VILNIUS UNIVERSITY
FACULTY OF MATHEMATICS AND INFORMATICS
INSTITUTE OF COMPUTER SCIENCE
DEPARTMENT OF COMPUTATIONAL AND DATA MODELING

Final Bachelor Thesis

Board Game AI Development Using Alpha-Beta Pruning

Done by:
Lukas Klimas signature

Supervisor:
J. Asist. Rokas Astrauskas

Vilnius
2020

Contents

Abstract

The main goal of the thesis was to create an Artificial Intelligence solution for a board game "Gipf" using alpha-beta pruning algorithm. This algorithm is precomputing game strategies for every possible step prior to next turn. The board game itself is an abstract type, thus do not have any random or luck based events and relies on player strategies. The game was created using Unity software. For comparison reasons two solutions were created and results compared using two algorithms - Alpha-Beta pruning and Minimax. Created solution was tested against other AI and randomized bot.

Santrauka

Stalo žaidimo dirbtinio intelekto kūrimas naudojant alfa beta paiešką

Pagrindinis šio darbo tikslas buvo sukurti dirbtinį intelektą stalo žaidimui "Gipf", naudojant "Alpha-Beta pruning" algoritmą. Prieš sekantį ėjimą, šis algoritmas iš anksto apskaičiuoja žaidimo strategijas kiekvienam įmanomam žingsniui. Pats stalo žaidimas yra abstraktaus tipo, todėl jame nėra jokių atsitiktinių ar sėkme pagrįstų įvykių, todėl jis priklauso nuo žaidėjų strategijų. Žaidimas buvo įgyvendintas naudojant "Unity" programinę įrangą. Palyginimo sumetimais buvo sukurti du sprendimai ir palyginti rezultatai, naudojant du algoritmus - "Alfa-Beta pruning" ir "Minimax". Sukurtas sprendimas buvo išbandytas su kitu dirbtiniu intelektu ir atsitiktinius ėjimus darantį botą.

Introduction

Nowadays, video game industry is rapidly improving and gaining more and more interest. This industry inspires innovation by constantly pushing the boundaries of what is possible, thus encourage people to create new technology. Because of this rapid change, a lot of traditional strategy board games are loosing the interest. The creation of digitized versions with Artificial Intelligence would reintroduce those games to a big audience of people.
A good example of digitalized board game is Chess. With a lot of different implementations of this game use Minimax or Alpha-Beta pruning algorithms [8]. Minimax is an algorithm which begins with the assumption that the AI will always move to maximise the score given by the evaluation function, and the opponent will always move to minimise it. For each node, we can recursively define its score to be the maximum of its children's scores if it is the max-player's turn, or minimum otherwise. The leaves are scored by the evaluation function [9].
Alpha beta pruning is an improved Minimax algorithm. The algorithm does not calculate all nodes - cuts the nodes of the tree which are not affecting possible result.[9]

In this thesis:

- Board game: "Gipf" - this section will describe the development of the board game "Gipf" using Unity software and also will explain all the rules of the game.
- Artificial Intelligence section will describe two algorithms (Alpha-Beta pruning and Minimax) and their implementation capabilities.
- Solution variations and testing - will describe the differences between two algorithms by also giving some insights on artificial intelligence test cases used.

1 Board game: "Gipf"

1.1 Introduction

"Gipf" [2] is an abstract strategy board game for 2 players based on a simple concept: players introduce one piece into play, one after another. After achieving four-in-a-row, players remove their row and capture any of their opponent's pieces which extend that row.
The goal of the game is to create rows of at least 4 pieces, until the opponent's resources are depleted (do not have pieces to play). The game is a part of a "Gipf" [2] board game series and was published by "Don & Co" and designed by Kris Burm in 1997 (the game and the series have the same name).

1.2 Rules

The board is a hexagon that contains a pattern of lines (see Figure 1).
There are 24 dots at the edges of the pattern on the board. Each dot serve to position a piece before bringing it into play, however they are not a part of the play area. The play area is made up of 37 line intersections also known as spots (the central part of the board). Only the pieces covering a spot are part of the game. The lines indicate the directions in which pieces may be moved.

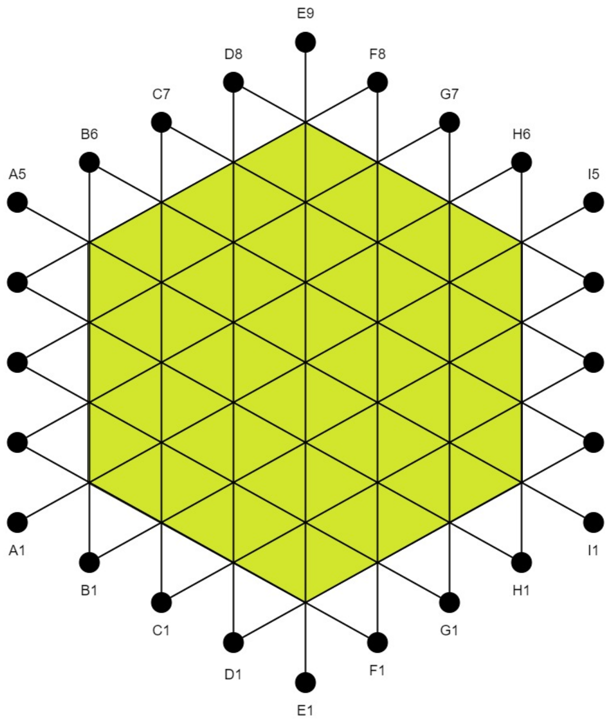

Figure 1. "Gipf" game board

At the start of the game players have 18 pieces per player while more pieces can be used either to play a longer game or to equate the strength of the players (stronger player has less).
White always begin the game. The board must be positioned between the players while the dot marked "E1" points at the player with the white pieces. The game starts with 3 pieces from each player on the board and counted as already in play (see Figure 2).

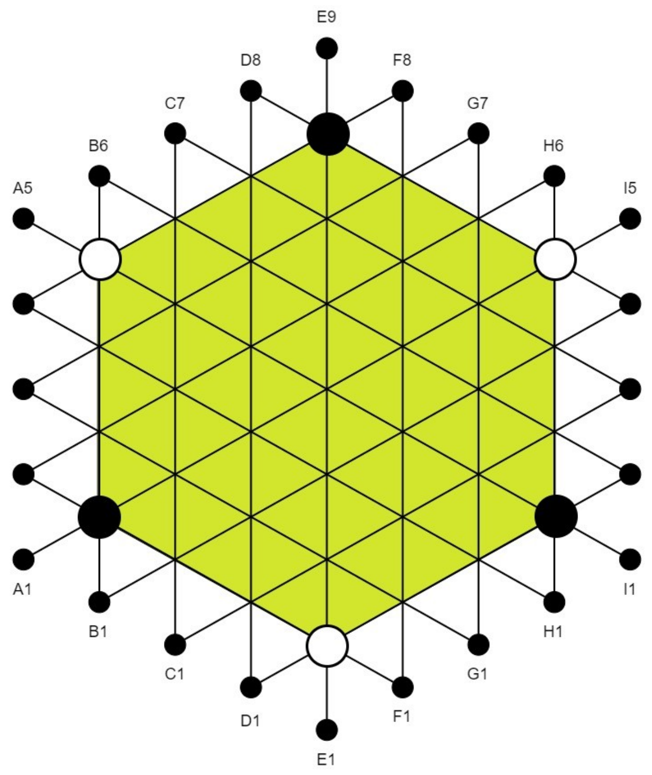

Figure 2. Starting board

A player must add one piece to the game and this must be done in 2 steps: one takes a piece from his reserve and places it on any of the 24 dots (step 1) and then moves it onto a spot in the play area (step 2).
A piece is not in play as long as it is covering a dot, meaning that the turn isn't completed. One has two options for the second step:

- A piece can be pushed from a dot onto a free spot in the play area. As soon as it moves in the direction of a spot, it cannot be changed. The move must be made.
- A piece can be pushed from a dot to a spot already occupied by another piece. In this case, the occupied spot must first be cleared: the piece occupying it (regardless of its color) must be moved to the next spot on the line. It repeats until one piece moves onto the cleared spot.

Note that all the pieces are moved in one and the same direction (see Figure 3). Once a player has touched a piece in the play area, the move must be completed. Piece can only be moved one spot at a time, never two or more without pushing other piece out of the play area (onto the dot at the opposite far end of a line).

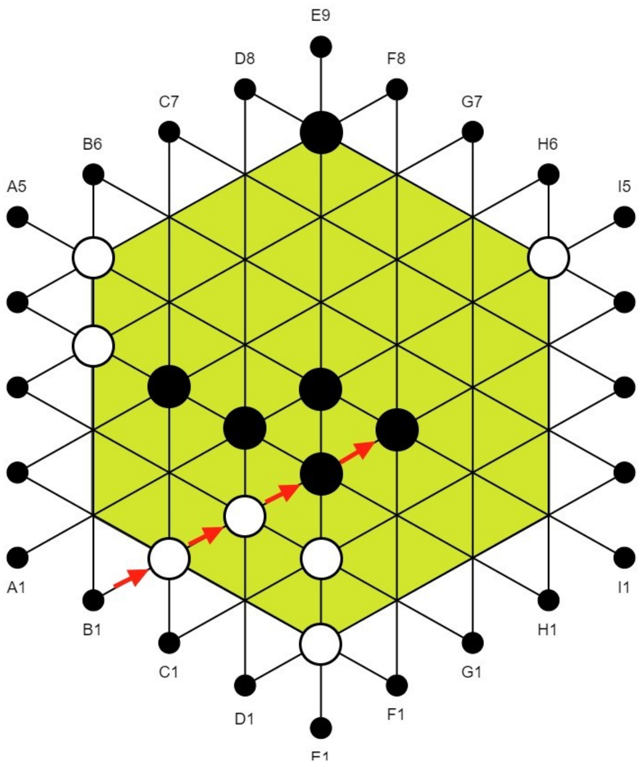

Figure 3. Movement of the pieces

As soon as 4 pieces of the same color are lined up next to each other, they must be taken from the board by the one playing with that color, while also taking all of the pieces that form a direct extension of them.
The pieces of the removing player's own color are returned to the reserve and the opponent's pieces are captured, thus removing the from the game.
Occurrences, of two rows of the same color are lined up at the same time, can be solved in 2 ways:

- if they do not intersect, both must be captured.
- if they intersect, the player whose turn may choose which row he will take

. When a situation occurs in which both players must take pieces, the player who caused the situation removes first.

The game ends when one of the players cannot bring a piece into play. Thus, if a player has no pieces left in his reserve, the other player is the winner.

1.3 Game creation

1.3.1 Tools

The primary software used for this project was Unity[3]. It is an IDE (integrated development environment), designed for game, application development, but also used in movie and architectural model creation. This IDE provides its own game engine and a lot of functionality that simplifies some parts of the game development.
The C# (C Sharp), the widely used programming language for Unity[3], was used for the scripts, that are operating the games functionality.
Unity provides a lot of flexibility when deploying the project, because it supports different types of platforms like: Windows, Linux, Android, IOS and more.

1.3.2 Gameplay

Starting the game, the player will appear at the main menu.
After pressing the play button, the selection menu appears, where the player can select to play against other player or the Artificial intelligence. After selecting one of the options, player will reach the main window (see Figure 4).

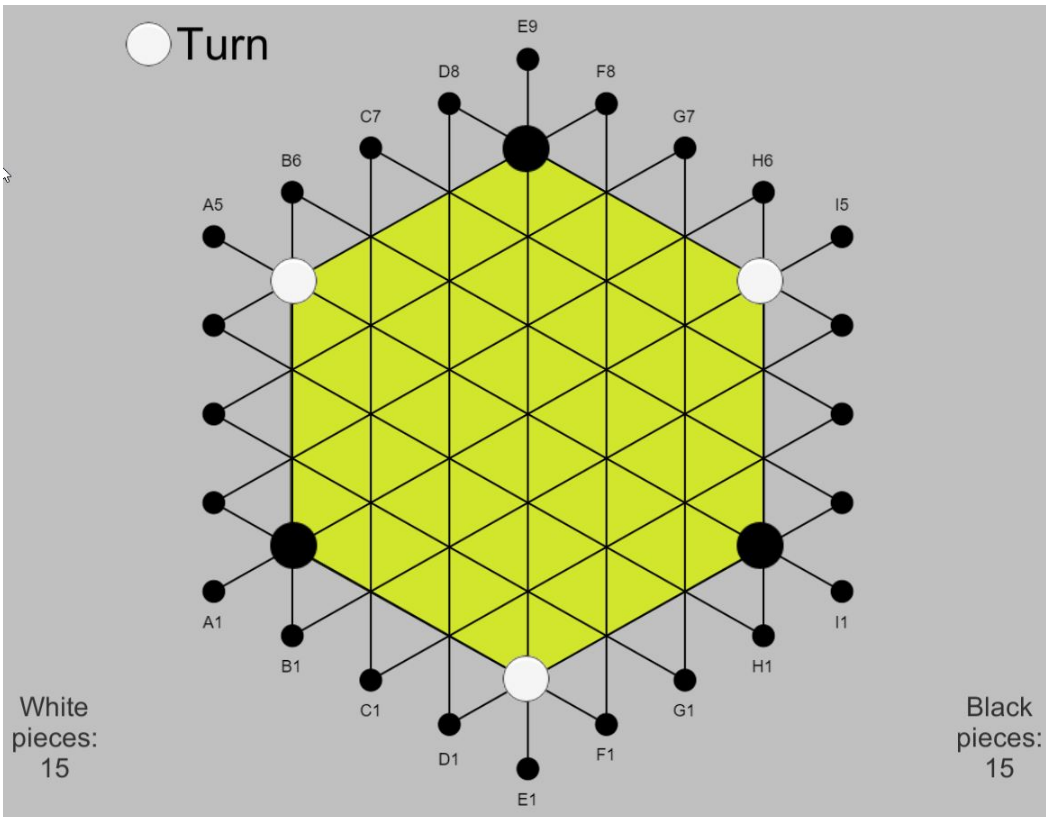

Figure 4. The main gameplay screen

On the top left, there is an indicator showing whose turn it is by showing the color of the piece. At the bottom right of the screen player will be able to see the amount of pieces in the black piece reserve, while on the bottom left side - white reserve.
The reserve amounts and turn indicator updates after every turn. A player can place a piece, by first pressing on one of the black dots. The players piece will appear on that dot. Afterwards, a second click is required on one of the intersections that are connected by a line thus making the piece move to that location.

1.3.3 Board

One of the unique features of "Gipf"[2] board game is its hexagon shaped board.
Because of the complex look of the board, it was hard to generate it from scratch.
Board was created using game object system provided by Unity engine [3], so the game board implementation was simplified.
The image of the board was placed on the flat panel object that is always represented as background on the game. For the board to be interactable, an invisible game object was placed on each playable spot (hereafter referred as positions) where the game piece can exist (black dots and line intersections) thus acting as viewpoints on the screen for pieces to be placed on.
The hexagon shape provided some difficulties while trying to find efficient way to enumerate all board positions. However, the solution was reached after reading a guide "Hexagonal grids" written by Amit Patel [4], that explained how to implement hexagonal grids in games.
In the Figure 5, the "Gipf"[2] board was represented as a hexagonal grid, where each grid peace was equivalent to a position in the original board (grey hexagons represent black dots in the original board). Then each grid peace was given three coordinates x, y and z. The only constraint when giving values to coordinates was that the sum of x, y and z must be equal to 0.

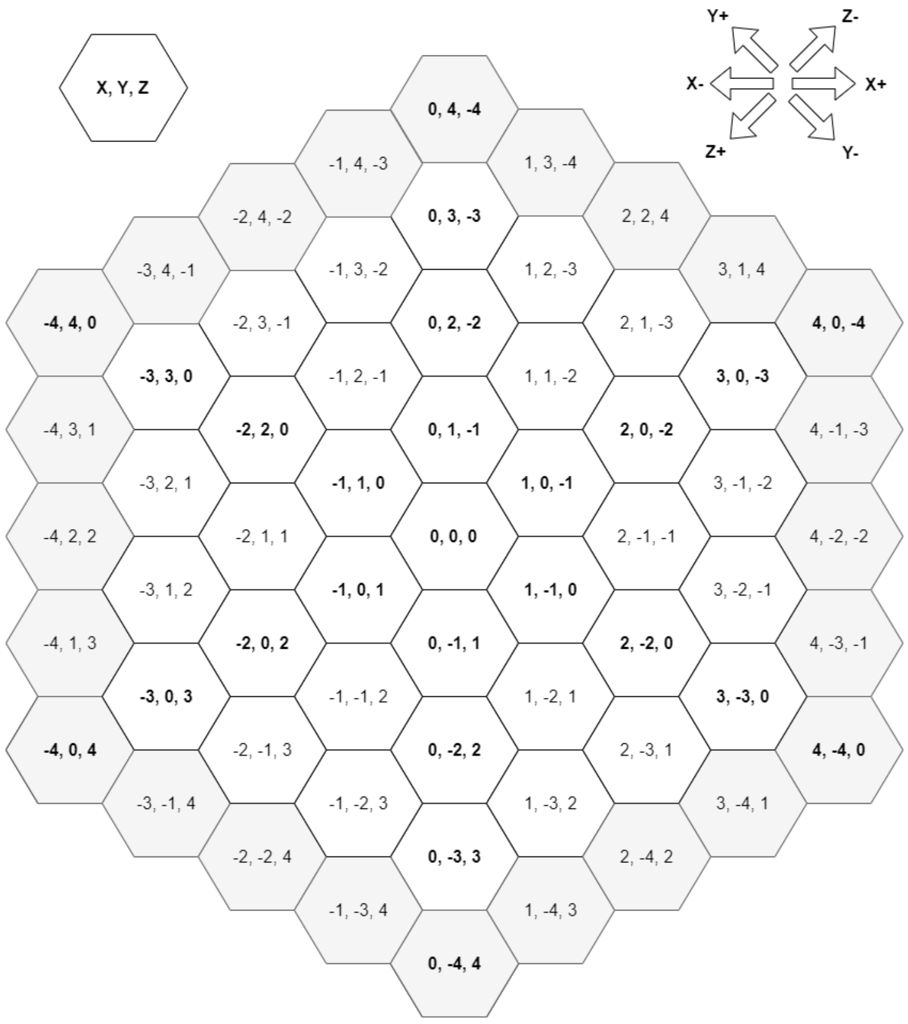

Figure 5. "Gipf" board represented as hexagonal grid with coordinates

Because one of the main parts of the game are sets of 4 pieces in a row, the code that checks the game board for them, plays a crucial part in "Gipf"[2] game development.
These coordinates help easily and quite flexibly iterate through the game board, thus making the check for 4 in a row more effective. However, another problem arises when trying to check the whole board. Because of the unique structure of the board, there are 21 line where 4 in a row can be made. To check all the lines was not hard, but not very effective way of doing the check. That is why another solution was made, by checking only those lines, where there was a change in the game state. To give you an example, in the Figure 6 there is 2 game states, left board is before the white move and the right – after.

The white player will place a piece to B1 and move it to C2 (piece movement shown with red arrows). If comparing both states, the only changes are in positions E4 and F4 (pieces circled in red). That is why board checking method thinks that 4 in a row can exist only in those lines that go through positions E4, F4 or both. This line selection lowers the amount of lines to be checked from 21 to 6 thus making the code a bit faster.

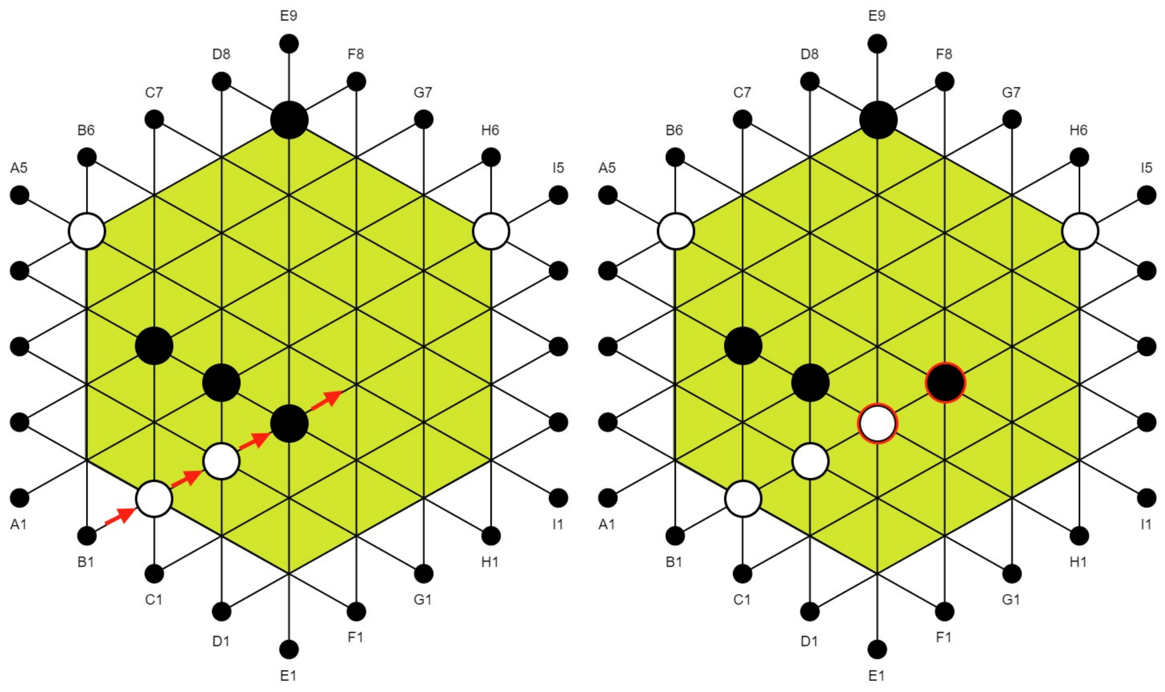

Figure 6. Game states before and after white move

2 Artificial Intelligence

2.1 Minimax

Minimax[1, 5] algorithm is a game theory algorithm that is used in game playing Artificial Intelligence (AI). This decision-making algorithm provides an optimal move for the player, assuming, both opponents are playing optimally. This algorithm is typically used in turn-based, two player, perfect information games. Perfect information games are those games where both players have the same information about the game, such as Chess[8], Tic-Tac-Toe, Go and more. In the Minimax algorithm [1, 5], one player is called the maximizer while the other player is the minimizer. If the game board has been evaluated with a score, one player tries to choose a game state with the maximum score, while the other chooses a state with the minimum score. In other words, the maximizer makes decisions to get the highest score, while the minimizer tries to get the lowest score by trying to counter opponent's moves.
The algorithm starts by generating the entire game-tree (see Figure 7), that has the current board state as it's root node and it evaluates all terminal nodes (nodes without any children), giving it a score.

At this moment (see Figure 8), the first maximizer nodes have a value of negative infinity thus starts comparing values with its children nodes, to find the highest value.

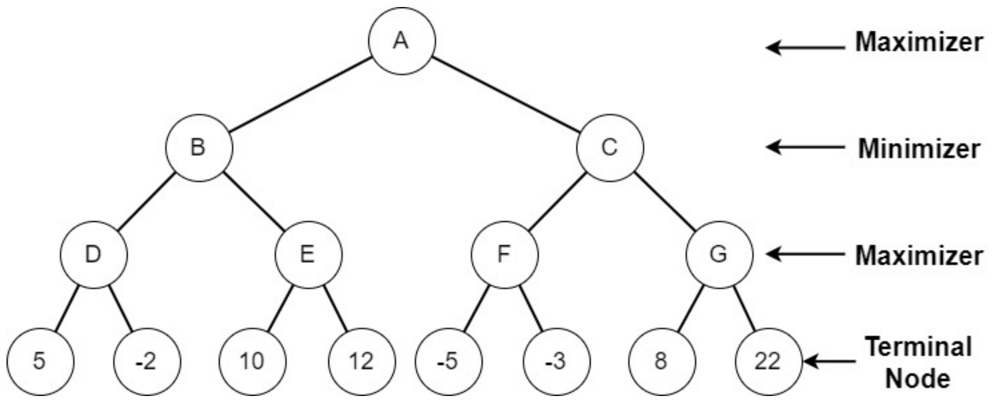

Figure 7. Minimax algorithm start

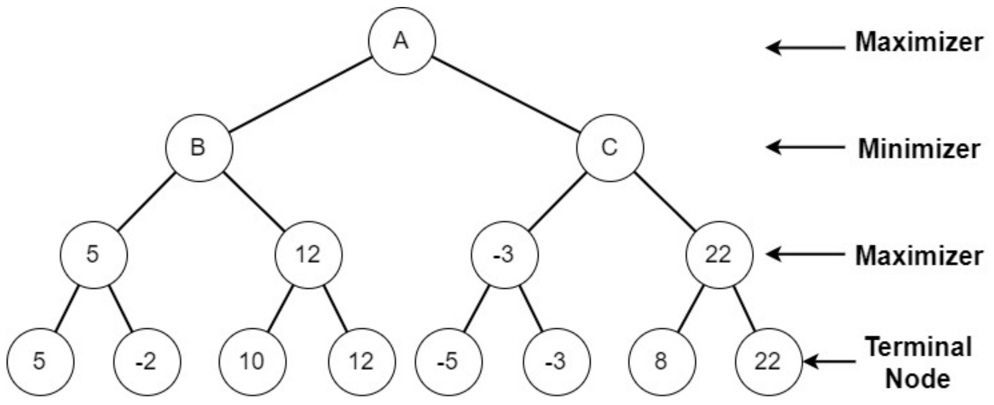

Figure 8. Minimax algorithm start

Afterwards, it is turn for the minimizer, which compares values from the maximizer nodes with positive infinity, to find the lowest value that can be seen in Figure 9. And finally, the root node, as a maximizer, gets the highest value of the previous nodes and determines that the left path is the most optimal one. Thus, the algorithm will make the move from the B node.

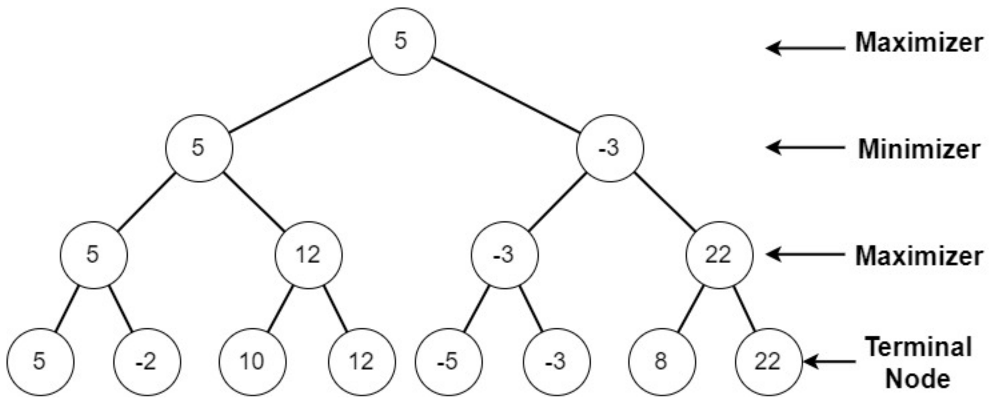

Figure 9. Minimax algorithm start

Although, Minimax algorithm[1, 5] is good at making decisions, it can be time-consuming to compute moves for games with high branching factor. For example, "Gipf" [2] board game can have up to 42 viable moves at one point in time, thus with only depth of 4 (depth is amount, that determines how many times game tree needs to branch) game tree will have over 3 000 000 nodes.

Algorithm 1. Minimax pseudocode

```
procedure MiniMax($node, depth, isMaximizingPlayer$)
    if $depth == 0$ or $nodeIsTerminalNode$ then
        return $nodeEvaluationValue$
    end if
    if $isMaximizingPlayer$ then                                    ▷ Maximizing player
        $valueMax = -\infty$
        for each $child\ of node$ do
            $valueMax$ = max($valueMax$, MiniMax($child, depth - 1, False$))
        end for
        return $valueMax$
    else                                                            ▷ Minimizing player
        $valueMin = +\infty$
        for each $child\ of node$ do
            $valueMin$ = min($valueMin$, MiniMax($child, depth - 1, True$))
        end for
        return $valueMin$
    end if
end procedure
```

The pseudocode of the Minimax algorithm[1, 5] can be seen in algorithm 1. The initial state of the algorithm would look something like this: $MiniMax(originNode, depth, True)$. In this case depth represents how many levels will the game tree have (depth value for game tree in Figure 9 is 3). $OriginNode$ and $node$ variables represent the game-state, while $isMaximizingPlayer$ is a Boolean variable that says how should the $node$ be evaluated, as maximizer or minimizer. The first if statement returns the score of the node if it is a terminal node, either by reaching specified depth or one of the players reach win condition. The code in lines from 5 to 17 tries to get the maximum value for maximizer and minimum for minimizer by recursively calling $MiniMax$ procedure. $Child$ variable represents a game-state that happened after game-state in $node$ variable.

2.2 Alpha-Beta pruning

Alpha-Beta pruning [7, 6] is optimized version of search algorithm Minimax[1, 5], that reduces the computation time by a huge factor. This allows the search to be much faster and even go into deeper levels (higher depth) in the game tree. This algorithm has two additional parameters: alpha and beta. Alpha is the best value that the maximizer currently can guarantee at that level or above and beta is the best value that the minimizer currently can guarantee at that level or above. The initial value of alpha is negative infinity and of beta is positive infinity. The Alpha-Beta algorithm works very similarly to Minimax, but the difference comes after each maximum or minimum value is found. The functionality of Alpha-Beta pruning will be explained with an example.

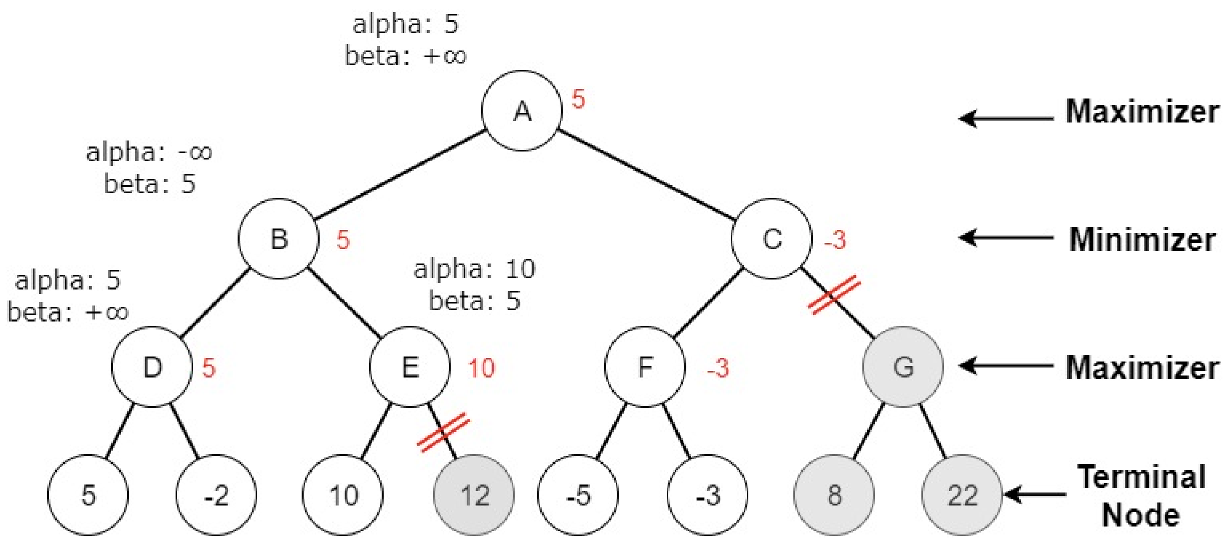

Figure 10. Alpha-Beta pruning algorithm finished calculation

Firstly, algorithm will reach node D, will evaluate left child node and will compare it with alpha value. Because alpha value is negative infinity and is smaller than 5, alpha will get value 5 and check if it needs to evaluate right node by comparing beta <= alpha (this time it is false). After evaluating and comparing right node value with the current alpha value, algorithm returns current alpha value to B node as final D nodes value. B node compares D value with beta to find lowest one (it is 5), assigns it to beta and checks beta <= alpha (gets false). At this moment, algorithm reaches E node with alpha equals negative infinity and beta = 5. After evaluating left child node, E makes alpha value equal 10 and checks beta <= alpha that return true thus sending value 10 as E nodes value. Because B node is minimizer, it returns 5 as its value. Node A goes to node C with values 5 for alpha and negative infinity for beta. Node C will get value -3 from F that is maximizer which evaluates both child nodes and returns the highest value to C. After assigning -3 to beta, C node checks beta <= alpha (-3 <= 5) and returns -3 without evaluating its right child – node G. Root node A compares both values to get the answer 5 as the best value.

For the board game, returning best score from the Alpha-Beta algorithm [7, 6] does not provide any benefit thus it was modified to be more object oriented. The pseudocode in Algorithm 2 represents the Alpha-Beta algorithm [7, 6] that was created and used for this thesis project. The initial call of the algorithm would look like this: AlphaBeta($node, 0, True, -\infty, +\infty$). The parameters of the procedure AlphaBeta:

- *isMaximizingPlayer* shows if the current node is maximizer or minimizer.
- *depth* show at what level is the current node.
- *node* parameter is an object that contains values like the colour of the piece played, board score, number of pieces in each reserve, etc (It is passed by reference).
- *alpha* and *beta* parameters, as name suggests, contain values of alpha and beta.

The first *If* statement checks if the node is not the terminal node (did not reach the depth limit and either players piece reserve is not equal to 0). The GetPlayableTurns method returns all playable turns for the game state in *node* object, that will be used while creating child nodes. Algorithm, after finding bigger value for the maximizing player, assigns the *turn*, that was made to reach child nodes state, to the *node* object if the current depth is 0 (the root node of the game tree is in depth 0). The pruning of the game tree happens when the current *beta* value is smaller or equal to current *alpha* value (can be seen in lines 16-18 and 22-24). The final *If* statement assigns score to node object if it is the terminal node.

Algorithm 2. Alpha-Beta pseudocode

```
1:  procedure AlphaBeta(ref $node, depth, isMaximizingPlayer, alpha, beta$)
2:      $valueMax = -\infty$
3:      $valueMin = +\infty$
4:      if $depth <> 4$ or NOT $nodeIsTerminalNode$ then
5:          $PlayableTurnsList$ = GetPlayableTurns($node.gameState$)
6:          for each $turn$ in $PlayableTurnsList$ do
7:              $ChildNode$ = NewChildNode($node, turn$)
8:              AlphaBeta(ref $ChildNode, depth + 1$, NOT $isMaximizingPlayer, alpha, beta$)
9:              if $isMaximizingPlayer$ then                          ▷ Maximizing player
10:                 if $ChildNode.score > valueMax$ then
11:                     $node.score = ChildNode.score$
12:                     if $depth == 0$ then
13:                         $node.madeTurn = ChildNode.madeTurn$
14:                     end if
15:                 end if
16:                 $alpha$ = max($alpha, valueMax$)
17:                 if $beta <= alpha$ then
18:                     break
19:                 end if
20:             else                                                  ▷ Minimizing player
21:                 $node.score$ = min($ChildNode.score, valueMin$)
22:                 $beta$ = min($beta, valueMin$)
23:                 if $beta <= alpha$ then
24:                     break
25:                 end if
26:             end if
27:         end for
28:     end if
29:     if $isTerminalNode$ then
30:         $node.score =$ EvaluateBoard( $node$)
31:     end if
32: end procedure
```

2.3 Board evaluation

For the search algorithm, whether it will be Minimax[1, 5] or Alpha-Beta algorithm [7, 6], to give very good results, the board has to be evaluated accordingly. The score of the terminal nodes is calculated by a specific formula, that determines what kind of strategy the algorithm should follow. The evaluation formula used for the "Gipf"game:

$$S = (k_1 \times sg \times WR) + (k_2 \times (-sg) \times BR) + (k_3 \times sg \times WB) + (k_4 \times (-sg) \times BB) + \\ + (k_5 \times sg \times WW) + (k_7 \times (-sg) \times WC) + (k_8 \times sg \times BC) \quad (2.1)$$

The members of the equation:

- S - the score of the board.
- k_n - multiplier that determines the importance of each part of the formula.
- sg - determines for which player score is calculated for. If algorithm plays as white pieces the value of sg is 1 and for black pieces sg = -1).
- WR/BR - the amount of pieces left in the reserve. WR - white reserve, BR - black reserve.
- WB/BB - the amount of pieces on the board, WB for white and BB for black.
- WW - determines the victory of one player. If white wins the value is $+\infty$ and $-\infty$ if black wins.
- WC/BC - the amount of pieces that were captured (removed from the game). WC = white, BC = black.

The evaluation formula, at first, was created without the count of the captured pieces. However, it was not performing with consistency, thus the missing part was added in.

3 Solution variations and testing

3.1 Comparison of Minimax[1, 5] and Alpha-Beta algorithms in the solution

The goal of this comparison was to compare the impact of usage of two different algorithms in the solution.
First solution was created using Minimax algorithm[1, 5].
Second solution was created using Alpha-Beta algorithm.
For the comparison of the solutions the following measures per depth were measured:

- Time for calculating the next turn.
- Quantity of nodes.

The testing hardware specification:

- CPU - Intel® Core™ i5-6600K quad core processor with 3.5GHz frequency.
- GPU - Geforce GTX 1060 3GB.
- RAM - 40 GB.

The depth is one of the main factors for Alpha-Beta and Minimax algorithms [1, 5] that determine the computation time of the game tree for each turn.
Table 1 shows what are the differences of those algorithms regarding time (in seconds) it took to compute a turn and how many nodes were evaluated.

Depth	**2**	**3**	**4**	**5**	**6**
Time$_{MiniMax}$	0.15	5.6	225.6	-	-
Time$_{\alpha\,-\,\beta}$	0.02	0.16	0.5	6	20.5
Time Alpha-Beta vs Minimax, %	13.33	2.86	0.22	-	-
Nodes$_{MiniMax}$	1806	75606	3142374	-	-
Nodes$_{\alpha\,-\,\beta}$	125	1930	5457	81020	228615
Nodes Alpha-Beta vs Minimax, %	6.92	2.55	0.174	-	-

Table 1. Time and node amount dependency to the depth of the algorithms

The results shows:

- In all cases Alpha-Beta algorithm is superior to Minimax algorithm.
- The higher depth is being used in the solution, the more effective Alpha-Beta algorithm is.
- In the case of depth 2, for calculating the next turn the Alpha-Beta algorithm took 13.33% of Minimax algorithm time and 6.92% of Minimax quantity of nodes.
- In the case of depth 3, for calculating the next turn the Alpha-Beta algorithm took 2.86% of Minimax algorithm time and 2.55% of Minimax quantity of nodes.
- In the case of depth 4, for calculating the next turn the Alpha-Beta algorithm took 0.22% of Minimax algorithm time and 0.174% of Minimax quantity of nodes.
- In the case of depth 5 and 6, the data of solution with Minimax algorithm was not recorded because of the lack of hardware capacity.

3.2 AI vs Randomize

The first testing faze for Alpha-Beta pruning algorithm was to defeat an algorithm which does random moves without looking at the game state. To achieve the results, the randomize algorithm was created. It was designed around the Unity engines method that gave a random number from a specified range. However, a number was not enough to make a move. Thus a list, that contain all possible moves, was created to be used in conjunction with the build it random method. Firstly, from all possible moves, only the playable ones were selected and put to the new list. Afterwards, a random index was generated by providing a range from 0 till the length of the new list to the Unity engines method. Thus, by using that index to receive a move from the playable move list, randomize algorithm was able to play a the game. The testing itself was very straight forward. The Alpha-Beta algorithm played 10 games against randomize algorithm by playing as white pieces and 10 games as black pieces. All games were won by AI.

3.3 AI vs AI

The goal of this testing was to check how Alpha-Beta algorithm based AIs are playing against each other. The difference between players is the type of strategy they are using.
The first AI is modified to focus more on capturing the opposing player pieces. This is achieved by increasing the value of the multiplier that is evaluating the amount of captured pieces in the formula.
The second AI strategy is to treat all factors (amount of pieces on the board, amount of pieces in the reserve and amount of captured pieces of the opponent) of evaluation formula equally. That is why all of the multipliers are set to the value of 1.
There were 20 games played. The first AI player won 17 games (85% win rate), second AI player won 3 games (15% win rate). The assumption can be made that the first strategy is more superior then the second one.

3.4 Testing on quizzes

The goal of this testing was to check how the AI is solving different quiz cases.
Quiz cases were taken from the game creators web page [2].
All quizzes have the answers, so the test results are known.

3.4.1 Quiz 1

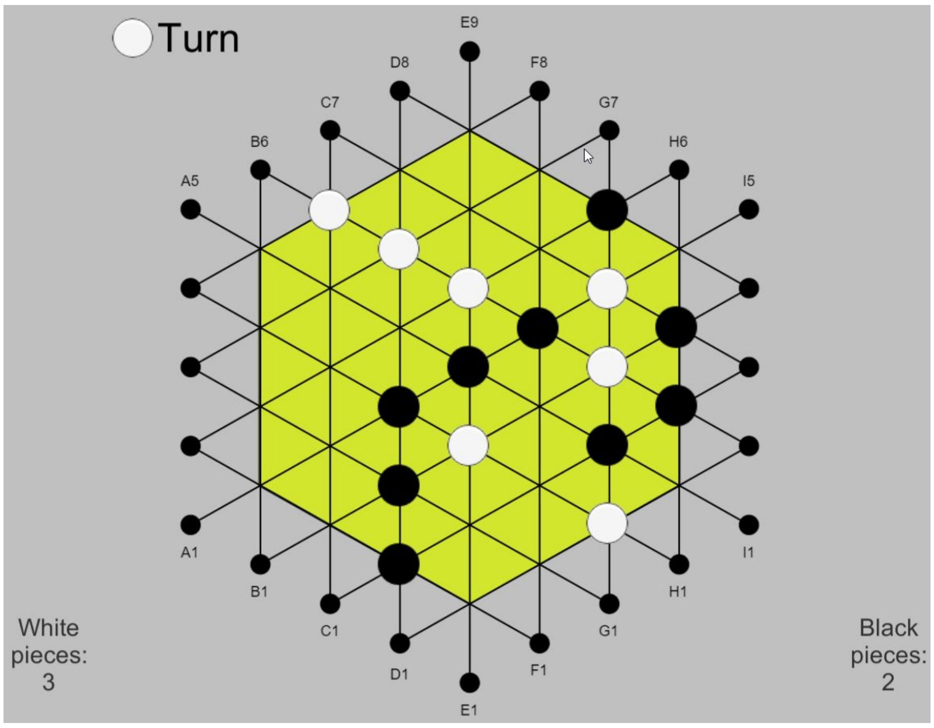

Figure 11. Quiz board

The quiz situation is shown in the Figure 11.
The challenge of the quiz:

- White are winning with 3 pieces in the reserve.
- Black have only 2 pieces in the reserve.
- White have to win the game in 6 turns.

The strategy of the AI was to focus on capturing of opponent pieces (the value of the multiplier that is evaluating the amount of captured pieces in the formula was increased) as mentioned in section 3.3.
The test result was positive - AI has solved the quiz after the 6 turns.

3.4.2 Quiz 2

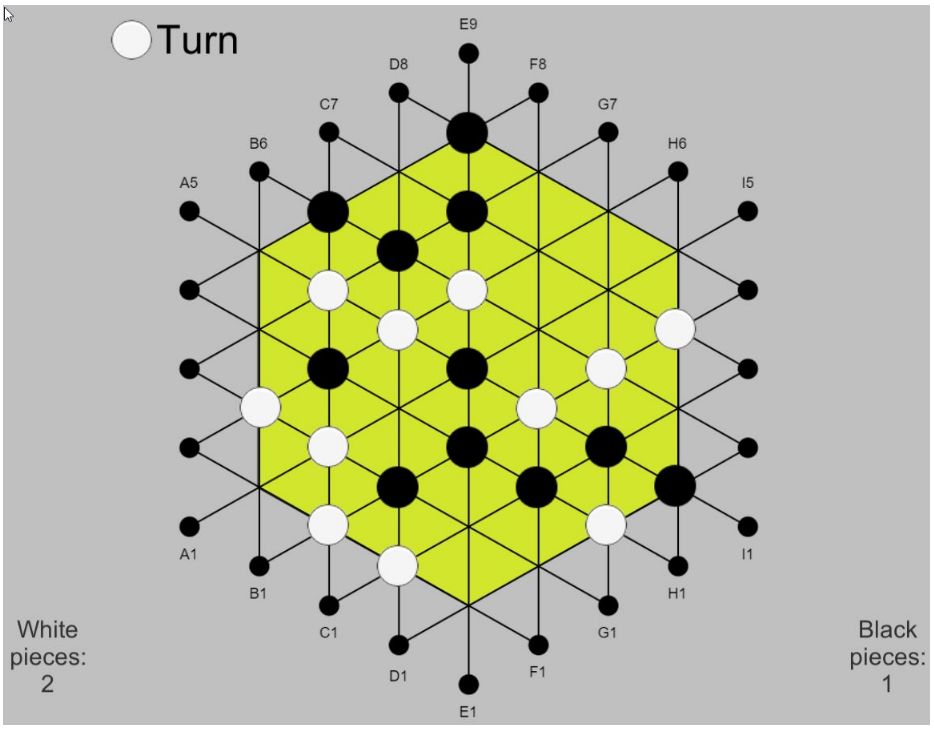

Figure 12. Quiz board

The quiz situation is shown in the Figure 12.
The challenge of the quiz:

- White are winning with 2 pieces in the reserve.
- Black have only 1 piece in the reserve.
- White have to win the game in 2 turns.

The strategy of the AI was to focus on capturing of opponent pieces (the value of the multiplier that is evaluating the amount of captured pieces in the formula was increased) as mentioned in section 3.3.
The test result was positive - AI has solved the quiz after the 2 turns.

3.4.3 Quiz 3

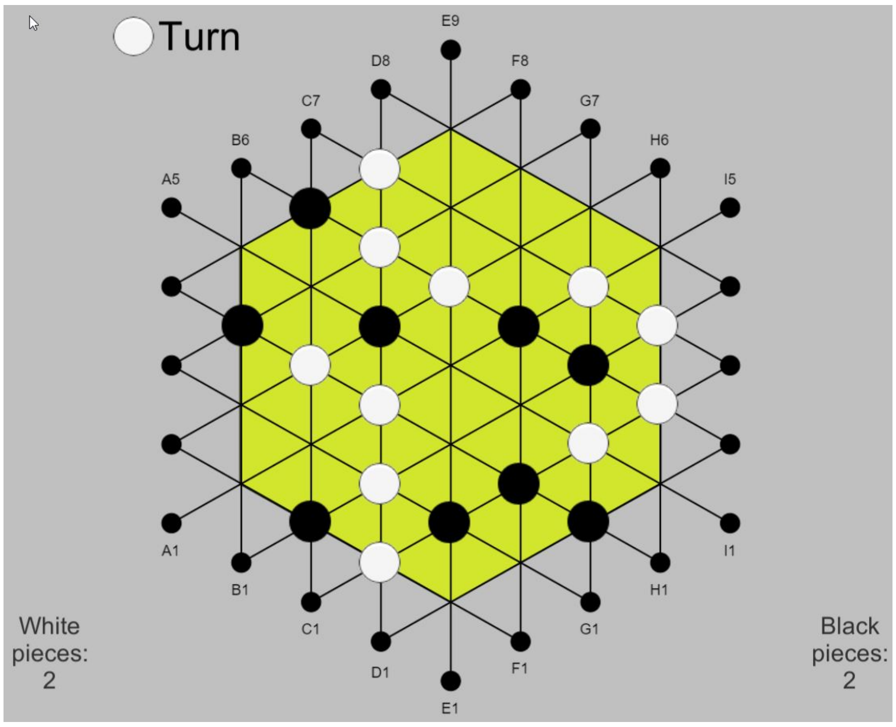

Figure 13. Quiz board

The quiz situation is shown in the Figure 13.
The challenge of the quiz:

- White are winning with 2 pieces in the reserve.
- Black have only 2 pieces in the reserve.
- White have to win the game in 2 turns.

The strategy of the AI was to focus on capturing of opponent pieces (the value of the multiplier that is evaluating the amount of captured pieces in the formula was increased) as mentioned in section 3.3.
The test result was positive - AI has solved the quiz after the 2 turns.

Conclusions and Recommendations

In this thesis there the "Gipf" board game was created using Artificial Intelligence Alpha-Beta pruning algorithm. The board game is an abstract type, thus do not have any random or luck based events and relies on player strategies. For comparison reasons two solutions were created and results compared using two algorithms - Alpha-Beta pruning and Minimax. The results show that Alpha-Beta pruning algorithm is more effective compared with Minimax algorithm.
Created solution was tested against other AI and randomized bot. Different strategies can be implemented by modifying the weight of the factors (amount of pieces on the board, amount of pieces in the reserve and amount of captured pieces of the opponent) in the evaluation function.
One of the AI was modified for strategy to focus more on capturing the opposing player pieces. The second AI strategy treated all factors of evaluation formula equally. The first strategy was more successful.
Possible improvements:

- Optimization of games workflow.
- Improvements to data model.
- Improvements to evaluation of the board.

References

[1] Artificial intelligence: Mini-max algorithm - javatpoint.

[2] Gipf is a game.

[3] Unity.

[4] Hexagonal grids, Mar 2013.

[5] Minimax algorithm in game theory: Set 1 (introduction), May 2019.

[6] Akshay L Aradhya. Minimax algorithm in game theory: Set 4 (alpha-beta pruning), Dec 2019.

[7] Rashmi Jain. Minimax algorithm with alpha-beta pruning, May 2019.

[8] Werda Putra and Lukman Heryawan. Applying alpha-beta algorithm in a chess engine. *Jurnal Teknosains*, 6:37, 08 2017.

[9] Kieran Sockalingam. Alpha beta pruning and its place in computer game play. 10 2015.