AlphaZero-inspirierte KI-Agenten im General Board Game Playing

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

vorgelegt von:	Johannes Scheiermann
Matrikel-Nr.:	11110001
Adresse:	St.-Vither Str. 11
	50933 Köln
	johannes.scheiermann@live.de

eingereicht bei:	Prof. Dr. rer. nat., Dipl.-Phys. Wolfgang Konen
Zweitgutachter/in:	Prof. Dr. rer. nat., Dipl.-Inf. Heinrich Klocke

Köln, 02.11.2020

Erklärung

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

Köln, 02.11.2020

Ort, Datum

Rechtsverbindliche Unterschrift

Kurzfassung

Die vorliegende Abschlussarbeit ist im Forschungsfeld der künstlichen Intelligenz angesiedelt und adressiert speziell den Bereich des General Board Game Playing. Inspiriert durch die erstaunliche Spielstärke von AlphaZero im hochkomplexen Brettspiel Go, untersucht diese Arbeit, ob es durch die Adaption der diesem Algorithmus zugrunde liegenden Kernprinzipien bereits ohne die Bereitstellung enormer Rechenkapazitäten möglich ist, beeindruckende Resultate zu erzielen. Dazu wird auf dem in diesem Forschungsbereich angesiedelten GBG-Framework aufgebaut. Dieses wird in der vorliegenden Arbeit um einen Agenten-Wrapper ergänzt, welcher auf einer Monte-Carlo-Baumsuche basiert und einen gegeben Spielagenten um einen Zukunftsblick erweitert, sodass Letzterer in der Lage ist, unter gegebenen Rechenkapazitäten bessere Spielergebnisse zu erzielen. Im Rahmen der Evaluation wird dargelegt, dass die Erweiterung eines Agenten um eine Monte-Carlo-Baumsuche bereits mit einer niedrigen Iterationszahl zu einer deutlichen Leistungssteigerung führen und dass dieser Ansatz vor allem bei komplexen Spielen wirksamer als ein Minimax-Algorithmus abschneiden kann. Mit dem implementierten MCTS-Wrapper konnte zudem der äußerst starke Othello-Agent Edax zum ersten Mal von einem Spielagenten des GBG-Framework bei einer Suchtiefe größer drei besiegt werden. So ließ sich durch die Kombination eines TD-N-Tupel-Agenten mit dem MCTS-Wrapper der Edax-Spielagent mit den Suchtiefen eins bis sechs mit einer durchschnittlichen Siegesrate von mehr als 50 % schlagen.

Inhaltsverzeichnis

Erklärung **I**

Kurzfassung **II**

Abbildungsverzeichnis **V**

Algorithmenverzeichnis **VI**

Abkürzungsverzeichnis **VII**

1 Einleitung **1**
1.1 Motivation . . . 1
1.2 GBG-Framework . . . 2
1.2.1 Einstieg in die künstliche Intelligenz . . . 2
1.2.2 Unterschiede zu GGP . . . 2
1.2.3 Nutzen in der Forschung . . . 3
1.3 Problemstellung . . . 3
1.4 Projektziel . . . 3
1.5 Projektablauf . . . 5
1.6 Verwandte Arbeiten . . . 5

2 Grundbegriffe der künstlichen Intelligenz **6**
2.1 Praktische Anwendungsfälle . . . 6
2.1.1 Expertensysteme . . . 6
2.1.2 Mustererkennung . . . 7
2.1.3 Mustervorhersage . . . 7
2.1.4 Robotik . . . 7
2.2 Maschinelles Lernen . . . 8
2.2.1 Überwachtes (Supervised) Lernen . . . 8
2.2.2 Unüberwachtes (Unsupervised) Lernen . . . 8
2.2.3 Bestärkendes (Reinforcement) Lernen . . . 9

3 Monte Carlo Tree Search **10**
3.1 Selektion . . . 11
3.1.1 Exploitation in der UCT-Formel . . . 12
3.1.2 Exploration in der UCT-Formel . . . 12
3.2 Expansion . . . 13
3.3 Simulation . . . 13
3.4 Backpropagation . . . 15
3.5 Weitere Iterationen . . . 16

4 Spiel-Komplexität **17**
4.1 Komplexität hinter Go . . . 17
4.2 Umgang mit astronomischen Spielbäumen . . . 18

5 Entwicklung von AlphaGo zu AlphaZero 20
5.1 AlphaGo . . . 20
5.2 AlphaGo Zero . . . 21
5.2.1 Architektonische Unterschiede zu AlphaGo . . . 21
5.2.2 Training . . . 21
5.2.3 Evaluation im direkten Vergleich mit AlphaGo . . . 22
5.3 AlphaZero . . . 23
5.3.1 Grundsätzliche Unterschiede zu AlphaGo Zero . . . 23
5.3.2 Training . . . 24
5.3.3 Evaluation im direkten Vergleich mit Weltmeister-Programmen . . . 24

6 Implementierung eines AlphaZero-inspirierten Agenten 25
6.1 Anforderungen . . . 25
6.2 Integration in die GBG-Umgebung . . . 26
6.3 Berücksichtigung von Passsituationen . . . 29
6.3.1 Spielaktionen . . . 29
6.3.2 Spielzustände . . . 32
6.4 Komponenten einer Monte-Carlo-Baumsuche . . . 34
6.4.1 Suchalgorithmus . . . 34
6.4.2 Suchbaumknoten . . . 37
6.5 Approximation von Spielzustandsbewertungen . . . 41

7 Evaluation des AlphaZero-inspirierten Agenten 42
7.1 TD-3-Tupel-Agent ohne Lookahead . . . 42
7.2 TD-3-Tupel-Agent mit Minimax-Wrapper . . . 43
7.3 Edax . . . 45
7.3.1 Aufbau des Experiments . . . 45
7.3.2 Beurteilung der Ergebnisse . . . 46

8 Hürden und Fallstricke 47
8.1 Erste entwickelte Version . . . 47
8.2 Zweite entwickelte Version . . . 48
8.3 Dritte entwickelte Version . . . 49
8.4 Anbindung von Ludii . . . 50

9 Fazit und Zukunftsausblick 51

Literaturverzeichnis 52

Anhang 56

Abbildungsverzeichnis

1 Bestandteile einer MCTS-Iteration . . . 11
2 MCTS-Selektion . . . 11
3 MCTS-Expansion . . . 13
4 MCTS-Simulation . . . 14
5 MCTS-Backpropagation . . . 15
6 Aktualisierter Monte-Carlo-Suchbaum . . . 16
7 Die ersten drei Ebenen des Spielbaums von Tic-Tac-Toe . . . 17
8 Eingrenzung der Suchbreite . . . 18
9 Eingrenzung der Suchtiefe . . . 18
10 Rotationen eines Othello-Spielbretts . . . 23
11 Konzept des MCTS-Wrappers . . . 25
12 Auswahl der Agenten-Parameter in der Arena-Ansicht des Spiels Othello . . . 26
13 Konfiguration des MCTS-Wrapper-Agenten in der GBG-Benutzeroberfläche 27
14 Zeilen 2-6 der *getNextAction2*-Implementierung . . . 28
15 Zeilen 8-10 der *getNextAction2*-Implementierung . . . 28
16 Zeilen 12-15 der *getNextAction2*-Implementierung . . . 29
17 Spezifikation der Aktionskomponenten als UML-Diagramm . . . 29
18 Spezifikation der GameStateIncludingPass-Klasse als UML-Diagramm . . . 32
19 Zeilen 2-4 des MCTS-Pseudocodes . . . 35
20 Zeilen 6-11 des MCTS-Pseudocodes . . . 36
21 Zeilen 13-20 des MCTS-Pseudocodes . . . 37
22 Spezifikation der MCTSNode-Klasse als UML-Diagramm . . . 37
23 Zeilen 4-13 der *selectChild*-Implementierung . . . 40
24 Zeilen 15-22 der *selectChild*-Implementierung . . . 41
25 Spezifikation der Approximator-Komponente als UML-Diagramm . . . 41
26 Siegesraten und Zeitverhalten gegen einen blanken TD-3-Tupel-Agenten . . . 43
27 Vergleich von MCTS- und Max-N-Wrapper bezüglich Spielstärke und Zeitverhalten . . . 44
28 Siegesraten gegen Edax-Agenten mit unterschiedlichen Suchtiefen . . . 46

Algorithmenverzeichnis

1 Vereinfacht dargestellte Implementierung der getNextAction2-Methode durch den MCTS-Wrapper-Agenten 27
2 Anwendung einer regulären Spielaktion auf einen Spielzustand 30
3 Anwendung einer Passaktion auf einen Spielzustand 31
4 Rekursive Monte-Carlo-Baumsuche für Spiele mit zwei Teilnehmern 35
5 Selektionsmechanismus der Monte-Carlo-Baumsuche 39

Abkürzungsverzeichnis

GBG General Board Game

KI Künstliche Intelligenz

GGP General Game Playing

GDL Game Description Language

MCTS Monte Carlo Tree Search

TPU Tensor Processing Unit

GPU Graphics Processing Unit

UCT Upper Confidence Bound applied to Trees

PUCT Polynomial Upper Confidence Trees

DCNN Deep Convolutional Neural Network

CNN Convolutional Neural Network

UML Unified Modeling Language

1 Einleitung

Diese Arbeit adressiert das Forschungsfeld der künstlichen Intelligenz (KI). Auch wenn die Anfänge der KI bereits mehr als 60 Jahre zurückliegen[1], findet diese vor allem in letzter Zeit vermehrt den Weg in den menschlichen Alltag und gewinnt dabei zunehmend an Bedeutung. Die alltägliche Präsenz künstlicher Intelligenz lässt sich nicht immer bewusst wahrnehmen. So ist beispielsweise nicht jedem Endnutzer klar, dass die ihm präsentierten Medien oder Werbespots im Internet oft bewusst auf seine individuellen Interessen zugeschnitten werden. Dies lässt sich durch das Erkennen von Mustern in komplexen Nutzerdaten durch Verfahren der KI ermöglichen. Auf der anderen Seite gibt es Alltagssituationen, in denen der Einfluss künstlicher Intelligenz nicht zu übersehen ist. So kann das Fahren in einem autonom agierenden Fahrzeug gemischte Gefühle bei den Insassen hervorrufen oder gar an einen Science-Fiction-Film erinnern.

1.1 Motivation

Es gibt einiges, was die Forschung im Bereich der künstlichen Intelligenz so faszinierend macht. Einerseits ist die Interdisziplinarität in dieser Wissenschaft beeindruckend. So stammt die Inspiration für künstliche neuronale Netze aus der Hirnforschung. Doch auch Wissenschaften der Philosophie zeigen Interesse an der Thematik und diskutieren beispielsweise ethische Bedenken. Nicht zuletzt sind es solche Faktoren, die zu dieser Arbeit motiviert haben.
Künstliche Intelligenz kann ein emotionales Thema sein, denn sie ermöglicht, dass Maschinen ihrem Gegenüber auf einer zuvor nie da gewesenen, fast schon menschlichen Art begegnen können. Dies zeigte sich beispielsweise an dem hitzigen Go-Turnier zwischen dem Computerprogramm AlphaGo und dem zu diesem Zeitpunkt 18-fachen Weltmeister Lee Sedol im Jahr 2016. Im Laufe einer Woche konnte die künstliche Intelligenz vier von fünf Spielpartien und damit das Turnier für sich entscheiden [2]. Sedol zeigte sich von der Spielstärke der KI überrascht und kommentierte den 37. Spielzug wie folgt: ”I thought AlphaGo was based on probability calculation and it was merely a machine. But when I saw this move I changed my mind. Surely AlphaGo is creative.” [3].
Neben weiteren unkonventionellen Spielzügen stellte vor allem Zug 37 Jahrhunderte lange Go Erfahrung in Frage. Auch sonst gilt dieses Turnier als ein Durchbruch in der Geschichte der künstlichen Intelligenz. Denn bei Go handelt es sich um ein Spiel, welches sich aufgrund seiner mehr als 10^{170} [4, S. 174] möglichen Spielbrettkonfigurationen mit der Rechenleistung heutiger Computer nicht durch einen Brute-Force-Ansatz bezwingen lässt [5]. Damit hat AlphaGo als erstes Computerprogramm überhaupt einen professionellen Go-Spieler schlagen können [6, S. 484].
Bereits im Folgejahr dieses Meilensteins beschreibt DeepMind in einem Paper das Programm AlphaGo Zero als AlphaGo-Nachfolger. Dieses lernt Go ausschließlich durch das Spielen gegen sich selbst, ohne die Bereitstellung menschlichen Wissens und konnte von 100 Spielpartien gegen AlphaGo alle für sich entscheiden [7].

[1] Die Dartmouth-Konferenz im Jahr 1956 gilt als Geburtsstunde der KI [1]

Diese Ergebnisse zeigen, weshalb die Anwendung künstlicher Intelligenz in Brettspielen wichtig für die KI-Forschung ist. Denn durch kompetitives Austragen von Brettspielen lassen sich hilfreiche Erkenntnisse in Bezug auf bestimmte KI-Algorithmen gewinnen. So beschreibt DeepMind die für AlphaGo verwendeten Methoden als generisch und spricht die Hoffnung aus, diese eines Tages zum Zweck der Bewältigung gesellschaftlicher Herausforderungen wie der Klimamodellierung oder Analyse komplexer Krankheiten erweitern zu können [8].

1.2 GBG-Framework

Bei dem General Board Game (GBG) Playing and Learning Framework handelt es sich um eine in der Programmiersprache Java entwickelte Open-Source-Software[2], welche an der Technischen Hochschule Köln von Prof. Dr. Wolfgang Konen und seiner Forschungsgruppe entwickelt wird und die Anwendung künstlicher Intelligenz in Brettspielen untersucht.
Als Brettspiele werden im GBG-Kontext Spiele mit einer beliebigen Anzahl an Teilnehmern bezeichnet, solange diese ihre Spielzüge nach einander durchführen [9]. Die Spiele können dabei sowohl deterministisch als auch nichtdeterministisch sein.

1.2.1 Einstieg in die künstliche Intelligenz

Das GBG-Framework eignet sich nicht nur für wissenschaftliche Forschungszwecke. So beschreibt Konen [9] es als eine der Motivationen hinter dem Framework, Einstiegsbarrieren für Studierende in die Welt der künstlichen Intelligenz zu reduzieren. Die Infrastruktur des Framework lässt sich gut als Fundament für wissenschaftliche Arbeiten in diesem Bereich nutzen, sodass nicht alles von Grund auf selbst entwickelt werden muss. Beispielsweise stellt es Interfaces bereit [9], die lediglich implementiert werden müssen, um einen neuen KI-Agenten oder auch ein weiteres Spiel verfügbar zu machen. Das hat den angenehmen Seiteneffekt, dass die eigens entwickelten Komponenten in beliebigen Kombinationen mit allen weiteren im Framework vorhandenen Spielen und Agenten nutzbar sind. Durch diesen generischen Ansatz wird die Adaption bestehender Agenten für neu hinzugefügte Spiele überflüssig [9].

1.2.2 Unterschiede zu GGP

Der Forschungsbereich des GBG-Framework ähnelt im Groben zwar dem General Game Playing (GGP), weist aber unter genauerer Betrachtung einige markante Unterschiede auf. Denn GGP-Umgebungen stellen den Spiel-Agenten keine spielspezifischen Informationen zur Verfügung [10, S. 1], da Spiele auf einer abstrakteren Ebene behandelt werden. So geht GGP eng mit der Verwendung von Game Description Language (GDL) einher, wobei es sich bei Letzterem um eine Sprache handelt, in der die Spielregeln willkürlicher Spiele formal festgehalten werden können [10, S. 3]. Es ist damit also möglich, neue Spiele zu beschreiben und zur Laufzeit in einem GGP-System zu importieren, wo diese von den verfügbaren Agenten gespielt werden können. Das funktioniert selbst dann, wenn die Spiele dem System zuvor unbekannt waren.

[2] Der dokumentierte Quellcode steht auf https://github.com/WolfgangKonen/GBG zur Verfügung

Diese Universalität kommt aber nicht ausschließlich mit Vorteilen daher. In diesem Zusammenhang beschreibt Świechowski, dass KI-Methoden durch die Einschränkungen von GDL bislang nur begrenzte Erfolge im GGP verzeichnen könnten. Gründe dafür fänden sich beispielsweise im Mangel von verfügbaren spielspezifischen Informationen oder der eingeschränkten Performance wieder [11, S. 19].
Im Gegensatz dazu macht GBG keinen Gebrauch von GDL, denn neue Spiele müssen zur Kompilierzeit implementiert werden, bevor sie anschließend von beliebigen Agenten gespielt werden können. Damit zielt das GBG-Framework weniger auf Universalität, sondern vielmehr auf Performance ab [9].

1.2.3 Nutzen in der Forschung

Neben dem Vereinfachen des Einstiegs in die KI adressiert das GBG-Framework auch die Forschung. Denn es bietet der Wissenschaft als Plattform mit diversen bereits implementierten Spielen unter anderem die Möglichkeit, entwickelte KI-Agenten in Bezug auf ihre Spielstärke oder Allgemeingültigkeit zu evaluieren. Ein Vorteil hierbei ist die Präsenz spielspezifischer Informationen, welche es möglich machen, perfekt spielende Agenten zu implementieren. Durch den direkten Vergleich mit diesen können beispielsweise Aussagen darüber getroffen werden, wie gut eigene KI-Agenten abschneiden [9]. Außerdem lassen sich Spielagenten, welche auf Methoden der künstlichen Intelligenz basieren, in allen verfügbaren Spielen trainieren [9].

1.3 Problemstellung

Zweifelsfrei hat AlphaZero durch seine immense Spielstärke neue Maßstäbe setzen können. Äußerst bemerkenswert ist dabei, dass dieser generische Algorithmus ausgehend von Tabula rasa verschiedene Spiele eigenständig und ohne die Berücksichtigung menschlicher Erkenntnisse erlernen und auf höchstem Niveau spielen kann.
Um diese Durchbrüche zu erreichen, war gemäß Silver et al. [12] eine große Rechenleistung notwendig. So wurden alleine 5.000 Tensor Processing Units (TPUs) erster Generation beansprucht, um während des Trainings Spiele gegen sich selbst zu simulieren. Gleichzeitig wurden 16 TPUs zweiter Generation verwendet, um die neuronalen Netze hinter AlphaZero zu trainieren. Eine TPU erster Generation ist im Groben mit einer Titan V Graphics Processing Unit (GPU) vergleichbar [12], wobei Letztere gemäß dem Hersteller Nvidia eine Leistungsfähigkeit von 110 TeraFLOPS [13] aufweist. Die zweite Generation der TPUs wird auf Google's Cloud-Plattform dagegen mit 180 TeraFLOPS [14] beworben.
So wird unter Berücksichtigung dieser gewaltigen Rechenleistung klar, dass sich die von DeepMind mit AlphaZero erzielten Ergebnisse nicht ohne Weiteres im kleineren Rahmen unter Verwendung alltäglicher Hardware rekonstruieren lassen.

1.4 Projektziel

Trotz der Tatsache, dass AlphaZero unter Zuhilfenahme enormer Rechenkapazitäten entwickelt und trainiert wurde, befasst sich diese Arbeit mit den wesentlichen diesem Algorithmus zugrunde liegenden Prinzipien. So wird in diesem Rahmen evaluiert, ob Letztere

erst durch die Zuführung immenser Rechenleistung zu erfolgreichen Spielstrategien führen oder ob bereits durch den Einsatz alltäglicher Hardware neue Durchbrüche erzielt werden können. Dabei baut AlphaZero auf den beiden nachfolgend beschriebenen Prinzipien auf:

1. Durch die Verwendung einer Monte-Carlo-Baumsuche in Kombination mit einem neuronalen Netz reduziert AlphaZero Suchbäume auf intelligente Weise sowohl in ihrer Breite als auch Tiefe. Dadurch fokussiert sich die Baumsuche vorwiegend auf vielversprechende Spielbaumknoten, wodurch die verfügbare Rechenleistung mit einer höheren Effektivität eingesetzt wird. Eine detailliertere Beschreibung dieses Konzepts findet sich in Kapitel 4.2.

2. Der Trainingsprozess von AlphaZero steht in einer direkten Wechselwirkung mit der Monte-Carlo-Baumsuche. So findet vor jeder ausgewählten Spielaktion im Rahmen der generierten Selbstspiele eine Monte-Carlo-Baumsuche statt, wodurch im Trainingsverlauf einerseits stärkere Spielaktionen durchgeführt werden. Andererseits ergibt sich durch die Traversierungsverteilung der Knoten des Monte-Carlo-Suchbaums eine verbesserte Richtlinie zur Auswahl von Spielzügen. Letztere wird neben weiteren Parametern genutzt, um das verwendete neuronale Netzwerk zu trainieren. Weitere Informationen zu den Trainingsprozeduren von AlphaGo Zero und AlphaZero lassen sich den Kapiteln 5.2.2 und 5.3.2 entnehmen.

Zuvor wurde die Problematik des enormen Rechenbedarfs, welcher den durch AlphaZero erreichten Resultaten zugrunde liegt, aufgeführt. Diese Arbeit verfolgt das Ziel zu evaluieren, wie sich die grundlegenden Prinzipien von AlphaZero in einem kleineren Kontext verhalten. Als experimentelle Umgebung für diesen eingeschränkten Rahmen wird das GBG-Framework herangezogen, da dieses keiner außerordentlich hohen Rechenleistung bedarf. So lassen sich gemäß Konen [9] bereits mit dem Einsatz günstiger Hardwarekomponenten schnelle Ergebnisse erzielen. Damit ist es vor allem für diesen Forschungszweig interessant zu evaluieren, welche Effekte durch die Anwendung der Prinzipien von AlphaZero unter eingeschränktem Hardwareeinsatz im Umfeld des GBG-Framework erreicht werden können.
Da der im zweiten Prinzip dargelegte wechselseitige Trainingseffekt auf der im ersten Prinzip beschriebenen Monte-Carlo-Baumsuche basiert, richtet sich der Fokus dieser Arbeit bedingt durch die zugrunde liegende Zeitbeschränkung lediglich auf das erste dieser beiden Konzepte. Damit wird das Projekt von der folgenden zentralen Forschungsfrage geleitet:

`Können durch die Anwendung des AlphaZero zugrunde liegenden Prinzips, den Suchraum eines Spiels effektiv durch eine erweiterte Monte-Carlo-Baumsuche zu reduzieren, mit dem Einsatz alltäglicher Hardwarekomponenten neue Höchstleistungen innerhalb des GBG-Framework erreicht werden?`

1.5 Projektablauf

Zur Beantwortung der zuvor definierten zentralen Forschungsfrage soll im Rahmen des Projekts ein Spielagent in Anlehnung an AlphaZero entwickelt werden. Im Groben besteht AlphaZero aus einer Monte-Carlo-Baumsuche sowie einem Deep Convolutional Neural Network (DCNN). Bei Letzterem handelt es sich um eine Komponente, durch die im Rahmen der Monte-Carlo-Baumsuche Bewertungen von Spielzuständen approximiert werden. Damit wird der Suchraum in intelligenter Weise reduziert, was die Leistungsfähigkeit einer Monte-Carlo-Baumsuche verbessern kann. Durch die damit verbundene Komplexität ist die Entwicklung eines DCNNs im Zeitrahmen dieser Arbeit nicht realisierbar. Deshalb soll die durch dieses neuronale Netz durchgeführte Approximation durch ein Interface abstrahiert werden, sodass verschiedene Spielagenten des GBG-Framework zu dieser Aufgabe herangezogen werden können.
Nach der Entwicklung dieses Monte Carlo Tree Search (MCTS)-Wrapper-Agenten soll dessen Spielstärke im Brettspiel Othello evaluiert werden, um Informationen zur Beantwortung der Forschungsfrage zu gewinnen. Im Rahmen der Evaluation soll ein trainierter TD-3-Tupel-Agent als approximierende Komponente verwendet werden, da dieser bereits eine starke Performance im Spiel Othello und damit eine bewährte Kompetenz bezüglich der Bewertung von Spielzuständen aufweist. Dabei liegt dem Agenten mit einem TD-N-Tupel-Netzwerk eine sehr flache, aber breite Architektur eines neuronalen Netzes zugrunde.

1.6 Verwandte Arbeiten

Als thematisch nahe liegende Arbeiten seien zunächst die von DeepMind veröffentlichten Publikationen bezüglich AlphaGo [6], AlphaGo Zero [7] und AlphaZero [12] aufgeführt. Denn diese bieten erst die Grundlagen für weitere durch AlphaZero inspirierte Forschungsgegenstände.
In Kombination mit Othello fällt auf, dass AlphaZero angelehnten Forschungen häufig ein 6x6 Spielbrett zugrunde liegt. So beschreiben Wang et al. in [15] in diesem Bereich wie durch Verbesserungen der Monte-Carlo-Baumsuche das Selbstspiel des AlphaZero-Algorithmus verbessert werden kann, und evaluieren in [16] erneut im Rahmen von 6x6 Othello, wie sich die Parameter von AlphaZero auf den Trainingsprozess auswirken. Zudem zeigen Chang et al. [17] mit vielversprechenden Ergebnissen, inwiefern sich der AlphaZero-Algorithmus durch ihre Big-Win-Strategie für 6x6 Othello optimieren lässt.

Im Kontrast dazu befasst sich die vorliegende Arbeit nicht mit weiteren Optimierungen des AlphaZero-Algorithmus, sondern zunächst mit dessen Auswirkungen im Rahmen des GBG-Framework. Dabei liegt dem Forschungszweck ein Othello-Spielbrett der Größe 8x8 zugrunde.

2 Grundbegriffe der künstlichen Intelligenz

Bei der Wissenschaft von künstlicher Intelligenz handelt es sich um eine Teildisziplin der Informatik, dessen Forschungsgegenstand es ist, Maschinen lernen und in menschlicher Weise intelligente Entscheidungen treffen zu lassen. Auch wenn eine prägnante Definition der Bezeichnung "Künstliche Intelligenz", bedingt durch die Tatsache, dass keine allgemeingültige Definition des Wortes "Intelligenz" existiert[3], nicht trivial ist, gibt Elaine Rich's Aussage "Artificial Intelligence is the study of how to make computers do things at which, at the moment, people are better." [19] doch eine eloquente Zusammenfassung des Forschungsgebiets wieder.

2.1 Praktische Anwendungsfälle

Die praktischen Anwendungsfälle künstlicher Intelligenz sind breit gefächert, so finden sich KI-Algorithmen heutzutage in Handschrift- und Spracherkennung, Routenberechnung, Textübersetzung, Internetsuchmaschinen, autonomen Waffen, Industrierobotern oder persönlichen Assistenten wieder, nur um einige Beispiele zu nennen.
Trotz dieser Vielseitigkeit lassen sich die meisten KI-Systeme in folgende Kategorien einordnen:

2.1.1 Expertensysteme

Expertensysteme werden eingesetzt, wenn abgegrenztes und überschaubares Expertenwissen gekapselt werden soll, um daraus Schlussfolgerungen ziehen zu können [20, S. 12]. Im Groben besteht ein solches System aus Regeldefinitionen und einer Wissensbasis [5, S. 5]. Dadurch ist es Expertensystemen möglich, Spezialwissen spezifischer Fachbereiche bereitstellen zu können, da "spezialisiertes Expertenwissen [...] sehr viel leichter formalisiert einem Computer zur Verfügung gestellt werden kann als Allgemeinwissen und gesunder Menschenverstand [...]" [21, S. 60]. Expertensysteme konnten sich nach ihren Hochzeiten in den 1980er-Jahren nicht durchsetzen. Dies kann möglicherweise dadurch begründet sein, dass die implementierten Regeln häufig zu starr und die Systeme nur in begrenztem Maß lernfähig sind [5, S. 5].

Die für Ärzte kostenfrei zur Verfügung stehende Software LEXMED wurde an der Hochschule Ravensburg-Weingarten unter Prof. Ertel entwickelt und ist ein Beispiel für ein Expertensystem, welches im medizinischen Bereich verwendet wird, um die Diagnose von Blinddarmentzündungen zu unterstützen. So können gemäß Ertel beispielsweise Anfragen nach der "[...] Wahrscheinlichkeit für einen entzündeten Appendix, wenn der Patient ein 23-jähriger Mann mit Schmerzen im rechten Unterbauch und einem Leukozytenwert von 13000 ist[,] [...]" [1, S. 147] an das System gestellt werden. Manfred Schramm und Walter Rampf waren auch in die Entwicklung von LEXMED involviert und bewerten die 12-prozentige Fehlerdiagnoserate des Programms als sehr gering [22].

[3] Shane Legg und Marcus Hutter sammelten über Jahre hinweg 70 unterschiedliche Definitionen für den Begriff "Intelligenz" [18]

2.1.2 Mustererkennung

Die Kategorie Mustererkennung umfasst die Anwendungsfälle, in denen Computerprogramme das Erkennen von Mustern in Datenmengen erlernen. Die Umsetzung solcher Algorithmen kann durch die Implementierung von künstlichen neuronalen Netzen erfolgen [20, S. 201]. In der Praxis findet Mustererkennung beispielsweise beim Klassifizieren von Militärfahrzeugen auf Radar-Scans oder der Identifikation von Personen auf Fotos Anwendung [1, S. 276].

2.1.3 Mustervorhersage

In einigen Anwendungsfällen reicht es nicht aus, lediglich Muster in Datenmengen zu erkennen. Stattdessen bedarf es vielmehr einem tief greifenden Verständnis dieser Daten und Muster aufseiten der künstlichen Intelligenz, sodass möglichst genaue Zukunftsprognosen getroffen werden können. Durch Methoden des maschinellen Lernens können Programme diese Musterinterpretation sowie das Treffen von Zukunftsvorhersagen trainieren [5, S. 8]. Auch hier sind die praktischen Anwendungen breit gefächert. So nutzt das soziale Netzwerk Facebook entsprechende Algorithmen als prophylaktische Maßnahme, um suizidgefährdete Nutzer anhand ihrer Postings zu identifizieren [23, S. 42]. Weitere Anwendungsfälle sind beispielsweise Vorhersagen von Krankheits- oder Wetterverläufen [23].

2.1.4 Robotik

Die Robotik beschränkt sich nicht darauf, Programme lediglich auf Computern auszuführen, sondern gibt der Software durch die Bereitstellung entsprechender Hardware auch die Möglichkeit, mit der Außenwelt zu interagieren. Deshalb verfügen solche Hardware-Agenten zum einen über Sensoren, mit denen sie ihre Umgebung wahrnehmen können und zum anderen über Aktuatoren, durch die sie physisch mit dieser interagieren können [1, S. 12]. Praktische Anwendungsfälle von Robotern finden nicht unter Laborbedingungen statt, sodass unerwartete physische Einflüsse oder gar verzerrte Sensordaten im Betrieb nicht ausgeschlossen werden können. Aus diesem Grund ist in der Robotik häufig von einer nichtdeterministischen Umgebung auszugehen [1, S. 295]. Deshalb werden KI-Algorithmen in diesem Bereich klassischerweise durch Reinforcement-Learning-Ansätze trainiert [1, S. 281], wodurch sie eigenständig Strategien entwickeln, um ihre Aufgaben zu bewältigen, und es möglich ist, dass teilweise auch ein angemessener Umgang mit gewissen Ausnahmesituationen erlernt wird.

Buxmann und Schmidt [5, S. 127] nennen als denkbare praktische Einsatzmöglichkeiten der Robotik neben dem Ausführen von einfachen, sich wiederholenden Industrieaufgaben auch Tätigkeiten, welche die Roboter menschenähnlich wirken lassen. Beispiele für Letzteres liegen im Bereich der Pflege oder Unterstützung in Alltagssituationen wie dem Einkaufen.

2.2 Maschinelles Lernen

Für die Erschaffung intelligenter Maschinen ist es unabdingbar, dass diese lernfähig sind. So bezeichnet Wolfgang Ertel "[...] die Erforschung der Mechanismen des Lernens und die Entwicklung maschineller Lernverfahren [als] eines der wichtigsten Teilgebiete der KI [...]" [1, S. 179]. Maschinelles Lernen lässt sich dabei in die drei folgenden Kategorien unterteilen:

2.2.1 Überwachtes (Supervised) Lernen

In Verfahren dieser Kategorie erhält ein lernendes Programm neben einem Satz von Eingabe- auch die dazugehörigen Ausgabedaten [24, S. 528]. Hierbei ist es die Aufgabe des Algorithmus, eine Funktion zu finden, welche die bereitgestellten Eingaben auf die ebenso vorgegebenen Ausgaben approximiert [1, S. 281], sodass diese Funktion nach einem Training selbst für Eingaben außerhalb der Trainingsdaten korrekte Ausgabewerte annähert. Um mit dieser Methode Erfolge zu verzeichnen, sollte hinsichtlich der Trainingsdaten darauf geachtet werden, dass diese in ausreichender Menge und von guter Qualität zur Verfügung stehen [23, S. 13], da Ausreißer in den Daten die Allgemeingültigkeit der ermittelten Funktion verzerren.

Beispiel: Einem Algorithmus werden zahlreiche Bilder von Kleidungsstücken samt Bezeichnungen bereitgestellt, damit dieser lernt, für die Eingabe eines zuvor unbekannten Fotos von einem Kleidungsstück, dessen entsprechende Bezeichnung auszugeben.

2.2.2 Unüberwachtes (Unsupervised) Lernen

Bei dieser Art des maschinellen Lernens trainiert ein Programm, selbstständig Muster in Eingabedaten zu ermitteln. Da dem Algorithmus keine Auskunft über erwartete Ausgabewerte gegeben wird, ist dieser frei darin zu entscheiden, nach welchen Kriterien die Daten kategorisiert werden sollen [5, S. 10]. Der größte Unterschied zu einem Supervised-Learning-Algorithmus liegt also darin, dass dieser lernt, Datensätze nach vorgegeben Labels wie zum Beispiel Bezeichnungen zu clustern, während ein Unsupervised-Learning-Algorithmus nach eigenem Ermessen gruppiert.
So könnte Letzterer im Fall von Kleidung diese nach Farbe, Material oder gar Beschaffenheit kategorisieren. Das kann zu Paradigmenwechseln und damit neuen Erkenntnissen führen, beispielsweise wenn Klassifizierungsmöglichkeiten entdeckt werden, welche für Menschen nicht intuitiv erkennbar sind.

Beispiel: Ähnlich wie im vorherigen Beispiel soll eine KI lernen, Kleidung zu klassifizieren. Allerdings erhält das Programm dazu lediglich einen Trainingsdatensatz mit Bildern verschiedener Kleidungsstücke und diesmal keine Informationen zu dessen Bezeichnungen. So muss der Algorithmus eigenständig Muster finden, durch welche die Kleidung gruppiert werden kann.

2.2.3 Bestärkendes (Reinforcement) Lernen

Diese Methode unterscheidet sich von den beiden vorherigen dahingehend, dass ein Algorithmus ohne die Bereitstellung von Trainingsdaten [1, S. 281] lernen soll, ein Problem zu lösen. In der Regel sind diesem lediglich der aktuelle Zustand und die erlaubten Aktionen bekannt, nicht aber in welchem Zustand eine bestimmte Aktion sinnvoll ist [5, S. 11]. Gewählte Aktionen können abhängig davon, ob sie zielführend sind, von der Umgebung sowohl belohnt als auch bestraft werden [23, S. 16]. Das Lernen eines solchen Algorithmus stützt sich also darauf, Belohnungen zu maximieren und Strafen zu vermeiden [5, S. 11]. Ertel beschreibt [1, S. 281], dass dieser Lernprozess sich nicht sonderlich von dem eines Menschen unterscheidet, so können Stürze beim Erlernen des Radfahrens als Bestrafung und zurückgelegte Distanzen als Belohnung interpretiert werden.
Hierbei sollte beachtet werden, dass ein stumpfer Ansatz der Belohnungsmaximierung sich nicht immer als intelligent erweist und zu lokalen Maxima führen kann. So ist es beispielsweise in einigen Brettspielen keine Seltenheit, aus strategischen Gründen einen Zug zu wählen, welcher den Spielzustand kurzfristig verschlechtert, doch auf lange Sicht vorteilhafter ist. Beim Ausbeuten der maximalen Belohnung wird von Exploitation gesprochen, während das Erkunden alternativer, zunächst unattraktiv wirkender Aktionen als Exploration bezeichnet wird [23, S. 16]. Das Verhältnis dieser beiden Größen nimmt direkten Einfluss auf die Stärke von Reinforcement-Learning-Agenten und wird in der Fachsprache Exploration/Exploitation-Trade-Off genannt.

Beispiel: Ein Algorithmus soll das Schachspielen erlernen. Hierbei hat dieser nur Informationen darüber, in welchem Zustand sich das Spiel befindet und welche Spielzüge in einem bestimmten Spielzustand legitim sind, nicht aber ob und in welchem Maß einzelne Züge zielführend sind. Außerdem werden Siege belohnt und Niederlagen bestraft.

3 Monte Carlo Tree Search

Spielbäume komplexer Spiele wie Go weisen astronomische Größen auf, weshalb sie sich mit heutigen Rechnern nicht vollständig traversieren lassen [25, S. 159], um daraus perfekte Spielstrategien abzuleiten. Entgegen dieser Problematik bietet eine Monte-Carlo-Baumsuche [engl. Monte Carlo Tree Search (MCTS)] die Möglichkeit, bereits aus der partiellen Betrachtung solcher Suchbäume erfolgreiche Spielstrategien abzuleiten. Der Grundgedanke dahinter ist es, diese enormen Suchräume effektiv zu reduzieren, indem lediglich vielversprechende Knoten verfolgt werden [26, S. 1-2].
Der Zweck einer Monte-Carlo-Baumsuche besteht darin, ausgehend von einem Zustand s_0 zu ermitteln, welcher der verfügbaren Spielzüge gewählt werden soll, um die Wahrscheinlichkeit eines Siegs langfristig zu maximieren [26, S. 4].
Zu diesem Zweck baut der Algorithmus im Rahmen einer gegebenen Anzahl von Iterationen einen Monte-Carlo-Suchbaum auf. Zunächst besteht dieser Baum nur aus dem Wurzelknoten, welcher den bereitgestellten Ausgangszustand s_0 repräsentiert [27, S. 106]. Anschließend wird dieser Suchbaum durch jede durchgeführte Iteration um einen zusätzlichen Knoten erweitert, sofern noch nicht alle möglichen Zustandsknoten expandiert wurden [26, S. 4]. Bei jedem Kindknoten handelt es sich dabei um einen Folgezustand des entsprechenden Elternknotens. Der Folgezustand von s wird generiert, indem ein legitimer Spielzug ausgehend vom Zustand s simuliert wird.
Die Macht einer solchen Monte-Carlo-Baumsuche liegt definitiv in der Skalierung der Iterationen und damit der Größe des Monte-Carlo-Suchbaums. Denn je größer dieser wächst, desto mehr nähern sich dessen Ergebnisse an die eines Minimax-Algorithmus an [26, S. 4], wobei Letzterer darauf ausgelegt ist, eine optimale Spielstrategie zu ermitteln [24, S. 124]. So entsteht für den Algorithmus mit zunehmender Größe des Suchbaums ein tieferer "Einblick in die Zukunft", welcher im Folgenden als Lookahead bezeichnet wird und das Urteilsvermögen des Algorithmus dahingehend, wie zielführend bestimmte Aktionen langfristig sind und welche Konsequenzen sie mit sich bringen, verbessert. Im Beispiel von Brettspielen folgt daraus, dass ein MCTS-Agent mit zunehmender Iterationszahl stärker wird. Theoretisch konvergiert dabei die Wahrscheinlichkeit, zu jedem Zeitpunkt einen optimalen Spielzug zu wählen, nach einer unendlichen Anzahl von Iterationen gegen 100 % [28, S. 292].
Eine Monte-Carlo-Baumsuche kann vor dem Durchlaufen aller Iterationen vorzeitig terminiert werden. Anschließende Entscheidungen des Algorithmus basieren dann lediglich auf den Erkenntnissen der zuvor durchgeführten Iterationen. Aufgrund dieser Eigenschaft handelt es sich bei MCTS um einen Anytime-Algorithmus [27, S. 106]. Beispielsweise lässt sich dadurch die Berücksichtigung von Zeitlimits in den Algorithmus implementieren. In diesem Fall muss eine Zeitüberschreitung lediglich dazu führen, dass weitere Iterationen unterbunden werden.

Im Folgenden werden die vier Schritte [11, S. 3] einer MCTS-Iteration (vgl. Abbildung 1) entlang eines Beispiels für das Spiel Tic-Tac-Toe erläutert:

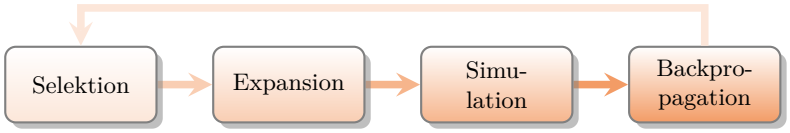

Abbildung 1 Bestandteile einer MCTS-Iteration

3.1 Selektion

Die Selektion ist der erste Schritt aller MCTS-Iterationen und beginnt jedes Mal bei dem Wurzelknoten des aufgebauten Monte-Carlo-Suchbaums [11, S. 2]. Beim Beispiel in Abbildung 2 ist dieser Knoten durch s_0 gekennzeichnet. Ausgehend von der Wurzel traversiert der Algorithmus die schon expandierten Knoten solange, bis ein Knoten erreicht wird, bei welchem mindestens ein Kindknoten nicht expandiert und damit unbekannt ist [26, S. 4]. Im Beispiel sind die drei Kindknoten von s_0 bereits expandiert, was zugleich bedeutet, dass zuvor drei MCTS-Iterationen durchgeführt wurden.

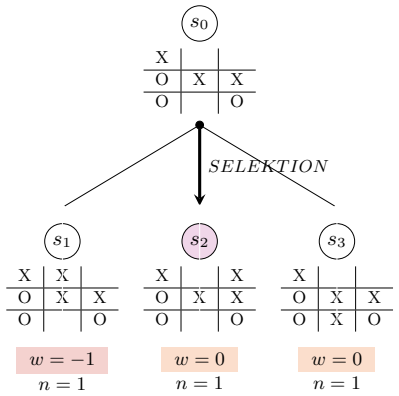

Abbildung 2 MCTS-Selektion

Da s_0 vollständig expandiert ist und somit für diesen Knoten keine zusätzliche Expansion stattfinden kann, muss auf Basis einer Policy entschieden werden, welcher Kindknoten im Rahmen der aktuellen Selektion durchlaufen und im weiteren Verlauf dieser MCTS-Iteration expandiert werden soll [26, S. 4]. Damit repräsentiert eine solche Policy die Richtlinie dafür, welche Knoten für den Algorithmus als langfristig vielversprechend gelten. Idealerweise bevorzugt diese Richtlinie einerseits Knoten, von denen bekannt ist, dass sie einen Sieg begünstigen. Wobei andererseits aber auch Knoten, welche bislang unerforscht waren, untersucht werden sollten, da diese unbekannte Potenziale bergen können. Eine Policy, mit der ein Ausgleich zwischen der Ausbeutung vorteilhafter und der Erkundung unerschlossener Knoten (Exploration/Exploitation-Trade-Off) sichergestellt werden kann, ist beispielsweise durch die Upper Confidence Bound applied to Trees (UCT)-Formel $\frac{w_i}{n_i} + c\sqrt{\frac{\ln N_i}{n_i}}$ gegeben [26, S. 6]. Dabei steht w_i für den Gesamtwert eines Knotens und n_i für die Häufigkeit, mit der dieser in den vorherigen MCTS-Iterationen insgesamt durchlaufen wurde. Bei c handelt es sich um einen Parameter, durch den das Bedürfnis nach Erkundung justiert werden kann.

In diesem Beispiel soll die UCT-Formel als Selektions-Policy dienen, wodurch der Kindknoten ausgewählt wird, welcher diese maximiert.

3.1.1 Exploitation in der UCT-Formel

Der erste Summand $\frac{w_i}{n_i}$ der UCT-Formel repräsentiert im Hinblick auf die Anzahl der Iterationen den durchschnittlichen Wert eines Knotens und begünstigt so die Ausbeutung von bewährten Knoten.

Beispiel: Wenn w_i die Häufigkeit der Siege angibt und die Pfade über einen bestimmten Kindknoten im Rahmen von 20 Simulationen jedes Mal zu einem Sieg führten, wird dieser Knoten mit einem arithmetischen Mittelwert von $\frac{20}{20}$ aus Ausbeutungssicht einem anderen Knoten mit einem beispielhaften Durchschnittswert von $\frac{10}{20}$ vorgezogen.

3.1.2 Exploration in der UCT-Formel

In dem zweiten Summanden $c\sqrt{\frac{\ln N_i}{n_i}}$ der UCT-Formel steht N_i für die Anzahl der MCTS-Iterationen, in denen der Vaterknoten selektiert wurde. Somit wird die Iterationsanzahl des Vaterknotens zu der des Kindknotens ins Verhältnis gesetzt, was die Erkundung wenig erforschter Kindknoten begünstigt.

Beispiel: Unter der Annahme, dass ein Vaterknoten 30 Mal traversiert wurde, sollen die Erkundungsbereitschaften für zwei Kindknoten ermittelt werden. Einer davon führte häufiger zu Siegen, wodurch dieser bereits 25 Selektionen hinter sich hat. Die Bereitschaft, diesen Knoten weiterhin zu erkunden, liegt also bei $c\sqrt{\frac{\ln 30}{25}} \approx 0,37c$.
Für einen weiteren Kindknoten, welcher zuvor erst zwei Mal durchlaufen wurde, liefert die Formel $c\sqrt{\frac{\ln 30}{2}} \approx 1,3c$. Dieser Knoten würde also aus Erkundungssicht unter der Annahme, dass es sich bei dem Explorationsfaktor c um einen positiven Wert handelt, bevorzugt werden.

Im Beispiel von Abbildung 2 wurden die Werte der Kindknoten s_1, s_2 und s_3 im Rahmen vorheriger Iterationen mit -1, 0 und 0 approximiert. Bedingt dadurch, dass die drei Knoten bisher mit derselben Häufigkeit jeweils ein Mal durchlaufen wurden, liefert der zweite Summand der UCT-Formel für alle das gleiche Ergebnis. Somit geschieht die Wahl des nächsten Knotens rein aus Ausbeutungssicht. Da die Knoten s_2 und s_3 mit Werten von jeweils 0 die erfolgversprechendsten sind, selektiert der Algorithmus zufällig den Knoten s_2. Weil kein Kindknoten von s_2 bekannt ist und deshalb die Selektion ausgehend von diesem Knoten nicht fortgeführt werden kann, ist der Selektionsschritt der aktuellen MCTS-Iteration abgeschlossen.

3.2 Expansion

In diesem Schritt soll der zuvor selektierte (vgl. Abbildung 2) Knoten s_2 expandiert werden. Abbildung 3 veranschaulicht, dass ausgehend von s_2, zwei unterschiedliche Spielzüge von Spieler O durchgeführt werden können. Das bedeutet gleichzeitig, dass der Knoten s_2 insgesamt zwei verschiedene Kindknoten expandieren kann.
Die Entscheidung darüber, welcher Spielzug simuliert und damit auch welcher Kindknoten expandiert werden soll, kann auf Basis spielspezifischer Heuristiken sowie zufällig getroffen werden. Letzteres trägt einen Teil dazu bei, dass sich eine Monte-Carlo-Baumsuche generisch auf verschiedene Spiele anwenden lässt.

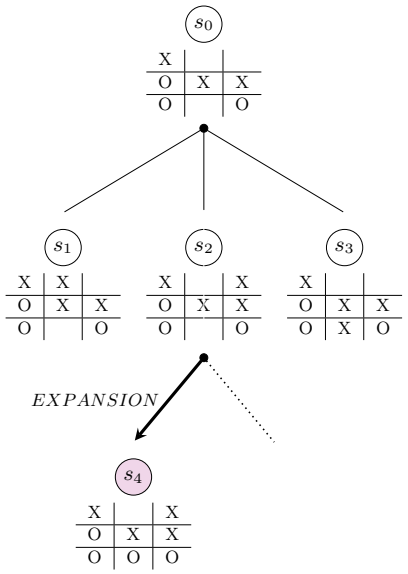

Abbildung 3 MCTS-Expansion

So finden MCTS-Algorithmen beispielsweise erfolgreich Anwendung im Bereich des GGP [11, S. 2], in dem spielenden Agenten aus Abstraktionsgründen gar keine spielspezifischen Informationen bereitgestellt werden können.

Im Beispiel von Abbildung 3 entscheidet sich der Algorithmus via Zufall für das Expandieren des ersten möglichen Folgezustands s_4, welcher in der Grafik durch eine rote Markierung hervorgehoben ist.

3.3 Simulation

Im Rahmen des Simulationsschrittes wird abgeschätzt, wie erstrebenswert der zuvor expandierte Spielzustand ist [26, S. 4]. Die Bewertung geschieht dabei aus der Sicht des Spielers, welcher ausgehend von diesem Zustand an der Reihe ist.
Im aktuellen Beispiel wurde ein Zug des Spielteilnehmers O simuliert (vgl. Abbildung 3), um den Knoten s_4 zu expandieren. Da im Spielzustand von s_4 Spieler X als Nächstes an der Reihe wäre, geschieht die Bewertung dieses Zustands aus dessen Sicht. Dabei sieht die Bewertungsgrundlage (vgl. Abbildung 4) vor, dass eine Niederlage mit -1, eine Patt-Situation mit 0 und ein Sieg mit +1 bewertet werden. In s_4 ist das Spiel bereits zugunsten von O entschieden, da dieser eine volle Dreierreihe aufweisen kann. Deshalb wird das Ergebnis aus Sicht von Spieler X als Niederlage mit dem Wert -1 bewertet. In diesem Fall ist es jedoch als Zufall zu betrachten, dass ein Beispiel gewählt wurde, in dem der zu bewertende Knoten bereits einen Endzustand des Spiels Tic-Tac-Toe repräsentiert.

Das zieht unausweichlich die Fragestellung nach sich, auf welcher Grundlage die Bewertung eines nicht entschiedenen Spielzustands erfolgt. Klassischerweise findet in diesem Fall ein Random-Rollout statt. Dabei werden ausgehend von einem zu bewertenden Knoten s so lange zufällige Züge für beide Spieler simuliert, bis ein terminierender Zustandsknoten s_T erreicht wird [27, S. 108]. Ein Knoten ist dann terminierend, wenn er einen Spielzustand enthält, bei dem das Spiel nicht mehr fortgeführt wird, da es entschieden ist. Dieser Endzustand wird anschließend wie im obigen Beispiel aus Sicht des Spielers bewertet, welcher im Zustand von s an der Reihe ist. Die im Rahmen des Rollouts simulierten zufälligen Spielzustände, welche sich zwischen s und s_T befinden, sind lediglich temporär [27, S. 108] und werden im Anschluss verworfen. Sie sind damit nur notwendig, um den Wert des nicht entschiedenen Spielzustands zu approximieren.

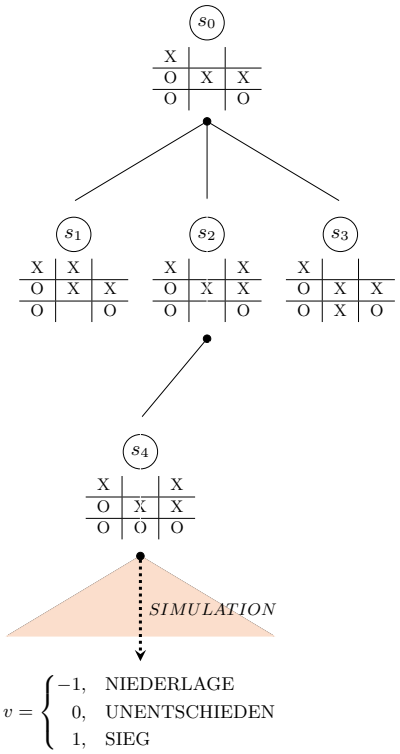

Abbildung 4 MCTS-Simulation

Der Grund, weshalb Random-Rollouts trotz des stochastischen Einflusses zu aussagekräftigen Bewertungen führen können, liegt in der statistischen Natur einer Monte-Carlo-Baumsuche. So ist es naheliegend, dass die Aussagekraft im Beispiel eines einzigen Random-Rollouts, welches auf einer hohen Spielbaum-Ebene beginnt und bis zu einem terminierenden Zustand durch sehr viele Ebenen hinweg Zufallszüge simuliert, gering ist. Doch mit zunehmender Anzahl an MCTS-Iterationen und damit ebenfalls Random-Rollouts von verschiedenen Knoten ausgehend formt sich ein statistisches Gesamtbild mit ansteigender Belastbarkeit.

Auch in diesem MCTS-Schritt lassen sich Optimierungen implementieren, indem beispielsweise die Auswahl der zu simulierenden Züge in den Rollouts nicht mehr zufällig [27, S. 108], sondern beispielsweise auf Basis von domänenspezifischen Heuristiken stattfindet. So könnten bestimmte Spielstärken möglicherweise mit weniger MCTS-Iterationen erreicht werden. Allerdings geht mit dieser Abhängigkeit von spielspezifischen Informationen die spielübegreifende generische Anwendbarkeit des Algorithmus verloren.

3.4 Backpropagation

Der im vorherigen Schritt approximierte Wert des expandierten Knotens s_4 soll nun im Rahmen der Backpropagation bis zum Wurzelknoten "zurückpropagiert" werden [27, S. 108]. Dadurch kann der MCTS-Algorithmus nach dem Durchführen der Iterationen ausgehend von der Wurzel des Monte-Carlo-Suchbaums die zielführendste Aktion wählen, um diese an die aufrufende Umgebung zurückzugegeben.

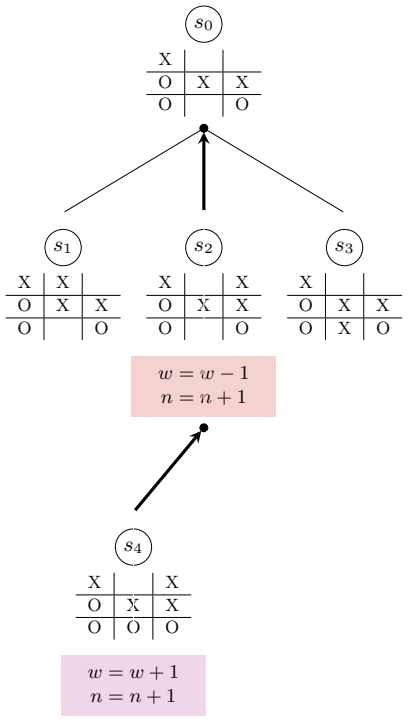

Abbildung 5 MCTS-Backpropagation

In diesem Beispiel gilt bei der Backpropagation zu beachten, dass der Wert initial und auch auf jeder durchlaufenen Baumebene negiert wird. Somit wird der Wert des Knotens s_4 um 1 inkrementiert, während der Wert des Knotens s_2 um 1 dekrementiert wird (vgl. Abbildung 5). Dies hat den Grund, dass der Wert w eines Spielzustands s stets aus der Sicht des Spielers betrachtet wird, welcher im vorherigen Zustand an der Reihe ist. Denn dieser hat mit seiner nächsten Aktion die Entscheidungsmacht darüber, ob das Spiel in den Zustand s wechseln wird.
Infolge der zuvor beschriebenen Negation wird der Knoten s_4 mit +1, anstatt -1 bewertet. Denn vom Knoten s_2, in dem O an der Reihe ist, ist es für diesen siegbringend eine Aktion zu wählen, die im Zustand s_4 resultiert. Um X aber davon abzuhalten, eine Aktion zu tätigen, die zur Spielbrettkonfiguration von s_2 führt, aus der O immer einen Sieg erzwingen kann, wird der Wert auf jeder Ebene negiert, so wie auch der Spieler, welcher am Zug ist mit jedem Ebenenwechsel alterniert.
Somit wird der Wert von s_2 um 1 dekrementiert, wodurch die UCT-Funktion im Rahmen der Selektionen weiterer MCTS-Iterationen diesen Knoten aufgrund der nun geringeren Ausbeute benachteiligen wird. Stattdessen werden dann die Kindknoten s_1 und s_3 bevorzugt erkundet. Dabei wird der Algorithmus gegebenenfalls auch herausfinden, dass s_3 der einzige Folgezustand ist, durch dessen Wahl ein Sieg des Spielers O ausgeschlossen ist.
Die in diesem Beispiel verwendete Methode, den zurückpropagierten Wert auf jeder Ebene zu negieren, impliziert die Anforderung, dass es sich um ein Spiel mit zwei Teilnehmern handelt. Außerdem darf es zugleich keine möglichen Passaktionen geben oder anderenfalls müssen diese entsprechend im Spielbaum berücksichtigt sein. Für alle anderen Spiele sind damit einige Anpassungen an dem hier vorgestellten Algorithmus notwendig.

3.5 Weitere Iterationen

Da die beschriebenen vier MCTS-Schritte erfolgreich durchgeführt wurden, ist damit zugleich die Iteration des Beispiels abgeschlossen. Abbildung 6 präsentiert den in diesem Rahmen aktualisierten Monte-Carlo-Suchbaum.
An dieser Stelle muss sich der MCTS-Algorithmus für eine der folgenden zwei Optionen entscheiden:

1. Eine weitere MCTS-Iteration durchführen, um den Lookahead zu vergrößern.
2. Die unter Berücksichtigung des aktuellen Lookaheads sinnvollste Aktion ausgehend vom Zustand s_0 an die aufrufende Umgebung zurückgeben.

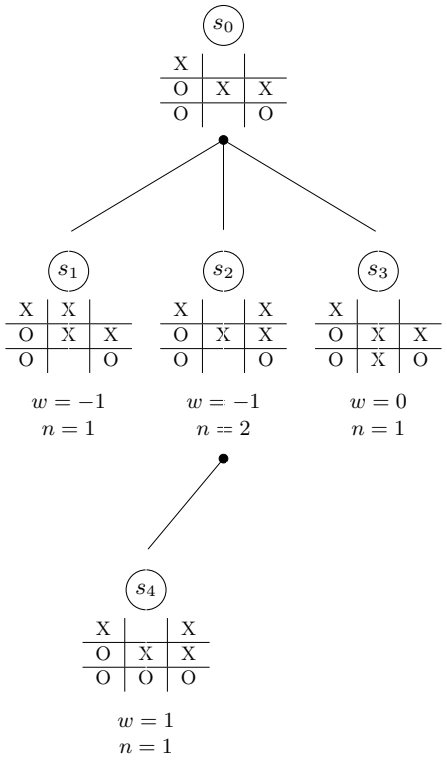

Abbildung 6 Aktualisierter Monte-Carlo-Suchbaum

Sofern die für den MCTS-Algorithmus konfigurierte Iterationszahl nicht erreicht und auch ein gegebenes Zeitlimit nicht überschritten ist, wird im Folgenden die erste Option, eine weitere MCTS-Iteration durchzuführen, gewählt. Dadurch wird der Monte-Carlo-Suchbaum weiter expandiert, wodurch die daraus abgeleitete Spielstrategie langfristig besser wird. In diesem Fall knüpft die nächste Iteration an den aktualisierten Suchbaum aus Abbildung 6 an.
Mit der zweiten Option, keine weitere Iteration durchzuführen, wird basierend auf dem aufgebauten Monte-Carlo-Suchbaum die für Spieler X als Nächstes durchzuführende Aktion ausgehend vom Zustandsknoten s_0 bestimmt. Diese Entscheidung basiert darauf, die Aktion zu wählen, welche in dem zielführendsten Kindknoten resultiert. Dabei kann dieser Kindknoten sowohl danach bewertet werden, ob er die Anzahl der Selektionen n_i oder alternativ den Durchschnittswert $\frac{w_i}{n_i}$ maximiert.

4 Spiel-Komplexität

4.1 Komplexität hinter Go

Lange Zeit galt das Brettspiel Go als eine der größten Herausforderungen für nichtmenschliche Spieler [6, S. 484], was unter anderem durch dessen astronomischen Suchraum bedingt war. So spielten zwar KI-Agenten wie zum Beispiel das Programm PACHI, welches im Jahr 2012 zu den leistungsfähigsten Open-Source-Programmen im Spiel Go zählte [29, S. 24], auf starkem Amateur-Level [29, S. 36], doch war es noch keinem Algorithmus gelungen, unter ausgeglichenen Bedingungen einen menschlichen Profispieler auf einem 19x19 Spielbrett zu besiegen.
Theoretisch lässt sich für jedes deterministische Spiel mit vollständigem Informationsgehalt durch die Traversierung des gesamten Spielbaums eine perfekte Spielstrategie finden [6, S. 484]. Dafür kommt beispielsweise der Minimax-Algorithmus zum Einsatz, welcher in Bezug auf seine Laufzeitkomplexität zusätzlich durch Alpha-Beta-Pruning optimiert werden kann. Auf der anderen Seite zeigt sich in der Praxis, dass der enorme Rechenbedarf nicht aufgebracht werden kann, um derartige Algorithmen auf die hochkomplexen Suchbäume von Spielen wie Go anzuwenden [25, S. 159]. Die Quantität der Knoten eines zu diesem Zweck benötigten Spielbaums lässt sich üblicherweise mit der Formel b^d bestimmen [6, S. 484]. Dabei handelt es sich bei b um die Anzahl an möglichen Spielzügen pro Runde und bei d um die Spieltiefe.
So weist bereits ein Spiel niedriger Komplexität wie Tic-Tac-Toe 255.168 verschiedene Spielverläufe auf [27, S. 45]. Im Fall von Tic-Tac-Toe lässt sich diese Zahl nicht ohne Weiteres durch b^d kalkulieren, da die Anzahl an möglichen Zügen b nicht konstant ist, sondern jede Runde um eins abnimmt. Demnach berechnen sich die 255.168 legitimen Spielverläufe durch $9! - \mathit{ungueltigePartien}$, wobei $\mathit{ungueltigePartien}$ die nicht validen Spielpartien beschreibt. Letztere sind beispielsweise gegeben, wenn eine entschiedene Spielpartie, anstatt zu terminieren, fortgeführt wird. Abbildung 7 zeigt, wie unübersichtlich ein solcher Spielbaum für das im Vergleich zu Go triviale Spiel Tic-Tac-Toe bereits nach drei von neun maximalen Ebenen ist.

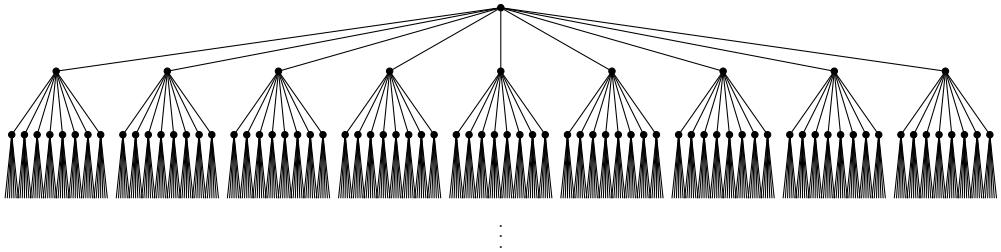

Abbildung 7 Die ersten drei Ebenen des Spielbaums von Tic-Tac-Toe

Im Gegensatz zu Tic-Tac-Toe findet Go nicht auf einem 3x3, sondern 19x19 Spielfeld statt, was einen enormen Komplexitätszuwachs bedingt.
Auch wenn gewisse Spielbrettzustände aus unterschiedlichen Spielverläufen resultieren können, müssen diese Zustände stets im Kontext der dazugehörigen Spielverläufe betrachtet werden. Daraus folgt, dass sich die etwa 10^{172} in Go möglichen Spielbrettkonfigurationen durch 10^{360} verschiedene Spielverläufe erreichen lassen [4, S. 174].

4.2 Umgang mit astronomischen Spielbäumen

DeepMinds Computerprogramm AlphaGo sorgte mit seinen Triumphen über Fan Hui (2. Professional Dan) [6, S. 488] im Jahr 2015 und Lee Sedol (9. Professional Dan und damit der höchste erreichbare Rang im Spiel Go) [2] im Jahr 2016 insofern für Verblüffung, als dass Experten den ersten Sieg einer Software gegen einen menschlichen Profispieler ohne Handicap im Spiel Go auf einem 19x19 Feld erst eine ganze Dekade später erwartet haben [6, S. 488].

Diese Experteneinschätzung ist nicht zuletzt durch die im Kapitel 4.1 beschriebene enorme Komplexität begründet. Somit erfordert das Ableiten einer vielversprechenden Spielstrategie aus einem astronomischen Go-Spielbaum, den Suchraum effektiv zu reduzieren. Das erreichten Silver et al. im Rahmen der AlphaGo-Entwicklung, indem sie von den zwei nachfolgend beschriebenen Ansätzen [6, S. 484] Gebrauch machten:

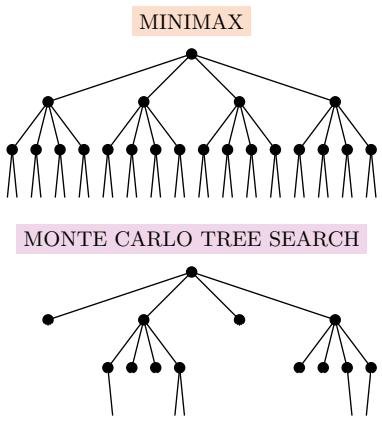

Abbildung 8 Eingrenzung der Suchbreite

1. Eingrenzung der Suchbreite
Abbildung 8 visualisiert die Möglichkeit, den zu durchsuchenden Raum in seiner Breite einzuschränken. So stellt diese zwei Arten einen Baum zu traversieren gegenüber. Der obere Suchbaum der Abbildung ist in seinen ersten drei Ebenen vollständig expandiert, da er durch einen Minimax-Algorithmus traversiert wurde. Der untere dagegen wurde durch die Anwendung einer Monte-Carlo-Baumsuche lediglich partiell expandiert. Auch wenn das Ergebnis des aufwendigeren Minimax-Algorithmus stets optimal sein wird [24, S. 124], so nähert sich das Resultat einer MCTS mit zunehmender Iterationszahl dem einer Minimax-Suche an [26, S. 4]. Dabei nimmt die praktische Umsetzbarkeit der Minimax-Suche ohnehin mit einer Skalierung der Komplexität ab, wodurch die Monte-Carlo-Baumsuche eine ressourceneinsparende Alternative darstellt. So sind im Beispiel verglichen mit der MCTS durch die Minimax-Suche mehr als das Dreifache an Folgezuständen expandiert worden.

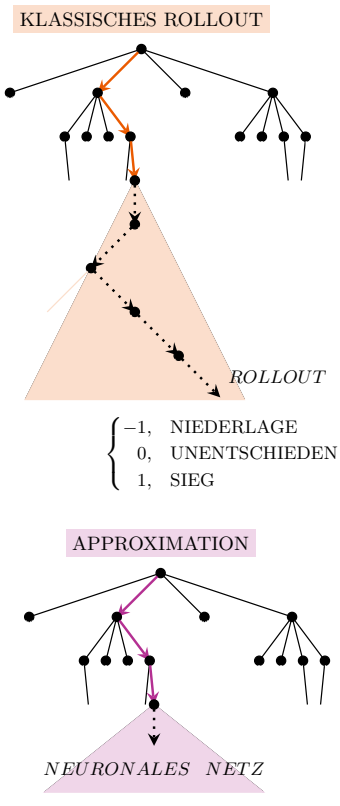

Abbildung 9 Eingrenzung der Suchtiefe

2. Eingrenzung der Suchtiefe

Neben der Breite kann ein Suchbaum zusätzlich in seiner Tiefe eingeschränkt werden. Dies kann beispielsweise erreicht werden, indem dieser ab einer bestimmten Suchtiefe T "gestutzt" wird, sodass die Bewertungen von Spielzuständen betroffener Zustandsknoten auf Ebene T durch entsprechende Funktionen approximiert werden. Das bedeutet, dass es nicht mehr notwendig ist, den Suchbaum ausgehend von einem solchen Knoten s_T bis zu den Baumblättern zu durchlaufen, um dort einen terminierenden Spielzustand vorzufinden, auf dessen Basis ein Wert für s_T abgeleitet werden kann.

Abbildung 9 visualisiert, wie so eine Eingrenzung der Suchtiefe aussehen kann. Die Illustration knüpft an den Suchbaum des vorherigeren Beispiels aus Abbildung 8 an, welcher zuvor im Rahmen einer Monte-Carlo-Baumsuche in seiner Breite eingegrenzt wurde. Der obere Baum in Abbildung 9 zeigt, wie klassischerweise die Fortführung dieser Monte-Carlo-Baumsuche durch ein Rollout aussieht. Dabei werden solange weitere Spielzüge simuliert, bis ein Endzustand erreicht wird. Abhängig von dem Spielresultat dieses Endzustands wird eine Bewertung für den "ausgerollten" Knoten abgeleitet. Dieses Verfahren bedarf vor allem bei riesigen Suchbäumen erhöhte Rechenleistung, da in jeder MCTS-Iteration ein Rollout ausgehend von dem zuvor expandierten Zustandsknoten durchgeführt wird.

Dieser Rollout-Prozess kann alternativ durch eine approximierende Funktion ersetzt werden [6, S. 484]. Diese kann sich beispielsweise auf Heuristiken stützen oder, wie im unteren Beispiel von Abbildung 9, auch auf künstlichen neuronalen Netzen aufbauen. Bei Letzteren handelt es sich um universale Funktionsapproximatoren, welche im Rahmen der Neuroinformatik erforscht werden und in Anlehnung an die "[...] biologischen Vernetzungen im Gehirn von Säugetieren [...]" [23, S. 18] die Lern- und Denkprozesse eines solchen Gehirns simulieren [5, S. 13].

Eine MCTS basiert auf Statistik, wodurch die Bewertungen von Spielzuständen erst bei einer Vielzahl von Iterationen an Aussagekraft gewinnen. Vor allem wenn die zugrunde liegenden Spiele eine hohe Komplexität aufweisen, bedarf es einer umso größeren Iterationsanzahl. Der dafür notwendige Rechenaufwand kann durch die Verwendung eines neuronalen Netzes reduziert werden. Dieses benötigt zwar eine rechenaufwendige Trainingsprozedur, um die Bewertung von Spielzuständen zu erlernen, doch muss ein solches Training lediglich einmal durchgeführt werden. Abhängig von der Komplexität des neuronalen Netzes kann es im Anschluss die Approximation von Zustandswerten möglicherweise bei einem, verglichen mit einem MCTS-Rollout, geringeren Bedarf an Rechenleistung durchführen. Darüber hinaus würde es bei einem Spiel wie Go eine enorme Anzahl an Random-Rollouts erfordern, um überhaupt einigermaßen aussagekräftige Ergebnisse zu erhalten. Wohingegen ein neuronales Netz lernen kann komplexe Spiele bereits in frühen Spielphasen zielführend zu bewerten.

5 Entwicklung von AlphaGo zu AlphaZero

5.1 AlphaGo

Die beiden im vorherigen Kapitel beschriebenen Ansätze zum Umgang mit hochkomplexen Suchbäumen boten dem Team von DeepMind das Grundgerüst für die Entwicklung von AlphaGo [6, S. 484]. Ein Programm, welches im Jahr 2016 in der südkoreanischen Hauptstadt Seoul im Rahmen eines Go-Turniers den professionellen Spieler Lee Sedol besiegte [2]. AlphaGo besteht aus der Kombination von tiefen neuronalen Netzen und einer Monte-Carlo-Baumsuche [6, S. 484]. Letztere reduziert den Suchraum effektiv in seiner Breite. Bei den verwendeten neuronalen Netzen handelt es sich konkret um DCNNs. Diese sind vor allem in visuellen Domänen wie der Klassifizierung von Bildern sehr leistungsfähig [30, S. 84]. AlphaGo verfügt über zwei solche DCNNs. Bei einem davon handelt es sich um ein Policy-Netzwerk p_σ, welches darauf trainiert ist, für die Eingabe eines Spielzustands s einen Vektor der Wahrscheinlichkeiten möglicher Spielzüge $\vec{p}$ zurückzugeben [6, S. 485]. Anhand von $\vec{p}$ lassen sich also Aussagen darüber treffen, wie zielführend die ausgehend von s verfügbaren Züge vom Netzwerk jeweils eingeschätzt werden. Damit nimmt p_σ als Policy Einfluss auf die Selektion der Monte-Carlo-Baumsuche [6, S. 486]. Das zweite DCNN stellt ein Value-Netzwerk v_θ dar und hat die Aufgabe, für die Eingabe eines Spielzustands s eine Ausgabe v zu erzeugen. Bei Letzterer handelt es sich um eine skalare Bewertung von s aus der Sicht des Spielers, welcher in diesem Spielzustand an der Reihe ist [6, S. 485]. Dieser approximierte Wert v wird im Simulationsschritt der Monte-Carlo-Baumsuche zur Zustandsbewertung herangezogen. Nichtsdestotrotz findet in diesem Schritt zusätzlich ein Rollout statt, welches jedoch nicht zufällig ist, sondern auf einer schnellen und trainierten Rollout-Policy p_π basiert. Das Ergebnis des Rollouts fließt neben dem approximierten Wert v in die Bewertung eines Spielzustands ein [6, S. 486].

Training

Der Trainingsprozess hinter AlphaGo setzt neben Supervised- auch auf Reinforcement-Learning. Konkret sieht das Verfahren dabei so aus, dass im Rahmen von Supervised-Learning eine schnelle Rollout-Policy p_π und ein Policy-Netzwerk p_σ darauf trainiert werden, Spielzüge menschlicher Experten vorherzusagen [6, S. 484-485].
Anschließend wird ein in Struktur und Gewichten zu p_σ identisches Policy-Netzwerk p_p initialisiert [6, S. 485], welches damit den Trainingsfortschritt von p_σ Züge menschlicher Spieler zu prognostizieren übernimmt. Basierend auf einem Reinforcement-Learning-Ansatz spielt p_p im folgenden Spiele gegen sich selbst, wobei das Erringen von Siegen belohnt wird [6, S. 485]. Durch den Einfluss von p_σ führt p_p initial Züge durch, von denen es glaubt, dass ein Mensch diese wählen würde. Im Lernprozess ändert das Netzwerk seine Strategie jedoch dahingehend, sich für Aktionen zu entscheiden, von denen es gelernt hat, dass sie die Wahrscheinlichkeit eines langfristigen Siegs begünstigen [6, S. 484]. Die durch die Partien von p_p gegen sich selbst generierten Spielpositionen werden samt der daraus resultierenden Spielergebnisse gespeichert. Dadurch ist es dem Value-Netzwerk v_θ möglich, diese Daten im Nachhinein zu durchlaufen und auf deren Basis zu lernen, Spielbrettzustände dahingehend zu bewerten, ob sie zu einem Sieg führen oder nicht [6, S. 485-486].

5.2 AlphaGo Zero

Als direkter Nachfolger von AlphaGo setzt AlphaGo Zero vollständig auf Reinforcement-Learning [7, S. 1]. Damit entwickelt das Programm seine Spielstrategien eigenständig, wodurch diesen keine menschlichen Einflüsse zugrunde liegen. Der Algorithmus lernt Go also ohne Vorkenntnisse ausschließlich durch das Spielen gegen sich selbst. So wählt das Programm Spielzüge in den ersten Trainingsiterationen bedingt durch die mangelnde Erfahrung rein zufällig und verbessert sich kontinuierlich im weiteren Verlauf des Trainings [7, S. 2].

5.2.1 Architektonische Unterschiede zu AlphaGo

Dem Algorithmus AlphaGo Zero liegt im Gegensatz zu seinem Vorgänger lediglich ein tiefes neuronales Netzwerk f_θ zugrunde [7, S. 2]. Dieses gibt für die Eingabe eines Spielzustands s sowohl eine Policy $\vec{p}$ als auch eine skalare Bewertung v des Zustands s zurück [7, S. 3]. Es gilt $f_\theta(s) = (\vec{p}, v)$. Damit löst f_θ sowohl das Policy-Netzwerk p_σ von AlphaGo als auch dessen Value-Netzwerk v_θ ab. Zugleich wird das Netzwerk p_p überflüssig, welches in der Vorgängerversion unter anderem dafür genutzt wurde, Selbstspieldaten zu generieren, die für das Training von v_θ verwendet wurden. Zwar äußerte sich die Entscheidung, lediglich eins statt zwei neuronale Netze einzusetzen, in einer verschlechterten Vorhersage von Spielzügen, doch wirkte sie sich positiv auf die Spielstärke aus [7, S. 8].

5.2.2 Training

Auch die Trainingsprozedur von AlphaGo Zero wurde angepasst. Einerseits muss lediglich das neuronale Netzwerk f_θ trainiert werden, andererseits basiert dieser Prozess auf einem neuartigen Trainingsalgorithmus des Reinforcement-Learnings [7, S. 3]. Der Algorithmus bezieht eine Monte-Carlo-Baumsuche mit ein, wodurch eine Wechselwirkung zwischen dieser und dem trainierten Netzwerk entsteht. Das Training besteht aus den drei im Folgenden vereinfacht beschriebenen Prozessen, welche asynchron zueinander gleichzeitig durchgeführt werden [7, S. 23-24]:

1. **Generierung von Selbstspieldaten durch das stärkste Netzwerk f_{θ_*}**
   Das stärkste neuronale Netz der Trainingsprozedur f_{θ_*} spielt in jeder Iteration 25.000 Partien gegen sich selbst [7, S. 24], wobei f_{θ_*} zu Beginn des Trainings durch das zu trainierende Netzwerk f_θ initialisiert wird. Im Rahmen des Selbstspiels wird in jedem Spielzustand s_t eine von f_{θ_*} geleitete [7, S. 3] Monte-Carlo-Baumsuche mit 1.600 Simulationen ausgeführt [7, S. 24]. Durch die MCTS entsteht für jeden Knoten s_t eine Wahrscheinlichkeitsverteilung $\vec{\pi}_t$, welche in ihrer Aussagekraft zwar durch das neuronale Netz f_{θ_*} geprägt ist, doch im Endeffekt eine deutlich stärkere Policy darstellt als dessen Wahrscheinlichkeitsverteilung $\vec{p_t}$ [7, S. 3].

   Während eines Selbstspiels wird für jeden Spielzustand s_t ein Datensatz der Form $(s_t, \vec{\pi}_t, z_t)$ gespeichert [7, S. 6]. Das bedeutet, dass zu jedem Zustand s_t auch dessen durch die MCTS verbesserte Policy zur Auswahl der nächsten Aktion $\vec{\pi}_t$ festgeschrieben wird. Zudem wird am Ende eines Selbstspiels der Wert z_t für alle hinterlegten

Datensätze der entsprechenden Partie gesetzt. Das hat den Grund, dass das Ergebnis einer Partie $z_t \in \{-1, +1\}$ erst am Ende bekannt ist und somit den zugehörigen Spielzuständen nicht vorher zugeordnet werden kann [7, S. 6].

2. **Optimierung der Gewichte von f_{θ_i}**
   In diesem Prozess werden zufällig Stichproben aus den Selbstspieldaten der letzten 500.000 Spiele von f_{θ_*} gegen sich selbst entnommen [7, S. 23-24]. Diese Trainingsdatensätze haben jeweils die Form $(s_t, \vec{\pi}_t, z_t)$. Für die Eingabe von s_t liefert das stärkste Netzwerk die Ausgabe $f_{\theta_*}(s_t) = (\vec{p}_t, v_t)$.

   Im Rahmen eines Trainingsschritts werden die Gewichte des Netzwerks durch ein stochastisches Gradientenverfahren so optimiert [7, S. 24], dass unter Eingabe des Zustands s_t einerseits die ausgegebene Policy $\vec{p}_t$ des Netzwerks an die durch die Monte-Carlo-Baumsuche verbesserte Policy $\vec{\pi}_t$ sowie die Ausgabe v_t an das Resultat des Selbstspiels z_t angenähert werden [7, S. 3]. Dabei wird nach jeweils 1.000 Trainingsschritten ein Zwischenstand f_{θ_i} des trainierten Netzes erzeugt [7, S. 24].

3. **Evaluation der aktualisierten Netzwerke f_{θ_i}**
   Jeder neue Zwischenstand eines trainierten Netzwerks f_{θ_i} spielt zur Evaluation 400 Spiele gegen das aktuell stärkste neuronale Netz f_{θ_*} [7, S. 24], wobei zur Auswahl eines Spielzugs jeweils 1.600 MCTS-Simulationen durchgeführt werden. Sofern f_{θ_i} eine Siegesrate von mehr als 55 % aufweisen kann, wird es zum neuen stärksten Netzwerk f_{θ_*} [7, S. 24].

5.2.3 Evaluation im direkten Vergleich mit AlphaGo

Die Version von AlphaGo, welche den Profispieler Lee Sedol besiegte, wird als AlphaGo Lee bezeichnet. Um die Spielstärke von AlphaZero zu evaluieren, ließ das DeepMind-Team das Programm in 100 Spielen gegen seinen Vorgänger AlphaGo Lee antreten. In diesen Partien galten die gleichen Bedingungen wie bei dem Turnier im Jahr 2016 zwischen AlphaGo Lee und Lee Sedol [7, S. 8].
Bereits nach einer Trainingszeit von etwa 72 Stunden war AlphaGo Zero dazu im Stande, seinen Vorgänger mit einem Spielstand von 100 zu 0 zu besiegen [7, S. 8]. Vergleichsweise war für AlphaGo Lee ein Training über mehrere Monate hinweg notwendig [7, S. 8]. Während der Evaluation lief AlphaGo Zero auf einer Maschine mit 4 TPUs, wohingegen AlphaGo Lee ein verteiltes System mit insgesamt 47 TPUs beanspruchte [7, S. 8]. Diese Resultate zeigten, dass Reinforcement-Learning-Ansätze auch in hochkomplexen Anwendungsfällen wirkungsvoll einsetzbar sind und sogar zu besseren Ergebnissen führen können als Methoden des Supervised-Learnings.

5.3 AlphaZero

AlphaGo Zero hat bewiesen, dass außerordentliche Erfolge selbst ohne die Berücksichtigung menschlichen Expertenwissens erreicht werden können. Demnach ist der Gedanke nicht weit entfernt, das dahinterliegende Konzept auch spieleübergreifend einzusetzen. Daraus resultierte DeepMinds nächster Schritt, den Algorithmus hinter AlphaGo Zero zu generalisieren, sodass dieser nicht mehr auf die Go-Domäne beschränkt ist. Es schlug die Geburtsstunde von AlphaZero.

5.3.1 Grundsätzliche Unterschiede zu AlphaGo Zero

Aufgrund seiner Allgemeingültigkeit unterscheidet sich der Algorithmus AlphaZero in einigen Grundsätzen von AlphaGo Zero.
Nennenswert ist dabei, dass AlphaGo Zero die Wahrscheinlichkeit eines Siegs maximiert. Denn im Go ist es möglich, die Spielregeln so auszulegen, dass aus der Sicht eines Spielers stets ein binäres Ergebnis erwartet werden kann, ein Sieg oder eine Niederlage. Um auch solche Spiele abzudecken, die in einem Unentschieden resultieren können, maximiert AlphaZero nicht die Siegeswahrscheinlichkeit, sondern das erwartete Spielergebnis [12].
Des Weiteren nutzt AlphaGo Zero Spielbrettsymmetrien aus [12]. Solche Symmetrien sind beispielsweise dann gegeben, wenn das Spiegeln oder Rotieren eines Spielbretts in validen Spielzuständen resultiert. Abbildung 10 zeigt Symmetrien eines Othello-Spielbretts, welche durch Rotation entstanden sind. Dabei sei erwähnt, dass einige dieser Spielzustände nicht von der üblichen Othello-Startaufstellung aus zustande kommen können, was aber nichts daran ändert, dass es sich dabei um valide Spielzustände in dem Sinne handelt, als das entsprechende Partien ausgehend von diesen Zuständen regelkonform fortgeführt werden können.

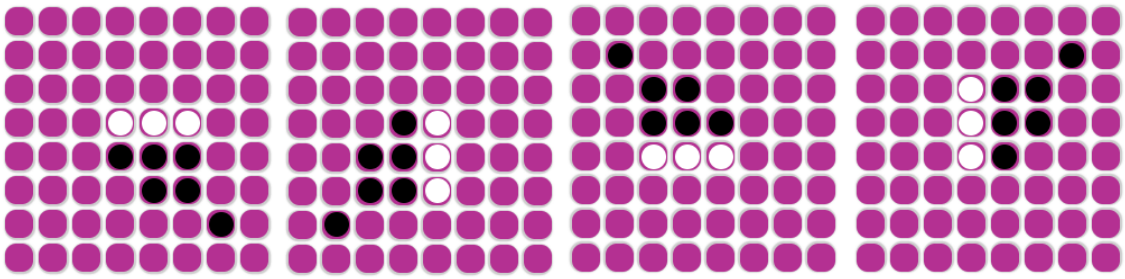

Abbildung 10 Rotationen eines Othello-Spielbretts

Durch die Generierung symmetrischer Spielzustände kann die Menge der Trainingsdaten signifikant erhöht werden. Dieses Vorgehen ist bei dem generischen Ansatz von AlphaZero nicht weiter möglich, da der Algorithmus nicht von Symmetrien ausgehen kann. So lässt sich ein Schachbrett zum Beispiel nicht um 90° drehen, da Bauern sich nur vorwärts bewegen können.

5.3.2 Training

Der Trainingsprozess von AlphaZero unterscheidet sich nicht allzu sehr von dem der Vorgängerversion AlphaGo Zero (vgl. Kapitel 5.2.2). Ein wesentlicher Unterschied besteht darin, dass der Prozess zur *Evaluation der aktualisierten Netzwerke* f_{θ_i} wegfällt. Das liegt daran, dass im Trainingsprozess von AlphaZero lediglich eine Version des neuronalen Netzes f_θ existiert. Das Netz f_θ spielt durchgehend gegen sich selbst, während seine Gewichte kontinuierlich aktualisiert werden [12].

5.3.3 Evaluation im direkten Vergleich mit Weltmeister-Programmen

Um die Leistungsfähigkeit von AlphaZero zu validieren, trat der Algorithmus in den Spielen Schach, Shogi und Go gegen Weltmeister-Programme an. Vor allem die Vergleiche mit den Schach- und Shogi-Programmen waren aus dem Grund interessant, als das zu diesem Zeitpunkt solche Programme in der Regel aus Alpha-Beta-Suchen bestanden, welche durch domänenspezifische Adaptionen optimiert wurden [12].
Im Schach spielte AlphaZero gegen die Stockfish-Version, welche im Jahr 2016 den ersten Platz im Top Chess Engine Championship einnahm. Von 1.000 Spielpartien gingen dabei sechs Siege an Stockfish und 155 an AlphaZero. Die restlichen Partien endeten im Unentschieden [12].
Bei Shogi handelt es sich um eine japanische Version des Schachspiels. In diesem Spiel gewann AlphaZero 91.2 % der Partien gegen das Programm Elmo, welches im Jahr 2017 den ersten Platz des durch die Computer Shogi Association veranstalteten World Computer Shogi Championships einnahm [12].
Auch im Go konnte sich AlphaZero durchsetzen. Hier trat der Algorithmus gegen die aktuellste Version seines Vorgängers AlphaGo Zero an. Beide Spielagenten haben vor dieser Evaluation jeweils 700.000 Trainingsiterationen durchlaufen. Der Vergleich resultierte in einer 61-prozentigen Siegesrate für AlphaZero [12]. Dieses Ergebnis ist insofern aufschlussreich, als das es zeigt, dass ein generischer Ansatz sich selbst dann gegen einen domänenspezifischen Algorithmus durchsetzen kann, wenn Letzterer durch das Ausnutzen von Spielbrettsymmetrien Trainingsdaten augmentiert.

6 Implementierung eines AlphaZero-inspirierten Agenten

In diesem Kapitel wird die Entwicklung eines Agenten dokumentiert, welcher die grundlegenden Prinzipien von Alpha-Zero adaptiert. Dieser wird architektonisch in Form eines MCTS-Wrappers für das GBG-Framework umgesetzt und basiert wie auch AlphaZero auf einer Monte-Carlo-Baumsuche. Ein Unterschied besteht jedoch darin, dass Bewertungen von Spielzuständen nicht durch ein neuronales Netz approximiert werden, sondern durch einen gegeben Spielagenten des GBG-Framework. Nichtsdestotrotz wird die Software-Architektur diese Approximation durch ein Interface abstrahieren, sodass in zukünftigen Anwendungsfällen durchaus ein neuronales Netz zu diesem Zweck herangezogen werden kann.
Das Ziel dieses Kapitels besteht in der Vermittlung grundlegender Konzepte der Implementierung. Dazu werden Algorithmen durch Pseudocode dargestellt. Diese Darstellungsform hat nicht den Anspruch, den zugrunde liegenden Java-Quellcode identisch nachzubilden. So werden beispielsweise mehrzeilige Anweisungsblöcke, triviale Hilfsmethoden oder auch programmiersprachenspezifische Syntaxelemente der tatsächlichen Implementierung vereinfacht dargestellt, sofern dadurch neben der Lesbarkeit des Quellcodes zugleich die Nachvollziehbarkeit der angewandten Konzepte verbessert wird.
Im Gegenzug ist der im Rahmen dieses Kapitels entstandene lauffähige Java-Quellcode als Teil des quelloffenen GBG-Framework unter `https://github.com/WolfgangKonen/GBG` abrufbar. In dem entsprechenden Repository befindet sich dieser hauptsächlich in dem Verzeichnis `src\controllers\MCTSWrapper`.

6.1 Anforderungen

Abbildung 11 visualisiert die vier wesentlichen Anforderungen an den zu implementierenden MCTS-Wrapper-Agenten.

1. Der modellierte Anwendungsfall wird initiiert, indem die Spielumgebung für einen bestimmten Spielzustand s die nächste durchzuführende Spielaktion des Agenten anfragt. Das erfordert eine Integration des MCTS-Wrapper-Agenten in die GBG-Spielumgebung.

2. Bereits der in Kapitel 3.4 beschriebene MCTS-Algorithmus war im Rahmen der Backpropagation darauf angewiesen, dass Spielzustände, in denen ein Spieler passen muss, explizit als Spielbaumknoten modelliert sind.

   So erwähnen auch Silver et al. [7, S. 3] diese Anforderung in ihrer Publikation zu AlphaGo Zero.

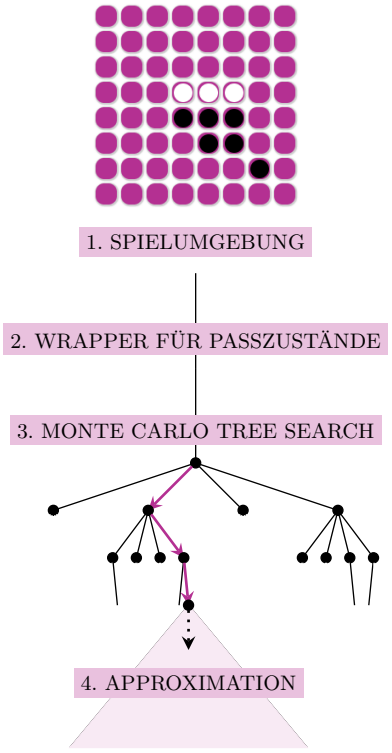

Abbildung 11 Konzept des MCTS-Wrappers

Dem entgegen zeigt sich im GBG-Framework zum aktuellen Zeitpunkt das Verhalten, dass Spielzustände mit Passsituationen implizit übersprungen werden.
Führt im Spiel Othello beispielsweise Spieler b eine Aktion aus, die dafür sorgt, dass der Gegenspieler w in der nächsten Runde keine Spielaktion durchführen kann, so ist nach dem Spielzug erneut b an der Reihe. Dieses Überspringen von Passzuständen ist notwendig, weil das Framework keine expliziten Passaktionen modelliert, die das Verlassen solcher Zustände ermöglichen.
Der MCTS-Wrapper-Agent generiert die Zustandsknoten des Monte-Carlo-Suchbaums durch das Simulieren von Spielaktionen. Hierbei wird aus den zuvor beschriebenen Gründen eine Komponente benötigt, welche dafür sorgt, dass Passsituationen nicht mehr übersprungen werden. Stattdessen sollen diese ein Teil des aufgebauten Suchbaums werden. Das impliziert zugleich die Notwendigkeit von Passaktionen, dessen Anwendung auf einen Passzustand s in einem Folgezustand s' resultiert.

3. Nachdem sichergestellt ist, dass Passzustände nicht weiter übersprungen werden, wird eine Monte-Carlo-Baumsuche benötigt, welche auf dieser Basis einen Suchbaum aufbaut, der auch entsprechende Knoten für Passzustände aufweist.

4. Des Weiteren bedarf der MCTS-Wrapper-Agent einer Software-Komponente zur Approximation der Bewertungen von Spielzuständen. Diese ersetzt im Simulationsschritt der Monte-Carlo-Baumsuche die Random-Rollouts.

Die nächsten vier Unterkapitel beschäftigen sich jeweils mit der Implementierung von Software-Komponenten, welche in ihrer Gesamtheit den MCTS-Wrapper-Agenten repräsentieren und die vier in diesem Kapitel beschriebenen Anforderungen an diesen Agenten sicherstellen.

6.2 Integration in die GBG-Umgebung

Dieser Abschnitt befasst sich mit der Integration des MCTS-Wrapper-Agenten in das GBG-Framework sowie dessen grafische Benutzeroberfläche.

Abbildung 12 zeigt beispielhaft die Arena-Ansicht des Framework für das Brettspiel Othello. Aus dieser Ansicht heraus können sowohl lernfähige KI-Agenten im Spiel Othello trainiert als auch Spielpartien zwischen verschiedenen Agenten organisiert werden. Dazu stellt das Framework zahlreiche Konfigurationsmöglichkeiten in der Benutzeroberfläche zur Verfügung. Der Abbildung lässt sich entnehmen, dass ein TD-3-Tupel-Agent als schwarzer und der Agent Edax als weißer Spieler ausgewählt sind.

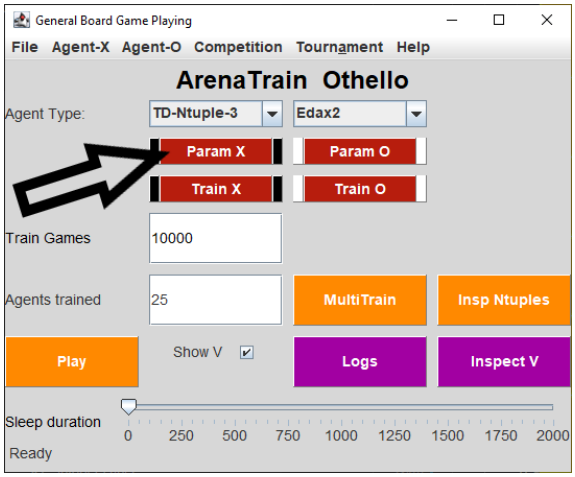

Abbildung 12 Auswahl der Agenten-Parameter in der Arena-Ansicht des Spiels Othello

Der Grundgedanke hinter dem MCTS-Wrapper-Agenten sieht vor, dass dieser einen beliebigen GBG-Agenten ummanteln kann. Dadurch wird es möglich, Letzteren im Rahmen der durchgeführten MCTS-Iterationen zur Evaluierung von Spielzuständen heranzuziehen.

Um beispielsweise für den ausgewählten TD-3-Tupel-Agenten den MCTS-Wrapper zu aktivieren, kann der in Abbildung 12 markierte, mit ”Param X” bezeichnete Button selektiert werden. Dieser öffnet ein Fenster, in dem der entsprechende Spielagent X konfiguriert werden kann (vgl. Abbildung 13). Unter dem Reiter ”Other pars” lässt sich anschließend der Parameter ”Wrapper MCTS” mit einem Wert $n \in \{0, 1, 2, 3, ...\}$ belegen, welcher die Anzahl der durchzuführenden MCTS-Iterationen vorgibt. Damit wird der MCTS-Wrapper automatisch verwendet, sofern eine positive Iterationszahl hinterlegt ist.

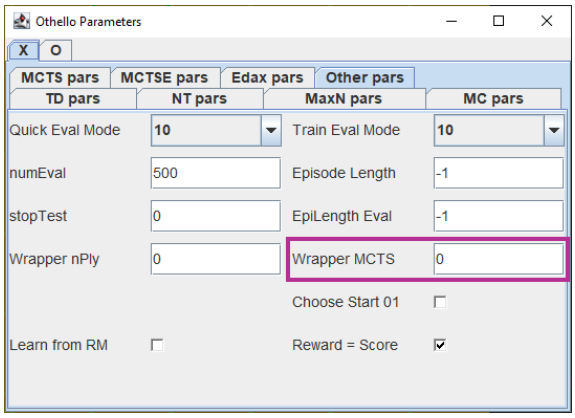

Abbildung 13 Konfiguration des MCTS-Wrapper-Agenten in der grafischen GBG-Benutzeroberfläche

Algorithmus 1: Vereinfacht dargestellte Implementierung der von *PlayAgent* geerbten getNextAction2-Methode durch den MCTS-Wrapper-Agenten.

Input : StateObservation-Instanz *sob*.
Output: Die nächste ausgehend von *sob* durchzuführende Spielaktion.

```
1  Function getNextAction2(sob):
2      if searchTree is not initialized then
3          node ← new MCTSNode(new GameStateIncludingPass(sob))
4      else
5          node ← searchTree.getNodeFor(sob)
6      end
7
8      for i ← 1 to iterationCount do
9          monteCarloTreeSearch(node)
10     end
11
12     nextAction ← node.actionThatMaximizesVisitCount()
13     searchTree ← node.getChild(nextAction)
14
15     return nextAction
16 End Function
```

Damit eine Klasse im GBG-Framework als Spielagent gelten kann, muss diese das Interface *PlayAgent* implementieren. Um diesen Prozess zu vereinfachen, stellt das Framework die abstrakte *AgentBase*-Klasse zur Verfügung, welche bereits einen Großteil der von *PlayAgent* vorausgesetzten Funktionalitäten implementiert. So müssen von *AgentBase* erbende Klassen lediglich die Methoden *getNextAction2* und *getScore* implementieren.
Der MCTS-Wrapper-Agent wird durch die von *AgentBase* abgeleitete Klasse *MCTSWrapperAgent* repräsentiert. Zur Implementierung der *getScore*-Methode delegiert diese den Aufruf lediglich an den ummantelten Spielagenten. Außerdem muss der Agent eine *getNextAction2*-Methode bereitstellen, dessen Umsetzung vereinfacht in dem obigen Algorithmus 1 dargestellt und nachfolgend erläutert wird.
Im ersten Schritt (vgl. Abbildung 14) prüft die Methode, ob bereits ein Monte-Carlo-Suchbaum aufgebaut wurde. So ist zu Beginn einer Partie zunächst kein Suchbaum vorhanden, da im Rahmen des Spiels noch keine MCTS-Iterationen stattfinden konnten. In diesem Fall wird ein neuer Suchbaumknoten, welcher den gegeben Spielzustand *sob* repräsentiert erzeugt. Die Implementierung der dazu verwendeten *MCTSNode*-Klasse wird in Kapitel 6.4.2 beschrieben. Um sicherzustellen, dass auch Spielzustände, in denen ein Spieler passen muss, berücksichtigt werden, wird zu diesem Zweck eine Instanz der Klasse *GameStateIncludingPass* verwendet, dessen Funktionsweise in Kapitel 6.3 dokumentiert ist.

```
if searchTree is not initialized then
 | node ← new MCTSNode(new GameStateIncludingPass(sob))
else
 | node ← searchTree.getNodeFor(sob)
end
```

Abbildung 14 Zeilen 2-6 der *getNextAction2*-Implementierung

Sofern bereits ein zwischengespeicherter Suchbaum existiert, ist das Erzeugen eines den Spielzustand *sob* repräsentierenden Zustandsknotens gegebenenfalls nicht notwendig, wenn dieser aus dem vorhandenen Suchbaum extrahiert werden kann. Anderenfalls war dieser Knoten in vorangegangenen MCTS-Iterationen nicht vielversprechend genug, um expandiert zu werden. In diesem Fall wird dieser als Instanz der Klasse *MCTSNode* erzeugt. Die Zwischenspeicherung des Suchbaums über ein Spiel hinweg hat den Vorteil, dass dieser nicht bei jedem Aufruf von *getNextAction2* von Grund auf neu aufgebaut werden muss. Stattdessen kann er in jeder Runde stetig erweitert werden. Damit liegen dem Suchbaum im Gesamten mehr MCTS-Iterationen zugrunde, woraus ein stärkeres Spielverhalten resultiert.
Sobald der den aktuellen Spielzustand *sob* repräsentierende Suchbaumknoten ermittelt wurde, wird ausgehend von diesem Knoten eine durch *iterationCount* vorgegebene Anzahl von MCTS-Iterationen durchgeführt (vgl. Abbildung 15).

```
for i ← 1 to iterationCount do
 | monteCarloTreeSearch(node)
end
```

Abbildung 15 Zeilen 8-10 der *getNextAction2*-Implementierung

Im Anschluss an die MCTS-Iterationen wird als nächste Spielaktion diejenige ausgewählt, die aus Sicht von *node* zu dem Kindknoten führt, welcher im Rahmen aller durchgeführten MCTS-Iterationen am häufigsten traversiert wurde (vgl. Abbildung 16). Außerdem wird dieser als aktueller Suchbaum zwischengespeichert, sodass weitere MCTS-Iterationen auf diesem aufbauen können.

```
nextAction ← node.actionThatMaximizesVisitCount()
searchTree ← node.getChild(nextAction)

return nextAction
```

Abbildung 16 Zeilen 12-15 der *getNextAction2*-Implementierung

6.3 Berücksichtigung von Passsituationen

6.3.1 Spielaktionen

In diesem Kapitel wird in Anlehnung an die zweite Anforderung aus Kapitel 6.1 eine softwaretechnische Lösung modelliert, welche dem Verhalten des GBG-Framework Spielzustände, in denen ein Spieler passen muss, zu überspringen, entgegenwirkt. Wie zuvor beschrieben resultiert dieses Verhalten aus dem Mangel einer Modellierung von Passaktionen in der Software. Der erste Teil des für diesen Anwendungsfall erarbeiteten Lösungsansatzes beschäftigt sich mit der Erweiterung des Framework um solche Passaktionen und wird durch das Klassendiagramm in Abbildung 17 veranschaulicht.

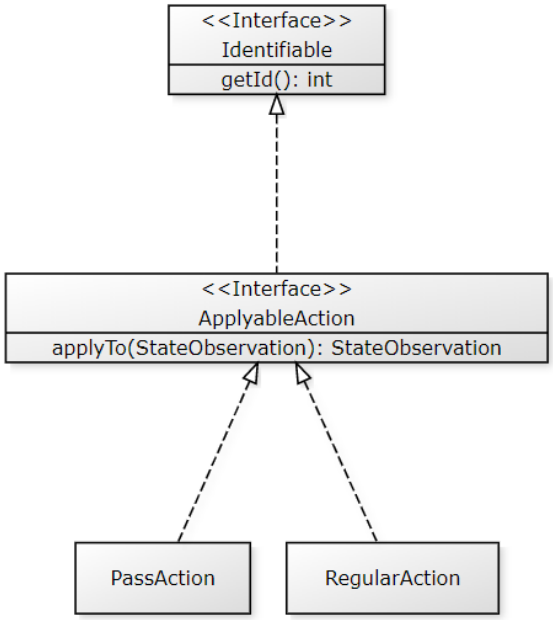

Abbildung 17 Spezifikation der Aktionskomponenten als UML-Diagramm

Die Anforderung, dass Spielzustände von Passsituationen im Suchbaum explizit dargestellt werden, impliziert zugleich die Notwendigkeit, deren Folgezustände ermitteln zu können. Anderenfalls könnten Suchbäume an solchen Knoten nicht fortgeführt werden.
Diese Ermittlung von Folgezuständen erfolgt üblicherweise durch das Simulieren von entsprechenden Spielaktionen, was zu der Designentscheidung führt, dieses Verhalten durch die *applyTo*-Methode des *ApplyableAction*-Interface zu abstrahieren. So repräsentiert diese Methode die in einem Folgezustand s' resultierende Anwendung einer Aktion auf einen Zustand s. Da *ApplyableAction* von den Klassen *RegularAction* und *PassAction* implementiert wird, definieren diese für sich, wie die von ihnen repräsentierte Aktionsart auf Spielzustände angewendet wird. Damit handelt es sich hierbei um einen polymorphen Lösungsansatz. Denn der MCTS-Algorithmus arbeitet lediglich mit *ApplyableAction*-Instanzen und weiß somit nicht, welches Verhalten sich hinter einzelnen Aktionsobjekten verbirgt.

1. Reguläre Aktionen

Reguläre Spielaktionen werden durch die Klasse *RegularAction* repräsentiert. Diese basiert intern auf einer ummantelten Instanz der *ACTIONS*-Klasse, bei der es sich um eine native Repräsentationsform von Aktionen im GBG-Framework handelt. Durch diese *ACTIONS*-Instanz lassen sich die von den Interfaces vorgegebenen Methoden *getId* und *applyTo* in *RegularAction* implementieren. So werden einerseits Aufrufe von *getId* lediglich an die *toInt*-Methode der ummantelten *ACTIONS*-Instanz delegiert. Andererseits wird Letztere auch zur im Algorithmus 2 veranschaulichten Implementierung der *applyTo*-Methode herangezogen.

Algorithmus 2: Anwendung einer regulären Spielaktion auf einen Spielzustand.

Input : StateObservation-Instanz *sob*, auf die die Aktion angewendet werden soll.

Output: Der aus der Anwendung der Aktion resultierende Folgezustand.

```
1  Function applyRegularActionTo(sob):
2      followingState ← sob.copy()
3      followingState.advance(action)
4
5      if followingState.player equals sob.player then
           /* pass situation was skipped                    */
6          passState ← followingState.previousState
7          return passState
8      else
9          return followingState
10     end
11 End Function
```

Der obige Pseudocode wendet eine reguläre Aktion auf einen gegebenen Spielzustand *sob* an. Dazu wird dieser zunächst kopiert, bevor dessen native *advance*-Methode aufgerufen wird, um die ummantelte Spielaktion *action* auf diesen Zustand anzuwenden. Da diese *advance*-Methode auftretende Passsituationen überspringt, wird abschließend überprüft, ob in dem Folgezustand *followingState* nach wie vor derselbe Spieler am Zug ist wie im Ausgangszustand *sob*. Das würde bedeuten, dass die Anwendung von *action* auf *sob* zu einer Passsituation geführt hat, welche automatisch übersprungen wurde. In diesem Fall handelt es sich bei dem Vorgängerzustand von *followingState* um diesen Passzustand und zugleich um den tatsächlichen Nachfolgezustand von *sob*. Sofern keine Passsituation übersprungen wurde, ist der Folgezustand dagegen durch *followingState* gegeben.

2. Passaktionen

Passaktionen werden durch die *PassAction*-Klasse modelliert. Da es sich bei diesen um Pseudoaktionen handelt, lassen sie sich nicht wie reguläre Spielaktionen durch den Index einer adressierten Spielfeldposition identifizieren. Nichtsdestotrotz wird auch für Passaktionen eine Identifikationsmöglichkeit benötigt. Denn die Klasse *MCTSNode* (vgl. Kapitel 6.4.2) ordnet kindknotenbezogene Informationen wie akkumulierte Bewertungen oder Anzahlen der Traversierungen durch Zuordnungstabellen den zu diesen Kindknoten führenden Spielaktionen zu. Dabei stellt in einem Spielzustand s der numerische Identifikator einer verfügbaren Spielaktion den Schlüssel dar, mit dem beispielsweise die Wahrscheinlichkeit, diese Aktion ausgehend von s zu wählen, abgefragt werden kann.
So gibt die *getId*-Methode für alle Instanzen der Klasse *PassAction* eine konstante, dem kleinstmöglichen *Integer*-Wert entsprechende Zahl als ID zurück. Mit dieser wird die Wahrscheinlichkeit einer Kollision mit IDs regulärer Aktionen minimiert, da Letztere sich zumeist auf Spielbrettindizes und damit auf Werte $id_i \geq 0$ beziehen.
Im Folgenden wird zudem das von *PassAction* implementierte Verhalten der *applyTo*-Methode vorgestellt (vgl. Algorithmus 3).

Algorithmus 3: Anwendung einer Passaktion auf einen Spielzustand.

Input : StateObservation-Instanz *sob*, auf die die Aktion angewendet werden soll.

Output: Der aus der Anwendung der Aktion resultierende Folgezustand.

```
1 Function applyPassActionTo(sob):
2     followingState ← sob.copy()
3     followingState.player ← followingState.getPlayerOfTheNextRound()
4     followingState.updateAvailableActions()
5     return followingState
6 End Function
```

Wie auch im Algorithmus 2 besteht der erste Schritt der hier dargestellten Programmlogik darin, eine Kopie des gegebenen Spielzustands *sob* in $followingState$ zu speichern. Anschließend wird auf $followingState$ ein Pass simuliert. Dieser äußert sich entgegen einer regulären Aktion nicht in einer Manipulierung der Spielbrettkonfiguration. Stattdessen kommt unter Beibehaltung dieser Konfiguration der nächste Spielteilnehmer zum Zug (vgl. Zeile 3).
Durch den Aufruf von *updateAvailableActions* in Zeile 4, werden die in $followingState$ hinterlegten möglichen Spielaktionen aktualisiert, indem sie durch die verfügbaren Aktionen des Spielers, welcher nun am Zug ist, überschrieben werden.

6.3.2 Spielzustände

Nachdem nun das GBG-Framework um Softwarekomponenten erweitert wurde, die eine Verwendung von Passaktionen ermöglichen, bedarf es einer mit den neuen Aktionskomponenten kompatiblen Spielzustandsklasse. Mit *GameStateIncludingPass* wird in Abbildung 18 eine diesem Zweck gerecht werdende Klasse spezifiziert. Im dargestellten Unified Modeling Language (UML)-Diagramm ist erkennbar, dass diese sich als Wrapper um eine *StateObservation*-Instanz legt. Dabei ermöglicht dieses *StateObservation*-Objekt intern erst die Implementierung der von *GameStateIncludingPass* bereitgestellten Methoden. Letztere sind in Anlehnung an die Anforderungen des in Kapitel 6.4 implementierten MCTS-Algorithmus spezifiziert und werden im Folgenden erläutert.

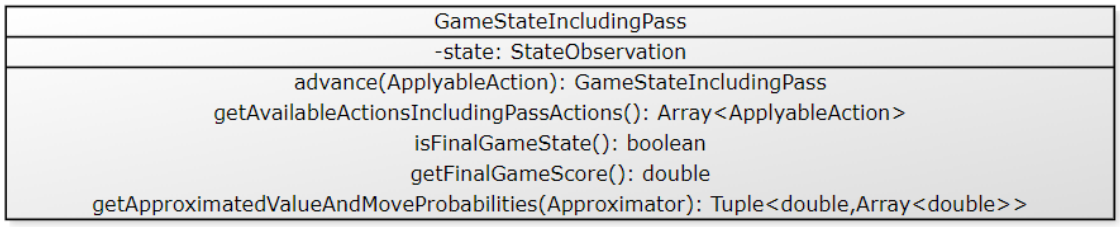

Abbildung 18 Spezifikation der GameStateIncludingPass-Klasse als UML-Diagramm

1. getAvailableActionsIncludingPassActions

Die Methode *getAvailableActionsIncludingPassActions* wird im Rahmen der MCTS verwendet und gibt die verfügbaren Spielaktionen ausgehend von einem Zustandsknoten s zurück. Diese werden zur Anwendung auf den Spielzustand s benötigt, um dessen entsprechende Kindknoten zu ermitteln.
Diese anwendbaren Spielaktionen werden jeweils als Instanzen des in Kapitel 6.3.1 spezifizierten *ApplyableAction*-Typs in einem Array zurückgegeben. Damit gewährleistet die *GameStateIncludingPass*-Klasse, dass selbst im Fall einer Passsituation s_p eine Aktion verfügbar ist, die zu dem Folgezustand s'_p führt. So können Passzustände verlassen und Suchbäume an betroffenen Knoten fortgesetzt werden.

Funktionsweise: Der Algorithmus hinter *getAvailableActionsIncludingPassActions* delegiert die Anfrage nach verfügbaren Spielzügen zunächst an die ummantelte *StateObservation*-Instanz *sob*. Wenn die zurückgegebene Menge möglicher Spielaktionen leer ist, handelt es sich bei *sob* um eine Passsituation. In diesem Fall gibt die Methode ein Array, das eine erzeugte *PassAction*-Instanz enthält, zurück.
Sofern in *sob* doch Spielzüge möglich sind, wird aus jeder verfügbaren Spielaktion ein Objekt des Typs *RegularAction* konstruiert. In diesem Fall wird anschließend ein Array zurückgegeben, welches diese erzeugten *RegularAction*-Instanzen enthält.

2. advance

Mit *advance* lässt sich eine übergebene Spielaktion des Typs *ApplyableAction* auf den durch *GameStateIncludingPass* repräsentierten Zustand anwenden.

Funktionsweise: Wie genau sich das Anwenden dieser *ApplyableAction*-Instanz auf den Spielzustand äußert, hängt letztlich von der konkreten, der Aktion zugrunde liegenden Implementierung ab. So dokumentiert Kapitel 6.3.1 jeweils das Verhalten der im Rahmen dieser Arbeit spezifizierten Aktionstypen *RegularAction* und *PassAction*.
In der *advance*-Methode wird die von der anwendbaren Aktion bereitgestellte *applyTo*-Methode auf die ummantelte *StateObservation*-Instanz angewendet. Diese polymorphe Operation resultiert in einem Folgezustand vom Typ *StateObservation*, welcher im Anschluss durch die *GameStateIncludingPass*-Klasse ummantelt und zurückgegeben wird.

3. isFinalGameState

Die *isFinalGameState*-Methode dient dem MCTS-Algorithmus als Abbruchbedingung. So wird diese verwendet, um zu ermitteln, ob ein Spielzustand bereits entschieden ist. In diesem Fall würde sich ein Spiel nicht fortführen und der dazugehörige Suchbaum sich an dem entsprechenden Zustandsknoten nicht weiter expandieren lassen.

Funktionsweise: Die Implementierung hinter *isFinalGameState* besteht lediglich darin, Anfragen an die *isGameOver*-Methode der ummantelten *StateObservation*-Instanz zu delegieren.

4. getFinalGameScore

Sofern im Rahmen einer MCTS die Bewertung eines terminierenden Spielzustands benötigt wird, ist es nicht notwendig, diese zu approximieren, da das tatsächliche Spielergebnis bereits vorliegt und durch *getFinalGameScore* abgefragt werden kann.

Funktionsweise: Auch diese Methode gibt Aufrufe an die zugrunde liegende *StateObservation*-Instanz weiter. Denn Letztere bietet eine *getGameScore*-Methode, welche den gewünschten Wert zurückgibt.

5. getApproximatedValueAndMoveProbabilities

Für den Fall, dass ein Zustand s bewertet werden soll, der noch nicht entschieden ist, lassen sich die Werte v und $\vec{p}$ durch *getApproximatedValueAndMoveProbabilities* approximieren. Dabei handelt es sich bei v um eine skalare Bewertung von s aus Sicht des Spielers, der im Spielzustand s am Zug ist und bei $\vec{p}$ um eine Policy, welche die von s ausgehenden Spielaktionen aus der Perspektive desselben Spielteilnehmers beurteilt. Diese Werte werden von einer Monte-Carlo-Baumsuche zur Ermittlung von zu expandierenden Spielbaumknoten herangezogen.

Funktionsweise: Um das zurückzugebende Tupel $(v, \vec{p})$ zu ermitteln, delegiert *getApproximatedValueAndMoveProbabilities* einen Aufruf grundsätzlich an die *predict*-Methode der als Parameter erwarteten *Approximator*-Instanz und gibt dessen Berechnung zurück.

Dabei ist für Passsituationen ein Ausnahmefall zu beachten. Da die *StateObservation*-Instanz einer Passsituation keine verfügbaren Spielzüge bereitstellt, kann auf dieser Basis ebenso keine Bewertung des zugrunde liegenden Zustands stattfinden. Um dieses Problem zu lösen, findet die Evaluation eines solchen Spielzustands s, in dem Spieler O passen muss, zunächst aus der Perspektive des Gegenspielers X statt, woraus $(v_x, \vec{p_x})$ resultiert. Die Policy $\vec{p_x}$ kann in diesem Fall verworfen werden, da sie sich auf die Sicht von X bezieht und damit für den Spielteilnehmer O irrelevant ist. Schließlich kann Letzterer ohnehin nur passen, weshalb seine Policy die 100-prozentige Wahrscheinlichkeit einer Passaktion und damit den Wert $\begin{pmatrix}1\end{pmatrix}$ vorsieht. Zudem ergibt sich die skalare Bewertung des Spielzustands aus Sicht von O durch die Negation der Beurteilung von X.
So gibt *getApproximatedValueAndMoveProbabilities* für diese Passsituation das Tupel $(-v_x, \begin{pmatrix}1\end{pmatrix})$ zurück.

6.4 Komponenten einer Monte-Carlo-Baumsuche

6.4.1 Suchalgorithmus

In den ersten Schritten des Entwicklungsprozesses entstanden zwei experimentelle Prototypen einer Monte-Carlo-Baumsuche, welche sich beide in kompetitiven Vergleichen durch ein äußerst schwaches Spielverhalten äußerten. Die Ursachen für diese initialen Startschwierigkeiten und die damit verbundenen Herausforderungen werden in Kapitel 8 aufgegriffen und näher erläutert. Auch wenn das Debuggen dieser Versionen diverse Erkenntnisse lieferte, kam der wirkungsvolle Algorithmus 4, welcher im Folgenden beschrieben wird, durch die Inspiration von [31] zustande.

Beschränkung auf 2-Spieler-Spiele
Der durch Algorithmus 4 dargestellte rekursive Ansatz einer Monte-Carlo-Baumsuche bringt die Einschränkung mit sich, dass er auf Spiele mit zwei Teilnehmern beschränkt ist. Diese Eigenschaft ist dadurch bedingt, dass ein zurückgegebener Zustandswert auf jeder Ebene des Rekursionsstacks negiert wird, was dem Backpropagation-Schritt der Monte-Carlo-Baumsuche entspricht.
Eine Begründung für diesen Ansatz findet sich bereits in Kapitel 3.4 wieder und hängt damit zusammen, dass mit jedem Ebenenwechsel auch der Spieler, aus dessen Sicht ein Spielzustand bewertet wird, alterniert. Für 2-Spieler-Spiele reicht demnach eine einfache Negation auf jeder Ebene aus, um diesen Sichtwechsel darzustellen.
Im Pseudocode von Algorithmus 4 ist dieses Verhalten in den Code-Zeilen 3, 10 und 20 durch die jeweils vorangestellten Minusoperatoren ersichtlich. Damit sollte sich die Programmlogik in der Theorie durch das Entfernen dieser Operatoren in den entsprechenden Code-Zeilen für 1-Spieler-Spiele adaptieren lassen.

Pseudocode

Nachdem die für den MCTS-Wrapper-Agenten implementierte Monte-Carlo-Baumsuche nachfolgend in ihrer Gesamtheit dargestellt wird, werden anschließend einzelne Code-Bestandteile im Detail erläutert.

Algorithmus 4: Rekursive Monte-Carlo-Baumsuche für Spiele mit zwei Teilnehmern. Dieser Algorithmus entstand in Anlehnung an [31].

Input : Suchbaumknoten *node* vom Typ $MCTSNode$, an dem die Baumsuche beginnen soll.

Output: Die auf jeder Rekursionsebene negierte Bewertung des Spielzustands eines erreichten Blattknotens.

```
1  Function monteCarloTreeSearch(node):
2      if node represents a terminating game state then
3          return −node.finalGameScore
4      end
5
6      if node is not expanded then
7          (value, moveProbabilities) ← node.predict(approximator)
8          node.p ← moveProbabilities
9          node.expanded ← true
10         return −value
11     end
12
13     (action, childNode) ← node.selectChild()
14     childValue ← monteCarloTreeSearch(childNode)
15
16     node.w[action] ← node.w[action]+childValue
17     node.n[action] ← node.n[action]+1
18     node.q[action] ← node.w[action]/node.n[action]
19
20     return −childValue
21 End Function
```

Abbildung 19 präsentiert die erste bedingte Anweisung des MCTS-Algorithmus, welche zugleich eine Abbruchbedingung darstellt. Denn sofern der Knoten *node* einem terminierenden Spielzustand entspricht, kann dieser nicht weiter expandiert werden. Da somit ein entschiedenes Spiel vorliegt, lässt sich dessen tatsächliches Spielergebnis ohne die Notwendigkeit einer Approximation negiert zurückgeben.

```
if node represents a terminating game state then
    return −node.finalGameScore
end
```

Abbildung 19 Zeilen 2-4 des MCTS-Pseudocodes

Sofern es sich bei *node* nicht um einen terminierenden Spielzustand handelt und die in Abbildung 19 beschriebene Abbruchbedingung damit nicht wirksam wird, kommt möglicherweise der nächste in Abbildung 20 aufgeführte Code-Block zum Einsatz. Letzterer wird dabei nur ausgeführt, wenn *node* im Rahmen der bisherigen MCTS-Iterationen noch nicht expandiert wurde und somit einen Blattknoten des Monte-Carlo-Suchbaums darstellt. In diesem Fall wird die Expansion im Folgenden durchgeführt. Zu diesem Zweck wird zunächst *node*'s Spielzustand durch einen gegeben Approximator (vgl. Kapitel 6.5) evaluiert. Daraus resultiert ein Tupel, welches neben der skalaren Bewertung *value* des Zustands auch die Beurteilung *moveProbabilities* möglicher Spielaktionen enthält.
Im nächsten Schritt wird der Wahrscheinlichkeitsvektor *moveProbabilities* in dem Knoten *node* gespeichert. Das ist notwendig, damit im Selektionsschritt zukünftiger MCTS-Iterationen abgeschätzt werden kann, welcher von *node*'s Kindknoten expandiert werden soll. Ein Beweis dafür, dass mindestens ein solcher Nachfolgeknoten ermittelt werden kann, besteht darin, dass es sich wie zuvor erläutert, bei *node* nicht um einen terminierenden Spielzustand handelt. Die letzte Anweisung des erläuterten Code-Blocks gibt die approximierte skalare Bewertung *value* negiert zurück.

```
if node is not expanded then
    (value, movePropabilities) ← node.predict(approximator)
    node.p ← movePropabilities
    node.expanded ← true
    return −value
end
```

Abbildung 20 Zeilen 6-11 des MCTS-Pseudocodes

Falls ein Methodenaufruf von *monteCarloTreeSearch* die in Abbildung 21 dargestellten Code-Anweisungen erreicht, bedeutet das, dass keine der beiden vorherigen Abbruchbedingungen zur Anwendung kam. In diesem Fall ist mit dem Parameter *node* ein bereits expandierter Knoten, der einen nicht terminierenden Spielzustand repräsentiert gegeben. Um den nächsten in dieser MCTS-Iteration zu expandierenden Nachfolgeknoten von *node* zu ermitteln, werden via Selektion dessen vielversprechendster Kindknoten *childNode* sowie die zu diesem führende Aktion ausgewählt. Im Anschluss findet ein auf *childNode* basierender rekursiver Aufruf von *monteCarloTreeSearch* statt.
Sofern es sich auch bei *childNode* um einen bereits expandierten Knoten mit einem nicht terminierenden Spielzustand handelt, kommt es zu einem weiteren Selbstaufruf der Methode für dessen vielversprechendsten Kindknoten. Der dadurch entstehende Rekursionsbaum wächst so lange an, bis ein Zustandsknoten, welcher eine Abbruchbedingung erfüllt, gefunden wird. An dieser Stelle wird dessen Zustandsbewertung ermittelt und mit einer Negation auf jeder Rekursionsebene durch den Rekursionsstack hinweg bis zur Wurzel des Monte-Carlo-Suchbaums zurückgegeben.
Im Rahmen dieses Rekursionsrücklaufs wird die entsprechende Bewertung auf jeder Ebene in der Variable *childValue* gespeichert und zur Aktualisierung der Knoteninformationen verwendet. Bei Letzteren handelt es sich neben dem Besuchszähler n eines Knotens auch um dessen Gesamtbewertung w und Durchschnittsbewertung q.

$(action, childNode) \leftarrow node.\text{selectChild}()$
$childValue \leftarrow \text{monteCarloTreeSearch}(childNode)$

$node.\text{w}[action] \leftarrow node.\text{w}[action]+childValue$
$node.\text{n}[action] \leftarrow node.\text{n}[action]+1$
$node.\text{q}[action] \leftarrow node.\text{w}[action]/node.\text{n}[action]$

return $-childValue$

Abbildung 21 Zeilen 13-20 des MCTS-Pseudocodes

6.4.2 Suchbaumknoten

Im vorherigen Kapitel 6.4.1 wurde die implementierte Monte-Carlo-Baumsuche des MCTS-Wrapper-Agenten dokumentiert. In der Beschreibung des Algorithmus zeigt sich, dass dieser Gebrauch von der einen Suchbaumknoten modellierenden *MCTSNode*-Klasse macht. Das UML-Diagramm in Abbildung 22 spezifiziert neben den wichtigsten öffentlichen Instanzvariablen auch die *selectChild*-Methode dieser Klasse. Diese Komponenten werden nachfolgend näher erläutert.

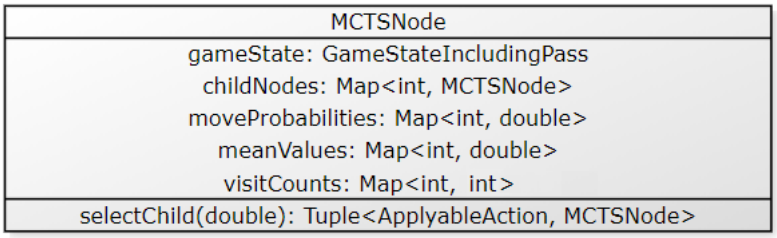

Abbildung 22 Spezifikation der MCTSNode-Klasse als UML-Diagramm

Instanzvariablen

- Mit *gameState* ist die wichtigste Instanzvariable der *MCTSNode*-Klasse gegeben, da diese den in einem Suchbaumknoten gespeicherten Spielzustand enthält. Dabei fällt auf, dass zu diesem Zweck nicht auf den Typ *StateObservation* zurückgegriffen wird, welcher eine native Zustandsmodellierung des GBG-Framework darstellt. Stattdessen wird der Spielzustand eines Suchbaumknotens durch die in Kapitel 6.3.2 implementierte Wrapper-Klasse *GameStateIncludingPass* abgebildet. Letztere stellt implizit sicher, dass im Suchbaum auch Passzustände berücksichtigt werden.

- Durch *childNodes* ist eine Zuordnungstabelle gegeben, in der die direkten Nachfolgeknoten des repräsentierten Suchbaumknotens gespeichert sind. Dabei wird einem solchen Kindknoten als Schlüssel jeweils die ID der zu diesem Knoten führenden Spielaktion zugewiesen.

Die Darstellung von Spielzuständen durch *GameStateIncludingPass*-Instanzen impliziert die Notwendigkeit der Verwendung des *ApplyableAction*-Interface zur Abbildung von Spielzügen, da nur solche mit *GameStateIncludingPass*-Zuständen kompatibel sind. Zudem implementieren diese Aktionen ebenso das *Identifiable*-Interface, sodass gewährleistet ist, dass alle in diesem Rahmen auftretenden Spielaktionen inklusive Passaktionen eine numerische ID bereitstellen, welche als Schlüssel für die Zuordnungstabellen verwendet wird.

Damit ist die Architektur der *MCTSNode*-Klasse an eine verkettete Liste angelehnt. Demnach stellt eine Instanz dieser Klasse neben einem Suchbaumknoten k gleichzeitig einen Suchbaum dar, welcher eine Teilmenge des gesamten Monte-Carlo-Suchbaums repräsentiert und dessen Wurzel durch k gegeben ist. So ist eine Traversierung des Suchbaums erst deshalb möglich, weil jeder Baumknoten zugleich seine Kindknoten enthält.

Der Grund, weshalb einem solchen aber keine Referenz zu dessen Elternknoten vorliegt, ist darin begründet, dass diese aufgrund des rekursiven Lösungsansatzes nicht nötig ist. In einem iterativen MCTS-Algorithmus könnte sie verwendet werden, um im Rahmen der Backpropagation den Pfad von einem Blattknoten zur Baumwurzel zu bestimmen. In dem in dieser Arbeit vorgestellten rekursiven Ansatz einer Monte-Carlo-Baumsuche ist der für die Rückführung benötigte Pfad dagegen inhärent durch den aufgebauten Rekursionsstack gegeben.

- Ähnlich wie bei $childNodes$ handelt es sich bei $movePropabilities$, $meanValues$ und $visitCounts$ um Zuordnungstabellen, in denen kindknotenbezogene Informationen den IDs der zu diesen Knoten führenden Aktionen zugeordnet werden.
  - Mit $movePropabilities$ ist in diesem Rahmen eine Policy gegeben, die eine Wahrscheinlichkeitsverteilung bezüglich der verfügbaren Spielaktionen und damit auch den aus der Anwendung dieser resultierenden Kindknoten darstellt.
  - Durch $meanValues$ wird jedem Nachfolgeknoten dessen durch die bisherigen MCTS-Iterationen ermittelte durchschnittliche Bewertung zugewiesen.
  - Dagegen gibt $visitCounts$ Auskunft darüber, mit welcher Häufigkeit die referenzierten Kindknoten im Rahmen der Monte-Carlo-Baumsuche selektiert und damit auch traversiert wurden.

Selektion von Kindknoten

In Anlehnung an den Selektionsschritt einer Monte-Carlo-Baumsuche bietet die *selectChild*-Methode eine Möglichkeit, ausgehend von einem Knoten k, dessen vielversprechendsten Kindknoten k' zu ermitteln. Die Bewertung geschieht dabei aus Sicht des Spielers, der im Spielzustand von k am Zug ist und damit durch die Wahl seiner Spielaktion das Spiel dahingehend beeinflussen kann, in den Zustand k' überzugehen. Der implementierte Mechanismus hinter der Methode wird nachfolgend durch Algorithmus 5 dargestellt und entlang des Pseudocodes erläutert.

Algorithmus 5: Selektionsmechanismus der Monte-Carlo-Baumsuche.

Output: Ein Tupel, welches neben dem nächsten zielführendsten Kindknoten (*MCTSNode*) auch die Spielaktion (*ApplyableAction*) enthält, welche zu diesem führt.

```
1  Function selectChild():
2      availableActions ← gameState.getAvailableActionsIncludingPassActions()
3
4      bestValue ← NEGATIVE INFINITY
5      bestAction ← NULL
6
7      foreach action a in the set availableActions do
8          value ← getPUCTvalueFor(a)
9          if value is greater than bestValue then
10             bestValue ← value
11             bestAction ← a
12         end
13     end
14
15     if childNodes contains bestAction's id as a key then
16         child ← childNodes.getNodeByActionId(bestAction.getId())
17     else
18         child ← new MCTSNode(gameState.advance(bestAction))
19         childNodes.put(bestAction.getId(), child)
20     end
21
22     return (bestAction, child)
23 End Function
```

In Zeile 2 des oben vorgestellten Pseudocodes wird zunächst *gameState*'s *getAvailableActionsIncludingPassActions*-Methode verwendet, um die möglichen Spielaktionen ausgehend von dem repräsentierten Spielzustand abzufragen. Diese werden im Anschluss in *availableActions* gespeichert. Das in dieser Arbeit bereits beschriebene Verhalten von *getAvailableActionsIncludingPassActions* sorgt dafür, dass bei einer Passsituation keine leere Menge von Spielaktionen zurückgegeben wird. So ist in diesem Fall zumindest eine *PassAction*-Instanz präsent.

Die IDs der verfügbaren Spielaktionen aus *availableActions* werden anschließend verwendet, um suchbaumknotenspezifische Informationen auszulesen. Aus diesen werden nachfolgend jeweils die Polynomial Upper Confidence Trees (PUCT)-Bewertungen der entsprechenden Kindknoten berechnet (vgl. Abbildung 23). Dabei wird diejenige Spielaktion als zielführendste bewertet und in *bestAction* gespeichert, die den größten PUCT-Wert aufweist.

```
bestValue ← NEGATIVE INFINITY
bestAction ← NULL

foreach action a in the set availableActions do
    value ← getPUCTvalueFor(a)
    if value is greater than bestValue then
        bestValue ← value
        bestAction ← a
    end
end
```

Abbildung 23 Zeilen 4-13 der *selectChild*-Implementierung

Der Grund, weshalb Aktionen in dieser Implementierung basierend auf dem PUCT- anstatt UCT-Algorithmus selektiert werden, besteht darin, dass die PUCT-Formel die approximierte Bewertung $(v, \vec{p})$ eines Spielzustands berücksichtigt.
So bestimmen auch Silver et al. [7, S. 26] die nächste auszuwählende Spielaktion durch $\mathbf{argmax_a}(Q(s_t, a) + U(s_t, a))$. Dabei wird U wie folgt definiert:

$$U(s,a) = c_{puct} P(s,a) \frac{\sqrt{\sum_b N(s,b)}}{1+N(s,a)}$$

- s_t: Beschreibt einen Spielzustand s zum Zeitpunkt t.
- a: Die zur Ermittlung der PUCT-Bewertung herangezogene Spielaktion.
- $P(s,a)$: Liefert die Wahrscheinlichkeit, die Aktion a ausgehend vom Spielzustand s zu wählen. Somit wird durch P der approximierte Wahrscheinlichkeitsvektor $\vec{p}$ berücksichtigt.
- $N(s,a)$: Gibt die Anzahl bereits erfolgter Selektionen von a ausgehend vom Zustand s zurück.
- $Q(s,a)$: Beantwortet die Anfrage nach der durchschnittlichen Bewertung des Spielzustands, welcher ausgehend von s durch die Anwendung von a erreichbar ist. Dieser Wert basiert grundsätzlich auf den approximierten skalaren Zustandsbewertungen v_i im Rahmen der MCTS-Iterationen.
- c_{puct}: Konstanter Wert zur Regulierung des Erkundungsdrangs.

Nachdem die zielführendste Aktion *bestAction* ermittelt wurde, wird der aus dieser resultierende Folgeknoten in der *childNodes*-Zuordnungstabelle gesucht (vgl. Abbildung 24). Sofern der Knoten noch nicht in dieser Tabelle existiert, wird er durch eine neue *MCTSNode*-Instanz initialisiert. Zum Schluss wird der ausgewählte Kindknoten mit der zu diesem führenden Spielaktion als Tupel zurückgegeben.

```
if childNodes contains bestAction's id as a key then
  | child ← childNodes.getNodeByActionId(bestAction.getId())
else
  | child ← new MCTSNode(gameState.advance(bestAction))
  | childNodes.put(bestAction.getId(), child)
end

return (bestAction, child)
```

Abbildung 24 Zeilen 15-22 der *selectChild*-Implementierung

6.5 Approximation von Spielzustandsbewertungen

Dieses Kapitel beschäftigt sich mit der Konzeption sowie Implementierung der approximierenden Komponente, welche im MCTS-Wrapper-Agenten die Random-Rollouts einer Monte-Carlo-Baumsuche ersetzt.

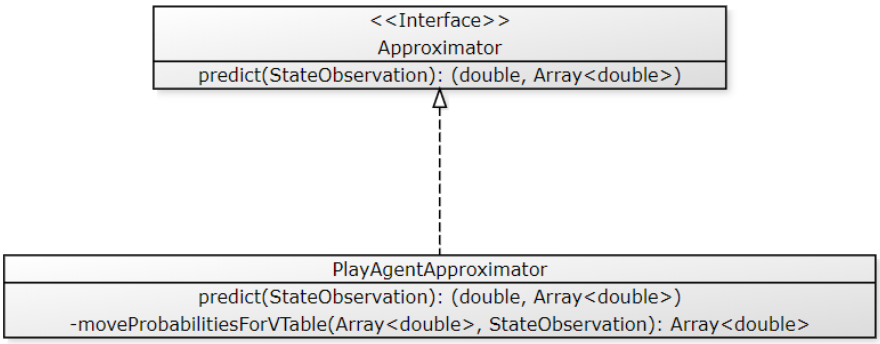

Abbildung 25 Spezifikation der Approximator-Komponente als UML-Diagramm

Abbildung 25 visualisiert in einem Klassendiagramm die Architektur hinter der besagten Approximator-Komponente. Dabei ist in erster Linie das spezifizierte *Approximator*-Interface von Bedeutung, da der Entwicklungsprozess aller anderen Komponenten gegen dieses Interface verläuft, um Abhängigkeiten von der *PlayAgentApproximator*-Klasse zu vermeiden.
Letztere ist eine konkrete Implementierung des *Approximator*-Interface, welche die Evaluation auf Basis eines gegebenen Spielagenten durchführt. Ein *PlayAgentApproximator* verwendet zur Bewertung eines Spielzustands s die native Bewertungsfunktion des zugrunde liegenden Agenten, woraus sich der skalare Wert v ableitet. Um den Vektor $\vec{p}$ zu berechnen, werden auf gleiche Weise die Folgezustände von s bewertet und in ihrer Gesamtheit in eine Softmax-Funktion gegeben. So ergibt sich die Ausgabe $(v, \vec{p})$.
Da Software-Abhängigkeiten lediglich gegenüber dem *Approximator*-Interface bestehen, muss im weiteren Verlauf auf der Einstiegsebene des Programms via Dependency Injection das gewünschte Verhalten injiziert werden. Zwar wird im Rahmen dieser Arbeit nur der *PlayAgentApproximator* benötigt, doch ermöglicht dieses Design, polymorphe Verhaltensweisen zu implementieren. So resultiert daraus beispielsweise die Möglichkeit, in einem zukünftigen Projekt ohne die Notwendigkeit größerer Softwareanpassungen, ähnlich wie bei AlphaZero ein DCNN zur Approximation heranzuziehen.

7 Evaluation des AlphaZero-inspirierten Agenten

Dieses Kapitel widmet sich der Evaluation des im vorangegangenen Kapitel entwickelten MCTS-Wrapper-Agenten. In diesem Rahmen wird in allen Experimenten der gleiche TD-3-Tupel-Agent als approximierende Komponente verwendet. Dieser auf einem neuronalen N-Tupel-Netz basierende Spielagent des GBG-Framework verfügt bereits über eine starke Spielleistung und ist damit in der Lage, Spielsituationen mit hinreichender Aussagekraft zu bewerten. Ohne einen Lookahead wird dieser Agent im Folgenden durch TD3T gekennzeichnet, während dieser in Erweiterung durch den MCTS-Wrapper als MCTS[TD3T]$_i$ bezeichnet wird, wobei i die Anzahl der MCTS-Iterationen angibt.

Der TD3T-Agent wird in der gesamten Evaluation durch den zum 27.10.2020 im GBG-Framework unter `agents\Othello\TCL3-100_7_250k-lam05_P4_nPly2-FAm.agt` abgerufenen Spielagenten repräsentiert.

Alle Experimente dieses Kapitels beziehen sich auf das Brettspiel Othello.

7.1 TD-3-Tupel-Agent ohne Lookahead

Im ersten Schritt des Evaluationsprozesses soll zunächst ermittelt werden, welchen Einfluss der durch einen Wrapper entstehende Lookahead auf die Spielstärke eines Agenten nimmt. Zu diesem Zweck wird der TD3T-Agent mit dem durch den MCTS-Wrapper erweiterten MCTS[TD3T]-Agenten verglichen. Im Rahmen dieses Experiments werden TD3T dabei 249 Versionen des MCTS[TD3T]-Agenten gegenübergestellt, deren Anzahlen von MCTS-Iterationen von zwei bis 250 reichen. Allen diesen 249 Agentenvergleichen liegen simulierte Spielpartien aus jeweils 244 verschiedenen Startpositionen zugrunde, wobei ausgehend von jeder Startposition die Agenten in jeder der beiden Spielrollen Schwarz und Weiß gegeneinander antreten.
Das Ergebnis dieser Evaluation wird durch Abbildung 26 visualisiert. Dieser lässt sich entnehmen, dass bereits durch geringe Iterationszahlen ein positiver Effekt erzielt werden kann. So sind lediglich 32 MCTS-Iterationen notwendig, um eine 80-prozentige Siegesrate gegen TD3T zu erreichen. Es ist einleuchtend, dass ein Lookahead sich grundsätzlich positiv auf die Spielstärke auswirken sollte. Wichtig ist also die Berücksichtigung des damit verbundenen erhöhten Rechenbedarfs. So visualisiert Abbildung 26 auch die von den Agenten zur Ermittlung einer Spielaktion durchschnittlich benötigte Rechenzeit in Millisekunden. Es lässt sich erkennen, dass TD3T konstant in etwa eine viertel Millisekunde zur Bestimmung eines Spielzugs benötigt, wohingegen der Rechenbedarf von MCTS[TD3T] mit zunehmender Iterationszahl linear ansteigt. So bedarf Letzterer bei 250 MCTS-Iterationen ungefähr 67 ms zur Aktionsberechnung.
Um den zum Aufbau des Lookaheads notwendigen Rechenbedarf nicht zu vernachlässigen, stellt das abgebildete Chart auch das Verhältnis der Siegesrate zur benötigten Rechenzeit dar. Dabei lässt sich erkennen, dass das Maximum dieser Kurve bei 15 MCTS-Iterationen liegt. So erreicht MCTS[TD3T]$_{15}$ eine fast 68-prozentige Siegesrate bei einem durchschnittlichen Rechenaufwand von unter 3,5 ms pro Spielzug.

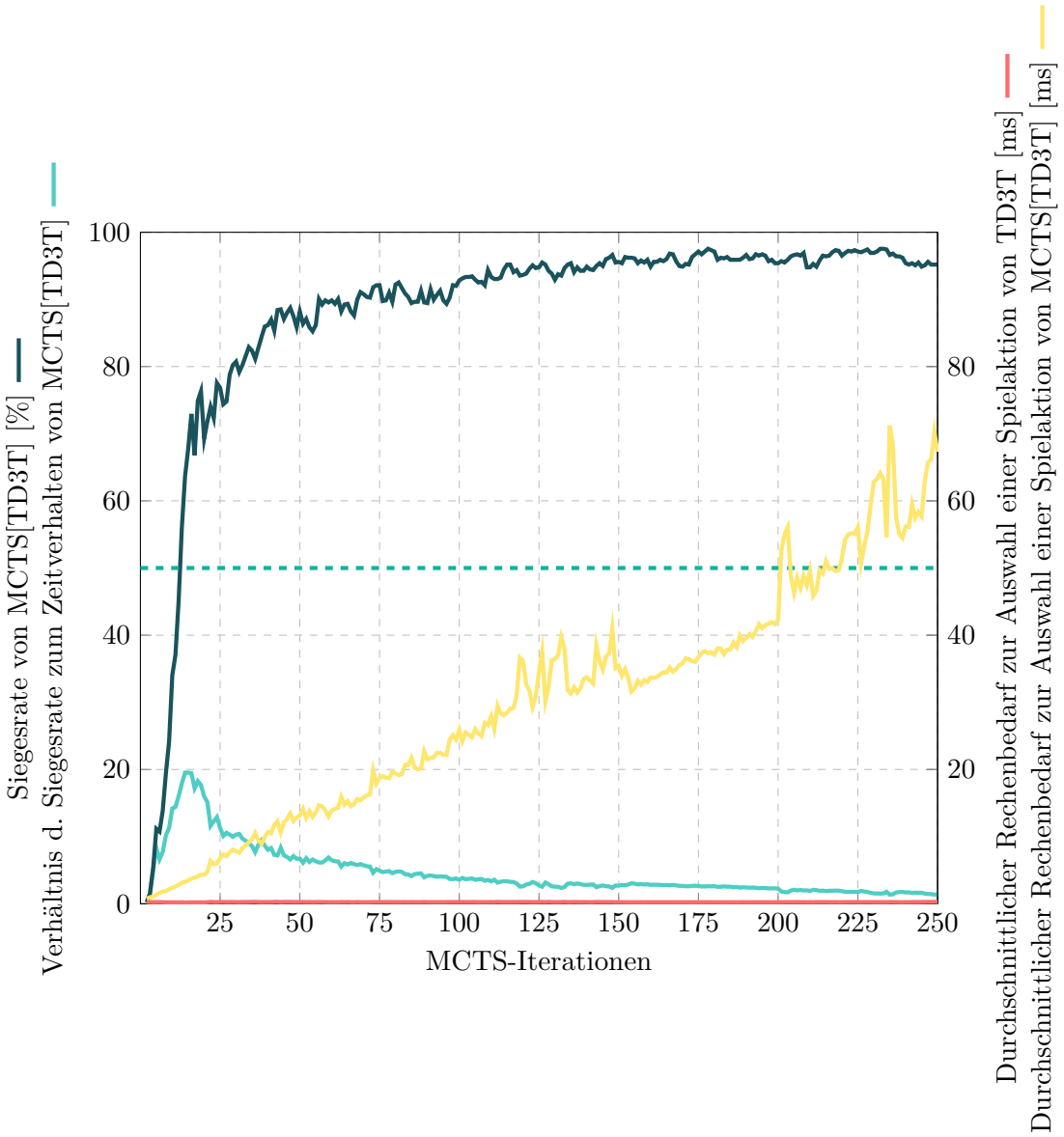

Abbildung 26 Siegesraten und Zeitverhalten aus der Sicht von MCTS[TD3T] gegen den blanken TD3T-Agenten.

7.2 TD-3-Tupel-Agent mit Minimax-Wrapper

Aus der vorherigen Evaluation in Kapitel 7.1 ließ sich ableiten, dass die Erweiterung eines Spielagenten um einen Lookahead ein hilfreiches Mittel ist, um dessen Spielstärke zu verbessern. Dennoch ist es schwierig, pauschale Aussagen über die Sinnhaftigkeit der Verwendung eines Lookahead-erzeugenden Wrappers zu treffen. So ist letztlich der damit verbundene Trade-off in Bezug auf die Rechenzeit für solche Entscheidungen ausschlaggebend. Beispielsweise stellt der Minimax-Algorithmus ein Suchverfahren dar, welches deterministisch zu einer optimalen Spielstrategie führt. Doch werden in Kapitel 4.1 Gründe dafür angeführt, weshalb ein solcher Algorithmus an komplexen Spielen wie Go scheitert.

Durch die MaxN2Wrapper-Klasse stellt das GBG-Framework eine generische Form des Minimax-Algorithmus bereit, welche sich auf beliebige N-Spieler-Spiele anwenden lässt. Diese Klasse lässt sich wie auch der im Rahmen dieser Arbeit implementierte Agent als Wrapper um eine PlayAgent-Instanz legen. Dadurch verbessert sie dessen Spielleistung durch eine Minimax-Suche, welche alle Spielbaumknoten innerhalb einer gegebenen Suchtiefe berücksichtigt. So ist ein Vergleich dieses Max-N-Wrappers mit dem implementierten MCTS-Wrapper vor allem unter Berücksichtigung des Trade-offs zwischen Spielstärke und benötigter Rechenleistung interessant. In den durch Abbildung 27 dargestellten Evaluationen werden Max-N-Wrapper mit den Suchtiefen eins bis drei verwendet. Dargestellt werden dabei die Siegesraten aus Sicht von MCTS[TD3T] sowie die pro Zug benötigte Rechenzeit der Spielagenten in Millisekunden. Aus den Charts ist ersichtlich, dass MCTS[TD3T] weniger als 100-MCTS-Iterationen benötigt, um gegen MaxN[TD3T]$_1$ und MaxN[TD3T]$_2$ jeweils eine Siegesrate von über 50 % zu erzielen. Allerdings erreicht MCTS[TD3T] dies lediglich durch einen höheren Rechenbedarf. Das liegt vor allem darin begründet, dass die benötigte Rechenleistung des Max-N-Wrappers erst bei größeren Suchtiefen signifikant ansteigt, was sich bei MaxN[TD3T]$_3$ beobachten lässt. So benötigt dieser Agent im Rahmen der Evaluation mehr Rechenkapazitäten als MCTS[TD3T], obwohl Letzterer schon ab 200 MCTS-Iterationen eine vergleichbare Spielstärke aufbringt.

Damit zeigt sich, dass vor allem für hochkomplexe Spiele eine Monte-Carlo-Baumsuche sinnvoller sein kann als eine Minimax-Suche. Denn komplexe Brettspiele erfordern häufig hohe Suchtiefen, durch die eine Minimax-Suche an ihren Grenzen geraten kann.

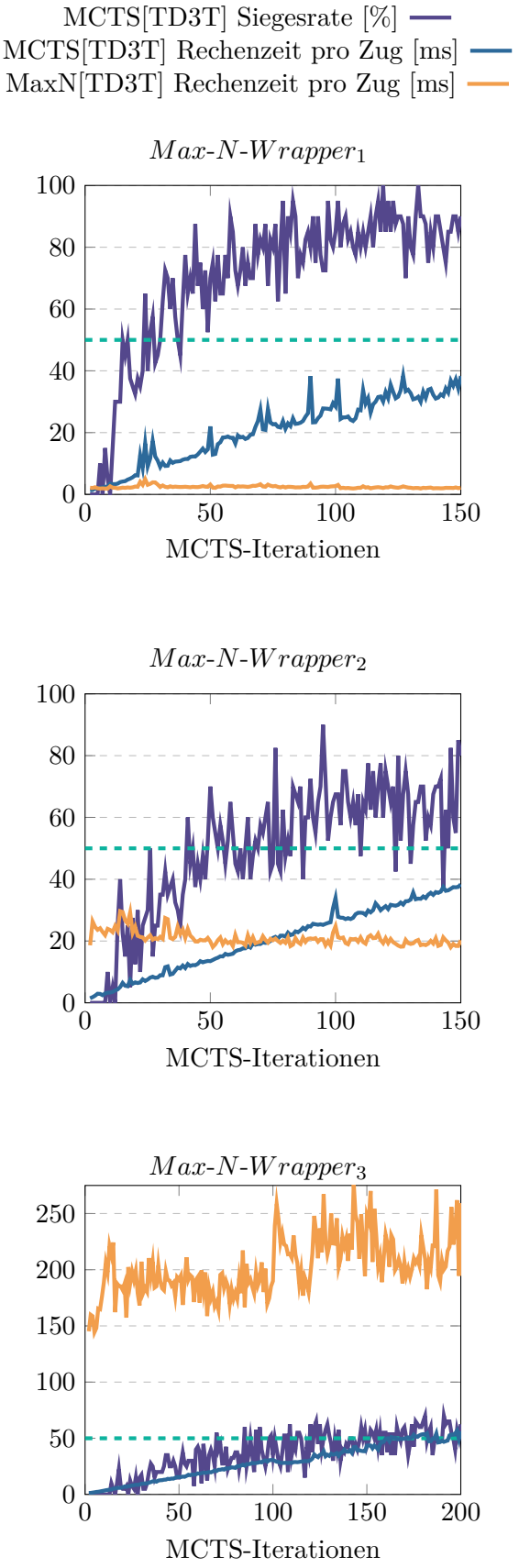

Abbildung 27 Vergleich von MCTS- und Max-N-Wrapper bezüglich Spielstärke und Zeitverhalten.

7.3 Edax

Bei Edax handelt es sich um ein hochoptimiertes, von Richard Delorme in der Programmiersprache C entwickeltes Othello-Programm[4], welches eine sehr starke Spielstärke aufweist.

7.3.1 Aufbau des Experiments

Das GBG-Framework stellt bereits eine Integration des Edax-Programms zur Verfügung, wodurch gewährleistet ist, dass Letzteres sich wie eine native *PlayAgent*-Instanz verhält. Das vereinfacht den Evaluationsablauf ungemein. Ein Edax-Agent mit der Suchtiefe n wird im Folgenden durch Edax_n bezeichnet. Obwohl die Software dazu im Stande ist, weitaus tiefer zu suchen, bewerten Liskowski et al. [33, S. 8] den Agenten ab einer Suchtiefe größer drei als so stark, dass es sich schwierig gestalte, die Siegeschancen gegen Edax zu erhöhen. Dementsprechend fiel die Siegesrate ihres entwickelten Convolutional Neural Networks (CNNs) gegen Edax_3 unter die 50-Prozent-Marke.
In stichprobenartigen Spielen zeigte sich bereits, dass der MCTS[TD3T]-Agent bei 10.000 MCTS-Iterationen als schwarzer Spieler in der Lage ist, Edax_{1-9} zu besiegen. Diesen Ergebnissen fehlt es jedoch an Aussagekraft, da sowohl der zur Approximation verwendete TD-3-Tupel-Agent als auch Edax Spielzüge deterministisch wählen. Das bedeutet, dass Spiele mit gleichen Startzuständen immer in identischen Endzuständen resultieren. Um Evaluationsergebnisse mit einer höheren Belastbarkeit zu erhalten, ist es damit notwendig, eine Vielzahl von Spielpartien ausgehend von verschiedenen Startpositionen zu simulieren. Dabei sollen die teilnehmenden Spielagenten in jeder Ausgangsposition die Möglichkeit haben, beide verfügbaren Spielrollen einzunehmen. Durch dieses Vorgehen lässt sich ein größerer Umfang an unterschiedlichen Spieldaten erheben.
Konkret werden in der nachfolgenden Evaluation die Agenten MCTS[TD3T] und Edax ausgehend von zehn verschiedenen Startpositionen gegen einander antreten. Dabei spielt MCTS[TD3T] jeweils mit den Iterationsanzahlen $\{0, 1.000, 2.000, 3.000, ..., 10.000\}$ gegen Edax_n-Instanzen mit Suchtiefen von eins bis zehn. In Summe ergeben sich damit 2.200 zu simulierende Spielpartien. Da diese bereits etwa 24 Stunden Rechenzeit in Anspruch nehmen, ist in diesem Rahmen eine umfangreichere Erhebung bedingt durch die gegebene Zeitbeschränkung der vorliegenden Arbeit nicht möglich.
Zuletzt sei erwähnt, dass zum Zweck einer höheren Variation und Aussagekraft der erhobenen Daten nicht alle Spielpartien dieselben zehn Ausgangsposition verwenden. Vielmehr werden für jede der 110 eindeutigen Teilnehmerkonstellationen zwischen den Agenten per Zufall zehn unterschiedliche Ausgangspositionen bestimmt. Die stichprobenartige Selektion von Startpositionen basiert dabei auf einer bereits vorhandenen Funktionalität des GBG-Framework, welche 244 verschiedene Othello-Startkonfigurationen bereitstellt.

[4] Der quelloffene Edax-Programmcode ist unter [32] zugänglich

7.3.2 Beurteilung der Ergebnisse

Die Resultate des im Vorfeld beschriebenen Evaluationsprozesses sind nachfolgend durch Abbildung 28 visualisiert.

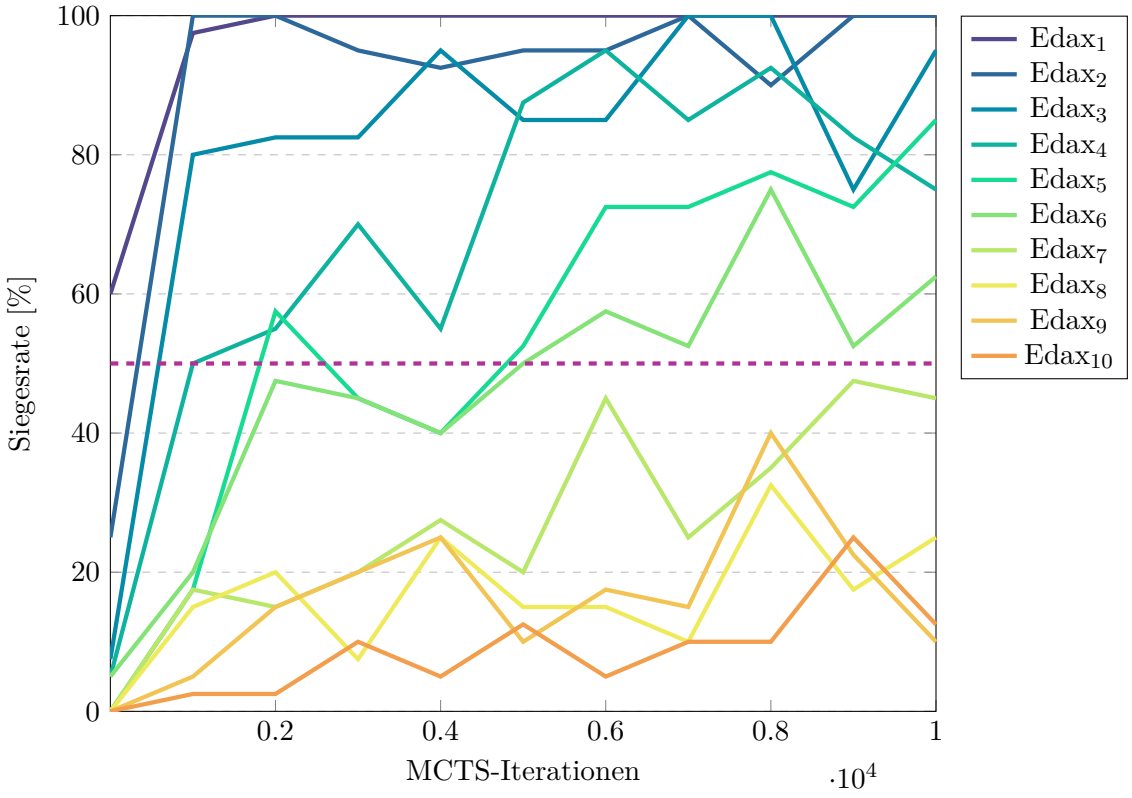

Abbildung 28 Siegesraten aus der Sicht von MCTS[TD3T] gegen Edax-Agenten mit unterschiedlichen Suchtiefen.

Aus dem obigen Chart lässt sich ableiten, dass die zuvor beschriebenen stichprobenartigen Siege gegen Edax$_{7-9}$ nicht als aussagekräftig gewertet werden können. Das ist darin begründet, dass die zugrunde liegenden Siegesraten zu keinem Zeitpunkt die 50-Prozent-Marke überschreiten, was jedoch notwendig ist, um die Überlegenheit eines Spielagenten schlussfolgern zu können. Damit lässt sich aus der Grafik ableiten, dass der MCTS[TD3T]-Agent ab 5.000 Iterationen den Agenten Edax$_{1-6}$ überlegen ist. Ab einer Suchtiefe von sieben kann jedoch keiner der evaluierten MCTS[TD3T]-Agenten mithalten.
Mit der Überlegenheit von MCTS[TD3T]$_{\geq 5.000}$ gegenüber Edax$_{1-6}$ ist ein äußerst zufriedenstellendes Resultat erreicht worden, da Edax's Spielstärke nicht zu vernachlässigen ist. So ist dem Autor dieser Arbeit kein Agent des GBG-Framework bekannt, der Edax zuvor ab einer Suchtiefe von drei besiegen konnte. Zudem beschreiben Liskowski et al. das Edax-Programm als stärksten Open-Source-Agenten für das Spiel Othello [33, S. 1].
Dennoch sollte berücksichtigt werden, dass die hier präsentierten Ergebnisse nicht eine solche Aussagekraft wie die Evaluationen von Liskowski et al. bieten können. So ließen Letztere ihren entwickelten CNN-Agenten ausgehend von 1.000 verschiedene Startpositionen gegen Edax antreten [33, S. 8].

8 Hürden und Fallstricke

In diesem Kapitel werden einige Hürden und Fallstricke erläutert, mit denen der Autor im Zusammenhang mit der vorliegenden Arbeit konfrontiert war. Denn vor allem die Entwicklung eines AlphaZero-inspirierten Agenten gestaltete sich komplizierter als zunächst angenommen. Damit sind die meisten entstandenen Herausforderungen dem Entwicklungsprozess zuzuordnen. So gingen der ersten funktionierenden Version des MCTS-Wrappers zwei weniger erfolgreiche Prototypen voraus. Diese führten neben einigen frustrierenden Arbeitstagen zu einer Abweichung des im Vorfeld definierten Projektplans.

8.1 Erste entwickelte Version

Wie auch die finale Version wurde der erste Prototyp des MCTS-Wrapper-Agenten gegen einen TD-3-Tupel-Agenten implementiert. Allerdings waren die Funktionalitäten des Agenten eng mit der Monte-Carlo-Baumsuche verdrahtet, was eine erhöhte Komplexität des Quellcodes nach sich zog. Dies ist unter anderem darin begründet, dass der Implementierungsprozess bereits in frühen Phasen des Projekts begann, da er eine Möglichkeit bot, die aus verschiedener Literatur gelernten Erkenntnisse durch eine praktische Umsetzung zu verinnerlichen. Das resultierte vor allem in den Anfangsphasen der Softwareimplementierung in diversen Programmierfehlern.
Nachdem dieser erste Prototyp fertiggestellt war, zeigte sich, dass er nicht im Ansatz siegorientiert spielte. Genau genommen ließ sich dem Verhalten des Spielagenten keine konkrete Strategie entnehmen. So schienen zwischenzeitlich einige Spielaktionen vielversprechend, während andere den Eindruck erweckten, der Agent würde eine Niederlage anstreben. Ein Vergleich des um den MCTS-Wrapper erweiterten TD-3-Tupel-Agenten mit der gleichen Version seiner selbst, die nicht um einen zusätzlichen Lookahead augmentiert wurde, zeigte, dass Letztere deutlich stärker spielte. Denn diese gewann jedes Spiel gegen den durch die Monte-Carlo-Baumsuche erweiterten Agenten. Damit war klar, dass ein schwerwiegendes Problem in der Software besteht.
Da sich der durch die Monte-Carlo-Baumsuche gewonnene Lookahead negativ auf die Spielleistung auswirkte, wurde im Folgenden nach Fehlern in der MCTS-Implementierung gesucht. Zur Überprüfung der Funktionsweise des Algorithmus ist anschließend der approximierende Agent durch ein Random-Rollout-Verfahren ersetzt worden, was unmittelbar eine signifikante Verbesserung der Spielstärke nach sich zog. Daraus leitete sich die Erkenntnis ab, dass die Ursache in der approximierten Bewertung $(v, \vec{p})$ des Spielagenten liegen muss.
Letztlich stellte sich heraus, dass die von der Klasse *TDNTuple3Agt* bereitgestellte *getScore*-Methode nicht ordnungsgemäß verwendet wurde. So unterstützt diese Methode die Bewertung eines übergebenen Spielzustands nur bedingt. Diese gibt zwar immer einen skalaren Wert zurück, doch ist dieser lediglich dann aussagekräftig, wenn ein Spielzustand s aus der Sicht des Spielers bewertet wurde, der im vorherigen Zustand am Zug war und damit den Spielzustand s erzeugt hat. Anderenfalls liegt die approximierte Bewertung weniger dem Lernprozess des TD-N-Tupel-Netzwerks als dessen zufällig initialisierten Gewichten zugrunde.

8.2 Zweite entwickelte Version

Auch wenn der erste implementierte Prototyp keine Spielerfolge verzeichnen konnte, so brachte dieser neben einem enormen Lerneffekt die Erkenntnis mit sich, wie eine ordnungsgemäße Verwendung der *getScore*-Methode aussieht. Allerdings gilt es die Ausnahmesituation zu beachten, dass die von *PlayAgent* spezifizierte *getScore(StateObservation)*-Methode von *TDNTuple3Agt* lediglich durch einen erzeugten Laufzeitfehler implementiert wird. Bedingt durch die Arbeitsweise des zugrunde liegenden neuronalen Netzwerks stellt die Klasse diese Methode mit einer um einen zusätzlichen Parameter erweiterten Signatur durch *getScore(StateObservation, StateObservation)* zur Verfügung. Diese Abweichung von der Spezifizierung des *PlayAgent*-Interface erweist sich zunächst als unerwartetes Verhalten und damit als Verletzung des Liskovschen Substitutionsprinzips objektorientierter Programmierung. Denn die Funktionalität ist damit lediglich Objekten der Klasse *TDNTuple3Agt* vorbehalten und für *PlayAgent*-Instanzen nur nach einer obligatorischen Typumwandlung zugänglich.
Um zur Approximation unterschiedliche *PlayAgent*-Instanzen heranziehen zu können, wird in der zweiten Version des MCTS-Wrapper-Agenten die *getNextAction2*-Methode zur Zustandsbewertung verwendet. Denn diese wird in Regel von jeder Klasse, welche das *PlayAgent*-Interface implementiert, so zur Verfügung gestellt, dass nicht mit einem unerwarteten Verhalten gerechnet werden muss. Zudem enthält die Rückgabe von *getNextAction2* durch *vTable* einen Vektor mit skalaren Bewertungen der ausgehend von einem Zustand s verfügbaren Spielaktionen. Diese Einschätzungen ergeben sich dabei aus der Perspektive des Spielteilnehmers, welcher in s am Zug ist. Damit besteht ein Unterschied zur *getScore*-Methode, die eine Bewertung aus Sicht des Spielers bereitstellt, dessen Aktion den Spielzustand s erzeugt hat.
Die konkrete Planung des zweiten Prototyps sah aufgrund der in der ersten Version aufegretenen Problematik vor, die Implementierung der Monte-Carlo-Baumsuche zunächst auszulassen und die auf einem Spielagenten gestützte Approximation von Zustandsbewertungen zu fokussieren. Zu diesem Zweck wurde der bereits im GBG-Framework verfügbare MCTS-Agent als Basis genommen. Letzterer wurde anschließend so umgebaut, dass dessen Random-Rollouts durch den auf der *getNextAction2*-Methode basierenden Approximationsvorgang des TD-3-Tupel-Agenten ersetzt wurden.
Die Funktionsweise des daraus entstandenen Prototyps wurde zunächst unter der Annahme bewertet, dass ein minimaler MCTS-Lookahead, welcher lediglich die direkten Kindknoten eines Zustands s berücksichtigt, die gleiche Spielstärke aufweist wie der blanke zugrunde liegende Agent ohne zusätzlichen Lookahead. Um das zu überprüfen, musste die Anzahl an durchzuführenden MCTS-Iterationen in jedem Spielzug dynamisch festgelegt werden. Das hat Grund, dass jede MCTS-Iteration den Suchbaum um genau einen Knoten erweitert. Somit erfordert beispielsweise ein Spielzustand s, von dem aus vier verschiedene Spielaktionen möglich sind, auch vier MCTS-Iterationen zur Expansion aller Kindknoten. In der Tat zeigte sich diese Überprüfung erfolgreich. So selektierte der um einen MCTS-Lookahead erweiterte TD-3-Tupel-Agent verglichen mit dem blanken TD-3-Tupel-Agenten in jeder Spielrunde dieses Experiments stets einen gleichenwertigen Spielzug.

Der Prototyp wies jedoch in Kombination mit höheren Iterationszahlen eine verschlechterte Spielleistung auf. Um eine Begründung für dieses Verhalten zu finden, ging die anschließende Fehlersuche von den im Folgenden beschriebenen drei jeweils möglichen Ursachen aus:

- Möglicherweise verläuft die Backpropagation fehlerhaft, da Passaktionen im GBG-Framework nicht abgebildet werden und somit im Rahmen der *advance*-Methode auftretende Passzustände übersprungen werden.
- Des Weiteren kann das Verhalten teilweise durch die verwendete UCT-Policy bedingt sein, da diese im Gegensatz zum PUCT-Algorithmus keinen Gebrauch von den approximierten Aktionswahrscheinlichkeiten $\vec{p}$ macht.
- In dem Fall, dass die Berücksichtigung der vorherigen Punkte keine Abhilfe schafft, könnte ein Training des TD-3-Tupel-Agenten unter Zuhilfenahme einer Monte-Carlo-Baumsuche mit einer hohen Iterationszahl das Spielverhalten verbessern.

8.3 Dritte entwickelte Version

Zuletzt bot der in [31] präsentierte rekursive Ansatz einer Monte-Carlo-Baumsuche die Inspiration für die erste funktionsfähige Version des MCTS-Wrappers. In dieser wird zur Selektion der PUCT-Algorithmus verwendet und die Berücksichtigung von Passaktionen durch eine entsprechende Softwarearchitektur gewährleistet. Bereits der daraus resultierende MCTS-Wrapper-Agent war im Stande, die gleiche Spielleistung aufzubringen wie der finale in diesem Projekt entstandene Agent. Allerdings fiel auf, dass der entwickelte Spielagent vor allem in späten Spielverläufen und bei hohen Iterationszahlen mehr Zeit zur Auswahl von Spielaktionen benötigte. So waren bei 10.000 MCTS-Iterationen in den Endzügen einer Spielpartie teilweise mehr als zehn Minuten zur Bestimmung einzelner Spielzüge notwendig. Im Vergleich dazu benötigte eine Aktionsauswahl im Anfangsstadium eines Spiels nur einige Sekunden.
Es stellte sich heraus, dass dieses Verhalten durch das Speichern von kindknotenspezifischen Informationen in global zugänglichen Zuordnungstabellen begründet war. Letztere ersetzten in dieser Version den Monte-Carlo-Suchbaum, indem sie alle für die Baumsuche relevanten Daten speicherten. Die Datenmenge wurde in Kombination mit hohen Iterationszahlen im Verlauf eines Spiels so groß, dass die regelmäßig stattfindenden Extraktionen von knotenzugehörigen Informationen einen enormen Rechenbedarf für Suchoperationen nach sich zogen.
In diesem Rahmen ist die *MCTSNode*-Klasse entstanden, welche als verkettete Liste von Baumknoten zugleich den gesamten Suchbaum repräsentiert. Diese Architektur macht das Speichern knotenspezifischer Informationen in globalen Zuordnungstabellen überflüssig, da die Daten direkt durch die entsprechenden Knoten bereitgestellt werden. Durch diese Anpassung entstand die finale Version des MCTS-Wrapper-Agenten.

8.4 Anbindung von Ludii

Bei dem durch die Universität Maastricht entwickelten General Game System Ludii handelt es sich um eine Plattform, die zunächst dem GGP zugeordnet werden kann. Jedoch besteht die Ausnahme, dass Ludii keinen Gebrauch von GDL macht, sondern eine eigene deklarative Sprache bereitstellt, mit der Spiele der Ludii-Umgebung zur Laufzeit zugeführt werden können. Letztere definiert sogenannte Ludeme, welche jeweils eine Repräsentation einzelner Spielkonzepte darstellen und damit bausteinartig zu den verschiedensten Brettspielen zusammengesetzt werden können [34].
In einem Projekt, welches der vorliegenden Arbeit vorausging, wurde bereits eine Schnittstelle entwickelt, mit der sich exportierte Spielagenten des GBG-Framework in der Ludii-Umgebung der Version 1.0.2 importieren lassen. Diese ermöglicht es, dass die GBG-Agenten im kompetitiven Vergleich gegen die von Ludii bereitgestellten Agenten antreten können. Ursprünglich war im Rahmen dieser Arbeit die Evaluation eines um den MCTS-Wrapper erweiterten TD-3-Tupel-Agenten in der Ludii-Umgebung geplant. Allerdings entstanden dabei diverse Komplikationen, da der entwickelte MCTS-Wrapper einige Abhängigkeiten benötigt, die nicht durch das Ludii-Interface bereitgestellt werden, da diese im vorherigen Projekt nicht notwendig waren. Ein Beispiel hierfür ist durch die Liste der vergangenen Spielaktionen einer StateObservation-Instanz gegeben, welche dazu verwendet wird, den aufgebauten Monte-Carlo-Suchbaum über ein Spiel hinweg fortzuführen. Trotz der nachträglichen Anpassung der Ludii-Schnittstelle kam es im Rahmen dieser Arbeit dennoch zu Laufzeitfehlern, wenn das Interface in Kombination mit dem MCTS-Wrapper genutzt wurde. Es war im zeitlichen Umfang dieses Projekts nicht möglich, die dahinter liegenenden Ursachen ausfindig zu machen und zu beheben.
Bedingt durch diese Problematik wurde damit die Evaluation des MCTS-Wrappers in der Ludii-Umgebung verworfen. Zudem kommt die Tatsache, dass der im GBG-Framework verfügbare Spielagent `TCL3-fixed6_250k-lam05_P4_H001-diff2-FAm.agt.zip` im Rahmen des vorangegangenen Informatikprojekts ohnehin den stärksten Ludii-Agenten der Version 1.0.2 ohne die Notwendigkeit eines Lookaheads bei 100 Spielen mit einer 60-prozentigen Siegesrate geschlagen hat. Aus diesen Gründen und nicht zuletzt auch deshalb, weil das Ludii-Interface die Evaluation ausgehend von verschiedenen Startpositionen nicht unterstützt, wurde die Entscheidung getroffen, die für einen Umbau des Quellcodes erforderlichen Kapazitäten stattdessen zur Intensivierung der verbleibenden drei Evaluationen einzusetzen.

9 Fazit und Zukunftsausblick

Diese Arbeit galt der Untersuchung, ob die Adaption der AlphaZero zugrunde liegenden Prinzipien bereits im kleinen Rahmen zu neuen Höchstleistungen führt. So konnte in dem Projekt ein Wrapper-Agent für das GBG-Framework entwickelt werden, der basierend auf einer Monte-Carlo-Baumsuche einen Zukunftsblick für einen gegeben Agenten aufbaut und so dessen Spielleistung signifikant verbessert. Durch die Evaluation wurde gezeigt, dass diese Leistungssteigerung bereits mit geringen Iterationsanzahlen messbar ist. Zudem wurde dargestellt, dass der MCTS-Wrapper für komplexe Spiele einem Max-N-Wrapper vorgezogen werden sollte. Denn Letzterer wies in den Experimenten bei zunehmender Spielkomplexität in Relation zu dem benötigten Rechenbedarf schlechtere Ergebnisse auf. Das aufschlussreichste Resultat dieser Arbeit bestand darin, dass der MCTS-Wrapper unter Verwendung eines TD-3-Tupel-Agenten zur Bewertung von Spielzuständen im GBG-Framework neue Höchstleitungen in Othello-Partien gegen den hochoptimierten Edax-Agenten erreichte. So konnte Edax selbst mit der Suchtiefe sechs ausgehend von verschiedenen Startpositionen mit einer Siegesrate von mehr als 50 Prozent geschlagen werden. Damit liegt im Rahmen des GBG-Framework das bisher stärkste Spielverhalten gegen den Edax-Agenten vor.

Aufgrund der gegebenen Zeitbeschränkung dieser Arbeit konnte nicht jede aufgetretene Fragestellung untersucht und auch nicht jeder Idee nachgegangen werden. So bietet das durchgeführte Projekt eine Grundlage für weitere vertiefende Forschungsmöglichkeiten in diesem Bereich. Im Folgenden werden einige interessante Ideen für zukünftige Forschungen aufgeführt:

- Aufgrund der softwarearchitektonischen Gegebenheiten des GBG-Framework ist der implementierte MCTS-Wrapper auf das Spiel Othello beschränkt. Es wäre aufschlussreich zu ermitteln, wie sich der durch die Monte-Carlo-Baumsuche aufgebaute Lookahead in weiteren Brettspielen schlägt. Dieser Anwendungsfall bedarf einige Anpassungen des StateObservation-Interface. Zudem ist die Monte-Carlo-Baumsuche bedingt durch den rekursiven Ansatz auf 2-Spieler-Spiele beschränkt. Auch dieser Algorithmus bedarf einige Anpassungen zur Erweiterung des MCTS-Wrappers auf beliebige Spiele.

- Die Architektur der entwickelten Software-Komponenten macht es möglich, ohne großen Aufwand weitere Approximatoren zu implementieren. Das bietet eine gute Grundlage, um beispielsweise zu ermitteln, ob die Ergebnisse verbessert werden können, wenn als approximierende Komponenten wie auch bei AlphaZero auf ein DCNN gesetzt wird.

- Eine äußerst interessante Form der Erweiterung der Software besteht in der Adaption des zweiten AlphaZero zugrunde liegenden Prinzips. Da bereits das bloße Erzeugen eines Lookaheads durch eine Monte-Carlo-Baumsuche zu sehr zufriedenstellenden Resultaten führte, wäre es interessant zu evaluieren, inwiefern sich noch bessere Ergebnisse erzielen lassen, wenn eine solche Monte-Carlo-Baumsuche auch in den Trainingsprozess von trainierbaren Spielagenten eingebunden wird.

Literatur

[1] Wolfgang Ertel. *Grundkurs Künstliche Intelligenz: Eine praxisorientierte Einführung*. 1. Aufl. Computational Intelligence. OCLC: 199186307. Wiesbaden: Vieweg, 2008. 334 S. ISBN: 978-3-528-05924-8.

[2] *The Google DeepMind Challenge Match, March 2016*. AlphaGo Korea | DeepMind. publisher: DeepMind. URL: https://deepmind.com/alphago-korea (besucht am 27. 09. 2020).

[3] David Silver u. a. *AlphaZero: Shedding new light on chess, shogi, and Go*. publisher: DeepMind. 6. Dez. 2018. URL: https://deepmind.com/blog/article/alphazero-shedding-new-light-grand-games-chess-shogi-and-go (besucht am 26. 09. 2020).

[4] Louis Victor Allis. „Searching for Solutions in Games and Artificial Intelligence". OCLC: 60586781. Diss. Wageningen: Ponsen & Looijen, 1994.

[5] Peter Buxmann und Holger Schmidt, Hrsg. *Künstliche Intelligenz: Mit Algorithmen zum wirtschaftlichen Erfolg*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2019. ISBN: 978-3-662-57567-3 978-3-662-57568-0. DOI: 10.1007/978-3-662-57568-0. URL: http://link.springer.com/10.1007/978-3-662-57568-0 (besucht am 26. 09. 2020).

[6] David Silver u. a. „Mastering the game of Go with deep neural networks and tree search". In: *Nature* 529.7587 (Jan. 2016), S. 484–489. ISSN: 0028-0836, 1476-4687. DOI: 10.1038/nature16961. URL: http://www.nature.com/articles/nature16961 (besucht am 03. 10. 2020).

[7] David Silver u. a. „Mastering the game of Go without human knowledge". In: *Nature* 550.7676 (Okt. 2017). Number: 7676, S. 354–359. ISSN: 0028-0836, 1476-4687. DOI: 10.1038/nature24270. URL: http://www.nature.com/articles/nature24270 (besucht am 27. 09. 2020).

[8] David Silver und Demis Hassabis. *AlphaGo: Mastering the ancient game of Go with Machine Learning*. Google AI Blog. 27. Jan. 2016. URL: http://ai.googleblog.com/2016/01/alphago-mastering-ancient-game-of-go.html (besucht am 27. 09. 2020).

[9] Wolfgang Konen. „General Board Game Playing for Education and Research in Generic AI Game Learning". In: *IEEE Conference on Games*. Hrsg. von Diego Perez, Sanaz Mostaghim und Simon Lucas. London, 20. Aug. 2019. URL: https://arxiv.org/pdf/1907.06508.

[10] Michael Thielscher. „General Game Playing in AI Research and Education". In: *KI 2011: Advances in Artificial Intelligence*. Hrsg. von Joscha Bach und Stefan Edelkamp. Bd. 7006. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, S. 26–37. ISBN: 978-3-642-24454-4 978-3-642-24455-1. DOI: 10.1007/978-3-642-24455-1_3. URL: http://link.springer.com/10.1007/978-3-642-24455-1_3 (besucht am 29. 09. 2020).

[11] Maciej Świechowski u. a. „Recent Advances in General Game Playing“. In: *The Scientific World Journal* 2015 (2015), S. 1–22. ISSN: 2356-6140, 1537-744X. DOI: 10.1155/2015/986262. URL: http://www.hindawi.com/journals/tswj/2015/986262/ (besucht am 29. 09. 2020).

[12] David Silver u. a. „A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play“. In: *Science* 362.6419 (7. Dez. 2018), S. 1140–1144. ISSN: 0036-8075, 1095-9203. DOI: 10.1126/science.aar6404. URL: https://www.sciencemag.org/lookup/doi/10.1126/science.aar6404 (besucht am 02. 10. 2020).

[13] *Introducing NVIDIA TITAN V: The World's Most Powerful PC Graphics Card*. publisher: Nvidia Corporation. URL: https://www.nvidia.com/en-us/titan/titan-v/ (besucht am 02. 10. 2020).

[14] *Cloud TPU*. Google Cloud. publisher: Google Inc. URL: https://cloud.google.com/tpu (besucht am 03. 10. 2020).

[15] Hui Wang, Mike Preuss und Aske Plaat. „Warm-Start AlphaZero Self-Play Search Enhancements“. In: *arXiv:2004.12357 [cs]* (26. Apr. 2020). arXiv: 2004.12357. URL: http://arxiv.org/abs/2004.12357 (besucht am 30. 10. 2020).

[16] Hui Wang u. a. „Hyper-Parameter Sweep on AlphaZero General“. In: *arXiv:1903.08129 [cs]* (19. März 2019). arXiv: 1903.08129. URL: http://arxiv.org/abs/1903.08129 (besucht am 30. 10. 2020).

[17] Nai-Yuan Chang u. a. „The Big Win Strategy on Multi-Value Network: An Improvement over AlphaZero Approach for 6x6 Othello“. In: *Proceedings of the 2018 International Conference on Machine Learning and Machine Intelligence - MLMI2018*. the 2018 International Conference. Ha Noi, Viet Nam: ACM Press, 2018, S. 78–81. ISBN: 978-1-4503-6556-7. DOI: 10.1145/3278312.3278325. URL: http://dl.acm.org/citation.cfm?doid=3278312.3278325 (besucht am 30. 10. 2020).

[18] Shane Legg und Marcus Hutter. „A Collection of Definitions of Intelligence“. In: *arXiv:0706.3639 [cs]* (25. Juni 2007). arXiv: 0706.3639. URL: http://arxiv.org/abs/0706.3639 (besucht am 26. 09. 2020).

[19] Elaine Rich. *Artificial Intelligence*. McGraw-Hill Series in Artificial Intelligence. New York: McGraw-Hill, 1983. 436 S. ISBN: 978-0-07-052261-9.

[20] Klaus Mainzer. *Künstliche Intelligenz – Wann übernehmen die Maschinen?* Technik im Fokus. Berlin, Heidelberg: Springer Berlin Heidelberg, 2019. ISBN: 978-3-662-58045-5 978-3-662-58046-2. DOI: 10.1007/978-3-662-58046-2. URL: http://link.springer.com/10.1007/978-3-662-58046-2 (besucht am 26. 09. 2020).

[21] Ulrike Barthelmeß und Ulrich Furbach. *Künstliche Intelligenz aus ungewohnten Perspektiven: Ein Rundgang mit Bergson, Proust und Nabokov*. Wiesbaden: Springer Fachmedien Wiesbaden, 2019. ISBN: 978-3-658-24569-6 978-3-658-

24570-2. DOI: 10.1007/978-3-658-24570-2. URL: http://link.springer.com/10.1007/978-3-658-24570-2 (besucht am 26. 09. 2020).

[22] Manfred Schramm und Walter Rampf. „Lexmed: Diagnosesystem für Blinddarmentzündung“. In: *Dtsch Arztebl International* 98.36 (2001), S. 10. URL: https://www.aerzteblatt.de/int/article.asp?id=28556.

[23] Phil Wennker. *Künstliche Intelligenz in der Praxis: Anwendung in Unternehmen und Branchen: KI wettbewerbs- und zukunftsorientiert einsetzen*. Wiesbaden: Springer Fachmedien Wiesbaden, 2020. ISBN: 978-3-658-30479-9 978-3-658-30480-5. DOI: 10.1007/978-3-658-30480-5. URL: http://link.springer.com/10.1007/978-3-658-30480-5 (besucht am 26. 09. 2020).

[24] Stuart J. Russell und Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall Series in Artificial Intelligence. Englewood Cliffs, N.J: Prentice Hall, 1995. 932 S. ISBN: 978-0-13-103805-9.

[25] B. Bouzy und B. Helmstetter. „Monte-Carlo Go Developments“. In: *Advances in Computer Games*. Hrsg. von H. Jaap Herik, Hiroyuki Iida und Ernst A. Heinz. Boston, MA: Springer US, 2004, S. 159–174. ISBN: 978-1-4757-4424-8 978-0-387-35706-5. DOI: 10.1007/978-0-387-35706-5_11. URL: http://link.springer.com/10.1007/978-0-387-35706-5_11 (besucht am 06. 10. 2020).

[26] Sylvain Gelly und David Silver. „Monte-Carlo tree search and rapid action value estimation in computer Go“. In: *Artificial Intelligence* 175.11 (Juli 2011), S. 1856–1875. ISSN: 00043702. DOI: 10.1016/j.artint.2011.03.007. URL: https://linkinghub.elsevier.com/retrieve/pii/S000437021100052X (besucht am 06. 10. 2020).

[27] Uwe Lorenz. *Reinforcement Learning: Aktuelle Ansätze verstehen - mit Beispielen in Java und Greenfoot*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2020. ISBN: 978-3-662-61650-5 978-3-662-61651-2. DOI: 10.1007/978-3-662-61651-2. URL: http://link.springer.com/10.1007/978-3-662-61651-2 (besucht am 05. 10. 2020).

[28] Levente Kocsis und Csaba Szepesvári. „Bandit Based Monte-Carlo Planning“. In: *Machine Learning: ECML 2006*. Hrsg. von Johannes Fürnkranz, Tobias Scheffer und Myra Spiliopoulou. Bearb. von David Hutchison u. a. Bd. 4212. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, S. 282–293. ISBN: 978-3-540-45375-8 978-3-540-46056-5. DOI: 10.1007/11871842_29. URL: http://link.springer.com/10.1007/11871842_29 (besucht am 06. 10. 2020).

[29] Petr Baudiš und Jean-loup Gailly. „PACHI: State of the Art Open Source Go Program“. In: *Advances in Computer Games*. Hrsg. von H. Jaap van den Herik und Aske Plaat. Bearb. von David Hutchison u. a. Bd. 7168. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, S. 24–38. ISBN: 978-3-642-31865-8 978-3-642-31866-5. DOI: 10.1007/978-3-642-31866-5_3. URL: http://link.springer.com/10.1007/978-3-642-31866-5_3 (besucht am 11. 10. 2020).

[30] Alex Krizhevsky, Ilya Sutskever und Geoffrey E. Hinton. „ImageNet classification with deep convolutional neural networks“. In: *Communications of the ACM* 60.6 (24. Mai 2017), S. 84–90. ISSN: 0001-0782, 1557-7317. DOI: 10.1145/3065386. URL: https://dl.acm.org/doi/10.1145/3065386 (besucht am 14. 10. 2020).

[31] Surag Nair. *A Simple Alpha(Go) Zero Tutorial*. 29. Dez. 2017. URL: https://web.stanford.edu/~surag/posts/alphazero.html (besucht am 18. 10. 2020).

[32] Richard Delorme. *abulmo/edax-reversi*. URL: https://github.com/abulmo/edax-reversi (besucht am 27. 10. 2020).

[33] Pawel Liskowski, Wojciech Jaskowski und Krzysztof Krawiec. „Learning to Play Othello With Deep Neural Networks“. In: *IEEE Transactions on Games* 10.4 (Dez. 2018), S. 354–364. ISSN: 2475-1502, 2475-1510. DOI: 10.1109/TG.2018.2799997. URL: https://ieeexplore.ieee.org/document/8276588/ (besucht am 27. 10. 2020).

[34] Éric Piette u. a. „Ludii - The Ludemic General Game System“. In: *arXiv:1905.05013 [cs]* (21. Feb. 2020). arXiv: 1905.05013. URL: http://arxiv.org/abs/1905.05013 (besucht am 30. 10. 2020).

Anhang

Die im Rahmen der Evaluation erhobenen Spieldaten befinden sich zusammen mit dem in diesem Projekt entwickelten Quellcode auf einem USB Stick, welcher als digitaler Anhang dieser Arbeit auf der letzten Seite angefügt ist.