



# INFORMATICUP 2021 SPE\_ED

Universität Augsburg

## Abschlussbericht

vorgelegt von

**Julius Brandl**  
**Nicolas Breinl**  
**Maximilian Demmler**  
**Lukas Hartmann**

im Januar 2021

## Abstract

*Im vorliegenden Bericht wird die Entwicklung einer Software beschrieben, die das Multiplayer-Spiel Spe\_ed selbstständig spielt. Nach der Evaluierung von klassischen Robotik- und RL-Ansätzen wird ein Minimax-Ansatz aus der Spieltheorie vorgestellt und für das Problem weiterentwickelt. Der Multi-Minimax-Algorithmus erlaubt die Anwendung auf mehr als zwei Spieler und eine Voronoi-Heuristik wird als passende Evaluierungsfunktion für Spe\_ed eingeführt. Der Aufbau der Python-Software wird vorgestellt und verschiedene Versionen gegeneinander evaluiert. Dabei zeigt SW-RG-V-Multi-Minimax Vorteile gegenüber anderen Ansätzen. Auswertungen gegen andere Teams auf dem Live-Server ergeben Gewinnraten von mehr als 80%.*

# Inhaltsverzeichnis

Abstract . . . . .	2
<b>1 Einleitung</b>	<b>6</b>
<b>2 Ansätze und Related Work</b>	<b>8</b>
2.1 Trajektorienplanung . . . . .	8
2.2 Reinforcement Learning . . . . .	9
2.3 Spieltheorie . . . . .	11
<b>3 Voronoi-Basierter Multi-Minimax Algorithmus</b>	<b>14</b>
3.1 Multi-Minimax . . . . .	14
3.2 Voronoi-Heuristik . . . . .	15
3.3 Erweiterungen . . . . .	16
3.3.1 Vorsortierung der Aktionen . . . . .	16
3.3.2 Endgame-Erkennung . . . . .	17
3.3.3 Wall-Hugging . . . . .	17
3.3.4 Reduktion der betrachteten Gegner . . . . .	17
3.3.5 Einschränkung der eigenen Geschwindigkeit . . . . .	17
3.3.6 Depth-First Iterative Deepening . . . . .	18
3.3.7 Multiprocessing . . . . .	18
3.3.8 Cython . . . . .	19
3.3.9 Sliding Window . . . . .	19
3.3.10 Kamikaze . . . . .	19
<b>4 Software</b>	<b>21</b>
4.1 Architektur und Modellierung . . . . .	21
4.2 Software-Testing . . . . .	24
<b>5 Ergebnisse</b>	<b>26</b>
5.1 Live-Server . . . . .	26
5.2 Offline-Versionsauswertung . . . . .	27
<b>6 Diskussion</b>	<b>29</b>

# Abbildungsverzeichnis

2.1	Minimax mit Alpha-Beta-Pruning . . . . .	13
3.1	Multi-Minimax . . . . .	15
3.2	Voronoi-Diagramm . . . . .	16
4.1	Komponentendiagramm unseres Projekts . . . . .	22
4.2	Vereinfachtes Klassendiagramm der Modell-Architektur . . . . .	23
4.3	Code-Beispiel zur Erstellung und Ausführung des Modells . . . . .	24
4.4	Testabdeckung für das <i>Core</i> -Softwaremodul . . . . .	25

# Tabellenverzeichnis

5.1	Auswertung der Spiel-Parameter . . . . .	27
5.2	Auswertungsparameter . . . . .	28
5.3	Auswertung verschiedener Versionen . . . . .	28

# 1 Einleitung

In den folgenden Abschnitten werden wir Ihnen den Verlauf unserer Teilnahme am *InformatiCup 2021* präsentieren und einen Überblick über die weiteren Kapitel bieten. Die gesamte Ausarbeitung ist bewusst in einem lockeren Ton und mit direkter Ansprache formuliert, um den Lesefluss angenehmer zu gestalten.

In den ersten Tagen nach dem Start des Wettbewerbs haben wir uns mit der Aufgabenstellung vertraut gemacht und besondere Herausforderungen erarbeitet. Dabei ist uns aufgefallen, dass *Spe\_ed* im Prinzip eine Erweiterung des Spielmodus *Light Cycles* aus dem Arcade-Spiel *Tron*<sup>1</sup> ist. Wesentliche Unterschiede zu *Light Cycles* sind:

- Die Spielfeldgröße ist variabel
- Die Anzahl der Spieler ist variabel
- Die Zeit für einen Zug ist beschränkt und variabel
- Spieler können unterschiedliche Geschwindigkeiten besitzen
- Alle 6 Runden entstehen Lücken in den Spuren schneller Spieler

Diese Erweiterungen stellen zugleich auch die besonders herausfordernden Eigenschaften des Spiels dar. Dennoch bieten bereits existierende algorithmische Ansätze für *Light Cycles* eine gute Grundlage für *Spe\_ed*. In den Kapiteln 2 und 3 gehen wir unter anderem genauer darauf ein, von welchen *Light Cycles* -Ansätzen wir uns inspirieren lassen haben und wie wir mit den genannten Herausforderungen umgehen.

Nachdem wir uns ein gutes Bild von der Aufgabenstellung gemacht hatten, haben wir durch altbewährtes Brainstorming und Mindmapping versucht, ein möglichst breites Spektrum an Lösungsansätzen zu erarbeiten. Dabei haben wir die drei - auf der InformatiCup-Website<sup>2</sup> angerissenen - übergeordneten Lösungsansätze *Spieltheorie*, *künstliche Intelligenz* und *Planungsheuristik* als Ausgangspunkt verwendet. Folgende Kernideen haben wir letztendlich als am vielversprechendsten erachtet:

- Trajektorienplanung
- Reinforcement Learning (RL)
- Tiefensuche

---

<sup>1</sup>[https://de.wikipedia.org/wiki/Tron\\_\(Computerspiel\)](https://de.wikipedia.org/wiki/Tron_(Computerspiel))

<sup>2</sup><https://informatocup.github.io/challenges/spe-ed>

Diese Ansätze haben wir in der initialen Projektphase parallel näher untersucht und Prototypen implementiert. In Kapitel 2 bieten wir einen Überblick über verwandte Arbeiten zu den jeweiligen Ansätzen, heben Schwächen und Stärken hervor und begründen die Entscheidung die Tiefensuche als finalen Ansatz zu wählen. Da Spe\_ed nur über eine Webschnittstelle gespielt werden kann, haben wir zu diesem Zeitpunkt außerdem entschieden, das Spiel nachzuimplementieren. Zum einen ermöglicht das unsere Ansätze gezielt und effizient offline testen zu können. Zum anderen ist es für RL und Tiefensuche grundlegend notwendig Zugriff auf ein vom Algorithmus kontrollierbares Modell von Spe\_ed zu haben. Kapitel 4 beschäftigt sich näher mit unserer Vorgehensweise bei der Modellierung von Spe\_ed.

Den größten Abschnitt des Wettbewerbs haben wir verwendet, um die Tiefensuche zu implementieren und durch Erweiterungen und Optimierungen zu ergänzen. Im Speziellen haben wir einen Multi-Minimax-Algorithmus entworfen, der die sogenannte Voronoi-Heuristik als Hauptevaluationskriterium verwendet. In Kapitel 3 wird der Algorithmus im Detail vorgestellt.

Letztendlich hatten wir gegen Ende des Wettbewerbs mehrere Varianten des Algorithmus mit unterschiedlichen Erweiterungen implementiert. Um eine finale Abgabe zu bestimmen, haben wir ausgewählte Varianten in unserem Modell gegeneinander antreten lassen und die Spiele ausgewertet. Zusätzlich haben wir erfolgreiche Varianten auch auf dem Online-Server des InformatiCup getestet. In Kapitel 5 präsentieren wir die Ergebnisse dieser Auswertungen.

Alles in allem können wir behaupten, dass unsere Lösung gute Resultate erzielt. Trotzdem ist uns natürlich bewusst, dass unser Algorithmus - wie vermutlich jeder Algorithmus mit derzeitiger Rechenleistung - nicht in allen Spielpositionen die allgemeingültig beste Aktion wählt. Daher betrachten wir unseren Ansatz in Kapitel 6 in einem kritischen Licht und bieten Ideen für zukünftige Forschungen.

## 2 Ansätze und Related Work

Wir haben *Trajektorienplanung*, *Reinforcement Learning* und *Spieltheorie* als vielversprechendste Ansätze für Spe\_ed erachtet und genauer unter die Lupe genommen. Die folgenden Unterkapitel fassen unsere Erkenntnisse zur Anwendbarkeit der jeweiligen Ansätze und Untersuchungen verwandter Arbeiten zusammen.

### 2.1 Trajektorienplanung

Spe\_ed kann wie folgt betrachtet werden: Für einen Agenten müssen Steuerbefehle erzeugt werden, die zu einer Reihe von Wegpunkten mit Geschwindigkeiten - also einer Trajektorie - führen. Der Pfad soll zudem Kollisionen vermeiden. Diese Betrachtungsweise führt zum Ansatz, das Problem mithilfe von Trajektorienplanung aus der klassischen Robotik zu lösen. Einige der prominentesten Vertreter dieser Algorithmen sind Vector Field Histogram (VFH) [1], Rapidly-exploring Random Tree (RRT)[2], Voronoi-Diagramm [3], und Artificial Potential Field (APF) [4]. Wir haben uns entschieden, den letztgenannten Ansatz zu testen, da dieser die Umgebung in einer für Spe\_ed passend erscheinenden Weise darstellt. Beim APF wird die Umgebung als ein Feld von Potenzialen modelliert, bspw. indem Hindernisse ein hohes abstoßendes Potenzial erhalten. Der Agent bestimmt dann zur Pfadplanung den Weg des steilsten Abstiegs (über den Gradient). Passend für Spe\_ed, da große Freiflächen bevorzugt und Gegner gemieden werden. Zudem kann der Algorithmus mit evolutionären Ansätzen kombiniert werden, woraus bspw. das Evolutionary Artificial Potential Field (EAPF) entsteht. [4]

Der Lösungsansatz alleine über Trajektorienplanung bringt allerdings einige Probleme mit sich. Das Bedeutendste ist, dass Trajektorienplanung alleine keine Strategie finden kann. Vielmehr implementiert es eine zufällige oder vom Entwickler erdachte Strategie, z.B. „halte möglichst großen Abstand zu Hindernissen“. Das ist auch dadurch bedingt, dass in der klassischen Robotik meist ein Ziel wie „fahre diesen Punkt an“ oder „halte genau diesen Abstand zum Fahrer vor dir“ vorgegeben ist. In unserer Aufgabe ist das Ziel zu abstrakt („gewinne das Spiel“/„Überlebe länger als die anderen“). Besonders das evaluierte APF schafft es mit der typischen Parameterkonfiguration nicht Gegner abzuschneiden oder den Platz nahe an Wänden effizient zu nutzen.<sup>1</sup> Aus den genannten Gründen wurde dieser als alleiniger wieder verworfen, allerdings taucht das Voronoi-Diagramm im finalen Ansatz auf.

---

<sup>1</sup>Wände erhalten normalerweise ein abstoßendes Potenzial, allerdings ist es an sich sinnvoll, nahe heranzufahren.



## 2.2 Reinforcement Learning

In den letzten Jahren haben RL-Algorithmen beeindruckende Fortschritte gezeigt. Mittlerweile übertreffen sie beispielsweise menschliche Spitzenleistungen in Atari-Spielen, Schach, Shogi, Go und Dota 2 [5, 6, 7]. Daher ist es nur naheliegend, RL auch für den diesjährigen InformatiCup als Ansatz in Betracht zu ziehen.

RL beschäftigt sich mit der Fragestellung, wie Software-Agenten lernen können, eine bestimmte Aufgabe in einer bestimmten *Umgebung* zu meistern [8]. Im Bezug auf den diesjährigen InformatiCup ist die Umgebung das Spiel `Spe_ed`, der Agent ein Spieler und die Aufgabe zu gewinnen. Es gibt allerdings noch ein paar Eigenheiten von `Spe_ed`, die die Anwendung von RL erschweren: (1) Die Spielfeldgröße ist variabel, (2) Die Anzahl der Gegenspieler ist variabel, (3) Die *Policy* (Verhaltensweise) der Gegenspieler ist unbekannt.

Der Lernprozess bei RL wird typischerweise durch den Markov Decision Process (MDP) beschrieben: Der Agent führt in jedem Zeitschritt  $t$  eine Aktion  $a_t$  aus, die den Zustand  $s_t$  der Umgebung verändert. Nachdem  $a_t$  ausgeführt wurde, empfängt der Agent von der Umgebung einen neuen Zustand  $s_{t+1}$  und eine Belohnung  $r_{t+1}$ . Die *Policy*  $\pi$  eines Agenten legt die Aktion fest, die er in einem bestimmten Zustand ausführt. Genauer ist  $\pi(a|s)$  die Wahrscheinlichkeit, dass der Agent die Aktion  $a$  im Zustand  $s$  wählt. Ziel eines RL-Algorithmus ist, eine *Policy* zu erlernen, die die akkumulierten, in Zukunft erwarteten Belohnungen - kurz: *Return* - maximiert. [8]

Um RL auf `Spe_ed` anzuwenden, müssen wir also erst einmal den Zustandsraum modellieren. Dabei kommen einem bereits die Variabilität der Spielfeldgröße und der Spieleranzahl in die Quere. Denn an sich könnte man einen Spielzustand vollständig beschreiben, wenn man den Zustandsraum mit folgenden Informationen ausstattet:

- Die Belegung der Felder auf dem Spielfeld (mit möglichen Zahlencodierungen -1 bis # Spieler)
- Die Positionen aktiver Spieler
- Die Bewegungsrichtungen aktiver Spieler
- Die Geschwindigkeiten aktiver Spieler
- Einen Rundenzähler

RL-Algorithmen fordern allerdings eine Zustandsdarstellung mit einer festen Anzahl an Zustandsattributen. Zur Lösung dieses Problems gibt es zwei Ansätze: (1) Die maximal mögliche Zustandsgröße wählen und kleinere Zustände mit bestimmten *Padding-Werten* auf die maximale Größe erweitern. (2) Nur einen Ausschnitt der Umgebung mit fester Größe betrachten (z.B. nur  $x$  Felder in jede Richtung um den Agenten herum und nur die nächsten  $y$  anderen Spieler betrachten). Dabei können beide Ansätze miteinander kombiniert werden. In jedem

Fall steht jedoch fest, dass der Zustandsraum sehr groß ist - ein Spielfeld der Größe 50x50 mit 5 Spielern hätte mit den oben aufgelisteten Informationen modelliert

$$\text{Höhe} * \text{Breite} + 4 * \text{Spieler} + 1 = 50 * 50 + 4 * 5 + 1 = 2.521$$

Zustandsattribute. Um den Ansatz nicht von vorn herein zu verkomplizieren, haben wir zunächst eine kleine, feste Spielfeldgröße und nur einen Gegenspieler als Umgebung angenommen.

Bevor wir uns nun mit den eigentlichen RL-Algorithmen beschäftigen können, muss allerdings noch eine weitere Frage geklärt werden: Gegen wen spielt unser Agent eigentlich während des Trainings? Man kann den Agent zwar auf einem leeren Spielfeld trainieren, allerdings würde dieser wohl lediglich lernen, alle Felder so langsam wie möglich auszufüllen, um die unausweichliche Eliminierung so lange wie möglich hinauszuzögern.<sup>2</sup> Es kommt auch nicht infrage den Agent gegen einen eigens entwickelten Gegenspieler spielen zu lassen, denn unser Agent würde nur lernen, gegen diesen einen Gegner zu gewinnen. Der derzeit beste Ansatz für solch ein Problem ist das sogenannte *Self-Play*, wie es beispielsweise *Google* und *DeepMind* bei *AlphaZero* [6] und *OpenAI Five* [7] verwenden. Hierbei tritt der Agent immer wieder gegen sich selbst an und lernt somit automatisch gegen Gegenspieler mit unterschiedlicher Policy zu spielen. Um RL-Agenten letztendlich in *Spe\_ed* trainieren zu können, haben wir unser *Spe\_ed*-Modell in ein *Gym-Environment* [9] eingekapselt und die Schnittstellen vektorisiert, um *Self-Play* zu ermöglichen.

Der wohl bekannteste RL-Ansatz ist das 1989 von Chris Watkins vorgestellte *Q-Learning* [10]. *Q-Learning* lernt die optimale *Aktionswert-Funktion*  $q_*$  mit  $Q$  zu approximieren.  $Q$  wird dabei tabellarisch dargestellt, weshalb der Ansatz nur auf Umgebungen mit diskreter Zustands- und Aktionsdarstellung anwendbar ist. Das ist vorerst kein Problem, da beides bei *Spe\_ed* diskret dargestellt werden kann. Was hingegen problematisch ist, ist die Größe des Zustandsraumes. Bereits auf einem 3x3 Spielfeld mit 2 Spielern gibt es  $4^{3*3} = 262.144$  mögliche Belegungen des Spielfelds.<sup>3</sup> Das bedeutet, dass alleine durch ein 3x3 Spielfeld - von Informationen über Spielerpositionen etc. abgesehen - eine  $Q$ -Tabelle mit 262.144 Einträgen benötigt wird. Man kann die  $Q$ -Tabelle zwar dynamisch gestalten und den Zustandsraum durch Symmetrieüberlegungen etwas verkleinern, allerdings explodiert der Zustandsraum trotzdem spätestens bei Spielfeldern größer als 5x5 zu nicht umsetzbaren Größen. Wir konnten *Q-Learning*-Agenten zwar erfolgreich auf einem 3x3 Spielfeld trainieren, allerdings können solch kleine Spielfelder auch durch Tiefensuche in sehr kurzer Zeit optimal gelöst werden. Daher ist ein tabellarischer Ansatz ungeeignet.

2013 haben Mnih et al. einen neuen Meilenstein im Bereich des RL gesetzt, indem Sie Deep Neural Networks (DNNs) als Funktionsapproximatoren zur Approximation der  $Q$ -Funktion beim *Q-Learning* verwenden [5, 11]. Das dabei entstandene Deep  $Q$ -Network (DQN) haben Minh et al. erfolgreich verwendet, um Atari 2600 Spiele zu spielen. Im Gegensatz zum

---

<sup>2</sup>Der Agent erhält die Belohnung 1 für einen Siegeszug, -1 bei Eliminierung und sonst 0.

<sup>3</sup>Die Basis der Berechnung ergibt sich durch  $\#Spieler + 2$ , da dies der Anzahl der möglichen Belegungen eines Feldes entspricht: -1 (Kollision), 0 (Frei), x (Spieler x)

tabellarischen Q-Learning können DQNs kontinuierliche Zustandsdarstellungen als Eingang verwenden. Das reduziert die Größe der Zustandsdarstellung bei `Spe_ed` drastisch - ein 3x3 Spielfeld kann nun unabhängig von der Anzahl der möglichen Belegung eines Feldes mit  $3 * 3 = 9$  Zustandsattributen beschrieben werden. Des Weiteren können DQNs Verallgemeinerungen über Spielzustände lernen. Das hat den Vorteil, dass sie nicht jeden konkreten Spielzustand gesehen haben müssen, um ihn bewerten zu können.

Statt dem standardmäßigen Multilayer Perceptron (MLP) haben Mnih et al. ein Convolutional Neural Network (CNN) als Netzwerkarchitektur verwendet, da sich CNNs besonders gut für die Verarbeitung von Bildern (bzw. *Spiel-Frames*) eignen. Bei `Spe_ed` reicht eine rein bildliche Darstellung des Spielfelds allerdings nicht zur Beschreibung des Zustands aus - es fehlen die Informationen über Spielerpositionen, -geschwindigkeiten und -bewegungsrichtungen. Daher haben wir eine reine MLP-Architektur und eine Architektur mit getrennten CNN und MLP-Eingangslayer, die in tieferen Layern zusammengeführt werden, auf `Spe_ed` getestet. Leider haben beide Architekturen jedoch auch nach längerer Trainingszeit keine guten Ergebnisse geliefert.

Zu diesem Zeitpunkt haben wir uns dafür entschieden, den RL-Ansatz nicht weiter zu verfolgen und uns auf die Tiefensuche zu konzentrieren. Folgende Probleme vermuten wir als Hauptgründe für das Scheitern: (1) Der Zustandsraum ist selbst kontinuierlich betrachtet sehr groß - deutlich größer als beispielsweise beim Schach; (2) aufgrund unserer begrenzten Ressourcen können wir die DQN-Architektur nicht tief/groß genug wählen und das DQN nicht lange genug trainieren; (3) Self-Play kann den Trainingsverlauf destabilisieren und benötigt für erfolgreiche Ergebnisse besonders viel Trainingszeit [12].

## 2.3 Spieltheorie

Natürlich darf bei den Ansätzen zur Lösung eines Spiels auf keinen Fall die Spieltheorie vergessen werden. Laut Wikipedia ist die Spieltheorie „eine mathematische Theorie, in der Entscheidungssituationen modelliert werden, in denen mehrere Beteiligte miteinander interagieren. Sie versucht dabei unter anderem, das rationale Entscheidungsverhalten in sozialen Konfliktsituationen davon abzuleiten“ [13] - klingt sehr passend.

Bei `Spe_ed` handelt es sich um ein Konstantsummenspiel (spieltheoretisch äquivalent zum Nullsummenspiel) mit perfekter Information. Wir kennen also das gesamte bisherige Spielgeschehen und wenn wir gewinnen, verliert der Gegner. Ein simpler Gedanke, eine Strategie für ein derartiges Spiel zu entwerfen ist, alle möglichen Verläufe aufzuzählen und den Gewinner am Ende zu bestimmen. Somit könnten wir in jeder Situation den Zug wählen, bei dem wir am Ende gewinnen.<sup>4</sup>

---

<sup>4</sup>Vorausgesetzt, die Gegner spielen nicht perfekt. Wenn alle Spieler perfekt spielen, gewinnt der mit der besten Startposition.

Da Spe\_ed aus spieltheoretischer Sicht damit dem Schach sehr ähnlich ist, erwies sich eine Analyse der dort verwendeten Techniken als sinnvoll. Die meisten Algorithmen für Schach verwenden diese Suchbäume bzw. Spielbäume in verschiedenen Formen und Kombinationen. Für uns erschien der Minimax-Algorithmus gut anwendbar.

Dabei wird ein Suchbaum aufgebaut, der alternierend Ebenen enthält in denen der eigene Spieler bzw. der gegnerische Spieler am Zug ist. Der eigene Spieler versucht seinen Gewinn zu maximieren, daher sind dies die maximierenden Levels. Die Gegner versuchen den Gewinn vom eigenen Spieler zu minimieren, dies sind daher die minimierenden Levels. [14, 15]

Versuchen wir einen Suchbaum nun am Beispiel eines 4x4 Feldes mit zwei Spielern aufzustellen. Für den ersten Zug hat jeder Spieler 5 Möglichkeiten seine Aktion zu wählen,<sup>5</sup> das ergibt  $5^2 = 25$  Spielsituationen nach Runde eins. Nun kann wieder jeder Agent aus 5 Aktionen wählen, dadurch kommen  $5^2 = 25$  Möglichkeiten zustande, in Runde zwei zu agieren. Insgesamt können also auf 25 Situationen 25 Möglichkeiten gebaut werden, für Runde zwei ergeben sich dadurch  $(5^2)^2 = 625$  Situationen. In Runde drei stehen nun 625 Situationen zur Verfügung, wieder gibt es 25 Kombinationen darauf zu agieren, also  $625 \cdot 25 = 15625$ . In Runde vier gibt es bereits 390.625 Zustände, in Runde fünf 9.765.625, in Runde  $n$ :  $(5^2)^n$ . Bei einem Spiel auf 4x4 mit zwei Spielern kann es maximal  $16/2 - 1 = 7$  Runden geben, dafür gibt es 6.103.515.625 Spielverläufe. Der Zustandsraum ist also (v.a. für grössere Felder) zu gewaltig, um alle Verläufe aufzuzählen. Allerdings erweisen sich einige Pfade relativ schnell als unbrauchbar wegen eines zu schlechten Ergebnisses in diesen Pfaden. Um diese Pfade auszuschließen und damit eine Beschleunigung zu erreichen, ist ein vielgenutztes Verfahren das Alpha-Beta-Pruning [16]. Hierzu werden zwei Werte *Alpha* und *Beta* eingeführt, die den minimalen Wert, den der maximierende Spieler bzw. den maximalen Wert, den der minimierende Spieler erreichen kann, darstellen. Sobald in einem Teilbaum klar ist, dass dieser kein besseres Ergebnis (bzw. schlechteres bei gegnerischem Spieler) erreichen kann, wird dieser ausgeschlossen. Dies ist in Abb. 2.1 dargestellt.

Trotz der Reduktion der Pfade durch Alpha-Beta-Pruning ist es meist nicht möglich, alle Pfade bis auf die unterste Tiefe zu berechnen. Der Suchbaum kann stattdessen nur bis zu einer gewissen Tiefe durchsucht werden. Allerdings steht der Sieger in dieser Tiefe oftmals noch nicht fest - wie also zwischen den Aktionen wählen? An dieser Stelle kommt nun die Bewertungsfunktion ins Spiel. So wird bspw. mittels einer Heuristik die Situation bewertet. Für Spe\_ed hat sich Voronoi als eine passende Bewertungsheuristik ergeben.

Die meisten Arbeiten beschäftigen sich mit Zwei-Spieler-Problemen. Bei Spe\_ed handelt es sich jedoch um ein Multi-Player-Problem. Perez und Oommen stellen in [18] eine einfache Erweiterung des Minimax-Algorithmus auf mehrere Spieler vor.

---

<sup>5</sup>Praktisch können es auch weniger sein, in der ersten Runde bspw. führt slow\_down zu Geschwindigkeit 0. Hier soll es allerdings um eine Worst-Case Abschätzung gehen.

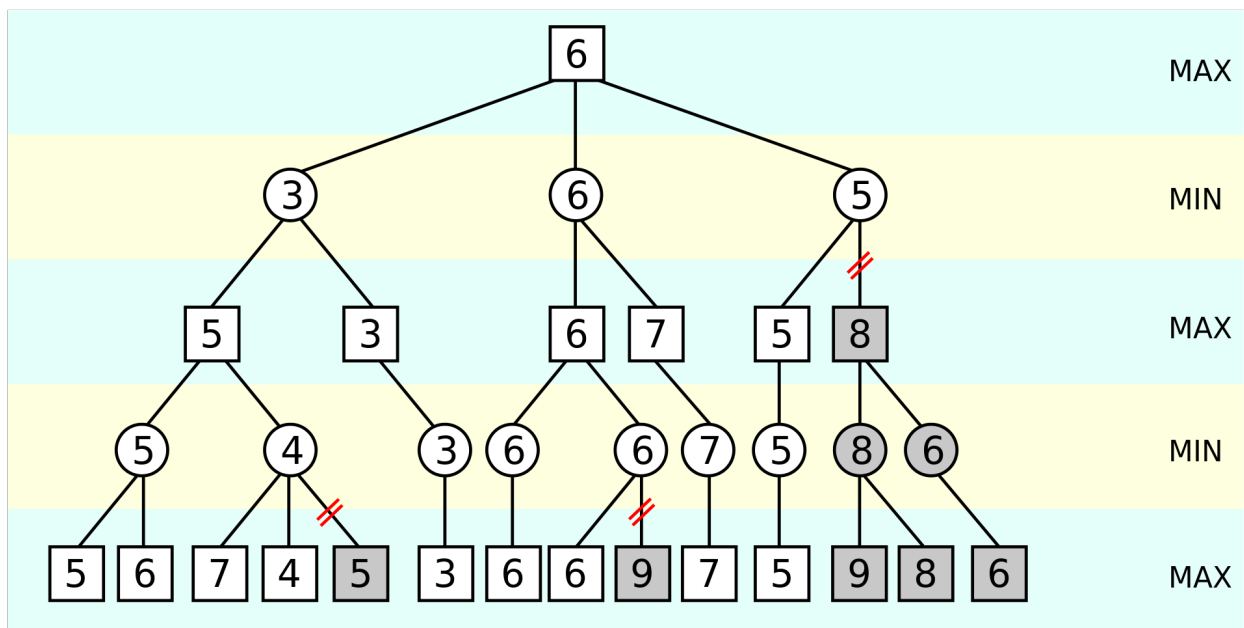


Abbildung 2.1: Minimax mit Alpha-Beta-Pruning. Graue Pfade werden nicht beachtet, da sie auf jeden Fall nichts mehr am Ergebnis verbessern. Abbildung aus [17] entnommen.

## 3 Voronoi-Basierter Multi-Minimax Algorithmus

In diesem Kapitel stellen wir den Multi-Minimax-Algorithmus und eine auf Voronoi-Diagrammen basierende Situationsbewertung vor. Zudem beschreiben wir zusätzliche Erweiterungen, die den Algorithmus bestmöglich auf `Spe_ed` anpassen.

### 3.1 Multi-Minimax

Es existieren bereits mehrere Ansätze um Minimax auf Multiplayer-Spiele zu erweitern: *Paranoid* [19], *Max<sup>n</sup>* [20], *Best Reply Search* [21] und Multi-Minimax [18]. Multi-Minimax wurde in der Arbeit von Perez und Oommen gegen die oben genannten Algorithmen am Spiel *Light Cycles* getestet. Der vorgestellte Algorithmus erzielte dabei sowohl bei 4-Spieler-Situationen als auch bei 6-Spieler-Situationen bei über 20.000 getesteten Spielen die besten Ergebnisse. Darüber hinaus bietet er uns den Vorteil, dass er bereits für simultane Spiele, also Spiele mit gleichzeitigen Zügen ausgelegt ist. Der Algorithmus läuft in  $O(nb^d)$  für  $n$  = Spieleranzahl,  $b$  = *branchingfactor*<sup>1</sup> und  $d$  = Anzahl der Spielrunden [18]. Daher verwenden wir Multi-Minimax auch für unseren Lösungsansatz.

Beim Multi-Minimax-Algorithmus tritt der maximierende Spieler immer isoliert gegen einen minimierenden Spieler an. Nachdem jeder Gegenspieler isoliert betrachtet wurde, wird die eigene Aktion basierend auf dem besten Gegenspieler gewählt. Dabei verwendet der Algorithmus Alpha-Beta Pruning und die in Kapitel 2.3 beschriebene Minimax-Routine. In Abbildung 3.1 ist ein Entscheidungsbaum der Tiefe drei abgebildet.

Die Heuristik die Perez und Oommen für die Situationsbewertung verwenden, ist simpel. Falls der Gegenspieler nicht mehr aktiv ist und der eigene Spieler noch mögliche Züge hat, die nicht zum Tod führen, wird die Situation mit dem best-möglichen Wert unendlich bewertet. Andernfalls wird die Situation mit der erreichten Suchtiefe bewertet. Diese Heuristik ist sehr praktisch für kleine Spielfelder (12x12 im Fall von Perez und Oommen), da sie wenig Rechenaufwand benötigt und somit fast alle möglichen Züge berechnet werden können. In unserem Fall ist sie aber nicht ideal, weil das Spielfeld bei `Spe_ed` bis 80x80 groß werden kann. Dafür ist eine Heuristik notwendig, die Informationen über das ganze Spielfeld liefert

---

<sup>1</sup>Anzahl der Aktionen

und einschätzt, wie gut ein Spieler momentan ist - unabhängig davon, ob er in  $x$  Zügen sterben würde. Deshalb haben wir die in Kapitel 3 vorgestellte Voronoi-Heuristik gewählt.

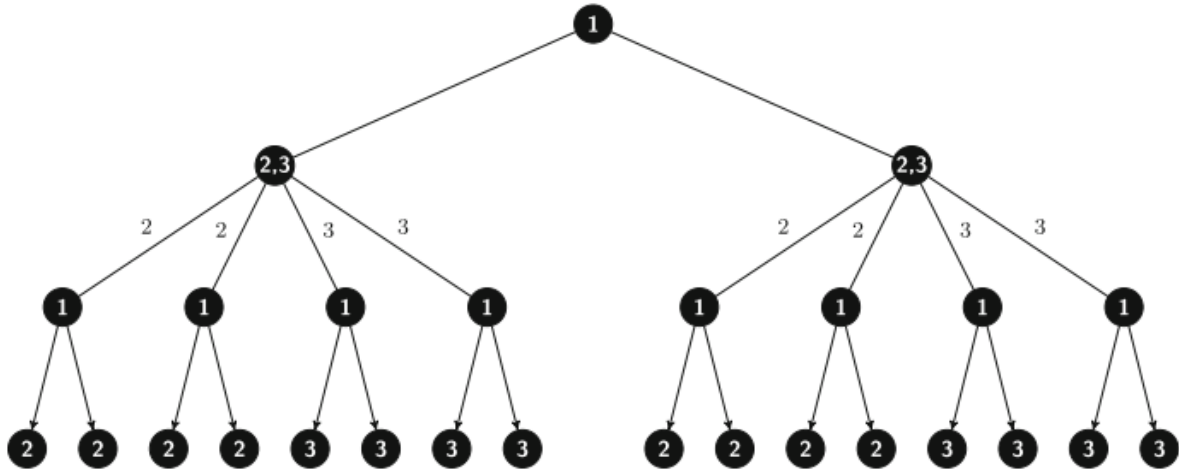


Abbildung 3.1: Multi-Minimax Repräsentation für 3 Spieler und branching-Faktor 2. Die Knoten geben an, welcher Spieler als nächstes an der Reihe ist, die Kanten repräsentieren die Aktionen. Grafik aus *Multi-Minimax: A New AI Paradigm for Simultaneously-Played Multi-player Games* [18]

## 3.2 Voronoi-Heuristik

Ein Voronoi-Diagramm (vgl. Abbildung 3.2) gibt an, welche Felder des Spielfeldes von welchem Spieler zuerst erreicht werden können. Felder die gleichzeitig erreicht werden können werden als *battlefront* bezeichnet (Lila Felder in Abbildung 3.2). Die Idee zu diesem Ansatz kam erstmals während der *Google AI Challenge*<sup>2</sup> auf [22]. Die Voronoi-Heuristik bewertet nun eine Spielsituation basierend auf der Differenz der zuerst erreichbaren Felder des eigenen Agenten und des jeweiligen Min-Agenten. Da wir bei *Spe\_ed* im Gegensatz zur *Google AI Challenge* verschiedene Geschwindigkeiten haben, müssten wir das Voronoi-Diagramm unter Betrachtung der derzeitigen Geschwindigkeiten und möglicher Geschwindigkeitsänderungen aller Spieler berechnen. Wir haben diese Erweiterung implementiert und evaluiert, sind aber zu dem Schluss gekommen, dass der gewonnene Vorteil im Vergleich zum stark erhöhten Rechenaufwand zu gering ist. Eine Erweiterung und Verbesserung der Voronoi-Heuristik für das klassische *Tron*-Game ist die *Tree-of-Chambers*-Heuristik [22, 23]. Diese ist jedoch für Spielfelder mit vielen einzelnen Lücken ungeeignet [23]. Da bei *Spe\_ed* ein zerklüftetes Spielfeld durch das regelmäßige Entstehen von Lücken häufiger vorkommt, haben wir uns schlussendlich für die Voronoi-Heuristik ohne *Tree-of-Chambers*-Erweiterung entschieden.

<sup>2</sup>[https://de.wikipedia.org/wiki/AI\\_Challenge](https://de.wikipedia.org/wiki/AI_Challenge)

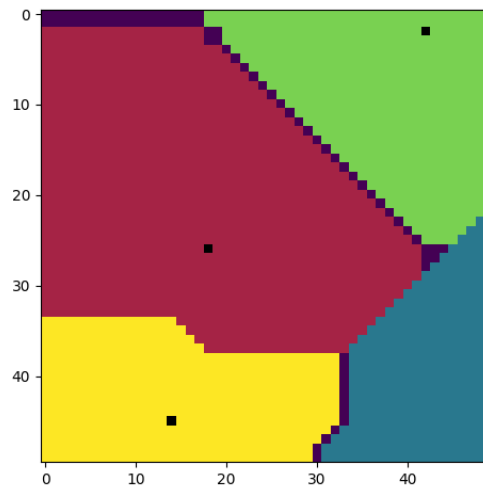


Abbildung 3.2: Voronoi-Diagramm für vier Spieler auf einem 50x50 Feld. Spielerpositionen in Schwarz dargestellt; Voronoi-Regionen der jeweiligen Spieler in Grün, Rot, Gelb und Türkis; Battlefronts in Lila.

### 3.3 Erweiterungen

Die beiden vorgestellten Grundkonzepte *Multi-Minimax* und *Voronoi-Heuristik* haben wir durch einige Erweiterungen angepasst, um sie bestmöglich für *Spe\_ed* verwenden zu können. Diese Erweiterungen werden im Folgenden vorgestellt.

#### 3.3.1 Vorsortierung der Aktionen

Beim Alpha-Beta-Pruning können mehr Pfade eliminiert werden, wenn die Aktionen für die Multi-Minimax-Suche nach ihrer Ausführungswahrscheinlichkeit vorsortiert werden. Durch Beobachtungen hat sich folgende Reihenfolge als optimal ergeben:

1. change\_nothing
2. turn\_left
3. turn\_right
4. speed\_up
5. slow\_down



### 3.3.2 Endgame-Erkennung

Wir überprüfen zu Beginn des Algorithmus ob wir uns in einem *Endgame* befinden. Wir befinden uns in einem *Endgame*, sobald wir keinen Gegner mehr direkt erreichen können, ohne dabei über eine Mauer zu springen. Ein Endgame wird wieder aufgelöst, sobald wir in den Bereich eines Gegners springen oder ein Gegner in unseren Bereich springt. Sobald wir uns in einem Endgame befinden, simulieren wir nur die eigenen Züge und ignorieren das Verhalten der Gegner. Dies erlaubt uns, eine größere Tiefe für den Multi-Minimax Algorithmus zu erreichen und somit in einer gewinnenden Position unseren verfügbaren Bereich nahezu perfekt auszufüllen und in einer verlierenden Position mehr Möglichkeiten zu evaluieren, um aus unserem Bereich heraus zuspringen und somit das Endgame aufzulösen.

### 3.3.3 Wall-Hugging

Wir fügen, wie es Andy Sloane in [22] beschreibt, einen *territory-bonus* hinzu. Dabei werden offene Felder aus der eigenen Voronoi-Region höher gewichtet als Felder mit mehr Wänden. Dies führt automatisch dazu, dass unser Agent im Falle eines Voronoi-Gleichstands eher an Wänden entlang fährt und somit seinen eigenen Platz weniger verbaut.

### 3.3.4 Reduktion der betrachteten Gegner

Da bei sechs Spielern der Berechnungsaufwand für Multi-Minimax im Vergleich zu weniger Spielern stark erhöht ist, schränken wir die betrachteten Gegenspieler ein. Wir ermitteln gleichzeitig zur Endgame Erkennung die Gegenspieler, deren Voronoi-Regionen auf unsere treffen. Wir schließen Gegenspieler aus, die wir nicht erreichen können, weil sie eine Wand von uns trennt oder ein weiterer Spieler dazwischen positioniert ist. Wir gehen davon aus, dass das Verhalten dieser Spieler für uns vorerst nicht relevant ist. Wären wir zum Beispiel in Abbildung 3.2 der Spieler mit der grünen Voronoi-Region, würden wir den Spieler mit der gelben Voronoi-Region vorerst nicht betrachten, da diese Voronoi-Regionen nicht aufeinandertreffen.

### 3.3.5 Einschränkung der eigenen Geschwindigkeit

Wir haben beobachtet, dass unser Agent bei höheren Geschwindigkeiten teilweise schlechte Entscheidungen trifft. Die Suchtiefe, die wir im Falle von knappen *Deadlines* (unter 8 Sekunden) erreichen, reicht nicht aus, um bei hohen Geschwindigkeiten Kollisionen zu vermeiden. Deshalb haben wir uns entschieden, die maximale Geschwindigkeit unseres Agenten auf drei zu beschränken. Diese Geschwindigkeit ermöglicht es uns trotzdem noch die regelmäßig entstehende Lücke geschickt zu nutzen. Wir heben diese Einschränkung auf, sobald wir uns im Endgame befinden. Dadurch kann unser Agent mehr Möglichkeiten berechnen, im Falle eines

verlierenden Endgames aus dem eigenen Bereich zu springen und eine verlierende Position zu einer gewinnenden zu drehen.

### 3.3.6 Depth-First Iterative Deepening

Für die Berechnung der nächsten Aktion wird in `Spe_ed` jeweils eine Zugzeit vorgegeben. Somit müssten wir bei der Tiefensuche die maximale Tiefe für den Suchbaum genau so wählen, dass alle Äste in der vorgegebenen Zeit abgearbeitet werden. Dies ist offensichtlich sehr fehleranfällig. Wählen wir hingegen einen Breitensuche-Ansatz, wird schnell viel Speicherplatz benötigt. Einen Kompromiss zwischen beiden Varianten bietet Depth-First Iterative Deepening (DFID). Dabei wird die Tiefensuche auf eine maximale Tiefe begrenzt, sobald diese Tiefe abgeschlossen ist, wird die maximale Tiefe erhöht und der Baum erneut durchlaufen. Das verursacht zwar, dass die oberen Bereiche des Baumes (unnötig) mehrfach berechnet werden, da der Baum nach unten allerdings exponentiell größer wird, ist dies verschmerzbar. [24]

Somit wird die gewählte Aktion mit steigender Suchtiefe inkrementell verbessert. Zum einen garantieren wir damit, dass unser Algorithmus auch bei kurzen Zugzeiten eine passende Aktion findet. Zum anderen nutzt der Algorithmus die volle Zugzeit aus und sucht bei längeren Zeiten nach Lösungen in tieferen Suchebenen. Wir starten die Berechnung der auszuführenden Aktion in einem Subprozess, während im Hauptprozess die aktuelle Serverzeit abgefragt und die verfügbare Zeit berechnet wird. Sobald die Zeit abgelaufen ist, terminiert der Hauptprozess das DFID und erhält über eine zwischen den Prozessen geteilte Variable die auszuführende Aktion. Zur Kommunikation mit dem Server verwenden wir Python's *asyncio*<sup>3</sup>.

### 3.3.7 Multiprocessing

Zur Verbesserung der Berechnungsgeschwindigkeit wird *Multiprocessing* verwendet. Dabei wird die DFID Methode genutzt, jedoch werden die Berechnungen für die verschiedenen Tiefen gleichzeitig gestartet. Dabei werden bis zu 6 Tiefen gleichzeitig berechnet, aber nicht mehr, als der Prozessor verschiedene Threads zur Verfügung stellt. Die auszuführende Aktion wird bei jedem Abschluss einer neuen Tiefe aktualisiert. Nach Ende der vorgegebenen Berechnungszeit werden alle Prozesse gestoppt und die berechnete Aktion zurückgegeben.

Außerdem wird anfangs ein Prozess erstellt, der MultiMiniMax mit Tiefe 2 und ohne Voronoi berechnet. Auf dieses Ergebnis kann zurückgegriffen werden, falls nur eine sehr kurze Berechnungszeit vorhanden ist. Dadurch kann, falls es die Spielsituation erlaubt, immer eine Aktion gefunden werden, mit der der Agent im nächsten Schritt nicht stirbt. Da die Berechnungsdauer bei größeren Tiefen allerdings exponentiell steigt, steigert das Multiprocessing

---

<sup>3</sup><https://docs.python.org/3/library/asyncio.html>

die erreichten Tiefen in der Praxis nicht und führt bei der Ausführung unter verschiedenen Betriebssystemen eventuell zu Zeitverzögerungen beim Erstellen der Prozesse. Daher verwenden wir den Ansatz nicht für unsere finale Abgabe.

### 3.3.8 Cython

Damit die Performance noch weiter verbessert werden kann, wird *Cython*<sup>4</sup> verwendet. Dadurch kann der Python-Code in schnelleren C-Code kompiliert werden. Zur einfacheren Wart- und Erweiterbarkeit wird aber darauf verzichtet statisches Typing, also die explizite Angabe des Datentyps einer Variable, zu nutzen. Somit müssen nach Erweiterung des Python-Codes keine weiteren Änderungen erfolgen, sondern die geänderten Dateien müssen lediglich neu gebaut werden. Durch das Kompilieren des Voronoi-Moduls konnte so eine Geschwindigkeitssteigerung von ca. 8% erreicht werden.

### 3.3.9 Sliding Window

Um die Komplexität des Problems zu verkleinern, wird nur ein Teil des Spielfelds betrachtet. Der Ausschnitt (Sliding Window), der betrachtet wird, wandert dabei mit der aktuellen Position unseres Agenten. Als Kompromiss zwischen Komplexität und einer trotzdem ausreichenden Spielbeschreibung hat sich eine Größe von 20 Feldern in jede Richtung als guter Wert für die Größe des Fensters herausgestellt. Hierdurch werden nur Gegner betrachtet, die sich in der Nähe unseres Agenten befinden und ihn in nächster Zeit beeinflussen können. Gegner außerhalb dieses Sliding-Window werden ignoriert, wodurch eine weitere Komplexitätsminimierung stattfindet.

Für den Fall, dass sich kein Gegner in unserem Fenster befindet, wird das Fenster so weit erweitert, sodass sich mindestens ein anderer Spieler in dem Fenster befindet. Darüber hinaus wird das Fenster um einen festen Offset erweitert.

Die Erweiterung wird umgesetzt, indem die vom Server erhaltene Spielbeschreibung bereits vor der Überführung in unser Modell passend zugeschnitten wird. Somit sind keine weiteren Änderungen an den verwendeten Algorithmen notwendig und die Erweiterung ist leicht austauschbar.

### 3.3.10 Kamikaze

Bisher erhält der Agent in der Situationsbewertung keinen Bonus dafür, Gegner zu eliminieren. Falls der Agent der eigenen Eliminierung im nächsten Schritt nicht mehr entkommen kann führt er somit eine zufällige Aktion aus. Die Erweiterung *Kamikaze* sorgt dafür, dass er in solchen Fällen, falls möglich, eine Aktion wählt, die einen Gegner eliminiert (und somit

---

<sup>4</sup><https://cython.org/>

eventuell zu einem Unentschieden statt einer Niederlage führt). Wir verallgemeinern den Bonus für die Eliminierung von Gegenspielern bewusst nicht auf alle Spielpositionen, um zu verhindern, dass der Agent die Eliminierung von Gegnern über bessere Strategien hinweg wählt.

## 4 Software

Wie bereits in Kapitel 1 erwähnt, war die Nachimplementierung/Simulation von *Spe\_ed* unser erster Implementierungsschritt. Da die eingesetzten Modellierungstechniken- und Frameworks einen maßgeblichen Einfluss auf die gesamte Architektur haben, stellen wir unser Vorgehen bei der Modellierung in diesem Kapitel genauer vor. Außerdem gehen wir auf die Punkte Software-Architektur, Software-Testing und Wartbarkeit/Erweiterbarkeit ein.

Zunächst noch ein paar allgemeinere Angaben zur Software: Wir haben uns für die Programmiersprache *Python* entschieden, da sie ein breites Spektrum an bewährten Werkzeugen für Modellierung, RL und Datenanalyse bietet. Wir verwenden die Python-Version 3.7, unsere Implementierungen sind streng nach den Coding Conventions des *PEP 8 Style Guide*<sup>1</sup> formatiert und *Core*-Klassen und -Funktionen sind durchgehend mit *Docstrings*<sup>2</sup> dokumentiert.

### 4.1 Architektur und Modellierung

Die Software-Architektur unseres Projektes ist in die Software-Pakete *Core*, *Evaluation* und *Scripts* geteilt. *Core* stellt den funktionalen Kern unseres Projekts dar. Es enthält das *Spe\_ed*-Modell und die implementierten Spieler-Algorithmen (*Agents*). *Evaluation* beinhaltet unsere Auswertungssoftware und *Scripts* enthält beispielsweise Skripte zur Online-Ausführung unserer Agenten. Abbildung 4.1 zeigt den beschriebenen Aufbau und die Abhängigkeitshierarchie zwischen den Software-Paketten.

Unsere grundlegende Idee hinter der Modellierung von *Spe\_ed* ist, dass das Modell wie eine Black-Box verwendet werden kann, deren Schnittstellen mit denen des originalen Spiels übereinstimmen. Die Input- und Output-Formate des Modells entsprechen daher exakt den JSON-Formaten des originalen Spiels. Somit kann eine entwickelte Spieler-Software ohne Änderungen sowohl im Modell als auch im Original ausgeführt werden.

Als Modellierungsansatz haben wir uns für Agent-Based Modelling (ABM) entschieden. ABM verfolgt einen *Bottom-Up*-Ansatz zur Modellierung von Simulationen. Das bedeutet, dass der Ablauf der Simulation nicht zentral von 'oben' herab gesteuert wird, sondern dass die autonomen Individuen der Simulation (genannt *Agenten*) den Simulationsverlauf durch ihr jeweiliges Verhalten von 'unten' aus bestimmen. Die zentralen Komponenten von ABM sind (1) mehrere Agenten und (2) die Umgebung, in der sich die Agenten befinden. ABM

---

<sup>1</sup><https://www.python.org/dev/peps/pep-0008/>

<sup>2</sup><https://www.python.org/dev/peps/pep-0257/>

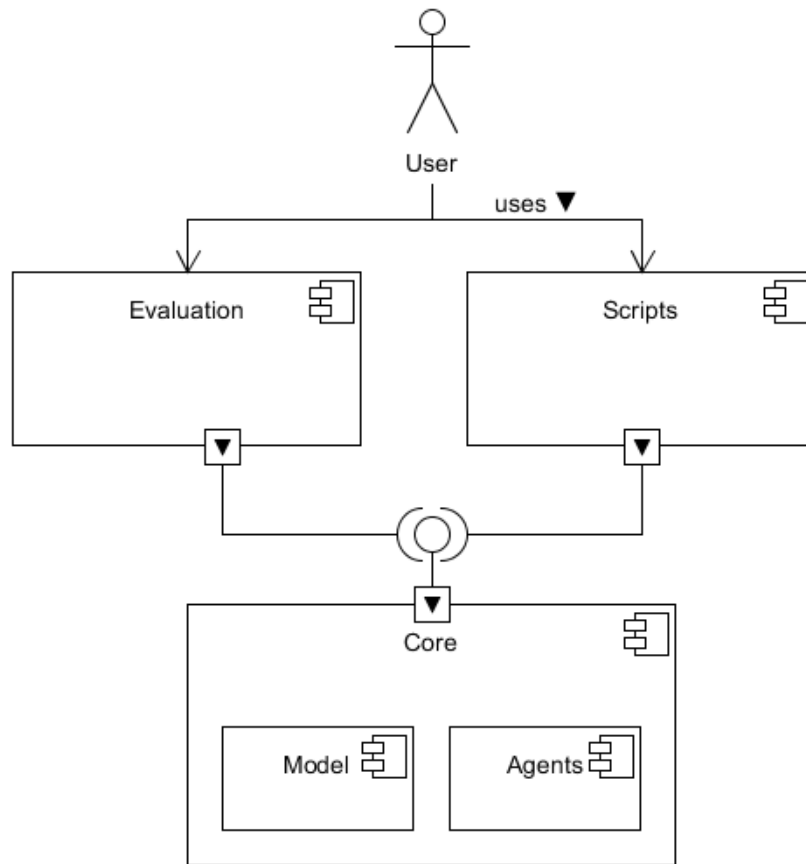


Abbildung 4.1: Komponentendiagramm unseres Projekts

eignet sich besonders gut zur Modellierung und Simulation von *Spe\_ed*, da *Spe\_ed* ein Multi-Agenten-System ist, in dem die Agenten ein eigenes - nicht zentral vorgegebenes - Verhalten besitzen. Bei Simulationen wird zudem zwischen *ereignisbasierten* (auch *asynchronen*) und *zeitorientierten* (auch *synchronen*) Modellen unterschieden. Bei *Spe\_ed* handelt es sich um ein System in dem alle Agenten in festen Zeitschritten eine Entscheidung treffen. Deshalb ist es sinnvoll ein zeitorientiertes Modell zu wählen. [25, 26]

Um unsere Software möglichst effizient, wartbar und leicht verständlich zu gestalten nutzen wir ein Simulationsframework. Auf Basis der oben genannten Kriterien haben wir *Mesa* [27] als Framework für die Implementierung unseres *Spe\_ed*-Modells gewählt. *Mesa* wurde 2015 als erstes ABM-Framework für Python veröffentlicht. Es ist *open-source* und orientiert sich an anderen ABM-Tools und -Frameworks, wie *NetLogo* [28], *Repast* [29] und *MASON* [30]. *Mesa* besitzt zudem einen leichtgewichtigen und somit effizienten Framework-Kern. Das Framework ordnet seine Komponenten drei verschiedenen Modulen zu: Modellierung, Visualisierung und Analyse. Wir beschäftigen uns im Folgenden zwar lediglich mit der Modellierung, allerdings ist es nennenswert, dass wir unser Modell durch die Nutzung von *Mesa* ohne beachtlichen Mehraufwand web-basiert visualisieren und analysieren können.

Das Modellierungsmodul von Mesa implementiert den ABM-Ansatz. Es besteht aus den Basisklassen *Model*, *Agent*, *Scheduler* und *Space*. Die *Model*-Klasse enthält Modellparameter und steuert (startet, stoppt, etc.) den Ablauf einer Simulation. In der *Agent*-Klasse wird definiert, welche Prozesse ein Agent bei seiner Aktivierung durchläuft - die Aktivierung eines Agenten entspricht bei uns der Ausführung eines Zuges. Der *Scheduler* verwaltet die Simulationszeit. Er kennt alle Agenten und aktiviert diese in jedem Zeitschritt. Die *Space*-Klasse beschreibt den Raum, in dem sich die Agenten befinden. Mesa bietet von Haus aus mehrere Unterklassen der *Scheduler*- und *Space*-Klassen an. In unserem Anwendungsfall wird der *SimultaneousActivation*-Scheduler und der *MultiGrid*-Space verwendet.

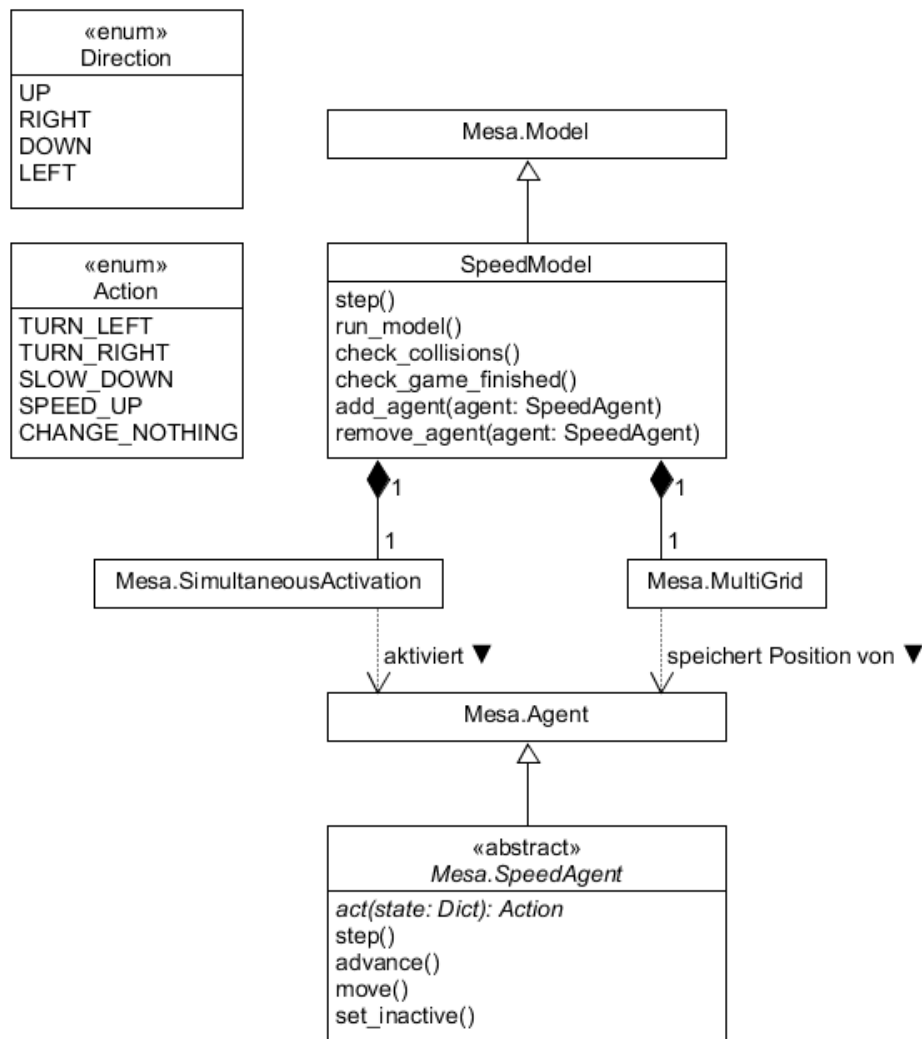


Abbildung 4.2: Vereinfachtes Klassendiagramm der Modell-Architektur

Abbildung 4.2 zeigt ein vereinfachtes Klassendiagramm unserer Modell-Architektur. Klassen die Bestandteil von Mesa sind, sind mit dem Präfix *Mesa.* versehen. Im Mittelpunkt stehen

die Klassen *SpeedModel* und *SpeedAgent*. Diese sind eigens implementiert und beschreiben die Funktionsweise des Modells. Die *SpeedAgent*-Klasse ist abstrakt und besitzt die abstrakte Methode *act(state)*. Hier haben wir die oben genannte Black-Box-Schnittstelle platziert. Der Übergabeparameter *state* entspricht dem JSON-Format eines Spielzustands von *Spe\_ed* und der Rückgabewert ist die Aktion für die sich der Spieler entschieden hat. Um einen bestimmten Agenten zu entwerfen muss also lediglich eine Klasse erstellt werden, die von *SpeedAgent* erbt und die *act*-Methode implementiert. Diese Struktur macht unsere Software leicht um neue Agenten erweiterbar.

Um die einfache Verwendung unserer Software zu demonstrieren, zeigen wir in Abbildung 4.3 wie beispielsweise ein Spiel mit einem *RandomAgent* (Agent der zufällige Aktionen trifft) und einem *MultiMinimaxAgent* (siehe Kapitel 3) gestartet wird. Das Beispiel verdeutlicht außerdem, dass Modellparameter wie die Spielfeldgröße variabel und leicht einstellbar sind.

```
from src.model import SpeedModel
from src.agents import MultiMiniMaxAgent, RandomAgent

if __name__ == "__main__":
    agents = [RandomAgent, MultiMiniMaxAgent]
    model = SpeedModel(width=60, height=60, nb_agents=2, agent_classes=agents)
    model.run_model()
```

Abbildung 4.3: Code-Beispiel zur Erstellung und Ausführung des Modells

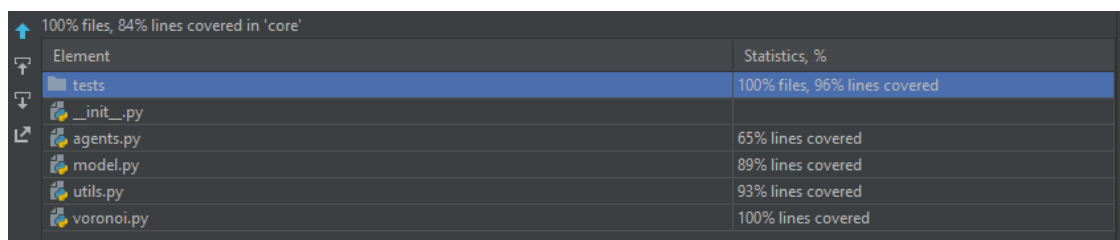
## 4.2 Software-Testing

Das Schreiben von Software-Tests war für uns aus zwei Gründen besonders wichtig. Zum einen mussten wir sicher gehen können, dass unser Modell mit dem originalen Spiel übereinstimmt. Zum anderen war es wichtig, dass wir Erweiterungen und Optimierungen an den algorithmischen Ansätzen leicht validieren können. Die Tests sind - wie bei *Unit Tests* in Python standardmäßig vorgesehen - in Software-Paketen namens *tests*, auf gleicher Ebene wie das zu testende Modul, eingebettet.

Beim Testen des Modells kommt uns erneut die Black-Box-Eigenschaft zugute. Wir verwenden JSON-Dateien von gespeicherten Original-Spielen und gleichen in den Tests die Spielzustände zu jedem Zeitschritt mit denen unseres Modells ab. Dabei laden wir zuerst den initialen Spielzustand eines gespeicherten Spiels in das Modell, extrahieren dann die Aktionen der Spieler-Agenten zu den jeweiligen Zeitschritten und simulieren diese Aktionen dann auf unserem Modell. Wenn das Modell alle 58 gespeicherten Spiele korrekt simuliert, ist der Modell-Test bestanden. Auf diese Weise erhalten wir deutlich umfangreichere Tests, als durch manuelles Erstellen von Test-Spielzuständen möglich wäre.



Die implementierten Agenten und deren Algorithmen (Multi-Minimax, Voronoi, etc), werden mit Unit-Tests abgedeckt. Dabei testen wir, insbesondere ob die Agenten bei ausreichender Suchtiefe in verschiedenen Spielsituationen den besten Zug finden. Zu allen übrigen *Core*-Softwaremodulen (wie z.B. *utils*) haben wir ebenfalls Unit Tests geschrieben. Wie in Abbildung 4.4 zu sehen erreichen wir insgesamt eine Testabdeckung von 84% für das *Core*-Softwaremodul.



100% files, 84% lines covered in 'core'	
Element	Statistics, %
tests	100% files, 96% lines covered
__init__.py	
agents.py	65% lines covered
model.py	89% lines covered
utils.py	93% lines covered
voronoi.py	100% lines covered

Abbildung 4.4: Testabdeckung für das *Core*-Softwaremodul

## 5 Ergebnisse

Um unsere Software auszuwerten haben wir sie auf dem Live-Server und mit Hilfe unseres Modells getestet. In den folgenden Unterkapiteln stellen wir die resultierenden Ergebnisse vor und interpretieren ihre Bedeutung.

### 5.1 Live-Server

Wir haben über den Verlauf des Wettbewerbs verschiedene Versionen unserer Lösung auf dem Live-Server antreten lassen. Zum einen um unsere eigene Lösung zu evaluieren und zum anderen um Informationen über das Spiel und über die Gegenspieler zu sammeln. Abbildung 5.1 zeigt die Ergebnisse der gesammelten Informationen über die Spielfeldbreite, Spielfeldhöhe und die verfügbare Zeit pro Zug. In zwei beobachteten Spielen hatte das Spielfeld eine Größe von 80x80. Da wir weder für die Breite noch für die Höhe des Spielfelds jemals einen Wert  $> 80$  beobachtet haben, können wir annehmen, dass es sich bei 80x80 um die maximale Spielfeldgröße handelt. Das kleinste von uns beobachtete Spielfeld hatte eine Größe von 43x41. Dadurch können wir approximiert von einem kleinsten Spielfeld der Größe 41x41 ausgehen. Die von uns ermittelte durchschnittliche Zugzeit beträgt  $\sim 10$  Sekunden. Unter Berücksichtigung der Latenz nehmen wir an, dass die Zugzeiten Zufallszahlen zwischen  $\sim 5$  und  $\sim 15$  Sekunden sind.

Da unserer Lösungsansatz sowohl mit variablen Spielfeldgrößen als auch mit variablen Zugzeiten umgehen kann, haben diese Erkenntnisse keinen direkten Einfluss auf unseren Lösungsansatz. Jedoch konnten wir die gesammelten Informationen vor allem für eigene Offline-Auswertungen verwenden, um den Live-Server so gut wie möglich nachzubilden und unsere Ansätze mit verschiedenen Parametern darauf zu testen.

Wir haben über die letzten beiden Wochen des Wettbewerbes auf dem zur Verfügung gestellten Live-Server eine in 120 Spielen eine Gewinnrate von 82% erzielt. Fragwürdig ist jedoch wie aussagekräftig dieser Wert ist, da wir jeweils verschiedene Versionen unserer Lösung getestet haben, und wir bei vielen der Spielen vermuten, dass es sich bei unseren Gegnern vermehrt um Bots der Wettbewerbsveranstalter und nicht um andere Teams gehandelt hat.

Metrik	Höhe	Breite	Zugzeit
Minimum	41	41	3,26s
Maximum	80	80	14,94s
Mittelwert	60.13	61	9,66s

Tabelle 5.1: Auswertung der Spiel-Parameter

## 5.2 Offline-Versionsauswertung

Um aus den verschiedenen Versionen unserer Agenten eine finale Version für die Abgabe zu bestimmen haben wir ausgewählte Versionen im Modell gegeneinander spielen lassen. Im Folgenden werden die verwendeten Versionen beschrieben, wobei sich die Erweiterungen auf Kapitel 3.3 beziehen:

- **Multi-Minimax:** Multi-Minimax-Algorithmus [18] ohne weitere Erweiterungen
- **V-Multi-Minimax:** Multi-Minimax mit Voronoi-basierter Positionsauswertung und den Erweiterungen *Cython*, *Wall-Hugging*, *Vorsortierung der Aktionen*, *Endgame Erkennung*, *Einschränkung der eigenen Geschwindigkeit*, *DFID* und *Kamikaze*
- **RG-V-Multi-Minimax:** V-Multi-Minimax mit der Erweiterung *Reduktion der betrachteten Gegner*
- **SW-RG-V-Multi-Minimax V1:** RG-V-Multi-Minimax mit der Erweiterung *Sliding Window* mit Fenstergröße 40 und Offset 5
- **SW-RG-V-Multi-Minimax V2:** RG-V-Multi-Minimax mit der Erweiterung *Sliding Window* mit Fenstergröße 30 und Offset 3

Damit die Erkenntnisse der Auswertungen im Modell möglichst gut auf die Live-Instanz von *Spe\_ed* übertragbar sind, haben wir die Auswertungsparameter an den Ergebnissen aus Tabelle 5.1 orientiert. Dabei muss bei den beobachteten Zugzeiten noch die mögliche Latenz berücksichtigt werden. Tabelle 5.2 zeigt die gewählten Auswertungsparameter. Wir verwenden ein Spielfeld der Größe 60x60 und wählen eine Zugzeit zufällig zwischen 5 und 15 Sekunden. Da Spiele auf Feldern dieser Größe häufig sehr lange dauern (im Schnitt ca. 6 Stunden), ist es leider nicht möglich eine hohe Anzahl an Versuchswiederholungen durchzuführen. *Spe\_ed* ist allerdings ein stark initialisierungsabhängiges Spiel - Spieler mit schlechten Startpositionen haben geringere Gewinnchancen. Um die Aussagekraft der Ergebnisse bei wenigen Versuchswiederholungen trotzdem hoch zu halten, initialisieren wir die Positionen nicht zufällig. Stattdessen generieren wir für alle 5 Agenten für jeweils 5 Spiele Positionen und rotieren diese innerhalb der 5 Spiele durch, sodass jeder Agent einmal von jeder Startposition aus startet. Agent 1 erhält also in Spiel 2 die Position die Agent 2 in Spiel 1 hatte und Agent 2 die alte Position von Agent 3 usw. Nach 5 gespielten Partien werden neue Positionen generiert. Somit ist die Initialisierung fair und die Auswertungen sind bereits bei geringer Anzahl an Versuchswiederholungen aussagekräftig.

Höhe	Breite	# Spieler	Zugzeit	Versuchswiederholungen
60	60	5	rand(5, 15)	60

Tabelle 5.2: Auswertungsparameter

Algorithmus	Gewinnrate	Platzierung
Multi-Minimax	0%	4,22
V-Multi-Minimax	11,67%	3,67
RG-V-Multi-Minimax	25%	2,62
SW-RG-V-Multi-Minimax V1	35%	2,3
SW-RG-V-Multi-Minimax V2	28,33%	2,1

Tabelle 5.3: Durchschnittswerte der Auswertung verschiedener Versionen in 60 Spielen

Tabelle 5.3 zeigt die Ergebnisse der Auswertung. Dargestellt sind die Gewinnraten der einzelnen Agenten sowie deren durchschnittliche Platzierung. Die Platzierung ergibt sich über die Ausscheidungsreihenfolge, wobei der Gewinner Platz 1 erhält.<sup>1</sup> Es ist eindeutig erkennbar, dass sich die Voronoi-basierte Situationsauswertung auszahlt, da alle Voronoi-basierten Algorithmen den reinen Multi-Minimax-Algorithmus weit übertreffen. Die Erweiterungen *Sliding Window*- und *Reduktion der betrachteten Gegner* zeigen zudem wiederum eine Verbesserung im Vergleich zum V-Multi-Minimax-Agent. Das liegt daran, dass sie es schaffen die Komplexität der Situationsauswertung zu reduzieren, um eine höhere Suchtiefe zu erreichen, ohne relevante Informationen über die Umgebung zu vernachlässigen. Die beiden SW-RG-V-Multi-Minimax-Varianten schneiden am besten ab. Grund dafür ist vermutlich, dass sie im Schnitt tiefere Suchebenen als RG-V-Multi-Minimax erreichen und daher auch bei geringen Zugzeiten gut durchdachte Aktionen wählen. Interessanterweise erreicht Variante 2 (V2) der SW-RG-V-Multi-Minimax-Agenten im Schnitt eine leicht bessere Platzierung (2,1) als Variante 2 (2,3), obwohl Variante 1 eine höhere Gewinnrate erzielt (35% zu 28,33%). Das lässt vermuten, dass Variante 1 noch aggressiver gespielt hat, um wenn möglich einen Sieg zu erringen. Da die Wettbewerbsveranstalter die Gewinnrate als Hauptauswertungskriterium ausgeben<sup>2</sup> erachten wir SW-RG-V-Multi-Minimax V2 als unseren besten Agenten und verwenden diesen in der finalen Abgabe.

<sup>1</sup>Bei Ausscheidung in der gleichen Runde wird eine Platzierung geteilt - beide erhalten die bessere Platzierung.

<sup>2</sup><https://github.com/informatiCup/InformatiCup2021/issues/15>

## 6 Diskussion

Auf unseren Weg zur in Kapitel 3 vorgestellten Lösung sind wir mit den drei Möglichkeiten Trajektorienplanung, RL und Spieltheorie gestartet. Nachdem Trajektorienplanung kaum Strategie entwickeln konnte und RL ebenfalls keine beachtlichen Erfolge vorweisen konnte, intensivierten wir einen Minimax-Ansatz aus der Spieltheorie - auch weil dieser in ähnlichen Spielen wie bspw. Schach verwendet wird. Der Ansatz wurde zum Multi-Minimax-Algorithmus von Perez und Oommen erweitert und wiederum durch Voronoi-Evaluierung sowie weitere Heuristiken ergänzt.

Die zugehörige Python-Software implementiert die genannten Lösungen nach dem ABM-Schema. Wichtige Bestandteile der Software werden mit Tests abgedeckt. Die verschiedenen Lösungen und Erweiterungen wurden mithilfe des Modells untereinander verglichen und ausgewertet. Auswertungen am Live-Server ergaben hohe Gewinnraten ( 85%).

In der vorliegenden Arbeit werden state-of-the-art Algorithmen aus dem Bereich Spieltheorie verwendet und für das Spiel `Spe_ed` erweitert und optimiert. Der modulare und generische Code lässt sich gut erweitern. Mit der *Cython*-Erweiterung wird ein Geschwindigkeitsboost erreicht, allerdings ließe sich die Geschwindigkeit (und damit die erreichte Suchtiefe) bei Verwendung einer anderen Programmiersprache vermutlich noch einmal deutlich steigern. Die Begrenzung der maximalen Geschwindigkeit des Agenten kann zudem dazu führen, dass gute Strategien ausgeschlossen werden.

Zukünftige Arbeit an dieser Problemstellung könnte die Implementierung des Voronoi-Multi-Minimax Algorithmus in *C++* sein. Zudem könnten ausführliche Parameterstudien zur Fenstergröße unseres SW-RG-V-Mult-Minimax-Algorithmus die Performance noch leicht steigern. Mit viel Rechenpower und ausreichend Zeit könnte sich zudem ein RL-Ansatz lohnen und bessere Resultate erzielen. Ein Vorteil des RL ist, dass die Ausführung eines trainierten Modells in der Regel schnell vonstatten geht.

# Abkürzungsverzeichnis

**ABM** Agent-Based Modelling.

**APF** Artificial Potential Field.

**CNN** Convolutional Neural Network.

**DFID** Depth-First Iterative Deepening.

**DNN** Deep Neural Network.

**DQN** Deep Q-Network.

**EAPF** Evolutionary Artificial Potential Field.

**MDP** Markov Decision Process.

**MLP** Multilayer Perceptron.

**RL** Reinforcement Learning.

**RRT** Rapidly-exploring Random Tree.

**VFH** Vector Field Histogram.

# Literaturverzeichnis

- [1] BORENSTEIN, J. ; KOREN, Y.: The vector field histogram-fast obstacle avoidance for mobile robots. In: *1042-296X* 7, Nr. 3, S. 278–288. <http://dx.doi.org/10.1109/70.88137>. – DOI 10.1109/70.88137. – ISSN 1042–296X
- [2] S. KARAMAN ; M. R. WALTER ; A. PEREZ ; E. FRAZZOLI ; S. TELLER: Anytime Motion Planning using the RRT\*. In: *2011 IEEE International Conference on Robotics and Automation*, 2011, S. 1478–1483
- [3] GARRIDO, S. ; MORENO, L. ; ABDERRAHIM, M. ; MARTIN, F.: Path Planning for Mobile Robot Navigation using Voronoi Diagram and Fast Marching. In: *2006 IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2006, S. 2376–2381
- [4] RAJA, P. ; PUGAZHENTHI, S.: Optimal path planning of mobile robots: A review. In: *International Journal of the Physical Sciences* 7 (2012). <http://dx.doi.org/10.5897/IJPS11.1745>. – DOI 10.5897/IJPS11.1745
- [5] MNIH, Volodymyr ; KAVUKCUOGLU, Koray ; SILVER, David ; GRAVES, Alex ; ANTONOGLOU, Ioannis ; WIERSTRA, Daan ; RIEDMILLER, Martin: *Playing Atari with Deep Reinforcement Learning*. <http://arxiv.org/pdf/1312.5602v1>
- [6] SILVER, David ; HUBERT, Thomas ; SCHRITTWIESER, Julian ; ANTONOGLOU, Ioannis ; LAI, Matthew ; GUEZ, Arthur ; LANCTOT, Marc ; SIFRE, Laurent ; KUMARAN, Dharmashan ; GRAEPEL, Thore ; LILICRAP, Timothy ; SIMONYAN, Karen ; HASSABIS, Demis: A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play. In: *Science (New York, N.Y.)* 362 (2018), Nr. 6419, S. 1140–1144. <http://dx.doi.org/10.1126/science.aar6404>. – DOI 10.1126/science.aar6404
- [7] OPENAI ; : ; BERNER, Christopher ; BROCKMAN, Greg ; CHAN, Brooke ; CHEUNG, Vicki ; DEBIAK, Przemysław ; DENNISON, Christy ; FARHI, David ; FISCHER, Quirin ; HASHME, Shariq ; HESSE, Chris ; JÓZEFOWICZ, Rafal ; GRAY, Scott ; OLSSON, Catherine ; PACHOCKI, Jakub ; PETROV, Michael ; PINTO, HENRIQUE PONDÉ DE OLIVEIRA ; RAIMAN, Jonathan ; SALIMANS, Tim ; SCHLATTER, Jeremy ; SCHNEIDER, Jonas ; SIDOR, Szymon ; SUTSKEVER, Ilya ; TANG, Jie ; WOLSKI, Filip ; ZHANG, Susan: *Dota 2 with Large Scale Deep Reinforcement Learning*. <https://arxiv.org/pdf/1912.06680>
- [8] SUTTON, Richard S. ; BARTO, Andrew G.: *Reinforcement learning: An introduction*. [Nachdr.]. Cambridge, Mass. : MIT Press, 1998 (A Bradford book). – ISBN 9780262193986

- 
- [9] BROCKMAN, Greg ; CHEUNG, Vicki ; PETTERSSON, Ludwig ; SCHNEIDER, Jonas ; SCHULMAN, John ; TANG, Jie ; ZAREMBA, Wojciech: *OpenAI Gym*. <https://arxiv.org/pdf/1606.01540>
- [10] WATKINS, Christopher J. C. H. ; DAYAN, Peter: Q-learning. In: *Machine Learning* 8 (1992), Nr. 3-4, S. 279–292. <http://dx.doi.org/10.1007/BF00992698>. – DOI 10.1007/BF00992698. – ISSN 0885–6125
- [11] MNIH, Volodymyr ; KAVUKCUOGLU, Koray ; SILVER, David ; RUSU, Andrei A. ; VENESS, Joel ; BELLEMARE, Marc G. ; GRAVES, Alex ; RIEDMILLER, Martin ; FIDJELAND, Andreas K. ; OSTROVSKI, Georg ; PETERSEN, Stig ; BEATTIE, Charles ; SADIK, Amir ; ANTONOGLOU, Ioannis ; KING, Helen ; KUMARAN, Dharshan ; WIERSTRA, Daan ; LEGG, Shane ; HASSABIS, Demis: Human-level control through deep reinforcement learning. In: *Nature* 518 (2015), Nr. 7540, S. 529–533. <http://dx.doi.org/10.1038/nature14236>. – DOI 10.1038/nature14236
- [12] BAI, Yu ; JIN, Chi ; YU, Tiancheng: *Near-Optimal Reinforcement Learning with Self-Play*. <http://arxiv.org/pdf/2006.12007v2>
- [13] <https://de.wikipedia.org/wiki/Spieltheorie>
- [14] SHANNON, Claude E.: XXII. Programming a computer for playing chess. In: *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science* 41 (1950), Nr. 314, S. 256–275
- [15] CAMPBELL, Murray S. ; MARSLAND, T. A.: A comparison of minimax tree search algorithms. In: *Artificial Intelligence* 20 (1983), Nr. 4, S. 347–367. – ISSN 0004–3702
- [16] KNUTH, Donald E. ; MOORE, Ronald W.: An analysis of alpha-beta pruning. In: *Artificial Intelligence* 6 (1975), Nr. 4, S. 293–326. – ISSN 0004–3702
- [17] [https://en.wikipedia.org/wiki/Alpha%E2%80%93beta\\_pruning](https://en.wikipedia.org/wiki/Alpha%E2%80%93beta_pruning)
- [18] PEREZ, Nicolas ; OOMMEN, B. J.: Multi-Minimax: A New AI Paradigm for Simultaneously-Played Multi-player Games. Version: 2019. [http://dx.doi.org/10.1007/978-3-030-35288-2\\_4](http://dx.doi.org/10.1007/978-3-030-35288-2_4). In: BIRUKOU (Hrsg.) ; LIU (Hrsg.): *AI 2019: Advances in Artificial Intelligence* Bd. 11919. Cham : Springer International Publishing, 2019. – DOI 10.1007/978-3-030-35288-2\_4. – ISBN 978-3-030-35287-5, S. 41–53
- [19] STURTEVANT, Nathan: Current Challenges in Multi-player Game Search. Version: 2006. [http://dx.doi.org/10.1007/11674399\\_20](http://dx.doi.org/10.1007/11674399_20). In: *Computers and Games* Bd. 3846. Berlin, Heidelberg : Springer Nature, 2006. – DOI 10.1007/11674399\_20. – ISBN 978-3-540-32488-1, S. 285–300
- [20] LUCKHARDT, Carol ; IRANI, Keki: *An Algorithmic Solution of N-Person Games*. 1986 <https://www.aaai.org/papers/aaai/1986/aaai86-025.pdf>



- [21] SCHADD, Maarten P. D. ; WINANDS, Mark H. M.: Best Reply Search for Multiplayer Games. In: *IEEE Transactions on Computational Intelligence and AI in Games* 3 (2011), Nr. 1, S. 57–66. <http://dx.doi.org/10.1109/tciaig.2011.2107323>. – DOI 10.1109/tciaig.2011.2107323. – ISSN 1943–068X
- [22] ANDY SLOANE: *Google AI Challenge post-mortem*. <https://www.aik0n.net/2010/03/04/google-ai-postmortem.html>. Version: 2010
- [23] KANG, Lukas: *Endgame Detection in Tron*. 2012
- [24] KORF, Richard E.: Depth-first iterative-deepening. In: *Artificial Intelligence* 27 (1985), Nr. 1, 97–109. [http://dx.doi.org/10.1016/0004-3702\(85\)90084-0](http://dx.doi.org/10.1016/0004-3702(85)90084-0). – DOI 10.1016/0004-3702(85)90084-0. – ISSN 0004–3702
- [25] BERNHARDT, Ken: Agent-based modeling in transportation. In: *Artificial Intelligence in Transportation* (2007), S. 72–80
- [26] MACAL, Charles M. ; NORTH, Michael J.: Agent-based modeling and simulation. In: ROSSETTI, Manuel D. (Hrsg.): *Proceedings of the 2009 Winter Simulation Conference*. Piscataway, NJ : IEEE, 2009. – ISBN 978–1–4244–5770–0, S. 86–98
- [27] MASAD, David ; KAZIL, Jacqueline: Mesa: An Agent-Based Modeling Framework. In: *Proceedings of the 14th Python in Science Conference, SciPy, 2015* (Proceedings of the Python in Science Conference), S. 51–58
- [28] WILENSKY, Uri ; TISUE, Seth: NetLogo: Design and Implementation of a Multi-Agent Modeling Environment. (2004)
- [29] NORTH, Michael J. ; COLLIER, Nicholson T. ; OZIK, Jonathan ; TATARA, Eric R. ; MACAL, Charles M. ; BRAGEN, Mark ; SYDELKO, Pam: Complex adaptive systems modeling with Repast Symphony. In: *Complex Adaptive Systems Modeling* 1 (2013), Nr. 1, S. 1035. <http://dx.doi.org/10.1186/2194-3206-1-3>. – DOI 10.1186/2194-3206-1-3
- [30] LUKE, Sean ; CIOFFI-REVILLA, Claudio ; PANAIT, Liviu ; SULLIVAN, Keith ; BALAN, Gabriel: MASON: A Multiagent Simulation Environment. In: *SIMULATION* (2005), S. 517–527. – ISSN 0037–5497