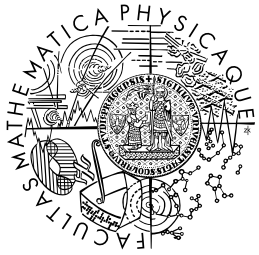


FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

BACHELOR THESIS

Avazagha Ahmadov

Tournament environment for the board game Hive with an implementation of sample bots

Department of Theoretical Computer Science and Mathematical Logic

Supervisor of the bachelor thesis: Mgr. Vladan Majerech, Dr.

Study programme: Computer Science with a specialization in Artificial Intelligence

Prague 2024

I declare that I carried out this bachelor thesis on my own, and only with the cited sources, literature and other professional sources. I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ....

Author's signature

First and foremost, I would like to express my deepest gratitude to my thesis supervisor, Mgr. Vladan Majerech, Dr., for his guidance and support throughout the research process.

I extend my heartfelt thanks to my family for their unwavering support and belief in my abilities, which has been a constant source of motivation.

I am also profoundly grateful to my closest friends for their exceptional assistance and support when I needed it the most.

Title: Tournament environment for the board game Hive with an implementation of sample bots

Author: Avazagha Ahmadov

Department: Department of Theoretical Computer Science and Mathematical Logic

Supervisor: Mgr. Vladan Majerech, Dr., Department of Theoretical Computer Science and Mathematical Logic

Abstract: This thesis addresses the performance of AI algorithms in the board game Hive. It implements and evaluates Alpha-Beta pruning and Monte Carlo Tree Search in this complex game setting. The study demonstrates that Alpha-Beta significantly outperforms MCTS, achieving perfect results against baselines. Additionally, the Alpha-Beta agent won against an intermediate-level human player and nearly won against an expert-level player. Compared to previous works, this research achieves better performance against human players.

Keywords: Monte Carlo Tree Search, Minimax, Alpha-Beta pruning, board game Hive

Contents

Introduction

Artificial intelligence (AI) encompasses a wide range of techniques designed to emulate human cognitive functions, with applications spanning from simple automated tasks to complex decision-making processes. At its core, AI aims to not only mimic human intelligence but also to enhance decision-making capabilities beyond human limits through the use of sophisticated algorithms and neural networks.

The study of artificial intelligence through board games offers a controlled yet complex environment for investigating algorithmic decision-making. This context allows for precise manipulation of variables and transparent measurement of outcomes, providing deep insights into the strategies and adaptability of AI algorithms. This thesis builds on this premise by implementing the board game Hive together with sample bots and tries to answer this question: How do different AI algorithms perform in a complex and unconventional game setting like Hive?

Hive is a two-player strategy board game that offers a unique challenge due to its lack of a fixed board. Similarly to chess, each piece follows a specific set of movement rules. However, unlike chess, the layout of Hive is dynamic and continuously changes with each move. Moreover, pieces in Hive cannot be removed once placed, leading to an increasingly complex game state as the game progresses. These features make predicting and analyzing potential moves and outcomes a demanding task.

To address this complexity, we focus on two prominent AI algorithms as strategies for the bots: Alpha-Beta pruning, known for its depth-first search optimization in zero-sum games, and Monte-Carlo Tree Search (MCTS), which uses randomized simulations to make decisions. Both algorithms have shown substantial success in structured game environments, such as chess and Go ([1] and [2]). However, Hive's unique gameplay introduces variable factors that challenge these algorithms differently, providing a fresh context for assessing their adaptability and strategic depth. In this thesis, the primary objective is to implement and evaluate the two AI strategies for the game of Hive. Specifically, we aim to implement AI agents using Monte Carlo Tree Search and Alpha-Beta pruning techniques, develop a software platform to facilitate the testing and evaluation of these AI agents, and conduct a comparative analysis of the performance of MCTS and Alpha-Beta pruning against baseline agents and human players.

In Chapter 1, we present background knowledge, the game rules, the winning conditions, and some terminology that will be used later in this discussion. Chapter 2 reviews the current state of research on the game. While acknowledging previous efforts to integrate artificial intelligence into Hive, we identify and propose potential refinements. In Chapters 3 and 4, we introduce the methodologies, specifically implementations of Alpha Beta pruning and Monte-Carlo Tree Search, respectively. In Chapter 5, we discuss how we developed the software to accommodate our needs. Chapter 6 evaluates these strategies against baseline Random agents and human players. Finally, in the Conclusion, we will highlight the effectiveness of these algorithms in Hive.

1 Hive

Hive is a bug-themed game designed by John Yianny in 2001 and published by Gen42 Games in 2015 [3]. Hive can be categorized as a two-player, zero-sum (one player's advantage is another player's loss), turn-based (each player waits for the opponent to finish their turn before starting theirs), and perfect information game (all players are fully informed of all previous moves and decisions made by other players).

In Section 1.1, we will demonstrate how to set up the game and describe movement types that a piece can have. In Section 1.2, we will introduce different types of pieces and the movement styles they use. Sections 1.3 and 1.4 introduce the general rules applied to all the pieces in the play. Section 1.5 will describe an objective of the game. In section 1.6, we will discuss why Hive is an interesting subject for this research.

1.1 Set Up for the Game and Movement

Although the game has no physical board, it can be imagined as being played on an infinite plane of interconnected hexagons. Each tile of this imaginary board represents a *stack* of pieces. From now on, *tile* and *stack* will be used interchangeably. *Stacks* can be of different sizes, if it does not contain any piece, we consider this stack to be empty and refer to it as an *empty tile*. If *stack* contains only one piece, we will refer to it as a *ground tile*. *Ground level* refers to a set of *ground tiles*. All pieces are assumed to be positioned on the *ground level* unless specified otherwise.

The movement of pieces in Hive is determined by the hexagonal shape of the tiles. Each tile must be placed so that one of its edges touches the edge of an adjacent tile. Moving a piece one "space" means shifting it to a different, imaginary hexagonal area adjacent to both its current position and another piece. This movement is called *crawling*, or *sliding*. When we mention a piece *crawling* for X amount of tiles, we mean there is a continuous sequence of length X of valid crawls. Each such *crawl* in the sequence can be referred to as a *step*.

Additionally, certain pieces can move onto the top of an adjacent higher *stack*, this movement is called a *climb up*. Once on top, a certain piece can move across the tops of adjacent *tiles*, adhering to the *sliding* definition. Such movement can be referred to as *sliding on top* of the Hive. The position of these pieces can be called *on top of the Hive*. A piece can descend from the higher *stack* to an adjacent lower one, or an *empty tile*. Such movement is referred to as a *climb down*.

Moreover, certain pieces can *leap*, or *jump*, over one or more *tiles* arranged in a straight line landing on the opposite side of the line. A straight line of *tiles* is a continuous sequence of adjacent *tiles* where each tile shares an edge with the next tile in the sequence, forming a direct path, meaning the *tiles* must form a single, uninterrupted line without changing direction. The *leap* must be in the direction of one of the piece's edges, not its corners.

Hive is a two-player game where each player is assigned a color (white or black) that matches the color of their tiles. From now on, we will refer to these players as player White and player Black. Each player receives 11 pieces in their assigned

color which they store in their *hand*. *Hand* refers to a set of pieces that have not been introduced to the board. Initially, this imaginary board is empty, and the game is started by player White, placing one of their pieces to the center of the board. This introduction of a piece from the *hand* to the board is referred to as *placing*, or *introducing*, a piece. A newly *introduced* piece must be adjacent to the rest of the Hive.

In this context, *ply* refers to a discrete unit of time during which a player can perform a move (*place* or *move* a piece), and a complete *turn* consists of both players completing their respective plies. On each *ply*, a player can either *place* a new piece or *move* a piece according to its specific abilities. When *placing* a new piece, it must only be adjacent to the player's pieces and cannot be adjacent to any of the opponent's pieces, with the only exception being the first ply, where player Black needs to place their piece next to the one of player White. This placing rule is also referred to as a **piece introduction rule**.

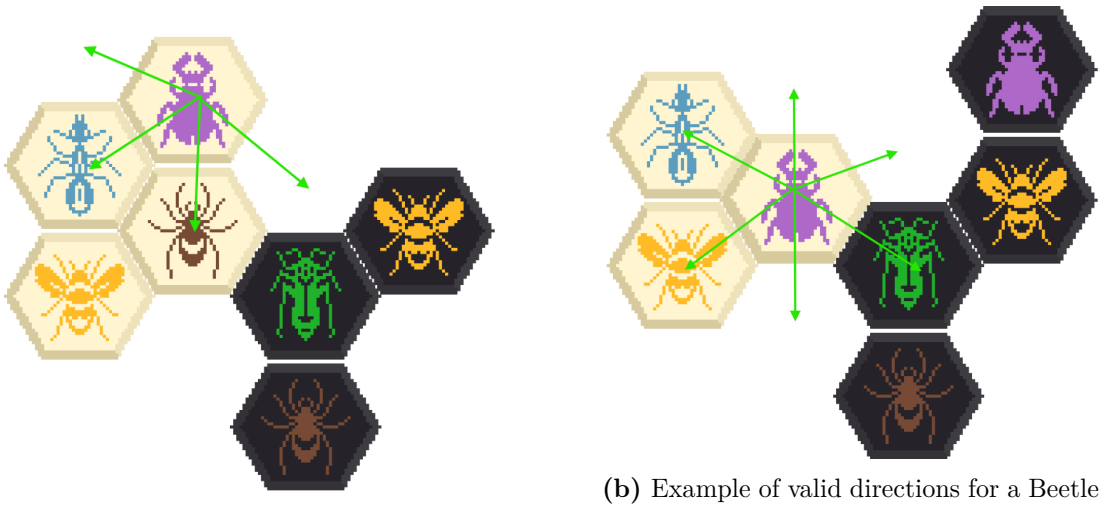
1.2 Bugs

Each piece in the game can also be referred to as a *bug*. Below, we will introduce *bug* types and their corresponding movement abilities. We assume that these moves are performed during one ply.

Queen Bee. Each player receives only one Queen Bee. This bug can only crawl one tile per ply. Figure 1.1 demonstrates this. Despite its limited range, Queen Bee is the most important piece in the Hive. A player is allowed to move their pieces if and only if their Queen Bee is introduced to the board. Queen Bee has to be introduced within the first four turns except for the first turn. The objective of the game (section 1.5) also demonstrates to us how important it is to protect the Queen Bee.

Figure 1.1 Example for valid Queen Moves

Beetle. Each player receives two Beetles. The Beetle, similarly to Queen Bee, slides only one space per ply. Figure 1.2a depicts this. Additionally, it can climb up, crawl on top of the Hive, and climb down. Figure 1.2b demonstrates additional movement abilities. All the pieces below the Beetle can not be moved and the color of the tile becomes the color of the top Beetle. Beetles can be stacked up on top of each other, meaning they are a part of the same stack. When it is first placed, Beetle is introduced in the same way as other pieces, meaning we can not directly place it on top of the Hive.



(a) Example of valid directions for a Beetle on the ground.

(b) Example of valid directions for a Beetle on the top of the hive. The white spider is under the Beetle in the stack.

Figure 1.2 Beetle movement

Grasshopper. Each player receives three Grasshoppers. The Grasshopper cannot crawl around like the other pieces in the Hive. Its main mechanic is to jump over at least one piece and land into the first unoccupied space along a straight line of tiles. Figure 1.3 shows valid destinations for a Grasshopper.

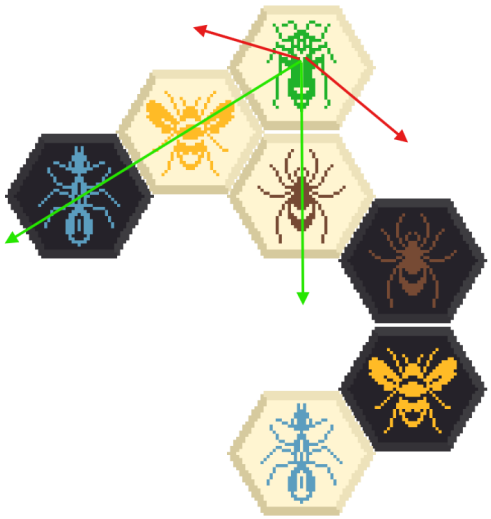


Figure 1.3 Grasshopper valid movement example.

Spider. Each player receives two Spiders. The Spider crawls exactly 3 tiles in one ply. It is not allowed to retrace its steps. Meaning, in the same continuous sequence of tiles, it is not allowed to repeat the tiles. It can only travel around pieces that it is adjacent to during each step, and it cannot move to a piece with which it is not directly adjacent. Figure 1.4 demonstrates this.

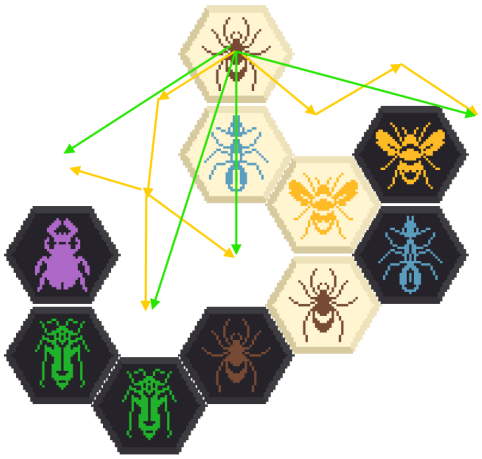


Figure 1.4 Example of valid Spider movement destinations. Yellow arrows represent the steps a Spider takes to move to these valid destination tiles, while green lines represent the valid destinations.

Soldier Ant. Each player receives three Soldier Ants. The Soldier Ant can crawl to any position around the Hive as long as the restriction rules are followed (see Sections 1.3 and 1.4), making it arguably one of the strongest pieces in the game for its ranged mobility. Soldier Ants are a stronger version of Spiders, they follow the same movement pattern, but Soldier Ants are allowed to crawl for any amount of tiles.

1.3 One Hive Rule

A piece cannot be moved in such a way that it results in two or more separate groups of pieces, either during or after its movement. A group refers to a collection of pieces that are adjacent to each other in such a way that each piece is directly adjacent to at least one other piece in that group. If a piece's movement temporarily disconnects the hive (meaning two or more groups can be observed), even if it reconnects afterward, the move is considered to be illegal. This rule supports a strategy where a player can *pin* an opponent's piece by positioning their piece in such a way that moving the opponent's piece would break the hive, immobilizing it. Figure 1.5 depicts the rule, we can also observe how both player White and player Black pinned their own Queen Bees via Soldier Ants.

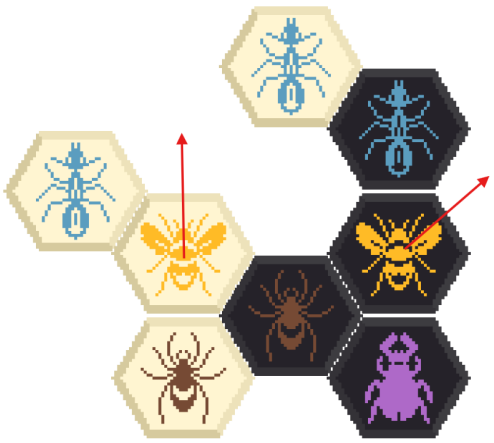


Figure 1.5 Example of moves breaking the One Hive Rule. Moving the white Queen Bee will temporarily break the Hive in two. Moving the black Queen Bee in any direction will cause the same.

1.4 Freedom To Move

This rule restricts pieces from moving through a narrow gap between two tiles, where such a gap is defined by configuration where the corners of the two tiles directly face each other, creating a constricted passage. In Figure 1.6b, we can see how the corners of the tiles that contain white Soldier Ant and black Queen Bee are facing each other, creating a narrow gap that does not allow the player Black's Ant to slide through.

A piece that is surrounded in such a way that it cannot slide out of its position must not be moved. Figure 1.6a demonstrates how the gap between player Black's Soldier Ant and a Queen Bee makes player White's Soldier Ant trapped and now the trapped Soldier Ant does not have another way out. The exceptions are the Grasshoppers, which are capable of jumping into or out of space, and the Beetles, which can climb up or down. Likewise, no piece should be moved into a space that it cannot slide into. Figure 1.6b shows this. However, when introduced to the game, a piece may be placed in a surrounded space, provided it is not adjacent to an enemy piece.

(a) In this case, Ant is trapped and can not move anywhere.

(b) Here, Ant can not follow the red arrow, but it can crawl anywhere else on the grid.

Figure 1.6 Examples of invalid moves due to Freedom to move rule.

1.5 Objective of The Game

Objective of the game is to surround the enemy's Queen Bee while also trying to prevent an opponent from surrounding our Queen Bee. The color of the surrounding pieces does not matter. The first player to surround their opponent's Queen Bee wins. Figure 1.7a demonstrates a case where player White wins. If both queens get surrounded simultaneously, the game ends in a draw. Figure1.7b shows the example of a draw. Although not specified in the official rules, we have imposed a cap of 100 turns per game as an additional constraint for our study. If the game reaches 100 turns, it is automatically considered a draw.

(a) Black's Queen Bee is surrounded. White Wins.

(b) Both Queen Bees are surrounded. It's a Draw.

Figure 1.7 Outcomes when Queen Bees are surrounded.

1.6 Why Hive?

In this section, we explore challenges presented by Hive by comparing it to Chess. Chess is typically characterized by an average branching factor of approximately 35 [4], whereas Hive exhibits a higher branching factor of around 60 [5]. The branching factor refers to an average number of valid moves per ply. Unlike Chess, where the capture of a piece allows a player to permanently remove a threat from the board, simplifying the game state, Hive introduces a different mechanic – pinning. This tactic leverages the One Hive Rule to position pieces to immobilize an opponent's piece. Pinning does not eliminate a piece but restricts its mobility, allowing a player to neutralize a threat temporarily. Nevertheless, the pinned piece remains on the board, posing a latent threat that can re-emerge if the pin is released. Consequently, the board only expands and becomes increasingly complex as the game progresses. Additionally, the lack of a fixed board representation presents an interesting challenge from a developer's perspective: how does one effectively encode the game state for algorithms? Furthermore, some pieces, like Beetles, can stack on top of other pieces. This introduces another dimension for the board to be worried about and somehow account for. These unique game mechanics in Hive require different strategic thinking and affect how well traditional algorithms like Alpha-Beta pruning and Monte Carlo Tree Search (MCTS) perform. Because of these complexities, Hive offers a valuable opportunity for expanding research in game theory and artificial

intelligence. We believe that studying Hive is not only worthwhile but could also inspire new approaches in algorithmic strategies.

2 Related Work

We are not the first to introduce Artificial Intelligence (AI) into Hive. A unique approach was performed by Connor Michael McGuile, where he utilized Swarm AI as an agent strategy for the game [6]. However, there are several issues with the methodologies used in their evaluations. To simplify the experiment, McGuile focused predominantly on the endgame by initiating gameplay with a board set up through 20 random moves before allowing AI agents to proceed. This method of random initialization, while reducing complexity, also omits critical strategic elements introduced during the opening phases of the game. Typically, the pieces placed during the first four moves remain static, thus immobilizing them for the game's duration. Consequently, starting with certain pieces, such as Soldier Ants, can substantially disadvantage a player later in the match.

Another quite interesting approach was using reinforcement learning in the report by Rikard Blixt and Anders Ye [7]. They implemented a pair of simple agents with only a knowledge of the game rules and no tactics and had them play against each other as a learning mechanism. Their experiment encountered some complications. The learning process proved to be computationally intensive and time-consuming. It required approximately 22 hours for the agents to complete 300 matches, a relatively small number for acquiring proficiency in a complex game like Hive. Moreover, Blixt and Ye estimated that, given the computational capabilities of a high-end personal computer at the time, it could take between 140 to 1400 years for an AI agent to reach a competent level through this training method.

Barbara Ulrike Konz implemented agents using different configurations of Monte Carlo Tree Search (MCTS), specifically a basic MCTS and MCTS enhanced with Upper Confidence Bounds applied to Trees (UCT) [8]. MCTS is a heuristic search algorithm for decision processes. At the same time, UCT is a specific strategy within MCTS that balances the exploration of unexplored moves with the exploitation of known rewarding moves, using the Upper Confidence Bound (UCB) formula. Both agents outperformed a baseline agent that executed only random moves, achieving winning rates of approximately 77% and 83%, respectively. When comparing the two methods directly, the well-tuned UCT agent outperformed the basic MCTS agent, with an average win rate of 45% compared to only 18% achieved by MCTS. Despite these successes, both strategies exhibited limitations when competing against human players. In addition, Konz's implementations encountered challenges related to time management. Each turn took up to 36 seconds on average, rendering the process computationally expensive and resulting in fewer MCTS simulations per move.

Tamás Bunth implemented a deep reinforcement learning framework inspired by AlphaZero to train artificial intelligence to play Hive [9]. His approach integrated Monte-Carlo Tree Search (MCTS) with a continuously improving neural network, enabling the agent to learn and adapt through selfplay. This method allowed the agent to achieve a 71% winning rate against a random agent, with the remaining outcomes being either draws or losses. However, a significant limitation of Bunth's methodology was its exclusive focus on AI versus AI matches, omitting human players. While the agent performed well against a random agent, this does

not necessarily translate to equivalent or superior performance against human opponents.

A recent study conducted by Danilo de Goede, Duncan Kampert, and Ana Lucia Varbanescu explores the complexity and computational costs associated with training a reinforcement learning agent for Hive [10]. Their research adopts the AlphaZero approach, where the agent improves through self-play and significantly outperforms a random agent. The study employed the Elo rating system to assess agents' performance. The study did not include the agents competing against human opponents. The Elo rating system is a method for calculating the relative skill levels of players in two-player games. Minimax was the best-performing agent, with an Elo rating of 1355, outperforming the next best agent by 292 Elo points, which was the self-play agent achieving an Elo rating of 1063 after only 24 hours of training. The authors measured the playing speed, exploration costs, and energy consumption per game instance. Their findings suggest that identifying the optimal AlphaZero agent configuration for Hive would require an extended training period, potentially spanning tens of node-years, where a node-year refers to the computational effort expended by one computational unit running continuously for one year.

In 2021, Duncan Kamper, Ana-Lucia Varbanescu, Matthias Müller Brockhausen, and Aske Plaat [5] implemented both AlphaBeta pruning and Monte-Carlo tree search (MCTS) as strategies for AI agents. They tested these agents against a Random agent and each other, comparing the winning rates and employing an Elo system to show the difference in strength between the bots. Moreover, the agents played against a human but showed poor results. One area for potential improvement is their board evaluation function, which relies on straightforward strategies, such as counting the pieces surrounding each queen. We believe we could refine this approach and will present our implementation of some strategies later in the discussion.

3 Monte-Carlo Tree Search

In this chapter, we will introduce the Monte-Carlo Tree Search (MCTS) algorithm and provide a detailed explanation of its procedural steps (selection, expansion, simulation, backpropagation). Rémi Coulom coined the Monte-Carlo Tree search in his 2006 work [11]. Additionally, we will discuss the types of games that are most suitable for applying this algorithm.

In Section 3.1, we will introduce the background knowledge necessary to understand the concept of MCTS in the context of our research. In Section 3.2, we will briefly overview the MCTS algorithm, explain briefly what happens during an algorithm iteration, and provide details on why Hive is a suitable game to test the effectiveness of Monte-Carlo Tree Search. Sections from 3.3 to 3.6 will explain each step of the iteration of the algorithm. In Section 3.7, we will demonstrate how the algorithm chooses the best move, and in Section 3.8, we will introduce a technique that will help us to speed up the algorithm. Section 3.9 provides a pseudocode of the algorithm.

3.1 Background

A *game state* is defined as a specific snapshot that captures all the essential information about the game, such as piece positions, whose ply it is to play, and the history of moves that have occurred, which collectively defines the status of a game at any particular point in its progression. *End game state*, or *end state* (node), refers to the game state of the finished game (when the game ends in a win, draw, or loss). A *game state tree* is a tree data structure where each node represents a game state, and each edge (a direct connection from a node "A" to a node "B") represents a move taken by players, such as moving or placing a piece, that transitions from game state "A" to game state "B". In this case, the node "B" is a child of the node "A" and "A" is a parent of "B".

Term $Children(node)$, or $N(node)$ refers to a list of children (neighbors) of the node. The game state tree can be seen as a directed tree graph, where nodes store the game state, the number of wins observed in children, the number of visits by the traversal, and the parent of the node. The edges store the moves that transition the game states. The number of wins and visits in the nodes will be defined later on during the tree traversal. From now on, the terms game state and node, as well as move and edge, will be used interchangeably.

The *root* of the game state tree is an initial game state, where all the pieces are in hand, and it is a player White's ply. We say that root has no parent. The leaf of the game state tree is a node with no children. Moreover, a *path* between nodes "A" and "B" in the game state tree is a sequence of edges and nodes that leads from node "A" to node "B". *Tree traversal* refers to a process of visiting each node in a tree data structure. One specific type of tree traversal is Monte-Carlo Tree Search (MCTS).

In Monte-Carlo Tree Search, the traversal of the game state tree begins from the root and progresses towards the leaves. The algorithm faces multiple options (edges) at each node, each leading to a new potential game state. Selection constructs a path from the root to the leaf, where this leaf is referred to as the

selected leaf. The choice of which edge to follow is determined by a selection policy.

A *policy* in MCTS is a method of selecting the most promising edge. The most promising edge refers to a move that is expected to lead to better outcomes (nodes with higher value) based on previous exploration. This policy often involves calculating a value that balances the potential benefits of exploitation (favoring paths that have historically led to wins) and exploration (testing less promising paths).

One such policy is the Upper Confidence Bound applied to Trees (UCT), which uses statistical measures, such as the number of visits and the winning rate of the node, to evaluate the potential of each move. Moves with higher potential are promising and more likely to be chosen for further traversal. Upon reaching the leaves, they are expanded by adding new children, which then become the new leaves, while their parent is called an expanded node.

MCTS then performs a *rollout*, or a *playout*, on each of the children. A rollout refers to a process of starting from a given game state and playing the game stochastically, following a probability distribution, until reaching the end state. The result of the rollout is a number between 0 and 1 that directly corresponds to the outcome of the end state (win, draw, or loss). The result of the playout is assigned to that child's number of wins. Subsequently, the number of wins of each non-leaf node is updated based on the number of wins observed from its descendants. The visit count of each node on the path is also incremented. This traversal is repeated until a sufficient number of iterations have been performed. The number of wins and visits does not reset by the end of the iteration.

3.2 Overview of MCTS

According to Chaslot(2010) [12], Monte-Carlo Tree Search is a best-first search algorithm (a type of search algorithm, which during each step of the traversal chooses the best edge), where it primarily employs Monte Carlo simulations (rollout) to gather value estimates that direct the search towards the most promising paths in the game state tree. Essentially, MCTS focuses on the more promising nodes (game states that historically lead to wins), thereby bypassing the need to explore every possibility, which would be impractical exhaustively.

Chaslot (2010) [12] describes that an iteration of Monte-Carlo Tree Search consists of four phases: selection, expansion, simulation, and backpropagation. Initially, the selection phase involves navigation from the root node to a leaf node and selecting that leaf node for expansion.

Subsequently, in the expansion phase, the tree grows by adding new children to the selected leaf. We add new children and create edges (possible valid moves) between the selected node and its new children. Now, that previously selected leaf becomes an expanded node, and its children are leaves.

The third phase, simulation, entails playing out the game from the children of the expanded node to an end state. This is called a playout, or rollout. The result of the playout gets assigned to the win count of that child. Finally, the backpropagation phase occurs, where the simulation results are propagated back from the leaves to the root. During this propagation, MCTS updates each node's visit and win count along the path formed by the selection. The exact method

of updates will be discussed in Section 3.6. After running the algorithm for a sufficient number of iterations (at least 100), we get a game state tree. To retrieve a move for the AI to take. We choose the best child of the root, where we define the best child in Section 3.7. Figure 3.1 depicts visual outline of the algorithm by Chaslot(2010).

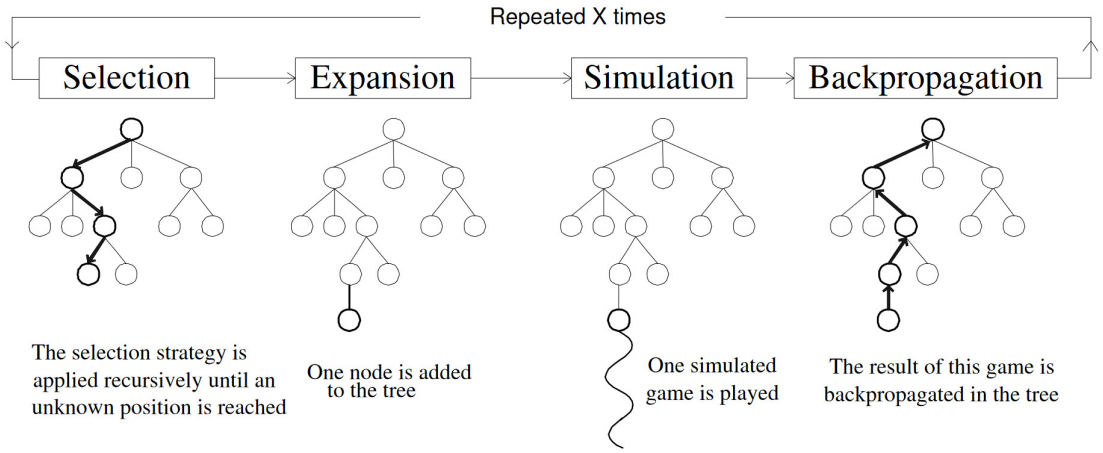


Figure 3.1 Outline of Monte-Carlo Tree Search [12]

To effectively deploy Monte Carlo Tree Search (MCTS) as an agent within a game, several criteria must be satisfied [12]. The game must feature bounded payoffs, which means that the outcomes of the game are constrained within a specific range. It should also have perfect information, where all elements of the game state and moves are known to all players. Additionally, there should be a predefined maximum length, limiting the number of turns. Hive serves as an exemplary model of a game meeting these conditions. We can bound the results of the end state (win and loss) to 1 and 0, respectively, with a draw assigned a value of 0.5. According to the game's rules, both players possess comprehensive knowledge of the pieces on the board and in hand, meaning that Hive features perfect information. Although the rules do not explicitly define the game's maximum number of turns, we pragmatically cap it at 100 for analytical convenience.

3.3 Selection

During selection, we will recursively traverse from the root to the most promising leaf by choosing the most promising edge at each step of the traversal. Upon reaching the leaf, it is referred to as a selected leaf, and the path containing all the nodes and edges from the root to the leaf is called a selection path. The most promising edge is the edge that leads to the node with a maximal policy value. An important aspect of such policies is the balance between exploration and exploitation. On one side, exploitation involves selecting edges that appear most promising, thereby optimizing the immediate outcome. On the other side, exploration is crucial for investigating various alternatives, including those that might initially seem less favorable, to ensure that superior solutions are not inadvertently disregarded. Although various policies exist, this discussion focuses on the Upper Confidence bounds for Trees (UCT) as the policy of our choice. Kocsis

and Szepesvári introduced it in their 2006 work [13]. The formula for *UCT* of i-th child of a node p in the tree can be defined as follows:

$$\mathrm{UCT}(i) = V_i + C\sqrt{\frac{\ln N_p}{N_i}} \tag{3.1}$$

where

- V_i is the winning rate of the node i . It is calculated by dividing the number of wins by the number of visits of the node i,
- C is the exploration parameter, which controls the degree of exploration (generally, this is a non-negative number, where sufficient range depends on the game),
- N_p is the number of visits of the node p,
- N_i is the number of visits of the node i.

During each step of selection, we choose an edge that leads to child i such that

$$i = \arg \max_{i \in \mathrm{Children}(p)} \mathrm{UCT}(i)$$

where arg max refers to a function that is used to find the argument (input) that gives the maximum value of a given function. In our case, the given function is $\mathrm{UCT}(i)$.

3.4 Expansion

Expansion is the most straightforward part of the algorithm. We add new nodes to the MCTS tree. In certain games, such as Hive and Chess, storing the entirety of the game tree is unfeasible because of the high branching factor. Therefore, we decide how the selected leaf should be expanded with new children during expansion. For every valid move at the selected leaf state, we add an edge and the resulting game state that this edge leads to as children of the selected node. This selected node is now called an expanded node.

3.5 Simulation

Simulation is a process that starts from the children of an expanded node and performs a playout of the game until we reach the end state by playing moves stochastically. A knowledge-based heuristic is frequently used to select the moves during a playout. However, for simplicity, we will focus on selecting moves randomly (each valid move has an equal probability of being chosen). The result of a playout is a number that indicates the outcome of the end state: a winning end state results in a score of 1, a losing end state in 0, and a draw in 0.5, as mentioned in Section 3.2

3.6 Backpropagation

In the backpropagation phase of Monte Carlo Tree Search, the results from playouts are propagated from the leaf nodes back to the root. During this process, the statistical values of each node, specifically the visit count and the node value, are updated. Starting from the leaf, we increment the visit count for each node in the selection path and add the result from the playout to each node's number of wins count.

3.7 Choosing The Best Move

After the completion of the algorithm, resulting in a game state tree, the process of selecting the most promising move (move that will lead to an advantageous position for the agent) becomes straightforward. This selection involves choosing the best child of the root node. Several strategies can be employed to determine this best child. According to Chaslot (2010) [12], there are various methods such as the Max Child (selecting the node with the highest win count), Robust Child (node with the highest visit count), and Robust-Max Child (node scoring highest in both win and visit count). Chaslot notes that these strategies have no significant performance difference when decisions are not time-constrained. In our discussion, we will focus only on using Max Child.

3.8 Parallelization

Parallelization is a computational technique that divides tasks into smaller, often simpler, parts that can be processed independently and simultaneously rather than sequentially. As Chaslot (2010) [12] denotes, three types of parallelization can be applied to MCTS: leaf parallelization, root parallelization, and tree parallelization. Leaf parallelization involves performing multiple simulations and backpropagations concurrently. These are independent of each other for each child of the expanded node, which makes this method easy to implement and very effective. Root parallelization involves multiple instances of the algorithm starting with the same root but following different paths through the tree. After the algorithms are done, the results from all trees are aggregated to make a final decision (now, we choose the best child of the root among multiple trees). Tree parallelization refers to a parallelization of the iterations of the algorithm. Here, we have only one instance of the algorithm, but different parts of the tree are processed in parallel, with each processor focusing on a different subtree. In our implementation, we will focus mainly on leaf parallelization.

3.9 Algorithm

In this Section, we present the pseudocode demonstrating how the Monte-Carlo Tree Search operates. The ideas behind the functions *Select*, *Expand*, *Simulate*, and *Backpropagate* are explained in the Sections from 3.3 to 3.6, respectively.

The selection of the final move is demonstrated by the *BestChild* function, which returns a *node*, where

$$node = \arg \max_{child \in N(root)} (child.visitCount)$$

MaxIterations is the total number of iterations for the algorithm to perform. It is set in advance. Generally, at least 100 iterations are sufficient. Additionally, the function *ValidMoves* receives game state as input and outputs a collection of all the valid moves a player can have at the given game state. The function *AddNewChild* receives a game state and move as input and outputs a new game state that is a result of playing the input move on the input game state.

PlayGame function refers to the playouts we mentioned in Section 3.5, it performs stochastic play and updates node state accordingly. Subsequently, the function *Result* takes the end state and returns a number between 0 and 1, indicating the outcome of the end state.

Algorithm 1 Monte Carlo Tree Search

```
1: procedure MCTS(root)
2:     node ← root
3:     iteration ← 0
4:     while iteration ≤MaxIterations do
5:         node ← SELECT(node)
6:         if node.State is not EndState then
7:             node ← EXPAND(node)
8:         end if
9:         reward ← SIMULATE(node)
10:        BACKPROPAGATE(node, reward)
11:    end while
12:    return BESTCHILD(root)
13: end procedure

14: function SELECT(node)
15:    while node is not a leaf do
16:        node ← arg max_{child∈N(node)} UCT(child)
17:    end while
18:    return node
19: end function

20: function EXPAND(node)
21:    for move in VALIDMOVES(node.State) do
22:        newNode ← ADDNEWCHILD(node, move)
23:    end for
24:    return newNode
25: end function

26: function SIMULATE(node)
27:    while node.State is not EndState do
28:        PlayGame(node)
29:    end while
30:    return RESULT(node)
31: end function

32: procedure BACKPROPAGATE(node, reward)
33:    while node is not null do
34:        node.visitCount ← node.visitCount + 1
35:        node.winCount ← node.winCount + reward
36:        node ← node.parent
37:    end while
38: end procedure
```

4 Minimax and Alpha-Beta pruning

In this chapter, we will introduce the Minimax algorithm and a technique called Alpha-beta pruning applied to the Minimax. We will explain how the algorithm works and how we adopted it for Hive. The Minimax algorithm is derived initially from von Neumann's Minimax theorem [14]. Whereas Alpha-Beta pruning was discovered independently by multiple researchers around the 1960s. Still, the refined version of the algorithm that is widely used today was introduced in the 1975 work of Donald Knuth and Ronald W. Moore [15].

In Section 4.1, we will provide the essential background knowledge needed to understand the concept of Minimax within the scope of our research. In Section 4.2, we will explain how Minimax works, including a pseudocode on Minimax implementation, and point out some of the disadvantages of using Minimax with no techniques applied. In Section 4.3, we will discuss the improvement of Minimax through Alpha Beta pruning, detailing how it works and providing a pseudocode demonstrating the implementation of Alpha Beta pruning applied to Minimax. In Section 4.4, we will define an evaluation function and introduce Hive-specific strategies implemented into an evaluation function used by the Minimax algorithm.

4.1 Background

Definitions that were introduced in the Section 3.1 remain applicable in this context and will be employed similarly, with the only exception of the node structure used in the game state trees. In the context of Monte-Carlo Tree Search, we said that nodes in the game state tree consist of the following information: game state, win count, visit count, and a parent. Herein, in the context of Minimax, nodes will consist only of a game state, value, and parent.

Additionally, we define the depth of a node in the game state tree to be the length of the path from that node to the root and the height of the node to be the length of the longest path from that node to a leaf. The depth of the game state tree is the maximum depth of any node in the game state tree. A level of the game state tree refers to a set of nodes with a specific depth in the game tree. The root node is at level 0, its direct children are at level 1, and subsequent descendants follow at increasing levels.

We also define a node's value by stating that a leaf's value is calculated by an evaluation function. An evaluation function is a method that estimates the favourability of the game state for a specific player. The value of a non-leaf node depends on the player's ply in the game state. If it is the maximizing player's ply, then the node's value is the maximum value of its children. If it is the minimizing player's ply, then the node's value is the minimum value of its children. A maximizing player refers to a player who aims to maximize the evaluation function. In other words, a maximizing player chooses the moves that increase their chance of winning. Conversely, a minimizing player aims to minimize the score of the evaluation function, essentially reducing the maximizing player's score as much as possible.

Generally, the evaluation function is designed to assess a specific player's score. In our context, the evaluation function returns the desirability of the game state only from the perspective of the maximizing player. This means that if our agent (the maximizing player) is playing as White, the evaluation function will return scores for the player White. Similarly, if our agent starts as Black, the evaluation function will return scores for the player Black. In our case, the root of the game state tree corresponds to a maximizer's node.

In this context, an agent refers to an algorithm that plays a game by making decisions based on a specific strategy. An agent that uses the Minimax algorithm as its strategy is called a Minimax agent, while an agent that uses the Alpha-Beta algorithm for its strategy is called an Alpha-Beta agent.

4.2 Minimax

The idea behind Minimax is rooted in zero-sum games, where one player's gain is another player's loss. Starting with the current game state at the root, the algorithm expands all the available leaves to all possible future game states, alternating between the two players until it reaches the end state or a predetermined depth of the game state tree. An expansion of a leaf refers to a process of adding new children for each valid move available at that leaf. We add the children by creating edges representing the available move that leads to the resulting game state (a newly added child). Consequently, children become the new leaves, and we repeat this process until reaching an end state or a predetermined depth of the game state tree.

In the Minimax algorithm for two-player games, the players are typically referred to as the maximizer and the minimizer. In our case, we say that the Minimax agent is the maximizer and the opponent is the minimizer. From now on, in this chapter, we will use the Minimax agent and the maximizing player (maximizer) interchangeably. The same goes for the Minimax agent's opponent and the minimizing player (minimizer). The maximizer tries to maximize their score, while the minimizer tries to minimize the maximizer's score. The score is obtained from the evaluation function. The algorithm recursively explores all possible moves, assuming both players play optimally.

When reaching an end state or a predetermined depth, Minimax evaluates each leaf node by assigning it a value obtained from the evaluation function, which takes the game state of the leaf as input. The details on implementing the evaluation function can be seen in Section 4.4. For now, we say that it returns an integer between $-\infty$ and ∞, which describes how desirable the game state is for the maximizer.

After assigning these values, the algorithm backpropagates them up the game state tree. Backpropagation in this context refers to propagating these evaluated values from the leaf nodes back up to the tree's root. At each level of the game state tree, the algorithm alternates between two players: maximizer and minimizer. If the current level represents the maximizer's ply, the algorithm seeks to maximize the value, selecting the highest value among the child nodes during the backpropagation. Conversely, if the current level represents the minimizer's ply, the algorithm seeks to minimize the value, selecting the lowest value among the child nodes. This alternation continues up the tree, ensuring that at each

node, the player considers the optimal moves of the opponent.

By the time backpropagation reaches the root, game state tree represents all possible sequences of moves and counter-moves up to a certain length (the predetermined depth), enabling the algorithm to determine the optimal initial move. This move corresponds to the edge leading to the root's child node with the highest value.

In Subsection 4.2.1, we will introduce the pseudocode for the Minimax algorithm. In Subsection 4.2.2, we will discuss why using the Minimax algorithm can have a significant disadvantage and how we can solve this issue.

4.2.1 Algorithm

Below, we present a pseudocode for the Minimax algorithm for a two-player zero-sum game. The function *IsEndNode* receives a node as an input and outputs a boolean value, whether the node is an end state or not. The function *Evaluate* represents the evaluation function, which takes a node as an input and returns an integer. The *maximizingPlayer* is a boolean input argument indicating for which player the calculation is being performed; **true** corresponds to the maximizing player and **false** corresponds to the minimizing player. Functions *max* and *min* return a maximum and a minimum of the two inputs, respectively. The function *GenerateChildren* takes a node as an input and returns a list of all possible children of the node.

To use the *Minimax* function, it is called based on the game's setup. For example, if we want our Minimax agent to play as the player Black and "think" three plies ahead, we would call the Minimax function for each available move with the arguments: *node* set to the resulting game state if we perform the available move, depth set to 3, and *maximizingPlayer* set to **true**. After gathering the results returned by the Minimax function for each valid move, we choose the move that has the highest Minimax value assigned to it. The Minimax value represents the optimal score that the maximizer can achieve, assuming both players play optimally from that point onward. By selecting the move with the highest Minimax value, the agent ensures that it is making the best possible move given the anticipated responses of the opponent.

Algorithm 2 Minimax Algorithm

```
1: function MINIMAX(node, depth, maximizingPlayer)
2:     if depth == 0 or ISENDNODE(node) then
3:         return EVALUATE(node)
4:     end if
5:     if maximizingPlayer then
6:         maxEval ← −∞
7:         for all child in GENERATECHILDREN(node) do
8:             eval ← MINIMAX(child, depth - 1, false)
9:             maxEval ← max(maxEval, eval)
10:        end for
11:        return maxEval
12:    else
13:        minEval ← ∞
14:        for all child in GENERATECHILDREN(node) do
15:            eval ← MINIMAX(child, depth - 1, true)
16:            minEval ← min(minEval, eval)
17:        end for
18:        return minEval
19:    end if
20: end function
```

4.2.2 Problems with Minimax

While the Minimax algorithm is effective in theory, it faces significant practical challenges, especially in games with a large branching factor like Hive or Chess. A branching factor refers to an average or constant number of valid moves available at each game state. The primary issue is the exponential growth of the game tree, which makes it computationally expensive to evaluate all possible moves. This leads to a time complexity of $O(b^d)$, where b is the branching factor and d is the depth of the game state tree. Figure 4.1 demonstrates an example of an arbitrary game state tree generated by the Minimax algorithm.

The computational burden becomes evident when considering a game like Hive with an average branching factor of 60. An optimization technique known as Alpha-Beta pruning can be applied to the Minimax algorithm to mitigate such computational costs.

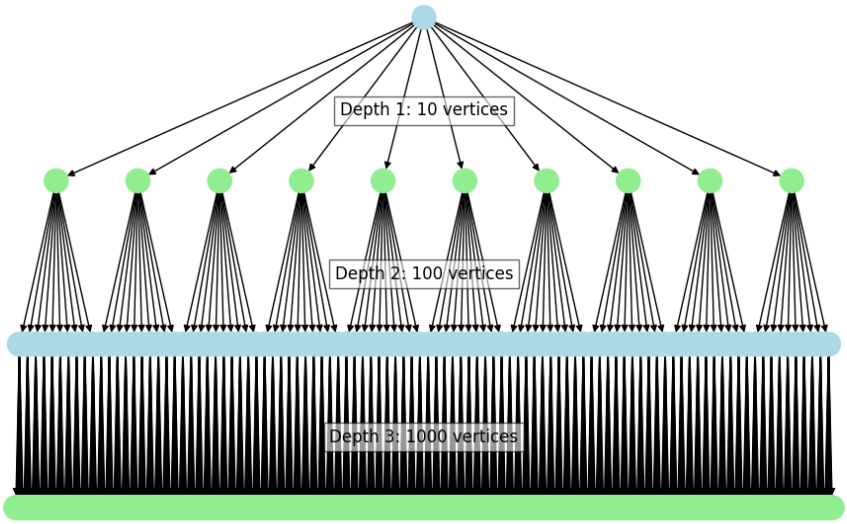


Figure 4.1 Illustration of a game state tree with a branching factor of ten and a depth of three. Blue nodes represent the maximizer's ply and green nodes represent the minimizer's ply.

4.3 Alpha-Beta pruning

Alpha-Beta pruning is a technique applied to the Minimax algorithm, where pruning refers to a process of eliminating branches in the game state tree. From now on, when referring to Alpha-Beta search, we mean Minimax with Alpha-Beta pruning technique applied. Alpha-Beta pruning works by keeping track of two special values, α and β.

The α value represents the minimum score that maximizing player is assured of at any point during the search. This means that, given the current state of the game state tree traversal, the maximizing player can guarantee a score of at least α, regardless of how the minimizing player plays from that point onwards. Initially, α is set to $-\infty$, representing the worst possible score for the maximizing player. Similarly, the β value is the maximum score that minimizing player is assured of at any point during the search. This means that, given the current state of the game state tree exploration, the minimizing player can guarantee a score of at most β, regardless of how the maximizing player plays from that point onwards. At the start of the algorithm, β is set to ∞, representing the worst possible score for the minimizing player.

In Alpha-Beta search, the values of α and β are updated during the search. For the maximizing player, α is increased to the value of a node if this node provides a higher score than the current α. Conversely, for the minimizing player, β is decreased to the node's value if this node offers a lower score than the current β.

Pruning occurs when β is less than or equal to α. In such cases, further exploration of the current branch is unnecessary. This is because the maximizing player has a better option elsewhere that is at least as good as the current α, and the minimizing player can avoid this branch since they can achieve a result no worse than the current β in another branch. This mechanism efficiently cuts

off irrelevant branches of the game tree, significantly reducing the number of nodes that need to be evaluated, which accelerates the decision-making process by focusing only on branches that could influence the final outcome.

In subsection 4.3.1, we will compare the Minimax algorithm with Alpha-Beta search and discuss how it solves the issue of the Minimax algorithm. In subsection 4.3.2, we will demonstrate other techniques that could improve the performance of Alpha-Beta search even more. In subsection 4.3.3, we will present the updated pseudocode that implements the Alpha-Beta pruning.

4.3.1 Comparison with Minimax

The Minimax algorithm evaluates every possible node in the game state tree up to a certain depth, leading to an exponential growth in the number of nodes as the depth increases. On the other hand, Alpha-Beta search reduces this number by effectively ignoring large portions of the tree. In the best-case scenario, when we prune at every level, Alpha-Beta can reduce the time complexity from $O(b^d)$ to $O(b^{d/2})$. This makes the search process more manageable and allows for deeper exploration within the same time constraints. In the worst-case scenario, however, the time complexity stays at $O(b^d)$ when we can not prune any of the branches, leaving us with the full game state tree.

This reduction in complexity is primarily due to effective move ordering in Alpha-Beta pruning. The algorithm can achieve a significant speedup by selecting the best move first. This happens because the earlier the best move is considered, the more influential the pruning process becomes, leading to fewer nodes being evaluated. Conversely, if the best move is selected late in the ordering, the pruning has minimal effect, resulting in almost no speedup. To achieve effective move ordering, we will use our heuristic. A heuristic function is a rule-of-thumb strategy designed to find an approximate solution when the exact solution is not necessary, in our case, the ranking of the moves. In our heuristic, we define the following importance ordering:

1. Adding a piece adjacent to the enemy Queen.
2. Moving a piece to a coordinate adjacent to the enemy Queen.
3. Adding a piece anywhere.
4. Moving a piece anywhere.

For the tiebreakers:

- When moving or adding pieces anywhere, the order of priority is Soldier Ant > Grasshopper or Beetle > Spider.
- When moving or adding pieces to be adjacent to the enemy Queen, the order of priority is Spider > Grasshopper or Beetle > Soldier Ant.

Using this heuristic, we sort the list of valid moves available for the Alpha-Beta agent and start the search.

4.3.2 Transposition Tables

Another technique that we could apply is a transposition table. A transposition refers to a game state or position that can be reached through different sequences of moves. Transposition tables store these game states, or their evaluations in our case. By identifying these repetitions, the search algorithm avoids redundant evaluations, improving efficiency.

To hash and store game states, we utilize *Zobrist hashing* [16]. In chess, a random number is assigned to each potential piece on each square at the beginning of the game. The hash of any game state is computed by combining these numbers using exclusive OR (XOR) operations based on the pieces present in the state. For Hive, we adapt this concept to account for the absence of a fixed board and the ability to stack pieces. We encode space for all possible piece positions using an axial hex coordinate system (Q, R), where each piece has coordinates q and r representing columns and rows. This system aligns with the hexagonal grid structure, allowing each hexagon to have six neighbors. Given 22 total pieces in the game, the hypothetical situation where pieces stretch from the center to one of the borders requires a coordinate shift. Instead of (-22,-22) to (22,22), we shift to (0,0) to (44,44). Additionally, the maximum stack size is five due to the presence of four Beetle pieces. Hence, we consider a layer of 44 x 44 coordinates for each piece position, with five layers for each stack height.

We need to initialize the table dimensions for the Zobrist hash table. An entry in the Zobrist hash table is defined by the axial hex coordinate of the piece, the piece itself, and the height of the piece (position in the stack). Each entry is assigned a random number during initialization. So the final dimensions of the Zobrist hash table are

$$44 \times 44 \times 22 \times 5$$

After initializing the Zobrist hash table, we can compute the hash for the current game state being observed by the agent. This hash serves as an entry in the transposition table, mapping hash values to game state evaluation values. If the agent already has an entry in the transposition table for the hash of the current game state, it will return the corresponding value. Otherwise, the agent will calculate the evaluation and add a new entry to the transposition table. By remembering already computed states, we save some computational time.

4.3.3 Algorithm

Below, we introduce a pseudocode for the Alpha-Beta search algorithm. The function definitions from the Section 4.2 are also applicable here. We also say that $transpositionTable$ refers to the transposition table introduced in subsection 4.3.2, implemented as a hash table, and $node.ZobristHash$ is the Zobrist Hash corresponding to that game state. We use α and β values for pruning.

Algorithm 3 Alpha-Beta Search

```
1: function AlphaBeta(node, depth, α, β, maximizingPlayer)
2:     if transpositionTable.ContainsKey(node.ZobristHash) then
3:         return transpositionTable[node.ZobristHash]
4:     end if
5:     if depth == 0 or IsEndNode(node) then
6:         eval ← Evaluate(node)
7:         transpositionTable[node.ZobristHash] ← eval
8:         return eval
9:     end if
10:    if maximizingPlayer then
11:        maxEval ← −∞
12:        for all child in GenerateChildren(node) do
13:            eval ← AlphaBeta(child, depth - 1, α, β, false)
14:            maxEval ← max(maxEval, eval)
15:            if β ≤ α then
16:                break                                  ▷ Beta cut-off
17:            end if
18:        end for
19:        return maxEval
20:    else
21:        minEval ← ∞
22:        for all child in GenerateChildren(node) do
23:            eval ← AlphaBeta(child, depth - 1, α, β, true)
24:            minEval ← min(minEval, eval)
25:            if β ≤ α then
26:                break                                  ▷ Alpha cut-off
27:            end if
28:        end for
29:        return minEval
30:    end if
31: end function
```

4.4 Evaluation Function

The efficiency of Alpha-Beta pruning is heavily dependent on the quality of the evaluation function used at the terminal nodes. A good evaluation function accurately estimates the desirability of a position, enabling more effective pruning. If the evaluation function is poor, it can lead to incorrect pruning decisions, thereby reducing the effectiveness of the algorithm.

The evaluation function, $e(s)$, is a function that takes a game state s as input and returns a numerical value representing the favourability of the board for the maximizing player. In this context, the "favorability of the board" refers to how advantageous the current game state is for the maximizing player compared to the minimizing player. The output range of the evaluation function is $[-\infty, \infty]$, where $-\infty$ represents a completely lost game state, ∞ represents a completely won game state, and 0 represents a draw.

The evaluation function calculates a difference between the *advantage* and *threat* for any other game state. The *advantage* represents the strength and potential of maximizer's pieces on the board, while the *threat* represents the potential and strength of the minimizer's pieces. Hence, the evaluation function can be expressed as:

$$e(s) = advantage - threat$$

In this context, a positive value of $e(s)$ indicates a more favorable position for the maximizing player, while a negative value indicates a more favorable position for the minimizing player.

The *advantage* is calculated as follows:

- For the immobility of the opponent's Queen, we add 30000 to the *advantage*.
- For each piece adjacent to the opponent's Queen, we add a bonus *advantage* score. The non-Ant pieces of the maximizer are adding 5500 to the advantage score, while maximizer's Soldier Ant and any of the immobile *minimizer's pieces* each add 5000 to the advantage.
- The general mobility of the maximizer's pieces on the board results in an additional advantage of $200 + PieceScore * 10$. The piece score is a numerical value assigned to each type of piece. We say that the piece scores of Queen Bee, Soldier Ant, Grasshopper, Beetle, and Spider are 100, 6, 4, 4, and 2, respectively.
- For each mobile Soldier Ant of the maximizer we add a bonus of 100.
- For each mobile and well-placed maximizer's Beetle, we add a bonus. If the Beetle is on top of the enemy Queen, the update is 100. Otherwise (on the ground or on top of the Hive), the *advantage* is scaled inversely by the hexagonal distance to the enemy Queen. Meaning that if the Beetle is too far from the enemy Queen, the value of it is trivial.

The *threat* is calculated similarly to the *advantage*, but with the piece possession flipped from the maximizing player to the minimizing player. Everything advantageous for the minimizing player is a *threat* for the maximizing player. This thorough evaluation ensures that all potential threats are considered, enhancing the algorithm's effectiveness.

5 Implementation

In this chapter, we present the implementation details and the development process of the software designed to achieve the objectives outlined in the Introduction. Our primary focus is on the game logic and AI integration. The source code includes comments that thoroughly explain each method used. It is important to note that this chapter does not cover the technical details of the graphical user interface (GUI). Instructions for users regarding the GUI can be found in attachments (attachment A, User Documentation). At the same time, developers can refer to the in-code comments for detailed descriptions of each method.

The chosen programming language is C# due to our familiarity with the .NET Framework. For the graphical user interface (GUI), we use the Monogame library (v3.8.1.303) because of the extensive documentation, open-source, and familiarity with the library.

This chapter serves as a concise overview of the ideas behind the implementation and a reference for developers interested in extending the project. In Section 5.1, we will explain in detail how we encode the pieces, piece positions, moves, and the game state, along with implement the game logic. In Section 5.2, we will demonstrate the integration of AI agents into the game, detailing how the engine communicates with the agents and the methods used for encoding the AI agents into our software. In Section 5.3, we will present the Universal Hive Protocol, a standardized method of communication between the engines and the user interfaces in Hive, and describe its implementation in our software.

5.1 Implementing Hive

To perform any of the desired experiments, we must first and foremost implement the game. This Section will present the ideas behind implementing different parts of the game.

In Subsection 5.1.1, we will introduce the class that represents the positional encoding of tiles in the game, using the axial hex coordinate system as its basis. Next, in Subsection 5.1.2, we will present the piece and move notations, along with methods of encoding piece and move notations in the code. In Subsection 5.1.3, we will demonstrate a class responsible for encoding game states for Hive, detailing the components and the chosen data structure for board representation. Finally, in Subsection 5.1.4, we will present the class responsible for the game logic implementation, covering rule verification, updates within the game state, and invoking agents to make decisions.

5.1.1 Position Encoding

In Chapter 4, Subsection 4.3.2, we discussed using an axial hex coordinate system for encoding the positions of the pieces. The `HexCoord` represents a coordinate on a hexagonal grid and contains the methods supporting various operations on these coordinates. An instance of `HexCoord` is an axial hex coordinate. A special instance of `HexCoord` is a static read-only value of (-99, -99) that represents a position of pieces that are still in the Hand and called a `HandCoord`.

Static methods in the class allow various operations with these coordinates, such as obtaining the immediate neighbors of a coordinate, getting the hex distance between the two coordinates, and checking adjacency.

5.1.2 Piece And Move Encoding

Moving on to the piece encoding: How do we represent the pieces in the game? Fortunately, the BoardSpace.net community developed a de facto notation [17] that we will use. We assume all pieces are placed in the "pointy-top" direction on the board, meaning their hexagonal tiles have a corner at the top. Each piece's notation starts with the player's color: "w" for White and "b" for Black, always in lowercase. The second letter indicates the piece type, always capitalized: "Q" for the Queen Bee, "A" for the Soldier Ant, "G" for the Grasshopper, "B" for the Beetle, and "S" for the Spider. The third letter indicates the piece number, which is used to differentiate among multiple pieces of the same type (except the Queen Bee). For example, the notation "wS2" means the second white spider, and "bQ" means the black Queen Bee. In our implementation, we encode each piece as an `Enum Piece` in their respective notation. This way, each `Piece` gets assigned a unique numerical value with it (which will be helpful later) and is easy to read in the code. This straightforward approach provides clarity, ensures consistency, and makes the code easier to maintain and update.

When it comes to move notations, it gets a form of "<moving_piece_notation><space><destination>", where the destination is a combination of piece notation and positional indicator. The moving piece is identified first without specifying its initial position, as it either comes from *the Hand* or moves from another location on the board. Next, one of the pieces adjacent to the destination of the moving piece is identified. This reference piece is chosen from several possible options, and the specific choice does not impact the validity of the move.

The position of the moving piece relative to the reference piece is indicated by using "/", "\", or "-" (positional indicators) either preceding or following the piece notation, depending on whether the moving piece is placed to the left or right of the reference piece. Their "pointy-top" placement determines the left or right side of the pieces. If we draw an imaginary line through the top and bottom corners of the tile, we can identify the left and right sides of the piece. For example, the move notation "wA1 bQ-" refers to moving the first White Ant to the right middle side of the Black Queen Bee. If the move is a first move ("wS1", for example, is a move notation for placing the first white Spider as an opening move) or a climb up move ("bB1 wQ", for example, is a move indicating a first black Beetle climbing on top of the white Queen Bee). In that case, no positional indicator is used.

The `Move` class in our implementation handles these notations. Each `Move` instance stores an `Enum` of the moving piece, an `Enum` of the end piece, and the direction indicator of that move. It also includes methods to construct the move strings from the instances and to parse move strings into the `Move` instances. These methods also syntactically validate the move notations.

5.1.3 Game State Encoding

As we mentioned earlier, the game of Hive has no fixed board. However, we can imagine an infinite (or large enough) grid on the axial hex coordinate system that accommodates all the possible positions of the pieces. However, given that we only have 22 pieces, this approach would be impractical and memory-intensive.

An alternative approach is using a hash map as our data structure. A hash map stores key-value pairs, where the key is the axial hex coordinate, and the value is a stack containing the pieces. A hash map is a data structure that maps keys to values using a hash function to determine an index where the values are stored. A stack is a data structure that follows the Last In, First Out (LIFO) principle, meaning the last piece added is the first one removed.

To accommodate for parallelization and ensure safe access to the hash map from different threads, C# offers the `ConcurrentDictionary<K, V>` data structure, which implements a thread-safe hash map. By assigning appropriate key and value types, we can initialize the playing grid as `ConcurrentDictionary<HexCoord, Stack<Piece>> Grid`. This `Grid` is a significant part of the game state representation, storing all pieces in play. Additionally, we define an array called `PiecePositions` of size 22, using `Enum` numerical value of a `Piece` as an index in the array to store the `HexCoord` of that piece. This array allows a quick retrieval of piece positions. The pieces still in hand will have `HandCoord` as their value in the array.

The game state needs to perfectly describe the essential information about the game at any point in time. This is why the `GameState` class encapsulates all of this. It contains the `ConcurrentDictionary<HexCoord, Stack<Piece>> Grid` and the `PiecePositions` array to capture the board. Additionally, it includes the `GameHistory` class, which stores all the moves previously made in the correct order that led to the *current state*. The `GameState` class also includes fields of the `Player` class type, representing each player, and the `BoardState` field, which is an `Enum` that indicates if the game has not started, is in progress, ended in a draw, or resulted in a win for either black or white. Finally, it has an indicator for whose turn it is, storing an `Enum TeamColor` representing the current player's color to move.

5.1.4 Game Logic Implementation

The `GameState` class only contains the information about the state of the game without providing any of the tools to modify this state in any way. This is where we introduce the `Engine` class. This class contains a collection of static methods for enforcing the game rules and updating the `GameState` instance accordingly. Essential methods within the `Engine` class include `IsBreakingHive` and `CanSlideIn`, which enforce the One Hive Rule and the Freedom To Move rule, respectively. The `IsBreakingHive` method temporarily removes the piece in question and performs the connectivity check using the Depth-First Search algorithm. The `CanSlideIn` method checks for the existence of the neighbors that can create the small gap through which the piece can not crawl.

`Engine` class also handles the piece introduction rule using the `CanAddPieceTo` method, which makes a simple check for the color of the immediate neighbors. Another key method is `IsValidPieceMove`, which validates specific piece move-

ments. For the Queen Bee and the Beetle, we check the adjacency. For the Spider and Soldier Ant, we check the existence of the path (for the spider, we also specify the length of the path). For the Grasshopper, we check the existence of at least one piece in the direction of the jump and identify the first empty space.

Most of the AI agents need valid moves to make a decision. The `Engine` class also contains the `GetValidMoves` method that generates all the valid moves for the given `GameState` and the current player. Most of these rule validations are called within a `PlayMove` method, which updates the `GameState` instance when a move is performed.

The game loop is managed through the `Play` (non-static) method in the `Engine` class. The `Engine` class contains an instance of the `GameState`, a private field that we will refer to as the current game state. When `Play` is called, it determines the current player via the `GetPlayer` method. If the current player is a human, the method processes the move from the input argument, updates the current game state accordingly via the `PlayMove` method, and performs self-call with no arguments if the next player is AI. If the current player is AI, it requests the move to play for the AI via the `CalculateBestMove` from the `Player` instance representing the player, updates the current game state accordingly via the `PlayMove`, and recursively calls `Play` with no arguments. This implementation of `Play` is designed to handle interactions between different types of players, whether human or AI.

5.2 Implementing the AI agents

In the previous Section, we discussed how the `Engine` class requests a response from the agent by invoking the `CalculateBestMove` method from the `Player` class. Each AI algorithm can be considered a strategy the `Player` class utilizes. To ensure compatibility with the `Engine` class, each such strategy must implement `IEngineStrategy` interface, which only contains the `CalculateBestMove` method.

The software uses five strategies: Human, Random, Mate In One, MCTS, and Alpha-Beta. The `HumanStrategy` serves as a placeholder for human players, does not execute computations for finding the best move, and is used in identifying non-AI players. The `RandomStrategy` selects moves randomly, providing no strategic depth. The `MateInOne` can identify a winning condition in one move and finish the game but otherwise plays moves randomly. The `MCTSStrategy` employs the Monte Carlo Tree Search algorithm, performing a deep copy on the current state and executing the MCTS algorithm to build the game state tree and choose the optimal move. Similarly, the `AlphaBetaStrategy` performs a deep copy on the current state and utilizes the Alpha-Beta algorithm to construct the game state tree and choose the best move. All these strategies are designed such that their respective algorithms process the game state and determine the best move, which is then outputted through the `CalculateBestMove` method. This design ensures that the Engine class can consistently interact with each strategy, regardless of the underlying algorithm.

5.3 Universal Hive Protocol

The Universal Hive Protocol (UHP), created by Jon Thysell [18], is an open communication protocol for software versions of the board game Hive. Implementing an engine to be UHP-compliant ensures compatibility and interoperability with any UHP-compliant viewer (user interface applications). This allows developers to focus on specific components, such as AI agents and game logic, without needing to build a complete system. The UHP enables communication between the engine and the user interface through a standardized set of textual commands. Our implementation of the `Engine` class adheres to UHP standards. Moreover, the `UHPMessageHandler` class implements the functionality of the Universal Hive Protocol, providing a set of commands a user or software can utilize. This class implements various commands and allows the engine to process the UHP messages efficiently.

A UHP-compliant engine is responsible for all of the logic necessary to play a complete game of Hive. This includes maintaining an accurate representation of the current game state, such as the configuration of the hive (the position of all pieces), determining whose turn it is, identifying valid moves, maintaining a history of moves played, and recognizing when the game has concluded. Additionally, the engine must be capable of performing various actions, such as starting a new game, playing moves, and undoing moves. Furthermore, given the current game state, it should be able to recommend the best possible move.

The `UHPMessageHandler` is crucial for making the `Engine` UHP-compliant. It provides an interface for parsing and executing UHP commands, ensuring the engine responds correctly to any valid UHP message it receives. This class encapsulates multiple command objects, each implementing the `ICommand` interface, standardizing the execution of various commands within the engine. The `UHPMessageHandler` class serves as a bridge of communication between the game logic and the graphical user interface (GUI) in our software. The user input from the GUI gets processed and converted into a UHP command, which is then sent to a `UHPMessageHandler`. The `UHPMessageHandler` invokes the appropriate command, which then performs the necessary actions using the `Engine` class. Our Viewer part of the software is not fully UHP-compliant. It utilizes a non-UHP query from the `UHPMessageHandler` to retrieve the current game state instance, which it then uses to render the state of the game in the application.

6 Results

In this chapter, we will introduce the results of the experiments conducted using the software detailed in Chapter 5. All experiments were run on a laptop with an Intel i5-10300H CPU, which supports a maximum parallelization degree of 8. The configurations for the Monte Carlo Tree Search algorithm were set to a maximum of 300 iterations, the exploration parameter of the UCT formula was set to 100, and all simulations were run until the simulated game was over. These configurations provided the best results against a Random agent, which is why they were selected. We chose a maximum depth of 3 for the Alpha-Beta algorithm, as this offered a balanced trade-off between time and accuracy. A maximum depth of 4 resulted in significantly slower performance, with an average of several minutes per ply.

In Section 6.1, we will introduce the first baseline, a Random agent, which plays only random moves. We compare the effectiveness of Alpha-Beta and Monte Carlo Tree Search agents against this first baseline. In Section 6.2, we will introduce the second baseline agent, referred to as the "Mate In One" agent. This agent can identify a winning move and execute it to complete the game, but if there is no such move, it defaults to performing random moves. In Section 6.3, we will put Alpha-Beta and MCTS agents against each other and compare their performances. In Section 6.4, we will put the winner from Section 6.3 against an intermediate human player and an expert human player.

6.1 AI Agents Versus a Random Agent

A random agent is an agent that, given a list of all valid moves at its current turn, will select a random move from that list and execute it in-game. It has not understanding of strategy or tactics.

In this experiment, we put the Monte Carlo Tree Search agent against a Random agent. We simulated 100 games, with the MCTS agent starting as player White in 50 games and as player Black in the other 50 games. The results of this experiment are summarized in the Table 6.1.

MCTS Starts As	Wins	Draws	Losses	Avg Turn Count
White	66%	30%	4%	58.54
Black	76%	24%	0%	50.14

Table 6.1 MCTS vs Random

The format of the table will be the same for all of the experiments. "Player Starts As" indicates the color the player started as (White or Black) during the experiment. In the tables, instead of "player" we will use the name of the strategy used by the agent. "Wins", "Draws", "Losses" represent the percentage of games that resulted in a win, draw, or a loss for that player. In some tables, when the number of experiments is quite low, next to the percentages in parenthesis, we will include the exact number of wins, draws, or losses. "Avg Turn Count" refers to the average number of turns to complete each game.

From the Table 6.1, we can observe the MCTS strategy winning most of the games, beating the Random agent in most cases. However, we can also observe a high average turn count in the experiments. This is due to the large number of draws that occurred because of reaching the limit of 100 turns per game, demonstrating how MCTS struggles to close out the games against such a basic strategy. We can also observe a small number of losses that occurred during MCTS starting as white; this can highlight a potential weakness of this strategy, where it fails to defend itself against even random moves.

Next, we put the Alpha-Beta agent against a Random agent. Again, we simulated 100 games, where the Alpha-Beta agent started as player White in 50 of those games and as player Black in the rest. The results of this experiment are summarized in the Table 6.2 below.

Alpha-Beta Starts As	Wins	Draws	Losses	Avg Turn Count
White	100%	0%	0%	15.64
Black	100%	0%	0%	15.76

Table 6.2 Alpha-Beta vs Random

The Alpha-Beta agent won 100% of the games regardless of starting as White or Black, with average turn counts of 15.64 and 15.76, respectively. This demonstrates the superior performance of the Alpha-Beta agent against the Random agent.

6.2 AI Agents Versus a "Mate In One" agent

The "Mate In One", similarly to the Random agent, has no understanding of tactics or strategy. Given a list of valid moves, it will iterate over each of them to see which will result in an instant win for the agent. If none of such moves exist in the list, the "Mate In One" agent defaults to playing a random move.

In this experiment, we put the MCTS agent against a "Mate In One" agent. We simulated 100 games, with the Monte Carlo Tree Search agent starting as player White in 50 games and as player Black in the other 50 games. The results of this experiment are summarized in the Table 6.3.

MCTS Starts As	Wins	Draws	Losses	Avg Turn Count
White	38%	48%	14%	68.66
Black	38%	46%	16%	62.46

Table 6.3 MCTS vs Mate In One

From the Table 6.3, we can observe the MCTS strategy having the majority of the wins, however not as confident as previously. The "Mate In One" agent exploited the weak defense from the MCTS agent, resulting in 14-16% winning rates in its favor. Again, we can observe a high average turn count (both above 60 turns). This can be explained by the fact that most of the games resulted in a draw due to exceeding the turn limit of 100 turns (almost 50% in both cases), which further shows how MCTS struggles to close out a game.

In the next experiment, we put the Alpha-Beta agent against a "Mate In One" agent. Again, we simulated 100 games, where the Alpha-Beta agent started as player White in 50 of the games and as player Black in the rest. The results of this experiment are summarized in the Table 6.4.

Alpha-Beta Starts As	Wins	Draws	Losses	Avg Turn Count
White	100%	0%	0%	14.36
Black	100%	0%	0%	14.34

Table 6.4 Alpha-Beta vs Mate In One

As we can observe from the Table 6.4, the Alpha-Beta agent demonstrates a superior performance against the "Mate In One". Not a single loss out of the 100 games, and the games were all relatively short (both around 14 turns on average).

6.3 Alpha-Beta Agent Versus Monte Carlo Tree Search Agent

Building on the previous experiments where both AI agents competed against Random and MateInOne agents, it was evident that the Alpha-Beta agent outperformed the MCTS agent in terms of the winning rates. We experimented with the Alpha-Beta agent against the MCTS agent to further evaluate their performance. A total of 20 games were simulated, with the Alpha-Beta agent starting as White in 10 games and the MCTS agent starting as White in the remaining 10 games. The results are summarized in Table 6.5.

Alpha-Beta Starts As	Wins	Draws	Losses	Avg Turn Count
White	100%	0%	0%	13.6
Black	100%	0%	0%	20.5

Table 6.5 Alpha-Beta vs MCTS

The Table 6.5 shows that the Alpha-Beta agent won all 20 games regardless of its starting color, with an average turn count of 13.6 when starting as White and 20.5 when starting as Black.

6.4 AI Agent Versus a Human

This section evaluates the performance of the Alpha-Beta agent against human players with varying levels of experience. We define an intermediate player as someone with a fundamental understanding of the game rules, basic strategies, and common tactics. They have accumulated approximately one year of experience, which typically includes regular play and exposure to a variety of in-game scenarios. This level of experience allows the player to make competent moves, anticipate opponent strategies to a certain extent, and execute basic game plans effectively.

In contrast, we define an expert player as someone with a profound understanding of the game, advanced strategies, and complex tactics. They have accumulated at least ten years of experience, including extensive practice and deep analytical study of the game. This level of experience enables the player to make optimal moves and counter opponent strategies consistently. Expert players can think several moves ahead, adapt to changing game dynamics and exhibit a high level of strategic foresight and tactical precision.

The Alpha-Beta's dominance was evident in the previous experiments, establishing it as the candidate for this experiment. Consequently, we set up a series of 10 games between the Alpha-Beta agent and an intermediate human player, with the agent alternating between playing as White and Black. The outcomes of these games are presented in Table 6.6.

Human Starts As	Wins	Draws	Losses	Avg Turn Count
White	40% (2)	20% (1)	40% (2)	16.2
Black	40% (2)	0% (0)	60% (3)	16.4

Table 6.6 Intermediate vs Alpha-Beta

The results from the Table 6.6 reveal that the Alpha-Beta demonstrated a notable performance against an intermediate-level human player. Out of 10 games, Alpha-Beta managed to win five, draw one, and lose four games, all with a relatively small average turn count. These outcomes highlight our refinements in the AI's capabilities for the board game Hive. Previous attempts, as discussed in the Related Work chapter, struggled to perform well against human opponents. While these results are promising, it is important to note that the intermediate level is not the highest benchmark; we must also evaluate performance against an expert player.

After observing successful results, we put the Alpha-Beta agent against an expert human player. The format remains the same. The only difference is the skill level of the player.

Human Starts As	Wins	Draws	Losses	Avg Turn Count
White	60% (3)	0% (0)	40% (2)	16.4
Black	40% (2)	20% (1)	40% (2)	18.4

Table 6.7 Expert vs Alpha-Beta

The Table 6.7 reveals the winner of the last experiment to be a human. However, the margin was narrow. Out of 10 games, Alpha-Beta won four, lost five, and drew one game. Despite its defeat, the agent performed relatively well. These outcomes highlight the advancements in enhancing the AI's capabilities, though further refinements are necessary to improve decision-making quality. The subsequent chapter will explore some of these potential improvements.

Conclusion

The primary focus of the thesis was to implement and evaluate two AI strategies, Alpha-Beta and Monte Carlo Tree Search, in the board game Hive. To accomplish this, we had to create a software platform that facilitated the integration and evaluation of these AI agents. We consider all of the intended goals to be successfully achieved.

A highlight from the comparative analysis was the dominant performance of the Alpha-Beta algorithm. It demonstrated perfect results against the two baselines and the MCTS algorithm. It won against an intermediate-skilled human player and almost won against an expert human player.

Despite this, we acknowledge areas for improvement. One such area is the execution time for the algorithms. For simplicity, during the experiments, we mainly focused on the quality of the decisions rather than their speed. Therefore, we set no time limitations per move for any of the algorithms. This was most noticeable when performing the experiments with the MCTS algorithm; often, it would take several minutes for the agent to decide on a move. The time issue was also noticeable for the Alpha-Beta agent, but not to the same degree. Implementing iterative deepening or a better heuristic for move ordering could speed up the decision-making process of the Alpha-Beta algorithm.

Another such area is the Monte Carlo Tree Search implementation. By incorporating a heuristic for simulations to play more promising moves instead of uniformly random ones, the efficiency and accuracy of the algorithm could be enhanced. Faster simulations would lead to quicker games, allowing for more iterations of the algorithm and, consequently, more accurate outcomes. A different method of parallelization could also be used for improving the algorithm.

In conclusion, while the Alpha-Beta algorithm outperformed MCTS in our experiments, several approaches exist to enhance both algorithms' performance and efficiency. Future work could focus on optimizing decision-making speed and refining heuristics to further improve the AI agents' performance in Hive.

Bibliography

1. Hsu, Feng-hsiung. IBM's deep blue chess grandmaster chips. *IEEE micro*. 1999, vol. 19, no. 2, pp. 70–81.
2. Silver, David; Schrittwieser, Julian; Simonyan, Karen; Antonoglou, Ioannis; Huang, Aja; Guez, Arthur; Hubert, Thomas; Baker, Lucas; Lai, Matthew; Bolton, Adrian, et al. Mastering the game of go without human knowledge. *nature*. 2017, vol. 550, no. 7676, pp. 354–359.
3. Gen42. *Official website for the Game of Hive* [online]. 2015. [visited on 2024-06-28]. Available from: `https://www.gen42.com/games/hive`.
4. Burmeister, Jay; Wiles, Janet. The challenge of Go as a domain for AI research: a comparison between Go and chess. In: *Proceedings of Third Australian and New Zealand Conference on Intelligent Information Systems. ANZIIS-95*. IEEE, 1995, pp. 181–186.
5. Kampert, Duncan; Varbanescu, Ana-Lucia; Müller-Brockhausen, Matthias; Plaat, Aske. Mimicking the Human Approach in the Game of Hive. In: *2021 IEEE Symposium Series on Computational Intelligence (SSCI)*. 2021, pp. 1–8. Available from DOI: `10.1109/SSCI50451.2021.9659999`.
6. McGuile, Connor Michael. *Swarm Artificial Intelligence in Hive* [online]. 2020. [visited on 2024-06-19]. Available from: `https://info.cs.st-andrews.ac.uk/student-handbook/files/project-library/cs5099/cmm40-Final_report.pdf`. MA thesis. University of St Andrews.
7. Blixt, Rikard; Ye, Anders. *Reinforcement learning ai to hive* [online]. 2013. [visited on 2024-06-19]. Available from: `https://www.diva-portal.org/smash/record.jsf?pid=diva2%3A668701&dswid=-4144`.
8. Konz, Barbara Ulrike. *Applying Monte Carlo Tree Search to the Strategic Game Hive* [online]. 2012. [visited on 2024-06-19]. Available from: `https://www.aot.tu-berlin.de/fileadmin/files/lehre/diplomarbeit/BA_Barbara_Konz.pdf`. Bachelor's Thesis. Technical University of Berlin.
9. Bunth, Tamás. *Solving Hive Board Game with Deep Reinforcement Learning* [online]. 2019. [visited on 2024-06-19]. Tech. rep. Budapest University of Technology and Economics. Available from: `http://tdk.bme.hu/VIK/DownloadPaper/Mely-megerositeses-tanulas-alapu-Hive-jatekos`.
10. Goede, Danilo de; Kampert, Duncan; Varbanescu, Ana Lucia. The Cost of Reinforcement Learning for Game Engines: The AZ-Hive Case-study. In: *Proceedings of the 2022 ACM/SPEC on International Conference on Performance Engineering* [online]. 2022, pp. 145–152 [visited on 2024-06-19]. Available from: `https://dl.acm.org/doi/abs/10.1145/3489525.3511685`.
11. Coulom, Rémi. Efficient selectivity and backup operators in Monte-Carlo tree search. In: *International conference on computers and games*. Springer, 2006, pp. 72–83.

12. Chaslot, Guillaume Maurice Jean-Bernard. *Monte-Carlo Tree Search* [online]. [N.d.]. [visited on 2024-06-19]. Available from: https://project.dke.maastrichtuniversity.nl/games/files/phd/Chaslot_thesis.pdf.

13. Kocsis, Levente; Szepesvári, Csaba. Bandit based monte-carlo planning. In: *European conference on machine learning*. Springer, 2006, pp. 282–293.

14. Neumann, J v. Zur theorie der gesellschaftsspiele. *Mathematische annalen*. 1928, vol. 100, no. 1, pp. 295–320.

15. Knuth, Donald E.; Moore, Ronald W. An analysis of alpha-beta pruning. *Artificial Intelligence*. 1975, vol. 6, no. 4, pp. 293–326. ISSN 0004-3702. Available from DOI: https://doi.org/10.1016/0004-3702(75)90019-3.

16. Zobrist, Albert L. A new hashing method with application for game playing. *ICGA Journal*. 1990, vol. 13, no. 2, pp. 69–73.

17. BoardSpace.com. *Hive Notation* [online]. [N.d.]. [visited on 2024-07-11]. Available from: http://www.boardspace.net/english/about_hive_notation.html.

18. Thysell, Jon. *Univeral Hive Protocol* [online]. 2015. [visited on 2024-07-13]. Available from: https://github.com/jonthysell/Mzinga/wiki/UniversalHiveProtocol.

List of Figures

List of Tables

A User Documentation

This section demonstrates how to install and run the application developed as part of this thesis. The software is called HiveEngineMFF and is built to simulate the board game Hive and facilitate the integration of the AI agents. It offers only a graphical user interface and is supported on Windows and Linux operating systems.

In Section A.1, we will present the necessary dependencies needed for installing and running the application. In Section A.2, we will demonstrate the steps needed for running the application using the source code. In Section A.3, we will explain how to use the application.

A.1 Dependencies

The only installation requirement is:

- .NET SDK 6.0

After installing this, a user can run the application, which is instructed in A.2.

For further development, the following dependencies were used in developing the software:

- Monogame framework (version 3.8.1.303), more precisely, these two packages from the framework (both can be acquired as a NuGet package, but recommended to download the framework from the official website of Monogame):
  - MonoGame.Framework.DesktopGL (version 3.8.1.303)
  - MonoGame.Content.Builder.Task (version 3.8.1.303)
- TinyDialogsNet (version 2.0.0). (can be acquired as a NuGet package)

These are not required for the user who wants to run the application.

A.2 Installation

To install the application from the source code, a user must install all the required installation dependencies. This guide works for both Linux and Windows

1. Clone the following GitLab repository containing the source code for the project:

```
git clone https://gitlab.mff.cuni.cz/teaching/nprg045/majerech/ahmadov.git
```

2. Navigate to the cloned directory in the terminal. Then navigate to the "Hive" folder and run the following command:

```
dotnet run
```

After running the command, a GUI application will appear.

A.3 Running the Application

When starting the application, a user will be met with the main menu screen. Figure A.1 represents that.

Figure A.1 Main Menu in the application

Users can navigate the application using either a keyboard or a mouse. When using the keyboard, the arrow up and arrow down keys allow users to move the selection to the button above or below the currently selected one, respectively. The "Enter" key can then activate the selected button. Alternatively, users can navigate with a mouse by hovering over the desired button and clicking the left mouse button once to activate it. It is important to note that the mouse cursor takes priority in the selection of buttons.

We describe each button of the Main Menu as follows:

- Selecting the `Start a Game` option initiates a new game instance with the current settings. This action transitions the interface into the Game View State, which will be detailed later. Essentially, this means that either a game or a simulation will start. A simulation involves an instance where both player strategies are the AI agents, whereas a normal game can be played between human players or between a human player and an AI agent.

- Selecting `Settings` transitions the interface into Settings Menu. The controls for the Settings Menu are the same, meaning we can choose and activate buttons via keyboard or mouse. The list of all the options and descriptions will be detailed later in the document.

- Selecting `Review Saved File` will open a File Dialog, allowing users to navigate to and select a saved game file (*.hivesave.txt). Once a file is selected, the interface transitions into a Review Mode, letting users review a saved game instance. Further details on the Review Mode are discussed later.

- Selecting `Exit` closes the application.

In Subsection A.3.1, we will detail each settings option in the Settings Menu. In Subsection A.3.2, we will demonstrate how to use the application for playing the game or running simulations. In Subsection A.3.3, we will explain how to review the saved game files using the application. The experiments run during the Results Chapter are also considered to be saved files.

A.3.1 Settings

When selecting the Settings option in the Main Menu, the application transitions into Settings Menu, which can be seen in Figure A.2.

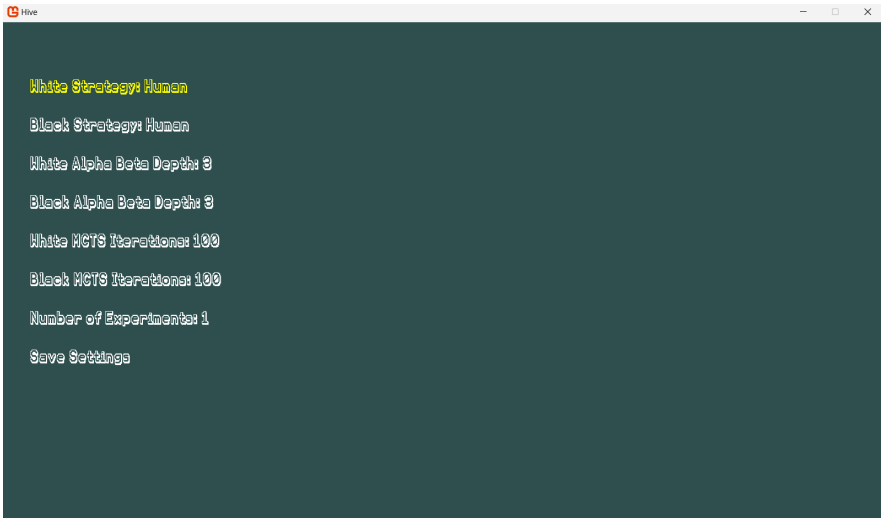


Figure A.2 Settings Menu

Here, a user can set up the game's settings, mainly the players' strategies, agent-specific settings, and the number of simulations to run an experiment. Each button has certain possible values they can represent. The value selection is cyclical. By pressing "Enter" or left mouse click on selected buttons, a user can change to the next value. We define each button of the Settings Menu as follows:

- `White Strategy`. A strategy for the player White. The values are Human, MCTS, AlphaBeta, Random, and MateInOne. Each of these values represents an AI strategy or a human player.
- `Black Strategy`. A strategy for the player Black. The values are the same as in the White Strategy button.
- `White Alpha Beta Depth`. Chooses the maximal depth for the Alpha-Beta search tree if the player white uses Alpha-Beta as its strategy. The values are from 1 to 5.
- `Black Alpha Beta Depth`. Chooses the maximal depth for the Alpha-Beta agent playing as player black. The values are from 1 to 5.
- `White MCTS Iterations`. Chooses the total number of iterations for the MCTS agent playing as the player White. The values are 10, 20, 50, 100, 300.

- `Black MCTS Iterations`. Chooses the total number of iterations for the MCTS agent playing as the player Black. The values are 10, 20, 50, 100, 300.
- `Number of Experiments`. A setting for simulations. Determines the number of simulations to run. So, suppose we have a simulation for Alpha-Beta versus a Random and set a simulation number to 100. In that case, this experiment will run 100 games of Alpha-Beta playing against the Random agent. The values are 1, 5, 10, 25, 50, 100, 200, 1000.
- `Save Settings`. Saves the settings for the game.

A.3.2 Playing the Game

When selecting the Start a Game option in the Main Menu, the application transitions into Game View. Here, a user can, depending on the settings, play against another human, against an AI agent, or observe the simulations happening in real time. Figure A.3 depicts the Game View state.

Figure A.3 Game View state of the application. Human Vs Human.

The Game View can be controlled via a mouse. The left mouse button is used to move pieces, introduce pieces, or press In-Game Interface buttons (save, skip, or undo). Hovering over a stack of pieces with a mouse cursor will reveal all the pieces in that stack. The pieces, from the bottom of the stack to the top, will be displayed at the middle and bottom of part of the window. They will appear from left to right, where the most left piece is the bottom piece, and the most right piece is the one at the top of the stack. The right mouse button moves the camera; by holding and dragging it, a user can move their camera to observe the board better. A user can also use a scroll wheel to zoom in or zoom out the camera for a clear view of the board.

The Game View also recognizes keyboard input; however, it is not enough to play the game. For In-Game Interface buttons, a player can activate skip, save, and undo by pressing “P”, “S”, and “U”, respectively. A player can zoom in and

zoom out by pressing the "+" and "-" keys, respectively. Additionally, by pressing "R" on the keyboard, a user will move the camera back to the center.

The set of pieces on the sides represents the pieces in Hand. To introduce a new piece from hand to the board, a player can drag (by pressing the left mouse button and holding till the desired destination) the piece of their color from one of the sides and drop it (releasing the left mouse button) where they think the piece should be placed. For turn one, when player White starts the game, dragging and dropping the piece will always drop it in the center. A player must drag and drop to the desired position for the rest of the situations. The desired position can be the center of the imaginary hexagon-shaped tile neighboring any of the pieces (given that it complies with the game rules). The game will automatically place the tile in its right position if the drop happens near the center of the desired tile. If no more pieces from the hand are available of a specific type, the game will not let the user drag in more pieces of that type. We also use dragging motion to move a piece on the board. A user can click and hold the left mouse button to select the piece to move and drop it in the place the user desires as long as it complies with the game rules.

In the bottom left corner of the window, text represents the turn count and current turn.

The error messages, notifying a player for attempting to perform an illegal move, will appear as floating messages that fade away within 5 seconds. Figure A.4 demonstrates that.

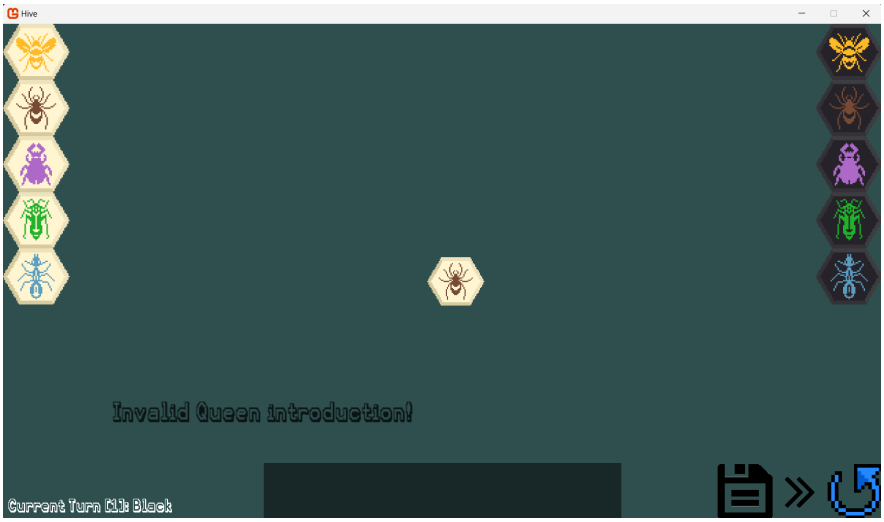


Figure A.4 A message pops up when the player Black tries to introduce Queen Bee illegally on turn 1. It floats up slowly and fades away in 5 seconds.

The three button icons at the bottom right corner of the screen are called In-Game Interface buttons. There are three buttons, each with an action. The actions of the buttons, in order, are save, skip, and undo. Saving a game will prompt the application to open a Save File Dialog, where a user can give a name for a file that will be saved as a .hivesave.txt file in the SaveFiles folder. Skip refers to an action of playing the move "pass". By performing a pass move, a user skips their turn without moving or adding a piece; this is only allowed when the player has no legal moves available. Undo refers to an action of undoing the last move. If the Game View is set up for a human vs human, the button will undo

only one move. If the Game View is set up for an AI agent versus a Human player, the button will undo two moves so that the player can play again. These In-Game Interface buttons can also be activated via the keyboard. By pressing the “S” key, a user activates the Save button; by pressing the “U” key, a user activates the Undo button; and by pressing the “P”, a user activates the Skip (Pass) button. The In-Game Interface buttons are present only for non-simulations. This means that if, in the Settings Menu, a user assigns both strategies to be non-human, then the Game View starts the experiment run for the number of times specified in the settings. During this period, the user can only move the camera and hover over stacks of pieces on the board; no modifications by the user to the game state are allowed. All the experiments are automatically saved into the SaveFiles folder.

When the game is over or the experiments are finished, a message in the middle of the window announces the winner of the last game. When the game is over, a user cannot modify the game state by performing or undoing moves. From here, a user can press the “M” button on their keyboard to go back to the Main Menu or close the application by pressing the “ESC” button. Figure A.5 demonstrates that. It is important to note that during the Game View state of the application, a user can not go back to the Main Menu before the game or the simulation is over.

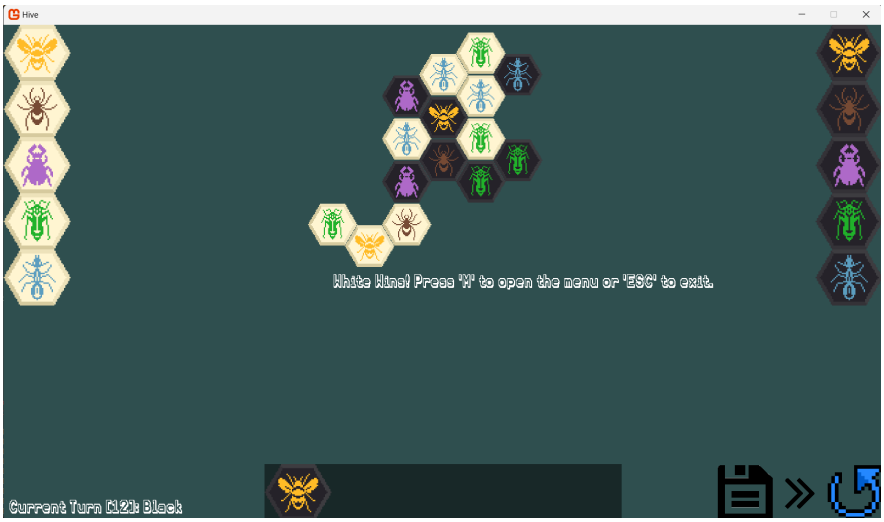


Figure A.5 Game Over. Human vs Random. White wins.

A.3.3 Reviewing Saved Files

When selecting the Review Saved File option in the Main Menu, a user will be prompted to choose a save file from the File Dialog; then, after selecting the desired save file, the application transitions into a Review state. Here, a user can observe how the saved game developed over the turns. A user cannot modify the game state; they can only observe the next or previous moves during that game (with respect to the current turn). We can notice how the In-Game Interface buttons for the Review state differ. The buttons introduced earlier are not present since we cannot update the game state. However, the Review contains two arrow-shaped buttons. The left-most button (arrow pointing to the left) performs an action of reviewing the previous move. This action can also be called by pressing the left arrow key on the keyboard. The right-most button (arrow pointing to the

right) performs an action of reviewing the next move. This action can also be called by pressing the right arrow key on the keyboard. It is important to note that each left mouse button click on the arrows performs the action once while pressing and holding the left and right arrow keys, which will perform the action multiple times. A user can leave the Review mode at any point by pressing the "M" button on the keyboard. Below, Figure A.6 demonstrates that.

Figure A.6 A Review state of the application. It shows the start of the game when no moves have been played yet.

B Attachments

Below, we present the attachments that are a part of a digital archive.

B.1 Source Code

This folder contains all the source code and the necessary files used to build the HiveEngineMFF application as a part of this thesis. This folder will also include a folder named SavedFiles, where all the game files of the experiments performed are stored.

B.2 Executables

This folder contains two folders, each named after the operating system and containing the executables for Windows and Linux operating systems. For Windows, it is enough to run the Hive.exe file. For Linux, it might require the execute permission. One can achieve that by running `chmod +x Hive`.

B.3 GitLab Repository

The GitLab repository, containing the source code, can be found at `https://gitlab.mff.cuni.cz/teaching/nprg045/majerech/ahmadov`