Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Konstantin Böhm

Maschinelles Lernen: Vergleich von Monte Carlo Tree Search und Reinforcement Learning am Beispiel eines Perfect Information Game

Fakultät Technik und Informatik
Studiendepartment Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Konstantin Böhm

Maschinelles Lernen: Vergleich von Monte Carlo Tree Search und Reinforcement Learning am Beispiel eines Perfect Information Game

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung

im Studiengang Bachelor of Science Angewandte Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Michael Neitzke
Zweitgutachter: Prof. Dr.-Ing. Andreas Meisel

Eingereicht am: 15. Dezember 2016

Konstantin Böhm

Thema der Arbeit
Maschinelles Lernen: Vergleich von Monte Carlo Tree Search und Reinforcement Learning am Beispiel eines Perfect Information Game

Stichworte
Künstliche Intelligenz, Monte Carlo Tree Search, Bestärkendes Lernen, Maschinelles Lernen, Q-Learning, Vier Gewinnt

Kurzzusammenfassung
Inhalt dieser Arbeit ist der Vergleich von zwei Verfahren aus dem Bereich des maschinellen Lernens: *Monte Carlo Tree Search* (MCTS) und *Q-Learning*. Dafür sind zwei künstliche Intelligenzen mit diesen Verfahren entwickelt worden und in dem Spiel *Vier Gewinnt* gegeneinander angetreten. Es wird gezeigt, dass bei überschaubaren Zustandsräumen das Lernverfahren *Q-Learning* bessere Ergebnisse liefert, wohingegen MCTS auch bei zu großen Zustandsräumen noch gute Ergebnisse durch Berechnung zur Laufzeit und Aufspannen eines Suchbaums liefern kann.

Konstantin Böhm

Title of the paper
Machine Learning: Comparing Monte Carlo Tree Search and Reinforcement Learning Using the Example of a Perfect Information Game

Keywords
Artificial Intelligence, Monte Carlo Tree Search, Reinforcement Learning, Machine Learning, Q-Learning, Connect Four

Abstract
The content of this work is the comparison of two techniques in the field of machine learning: *Monte Carlo Tree Search* (MCTS) and *Q-Learning*. To achieve that two artificial intelligences with these techniques were constructed and then competed against each other in the game of *Connect Four*. It is shown that in manageable state spaces the learning method Q-Learning provides better results, while MCTS can still provide good results even in the case of oversized state spaces by calculating during runtime and building a search tree.

Inhaltsverzeichnis

Abbildungsverzeichnis

List of Algorithms

Danksagung

Zunächst möchte ich mich bei Christiane Böhm, Daniel Röttger und Prof. Dr. Michael Böhm bedanken, die meine Arbeit Korrektur gelesen haben und mir durch konstruktive Kritik geholfen haben, diese Arbeit zu verbessern.

Ganz besonderer Dank gilt an dieser Stelle Prof. Dr. Michael Neitzke nicht nur für die Annahme der Arbeit, sondern auch für die Unterstützung bei der Umsetzung. Darüber hinaus möchte ich mich auch für lehrreiche und spannende Vorlesungen und das Projekt "Lernende Agenten" bedanken, wodurch mein Interesse an dem Thema der künstlichen Intelligenz erst geweckt wurde.

Schließlich möchte ich mich auch bei meinem Arbeitgeber Felix Gliesche bedanken, der mich nicht nur mit flexiblen Arbeitszeiten, sondern auch mit der Möglichkeit, in seinem Büroraum die Arbeit umzusetzen zu können, unterstützt hat.

1 Einleitung

Es gibt unzählige Brettspiele, die vermeintlich simpel erscheinen, da sie meist aus einfachen Spielbrettern und einer überschaubaren Menge an Spielsteinen bestehen, wie z.B. Schach, Mühle oder Dame. Betrachtet man die Spiele genauer, merkt man allerdings schnell, dass es doch deutlich schwieriger ist diese Spiele zu meistern. Schach hat zum Beispiel geschätzte $10^{46.25}$ (Chinchalkar, 1996) verschiedene Positionen, in denen die Figuren angeordnet sein können. Ein Mensch wird wahrscheinlich niemals so viele Positionen berücksichtigen können. Künstliche Intelligenzen hingegen besitzen das Potential den Menschen deutlich zu übertreffen und tun dies auch schon in vielen Spielen, wie zum Beispiel Schach (Levy, 2005) und dem altem chinesischen Brettspiel Go (BBC News, 2016), erfolgreich. Ein $19 * 19$ Go-Spielfeld besitzt sogar geschätzte $2.08168199382 * 10^{170}$ verschiedene Positionen und ist damit noch komplexer als Schach. Das Programm *AlphaGo* besiegte im März 2016 erstmals einen der besten Go-Spieler auf einem $19 * 19$ Spielfeld, was als ein bahnbrechendes Ereignis in der künstlichen Intelligenz angesehen wird (BBC News, 2016). *AlphaGo* nutze dabei die Technik des *Monte Carlo Tree Search* für seinen Erfolg. Inhalt dieser Arbeit wird es sein, genau diese Technik zu betrachten und es mit einem älteren Verfahren aus dem Bereich des maschinellen Lernens zu vergleichen.

1.1 Zielsetzung

Ziel dieser Arbeit ist es, das Verfahren der *Monte Carlo Tree Search* zu untersuchen. Dafür soll es mit einem älteren Verfahren aus dem Bereich des maschinellen Lernens verglichen werden. Die Lernverfahren des *Reinforcement Learnings* haben sich schon über Jahre als bewährte Varianten erwiesen. In dieser Arbeit wurde sich für die Technik *Q-Learning* entschieden, da es nicht nur bewiesen den optimalen Lösungsweg finden kann (Russell und Norvig, 2009, 830), sondern auch kein Wissen über seine Umwelt benötigt (Russell und Norvig, 2009, 843) und damit einfacher zu implementieren ist.

Im Bereich der künstlichen Intelligenz gibt es viele Anwendungsfelder, wie zum Beispiel bei Suchmaschinen, Robotern, selbstfahrende Fahrzeuge, der Gesichtserkennung und im Flugzeugbau. Da allerdings viele der genannten Beispiele häufig hoch komplex sind, würde es den Rahmen dieser Arbeit sprengen, wenn man versucht eine künstliche Intelligenz (KI) an einem

solchen Beispiel zu vergleichen. Daher werden stattdessen häufig Computerspiele genutzt, die je nach Spiel beliebig komplex sein können, aber durch klare Regeln leicht umzusetzen sind. Daher werden auch in dieser Arbeit die beiden Verfahren mit Hilfe von einem Spiel verglichen.

Ziel wird es sein, zwei KI zu entwickeln, gegeneinander in einem Spiel antreten zu lassen und dann die Ergebnisse zu untersuchen. Es sollen dabei die Stärken und Schwächen der Verfahren aufgezeigt und gegenüber gestellt werden. Besonders das Verfahren des *Monte Carlo Tree Search*, das mit *AlphaGo* zuletzt einen großen Erfolg erreicht hat, ist dabei interessant zu betrachten.

1.2 Aufbau

Diese Arbeit wurde in fünf Kapitel unterteilt. Das zweite Kapitel stellt das gewählte Spiel für den Vergleich von den beiden künstlichen Intelligenzen *Vier Gewinnt* vor. Es werden dabei die Spielregeln erklärt und einige wichtige Fakten zu dem Spiel erläutert.

Um den Inhalt besser verstehen zu können, ist es wichtig die wissenschaftliche Basis dieser Arbeit zu kennen. Daher gibt das dritte Kapitel nicht nur einen Einstieg in das Thema des *Reinforcement Learning* und der *Monte Carlo Tree Search*, sondern erklärt auch noch die Grundlagen, die für die beiden Algorithmen von Bedeutung sind.

Das vierte Kapitel erklärt auf welche Art und Weise die beiden Algorithmen umgesetzt wurden und welche Technologien dabei zum Einsatz kamen. Es wird mit dem Konzept der Implementation eingeleitet und dann liegt der Fokus auf der Implementation des Spiels und der künstlichen Intelligenzen.

Welche Experimente gemacht wurden, um die beiden Algorithmen zu vergleichen, wird im fünften Kapitel erzählt. Zudem werden auch die Ergebnisse der Experimente ausgewertet und die beiden Algorithmen gegenübergestellt.

Im letzten Kapitel wird das Ergebnis dieser Arbeit noch einmal zusammengefasst und darüber hinaus wird ein Ausblick gegeben, wo man ansetzen könnte, wenn man das Projekt noch verbessern bzw. erweitern möchte.

2 Vier Gewinnt

Um Monte Carlo Tree Search und Reinforcement Learning zu vergleichen, wurde in dieser Arbeit das Spiel *Vier Gewinnt* gewählt. Das Spiel besitzt keinen Zufallsfaktor, überschaubare Regeln und genügend Komplexität (siehe Unterkapitel "Komplexität"), um es für den Vergleich der beiden Algorithmen interessant zu machen. Erst werden in diesem Kapitel die Regeln des Spiels vorgestellt und danach auf die Komplexität des Spiels eingegangen. Um später die Umsetzung der künstlichen Spieler zu erleichtern, wird auch noch der beste Eröffnungszug erläutert.

2.1 Spielregeln

Vier Gewinnt ist ein Strategiespiel für zwei Spieler. Jedem Spieler wird eine eindeutige Farbe zugeordnet (i.d.R. Gelb und Rot) und von dieser Farbe besitzt jeder Spieler jeweils 21 identische Spielsteine. Es gibt ein senkrecht stehendes viereckiges Spielfeld, das aus 7 Spalten besteht, die jeweils 6 Zellen besitzen. Jede Zelle kann nur genau einen Spielstein beinhalten. Das Spielfeld ist zu Beginn leer. Bei einem Spielzug (kurz Zug) darf ein Spieler einen Spielstein in eine der Spalten werfen, solange diese nicht schon vollends belegt ist. Der Spielstein rutscht dabei immer auf die unterste nicht besetzte Zelle der Spalte und belegt diese. Die Spieler machen abwechselnd ihre Züge. Es gibt keine genaue Regel, die besagt, welcher Spieler beginnt. In dieser Arbeit bekommen die Spieler die Farben Schwarz und Weiß, dabei wird immer angenommen, dass der Spieler mit der Farbe Weiß das Spiel beginnt und als Spieler 1 bezeichnet wird. Ziel der Spieler ist es nun vier ihrer Spielsteine in einer Reihe zu platzieren. Die Reihen dürfen horizontal, vertikal und auch diagonal sein. Wenn ein Spieler vier Spielsteine in einer Reihe hat, gilt das Spiel als beendet und dieser Spieler hat gewonnen - der andere Spieler verloren. Wurden allerdings alle Spielsteine gelegt und kein Spieler konnte vier Spielsteine in eine Reihe legen, gilt das Spiel als unentschieden.

Abbildungen 2.1, 2.2, 2.3 zeigen drei verschiedene Siegsituationen für den Spieler Weiß und Abbildung 2.4 zeigt ein unentschiedenes Spiel:

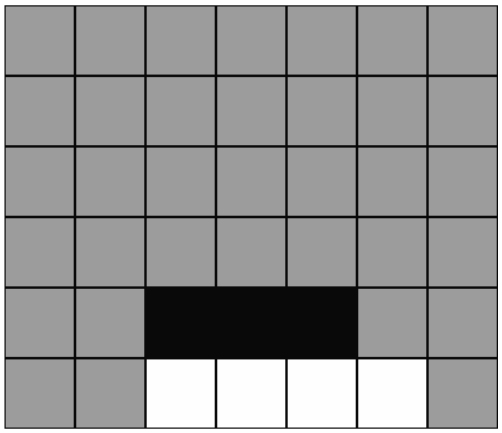

Abbildung 2.1: Weiß gewinnt horizontal

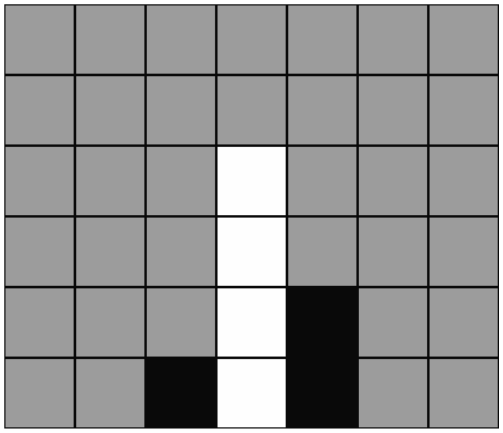

Abbildung 2.2: Weiß gewinnt vertikal

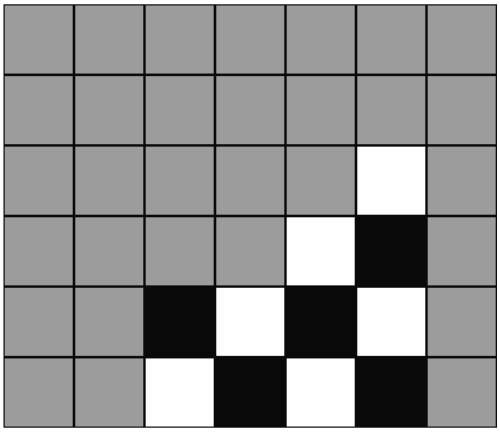

Abbildung 2.3: Weiß gewinnt diagonal

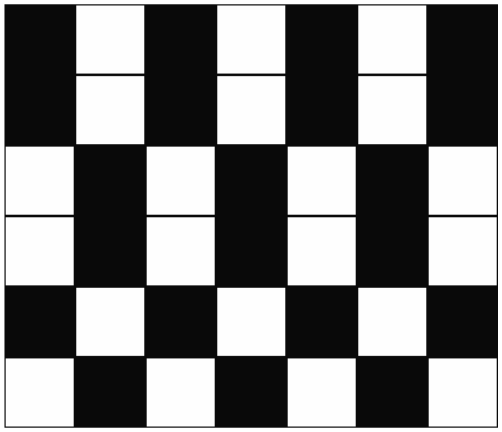

Abbildung 2.4: Unentschieden

2.2 Komplexität

Das Spielfeld besteht aus $7 * 6 = 42$ Zellen und jede Zelle kann 3 Zustände annehmen: Leer, Schwarz oder Weiß. Daher kann man grob sagen, dass das Spiel $3^{42} (\geq 10^{20})$ verschiedene Zustände annehmen kann. Diese Zahl ist allerdings sehr ungenau, da viele Zustände nie erreicht werden können, denn z.B. wird hier nicht berücksichtigt, dass sich die Anzahl zwischen schwarzen und weißen Spielsteinen maximal um eins unterscheiden kann. Zudem können einige Zustände nicht erreicht werden, bei denen mehrere Siegbedingungen erfüllt wurden (z.B. können nicht beide Spieler eine Viererreihe gleichzeitig haben). Wenn diese und noch andere nicht erreichbaren (illegalen) Zustände herausrechnet werden, kommt man vom leeren Spielfeld auf $4, 531, 985, 219, 092$ verschiedene Zustände (Edelkamp und Kissmann, 2008, 186).

2.3 Eröffnung

Allis (1988) ist erstmals zu dem Ergebnis gekommen, dass der beginnende Spieler immer gewinnen kann, solange er den ersten Stein in die mittlere Spalte wirft und danach immer perfekt spielt. Daher ist die bestmögliche Eröffnung den Stein in der mittleren Spalte zu positionieren.

3 Grundlagen

In diesem Kapitel werden die Grundlagen für diese Arbeit erläutert. Inhalt dieser Arbeit ist es zwei Algorithmen der künstlichen Intelligenz anhand des Spiels *Vier Gewinnt* zu vergleichen. Daher wird im ersten Unterkapitel auf einige für dieses Thema wichtigen Begriffe der Spieltheorie eingegangen. Danach werden Multi-Armed Bandits und einige damit verbundene Techniken, die wichtig für das Verständnis der beiden Algorithmen sind, im zweiten Unterkapitel vorgestellt. Der erste Algorithmus stammt aus dem Bereich des maschinellen Lernens, genauer dem *Reinforcement Learning*, das im dritten Unterkapitel erläutert wird. Der zweite Algorithmus entspringt einer Familie von Algorithmen, die als *Monte Carlo Tree Search* bezeichnet werden. *Monte Carlo Tree Search* basiert auf dem Verfahren von *Monte Carlo methods*, welches das vierte Unterkapitel bilden. Im fünften und letzten Unterkapitel wird dann *Monte Carlo Tree Search* vorgestellt.

3.1 Spieltheorie

Eines der größten Anwendungsfelder der künstlichen Intelligenz sind Spiele. Die Spieltheorie ist ein Untergebiet der Mathematik und befasst sich mit Entscheidungsproblemen in Form von Spielen. In diesem Abschnitt werden die für diese Arbeit relevanten Begriffe aus der Spieltheorie erläutert. Wenn nicht anders angegeben, wurden die Inhalte nach Russell und Norvig (2009, 161-163) wiedergegeben.

3.1.1 Das Spiel

Ein Spiel wird aus folgenden Elementen definiert:

1. s_0: Der initiale Zustand des Spiels.
2. $PLAYER(s)$: Bestimmt welcher Spieler im Zustand s an der Reihe ist.
3. $ACTIONS(s)$: Beschreibt alle möglichen (erlaubten) Aktionen bzw. Züge im Zustand s.

4. $RESULT(s, a)$: Wird als das *transition model* bezeichnet und gibt an, welches Ergebnis eine Aktion a im Zustand s ergibt.

5. $TERMINAL-TEST(s, a)$: Der *terminal-test* testet, ob eine Aktion a im Zustand s zu einem terminalen Zustand[1] führt. Der Test ist *wahr*, wenn das Spiel zu Ende ist, und *falsch*, wenn nicht.

6. $UTILITY(s, p)$: Die *utility*-Funktion bestimmt das Ergebnis des Spiels. Jeder Spieler p im Zustand s bekommt eine Belohnung, die ein numerischer Wert ist. Bei *Vier Gewinnt* bekommt der Gewinner $+1$ und der Verlierer -1 und bei einem Unentschieden bekommen beide Spieler jeweils $+\frac{1}{2}$

Es gibt viele verschiedene Arten von Spielen, die in über fünf verschiedene Charakteristiken klassifiziert werden können (Browne u. a., 2012, 4):

1. *Zero-Sum*: Als *zero-sum* wird ein Spiel bezeichnet, bei dem in jedem Durchlauf die aufsummierte Belohnung aller Spieler gleich ist.

2. *Information*: Gibt an, ob der Zustand eines Spiels für die Spieler teilweise (z.B. verdeckte Karten) oder komplett sichtbar ist.

3. *Determinism*: Wenn ein Spiel keinen Zufallsfaktor (z.B. in Form von Würfeln) besitzt, ist es deterministisch.

4. *Sequential*: Sequenziell ist ein Spiel dann, wenn Spieler ihre Züge nacheinander machen. Wenn die Züge gleichzeitig stattfinden ist es nicht-sequenziell.

5. *Discrete*: Als diskret werden Züge in einem Spiel dann bezeichnet, wenn sie nicht zur Echtzeit stattfinden.

Das Spiel *Vier Gewinnt* wird als deterministisches, sequentielles, zwei Spieler, *zero-sum* Spiel mit perfekter Information bezeichnet.

3.1.2 Spielbaum und Suchbaum

Der Spielbaum ist ein Baum, bei dem der initiale Zustand den Wurzelknoten, die Züge die Kanten und die Spielzustände die Knoten bilden. Die terminalen Zustände des Spiels sind die Blattknoten. Ein Spielbaum umfasst das komplette Spiel, wohingegen ein Suchbaum nur ein Teilbaum des Spielbaums beschreibt. Der Spielbaum ist häufig so groß (z.B. bei Schach), dass

[1] Zustände, die das Ende eines Spiels beschreiben

er nicht abgebildet werden kann und daher ein Suchbaum als Ersatz aufgebaut wird, um auf Grundlage dessen die Entscheidung über die nächste Aktion treffen zu können.

3.2 Multi-Armed Bandits

Ein Glückspieler hat die Wahl mehrere einarmige Banditen[2] zu bespielen, die unterschiedliche Wahrscheinlichkeiten haben Gewinne auszuzahlen. Sein Ziel ist es den Automaten zu finden, der seine Gewinne maximiert. Das Problem ist es nun die Gewinnchancen der einzelnen Automaten zu erkunden (Exploration), aber gleichzeitig auch den maximalen Gewinn (Exploitation) zu erzielen. Dieses Problem bezeichnet man als *Multi-Armed Bandit Problem* und ist ein Beispiel für ein Exploration-Exploitation Dilemma (Whitehouse, 2014, 22-23).

3.2.1 K-Armed Bandit Problem

Die einfachste Form des *Multi-Armed Bandit Problem* wird auch als *K-Armed Bandit Problem* bezeichnet (Auer u. a., 2002, 235). Das Ziel dieses Entscheidungsproblems ist es, zwischen K Aktionen (z.B. die einarmigen Banditen) die optimale Aktion so zu wählen, dass sich die Belohnung R stetig erhöht. Ein Multi-Armed Bandit kann dann als Reihe von Zufallsergebnissen $X_{i,n}$ für $1 \leq i \leq K$ und $n \geq 1$ angesehen werden, wobei i die gewählte Aktion beschreibt und n die Anzahl aller Durchläufe (Kocsis und Szepesvári, 2006, 3-4; Browne u. a., 2012, 5). Aufeinanderfolgendes Ausspielen der Aktion i ergibt die Ergebnisse $X_{i,1}, X_{i,2}, ...$, die alle auf gleiche Weise, aber unabhängig voneinander, durch eine unbekannte Funktion mit der Erwartung $\mathbb{E}[X_{i,n}] = \mu_i$ errechnet werden.

3.2.2 ϵ-Greedy

Die einfachste Methode, um die Belohnung eines *K-Armed Bandit Problems* zu maximieren, wäre es immer den Banditen zu wählen, der den höchsten Gewinn ausschüttet. Da man allerdings nicht mit Gewissheit sagen kann, welcher Bandit der beste ist, muss man die Gewinnchancen der Banditen erkunden. Um die Gewinnchancen zu erkunden und dennoch die Gewinne nahe des Maximums zu halten, wählt man fast immer die beste bekannte Aktion und variiert nur zu einer geringen Wahrscheinlichkeit ϵ. Haben wir $n \to \infty$ Durchläufe, dann stellen wir sicher, dass jede Aktion i gegen ∞ Male gewählt wird. Dadurch nähert sich der geschätzte Wert $Q_n(i)$ der Aktion i dem wahren Wert $Q^*(i)$ der Aktion an. Dabei nähert sich die Wahrscheinlichkeit die optimale Aktion zu wählen auf $1 - \epsilon$ an. Dieses Verfahren nennt man *ϵ-greedy action selection* und ist eine Lösung für das Exploration-Exploitation Dilemma. (Sutton und Barto, 1998, 155-156)

[2] Glücksspielautomat, bei dem man einen Hebel zieht, um Walzen mit Symbolen in Bewegung zu setzen, die beim Anhalten das Resultat zeigen.

3.2.3 Regret

K-Armed Bandit Problems können durch eine *policy* (Verfahrensweise) gelöst werden, die mit Hilfe aller bisher gewählten Aktionen und deren Belohnungen entscheidet, welche Aktion als nächstes gewählt wird (Kocsis und Szepesvári, 2006, 4). Diese *policy* π sollte versuchen das *regret* (Bedauern) so gering wie möglich zu halten. Das *regret* R von π nach n Durchläufen ist definiert als:

$$R_n = \mu^\star n - \mu_j \sum_{j=1}^{K} \mathbb{E}[T_j(n)],$$

wo $\mu^\star$ die bestmögliche zu erwartende Belohnung ist und $\mathbb{E}[T_j(n)]$ die zu erwartende Anzahl, die die Aktion j in den Durchläufen n gewählt wird (Browne u. a., 2012, 5). Das *regret* steht also für den Verlust, der durch die *policy*, die nicht die beste Aktion gewählt hat, verursacht wurde (Kocsis und Szepesvári, 2006, 4).

3.2.4 Upper Confidence Bound

Die *upper confidence bound* (UCB) *policy* ist eine andere Strategie, die das Problem des Exploration-Exploitation Dilemma löst. Die einfachste Variante der UCB wurde von (Auer u. a., 2002) vorgeschlagen und wird als UCB1 bezeichnet, welches ein zu erwartendes logarithmisches Wachstum des *regret* konstant über n besitzt und dabei kein Wissen über die Belohnungen haben muss, solange diese zwischen 0 und 1 begrenzt sind. UCB1 spielt die Aktion j, die

$$\text{UCB1} = \overline{X}_j + \sqrt{\frac{2 \ln n}{n_j}}$$

maximiert. Es gilt, dass $\overline{X}_j$ die durchschnittliche Belohnung der Aktion j ist, n_j die Anzahl der Male, die j gewählt wurde, und n ist die Anzahl aller Durchläufe. Der linke Teil der Formel, also die durchschnittliche Belohnung, bewirkt, dass besonders gute Aktionen gewählt werden und damit die Exploitation sichergestellt ist. Der rechte Teil hingegen, *bias term*, erhöht sich für Aktionen, die wenig gewählt wurden, stetig und stellt damit sicher, dass auch die Exploration gegeben ist. (Browne u. a., 2012, 5)

3.3 Reinforcement Learning

Dieses Unterkapitel ist ein grober Einstieg in das Thema des *Reinforcement Learning* (bestärkendes Lernen) und wird anhand der Technik des *Q-Learning* konkretisiert. *Reinforcement Learning* (RL) kann als das Problem beschrieben werden, ein Ziel mit Hilfe von Lernen durch Interaktion zu erreichen, wobei ein System, welches lernt und entscheidet, als ein Agent und alles außerhalb dieses Systems als die Umwelt bezeichnet wird (Sutton und Barto, 1998, 195). Beim RL bekommt der Agent, wenn er sich für Aktionen entscheidet und diese ausführt, Rückmeldung von der Umwelt in Form von Belohnungen und kann dadurch seine Aktionen bewerten und später Annahmen treffen (Russell und Norvig, 2009, 830).

Genauer erklärt am Beispiel von *Vier gewinnt*, kennt also ein RL-Agent zu jedem Zeitpunkt den Zustand des Spiels (das Spielbrett) und alle möglichen Aktionen in diesem Zustand (bespielbare Spalten). Wirft der Agent nun einen Stein in eine der Spalten, bekommt er eine Rückmeldung von der Umwelt in Form einer Belohnung. Durch die Rückmeldung kann der Agent nun die Aktion in diesem Zustand bewerten und auf Grundlage dieser Bewertung das nächste Mal entscheiden, welche Aktion er in diesem Zustand wählt. Der Agent hat also gelernt. Wird dieser Vorgang endlich oft wiederholt, wird der Agent irgendwann die optimale Strategie, *policy* genannt, lernen (Russell und Norvig, 2009, 830).

3.3.1 Markov Decision Process

Nach Russell und Norvig (2009, 646-647):

Ein sequenzielles Entscheidungsproblem, das in einer nicht deterministischen und voll beobachtbaren Umgebung stattfindet, positive Belohnungen besitzt und die *Markow-Annahme* für Zustandsübergänge erfüllt, kann als *Markov decision processes* (Markow-Entscheidungsproblem) bezeichnet werden. Die *Markow-Annahme* gilt, wenn die Wahrscheinlichkeit von Zustand s in den Zustand s' zu gelangen nur von s abhängt und nicht von vorangegangenen Zuständen. Ein *Markov decision processes* (MDP) wird definiert als eine Menge von Zuständen, für die gilt:

1. s_0: Der initiale Zustand.
2. $A(s)$: Alle Aktionen im Zustand s.
3. $P(s'|s, a)$: Die Wahrscheinlichkeit von Zustand s durch Aktion a in den Zustand s' zu gelangen (*transition model*).
4. $R(s, a, s')$: Die Belohnung, um von Zustand s durch Aktion a in den Zustand s' zu gelangen.

Um ein MDP zu lösen, gilt es eine *policy* π zu finden, wobei $\pi(s)$ die Aktion beschreibt, die von π für den Zustand s vorgeschlagen ist. Ziel ist es nun die *policy* π zu finden, die zu der höchsten zu erwartenden Belohnung [3] führt. Eine solche *policy* wird als optimal bezeichnet.

3.3.2 Q-Learning

Die Aufgabe des *Reinforcement Learning* (RL) ist eine optimale *policy* für ein *Markov decision processes* zu finden bzw. zu lernen (Russell und Norvig, 2009, 830). *Q-Learning* ist ein solches Verfahren des RL. Das Besondere an diesem Verfahren ist, dass es keine Informationen über das *transition model* $P(s'|s,a)$ benötigt. Es wird daher als *model-free* bezeichnet (Russell und Norvig, 2009, 843). Beim *Q-Learning* lernt der Agent den Wert von Zustand-Aktionspaaren, sogenannten *Q-Values*. Wenn Aktion a in Zustand s ausgeführt wird, um in Zustand s' zu gelangen, wird die Aktualisierungsfunktion für *Q-Values* ausgeführt und ist dabei definiert als (Russell und Norvig, 2009, 844):

$$Q(s,a) \leftarrow Q(s,a) + \alpha(R(s) + \gamma \max_{a'} Q(s',a') - Q(s,a)),$$

wobei gilt:

1. $Q(s,a)$: Der alte *Q-Value* oder falls noch nicht definiert 0.
2. $0 \leq \alpha \leq 1$: Der α-Wert gibt die Lernrate an, also wie stark neu Gelerntes gewichtet wird. Falls die Lernrate abhängig von der Anzahl der Durchläufe t sinkt, können eventuell Fehler verringert und Ergebnisse verbessert werden (Russell und Norvig, 2009, 724-725, 836-837):
   $$\alpha(t) = O(1/t)$$
   würde diese Bedingung erfüllen.
3. $R(s)$: Die Belohnung nach dem Zustand s.
4. $0 \leq \gamma \leq 1$: Der sogenannte *discount factor* gewichtet zukünftige Belohnungen. Gegen 0 sind die Belohnungen in ferner Zukunft zu vernachlässigen. Wenn der *discount factor* gegen 1 geht, zielt der Agent dagegen auf hohe Belohnungen in ferner Zukunft. Wenn die Umwelt nicht endlich ist, also keinen terminalen Zustand hat oder erreicht, muss $\gamma < 1$ gewählt werden, da sonst die Belohnungen gegen ∞ gehen würden. (Russell und Norvig, 2009, 649-650)

[3] Da ein MDP nicht deterministisch ist, kann nur eine Annahme über die Belohnung getroffen werden.

5. $\max_{a'} Q(s', a')$: Es wird aus allen *Q-Values* im Zustand s' der höchste Wert mit der Aktion a' gewählt.

3.4 Monte Carlo Method

Chaslot (2010, 4) fasst die *Monte Carlo methods* in Spielen wie folgt zusammen: Das Verfahren der *Monte Carlo methods* sucht den wahren Wert eines Zustands in einem Spiel, indem es von diesem Zustand aus zufällige, endliche Simulationen (auch *playouts* oder *rollouts* genannt) ausführt. Der Durchschnitt aller Ergebnisse der terminalen Zustände in diesen Simulationen wird gemittelt und als Bewertung des Zustandes herangezogen.

Die Funktion $E_n(P)$ ist der Mittelwert nach n Simulationen des Zustandes P und wird formal beschrieben als:

$$E_n(P) = \frac{1}{n} \sum R_i,$$

wobei R_i die Bewertung R der Simulation i ist.

3.5 Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) beschreibt eine Familie von Algorithmen, deren Ziel es ist, die vielversprechendsten Züge zu finden. Die Anfänge von MCTS Algorithmen finden sich in der Arbeit von Abramson wieder. Er zeigte, dass sich zufällige Spieldurchläufe, ähnlich wie in *Monte Carlo methods*, nutzen lassen, um Züge in Spielen zu bewerten (Abramson, 1987). Darauf hin wandte 1993 Brügmann ein Verfahren an, um ein Programm für das Brettspiel Go zu entwickeln, das mit Hife zufälliger Spieldurchläufe Züge bewertet (Brügmann, 1993).

Der Durchbruch für MCTS kam 2006, nachdem Rémi Coulom (Coulom, 2006) und L. Kocsis und Cs. Szepesvári (Kocsis und Szepesvári, 2006) unabhängig voneinander die Suche in Baumstrukturen mit Monte Carlo Simulationen kombinierten. Besonders Rémi Coulom prägte den Namen Monte Carlo Tree Search.

Kurz darauf wurden deutliche Erfolge im Bereich Computer Go mit Hilfe von MCTS Algorithmen erreicht. Im selben Jahr nutzte S. Gelly et al. das Verfahren, um das sehr erfolgreiche Computer Go Programm *MoGo* zu entwickeln (Gelly u. a., 2006). Im Jahr darauf entwickelte S. Gelly mit D. Silver das bis dato beste Computer Go Programm (Gelly und Silver, 2007).

3.5.1 Der Algorithmus

Der Monte Carlo Tree Search (MCTS) Algorithmus ist eine Form der Bestensuche[4] in einem Suchbaum basierend auf Monte-Carlo-Simulationen (Chaslot, 2010, 16). MCTS kann für jedes Spiel angewandt werden, solange es endlich ist (Chaslot u. a., 2008). Ein entscheidender Vorteil der MCTS Algorithmen gegenüber anderen Verfahren, wie zum Beispiel den Algorithmen A* (Russell und Norvig, 2009, 93 ff.) und Minimax (Russell und Norvig, 2009, 165 ff.), ist, dass der Algorithmus nur am Ende des Durchlaufs eine Bewertungsfunktion benötigt. Dadurch werden für einen nicht terminalen Zustand keine Heuristiken benötigt, die häufig sehr schwierig zu ermitteln sind.

Wie Abbildung 3.1 zeigt, besteht eine Iteration des Basis-Algorithmus aus folgenden vier Schritten (Chaslot u. a., 2008, 216):

[4] Die Bestensuche ist eine Suche in einem Graphen. Bei jeder Iteration der Suche wird der Knoten gewählt, der durch eine Bewertungsfunktion (Heuristik) als am besten gewertet wurde (Russell und Norvig, 2009, 92).

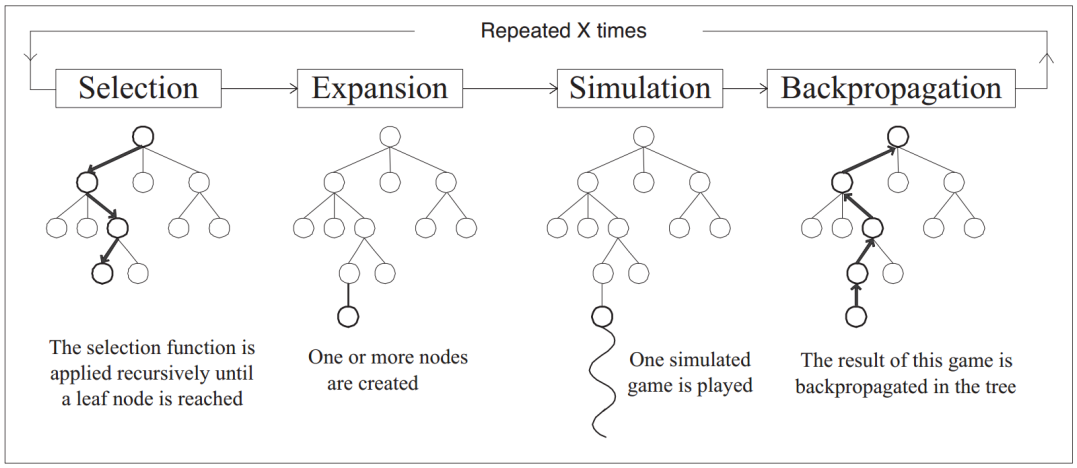

Abbildung 3.1: Skizze einer Monte Carlo Tree Search (Chaslot u. a., 2008, 216)

1. *Selection*: Ziel ist es, einen Zustand im Spielbaum zu finden, der erweiterbar ist. Erweiterbar ist ein Zustand dann, wenn er nicht terminal ist und noch nicht besuchte Züge hat. Je nach gewähltem Verfahren, *tree policy* genannt (Browne u. a., 2012, 6), werden nun Züge gewählt, bei denen besonders vielversprechende, aber auch selten besuchte Zustände erreicht werden, bis ein erweiterbarer Zustand gefunden wurde.

2. *Expansion*: Es wird ein noch nicht besuchter Zug des in Schritt eins gewählten Zustands zufällig gewählt und ausgeführt. Der Spielbaum wird nun um den neuen entstandenen Zustand erweitert.

3. *Simulation*: Der Rest des Spiels wird nun mit einem Zufallsverfahren, *default policy* genannt (Browne u. a., 2012, 6), bis zum Ende simuliert.

4. *Backpropagation*: Alle besuchten Zustände werden über das aus dem vorherigen Schritt resultierende Ergebnis benachrichtigt und die Statistiken dieser Zustände angepasst.

Die hauptsächlichen Unterschiede der verschiedenen MCTS Algorithmen ergeben sich aus den gewählten *policies* bei den Schritten der *Selection* und der *Simulation* (*tree policy* und *default policy*).

3.5.2 Upper Confidence Bound für Bäume

Eine der bekanntesten Varianten von MCTS Algorithmen ist die *Upper Confidence Bound für Bäume* (UCT). Der Erfolg von MCTS, besonders bei dem Spiel Go, ist hauptsächlich auf diesen

Algorithmus zurückzuführen (Browne u. a., 2012, 7). Der UCT Algorithmus wurde erstmals durch Kocsis und Szepesvári (2006) entwickelt und im selben Jahr durch Kocsis u. a. (2006) verbessert. Entscheidend für den Erfolg war es UCB1 als *tree policy* einzusetzen. Kocsis und Szepesvári (2006, 5) beschreiben den Algorithmus folgendermaßen: Bei der *tree policy* wird jeder besuchte Knoten als unabhängiges *Multi-Armed Bandit Problem* betrachtet. Die Kindknoten entsprechen jeweils den möglichen Aktionen. Der zu erwartende *reward* dieser Aktionen wird aus allen Pfaden, die diesen Knoten als Ursprung haben, gemittelt. Im Zustand s, der Tiefe d, wird die Aktion a gewählt, die

$$\text{UCT} = Q_t(s, a, d) + c_{N_{s,d}(t), N_{s,a,d}(t)}$$

maximiert. $Q_t(s, a, d)$ ist hier der zu erwartende Wert der Aktion a in Zustand s der Tiefe d zum Zeitpunkt t und $c_{N_{s,d}(t), N_{s,a,d}(t)}$ der *bias term*. $N_{s,d}(t)$ beschreibt die Anzahl der Durchläufe vom Zustand s mit der Tiefe d zum Zeitpunkt t und $N_{s,a,d}(t)$ ist die Anzahl, die Aktion a im Zustand s in der Tiefe d zum Zeitpunkt t gewählt wurde.

UCB wird genutzt, um den Knoten zu finden, der als nächstes erweitert wird. Da allerdings die Wahrscheinlichkeit sinkt, dass ein Knoten gewählt wird, je tiefer er im Baum ist, muss der *bias term* angepasst werden, damit die richtige Balance zwischen Exploitation und Exploration immer noch gegeben ist. Daher muss nach Kocsis und Szepesvári (2006, 5-6) der *bias term* $c_{t,s}$ folgendermaßen angepasst werden:

$$c_{t,s} = 2C_p \sqrt{\frac{\ln t}{s}},$$

wobei C_p eine Konstante ist und es gilt $C_p > 0$.

Haben mehrere Aktionen den gleichen UCT-Wert, sollte in der Regel eine zufällige Aktion gewählt werden (Kocsis u. a., 2006, 4). Sollte eine Aktion noch nie gewählt worden sein, wird der UCT-Wert als ∞ angesehen und damit sicher gestellt, dass jede Aktion berücksichtigt wird (Browne u. a., 2012, 7-8).

4 Implementation

In diesem Kapitel wird die Umsetzung des Projektes erläutert. Es wird zuerst auf das allgemeine Konzept und danach auf die Implementation eingegangen. Die Implementation umfasst das Spiel *Vier Gewinnt* und die beiden künstlichen Intelligenzen und deren Algorithmen.

4.1 Konzept

Das grundsätzliche Konzept dieses Projektes war es ein Programm zu entwickeln, in dem es möglich ist zwei künstliche Intelligenzen (KI) in dem Spiel *Vier Gewinnt* gegeneinander antreten zu lassen und die Ergebnisse auswerten zu können. Es wurde dabei zwischen drei Anwendungsfällen des Programms unterschieden:

1. Einzelspiel: Das Einzelspiel bietet die Möglichkeit ein einzelnes Spiel zu spielen, das visuell dargestellt ist. Man kann sowohl als Mensch gegen die KI spielen, als auch zwei KI gegeneinander antreten lassen.
2. Massenspiel: Bei dem Massenspiel sollen zwei KI automatisiert eine große Anzahl an Runden gegeneinander spielen können.
3. Statistiken: Die Statistiken bilden die Ergebnisse der Massenspiele und veranschaulichen diese.

4.2 Architektur

Das Projekt "connect-four" wurde komplett mit der Skriptsprache *JavaScript* umgesetzt und ist eine Webanwendung, die mit dem Framework *Node.js* (Node.js Foundation, 2016) in der Version 6.9.1 realisiert wurde. Man kann das System in drei Teile gliedern:

1. Client: Der Client ist nur für die Anzeige der Daten zuständig und besitzt kein Wissen über die Logiken des Spiels oder der KI und wird daher in dieser Arbeit vernachlässigt. Der clientseitige Code ist komplett mit *React* (Facebook Inc., 2016) in der Version 15.0.2 umgesetzt.

2. Server: Der Server liefert die Daten an den Client. Hier befindet sich die Spiellogik und die KI.

3. Datenbank: Als Datenbank wurde *LevelDB* (Ghemawat und Dean, 2016) verwendet. Hier werden die Erfahrungswerte der KI und die Statistiken der Massenspiele gespeichert. Da die Datenbank nur für das Lesen und Schreiben von Daten zuständig ist, wird hier ebenfalls nicht weiter darauf eingegangen.

4.3 Implementation des Spiels

Die Umsetzung von *Vier Gewinnt* wurde so schlank wie möglich gehalten, um keine zusätzliche Komplexität zu schaffen und es vor allem so performant wie möglich zu halten. Besonders für *Monte Carlo Tree Search* ist es wichtig, dass das Spiel sehr performant ist, da viele Simulationen des Spiels gemacht werden müssen. *Vier Gewinnt* ist in der Klasse *Game* beschrieben. Die Klasse umfasst fünf Attribute, die alle Informationen über das Spiel enthalten:

1. *grid*: Ein 2D Array aus den Zahlen der Menge $\{0; 1; 2; \}$, welches das Spielfeld beschreibt. 0 bedeutet, dass eine Zelle leer ist und 1 oder 2 besagen welcher Spieler diese Zelle belegt. Der Index des ersten Arrays benennt die X-Koordinate und der Index der inneren Arrays die Y-Koordinate des Spielfelds.

2. *isFinished*: Ein Wahrheitswert, der *wahr* ist, wenn das Spiel sich in einem terminalen Zustand befindet - anderfalls *falsch*.

3. *currentPlayer*: Eine Zahl, die entweder 1 oder 2 ist und besagt, welcher Spieler an der Reihe ist.

4. *result*: Ist mit dem Ergebnis belegt, sobald *isFinished* wahr ist. Das Ergebnis enthält den Index des Spielers, der gewonnen hat oder bei einem Unentschieden die Zeichenkette "draw".

5. *id*: Eine eindeutige ID der Instanz der Klasse.

Zudem besitzt die Klasse nach außen hin noch die beiden wichtigen Funktionen *getValidActions* und *makeAction*. Erstere gibt ein Array zurück, welches alle möglichen Züge enthält. Die zweite Funktion benötigt nun einen dieser möglichen Züge als Eingabeparameter und führt diesen aus. Das Einzige, das eine Spieler-Implementation nun ermöglichen muss, ist einen der möglichen Züge wählen zu können.

4.4 Implementation der Künstlichen Intelligenz

Die zwei verschiedenen künstlichen Intelligenzen, die für dieses Projekt implementiert wurden, sind in den Unterkapiteln aufgeführt worden. Es wurde noch zusätzlich eine simple KI entwickelt, die auf zufälliger Ebene handelt und zu Beginn kurz erläutert wird. Um Komplexität zu sparen, wurde jede dieser KI so modifiziert, dass diese immer mit dem besten Zug (siehe 2.3) beginnen.

4.4.1 Random

Diese simple künstliche Intelligenz, die jede Aktion zufällig wählt, wurde hauptsächlich als performanter Gegenspieler für die anderen KI entwickelt, damit es möglich ist schnell und neutral Parameter-Konfigurationen zu testen und die Korrektheit der Algorithmen festzustellen.

4.4.2 Q-Learning

Die Umsetzung des *Q-Learning*-Agenten wurde in zwei verschiedene Klassen aufgeteilt. Die erste Klasse *QLearning* beinhaltet das Verfahren an sich und die zweite Klasse *Experience* ist eine Hilfsklasse, die für das Schreiben und Lesen der Erfahrungswerte des Agenten aus der Datenbank zuständig ist.

Experience-Klasse

Die Klasse *Experience* besitzt die Möglichkeit über einfache *get* (lesen) und *set* (schreiben) Methoden hinaus auch noch die komplexere Funktion, den besten *Q-Value* für einen Zustand zu finden. Wenn mehrere beste *Q-Values* mit dem selben Wert für einen Zustand existieren, wird einer aus diesen per Zufall gewählt und zurückgegeben. Ein Zustand wird hier nicht durch das ursprüngliche 2D Array des Spiels (siehe 4.3) repräsentiert, sondern in eine Zeichenkette umgewandelt. Es werden dabei die Inhalte des Arrays aneinandergereiht. Um die Zeichenketten kürzer zu halten, werden die leeren Zellen einer Spalte nur durch eine und nicht mehrere Nullen dargestellt. Betrachten wir die vier unterschiedlichen Spielsituationen in Abbildungen 4.1 bis 4.4, werden diese folgendermaßen als Zeichenkette dargestellt:

1. 4.1: "0000000"
2. 4.2: "00102201000"
3. 4.3: "001212120000"
4. 4.4: "121122212211121122212211121122212211121122"

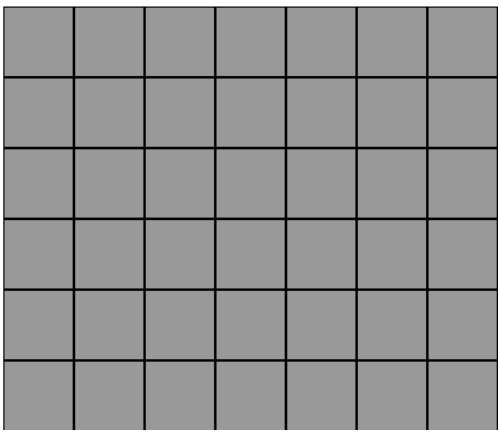

Abbildung 4.1: Leeres Spielfeld

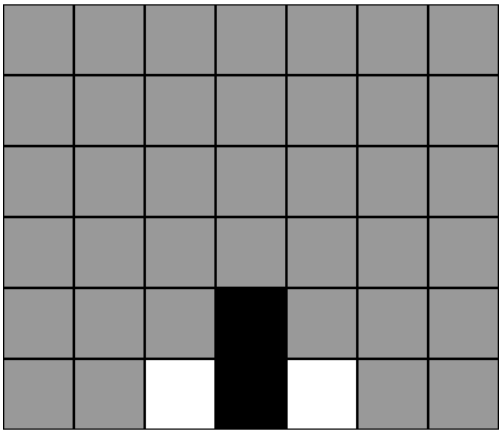

Abbildung 4.2: Beispiel-Zustand

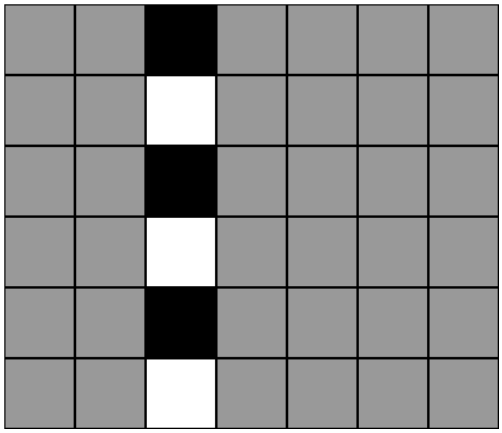

Abbildung 4.3: Spalte gefüllt

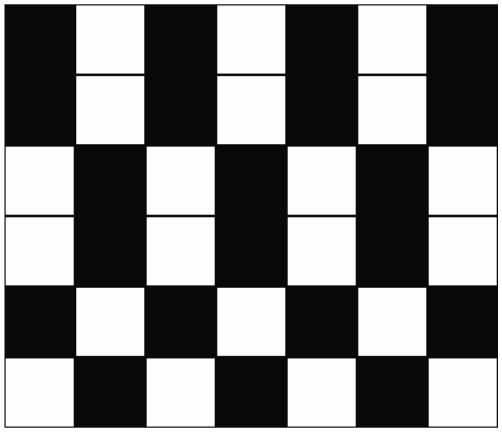

Abbildung 4.4: Komplett gefülltes Spielfeld

Gilt es nun ein Zustand-Aktionspaar abzubilden, dann wird dem Zustand noch der Index der Spalte von der Aktion angefügt. Die Aktion wird dabei von dem Zustand mit einem Punkt getrennt. Der Index startet dabei immer bei 0 für die erste Spalte. Möchte man die Aktion, bei der man den Stein in die erste Spalte wirft, für das leere Spielfeld (Abbildung 4.1) abbilden, ergibt sich also: "0000000.0".

QLearning-Klasse

Die Klasse *QLearning* ist ein Agent, der das Verfahren des *Q-Learning* aus 3.3.2 umsetzt. Es wird zusätzlich die ϵ-*greedy action selection* (siehe 3.2.2) genutzt, um das Exploration-Exploitation Dilemma zu lösen.

Bei der Initialisierung der *QLearning*-Klasse können folgende Konfigurationen gesetzt werden:

1. id: Ein Identifikator, der diese Klasse eindeutig macht. Wird auch als Kennung für die Datenbank benutzt.
2. $playerId$: Spieler-Index.
3. $rewards$: Eine Tabelle, die die Belohnungen widerspiegelt. In dieser Arbeit gilt immer:
$$\begin{cases} 100, & \text{falls Sieg} \\ -100, & \text{falls Niederlage} \\ 0, & \text{sonst} \end{cases}$$
4. $dynamicAlpha$: Wahrheitswert, der angibt, ob die Lernrate abhängig von der Anzahl der Spielrunden ist.
5. $alpha_0$: Wenn die Lernrate abhängig von der Anzahl der Spielrunden ist, gibt $alpha_0$ den initialen Wert der Lernrate an, ansonsten handelt es sich um einen konstanten α-Wert für die Aktualisierungsfunktion (siehe 3.3.2).
6. $gamma$: *discount factor* γ für die Aktualisierungsfunktion.
7. $epsilon$: Explorationswahrscheinlichkeit ϵ des Algorithmus und wurde in dieser Arbeit immer auf 0.001 gesetzt, wenn nicht anders angegeben.

Nach außen besitzt die *QLearning*-Klasse nur zwei wichtige Funktionen. Die erste Funktion *selectAction* wählt mit Hilfe einer Instanz des Spiels als Eingabeparameter eine Aktion. Hat das Spiel einen terminalen Zustand erreicht, wird die andere Funktion *endGame* mit dem Ergebnis des Spiels aufgerufen. Algorithmus 1 zeigt den Pseudocode für die Funktion *selectAction*. Wichtig ist hier zu beachten, dass $lastStateActionValue$ ein Klassenattribut der *QLearning*-Klasse ist und kein Eingabeparameter. Die Aktualisierungsfunktion des *Q-Learning* benötigt Werte und Belohnungen aus dem Folgezustand. Diese sind allerdings zum eigentlichen Zeitpunkt noch nicht bekannt und daher wird erst mit Hilfe des $lastStateActionValue$ eine Aktion später aktualisiert. Im Pseudocode von *selectAction* werden einige Pseudo-Funktionen aufgerufen, die hier kurz erläutert werden:

1. $epsilonGreedy$: Ist wahr, wenn ein Zufallswert $0 \leq random \leq 1$ kleiner ist als der anfangs gesetzte $epsilon$-Wert der Klasse *QLearning*. Hier wird das anfangs erwähnte Verfahren der ϵ-*greedy action selection* angewandt.

Algorithm 1 Pseudocode für die Funktion $selectAction$

Require: {
$state$, aktueller Zustand des Spiels.
$possibleActions$, alle möglichen Aktionen im Zustand $state$.
$lastStateActionValue$, Klassenattribut, initial $null$.
}

Ensure: $action$, (beste) Aktion aus $possibleActions$.

$state, action, value \leftarrow bestStateActionValue(state, action)$
if $lastStateActionValue$ **not** $null$ **then**
  $oldState, oldAction, oldValue \leftarrow lastStateActionValue$
  $newValue \leftarrow calculateQValue(oldValue, value, 0)$
  $setStateActionValue(oldState, oldAction, newValue)$
end if
if $epsilonGreedy()$ **then**
  $action \leftarrow$ wähle zufällige Aktion aus $possibleActions$
  $state, action, value \leftarrow getStateActionValue(state, action)$
end if
$lastStateActionValue \leftarrow state, action, value$
return $action$

2. $getStateActionValue(state, action)$: Gibt ein Zustand-Aktionspaar mit dem zugehörigen *Q-Value* zurück. Wird auf der *Experience*-Klasse aufgerufen.

3. $setStateActionValue(state, action, value)$: Persistiert ein Zustand-Aktionspaar mit dem zugehörigen *Q-Value* in der Datenbank. Wird auf der *Experience*-Klasse aufgerufen.

4. $bestStateActionValue(state, action)$: Gibt das Zustand-Aktionspaar mit dem besten *Q-Value* zurück. Wird auf der *Experience*-Klasse aufgerufen.

5. $calculateQValue(oldValue, bestValue, reward)$: Berechnet den neuen Q-Wert nach der Aktualisierungsfunktion des *Q-Learning* (siehe 3.3.2). Der Pseudocode der Funktion sieht folgender Maßen aus:

   Require: $oldValue, bestValue, reward$
   Ensure: $qValue$, neuer *Q-Value*
   $qValue = oldValue + \alpha * (reward + \gamma * bestValue - oldValue)$
   return $qValue$

Da *selectAction* den *Q-Value* erst einen Spielzug später aktualisiert, wird die Aktualisierung für den *Q-Value* des letzten Zustands nicht ausgeführt. Um dieses Problem zu lösen, gibt es die

zweite Funktion *endGame*, die immer dann aufgerufen wird, nachdem ein terminaler Zustand erreicht wurde. Algorithmus 2 zeigt den Pseudocode der Funktion, wobei $getReward(result)$ die Belohnung aus $rewards$ je nach Ergebnis zurückgibt. $setStateActionValue$ und $calculateQValue$ sind exakt dieselben Funktionen, wie bei *selectAction*. Wenn α abhängig von der Anzahl der Episoden berechnet werden soll, wurde die Berechnung

$$\alpha = \frac{\alpha_0}{\alpha_0 + t}$$

gewählt, wobei t die Anzahl an Episoden ist. Diese Berechnung erfüllt die anfangs erläuterte Bedingung $\alpha(t) = O(1/t)$ aus Kapitel 3.3.2.

Algorithm 2 Pseudocode für die Funktion $endGame$

Require: {
$result$, Ergebnis des Spiels.
$episodes$, Klassenattribut, welches die Anzahl der Spielrunden widerspiegelt, initial 0.
$dynamicAlpha$, Klassenattribut, das besagt, ob die Lernrate abhängig von den Episoden berechnet werden soll.
$lastStateActionValue$, letztes Zustand-Aktionspaar mit *Q-Value*.
}

$state, action, value \leftarrow lastStateActionValue$
$newValue \leftarrow calculateQValue(oldValue, value, getReward(result))$
$setStateActionValue(state, action, newValue)$
$episodes \leftarrow episodes + 1$
if Lernrate α ist abhängig von der Anzahl der Episoden **then**
  berechne α neu
end if
return

4.4.3 Monte Carlo Tree Search

Die künstliche Intelligenz für Monte Carlo Tree Search (MCTS) wurde strikt nach 3.5.1 implementiert und nutzt die Variante des UCT (siehe 3.5.2) als *tree policy*. Die Umsetzung wurde durch vier Klassen realisiert, die hier erst aufgelistet und nachfolgend im Detail erläutert werden:

1. *MonteCarloTreeSearch*: Die eigentliche Klasse der künstlichen Intelligenz. Dieser Agent bietet die Funktionalität die nächste auszuführende Aktion zu finden.

2. *RandomMCTS*: Hier findet sich die Umsetzung der *default policy* des MCTS. Der Algorithmus ermöglicht es ein Spiel auf zufälliger Basis zu simulieren.
3. *Node*: Die Knoten des Suchbaums vom MCTS-Algorithmus werden durch eine Instanz dieser Klasse abgebildet.
4. *Root*: Ist als Kindklasse von *Node* abgeleitet und repräsentiert den Wurzelknoten des Suchbaums.

MonteCarloTreeSearch

Um den MCTS Algorithmus zu konfigurieren, kann ein Parameter im Konstruktor angegeben werden. Dieser gibt die Anzahl an Iterationen an, die der Algorithmus durchläuft, bis er abbricht und das Ergebnis ausgibt.

Der Algorithmus wählt die nächste zu spielende Aktion über die Funktion *selectAction*, die als Eingabeparameter eine Instanz des Spiels benötigt. Dabei wird ein neuer Wurzelknoten mit Hilfe der Klasse *Root* konstruiert und die Funktion *exploreAndFind* der Klasse aufgerufen, die versucht die beste Aktion zu finden, indem ein Suchbaum aufgebaut wird.

RandomMCTS

Mit der Klasse *RandomMCTS* lässt sich ein Spiel zufällig zu Ende spielen. Wenn ein Spieler allerdings einen Zug machen kann, der zum Spielende führen würde, wird dieser sofort gewählt und das Ergebnis des Spiels zurückgegeben. So wird nicht nur unnötige Rechenzeit gespart, sondern auch Zustände ausgelassen, die gegen ein perfekten Spieler nie erreicht werden würden.

Node

Diese Klasse bildet einen Zustand des Spiels ab. Die Klasse speichert dabei die momentane Instanz des Spiels (*game*) und den Zug (*action*) der benötigt wurde, um in den momentanen Zustand des Spiels zu gelangen. Der Zug besteht aus einem Spieler- und Spaltenindex. Zudem wird dem Knoten sein Vaterknoten (*parent*) bekannt gemacht und er merkt sich alle noch nicht besuchten Züge (*unvisitedActions*), die von diesem Zustand aus möglich sind. Es gibt außerdem zwei Zähler: Einer zählt wie häufig dieser Knoten besucht wurde (*visits*), der andere zählt wie häufig er dabei zum Sieg bzw. Unentschieden geführt hat (*wins*).

Der Schritt der *Expansion* des MCTS ist durch die Funktion *expand* implementiert. Beim Expandieren eines Knoten wird einer der noch nicht besuchten Züge zufällig gewählt und aus

der Liste entfernt. Dieser Zug wird dann auf einer Kopie des Spiels ausgeführt. Mit Hilfe der neuen Instanz des Spiels und dem gewählten Zug wird dann ein neuer Kindknoten erstellt. Jeder erstellte Kindknoten(*children*) wird sich gemerkt.

Um von einem Knoten aus das Ende des Spiels zu simulieren, wird die Funktion *finishRandom* aufgerufen (*Simulation*). Hierbei wird mit Hilfe der Klasse *RandomMCTS* das Spiel bis zum Ende gespielt und mit dem Ergebnis der Simulation die Funktion *update* aufgerufen (*Backpropagation*).

Bei dem Aufruf von *update* wird *visits* inkrementiert und je nach Ergebnis eventuell auch der Sieg-Zähler. Hat der Spieler gewonnen, der den Zug in diesem Zustand ausgeführt hat, wird *wins* um 1 erhöht, bei einem Unentschieden um 0.5. Danach wird *update* mit dem gleichen Parameter auf dem *parent* aufgerufen.

Mit Hilfe der Zähler kann jeder Knoten seinen UCT-Wert über die Funktion *uctValue* errechnen, dabei gilt:

$$UCT = wins/visits + 2 * C_p * \sqrt{\frac{\ln parent.visits}{visits}},$$

wobei $C_p = \frac{1}{\sqrt{2}}$, wie von Kocsis und Szepesvári (2006, 7) vorgeschlagen, gewählt wurde.

Root

Der Wurzelknoten des Suchbaums wird durch die *Root*-Klasse repräsentiert. Die Klasse implementiert alle Funktionen und Parameter von der Vaterklasse *Node*. Sie besitzt allerdings weder eine *action* noch einen *parent*. Daher ist auch die *update*-Funktion überschrieben und inkrementiert nur die *visits* beim Aufrufen.

Die Klasse bildet den Ausgangspunkt einer MCTS. Die Suche wird dabei über die Funktion *exploreAndFind* eingeleitet, die als Eingabeparameter die maximale Anzahl an Iterationen der Suche benötigt. Bei jeder Iteration wird ein Knoten gewählt (Selection).

Dieser Knoten wird durch eine Suche in der Tiefe des Baumes gefunden, wobei immer der Kindknoten gewählt wird, der den höchsten UCT-Wert besitzt. Falls mehrere Knoten sich den höchsten Wert teilen, wird ein zufälliger gewählt. Die Suche hat ein Ende, wenn der Knoten in einem terminalen Zustand ist (Blattknoten) oder noch nicht besuchte Züge besitzt. Ist der gefundene Knoten in keinem terminalen Zustand und besitzt noch nicht besuchte Züge wird der Suchbaum durch *expand* erweitert. Die Suche findet nun entweder den Blattknoten oder den Knoten, der durch das Expandieren entstanden ist.

Algorithm 3 Pseudocode für die Funktion $exploreAndFind$

Require: {
$rootNode$, der Wurzelknoten, auf dem die Funktion aufgerufen wird.
$maxIterations$, Anzahl an Iterationen, bevor ein Zug gewählt wird.
}
Ensure: $action$, der (beste) Spielzug.
for $i = 0$ **to** $maxIterations$ **do**
$node = rootNode$
while not ($node$ ist terminal **or** $node.unvisitedActions$ ist nicht leer) **do**
$node =$ Wähle das beste Kind in $node.children$ abhängig vom UCT-Wert
end while
if not $node$ ist terminal **then**
$node = node.expand()$
end if
$node.finishRandom()$
end for
$bestChild =$ Wähle das beste Kind in $rootNode.children$ abhängig von der Siegrate
return $bestChild.action$

Nachdem ein Knoten gewählt wurde, wird mit *finishRandom* die *Simulation* ausgeführt und das Ergebnis mit *update* backpropagiert. Zuletzt wird der Zug des besten Kindknotens vom Wurzelknoten zurückgegeben. Als bester Knoten wird derjenige bezeichnet, der die höchste Siegrate $\frac{wins}{visits}$ hat.

Der Algorithmus 3 verdeutlicht das Vorgehen der Funktion *exploreAndFind* anhand eines Pseudocodes:

5 Experimente und Ergebnisse

Da es beim *Q-Learning* viele zu konfigurierende Parameter gibt und diese deutliche Auswirkungen auf die Leistung haben können, wurde versucht die beste Konfiguration gegen einen MCTS-Agenten zu finden. Daher wird zuerst beschrieben, wie die Konfiguration ermittelt wurde. Danach wird versucht die beiden Algorithmen in einem direkten Duell zu vergleichen und erklärt, warum dies nicht so einfach möglich ist. Im letzten Unterkapitel werden dann die Schlussfolgerungen dieser Arbeit präsentiert.

5.1 Q-Learning vs. Random

Um sehr schnell zu sehen, welche Auswirkungen verschiedene plausible Parameter-Konfigurationen haben können, wurden diese gegen den Zufalls-Agenten getestet. Da es zum Teil größere Schwankungen bei mehreren Durchläufen mit gleicher Konfiguration gab, wurde jede Konfiguration dreimal mit 250000 Episoden überprüft und die Ergebnisse in Form der Siegrate gemittelt. Es kamen dabei sieben verschiedene Konfigurationen zum Einsatz:

1. *Q-Learning-G1DA1000*: $dynamicAlpha = wahr, alpha_0 = 1000, gamma = 1$
2. *Q-Learning-G1DA10*: $dynamicAlpha = wahr, alpha_0 = 10, gamma = 1$
3. *Q-Learning-G1A1*: $dynamicAlpha = falsch, alpha_0 = 1, gamma = 1$
4. *Q-Learning-G1A0.5*: $dynamicAlpha = falsch, alpha_0 = 0.5, gamma = 1$
5. *Q-Learning-G1A0.1*: $dynamicAlpha = falsch, alpha_0 = 0.1, gamma = 1$
6. *Q-Learning-G0.9A1*: $dynamicAlpha = falsch, alpha_0 = 1, gamma = 0.9$
7. *Q-Learning-G0.9A0.1*: $dynamicAlpha = falsch, alpha_0 = 0.1, gamma = 0.9$

Die Abbildung 5.1 zeigt die Ergebnisse dieser Tests. Deutlich zu erkennen ist, dass jede der Kurven sehr ähnlich zu einander steigen. Die gewählten Konfigurationen weichen in ihren

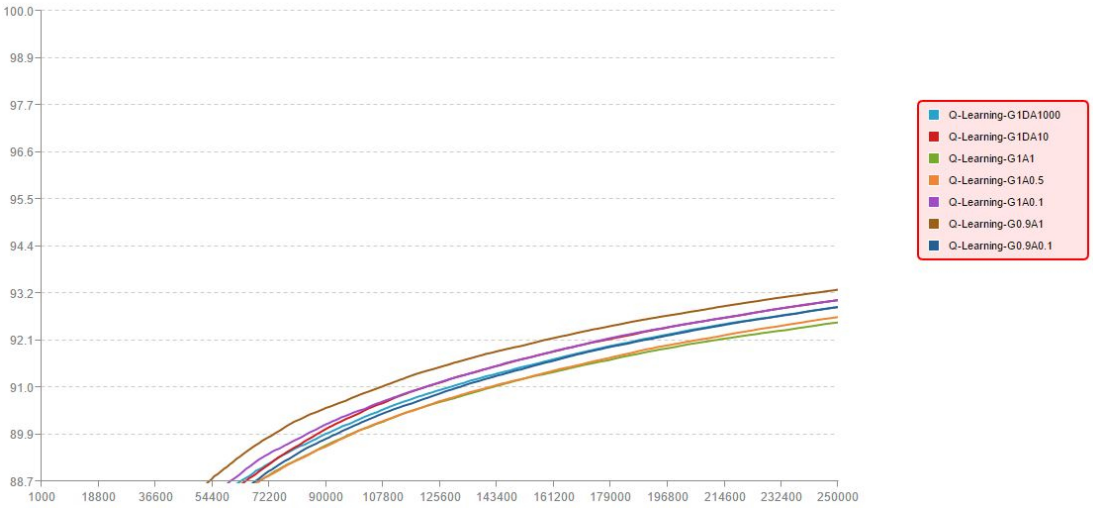

Abbildung 5.1: Verschiedene *Q-Learning*-Agenten gegen einen Zufalls-Agenten
Die X-Achse gibt die Anzahl der Episoden an und die Y-Achse die Siegrate in Prozent.

Endergebnissen maximal um 0.78% ab. Den Höchstwert von 93.32% nach 250000 Episoden erreicht der Agent *Q-Learning-G0.9A1*.

Da die Leistung der verschiedenen Konfigurationen sich kaum unterscheiden und - wie anfangs erwähnt - jeder der einzelnen Durchläufe vor dem Mitteln geschwankt hat, kann auf Grundlage dieser Test nicht wirklich eine genaue Aussage getroffen werden, welche die beste Konfiguration ist. Zudem ist auch nicht gegeben, dass eine solche Konfiguration gegen den MCTS-Agenten oder einen Menschen genauso abschneiden würde. Daher wurden zusätzlich noch verschiedene Konfigurationen gegen einen MCTS-Agenten getestet. Hier ist das große Problem, dass der MCTS-Agent deutlich langsamer als ein *Q-Learning*-Agent ist und daher Tests extrem lange dauern können.

5.2 MCTS vs. Random

Gesucht wurde nun eine Konfiguration, die die Rechenzeit bei einem MCTS-Agenten so gering wie möglich hält, sodass aber gleichzeitig noch eine gute Leistung erzielt wird. Daher wurde versucht herauszufinden, welche Anzahl an Durchläufen die beste ist, damit diese Bedingungen erfüllt sind. Dabei hat sich ergeben, dass ein Agent mit 20 maximalen Spielen pro Zug noch performant genug ist. Gleichzeitig hatte er bei drei Testdurchläufen mit 50000 Episoden gegen den Zufallsagenten eine gemittelte Siegrate von 99.98%. Daher werden 20 maximale Durchläufe für einen schwachen MCTS-Agent als ausreichend erachtet, um für die Parameter-Test geeignet zu sein.

5.3 Q-Learning vs. schwachen MCTS

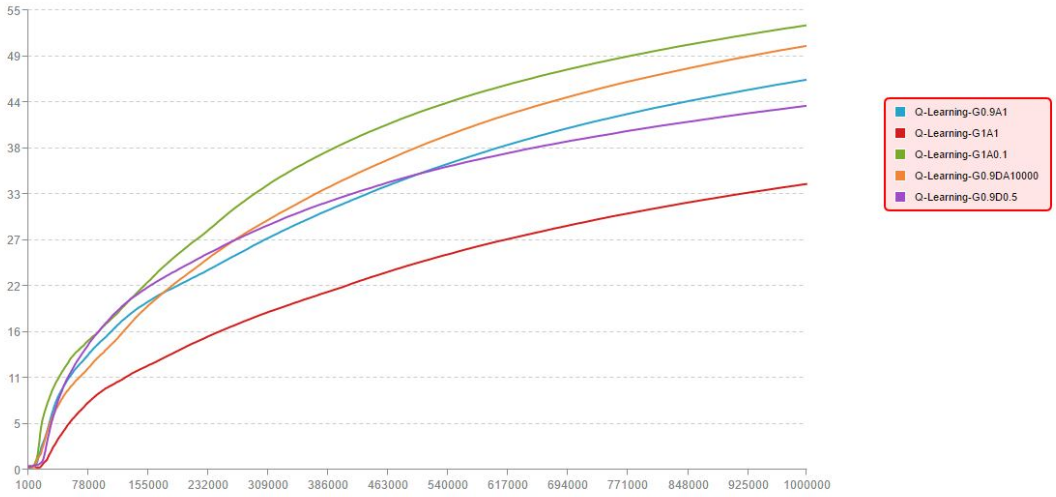

Abbildung 5.2: Verschiedene *Q-Learning*-Agenten gegen einen schwachen MCTS-Agenten
Die X-Achse gibt die Anzahl der Episoden an und die Y-Achse die Siegrate in Prozent.

Wie im vorherigen Unterkapitel gezeigt wurde, kann selbst ein schwacher MCTS-Agent gute Leistungen gegen den Zufalls-Agenten erzielen. Daher wurde ein solcher Agent nun genutzt, um die verschiedenen Parameter des *Q-Learning* zu erproben. Hierbei wurde jede Konfiguration mit 250000 Episoden gegen den schwachen MCTS-Agenten überprüft. Es kamen dabei folgende Agenten-Konfigurationen zum Einsatz:

1. *Q-Learning-G0.9A1*: $dynamicAlpha = falsch, alpha_0 = 1, gamma = 0.9$
2. *Q-Learning-G1A1*: $dynamicAlpha = falsch, alpha_0 = 1, gamma = 1$
3. *Q-Learning-G1A0.1*: $dynamicAlpha = falsch, alpha_0 = 0.1, gamma = 1$
4. *Q-Learning-G0.9DA10000*: $dynamicAlpha = wahr, alpha_0 = 10000, gamma = 0.9$
5. *Q-Learning-G0.9A0.5*: $dynamicAlpha = falsch, alpha_0 = 0.5, gamma = 0.9$

Die Ergebnisse wurden in der Abbildung 5.2 festgehalten. In der Abbildung kann man sofort erkennen, dass der *Q-Learning*-Agent mit $dynamicAlpha = falsch$, $alpha = 0.1$ und $gamma = 1$ die beste Leistung erzielt hat. Allerdings liegt der *Q-Learning*-Agent mit $dynamicAlpha = wahr$, $alpha = 10000$ und $gamma = 0.9$ nur knapp hinter dem ersten

und scheint sich immer weiter an ihn anzunähern. Dennoch wurde in dieser Arbeit die erste Konfiguration gewählt, denn es sei an dieser Stelle angemerkt, dass es nicht zwingend notwendig ist, die beste Konfiguration zu finden. Sicherlich könnten gerade im späteren Verlauf andere Konfigurationen bessere Ergebnisse erzielen. Da es aber nicht das Ziel dieser Arbeit ist, die bestmögliche KI zu entwickeln, sondern nur die Verfahren zu vergleichen, wird diese Konfiguration als ausreichend erachtet.

5.4 Q-Learning vs. MCTS

Die Stärke eines MCTS-Agent ist besonders abhängig von der Anzahl der Durchläufe, die ihm zur Verfügung stehen. Die wohl wichtigste Eigenschaft des *Q-Learning*-Agenten hingegen ist die Tatsache, dass er über die Zeit lernt und sich verbessert. Daher kommt man bei dem Vergleich der beiden Algorithmen sehr schnell in ein Dilemma. Erhöht man die maximale Anzahl an Durchläufen des MCTS-Agenten, benötigt er immer mehr Rechenzeit für die einzelnen Züge. Dadurch muss der *Q-Learning*-Agent deutlich länger auf seinen Zug warten und benötigt dadurch auch immer mehr Zeit zum Lernen. Deshalb wurde in dieser Arbeit versucht zu zeigen, wie schnell der *Q-Learning*-Agent lernt, wenn die maximale Anzahl der Durchläufe eines MCTS-Agenten erhöht wird. Daher ist ein *Q-Learning*-Agent mit der Konfiguration $dynamicAlpha = falsch$, $alpha = 0.1$ und $gamma = 1$ (siehe 5.3) gegen drei verschieden starke MCTS in jeweils 250000 Episoden angetreten.

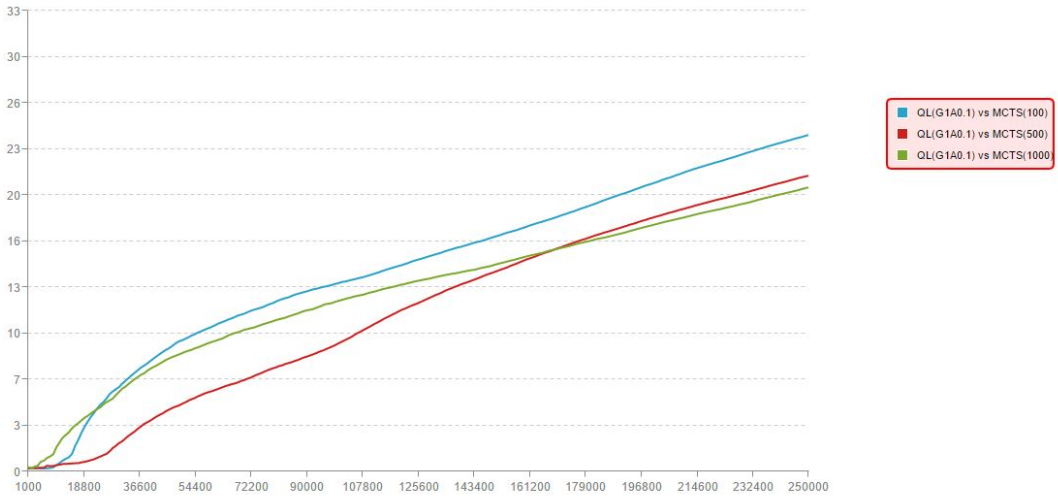

Abbildung 5.3: Ein *Q-Learning*-Agent gegen unterschiedlich starke MCTS-Agenten
Die X-Achse gibt die Anzahl der Episoden an und die Y-Achse die Siegrate in Prozent.

Die Ergebnisse dieser Spiele werden in Abbildung 5.3 gezeigt. Der *Q-Learning*-Agent verbessert sich in den drei Beispielen stetig. Sehr wahrscheinlich ist es, dass er irgendwann den MCTS-Agent in den meisten Fällen schlagen würde, wenn die Episoden gegen ∞ gehen. Das lässt sich damit begründen, dass *Q-Learning* darauf ausgelegt ist irgendwann die beste *policy* zu finden (siehe 3.3.2) und ein Spieler, der das Spiel beginnt, theoretisch das Spiel immer gewinnen kann (siehe 2.3), wenn er denn perfekt spielt. Da der Agent allerdings eine

Explorationswahrscheinlichkeit besitzt, wird verhindert, dass der Agent irgendwann perfekt spielt.

Auffällig ist auch in Abbildung 5.3, dass der Agent, der gegen den MCTS-Agenten mit 500 maximalen Spielen pro Zug antritt, anfangs länger als die anderen eine sehr geringe Steigerung in der Siegrate hat. Dieses Verhalten konnte auch bei anderen Beispielen festgestellt werden und kann damit begründet werden, dass es durch das zufällige Agieren zu Beginn immer unterschiedlich lange dauert, bis der *Q-Learning*-Agent eine Strategie gegen den MCTS-Agenten gelernt hat.

Würde man den MCTS immer mehr maximale Spiele zur Verfügung stellen, würde der *Q-Learning*-Agent wahrscheinlich immer länger brauchen, um die Strategie des MCTS-Agenten zu lernen. Allerdings ist es nicht möglich die Anzahl der maximalen Spiele unendlich zu erweitern, denn schnell wird der Arbeitsspeicher nicht mehr ausreichen, um den Suchbaum zu erweitern.

Daher ist abschließend zu sagen, dass der *Q-Learning*-Agent den MCTS-Agent immer schlagen würde, wenn er denn lange genug gegen diesen antreten würde. Dennoch bietet der MCTS-Agent eine gute Herausforderung und zeigt sehr gute Ergebnisse, ohne dabei dauerhaft gelerntes zu speichern.

5.5 Ergebnisse

Wie schon im Unterkapitel zuvor erwähnt, gestaltet sich der Vergleich der Algorithmen anhand der Leistung schwierig. Die Stärken und Schwächen hängen bei beiden Algorithmen zum größten Teil von den Anforderungen an einen Agenten ab. Dennoch kann gut verglichen werden, welcher Aufwand betrieben werden muss, um einen guten Agenten mit den jeweiligen Verfahren zu erschaffen.

Die beiden Algorithmen haben den Vorteil gemeinsam, dass sie nicht zwingend spezielles Wissen über ihre Umwelt kennen müssen. *Q-Learning* benötigt nur Zustandsbeschreibungen und Aktionen, wohingegen der MCTS-Agent noch zusätzlich die Möglichkeit haben muss Simulationen in der Umwelt durchzuführen. Beide Agenten besitzen keinerlei Wissen über ihre Umwelt und benötigen es auch nicht, um das gegebene Problem zu lösen. Dennoch können beide Algorithmen davon profitieren, wenn sie Wissen über ihre Umwelt ausnutzen würden. Für die *default policy* des MCTS-Agenten wird zum Beispiel Wissen über das Spiel genutzt, damit die Simulation immer beendet wird, wenn ein terminaler Zustand erreicht werden kann, wodurch die Rechenzeit verkürzt und das Ergebnis genauer wird. Denn es werden keine Zustände mehr in Betracht gezogen, die ein perfekter Gegenspieler nie erreichen würde. Der *Q-Learning*-Agent könnte natürlich auch noch verbessert werden, indem immer sicher eine eigene Dreierreihe vervollständigt oder eine gegnerische verbaut wird, wenn es denn möglich ist.

Da beide Verfahren kein spezielles Wissen über ihre Umwelt benötigen, können beiden Algorithmen relativ simpel umgesetzt werden. Der MCTS-Agent benötigt allerdings keine Datenbank und hat zudem deutlich weniger Parameter zum konfigurieren als der *Q-Learning* Agent. Der Aufwand einen Agenten mit MCTS zum implementieren ist daher ein wenig geringer als bei einem Agenten *Q-Learning*. Die fehlende Datenbank beim Verfahren des MCTS erschwert allerdings das *Debugging*. Mit einer Datenbank kann man schnell Werte von Zuständen überprüfen, wohingegen eine Anwendung, die alles zur Laufzeit generiert, am Ende alles wieder verwirft.

Die offensichtlichsten Vorteile der beiden Algorithmen sind gleichzeitig ihre größten Nachteile und sehr abhängig von den Anforderungen, die man an eine künstliche Intelligenz hat. Der *Q-Learning*-Agent speichert Gelerntes und kann daher sehr schnell die Lösung für eine Situation liefern, ohne zur Laufzeit viel zu berechnen. Allerdings dauert das Lernen im Vorfeld oft sehr lange und muss eventuell häufig wiederholt werden. Je nach Anwendungsfall kann es auch sein, dass das Problem zu viele Zustände besitzt und es nicht möglich ist alle Zustände in einer Datenbank zu speichern. Können nicht alle Zustände berücksichtigt werden, weil

die Anzahl so groß ist, kann der Zustandsraum verkleinert werden. Damit der Zustandsraum verkleinert werden kann, müssen Zustände allerdings zusammengefasst werden, was zum einen häufig nicht einfach ist und zum anderen bedeuten kann, dass die Leistung des Agenten abnehmen kann, da Informationen verloren gehen könnten.

Bei dem Verfahren des MCTS ist der offensichtliche Nachteil natürlich, dass die Berechnung zur Laufzeit statt findet. Soll zum Beispiel eine KI Auffahrunfälle verhindern, muss sie innerhalb kürzester Zeit reagieren können und hat eventuell nicht genügend Zeit, um die beste Lösung zur Laufzeit zu finden. Gilt es allerdings eine KI zu entwickeln, die Schach meistern soll, ist es kein Problem, wenn sie mehrere Sekunden benötigt, um den nächsten Zug zu planen. Die Berechnung zur Laufzeit bringt auch große Flexibilität mit. Würde sich das Spielfeld bei Vier Gewinnt vergrößern oder verkleinern, könnte der MCTS-Agent ohne Anpassungen machen zu müssen immer noch genauso gut eine Lösung finden. Diese Flexibilität besitzt ein *Q-Learning*-Agent nur, wenn die Zustandsbeschreibung so angepasst wurde, dass eine Veränderung in der Spielfeldgröße berücksichtigt wird. Ansonsten müsste der Agent von vorne beginnen und alles neu lernen.

Ist zudem noch der Zustandsraum schlichtweg zu groß und es gibt auch keine Möglichkeit ihn sinnvoll zu verkleinern, zeigt sich die wahre Stärke des MCTS-Agenten. Denn das Verfahren muss nicht den kompletten Zustandsraum bzw. Spielbaum aufbauen, sondern baut vielmehr einen Suchbaum in vielversprechenden Zweigen auf. Man kann den Agenten zu jeder Zeit stoppen und das bis dahin beste Ergebnis zurückgeben. Es bedeutet zwar nicht, dass es das bestmögliche Ergebnis ist, aber wahrscheinlich ein gutes. Dadurch hat man die Möglichkeit Lösungen für Probleme zu finden, die vorher zu komplex waren. Das Ergebnis einer MCTS hängt daher von der Anzahl der Iterationen ab und damit von der Größe des Suchbaums. Allerdings kann ein Suchbaum nicht unendlich groß werden, da der Arbeitsspeicher in einem Computer nicht unendlich ist.

Abschließend lässt sich sagen, dass beide Algorithmen gut zu implementieren sind. Wichtig ist es den jeweiligen Anwendungsfall genau zu betrachten und sich den Anforderungen an eine KI gewiss zu werden. Denn das mögliche Verfahren sollte abhängig von den Anforderungen gewählt werden.

6 Fazit

Dieses Kapitel bildet den Abschluss der Arbeit und fasst die Ergebnisse der Arbeit noch einmal kurz zusammen. Danach wird noch ein Ausblick gegeben, wo bei zukünftigen Arbeiten angesetzt werden kann und wie dieses Projekt eventuell noch verbessert werden könnte.

6.1 Zusammenfassung

In dieser Arbeit wurde *Monte Carlo Tree Search* (MCTS) mit der bewährten Technik des *Q-Learning* aus dem Bereich des *Reinforcement Learnings* verglichen. Mit diesen beiden Verfahren wurden zwei künstliche Intelligenzen entwickelt. Diese beiden sind dann in dem Spiel *Vier Gewinnt* gegeneinander angetreten und die Ergebnisse wurden in dieser Arbeit ausgewertet.

Beide Verfahren benötigen kein Wissen über ihre Umwelt und können daher nicht nur einfach umgesetzt werden, sondern auch mit keinen bis wenig Änderungen in eine andere Umwelt übernommen werden.

Ist der Zustandsraum zu groß, um erfolgreich erlernt zu werden oder soll der Agent sehr flexibel auf Änderungen der Umwelt reagieren können, hat MCTS gezeigt, dass es das bessere Verfahren ist. Die MCTS nutzt eine Suche im Spielbaum, um einen vielversprechenden Ast zu finden und bewertet dabei die Zustände mit Hilfe von Zufallsexperimenten. Da der aufgespannte Suchbaum zu jeder Zeit ein Ergebnis kennt, kann der Algorithmus zu jeder Zeit abbrechen und damit auch Ergebnisse bei riesigen Zustandsräumen liefern.

Q-Learning hingegen lernt im Vorfeld über sehr viele Versuche den besten Lösungsweg und zeigt damit generell bessere Ergebnisse, wenn der Zustandsraum überschaubar ist. Besonders wenn es das Ziel ist, den bestmöglichen Lösungsweg zu finden oder auch wenn eine Aktion in kürzester Zeit gewählt werden muss, ist *Q-Learning* die bessere Alternative.

Der Vergleich anhand des Spiels *Vier Gewinnt* hat besonders gezeigt, dass es nicht einfach ist die Leistung der beiden Algorithmen gegenüber zu stellen. Denn erhöht man die Stärke des MCTS-Agenten, dann erhöht sich auch die Lernzeit des *Q-Learning*-Agenten und man gerät sehr schnell in ein Dilemma. Es kann dann extrem lange dauern die beiden Agenten gegeneinander antreten zu lassen. Dennoch ist klar geworden, dass beide Algorithmen ihre

Stärken und Schwächen besitzen und je nach Anwendungsfall die richtige Wahl darstellen können.

6.2 Ausblick

Der Vergleich zwischen *Q-Learning* und Monte Carlo Tree Search stellte sich durch die Tatsache, dass Monte Carlo Tree Search sehr lange Rechenzeiten besitzen kann, als sehr schwierig heraus. Man könnte daher versuchen den MCTS-Algorithmus zu parallelisieren, um das Lernen eines Agenten deutlich zu beschleunigen. In der Phase der *Selection*, *Expansion* und *Simulation* ist dies sicherlich gut umzusetzen, allerdings wird die Parallelisierung der Phase der *Backpropagation* eine besondere Herausforderung darstellen, da die Werte der Knoten in Abhängigkeit zueinander berechnet werden (für eine Übersicht der Phasen der MCTS siehe 3.1). Diese Verbesserung hätte äußerst großen Wert, weil jede Variante des MCTS unabhängig von seiner Umwelt davon profitieren könnte.

Interessant wäre es natürlich MCTS mit anderen Verfahren aus dem Bereich der künstlichen Intelligenz zu vergleichen. Besonders interessant wären hier andere Suchverfahren, wie die Alpha-Beta-Suche. Auch der Vergleich der beiden Algorithmen anhand eines noch komplexeres Spiels könnte wissenswerte Erkenntnisse hervorbringen und die Stärken des MCTS noch mehr hervorheben.

Spannend wäre es auch zu versuchen, die beiden Verfahren miteinander zu kombinieren, um eventuell Stärken der beiden nutzen zu können. Denkbar wäre es Teile des Suchbaums in kritischen Situationen ähnlich wie beim RL zu speichern, um Gelerntes nicht komplett zu verwerfen und über die Zeit zu verbessern. Man könnte vielleicht den *Q-Learning*-Agenten verbessern, indem man ihn nicht komplett zufällig handeln lässt, wenn er sich in unbekannten Zuständen befindet, sondern mit Hilfe eines MCTS-Verfahrens.

Literaturverzeichnis

[Abramson 1987] Abramson, Bruce: The Expected-Outcome Model of Two-Player Games / Columbia University. 1987. – Forschungsbericht

[Allis 1988] Allis, Louis V.: *A knowledge-based approach of connect-four.* Citeseer, 1988

[Auer u. a. 2002] Auer, Peter ; Cesa-Bianchi, Nicolo ; Fischer, Paul: Finite-time analysis of the multiarmed bandit problem. In: *Machine learning* 47 (2002), Nr. 2-3, S. 235–256

[BBC News 2016] BBC News: *Artificial intelligence: Google's AlphaGo beats Go master Lee Se-dol.* 2016. – URL http://www.bbc.com/news/technology-35785875. – Zugriffsdatum: 2016-11-30

[Browne u. a. 2012] Browne, Cameron B. ; Powley, Edward ; Whitehouse, Daniel ; Lucas, Simon M. ; Cowling, Peter I. ; Rohlfshagen, Philipp ; Tavener, Stephen ; Perez, Diego ; Samothrakis, Spyridon ; Colton, Simon: A survey of monte carlo tree search methods. In: *IEEE Transactions on Computational Intelligence and AI in Games* 4 (2012), Nr. 1, S. 1–43

[Brügmann 1993] Brügmann, Bernd: Monte carlo go / Citeseer. 1993. – Forschungsbericht

[Chaslot 2010] Chaslot, Guillaume: Monte-carlo tree search. In: *Maastricht: Universiteit Maastricht* (2010)

[Chaslot u. a. 2008] Chaslot, Guillaume ; Bakkes, Sander ; Szita, Istvan ; Spronck, Pieter: Monte-Carlo Tree Search: A New Framework for Game AI. In: *AIIDE*, 2008

[Chinchalkar 1996] Chinchalkar, Shirish: An upper bound for the number of reachable positions. In: *ICCA JOURNAL* 19 (1996), Nr. 3, S. 181–183

[Coulom 2006] Coulom, Rémi: Efficient selectivity and backup operators in Monte-Carlo tree search. In: *International Conference on Computers and Games* Springer (Veranst.), 2006, S. 72–83

[Edelkamp und Kissmann 2008] Edelkamp, Stefan ; Kissmann, Peter: Symbolic classification of general two-player games. In: *Annual Conference on Artificial Intelligence* Springer (Veranst.), 2008, S. 185–192

[Facebook Inc. 2016] Facebook Inc.: *A JavaScript library for building user interfaces - React.* 2016. – URL https://facebook.github.io/react/. – Zugriffsdatum: 2016-11-28

[Gelly und Silver 2007] Gelly, Sylvain ; Silver, David: Combining online and offline knowledge in UCT. In: *Proceedings of the 24th international conference on Machine learning* ACM (Veranst.), 2007, S. 273–280

[Gelly u. a. 2006] Gelly, Sylvain ; Wang, Yizao ; Teytaud, Olivier ; Patterns, Modification U. ; Tao, Projet: Modification of UCT with patterns in Monte-Carlo Go. (2006)

[Ghemawat und Dean 2016] Ghemawat, Sanjay ; Dean, Jeff: *LevelDB.* 2016. – URL https://github.com/google/leveldb. – Zugriffsdatum: 2016-11-28

[Kocsis und Szepesvári 2006] Kocsis, Levente ; Szepesvári, Csaba: Bandit based monte-carlo planning. In: *European conference on machine learning* Springer (Veranst.), 2006, S. 282–293

[Kocsis u. a. 2006] Kocsis, Levente ; Szepesvári, Csaba ; Willemson, Jan: Improved monte-carlo search. In: *Univ. Tartu, Estonia, Tech. Rep* 1 (2006)

[Levy 2005] Levy, Arbiter D.: *8:4 final score for the machines ? what next?* 2005. – URL http://en.chessbase.com/post/8-4-final-score-for-the-machines-what-next-. – Zugriffsdatum: 2016-11-30

[Node.js Foundation 2016] Node.js Foundation: *Node.js.* 2016. – URL https://nodejs.org/. – Zugriffsdatum: 2016-11-28

[Russell und Norvig 2009] Russell, Stuart ; Norvig, Peter: *Artificial Intelligence: A Modern Approach.* 3rd. Upper Saddle River, NJ, USA : Prentice Hall Press, 2009. – ISBN 0136042597, 9780136042594

[Sutton und Barto 1998] Sutton, Richard S. ; Barto, Andrew G.: *Reinforcement learning: An introduction.* MIT press Cambridge, 1998

[Whitehouse 2014] Whitehouse, Daniel: *Monte Carlo Tree Search for games with Hidden Information and Uncertainty*, University of York, Dissertation, 2014

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 15. Dezember 2016 Konstantin Böhm