Hanoi University of Science & Technology

School of Information & Technology

================88================

INTRODUCTION TO ARTIFICIAL INTELLIGENCE

CAPSTONE PROJECT REPORT

An intelligent agent that plays Checkers

**Students:**

Le Trung Kien - 20214907

Hoang Minh Hoang - 20214900

Nguyen The An - 20210006

Do Dinh Kien - 20214906

**Class**: 136462

**Lecturer**: Dr. Nguyen Nhat Quang

Table of Contents

[Introduction 2](#_Toc123512229)

[1. What is Checkers? 2](#_Toc123512230)

[2. History of Checkers 2](#_Toc123512231)

[3. Rules of Checkers 2](#_Toc123512232)

[4. Problem description 5](#_Toc123512233)

[Algorithms 5](#_Toc123512234)

[1. Random 6](#_Toc123512235)

[2. Minimax 6](#_Toc123512236)

[3. Minimax variants 9](#_Toc123512237)

[i. Alpha-beta pruning 9](#_Toc123512238)

[ii. Negamax and Negamax with Alpha-beta pruning 14](#_Toc123512239)

[Heuristics function ideas 15](#_Toc123512240)

[1. Naive heuristic 16](#_Toc123512241)

[2. A bit more complex heuristic 17](#_Toc123512242)

[3. Our final heuristic 18](#_Toc123512243)

[Analysis 19](#_Toc123512244)

[1. Comparing between minimax and alpha beta using same heuristic function 19](#_Toc123512245)

[2. Comparing between different heuristic functions 23](#_Toc123512246)

[Conclusion and further improvements 25](#_Toc123512247)

[References 28](#_Toc123512248)

[Appendix: Project Contribution 29](#_Toc123512249)

# Introduction

## What is Checkers?

[Checkers](https://www.britannica.com/topic/checkers), which is also known as Draughts, is a strategy board game that involves diagonal moves of game pieces and capture of opponent’s pieces by jump moves.

The game is played by two players with 12 pieces each. The name Draughts means “to draw” or “to move”. The main objective is to capture as many of the opponent’s pieces as possible until the opponent has no more potential moves. Also, another objective is to get to the opponent’s side of the board in order to crown their pieces king along the way.

## History of Checkers

**While the oldest known Checkers artifact unearthed in Iraq dates back to 3000 BC, the earliest known version of the game comes from a game called Alquerque, which was played in ancient Egypt in 1400 BC. After years of being played the Egyptian way, in 1100 AD, a Frenchman adapted it to be played on a Chessboard with 12 pieces per player.**

Around the middle of the 1500s, it was common to see books written on the game. This continued until the 1750s when an English mathematician ([William Payne](https://www.rct.uk/collection/1074703/an-introduction-to-the-game-of-draughts-containing-fifty-select-games-together)) created official rules of Checkers in a treatise that he wrote. Once this was done, the game became known as Checkers in the United States of America, but was called (and still is) Draughts in the United Kingdom.

With actual rules in place and an official name, Checkers grew in popularity and was eagerly played by many. It became so popular that the first Checkers World Championship was held in [1840](http://www.wcdf.net/champions_hist.htm). It was around this time that people started paying more attention to the technicalities of the game and how strategy can be used to create an advantage.

## Rules of Checkers

There are many variations of Checkers that are played around the world but in our project, we will only consider the most casual variant that are stated below.

Checkers is a strategy board game for two players with a classic 8x8 chessboard, only the dark squares are used. It is positioned so that each player has a light square on the right-side corner. Each player will have 12 pieces (discs) each. Typically, the discs are flat and round. The color of one set is black and the other red or white or beige.

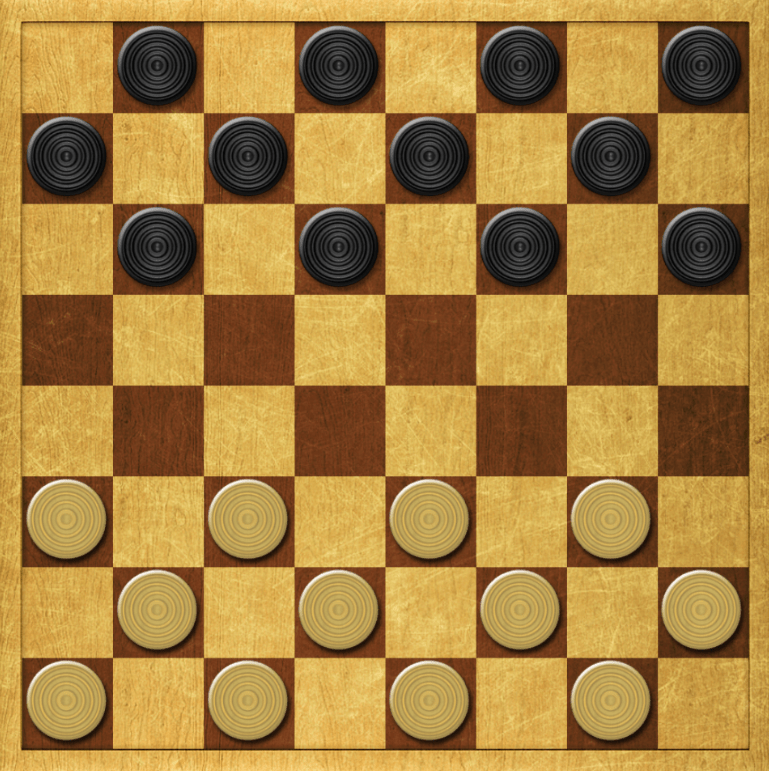


Figure 1: A Checkers board

Each player places their discs on the 12 dark squares closest to him or her. White opens the game, then players alternate turns. In our game black will be “o” and white will be “x” respectively.

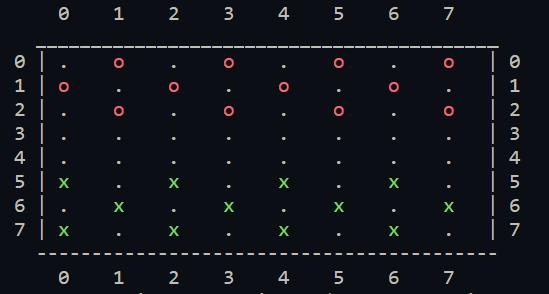


Figure 2: Our Checkers board

The pieces always move diagonally, and single pieces are always limited to forward moves. A piece making a non-capturing move may only move one square.

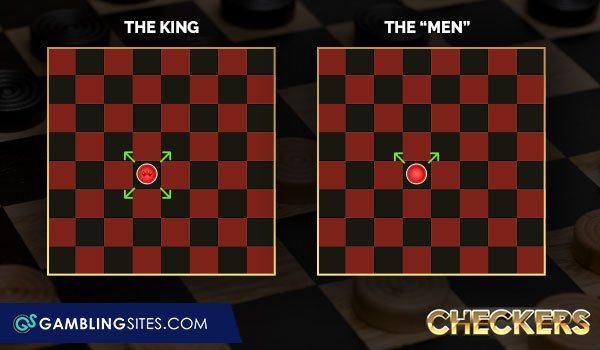


Figure 3: Moves for normal piece

To capture a piece of your opponent, your piece leaps through the opponent’s piece and lands in a straight diagonal line on the other side (the landing square must be empty). When a piece is captured, it is removed from the board. Only one piece can be captured in a single jump, but multiple jumps can be made in 1 turn. The capture is compulsory for some variations of the game but in our version, you can choose whether forced capture is applied.

When a piece reaches the furthest row, it is crowned and becomes king. The king is still limited to moving diagonally but can move both forward and backward.

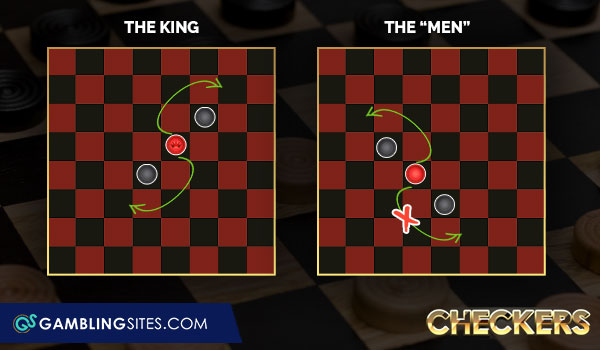


Figure 4: How a normal piece captures vs how a king capture

The game ends when one of the players cannot make a move. If during 50 moves (25 by each player), no piece has been captured then the game will be considered a tie.

## Problem description

We will implement an intelligent agent that can play Checkers. This agent has the following properties:

* **Performance measure**: high winning rate, good computation time, ability to play moves like experts
* **Environment**: board, game rules, black and white pieces, players
* **Actuator**: make a move
* **Sensor**: observe the state of the board

The game is perfect information, competitive multiagent, deterministic, static, discrete, and sequential.

Based on the game rules, this is a zero-sum game. With the goal to win the game by capturing more pieces and preserving our own, our agent is goal-based agent.

From those properties, the agent uses adversarial search

The game can be represented as follows:

* Initial state
  + An 8x8 matrix BOARD, filled with “.” for blank squares, “x” for white pieces (“X” if king), “o” for black pieces (“O” if king).
* Players: Player 1 – White (Human) and Player 2 – Black (AI)
* Successor function: A list of available legal moves and corresponding state of that turn
* Goal test: if either player has no morel legal moves
* Evaluation: the scores of both players

# Algorithms

To implement the agent, we will use the below algorithms:

* Random strategy
* Minimax with strategy– based evaluation heuristic function

In our project, we will mainly focus on the Minimax algorithm. Random will be used as benchmark for the performance and effectiveness of Minimax algorithm.

## 1. Random

As the name suggests, this algorithm chooses a move out of a set of valid moves randomly.

Time complexity and space complexity: O(1) (assume that the implementation of random function is negligible). We use the random library in Python to implement the algorithm.

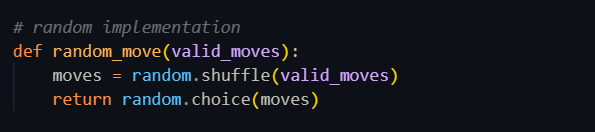


Figure 5: Strategy implementation with random library in Python

In our program, we shuffle the valid\_moves set first to avoid repetitive results.

## 2. Minimax

Since Checkers is a two-player, deterministic, fully observable, zero-sum game, Minimax is a reasonable approach for this problem.

Minimax algorithm is a recursive algorithm which is used in decision-making and game theory especially in AI game. It provides optimal moves for the player, assuming that the opponent is also playing optimally. For example, considering two opponents: ***Max*** and ***Min*** playing. ***Max*** will try to maximize the value, while ***Min*** will choose whatever value is the minimum. The algorithm performs a **depth-first search** (DFS) which means it will explore the complete game tree as deep as possible, all the way down to the leaf nodes. The algorithm is shown below with an illustrative example.

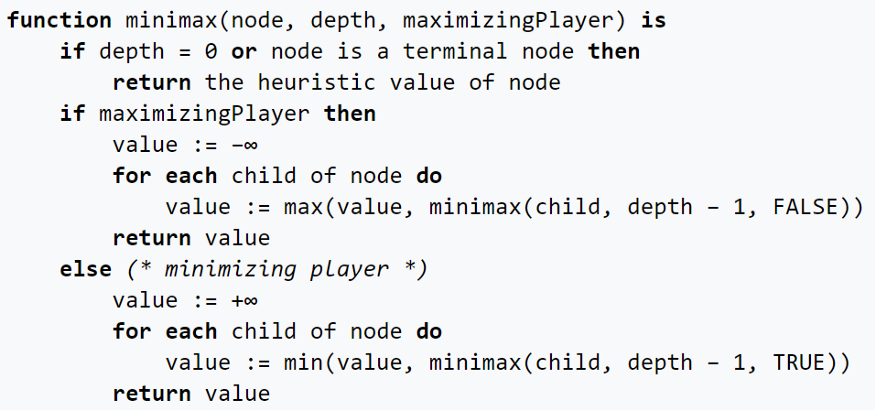


Figure 6: Minimax pseudocode

Initially, the algorithm generates the entire game tree and produces the utility values for the terminal states by applying the utility function. For example, in the below tree diagram, let us take A as the tree's initial state. Suppose maximizer takes the first turn, which has a worst-case initial value that equals negative infinity. Then, the minimizer will take the next turn, which has a worst-case initial value that equals positive infinity.

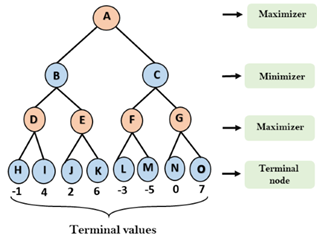
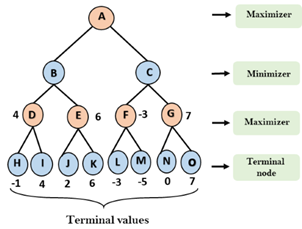
 

Figure 7: Search tree example

The above example demonstrates clearly how Minimax calculates the best move with a depth of 4 (the algorithm searches 4 moves ahead). We consider the third row, which is the Maximizer’s turn.

* D = max(-1, 4) = 4
* E = max(2, 6) = 6
* F = max(-3, -5) = -3

G = max(0, 7) = 7

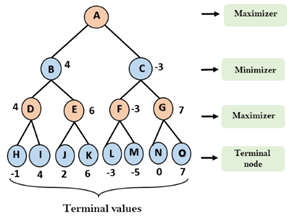
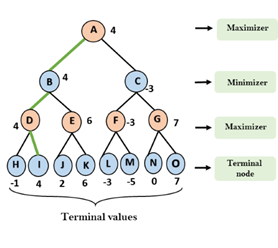
 

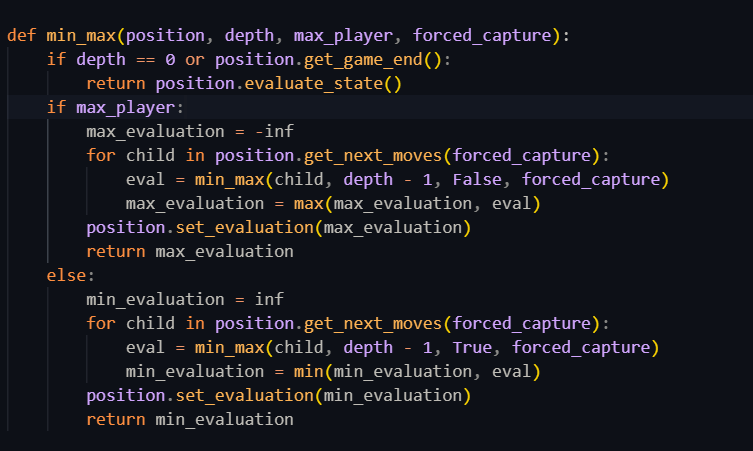
Figure 7: Search tree example

Now it’s the Minimizer's turn:

* B = min(4, 6) = 4
* C = min(-3, 7) = -3

Finally, the Maximizer determines its best move: max(4, -3) = 4. So given that the Minimizer plays the optimal move, the path of the game will be A à B à D à I.

Below is our implementation of the Minimax algorithm:

Figure 8: Minimax implementation

In our implementation, the min\_max function takes 4 arguments:

* **position**: the state of the game at the time this function is called.
* **depth**: referes to the number of future moves to evaluate in the game tree. In our program, we will have depth=6 to balance the computation time.
* **max\_player**: set to True if Black (maximizing), False if White (minimizing).
* **forced\_capture**: whether we want compolsury capturing in our game (if there exists a capturing move then we must make that move). Usually this is set to True.

The function returns the evaluation of the state, or the heuristic value of the state (position.evaluate\_state). We will discuss how we computed the heuristic value in later parts of the report.

**Time complexity**: O(bm), where *b* is the branching factor of the game tree, in this case the number of possible moves for each turn; *m* is the depth of the game.

**Space complexity**: O(bm), based on depth first search (DFS).

## 3. Minimax variants

### Alpha-beta pruning

Since Minimax has to explore all the branches in the game tree, the computation time becomes extremely expensive. However, in practice, this is not necessary as we do not need to explore all the branches. While we are exploring, we will see that there are branches that surely cannot improve the result. Therefore, we can “prune” them to save time without affecting the solution.

We will set 2 extra arguments to the function: **alpha (α)** and **beta (β).** Alpha denotes the best move’s value explored for the MAX player, while beta denotes the best move’s value explored for the MIN player.

Initially, we will set α = **-∞,** β = **+∞.** Here is the pseudocode for Minimax with Alpha-Beta pruning

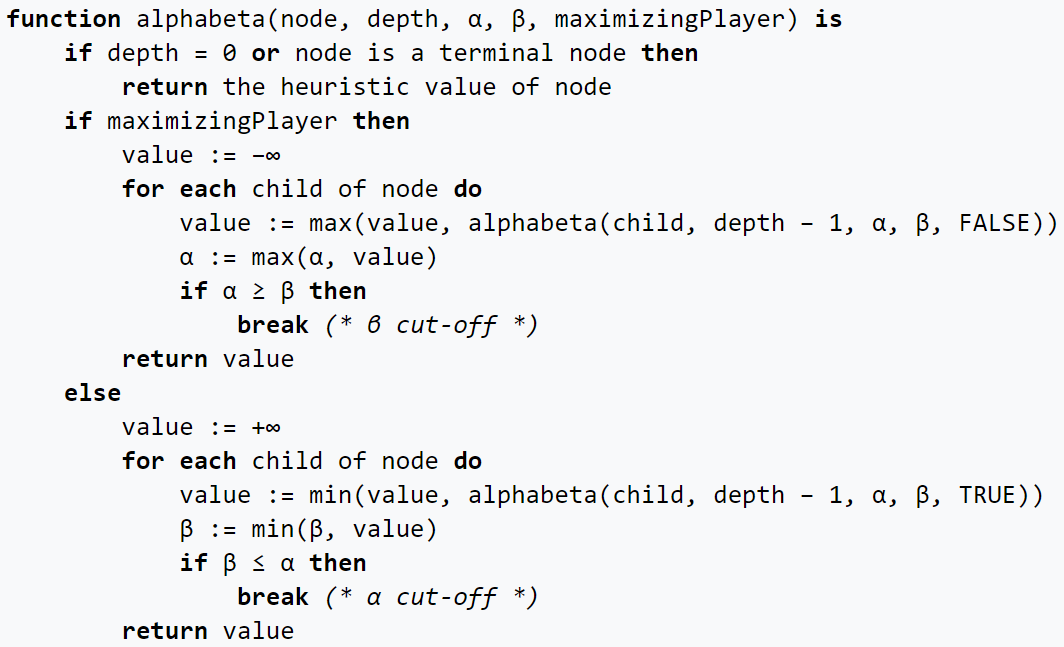


Figure 9: Minimax w/ AB pruning pseudocode

We will dive deeper into how AB can reduce the computation time. Let’s consider the following example:

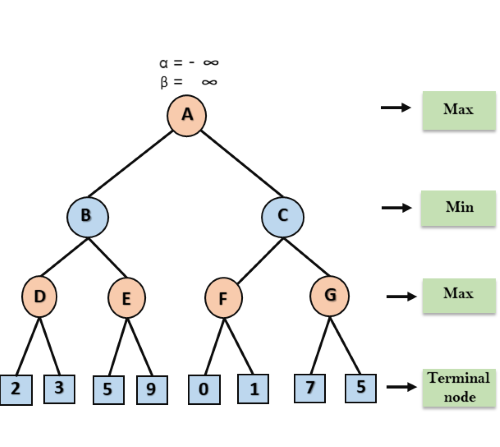


Figure 10: Search tree w/ AB pruning example

At first, we pass down α = **-∞,** β = **+∞** from node A down to B then D.

At node D, the value of α is calculated as it’s MAX turn. The value of α is compared to 2 then 3, then max(2,3) = 3 will be the value of the node D and α=3.

Now it backtracks to B, where β = min(3, **+∞)** as it’s MIN turn, hence now at node B α = **-∞,** β = 3.

At node E, Max will take its turn, and the value of α will change. The current value of alpha will be compared with 5, so max(**-∞,** 5)=5. Hence at node E α=5, β = 3. Since β<= α, then the right successor of E will be pruned, and the algorithm will not traverse it, and the value at node E will be 5.

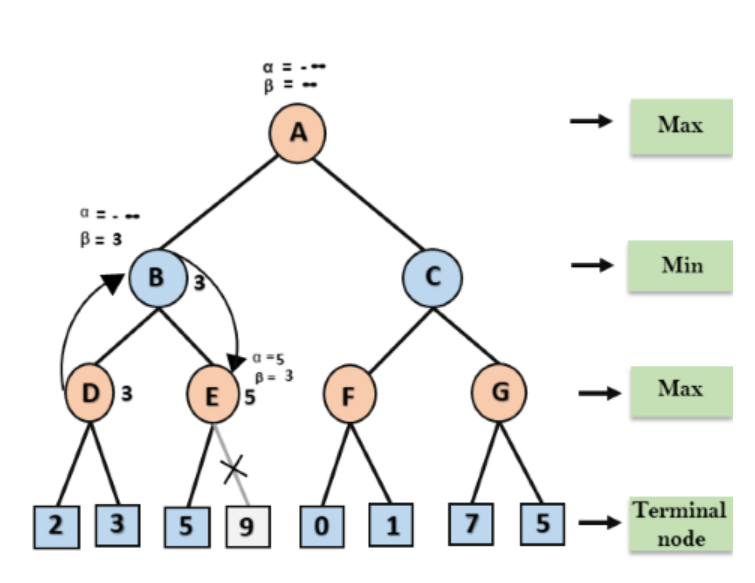


Figure 11: Search tree w/ AB pruning example

At the next step, the algorithm backtracks from node B to node A, where now the value at node A. At A, the value of α=max(- **∞**,3)=3, and β= **+∞**, these two values now pass to right successor of A which is node C.

At node C, Min will take it turn, α=3 and β= **+∞**, and the same values will be passed into node F.

At node F, again the value of α will be compared with the left child which is 0, and max(3,0)=3, and then compared with right child which is 1, and max(3,1)=3, so now the value of α =3, but the value of node F is 1.

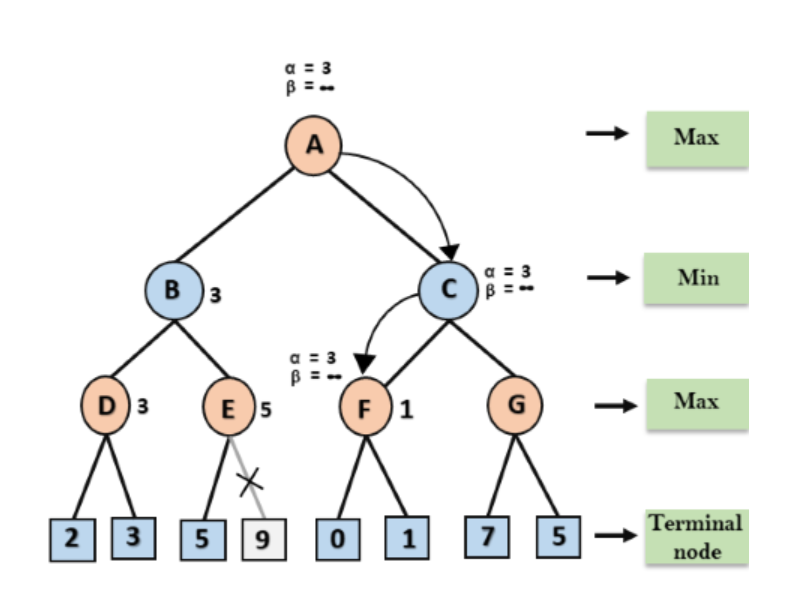


Figure 12: Search tree w/ AB pruning example

Node F returns value 1 to node C, at C α =3 and β = **∞**, now β =min(1, **∞**)=1. Now at C, α =3 and β =1 ,and now we get β <= α, so the next child of C is G will be pruned, and the algorithm will not compute the entire sub-tree G.

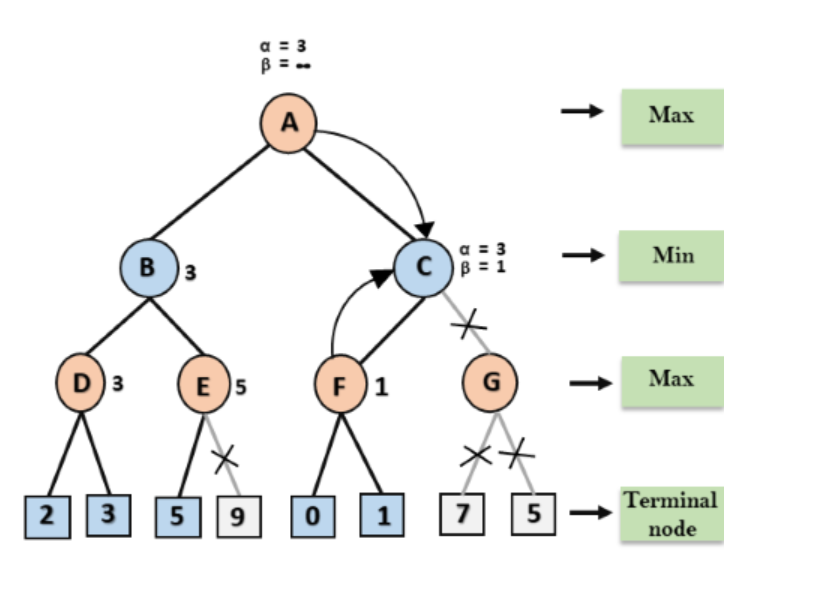


Figure 13: Search tree w/ AB pruning example

C now returns the value of 1 to A. Here the best value for A is max(3,1)=3. Now we get the final game tree as following image:

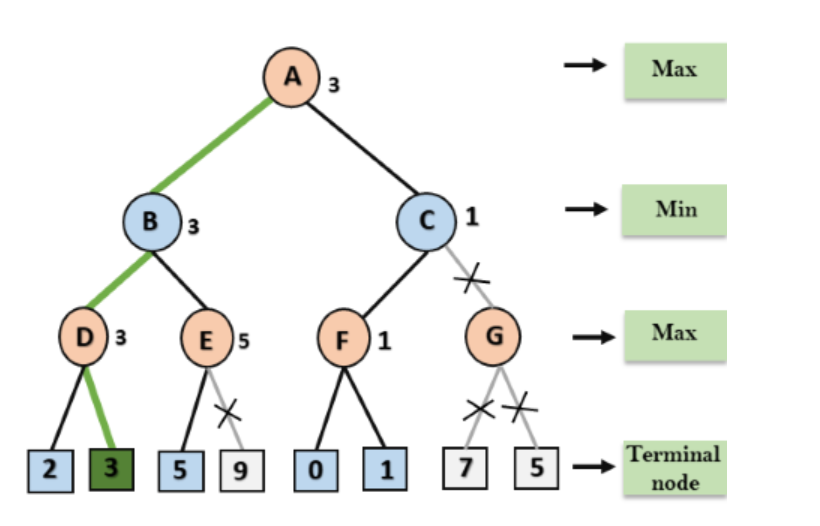


Figure 14: Search tree w/ AB pruning example

Here is our implementation. We use the math library in Python to denote **∞** as inf:



Figure 15: Minimax w/ AB pruning implementation

**Time complexity**: depends on the order of moves consider.

* In worst case, AB pruning consider an order of moves where no pruning is possible, making it just like normal Minimax: **O(bm)**
* In best case, the number of pruning is optimal: **O(bm/2)**

### Negamax and Negamax with Alpha-beta pruning

This is a simplified variant of Minimax. The idea of this variant comes from the following feature:

By using the equation, the if-else statement in Minimax can be replaced by a single line of code as shown in the figure:

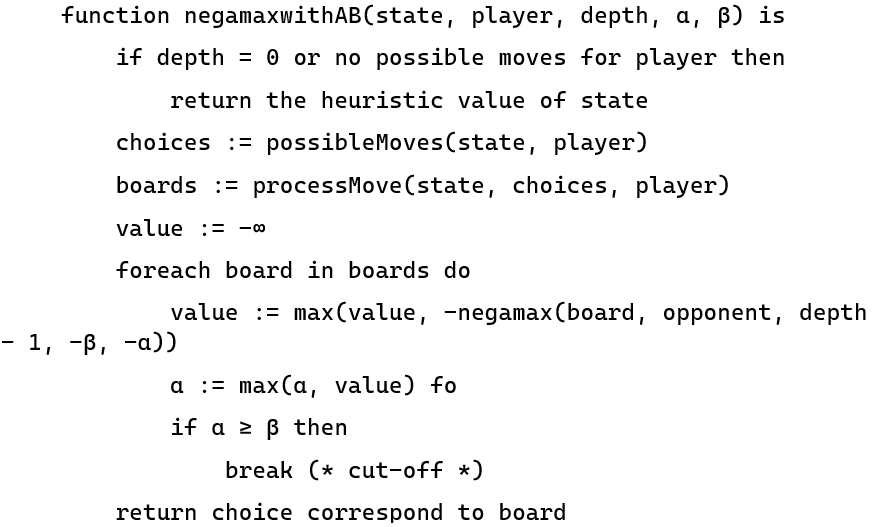


Figure 16: Negamax with AB pruning pseudocode

Since Negamax is basically a simplified version of Minimax and the computation time remains the same, we do not implement it in our project.

# Heuristic function ideas

As we mentioned before, heuristic function is necessary as this is how the Minimax algorithm will evaluate the state of the game in order to make the best choice possible.

As a matter of fact, in practice, heuristic function is the element that separates a great AI to a decent one since Minimax accuracy can only reach a certain limit, as there is a depth limit- how far the search will expand. This limit exists due to the fact that exploring the whole game tree is nearly impossible and impractical.

Below are some of the heurist’/ic functions that we have tried. From now on we will denote **Bn - Wn = fn** with:

* **Bn:** the number of black pieces corresponding to the n-th feature of the heuristic
* **Wn:** the number of white pieces corresponding to the n-th feature of the heuristic
* **fn**: the score (piece number difference between black and white) corresponding to the n-th feature of the heuristic

## Naive heuristic

This heuristic function only counts the number of pieces on the board, applying different weights for kings and pawns respectively. We consider a king to be twice as valuable compared to a pawn. The evaluation can be expressed as follows

* B1 = number of black pawns
* W1 = number of white pawns
* B2 = number of black kings
* W2 = number of white kings

à

(f1 = B1 - W1, f2 = B2 – W2)

Below is our implementation:

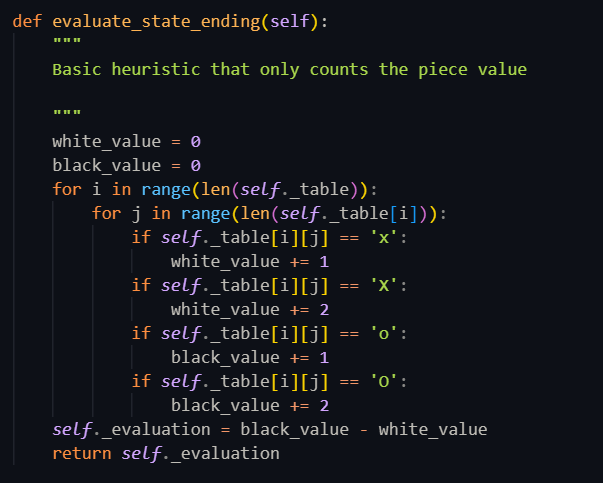
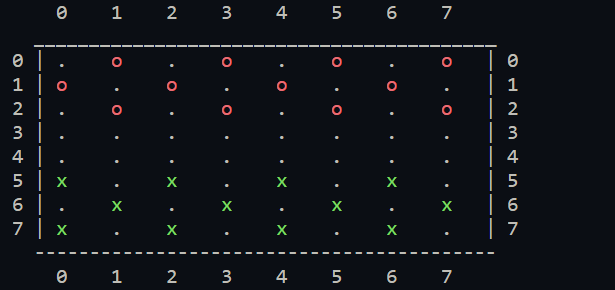


Figure 17: Naïve heuristic implementation

This heuristic function makes the AI decent but there are still many ways to improve it since humans can still beat it quite regularly.

## A bit more complex heuristic

The next heuristic we tested is still considered basic but considers the piece’s position as well. Here is our game board:

We can see that whoever controls the middle of the board i.e row 3, 4 and column 2, 3, 4, 5, will have an advantage since they can be more flexible in moving and capturing the opponents’ pieces. We denote the black pawns (o) and white pawns (x) in this area as ***B1*** and ***W1*** respectively.

Additionally, pieces about to approach and in the opponents’ territory i.e white pieces from row 3 to 0, black pieces from row 4 to 7, can become kings much easier so it’s reasonable for them to be more valuable than other pieces. We denote them ***B2*** and ***W2***, corresponding to black and white pawns in these areas. Other pawns will be ***B3*** and ***W3*** accordingly.

Finally with the kings, they are more valuable than any other pieces so the number of kings will be ***B4*** for black and ***W4*** for white.

After considering these attributes and their influence on the game, we decide their best weights are (5, 4.5, 4, 6).

The final evaluation will be:

The AI fairs pretty well. When playing AI vs AI, the latter heuristic had a small edge over the first, producing a slightly better win rate. It also coped really well against humans.

## Our final heuristic

The previous functions are all basic heuristics. Humans could still beat the AI if played correctly, so we know our heuristic could be better.

After consulting sources from the Internet (<https://www.wikihow.com/Win-at-Checkers> and <https://github.com/kevingregor/Checkers/blob/master/Final%20Project%20Report.pdf>), we came up with a set of new features for the heuristic function. W will stand for number of whites, B for number of blacks, numbers with them denoting the features we are considering (e.g W1, B1). Our function now includes weights in this order:

* **Number of regular “pawn” pieces** (**)**
* **Number of king pieces** *()*
* **Number of pieces in the back row** (keep your pieces in the final row to prevent your opponent from getting a king and to prevent your pieces from being taken) ()
* **Number of pieces in the middle 4 columns and middle 2 rows** (controlling the middle of the board) ()
* **Number of pieces in the middle 2 rows but not in the middle 4 columns** ()
* **Number of pieces that can be taken by the opponent on the next turn** ()
* **Number of pieces that cannot be taken until pieces behind it (or itself) are moved** ()

To evaluate our heuristic function, we simply play it against the AI with previous heuristics and ourselves. Since we need different weights for each feature of the heuristic function to determine which feature is more important than the others, we tested out different weights and choose whatever has the highest win rate to be our “best” heuristic in our environment.

After multiple tests, we found what appeared to be the “optimal” weights for our heuristic: **(5, 7.75, 4, 2.5, 0.5, -3, 3),** format as(pieces, kings, back row, mid box, mid rows, vulnerable, protected). Refer to the prementioned report ([click here](https://github.com/kevingregor/Checkers/blob/master/Final%20Project%20Report.pdf)) for more detail on how we can come up with the weights.

The final evaluation will be:

This heuristic function is proved to be the most efficient with much higher winrate when playing against the three mentioned. formers. If you want our detail implementation, please refer to the source code.

# Analysis

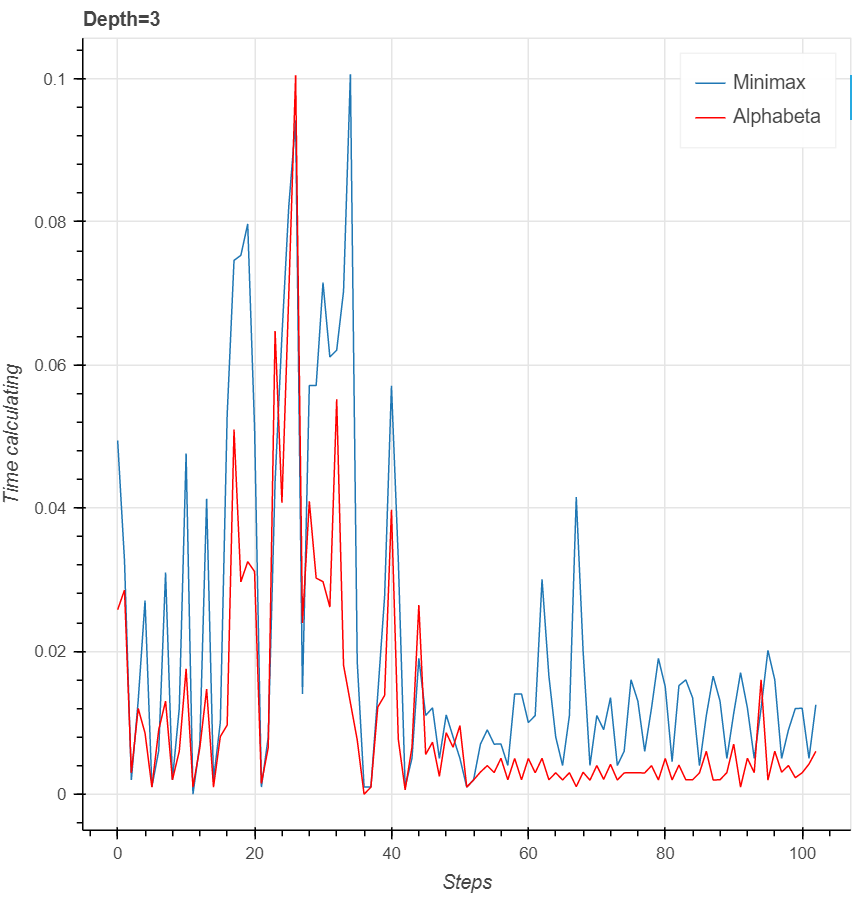
To analyze the performance of Minimax and our heuristic function, we have set up the matches as follow:

* AI versus AI: We let the algorithm play with each other. In our implementation, we record the differences between time calcuting when using various algorithms.
* In our project, we start with initial depth 3 for each algorithm. By using three heuristic functions, we have tested several times in these following cases:
  + Minimax using complex heuristic function named ‘evaluate\_state’.
  + Alpha-beta using complex heuristic function named ‘evaluate\_state’.
  + Alpha-beta using naive heuristic function named ‘evaluate\_state\_ending’.
  + Alpha-beta using best heuristic avaiable named ‘heuristic’.

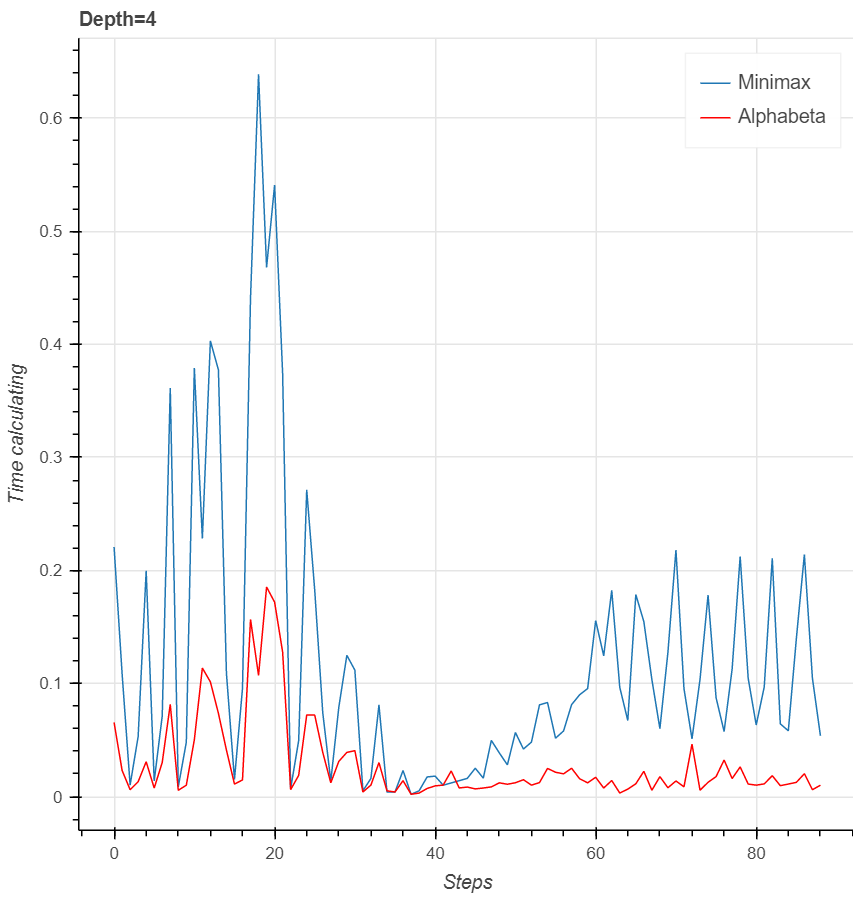
## Comparing Minimax with and without AB pruning

In this section, we conduct to observe how it reduces time for calculating when use Alpha - Beta pruning compared to minimax, applying the heuristic function ‘evaluate\_state’. Since our Alpha - Beta pruning implementation uses the same heuristic evaluation function, in every turn we also calculate the time and number of visited game nodes that both algorithms have used.

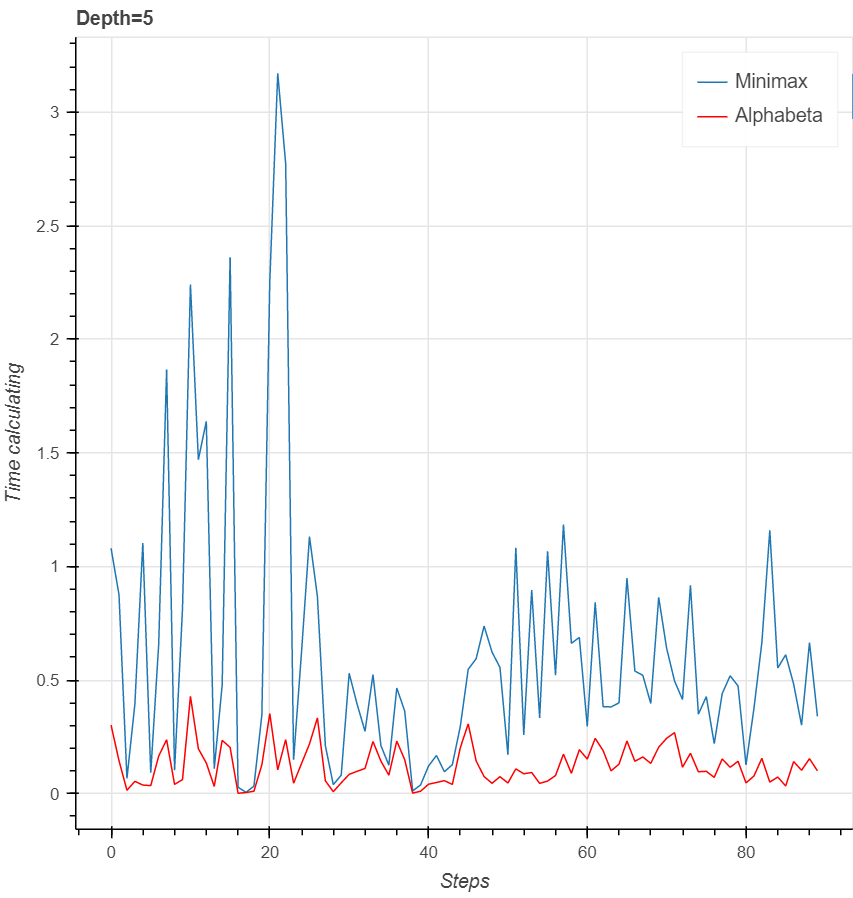
Time calculating in recorded:



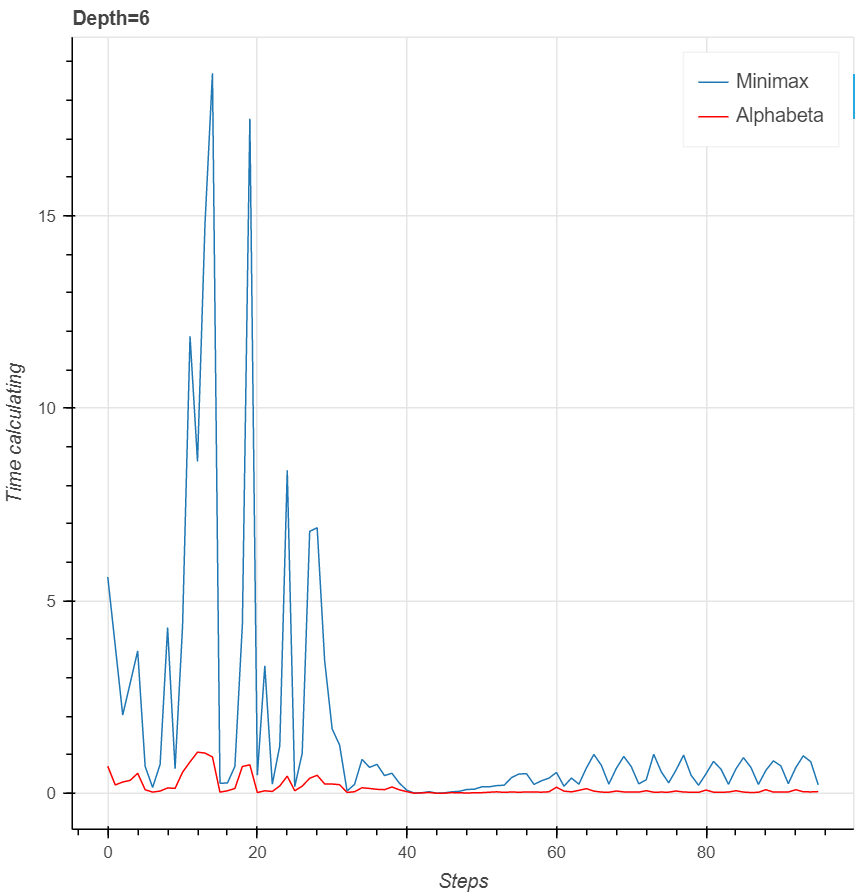
*Plot1.Comparison with depth=3*



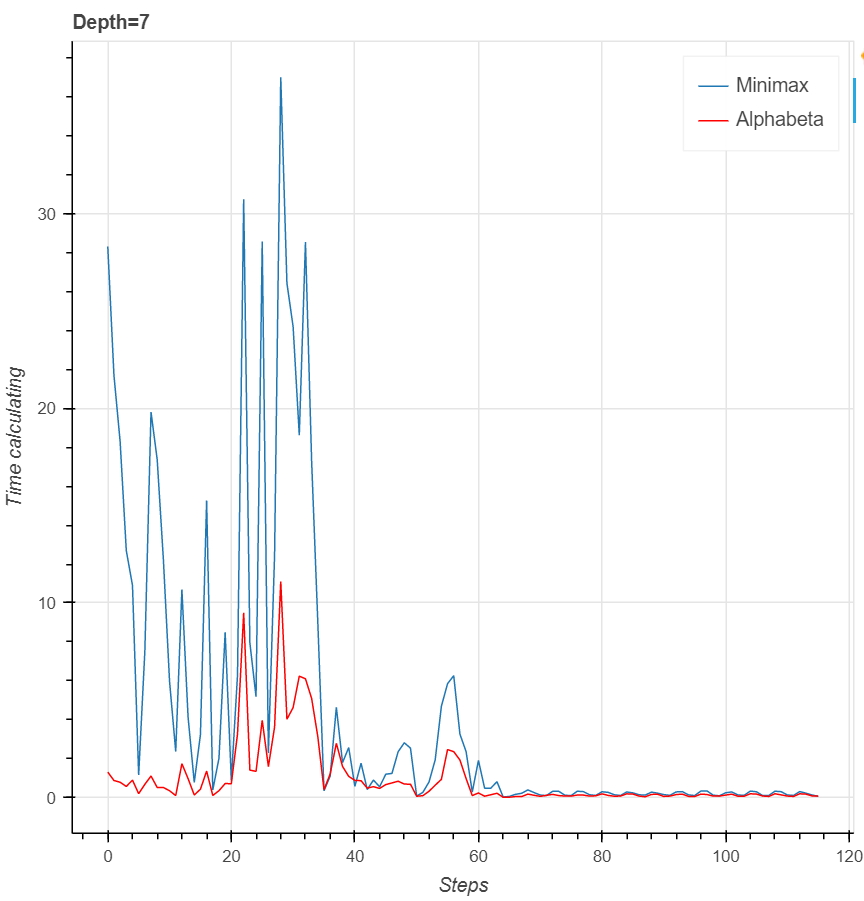
*Plot2. Comparison with depth=4*



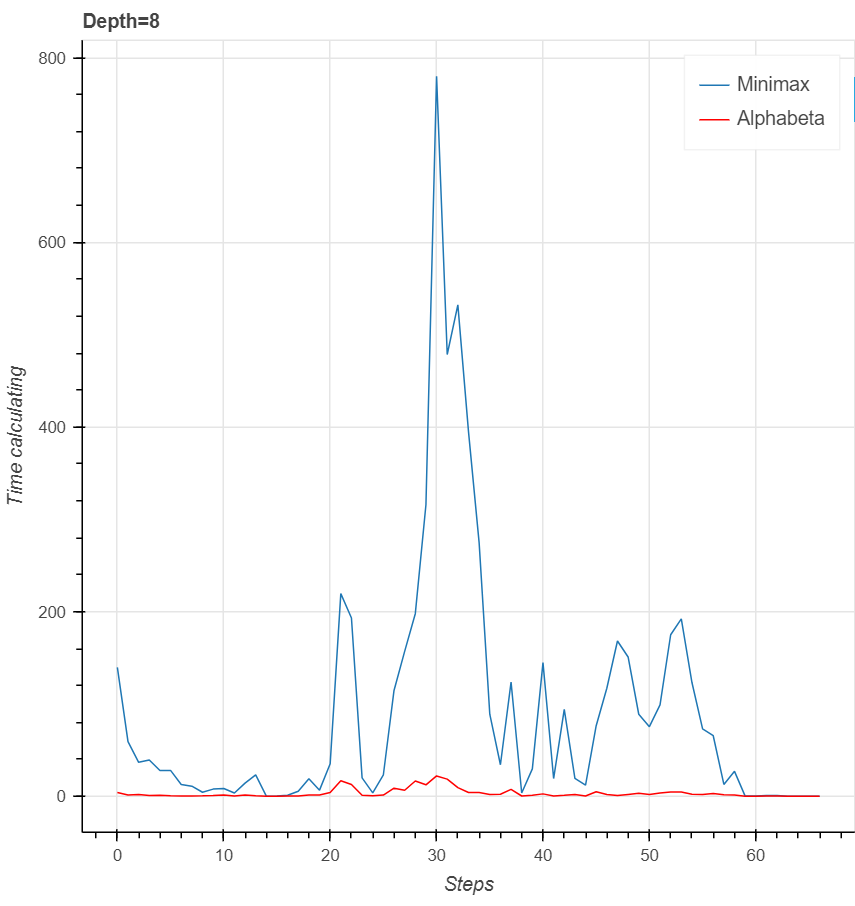
*Plot3.Comparison with depth=5*



*Plot4.Comparison with depth=6*

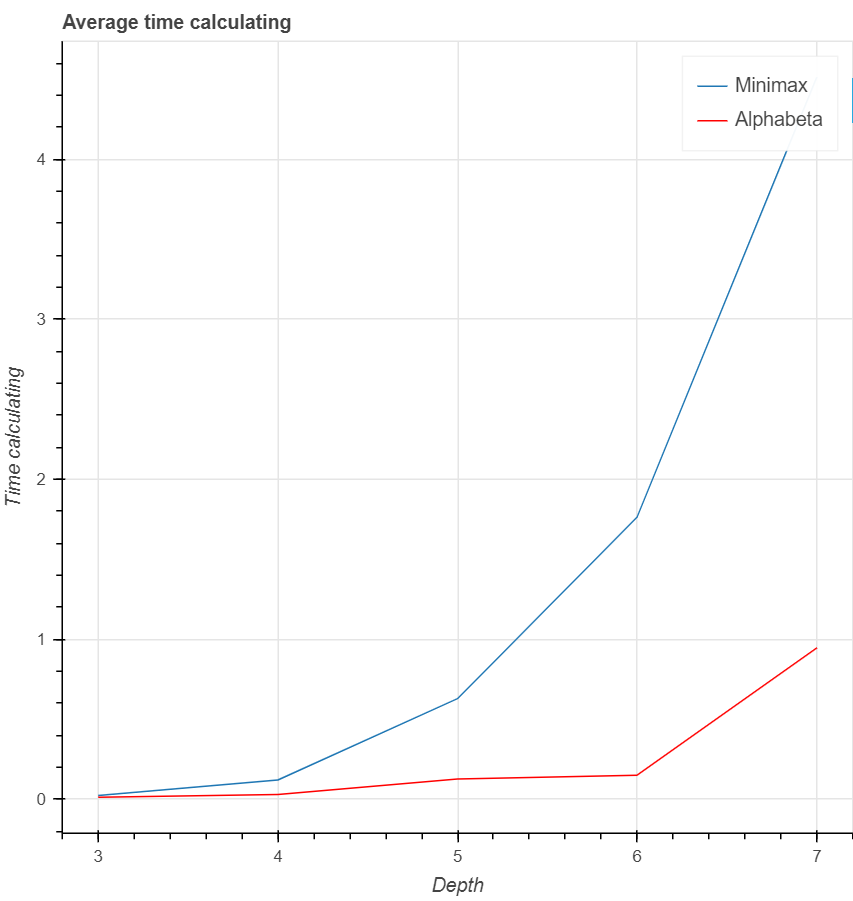


*Plot5.Comparison with depth=7*



*Plot6.Comparison with depth=8*

Now we get average calculating time:



*Plot7. Comparing average calculating time*

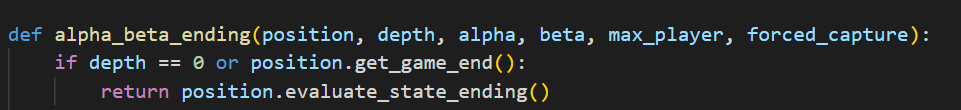
With the data collected from conducting implementation, we calculate average time for each algorithm to see how far has it reduced when using Alpha - Beta pruning.

Obviously, Minimax takes far more time space when calculating compared to Alpha - Beta pruning. With increasing depth, nearly double the steps will be considered, so it would take a large amount of time to run through all data, while Alpha - Beta pruning will ignore unnecessary nodes. Hence with bigger depth, the difference between two algorithms becomes more significant.

## Comparing different heuristic functions

From the above section, we can claim that Minimax with AB pruning is more efficient than normal Minimax. However, we have tried to find out some factors that can contribute to the success of the game. To see that our final heuristic is the optimal, we conduct the second observation.

First, we set the naive heuristic function ‘evaluate\_state\_ending’ for COM player.



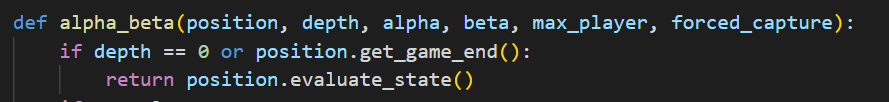
* **AI versus human**:

After setting the naive heuristic function for COM player, we try to play directly with it. After playing 20 games, we get the following result:

|  |  |  |  |
| --- | --- | --- | --- |
| **Human won** | **Draw** | **COM won** | **Total games** |
| 3 | 5 | 12 | 20 |

With win rate 60%, this heuristic function could win player at intermediate level. This winrate can be ‘accepted’ since we only consider the number of white and black left on the board.

Next, we will consider the winrate of AI if we use a more complex function called ‘evaluate\_state’ , which we mentioned in the above section. This function will not only consider the number of black and white left on the board, but also evaluate whether the position of each piece is good or not.

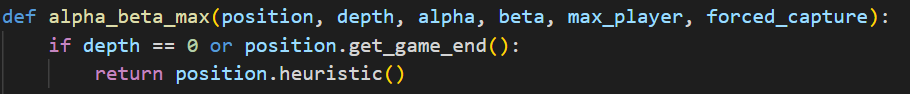


After setting this heuristic function for COM player, we continue to play 20 games with AI player and get the following result:

|  |  |  |  |
| --- | --- | --- | --- |
| **AI won** | **Draw games** | **Human won** | **Total games** |
| 15 | 4 | 1 | 20 |

When played with this COM player, we found it more difficult when AI player tried to keep it number of pieces, besides it always tried to ‘go with group’, means they always stay at adjacent position in diagonal direction, or remained at its postion at two sides so that we can’t find way to kill it.

Finally, we tried to play with our most optimal heuristic function, called ‘heuristic’.



|  |  |  |  |
| --- | --- | --- | --- |
| **AI won** | **Draws** | **Human won** | **Total games** |
| 18 | 2 | 0 | 20 |

This AI player guarantees a larger winrate compared to two aboved AI players. The reason is that it has more criteria that can correctly evaluate the state of the game.

* **AI versus AI**:

After playing directly with AI player, we make both Player1 and Player2 become AI player to see the differences between separate heuristic functions.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Player 1** | **Player 2** | **P1 won** | **Draw** | **P2 won** | **Total games** |
| Naive | Medium | 5 | 5 | 10 | 20 |
| Naive | Optimal | 2 | 3 | 15 | 20 |
| Medium | Optimal | 4 | 3 | 13 | 20 |

After conducting , we claim these two conclusions :

* The optimal heuristic function ‘heuristic’ has far more significant winrate compared to ‘evaluate\_state\_ending’ and ‘evaluate\_state’.
* Although the winrate has increased when using the optimal heuristic function, it can’t not guarantee a 100% win for each game, since it can only make decision at level of individual moves, while human play differently – they consider a higher-level goal, they use the goal to create plausible plans, ….

# Conclusion and further improvements

In this project, we have studied adversarial search algorithm Minimax by implementing it in the game Checkers. We have analyzed the game features to come up with a heuristic function to evaluate the state of the game. We observed the performance of Minimax in different matches against various opponents, and through observations, we have some comments regarding Minimax in our game:

* Our heuristic evaluation function is not entirely optimal. We find out that in final stages of the game, when there are only a few pieces left and the AI is winning, when we try to move a piece in and out of the corner of the board, the AI would in most cases unable to lure the piece out, even though we humans can clearly see there are moves available. One of the reasons could be that the AI may have to sacrifice 2 pieces in order to get 1 piece of the opponent, and this contradicts its heuristic function.
* Minimax works correctly given the opponent also plays optimally according to Minimax’s evaluation. However, we can clearly see that our first 2 heuristics or humans don’t always produce an optimal move. The suboptimality of the opponent is a problem of Minimax.

We have some suggestions to improve this project:

* Improving the heuristic function, considering the phase of the game (beginning, middle, end) to make better decision. In our game, we have tried using the optimal heuristic until there are less than 6 pieces left, then we switch to the first heuristic that only counts the pieces. However, the result doesn’t bring much difference
* Minimax is a **type A** strategy, which considers all of the possible moves. We can try using **type B** strategies, which ignore all the moves that look bad and follow the branch that looks as “reasonable” as far as possible.
* A more advanced search algorithm can be used, such as the Monte-Carlo Tree Search, … Also, we can consider using Machine Learning techniques such as Reinforcement Learning, Deep Learning… In practice, this has been applied to “solve” Checkers and Chinook is a shining example of these techniques.

During this project, we have experienced many difficulties in our group project, and we have learnt many things by solving them. Four of the biggest struggles we have to deal with are differences in schedule that make us hard to work on the project together; finding a way to improve our heuristic function; game code that we referenced was not fully implemented and the lack of resources about the game.

So, this is how with the problems, one by one:

* About the difference in schedule, it’s hard for us to reschedule everything because most of them are fixed, so we came up with an idea that is using Google Meets to make online conferences. From there we can still stay at home and share our work together.
* About improving the heuristic function, we didn’t have any ideas about it at first. But after doing some research and many tries, we realized that there are some important things that we haven’t considered such as the stages of the game or the positions of pieces. And then we came up with the article containing heuristic function idea that we mentioned above.
* About the game code, originally the code did not implement the game as we wanted it. There was no multiple capturing, so we had to implement that ourselves, which takes up a lot of time. WAe had to look through the game G.U.I, rules code to find where we needed to fix to make our game similar to international Checkers rules.
* About the lack of resources, the only way to solve this is to spend more time working on it by ourselves.

These are some of the most notable difficulties we met in the process of this project, and we learnt valuable lessons from them. This is why a group project is better than normal exams since it forces us to do more research outside the class and deal with problems that we will probably face in later projects.

# References

[1]: “History of checkers easily explained with pictures,” [Online]. Available: <https://www.gamesver.com/history-of-checkers-easily-explained-with-pictures/>

[2]: “Checkers”, [Online]. Available: <https://www.gamblingsites.com/skill-games/checkers/>

[3]: T. Q. Khoat, "Lecture slide of Introduction to Artificial Intelligence course, IT3160E," [Online]. Available: <https://users.soict.hust.edu.vn/khoattq/lectures/IT3160E-AI/>

[3]: “Minimax algorithm in AI,” [Online]. Available: <https://www.javatpoint.com/mini-max-algorithm-in-ai>

[4]: Jonathan C.T Kuo, “Artificial Intelligence at Play – Connect Four (Minimax algorithm explained),” [Online], 2020. Available: <https://medium.com/analytics-vidhya/artificial-intelligence-at-play-connect-four-minimax-algorithm-explained-3b5fc32e4a4f>

[5]: “Negamax,” [Online]. Available: <https://en.wikipedia.org/wiki/Negamax>

[6]: “Win at checkers,” [Online]. Available: <https://www.wikihow.com/Win-at-Checkers>

[7]: Kevin Gregor, “CS4701: Final Project Report Optimal Heuristic For Checkers,” [Online], 2017. Available: <https://github.com/kevingregor/Checkers/blob/master/Final%20Project%20Report.pdf>

[8]: Marko Njegomir, Checkers game G.U.I, rules, …. Available: <https://github.com/njmarko/alpha-beta-pruning-minmax-checkers>

# Appendix: Project Contribution

|  |  |
| --- | --- |
| Name | Tasks |
| Lê Trung Kiên | Analyze game strategies, search document, implement Minimax and Alpha beta, implement gameplay, set up AI vs human matches, implement optimal heuristic function, format report, leader. |
| Nguyễn Thế An | Implement gameplay, search document, implement Minimax and Alpha beta, analyze game strategies, analyze heuristic function, set up AI vs AI matches, collect record, presenter. |
| Hoàng Minh Hoàng | Implement gameplay, implement Alpha beta, analyze and test with heuristic function, set up AI vs human matches and collect record, plot graphs and analyze result, format report, presenter. |
| Đỗ Đình Kiên | Implement gameplay, implement Minimax, set up AI vs AI matches and collect record, analyze and test with heuristic function, format slide, presenter. |