Minimax based Artificial Intelligences for Tourality

Minimax basierte Künstliche Intelligenzen für Tourality
Bachelor-Thesis von Daniel Tanneberg
Mai 2013

TECHNISCHE UNIVERSITÄT DARMSTADT

Fachbereich Informatik
Knowledge Engineering Group

Minimax based Artificial Intelligences for Tourality
Minimax basierte Künstliche Intelligenzen für Tourality

Vorgelegte Bachelor-Thesis von Daniel Tanneberg

1. Gutachten: Prof. Dr. Johannes Fürnkranz
2. Gutachten: Dr. Jens Gallenbacher

Tag der Einreichung:

Erklärung zur Bachelor-Thesis

Hiermit versichere ich, die vorliegende Bachelor-Thesis ohne Hilfe Dritter nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus Quellen entnommen wurden, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, den 13.05.2013

(Daniel Tanneberg)

Summary

This Bachelor-Thesis is about the *Minimax* algorithm and its *AlphaBeta-Pruning* extension, which is known from Game-Theory, and the application of it to the Tourality-Game. For this purpose the *Entwicklungs und Simulationsumgebung* **ESU** is used, which was developed for the *Bundeswettbewerb Informatik*. The goal is to develop several strong opponents for the participants of the *Bundeswettbewerb Informatik*.

The behavior in the game of the Minimax algorithm is investigated compared to several other algorithms and additionally the influence on the behavior of different parameters of the Minimax algorithm is evaluated. This evaluation is done against three algorithms from different fields and the parameters are investigated in small groups or individual.

The thesis starts with an introduction to the used **ESU** framework, the general explanation of the Tourality-Game and afterwards the used implementation of it based on the **ESU** framework. Next the main problem which has to be solved and the two underlying algorithms are explained in general. The *Breadth First Search* algorithms, known from graph-theory, and the *Minimax* algorithm with its *AlphaBeta-Pruning* extension. The next section is about the specific implementation of the algorithms using the framework and applied to the game, different versions of them and how the informations gathered from the *Breadth First Search* are used while creating the *Minimax* game tree. Afterwards the behavior of the Minimax algorithm is evaluated through evaluation tournaments against the artificial intelligences based on the different other algorithms and the parameter evaluation tournaments. In the last section the results are recapped and some possible improvements and extensions are discussed.

Zusammenfassung

Die vorliegende Bachelor-Thesis beschäftigt sich damit den aus der Spieletheorie bekannten Minimax Algorithmus und dessen *AlphaBeta-Pruning* Erweiterung auf das Tourality Spiel anzuwenden und zu untersuchen. Dazu wird die *Entwicklungs und Simulationsumgebung* **ESU** verwendet, die für den *Bundeswettbewerb Informatik* entwickelt wurde. Ziel ist es dadurch verschiedene spielstarke Gegner für die Teilnehmer des *Bundeswettbewerb Informatik* zu entwickeln.

Der Minimax Algorithmus wird gegen verschiedene andere Algorithmen im Hinblick auf die Spielstärke verglichen und außerdem wird der Einfluss von verschiedenen Parametern auf das Spielverhalten der Minimax KI untersucht. Getestet wird gegen drei verschiedene Algorithmen aus unterschiedlichen Bereichen und die Parameter werden in verschiedenen kleinen Gruppen oder einzeln genauer betrachtet.

Die Arbeit beginnt mit einer Einführung in das verwendete **ESU** Framework, der Beschreibung des Tourality Spiels allgemein und anschließend der auf diesem Framework basierende und verwendeten Implementierung des Spiels. Nachfolgend wird das zu lösende Hauptproblem erklärt und die zwei verwendeten Hauptalgorithmen allgemein vorgestellt. Der aus der Graphentheorie bekannte *Breadth First Search* und der *Minimax* Algorithmus inklusive der *AlphaBeta-Pruning* Erweiterung. Im nächsten Abschnitt geht es um die spezielle Implementierung der Algorithmen innerhalb des Frameworks und auf das Spiel angewandt, die verschiedenen Varianten und wie die Informationen des *Breadth First Search* Algorithmus schon beim Aufbau des *Minimax*-Spielbaums verwendet werden. Anschließend folgt die Evaluierung des Spielverhaltens durch Turniere gegen die auf den verschiedenen anderen Algorithmen basierenden Künstliche Intelligenzen und die Turniere für die Parameter Evaluation. Abschließend werden die Ergebnisse zusammengefasst und mögliche Erweiterungen und Verbesserungen betrachtet.

Contents

List of Figures

List of Tables

1 Introduction

1.1 Motivation

The idea and motivation for this work is principally based on two thoughts: to provide some reference AIs to the participants of the BWINF, see **Section 1.2** and **Section 1.3**, which they can challenge to test themselves and to analyse the behaviour of the chosen Minimax algorithm and its AlphaBeta-Pruning extension in that particular game. To evaluate the strength of the Minimax based AIs, they play against several reference AIs based on different other algorithms to analyse their performance. Additionally one Minimax based AI is chosen to analyse the different parameters of the algorithm. For this purpose the parameters to test will be changed and this version plays against the unchanged version. By this the influence on the behaviour of the AI by the different parameters is analysed and the crucial parameters of the algorithm identified.

All evaluations are done by tournaments with many challenges and several values are calculated to sort the AIs, or the parameter values, by their strength or influence depending on different criteria.

1.2 Bundeswettbewerb Informatik

The *Bundeswettbewerb Informatik*[1] **BWINF** is a competition for young people up to 21 years, who haven't finished their job education and haven't started to study. It is carried by the *Gesellschaft für Informatik e. V.*[2], the *Fraunhofer-Verbund IuK-Technologie*[3], the *Max-Planck-Institut für Informatik*[4] and supported by the *Bundesministerium für Bildung and Forschung*[5].

It starts in September, takes about a year and consists of three rounds. The first round consists of five tasks, the second of three tasks and the third round is a colloquium for the approximately 30 best. The participants get to talk to computer scientists and have to solve two computer science problems in a team.

The **ESU**, see **Section 2**, framework and toolkit used in this work was developed for round two of the competition 2012/2013. But due to it's specification and functionality it will be used in future competitions too. ESU is mostly used for the implementation of the games but can be used to develop the AIs as well.

To make the whole task more exciting and fun for the participants, the idea about AIs and a game competition was introduced. Thus the participants don't have to solve only some abstract exercises, rather they can *play* against each other and instantly see how good their work is. These challenges give higher motivation to improve themselves, so they won't lose against another participant.

1.3 Online-Tournament

To afford this competition and interactive playing ESU isn't enough and the **BWINF KI Wettbewerbs-Plattform**[6] was developed additionally. On this online platform you can upload and edit your AIs, or even develop them only online without using ESU. There you can challenge the other participants and compete yourself with them.

As ESU the online platform is complete game independent, which means the admin can easily upload a new game and all the users can play the new game without changing anything, except they have to develop new AIs of course.

[1] http://www.bundeswettbewerb-informatik.de/
[2] http://www.gi.de/
[3] http://www.iuk.fraunhofer.de/
[4] http://www.mpi-inf.mpg.de/
[5] http://www.bmbf.de/
[6] http://turnier.bundeswettbewerb-informatik.de/

2 ESU & Tourality

2.1 ESU

The *Entwicklungs und Simulationsumgebung* **ESU** was designed and developed by three students of the **TU Darmstadt** - Markus Schröder, Sven Hertling and Daniel Tanneberg - for the use in the Bundeswettbewerb Informatik. The requirements were to build a system that could be used to implement a large amount of different games and automatically generates code for the game and templates in different programming languages for the Artificial Intelligences.

It should be easy to use and helping you with not game-related details such as the communication with the AIs, so you can focus on the game itself. The key principle was independence: operating-system independent, game independent and programming language independent.

2.1.1 Functionality

The functions of **ESU** are completely focused on implementing the behaviour of the game. All you have to do is to think about how you fit your game into the model used in **ESU** (see **Section 2.1.2**). A game is divided in several modules, the logic of the game (called *Spiellogik*), the world (called *Startzustand*) and the view of the game (called *Spielansicht*). So you have one definition of the game, but you can have several different implementations of it, just by implementing different logics. With the world module you can design different worlds (or levels) and with the view module you can have different ways of how the game looks like. All this modules of one game definition can be combined in every combination. After you have modeled your game, i.e. the definition of it see **Section 2.1.2**, **ESU** generates templates for all the modules. The templates for the AIs are generated in all supported languages.

2.1.2 Implementing a Game

The first step you have to do if you want to implement a game is the definition of it. The definition consists of two things:

1) the objects needed for the game

2) the actions an AI can do

The objects can have several attributes and the actions can have several parameters. After you have finished the definition you can move on to fill it with life.

To do this you'll have to create a new logic and a new world, a view is optional. When creating a new logic, **ESU** uses your definition created previously to generate a template for you which already implements a whole basic communication with the AIs as well as main game loop and helping functions. This means you just have to adjust it to your implementation of your game and implement the specific logic of it. Creating a world works the same way, you have automatically created functions for all your defined objects which you can place in the world to create a level.

2.1.3 Implementing an AI

To create an AI with **ESU** you just have to select the programming language you want to use and you'll get a template generated. Just like implementing the logic for the game you'll get everything that's not game-related out of the box and you just have to fill one method with your code. This method is automatically called from the simulator in **ESU** every time it's your turn. You get the actual worldstate as a parameter every turn.

In the definition of the game all available actions were declared. So to make your AI play you have to implement how to choose the *best* action(s) for this turn.

2.2 The Tourality-Game

Tourality is a grid based competitive multi-player game. The world consists of *obstacles* and *leaves*, which are the points you have to gather. If all leaves are gathered, or no player has time left to play, the game is over and the player with the most points wins. So the **goal** of the game is to gather as many leaves as possible, or/and at least more than your opponent.

2.2.1 Game-Definition

You can have very different implementations of this game with different sets of rules used. The version of the game used for this work is the version defined for the BWINF and has the following **rules**:

- the size of the world is 20x20 fields
- there are 40 obstacles and 40 leaves in it, randomly distributed
- you can't overstep the world
- the AIs have full information about the state
- two AIs can't stand on the same field
- a leaf is gathered automatically if the AI enters its field
- one challenge consists of two sweeps, with swapped starting positions and turn order

and **restrictions**:

- for one action the AI has 10 seconds to answer
- for one whole sweep the AI has 20 seconds
- the AI is allowed to use 512 MB of RAM

How this is transfered into the game definition of ESU you will see in **Section 2.2.2** and **Section 2.2.3**.

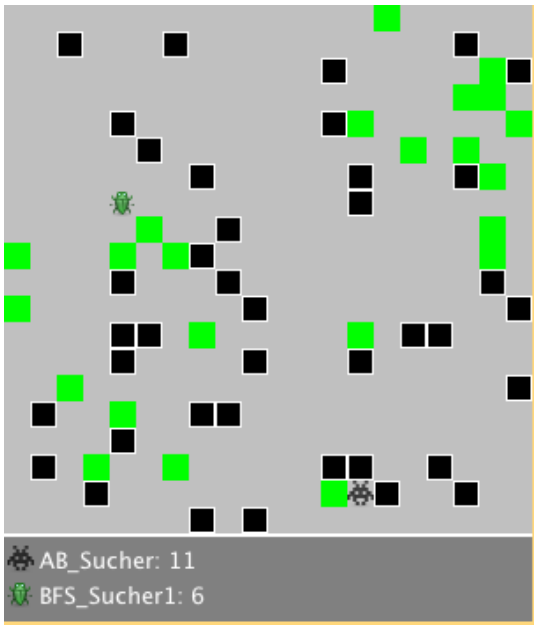

Figure 1: Screenshot of Tourality with a simple viewer. Black boxes are obstacles, green boxes are leaves.

2.2.2 World

A world contains of the objects you define. By placing them into the world you can create one configuration, or level. As described before you have to define a object for every type of object needed for the game. Every object can have several parameters. The game definition seen in **Section 2.2.1** leads to the following object declarations for Tourality in ESU:

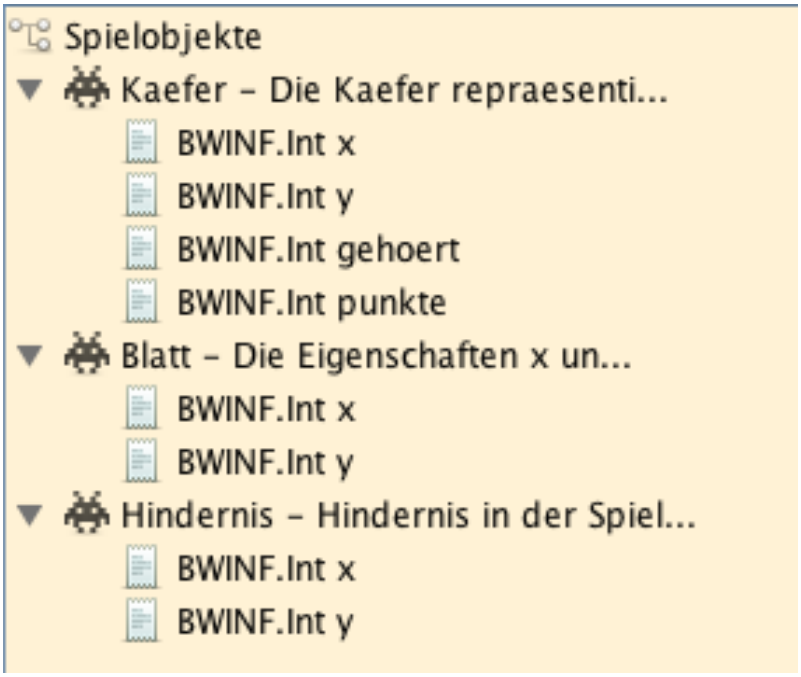

Figure 2: Definition in ESU of the objects for Tourality

Kaefer - represents the AI in the world and has the following parameters:

- *Integer* **x** - the x position of it
- *Integer* **y** - the y position of it
- *Integer* **gehoert** - *ID* of the AI, to find *your* Kaefer
- *Integer* **punkte** - points/leaves gathered so far

Blatt - represents a leaf and has the following parameters:

- *Integer* **x** - the x position of it
- *Integer* **y** - the y position of it

Hindernis - represents an obstacle and has the following parameters:

- *Integer* **x** - the x position of it
- *Integer* **y** - the y position of it

2.2.3 Actions

Here all actions that an AI can do have to be defined. In this implementation of Tourality all actions are deterministic and without parameters, but actions can have parameters just like objects too. The game definition seen in **Section 2.2.1** leads to the following action declarations for Tourality in ESU:

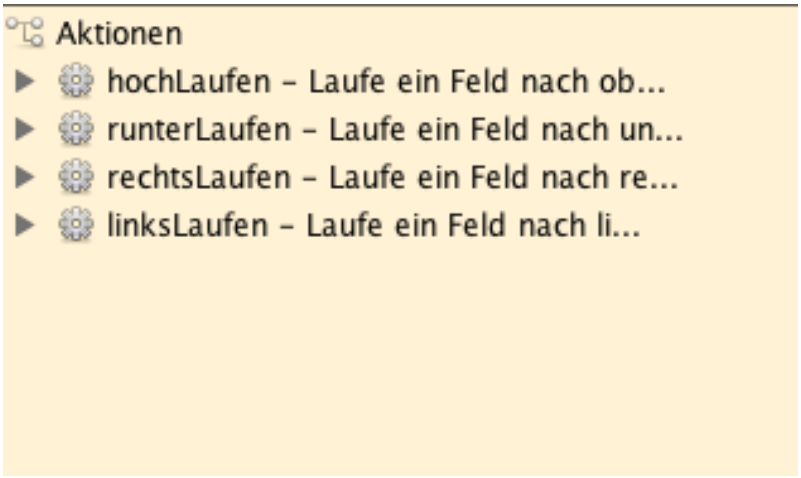

Figure 3: Definition in ESU of the actions for Tourality

- *hochLaufen* - move one step **up**
- *runterLaufen* - move one step **down**
- *rechtsLaufen* - move one step **right**
- *linksLaufen* - move one step **left**

3 Underlying Algorithms

In this section the problem you need to solve for playing the Tourality-Game will be described shortly and afterwards we'll take a general look on the two algorithms that are used for this in this thesis. The algorithms will be described in general to understand how they work and their properties before we will adapt them on the Tourality-Game in **Section 4**.

3.1 What's the problem to solve?

After we know the game with all of its settings we want to play it, thus we need to understand what we have to *do* to play it, i.e. the problem we need to solve. As the *goal* of the game was defined as gathering as many leaves as possible, it seems to be clear that the main problem we have to solve is finding ways, i.e. sequences of actions, to get to the leaves. In a *clear* world, without obstacles, the ways could be found very easily by just calculating the difference between the coordinates of your *Kaefer* and the leaf you want to reach and then use the appropriate action. But in our world there are obstacles, thus we need to find ways to the leaves which *walk around* the obstacles. Due to the used Tourality-Game has a discrete world and actions we can easily handle it as a graph or tree, we will see this in more detail later on in **Section 4**. But by handling the game as a graph or tree we can use a whole bunch of well known algorithms. To find the ways to the leaves the *Breadth First Search* algorithm and for dealing with the competitive game environment the *Minimax* algorithm with its *AlphaBeta-Pruning* extension were chosen. Let's take a general look on the algorithms to understand them and be able to use them later on.

3.2 Breadth First Search

The *Breadth First Search*[7] (**BFS**) is a standard algorithm and a classic algorithm for searching a graph or tree. Given a starting node **s** the BFS searches the graph systematically until it reaches the defined goal, or until it visited all nodes (if the graph is finite). Additionally it automatically returns a search tree containing all reachable nodes from the given starting node **s** as the root. BFS finds the distance, counted in edges used, from the starting node to all reachable nodes. By recursive going backwards from a node to the startnode you can construct the shortest way found by the algorithm. So relating to our problem described before, we can use the BFS to find ways, i.e. sequences of actions, to one or more leaves by using the position of our *Kaefer* as the starting node and defining the leaf or leaves as the goal. You will see this in more detail later on in **Section 4**.

The basic idea of BFS is to search first the whole *breadth* of the nodes before searching *deeper*, i.e. it discovers all nodes with distance d from s before a node with distance $d+1$ is found.

When performing BFS on a graph all nodes have a state which shows if the node has already been seen and searched deeper from it. At the start all nodes have the state *unseen* and the *searchqueue* contains only the starting node **s**. The state *seen* means that the node has been found by the algorithm, but hasn't looked further from it. If a node becomes *seen* the first time, it is added to the *searchqueue*, which contains all nodes that will be used for further exploration. Because each node is only added once when it has been discovered the first time, every node has exactly one parent node. When a node is removed from the *searchqueue* the algorithm searches all its children and all that are *unseen* are added to the *searchqueue*. The removed node gets the state *explored* and the next one is removed from the *searchqueue*. The nodes in the *searchqueue* form the current *search-boarder*, separating the *seen* and *unseen* nodes. The algorithm terminates if the searchgoal has been reached, i.e. if it's removed from the searchqueue to be explored, or if all nodes are *explored* (this of course only holds for finite graphs). The searchgoal can contain more than only one node. If several nodes should be found the terminate test needs to be extended, because the algorithm can't just terminate the search if a node is added or removed from the queue. For example the searchgoal can be a list of nodes and when a node of it is found it gets removed from the list and the BFS terminates if the searchgoal list is empty.

[7] introduced in Moore (1959) and Lee (1961)

The following example[8] uses colors to represent the states of the nodes. *Unseen* nodes are *white*, *seen* nodes are *lightgrey* and *explored* nodes are *darkgrey*. The value inside the nodes represent the distance to start node **s** as edges needed to get there. **Q** represents the *searchqueue*, which contains the *seen* nodes, with the distance of every node inside of it. The **bold** edges are *used* by the algorithm. The searchgoal contains only **v**. Again, you can think about **s** as the position of our *Kaefer*, **v** as a leaf we want to reach and the graph as the world. Thus you can relate this basic example to our certain problem.

1) all nodes except **s** are initially *white* with distance ∞, **Q** contains {(**s**,0)}

2) **s** got removed from **Q** and is explored, found **r** and **w**, **Q** = {(**w**,1),(**r**,1)}

3) **w** got removed from **Q** and is explored, found **t** and **x**, **Q** = {(**r**,1),(**t**,2),(**x**,2)}

4) **r** got removed from **Q** and is explored, found **v**, **Q** = {(**t**,2),(**x**,2),(**v**,2)}
- *Note*: searchgoal **v** is added to **Q**, but isn't *found* yet

5) **t** got removed from **Q** and is explored, found **u**, **Q** = {(**x**,2),(**v**,2),(**u**,3)}
- *Note*: **t** *sees* **x** too, but **x** is already *seen*, so this edge isn't considered

6) **x** got removed from **Q** and is explored, found **y**, **Q** = {(**v**,2),(**u**,3),(**y**,3)}

In the next step **v** will be removed from **Q** and the algorithm terminates.

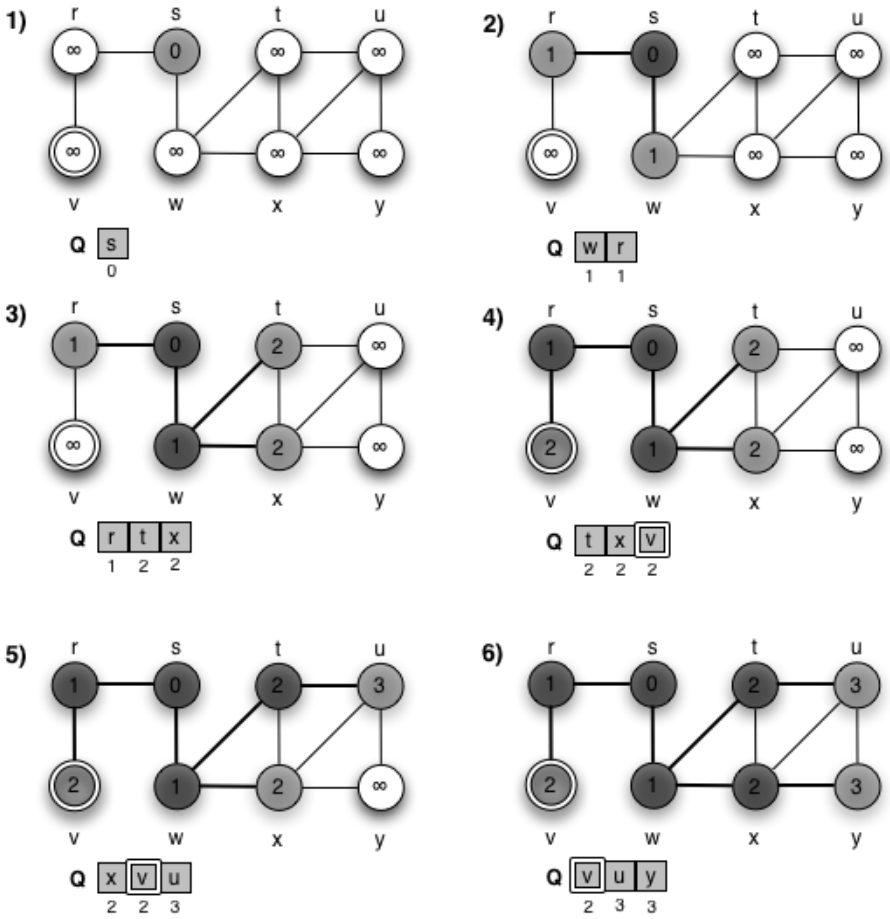

Figure 4: BFS on example graph

[8] adapted example based on Cormen et al. (2001)

3.3 Minimax

The *Minimax*[9] algorithm finds an *optimal* strategy for a game by searching the *game tree*. It was developed for *deterministic perfect-information two-player zero-sum* games but can be extended and used for other types of games as well. Although it can be used for *n-player* games, we talk about the *two-player* version, because this is the one we'll use for Tourality. So we have the player **Max**, who tries to maximize his outcome, and the player **Min**, who tries to minimize *Max*'s outcome.

3.3.1 Deterministic perfect-information zero-sum game

The Tourality implementation used in this work is a *deterministic perfect-information two-player zero-sum* game, so we'll have to explain those properties first.

Deterministic in this context means that there is no stochasticity that influences the game. All states and all actions are complete deterministic. So if an AI chooses the action *a* in state *s* it will *always* end up in the next state *s'*. In Tourality this implies, whenever an AI chooses to make the action *runterLaufen* - and there's no obstacle or the border of the world - it will move one field down. Similar for all other actions.

In a game with **perfect-information** the players have *full* access to the world state. They can observe all objects in the world, all other players, all moves, i.e. everything that's important for the game and to make a decision. For the Tourality game this means the AIs have vision of all leaves, all obstacles and all AIs.

The term **zero-sum** describes games, in which the gain of one player is the loss for the other one, and when adding up these *values* of all players it results in *zero*. So every player tries to maximize his value, or equivalent tries to minimize the other players value. *Values* in this context don't have to be the direct points you can achieve in the game. It's the result of the *utility* function, which evaluates the game state in the terminal nodes of the game tree and it's typically from the point of view of the *Max* player.

3.3.2 Game tree

We said that *Minimax* works by searching the *game tree* for an *optimal* strategy. So what is a *game tree*? In a **game tree** the nodes represent a *game state* and the edges represent possible *actions* in that state. The starting configuration of the world is the root and by alternating moves of the players the tree is created. The leaves represent final situations of the game, i.e. situations in which the winner is known, and where *value* of the game is computed depending on the state. In every node *all possible* moves of the player who is next to act are considered and will create child nodes which represent the changed world state in which the chosen action resulted. The algorithm *simulates* **all** variations of the game. Because of that the whole game tree contains *all* possible game situations and, more important, *all* possible strategies, which could be played from the starting configuration. So we *just* need to extract an *optimal* one out of all possibilities. Relating to our game and problem, it simulates all possible variations of actions the two *Kaefer* can do and due to seeing *all* variations of how the game could be played, it can tell you what's the best action you should do, i.e. what's the best position to move to. We will see a game related example in **Section 4.3.1**.

[9] von Neumann and Morgenstern (1944)

The following game tree[10] is for a very simple 2-player game, where every player has three actions to choose out of and the game consists of only one turn for each player. **Max** starts and **Min** reacts to his choice. Due to three actions and one turn for each, the game tree consists of 9 leaves which show the different *values* for different terminal situations of the game.

By looking at this simple example it's very easy to understand that a real game tree, even for very simple real games like Tic-Tac-Toe, becomes very large and will contain a lot of different *strategies*. But after we have understood the game tree in general, we can move on how to search it for an *optimal* strategy.

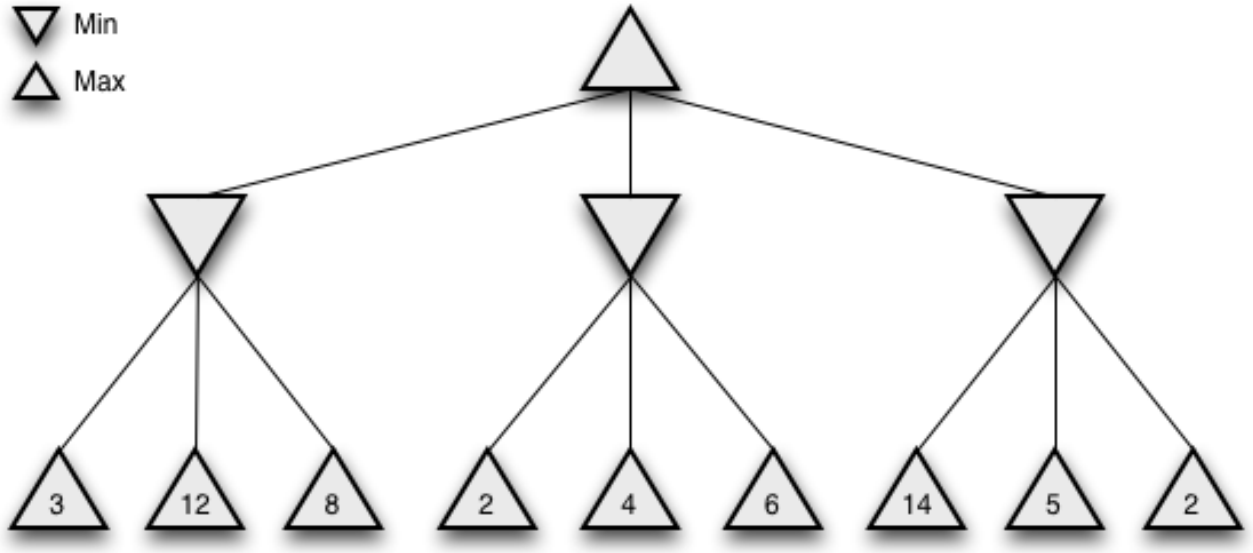

Figure 5: Game tree for an example game

[10] example taken from Russell and Norvig (2011)

3.3.3 Optimal strategy

We talked a lot about *optimal* strategies or playing *optimal*, but what is meant by this?
In a single-player game an *optimal* solution would simply be a series of actions which leads to a *winning* final state and could be found by different search algorithms. But in our two-player game this becomes an *adversarial search problem*[11], because we have to find a strategy that takes the opponents actions into account. So here an optimal solution is a strategy of how to choose your own action considering the response action of the opponent. One important assumption is that the opponent always plays *perfectly*, i.e. he never chooses an action that's *bad* for him. The resulting optimal solution is *always* depending on the used utility function, i.e. it is an optimal solution for a particular utility function. When using another utility function it may, and often will, result in a different strategy, which is *optimal* too, but referred to the new utility function. But how to find this *optimal* strategy for a given utility function?

For a certain game tree we can determine the optimal strategy by the **Minimax**-value of each node. This value represents the gain for *Max* to be in this state, always assumed that both players play optimal. The **Minimax**-value of a leaf node is the result of the utility function of the state represented by this leaf. As defined before *Max* tries to maximize his gain, so he always chooses the action which gives him the highest gain on his nodes. Similar *Min* tries to minimize *Max*'s gain, so he always chooses the action which leads to the highest loss for *Max*. The **Minimax**-value of each node **s** is defined by[12]:

$$Minimax(s) = \begin{cases} Utility(s) & \text{if s is a terminal node} \\ max_{a \in Actions} Minimax(Result(s,a)) & \text{if Player(s) = Max} \\ min_{a \in Actions} Minimax(Result(s,a)) & \text{if Player(s) = Min} \end{cases}$$

Figure 6: Minimax definition

If we apply this algorithm to the game tree of our example game from before we get the following result. The highlighted edges represents the actions taken by *Max* and *Min*. The value inside the nodes are the Minimax-values of them.

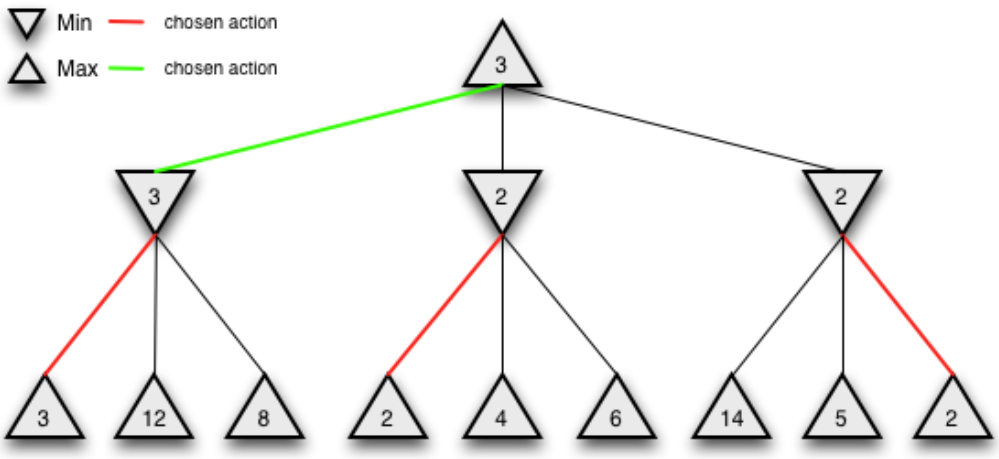

Figure 7: Game tree with Minimax-values

[11] searching in a competitive environment with competing goals of the AIs
[12] definition taken from Russell and Norvig (2011)

Starting at the root of the game tree the Minimax-algorithm recursively moves deeper until it reaches the leaves. The leaves are the recursion anchors and they get their value by the *utility* function. These values are outreached upward and the parent node can do its decision based on if it is a **Max** or **Min** node. So in our example, after the first three leaves have been reached, the first Min node can choose its best action which in this case will be the action leading to a result of 3 and this is getting the value for the node. Similar the other two Min nodes are evaluated and get a value of 2. After all child nodes of the Max node have been considered, the Max node can choose its best action which will lead to a result of 3 for it. So the result of the game if both players play optimal is 3 and the optimal strategy is given by the actions which lead to the terminal state with value 3.

As we already noted before game trees for real games are just too big to be searched in a certain time, what is quite important in most of the games, due to normally you have to perform your action in a certain time. To manage this the game tree is **limited** to a given **depth**. The *terminal test* becomes a *cutoff test*, where nodes at the limited depth are treated as terminal nodes. Also the *utility* function is replaced by a *evaluation* function, because we now don't value terminal states of the game, but game states that occur within the game. These states need to be evaluated with a *heuristic* function which tells how good this state seems to be, i.e. how likely it is to get to a winning terminal state when taking this action. Obviously creating this evaluation function is the most important part to get a strong artificial intelligence. The **evaluation** function often uses a linear combination of several weighted features of the actual state to estimate the value of it. We will see this in **Section 4**.

3.3.4 AlphaBeta-Pruning

We now know how to get an optimal solution with the Minimax-Algorithm, but if we take a closer look at our Game-tree with the Minimax-values, isn't it considering some unnecessary nodes? The second Min node gets his value 2 from the first move, and due to the other two moves have higher values, the Min node doesn't *need* them. And due to the number of nodes grows exponential with the number of moves and the search deep, this is very crucial. So can we improve the search to be faster by ignoring *unnecessary* nodes without affecting our result?

AlphaBeta-Pruning[13] is doing exactly this, it *ignores* the nodes that don't influence the chosen action by using the knowledge the Minimax-Algorithm has gathered so far. When the second Min node in our example is getting evaluated, we already know that the first Min node got a value of 3. So the Max node above will get at least a value of 3 too, because it will never choose an action that will lead to a lower value. To use this knowledge we have to carry those informations along the tree. This is done by two values: **alpha** and **beta** which get updated along the execution.

- α = the value Max can achieve **at least so far**
- β = the value Max can achieve **at most so far**,
  i.e. the value to which Min can **limit** the gain of Max

The range [α, β] is called **search-window** and limits the moves which will be considered. Only if the value is inside of the search-window the move will be used, otherwise it will be pruned. The window is initialized with [-∞, +∞] and while the algorithm runs this window is getting smaller, more actions get pruned and the algorithm terminates faster. The goal is to shrink the window as fast as possible to prune as many moves as possible. By using the search window to prune unnecessary nodes, there are two different kinds of cutoffs:

- α**-Cutoff**: If there is an action with value $v \leq \alpha$ at a **Min** node, we can prune all other alternatives, because **Max** can already achieve a gain of α in a previous subtree, i.e. Max will never choose the action leading to the actual subtree.
- β**-Cutoff**: If there is an action with value $v \geq \beta$ at a **Max** node, we can prune all other alternatives, because **Min** can already limit the gain of Max to β in a previous subtree, i.e. Min will never choose the action leading to the actual subtree.

With this definition we can now reduce the numbers of nodes that have to be evaluated by the Minimax-Algorithm. Because we start with the maximal initial search window [-∞, +∞] and let the window shrink by values that already have been evaluated, we only prune nodes that are *save* to prune, i.e. nodes that have no influence to the Minimax-value of the Game tree. Often it's even possible to not only prune single nodes, but instead to prune whole subtrees, what of course gives a significant speed boost.

As we know, the speed up factor is related with the search window and how fast it shrinks. So it seems to be even better to start with a smaller search window to have more prunes, this is called **Aspiration Search**. But it has a disadvantage: due to the artificial limit at the start, where you don't know a value to relate on, it may occur that you will prune a necessary node or subtree. In this case you have to research the game tree with a larger window.

[13] Newell et al. (1958), Hart and Edwards (1961), Hart et al. (1972) and Knuth (1975)

Now that we know how to build a game tree, how to get an optimal strategy out of it and even how to speed up the search, we can put this in an algorithm[14]:

```
function AlphaBeta-Search(state) returns action a
    v ← Max-Value(state, −∞, +∞)
    return action a ∈ Actions(state) with value v

function Max-Value(state, α, β) returns value v
    if state is terminal return Utility(state)
    v ← −∞
    for each a ∈ Actions(state) do
        v ← Max(v, Min-Value(Result(state, a), α, β))
        if v ≥ β return v
        α ← Max(α, v)

function Min-Value(state, α, β) returns value v
    if state is terminal return Utility(state)
    v ← +∞
    for each a ∈ Actions(state) do
        v ← Min(v, Max-Value(Result(state, a), α, β))
        if v ≤ α return v
        β ← Min(β, v)
```

Figure 8: Pseudocode for Minimax with AlphaBeta-Pruning

Undefined functions in the code above:

- *Actions(state)*: a collection of all possible actions in the given state
- *Utility(state)*: call of the utility function, which evaluates the given state and returns his value
- *Result(state, a)*: returns the resulting state if taking action a in the the given *state*

And as mentioned before, when using a fixed depth, replace the *terminal test* with a *cutoff test*, and the *utility function* with a *evaluation function*.

In Figure 9 we used the algorithm above to improve the Minimax result of our example game tree.

1) The first terminal node has been evaluated, and the search window in the first Max node has been updated with this information.

2)-3) All terminal nodes of the first Max node have been evaluated, and the information got back to the root.

4) The first action of the second Min node has been evaluated, which resulted in a value $\leq \alpha$, so the other terminal nodes aren't considered at all as we already discovered at the beginning, but now we understand how it works.

5)-6) The last Min node has been evaluated and we got as expected the same result as with the Minimax-Algorithm, but with less nodes evaluated.

[14] adapted pseudocode algorithm based on Russell and Norvig (2011)

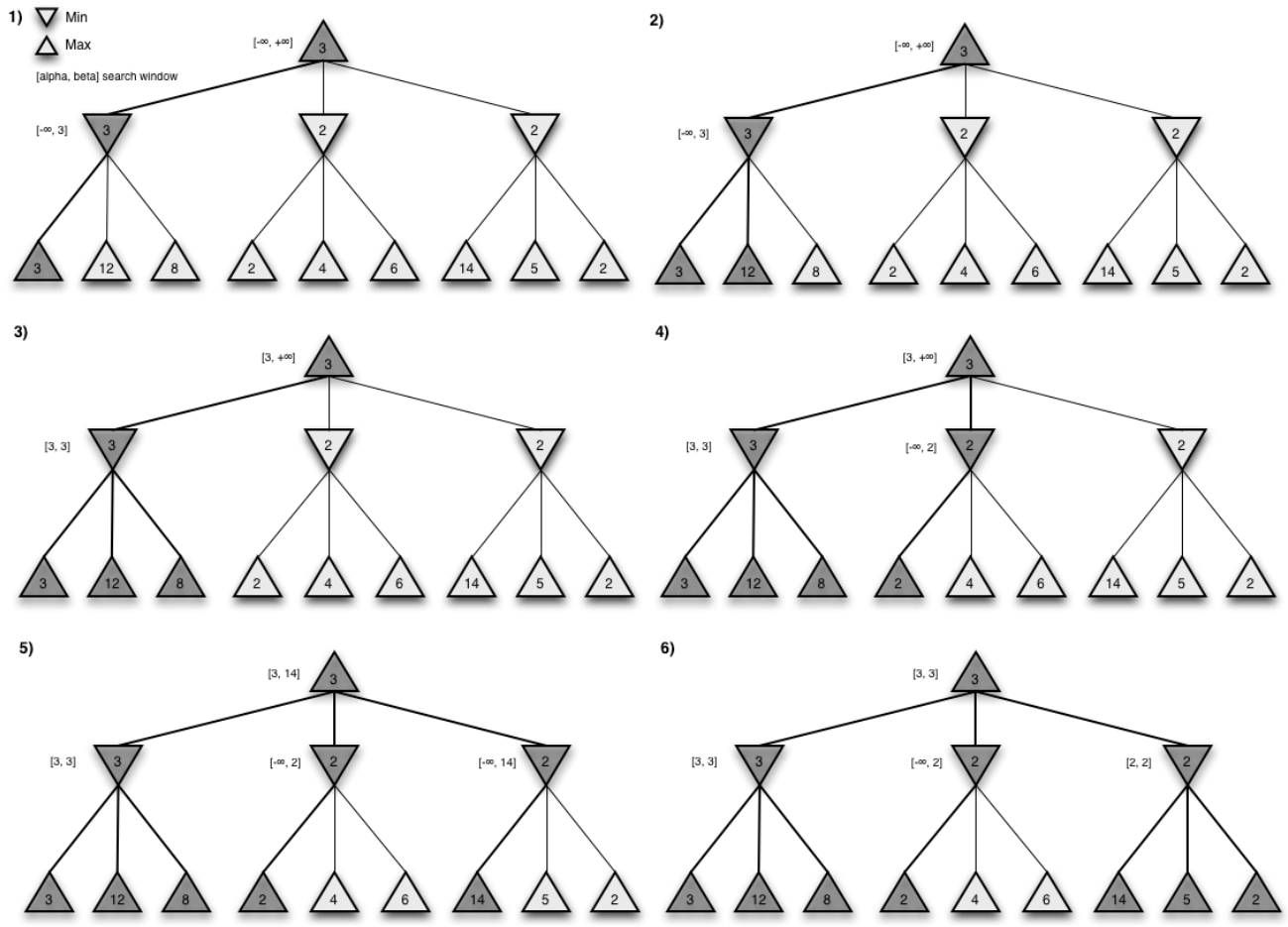

Figure 9: Game tree with AlphaBeta-Pruning

If we take a closer look at the example above, we see that the second and the third Min node have the same value. But at the third one all terminal nodes haven been evaluated unlike at the second Min node, where only the first terminal node got evaluated. The difference between these two cases is the *order* in which the terminal nodes, i.e. the actions, are considered. Obviously the third Min node wouldn't have to evaluate all terminal nodes, if it would evaluate the terminal node with value 2 at first too. The **move ordering** is very crucial for the performance of the AlphaBeta-Pruning. If *good* actions are evaluated first, it may lead to more and more valuable cutoffs. There are different ways to presort the actions for a node, but of course this will never be perfect. If we had a perfect move order, we would not have to use Minimax and AlphaBeta-Pruning to get an optimal strategy anymore, we already would have it. So having a good *guess* which actions are *good* really helps and improves the algorithm, but it's limited.

4 Specific Implementation

In this section we will combine and adapt the algorithms we've looked at in **Section 3** with the Tourality-Game we defined in **Section 2.2**, **Section 2.2.2** and **Section 2.2.3** in more detail.

4.1 The idea behind the AIs

After we now know all the several components, we have to combine them. We described the main problem in **Section 3.1**, the algorithms and the game as mentioned above. So how can the AI now use this to play the game?

The Breadth First Search algorithm is used to find a shortest way to a leaf, or alternatively different ways to one or more leaves. It returns a path, i.e. a sequence of actions, to reach the leaf which the AI uses to walk as fast as possible to a leaf going round the obstacles. By this the AI gathers all the leaves one after another.

The Minimax algorithm is used to get an optimal strategy, a sequence of actions, the AI can execute. It takes much more information into account than the Breadth First Search. By considering the opponents actions and *looking into all possible futures*, it executes the action which seems to be best. It simulates *possible futures* of the game and chooses the action that seems to lead to the best *future* for the AI.

We will take a closer look on the implementations of both algorithms in the next sections.

4.2 Breadth First Search

After we discussed the general functionality of the Breadth First Search in **Section 3.2**, we now want to use it for the Tourality-Game. Due to the game is based on a discrete world, consisting of fields and actions to change the position in this world, we can easily see it as a graph and thus use the Breadth First Search.

The starting node for the search is just the position of our *Kaefer* in the actual state of the world. The nodes get expanded by the *actions* the *Kaefer* can do, i.e. moving one field up, down, right or left. By using these actions on the nodes our search tree is created. The expanding function takes care of the obstacles and the borders of the world; the *Kaefer* can't enter a field on which an obstacle is placed and can't overstep the world borders. The enemy *Kaefer* is just treated as an obstacle. The implementation can deal with different searchgoals, i.e. it can be used with the whole actual world state with all leaves or with just one specific leaf. This is done by setting the *bfs_sicht* parameter which sets the search range, e.g. if $bfs\_sicht = 3$ all leaves within the range of $[x-3, x+3]$ and $[y-3, y+3]$ are considered. If there's no leaf inside this area, the parameter is automatically set to $bfs\_sicht = -1$ which means *unlimited* search deep, i.e. searching until all nearest leaves are found. The result of the algorithm is a list of different ways, i.e. a list of lists of *actions* to get to the leaves.

4.2.1 Finding shortest ways to all nearest leaves

This is the default case with $bfs\_sicht = -1$. The whole actual world state is passed to the Breadth First Search algorithm. It uses the actual position of the own *Kaefer* as the startnode and has no predefined searchgoal. The algorithm starts the search and as soon as it finds the first leaf, it saves the distance to it as the *horizont*. Now it searches for further ways to this leaf and other leaves with the same distance. The search terminates if a leaf with $distance > horizont$ is found.

Figure 10 shows the result of this Breadth First Search with $bfs\_sicht = -1$. There are two leaves with the shortest distance of 3 actions and for each of it two different ways were found. By looking at this example, you may wonder that there are even more ways to these leaves with distance 3. So why did the Breadth First Search algorithm not find them too?

Let's look at one of the not found ways to the right leaf of these two leaves, the way **right - down - right** is obviously a shortest way with distance 3 too. But this way isn't found by the algorithm because the node **below** the node after taking action **right** to find this way has already been *explored* when finding the 2nd and 3rd way (2nd and 3rd picture) in **Figure 10**. An *explored* node isn't used again to search further, so the 2nd action of the not-found path, **down**, isn't considered at all at this point. If the order in which the actions are searched is changed, this way could be found, but then other ways won't be. To really find *all* possible shortest ways, the algorithm has to be changed to reuse already *explored* nodes.

For our purpose the introduced version of the algorithm is quite perfect, because it returns all *really different* shortest ways.

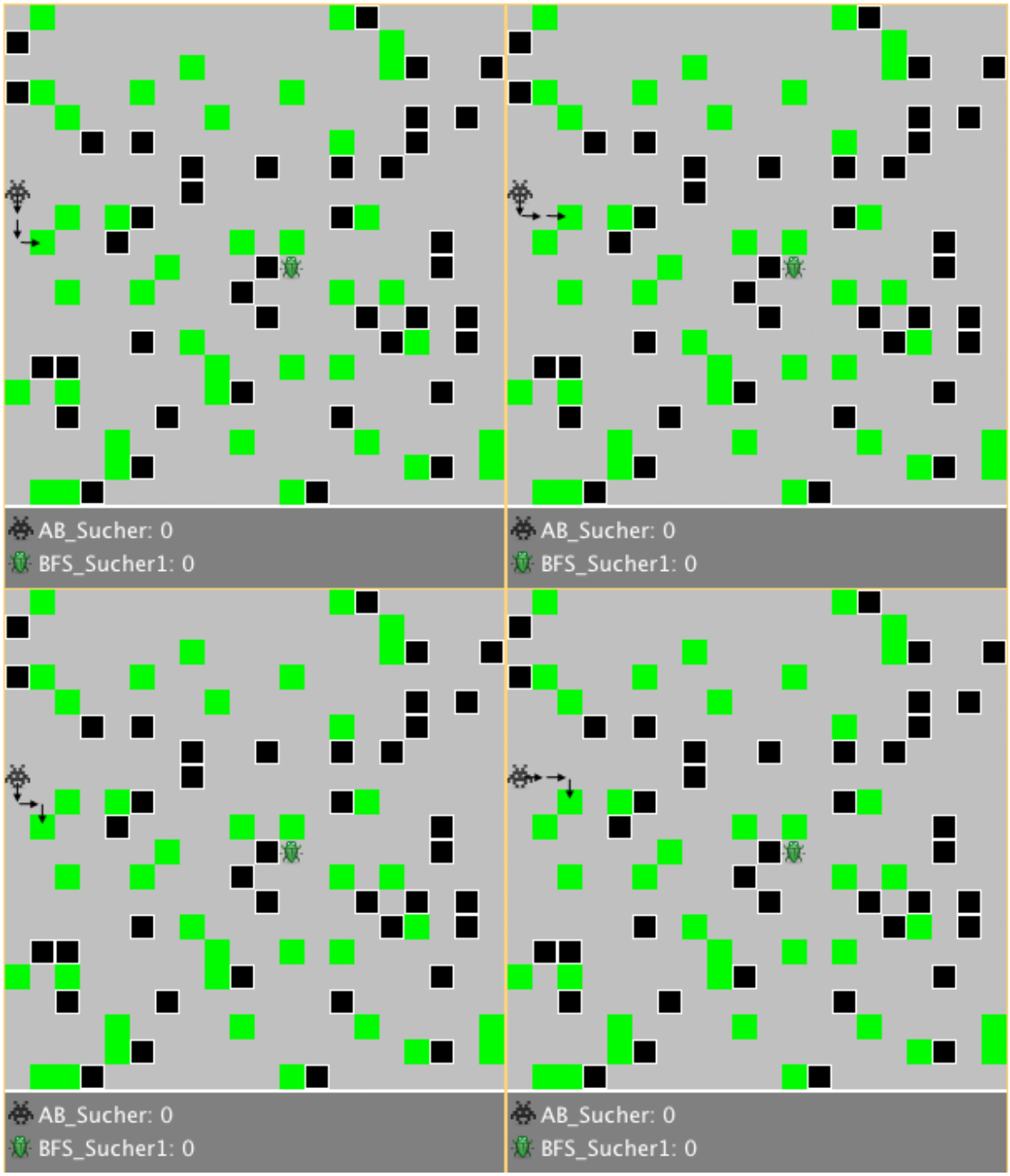

Figure 10: Different shortest ways to all nearest leaves

4.2.2 Finding shortest ways to all leaves in a limited range

The alternative version of this implementation of the Breadth First Search algorithm uses an adjustable search range. This is done by setting *bfs_sicht* > 0. When using it this way, the search isn't started with the whole actual world state, instead for each leaf inside the range a new search is started *with only this particular leaf* in the world. So at first each leaf is tested if it's inside the range and if that's true, a new search with this leaf as searchgoal is started. The single results of each search are combined in one overall result, containing all found ways. As already explained in **Section 4.2.1**, not *all* shortest ways are found here too. **Figure 11** shows an example with *bfs_sicht* = 3, so the highlighted 7x7 - effectively 7x5 since the area intersects the border of the world - area is searched and 5 ways for 3 leaves are found.

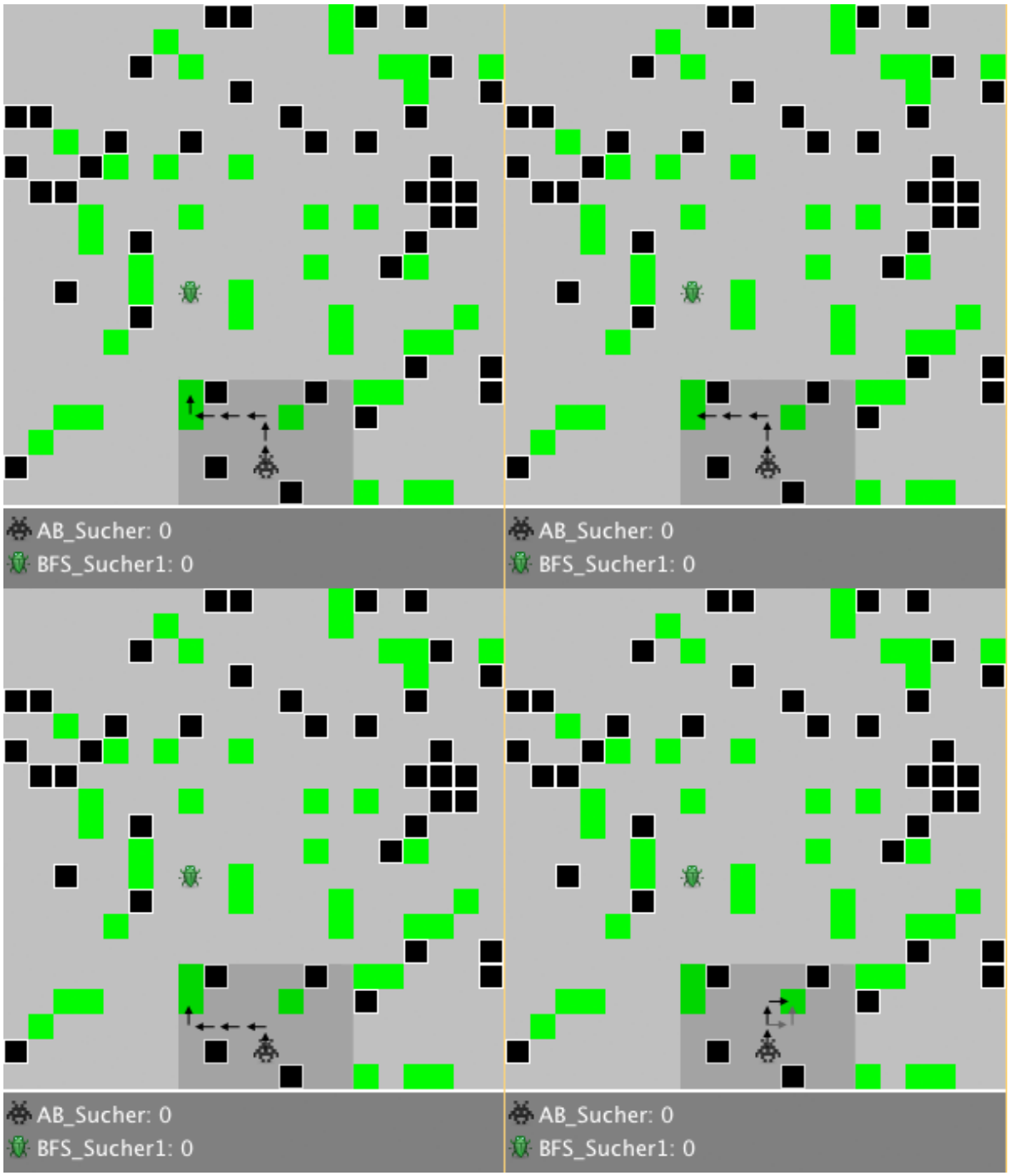

Figure 11: Different shortest ways to all leaves in a limited range

4.3 Minimax

Next we'll take a look on some details of the Minimax implementation and how the ways found by the Breadth First Search algorithm are used as an input to create the search tree and, together with additional information, to presort the possible actions for the move ordering for the AlphaBeta-Pruning.

4.3.1 Creating the tree

As we discussed in the Algorithm **Section 3** we can't search the whole game tree. Thus as the first crucial parameter of the Minimax implementation we have the search depth limit as *maxTiefe*. The search window $[\alpha, \beta]$ is initialised with $[-10000000, 10000000]$.

The option *weg_size* defines how long the resulted way found by the Minimax algorithm is, i.e. if $weg\_size = 1$ it just delivers the *next best* action, but e.g. if $weg\_size = 3$ you get the *next best **3*** actions. With a *longer* result, i.e. more resulting actions, the overall search-time is strongly reduced of course, because you don't need to search again every time it's your turn. Otherwise false assumptions of the enemy movement may lead to *bad* own moves and without researching they may sum up. We will see the different behaviour later in the evaluation.

The last two main parameters are *zug_faktor* and *zug_kosten*. They define the reward you'll get when gathering a leaf according how far away it was, e.g. with $zug\_faktor = 10$ and $zug\_kosten = 2$ a leaf gathered with 2 actions gives a reward of 8, but a leaf gathered with 4 actions only gives a reward of 4. After each action the *zug_faktor* is reduced by the *zug_kosten*. This is done to give near leaves a higher reward than further leaves and thus a higher priority.

Due to the fact *zug_faktor* is reduced by the *zug_kosten* after each action, these parameters depend on the search depth *maxTiefe*. The *maxTiefe* is always an odd number, because we want to start and end with an own action. That gives the constraint for the *zug_faktor* and the *zug_kosten*:

$$zug\_faktor \geq zug\_kosten \cdot ((maxTiefe+1)/2)$$
$$zug\_kosten = k \text{ with } k \in \mathbb{N}^+$$

So if we have a search depth $maxTiefe = 11$ which means 6 own actions to look ahead and we want near leaves to have a *strongly* higher reward than further ones, we choose $zug\_kosten = 3$ and get $zug\_faktor \geq 18$. The *zug_faktor* can be additionally increased by adding n * *zug_kosten*, $n \in \mathbb{N}$.

The search tree now is created with the actual world state as the root node and Max, which is *our player* in the algorithm, starts to search his actions. Each node in the search tree has a value named *wert* which saves the information we gather on the way down and it's initialised for the root node with 0 or the leaves gathered in the game so far. Max now expands the tree with all of his possible actions in this state which creates a new node for each action. The *wert* of these new nodes are always initialised with the *wert* of the parent node. If a leaf is gathered with an action the *wert* of the related node is increased by the current *zug_faktor* and the leaf is removed from the world for this particular path in the tree.

The expanding function has a parameter $schritt\_kosten = n$, $n \in \mathbb{N}$ which is subtracted from the *wert* of a node to put costs on each action. This can be used to amplify prioritizing short ways. With $schritt\_kosten = 0$ this is turned off.

We always talked about the information input of the Breadth First Search, but so far it wasn't used. So how is it used? For each possible action all of the ways found by the Breadth First Search are checked up if this action is part of the way *at this depth*, i.e. if it's the 3rd action in the Minimax algorithm of Max, the 3rd action of all Breadth First Search ways are compared to it. And for each match the parameter *bfs_dist* is added to the *wert* of the related node with $bfs\_dist \in \mathbb{N}$. By this actions, respectively ways, that follow the ways found by the Breadth First Search have a higher *wert* and thus get prioritized.

We can now easily do a move ordering by just sorting the nodes according to their *wert*, so we prefer, i.e. searching deeper first, paths which seem to be *good* so far. As we found out in **Section 3.3.4** this can speed up the algorithm.

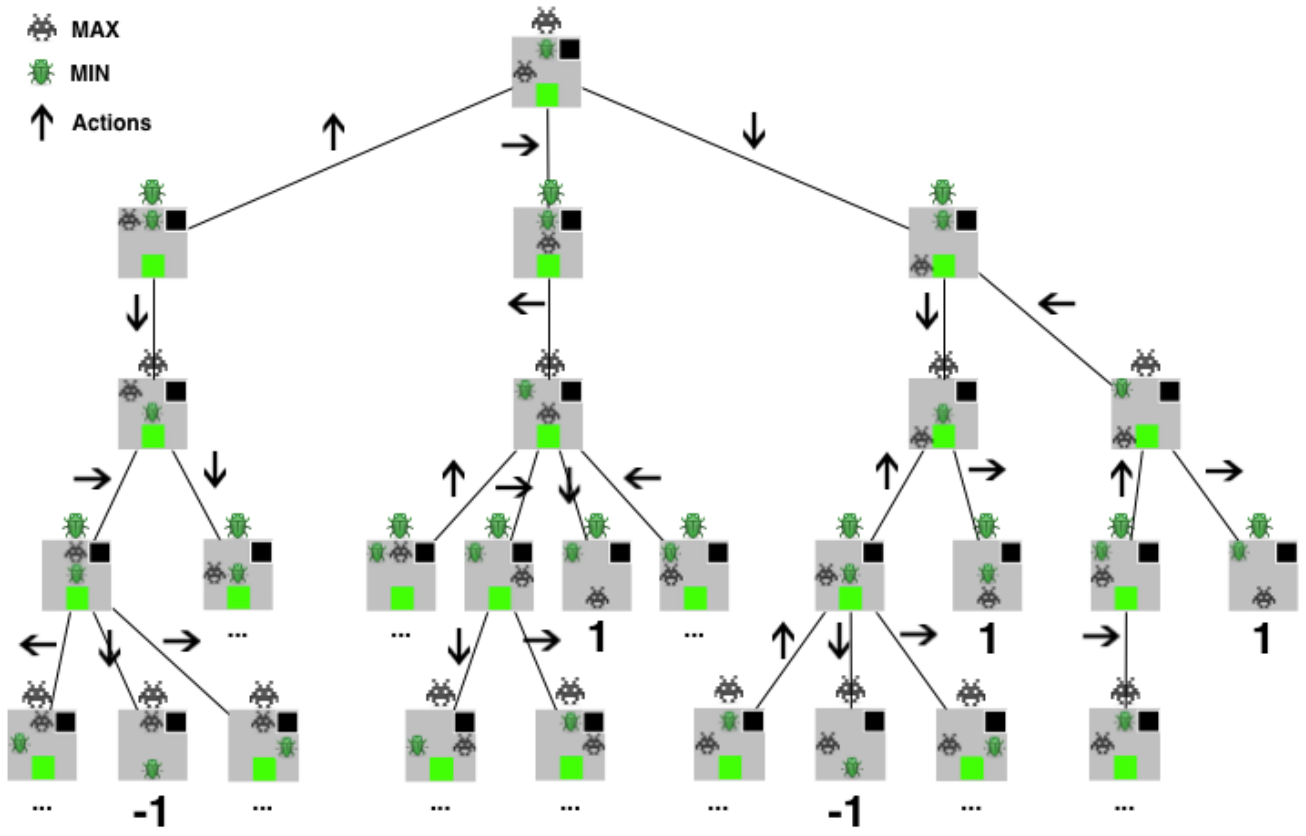

Figure 12: Minimax game tree example on Tourality

Figure 12 above shows a part of the game tree of a smaller version of Tourality to demonstrate the Minimax algorithm on Tourality. For this purpose the world in this example has only a size of 3x3 fields with one obstacle and one leaf in it.

As we know the nodes in the game tree represent game states and the edges represent the possible actions in this state. In the example above you can easily see this and how each action leads to a new node with a changed game state. The arrows represent the action done on this edge, i.e. ↑ stands for the action *hochLaufen*, which moves the actual player one field up. The icon above the nodes illustrates what a kind of node it is, i.e. if its a Max or a Min node.

So using the start configuration of the world as the root of the game tree and alternating between actions of Max and Min the game tree is created as we already know. Each **applicable** action, as you easily see in the example not all actions are available in each node, leads to a new node. Here Max has three actions to choose from at the root node and each action results in a different new node in which now Min applies its actions. As mentioned before the game tree is not complete, the nodes with **...** below aren't expanded anymore in this example to keep the example clear. The evaluation function in this case just computes a value based on the gathered leaves and due to there is only one leaf in the world the value of the evaluated node is 1 if Max gathered the leaf or -1 if Min gathered the leaf. By looking at a node with a value of 1 and going back upwards in the tree, Max can obtain a optimal strategy to end up in a winning state.

Although this is only a very small and simple example, we now understand the nodes and edges in the game tree and how to adapt the Minimax algorithm on Tourality. We see how each action changes the state and creates a new node, what was meant by saying simulating all possible futures and how the optimal strategy, the sequence of actions, is extracted from it.

4.3.2 Evaluation function

So far we are able to use the output of the Breadth First Search as an input to the Minimax algorithm, building the search tree and do a move ordering for the AlphaBeta-Pruning. But the most important part is still missing: the **evaluation function**, which defines the values of the nodes at the limited search depth and by this creating the foundation on which the best action is chosen.

Due to the states which are evaluated by the evaluation function aren't terminal states of the game, the evaluation function is a heuristic function which tries to predict how likely it is to win from this state, i.e. how good this state is. The evaluation function used in this work is a weighted linear combination of several features of the game state. With n=*number of features*, f = *a feature* and w = *the weight* of it, the general form is:[15]

$$Eval(state) = w_0 \cdot f_0 + w_1 \cdot f_1 + ... + w_n \cdot f_n$$
$$Eval(state) = \sum_{i=0}^{n} w_i \cdot f_i$$

Finding and combining good features is very crucial for the result of the algorithm. Due to the memory and time restrictions the evaluation function has to be as *simple* as possible because it is used on all nodes at the limited depth. For the linear combination the following features were used:

- **meinePunkte:** the amount of leaves MAX gathered
- **gegnerPunkte:** the amount of leaves the enemy, MIN, gathered
- **meinUmfeld:** the amount of leaves around MAX in a certain range, weighted by their distance, *ignoring* obstacles
- **gegnerUmfeld:** the amount of leaves around MIN in a certain range, weighted by their distance, *ignoring* obstacles
- **naehstesBlatt:** the distance from MAX to the nearest leaf, *ignoring* obstacles
- **gegner:** the distance from MAX to MIN, , *ignoring* obstacles
- **wert:** the *wert* of the node so far as explained in **Section 4.3.1**
- **wert2:** the *wert* of the node so far as explained in **Section 4.3.1**, but calculated for MIN

By changing the weights of the features or dropping some and including others, and the general parameters described in **Section 4.3.1** different AIs can be created.

[15] based on Russell and Norvig (2011)

5 AI Evaluation - Competitive Tournaments

To evaluate the strength and behaviour of the Minimax based AI, it is tested against different reference AIs based on different other algorithms. The Minimax based AI plays a tournament consisting of many challenges against each of them. The world in each challenge is completely random, i.e. the positions of all leaves, obstacles and AIs are randomly distributed. To get a proper mean 500 challenges are played in each tournament.

To evaluate the strength of the AIs, the Wins, Draws and lost challenges for each AI are counted. From this information the following different values are calculated:

- **Points:** sum of all challenges with 3 points for a win and 1 point for a draw
- **Won:** percentage on won challenges
- **Not-Lost:** percentage on not-lost challenges, i.e. won or draw
- **WD/L-Ratio:** ratio between not-lost challenges (W = wins, D = draws) and lost challenges (L = lost), calculated as: $(W + D)/L$
  This is directly related to *Not-Lost*, but it gives a factor instead of a percentage. The factor represents how much more challenges were not-lost compared to lost ones, i.e. a factor of 2.3 means: 2.3 times more challenges were not-lost than lost.
- **W/L-Ratio:** ratio between won challenges (W = wins) and lost challenges (L = lost), calculated as: W/L
  As the *WD/L-Ratio* this is directly related to the *Won* percentage, but gives a factor again representing how much more challenges were won compared to lost.

The *WD/L-Ratio* and *W/L-Ratio* are based on the KDA-Ratio[16] used for the DotA2 game. We now have several criteria to evaluate and compare the different AIs.

5.1 Evaluation AIs

As described before we want to evaluate the behaviour and strength of the Minimax based AI in comparison to some other algorithms. For this purpose three algorithms from different fields were chosen as the Reference AIs and will be explained shortly next.

5.1.1 Reference AI 1 - Uninformed Search

The first Reference AI is just a very simple *seeker* and based on the simplest version of the *Breadth First Search* algorithm explained previously. It searches until it found the first leaf and the first shortest way to it and then simply follows that way towards it. So it doesn't care about other nearest leaves, other ways, other further leaves, any other information and the only way the other AI is considered is that it is treated as an obstacle. It is an uninformed search that uses no further knowledge about the game and world and is basically the version explained in **Section 4.2.1**, it just terminates earlier.

5.1.2 Reference AI 2 - Informed Search

As the second Reference AI an improved *seeker* was chosen which is based on the *Best First Search* algorithm. This algorithm principally works the same as the Breadth First Search, but it uses further knowledge about the world. Instead of expanding all nodes with depth d completely before expanding a node with depth $d+1$ like the Breadth First Search, it expands the node which *seems* to be best, no matter which depth it has. For this it uses a heuristic function, similar to the evaluation function of the

[16] Kill-Death-Assist Ratio used on http://dotabuff.com/

Minimax algorithm, that calculates a value for each node describing how *good* it is. The value for each node used here is just the direct heuristic distance to the nearest leaf, thus the Best First Search always expands the node with the smallest valued distance to a leaf next and uses the knowledge of the world for this. The algorithm terminates when the first way to a leaf was found and simply follows that way.

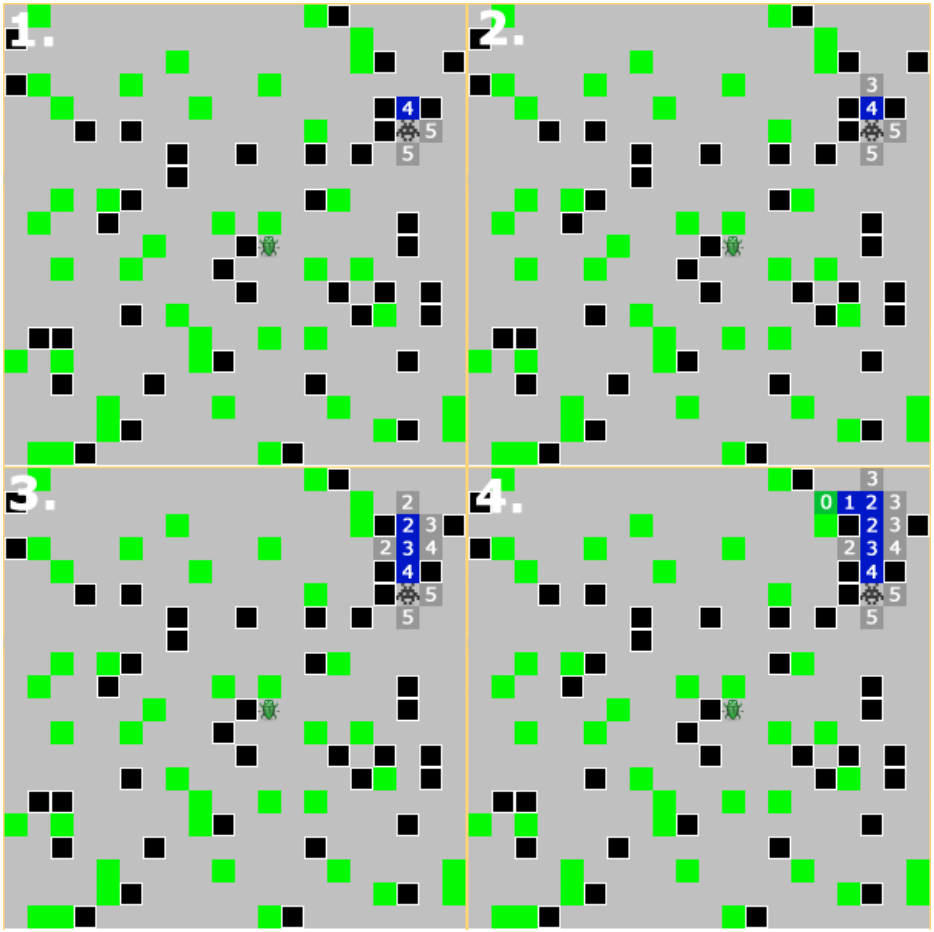

Figure 13: Best First Search AI example

As said before the Best First Search works very similar to the Breadth First Search and it builds a search tree in the same way, i.e. the nodes represent the position of the *Kaefer* and the edges the possible actions. **Figure 13** illustrates how the Best First Search algorithm finds a way to a leaf in Tourality. In the first picture the starting position is expanded, due to the obstacle left to it there are three possible actions. The dark gray fields are *seen* and in the searchqueue, just like at the Breadth First Search, and the blue field represents the node that is chosen to be expanded next. For each node the heuristic value is calculated which are the numbers on the fields in the picture and the algorithm now chooses the node which seems to be best so far, in our case the node with the lowest estimated distance to a leaf which is the node above the *Kaefer* with value 4. This node is expanded in the second picture and leads to a new node with value 3 which is added to the searchqueue and will be expanded next. In contrast to the Breadth First Search the new node with value 3 gets expanded next although it is *deeper*, because the Best First Search always chooses the *best* node no matter how deep it is. Repeating this until a position with a leaf on it is found, the algorithm finds a way to this leaf, represented by the blue fields. If you take a closer look at the last picture of the example, you will see that the found way isn't a shortest way. At the blue field with value 3 there are two child nodes with value 2 and by following the left one you will reach another leaf with a shorter way. This way isn't found by this Best First Search because it considers the action *hochLaufen* before the action *linksLaufen* when expanding a node and just terminates if the first way was found. Thus the algorithm doesn't always find a shortest way.

5.1.3 Reference AI 3 - Heuristic Method

Both of the first two Reference AIs use a search algorithm to find their best action. The third Reference AI was chosen to work completely different, without a search at all. Like the Best First Search based AI it uses the knowledge about the world to determine the best action to take just without a search. In each turn the AIs have up to 4 possible actions they can take, depending on obstacles or the border of the world around them. The Heuristic AI uses an evaluation function to calculate a value for each possible action in the current state and takes the action which seems to be best. Comparing to the two other Reference AIs it only looks one action ahead. The evaluation function doesn't use a search algorithm, it really only *looks* on the current state to determine the value. The used evaluation function is very simple and the calculated value for a node only depends on the heuristic distance to the nearest leaf and the number of leaves in an 3x3-area around the actual node. Additionally if the AI moves up the action for moving down gets a worse value in the next turn, and vice versa, to prevent getting stuck jumping between two positions. The same is done for left and right of course. A neighbouring field with a leaf on it directly gets a very good value.

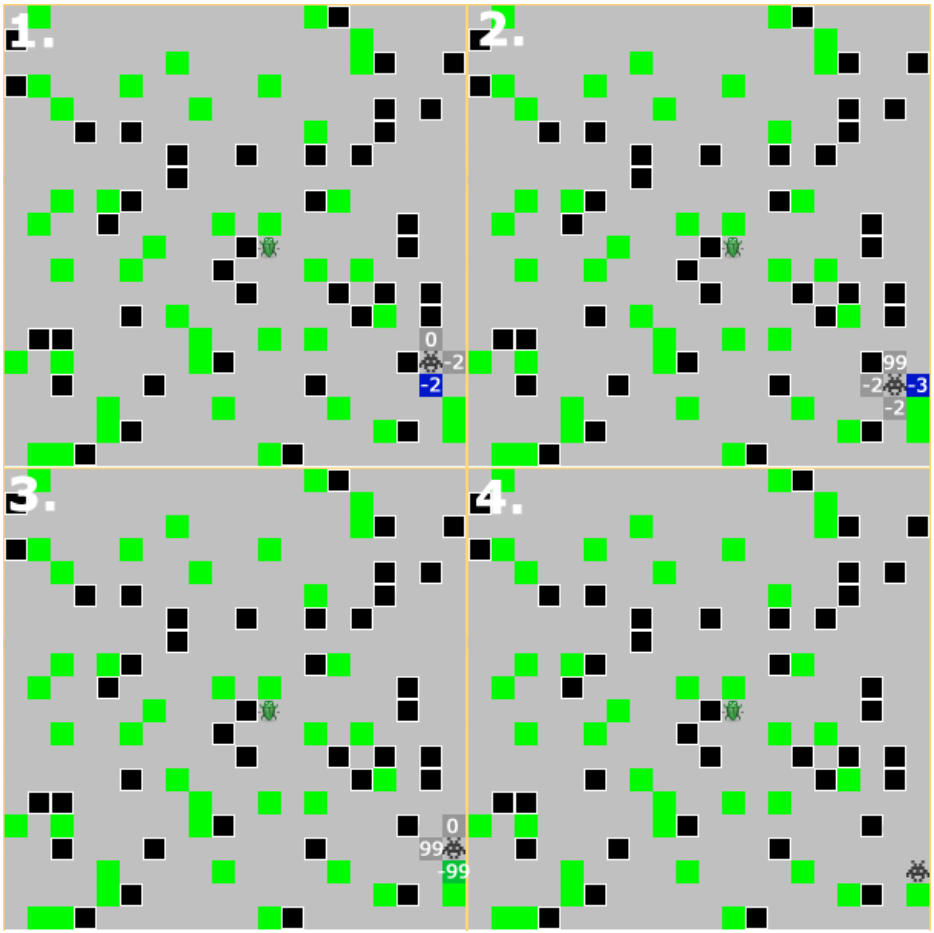

Figure 14: Heuristic AI example

In **Figure 14** the behaviour of this AI is illustrated. It calculates a value for each possible action at the current position and simply takes the *best* action, i.e. the one with the smallest value. If there are actions with the same value it chooses in the order up - down - right - left. The pictures show how this AI plays without using a search at all. It calculates a value for each action, without a search too, and immediately executes the *best* action.

5.1.4 Evaluation AI - Minimax

This is the AI that will represent the Minimax based AIs in the evaluation tournaments against the other algorithms. The Minimax based AI used for this has the following values for the parameters described in **Section 4.3**:

- **wert:** initialized with 0
- **maxTiefe:** 11 → looking 6 own and 5 opponents actions ahead
- **zug_faktor:** 18
- **zug_kosten:** 3
- **weg_size:** 1 → result is only the next best action
- **bfs_sicht:** -1 → uses the normal Breadth First Search, see **Section 4.2.1**
- **bfs_dist:** 2
- **schritt_kosten:** 0
- **Evaluation function:**
  $Eval(state) = 6 * wert + meinePunkte + meinUmfeld(range = 3) - 4 * naehstesBlatt$
  → see **Section 4.3.2**

The parameters generate an AI that discounts future rewards more compared to early rewards, has no costs for an action and prefers the ways found by the Breadth First Search before. The evaluation function calculates a value that is roughly just trying to maximize the gathered leaves on that path of the game tree and to get in a *good* position for the next action depending on the surrounding leaves.

5.2 Evaluation against reference AIs

Now we start to evaluate the behaviour and strength of the created Minimax based AI compared to the Reference AIs as described before. Thus we will play three evaluation tournaments consisting of 500 challenges each. The first evaluation tournament is played between the Minimax based AI and the Breadth First Search AI.

The played 500 challenges resulted in the following statistics:

AI	Wins	Draws	Lost	WD/L-Ratio	W/L-Ratio	Won	Not-Lost	Points
Minimax	258	90	152	2.29	1.70	51.60%	69.60%	864
BFS	152	90	258	0.94	0.59	30.40%	48.40%	546

Table 1: Evaluation Tournament against Reference AI 1

Regarding the results of the tournament we recognize that the Minimax based AI performs better than the Breadth First Search AI and has a *WD/L-Ratio* of 2.29, i.e. for each lost game the Minimax based AI plays in 2.29 games at least a draw. But the *WD/L-Ratio* of 0.94 of the Breadth First Search AI, which has *no* intelligence and is just an uninformed search, also shows that it is able to play at least a draw in almost half of the challenges. Having more information to use for the decision makes it harder to tune every part of it, but already that not perfectly optimized Minimax based AI shows a significantly better performance than an uninformed search based AI. Using the knowledge about the world, the state and the game itself the Minimax based AI can search and play goal-oriented and thus more efficiently.

In the next evaluation tournament the Minimax based AI plays against the Best First Search AI, i.e. against an informed search just like itself. Intuitively we would suppose that the Best First Search, the informed search, will perform better than the Breadth First Search, the uninformed search, due to the knowledge it uses. The evaluation tournament led to the following result:

AI	Wins	Draws	Lost	WD/L-Ratio	W/L-Ratio	Won	Not-Lost	Points
Minimax	332	59	109	3.59	3.05	66.40%	78.20%	1055
BestFS	109	59	332	0.51	0.33	21.80%	33.60%	386

Table 2: Evaluation Tournament against Reference AI 2

The statistics may be quite surprisingly, because they show that the Best First Search performed worse than the Breadth First Search. The *WD/L-Ratio* of the Minimax based AI increased to 3.59 and the *WD/L-Ratio* of the Best First Search is only 0.51, and the same holds for the *W/L-Ratio*. Overall the Best First Search AI wins less challenges and plays less draws, it just loses more. This isn't the behaviour we expected. Using the knowledge of the world and state, like the Minimax based AI, should increase the performance of the informed search AI, not decreasing it. To understand this we have to think back to the description of the Best First Search AI in **Section 5.1.2**. The heuristic the AI uses for the search, i.e. the information added to the search compared to the uninformed Breadth First Search AI, is only very small and limited. As said it just takes the heuristic distance to the nearest leaf to value the nodes for the search. Thus it sometimes tries to find a path to a leaf that seems to be near, nearer than all others, but in reality the path to get there may be way longer due to the obstacles which were ignored in the heuristic distance value of the nodes and it just terminates after it found the first way. As already noted in the description of this AI, the algorithm doesn't always find a shortest way, thus the AI makes a detour and this leads to the initially unexpected worse behaviour. The Minimax based AI uses much more information and in a way more complex way and significantly outperforms the simple Best First Search AI.

In the next evaluation tournament the Minimax based AI is tested against the third Reference AI, which is based on a completely different approach. Thinking back to **Section 5.1.3**, the third Reference AI uses an heuristic method without any search algorithm. Using no search and just trying to predict the best next action by *looking* at the current state shouldn't be able to perform well against the Minimax based AI. But the evaluation tournament against the Best First Search AI already surprised us with an unexpected result, so maybe the third evaluation tournament will do it again. Here is the result:

AI	Wins	Draws	Lost	WD/L-Ratio	W/L-Ratio	Won	Not-Lost	Points
Minimax	394	45	61	7.20	6.46	78.80%	87.80%	1227
Heuristic	61	45	394	0.27	0.15	12.20%	21.20%	228

Table 3: Evaluation Tournament against Reference AI 3

No surprise again, but a very clear result. The heuristic method AI has no chance against the Minimax based AI, which results in *WD/L-Ratios* of 7.20 compared to 0.27 and *W/L-Ratios* of 6.46 compared to 0.15. This is the most clear defeat in the three evaluation tournaments and kind of the result we expected. The simple heuristic evaluation function without a search can't keep up with the way more complex Minimax based AI.

We tested the Minimax algorithm against three different other approaches and based on these three evaluation tournaments, we found out that the Minimax based AI performs best. It defeated all other algorithms, some more clearly than others. The used Minimax based AI was kind of a simpler and only handcrafted version and already showed that good performance. An optimized version may even increase the performance. But to find the optimized version we need to know which are the crucial parameters and parts of the Minimax based AI and how to tune them. Thus we need to evaluate these parameters.

5.3 Evaluation of parameters

In the previous **Section 5.2** we evaluated the Minimax AI according to its performance in the tournaments it played against the different Reference AIs. We saw how the Minimax algorithm behaves compared to several other algorithms.

Now we want to analyse the behaviour of the Minimax based AI in more detail, i.e. analyse several of its parameters individually. For this we choose a Minimax based AI and turn on/off or change the values of parameters and let the original version playing against the modified. These small tournaments give information about the effect of the changed parameters, i.e. how crucial they are for the strength and behaviour of the AI.

5.3.1 Minimax search depth

The first parameter we'll a take closer look on is the main parameter of the Minimax algorithm: the search depth. As earlier mentioned in **Section 3.3** the game trees are way too big to be searched completely and thus the search has to be limited to a certain depth.

Due to this will terminate the search before the game reaches an end-situation, this parameter seems to be very crucial. The smaller the search depth is the less information is available to be used for finding the optimal strategy. The algorithm can't see what will happen further in the game and may evaluate an action too bad although it may be the best choice, or the other way round, but this will only be seen a few steps deeper. This is called the **horizon-effect**.

To analyse the influence of the search depth we create three different AIs, which are equal in all parameters and weights *except* in the search depth. The parameter for the search depth is called *maxTiefe*

in the implementation. We choose the three depths 9, 11 and 13 for the evaluation tournament, because 13 is the highest possible depth due to the time restrictions of the Tourality implementation for the BWINF. Only odd depths are used because we want that the last action is done by MAX, i.e. that the last action is an own action.

In the tournament each AI is playing 250 times against each other, so at the end all have played 500 challenges in the whole tournament.

This led to the following results:

AI	Wins	Draws	Lost	WD/L-Ratio	W/L-Ratio	Won	Not-Lost	Points
$maxTiefe = 9$	131	206	163	2.07	0.80	26.20%	67.40%	599
$maxTiefe = 11$	132	232	136	2.68	0.97	26.40%	72.80%	628
$maxTiefe = 13$	166	204	130	2.85	1.28	33.20%	74.00%	702

Table 4: Parameter-Evaluation Tournament: search depth

The result proves what we already intuitionally thought: the deeper the algorithm searches the stronger the AI becomes. No matter which criteria we look at, the deeper version is always better throughout the whole statistics. Thus the search depth is a very crucial parameter of the Minimax algorithm and a higher depth clearly improves the behaviour of the AI, but at the expense of the calculating time and needed memory and thus is limited too.

5.3.2 Rewards & discount factor

The next parameters that we want to take a closer look at are the rewards and discount factors. As described in **Section 4.3.1** the parameters which control the rewards and discount factors are *zug_kosten*, *zug_faktor*, *schrittkosten* and *bfs_dist*. We also already know that *zug_kosten* and *zug_faktor* are related and belong together, thus we will treat them together. The same holds for *schrittkosten* and *bfs_dist*.

The first pair of parameters that we will evaluate are *schrittkosten* and *bfs_dist*. As a short recap, the *schrittkosten* parameter is a discount factor, that is subtracted from the *wert* of the node in the game tree whenever an action is performed, i.e. each action you take has *schrittkosten* costs. The *bfs_dist* parameter is a reward, that is added to the *wert* of the node in the game tree, if the related action is part of a path found by the Breadth First Search algorithm, i.e. it's a reward you get when following the Breadth First Search paths. The settings of the two AIs for the first evaluation tournament are:

AI 1: $schrittkosten = 0$ and $bfs\_dist = 1$

AI 2: $schrittkosten = 1$ and $bfs\_dist = 3$

AI	Wins	Draws	Lost	WD/L-Ratio	W/L-Ratio	Won	Not-Lost	Points
AI 1	155	267	78	5.41	1.99	31.00%	84.40%	732
AI 2	78	267	155	2.23	0.50	15.60%	69.00%	501

Table 5: Parameter-Evaluation Tournament: Rewards & discountfactor 1

The statistics lead to a clear result: AI 1 performs significantly better than AI 2. The higher reward for following the Breadth First Search paths and the costs for each action are decreasing the strength of the Minimax based AI. Having no costs for an action and a lower reward for the Breadth First Search paths increase the performance of the Minimax algorithm, due to it plays more *independent* from the Breadth

First Search algorithm and only uses it as **one** information input.

Now we test the other parameter pair, the *zug_kosten* and the *zug_faktor*. Again as a short recap, the *zug_faktor* is a reward that is added to the *wert* of the node when picking up a leaf in the game tree. The *zug_kosten* is the discount factor which is subtracted from the *zug_faktor* for each action in the game tree, i.e. it lowers the reward of leaves that are further away. The settings of the two AIs for the second evaluation tournament are:

AI 1: $zug\_kosten = 3$ and $zug\_faktor = 18$

AI 2: $zug\_kosten = 1$ and $zug\_faktor = 6$

AI	Wins	Draws	Lost	WD/L-Ratio	W/L-Ratio	Won	Not-Lost	Points
AI 1	167	202	131	2.82	1.27	33.40%	73.80%	703
AI 2	131	202	167	1.99	0.78	26.20%	66.60%	595

Table 6: Parameter-Evaluation Tournament: Rewards & discountfactor 2

This time the performances of the two AIs are closer to each other, but AI 1 still has a noticeable advantage. Thus, giving leaves that can be gathered faster than others a significant higher reward increases the performance of Minimax algorithm. Additionally the higher reward for near leaves makes sure that the AI doesn't *ignore* a near leaf, e.g. doesn't move fast to an area with many leaves without gathering the near leaf on the run.

5.3.3 Greedy vs. *smart*

In the previous evaluation tournaments we changed single parameters of an AI and let it play against the unchanged version of itself. Because we didn't change anything else, both version of course used the same evaluation function. So in this section we will take two versions that have only some very small differences in their parameters, but a very different evaluation function.

The evaluation function of the first AI is defined as:
$Eval(state) = 5 * knoten.wert - 2 * knoten.wert2 + meinUmfeld(range = 3) - naehstesBlatt$

This AI uses 4 different factors to value the state. It not only tries to maximize its own gain, it additionally considers its environment and the actions taken by the opponent in the game tree path. Thus it tries to maximize its outcome, while also trying to get in a good position for the next turn and minimizing the outcome for the opponent. Due to the AI is considering the environment and the opponent and not only itself, it will be the *smart* AI in the evaluation tournament.

The evaluation function of the second AI is defined as:
$Eval(state) = 4 * knoten.wert + 2 * knoten.meinePunkte$

The evaluation function of the second AI is much easier than the one of the first AI. This AI doesn't really care about its environment and the opponent in the evaluation function. It just tries to maximize its own outcome, gathering as many leaves as possible, no matter what happens around it. Thus this will be the *greedy* AI in the evaluation tournament.

Again the evaluation tournament consists of 500 challenges between the *greedy* and the *smart* AI and led to the following statistics:

AI	Wins	Draws	Lost	WD/L-Ratio	W/L-Ratio	Won	Not-Lost	Points
smart	216	170	114	3.39	1.89	43.20%	77.20%	818
greedy	114	170	216	1.31	0.53	22.80%	56.80%	512

Table 7: Parameter-Evaluation Tournament: Greedy vs. *smart*

What seems to be intuitively right is verified by the result of the evaluation tournament: the *smart* AI performs better than the *greedy* AI. Trying to *steal* a leaf from the opponent, considering the actions of the opponent, don't just running to the nearest leave but instead going to an area with more leaves, i.e. considering the whole state and information improves the performance of the AI.

But the more complex the evaluation function becomes, i.e. if the AI becomes *smarter*, the harder it is to tune the weights for each factor.

5.3.4 One-Action Plan vs. *N*-Action Plan

Due to the Minimax algorithm creates a game tree up to a given depth, we can extract a sequence of the next best actions and not only one next best action. If we got the next best N actions out of the Minimax algorithm, we can play N turns without doing a new calculation again. Thus this would save a lot of calculating time. But on the other hand for the next *N-1* turns you don't react to the actions of your opponent. Although the Minimax algorithm simulates the actions of the opponent, it only considers the actions that it *believes* the opponent will do, not what he really does. So this could be a problem.

To evaluate the influence of different *N*-Action plans we create two different AIs again, which are equal in all parameters and weights *except* the planning size of their actions. The parameter that controls this is called *weg_size* in the implementation. In the first tournament an AI with $weg\_size = 1$ plays against an AI with $weg\_size = 3$, i.e. an AI which calculates a new Minimax run every turn and performing one resulting action against an AI which only calculates a new Minimax run after it performed 3 actions of the last run. As before they played 500 challenges for the following statistics:

AI	Wins	Draws	Lost	WD/L-Ratio	W/L-Ratio	Won	Not-Lost	Points
$weg\_size = 1$	251	104	145	2.45	1.73	50.20%	71.00%	857
$weg\_size = 3$	145	104	251	0.99	0.58	29.00%	49.80%	539

Table 8: Parameter-Evaluation Tournament: One-Action Plan vs. *N*-Action Plan 1

The result of the evaluation tournament is quite clear and matches with the thoughts we had before: calculating a new Minimax run in every turn and reacting to the performed actions of the opponent clearly improves the strength of the AI. The huge difference between the two AIs is even more surprisingly and interesting, because they use the exact same version of the Minimax algorithm. So reacting in each turn to the performed action of the opponent is very crucial for the strength, i.e. $weg\_size = 1$ seems to be the best setting.

We said that if we don't calculate a new Minimax run in every turn, we save a lot of time. So can we improve the performance of the $weg\_size = 3$ AI if we use this saved time? We already know that searching deeper improves the strength, but at the expense of time. Now we have some saved time we can put into a deeper search. Thus for the next evaluation tournament, we take the same two AIs as before, but increasing *maxTiefe* from 11 to 13 for the $weg\_size = 3$ AI.

AI	Wins	Draws	Lost	WD/L-Ratio	W/L-Ratio	Won	Not-Lost	Points
$weg\_size = 1$	213	141	146	2.42	1.46	42.60%	70.80%	780
$weg\_size = 3$	146	141	213	1.35	0.69	29.20%	57.40%	579

Table 9: Parameter-Evaluation Tournament: One-Action Plan vs. *N*-Action Plan 2

We see that the increased search depth has improved the performance of the $weg\_size = 3$ AI. So with a larger *weg_size* you save a lot of time which can be used for a deeper search, which can be very useful in a environment with restrictions. The disadvantage of not reacting to every action of the opponent immediately still affects the performance of the AI and calculating a new Minimax run every turn still gives the best performance.

5.3.5 *Normal*-BFS input vs. *Area*-BFS input

In **Section 4.2** we described two different implementations of the Breadth First Search algorithm and in **Section 4.3.1** we saw how we can use the information from the Breadth First Search while creating the Minimax game tree. Now in this section we want to test the two versions against each other. For this purpose we change the *bfs_sicht* parameter, which controls the Breadth First Search input for the Minimax algorithm as described in **Section 4.2** of our evaluation AI. One version plays with the *normal* Breadth First Search version, i.e. $bfs\_sicht = -1$, and the other one plays with the *Area* Breadth First Search version with a range of 5, i.e. $bfs\_sicht = 5$. As before all other parameters are equal.

AI	Wins	Draws	Lost	WD/L-Ratio	W/L-Ratio	Won	Not-Lost	Points
$bfs\_sicht = -1$	270	54	176	1.84	1.53	54.00%	64.80%	864
$bfs\_sicht = 5$	176	54	270	0.85	0.65	32.20%	46.00%	582

Table 10: Parameter-Evaluation Tournament: *Normal*-BFS input vs. *Area*-BFS input

The statistics of the evaluation tournament show a clearly better performance of the *normal* version as input than of the *area* version. This may be a bit unintuitive, because the *area* version provides more paths to more leaves, and thus more information about the world that the Minimax algorithm could use. Obviously the Minimax algorithm can't use that more information directly to improve its behaviour. The parameters and weights need to be adapted to deal with this more information. When only changing the Breadth First Search parameter, the *normal* Breadth First Search version as input for the Minimax algorithm leads to the best performance of the Minimax based AI.

6 Conclusion & Outlook

Our evaluation tournaments against the three different Reference AIs showed that the Minimax based AI is the best, *strongest* AI of them. The Minimax algorithm together with the AlphaBeta-Pruning uses the most information about the world and state in the most complex way. Acting more game dependent and goal oriented significantly improves the behaviour of the AI, i.e. informed search, with a *good* evaluation function, outperforms uninformed search. The Minimax based AI additionally takes the opponent behaviour into account and by this improves the behaviour even more and performs *smarter*, i.e. isn't just gathering as many leaves as possible, but tries to gather them in a smart way, e.g. *stealing* from the opponent. Dealing with the opponents behaviour is very crucial in a competitive environment to get a strong AI.

With the parameter evaluation tournaments we identified the crucial parts of the Minimax based AIs. The most crucial parameters are the search depth and the evaluation function. Searching deeper clearly improves the behaviour due to the AI *sees* further into the future and to the end of the challenge and by this can choose an action that leads more likely to the best situation for it. We also saw that giving closer leaves a higher reward, i.e. discounting future rewards, leads to a better behaviour of the AI. What we already found out at the evaluation tournaments was proved again when we tested a *greedy* against a *smart* AI, i.e. using more information, especially information about the opponent, is very crucial and the evaluation function defines the behaviour of the AI, i.e. what *kind* of AI it is. When we tried to speed up the AI by using more than one action of the Minimax search, we saw that the behaviour decreases because the AI couldn't react anymore to each action of the opponent immediately. But we also found out that we can use the saved time to search deeper and thus improve the behaviour again. These two parameters can be tuned together to get the best behaviour under the given restrictions of the environment. With the last parameter evaluation tournament we found out that just increasing the used amount of information doesn't automatically lead to a better performance. Using more information needs tuning all involved parameters and weights again and this becomes more complex.

Using all this knowledge and different AIs that were created, we got several reference AIs for the participants of the BWINF with different behaviour that they can challenge to test their own AIs and improve themselves.

We investigated the behaviour and strength of the Minimax based AI against several other approaches, evaluated the influence of different parameters and were able to adapt the Minimax algorithm to the certain game and setting of the BWINF. The resulting, completely handcrafted, Minimax based AI performed very well, even though it misses one very crucial part of (artificial) intelligence: learning; the ability to improve yourself from feedback of the environment and past experiences. The performance of the Minimax based AI may be increased through adding learning, e.g. using machine learning approaches for learning the weights of the factors in the evaluation function, or the different parameters used while creating the game tree. Besides weights and parameter learning, the setting of the used Tourality game is quite perfect for some Reinforcement Learning[17] like e.g. Q-Learning. Through the complete discrete and deterministic setting, basic approaches like Policy/Value Iteration can be adapted, or approximating approaches like Value Function Methods and Policy Search. They *just* need to be adapted to deal with the competitive environment. Another kind of approach that could improve the behaviour is Imitation Learning, even in two different styles: learning from a demonstrated game of a teacher or by using the behaviour of the opponent AI as input. Remember the game setting, each challenge consists of two sweeps with swapped positions. Thus the AI can look at the behaviour of the opponent in the first sweep and using this for its own behaviour in the second sweep and thus play maybe at least a draw in every challenge. So adding any kind of learning to our Minimax based AI to improve its performance, is a very interesting and exciting approach to be investigated and may lead the AI to behave more like something *intelligent*.

[17] see Russell and Norvig (2011) chapter 21 and Mitchell (1997) chapter 13

References

Cormen, T. H., Leiserson, C. E., Rivest, R., and Stein, C. (2001). *Introduction to Algorithms*. MIT Press, McGraw-Hill.

Hart, P. E., Nilsson, N. J., and Raphael, B. (1972). *Correction to 'A formal basis for the heuristic determination of minimum cost paths'*. SIGART Newsletter, 37, pp. 28-29.

Hart, T. P. and Edwards, D. J. (1961). *The tree prune (TP) algorithm. Artificial intelligence project memo 30*. Massachusetts Institute of Technology.

Knuth, D. E. (1975). *An analysis of alpha-beta pruning*. AIJ, 6(4), pp. 293-326.

Lee, C. Y. (1961). *An algorithm for path connection and its applications - IRE Transactions on Electronic Computers*. EC-10(3), pp. 346-365.

Millington, I. and Funge, J. (2009). *Artificial Intelligence for Games*. Morgan Kaufmann.

Mitchell, T. M. (1997). *Machine Learning*. McGraw-Hill.

Moore, E. F. (1959). *The shortest path through a maze - In Proceedings of the International Symposium on the Theory of Switching*. Harvard University Press, pp. 285-292.

Newell, A., Shaw, C., and Simon, H. (1958). *Chess Playing Programs and the Problem of Complexity - IBM Journal of Research and Development*. 4(2), pp. 320-335.

Russell, S. and Norvig, P. (2011). *Artificial Intelligence - A Modern Approach*. Pearson Education.

von Neumann, J. and Morgenstern, O. (1944). *Theory of Games and Economic Behavior*. Princeton University Press.

Appendix

Source-code snippet of the Minimax based AIs:

```
private int ab_decision(Node node, byte maxTiefe, int alpha, int beta, byte
    zug_faktor) {
    int value = max_value(node, alpha, beta, maxTiefe, zug_faktor);
    node.wert = value;
    return value;
}

private int max_value(Node node, int alpha, int beta, byte tiefe, byte
   zug_faktor) {
    if (tiefe == 0 || (blaetter.size() - node.blaetter.size()) == 0) {
        wert(node);
        return node.wert;
    }

    int value = -10000000;
    if (node.kinder == null)
        node.kinder = expand(node, meineID, zug_faktor);

    if (node.kinder != null && node.kinder.size() > 0) {
        Collections.sort(node.kinder);
        for (Node kind : node.kinder) {
            value = Math.max(value, min_value(kind, alpha, beta, (byte) (
               tiefe - 1), zug_faktor));

            if (value >= beta) {
                while (node.kinder.getLast() != kind)
                    node.kinder.removeLast();

                return value;
            }
            alpha = Math.max(alpha, value);
        }
    }

    node.wert = value;
    return value;
}

private int min_value(Node node, int alpha, int beta, byte tiefe, byte
   zug_faktor) {
    if (tiefe == 0 || (blaetter.size() - node.blaetter.size()) == 0) {
        wert(node);
        return node.wert;
    }

    int value = 10000000;
    if (node.kinder == null)
```

```
44          node.kinder = expand(node, gegnerID, zug_faktor);
45
46      if (node.kinder != null && node.kinder.size() > 0) {
47          Collections.sort(node.kinder);
48          Collections.reverse(node.kinder);
49          for (Node kind : node.kinder) {
50              value = Math.min(value, max_value(kind, alpha, beta, (byte) (
                    tiefe - 1), (byte) (zug_faktor - zug_kosten)));
51
52              if (value <= alpha) {
53                  while (node.kinder.getLast() != kind)
54                      node.kinder.removeLast();
55
56                  return value;
57              }
58              beta = Math.min(beta, value);
59          }
60      }
61
62      node.wert = value;
63      return value;
64  }
```

Listing 1: Minimax algorithm with AlphaBeta-Pruning in Java