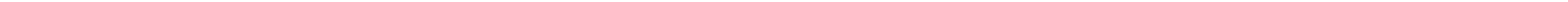
A Comparison Between the Usage of Flat and Structured Game Trees for Move Evaluation in Hearthstone

Master's Thesis

Markus Zopf
Knowledge Engineering Group
Technische Universität Darmstadt

TECHNISCHE
UNIVERSITÄT
DARMSTADT



Thesis Statement pursuant to § 22 paragraph 7 of APB TU Darmstadt

I herewith formally declare that I have written the submitted thesis independently. I did not use any outside support except for the quoted literature and other sources mentioned in the paper. I clearly marked and separately listed all of the literature and all of the other sources which I employed when producing this academic work, either literally or in content. This thesis has not been handed in or published before in the same or similar form.

In the submitted thesis the written copies and the electronic version are identical in content.

Darmstadt, April 7, 2015

Markus Zopf

Abstract

Since the beginning of research in the field of *Artificial Intelligence*, games provide challenging problems for intelligent systems. Chess, a fully observable zero-sum game without randomness, was one of the first games investigated in detail. In 1997, nearly 50 years after the first paper about computers playing chess, the Deep Blue system won a six-game match against the chess Grandmaster Garry Kasparov. A lot of other games with an even harder setup were investigated since then. For example, the multiplayer card game poker with hidden information and the game Go with an enormous amount of possible game states are still a challenging task.

Recently developed *Monte Carlo algorithms* try to handle the complexity of such games and achieve significantly better results than other approaches have before. Monte Carlo algorithms use random sampling to estimate the value of game states. They are particularly successful in games where it is hard to define a utility function to calculate the value of game states directly and where random simulations are easy to execute.

In this thesis, we investigate two different Monte Carlo approaches in the recently released card game *Hearthstone: Heroes of Warcraft*. This game combines various difficulties in game playing, like hidden information, randomness, and big game trees. *Upper Confidence Bound* approaches use the concept of bandits to represent moves shallowly in a game whereas *Upper Confidence Bound Applied to Trees* algorithms build structured trees to find the best move.

We find that both algorithms perform well against different random players. Win rates of about 0.90 can be achieved with low simulation counts used in both algorithms. Using higher simulation counts lead to even higher win rates of about 0.98. The direct comparisons of both algorithms show an unclear result: UCB surpasses UCT when only few simulations are used. This results from not widely enough expanded move trees in the UCT algorithm. If more simulations are available, UCT gets better and better and surpasses UCB. In experiments with the highest simulation counts, UCB beats UCT again.

We additionally find that both approaches have different weaknesses when they are applied to the game Hearthstone. These weaknesses result from the enormously high branching factor in Hearthstone and the way moves can be decomposed into atomic actions. To investigate the playing strength of both approaches better, we suggest evaluating the performance of other approaches based on rules or heuristics learned with reinforcement learning and experiment with enhancements for the UCB and the UCT algorithms.

Contents

1 Artificial Intelligence and Games **1**

1.1 Thesis Structure ... 1
1.2 Games as Measurement of Intelligence ... 3
1.2.1 Intelligence ... 3
1.2.2 Games ... 4
1.3 Games and Optimal Play ... 5
1.3.1 Simple Games ... 5
1.3.2 Multiplayer Games ... 6
1.3.3 Zero-Sum Games ... 7
1.3.4 Solving Games ... 8
1.3.5 Solving Two-Player Zero-Sum Games ... 8
1.3.6 Search Space Size ... 9
1.3.7 Randomness ... 10
1.3.8 Hidden Information ... 12
1.3.9 Pure and Mixed Strategies ... 15
1.4 Research Question ... 15
1.5 Similarities to Real World Problems ... 17

2 Hearthstone: Heroes of Warcraft **18**

2.1 Introduction ... 18
2.2 The Game ... 19
2.3 Cards ... 20
2.3.1 Minions ... 20
2.3.2 Spells ... 21
2.3.3 Weapons ... 21
2.4 Abilities ... 22
2.5 Composing a Move of Atomic Actions ... 24

3 Finding Solutions with Monte Carlo Methods **25**

3.1 The Need for Simulations ... 25
3.2 Game Trees in Hearthstone ... 26
3.3 Move Tree Complexity ... 28
3.4 Exploration/Exploitation Tradeoff ... 30
3.5 Bandit Approaches ... 30
3.5.1 Upper Confidence Bound ... 32
3.5.2 Final Move Selection ... 35
3.5.3 Anytime Property ... 36
3.5.4 Aheuristic Property ... 36
3.6 Monte Carlo Tree Search Approaches ... 37
3.6.1 Selection ... 38
3.6.2 Expansion ... 40
3.6.3 Simulation ... 41

3.6.4 Backpropagation ... 42
3.6.5 Upper Confidence Bound Applied to Trees ... 42
3.6.6 Asymmetric Property ... 43
3.7 Post Monte Carlo Strategies ... 43
3.7.1 Do Nothing ... 43
3.7.2 Additional Random Atomic Action ... 44
3.7.3 Additional Random Move ... 44
3.7.4 Additional Random Move with Maximum Length ... 44
3.8 The Parameter c ... 44
3.9 Random Approaches ... 44
3.9.1 Random Action Player ... 44
3.9.2 Random Move Player ... 45
3.10 Simulation of Games ... 46

4 Implementation Details **49**
4.1 General Information ... 49
4.2 Game State ... 49
4.2.1 Example ... 51
4.2.2 Actions & Events ... 53
4.2.3 Altering the Game State ... 55
4.3 Server and Player Interaction ... 56
4.4 Graphical User Interface ... 58

5 Experiments **60**
5.1 Configuration Overview ... 60
5.1.1 Random Action Player ... 60
5.1.2 Random Move Player ... 60
5.1.3 Random Move Player using Maximum Length Moves ... 61
5.1.4 Upper Confidence Bound Player ... 61
5.1.5 Upper Confidence Bound Applied to Trees Player ... 61
5.2 Confidence Interval ... 62
5.3 Evaluation Bias Caused by Evaluations in Different Game States ... 65
5.4 Comparison of the Random Players ... 66
5.5 Maximum Number of Bandits in UCB ... 69
5.6 Tree Size of UCT ... 70
5.7 Adjustment of the Parameter c ... 71
5.7.1 UCB ... 72
5.7.2 UCT ... 72
5.8 The Impact of the Number of Simulations ... 73
5.9 Using Post Monte Carlo Moves ... 74
5.10 Final Comparison between UCB and UCT ... 75

6 Conclusion **77**

7 **Further Work** 79
7.1 Pruning in Monte Carlo Algorithms 79
7.2 Improving Monte Carlo Methods 79
7.3 Game Tree Search 79
7.4 Usage of Score Functions and Reinforcement Learning 80
7.5 Opponent Modelling 80
7.6 Accuracy of Random Simulations 80

List of Figures 81

List of Tables 82

List of Listings 83

References 84

1 Artificial Intelligence and Games

In the first chapter of this thesis, we give an introduction into the topics *Intelligence* and *Games* and describe why and how they are strongly related. After the introduction to the concept of games and game trees, we are able to outline the idea of the thesis' topic and state the research question which is investigated in this work. In order to be able to analyze concrete questions we need further, more details information about the two approaches, which are compared in this thesis. In the end of this section, we state an example how the learned knowledge about game playing can be used in real-world problems.

Before we start with the topics *Intelligence* and *Games* we give a detailed overview about the thesis structure, a short insight in each section, and explain how the sections belong to each other.

1.1 Thesis Structure

As described above, we start to give an introduction to the topics *Intelligence*, *Artificial Intelligence*, and *Games* in section 1.2. There, we give an introduction to the complex topic of intelligence and how intelligence can be defined in different ways. This is important since we all have an intuition for intelligence and what it means, but when we investigate this concept in detail we will see that it is hard to give a precise definition about what intelligence exactly is and how it can be measured. As a result, we choose one of the four introduced definitions which should then be used as reference for writing about intelligence.

After the definition of the term intelligence, we define what a *game* is and what it means to play a game in an optimal way in section 1.3. Again, this is important as everyone has an intuition of the term *game* but this intuition may vary from person to person. For example, stock trading can easily be viewed as a game in the sense of this thesis whereas soccer will not. Therefore, we will define what a game is in the context of this thesis to clarify what we are writing about. To do so, we start with a very simple definition of a game in section 1.3.1. Since this first game definition does only cover very simple and therefore not very challenging games we will add more and more features to the definition, like multiple players, randomness, and hidden information in the subsections of section 1.3.

After having modelled games in a way to cover the game Hearthstone, we give an introduction to the research topic of this thesis in section 1.4. We introduce in this part two different approaches which can be used by an artificial player to play the game Hearthstone. This introduction will be rather general since the two approaches can hardly be explained sufficiently enough to understand the underlying notion in a few sentences. Therefore, we will only outline the two approaches before we explain them in detail in chapter 3. But not only do the concepts behind the two approaches have to be explained in detail before the research question can be fully understood. Also the game Hearthstone has to be investigated in detail since it has a special property, in comparison to other, in AI research commonly investigated games. In this game, it is possible to decompose moves into atomic actions or, differently stated, multiple atomic actions can be combined to a single move. But before we go into details of Hearthstone, we give an example to a real world problem as the gained knowledge in game playing can be transferred to such a situation. We already mentioned the example of stock trading. Most people would not define stock trading as a game because there is real money involved. But from the point of view from artificial intelligence, there is no big difference between, for example, playing poker and stock trading. Therefore, the research in games is not just a theoretically interesting task but also has practical applications.

After the introduction in artificial intelligence and games in chapter 1, we describe the game Hearthstone in chapter 2. We will at first provide some information about the background of the game and describe some general aspects before we start to explain the rules in detail. From the rules of the

game results the special action composition, which was already mentioned above. We will explain this property of the game in detail in section 2.5 since this is the basic for how the further two approaches are applied to play the game.

The two different approaches, the flat and the structured approach, which were already suggested in research question in section 1.4, are explained in detail in chapter 3. But before we start to do so, we investigate the game tree of the game Hearthstone and have a closer look on the move tree and their sizes. The result will be that the game tree is too big to be investigated exhaustively and even the move tree, which can be created with the atomic actions in a given game state, is too big to be built in many cases. Therefore, we have to handle the so called exploration/exploitation dilemma which is explained in section 3.4. The two approaches introduced in the sections 3.5 and 3.6 both give a solution to handle the exploration/exploitation dilemma. Both of them use simulations to estimate the value of a move. Therefore, they are called Monte Carlo algorithms.

In section 3.5, we introduce the flat bandit approach and the most important algorithm of this family. It is named after the key idea used in this approach, the *Upper Confidence Bound (UCB)*. The algorithm builds complete moves at first and then tries to find out which move is the best. In sections 3.5.3 and 3.5.4, two important properties of the upper confidence bound approach are explained. The anytime property says that the algorithm can be interrupted at any time and still delivers a useful result and due to the Aheuristic property no heuristic is needed to run the algorithm.

The second structured approach, called *Upper Confidence Bound Applied to Trees (UCT)*, is explained in section 3.6. The UCT algorithm has, like the UCB approach, the anytime and Aheuristic properties. Moreover, the algorithm asymmetrically grows a search tree which is explained in section 3.6.6

Chapter 4 gives some insights into the implementation of the game. Since there is no open source implementation of the ruleset available we had to implement the game on our own. First, we give some general information about the used software in section 4.1 and then describe some key concepts like the representation of the game state, and the interaction between server and player, and demonstrate a simple graphical user interface in the sections 4.2, 4.3, and 4.4. The descriptions will not go into the concrete code but stay on an abstract level.

The experiments executed to address the research question are explained in chapter 5 along with their results and the conclusions which we can be concluded out of the results. We first introduce the different players in section 5.1 before we investigate how many games must be executed to get a reliable result in section 5.2. We compare the performance of the random players in section 5.4 to provide evidence for the assumptions made in section 3.9. We then investigate the number of bandits used in the UCB algorithms and the tree size of the UCT algorithm in section 5.5 and 5.6. The adjustment of the parameter c is investigated in section 5.7 for both algorithms. We see in section 5.8 the performance of the different algorithms and improve the UCT algorithm in section 5.9 with post Monte Carlo moves. A final comparison of the UCB and the UCT algorithm is investigated in section 5.10.

We conclude the findings in chapter 6 and describe some ideas for further work in chapter 7. These include pruning in Monte Carlo trees in section 7.1, improvement of the simulation in the Monte Carlo algorithms in section 7.2 and extensions to the tree search in section 7.3. Furthermore, we suggest investigating the suitability of reinforcement learning algorithms in section 7.4 and the integration of opponent modelling in section 7.5. Section 7.6 comprises thoughts of the accuracy of the random simulation used in the Monte Carlo algorithms.

1.2 Games as Measurement of Intelligence

In this section, we give a short introduction to the concept of intelligence and how games can be used to measure the intelligence of a system, either artificial or not. In the first section 1.2.1 we give four definitions of the term intelligence and choose one that fits to the topic of this thesis best. In 1.2.2 we will briefly describe why games are interesting for research in the field of artificial intelligence.

1.2.1 Intelligence

Before we describe how games and artificial intelligence are connected, we start to give a short introduction to the general topic *intelligence*, since it is not a trivial task to describe what is meant by the term intelligence. As described in [*1*], there are different definitions of the term intelligence in the literature. They can be categorized by two dimensions: by *how a system works* and by *how a system acts*. Some definitions are strongly *human-focused*, as the human is by now the most intelligent system we know. Other definitions do not use this human-focused approach but rather a concept called *rationality*. A system is rational if it does the right thing to achieve goals.

Systems may, as mentioned above, be classified as intelligent if they work like humans do since humans are the most intelligent systems we know. The problem here is that it is not easy to understand how humans work. The brain, which gets inputs from sensors like the ears or the touch sense and produces outputs in form of speaking or drawing, is a complex system. The way of working of this system is still not understood completely. Therefore, it is not very helpful to define a system as intelligent as long as the way of working of the reference system is not fully understood. Approaches concerning this aspect of intelligence may lead to an artificial brain rather than an artificial intelligence. Such an artificial brain can certainly be intelligent but not everything that is intelligent has to work like a human brain.

Another approach concerning the working style independent from humans is the concept of systems which act rational. This means that a system is not classified as intelligent if it works like a human but rather if it works in a way that leads to the achievement of goals. Such a system would be based on logical rules and conclusions.

The both approaches above focus on the working style of the systems to classify them as intelligent. But if we reflect why we call intelligent people like Einstein, da Vinci or Newton intelligent, we will see that we call them intelligent not because we know how they have worked, but because we know what they have achieved. So it might be more helpful to define intelligence not by the way of working but rather on what the systems can achieve.

Again, we will start with the human-centered approach. Alan Turing proposed to define a system as intelligent if it can succeed in his *Imitation Game* [*2*], which is the origin of the famous Turing-Test. In this test, an interrogator asks two systems questions. One of the systems is a human and the other system is a computer. Both systems have the task to convince the interrogator that they are humanoid. If the interrogator cannot decide after several questions which one of the systems is the human and which one is the computer, the computer has passed the test and can be called intelligent. The name of the game origins from the task for the computer system as it has to *imitate* a human. In comparison to the approaches above, the way of working of the systems is irrelevant to pass the Turing-Test. Only the actions of the system are important. Recently, the findings in the field of behavioral economics of Nobel laureate Daniel Kahneman show that the assumption of human rationality is questionable. Exactly like artificial systems, the capabilities of humans are bound by time and storage limitations. It is, for example, impossible to find the lowest price for a widely spread product. If we want to buy a specific sort of apples there will be a lot of sources to get them from and all of these sources might have different prices. An intelligent system would select the source with the lowest price. As humans

cannot compare all the prices they will take a good offer but probably not the best. Furthermore, human decisions are strongly influenced by emotions. This makes it even more problematic to define the behavior of humans as the golden standard for intelligent systems.

In comparison to the Turing-Test, which has until now not been passed by an artificial system, the last definition does not focus on behaving like a human but like a rational system. A rational system is a system which will do its best to achieve a given goal. This could mean, for example, that buying the apples to the lowest price. A system which achieves a lower price will be more intelligent as a system which buys the apple for a higher price.

Table 1 summarizes the four possible definitions given above.

	humanoid	rational
think	A system which thinks like a human will imitate the way of working of the human brain.	A system which thinks rational will use logical rules and conclusions to make decisions.
act	A system which acts like a human will pass the Turing test and therefore will not be distinguishable from humans if the way of thinking is not considered.	A system which acts rational will make decisions accomplishing a predefined goal. The better the goal is accomplished the more intelligent the system is.

Table 1: Overview of four possibilities to define intelligence

We focus in this thesis on the concept of rational systems and say that systems are intelligent when they act rational. With this definition, we can now answer the question “Is a system intelligent?” with “Yes”, when it achieves good results in various tasks. As we cannot build a system which finds a solution for each problem easily, we focus on a special kind of problems, called games.

1.2.2 Games

As in the previous section described, we will analyze the rational behavior of systems to classify them as intelligent and to determine the amount of intelligence of a system. The more rational a system behaves, the more intelligent it is. A more rational system in this context is a system that can achieve better results than another system. To test if a system behaves rational, we can investigate how successful a system plays games. Non-physical games are well suited to test the rationality of systems, as there are rules which can easily be formalized as well as the outcomes of the actions and the goal, which should be accomplished. For example in chess, the rules are quite simple since they mainly consist of an initial setup position, the move abilities of the pieces, the alternating move sequence of the two players, and end conditions. By playing games, one can test the intelligence of a system by evaluating how good the goals of a game are achieved. To continue the chess example, we can say that a system, which wins against another system more often than the other way around, is more intelligent than its opponent, since it achieves the goal of checkmating the opponent more often. As such games are very well suited to investigate intelligence, games were always an interesting research topic since the beginnings in the field of artificial intelligence. In the beginning of AI research in 1950, shortly after the invention of programmable computers, chess was already in the focus. Since then, the performance of the agents improved constantly until today. Nowadays, game-playing AIs are capable of beating the best human players in games like chess, backgammon, and poker.

To make an early distinction, we will neither investigate physical games, like soccer or tennis, where physical attributes like strength or speed are important, nor real-time games or games of skill like

pinball, billiard, or coconut shy. We will focus on non-physical games where strategic decisions must be made to achieve goals in the game. We will therefore not investigate games which need a physical presence of the artificial intelligence in the form of a robot with arms or legs.

To close the introduction to games, we cite Richard Sutton who said that "games are to AI researchers what fruit flies are to biology – a stripped-back system in which to test theories".

1.3 Games and Optimal Play

In the previous section, we already talked about games and that they are a good possibility to investigate the intelligence of a system. We now introduce games in a more formal way to give a precise definition what games are in the context of this thesis. To do so, we start by introducing the concept of a simple game. This simple game will then be extended by more components and possibilities. But we will not only define what games are but also explain which of them can be solved. What *solving a game* exactly means is described in section 1.3.4.

1.3.1 Simple Games

Before we will start to discuss properties of and difficulties in games, we will give a definition of what we will call *a simple game*. This definition of a simple game will then be further extended to cover more complex games.

The early definitions of games come from John von Neumann and Émile Borel. This *normal form* called definition strongly focuses on possible strategies of players and payoff functions, which represent the outcomes of the games and does not represent the sequencing of the players' possible moves and the player's choices at every decision point in games explicitly. As this information will later be very important to define sequential algorithms we will use the so call *extensive form* to define a game.

Definition:
A *simple game* in extensive form is the structure $\Gamma' = \{p, N, s_0, T, p, u\}$, where

- p is a player,
- $N = \{s_0, \dots, s_n\}$ is a set of $n + 1$ game states,
- s_0 is an initial game state,
- $T \subset N$ is a set of terminal game states,
- $p : N \to D$ a predecessor function which defines the game rules, and
- $u : T \to \mathbb{Z}$ is a utility (or payoff) function for terminal states.

For simplicity we additionally define

- $D = N \setminus T$ as the set of decision (or non-terminal) game states, and
- $M(s_y) = \{s_x : p(s_x) = s_y\}$ as the set of possible moves, which can be made in game state s_y

To give an example, we will consider the following game, which should be one of the simplest games imaginable:

"There is one player p. This player can choose from one of three options: Option A, B, and C. If he chooses option A he will receive a reward of 3 points, for option B he will get 5 points, and for option C he will receive 1 point. The goal of the game is to earn as much points as possible."

To represent this simple game in extensive form, we will set

- $N = \{s_0, s_1, s_2, s_3\}$,
- s_0 as the initial game state,
- $T = \{s_1, s_2, s_3\}$ and get $D = \{s_0\}$,
- $p = \{(s_1, s_0), (s_2, s_0), (s_3, s_0)\}$ and
- $u = \{(s_1, 3), (s_2, 5), (s_3, 1)\}$.

This representation can easily be transformed into a rooted, connected, directed graph without loops: *a tree*. We take the set N as nodes of the tree and derive the edges of the tree from the function p. If we add the values of the utility function u to the terminal nodes we get a *game tree*. In Figure 1 we see the game tree for the simple game mentioned above.

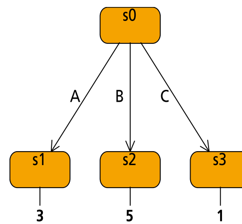


Figure 1: A drawing of the game tree for the simple one player game

We see s_0 as root node at the top of the tree. Outgoing from game state s_0, the player has the three options A, B, and C to get to the game states s_1, s_2, and s_3 respectively. The terminal game states are annotated with the values of the utility function u.

For this simple game we can easily see what we have to do if we want to maximize the outcome while playing the game. As option B gives us the maximal outcome of 5 points, we will always take option B.

1.3.2 Multiplayer Games

To be able to speak not just about such trivial games, we now describe a more complex structure which will be able to model multiple player games.

Definition:
A *game* in extensive form is the structure $\Gamma = \{P, N, s_0, T, p, u\}$, where

- $P = \{p_1, \dots, p_n\}$ is a set of p players,
- $N = \{s_0, \dots, s_n\}$ is a set of $n + 1$ game states,
- s_0 is an initial game state,
- $T \subset N$ is a set of terminal game states,
- $p : N \to D$ a predecessor function which defines the game rules, and
- $u : T \to \mathbb{Z}^p$ is a utility (or payoff) function for terminal states for each player.

Additionally, we define sets to model which player has to act in the game states. For each player p_i we set $A_i = \bigcup s_j$ for all game states s_j where player p_i has to act.

Every non-terminal game state now additionally stores the information which player has to make a move in the given game state. The utility function now provides a vector containing a utility value for each player in the game.

To get an example of a game tree for a two player game, see Figure 2.

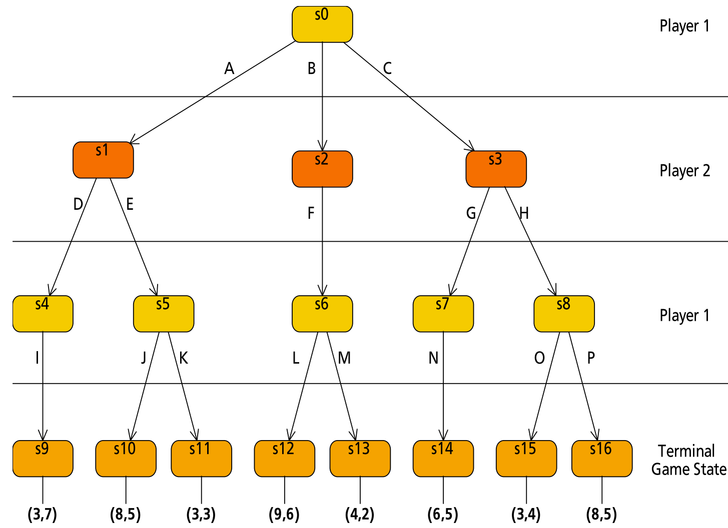


Figure 2: A simple two player game

This game can be modeled in extensive form as

- $P = \{p_1, p_2\}$,
- $\{s_0, s_1, s_2, s_3, s_4, s_5, s_6, s_7, s_8, s_9, s_{10}, s_{11}, s_{12}, s_{13}, s_{14}, s_{15}, s_{16}\}$,
- s_0 as the initial game state,
- $T = \{s_9, s_{10}, s_{11}, s_{12}, s_{13}, s_{14}, s_{15}, s_{16}\}$ and get $D = \{s_0, s_1, s_2, s_3, s_4, s_5, s_6, s_7, s_8\}$,
- $p = \left\{ \begin{matrix} (s_1, s_0), (s_2, s_{02}), (s_3, s_0), (s_4, s_1), (s_5, s_1), (s_6, s_2), (s_7, s_3), (s_8, s_3), (s_9, s_4), \\ (s_{10}, s_5), (s_{11}, s_5), (s_{12}, s_6), (s_{13}, s_6), (s_{14}, s_7), (s_{15}, s_8), (s_{16}, s_8) \end{matrix} \right\}$, and
- $u = \left\{ \begin{matrix} (s_9, (3,7)), (s_{10}, (8,5)), (s_{11}, (3,3)), (s_{12}, (9,6)), \\ (s_{13}, (4,2)), (s_{14}, (6,5)), (s_{15}, (3,4)), (s_{16}, (8,5)) \end{matrix} \right\}$.

1.3.3 Zero-Sum Games

Zero-sum games are a special kind of games. In games like the game displayed in Figure 2, the utility function can have arbitrary values for each player. So the utility values do not depend on each other. In zero-sum games, there is a restriction on these values. As the name suggests, in zero-sum games the values of the utility function have to sum up to zero. We can formalize this by adding a constraint to the values of u:

$$\forall \begin{pmatrix} x_1 \\ \vdots \\ x_p \end{pmatrix} \in \mathbb{Z}^p, t \in T : u(t) = \begin{pmatrix} x_1 \\ \vdots \\ x_p \end{pmatrix} \Rightarrow \sum_{i=1}^{p} x_i = 0$$

That means that no player can gain a bigger reward without reducing the rewards of the other players. Furthermore, for a two player game, it is sufficient to state the value of the utility for a terminal state for the first player as the value of the second player then is the same value negated. In the following two player game drawings, we always display the utility value for the first player. As describe above, both players want to maximize their utility value. As the utility value for the second player equals in

zero-sum two player games the additive inverse of the utility value of the first player, the second player's goal is to minimize the utility value of the first player. By doing that, he maximizes his own utility value. Therefore, we will from now on call the first player p_1 MAX, as he wants to maximize the displayed value in the game tree. The second player p_2 will analogously called MIN, as he wants to minimize this value. Figure 3 gives an example of a two player zero-sum game tree.

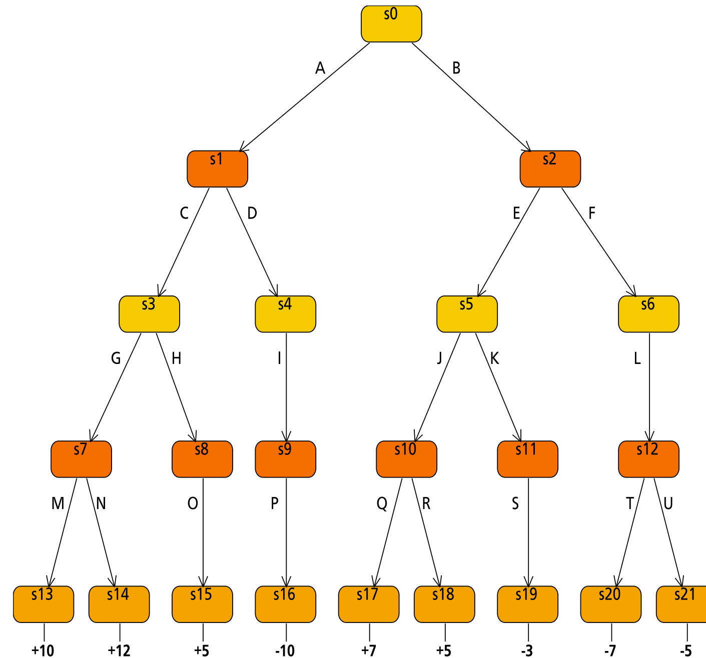


Figure 3: Game tree of a two player zero-sum game

1.3.4 Solving Games

We now make a short break in extending the simple game from section 1.3.1 since we reached a point where we can already define sufficient enough complex games to be interesting. As the goal of a player playing a game is to win as much as possible we have to know in which game states a player can expect a good outcome and which game states are not promising. To do so, we introduce the concept of solving games in a weak and in strong way.

1.3.5 Solving Two-Player Zero-Sum Games

Solving a game in a strong way means that a player can determine for every possible game state the optimal move which leads this player to the best possible outcome. For a simple single-player game as defined in 1.3.1, this means that, theoretically, it is possible to solve every game by simply searching the complete game tree for the path which leads to the best outcome.

In a two player zero-sum game, this is not easily possible because one player alone cannot choose the complete path from the initial game state to a final game state. The first player, for example, can only decide which path is taken in the game tree when it is his move. If it is up to the second player to make a move the path cannot be influenced by the first player. But the problem is not just that one player cannot influence the decision of the other player. As we consider zero-sum games, the other player has the exact opposite target. So he is interested in foiling the plans of the first player as much as possible. Nevertheless, there is an algorithm that solves this problem for both players, if both players play rational. This simple recursive algorithm is called Minimax-algorithm because one player tries to

minimize and the other player tries to maximize the resulting value. The pseudocode of the algorithm is written in Listing 1.

```
function minimax(state):move
v ← maxValue(state)
return move to get to successor state in p(state) with value v

function maxValue(state):integer
if state ∈ T
   return u(state)
else
   v ← −∞
   for each successor with p(state) = successor do
      v ←max(v, minValue(successor))
   return v

function minValue(state):integer
if state ∈ T
   return u(state)
else
   v ← +∞
   for each successor with p(state) = successor do
      v ←min(v, maxValue(successor))
   return v
```

Listing 1: Pseudocode of the minimax algorithm

1.3.6 Search Space Size

With a minimax algorithm, even more complex games like Tic-tac-toe, Connect Four, Nine Men's Morris, Chess, and Go can theoretically be solved in and optimal way. But in practice, the number of possible courses of the games is too big to be evaluated exhaustively. In Table 2 an overview of the complexity of these games is given.

Game	Average game length (in plies)	Average branching factor	Number of game states (as log to base 10)	Number of courses of the game (as log to base 10)
Tic-tac-toe	9	4	3	5
Connect Four	4	36	13	21
Nine Men's Morris	10	50	10	50
Chess	35	80	47	123
Go (19x19)	250	150	171	360

Table 2: Overview of the complexity of 5 well-known games

The number of courses of the game in the last column is an estimation for the number of game state nodes must be visited of a minimax algorithm to determine the optimal play for the players. As this is practically possible for a simple game like Tic-tac-toe, it becomes infeasible for more complex games like Chess or Go. Alpha-beta pruning [1] can help to reduce the number of game states which must be visited to evaluate the best move in a run of a minimax algorithm. Nevertheless, the number of remaining game states is too big to be manageable in practice.

1.3.7 Randomness

The games mentioned above were always strictly deterministic. There were no changes of the game state that were not controlled by one of the players. But in a lot of games, randomness is involved. Examples for randomness are rolling a dice in a board game or drawing a card in a card game. To be able to describe such games in a formal way like in the examples above, we have to further expand the definition of a game. To do this, we add an additional player p_0 to the set of players. This player is called Nature or Chance. He will make a move in the game tree every time when a random event happens. To describe the moves of Chance we will additionally add a probability distribution for every game state where a random event occurs. Note that these probability distributions do not have to be known by the players. It is easily conceivable that the players do not know all or some of the distributions.

Definition:
A *game with randomness* in extensive form is the structure $\Gamma = \{P, N, s_0, T, p, u, Q\}$, where

- $P = \{p_0, p_1, \dots, p_n\}$ is a set of $p + 1$ players,
- $N = \{s_0, \dots, s_n\}$ is a set of $n + 1$ game states,
- s_0 is an initial game state,
- $T \subset N$ is a set of terminal game states,
- $p : N \to D$ a predecessor function which defines the game rules,
- $u : T \to \mathbb{Z}^p$ is a utility (or payoff) function for terminal states for each (non-nature) player, and
- $Q = \{q_c: M(c) \to [0,1], c \in C \subset D\}$ is a set of probability distributions which contains a probability distribution q_c for every chance node c. The set C models the set of all chance nodes in a game tree.

Note also that there is no utility value for the Chance player given. His moves are only depending on the defined probability distributions. The Chance player does not have any interest in making good moves or avoid bad ones. He therefore does not try to minimize or maximizes his utility value for the played game. Figure 4 shows an example game tree for a zero-sum game with chance nodes.

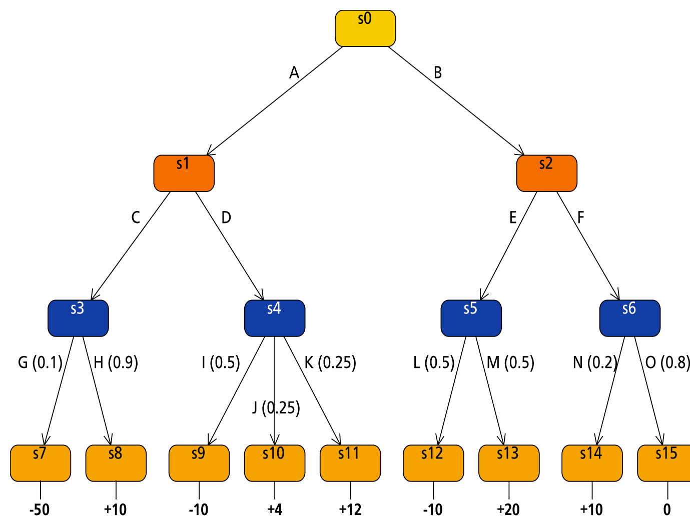


Figure 4: Game tree of a game with chance nodes (s3-s6)

As we see, MAX has to make a decision in s_0: He can either take option A or option B. MIN then can make the moves C or D or has to choose from moves E and F. After MIN has made his choice, the player CHANCE makes his "move". In game state s_4 for example, he will play move I with probability

0.5, move J with probability 0.25, and he will take option K with probability 0.25. The strategies in which a player chooses different actions in the very same game state is called a mixed strategy, which is explained in more detail in section 1.3.9. In this game, we will assume that the players MAX and MIN both know the probabilities of the strategies of CHANCE.

Like in the two-player zero-sum game without randomness, there is an algorithm to theoretically play at each game state an optimal move in games with randomness under the condition that all players play rational in the sense defined above. This algorithm is called Expectiminimax or shorter Expectimax. A pseudocode for the algorithm is given in Listing 2.

```
function expetimax(state):move
v ← value(state)
return move to get to successor state in p(state) with value v

function value(state):integer
if s ∈ T
   return u(state)
else if state ∈ A_1
   return maxValue(state)
else if state ∈ A_2
   return minValue(state)
else
   return avgValue(state)

function maxValue(state):integer
v ← −∞
for each successor with p(state) = successor do
   v ←max(v, value(successor))
return v

function minValue(state):integer
v ← +∞
for each successor with p(state) = successor do
   v ←min(v, value (successor))
return v

function avgValue(state):integer
v ← 0
for each move ∈ M(s_i) do
   v ← v + prop(move) * value(m(move))
return v
```

Listing 2: Pseudocode of the Expectimax algorithm

If we apply the expectimax algorithm to the game in Figure 4 we get the solution displayed in Figure 5.

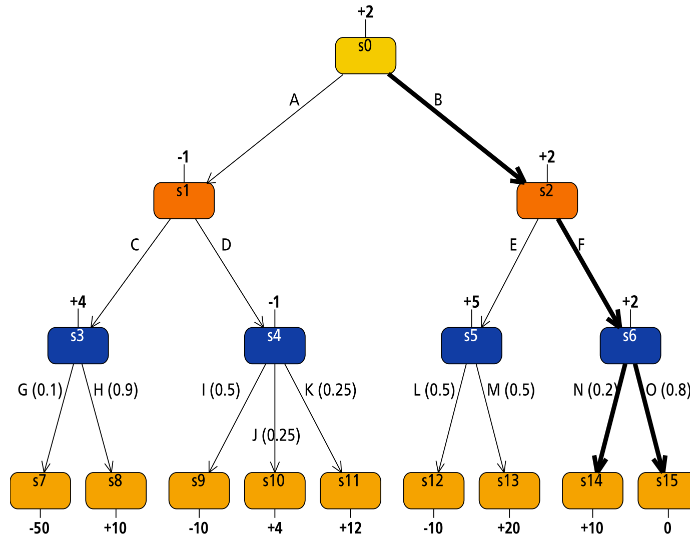


Figure 5: Expectimax solution of the game displayed in Figure 4

To evaluate which move is better, A or B, MAX has to evaluate which move would MIN choose in game states s_1 and s_2. MIN on the other hand has to evaluate which move CHANCE would make in the game states $s_3 - s_6$ to evaluate which move would be the optimal play for him. As MIN knows the probability distributions of CHANCE, he can calculate the average outcome if he decides to play the moves C to F. If he is in game state s_1 he will get for move C an average outcome of $+4$ and for move D he will get an outcome of -1. As he is interested in minimizing the utility function value, he will choose option D. Analogously, he will choose option F in game state s_2. Now MAX can evaluate that choosing option B will be better than option A and he will get $+2$ points as an average outcome of the game by playing option B. The path which will be played is displayed bold.

1.3.8 Hidden Information

In all games described above, the state of the game was completely visible to all players at any time. Each player had exactly the same knowledge about the game. This is true for a lot of board games like Connect Four, Chess, or Backgammon. But there are also games where a player has access to some information and the other player or the other players do not. For example, in some card games the players often have cards in theirs hands. These cards are only visible to the owner and are hidden from the opponents. In *Secret Mission Risk*, a variation of the classical game *Risk*, each player draws a secret mission at the beginning of the game which is only visible to the owner. The goal is to fulfill this mission to win the game. Therefore, the mission is hidden information for the other players. It is important to note that information which is not known by any players of a game is not called hidden information in the sense of this chapter. For example, in the card game *Uno*, there is a card stack from which all players draw cards during the game. As the ordering of the cards in the card stack is not known to any player, this is not hidden information for the players. Drawing a card would rather be modelled as a move of nature than revealing hidden information. As the knowledge of which card was drawn is only visible to one player (the player who has drawn the card), he now holds hidden information in comparison to the other players.

To model the hidden information, we introduce the partition

- $\boldsymbol{H_i}$, which is a set of partitions of the set D. The set $\boldsymbol{H}_i$ is the information set of player i.

The interpretation of an information set of a player is that all game state in this information set are indistinguishable for the player. For example $\{s_1, s_2, s_3\} \in \boldsymbol{H}_i$ means that the three states s_1, s_2, and

$s_3$are indistinguishable for player i. Player i therefore cannot distinguish if a game is in the state s_1, s_2, or s_3. Figure 6 illustrates such a situation.

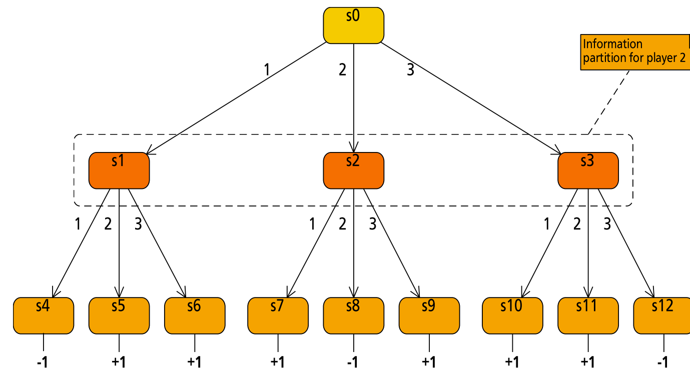


Figure 6: A game tree with an information partition of game states s1-s3 for player 2

The interpretation of this game tree is the following: MAX has to imagine a number between 1 and 3 and MIN has to guess this number. If he guesses right he wins a point, otherwise he will lose one. After MAX chose a number, MIN knows that MAX chose a number but does not know which. Therefore, MIN does not know if he is in game state s_1, s_2 or s_3. As MAX does know this, he has hidden information compared to MIN. Would MIN know the actual game state, he could easily choose the move which would lead him to a good outcome of -1. But since he does not know the actual state he can only guess in which state the game is and get an average outcome of $+1$.

The important difference between information which is only known by one player or some players (for example hand cards) and information which is not known by any players (for example which card will be drawn next) is that, for the latter, an expectimax algorithm can theoretically be used to get an optimal move if the random distribution for all moves of nature are known. For games where the former situation occurs, this is not possible in general, as the players may not know at all game states in which game state they exactly are. For example, in the game of poker, even for absolute masters it is not possible to say in which game state they are. Or in other words, even they can never be sure about their opponent's hand cards. Furthermore, different experts of a game like poker will surly have different ratings of the very same game situation. These different views of the same situation may be based on different experiences about the opponents made in the past. An expert A may have been tricked a lot by his opponent in the past whereas expert B observed a more straight forward play of this player.

Furthermore, it is important to note that the available moves of a player only depend on the current information partition and not on the actual game state in the partition. In the example above, MIN knows that he is in the information partition of the game state s_1, s_2, and s_3. As those three game states are in an information partition, the possible moves in the game states s_1, s_2 and s_3 are the same. In every game state MIN has the option to choose between move 1, 2, and 3. This is the case every time a player cannot distinguish the game states in an information partition because he must know which moves he can make. If the available moves would depend on the actual game state in an information partition, the player could distinguish the game states. But this is per definition not possible in an information partition. Therefore, the available moves must be the same for each game state in a specific information partition.

We give a further example in a game in which both players have hidden information because it is important to understand in which way such hidden information influences the size of the related game tree. We will further use the details explained in this example in section 3.2 to model the game tree of

Hearthstone and calculate its size correctly. To investigate this situation in detail, we take a look at a simple game in Figure 7.

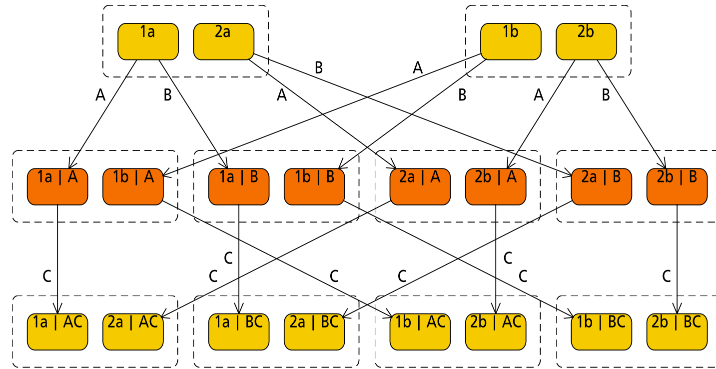


Figure 7: A game tree in which both players have hidden information

In this game, we imagine that both players, MAX and MIN, have hidden information from the beginning of the game. For example, this could be a secret mission, like in the game *Secret Mission Risk* which is only revealed once one player wins, or an initial start hand in a card game. The latter will be the case in section 3.2. To simplify the situation, both players only have few hidden information in the game state plotted above. We model this hidden information as variables max for player MAX and min for player MIN. The variable max can only have to states: 1 or 2. Analogously, the variable min can only have the states a and b. To model the game tree in detail and how the information partitions are composed, for each game state we show the values of the two variables and additionally which moves were executed to reach that game state. Figure 8 shows a game state with hidden information $2a$ which means that $max = 2$ and $min = a$. Additionally, the move A was executed to reach the game state, which is visible to both players.

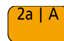


Figure 8: Example game state with hidden information 2a and move A

As, for example, min is hidden information for the player MAX, MAX cannot see if min has the value 1 or 2. Only MIN knows what the value of min is. Therefore, we see in Figure 7 that the game states $1a$ and $2a$ are indistinguishable for player MAX and that they build an information partition for MAX. The same holds for the game states $1b$ and $2b$.

MAX has to decide in the first ply if he plays option A or option B. As mentioned above, the moves a player can play do not depend on a specific game state in a partition set. But as his hidden information is not revealed until the end of the game, it is necessary that he can play exactly the options A and B in every information partition in this ply. So for example, it is not possible that he can play options A or B in the first set and C or D in the second set. If this was the case, MIN could, depending on the move MAX made, detect in which state MAX's hidden variable is. MIN then could distinguish the game states $1a \mid A$ and $1b \mid A$ (which would then be $1b \mid C$ or $1b \mid D$). This would mean that the hidden information of MAX is revealed.

As MAX has two options to play in every game state in this ply, the next ply of MIN has exactly two times of the game states as in the first ply. In the second ply, MIN has only one option to play: move C. As argued above, option C is possible in every game state. The number of game states equals therefore the number if game states in the ply before. It is important to note that the number of game states does not grow exponentially after each ply, but is only multiplied with the number of available moves in the

ply before. This is the case, because the hidden information is fixed in every path in the game tree except a move change the information explicitly. The game state $1b \mid XY$ cannot be reached with move Y if the game state $1a \mid X$ was visited during the game without making a move, which changes the value of the variable.

1.3.9 Pure and Mixed Strategies

In the games above, we always assumed that all players play a rational strategy. This means that each player plays at each time the move, which will optimize his outcome under the assumption that all other players will play optimal. This is called a *pure strategy*. But playing a pure strategy must not always be the best solution. Especially when games are played multiple times, players can adapt their behavior to exploit this behavior. For example, we take the game described in Figure 6 on page 13. If the first player MAX always chose the number 3, MIN could discover this behavior and adapt his move in the following turn. He could then always guess number 3 and would always win the game. So it is not a good idea for MAX to play the pure strategy "3". He should vary its moves from game to game and sometimes take number 1, sometimes number 2 and he also should choose number 3 every now and then. This way, MIN cannot exploit the strategy of MAX. As MAX is mixing the three options he has in game state s_0 this is called a *mixed strategy*.
To consider a slightly other game as in Figure 6, we will look at the two player zero-sum game Rock-Paper-Scissors. In this game, both players choose at the same time one of the options "rock", "paper", and "scissors". If both players choose the same option, it is a draw. If they choose different options, the following rules apply: rock beats scissors, which means the player who chose "rock" wins and the other player loses. Furthermore, scissors beats paper and paper beats rock. As above, a pure strategy could be easily exploited. Only a mixed strategy in which both players take the three options at random using a uniform distribution guarantees that the strategy is not exploitable. The situation in which no player can change its strategy without becoming exploitable is called *Nash-equilibrium* [*3*,*4*].

In the game in Figure 6, MAX switched from a pure to a mixed strategy. But in doing so he still played an optimal move as all three options had an expected outcome of +1 (assuming that MIN will play optimally). But there are also cases where playing the optimal move is not always the option with the best outcome. This is the case when it is known that the opponent will not play optimally, too. Therefore, sometimes it is better to not play the optimal move. This can have mainly two reasons. Firstly, it is possible to change the own strategy to exploit weaknesses in the opponent's strategy. For example, if one player knows that the other player often plays "Rock" he can exploit this behavior by playing "Paper" more often. The second reason is to trick the opponent and make him believe something about hidden information which is not true. To show this, poker is a good example. One of the main sub-goals in playing poker well is to guess the opponent's hand cards. If one player can give a good guess about the hand cards of another player it is more likely to gain a better outcome. Because guessing the cards of the opponent is so important, it is important to trick the opponent in a way that he will make a wrong guess. Bluffing is such a move: When a player has poor hand cards he nevertheless can make a high bid to represent a strong hand. The opponent sees this bid and then might make a wrong guess about the player's presumably good cards. The player has changed from an optimal play to another strategy to trick his opponent and induced him to play a weak strategy.

1.4 Research Question

As pointed out in the sections 1.3.6, 1.3.8, and 1.3.9 there are at least three reasons why it is not possible to play games optimally in practice. Limited computing power and limited storage space are two problems which make an optimal solution practically infeasible to calculate for a game. The limited time to make a decision and the size of the game tree are the reasons for this. Furthermore, no

optimal solutions are available for games with hidden information, as there is always a guessing about the hidden information and strategies of the other players involved.

The target of this master's thesis is to investigate two approaches to find good solutions for such hard games where no optimal solution can be computed. The question in this thesis is if there results a difference in playing strength when the game trees are built and discovered in different ways. Therefore, two different types of approaches are compared. The first approaches (bandit approaches, see 3.5) use a flat but wider game tree whereas the second approaches (tree approaches, see 3.6) build a more structured and deeper tree.

To give an example, we imagine the following situation. A player has three options: option A, B, and C. The player must choose exactly two options from this list. The ordering of the options (for example, do A first and then B or the other way around) does matter. Therefore, there are 6 different possible combinations for his move: AB, AC, BA, BC, CA, and CB. To evaluate which choice is the best, he tries both approaches mentioned above. The structured approach to solve this problem is illustrated in Figure 9 and the flat, but wider method, is shown in Figure 10.

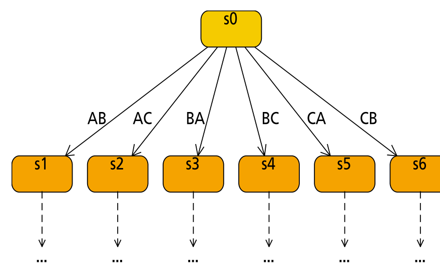


Figure 9: Illustration of the flat approach

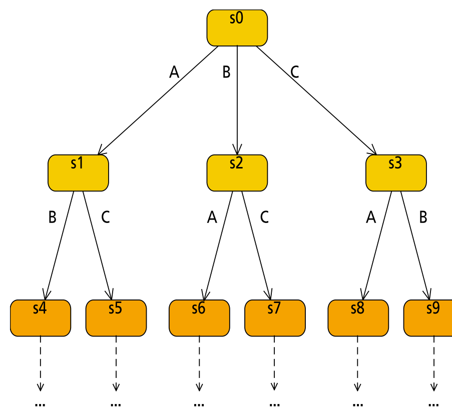


Figure 10: Illustration of the structured approach

In the flat approach, the player evaluates all possible combinations equally. The solution BA for example is evaluated in the exact same way as the move BC is considered. The important part here is that no knowledge is transformed from the former to the latter option as the two options do not share any structure or properties from the view of the flat approach. For this method, BA is not related at all with BC.

In the structured approach, on the other hand, the player gets to an intermediate step s_2 if he wants to evaluate BA or BC. Then, the player can incorporate that the game states s_6 and s_7 both are successors of state s_2 and therefore can transfer knowledge between the two moves BA and BC.

The hypothesis of this thesis is that the structured approach should be able to detect and use this difference. For example, if the player discovers in the flat approach that option BA is not a good move, he has not learned anything about BC. The structured method can detect that doing B is the reason why BA is not a good move and then conclude that BC is not a good move, too. With this reasoning it should be possible to get better results with the same computational effort compared to the flat method. But as we see in the scenario above, the flat approach has fewer nodes than the structured approach. This can be an advantage in bigger problems.

The goal of this thesis is to compare these two methods and find out if the structured approach has an advantage over the flat approach in a very specific situation: in the game of Hearthstone.

1.5 Similarities to Real World Problems

After the definition of games and how games can be used to define and measure intelligence we now give some examples how the gained knowledge of research can be transferred to real-world problems. Such examples show that games are not just a possibility to test theories of artificial intelligence. Many real-world problems like stock trading or political negotiations can be modeled as games. If we do this, we can directly apply algorithms which were originally implemented to play games to such situations.

In stock trading, there are, like in the games described before, a set of players which all aim to optimize a utility function. We can therefore say that these players are rational agents playing the game of stock trading. There is a fixed set of rules, too. The players can buy and sell stocks, bonds, and other securities. To do so, they have to invest money to buy stocks or obtain money by selling stocks. The utility function which should be optimized is the amount of available money after a given time. Furthermore, we not have only one player but many agents acting adversarial, since stock trading can be viewed as a zero-sum game. The agents also have hidden information about the companies in the stock marked which they do not share with the other participants. Finding out what strategies are followed by other players can lead to a significant advantage in obtaining more money than the other players. Furthermore, the price trends of the traded goods are depending not only on visible information. Therefore, this can be viewed as a source of randomness.

We see that the concepts in stock trading, like the price of the goods, the stock traders, and the outcome of activities, can be modeled as concepts already known in games. Therefore, methods which obtain good results in games may lead to good results in stock trading since the problems are strongly related.

2 Hearthstone: Heroes of Warcraft

In this chapter, we describe the game *Hearthstone: Heroes of Warcraft*, or short *Hearthstone*. We will start with a short introduction to the game and describe the course of a game in section 2.1. In the following sections further details are described.

2.1 Introduction

Hearthstone: Heroes of Warcraft is a relatively new, but already widely spread, online collectible card game. It was released on March 11, 2014 by Blizzard Entertainment. By September 2014, only six month after the release, Blizzard announced that there are over 20 million registered accounts worldwide. By January 2015 the limit of 25 million accounts was reached. This success has different reasons. First, the company Blizzard Entertainment is well known in the gaming scene. Since Hearthstone is a part of the Warcraft universe, many players from other games related to this genre are familiar with the concepts of the game. Furthermore, the game is free-to-play which means that creating an account and playing the game is free of charge. The basic rules are fairly easy to learn but it is assumed that mastering the game is a difficult task just as in other popular games like chess or poker.

In Hearthstone, two players play against each other. Both players act alternatingly in a round based manner. This means that in a given round during the game, only one player has to act whereas the other player is passive. The passive player can only observe the actions of the other player but cannot interfere with the play of the active player. A round in Hearthstone usually has a duration of 30 seconds. During this time, the active player has to make his decisions and play his moves. When the player finished his move he can actively finish his ply to let the other player play his move. When the 30 seconds are over the current ply is finished automatically and the next player has to act.

Each player is represented in the game by a hero. Before a standard game in Hearthstone begins, each player has to choose with which hero he will play. Each hero (there are nine different heroes available) has unique skills and therefore enables different strategies. The heroes have an initial health value of 30 health points. The goal of the game is to reduce the heath points of the opponent's hero to or below zero. The game is then immediately finished and the surviving player wins the game whereas the other player has lost it. If the heroes of both players fall below zero health points at the same time the game is tied. Therefore, the game is a zero-sum game as described in section 1.3.3. During the game, the players have to make a tradeoff between attacking the opponent's hero and protecting their own hero.

Each player has, in addition to the choice of the hero, to choose with which cards he wants to play. Each player chooses 30 cards out of a set of available cards in the game which build the players decks. Depending on which hero the player chose before, only a subset of the overall available cards in the game can be chosen. The cards have very different properties and effects. Spell cards, for example, can deal damage or heal the hero. Minion cards enable the player to summon minions which fight for their owners. They are the most important concept in the game of Hearthstone since they can deal a lot of damage and protect the hero in a passive way. If one player has a lot of minions summoned, the other player is forced to attack these minions before he can attack the hero. If he would not do this, the minions can deal a lot of damage in a few rounds and win the game. Weapon cards give the player another opportunity to deal damage in the game.

After both players have chosen their heroes and their decks the game starts with an initialization phase. At the beginning, the first player is drawn. The first player draws three cards and the second player draws four cards from theirs shuffled decks. Both players can then look at the cards and can decide for each card if they want to play with it from the beginning or if they want to draw another card and put the original card back into the deck. If a card is, for example, more useful in the late

game than in the beginning, it would make sense to put this card back into the deck and draw another card which could be more helpful in the beginning of the game. At the end of the initialization phase, the second player gets an additional special card, called *The Coin*. This card should lower the disadvantage for this player by acting second.

After the initializing phase is finished, the main phase of the game starts. We focus in this thesis on this main phase. The initial phase cannot be influenced by the players. The decks are either build at random from all available cards or a fixed deck is used. The players in the experiments in section 5 always take the originally drawn cards and do not put bad cards back in the deck. Since this holds for both players, there should not be a disadvantage for any player.

As described above, the main phase of the game is turn-based. In each ply a player has an amount of action points available which he can use to execute actions. Since the game plays in the context of a fantasy world, these action points a called *mana*. Playing cards and using the special ability of the hero cost a certain amount of mana. In the first round, both players have only one mana available. In each round, a player gets one additional mana point until the maximum of ten mana points is reached. The more mana is available the more strategic options the players have. In later rounds, with more mana available, more expensive cards can be played. These more expensive cards are usually stronger than the low cost cards which can be played at the beginning.

The goal of each round is to use the available mana, the available minion and spell cards, the summoned minions and equipped weapons strategically to damage the opponent's minions and the opponent's hero and therefore increase the winning probability until the game is finished. Revisiting the definition of intelligence from section 1.2, we can say that finding solutions for this task need a certain kind of intelligence. The more intelligent and therefore the more rational a system is the better should it achieve this goal.

2.2 The Game

In the following sections, we give an overview about the game and the elements of which a Hearthstone game consists of. In Figure 11 we see a screenshot of the game which gives a first impression of the game. The both players are on the bottom and the on top of the battlefield. The bottom player represents the own player and the top player is the opponent. We furthermore see the own hand cards whereas the hand cards of the opponent are not visible. In the center of the screen we see the battlefield with 3 minions. The upper two minions belong to the opponent and the lower minion belongs to our player. In the bottom right corner the amount if mana is displayed. In this screenshot, the player has already used two mana points and has still one mana point available. The opponent's hero has a weapon equipped which enables the hero to attack.

Figure 11: Screenshot of the game Hearthstone

2.3 Cards

Cards are the most basic element in the game Hearthstone. Both players have their own card deck of 30 cards from which they draw cards during the game. Furthermore, they have hand cards which are not visible to the other player. We will describe the three card types *minion*, *spell*, and *weapon* in the next sections and provide an example for each.

2.3.1 Minions

Figure 12: The Hearthstone minion card Stormwind Champion[1]

An example of a minion card is displayed in Figure 12. All cards have a cost value in the "currency" mana. This cost value is displayed in the upper left corner of each card. In the example in Figure 12 the card has a cost value of seven mana points. Since this card is from the type of minion cards, the card also has an attack and a health value. The attack value of a card is displayed in the bottom left corner of the card whereas the health value is displayed in the bottom right corner. In the example, the attack and the health value are six. Cards can also have additional text. The *Stormwind Champion* has

[1] "Stormwind Champion" licensed by Curse under CC BY-NC-SA 3.0 via http://hearthstone.gamepedia.com/Stormwind_Champion

the text “Your other minions have +1/+1”. The text describes additional abilities of the card. In this example, all other friendly minions will get an enhancement of 1 point on the attack and the health value.

Minion cards are used to summon minions onto the battlefield. When they are summoned they can be used to attack the opponent’s hero or his minions.

2.3.2 Spells

Figure 13: The Hearthstone spell card Fireball[2]

An example for a spell card is displayed in Figure 13. Like the minion cards, spell cards have a cost value which is displayed in the upper left corner. *Fireball* costs therefore four mana points to cast. Unlike minion cards, spell cards do not have an attack or a health value. The effect of a spell card is defined by its text. *Fireball* will apply the effect “Deal 6 damage”.

2.3.3 Weapons

Figure 14: The Hearthstone weapon card Perdition's Blade[3]

Weapons are the third and last card type available in Hearthstone. In comparison to minion and spell cards, weapon cards have an attack and a durability value. The attack value defines how much damage is caused when the weapon is used and the durability defines how often the weapon can be used until it is destroyed.

[2] “Fireball” licensed by Curse under CC BY-NC-SA 3.0 via http://hearthstone.gamepedia.com/Fireball

[3] „Perdition's Blade“ licensed by Curse under CC BY-NC-SA 3.0 via http://hearthstone.gamepedia.com/Perdition's_Blade

2.4 Abilities

Abilities are special effects of cards which give the cards some additional behavior on top of their basic abilities. For example, some cards trigger an extra effect when they are played, make minions immune against damage or heal a player. The exact characteristic of the abilities differs a lot from card to card. In the following, the different abilities are described.

Battlecry

A battlecry is an effect which is activated when a card is played by a player. Battlecries are typically an ability of minions, but weapons can also have the battlecry ability. Examples for battlecries are

- “Restore 4 Health to your hero”,
- “Deal 1 damage”,
- “Draw a card”,
- “Summon a 1/1 Murloc Scout”, and
- “Deal 6 damage randomly split between all other characters”.

Charge

Charge is an ability which lets a minion attack directly after it was summoned. In general, minions cannot attack in the same round in which they are summoned. Charge changes this and offers the possibility to deal damage immediately, which is a valuable feature in many situations.

Combo

A combo effect is triggered when the card with the combo ability is not the first card which is played in the current ply. A combo effect can improve the original effects of cards or add completely new effects. Examples for combo abilities are

- “Summon a 2/1 Defias Bandit”,
- “Give a random friendly minion +3 Attack”,
- “Deal 1 damage. Combo: Deal 2 instead”, or
- “Deal 2 damage to the enemy hero. Combo: Return this to your hand next turn”.

Deathrattle

Deathrattles enable minions and weapons to apply an effect when they are destroyed. The effect is not triggered when they were silenced before they die or are destroyed. Some examples for deathrattles are

- “Your opponent draws a card”,
- “Summon a 2/1 Damaged Golem”,
- “Deal 1 damage to all minions”, or
- “Equip a 5/3 Ashbringer”.

Divine Shield

A divine shield protects a minion from any damage. If a minion with divine shield is hit by an attacker, the divine shield is removed but the minion does not take any damage. The same applies when a minion with divine shield attacks another minion which has an attack value greater than zero.

Enrage

Enrage is an ability which improves a minion when it is damaged. If the minion is healed to full health, the enrage effect will be removed until the minion is damaged again. The enrage effect increases the attack value of a minion or a weapon. Common effects are "+3 Attack", "Your weapon has +2 Attack", or "Windfury and +1 Attack"

Freeze

Freeze is not a positive ability but also a negative one. Frozen minions and heroes cannot attack until the ply of the player to whom they belong is finished.

Immune

A character which is immune cannot be attacked or targeted by any spells or battlecries. The difference to the divine shield is that immune characters can be damaged by random effects and by effects which do not need a target like "Deal 2 damage to all enemy minions and Freeze them".

Overload

The overload ability reduces the amount of available mana in the next round by the stated amount. A player which is affected by overload cannot use the full amount of mana in the next round which can be a big disadvantage. In return, the player can use relatively strong cards with lower mana costs in the current ply.

Silence

Silence is an ability of minions or spells that can remove all effects from a minion. For example, it could be a good move to use silence to remove a divine shield of a minion or to remove enchantments, which were casted on a minion.

Stealth

Stealth is an ability preventing the minion to become the target of attacks, spells or other effects which need a target. This ability is active until the stealthed minion attacks. Then the ability is removed and the minion can be targeted like other minions.

Taunt

Taunt is a minion ability which forces the opponent to attack this minion first. Therefore, minions with taunt serve as a protection for other minions without taunt, and the hero.

Windfury

The windfury ability increases the number of attacks available in a ply of a minion. Instead of only one attack, which is common for minions, a minion with windfury can attack twice which enables the minion to deal a lot of damage if it is alive for a few rounds. But not only minions can have this ability. The windfury ability is also available for weapons.

2.5 Composing a Move of Atomic Actions

One important difference in Hearthstone in comparison to other games is the type of moves a player can make. Hearthstone is a card game with additional objects like minions and weapons. This means that a player can play cards, and use his minions, or weapons to attack. Therefore, a player cannot just make a single atomic move like moving a piece in chess or making a bet in poker. On the contrary, he can make many atomic actions until his mana pool is empty and all his minions are exhausted. Therefore, a move in Hearthstone can be composed of several atomic actions.

3 Finding Solutions with Monte Carlo Methods

In the third chapter of this thesis, we will now investigate methods enabling systems to play the game Hearthstone. Playing the game means a system has to select good moves out of the set of available moves.

We therefore combine in this chapter the abstract games modeled in section 1.3 and the concrete game Hearthstone from chapter 2. We use the concept of game trees to model all courses of the game in section 3.2. We furthermore show that the resulting game tree is too big to be searched exhaustively. We therefore focus on the move tree described in section 3.3 and analyze its size. We will see that the complexity of the move tree for a single game state is too big to be analyzed in detail even if it is only a small part of the complete game tree. We therefore have to choose what moves we will investigate in detail and what moves are likely so bad that we should not spend much computation time on them.

We will then describe the first approach, called bandit approach, to solve this problem in section 3.5. This approach will investigate complete moves in Hearthstone and will not be able to transfer knowledge between similar moves. This approach is the flat approach mentioned in the title of this thesis, since applying this approach during the game leads to a flat and wide game tree.

The second approach is described in section 3.6 and is the structured approach mentioned in the title. This algorithm builds the move tree out of the atomic actions available and therefore leads to deeper but also not so wide game trees as the flat approach.

Before we introduce the game and move trees for Hearthstone, we describe a very important problem besides the size of the trees in section 3.1. In this section, we describe that we cannot easily determine the value of a move and therefore will use simulations to estimate the value of actions and moves.

3.1 The Need for Simulations

As written in this chapter's introduction it is not an easy task to determine if a move or even an atomic action is good in the sense of leading to a desirable outcome in the game Hearthstone. For example, in Hearthstone we can have the possibility to play an offensive or a defensive move. This could be attacking the opponent's hero or protect the own hero by playing a taunt minion. Another example would be if we have to choose between dealing instant damage by casting a spell on the opponent or rather equipping a weapon to the hero to be able to deal more damage in the long run. In these examples, we have to choose between two options. Therefore it would be sufficient to know which of the two was the better one. But during the game we will face a lot of concurrent options. In this case we would give each option a fixed value to describe how good this move is. We then could just select the best move and play it. This is possible in strongly solved games. There, we can determine for each game state which move will lead to the best possible outcome.

In games where this is not possible, we can use a heuristic which gives us an estimate how good a move is. For example, in chess we can assign each piece on the board a value [5]. If we summarize them we will get a notion about how probable it is to win the game. For example, we can give a queen nine points and a rook five. This means a game state where we have two rooks is slightly better than a game state with only one queen. If we have to choose between a move which will lead us to a game state where we either have one queen or two rooks, this heuristic tells us that we should prefer the game state with the two rooks as their points summarize to ten. But not only the numbers and types of pieces on the chessboard are important. The position of the pieces is often the major factor which determines whether a player wins or loses a game. The heuristic that only counts the points of the pieces does not include this factor and therefore will not lead to the best game state in every situation. Since heuristics only consider parts of the whole knowledge, it does not always provide us with the

right answer. Therefore, heuristics are wrong from time to time and not as good as the predictions of completely solved games.

But there are also games where it is difficult to even formulate a heuristic to roughly rate the moves. In the game Go, for example, this is the case [6]. In such games, no weighted linear combination of a simple feature can easily be handcrafted to estimate the outcome of a game state accurately. The same holds for the game Hearthstone. It is difficult to say that, for example, a special minion can be rated with a specific point value. The value of each minion depends on other minions, synergy effects with other cards, the status of the opponent's minions, and a lot more. Since we cannot estimate a value of a game state precisely enough with a heuristic, we will use another approach to rate them. To estimate the outcome of a game state, we will use random simulations which simulate the game further until an end game state is reached. This end game state can easily be rated, since for all game end game states there is per definition a utility value defined in the game rules. If we simulate a lot of games in such a way, the average outcome of these simulations should estimate the value of the game state. Since we use random simulations to estimate the value of a game state, such approaches are called Monte Carlo algorithms.

3.2 Game Trees in Hearthstone

In section 1.3, the concept of game trees was explained. As the game Hearthstone is a sequential, two-player zero-sum game with randomness, hidden information and limited time in the sense of sections 1.3.2, 1.3.3, 1.3.7, and 1.3.8, it is theoretically possible to construct a game tree for this game. We skip in the following sections the initialization phase of the game, as those details are not in the scope of this thesis, and always assume that the game is already in the main phase. A game tree in the main phase will look like pictured in Figure 15.

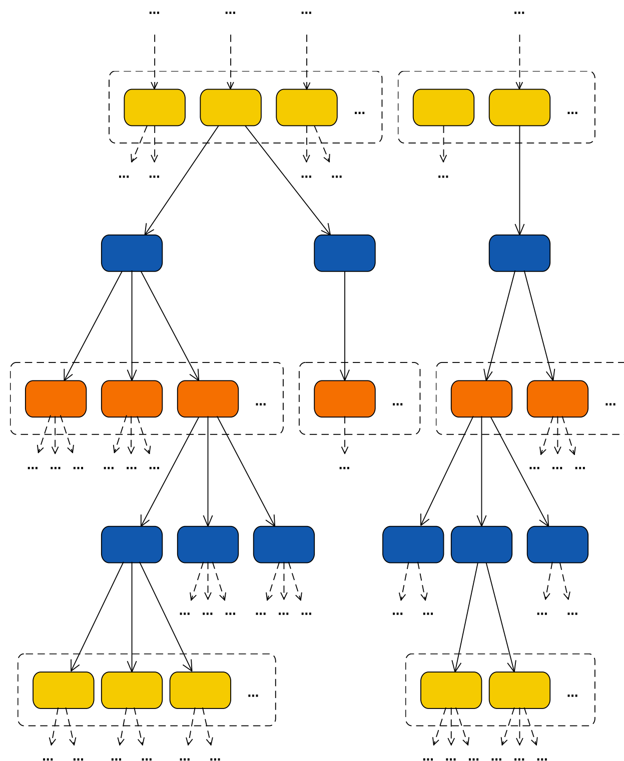


Figure 15: A snippet of a game tree for the game Hearthstone

Assume the game has just started and the first player who is, like before, called MAX has to make his first move in the first ply. At this early point, MAX is already in a set of enormously big information partitions because he and his opponent MIN have already hidden information. At first, there are the hand cards of MIN: MIN has in this situation four cards in his hand which are not visible for MAX. As the ordering of the cards does not matter, we can calculate the number of possibilities using the binomial coefficient $\binom{n}{k}$, where n equals the number of available cards in a set and k equals the number of cards which are drawn from the set. As each card can occur twice in a player's hand and the number of available cards for each player to choose from to form his deck equals approximately 125 cards we set $n = 250$. We further set $k = 4$ as four cards are drawn from the set and get roughly $1.6 * 10^8$ possibilities for the hand cards of MIN. Furthermore, MIN knows which cards are still in his deck, which is also hidden information of MIN. As MIN has already drawn four cards, there are only 26 cards in his deck. Since neither MIN nor MAX know the ordering of the cards in the deck (or in other words: there is no ordering until it is defined by a move of nature), the ordering is no hidden information. Therefore, the number of possibilities can again be calculated with the binomial coefficient. As result we get approximately $1.4 * 10^{35}$ possibilities for the deck of MIN. As the number of hand card combinations is negligible in comparison to the number of deck combinations we can say that MAX's set of information partitions consists of about $1.4 * 10^{35}$ possible game states in each partition at the very first move. Given MAX has not played a card, MAX will have at this point three hidden cards in front of MIN, which will result in $2.6 * 10^6$ possibilities, and 27 cards in his deck resulting as above in $1.2 * 10^{36}$ possibilities for MAX's deck. This means that at this early point in the game, we have about $1.2 * 10^{36}$ partition sets for MAX with $1.4 * 10^{35}$ game states in each set. This summarizes to a total number of $1.7 * 10^{71}$ game states in the first layer of the game tree. To get a feeling for such a big number, we compare this to the total amount of possible game states in chess, which is about 10^{47} (see Table 2 in section 1.3.6 for more details).

In the example in Figure 7 in section 1.3.8, no hidden information was revealed by the moves of the two players. As this was assumed, all the partition sets of a player in a fixed ply consisted of the same possible moves. In a real situation like in the game Hearthstone a player can reveal some of his hidden information for example by playing a card. Than the information is visible to all players and is no longer hidden. Therefore, it is now possible that different information sets of a player have a different set of possible actions. The other player than can reduce the number of indistinguishable game states in the information partitions.

After MAX has made his move, for which only a few or just one action at all will be available in the first ply because of the low mana amount and the absence of minions on the board, and finished the ply, CHANCE will be next to act. Assuming that MAX has only finished his ply without making other actions, there will be as many game states in the second layer of the game tree as in the first one. CHANCE acts, as described in 1.3.7, according to a given probability distribution, and will draw a card out of the deck of MIN and put the drawn card in MIN's hand. The probability distribution will be a uniform distribution in every case when cards are drawn which is known by both players. As 26 cards are available in the deck of MIN, the branching factor at this game state would be 26 for each possible game state. This would result in a number of $4.4 * 10^{72}$ possible game states in the third layer of the game tree.

After the move by nature was made, MIN has to act. He is in an analogously situation as MAX before and will have only a few possibilities to act. During the game, the number of possible moves will increase in comparison to the first plies due to the increasing mana amount of both players and the increasing amount of minions on the board. Therefore, the branching factor at the player nodes will increase as well. This is described in detail in the following section.

3.3 Move Tree Complexity

As mentioned in 2.5, a move in Hearthstone consists of multiple atomic actions. These atomic actions can be combined and ordered in many different ways.

If we assume, that each atomic action can be independently played without being influenced by random effects, we get an impression for the number of possible action sequences. This number is not fixed in every situation, because some actions depend on each other. For example, drawing a card in the middle of a ply can provide new possibilities to act where using the last amount of mana available in a ply can limit the options. But the calculated number provides an impression of the magnitude of possible action sequences. An example of the deviation from the upper bound is given below. In the following part, we do not explicitly model the action "finish ply". Therefore, for example "having no action to perform" means that the player can only finish the ply.

If a player does not have any actions to perform in a game state the only possibility is to stay in this game state and finish the turn. Therefore only an empty action sequence is possible to play in the ply.

If only action A is possible in the current game state s_0, a player has two options for a reachable game state. He can do nothing and will stay in the current game state, which is equal to an empty action sequence. One reachable game state therefore is s_0. The second possibility is to do action A and get to game state s_1, which is the second reachable game state for the player. The action sequence then would be A. He therefore has two possibilities for a reachable game state and therefore two possible action sequences to choose from as shown in Figure 16.

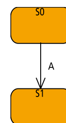


Figure 16: Possible game states reachable with one action available

If there are two actions A and B available the player has to choose from five possible action sequences. As in the upper case, he can do nothing performing an empty action sequence and stay in game state s_0. He also has the possibility to perform action A and then finish his ply. This behavior would end up in game state s_1, as shown in Figure 17. Performing action A and then B end up in game state s_3. Analogously performing action sequence B ends up in game state s_2 and performing action sequence B-A in game state s_4. Therefore five different action sequences can end up in five different game states.

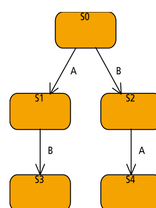


Figure 17: Possible game states reachable with 2 actions available

The possible action sequences with three actions available are shown in Figure 18. Each game state from s_0 to s_{15} is a valid reachable game state and therefore a player has 16 possible action sequences to choose from. For example, the action sequence B-A-C would result in game state s_{12} and C-A would result in s_8.

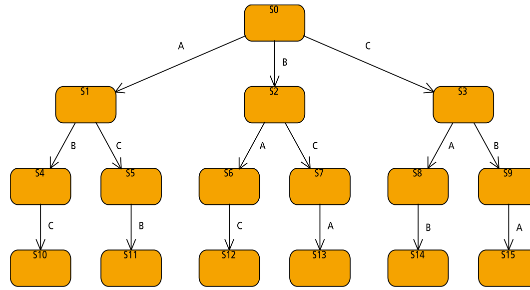


Figure 18: Possible game states reachable with 3 actions available

As we can see, the reachable game state tree for one possible action consists of one time the game state tree for zero possible actions plus the initial state; the game state tree for two possible actions consists of two times the game state tree for one possible action plus the initial state; the game state tree for three possible actions consists of three times the game state tree for two possible action plus the initial state and so on. This is analogously true for each number n of possible atomic actions. It holds, because for the first decision are n choices possible. If the player has chosen one of the possibilities he reached a game state where he has only $n-1$ possible choices to perform out of the n initial choices. Therefore, he has n times the tree for $n-1$ atomic actions. He can furthermore always stay in the initial game state. Therefore, we can derive a recursive function $f(n)$ defined as

$$f(n) = \begin{cases} 1;\ \text{for } n = 0 \\ n * f(n-1) + 1; \text{for } n > 0 \end{cases}$$

for the amount of possible reachable game states respectively for the amount of possible action sequences for n possible atomic actions. Another non-recursive equal definition would be

$$f(n) = \begin{cases} 1;\ \text{for } n = 0 \\ n! + \sum_{i=1}^{n} \frac{n!}{i!}; \text{for } n > 0 \end{cases}$$

To get an impression of the enormous amount of action sequences which are possible with n independent atomic actions the first 15 values of $f(n)$ are listed in Table 3.

n	$f(n)$	$\log_{10} f(n)$
0	1	0
1	2	0.301
2	5	0.699
3	16	1.204
4	65	1.813
5	326	2.513
6	1,957	3.292
7	13,700	4.134
8	109,601	5.040
9	986,410	5.994
10	9,864,101	6.994
11	108,505,112	8.035
12	1,302,061,345	9.115
13	16,926,797,486	10.229
14	236,975,164,805	11.374

Table 3: Amount of possible action sequences for up to 14 atomic actions

3.4 Exploration/Exploitation Tradeoff

As explained in section 3.3, the move tree for moves in Hearthstone can be enormously large; too large to be built completely. In addition to the large tree, the game states which are reached by the moves cannot be evaluated with a heuristic function as described in section 3.1. The combination of these two problems leads us to the *exploration/exploitation dilemma*. If we want to find the best possible move from the move tree, we have to explore the tree as much as possible. But exploring the tree is not sufficient to find a good move. We also have to evaluate the nodes in the tree sufficiently good to get a precise estimate of the value of a move. Due to computational limitations, we cannot explore the whole tree exhaustively and evaluate each node with simulations frequently enough. We therefore have to choose which parts of the tree we want to investigate. Since we want to find a good move in the move tree, we are interested to investigate the promising parts of the tree in detail and not to spend much time with bad moves in the unpromising part. We have to choose if we want to investigate good moves further and get a better estimation of the utility value of the promising moves or if we want to explore new parts of the move tree to maybe find even better moves than the current best ones. This tradeoff between the extensive exploration of the move tree and the exploitation of the knowledge about the promising parts of the move tree is called the *exploration/exploitation tradeoff*.

We now introduce two approaches in detail which try to handle the exploration/exploitation tradeoff. The first one is the flat approach as mentioned in the initial research question of this thesis in section 1.4. It is called *bandit approach* and will only investigate complete moves which correspond to paths in the move tree introduced in section 3.3. The structured approach, called *Upper Confidence Bound Applied to Trees*, will, on the other hand, investigate the structure of the move tree in detail. For that, he will investigate not the complete paths in the move tree but the single actions made to construct the paths.

3.5 Bandit Approaches

The first approaches to handle the tradeoff between exploring new moves and exploiting the knowledge of already investigated moves, which we will investigate in detail, are called *multi-armed bandit approaches*, or short *bandit approaches*. The notion of the name is explained in the following.

Imagine a classical slot machine. To play with such a machine, it is necessary to put a coin in the coin slot (therefore the name slot machine) and start the game. If the game is started, there is no possibility to influence the game. This means there is no further strategy required to play the game. The result of a game with a slot machine is a single reward. This reward is determined with a stationary probability distribution which is unknown to a new player. After playing the first game, the player can evaluate the reward and can consider playing again. As he plays more often, he gets to know the machine and can estimate the probability distribution with each round better. Therefore, he can estimate associated values like the expected value or the variance of the probability distribution.

This situation relates to the situation of an evaluation of a single move in Hearthstone. A player can make a move and get a reward for that move. As described in section 3.1, we cannot measure the value of a game state exactly. Hence, we simulate a rollout to the end of the game and evaluate the result of the game. This rollout can of course have different outcomes as the players might play randomly or other random effects are involved. Therefore, we get as result either a value of $+1$ for a win or -1 for a loss. The outcome of a simulation is therefore a random variable with a probability distribution like in the slot machine example above. The expected value tells us the expected result of an average game. A value of -0.3 for example means that if we play the game, we will expect that the game will end with a reward of -0.3 in average.

To come back to the slot machines, we will now assume that we do not have a single slot machine to play on but more than one, maybe hundreds or thousands of machines. But the machines are not equal. Each machine has its own probability distribution and therefore its own expected value and variance. The goal is now to find out which machine we should play to get the biggest reward. As we can only play one machine at a time, we have to choose one of the slot machines to play on. The first choice is easy for a new player: Since he does not know anything about the machines, the best strategy for the player is to just play a random machine. He will get his reward as described above for the game. But now the decision which machine to play is not so easy anymore. Should the player play the very same machine again? Or should he switch to another, unknown machine? Assume that the player got a high reward in the first play. He may tend to play the same machine again, as the result was beneficial. But maybe the result was only an exception. Maybe the true mean of the underlying random distribution is much smaller (or even negative if the wager is subtracted) and the high reward was a result of a high variance. The player is now in a situation called *Exploration/Exploitation-Dilemma*. On the one hand, the player wants to exploit his learned knowledge about the machines he played on earlier. But on the other hand, he also has to explore the other slot machines to find out if there are machines that will perform better.

The situation with a lot of slot machines now relates to the situation in which a player can make a lot of moves in Hearthstone. A player now has to choose which moves he will evaluate further. He can either evaluate moves, which seems to be a good choice to play and exploit his previous learned knowledge of the moves, or he can explore other moves where the uncertainty about the real outcome is bigger. Maybe one of the so far less investigated moves will be the real, overall best move. Due to the limited time and the simulation of games being time consuming, the player has to decide which strategy he will use to find out which move will be the best.

A concrete algorithm, which is explained in detail in section 3.5.1, makes a lot of iterations of the method described above to find out which move is the best one to actually make. A generic algorithm is given in Listing 3.

```
function banditAlgorithm(possibleMoves, iterations):move
{b_1, ..., b_i} ←generateBandits(possibleMoves)
for i = 1, 2, ..., iterations do
    b ←selectBandit({b_1, ..., b_i})
    r ←playBandit(b)
    b ←updateBandit(r)
return bestBandit({b_1, ..., b_i}) → move
```

Listing 3: Pseudocode for a generic bandit algorithm

The algorithms' inputs are the possible moves which should be analyzed and a number of iterations which indicate how many times the algorithm should play a bandit. This latter parameter could be changed not to give a fixed number of iterations to make but a time how long the algorithm should play bandits. Another alternative would be to start the bandit algorithm without assigning any indication when to stop and interrupt the algorithm at any arbitrary time. This property is called *anytime property* and is explained in 3.5.3 in detail.

The algorithm then creates the actual bandits $\{b_1, \ldots, b_i\}$, which means that additionally to the moves some other values are stored. This could be for example a number that denotes how many times a bandit was chosen for evaluation or results of the simulations so far. After initializing the bandits, one of the bandits b is selected. b is then played and a reward r is obtained. Playing a bandit in Hearthstone means simulating a game to the end and getting the reward by losing or winning the game. This simulation can be done in different ways which is explained in detail in section 3.10. The reward r is then used to update the bandit b. This means, the next time the algorithm has to select a bandit, it has more knowledge about the bandits and can use this knowledge gain to make the next decision. As described above, this could mean that the algorithm will play another bandit if the last iteration resulted in a bad reward or that the algorithm will investigate the last played bandit further because the result was good and he wants to be more certain about this outcome. In the last step, the best bandit is selected and the move which is associated with this bandit is returned. Since all bandits are handled equally and only complete moves are considered, the structure build by these algorithms will look like the flat approach in Figure 9 on page 16.

In the following section, we describe an algorithm to handle the exploration/exploitation-dilemma It gives an answer to the question which move should be investigated during the next run of the simulation. This means, it mainly implements the `selectBandit` method responsible for choosing the bandit that should be investigated next.

3.5.1 Upper Confidence Bound

The *Upper Confidence Bound* [7] (*UCB)* uses an intelligent, adaptive method to consider each bandit in each iteration and to select the one further to be investigated. To do so, UCB calculates the so called UCB values for each bandit and then selects the bandit with the highest UCB value. We define the UCB value as the UCB1 value defined in [8] with an additionally parameter c.

To do so, we set the UCB value for bandit j to

$$\bar{x}_j + c * \sqrt{\frac{2 * ln(n)}{n_j}}$$

where

- $\bar{x}_j$ is the average outcome observed for bandit j,
- n_j is the number of times bandit j has been evaluated already,
- n is the overall number of iterations made, and
- c is, as mentioned above, an additional parameter to control the exploration/exploitation tradeoff.

For $n_j = 0$ we define the UCB value as $+\infty$.

The UCB value balances the tradeoff between exploration and exploitation. A high value for $\bar{x}_j$ encourages the exploitation of good moves, as $\bar{x}_j$ is the average outcome of a bandit j. The better a bandit j performed in the past, the more probable it should be that it is played again, as this could be a good move of which the outcome should be estimated fairly accurately. The second addend encourages exploitation as it describes how often a bandit was played in the past in relation to the other bandits. Therefore, it represents the uncertainty of the value of $\bar{x}_j$. The parameter c controls the exploration/exploitation tradeoff, with which it can define how the ratio of exploration and exploitation should be. With a bigger c the algorithm tends to be more explorative.

In the following two figures, we see a visualization of the UCB value for the two bandits, Alice and Bob. Alice has a win rate of 0.70 and Bob has a win rate of 0.30. These win rates are unknown for the algorithm. The visualizations show the behavior of the algorithm during 200 iterations. In every iteration, the algorithm has to choose which bandit should be simulated. The algorithm chooses the bandit with the highest UCB value in every iteration. We see in Figure 19 that, after a few simulations, the UCB value of the both bandits is nearly equal. But we also see that this is achieved by choosing the bandit Alice more often than the bandit Bob.

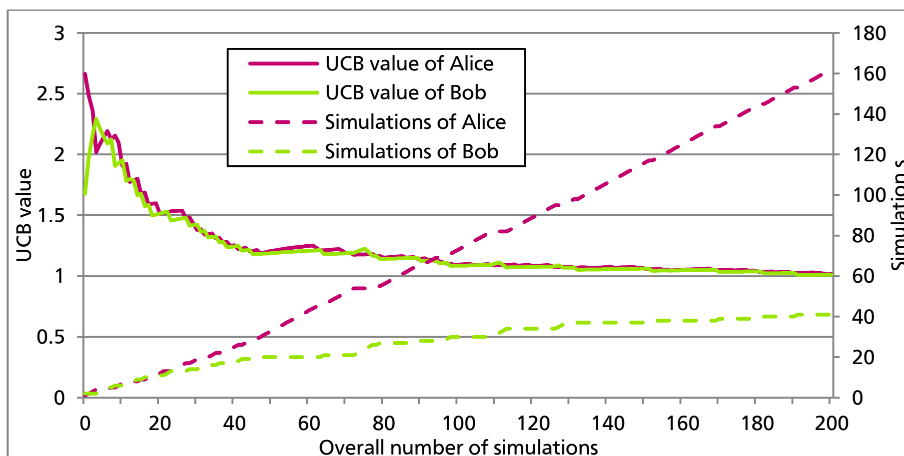


Figure 19: Diagram of the relation between the UCB value and the number of simulations for each bandit

Figure 20 shows the two different components of the UCB value. The expected outcomes of both algorithms converge against their true outcome of 0.70 for Alice and 0.30 for Bob. The exploration

value of the better bandit Alice is after a few simulations lower than the exploration value of Bob because Alice is more often simulated than Bob during the execution of the algorithm.

We can see in these diagrams nicely, how the expected value of a bandit affects its exploration in the UCB algorithm.

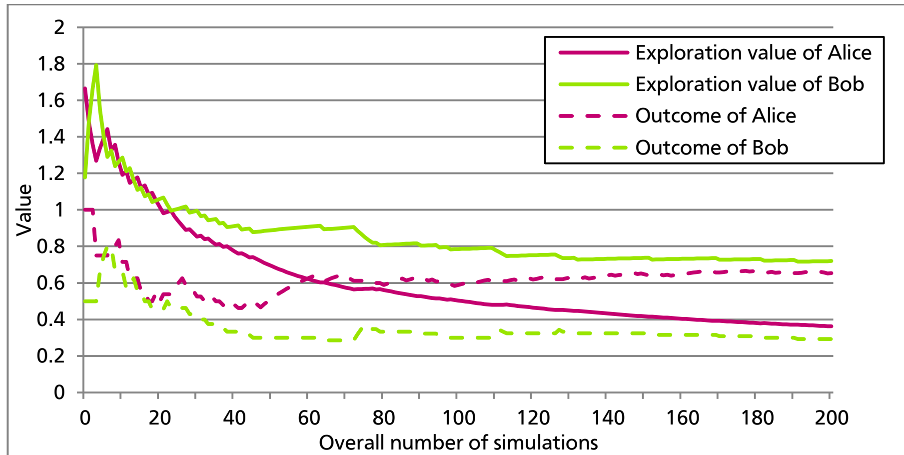


Figure 20: Diagram of the two component of the UCB value: the exploration and the exploitation

Listing 4 shows a pseudocode of the UCB algorithm. We see the implementation for the iterations of the algorithm in the function `ucbAlgorithm` and the calculation of the UCB value in the function `selectBandit`. The function `updateBandit` populates the results of a simulation returned from the generic function `playBandit`. `bestBandit` selects the bandit with the highest win rate at the end of the algorithm.

```
function ucbAlgorithm(possibleMoves, iterations):move
{b_1, ..., b_i} ←generateBandits(possibleMoves)
for i = 1,2,...,iterations do
   b ←selectBandit({b_1, ..., b_i})
   r ←playBandit(b)
   b ←updateBandit(b,r)
return bestBandit({b_1, ..., b_i}) → move
```

```
function generateBandits(possibleMoves):bandits
B ← ∅
for each move ∈ possibleMoves do
```

$$B \leftarrow B \cup \left\{ \begin{pmatrix} move \\ \bar{x} \leftarrow 0 \\ n \leftarrow 0 \end{pmatrix} \right\}$$

```
return B
```

```
function selectBandit({b_1, ..., b_i}):bandit
```

$$\textbf{return } \operatorname{argmax}_{b \in \{b_1, \ldots, b_i\}} \left(b \rightarrow \bar{x}_j + c * \sqrt{\frac{2 * ln(n)}{b \rightarrow n_j}} \right)$$

```
function updateBandit(bandit,reward):bandit
```

$$bandit \leftarrow \left\{ \begin{pmatrix} bandit \rightarrow move \\ bandit \rightarrow \bar{x} * \frac{n-1}{n} + r * \frac{1}{n} \\ bandit \rightarrow n + 1 \end{pmatrix} \right\}$$

```
return bandit
```

```
function bestBandit({b_1, ..., b_i}):bandit
```

$$\textbf{return } \operatorname{argmax}_{b \in \{b_1, \ldots, b_i\}} \left(b \rightarrow \bar{x}_j \right)$$

Listing 4: Pseudocode of the UCB algorithm

3.5.2 Final Move Selection

In Listing 4, a concrete implementation of the `bestBandit` function is given. In this version we select the bandit with the highest expected outcome after the simulation loop has terminated. This selection criterion is called *max reward bandit* as it selects the bandit with the maximum reward. But this does not have to be the best choice. The approximation of the expected value can be very inaccurate if we have only executed a few simulations. Then there might be bandits which are only visited one time. If they get a win as simulation result they have the best possible estimated win rate of 1.0 and are likely to be chosen as the best bandit. In such a situation with few simulations or a lot of bandits it might be better to use another selection criterion. Therefore, we describe two additional selection criteria to select the best bandit in the `bestBandit` method.

The bandit with the most visits is called the *robust bandit* since the expected outcome of this bandit is the result of more simulations in comparison to the other bandits. Therefore, the robust bandit selection criterion prefers a bandit which has an expected outcome of 0.8 based on 100 evaluations over a bandit with an expected outcome of 0.9 based on only 10 evaluations.

The *max wins bandit* is the third method to select the best bandit. This criterion chooses the bandit which has gathered the most wins during the execution of the algorithm. Therefore, it does not matter if a bandit has 10 wins and 0 losses or 10 wins and 10 losses. Both bandits are equally good from the viewpoint of this criterion. But since the UCB algorithm selects the first bandit more often than the

second bandit the constellation with a 10/0 and a 10/10 bandit is not possible at the end of the simulation phase. The first bandit would have been visited more often before a 9/10 or a 10/9 bandit will get another evaluation. Therefore, we will use this as default for the experiments in section 5.

We show the isometrics of the three different best child selection criteria in Figure 21. All bandits on each line are rated equally good. The max reward criterion is displayed in orange, the robust criterion is purple and the max wins criterion is green.

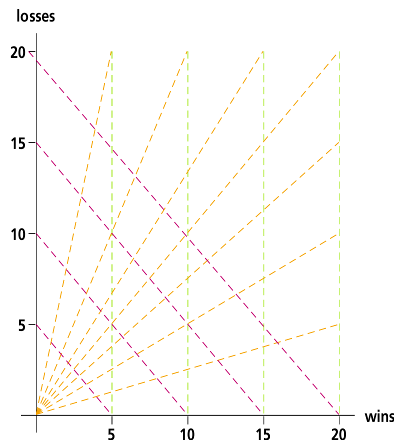


Figure 21: Visualization of the isometrics for the different UCB best child selection criteria

3.5.3 Anytime Property

As suggested in the introduction to bandit approaches in 3.5, the algorithms of this family share a common property, the *anytime property*. Anytime algorithms [*9*], also called *interruptible algorithms,* can return a valid solution for a task at any time. In comparison to classical algorithms, they do not have a contract on which result should be delivered at which input. Rather than this, they can return different solutions for the same input, depending on how much time they have to compute the result. It is expected that they find better and better solutions the longer they run. They can be used to approach the optimal solution in problems where the search space is too big to find the best solution, like in the *Traveling Salesman Problem* [*9*], or in problems where no practical computable optimal solution exists. For this, the Newton–Raphson Method [*10*] for finding the square root of a number is a good example. As it is expected that the bandits used in the algorithms above reach the real expected value of a game state only after an infinite amount of iterations, these algorithms are related to the latter type of problems, where anytime algorithms are useful. In Hearthstone, there is a limited time to evaluate which move would be leading to the best result. In such a situation, an algorithm with the anytime property is desirable as one can allocate as much time as available to the algorithm and retrieve the most accurate result possible.

3.5.4 Aheuristic Property

The bandit approaches in this section have another important property besides the anytime property. Since the bandit approaches are using random simulations to estimate the value of a bandit, they do not use heuristics which assign utility values to the bandits. This is called the Aheursitic property. As described in 3.1, this is a very important property to be able to estimate the value of bandits when no good heuristic was found to do this.

3.6 Monte Carlo Tree Search Approaches

In this chapter, we describe another family of algorithms to find a tradeoff for the exploration/exploitation-dilemma called *Monte Carlo Tree Search* (*MCTS)* [*11*]. The bandit algorithms described in section 3.5 consider always complete moves in Hearthstone. They consider the moves separately and have no ability to transfer learned knowledge to other bandits. But the moves in Hearthstone can be decomposed into atomic actions. Two moves may share a single atomic action or even many atomic actions. If one of these actions can be considered as really good or bad, this could be transferred to all moves that contain this atomic action. The tree approaches described in this section may be capable of exploiting this characteristic of the moves. As they do not see the complete moves but the atomic actions producing the moves, the assumption is that they can perform better than the bandit algorithms by using this information. The structure of these tree approaches will be, as the name suggests, a tree and will look like the structured illustration in Figure 10 on page 16.

The algorithms in this chapter build a move tree during the execution like described in section 3.3. The exploration of the progressively built trees is guided by the results of the previous exploration of that tree. Results are obtained, like by the bandit approaches, with simulations, which should predict the outcome of the moves. Further information about this can be found in section 3.10. During the runtime of the algorithms, the evaluation results of the single moves are supposed to become more precise since more simulations can be executed for the moves.

Since the move tree may be too big to be built completely, big move trees are only built partially. For the bandit approaches in section 3.5, it is necessary to roughly stop the generation of more bandits (more complete moves) when the available storage space is exhausted. If the storage space is exhausted, MCTS methods also have to stop the generation of more tree nodes. But as the tree built so far was built with the goal to explore promising parts of the tree at first, the probability is higher that more bad moves are not generated. In the bandit approaches, the truncation of moves does not have this property. There, moves are built independently of their suitability. Good and bad moves are therefore built equally until the threshold of available space is reached.

Just like the bandit approaches in section 3.5, the MCTS algorithms can run as long as some computational budget is reached and stop at any time. Therefore, the algorithms also have the anytime property described in 3.5.3. Since they do not need heuristics to evaluate moves, they have the non-heuristic property described in section 3.5.4, too.

The MCTS algorithms apply four different steps per iteration: selection, expansion, simulation, and backpropagation. These different phases are explained in detail below. We will also use an example to demonstrate the effects of each step. To do this, we will use a move tree consisting of three atomic actions A, B, and C. The player furthermore always has the option to finish his ply by executing action F. To be able to illustrate the process best, we will skip the first iterations and start with the partially plotted move tree in Figure 22. To get to this state, the algorithm has already made 13 iterations.

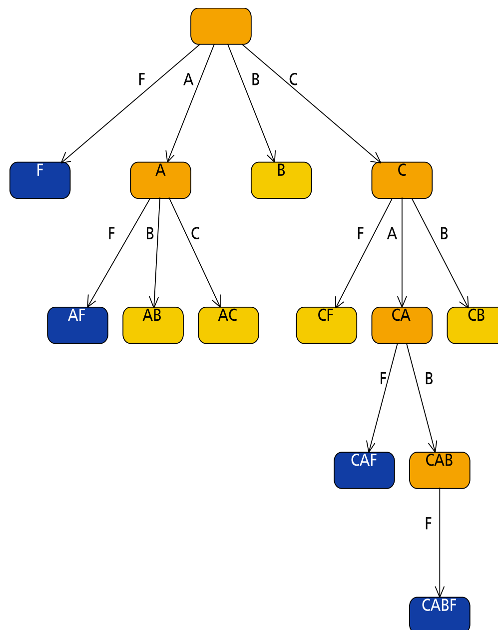


Figure 22: Initial move tree used to demonstrate an MCTS iteration

It is important to say that we use a slightly modified version of the classical MCTS approach to evaluate moves in Hearthstone. A standard MCTS algorithm would build the complete game tree if enough iterations were executed. By doing so, such an MCTS algorithm can reach true final game states in which the game is finished and the winner can definitely be determined. Since we are building the move tree for a just single ply without switching the player, we will not reach a true terminal game state. Instead, we will reach game states in which the player cannot make an additional action. This could be the case for example because of the absence of further cards to play or when the player does not have enough mana to proceed. The move tree is at such a point complete and not further expanded by switching the player. This means, game states reached by expanding the move tree will not be terminal states and therefore will not have a fixed outcome. The outcome of such a game state will instead be only an approximation obtained by running simulations. We will in the following part mainly describe the to this circumstance adapted version of the algorithm. When we differ from the original MCTS approach we will state this and also explain how the original approach would have worked.

3.6.1 Selection

In the first phase of an MCTS algorithm, called *selection phase*, a node to be further investigated is selected. This node should be something like the most urgent node to analyze. Therefore, the algorithm in this step faces the exploration/exploitation-dilemma explained in 3.4. To deal with this situation, the algorithm will implement a utility function which indicates the node to be selected.

The algorithm starts at the root node and descends recursively through the tree. There are three possible states in which each node can be. The node can either be

a not fully expanded or completely unexpanded and non-terminal,
a fully expanded and non-terminal, or
a terminal node.

The first possibility holds for game states, in which atomic actions are available, which have not been further evaluated in this game state. If the algorithm runs its first iteration, this will hold for the root

node of the move tree, since no actions have been evaluated at all. In a fully expanded but non-terminal node all possible actions will have been evaluated at least once. This means, that for all available actions in the corresponding game state a child is attached and it is not possible to add more children to this node. In a terminal node, which is the last possibility, no further actions are possible. A terminal node has therefore no children. In the case of a move tree, each usage of the action “finish ply” (indicated with the transition F) will lead to a terminal node and each terminal node will only be reached by executing this action.

As mentioned above, the algorithm can encounter three possible states of nodes during the recursively descend in the selection phase. If the current node is a fully expanded, non-terminal node, the algorithm chooses one of the children and continues the descending. This selection is the crucial point in this step, since it guides the algorithm through the tree. The method used for the selection is called *tree policy*, as it is used to navigate through the tree. Since this policy is one crucial part, influencing the performance of the algorithm essentially, we will have a closer look at one special tree policy that uses the *Upper Confidence Bound* already introduced in section 3.5.1. If the selection step faces a non-terminal, not fully expanded node or a completely unexpanded node there are two possible options available: The algorithm can either select this node and go to the next step of the algorithm, or it can choose an already expanded child, if there are any, and continue the descend. The tree policy mentioned above is responsible for this decision, too. If the node is fully unexpanded, the selection phase will terminate. The last possibility is facing a terminal node. In this case, the selection phase also terminates and goes to the next phase. To continue the example in Figure 22, we show in Figure 23 how a node is selected. In the first level of the tree, at the root node, the tree policy will choose a child because the node is non-terminal and already fully expanded. Action A is chosen by the tree policy in this example. After selecting the child A, the selection policy faces another fully expanded node. Again, the policy has to choose one action and decides to choose action C, which leads to node AC. This node is a non-terminal node, which is not expanded at all. Therefore, the selection phase will terminate at this node by selecting the node AC and the MCTS algorithm continues with the next phase, the *expansion phase*.

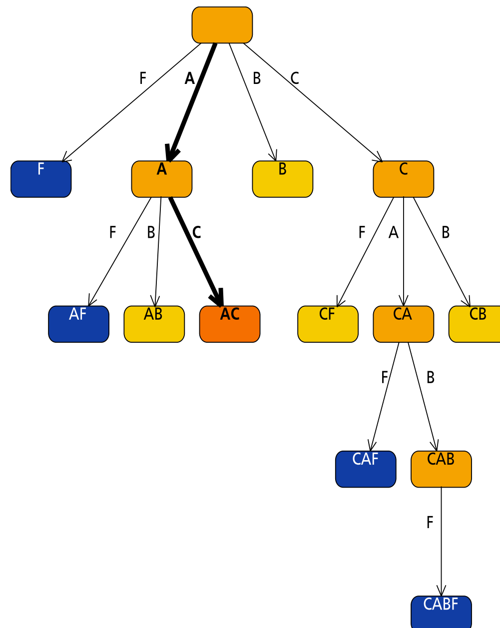


Figure 23: Visualization of the MCTS selection phase

3.6.2 Expansion

In the second phase of an MCTS algorithm, the *expansion phase*, the previous selected node is expanded. The node can only be in two different states at this point. The node can either be

- a not fully expanded or complete unexpanded and non-terminal, or
- a terminal node.

A fully expanded and non-terminal node cannot occur at this step because the selection phase would have continued descending and therefore would have selected another node.

If the node is not fully expanded or completely unexpanded, the expansion step will expand one or more nodes, according to the available actions in the selected game state, and will add the created nodes to the move tree. The moves for which new nodes should be created will be selected at random, because there is no knowledge about the moves and the resulting game states available. As illustrated in Figure 24, node AC is expanded by creating node ACB and the new node is attached to the selected one. If the node is extended, the expansion step is completed and the next phase, called *simulation phase*, starts with the newly created node. In the case of a terminal node, the expansion phase does nothing and just continues with the simulation phase.

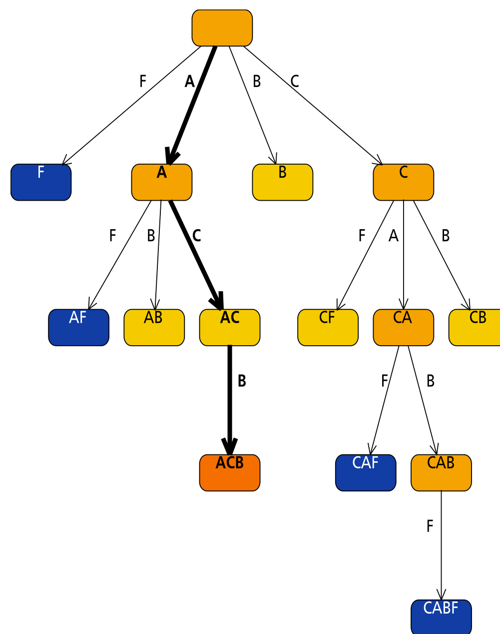


Figure 24: Visualization of the MCTS expansion phase

3.6.3 Simulation

In the *simulation phase*, the third step of the MCTS algorithm, one key idea of the MCTS is applied. As we assumed that we cannot exactly calculate the utility value of a game state with a utility function, the algorithm simulates the actions of the players, like the UCB approach in section 3.5.1, until a terminal game state is reached. The method used to simulate a game to a terminal game state is called *default policy*. As not only a random playout is possible, we will explain different default policies in section 3.10 and make some further comments. The rollout, outgoing from the newly created node ACB, is displayed in Figure 25. The outcome of the rollout in this example is $+1$ which should mean that the player running the algorithm won. After the simulation phase has reached a terminal game state, the algorithm continues with the last phase, the *backpropagation*.

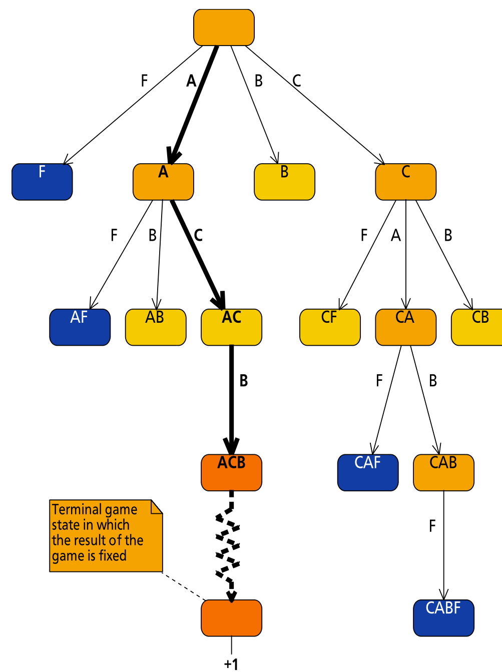


Figure 25: Visualization of the MCTS simulation phase

3.6.4 Backpropagation

The *backpropagation* phase is the fourth and last step of an MCTS algorithm. In this step, the outcome of the simulation is assigned to the simulated node and back-propagated trough the tree. To back-propagate the outcome, the path from the simulated node to the root node is taken backwards. On every node on this path, the outcome is incorporated into the current rating of the node. Figure 26 visualizes this behavior by continuing the example. At first, the outcome +1 is taken as utility value for the node ACB. The outcome is then back-propagated to the nodes AC, A, and the root node. Therefore, the backpropagation of the outcome updates the utility values of the nodes. This is done because the algorithm now has more information about all nodes on the taken path. In the example, a win was reached by the simulation. This means for each node on the path, that the result "win" is now more likely than before the rollout. The simulated path outgoing from the node ABC exclusively to the node with the terminal game state inclusively is discarded and the MCTS algorithm can make the next iteration, if the computational budget is not exhausted until now.

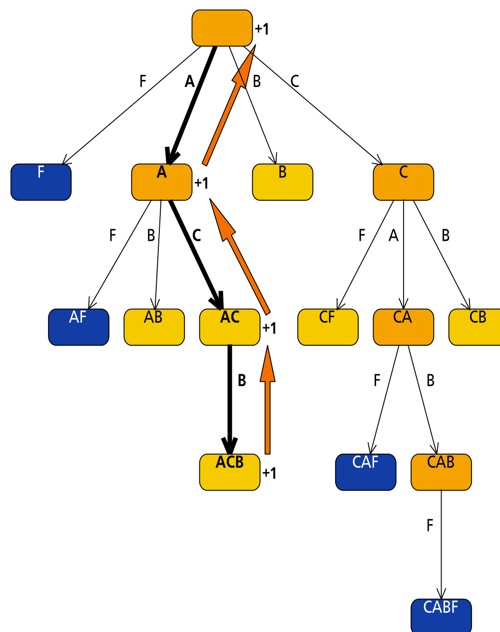


Figure 26: Visualization of the MCTS backpropagation phase

3.6.5 Upper Confidence Bound Applied to Trees

In this section, we will introduce a special kind of a Monte Carlo Tree Search. The *Upper Confidence Bound Applied to Trees (UCT)* [12], is one of the most popular approaches to guide the exploration of a tree in an MCTS algorithm. As it is used to navigate through the tree in the selection phase, it handles the exploration/exploitation-dilemma and is therefore a concrete implementation of the generic tree policy mentioned in section 3.6.3.

The UCT value is similarly defined as the UCB value in section 3.5.1. The UCT value for node j equals

$$\bar{x}_j + c * \sqrt{\frac{2 * ln(n)}{n_j}}$$

where

- $\bar{x}_j$ is the average outcome observed for node j,
- n_j is the number of times node j has been visited,
- n is the number of visits of the parent node of node j, and
- c is an additional parameter to control the exploration/exploitation tradeoff.

For $n_j = 0$ we define the UCT value as $+\infty$.

3.6.6 Asymmetric Property

In comparison to the bandit approaches in section 3.5, the tree approaches described in this section build the move tree during the runtime of the algorithm. This is a big advantage since no time consuming building of the bandits is required. Another big advantage is the asymmetric property resulting from this implied tree building. The algorithm leads the search during its runtime to promising parts of the search tree. This is done by the tree heuristic described in section 3.6.1. This tree heuristic does not only determine which moves are investigated in more detail but determines which parts of the tree are actually built. Therefore, these approaches are furthermore less storage consuming than the bandit approaches. Since the tree building is implied in the algorithm there is no need for a time consuming construction of all theoretically possible options like in the bandit approaches. However, this leads to another problem. When the tree approache does not have enough simulations available to build sufficiently big and deep trees, it has to deal with not fully expanded move trees in the game Hearthstone. As shorter moves are rather inferior to longer moves, we describe this problem in the following section in detail.

3.7 Post Monte Carlo Strategies

It is possible that actions are still available after a Monte Carlo tree search algorithm has finished its main computation loop. Especially in the case when only few simulations are available it is very likely that there are remaining actions in a ply. Shorter moves are not in general worse than longer moves but in most cases they are. If we make a short move in situations where we can make a long move too, we may be wasting actions which would improve the game state. This could mean, for example, that minions are wasting their attacking possibilities or that minions are summoned in later plies where they are not as effective anymore. But this phenomenon does not only occur in MCTS algorithms. Bandit approaches can also select moves which can be further extended by executing additional actions. Since not executing possible actions is rather bad in Hearthstone, we introduce three other possible approaches to the standard approach “do nothing” for post Monte Carlo strategies.

3.7.1 Do Nothing

The first strategy is also the simplest one. Even if there are additional actions available after the execution of the actions which were selected by the Monte Carlo approach, we just can do nothing additionally and finish the ply.

3.7.2 Additional Random Atomic Action

The second approach is an execution of random actions. Like the random action approach described in section 3.9.1, this post Monte Carlo approach executes random actions until no further actions are available. This strategy is, the second fastest of the four strategies, after the “do nothing”-strategy.

3.7.3 Additional Random Move

The third strategy plays additional random moves. This behavior is similar to the random move player in section 3.9.2. It builds all possible moves and then selects one of them randomly.

3.7.4 Additional Random Move with Maximum Length

The last post Monte Carlo strategy is similar to the previously described method. In comparison to the “random move strategy”, the “random move strategy with maximum move lengths” will preselect the longest moves generated in the move generation step and then select one of them at random. This guarantees that after this strategy was applied, no other actions can be made.

3.8 The Parameter c

As described in [*13*], the concrete value for the parameter c can have a big impact on the performance of the algorithm. Therefore, we will investigate several option for this parameter in experiments in section 5.7 for the UCB and the UCT algorithm.

3.9 Random Approaches

To be able to investigate the behavior of the Monte Carlo approaches better, we also implement two players who act randomly and will act as baseline algorithms in some experiments. The random players are no Monte-Carlo algorithms since they do not simulate games states to estimate the value of moves. Below, two different random players are described. The first one will play random actions until there are no more actions available. The second one will enumerate all available moves and select one move at random. The advantages and disadvantages of the two approaches are also discussed in detail.

3.9.1 Random Action Player

The random action player will play random atomic actions. This means that the player will enumerate all available actions for a given game state and will choose one of these actions at random. The selection of random actions is repeated until no more actions are available. This is done because using all actions and all mana available (which results in longer moves) will be in the most cases a better play than not using them.

This strategy can be executed fast since there are only few actions available at each game state. Only the combination of these actions results in a vast amount of possible moves but the random action player has not to deal with this circumstance. Therefore, no computational power is needed to generate all possible moves and the player can play his moves really fast. But this strategy also has a drawback.

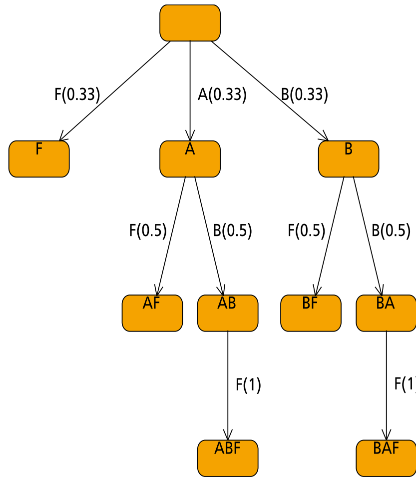


Figure 27: Move tree for a random action player

We see in Figure 27 a move tree for a random action player. In the given game state, there are three actions available at the root node: action A, B, and F, where F means that the current ply is finished and no other actions can be executed. Additionally to the available actions, the selection probability is displayed if the random player has to select one of the actions. For example, in the root node, the probability of choosing action A is 0.333 because there are three actions available. The algorithm will now choose an action at each node to get to a following game state and continue this until no actions are available. For example, the random action player could choose action A in the root node leading to game state A. There he could choose action F and get to game state AF. In this game state, there are no more actions available and the ply of the random action player would be finished.

We can now calculate the probability for each move. The probabilities and lengths for all complete moves, which always end by executing the finish ply move F, are listed in Table 4.

Move	Probability	Move length
F	0.333	1
AF	0.166	2
BF	0.166	2
ABF	0.166	3
BAF	0.166	3

Table 4: Available moves, selection probabilities and move lengths for the move tree in Figure 27

As we see, the probabilities for longer moves are smaller than for shorter moves, which lead the random action player to playing shorter moves. This effect would be even bigger for bigger trees with more actions available. For the given example, this results in an average move length of 2 for the random action selection strategy of this player.

3.9.2 Random Move Player

The second random player does not select atomic actions at random, like the player described above, but rather complete moves at random. This means that the random move player has to generate all possible moves in the beginning of a ply. This could, depending on the available actions, be a vast amount as described in section 3.3.

Furthermore, generating all possible moves is not a trivial task in the game Hearthstone. To generate the moves, the algorithm has to generate all actions at each game state. Then, it has to apply all the actions to the game state to get to the following game states in which it can then generate all available actions for the new game states. Since the application of actions to game states in Hearthstone is a computational challenging task, this behavior consumes a not to neglecting amount of time. This is a big disadvantage of the random move player in comparison to the random action player. The amount of available moves can be even so big, that the generation of more moves must be abort due to storage limitations. In this case, the method will not be able to choose from all available moves, but only from a subset. The impact of this disadvantage can be decrease by pruning duplicate moves.

However, there is also a big advantage of this method. We described above, that the random action player prefers short moves in average. As this approach chooses uniformly distributed out of all available moves, shorter moves are not preferred in comparison to longer moves. In the example in Figure 27 and Table 4, each move would be selected with a probability of 0.2. This would result in an average move length of 2.2. The difference in this small example is relatively small, but in bigger, more realistic move trees the difference will be significant. Additionally to the uniformly distributed selection, there will be relatively more long moves in comparison to shorter moves when more actions are available. For example, if we have 10 actions available, which are freely combinable, we will have only one move with length 1 (the finish ply action), 30,240 moves with length 5, and 3,628,800 moves with length 10.

Another drawback of this method is that after selecting a random move, there can still be available actions to execute. For example, the algorithm could choose at random a shorter move which leads to a game state in which further actions are available. To handle this situation, the algorithm could start again at the new game state and could choose another random move. As there are fewer actions available as in the original game state, this second iteration would consume less time.

Another possibility would be to use the random action player to utilize the remaining actions. As the random move player should have selected a relatively long move, the property of the random action player to prefer shorter moves would not be resulting in short moves.

A third approach for the problem would be not to draw randomly out of all generated moves. Instead of this standard behavior, the algorithm could throw away all short moves and select a random move only out of longer moves. In doing so, the move lengths would also be maximized and the probability that unused actions are still available after the execution of the randomly drawn move would be minimized.

We refer to the experiments in section 5.4, which will investigate the statements made in this section further.

3.10 Simulation of Games

One important part of each Monte Carlo algorithm is to simulate/roll out games from a specific game state until a terminal game state is reached. By doing so, an estimate for the utility value of the game state should be generated because it is not possible to directly calculate the value with a utility function. The default strategy for simulating games in both, the bandit approaches and the tree approach, is to let both players randomly select moves until a terminal game state is reached. This strategy has some advantages. Firstly, the strategy is easy to implement since the generation of possible moves should be an easy task. Secondly, the method should be one of the fastest ways to get to a terminal game state since there is no computational power consumed to evaluate and compare different moves. The performance is quite important because the Monte Carlo methods use most of their time to simulate games and the number of games, which can be simulated within the time limit

given by the game rules, is a major factor influencing the performance of the method. Generally said, the more games can be simulated, the better the algorithm will perform. Therefore, it is important that one simulation takes as little time as possible, which is fulfilled by the random simulation. Furthermore, random simulations do not need domain knowledge. Hence, random simulations can be executed without adaptions for a wide variety of problems and not just for specific games. The last mentioned advantage is that random simulations will explore different areas of the search space. Therefore, the resulting average values are composed of very different paths in the game tree. The problem with random simulation is that different moves are included in the estimate with the same weight, which will not be a realistic assumption for rational players. The random simulation will make very good moves as commonly as very bad moves. But a rational player will, even if he is not an expert of the game, not have this property. For example, if it is possible to checkmating an opponent in chess, a rational player will be doing this to win the game immediately. Furthermore, most of the time it is easy to discover that a player has the ability to checkmate his opponent. However, a random player will not see this chance and will instead make a random move. As the branching factor in chess is big, the chance of hitting the one rational move by randomly playing a move is low. The playing skill of the opponent is therefore strongly underestimated which will lead to irrational and therefore wrong estimates of a game state. The same underestimation holds for the player itself as he simulates his play with a random play, too. One possibility to address this problem is the usage of simple pre-defined heuristics [*14*] to at least minimally guide the players to not make the worst moves. The usage of domain knowledge can help to bias the moves during the simulation and make the simulation much more realistic. Such a rule-based policy should be fast enough to not slow down the simulation much in comparison to a random simulation. As the rollouts get more realistic, the average outcome of the simulations should estimate the true utility value of a game state more realistically. This is an important point when thinking about what we really want to have as a result of the whole algorithm. Simulating an infinite amount of games with randomly playing players will lead to a strategy that is best against a randomly playing player and not, as desired, against a rational player. To converge against a realistic outcome it is not sufficient to just simulate a lot of games but rather to simulate a lot of realistic games. So, using realistic players during the simulation would not only need fewer simulations to converge but also converge against a more realistic limit. Therefore, the assumption, that we can estimate the true value of an action using random simulations, is wrong in general. We can just hope that random simulations are not too far away from the true utility value.

To simulate Hearthstone games, we will use random playouts as well as a simple heuristic version based on expert knowledge and compare both of them. As the thesis focusses on a comparison of flat and structured approaches, we will not only analyze if a method can gain playing skill by using a heuristic in the simulations but also analyze which of the both approaches benefits more by doing so.

Another possible improvement for the simulation would be using an opponent model to better simulate the opponent's moves. If we, for example in a game of Tic-Tac-Toe, detect that a player does not play the not exploitable Nash equilibrium strategy by playing each of the three possible moves uniformly distributed, the simulation of the games can be adapted to this. For example, if the player plays more often the move "Stone", we can use this information in the simulations and not simulate a random player who plays each move uniformly distributed, but bias his moves accordingly.

As we have not implemented an opponent modelling for Hearthstone, we will not use this possible improvement to simulate the games. But for Hearthstone being a game with hidden information, not only the player's strategy can be estimated but also his hidden information. To simulate the opponent's moves in a game with hidden information, it is at least necessary to give a guess about the hidden information. In Hearthstone we have to make a guess about the hidden cards of the opponent, considering they are essential for his next possible moves. As we do not have implemented a sophisticated opponent modelling, we will assign for each simulation new cards to the opponent's hand and deck.

The random effects during the simulation are handled similar to the assignment of cards. As a player simulating the game cannot know how the moves by nature will look like, we assign a random seed to the game state that should be simulated. The player can then simulate the moves by nature like drawing cards by executing effects with randomness. The random seed is, like the assignment of the opponent's cards, newly initialized for each simulation to guarantee that the simulation will discover more of the underlying state space.

4 Implementation Details

As mentioned in section 2.1, the game Hearthstone was developed and implemented by Blizzard Entertainment in 2013/2014. Since the company did not release the source code of the game and did not publish an API to play the game with non-human players (this is actually prohibited by the end user license agreement) it was necessary to re-implement the game for experimentation purposes. Without doing this, it would not be possible to test the playing strength of different approaches and investigate the impact of various extensions and parametrizations of the used algorithms. Furthermore, it would not be possible to play thousands of games in the online version of the game.

4.1 General Information

The code implementing the game was written in Java 1.8 using the Eclipse IDE. A total number of 24,000 lines of code were written. The biggest parts of the code are used to implement the model including the game state, the game rules and the cards. Other big parts of the source code were written for the players and the experimentation environment, and for JUnit testing, which has saved a lot of implementation time by finding bugs fast and early. Apache Subversion was used as software versioning and revision control system. For ease of use, TortoiseSVN was used as a client for the SVN server. The experiments were distributed on different computing systems to be able to run the necessary amount of simulations in an appropriate time. The results of the different experiments were collected on a centralized MySQL server. phpMyAdmin and HeidiSQL were used as clients for the MySQL server. To find possibilities to improve the performance of the system we used VisualVM.

In the following sections, some key features of the implementation are described. Given that the implementation of the game is not the main target of this thesis, the descriptions are not comprehensive but rather focus on some crucial implementation aspects. Furthermore, we will not go into code details but only describe the applied functional principles.

4.2 Game State

The game state is used to model a specific state in which the game can be. It contains all relevant information about the game. In chess, this would be the position of all pieces and which player has to act. Furthermore, one could store timing information if time limits should be applied for the players. In Hearthstone, a game state must hold much more information than in Chess. For example, we must store which minions are on the board, whether they have any enchantment and can attack in the current round or not, the hand and deck cards of the players, equipped weapons, and much more. To give an impression of the complexity of Hearthstone, we show an entity-relationship model of the game state in Figure 28.

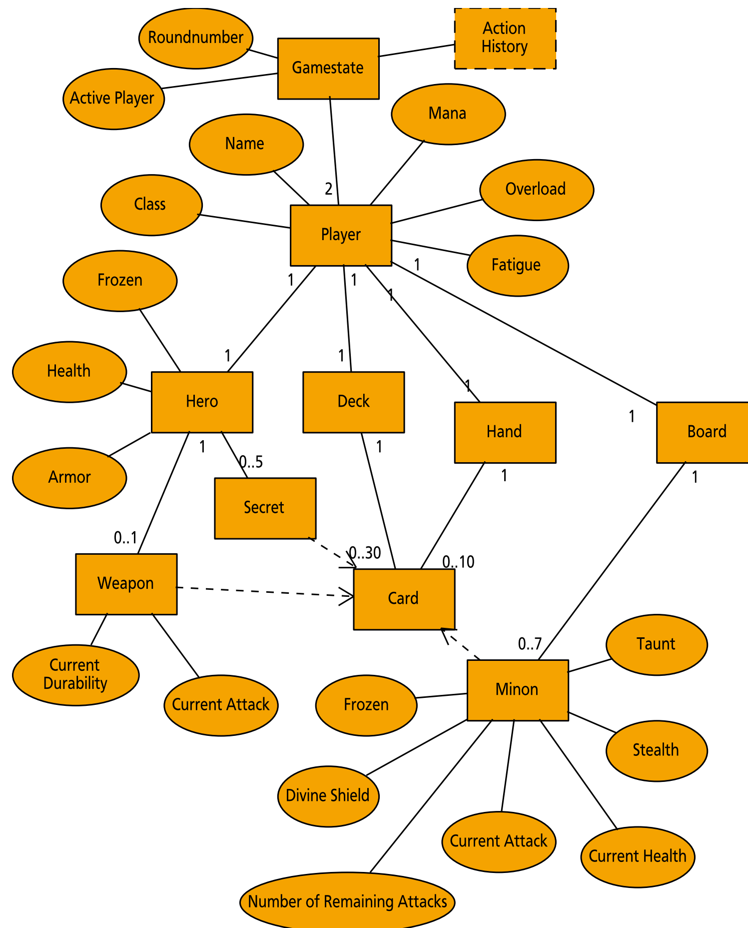


Figure 28: Entity-relationship model of the game state

The game state consists of exactly two players and, in addition, saves the current round number and an action history. The action history is optional and not necessary to save the current game state but can provide useful information. For example, the consideration of already played cards can give an advantage on further decisions. A player can participate in an infinite amount of games, therefore he can be part of an infinite amount of game states. A player has a unique name for identification purposes. He also has an amount of available mana when he is the currently active player and an amount of overload for the next round. The round number of the first fatigue is also stored. Furthermore, a player can have up to three secrets. A player has exactly one deck, one hand, one board and one hero. The Deck consists of cards without other attributes. A hand consists of modified cards, as they can have another mana cost value as the origin related card. A hero has an amount of health, armor and can be frozen. He can also have a weapon equipped which has a durability and an attack. The weapon can have some special abilities like healing the hero every time when the weapon is used, so a reference to the related card is necessary to model a proper game state. The board consists of minions which have a lot of additional attributes. At first, it is necessary to store the related card for the abilities and special game situations like retrieving minions from the board back to the hand. Like the hero, a minion can be frozen and has an attack and a health value. A minion can have in addition the abilities taunt, divine shield and stealth. Furthermore, the number of remaining attacks must be stored. All lists, like the board, the secrets, and the deck are ordered lists.

4.2.1 Example

To show an example of a game state, the situation in Figure 29 is modeled in Figure 30. For simplicity, the action history and the decks of the players are not illustrated. But they are important in a full description of the game state.

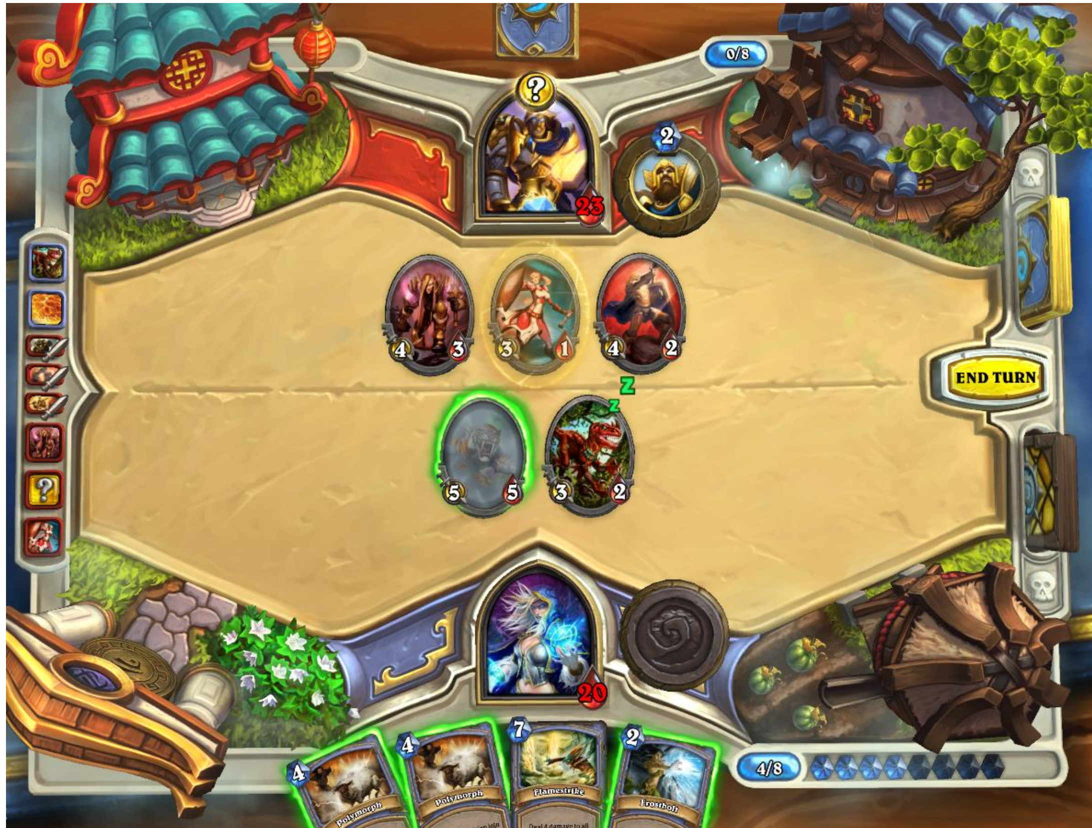


Figure 29: A typical game situation in Hearthstone

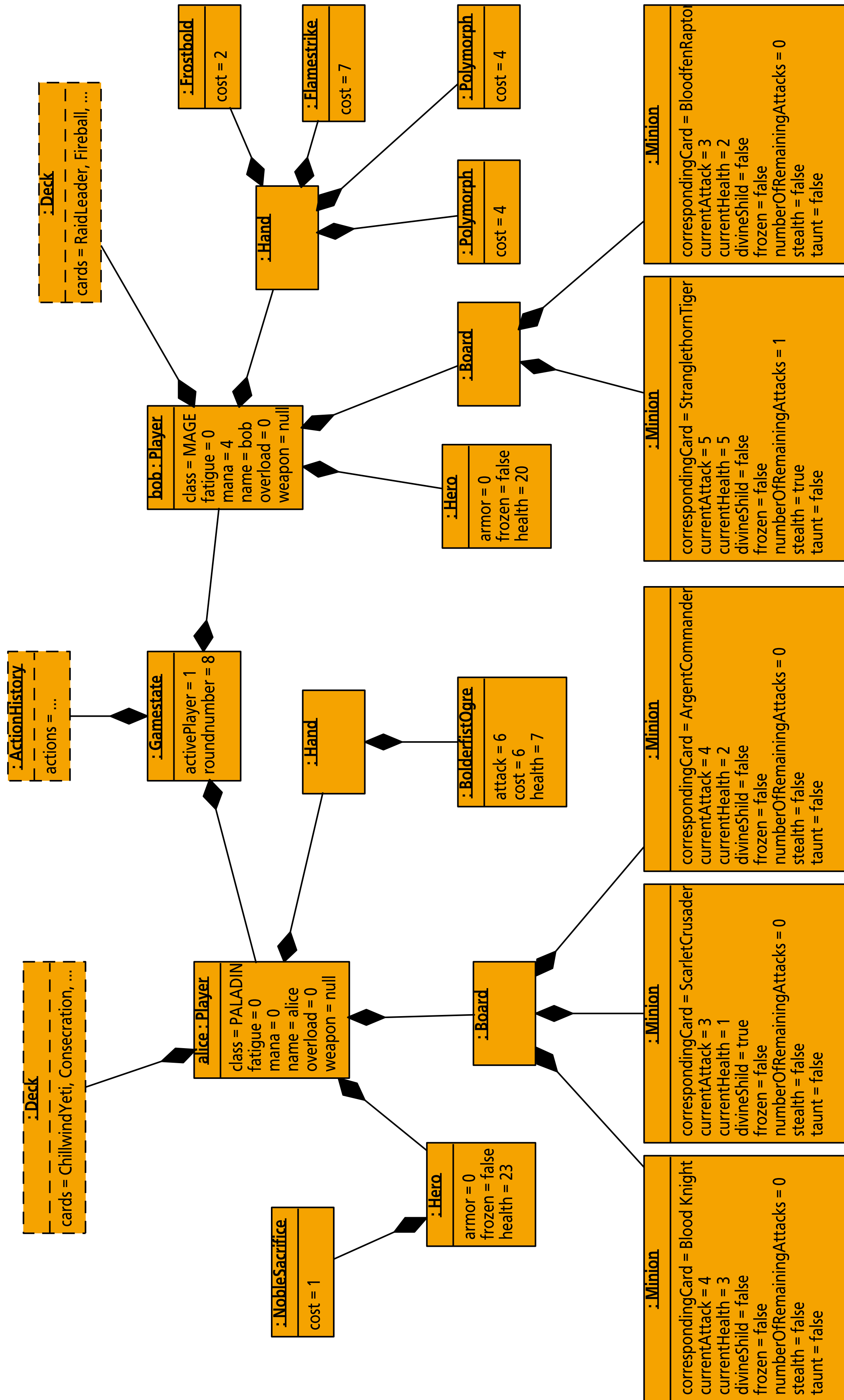


Figure 30: A sample game state in Hearthstone

The root of the game state stores the round number 8 and holds the two players, Alice and Bob. Furthermore, it stores player 1 (Bob) as the active player. The variable would be zero if Alice was the active player.

Alice has chosen Paladin as her class. She has no weapon equipped, no overload and has no fatigue. Zero mana points indicate that she used all of her eight available mana points in her previous turn. Her hero has zero armor, is not frozen and has 23 health points. The hero has a secret called Noble Sacrifice which cannot be seen in Figure 29. Alice has three minions on the board: A Blood Knight, a Scarlet Crusader and an Argent Commander. The Blood Knight, for example, has a current attack value of four, a current health of three and is not frozen, not in stealth, not in taunt and has no divine shield equipped. The number of remaining attacks of zero indicates that this minion has used all of its attacks in the last round. The Scarlet Crusader and the Argent Commander are modelled analogously. In her hand, Alice furthermore holds a Bolderfist Ogre card which is also not visible in Figure 29. The current attack value of this card is six, its health value is seven and the cost value is six. As mentioned above, the deck is not modelled in detail. In Figure 30, only the next two cards in the deck are hinted: Alice will draw Chillwind Yeti and Consecration as next cards. This information is not visible to the player but storing the ordering of the deck is important as two different continuations of the game should produce the same card draws.

Bob's class is Mage. He has like Alice no weapon, no fatigue, and no overload. Four mana points denote that he has four mana points left in this turn. His hero has zero armor, is not frozen and has 20 health points remaining. On his hand are four cards: two Polymorphs, a Flamestrike and a Frostbolt with costs of four, seven and two respectively. On the board, Bob has two minions: a Stranglethorne Tiger and a Bloodfen Raptor. The Stranglethorne Tiger has five attack and health points, is not frozen, not in taunt and has no divine shield. But he is in stealth and has one remaining attack in this turn. The Bloodfen Raptor in contrast has no remaining attacks in this turn. Like with Alice, the deck is only hinted and shows that a Raid Leader and a Fireball are the next cards to be drawn.

4.2.2 Actions & Events

The game state of Hearthstone is constructed as an event-driven system. The players, the minions on the board, the weapons, and the game state itself are entities in this system. All these entities can create, modify and react on events in the system. This complex system is necessary to make it possible to compute the impact of player actions correctly. The events are divided in pre- and post-impact events. Pre-impact events are triggered first and then the effects of the event are applied. Finally, the corresponding post-impact effects are applied. For example, there is a pre- and a post-impact event for summoning a minion. A secret must be able to abort the summoning process and therefore can react on the pre-impact event. Some minions will react on the corresponding post-impact event. "After you summon a minion, deal 1 damage to a random enemy" is an example card text for this.

In the following three tables, we give an overview of all events available. Table 5 lists all events correlating directly to an action playable by a player. These events represent all available actions the players can make during a game. Table 6 lists the remaining pre-impact events. For all the events in Table 6 a corresponding post-impact event exists. These are not listed. In Table 7 two events with no corresponding pre-impact event are listed.

Event	Description
AttackWithHeroEvent	Triggered when a player attacks with his hero.
AttackWithMinionEvent	Triggered when a player attacks an enemy with a minion on the board.
EndPlyEvent	Triggered when a player finishes his turn.
PlayCardEvent	Triggered when a player plays a card from his hand cards. Examples are summoning a minion, casting a spell, or applying a secret.
UseHeroPowerEvent	Triggered when a player uses the special ability of his hero. This can be with or without a target for example in the cases of the hero powers “Reinforce” and “Pyroblast”

Table 5: Listing of all player pre-impact events

Event	Description
CastSpellEvent	Triggered by a PlayCardEvent if the corresponding card is a spell card.
DamageCharacterEvent	Triggered when a character should be damaged. This can occur after an AttackWithMinionEvent or a CastSpellEvent. The game state will directly apply the effects of this event to the character which should be damaged. This will be in most cases a reduction of the health points of the character.
DamageCharactersEvent	Analogously to the DamageCharacterEvent but for more than one character at a time.
DestroyMinionEvent	Triggered when a minion should be destroyed. This can occur when the health points of a minion are reduced below zero or when a spell destroys a minion directly.
DestroyWeaponEvent	Triggered when a weapon should be destroyed. This can happen when a weapon’s durability reaches zero or when a weapon is destroyed by another effect directly.
DrawCardEvent	Triggered when a player should draw a card, for example, at the beginning of a ply.
EquipWeaponEvent	Triggered when a player equips a weapon.
HealCharacterEvent	Triggered when a character should be healed. Analogously to the DamageCharacterEvent, the game state will increase the health points of a character if a HealCharacterEvent occurs.
HealCharactersEvent	Analogously to the HealCharacterEvent but for more than one character at a time.
RemoveCardEvent	Triggered when a card should be removed and can be triggered after a PlayCardEvent. The game state will remove the appropriate card if a RemoveCardEvent is triggered.
RemoveMinionEvent	Triggered when a minion should be removed from the board. This occurs, for example, when a minion was destroyed.
RemoveSecretEvent	Triggered when a secret should be removed.
SilenceMinionEvent	Triggered when a minion should be silenced. The game state will remove all enchantments of the corresponding minion and set it to “silenced” if this event is triggered.
SummonMinionEvent	Triggered when a minion should be summoned. When such an event occurs, the game state will create the minion and put it on the board.
TransformMinionEvent	Triggered when a minion is transformed. The game state will remove the target minion and replace it with the new minion.

Table 6: Listing of all non-player pre-impact events

Event	Description
PlyEndedEvent	Triggered when the current ply was finished.
PlyStartedEvent	Triggered when a new ply was started.

Table 7: Listing of two post-impact events

4.2.3 Altering the Game State

As described in section 4.2, the modelling of a Hearthstone game state is not as simple as in board games like Chess or Go. But the modelling of a game state is not the only difficult task if we want to implement the game. The moves made by the players are causing changes in the game state. In Chess a move is only the position change of a piece. If one player wants to move his king from field e1 to e2 the only change in the game state is removing the king in field e1 and adding the king to field e2. However, in Hearthstone, it is a challenging task to calculate what the game state will look like after a move was executed. There might be various entities like minions, secrets, and cards which will be affected by a move or affecting the move itself in different ways. For example, playing a card does not always have the same result. In one game state, a secret might be active which cancels the playing of the card and the card will have no effect at all except of expiring of the secret. In another game state, a minion might be gaining attack value by attacking a random other minion when a card is played. There are also situations where it is important whether the effect triggered by the card playing event is triggered before or after the effect of the card is applied. If the played card summons a minion, it is important if a minion triggers its attacking effect before or after the minion was summoned. The recently summoned minion can only be a target for the attacking minion in the latter case. Because of these calculation difficulties, it was necessary to implement a sophisticated game state change management to calculate the impact of moves.

The events triggered by players and objects in the game state, like minions and cards, are at first added to a queue storing all events that are not applied to the game state until now. This queue is processed by removing the first element of the list and applying the effects of the removed event. Applying the effects can mean that health points of a minion are removed if a new minion is summoned. But an effect can also emit new events. For example, if an effect decreases the health points of a minion below zero, it will emit a new event to destroy that minion. Such new events are added at the end of the queue. We give an example of a queue in Figure 31.

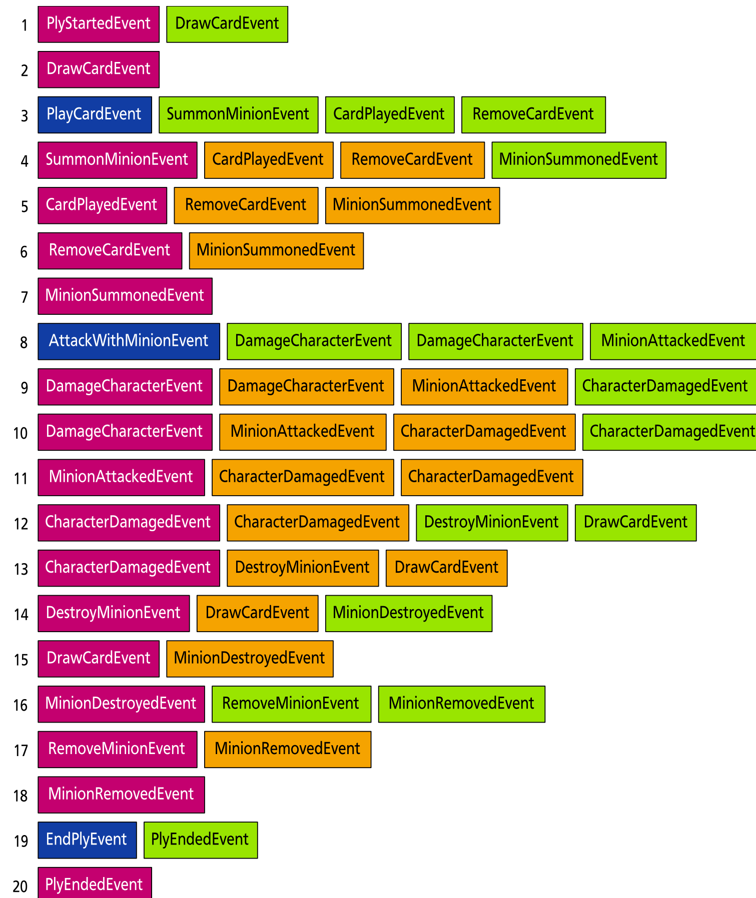


Figure 31: Visualization of the event queue

In purple we see the events currently applied. Events triggered by a player are blue. New events, triggered by the currently applied event are green. In line eight, for example, an AttackWithMinionEvent is applied. This event triggers three new events: two DamageCharacterEvents and one MinionAttackedEvent. The game state will react on the DamageCharacterEvents and will decrease the health points of the damaged minions accordingly in lines nine and ten. Furthermore, we see in these lines that the game state emits two new CharacterDamagedEvents. In line 12, the first CharacterDamagedEvent is applied to the entities in the game state. A minion is being destroyed and a DestroyMinionEvent is generated by the game state. Additionally, a DrawCardEvent is created by a minion because the text of the corresponding card says “Whenever this minion takes damage, draw a card”.

4.3 Server and Player Interaction

In this section, we will give a short introduction on how experiments are executed and how the server and player interaction during games works. Figure 32 visualizes the process of running an experiment.

Figure 32: Sequence diagram of an experiment with the server/player interaction

In the top left corner we see instance of the experiment class. This object is responsible to run an experiment. An experiment consists of an arbitrary number of single games. To run a single game, the experiment first creates the two players who should play in the game. In this case we see the instances Alice and Bob which are created in 1 and 2. Alice and Bob can be instances of a flat approach, a tree approach, or a random player. To run a single game, we use an instances of the SingleGameRunner class. In this example we see only one instance of the SingleGameRunner class which is created in 3. This means, in this experiment, we will only run one game. If we want to run more games in one experiment we would create more instances of the SingleGameRunner class. Every player and every instance of SingleGameRunner runs in an own thread. Therefore, the games in an experiment can be run in parallel which makes it possible to run multiple games on one computer at the same time. The singleGameRunner in the example creates the game server which then creates the game state. After the game initialization phase we see the main loop of the game in purple. This part is executed repeatedly until the game is finished. In the example, Alice has to act first. We see the move loop for

Alice in the upper green box. In the blue box, she computes which action she should make and then sends the corresponding event to the game server. The game server will update the game state by applying the event of Alice. Then, Alice can again compute the next action of finishing the ply and let Bob compute his move. After the game has finished, both players are notified about this and the results are returned to the experiment. The experiment collects the results of all games. After all games are terminated (either by finishing the game regularly or exceeding a time limit) the collected results are summarized and written to a database.

4.4 Graphical User Interface

To visualize the course of a game, a graphical user interface (GUI) was implemented. Figure 33 shows a screenshot of the GUI during a game.

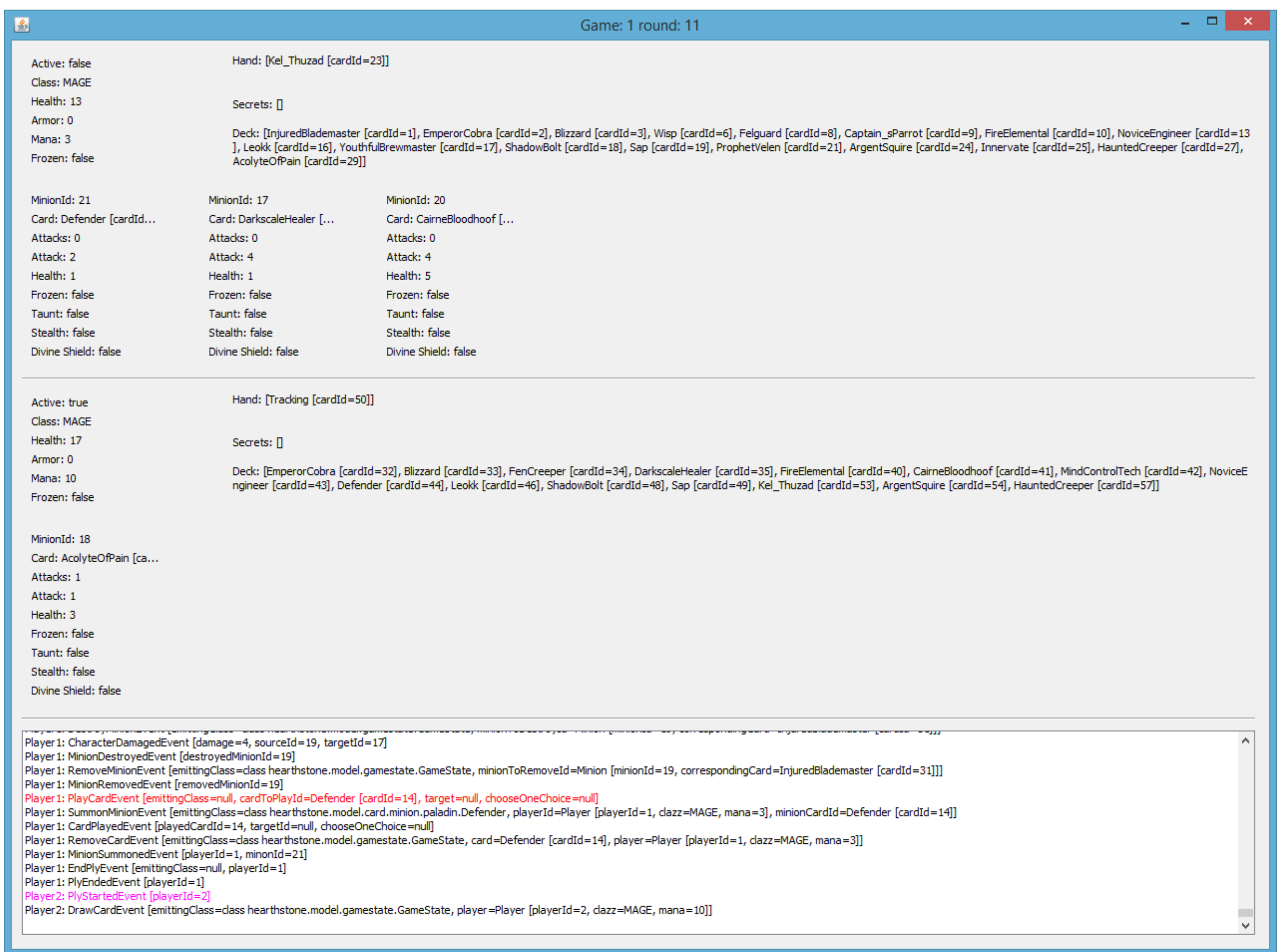


Figure 33: Screenshot of the graphical user interface

The GUI is separated into three parts: the upper part shows information about the first player, the part in the center visualizes the second player and in the lower part historical information are displayed.

For each player we see if the player is active. The active player has to act. Furthermore, the class of each player, the hero's health and armor, the available amount of mana, and if the hero is frozen is displayed. On the right of the basic information we see the players hand, his applied secrets and his deck. In the lower part of the two player boxes the minions on the board are displayed. In this game,

the first player has three minions and the second player only has one minion on the board. For each minion an identification number is displayed. Moreover, we show the corresponding card, the number of remaining attacks in the current ply, the attack and health value of the minion, and the Boolean values for the attributes frozen, taunt, stealth, and divine shield.

In the lower part of the GUI, we show detailed information about the events which were applied to the game state. In red we see an event caused by player 1. This event was caused because player 1 played a card. The black events are internal events which are in the most cases subsequent events of player events.

The view on both players in this GUI is some kind of omniscient view. During the game, the players cannot look into the decks of their opponent's hand. Therefore, the GUI does not serve as playing interface but rather for spectating purposes.

5 Experiments

5.1 Configuration Overview

In this section, we want to give an overview over the implemented Hearthstone players which correspond to the previously described algorithms. Furthermore, the configuration possibilities for each algorithm are described. The configurations are not just a technical detail but can rather heavily influence the performance of the algorithm. Additionally to the parameter name, we give a short description of the parameter, a range in which the values of the parameter will usually be or example values for the parameter, and a default value for the parameter.

Overall, we have five different player types ranging from the most simple random action player to the two more storage and time consuming random move player to the two most sophisticated Monte Carlo based players. The more complex the players are the more configuration parameters they have. During the experiments, we investigate the impact of these parameters. In the end, we will use the learned knowledge to configure the best possible configuration for each player to play in a final tournament.

5.1.1 Random Action Player

The “random action player” (RAP) corresponds to the introduced algorithm in section 3.9.1. The algorithm will always play a random atomic action until no more actions are possible. Table 8 lists the available parameters for the random action player.

Parameter	Value range	Description
Generate EndPlyEvent?	true, false; default: true	The parameter influences whether the algorithm should consider the EndPlyEvent at each decision or not. If the EndPlyEvent is not generated, the algorithm should make longer moves since the effect for short moves described in 3.9.1 is moderated.

Table 8: Parameter description for the random action player

5.1.2 Random Move Player

The “random move player” (RMP) corresponds to the introduced algorithm in section 3.9.2. The algorithm will always play a complete random move. Table 9 lists the available parameters for the random move player.

Parameter	Value range	Description
Maximum number of moves	default: 10,000	Sets the maximum number of moves to be generated. Since there can be a vast amount of moves available in a game state (see 3.3 for details) there must be an upper bound to limit the number of generated moves due to storage space limitations. We investigate this parameter in section 5.5.
Use pruning?	true, false; default: false	This parameter specifies if pruning for moves which are leading to already seen game states should be used. The parameter influences the runtime of the algorithm since the comparison of game states consumes additional time. Furthermore, using pruning can influence the move length and therefore the strength of the algorithm.

Table 9: Parameter description for the random move player

5.1.3 Random Move Player using Maximum Length Moves

The "random move player using maximum length moves" (RMPmax) works similarly to the algorithm above. But in this version, not all moves are considered to be drawn. Only one move is randomly selected out of the longest moves. The parameters equal the parameters of the random move player.

5.1.4 Upper Confidence Bound Player

The "upper confidence bound player" (UCB) corresponds to the flat algorithm described in section 3.5.1. The player will create a set of bandits corresponding to the available moves in a game state and will simulate the bandits according to their UCB value. Table 10 lists the available parameters for the upper confidence bound player.

Parameter	Value range	Description
Maximum number of bandits		(see "Maximum number of moves" in Table 9)
Number of evaluations	0 – 64,000	Sets the number of evaluations to be executed. One evaluation means that the algorithm executes the UCB values of all nodes, retrieves the node with the best UCB value and simulates the corresponding game state further until a terminal game state is reached. This is the most important parameter of the UCB algorithm.
c	0.01, 0.1, 1, 1.414, 2, 10; default: 1.414	With the parameter c, the tradeoff between exploration and exploitation can be regulated. The impact of the parameter setting is investigated in section 5.7.1.
Post Monte Carlo strategy		Specifies which post-Monte Carlo strategy the algorithm should use as described in 3.7.

Table 10: Parameter description for the upper confidence bound player

5.1.5 Upper Confidence Bound Applied to Trees Player

The "Upper Confidence Bound Applied to Trees player" (UCT) corresponds to the structured algorithm described in section 3.6.5. The player will create a move tree corresponding to the available actions in a game state and will expand the tree according to the UCT algorithm. Table 11 lists the available parameters for the UCT player.

Parameter	Value range	Description
Number of evaluations		(similar to "Number of evaluations" in Table 10)
c		(see "c" in Table 10). The impact of the parameter setting for UCT is investigated in section 5.7.2.
Post-Monte Carlo strategy		(like "post-Monte Carlo strategy" in Table 10)

Table 11: Parameter description for the Upper Confidence Bound Applied to Trees player

5.2 Confidence Interval

In this section, we will document the executed tests, their results and the consequential conclusions.

As we often compare different agents, one result of the experiments will often be the average winning rate of an agent. As this estimates the true winning rate, which cannot be calculated, the estimation of the winning rate is a point estimation. To give an impression of how precise the results are, we will also give an interval estimation for the confidence intervals. We can then say, for an experiment and a given percentage, how probable it is that the result of a repetition of the experiment will fall in the stated interval. For this, we will use the Clopper-Pearson interval [*15*] to calculate the confidence intervals since the outcome of the experiments will be binomially distributed. Because the outcome of an experiment can only be *win* or *lose,* the experiments can be modelled as Bernoulli processes and are, hence, binomially distributed. To achieve this, *ties* will be ignored.

To give an impression, why confidence intervals are important to judge experiment results, Table 12 gives results for experiments, where two random players play against each other. For this experiment, the true outcome is known. As both players have the same strategy and the same starting conditions, the result should be a winning chance of 0.5 for each player. In the table we state the number of games executed, the number of wins and losses, the resulting win percentage, and the length of the confidence intervals for 0.9, 0.95, and 0.99 ($\alpha = 0.1$, $\alpha = 0.05$, $\alpha = 0.01$).

Number of games	Wins	Losses	Winning probability	Standard deviation	Interval length for $\alpha = 0.1$	Interval length for $\alpha = 0.05$	Interval length for $\alpha = 0.01$
10	7	3	0.7000	0.4830	0.5194	0.5857	0.6981
20	12	8	0.6000	0.5026	0.3893	0.4483	0.5549
30	14	16	0.4667	0.5074	0.3216	0.3733	0.4694
40	20	20	0.5000	0.5064	0.2778	0.3240	0.4108
50	26	24	0.5200	0.5047	0.2473	0.2892	0.3687
60	31	29	0.5167	0.5039	0.2251	0.2637	0.3375
70	34	36	0.4857	0.5034	0.2078	0.2439	0.3129
80	41	39	0.5125	0.5030	0.1939	0.2278	0.2929
90	44	46	0.4889	0.5027	0.1824	0.2146	0.2763
100	49	51	0.4900	0.5024	0.1727	0.2033	0.2622
199	100	99	0.5025	0.5013	0.1210	0.1430	0.1858
299	150	149	0.5017	0.5008	0.0982	0.1162	0.1513
399	201	198	0.5038	0.5006	0.0846	0.1003	0.1308
498	247	251	0.4960	0.5005	0.0756	0.0896	0.1170
597	302	295	0.5059	0.5004	0.0689	0.0817	0.1067
692	350	342	0.5058	0.5003	0.0639	0.0758	0.0991
792	394	398	0.4975	0.5003	0.0596	0.0708	0.0926
892	438	454	0.4910	0.5002	0.0561	0.0666	0.0872
991	476	515	0.4803	0.4999	0.0532	0.0631	0.0826
1985	979	1006	0.4932	0.5001	0.0374	0.0445	0.0583
2975	1473	1502	0.4951	0.5001	0.0305	0.0363	0.0475
3971	1968	2003	0.4956	0.5000	0.0263	0.0313	0.0411
4961	2456	2505	0.4951	0.5000	0.0235	0.0280	0.0368
5949	2962	2987	0.4979	0.5000	0.0215	0.0256	0.0336
6939	3474	3465	0.5006	0.5000	0.0199	0.0237	0.0311
7929	3984	3945	0.5025	0.5000	0.0186	0.0221	0.0290
8924	4468	4456	0.5007	0.5000	0.0175	0.0209	0.0274
9914	4967	4947	0.5010	0.5000	0.0166	0.0198	0.0260

Table 12: Results for simulations of two random players for different confidence intervals

Figure 34, Figure 35, and Figure 36 give an example how the confidence intervals change with continuously increasing number of games evaluated for a value of $\alpha = 0.05$ which means that 95 % of the repetitions of the experiment will fall within the interval. Figure 34 shows a complete view over 10,000 games played. We see that the confidence interval decreases quickly in the first games and slower in the later games. Therefore, we have to simulate a lot of games to get the interval smaller than for example 1 % deviation in each direction and an infinite number of games to get it arbitrarily small. As we do not want to simulate a really large number of games due to computational power limits, we will focus on a maximum of 2.5 % deviation in each direction with a confidence of 95 % ($\alpha = 0.05$) and calibrate the number of games for each experiment accordingly. This will result in a confidence interval of a length of maximal 0.05, since the results of an experiment will be within 0 and 1 ("0" for 0 % won games and "1" for 100 % won games).

As the confidence interval size decreases during the experiment, we show in Figure 35 a detailed view of the interval in which the interesting part between 0.45 and 0.55 can be investigated better. In Figure 36 we show the details of the first 2,000 games played as we are interested in a confidence interval length of 0.05. Figure 36 better shows the number of games needed for that result than the other figures. Here we can see that we have to simulate about 1,600 games to get a result that will fall within an interval of length 0.05 with a probability of 0.95. The number of 1,600 games can also be derived from Table 12.

When we use another configuration than $\alpha = 0.05$, a confidence interval length of 0.05, and 1,600 games in an experiment that compares different algorithms, this will be explicitly noted.

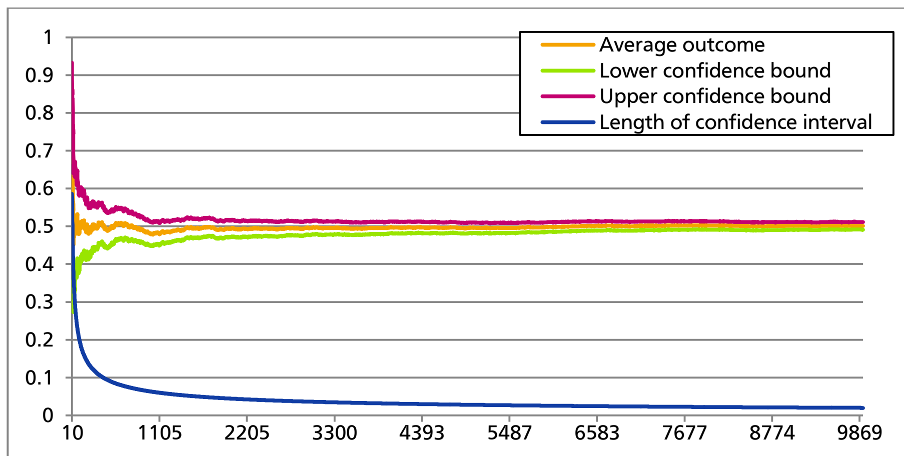


Figure 34: Illustration of the confidence interval for two random players and $\alpha = 0.05$

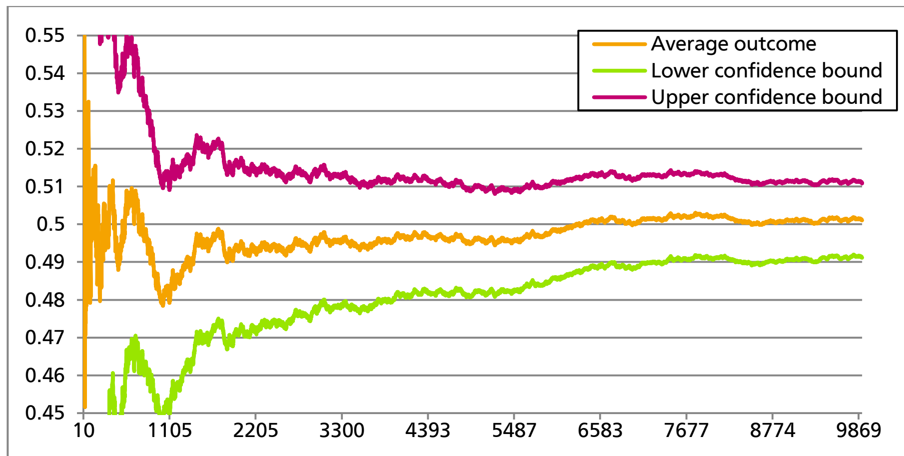


Figure 35: Detailed view of the confidence interval for two random players and $\alpha = 0.05$

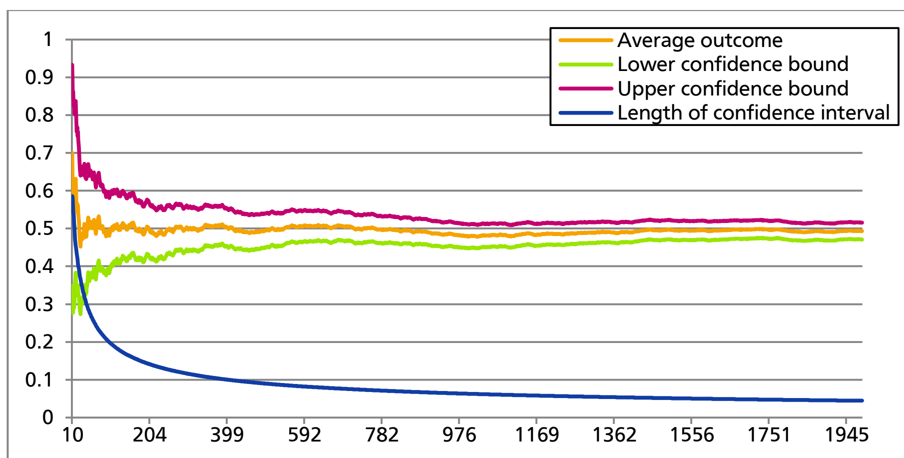


Figure 36: Detailed view of the confidence interval for two random players and $\alpha = 0.05$ for 2000 games

To validate the calculation, we repeated an experiment with 1,600 games 300 times. The result is plotted in Figure 37 with a vertical scatter for better visualization. For 300 games and a probability of 0.95 that the outcome of an experiment is in an interval of length 0.05 with a center of 0.50, we expect that 15 experimentation results do not fall within the interval. In Figure 37 we can see that six experiment results are below the lower bound and five are above the upper bound. This equals 4.33 % of the number of experiments executed which substantiate the calculation above.

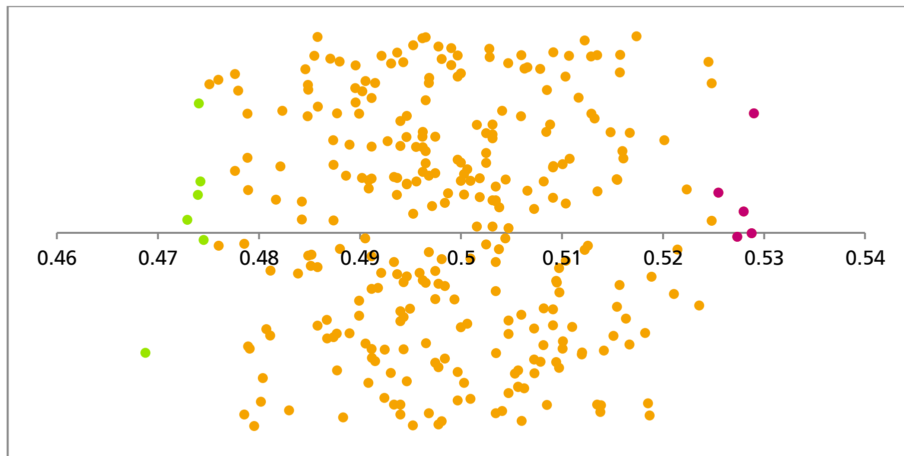


Figure 37: Scattered plot of the results of 300 games. The results are green below the lower confidence bound, purple above the upper confidence bound, and gold within the confidence interval.

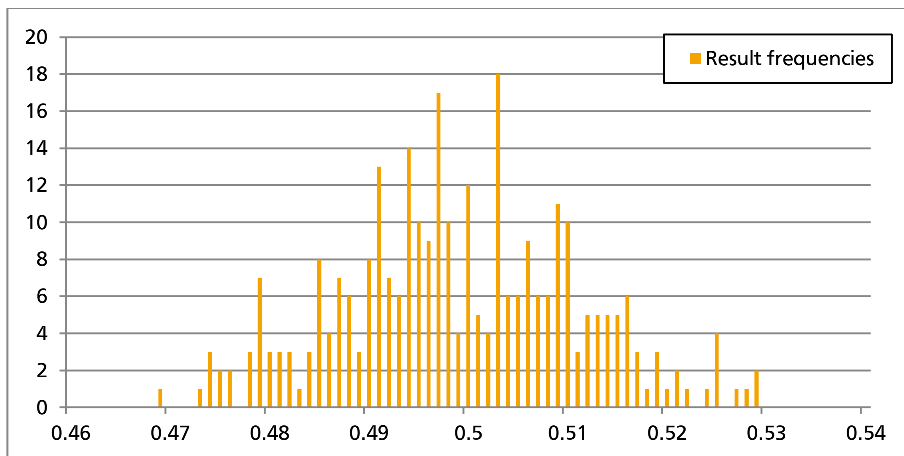


Figure 38: Frequency distribution of the results shown in Figure 37.

5.3 Evaluation Bias Caused by Evaluations in Different Game States

For experiments comparing different resulting move lengths or the remaining amount of mana, it is important that these evaluations are made independently from the playing style of the players because this would introduce a bias. Therefore it is important, not just to evaluate such target values for a player when this player has to act. Every time when such values are evaluated and collected for a player, they must be evaluated and collected for the other player, too.

For example, we will imagine two players, MAX and MIN. The players are applying different playing strategies. MAX could act randomly, whereas MIN uses a bandit approach to choose which move he wants to play. We want to know who of the two players makes longer moves since making longer moves is a sign of a stronger play. Wasting an attack possibility of a minion on the opponent hero, for example, will in most cases be a bad play. We further assume for this example that MINs bandit approach is better than MAXs random play. As MIN plays better, he will more often be in a game state in which he has more possibilities to act in comparison to MAX, since stronger play often leads to situation in which the stronger player controls the board and gains therefore the board control. If we now compare the average move length of MIN in the game states where MIN has to act, we will find out, that MIN makes longer moves than MAX. But doing such an evaluation will not allow the conclusion that MINs strategy will on average play longer moves than MAXs random play. This is not

possible because MIN can on average make longer moves than MAX due to his board control. This is a reason for its own why MIN will make longer moves. If we check how long MINs moves would be, if he acted when actually MAX has to act, the average move length will be lower than the average move length of his own moves.

Furthermore, this method would generate average move length values which depend on the opponent's algorithm. Against a weaker player, the average move length can for example be higher than against a stronger player. Even the selected deck can influence the average move length. If a player opted for a deck with many weak cards, he will make in average longer moves than a player who chose a deck containing more strong cards. If we now compare the two players, we may conclude incorrectly that the player with the many weak cards will make longer moves. But the real reason for the longer moves in this example would result from of the possibility of the player with the weak cards to play a lot of them and have many minions on the board and not on the algorithm itself.

Instead of collecting such measurement values on disjoint sets of game states (for each of the two players his own set of game states where he has to act), we have to collect the values on the same set of game states. This means that collecting the values during normal play is not useful. The better evaluation method is to generate games states during normal play and evaluate these game states for all the methods which should be tested. For example, we can play with two random players and every time a player has to act we will compute the moves which would have been selected by other strategies and calculate the lengths of these moves. Then, the random player plays his move to continue the game and the next player has to act. Now we can again calculate which move would have been selected by other strategies and play again a random move to continue the game.

In Table 13 we see the result of the evaluation of 1,600 games. As the random action player plays worse than the UCB player, he is on average in more game states in which only a few actions are possible. Therefore, he will make shorter moves. This effect is clearly visible if we investigate the average move length of the UCB player. In his own game states, he has an average of 3.89 and in the game states of his opponent he only makes moves with an average length of 3.62. If we evaluate the move length in both game sets, we get a more precise result for the true playing strength.

	Evaluation in game state set of random player	Evaluation in game state set of UCB player	Evaluation in both game state sets
Number of game states	15,110	16,576	31,686
Average actions available	9.84	13.41	11.71
Avg. random move length	2.80	2.99	2.90
Avg. UCB move length	3.62	3.89	3.76

Table 13: Comparison of average move length in different game state sets evaluated in 1,600 games

5.4 Comparison of the Random Players

In this experiment, we investigate the behavior of the different random players described in section 3.9. In the experiment, four different random players play against a UCT player. To get meaningful results for the weak random players, the UCT player only uses 40 simulations, which also results in a weak performance. The results of the experiments are listed in Table 14.

	Evaluated game states	Expansion aborting rate	Average game states created	Average actions available	Average move length	Win rate
RAP	55,222	-	3.53	14.63	3.53	0.432
RMP without pruning	46,952	0.034	3,523	13.70	4.14	0.585
RMP with pruning	50,716	0.028	2,330	15.64	3.81	0.508
RMP max length	42,433	0.026	1,566	12.76	4.38	0.688

Table 14: Evaluation results to investigate the behavior of random players against a UCT player with 40 simulations

The columns are explained in the following list:

- Evaluated game states: The number of game states, which were evaluated by the algorithm.
- Expansion aborting rate: The fraction of expansions which were aborted due to storage limitations. This column is only filled for the random move player, as the random action player does not have to abort any expansions.
- Average game states created: Average game states created for each game state where the algorithm was applied. For RAP this value equals the average move length as this algorithm does not create other games states except the one which is reached by applying RAP.
- Average actions available: The average number of actions which were available at the start of each ply.
- Average move length: The average length of the moves which the algorithm would have played in each game state.
- Win rate: The win rate of the algorithm.

The rows are explained in the following list:

- RAP: The random action player as described in 3.9.1.
- RMP without pruning: The random move player as described in 3.9.2. In this version, no pruning was used and the limit for the maximum amount of moves generated was set to 10,000.
- RMP with pruning: Same as above with pruning.
- RMP max length: Same as above, again with pruning.

We see, as expected in section 3.9.1, that the random action player plays shorter moves than the random move player. Surprisingly, RMP without pruning performs better than RMP with pruning. The reason for this result can be found in the slight difference of move length distribution. The pruning of moves leading to duplicate game states does not prune all moves lengths equally. The probability is higher for longer moves that there is another move of the same length leading to the same game state. For example, move of length 1 can never be pruned because no other move will lead to the same game state. This pruning biases the fractions of moves with the same length and then results in a significant difference of playing strength.

Table 15 compares the distribution of the move lengths for the RMP approach with and without pruning.

Move length	Average moves in RMP without pruning	Fraction in RMP without pruning	Average moves in RMP with pruning	Fraction in RMP with pruning
0	1.00	0.000	1.00	0.000
1	13.70	0.004	13.37	0.006
2	50.78	0.014	52.95	0.023
3	219.06	0.062	179.78	0.077
4	621.32	0.176	395.03	0.170
5	917.77	0.261	548.34	0.235
6	775.31	0.220	502.95	0.216
7	467.99	0.133	332.49	0.143
8	245.92	0.070	175.97	0.076
9	121.99	0.035	80.61	0.035
9+	88.65	0.025	47.30	0.020
Average move length	4.14		3.81	

Table 15: Distribution of the move lengths with and without pruning

The columns are explained in the following list:

- Move length: The length of the moves listed in the corresponding row.
- Average moves in RMP without pruning: The average amount of moves in the RMP approach without pruning.
- Fraction in RMP without pruning: The fraction of moves with the corresponding move length in the RMP approach without pruning.
- Average moves in RMP with pruning: Same as above, with pruning.
- Fraction in RMP with pruning: Same as above, with pruning.

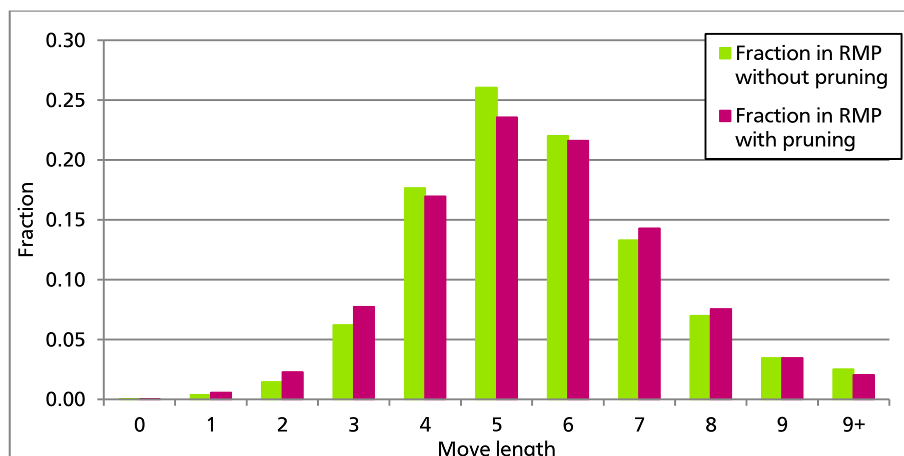


Figure 39: Visualization of the results in Table 15

In Figure 39 we see that the move lengths are similar distributed in both versions of the random move player. Nevertheless, this results in a difference of the average move length of 0.33, which leads to a significant difference in the playing strength.

5.5 Maximum Number of Bandits in UCB

Due to storage limitations it is necessary to limit the maximum number of bandits in the UCB algorithm as described in section 5.1.2. In this section we investigate how big the impact of this limitation is for different numbers of available simulations. The results of the experiments are visualized in Figure 35.

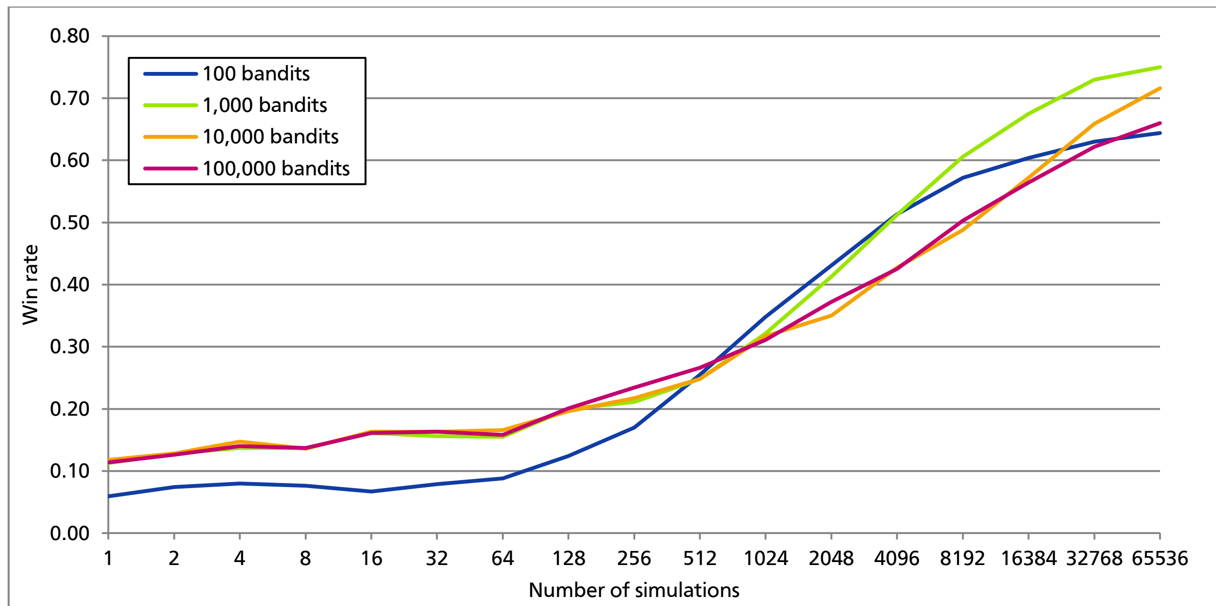


Figure 40: Visualization of the impact of different maximal amounts of bandits for UCB

In this experiment we used a UCT player with 1,000 simulations and without post-Monte Carlo play as opponent for the different UCB configurations. The performance of the UCB algorithm with a limit of 100 bandits is displayed in blue, green is the bandit with a limit of 1,000 bandits, UCB with a limit of 10,000 bandits is orange and 100,000 bandits could generate the purple UCB algorithm. We see that the last three configurations perform equally when only a few simulations are made. Only the algorithm with 100 bandits performs worse. If we increase the number of simulations all UCB players perform better. Surprisingly, the 100 bandits UCB algorithm increases its performance faster than the others. The reason for this behavior might be the fact that this algorithm investigates its fewer bandits better than the other algorithms. For example, the 100 bandits UCB can simulate its bandits with 1,024 simulations on average 10 times. The configuration with 10,000 bandits can only investigate 10 % of its bandits once and the other bandits zero times. When we further increase the number of simulations we see that performance increase of the 100 and the 1,000 bandits configuration becomes slower while the performance increase of the other algorithms becomes even faster.

It seems to be better to throw away some bandits before the UCB algorithm starts and to evaluate fewer bandits more precisely than to evaluate all possible bandits with less depth. The configurations with more bandits only perform better when we significantly increase the amount of simulations. Since the simulations in Hearthstone are rather complex and time consuming, we can only simulate the bandits up to a lower five digit number. In this region, a configuration with a maximum of 1,000 bandits seems to be the best choice.

5.6 Tree Size of UCT

We will now have a detailed look at the average tree size and other associated values for the UCT algorithm for different simulation counts. Table 16 lists the results of the experiments.

Simulations	Avg. tree size	Evaluated game states	Average actions available	Average move length	Average mana available	Average mana used	Average mana unused	Win rate
0	1	31,604	17.14	1.00	5.49	0.00	5.49	0.000
1	18	40,934	16.80	1.89	6.48	2.11	4.36	0.016
2	28	44,467	16.64	2.13	6.76	2.53	4.22	0.056
3	37	46,356	16.33	2.25	6.88	2.72	4.16	0.080
4	47	47,773	16.10	2.32	6.97	2.87	4.10	0.133
5	56	48,960	16.08	2.37	7.04	2.94	4.10	0.140
10	100	50,783	15.63	2.50	7.14	3.15	3.99	0.195
20	179	52,742	15.41	2.64	7.25	3.37	3.88	0.323
30	241	52,736	15.20	2.74	7.24	3.52	3.72	0.443
40	295	52,065	15.02	2.79	7.21	3.60	3.61	0.458
50	335	52,273	14.71	2.84	7.22	3.65	3.57	0.533
100	564	50,993	14.58	2.95	7.15	3.81	3.34	0.638
200	973	48,475	14.58	3.03	7.01	3.94	3.07	0.748
300	1,303	47,073	14.33	3.07	6.92	4.00	2.92	0.800
400	1,627	46,222	14.30	3.10	6.87	4.05	2.82	0.841
500	1,922	45,214	14.25	3.12	6.80	4.06	2.74	0.867
1,000	3,145	42,328	13.97	3.17	6.59	4.12	2.47	0.919
2,000	4,874	38,741	13.55	3.19	6.29	4.10	2.19	0.949
3,000	6,616	37,083	13.55	3.22	6.14	4.11	2.03	0.965
4,000	7,333	35,929	13.00	3.22	6.02	4.06	1.96	0.975
5,000	8,071	34,572	12.81	3.21	5.87	4.05	1.82	0.970
10,000	9,263	30,850	11.72	3.19	5.43	3.92	1.51	0.983

Table 16: Evaluation results to investigate the behavior of the tree size of UCT

The columns are explained in the following list:

- Simulations: The number of simulations used by the UCT algorithm.
- Avg. tree size: The average tree size of the tree built by the UCT algorithm.
- Evaluated game states: The total number of game states which were evaluated by the UCT algorithm.
- Average actions available: The average number of actions which were available at the start of each ply.
- Average move length: The average length of the moves which UCT would have played in each game state.
- Average mana available: The average amount of mana available at the start of each ply.
- Average mana used: The average amount of mana used by the UCT algorithm.
- Average mana unused: The average amount of mana unused by the UCT algorithm.
- Win rate: The win rate of the UCT algorithm.

In the experiment, a UCT algorithm was used, parameterized with c=1.414. Furthermore, UCT did not add additional random actions after the tree search phase. The most visited child, searched iteratively in the tree, was selected as the best child. The opponent was a random action player. 1,600 games

were played for each simulation count value to get exact results. For games with more than or equal to 2,000 games, some games did not terminate within a time limit of 30 minutes and were aborted. These games are not included in the results. Therefore, the column “Evaluated game states” of these rows is not comparable to the other rows. The other columns are not affected.

We see that with an increasing amount of simulations executed, the average tree size of UCT increases as well. The average move length increases and maybe more important, the average mana which is not used by the player decreases with an increasing simulation count. The win rate increases as well. This means that the UCT player profits from bigger trees as he can then better exploit the available amount of mana, which results in a higher win rate. Further experiments suggest that the tree size will increase further with an increasing simulation count, which may lead to an even better play. Unfortunately, it was not possible to play such big amounts of simulations due to computational limitations. Figure 41 visualizes the win rates of the experiments.

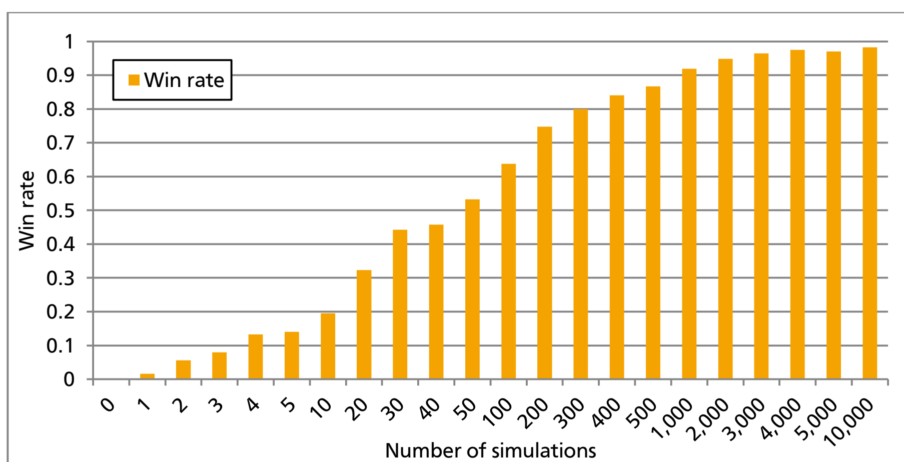


Figure 41: Visualization of the win rates in Table 16

Furthermore, we see a decreasing amount of game states evaluated during the experiments. The reason for this effect is the increasing playing strength of the UCT algorithm. Since UCT is playing better and better, the games become shorter because UCT can win the games earlier. A similar effect is observable in the column “Average mana available”. With an increasing playing strength of UCT the games become shorter and the amount of mana, which is available in average in the game states, decreases.

5.7 Adjustment of the Parameter c

As described in section 3.8, UCB and UCT can both be parametrized with a parameter c. This parameter defines a ratio between exploration and exploitation. In this section, we want to investigate how this parameter influences the performance of the algorithms. We start with an experiment with the UCB algorithm in section 5.7.1 and try different values for c in section 5.7.2 for the UCT algorithm.

We investigate in both experiments how the parameter c influences the playing strength of UCB and UCT for different simulation counts. In both experiments, we use the same opponent for all simulations counts. This opponent is a UCT player without post Monte Carlo moves and a fixed simulation count of 512 simulations for each decision. We set the parameter to four different values. The default value of 1.414 is tested along with the values 0.01, 0.1 and 10 whereas a lower value for c means that the exploration is less important. A value of 0 would mean that the algorithm should not

explore at all whereas a value of ∞ would ignore the current expected values for the possible choices (bandits in UCB and atomic actions in UCT) and therefore would not exploit at all.

5.7.1 UCB

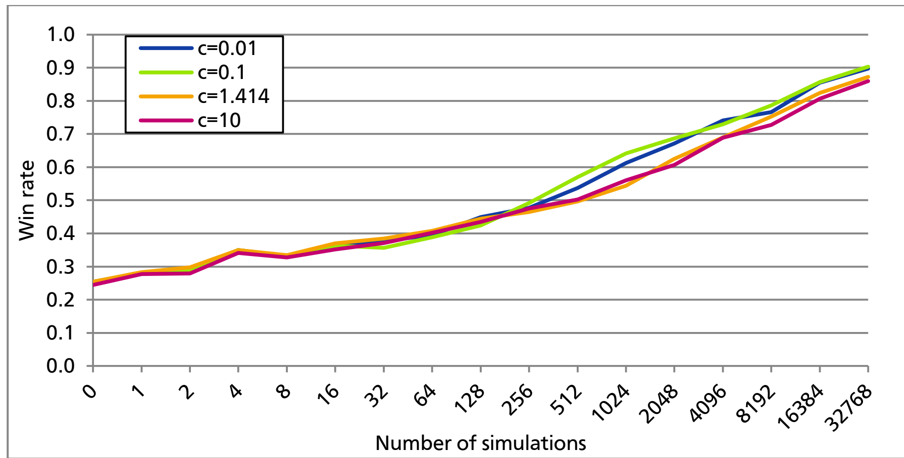


Figure 42: Results of the experiments for the UCB algorithm for different values of the parameter c

The performance of the UCB algorithm with different values for the parameter c is displayed in Figure 42. As we can see, the algorithms perform similar with each parametrization, when only few simulations are available. But in the right half of the diagram, in situations with more than 256 available simulations, the parametrization $c = 0.1$ is ahead of the value 0.01 and significantly better than the other values. We conclude that less exploration and more exploitation is better for the UCB algorithm in situations where more simulations are available.

5.7.2 UCT

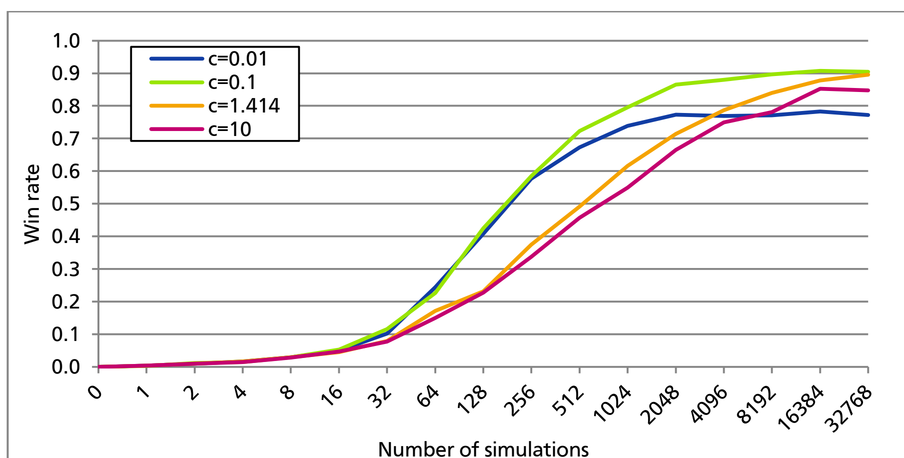


Figure 43: Results of the experiments for the UCT algorithm for different values of the parameter c

In Figure 43 the results for the parametrization of c used in the UCT algorithm are displayed. We see that the impact of the different values is even more important than in the UCB algorithm. The performance of the algorithms running with $c = 1.414$ and $c = 10$ are already worse in the

experiments with 32 available simulations compared to the values 0.1 and 0.01. This means that too much exploration is a problem for the UCT algorithm or in other words: the UCT algorithm should start early with the exploitation of learned knowledge. This is fully explainable since the UCT algorithm gathers knowledge about atomic actions and not about complete moves. Therefore, UCT can use its knowledge of the atomic actions to direct the remaining computational power to promising regions of the search tree. In the experiments with a higher amount of available simulations the gap between the more exploiting and the more exploring versions of UCT shrinks but still shows a significant difference.

We therefore conclude that the parametrization of the algorithm UCT is more important than in UCB. In both algorithms we see that too much exploration leads to worse results in comparison to using more exploitation.

5.8 The Impact of the Number of Simulations

If we want to compare a UCB bandit approach with a UCT tree approach, one important parameter for both algorithms is the number of simulations they can use to estimate the true utility value of an atomic action or a complete move. To investigate this circumstance we list the result of experiments where the number of simulations was varied in Table 17.

Simulations	Evaluated game states		Average move length		Win rate	
	UCB	UCT	UCB	UCT	UCB	UCT
0	13,016	12,222	4.07	1.00	1.00	0.00
1	16,259	15,499	3.97	1.87	0.996	0.004
2	17,717	16,977	3.93	2.10	0.98	0.02
4	18,710	17,993	3.88	2.28	0.96	0.04
8	19,026	18,336	3.85	2.41	0.952	0.048
16	19,782	19,136	3.82	2.56	0.930	0.070
32	20,562	20,015	3.76	2.72	0.854	0.146
64	20,871	20,427	3.71	2.85	0.780	0.220
128	20,878	20,582	3.64	2.94	0.684	0.316
256	20,531	20,336	3.60	3.03	0.617	0.383
512	20,151	20,144	3.55	3.10	0.497	0.503
1,024	19,382	19,441	3.55	3.16	0.458	0.542
2,048	18,747	18,878	3.52	3.20	0.413	0.587
4,096	18,009	18,189	3.53	3.24	0.387	0.613
8,192	17,346	17,548	3.53	3.26	0.373	0.627
16,384	15,663	15,794	3.55	3.25	0.404	0.596
32,768	13,441	13,469	3.56	3.23	0.455	0.545
65,536	11,676	11,616	3.57	3.22	0.499	0.501

Table 17: Results of the experiment UCB versus UCT without post-calculation moves

For this experiment, no post-calculation moves were made in both algorithms. This means that the UCT algorithm cannot perform very well because it cannot build an appropriately big tree and therefore does not use the available mana. But UCB also has a problem with a small number of simulations. Since there are a lot of bandits to evaluate, especially without pruning, UCB cannot evaluate them well enough. Therefore, the selection of a bandit after only a few simulations is rather random. However, the results show that such a mostly randomly selected move is better than a short move that was investigated by only a few simulations. The number of evaluated game states varies for both algorithms because in this experiment we evaluated the algorithms only in their own game states.

5.9 Using Post Monte Carlo Moves

We will now show how the result changes if we apply random post-calculation moves. To do so, we repeat the experiment above with the same setup but add a random post-calculation move to both players. We will add the fast random action player and the slower, more storage-consuming random move player max length described in section 5.4.

Simulations	Evaluated game states		Average move length		Win rate	
	UCB	UCT	UCB	UCT	UCB	UCT
0	17,499	17,160	3.80	2.99	0.743	0.257
1	17,903	17,653	3.69	3.18	0.678	0.322
2	17,787	17,512	3.71	3.20	0.690	0.310
4	17,946	17,706	3.70	3.22	0.666	0.334
8	17,926	17,703	3.70	3.27	0.663	0.337
16	17,915	17,731	3.66	3.31	0.622	0.378
32	18,069	17,954	3.64	3.35	0.582	0.418
64	18,072	18,000	3.62	3.4	0.548	0.452
128	18,112	18,137	3.58	3.44	0.488	0.512
256	17,984	18,059	3.59	3.46	0.458	0.542
512	17,726	17,867	3.57	3.47	0.409	0.591
1,024	17,542	17,705	3.55	3.51	0.395	0.605
2,048	17,003	17,193	3.54	3.50	0.382	0.618
4,096	16,803	16,992	3.56	3.51	0.386	0.614
8,192	16,199	16,372	3.56	3.50	0.386	0.614
16,384	15,407	15,542	3.57	3.47	0.407	0.593
32,768	14,105	14,163	3.58	3.43	0.442	0.558
65,536	12,223	12,178	3.63	3.36	0.506	0.494

Table 18: Results of the experiment UCB versus UCT with RAP post-calculation moves

Simulations	Evaluated game states		Average move length		Win rate	
	UCB	UCT	UCB	UCT	UCB	UCT
0	15,256	15,316	3.61	3.66	0.459	0.541
1	15,841	15,805	3.65	3.56	0.538	0.462
2	15,987	15,916	3.66	3.54	0.553	0.447
4	16,024	15,988	3.67	3.54	0.554	0.446
8	16,118	16,048	3.67	3.53	0.565	0.435
16	16,320	16,281	3.65	3.56	0.539	0.461
32	16,464	16,441	3.64	3.56	0.521	0.479
64	16,549	16,582	3.63	3.59	0.487	0.513
128	16,542	16,614	3.61	3.60	0.462	0.538
256	16,727	16,881	3.58	3.62	0.409	0.591
512	16,505	16,675	3.58	3.62	0.397	0.603
1,024	16,365	16,553	3.58	3.63	0.384	0.616
2,048	16,307	16,503	3.57	3.62	0.386	0.614
4,096	15,921	16,139	3.57	3.62	0.374	0.626
8,192	15,717	15,899	3.59	3.60	0.391	0.609
16,384	15,064	15,220	3.58	3.58	0.397	0.603
32,768	13,919	13,991	3.60	3.52	0.447	0.553
65,536	12,315	12,238	3.65	3.45	0.524	0.476

Table 19: Results of the experiment UCB versus UCT with RMP max length post-calculation moves

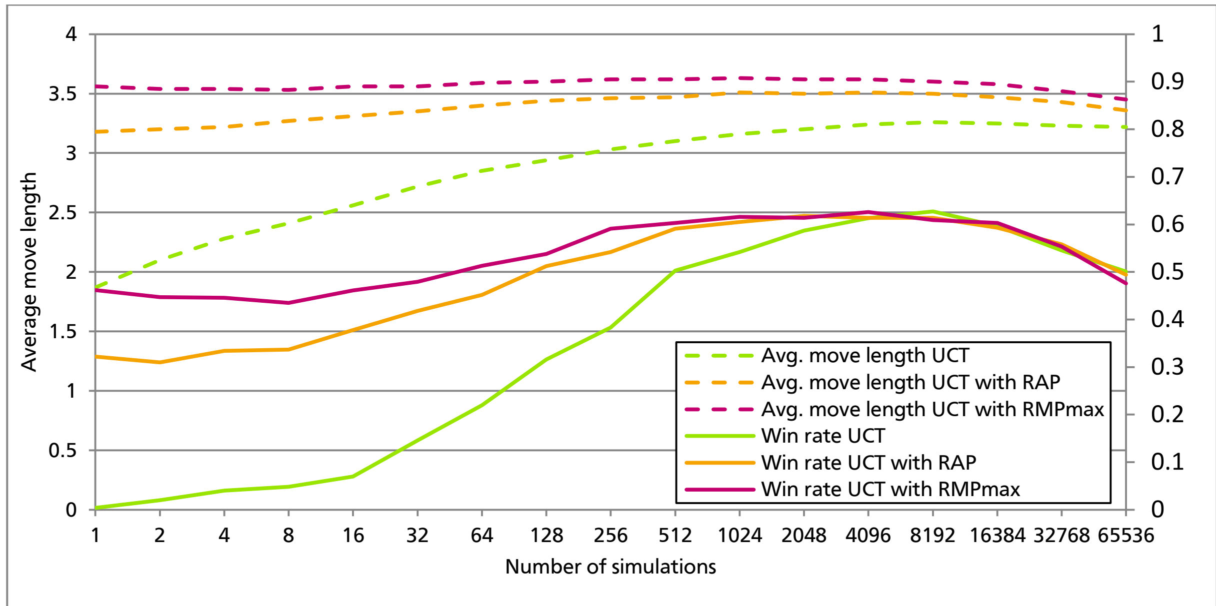


Figure 44: Visualization of the move lengths and the win rates for different post-calculation methods

5.10 Final Comparison between UCB and UCT

In the last experiments, we compare the performance of UCB and UCT using their best configurations. This means, we set the parameter $c = 0.1$ and used RMPmax as post Monte Carlo phase. In Figure 45, we see a comparison of the UCT algorithm and the UCB algorithm against the two random players RAP and RMPmax. We see a better performance of the UCB algorithm when only few simulations are available. In the middle of the diagram, beginning with 64 available simulations to 2,048 available simulations the UCT algorithm surpasses UCB. This advantage of UCT can be reduced by UCB using a high simulation count. In the end, UCB is a little bit ahead compared to UCT.

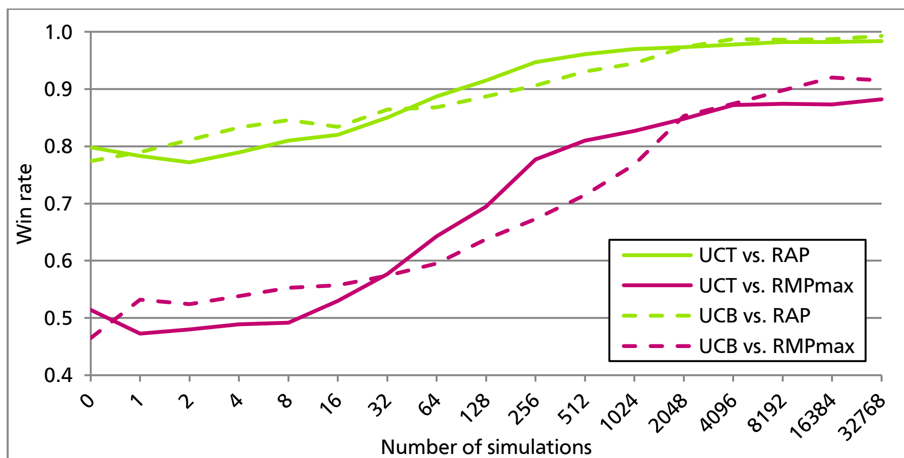


Figure 45: UCB (dashed) and UCT (solid) versus RAP and RMPmax

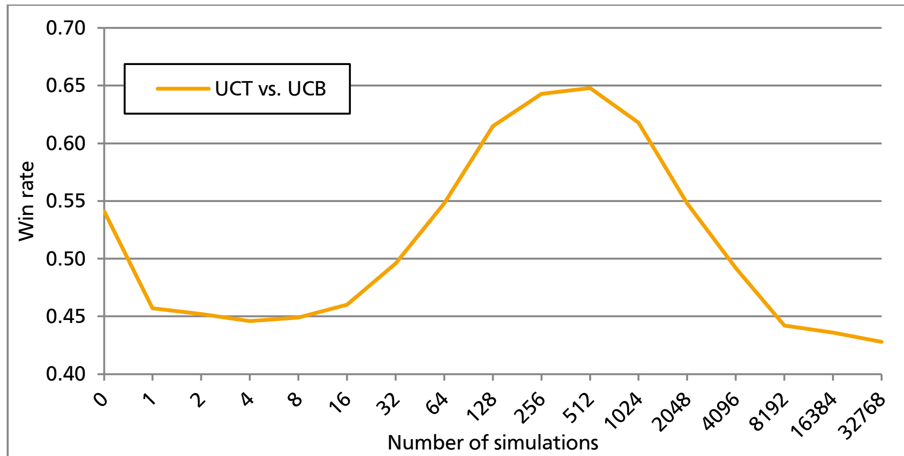


Figure 46: Win rate of the UCT algorithm against UCB

Figure 46 compares UCT and UCB directly. Similar to the experiment in Figure 45, we see that UCB is better when only few simulations are available. UCT surpasses UCB when more simulations are available and with the experiments with the highest amounts of available simulations UCB is the better algorithm again.

6 Conclusion

To begin with the conclusion, we revisit the research question of this thesis stated in section 1.4. As described, we wanted to find out if the structured Monte Carlo tree search approach can achieve better results than the flat Monte Carlo bandit approach in the game Hearthstone. To get an answer to this question, we implemented the game, a UCB bandit algorithm, a UCT tree algorithm, and different types of random players. The hypothesis stated in the research question was that the UCT algorithm should be able to gain an advantage over the UCB approach, because UCT should be able to transfer knowledge about atomic actions between moves. We could not confirm that hypothesis.

We found that the game tree in Hearthstone is enormously big due to the combination possibilities of many cards and the fact that the hand cards and the deck cards of a player are hidden information for the other player (see 3.2). Therefore, it is not easily possible to reduce the game tree to be better manageable. Another factor influencing the size of the game tree is the high branching factor in Hearthstone which results from the combination possibilities of atomic actions (see 3.3).

We therefore reduced the search range to the move tree and did not investigate the whole game tree. This means, the bandit approach created only bandits that cover the possible moves for one ply and the tree approach did not build the tree beyond the end of the current ply. Nevertheless, the simulations made to estimate the value of the moves and actions were executed until the game was finished, i.e. through the complete game tree.

The implemented UCB algorithm creates one bandit for each possible move for a given game state. This means that the game tree becomes wide and shallow in comparison to the UCT approach. Some disadvantages of the bandit approaches in comparison to the tree approaches result from this bandit strategy in addition with big move trees. The first disadvantage is the creation of the moves themselves. Since there might be thousands of possible moves in a game state, thousands of bandits have to be created. Since the creation of moves and the calculation of the effects of actions is not a trivial task in Hearthstone, this operation needs a significant big computational effort. We did not investigate timing conditions in detail in this thesis but during the execution of the experiments it became apparent that this is a major drawback of the UCB algorithm. We observed this effect not only during the execution of the UCB algorithm but also on the random, bandit-based approaches RMP (see 3.7.3) and RMPmax (see 3.7.4). We therefore conclude that this problem belongs to the general bandit creation and is not a disadvantage of UCB alone. Another big disadvantage is UCB specific. It results directly from the number of available moves. Since UCB has to calculate the UCB value for all bandits in each iteration, this becomes another computational problem. If we assume an average branching factor of 10,000 and 10,000 iterations, the UCB player has to make 100 million UCB value calculations every time he has to make a move. Since these two disadvantages are related to the computational time they are not present in the experiment results. Nevertheless, they both have a huge impact on the performance of the UCB algorithm.

The UCT algorithm creates a move tree for a given game state in contrast to the UCB approach. This means that the corresponding game tree using a UCT player will become deeper and narrower than the game tree resulting from the UCB player. The UCT approach does not have these two disadvantages which result from the number of available moves mentioned for the UCB player. Since the UCT algorithm creates the move tree partially on demand, there is no need to generate all moves before the algorithm starts. Furthermore, the amount of generated moves is lower than using UCB because the move tree does not have to be built completely. If the algorithm finds out that an action will most likely not be part of a good move, UCT does not explore this part of the move tree whereas UCB would create these moves. Furthermore, UCT does not have to evaluate the UCB value for all moves in each iteration like UCB does. Only all children of the nodes on the path taken through the tree have to be evaluated. This results in a much lower computational effort.

The next conclusion concerns the move selection of the UCT algorithm. Since we decompose a move for the UCT player into atomic actions, the goal was not to select to best child of the root node in the move tree but to find the best node within the tree. This task differs significantly from the task to be solved in games like chess and Go where only one action is possible in each ply. The first important difference is that the anytime property of the UCT algorithm is violated to some extent. It is still true that we can stop the algorithm at any time and get a result, but the experiments show (see 5.6) that the performance of the UCT algorithm does not only increase because of better evaluated moves. A major factor for the performance is that the move tree was built deep enough to cover long moves. If we stop the UCT algorithm early, it will return a short and therefore most likely an incomplete move. This incomplete move will be worse than a longer move in most cases. If the UCT algorithm does not have a sufficient amount of simulations available to build the tree and to discover complete moves it will perform poor. UCB does not have this disadvantage because it generates all possible moves in advance. We compensate this weakness of the UCT algorithm by adding randomly selected moves to the incomplete moves returned by UCT. The experiments show a significant increase in the win rate when only few simulations (< 1,000) are executed (see 5.9). Using the fast RAP extension resulted in a worse performance than using the more computational expensive RMP and RMPmax extensions. One of the major findings in this thesis is therefore that the UCT performs worse than UCB when only few simulations are available in a game where the moves consist of atomic actions. This result was not expected since, in other games, the UCT algorithm performs better than UCB when only few simulations are available. Hence, this is an interesting finding.

Furthermore, we found that the simulations in the game of Hearthstone are rather slow. This is an issue because these algorithms rely on the possibility that they can execute a big amount of simulations to estimate the value of a game state. The effect of atomic actions and therefore complete moves in Hearthstone are complex to evaluate (see 4.2.3). This is problematic since this limits the overall amount of simulations executable by the algorithms. During the experiments, the algorithms had a limit of about 65,000 simulations. If we put this in relation to the big amount of moves, 65,000 simulations are not much. Therefore, Monte Carlo algorithms might not be the best choice for the game Hearthstone and we suggest evaluating other methods in section 7.4 to judge the suitability of Monte Carlo methods in general. However, the increasing performance of the UCB algorithm in comparison to UCT together with an increasing amount of simulations indicate that moves can be simulated sufficiently precise to compensate the advantage of UCT which best performs with a medium amount of simulations.

Finally, we conclude that the game Hearthstone seems to be another challenging game for artificial agents and offers many more research questions. We give some pointers to newly raised questions in chapter 7 and would be very interested in further work addressing this game.

7 Further Work

Last but not least, we describe some further work which would be very interesting to investigate and will probable lead to new insights in the algorithms and in possible improvements. Due to the time limitations for this thesis it was not possible to investigate them in detail. Therefore, we phrase them for further research.

7.1 Pruning in Monte Carlo Algorithms

In the game of Hearthstone, the effect of an action sequence generally depends on the order of the actions in the sequence. But this is not always true. For some actions it does not matter in which order they are played. They lead to the same game state. For example, if there is an action A, which summons a minion, an action B, which draws a card, and an action C, which uses the special hero power of a player. If we assume that these three actions are independent from each other and if we can and want apply all three actions in a single ply, there are six possible permutations to do this: ABC, ACB, BAC, BCA, CAB, and CBA. Since they are independent and therefore do not affect the outcome of another action, it does not matter in which order they are played. This means, a player can choose an arbitrary sequence of the six available sequences and will always get to the very same game state. Therefore, it is ineffective to investigate all six permutations. It is sufficient to investigate only one of them since the utility value of all six sequences is the same. We can therefore pick an arbitrary sequence and drop the rest during the execution of the Monte Carlo algorithms. The algorithms then can evaluate the remaining sequence better and get a better estimation of the utility value for all six sequences. If the sequences are followed by additional actions possible, the effect is even bigger. For example, drawing a card because of applying action B can create an additional action D.

7.2 Improving Monte Carlo Methods

The introduced Monte Carlo methods used random simulation to estimate the value of game states. As described in section 7.6, it is questionable if random simulations deliver a good estimation of the utility value. Therefore, both algorithms, UCB and UCT, could benefit from a more sophisticated simulation. The simulations of the game could be more precise. For example, in the algorithms a simple heuristic could be used to control the simulated players. This would lead to better results because the simulated players would play more intelligently and more like real players would play. The drawback of this method would be an even more complex and therefore more computational expensive simulation.

7.3 Game Tree Search

The search investigated in this thesis was limited to the move tree because the move tree is already very large in average. Nevertheless, an expansion of the search could be beneficial. Both algorithms are suited to search beyond the current ply. This improvement would be very interesting in combination with the pruning described in section 7.1 because then more parts of the game tree could be investigated. However, due to the large branching factor induced by the assignment of the hidden cards it is questionable if the deeper and therefore more imprecise exploration of the game tree would be beneficial.

7.4 Usage of Score Functions and Reinforcement Learning

Both approaches, UCB and UCT, use random simulations to estimate the utility value of a game state. We found that executing simulations in Hearthstone is rather slow and it is not clear how well random simulations estimate the utility value of a game state. Therefore, it would be interesting to investigate other approaches and compare the performance to the Monte Carlo algorithms. One approach would be estimating the value of the game states with a utility function. To do this, we have to extract features from the game state. This could be, for example, the health and attack values of the heroes and minions, the sum of the mana values of the minions on the battlefield, the round number, the hand cards, and many more. If these values are combined in a weighted sum, they can maybe give a good estimate of the utility function of game states. The weights could be learned with reinforcement learning and self-play.

7.5 Opponent Modelling

In the current implementation of the random simulations, the hand and deck cards of the opponent are assigned randomly. This means that the chance of a specific card to be or not to be in the hand of the opponent is the same for every card. Following from this, the playing strength of the randomly simulated opponent is rather low since synergy effects of the cards are not considered. But a rational player will build his deck in a way that he can use such synergy effects. For example, some decks are built to produce as much pressure as possible on the opponent within the first rounds. If a player with such a rush deck does not win in the first rounds he will probably lose because the cards are not well suited to player in later rounds. If such a constellation can be detected by a player, it would be very helpful to incorporate this knowledge about the opponent. The player could then assign different probabilities to cards to be in the opponent's hand. In the example with the rush deck, the player could assign low cost cards to the hand of the player with a higher probability than more expensive cards and therefore could simulate the behavior of the opponent more precisely.

7.6 Accuracy of Random Simulations

Furthermore, an investigation on the preconditions for successful Monte Carlo simulations would be interesting. Monte Carlo simulations need fast simulations which estimate the utility value of a given game state. In the conclusions, we described that the simulations in Hearthstone are rather slow and, considering the big number of moves, only few simulations can be executed. But fast simulations are not the only precondition for successful Monte Carlo algorithms. It is vital that the simulations made for a game state converge against a good estimate for the true utility value of the game state. This precondition may or may not be fulfilled in Hearthstone and it is not clear how this could be investigated. One possibility would be to compare the estimated value obtained by random simulation with an average value obtained by playing with strong players from this game state forward. Another chance to get a notion of the precision of the estimated value would be to use learned or handcrafted utility functions.

List of Figures

Figure 1: A drawing of the game tree for the simple one player game 6
Figure 2: A simple two player game 7
Figure 3: Game tree of a two player zero-sum game 8
Figure 4: Game tree of a game with chance nodes (s3-s6) 10
Figure 5: Expectimax solution of the game displayed in Figure 4 12
Figure 6: A game tree with an information partition of game states s1-s3 for player 2 13
Figure 7: A game tree in which both players have hidden information 14
Figure 8: Example game state with hidden information 2a and move A 14
Figure 9: Illustration of the flat approach 16
Figure 10: Illustration of the structured approach 16
Figure 11: Screenshot of the game Hearthstone 20
Figure 12: The Hearthstone minion card Stormwind Champion 20
Figure 13: The Hearthstone spell card Fireball 21
Figure 14: The Hearthstone weapon card Perdition's Blade 21
Figure 15: A snippet of a game tree for the game Hearthstone 26
Figure 16: Possible game states reachable with one action available 28
Figure 17: Possible game states reachable with 2 actions available 28
Figure 18: Possible game states reachable with 3 actions available 29
Figure 19: Diagram of the relation between the UCB value and the number of simulations for each bandit 33
Figure 20: Diagram of the two component of the UCB value: the exploration and the exploitation 34
Figure 21: Visualization of the isometrics for the different UCB best child selection criteria 36
Figure 22: Initial move tree used to demonstrate an MCTS iteration 38
Figure 23: Visualization of the MCTS selection phase 39
Figure 24: Visualization of the MCTS expansion phase 40
Figure 25: Visualization of the MCTS simulation phase 41
Figure 26: Visualization of the MCTS backpropagation phase 42
Figure 27: Move tree for a random action player 45
Figure 28: Entity-relationship model of the game state 50
Figure 29: A typical game situation in Hearthstone 51
Figure 30: A sample game state in Hearthstone 52
Figure 31: Visualization of the event queue 56
Figure 32: Sequence diagram of an experiment with the server/player interaction 57
Figure 33: Screenshot of the graphical user interface 58
Figure 34: Illustration of the confidence interval for two random players and $\alpha = 0.05$ 63
Figure 35: Detailed view of the confidence interval for two random players and $\alpha = 0.05$ 64
Figure 36: Detailed view of the confidence interval for two random players and $\alpha = 0.05$ for 2000 games 64
Figure 37: Scattered plot of the results of 300 games. The results are green below the lower confidence bound, purple above the upper confidence bound, and gold within the confidence interval. 65
Figure 38: Frequency distribution of the results shown in Figure 37. 65
Figure 39: Visualization of the results in Table 15 68
Figure 40: Visualization of the impact of different maximal amounts of bandits for UCB 69
Figure 41: Visualization of the win rates in Table 16 71
Figure 42: Results of the experiments for the UCB algorithm for different values of the parameter c 72
Figure 43: Results of the experiments for the UCT algorithm for different values of the parameter c 72
Figure 44: Visualization of the move lengths and the win rates for different post-calculation methods 75
Figure 45: UCB (dashed) and UCT (solid) versus RAP and RMPmax 75
Figure 46: Win rate of the UCT algorithm against UCB 76

List of Tables

Table 1: Overview of four possibilities to define intelligence....4
Table 2: Overview of the complexity of 5 well-known games....9
Table 3: Amount of possible action sequences for up to 14 atomic actions....30
Table 4: Available moves, selection probabilities and move lengths for the move tree in Figure 27....45
Table 5: Listing of all player pre-impact events....54
Table 6: Listing of all non-player pre-impact events....54
Table 7: Listing of two post-impact events....55
Table 8: Parameter description for the random action player....60
Table 9: Parameter description for the random move player....60
Table 10: Parameter description for the upper confidence bound player....61
Table 11: Parameter description for the Upper Confidence Bound Applied to Trees player....61
Table 12: Results for simulations of two random players for different confidence intervals....62
Table 13: Comparison of average move length in different game state sets evaluated in 1,600 games 66
Table 14: Evaluation results to investigate the behavior of random players against a UCT player with 40 simulations....67
Table 15: Distribution of the move lengths with and without pruning....68
Table 16: Evaluation results to investigate the behavior of the tree size of UCT....70
Table 17: Results of the experiment UCB versus UCT without post-calculation moves....73
Table 18: Results of the experiment UCB versus UCT with RAP post-calculation moves....74
Table 19: Results of the experiment UCB versus UCT with RMP max length post-calculation moves...74

List of Listings

Listing 1: Pseudocode of the minimax algorithm....9
Listing 2: Pseudocode of the Expectimax algorithm 11
Listing 3: Pseudocode for a generic bandit algorithm....32
Listing 4: Pseudocode of the UCB algorithm 35

References

[1] Stuart J. Russell and Peter Norvig, *Artificial Intelligence: A Modern Approach*, 3rd ed.: Pearson Education Inc., 2010.

[2] Alan M. Turing, "Computing Machinery and Intelligence," *Mind*, no. 49, pp. 433-460, 1950.

[3] John Nash, "Non-Cooperative Games," *The Annals of Mathematics*, vol. 54, no. 2, pp. 286-295, 1951.

[4] Martin J. Osborne and Ariel Rubinstein, *A Course in Game Theory*. Cambridge, Massachusetts, USA: MIT Press, 1994.

[5] Peter Kurzdorfer, *The Everything Chess Basics Book*.: Adams Media, 2003.

[6] Martin Müller, "Computer Go," *Artificial Intelligence*, vol. 134, pp. 145–179, 2002.

[7] Peter Auer, "Using confidence bounds for exploitation-exploration trade-offs," *Journal of Machine Learning Research*, vol. 3, pp. 397–422, 2002.

[8] Peter Auer, Nicolo Cesa-Bianchi, and Paul Fischer, "Finite-time Analysis of the Multiarmed Bandit Problem," *Machine Learning*, vol. 47, no. 2-3, pp. 235-256, 2002.

[9] Shlomo Zilberstein, "Using anytime algorithms in intelligent systems," *AI Magazine*, vol. 17, no. 3, pp. 73-83, 1996.

[10] John Hubbard, Dierk Schleicher, and Scott Sutherland, "How to find all roots of complex polynomials by Newton's method," *Invent. Math.*, vol. 146, pp. 1–33, 2001.

[11] Rémi Coulom, "Effcient Selectivity and Backup Operators in Monte-Carlo Tree Search," in *5th International Conference on Computer and Games*, Turin, 2006, pp. 72–83.

[12] Levente Kocsis and Csaba Szepesvári, "Bandit based Monte-Carlo Planning," in *Proceedings of the 15th European Conference on Machine Learning*, Berlin, 2006, pp. 282–293.

[13] Richard J. Lorentz, "Amazons Discover Monte-Carlo," in *Lecture Notes in Computer Science Volume 5131*, Beijing, China, 2008, pp. 13-24.

[14] David Silver and Gerald Tesauro, "Monte-Carlo Simulation Balancing," in *Proceedings of the 26th Annual International Conference on Machine Learning*, Montreal, 2009, pp. 945–952.

[15] C. J. Clopper and E. S. Pearson, "The Use of Confidence or Fiducial Limits Illustrated in the Case of the Binomial," *Biometrika*, vol. 26, no. 4, pp. 404-413, 1934.