Technology
Arts Sciences
TH Köln

Gummersbach, im Juli 2017

Selbstlernende Agenten für das skalierbare Spiel Hex:
Untersuchung verschiedener KI-Verfahren im GBG-Framework

Bachelorarbeit

ausgearbeitet von

Kevin D. Galitzki

zur Erlangung des akademischen Grades

Bachelor of Science (B.Sc.)

vorgelegt an der

Technischen Hochschule Köln

Campus Gummersbach

Fakultät für Informatik und Ingenieurwissenschaften

Erster Prüfer:	Prof. Dr. Wolfgang Konen Technische Hochschule Köln
Zweiter Prüfer:	Prof. Dr. Christian Kohls Technische Hochschule Köln
Bearbeitungsdauer:	Neun Wochen
Studiengang:	Allgemeine Informatik
Matrikelnummer:	11083034

Kurzfassung

Diese Arbeit dokumentiert die Anwendung verschiedener KI-Verfahren auf das Brettspiel Hex. Im Fokus steht dabei das Training der Agenten durch *Self-Play*, ohne jeglichen Lehrer oder spieltheoretisches Vorwissen. Dabei werden die Vorteile und die Grenzen der verschiedenen Verfahren analysiert und zusammengefasst. Neben dem Minimax-Algorithmus kommt mit der *Monte-Carlo Tree Search* auch eine moderne Form der Baumsuche zum Einsatz. Weiterhin werden zwei verschiedene Ausprägungen des *Temporal DIfference Learnings* eingesetzt, um die Spielfunktion von Hex zu erlernen.

Die für diese Arbeit implementierte Variante des Spiels Hex ist vollständig in seiner Größe skalierbar, sodass die Leistung der verschiedenen KI-Verfahren auch in den Kontext der Komplexität des Spielbaums gesetzt werden kann. Weiterhin wurden Möglichkeiten implementiert, die Bewertungen der einzelnen Agenten auf dem Spielbrett zu visualisieren, um deren Entscheidungsprozess nachvollziehen zu können.

Mit dem Temporal-Difference-Agenten, der ein N-Tupel-System als Merkmal verwendet, wurde ein gut funktionierendes Verfahren für das Erlenen der Spielfunktion von Hex gefunden. Bis einschließlich dem 7×7-Spielbrett konnte der Weltmeister im Computer-Hex aus dem Jahr 2000 besiegt werden, ohne dass dem Agenten Vorwissen über das Spiel vermittelt wurde. Dadurch, dass der N-Tupel-Agent seine Merkmale prozedural generieren kann, eignet er sich sehr gut für den Einsatz auf einem skalierbaren Spielbrett.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1. Einleitung

1.1 Motivation

Maschinelles Lernen ist ein Thema, dass eine immer größer werdende Bedeutung für das alltägliche Leben bekommt. Schon seit langer Zeit gibt es Maschinen, die den Menschen mühsame, monotone Arbeit abnehmen können. Durch Fortschritte im Bereich der künstlichen Intelligenz wird es jedoch immer denkbarer, dass Maschinen auch Aufgaben übernehmen können, die Kreativität und eine „menschliche Hand“ benötigen. Betrachtet man das menschliche Gehirn als ein Netzwerk aus Neuronen und Synapsen das über Jahrzehnte mit Eingabedaten trainiert wird, ist es vorstellbar, dass Computer die Leistung unseres Gehirns irgendwann übertreffen können.

Stetig anwachsende Sammlungen an Trainingsdaten und immer bessere Algorithmen ermöglichen es Computern, erstaunliche Aufgaben zu lösen. Es wird erwartet, dass in wenigen Jahren Autos erhältlich sind, die vollkommen selbstständig und vor allem sicherer als Menschen fahren können. Bilder, Videos und Texte werden schon heute automatisch nach ihren Inhalten klassifiziert. Spracherkennung und -synthese ist so weit fortgeschritten, dass einfache Unterhaltungen mit Maschinen gehalten werden können. All diese Dinge haben gemeinsam, dass es eine praktisch unbegrenzte Menge verschiedener Eingabedaten gibt, für die nicht alle eine korrekte Ausgabe vorprogrammiert werden kann. Durch die Anwendung von maschinellen Lernverfahren wie zum Beispiel *Reinforcement Learning* sind Maschinen in der Lage, selbstständig Lösungen zu finden oder anzunähern. Bis zur Entwicklung einer allgemeinen künstlichen Intelligenz ist es jedoch noch ein langer Weg, weshalb im Bereich des maschinellen Lernens sehr viel Forschung und Entwicklung betrieben wird.

Spiele stellen einen guten Einstieg in diesen Bereich dar. Obwohl Computer für die Lösung mathematischer Probleme erfunden wurden, sind viele Spiele so komplex, dass die vollständige Lösung heutzutage noch nicht möglich ist. Maschinelles Lernen ermöglicht Computern, eine hohe Spielleistung zu erreichen, ohne dass jeder Zustand des Spiels berechnet wurde und ohne dass von Menschen komplizierte, maßgeschneiderte Heuristiken entworfen werden. Insbesondere wurde für diese Arbeit das Spiel Hex gewählt, da das Spielbrett von Hex beliebig in seiner Größe skaliert werden kann. Dadurch kann mit der Untersuchung kleiner Spielbretter begonnen und die Komplexität dann nach und nach erhöht werden.

1.2 Aufgabenstellung

Ziel dieser Arbeit ist eine Untersuchung der im GBG-Framework implementierten KI-Verfahren am Beispiel des Brettspiels Hex. Das „General Board Game Playing and Learning Framework“ implementiert eine Reihe verschiedener Agenten, die ohne Vorwissen und allein durch Self-Play die für das Framework entwickelten Spiele lernen können [1].

Hierzu wird Hex unter Verwendung der vom GBG-Framework angebotenen Interfaces implementiert. Anschließend werden die einzelnen Agenten des GBG-Frameworks am Spiel getestet und deren Leistung im Detail analysiert. Das Ziel ist dabei nicht, Agenten zu erstellen, die mit den weltbesten Hex-Agenten mithalten können. Im Fokus steht viel mehr eine Untersuchung, ob im relativ kurzen Zeitrahmen einer Bachelorarbeit zumindest für einige Spielbretter ein gut spielender Agent entwickelt werden kann. Weiterhin ist die Frage, welche Leistung erreicht werden kann, wenn nach dem Paradigma „General Board Game Playing“ kein spielspezifisches Wissen verwendet wird, um die Leistung der Agenten zu verbessern. Dass ein erfolgreiches Lernen der Spielfunktion durch alle Agenten möglich ist, ist dabei zu Beginn der Arbeit noch nicht garantiert.

1.3 Gliederung

Das zweite Kapitel dieser Arbeit befasst sich mit dem Spiel Hex. Neben den Spielregeln und ein paar Strategien wird im Anschluss auch ein Überblick über den Stand der Forschung im Bereich der maschinellen Hex-Spielern geboten. Kapitel drei beschreibt sowohl das GBG-Framework als auch die Funktionsweise der für Hex relevanten Agenten auf eine verständliche Art und Weise. Im vierten Kapitel wird erläutert, wie die Leistung der Agenten gemessen wird und welche Schwierigkeiten dabei auftreten können. Kapitel fünf beschreibt sowohl die Erwartungen an die einzelnen Agenten als auch die tatsächlich gemessenen Ergebnisse. Darauf folgt in Kapitel 6 eine Diskussion, in der die Ergebnisse bewertet und Vorschläge für weitere Arbeiten gegeben werden. Im letzten Kapitel wird die Arbeit noch einmal zusammengefasst.

2. Hex

2.1 Grundlagen

2.1.1 Spielregeln

Hex ist ein rundenbasiertes Strategiespiel, bei dem zwei Spieler abwechselnd Steine auf noch unbesetzte Felder legen. Erfunden wurde das Spiel in den 1940ern von Piet Hein und ein paar Jahre danach unabhängig auch vom Nobelpreisträger John Nash. Die Felder des Spielbretts haben eine hexagonale Form, die dem Spiel seinen Namen verleiht. Den Spielern werden sowohl farblich markierte Spielsteine als auch zwei gegenüberliegende Spielbrettseiten in der gleichen Farbe zugewiesen, üblicherweise schwarz und weiß oder rot und blau. Schwarz, beziehungsweise rot, hat den ersten Zug[1]. Ziel des Spiels ist es, mit den eigenen Spielsteinen eine zusammenhängende Kette zu bilden, die die beiden Ränder mit der Farbe der eigenen Spielsteine verbindet.

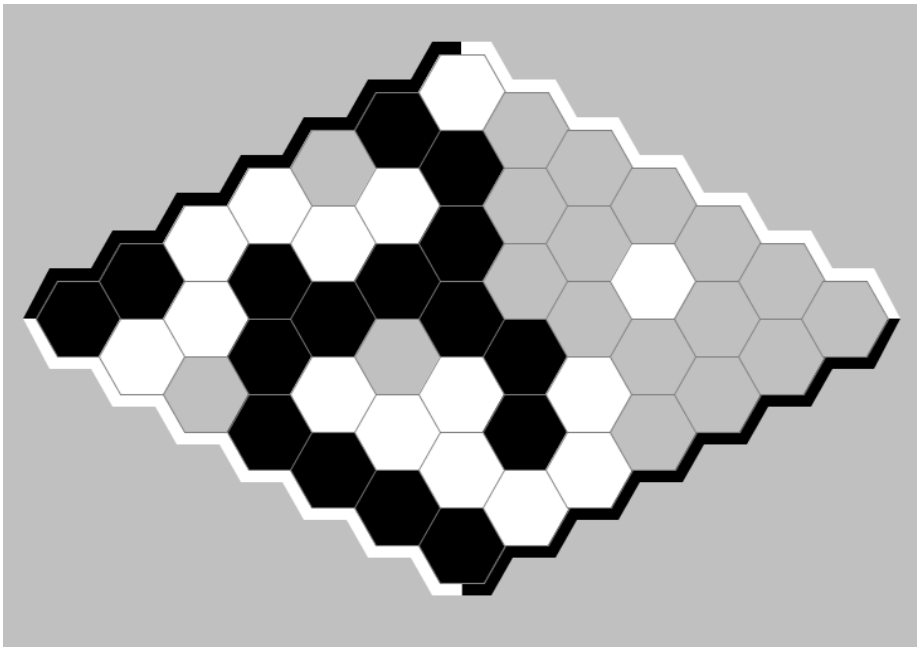

Abbildung 1: Ein 7x7-Spielbrett, bei dem der Spieler mit den schwarzen Steinen gewonnen hat.

Das Spielbrett besteht in der Regel aus $n \times n$ Feldern, die meistens in Form einer Raute angeordnet sind. Übliche Größen für das Spielbrett sind 11×11 oder 13×13, wobei das Spielbrett auch eine beliebige andere Größe annehmen kann. Es muss am Ende des Spiels immer einen Gewinner geben, ein Unentschieden ist nicht möglich. Ein Beweis dafür wurde von David Berman im Jahr 1976 veröffentlicht [2]. Weiterhin zeigte John Nash im Jahr 1952 mit einem Widerspruchsbeweis, dass es immer eine Strategie geben muss, mit der der zuerst ziehende Spieler sicher gewinnen kann [3]. Gäbe es eine optimale Strategie für den zweiten Spieler, dann könnte der erste Spieler in seinem ersten Zug einfach einen zufälligen Stein

[1] Ein Zug bezeichnet hier die Aktion eines Spielers. Unter Verwendung von Schach-Terminologie wäre dies ein Halbzug.

setzen und dann annehmen, er würde als zweiter Spieler spielen. Einen zusätzlichen Stein auf dem Spielbrett zu haben kann in Hex keinen Nachteil bringen. Deshalb kann es keine optimale Strategie für den zweiten Spieler geben. Deshalb hat der erste Spieler einen signifikanten Vorteil, obwohl für die üblicherweise verwendeten Spielbrettgrößen keine optimale Strategie bekannt ist. Die meisten Spieler gehen jedoch davon aus, dass das mittlere Feld auf jeder Spielbrettgröße zu den besten Eröffnungszügen gehört.

In der Praxis kommt oft die sogenannte „Swap Rule“ (Tauschregel) zum Einsatz. Nach dieser Regel für das Spielbrett mit $n \times n$ Feldern hat der zweite Spieler direkt im Anschluss an den ersten Zug des ersten Spielers die Möglichkeit, die Farben beider Spieler neu festzulegen. Entscheidet der zweite Spieler, dass er die Farben tauschen möchte, übernimmt er den bereits platzierten Stein und der Spieler der diesen Stein platziert hat führt auch den zweiten Zug aus, spielt aber von dort an mit der anderen Farbe. Wird mit dieser Regel gespielt, hat der Spieler mit dem Entscheidungsrecht die gewinnbringende Strategie, jedoch wird das Spiel als deutlich fairer angesehen, als wenn ohne diese Regel gespielt wird. Diese Sonderregel wird jedoch bei der für diese Arbeit implementierten Version von Hex nicht verwendet.

2.1.2 Merkmale

Um die Komplexität eines Spiels zu beurteilen, werden gewisse Merkmale betrachtet. Unter anderem sind dies der Verzweigungsfaktor, die Zustandsraum-Komplexität und die Spielbaumgröße.

Der Verzweigungsfaktor eines Spiels beschreibt, wie viele möglichen Aktionen einem Spieler zu einem gewissen Zeitpunkt zur Verfügung stehen. Da die einzige Beschränkung bei der Platzierung eines Steins ist, dass ein unbelegtes Feld gewählt werden muss, ist der Verzweigungsfaktor in Hex vergleichsweise hoch. Bei einem $n \times n$-Spielbrett startet das Spiel mit einem Verzweigungsfaktor von n^2 und nimmt mit jedem Zug um eins ab. Ein 11×11-Spielbrett hat somit zu Beginn einen Verzweigungsfaktor von 121. Die meisten anderen Spiele, wie zum Beispiel Schach, haben entweder weniger Felder oder Regeln, die die möglichen Züge weiter eingrenzen, und damit einen geringeren Verzweigungsfaktor als Hex.

Die Zustandsraum-Komplexität eines Spiels beschreibt die Anzahl aller gültigen Zustände eines Spiels. Da dieser Wert oftmals schwierig zu berechnen ist, wird meistens eine Abschätzung angegeben. Für jedes Feld, um das die Länge einer Spielbrettseite erhöht wird, steigt nach Henderson et al. die Anzahl der möglichen legalen Spielzustände um mehrere Größenordnungen an [4]. Van den Herik et al. schätzten im Jahr 2002 die Größe des Zustandsraums beim 11×11-Spielbrett auf 10^{57}, im Vergleich zu 10^{46} bei Schach [5].

Mit der Spielbaumgröße ist die Anzahl aller Blattknoten im Suchbaum des Spiels gemeint und beschreibt damit die Anzahl aller gültigen Spielverläufe. Dieser Wert ist um einiges größer als die Zustandsraum-Komplexität, da in der Regel mehrere Spielverläufe zum gleichen Zustand führen können. Henderson et al. schätzen die Anzahl aller möglichen legalen Zugfolgen auf ungefähr die Anzahl aller Spielbrettzustände, bei denen das Spielbrett

bis zu halb voll ist [4] (vgl. Abbildung 2). Van den Herik et al. schätzen die Spielbaumgröße von Hex beim 11×11 -Spielbrett auf 10^{98}, die von Schach jedoch mit 10^{123} auf erheblich höher [5].

1×1	2×2	3×3	4×4	5×5	6×6	7×7	8×8
1	9	554	7.6e5	4.0e9	4.0e14	1.5e20	1.0e27

Abbildung 2: Anzahl aller Spielzustände, bei denen das Spielbrett bis zu halb voll ist.
Quelle der Grafik: [4], p. 505

Weiterhin hat das Spielbrett von Hex genau eine Symmetrie, und zwar die Rotation um 180°.

2.1.3 Strategie

Aufgrund der hohen Komplexität von Hex wird es schon bei kleinen Spielbrettern unmöglich, eine einfache Baumsuche zu verwenden, um eine optimale Strategie zu finden. Für gutes Spielen ist es essentiell, lokale Muster auf dem Spielbrett zu erkennen. Eins der wichtigsten Muster wird in der Literatur häufig als „*Virtual Connection*“ (virtuelle Verbindung) bezeichnet. Nach Hayward et al. [6] besteht zwischen zwei Steinen eines Spielers eine virtuelle Verbindung, wenn der Spieler die beiden Steine sicher verbinden kann, selbst wenn zuerst der gegnerische Spieler am Zug ist. Weiterhin nennen Hayward et al. Verbindungen, die nur während des Zug des Spielers zustande kommen können, „Weak Connection“ (schwache Verbindung) oder „Prelink“. Virtuelle Verbindungen können schon einen Hinweis auf den Ausgang des Spiels geben, bevor die Verbindungen tatsächlich hergestellt wurden. Abbildung 3 zeigt ein Beispiel für eine solche virtuelle Verbindung. Selbst wenn der weiße Spieler am Zug ist, kann er nicht verhindern, dass die beiden Steine des schwarzen Spielers verbunden werden.

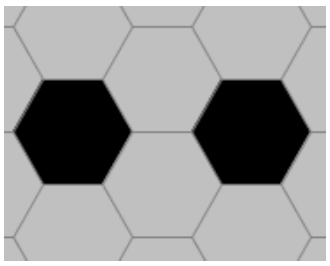

Abbildung 3: Virtuelle Verbindung zweier Steine des schwarzen Spielers

Van Rijswick [7] beschreibt ein weiteres Muster, das den virtuellen Verbindungen ähnlich ist. Sogenannte „Decomposition Patterns“ (Zerfallsmuster) stellen ebenfalls eine sichere Verbindung zwischen zwei Gruppen von Steinen her, können jedoch sogar gegnerische Steine enthalten und nicht nur leere Felder.

Henderson et al. [4] beschreiben außerdem das Konzept von toten Zellen, bei denen eine Menge von leeren Feldern so von den Steinen eines oder beider Spieler eingeschlossen sind, dass es keinem Spieler einen Vorteil verschaffen würde, auf diesen Feldern einen Stein zu setzen. Die dunkelgrauen Felder in Abbildung 4 sind solche toten Zellen.

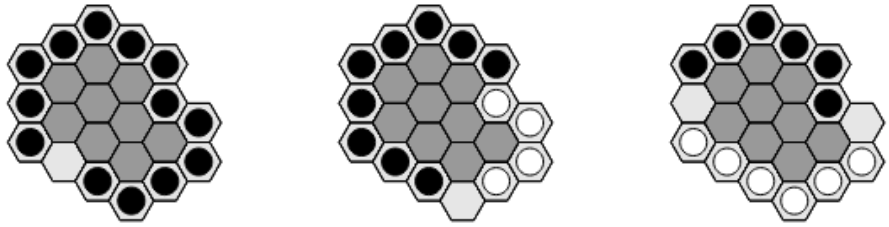

Abbildung 4: Beispiel für tote Zellen (dunkelgrau)
Quelle der Grafik: [4], p. 507

Diese Muster können in jeweils in mehreren Varianten vorkommen. Der Algorithmus von Henderson et al. erkennt für zur vollständigen Lösung des 8×8-Spielbrettes insgesamt 273 verschiedene lokale Muster [4]. Gute Spieler, egal ob menschlich oder maschinell, müssen ebenfalls ein Verständnis für die in Hex auftretenden Muster haben.

2.2 Stand der Forschung

2.2.1 Lösungen des Spiels

Aufgrund der hohen Komplexität ist es schwierig, vollständige Lösungen für Hex zu finden. Deswegen wird auch heute noch nach Lösung für immer größere Hex-Spielbretter gesucht.

Hayward et al. [6] fanden im Jahr 2004 die Lösungen für jeden der 49 Eröffnungszüge auf dem 7×7-Spielbrett. Neben virtuellen Verbindungen machten sie dabei von Regeln gebrauch, die den Zustandsraum verkleinern. Züge, die beweisbar schlechter sind als andere, wurden ignoriert, um die Berechnung zu beschleunigen.

Henderson et al. [4] lösten im Jahr 2009 alle 8×8-Eröffnungszüge von Hex. Der Algorithmus baut dabei auf dem von Hayward et al. auf. Der Teil des Algorithmus, der für die Erkennung von minderwertigen Zügen verantwortlich ist, wurde stark erweitert. Wie bereits erwähnt erkennt dieser Algorithmus 273 verschiedene Muster und kann deren Bedeutung auf den Endzustand des Spiels einschätzen. Die Berechnung aller Eröffnungszüge dauerte knapp unter zwei Wochen.

Die 9×9-Eröffnungszüge wurden von Jakub Pawlewicz und Ryan B. Hayward gelöst. Ihre Methodik wurde im Jahr 2014 veröffentlicht [8]. Dabei wurde ein völlig neuer Ansatz verwendet, der nicht auf den Methoden der bisherigen Lösungen aufbaut. Die Technik, bezeichnet als „Scalable Parallel Depth-First Proof Number Search“, verwendet dabei keinerlei Vorwissen über Hex oder dessen mögliche Muster und ist somit auf weitere Probleme anwendbar. Die Gesamtzeit zur Lösung aller Züge wurde nicht angegeben, jedoch dauerte die Lösung des längsten Eröffnungszugs 111 Tage. Selbst ein Eröffnungszug des

10×10-Spielbretts wurde mit diesem Verfahren gelöst, was zum Zeitpunkt der Veröffentlichung noch niemandem gelungen ist. Die Lösung dieses Zugs dauerte 63 Tage.

2.2.2 Wettbewerb

Die *International Computer Games Association* (ICGA) führt jedes Jahr einen Wettbewerb mit dem Namen *The Computer Games Olympiad* durch. Zwischen 2000 und 2013 war neunmal Hex im Wettbewerb enthalten.

Im Jahr 2000 gewann das Programm *Hexy* von Vadim V. Anshelevich. Dies ist auch das einzige Jahr, in dem Hexy an dem Wettbewerb teilgenommen hat. Dem Autor nach verwendet Hexy keine tiefe Baumsuche und konzentriert sich vollkommen darauf, lokale Muster zu entdecken und auszunutzen [9].

Die nächsten drei Wettbewerbe zwischen 2003 und 2006 gewann ein Programm mit dem Namen *Six.* Der Autor, Gábor Melis, verweist bezüglich der Methodik des Programms auf die Publikation von V. V. Anshelevich [9], da sein Programm auf dem Verfahren von Hexy aufbaut. Dabei wird eine Alpha-Beta-Suche[2] verwendet, die aber nur zwei Züge tief geht. Six nahm auch 2008 und 2009 am Wettbewerb teil, erreichte aber jedoch nur jeweils den dritten Platz.

Ein weiteres Programm namens *Wolve* nahm zwischen 2006 und 2011 insgesamt fünfmal am Wettbewerb teil. Im Jahr 2008 erreichte Wolve dabei den ersten Platz. Arneson et al. [10] beschreiben die Vorgehensweise von Wolve als ähnlich zu der von Six, Wolve verbessert jedoch die sogenannte „Inferior Cell Engine (ICE)“, die schlechte Züge aus dem Suchbaum verwirft.

MoHex, das von einem Teil der Entwickler geschrieben wurde, die auch an Wolve gearbeitet haben, gewann seit 2009 jeden Wettbewerb. MoHex baut laut [11] auf der Codebasis von Fuego auf. Fuego ist ein Programm für das Spiel *Go* und implementiert den UCT-Algorithmus, der bei der Monte-Carlo-Tree-Search (kurz: MCTS) zum Einsatz kommt. MoHex kombiniert den Einsatz von MCTS mit den Methoden zur Analyse lokaler Muster, die schon bei Wolve erfolgreich verwendet wurden und erreicht so eine Spielstärke, die derzeit allen anderen bekannten Programmen überlegen ist.

Im Anhang A1 sind die Gewinner der Wettbewerbe tabellarisch zusammengefasst.

2.2.3 Weitere Forschungsergebnisse

Browne et al. veröffentlichten im Jahr 2012 einen umfassenden Bericht über *Monte-Carlo Tree Search* (MCTS), fünf Jahre nachdem die Methode erstmals beschrieben wurde [12]. Die Autoren beschreiben, dass MCTS ohne Vorwissen für eine große Anzahl an Spielen effektiv eingesetzt werden kann und erwähnen dabei auch speziell *General Game Playing* als ein

[2] Optimierte Variante der klassischen Baumsuche (Minimax).

Einsatzgebiet. Durch zusätzliches Wissen, das in Form einer angepassten *tree* oder *simulation policy* implementiert wird, kann die Leistung verbessert werden. Eine kurze Einführung in MCTS folgt in Kapitel 3.3 dieser Arbeit.

Broderick et al., die mit MoHex erfolgreich das *Monte-Carlo-Tree-Search*-Verfahren auf Hex angewandt haben, veröffentlichten im Jahr 2010 einen Bericht darüber, wie sie das Verfahren in Anwendung auf Hex optimiert haben [13]. Dies gelang ihnen, indem sie die Monte-Carlo-Simulationen auf ein reduziertes Spielbrett angewandt haben, anstatt auf das Vollständige. Weiterhin verwendet die *Rollout*- oder Simulationsphase des Algorithmus Vorwissen über das Spiel, um die Effizienz zu erhöhen, wobei jedoch nur ein einziges Muster zur Anwendung kommt.

Auch *Temporal Difference Learning* wurde bereits in Anwendung auf Hex eingesetzt. Kahl et al. verwendeten ein neuronales Netz mit einer *Hidden Layer* um die Spielfunktion des bereits erwähnten Hex-Agenten *Six* zu erlernen [14]. Als Eingabedaten wurden dafür die Zustände aller Felder auf dem Spielbrett verwendet (mit einem Zahlenwert von -1, 0 oder +1) und zusätzlich die Zustandsbewertung von *Six*. Dadurch, dass die Spielfunktion durch ein neuronales Netz erlernt wurde, konnte eine vergleichbar gute Leistung wie die von *Six* erreicht werden, die jedoch für jeden Zug deutlich weniger Zeit benötigt. Diese Ersparnis in der Rechenzeit verwendeten Kahl et al. dann, um eine tiefere Baumsuche als *Six* durchzuführen und somit dessen Leistung zu übertreffen.

Es konnte keine Literatur gefunden werden, in denen Agenten, die durch Self-Play lernen, vollständig ohne Vorwissen über das Spiel auf Hex angewandt wurden. Im Speziellen scheint diese Arbeit die erste zu sein, in der ein N-Tupel-System auf Hex angewandt wurde, um die Spielfunktion zu erlernen.

3. GBG-Framework

3.1 Beschreibung

Das „General Board Game Playing and Learning Framework“ wurde von Wolfgang Konen entwickelt. Das Framework wurde in Java geschrieben und thematisiert die Anwendung von KI-Verfahren auf Brettspiele. Sowohl für die Spiele, als auch für die Agenten, bietet das Framework eine Reihe von Interfaces, die implementiert werden können. Implementiert ein Spiel diese Schnittstellen, können die im GBG-Framework enthaltenen KI-Verfahren auf das Spiel angewendet werden, ohne dass die Agenten angepasst werden müssen. Ebenso können neu hinzugefügte Agenten direkt mit allen bestehenden Spielen getestet werden. Die einzigen Informationen, die die Agenten dafür unbedingt vom Spiel benötigen, sind die für den Agenten möglichen Aktionen zu einem bestimmten Zustand und im Falle eines Endzustands eine Information darüber, wer das Spiel gewonnen hat. Optional können für einzelne Agenten weitere Informationen vom Spiel durch Implementierung der zugehörigen Interfaces bereitgestellt werden. Eine detaillierte Beschreibung des GBG-Frameworks und der Interfaces sind im technischen Report von Wolfgang Konen [1] zu finden.

Ziel jedes Agenten ist es, eine Spielfunktion $V(s_t)$ zu erlernen, die den Zustand s zum Zeitpunkt t bewertet. Die Spielfunktion hat dabei in der Regel einen reellen Wert im Intervall $[-1, +1]$, wobei aus der Sicht des bewertenden Agenten -1 eine sichere Niederlage bedeutet und $+1$ ein sicherer Sieg. Endzustände haben in Hex zwangsläufig eine Bewertung von exakt -1 oder $+1$. Jedes Mal, wenn der Agent eine Entscheidung treffen muss, wird die Aktion gewählt, die die Bewertung des Folgezustands durch die Spielfunktion maximiert. Die Spielfunktion ist dann optimal, wenn jeder Zustand korrekt bewertet wird. In der Praxis kann es je nach Komplexität des Spiels sehr schwierig bis gar unmöglich werden, eine vollständig korrekte Spielfunktion zu finden. Ziel ist deshalb eine bestmögliche Annäherung an die optimale Spielfunktion.

Zum Zeitpunkt des Schreibens sind folgende KI-Verfahren im GBG-Framework verfügbar:

- Temporal Difference
- Temporal Difference (N-Tupel)
- Monte Carlo (ohne Suchbaum)
- Monte Carlo Tree Search
- Monte Carlo Tree Search Expectimax
- Minimax
- Random
- Human

„Human“ hat dabei eine Sonderstellung, da dieser Agent auf Benutzereingaben über die grafische Benutzerschnittstelle (GUI) des Spiels wartet. So können die implementierten Spiele auch von Menschen gespielt werden.

Die Agenten „Monte Carlo“ (ohne Suchbaum) und „Monte Carlo Tree Search Expectimax“ sind Varianten des „Monte Carlo Tree Search”-Agenten für Spiele mit stochastischen Elementen. Da sie für Hex nicht von Bedeutung sind werden sie in dieser Arbeit nicht genauer betrachtet.

Der „Random“-Agent ist nur zu Testzwecken gedacht und erlernt nicht die Spielfunktion, sondern weist jedem Zustand eine vollkommen zufällige Bewertung zu. Er wird deshalb nicht weiter beschrieben, jedoch wurde er eingesetzt, um die Implementierung des Spiels zu überprüfen.

In den folgenden Abschnitten dieses Kapitels werden die verbleibenden Agenten genauer erläutert.

3.2 Minimax

Für deterministische Spiele mit perfekter Information, von denen Hex eins ist, ist es möglich, die Auswirkungen jeder möglichen Zugfolge zu berechnen um den bestmöglichen nächsten Zug zu bestimmen. Das ist genau die Vorgehensweise des Minimax-Agenten. Er geht dabei von perfekter Spielweise seines Gegners aus und versucht, den Erfolg seines Gegenspielers zu minimieren, während er gleichzeitig seinen eigenen Erfolg maximiert [15].

Der Minimax-Agent erstellt dafür einen Suchbaum, dessen Wurzel der aktuelle Spielzustand ist. Ausgehend von der Wurzel wird mit dem Wissen über die möglichen gültigen Züge jeder Zug simuliert und der daraus resultierende Zustand als Kind-Knoten im Baum gespeichert. Wenn alle Folgezüge ausgehend von der Wurzel durchgespielt wurden, wird der Prozess ausgehend von den Kind-Knoten erneut durchgeführt. Dieser Prozess wird entweder so lange wiederholt, bis der Baum eine festgelegte Maximalhöhe erreicht hat, oder bis für jede Zugfolge ein Endzustand erreicht wurde. Wenn der Baum vollständig aufgebaut ist, werden für alle Blätter die jeweilige Belohnung oder Bestrafung berechnet, je nachdem ob der Agent in diesem Endzustand gewinnen oder verlieren würde. Für Zustände, bei denen der Agent am Zug ist, wird der maximale Wert aller Kind-Knoten gewählt (Maximierung). Zustände, bei denen der Gegenspieler am Zug ist, nehmen das Minimum aller Kind-Knoten an (Minimierung) [16]. Eine Veranschaulichung davon ist in Abbildung 5 zu sehen.

Gibt es mehrere mögliche Züge, die eine gleich gute Bewertung besitzen, wird in der Implementierung des Agenten im GBG-Framework ein zufälliger Zug mit dieser Bewertung gewählt. Dies hat zwei Folgen:

1) Der Minimax-Agent legt keinen Wert darauf, ein Spiel schnell zu beenden. Ist das Spiel schon weit fortgeschritten und es gibt mehrere sichere Wege zum Ziel, wählt der Agent unter Umständen einen viel längeren Weg als nötig.
2) Existiert für den Gegenspieler eine optimale Strategie, bewertet der Minimax-Agent alle seiner Möglichkeiten mit einer sicheren Niederlage und wählt demnach ein zufälliges Feld.

Insbesondere der zweite Punkt muss beim Einsatz des Agenten als Evaluator bedacht werden (siehe Kapitel 4).

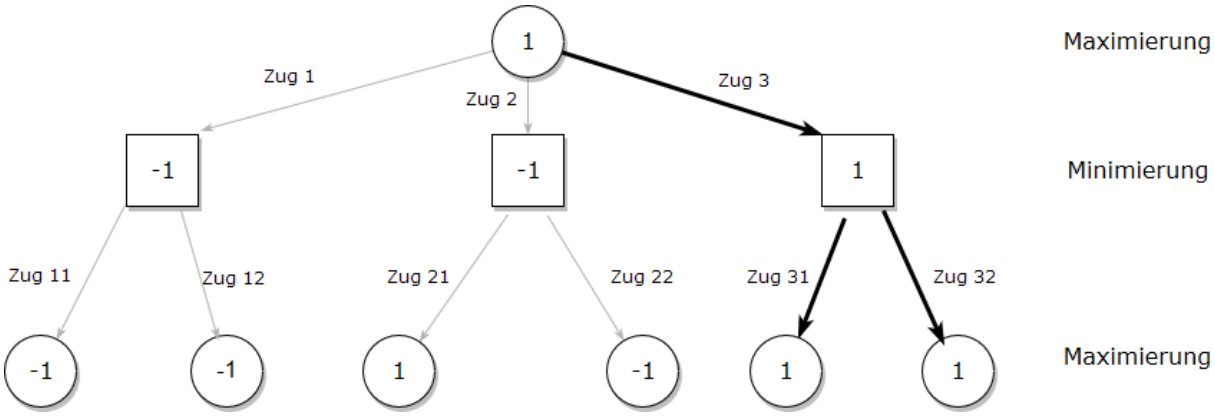

Abbildung 5: Veranschaulichung eines Suchbaums des Minimax-Agenten
Belohnungen haben den Wert 1, Bestrafungen den Wert -1. Während den Zügen des Agenten (Kreise) wird das Maximum der Kind-Knoten gewählt, während dem Zug des Gegenspielers das Minimum (Quadrate). Gespielt wird Zug 3, da mit diesem Zug selbst bei perfektem Spiel des Gegenspielers sicher gewonnen wird.

Der Minimax-Agent findet nach Definition eine optimale Strategie, wenn der Suchbaum nicht begrenzt wird und wenn eine solche Strategie existiert. Warum wird er also nicht als ultimative Lösung für alle Spiele verwendet? In den meisten Spielen würde der Suchbaum unfassbar groß werden, wenn wirklich jeder Zustand besucht werden soll. Das beschränkt den Einsatz des (unbegrenzten) Minimax-Agenten also auf weniger komplexe Spiele.

Begrenzt man die Höhe des Suchbaums, kann der Agent auch in Situationen eingesetzt werden, die sonst einen zu großen Suchbaum besitzen würden. Dadurch wird jedoch die Sichtweite des Agenten begrenzt, was zwangsläufig seine Leistung mindert. In der Praxis wird häufig ein „*alpha-beta-cutoff*“ eingesetzt, mit dem aus Sicht des Minimax-Agenten offensichtlich schlechte Teilbäume nicht durchsucht werden. Nach Borovska und Lazarova kann die Implementierung von *alpha-beta-cutoffs* einen deutlichen Geschwindigkeitszuwachs bedeuten und weiterhin die Größe des Baums reduzieren [15].

3.3 Monte-Carlo Tree Search

Die *Monte-Carlo Tree Search* (MCTS) setzt auf stochastische Verfahren um im Kontext von Spielen den optimalen Zug zu finden. Wie beim Minimax-Algorithmus wird ein Suchbaum von Spielzuständen erstellt, die nach gewissen Kriterien bewertet werden. Anders als bei Minimax wird jedoch nicht versucht, jeden Zustand zu besuchen, weshalb MCTS in vielen Situationen effizienter ist.

Nach Chaslot et al. [17] besteht der Algorithmus aus vier Schritten:

1) **Auswahl**: Die sogenannte *tree policy* wählt einen Knoten aus, der erweitert werden soll. Dabei wird versucht, eine Balance zwischen dem Erkunden neuer Zustände (*Exploration*) und tieferem Erforschen von vielversprechenden Zuständen (*Exploitation*) zu finden.
2) **Erweiterung**: Jedes Mal, wenn ein neuer Zustand entdeckt wird, wird dieser zum Suchbaum hinzugefügt.
3) **Simulation**: Von dem Zustand, den die *tree policy* ausgewählt hat, wird nun unter Verwendung der *default policy* eine Simulation (auch manchmal Rollout genannt) des Spiels durchgeführt, bis ein Endzustand erreicht wurde.
4) **Rückpropagierung**: Ausgehend vom Endzustand der Simulation wird die erreichte Belohnung/Bestrafung zu jedem besuchten Knoten hinzuaddiert. Außerdem wird für jeden Knoten ein Zähler erhöht, der angibt, wie oft der Knoten schon besucht wurde.

Der am Ende gewählte Zug ist in der Regel derjenige, der zum Zustand mit der höchsten durchschnittlichen Bewertung führt.

Die Auswahl der richtigen *policies* für Schritte 1) und 3) sind dabei von hoher Bedeutung. Beide haben einen großen Einfluss darauf, wie viel Nutzen aus jeder Simulation gewonnen werden kann. Die im GBG-Framework genutzt *tree policy* nennt sich *Upper Confidence Bound for Trees*, oder kurz UCT. Demnach wird der zu expandierende Knoten ausgewählt, indem der Knoten mit dem höchsten UCT-Wert berechnet wird:

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

$\overline{X}_c$ ist dabei die durchschnittliche Belohnung des Kind-Knoten c. Weiterhin ist n der Zähler, wie oft der aktuelle (Vater-) Knoten besucht wurde. Entsprechend ist n_c die Höhe des Zählers, wie oft der Kind-Knoten bereits besucht wurde. Die Konstante K ist ein Parameter, mit dem sich die Explorationsrate des MCTS-Agenten anpassen lässt. r ist ein sehr kleiner Zufallswert, der nur verwendet wird, um einen möglichen Gleichstand zwischen zwei oder mehr Knoten mit gleichem UCT-Wert aufzulösen.

Die im GBG-Framework verwendete *default policy* ist recht simpel, denn es wird aus der Menge aller möglichen Züge einfach nur ein zufälliger Zug ausgewählt. Das ist verständlich, denn ein Ziel des GBG-Frameworks ist es, dass bestehende Agenten ohne großen Aufwand für neue Spiele eingesetzt werden können. Eine gute *default policy* benötigt Vorwissen über das Spiel, um effizientere Entscheidungen zu treffen. Diese würde, wenn sie richtig implementiert wurde, dafür sorgen, dass die Leistung des Agenten schneller mit den durchgeführten Iterationen ansteigt. Im GBG-Framework gibt es derzeit keine Möglichkeit, eine spezielle *default policy* für ein Spiel zu implementieren.

Wenn dem MCTS-Agenten genügend Rechenzeit und Speicher zur Verfügung stehen, kann seine Leistung durchaus mit der des Minimax-Agenten vergleichbar sein. Nach Kocsis und Szepesvári [18] konvergiert die Fehlerwahrscheinlichkeit des Wurzel-Knotens mit steigender Anzahl an Iterationen gegen 0. Ein Fehler ist in diesem Kontext, wenn ein nicht perfekter Zug

gewählt wird. In der Praxis kann es jedoch schwierig sein, zu bestimmen, wann genügend Iterationen für eine perfekte Entscheidung durchgeführt wurden.

Zur Anpassung des MCTS-Agenten an das Spiel gibt es im GBG-Framework eine Reihe von Parametern, wie in Abbildung 6 zu sehen ist:

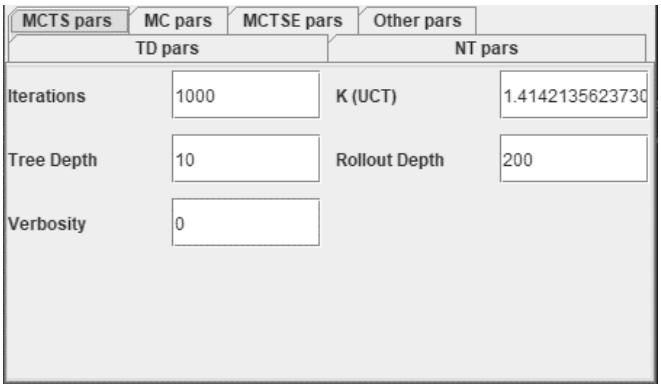

Abbildung 6: Das Einstellungsfenster für den MCTS-Agenten im GBG-Framework

Iterations: Die Anzahl der Simulationen, die pro anfallender Entscheidung durchgeführt werden. In anderen Implementierungen kommt es auch vor, dass stattdessen die Zeit in Sekunden angegeben wird, die dem Algorithmus für seine Simulationen zur Verfügung steht. Die tatsächliche Anzahl der Iterationen anzugeben hat den Vorteil, dass die Spielleistung des Agenten nicht von der Rechenleistung des verwendeten Computers abhängt.

K (UCT): Wie bereits erwähnt wurde, ist K ein Faktor, der die Explorationsrate beeinflusst. Je größer diese Konstante ist, desto eher wird versucht, unbesuchte Knoten zu erforschen. Standardmäßig wird im GBG-Framework $K = \sqrt{2}$ verwendet, jedoch muss dies nicht für alle Spiele der optimale Wert sein.

Tree Depth: Die Tree Depth ist die Maximalhöhe des erstellten Suchbaums. Wird diese Höhe erreicht, werden keine weiteren Kind-Knoten zum Baum hinzugefügt, wenn diese die Grenze überschreiten würden. Anders als beim Minimax-Agenten können jedoch auch für die Knoten an der Höhenbegrenzung Bewertungen abgegeben werden, selbst wenn diese Knoten keine Endzustände darstellen. Der Grund dafür ist, dass auch bei diesen Randknoten eine vollständige Simulation des Spiels bis zu einem Endzustand stattfindet. Dabei besuchte Knoten werden nur nicht zum Baum hinzugefügt.

Rollout Depth: Dieser Parameter begrenzt die maximale Anzahl an Zügen pro Simulation. In manchen Spielen kann es vorkommen, dass es eine sehr große Anzahl an Zügen geben kann, bevor das Spiel ein Ende findet. Auch wäre vorstellbar, dass es bei bestimmten Spielbrettzuständen gar kein Ende gibt. Begrenzt man die Länge der Simulation, verhindert man in solchen Fällen eine Endlosschleife im Algorithmus. Für Spiele, die sicher einen Endzustand erreichen, sollte die Rollout Depth nicht begrenzt werden.

3.4 Temporal Difference

Temporal Difference Learning (TDL) ist eine Methode, die für maschinelles Lernen, im speziellen für *„Reinforcement Learning“* (bestärkendes Lernen) eingesetzt wird. Erstmalig fand TDL Verwendung im Jahr 1959, als A. L. Samuel [19] die Methode auf das Spiel Dame angewandt hat. Berühmt wurde TDL jedoch erst durch Tesauros TD-Backgammon [20], das durch TDL gelernt hat, auf Expertenebene zu spielen.

Das Problem, dass *Temporal Difference Learning* zu lösen versucht, ist, dass in manchen Umgebungen nur verzögert eine Belohnung/Bestrafung erhalten wird. Dadurch kann erst dann über die Leistung eines Agenten geurteilt werden, wenn ein Endzustand erreicht wurde. Die Information, die für die Bewertung eines Zustands notwendig ist, liegt also in der Zukunft.

3.4.1 Algorithmus

Anders als die bisher vorgestellten Agenten, muss der *Temporal-Difference*-Agent (im Folgenden TD-Agent genannt) vor seiner Verwendung trainiert werden. Im klassischen TDL wird eine *Lookup Table* (LUT) verwendet, um eine Belohnung, die erst in der Zukunft erhalten wird, den getätigten Aktionen und den daraus resultierenden Zuständen zuzuweisen. Für viele Spiele wäre die Verwendung einer LUT nicht sinnvoll. Es kann viel zu viele Zustände geben, um alle zu besuchen [21]. Deswegen wird TDL oft unter Verwendung eines neuronalen Netzes implementiert, mit dem auch Abschätzungen für bisher unbesuchte Zustände abgegeben werden können.

Dabei wird eine numerische Repräsentation des Spielbrettzustands verwendet, die *„Feature Vector“* bezeichnet wird. Dieser Vektor besteht aus *Features* (zu Deutsch: Merkmalen), die in gewisser Relation zum Zustand des Spielbrettes stehen. Ziel des TDL ist es, eine Funktion zu finden, die für jeden Feature-Vektor, der als Eingabedaten verwendet wird, die korrekte Ausgabe erzeugt. Im Fall eines Brettspiels wäre die korrekte Ausgabe zum Beispiel die Wahrscheinlichkeit, dass ein bestimmter Zug zum Sieg des Spiels führt. In der Praxis kann eine exakt richtige Bewertung jedes Spielzustands für die meisten Brettspiele nicht erlernt werden. Wichtig ist deshalb eine Minimierung der durchschnittlichen Abweichung von dieser korrekten Bewertung.

Nach Konen und Bartz-Beielstein verwendet die zu erlernende Funktion $f(w; g(s))$ sowohl einen Gewichtungs-Vektor w als auch den Feature-Vektor $g(s)$ als Parameter [21]. Bei der Initialisierung des Agenten werden zu Beginn vollkommen zufällige Gewichtungen vergeben. Durch TDL werden iterativ die Gewichtungen w angepasst, sodass sich die Ausgabe der Funktion immer weiter der korrekten Ausgabe annähert. Nach einem erfolgreichen Training liefert das Skalarprodukt des Eingabevektors und dem Gewichtungs-Vektor

$$f\big(w; g(s)\big) = w \cdot g(s)$$

(vgl. [21]) eine Ausgabe, die möglichst nah an der korrekten Bewertung liegt. Oft wird auf das Ergebnis noch eine Sigmoid-Funktion angewendet, die die Ausgabewerte auf einen

gewissen Wertebereich beschränkt. Im GBG-Framework kommt dafür die Sigmoid-Funktion $\sigma = tanh$ zum Einsatz.

Diese Art von Funktion, die genau einen Gewichtungs-Vektor verwendet, wird als ein lineares Netz bezeichnet. Um die Gewichtungen des linearen Netzes anzupassen, muss eine Funktion existieren, die die berechnete Ausgabe des Netzes mit dem tatsächlich korrekten Wert vergleicht und deren Abweichung berechnet. Diese sogenannte Fehlerfunktion kennt jedoch in den meisten Brettspielen nur dann den tatsächlichen Wert eines Spielzustands, wenn dieser ein Endzustand ist. Für alle weiteren Aktualisierungen der Gewichtungen während des Spiels berechnet die Fehlerfunktion die Differenz des momentanen geschätzten Ausgabewerts zu der vorherigen Schätzung. Von dem Ergebnis der Fehlerfunktion wird die partielle Ableitung gebildet um einen Gradienten zu berechnen. Mit diesem kann unter Verwendung des Gradientenverfahrens (engl. *gradient descent*) der Ausgabewert der Fehlerfunktion nach und nach minimiert werden, indem der Gradient zur Gewichtung des aktuellen Zustands hinzuaddiert wird. Dadurch wird erreicht, dass die Genauigkeit der Vorhersagen des Netzes mit jedem Lernschritt ansteigt.

Damit ein lineares Netz erfolgreich lernen kann, muss eine direkte, lineare Beziehung zwischen den Eingabedaten und der korrekten Ausgabe bestehen. Besteht jedoch eine komplexere Beziehung zwischen den einzelnen Features des Eingabevektors und der korrekten Ausgabe, müssen eine oder mehrere sogenannter „*Hidden Layers*“ (versteckte Schichten) im Netz eingesetzt werden. Jeder dieser *Hidden Layer* besitzt einen eigenen Gewichtungs-Vektor und verwendet als Eingabewerte die Ausgabe des vorherigen Layer. So kann das Netz selbstständig eine vollkommen andere Repräsentation der Eingabedaten entwickeln, die dann nach der Multiplikation mit dem Gewichtungs-Vektor der letzten Hidden Layer eine direkte Beziehung zum gewünschten Ausgabewert hat. Für die Ausgabe des Netzes wird dann mit der Fehlerfunktion die Abweichung vom korrekten Ergebnis berechnet. Anschließend wird der berechnete Fehler von hinten nach vorne auf alle Layer angewendet um die Gewichtungen zu aktualisieren, je nachdem, welchen Einfluss sie auf das Endergebnis hatten. Dieser Prozess wird „*Backpropagation*“, oder *Fehlerrückführung* genannt. Das GBG-Framework bietet sowohl die Möglichkeit, ein lineares Netz, als auch ein Netz mit einer Hidden Layer zu verwenden.

Das GBG-Framework implementiert außerdem eine Erweiterung des TD-Algorithmus mit der Bezeichnung TD(λ). Diese Erweiterung fügt sogenannte „*eligibility traces*“ hinzu. Für jeden Zustand wird zusätzlich ein Wert zwischen [0, 1] gespeichert, der mit 1 initialisiert wird und mit der Zeit abhängig von Parameter λ abnimmt. Findet eine Aktualisierung der Gewichtungen statt, wird diese Aktualisierung auf alle Zustände rückwirkend angewandt und nicht nur auf den vorherigen. Wie stark die Anpassung älterer Zustände ist, hängt von der Höhe dieses *eligibility-trace*-Werts ab. Je älter der Zustand, desto geringer der Wert. Ziel dieser Erweiterung ist, dass weniger Trainingsschritte notwendig sind, um eine gute Leistung zu erzielen.

3.4.2 Features

Die Auswahl effektiver Features ist enorm wichtig für einen guten Lernerfolg des TD-Agenten. Ein Feature kann im Zusammenhang mit Brettspielen alles sein, was numerisch einen Teil des Spielzustands abbildet. Für das finden passender Features ist Kreativität nötig. Es wäre möglich, einfach jedem Feld des Spielbretts einen Wert zuzuweisen, abhängig davon, welcher Spieler das Feld belegt. Damit würde das Training jedoch sehr lange dauern, wenn es überhaupt funktioniert. Effektiver ist es, die Daten ein wenig aufzubereiten. So könnte zum Beispiel die Anzahl der Spielsteine jedes Spielers gezählt und als ein Teil des Feature-Vektors verwendet werden. Mehr Features zu haben ist in der Regel besser als Wenige [22], da dem Agenten so mehr Informationen über den Spielzustand zur Verfügung stehen. Wenn unwichtige Features im Vektor vorkommen, sollte das Netz durch eine entsprechende Anpassung der Gewichtungsmatrix dafür sorgen, dass diese nicht stark in das Ergebnis einfließen. Insgesamt sollte der Feature-Vektor jedoch einen guten Hinweis auf den Wert des Spielzustands geben. Wenn der exakt gleiche Eingabe-Vektor für mehrere verschiedene Spielzustände mit stark unterschiedlicher Bewertung vorkommt, ist ein effektives Lernen unter Umständen nicht möglich. Je mehr Features verwendet werden, desto geringer ist die Wahrscheinlichkeit, dass dieser Fall eintritt.

3.4.3 Parameter

Um den TD-Algorithmus an ein Spiel anpassen zu können, gibt es in der Implementierung des Agenten im GBG-Framework eine Reihe von Parametern (siehe untenstehende Abbildung 7).

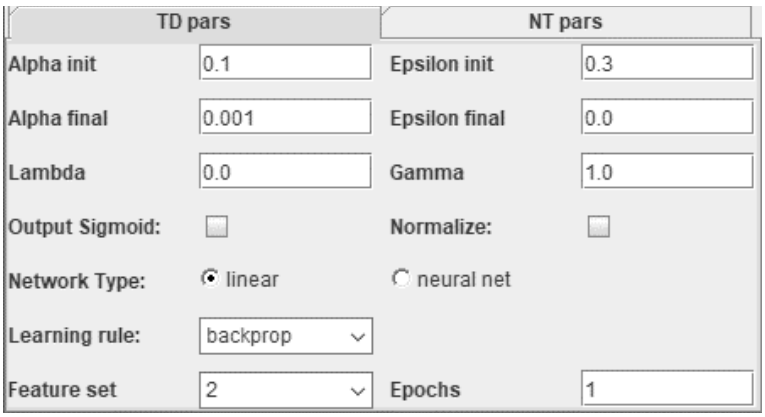

Abbildung 7: Einstellungsfenster für den TD-Agenten im GBG-Framework

***Alpha* (α)**: Dieser Wert entspricht der Lernrate des Agenten. Er wird mit dem Gradienten der Fehlerfunktion multipliziert und beeinflusst die Schrittweite, mit der sich dem Minimum der Fehlerfunktion angenähert wird. Wird ein zu hoher Wert für α gewählt, kann das Minimum verfehlt werden, wodurch der Fehler wieder ansteigt. Wird ein zu kleiner Wert gewählt, kann das Training unnötig lange dauern. Das GBG-Framework bietet die Möglichkeit, einen unterschiedlichen Start- und Endwert für α zu

verwenden, sodass die Lernrate im Laufe des Trainings sinkt um eine erfolgreiche Konvergenz auf das Minimum wahrscheinlicher zu machen.

***Epsilon* (ε)**: Mit diesem Parameter lässt sich die Wahrscheinlichkeit für Zufallszüge (Explorationsrate) bestimmen, mit $0 \leq \varepsilon \leq 1$. Da der Agent nach Definition rein deterministisch handelt, kann es passieren, dass während des Trainings immer wieder die gleichen Züge durchgeführt werden. Dadurch werden andere Zustände, die möglicherweise erfolgsversprechender sein können, nie besucht. Wird der Agent mit einer gewissen Wahrscheinlichkeit zu einem Zufallszug gezwungen, werden eine größere Breite an Zuständen besucht, was eine bessere Spielleistung zur Folge haben kann. Wird ein zu hoher Wert gewählt, wird Trainingszeit verschwendet, indem sehr unvorteilhafte Zustände erforscht werden. Ebenso wie α kann auch ε mit einem Start- und Endwert versehen werden, sodass die Explorationsrate im Laufe des Trainings abnehmen kann.

***Lambda* (λ)**: Der Parameter λ, mit $0 \leq \lambda \leq 1$, entspricht der Zerfallsrate der *eligibility traces.* Ein Wert von 0 entspricht sofortigem Zerfall, wodurch die TD(λ)-Erweiterung deaktiviert wird. Ein Wert von 1 ähnelt dem Verhalten des MCTS-Agenten, da die Belohnung im Endzustand vollständig auf alle Zwischenzustände angewandt wird.

***Gamma* (γ)**: Dies ist der sogenannte Discount-Faktor, mit $0 \leq \gamma \leq 1$. Bei der Bewertung eines Zustands durch den Agenten werden in gewissem Grad auch mögliche zukünftige Zustände betrachtet. Hat γ den Wert 0, haben zukünftige Bewertungen keinen Einfluss auf den aktuellen Zustand. Läuft γ gegen 1, wird der Einfluss der möglichen zukünftigen Zustände größer. Bei deterministischen Spielen macht es Sinn, einen Wert von 1 zu wählen, da die Folgezustände berechenbar sind. Nicht deterministische Spiele haben das Problem, dass die Zukunft ungewiss sein kann, weshalb der Einfluss der zukünftigen Zustände begrenzt werden sollte.

Output Sigmoid: Mit diesem Parameter lässt sich die Verwendung der Sigmoid-Funktion ein- oder ausschalten.

Normalize: Dieser Parameter, der entweder an oder aus sein kann, beschränkt die Eingabewerte für die Sigmoid-Funktion auf den Intervall $[-1, +1]$. Dies ist sinnvoll für Spiele, die nicht nur Gewinn oder Verlust als Ergebnis haben, sondern auch einen Punktestand besitzen.

Learning Rule: Für die Methode der Fehlerrückführung kann zwischen „backpropagation“ oder der Alternative „RPROP“ („resilient backpropagation“) gewählt werden.

3.5 Temporal Difference (N-Tupel)

Während sowohl Temporal Difference Learning als auch N-Tupel-System schon seit einiger Zeit im Einsatz waren, verwendete Simon M. Lucas [23] im Jahr 2008 erstmals N-Tupel-Systeme in Verbindung mit TDL im Kontext von Spielen. Er zeigte, dass das Spiel Othello mit wenigen tausenden Spielen von einem Computer erlernt werden konnte. Die N-Tupel

werden dafür als Features für den TD-Algorithmus verwendet, sodass kein spielspezifisches Wissen mehr integriert werden muss. Im Folgenden wird der Agent, der ein N-Tupel-System verwendet, TDNT-Agent genannt.

3.5.1 Algorithmus

Jeder N-Tupel ist eine Reihe von genau n Feldern des Spielbretts, die im Kontext des N-Tupel-Systems auch als Abtastpunkte betrachtet werden. Jeder Abtastpunkt kann nur einmal in einem Tupel enthalten sein, jedoch mehrmals im gesamten System mit mehreren verschiedenen N-Tupel. Der Maximalwert für n, also die Maximallänge der Tupel, entspricht damit der Anzahl der Felder auf dem Spielbrett. Tupel, die alle Abtastpunkte enthalten, sind jedoch in den meisten Fällen nicht sehr effektiv.

Nach Lucas [23] werden die Tupel in Form einer *Lookup-Table* (LUT) gespeichert, die jede mögliche Kombination aller Zustände der Abtastpunkte enthält. Umgesetzt wird das durch ein Array mit der Größe s^n, wenn s die Anzahl der möglichen Zustände pro Abtastpunkt ist. Berechnet wird jeweils der Index der Zustandskombination, die in dem N-Tupel zum jeweiligen Zeitpunkt vorzufinden ist. In welcher Reihenfolge die Abtastfelder gewählt werden hat keine Auswirkung auf den Lernerfolg, solange die Reihenfolge während und nach dem Training konstant bleibt.

Zum Zeitpunkt des Schreibens gibt es im GBG-Framework drei Wege, die N-Tupel auszuwählen.

Die erste Möglichkeit ist, die Tupel manuell festzulegen. Hierbei kann der Entwickler, der das Spiel implementiert, Tupel festlegen, die für das Spiel gut funktionieren. Dadurch kann es dem Agenten einfacher gemacht werden, wichtige Muster zu erkennen. Weiterhin erhöht dies die Reproduzierbarkeit der Trainingsergebnisse gegenüber den anderen Methoden zur Tupel-Wahl, die nicht deterministisch arbeiten.

Eine zweite Möglichkeit ist ein Algorithmus zur automatischen Auswahl der Tupel mit der Bezeichnung „Random Walk“. Dieser Algorithmus wählt ein zufälliges Feld auf dem Spielbrett als ersten Abtastpunkt und „läuft“ von dort aus zu benachbarten, noch nicht besuchten Feldern, die dann auch als Abtastpunkte in den Tupel aufgenommen werden. Der Random Walk stoppt, sobald genau n Abtastpunkte gefunden wurden. Dafür benötigt der Algorithmus Wissen über die Nachbarschaftsbeziehungen jedes Feldes, wofür der Entwickler des Spiels eine Interfacefunktion im GBG-Framework implementieren muss, die für jedes Feld alle Nachbarfelder zurückliefert. Dieser Algorithmus wurde auch von Lucas [23] erfolgreich eingesetzt, um das Spiel Othello zu lernen.

Die dritte Möglichkeit ist ein Algorithmus mit der Bezeichnung „Random Point“. Dieser Algorithmus wählt seine Abtastpunkte vollkommen zufällig auf dem Spielbrett. Sie hängen also nicht wie beim „Random Walk“-Algorithmus in einer Kette zusammen. Dadurch braucht der Algorithmus kein Wissen über die Nachbarschaftsbeziehungen, ist jedoch in den meisten Fällen nicht in der Lage, lokale Muster zu erkennen.

Der TDNT-Agent kann bei der Auswahl der N-Tupel die Symmetrien des Spielbretts nutzen, wenn solche existieren und diese Information von der Implementierung des Spiels zur Verfügung gestellt wird. Die dafür zu implementierende Funktion liefert zu einem beliebigen Zustand alle weiteren Zustände zurück, die nach den Symmetrien äquivalent zum Eingabezustand sind. Diese Information wird vom Agenten bei der Wahl der Abtastpunkte verwendet, sodass weniger Tupel für die gleiche Leistung benötigt werden.

3.5.2 Parameter

Da der TDNT-Agent auch TDL verwendet, sind die Parameter des TD-Agenten ebenfalls für den TDNT-Agenten relevant. Die Unterschiede zum TD-Agenten sind, dass die Sigmoid-Funktion nicht deaktiviert werden kann, dass immer ein lineares Netz verwendet wird und dass als *Learning Rule* immer *backpropagation* verwendet wird.

Neu hinzu kommen im Vergleich zum TD-Agenten folgende Parameter:

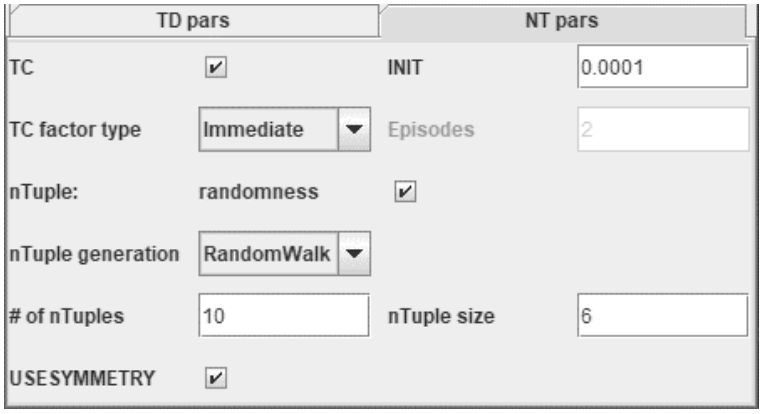

Abbildung 8: Einstellungsfenster des TDNT-Agenten im GBG-Framework

TC (Temporal Coherence): Wenn dieser Parameter aktiviert wird, wird die Methode *Temporal Coherence* verwendet um die Lernrate α automatisch anzupassen. Bagheri und Thill [24] untersuchten diese Methode zur Beschleunigung des Lernvorgangs am Spiel *Vier Gewinnt*, hatten jedoch nur mäßigen Erfolg damit. Nach ihren Untersuchungen hatte die Verwendung von *eligibility traces* eine größere Auswirkung auf die Lerngeschwindigkeit.

Randomness: Mit diesem Parameter lässt sich zwischen vordefinierten und zufallsgenerierten N-Tupel wechseln. Wenn Randomness aktiv ist, kann zwischen *Random Walk* und *Random Point* gewählt werden. Weiterhin besteht dann die Möglichkeit, die Anzahl und die Länge der zu verwendenden N-Tupel festzulegen.

Use Symmetry: Hiermit kann die Verwendung von Spielbrettsymmetrien an- oder ausgeschaltet werden. Wenn es keine Symmetrien zu dem gewählten Spiel gibt oder das Spiel die zugehörige Funktion nicht implementiert, kann diese Optimierung deaktiviert werden.

3.6 Implementierung von Hex

Hex wurde nah an den Interfaces des GBG-Frameworks implementiert. Die drei wichtigsten Klassen, die dafür geschrieben wurden, sind `GameBoardHex`, `StateOberserverHex` und `HexUtils`.

`StateOberserverHex` implementiert das GBG-Interface `StateObservation` und verwaltet damit den jeweiligen Spielzustand. Die Klasse besitzt das Array, das die Spielfelder enthält, bestimmt den Spieler der aktuell am Zug ist und implementiert die Spielelogik.

`HexUtils` ist eine statische Hilfsklasse, die Funktionalität implementiert die nicht unbedingt an ein Objekt gebunden sein muss. Viele der notwendigen Berechnungen sind in dieser Klasse untergebracht und können von allen anderen Klassen aufgerufen werden, ohne dass ein Objekt instanziiert werden muss.

`GameBoardHex` beinhaltet den Code, der für die Darstellung des Spielbretts auf dem Bildschirm verantwortlich ist. Das GUI von Hex besteht aus einem `JPanel`, auf dem keinerlei GUI-Elemente von Java (Buttons o.Ä.) vorhanden sind. Die Felder werden stattdessen direkt auf das `JPanel` gezeichnet. Das Spielbrett ist vollständig in seiner Größe skalierbar, sowohl in der Anzahl als auch in der Größe der einzelnen Felder. Jedes Feld, implementiert durch die Klasse `HexTile`, besitzt einen von drei möglichen Zuständen (unbelegt oder die Identifikationsnummer des jeweiligen Spielers), die Position auf dem Spielbrett, ein `Polygon`-Objekt und eine Feldbewertung. Die Koordinate entspricht der Position des Felds im Array des Spielbretts, zum Beispiel (1, 1) im Falle des mittleren Felds auf einem 3×3-Spielbrett. Das `Polygon`-Objekt speichert die Koordinaten der einzelnen Eckpunkte des Hexagons innerhalb des Fensters in Pixeln. Die Feldbewertung wird von dem jeweiligen Agenten gesetzt, der gerade am Zug ist, und entspricht der Bewertung des Folgezustands, der eintreten würde, wenn der Agent einen Stein auf das jeweilige Feld setzen würde.

Da die Feldbewertung für die manuelle Evaluation der Agenten recht nützlich ist, wurde ein wenig Zeit dafür verwendet, die Feldbewertungen auf dem Spielbrett farbig zu visualisieren. Da die Bewertung jedes Agenten in einem bekannten Bereich von [-1, +1] liegt, kann die Bewertung durch die Berechnung passender Farbwerte auf einen Blick schnell vermittelt werden. Der berechnete Farbton verläuft von Rot bei einer Bewertung von -1 über Gelb bei einer Bewertung von 0 bis zu Grün bei einer Bewertung von +1. Da die meisten Bewertungen mit „0,[…]“ oder „-0,[…]“ beginnen und der interessante Teil erst hinter dem Komma steht, wurden die Bewertungen der Felder mit 1.000 multipliziert und danach gerundet. Somit befinden sich die Beschriftungen im Intervall [-1.000, +1.000] und bestehen nur aus ganzen Zahlen, was dem Benutzer die gleiche Information bietet, aber etwas Platz spart.

Weiterhin wird der zuletzt getätigte Zug mit einer roten Umrandung hervorgehoben, was besonders hilft, wenn ein Mensch gegen einen Agenten spielt und sich nicht sicher ist, welches Feld vom Agenten im letzten Zug gewählt wurde. Außerdem ist das zuletzt vom Agenten gewählte Feld auch mit der Bewertung beschriftet, die der Agent dem Feld zugeschrieben hat, bevor er seinen Zug getätigt hat. Alle bereits belegten Felder, die nicht

im letzten Zug gewählt wurden, haben keine Beschriftung. Da es für den Agenten kein gültiger Zug wäre, ein bereits belegtes Feld zu wählen, wird der Zustand der darauf folgt auch nicht bewertet.

Durch diese Farbkodierung wird die Bewertung der Agentenstärke erheblich erleichtert. Besonders auf größeren Spielbrettern wäre es sonst mühsam, alle Felder zu vergleichen, wenn die Felder nur beschriftet wären. Ein Beispiel für die Farbkodierung während eines Spiels ist in Abbildung 9 zu sehen.

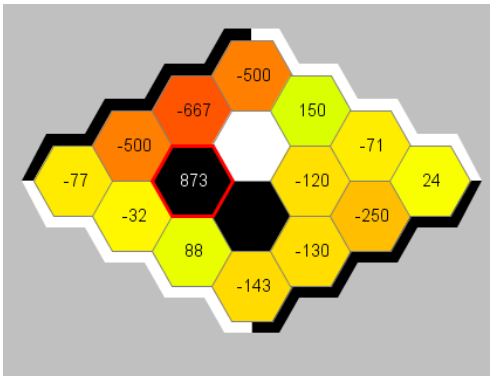

Abbildung 9: Farbkodierung der Feldbewertungen eines laufenden Spiels.

Das GBG-Framework bietet auch die Möglichkeit, das leere Spielbrett durch einen Agenten bewerten zu lassen, um die Bewertungen der Eröffnungszüge anzuzeigen. Die Bewertung der Eröffnungszüge eines 4×4-Spielbrett durch einen MCTS-Agenten ist in Abbildung 10 zu sehen.

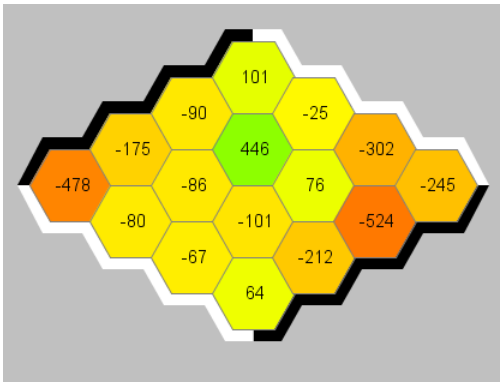

Abbildung 10: Farbkodierung der Bewertung aller möglichen Eröffnungszüge durch einen MCTS-Agenten

4. Evaluation

Während die eingesetzten Agenten in der Lage sind, beliebige Spielzustände zu bewerten, muss auch untersucht werden, ob diese Bewertungen nahe an der optimalen Spielfunktion liegt. Dies betrifft vor allem die TD-Agenten, bei denen eine Konvergenz auf die optimale Spielfunktion nicht vorbestimmt ist. Die TD-Agenten lernen ausschließlich durch Self-Play und es gibt keinen allwissenden „Lehrer“, der Abweichungen vom Optimum korrigiert. Eine möglichst objektive Einschätzung der Spielstärke ist deswegen wichtig, jedoch alles andere als trivial.

Der einzige Agent, der im GBG-Framework integriert ist und für eine objektive Evaluation eingesetzt werden kann, ist der Minimax-Agent. Wird seine Suchbaumhöhe nicht begrenzt, findet er nach Definition die optimale Spielfunktion. Wie bereits geschildert wird sein Suchbaum schnell so umfangreich, dass er nur für kleinere Spielbretter effektiv eingesetzt werden kann. Auf dem verwendeten Rechner mit 8 GB Arbeitsspeicher ist das 4x4-Spielbrett das Größte, für das der vollständige Suchbaum erstellt werden kann.

Die Evaluation durch den Minimax-Agenten verläuft so, dass der zu evaluierende Agent den ersten Zug hat, während der Minimax-Agent als zweiter Spieler spielt. Hätte der Minimax-Agent den ersten Zug, würde er aufgrund der in Hex existierenden optimalen Strategie für den ersten Spieler immer gewinnen. Ziel für den zu evaluierenden Agenten ist es, zuverlässig gegen Minimax gewinnen zu können. Dabei muss berücksichtigt werden, dass der Minimax-Agent reine Zufallszüge wählt, wenn für den zu evaluierenden Agenten im aktuellen Zustand eine optimale Strategie existiert. Je nachdem, welche Züge der Minimax-Agent als Zufallszüge gewählt hat, kann es für ihn nicht möglich sein, Schwächen in der Strategie seines Gegenspielers auszunutzen. Dieses Verhalten ist jedoch nicht unbedingt ein Schwachpunkt des Minimax-Agenten. Dadurch, dass er nicht deterministisch handelt, kann bei der Evaluation eine größere Anzahl an Zuständen abgedeckt werden, sodass auch nur minimale Abweichungen von der optimalen Spielfunktion zum Vorschein kommen können. Voraussetzung dafür ist jedoch, dass beim Vergleich eine genügend hohe Anzahl an Spielen gespielt wird. Da die Implementierung des Minimax-Agenten im GBG-Framework einen einmal erstellten Suchbaum für Folgespiele wiederverwendet, ist es problemlos möglich, eine große Anzahl an Evaluationsspielen durchzuführen.

Für Spielbretter mit der Größe 5x5 und größer wird die Evaluation schwieriger, da kein beweisbar perfekt spielender Agent mehr zur Verfügung steht. Ein möglicher Ersatz für den Minimax-Agenten ist der MCTS-Agent, der, wie von Kocsis und Szepesvári [18] gezeigt, mit steigender Anzahl an Iterationen zur optimalen Spielfunktion konvergiert. Die Schwierigkeit ist dann jedoch, die Anzahl der benötigten Iterationen zu finden, um eine ausreichend hohe Spielleistung zu erreichen. Für das 4x4-Spielbrett kann dies noch durch Vergleich mit dem Minimax-Agenten bestimmt werden, bei größeren Spielbrettern muss abgeschätzt werden. Ein Hilfsmittel bei der Abschätzung ist die Messung der Gewinnrate des MCTS-Agenten im Spiel gegen sich selbst. Spielt der Agent sehr schlecht, ist eine dem Random-Agenten ähnelnde Leistung zu beobachten. Je höher die Iterationen, desto mehr nähert sich die Gewinnrate des Agenten mit dem ersten Zug der optimalen 100% an. Zwar ist eine

Gewinnrate von 100% im Self-Play noch kein Beweis dafür, dass eine optimale Strategie gefunden wurde, jedoch ist dies ein gutes Indiz dafür, dass der Agent relativ gut spielt.

Dadurch, dass die *Default Policy* des MCTS-Agenten die Züge während der Simulation vollkommen zufällig wählt, ist der MCTS-Agent ebenso wie der Minimax-Agent nicht deterministisch. Deshalb müssen eine gewisse Anzahl an Evaluationsspielen stattfinden, um eine statistisch signifikante Aussage über die Leistung des zu evaluierenden Agenten treffen zu können. Weiterhin baut der MCTS-Agent für jeden durchgeführten Zug seinen Suchbaum erneut auf, weshalb die Berechnung ausreichend vieler Simulationsschritte für eine gute Leistung schnell sehr zeitintensiv wird, wenn hunderte oder tausende Evaluationsspiele gespielt werden müssen. Beim Einsatz des MCTS-Agenten werden eher wenige Spiele mit hohen Iterationen gespielt, als umgekehrt, wenn beides in Kombination nicht möglich ist. Die Wahrscheinlichkeit, eine Schwachstelle im zu evaluierenden Agenten zu finden, ist dadurch höher.

Weiterhin wurden auch außerhalb des GBG-Frameworks nach verwendbaren Evaluatoren gesucht. Die Forschungsgruppe „Computer Hex Group“ von der University of Alberta stellen auf ihrer Webseite den Quelltext für ihre Agenten MoHex und Wolve von 2011 zur Verfügung. Es wurde versucht, den Quelltext zu kompilieren, jedoch erwies sich dies aufgrund der Abhängigkeit von mittlerweile sehr alten Versionen der verwendeten Software-Bibliotheken als schwierig. Versuche, die benötigten Versionen der Software-Bibliotheken zu finden, sowie den Quelltext per Hand an neuere Versionen der Bibliotheken anzupassen, sind beide fehlgeschlagen.

Ein weiterer externer Agent, der sich für den Einsatz als Evaluator eignet, ist „Hexy“, der im Jahr 2000 den Hex-Wettbewerb der „*Computer Games Olympiad*“ gewann. Hexy ist vorkompiliert von der Webseite des Entwicklers zu beziehen. Der Entwickler, Vadim V. Anshelevich, hat eine Genehmigung für die Verwendung von Hexy für diese Arbeit erteilt. Da Hexy nicht ohne größeren Aufwand für automatische Evaluation integriert werden kann, findet die Evaluation von Hand statt. Dafür spielt der zu evaluierende Agent im GBG-Framework gegen den Human-Agenten, während im Hexy-Programm der Hexy-Agent als Spieler mit dem zweiten Zug ebenfalls gegen einen menschlichen Spieler spielt. Die Züge beider Agenten werden dann per Hand in das jeweils andere Programm übertragen. Da sowohl der TD- und TDNT-Agent als auch Hexy vollkommen deterministisch spielen, ist nur ein Vergleichsspiel notwendig. Hexy erlaubt die Verwendung verschiedener Schwierigkeitsstufen, es wird jedoch ausschließlich mit der höchsten Stufe („Expert“) verglichen.

5. Ergebnisse

5.1 Minimax

Spielt der Agent eine große Anzahl an Spielen gegen sich selbst, gewinnt der Agent mit dem ersten Zug zuverlässig jedes Spiel. Dieses Ergebnis ist zu erwarten gewesen, jedoch dient es auch zur Überprüfung der korrekten Implementierung des Spiels und des Agenten. Getestet wurden zu dem Zweck jeweils 10^5 Spiele. Es wurde versucht den Agenten auf dem 5×5-Spielbrett einzusetzen, jedoch reichen die 8 GB Arbeitsspeicher des verwendeten Rechners dafür nicht aus.

5.2 Random

Der Random-Agent ist nicht sehr aussagekräftig, wenn er mit anderen Agenten verglichen wird. Es wurde jedoch ein Test mit dem Agenten gegen sich selbst durchgeführt. Erwartet wurde eine ungefähre Gewinnrate des Agenten mit dem ersten Zug von etwa 50%. Für den ersten Spieler existieren zwar optimale Strategien, aufgrund der hohen Spielbaumgröße sollten diese aber keinen großen Einfluss auf das Ergebnis haben, da der Random-Agent zufällige Züge wählt.

Wie in untenstehender Abbildung 11 zu sehen ist, hängt das Ergebnis davon ab, ob die Anzahl der Spielfelder auf dem Spielbrett gerade oder ungerade ist. Auf Spielbrettern mit gerader Anzahl an Feldern ist die Gewinnchance für beide Random-Agenten 50%, während auf Spielbrettern mit ungerader Anzahl an Feldern ein klarer Vorteil für den Spieler mit dem ersten Zug besteht. Je größer dabei das Spielbrett ist, desto geringer wird der Vorteil für den ersten Spieler. Ein Spielbrett mit nur einem Feld bedeutet natürlich einen automatischen Sieg für den ersten Spieler. Auf dem 11×11-Spielbrett war die Gewinnchance für den ersten Spieler noch bei ca. 52%.

Dieses Ergebnis hat jedoch keinen großen Einfluss auf die Testergebnisse der anderen Agenten. Da immer eine optimale Strategie für den ersten Spieler besteht, sollte ein gut spielender Agent als erster Spieler unabhängig von der Spielbrettgröße sicher gewinnen können. Diese Ergebnisse wären nur dann relevant, wenn ein nicht perfekt spielender Agent mit dem Random-Agenten verglichen wird. Da der Random-Agent kein guter Maßstab ist, ist dies jedoch nicht zu empfehlen.

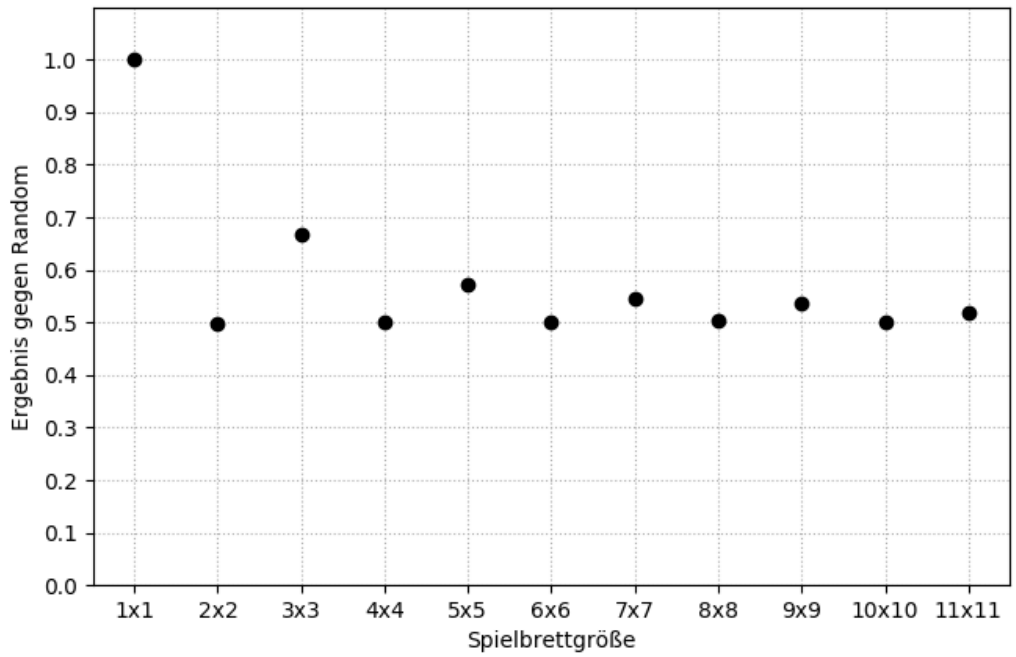

Abbildung 11: Ergebnis des Random-Agenten gegen sich selbst auf verschiedenen Spielbrettgrößen. Auf Spielbrettern mit gerader Anzahl an Feldern ist die Gewinnrate bei 50%, bei ungeraden gibt es einen Vorteil für den Spieler mit dem ersten Zug.

5.3 Monte-Carlo Tree Search

Der MCTS-Agent braucht kein Vorwissen über das Spiel, um sich der optimalen Spielstrategie anzunähern. Die Spielstärke hängt davon ab, wie groß der Zustandsraum ist und wie viele Iterationen des Algorithmus durchgeführt werden. Dem MCTS-Agenten ist es möglich, bei kleineren Spielfeldern wie der Minimax-Agent perfekt zu spielen, ohne dass er einen ähnlich großen Suchbaum aufbauen muss. Vielversprechend ist, dass auch MoHex, der weltbeste Hex-Agent, das MCTS-Verfahren einsetzt. Dabei muss jedoch beachtet werden, dass bei MoHex der durchsuchte Zustandsraum durch verschiedene Verfahren verkleinert wird, um die benötigte Zeit pro Zug zu reduzieren und den Suchbaum klein zu halten. Ziel des GBG-Frameworks ist es jedoch, Agenten bereitzustellen, die für eine Vielzahl von Spielen eingesetzt werden können, weswegen kein spieltheoretisches Wissen integriert wird, um die Leistung des MCTS-Agenten zu optimieren. Deshalb kann für größere Spielbretter keine vergleichbar gute Leistung wie die der weltbesten Agenten erwartet werden. Auf kleineren Spielbrettern sollte es jedoch in angemessener Zeit pro Zug möglich sein, einen perfekt spielenden Agenten zu erhalten. Dies wird im Folgenden untersucht.

2×2-Spielbrett

Auf dem 2×2-Spielbrett ist wie erwartet eine schnelle Konvergenz auf eine optimale Spielweise zu beobachten. Getestet wurde die Leistung gegen Minimax in jeweils 100.000 Vergleichsspielen unter Verwendung von $\mathrm{K} = \sqrt{2}$ als Explorationsfaktor. Beginnend bei nur vier Iterationen pro Zug wurde bereits eine Gewinnrate von 68,9% gemessen. Weniger als vier Iterationen sind in der Implementierung des MCTS-Agenten im GBG nicht möglich. Bei 25 Iterationen betrug die Gewinnrate bereits 99,7%. Schon bei 30 Iterationen gewann der MCTS-Agent zuverlässig jedes Spiel gegen den Minimax-Agenten.

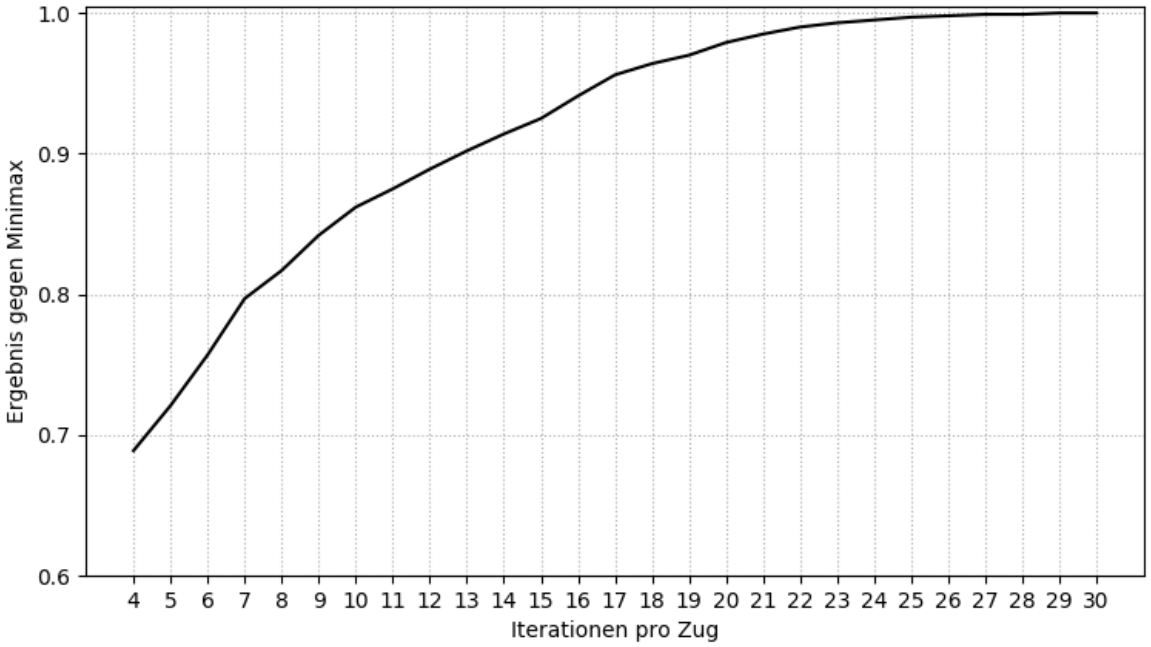

Abbildung 12: MCTS gegen Minimax auf dem 2×2-Spielbrett mit $K = \sqrt{2}$.
Jeweils 100.000 Vergleichsspiele.

3×3-Spielbrett

Auf einem 3×3-Spielbrett ist auch noch zu erwarten, dass eine Konvergenz auf die optimale Spielweise ohne viel Rechenzeit pro Zug möglich ist. In den folgenden Vergleichen wurden jeweils 1.000 Spiele gespielt. Die dabei gemessene Leistung gegen Minimax ist in folgender Tabelle 1 zu sehen:

Iterationen	Gewinnrate
50	67,9%
100	82,0%
500	98,3%
1.000	99,7%
1.200	**100%**

Tabelle 1: MCTS gegen Minimax (3x3), $K = \sqrt{2}$

Ab ca. 1200 Iterationen pro Zug erreicht der MCTS-Agent unter Verwendung von $\mathrm{K} = \sqrt{2}$ eine perfekte Spielweise auf dem 3×3-Spielbrett. Die Spiele dauerten jedoch schon deutlich länger, sodass jeweils nur 1.000 Vergleichsspiele durchgeführt wurden. Deshalb wurde als nächstes versucht, die Leistung des Agenten zu optimieren, ohne dabei die Anzahl der Iterationen zu erhöhen. Die Parameter, die beim MCTS-Agenten angepasst werden können, sind die *Rollout Depth*, die Baumhöhe und die Explorationskonstante K. Die *Rollout Depth* zu begrenzen macht bei Hex keinen Sinn, da die Spiele eine berechenbare Maximallänge besitzen, die auch nicht sehr hoch ist (neun Züge im Falle des 3×3-Spielbretts). Die Baumhöhe zu begrenzen würde bedeuten, dass es keine Garantie mehr für die Konvergenz auf eine optimale Strategie mehr gibt. Der verbleibende Parameter ist also der Explorationsfaktor K, dessen Auswirkung im Folgenden geprüft wird.

Ausgehend von der Gewinnrate von 98,3% bei 500 Iterationen des Algorithmus wurde geprüft, ob die Gewinnrate durch Wahl eines anderen Werts für K verbessert werden kann:

K	Gewinnrate
0	71,3%
0,01	78,6%
0,1	87,1%
0,5	96,3%
1	**98,5%**
$\sqrt{2}$	98,3%
5	95,1%
100	94,5%
10.000	93,9%

Tabelle 2: MCTS gegen Minimax (3x3), Iterationen: 500

Wie in Tabelle 2 zu sehen ist, scheint ein zu geringer Explorationsfaktor der Leistung des Agenten mehr zu Schaden, als ein zu hoher Faktor. Im Gegensatz zu dem vorher verwendeten Faktor $\mathrm{K} = \sqrt{2}$ steigert ein leicht gesenkter Explorationsfaktor $K = 1$ im Falle eines 3×3 Spielfeldes die Leistung des Agenten, wobei der Unterschied nur gering ist. Um das Risiko auf einen Messfehler zu verringern wurden diese beiden Werte für K mit 10.000 Vergleichsspielen gemessen, während bei allen anderen Werten nur 1.000 Vergleichsspiele gespielt wurden.

Dieses Ergebnis ist unerwartet, denn Arneson et al. berichteten in Ihrer Untersuchung von MCTS in Anwendung auf Hex, dass ihr Agent *MoHex* am stärksten spielt, wenn ein Explorationsfaktor von null gewählt wird [13]. Sie verwenden jedoch für die Auswahl des erfolgversprechendsten Kind-Knotens die sogenannte *all-moves-as-first*-Heuristik (AMAF). Nach Hembold und Parker-Wood erhöht die AMAF-Heuristik das Wissen, das aus jeder Simulation gewonnen wird [25]. Dies geschieht, indem der Simulationsschritt nicht nur die Bewertung aller Knoten aktualisiert, die bei der Simulation involviert waren, sondern auch einige der Kind-Knoten, die nicht direkt besucht wurden. Dieser Unterschied in der *tree policy* könnte der Grund dafür sein, warum hier ein von Arneson et al. abweichendes Ergebnis im Bezug zum Explorationsfaktor gemessen wurde.

4×4-Spielbrett

Für das nächst-größere Spielbrett wurde zuerst erneut versucht, die benötigte Anzahl an Iterationen für gutes Spielen unter Verwendung des Explorationsfaktors $\mathrm{K} = \sqrt{2}$ zu finden:

Iterationen	Gewinnrate
500	59,5%
1.000	73,1%
5.000	93,9%
10.000	99,0%
12.000	**99,4%**

Tabelle 3: MCTS gegen Minimax (4x4), $K = \sqrt{2}$, jeweils 1.000 Spiele

Um den vergrößerten Zustandsraum auf dem größeren Spielbrett zu berücksichtigen, wurden die Iterationen des Algorithmus pro Zug verzehnfacht. Dennoch sind durch die schnell ansteigende Komplexität niedrigere Gewinnraten zu beobachten. Da die benötigte Zeit für die 1.000 Vergleichsspiele durch die erhöhte Anzahl an Iterationen wieder deutlich angestiegen ist, wurde auch hier wieder ein Faktor K gesucht, der die Gewinnrate bei einer festen Anzahl an Iterationen optimiert. Dafür wurden 5.000 Iterationen gewählt, bei denen $\mathrm{K} = \sqrt{2}$ zu einer Leistung von 93,9% führte.

K	Gewinnrate
0	46,5%
0,01	51,2%
0,5	89,5%
1	**95,7%**
$\sqrt{2}$	93,9%
5	88,5%
10.000	85,7%

Tabelle 4: MCTS gegen Minimax (4x4), Iterationen: 5.000, jeweils 1.000 Spiele

Ebenso wie auf dem 3×3-Spielbrett erweist sich ein Explorationsfaktor $K = 1$ von den getesteten als der effektivste, diesmal mit etwas deutlicherem Unterschied. Insgesamt ist die Bedeutung des Parameters gegenüber dem 3×3-Spielbrett gestiegen, denn auch die Extremwerte haben einen größeren negativen Einfluss auf die Leistung. Dies könnte jedoch auch daran liegen, dass die Ausgangsgewinnrate von 93,9% beim 4×4-Spielbrett gegenüber 98,4% beim 3×3-Spielbrett geringer war.

Aufgrund dieser Ergebnisse wurde bei Spielbrettern größer als 4×4 auch $K = 1$ gewählt. Es ist jedoch nicht erwiesen, dass dieser Wert immer der beste für Hex ist. Es wäre möglich, dass es eine Beziehung zwischen K und der Spielbrettgröße gibt, die sich beim Vergleich zwischen 3×3 und 4×4 noch nicht verdeutlicht hat.

Da ab 5×5 nicht mehr gegen Minimax getestet werden kann, ist es noch sinnvoll zu testen, welche Gewinnrate der MCTS-Agent im Spiel gegen sich selbst aufweist:

Iterationen	Gewinnrate
1.000	80,2%
5.000	96,6%
10.000	**99,8%**

Tabelle 5: MCTS gegen MCTS (4x4), $K = 1$

Ein direkter Vergleich zu den Ergebnissen gegen den Minimax-Agenten ist wegen der Verwendung eines anderen Wertes für K nicht möglich. Einzig für 5.000 Iterationen wurde auch mit $K = 1$ gegen Minimax getestet. Die Gewinnrate gegen MCTS ist in diesem Fall wie zu erwarten höher: Der MCTS-Agent kann Schwächen seines Gegners nicht so gut ausnutzen, obwohl auch der Minimax-Agent in dieser Hinsicht nicht perfekt ist (siehe Kapitel 4). Dennoch kann durch das Spiel gegen sich selbst für größere Spielfelder grob abgeschätzt werden, ob der Agent bei den gewählten Iterationen einigermaßen gut spielt. Das Problem wird dabei jedoch werden, dass eine statistisch signifikante Anzahl an Vergleichsspielen immer länger dauert.

5×5-Spielbrett

Aufgrund des immer größer werdenden Zeitaufwands wurden beim 5×5-Spielbrett nur 100 Vergleichsspiele gespielt. Unter Verwendung von $K = 1$ wurde im Test MCTS gegen MCTS Folgendes gemessen:

Iterationen	Gewinnrate	Besiegt Hexy
1.000	66%	Nein
5.000	80%	Nein
10.000	87%	**Ja**
50.000	95%	**Ja**
100.000	**100%**	**Ja**

Tabelle 6: MCTS gegen MCTS (5x5), $K = 1$, 100 Spiele

Der Trend scheint zu sein, dass jede Vergrößerung des Spielbretts ungefähr eine Verzehnfachung der Iterationen pro Zug benötigt und der Zeitaufwand pro Spiel steigt entsprechend mit an. Erstmals wurde auch Hexy als Evaluator zur Hilfe genommen. Dass MCTS schon ab 10.000 Iterationen gewinnt, lässt sich wahrscheinlich dadurch erklären, dass Hexy keine tiefe Baumsuche verwendet. Hexy wurde für die Erkennung von Mustern auf dem Spielbrett konzipiert, was auf größeren Spielbrettern gut funktioniert, da dort eine Baumsuche unter Umständen zu aufwändig ist. Auf kleineren Spielbrettern wie 5×5 hat der MCTS-Agent jedoch einen klaren Vorteil. Weiterhin sollte auch erwähnt werden, dass die Züge vom MCTS-Agenten ein paar Sekunden dauern, während Hexy auf dieser Spielbrettgröße noch weniger als eine Sekunde braucht.

Zusammenfassung

Größere Spielbretter als 5×5 wurden nicht getestet ist, da weitere Erkenntnisse unwahrscheinlich sind. Die Erwartung, dass der MCTS-Agent auf kleineren Spielbrettern als perfekt spielender Agent einsetzbar ist, hat sich bestätigt. Weiterhin hat sich gezeigt, dass in Hex ein Wert von 1 für den Parameter K die Leistung des Agenten leicht verbessert. Für den weiteren Einsatz des MCTS-Agenten als Evaluator für den TD-Agenten und den TDNT-Agenten hat sich gezeigt, dass für ein Spielbrett mit der Größe $n \times n$ etwa 10^n Iterationen für nahezu perfektes Spielen des MCTS-Agenten benötigt werden. Der Einsatz des MCTS-Agenten für größere Spielbretter wird wegen des schnell steigenden Zeitaufwands problematisch.

5.4 Temporal Difference

Für den TD-Agenten mussten geeignete Features (Merkmale) gefunden werden, die zu jedem Zeitpunkt den Zustand des Spielbretts treffend beschreiben. Erste Versuche, die implementierten Feature-Modi zu testen, waren relativ erfolglos, da keine Konvergenz auf eine gute Spielweise stattgefunden hat. Um einen Fehler im TD-Agenten auszuschließen, wurde ein Feature-Modus erstellt, der auf dem 2×2-Spielbrett ein N-Tupel nachbildet. Der N-Tupel-Agent, der ebenfalls TDL zum Erlernen der Spielfunktion verwendet, funktionierte nach ein paar Startschwierigkeiten nämlich relativ gut. Mit diesem Feature-Modus fand eine erfolgreiche Konvergenz auf die perfekte Spielfunktion statt, weshalb ein Fehler im TD-Agenten ausgeschlossen werden kann. Damit dieser Feature-Modus erfolgreich lernen konnte, musste die Lernschrittweite auf $\alpha_{init} = 10$ und $\alpha_{final} = 0{,}1$ gesetzt werden.

Somit stand fest, dass die Probleme mit dem Agenten mit der Wahl der Features zusammenhängen. Die Feature-Modi, die implementiert wurden, werden im Folgenden beschrieben.

5.4.1 Feature-Modus 1

Für kleinere Spielbrettgrößen wurde ein Feature-Modus erstellt, der als Eingabevektor die Belegung jedes Feldes in einen Zahlenwert übersetzt. Dadurch bleibt es dem Agenten selber überlassen, während seiner Trainingsspiele wichtige Muster auf dem Spielbrett zu erkennen und in den Gewichtungen des Netzes abzubilden. Der Vorteil dabei ist, dass dieser Feature-Modus einfach zu implementieren ist, problemlos mit der Spielbrettgröße skaliert und dem Agenten kein Vorwissen zum Spiel gibt. Es ist jedoch möglich, dass das Training auf diese Art und Weise sehr lange dauert. Für die folgenden Tests wurde deshalb das 3×3-Spielbrett verwendet.

Der Feature-Vektor dieses Modus hat die Länge $n^2 + 1$. Jedes Feld des Spielbretts hat einen eigenen Eintrag im Vektor. Der Wert für jedes Feld beträgt 0, wenn das Feld dem schwarzen Spieler gehört, 1, wenn es dem weißen Spieler gehört und -1 wenn kein Spieler das Feld

belegt. Zusätzlich wird auch die Identifikationsnummer des Spielers im Vektor angegeben, der gerade am Zug ist (0 oder 1).

Variation der Lernschrittweite

Als erstes wurde bestimmt, in welchem Bereich die Lernschrittweite α liegen sollte, damit ein erfolgreiches Lernen der Spielfunktion möglich ist. Wie in Abbildung 13 zu sehen ist, hat sich eine Schrittweite von $\alpha_{init} = 1$ zu Beginn des Trainings und $\alpha_{final} = 0{,}01$ gegen Ende des Trainings als am effektivsten herausgestellt. Die Abbildung zeigt den durchschnittlichen Trainingsverlauf nach zehn Durchläufen pro Konfiguration. Evaluiert wurde durch 100 Evaluationsspiele gegen Minimax nach jeweils 100 Trainingsspielen. Insgesamt wurde das lineare Netz für 100.000 Spiele trainiert. Dabei wurde eine Sigmoid-Funktion auf den Ausgabewert angewandt. Es ist deutlich zu sehen, dass keine Konvergenz auf eine perfekte Spielweise stattgefunden hat.

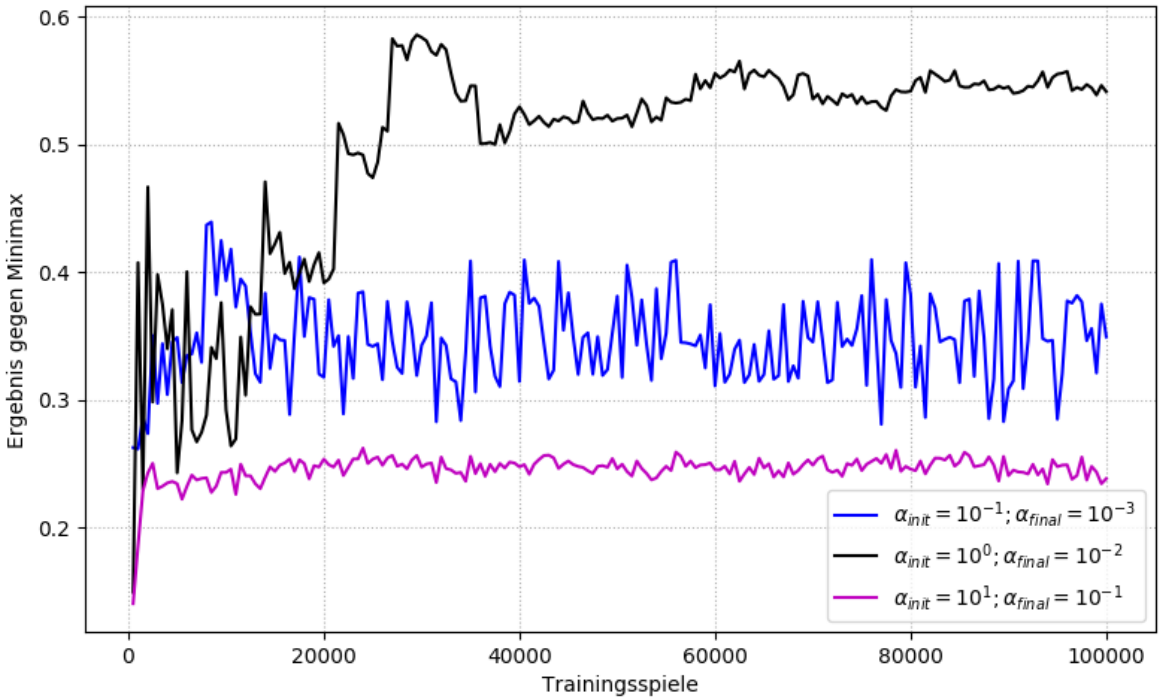

Abbildung 13: Einfluss der Lernschrittweite auf den TD-Agenten

Verwendung der Sigmoid-Funktion

Nach Konen kann es manchmal vorteilhaft sein, keine Sigmoid-Funktion zu verwenden [22]. Wie in Abbildung 14 zu sehen ist, scheint die Verwendung der Sigmoid-Funktion für diesen Feature-Modus jedoch zu einer besseren Leistung zu führen. Für die Lernschrittweite wurde das beste Ergebnis aus dem letzten Test verwendet. Erneut wurde der Durchschnitt aus zehn Trainingsverläufen gebildet.

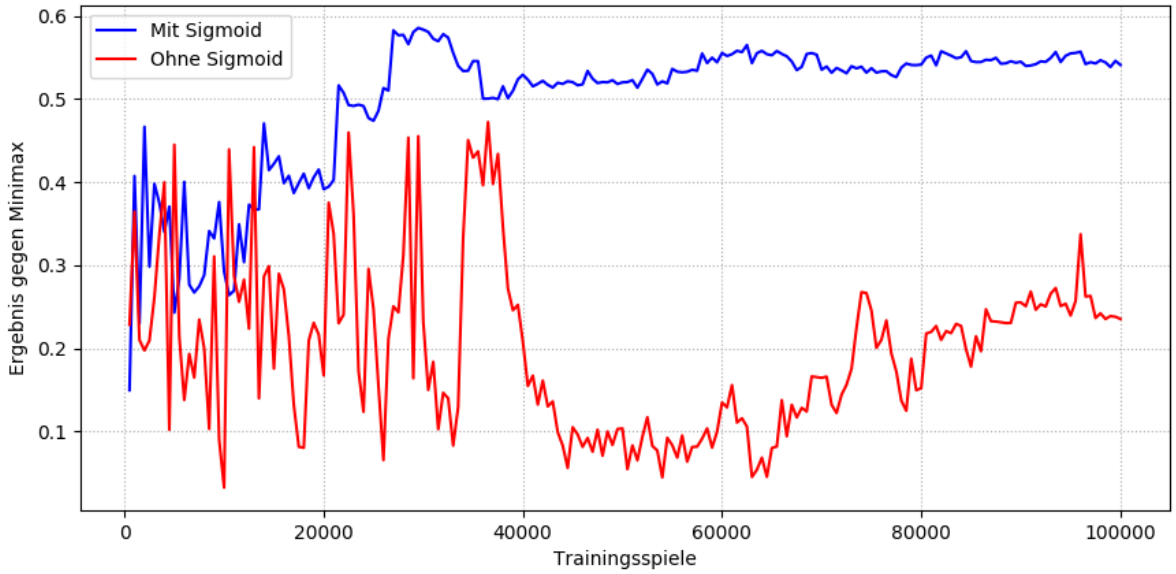

Abbildung 14: Auswirkung der Sigmoid-Funktion auf den TD-Agenten (3x3-Spielbrett, lineares Netz)

Verwendung eines neuronalen Netzes

Als nächstes wurde gemessen, ob such durch die Verwendung eines neuronalen Netzes ein besseres Ergebnis erzielen lässt. Wie in Abbildung 15 unschwer zu erkennen ist, ist dies der Fall. Unter Verwendung eines neuronalen Netzes wurde nach dem Training eine durchschnittliche Gewinnrate von ca. 87% erzielt, während beim linearen Netz nur 54% erreicht wurden. Dabei ist jedoch auch die eingezeichnete Standardabweichung zu beachten. Das lineare Netz war durchaus in der Lage der optimalen Spielfunktion nahe zu kommen, hatte aber öfters vollkommene Fehlschläge beim Training, die den Durchschnitt senken. Ignoriert man diese Fehlschläge, würde der Erfolg des linearen Netzes bei ungefähr 83% liegen. Weiterhin ist zu sehen, dass das neuronale Netz ein längeres Training benötigte. Der Graph des neuronalen Netzes flachte erst nach 60.000 bis 70.000 Trainingsspielen ab, während das lineare Netz schon ab ca. 40.000 Trainingsspielen keinen Fortschritt mehr vorweisen konnte. Dennoch ist das Ergebnis nicht ganz wie erhofft, denn es ist nicht gelungen, eine Parameter-Konfiguration zu finden, die sich zuverlässig einer perfekten Spielweise annähert.

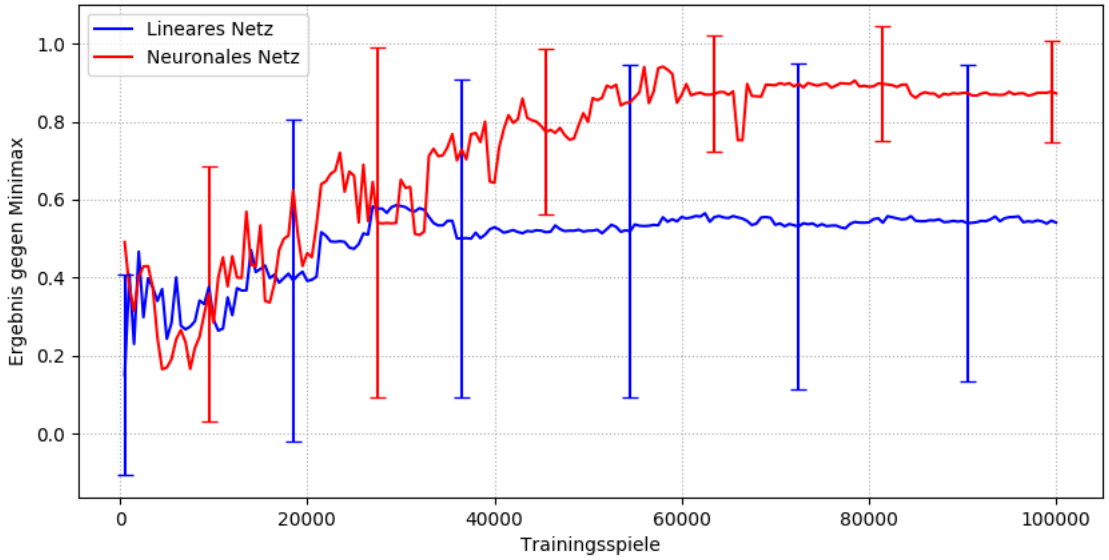

Abbildung 15: Vergleich zwischen linearem und neuronalem Netz auf dem 3x3-Spielbrett

5.4.2 Feature-Modus 2

Für den zweiten Feature-Modus wurde versucht, Vorwissen über das Spiel zu integrieren, um so vielleicht eine bessere Konvergenz auf eine perfekte Spielfunktion zu erreichen.

Implementierte Features

Das erste Feature in diesem Modus ist die Länge der längsten Kette, die der Agent gebildet hat. Dafür wird nur die Länge in die Richtung gezählt, in die der Agent spielen muss, um das Spiel zu gewinnen. So würde für den schwarzen Spieler jegliche Bewegung in Richtung einer der beiden weißen Ränder nicht zur Länge der Kette beitragen. Dadurch wird gewährleistet, dass die Länge der Kette einen genaueren Hinweis darauf gibt, wie gut die Chancen des Agenten sind, das Spiel zu gewinnen. Damit ein Vergleich zu seinem Gegenspieler stattfinden kann, wird als ein weiteres Feature auch die längste Kette des Gegenspielers berechnet.

In Hex können sich verschiedene Muster auf dem Spielbrett bilden, die einen großen Einfluss auf den Ausgang eines Spiels haben können [4]. Ein häufig vorkommendes Muster war bereits in Abbildung 3 zu sehen. Die beiden schwarzen Steine teilen sich zwei Nachbarfelder, die beide noch unbesetzt sind. In diesem Fall macht es Sinn, die Steine als bereits verbunden zu betrachten, da der Gegenspieler nichts mehr tun kann, um das Verbinden der Steine zu verhindern. Bei der Berechnung der bereits erwähnten längsten Ketten fließt das Wissen über diese virtuellen Verbindungen mit ein. Gibt es zum Beispiel zwei Ketten mit der Länge zwei, die durch eine virtuelle Verbindung verbunden sind, so zählt dies als nur eine Kette mit der Länge vier. Die virtuelle Verbindung wird erst dann mitgezählt, wenn eine tatsächliche Verbindung besteht. Die Anzahl der existierenden virtuellen Verbindungen pro Spieler wird dabei gezählt und als ein weiteres Feature verwendet.

Die bis jetzt definierten Features sind für Spielbretter nützlich, auf denen schon ein paar Spielzüge stattgefunden haben. Wird der Eingabevektor nach nur einem Zug betrachtet, sieht dieser immer gleich aus, egal welches Feld gewählt wurde. Um auch den ersten Zug zu

bewerten, wurde die Summe der unbesetzten Nachbarfelder um die eigenen Felder herum als zwei weitere Features definiert; Einmal für den Agenten und einmal für seinen Gegenspieler. Die Idee hinter diesem Feature ist, dass diese Summe den Grad der Beweglichkeit darstellt, den der Spieler hat. Je kleiner dieser Wert ist, desto eher können Manöver geblockt werden, insbesondere wenn das Wissen über virtuelle Verbindungen mit einbezogen wird. Außerdem gibt dieser Wert eine relativ gute Bewertung des ersten Zuges bei Spielbrettern der Größe 2x2 und 3x3. Bei diesen Spielbrettgrößen gibt es eine recht hohe Korrelation zwischen der Anzahl an Nachbarfeldern und dem tatsächlichen Nutzen eines Feldes.

Ab 4x4 könnte dieses Feature jedoch mehr schaden als nützen, denn es existieren nur optimale Strategien für die Hälfte der Felder mit sechs Nachbarn, während es auch optimale Strategien für die zwei Felder mit nur drei Nachbarn gibt, wie in Abbildung 16 zu sehen ist.

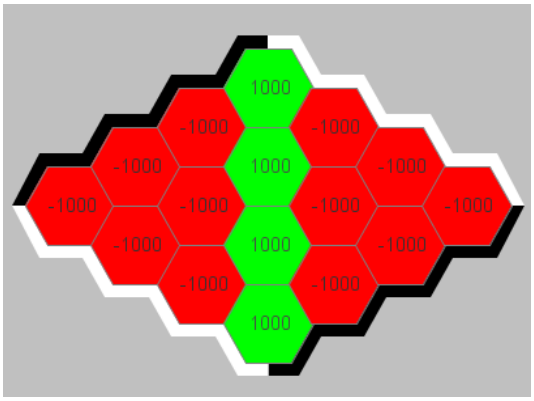

Abbildung 16: Bewertung der Eröffnungszüge des 4x4-Spielbretts durch Minimax. Für die grünen Felder existiert eine optimale Strategie, für die roten nicht.

Um ein dem Agenten wenig Kontext für die bereits aufgeführten Features zu bieten, wird zuletzt noch die Anzahl der Steine des Agenten als ein Feature definiert.

Es gibt nun also sieben Features, die hier noch einmal tabellarisch zusammengefasst werden:

Feature	Wertebereich	Datentyp
Länge der längsten Kette des Agenten	$[0, n]$	Ganzzahl
Anzahl der virtuellen Verbindungen des Agenten	$[0, n^2]$	Ganzzahl
Summe der unbesetzten Nachbarfelder des Agenten	$[0, n^2]$	Ganzzahl
Länge der längsten Kette des Gegenspielers	$[0, n]$	Ganzzahl
Anzahl der virtuellen Verbindungen des Gegenspielers	$[0, n^2]$	Ganzzahl
Summe der unbesetzten Nachbarfelder des Gegenspielers	$[0, n^2]$	Ganzzahl
Anzahl der Steine des Agenten	$[0, \frac{n^2}{2} + 1]$	Ganzzahl

Tabelle 7: Auflistung aller Features im zweiten Feature-Modus

Mögliche weitere Features

Außer den sieben bereits implementierten Features ließen sich noch Weitere vorstellen. Eine Schwachstelle des vorgestellten Feature-Modus ist es, dass in den Features nicht repräsentiert wird, ob der Weg von der längsten Kette zum noch nicht verbundenen Spielfeldrand durch den Gegenspieler blockiert wurde. Der Agent kann zwar eine Kette der Länge $n - 1$ haben, wonach im Optimalfall nur noch ein Zug zum Sieg fehlt, jedoch ist es möglich, dass dafür ein langer Umweg genommen werden muss, sodass der Agent doch verliert. Ein mögliches Feature, das diese Situation erkenntlich werden lässt, wäre die minimale Distanz zum Ziel. Dazu könnte ausgehend von dem oder den noch nicht verbundenen Ende(n) der längsten Kette ein Wegfindungsalgorithmus über die noch unbesetzten Felder laufen.

Das Problem, dass die Anfangszustände eines Spiels nicht gut abgebildet werden, könnte durch ein Feature gelöst werden, das den ersten Zug des Agenten mit bereits bekannten Lösungen vergleicht. Dies ginge jedoch nur bis zur Spielbrettgröße 9x9, da zum Zeitpunkt des Schreibens noch nicht alle Lösungen für 10x10 und größer bekannt sind. Die Verwendung eines solchen Features wäre jedoch fragwürdig, da so dem Agenten indirekt schon Teile einer Lösung als Eingabe gegeben werden. Ziel sollte es sein, dass der Agent selber auf die Lösung kommt.

Variation der Lernschrittweite

Wie beim ersten Feature-Modus wurde versucht, die Parameterwahl des zweiten Feature-Modus zu optimieren. Erneut wurde zuerst bestimmt, in welchem Bereich die Lernschrittweite liegen sollte. Wie in Abbildung 17 zu sehen ist, sind die Unterschiede in der Leistung nicht sehr groß, obwohl zwischen jeder Konfiguration eine Größenordnung liegt. Verwendet wurde ein lineares Netz mit Verwendung der Sigmoid-Funktion. Erneut wurde der aus jeweils zehn Trainingsverläufen pro Konfiguration der Durchschnitt gebildet. Die durchschnittlich beste gemessene Leistung nach dem Training liegt jedoch bei gerade mal 50%. Weil dieser Feature-Modus mehr Zeit für die Berechnung braucht, wurden nur 10.000 Trainingsspiele gespielt. Dem Verlauf der Graphen nach scheint es aber, als würden weitere Trainingsspiele die Leistung nicht verbessern.

Verwendung der Sigmoid-Funktion

Um zu testen, ob die Verwendung der Sigmoid-Funktion die Leistung des Agenten behindert, wurde die Leistung ohne Sigmoid-Funktion gemessen. Wie in Abbildung 18 zu sehen ist, verschlechtert sich die Leistung des Agenten auch bei diesem Feature-Modus, wenn die Sigmoid-Funktion nicht verwendet wird.

Verwendung eines neuronalen Netzes

Beim ersten Feature-Modus hatte die Verwendung eines neuronalen statt eines linearen Netzes einen deutlichen Leistungszuwachs ermöglicht. Wie in Abbildung 19 zu sehen ist das beim zweiten Feature-Modus leider nicht der Fall.

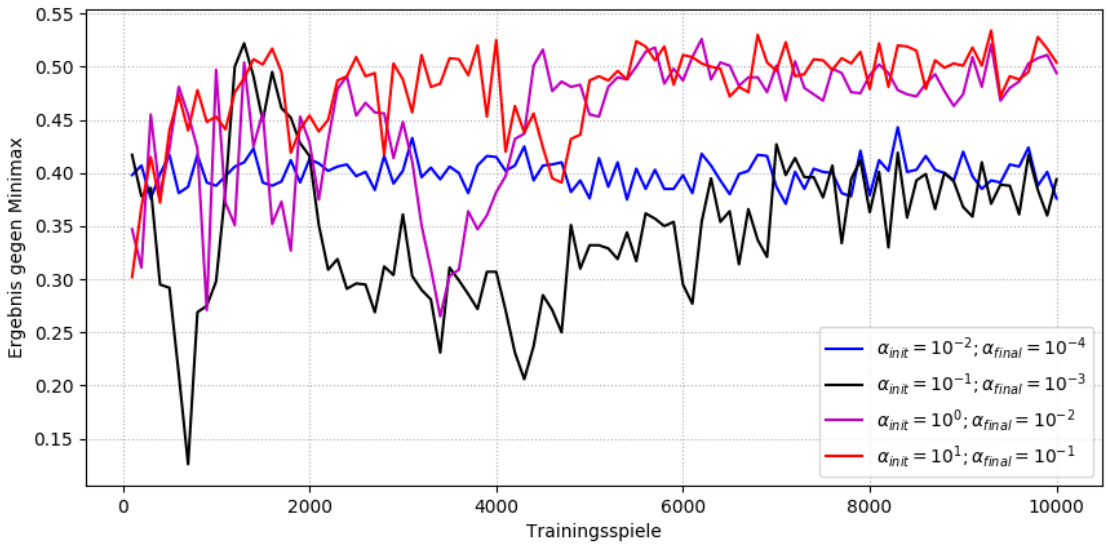

Abbildung 17: Variation der Lernschrittweite für den zweiten Feature-Modus

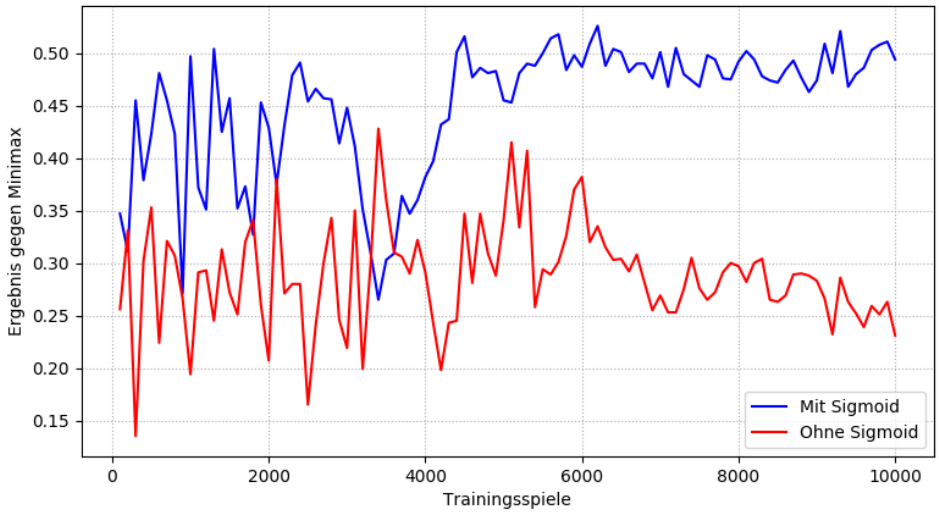

Abbildung 18: Einfluss der Sigmoid-Funktion auf den zweiten Feature-Modus

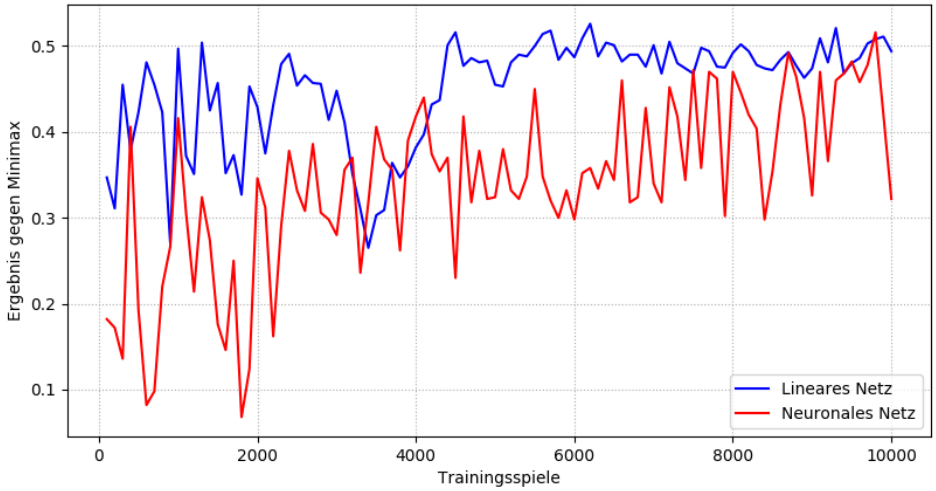

Abbildung 19: Vergleich zwischen neuronalem und linearem Netz beim zweiten Feature-Modus

5.4.3 Feature-Modus 3

Der zweite Feature-Modus verwendet viel Zeit dafür, die längste Kette zu berechnen. Effizienter wäre es, zu den dafür untersuchten Feldern weitere Informationen zu speichern, um diese für weitere Features verwenden zu können. Da der Ansatz des zweiten Feature-Modus sich nicht groß optimieren lässt, wurde mit einem neuen Ansatz ein dritter Feature-Modus erstellt.

Der dritte Feature-Modus untersucht jedes Feld des Spielers und versucht dabei die Beziehungen des jeweiligen Felds zu umliegenden Feldern zu ermitteln. Dabei entsteht eine Liste aus Verbindungen, die dann im Anschluss für die Features weiter aufgearbeitet werden kann. Die Verbindungen, die dabei erkannt werden, sind die Folgenden:

1) Direkte Verbindungen: Eine direkte Verbindung zwischen zwei Feldern besteht, wenn diese direkt aneinander anliegen.
2) Schwache Verbindungen: Zwei Felder besitzen eine schwache Verbindung, wenn zwischen ihnen genau ein leeres Feld liegt. Wenn der Spieler in seinem Zug das leere Feld nicht wählt, riskiert er, dass die Verbindung vom Gegenspieler geblockt wird.
3) Virtuelle Verbindungen: Diese Art von Verbindung existiert, wenn zwei Felder des gleichen Spielers durch zwei leere Felder verbunden sind.

Die Ränder des Spielbretts werden dabei durch zwei virtuelle Felder mit den Koordinaten (-1, -1) für den ersten Rand und (-2, -2) für den Zweiten repräsentiert, damit diese auch in den Verbindungen abgebildet werden. Nachdem die Liste der Verbindungen erstellt wurde, müssen im Anschluss nur noch diese Verbindungen betrachtet werden, um zum Beispiel die Länge der längsten Kette zu bestimmen.

Die Features, die der zweite Feature-Modus verwendet hat, sind im dritten Feature-Modus auch enthalten. Zusätzlich kommen hinzu:

Feature	Wertebereich	Datentyp
Anzahl der direkten Verbindungen des Agenten	$[0, n^2]$	Ganzzahl
Anzahl der schwachen Verbindungen des Agenten	$[0, n^2]$	Ganzzahl
Anzahl der direkten Verbindungen des Gegenspielers	$[0, n^2]$	Ganzzahl
Anzahl der schwachen Verbindungen des Gegenspielers	$[0, n^2]$	Ganzzahl

Tabelle 8: Neue Features des dritten Feature-Modus

Um den Geschwindigkeitszuwachs gegenüber dem zweiten Feature-Modus zu testen wurde ein lineares Netz für 100.000 Trainingsspiele auf einem 4×4-Spielbrett trainiert. Der zweite Feature-Modus brauchte für das Training 84 Sekunden, der dritte Feature-Modus benötigte 63 Sekunden. Zwischen den Tests wurde das Programm neu gestartet. Der Geschwindigkeitszuwachs ist nicht so hoch wie erhofft, aber dennoch ist es eine Verbesserung.

Zusätzlich wurde erneut auf dem 3×3-Spielbrett die Leistung gemessen.

Variation der Lernschrittweite

Auch bei der Leistung des dritten Feature-Modus zeigte sich keine erfolgreiche Konvergenz auf das Optimum. Die Werte $\alpha_{init} = 0{,}1$ zu Beginn des Trainings und $\alpha_{final} = 0{,}001$ gegen Ende des Trainings haben sich jedoch als um einiges besser herausgestellt, als die anderen Konfigurationen, wenn auch mit sehr großer Varianz. In unten stehender Abbildung 20 sind die durchschnittlichen Trainingsverläufe nach je 20 Durchläufen zu sehen.

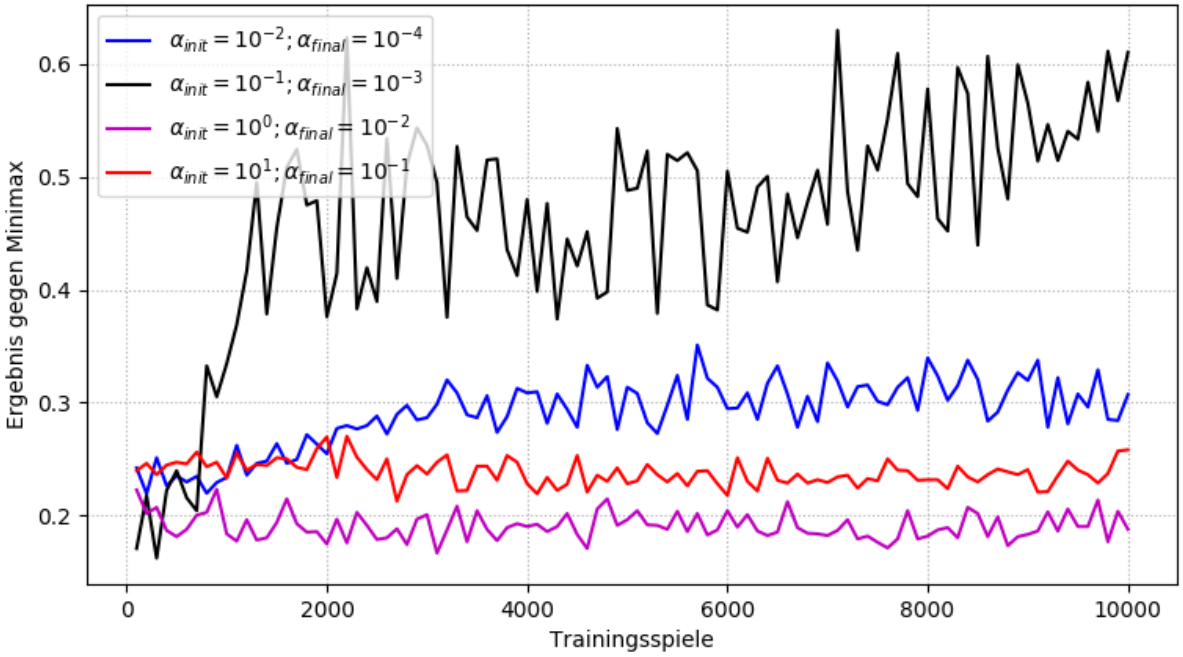

Abbildung 20: Variation der Lernschrittweite beim dritten Feature-Modus

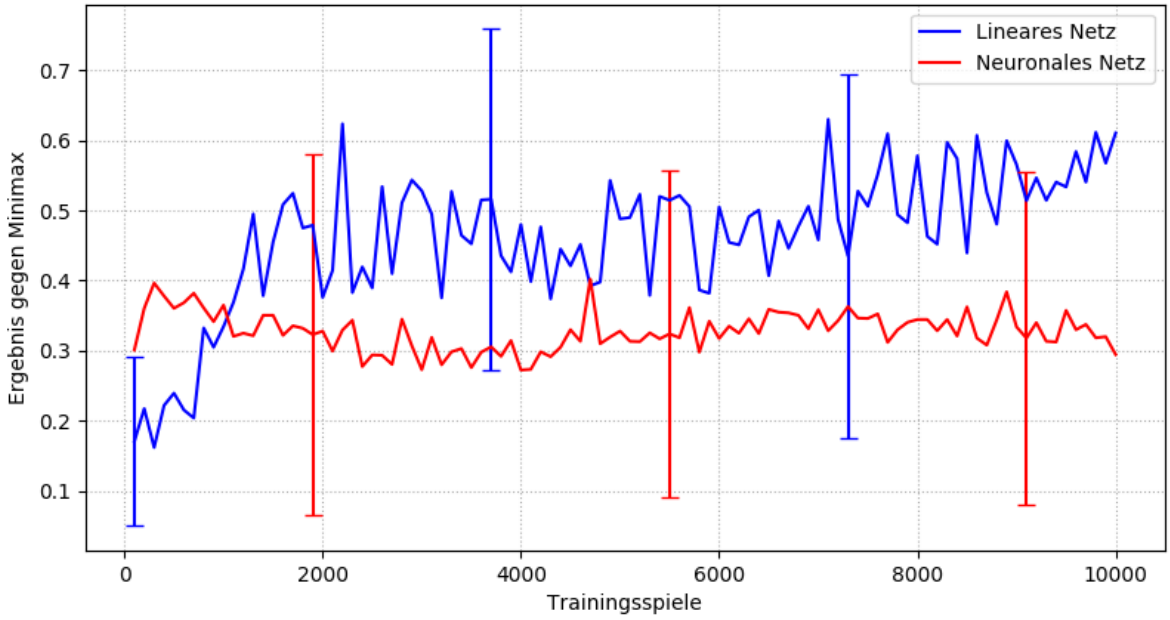

Abbildung 21: Verwendung eines neuronalen Netzes für den dritten Feature-Modus

Verwendung eines neuronalen Netzes

Wie auch beim zweiten Feature-Modus ist das Ergebnis unter Verwendung eines neuronalen Netzes noch schlechter. In beiden Fällen gab es sehr große Abweichungen vom Durchschnitt, wie in Abbildung 21 zu sehen ist. Gewählt wurde die Lernschrittweite, die im letzten Test am besten abgeschnitten hat. Erneut wurden 20 Trainingsdurchläufe pro Konfiguration gemessen.

5.4.4 Zusammenfassung

Der erste Feature-Modus, der nur die rohen Spielbrett-Informationen verwendet, hat bei der Verwendung eines neuronalen Netzes einen einigermaßen guten Erfolg erzielen können. Es wurde jedoch vermutet, dass dieser Feature-Modus nicht auf größere Spielbretter anwendbar ist. Um dies zu testen wurde abschließend ein Agent mit der besten ermittelten Konfiguration auf einem 4×4-Spielbrett trainiert. Dafür wurde eine Trainingsdauer von 1 Mio. Trainingsspielen gewählt. Das Training dauerte etwa eine Viertelstunde und der Agent gewann nach dem Training kein einziges Spiel gegen Minimax, weshalb auf die Erstellung eines aussagekräftigen Graphen des Trainingsverlaufs, für den mehrere Durchläufe notwendig wären, verzichtet wurde.

Mit dem zweiten und dem dritten Feature-Modus war kein erfolgreiches Lernen der Spielfunktion möglich. Der Grund dafür ist sehr wahrscheinlich, dass die gewählten Features den Spielzustand nicht gut genug beschreiben. Zwar konnten mit beiden ungefähr die Hälfte der Spiele gegen Minimax gewonnen werden, jedoch ist das auf einem 3×3-Spielbrett auch nicht sehr schwer, wenn man den ersten Zug hat. Weiterhin ist die Implementierung selbst nach der Optimierung im dritten Feature-Modus nicht sehr effizient, was sich bereits auf dem 3×3-Spielbrett in der Trainingsdauer bemerkbar gemacht hat. Mehr Arbeit ist nötig, einen Feature-Modus zu entwerfen, der den Spielzustand genauer beschreibt und dessen Laufzeit besser skaliert, sodass auch ein Training auf größeren Spielbrettern möglich wird. Da der N-Tupel-Agent, der im folgenden Abschnitt beschrieben wird, deutlich erfolgreicher war, wurde jedoch mehr Zeit in diesen investiert.

5.5 Temporal Difference (N-Tupel)

Wie auch beim TD-Agenten müssen für die Verwendung des TDNT-Agenten im GBG-Framework ein paar Funktionen implementiert werden. Der Unterschied ist jedoch, dass die Features automatisch generiert werden, weshalb die Implementierung sehr viel einfacher ist und weniger Kreativität benötigt. Die zu implementierenden Funktionen beschreiben nur das Spielbrett, damit die automatische Auswahl der N-Tupel stattfinden kann. Implementiert wurden eine Reihe von Funktionen die für die Funktion des Random-Walk-Algorithmus notwendig sind, eine Funktion die vordefinierte N-Tupel festlegt und eine Funktion die zu einem beliebigen Zustand nach der Symmetrie von Hex äquivalente Zustände zurückgibt.

Als die ersten Versuche mit dem TDNT-Agenten durchgeführt wurden waren die Ergebnisse enttäuschend. Der Agent schien auch auf kleineren Spielbrettern keinen Lernfortschritt zu erzielen. Da der Erfolg des Trainings sehr stark von den gewählten Parametern abhängen kann, wurde zuerst vermutet, dass einfach noch keine passende Parameter-Konfiguration gefunden wurde. Es stellte sich jedoch heraus, dass dies an einem Fehler in der Implementierung der Spielfeldsymmetrien lag. Während der Implementierung wurde davon ausgegangen, dass das Spielbrett von Hex drei Symmetrien hat: Zwei Spiegelungs-symmetrien und eine Rotationssymmetrie. Betrachtet man die für die Repräsentation des Spielbretts gewählte Datenstruktur, kann dieser Fehler leicht passieren. Da das Spielbrett in einem zweidimensionalen Array gespeichert ist, wurde während der Überlegungen zu den Symmetrien das Nachbarschaftsverhältnis durch die hexagonale Geometrie der Felder nicht berücksichtigt. Nachdem die fehlerhaften Spiegelungssymmetrien entfernt wurden, konnte sofort ein Lernfortschritt festgestellt werden.

Für die Implementierung der vordefinierten Tupel wurden alle Diagonalen gewählt, die zwei zusammengehörende Spielbrettseiten verbinden. Zusätzlich wurde die gerade vertikale Linie in der Mitte des Spielbrettes, die zwei der Ecken verbindet, als Tupel gewählt. Auf dieser Linie befinden sich nur Felder, für die beim Eröffnungszug eine optimale Strategie existiert. Die gewählten Tupel skalieren problemlos mit der Spielbrettgröße. Es gibt immer genau $2 * n + 1$ Tupel mit der Länge n auf einem Spielbrett der Größe $n \times n$. Eine Veranschaulichung der gewählten Tupel ist in untenstehender Abbildung 22 zu sehen. Ob die Auswahl dieser Tupel effektiv ist, wird im Folgenden getestet.

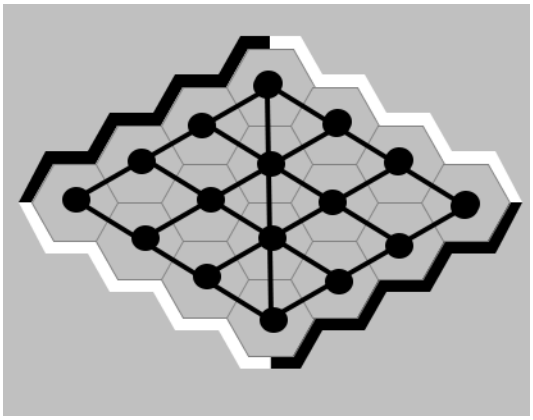

Abbildung 22: Veranschaulichung der festgelegten Tupel.
Die Punkte entsprechen den Abtastpunkten, die verbindenden Linien sind die Tupel. Jedes Tupel enthält exakt vier Abtastpunkte und verbindet jeweils in einer geraden Linie zwei Ränder des Spielbretts.

Wenn nicht anders angegeben, wurden beim Training folgende Parameter verwendet:

$$\alpha_{init} = \alpha_{final} = 0{,}001$$

$$\varepsilon_{init} = 0{,}3$$

$$\varepsilon_{final} = 0$$

$$\gamma = 1$$

$$\lambda = 0$$

2×2-Spielbrett

Auf einem Spielbrett der Größe 2×2 können nur 24 verschiedene Spielabläufe vorkommen, unter Berücksichtigung der Symmetrie sogar nur zwölf. Getestet wurden die vordefinierten Tupel und *Random Walk* mit jeweils zwei Tupel der Längen zwei und drei und einem Tupel der Länge vier. Auf dem 2×2-Spielbrett waren noch keine großen Unterschiede ersichtlich. In der Regel wurde zwischen 10 und 30 Spielen eine stabile optimale Strategie erlernt, je nachdem, wie effizient die zufällig gewählten Tupel waren. Mit nur 100 Trainingsspielen ist die Wahrscheinlichkeit sehr hoch, einen perfekt spielenden Agenten zu erhalten.

3×3-Spielbrett

Das nächst-größere Spielbrett mit der Größe 3×3 bereitet dem TDNT-Agenten ebenso keine Probleme. In der Regel sind 1.000 Trainingsspiele unter Verwendung der vordefinierten Tupel ausreichend, um im Anschluss jedes Spiel gegen den Minimax-Agenten zu gewinnen. Es gab jedoch auch Ausnahmefälle, wo eine Konvergenz auf das Optimum deutlich länger gedauert hat. Die Variation im Training hängt wahrscheinlich von den gewählten erzwungenen Zufallszügen ab. Weiterhin wurde *Random Walk* mit jeweils fünf Tupel der Längen drei und vier getestet, welche ebenfalls innerhalb von 1.000 Spielen eine perfekte Strategie erlernt haben.

4×4-Spielbrett

Da das 4×4-Spielbrett das größte ist, bei dem durch den Minimax Agenten noch ein objektiv gut spielender Agent verfügbar ist, wurde eine Vielzahl von Konfigurationen getestet. Die Ergebnisse werden nun im Folgenden dargestellt.

Vordefinierte Tupel

Zuerst wurde für das 4×4-Spielbrett versucht, einfach nur die Anzahl der Trainingsspiele im Vergleich zur vorherigen Spielbrettgröße um eine weitere Größenordnung zu erhöhen. Da das nicht ausreichte, wurde die Anzahl der Trainingsspiele noch weiter erhöht. Die Ergebnisse sehen wie folgt aus:

Trainingsspiele	Gewinnrate
10.000	63,2%
100.000	**80,2%**
1.000.000	**80,2%**

Tabelle 9: TDNT (vordefiniert) gegen Minimax (10.000 Evaluationsspiele, 4x4)

Ab einer gewissen Anzahl an Spielen, die zwischen 10.000 und 100.000 liegt, sorgen weitere Trainingsspiele zu keiner Verbesserung der Leistung des Agenten. Es ist möglich, dass der Agent mit den gewählten Tupel nicht in der Lage ist, wichtige Muster auf dem 4×4-Spielbrett zu erkennen. Deshalb wurde als nächstes der Random Walk verwendet, um die Tupel zu generieren. Die vordefinierten Tupel auf einem Spielbrett dieser Größe haben die Länge vier

und es kommen neun Tupel zum Einsatz. Damit ein direkter Vergleich stattfinden kann wurde für den Random Walk ebenfalls die Länge vier verwendet und zuerst nur die Anzahl der Tupel variiert. Die Zahl der Trainingsspiele betrug 10.000, wo die Gewinnrate der vordefinierten Tupel mit 63,2% noch stark verbesserungswürdig ist.

Random Walk bei verschiedener Tupel-Anzahl

Wie in untenstehender Abbildung 23 zu sehen ist, wurden die vordefinierten Tupel offenbar nicht gut gewählt. Die durch den Random Walk gewählten Tupel erreichten bei gleicher Trainingsdauer, gleicher Anzahl an Tupel und mit der gleichen Tupel-Länge eine sehr viel höhere Gewinnrate gegen Minimax. Schon ab vier zufällig gewählten Tupel wurde ein besseres Ergebnis gemessen. Zu beachten ist bei diesen Ergebnissen, dass die Auswahl der Tupel zufällig stattfindet und insbesondere bei einer geringen Anzahl an Tupel stark variieren kann, was an der hohen Standardabweichung gut zu beobachten ist. Deswegen wurde der Agent jeweils fünf Mal trainiert und die durchschnittliche Leistung der Agenten nach dem Training berechnet.

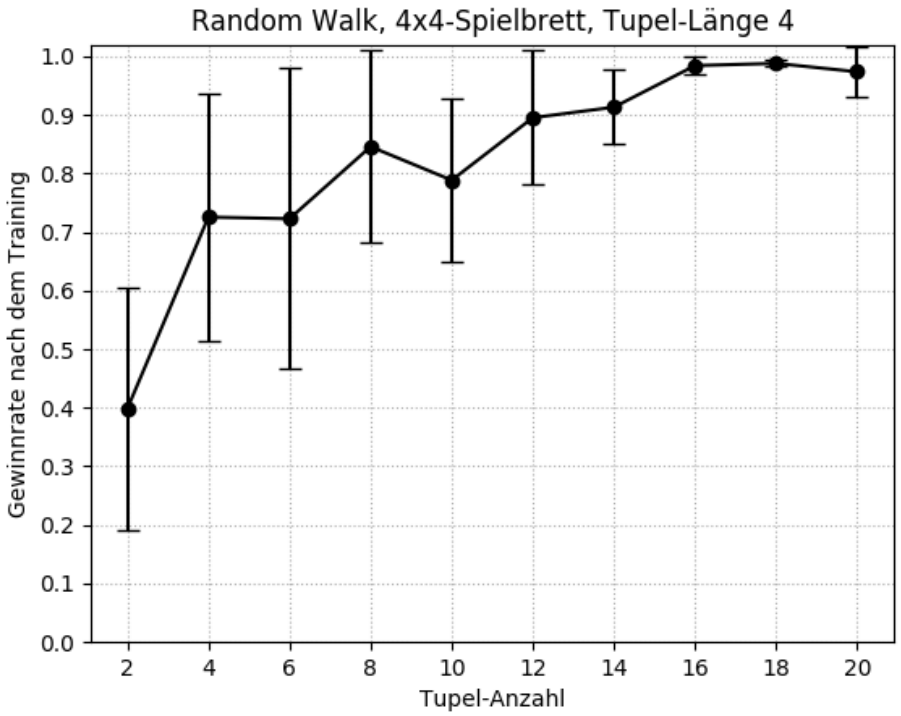

Abbildung 23: Ergebnisse des TDNT-Agenten gegen Minimax in Abhängigkeit von der Tupel-Anzahl. Ergebnisse jeweils nach 10.000 Trainingsspielen. Es wurden für jeden Schritt fünf Agenten trainiert, im Graphen zu sehen ist die durchschnittliche Leistung der fünf Agenten und die jeweilige Standardabweichung.

Random Walk bei verschiedener Tupel-Länge

Als nächstes wurde getestet, welchen Einfluss die Länge der Tupel hat. Damit die Ergebnisse möglichst wenig durch eine mögliche schlechte Wahl der Tupel beeinflusst werden, wurde eine hohe Tupel-Anzahl von 50 gewählt. In Abbildung 24 sind die Trainingsverläufe für Agenten verschiedener Tupel-Längen zwischen zwei und zehn aufgeführt. Eine Evaluation

der Leistung fand alle 100 Trainingsspiele statt, wo jeweils 1.000 Evaluationsspiele gegen Minimax gespielt wurden.

Die Agenten mit Tupel-Längen zwischen drei und fünf konvergierten auf eine optimale Spielweise. Der Tupel mit nur zwei Abtastpunkten pro Tupel erreichte am Ende eine Gewinnrate von knapp über 70%. Erstaunlicherweise reichten schon drei Abtastpunkte aus, um eine optimale Strategie zu finden. Drei Abtastpunkte sind noch nicht genug, um virtuelle Verbindungen erkennen und von schwachen Verbindungen unterscheiden zu können, denn dafür werden mindestens vier Abtastpunkte benötigt. Zwischen den Tupel-Längen drei bis fünf gab es insgesamt wenige Unterschiede. Es wurde eine sehr ähnliche Lerngeschwindigkeit in Bezug zu den absolvierten Trainingsspielen festgestellt. Bei der benötigten Zeit der Spiele wurde subjektiv ein leichter Anstieg der Trainingsdauer mit steigender-Tupel-Länge bemerkt, der sich beim Agenten mit einer Tupel-Länge von zehn deutlich bestätigt hat. Das Training dieses Agenten hat am längsten gedauert und er hat sich außerdem nicht zuverlässig in jedem Trainingsdurchlauf dem Optimum angenähert.

Für alle weiteren Tests wird aufgrund dieser Ergebnisse eine Tupel-Länge von vier gewählt. Diese Tupel-Länge konnte zu Beginn des Trainings den schnellsten Lernerfolg vorweisen und ist gleichzeitig schneller in der Berechnung als die Tupel der Länge fünf. Der Agent mit Tupel-Länge drei hatte ein paar bedenkliche Ausreißer und zu Beginn des Trainings einen langsamen Lernfortschritt.

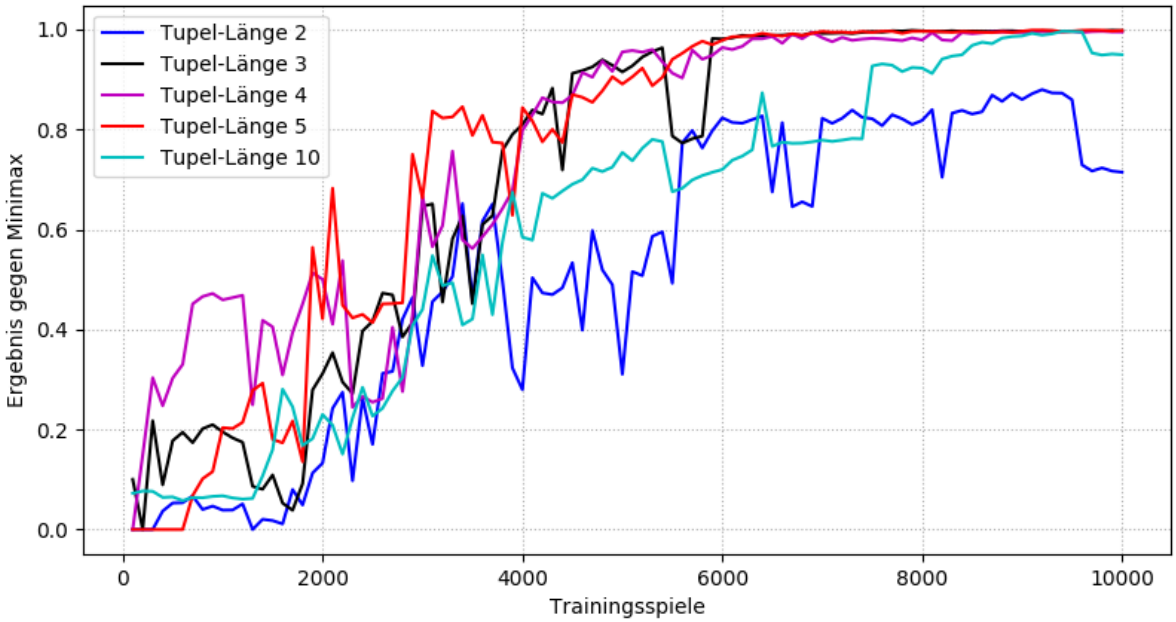

Abbildung 24: Trainingsverlauf von fünf TDNT-Agenten mit unterschiedlicher Tupel-Länge. Evaluiert wurde alle 100 Trainingsspiele mit jeweils 1.000 Evaluationsspielen gegen Minimax. Durchschnitt aus je fünf Trainingsdurchläufen.

Auswirkung der Symmetrie auf die Lerngeschwindigkeit

Alle bisherigen Experimente wurden unter Verwendung des Wissens über die Symmetrie des Spielbretts durchgeführt. Im nächsten Schritt wurde bestimmt, wie groß die

Auswirkungen der Symmetrie auf die Lerngeschwindigkeit sind. Als Basis diente dafür ein *Random Walk* mit 50 Tupel der Länge vier. Für diesen Test wurden 20.000 Trainingsspiele durchgeführt, da vermutet wurde, dass der Agent ohne die Symmetrie länger für ein erfolgreiches Training brauchen wird. Jeder Verlauf wurde 25-mal gemessen und daraus wurde der Durchschnitt gebildet. Das Ergebnis des Tests ist in Abbildung 25 zu sehen. Unter Verwendung der Spielbrettsymmetrie wurde konsistent ein besserer Lernerfolg gemessen und in der Regel auch das Optimum erreicht. Wahrscheinlich lässt sich auch ohne die Verwendung der Symmetrien eine perfekte Strategie erlernen, jedoch werden dafür mit den verwendeten Parametern mehr als 20.000 Trainingsspiele benötigt. Der Unterschied ist insgesamt jedoch geringer als zu erwarten war. Der Grund dafür könnte sein, dass mit 50 eine vergleichsweise große Anzahl an Tupel gewählt wurde. Auch interessant ist, dass in diesem Test eine langsamere Konvergenz auf eine perfekte Strategie gemessen wurde, als im letzten Test. In Abbildung 24 benötigte der Agent mit 50 Tupel der Länge vier mit Verwendung der Symmetrie ca. 6.000 Spiele, bevor die optimale Strategie gefunden wurde. In diesem Test lag die Leistung im Durchschnitt nach 6.000 Spielen bei ca. 80%. Während die Lernschrittweite α während des Trainings konstant blieb, variierte jedoch der Zufallszug-Parameter ε von 0,3 beim Start des Trainings zu 0 gegen Ende des Trainings. Der aufgrund der doppelt so langen Trainingsdauer verlangsamte Abstieg dieses Parameters könnte eine mögliche Ursache dafür sein.

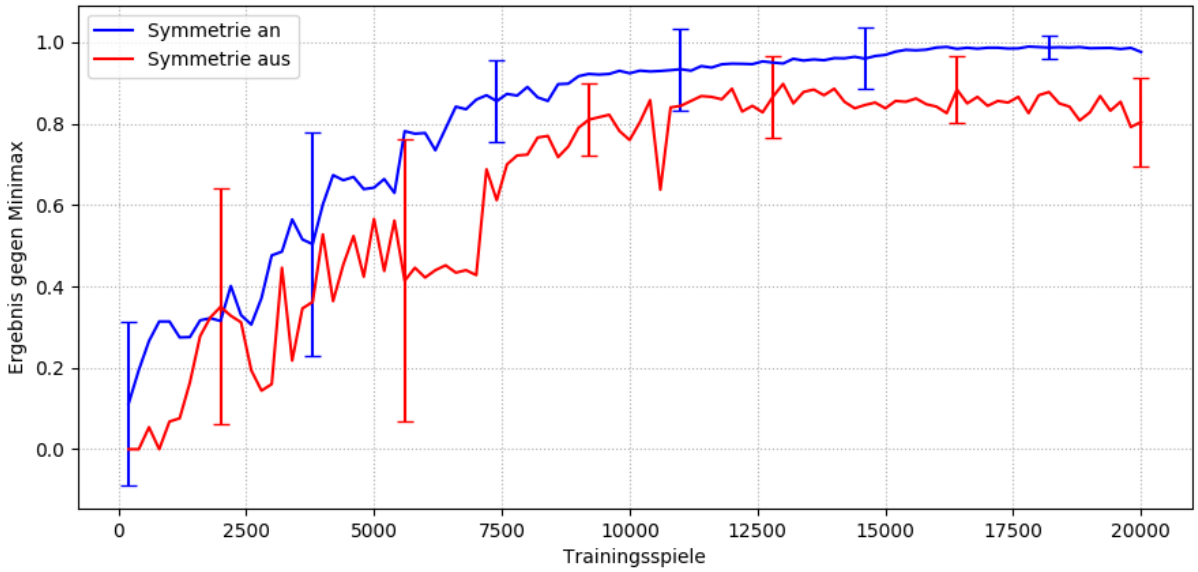

Abbildung 25: Trainingsverlauf mit und ohne Verwendung der Spielbrettsymmetrie.
Es wurden jeweils 25 Messungen durchgeführt; Im Graphen zu sehen ist der Durchschnitt und die Standardabweichung

Variation der Zufallszugwahrscheinlichkeit ε

Aufgrund des langsameren Lernfortschritts im letzten Test wurde im Weiteren analysiert, welchen Einfluss der Zufallszug-Parameter ε hat. Dafür wurde der Trainingsverlauf mehrerer Konfiguration jeweils zehn Mal gemessen und der Durchschnitt gebildet. Erneut wurden 50 Tupel der Länge vier mit aktivierter Symmetrie verwendet. Das Ergebnis, welches in Abbildung 26 zu sehen ist, ist erstaunlich. Jede der getesteten Konfigurationen hatte nach den 20.000 Trainingsspielen eine durchschnittliche Erfolgsquote gegen Minimax von über

94%. Das trifft selbst auf den Agenten zu, der mit $\varepsilon_{init} = \varepsilon_{final} = 1$ trainiert wurde und damit während des Trainings ausschließlich zufällige Züge gewählt hat, als auch auf den Agenten der komplett ohne Zufallszüge trainiert hat. Ohne erzwungene Zufallszüge zu trainieren bedeutet, dass die Exploration von wenig erforschten Zuständen allein über die Aktualisierung der Gewichtungen stattfindet. Weiterhin ist bemerkenswert, dass zu Beginn eine geringe Zufallszugwahrscheinlichkeit effektiver zu sein scheint, während der einzige Agent der zuverlässig eine perfekte Strategie erlernt hat derjenige war, der am Ende des Trainings noch eine Zufallszugwahrscheinlichkeit von 30% hat. Es ist vorstellbar, dass das Ergebnis auf größeren Spielbrettern anders aussehen würde. Ab einer gewissen Spielbrettgröße wäre es fast undenkbar, dass ein Agent, der beim Training vollkommen zufällig spielt, in einem akzeptablen Zeitraum auf eine nahezu optimale Strategie stößt.

Die Hypothese, dass der beim Vergleich der Symmetrie beobachtete langsamere Trainingsfortschritt an einer langsamer fallenden Zufallszugwahrscheinlichkeit gelegen hat, scheint sich bestätigt zu haben. Niedrigere Werte für ε als der vorher verwendete Startwert von 0,3 scheinen den Trainingserfolg zu beschleunigen. Jedoch wurde auch hier mit keiner Konfiguration ein ähnlich gutes Ergebnis nach 6.000 Spielen erreicht, wie in Abbildung 24. Dieser Umstand wird später noch genauer betrachtet.

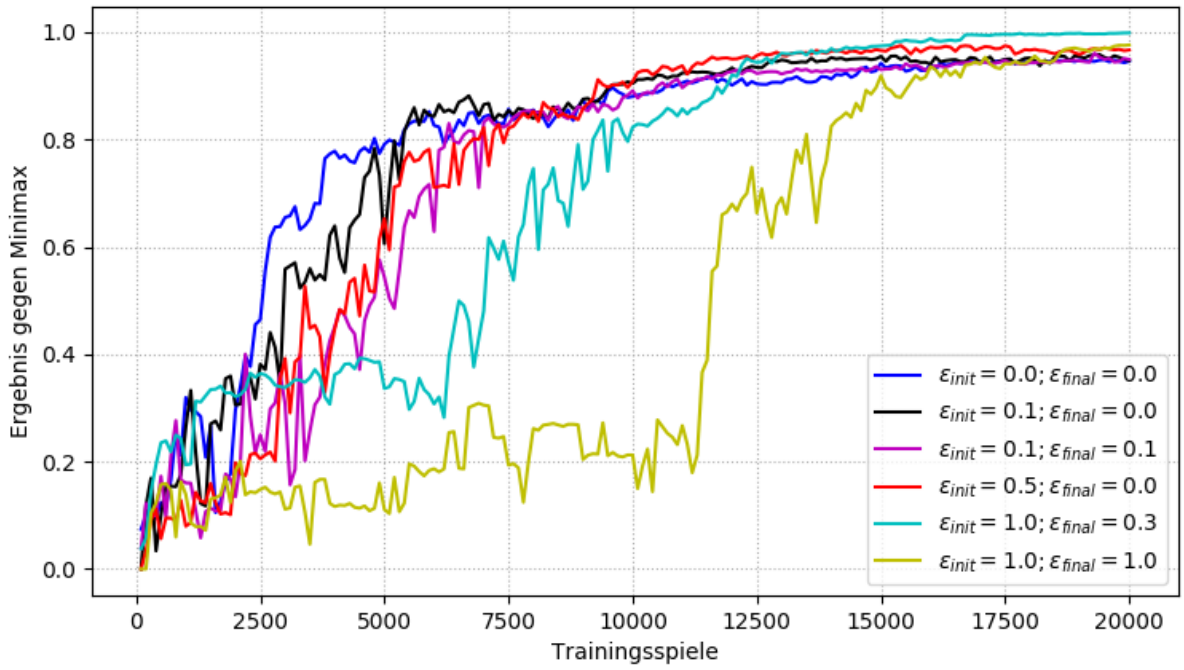

Abbildung 26: Trainingsverlauf des TDNT-Agenten mit verschiedenen Werten für den Parameter ε. Durchschnitt aus jeweils zehn Trainingsverläufen.

Variation der Lernschrittweite α

Den wohl direktesten Einfluss auf den Erfolg und die Geschwindigkeit des Trainings hat die Lernschrittweite α. Für diesen Test wurden erneut 50 Tupel der Länge vier und außerdem eine Zufallszugwahrscheinlichkeit von $\varepsilon_{init} = 0.1$ und $\varepsilon_{final} = 0$ mit aktivierter Symmetrie verwendet. Um den effektiven Bereich für α zu bestimmen, wurden Messungen mit festem

α im Bereich 10^{-1} bis 10^{-5} durchgeführt, deren Ergebnis in Abbildung 27 zu sehen ist. Die beiden Extremwerte schnitten dabei am schlechtesten ab, während die Werte 10^{-2} bis 10^{-4} nach 20.000 Trainingsspielen eine hohe Gewinnrate von mindestens 90% gegen Minimax erreichten. Weiterhin wurde eine von 10^{-2} auf 10^{-3} sinkende, sowie eine von 10^{-4} auf 10^{-3} steigende Schrittweite getestet. Von diesen schnitt der Agent mit der sinkenden Schrittweite am besten ab und erreichte schon nach ca. 3.000 Trainingsspielen das erste Mal eine Gewinnrate von durchschnittlich 80%. Weiterhin ist dies die einzige Konfiguration, die in allen Testläufen eine stabile perfekte Strategie erlernt hat und stellt damit eine Verbesserung gegenüber der vorherigen verwendeten Konfiguration von $\alpha_{init} = \alpha_{final} = 10^{-3}$ dar.

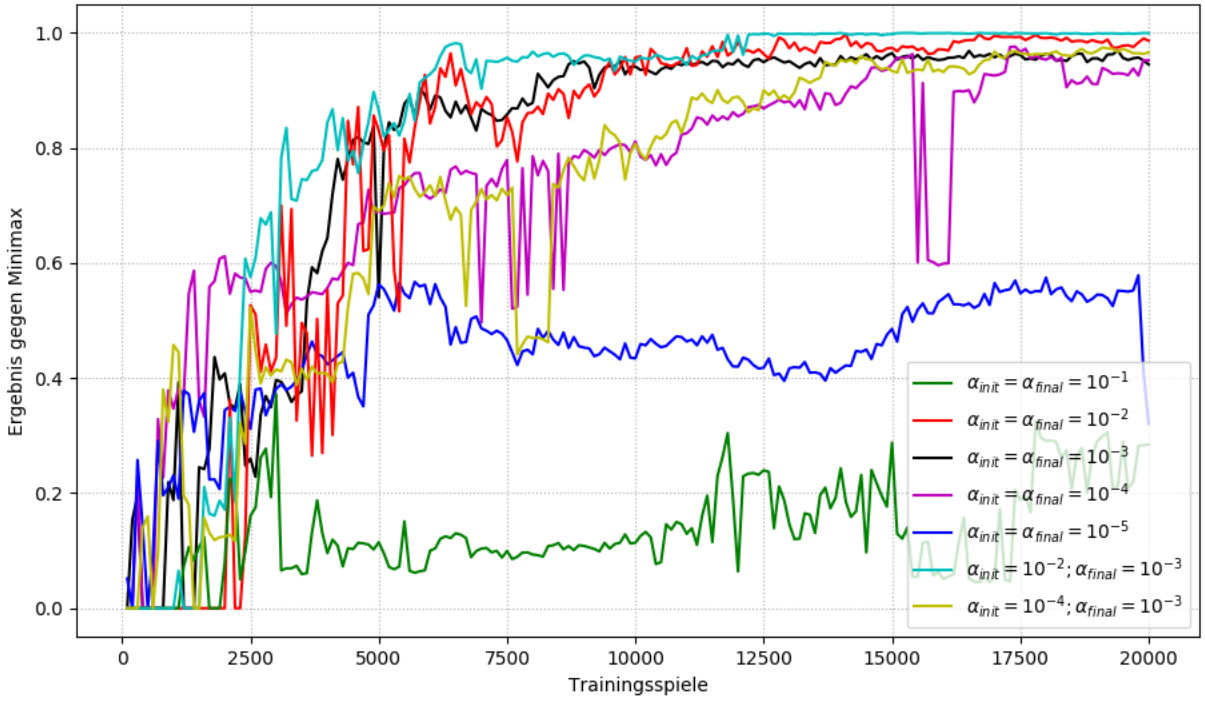

Abbildung 27: Trainingsverlauf des TDNT-Agenten mit unterschiedlicher Lernschrittweite. Durchschnitt aus drei Messungen.

Verwendung von *eligibility traces*

Der letzte Parameter, der noch zu testen ist, ist λ. Da die *eligibility traces* das Training beschleunigen sollten, wurde eine Trainingsdauer von 10.000 Spielen gewählt. Erneut wurden 50 Tupel der Länge vier mit aktivierter Symmetrie verwendet. Die restlichen Parameter wurden wie folgt gewählt: $\alpha_{init} = 10^{-2}$, $\alpha_{final} = 10^{-3}$, $\varepsilon_{init} = 0.1$, $\varepsilon_{final} = 0$. Das Ergebnis in Abbildung 28 zeigt, dass die Verwendung von *eligibility traces* das Training nicht beschleunigt und stattdessen anscheinend sogar verlangsamt. Der Agent ohne *eligibility traces* ($\lambda = 0$) hatte aus den jeweils fünf Messungen im Durchschnitt den schnellsten Lernerfolg und erreichte eine nahezu optimale Strategie mit einer durchschnittlichen Gewinnrate von über 99% in knapp über 5.000 Trainingsspielen, während unter Verwendung von $\lambda = 0{,}1$ oder höher mindestens 7.000 Trainingsspiele benötigt wurden. Es scheint also, als würde das direkte Anwenden der Belohnung/Bestrafung auf

frühere Zustände auf dem 4×4-Spielbrett von Hex nicht sinnvoll sein, selbst wenn diese abgeschwächt werden. Der zugehörige Graph ist in Abbildung 28 zu sehen.

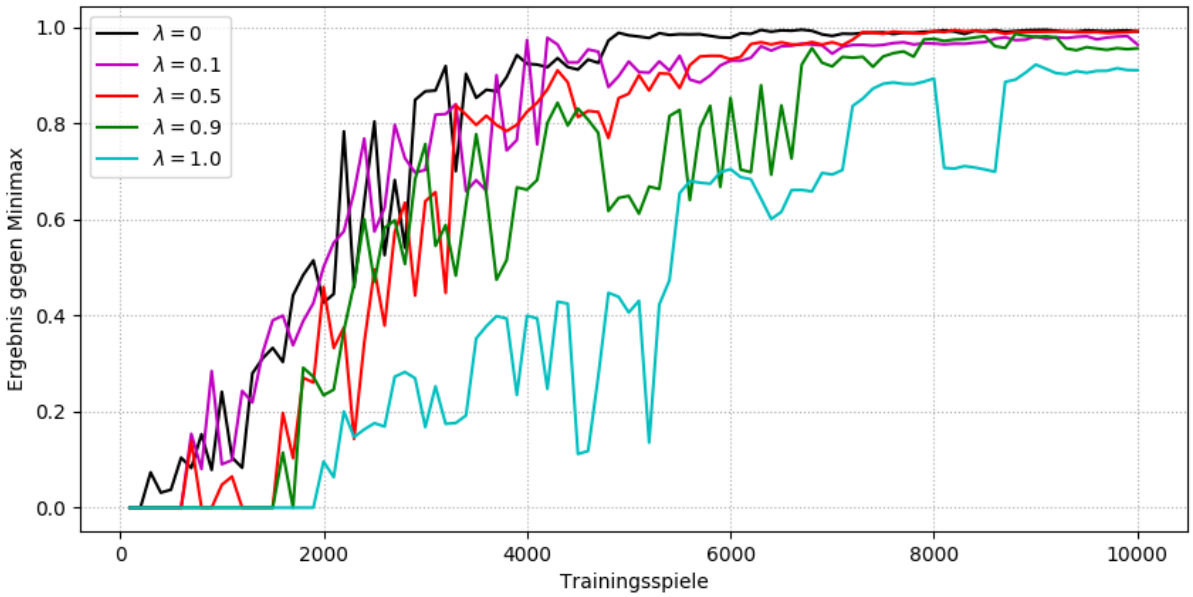

Abbildung 28: Auswirkungen von eligibility traces auf den Trainingsverlauf des TDNT-Agenten. Durchschnitt aus jeweils fünf Messungen.

Einfluss der Trainingsdauer auf die Lerngeschwindigkeit

Wie bereits erwähnt schein es in Abbildung 24 und Abbildung 25 einen Unterschied in der Lerngeschwindigkeit gegeben zu haben, der sich im darauf folgenden Test des Parameter ε noch nicht eindeutig erklären ließ. In einem weiteren Test wurden drei TDNT-Agenten mit 50 Tupel der Länge vier, aktivierter Symmetrie, fester Lernschrittweite von $\alpha = 0{,}001$, ohne erzwungene Zufallszüge und ohne *eligibility trace*s verglichen. Der einzige Unterschied in der Konfiguration war die Anzahl der Trainingsspiele. Da es keine variablen Parameter gab, wurde erwartet, dass die Kurve des Agenten mit der längsten Trainingsdauer nur eine Fortführung der anderen Agenten auf dem Graphen darstellt. Um den Einfluss von Messungenauigkeiten zu reduzieren wurden alle Agenten 50-mal trainiert und der Durchschnitt gebildet. Wie in Abbildung 29 zu sehen ist, gibt es anscheinend einen Einfluss der Trainingsdauer auf die Geschwindigkeit der Konvergenz auf eine nahezu optimale Strategie, die aus den Parametern nicht offensichtlich wird. Eine kürzere Trainingsdauer bedeutet anscheinend ein schnelleres Lernen, jedoch gibt es bei zu geringer Trainingsdauer mehr Varianz in der zu erwartenden Leistung nach dem Training. Ein längeres Training kostet zwar mehr Zeit, erhöht aber die Wahrscheinlichkeit auf eine erfolgreiche Konvergenz auf eine nahezu optimale Strategie. Was genau den Unterschied in der Lerngeschwindigkeit verursacht konnte während des Bearbeitungszeitraums dieser Arbeit nicht ermittelt werden.

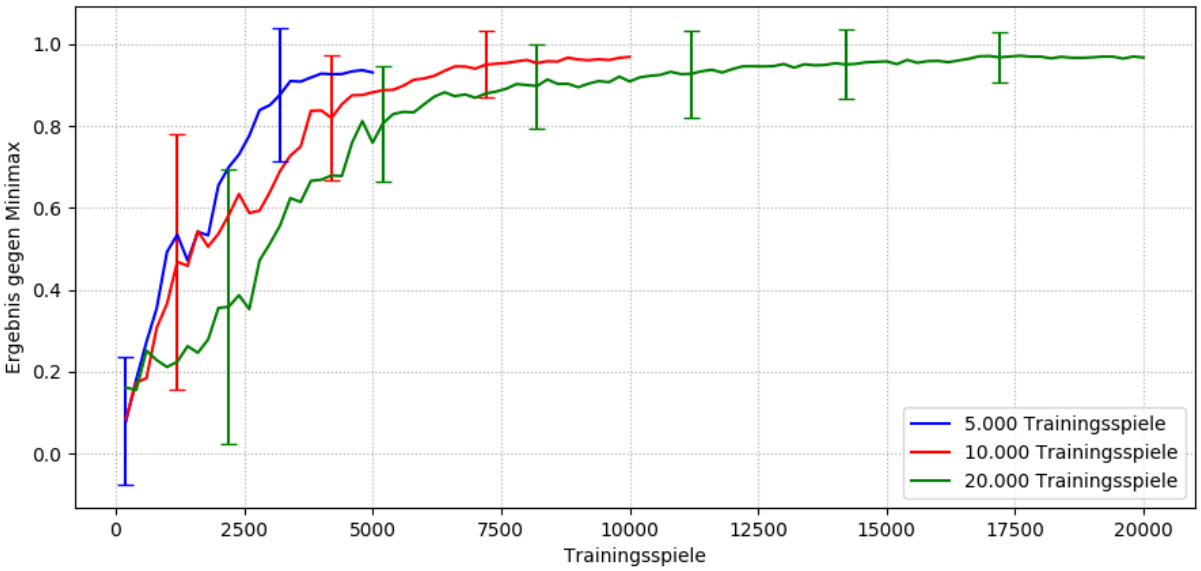

Abbildung 29: Einfluss der Trainigsdauer auf die Lerngeschwindigkeit

5×5-Spielbrett

Aufbauend auf den Ergebnissen vom 4x4-Spielbrett wurde im Anschluss versucht, für größere Spielbretter als 4×4 einen guten Agenten zu trainieren. Erstmalig wird auch die Evaluation der trainierten Agenten zu einer Herausforderung, da der Minimax-Agent nicht mehr zur Verfügung steht. Aus diesem Grund ist es nicht mehr möglich, einen aussagekräftigen Graphen des Trainingsverlaufs zu erstellen. Während jedes Trainingsverlaufs, der in den Graphen abgebildet wurde, wurden 100 Evaluationen in regelmäßigen Abständen durchgeführt, die jeweils 1.000 Spiele gegen Minimax beinhalteten. Dies ist nur möglich, weil der Minimax-Agent einen einmal aufgebauten Suchbaum für folgende Evaluationen wiederverwendet. Der MCTS-Agent führt seine Simulationsschritte für jeden Zug erneut durch, was ihn während der Evaluation sehr viel langsamer macht. Deshalb wird die Leistung der folgenden Agenten nur als Gewinnrate nach dem Training angegeben.

Da bei den Tests auf dem 4×4-Spielbrett mit 50 eine wahrscheinlich übermäßig große Anzahl an Tupel verwendet wurde, wurde zuerst getestet, wie gut ein Agent mit ebenfalls 50 Tupel der Länge vier auf dem 5×5-Spielbrett zu spielen lernt. Die sonstigen Parameter wurden wie folgt gewählt:

- Die Lernschrittweite betrug $\alpha_{init} = 0{,}01$ und $\alpha_{final} = 0{,}001$, welche auf dem 4×4-Spielbrett den besten Lernerfolg vorweisen konnte.
- Die Zufallszug-Wahrscheinlichkeit wurde auf $\varepsilon_{init} = 0{,}1$ und $\varepsilon_{final} = 0$ gesetzt. Während eine höhere Wahrscheinlichkeit gegen Ende des Trainings einen höheren Erfolg zeigte, ist jedoch fraglich, ob dies auch für Spielbretter mit größerem Zustandsraum gilt.
- Die Symmetrie wurde berücksichtigt.
- Eligibility traces wurden nicht eingesetzt ($\lambda = 0$).

Um eine ungefähre Abschätzung dafür zu bekommen, wie groß die Anzahl an Trainingsspielen sein muss, wurde deren Anzahl variiert, wie in folgender Tabelle 10 zu sehen ist:

Trainingsspiele	Gewinnrate (MCTS)	Schlägt Hexy
10.000	0%	Nein
20.000	50%	Nein
50.000	**100%**	**Ja**

Tabelle 10: Trainingsergebnis eines TDNT-Agenten auf dem 5x5-Spielbrett. Zehn Evaluationsspiele gegen MCTS mit 100.000 Iterationen.

Im Vergleich gegen MCTS wurden zehn Spiele mit 100.000 Iterationen pro Zug gespielt. Die genauen prozentuellen Angaben sind daher definitiv nicht aussagekräftig. Dass der TDNT-Agent nach 50.000 Trainingsspielen jedoch alle zehn gespielten Spiele gegen den MCTS-Agenten, sowie das Spiel gegen Hexy gewonnen hat, ist ein gutes Indiz dafür, dass eine gute Strategie erlernt wurde. Bei den Spielen gegen Hexy war interessant, dass die ersten paar Züge für alle TDNT-Agenten absolut identisch verliefen. Im Falle des TDNT-Agenten mit 20.000 Trainingsspielen waren die ersten fünf Züge im Vergleich mit dem Agenten mit 50.000 Trainingsspielen identisch. Unterschiede in der Spielweise waren erst im späteren Spielverlauf zu sehen. Die Bewertung der Eröffnungszüge des Agenten nach 50.000 Trainingsspielen ist in untenstehender Abbildung 30 zu sehen. In Abbildung 31 folgt die terminale Spielstellung des Spiels gegen Hexy.

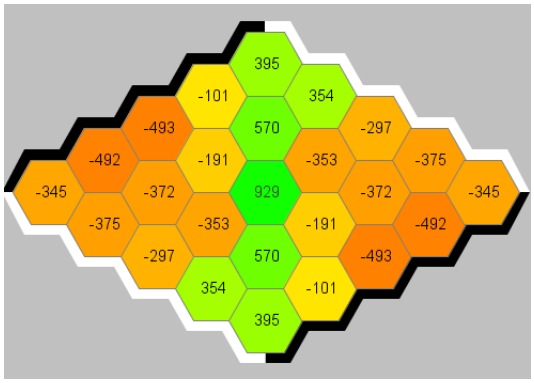

Abbildung 30: Bewertung der Eröffnungszüge des TDNT-Agenten nach 50.000 Trainingsspielen

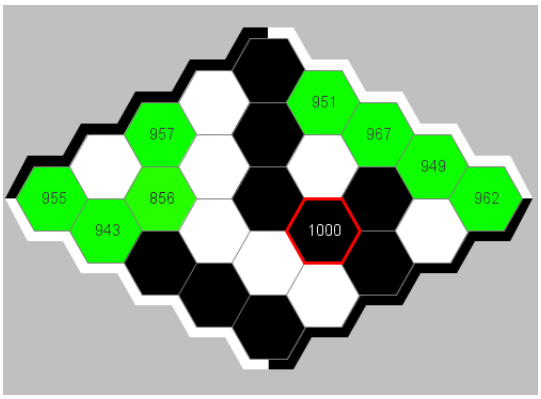

Abbildung 31: Endzustand des TDNT-Agenten (schwarz) gegen Hexy (weiß) auf dem 5x5-Spielbrett

6×6-Spielbrett

Für das 6×6-Spielbrett wurde zuerst gemessen, wie weit der TDNT-Agent auf dem größeren Spielbrett kommt, wenn er weiterhin nur 50.000 Trainingsspiele bei gleichen Parametern wie auf dem 5×5-Spielbrett durchführt. Eine genaue Auflistung der Parameter ist im vorherigen Abschnitt zu finden. Vergleiche gegen MCTS und Hexy haben schnell gezeigt, dass 50.000 Trainingsspiele nicht annähernd ausreichen. Daraufhin wurde die Anzahl der Trainingsspiele auf 200.000 Spiele vervierfacht, was ebenfalls keine guten Ergebnisse brachte. Selbst nach 500.000 Trainingsspielen spielte der TDNT-Agent nicht gut genug, um Hexy oder MCTS zu besiegen. Der Grund dafür könnte sein, dass die weiterhin verwendeten 50 Tupel der Länge vier nicht mehr ausreichten, um genügend Muster auf dem Spielbrett zu erkennen.

Um diese Hypothese zu testen wurde die Anzahl der Tupel verfünffacht während alle anderen Parameter gleichgeblieben sind. Die Trainingszeit hat sich dabei mehr als verdoppelt, die Leistung des Agenten war jedoch überzeugend. Tabelle 11 fasst die Ergebnisse zusammen.

Trainingsspiele	Tupel-Anzahl	Schlägt MCTS	Schlägt Hexy	Trainingsdauer
50.000	50	Nein	Nein	< 5 Minuten
200.000	50	Nein	Nein	ca. 10 Minuten
500.000	50	Nein	Nein	ca. 20 Minuten
500.000	250	**Ja**	**Ja**	ca. 50 Minuten

Tabelle 11: Trainingsergebnisse des TDNT-Agenten auf dem 6x6-Spielbrett

Sowohl Hexy als auch der MCTS-Agent mit 1.000.000 Iterationen und unbegrenzter Suchbaum-Höhe konnten von dem Agenten besiegt werden. Die terminale Spielstellung ist in Abbildung 32 zu sehen.

Sowohl in Abbildung 32 als auch in Abbildung 33 ist auffällig, dass die Feldbewertungen sehr nah an den Extremwerten sind. In Abbildung 32 ist keine Unterscheidung der einzelnen Felder mehr erkennbar, wenn die reellen Bewertungen aus dem Intervall [-1, +1] als ganze Zahlen im Bereich [-1.000, +1.000] dargestellt werden. Ebenso sind die Bewertungen der Eröffnungszüge sehr ausgeprägt. Es ist möglich, dass durch die erhöhte Anzahl an Eingabedaten eine geringere Lernschrittweite verwendet werden sollte. Weiterhin ist es möglich, dass 500.000 Trainingsspiele deutlich zu hoch gewählt sind, wenn 250 Tupel zum Einsatz kommen.

Die genaue Ursache wurde jedoch auf dem 6×6-Spielbrett nicht weiter untersucht. Diese Überlegungen flossen jedoch in das Training eines Agenten für das nächst-größere Spielbrett ein.

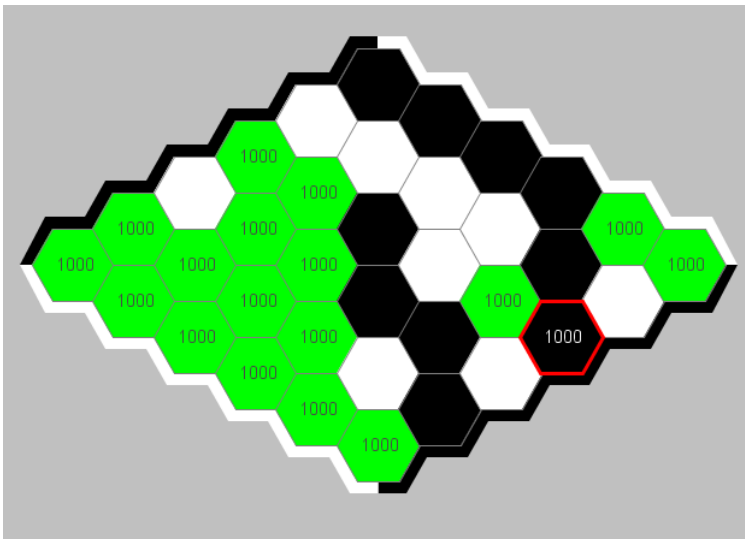

Abbildung 32: Terminale Spielstellung des TDNT-Agenten (schwarz) gegen Hexy (weiß) auf dem 6x6-Brett

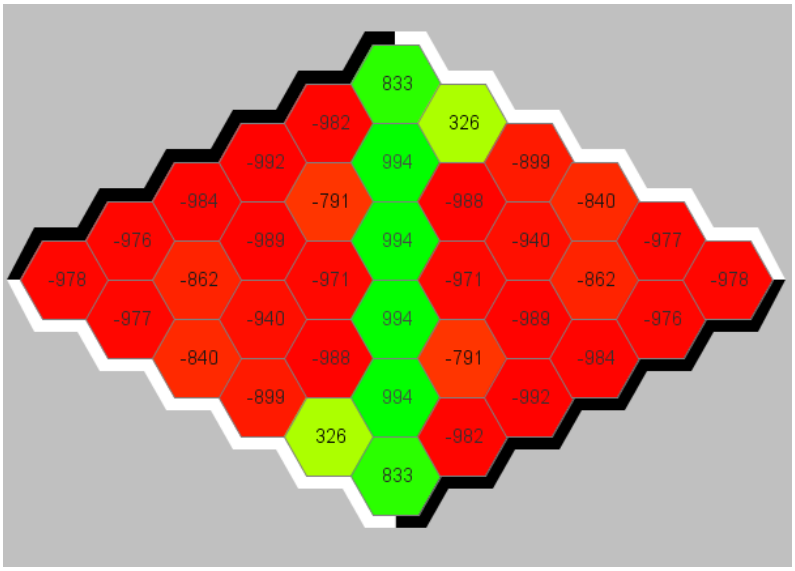

Abbildung 33: Bewertung der 6x6-Eröffnungszüge durch TDNT

7×7-Spielbrett

Aufgrund der Beobachtungen vom 6×6-Spielbrett wurde die Anzahl der Trainingsspiele auf dem 7×7-Spielbrett nicht erhöht. Die Lernschrittweite wurde jedoch auf $\alpha_{init} = 0{,}005$ und $\alpha_{final} = 0{,}0005$ halbiert. Erneut wurden 250 Tupel der Länge vier verwendet. Die restlichen Parameter stimmen mit denen aus dem Abschnitt des 5×5-Spielbretts überein. Das reduzieren der Lernschrittweite hat möglicherweise die Bewertungen im späteren Spielverlauf leicht verbessert. Zwar sind die Bewertungen immer noch sehr nah am Extremwert, jedoch lässt sich zumindest eine Unterscheidung der Felder feststellen. Wie in Abbildung 34 in der terminalen Spielstellung zu sehen ist, befinden sich die Feldbewertungen im Intervall [994, 1000].

Das Training dieses Agenten dauerte mit ca. 95 Minuten fast doppelt so lang, wie das Training des Agenten auf dem 6×6-Spielbrett, obwohl sich nur die Lernschrittweite verändert hat. Dies kann wahrscheinlich dem größeren Spielbrett zugeschrieben werden, durch das sich die durchschnittliche Anzahl der Züge pro Trainingsspiel erhöht. Der Agent war nach dem Training in der Lage, Hexy zu besiegen. Gegen MCTS konnte nicht mehr getestet werden. Selbst wenn die Suchbaum-Höhe auf fünf begrenzt wurde, erreichte der MCTS-Agent die Arbeitsspeicher-Grenze des verwendeten Systems.

Die Bewertungen der Eröffnungszüge durch diesen TDNT-Agenten sind in Abbildung 34 zu sehen.

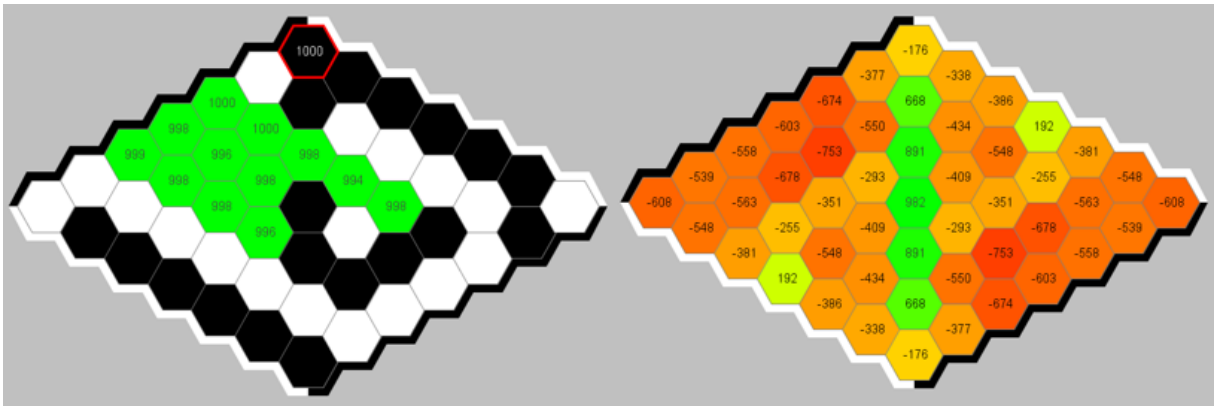

Abbildung 34: Terminale Spielstellung auf dem 7x7-Spielbrett (links), Bewertung der 7x7-Eröffnungszüge durch TDNT (rechts)

8×8-Spielbrett

Auf dem 8×8-Spielbrett ist es nicht gelungen, einen TDNT-Agenten zu trainieren, der Hexy besiegt. Die höchste getestete Konfiguration verwendete 400 Tupel der Länge vier mit 1 Mio. Trainingsspielen. Die restlichen Parameter gleichen denen des 7×7-Agenten. Schon nach einigen Zügen zeigte sich erneut das Problem, dass alle Felder die Bewertung 1.000 hatten. Um zu testen, ob die Anfangszüge des TDNT-Agenten gut gewählt waren, wurde der sogenannte Demo-Modus des Hexy-Programms verwendet. Mit diesem lässt sich ein angefangenes Spiel von Hexy zu Ende spielen. Bis zu 15 Züge können vom TDNT-Agenten gewählt werden, damit Hexy im Anschluss noch für ihn gewinnen kann. Die Bewertung der Eröffnungszüge durch diesen TDNT-Agenten ist in Abbildung 35 zu sehen.

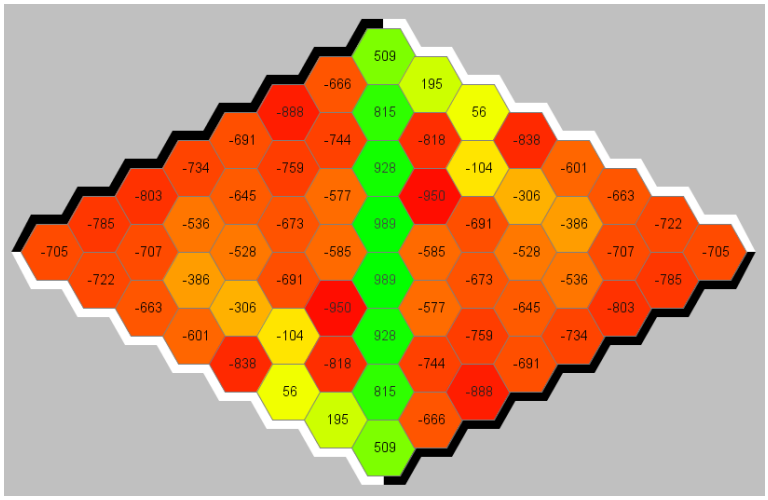

Abbildung 35: Bewertung der 8x8-Eröffnungszüge durch TDNT

6. Diskussion

Mit dem MCTS- und auch dem TDNT-Agenten wurden zwei KI-Verfahren gefunden, die sehr gut auf das Spiel Hex anwendbar sind.

Besonders interessant ist das Ergebnis des N-Tupel-Systems. Wie bereits in Kapitel 2 erwähnt, ist dies möglicherweise die erste Arbeit, die die Anwendung eines N-Tupel-Systems auf Hex dokumentiert. Nachdem der Einfluss der verschiedenen Parameter des TD-Algorithmus bestimmt wurde, ist es gelungen, für alle Spielbrettgrößen bis einschließlich 7×7 Agenten zu trainieren, die mit Hexy den Weltmeister der *Computer Games Olympiad* 2000 besiegen konnten. Kahl et al. trainierten ebenfalls mit TDL einen Agenten, der eine hohe Spielstärke erreichte und *Six* besiegte [14]. Dafür verwendeten sie jedoch die Feldbewertung von *Six* für das Training und damit einen Lehrer, der das Spiel bereits beherrschte. Das in dieser Arbeit verwendete N-Tupel-System war in der Lage, ein ähnlich gutes Ergebnis zu erreichen, ohne dass ein Lehrersignal verwendet wurde. Zwar ist *Hexy* ein schwächerer Spieler als *Six* und damit einfacher zu besiegen, jedoch konvergiert die Leistung des TDNT-Agenten nach genügend Trainingsspielen in der Regel auf eine Leistung nahe des Optimums. Der trainierte TDNT-Agent wurde nicht gegen *Six* getestet, es ist jedoch vorstellbar, dass im Falle einer Niederlage ein etwas längeres Training ausreichend wäre, um auch *Six* zu besiegen.

Vorteilhaft ist auch, dass der TDNT-Agent nach seinem Training eine praktisch sofortige Entscheidung bezüglich des nächsten Zuges treffen kann. Agenten wie MCTS oder Heuristiken, die nach Mustern suchen, benötigen zum Teil relativ lange, um einen Zug zu tätigen. Untersuchungen am TDNT-Agenten haben weiterhin gezeigt, dass der Agent schon früh gute Anfangszüge findet. Ein langes Training ist nötig, um auch die späteren Züge, tief im Spielbaum, zu erlernen. Eine mögliche Strategie, sich die geringe benötigte Zeit pro Zug zu Nutze zu machen, wäre im späteren Spielverlauf zusätzlich das MCTS-Verfahren einzusetzen, um auf der reduzierten verbleibenden Zustandsmenge Simulationen der verbleibenden Züge durchzuführen. In Kombination beider Techniken wäre es vorstellbar, einen noch stärkeren Agenten zu erhalten, der weiterhin kein spielspezifisches Wissen verwendet.

Ein anderer Ansatz, die Leistung des TDNT-Agenten weiter zu verbessern, wäre das Finden optimaler vordefinierter Tupel für Hex. Dabei wird jedoch indirekt Vorwissen über das Spiel vermittelt, sodass die Ergebnisse nicht oder nur begrenzt auf andere Anwendungszwecke übertragbar wären. Weiterhin wurden in dieser Arbeit nur N-Tupel-Systeme mit Tupel gleichen Länge verwendet. Es könnte jedoch auch sein, dass ein Tupel-System mit Tupel unterschiedlicher Länge noch bessere Ergebnisse erzielen können.

Mehr Arbeit ist nötig, um eine Feature-Kombination zu finden, mit der der TD-Agent auf Hex anwendbar ist. Die rohen Spielbrettinformationen, die am ehesten dem „General Board Game Playing“-Paradigma entsprechen, sind nicht hinreichend, um die Spielfunktion zu erlernen. Wenn dem TD-Agenten Vorwissen über das Spiel in Form geeigneter Features vermittelt wird, könnte der Agent vermutlich eine überzeugende Leistung erreichen. Zwar

ist dies in der Bearbeitungszeit dieser Arbeit nicht gelungen, jedoch wäre dies ein interessantes Thema für zukünftige Arbeiten.

Die Monte-Carlo Tree Search, die von einigen der weltbesten Hex-Agenten erfolgreich verwendet wird, ist in der reinen Form, ohne Vorwissen über das Spiel, relativ begrenzt einzusetzen. Dies äußert sich sowohl in der schnell ansteigenden benötigten Zeit pro Zug, als auch im Speicherverbrauch, wenn ein großer Baum aufgebaut wird. Um mit diesem KI-Verfahren weiter zu kommen, muss wahrscheinlich spieltheoretisches Wissen integriert werden, sowohl in der Auswahl des nächsten zu erweiternden Baum-Knoten, als auch im Verlauf der Simulationen. Weiterhin könnte versucht werden, einen Explorationsfaktor von null zu wählen und die in Kapitel 5.3 erwähnte AMAF-Heuristik zu implementieren, die auch von Arneson et al. erfolgreich verwendet wurde [13].

7. Zusammenfassung

Zur Untersuchung der KI-Verfahren im GBG-Framework wurde eine skalierbare Version von Hex implementiert, welche sowohl für Agenten, als auch für Menschen spielbar ist. Die Entscheidungen der Agenten werden für einen guten Überblick anschaulich dargestellt. Im Anschluss wurden die auf Hex anwendbaren KI-Verfahren im Detail untersucht.

Für die Evaluation der Agenten wurden mehrere verschiedene Ansätze verwendet. Auf Spielbrettern, die es noch ermöglichten, wurde der Minimax-Agent zur Evaluation eingesetzt. Mit seiner hohen Geschwindigkeit nach der Erstellung des Suchbaums konnte eine große Menge an Daten generiert werden. Auf größeren Spielbrettern wurde ein MCTS-Agent mit hohen Iterationen, sowie Hexy von V. V. Anshelevich eingesetzt. Nachdem der MCTS-Agent nicht mehr eingesetzt und Hexy nicht mehr verwendet werden konnte, wurde der Demo-Modus von Hexy verwendet, um wenigstens die Qualität der Eröffnungszüge des zu evaluierenden Agenten zu bestimmen.

Der Minimax-Agent erreichte, wie zu erwarten war, eine perfekte Spielweise. Dadurch, dass der gesamte Zustandsraum vom Minimax-Agenten durchsucht wird, konnte er jedoch nur für kleinere Spielbretter eingesetzt werden. Auf dem verwendeten Rechner mit 8 GB Arbeitsspeicher war der Einsatz des Minimax-Agenten im GBG-Framework nur bis einschließlich des 4×4-Spielbretts möglich.

Der Monte-Carlo-Tree-Search-Agent (MCTS-Agent) verwendet ähnlich wie der Minimax-Agent eine Baumsuche, durchsucht dabei aber nicht jeden Spielzustand. Der MCTS-Agent verwendet eine Heuristik, um zu bestimmen, welche Zustände besucht werden. Die verwendete Heuristik mit dem Namen „UCT“ besteht aus einem Teil der für *Exploitation* verwendet wird, welcher dafür sorgt, dass vielversprechende Zustände genauer betrachtet werden, und einem *Exploration*-Teil, der für die Erforschung unbekannter Zustände verantwortlich ist. Verglichen mit dem Minimax-Agenten ist er effizienter im Bezug zum Speicherverbrauch. Erst ab dem 7×7-Spielbrett war der Einsatz des Agenten aufgrund des Speicherverbrauchs nicht mehr möglich. Jedoch baut der MCTS-Agent seinen Suchbaum für jeden Zug neu auf, weshalb er langsamer ist, als ein Minimax-Agent, der seinen Suchbaum wiederverwenden kann. Ebenso wie der Minimax-Agent erreicht der MCTS-Agent eine optimale Spielweise in Hex, wenn ihm genügend Rechenzeit gegeben wird. Die Anzahl der Iterationen, die dafür notwendig sind, wurde als ungefähr 10^n auf einem $n \times n$-Spielbrett bestimmt.

Der TD-Agent, der auf *Temporal Difference Learning* setzt, ist der einzige Agent, bei dem versucht wurde, Vorwissen über das Spiel zu integrieren. Die Auswahl der Features, die für diesen Zweck verwendet werden, ist dabei enorm wichtig für den Erfolg des Agenten. Die entwickelten Feature-Modi konnten jedoch nicht für einen stabilen Lernerfolg sorgen. Der einzige brauchbare Feature-Modus war der, der die rohen Spielbrett-Informationen verwendet. Dieser funktioniert jedoch nur auf sehr kleinen Spielbrettern relativ zuverlässig. Um einen besseres Trainingsergebnis zu erzielen, müssten weitere Features gefunden und

implementiert werden, damit der Agent ein vollständigeres Bild über den Spielzustand bekommt.

Der TDNT-Agent, der ebenfalls auf *Temporal Difference Learning* setzt und dabei N-Tupel-Systeme als Features verwendet, zeigte eine hervorragende Leistung in Anwendung auf Hex. Nachdem auf dem 4×4-Spielbrett der Einfluss der verschiedenen Parameter auf die Leistung des Agenten bestimmt wurde, wurden mit diesem Wissen bis zum 8×8-Spielbrett Agenten trainiert. Bis einschließlich 7×7 gelang es nach dem Training, Hexy, den Weltmeister der *Computer Games Olympiad* 2000, zu besiegen. Die Tupel des TDNT-Agenten wurden zufällig nach dem *Random Walk*-Prinzip gewählt, sodass dieser Agent kein spieltheoretisches Wissen verwendet hat. Dadurch, dass die Tupel prozedural generiert werden, eignet sich der Agent auch hervorragend für den Einsatz auf einem skalierbaren Spielbrett. Schlecht gewählte Tupel gehen dabei in der Menge unter, da der Agent durch Anpassung seiner Gewichtungen lernt, unnütze Merkmale nicht zu beachten. Die Ergebnisse des Agenten sind zudem unter dem Wissen über die verwendeten Parameter auch sehr gut reproduzierbar, da in aller Regel ein sehr stabiler Lernerfolg gemessen wurde.

Abschließend kann gesagt werden, dass das „General Board Game Playing“-Paradigma erfolgreich auf Hex angewandt werden kann. Mehrere Agenten erreichten eine sehr starke Spielweise, ohne dass Ihnen Vorwissen über das Spiel gegeben wurde. Obwohl es in Hex eine Vielzahl an Mustern gibt, die von guten Spielern erkannt werden müssen, konnten diese Agenten zuverlässig mit Minimax einen perfekten Spieler besiegen. Die verwendeten KI-Verfahren sind so generell einsetzbar, dass sie kein Wissen über die auf dem Spielbrett auftauchenden Muster brauchen, oder sich dieses Wissen selbst aneignen.

Literaturverzeichnis

[1] W. Konen, „The GBG Class Interface Tutorial," June 2017.

[2] D. Berman, "Hex Must Have a Winner: An Inductive Proof," *Mathematics Magazine,* vol. 49, no. 2, pp. 85-86, März 1976.

[3] J. Nash, „Some games and machines for playing them," RAND, 1952.

[4] P. Henderson, B. Arneson und R. B. Hayward, „Solving 8x8 Hex," in *21st International Joint Conference on Artificial Intelligence (IJCAI 09)*, 2009.

[5] H. J. van den Herik, J. W. Uiterwijk und J. van Rijswijck, „Games solved: Now and in the future," *Artificial Intelligence,* pp. 277-311, 2002.

[6] R. B. Hayward, Y. Björnsson, M. Johanson, M. Kan, N. Po und J. van Rijswijck, „Solving 7x7 Hex: Virtual connections and game-state reduction," *Advances in Computer Games,* Bd. 263, pp. 261-278, 2004.

[7] J. van Rijswijck, „Search and evaluation in Hex," Edmonton, Alberta, 2002.

[8] J. Pawlewicz und R. B. Hayward, „Scalable Parallel DFPN Search," *Computers and Games,* pp. 138-150, 2014.

[9] V. V. Anshelevich, „The game of Hex: An automatic theorem proving approach to game programming," in *AAAI/IAAI*, 2000.

[10] B. Arneson, R. Hayward und P. Henderson, „Wolve 2008 Wins Hex Tournament," *ICGA Journal,* Bd. 32, Nr. 1, pp. 49-53, März 2009.

[11] B. Arneson, R. Hayward und P. Henderson, „MoHex Wins Hex Tournament," *ICGA Journal,* Bd. 32, Nr. 2, pp. 114-116, Juni 2009.

[12] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis und S. Colton, „A survey of monte carlo tree search methods," *IEEE Transactions on Computational Intelligence and AI in games,* Bd. 4, Nr. 1, pp. 1-43, 2012.

[13] B. Arneson, R. B. Hayward und P. Henderson, „Monte Carlo tree search in Hex," *IEEE Transactions on Computational Intelligence and AI in Games,* Bd. 2, Nr. 4, pp. 251-258, 2010.

[14] K. Kahl, S. Edelkamp und L. Hildebrand, „Learning How to Play Hex," *KI 2007: Advances in Artificial Intelligence,* pp. 382-396, 2007.

[15] P. Borovska und M. Lazarova, „Efficiency of Parallel Minimax Algorithm for Game Tree Search," in *Proceedings of the 2007 international conference on Computer systems and technologies*, 2007.

[16] R. L. Rivest, „Game tree searching by min/max approximation," *Artificial Intelligence,* Bd. 34, Nr. 1, pp. 77-96, 1987.

[17] G. Chaslot, S. Bakkes, I. Szita und P. Spronck, „Monte-Carlo Tree Search: A New Framework for Game AI," in *Proceedings of the Fourth Artificial Intelligence and Interactive Digital Entertainment Conference*, 2008.

[18] L. Kocsis und C. Szepesvári, „Bandit based monte-carlo planning," *ECML,* Bd. 6, pp. 282-293, 2006.

[19] A. L. Samuel, „Some studies in machine learning using the game of checkers," *IBM Journal of research and development,* Bd. 3, Nr. 3, pp. 210-229, July 1959.

[20] G. Tesauro, „TD-Gammon, A Self-Teaching Backgammon Program, Achieves Master-Level Play," *AAAI 93 Fall Symposium - Games: Planning and Learning,* Bd. 2, 1993.

[21] W. Konen und T. Bartz-Beielstein, „Reinforcement Learning: Insights from Interesting Failures in Parameter Selection," in *PPSN*, 2008.

[22] W. Konen, „Reinforcement Learning for Board Games: The Temporal Difference Algorithm," 2015.

[23] S. M. Lucas, „Learning to play Othello with n-tuple systems," *Australian Journal of Intelligent Information Processing,* Bd. 4, pp. 1-20, 2008.

[24] S. Bagheri und M. Thill, „Temporal Coherence in TD-Learning for Strategic Board Games," 2014.

[25] D. P. Helmbold und A. Parker-Wood, „All-Moves-As-First Heuristics in Monte-Carlo Go," *Proceedings of the 2009 International Conference on Artificial Intelligence,* pp. 605-610, 2009.

[26] ICGA, „Hex (ICGA Tournaments)," [Online]. Available: https://www.game-ai-forum.org/icga-tournaments/game.php?id=7. [Zugriff am 22 Juli 2017].

[27] R. Hayward, B. Arneson, S.-C. Huang und J. Pawlewicz, „MoHex wins Hex tournament," *ICGA Journal,* Bd. 36, Nr. 3, pp. 180-183, 2013.

Anhang

A. 1: Gewinner der „*Computer Games Olympiad*“ (2000-2013), Hex

Edition	Event	Teilnehmer	Gewinner
17	Yokohama 2013	4	MoHex
16	Tilburg 2011	3	MoHex
15	Kanazawa 2010	5	MoHex
14	Pamplona 2009	4	MoHex 2009
13	Beijing 2008	4	Wolve 2008
11	Turin 2006	3	Six
9	Ramat-Gan 2004	2	Six
8	Graz 2003	2	Six
5	London 2000	3	Hexy

Tabelle 12: Die Gewinner der Hex Wettbewerbe zwischen 2000 und 2013. Quelle: [26]
Der Gewinner des Yokohama 2013 Events wurde ergänzt, da dieser nicht eingetragen war. Nach Hayward et al. [27] hat erneut MoHex gewonnen.

Erklärung

Ich versichere, die von mir vorgelegte Arbeit selbständig verfasst zu haben. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder nicht veröffentlichten Arbeiten anderer 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.

Ort, Datum

Unterschrift