PRACA DYPLOMOWA INŻYNIERSKA

An analysis of min-max algorithm implementations in logic games

Wydział Fizyki Technicznej, Informatyki i Matematyki Stosowanej
Promotor: dr hab. Inż. Włodzimierz Mosorow
Dyplomant: Paweł Ciążyński
Nr albumu: 171120
Kierunek: Informatyka
Specjalność: Information Technology

Łódź 13.09.2017

Lodz University of Technology

Faculty of Technical Physics, Information Technology and Applied Mathematics,

Information Technology

Bachelor of Engineering Thesis

An analysis of min-max algorithm implementations in logic games

Paweł Ciążyński

Student's number: 171120

Supervisor:

PhD Włodzimierz Mosorow

Łódź, 2017

POLITECHNIKA ŁÓDZKA
Wydział Fizyki Technicznej, Informatyki i Matematyki Stosowanej

Paweł Ciążyński

PRACA DYPLOMOWA INŻYNIERSKA

Analiza implementacji algorytmu min-max
w grach logicznych

Łódź, 2017 r.
Opiekun: Dr hab. Inż. Włodzimierz Mosorow

STRESZCZENIE

Człowiek ma styczność z grami cały czas. Niekiedy nie zdaje sobie nawet z tego sprawy. Czy można nazwać grą moment, w którym człowiek śpieszy się do pracy i próbuje dojechać tam w krótszym czasie niż jego współpracownicy? To także może być grą, a jak każdą grę – można ją rozwiązać. W swojej pracy pragnę przedstawić czytelnikowi w jaki sposób można tego dokonać i co dokładnie znaczy „rozwiązać grę". Praca jest podzielona na cztery części. W pierwszej części przybliżam czytelnikowi teoretyczne aspekty sztucznej inteligencji w grach logicznych. Wprowadzam różne sposoby rozwiązywania gier. W drugiej części przedstawiam na czym polega algorytm min-max, jego złożoność oraz różne odmiany. Trzecia część jest częścią praktyczną, gdzie przedstawiam implementację oraz działanie algorytmów dla dwóch gier logicznych. Ostatnia, czwarta część podsumowuje i analizuje wyniki działania różnych odmian algorytmu dla poszczególnych gier oraz prezentuje, która odmiana algorytmu min-max działa najlepiej.

Słowa kluczowe: min-max, gry logiczne, kółko i krzyżyk, czwórki, połącz cztery, drzewo gry

Lodz University of Technology
FACULTY OF *Technical Physics, Information Technology and Applied Mathematics*
Information Technology

Paweł Ciążyński

BSc THESIS

An analysis and comparison of different min-max algorithm implementations in logic games

Lodz, 2017

Supervisor: PhD Włodzimierz Mosorow

ABSTRACT

People have contact with games on daily basis. Sometimes they even are not aware of it. Can we call it a game, when someone is in hurry to get to work and tries to reach his destination in as little time as possible, ideally in shorter time than his colleagues? It can also be called a game and as all other games – it can be solved. In my diploma thesis, I want to present to the reader how can it be achieved and also what does it mean to "solve a game". This thesis is divided into four chapters. In the first chapter, I present theoretical aspects of the Artificial Intelligence (AI) in logic games. I introduce different approaches of solving games. In the second chapter I describe how a Min-Max algorithm works together with its different variant. The third chapter is a practical part, where I show an implementation and results of a Min-Max algorithms in two logic games. The last part of this diploma thesis summarizes and analyzes results of different variants of a Min-Max algorithm for given games.

Keywords: min-max, logic games, tic-tac-toe, connect-four, game tree

Contents

1. Introduction

In these days, computers are very fast machines and can do tons of calculations in a split of second. However, sometimes these computers are not fast enough. The speed is sufficient for solving many small logic games like tic-tac-toe, but people also would like to solve very complicated games like, for example chess.

In my engineering thesis, I want to present different approaches on solving logic games. Later on, I will concentrate mostly on Min-max algorithm and its various implementations. I will write two games in java programming language (Connect Four and Tic-tac-toe). Then the goal of my thesis is to write and present how to solve the game and solve it using different variations of my Min-max algorithm. I do not aim to write perfect artificial intelligence program for these two games, but to present how these algorithms work in theory and in practice.

2. Definitions

To properly understand what is a Min-max algorithm and how it works, at first one must know some definitions. At the beginning a definition of the game will be presented together with some examples of games. The next term that needs to be understood is a game tree. At the end of this chapter, I write about the definition of artificial intelligence and also what does it mean to solve a game.

2.1. Definition of a game

A game is a physical or mental activity with set rules, in which participates one or more players. Games are undertaken for enjoyment and sometimes even used as educational tool.

There are different types of games. We can divide games into few categories while taking into account different aspects of them:

- Number of players:
  - No players (e.g. game of life)
  - One player (e.g. solitaire)
  - Two players (e.g. tennis, tic-tac-toe, chess)
  - More players (e.g. soccer)
- Amount of sum:
  - Zero-sum game (e.g. tic-tac-toe, chess) – it's a game in which the gain (loss) of one player is a loss (gain) for another player.
  - Non-zero-sum game – a game in which the gain for one player does not necessary means loss for the opponent. Example of this game can be Prisoner's dilemma or Battle of the sexes.
- Information to which player has access:
  - Complete information – situation in which the player has full knowledge about the board situation and strategies available to the other player.
  - Non-perfect information – player knows only some part of the board situation or moves available to his opponent.
- Determinism:
  - Deterministic (e.g. connect-four) – a game in which there is no random factor.

– Non-deterministic (e.g. poker, dices) – random factor is a common thing in this game.

	Number of players	**Amount of sum**	**Information**	**Determinism**
Chess	2	Zero-sum	Complete	Deterministic
Solitaire	1	Zero-sum	Non-perfect	Non-deterministic
Game of life	0	Non-zero-sum	Complete	Deterministic
Soccer	22	Zero-sum	Non-perfect	Non-deterministic
Poker	2 – 8	Non-zero-sum	Non-perfect	Non-deterministic
Connect Four	2	Zero-sum	Complete	Deterministic
Prisoners' dilemma	2	Non-zero-sum	Complete	Deterministic

Table 1. Comparison of different games and their characteristics.

Logic game is a type of game in which only matters intelligence of a player, ability to find best solution to the given problem. Unlike in physical games, logic games don't need a player to be played (e.g. game of life).

2.2. Game tree

A game tree is a graph which nodes represent states in the game and connections between them are different possible moves. These graphs are used to present and check all possible moves in subsequent turns.

Different games have got different graph complexity. It means that number of moves players can perform in each turn is different for each game. For example game chess has got a lot more complex graph than the game Connect-four. Graph complexity also depends on the round of the game. For example, in game tic-tac-toe, player who moves first can place his pawn on one out of nine fields, but next player can place his pawn on only one out of eight fields, etc.

Easy and simple way to estimate the games upper graph complexity is to count all possible states on the board. For example, in the game tic-tac-toe, all possible states are all variations with repetitions with set of 3 elements placed on 9 fields. It gives us $V=n^k=3^9=19\ 183$ different states.

However, this is only the upper estimation of graph complexity. Many games have got symmetric boards. It means that for each axis of symmetry we can divide the amount of states by two, because there exists exactly the same game state, but only rotated or reflected. For example, board of game tic-tac-toe has got four axes of symmetry (two diagonal, vertical and horizontal) and one axis of rotation.

2.3. Definition of Artificial Intelligence

Artificial Intelligence (AI) is the ability of a computer program to perform humanlike actions and decisions. Artificial Intelligence can also act as an expert system, or a program for the perception and recognition of shapes.

In 1950 Alan Mathison Turing proposed that ability of a computer system to act as a human being in a conversation with other people can be a test for system's intelligence. (Turing test).

John McCarthy for the first time introduced a term 'Artificial Intelligence' in 1955. He defined it as 'the science and engineering of making intelligent machines'. In modern world AI is used to solve many different problems, such as acting as an expert system, data mining, logistics, voice recognition and many others.

2.4. Solving games

In 1913 Ernst Zermelo published an article in which he analyzed the game chess if for each position on the board there is a mathematical way to determine result of the game (win or loss) and to determine next best move for a player. However, Zermelo didn't answered whether

the starting position provides a win for any player, he also stated that if it was a true statement, playing chess would be pointless, because each time we would know the winner even before the start of the game.

What does it mean to solve a game? Solving a game means that we found set of moves that will lead the player to the end of the game where given player wins, despite of his opponents' moves.

3. Game solving Methods

This chapter focuses on presenting how solving a game works and what approaches are used to solve a game. The first part of this chapter closes how in general solving a game is performed. Then, the thesis compares two graph searching methods – DFS and BFS. At the end, it presents how a Min-max algorithm and its variation called Alpha-Beta pruning work.

3.1. How to solve the game?

There are many ways to solve the game. In my thesis, I will only concentrate on searching graphs of games. Computer algorithm can create all possible moves, that players can take and then search through the graph. There are also many ways to find the solution of the graph. We can divide them into two main groups of algorithms: Depth-first search (DFS) and Breadth-first search (BFS).

3.1.1. Depth-first search (DFS)

This algorithm moves from the top of the graph to the bottom in vertical direction. Algorithm begins from root node (current state of the game) and checks its first child (first possible move from this state of the game). Then it checks the first child of the next node, and so on. When algorithm reaches the state in which there are no more children of currently visiting node, it goes back to the point where it could choose different path. This algorithm usually finds the solution pretty quickly, however it most likely will not be the most optimal solution.

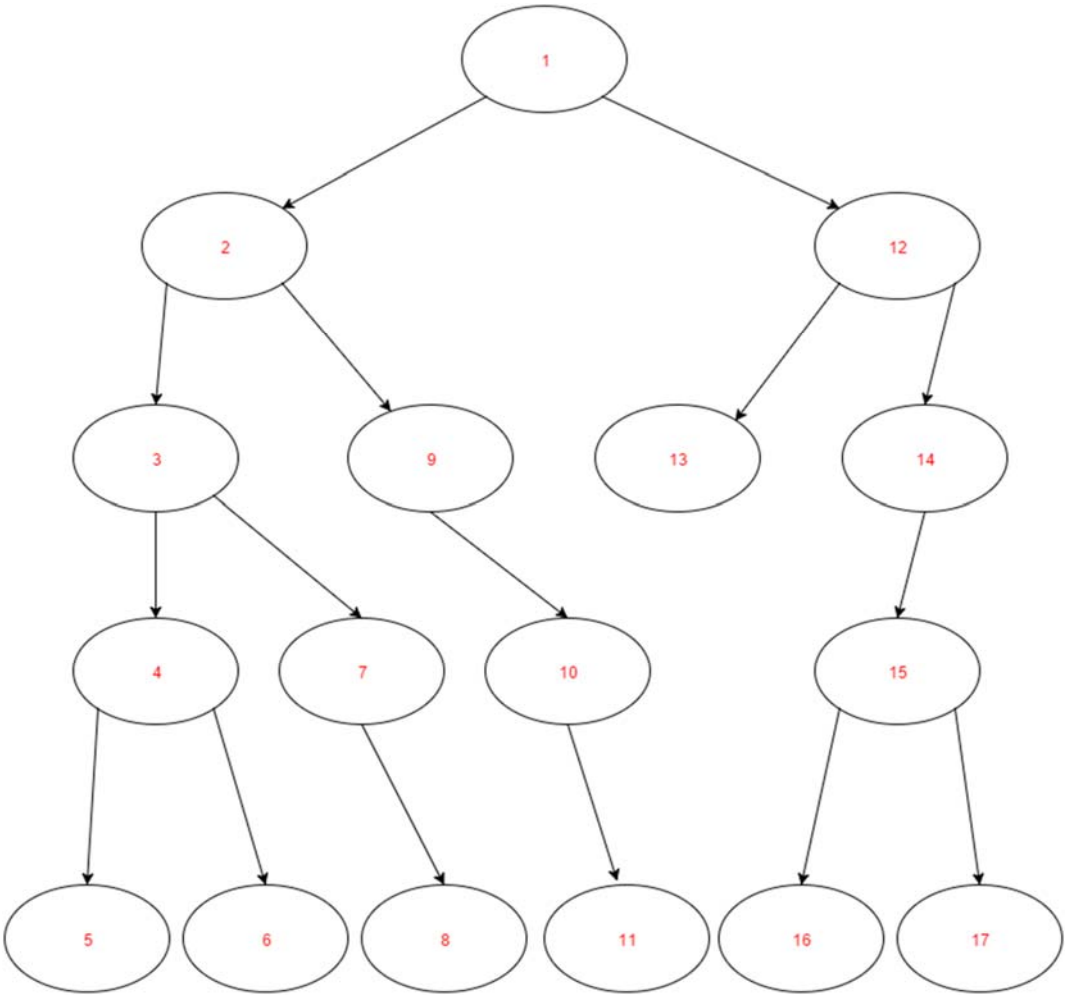

Figure 1. Graph representing the sequence of search using DFS. (source: self-prepared)

3.1.2. Breadth-first search (BFS)

In this method computer moves from the top of the graph (current state of the game) to the bottom in horizontal direction. It means that algorithm visits neighbor nodes before it progress to visit child nodes in the graph. Using this algorithm, we are certain to find the shortest possible solution.

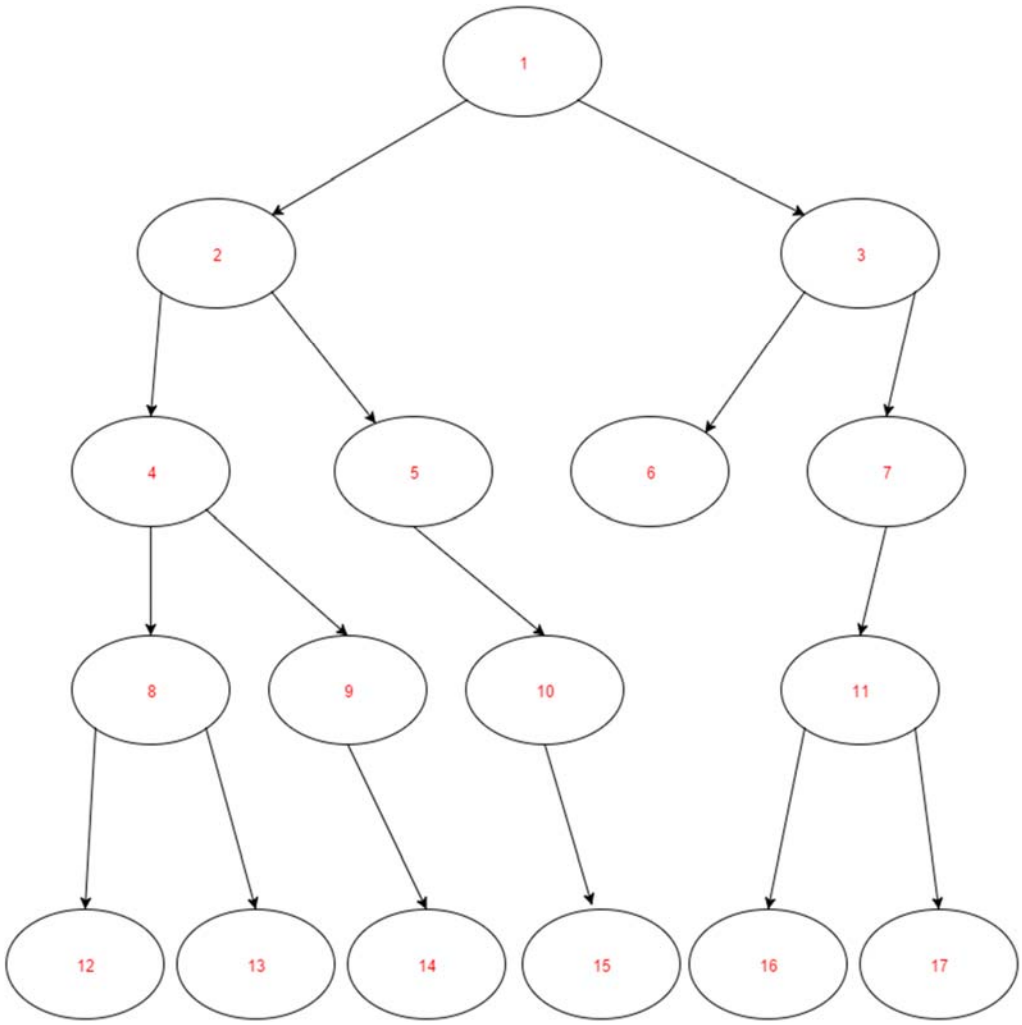

Figure 2. Graph representing the sequence of search using BFS. (source: self-prepared)

3.2. Comparison of DFS and BFS

Depth-first search and Breadth-first search are two different approaches for graph searching. Time of finding expected node in the tree depends on the given graph. Let us consider the graph below. This is purely random generated graph in which we want to find the node with value greater than 100.

At first let us analyze with DFS algorithm. As we can see, it finds the node with value greater than 100 after checking 5 nodes.

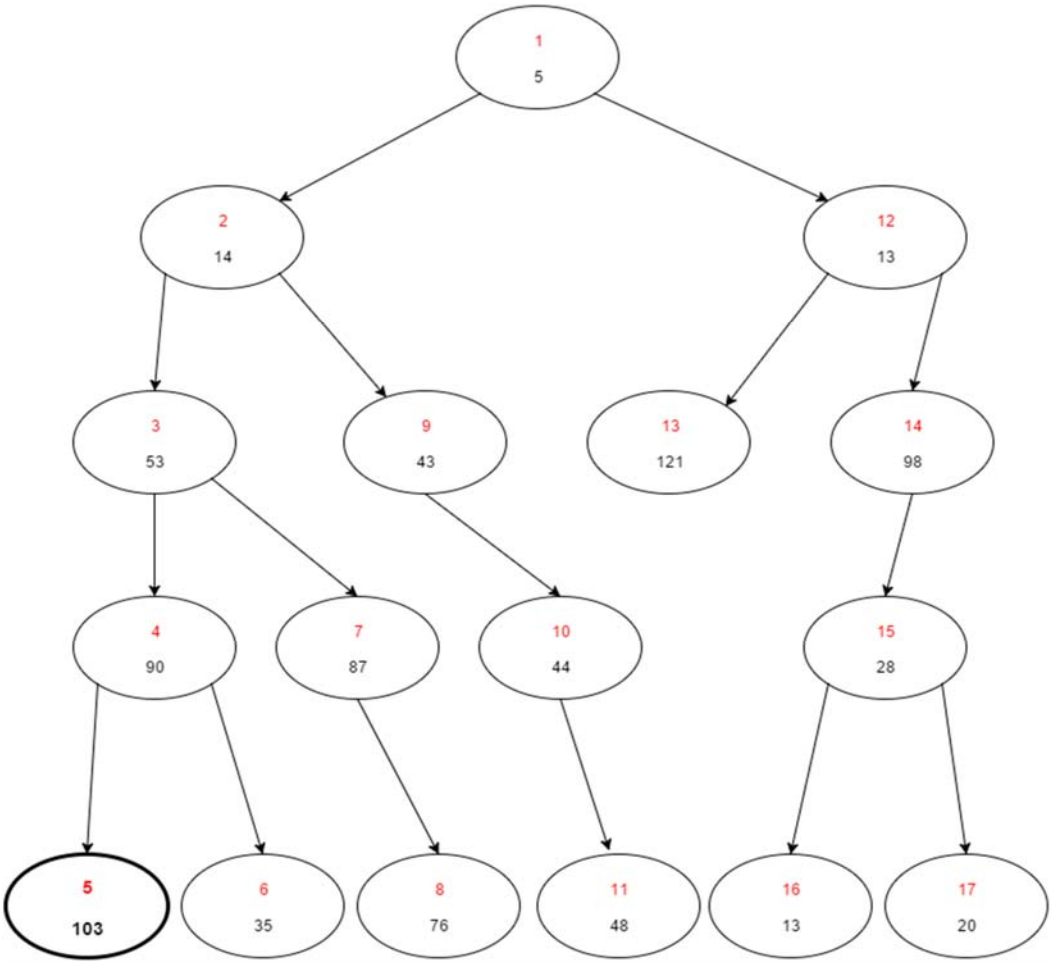

Figure 3. Graph showing how DFS finds node with value greater than 100. (source: self-prepared)

Now let us consider solving the same problem with BFS algorithm. Using BFS approach we found desired node in after checking 6 nodes. Also, this time we found completely different node with value greater than 100. It is due to the fact, that we first check all the nodes which are children of given node.

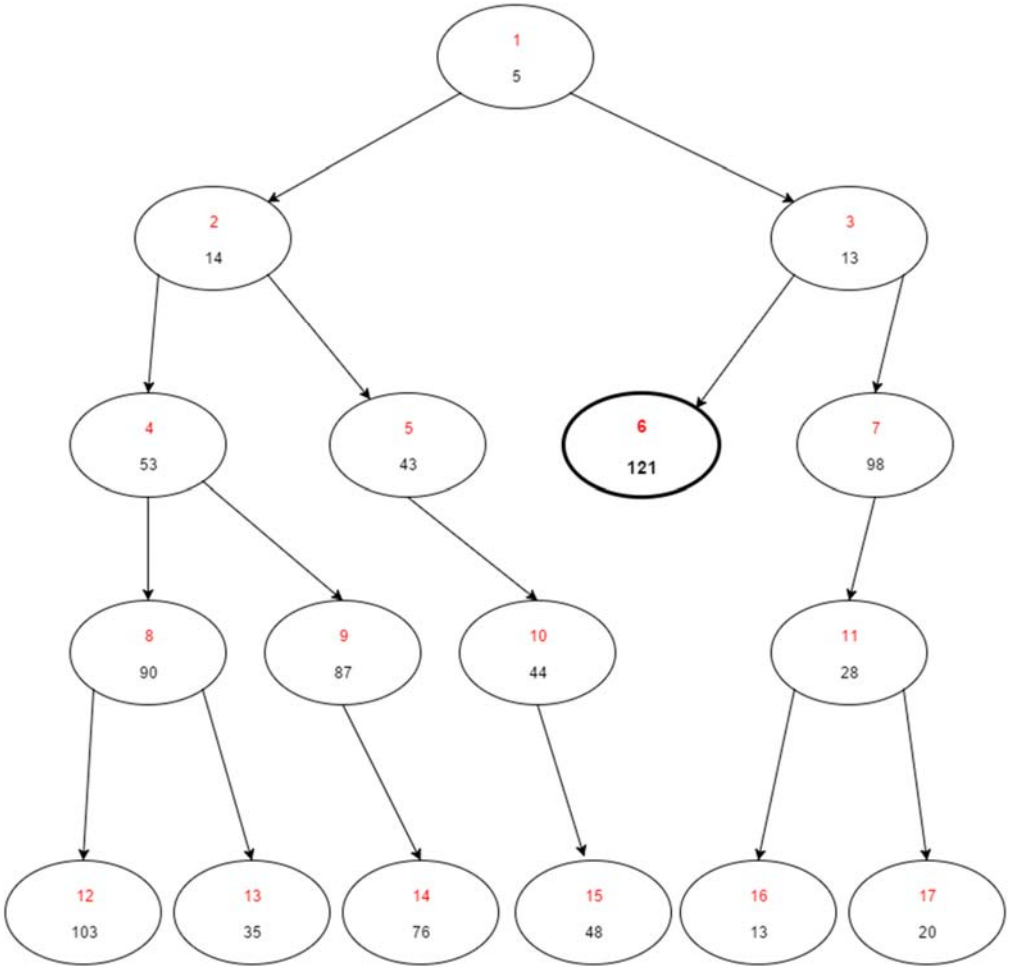

Figure 4. Graph showing how BFS finds node with value greater than 100. (source: self-prepared)

3.3. Min-max

Up to this point we discussed solving graphs without the division for two players, we only wanted to find one node with given properties. This can be useful in many situations, but let us consider a game for two players which compete with each other.

This is where we will use a min-max algorithm. This algorithms excels in solving games where two players face each other. Min-max algorithm can find the greatest value that given player can be sure to obtain without the knowledge of what actions his opponent will perform. In other words it minimizes the maximum risk for a player.

For the sake of example let the rules of this game be as following: players move in turns, each player want to get biggest positive score, each player can make one of two moves, the game ends when any player gets $+\infty$ points.

3.3.1. Basic min-max

In the basic version of min-max algorithm, for each possible move a player can take, we check all possible moves the other player can perform and then determine what is the worst possible combination of moves. Then the player should choose a move that gives him the largest possible value of the lowest possible values. Min-max algorithm uses DFS (depth first search) strategy.

Now let us see what would the graph of this game look like.

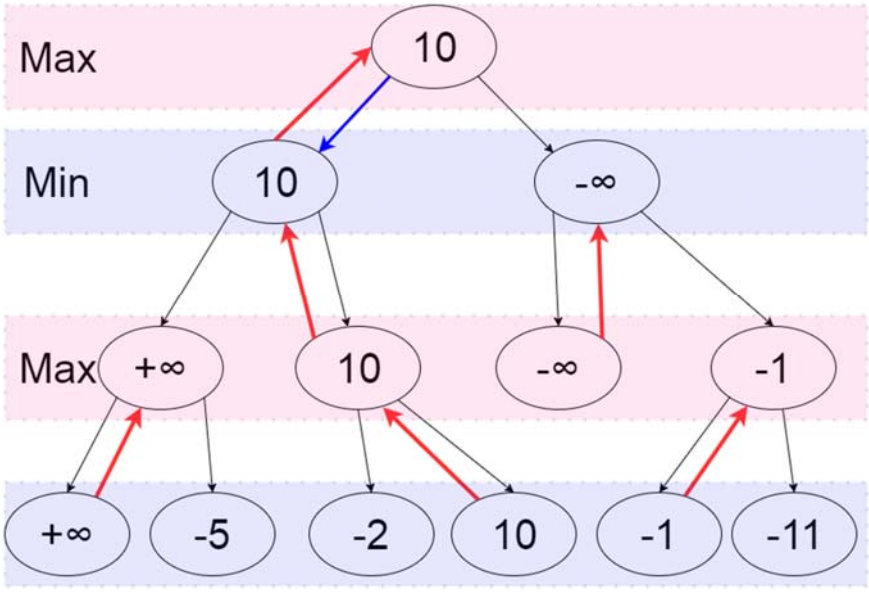

Figure 5. Graph showing how Min-Max algorithm works. (source: self-prepared)

The +∞ means that a player won the game, while -∞ means that he lost a game. In the 'max' nodes we calculate the best move for us to take and in the 'min' nodes we calculate the worst move for us that the other player can take.

From this graph, we can deduce that in 3 moves the player will be more likely to win than to lose, because he can obtain the position where he is 10 points ahead of the other player. Despite the fact that the player can win the game in 3 moves (see the node with value of +∞), it is unlikely for the player to win, because his opponent will not choose this path.

3.3.2. Alpha-Beta cuts

There also exists improved version of min-max algorithm which is called a min-max with alpha-beta cuts or a min-max with alpha-beta pruning. It is a combination of two different variants of min-max algorithm. One of them is called alpha cuts and the other is beta cuts. Alpha cut version cuts branches where maximizer can't find better option than in any already explored nodes along path to the root, while beta cut version cuts branches where minimizer can't find better option. This version of the algorithm improves its performance dramatically, without affecting the result. This algorithm tries to minimalize the number of nodes to by evaluated by min-max algorithm by 'cutting off' nodes that are for sure worse than other previously evaluated ones. It decreases the number of decision the algorithm would take and thus increases its performance.

The following graph shows where the cuts would be performed if the graph is the same as the one previously examined.

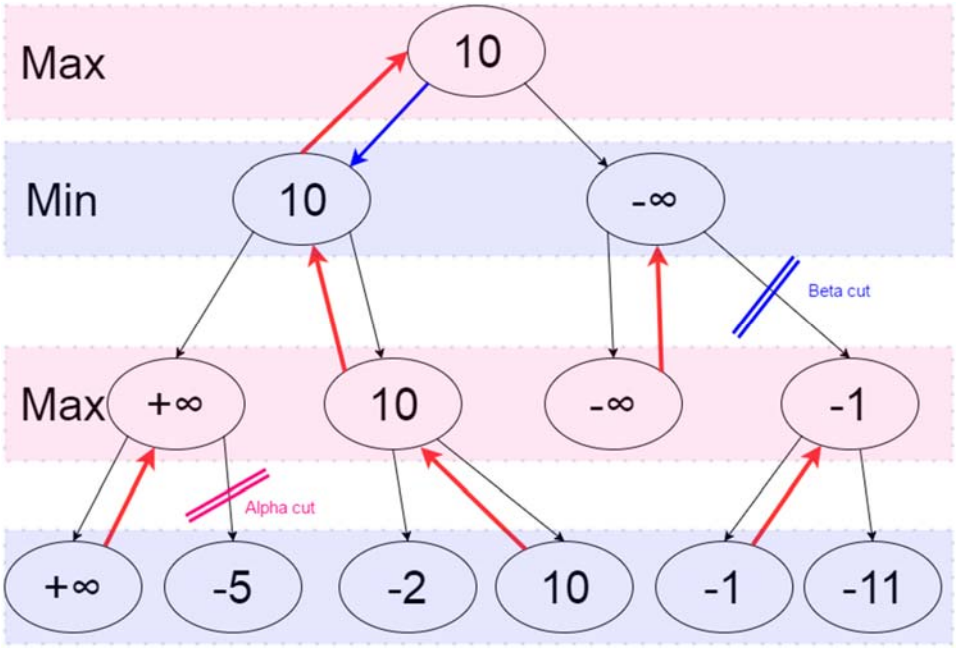

Figure 6. Graph showing how Min-Max variation called Alpha-Beta pruning works. (source: self-prepared)

4. Project Implementation

As a practical part of my bachelor thesis I implemented min-max algorithm in two games: Connect-four and Tic-tac-toe. In this chapter, firstly I will discuss the rules of these two games, then I will present the common min-max implementation for these two games. Next I will consider each game individually and present its heuristics, specific implementation and the results for different min-max algorithms executions.

4.1. Rules of games

It is impossible to design a program without knowing the specification. To properly implement a program, firstly one must know what given program should do. The same goes for implementing a logic game. In this case, the rules of both games should be clear.

4.1.1. Connect-four

Rules of Connect-four game are very simple. This is a zero-sum, two-player game in which players make their moves in turns. Players are dropping ‘X’ and ‘O’ tokens from the top of the board with grid diameter of seven columns and six rows. The pieces fall straight down and stops on the first unoccupied cell of the grid. The goal of the game is to place four discs to form a line in a horizontal, vertical or diagonal direction.

Below is presented the example state of board where player who places ‘X’ tokens won the game. As we can see, four ‘X’ tokens are placed in the row.

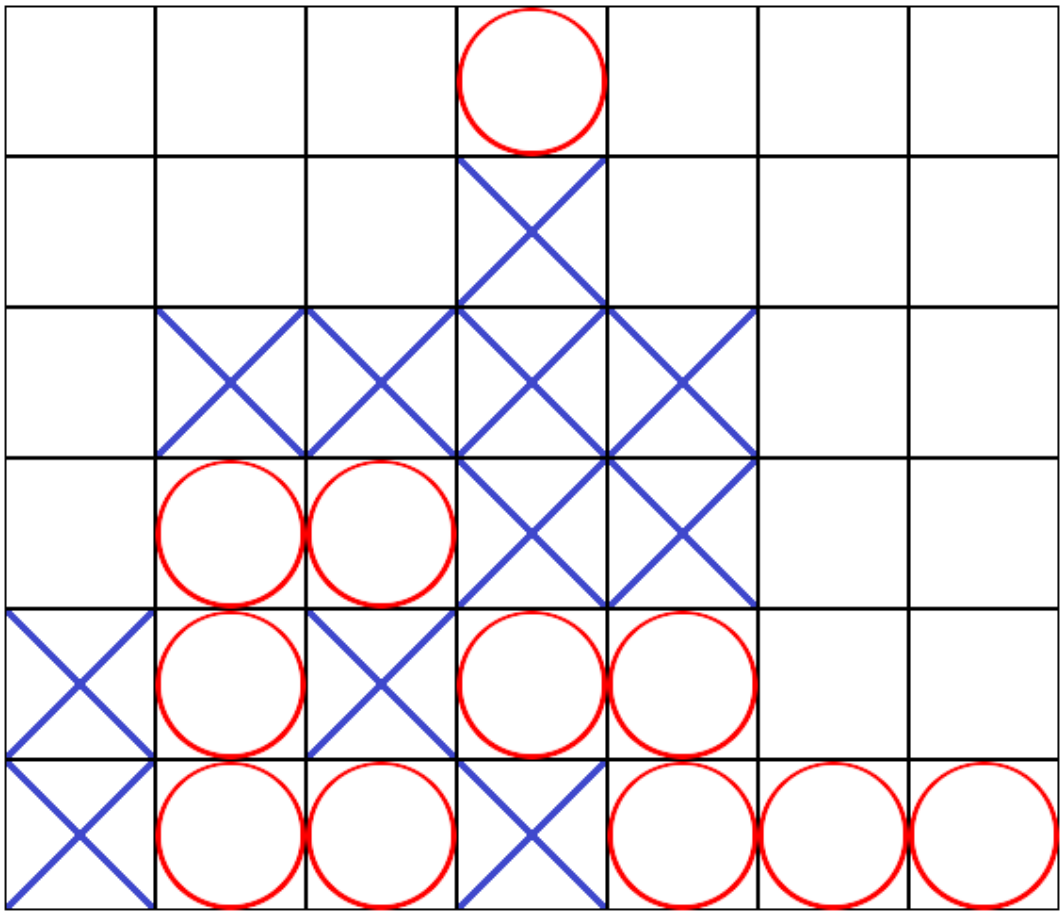

Figure 7. Example state of the board for Connect-Four game where player owning crosses wins the game. (source: self-prepared)

4.1.2. Tic-tac-toe

Tic-tac-toe is a zero-sum, two-player game, where players make their moves in turns. One player places ‘X’ tokens and the other places ‘O’ tokens on the three by three grid board. The first players to place three tokens in the same column, row or in diagonal wins.

4.2. Common code

During the implementation of both games, I’ve tried to design it in a way, that will be easy to add more games into the program. It means that the main logic for finding the best possible move of the algorithm remains common for all the games or at least it is easy to adapt the code to work with new type of game. The whole project is written in Java programming language.

4.2.1. Implementation

The main part of this project is to implement Min-max algorithm and its different variations. My approach is to separate min-max algorithm from games which uses this algorithm. For the sole purpose of min-max are responsible MinMax class and Node class. Also, there are two interfaces: Board and Move.

Board interface is responsible for providing methods like evaluateBoard(int player), makeMove(Move move), generateMoves(). Board interface is later implemented by game-specific classes like ConnectFourBoard and TicTacToeBoard. These classes are responsible for storing board state. Move interface provides methods getPlayer() and getMove().

Node class holds given Board, Move, list of Node children and value of evaluated board. This class is responsible for representing the game tree. The class with core logic is MinMax class. It provides methods to generate game tree from given board state and find the best move with min-max algorithm and alpha-beta pruning.

Algorithms for generating game tree and selecting the best possible move are presented below:

```
GenerateTree(Node node, int depth){
        if depth is zero or game is finished{
                set value of node to heuristic value of board
                return
        }
        for each move from possible mobes{
                make move on board to create newBoard
                create newNode with newBoard
                add newNode as child of node
                GenerateTree(newNode, depth - 1)
        }
}
```

Snippet 1. Pseudo-code of an algorithm for generating a game tree.

```
MaxValue(Node node){
        if node has no children{
                return value of node
        }
        int bestValue = -INFINITY
```

```
        for each child of node{
                value = MinValue(child)
                set value of child node to value
                bestValue = max(bestValue, value)
        }
        return bestValue
}

MinValue(Node node){
        if node has no children{
                return value of node
        }
        int bestValue = +INFINITY
        for each child of node{
                value = MaxValue(child)
                set value of child node to value
                bestValue = min(bestValue, value)
        }
        return bestValue
}
```

Snippet 2. Pseudo-code of a generic Min-Max algorithm.

Method generateTree(Node, int) from snippet 1 is responsible for creating a game tree from given state of the board. It works recursively and finishes when given depth is reached or the game is either won, lost or drawn. It takes all possible moves that can be done from given board state and for each of them, algorithm generates new board after given move is made. Then, a method calls itself with newly created node and decrements depth by one. When this algorithm reaches its end, there is created a game tree from given state and with given depth.

Now let us proceed to the MinMax algorithm itself. As we can see in the snippet 2, it consists of two similar methods: MinValue and MaxValue. These two methods are called recursively in rotation. MaxValue returns the biggest value from all children of given node, while MinValue returns the lowest value from given node's children. Thanks to this way of calling methods, at the end we obtain a game tree where each node has given value determined by a min-max algorithm. Now at the end, program should choose a child of a node, which has the highest value.

The algorithm presented above is a basic min-max algorithm. There is also enhanced version called alpha-beta cuts. Logic behind this variant was described in previous chapter. The pseudocode for this algorithm can be written as follows:

```
alphaBeta(Node node, int alpha, int beta, boolean maximizing) {
        if node has no children{
                return value of node
        }
        if maximizing player{
                for each child of node{
                        int tempVal = alphaBeta(childNode, alpha, beta, !maximizing)
                        alpha = max(tempVal, alpha)
                        childNode.setValue(alpha);
                        if alpha is greater or equal to beta{
                                return alpha
                        }
                }
                return alpha
        } else {
                for each child of node{
                        int tempVal = alphaBeta(childNode, alpha, beta, !maximizing)
                        beta = min(tempVal, beta)
                        if alpha is greater or equal to beta {
                                return beta
                        }
                }
                return beta
        }
}
```

Snippet 3. Pseudo-code for Alpha-Beta pruning algorithm. Generating a tree is done before applying this algorithm.

User has the possibility of choosing whether basic min-max, or alpha-beta pruning min-max should be used. Furthermore, the times of computing the best move using either algorithm are logged to a file. This will be helpful for further analysis.

4.3. Tic-tac-toe

Tic-tac-toe is relatively easy game to implement. The simplest approach would be to receive one of 9 possible inputs (there are 9 fields on the board), process the data and present new state of the board to the user. In my implementation, I provided graphical user interface (GUI) which is presented below. It consists of the board representation in the top part of the window, buttons to select given tile in the bottom part of the window and few utility buttons to the left. Use AI button changes whether user wants to use AI opponent or play in turns with another user. After clicking one out of nine buttons to choose a tile, proper token is placed at corresponding tile. When either player wins the game or the game results in a draw, proper message box is displayed.

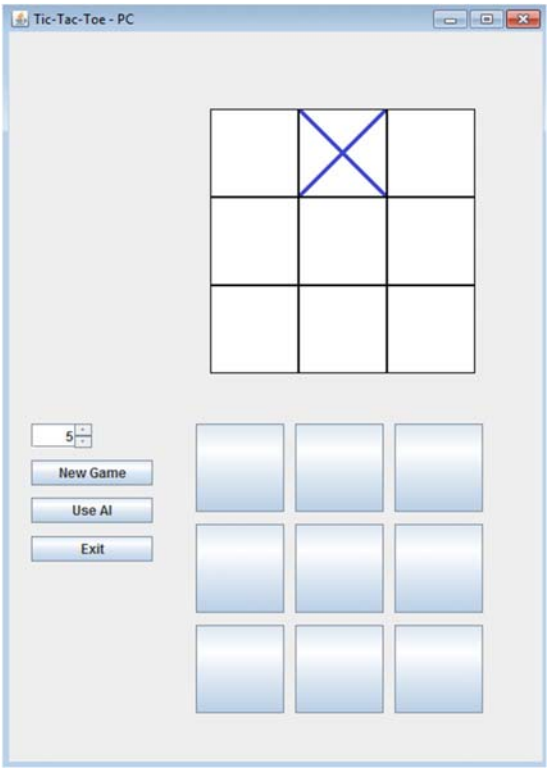

Figure 8. Screenshot from the program containing the main window for playing a Tic-Tac-Toe game. (source: self-prepared)

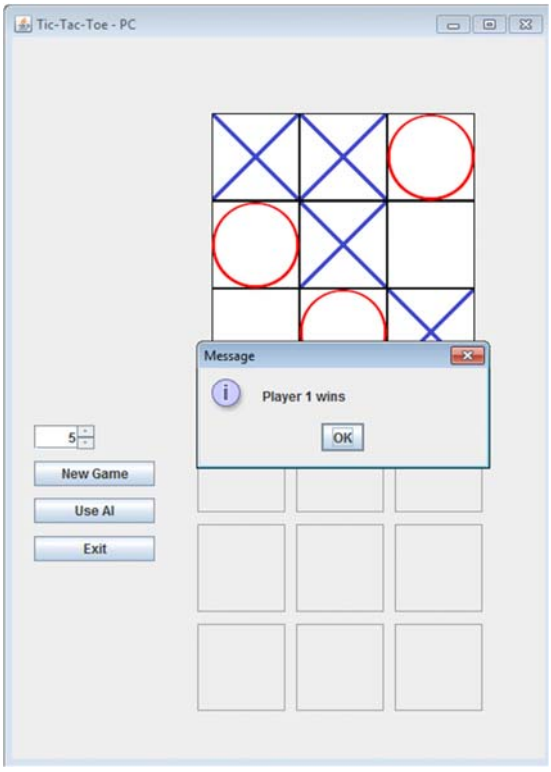

Figure 9. Screenshot from the program containing the win message. (source: self-prepared)

4.3.1. Heuristics

As the Tic-tac-toe game is very short game, which at most lasts 9 moves, the program can quickly reach bottom of the game tree. Knowing this, it is possible to bure force the graph and the heuristics can be very simple.

if player wins the board – return **+10**

else if player loses the board – return **-10**

else return **0**

Because of these heuristics, only moves that results in a won or lost game have non-zero value.

4.3.2. Implementation

The implementation of Tic-tac-toe game is stored in three classes: TicTacToeMainWindow, TicTacToeBoard, TicTacToeMove. The first one only works as a GUI for a user, so it only displays the window, receives an input and passes it to the TicTacToeBoard class where the whole logic lies, and displays the new, updated state of the board.

The second class is TicTacToeBoard. This class implements Board interface, so that MinMax class can operate on this class. TicTacToeBoard stores board state as a 2d array of integers with dimensions 3x3 and the player who is about to move. All methods which are provided by Board interface are properly implemented.

The third class is TicTacToeMove, which is just a holder class for a player and a move which he wants to perform stored as an integer.

4.3.3. Results

Presented results are acquired by running my program on two machines: Lenovo Y580 and HP EliteBook 840 G3. Depth of the algorithm was set to 9, which is maximum value of moves to be checked in tic-tac-toe game. Below table and graph show how many moves are possible for given move in game.

Move of the game	Number of moves to be checked
1	255 168
2	27 732
3	3 468
4	457
5	94
6	21
7	6
8	2
9	1

Table 2. Shows how many possible sequences of moves are possible in the Tic-Tac-Toe game during each turn of this game.

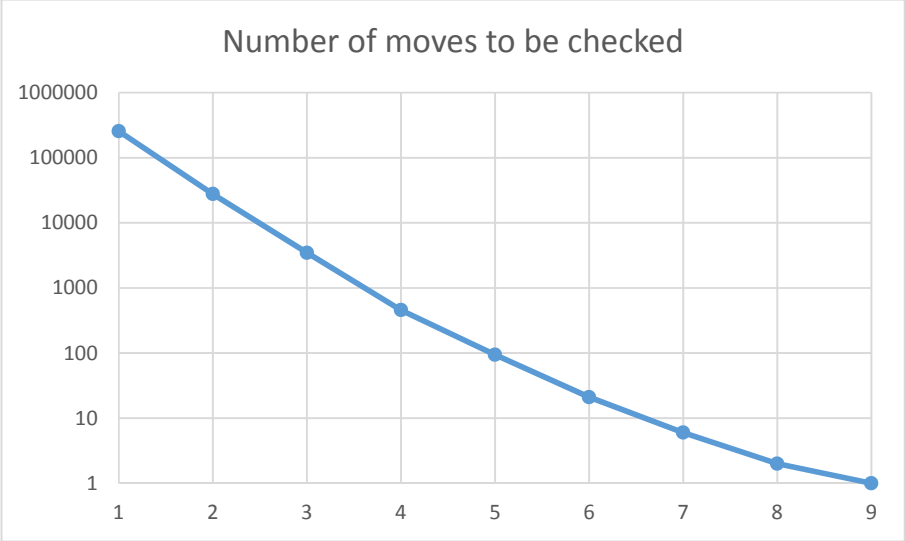

Figure 10. Graph showing logarithmic dependency of number of moves to be check for the given move od the game.

As we can notice for the tic-tac-toe game, number of moves to be checked are lowering exponentially according to the length of the game.

The table below shows how much time it takes my program to calculate which move to make. Generate Tree column says how much time it took to generate the game tree 9 moves deep. Then the MinMax column contains how much it took the algorithm to pick the best move. All the times shown in the table are in milliseconds.

Iteration	Depth	Checks	Generate Tree	MinMax	Sum
1	9	255 168	1245.00684	42.62264	1287.62949
2	9	27 732	43.32857	4.37095	47.69952
3	9	3 468	3.36871	0.44211	3.81081
4	9	457	0.43119	0.03618	0.46737
5	9	94	0.10395	0.00684	0.11079
6	9	21	0.04184	0.00329	0.04513
7	9	6	0.01855	0.00224	0.02079
8	9	2	0.01329	0.00171	0.01500
9	9	1	0.00934	0.00145	0.01079

Table 3. Presents data gathered during execution of basic Min-Max with depth set to 9 for a Tic-Tac-Toe game.

Data from this table can be also presented using graph. This will be helpful later in analysis of the results.

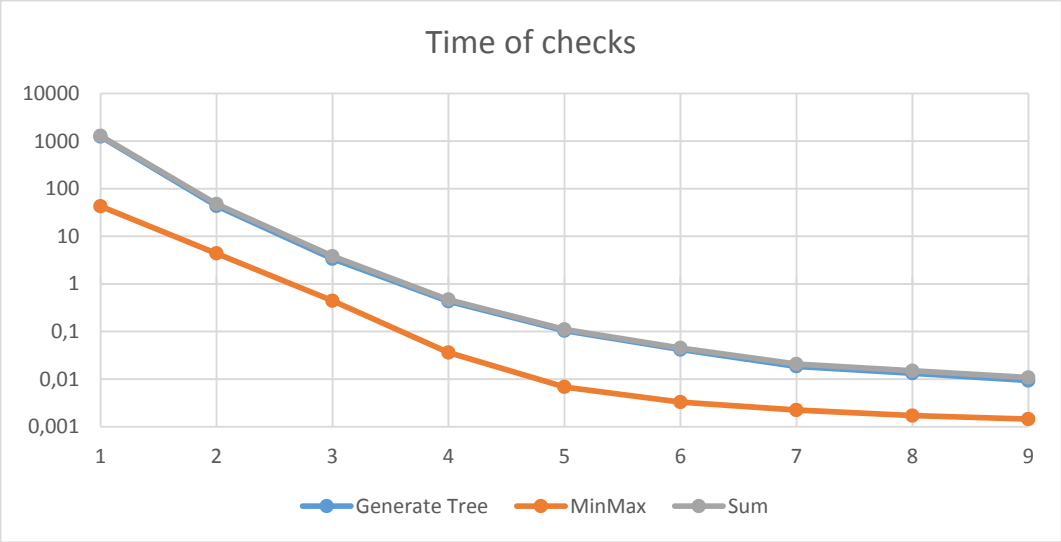

Figure 11. Logarithmic graph representing how much time it takes for a simple Min-Max algorithm to generate a game tree and to search through this tree to find best move.

During gathering results for simple Min-Max as well as Alpha-Beta pruning I found that most of the time is spent on generating a game tree. It means that my implementation of Alpha-Beta pruning algorithm is not working well. It only decreases time of searching through already constructed game tree. Because of that I implemented improved version of Alpha-Beta pruning, where generating a game tree is included in the Alpha-Beta pruning algorithm.

The snipped below shows the pseudo-code of new, better Alpha-Beta pruning version of a Min-Max algorithm.

```
improvedAlphaBeta(Node node, int alpha, int beta, boolean maximizing, int depth) {
        if depth is 0{
                return the heuristic value of board
        }
        if maximizing player{
                for each move from possible moves{
                        make move on board to create newBoard
                        int tempVal = improvedAlphaBeta(newBoard, alpha, beta, !maximizing,
depth-1)
                        alpha = max (tempVal, alpha)
                        if alpha is greater or equal to beta{
                                return alpha
                        }
                }
                return alpha
        } else {
                for each move from possible moves {
                        make move on board to create newBoard
                        int tempVal = improvedAlphaBeta(newBoard, alpha, beta, !maximizing,
depth-1)
                        beta = min(tempVal, beta)
                        if alpha is greater or equal to beta {
                                return beta
                        }
```

```
                }
                return beta
        }
}
```

Snippet 4. Improved version of Alpha-Beta pruning algorithm, where generation of moves and boards is moved into an algorithm itself.

Depth	Checks	Cuts	Time	Time/Checks
9	13 373	5 750	7.776857	0.00058
9	965	482	0.721196	0.00075
9	38	19	0.026665	0.00070
9	75	37	0.062766	0.00084
9	64	32	0.049228	0.00077
9	25	12	0.020923	0.00084
9	6	3	0.024614	0.00410
9	5	2	0.005333	0.00107
9	2	1	0.006154	0.00308

Table 4. Contains the time of finding best possible move together with generating a game tree. Moreover, it contains how many moves were checked and how many branches were cut.

The table above presents results of using this implementation of Alpha-Beta algorithm. As we can see from the table, the time of execution fell drastically.

4.4. Connect-four

My implementation of Connect four game is similar to the Tic-tac-toe implementation mentioned earlier. It takes as an input one of seven possible moves, process data and display the results. The GUI is also designed similarly as in the previous game. The main part represents the current state of the board, at the bottom there are seven input buttons, and to the left there are the utility buttons.

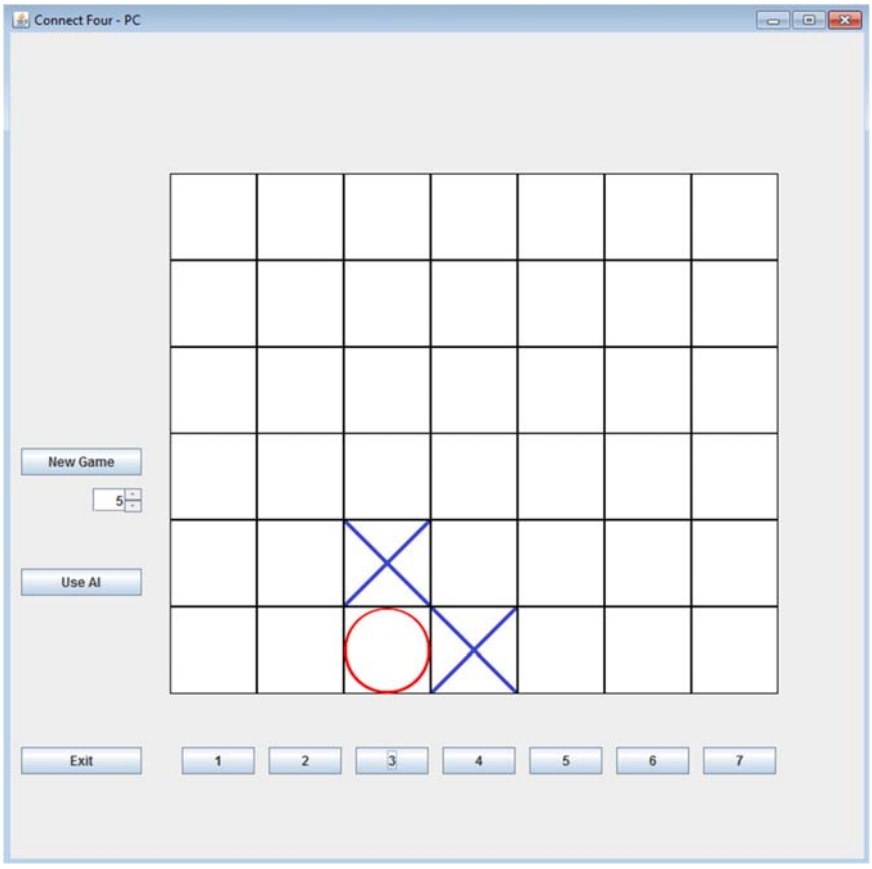

Figure 12. Screenshot from the program containing the main window for playing a Connect-Four game. (source: self-prepared)

In Connect-Four game there are 42 fields on the board, so the game can take at most 42 moves and at least 7 moves. Each field can take one of 3 values (empty, player1, player2), so the upper bound of the number of possible board states is $3^{42} \sim\sim 10^{20}$. However, many of these positions are illegal and the real upper bound is about 10^{12}. If the algorithm was supposed to search through all these positions it would take very long time. In my implementation, user can select how many moves ahead (how deep) the algorithm should look. This functionality allows us to greatly reduce the computation time, but also reduces the accuracy of selecting the best possible move. The table below shows how the number of nodes in game tree depends on the depth.

Depth	Checked moves
1	7
3	343
5	16 807
7	823 543
9	40 353 607

Table 5. Presents how many moves needs to be checked for given depth in Connect-Four game

4.4.1. Heuristics

Connect-Four game is more complex game than Tic-tac-toe, so the same heuristics cannot be used. With the knowledge, that brute-forcing search would take too much time, some other heuristics need to be designed. Each two-in-row, three-in-row and four-in-row most likely approaches the player to a win of the game. Moreover, tokens placed more centrally results in more possible four-in-row moves. Thanks to this, my heuristics look as following:

each two-in-row – value = value + 10
each three-in-row – value = value + 100
if there is four-in-row - value = 10000000
each opponent's two-in-row – value = result - 10
each opponent's three-in-row – value = result - 100
if there is opponent's four-in-row - value = -10000000

Thanks to these heuristics, all possible states of the board have their values. The algorithm does not need to compute all the moves at once, but only the given number of moves.

4.4.2. Implementation

My Connect-Four game consists of 3 classes: ConnectFourMainWindow, ConnectFourBoard, and ConnectFourMove. Like in the previous implementation, ConnectFourMainWindow is responsible for providing proper GUI to the user, ConnectFourMove represents the move to be done. ConnectFourBoard is the core of the Connect-Four game. It implements Board interface with all its required methods.

4.4.3. Results

The results for Connect-Four game are also gathered by running the program on two machines: Lenovo Y580 and HP EliteBook 840 G3.

As mentioned earlier, in Connect-Four game, game tree can grow to very big size. Because of it we cannot check each possible move down to the end of game. The tables below show how much in milliseconds it takes to compute the best possible move for search with depth 7 in the example game. Generate Tree column contains how much it took for the program to generate the game graph of given depth, then MinMax column says how much it took to run basic MinMax over this game graph. The last column states how much time on average it took to calculate one move.

Depth	Checks	Generate Tree	MinMax	Sum	Time/Sum
7	823494	5000.763	59.060046	5059.8227	0.0061
7	822696	2777.629	41.074636	2818.7035	0.0034
7	815094	2990.136	41.067662	3031.2038	0.0037
7	606438	2595.779	37.321334	2633.1007	0.0043
7	598703	1529.608	30.930178	1560.5384	0.0026
7	559286	1700.515	37.146982	1737.6623	0.0031
7	557816	1575.222	29.114037	1604.3358	0.0029
7	512211	1869.427	27.141596	1896.5686	0.0037
7	512127	1395.575	27.042727	1422.6175	0.0028
7	504462	1297.595	26.271062	1323.8658	0.0026
7	465822	953.2398	25.857129	979.0970	0.0021
7	421757	1034.332	23.832586	1058.1649	0.0025
7	215915	425.5311	13.272552	438.8036	0.0020
7	139839	397.2186	9.51884	406.7375	0.0029
7	37457	89.03805	2.671084	91.7091	0.0024
7	19761	53.86351	1.845678	55.7092	0.0028
7	4515	14.39538	0.514853	14.9102	0.0033
7	420	5.258479	0.132918	5.3914	0.0128

Table 6. Presents data gathered during execution of basic Min-Max with depth set to 7.

As we can see from the table, number of moves to check decreases as the game goes further. Because of this all the times also should decrease.

The graph below represents how much time it takes to check one move with the set depth of 7. As we can see from this graph, time to check one move does not depend on how many moves the algorithm is about to check.

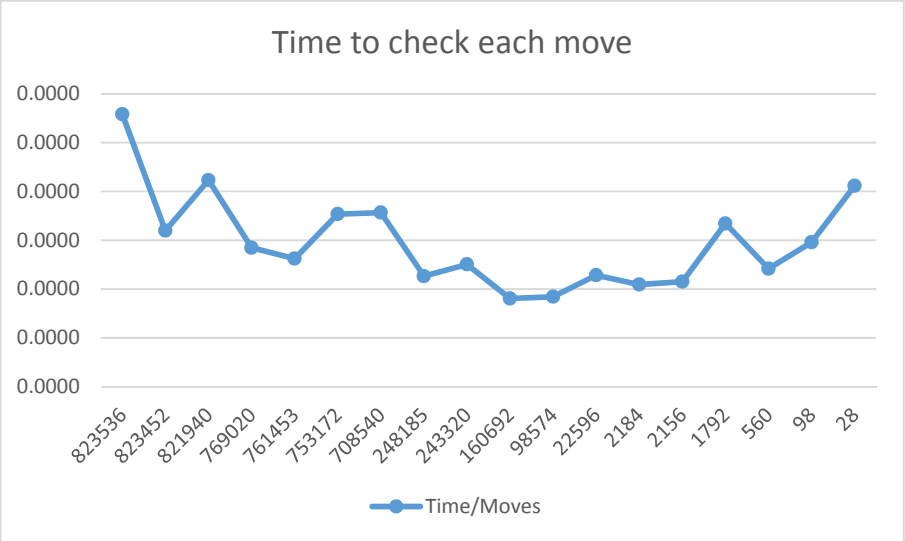

Figure 13. Graph representing how much time it takes to check one possible move using simple Min-Max, depending on the amount of checked moves in given run.

The table below shows how much time in milliseconds on average it takes to compute best possible move depending on the set depth. Average time of finding best move increases with greater depth, however the average time to check one move decreases.

Depth	Average time to check one move	Average time to find best move
3	0.003169	1.272
4	0.003057	8.771
5	0.002727	51.875
6	0.002398	267.181
7	0.002235	3 224.028

Table 7. Average time to check one move, depending on the depth of search.

All previous results were gathered using generic Min-max algorithm. Now let us focus on alpha-beta pruning version of a Min-max algorithm.

Depth	Checks	Generate Tree	MinMax	Sum	Time/Moves
7	823536	4590.3634	6.2626	4596.6260	0.0056
7	823452	2629.0482	3.3147	2632.3628	0.0032
7	821940	3473.7334	4.1921	3477.9255	0.0042
7	769020	2186.4648	2.7313	2189.1961	0.0028
7	761453	1987.7427	10.0322	1997.7749	0.0026
7	753172	2658.3283	4.5129	2662.8413	0.0035

7	708540	2522.7784	4.3238	2527.1023	0.0036
7	248185	556.1601	5.7153	561.8754	0.0023
7	243320	606.0769	3.7799	609.8568	0.0025
7	160692	287.2358	3.1013	290.3371	0.0018
7	98574	179.0113	2.6140	181.6253	0.0018
7	22596	50.9448	0.6568	51.6016	0.0023
7	2184	4.5359	0.0336	4.5696	0.0021
7	2156	4.6290	0.0115	4.6405	0.0022
7	1792	5.9738	0.0168	5.9906	0.0033
7	560	1.3492	0.0049	1.3542	0.0024
7	98	0.2868	0.0033	0.2900	0.0030
7	28	0.1136	0.0016	0.1153	0.0041
7	1	0.0185	0.0008	0.0193	0.0193
7	1	0.0172	0.0008	0.0181	0.0181

Table 8. Presents data gathered during execution of Alpha-Beta pruning Min-Max with depth of 7.

The same as we did in the generic Min-max, below is the graph representing how much time it takes to check one move using my implementation of alpha-beta pruning min-max algorithm.

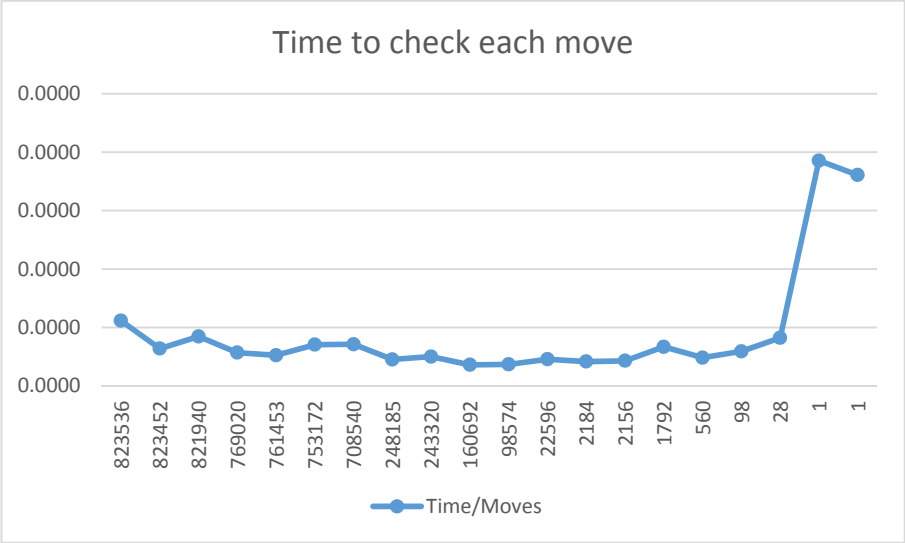

Figure 14. Graph representing how much time it takes to check one possible move using Alpha-Beta pruning, depending on the amount of checked moves in given run.

It is noticeable, that the time of finding best node in the graph lowers. However, the time of generating a game tree stays the same. It is dictated by the fact, that in my implementation generation of the tree takes place before a Min-max calculation.

Because of this, I implemented another version of alpha-beta pruning min-max algorithm. Now It moves generation of a game tree into the search through the game tree. It improved results drastically. Below is a table representing how much time it took for new alpha-beta implementation, next to the number of moves checked and number of cut branches.

Depth	Checks	Cuts	Time [ms]	Time/Checks [ms]
7	16 910	4 271	46.79	0.00277
7	8 636	1 378	16.07	0.00186
7	19 236	5 216	21.50	0.00112
7	28 285	8 481	36.65	0.00130
7	20 441	5 457	29.53	0.00144
7	11 570	2 140	17.32	0.00150
7	24 091	6 119	22.82	0.00095
7	23 305	6 062	21.06	0.00090
7	18 482	3 875	22.34	0.00121
7	23 580	5 500	27.87	0.00118
7	16 028	3 326	16.85	0.00105
7	19 469	4 588	19.94	0.00102
7	11 880	2 853	9.10	0.00077
7	5 128	1 283	6.96	0.00136
7	915	257	1.42	0.00155

Table 9. Presents how much time it takes to find best move, together with the number of check and cuts using better version of an Alpha-Beta pruning algorithm.

There we obtain huge boost in performance of the program. However, the time it takes to check one move stays similar to previous implementations. Each cut lowers the amount of checked nodes by more than one. That is why values in the ‘Checks’ column are so small compared to previous implementations.

Depth	Checks	Cuts	Time[ms]
5	982	149	6.95
6	6 984	2 210	11.10
7	13 167	2 540	22.37
8	123 509	37 496	147.11
9	167 564	39 037	137.90
10	952 511	272 730	887.16
11	1 849 059	522 285	1 926.24
12	16 501 759	4 949 789	13 807.39
13	31 786 281	10 805 507	27 591.89
14	335 138 373	101 386 483	285 578.33

Table 10. Average number of checks, cuts and average time of finding best move for given depth using improved Alpha-Beta pruning implementation in Connect-Four game.

5. Analysis of results

This chapter contains an analysis of all gathered results from previously performed experiments. The results were collected by running a program on two machines. Lenovo Y580 and HP EliteBook 840 G3. However, the results were identical, so only results from HP EliteBook 840 G3 are were presented in this thesis.

5.1. Tic-tac-toe

Tic-tac-toe game is very simple game. It means that the game tree is relatively small. It contains only 255 168 nodes. Because of this, it is possible to search through the whole tree and find the best possible move very fast. The only move that takes more than 50 milliseconds to compute using simple Min-Max is the first move.

Implementing improved version of Alpha-Beta pruning worked very good. It is working about 150 times faster than the generic Min-Max for the Tic-Tac-Toe game. This gives us almost instant response from the computer player when we play a game.

I have run the computer vs computer using my program many times for depth set to 9 and it always resulted in a draw. Then I started running computer vs computer for different depths. The results are interesting.

	Player 1									
Player 2	Depth	1	2	3	4	5	6	7	8	9
	1	Player 1	Player 1	Player 1	Player 1	Player 1	Player 1	Player 1	Player 1	Player 1
	2	Player 1	Player 1	Player 1	Player 1	Player 1	Player 1	Player 1	Player 1	Player 1
	3	Player 1	Player 1	Player 1	Player 1	Player 1	Player 1	Player 1	Player 1	Player 1
	4	Player 2	Draw 0	Draw 0	Draw 0	Player 1	Player 1	Player 1	Player 1	Player 1
	5	Player 2	Draw 0	Draw 0	Draw 0	Player 1	Player 1	Player 1	Player 1	Player 1
	6	Player 2	Draw 0	Draw 0	Draw 0	Draw 0	Draw 0	Draw 0	Draw 0	Draw 0
	7	Player 2	Draw 0	Draw 0	Draw 0	Draw 0	Draw 0	Draw 0	Draw 0	Draw 0
	8	Player 2	Draw 0	Draw 0	Draw 0	Draw 0	Draw 0	Draw 0	Draw 0	Draw 0
	9	Player 2	Draw 0	Draw 0	Draw 0	Draw 0	Draw 0	Draw 0	Draw 0	Draw 0

Table 11. Result of computer vs computer. First column and first row tells what depth was set to each player.

After looking at the results presented in the table 11, it is clear that Player 1 (moved first) won about 46% of all the games, while Player 2 (moved second) won only about 8% of his games. The other 46% of games resulted in a draw.

It is also worth mentioning, that Player 2 won only if Player 1 depth was set to 1. It means that Player 1 didn't even try to interrupt Player 2.

5.2. Connect-Four

Connect-Four is more complex game than a Tic-Tac-Toe. The maximum length of Connect-Four game can reach up to 42 moves, which gives us about 10^{12} possible games. Also, for great part of the game player can make one of seven possible moves. What is more, evaluation of the board is more complicated, because instead of 3 pieces we need to build 4 in a row. Taking all of that into account, it is not possible to evaluate all possible board states maintaining small computation times, so more advanced heuristics need to be used.

Running a simple Min-Max using HP EliteBook 840 G3 worked only up to depth of 7. After selecting the depth to 8, my implementation failed to calculate anything due to high memory usage. Only after using improved Alpha-Beta pruning I managed to run the algorithm for greater depths. As table 10 shows, the time for computing best possible move for depth set to 7 took an average of 22 milliseconds, while using a simple Min-max algorithm it took about 3000 milliseconds. That is about 135 times improvement, so improvement is similar to the one in Tic-Tac-Toe game.

When the depth of search was set to 14, my implementation of the algorithm still worked, but it took about 4 minutes and 45 seconds to calculate one move. With this depth set, recreational play with computer takes a lot of time.

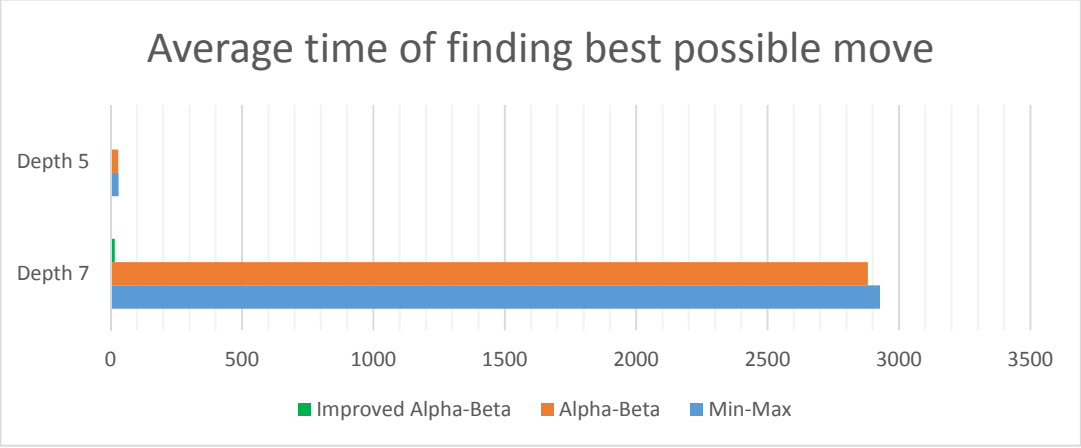

Figure 15. Graph representing how much time it take for given algorithm to find best possible move for given depth.

6. Conclusions

The purpose of this thesis was to introduce to the reader foundations of an Artificial Intelligence (AI) in logic games, what does it mean to solve a game, and also how a Min-Max algorithm together with an Alpha-Beta pruning variant works. I presented two approaches to an Alpha-Beta variant, one almost useless approach and the proper one. The first approach generated the whole game tree for given depth like in normal Min-Max algorithm. It took most of the time required to compute best possible move. Later, I managed to correctly implement it and combine generating a game tree with the Min-Max algorithm. This approach shortened the time needed for a program by over 100 times.

Another thing to watch out for is testing the implemented AI using properly written game which have no bugs in it. During the implementation of given algorithms, a lot of time was lost on debugging an application to only find out, that it contains a bug which makes the game unwinnable. However, I finally managed to locate all the bugs and games are working correctly.

In the future, people will most likely create artificial intelligence for games which are unbeatable for human being, or all possible logic games will be solved. After that, we – people – will only sit and observe how two machines compete against each other. However, until that happens people can still beat AI in many complex logic games like Chess or Go. However even in these games more and more times the AI wins again humans.

7. Bibliography

[1] Paul Walker, *Zermelo and the Early History of Game Theory*, 10.08.2017, http://www.econ.canterbury.ac.nz/personal_pages/paul_walker/pubs/zermelo-geb.pdf

[2] Ronald L. Rivest, *Game Tree Searching by Min/Max Approximation*, 10.08.2017, https://people.csail.mit.edu/rivest/pubs/Riv87c.pdf

[3] Paulo Pinto, *Introducing the Min-Max Algorithm*, 08.09.2017, http://www.progtools.org/games/tutorials/ai_contest/minmax_contest.pdf

[4] Jeanne Boyarsky and Scott Selikoff, *Oracle Certified Associate Java SE 8 Programmer I Study Guide*, 2015, ISBN: 978-1-118-95740-0