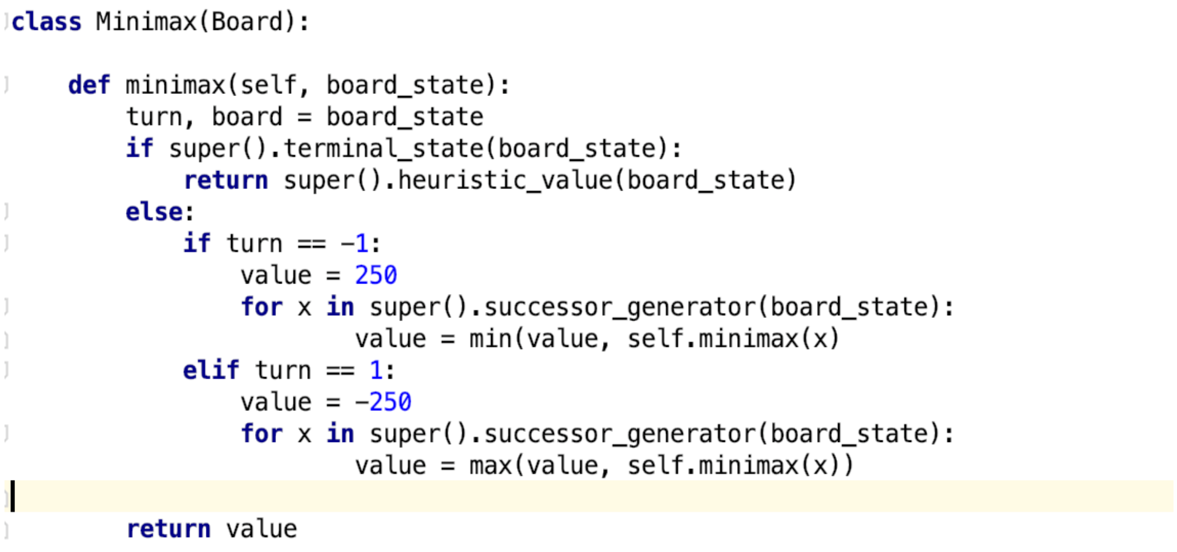
In Fig. 3. the algorithm first computes the entire decision paths from left to right. These decisions return with a utility value such as 100 and -100 those coincide win and lose positions respectively. The min player always chooses the lowest scores since it acts as “opponent” in the algorithm; therefore, it will enforce the max player to decide between worst scenarios as much as possible. The max player selects the highest score to increase utility gain. The algorithm works in this way; the max player first goes left, and the min player begins to play. The min player chooses “-100” as it is lower than 100. After returning with -100, the red circle (in Fig. 3) is assigned as -100. The max player goes left and the min player takes the turn. The min player selects 100 as a minimum value since there is no lower value on this leaf. The green circle is assigned as 100. The max player takes the turn and selects maximum value between the red and the green circle. Eventually, the Minimax algorithm chooses the green circle as a move. Battle of these two players enables the algorithm to choose the best move. Even for more complex decision trees, it gives the best result.

3.2.1. Minimax Pseudocode

```
function minimax(node, maximizingPlayer) is
    if node is a terminal node then
        return the value of node
    if maximizingPlayer then
        value := −∞
        for each child of node do
            value := max(value, minimax(child, FALSE))
        return value
    else (* minimizing player *)
        value := +∞
        for each child of node do
            value := min(value, minimax(child, TRUE))
        return value
```

[24]

3.2.2. Minimax Python Implementation



```python
class Minimax(Board):

    def minimax(self, board_state):
        turn, board = board_state
        if super().terminal_state(board_state):
            return super().heuristic_value(board_state)
        else:
            if turn == -1:
                value = 250
                for x in super().successor_generator(board_state):
                        value = min(value, self.minimax(x)
            elif turn == 1:
                value = -250
                for x in super().successor_generator(board_state):
                        value = max(value, self.minimax(x))

        return value
```

Fig. 4: Minimax Python Implementation

To code Minimax, I refer above pseudo-codes by applying some modifications (Fig. 4). “board_state” represents the node. “maximizingPlayer” in the pseudocode has not been used in our codes. Instead, I added this parameter to the “board_state” which is also assigned as “turn” in the codes therefore, we can apply to toggle that way. The algorithm, first, checks if the node is a terminal node or not. If it is, the utility value of the terminal node is assigned.

Otherwise, the algorithm begins to explore by creating children of the node. To do that, I apply "successor_generator", is a superclass method creates children of the current state. "successor_generator" first checks all the squares on the board and finds the values that are compatible with the turn. For example, if the turn value is 1 that means, the first team plays. The function finds all the ones throughout the board and checks if the forward square is empty or not. If there is no pawn, represented as "zero (0)", the function creates a child that refers a possible move. In order to realise forward move, reference pawn is being deleted from the matrix and a new pawn is assigned for the forward square. In this way, the pawn can be considered moving forward. The modified new board state is added to the list as a successor (child, possible move). After, the function checks the diagonal square if there is an enemy or not for capturing move. In this scenario, if it comes up an opponent pawn such as "-1", the reference pawn is deleted from the matrix to realise "capturing move", and the enemy pawn position is assigned as "1". The function adds this move to the list as another child.
A pawn can create a maximum of 3 children due to limited possible moves; those are an enemy pawn on the left, forward square, an enemy pawn on the right. Then, children create other children. Since the algorithm recursively calls itself, it starts to evaluate from the bottom of the decision tree. After reaching the terminal states, it returns with assigned utility values and starts to collect itself by comparing with "min()" and "max()" python functions considering players' turn. When the function reached the top of the decision tree, it gives the final result.

3.3. Alpha-Beta Pruning

Minimax algorithm is ideal for the games which need less computation power. As the algorithm explores all the decision paths, it is taking too much time to select next move even for simple decision moments those are very clear for what to play in order to win. There are some strategies that reduce complexity and accelerate the algorithm in terms of decision time. One of those methods is Alpha-Beta Pruning that removes unnecessary decision paths. It is worth to emphasize that Minimax and Alpha-Beta Pruning give the same final decision regardless of how the game or tree is complex. However, Alpha-Beta Pruning can reduce computation time significantly for some scenarios as it applies to prune which does not affect utility values but influence paths. The logic behind this algorithm is that it stops evaluating a move when it realises the move is worse than previously assessed move. Here is the strategy shows how it works;

Alpha: Best value as highest so far computed by Max Player along the tree path
Beta: Best value as lowest so far computed by Min player along the tree path

Initially**,** alpha is negative infinity and beta is positive infinity. Therefore, any new input value explored by the algorithm can be stored in these parameters since the infinite values are the worst-case scenarios. The main condition for pruning is;

$$Alpha => Beta$$

(1)

If this condition is fulfilled, Alpha-Beta pruning is realised.

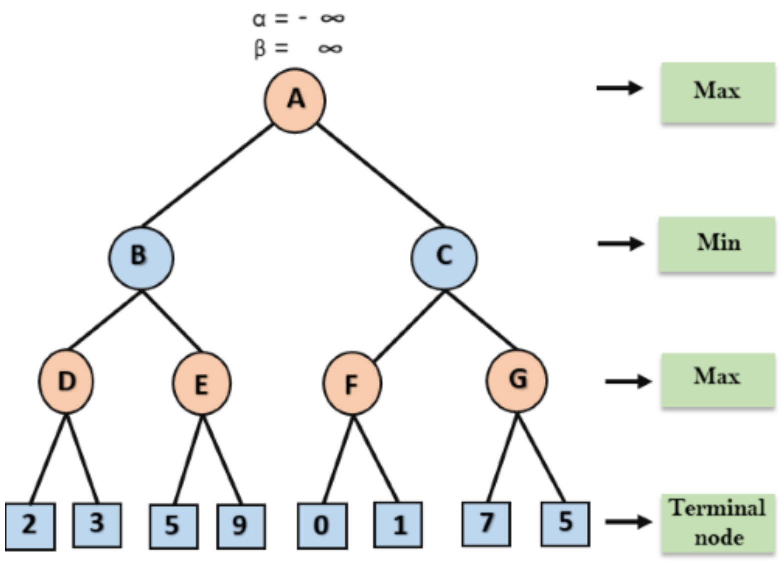


Fig. 5: Alpha-Beta Pruning with a Simple Decision Tree [25].

In Fig. 5, we explain Alpha-Beta Pruning on an example tree. As mentioned above, initial values are negative infinity and positive infinity. We pass these values from node A to node B and after, node D. Since the Minimax algorithm starts exploring from left to right, we follow the same way here. Node D is a max player, therefore, chooses max value between 2 and 3. Alpha parameter only changes if the player is a max player, and B only changes if the turn is a min player's turn. As a result, our alpha parameter becomes the same as node D which is 3.

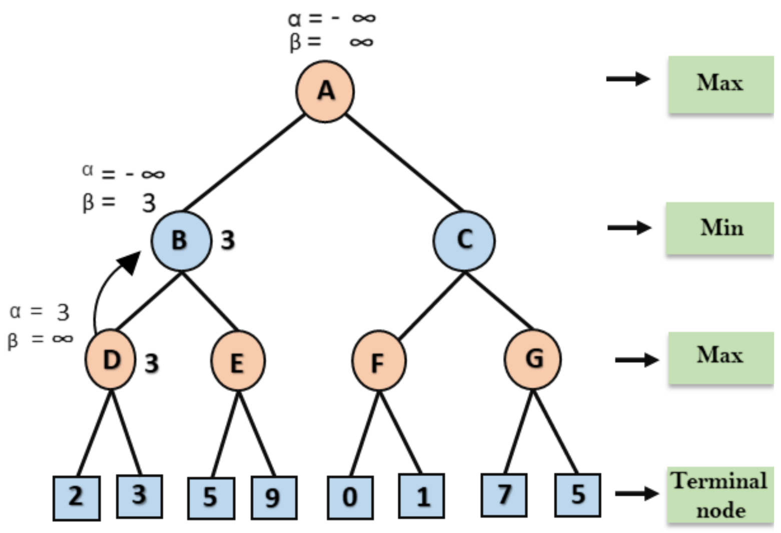


Fig. 6: Alpha-Beta Pruning; Alpha and Beta values of Node D and Node B [25].

Node B’s alpha and beta values were +∞, -∞ respectively. Now, as it is a min player’s turn, beta value is changing. To assign beta, we compare previous node D’s alpha, beta and node B’s beta values those are 3, ∞ and ∞ respectively. Due to min player, we assign minimum value for the beta on node B which is 3. Therefore, our new alpha and beta values of node B are -∞ and 3, that is also shown in Fig. 6

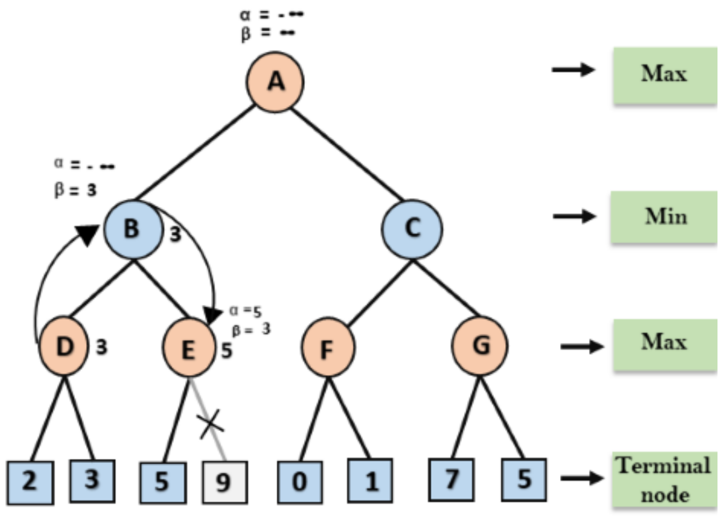


Fig. 7: Alpha-Beta Pruning; Pruning of the Node E’s leaf [25].

Now, we copy these values from node B to node E, therefore, node E is possessing the same alpha-beta values which are -∞ and 3. Since node E refers to a max player, the node is assigned with a maximum value. For the alpha parameter, 5 and -∞ are compared and the bigger one is chosen for new alpha value. As 5 is bigger than -∞, new alpha-beta values become 5 and 3. Here, the main condition (1) is fulfilled where alpha is equal or bigger than beta. For the next step, we don’t need to compute the rest of the path under the node E due to pruning. Whatever the value is in the next leaf, it will not affect overall results. Therefore, the value in the leaf (grey square in Fig. 7) is not important at all.

After pruning, since node B refers a min player; only the beta value is updated with the value come from the comparison between B’s beta (3), E’s alpha (5) and E’s beta (3). As the smallest number is 3 among those three values, we don’t go for an update on B’s beta since it was already 3.

Node A coincides a max player, thus, only the alpha parameter is updated. To change A’s alpha, we compare B’s alpha, B’s beta and A’s alpha values those are equal to -∞, 3, -∞ respectively. The biggest number among those is 3, hence, our new alpha value that belongs to node A is 3.

For the node C, the alpha-beta parameters are passed directly. Node C passes it to node F without changing any parameter. As a result, node A, node C, node F possess the same alpha-beta values on this step that are (3, ∞). In order to update node F, the leaves are being considered. As node F comes with a max player, we choose the maximum value for the alpha parameter. However, 3 is bigger than 0 and 1, that means the alpha is not changing.

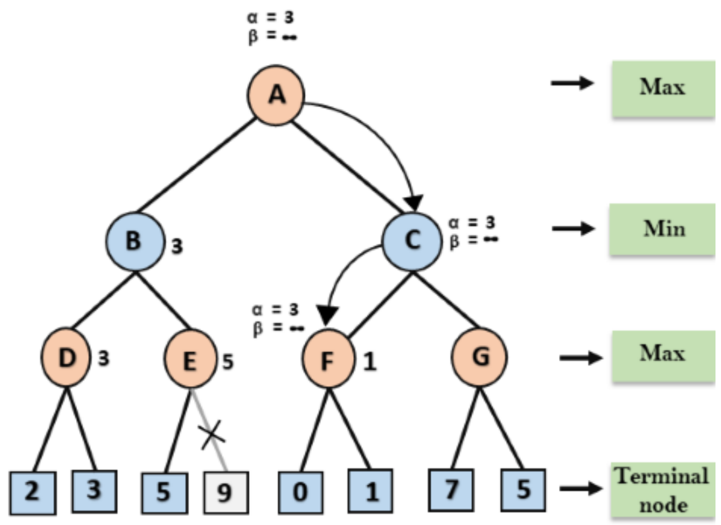


Fig. 8: Alpha-Beta Pruning: Alpha and Beta values of Node C and Node F [25].

To update node C's beta, only the beta is updated as node C coincides a min player, we compare F's alpha, F's beta and C's beta those are 3, ∞, and ∞ respectively. Consequently, our new beta value is 3. Here is again a pruning happens since alpha equals to beta just as mentioned in the equation (1). As a result, the algorithm doesn't compute the G node and the rest of the path under G node Fig. 8.

It is important to mention that alpha-beta values can be compared with node values too (Fail-Soft) [26]; however, instead of referring node values, we prefer only alpha and beta values to create a logic chain to explain this method (Fail-Hard). In the bigger picture, the logic is the same and gives the same result for this problem. Python Implementation refers to Fail-Soft Alpha-beta.

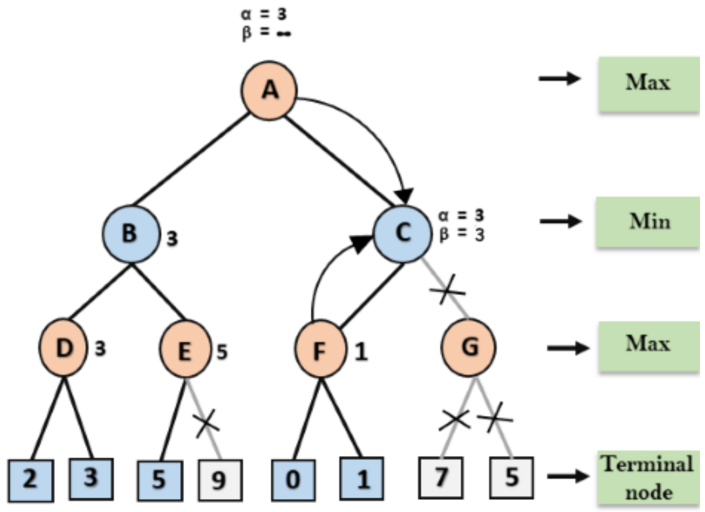


Fig. 9: Alpha-Beta Pruning: Pruning of Node C's Branch [25].

In sum, after applying to prune, nearly half of the tree is not being computed. In this way, the algorithm saves significant time and increases decision speed for some scenarios. The order of the moves plays a huge role in this algorithm since the algorithm follows a computing strategy from left to right. The overall result can be seen in Fig. 9 returns with 3 as utility value. Although there are higher utility values on the terminal nodes, the algorithm suggests node D's path as the safest option by following the green path on the Fig.10.

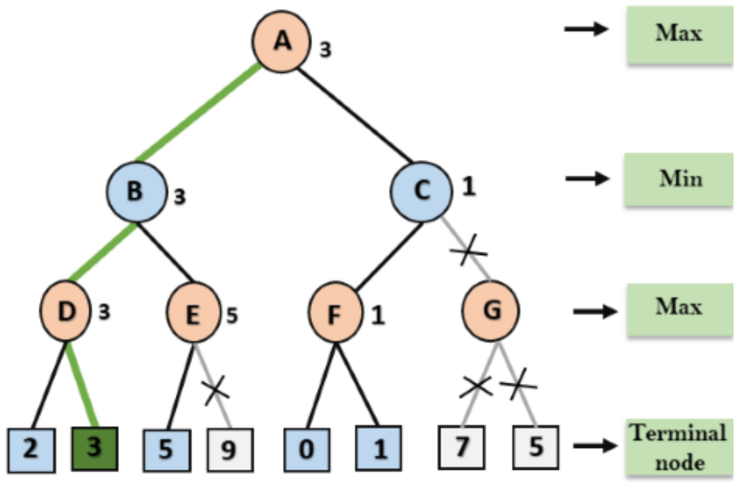


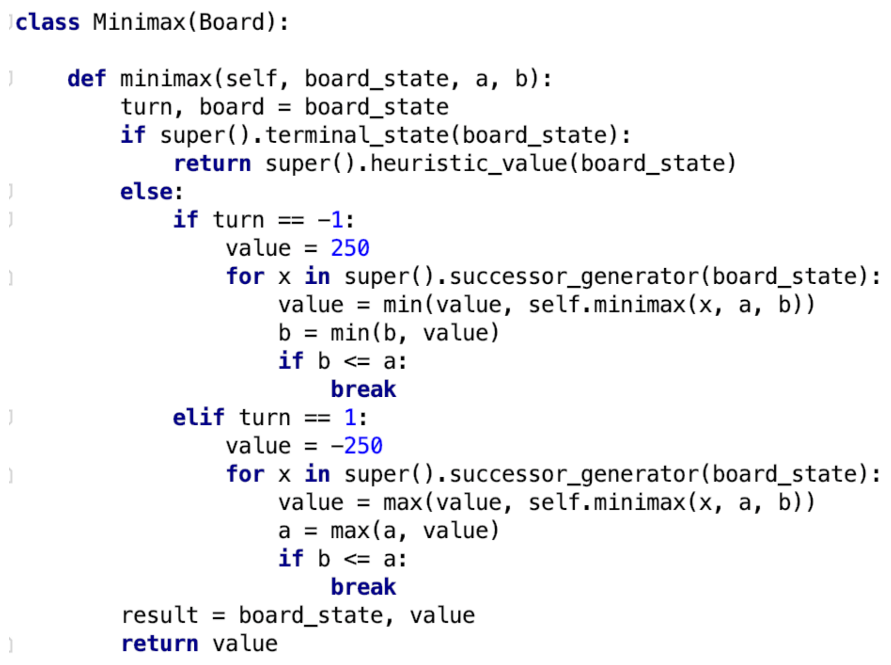
Fig. 10: Alpha-Beta Pruning: Deciding the Winner Move [25].

3.3.1 Alpha-Beta Pseudocode

```
function alphabeta(node, α, β, maximizingPlayer) is
    if node is a terminal node then
        return the heuristic value of node
    if maximizingPlayer then
        value := −∞
        for each child of node do
            value := max(value, alphabeta(child, α, β, FALSE))
            α := max(α, value)
            if α ≥ β then
                break (* β cut-off *)
        return value
    else
        value := +∞
        for each child of node do
            value := min(value, alphabeta(child, α, β, TRUE))
            β := min(β, value)
            if α ≥ β then
                break (* α cut-off *)


[24]
```

3.3.2. Alpha-Beta Python Implementation



```python
class Minimax(Board):

    def minimax(self, board_state, a, b):
        turn, board = board_state
        if super().terminal_state(board_state):
            return super().heuristic_value(board_state)
        else:
            if turn == -1:
                value = 250
                for x in super().successor_generator(board_state):
                    value = min(value, self.minimax(x, a, b))
                    b = min(b, value)
                    if b <= a:
                        break
            elif turn == 1:
                value = -250
                for x in super().successor_generator(board_state):
                    value = max(value, self.minimax(x, a, b))
                    a = max(a, value)
                    if b <= a:
                        break
        result = board_state, value
        return value
```

Fig. 11: Alpha-Beta Python Implementation.

Same as Minimax, we add two parameters to the function which are “a” and “b” in order to code alpha-beta pruning strategy (Fig. 11). Initially, “a” and “b” should be set with -∞ and ∞ respectively as worst-case scenarios as in the original implementation. However, in the codes, these infinite numbers represented with 250 which is a high number considering terminal state and heuristic utilities but dangerous to be used in implementations. I chose it due to integer structure. For this reason, our initial (a, b) values are equal to (-250, 250) correspondingly that do the same work. The algorithm follows a recursive way to decide final move same as Minimax.

3.4. Depth-Cut Method

To reduce computation time and increase the speed of decisions, I implement the depth-cut method that applies a heuristic strategy for a final decision after reaching a certain depth. Since some games have more complex decision trees, are difficult to be computed by Minimax or Alpha-Beta algorithms, such as Chess, Go or the Pawn Game, with bigger board size 8x8 or 9x9 instead of 4x4, depth-cut method assists us on that problem.

The algorithm works that way; first, it checks terminal states that return with “win” or “lose” utility. Then, if there is not a terminal state through the path, the algorithm explores deeper decision layers until it stops. Stopping is realised when the algorithm reaches a certain depth layer determined by a user. Later, the algorithm applies a heuristic estimation function, that is explained in the next chapter, referring the child on the last depth layer and returns with a heuristic utility value. The heuristic utility value represents how the selected board state is close to win or lose (Fig. 12). After that, the algorithm acts the same as Minimax, that chooses the final decision among those utilities by considering min and max players.

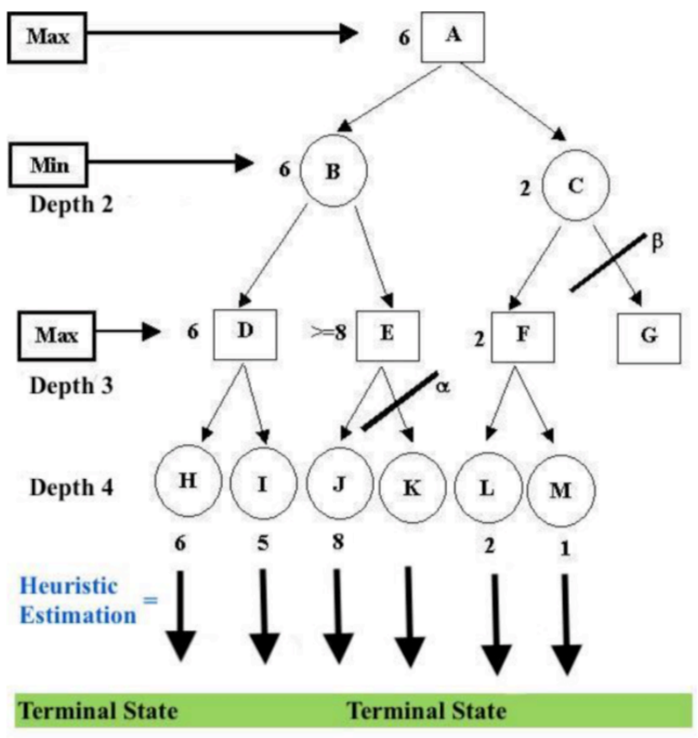


Fig. 12: Depth-Cut with Heuristic Evaluation

In Fig. 12, the Depth-Cut algorithm begins to explore from Node A that is the first depth layer and main node. Then, it goes left and visits the node B and D sequentially. Since the depth limit is 4 determined by a user, here is determined by us, it explores the node H and I together. After these nodes, the depth search is cut by the algorithm as we don't want to compute the entire decision tree. From that point, heuristic equations are called by the algorithm and the node H returns with "6 that short-cut value illustrates how the H is close to a terminal state according to equations. The node I follows the same way and returns with "5". Since node D corresponds a max player, it selects a move that has a higher heuristic value which is 6. At the same time, the alpha value is also assigned as 6. Since it is a max player, only the alpha changes. For node B, beta is assigned as 6 since it is lower than infinite. After, the algorithm starts to explore node E by transferring alpha-beta values from the parent node (Node B). Now, node E is possessing (-∞, 6) alpha-beta values. Node J is checked for the update. Then, the new node and alpha-beta values are updated. Since Node J corresponds a max player, the bigger number is chosen for the alpha value between (-∞) and 8 which is 8. Here a pruning realises as alpha is bigger than beta value; therefore, we can pass the node K without computing it. When the algorithm goes back, it compares the values for node B, that correspond a min player, to choose the minimum score that is Node D's value "6" as it is lower than 8. Then, the algorithm explores node C by copying alpha-beta values from A which are (6, ∞). Node C passes these values to node F, and it compares the values between alpha (6), L (2) and M (1) for the new alpha value of node F that refers a max player. Therefore, the new alpha value becomes 6 since it is the highest number between 6, 2 and 1. For Node C's new beta value, since it is a min player, it is selected between node F's alpha (6), node F's beta (∞) and node C's beta (∞) for the minimum value. Due to the outcome of this node, which is 6, pruning happens. When prune happens, the algorithm doesn't compute the next branches under the C node.
For the decision part, Node F becomes 2 as it is a max player, and node C becomes 2 as well due to lack of an alternative path. Finally, node A decides the maximum value between node C and node B that is resulted in 6, that means node B is selected as the final decision.

3.4.1. Depth-Cut Pseudocode

```
Function alphabeta(node, depth, α, β, maximizingPlayer) is
    if depth = 0 or node is a terminal node then
        return the heuristic value of node
    if maximizingPlayer then
        value := −∞
        for each child of node do
            value := max(value, alphabeta(child, depth − 1, α, β, FALSE))
            α := max(α, value)
            if α ≥ β then
                break (* β cut-off *)
        return value
    else
        value := +∞
        for each child of node do
            value := min(value, alphabeta(child, depth − 1, α, β, TRUE))
            β := min(β, value)
            if α ≥ β then
                break (* α cut-off *)
        return value
```

[24]

3.4.2. Depth-cut Python Implementation

```python
def minimax(self, board_state, a, b, depth = None):
    turn, board = board_state
    if super().terminal_state(board_state, depth):
        return super().heuristic_value(board_state, depth)
    else:
        if turn == -1:
            value = 250
            for x in super().successor_generator(board_state):
                if depth is not None:
                    value = min(value, self.minimax(x, a, b, depth - 1))
                else:
                    value = min(value, self.minimax(x, a, b))
                b = min(b, value)
                if b <= a:
                    break
        elif turn == 1:
            value = -250
            for x in super().successor_generator(board_state):
                if depth is not None:
                    value = max(value, self.minimax(x, a, b, depth - 1))
                else:
                    value = max(value, self.minimax(x, a, b))
                a = max(a, value)
                if b <= a:
                    break
    result = board_state, value
    return value
```

Fig. 13: Minimax with Depth-Cut Python Implementation

As can be seen in Fig. 13, we have a new parameter “depth” that determines the maximum depth layer that the algorithm will be looking for. That parameter is checked as a part of the terminal state. Normally, in pseudocode of this algorithm, node and depth are defined separately, however, in our codes, I coded it under same function “terminal_state” that checks depth and terminal nodes together. While calling itself recursively, it decreases the depth number on each call. In this way, the function is being able to apply a heuristic function for the bottom depth layers. To assign heuristics to the last (leaf) nodes, the algorithm calls “heuristic_value” function (explained in 3.5. in more detail) that assigns not only heuristics but also terminal state utilities.

In “heuristic_value” function, the depth parameter is set as a very high number, “5e103”. In sense, that is equal to Minimax as there is no sort of that big decision tree for the Pawn Game to compute that high depth number. The infinite number could be chosen, however, selecting a high number instead of “infinite” helped us to estimate “*shortest path*” for the winning positions.

In Minimax or Alpha-Beta Pruning, it is possible to have two or more moves that have the same heuristic value or utility. In these scenarios, the max player, always plays first in the python implementation, chooses an index that comes first between those equal maximum utilities. This selection method causes some issues such as repeating the same game strategy for the same board states. A winner successor which reaches a game win faster than other winner successors is always better. Therefore, to find the shortest winning path, the depth parameter has been added to the utility values since it is decreased by each step due to recursion. To not influence utilities significantly, the depth number is multiplied with "0,001", much lower number than utilities, and added to the utility values. In this way, this issue can be reduced considerably but not completely.

3.5. Heuristic Function

For the heuristic estimation, I use two types of heuristics. These are called simple heuristic and advanced heuristic through the paper. The heuristic functions help us to have insight into how the board state is close to win or lose. The decision does not have to be perfect or optimal but should be good enough. There are some features for a heuristic function to be counted as a heuristic strategy. Those are including optimality, completeness, execution time, accuracy and precision but not limited with these. Therefore, I tried to create two types of heuristic also considering game rules.

3.5.1. Heuristic 1 (Simple)

As a first heuristic function, I preferred to apply a simple one at the beginning that accelerates the experiments to observe the depth-cut algorithm against other players such as an MCTS, random or human player (explained in *Experiments* section). For this reason, the pawn numbers have been considered for the simple heuristic. If the main player, the yellow team (Fig. 14), has more pawns or the opponent player, purple team, has fewer pawns, the heuristic function outcomes a higher score. As a result, the main player is possessing a tendency to capture enemy pawns during the game. Since one of the winning ways is to capture all enemy pawns, this simple heuristic method can be implemented for the experiments.

$$\begin{aligned} &Pawn\ Number\ Score \\ &= (Yellow\ Team\ Pawn\ Numbers - Purple\ Team\ Pawn\ Numbers + 1) \\ &* (|Yellow\ Team\ Pawn\ Numbers - Purple\ Team\ Pawn\ Numbers|) \end{aligned}$$

(2)

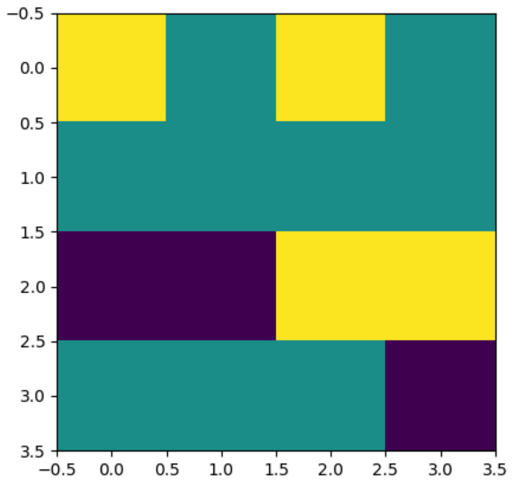


Fig. 14: A 4x4 Board State Sample from the Experiments.

To see the value differences in a better way, I emphasised this score by multiplying itself using the above equation and assigned it to utility value. If the users prefer to apply this heuristic method during the experiments, they have to assign "heuristic_parameter" as "True" in the codes (in players.py, files shared in appendixes). In Fig. 14, the yellow team has 4 pawns while the purple team has 3 pawns. Our "Pawn Number Score" (2) can be obtained by (4-3+1)*(|4-3|) that is equal to "2". As a result, the utility value (3) is assigned as 2 if that was a successor board state. On the other hand, it is clear to say that, the next move would be capturing the purple pawn according to this heuristic equation instead of going forward, those are both winning positions.

$$Utility\ Value = Pawn\ Number\ Score$$

(3)

3.5.2 Heuristic 2 (Advanced)

As a second heuristic function**,** I decided to apply an advanced one that considers all winning ways in the function. Therefore, not only the number of pawns but also the position of the pawns considered. For the positions, Promotion Score (4) and Distance Score (5) taken into account.

$$\begin{aligned} &Promotion\ Score \\ &= (Total\ square\ numbers\ of\ Purple\ team\ to\ reach\ for\ promotion \\ &- total\ square\ numbers\ of\ Yellow\ team\ to\ reach\ for\ promotion) \end{aligned}$$

(4)

Promotion Score emphasises the importance of the pawn positions' distance between the last board lines and pawns. In this way, the algorithm recognises that reaching to the last lines is important for winning. Additionally, preventing enemy pawns for not reaching the last line is being taken account with this strategy.

$$Distance\ Score = (-1^{distance\ value+1}) * distance\ value$$

(5)

Distance Scores emphasises the importance of the positions between pawns. According to the winning situation, if the enemy does not have any move to play, you win. Considering that game rule, the algorithm endorses its heuristic with Distance Score equation. As odd and even square numbers between pawns play a huge role in winning and losing, an exponential expression was used in the equation. In this way, the Distance Score can be negative if the pawn positions risk winning.

$$Utility\ Value = Promotion\ score + 2 * Pawn\ Number\ Score + Distance\ Score$$

(6)

Utility Value (6) is defined above including pawn number score with promotion and distance score. The reason for multiplying with 2 is that Pawn Number Score is usually lower than the other two, therefore, it is emphasised more than others in the equation.

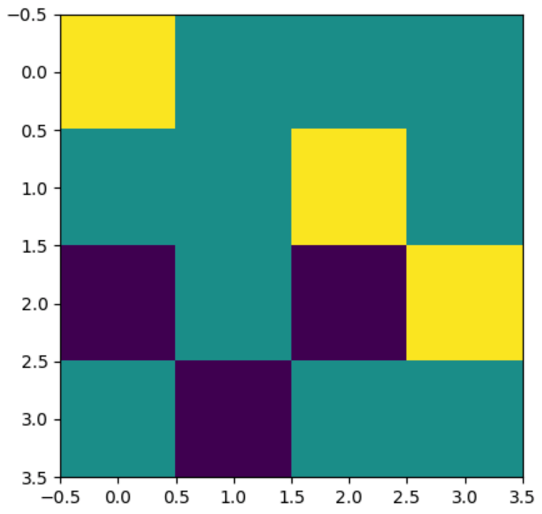


Fig. 15: Another Board Sample from the Experiments

In Fig. 15, estimation of Utility Value according to advance heuristic follows these steps; first, from left to right, Yellow Team Pawn square numbers (4) are 3+0+2+1 equal to 6 according to

(4). Purple Team Pawn square numbers are 2+3+2+0 equal to 7 (4). In this case, the Pawn Number Score (4) is 7-6 = 1.

The pawn differences between two team is 3 (Yellow) - 3 (Purple) = 0 according to equation (2). In this case, the pawn score (2) is (0+1) *(|0|) = 0.

Distance Value (5) is 1+0+0+0 = 1 since there is only one square gap between yellow team pawns and purple team pawns. Therefore, our distance score (5) is equal to 1 when distance value is put in the equation.

In sum, the utility value (6) is 1 (Promotion Score) + 2*0 (Pawn Number Score) + 1 (Distance Score) = 2. That is only one heuristic value among other children. After that estimation, the depth-cut algorithm decides the highest score among these values when it is a max player's turn.

3.6 MCTS

Monte Carlo Tree Search is a heuristic searching method using randomness for the complex game decision trees that are increased exponentially when the game goes forward. The difference between MCTS and Minimax, MCTS does not need to know board positions while deciding new moves. Therefore, MCTS does not apply an explicit heuristic evaluation function. MCTS also has an adaptive strategy to explore the moves that is called UCB1.

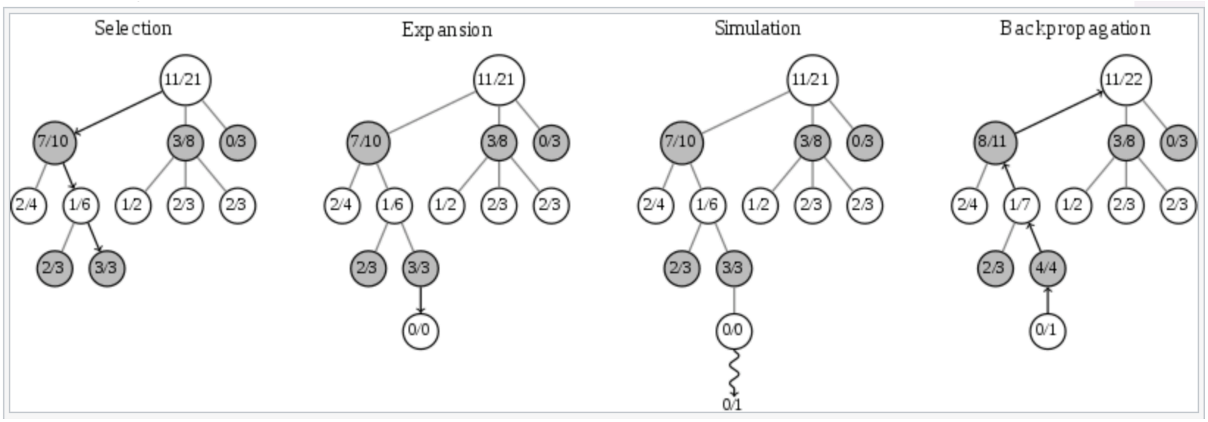


Fig. 16: A Rollout: Steps of Monte Carlo Three Search [27].

The MCTS with UCB1 consists of 4 steps (Fig. 16):

• *Selection*: The algorithm starts from the main node and chooses a successor node until it reaches a leaf node. The main node represents a current board state, and a leaf node means to have no simulation (rollout) initialised yet. Selecting a successor node (child node) can be challenging and tricky as each time if the algorithm chooses random children, it might not come up with a statistically promising move since the distribution of the statistics can be completely random or unmeaningful. There are some methods that increase the efficiency of the algorithm by finding more promising paths and rewards for complex decision trees. Here, UCB1 strategy is applied that optimises exploration and exploitation process while searching throughout the tree (Fig. 17). Using the below equation, the selection function chooses the highest UCT value among the children. That reduces randomness considerably.

$$UCT_j = X_j + C * \sqrt{\frac{ln(n)}{n_j}}$$

Fig. 17: Upper Confidence Tree Formulation [28].

- X_j: The win ratio of the successor
- n: Visited times for parent node
- n_j: Visited times for child node
- C: Trade-off parameter between exploration and exploitation (Exploration Weight)

The constant value C is usually chosen empirically. However, in the codes, it has been ignored by chosen as 1. X_j is the first component corresponds to exploitation. The second component corresponds to exploration that is high in the beginning and getting lower after a few iterations. Therefore, the equation helps us to follow a random way in the beginning for exploring the undiscovered paths. After a while, the algorithm selects the children in a more adaptive way depending on the statistics created by its previous experiences during searching.

• *Expansion:* This function adds a new successor (or successors) to the tree if the leaf node is not a terminal node.

• *Simulation:* This function starts to play by itself with randomly chosen moves from the expanded child node until reaching a terminal state.

• *Backpropagation:* This function updates the statistical values on the path from the expanded node to the main node with the results of the rollout. In Fig, 16, simulation results in a lose for the white nodes. Therefore, the backpropagation updates white notes by adding 1 to denominators. For the grey nodes, the numerator and denominator are either added one by backpropagation. As a result, the algorithm can recognise risky paths in a better way while exploring and exploiting.

The algorithm follows the above steps in order repeatedly; selection, expansion, simulation, backpropagation. In the codes and paper, these four ordered steps will be called “rollout”.

Chapter 4
Tests

4.1. Players

Experiments have been conducted between 4 players.

• Random Player

• Human Player

• Minimax with Depth-Cut Player

• Monte Carlo Three Search with UCT Player

The random player chooses a move between successors, belong to the current board state, randomly. On the other hand, the human player enables a user to choose a move from a provided list of successors.

4.2. Unit Test, Game Dynamic and Successor Checks

In order to observe the codes that including the logic of the moves, game rules and the environment, initially random player has been created. With a random player, the game dynamics can be observed with the human eye as a random player can play against itself in the beginning. After ensuring the game dynamics, a unit test was created for a detailed check to assure that moves are realised properly without neglecting any game rule.

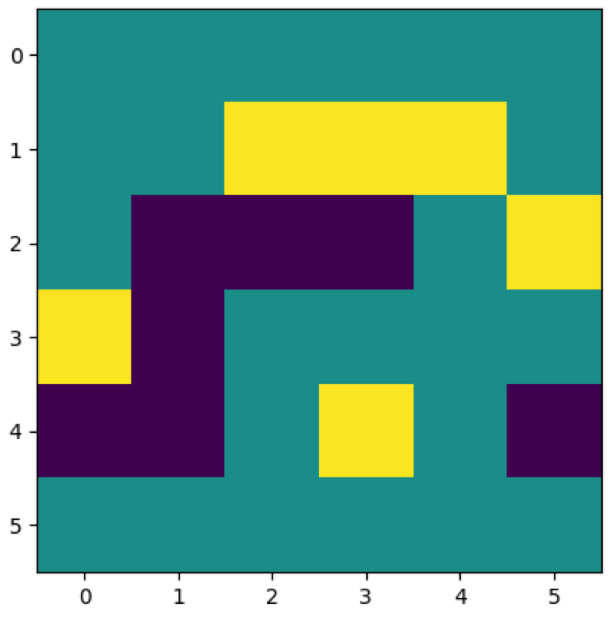


Fig. 18: A 6x6 Board Sample for Unit Test

For the unit test, I created a reference board state (shown in Fig. 18) and determined the turn as the yellow team's turn. The board size was determined as 6x6 for a unit test. The pawn positions were determined this way intentionally to check some specific situations such as diagonal capturing moves, successors, forward moves, forward enemy moves, diagonal enemy captures, two threat for one alliance pawn, two threat for one enemy pawn and stuck pawns (have no moving ability). First, the successors of the reference board for each team were spotted one by one and added to a list. Secondly, "successor_creator" function implemented to the reference board state to create successors. Then, these two lists; manually and automatic created one are compared and tested every time when we run main.py. If the move unit test is successful, the algorithm asks for choosing the next experiment's players to the user.

In order to have more control over positions, I added a human player that provides successor choice to the user for selecting one as next move. Therefore, it made the Pawn Game more flexible, can be played between not only algorithms but also humans.

For the Minimax algorithm, I added a depth-cut algorithm that covers Minimax due to depth parameter. When high number selected for this parameter such as 25,000, the algorithm acts just as Minimax since there are not enough depth layers on the decision tree for this number. Therefore, the algorithm has to compute the entire decision tree same as Minimax. In order to ensure that the algorithm works properly, I added a unit test to check the terminal state's utilities. I created 5 reference board states those are winning and losing positions for each team. If the player applies minimax, it should return with utility value. As a result, I manually assigned the utilities to a list for comparison in output utilities of Minimax. If they are equal, the algorithm passes the test. In this way, we could assure that when the codes are modified or updated, we do not need to worry about utility output as they will be checked automatically when the main.py runs (shown in appendixes files).

The depth-cut algorithm has two optional parameters that should be determined by modifying the codes manually instead of user input; those are depth and heuristic type. Depth-level, in the experiments, has been chosen as 2, 3, 4 and 5. For the heuristic type, I used both heuristics in the experiments; simple and advanced one.

However, since Minimax type algorithms give same results due to deterministic structure of the codes that was always choosing the first index for the max player between winners, the game was played the same way each time without any different move. Therefore, I added a shuffling method that shuffles successors at every step. In this way, if the algorithm has many winners that have the highest utility values, each game, it can choose a different one. That led experiments to be varied and properly analysed.

For the Monte Carlo Tree Search Player, I integrated Luke Harold's codes [29] to my codes. These codes designed for MCTS approach on board games such as tic-tac-toe or similar simple games. For the Pawn Game, other metaclass functions of the codes had to be coded from scratch considering game rules. In this way, the integration of the codes realised. The MCTS player has one optional parameter, named "rollout", determines the number of rollouts. In this way, the experiments can be varied. The number of the rollouts has been selected as 1, 10, 100 and 500.

Each experiment has been run 199 times that is equal to "one match" during the paper although technically 199 matches have been played. Each win also named "score" in the results section. For instance, a random player against a random player experiment, the computer restarted the initial board state, and 199 games played. First random player gained 104 scores that means 104 matches have been won in total. The experiments in the table conducted using parallel computation technique that can reach maximum 10 different games running at the same time.

First random player experiments conducted as a first player and second player as can be seen in the table. After that, Minimax experiments conducted between simplified heuristic and advanced heuristics including being against a random player. After, MCTS experiments realised, and the results automatically saved after each experiment by Python codes.

Player turns have been changed to obtain effects of player turns, and all experiments re-performed again. Board Size has been changed and some experiments repeated again.

Chapter 5 Experiments

5.1. Experiments

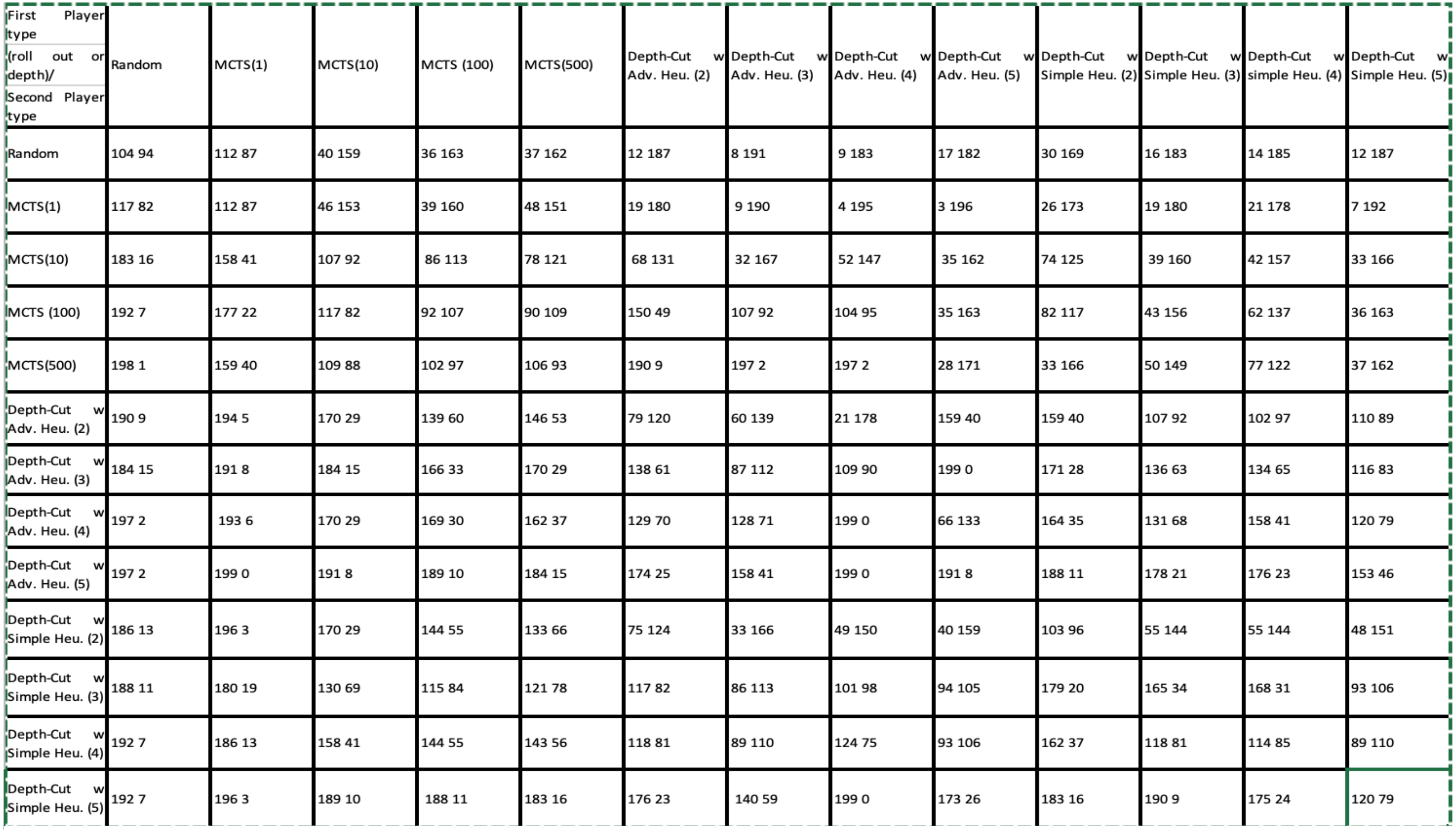
Minimax, first, applied to the board. Each game, although the successors shuffled each time when created automatically, it won all the games against a random player around 600 games (200 for 4x4, 200 for 5x5, 200 for 6x6). The algorithm also has been compared in MCTS that resulted in winning of Minimax each time.

Alpha-beta pruning showed its efficiency by reducing the time in terms of game durations. The results against a random player and MCTS player did not change but the decision speed approximately doubled for each move in different board sizes.

	First Player/Second Player	MCTS100	Depth-Cut4
Board 4x4	MCTS100	11.00s 120-79	10.63s 104-95
	Depth-Cut4	8.69s 76-123	
Board 5x5	MCTS100		39.27s 96-103
	Depth-Cut4		
Board 6x6	MCTS100		162.25s 85-104
	Depth-Cut4		

Fig. 19: Experiment Table 1

In Fig.19, the Depth-Cut4 algorithm uses the advanced heuristic function. When the Depth-cut plays first, MCTS reaches more score for 4x4 board sizes. While the MCTS vs Depth-Cut has 104-95 scores, Depth-Cut vs MCTS reaches 76-123. The winner's score is differing remarkably. Apart from this, increasing board sizes makes the game slower as expected, and depth-cut starts to have more scores according to Fig. 19. Each game for 6x6 lasted approximately 3 minutes. The other experiment has been conducted between MCTS100 vs MCTS100 to see the first move's importance. The results illustrated that it plays a major role as the difference between scores is nearly 40 out of 200. After these experiments, we extended and conducted new experiments by using 4x4 board size. Therefore, the gaps in the table (except the first column) represent non conducted experiments that were left intentionally.



First Player type (roll out or depth)/ Second Player type	Random	MCTS(1)	MCTS(10)	MCTS (100)	MCTS(500)	Depth-Cut w Adv. Heu. (2)	Depth-Cut w Adv. Heu. (3)	Depth-Cut w Adv. Heu. (4)	Depth-Cut w Adv. Heu. (5)	Depth-Cut w Simple Heu. (2)	Depth-Cut w Simple Heu. (3)	Depth-Cut w simple Heu. (4)	Depth-Cut w Simple Heu. (5)
Random	104 94	112 87	40 159	36 163	37 162	12 187	8 191	9 183	17 182	30 169	16 183	14 185	12 187
MCTS(1)	117 82	112 87	46 153	39 160	48 151	19 180	9 190	4 195	3 196	26 173	19 180	21 178	7 192
MCTS(10)	183 16	158 41	107 92	86 113	78 121	68 131	32 167	52 147	35 162	74 125	39 160	42 157	33 166
MCTS (100)	192 7	177 22	117 82	92 107	90 109	150 49	107 92	104 95	35 163	82 117	43 156	62 137	36 163
MCTS(500)	198 1	159 40	109 88	102 97	106 93	190 9	197 2	197 2	28 171	33 166	50 149	77 122	37 162
Depth-Cut w Adv. Heu. (2)	190 9	194 5	170 29	139 60	146 53	79 120	60 139	21 178	159 40	159 40	107 92	102 97	110 89
Depth-Cut w Adv. Heu. (3)	184 15	191 8	184 15	166 33	170 29	138 61	87 112	109 90	199 0	171 28	136 63	134 65	116 83
Depth-Cut w Adv. Heu. (4)	197 2	193 6	170 29	169 30	162 37	129 70	128 71	199 0	66 133	164 35	131 68	158 41	120 79
Depth-Cut w Adv. Heu. (5)	197 2	199 0	191 8	189 10	184 15	174 25	158 41	199 0	191 8	188 11	178 21	176 23	153 46
Depth-Cut w Simple Heu. (2)	186 13	196 3	170 29	144 55	133 66	75 124	33 166	49 150	40 159	103 96	55 144	55 144	48 151
Depth-Cut w Simple Heu. (3)	188 11	180 19	130 69	115 84	121 78	117 82	86 113	101 98	94 105	179 20	165 34	168 31	93 106
Depth-Cut w Simple Heu. (4)	192 7	186 13	158 41	144 55	143 56	118 81	89 110	124 75	93 106	162 37	118 81	114 85	89 110
Depth-Cut w Simple Heu. (5)	192 7	196 3	189 10	188 11	183 16	176 23	140 59	199 0	173 26	183 16	190 9	175 24	120 79

Fig. 20: Experiment Table 2

The first column represents the first player that includes some symbolisations such as “(10)” refers 10 rollouts if it is an MCTS player. Otherwise, it represents a depth parameter for heuristic evaluation of the depth-cut algorithm. First raw represents second players. If MCTS(1) from the first column and Depth-Cut with Advanced Heuristic intersects on the table, that means MCTS(1) played first against the Depth-cut with Advanced Heuristic Player.

The results indicate that random player vs random player overall score is equal which is expected as they both are the same algorithm but the first player gains approximately 5% more scores.

The number of rollouts against random player illustrates that MCTS wins more matches when rollouts increased e.g. MCTS (1) gathers 117, MCTS (10) gathers 183, MCTS (100) gathers 192, MCTS (500) gathers 198 scores out of 199 matches. As a second player, it gives different results; MCTS(1) gathers fewer scores than a random player. MCTS (100) reaches 163 scores, and MCTS (500) reaches 162 scores; therefore, 100 and 500 rollouts don’t show big differences against a random player as a second player.

On the other hand, advanced heuristic indicates fluctuating results as a first player against a random player by getting these scores 190, 184, 197, 197 for the depth parameters 2, 3, 4, 5 respectively. As a second player, it gathers 187, 191, 183, 182 scores against a random player. The simple heuristics shows more compatible results; depth level and scores increase at the same time as a first (scores respectively; 186, 188, 192, 192) and second player (scores respectively; 169, 183, 185, 187).

MCTS vs MCTS results indicated compatible results; proportional increasing happens between the scores and rollouts. Another point is that, while the difference between MCTS (1) and MCTS (10) was very notable approximately 60 scores, it did not react the same way between MCTS (100) and MCTS (500) reached nearly 20 scores as a first and second player.

Simple Heuristic vs Simple heuristic illustrated compatible results same as MCTS vs MCTS due to increasing depth and score relation but only from second player’s perspective. From first player’s window, surprisingly, “odd” depth limit numbers (3, 5) collect more scores than “even” depth limit numbers (2, 4).

Simple Heuristic vs MCTS indicated that depth (2) reaches more scores than depth (3), and can reach more scores than depth (4) as a first player. MCTS showed more compatible results; proportional increase in the rollouts and scores against the simple heuristic player as a second player. When it comes to comparison in MCTS vs Simple heuristic, the odd-even effect appears here too, depth (3, 5) collects more scores than depth (2, 4). MCTS, as a first player, illustrates compatible results too apart from MCTS (500) vs simple heuristic (2) match (MCTS score: 33) that is a huge decreasing in the score when compared with MCTS(100) vs simple heuristic(2) (MCTS: 82).

Simple heuristic vs advanced heuristic showed compatible results for the first player. For the advanced heuristic (2nd[*]) odds-even effect has been observed but this time the depth number (3) showed the highest score; 83, 113, 98, 105 for 2, 3, 4, 5 respectively.

MCTS vs Advanced heuristic showed rollout's power against all depth levels except advanced heuristic (5). Increasing rollout boosted gained scores tremendously but advanced heuristic (5) showed a surprising resistance. While MCTS (500) vs advanced heuristic (4) gives 197-2 scores, MCTS (500) vs advanced heuristic (5) gives 28-171. The difference between the scores is significantly notable. Advanced heuristic vs MCTS resulted in winning of the first player by a large margin although rollouts showed its effect that could not achieve any winning in the total score.

The most interesting data have been observed during the advance heuristic vs advanced heuristic matches due to score distribution. While increasing depth number usually causes more scores, advanced heuristic (4), as a second player, suddenly loses all his powers against adv. Heuristic (4) and (5) by gaining no score. Same results happen to advanced heuristic (3) vs advanced heuristic (5) that resulted in 199-0. Later experiments adv. heu (5) vs adv. heu (5) give similar results but also gaining some scores, 191-8. Interestingly, adv. heu. (4) vs adv. heu. (5) reaches 66-133 score that is extremely contrasting with its neighbour experiments on the table.

Apart from rollout, depth and score correlations, advanced heuristic (1st) most of the time achieved more scores than simple heuristic player against all players. Also, advanced heuristic (1st) did not lose any match against random, simple heuristic and MCTS player including all parameters. On the other hand, simple heuristic (1st) lost some matches against advanced heuristic.

Interestingly, MCTS (1st and 2nd) lost all the games against simple heuristic but MCTS (1st) won against the advanced heuristic player in some cases.

[*] 1st = As a first player, 2nd = As a second player.

Chapter 6
Discussion and Evaluation

6.1. Discussion

Can minimax and alpha-beta be applied for the Pawn Game to compute entire decision trees for winning move?

After coding of the Pawn Game rules, minimax has been successfully applied for this discrete deterministic zero-sum game by creating successor creator and static evaluation functions. Using dynamic programming, recursive calling method has been implemented. The algorithm not only approved by experiments against random or MCTS player but also unit tests confirmed its accurate implementation by returning with the corresponded utilities. As a result, the answer to this question has been confirmed positively.

Can we add heuristic search to alpha-beta pruning method?

Due to the high amount of computation power, following Shannon's suggestion in 1950 [7], heuristic search has been applied. The game durations and score relations illustrated that depth-cut algorithm works well either for the first and second player with a tremendous increment of the algorithm's decision speed. This implementation increased the number of experiments during the thesis. Also, the successor list has been checked manually with the human eye if the returned utility values are responding properly or not. The implementation not only reduced time for decisions but also gave good enough moves that beat other action recommenders such as random and MCTS. In sum, experiments have been proved that depth-cut algorithm can be added alpha-beta.

Is a player's turn important on the Pawn game?

Experiment results clearly emphasised that playing first gives a very high chance of winning and gaining more scores to the players. That been proved by the result of Random vs Random, MCTS vs MCTS, Simple vs Simple Heuristic, and some of the Advanced vs Advance heuristic algorithm matches. There are few exceptions in Advanced vs Advance heuristic matches observed but these exceptions might stem from the heuristic structure of the algorithm that also have been observed in other experiments as well e. g. the reaction of the algorithm is changing from winning score (190+) to losing score (20-) though with increased depth (shown in Fig. 20). In overall, the first player has a very high tendency to gain more scores and win the rounds.

Does board size influence the players' efficiency?

For the board size effects, only two algorithm's matches have been observed those are Monte Carlo Tree Search and Depth-Cut with Advanced Heuristic. The experiments indicated that increasing board sizes change the scores to the advantage of depth-cut with the advanced heuristic algorithm. The MCTS algorithm starts to lose scores when the board size is increasing. Due to the duration of the experiment, we could not gather many data; however, considering the obtained data through these experiments, change in the scores has been observed.

How the heuristic is effective on the scores against a random player or other action recommender players? And can we enhance it?

The results showed that advanced heuristic is better than all other action recommenders as a first player even using shallow depth parameters against other deeper parameters that include high rollouts. The only exception was itself. It lost some matches when played against an advanced heuristic player. As a second player, advanced heuristic, still, was powerfull enough but not as much as simple heuristic against an MCTS player.

Another important point through the experiment is that the reaction of the algorithms can show change against from one player to another player. The depth-cut algorithms did not react the same way for every player. Even the player turns and depth parameters, as mentioned before, change the reaction of the algorithm surprisingly. The experiments also showed that heuristic can be enhanced considering other game rules or pawn positions.

Can MCTS successfully be applied to the Pawn Game? How the rollout is effective on the scores against a random player or other action recommender players?

According to experiments, the rollout parameter most of the time showed that if the rollout increases, the MCTS algorithm plays more intelligently and gains more scores against other action recommenders especially a random player, which is a good test player for the experiments. Proportional increasing of the scores and rollouts has approved algorithm finds better ways to win using an adaptive strategy. When the rollout is low, it acts as a random player. Consequently, it is fair to say that MCTS has been applied successfully, and the rollout has a significant effect on a player's move for more intelligent strategies.

Is MCTS a better player than Minimax with Depth-Cut on the Pawn Game?

The tables in Fig. 19 and Fig. 20 illustrated that MCTS gathers less score than depth-cut players. Additionally, it lost all the battles in between Advance Heuristic vs MCTS, and Simple Heuristic vs MCTS, MCTS vs Simple Heuristic. Expanding board sizes also indicated that is a disadvantage of MCTS player as the algorithm started to lose scores. Consequently, all these findings proved that depth-cut methods are better players than MCTS with the given board size and game rules.

6.2. Evaluation

In the beginning, I was expecting that MCTS can beat another minimax with depth-limit heuristic algorithms; however, MCTS mostly lost the matches. Also, the expanding Pawn board was expected to be to the advantage of MCTS. The results showed the opposite of these expectations. Moreover, an advanced heuristic could have played more intelligently against all other players as the second player but it did not. The table of advance heuristic vs advance heuristic is still a mystery. The fluctuation in the scores during these matches is noteworthy. The one insight can be that since the equation of advance heuristic contains exponential expressions, and the algorithm knows opponent's players (itself) till a depth level, sudden score losing can be encountered due to player's turn. However, considering the project duration was not enough to elaborate on this mystery.

Not only the project duration, but I also could not do sufficient investigation due to computation limitations. For example, I was planning to do experiments with bigger board sizes such as 9x9 or 10x10, however, due to slow computation, I was only able to do research with 4x4 sizes that can be counted small board size. Not only this but also I could not increase rollout and depth parameters to compare in a wide range. Even depth-cut (6) and MCTS (500) were taking too much time. In 4 hours, I was only able to obtain 5-10 experiment results that run parallel way.

Additionally, parallel running was only allowed to do maximum 10-15 experiments at the same time. Consequently, acquiring experiment results took more than 2 weeks.

Chapter 7

Conclusion

7.1. Conclusion

In sum, it is a popular issue that computing all decision tree such Go, Chess or Pawn (with bigger board size) type of games due to the complexity of the paths for each decision is challenging. It is fair to say, considering experiment results, to reduce these issues, heuristic approaches are strongly necessary. Even if the computation power problem is solved, for the other complex real-life problems, heuristic always will help us to follow a short-cut way with good enough solutions under uncertainty. During the project, the computation power was enough to estimate entire moves in the game which has a deterministic structure, but heuristic implementation enabled it to play as fast as a human or in some cases faster than a human with fairly intelligent moves.

In this thesis, not only heuristic's important but also what kind of heuristic is necessary for the types of Pawn game has been discussed during the paper. The results indicated that for these type of board states and rules, Minimax with heuristic types of algorithms, horizontal search, achieve more scores than Monte Carlo Tree Search that uses probabilistic heuristic search, in other words, vertical search. However, in other types of games such as Go, Monte Carlo Tree Search has approved its success comparison in minimax with the depth-cut type of algorithms. Minimax is already not efficient to estimate all decision tree for Go type of games due to complexity, therefore, a heuristic strategy is sort of necessary for these kinds of games. Even with heuristic implementation, it could not achieve the same score such as MCTS.

7.2. Future Work

As future work, the algorithms can be compared on other types of games to have better insight for what causes depth-cut and MCTS being more powerful or less powerful. In this way, the algorithm can be analysed more objectively not only relying on one type of game such as Pawn but also other types of games that have imperfect information or stochastic environment. Experimenting on the other discrete, zero-sum, deterministic and perfect information games might give the same type of results as in this report. As a result, the environment can be altered for later experiments.

The other searching algorithms can be experimented such as SSS*, PSV and NegaScout to observe the advantages and disadvantages of these algorithms and how they react on the Pawn Game. Further, a hybrid model that includes Minimax and MCTS together can be applied just as H. Baier and M. Winands have implemented before [30]. In their study, they combine shallow depth-cut minimax search with Monte Carlo Tree Search framework to solve Connect-4, Breakthrough, Othello, and Catch the Lion games.

Additionally, a Heuristic Evaluation Framework [31] can be applied that might measure how the heuristic estimation function is good enough by considering the ground truth of the moves obtained by Minimax. Each suggested move can be compared with the best moved and output a measurable value as the outcome of this process. As I applied only two types of heuristics, which heuristic is better is usually uncertain at the beginning. During the experiment, I kept asking these question myself; are these heuristics good or representative enough? Or is there any better heuristic that needs less game information but gives better results comparison in the applied ones? Consequently, it would be an efficient idea to invent a mechanism that estimates a heuristic evaluation function's quality with only a few experiments rather than a thousand matches. That would be a supportive indicator for the heuristic analyses.

References

[1] J. Pearl, “Heuristics: intelligent search strategies for computer problem solving,” 1984.

[2] S. B.-T. Guardian and undefined 2016, “AlphaGo seals 4-1 victory over Go grandmaster Lee Sedol,” *fbe.hku.hk*.

[3] J. v. Neumann, “Zur Theorie der Gesellschaftsspiele,” *Math. Ann.*, vol. 100, no. 1, pp. 295–320, Dec. 1928.

[4] D. R. Bellhouse and N. Fillion, “Le her and other problems in probability discussed by bernoulli, montmort and waldegrave,” *Stat. Sci.*, vol. 30, no. 1, pp. 26–39, 2015.

[5] K. Fan, “Minimax Theorems,” *Proc. Natl. Acad. Sci.*, vol. 39, no. 1, pp. 42–47, Jan. 1953.

[6] M. M. Alberto Ferreira, M. Andrade, M. Cristina Peixoto Matos, J. António Filipe, and M. Pacheco Coelho, “Minimax Theorem and Nash Equilibrium,” 2012.

[7] C. E. Shannon, “XXII. Programming a Computer for Playing Chess 1,” 1950.

[8] A. Newell and H. A. Simon, “Computer Science as Empirical Inquiry: Symbols and Search,” *Commun. ACM*, vol. 19, no. 3, pp. 113–126, Mar. 1976.

[9] D. J. Edwards and T. P. Hart, “The Alpha–beta Heuristic (AIM-030),” Dec. 1961.

[10] D. E. Knuth and R. W. Moore, “An analysis of alpha-beta pruning,” *Artif. Intell.*, vol. 6, no. 4, pp. 293–326, 1975.

[11] “Dennis Breuker’s Ph.D. Thesis.” [Online]. Available: https://breukerd.home.xs4all.nl/thesis/. [Accessed: 02-Jan-2020].

[12] R. E. Korf, “Depth-First Iterative-Deepening: i z An Optimal Admissible Tree Search*.”

[13] A. Reinefeld, “An Improvement to the Scout Tree Search Algorithm,” *ICGA J.*, vol. 6, no. 4, pp. 4–14, Jan. 2018.

[14] J. Pearl, “A SIMPLE GAME-SEARCHING ALGORITHM WITH PROVEN OPTIMAL PROPERTIES,” 1980.

[15] G. C. Stockman, “A minimax algorithm better than alpha-beta?,” *Artif. Intell.*, vol. 12, no. 2, pp. 179–196, Aug. 1979.

[16] A. Plaat, J. Schaeffer, W. Pijls, and A. de Bruin, “Best-first fixed-depth minimax algorithms,” *Artif. Intell.*, vol. 87, no. 1–2, pp. 255–293, Nov. 1996.

[17] T. A. Marsland and M. Campbell, “Parallel Search of Strongly Ordered Game Trees,” *ACM Comput. Surv.*, vol. 14, no. 4, pp. 533–551, Dec. 1982.

[18] “Computer Chess Club Archives.” [Online]. Available: https://www.stmintz.com/ccc/index.php?id=86134. [Accessed: 02-Jan-2020].

[19] B. Abramson, “An analysis of expected-outcome,” *J. Exp. Theor. Artif. Intell.*, vol. 2, no. 1, pp. 55–73, 1990.

[20] D. Silver *et al.*, “Mastering the game of Go with deep neural networks and tree search,” *Nature*, vol. 529, no. 7587, pp. 484–489, Jan. 2016.

[21] L. Kocsis and C. Szepesvári, “Bandit based Monte-Carlo planning,” in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2006, vol. 4212 LNAI, pp. 282–293.

[22] “Chess Corner - Chess Tutorial - The Pawn Game.” [Online]. Available:

http://www.chesscorner.com/tutorial/basic/pawngame/pawngame.htm. [Accessed: 03-Jan-2020].

[23] “Minimax Algorithm with Alpha-beta pruning | HackerEarth Blog.” [Online]. Available: https://www.hackerearth.com/blog/developers/minimax-algorithm-alpha-beta-pruning/. [Accessed: 03-Jan-2020].

[24] S. J. Russell and P. Norvig, “1 - Introduction,” *Artif. Intell. A Mod. Approach*, p. 31, 2003.

[25] “Artificial Intelligence | Alpha-Beta Pruning - Javatpoint.” [Online]. Available: https://www.javatpoint.com/ai-alpha-beta-pruning. [Accessed: 03-Jan-2020].

[26] J. P. Fishburn, “An optimization of alpha-beta search,” *ACM SIGART Bull.*, no. 72, pp. 29–31, Jul. 1980.

[27] “File:MCTS (English) - Updated 2017-11-19.svg - Wikimedia Commons.” [Online]. Available: https://commons.wikimedia.org/wiki/File:MCTS_(English)_-_Updated_2017-11-19.svg. [Accessed: 03-Jan-2020].

[28] “Game AIs with Minimax and Monte Carlo Tree Search - Towards Data Science.” [Online]. Available: https://towardsdatascience.com/game-ais-with-minimax-and-monte-carlo-tree-search-af2a177361b0. [Accessed: 03-Jan-2020].

[29] “Monte Carlo tree search (MCTS) minimal implementation in Python 3, with a tic-tac-toe example gameplay.” [Online]. Available: https://gist.github.com/qpwo/c538c6f73727e254fdc7fab81024f6e1. [Accessed: 04-Jan-2020].

[30] H. Baier and M. H. M. Winands, “IEEE TRANSACTIONS ON COMPUTATIONAL INTELLIGENCE AND AI IN GAMES 1 MCTS-Minimax Hybrids.”

[31] N. Neší and M. S. Research Thesis, “INTRODUCING HEURISTIC FUNCTION EVALUATION FRAMEWORK,” 2016.

Appendices

Github link of the Project;

https://github.com/verdoganai/Heuristic_Analysis
[Accessed Date: 13 January 2020]

MCTS.py

```
"""
A minimal implementation of Monte Carlo tree search (MCTS) in Python 3.
Luke Harold Miles, November 2018, Public Domain Dedication

"""

from abc import ABC, abstractmethod
from collections import defaultdict
import math
from PawnLogic import Board


class MCTS:
    "Monte Carlo tree search"
    def __init__(self, exploration_weight=1):
#       super(MCTS, self).__init__(n, default_team)
        self.Q = defaultdict(int)  # total score of each node
        self.N = defaultdict(int)  # total visit count for each node
        self.children = dict()  # children of each node
        self.exploration_weight = exploration_weight

    def choose(self, node):
        "Choose the best successor of node"
        if node not in self.children:
            return node.find_random_child()
        print('choose_list', self.children[node])
        def score(n):
            if self.N[n] == 0:
                return float('-inf')
            print('self.Q[n]/self.N[n]', self.Q[n]/self.N[n])
            return self.Q[n] / self.N[n]  # average score
        return max(self.children[node], key=score)

    def do_rollout(self, node):
        "Make the tree one layer better"
        path = self.select(node)
        leaf = path[-1]
        self.expand(leaf)
        reward = self.simulate(leaf)
        self.backpropagate(path, reward)

    def select(self, node):
        "Find an unexplored descendent of `node`"
        path = []
        while True:
            path.append(node)
            if node not in self.children or not self.children[node]:
                # node is either unexplored or terminal
                return path
            unexplored = self.children[node] - self.children.keys()
            if unexplored:
                n = unexplored.pop()
                path.append(n)
                return path
            node = self.uct_select(node)  # descend a layer deeper

    def expand(self, node):
        "Update the `children` dict with the children of `node`"
        if node in self.children:
            return  # already expanded
        self.children[node] = node.find_children()

    def simulate(self, node):
        "Returns the reward for a random simulation (to completion) of `node`"
        while True:
            node2 = node.find_random_child()
            if node2 is None:
                return node.reward()
            node = node2

    def backpropagate(self, path, reward):
        "Send the reward back up to the ancestors of the leaf"
        for node in path:
            reward = 1 - reward  # 1 for me is 0 for my enemy, and vice versa

            self.N[node] += 1
            self.Q[node] += reward

    def uct_select(self, node):
        "Select a child of node, balancing exploration & exploitation"

        # All children of node must be expanded:
        assert all(n in self.children for n in self.children[node])

        log_N_vertex = math.log(self.N[node])

        def uct(n):
            "Upper confidence bound for trees"
            return self.Q[n] / self.N[n] + \
                self.exploration_weight * math.sqrt(log_N_vertex / self.N[n])

        return max(self.children[node], key=uct)
```

MCTSplayer.py

```
import random

from collections import defaultdict, namedtuple
import math
from PawnLogic import Board
import numpy as np
from unit_test import *
from random import choice
from MCTS import MCTS
import matplotlib.pyplot as plt

_PBS = namedtuple("PawnBoardState", "turnx state winner terminal")

class PawnBoardState(_PBS, Board):
    def __init__(self, turnx=None, state=None, winner=None, terminal = None):
        super(PawnBoardState, self).__init__()

    def find_children(board_state): #
        if board_state.terminal:
            return set() # remove similar nodes.
        return {
            board_state.make_move(i) for i in
range(len(super().successor_generator(board_state)))
        }

    def find_random_child(board_state):
        if board_state.terminal:
            return None
        succ_list = super().successor_generator(board_state)
        if succ_list:
            random_move = random.randint(0, len(succ_list) - 1)
            return board_state.make_move(random_move)
        else:
            raise 'No Child'
            return None

    def is_terminal(board_state): #terminal check
        return board_state.terminal

    def reward(board_state):
        # find_winner finds winner considering heuristic.
        #  As we don't have depth-cut, the heuristic function will be assigned as
100 or -100
        # that will be polarised as True (1) and False(0).
        # if it is not terminal state and returns that means it is a draw but in
our game, there is no draw.

        value = super().find_winner(board_state)
        if value == True: return 1
        elif value == False: return 0
        elif value is None: return 0.5

    def make_move(board_state, index): # this is for roll-out not human player.
        # But for testing, this can be used as a human player (defines index before
successors.)
        # using index helps us to create random children.
        state_list = super().successor_generator(board_state)
        selected_board = state_list[index]
        turnx, state = selected_board
        winner = super().find_winner(selected_board)
        is_terminal = super().terminal_state(selected_board)
        state = super().hash_converter(state)
        return PawnBoardState(turnx, state, winner, is_terminal)


def new_pawn_board(board_state, winner=None, terminal = False):
    turnx, state = board_state
    state = [tuple(l) for l in state]
    return PawnBoardState(turnx, tuple(state), winner, terminal)


def get_shape(board):
    n = np.asarray(board[1]).shape
    return n

if __name__ == "__main__":
    initial_state = (-1, [[0, 0, 0, 0, 0, 0],
                          [0, 0, 0, 1, 0, 0],
                          [0, 1, 1, 0, 0, 0],
                          [0, -1, 0, 0, -1, -1],
                          [0, 0, 0, 0, 0, 0],
                          [0, 0, 0, 0, 0, 0]])
    terminal_state = (1, [[0, 0, 0, 0, 0, 0],
                          [0, 0, 0, 1, 0, 0],
                          [0, 0, 0, 0, 0, 0],
                          [0, -1, 0, 0, -1, -1],
                         [0, 0, 0, 0, 0, 0],
                         [0, 0, 0, 0, 0, 0]])

    tree = MCTS()
    size_board = get_shape(initial_state)
    new_state = new_pawn_board(initial_state)
    fig = plt.figure()
    while True: # MCTS vs MCTS
        plt.imshow(new_state[1])
        plt.pause(2)
        print(new_state)
        for _ in range(100):
            tree.do_rollout(new_state)
        new_state = tree.choose(new_state)
        if new_state.terminal:
            break
        print('new_state after move', new_state)
        plt.imshow(new_state[1])
        plt.pause(2)
        for _ in range(100):
            tree.do_rollout(new_state)
        new_state = tree.choose(new_state)
        plt.imshow(new_state[1])
        plt.pause(2)
        if new_state.terminal:
            break
```

PawnLogic.py

```
import random
import numpy as np
import copy
import matplotlib
matplotlib.use('TkAgg')
import matplotlib.pyplot as plt
from math import sqrt

'''Board class for the game of Pawn Game.
Default board size is 6x6.
Board data:
  1=white, -1=black, 0=empty
  first dim is column , 2nd is row:
     pieces[0][0] is the top left square,
     pieces[5][0] is the bottom left square,

Squares are stored and manipulated as (x,y) tuples.

Author: Vural Erdogan, github.com/vuralerdoganai
Date: March 6, 2019.

'''

class Board(): # includes board rules and successor creator

    def __init__(self, n = (4, 4), default_team = None, turn = None, advance_option
= False):

        self.advance_option = advance_option
        # visualise all the successors if you choose none
        self.x = n[0]
        self.y = n[1]
        self.default_team = default_team
        self.turn = turn
        # Create the initial board array.
        self.pieces = [None] * self.x
        for i in range(self.x):
            self.pieces[i] = [0] * self.y
        self.pieces[0]=[1] * self.y # First team default position
        self.pieces[-1]=[-1] * self.y #second team default position


    def __getitem__(self, index):
        return self.pieces[index]

    def create_random_index(self, team_color):  # add [][] indexer syntax to the
Board
        x = random.randint(1, self.x - 2)
        y = random.randint(0, self.y - 1)
        self.pieces[x][y] = team_color

    def random_board(self): # random board state creater without 'turn'
        for i in range(self.x):
            self.pieces[i]=[0]*self.y

        random_piece_number = self.x # create maksimum 12 pieces. Max 6 for each
team.
        for i in range(random_piece_number):
            self.create_random_index(1)
            self.create_random_index(-1)

        fig2 = plt.figure(figsize=(3, 3))
        plt.imshow(self.pieces, interpolation='nearest')

    def get_pawn_positions(self): # help to get pawn positions with the team colour
'-1' or '1'
        pawn_position_list = set()
        for y in range(self.y):
            for x in range(self.x):
                if not self.pieces[x][y] == 0:
                    pawn_position_list.add((self.pieces[x][y],(x, y)))
        return list(pawn_position_list)

    def get_cordinates(self, active_pawn): # Active Pawn represents next possible
state of a pawn.
        x, y = active_pawn[1]
        return x, y

    def check_empty(self, active_pawn): # check the enemy pawn if it is empty or
not to move forward
        all_pawn_positions = self.get_pawn_positions()
        for pawn in all_pawn_positions:
            if active_pawn[1] == pawn[1] :
                return False
        return True

    def convert_board_state(self, board_state):
        turn, state = board_state
        assert self.default_team == -1, "Vural we've got a problem"
        new_board_state = np.array([np.array(xi) * -1 for xi in state])
        turn, board = -turn, new_board_state
        board = np.flipud(board)
        board = [list(x) for x in board]
        new_board_state = (turn, board)
        return new_board_state

    def get_board_coppy(self): # help to copy the initial board position.
        coppy_board = copy.deepcopy(self.pieces)
        return coppy_board
    def move_executor(self, action_tuple): # execute the move from St1 to St2.
        position1, position2 = self.get_positions_from_action_tuple(action_tuple)
        position_1_cord_x, position_1_cord_y = self.get_cordinates(position1)
        position_2_cord_x, position_2_cord_y = self.get_cordinates(position2)
        new_board = self.get_board_coppy()
        if isinstance(new_board[1], tuple):
            new_board = [list(x) for x in new_board]
        new_board[position_1_cord_x][position_1_cord_y] = 0 # delete old position
        new_board[position_2_cord_x][position_2_cord_y] = position2[0]
        return new_board

    def get_enemy_pawn(self, reference_pawn): # diagonal move
        all_pawn_positions = self.get_pawn_positions()
        new_successor=[]
        for target_pawn in all_pawn_positions: # target_pawn includes team colour
and pawn cordinates. (1, (3, 4)).
            target_pawn_team, target_pawn_cordinates = target_pawn
            reference_pawn_team, reference_pawn_cordinates = reference_pawn
            diagonal_difference = np.subtract(reference_pawn_cordinates,
target_pawn_cordinates)
            # e.g. (1, (3, 4)) vs (-1, (4, 5)). Team colour also helps us to
realise this equation.
            if (-target_pawn_team == reference_pawn_team) and
(diagonal_difference[0] == -1*reference_pawn_team) and(abs(diagonal_difference[1])
== 1):
                new_pawn_position = (-target_pawn_team,) + target_pawn[1:]
                executed_board_position = self.move_executor((reference_pawn,
new_pawn_position))
                new_successor.append((-reference_pawn_team,
executed_board_position))
                continue
        if new_successor == None:
            return False
        return new_successor

    def extract_legal_moves_considering_team(self, created_move_list, team = None):
        if team:
            created_move_list = [x for x in created_move_list if x[0] == -team]
        return created_move_list

    def successor_generator(self, *input_state):
        for state in input_state:    # this input state is optional to start from
any board state.
            self.board_position_assigner(state)
        legal_moves=[]
        all_pawn_positions = self.get_pawn_positions()

        #First move creator (forward)
        for active_pawn in all_pawn_positions:
            target_location = (active_pawn[0],
((active_pawn[1][0]+active_pawn[0]),) + active_pawn[1][1:])
            try:
                assert self.check_empty(target_location)
                # legal_moves.append((x, move1))
                new_successor = self.move_executor((active_pawn, target_location))
                legal_moves.append((-active_pawn[0], new_successor))
            except AssertionError as error:
                continue
        #Second move creator (diagonal)
        for active_pawn in all_pawn_positions:
            try:
                assert self.get_enemy_pawn(active_pawn)
                target_pawn = self.get_enemy_pawn(active_pawn)
                legal_moves.extend(target_pawn)
            except AssertionError as error:
                continue
        extracted_legal_moves =
self.extract_legal_moves_considering_team(legal_moves, self.turn)
        random.shuffle(extracted_legal_moves)
        return extracted_legal_moves

    def board_position_assigner(self, state): # assign any board position and team
as initial state.
        self.turn = state[0]
        self.pieces[:]= state[1]

    def random_move(self, list_moves): # random move creator from pawn position.
        random_move = random.randint(0, len(list_moves) - 1)
        return list_moves[random_move]

    def find_winner(self, board_state):
        if self.terminal_state(board_state):
            value = self.heuristic_value(board_state)
            return True if value > 0 else False
        else:
            return None

    def hash_converter(self, board_state):
        nested_lst_of_tuples = [tuple(l) for l in board_state]
        return tuple(nested_lst_of_tuples)

    def terminal_state(self, board_state, depth=None): # winning positions
        if depth == 0:
            return True
        turn, board = board_state[0], board_state[1]
        if -1 in board[0] or 1 in board[-1]:   # check the last rows if there is a
pawn or not. We are yellow as default.
            return True     #turn will be always "1"
        created_moves = self.successor_generator(board_state)
        if not created_moves: # no moves due to no pawn or pawn stacks
            return True
        else:
            return False

    def heuristic_value(self, board_state, depth=5e103):

        assert self.terminal_state(board_state, depth) # manhattan distance has
been used for heuristic.
        turn, board = board_state[0], board_state[1]
        if -1 in board[0]:  # check the last rows if there is a pawn or not. We are
yellow as default.
            return -(100+depth*0.001)  # turn will be always "1"
        if 1 in board[-1]:
            return 100+depth*0.001  # turn will be alway "-1"
        created_moves = self.successor_generator(board_state)
        if not created_moves:  # no moves due to no pawn or pawn stacks
            return (-(100+depth*0.001))*turn
        if depth is not 5e103:
            self.board_position_assigner(board_state)
            pawn_positions_list = self.get_pawn_positions()
            yellow_team = []
            purple_team = []
            board_size_x, board_size_y = np.asarray(board).shape
            promotion_value = 0
            for pawn in pawn_positions_list:
                if 1 == pawn[0]: # team colour is pawn[0]
                    promotion_value += pawn[1][0] - (len(board)-1)
                    yellow_team.append(pawn)
                elif -1 == pawn[0]:
                    promotion_value +=  pawn[1][0]
                    purple_team.append(pawn)
                else:
                    raise Exception('pawn list has some values different from team 
values.')

            pawn_number_difference_value = (len(yellow_team) - len(purple_team) + 
1)* abs(len(yellow_team)-len(purple_team))

            cols = [None] * len(board)
            distance_score = 0
            for row in board:
                for j, col in enumerate(row):
                    if cols[j] is None:
                        if col == turn:
                            cols[j] = 0
                    elif col == -turn:
                        distance_score += cols[j]
                    else:
                        cols[j] += 1

            distance_score = ((-1)**(distance_score+1))*distance_score

            utility_value = promotion_value + 2*pawn_number_difference_value + 
distance_score

            if self.advance_option == True:
                utility_value = pawn_number_difference_value

            return (utility_value)*turn


    def get_positions_from_action_tuple(self, positions):
        first_position = positions[0]
        second_position = positions[1]
        return first_position, second_position

if __name__ == '__main__':
        new_board = Board(default_team = 1, turn = 1, n=(5,5))
        new_board.random_board()
        successor_list = new_board.successor_generator()
        print('successors:', list(successor_list), len(successor_list))
        fig=plt.figure(figsize=(8, 5))
        row = int(sqrt(len(successor_list))) + 1
        column = int(sqrt(len(successor_list))) +1
        print(row, column)
        for i in range(1, len(successor_list)+1):
            img = successor_list[i-1][1]
            fig.add_subplot(row, column, i)
            plt.imshow(img)
        plt.show()
        # z = x.random_move(y, 1)
        # x.move_executor(z)
        # z = x.random_move(y, 1)
        # x.move_executor(z)
        # print(z)
```

players.py

```
import random
from minimax import Minimax
from PawnLogic import Board
import numpy as np
```

```
from MCTSplayer import PawnBoardState
from MCTS import MCTS

class Player:

    def minimax_player(self, state, depth=2500000, team = 1,
heuristic_parameter=True):  # creates first successors to implement minimax
algorithm
        new_shape_x = np.asarray(state[1]).shape
        player1 = Minimax(n = new_shape_x, default_team = team, advance_option =
heuristic_parameter)
        print('default_team', team, player1.default_team)
        if team == -1:
            state = player1.convert_board_state(state)
        best_move = player1.decision_maker(state, depth)
        chosen_succ, utility = best_move
        if team == -1:
            chosen_succ = player1.convert_board_state(chosen_succ)
        return chosen_succ

    def random_player(self, state):  # choose random state from successors.
        new_shape_x = np.asarray(state[1]).shape
        player2 = Board(n=new_shape_x)
        succ_list = player2.successor_generator(state)
        random_move = random.randint(0, len(succ_list) - 1)
        return succ_list[random_move]

    def human_player(self, state):
        new_shape_x = np.asarray(state[1]).shape
        player3 = Board(n=new_shape_x)
        succ_list = player3.successor_generator(state)
        print('Current State is:', state)
        for x, elem in enumerate(succ_list):
            print('{0}:{1}'.format(x, elem[1]))
        player_move = int(input('Choose your move number:'))
        return succ_list[player_move]

    def mcts_player(self, state, roll_out = 1, team = 1):
        new_shape_x = np.asarray(state[1]).shape
        player4 = Board(n=new_shape_x, default_team = team)
        if team == -1:
             state = player4.convert_board_state(state)
        turnx, board = state
        board = [tuple(l) for l in board]
        state = PawnBoardState(turnx, tuple(board), winner = None, terminal =
False)
        tree = MCTS()
        for _ in range(roll_out):
            tree.do_rollout(state)
        state = tree.choose(state)
        turnx, board = state[0], state[1]
        new_board = [list(l) for l in board]
        new_state = [turnx, new_board]
        if team == -1:
            new_state = player4.convert_board_state(new_state)
        return new_state
```

unit_test.py

```
import unittest
from PawnLogic import Board
import numpy as np
import matplotlib.pyplot as plt
class unit_testing(unittest.TestCase):
    def test_moves(self):


        reference_board_state = ([[0, 0, 0, 0, 0, 0],
                                [0, 0, 1, 1, 1, 0],
                                [0, -1, -1, -1, 0, 1],
                                [1, -1, 0, 0, 0, 0],
                                [-1, -1, 0, 1, 0, -1],
                                [0, 0, 0, 0, 0, 0]])

        board_size = np.asarray(reference_board_state).shape
        test_board = Board(n=board_size)
        teams = {'Purple': -1,
                 'Yellow': 1}

        compare_yellow_team = (teams['Yellow'], reference_board_state)
        test_board.board_position_assigner(compare_yellow_team)
        yellow_successor_list = test_board.successor_generator()

        test_yellow_team_successors = [(-1, [[0, 0, 0, 0, 0, 0], [0, 0, 1, 1, 1,
0], [0, -1, -1, -1, 0, 1], [1, -1, 0, 0, 0, 0], [-1, -1, 0, 0, 0, -1], [0, 0, 0, 1,
0, 0]]),
                                       (-1, [[0, 0, 0, 0, 0, 0], [0, 0, 1, 1, 1,
0], [0, -1, -1, -1, 0, 0], [1, -1, 0, 0, 0, 1], [-1, -1, 0, 1, 0, -1], [0, 0, 0, 0,
0, 0]]),
                                       (-1, [[0, 0, 0, 0, 0, 0], [0, 0, 1, 1, 0,
0], [0, -1, -1, -1, 1, 1], [1, -1, 0, 0, 0, 0], [-1, -1, 0, 1, 0, -1], [0, 0, 0, 0,
0, 0]]),
                                       (-1, [[0, 0, 0, 0, 0, 0], [0, 0, 1, 0, 1,
0], [0, -1, 1, -1, 0, 1], [1, -1, 0, 0, 0, 0], [-1, -1, 0, 1, 0, -1], [0, 0, 0, 0,
0, 0]]),
                                       (-1, [[0, 0, 0, 0, 0, 0], [0, 0, 1, 1, 0,
0], [0, -1, -1, 1, 0, 1], [1, -1, 0, 0, 0, 0], [-1, -1, 0, 1, 0, -1], [0, 0, 0, 0,
0, 0]]),
                                       (-1, [[0, 0, 0, 0, 0, 0], [0, 0, 1, 1, 1,
0], [0, -1, -1, -1, 0, 1], [0, -1, 0, 0, 0, 0], [-1, 1, 0, 1, 0, -1], [0, 0, 0, 0,
0, 0]]),
                                       (-1, [[0, 0, 0, 0, 0, 0], [0, 0, 0, 1, 1,
0], [0, 1, -1, -1, 0, 1], [1, -1, 0, 0, 0, 0], [-1, -1, 0, 1, 0, -1], [0, 0, 0, 0,
0, 0]]),
                                       (-1, [[0, 0, 0, 0, 0, 0], [0, 0, 0, 1, 1,
0], [0, -1, -1, 1, 0, 1], [1, -1, 0, 0, 0, 0], [-1, -1, 0, 1, 0, -1], [0, 0, 0, 0,
0, 0]])]
        self.assertEqual(sorted(yellow_successor_list),
sorted(test_yellow_team_successors))

        test_purple_team_sucessors = [(1, [[0, 0, 0, 0, 0, 0], [0, -1, 1, 1, 1, 0],
[0, 0, -1, -1, 0, 1], [1, -1, 0, 0, 0, 0], [-1, -1, 0, 1, 0, -1], [0, 0, 0, 0, 0,
0]]),
                                      (1, [[0, 0, 0, 0, 0, 0], [0, 0, 1, 1, 1, 0],
[0, -1, -1, -1, 0, 1], [1, -1, 0, 0, 0, -1], [-1, -1, 0, 1, 0, 0], [0, 0, 0, 0, 0,
0]]),
                                      (1, [[0, 0, 0, 0, 0, 0], [0, 0, -1, 1, 1, 0],
[0, 0, -1, -1, 0, 1], [1, -1, 0, 0, 0, 0], [-1, -1, 0, 1, 0, -1], [0, 0, 0, 0, 0,
0]]),
                                      (1, [[0, 0, 0, 0, 0, 0], [0, 0, 1, 1, -1, 0],
[0, -1, -1, 0, 0, 1], [1, -1, 0, 0, 0, 0], [-1, -1, 0, 1, 0, -1], [0, 0, 0, 0, 0,
0]]),
                                      (1, [[0, 0, 0, 0, 0, 0], [0, 0, -1, 1, 1, 0],
[0, -1, -1, 0, 0, 1], [1, -1, 0, 0, 0, 0], [-1, -1, 0, 1, 0, -1], [0, 0, 0, 0, 0,
0]]),
                                      (1, [[0, 0, 0, 0, 0, 0], [0, 0, 1, -1, 1, 0],
[0, -1, 0, -1, 0, 1], [1, -1, 0, 0, 0, 0], [-1, -1, 0, 1, 0, -1], [0, 0, 0, 0, 0,
0]]),
                                        (1, [[0, 0, 0, 0, 0, 0], [0, 0, 1, 1, 1, 0],
[0, -1, -1, -1, 0, 1], [-1, -1, 0, 0, 0, 0], [-1, 0, 0, 1, 0, -1], [0, 0, 0, 0, 0,
0]])]

        compare_purple_team = (teams['Purple'], reference_board_state)
        test_board.board_position_assigner(compare_purple_team)
        purple_successor_list = test_board.successor_generator()
        self.assertEqual(sorted(purple_successor_list),
sorted(test_purple_team_sucessors))
        print('succesful move test')

    def toggle(self, state):
        turn, board = state
        turn *=-1
        new_state = (turn,) + state[1:]
        return new_state

    def winning_positions(self):

        reference_board_states0 = (1, [[0, 0, 0, 0, -1, 0],
                                       [0, 0, 0, 0, 0, 0],
                                       [0, 0, 0, 0, 0, 0],
                                       [0, 0, 0, 0, 0, 0],
                                       [0, 1, 0, 0, 0, 0],
                                       [0, 0, 0, 0, 0, 0]])

        reference_board_states1 = (-1,[[0, 0, 0, 0, 0, 0],
                                      [0, 0, 0, 0, -1, 0],
                                      [0, 0, 0, 0, 0, 0],
                                      [0, 0, 0, 0, 0, 0],
                                      [0, 0, 0, 0, 0, 0],
                                      [0, 1, 0, 0, 0, 0]])

        reference_board_states2 = (1, [[0, 0, 0, 0, 0, 0],
                                      [0, 0, 0, 0, 0, 1],
                                      [0, 0, 0, 1, 0, -1],
                                      [0, 1, 0, -1, 0, 0],
                                      [0, -1, 0, 0, 0, 0],
                                      [0, 0, 0, 0, 0, 0]])

        reference_board_states3 = (-1,[[0, 0, 0, 0, 0, 0],
                                      [0, 0, 0, 0, 0, 0],
                                      [0, 1, 0, 1, 0, 0],
                                      [0, 0, 0, 0, 1, 0],
                                      [0, 0, 0, 0, 0, 0],
                                      [0, 0, 0, 0, 0, 0]])

        reference_board_states4 = (1, [[0, 0, 0, 0, 0, 0],
                                       [0, 0, 0, 0, 0, 0],
                                       [0, -1, 0, -1, 0, 0],
                                       [0, 0, 0, 0, -1, 0],
                                       [0, 0, 0, 0, 0, 0],
                                       [0, 0, 0, 0, 0, 0]])

        board_size = np.asarray(reference_board_states0[1]).shape
        test_board = Board(n=board_size, default_team = 1)
        teams = {'Purple': -1,
                 'Yellow': 1}

        terminal_state_value = test_board.heuristic_value(reference_board_states0)
        self.assertEqual(terminal_state_value, -5e100) # Reference high number for
checking.
        terminal_state_value = test_board.heuristic_value(reference_board_states2)
        self.assertEqual(terminal_state_value, -5e100)

        terminal_state_value = test_board.heuristic_value(reference_board_states4)
        self.assertEqual(terminal_state_value, -5e100)

        print('Test Yellow Team done')

        terminal_state_value = 
test_board.heuristic_value((reference_board_states1))
        self.assertEqual(terminal_state_value, 5e100)

        terminal_state_value = 
test_board.heuristic_value(self.toggle(reference_board_states2))
        self.assertEqual(terminal_state_value, 5e100)

        terminal_state_value = test_board.heuristic_value(reference_board_states3)
        self.assertEqual(terminal_state_value, 5e100)
        print('Test Purple Team done')

        print('unit test succesful')
```