Technische Universität Berlin

Applying Monte Carlo Tree Search to the Strategic Game Hive

Bachelorarbeit
am Fachgebiet Agententechnologien in betrieblichen Anwendungen und der Telekommunikation (AOT)
Prof. Dr.-Ing. habil. Sahin Albayrak
Fakultät IV Elektrotechnik und Informatik
Technische Universität Berlin

vorgelegt von
Barbara Ulrike Konz

Betreuer: Prof. Dr.-Ing. habil. Sahin Albayrak,
Dr.-Ing. Stefan Fricke

Barbara Ulrike Konz
Matrikelnummer: 319732
Otto-Suhr-Allee 36
10585 Berlin

Erklärung der Urheberschaft

Ich erkläre hiermit an Eides statt, dass ich die vorliegende Arbeit ohne Hilfe Dritter und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe; die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht. Die Arbeit wurde bisher in gleicher oder ähnlicher Form in keiner anderen Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht.

Ort, Datum Unterschrift

Zusammenfassung

In dieser Arbeit wurde das Spiel HIVE und eine dazugehörige künstliche Intelligenz (KI) implementiert. Damit die HIVE-KI einen für sie günstigen Zug findet, wird *Monte-Carlo Tree Search* (MCTS) verwendet. Dieser Algorithmus kann mit *Upper Confidence Bound Applied on Trees* (UCT), einer Auswahlstrategie, verbessert werden.

Zur Evaluierung wurden sowohl mehrere Partien von unterschiedlichen KI-Spielern untereinander gespielt, als auch beobachtet, wie sich die KIs in unterschiedlichen Situationen verhalten und wie verlässlich dieses Verhalten ist. Dabei hat sich herausgestellt, dass beide, MCTS alleine und unter Verwendung von UCT deutlich bessere Züge wählen, als ein zufallsbasierter Zuggenerator. Außerdem wurde auch deutlich, dass MCTS mit UCT in der Regel besser spielt als MCTS alleine.

Contents

Chapter 1

Introduction

Playing games has a long tradition in human culture. There are many different ways a game can look like. An important category of games are board games. One of the most famous board games is CHESS. According to Cazaux (2012) early forms of CHESS are at least 1400 years old. Tournaments have an important role for games and so it is no surprise that when computers were invented one goal became to beat human players with a machine Coles (2012). In order to win the computer has to make a decision. In CHESS this could mean to choose the move with the best possible outcome for a player.

In many games several players have to make a decision one after another. This is also called sequential decision making and has been a challenge in a wide variety of fields, ranging from budget setting to network monitoring to artificial intelligence (AI) (Littman, 1996). The main task is to maximize the result by selecting the best possible actions, i.e. by making the right decision.

In the field of AI, two-player games are a classical and excellent test bed. They provide a simple micro-world with a comparatively simple set of rules and only two actors. For many two-player games (e.g., CHESS, CHECKERS, BACKGAMMON) human players have been defeated by programs using brute-force tree search with knowledge-based heuristics or reinforcement learning (Gelly et al., 2012). In contrast there are other games, like GO, for which the approach of domain knowledge-based tree search has failed to produce reasonable good results. Therefore, other methods needed to be found.

After a long time of research, Monte-Carlo tree search (MCTS) was introduced. This novel way of making decisions led to big improvements for AIs especially in GO. MCTS gets along with minimal or no prior knowledge and bases its decisions on simulating random games in self-play. Later, several extensions and refinements have been developed and MCTS was applied successfully to many games, especially classical two-player games (Gelly et al., 2012).

1.1 Motivation

This work focuses on the implementation of an artificial intelligence (AI) for playing the game HIVE. For that *Monte-Carlo tree search* (MCTS) is applied to the game. HIVE is a quite recently developed two-player strategic game. Unlike CHESS or GO it has an open game field without borders. Furthermore, a variation of MCTS which uses *Upper Confidence Bound Applied to Trees* (UCT) has been implemented. This leads to the following research questions:

- Can Monte-Carlo tree search be successfully applied to the game HIVE, and how do different Monte-Carlo tree search variations perform?
- Which of the two approaches, simple MCTS and MCTS with UCT, turns out better? Why is one AI better than the other?
- If two AIs of the same type are playing against each other do minor variations improve the game-play?

The following chapters attempt to find answers to these questions.

1.2 Related Work

MCTS has been applied to a wide range of games. However, in this work, the focus will be confined to the following three examples.

GO is one of the most famous and most important examples how Monte-Carlo tree search can improve the gameplay of an artificial intelligence. As described by Gelly et al. (2012) programs have made a great pace from weak kyu level to professional dan level in 9×9 GO, and to strong amateur dan level in 19×19 only by using MCTs instead of knowledge-based heuristics. Clearly, MCTS fits better for GO than previously used methods for evaluating positions and value of tokens.

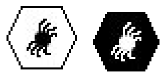
Another approach was made by Nijssen (2007) who applied MCTS to OTHELLO. In this case, the AI was able to play well against simple AI players, but hardly had a chance against experienced human players or strong AI players.

But these were only examples for two-player games. Monte-Carlo tree search can also be used for multi-player games. As mentioned in Chaslot et al. (2009) MCTS has

(a) White and black Beetle

(b) White and black Ant

(c) White and black Queen Bee


(d) White and black Spider

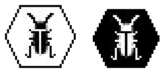

(e) White and black Grasshopper

Figure 1.1: Tokens for all types of insects

successfully been used for an artificial intelligence in the game SETTLERS OF CATAN and beaten all preceding implementations of AIs. Furthermore, Monte-Carlo tree search is applied on video games to make AIs behave human-like (Chaslot et al., 2008).

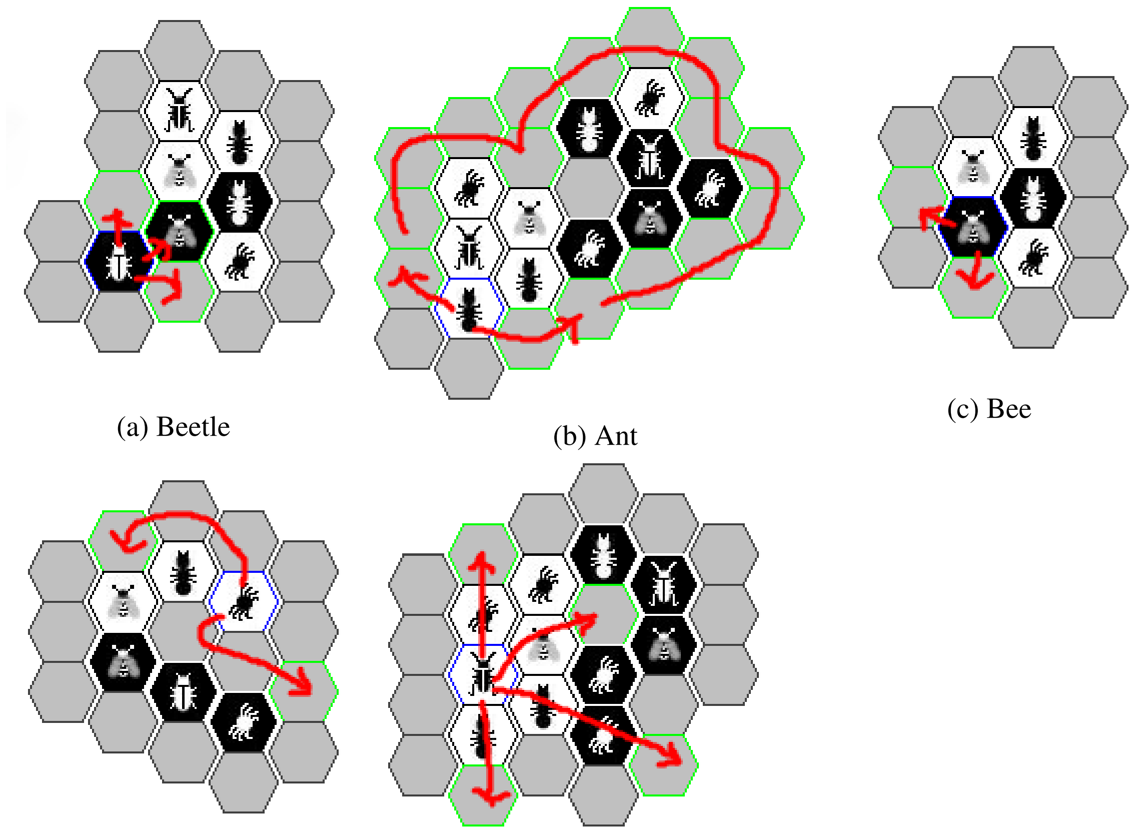
1.3 The Game Hive

This section basically focuses on explaining the rules of HIVE followed by its comparison to GO and CHESS to see whether there are similarities to one of these games.

1.3.1 The Rules of Hive

HIVE is a two-player strategic game with open information. There are two possible colors to play, namely black and white. Each player has eleven pieces. The following rules are from the game created by Yianni (2007, cf.). There are five types of tokens. An example for each insect is shown in Figure 1.1. How they can be moved in is displayed in Figure 1.2. The blue framed field indicates the piece to move and the green framed fields and the arrows show potential fields an insect can go to. The rules for all five types of insects will be explained in detail:

- The **Beetle** can only be moved one space per turn. Unlike any other insect it can be placed on top of others. There can be stacks of several beetles. See Figure 1.1a for the stones and Figure 1.2a for possible moves.
- The **Ant** can be moved around other tokens. The ant can move as many spaces as necessary. This makes the ant to a very valuable piece in the field. The stones are shown in Figure 1.1b. See Figure 1.2b for possible moves.
- The **Bee**, like the beetle, can be moved one space per turn. For reference, the symbols of the bee are shown in Figure 1.1c. In Figure 1.2c all possible moves can be seen.


(a) Beetle (b) Ant (c) Bee

(d) Spider (e) Grasshopper

Figure 1.2: Possible moves for all types of insects

- The **Spider** shown in Figure 1.1d, can be moved three spaces per turn. It must move in a path and cannot visit a space twice during one move. See Figure 1.2d for possible moves.
- The **Grasshopper** shown in Figure 1.1e, can jump over other insects, but only in one direction until the next empty space is reached. There has to be at least one token to jump over. See Figure 1.2e for possible moves.

The game starts with one player placing a piece on the field. As long as a players queen is not placed on the field only new pieces can be put on the field, but the player is not allowed to make moves. New pieces can only be placed next to the players own pieces and are not allowed to touch the opponent's insects. The second turn is an exception of this rule, because only one piece is on the field and it is of the opponents color. If the queen is not played until a players fourth turn it has to be placed. Once the bee is put on the field the player is allowed to move his pieces on the field as well. It is not allowed to part the hive during or at the end of a move. This means it is not allowed to have two separate clusters which are not connected with each other. A piece must always touch at least on other piece even during moving. To make a move between two other tokens there has to be enough space for a token to pass through. This situation is shown in Figure 1.2b where the white ant cannot reach the empty field between the white queen and the black ant. The object of the game is to surround the opponents Queen Bee. If both the queens are completely surrounded by other insects, the game results in draw.

1.3.2 Comparison to Chess and Go

CHESS and GO are two well-known classical two-player games with a given field. Although both are the same type of game, different approaches, knowledge-based heuristics or MCTS, are necessary to get a well performing AI. Therefore, HIVE will be compared to both games to see which concept might be best.

According to Gelly et al. (2012) games like GO are a challenge for classical AI approaches for several reasons. First, the combinatorial complexity of GO is huge. It has approximately 200 possible moves per turn, whereas CHESS has only 37 moves. With around 70 moves HIVE lies in between those games. Furthermore, for GO there are more than 10^{170} possible moves compared to much less in CHESS with 10^{47}. According to the fact that Hive has an open field unlike both CHESS and GO, and that is has hexagon fields instead of squares, which increases the number of combinations gives the assumption of a very high number of moves.

The long-term influence that a clever placing of stones at the beginning of a game can have is another factor which makes it difficult for classical approaches like Minimax to perform well in GO. This is also a problem for HIVE. Clever moves at the beginning can decide the winner of a game. For instance, clever positioned stones at the beginning can immobilize opponents insect later in the game.

1.3.3 Choosing the Appropriate Search Algorithm for Hive

After comparing the two classical games GO and CHESS to HIVE, it is important to decide which of both approaches, knowledge-based (Minimax) or MCTS, fits better. For that, both algorithms have to be examined closer. A more in-depth explanation of MCTS can be found in Chapter 2. A short introduction to Minimax is given here. According to Pardalos (1995) Minimax is a depth-first search. Usually, a depth is given and a search does not go further even though the end of a game is not reached, yet. An evaluator calculates the score of each node for all nodes at the terminal level by using an evaluation function. The score and the path are propagated back to the root, where the best move will be chosen.

As stated earlier, HIVE has a branching factor of about 70 nodes per move. Usually at the beginning and at the end it is less but in the middle part of the game it can be much higher as well. Then a branching factor of more than 100 is possible. This suggests to use MCTS instead of Minimax, because Minimax would need too much time and resources during the generation of a tree.

It is also necessary to design a good evaluation function for Minimax. CHESS and GO are traditional and famous games, for which a lot of people developed strategies how to play best. On the other hand HIVE is a recently developed game. This makes it difficult to decide how do evaluate the setting of a field and certain positions. Considering these two facts, it is appropriate to use MCTS for a first approach to implement an artificial intelligence for HIVE.

1.4 Further Structure

In this section, a short summary of the following chapters is given. Chapter 2 focuses on the explanation of MCTS and its four steps. Additionally, a further illustration of UCT is presented. After that, Chapter 3 summarizes the design and implementation of HIVE and all types of AIs. The evaluation of the results of the game-play and situation evaluation are shown and discussed in Chapter 4. The last chapter answers the earlier stated research questions and illustrates the perspective of future work

Chapter 2

Monte-Carlo Tree Search

This chapter will focus on defining Monte-Carlo tree search algorithm in a more granule level. It will also be discussed how MCTS is combined with Upper Confidence bounds applied to Trees (UCT) as summarized in Chaslot (2010, p.18 ff.).

2.1 MCTS - An Introduction

The aim for a well performing AI is to find the best among all possible moves. These moves are also called candidate moves. If there is no candidate move, then the player has to pass. If there is only one candidate move this is by definition the best move. If there are more than one candidate moves, the player has to make a decision (Nijssen, 2007).

As described in Chaslot (2010), Monte-Carlo Tree Search is a best-first search method that does not require a positional evaluation function and has a randomized exploration of the search space. This means the game tree is build by choosing steps randomly. Every node in the tree represents a possible future situation in the game and the root is the current situation on the field. According to Chaslot (2010), a game has to satisfy the following conditions so MCTS can be used:

- **The game scores are bounded:** Bounded means there is a strict definition of win, lose, and draw. No other outcomes for a player are possible.
- **Perfect information is given:** All rules are known as well as all tokens in and outside the field.
- **Simulations are limited terminate relatively fast** The length of a game is limited.

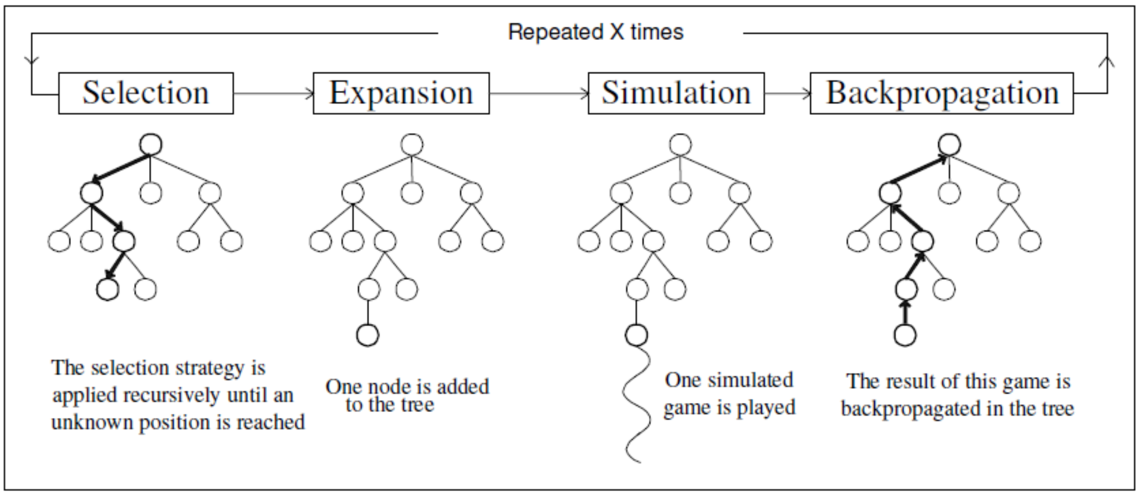

Repeated X times
Selection
Expansion
Simulation
Backpropagation
The selection strategy is applied recursively until an unknown position is reached
One node is added to the tree
One simulated game is played
The result of this game is backpropagated in the tree

Figure 2.1: Outline of Monte-Carlo Tree Search (Chaslot, 2010).

For HIVE, the first two conditions are satisfied. It is clearly defined how to win or lose and, furthermore, both players can see all pieces in- and outside the field at every time during the game. The last point is be a problem as a complete randomly played game can consist of up to several thousand moves. Therefore, for HIVE it was necessary to stop unfinished games after a set amount of turns. If a game is not finished until then it is counted as draw. Further information is given in Chapter 3.

The Monte-Carlo tree search consists of four major steps which are presented in Figure 2.1. Those are Selection, Expansion, Simulation, and Backpropagation and are repeated as long as the calculation time given to the artificial intelligence allows. They will be discussed more precisely in the next sections. During the calculation time, a tree is built. Each node i in this tree represents a possible future situation. The root depicts the current situation on the field. In every node at least the following two values are stored: the current value v_i of the represented position, for instance a sum of all loses and wins so far, and a visit count n_i which is increased every time the selection step finds a path through this node.

The pseudo code of MCTS is given in Figure 2.1 as it can be found in Chaslot (2010). T denotes the set of all nodes in the expanded tree. $Selection(NodeN)$ is the procedure that represents the selection step and returns the child of a current node N. The procedure $Expansion(NodeN)$ expands a node and adds its children to the tree if it is not expanded yet. $Simulation(NodeN)$ simulates a game from the given node to the end and returns the result as chosen by the programmer, e.g., $R \in \{-1, 0, 1\}$. $Backpropagation(NodeN, R)$ is a procedure that updates the value of a node N using the result R of the simulated game. $\mathcal{N}_c(NodeN)$ is the list of children of a node N. Better explanation of all four steps can be found in the following sections.

Algorithm 2.1 Pseudo-code for MCTS

```
Require: $root\_node$
  while ($has\_time$) do
    $current\_node \leftarrow root\_node$
    while ($current\_node \in T$) do
      $last\_node \leftarrow current\_node$
      $current\_node \leftarrow Selection(current\_node)$
    end while
    $last\_node \leftarrow Expansion(last\_node)$
    $R \leftarrow Simualtion(last\_node)$
    $current\_node \leftarrow last\_node$
    while ($current\_node \in T$) do
      $Backpropagation(current\_node, R)$
      $current\_node \leftarrow current\_node.parent$
    end while
  end while
  return $best\_move = argmax_{N \in \mathcal{N}_c}(root\_node)$
```

2.1.1 Selection

During the selection ste,p a path through the already existing tree is traced. At the beginning the current node is the root. A node N of the current nodes children is chosen. The chosen node replaces the current node. This is applied recursively on the tree until a leaf L is reached. A leaf could mean the game is finished or if the game is not, the possibility of expanding the tree further. A balance between exploitation (i.e., looking to find the best results so far) and exploration (i.e., evaluating new and undetected or uncertain moves) needs to be found. To achieve this goal a good *selection strategy* is important. One of these strategies is UCT which will be described later in this chapter. The simplest way is to select randomly, but other selection strategies can be used.

2.1.2 Expansion

In the expansion step, new nodes are added to the tree. According to Chaslot (2010, p.22), the most popular *expansion strategy* is to expand one node per simulated game. This is a simple and easy to implement method that does not occupy to much memory. Other suggested strategies are to set certain depths after node or only expand a node i if a certain number n_i of simulations have been made through the chosen node.

Generally, the effect on the result of the game is small and therefore the strategy is best chosen in terms of resources.

2.1.3 Simulation

The next step is the simulation step and also called play-out. Here, a game is played until the end or interrupted. There are different possible *simulation strategies*. The most common one is to choose moves randomly until the game is over. Other strategies can improve the level of performance significantly. This could be, for instance, using knowledge like in evaluation function based methods.

It is a trade-off between better performance and calculation time to add knowledge-based decision to the procedure of choosing the next move in a simulation strategy. Knowledge-based decision can increase the level of playing by choosing more reliable moves and more accurate strategies, but it is also expensive in memory and therefore the number of simulations will decrease. Another trade-off is the fact that completely random simulations are cheap in memory but often weak in playing and choosing unrealistic moves. On the other hand, too deterministic strategies, which often choose the same moves are not the best choice either. In this case a balance between exploration and exploitation has to be found.

2.1.4 Backpropagation

In this step, the result of a simulation is propagated from the leaf L through all visited nodes to the root node. These results are counted in $\boldsymbol{R}_k$. This could mean, for instance:

$$R_k = \begin{cases} +1, & \text{if the game is won,} \\ -1, & \text{if the game is lost,} \\ +/-0, & \text{if game ends in draw.} \end{cases}$$

$\boldsymbol{R}_k$ plays part in an end value v_L for each node. v_L is the value of a leaf L in which $\boldsymbol{R}_k$ is used. The simplest strategy is to sum all results. This leads to the simple function $v_L = \boldsymbol{R}_k$. v_L is then propagated from the leaf L to the root.

There are several other *backpropagation strategies*.

- **Average:** This strategy takes the plain average of the results of all simulated games made through this node: $v_L = (\sum_k R_k/n_L)$ (Chaslot, 2010).
- **Max:** Max tries to model the noise of the value of a node to get better results.

- **Informed Average:** Informed Average converges faster to a best value but may not play as well as Average.
- **Mix:** This strategy considers a weight to the average of a node.

2.1.5 Selection of Final Move

After the four main steps a final candiate move needs to be chosen. For that several *selection strategies* are possible:

- **Max child** The child with the highest value v_i is chosen.
- **Robust child** The child with the highest visit count n_i is chosen.
- **Robust-max child** The combination of highest visit count n_i and highest value v_i is chosen. In case there is no robust child more simulations need to be played.
- **Secure child** The secure child maximizes a lower confidence bound, i.e., which maximizes the quantity $v_i + \frac{A}{\sqrt{n_i}}$, where A is a parameter, v_i is the nodes value, and n_i is the nodes visit count (Chaslot, 2010, p.25).

According to Chaslot (2010, p.25), there is no significant difference in performance between the methods mentioned above.

2.2 UCT - Upper Confidence Bounds Applied to Trees

Chaslot (2010) describes several selection strategies. One of these strategies is UCT and will be further explained in this section.

UCT is an easy to implement method to select a candidate node during selection step. Originally, it was developed for the Multi-Armed Bandit (MAB) problem. The MAB problem pictures a scenario with a gambling device and a player who tries to maximize the reward he gets from it. At each step of the gambling the player has to decide which of the N arms of the device to choose for this aim. The reward usually follows a stochastic distribution. The selection in MCTS can be seen as a MAB problem for a node. The moves are equivalent to the arms of the device and the problem is to decide which move to choose. The artificial intelligence needs to find the optimal move knowing the past results. UCT selects the child k of the node p which best satisfies the following formula:

$$k \in argmax_{i \in I} \left(v_i + C\sqrt{\frac{\ln n_p}{n_i}} \right) \tag{2.1}$$

where I is the set of nodes reachable from the current node p, v_i is the value of the node i, n_i is the visit count of i, and n_p is the visit count of p. C is a constant and needs to be tuned experimentally. UCT is used to find a balance between exploitation and exploration (Kocsis and Szepesvári, 2006). C also influences the character of an AI. If C has a low value the AI plays defensive. The higher it get the more offensive the AI plays. Other strategies related or developed from UCT exist but are not further discussed here. It should be noted that all selection strategies are game-independent and do not use any kind of domain knowledge to evaluate a move.

In this chapter, Monte-Carlo tree search was presented. Furthermore, upper confidence bound applied to trees was introduced. UCT can improve the performance of an artificial intelligence compared to an AI only using the basic algorithm. The following chapter will focus on how the MCTS algorithm as well as UCT is implemented and used for the game HIVE.

Chapter 3

Design and Implementation

In this chapter, the implementation of HIVE will briefly be introduced. After that it is shown how the different types of AIs work and the implementations of the different strategies will be described.

3.1 Concept of Implementation

At the time this chapter was written and the game was implemented, to my knowledge no other working implementation of HIVE existed. Therefore, a complete game was implemented which can be used by two human players, one artificial intelligence and a human player, or two AIs. The GUI of the game can be turned off to make gameplay for two AIs faster. It is also possible to save and load games.

The strategies used by the resulting artificial intelligence are three different variations of the simple Monte-Carlo tree search, upper confidence bound applied to trees with different constants C, and a random AI for testing purpose.

3.2 Implementation of the Game

The game is implemented using the Model View Control (MVC) pattern. The game consists of four parts: the model, the view, the controller, and the artificial intelligence(s) which will be described in the next section. A class diagram for the game can be found in Figure 3.1.

3.2.1 Model

The model is representing all information about the game. The field is saved as a 2D-array (Field). If a single field in the array is used, the information of an insect (Node) is stored there. Around every piece that is not only surrounded by other insects the neighbor fields are completed with nodes representing empty fields. Therefore, the field is dynamic. The empty fields are needed for the GUI so that a human player can click on a field where he wants to place a token. Only fields around insects are needed because pieces are not allowed to sit alone. There also are lists for all unplaced pieces and pieces which are underneath beetles. The insects use a strategy pattern to calculate the individual ways of making moves. According to the fact that HIVE does not have a field with borders, the field can be shifted up, down, right, or left if pieces are getting to close to one of the borders of the GUI. In the model, a given move is evaluated considering if it is a possible move or not.

3.2.2 Controller

The controller is responsible for the communication between artificial intelligence, model, and view. If the artificial intelligence or the human player makes a move it is first sent to the controller and from there to the model. In the model the game representation is updated with the new move. At last, the controller calls an update for the view and all AIs. Figure 3.2 shows a sequence diagram for a 2-ply. This means every player makes one move.

3.2.3 View

The View represents the GUI for a human player. It is an observer pattern which gets notified every time a move was played. If one or both players are human it has also a listener to get the human players moves. The player has to click on a piece and then gets all candidate moves for this token marked.
The painting of the insects types and the activity status, which tells if a token was clicked or not, are implemented with a strategy pattern.

3.3 Implementation of the Artificial Intelligence

In this section it will be explained how the AIs are implemented and what are the differences between them.

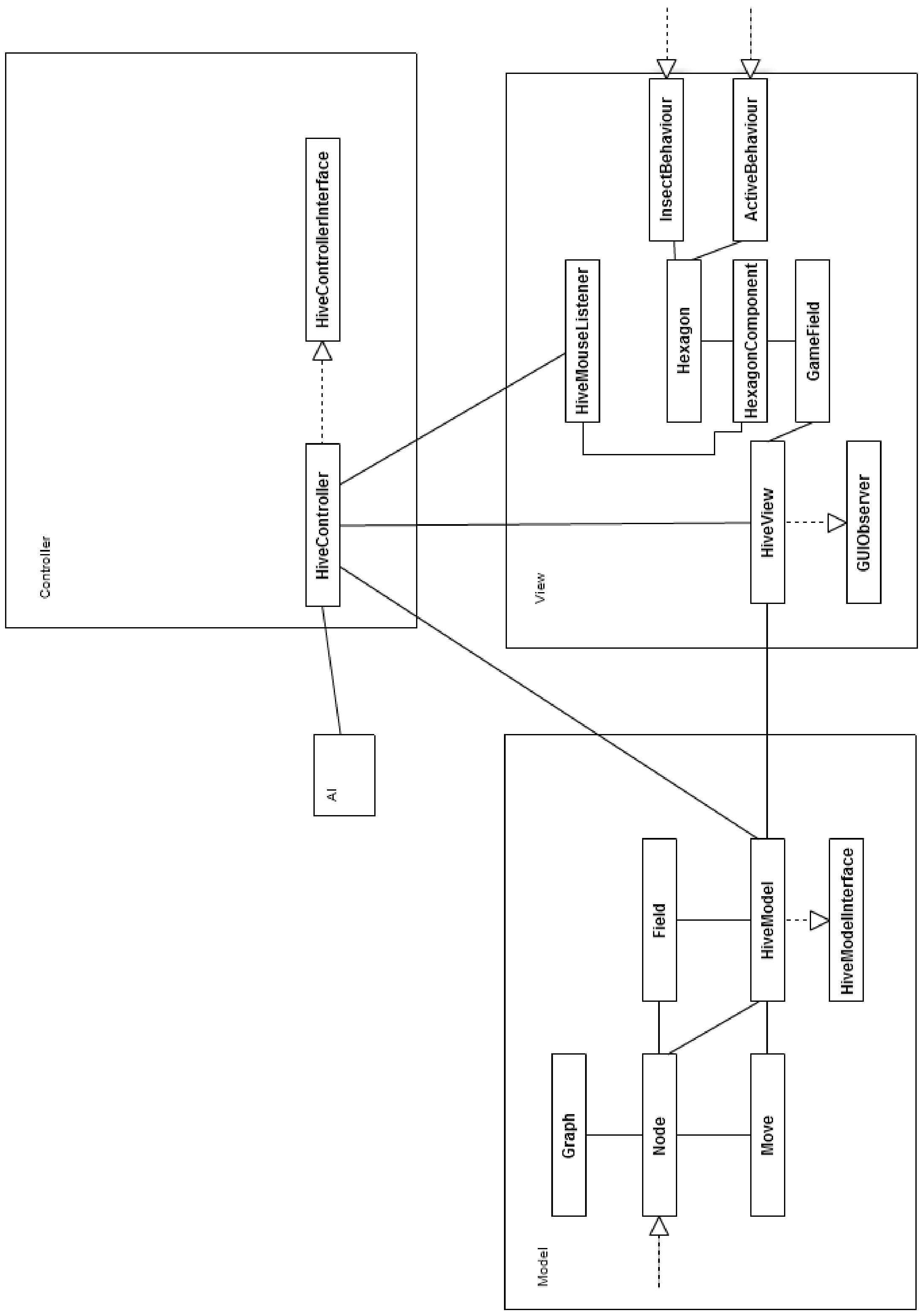


Figure 3.1: Class diagram of the game Hive

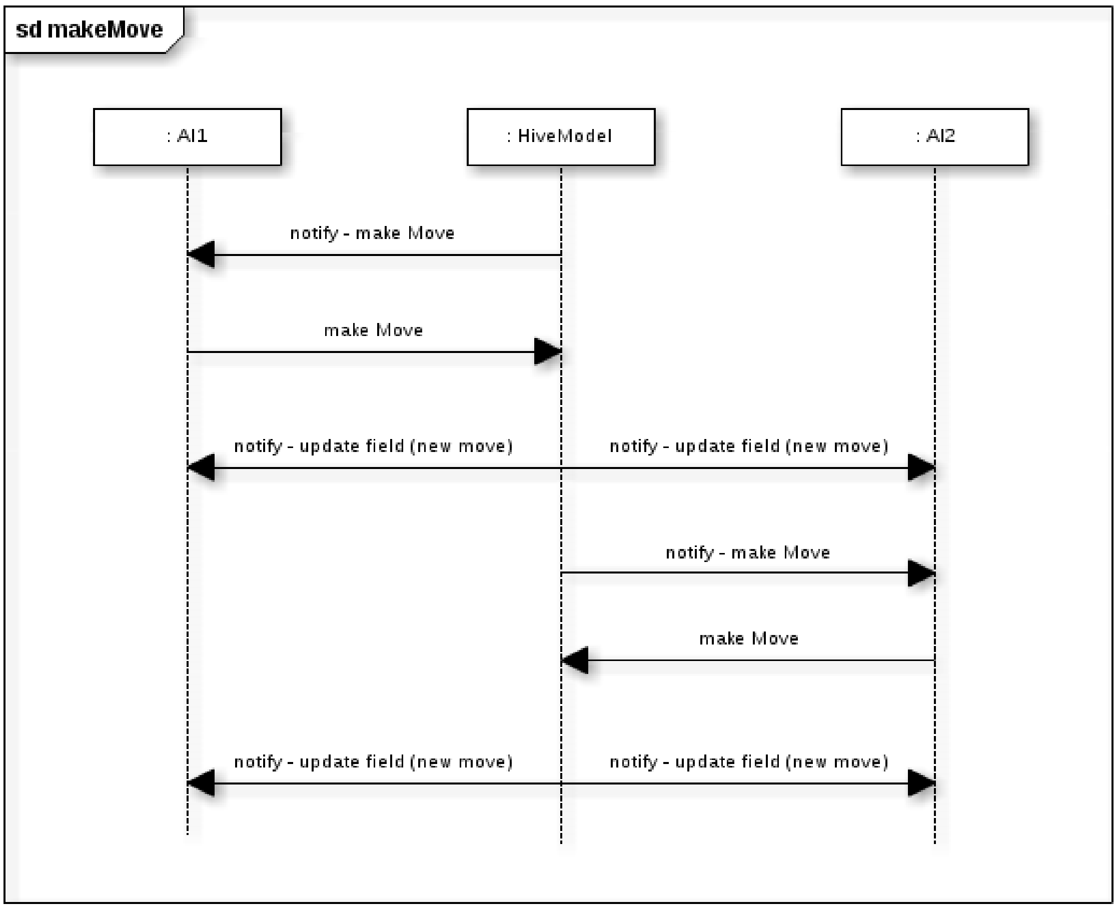

sd makeMove
: AI1
: HiveModel
: AI2
notify - make Move
make Move
notify - update field (new move)
notify - update field (new move)
notify - make Move
make Move
notify - update field (new move)
notify - update field (new move)

Figure 3.2: Sequence diagram for 2-ply

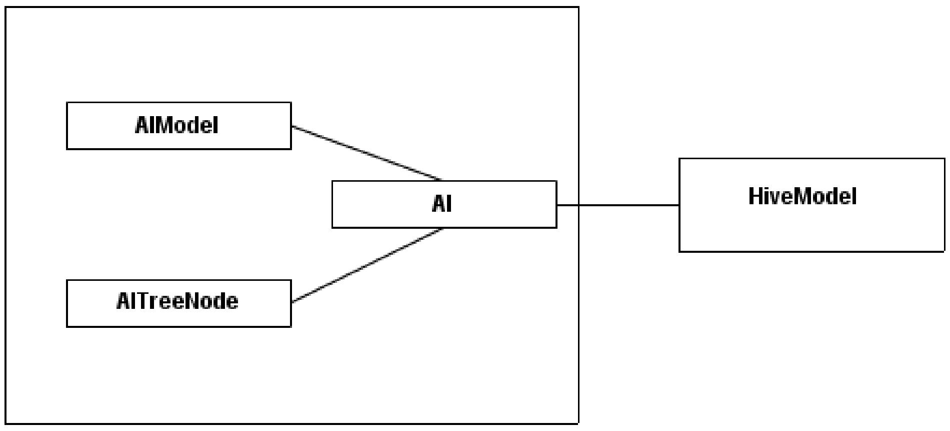


Figure 3.3: Class diagram of an AI

There are three implemented types of artificial intelligences.

- Monte-Carlo tree search,
- upper confidence bound applied to trees,
- and a random move selector.

The random move selector chooses one of the candidate moves randomly. There are two different random AIs: one is completely random and the second one recognizes win in one turn. It was decided to use the second type to shorten simulated games. Completely random AIs play games with an average of more than 1000 turns if both player are random players. Two AIs which can recognize win in one turn only have an average of less than 500 turns.

The artificial intelligence is an observer of the game model. The procedure of a 2-ply can be viewed in Figure 3.2. The first player receives a notification every time a move of the artificial intelligence is expected. This move is sent to the HiveModel. After checking if the given move is valid a notification for a newly made move is transmitted to both players. This is repeated for the second AI. If one or both players are human the notifications are sent to the GUI. In the sequence diagram the actors AI1 and AI2 are replaced by one GUI actor.

A class diagram for a common AI can be seen in Figure 3.3. One of the classes, the AIModel, is representing the game as the model does in the game itselfe. There all the calculation takes place. It is reset to the current situation after each simulation of a game.

Additionally, there is an AI class which takes care of the resetting of the model and the calculation time. Each game is simulated randomly. Therefore, due to an open game field a game can have several thousand turns. To avoid this, an unfinished game is usually ended after 170 turns and considered a draw. Further information can be found in Section 3.4. The time the artificial intelligence gets for calculating the best move is not a set time interval. At first, it is determined how many candidate moves exist for the root. This number is multiplied by a certain constant. This makes sure the artificial intelligence can calculate enough games to get a representative result.

The last class is representing the tree (TreeNode) built during computation where the actual simulations proceed. The doGame() method is only used for the root and initiates the computation. Then, the tree is expanded and built depending on the chosen type of artificial intelligence.
In the next section, it will first be discussed how the MCTS algorithm was implemented and afterwards how the UCT strategy can be added.

3.3.1 Monte-Carlo Tree Search

Three different variations of MCTS were considered in this work:

- Basic MCTS
- MCTS-A1
- MCTS-A2

Except for the backpropagation step, they are equivalent. In the selection step, nodes are chosen randomly until a leaf is reached and MCTS switches to the expansion step. There all children of a node are added to the tree and one of them is chosen to be the new current node. During the simulation step, a token on the field is selected randomly and for this token all candidate moves are calculated and available to be new current node. If a token cannot be moved, another one is chosen. This happens recursively until the end of a game is reached. This method is faster than to find candidate moves for all tokens of a player and then choose a move.

The basic version of the MCTS implementations counts the results of all simulated games as it was explained in Chapter 2:

$$R_k = \begin{cases} +1, & \text{if the game is won,} \\ -1, & \text{if the game is lost,} \\ +/-0, & \text{if game ends in draw.} \end{cases}$$

At the end the node with the highest value is selected as next move. MCTS-A1 and MCTS-A2 use both different types of the average function. MCTS-A1 calculates the average or value v_L of a node L by dividing the number of games won R_k by the number of all simulated games n_{all} during a simulation: $v_L = \frac{R_k}{n_{all}}$. For MCTS-A2 v_L was calculated by dividing R_k by the number of all simulated games n_L which went through the particular node L: $v_L = \frac{R_k}{n_L}$. The second way is suggested by Chaslot (2010) and also said to perform better than the backpropagation used in the simple MCTS.

A comparison between these types of AI can be found in Chapter 4 where the evaluation is discussed.

3.3.2 Upper Confidence Bound Applied to Trees

The artificial intelligence using upper confidence bound applied to trees is a variation of the previously described MCTS. The difference can be found in the selection step. Here the Formula 2.1 is used. There are different possibilities to apply it:

- every move is chosen according to the formula,
- in every node with an even/odd number of visits the next move is selected according to the formula,
- Formula 2.1 is used after a threshold T of visits of a node is passed, before that another strategy, like random selection, is used.

As described by Kocsis and Szepesvári (2006), UCT is used to balance exploration and exploitation. Threshold T determines if a move is chosen randomly or if UCT is applied. Every time a node is reached which has not met this threshold, yet, the next move is chosen randomly. Since the number of simulated games is relatively low, the used threshold was $T = 1$ for one AI (for instance the number of simulations is $20\cdot$ candidate moves). Another AI selects randomly at every node with an even visit count and using UCT for a node with an odd visit count. This AI is used later in the experiments. For this version of UCT the constant C in Formula 2.1 was changed in different experiments (i.e., $C = 0.5, 50, 100, 200, 500, 1000)$. A second type of UCT which was tested in an experiment only uses Formula 2.1. This AI is called UCT_C.

3.4 Time Management

After testing the AI, it turned out it takes more time to calculate than expected. The first thought might be to give every AI a set amount of time to calculate the next move. But during the calculation, time differs widely from turn to turn. This derives from the fact that games are simulated randomly. Since HIVE does not have any rules which terminate the game eventually (i.e., it has no number of maximum turns possible for a game), the computing time might be shorter if unfinished games are interrupted eventually. The interrupted game will be dismissed and counted as draw. A balance between finishing enough games and not wasting too much time for unfinished games needs to be found. To see which number of games before interruption leads to the shortest calculation time, a few test games were played. The results can be seen in Table 3.1. According to these results the shortest computing time lies between 100 and 200 moves before interruptions. For that reason 170 turns were chosen for all further tests with for types of AIs.

N_{ip}	Average time per turn
100	32.82 s
170	27.25 s
200	35.27 s
300	36.06 s
no interruption	45.71 s

Table 3.1: Average computing time per turn

Chapter 4

Evaluation

In this chapter, the results of the game-play between all implemented types of AIs will be discussed. For that purpose, every AI had to play at least 50 matches against another AI with each color. This makes at least 100 matches for every AI. In HIVE, usually the youngest player begins independent of his color. For the evaluation the white player starts like in chess. This rule makes sure that every artificial intelligence can begin in some matches and be second in others.

4.1 Tactics and Game Development

In this section, typical moves AIs often do and the way how games usually develop will be shown. A common characteristic, especially for weaker AIs, is to bring all pieces into the field before they start to move any of them. This often happens in a set order: the first piece in the field is the queen bee, followed by spiders, grasshoppers, beetles, and ants in the mentioned order. This usually leads to patterns on the field where the AIs cannot make many moves. As can be seen in Figure 4.1, both AIs have many of their stones in the field but comparatively less candidate moves. Only the marked pieces at the tips of the long chains can be moved without parting the hive. Therefore, there are only three different stones for each player depending who's turn it is. This opening strategy is dangerous if played against a player who starts to move pieces towards the opponents bee as soon as possible. A human player at beginners level is able to win with less than 20 turns in this situation.

On the other hand, the opening combination bee - spider - spider is suggested by Hiv (2012) as a favorite opening combination. The opening should happen in a V shaped formation as can be seen in Figure 4.2. An order like this makes an early blocking of the opponents bee possible. The opening formation of the AI is usually not V shaped but in

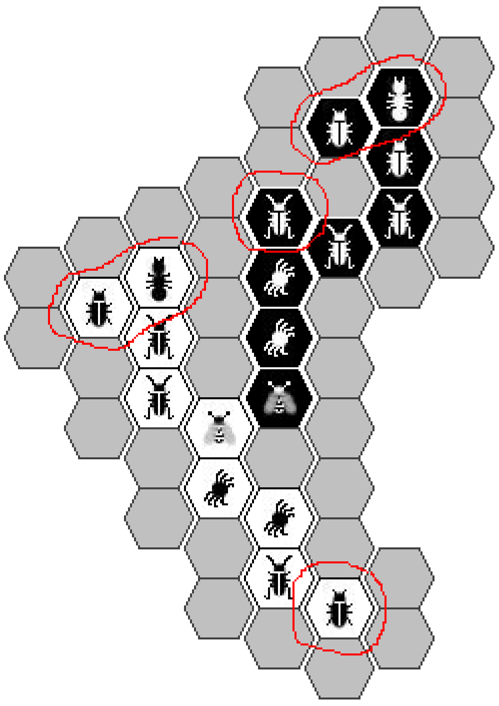


Figure 4.1: Opening situation (MCTS vs. MCTS)

a few cases spiders are moved towards the opponents bee and used to bring stones into the field on the opponents side.

4.2 Gameplay

In this section, there is a brief introduction to the experiments and the involved AIs, as well as tables to present the results of the game-plays. Games can be won, lost, end in draw or be interrupted. If a game takes more than a set time, it is interrupted because often games result in repeating loops. If not explicitly mentioned the number of games N_p played per simulation is $20 \cdot M_p$ where M_p is the number of candidate moves for a player. AIs with $N_p = 20 \cdot M_p$ are relatively weak but AIs with an higher amount of time needed too much time to get enough simulations for complete games.

4.2.1 Random vs. MCTS

The first experiment is to see whether the basic MCTS is better than a random player. All moves are chosen randomly if there is the opportunity to win in one turn, the random AI can recognize it. Assumptions would be that a MCTS based AI is stronger than

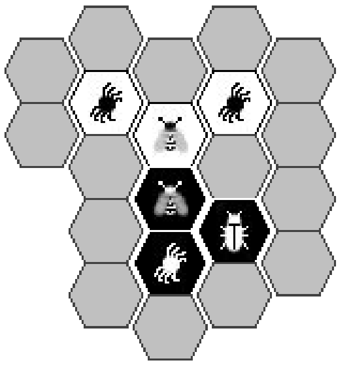


Figure 4.2: V shape for player white

White	MCTS wins	Random wins	Draw	Interrupted
MCTS	76.9%	0.0%	0.0%	23.1%
Random	85.%4	3.0%	0.0%	11.6%
Average wins	81.1%	1.5%	0.0%	17.4%

Table 4.1: MCTS vs. Random (73 games)

randomly played moves. As Table 4.1 shows this is true. 81.1 % games were won by MCTS. So the MCTS is definitely better than a random player and finishes a non interrupted game with the average of 36.7 turns. 17.8% of all games needed to be interrupted because of loops.

4.2.2 Random vs. UCT

The second experiment is random against UCT ($C = 100$). UCT won 83.7 % of all games. As it can be seen in Table 4.2 there still have been 15.0 % of interrupted games. UCT needs an average number of turns of 34.8 % to finish a game against a random AI. Therefore, this experiment was the one which needed the least turns of all experiments to be finished. Both experiments involving a random AI show that the two strategies, MCTS and UCT, are definitely getting better results than randomly chosen moves.

4.2.3 MCTS vs. MCTS

In this experiment, two MTCS-AIs played against each other. As described earlier, there are three different types of MCTS. For the experiment the basic MCTS played against MCTS-A1 and MCTS-A2, The results of these experiments can be seen in both tables.

White	UCT wins	Random wins	Draw	Interrupted
UCT	79.2%	2.7%	0.0%	18.1%
Random	88.1%	0.0%	0.0%	11.9%
Average wins	83.7%	1.3%	0.0%	15.0%

Table 4.2: UCT vs. Random (61 games)

White	MCTS wins	MCTS-A1 wins	Draw	Interrupted
MCTS	48.3%	34.7%	0.0%	17.0%
MCTS-A1	51.8%	31.8%	1.0%	15.4%
Average wins	50.1%	33.2%	0.5%	16.2%

Table 4.3: MCTS vs. MCTS-A1 (201 games)

The basic MCTS wins 50.1 % of all games whereas MCTS-A1 only wins 33.2 %. 16.2 % of all parties were interrupted due to a time out. In general, a none interrupted game ends after around 46.43 turns.

, in the second experiment the basic MCTS performs twice as good as MCTS-A2. The basic MCTS wins 40.9 % of all games, whereas the MCTS-A2 only 20.7%. The average number of turns is 54.66. For HIVE the average backpropagation seems not to work better than simple counting. The basic MCTS performs better than a MCTS using the average. Therefore, for all following games the better performing MCTS backpropagation is used.

4.2.4 MCTS vs. UCT

In this type of experiment MCTS and UCT played against each other in different series. Those series varied in the value of the constant C in the UCT. If $C = 0$, UCT is equal to the basic MCTS. According to Chaslot (2010), the *character* of an AI is influenced by this constant. The Hive-AI plays defensive for a small C and gets more aggressive if C is increased. To test which value of C achieves the best performance, the following experiments have been made. Six series each with a different value C (i.e., $C = 0.5, 50, 100, 200, 500, 1000$) took place. The results are presented in following Tables: 4.5 - 4.10, respectively. In four of these six experiments UCT played better than MCTS which can be seen in Figure 4.3. Exceptions are $C = 0.5$ and $C = 1000$. In the first case, players with starting color black are slightly better, whereas in the second case white players are about 8 % better which makes about a third more winnings. For $C = 0.5$, UCT plays too defensive and does not use opportunities to attack the opponent.

White	MCTS wins	MCTS-A2 wins	Draw	Interrupted
MCTS	40.1%	22.4%	0.0%	37.5%
MCTS-A2	41.7%	18.9%	1.2%	38.2%
Average wins	40.9%	20.7%	0.6%	37.8%

Table 4.4: MCTS vs. MCTS-A2 (191 games)

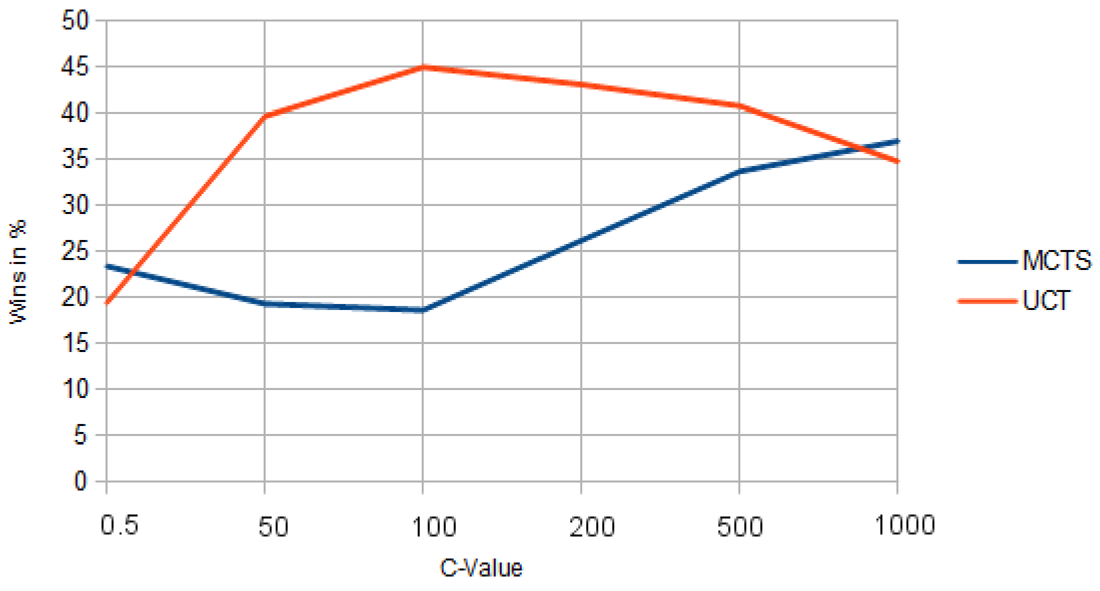


Figure 4.3: Winning rate in % of MCTS and UCT

For $C = 1000$, the AI is too aggressive and therefore does not defend its bee enough. The best results for UCT are achieved by $C = 100$ where UCT wins more than twice as much games as MCTS.

The average number of turns until a game is finished, interruptions not considered, are 55.7 turns for $C = 0.5$, 46.25 turns for $C = 50$, 42.8 turns for $C = 100$, 44.77 turns for $C = 200$, 48.7 turns for $C = 500$, and 50.29 turns for $C = 100$. If $C = 100$ UCT wins 45.0% of all games. The smaller or higher C gets, the less of all games were won by UCT. Because of its performance and its average of 42.4 turns for a finished game, UCT with $C = 100$ is the best player. Here the balance between aggressive and defensive play is found.

White	MCTS wins	UCT wins	Draw	Interrupted
MCTS	19.8%	21.6%	2.8%	56.3%
UCT	27.0%	17.2%	1.0%	54.0%
Average wins	23.4%	19.4%	1.9%	55.3%

Table 4.5: MCTS vs. UCT ($C = 0.5$, 222 games)

White	MCTS wins	UCT wins	Draw	Interrupted
MCTS	17.5%	40.0%	1.2%	41.3%
UCT	20.6%	38.6%	1.2%	37.4%
Average wins	19.3%	39.3%	1.2%	40.2%

Table 4.6: MCTS vs. UCT ($C = 50$, 166 games)

White	MCTS wins	UCT wins	Draw	Interrupted
MCTS	19.8%	47.0%	2.2%	31.0%
UCT	17.2%	43.0%	3.8%	36.0%
Average wins	18.6%	45.0%	2.8%	33.7%

Table 4.7: MCTS vs. UCT ($C = 100$, 151 games)

White	MCTS wins	UCT wins	Draw	Interrupted
MCTS	27.6%	44.6%	4.7%	23.1%
UCTs	25.0%	41.7%	2.9%	30.4%
Average wins	26.3%	43.2%	3.8%	26.7%

Table 4.8: MCTS vs. UCT ($C = 200$, 137 games)

White	MCTS wins	UCT wins	Draw	Interrupted
MCTS	31.9%	41.1%	3.0%	24.0%
UCT	34.3%	40.5%	3.0%	22.2%
Average wins	33.7%	40.8%	3.0%	23.1%

Table 4.9: MCTS vs. UCT ($C = 500$, 130 games)

White	MCTS wins	UCT wins	Draw	Interrupted
MCTS	45.3%	27.5%	3.0%	24.2%
UCT	27.6%	40.0%	1.4%	25.6%
Average wins	37.5%	33.8%	2.2%	26.5%

Table 4.10: MCTS vs. UCT ($C = 1000$, 135 games)

White	UCT wins	UCT_C wins	Draw	Interrupted
UCT	35.1%	35.1%	3.8%	26.0%
UCT_C	40.6%	33.6%	4.2%	21.6%
Average wins	37.8%	34.4%	4.0%	23.8%

Table 4.11: UCT vs. UCT_C (176 games)

p	draw MCTS (%)	draw UCT (%)
20	15	2.5
40	32.5	30
80	65	65
200	90	95

Table 4.12: Decision between lose and draw for black - results

4.2.5 UCT vs. UCT

In the last experiment, two UCT-AIs played against each other. The only difference can be found in the amount of selections using the UCT formula. The usual UCT-AI used in the game plays before has played every even game randomly as the basic MCTS does and in the rest of the simulations using the UCT formula for selecting moves. The results of this experiment is shown in Table 4.11. The usual UCT won 37.8 % and is therefore slightly better than the AI using the UCT strategy for every selection.

4.3 Game Situations

In this section, a collection of situations in games are shown. Two types of AI, MCTS and UCT with $C = 100$ were tested 40 times each per situation and different p to see how they react and if the reaction is reliable. The constant p is used to compute the number of games played per simulation. The following equation describes the number of games during a simulation. $N_p = p \cdot M_p$, where M_p is the number of candidate moves and N_p the number of played games during the simulation.

4.3.1 Decision Between Lose and Draw

Figure 4.4 shows a situation where black can only move the red circled ant. In general, black can place this token anywhere around the hive but every other position than 1 or 2 leads to lose the game for black. If the ant is placed at position 1 the game ends in draw,

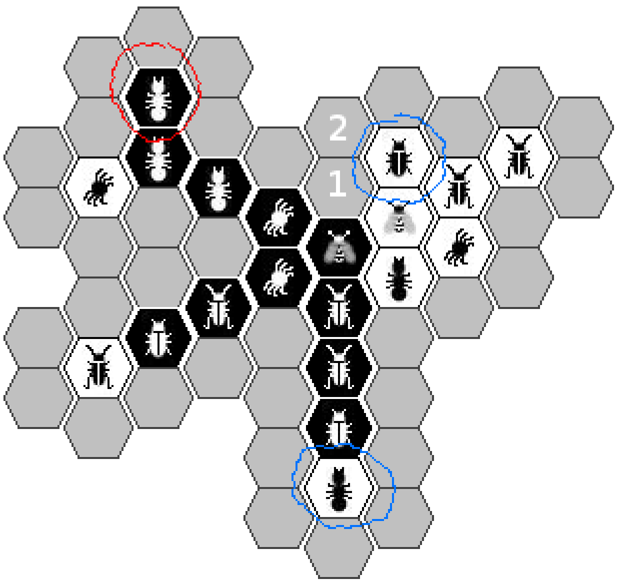


Figure 4.4: Decision between lose and draw for black

if placed at position 2 no white piece can move to position 1 and is therefore saved. After that in the next turn black has to move the ant again which also leads to a certain lose when white moves its beetle, which borders position 1 and 2, to position 1. If the ant is placed anywhere else around the beetle so the beetle cannot move, white will most probably end the game with the blue circled ant in draw. Therefore, for black the best and safest move would be to end the game in draw with moving its ant to position 1.

In Table 4.12, the results for the games can be seen. In this situation MCTS performs more than 10% better than UCT with $p = 20$. Apart from the fact that nodes are not visited often enough to get reliable results, the value v of the node is always 0.0 as it only results in draw. So its chance to be visited gets smaller after every visit, whereas in MCTS the chances are the same throughout the whole simulation. If UCT has more games than $p = 20$ to play, it tends to get a "favorite move" which is visited more often at the end and usually the one chosen as best move.

4.3.2 Win in One Turn

In this section, a situation is tested which can be ended with win for black in one turn. It can be seen in Figure 4.5. Black has two tokens to use to end the game. These are the two red circled ones. The blue circled ant can also be moved but cannot finish the game. During all simulations the AIs never used the ant. As shown in Table 4.13, both

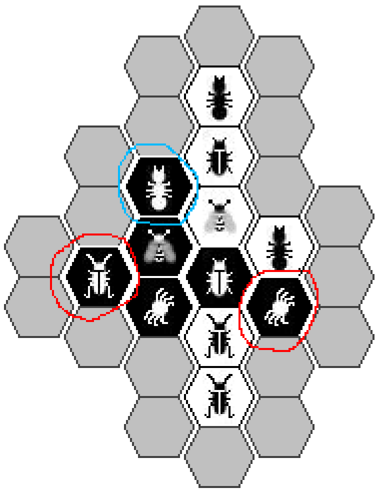


Figure 4.5: Win in one turn for black

p	win MCTS (%)	win UCT (%)
20	97.5	92.5
40	100.0	97.5
80	100.0	100.0
200	100.0	100.0

Table 4.13: Win in one turn for black - results

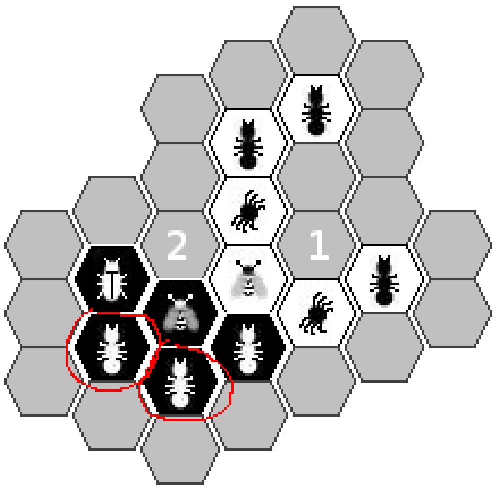


Figure 4.6: Win in two turns for black

p	lose MCTS (%)	lose UCT (%)
20	85	100.0
40	67.5	100.0
80	42.5	100.0
200	12.5	100.0

Table 4.14: Win in one turn for black - results

ALs perform mostly identical. Win in one turn is recognized in more than 90 % of the cases even for the AIs with the lowest number of simulations.

4.3.3 Win in Two Turns and Recognizing Mate

The last situation that was tested is presented in Figure 4.6. Black needs to move one stone, so white cannot set it mate in one turn. Therefore, black needs to set one of the red circled ants to position 1. This move removes one token around the own bee and encircles the opponents bee. Black could also use the other ant but it would not bringing black closer to win. Most probably white would put one of its movable ants to the position left by black or try to make the black beetle or the second red circled ant immovable. If white places a stone directly on the field above the black beetle black cannot set white mate. However, this move was never made by any AI.

The results can be seen in Table 4.14. Here UCT plays incredibly bad. It always places a piece on position 2 and makes it possible for white to win. This results from

$C = 100$. If C is set lower (i.e., $C = 0.5$), it starts to block white pieces. According to the last situation, it might help UCT to get a knowledge-based adaption to tell UCT the value of C and thus decide if the AI needs to play more defensively or more aggressive. The results in this chapter show that in general, if tuned right, UCT performs better than MCTS. However, in certain situations UCT is defeated by MCTS. Increasing the number of simulations can improve the performance of both AIs.

In this section, the evaluation of all AIs has been presented. In the first part, whole matches were played, whereas later single situations were tested. In general, UCT performs better in whole games. However, in some situations MCTS can outperform UCT. In Chapter 5, the problem statement from the beginning will be answered and future work is suggested.

Chapter 5

Conclusion

In this chapter answers to the research questions with summarizing important points and conclusions will be discussed. Finally, suggestions for the future are given.

5.1 Answering the Research Questions

Can Monte-Carlo tree search be applied to the game HIVE, and how do different Monte-Carlo tree search variations perform? Monte-Carlo tree search and upper confidence bound can be successfully applied to HIVE and is sufficient to play the game. However, performance is a problem and compared to a human player quite poor. Therefore, improvements are necessary. That might be possible by increasing the number of simulated games or using domain knowledge. Different types of applying domain knowledge are discussed in Section 5.2 as well as the approach of using Minimax.

Which of the two approaches of MCTS and UCT turns out better? As seen in the previous chapter, both AIs played successfully against a random AI. Furthermore, it showed that UCT performs better than Monte-Carlo tree search. For that, UCT needs to be tuned right. Here the constant C plays an important role. As seen above, too small or too big values can make the UCT play worse than MCTS. For example, $C = 0.5$ makes UCT too defensive and thus, UCT loses many of the matches.

Why is one AI better than the other? Many factors are important to decide why one performs better than the other one. The most important difference between a well tuned UCT and MCTS is, as described by Kocsis and Szepesvári (2006), the fact that UCT usually has a better exploration-exploitation balance. With MCTS, every move has the same chances to be chosen for simulation. But UCT also considers the value v_i of a node i which tells if a node had already led to winning games or losing them.

If two AIs of the same type are playing against each other can minor variations improve the game play? This question can be answered with looking at the matches at Section 4.2.3. In this case, the backpropagation steps were changed and results show that some methods are clearly better than others. In other cases, like in Section 4.2.5, other changes can have almost no effect. There, a UCT which only uses the selection function if a node has an odd visit count played against an AI which selected all games with help of the selection function. The first type of UCT performs only slightly better.

5.2 Future Work

In this work, only the basic MCTS and UCT were used to see whether HIVE can be played by an artificial intelligence. Of course, there are several numbers of other approaches for that.

The most obvious one is the Minimax strategy. During the gameplays of the matchs between MCTS vs. UCT, a strategy of UCT could be observed: The AI reduced the opponents number of moves as much as possible, whereas its own number of moves was held high. Giving rewards for own moves and punishments for opponents moves might be a good start for a heuristic function. Also the type of insect which can make a move might be interesting to consider, as described by Yianni (2007) the ant is said to be the strongest stone. One ant can have more than 80 candidate moves, while a spider has a maximum of four.

According to Nijssen (2007), another way to improve a basic Monte-Carlo tree search is *preprocessing*. The basic MCTS treats all candidate moves equally. This means strategical good as well as bad moves are simulated with the same probability. A preprocessor analyzes all candidate moves and gives scores to all of them. The moves with the higher scores are preferred and are presumably the good moves. Usually, a preprocessor uses domain knowledge to score moves. There are two types of preprocessors:

- **Fixed Selectivity Preprocessor (FSP)**: A fixed number of moves N_s is chosen to pass on to the MCTS algorithm. This means only these moves can be done and only with them simulations are made. The bad ones are not considered as candidate moves any longer.

- **Variable Selectivity Preprocessor (VSP)**: This preprocessor passes on a variable number of moves. The VSP selects moves by comparing them to the average score of all candidate moves. At least a given percentage p_s of the average has to be chosen. If p_s is lowered more, moves are passed on.

Another strategy mentioned by Nijssen (2007) is *pseudo-random move generation*. Here, moves are scored as they are scored by preprocessing. The higher the score is, the more a move is preferred playing during simulation. More likely progressions are played more often than less likely ones. However, a big disadvantage is the computing time for the scores. Therefore, pseudo-random move generation is only used for the first N_d moves. After that the game is simulated by randomly choosing moves.

A handicap for computing time in HIVE is the fact that games need to be interrupted during simulation. For every finished game there are about 10 interrupted ones. This means a lot of time is not used to find a solution and therefore wasted. Using domain knowledge as suggested above might help to select moves more win-oriented and shorten simulated games. Finding a solution for this problem could save a lot of computing time and make the AI a lot faster and presumably performing better.

This thesis is foundation for an artificial intelligence for the game HIVE. As described in the last chapter, many possibilities for further improvements exist.

Bibliography

Hive -tips & tricks. http://www.gen42.com/hive-tips/70, 2012. Last visited: May 29, 2012. 22

Jean-Louis Cazaux. History of chess. http://history.chess.free.fr//history.htm, 2012. Last visited: June 28, 2012. 1

Guillaume Chaslot, Sander Bakkes, Istvan Szita, and Pieter Spronck. Monte-carlo tree search: A new framework for game ai. Association for the Advancement of Artificial Intelligence, 2008. 3

Guillaume Chaslot, Istvan Szita, and Pieter Spronck. Monte-carlo tree search in settlers of catan. In Proceedings of the Twelfth International Advances in Computer Games Conference (ACG'09), 2009. 2

Guillaume Maurice Jean-Bernard Chaslot. *Monte-Carlo Tree Search*. PhD thesis, University Maastricht, 2010. 8, 9, 10, 11, 12, 19, 25, 38

L. Stephen Coles. Computer chess: The drosophila of ai. http://www.drdobbs.com/parallel/computer-chess-the-drosophila-of-ai/184405171, 2012. Last visited: June 28, 2012. 1

Sylvain Gelly, Levente Kocsis, Marc Schoenauer, Michele Sebag, David Silver, Csaba Szepesvari, and Olivier Teytaud. The Grand Challenge of Computer Go: Monte Carlo Tree Search and Extensions. *Communications of the ACM*, 55, March 2012. 1, 2, 5

L. Kocsis and C. Szepesvári. Bandit based monte-carlo planning. *Machine Learning: ECML 2006*, pages 282–293, 2006. 13, 20, 33

Michael Lederman Littman. *Algorithms for Sequential Decision Making*. PhD thesis, Brown University, 1996. 1

J. A. M. Nijssen. Playing othello using monte carlo. Bachelor's thesis, Maastricht University, 2007. 2, 8, 34, 35

P.M. Pardalos. *Minimax and applications*, volume 4. Springer, 1995. 6

John Yianni. Hive - a game buzzing with possibilities. Huch! & friends, 2007. 3, 34

List of Figures

List of Tables

List of Algorithms

List of Abbreviations

AI	Artificial intelligence
MCTS	Monte-Carlo Tree Search
UCT	Upper Confidence bounds applied to Trees
FSP	Fixed Selectivity Preprocessor
VSP	Variable Selectivity Preprocessor