FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

BACHELOR THESIS

Lucie Kunčarová

Artificial Intelligence for the Hex Game

Department of Theoretical Computer Science and Mathematical Logic

Supervisor of the bachelor thesis:	Mgr. Martin Pilát, Ph.D.
Study programme:	Computer Science
Study branch:	General Computer Science

Prague 2024

I would like to thank my supervisor, Mgr. Martin Pilát, Ph.D., for his valuable insights and advice, as well as a lot of patience and time. I would also like to thank my family and friends for supporting me every step of the way.

Title: Artificial Intelligence for the Hex Game

Author: Lucie Kunčarová

Department: Department of Theoretical Computer Science and Mathematical Logic

Supervisor: Mgr. Martin Pilát, Ph.D., Department of Theoretical Computer Science and Mathematical Logic

Abstract: Hex is a strategic, two-player board game. The goal of this work is to design and implement the Hex game itself and a couple of AI agents that are able to play Hex. First, we analyzed Hex from the game theory point of view and described some of the popular AI algorithms used for similar games. We also listed some of the current research progress in the field of exploring Hex and creating AI agents for it. Finally, we implemented the three AI agents and experimentally tested which performed the best and with which parameters. The AI agents implemented are: a heuristic agent using heuristics to make its moves; a more advanced minimax agent using the minimax algorithm with alpha-beta pruning; and a Monte Carlo Tree Search agent using the probabilistic Monte Carlo Tree Search approach.

Keywords: Hex, board game, artificial intelligence, minimax algorithm, monte carlo tree search

Contents

Introduction

Humans have enjoyed playing board games ever since we could remember, long before the time of computers. Therefore, it is no surprise that there are hundreds of new and creative board games created every year.

Nowadays, we have the luxury to enjoy and challenge ourselves even on our own, whenever we feel like it. And that is exactly what this thesis aims to do. Provide the right opponent for everybody in the game of Hex.

Hex is a two-player abstract strategy game invented by the Danish mathematician Piet Hein in 1942. What is interesting about the discovery of the game is that the famous American mathematician John Nash supposedly also invented the game independently in 1948. And since John Nash played Hex a lot during his studies at Princeton, it is also known as Nash. We can also glimpse John playing Hex in his bibliographical movie, The Beautiful Mind.

As the interest of famous mathematicians suggests, there are some complex qualities of Hex worth studying. And many people did; there are already plenty of mathematical findings and very good AIs that have already beat the best players in the world. In this thesis, we will first implement the game itself, summarize the current research progress, and also build a few of our own artificial intelligences capable of playing Hex against both real players and other AIs. With the aim of making the game more accessible to people by giving them the possibility to choose the right difficulty. Because even though many excellent AIs already exist, they might be a bit too difficult to play against or take too long to make a move for real-time games, considering their complexity.

For implementing the game, we will use Python, and it will be possible to either play against an AI or a local friend.

1. The Hex Game

In this chapter, we will describe how to play the game of Hex and later analyze its properties.

1.1 Introduction to Hex

Hex is a captivating and strategic board game. Traditionally played on a symmetrical 11x11 rhombus board with hexagonal cells. But in theory even asymmetrical boards with unequal side lengths are possible, as well as any other dimensions, 13x13 and 19x19 being particularly popular and studied.

The game was first published in a Danish newspaper, Politiken, on December 26, 1942. The name of the game went through some changes before becoming known as Hex. The inventor, Peter Hein, originally named the game Polygon in his article. Later, it was marketed in Denmark under the name Con-tac-tix. And finally almost ten years later, it was popularized by the Parker brothers as Hex, probably due to the nature of the game board.

The game has very simple rules but the strategy is rather complex. There is no element of chance and has a perfect information structure. The players need to anticipate their opponent's moves, block strategic paths, and create their own winning strategies.

1.2 Rules

As mentioned above, Hex is a two-player strategy game traditionally played on 11x11 rhombic board with hexagonal cells. Each player is assigned a color, usually red and blue or black and white for high contrast, but any color combination is possible. Both players are also assigned two opposite sides of the game board, all the four corner cells belong to both adjacent sides.

Players then take turns in coloring one of the blank cells with their allocated color, claiming it as theirs. It is not allowed to repaint an already claimed cell or skip a turn entirely. The goal of the game is then very simple; link your two opposite sides of the game board by a connected path consisting of cells painted in your color. The player to connect their sides first wins. It is common for the loser to resign when the winner is clear, even though the game is not officially over yet.

1.3 Analysis

Now that we know how Hex works, let us analyze the properties of the game. It is essential for later choosing the right algorithms for both its implementation and the design of its AI agents.

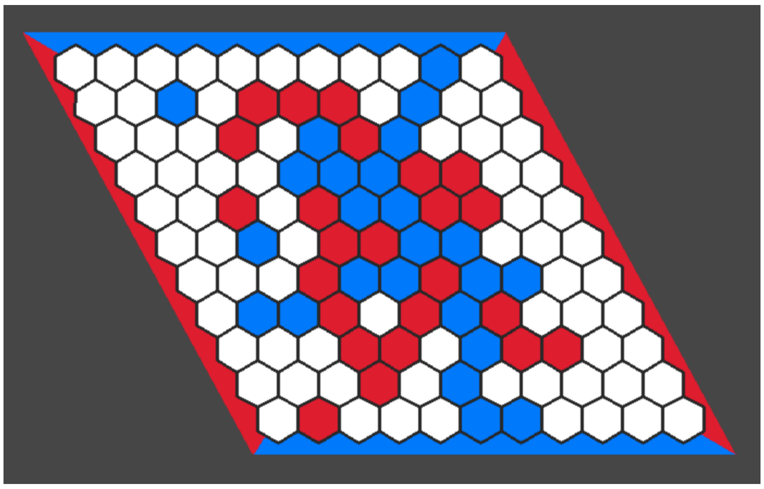

Figure 1.1: A finished 11x11 Hex game where Blue won

1.3.1 Game Theory Point of View

Hex is a zero-sum game, meaning that one player's gain is exactly the other player's loss, so the total value of the outcomes for both players sums to zero.

Hex is also a game with perfect information. Both players are perfectly aware of all the moves that have been played and also of all the possible moves that can be played at any given moment.

Hex is a deterministic game. Since there is no element of chance or surprise, the outcome is fully determined by the players' choices and actions.

Hex is a non-cooperative game. There is no reason to cooperate with the other player under any circumstances.

Hex is a symmetric game if we choose to focus on the more common version of the game. Meaning the available strategies and payoffs for both players are the same. In other words, in terms of their roles, the players are indistinguishable, and they face the same choices and outcomes.

1.3.2 Complexity

In 1981, a German mathematician Stefan Reisch proved that Hex is PSPACE-complete [Reisch, 1981]. Meaning that it is at least as hard as the hardest problems in polynomial space complexity. This implies that finding an optimal strategy algorithmically can be computationally challenging.

However, we have to take into consideration that the board size of the game has a great impact on the computational complexity of the game. Conveniently, Hex can be represented as a tree graph, but we have to be careful because when we increase the board size, the branching factor also increases.

1.3.3 Properties

What is interesting about the game is that it can never end in a draw. Meaning that even if all the cells are claimed by one of the players, there always exists a connected path linking either the left and right sides of the board, or the top and bottom sides, with their respective colors, of course. This phenomenon is also known as the Hex theorem, and it was known to even John Nash. The proof is non-trivial, and there are many different ones. For example a proof by Gale [1979].

As a consequence of the Hex theorem, defending is also attacking. If a player at every move plays defensively; blocking the other player from connecting their sides of the board, this strategy inadvertently leads to the defensive player's victory.

Another very important property of the game is that the first player has a winning strategy, as proven by John Nash in 1952. To remedy this fact, a swap rule can be added. When the second player is about to make their first move, they can choose to either play normally or swap colors with their opponent, and the game continues with the first player playing again, but with the swapped colors.

1.3.4 Basic Strategies

As mentioned above, there are many Hex strategies, and none of them are too straight-forward. But after taking a closer look at the structure of the game, we can observe a few things.

One of the simplest ways to advance is by using a two-bridge. Two-bridge is a diamond-shaped formation where a player has claimed two cells that are two cells apart, as seen in Figure 1.2. We can observe that the opposing player cannot later prevent the red player from joining their cells in a two-bridge, no matter what. Any possible rotation of the shown two-bridge is again a valid two-bridge.

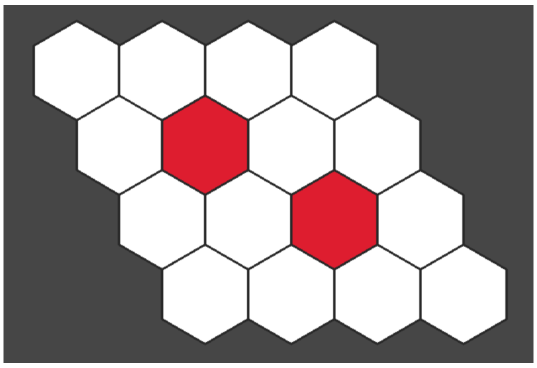

Figure 1.2: A red two-bridge

And as a result, it is very beneficial to use such bridges in the game, as a player can advance across the game board twice as fast as when connecting the adjacent cells. For visualization, see Figures 1.3 and 1.4.

While advancing using two-bridges is a good practice, it is only good at a local level and can also be blocked. Let us also hint at some more general principles. We can observe that a position is only as good as its weakest link. Therefore, a player should generally try to either make their weakest link stronger or make their opponent's weakest link even weaker. Achieving both at the same time is ideal.

For illustration, let us look at Figure 1.5. For the blue player, the cell marked with a black star is their weakest link, as they used two-bridges for the rest of the

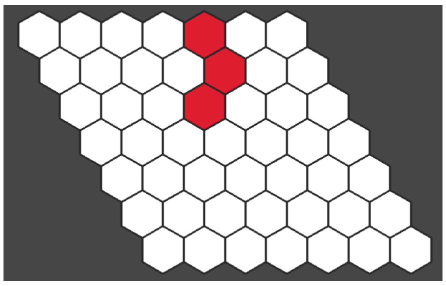

Figure 1.3: Adjacent expanding

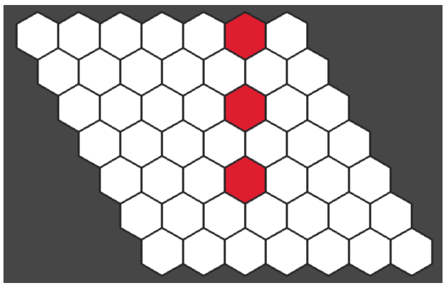

Figure 1.4: Bridge expanding

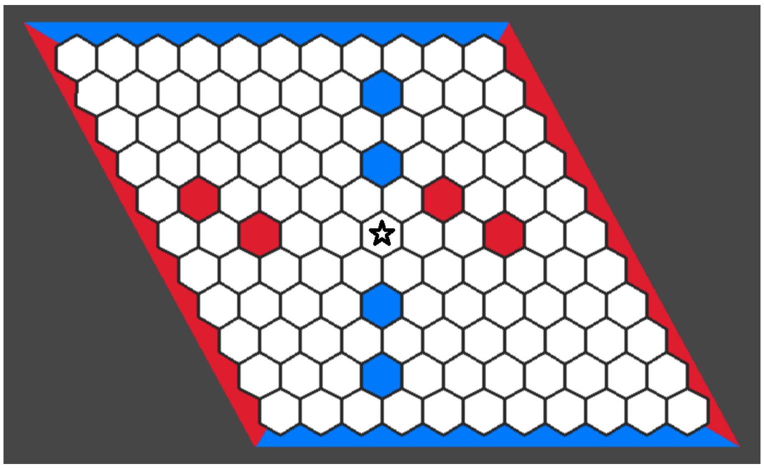

Figure 1.5: Weakest link

game board, assuring their future connection. Simultaneously, the cell marked with a black star is also the red player's weakest link for similar reasons. As a result, whoever will play next will be smart to claim the black star cell because it is both theirs and their opponent's weakest link. Specifically for this scenario, whoever claims the black star cell will also win the whole game.

Many other local and more general strategies can be adapted. Their specification and analysis in more detail was described by [Browne, 2000].

2. Artificial Intelligence

In this chapter, we will look into artificial intelligence approaches suitable for the game of Hex and later used in its implementation. To be more specific, we will be implementing an agent. An agent is an entity designed to autonomously make decisions in order to reach a specific goal. They are able to perceive their environment and act accordingly. In our case, the environment is the current state of the Hex board, and the goal is to win the game by following its rules.

Let us look at some algorithms and approaches that can be used by such agents. The description of some of the agents is inspired by a book by Russell [2010].

2.1 Minimax

The Minimax algorithm is a popular decision-making algorithm used for two-player games. Its main idea is to find an optimal move for a player while assuming the opponent also plays optimally. The name ”Minimax” reflects the strategy of minimizing the maximum possible loss.

Minimax is a type of backtracking and recursive algorithm, and it represents the game as a tree. One of the players is typically called a maximizer, and the other a minimizer. The maximizer tries to get the highest possible score, while the minimizer the lowest. Every game state can be assigned a value, which tends to be higher when the game state is favorable for the maximizer and lower when it is favorable for the minimizer. The value itself is computed using some sort of heuristics that are specific to different kinds of games.

The algorithm itself works in the following way:

1. Generate game tree.

   The current game state is used as the root of the game tree, and then the tree itself is generated in the following way: We recursively take into account all possible moves, and the game states that are the result of the moves being played are added as children to their parent nodes. This way, we recursively continue until terminal game states are reached, and these will be our leaves.

2. Assign values to leaves.

   All the leaves are assigned a value using a static evaluation function.

3. Assign values to internal nodes.

   In this step, we will backtrack from the leaves back to the root. Proclaiming the root as a node at depth zero, we do the following: For each internal node at an even depth, it is the maximizer’s turn, so we choose the child with the highest value and use this value for our current node. On the other hand, if the node is at an odd depth, it is the minimizer’s turn, and we choose the lowest value of all our children.

4. Choosing a move.

   Looking at the children of the root, we choose the child with the highest value, and as our move, we choose the move that led to this child.

When applied to the Hex game, which is deterministic, zero-sum, and finite, Minimax actually guarantees finding the optimal move. However, that is only true when the game tree is fully searched. Which poses a big problem for a game with such a big branching factor, making it basically impossible to use in its current state. Therefore, if we want to use Minimax for one of our agents, we have to modify it to make it faster.

2.1.1 Alpha-Beta Pruning

In the first improvement, we will show that not all branches of the game tree need to be explored. It is based on the idea that if we find a move that proves to be worse than an already previously examined one, there is no need to examine it further because the Minimax algorithm will never choose it anyway. And therefore, we can cut the tree branch there, saving precious computational time. This cutting of branches is called pruning.

The pruning process can be applied at any level of the tree and involves two additional parameters, typically called alpha and beta, hence the name alpha-beta pruning.

The parameters are defined as:

- Alpha:

  The best choice (highest value) that can be found on the path of the maximizer, at that level or above. The alpha parameter is initialized as negative infinity, $-\infty$.

- Beta:

  The best choice (lowest value) that can be found on the path of the minimizer, at that level or above. The beta parameter is initialized as positive infinity, $+\infty$.

The alpha-beta pruning then works in the following way:

1. As we traverse the tree using the Minimax algorithm, we keep track of the values of alpha and beta.
2. When we are examining a node, we explore all possible moves from that node. If it is the maximizer's turn and we find a move with a better value than alpha (higher than alpha), we update the alpha parameter with this value.
3. If it is the minimizer's turn and we find a move with a better value than beta (smaller than beta), we update the beta parameter with this value.
4. At every node we examine, we check whether alpha becomes greater or equal to beta. If yes, then the maximizer has a choice that is at least as good as a choice available to the minimizer at a higher level. That means the current node will never be chosen by Minimax, and its entire subtree will never be traversed; therefore, it can be pruned.

Note that the alpha value is only modified by the maximizer, and the beta value is only modified by the minimizer. And when backtracking the tree, it is the child nodes' values that are passed on to their parents, not the alpha or beta values themselves.

Since alpha-beta pruning does not influence the decision process of the Minimax algorithm, both standard Minimax and Minimax with alpha-beta pruning return the same output, given the same input. But since the game tree can only get smaller with alpha-beta pruning, the algorithm can only get faster.

An example of a pruned game tree can be seen in Figure 2.1.

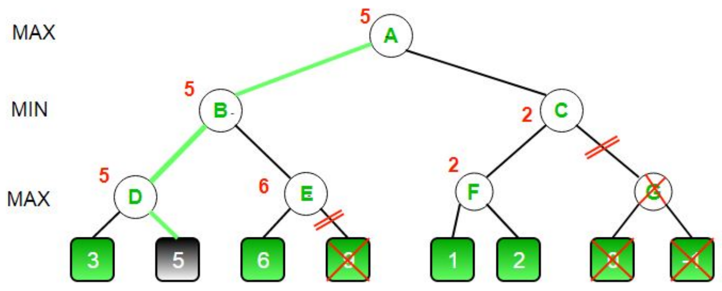

Figure 2.1: Minimax with alpha-beta pruning example. Source: Shevtekar et al. [2022]

In this specific example in Figure 2.1, two branches were pruned. First, the right child of node E, because the minimizing player in node B is choosing the minimum of its children, and its options so far before evaluating the right child of node E are: 5 from D and 6 from E. And because E is a maximizing player, the final number can only be either 6 or something higher; otherwise, E would not choose it. B will therefore always choose the 5 from D, making the value of the right child of E irrelevant and therefore pruned. The other branch was pruned similarly, even managing to prune the whole sub-tree.

2.1.2 Depth Restriction

Because the branching factor of Hex is very large, Minimax with alpha-beta pruning would still take too long to finish. Therefore, it still needs to be modified for our purposes. One such modification is restricting the maximum depth we can reach while searching the game tree. We will only search up to a previously fixed depth and then treat the nodes at such depth as terminal nodes, meaning they will be assigned a value by some evaluation function. These values will then be back-propagated as in standard Minimax.

By restricting the depth, we can easily influence the computation time of the algorithm, but we also lose the guarantee of an optimal move being chosen.

Generally speaking, the larger the depth we are allowed to explore, the better the results produced by the Minimax algorithm. So not only is restricting the depth good for efficiency management, but it can also be used to influence the difficulty of a game played against such a Minimax AI agent.

2.1.3 Negamax

Before we get to the implementation, let me introduce a variant of Minimax. Negamax makes use of the zero-sum property of Hex. So because one player's gain is exactly equal to the other player's loss, we do not need a maximizer and a minimizer, but a procedure that maximizes the negation of the values of its children and chooses a move accordingly.

Negamax has the same efficiency and performance as Minimax; it is just commonly used for simplification of the implementation process.

2.1.4 Pseudo Code

Here is a pseudo code for the modified Minimax algorithm.

```
function NEGAMAX(player, board, depth, alpha, beta)
    best_value ← $-\infty$
    if depth = 0 or board.is_game_over() then
        value ← EVALUATE(player, board)
        return value
    end if
    moves ← board.get_moves()
    for each move in moves do
        board.make_move(move)
        SWITCH_PLAYERS(player)
        new_value ← NEGAMAX(player, board, depth − 1, −alpha, −beta)
        new_value ← −new_value
        if new_value > best_value then
            best_value ← new_value
        end if
        board.take_move(move)
        alpha ← max(alpha, best_value)
        if alpha $\geq$ beta then
            break
        end if
    end for
end function
```

The first call of the algorithm would be, for example for the first player and depth equal to three:

NEGAMAX(1, board, 3, $-\infty$, $+\infty$)

2.2 Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) [Chaslot et al., 2008] is another algorithm suitable for Hex. It is in some sense similar to Minimax, as it also represents the game as a tree and gives its nodes values. However, unlike Minimax, MCTS does not search the entire game tree for all possible states but only takes into consideration the states that, in a way, guarantee some success. It is based on Monte Carlo strategies, which rely on random sampling and statistical evaluation using tree search techniques.

MCTS dates back to 2006, when Rémi Coulom presented using the Monte Carlo method on a game-tree search. Nowadays, it is a very popular algorithm for all sorts of board games. In combination with artificial neural networks, MCTS was also used in 2015 in the famous AlphaGo program by Google Deepmind [Silver et al., 2016], which is known for being the first program to ever beat a professional Go player on a 19x19 board.

The key idea of MCTS is to incrementally build and refine a game tree, focusing mainly on parts of the tree that are more likely to be favorable for the current player. It is randomly simulating game play-outs and updating node values based on the play-outs' results. Over time, MCTS converges to an optimal move decision without having almost any knowledge about the game itself.

A big advantage over Minimax is that MCTS does not require any domain-specific knowledge. In order to implement a good Minimax algorithm with alpha-beta pruning, a good evaluation function is needed. Therefore, a deep knowledge of the game itself and its strategies is needed. Whereas by evaluating the game tree nodes based on random simulations, when implementing a MCTS algorithm, no domain knowledge is required.

The algorithm itself is divided into four main parts: selection, expansion, simulation, and back-propagation. They are executed exactly in this order and repeated. Figure 2.2 shows the algorithm schema.

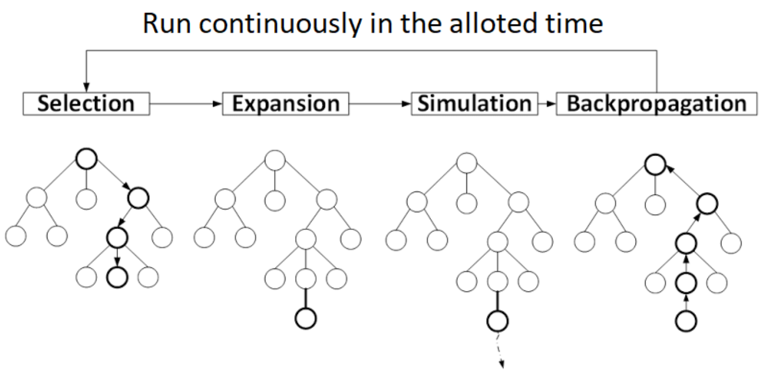

Figure 2.2: MCTS Algorithm Schema. Source: Świechowski and Tajmajer [2021]

2.2.1 Selection

The aim of selection is to, by traversing the game tree from its root, select the best node. The selection aims to balance exploration, not-yet-well-explored nodes, and exploitation, the best-explored nodes so far. It is done using a formula called the upper confidence bound for trees.

$$UCT(v) = \frac{w(v)}{n(v)} + c \times \sqrt{\frac{\ln N}{n(v)}}$$

Where:

- $UCT(v)$ is the upper confidence bound for node v.

- $w(v)$ is the total number of wins of v.
- $n(v)$ is the number of times node v has been visited.
- N is the total number of times the parent node of v has been visited.
- c is the exploration parameter.

The first part of the formula corresponds to exploitation, as it favors nodes with a high win ratio. The second part of the formula corresponds to exploration; it favors nodes with few visits. The exploration parameter is a constant, and its value is usually chosen close to $\sqrt{2}$.

So the child node that returns the highest UCT(v) value when traversing the tree is the one that will get selected.

2.2.2 Expansion

The goal of expansion is to expand our already-chosen node through the selection process. Expanding means adding children to this node, similarly as in Minimax, by going through all possible moves from the chosen node and adding the resulting states of the game as its children. In cases of limited memory or time, not necessarily all children have to be added; only one child will also suffice. If we select the same node in the future, more children can be added if they exist.

2.2.3 Simulation

In simulation, we randomly choose a child of the originally selected node; if only one child was created during expansion, we choose this one. From this node, we run a random full play-out of the game. This means playing uniformly randomly until the terminal state of the game is reached.

2.2.4 Back-propagation

The aim of back-propagation is to pass on the simulation result information back to its parents and grandparents. Since Hex is a two-player game, we can assign these values to the node from which we ran the simulation based on how the simulation ended.

- Win = 1
- Loss = 0

Figure 2.3 shows how back-propagation works. For the sake of visualization, the values of the nodes are:

$$\frac{\text{wins}}{\text{games played}}$$

2.2.5 Move Choosing

The last step of the algorithm is to choose the best move to play. This is done by looking at the children of the root, as they represent possible moves from our initial game state. We choose the child with the highest number of visits because

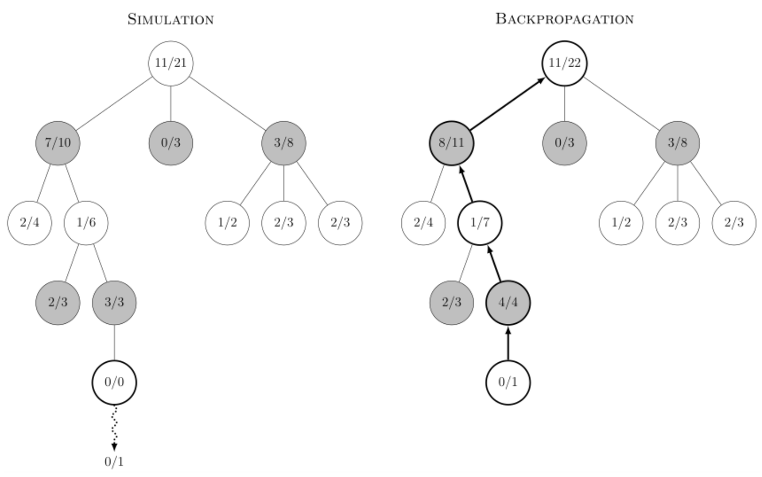

Figure 2.3: MCTS Back-propagation. Source: Moss [2020] Licensed under CC BY-SA 4.0.

this node was thoroughly explored and simulated and therefore must have a high probability of success.

2.2.6 Pseudo Code

Here is the pseudo code for the Monte Carlo Tree Search algorithm.

```
function MCTS(root)
    while resources left do
        leaf ← SELECT(root)
        if leaf not fully expanded then
            EXPAND(leaf)
        end if
        node ← SELECT_RANDOM_CHILD(leaf)
        if node is terminal then
            BACKPROPAGATE(node, node_winner)
        else
            result ← SIMULATE(node)
            BACKPROPAGATE(node, result)
        end if
    end while
    best_child ← FIND_BEST_CHILD(root)
    return best_child
end function

function SELECT(root)
    node ← root
    while node has children do
        for each child in node_children do
            if child is unvisited then
                return child
            end if
            child_uct ← CALCULATE_UCT(child)
        end for
        node ← randomly choose one of the children with the max UCT value
    end while
    return node                                              ▷ leaf found
end function

function EXPAND(node)
    possible_moves ← get all moves that can be played on the current board
    for each move in possible_moves do
        new_node ← play move on current board
        add new_node to node children
    end for
end function

function SIMULATE(node)
    curr_board ← make a copy of the node board
    while not game over do
        move ← choose random move that can be played on curr_board
        make move on curr_board
    end while
    return winning player
end function

function BACKPROPAGATE(node, result)
    increase number of visits of node
    if root player won then
        increase number of wins of node
    end if
    BACKPROPAGATE(node_parent, result)
end function

function FIND_BEST_CHILD(root)
    return child with the most visits
end function
```

When the turn of a MCTS player comes, we simply get the move to be played as:

move ← MCTS(current game board state)

3. Related Work

This chapter focuses on the current progress of Hex AI research. We will briefly mention some remarkable projects whose goal was to develop the best possible AI agents for playing Hex. The first ever Hex playing agent has been created by Claude Shannon and E.F. Moore in 1953 [Shannon, 1953]. Since 2000, an international Hex competition for Hex AIs organized by ICGA is being held annually and is called the Computer Olympiad [International Computer Games Association, 2021]. Board sizes used at the Computer Olympiad are 11x11 and 13x13.

3.1 Polygames: Improved Zero Learning

Polygames [Tristan Cazenave, 2020] is an open-source framework for zero-shot learning with multiple libraries, one of which is for Hex. Zero-shot learning is a machine learning practice where a model is trained to recognize classes it has not seen during training. Polygames uses a combination of Monte Carlo Tree Search and Deep Learning. It uses clever tricks to achieve board-size invariance. For example, by using computer vision, global pooling, and fully convolutional neural networks. The whole framework was inspired by DeepMind's AlphaZero, which is a version of an already-mentioned AlphaGo.

Their AI was, for example, able to beat strong human players in 19x19 Hex, including Arek Kulczycki, a player with the best ELO rank (2248 at that time) on LittleGolem (a popular online game system where games are played on a web browser, one turn at a time). Polygames also won against another framework, Galvanise-Zero[1]. A single Quadro GP100 was used with a 5 minute per move limit.

Polygames is very good, not only in Hex, as it can be trained for many other board games. Peak performance is usually achieved after nine days of training. However, it is unsuitable for our purposes as it requires too much time to make a move and probably would be too good to play against rookie players.

3.2 DeepEZO

EZO [Takada, 2019] is an algorithm that also represents the game as a game tree and uses iterative deepening depth-first search [Korf, 1985] created by Kei Takeda in 2013. It was very successful in the Computer Olympiad from 2013 to 2018, ranking among the top three multiple times. Over the years, EZO used different evaluation functions to achieve good results: a combination of global and local evaluations, later value and policy functions consisting of 12 network characteristics. In 2018, came to be the most successful version called DeepEZO [Takada et al., 2020], using value and policy functions based on convolutional neural networks. DeepEZO uses the minimum search width of three and only searches nodes with high probability values computed by the evaluation function.

[1] `https://github.com/richemslie/galvanise_zero`

3.3 MoHex-3HNN

MoHex [Henderson, 2010] is an algorithm using Monte Carlo Tree Search, as explained in Section 2.2. It has won first place at the Computer Olympiad in years from 2009 to 2011. In 2013, a group of researchers improved MoHex and introduced MoHex2.0 [Arneson et al., 2011]. They used Hex game records played by expert players to learn board patterns. Also compared to the already-introduced UCT parameter used in the tree search in Section 2.2.1, MoHex2.0 uses a smart parameter tuning called Confident Local Optimization [Coulom, 2011] that discards samples with an estimated value confidently inferior to the mean of all samples. MoHex2.0 was a big success, and it held the title of Hex world champion of the Computer Olympiad in the years from 2013 to 2017. Another improvement came in 2017 when the prediction model was combined with deep convolutional neural networks to create MoHex-CNN [Gao et al., 2017]. In 2017, MoHex-CNN beat Mohex2.0 at Hex 13x13 and won the gold medal. In 2018, MoHex-CNN was further improved by the same group of researchers, which leads us to the most successful version so far called MoHex-3HNN [Gao et al., 2018], which uses a neural network model called Three Head Neural Network. And it became the Hex world champion in 2018, outperforming MoHex-CNN.

3.4 Wolve

Wolve [Henderson, 2010] is an algorithm using a bit different approach than the previously mentioned algorithms. It models Hex positions as electrical resistor circuits and uses the resistance values of the circuit as the evaluation function. It is called Shannon's electric circuit evaluation function [Anshelevich, 2001]. As for the tree search algorithm, iterative deepening depth-first search is adapted. From 2006 to 2011, it was very successful in the Computer Olympiad, winning the one in 2008 and then placing second in the other years. Wolve is the best computer Hex player using Shannon's evaluation function and alpha-beta pruning, as explained in Section 2.1.1.

4. Framework

In this chapter, we will talk about the implementation of the game and the AI agents. The game is programmed in Python, a general-purpose, high-level programming language. The code itself is segmented into classes, therefore using an object-oriented approach. We will briefly describe the classes used and explain the overall structure of the project.

4.1 Game Implementation

Since the project can be used as the game only, without using the AI agents, we will first describe the implementation and structure of the game and its graphical user interface (GUI).

4.1.1 Cell

Cell is a class representing one hexagonal cell on the game board. It holds all the necessary information we need to clearly identify a cell on the game board. Its coordinates on the game board, and the player who is occupying the cell. When playing with a graphical interface, even more information is stored, like the vertices coordinates, which are needed to detect whether the player has clicked on that particular cell.

4.1.2 GameBoard

GameBoard is a class representing the game board. Internally, the game board is a two-dimensional array of hexagonal cells.

The GameBoard also contains a function responsible for checking for a game-over. The method chosen to effectively check whether the game has ended is Union Find.

Union Find
Union Find is a disjoint-set data structure [Galler and Fischer, 1964]. It categorizes objects into different sets and chooses a representative element to denote the set. The set is usually represented as a tree graph, and the root of the tree serves as the representative element. So in order to check whether two elements are in the same set, we just need to compare their roots. In order to make this operation fast, we need to keep the respective trees shallow. This can be done by using path compression. The basic idea is to make each node on the path from a node to the root directly point to the root. To find the root of a node, we need to traverse up the tree until we find the root. During the traversal, every node encountered is updated to point directly to the root.

In our case, we store a collection of trees that contain occupied hex cells that are connected on the game board. For determining whether a player has won, we check if the player's respective borders are connected (in the same set). For player one (the red player), the borders that need to be connected are left and right, whereas for player two (the blue player), they are top and bottom. To achieve even better efficiency, we only start checking for a winning player once at least

$$2 \times \min(\mathit{board\_width}, \mathit{board\_height}) - 1$$

moves have been played, because otherwise the game cannot be over yet.

The GameBoard is also responsible for making all the moves appropriately and checking their validity.

4.1.3 GUI

GUI is a class handling all graphical matters. It draws the current state of the game board as well as detects mouse clicks and clicked cells done by the user when it is their turn to play. It also converts the coordinates of these mouse clicks to two-dimensional array positions so they can be used by the internal representation of the game board, GameBoard, to make the move. After the move was played in GameBoard, the GUI uses the current state of the GameBoard to redraw the new state of the game.

GUI is therefore a class that encapsulates GameBoard and Cell. And they all communicate to achieve a visually pleasing Hex game.

A set of Python modules called Pygame [1] was used for the implementation GUI.

4.1.4 GUIGame and Game

GUIGame and Game are classes containing the main logic for controlling the flow of the game. They are two different classes because the game has two modes: a game with a graphical user interface using GUI, or a game in the console using text outputs. For that reason, whenever a game is played, only one of these classes is used. They have the same functionality, and the only difference is that GUIGame uses GUI as its internal game board representation and Game uses GameBoard.

4.1.5 Player

Player is a parent class of all types of players this project offers. The possible players are:

- Human player
- Random player
- Heuristic Player
- AI player using Minimax
- AI player using Monte Carlo Tree Search

Player has two mandatory functions for getting a move: one is for the console version of the game, and the other is for a game with GUI.

A child class of Player called HumanPlayer is relevant when the user themselves wants to play. For the console version, the user is required to manually enter

[1] https://www.pygame.org

the valid coordinates of the move they want to play. Whereas for the game with GUI, they need to mouse-click on the position of their desired cell.

The other child classes will be described more in detail in AI Implementation.

4.1.6 Main

Main is the execution point of the whole application. It contains some examples of how the game can be run as well as all the information the user needs in order to play their desired game or watch the AIs play against each other.

4.2 AI Implementation

Now let us look into the implementation of all the AI agents.

4.2.1 RandomAI

The most simple but useful agent implemented is a random player. With no improvements, it just randomly finds an empty position and plays there. Even though it is possible to play against a random player, this agent is more commonly used as a tool for testing the other, more clever agents. A random player is very fast, and we can see how the currently tested opposing agent behaves in different scenarios because it plays randomly. On the other hand, if we want to see a certain behavior of the tested agent more in depth, we can also fix the random seed of the random player and get the same sequence of moves.

4.2.2 HeuristicAI

A heuristic player is an agent that, given the current state of the game board, evaluates all the possible moves and then chooses the best one. Therefore, the performance depends solely on the choice of the heuristic evaluation function.

The evaluation function used is based on the connectivity of the board with the use of Dijkstra's algorithm.

Dijkstra's Algorithm
Dijkstra's algorithm [Dijkstra, 1959] is a path-finding algorithm invented in 1956 by a Dutch computer scientist, Edsger W. Dijkstra. The goal of the algorithm is to find the shortest path from a node to all the other nodes. It iteratively chooses the closest unvisited neighbor and updates the distances to all its neighbors based on that path length.

The evaluation function assigns a value to the given state of the game board. Specifically, it initializes a matrix with the same dimensions as the game board, where each cell's value represents the least number of moves required to connect to the opposite side. The values are obtained using Dijkstra's algorithm. We then simply take the minimum value in the first column (for player one) or row (for player two) as the connectivity score. Ultimately, the connectivity scores of both the current player and the opposing player are calculated, and their difference is taken as the final value for that given state of the game board.

After computing the scores for all the possible moves on the game board. The heuristic player simply chooses to play the position with the maximum value. Since it is beneficial for the current player to get a higher score from the evaluation function.

4.2.3 MinimaxAI

The Minimax agent is, in a way, a more advanced version of the heuristic agent. It uses the modified version of the Minimax algorithm described in Section 2.1, specifically Negamax with alpha-beta pruning. It also uses the exact same evaluation function as the heuristic player, described in Section 4.2.2.

Since both the Minimax and the heuristic agent use the exact same evaluation function, its logic is contained in a special class serving as a library called **Heuristics**.

The advantage of the Minimax agent over the heuristic player is that it can search the game tree to a bigger depth, which should, in theory, make it ultimately better, but it can also take way longer.

4.2.4 MCTSAI

The last implemented AI agent uses a Monte Carlo Tree Search approach described in Section 2.2. For its implementation three other classes were used.

MCTSBoard
In order to run many simulations during a turn, the game board representation needs to be as light as possible. That is why a lite version of the original game board, called MCTSBoard, is created. It converts the original representation to a NumPy array for faster computation. NumPy [2] is a Python library specifically designed for scientific computations, making many array operations significantly faster than in Python itself. It also makes use of a modified version of Union Find, a new class **MCTSUnionFind**, that does not support taking moves back, as the one in the original game board does. Since unlike in Minimax, in MCTS we are only adding the moves.

MCTSNode
When building the Monte Carlo Search Tree a class MCTSNode representing a single node is used. It contains all the necessary MCTS steps, selection, expansion, simulation and back-propagation. This allows us to separate the logic when building the tree in the main MCTSAI class.

MCTSAI then contains only functions for choosing the best move. Its computation time can be either limited by a specific number of simulations or by time.

4.3 Experiments

The last part of the framework focuses on running many experiments to see how well the AI agents play against one another. For these purposes, the class **Experiment** exists. It uses the console version of the game because of its better

[2]https://numpy.org

time efficiency. One experiment is designed to simulate 100 Hex games and save the results to a CSV file. The results include four columns of data: player one and the games it won, player two and the games it won, the number of moves that were needed for the game to be over, and the time in seconds the game lasted.

To run an experiment, it is necessary to create an instance of the Experiment class in the main class with all the desired parameters and then execute the function experiment(). The detailed format of the parameters is described in the Python file itself, along with some examples.

Here is a brief overview of them:

1. **file_name**: string name of the CSV file to save the results in
2. **AI player one**: "minimax_ai", "mcts_ai", "heuristic_ai" or "rand_ai"
3. **AI player two**: "minimax_ai", "mcts_ai", "heuristic_ai" or "rand_ai"
4. **height**: integer from 3 to 15
5. **width**: integer from 3 to 15
6. optionally **minimax_depth**: integer, recommended range is 1 to 3, default value=2
7. optionally **mcts_time_allowed**: float, default value=9999999
8. optionally **mcts_number_of_simulations**: integer, default value=9999999

Some examples:

```
experiment1 = Experiment("heuristic_vs_mcts_time5_11x11.csv",
"heuristic_ai", "mcts_ai", 11, 11, mcts_time=5)
experiment1.experiment()

experiment2 = Experiment("mcts_vs_minimax_depth1_sims4000_7x7.csv",
"mcts_ai", "minimax_ai", 7, 7, minimax_depth=1,
mcts_simulations=4000)
experiment2.experiment()

experiment3 = Experiment("heuristic_vs_minimax_depth2_7x7.csv",
"heuristic_ai", "minimax_ai", 7, 7)
experiment3.experiment()
```

4.4 How to Run

Excluding the experiments, the game itself can be run from the command line. There are two versions of the game:

1. **With Graphical Interface (GUI)**
   In the command line, run as:
   ```
   python main.py gui height width player1 player2
   --minimax_depth depth --mcts_time time
   --mcts_simulations simulations
   ```
2. **Without GUI in console**, mainly for testing AIs

In the command line run as:

```
python main.py console height width player1 player2
--minimax_depth depth --mcts_time time
--mcts_simulations simulations
```

Where:

- `height` and `width` are integers from 1 to 15 expressing the dimensions of the board
- `player1` and `player2` are player options described below in player options.
- `depth` is an optional integer parameter for the Minimax agent, setting its search depth. The default value is set to 2.
- `time` is an optional float parameter for the MCTS agent, setting time allowed per move in seconds. The default value is set to 9999999.
- `simulations` is an optional integer parameter for the MCTS agent, setting number of simulations per move. The default value is set to 9999999.

Player options:

1. **Human player**: Use string ”human”.
    - The format of a move in console is ”row, column” as integers.
2. **Random player**: Use string ”rand_ai”.
3. **Heuristic player**: Use string ”heuristic_ai”.
4. **Minimax player**: Use string ”minimax_ai”.
5. **Monte Carlo Tree Search player**: Use string ”mcts_ai”.

For the MCTS player, the parameter that runs out first is the one that will end the move. If the user sets only one, the other is automatically set to a very high number (9999999), so it does not interfere with the user's choice. However, if the user does not set any of these parameters, a default of 10000 simulations will be set to prevent the code from running for a very long time.

5. Experiments

In this chapter we will look into how good the AI agents are when playing against each other and discuss the results.

For every combination of agents, multiple experiments were conducted. One experiment consists of a simulation of one hundred games, and since playing first is a clear advantage, every experiment was also repeated with a swapped first and second player. Because many agents contain parameters that influence their performance, experiments are repeated for a selected few of them. The last thing taken into account is game board size. All experiments are therefore repeated for symmetric game boards of sizes 7x7 and 11x11.

For every two ordered agents tested, there are two tables, one for results on a game board of size 7x7 and the other for size 11x11. Each table has the following format: player one and its variations in column one, and player two and its variations in column two. The third column is for results, and it is divided into two parts. The first is the ratio of games won by each agent, with the parameters set next to the agent's name on the current row. The ratio specifically is *number of wins of the first player:number of wins of the second player*, and since the total number of games is 100, the numbers can also be viewed as percentages. The other part of the result column is the average number of moves the agents took before the game was over.

All the experiments were run on a personal computer with the following technical specifications:

- **Processor (CPU)**: 11th Gen Intel(R) Core(TM) i7-1165G7 @ 2.80GHz 2.80 GHz
- **Memory (RAM)**: 16 GB DDR4 3200
- **Graphics Processing Unit (GPU)**: NVIDIA GeForce MX450 2 GB
- **Storage**: 1 TB SSD PCle
- **Operating System**: Windows 11 Home, Version 23H2

5.1 Heuristic AI vs. MCTS AI

For the combination of the heuristic player and the MCTS player, the only parameter we can set is the time we allow the MCTS agent to make its move. The two following tables show the results for experiments simulated on the game board of size 7x7.

Player 1	**Player 2**	**Result**	
Heuristic	MCTS 1s	39:61	19.7 moves
Heuristic	MCTS 2s	19:81	18.8 moves
Heuristic	MCTS 4s	13:87	17.6 moves
Heuristic	MCTS 6s	8:92	16.9 moves

As we can see, the MCTS agent performs much better on this small board game size, even though it has the disadvantage of the second player. As we allow the

MCTS agent more time, the games also resolve with fewer overall moves.

Let us move on to the game board of size 11x11.

Player 1	Player 2	Result	
Heuristic	MCTS 1s	100:0	22.9 moves
Heuristic	MCTS 2s	98:2	24.7 moves
Heuristic	MCTS 5s	68:32	34.6 moves
Heuristic	MCTS 10s	23:77	38.8 moves
Heuristic	MCTS 15s	17:83	39.5 moves

When we increased the board size, the MCTS agent needed much more time to beat the heuristic one. Even not winning a single match with an allowed time of one second. As expected, with more time, the performance of the MCTS improves, but waiting for 15 seconds while playing against a player in real time seems too much.

5.2 MCTS AI vs. Heuristic AI

Let us now look how the results change when the MCTS agent gets the chance to play first. Starting with game board 7x7.

Player 1	Player 2	Result	
MCTS 1s	Heuristic	72:28	20.0 moves
MCTS 2s	Heuristic	91:9	20.3 moves
MCTS 4s	Heuristic	91:9	17.8 moves
MCTS 6s	Heuristic	98:2	16.7 moves

As expected, the MCTS agents fares a little better compared to when it played second. And it is safe to say that it performs better than the heuristic player on the 7x7 board.

The following table is for game board 11x11.

Player 1	Player 2	Result	
MCTS 1s	Heuristic	0:100	24.9 moves
MCTS 2s	Heuristic	0:100	23.6 moves
MCTS 5s	Heuristic	48:52	38.6 moves
MCTS 10s	Heuristic	80:20	43.1 moves

For one and two seconds per MCTS move, it seems it does not matter which agent plays first, as the heuristic player wins almost all the matches. The MCTS agent simply does not have enough time to compute a good enough move. But for the allowed five seconds and more, there is a noticeable difference in the results when the MCTS agent plays first. For five seconds, winning almost half the games as compared to 32% when playing second.

5.3 Heuristic AI vs. Minimax AI

When conducting experiments for the heuristic and the Minimax agent, only the parameter of Minimax depth can be set. And as the game of Hex has a high branching factor, the maximum depth tested is three.

Comparing the heuristic and the Minimax agent is interesting because they both use the same evaluation function, and the only difference between them is the depth they search into. Therefore, in theory, the Minimax agent with depth set to one should be identical to the heuristic agent. It will be tested experimentally in the following tables, starting with a game board of size 7x7.

Player 1	**Player 2**	**Result**	
Heuristic	Minimax d=1	52:48	18.9 moves
Heuristic	Minimax d=2	20:80	17.2 moves
Heuristic	Minimax d=3	17:83	16.8 moves

For the first experiment in the table, a higher win rate was expected for the first player, but as only 100 games were simulated, the relatively small difference could be a part of the statistical error. But as expected, the deeper the Minimax agent is allowed to search, the better it performs, even with the disadvantage of being the second player.

Next, we have the results on a board game of size 11x11.

Player 1	**Player 2**	**Result**	
Heuristic	Minimax d=1	53:47	33.3 moves
Heuristic	Minimax d=2	16:84	29.8 moves

With the increase in game board size, the minimax search time also increases. And as the results for minimax with depth equal to two are already very good with a win rate of 84% and the disadvantage of the second player, the experiment for minimax with depth equal to three will not be conducted in this case.

In both experiments in the table, both the ratios are similar to those run on the 7x7 board, and same as for the 7x7 experiments, when minimax searches deeper, the game is over in less moves on average, suggesting that it plays a little more smartly.

5.4 Minimax AI vs. Heuristic AI

Next, let us compare how the results change when we swap the order of the players. Starting with the table for the game board 7x7.

Player 1	**Player 2**	**Result**	
Minimax d=1	Heuristic	57:43	19.3 moves
Minimax d=2	Heuristic	96:4	15.7 moves
Minimax d=3	Heuristic	98:2	17.1 moves

For minimax with depth equal to one, the first player has the upper hand again, same as when the order of the players was swapped. The results are a little in favour of the Minimax agent, which again is just a statistical flaw. When the search depth is increased, minimax dominates the games already at depth equal to two, also taking on average fewer moves to win.

For a game board of size 11x11, the results are the following:

Player 1	**Player 2**	**Result**	
Minimax d=1	Heuristic	53:47	32.8 moves
Minimax d=2	Heuristic	96:4	30.8 moves

For the same reason as in the previous chapter, depth equal to three is not included.

Interestingly, the win ratio for the first experiment is exactly the same as when the agents played in the opposite order, supporting the theoretical result that the heuristic agent and the Minimax agent with d = 1 are, in fact, the same agent.

5.5 Minimax AI vs. MCTS AI

Probably the most interesting pair to experiment on is the Minimax and the MCTS agent, as they are programmed to be the two best-performing ones. The parameters we can set are again the depth of search for the Minimax agent and the allowed time per move for the MCTS agent. This could be unfair since the parameter of the MCTS agent depends on the performance of the computer it is being run on, so a few more experiments with a new parameter number of simulations per move will also be conducted. These are the results for a game board of size 7x7:

Player 1	**Player 2**	**Result**	
Minimax d=2	MCTS 2s	69:31	16.8 moves
Minimax d=2	MCTS 3s	65:45	16.2 moves
Minimax d=2	MCTS 5s	60:40	16.0 moves
Minimax d=2	MCTS 7s	58:42	15.2 moves
Minimax d=2	MCTS 10s	57:43	15.3 moves
Minimax d=2	MCTS 15s	40:60	15.3 moves
Minimax d=2	MCTS 3000 sim	33:17	16.3 moves
Minimax d=2	MCTS 5000 sim	33:17	15.8 moves
Minimax d=2	MCTS 10000 sim	32:18	15.7 moves
Minimax d=3	MCTS 2s	81:19	18.3 moves
Minimax d=3	MCTS 6s	83:17	18.3 moves

The Minimax agent seems to be playing better than the time-limited MCTS agent, as most of its win rates are above 50%. The MCTS agent starts beating the Minimax agent with d=2 only with 15s per move, which is quite a lot for an amateur real-time game on such a relatively small board size. There seems to be not much improvement for the MCTS agent when it gets an allowed time of 5s, 7s, or 10s. It is important to note that, due to time reasons, some of the experiments for this particular pair of agents were conducted on a different personal computer with similar technical specifications.

For the experiments with a fixed number of simulations, only 50 game simulations were run for a single experiment. So the result cannot be viewed as a win percentage in this case. We see that the win rate remains really similar, suggesting that it can do 10,000 simulations in around 3 seconds. The number of simulations could therefore be increased even more.

For minimax d=3, only two experiments were run, as this setting for minimax is a little too time-consuming for our purposes, and the two experiments that were run with this depth had similar win rates. For similar reasons, no experiments for the Minimax agent with d=3 were run for the game board of size 11x11, and every experiment consists of only 50 game simulations. Here are the results:

Player 1	Player 2	Result	
Minimax d=2	MCTS 2s	50:0	23.5 moves
Minimax d=2	MCTS 6s	44:6	27.2 moves
Minimax d=2	MCTS 8s	37:13	30.1 moves

Even 15s for a MCTS move were not enough to beat the Minimax agent. But the Minimax agent also took its time to make a move. On closer inspection, computed from the additional time data in the experiment, the Minimax agent took on average 41s to make its move. Making it compete against the MCTS agent with a maximum allowed time of only 15 seconds seems unfair. The Minimax player, therefore, might not be suitable to play against in an amateur match as it takes too long to make a move, and its implementation does not provide any means of making it play faster or stopping mid-search as the MCTS does.

5.6 MCTS AI vs. Minimax AI

Let us now swap the playing order of the agents and again note that some of the experiments were run on a different computer. Starting with the results on board 7x7:

Player 1	Player 2	Result	
MCTS 2s	Minimax d=2	62:38	18.6 moves
MCTS 3s	Minimax d=2	75:25	18.0 moves
MCTS 5s	Minimax d=2	74:26	16.4 moves
MCTS 7s	Minimax d=2	82:18	15.3 moves
MCTS 10s	Minimax d=2	84:16	15.9 moves
MCTS 15s	Minimax d=2	88:12	14.9 moves
MCTS 3000 sim	Minimax d=2	37:13	19.0 moves
MCTS 5000 sim	Minimax d=2	37:13	17.4 moves
MCTS 10000 sim	Minimax d=2	42:8	16.9 moves
MCTS 2s	Minimax d=3	46:54	20.6 moves
MCTS 6s	Minimax d=3	62:38	18.2 moves

The advantage of the first player seems to be more important than ever for this pair of agents, as the MCTS agent has a higher win rate for all of the matches and their chosen setting except for Minimax d=3 and only two allowed seconds per move.

As for the experiments in simulations, again, only 50 game simulations are included in a single experiment. And judging by the similar win rates again, we can see that it takes about 3–5 seconds to make 5000 simulations and about 7 seconds to make 10,000 of them. The fact that it takes a little longer than when the agents played in the opposite order might be due to the fact that the computers' computation power was consumed by some other processes at the time of the experiment's execution. And again, it would be a good idea to run more experiments with an even higher number of simulations.

Now for the results on board 11x11, for time reasons, every experiment consists of only 50 game simulations. The results are the following:

Player 1	Player 2	Result	
MCTS 2s	Minimax d=2	2:48	26.9 moves
MCTS 6s	Minimax d=2	15:35	35.0 moves
MCTS 8s	Minimax d=2	14:36	33.7 moves

Even though the MCTS agent had the advantage of playing first, it still did not seem to have enough time to enough explore the best moves and scored worse than the Minimax agent. Again, we checked for the Minimax agent's average time to make a move and came up with 38 seconds, which is again way more than the maximum allowed time per MCTS move.

5.7 Heuristic AI vs. Random AI

To make sure that all the agents above play reasonably well, they were also tested against a randomly playing agent. This agent was mainly useful as a testing tool during programming and now will also serve as a safety check.

Starting with a game board 7x7.

Player 1	Player 2	Result	
Heuristic	Random	100:0	13.6 moves

As expected, the heuristic agent won 100% of the games. The average number of moves also got very close to the minimum number of moves a game needs to be over. In this case, since the random agent starts, it is $2 \cdot \text{board\_width} \cdot \text{board\_height} - 1$, which in our current experiment is equal to 13.

Next, the 11x11 board:

Player 1	Player 2	Result	
Heuristic	Random	100:0	21.9 moves

Similarly as on board 7x7, the heuristic agent won everything, and the average number of moves also got very close to the minimum possible, which in this case is 21.

5.8 Random AI vs. Heuristic AI

To keep things fair, we also swapped the order of players, letting the random agent get a chance to play first.

First, the 7x7 results:

Player 1	Player 2	Result	
Random	Heuristic	0:100	14.6 moves

Again, a full victory for the heuristic player. Since the heuristic agent plays second, the minimum number of moves for a valid game is $2 \cdot \text{board\_width} \cdot \text{board\_height}$. So for a game board of 7x7, it is equal to 14. Which is again very close to the average number of moves in the experiment.

Player 1	Player 2	Result	
Random	Heuristic	0:100	22.9 moves

Same as all the other cases for this pair of agents, 100% victory for the heuristic agent playing on average with nearly the minimum number of moves possible.

5.9 MCTS AI vs. Random AI

Next up is a test for the Monte Carlo Tree Search agent, starting on game board 7x7.

Player 1	Player 2	Result	
MCTS 0.5s	Random	100:0	20.5 moves

Since the MCTS agent has already won all the games, even with a very limited allowed time per move, no other experiment was conducted. However, unlike the heuristic agent, the MCTS agent needed, on average, three more rounds of the game. Where round is when both players play their move.

Results for game board 11x11:

Player 1	Player 2	Result	
MCTS 0.5s	Random	99:1	48.0 moves

Here, the random agent managed to get the better of the MCTS agent once, but still, the results are very favorable for the MCTS agent. And again, the average number of moves is much higher than the minimum possible, needing almost 13 more rounds on average.

5.10 Random AI vs. MCTS AI

For the sake of completeness, let us swap the order of the players. Starting with 7x7:

Player 1	Player 2	Result	
Random	MCTS 0.5s	0:100	21.4 moves

All the games were again won by the MCTS agent, but the average number of moves increased even more.

For the game board 11x11:

Player 1	Player 2	Result	
Random	MCTS 0.5s	1:99	87.9 moves

Here, almost 33 more rounds on average were needed to complete the game than the possible 11. Which is quite a significant difference but surely easily fixable by allowing the MCTS agent more time per move.

5.11 Minimax AI vs. Random AI

The last agent to test is the Minimax agent. The results for game board 7x7 are the following:

Player 1	Player 2	Result	
Minimax d=2	Random	100:0	13.6 moves

Similar to the results for the heuristic agent in both win rate and the average number of moves per game.

Next, results for board 11x11:

Player 1	Player 2	Result	
Minimax d=2	Random	100:0	23.8 moves

Interestingly, the Minimax agent with depth equal to two needed on average one more round to finish the game compared to the heuristic agent.

5.12 Random AI vs. Minimax AI

The last set of experiments, giving the Minimax agent the disadvantage of playing second. Starting with a game board of size 7x7:

Player 1	Player 2	Result	
Random	Minimax d=2	0:100	15.0 moves

Again, a clear win for the Minimax agent, needing on average close to the minimum number of moves.

And for the last experiment on a game board of size 11x11:

Player 1	Player 2	Result	
Random	Minimax d=2	0:100	24.0 moves

The Minimax agent again won 100% of the games and, on average, needed only one more round than the possible minimum.

Conclusion

In this work, we introduced the Hex game and analyzed it from a game theory point of view, also suggesting some simple strategies on how to play it. We have also described some popular AI algorithms suitable for Hex and similar games, such as the Minimax algorithm or the Monte Carlo Tree Search algorithm. Later, we explored already-existing Hex AIs and showed how popular and profusely researched this field is. However, these AIs, even though very good, are a little too good or too slow for a friendly amateur real-time game.

After the theoretical preparation, we implemented the game and the user interface for it. We also implemented four AI agents and experimentally tested them against each other. The agents were the following: the Heuristic agent using heuristics, the Minimax agent using the Minimax algorithm with alpha-beta pruning, the MCTS agent using the Monte Carlo Tree Search algorithm, and finally, the Random agent choosing its moves randomly.

From the experiments, we concluded that, unsurprisingly, the worst agent is the random one, the second worst is the heuristic one, as it uses the same evaluation function as the Minimax agent but does not search as deep, and the two best ones are the Minimax agent and the MCTS agent. Between these two, it really depended on the order of the players, as playing first has proven to be a big advantage. The experiments also showed that the Minimax agent with the depth of search of at least two is not suitable for game boards bigger than 11x11, as already on this board size it takes too much time to make a move, making the MCTS and the heuristic agent better choices.

To choose the difficulty for a game, one simply chooses an AI agent to play against and sets the parameters accordingly. The more time or the more simulations we allow the MCTS agent to have, the harder the game. Similarly, for the Minimax agent, the deeper we allow it to search, the harder the game.

5.13 Future Expansions

There are multiple ways this work can also be expanded in the future. The design allows for new AI agents to be simply added, or possibly combined with the already-implemented ones. There are also some algorithmic improvements that can be made to the existing AIs, like adding transposition tables to the Minimax agent or integrating neural networks into the MCTS agent.

It would also make for a nice addition to run more experiments to determine the best-performing agents more accurately. In particular, with a higher number of simulations for the MCTS agent, as unlike an allowed time per move, they are not so dependent on the performance of the computer the experiments are being run on.

Because of the importance of the order of the players, something like a swap rule can be implemented—a rule that makes playing first and second evenly good.

Bibliography

Vadim Anshelevich. The game of hex: An automatic theorem proving approach to game programming. 04 2001.

Broderick Arneson, Ryan Hayward, and Philip Henderson. Monte carlo tree search in hex. *Computational Intelligence and AI in Games, IEEE Transactions on*, 2:251 – 258, 01 2011. doi: 10.1109/TCIAIG.2010.2067212.

Cameron Browne. *Hex strategy - making the right connections.* 01 2000. ISBN 978-1-56881-117-8.

Guillaume Chaslot, Sander Bakkes, Istvan Szita, and Pieter Spronck. Monte-carlo tree search: A new framework for game ai. 01 2008.

Rémi Coulom. CLOP: Confident local optimization for noisy black-box parameter tuning. 11 2011. ISBN 978-3-642-31865-8. doi: 10.1007/978-3-642-31866-5_13.

E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959. doi: 10.1007/BF01386390.

David Gale. The game of hex and the brouwer fixed-point theorem. *The American Mathematical Monthly*, 86(10):818–827, 1979. doi: 10.1080/00029890.1979.11994922.

Bernard A Galler and Michael J Fischer. An improved equivalence algorithm. *Communications of the ACM*, 5 1964. doi: 10.1145/364099.364331.

Chao Gao, Ryan Hayward, and Martin Müller. Move prediction using deep convolutional neural networks in hex. *IEEE Transactions on Games*, PP:1–1, 12 2017. doi: 10.1109/TG.2017.2785042.

Chao Gao, Martin Müller, and Ryan Hayward. Three-head neural network architecture for monte carlo tree search. pages 3762–3768, 07 2018. doi: 10.24963/ijcai.2018/523.

Philip Henderson. *Playing and solving the game of Hex.* PhD thesis, University of Alberta, 2010.

International Computer Games Association. Computer Olympiad, 2021. URL `https://icga.org/?page_id=1566`.

Richard E. Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence*, 27(1):97–109, 1985. ISSN 0004-3702. doi: https://doi.org/10.1016/0004-3702(85)90084-0. URL `https://www.sciencedirect.com/science/article/pii/0004370285900840`.

Robert Moss. Mcts steps, 2020. URL `https://commons.wikimedia.org/wiki/File:MCTS-steps.svg`. Licensed under the Creative Commons Attribution-Share Alike 4.0 International license. `https://creativecommons.org/licenses/by-sa/4.0/`.

Stefan Reisch. Hex ist pspace-vollständig. *Acta Informatica*, 15, 6 1981. doi: 10.1007/BF00288964.

Stuart J. Russell. *Artificial intelligence: a modern approach.* 3d edition. Upper Saddle River, NJ : Prentice Hall, 2010. ISBN 978-1-292-40113-3.

Claude E. Shannon. Computers and automata. *Proceedings of the IRE*, 41(10): 1234–1241, 1953. doi: 10.1109/JRPROC.1953.274273.

Sumit Shevtekar, Mugdha Malpe, and Mohammed Bhaila. Analysis of game tree search algorithms using minimax algorithm and alpha-beta pruning. *International Journal of Scientific Research in Computer Science, Engineering and Information Technology*, pages 328–333, 11 2022. doi: 10.32628/CSEIT1228644.

David Silver, Aja Huang, Christopher Maddison, Arthur Guez, Laurent Sifre, George Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature*, 529:484–489, 01 2016. doi: 10.1038/nature16961.

Kei Takada. *A Study on Learning Algorithms of Value and Policy Functions in Hex.* PhD thesis, Hokkaido University, 2019.

Kei Takada, Hiroyuki Iizuka, and Masahito Yamamoto. Reinforcement learning to create value and policy functions using minimax tree search in hex. *IEEE Transactions on Games*, 12(1):63–73, 2020. doi: 10.1109/TG.2019.2893343.

Guan-Wei Chen Shi-Yu Chen Xian-Dong Chiu et al. Tristan Cazenave, Yen-Chi Chen. Polygames: Improved zero learning. *International Computer Games Association Journal*, 42(4):244–256, 2020.

Maciej Świechowski and Tomasz Tajmajer. A practical solution to handling randomness and imperfect information in monte carlo tree search. pages 101–110, 09 2021. doi: 10.15439/2021F3.

List of Figures

A. Attachments

The attachment contains the following folders:

- **Source codes**
  All of the source codes implemented for this thesis and described in Chapter 4 can be found here. Including the implementation of both the game itself and the AI agents.
- **Experiment results**
  This folder contains all of the experiments conducted and summed up in Chapter 5. Each file is named in the following format:
  AI1_vs_AI2_ParametersAndTheirSetting_BoardSize.csv
- **Documentation**
  This folder serves as a user guide to the project and it contains the following files:
  - README.md
  - requirements.txt
  - Specification.md