# Universiteit Leiden
# Opleiding Informatica

A Monte Carlo strategy for the game of Hive


Name: Nils Out

Date: 26/01/2017

1st supervisor: H.J. Hoogeboom
2nd supervisor: W.A. Kosters


BACHELOR THESIS


Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

# Abstract

Hive is a two-player strategy board game. Andreas Brytting and Johan Nygren implemented this game in 2013 and constructed an AI agent which can play the game based on the Minimax algorithm. In this thesis, we propose an adapted AI agent based on Pure Monte Carlo Game Search which played against the Minimax AI agent. Due to the characteristics of the game, Monte Carlo turned out to be an unsuitable AI algorithm for this game and could only beat the Minimax AI agent in the long-term with specific configurations.

# Contents

# 1 Introduction

In the past few years, many games are provided with an improved Artificial Intelligence (AI) agent based on Monte Carlo algorithms. On March 16th 2016, Google's AI agent for the game Go, called *AlphaGo*, defeated the number 1 ranked Go player in the world using a combination of Monte Carlo Tree Search and an Artificial Neural Network (deep learning) [1]. This breakthrough arouses interest in the question whether other boardgames will also benefit from the power of Monte Carlo.

Hive is a two-player strategy board game with twenty-two hexagonal pieces. Each player has eleven pieces in their own colour, representing different kinds of bugs that have their own way of moving on the board. The set of bugs consists of Ants, Spiders, Grasshoppers, Beetles and one Queen Bee. The goal of the game is to be the first to get the opponent's Queen completely surrounded.

Andreas Brytting and Johan Nygren implemented this game and constructed an AI agent based on the Minimax algorithm for their bachelor thesis at the KTH University in Stockholm in 2013 [3]. As an extension to their project, I added a new AI agent based on the Monte Carlo algorithm to find out whether the power of the algorithm also applies to the game of Hive. The usage of the same code made it possible to set up games between the Minimax AI agent and my Monte Carlo AI agent.

In Chapter 2, I provide more information about the game itself. I explain the general game rules and the characteristics of the different pieces for a better understanding of the game. Chapter 3 is a summary of Brytting and Nygren's thesis. It describes the defensive and aggressive strategies they construct and the evaluation algorithms that were used. Chapter 4 discusses my approach of the project, including a short attempt to implement the game by myself. After receiving the program code, my approach consisted of fixing bugs that I discovered while exploring the code of the implementation. Section 4.2 starts with more general information about the Monte Carlo algorithm, followed by a report about my modifications to the Monte Carlo algorithm and how I implemented the algorithm. Chapter 5 describes the two kinds of experiments I carried out with the AI agents. The first set of experiments consists of matches between my Monte Carlo agent and Brytting and Nygren's Minimax agent, testing their different variables. The second set of experiments are self-play games of the Monte Carlo agent with different configurations. Chapter 6 shows and discusses the results of these experiments, followed by the discussion of the conclusions from the results and the observations of the project.

## 1.1 Project goal

The project goal was to construct a new Artificial Intelligence agent for the game of Hive. This agent had to be based on the Monte Carlo algorithm. From the moment I had access to the code of Brytting and Nygren, a second goal was to beat the AI agents they designed and to improve the Monte Carlo agent so that it scored better on average in games against them. The goal was in fact to make a better AI agent than the Minimax agent of Brytting and Nygren.

# 2 Rules of the game

In order to get a better view on the subject, it is necessary to know and understand the game rules of Hive. This chapter describes the game rules per subject, based on the official game rules [2]. The images I included are made by Brytting and Nygren [3].

## 2.1 General gameplay

Hive is a turn-based strategy board game for two players. The board is defined by the pieces placed during the game, but the game starts with any possible empty horizontal surface. Each player has the same set of eleven hexagonal pieces which they can place and move during the game. After a while, the board will look like an unfinished honeycomb with multiple filled or empty hexes. The board is actually build upon an invisible hexagonal grid that can be used to count the positions to move to. Pieces are white or black and have coloured simplified images of bugs on top. There are different kinds of bugs which all behave in a particular way. One of these bugs is the Queen Bee and it is the object of the game to surround the opponent's Queen. The Queen is surrounded when all six sides are occupied by pieces, independent of their colour. There are furthermore three Grasshoppers, two Spiders, two Beetles and three Ants. Further descriptions on these pieces are made later in this chapter. The game ends when one of the Queens is completely surrounded, then that Queen's player loses the game. In the case that the piece that is moved last completes the surroundings of the both Queens at the same time, the game is a draw. A draw is also agreed if both players play the same two pieces (or the same two moves) over and over again, without any possibility of the stalemate being resolved.

## 2.2 Placing and moving of pieces

The player using the white pieces starts the game with the placement of one of his pieces in the middle of the surface. The next player places a black piece side-to-side to the white piece. From this moment, pieces from hand can only be placed next to pieces of the same colour, without touching an opponent's piece. Players can choose each turn between placing a piece or moving an already placed piece on the board. Once a piece is placed, it cannot be removed from the board in any way. Each player has to place his or her Queen within the first four turns. If the Queen is not placed after the third turn of a player, the player is forced to place the Queen at the fourth turn. Only after the Queen is placed, a player is able to move his pieces around the board and is not forced to place pieces any more. It is even possible to win the game without placing all pieces.

When a piece is moving, it has to stay connected with at least one side to a piece in the Hive at all times. It is allowed to move pieces to positions where they touch one or more opponent's pieces. If a player cannot move or place any piece, the opponent takes over the turn until the player is able to place or move again.

## 2.3 Restrictions on the movement of pieces

There are two general movement rules which must be respected at all time.

**One Hive rule**

The first one is the One Hive Rule. This rule states that the Hive, the string of all connected pieces, must be one component at all time. This means that a movement of a piece is forbidden when the Hive is split up during or after the movement. For an example, see the leftmost black Beetle in Figure 1. This piece cannot move, because the Grasshopper would otherwise be separated from the Hive.

**Freedom to Move**

The second rule is called the Freedom to Move rule. With this restriction, a piece cannot move to a position if it is not physically reachable. This means that a piece cannot be lifted or disrupt other pieces to reach such a position. The piece must be able to "slice" into his desired position, otherwise the move is forbidden. For an example, see Figure 1.

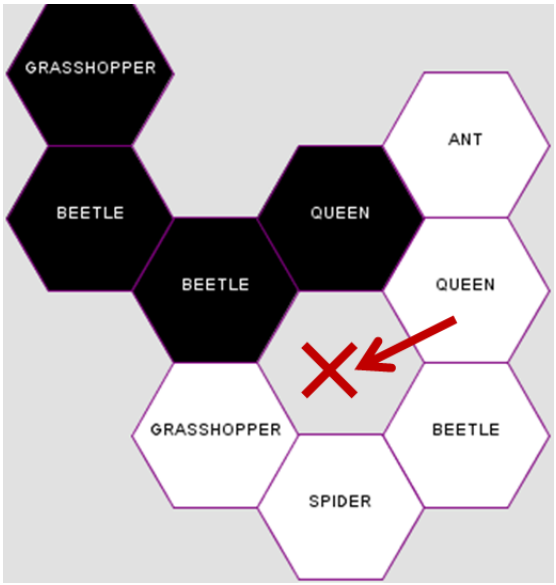


Figure 1: The white Queen cannot slide into this position because it is blocked by the black Queen and the white Beetle.

## 2.4 Description of the Creatures

**Queen Bee**

The Queen Bee is the weakest piece of the game and may move only one position each turn along the Hive. See Figure 2. Like the King in the game of Chess, the Queen Bee is a slow mover, but must be protected at all time, because a trapped Queen may result in losing the game. Despite its limited possibilities, a strategic movement of the Queen can disrupt the opponent's plans and influence the rest of the game.

**Beetle**

The Beetle has the same limited movement as the Queen Bee, but has an extra feature. Unlike any other piece, the Beetle may not only move along the Hive, but also on top of the Hive itself. In other words, the Beetle can climb on top of another piece and when it does, the piece underneath the Beetle is not able to move. During the occupation, the stack of the pieces takes on the colour of the Beetle for placement purposes. This means that a white piece can be placed next to a black piece if a white Beetle is on top of that black piece. The only way to stop a Beetle that is on top of the Hive is to move another Beetle on top of it. It is permitted to pile up multiple Beetles, but only the Beetle on top is able to move. Due to his extra feature, the Beetle can reach positions which are otherwise blocked by the Freedom to Move rule. However, the black Beetle in Figure 3 is already pinned by the One Hive Rule, but could otherwise not slide directly in between the Grasshopper and the Ant, blocked by the Grasshopper and the black Queen. Would the Beetle be on top of another piece in this situation, then it is allowed to slide into this position. Beetles are subjected to the general rules of placement, so they cannot be placed directly on top of the Hive, but can be moved there afterwards.

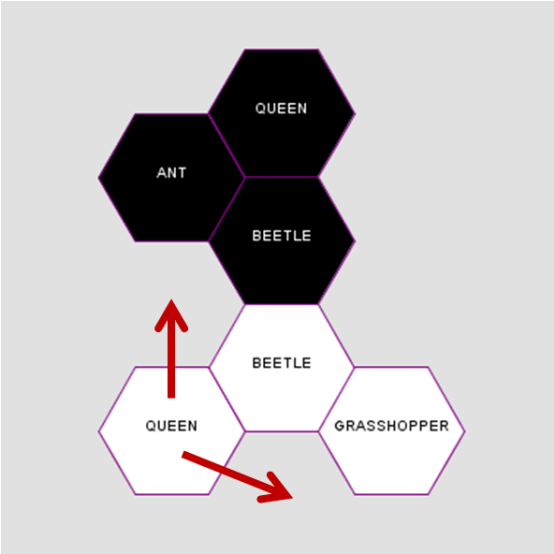


Figure 2: The Queen can only move one hex.

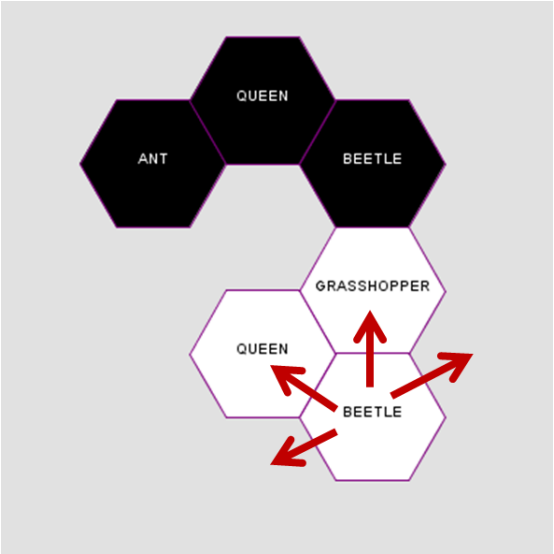


Figure 3: Beetles can climb on top of a piece in the Hive.

### Grasshopper

This piece moves in a completely different way than the rest of the creatures. Just like real grasshoppers in the outside world, this piece jumps over the Hive instead of moving along it. The movement may take place in every straight direction where they are directly connected to other pieces of the Hive. It jumps in the chosen direction in a straight line over as many possible pieces to the next free position. See Figure 4. This free position may be otherwise unreachable by the Freedom to Move rule.

### Spider

Spiders can move fast, but are not very flexible, since they may only move exactly three positions along the Hive, no less, no more. See Figure 5. They may not backtrack and must always be connected to at least one other piece every step of the way.

### Ant

The Ant is the strongest piece in the game. He is able to move to any position at the outside of the Hive while respecting the Freedom to Move rule. See Figure 6.

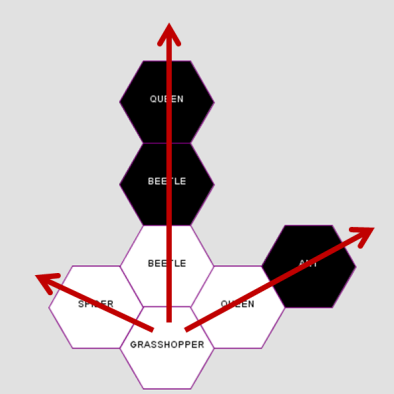


Figure 4: Grasshoppers jump over other pieces.

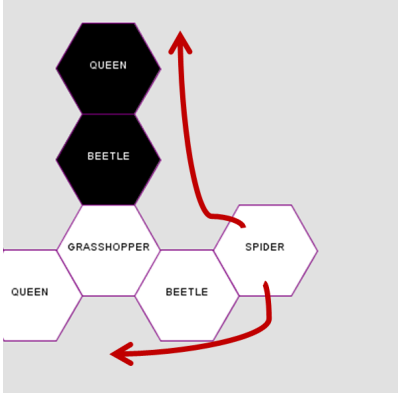


Figure 5: Spiders move exactly three hexes.

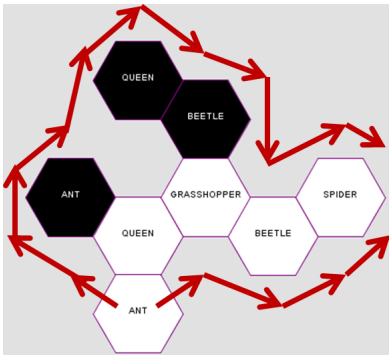


Figure 6: Ants can move to any unrestricted area.

# 3 Related work

There has been little published scientific research about the game of Hive so far. One of the important publications is a bachelor thesis written by Andreas Brytting and Johan Nygren at the KTH University in Sweden, published in April 2013 [3]. My thesis has a strong connection with this essay, as my AI strategy is built upon this publication and uses the program made by Brytting and Nygren. Their essay focussed more on the application of existing strategies and tactics. Their purpose was to find and analyse several strategies and find out which was the best. To do so, they assigned scores to the possible board-states and used the *Minimax algorithm* [4] to choose which move provided the best result. With Minimax, a full gametree is constructed upto a certain depth. Every possible state in this gametree is valued on their characteristics. The state with maximal feasible result for the player and minimal possible result for the opponent gets selected ultimately. Minimax was chosen because it is very well suited for turn-based games with two players where each player has perfect information on the current state of the game. Minimax can be very time-consuming, so they applied *Alpha Beta pruning* [5, p. 167] and only calculated states 3 turns in advance to save time. The Alpha Beta algorithm prunes the gametree and stops exploring a subtree when -for example- the minimal score at the given root node $v$ of the subtree is not greater than the maximum score at the parent of node $v$.

Brytting and Nygren implemented a couple strategies to construct an *offensive* and a *defensive AI agent*. The following two paragraphs describe the strategies and the scores awarded to the moves for the two AIs.

The defensive strategy score is based on the number of empty hexes next to the player's Queen and the construction of circles around the Queen. *Circling* is a good method to prevent the opponent from moving pieces next to the Queen. A circle structure makes one of the empty hexes adjacent to the Queen hard to reach, because the Spider and the Ant cannot slide in this locked position. Only the Beetle and the Grasshopper are able to go there, but that can take a lot of time. Setting up circles is a major goal at the beginning of the game. Because the pieces forming the circle can not be used for a long time, it is stimulated to place Beetles and Grasshoppers in the early part of the game. However, playing defensive all the time does not make significant progress to win the game in the end. To avoid spending pieces for defensive purposes only, Brytting and Nygren set a limit on achievable points for defensive moves and formations. If the limit is reached, points can only be earned with moves with offensive characteristics, so the strategy switches from defensive to offensive for that particular state.

The offensive strategy awards points for the amount of players playable pieces, where pieces are given values based on their importance. Furthermore, this strategy is focussed on the amount of pieces adjacent to the opponent's Queen and pinning or blocking high value pieces of the opponent. The tactics *Pinning* and *Blocking* make use of the One Hive rule and the Freedom to Move rule, respectively. If a piece is moved next to a single piece of the opponent, the latter would break the Hive if it would move away, i.e., separating the opponent's piece from the rest of the Hive. Therefore, one can lock a piece and pin the piece to its current position, making it unable to move. This is called pinning. Blocking

is done by moving pieces in a position where an opponent's piece cannot slide out of the current position, or is temporarily limited in its choice of positions to move to. Think of an Ant inside a Hive bay with a narrow entrance. This Ant cannot move along the outside of the Hive, but only on the inside.

*Beetle dropping* is another strategy implemented by the authors. With Beetle dropping, a player places a Beetle from hand as close as is possible to the opponent's Queen. The Beetle moves towards the Queen and climbs on top of her. By doing this, it may be possible to place pieces directly next to the Queen, since the Queen's position adopted the colour of the Beetle. Succeeding this will save a lot of time in comparison with the general approach of placing pieces and try to reach the Queen.

Brytting and Nygren played 48 test games with offensive and defensive strategy AI as players. For each test, at least one variable was changed to a small extent to avoid the AI from repeating the same game over and over again. The changed variables were the points assigned to creating circles, pinning opponents pieces or empty hexes next to the players Queen. Almost every game resulted in a tie. The defensive AI strategy won four games, the offensive only one.

# 4 Problem approach

In this chapter, I describe my approach to the project. At first, I discuss my attempt to implement the game by myself. After I received the program code from Brytting and Nygren, I decided to base my experiments on that, after fixing some bugs in the existing code. In section 4.2, I introduce the general Monte Carlo algorithm and after that, I explain the modifications I made to make the algorithm more suitable for this game.

## 4.1 Implementation of the game

The original plan for this project was to implement the game all by myself. Since it was known that the Monte Carlo algorithm would benefit from fast calculations, the programming focus was put on efficient use of memory and loops. Because of the set-up and lay-out of the game, it was also necessary to visualise the game over time. Although I was more experienced with *C++*, visualisation would be a bit harsh to do, so I choose the same programming platform as Brytting and Nygren: *Processing* [6]. While programming step by step, I started with the design of the board which I would store in a two-dimensional array. The hexagonal shape of the pieces makes it complicated to configure the right connection between adjacent pieces without wasting memory space. Therefore I figured out a way to store the pieces side to side and handle pieces in odd columns as they were shifted a half-sized piece down. The pieces could be stored horizontal to each other, while actually being diagonally connected in the game. It turned out that this idea was not very innovative since *Amit Patel* calls the usage of Offset coordinates *"the most common approach"* on his website *Red Blob Games* [7].

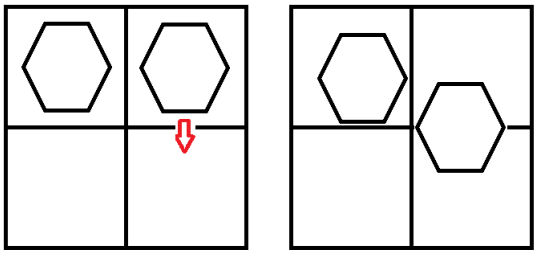

Figure 7: Real storage in comparison with semantic.

Before I could expand this idea and a few others, I had been given the advise to request the programming code of Brytting and Nygren from the KTH University. This saved me a lot of time and prevented me from reinventing the wheel, since I tried to accomplish the exact same thing Brytting and Nygren already did. While building on the same program, this even made it possible to compete our different AI agents. After I received the code, I examined the structure of it and made a random player in order to test the functionality. During simulations, both players did seem to ignore the One Hive Rule. This resulted in pieces torn loose from the Hive, unable to move since they were disconnected from the Hive. Research to another way to prevent pieces from breaking the One Hive Rule brought me to the *Floodfill algorithm*. This algorithm recursively marks every neighbour of every piece, starting from the centre of the board. A move was disapproved by this algorithm if the resulting state had more pieces on the board than marked pieces from the floodfill function, meaning that the Hive was broken. Unfortunately, this algorithm

has never functioned well, because Processing protested against the excessive use of so many recursive calls.

A more suitable method I studied later is called *Articulation Points* or *Cut Vertices* in connected undirected graphs. Nodes are articulation points *if and only if* removing one of them disconnects the graph. Tracing articulation points is important in general when testing the vulnerability of networks, because a malfunction of one of these points could split up the network and cause connection problems. From the perspective of Hive, articulation points are the pieces whose movements damages the One Hive Rule. The algorithm to trace these points starts with a *Depth First Search*(dfs) walk through the graph. This means that, starting from root *node v*, the first available child *node x* of *v* gets marked as *visited*. If present, the first child node of *x* gets marked and so on, until a node does not have children (i.e., a leaf). From there, the search walks back to the nearest node with one or more unvisited children. The order of visiting (*dfs#*) is stored for each node, since this is essential for the later algorithm. The algorithm is defined as follows: [8]

**Definition 1.**

1. *Function LOW(v) is the lowest dfs# of any vertex that is either in the dfs subtree rooted at v (including v itself) or connected to a vertex in that subtree by a back edge.*

2. *A back edge is a path of unvisited edges from one of the children of node v to an ancestor of v.*

3. *If some child x of v has LOW(x)≥ dfs(v), then v is an articulation point.*

In other words, node $v$ is an articulation point if node $v$ is the only connection between the children of $v$ and the rest of the graph.

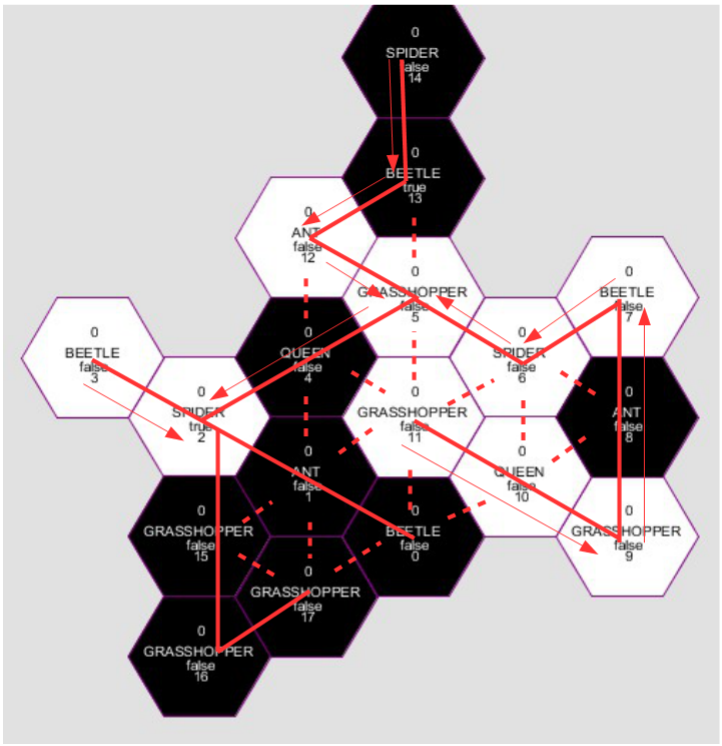


Figure 8: Example of finding articulation points.

| Piece | dfs# | LOW(v) |
|---|---|---|
| Black B | 0 | 0 |
| Black A | 1 | 0 |
| **White S** | **2** | **0** |
| White B | 3 | 3 |
| Black Q | 4 | 0 |
| White G | 5 | 0 |
| White S | 6 | 0 |
| White B | 7 | 0 |
| Black A | 8 | 0 |
| White G | 9 | 0 |
| White Q | 10 | 0 |
| White G | 11 | 0 |
| White A | 12 | 0 |
| **Black B** | **13** | **0** |
| Black S | 14 | 14 |
| Black G | 15 | 0 |
| Black G | 16 | 0 |
| Black G | 17 | 0 |

Table 1: Values of the pieces.

In Figure 8 are different red lines and arrows. The thick lines represent the Depth First Search as executed by the program, as can be seen from the number printed under the true/false statements at the pieces. This number is actually the *dfs#*. The small arrows shows the direction of backtracking when a leaf is reached or when a piece has only already visited children. The little dotted lines indicate the present connections between pieces and these connections can be used as back edges to reach the lowest possible *dfs#*. In this example, only the white Spider with *dfs#* 2 and the black Beetle are articulation points and therefore pinned by the One Hive Rule. Both pieces have a child piece with a value of LOW(x) greater than the *dfs#* of their parent. This two pieces are bold in the table.

After a closer look at the program class where Brytting and Nygren implemented their check on the maintenance of the One Hive Rule, it turned out that they already used the algorithm of tracing articulation points. However, debugging their implementation was a lot easier with the obtained knowledge. The bug arose because the list of pinned pieces was not refreshed at every turn and the algorithm was only set on the current state of the board, while temporary states needed to be checked also. Besides that, pieces underneath Beetles were not pinned properly, causing the Beetles to float on first level above the surface, possibly disconnected from the Hive.

The algorithm of tracing articulation points is much more efficient than the Floodfill algorithm, because the latter was called for every move of every piece in a state. The articulation points algorithm determines all the pinned pieces only once per state, which is a great discount in comparison with the Floodfill algorithm.

## 4.2 Monte Carlo

Algorithms for game players driven by Artificial Intelligence are trained to make better decisions during gameplay. Their improvements are provided by mathematicians and experienced professional players who help to implement strategic moves and tactics. This kind of algorithms use classical methods like Minimax to evaluate the possible moves. During the evaluation, a gametree is constructed. Depending on the game, the game tree can expand enormously, for example to a total of $35^n$ states in the game of Chess and over $100^n$ states in the game Go, with $n$ depth of the gametree. It is not accepted to wait an excessively long time for one evaluation, but on the other side, small game trees do not gain enough information. To achieve strategic moves for games with high branch factors, Monte Carlo methods are used. The origin of the method is found in the 1930s by Enrico Fermi, who used sampling methods to estimate quantities involved in controlled fission. After a game of solitaire, Stan Ulam realised in the 1940s that computers could be used to solve decision problems like card games [10]. Together with John von Neumann, Nick Metropolis and Edward Teller, Fermi and Ulam worked on the Manhattan Project [11] with a central role for the newborn Monte Carlo method. The name of the method is a reference to the famous casino in Monaco, a place Ulam's uncle visited many times after borrowing money from his family [12]. The first scientific publication of the Monte Carlo method in terms of Artificial Intelligence was written by Bruce Abramson in 1987. He designed a model based on the existing MC methods that calculated the expected outcome of moves in games [13].

### 4.2.1 Basic algorithm

The starting point for the construction of the Monte Carlo AI agent for the game of Hive was *Pure Monte Carlo Game Search*. This method consists of the most basic operation of Monte Carlo, just like Bruce Abramson defined it. With every turn in a game, a gametree with depth 1 is developed. The root of this gametree is the current state (the white node in Figure 8) and the children are possible moves for the current player (black nodes). With these possibles moves/states as roots, gametrees are developed furthermore. At every depth, a possible move gets randomly selected. The move of the opponent is also random selected. This repeats itself until an endgame is reached, a score is then returned to the root. This sequence of random selecting moves is called a playout. To achieve a high accuracy, each possible move does a couple of hundreds playouts. When all playouts are done, the average score is calculated for every possible move at the root and the best move gets selected for execution.[14]

### 4.2.2 Hive version

I made some modifications to the general Monte Carlo approach. I did this to save calculation time and to achieve better selective moves when playing. The first modification was the random selection of the next move. With a normal random selection, all possibles moves are listed and the random function selects one of them. In this way, every possible move has the same chance to get selected. Since every free moveable Ant has many more possible positions to move to, the Ant, as a piece, has a bigger chance to be selected. It would be better if every *piece* has the same chance to get selected. I think this is a more realistic approach because it is more likely to win the game when all the different

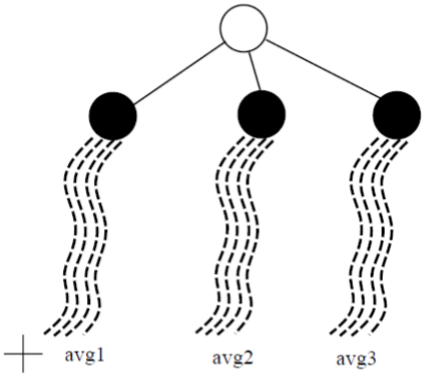


Figure 9: Schematic view of playout gametree, image from [15].

characteristics of all the other pieces are considered as well. When a random move is required, there is a choice between placing a new piece on board, or moving an existing piece. In the case of placing, a random available piece from hand gets placed at a random legal position, since this is not dependent on the type of creature. However, this approach provides pieces that have more copies a bigger chance to get placed at the first stage of the game. This was unfortunately hard to correct due to the implementation of the game, but I considered this as not significant enough to try.

If a piece needs to be moved the first thing to do is listing the moveable pieces and pick one of them. The selected piece may execute one of his available moves. Respecting the rules of the game, the freedom of choice of the algorithm is restricted when the Queen is not placed voluntarily in the third turn yet. In practice, this means the Queen is always placed at the third turn of the player, because the queen gets (almost) never selected for placement. This is not surprising, because logically the best way to prevent a loss is logically not placing the queen at all. Although the rules of the game do not force the placement of the Queen until the fourth turn, Brytting and Nygren stated that is it very disadvantageous to place the Queen in the fourth turn. Because the MC algorithm would otherwise have a bad start at every game, I forced the algorithm to place the Queen at the third turn. To restrict the calculation time for one turn, the random playouts in the algorithm stops the simulation after a fixed number of turns, instead of playing until a winning state. Besides a decrease in calculation time, this modification also improves the quality of the algorithm. Because after the move at the root of the gametree, there will be a lot of changes at the state, but it is unsure whether a won game is caused by the root move or by some of the random moves. It is even possible for random move to undo the root move by putting the piece back to his original position. With all of these uncertainties, the influence of the root move on the game result becomes less visible with each random move. Winning a simulation after many random moves should therefore not count in the valuation of the root move.

Keeping this argumentation in mind, the score for a possible move in my implementation of Monte Carlo is not based on the amount of playouts won only, but also takes the *depth of winning* into account. If a playout is won before the set depth limit (fixed number of turns), a counter gets incremented and the depth of the won playout is added to a total

depth sum. Because a player does not have direct influence on the opponent's moves, won playouts are not counted if all adjacent pieces around the Queen have the opponent's colour. When all the playouts are finished, the result score for each move is made up by the average depth of winning, decreased by the amount of playouts won. The move with the lowest score gets selected and executed. Although a score based on only the average worked as well, the subtraction was added after I ran a few tests. Sometimes I noticed a small gap in scores between the lowest and the second lowest score, but with a significant difference in won playouts, in favour of the second lowest. To increase the importance of won playouts in case of small variations in average depth of winning, the score is reduced by this number.

Since it is not forbidden to prevent the Monte Carlo algorithm from choosing unlogical moves after calculations, I modified the algorithm a little more. It is important to point out the difference between stimulating a good, human logical move and avoiding moves which nullify the player's progress to win the game. One of such moves is the emigration of a piece from a position next to the opponent's Queen to a less strategic position. With my modification, a move is not selected directly if the corresponding new state of the board reduces the number of pieces around the opponent's Queen. If this is the case, then the second best move is investigated and if necessary, the third best is also checked. Only when all three best moves remove an adjacent piece from the Queen, the first best move is still selected. With this approach, the Monte Carlo algorithm can select strategic moves based on the expected outcomes without external intervention. After a smart move is played, in this case getting a piece next to the opponent's Queen, the modification searches for alternatives if the Monte Carlo algorithm tries to remove such pieces from their strategic position. There are situations in which it is essential to remove an occupying piece from its position. For example, if this piece can pin or block an opponent's piece that is only one move away from surrounding the player's Queen. Such moves get better scores than almost all other moves that result in loss. For this reason, the modification do not keep investigating moves until a strategic looking move is found.

# 5 Experiments with Monte Carlo and Minimax

This chapter describes the experiments I carried out with the different Artificial Intelligence agents and corresponding parameters I had at my disposal. I distinguish between experiments with the Monte Carlo (MC) algorithm against the Minimax algorithm and experiments with Monte Carlo playing against itself. In all games, the algorithm is configured to play until 100 turns. After 100 turns, the game is stopped and recorded as a draw. This is necessary because Hive has a theoretical infinite game length and it is feasible to win a game within 100 turns, even with strategic players. If nothing is specified, my algorithm is set with 250 playouts for each possible move during the calculation of a move. As stated in the previous chapter, the playouts stop after a fixed number of turns. For almost all experiments, this number is 100. The first player has a great advantage with the starting move and this could have a influence on the rest of the game. To make the experiments more fair, the privilege of opening the game switches half-way during all the experiments.

## 5.1 Minimax matches

Due to time constraints, an experiment between the Minimax AI and the MC AI could not contain more than 50 games. Even when the experiments were executed on a remote server with quadcore Intel Xeon processors, running a 50 games experiment could take up to 16 hours. The bottleneck was the non-parallel programmed Minimax algorithm, while the individual playouts of the Monte Carlo algorithm could run in maximal 32 separated threads. In the table underneath, *Mod* stands for Modification. When an experiment is ran with strategy *Mod*, the algorithm uses the modified selection of scores to prevent bad moves, as described in the last paragraph in previous chapter. Otherwise the selection consists of searching the best score and execute the corresponding move immediately. The two strategies of the Minimax AI agent are *Def* (defensive) and *Agr* (agressive), further descriptions of these strategies can be found in Chapter 3. If nothing is specified, the Minimax algorithm builds and evaluates a gametree with a depth of three. Although it was the intention to test both strategies with the same settings, in test sessions it turned out that the aggressive strategy has a huge calculation time, which resulted in expired connections and running only 15 games during a two days continuous experiment. Therefore, the depth of the Minimax gametree is set at 2 while playing the aggressive strategy. Table 2 provides an overview of all experiments.

| Player 1 | Strategy | Player 2 | Strategy |
|---|---|---|---|
| Minimax | Def | Random | 0 |
| Minimax | Agr & depth = 2 | Random | 0 |
| Monte Carlo | No mod | Random | 0 |
| Monte Carlo | Mod | Random | 0 |
| Monte Carlo | No mod | Minimax | Def |
| Monte Carlo | Mod & 500 playouts | Minimax | Def |
| Monte Carlo | Mod | Minimax | Def |
| Monte Carlo | Mod | Minimax | Agr & depth = 2 |
| Monte Carlo | No mod | Minimax | Agr & depth = 2 |
| Monte Carlo | No mod & 500 playouts & depth = 50 | Minimax | Def |
| Monte Carlo | No mod & var playouts & depth | Minimax | Def |

Table 2: The configurations of the experiments.

## 5.2 Self-Play series

Besides the matches between the two AI algorithms, it is also quite interesting to study the characteristics of the Monte Carlo algorithm when it is playing against itself. The so called Self-Play series with Monte Carlo have their own field of research, with Ingo Althöfer as important person of interest. Althöfer describes a couple of (simple) games in his publication *Game Self-Play with Pure Monte-Carlo: The Basin Structure* [14] that are played by $MC(k)$ and $MC(2k)$ with $k$ number of playouts for every feasible move. The focus of the publication lies on the phenomenon of the percentage win *reduction* for $MC(k)$ when $k$ increases, following by a percentage win *increase* when $k$ is increased until a game specific threshold. This phenomenon is called Basin structure and has a limit of 50% winnings at the border of the basin. In this section, I will try to obtain a similar basin effect when two versions of the Monte Carlo algorithm are playing. Table 3 shows the set-up of the experiments done. To imitate the experiment as good as possible, the selection of best scores was adapted once again. While doing experiments with self-play series, the selection of best scores records moves with the same score of the current best move while searching. When two or more moves have the same best score, a move is chosen random. Thanks to the multi threaded Monte Carlo algorithm, it is affordable to play 100 games per experiment, which increases the accuracy of the results.

| Player 1 | Player 2 |
|---|---|
| MC(8) | MC(16) |
| MC(16) | MC(32) |
| MC(32) | MC(64) |
| MC(64) | MC(128) |
| MC(128) | MC(256) |
| MC(256) | MC(512) |

Table 3: Self-Play experiments with $MC(k)$ vs $MC(2k)$.

# 6 Results

This Chapter contains the results of the experiments described in the previous Chapter.

## 6.1 Minimax matches

As shown in Table 2, the experiments were run with a wide variation of parameters to investigate which modifications and configurations lead to the highest chance to win the game. The switch of making the first move did not provide remarkable insights. Both sub experiments resulted in the same ratio most of the time. Before I discuss the results of the Minimax vs Monte Carlo matches, I analyse the results from the played games with the random player. To prove that the AI agents are not retarded by nature, all four standard versions of the agents are set up to play against the random player, which selects a random possible move without any knowledge of the game. As shown in Table 4, the MC agent succeeded most of the time to win the played games. The few times the random player won, the MC agent was not able to foresee the problem of gathering his pieces next to his own Queen. And when the next random move surrounded the Queen by accident, the damage was done. Although the Minimax agent has better scores when playing against an advanced player, it won only half of the games with the random player. The reason for this is the defensive strategy that is more focused on pinning opponent's pieces than advancing towards the Queen to win the game. Most drawn games ended with the Minimax agent taking hostage all opponent's pieces and maintaining this situation.

Looking at the average turn of winning, the low value of the Minimax agents can be explained with the previous argument and with the basis of the Minimax algorithm. While the Monte Carlo algorithm selects the move with the highest expected outcome without taking the possible moves of the opponent into account, Minimax does. Minimax assumes that the opponent plays as optimal as Minimax itself. Because of that, Minimax does not select moves which are by itself better, but which provide the opponent a possibility to make a good move. In other words, Minimax is more focussed on preventing the opponent from making strategic moves than making progress in winning the game. The aggressive strategy seems to outperform this behaviour and ended two-thirds of the played games in draws. This could also be caused by the incremented depth of the searchtree, meaning the agent has too little information to get to strategic or winning moves.

| Exp. # | Player 1 | Strategy | Player 2 | P1 Won | P2 Won | Draw | Avg turn win |
|---|---|---|---|---|---|---|---|
| 1 | Minimax | Def | Random | 0 | 25 | 25 | 42.61 |
| 2 | Minimax | Agr & d=2 | Random | 16 | 0 | 34 | 37.23 |
| 3 | Monte Carlo | No mod | Random | 48 | 1 | 1 | 28.17 |
| 4 | Monte Carlo | Mod | Random | 47 | 1 | 2 | 28.96 |

Table 4: Results of random games.

The following paragraph contains analyses of the experiments and the corresponding results. The most eye-catching result is the high win rate for the Monte Carlo agent in Ex-

periment 8. Compare this with Experiment 7, where Monte Carlo had the same settings, but played against the Defensive strategy of the Minimax agent. A simple explanation would be that the MC agent can handle the aggressive strategy better than the defensive one. Unfortunately, this is probably the wrong conclusion. As stated before, experiments shut down due to time expiration when using the aggressive strategy on standard depth 3. Therefore, the depth was set back to 2. This modification has a huge impact on the quality of the algorithm, since most strategic moves require a preparation of multiple moves ahead. This is also observable by looking at the difference in the amount of states/moves that Minimax evaluates. With an estimated average of 45 possible moves each turn, the modified strategy can evaluate $45^2$ states = 2025, while it could evaluate $45^3 = 91125$ states under normal condition. Therefore, the high win rate is likely caused by the weakening of the Minimax agent.

The next comparison is between Experiments 5 and 7, which differ in the modification described in Section 4.2.2, with the purpose of making the MC agent a little smarter. The experiments show that this modification is not as effective as expected. Maybe the badness of the movement to prevent was not significantly and the agent has been forced to keep pieces next to opponent's Queen while trying to prevent the opponent from winning. When playing against the aggressive strategy in Experiments 8 and 9, the modification seems to improve the win chances, although it is easier to win from the impaired aggressive strategy player anyway.

Another attempt to tune the algorithm for a better win rate, was Experiment 6 where I increased the number of playouts from 250 to 500 loops. As shown in the table, there is a small improvement between Experiments 5 and 6. However, the win rate of Minimax is unchanged and it seems the modification only turns some drawn games into winnings. Besides that, this experiment took approximately twice the time to run and, in the end, it is not worth the time invested.

To bisect this huge calculation time, I ran Experiment 10 with the same amount of loops, but with a playout depth limit of 50, where I used a limit of 100 before. This resulted in the best win statistics of any game between the MC agent and the Defensive strategy of the Minimax AI agent. Although the ratio shows that MC is more than twice as good as the Minimax agent with these configurations, the high amount of draws is only caused by the Minimax agent. Lots of draws are achieved when the Minimax agent pins almost all of the pieces of the MC agent, or prevents the agent from surrounding the opponent's Queen. This behaviour is also proof of the smart defensive strategy of the Minimax agent, which chooses a draw over winning.

My last experiment was a *real* experiment, because it was set up different than usual and I had no expectation of what the result might be. The initial idea was that the MC agent always has the same playout depth limit, independent of the progress of the current game. In this way, playouts could find high win rates assuming the game will go on for 100 turns more, while the actual game is nearly marked as a draw by my configuration of the experiments. To stimulate moves which can result in a win before the game has come to an end, the playout depth limit was defined as $(100 - current.turn)$, where $current.turn$ is a integer containing the amount of turns done. Because the probability on winning

playouts is decreasing when the depth limit is decreasing, the amount of loops was increased with the same ratio to compensate this. The amount of loops was thus defined as $250 * (100/(100 - current.turn))$. The result is shown in the table by Experiment 11. The test did not work out as well as I expected, but the result is not that bad when compared to Experiments 5,6 and 7 which have lower profits for the MC agent and higher win rates for the Minimax agent.

| Exp. # | P1 | Strategy | P2 | Strategy | P1 Won | P2 Won | Draw |
|---|---|---|---|---|---|---|---|
| 5 | MC | No mod | Minimax | Def | 9 | 24 | 17 |
| 6 | MC | Mod & 500 p.o.'s | Minimax | Def | 13 | 25 | 12 |
| 7 | MC | Mod | Minimax | Def | 10 | 25 | 15 |
| 8 | MC | Mod | Minimax | Agr & d=2 | 33 | 5 | 12 |
| 9 | MC | No mod | Minimax | Agr & d=2 | 28 | 7 | 15 |
| 10 | MC | No mod & 500 p.o.'s & d=50 | Minimax | Def | 24 | 11 | 15 |
| 11 | MC | No mod & var p.o.'s & depth | Minimax | Def | 15 | 20 | 15 |

## 6.2 Self-Play series

The results showed that the game of Hive with this particular (modified) Monte Carlo agent does not provide a basin structure. This is caused by the low amount of played games for each experiment in comparison with Althoefer who ran at least 10.000 games for his experiments. Another important difference is that Althoefer used small games with finite game moves and finite game length. The so called *boardfilling games* are far more suitable for these kind of experiments.

However, one observation of the self-play series can be made. There seems to be a turning point half way the series. From MC(64) vs MC(128), the amount of loops seems to be high enough to contribute to a convincing winning. Before then, the amount of loops is too low to obtain enough reliable statistics causing the algorithm to play almost random.

| Player 1 | Player 2 | P1 Won | P2 Won | Draw |
|---|---|---|---|---|
| MC(8) | MC(16) | 44 | 49 | 7 |
| MC(16) | MC(32) | 44 | 55 | 1 |
| MC(32) | MC(64) | 52 | 47 | 1 |
| MC(64) | MC(128) | 37 | 63 | 0 |
| MC(128) | MC(256) | 29 | 71 | 0 |
| MC(256) | MC(512) | 36 | 64 | 0 |

# Conclusion

For this project, I constructed an Artificial Intelligence agent based on Pure Monte Carlo Game Search in the hope of improving it enough to beat the Minimax AI agent of Brytting and Nygren in the long-term. While doing this, I made some observations.

*Processing* is a nice programming platform to work with, but it is not suitable to execute many calculations within a time constraint. Thanks to the remote access servers of Leiden University, I was able to run my experiments in a reasonable amount of time. The project goal is only partly achieved. I managed to construct the Monte Carlo agent in a way that it was able to beat the Minimax agent constructed by Brytting and Nygren. However, since Brytting and Nygren did a great job with their implementation of the strategies and in a testset of multiple games, I expect the Minimax agent to win more games. Key to their success is the valuation of the pieces, which provides a huge source of information about the effect on the rest of the game of moving a particular piece. Although the defensive Minimax agent was able to beat the Monte Carlo agent almost every time, the experiments proved that the MC agent is at its best when using 500 playouts that play random until 50 turns ahead. Choosing these settings could make my Monte Carlo AI agent a strong Hive player.

Due to many characteristics of the game of Hive, the use of Monte Carlo as an AI agent is not recommended. I think that the algorithm has more success with games with a finite game length and universal pieces. With a finite game length, the random playouts to gain statistics about a possible move are limited which provides a more reliable expected outcome. However, the MC algorithm could had a better performance if the game, and so the algorithm, was implemented in a faster programming language. Further research could be done on doing this.

# References


[1] S. Gibbs, *Googles AI AlphaGo to take on World No 1 Lee Se-dol in Live Broadcast*, The Guardian, https://www.theguardian.com/technology/2016/feb/05/google-ai-alphago-world-no-1-lee-se-dol-live-broadcast, [Accessed on 26/01/2017].

[2] J. Yianni, *Hive: a Game Buzzing With Possibilities*, Gen42, http://www.gen42.com/downloads/rules/Hive_Rules.pdf, 2010.

[3] A. Brytting, J. Nygren, *Strategies in Hive*, KTH University, Stockholm, 2013, http://www.csc.kth.se/utbildning/kth/kurser/DD143X/dkand13/Group5Roberto/final/Andreas.Brytting.Johan.Nygren.report.pdf, [Accessed on 26/01/2017].

[4] E. Mayefsky, F. Anene, M. Sirota, Stanford University, 2003, *Strategies and Tactics for Intelligent Search* https://cs.stanford.edu/people/eroberts/courses/soco/projects/2003-04/intelligent-search/minimax.html, [Accessed on 26/01/2017].

[5] S. Russell, P. Norvig, *Artificial Intelligence: A Modern Approach*, Pearson Education Inc., 3nd edition, 2010.

[6] B. Fry, C. Reas, Processing Foundation, https://processing.org/, [Accessed on 26/01/2017].

[7] A. Patel, *Hexagonal Grids: Coordinate Systems*, http://www.redblobgames.com/grids/hexagons/#coordinates, [Accessed on 29/01/2017].

[8] L. Ruzzo, *CSE 421: Intro to Algorithms*, University of Washington, 2004, https://courses.cs.washington.edu/courses/cse421/04su/slides/artic.pdf, [Accessed on 26/01/2017].

[9] Gaojie He, *Monte Carlo Method Applied in Board Game AI*, Norwegian University of Science and Technology, 2009, http://www.idi.ntnu.no/~elster/tdt24/tdt24-f09/gaojie-ai-monte-carlo.pdf, [Accessed on 26/01/2017].

[10] T. James, D. Reeve, S. Dehghan Nasiri, University of Lancaster, *Monte Carlo Simulation: A Brief History*, https://www.lancaster.ac.uk/pg/jamest/Group/intro2.html, [Accessed on 26/01/2017].

[11] *Wikipedia: Manhattan Project*, https://en.wikipedia.org/wiki/Manhattan_Project, [Accessed on 26/01/2017].

[12] M. Mascagni, *Monte Carlo Methods: Early History and The Basics, slides*, Florida State University, 2015, http://www.cs.fsu.edu/~mascagni/MC_Basics.pdf, [Accessed on 26/01/2017].

[13] B. Abramson, *The Expected-Outcome Model of Two Player Games*, Morgan Kaufmann, 1991.

[14] I. Althöfer, *On Board-Filling Games with Random-Turn Order and Monte Carlo Perfectness*, Proceedings of Advances in Computer Games (ACG 2011), 2011, LNCS 7168, pages 258–269.

[15] Tsan-sheng H., *Monte-Carlo Game Tree Search, slides*, Academia Sinica, Taiwan, 2014 `http://www.iis.sinica.edu.tw/~tshsu/tcg/2014/slides/slide9.pdf`, [Accessed on 26/01/2017].