

# **informatiCup 2021 - Team luceki**

## **Theoretische Ausarbeitung**

Max Lütkemeyer, Leon Cena, Robin Killewald

17. Januar 2021



# Inhaltsverzeichnis

<b>1 Einleitung</b>	<b>4</b>
1.1 Spielprinzip und Regeln . . . . .	4
1.1.1 Aufgabe und Vorstellung . . . . .	4
1.1.2 Spielvorstellung . . . . .	4
1.2 Motivation & Relevanz . . . . .	4
1.3 Vorüberlegungen und Prioritäten . . . . .	6
1.4 Organisation der Teamarbeit . . . . .	6
<b>2 Grundlagen</b>	<b>8</b>
2.1 Programmiersprache & Umgebung . . . . .	8
2.2 Websockets . . . . .	8
2.3 Gulp . . . . .	8
2.4 Künstliche Intelligenz . . . . .	9
<b>3 Struktur</b>	<b>10</b>
3.1 Einführung in die Struktur . . . . .	10
3.2 App . . . . .	11
3.3 Taktik Controller . . . . .	12
3.4 Monitoring . . . . .	14
3.5 Client . . . . .	15
3.6 Datenbank . . . . .	16
3.7 Testserver . . . . .	17
<b>4 Taktiken</b>	<b>18</b>
4.1 exampleTactic . . . . .	18
4.1.1 Intention . . . . .	18
4.1.2 Einsatzmöglichkeiten . . . . .	18
4.2 dontHit . . . . .	18
4.2.1 Grundgedanke . . . . .	18
4.2.2 Implementierung . . . . .	20
4.2.3 Randomisierung . . . . .	21
4.2.4 Semi-Randomisierung . . . . .	21
4.3 recursiveBnB . . . . .	22
4.3.1 Beschreibung . . . . .	22
4.3.2 Rekursion . . . . .	23
4.3.3 Branch and Bound . . . . .	24
4.3.4 Randomisierung und Tiefensuche . . . . .	27
4.3.5 Implementierung . . . . .	28
4.4 dangerFields . . . . .	29
4.4.1 Beschreibung . . . . .	29
4.4.2 Heatmap . . . . .	30
4.4.3 A*-Algorithmus . . . . .	31

4.4.4	Implementierung . . . . .	35
4.5	tfModels . . . . .	35
4.5.1	Einstieg . . . . .	35
4.5.2	Grundlagen . . . . .	36
4.5.3	Implementierung . . . . .	38
4.5.4	Optimierung . . . . .	41
<b>5</b>	<b>Analyse</b>	<b>43</b>
5.1	Vergleich und Diskussion . . . . .	43
5.2	Laufzeitanalyse . . . . .	46
5.3	Benutzeroberfläche . . . . .	53
5.4	Datenbank . . . . .	58
5.5	Testen . . . . .	62
5.6	Datenanalyse . . . . .	65
<b>6</b>	<b>Abschlussdiskussion</b>	<b>68</b>
6.1	Zusammenfassung . . . . .	68
6.2	Reflexion . . . . .	68
6.3	Ausblick . . . . .	69

# 1 Einleitung

## 1.1 Spielprinzip und Regeln

### 1.1.1 Aufgabe und Vorstellung

Das folgende Dokument beinhaltet eine theoretische Auseinandersetzung mit der Aufgabe des informatiCups 2021 und unserer Lösung. Wir sind Max Lütkemeyer, Leon Cena und Robin Killewald, Wirtschaftsinformatik Studenten der Westfälischen Wilhelms-Universität Münster im dritten Semester, und bilden das Team Luceki.

Bei Spe\_ed<sup>1</sup> handelt es sich um ein Spiel in dem mehrere Parteien gegeneinander antreten. In der Aufgabe wird gefordert, dass die Teilnehmer eine Software schreiben, die Spe\_ed alleine spielen kann. Darauf basiert unsere theoretische Ausarbeitung, das von uns entwickelte System, welches wir gegen die Konkurrenz antreten lassen.

### 1.1.2 Spielvorstellung

In spe\_ed treffen bis zu fünf Spieler auf einem Feld von variabler Größe zusammen. Diese werden im Folgenden auch als Agenten bezeichnet. Sie bewegen sich nach vorgegebenen Regeln. Dabei ziehen sie eine Spur hinter sich her. Wer auf einen Spielfeldrand oder eine Spur trifft, scheidet aus.

Jede Runde bewegen sich alle Agenten simultan, nachdem sie dem Server ihren Zug innerhalb einer Deadline mitgeteilt haben. Zur Auswahl stehen hierbei folgende Möglichkeiten: die Bewegung beizubehalten, die Geschwindigkeit zu erhöhen bzw. zu verringern oder sich mit bleibender Geschwindigkeit nach rechts bzw. links zu drehen. Die Geschwindigkeit gibt an, wieviele Felder sich der Kopf des Agenten bewegt. Sie liegt initial bei 1 und darf Werte in  $\{x \in \mathbb{N} | 1 \leq x \leq 10\}$  annehmen.

Jede sechste Runde springen Spieler, deren Geschwindigkeit 2 übersteigt, um ein Feld weniger als der Wert ihrer Geschwindigkeit selbst. Einfacher ausgedrückt wird wie in einem regulären Zug verfahren, nur dass lediglich das erste und letzte Feld der Bewegung eine Spur erhält. Übersprungene Felder erhalten keine Spur. Die zuletzt aktive Partei gewinnt das Spiel.

## 1.2 Motivation & Relevanz

Computer verschiedenster Formen bestimmen unseren Alltag in nahezu allen Bereichen. Ein Leben ohne diese vorhandene Hard- und Software scheint für unsere Generation nicht mehr realistisch, obwohl die vielen Formen von Technologien in ihrer heutigen Ausprägung noch nicht viele Jahre existieren. Dazu zählen Smartphones, Smartwatches aber auch der Personal Computer selbst. Ein Bereich, der die meisten dieser Plattformen

---

<sup>1</sup>[https://github.com/InformatiCup/InformatiCup2021/blob/master/spe\\_ed.pdf](https://github.com/InformatiCup/InformatiCup2021/blob/master/spe_ed.pdf), zuletzt abgerufen am 17.01.2020

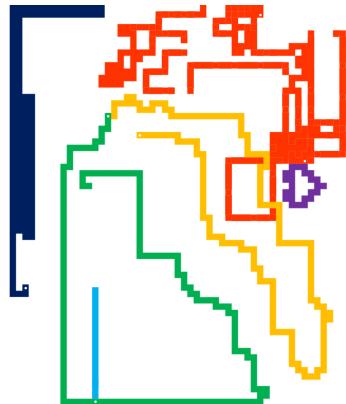


Abbildung 1: Visualisierung eines Spiels

mit einbindet, ist die Welt der Spiele, dessen Entwicklung sich nicht weniger stark auf die Menschheit und ihre Kultur auswirkt.

Wie bereits in der Aufgabenstellung erwähnt wird, gelten Computerspiele als Technologietreiber der Informatik. Hierzu gibt Felix Falk, Geschäftsführer des game - Verband der deutschen Games-Branche, in einem Interview ein konkretes Beispiel: „Die Games-Branche ist besonders innovativ und offen gegenüber neuen Technologien. Häufig ist es sogar so, dass neue Technologien durch die Games-Branche erst einem größeren Publikum bekannt werden. 'Pokémon GO', um hier nur ein Beispiel zu nennen, hat Millionen Menschen Augmented Reality gezeigt. Diese Experimentierfreude der Games-Entwickler ist aber auch deswegen immer wieder so erfolgreich, weil auch die Spielerinnen und Spieler selbst besonders offen gegenüber Neuheiten sind. Auch dadurch ist die Games-Branche immer wieder ein Technologietreiber, wovon wiederum andere Industriebereiche profitieren, denen die Technologien von Games – nehmen wir beispielsweise Virtual Reality oder den Bereich der Simulationen – dabei helfen, noch schneller, noch moderner und noch dynamischer zu werden.“<sup>2</sup>

Das Spiel spe\_ed scheint auf den ersten Blick wenig aufwändig und vergleichbar mit Snake<sup>3</sup>, das als Computerspiel seinen frühen Ursprung bereits in 1976 fand. Jedoch verlangt die Aufgabenstellung nicht das alleinige Spielen, sondern vielmehr ein Problemlösen, indem ein Computer autonom möglichst gut spielen können soll. Die einzelnen Aspekte des Spiels geben einen logischen Rahmen, der für die Automatisierung notwendig ist. Rundenbasiert kann aus den immer selben fünf möglichen Zügen gewählt werden. Alle Spieler haben vorgegebene Grenzen in Position und Geschwindigkeit. Vor dem Hintergrund des Problemlösens verhält sich das Spiel ähnlich wie andere bekannte Pro-

---

<sup>2</sup>[Hooffacker and Bigl, 2020]

<sup>3</sup><https://www.digit.in/features/mobile-phones/a-brief-history-of-snake-33913.html>),  
zuletzt abgerufen am 17.01.2020

blemstellungen der Informatik. Unsere Lösungsansätze haben uns vor neue Teilprobleme, wie dem shortest path problem<sup>4</sup>, gestellt. Motivation für das Projekt finden wir also nicht grundlegend im Spiel selbst, sondern in der Problemstellung, die damit einher geht. Das Automatisieren von Prozessen, hier dem Spielen von spe\_ed, und dem Konstruieren eines Software-Systems zu diesem Zweck hat uns angetrieben und motiviert am Wettbewerb teilzunehmen.

### 1.3 Vorüberlegungen und Prioritäten

Bevor wir mit dem Programmieren beginnen konnten, mussten wir uns die Frage stellen, welche Arten von Algorithmen möglich und sinnvoll sind. Wir kamen zu dem Entschluss, mit regelbasierten und selbstlernenden Algorithmen zu arbeiten.

Die einzelnen Herangehensweisen sind in verschiedenen Taktiken untergebracht worden. Beim Wechsel der Taktiken muss beispielsweise nichts an der Verbindung der Websockets geändert werden. Zu diesem Zweck gibt es einen Verwalter für die Taktiken.<sup>5</sup> Außerdem haben Strategien häufig gleiche Anforderungen, die als Methoden formuliert werden können. Bspw. kann das Prüfen, ob ein Feld frei ist, ausgelagert werden. Hierzu dient eine abstrakte Superklasse tactic<sup>6</sup>.

### 1.4 Organisation der Teamarbeit

Begonnen hat die Organisation unserer Teamarbeit Anfang Oktober mit der Recherche rund um den informatiCup. Neben der Aufgabenstellung haben wir uns die vorherigen Aufgaben und Lösungen angesehen und schließlich mit dem Planen begonnen. Dazu haben wir den verfügbaren Zeitrahmen betrachtet und einen and die Teilaufgabenintensität angepassten Plan erstellt. Die verfügbaren Zeitintervalle haben wir innerhalb unserer Gruppe den verschiedenen Hauptaufgaben zugeteilt. Am Anfang stand nach der ersten Organisation zunächst die Planung der Struktur für unsere Software an. Wie auch in den weiteren Phasen des Projekts haben wir verschiedene Teilziele festgelegt und diese bearbeitet.

Dem Einfluss von Covid-19 geschuldet, ist das Zusammenleben, auch rund um die Universitäten, zu einem großen Teil der Zeit des Wettbewerbs verlangsamt und beeinträchtigt worden. Deshalb haben wir den größten Zeitraum unserer Arbeit räumlich getrennt und online zusammengearbeitet. Dieses haben unter anderem Anwendungen wie GitHub, Zoom und Discord ermöglicht, sodass wir alle zu dem Entschluss kommen, dass ein produktives Arbeiten am Cup durch die Pandemie nicht verhindert wurde. Viel mehr hat die, durch Home Office erhöhte, Flexibilität mehr Zeit und Raum geschaffen, um am Projekt zu arbeiten.

---

<sup>4</sup>[Yu and Yang, 1998]

<sup>5</sup>s. tacticController

<sup>6</sup>s. Superklasse tactic

In regelmäßigen Zeitabständen, im Schnitt etwa alle fünf Tage, gab es ein virtuelles Zusammentreffen für die weitere Planung, Aufgabenverteilung, Problemlösung und oft auch für das gemeinsame Arbeiten an hartnäckigeren Themen oder den Schnittstellen der aktuellen Aufgaben. Dabei lag der Fokus zu Beginn darin, eine Umgebung, in der wir Entwickeln und Spiele analysieren können, zu erschaffen. Jeder von uns hat selbstverständlich Gebiete, in denen er seine Stärken zeigen kann. Deswegen haben wir oft gleichzeitig an Front- und Backend gearbeitet. Beispielsweise konnte ein Teil des Teams bereits die erste Taktik schreiben, während die Website zur Darstellung noch nicht vollständig implementiert war und der andere Teil daran arbeitete, unser System zu testen und antreten zu lassen.

Nachdem das Grundgerüst stand, folgte die Phase, in welcher wir uns mehr auf die Spieltaktiken selbst konzentrieren konnten. Dabei wurde zum Teil alleine und zum Teil zusammen am Code gearbeitet. Bereits in den Phasen der Implementierung haben wir bei der Recherche von möglichen Ansätzen mit Blick auf die schriftliche Ausarbeitung und deren Anforderungen Punkte festgehalten. Modelle, Ansätze und Literatur wurden in einer Cloud gesammelt. Oft haben Skizzen, Screenshots und sogar aufgenommene Videos geholfen, im Team auf den neusten Stand zu kommen, auch wenn an verschiedenen Stellen gearbeitet wurde.

Für den Code sowie die theoretische Ausarbeitung gab es von Beginn an zwei GitHub repositories. Es war oft hilfreich vor der eigenen Arbeit die gemachten Veränderungen zu lesen oder auch auf ältere Versionen des Codes flexibel zurückgreifen zu können. Ebenfalls nützlich waren Kommentare im Code.

Was die theoretische Ausarbeitung betrifft, erfolgte die genaue Umsetzung im letzten Zeitabschnitt unserer Planung. Für die einzelnen Themen wurde zusammen eine Struktur gewählt und für die Teiltexte jeweils gemeinsam die relevanten Punkte in einem Brainstorming gesammelt. Das Ausformulieren der Texte erfolgte wie auch das Schreiben des Codes nach Plan. In festen Zeitabständen wurden neu geschriebene Texte zusammen gelesen und überarbeitet.

## 2 Grundlagen

### 2.1 Programmiersprache & Umgebung

Der Hauptbestandteil der Software ist in JavaScript programmiert. Es werden Features von ECMAScript benutzt, weshalb der Code mit Babel<sup>7</sup> in JavaScript kompiliert wird. Das passiert automatisch. JavaScript wurde ursprünglich im Browser zur Manipulation von Webseiten genutzt. Die Möglichkeit, JavaScript auch Eigenständig zu verwenden, wird durch Node.js ermöglicht. Im Projekt greifen wir häufig auf asynchrone Funktion und Callbacks<sup>8</sup> zurück. Die neue Möglichkeit, auch in JavaScript Klassen verwenden zu können, wurde z.B. in den Taktiken genutzt. Wir haben uns für JavaScript entschieden, da es eine schnelle sowie viel verwendete Programmiersprache ist und wir ein Monitoring früh geplant hatten. Für maschinelles Lernen und Künstliche Intelligenz gibt es tensorflow.js, welches man im Browser und in Node.js verwenden kann.

### 2.2 Websockets

In der Software werden Websockets verwendet, um in Echtzeit Daten auszutauschen. Die Kommunikation mit dem Spielserver wird über das Paket `ws`<sup>9</sup> realisiert. Es ermöglicht die Erstellung eines Websocket Clients und eines Websocket Servers. Für die Kommunikation mit dem Monitoring wird `Socket.IO`<sup>10</sup> eingesetzt. Es erstellt einen Websocket Server, mit dem der Client kommuniziert. Wir haben uns dafür entschieden, weil es bei Socket.IO vorgefertigte Funktionen gibt, die vieles vereinfachen.

### 2.3 Gulp

Um Dateien für einen Webserver zu optimieren, nutzen wir Gulp.<sup>11</sup> Gulp ist ein Task Runner<sup>12</sup>, mit dem wir u.A. Sass Dateien in optimierte, komprimierte CSS Dateien konvertieren. Um alle Client Dateien zu konvertieren, muss man `npm run build` ausführen. In unserer gulpfile<sup>13</sup> sind alle unsere verschiedenen Tasks definiert:

#### Sass Task

Der Task kompiliert Sass zu CSS, setzt automatisch CSS Präfixe und komprimiert abschließend die Ausgabe. Außerdem wird eine Source Map erstellt, um im Browser Fehler zu erkennen. Obwohl der Code auf eine Zeile komprimiert wird, kann so bei der Ausgabe von Fehlern die Zeile mit angegeben werden.

#### Skript Task

Der Task kompiliert durch Babel JavaScript, das für viele Browser verständlich

---

<sup>7</sup><https://babeljs.io/>, zuletzt abgerufen am 17.01.2020

<sup>8</sup>[https://www.w3schools.com/js/js\\_callback.asp](https://www.w3schools.com/js/js_callback.asp), zuletzt abgerufen am 17.01.2020

<sup>9</sup><https://www.npmjs.com/package/ws>, zuletzt abgerufen am 17.01.2020

<sup>10</sup><https://socket.io/>, zuletzt abgerufen am 17.01.2020

<sup>11</sup><https://gulpjs.com>, zuletzt abgerufen am 17.01.2020

<sup>12</sup><https://medium.com/@marcolilli/task-runner-ced8e4113ce6>

<sup>13</sup>Orientiert an <https://www.youtube.com/watch?v=GMakamOBawA>, zuletzt abgerufen am 17.01.2020

ist. Zudem wird die Ausgabe verkleinert und eine Source Map erstellt. Es existiert zusätzlich noch ein *libraries Task*, welcher heruntergeladene Bibliotheken in den Zielordner kopiert.

### Medien Tasks

Die Tasks *img* und *sounds* kopieren die jeweiligen Daten in den Zielordner.

## 2.4 Künstliche Intelligenz

Ab wann spricht man von künstlicher Intelligenz?

“In der Theorie reden wir von Künstlicher Intelligenz, wenn ein Computer Probleme löst, für deren Lösung eigentlich die Intelligenz eines Menschen benötigt wird. Grundsätzlich unterscheidet man zwischen einer starken und einer schwachen KI. Während die starke KI das komplette menschliche Denken mechanisieren soll, dient eine schwache KI zur Lösung konkreter Anwendungsprobleme.”<sup>14</sup>

Die Grenzen des Begriffs sind nicht unbedingt strikt definiert. So wird oft über künstliche Intelligenz gesprochen, wenn im eigentlichen Sinne intelligente Algorithmen gemeint sind. Der Begriff der Intelligenz selbst hat keine klare Definition. So können wir ableiten, dass es sich bei unseren Algorithmen recursiveBnB oder dangerFilds (genaueres dazu in Kapitel 4) ebenfalls um künstliche Intelligenzen handelt, wenn wir annehmen, dass deren Spielweise intelligent ist.

Allerdings wollen wir diese Diskussion nicht ins Detail debattieren. Viel interessanter ist die Verwendung von neuronalen Netzen, die unserer Taktik tfModels ihre Intelligenz verleiht. Zusätzlich zu unseren regelbasierten Algorithmen haben wir also eine schwache KI entwickelt, die, genau wie oben beschrieben, ein spezielles Problem lösen soll: Das Spiel spe\_ed spielen und möglichst oft gewinnen. Die Hoffnung dabei ist, dass die KI Szenarien erkennt und zuordnet, an die wir in der Entwicklung nicht gedacht haben.

---

<sup>14</sup><https://weissenberg-solutions.de/was-ist-kuenstliche-intelligenz>, zuletzt abgerufen am 17.01.2020

## 3 Struktur

### 3.1 Einführung in die Struktur

Wir haben die Software so aufgebaut, dass sie aus vielen Modulen besteht. Das Modul *App* ist die Hauptkomponente, welche die einzelnen Module verwaltet und die Kommunikation mit dem Spielserver bestimmt. Der *TacticController* kümmert sich um die Bestimmung des nächsten Zuges, das Monitoring ermöglicht die Visualisierung und Steuerung unserer Software und das Modul *Datenbank* sorgt dafür, dass Spiele gespeichert und abgerufen werden können. Das Modul *Monitoring* ist hier aufgeteilt in Server und Client.

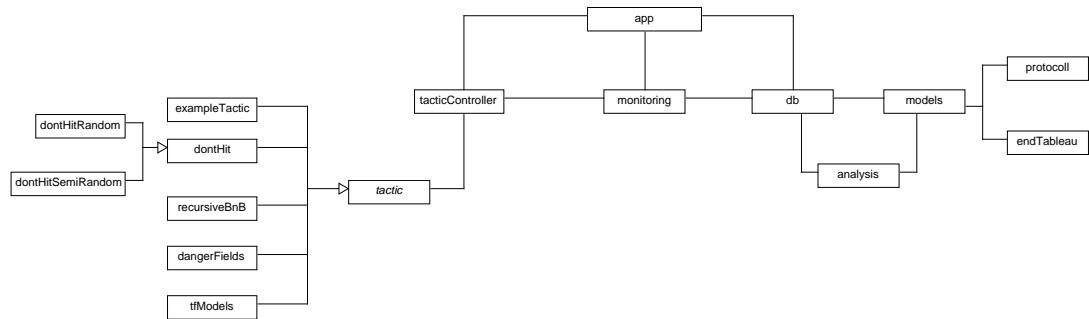


Abbildung 2: Theoretische Struktur der Software)

Des Weiteren gibt es den Trainingsprozess für unsere künstliche Intelligenz.

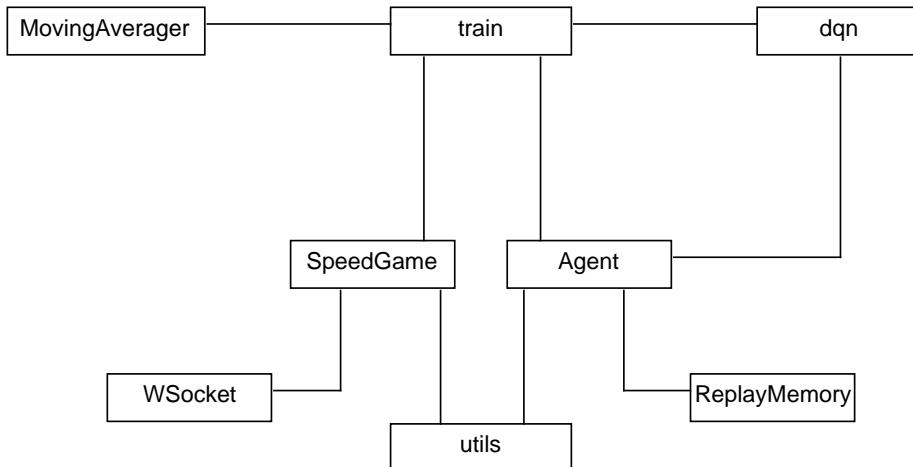


Abbildung 3: Trainingskomponenten für Reinforcement Learning mit Tensorflow.js

Zum besseren Verständnis haben wir außerdem ein “UML nahe” Schema erstellt, womit man schnell sehen kann, was in einem Modul genutzt wird.

name
npm imports
variables
privateMethod(n: number) + publicMethod()

Abbildung 4: Beispiel Diagramm

In der obersten Zeile steht der Name des Moduls. Anschließend sind alle Bibliotheken, die genutzt werden, in der zweiten Zeile ausgeführt. Danach folgen alle globalen Variablen, wobei diese *öffentlich* oder *privat* sein können, d.h. ob sie exportiert werden oder nicht. Zum Schluss kommen dann alle Methoden. Wenn ein Modul als Klasse implementiert ist, wird auf die Sichtbarkeit verzichtet, da die Klasse öffentlich ist.

### 3.2 App

Das Modul App ist die Hauptkomponente unserer Software.

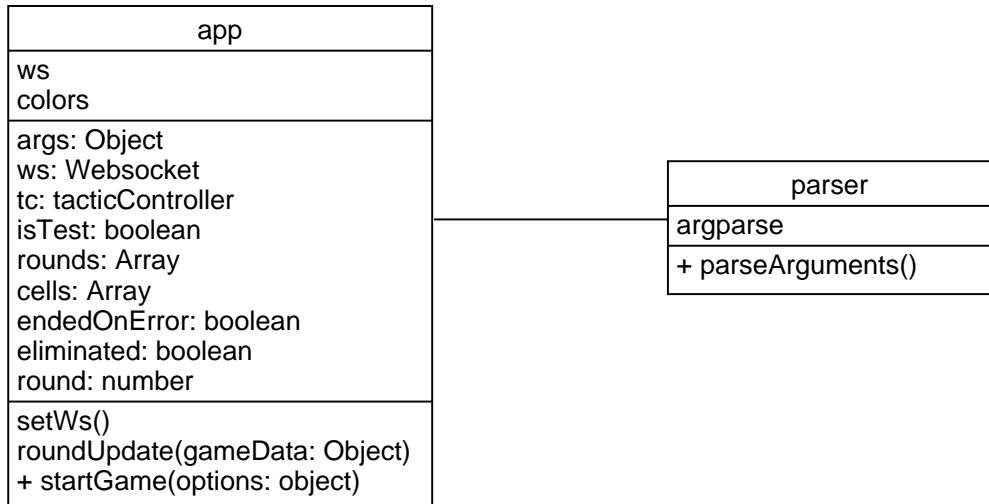


Abbildung 5: App

Es steuert die Zusammenarbeit zwischen den Modulen und verwaltet die Verbindung mit dem Spielserver. Außerdem liest es mit dem Parser alle Argumente der Kommandozeilen aus. Die Verbindung mit dem Spielserver ist wie folgt festgelegt (als Kurzbeschreibung):

- **open:**  
Reset der Variablen *rounds*, *endedOnError* und *eliminated*
- **error:**  
setzte *endedOnError* = *true*
- **close:**  
**Abbruch, falls *endedOnError***  
Das Spiel-Protokoll in der Datenbank speichern.  
Information ans Monitoring, dass es die letzte Runde war.  
Falls Auto Start gewünscht, in ein neues Spiel starten.
- **message:**  
Aus der Nachricht ein Objekt erstellen.  
Protokolldaten aktualisieren.  
Falls Spieler ausgeschieden, leere Antwort senden und Methode beenden.  
Ansonsten den nächsten Zug berechnen lassen und zum Spielserver senden.

Zusätzlich wird bei jedem Event noch eine Nachricht zur Konsole und zum Client geschickt. Folgende Kommandozeilenargumente sind verfügbar:

Argument	Typ	Default Wert
<code>-url</code>	string	"wss://msoll.de/spe_ed"
<code>-key</code>	string	"keinKey"
<code>-timeUrl</code>	string	"https://msoll.de/spe_ed_time"
<code>-clientPort</code>	int	443
<code>-test</code>	int	0
<code>-autoStart</code>	string	"no"

### 3.3 Taktik Controller

Der Taktik Controller hat die Aufgabe, den nächsten Zug zu bestimmen. Er verwaltet alle Taktiken und organisiert deren Instanziierung, Initialisierung und Verhalten. Die Assoziation zu den Taktiken wurde in *Abbildung 2* der Übersicht halber nicht vollständig gezeigt. Die Sichtweise wurde gewählt, da der Taktik Controller ausschließlich Funktionen aus der Superklasse *tactic* nutzt.

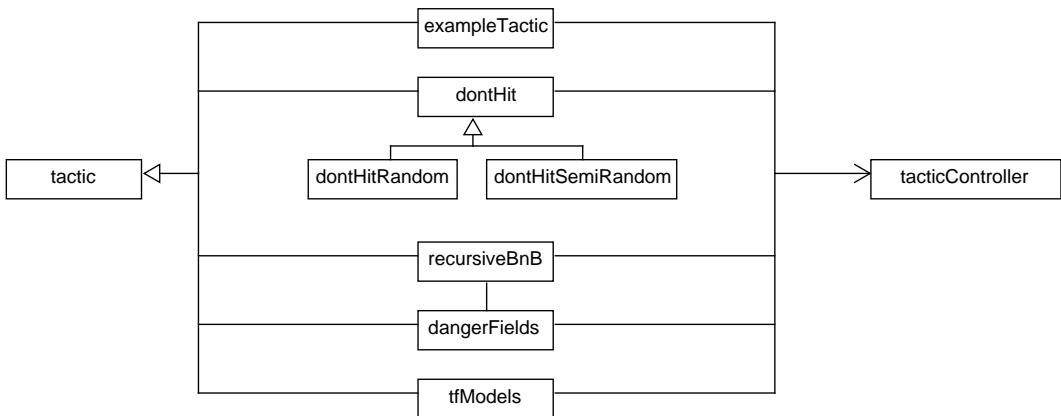


Abbildung 6: Detaillierte Assoziationen des Taktik Controllers

Der Vorteil eines solchen Aufbaus liegt darin, ohne Ändern des Codes, verschiedene Taktiken nutzen zu können, was eine hohe Wartbarkeit bietet. Er ist als Klasse implementiert.

tacticController
colors
tactic: tactic alreadyInit: boolean move: String
constructor() async nextMove(gameData: Object): String init(options: Object) getActiveName(): String

Abbildung 7: Die Klasse *tacticController*

Wichtig ist, dass die Methode *nextMove()* asynchron genutzt werden kann. Die Bestimmung des nächsten Zuges kann viel Zeit in Anspruch nehmen, währenddessen das Programm trotzdem weiterlaufen soll. In ihr wird zusätzlich noch die Zeit der Zugberechnung gemessen und sie sendet alle Visualisierungsdaten der Taktiken zum Monitoring.

### Superklasse tactic

Die theoretisch *abstrakte* Klasse *tactic* ist die Generalisierung aller Taktiken. Sie verwaltet alle Informationen zu dem aktuellen Spiel, sodass jede Taktik einfach darauf zugreifen kann. Wenn *nextMove()* nicht von der Unterkelas implementiert wird, tritt ein Fehler auf. Außerdem bestimmt sie die Zeitdifferenz zwischen dem Spielserver und dem Client

über einen https Request und speichert diese Differenz in dem Attribut *serverTimeDiff* ab.

tactic
https
url
<pre>name: string lastMove: string serverTimeDiff: number dataToSend: Object timeLeft: number height: int width: int cells: Array players: Object you: int running: boolean deadline: number round: int gameData: Object</pre>
<pre>constructor(name: String) init() nextMove() updateGameData(gameData: Object) inTime(): boolean setServerTimeDiff(timeUrl: string) amIAalone(): boolean amIAaloneFill(image: Array, sr: int, sc: int, newColor: int, current: int): boolean</pre>

Abbildung 8: Superklasse *tactic*

Mit *inTime()* kann eine Taktik testen, ob noch genügend Zeit zur Berechnung vorhanden ist. Sobald weniger als 1 Sekunde übrig bleibt, gibt sie *false* zurück. *amIAalone()* findet heraus, ob der Spieler alleine in einem eingezäunten Bereich ist. Dazu wird ein Floodfill<sup>15</sup> Algoirthmus verwendet.

### 3.4 Monitoring

Der serverseitige Teil des Monitoring besteht aus einem Webserver, der mit dem Webframework express<sup>16</sup> konfiguriert ist und einem socket.io<sup>17</sup> Websocket Server, welcher die Echtzeitkommunikation und Steuerung unseres Programmes ermöglicht.

---

<sup>15</sup><https://www.geeksforgeeks.org/flood-fill-algorithm-implement-fill-paint/>, zuletzt abgerufen am 17.01.2020

<sup>16</sup><https://expressjs.com/de/>, zuletzt abgerufen am 17.01.2020

<sup>17</sup><https://socket.io/>, zuletzt abgerufen am 17.01.2020

monitoring
express http https fs path
expressApp: express httpServer webServer io dirs + autoStart + lastOptions
getDirectories(source: string): Array initIo() + initWebserver(HTTPSPORT: int) + update(gameData: Object) + queue() + sendMove(move: string) + sendMessage(message: string) + sendTacticData(data: Object) + sendGameEnd(option: string)

Abbildung 9: Webserver & Socket.io Server

Viele Methoden sind dazu da, Daten an den Client zu senden. Eine andere Herangehensweise wäre *eine* Methode zum Senden zu haben, in der man JSON-Objekte mit einem Atribut sendet. Wichtig hier ist *initIo()*, welche den Socket einstellt. Der Client kann folgende Anfragen stellen:

- **status** liefert den aktuellen Wert von autostart
- **startGame** startet ein Spiel
- **getHistoryList** liefert eine Liste der letzten Spiele aus der Datenbank
- **getGame** liefert ein bestimmtes Spiel aus der Datenbank
- **autostart** de-/aktiviert den Autostart

Wenn kein spezieller Port angegeben wurde, werden zwei Webserver gestartet. Ein http Server, welcher einen Benutzer zu dem https Server weiterleitet und ein https Server, welcher der Hauptserver ist. Wird ein spezieller Port angegeben, wird nur der https Server gestartet.

### 3.5 Client

Der clientseitge Teil des Monitoring besteht aus der Steuerung & Visualisierung eines Spieles und der Benutzeroberfläche für die Datenbank (History). Es werden dynamisch

html Seiten gerendert mit der Viewengine ejs<sup>18</sup>. Es gibt die index und history Seite, welche beide die Komponenten head, footer und leftSidebar importieren, damit weniger boilerplate Code geschrieben werden musste und es weniger Code Duplikation gibt. Sass und JavaScript Dateien werden mit gulp kompiliert und optimiert. Es gibt eine color.sass Datei, welche alle Farben des Clients beinhaltet, eine für den Modifizierten switch button, eine für das generelle Aussehen und jeweils eine für index und history. Skript Dateien sind aufgeteilt in UX und Angelegenheiten bezogen auf den Socket . Es gibt utils, welche gemeinsames von index und history beinhaltet. Die Dateien werden verkleinert und zu normalem JavaSkript gewandelt.

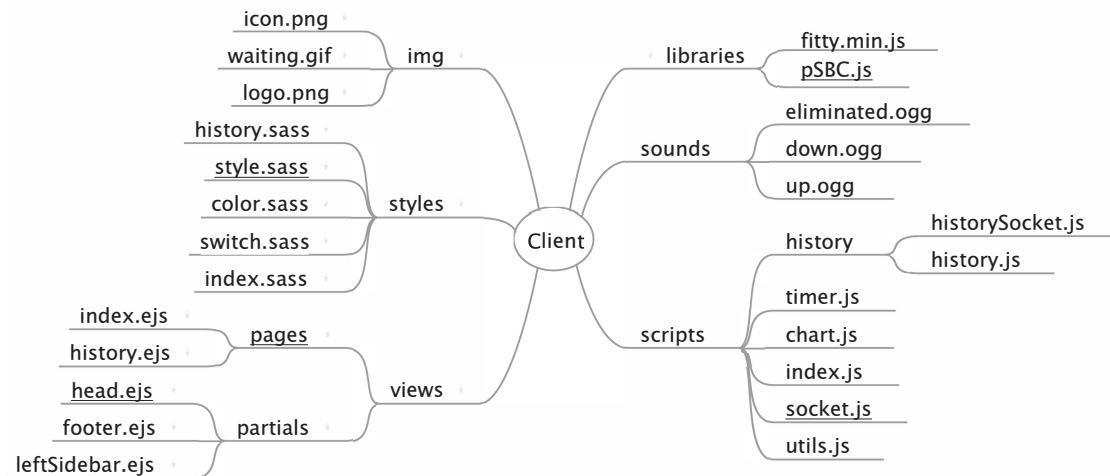


Abbildung 10: Struktur des Clients

### 3.6 Datenbank

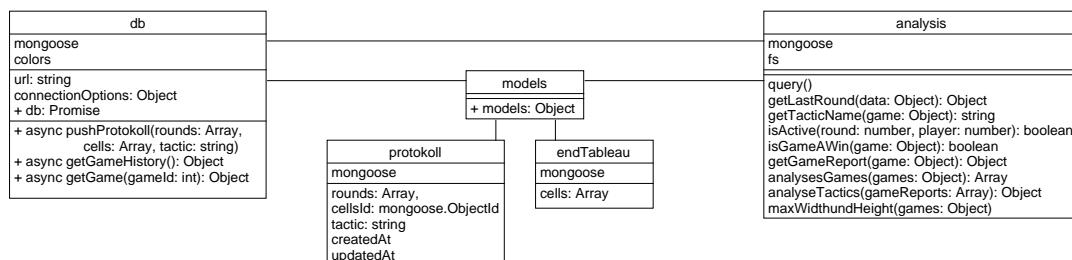


Abbildung 11: Datenbank Komponenten

Als Datenbankmanagementsystem wird MongoDB genutzt. MongoDB ist eine NoSQL Datenbank, welche mit JSON Objekten arbeitet, was ziemlich gut zu diesem Projekt

<sup>18</sup><https://ejs.co/>, zuletzt abgerufen am 17.01.2020

passt, da vom Spielserver und unserer Architektur ausschließlich mit JSON Objekten gearbeitet wird. Die Datenbank hat zwei Collections *endTableaus* und *protokolls*. In *endTableaus* werden die letzten cells Arrays eines Spieles gespeichert und in *protokolls* alle *gameData* Objekte eines Spiels, außer dem cells Array, da dieses zu große Datenmengen verursacht. Man hätte die Daten auch deutlich effizienter speichern können. Da mit der gegebenen Speicherweise keine Probleme auftauchen, haben wir nicht weiter optimiert. Mehr zur Analyse mit der Datenbank in Kapitel 5.3.

### 3.7 Testserver

Das Antreten gegen andere Teams über den gegebenen Spielserver ist hilfreich und lässt uns unsere Leistung vergleichend betrachten. Jedoch dauert es bis zu fünf Minuten bis ein Spiel startet. Dazu kommt, dass innerhalb der Deadline auf jeden Spieler gewartet wird bis die Züge der Runde ausgeführt werden. Die kumulierten Wartezeiten pro Runde waren häufig durchschnittlich über fünf Sekunden, was oft längere Spiele mit bspw. 1000 Runden auf mehr als 1,5 Stunden gestreckt hat. Die Notwendigkeit, beim Debuggen auf Knopfdruck eine Runde mit vielen Spielern starten zu können, hat uns anfangs dazu bewegt, einen Testserver zu erstellen. Wir können dadurch auch unsere eigenen Taktiken gegeneinander antreten lassen und voneinander abgrenzen. Dabei funktioniert der Server ähnlich wie der Spielserver. Betrachtet man unseren sowie den Spielserver als Blackbox, so erfüllen sie den exakt selben Zweck.

Vor dem Spielstart wird auf die Verbindung von sechs Spielern gewartet. Wir können über mehrere Konsolen über verschiedene Ports beitreten. Dazu ist auch vereinfacht die automatisierte Möglichkeit über ein Batch bzw. Shell Startskript gegeben. Haben alle Spieler Autorun aktiviert, werden die Spiele direkt nach Ende und ohne Wartezeit neu gestartet. Das hat das Debuggen von seltenen Fehlern ermöglicht.

Der Server erstellt eine Instanz der Klasse Game. Diese erstellt ein Spielfeld und lässt die Spieler an zufälligen unterschiedlichen Stellen spawnen. Alle Clients senden ihre Züge und der Server führt diese mithilfe der Methode `executeMoves()` aus. Informationen über die Spieler werden analog zum Verhalten des Spielerversers aktualisiert, um anschließend jedem Spieler das individuelle JSON-Objekt zu senden. Dazu verwendet werden die Methoden `createGameData()` von Game und `broadcastGameData()` vom Server. Mit Spielern, Spielen und *gameData* wird objektorientiert gearbeitet.

## 4 Taktiken

### 4.1 exampleTactic

#### 4.1.1 Intention

Die *exampleTactic* verfolgt keine zielgerichtete strategische Funktion im Gegensatz zu den restlichen Taktiken. Die Intention dieser Taktik war es, eine Test-Möglichkeit zu schaffen. Somit bemüht sich die Taktik nicht, ein Spiel zu gewinnen. Dies ist der Grund, weshalb wir nicht detailliert auf die *exampleTactic* eingehen werden.

#### 4.1.2 Einsatzmöglichkeiten

Eine leere Taktik ohne strategische Elemente bietet zahlreiche Vorteile. Oft möchte man beim Entwickeln kleinere Features testen. In der frühen Phase der Entwicklung wurde diese Strategie benutzt, um zu gewährleisten, dass unser Agent eine korrekte Verbindung zum Server hat und in der Lage ist, Nachrichten und zukünftige Spielzüge zu empfangen bzw. zu versenden. Im Rahmen dieses Testes haben wir den Agenten geradeaus laufen lassen.

Eine andere spezifischere Einsatzmöglichkeit ist die Zeitmessung. Wir haben in dieser Taktik die Möglichkeit Code-Ausschnitte bezogen auf die serverseitige Zeitrestriktion zu testen. Dies muss man im Hinterkopf behalten, da im Verlauf der Entwicklung die Funktionen komplexer und rechenintensiver wurden. Daraus folgt eine dementsprechend höhere Rechenzeit. In dieser Taktik kann man mit der Methode *delayed* die Ausführung von Code verzögern, um die Rechenzeit von Features zu testen. Wir möchten hier anmerken, dass wir das Verzögern ausschließlich zu Testzwecken und das auch nur in ca. Zehn runden genutzt haben.

### 4.2 dontHit

#### 4.2.1 Grundgedanke

Die *dontHit*-Strategie ist unsere erste zielgerichtete Strategie, die sich zum Ziel gesetzt hat, das Spiel zu gewinnen. Diese Strategie basiert auf einem Algorithmus, der an Naivität kaum zu überbieten ist und nichtsdestotrotz ein Ausscheiden recht lange verhindert.

Wenn man sich in die Situation eines Agents hineinversetzt und überlegt, wie man auf ein auftauchendes Hindernis reagiert, dann ist Ausweichen eine intuitive Reaktion. Nach diesem Prinzip agiert die Taktik. Der Agent hat favorisierte Entscheidungen, die er ausführt, sofern der Spielzug nicht in Ausscheiden resultiert. Wenn der Spielzug nicht möglich ist, führt er eine alternative weniger priorisierte Entscheidung aus.

#### Formale Beschreibung:

Ein Spiel besteht aus  $n$  verschiedenen Runden. Der Spieler hat über das gesamte Spiel die Auswahloptionen  $o$  und entscheidet sich für einen Spielzug  $z_i$ , wobei  $i$  die Runde ist.

Im Verlauf des Spiels muss der Spieler in jeder Runde eine Entscheidung  $z_i$  treffen. Im ganzen Spiel trifft der Spieler somit  $n$  Entscheidungen für Spielzüge,  $d$  bildet die Menge aller Entscheidungen.

$$o = \{o_1, o_2, o_3\} = \{\text{change\_nothing}, \text{turn\_right}, \text{turn\_left}\}$$

$$z_i \in o$$

$$d = \{\text{change\_nothing}, \text{turn\_right}, \text{turn\_left}\}^n \in \{z_1, \dots, z_n\}$$

Für jede Auswahloption gibt es eine gespeicherte Priorität. Eine Priorität kann nicht mehrfach vergeben werden und jede Auswahloption braucht eine ihr zugewiesene Priorität. Die Priorität  $p_i$  ist der Auswahloption  $o_i$  zugeordnet. Die Prioritätenliste ist eine Permutation der Zahlen  $\langle 1, 2, 3 \rangle$ , wobei größere Zahlen höhere Prioritäten darstellen. Da somit jede Priorität einmal vergeben werden muss, ergeben sich folgende Eigenschaften.

$$p = \langle p_1, p_2, p_3 \rangle, \quad p \text{ ist eine Permutation von } \langle 1, 2, 3 \rangle$$

$$|p| = 3$$

#### **Beispielhafte Priorisierung:**

Wenn man nun möchte, dass der Agent mit größter Priorität geradeaus läuft und bei Hindernissen vorzugsweise nach rechts und alternativ nach links abbiegt, würde sich die Prioritätenliste  $\langle 1, 2, 3 \rangle$  ergeben. Mit dieser Priorisierung würde der Spieler nun die Hindernisse ablaufen und im Zweifel abbiegen, bis es keine Möglichkeit mehr gibt. Das passiert meistens, wenn man sich durch die starre Priorisierung, die oft in rechteckähnlichen Routen resultiert, selber einsperrt. Dies ist natürlich nur ein exemplarischer Fall, der zeigt, dass die Taktik keinesfalls optimal ist.

#### **Beispielhafter Verlauf:**

In der folgenden Abbildung kann man die gewählte Route gut nachvollziehen. Der Spieler ist in Süd-Ausrichtung gestartet und hat angefangen als Route ein Rechteck zu wählen. Da er nicht am Rand gestartet ist und nicht auf Hindernisse gestoßen ist, hat er das initiale Rechteck verlassen und ein zweites, größeres Rechteck gezeichnet. Da wir als erste Priorität *change\_nothing* gewählt haben, zeichnet unser Spieler das Rechteck von außen nach innen. Glücklicherweise hatte der Gegner in dieser Runde eine ähnliche Taktik, jedoch zeichnete dieser das Rechteck von innen, und ist uns dementsprechend kein Hindernis gewesen. Zum Ende des Spiels wird es jedoch kritisch, da man nicht mehr genug Platz hat um Rechtecke zu zeichnen und sich zwangsweise in eine Sackgasse begibt.

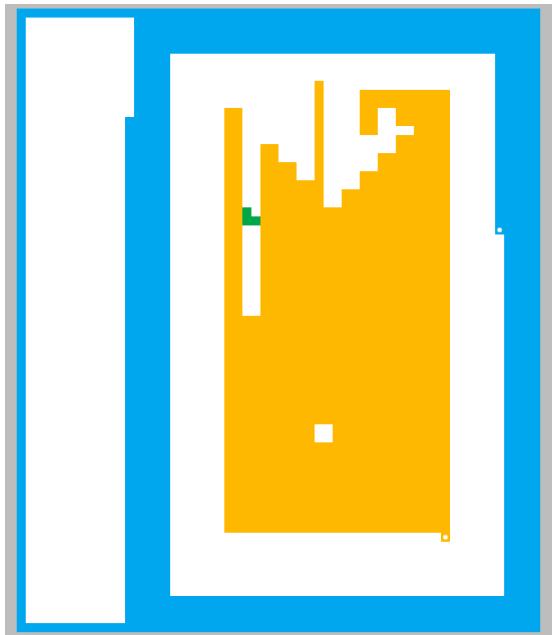


Abbildung 12: Ein beispielhafter Spielverlauf mit der dontHit Strategie (wir sind blau)

#### 4.2.2 Implementierung

Der dontHit Algorithmus ist iterativ implementiert. Zunächst wird die Priorisierung durch das Monitoring im Browser festgelegt. Der Nutzer kann die drei Auswahloptionen mit absteigender Priorität mithilfe des Mauszeigers durch Ziehen sortieren. Die Sortierung wird als Priorisierung in ein Array gelesen und der Taktik übergeben. Im Grunde wird bei jeder Kalkulation in der Reihenfolge der Prioritätenliste die jeweilige Auswahloption auf Zulässigkeit überprüft. Zulässig ist ein Zug, wenn er nicht in Ausscheiden resultiert. Sollte sich die Option als zulässig erweisen, wird sie zurückgegeben und infolgedessen ausgeführt. In so einem Fall sendet unser Agent *speed\_up*. Dies diente ursprünglich Debugging-Zwecken.

#### Überprüfung auf Zulässigkeit

Die Überprüfung auf Zulässigkeit ist in einem getrennten Skript *clear.js* ausgelagert. Es gibt drei clear-Methoden (front, right, left) die jeweils die benötigte Zelle im Spielfeld auf Zulässigkeit überprüfen. Die Zelle muss leer und innerhalb des Spielfeldes sein, damit unser Agent die sie als Punkt der Route benutzen kann. Ein simples Erhöhen der Koordinate ist jedoch nicht möglich, da man die Ausrichtung des Agenten berücksichtigen muss. Wenn wir *clear.front* ausführen möchten, um die Zelle vor unserem Agenten zu überprüfen, fällt auf, dass man für verschiedene Ausrichtungen verschiedene Zellen betrachten muss.

Man muss die clear-Methoden aus der Sicht des Agenten betrachten und nicht aus der Sicht des Nutzers, der eine Vogelperspektive des Spiels einnimmt. Wir haben die

clear-Methoden mithilfe einer *switch-Anweisung* definiert, indem wir für jede Ausrichtung des Spielers festgelegt haben, welche Zellen man auf Zulässigkeit überprüfen muss. Zusätzlich muss die Geschwindigkeit berücksichtigt werden. Dazu kann man die Anzahl der zu überprüfenden Zellen mit der Geschwindigkeit multiplizieren und somit die weiteren Zellen, die der Agent aufgrund der erhöhten Geschwindigkeit ebenfalls besuchen würde, nach den in der *switch-Anweisung* definierten Regeln auf Zulässigkeit prüfen.

### **abstrahierter Pseudocode**

Folgend sieht man einen abstrahierten Pseudocode, der die Funktion des DontHit Algorithmus darstellt.

---

#### **Algorithm 1** dontHit

---

```

1: for  $i$  in  $\{1, \dots, |p|\}$  do                                 $\triangleright p :=$  Prioritätenliste
2:   if überprüfe( $o_i$  mit Priorität  $p_i = 4 - i$ ) then     $\triangleright$  aktuell priorisierte Option
3:     return  $o_i$                                           $\triangleright$  Bei Zulässigkeit : Terminierung
4:   end if
5: end for

```

---

### **4.2.3 Randomisierung**

Beim Testen ist aufgefallen, dass der Agent sich oft in Sackgassen begibt, wenn er einer fixen Priorisierung folgt. Daher haben wir eine zweite Variante des dontHit-Algorithmus entwickelt, dontHitRandom. Dieser Algorithmus basiert auf dem zuvor vorgestellten dontHit Algorithmus, mit dem Unterschied, dass die Prioritäten zufällig bestimmt werden. Diese Randomisierung hat zur Folge, dass keine regelmäßigen Routen mehr gelaufen werden und die berechneten Routen nicht deterministisch sind. Dies hat natürlich Vorteile und Nachteile. Zum Beispiel verhindert man, dass der Agent sich in einem Rechteck einsperrt, andererseits besteht höhere Gefahr sich selbst den Weg abzuschneiden. Die Effektivität der Strategie wird später im Verlauf der Arbeit ausführlich besprochen.

Um den Algorithmus zu implementieren, muss man zu Beginn die Prioritätenliste zufällig anordnen. Die Randomisierung haben wir mit dem *Fisher-Yates-Verfahren*<sup>19</sup> realisiert. Dies ersetzt das manuelle Auswählen der Prioritäten im Monitoring. Der restliche Verlauf gleicht dem dontHit Algorithmus exakt, daher wird auch in dontHitRandom nach der Randomisierung eine Instanz von dontHit erstellt und mit der zufälligen Permutation der Prioritätenliste aufgerufen.

### **4.2.4 Semi-Randomisierung**

DontHitRandom agiert total randomisiert. Diese Taktik gleicht dontHitRandom. Es wird jedoch priorisiert immer der Schritt nach vorne gemacht. Sollte dies nicht möglich sein, wird zufällig entschieden, ob man links oder rechts abbiegt. Das hat den Vorteil, längere

---

<sup>19</sup><https://bost.ocks.org/mike/shuffle/>, zuletzt abgerufen am 17.01.2020

Strecken gehen zu können, ohne Gefahr, sich selbst den Weg abzuschneiden. Außerdem wird auch verhindert, dass man stets Rechtecke zeichnet, wie dontHit es tut, und sich dadurch das Spielfeld künstlich verkleinert. Die Strategie ist somit eine Kombination vom klassischen dontHit und dontHitRandom.

In der folgenden Grafik sieht man einen beispielhaften Verlauf mit dieser Taktik. Man kann beobachten, dass der Agent im markierten Feld rechts abgebogen ist, da er dies zufällig entschieden hat. Die starre Festlegung der Prioritätenliste hätte in diesem Fall dafür gesorgt, dass z.B. oft links abgebogen wird. Das Spielfeld hätte sich in diesem Fall stark verkleinert und die Chancen auf einen Sieg deutlich reduziert.

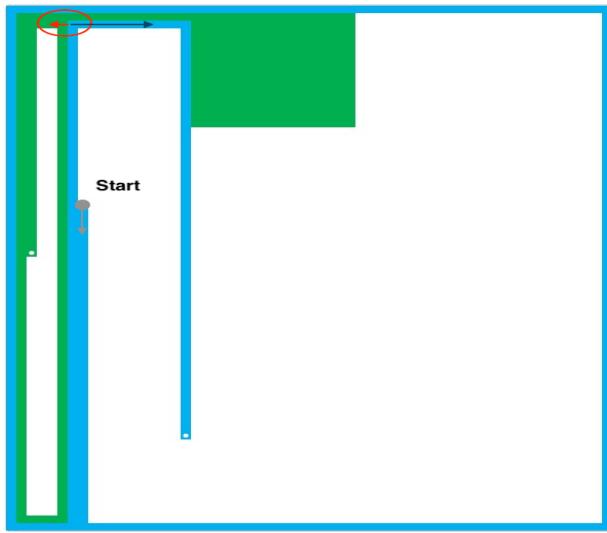


Abbildung 13: Beispielhafte Ausführung einer Runde mit der Taktik dontHitSemiRandom [blauer Spieler]

### 4.3 recursiveBnB

#### 4.3.1 Beschreibung

Die Taktik recursiveBnB nutzt das Prinzip der Rekursion<sup>20</sup> und viele weitere Techniken, um das eigene Spielverhalten in der Zukunft zu analysieren und darauf basierend in der Gegenwart eine Entscheidung zu treffen. Spielzüge werden simuliert, um zu testen, ob sie den Agenten aktiv halten. Dabei untersucht sie, welche Züge in der aktuellen Runde möglich sind, simulierte die möglichen Züge und betrachtet jeden simulierten Zug rekursiv weiter bis die vorgegebene maximale Tiefe von z.B. 30 Zügen erreicht ist oder keine Zeit mehr zum Suchen besteht. Die Taktik ist randomisiert, nutzt Tiefensuche und Branch and Bound, um in der geringen Zeit eine derartige Tiefe erreichen zu können.

---

<sup>20</sup>[Schmaranz, 2002]

### 4.3.2 Rekursion

Formal bedeutet Rekursion, dass ein Prozess auf etwas angewandt wird, um danach auch auf das Ergebnis des Prozesses selbst angewandt zu werden. Dadurch entsteht eine Schleife, wenn in der Programmierung beispielsweise eine Funktion oder Methode sich selbst aufruft. Ein rekursiver Prozess muss jedoch nicht ausschließlich eine programmierte Methode sein. Nimmt zum Beispiel ein Programm den Bildschirm auf und zeigt gleichzeitig die Aufnahme an, so entsteht ebenfalls eine Rekursion. Auch in Disziplinen außerhalb von Software kommen Rekursionen zum Einsatz. Beispielsweise definiert die Mathematik die Fibonacci-Folge<sup>21</sup> rekursiv. Die Rekursion in unserer Taktik ist im folgenden als Baum visualisiert. Die Rekursionstiefe  $n$  kann bei Aufruf der Taktik manuell eingegeben werden und beschreibt die Tiefe des Baums, nicht die Anzahl der Rekursionsaufrufe.

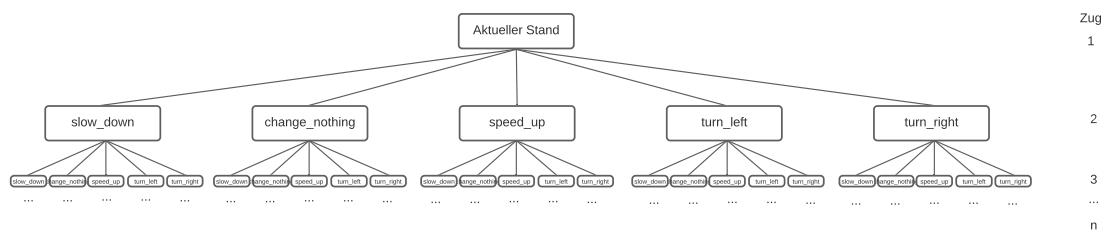


Abbildung 14: Rekursionsbaum

Die Anzahl der zu simulierenden Züge in der Rekursionstiefe  $n$  ist  $5^n$ , da in jedem zukünftigen Szenario jeweils fünf Möglichkeiten zur Auswahl stehen. Untenstehend wird der Verlauf der Funktion  $5^n$  für  $\{n \in \mathbb{N} | 1 \leq n \leq 10\}$  dargestellt:

$n$	1	2	3	4	5	6	7	8	9	10
$5^n$	5	25	125	625	3125	15625	78125	390625	1953125	9765625

Das ist ausschlaggebend für die Formel zur Berechnung der gesamten Anzahl der zu simulierenden Züge. Diese kann rekursiv aus der Summe der Anzahl in der untersten Tiefe  $5^n$  formuliert werden und der Anzahl aus Tiefen darüber  $f(n - 1)$ :

$$f(n) = \begin{cases} 5^n + f(n - 1) & n > 0 \\ 0 & \text{sonst} \end{cases}$$

Mithilfe der geometrischen Reihe<sup>22</sup> kann die rekursive Funktion in einer geschlossenen Form angegeben werden:

<sup>21</sup><https://www.geeksforgeeks.org/program-for-nth-fibonacci-number/>, zuletzt abgerufen am 17.01.2020

<sup>22</sup>[Cormen et al., 2009, Seite 1168 Gleichung A.5 und A.6]

$$\sum_{k=0}^n q^k = \frac{1 - q^{n+1}}{1 - q} \Rightarrow \sum_{k=1}^n 5^k = \frac{1 - 5^{n+1}}{1 - 5} - 1$$

Beispielsweise könnte ein Spieler mit Geschwindigkeit 2, Richtung rechts und einem  $n = 2$  folgende Spielzüge simulieren. Dabei sind die simulierten Spielzüge der aktuellen Runde dunkelgrau markiert und die Spielzüge der rekursiven Simulationen hellgrau dargestellt. Beispielhaft ist der Kopf in der aktuellen Runde, sowie nach einem `speed_up`, und einem `turn_right` eingezeichnet.

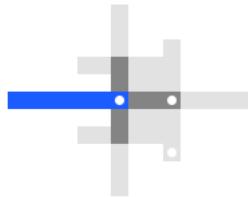


Abbildung 15: Visualisierung der Rekursion im Spielfeld

Da bereits bei  $n > 10$  die zu simulierenden Züge in der Ausführung die Deadline deutlich überschreiten, können nicht immer alle Züge naiv simuliert werden. Um für  $n > 10$  innerhalb der Deadline simulieren zu können, nutzen wir das Prinzip des Branch and Bound und der Tiefensuche.

#### 4.3.3 Branch and Bound

Die Methode des Branch and Bound stammt aus dem Bereich des Operations Research und setzt sich das Ziel, in einem Optimierungsproblem eine Lösung zu finden. Dabei agiert das Verfahren auf einem Entscheidungsbaum und nutzt Schranken, um nach und nach Äste des Baums zu entfernen. Je nach Optimierungsproblem, mit eigenem Entscheidungsbaum und Regeln für Schranken, sehen Algorithmen für Branch and Bound also immer unterschiedlich aus. Jedoch gibt es immer eine Unterteilung in Teilprobleme (Branch wie Verzweigung) und das Kürzen (Bound). <sup>23</sup>

In der Taktik recursiveBnB wird Branch and Bound nicht genutzt um auf die exakte Lösung des Problems zu kommen, sondern um die Lösungsmenge zu unserem Vorteil verringern. Es wird also eher der Ansatz des Kürzens mit einer Schranke verwendet, als ein ganzer BnB-Algorithmus<sup>24</sup>.

Da in den Spielregeln bereits die Beschränkung der Geschwindigkeit auf  $\{x \in \mathbb{N} | 1 \leq x \leq$

---

<sup>23</sup>[Grimme and Bossek, 2018, S. 47-51]

<sup>24</sup>[Boyd and Mattingley, 2007]

$10\}$  gegeben ist, können alle Äste, die diese Bedingung nicht mehr erfüllen, gekürzt werden. Der Zug `speed_up` wird nicht simuliert und weiter untersucht, wenn die Geschwindigkeit bereits 10 ist und der Zug `slow_down` nicht bei einer Geschwindigkeit von 1. Die Auswirkungen dieses Effekts werden im folgenden Szenario veranschaulicht: Angenommen man überprüft bei jedem Zug zu Beginn, ob `slow_down` möglich ist, und angenommen es ist nie möglich (da die Geschwindigkeit initial 1 ist und in diesem Szenario nicht erhöht wird), so kann in jeder Entscheidung der Pfad mit `slow_down` gekürzt werden.

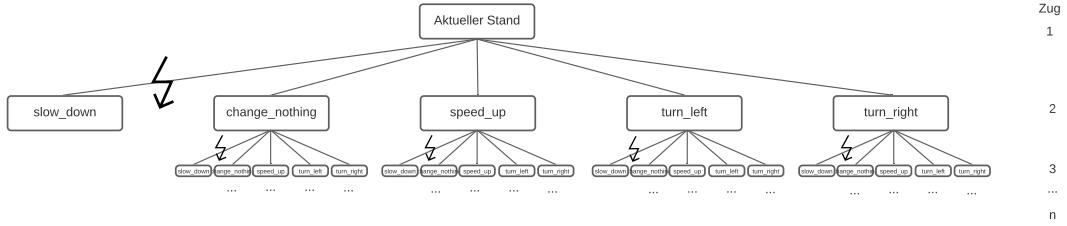


Abbildung 16: Rekursionsbaum mit Branch and Bound der Geschwindigkeit

Für eine Rekursionstiefe von  $n$  verringert sich die Anzahl der zu simulierenden Züge von  $n^5$  auf  $n^4$ . Die Funktionen werden in Abbildung 17 verglichen. Dass die Geschwindigkeit immer 1 bleibt und nie erhöht wird, ist kein abwegiges Szenario. Zur Reihenfolge und Priorisierung der Züge pro Aufruf folgt eine genauere Beschreibung im kommenden Absatz über die Tiefensuche.

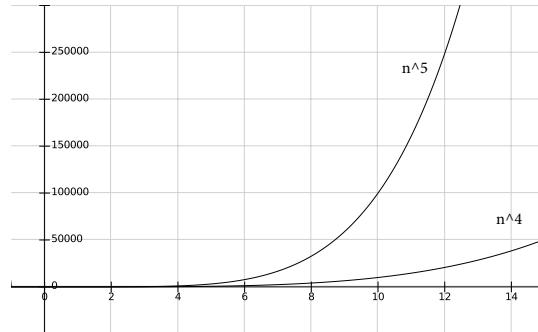


Abbildung 17: Vergleich der Szenarien

Ein weiterer, nicht leicht quantifizierbarer, Effekt ist das Fortschreiten des Spiels. Da die noch aktiven Spieler jede Runde einen Zug machen, und somit mehrere Felder färben, steigt die Wahrscheinlichkeit, auf ein Hindernis zu treffen. Da die Zellen permanent gefärbt werden, wird diese Wahrscheinlichkeit nicht sinken. Daraus folgt, dass mit fortschreitendem Spiel zunehmend immer mehr Teilbäume aus dem gesamten Problem-Baum aus der Beobachtung fallen (*Bound*), da kontinuierlich neue gegnerische Felder besetzt werden. Dieser Effekt mag zu Beginn kaum zu deutlichen Bounds führen, wird jedoch, je länger die Runde dauert, immer wichtiger und prominenter. In längeren Runden ist es

üblich, dass mehrere Spieler größere Spuren auf dem Spielfeld verteilt haben. Es kommt oft vor, dass unser Agent um Hindernisse vorbeimanövriert. Dank des Bounds werden viele Rechnungen erspart und somit auch ein erheblicher Teil der Laufzeit.

Wenn eine Runde bereits länger läuft, kommt der Agent oft in schwierige Situationen und ist nicht immer in der Lage auf Anhieb einen zulässigen Schritt zu finden. In solchen Fällen werden sehr viele Möglichkeiten simuliert. Durch die komplizierte Situation wird indirekt impliziert, dass die Wahrscheinlichkeit auf Hindernisse zu stoßen, deutlich größer ist als zu Beginn. In diesen Stationen wird der große Vorteil von Branch and Bound besonders sichtbar, denn aus einem großen komplizierten Problem werden mehrere kleinere Probleme. Da diese Probleme nicht zwingend weniger kompliziert sind, braucht man nach wie vor viel Zeit zum Simulieren. Das Bounden der unzulässigen Teilbäume gibt uns somit wertvolle Zeit, die wir brauchen, um die potentiell zulässigen Teilbäume zu untersuchen.

Es werden jedoch nicht ausschließlich die Züge mit nicht erlaubten Geschwindigkeiten ignoriert, sondern jeder Zug, der auf dem aktuellen Spielfeld zum Scheitern führt. Darunter fallen auch Züge, die in einem Spielfeldrand oder einer anderen Spur enden. Das gibt der Position des Agenten eine weiter Schranke. Erhalten bei einem Zug  $i$  Felder eine Spur und heißen die Koordinaten der neuen Spurfelder  $x_i$  und  $y_i$  und die Geschwindigkeit nach Ausführung des Zugs  $v$ , so zeigt  $f$ , ob ein Zug mit erlaubter Geschwindigkeit simuliert wird ( $f = 1$ ) oder nicht ( $f = 0$ ). Achtung, die Funktion ist folgendermaßen vereinfacht: Im Code wird zusätzlich überprüft, ob gesprungen wird und übersprungene Felder werden in der Summe vernachlässigt.

$$f = \begin{cases} 0 & \sum_{i=1}^v \text{cells}[x_i, y_i] = 0 \\ 1 & \text{sonst} \end{cases}$$

Beispielhaft ist für einen Agenten (blau) mit Geschwindigkeit 1 und der Richtung rechts für  $n = 2$  in einem belebten Spielfeld folgende Visualisierung der Rekursion aufgezeigt: siehe Abbildung 18. Er befindet sich am Spielfeldrand und in der Nähe eines anderen Spielers (grün). Gezeichnet sind außerdem sein Kopf nach den Zügen `turn_left` und `speed_up`. Im zugehörigen Rekursionsbaum Abbildung 19 sind nur noch Äste eingezeichnet, die simuliert und weiterhin betrachtet werden.

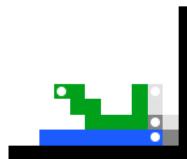


Abbildung 18: Visualisierung der Rekursion mit Branch and Bound im Spielfeld

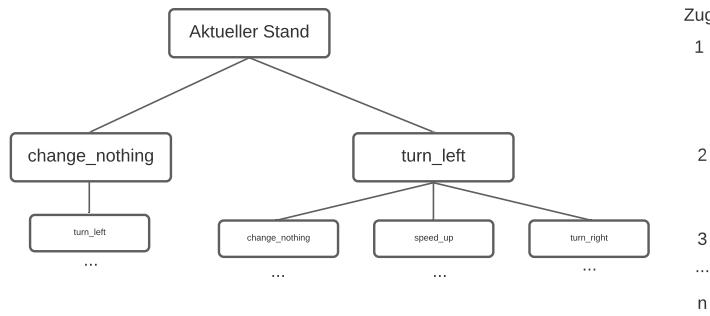


Abbildung 19: Rekursionsbaum im Beispiel

#### 4.3.4 Randomisierung und Tiefensuche

In der Taktik recursiveBnB kann man auswählen, ob Randomisierung eingesetzt werden soll oder nicht. In diesem Fall wird pro Runde, in der die Taktik genutzt wird, eine zufällige Rangfolge an Prioritäten für die Zugmöglichkeiten erstellt. Dazu wird eine Implementation des Fisher-Yates-Shuffle-Algorithmus<sup>25</sup> genutzt. Dieser erzeugt eine zufällige Permutation der Zugmöglichkeiten. Zur Verbesserung der Laufzeit wird innerhalb der Rekursion nicht mit einer Breitensuche BFS, sondern mit einer Tiefensuche TFS traversiert.<sup>26</sup> <sup>27</sup> Wird auf Randomisierung verzichtet, ist die Reihenfolge festgelegt.

Tiefensuche bedeutet, dass die Zugmöglichkeit, die auf der Prioritätenliste vorne steht, immer, wenn sie möglich ist, simuliert wird. Ist diese erste Wahl n mal möglich, wird nach n Aufrufen terminiert und keine andere Zugmöglichkeit betrachtet. In diesem Best Case Szenario ist die Rekursionstiefe gleich der Anzahl der insgesamt simulierten Züge. Ist die erste Wahl nicht möglich, wird die zweite überprüft. Um bei der Breitensuche eine Rekursionstiefe von n zu erreichen, ist der Best Case (angenommen alle Züge sind möglich) gleich dem Worst Case. Es werden alle Züge für alle Rekursionstiefen simuliert bis auf die Tiefe n, bei der nur ein Zug simuliert wird. Aus der Formel für die Gesamtanzahl an möglichen Simulationen ergibt sich "Gesamtanzahl - Anzahl auf Tiefe n + ein Zug auf Tiefe n":

$$\frac{1 - 5^{n+1}}{1 - 5} - 1 - 5^n + 1 = \frac{1 - 5^n}{1 - 5}$$

Somit ist es für die Tiefensuche von Vorteil, wenn viele Flächen frei sind. Für die Breitensuche wäre dies ein Nachteil. Werden Äste durch das Branch and Bound gekürzt, wirkt es sich auf BFS und TFS gleichermaßen aus.

<sup>25</sup><https://www.geeksforgeeks.org/shuffle-a-given-array-using-fisher-yates-shuffle-algorithm/>, zuletzt abgerufen am 17.01.2020

<sup>26</sup><http://www.tilman.de/uni/ws03/alp/tiefen-breitensuche.php>, zuletzt abgerufen am 17.01.2020

<sup>27</sup>[Cormen et al., 2009, Seite 594-597; 603-607]

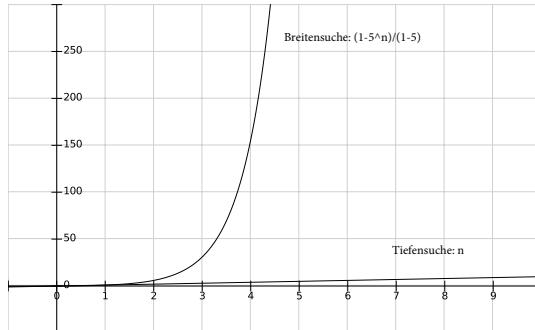


Abbildung 20: Vergleich der Best Case Szenarien BFS und TFS: zu simulierende Züge in Abhängigkeit von der Rekursionstiefe n

Ähnlich wie bei dontHit rechnet der Agent bei einer festgelegten Priorisierung mit freier Fläche nach konstanten Mustern. Jedoch stoppt er dieses Verhalten, wenn der Weg belegt ist. Beispielsweise wird in Form einer Raute verfahren, wenn die Abbiege-Züge auf der Prioritätenliste vorne stehen. Mit der Randomisierung der Züge sind solche Muster ausgeschlossen. Unabhängig von der Randomisierung kann der Spieler, wenn er eingeschlossen ist und etwas Platz zur Verfügung hat, gezielt springen. Er simuliert nur Mögliche Züge. Bei bspw.  $n = 30$  wird er spätestens einen Sprung planen, wenn keine 30 Züge mehr innerhalb des eingeschlossenen Feldes möglich sind. Dabei kann gezielt über ein bis acht Felder gesprungen werden. Dass die Taktik in solchen komplexen Fällen zu lange rechnet, kommt in weniger als einem Prozent der Runden vor und sorgt nur in ungünstigen Fällen für das Ausscheiden. Oft kann nach einem nicht kalkulierten change\_nothing erneut gerechnet werden.

#### 4.3.5 Implementierung

Die Taktik kann als ein Objekt erstellt werden, dem als Attribut die Rekursionstiefe übergeben wird. Soll diese Taktik einen Zug berechnen, wird die Methode nextMove() genutzt. Diese initialisiert den counter der Rekursionstiefe auf 0 und setzt den berechneten Zug auf change\_nothing. Kann die Berechnung des Zugs durch jegliche Art von Beeinträchtigung nicht vollendet werden, wird demnach change\_nothing zurückgegeben. Im Normalfall berechnet sie jedoch mit einer weiteren Methode calcMoveRec() den Zug, wobei das gameData Objekt, das der Server sendet, als Parameter übergeben wird.

Die Methode calcMoveRec() kreiert eine randomisierte Prioritätenliste. Ist die Rekursionstiefe mit einem Pfad an Zügen (bezogen auf den Rekursionsbaum) erreicht, wird der Zug zurückgegeben, der diesen Pfad begonnen hat. Sonst wird, sortiert nach den vergebenen Prioritäten, geprüft, ob die Züge möglich sind. Trifft das für einen zu, wird dieser simuliert und die Funktion für das nun dadurch manipulierte neue gameData Objekt rekursiv aufgerufen. Dabei wird der Counter erhöht. Am Ende der Methode steht erneut die Abfrage, ob die Tiefe erreicht wurde. Das wird mit den ausgelagerten Methoden in possible ermittelt. Diese funktionieren nach dem Prinzip der clear Methoden

des dontHit-Algorithmus, welche in 4.2.2 näher erklärt wurden. Anstatt ein Feld zu prüfen werden bei possible alle Felder getestet, die für den Zug benötigt werden.

Zur Simulation: Die Funktion simulateDecision() erhält die Spielernummer, gameData und die zu simulierende Entscheidung. Sie gibt final die manipulierte gameData zurück. Dabei sind Position und Schweif im Cells Array, sowie Geschwindigkeit und Richtung des Agenten je nach Zug entsprechend aktualisiert. Im abstrahierten Algorithmus wurden die Parameter der Funktion aus Übersichtsgründen weggelassen.

---

### Algorithm 2 calcMoveRec(gameData)

---

```

1: order = getRandomOrder()           ▷ randomisierte Prioritätenliste (Array)
2: if counter ≥ depth then          ▷ Tiefe bereits erreicht?
3:   return chosenMove             ▷ ersten Zug des Pfads zurückgeben
4: end if
5: for all move in order do        ▷ für jede Zugmöglichkeit
6:   if possible.move then         ▷ wenn Zug möglich
7:     if counter = 0 then        ▷ wenn erster Zug eines Pfads
8:       chosenMove = move       ▷ setze neuen chosenMove
9:     end if
10:    simulierteDaten = simulateDecision(gameData, move)    ▷ simulierte Zug
11:    counter++                ▷ erhöhe counter
12:    calcMoveRec(simulierteDaten) ▷ rufe Methode rekursiv auf
13:  end if                      ▷ für zuletzt simuliertes Szenario
14: end for
15: if counter ≥ Tiefe then        ▷ Tiefe erreicht?
16:   return chosenMove            ▷ ersten Zug des Pfads zurückgeben
17: end if

```

---

## 4.4 dangerFields

### 4.4.1 Beschreibung

Die Taktik dangerFields nutzt den A\*-Algorithmus<sup>28</sup>, um zu möglichst sicheren Orten zu kommen. Sie bewegt den Agenten dabei ängstlich von den anderen Spielern und allen Rändern des Spielfeldes weg. Um herauszufinden, wo sichere Orte sind, wird jede Runde eine *Heatmap*<sup>29</sup> vom Spielfeld erstellt. Anschließend wird mit Hilfe des *Flood Fill Algorithmus*<sup>30</sup> die größte sichere zusammenhängende Fläche ermittelt und als Ziel für den A\*-Algorithmus festgelegt. Da sich herausgestellt hat, dass die Taktik eher ineffizient mit dem verfügbaren Platz umgeht, wird am Anfang jeder Runde geprüft, ob der Agent

---

<sup>28</sup>[Herzog et al., 2014]

<sup>29</sup>[Leland Wilkinson, 2008]

<sup>30</sup><https://learnersbucket.com/examples/algorithms/flood-fill-algorithm-in-javascript/>, zuletzt abgerufen am 17.01.2020

alleine bzw. eingesperrt ist. Ist dies der Fall, nutzt die Taktik die Zugbestimmung von dontHit. Die Taktik verwendet den A\*Algorithmus von Brian Grinstead<sup>31</sup>, welcher auf einer anderen Definition für Felder basiert:

1. Ein Gewicht von 0 bezeichnet eine Wand.
2. Ein Gewicht kann nicht negativ sein.
3. Ein Gewicht darf nicht zwischen 0 und 1 liegen (exklusiv).
4. Ein Gewicht kann Dezimalwerte enthalten (größer als 1).

#### 4.4.2 Heatmap

Zuerst wird für jede Zelle  $\text{cells}[y,x]$  eine Transformation durchgeführt:

$$f(x) = \begin{cases} 1 & x = 0 \\ 0 & \text{sonst} \end{cases}$$

Anschließend wird die Heatmap in 2 Schritten erstellt.

1. Es werden 5 Kreise um jede 0 (Spieler) gezeichnet, wobei man mit dem größten Kreis anfängt. Pro Kreis verringert sich der Radius um 1.

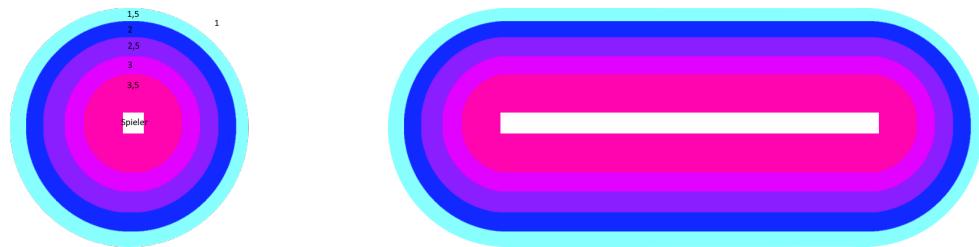


Abbildung 21: Um jedes belegte Feld werden Kreise gemalt

2. Wie in Schritt 1 werden jetzt die Kreise für die Ränder des Spielfeldes erstellt.

---

<sup>31</sup><https://bgrins.github.io/javascript-astar/demo/>, zuletzt abgerufen am 17.01.2020

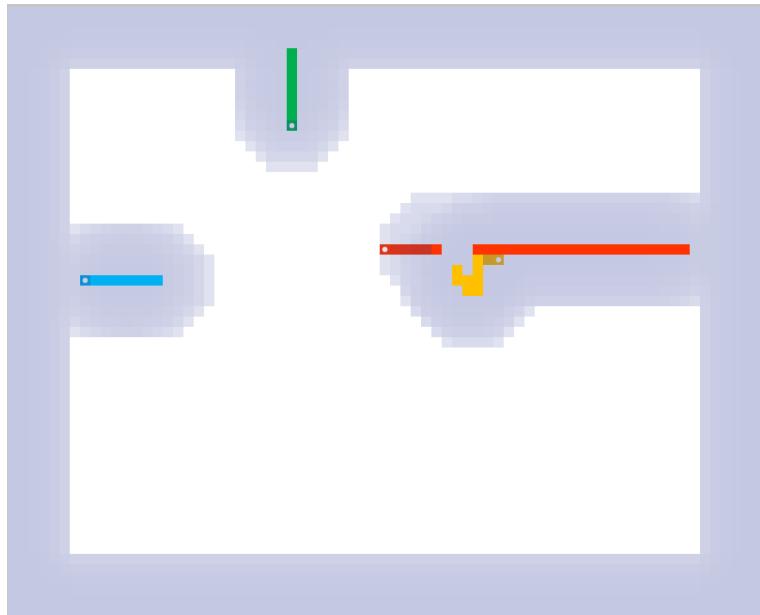


Abbildung 22: Visualisierung der Heatmap im Monitoring

#### 4.4.3 A\*-Algorithmus

Brian Grinstead hat 2 Varianten des A\*-Algorithmus veröffentlicht. Der originale Algorithmus verwendet eine Liste zur Speicherung von Knoten, wohingegen der Neuere einen *Binary Heap*<sup>32</sup> verwendet, welcher eine signifikante Leistungssteigerung bietet. Es folgt der effizientere Algorithmus nach Grinstead: <sup>33</sup>

---

<sup>32</sup>[Cormen et al., 2009, Seite 151-154]

<sup>33</sup><https://briangrinstead.com/blog/a-star-search-algorithm-in-javascript-updated>, zuletzt abgerufen am 17.01.2020

---

**Algorithm 3** A\* Search Algorithm in JavaScript (Updated)

```
1: push startNode onto openHeap
2: while openList is not empty do
3:   currentNode = pop from openHeap
4:   if currentNode is final then
5:     return the successful path
6:   end if
7:   set currentNode as closed
8:   for neighbor of currentNode do
9:     if neighbor is not set visited then
10:      save g, h, and f then save the current parent and set visited
11:      add neighbor to openHeap
12:    end if
13:    if neighbor is in openList but the current g is better than previous g then
14:      save g and f, then save the current parent
15:      reset position in openHeap (since f changed)
16:    end if
17:   end for
18: end while
```

---

Als Einstellung ist festgelegt, dass die Manhattan Heuristik<sup>34</sup> verwendet wird.

---

<sup>34</sup><https://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html#manhattan-distance>, zuletzt abgerufen am 17.01.2020

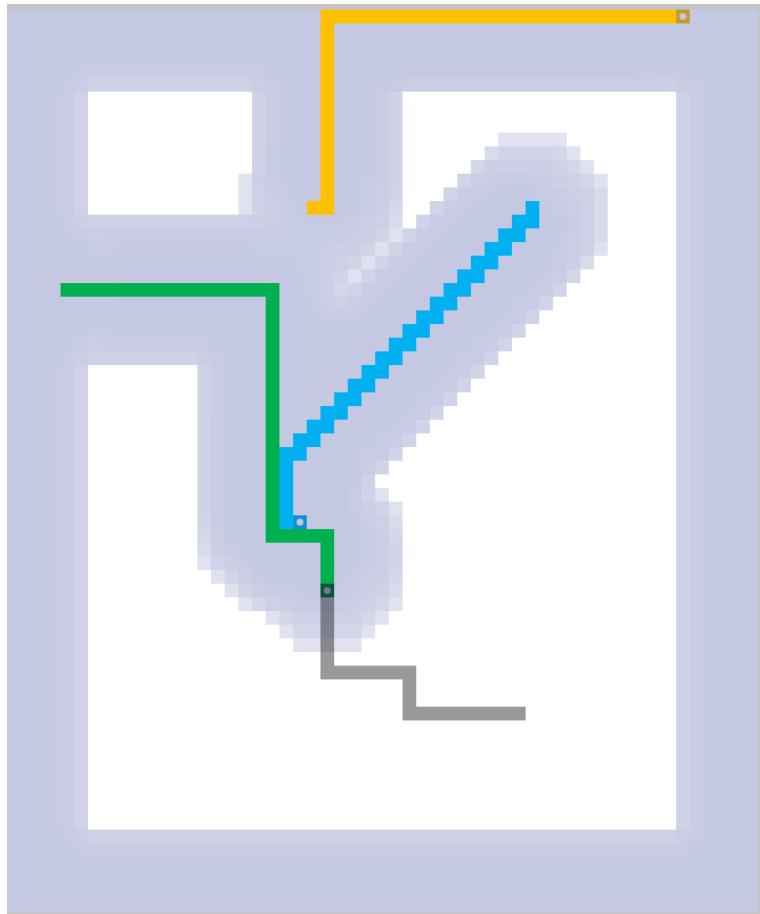


Abbildung 23: Visualisierung des Laufweges

Es stellt sich die Frage, welcher Punkt genau als Ziel dienen soll. Unser erster Ansatz war, ein beliebiges, noch *weißes* Feld zu finden und es als Ziel festzulegen. Falls kein weißes Feld gefunden werden sollte, nimmt man in die Suche Felder mit auf, die gefährlicher sind bzw. ein höheres Gewicht haben. Jedoch hat sich diese Methode nicht bewährt, da nicht unterschieden werden kann, ob es nur ein einzelnes weißes Feld ist oder eine riesige weiße Fläche.

Die neue Zielfestlegung für den A\*Algorithmus sucht nach der größten zusammenhängenden Fläche, in der Gewichte in einem Sicherheitsintervall liegen. Gefunden wird diese Fläche durch den Floodfill Algorithmus, der wie ein Fülleimer in einem Malprogramm funktioniert. Aus dieser Fläche wird dann ein zufälliges Feld als Ziel gewählt. Benutzt wird die Implementierung von Prashant Yadav<sup>35</sup>.

---

<sup>35</sup><https://learnersbucket.com/examples/algorithms/flood-fill-algorithm-in-javascript/>, zuletzt abgerufen am 17.01.2020

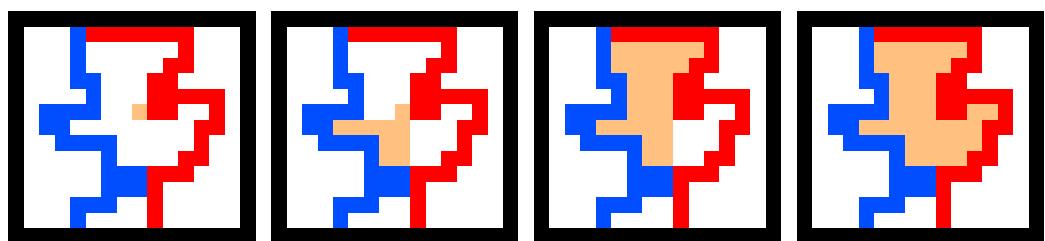


Abbildung 24: Verlauf des Flood Fill Algorithmus

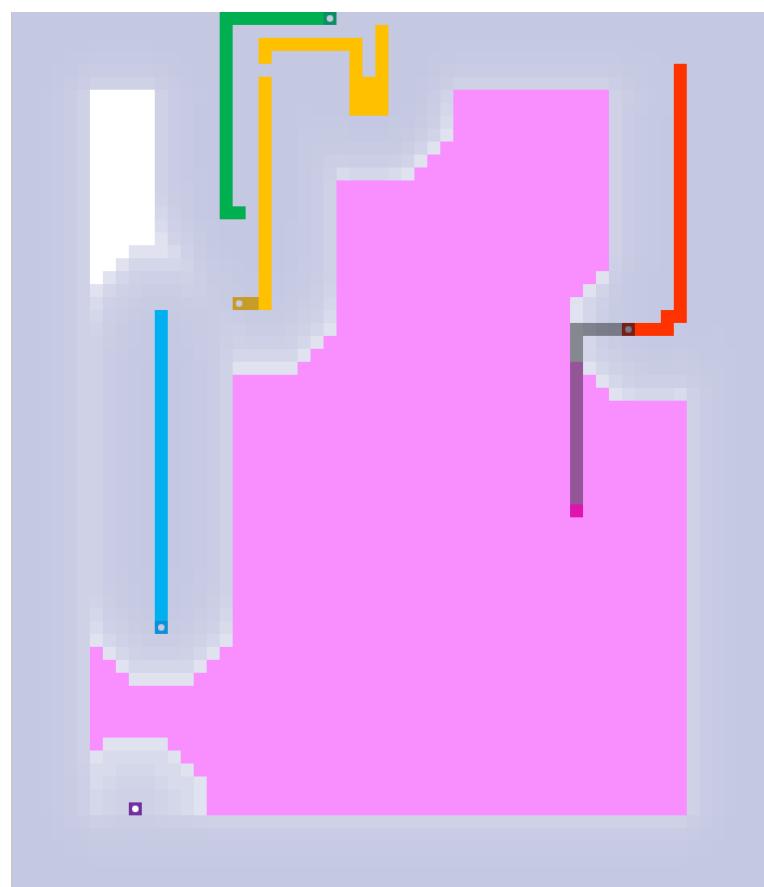


Abbildung 25: dangerFields im Monitoring (Rot)

#### 4.4.4 Implementierung

---

##### Algorithm 4 dangerFields

---

```
1: heatMap = copy(cells)                                ▷ cells Array kopieren
2: heatMap = transform()                               ▷ heatMap transformieren
3: heatMap = heat()                                    ▷ heatMap erstellen
4: safeValue = 1
5: while path == undefined do                      ▷ Solange kein Weg zu einem Ziel gefunden wird
6:   if not inTime() then                            ▷ Abbruch, wenn Zeit abläuft
7:     path = [[0,0]]
8:     break
9:   end if
10:  goal = findGoal(heatMap, safeValue)                ▷ Ziel für Astar mit FloodFill
11:  path = astar(heatMap, goal)                         ▷ Weg bestimmen mit A*
12:  safeValue++
13: end while
14: return moveDependingOnDirection(path)              ▷ Zug bestimmen mit Pfad
```

---

In der Implementierung wird, wenn die Brechnung zu lange dauert oder der Agent eingesperrt ist, eine alternative Taktik verwendet. Es wird dann recursiveBnB mit einem festen Pattern benutzt, damit der restliche Platz möglichst effizient ausgenutzt wird.

#### Optimierungen

Die Taktik nutzt Platz sehr ineffizient. Das liegt in der Natur des Vorgehens. Die Festlegung des Ziels für den A\*Algorithmus könnte schneller sein und es wäre gut, würde man den Prozess mit Trägheit versehen. Es passiert oft, dass das Ziel unkontrolliert hin und her springt. Auch schwierig ist das Szenario, wenn es zwei gleich große freie Flächen gibt. Dann läuft der Agent zu einer Fläche und verkleinert diese, wenn er ihr zu nah kommt. Anschließend ist die andere Fläche größer, weshalb das Ziel die Fläche wechselt und der Agent umdreht.

### 4.5 tfModels

#### 4.5.1 Einstieg

Bisher sind alle Taktiken, die wir entwickelt haben, einfache Algorithmen. Dabei haben wir die Regeln des Spiels abgeleitet um Algorithmen das Spiel nach fester Vorgabe ablaufen zu lassen. Dies ist nicht immer so einfach wie es scheint und man könnte jeder Taktik noch einen Ausnahmefall hinzufügen, damit sie noch besser läuft. Doch so würde man zu lange an der Entwicklung sitzen und wahrscheinlich niemals fertig werden. Betrachtet man andere nicht exakt gelöste Probleme wie das *Traveling Salesperson Problem (TSP)*<sup>36</sup>, dann reicht es oft aus, nur eine sehr gute Annäherung zu der optimalen Lösung zu finden.

---

<sup>36</sup>[Grimme and Bossek, 2018, Seite 67-76]

Man spricht bei der Funktion, die eine Lösung dieser Art liefert, von einer Heuristik. Genau diese wollen wir für spe\_ed benutztten. Die Taktik kann so autonom spielen, Hindernisse umlaufen, Lücken benutzen und gezielt Springen. Damit kann aus einer Gefangenschaft entflohen oder generell, sich von Anfang an so bewegt werden, dass die Gewinnchancen steigen. Wie findet man jedoch eine solche Heuristik? Wir benutztten die Herangehensweise, die die Menschheit als eine der neusten bahnbrechenden Entwicklungen bereichert - Neuronale Netze<sup>37</sup>. Problematisch dabei ist, dass wir ein solches Netz nicht vorliegen haben. Doch zum Glück kann man es genauso trainieren, wie wir Menschen auch etwas lernen. Einfacher wäre es, hätten wir ein paar perfekte Beispielentscheidungen. In diesem Fall kann man einfach das Netz auf diesen Daten trainieren lassen. Möglich ist das mit einer Verlustfunktion, wie dem Mean Squared Error (MSE)<sup>38</sup>, und einer Optimierungsfunktion, wie z.B. Adam<sup>39</sup>. Man spricht in diesem Fall von Supervised Learning<sup>40</sup>, da man mit beschrifteten Trainingsdaten arbeitet. Diese Daten haben wir aber nicht zur Verfügung, weswegen wir auf Reinforcement learning<sup>41</sup> zurückgreifen.

#### 4.5.2 Grundlagen

Betrachten wir die Entscheidungsfindung des besten Zuges erst formell: Wir haben eine Umgebung, in der ein Agent probiert, die beste Entscheidung zu treffen. Er kann die Qualität seiner Entscheidungen überprüfen, indem die Umgebung nach jeder Entscheidung den neuen Zustand und einen Gewinn zurück gibt. Man spricht dabei von einem Markovschen Entscheidungsprozess (engl. MDP). Dass die Entscheidung ausschließlich vom aktuellem Zustand abhängt, ist essentiell für weitere Teile des Codes.

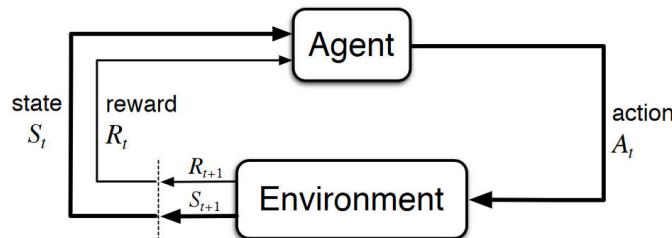


Abbildung 26: Markov Decision Process (MDP) Visualisierung<sup>42</sup>entnommen zur einfachen Veranschaulichung

Die Komponenten eines MDPs sind:

- Agent -> Die Taktik

<sup>37</sup><https://deeplizard.com/learn/video/wrBUkpiRvCA>

<sup>38</sup>[Heumann et al., 2016, Seite 51 Gleichung 3.21]

<sup>39</sup>[Plagwitz, 2018]

<sup>40</sup><https://datasolut.com/wiki/supervised-learning>, zuletzt abgerufen am 17.01.2020

<sup>41</sup>[Cai et al., 2020, Kapitel 11]

<sup>42</sup><https://deeplizard.com/learn/video/my207WNoyeA>, zuletzt abgerufen am 17.01.2020

- Environment -> Der Game Server
- State -> Die aktuelle Game Data ->  $\mathbf{S} = \{gameData_0, \dots, gameData_n\}$
- Action ->  $\mathbf{A} = \{\text{change\_nothing}, \text{turn\_left}, \text{turn\_right}, \text{speed\_up}, \text{slow\_down}\}$
- Reward ->  $\mathbf{R} = \{\text{death}, \text{one\_step}, \text{win}\}$

Der Agent probiert nur die erwartete Summe an zukünftigen Gewinnen zu maximieren.

$$G_t = R_{t+1} + \dots + R_T$$

Ursprünglich gibt es nur zwei verschiedene "Gewinne". Einen für den Fall des Gewinns und einen für den Fall des Verlustes. Dann bekäme der Agent nur selten Feedback für seine Entscheidung, was die Optimierung der Entscheidungen deutlich erschweren würde. Aus diesem Grund gibt es zudem Gewinne für das längere Überleben innerhalb einer Runde. Frühe Entscheidungen haben einen hohen Einfluss auf den Erfolg in der Zukunft inne, weshalb Gewinne mit dem Faktor  $\gamma$  diskontiert<sup>43</sup> werden.

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots + R_T$$

Nun benötigt man nur noch die Antwort auf die nächsten beiden Fragen:

- Was sind die Wahrscheinlichkeiten für einzelnen Aktionen in einem gegebenen Zustand?

Hierfür braucht man eine Richtlinie  $\pi$ , an den der Agent sich orientieren kann. Diese bestimmt die Wahrscheinlichkeiten, welche Aktion zu nehmen ist. Bei uns wird sie durch ein neurales Netzwerk (policy network) umgesetzt.

- Wie gut ist eine Aktion in einem Zustand?

Die Action-Value Funktion  $q_\pi(s, a)$  liefert die Bewertung eines Zustand-Aktions Paars, bzw. den Wert einer Aktion in der Verwendung von  $\pi$ . Man nennt sie auch *Q(uality)-Function*.

$$q_\pi(s, a) = E_\pi(G_t | S_t = s, A_t = a) = E_\pi\left(\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} | S_t = s, A_t = a\right)$$

Die optimale Richtlinie ist gefunden, wenn jeder erwartete, kumulierte und diskontierte Gewinn, in jedem Zustand, besser oder gleich ist als der Gewinn unter Benutzung einer anderen Richtlinie. Die optimale Q-Funktion ist durch die Bellmann Gleichung<sup>44</sup> definiert:

$$q_*(s, a) = E\left[R_{t+1} + \gamma \max_{a'} q_*(s', a')\right]$$

Mithilfe dieser Gleichungen werden die Gewichte zwischen den Neuronen optimiert. Doch wie geht der Agent vor? Er nutzt sein bisheriges Wissen oder erkundet er die Umgebung.

---

<sup>43</sup>[Cai et al., 2020] Gleichung 11.1

<sup>44</sup>[Cai et al., 2020] Gleichung 11.3

Dieses Problem wird auch "Tradeoff between exploration and exploitation" genannt. Beim Erkunden scheidet der Spieler schneller aus, beim Anwenden des Wissens lernt er weniger schnell. Als Lösung verwenden wir den Epsilon-Greedy Algorithmus<sup>45</sup>. Er generiert eine zufällige Zahl, aufgrund derer er sich zwischen den beiden Optionen entscheidet.

---

#### **Algorithm 5** epsilon greedy algorithm

---

```

1: if random_number > epsilon then
2:   choose action via exploitation
3: else
4:   choose action via exploration
5: end if
```

---

Wichtig ist noch, dass ein *Replay Memory* genutzt wird, damit Korrelationen zwischen aufeinanderfolgenden Zustands-Aktions Paaren gebrochen werden. Außerdem verwenden wir ein *target-Netzwerk*, damit der Optimierungsprozess stabil bleibt. Als Literatur empfehlen wir die Reinforcement Learning Video Reihe von deeplizard.com<sup>46</sup>.

Tabelle 1: spe\_ed als kanonische RL Formulierung

Abstraktes RL Konzept	Realisierung in spe_ed
Umgebung	Ein Object mit allen Spielerdaten und Informationen zum Spielfeld Grid
Aktion	(Diskret) 5 Auswahlmöglichkeiten: change_nothing, turn_left, turn_right, slow_down, speed_up
Gewinn	(Regelmäßige, positive und negative Gewinne) Spiel gewinnen - Großer positiver Gewinn (+50) Ausscheiden - Sehr Großer negativer Gewinn (-100) Eine Runde überleben - Kleiner positiver Gewinn (+1) Ein anderer Spieler scheidet aus - Positiver Gewinn (+10)
Observierung	(Kompletter Zustand, diskret) Der Agent kann zu jedem Zeitpunkt auf den ganzen Zustand zugreifen

#### 4.5.3 Implementierung

Zur Implementierung verwenden wir tensorflow.js<sup>47</sup>. Orientiert an Googles Snake-DQN<sup>48</sup> Beispiel, haben wir das Projekt für unsere Bedürfnisse angepasst und eingebunden. Der aktuelle Zustand wird als transformiertes Array repräsentiert. Zur Vereinfachung

<sup>45</sup><https://deeplizard.com/learn/video/mo96Nq1o1L8>, zuletzt abgerufen am 17.01.2020

<sup>46</sup><https://deeplizard.com/learn/video/nyjbcRQ-uQ8>, zuletzt abgerufen am 17.01.2020

<sup>47</sup><https://www.tensorflow.org/js>, zuletzt abgerufen am 17.01.2020

<sup>48</sup><https://github.com/tensorflow/tfjs-examples/tree/master/snake-dqn>, zuletzt abgerufen am 17.01.2020

gibt es nur die folgenden Aktionen:

$$A' = \{\text{change\_nothing}, \text{turn\_left}, \text{turn\_right}\}$$

Den Zustand zu erhalten ist eine Herausforderung. Ursprünglich hat eine Funktion einfach den neuen Zustand nach einer Aktion zurückgegeben. Da `spe_ed` aber über einen Server kommuniziert und diese Kommunikation in JavaScript standardmäßig über *callbacks* funktioniert, ist so ein Schema im engeren Sinne nicht gewollt. Jedoch sind wir bei dem Aufbau geblieben und benutzen ein Paket, das die Kommunikation mit Websockets über *Promises*<sup>49</sup> ermöglicht. Diese Implementierung ist nicht ideal, jedoch war der Aufwand, eine ganz neue Logik zu entwickeln, zu hoch. Es gibt zwei Teile, die zusammenspielen, damit ein Neurales Netz `spe_ed` spielen kann. Die Trainings Komponente und die Taktik, die ein trainiertes Modell zur Bestimmung des nächsten Zuges ausnutzt. Unser Modell (Neurales Netz) ist wie folgt aufgebaut:

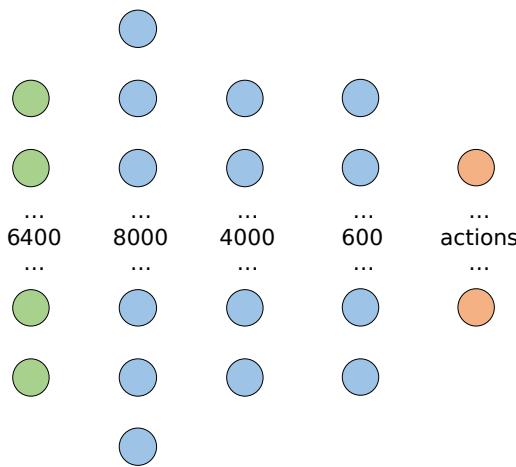


Abbildung 27: Unser DQN

Es werden Dense-Layer<sup>50</sup> verwendet, was bedeutet, dass jedes Neuron mit jedem Neuron des nächsten Layers verknüpft ist. Der Input-Layer hat  $80 \cdot 80^{51}$  Neuronen und der Output-Layer die Anzahl an verfügbaren Aktionen  $A$ . Das Training funktioniert durch einen Trainingsloop, welcher verschiedene Einstellungsmöglichkeiten bietet. Wir verwenden zum Trainieren folgende Werte der Hyperparameter:

<sup>49</sup><https://www.npmjs.com/package/ws-await>, zuletzt abgerufen am 17.01.2020

<sup>50</sup><https://deeplizard.com/learn/video/FK77zZxaBoI>, zuletzt abgerufen am 17.01.2020

<sup>51</sup>Max. Breite und Höhe, die bisher vorkam.

Größe replay memory	1000
batchSize	64
$\epsilon_0$	1
$\epsilon_T$	0,01
learning rate	0,01
$\gamma$	0,99

Am Anfang soll die Umgebung sehr stark erkundet werden ( $\epsilon_0$ ) und es soll am Ende noch die Chance für weitere Erkundungen geben ( $\epsilon_T$ ). Ähnlich beläuft es sich mit den Gewinnen:

one_step	1
player_died	10
win	50
death	-100

Der Agent soll auf keinen Fall sterben, das ist von höchster Priorität. Überleben wird jede Runde belohnt. Eliminiert er einen Spieler (es kann natürlich auch sein, dass Andere einfach so ausscheiden) gibt es einen extra Gewinn, damit der Agent gezielt Gegner eliminiert.

Der Trainings Loop funktioniert wie folgt.

---

#### Algorithm 6 train loop

---

```

1: Initialisieren des Replay Memorys mit 1000 State-Action-Reward-Done-nextState
   Tupeln
2: while true do
3:   Agent trainiert auf Replay Memory
4:   if Spiel zu ende then
5:     Berechnung des durchschnittlichen Gewinnes u.a.
6:     if Durchschnittlicher Gewinn > Bester Gewinn bisher then
7:       Neurales Netz abspeichern
8:     end if
9:   end if
10:  if Gespielte Runden % Jede so und so vielte Runde speichern == 0 then
11:    Gewichte vom policy Netzwerk ins target Netzwerk kopieren
12:  end if
13: end while

```

---

Im Monitoring werden alle verfügbaren Modelle geladen. Wenn neue Runden ankommen, wird das cells Array transformiert und das geladene Modell berechnet den nächsten (hoffentlich besten) Zug.

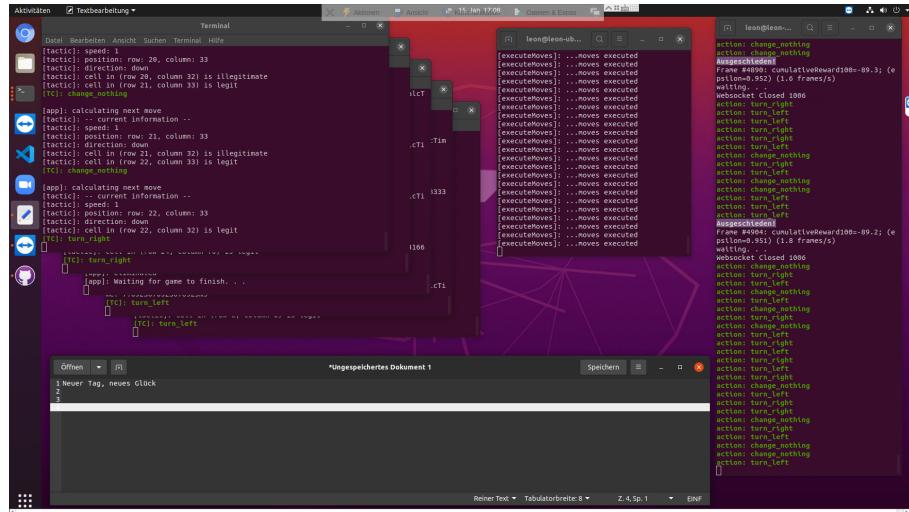


Abbildung 28: Das Modell wird trainiert (Links: 5 Instanzen unserer Taktiken, Mitte: Server, Rechts: Training)

#### 4.5.4 Optimierung

Dass in unserer WI Bibliothek nicht zu jeder Zeit einfach mehrere Werke kostenlos geliehen werden konnten, hat die Recherche in diesem Feld für uns erschwert. Unter der dadurch reduzierten Auswahl an hochwertigen Quellen und auch einer zeitlichen Komponente wurde die Nutzung der kompletten Macht von tensorflow.js und Neuralen Netzwerken beeinträchtigt.

Da die Spielzeit auf dem Spielserver deutlich zu hoch ist und durch das spezielle WebSocket Paket eine Protokollverletzung auftritt, kommt unser Testserver wie gerufen. Auf ihm können in tausenden Runden innerhalb von kurzer Zeit trainiert werden. Damit der Agent *speed\_up* und *slow\_down* einsetzen kann, muss die aktuelle Geschwindigkeit im Zustand repräsentiert sein. Dieses Problem erinnert daran, dass man bei einer Funktion an nur einem betrachteten Punkt nicht die Änderungsrate sieht. Dies kann man beheben, indem man anstatt nur den aktuellen *cells*, auch die *cells* von vorherigen Zuständen verwendet, also mehrere Frames in das Netz schickt.

Um dem Agenten Sprünge beizubringen, muss der Agent wissen, in welcher Runde er gerade ist, da man nur alle 6 Runden springen kann. Dies widerspricht wieder dem MDP, dass Entscheidungen ausschließlich nach dem aktuellem Zustand zu fällen sind. Man müsste also einen Wert *Runde* in dem Zustandstensor unterbringen o.Ä..

Zur Zeit nutzen wir Dense-Layer. Zur Performance- / Qualitätssteigerung wäre ein Conv-Layer evtl. besser geeignet, so wie es auch in der Implementierung von Googles Snake-DQN der Fall ist. Dieses erlaubt auch unterschiedlich große Eingaben, was bei einem Dense-Layer nicht möglich ist. Eine Einschränkung ist daher, dass in das Modell immer ein

80\*80 (max. Breite und Höhe, die bisher vorkam) Feld gespeist wird.

Da unsere Softwarelogik auf eine Server-Client Architektur setzt, dauert der Lernprozess sehr lange. Würde man das Spiel direkt in das Training implementieren, könnte man signifikant schneller lernen und der Einsatz von CUDA-Gpus würde sich lohnen. Zu einer besseren Softwarequalität würde beigetragen, wenn wir inputShapes<sup>52</sup> mit tiefstgehender Logik Verwendung gefunden hätten.

---

<sup>52</sup><https://js.tensorflow.org/api/latest/#sequential>, zuletzt abgerufen am 17.01.2020

## 5 Analyse

### 5.1 Vergleich und Diskussion

Die Erkenntnisse der folgenden Analysen haben wir unter anderem mit Hilfe unserer Analyse Methoden gewonnen. Diese werden in den folgenden Kapiteln, im Anschluss zum Vergleichs, detailliert beschrieben.

Nachdem die Funktionsweise der Taktiken erklärt wurde, werden sie jetzt diskutiert und bewertet. Erst werden wir uns mit der Eignung der verschiedenen Taktiken beschäftigen. Dafür stellt man die einzelnen Stärken und Schwächen gegenüber. Die Auswertung beruht auf theoretischen Überlegungen und einer empirischen Auswertung durch gespielte Spiele<sup>53</sup>.

Folgend wird die Notation aus dem viertel Kapitel verwendet:

Auswahloptionen:  $o = \{o_1, o_2, o_3\} = \{\text{change\_nothing}, \text{turn\_right}, \text{turn\_left}\}$

#### **exampleTactic**

Example Tactic hat keine strategische Relevanz. Da sie zum Testen von Features dient, ist sie für kompetitives Spielen nicht geeignet. Dies ist aber kein Nachteil, da es nicht der Intention der Taktik entspricht, ein kompetitives Spielen zu ermöglichen. Somit wird diese auch nicht, abseits Entwicklungszwecken, für Spiele eingesetzt.

#### **dontHit**

DontHit schließt ebenfalls die Varianten dontHitSemiRandom und DontHitRandom mit, ein. Diese Varianten müssen auf ihre individuellen Vorteile untersucht werden. Da dontHit und dontHitSemiRandom sehr ähnlich funktionieren, ist es besonders sinnvoll, diese zunächst auf die gemeinsamen Vorteile und Nachteile zu analysieren und zu vergleichen. Ein gemeinsamer Vorteil ist, dass die Algorithmen relativ stabil arbeiten.

Das bedeutet, dass der Agent in einer gegebenen Fläche zuverlässig manövrieren kann. Dabei scheidet er nicht aufgrund von eigenen Fehlern aus. Das bedeutet, dass der Algorithmus Kollisionen abfängt und durch die Prioritäten eine feste Vorschrift befolgt. Kombiniert mit einer freien eingerahmten Fläche kann der Agent diese sehr gut ausnutzen. Je nach der ausgewählten Prioritätenliste entstehen verschiedene Muster. Bei  $p = \langle 1, 2, 3 \rangle$ , das entspricht den, nach ihrer Priorität aufsteigend sortierten, Auswahloptionen (*change\_nothing*, *turn\_right*, *turn\_left*), entsteht ein Rechteck.

Die Sicht des Algorithmus ist sehr beschränkt. Es wird nur das nächste Feld überprüft, um nicht durch eine Kollision auszuscheiden. Wenn ein Hindernis unglücklich platziert ist, wird das eigene mögliche Spielfeld erheblich verkleinert und die Chancen auf

---

<sup>53</sup>vgl. Kapitel 5.6

einen Sieg sinken drastisch. Das ist für die meisten Spiele nicht praktisch. Als Resultat lässt sich sagen, dass der dontHit-Algorithmus bei weitem nicht optimal ist. Die Nachteile sind zu groß, als dass man ihn dauerhaft zum Spielen nutzen könnte.

Nun wird dontHit mit den sortierten Prioritäten (*change\_nothing*, *turn\_right*, *turn\_left*) betrachtet. Stellen wir uns vor, dass unser Agent eingesperrt ist und sich am Rand befindet. Der dontHit-Algorithmus würde die gegebene Fläche optimal nutzen. Da dieser linear arbeitet und das Polygon von außen abläuft, werden alle Zellen genutzt, denn wir befinden uns in einem eingesperrten Feld. Des Weiteren arbeitet die Taktik mit der minimalen Geschwindigkeit. Da unser Agent gefahrlos sein Polygon füllen kann, zieht er das Spiel zu unserem Vorteil in die maximale Länge. Sollte das eingekreiste Polygon vom Agenten groß genug sein, dann werden die Gegner es mit jedem weiteren Zug schwieriger haben eine freie Zelle zu finden. Eine mögliche Gefahr sind Gegner, die in unser erobertes Feld springen. Die Wahrscheinlichkeit, dass dieser Fall eintritt, sinkt jedoch mit jedem Zug, da unserer Agent stets von Außen nach Innen die Dicke des Polygons erhöht. Wir halten im Hinterkopf, dass es dieses optimale Szenario gibt. Es kann in anderen Taktiken genutzt werden.

### **dontHitRandom und dontHitSemeRandom**

Um dem Effekt des sich Einsperrens entgegen zu wirken wurden die randomisierten Varianten dontHitRandom und dontHitSemiRandom entwickelt. Zunächst wird die Qualität der ersten Variante besprochen.

Da dontHitRandom völlig randomisiert entscheidet erzeugt die Taktik sehr oft Situationen, die unseren Agenten direkt ausscheiden lassen. Der Algorithmus arbeitet nicht vorhersehbar und folgt dadurch, im Vergleich zu dontHit oder dontHitSemiRandom, keinen festen Routen. Die Taktik lässt den Agenten, unabhängig von den Aktionen der Gegner, ausscheiden. Das zeigt auch die empirische Analyse. Wir haben viele Runden, in denen wir gute Chancen hatten, verloren und konnten in keinem einzigen Spiel über 200 Spielzüge erreichen. Dennoch erkennt man Vorteile der randomisierten Herangehensweise.

Die semi-randomisierte Variante des dontHit-Algorithmus kombiniert die Vorteile der anderen Varianten. Der Agent gewinnt an Stabilität und variiert die Abbiegevorgänge. Als Ergebnis ist der Agent in der Lage, deutlich länger im Spiel zu überleben und es werden häufiger Spiele gewonnen. Die Auswertung erfolgte in einer frühen Phase des Wettbewerbs. Daher gibt es verhältnismäßig viele Siege.

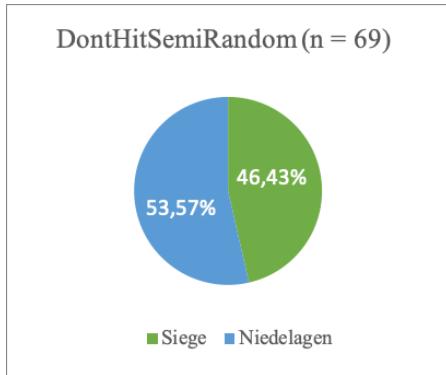


Abbildung 29: Auswertung der Taktik: dontHitSemiRandom

Die folgende Grafik<sup>30</sup> veranschaulicht die jeweiligen Vorteile der Implementierungen. DontHit nutzt den Platz effizienter als dontHitRandom. Jedoch biegt er immer wenn möglich, seiner Priorisierung nach, ab. DontHitSemiRandom biegt zufällig ab. Im Bild steuert er dadurch eine große und freie Fläche an.

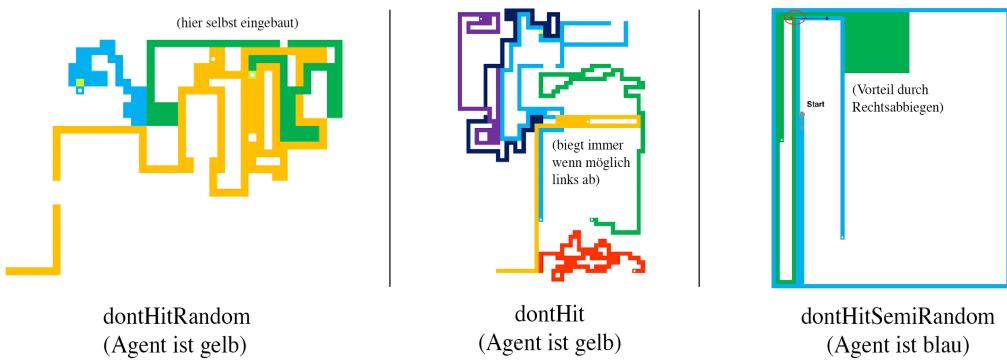


Abbildung 30: Vergleich der dontHit Varianten

### **recursiveBnB**

Die beschränkte Sichtweise ist der Hauptkritikpunkt der dontHit Strategie. RecursiveBnB versucht dieses Problem zu lösen. Wie in Kapitel 4.3 beschrieben, werden alle erreichbaren Spielzüge simuliert und ausgewertet, sofern die Rechenzeit dies zulässt. Die einzelnen Auswahloptionen lassen sich ebenfalls priorisieren. Lässt man recursiveBnB mit dontHit-ähnlichen Priorisierungen spielen, erhält man einen ähnlichen Verlauf. Der rekursive Ansatz verhindert aber, dass unser Spieler in offensichtliche Sackgassen navigiert. Somit ist recursiveBnB der klassischen dontHit-Strategie überlegen. recursiveBnB vereint das Beste aus den beiden Welten. Wir übernehmen die effiziente Platznutzung vom dontHit Algorithmus und erweitern die Strategie um eine Komponente. Aufgrund der hohen Suchtiefe werden extrem viele mögliche Züge analysiert. Das wirkt sich drastisch auf den Erfolg der Taktik aus. Beim Testen bemerkt man, dass der Agent oft Züge wählt, die

nicht offensichtlich sind aber unseren Agenten aus einer kritischen Situation retten. Beim automatisierten Testen lassen sich mit dieser Taktik deutlich mehr Siege beobachten als mit den vorherigen Taktiken. Wenn solche, für uns nicht intuitive, Züge gewählt werden, offenbart sich das Potential dieser Taktik. In diesen Fällen ist recursiveBnB besser als es ein Mensch sein könnte. Fragwürdige Züge sind oft Vorbereitungen auf Sprünge und planen das korrekte Abspringen mit der richtigen Geschwindigkeit zu einer Runde, in der auch gesprungen werden kann. Diese Leistung kann ein Mensch nicht innerhalb der Deadline in dieser Quantität (Anzahl der geprüften Züge) erbringen. Man kann die Priorisierung anpassen oder randomisiert durchführen. Das hat den Effekt, dass die Routen nicht mehr vorhersehbar sind. Die dontHit-ähnlichen Priorisierungen haben sich jedoch in unseren Tests als besonders erfolgreich erwiesen. Wir konnten mehr als jedes Dritte Spiel gewinnen.

Da bei jedem Aufruf mögliche Spielzüge simuliert werden, kommt es in kritischen Fällen zu einer erhöhten Rechenzeit. Dies stellt aber kein Problem dar, denn ein Time-Out wird in der Implementierung abgefangen. In diesen Situationen rechnet das Programm so viele Situationen wie möglich aus und wertet diese aus.

Ein entscheidender Vorteil, der von den anderen Taktiken gänzlich ignoriert wurde, ist das Springen. Das Programm erkennt in der Simulation mögliche Sprünge, indem es die Rundenzahl auswertet. Mit diesem Feature können wir mit dem Agenten aus Sackgassen ausbrechen, die zuvor als unlösbares Problem galten. RecursiveBnB eignet sich zum ständigen Einsatz und liefert gute Ergebnisse.

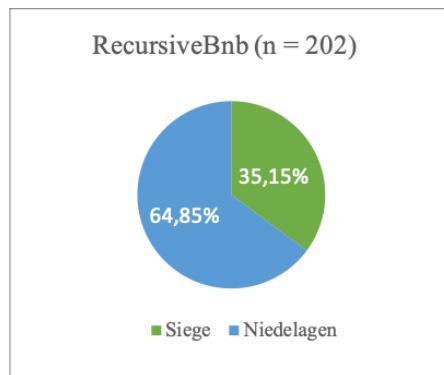


Abbildung 31: Auswertung der Taktik: RecursiveBnb

## 5.2 Laufzeitanalyse

Im Folgenden werden die Laufzeiten sowie Komplexität der einzelnen Taktiken untersucht. Dabei werden Laufzeiten gegebenenfalls in Abhängigkeit von Parametern betrachtet. Laufzeiten spielen eine besondere Rolle, da Berechnungen der Spielzüge nicht nur generell in realistischer Zeit erfolgen müssen, sondern zusätzlich innerhalb der vorgegebenen Deadline. Etwas weniger genau untersuchen wir die Speicherkomplexität unserer Taktiken. Denn im Gegensatz zur Zeit hat sich der Speicher bei keiner unserer Taktiken als

Bottleneck<sup>54</sup> erwiesen.

Die Komplexität von Algorithmen ist eine der Kriterien ihrer Analyse. Arbeitet ein Algorithmus effizient und ist von geringer Komplexität und führt seine Aufgabe schnell aus. Die Laufzeit von Algorithmen ist jedoch abhängig von der Leistung des Rechners und Faktoren wie dem Betriebssystem oder dem Compiler.

Da das exakte Zählen der elementaren Operationen oft zu aufwändig ist, muss geschätzt werden. Dies geschieht durch eine Konstante und wird asymptotische Komplexitätsanalyse genannt. Asymptotisch bedeutet hierbei, sich im mathematischen Sinne einer Linie immer wieder anzunähern. Die Analyse hängt von einem Faktor ab: der Eingabegröße  $n$ . Betrachtet wird innerhalb der einzelnen Taktiken jeweils die `nextMove()` Methode. Neben dem Pseudocode werden die Häufigkeiten der Ausführungen der jeweiligen Zeilen als Kommentar notiert, sodass sich über das Aggregieren dieser Häufigkeiten die Komplexitätsklasse angeben lässt. Hierbei wird bei Methodenaufrufen deren Komplexität angegeben, welche auf gleiche Weise bestimmt wurde.

Wir definieren als Komplexitätsklasse<sup>55</sup>  $\Theta$  (nach Schema der gegebenen Quellen) einer Funktion  $g(n)$ , alle Funktionen  $f(n)$ , die ab einem Startwert  $n_0$  zwischen zwei Varianten von  $g(n)$  (multipliziert mit zwei Faktoren  $c > 0$ ) liegen. Ist eine Methode bspw. von quadratischer Laufzeit und hat einige konstante Zeilenaufruufe, fallen diese ab einem bestimmten  $n_0$  nicht mehr ins Gewicht. Als Klasse wird demnach angegeben, was die Höhe der Laufzeit maßgeblich bestimmt. Aufgrund der niedrigen Aussagekraft der gewählten Parameter wird im folgenden die Wahl der geeigneten  $n_0$  und  $c$  nicht angegeben. Formal gilt jedoch für jede der von uns bestimmten Komplexitätsklassen für Methoden:

$$\Theta(g(n)) = f(n) \mid \exists c_1 > 0, c_2 > 0, n_0 \in \mathbb{N} \quad (1)$$

$$\forall n \geq n_0 : 0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$$

### Laufzeitanalyse von `dontHit`

`DontHit` ist konstant in Speicher und Laufzeit. Die For-Schleife iteriert über  $i$  von 0 bis 2 und terminiert nach maximal 3 Durchläufen, da  $i$  in jedem Lauf erhöht wird.

---

#### Algorithm 7 dontHit $\Theta(1), S(1)$

---

```

1: for  $i$  in  $\{1, \dots, |p|\}$  do                                ▷ konstant 3 Iterationen
2:   if überprüfe( $o_i$  mit Priorität  $p_i = 4 - i$ ) then    ▷ 1
3:     return  $o_i$                                          ▷ 1
4:   end if
5: end for

```

---

<sup>54</sup><https://kanbanize.com/lean-management/pull/what-is-bottleneck>, zuletzt abgerufen am 17.01.2020

<sup>55</sup>[Cormen et al., 2009, Seite 45-52]

Die Abwandlungen dontHitRandom und dontHitSemiRandom unterscheiden sich auch nicht in Laufzeit und Speicher. Der zugehörige Pseudocode wird für beide Varianten abstrahiert formuliert.

---

**Algorithm 8** dontHitVariants  $\Theta(1), S(1)$ 


---

```

1: p = shuffleArray([0,1,2])                                ▷ (Methodenaufruf insgesamt) 1
2: dh = new dontHit()                                         ▷ (Konstruktoraufruf insgesamt) 1
3: return dh.nextMove()                                       ▷ (Methodenaufruf insgesamt) 1

```

---

Der dontHit Konstruktor hat ausschließlich eine konstante Anzahl an Wertzuweisungen. Der Fisher-Yates-Shuffle-Algorithmus hat eine Komplexität von  $\Theta(n)$ , wobei  $n$  die hier konstante (3) Anzahl an Listenelementen ist.

---

**Algorithm 9** shuffleArray(array)  $\Theta(n), S(1)$ 


---

```

1: n = array.length                                         ▷ 1
2: while n > 0 do                                         ▷ n
3:   i = random remaining array index                      ▷ 1
4:   t = array[n]                                           ▷ 1
5:   array[n] = array[i]                                     ▷ 1
6:   array[i] = t                                           ▷ 1
7:   n - -                                                 ▷ 1
8: end while

```

---

### Laufzeitanalyse recursiveBnB

Die Komplexität von recursiveBnB wird maßgeblich durch die Rekursion bestimmt. Sie wird in Abhängigkeit zur variablen Rekursionstiefe betrachtet. Laufzeiten der Methoden, die innerhalb nextMove() aufgerufen werden, sind konstant. Um das zu zeigen, werden aufgerufene Methoden ebenfalls auf ihre Laufzeit untersucht. Die bei dieser Taktik relevante Rechnung erfolgt in der Methode calcMoveRec(), die in nextMove() aufgerufen wird. Sie bestimmt die Laufzeit der Taktik.

---

**Algorithm 10** calcMoveRec(gameData)  $\Theta(5^n), S(5^n)$ 

---

```
1: order = getRandomOrder()                                ▷ (Methodenaufruf insgesamt) 1
2: if counter ≥ depth then                                ▷ 1
3:   return chosenMove                                    ▷ 1
4: end if
5: for all move in order do                                ▷ 5
6:   if possible.move then                                ▷ (Methodenaufruf insgesamt) speed
7:     if counter = 0 then                                ▷ 1
8:       chosenMove = move                               ▷ 1
9:     end if
10:    simulierteDaten = simulateDecision(gameData, palyer, move)
11:    ▷ (Methodenaufruf insgesamt) speed
12:    counter++                                         ▷ 1
13:    calcMoveRec(simulierteDaten)                      ▷  $5^n$ 
14:  end if
15: end for
16: if counter ≥ Tiefe then                                ▷ 1
17:   return chosenMove                                    ▷ 1
18: end if
```

---

Die Methoden possible.move() sind für jede Zugmöglichkeit implementiert und unterscheiden sich nicht im Aufbau. Aus dem (auf alle Methoden possible generalisierten) Pseudocode ergibt sich, dass possible linear von der Geschwindigkeit abhängig ist. Diese ist maximal 10. Bei dieser Methode kann es hilfreich sein, den echten Code mit ausführlichen Kommentaren zu lesen (befindet sich im Ordner recursiveBnB). Ebenfalls wird die Methode simulateDecision() aufgerufen. Auch sie ist linear abhängig von der Geschwindigkeit.

---

**Algorithm 11** possible.move(gameData, playerID, round)  $\Theta(speed), S(1)$ 

---

```
1: headRow = player.y                                ▷ 1
2: headColumn = player.x                             ▷ 1
3: speedCounter = jump = 0                           ▷ 1
4: if round mod 6 = 0 and speed > 3 then          ▷ 1
5:   jump = speed-2                                 ▷ 1
6: end if
7: for 1..speed do                                ▷ speed
8:   switch(direction)                            ▷ 1
9:   case "up":                                    ▷ 1
10:  if isFieldLegit(...) then                  ▷ (Methodenaufruf insgesamt) 1
11:    speedCounter++                            ▷ 1
12:  end if
13:  case "down": ...                            ▷ (alle der 4 cases) 1
14:  i = i + jump                                ▷ 1
15:  speedCounter = speedCounter + jump           ▷ 1
16: end for
17: if speedCounter >= speed then                ▷ 1
18:   return true                                ▷ 1
19: else
20:   return false                               ▷ 1
21: end if
```

---

---

**Algorithm 12** isFieldLegit(gameData, x, y)  $\Theta(1), S(1)$ 

---

```
1: if x,y inside cells and cells[y][x] = 0 then      ▷ 1
2:   return true                                ▷ 1
3: else
4:   return false                               ▷ 1
5: end if
```

---

---

**Algorithm 13** simulateDecision(gameData, player, move)  $\Theta(speed), S(1)$ 


---

```

1: deep clone gameData                                ▷ 1
2: switch (decision)                                 ▷ 1
3:   case "speed_up":                               ▷ 1
4:     speed++                                     ▷ 1
5:     switch(direction)                           ▷ 1
6:       case "up":                                ▷ 1
7:         for 1..speed+1 do                      ▷ speed
8:           cells[...,...] = palyerID            ▷ 1
9:         end for
10:        case ...                                ▷ (alle der 4 cases) 1
11:        case ...                                ▷ (alle der 5 cases) 1
12: round++                                         ▷ 1
13: return gameData                                  ▷ 1

```

---

Best- und Worst Case der Rekursion wurden bereits in 4.3.2 als Funktion aufgestellt. Dabei ist der Best Case eine lineare Abhängigkeit von der Rekursionstiefe  $n$ , also  $\Theta(n)$ . Der Worst Case wurde mit der geometrischen Reihe in eine geschlossene Form überführt. Asymptotisch können Faktoren und Konstanten hier vernachlässigt werden. Zur Verdeutlichung wird der Ausdruck umgeformt und anschließend dessen Konstanten und Faktoren gestrichen. Es bleibt eine exponentielle Laufzeit von  $\Theta(5^n)$ .

$$\frac{1 - 5^{n+1}}{1 - 5} - 1 = \frac{1}{4} \cdot 5^{n+1} - \frac{5}{4}$$

Die Worst Case Laufzeit ist innerhalb der Deadline inakzeptabel. Aus der nachfolgenden Analyse ergibt sich, dass der Testrechner in einer Sekunde durchschnittlich 9924 Rekursionsaufrufe erreicht. Bei einer Tiefe von  $n = 30$  entspricht das einer Worst Case Rechenzeit von über 3,7 Milliarden Jahren. Es stellt sich die Frage, wie der Average Case aussieht. Da er von den Bounds (Kürzungen der Rekursionspfade) abhängt und diese wiederum vom Verhalten der anderen Spieler, liefert eine Messung der Durchschnitte über viele Runden eine empirische Antwort. In mehreren Spielen wurden 1059 Runden gemessen. Die durchschnittliche Anzahl an Rekursionsaufrufen bei einer Tiefe von  $n = 30$  ist 3337. Im Durchschnitt wurden pro Runde 13263,8 Bounds genutzt und 0,3 Sekunden gerechnet. Der Worst Case ist davon weit entfernt. Die Berechnungen und Ausgabe der durchschnittlichen Werte sind auskommentiert, können aber im Code von recursiveBnB nachvollzogen werden. In der folgenden Tabelle sind die gemessenen Spiele, ihre Rundenanzahl, die durchschnittliche Anzahl an Rekursionsaufrufen pro Runde, die durchschnittliche Anzahl von Bounds pro Runde, die durchschnittliche Rechenzeit pro Runde, die durchschnittliche Anzahl an Aufrufen pro Sekunde und die durchschnittliche Anzahl an Bounds pro Rekursionsaufruf dargestellt. Da rechenintensive Züge selten am Anfang der Runden stattfinden, sind ausschließlich Spiele mit einer Rundenanzahl ab 100 betrachtet worden.

<b>Spiel</b>	<b>Runden</b>	<b>Aufrufe</b>	<b>Bounds</b>	<b>Zeit [ms]</b>	<b>Aufrufe/s</b>	<b>Bounds/Aufruf</b>
1	117	4824	19274	364,3	13241,8	3,995
2	228	2443,2	9677,6	328,6	7435,2	3,961
3	100	51,3	109,7	9,7	5288,7	2,138
4	179	6443,5	25690,4	494,9	13019,8	3,987
5	130	5900,4	23510,9	328,4	17967,1	3,985
6	150	80	223,2	12,5	6400,0	2,790
7	155	3616,9	14361	591,4	6115,8	3,971
Durchschnitt	151,3	3337	13263,8	304,3	9924,1	3,547

### Laufzeitanalyse von dangerFields

Die Taktik dangerFields ist abhängig von der Spielfeldgröße, mit einer Breite von  $m$  und einer Höhe von  $n$ , und hat eine Komplexitätsklasse von  $\Theta(mn \cdot \log(mn))$ . Diese wird asymptotisch bestimmt vom A\*-Algorithmus, dessen Laufzeit nach Grinsteads optimierter Form<sup>33</sup>  $O(mn \cdot \log(mn))$  ist. Da die nextMove() Methode direkt und indirekt zwölf verschiedene Methoden aufruft, die eine niedrigere Obere Schranke in ihrer Laufzeit haben, werden hier nur ihre Komplexitätsklassen genannt. Die Methoden mit Laufzeit  $\Theta(mn)$  traversieren in verschiedenen Formen einen großen Teil des Spielfelds, um ihn beispielsweise mit Werten zu füllen. So kommt es, dass in nextMove() mindestens neun mal (vier davon aufgrund des Floodfill<sup>30</sup>) über das nahezu ganze Spielfeld iteriert wird. Verglichen mit dem Faktor von  $\log(mn)$ , der bei einem Feld von bspw. 100x100 4 ist, fällt der A\*-Algorithmus weniger ins Gewicht als die Summe der restlichen Methoden.

- astar():  $\Theta(mn \cdot \log(mn))$
- amIAalone():  $\Theta(mn)$
- amIAaloneFill():  $\Theta(mn)$
- alternativeTactic():  $\Theta(1)$
- transform():  $\Theta(mn)$
- heat():  $\Theta(mn)$
- markCircles():  $\Theta(mn)$
- clamp():  $\Theta(1)$
- findGoal():  $\Theta(mn)$
- findGoalFloodFill():  $\Theta(mn)$
- findGoalFill():  $\Theta(mn)$
- moveDependingOnDirection():  $\Theta(1)$

### **5.3 Benutzeroberfläche**

In den vorhergehenden Teilen dieser theoretischen Arbeit war oft die Rede von Spielen. Wir haben sehr viele Spiele gespielt, Taktiken entwickelt und Spiele ausgewertet. Es gibt jedoch keine offizielle Spieloberfläche, welche man sich anschauen kann. Somit wird einem schnell klar, dass man es ausschließlich mit einer riesigen Datenmenge an JSON-Objekten zu tun hat, ohne eine Visualisierung.

Alle Informationen, die wir verarbeiten, basieren auf diesen JSON-Objekten. Jede Runde wird serverseitig nur mit einem JSON-Objekt beschrieben. Würde es sich um ein kleines Spielfeld handeln und wäre die Rundenanzahl extrem gering, so könnte man eventuell den Spielverlauf anhand den JSON-Objekten nachvollziehen. Elegant wäre dieses Vorgehen definitiv nicht, Spaß würde es uns Entwicklern noch weniger machen. Wenn es bereits den Entwicklern keinen Spaß macht, dann wird es mit an Sicherheit grenzender Wahrscheinlichkeit den Nutzern ebenfalls keinen Spaß machen. Da die Softwarenutzung für den Endnutzer angenehm sein soll und wir als Team möglichst effizient arbeiten sowie komfortabel die Software testen wollen, haben wir uns entschieden eine detaillierte Benutzeroberfläche zu entwickeln.

Zu Beginn gab es verständlicherweise die Überlegungen, dass wir die Ausgabekonsole nutzen, könnten um uns die wichtigen Informationen anzeigen zu lassen. Diese Idee haben wir allerdings recht schnell verworfen, da sie unkomfortabel ist und sich ein Spielfeld dort schwer visualisieren. Außerdem gibt es eine begrenzte Möglichkeit als Nutzer mit dem Programm zu interagieren. Daher fiel die Wahl auf eine Web-Benutzeroberfläche. Diese hat mehrere Vorteile. Sie ist vollkommen individualisierbar, sieht ansprechend aus und bietet Möglichkeit zur Interaktion. Des Weiteren hat man, im Vergleich zu einer Konsole, deutlich mehr Visualisierungsmöglichkeiten. Es bringt jedoch auch Herausforderungen mit sich, da wir Zeit einplanen mussten, um die Oberfläche zu programmieren. Zeit haben wir bereits als ein kritisches Gut identifiziert. Nichtsdestotrotz haben wir uns für die Weboberfläche entschieden und so viel Zeit dafür zu investieren. Dies haben wir durch den möglichen Produktivitätsvorteilen begründet. Wir sehen große Vorteile darin, auf eine Infrastruktur zurückgreifen zu können, die uns ein gezieltes und komfortables Testen ermöglicht. Diese Infrastruktur bringt Werkzeuge mit, die uns das mühselige Auswerten der JSON-Objekten erspart hat.

Zunächst legten wir fest, welche Features unsere Benutzeroberfläche ermöglichen soll. Wir können zwischen den essenziellen Aufgaben und den sogenannten, nicht zwingend notwendigen, Werkzeugen unterschieden: Die wichtigste Aufgabe ist das korrekte Wiedergeben eines Spiels. Dies geschieht durch ein großes Spielfeld in der Mitte der Benutzeroberfläche. Diese Funktion ist für sämtliche Stakeholder, seien es die Entwickler, Nutzer oder Juroren, von großer Bedeutung. Man möchte schließlich immer dem Spiel folgen können. Des Weiteren möchte man ungern die verschiedenen Taktiken über die Konsole starten. Gleicher gilt für das Starten der Runde. Zudem sieht der Nutzer zu den verschiedenen Taktiken stets eine Beschreibung.

Das Design der Oberfläche wurde konsistent und benutzerfreundlich gehalten, damit sich auch Nutzer ohne besondere Kenntnisse gut zurechtfinden können. Für den Nutzer sind alle wichtigen Informationen dauerhaft ersichtlich und er kann auch, ohne die Regeln von Spe\_ed zu kennen, dem Spielgeschehen folgen. Der Nutzer wird auf wichtige Ereignisse durch Meldungen und Geräusche aufmerksam gemacht. Aus einer Systemmeldung kann der Nutzer direkt ablesen, woher sie Meldung kommt.



Abbildung 32: exemplarische Systemmeldungen der Benutzeroberfläche

Wir haben folgende Werkzeuge in erster Linie für uns selbst in die Oberfläche eingebaut, damit wir möglichst produktiv arbeiten können. Sie sind aber auch für die Endnutzer von Vorteil, da sie das Verständnis fördern.

- **Spielerübersicht:**

Oben links in der Oberfläche sieht man eine Liste der Spieler der aktuellen Runde. Diese sind nummeriert und geben an, ob ein Spieler aktiv(*active*) oder ausgeschieden(*out*) ist. Zum Ende einer Runde werden die temporären Namen bekannt gegeben und direkt auf der Benutzeroberfläche angezeigt. Man kann schnell den Sieger ausfindig machen, da dieser alleinig aktiv ist. Jeder Spieler erhält außerdem eine mit den Spuren konsistente Farbe und neben dem eigenen Spieler gibt es zusätzlich einen *you*-Vermerk. Man möchte schließlich direkt sehen können, welcher Spieler man selber ist, denn die Farben können spielerabhängig variieren.

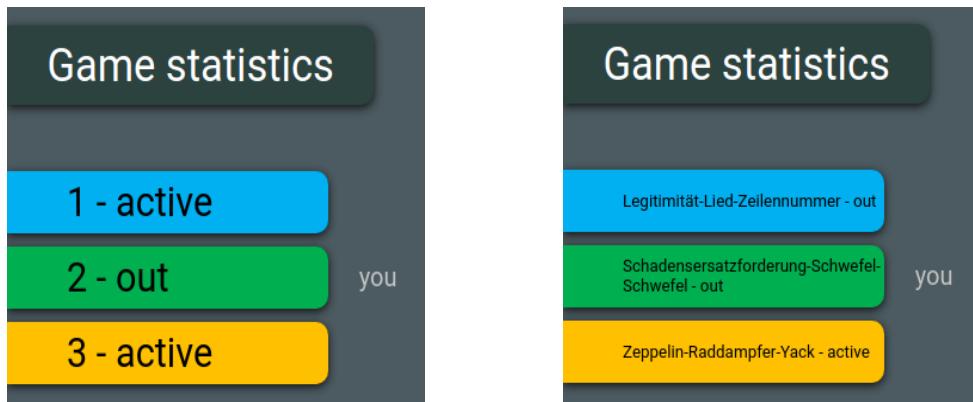


Abbildung 33: Spielerübersicht der Benutzeroberfläche; mit und ohne Anzeige der Teamnamen

- **Persönliche Statistiken**

Für uns als Entwickler gilt das besondere Interesse unserem eigenen Spieler. Um Taktiken auswerten zu können, braucht man ein breites Spektrum an Informationen. Wichtig ist unter anderem der zuletzt ausgewählte Schritt, da man die eigene Taktik beim Testen auf Korrektheit verifizieren möchte. Wenn ein Spielfeld dicht besiedelt ist, kann man nicht immer leicht den letzten Zug ablesen. Außerdem möchte man sich solche Informationen ungern selbst merken. Daher gibt es unter den persönlichen Statistiken eine kleine Konsole, die gespielte Spielzüge anzeigt. Dies hat sich als äußerst hilfreich erwiesen, da man auch Geschwindigkeitserhöhungen und Muster leichter erkennen kann.

Wenn es um komplexere Taktiken geht, spielt auch die Zeit pro Zug eine Rolle. Der Entwickler muss stets wissen, wie lange das Programm für einzelne Spielzüge rechnen muss. Einerseits will man sichergehen, dass das Programm für vermeintlich leichte Züge nicht zu viel Zeit benötigt, andererseits braucht man die Zeit auch um komplexe Methoden auf Eignung zu überprüfen. Ersteres lässt sich ohne Zeitangabe nicht direkt erkennen, da Zeitverzögerungen im Spiel auch von den anderen Gegnern ausgelöst werden können. Dafür haben wir einen Graphen in die Oberfläche eingebunden. Wenn man den Mauszeiger über Stellen des Graphs bewegt, wird einem die Rechenzeit dieser jeweiligen Runde angezeigt. Die Zeit der aktuellen Runde wird in einem gesonderten Fenster neben dem Graph angezeigt.

Wie in der Analyse der Taktiken beschrieben, gibt es Taktiken, die besonders stabil sind. Diese laufen zuverlässig in Spielen, die aus sehr vielen Taktiken bestehen. Um diese gut zu erkennen, möchte man als Entwickler jederzeit sehen können in welcher Runde man sich aktuell befindet. Dies wird ebenfalls in den eigenen Statistiken angezeigt.



Abbildung 34: persönliche Statistik der Benutzeroberfläche

- **Kontrollzentrum**

Die rechte Seite der Benutzeroberfläche ist das Kontrollzentrum.

Um ein Spiel zu starten muss der Nutzer eine Taktik auswählen. Diese kann er auf der rechten Seite in einer Dropdown-Liste auswählen. Für die Taktiken, die Parameter benötigen, kann man diese hier direkt einstellen. Für RecursiveBnB muss man zum Beispiel die Suchtiefe einstellen, dangerFields benötigt die Information, ob dontHit mitbenutzt werden soll, und bei dontHit kann man die Prioritätenliste editieren.

Für dangerFields gibt es zusätzliche Visualisierungsoptionen. Diese kann man im Kontrollzentrum aktivieren. Der Nutzer kann auswählen, ob er die Heatmap, den Weg, das Ziel und die Zielfelder sehen möchte, die für den dangerFields-Algorithmus eine Rolle spielen. In der folgenden Grafik wird das Setzen der notwendigen Parameter und die Aktivierung sonstiger Visualisierungsoptionen exemplarisch für DontHit und DangerFields illustriert.

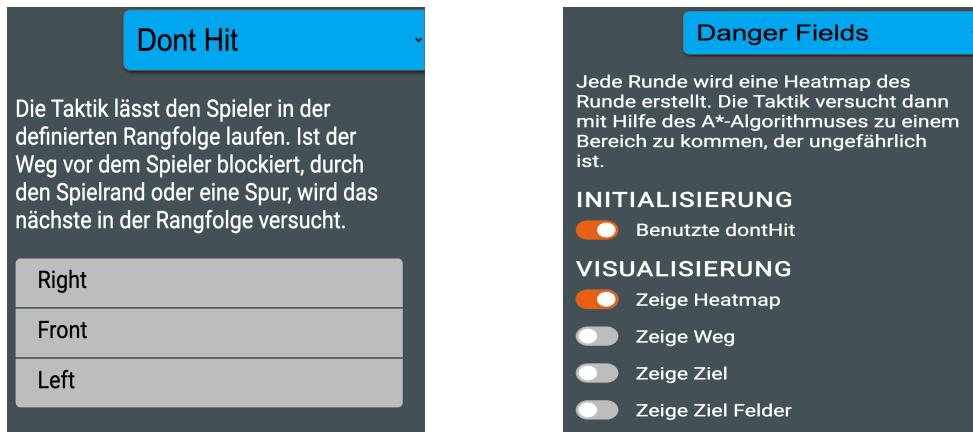


Abbildung 35: Setzen der Parameter einer Taktik in der Benutzeroberfläche

Wenn man das Spiel initiieren möchte, drückt man auf *Start Queue*. Der Spieler wartet nun bis ein Spiel vom Gameserver gestartet wird. Beim Start ertönt ein Hinweiston.

Möchte man eine Taktik dauerhaft spielen lassen, aktiviert man das AutoPlay-Feature. Dafür muss man die Checkbox neben dem *Start Queue* Knopf aktivieren. Durch das setzen dieser Option betritt unser Agent die Queue, um ein neues Spiel zu spielen.

Unten rechts befindet sich ein Knopf, den man drücken kann um die History komfortabel zu betreten. In der History kann man sich vergangene Spiele anschauen. Diese wird im folgenden Unterkapitel genauer erklärt.

Abschließend können wir definitiv reflektieren, dass der hohe Aufwand der Entwicklung dieser Benutzeroberfläche eindeutig im Verhältnis zum Nutzen steht. Die Oberfläche hat sich positiv auf unsere Produktivität und die Ästhetik der Software ausgewirkt. In der folgenden Grafik kann man die Benutzeroberfläche zusammenfassend erneut mit allen Elementen in Aktion betrachten. Dank der Benutzeroberfläche kann man augenblicklich die wichtigsten Informationen erkennen. In dem Fall haben somit vier Spieler gespielt, wir waren der grüne Spieler und haben das Spiel nach 332 Runden für uns entschieden. In dem Spiel haben wir die Taktik recursiveBnB mit einer Suchtiefe von 30 benutzt.

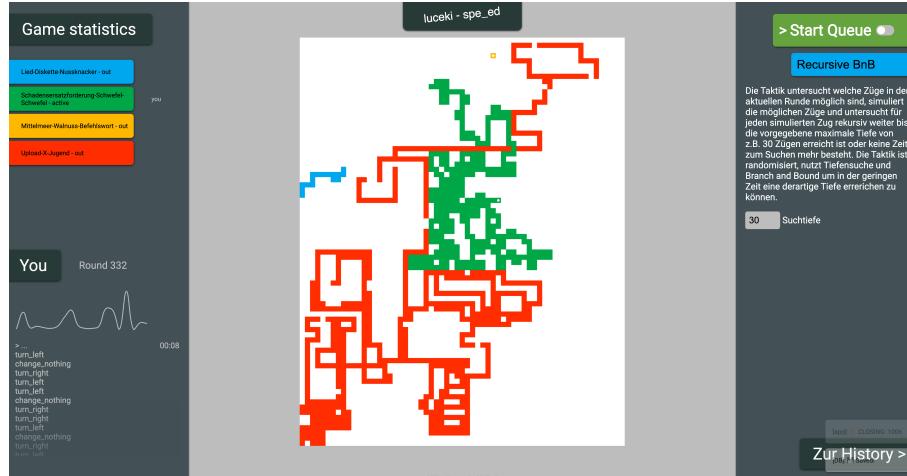


Abbildung 36: Benutzeroberfläche

## 5.4 Datenbank

Wie im vorherigen Kapitel kurz besprochen, haben wir eine Datenbank entwickelt um unsere Spiele speichern zu können. Dieses Feature haben wir History genannt. Eine datenbank-gestützte History bietet den verschiedenen Nutzergruppen mehrere Vorteile. Als Nutzer hat man somit stets die Möglichkeit sich vergangene Spiele anzuschauen. Man kann sich interessante Spieler erneut anschauen und als Entwickler kann man Spiele analysieren. Diese Möglichkeit haben wir ausgiebig genutzt. Im Folgenden stellen wir unsere History und die dahinterliegende Datenbank im Sinne der Analysemöglichkeiten vor.

### History - Weboberfläche

Die Oberfläche der History kann man einerseits durch den gleichnamigen Button im Monitoring aufrufen, andererseits kann man sie manuell über <https://localhost/history> öffnen. Ähnlich zur Benutzeroberfläche vom Monitoring, gibt es auf der linken Seite die Statistiken und in der Mitte ein Spielfeld. Auf der rechten Seite befinden sich die Einstellungen der History. Unten kann man sich das Spiel aussuchen, das man erneut betrachten möchte. Die Einträge sind nach dem Erstellungszeitpunkt absteigend sortiert und demnach benannt.



Abbildung 37: Spieldaten in der History

Nachdem der Nutzer ein Spiel auswählt, wird dieses geladen und oben in den Einstellungen erscheint die Rundensteuerung. In dieser wird die Taktik der ausgewählten Runde angezeigt. Darunter befindet sich eine Liste von Kacheln, die die einzelnen Runden des Spiels darstellen. Der Nutzer kann sehen, wie viele Runden gespielt wurden und sich direkt interessante Rundenzwischenstände anzeigen lassen. Dafür reicht ein Klick auf die jeweilige Kachel.

Danger Fields			
635	634	633	632
631	630	629	628
627	626	625	624
623	622	621	620
619	618	617	616
615	614	613	612
611	610	609	608

Abbildung 38: Rundenauswahl in der History

Es gibt ebenfalls eine Möglichkeit ein Spiel vollständig anzuschauen. Dafür gibt es den Play-Button. Dieser spielt das gesamte Spiel mithilfe der einzelnen Runden ab und visualisiert diese im Spielfeld. Die Abspielgeschwindigkeit ist variabel und wird mit einem Range Slider eingestellt.



Abbildung 39: Play-Feature History

In Kombination mit dem Autoplay-Feature vom Monitoring wird die Mächtigkeit der History besonders deutlich. Die History bietet den Einstieg in das Testen von Taktiken in Abwesenheit der Entwickler. Oft interessiert man sich nur für den groben Verlauf und dafür bedarf es nicht mehr als eine schnelle Visualisierung. Im Folgenden sieht man ein Bild, in dem die Oberfläche benutzt wurde, um sich ein exemplarisches Spiel anzuschauen.

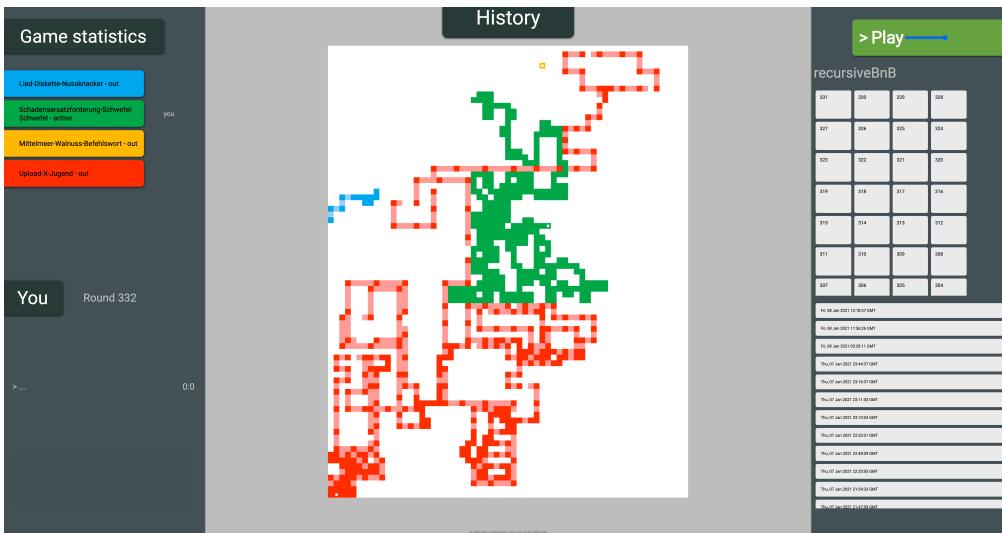


Abbildung 40: Weboberfläche History

Für genauere Details besteht die Möglichkeit sich gespeicherte JSON-Objekte der Spiele anzeigen zu lassen. Um dieses zu tun, müssen wir auf unsere Datenbank zugreifen (vgl. Kapitel 3.5). Das können wir im MongoDB Compass tun, dieses ist die grafische Benutzeroberfläche der Mongo Datenbank. Mit Compass kann man unter anderem einzelne Daten exportieren, betrachten und Suchabfragen formulieren. Außerdem kann man in dieser Anwendung die komplette Datensammlung einsehen.

Wenn die Analyse komplizierter wird, kann man die Abfragen in den Code einbinden. Dadurch werden eröffnet uns völlig neue Möglichkeiten. Wir können die abgerufenen Daten analysieren, speichern und verarbeiten. Dafür wird die MongoDB im Code angesprochen und die nötigen JSON-Objekte werden abgerufen, ins Programm geladen und analysiert. Dieser Vorgang wird im Kapitel 5.6 näher beschrieben.

Abbildung 41: Einsehen von Datensätzen in MongoDB Compass

Abbildung 42: Beispielhafte Query in MongoDB Compass

Abbildung 43: gezielte Suche eines defekten Datensatzes mit einer Game-ID

Mit diesen vielfältigen Möglichkeiten konnten wir die Daten gezielt filtern und dadurch

Rückschlüsse ziehen.

## 5.5 Testen

Wenn es in einem Projekt um Software geht, ist es unabdingbar sich mit der dazugehörigen Softwarequalität zu beschäftigen. Software ist bekanntlich ein immaterielles Produkt. Das hat zur Folge, dass das Bestimmen der Güte und des Entwicklungsfortschritts, im Vergleich zu anderen (materiellen) technischen Produkten nicht immer trivial ist. Oft ist es sogar problematisch eine erwünschte Qualität zu definieren das Quantifizieren ist in solchen Fällen besonders knifflig. Dies liegt unter anderem daran, dass man oft mit abstrakten Informationen arbeitet, die nicht immer greifbar sind und Güte stets von objektiven Einflüssen der verschiedenen Entwickler abhängig ist. Zu unserem Vorteil sind in diesem Projekt Zielkriterien definiert, die uns zwar die Definition der Qualität erleichtern, jedoch nicht das Quantifizieren abnehmen.<sup>56</sup>

Um eine effektive Arbeit zu gewährleisten ist es essentiell, sich um die Quantifizierung der Qualität Gedanken zu machen. Damit beschäftigt sich das Testen. Im Hinblick auf die Effizienz haben wir uns daher intensiv Gedanken gemacht, wie man den Code zielgerichtet testen kann. Dies lässt sich weiter spezifizieren:

### Manuelles Testen

Das ist ohne Diskussion eine sehr teure Angelegenheit im Software Engineering, deren Gebrauch, aus ökonomischen Gründen möglichst minimiert werden sollte. Für große oder auch für kleinere (studentische) Entwicklerteams gibt es stets Ressourcenknappheit. Diese können monetärer Natur sein, zum Beispiel beim Einsatz von kostenintensiven Equipment, wie gemietete Serverleistung, oder eher zeitlicher Natur sein. Für uns als Vollzeitstudierende war Zeit als Ressource besonders wichtig. Beim Implementieren von (komplexen) Funktionen braucht man regelmäßig direktes zielgerichtetes Feedback. Dementsprechend lassen sich manuelle Tests, zum Nachteil der Effizienz, nicht ganz verhindern.

Zum Start war unser Fokus die Funktion der geschaffenen Infrastruktur sowie die korrekte Kommunikation mit der API, sprich Senden und Empfangen der JSON-Spielberichte, zu gewährleisten. Erst wenn man sich selbstbewusst die Funktionen dieser Grundelemente attestieren kann, ist eine effiziente Arbeit möglich, da diese Funktionen das Grundgerüst unserer Arbeit sind. Mit inkonsistent funktionierender Infrastruktur arbeiten würde bei Fehlern schwerwiegende Folgen haben.

Beim Implementieren wurde das manuelle Testen oft verwendet. Meistens haben wir uns das Ziel gesetzt neue Features der Taktiken in Echtzeit auszuwerten. Dabei bedarf es menschlicher Einschätzung und somit ist an dieser Stelle manuelles Testen Pflicht. Man spricht von dynamischer Untersuchung<sup>57</sup>. Wir setzen Eingabemengen, welche Änderungen beinhalten, und können so das tatsächliche Verhalten der Software auswerten. Im Praxiskontext war es eine übliche Routine nach einer neuen Entwicklung ein Probespiel

---

<sup>56</sup>[Balzert, 2009, Seite 9-10]

<sup>57</sup>[Balzert, 2009, S. 67]

zu starten und unseren Agenten im Spiel zu beobachten. Dabei stellt man sich jedes mal verschiedene Fragen:

### 1. Vorangegangener Zustand

- Wie ist der momentane Entwicklungsstand der Software?  
Welche Funktionalitäten funktionieren bereits zuverlässig?
- Welches Verhalten kennen wir bereits aus vorherigen Tests?

### 2. Soll Zustand

- Was für ein Verhalten erwarten/erhoffen wir als Testergebnis?
- Wie können wir überhaupt anhand eines Testergebnisses darauf schließen, dass unsere Implementierung korrekt funktioniert?

Diese Frage mag trivial klingen, da man sich denken könnte, dass man dafür eben das Testergebnis hat. Das mag in der Tat bei einfachen iterativen Algorithmen (zum Beispiel die dontDie-Taktik) der Fall sein, da man ohne größeren Aufwand das Ergebnis problemlos deuten kann. Auf das Beispiel mit der dontDie-Taktik bezogen, würde man sehen, dass der Agent vor Hindernissen abbiegt. Dies geht jedoch nur auf so einer trivialen Weise, wenn man den Spielverlauf mit dem Monitoring sehr leicht ablesen kann und die Taktik nicht kompliziert ist. Bei komplexeren und vor allem rekursiven Algorithmen ist dies jedoch nicht mehr im gleichen Grade trivial. Bei wachsender Komplexität/Rekursionstiefe kann man die Funktion eines Algorithmus nicht mehr ad-hoc ablesen beziehungsweise wäre der nötige Aufwand um dies zu tun in keinem Verhältnis zum Nutzen. Es ist somit an der Zeit, sich auch Gedanken zu machen, wie man eine abstrakte Ergebnis zuverlässig überprüfen kann.

- Gibt es kritische Stellen, die fehleranfällig sind?  
Oft hat man eine Intuition, dass gewisse Code Fragmente fehleranfällig sind. Diese Stellen betrachtet man fokussiert.
- Was könnten mögliche Fehler sein?  
Dieser Punkt ähnelt dem vorherigen, ist jedoch konkreter. Wenn man weiß, dass man einen Fehler hat, muss man nicht zwingend Tests verwerfen oder abbrechen. Wenn man das berücksichtigt, ist es oft möglich, einen anderen unabhängigen Teil weiterhin zu testen. Im dritten Schritt, dem Testergebnis, muss man diesen Kompromiss jedoch im Hinterkopf behalten.

### 3. Ist Zustand / Testergebnis

- War der Test erfolgreich?  
Sind alle definierten Ziele erfüllt?
- Sofern der Test nicht erfolgreich war, wie groß ist die Abweichung vom Ziel?  
Man unterscheidet oft zwischen erwarteten und unerwarteten Fehlern. Oft offenbaren Tests auch Fehler von Problemen, denen man sich vor dem Test

nicht bewusst war. Dies ist, im Vergleich zum statischen Untersuchen<sup>58</sup>, ein bedeutender Vorteil. Danach folgt selbstverständlich die Fehlersuche. Im besten Fall kann man den Fehler zeitnah entdecken, beheben und durch einen neuen Test verifizieren.

Vergleichbar zur Überprüfung gilt, dass sich so ein Fragenkatalog recht trivial klingt. Bei komplizierten Taktiken, wie zum Beispiel RecursiveBnB, ist das Auswerten einer Runde nicht stets klar. Wenn man sich diesen Richtfragen vor und während eines Tests genau bewusst ist und weiß worauf man achten muss, kann es einem die Arbeit enorm erleichtern.

### Testszenarien

Zu Beginn des Kapitels wurde mehrfach betont, dass es wichtig ist, die Erkenntnisse gewinnen zu können ob ein Code wirklich funktioniert und dies an einem Test nachzuweisen. Dafür eignen sich Testszenarien besonders gut. Das hat sich für die Projektarbeit als besonders hilfreich erwiesen.

Beim Nutzen der API muss man immer eine Wartezeit, die bis zu fünf Minuten betragen kann, berücksichtigen. Ignoriert man diese, bekommt man eine Fehlermeldung und kann nicht testen. Außerdem ist das Spielverhalten nicht determinierbar, da andere Spieler ebenfalls spielen. Das Testen von kleineren Funktionen kann sich als Folge ziemlich mühselig und zeitaufwändig erweisen, da es eine randomisierte Gestalt annimmt. Will man zum Beispiel überprüfen, wie sich der Agent in einer Sackgasse verhält, müsste man solange spielen, bis man zufällig in einer Sackgasse landet. Diese kritische Stelle muss im Anschluss in den JSON-Dateien außerdem noch ausfindig gemacht werden. Das ist nicht effizient. Sobald man das Verhalten mit Gegnern üben möchte, kann es passieren, dass man unbestimmt lange warten muss, da nicht immer ein Spieler auf dem Server anwesend ist, der in der Lage ist, sich auf der Karte zu bewegen. Zu Beginn es Wettbewerbs war das ein besonders großes Problem, da viele gegnerische Teams noch nicht fortgeschritten in der Entwicklung waren.

Um den Prozess einfacher und vom Zufall unabhängig zu gestalten, haben wir Testszenarien entwickelt, auf die wir während der Entwicklung zurückgreifen konnten. Dafür haben wir eine abstrahierte Version des Spiels nachgebaut, die nur uns als einzigen Spieler hat und die Zellen des Spielfelds auf Wunsch belegt, um mögliche Szenarien abzubilden.

Diesen Prozess nennt man *Reverse Engineering*<sup>59</sup> <sup>60</sup>. Wir haben von einer niederen Ebene (das gegebene Spiel mit der API) in eine höhere Ebene (eigen erstellter Quellcode) durch zielgerichtetes Untersuchen der Strukturen der API-Kommunikation und der Spielregeln transformiert. Wenn ein Programm in unserem Testszenario funktioniert, dann können wir somit folgern, dass es im realen Spiel ebenfalls funktioniert. Diese Möglichkeit hat

---

<sup>58</sup>[Balzert, 2009, S. 67]

<sup>59</sup>[Balzert, 2009, S. 64]

<sup>60</sup>[Baumgartl, 2016, Folie 2 ]

Fehlersuche und Tests, vor allem am Anfang, vom Zufall unabhängiger gemacht und somit enorm vereinfacht und beschleunigt.

### **Test-Workflow**

Da man beim echten Testen auf dem Game Server die Wartezeit als Restriktion beachten muss, müsste man nach jedem Test bis zu 5 Minuten warten, wenn man den Test nicht auf einem Szenario auslagern kann. Da dies nicht zeiteffizient ist, haben wir beim Programmieren das Spiel im Hintergrund dauernd laufen lassen. Das steigert die Produktivität und ermöglicht uns Datensammlung, sowie das Bilden einer Intuition. Da man selbst anwesend ist, erhält man direktes Feedback. Aufgrund des geringen Stichprobenumfangs dieser parallelen Tests im Hintergrund kann man nicht immer konkrete Beurteilungen der Güte fällen. Dafür braucht es eine Auswertung mit einem größeren Stichprobenumfang.

### **(Halb-)Automatisches Testen**

Durch das AutoPlay-Feature können wir während unserer Abwesenheit massenhaft Runden mit unserem Agenten spielen und diese in der Datenbank abspeichern. Dies eröffnet uns nun die Möglichkeit unsere Algorithmen quantitativer zu testen. Man kann nun erkennen, ob der Algorithmus tendenziell bessere Ergebnisse leistet, da wir einen größeren Stichprobenumfang haben. Bei interessanten Datensätzen besteht immer die Möglichkeit, sich diese in der History persönlich erneut anzuschauen und daraus Erkenntnisse zu gewinnen. Bezuglich der großen Datensammlung bietet es sich an die Daten auszuwerten, grafisch aufzubereiten und zu interpretieren. Durch diese Analyse kann man zum Beispiel bestimmen, wie viel Prozent der Spiele der Agent gewinnen kann. Diese Erkenntnisse sind jedoch mit Vorsicht zu genießen, da das Verhalten der Gegner eine unbekannte Variable darstellt. Wir können nicht wissen, wie weit die Gegner in ihrer Implementierung sind. Eine Statistik mit zu großer Laufzeit könnte einen verfälschten Eindruck der Güte vermitteln. Nichtsdestotrotz ist es ein gutes Indiz, wenn man in mehreren aufgezeichneten Runden den erwünschten Soll-Zustand erkennt.

## **5.6 Datenanalyse**

Im Zuge der Analyse der Taktiken haben wir uns stets gefragt, wie gut eine Taktik ist. Wir konnten die Taktiken bereits stichprobenartig beobachten oder sie im Autoplay-Modus laufen lassen. Der Vorteil der Live-Analyse ist das direkte Feedback. Dafür dauert das Ausführen der Runden viele Minuten. Wir lassen unseren Agenten daher über einen längeren Zeitabschnitt im Autoplay-Modus laufen und sammeln Spieldaten. Diese liegen bekanntlich als JSON-Objekte vor. Sie manuell auswerten ist nicht effizient möglich, daher nutzen wir eine automatisierte Analyse. Mit den analysierten Ergebnissen kann man aussagekräftige Visualisierungen erstellen. Wie in Kapitel 5.3 bereits erwähnt, haben wir die Möglichkeit Abfragen an die Datenbank im Code einzubinden.

```
const queryResult = await models.Protokoll.find({tactic : "resursiveBnB"});
```

Abbildung 44: Exemplarische Datenabfrage im Code

Damit wir die Daten analysieren können, rufen wir diese zunächst mit einer Abfrage an die Datenbank ab und speichern diese. Wenn man die Daten nun hat, muss man sich überlegen, wie man diese analysieren kann. Wir haben uns entschieden, dass wir gerne wissen möchten, welche Taktik wie oft gewinnt. Daraus berechnen wir eine Sieg-Quote berechnen für jede Taktik.

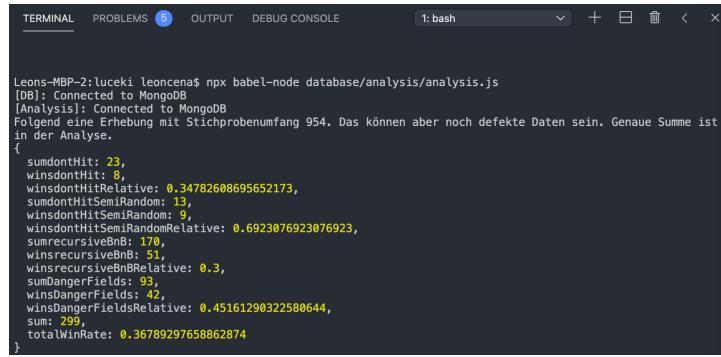
Das Auswerten der Daten geschieht iterativ. Von der Datenbank erhalten wir einen Array mit JSON-Objekten, folgend *games* genannt. Eine gewonnene Runde erkennt man daran, dass der eigene Agent (SpielerNr: *you*) in der letzten Runde als einziger Spieler noch aktiv ist. Dazu gibt es die Funktion *isGameAWin(game)*, die ein Spiel in Form eines JSON-Objekts übergeben bekommt und uns einen booleschen Wert zurückgibt. Die sonstigen interessanten Attribute (*tactic*, *amountRounds*, *timestamp*) können mit jeweiligen Methoden ebenfalls ermittelt werden.

Die Funktion *getGameReport(game)* nimmt ein Spiel in Form eines JSON-Objekts an und erstellt dafür einen GameReport. Dieser beinhaltet alle interessanten Informationen, die wir zum Auswerten benötigen. Den Zeitstempel kann man auch in den GameReport einfließen lassen. Dieser ist jedoch meistens nicht notwendig, daher haben wir uns ihn primär nicht anzeigen lassen, uns aber im Code vorbehalten ihn bei Bedarf anzeigen zu lassen.

```
{ tactic: 'recursiveBnB', won: true, amountRounds: 12 }
```

Abbildung 45: Exemplarischer GameReport

Nun kann man durch den Array *Games* iterieren. Das geschieht in der Methode *analyseGames(games)*. Jedes Element (ein einzelnes Spiel) wird zunächst auf Korrektheit überprüft. Dafür kontrollieren wir, ob es einen korrekten letzten Spielzug gibt. Ist das der Fall, erstellen wir den gameReport und fügen ihn in das Ergebnisarray GameReports. Wir erhalten somit am Ende der Ausführung einen Array mit allen gültigen GameReports. Diese bereiten wir mit der Funktion *analyzeTactics(gameReports)* nach. Wir summieren die Anzahl der Siege differenziert und geben diese in einem Array aggregiert aus. Damit berechnen wir die einzelnen Siegesquoten für die Taktiken, sowie eine Siegesquote für die gesamten übergebenen Spiele. Da die Datenquelle die Datenbank ist, erhalten wir bei jeder Ausführung stets die aktuellen Daten und müssen keine Daten manuell in das System einpflegen.



```
Leons-MBP-2:luceki leoncena$ npx babel-node database/analysis/analysis.js
[DB]: Connected to MongoDB
[Analysis]: Connected to MongoDB
Folgend eine Erhebung mit Stichprobenumfang 954. Das können aber noch defekte Daten sein. Genaue Summe ist
in der Analyse.
{
  sumdontHit: 23,
  winsdontHit: 1,
  winsdontHitRelative: 0.34782608695652173,
  sumdontHitSemiRandom: 13,
  winsdontHitSemiRandom: 9,
  winsdontHitSemiRandomRelative: 0.6923076923076923,
  sumrecursiveBnB: 170,
  winsrecursiveBnB: 51,
  winsrecursiveBnBRelative: 0.3,
  sumDangerFields: 93,
  winsDangerFields: 42,
  winsDangerFieldsRelative: 0.45161290322580644,
  sum: 299,
  totalWinRate: 0.36789297658862874
}
```

Abbildung 46: Beispielhafte abgeschlossene Analyse einer Stichprobe von Spielen (Anmerkung: Die Qualität der Taktik ist nicht final)

Somit haben wir nun alle nötigen Kennzahlen und können diese auswerten und gegebenenfalls visualisieren. Wenn man eine andere Stichprobe auswerten möchte, kann man die Datenbank-Abfrage anpassen und die Resultate auf Wunsch mit Code nachverarbeiten.

## 6 Abschlussdiskussion

### 6.1 Zusammenfassung

Rekapitulierend soll zunächst ein letzter Überblick zu den einzelnen Teilen unseres Softwaresystems gegeben werden. Strukturell gibt es die App und den Taktik Controller zum Verwalten des ganzen Systems und der Taktiken. Der Client und das Monitoring sorgen für eine graphische und interaktive Benutzeroberfläche. Zur Analyse und Verbesserung existiert eine Datenbank auf einem dauerhaft laufenden Server, eine Analyse Klasse für Statistiken unserer Taktiken und ein Testserver zum schnellen Spielen und Trainieren. Die Taktiken nutzen verschiedene Prinzipien und Algorithmen wie Rekursion, Branch and Bound, Randomisierung, Tiefensuche, neuronale Netze, A\*, FloodFill, Fisher-Yates und weitere. Da einzelne Taktiken in verschiedenen Situationen vorteilhafter sind als andere, werden sie oft untereinander gewechselt und genutzt. Beispielsweise kombiniert die Taktik recursiveBnB mit festgelegter spezieller Priorisierung die Fähigkeit einen gegebenen Platz optimal auszunutzen, wie es sonst dontHit macht, mit der Fähigkeit der Zukunftsprognose, die gezielte Sprünge und Vermeiden von Sackgassen erlaubt. Wiederrum darauf baut die dangerFields Taktik auf. Sie zeigt ihre Stärke zu Beginn von Spielen, indem sie große freie Flächen ansteuert. Wenn sie jedoch einmal eingeschlossen ist, wechselt sie auf recursiveBnB, um den Platz optimal zu nutzen und gegebenenfalls aus der eigenen Gefangenschaft zu springen. Nicht zu vergessen die Taktik tfModels, die unsere KI nutzt. Sie ist jedoch zum Zeitpunkt der Abgabe nicht fortgeschritten genug trainiert, um dangerFields zu übertreffen.

### 6.2 Reflexion

Nun möchten wir gerne über unsere Arbeit und dieses Projekt reflektieren. Mit diesen ungewohnten Umständen des online Arbeitens gingen besondere Herausforderungen und auch Chancen einher. Wir mussten uns auf viele Restriktionen einstellen und haben als Gruppe in diesen schweren Zeit versucht, bestmöglich zu arbeiten. Unserer Meinung nach ist uns das gut gelungen. Wir haben große Teile dieser Arbeit remote angefertigt und organisiert. Das hat erstaunlich gut funktioniert. Unsere anfänglichen Bemühungen in Hinsicht auf Zeitmanagement und Test-Infrastruktur haben unsere Gruppenarbeit gefördert.

Abschließen kann man sagen, dass wir stolz auf unsere finale Lösung sind. Unsere Algorithmen liefern beeindruckende Ergebnisse und gewinnen viele Spiele. Erstmalig haben wir aus dem Studium gelernte Konzepte praktisch angewendet, unsere Kenntnisse gefestigt und auf neue Anwendungsfelder übertragen.

Wir können von uns behaupten, dass wir definitiv an der Aufgabe gewachsen sind. Bei der Bearbeitung haben wir viele neue Techniken gelernt. Diese erstrecken sich von der serverseitigen JavaScript-Implementierung, der Server-Client-Architektur und *Docker* bis hin zu *node.js*, den theoretischen Hintergründen des Machine Learnings und der Nutzung von *tensorflow.js*. Bei der Implementierung des Spiels konnten wir einen Bezug

zu möglichen Projekten aus dem realen Umfeld herstellen. Vor allem beim Machine Learning sind uns die weitreichenden Anwendungsfelder besonders klar geworden. Im Rahmen dieser Arbeit haben wir mit Machine Learning ein Themenfeld entdeckt, welches uns begeistert und mit dem wir uns zukünftig detaillierter auseinander setzen werden.

### 6.3 Ausblick

Obwohl wir mit unseren Ergebnissen zufrieden sind, liegt es in unserer, zum Teil durch das Studium indoktrinierten, Natur sich Gedanken um weitere mögliche Optimierung zu machen. Wir sehen Potential einer möglichen Qualitätssteigerung. Diese starten mit den theoretischen Konzepten des Software Engineering. Zum Zeitpunkt des Projekts befinden wir uns im dritten Semester und haben die Vorlesung Software Engineering noch nicht gehört. Dementsprechend fehlen uns Kenntnisse, die de facto als Standard in Projekten der Gleichen gelten sollten. Diese theoretischen Konzepte sind unter anderem Entwurfsmuster, Modellierung und Tests.

Zudem sehen wir Chancen in der Benutzung von *TypeScript*. Dies würde den *JavaScript*-Code vereinfachen, sodass er leichter zum Lesen und Debuggen ist. Es ermöglicht starke Typisierung.

Eine Erwähnung noch zur IT-Sicherheit. Unser Monitoring verwendet zwar https (Zertifikat muss man manuell ersetzen). Dennoch liefert das Programm möglicherweise Angriffstellen, wenn man es auf einem öffentlich zugänglichen Server nutzt. Das Herunterladen von Modellen von unserem Server ist ein Beispiel hierfür.

Bezüglich des maschinellen Lernens sehen wir noch viel Potential für eine bessere KI. Der befristeten Zeit geschuldet war uns langes und ausgiebiges Trainieren der Modelle nicht möglich. Beim Training könnte man die Hyperparameter gezielt anpassen um den Lernfortschritt zu verbessern. Ebenfalls könnte man das Trainieren auf den öffentlichen Gameserver auslagern, um mit gegnerischen Aktionen zu trainieren. Unsere Lösung trainiert momentan ausschließlich mit Instanzen unserer eigenen Taktiken. Ein weitere Idee wäre es, das lokale Trainieren auf dem eingeben Testserver zu beschleunigen. Dafür könnte man auf die Server-Client-Architektur verzichten und in der gegebenen Zeit deutlich mehr Spiele angehen. Damit könnten wir die Rechenkraft einer performanten Grafikkarte wesentlich effizienter ausschöpfen, denn dies würde die Trainingsgeschwindigkeit um ein Vielfaches erhöhen.

## Literatur

- [Balzert, 2009] Balzert, H. (2009). *Lehrbuch der Softwaretechnik: Basiskonzepte und Requirements Engineering*. Spektrum Akademischer Verlag.
- [Baumgartl, 2016] Baumgartl, R. (2016). *Spezielle Techniken und Technologien der Informatik: Reverse Engineering*. Vorlesung HTW Dresden.  
Verfügbar unter: <http://www.informatik.htw-dresden.de/~robge/stti/v1/stti-01-einfuehrung.pdf>; zuletzt abgerufen: 03.01.2021.
- [Boyd and Mattingley, 2007] Boyd, S. and Mattingley, J. (2007). *Branch and Bound Methods*. Notizen zu EE364b, Winter 2006-07, Stanford University.
- [Cai et al., 2020] Cai, S., Bileschi, S., Nielsen, E. D., and Chollet, F. (2020). *Deep Learning with JavaScript*. Manning Publications.
- [Cormen et al., 2009] Cormen, T. H. D. C., Leiserson, C. E. M., Rivest, R. L. M., and Stein, C. C. U. (2009). *Introduction to Algorithms*. MIT Press Ltd.
- [Grimme and Bossek, 2018] Grimme, C. and Bossek, J. (2018). *Einführung in die Optimierung*. Springer Fachmedien Wiesbaden.
- [Herzog et al., 2014] Herzog, M., Riedl, W. F., Veldena, L., and München, T. U. (2014). Der a\* - algorithmus.  
Verfügbar unter: [https://www-m9.ma.tum.de/graph-algorithms/spp-a-star/index\\_de.html](https://www-m9.ma.tum.de/graph-algorithms/spp-a-star/index_de.html); zuletzt abgerufen: 03.01.2021.
- [Heumann et al., 2016] Heumann, C., Schomaker, M., and Shalabh (2016). *Introduction to Statistics and Data Analysis*. Springer International Publishing.
- [Hooffacker and Bigl, 2020] Hooffacker, G. and Bigl, B., editors (2020). *Science MashUp. Zukunft der Games*. Springer Fachmedien Wiesbaden.
- [Leland Wilkinson, 2008] Leland Wilkinson, M. F. (2008). The history of the cluster heat map.  
Verfügbar unter: <https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.165.4766&rep=rep1&type=pdf>; zuletzt abgerufen: 03.01.2021.
- [Plagwitz, 2018] Plagwitz, L. (2018). *Numerische Optimierung fuer Deep Learning Algorithmen*. Vorlesung WWU Münster.  
Verfügbar unter: [https://www.uni-muenster.de/AMM/num/Vorlesungen/Seminar\\_Wirth\\_Master\\_WS18\\_b/handouts/Plagwitz.pdf](https://www.uni-muenster.de/AMM/num/Vorlesungen/Seminar_Wirth_Master_WS18_b/handouts/Plagwitz.pdf); zuletzt abgerufen: 17.01.2021.
- [Schmaranz, 2002] Schmaranz, K. (2002). Rekursion. In *Xpert.press*. Springer Berlin Heidelberg.
- [Yu and Yang, 1998] Yu, G. and Yang, J. (1998). On the robust shortest path problem. *Computers & Operations Research*, 25(6):457 – 468.