Untersuchung von selbstlernenden Reinforcement Learning Agenten im computergenerierten Spiel Yavalath

Abschlussarbeit zur Erlangung des akademischen Grades
Bachelor of Science im Studiengang Informatik
an der Fakultät für Informatik und Ingenieurwissenschaften
der Technischen Hochschule Köln

vorgelegt von:	Ann Weitz
Matrikel-Nr.:	011 107 450
Adresse:	Am Sandberg 28
	51643 Gummersbach
	annweitz@posteo.net

eingereicht bei:	Prof. Dr. Wolfgang Konen
Zweitgutachter/in:	Prof. Dr. Boris Naujoks

Gummersbach, 19.04.2022

Erklärung

Ich versichere, die von mir vorgelegte Arbeit selbstständig verfasst zu haben. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder nicht veröffentlichten Arbeiten anderer oder der Verfasserin/des Verfassers selbst entnommen sind, habe ich als entnommen kenntlich gemacht. Sämtliche Quellen und Hilfsmittel, die ich für die Arbeit benutzt habe, sind angegeben. Die Arbeit hat mit gleichem Inhalt bzw. in wesentlichen Teilen noch keiner anderen Prüfungsbehörde vorgelegen.

Gummersbach, 19.04.2022 Ort, Datum

Rechtsverbindliche Unterschrift

Kurzfassung

Diese Bachelorarbeit soll den Erfolg von unterschiedlich parametrisierten Reinforcement Learning Agenten innerhalb des General Board Game (GBG) Playing and Learning Frameworks für das Spiel Yavalath untersuchen. Erst wird ein Überblick über das GBG Framework und die darin enthaltenen Spiele sowie Agenten gegeben, bevor dann noch das Spiel Yavalath erläutert wird.

Im Hauptteil der Arbeit wird zuerst die Methode vorgestellt, mit der die Agenten evaluiert werden, bevor dann verschiedene Agenten parametrisiert und getestet werden. Relativ schnell konnte dadurch herausgestellt werden, dass Yavalath sehr gut durch Reinforcement Learning lernbar ist. Durch weiteres Verfeinern der Parameter, sowie Einsatz verschiedener Methoden im Reinforcement Learning, konnte danach der Lernerfolg noch deutlich verschnellert werden.

Abschließend werden noch verschiedene Werke der Literatur erläutert, die sich ebenfalls mit dem Themengebiet Lernen im Spiel Yavalath befasst haben und es wird ein Ausblick auf Bereiche gegeben, die in dem Umfang dieser Arbeit leider nicht angegangen werden konnten.

Inhalt

Erklärung ..... **I**
Kurzfassung ..... **II**
Tabellenverzeichnis ..... **V**
Abbildungsverzeichnis ..... **VI**
1 Einleitung **8**
1.1 Motivation 8
2 General Board Game Playing Framework (GBG) **10**
2.1 Überblick 10
2.2 Spiele 10
2.3 Agenten 11
2.3.1 Human 11
2.3.2 Random 11
2.3.3 Max-N 11
2.3.4 Monte Carlo Tree Search 12
2.3.5 Temporal Difference 13
2.3.6 Temporal Difference (N-Tupel) 15
3 Yavalath **16**
3.1 Hintergrund 16
3.2 Regeln 16
3.2.1 3-Spieler-Variante 17
3.2.2 First Move Advantage und die Swap-Regel 18
3.3 Strategien 18
3.3.1 Der Forcing-Move 18
3.3.2 Starke Muster 19
3.4 Ludi und Ludii 21
3.5 Implementierung in GBG 21
4 Selbstlernende Agenten in Yavalath **24**
4.1 Evaluationsstrategie 24
4.2 Ergebnisse 27
4.3 Einfluss verschiedener Parameter auf das Training 33
4.3.1 Ausschalten der Symmetrien 33
4.3.2 Eligibility Traces 35
4.3.3 Temporal Coherence 36
4.4 Finale Evaluation 38
5 Verwandte Werke **41**
5.1 Biased MCTS in Ludii 41
5.2 Deep Learning in Polygames 41
5.3 Transfer von Spielwissen über verschiedene Spielvarianten hinweg in Polygames 42

6 Zusammenfassung und Ausblick 44
7 Literaturverzeichnis 46
Anhang 48

Tabellenverzeichnis

Tabelle 1: Dauer der Evaluationsspiele und ihre Siegesrate 26

Tabelle 2: Zuordnung der verschiedenen Agenten zu den Abbildungen in denen sie genutzt werden und deren Namen in den Einstellungen 48

Tabelle 3: Einstellungen der MCTS Agenten 49

Tabelle 4: Einstellungen der TDN und TCL Agenten. Erklärung der Anmerkungen: NO SYM = Keine Symmetrie genutzt, ET = Eligibility Traces wurden verwendet, TC = Temporal Coherence wurde verwendet 49

Tabelle 5: Trainingsdauer bei unterschiedlichen Anzahlen von Trainingsspielen 49

Tabelle 6: Vorhandene Möglichkeiten zur Evaluation und deren Dauer pro Vorgang 50

Abbildungsverzeichnis

Abbildung 1: MaxN Suchbaum für 3 Spieler 12

Abbildung 2: Das Spielfeld von Yavalath 16

Abbildung 3: Yavalath 3 Spieler: Illustration der oben erklärten Regel der 3-Spieler-Variante 17

Abbildung 4: Ein einfacher Forcing-Move 18

Abbildung 5: Rätsel Forcing-Move: Wie gewinnt Schwarz in 2 Zügen? Schwarz ist am Zug. 19

Abbildung 6: Auflösung Rätsel Forcing-Move: Wie gewinnt Schwarz in 2 Zügen? 19

Abbildung 7: Kleines Dreieck-Muster und wie davon ausgehend gewonnen werden kann 20

Abbildung 8: Mittlere und große Dreieck-Muster 20

Abbildung 9: Interne Repräsentation des Yavalath-Spielfelds veranschaulicht 22

Abbildung 10: Evaluation GBG MCTS Agent mit 10.000 Iterationen vs. Ludii 25

Abbildung 11: Evaluation GBG MCTS Agent mit 25.000 Iterationen vs. Ludii 26

Abbildung 12: Trainingsergebnisse des ersten Agenten, TDN3_10_6_100.000 27

Abbildung 13: Trainingsergebnisse der verschiedenen Agenten mit 10, 25, 50 und 100 Tupeln 28

Abbildung 14: Trainingsergebnisse TDN3_100_7_300.000 29

Abbildung 15: Trainingsergebnisse TDN3_100_7_1.000.000 mit Standardabweichung über 3 trainierte Agenten und Trainingsdauer pro Episode in Sekunden 30

Abbildung 16: Entwicklung der Lernrate α bei verschiedenen Anzahlen von Trainingsspielen. Ausgehend von $\alpha_{init} = 1$ und $\alpha_{final} = 0.2$ 31

Abbildung 17: Entwicklung der zufälligen Zugrate Epsilon bei verschiedenen Anzahlen von Trainingsspielen. Ausgehend von $\varepsilon_{init} = 0.3$ und $\varepsilon_{final} = 0.05$ 32

Abbildung 18: TDN3_100_7_1.000.000 vs. Ludii (Biased MCTS (Uniform Payouts)) 33

Abbildung 19: Symmetrien des Yavalath Spielfeldes veranschaulicht an einem Tupel der Länge fünf (blau) 34

Abbildung 20: Vergleich der Ergebnisse eines Agenten mit ein- und ausgeschalteter Symmetrie 34

Abbildung 21: Trainingsergebnisse zum Einsatz von Eligibility Traces mit verschiedenen Zerfallsraten im Vergleich zum Training ohne 36

Abbildung 22: Vergleich der Ergebnisse des Agenten mit Temporal Coherence aktiviert (TCL3_100_7_300.000) und des Agenten ohne (TDN3_100_7_300.000) 37

Abbildung 23: Trainingsergebnisse mit Temporal Coherence Learning nach 100.000 Spielen 38

Abbildung 24: Trainingsergebnisse des finalen Agenten TCL3_200_7_25.000 39

Abbildung 25: Evaluation TCL3_200_7_25.000 gegen Ludii Biased MCTS 39

1 Einleitung

1.1 Motivation

Während die Anfänge der Künstlichen Intelligenz mittlerweile schon über 65 Jahre zurückliegen, gewann das Thema in den letzten Jahren auch in der breiten Öffentlichkeit immer mehr an Bedeutung. Die Anfänge nahm das Forschungsfeld im Jahr 1956 an einem Workshop am College von Dartmouth und auch dort wurde der Name Künstliche Intelligenz geprägt (Kaplan & Haenlein, 2019). In den Jahren seitdem ist das Thema immer schneller in den Fokus nicht nur der Forscher, die sich damit beschäftigen geraten, sondern auch den der Allgemeinheit. So werden viele bei dem Stichwort Künstliche Intelligenz direkt an die selbstfahrenden Elektroautos von Tesla denken. Tesla nutzt unter anderem ein digitales neuronales Netz, um die Eindrücke der Kameras zu verarbeiten und Objekte zu erkennen (Ingle & Phute, 2016).

Aber auch in anderen Bereichen hat die Künstliche Intelligenz viele Fortschritte gemacht. Ein weiterer ist der Bereich des Game Learning. So machte das Programm AlphaGo vor einigen Jahren Schlagzeilen in allen Zeitungen, als es während der Google DeepMind Challenge Lee Sedol, der als einer der besten Go Spieler weltweit galt, in vier von fünf Spielen schlug (Lee, 2016). Dies galt bis dahin aufgrund des besonders großen Verzweigungsfaktors von Go als nahezu unmöglich. Doch auch dieser Erfolg wurde wenig später schon übertroffen. Im Jahr 2017 veröffentlichte das Team hinter AlphaGo einen Artikel im Wissenschaftsjournal Nature und stellte den Nachfolger AlphaGo Zero vor. Im Gegensatz zu AlphaGo kam dieser komplett ohne menschliches Expertenwissen aus und lernte nur durch Reinforcement Learning während des Spielens gegen sich selbst (Silver, Schrittwieser, & Simonyan, 2017). AlphaGo Zero übertraf seinen Vorgänger bereits nach 3 Tagen.

Ziel dieser Arbeit ist es, den Erfolg oder Nicht-Erfolg von allgemeinen Reinforcement Learning Agenten im Spiel Yavalath innerhalb des General Board Game Playing and Learning Frameworks (Konen, 2019) zu untersuchen. In einer vorhergehenden Arbeit wurde die Grundlege dafür geschaffen, indem das Spiel Yavalath anhand der vorgegebenen Interfaces umgesetzt wurde (Weitz, 2021). Um die Spielstärke der erstellten Agenten zu testen, treten diese unter anderem in einem Wettbewerb gegen Agenten aus dem General Game System Ludii (Browne, Stephenson, Piette, & Soemers, 2019) an. An der Entstehung von Ludii war auch maßgeblich Cameron Browne beteiligt, der ebenfalls Autor des Spiels Yavalath ist.

Für die Verbindung zwischen Ludii und dem GBG Framework wurde 2020 von Johannes Scheiermann eine Schnittstelle geschaffen, die es ermöglicht Agenten gegeneinander antreten zu lassen (Scheiermann, 2020). Da diese Schnittstelle sich aber nur auf das Spiel Othello beschränkte und es in der Zwischenzeit auch seitens Ludii einige Updates gab, die eine Anpassung erforderten, wurde in einer weiteren vorangegangenen Arbeit diese durch eine neue allgemeinere Schnittstelle ersetzt (Weitz, 2022). Diese vereinfacht

es, neue Spiele und deren Agenten ebenfalls für die Schnittstelle verfügbar zu machen. Eines dieser Spiele war Yavalath.

Aufgrund der vorangegangenen Arbeit zu Yavalath und Ludii ist es jetzt besonders von Interesse, diese Themen noch einmal zu vertiefen und mit dem Themenkomplex Reinforcement Learning zu verknüpfen.

Einige der Fragen, die diese Arbeit zu beantworten versucht, sind folgende:

- Welcher Erfolg kann mit Reinforcement Learning Agenten im Spiel Yavalath erzielt werden?
- Welche Parameter beeinflussen den Lernerfolg am stärksten?
- Wurde dieses Thema vorher bereits in anderen Arbeiten angegangen, und wenn ja, wie erfolgreich?

Diese Arbeit wird anfangs einen Überblick über das General Board Game Playing and Learning Framework und die darin verfügbaren Agenten geben. Danach wird das Spiel Yavalath erläutert, sowie dessen Besonderheiten und Taktiken, die angewandt werden können. Im Hauptteil werden dann die Strategien vorgestellt, die zum Evaluieren und Parametrisieren der Agenten benutzt wurden, sowie die Ergebnisse, die durch diese herausgefunden wurden.

2 General Board Game Playing Framework (GBG)

2.1 Überblick

Das General Board Game (GBG) Playing and Learning Framework[1] wurde von Wolfgang Konen und der Forschungsgruppe um ihn herum entwickelt und wird ständig erweitert. Es wurde in Java programmiert und bietet standardisierte Klassen zum Entwickeln von eigenen Agenten und Spielen an. Durch die Standardisierung ist es dann möglich, nach Umsetzung eines neuen Spiels, dieses direkt von allen bereits vorhandenen Agenten spielen zu lassen. Der umgekehrte Fall ist auch möglich, neu implementierte Agenten sind direkt in der Lage alle bereits vorhandenen Spiele zu spielen.

Als Motivation für die Entwicklung des Frameworks beschreibt Konen unter anderem die einfache Möglichkeit, Studierende mit dem Einstieg in das Themengebiet des Game Learning vertraut zu machen, ohne dass diese den großen Aufwand betreiben müssen sowohl Agenten als auch Spiele komplett neu zu programmieren. Auch wird die Möglichkeit genannt, in manchen Spielen selbstlernende Agenten gegen bereits perfekt spielende Agenten antreten zu lassen, um die Möglichkeit zu erkunden, wie nahe an perfekt ein selbstlernender Agent herankommt (Konen, 2019).

2.2 Spiele

Zum Zeitpunkt dieser Bachelor-Arbeit umfasst das GBG-Framework zwölf verschiedene Spiele. Dabei handelt es sich sowohl um einfache deterministische Spiele wie Tic-Tac-Toe, als auch um komplizierte und nicht-deterministische Spiele wie Poker oder 2048.

Die vollständige Spieleliste umfasst momentan:

- Black Jack
- Vier gewinnt
- EinStein würfelt nicht
- Hex
- Poker
- Nim
- Othello
- Rubiks Cube
- Sim
- Tic-Tac-Toe
- Yavalath

[1] https://github.com/WolfgangKonen/GBG

- 2048

Die meisten dieser Spiele bieten auch noch verschiedene Parameter an, mit denen sich das Spiel verändern lässt, wie z.B. die Anzahl der Spieler oder die Größe des Spielfeldes.

Anleitungen wie neue Spiele anhand der vorhandenen Interfaces implementiert werden können und genauere Erklärungen zu diesen finden sich sowohl im GitHub des GBG Projektes als auch im technischen Report: *The GBG Class Interface Tutorial V2.3* (Konen, Oct 2021).

2.3 Agenten

Innerhalb des GBG Frameworks existiert eine Vielzahl an unterschiedlichen Agenten. Während diese Arbeit sich im späteren Verlauf speziell auf *Temporal Difference (N-Tupel)* Agenten konzentrieren wird, sollen die vorhandenen Möglichkeiten dennoch kurz vorgestellt werden.

2.3.1 Human

Wie der Name bereits vermuten lässt, ist der Human-Agent ein menschlicher Spieler. Über eine spielspezifische GUI wird das Spielbrett angezeigt, sowie die Spielzüge entgegengenommen. Dies ist zum Beispiel nützlich, wenn man andere Agenten in bestimmten Spielsituationen testen möchte und diese visualisieren will.

2.3.2 Random

Ähnlich wie der Human-Agent, hält der Name des Random-Agenten was er verspricht. Aus den im momentanen Spielzustand vorhandenen Aktionen wird eine zufällige ausgewählt und ausgeführt.

2.3.3 Max-N

In deterministischen Spielen ist es theoretisch möglich immer den bestmöglichen Zug zu bestimmen, indem ein Suchbaum des kompletten Spiels aufgebaut wird. Oft scheitert dies aber an der dafür nötigen immensen Rechenleistung. Dies ist auch die Funktionsweise des Minimax-Algorithmus. Diesen Namen hat er, weil versucht wird den Erfolgswert des Gegners zu minimieren, während der eigene maximiert wird. Ausgehend von dem aktuellen Spielzustand wird ein Suchbaum erstellt, der entweder über eine Variable limitiert ist, oder bis zum Ende des Spiels reicht.

Der Max-N Agent ist eine Erweiterung und Generalisierung des Minimax-Algorithmus auf deterministische Spiele mit mehr als zwei Spielern. Es wird angenommen, dass jeder Spieler versucht seinen eigenen Gewinn zu maximieren und denen der anderen Spieler gleichgültig gegenübersteht. An den Grenzknoten des Algorithmus wird eine spielspezifische Bewertungsfunktion angewendet, die dann ein N-Tupel zurückgibt. N steht in diesem Fall für die Anzahl der Spieler und die Komponenten des Tupels für den erwarteten Gewinn (Korf, 1991).

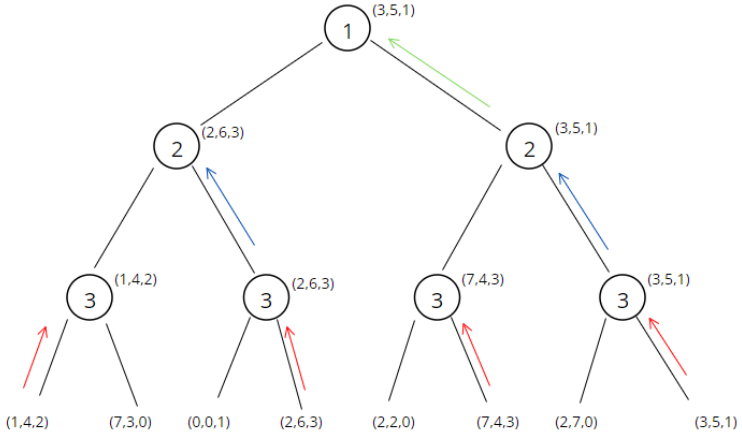

Abbildung 1: MaxN Suchbaum für 3 Spieler

Abbildung 1 zeigt eine Verdeutlichung des MaxN-Suchbaumes. Die eingefärbten Pfeile stellen dar, wie jeder Spieler versucht den eigenen Gewinn zu maximieren.

Innerhalb des GBG Frameworks gibt es auch die Möglichkeit, schon besuchte Zustände in einer HashMap zu speichern, um die Rechendauer des Algorithmus zu beschleunigen.

2.3.4 Monte Carlo Tree Search

Ähnlich wie bei dem MaxN-Algorithmus wird in der Monte Carlo Tree Search (MCTS) ein Suchbaum des Spiels ausgehend vom aktuellen Zustand erstellt. Hier wird jedoch nicht versucht jeden möglichen Spielzustand zu besuchen, sondern anhand verschiedener Strategien versucht den Suchraum zu verkleinern. Der MCTS-Algorithmus besteht aus vier Hauptschritten, die wie folgt definiert sind (Browne, et al., 2012):

- **Selektion:** Der Suchbaum wird von der Wurzel an rekursiv durchlaufen, bis ein Knoten erreicht wurde, der dem Selektionsalgorithmus entspricht und expandierbar ist. Expandierbar bedeutet in diesem Fall, dass der Knoten keinen Endzustand des Spiels darstellt und über noch nicht besuchte Kind-Knoten verfügt.
- **Expansion:** Einer oder mehrere der Kind-Knoten werden zum Suchbaum hinzugefügt.
- **Simulation:** Von dort aus wird mittels Selbstspiels eine Simulation durchgeführt, bis das Spiel einen Endzustand erreicht.
- **Rückpropagierung:** Das Ergebnis der Simulation wird durch den Suchbaum zurückgereicht und zu den Statistiken der Elternknoten hinzuaddiert.

Für die einzelnen Schritte des Algorithmus gibt es eine Menge verschiedener Strategien, wie genau diese umgesetzt werden können. So wird zum Beispiel für die Selektion in GBG die *Upper Confidence Bound for Trees (UCT)* Strategie verwendet.

UCT ist eine der beliebtesten Selektionsstrategien für den MCTS Algorithmus. Es schafft einen guten Ausgleich zwischen der Exploration, dem Aufsuchen neuer Zustände, die bis jetzt noch nicht abgetastet wurden und der Exploitation, dem Erkunden von Zuständen, die ein gutes Ergebnis versprechen (Browne, et al., 2012). Es wird versucht einen Kind-Knoten auszuwählen, der einen möglichst großen UCT-Wert aufweist. Innerhalb des GBG-Frameworks wird dieser wie folgt berechnet:

$$UCT = \bar{X}_c + K * \sqrt{\frac{\ln(n+1)}{n_c}} + r$$

Formel 1: Berechnung des UCT Wert innerhalb von GBG

X_c ist dabei die durchschnittliche Belohnung für den Kind-Knoten c. Diese berechnet sich aus dem zu erwartendem Punktestand für diesen durch die Anzahl wie oft dieser bereits besucht wurde. *K* ist eine Konstante, die die Explorationsrate des Agenten beeinflusst. *n* ist die Anzahl wie oft der momentane Knoten schon besucht wurde und entsprechend n_c die Anzahl wie oft der Kind-Knoten besucht wurde. *r* ist ein Zufallsfaktor, um eventuell entstehende Gleichstände aufzulösen.

2.3.5 Temporal Difference

Der Temporal Difference Learning Agent unterscheidet sich deutlich von den vorherigen Algorithmen, welche die Baumsuche benutzen und muss erst trainiert werden, bevor er im Spiel eingesetzt werden kann. Er hat Vorteile, wenn nicht jede Aktion direkt mit einer Belohnung verknüpft ist, sondern erst eine Reihe von Aktionen am Ende zu einer Belohnung führen. Dies ist bei vielen Brettspielen der Fall (Konen, 2015).

Ziel des Algorithmus ist es eine Spielfunktion $V(s_t)$ zu erlernen, die für einen Spielzustand *s* zum Zeitpunkt *t* eine möglichst nah an perfekte Abschätzung geben kann, wie erfolgreich dieser zum Sieg führt.

Sutton erklärt die Herangehensweise wie folgt mit einem Vergleich zur Wettervorhersage: Der klassische Denkansatz von Prognoselernen sieht vor sich nur auf den Fehler zwischen Prognose und eingetretenem Ereignis zu konzentrieren. Temporal Difference hingegen bezieht dabei die Fehler aller aufeinanderfolgenden Ereignisse mit ein. Würde man versuchen am Mittwoch das Wetter für Samstag hervorzusagen, bezieht der klassische Ansatz nur die Prognose von Mittwoch und das Ergebnis am Samstag mit ein, während TD auch alle dazwischenliegenden Tage berücksichtigt (Sutton, 1988).

Um die Spielfunktion zu erlernen, wird eine Funktion $f(w,s_t)$ gesucht, die $V(s_t)$ bestmöglich approximiert. Der Parameter *w* steht für die Gewichte, die anfangs zufällig initialisiert und dann im Laufe des Trainings verbessert werden (Konen & Bartz-Beielstein, 2008).

Das GBG Framework sieht zwei Möglichkeiten für die Approximation von $f(w,s_t)$ vor:

- Über eine generalisierte lineare Funktion, die das Skalarprodukt der Gewichte *w* und eines Feature-Vektors *g(s)* wiedergibt: $f(w, s_t) = w \cdot g(s)$.

- Ein neuronales Netz, dass die Gewichte *w* und den Feature-Vektor *g(s)* als Input hat.

Während des Trainings wird das sogenannte Fehlersignal δ_t berechnet, um später die Gewichte *w* anzupassen. Da in Brettspielen oft erst Belohnungen vergeben werden, wenn ein finaler Zustand erreicht wird, stellt dieses in den meisten Fällen die Differenz einer zukünftigen $V(s_{t+1})$ und der aktuellen Schätzung $V(s_t)$ dar.

$$\delta_t = r_{t+1} + \gamma * V(s_{t+1}) - V(s_t)$$

Formel 2: Formel für das Fehlersignal δ_t

In der Formel für das Fehlersignal ist die Variable r_{t+1} die Belohnung zum Zeitpunkt t+1. Wenn das Spiel noch keinen finalen Zustand erreicht hat, wird diese null sein und als Faktor nicht berücksichtigt. γ ist der sogenannte Temporal Discount Faktor, mit einem Wert von $0 \leq \gamma \leq 1$. Durch diesen wird berücksichtigt, dass die Spielfunktion $V(s_{t+1})$ ein möglicher Nachfolger von $V(s_t)$ sein kann, aber vielleicht nicht immer erreicht wird. Dadurch kann die Stärke, mit der zukünftige Zustände berücksichtigt werden beeinflusst werden. In deterministischen Spielen, von denen Yavalath eines ist, sind theoretisch alle Zukünfte berechenbar und es macht Sinn $\gamma = 1$ zu wählen.

Die Gewichte *w* werden in jedem Trainingsschritt weiter angepasst. Maßgeblich dabei sind das Fehlersignal δ und die Lernrate α: $w_{t+1} = w_t + \alpha * \delta_t * e_t$

Der Vektor e_t besteht unter anderem aus den sogenannten Eligibility Traces für die Gewichte. Diese beeinflussen, wie stark die Änderung von momentanen Zuständen auch rückwirkend auf ältere angewandt wird und berechnet sich wie folgt:

$$e_t = \gamma * \lambda * e_{t-1} + \nabla_w f(w_t, s_t)$$

Formel 3: Berechnung des Vektors e_t

Dies wird maßgeblich durch den Parameter λ beeinflusst, der die Zerfallsrate der Eligibility Traces beeinflusst. Ist $\lambda = 0$ haben diese keinen Einfluss, und dann ist der Vektor e_t nur noch der Gradient der Funktion $f(w_t,s_t)$.

Die Lernrate α beeinflusst, wie stark die Gewichte angepasst werden. Bei hohen Lernraten wird sich schneller dem Optimum angenähert, es ist aber auch leicht möglich, dass dort über das Ziel hinausgeschossen wird. Niedrigere Lernraten haben dieses Problem nicht, können das Training aber sehr in die Länge ziehen, wenn diese zu niedrig gewählt wurden.

Damit der Agent während des Trainings vernünftig lernen kann, ist eine richtige Wahl der Features enorm wichtig. Features müssen spielspezifisch vorher eingegeben werden. Ein Feature kann alles sein, was durch den Spielzustand des Brettes abgebildet werden kann, wie zum Beispiel die Muster verschiedener Steine auf dem Spielbrett.

2.3.6 Temporal Difference (N-Tupel)

Der TD-N-Tupel Agent verwendet statt den spielspezifischen Features N-Tupel Systeme. Ein N-Tupel ist eine Aneinanderreihung von *n* Feldern des Spielbrettes. Diese können entweder vorher festgelegt werden, ähnlich wie die Features des TD-Agenten, oder automatisch generiert werden. Dafür stehen zwei Parameter für die Tupel, die Länge und Anzahl, und zwei verschiedene Methoden zum Generieren dieser zur Verfügung:

- RandomWalk: Das Tupel fängt an einem zufälligen Spielfeld an und erstreckt sich von da aus über benachbarte Felder. Damit dies genutzt werden kann, muss die Information hinterlegt werden, wie benachbarte Felder errechnet werden.
- RandomPoint: Es werden verschiedene zufällige Punkte auf dem Spielbrett ausgewählt.

Um die Effizienz der Tupel zu steigern, können die Symmetrien des Spielfelds genutzt werden, um äquivalente Zustände eines Tupels zu berücksichtigen. Dadurch kann auch schon durch weniger Tupel ein größerer Zustandsraum abgedeckt werden.

3 Yavalath

3.1 Hintergrund

Yavalath ist ein von Ludi entwickeltes Brettspiel. Ludi ist ein von Cameron Browne programmiertes „General Game System“, das zum Spielen, Bewerten und Erstellen von neuen Spielen benutzt werden kann (Browne, 2011, S. 1). Dazu werden genetische Algorithmen genutzt, um die Regelsets von vorhandenen Spielen zu kombinieren und zu erweitern.

So konnte Ludi während seiner einwöchigen Laufzeit 19 praktikable Regelsets hervorbringen, eines dieser Spiele ist Yavalath (Browne, 2011, S. 51). Yavalath wird seit vielen Jahren als Brettspiel von Nestorgames vertrieben und verkauft. Die Tatsache, dass Yavalath das erste kommerziell erhältliche Brettspiel war, dass von einem Computerprogramm entwickelt wurde, war ein prominenter Punkt des Marketings (Browne, 2011, S. 83).

3.2 Regeln

Yavalath wird üblicherweise auf einem hexagonalen Spielbrett mit einer Kantenlänge von fünf Spielfeldern gespielt. Die Spielfelder sind ebenfalls hexagonal und bilden dadurch ein wabenartiges Muster wie in Abbildung 2 dargestellt.

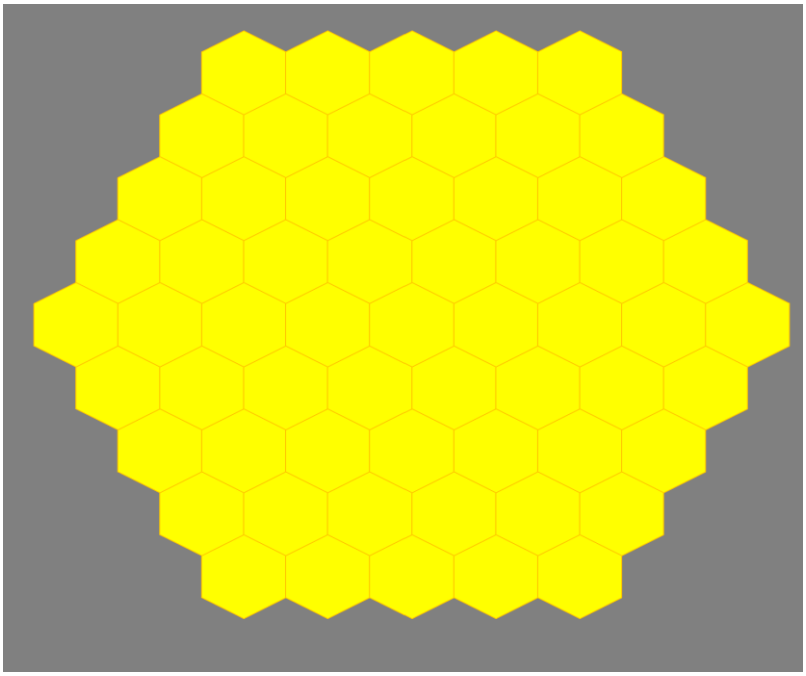

Abbildung 2: Das Spielfeld von Yavalath

In der üblichen Version spielen zwei Spieler gegeneinander, indem sie abwechselnd Spielsteine auf die Felder des Brettes legen. Ziel des Spiels ist es, eine Reihe von vier aneinanderhängenden Steinen der eigenen Farbe zu bilden. Die Besonderheit von Yavalath ist, dass dies geschehen muss, ohne vorher eine Reihe von drei Steinen zu bilden, da dies direkt zur Niederlage führt.

Ist das ganze Spielbrett mit Steinen belegt, ohne dass ein Spieler gewonnen oder verloren hat, kommt es zu einem Unentschieden. Dies passiert eher selten, da die Spieler im Laufe des Spiels, aufgrund der abnehmenden Möglichkeiten die Steine zu platzieren, oft gezwungen sind einen verlierenden Stein zu setzen.

3.2.1 3-Spieler-Variante

Während das Spiel normalerweise nur von zwei Spielern gespielt wird, existieren auch Regeln für eine Variante mit drei Spielern. Wie auch in der 2-Spieler-Variante, spielen hier alle Spieler gegeneinander, es existiert keine Team- oder Allianzbildung wie in manch anderen Spielen. Die grundlegenden Regeln des Spiels bleiben gleich: der erste Spieler, der eine Viererreihe zu Stande bekommt, gewinnt. Wer eine Dreierreihe bildet, scheidet aus dem Spiel aus. Die Spielsteine des ausgeschiedenen Spielers verbleiben auf dem Brett.

Als zusätzliche Regel kommt hinzu, dass der Spieler, der gerade am Zug ist, den nächsten Spieler blockieren muss, falls dieser sonst gewinnen würde. Ein Beispiel hierfür ist Abbildung 3. Der blaue Spieler ist am Zug. Würde obige Regel nicht existieren, könnte Blau sich mit dem nächsten Zug entscheiden, ob Schwarz gewinnt oder verliert. Blockiert Blau den schwarzen Spieler, der danach am Zug ist, am rot umrandeten, Feld verliert dieser entweder im nächsten Zug oder Weiß gewinnt das Spiel. Jeder andere Zug von Blau würde zu einem Sieg von Schwarz führen. So muss zuerst Blau den schwarzen Spieler am rot umrandeten Feld blockieren und danach muss Schwarz ebenfalls Weiß blockieren.

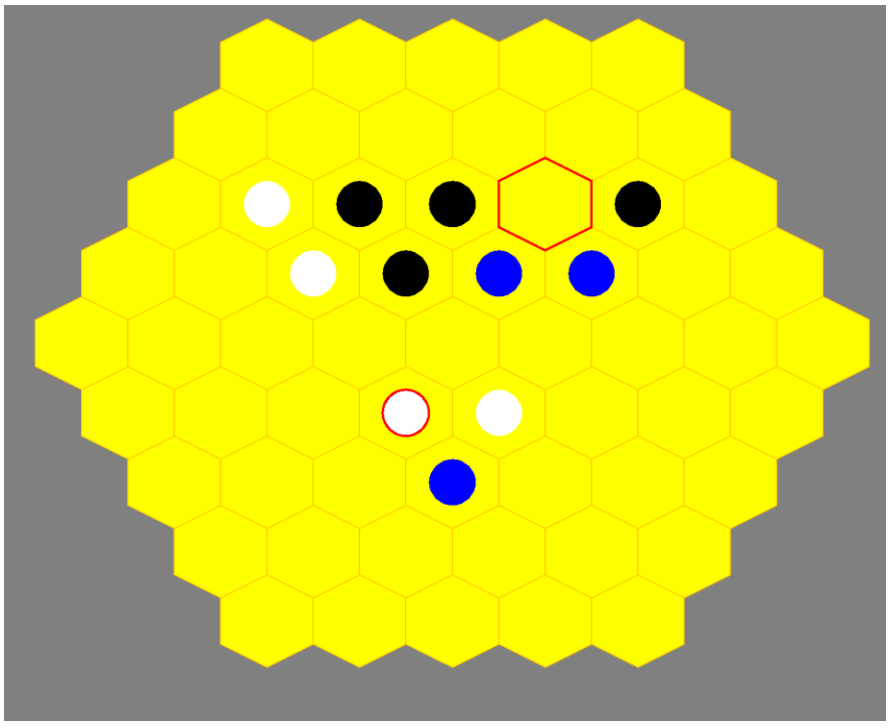

Abbildung 3: Yavalath 3 Spieler: Illustration der oben erklärten Regel der 3-Spieler-Variante

3.2.2 First Move Advantage und die Swap-Regel

In Yavalath hat der erste Spieler einen bedeutenden Vorteil, wenn dieser ungehindert setzen kann, wie er möchte. Die Felder in der Mitte des Brettes erlauben ein Dreieck zu bilden, gegen das sich der zweite Spieler nur schwer verteidigen kann (Browne, 2011, S. 79). Um dieses Ungleichgewicht zu beheben, wurde die Swap-Regel hinzugefügt. Bei dieser kann sich der nachziehende Spieler nach dem ersten Spielzug entscheiden, ob er die Farben tauschen möchte. Eröffnet der erste Spieler zu stark, kann Spieler Zwei sich also entscheiden, ob er diesen Stein stehlen möchte oder nicht. Die Swap-Regel findet nur während der Zwei-Spieler-Version Anwendung. Wird mit drei Spielern gespielt, wird diese nicht berücksichtigt.

3.3 Strategien

3.3.1 Der Forcing-Move

Einer der wichtigsten taktischen Züge in Yavalath ist der sogenannte Forcing-Move. Mit diesem zwingt man den Gegner, Steine an Plätze zu setzen, wo er sie eigentlich nicht hinsetzen möchte. Oft resultiert dies dann in einer Niederlage durch eine Dreierreihe.

Der Forcing-Move funktioniert, indem man einen Sieg androht, den der Gegner dann mit einem Stein an einer vorher geplanten Stelle verhindern muss. Ein einfaches Beispiel für einen Forcing-Move ist in Abbildung 4 zu sehen. Nachdem Weiß Stein 6 gesetzt hat, ist Schwarz gezwungen an der Position von Stein 7 zu blockieren und führt dabei die eigene Niederlage durch eine Dreierreihe herbei.

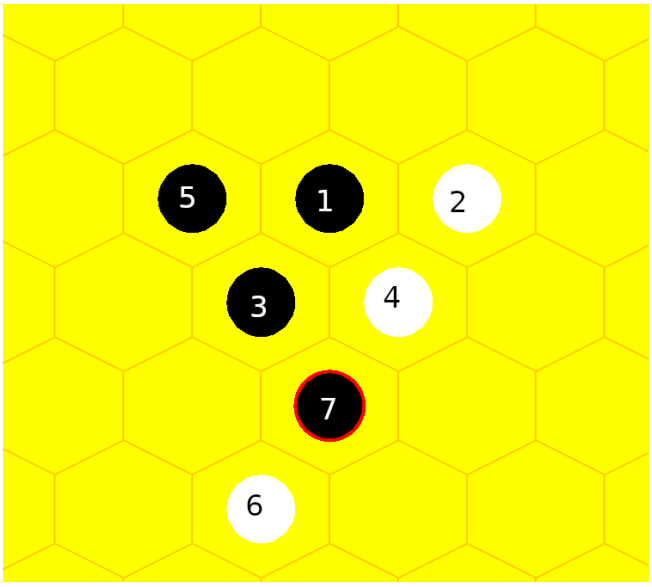

Abbildung 4: Ein einfacher Forcing-Move

Abbildung 5 und 6 zeigen ein weiteres Beispiel wie der Forcing-Move genutzt werden kann, um das Spiel zu gewinnen. Von der Ausgangslage in Abbildung 4 kann Schwarz in 2 Zügen gewinnen, ohne dass Weiß noch etwas dagegen machen kann. Die Auflösung dieses Rätsels findet sich in Abbildung 5.

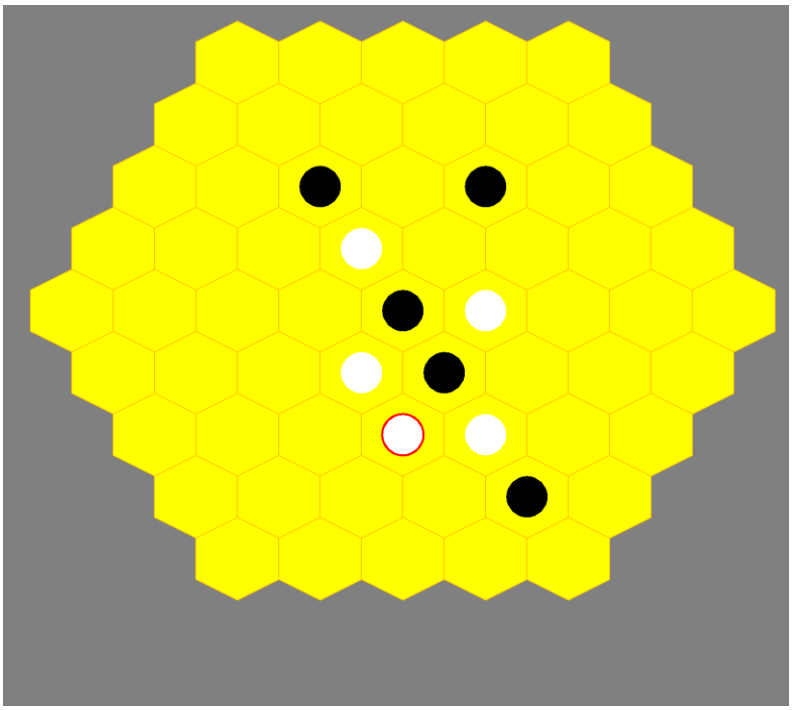

Abbildung 5: Rätsel Forcing-Move: Wie gewinnt Schwarz in 2 Zügen? Schwarz ist am Zug.

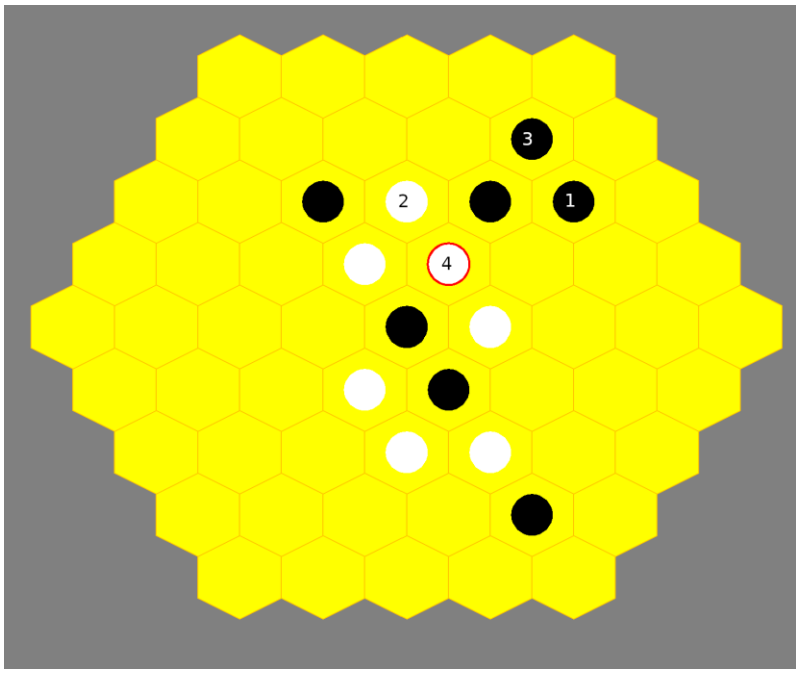

Abbildung 6: Auflösung Rätsel Forcing-Move: Wie gewinnt Schwarz in 2 Zügen?

3.3.2 Starke Muster

Starke Muster bilden in Yavalath unter anderem unblockierte Dreiecke von kleiner, mittlerer und großer Größe. Unblockierte Dreiecke sind Dreiecksformationen, an deren Seiten noch kein gegnerischer Spielstein grenzt. Durch diese hat man die Möglichkeit an drei verschiedenen Seiten anzugreifen und den Gegner Steine setzen zu lassen, die er nicht setzen möchte, wenn nicht sogar direkt zu gewinnen. Je nach Ausgangslage, in

die man die Dreiecke hinein gebaut hat, ist es auch möglich zu gewinnen, ohne dass der Gegner noch etwas dagegen machen kann. Ein Beispiel hierfür ist Abbildung 7.

Die Ausgangslage ist ein kleines Dreieck, in dessen Nähe noch keine Steine des Gegners gesetzt wurden und Schwarz ist am Zug. Durch die abgebildeten Züge ist es möglich, das Spiel durch mehrere aneinander gereihte Forcing-Moves zu gewinnen.

Ähnliche Lösungen gibt es auch für Dreiecke mittlerer und großer Größe, der Übersicht halber sind aber hier nur die Ausgangslagen in Abbildung 8 gezeigt.

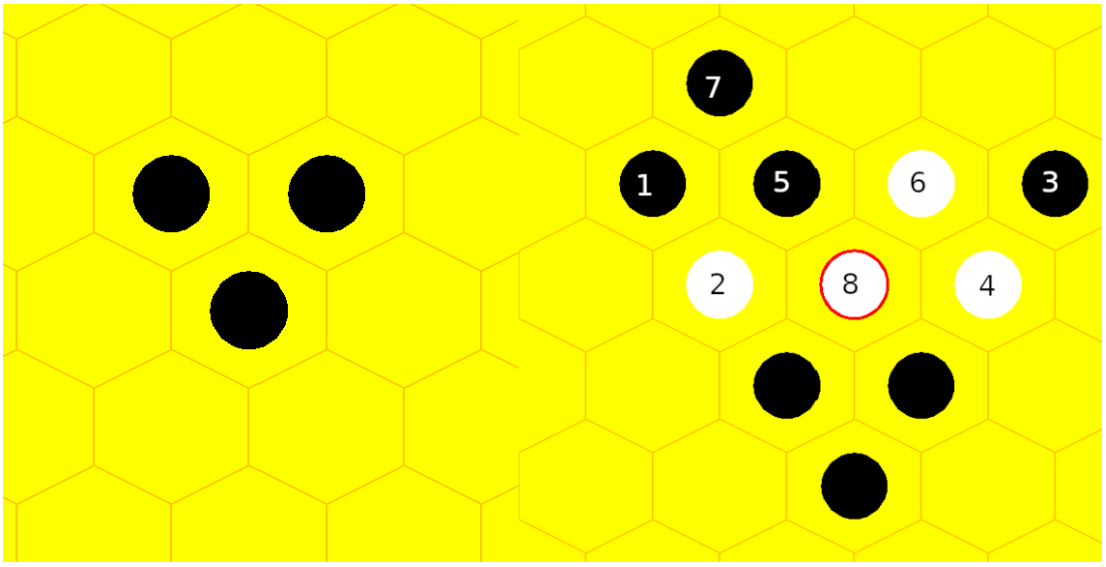

Abbildung 7: Kleines Dreieck-Muster und wie davon ausgehend gewonnen werden kann

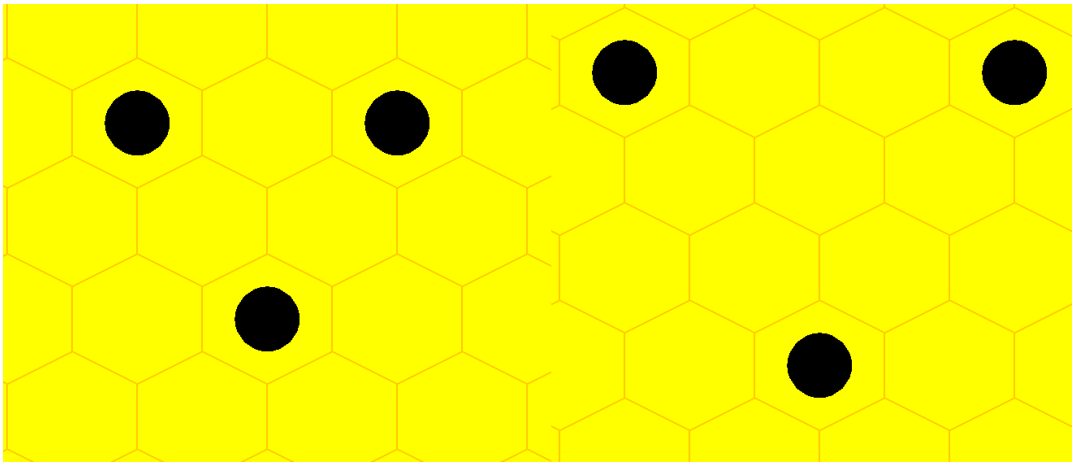

Abbildung 8: Mittlere und große Dreieck-Muster

Nicht so stark wie die Dreiecks-Muster, aber auch erwähnenswert, sind einfache Paare von Steinen, die an mindestens einer Seite nicht blockiert sind. Diese erlauben einen einfachen Forcing-Move an der offenen Seite. Jedoch sind diese Paare je nach Spielbrettzustand nicht ungefährlich, da der Gegner diese Chance schnell mit einem eigenen Forcing-Move in eine Niederlage verwandeln kann.

3.4 Ludi und Ludii

Der Hintergrund von Yavalath als ein von Ludi entwickeltes Spiel wurde in einem vorherigen Abschnitt schon kurz thematisiert. Ludi war ein in C++ geschriebenes General Game System, welches von Cameron Browne für seine Doktorarbeit entwickelt wurde. Das System basierte auf der Grundlage von Ludemen. Diese bieten eine einfache Möglichkeit Spiele in einer leicht verständlichen und modularen Sprache zu beschreiben (Browne, Stephenson, Piette, & Soemers, 2019).

Ludii ist ein neues, auf denselben Prinzipien wie Ludi aufgebautes, General Game System. Es wurde im Rahmen des Digital Ludeme Projektes entwickelt und 2019 veröffentlicht[2]. Das Projekt wird vom Europäischen Forschungsrat finanziert und von Cameron Browne an der Maastricht Universität geleitet und durchgeführt. Während es zwar auf denselben Prinzipien wie Ludi aufbaut, wurde Ludii von Grund auf neu designt, um ungenannte Unzulänglichkeiten von Ludi zu vermeiden (Browne, Stephenson, Piette, & Soemers, 2019, S. 3).

Ludii stellt über 1000 verschiedene Spiele zu Verfügung, sowie unterschiedliche Agenten, mit denen diese gespielt werden können. Es ist auch möglich Agenten in Ludii zu importieren und mit diesen zu spielen, sofern diese von der von Ludii bereitgestellten AI Klasse erben. Diese Möglichkeit nutzte Johannes Scheiermann 2020, um in einem Informatikprojekt an der TH Köln eine erste Version einer Schnittstelle zwischen Ludii und GBG zu schaffen. Während sich diese nur auf das Spiel Othello beschränkte, war das Projekt ein Erfolg und es konnten fortan unter bestimmten Bedingungen Spiele zwischen Ludii und GBG Agenten abgehalten werden (Scheiermann, 2020).

Die Schnittstelle wurde 2022 in einem Praxisprojekt an der TH Köln neu aufgesetzt und erweitert. Während sie auf den gleichen Grundlagen wie die vorherige Version aufbaute, ist die neue Version allgemeiner und leichter verständlich. Zwar muss noch für jedes Spiel ein gewisser Programmieraufwand betrieben werden, um die Schnittstelle zwischen GBG und Ludii nutzbar zu machen, doch ist dies mit einem guten Verständnis der beiden Systeme leicht machbar. Die Implementation der Schnittstelle und des Aufbaus von Ludii wird in dem zugehörigen Praxisprojekt genauer beschrieben (Weitz, 2022).

3.5 Implementierung in GBG

Die Implementierung des Spiels Yavalath wurde in einem vorherigem Informatikprojekt an der TH Köln durchgeführt (Weitz, 2021). Während der Umsetzung, führte die ungewöhnliche Brettform von Yavalath anfangs zu Problemen dies in einer vernünftigen Datenstruktur darzustellen. Die Lösung dieses Problems war es, das Feld mit den ungleichen Reihenlängen in der internen Darstellung, um einige Felder auf ein 9x9 Array zu erweitern. Dies ist in Abbildung 9 zu sehen.

[2] https://ludii.games

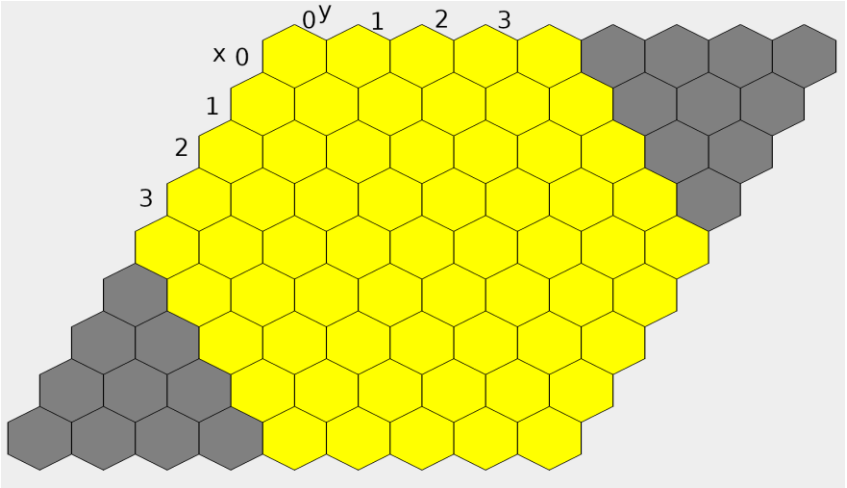

Abbildung 9: Interne Repräsentation des Yavalath-Spielfelds veranschaulicht

Die grauen Felder können dabei durch eine leichte Formel herausgerechnet werden:

$$|y - x| \geq boardSize$$

Ist der Betrag der y-Koordinate minus der x-Koordinate des Feldes größer oder gleich der Spielbrettgröße ist es ein invalides Feld und wird nicht weiter berücksichtigt. Die Spielbrettgröße bezeichnet hierbei die Länge des Spielbrettes in Feldern, am Beispiel der Abbildung wäre dies fünf.

Zu dem Zeitpunkt der Umsetzung wurde theorisiert, das dies unter anderem zu Problemen mit der Laufzeit führen könnte. Die Vergrößerung des Brettes löste zwar das Problem, aber erhöhte die Anzahl an Spielfeldern bei der Brettgröße fünf auch um circa 33 Prozent von 61 auf 81 (Weitz, 2021).

Während dies noch nicht genau analysiert wurde, trat durch die vermeintliche Lösung noch ein anderes Problem auf. Im GBG-Framework werden Aktionen mit einem kontinuierlichen Wert kodiert. Dieser befindet sich zwischen 0 und der maximalen Anzahl der Spielfelder minus 1, in Falle von Yavalath wären dies 60. In der Yavalath Implementierung in GBG war dies aber nicht der Fall. Auch die invaliden Aktionen haben einen Wert zugewiesen bekommen, welcher dann aber später nicht in der Liste an validen Aktionen auftauchte. So hatte in der Liste an Aktionen, die ausgeführt werden können, zum Beispiel die letzte Aktion Nummer 60 nicht auch den Aktionswert 60, sondern 80.

Die dadurch entstandenen Lücken und die Inkongruenz stellten ein Problem für die Implementierung des TD-N-Tupel Agenten dar. Dieser führt Aktionen über den Index aus, den sie in der Aktionsliste haben. Dadurch führte der Agent beim Lernen Aktionen aus, die eigentlich nicht gemeint waren und das Training konnte nicht sinnvoll stattfinden.

Um das Problem zu lösen, werden in einem Zwischenschritt die Aktionswerte über ein Mapping angepasst, damit an jeder Stelle der richtige Wert benutzt wird. Die Aktionsliste ist jetzt kontinuierlich und ohne Lücken und das Training des Agenten funktioniert ohne Probleme.

Auch wenn das hier entstandene Problem gelöst werden konnte, ist die Lösung dafür noch nicht zufriedenstellend. Es ist noch nicht klar ob und wie stark die zusätzlichen Felder das Training der Agenten verlängern.

4 Selbstlernende Agenten in Yavalath

4.1 Evaluationsstrategie

Die Evaluation des Agenten erfolgt in mehreren Stufen und an verschiedenen Punkten des Trainingsprozesses. Die Auswahl nach einem geeigneten Gegner für den zu evaluierenden Agenten gestaltete sich erst schwierig. Eine Recherche nach einem perfekt spielenden Yavalath-Agenten, der dann zur Evaluation in das GBG Framework integriert werden könnte, brachte leider keine Ergebnisse.

Auch der interne MaxN-Agent ist keine gute Wahl, da durch die große Anzahl an möglichen Spielzuständen keine hohe Größe für den Suchbaum gewählt werden kann. Bereits geringe Größen von drei oder vier sorgen dafür, dass einzelne Evaluationsspiele viele Minuten andauern. Die Wahl fiel schlussendlich auf verschiedene MCTS-Agenten, die je nach Einsatzpunkt unterschiedlich parametrisiert sind.

Der erste Punkt der Evaluation findet während des Trainingsprozesses statt. Dort wird alle *n* Spiele ein Aufruf an den spielspezifischen Evaluator getätigt, der diese dann mit den voreingestellten Einstellungen durchführt. Für diesen Punkt wurde ein MCTS-Agent mit den folgenden Parametern gewählt:

- Iterationen: 10.000
- Tree Depth: 25
- Rollout Depth: 2000

Der Selektor des Agenten ist *Upper Confidence Bound for Trees* und die Explorationsrate *k* des Algorithmus ist bei den standardmäßigen $\sqrt{2}$. Diese Einstellungen brachten einen guten Ausgleich zwischen Spielstärke und Evaluationsdauer mit sich. Die Anzahl an Evaluationen wurde, sofern nicht anders spezifiziert, auf zehn gesetzt. Dies sollte Ausreißer in den Datenpunkten möglichst geringhalten, ohne die Dauer des gesamten Trainingsprozesses allzu sehr in die Länge zu ziehen. Für die Evaluation spielt jeder Agent in jeder Rolle, um eventuell vorhandene Vorteile des ersten Zuges bei beiden Agenten auszugleichen.

Die zweite Evaluation des Agenten findet statt, nachdem das Training beendet wurde, um ein aussagekräftigeres Ergebnis zu bekommen. Wie bereits in der ersten Evaluation, wird hier auch gegen einen MCTS-Agenten des GBG-Frameworks gespielt. Die einzige Änderung im Vergleich zum Trainingsprozess ist ein Erhöhen der Iterationen von 10.000 auf 25.000. An diesem Punkt werden 100 Evaluationsspiele über das *Compete in all Roles* System des GBG-Frameworks gespielt. In diesem werden jeweils 100 Spiele für jeden Startspieler gespielt.

Die dritte Evaluation des Agenten findet dann über die Ludii-Schnittstelle statt. Zwar bietet auch Ludii nur einen MCTS-Agenten an, aber durch die vielen verschiedenen Möglichkeiten zur Auswahl der Strategie des Algorithmus, kann der Agent relativ gut auf verschiedene Spiele zugeschnitten werden. Für das Spiel Yavalath wird von Ludii die KI

Biased MCTS (Uniform Playouts) verwendet. Diese verwendet eine Variante der Selektionsstrategie von AlphaGo, welche wiederum als Variante von *PUCB1* beschrieben wird (Soemers D. , 2021c). *PUCB1* wurde zuerst von Christopher D. Rosin 2011 beschrieben. (Rosin, 2011).

Biased MCTS kombiniert die Monte-Carlo-Baum-Suche mit dem Lernen aus Selbstspiel. Statt einem *Deep Neural Network (DNN)* wie oft üblich, wird dort jedoch ein linearer Funktionsapproximator verwendet (Soemers, Piette, & Browne, 2019).

Auch benutzt der Ludii-Agent keine feste Anzahl an Iterationen, sondern führt so viele durch, wie er in einer bestimmten Zeitspanne schafft. Die Standardeinstellung, welche für diese Evaluation beibehalten wurde, liegt bei einer Sekunde. Die Anzahl an Iterationen, die in dieser Zeit geschafft wird, liegt anfangs bei circa 10.000, steigt aber zum Ende des Spiels hin rasant an. Dies hängt wahrscheinlich mit der verringerten Suchbaumgröße zu dem Zeitpunkt des Spiels zusammen.

Wie bereits in der zweiten Evaluation werden auch hier jeweils 100 Evaluationsspiele pro Startspieler durchgeführt. Leider ist eine Evaluation während des Trainings gegen den Ludii Agenten nicht möglich, da die Schnittstelle zwischen Ludii und GBG momentan nur in eine Richtung funktioniert. Agenten aus GBG können in Ludii geladen werden, andersherum aber nicht.

Um einen initialen Überblick über die Spielstärke der verschiedenen Agenten zu bekommen, wurden für die beiden, wie oben erklärt, parametrisierten MCTS-Agenten des GBG-Frameworks jeweils 100 Testspiele pro Seite gegen den Ludii-Agenten durchgeführt. Die Ergebnisse davon sind in den Diagrammen in Abbildung 10 und 11 zu sehen.

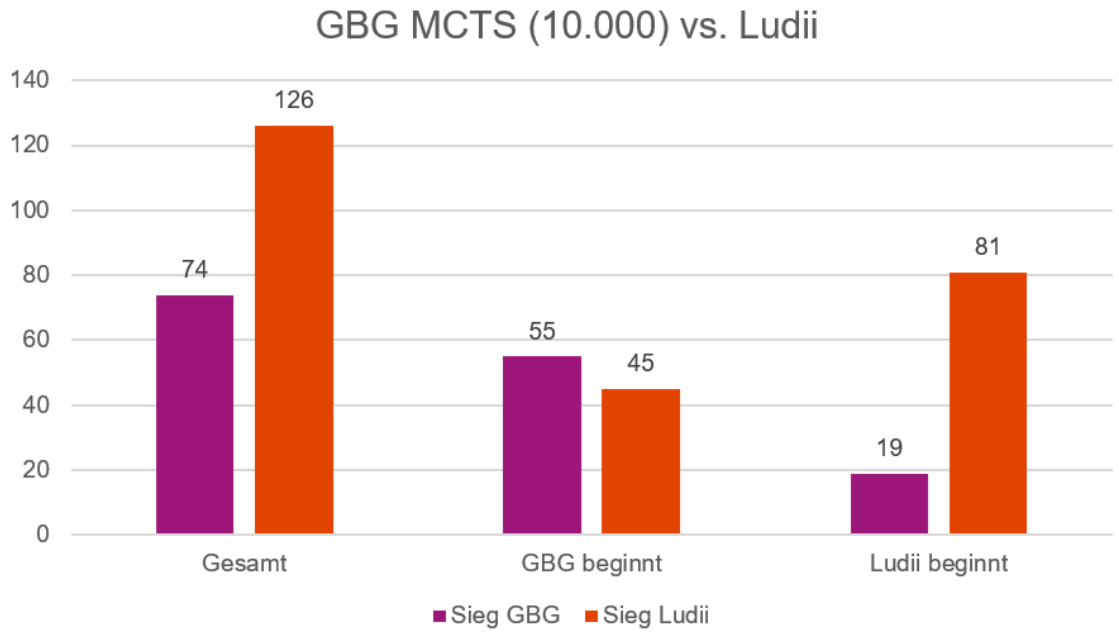

Abbildung 10: Evaluation GBG MCTS Agent mit 10.000 Iterationen vs. Ludii

Aus den Daten wird direkt der große Vorteil des ersten Zuges in Yavalath ersichtlich. Während Ludii diesen gut ausnutzt, und 81 von 100 Spielen gewinnt, schafft der MCTS-

Agent mit 10.000 Iterationen, siehe Abbildung 10 dies nicht wirklich. Trotz des Vorteils gewinnt er nur knapp über 50% der Spiele.

Auch eine Erhöhung der Iterationen auf 25.000 brachte keine großartigen Veränderungen mit sich, wie in Abbildung 11 erkennbar ist. Die Siegesrate, wenn Ludii zuerst zog, sank leicht. Führte der MCTS-Agent von GBG den ersten Zug aus, konnte dieser jetzt 62 von 100 Spielen für sich gewinnen.

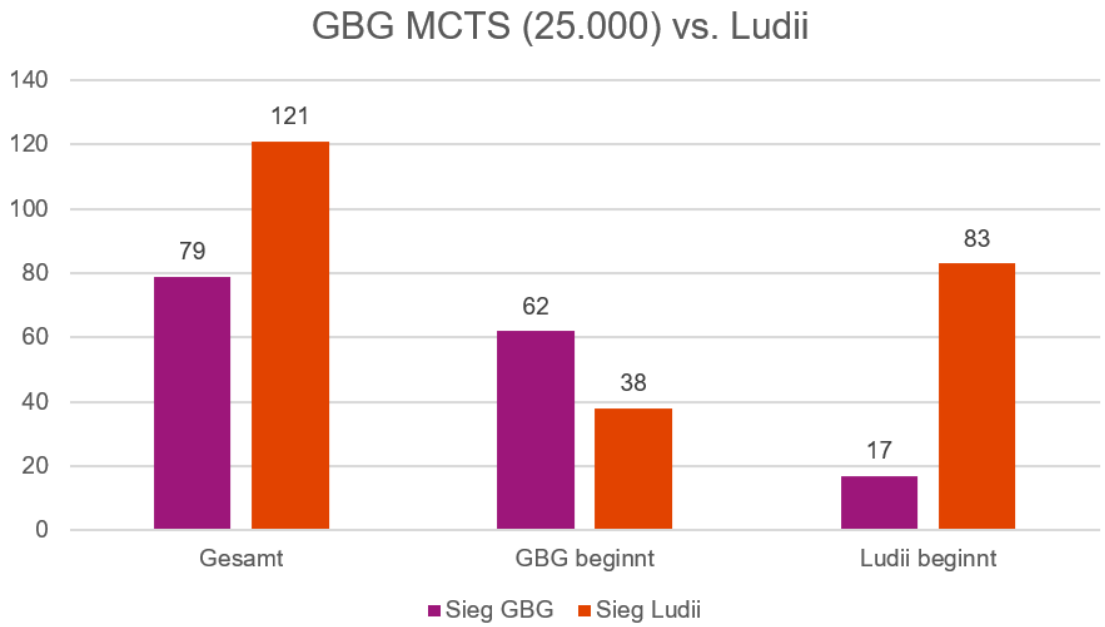

Abbildung 11: Evaluation GBG MCTS Agent mit 25.000 Iterationen vs. Ludii

Dass eine weitere Erhöhung der Iterationen diese Siegeschance noch weiter steigern könnte, ist naheliegend. Zwar könnte dadurch die Spielstärke zu evaluierender Agenten genauer beurteilt werden, aber lässt sich dies durch den dadurch entstehenden Zeitaufwand nicht rechtfertigen. Während die 200 Evaluationsspiele, jeweils 100 pro Seite, bei 10.000 Iterationen noch 15 Minuten benötigten, wurden für die Evaluationsspiele bei 25.000 Iterationen schon 38 Minuten gebraucht.

Iterationen	Dauer	Siegesrate
1	6 Minuten	0%
1.000	7 Minuten	11 %
5.000	10 Minuten	17 %
10.000	15 Minuten	37 %
25.000	38 Minuten	44.5 %

Tabelle 1: Dauer der Evaluationsspiele und ihre Siegesrate

Tabelle 1 zeigt weitere Tests zu der Dauer von Evaluationsspielen bei geringeren Iterationen und die zugehörige Siegesrate. Aus diesen Daten lässt sich eine grobe Formel extrapolieren, mit der die Dauer für höhere Iterationen geschätzt werden kann.

$$y = 3 * 10^{-8} * x^2 + 0.0006 * x + 6.1658$$

Formel 4: Polynomische Formel zur Extrapolation der Dauer bei höheren Iterationen

In der Formel steht *x* für die Anzahl an Iterationen und *y* für die Dauer in Minuten. Erhöht man die Iterationen auf 50.000 würde die Dauer der Evaluationsspiele schon auf knapp 2 Stunden ansteigen und bei einer weiteren Verdopplung auf 100.00 auf über 6 Stunden.

4.2 Ergebnisse

Erste Tests zum Training des TD-N-Tupel Agenten wurden mit den Standardeinstellungen des GBG-Frameworks für TD-Agenten durchgeführt. Dabei handelt es sich um eine konstante Lernrate mit $\alpha_{init} = \alpha_{final} = 0.2$, eine zufällige Zugrate von anfangs $\varepsilon_{init} = 0.3$ und auf $\varepsilon_{final} = 0.0$ abnehmend. Es wurden zehn zufällige, zusammenhängende Tupel der Länge 6 mit der RandomWalk-Methode erstellt. Die 12 Symmetrien des Spielbretts von Yavalath wurden zu Hilfe genommen. Die Trainingsdauer wurde auf 100.000 Spiele festgelegt, mit Evaluationen alle 10.000 Spiele. Sofern nicht anders angegeben, finden alle Evaluationen gegen den GBG MCTS-Agenten mit 10.000 Iterationen statt. Dies betrifft unter anderem alle Abbildungen, die die Entwicklung der Siegesrate über den Trainingsverlauf darstellen. Für die Datenerhebung wurden die Ergebnisse von 5 Agenten über je 10 Evaluationsspiele gemittelt.

Leider waren die Ergebnisse dieses Agenten, wie in Abbildung 12 zu sehen, nicht sehr zufriedenstellend. Die Siegesrate stieg anfangs leicht auf etwa 50% an, erreichte dort aber auch schnell ein Plateau, dass nicht mehr überkommen wurde. Mit einer Siegesrate von etwa 50% gegen den schwächeren GBG MCTS-Agenten mit 10.000 Iterationen blieb dieser Versuch hinter den Erwartungen zurück.

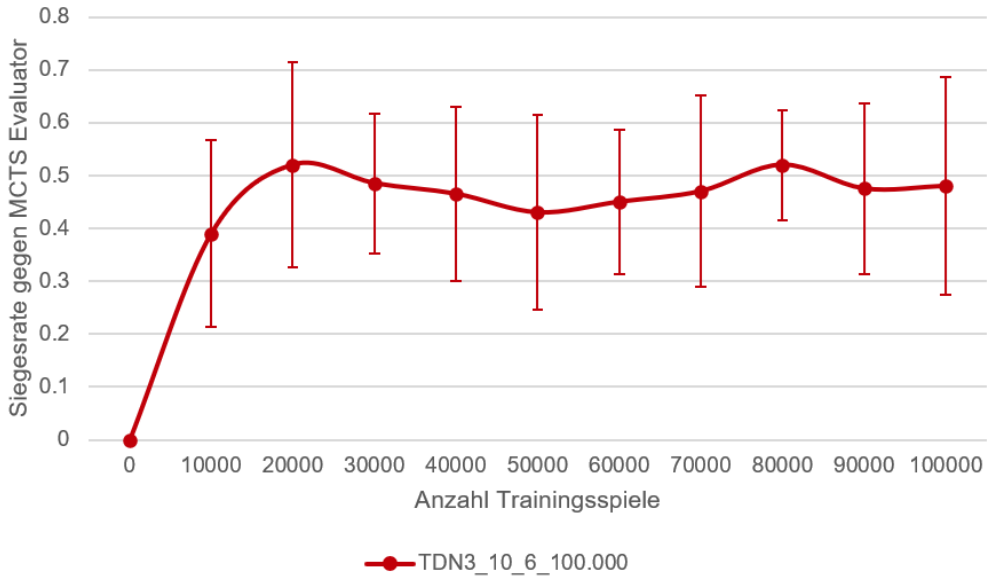

Abbildung 12: Trainingsergebnisse des ersten Agenten, TDN3_10_6_100.000

Die erste Vermutung, warum dies der Fall gewesen sein könnte, betraf die Tupel. Eventuell wurde mit den erstellten 10 Tupeln, trotz genutzter Symmetrie, der Raum der Spielmöglichkeiten nur ungenügend abgedeckt, was dazu führte, dass der Agent nicht vernünftig lernen konnte. Eine andere Möglichkeit war eine ungenügende Einstellung bei den Parametern, die das Lernen selbst betreffen. Vielleicht ist mit $\alpha_{init} = 0.2$ die initiale Lernrate zu niedrig gesetzt, und der Agent kommt sprichwörtlich nicht vom Fleck. Um dies zu überprüfen, ohne blindlings drauf loszutesten, wurde ein Blick in verwandte Literatur gewagt, die sich mit ähnlichen Themen beschäftigt.

So analysierte Simon M. Lucas 2008 den Einfluss von n-Tupeln im Temporal Difference Learning mit Blick auf das Spiel Othello. Dort konnte bereits mit 30 Tupeln ein signifikanter Erfolg erzielt werden. Es wurde ebenfalls darauf hingewiesen, dass zu erwarten ist, dass einige Tupel sich als nützlicher erweisen als andere (Lucas, 2008).

Für die nächsten Tests wurde daraufhin entschieden, verschiedene Anzahl an Tupeln zu testen und zu beobachten. Auch wurde die initiale Lernrate von α_{init} auf 1.0 gesetzt und nimmt im Laufe des Spiels wieder auf $\alpha_{final} = 0.2$ ab. Es wurden Tests für 10, 25, 50 und 100 Tupel durchgeführt und die Tupel Länge ebenfalls auf 7 erhöht. Für jeden dieser Tests wurden 150.000 Trainingsspiele mit Evaluationen alle 7.500 Spiele durchgeführt. Um etwaigen Ungenauigkeiten durch Datenausreißer vorzubeugen, wurden für jede Menge an Tupeln fünf Agenten trainiert und die Ergebnisse dann gemittelt.

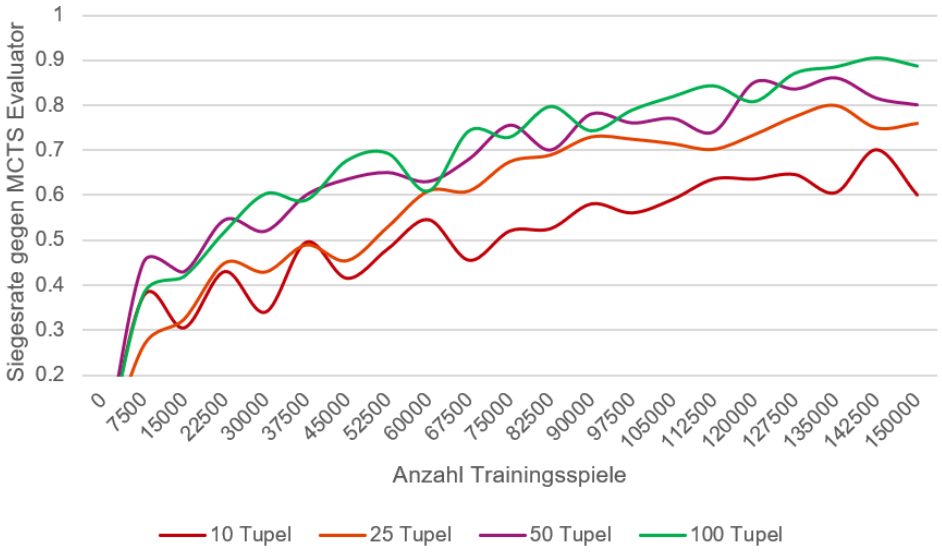

Abbildung 13: Trainingsergebnisse der verschiedenen Agenten mit 10, 25, 50 und 100 Tupeln

Die Ergebnisse dieser Testläufe sind in Abbildung 13 zu sehen. Dort werden die Testergebnisse der verschiedenen Agenten gezeigt. Bereits hier ist sichtbar, dass eine Erhöhung der Tupel signifikant bessere Ergebnisse liefert. Auch der 10-Tupel Agent hat eine kleine Verbesserung auf Siegesraten um die 60% erfahren, was vermutlich sowohl auf die geänderte Lernrate als auch die leicht erhöhte Anzahl an Trainingsspielen zurückzuführen ist. Der größte Unterschied geschieht aber bereits bei der ersten Erhöhung von 10 auf 25 Tupel. Während diese für die ersten 60.000 Trainingsspiele ähnliche

Siegesraten haben, hebt der 25-Tupel Agent sich ab dann ab und nähert sich letzten Endes einer Siegesrate von 80% an.

Auch die beiden 50- und 100-Tupel Agenten zeigen noch sehr gute Verbesserungen, wenn auch in absoluten Zahlen nicht mehr so signifikant wie die Änderung von 10 auf 25 Tupel. Ähnlich wie die 10- und 25-Tupel Agenten haben diese relativ ähnliche Siegesraten, bevor sie sich dann bei circa 100.000 Spielen trennen. Der 50-Tupel Agent erreicht zum Ende hin Siegesraten von 80-85% und der 100-Tupel Agent solche von 90%.

Für die Datenermittlung dieser Abbildung wurden die Ergebnisse von jeweils fünf Agenten gemittelt. Auf die Darstellung der Standardabweichung vom Mittel wird in dieser Abbildung aus Gründen der Übersichtlichkeit verzichtet.

Während der Trend bei allen Agenten noch zu einer steigenden Siegesrate neigt und weitere Trainingsspiele sehr wahrscheinlich auch bessere Ergebnisse liefern würden, ist dies aufgrund der damit verbundenen Rechendauer nicht für alle möglich. Aus diesem Grund wird ab hier auf dem besten Ergebnis des Agenten mit 100 Tupeln der Länge 7 aufgebaut, und alle anschließenden Agenten werden an diesem Punkt dieselbe Tupel Anzahl benutzen, sofern nicht anders gekennzeichnet.

Für die nächsten Tests wurde auf dem vorherigen 100 Tupel Agenten aufgebaut. Alle Parameter sind identisch, nur die Anzahl der Trainingsspiele wurde erhöht. Es wurden jeweils 300.000 und 1.000.000 Trainingsspiele gewählt. Evaluationen werden jede 5% durchgeführt, also bei jeweils allen 15.000 und 50.000 Spielen.

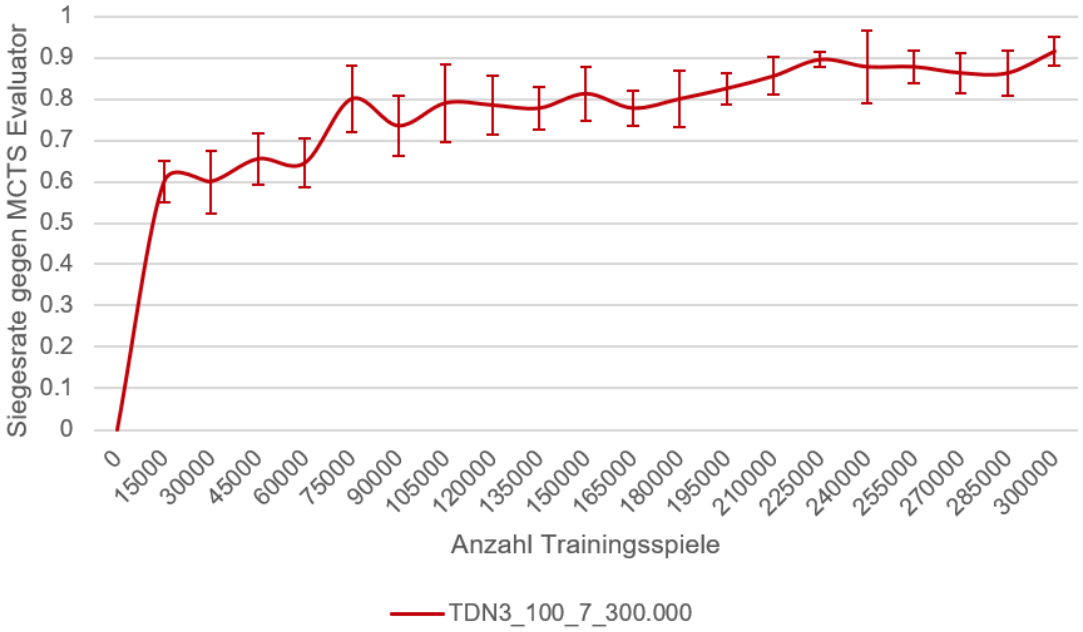

Abbildung 14: Trainingsergebnisse TDN3_100_7_300.000

Abbildung 14 zeigt die Ergebnisse des Agenten mit 300.000 Spielen. Die Kurve der Siegesrate des Agenten ähnelt der des Agenten mit 150.000 Spielen. Beide erreichen bei

circa 90.000 Spielen das erste Mal Siegesraten von 80%. Der weitere Anstieg auf 90% und höher zieht sich im neuen Agenten aber deutlich länger und gegen Ende hin scheinen sie ähnliche Werte zu erreichen.

Wieso zieht sich das Training des Agenten über einen längeren Zeitraum und wieso lässt sich nur gegen Ende eine leichte Verbesserung erkennen? Der einzige Unterschied im Vergleich zu dem vorherigen 100 Tupel Agenten liegt in der Anzahl der Trainingsspiele und sonst wurden keine Parameter verändert. Es sollte eigentlich zu erwarten sein, dass beide Agenten ähnliche Werte bei der gleichen Spielanzahl erreichen.

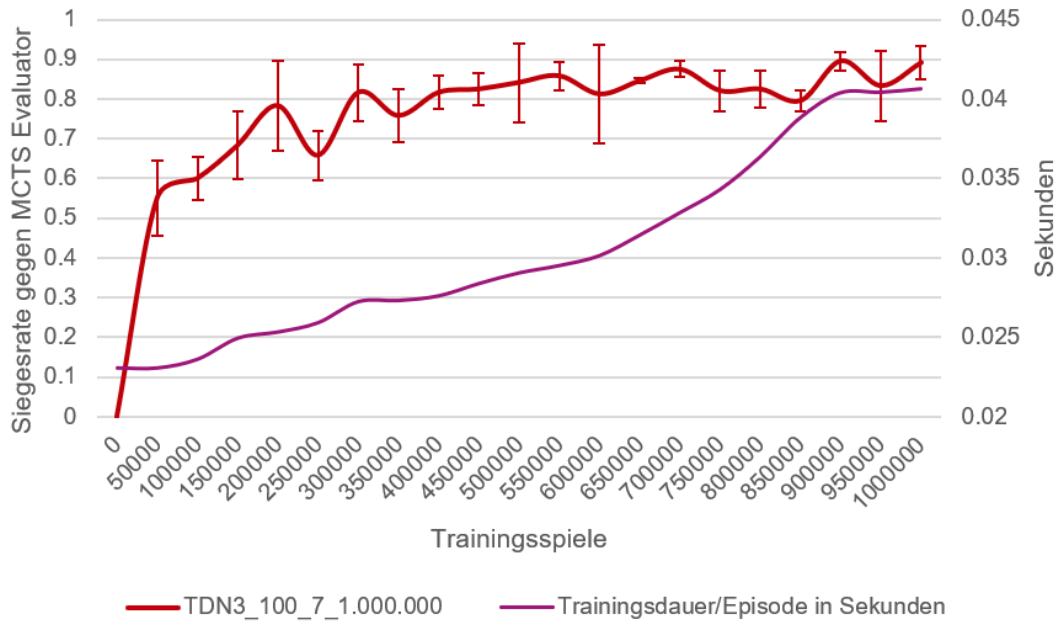

Abbildung 15: Trainingsergebnisse TDN3_100_7_1.000.000 mit Standardabweichung über 3 trainierte Agenten und Trainingsdauer pro Episode in Sekunden

Auch der Agent mit 1.000.000 Spielen weist eine ähnliche Problematik auf, wie in Abbildung 15 zu sehen ist. Die 80% Siegesrate, die von den beiden vorherigen Agenten schon ab ungefähr 90.000 Spielen erreicht wurde, tritt hier erst zwischen 400.000 und 500.000 Spielen konstant auf. Auch die Trainingsdauer des Agenten steigt bei höheren Trainingsspielen deutlich an. Beträgt die Dauer einer Episode anfangs noch etwa noch 0.0225 Sekunden, liegt diese am Ende des Trainings mit etwas über 0.04 Sekunden schon bei fast dem Doppelten. Diese Zeit steigt über die Trainingsdauer gleichmäßig an und erreicht bei circa 900.000 Spielen ein Maximum. Ob dies nur vorrübergehend ist und die Trainingsdauer pro Episode danach noch weiter steigt, ist nicht bekannt, da keine Daten von Agenten mit mehr als 1.000.000 Spielen erhoben wurden.

Ein Grund hierfür könnte an einer gewissen Ungenauigkeit der Datengrundlage legen. Während versucht wurde dies weitgehend zu vermeiden, liefern die 10 Evaluationsspiele pro Agenten und je Datenpunkt für die fünf trainierten Agenten nur jeweils 50 Datenpunkte. Die Standardabweichung der einzelnen Datenpunkte ist in Abbildung 13 und 14 zu sehen.

Ein weiterer Erklärungsansatz findet sich in den Parametern des Temporal Difference Learning. Während diese zwar für alle Agenten gleich eingestellt waren, sind diese nicht über die Trainingsdauer immer gleich. Die Lernrate α zum Beispiel wurde mit 1.0 initialisiert und näher sich im Laufe des Trainings auf 0.2 an. Bei einer unterschiedlichen Menge an Trainingsspielen hat diese also zum gleichen Zeitpunkt verschiedene Werte. Nach jedem Spiel wird die aktuelle Lernrate mit einem Faktor *k* multipliziert, um diese anzupassen. Die Berechnung des Faktors *k* geschieht nach der Formel 5.

$$k = \left(\frac{\alpha_{final}}{\alpha_{init}}\right)^{\frac{1}{maxGameNumber}}$$

Formel 5: Faktor k mit dem die Lernrate jedes Spiel multipliziert wird

$$\alpha(t) = \alpha_{init} * \left(\frac{\alpha_{final}}{\alpha_{init}}\right)^{\frac{t}{maxGameNumber}}$$

Formel 6: Berechnung der Lernrate α zum Zeitpunkt t

Formel 6 zeigt die Berechnung der Lernrate α zu einem bestimmten Zeitpunkt *t*, welches für die Anzahl der Trainingsspiele steht. Diese Formel wurde genutzt, um eine Kurve zu modellieren, wie sich die Lernrate bei den drei verschiedenen Anzahlen Trainingsspielen entwickelt. Dies ist in Abbildung 16 zu sehen. Während *α* bei 150.000 Spielen schon bei den finalen 0.2 angekommen ist, ist es bei 300.000 und 1.000.000 Spielen erst bei jeweils etwa 0.45 und 0.8.

Auswirkung der Lernrate auf das Training + Quellen.

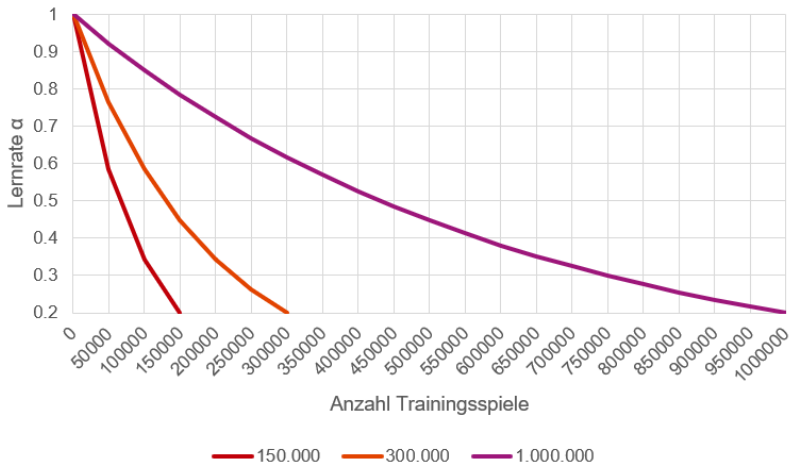

Abbildung 16: Entwicklung der Lernrate α bei verschiedenen Anzahlen von Trainingsspielen. Ausgehend von α_{init} = 1 und α_{final} = 0.2

Analog zur Lernrate wird auch der Parameter ε, welcher die Rate für zufällige Züge regelt, im Laufe des Trainings angepasst, sofern dieser nicht konstant gewählt wurde. Diese ändert sich von ε_{init} auf ε_{final} in einem linearen Abfall. Wurde zum Beispiel ein ε_{init} =

0.3 gewählt und ε_{final} beträgt 0.0, so ist ε bei der Hälfte der Spiele in der Mitte der beiden Parameter, was in diesem Fall 0.15 bedeuten würde.

Abbildung 17 zeigt diese Entwicklung für die bei den Agenten gewählte zufällige Zugrate von $\varepsilon_{init} = 0.3$ und $\varepsilon_{final} = 0.05$. Während diese zwar bei allen Agenten in Relation zur gesamten Spielanzahl gleich schnell sinkt, passiert dies bei Agenten mit weniger Spielen absolut gesehen früher.

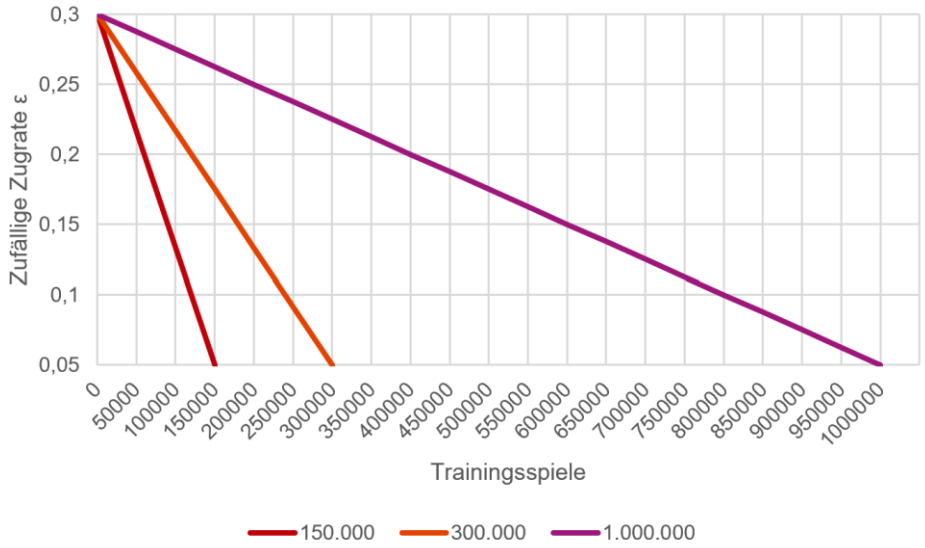

Abbildung 17: Entwicklung der zufälligen Zugrate Epsilon bei verschiedenen Anzahlen von Trainingsspielen. Ausgehend von $\varepsilon_{init} = 0.3$ und $\varepsilon_{final} = 0.05$

Auch wenn weitere Anpassungen der einzelnen Parameter sicherlich noch weitere Erfolge nach sich ziehen würden, ist dies doch mit erheblichem Aufwand verbunden. Um einen genaueren Einblick in die Spielstärke eines der Agenten zu bekommen, soll dieser jetzt gegen den Yavalath-Spieler von Ludii antreten. Dies geschieht über die GBG-Ludii-Schnittstelle. Der Agent, der evaluiert werden soll, ist einer mit 100 Tupeln der Länge sieben und 1.000.000 Trainingsspielen, was während der ersten Evaluation gegen den MCTS Agenten von GBG mit 10.000 Iterationen eine Siegesrate von etwa 90% erbrachte. Auf eine Evaluation gegen den etwas stärkeren MCTS Spieler mit 25.000 Iterationen wurde hier verzichtet, da dies nur bedingt bessere Aussagekraft über die Spielstärke hätte, aber auch sehr zeitintensiv wäre.

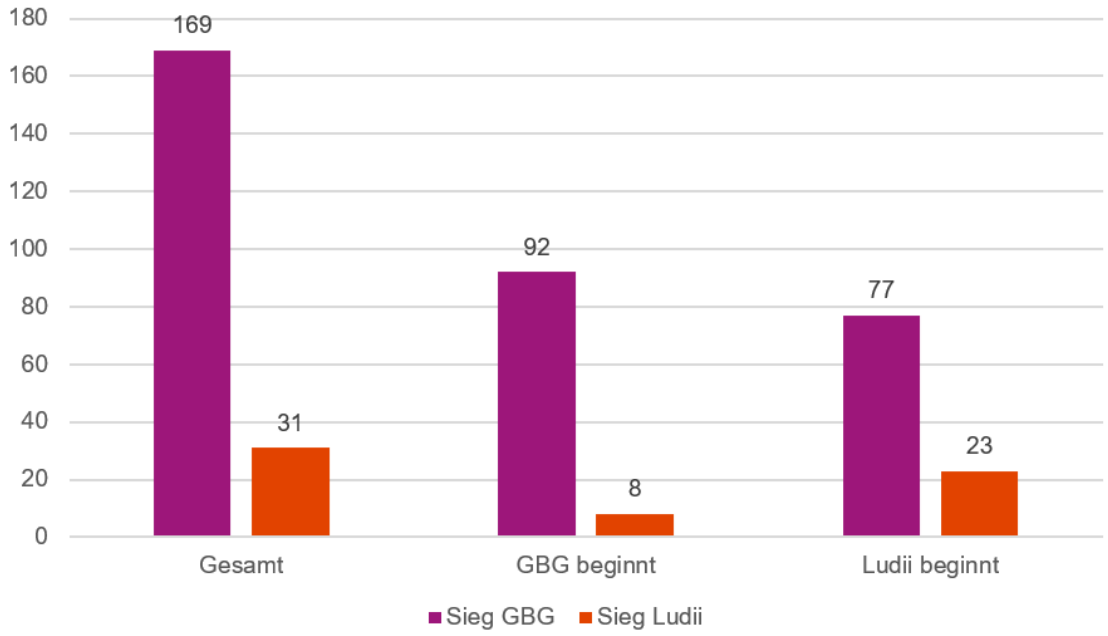

Abbildung 18: TDN3_100_7_1.000.000 vs. Ludii (Biased MCTS (Uniform Payouts))

Abbildung 18 zeigt die Ergebnisse dieser Evaluation. Sowohl wenn der Agent den ersten Zug macht als auch wenn er nachziehen muss, kann er den absoluten Großteil der Spiele für sich entscheiden. Während mit 15% doch ein nicht zu vernachlässigender Unterschied zwischen den beiden Datensets erkennbar ist, gewinnt der Agent durchschnittlich 84.5% aller Spiele für sich. Damit ist er sogar noch um einiges dominanter als der Ludii Agent gegen die beiden MCTS Variationen des GBG Frameworks.

4.3 Einfluss verschiedener Parameter auf das Training

In diesem Kapitel soll noch der Einfluss von verschiedenen Parametern auf den Trainingserfolg untersucht werden, um zu schauen an welcher Stelle Verbesserungen möglich sind.

4.3.1 Ausschalten der Symmetrien

Ein wichtiger Faktor des TD-N-Tupel Agenten, der das Training erleichtern kann, sind die Symmetrien der Spielbretter. Ein Spiel hat Symmetrien, wenn ein Spielzustand *s* auf dem Brett zum Beispiel gedreht oder gespiegelt werden kann und darauf ebenfalls ein valider Spielzustand entsteht. Durch symmetrische Spielzustände können mit einem Ausgangszustand direkt mehrere andere Zustände auch abgedeckt werden.

Die Symmetrien eines Spiels müssen spielspezifisch vorher definiert werden, damit der Agent über alle symmetrischen Spielzustände eines Ausgangszustands im Bilde ist. Im Spiel Yavalath existieren 12 Symmetrien. 6 davon sind Rotationssymmetrien, wenn das Brett um jeweils 60 Grad gedreht wird, sowie für jede dieser 6 noch eine zusätzliche Symmetrie, wenn man das Brett in der Vertikalen spiegelt.

Veranschaulich ist dies in Abbildung 19. Die Ausgangslage ist das blaue Tupel der Länge 5 (6, 13, 14, 21, 29). Durch die 12 Symmetrien des Spielfelds von Yavalath kann mit einem einzigen Tupel ein viel größerer Bereich abgedeckt werden.

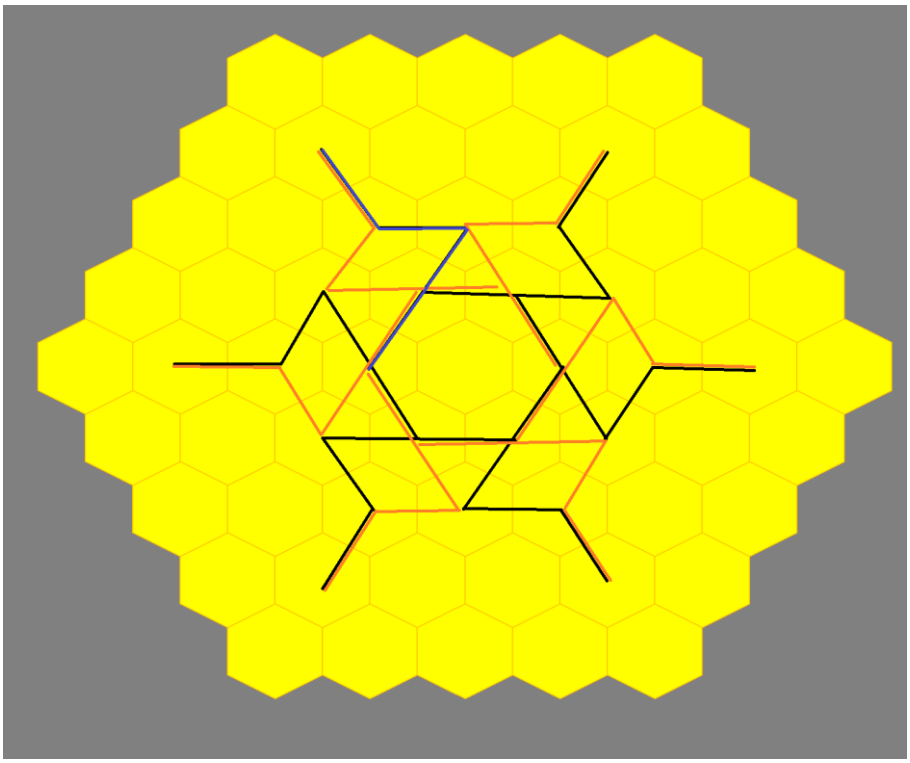

Abbildung 19: Symmetrien des Yavalath Spielfeldes veranschaulicht an einem Tupel der Länge fünf (blau)

Bei allen Agenten, die bis zu diesem Punkt trainiert wurden, ist dies mit aktiven Symmetrien geschehen. Es liegt nahe, dass ein Abschalten der Symmetrien einen negativen Einfluss auf das Training haben wird, trotzdem soll untersucht werden ob dies der Fall ist und falls ja wie stark.

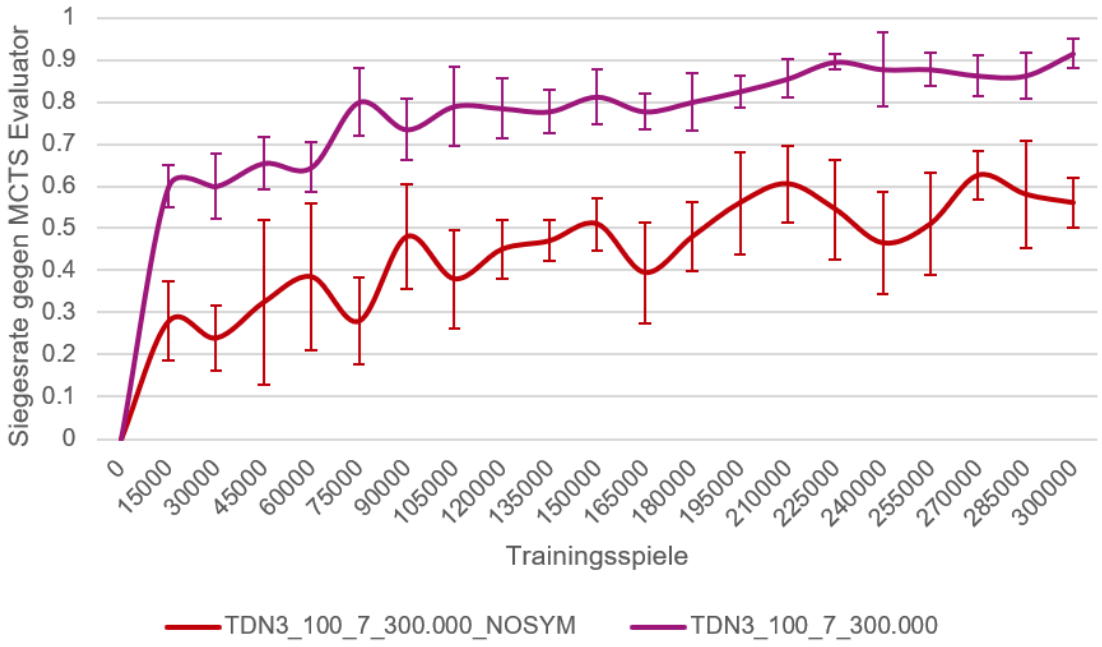

Abbildung 20: Vergleich der Ergebnisse eines Agenten mit ein- und ausgeschalteter Symmetrie

Die Ergebnisse dieses Tests sind wenig überraschend und in Abbildung 20 zu sehen. Während der Agent ohne Symmetrien zwar etwas schneller als vorher trainierte, blieb der Erfolg größtenteils aus. Erst nach über der Hälfte der Trainingsspiele schaffte er es über eine 50% Siegesrate hinaus und blieb schlussendlich bei circa 60% hängen. Damit ist er ähnlich stark wie der erste Testagent mit 10 Tupeln.

Während der Einsatz der Symmetrien also eine längere Trainingsdauer mit sich bringt, rechtfertigen die Ergebnisse dies durchaus und ein Ausschalten der Symmetrien scheint keinen Sinn zu machen.

4.3.2 Eligibility Traces

In ihren Experimenten zum Temporal Difference Learning im Spiel Vier Gewinnt, kamen Bagheri und Thill zum Ergebnis, dass dort der Einsatz von Eligibility Traces einen deutlichen Einfluss auf die Lerngeschwindigkeit der Agenten hatte. Ihre Experimente zeigten, dass dies die Anzahl benötigter Trainingsspiele, um das Spiel zu lernen, um einen Faktor 2 reduzieren konnte (Bagheri & Thill, 2014).

2017 untersuchte Galitzki ebenfalls den Einsatz von Eligibility Traces, dort am Beispiel des Spiels Hex. Es wurde auch der Einfluss der Zerfallsrate λ in verschiedenen Werten getestet. Anders als Bagheri und Thill jedoch, kam Galitzki zu dem Schluss, das Eligibility Traces in diesem Spiel einen negativen Einfluss auf die Lerngeschwindigkeit hatten. Je höher λ gewählt wurde, desto mehr Spiele wurden benötigt, um Hex zu lernen (Galitzki, 2017).

Um herauszufinden, wie dies bei dem Spiel Yavalath ausfällt, wurde auch hier der Einsatz von Eligibility Traces erprobt. Yavalath und Vier Gewinnt weisen viele Ähnlichkeiten auf. Beides sind Spiele, bei denen Reihen von vier Spielsteinen gebildet werden müssen, um zu gewinnen. Ein Unterschied ist aber die in Yavalath existierende Regel, dass drei Steine in einer Reihe verlieren. Die Vermutung lag daher nahe, dass sich das Ergebnis des Einsatzes von Eligibility Traces eher an dem von Vier Gewinnt als dem von Hex richtet.

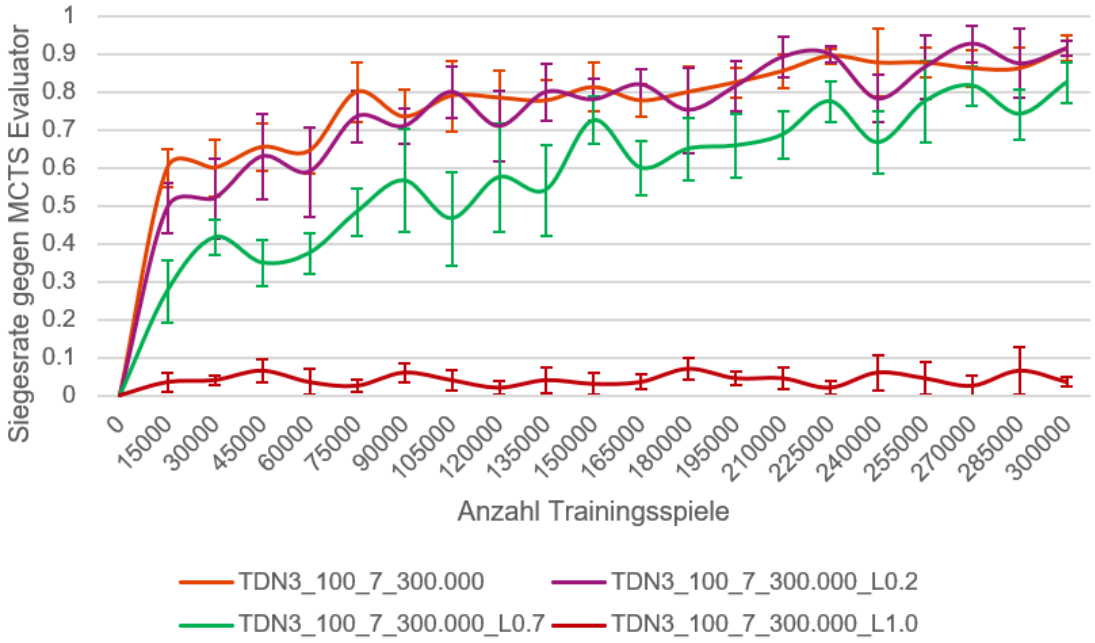

Abbildung 21: Trainingsergebnisse zum Einsatz von Eligibility Traces mit verschiedenen Zerfallsraten im Vergleich zum Training ohne

Abbildung 21 zeigt die Ergebnisse dieses Trainings. Es ist sichtbar, dass, ähnlich wie bei Hex, der Einsatz von Eligibility Traces auch bei Yavalath die Lerngeschwindigkeit negativ beeinflusst. Je höher die Zerfallsrate λ gewählt wird, desto mehr Spiele werden gebraucht, bis der Agent ähnliche Siegesrate wie ohne den Einsatz von ET erreicht. Im Falle einer Zerfallsrate λ = 1.0 ist dies sogar nie der Fall. Dieser Agent hat schon Probleme eine Siegesrate von 10% zu erzielen. Ein Lernerfolg ist hier nicht sichtbar.

4.3.3 Temporal Coherence

Temporal Coherence ist eine Methode zur automatischen Anpassung der Lernrate α während des Trainings. Da das Einstellen der Parameter hier oft viel Versuch und Irrtum erfordert, bis eine gute Option gefunden wurde, suchten Beal und Smith eine Methode, um dies zu verbessern.

In ihrer Methode wird die Lernrate als Verhältnis aus dem Betrag der akkumulierten Veränderung des Gewichtes, sowie der akkumulierten absoluten Veränderung des Gewichtes gebildet. Ebenso arbeitet ihre Methode nicht mit einer Lernrate für alle Gewichte, sondern mit einer separaten Lernrate für jedes einzelne Gewicht. Dies sorgt dafür, dass wenn einige Gewichte schon ein Optimum erreicht haben, andere aber noch nicht, diese auch unterschiedlich behandelt werden können (Beal & Smith, 1999).

Formel 7: Anpassung der Lernrate α nach der Temporal Coherence Methode

$$\alpha_i = \frac{|N_i|}{A_i} \qquad N_i \leftarrow N_i + \sum_{t=1}^{end-1} r_{i,t} \qquad A_i \leftarrow A_i + \sum_{t=1}^{end-1} r_{i,t}$$

Dabei ist $r_{i,t}$ die empfohlene Veränderung des Gewichtes *i* zur Prognose *t*.

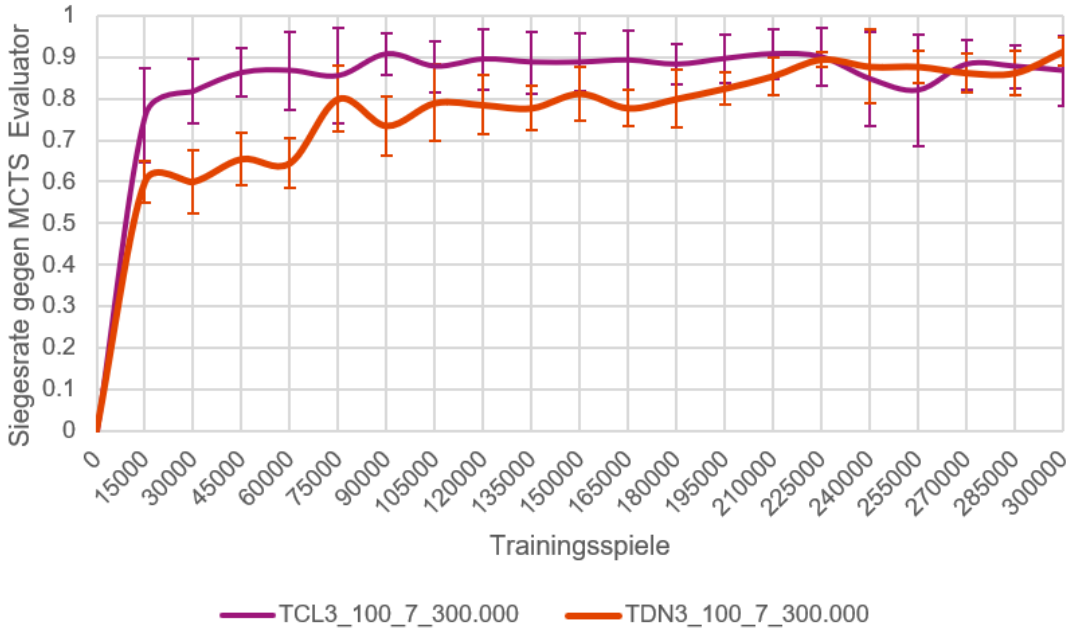

Abbildung 22: Vergleich der Ergebnisse des Agenten mit Temporal Coherence aktiviert (TCL3_100_7_300.000) und des Agenten ohne (TDN3_100_7_300.000)

Abbildung 22 zeigt die Ergebnisse eines Agenten, wenn die Temporal Coherence (TC) Methode zugeschaltet wird, im Vergleich zu einem Agenten mit denselben Parametern, aber ohne TC. Der sprunghafte Anstieg des TC-Agenten direkt zu Beginn des Trainings ist klar sichtbar. Direkt bei der ersten Evaluation hat der Agent eine Siegesrate von 80% und schon nach 90.000 Trainingsspielen wird eine Siegesrate von 90% erreicht und dann relativ stabil gehalten. Dies stellt einen bedeutenden Fortschritt zu den vorherigen Agenten dar. Zwar kommen beide am Ende zu einer ähnlichen Siegesrate, doch erreicht der TC-Agent diese in nur einem Bruchteil der Spiele.

Um diesen Erfolg noch genauer zu untersuchen, wurde eine weitere Evaluation eines Agenten mit denselben Parametern, aber weniger Trainingsspielen durchgeführt. Dafür wurden 100.000 Spiele, mit Evaluationen alle 5.000 Spiele ausgewählt. Um möglichst ähnliche Bedingungen herzustellen, wurde die zufällige Zugrate ε angepasst. Der vorherige TC-Agent startete mit $\varepsilon_{init} = 0.3$ und endete mit $\varepsilon_{final} = 0.05$. Da diese während des Trainings linear abfällt, lässt sich leicht berechnen, wo der Wert von ε zum 100.000 Spiel liegt.

$$\varepsilon_{final} = 0.3 - (0.3 - 0.05) * \left(\frac{1}{3}\right) = \frac{13}{60} \approx 0.21\overline{6}$$

Auch die finale Lernrate α_{final} wird angepasst, um möglichst identische Bedingungen zu schaffen.

$$\alpha_{final} = 1.0 * \left(\frac{0.2}{1.0}\right)^{\frac{100000}{300000}} \approx 0.585$$

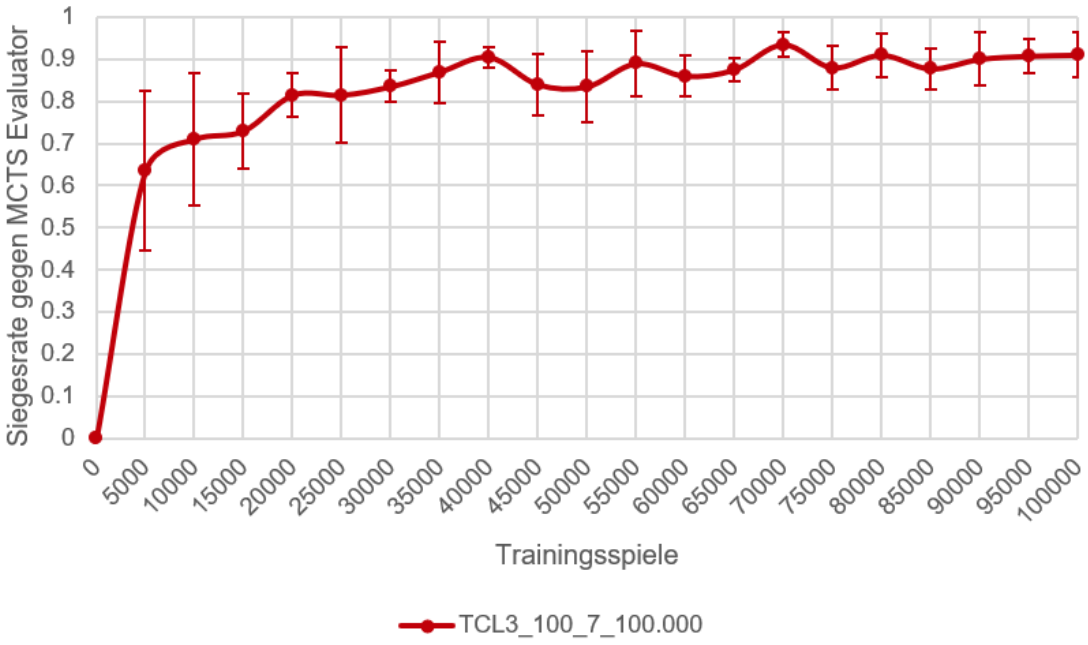

Abbildung 23: Trainingsergebnisse mit Temporal Coherence Learning nach 100.000 Spielen

Abbildung 23 zeigt die Ergebnisse dieses so trainierten Agenten. Wie zu erwarten, ist die Kurve der Siegesrate sehr ähnlich zu dem vorherigen Agenten. Schon nach wenigen Tausend Spielen ist eine Siegesrate von mehr als 70% erreicht und genau wie bei der vorherigen Messung nach 15.000 Spielen eine von 80%. Leichte Unterschiede, wie dass die neue Messung bereits bei 40.000 Spielen die 90% Marke erreicht, sind durch die begrenzte Anzahl der Evaluationsspiele und das Mittel aus nur 5 Agenten erklärbar.

4.4 Finale Evaluation

Um aus den durch die vorherigen Evaluationen gelernten Lektionen noch Resümee zu ziehen, wurde noch ein letzter Agent trainiert, der dann gegen den Ludii-Agenten antreten sollte. Dafür wurden folgende Parameter gewählt:

- Die Lernrate beträgt anfangs $\alpha_{init} = 1.0$ und nimmt auf $\alpha_{final} = 0.6$ ab.
- Die zufällige Zugrate beträgt anfangs $\varepsilon_{init} = 0.3$ und nimmt auf $\varepsilon_{final} = 0.1$ ab.
- Die Anzahl der Tupel wurde auf 200 erhöht, die Länge der Tupel wird mit 7 beibehalten.
- Temporal Coherence ist aktiviert.
- Eligibility Traces werden nicht benutzt.
- Es werden 25.000 Trainingsspiele durchgeführt, mit Evaluationen alle 2.500 Spiele.

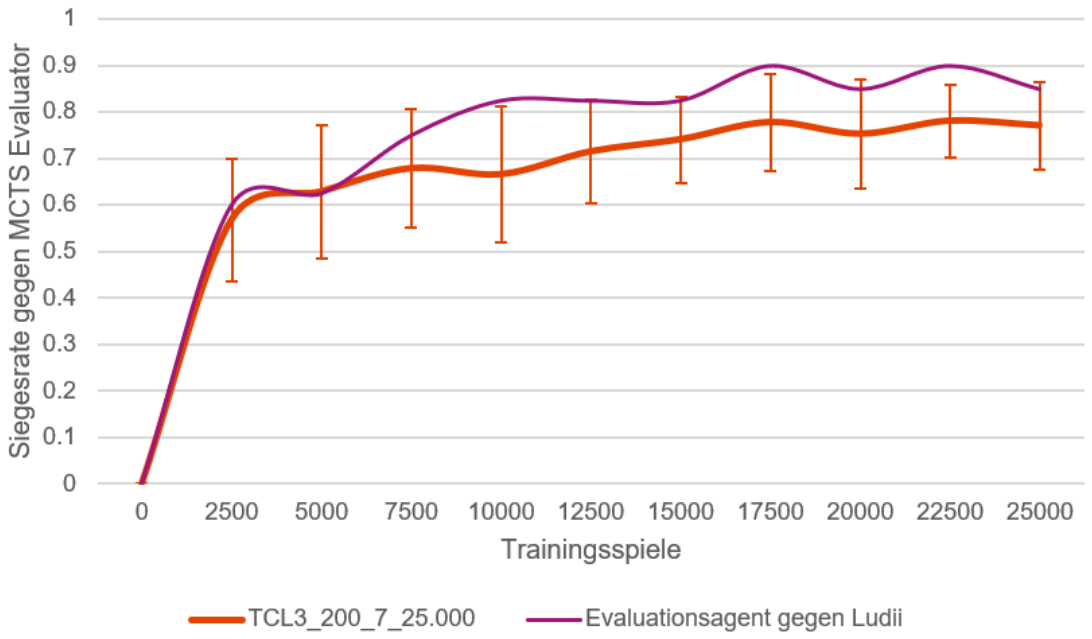

Abbildung 24: Trainingsergebnisse des finalen Agenten TCL3_200_7_25.000

Abbildung 24 zeigt die Ergebnisse dieses so trainierten Agenten. Es wurden 25 verschiedene Agenten trainiert und die Ergebnisse dann gemittelt. Die Resultate sind etwa deckungsgleich mit denen des vorherigen Agenten über 100.000 Trainingsspiele. Beide erreichen Siegesraten von 80% gegen den Evaluator bei circa 20.000 Spielen. Die Erhöhung der Tupel von 100 auf 200 scheint keinen signifikanten Unterschied mehr gemacht zu haben.

Der Agent, der jetzt gegen Ludii antreten soll, ist, wie bei Multitrain im GBG Framework üblich, der zuletzt trainierte Agent. Dieser liegt eher am oberen Ende der Abweichung vom Mittel, dargestellt durch die lilafarbene Linie in Abbildung 24.

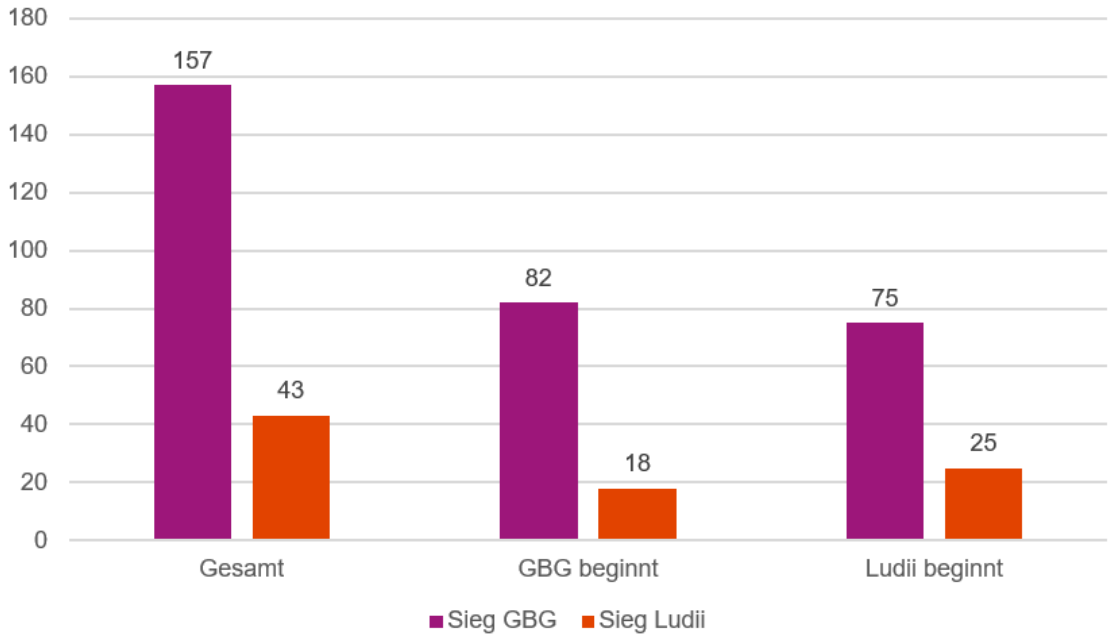

Abbildung 25: Evaluation TCL3_200_7_25.000 gegen Ludii Biased MCTS

Es wurden, wie vorher bereits auch, jeweils 100 Evaluationsspiele pro Seite ausgetragen. Die Ergebnisse davon sind in Abbildung 25 zu sehen. Über beide Seiten gewinnt der neue Agent 78.5% der Spiele gegen den Biased MCTS Agenten von Ludii. Im Vergleich zur letzten Evaluation gegen Ludii wurden wie zu erwarten war einige Prozent in der Siegesrate eingebüßt. Überraschend ist, dass dies hauptsächlich bei den Spielen passiert ist, bei denen der Agent selbst angefangen hat.

Trotzdem ist dies ein hervorragendes Ergebnis. Für den Verlust von 6% an der Siegesrate, wurde ein deutlicher Unterschied in der benötigten Anzahl an der Trainingsspiele erkauft. Von ehemals 1.000.000 Spielen konnte dies auf 25.000 gedrückt werden, was einer 40-fachen Reduktion entspricht.

5 Verwandte Werke

5.1 Biased MCTS in Ludii

In einem vorherigen Kapitel wurde schon der Biased MCTS Spieler von Ludii angesprochen, gegen den auch evaluiert wurde. Dieser nutzt die Monte-Carlo-Baumsuche in Verbindung mit einem linearem Funktionsapproximator, um im Selbstspiel zu lernen. Es werden simultan gegebene Features weiterentwickelt, sowie die Gewichtungen gelernt.

Die Vorteile dieser Strategie, im Vergleich zu dem oft sonst üblichen *Deep Neural Network* (DNN) sind unter anderem die Allgemeinheit dieser Methode, sowie der notwenige Rechenaufwand. Üblicherweise müssen für jedes Spiel zeitaufwendige Features definiert werden und es wird teure Hardware benötigt und auch ein langwieriger Lernprozess. Biased MCTS umgeht diese Problematiken, wenn auch mit einem Verlust in der Performance einhergehend.

Es baut auf dem Prinzip der Expert Iteration (Anthony, Tian, & Barber, 2017) auf. Dort werden eine Lehrlings- und eine Expertenstrategie verwendet, um sich gegenseitig zu verbessern. Die Lehrlingsstrategie ist üblicherweise trainierbar und recheneffizient, wie ein DNN, während die Expertenstrategie mehr Überlegungen anstellt, wie zum Beispiel die Monte-Carlo-Baumsuche. Es werden die gelernten Verteilungen der Lehrlingsstrategie benutzt, um die Suche des Experten anzupassen und danach werden die Ergebnisse der Expertenstrategie benutzt, um wiederum die Verteilungen des Lehrlings zu verbessern. Die der Lehrlingsstrategie zugrundliegenden Features werden ebenfalls während des Trainings weiterentwickelt (Soemers, Piette, & Browne, 2019).

Diese Methode wurde dann in 10 verschiedenen Spielen gegen einen UCT-MCTS-Agenten getestet. Als Grundlage für den Biased-MCTS-Agenten dienten dabei 200 Selbstspiele zum Lernen. Die Ergebnisse dieses Tests zeigten bereits eine Verbesserung und hohe Siegesrate in mehr als der Hälfte der Spiele. Nach weiteren Anpassungen und erneuten 100 Trainingsspielen, wurde dies noch deutlicher.

Einziger Ausreißer in diesen Daten war das Spiel Yavalath. Während in allen anderen Spielen mindestens Siegesraten gegen UCT von 50% erzielt wurden, brachen diese dort stark ein und tendierten zu 25%. Als Ursache hierfür, wurden schließlich die zugrundeliegenden Features ermittelt. Nachdem ein manuell erstellter Satz von Features benutzt wurde, der auf die Regeln von Yavalath zugeschnitten war, konnte auch für Yavalath eine Siegesrate von 93% gegen den UCT-Agenten erzielt werden (Soemers, Piette, & Browne, 2019, S. 7).

5.2 Deep Learning in Polygames

Polygames ist ein in C++ geschriebenes Open-Source Framework zum Thema Zero Learning und kombiniert die Monte-Carlo-Baumsuche mit Deep Learning, ähnlich wie AlphaZero. Die Architektur benutzt ein Fully Convolutional Neural Network in dem Input und Output Channel dieselben Dimensionen der Spielbrettgröße aufweisen. Dies erlaubt

es, auf geringeren Spielbrettgrößen zu trainieren und die Ergebnisse auf größere zu übertragen. Ein Agent, der auf dem 13x13 Spielbrett von Hex trainiert wurde, war ohne weiteres Training auch direkt auf dem 19x19 Spielbrett ein starker Spieler. Nach einigen Feinanpassungen konnte dieser starke menschliche Spieler auf dem 19x19 Spielbrett besiegen (Cazenave, Chen, Chen, & Chen, 2021).

Da für Polygames, ähnlich wie im GBG-Framework, jedes Spiel einzeln in der Programmiersprache umgesetzt werden muss, wurde eine Schnittstelle zwischen den beiden Systemen geschaffen. Diese ermöglicht es, über ein in Polygames umgesetztes „Ludii-Spiel“ alle in Ludii vorhandenen Spiele und Spielvarianten zu laden und diese mit den in Polygames vorhandenen Agenten zu erproben.

Dabei wird die allgemeine Struktur von Ludii und der dahinterstehenden Spielbeschreibungssprache der Ludeme genutzt, um die Spielzustände von Ludii zu abstrahieren und als Input für das neuronale Netzwerk zu verwenden, ohne diese noch einmal spielspezifisch zu beschreiben.

Um die Umsetzung zu testen, wurden 15 verschiedene Spiele aus der Version 1.1.6 von Ludii ausgewählt und Agenten dafür trainiert. Diese wurden danach in je 300 Evaluationsspielen gegen einen normalen UCT-MCTS-Agenten, der nicht lernt, getestet. Die Ergebnisse des Tests waren recht eindeutig, in der Mehrheit der Spiele wurden überragende Siegesraten erzielt, und nur eines davon war ein starker Ausreißer, Lasca mit einer Siegesrate von nur 3.5%. Als Grund hierfür wurde spekuliert, dass Laska das einzige der Spiele ist, welches das Stapeln von mehreren Spielsteinen auf ein Feld involviert (Soemers D. J., Mella, Browne, & Teytaud, 2021a).

In Yavalath wurde eine Siegesrate von 97.3% erzielt. Leider erlaubt es diese Schnittstelle nicht, dass die Agenten von Polygames gegen die Agenten von Ludii antreten, was einen Vergleich zur Spielstärke der in dieser Arbeit erstellten Agenten erlauben würde. Der engste Vergleich ist wahrscheinlich zu den während der Evaluation in dieser Arbeit genutzten MCTS-Agenten des GBG-Frameworks möglich. Dort hätte der trainierte Polygames Agent mit der Siegesrate von 97.3% im Vergleich zu den TD-N-Tupel-Agenten mit Siegesraten von um die 90% die Nase leicht vorne.

5.3 Transfer von Spielwissen über verschiedene Spielvarianten hinweg in Polygames

Aufbauend auf der im vorherigen Kapitel beschriebenen Schnittstelle zwischen Polygames und Ludii, wurde der Transfer von Spielwissen über verschiedene Spielvarianten und auch Spiele hinweg untersucht (Soemers D. J., et al., 2021b).

Zuerst wurde untersucht, ob Training allein in einer Quellumgebung ausreicht, um einen Agenten, der nur in einer Zielumgebung trainiert wurde, in dieser zu schlagen. Dies wird auch als *Zero-Shot Transfer* bezeichnet. Während dies bei einigen Spielen über mehrere Varianten hinweg der Fall war, trat dies in Yavalath nur in zwei Fällen auf. Es wurden Spielbretter zwischen den Größen 3x3 und 8x8 getestet. Nur die Agenten, die auf den

6x6 und 7x7 Spielbrettern trainiert wurden, hatten einen Vorteil gegen den auf dem 8x8 Spielbrett trainierten Agenten, wenn auf diesem gespielt wurde. Und auch dort betrug die Siegesrate nur 58% für den 6x6, beziehungsweise 53.33% für den 7x7 Agenten. Generell ließ sich sagen, je näher die verschiedenen Spielbrettgrößen dabei waren, desto höhere Siegesraten wurden erzielt.

Als nächstes wurden ausgehend von den vorherigen Agenten noch Feinabstimmungen unternommen, bevor erneut evaluiert wurde. Dazu wurden die Agenten für die gleiche Dauer, die sie in ihrer vorherigen Umgebung trainiert wurden, auch in der Zielumgebung trainiert, in der sie dann spielen sollten. Die Ergebnisse dieses Tests waren deutlich vielversprechender als die vorherigen. Die meisten der Agenten erzielten Siegesraten zwischen 50 und 70%, egal aus welcher Spielgröße sie kamen und in welcher sie dann erneut trainiert und evaluiert wurden. Ausreißer hierbei waren jedoch die Spielbretter der Zielgröße 3x3 und 4x4, in denen die Mehrheit der Agenten noch eine Siegesrate von unter 50% erzielte, auch wenn sich diese im Vergleich zu vorher schon deutlich verbesserte.

Danach wurde der Zero-Shot Transfer von Spielwissen zwischen verschiedenen Spielen derselben Spielgattung untersucht. Für Yavalath waren dies Linienverbindungsspiele, bei denen eine Reihe gebildet werden muss, wie 6 Gewinnt oder Squava. Für Agenten, die in Yavalath trainiert wurden, waren die Ergebnisse eindeutig negativ. Einzig in den Spielen Pentalath und Squava, welche sehr eng mit Yavalath verwandt sind, konnte überhaupt eine Siegesrate jenseits der 0% erzielt werden. Und auch diese betrug jeweils nur 0.33% für Pentalath und 1.67% für Yavalath.

Etwas positiver, wenn auch immer noch nicht gut, sah die Sache für Agenten aus, die in anderen Spielen trainiert wurden und dann in Yavalath spielten. Dort schafften es alle Agenten mindestens eins der 300% Evaluationsspiele für sich zu gewinnen, aber mehr als die 2% des Squava-Agenten konnte nicht übertroffen werden.

Auch hier konnte wieder nach Feinabstimmungen ein deutlicher Unterschied zu vorher festgestellt werden. Analog wie bei den Spielvarianten, wurde die Agenten der unterschiedlichen Spiele noch für die gleiche Dauer im Zielspiel trainiert, bevor sie getestet wurden. Über die komplette Kategorie der Linienverbindungsspiele hinweg, wurde ein positiver Effekt beobachtet, und fast alle Agenten erreicht Siegesraten über 50%. In Yavalath erreichte der 6 Gewinnt Agent mit 74% die höchste Siegesrate, während andersherum auch der Yavalath-Agent am kompatibelsten mit 6 Gewinnt scheint und dort eine Siegesrate von 70.33% erreicht.

Zusammenfassend lässt sich sagen, dass gerade für das Beispiel Yavalath der Transfer von Spielwissen über Spielvarianten oder auch Spiele hinweg, nur Sinn macht, wenn danach Feinabstimmungen vorgenommen werden. Ohne diese werden nur in seltenen Fällen passable Ergebnisse erzielt.

6 Zusammenfassung und Ausblick

Diese Arbeit baute auf zwei vorherigen Projekten auf, in denen zuerst das Spiel Yavalath innerhalb des GBG Frameworks umgesetzt wurde (Weitz, 2021), und danach die Schnittstelle zwischen Ludii und dem GBG Framework verallgemeinert, sowie erneuert wurde (Weitz, 2022). Auf diesem Grundstein aufbauend, befasste sich diese Arbeit dann mit dem Erfolg von selbstlernenden Reinforcement Learning Agenten im Spiel Yavalath. Der Hintergrund von Yavalath als ein computergeneriertes Spiel, dass jetzt wiederum von Computern erlernt werden sollte, bot einen interessanten Hintergrund.

Es wurden verschiedene Strategien zur Evaluation der Agenten entwickelt. Da kein perfekt spielender Yavalath-Agent, der als idealer Evaluationsgegner dienen könnte, bekannt ist, wurden mehrere auf Monte-Carlo-Baumsuche aufbauende Ansätze genutzt. Darunter befand sich auch der Biased MCTS Agent des Ludii Frameworks, welcher ebenfalls über Training mit sich selbst lernt.

Relativ schnell wurden dadurch zufriedenstellende Agenten gefunden, die das Spiel gut lernen konnten. Im weiteren Verlauf der Arbeit wurde dieses Ergebnis noch verfeinert, in dem der Einfluss unterschiedlicher Parameter und Methoden untersucht wurde. Dadurch konnte auch die Anzahl benötigter Trainingsspiele drastisch gesenkt werden.

Leider konnten aufgrund der Zeitbeschränkung dieser Arbeit nicht alle aufkommenden Fragen beantwortet und Denkansätze erkundet werden. Einige dieser werden im Folgenden aufgeführt:

- In dieser Arbeit wurde nur die 2-Spieler-Variante von Yavalath betrachtet. Interessant wäre, ob ein ähnlicher Lernerfolg auch in der 3-Spieler-Variante erzielt werden kann. Spiele mit mehr als 2 Spielern fanden in der Forschung in diesem Themengebiet bis jetzt noch nicht viel Beachtung (Konen, 2019), weshalb dies ein sehr interessantes Thema darstellt.
- Als weitere Variante von Yavalath, wäre auch der Lernerfolg auf Spielbrettgrößen außerhalb des klassischen 5x5 Spielbretts interessant. Von (Soemers D. J., et al., 2021b) wurde schon gezeigt, dass diese erfolgreich gelernt werden können, doch werden dort nur verschiedene Varianten verglichen. Zwei interessante Fragen wären hier, wie hoch kann der Lernerfolg sein, und wie schnell tritt dieser ein, wenn die Größe des Spielbrettes verändert wird.
- Von (Scheiermann, 2020) wurde ein, auf den AlphaZero zugrundeliegenden Prinzipien aufgebauter, MCTS-Wrapper umgesetzt, der vorhandene Temporal-Difference Agenten um einen Zukunftsblick erweitert. Für das Spiel Othello konnte damit eine signifikante Verbesserung erzielt werden. Da der Wrapper innerhalb des GBG Frameworks auf Arena Seite geladen wird, war ein Einsatz gegen die Ludii Agenten nicht möglich. Erste Versuche die Schnittstelle analog anzupassen waren nicht erfolgreich. Dasselbe gilt für den MaxN-Wrapper. Während die Spiele mit diesem zwar problemlos durchliefen, war das Ergebnis von einzelnen

Spielen katastrophal und der Agent übersah die offensichtlichsten Züge, die zur Niederlage führten. Ob dies ein programmiertechnisches oder spielspezifisches Problem war, konnte in der geringen Testzeit dafür nicht ermittelt werden.

- Evaluationen während des Trainingsprozesses erfolgen innerhalb des GBG Frameworks momentan auch nur gegen Agenten, die dort schon umgesetzt wurden. Interessant wäre es, wenn bereits während der Evaluation die Ludii-Schnittstelle genutzt werden könnte, und somit bei einigen Spielen stärkere Evaluationsgegner zur Verfügung stünden.

7 Literaturverzeichnis

Anthony, T., Tian, Z., & Barber, D. (2017). Thinking fast and slow with deep learning and tree search. *Advances in Neural Information Processing Systems 30 (NIPS 2017)* (S. 5360-5370). Curran Associates, Inc., 2017.

Bagheri, S., & Thill, M. (2014). Temporal Coherence in TD-Learning for Strategic Board Games.

Beal, D. F., & Smith, M. C. (Juli 1999). Temporal coherence and prediction decay in TD learning. *IJCAI'99: Proceedings of the 16th international joint conference on Artifical intelligence - Volume 1*, S. 564–569.

Browne, C. (2011). *Evolutionary Game Design.* London: Springer.

Browne, C., Poley, E. J., Whitehouse, D., Lucas, S., Cowling, P. I., Rohlfshagen, P., . . . Colton, S. (März 2012). A Survey of Monte Carlo Tree Search Methods. *IEEE Transactions on Computational Intelligence and AI in Games, Volume 4, Issue 1*, S. 1-43.

Browne, C., Stephenson, M., Piette, E., & Soemers, D. (2019). A Practical Introduction to the Ludii General Game System. *Proceedings of Advances in Computer Games (ACG 2019).* Macau: Springer.

Cazenave, T., Chen, Y.-C., Chen, G.-W., & Chen, S.-Y. et al (2021). Polygames: Improved Zero Learning. *ICGA Journal*, S. 244-256.

Galitzki, K. (2017). Selbstlernende Agenten für das skalierbare Spiel Hex: Untersuchung verschiedener KI-Verfahren im GBG-Framework. Bachelorarbeit TH Köln. https://www.gm.fh-koeln.de/~konen/research/PaperPDF/BA-KevinGalitzki-final-2017.pdf abgerufen am 14.04.2022

Ingle, S., & Phute, M. (September 2016). Tesla Autopilot : Semi Autonomous Driving, an Uptick for Future. *International Research Journal of Engineering and Technology*, S. 369-372.

Kaplan, A., & Haenlein, M. (2019). Siri, Siri, in my hand: Who's the fairest in the land? On the interpretations, illustrations, and implications of artificial intelligence. *Business Horizons Volume 62, Issue 1*, S. 15-25.

Konen, W. (2015). *Reinforcement Learning for Board Games: The Temporal Difference Algorithm.* Research Center CIOP (Computational Intelligence, Optimization and Data Mining), Cologne University of Applied Science.

Konen, W. (2019). General Board Game Playing for Education and Research in Generic AI Game Learning. *2019 IEEE Conference on Games (CoG).* London.

Konen, W. (Oct 2021). *The GBG Class Interface Tutorial V2.3.*

Konen, W., & Bartz-Beielstein, T. (2008). Reinforcement Learning: Insights from Interesting Failures in Parameter Selection. *Parallel Problem Solving from Nature – PPSN X*, S. 478-487.

Korf, R. E. (February 1991). Multi-player alpha-beta pruning. *Artificial Intelligence Volume 48 Issue 1*, S. 99-111.

Lee, C.-S. (August 2016). Human vs. Computer Go: Review and Prospect. *IEEE Computational Intelligence Magazine*, S. 67-72.

Lucas, S. M. (2008). Learning to Play Othello. *Australian Journal of Intelligent* , S. 1-20.

Rosin, C. D. (2011). Multi-armed bandits with episode context. In J. Franco, *Annals of Mathematics and Artificial Intelligence, Volume 61, Issue 3* (S. 203-230). Springer Science+Business Media.

Scheiermann, J. (2020). *Sind (trainierte) general-purpose RL Agenten im Brettspiel Othello stärker als (untrainierte) General-Game-Playing Agenten?*. Informatikprojekt TH Köln. https://www.gm.fh-koeln.de/~konen/research/PaperPDF/INF-Prj-Scheiermann-2020-08.pdf abgerufen am 13.04.2022.

Silver, D., Schrittwieser, J., & Simonyan, K. (October 2017). Mastering the Game of Go without Human Knowledge. *Nature*, S. 354-359.

Soemers, D. (2021c). *Ludii Github Repository.* Von https://github.com/Ludeme/Ludii/blob/b03afb983251b5b9b28a1931da54be97a1e4c182/AI/src/search/mcts/selection/AG0Selection.java abgerufen

Soemers, D. J., Mella, V., Browne, C., & Teytaud, O. (2021a). Deep Learning for General Game Playing with Ludii and Polygames. *CoRR, abs/2101.09562*.

Soemers, D. J., Mella, V., Piette, E., Stephenson, M., Browne, C., & Teytaud, O. (2021b). Transfer of Fully Convolutional Policy-Value Networks Between Games and Game Variants. *CoRR, abs/2102.12375*.

Soemers, D. J., Piette, E., & Browne, C. (2019). Biasing MCTS with Features for General Games. *CoRR, abs/1903.08942*.

Sutton, R. S. (1988). Learning to Predict by the Methods of Temporal Differences. *Machine Learning 3*, S. 9-44.

Weitz, A. (2021). *Projektbericht zur Umsetzung des Spiels Yavalath innerhalb des General Board Game Playing Frameworks.* Informatikprojekt TH Köln.

Weitz, A. (2022). *Entwicklung einer allgemeinen Schnittstelle zwischen Ludii und dem GBG Framework.* Praxisprojekt TH Köln. https://www.gm.fh-koeln.de/~konen/research/PaperPDF/PP-Doku-Weitz-2022-02.pdf abgerufen am 15.04.2022

Anhang

A.1

Das folgende Verzeichnis zeigt in welchen Abbildungen die unterschiedlichen Agenten genutzt wurden und unter welchem Namen diese in den Einstellungstabellen wiederzufinden sind. Dies soll zusammen mit den Einstellungstabellen dabei helfen, die Ergebnisse dieser Arbeit leicht reproduzieren zu können.

Abbildung	Label	Einstellung
10	MCTS (10.000)	MCTS1
11	MCTS (25.000)	MCTS2
12	TDN3_10_6_100.000	TDN1
13	10 Tupel	TDN2
13	25 Tupel	TDN3
13	50 Tupel	TDN4
13	100 Tupel	TDN5
14,21,22	TDN3_100_7_300.000	TDN6
15,18	TDN3_100_7_1.000.000	TDN7
20	TDN3_100_7_300.000_NO-SYM	TDN8
21	TDN3_100_7_300.000_L0.2	TDN9
21	TDN3_100_7_300.000_L0.7	TDN10
21	TDN3_100_7_300.000_L1.0	TDN11
22	TCL3_100_7_300.000	TCL1
23	TCL3_100_7_100.000	TCL2
24,25	TCL3_100_7_25.000	TCL3

Tabelle 2: Zuordnung der verschiedenen Agenten zu den Abbildungen in denen sie genutzt werden und deren Namen in den Einstellungen

A.2

Einstel-lung	Iterati-ons	Selector	Tree Depth	K (UCT)	Rollout Depth	Norma-lize
MCTS1	10.000	UCT	25	1.414	2.000	True
MCTS2	25.000	UCT	25	1.414	2.000	True

Tabelle 3: Einstellungen der MCTS Agenten

A.3

Ein-stel-lung	α_{init}	α_{final}	ε_{init}	ε_{final}	λ	γ	Spiele	Tupel	Tupel Länge	Anmer-kungen
TDN1	0.2	0.2	0.3	0.0	0.0	1.0	100.000	10	6	-
TDN2	1.0	0.2	0.3	0.05	0.0	1.0	150.000	10	7	-
TDN3	1.0	0.2	0.3	0.05	0.0	1.0	150.000	25	7	-
TDN4	1.0	0.2	0.3	0.05	0.0	1.0	150.000	50	7	-
TDN5	1.0	0.2	0.3	0.05	0.0	1.0	150.000	100	7	-
TDN6	1.0	0.2	0.3	0.05	0.0	1.0	300.000	100	7	-
TDN7	1.0	0.2	0.3	0.05	0.0	1.0	1.000.000	100	7	-
TDN8	1.0	0.2	0.3	0.05	0.0	1.0	300.000	100	7	NO SYM
TDN9	1.0	0.2	0.3	0.05	0.2	1.0	300.000	100	7	ET
TDN10	1.0	0.2	0.3	0.05	0.7	1.0	300.000	100	7	ET
TDN11	1.0	0.2	0.3	0.05	1.0	1.0	300.000	100	7	ET
TCL1	1.0	0.2	0.3	0.05	0.0	1.0	300.000	100	7	TC
TCL2	1.0	0.585	0.3	0.21666	0.0	1.0	100.000	100	7	TC
TCL3	1.0	0.6	0.3	0.1	0.0	1.0	25.000	200	7	TC

Tabelle 4: Einstellungen der TDN und TCL Agenten. Erklärung der Anmerkungen: NO SYM = Keine Symmetrie genutzt, ET = Eligibility Traces wurden verwendet, TC = Temporal Coherence wurde verwendet

B.1

Die nachfolgende Tabelle zeigt die durchschnittliche Trainingsdauer, die bei verschiedenen Anzahlen von Spielen benötigt wird. Diese wurde von jeweils von mehreren Agenten gemittelt.

Trainingsspiele	Dauer (in Sek.)	Standardabweichung	Agenten
25.000	784s	± 303 Sekunden	29
100.000	1798s	± 196 Sekunden	10
150.000	2543s	± 878 Sekunden	10
300.000	8807s	± 905 Sekunden	15
1.000.000	29.041s	± 4807 Sekunden	4

Tabelle 5: Trainingsdauer bei unterschiedlichen Anzahlen von Trainingsspielen

Zusätzlich zu der reinen Trainingszeit, nimmt auch jede Evaluation des Agenten während des Trainings Zeit in Anspruch. Dies ist von Evaluator zu Evaluator unterschiedlich und in der nachfolgenden Tabelle aufgelistet. Die Evaluatoren, die mit *DiffStarts* gekennzeichnet sind, evaluieren in jeder Episode ausgehend von einem normalen Start mit leerem Brett auch alle 1-Ply-States, Spielzustände, bei denen schon der erste Stein gesetzt wurde.

Evaluator	Anmerkung	Episoden	Evaluationsdauer
Random	-	10	>1 Sekunde
MaxN	N = 2	10	60 ± 5 Sekunden
MCTS	Entspricht MCTS1 aus Tabelle 3	10	35 ± 6 Sekunden
MaxN-DiffStarts	N = 2	1	378 ± 20 Sekunden
MCTS-DiffStarts	Entspricht MCTS1 aus Tabelle 3	1	221 ± 7 Sekunden

Tabelle 6: Vorhandene Möglichkeiten zur Evaluation und deren Dauer pro Vorgang

Zu beachten ist auch, dass sowohl Trainings-, als auch Evaluationsdauer aufgrund der zugrundeliegenden Hardware deutlich variieren können.

Der gesamte entwickelte Quellcode ist in dem GitHub-Repository

https://github.com/WolfgangKonen/GBG einsehbar.