Untersuchung der Lernfähigkeit verschiedener Verfahren am Beispiel von Computerspielen

Abschlussarbeit
zur Erlangung des akademischen Grades
Bachelor of Science (B.Sc.)

Thilo Stegemann
s0539757
Angewandte Informatik

19. März 2017

Hochschule für Technik
und Wirtschaft Berlin
University of Applied Sciences

Erstprüfer: Prof. Dr. Burkhard Messer
Zweitprüferin: Prof. Dr. Adrianna Alexander

Abstrakt

Die vorliegende Bachelorarbeit gibt einen Überblick über Theorien des verstärkenden Lernens. Das TD-Q-Lernen ist ein verstärkendes Lernverfahren. In dieser Arbeit wird das TD-Q-Lernen, mit einer tabellarischen Repräsentation der Q-Funktion, erläutert, implementiert, getestet und beurteilt. Ein Agent der das TD-Q-Lernen anwendet, wird Strategien für die Strategiespiele Tic Tac Toe und Reversi lernen. Ein Zufallsagent und ein vorausschauender Heuristik Agent werden ebenfalls implementiert. Die Implementierung des vorausschauenden Heuristik Agenten wird eine Kombination aus einer Iterativen-Alpha-Beta-Suche und einer Heuristik.

In Testspielen wird der selbstlernende TD-Q-Agent gegen den Zufallsagenten und den vorausschauenden Heuristik Agenten antreten. Die Auswertung der Testergebnisse gibt Aufschluss über die Leistungsfähigkeit und die Grenzen des TD-Q-Lernens. Der Heuristik Agent wird in den Testspielen den TD-Q-Agenten eindeutig besiegen. Das bestätigt, warum in der Praxis überwiegend von Menschen optimierte Heuristiken und keine Lernverfahren für das Spielen von Strategiespielturnieren eingesetzte werden. Erkenntnis dieser Arbeit u.a. ist: Die tabellarische Darstellung der Q-Funktion für das TD-Q-Lernen ist ausschließlich erfolgreich für sehr kleine Zustands- und Aktionsräume anwendbar. Für größer dimensionierte Zustands- und Aktionsräume ist die parametrisierte Funktionsdarstellung der Q-Funktion erforderlich.

Abbildungsverzeichnis

Inhaltsverzeichnis

Kapitel 1

Projektvision

Viele Menschen spielen gerne Strategiespiele gegen andere Menschen oder gegen einen Computer.
Sie veranstalten große Meisterschaften in Schach und Poker. "... Schach - zumindest in der Form des Turnierschachs - ist heute unbestreitbar als Sport anzusehen ... [Wey77]" Schach ist demnach nicht nur ein Spiel, sondern auch ein anerkannter Turniersport.

Der Reiz eines Strategiespiels ist vermutlich die Entwicklung und Verbesserung der Strategie. Der Mensch lernt seine Strategien durch ständiges trainieren, verlieren, siegen, analysieren und anpassen. Er kann seine Strategie auch aus Büchern oder von einem Lehrer lernen. Eine Strategie, die sich sehr oft in der Praxis bewährt hat und viele wichtige Aspekte und Spielregeln beachtet, wird die Gewinnchancen eines Spielers verbessern.

"Zum ersten mal hat der seit zehn Jahren amtierende Schachweltmeister Garri Kasparow, den viele für den stärksten Spieler aller Zeiten halten, eine normale Turnierpartie gegen einen Schachcomputer verloren. In Philadelphia musste der Champion in der ersten von sechs Partien eines Zweikampfs gegen das auf einem IBM-Großrechner laufende Schachprogramm Deep Blue nach 37 Zügen die Waffen strecken. [Nea96]"

Dementsprechend kündigt sich eine Veränderung in den Turnieren und Meisterschaften der Strategiespiele an. Immer mehr menschliche Meister der Strategien werden von Computern besiegt.

Wir wollen daher folgende Fragen in dieser Arbeit behandeln:
Wie spielt ein Computer Strategiespiele oder wie entwickelt er Strategien? Lernen die Computer ihre Strategien oder werden ihnen explizit Strategien vorgegeben? Werden lernende Computerprogramme, in nächster Zeit, Turniere und Meisterschaften gewinnen?

1.1 Zielsetzung

Das Ziel der Arbeit ist es, den bereits existierenden Lernalgorithmus des TD-Q-Lernens zu implementieren und dessen Leistungsfähigkeit und Grenzen zu untersuchen. Das daraus resultierende Lernverfahren, soll eigenständig und automatisch Strategien für zwei Computerspiele erlernen. Wir bezeichnen die Implementierung des TD-Q-Lernens fortan als TD-Q-Agenten.

Der TD-Q-Agent soll Strategien für die selbst programmierten Computerspiele Tic Tac Toe und Reversi erlernen. Wir werden die Strategiespiele Tic Tac Toe mit 9 Spielfeldern, Tic Tac Toe mit 16 Spielfeldern und Reversi mit 64 Spielfeldern eigenständig implementieren.

Wir beurteilen die Leistungsfähigkeit des implementierten TD-Q-Agenten, anhand von Testspielen gegen andere Agenten mit unterschiedlichen Strategien. Wir werden dafür innerhalb dieser Arbeit einen Zufallsagenten und einen vorausschauenden Heuristik Agent entwickeln und implementieren.

1.2 Quantifizierung der Ziele

Wir unterteilen das Lernen des TD-Q-Agenten in drei Phasen. In der ersten Phase lernt der TD-Q-Agent in 100 Trainingsspielen eine Strategie. Trainingsspiel bedeutet: Der TD-Q-Agent spielt und lernt gegen sich selbst. Wir erhören die Anzahl der Trainingsspiele in der zweiten Phase auf 1.000 und in der dritten Phase auf 10.000 Trainingsspiele.

Für jede Lernphase entsteht eine eigene Strategie. Insgesamt lernt der TD-Q-Agent demnach 9 Strategien. 3 Strategien für das 9 Spielfelder Tic Tac Toe, 3 Strategien für das 16 Spielfelder Tic Tac Toe und 3 Strategien für das 64 Spielfelder Reversi.

Nach Abschluss jeder Lernphase wird der TD-Q-Agent mit der gelernten Strategie, in genau 100 Testspielen, gegen den vorausschauenden Heuristik Agenten und den Zufallsagenten spielen. Der Heuristik Agent wird ebenfalls in 100 Testspielen gegen den Zufallsagenten spielen.

Wir differenzieren die Testspiele außerdem nach dem jeweils beginnenden Agenten, d.h. pro gelernter Strategie, beginnt 100 mal die gelernte Strategie und 100 mal der Gegenspieler. Insgesamt werden 200 Testspiele pro gelernter Strategie durchgeführt.

1.3 Aufbau des Projekts

Zuerst werden die Spielregeln der beiden ausgesuchten Computerspiele definiert. Die Computerspiele und entsprechende Tests werden programmiert. Die Programmierung der Computerspiele und der Tests wird nicht explizit aufgeführt, kann jedoch der beiliegenden Software-CD entnommen werden.

In den nächsten beiden Schritten werden die Grundlagen für die Implementierung des Heuristik Agenten und des TD-Q-Agenten ermittelt und erstellt.

Für die Erstellung der Agenten werden danach die Anforderungen definiert und anschließend modelliert.

Die definierten Anforderungen werden Implementiert. Der Algorithmus der das vorausschauen des Heuristik Agenten realisiert und der Algorithmus der das eigenständige Lernen des TD-Q-Agenten realisiert werden ausführlich in dieser Arbeit beschrieben. Die kompletten implementierten Anforderungen und die benötigten Algorithmen für die Agenten sind auf der Software-CD vorhanden.

Anschließend beginnen die geplanten Trainingsphasen und Testspiele.

Der letzte Schritt ist die Auswertung der Testergebnisse und die Beurteilung der Leistungsfähigkeit der Agenten.

1.4 Ergebnisse

Innerhalb dieser Arbeit wollen wir verschiedene Aussagen der Literatur bestätigen: Die Dimensionalität ist ein großes Problem der Lernalgorithmen. Lernverfahren sind momentan noch ungeeignet für das Lernen von komplexen Strategiespielen und sie werden von manuell optimierten Heuristiken dominiert.
Dahingehend stellen wir folgende Hypothesen auf:

1. Der Heuristik Agent wird in beiden Strategiespielen gegen den lernenden Agenten mindestens 60% aller Testspiele gewinnen.
2. Das TD-Q-Lernen kann, innerhalb von maximal 10.000 Trainingsspielen gegen sich selbst, keine Strategie entwickeln, die in 100 Testspielen häufiger Gewinnt, als der vorausschauende Heuristik Agent.
3. Die Konvergenzgeschwindigkeit, ist die Geschwindigkeit, die das TD-Q-Lernen für das Lernen einer optimalen Strategie benötigt. Sie ist voraussichtlich stark von der Dimensionalität des Ausgangsproblems abhängig, d.h. das TD-Q-Lernen ist nur auf sehr einfache bzw. niedrig dimensionierte Probleme anwendbar.

Kapitel 2

Strategiespiele und Spielregeln

In den ersten beiden Unterkapiteln werden die Strategiespiele Tic Tac Toe (Abschnitt 2.1) und Reversi (Abschnitt 2.2) vorgestellt und die Regeln dieser beiden Spiele werden festgelegt. Diese beiden Strategiespiele dienen als Umgebungen für den lernenden Agenten (TD-Q-Agent). Der TD-Q-Agent soll, innerhalb dieser beiden unbekannten Umgebungen, eine möglichst optimale Verhaltensstrategie lernen.

2.1 Das Strategiespiel Tic Tac Toe

In diesem Abschnitt definieren wir die Regeln des Strategiespiels Tic Tac Toe. Tic Tac Toe ist ein Spiel, welches von genau zwei Spielern gespielt wird. Während eines gesamten Spiels (eine Partie) darf ein Spieler nur Kreuze setzen und der andere Spieler nur Kreise. Wir können uns die Kreuze und Kreise als Spielfiguren vorstellen. Eine Spielfigur die auf das Spielfeld gesetzt wurde, darf seine Position nicht mehr verändern. Das klassische Tic Tac Toe hat 9 Spielfelder (ein 3 x 3 Spielbrett). Innerhalb dieser Arbeit betrachten wir auch ein Tic Tac Toe Spiel mit 16 Spielfeldern (ein 4 x 4 Spielbrett). Der beginnende Spieler muss Kreuzspielfiguren setzen und der nachziehende Spieler Kreisspielfiguren.

Spielzüge jeder Spieler setzt abwechselnd ein Kreuz bzw. einen Kreis in ein Spielfeld des Spielbretts. Eine Spielfigur kann in jedes freie Spielfeld gesetzt werden, außer dieses ist bereits mit einer anderen Spielfigur besetzt. Die Spieler führen solange ihre Spielzüge aus, bis eine Siegesformation eintritt oder alle Spielfelder besetzt sind.

Ziel des Spiels ist es, vier Kreuze bzw. vier Kreise in einer bestimmten Position anzuordnen (Siegesformation). Es existieren mehrere unterschiedliche Anordnungsmöglichkeiten von Spielfiguren, die das Spiel beenden und einen Sieg herbeiführen. Bei einem 4 x 4 Spielfeld existieren vier vertikale, vier horizontale und zwei diagonale Anordnungsmöglichkeiten der Spielfiguren, welche einen Sieg herbeiführen würden. Insgesamt zehn verschiedene Siegesanordnungen für beide Spieler. Sind alle Spielfelder besetzt und für keinen der Spieler ist eine Siegesformation aufgetreten, dann gewinnt beziehungsweise verliert keiner der beiden Spieler und es entsteht ein Unentschieden.

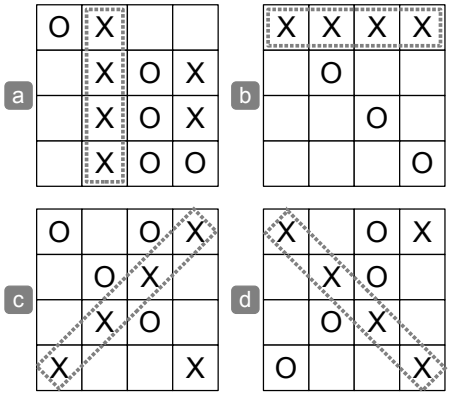


Abbildung 2.1 Tic Tac Toe Siegesformationen für den Kreuzspieler.

Vier mögliche Siegesformationen sind in Abbildung 2.1 dargestellt. (a) Kreuz gewinnt, mit einer vertikalen Siegesformation. (b) Kreuz gewinnt, mit einer horizontalen Siegesformation. (c) Kreuz gewinnt, mit einer diagonalen Siegesformationen. (d) Kreuz gewinnt, mit einer diagonalen Siegesformation.

2.2 Das Strategiespiel Reversi

In diesem Abschnitt definieren wir die Regeln des Strategiespiels Reversi. Reversi ist ein komplexeres Strategiespiel als Tic Tac Toe, weil Reversi 64 Spielfelder (ein 8 x 8 Spielbrett) hat. Das Reversi Spielbrett ist um den Faktor 4 größer, als das 4 x 4 Tic Tac Toe Spielbrett.

Reversi oder auch Othello genant, ist ein Spiel für zwei Personen die gegeneinander antreten. Eine Person setzt weiße runde Spielsteine und die andere Person schwarze runde Spielsteine. Jede neue Partie Reversie beginnt im selben Ausgangszustand (siehe Abbildung 2.2). Die Spieler setzen nacheinander genau einen Spielstein und einmal gesetzte Spielsteine können ihre Position nicht mehr verändern.

Anmerkung zu Abbildung 2.2 Die äußeren weiß hinterlegten Reihen, in denen sich Zahlen befinden, dienen dazu, die Positionen der einzelnen Spielfelder genau zu definieren. In der Ausgangsspielsituation befinden sich bereits 2 weiße Spielsteine, an den Positionen (3,4) und (4,3) und zwei schwarze Spielsteine, an den Positionen (3,3) und (4,4).

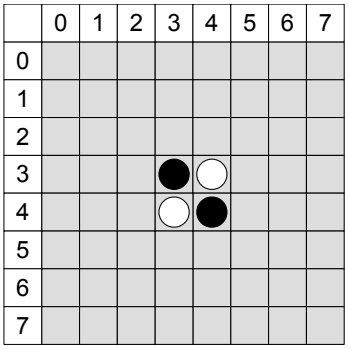


Abbildung 2.2 Ausgangsspielzustand Reversi.

Eine Spielregel von Reversi ist, dass gesetzte Spielsteine ihre Farbe ändern können, wenn sie vertikal, horizontal oder diagonal von einem Spielstein des Gegenspielers eingeschlossen werden. Dann wechseln die Spielsteine ihre Farbe und gehören dem Gegenspieler. Ein korrekter Spielzug muss immer mindestens einen gegnerischen Spielstein erobern.
Weiterhin darf ein Spielstein nur dann gesetzt werden, wenn ein anderer Spielstein, in einer diagonalen, vertikalen oder horizontalen Linie, existiert. Es dürfen auch keine freien Felder zwischen dem zu setzenden Steinen liegen.

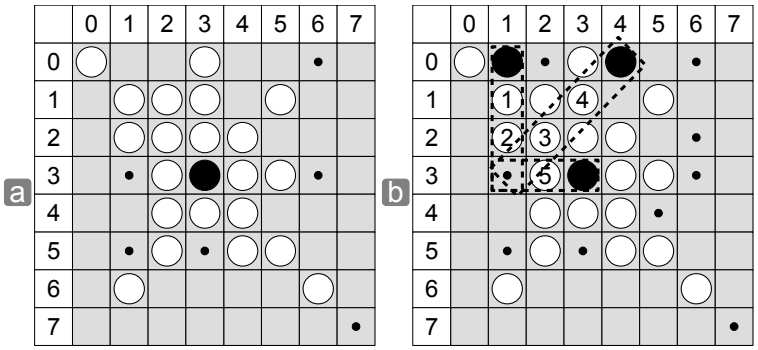


Abbildung 2.3 Fortgeschrittene Spielzugmöglichkeiten Reversi.

Anmerkung zu Abbildung 2.3, diese zeigt zwei möglicherweise auftretende Spielsituationen, die einzig verdeutlichen sollen, welche Zugmöglichkeiten der Spieler mit den schwarzen Spielsteinen hat und warum nur diese Züge möglich sind. Die kleinen schwarzen Punkte zeigen die Positionen an denen ein schwarzer Spielstein gesetzt werden darf. (a) Eine fortgeschrittene Spielsituation mit maximal einem möglichen schwarzen Spieltein. (b) Eine Spielsituation mit maximal 3 möglichen schwarzen Spielsteinen für die Position (3,1).

Abbildung 2.3 zeigt zwei verschiedene Reversi Spielsituationen (Schwarz ist am Zug). Die kleinen schwarzen Punkte symbolisieren zulässige Spielzüge. In Spielsituation (a) hat Spieler Schwarz genau 6 Spielzugmöglichkeiten. In jedem dieser Spielzüge erobert er mindestens einen weißen Spielstein und die Reihe wird nicht durch einen schwarzen Spielsein unterbrochen. In Spielsituation (b) hat Spieler Schwarz 9 Spielzugmöglichkeiten. Das setzen eines Spielsteins auf Position (3, 1) würde dem schwarzen Spieler 5 weiße Spielsteine einbringen, da mehrere schwarze Spielsteine diagonal, horizontal und vertikal an diese Position angrenzen. Die 5 eroberten weißen Spielsteine würden dann die schwarze Spielfarbe annehmen.

Ziel des Spiels ist es, am Ende des Spiels mehr Spielsteine seiner eigenen Farbe zu haben, als der Gegner Spielsteine in seiner Farbe hat. Das Spiel endet, wenn keiner der beiden Spieler mehr einen Spielstein, nach den Regeln des Spiels, auf das Spielbrett setzen kann.

Kapitel 3

Realisierung des Heuristik Agenten

Für die Realisierung des Heuristik Agenten müssen wir ein Verfahren implementieren, welches, in angemessener Rechenzeit, möglichst optimale Spielzüge für jeden Spielzustand berechnet.

Dazu gibt es das Verfahren der Minimax-Suche (Abschnitt 3.1), die durch die Alpha-Beta-Kürzung (Abschnitt 3.2) optimiert wird.

Die Minimax-Suche erstellt und durchsucht einen Suchbaum. Diese Suchbäume für Strategiespiele nehmen sehr große Dimensionen an, deshalb muss der Suchbaum abgeschnitten werden und die Suchtiefe begrenzt werden. Dazu bedienen wir uns der Alpha-Beta-Kürzung und der iterativ vertiefenden Tiefensuche (Abschnitt 3.3).

Ein Suchbaum besteht aus einem Wurzelknoten, mehreren inneren Knoten und mehreren Blattknoten. Nur die Blattknoten liefern Ergebnisse des Spiels. Ein Suchbaum für Spielzustände, wird auch als Spielbaum bezeichnet.

Durch das Begrenzen der Suchtiefe, werden keine Blattknoten gefunden, d.h. keine Spielergebnisse.

Deshalb bedarf es zusätzlich der Heuristiken (Abschnitt 3.4), die dazu führen, dass die Spielzustände auch vor erreichen eines Blattknotens bewertet werden können und somit auch Ergebnisse liefern.

3.1 Minimax-Suche

Die Minimax-Suche ist eine Spieltheorie die wir für unsere Strategiespiele Tic Tac Toe und Reversi anwenden könne, weil sie, deterministische und vollständig überschaubare Nullsummenspiele sind. Ein Nullsummenspiel bedeutet, gewinnt ein Spieler eine Partie, dann verliert der Gegenspieler automatisch in gleicher Höhe. In deterministischen Spielen treten keine zufälligen Zustandsübergänge auf und in vollständig überschaubaren Spielen existieren keine unbekannten Spielinformationen (vgl. [RN12, S. 206]).

Nachfolgend erläutern wir die von Russell und Norvig beschriebene Minimax-Suche.

"In einem normalen Suchproblem wäre die optimale Lösung eine Folge von Aktionen die zu einem Zielzustand führt - einem Endzustand, bei dem es sich um einen Gewinn handelt. In einer adversialen Suche dagegen hat Min auch noch etwas zu sagen. Max muss also eine mögliche Strategie finden, die den Zug von Max ab dem Ausgangszustand angibt und dann die Züge von Max in den Zuständen, die aus den einzelnen Gengenzügen von Min auf diese Züge resultieren usw. [RN12, S. 208]"
Anders ausgedrückt, berücksichtigt die Minimax-Suche, gegenüber anderen uninformierten Suchverfahren (z.B. Breitensuche oder Tiefensuche), dass ein Gegenspieler existiert. Der Gegenspieler führt den für sich optimalen Zug aus, d.h. er wird den anderen Spieler, wann immer es geht, behindern. Ein Spieler wird als MAX bezeichnet und der Gegenspieler als MIN. Spieler MAX versucht einen maximalen Gewinn für sich zu erlangen und Spieler MIN versucht den erreichbaren Gewinn von MAX zu minimieren.

In Abbildung 3.1 wird der Ablauf der Minimax-Suche veranschaulicht. Der Minimax-Suchbaum berücksichtigt jeden Zustand, indem sich die Spielwelt befinden kann. Im ersten Spielzug könnte Spieler MAX (Maximumknoten) sein Kreuzspielstein in die obere linke Ecke setzen, daraus ergeben sich neue Zustandsmöglichkeiten. Spieler MIN (Minimumknoten) könnte seinen Kreisspielstein ein Feld weiter rechts und in die selbe Reihe wie Spieler MAX setzen. Die Abbildung bzw. die Minimax-Suche muss rekursiv betrachtet werden, denn erst in den Blattknoten des Suchbaums, sind die Spielergebnisse zu finden. Von seinen Blattknoten ausgehend entscheidet sich MIN für den geringsten Nutzwert und MAX für den höchsten Nutzwert. Die Entscheidungen stehen in direkter Abhängigkeit zur vorherigen Entscheidung des Gegenspielers.

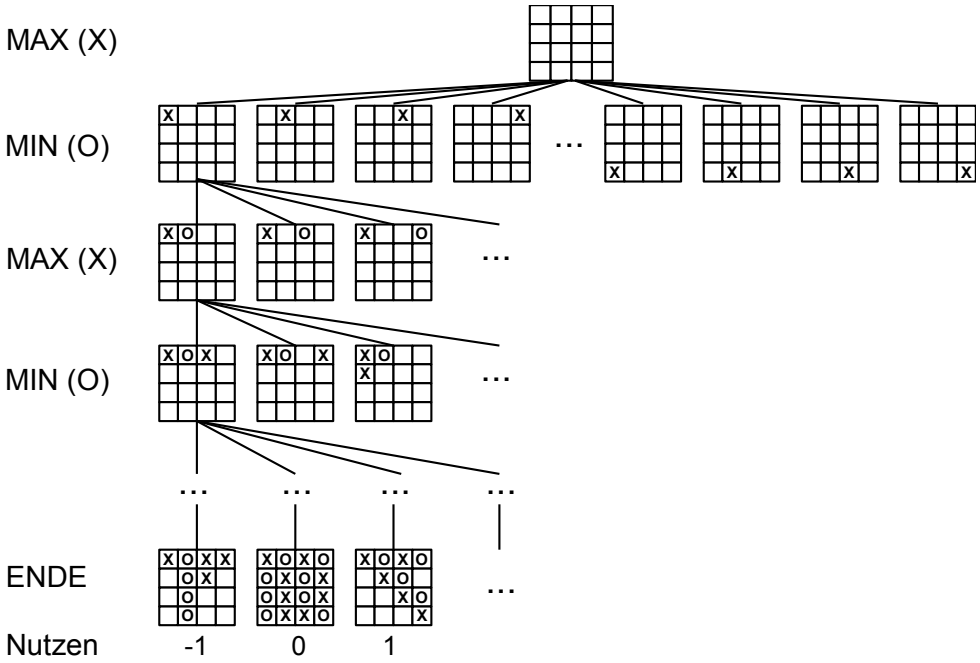


Abbildung 3.1 Ein (partieller) Suchbaum vgl. [RN12, S. 208]

Wolfgang Ertel beschreibt in dem nachfolgenden Zitat das Verhältnis von Problemkomplexität und Suchbaumgröße, was belegt, das Suchbäume für Strategiespiele enorme Dimensionen annehmen.

"Der effektive Verzweigungsfaktor beim Schachspiel liegt etwa bei 30 bis 35. Bei einem typischen Spiel mit 50 Zügen pro Spieler hat der Suchbaum dann mehr als $30^{100} \approx 10^{148}$ Blattknoten. Der Suchbaum lässt sich also bei weitem nicht vollständig explorieren. Hinzu kommt, dass beim Schachspiel oft mit Zeitbeschränkung gespielt wird. Wegen dieser Realzeitanforderung wird die Tiefe des Suchbaums auf eine passende Tiefe, zum Beispiel acht Halbzüge, beschränkt. [Ert13, S. 114 f.]"

3.2 Alpha-Beta-Kürzung

Eine Möglichkeit die Rechenzeit der Minimax-Suche zu verbessern, ist das Kürzen oder Beschneiden des Suchbaums (eng. Pruning).
Wolfgang Ertel erklärt die Alpha-Beta Suche wie folgt vgl. [Ert13, S. 116]:
Beim Alpha-Beta-Kürzen wird der Teil des Suchbaums beschnitten, der keinen Effekt auf das Ergebnis der Minimax-Suche hat. Der Minimax Algorithmus wird um zwei Parameter Alpha und Beta ergänzt. Die Bewertung erfolgt an jedem Blattknoten des Suchbaums. Alpha enthält den aktuell größten Wert, für jeden Maximumknoten, der bisher bei der Traversierung (Erkundung oder das Durchlaufen) des Suchbaums gefunden wurde. In Beta wird für jeden Minimumknoten der bisher kleinste gefundene Wert gespeichert. Ist Beta an einem Minimumknoten kleiner oder gleich Alpha ($Beta \leq Alpha$), so kann die Suche unterhalb von diesem Minimumknoten abgebrochen werden. Ist Alpha an einem Maximumknoten größer oder gleich Beta ($Alpha \geq Beta$), so kann die Suche unterhalb von diesem Maximumknoten abgebrochen werden.

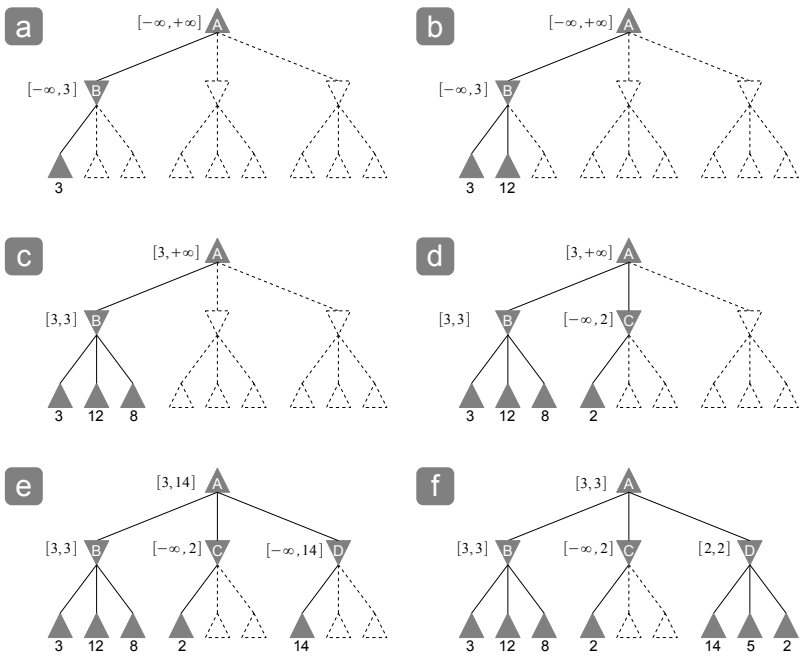


Abbildung 3.2 Ein Alpha Beta Suchbaum [RN12, S. 213].

Verdeutlichen wir das Alpha-Beta-Pruning an Hand eines Beispiels (Abbildung 3.2). Die nachfolgenden Erklärungen zur Abbildung 3.2 sind in ähnlicher Form in der Literatur zu finden [RN12, S. 212 ff.].

Ein Dreieck mit der Spitze nach oben ist ein Maximumknoten und ein Dreieck mit der Spitze nach unten ist ein Minimumknoten. Leere Dreiecke ohne einen bezeichnenden Buchstaben und gestrichelter Umrandung sind noch nicht explorierte Knoten. Durchgängige Linien verweisen auf bereits besuchte Pfade und gestrichelte Linien verweisen auf noch nicht besuchte Pfade. Die Zahlen unterhalb der Blattknoten sind die Nutzwerte, die der maximierende Spieler erhält, wenn er den Pfad bis zu diesem Blattknoten durchschreitet.

(a) Minimumknoten B findet einen Nutzwert 3, da dieser Wert der bisher kleinste gefundene Wert ist wird er in Beta gespeichert.

(b) Der Minimumknoten B exploriert einen zweiten möglichen Nutzwert 12. Dieser Wert ist höher als der vorher gefundene und in Beta gespeicherte Wert 3, daher wird der minimierende Spieler versuchen diesen Nutzwert für den maximierenden Spieler zu vermeiden. Der neue Wert wird vom Minimumknoten B ignoriert und Beta bleibt unverändert.

(c) Minimumknoten B findet den Wert 8, dieser ist genau wie 12 größer als 3 und daher wird Spieler MIN vermeiden, dass Spieler MAX zu diesem Spielergebnis gelangt. Minimumknoten B hat alle seine nachfolgenden Knoten exploriert. Maximumknoten A wird vom Minimumknoten B maximal den Nutzwert 3 erhalten, somit ergibt sich für den Maximumknoten A, dass dieser mindestens den Nutzwert 3 erreichen kann.

(d) Ein weiterer Minimumknoten ist C. Der erste Blattknoten von C liefert einen Nutzwert von 2, weil dieser Wert der erste gefundene Wert unterhalb des Minimumknotens C ist, wird er in Beta gespeichert. C wird Maximumknoten A maximal einen Nutzwert 2 liefern. A wiederum kann durch Minimumknoten B bereits einen minimalen Nutzwert von 3 erhalten und hat diesen in Alpha gespeichert. Es gilt $Beta \leq Alpha$ und es ist nicht notwendig die Knoten unterhalb von C weiter zu explorieren. Selbst wenn ein größerer Nutzwert gefunden werden würde, entscheidet sich der minimierende Spieler trotzdem für den kleineren Wert und würde ein kleinerer Nutzwert als 2 gefunden werden, dann entscheidet sich der maximierende Spieler für den Nutzwert 3, den Minimumknoten B liefert. Folglich kann der Suchbaum an dieser Stelle abgeschnitten werden, weil weitere gefundene Nutzwerte keinen Einfluss mehr auf das Ergebnis haben.

(e) Der letzte von A zu erreichende Minimumknoten wird exploriert. Der erste Blattknoten unterhalb des Minimumknoten D liefert den Nutzwert 14. Dieser Wert wäre für Maximumknoten A eine starke Verbesserung, weil dieser bisher nur maximal einen Nutzwert von 3 erreichen konnte. Der minimierende Spieler hat noch zwei weitere Möglichkeiten(Knoten) zu explorieren und daher wird er versuchen einen geringeren Nutzwert als 14 zu finden.

(f) Minimumknoten D findet in den beiden letzten Blattknoten die Nutzwerte 5 und 2. Der minimierende Spieler wählt die Möglichkeit mit dem geringsten Nutzwert 2. Dieser Nutzwert wird zum neuen Beta Wert. Der Suchbaum wird unterhalb vom Minimumknoten D jedoch nicht abgeschnitten, weil der Nutzwert 2 erst im zuletzt explorierten Knoten gefunden wurde. Theoretisch könnten zwei Pfade unterhalb des Minimumknoten D abgeschnitten werden, wenn der Blattknoten mit dem Nutzwert 2 zuerst exploriert worden wäre.

Durch die Alpha-Beta-Kürzung kann ein großer Teil des Suchbaums abgeschnitten werden, ohne das Ergebnis zu beeinflussen. Die Rechenzeit für die Exploration des Suchbaumes, ist an dieser Stelle immer noch zu hoch, deshalb ist eine Suchtiefenbegrenzung erforderlich. Daher wird das Prinzip des Begrenzens der Suchbaumtiefe der iterativ vertiefende Tiefensuche angewendet.

3.3 Iterativ vertiefende Tiefensuche

Um dieses Verfahren zu beschreiben, fassen wir die Ausführungen, zum Thema iterativ vertiefende Tiefensuche, von Russell und Norvig vgl. [RN12, S. 116] zusammen:

Die iterativ vertiefende Tiefensuche (eng. Iterative Deepening) ist ein kombinatorisches, uninformiertes Suchbaumverfahren und kombiniert die Breitensuche mit der Tiefensuche. Die Suchstrategien der uninformierten Suchverfahren haben keine zusätzlichen Informationen über Zustände, außer den in der Problemdefinition vorgegebenen. Alles was sie tun können, ist, Nachfolger zu erzeugen und einen Zielzustand von einem Nichtzielzustand zu unterscheiden. Ein Nachfolger, ist ein Suchbaumknoten, der unterhalb eines anderen Suchbaumknotens existiert. Die Reihenfolge der Suche ist entscheidend für die Unterscheidung der einzelnen uninformierten Suchverfahren.

Die Breitensuche expandiert (erweitert oder vergrößert) zuerst alle Nachfolger, die in derselben Tiefe liegen, beginnend mit dem Wurzelknoten. Diesen Schritt wiederholt die Breitensuche bis ein gesuchtes Ergebnis gefunden wird oder der Suchbaum vollständig exploriert (erkundet) ist.

Die Tiefensuche exploriert zuerst die tiefsten Knoten des Suchbaums (eng. Depth-first). Erreicht die Tiefensuche einen Endknoten, der nicht dem gesuchten Ergebnis entspricht, dann werden die alternativen Knoten, die sich eine Tiefenebene höher befinden, exploriert.

Kombinieren wir die uninformierten Suchverfahren Tiefen- und Breitensuche miteinander und mit einer Grenze für die Suchtiefe, erhalten wir die iterative vertiefende Tiefensuche. Diese expandiert zuerst die Nachfolger des Wurzelknotens der Suchtiefe 1. Sind alle Knoten auf dieser Ebene exploriert, dann wird die Schranke für die aktuelle Suchtiefe um 1 erhöht (Iteration) und die Knoten der Suchtiefe 2 werden expandiert. Diese Schritte wiederholt die iterativ vertiefende Tiefensuche bis ein Ziel gefunden wird.

Für diese Arbeit ist dieses Verfahren relevant, weil der vorausschauende Heuristik Agent bzw. die in seiner Implementierung realisierte Alpha-Beta Suche, an die iterativ vertiefende Tiefensuche angepasst wird. Der Agent wird also nicht den gesamten Zustandsraum (Suchbaum) durchsuchen, sondern seine Zugvorausschau wird auf eine bestimmte Anzahl von Zügen begrenzt werden. Meistens wird eine Zugvorausschau von 2 Zügen nicht ausreichen, um einen Blattknoten des Suchbaumes zu erreichen, daher führen wir noch Heuristiken (Bewertungsfunktionen) ein.

3.4 Heuristik

”Heuristiken sind Problemlösungsstrategien, die in vielen Fällen zu einer schnelleren Lösung führen als die uninformierte Suche. Es gibt jedoch keine Garantie hierfür. Die heuristische Suche kann auch viel mehr Rechenzeit beanspruchen und letztlich dazu führen, dass die Lösung nicht gefunden wird. [Ert13, S. 105]”

Wir leiten aus der Definition von Wolfgang Ertel folgendes ab:
Eine Heuristik (Bewertungsfunktion) berechnet eine Gewinnchance für einen gegebenen Spielzustand, d.h. ob der Spieler in diesem Spielzustand eher gewinnen oder verlieren könnte. Die Verwendung einer Heuristik ist keine Garantie für ein korrektes Ergebnis. In der Regel, wird für bessere Rechenzeit, ein mögliches schlechteres Ergebnis akzeptiert, d.h. eine heuristische Zustandsbewertung muss nicht dem

wahren Nutzen des Zustands entsprechen. Die Qualität einer Heuristik ist demnach ausschlaggebend für das Spielergebnis. Ein schlechte Stellungsbewertung (Heuristik), kann schlechte Spielzüge verursachen oder fatale Spielzüge des Gegners übersehen.

Die Verwendung einer Heuristik ermöglicht es, Knoten eines Spielbaums zu bewerten, die keine Blattknoten sind. Ohne Heuristiken liefern nur Blattknoten Spielergebnisse und mit Heuristik liefern alle Knoten Spielergebnisse.

Wolfgang Ertel schreibt sinngemäß zur Heuristik eines Schachspiels (vgl. [Ert13, S. 118]):
Diese Schach Heuristik ist entstanden aus der Zusammenarbeit von Schachexperten und Wissensingenieuren. Die Schachexperten verfügen über Wissen und Erfahrungen bezüglich des Schachspiels, der Strategien, guter Zugstellungen und schlechter Zugstellungen. Der Wissensingenieur hat die meist sehr schwierige Aufgabe dieses Wissen in eine, für ein Programm, anwendbare Form zu bringen.

Eine Bewertungsfunktion B(s) für ein Schachspiel enthält folgende Elemente, wobei s der Parameter für den Spielzustand ist [Ert13, S. 119]:

B(s) = a_1 x Material + a_2 x Bauernstruktur + a_3 x Königssicherheit
+ a_4 x Springer im Zentrum + a_5 x Läufer Diagonalabdeckung + ...,

das mit Abstand wichtigste Feature (Merkmal) "Material" nach der Formel

Material = Material(eigenes Team) - Material(Gegner)

Material(Team) = Anzahl Bauern(Team) x 100 + Anzahl Springer(Team) x 300
+ Anzahl Läufer(Team) x 300 + Anzahl Türme(Team) x 500
+ Anzahl Damen(Team) x 900

Nachfolgend liefern wir eine formale, allgemeine Darstellung einer Heuristik. Formal definieren Russell und Norvig Bewertungsfunktionen [RN12, S. 218]:

$$\hat{U}_\theta(s) = \theta_1 f_1(s) + \theta_2 f_2(s) + ... + \theta_n f_n(s),$$

als eine gewichtete Lineare Funktion einer Menge von Merkmalen (oder Basisfunktionen) $f_1, ..., f_n$. Die Parameter $\theta = \theta_1, ...\theta_n$ sind die Gewichtungen der einzelnen Merkmale, d.h. ein Parameter bestimmt, wie "wichtig" ein Merkmal ist.

3.4.1 Tic Tac Toe Heuristik

Das erste Merkmal unserer Tic Tac Toe Heuristik ist die Kontrolle der mittleren Spielfelder. Spielsituationen in denen die mittleren Spielfelder mit eigenen Spielfiguren besetzt sind, erhalten eine höhere Bewertung. Die erste Spielfigur soll in die mittleren Spielfelder gesetzt werden. Die zweite Spielfigur soll in die vom Gegenspieler nicht gestörte mittlere Position gesetzt werden.

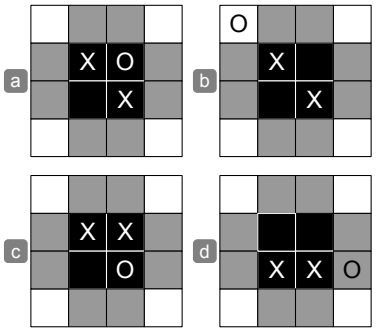


Abbildung 3.3 Tic Tac Toe Eröffnungssituationen.

Abbildung 3.3 zeigt 4 verschiedene Spielsituationen. Wir werden das erste Merkmal der Tic Tac Toe Heuristik an diesem Beispiel erklären. Spielsituation (a) wird, nach der Definition des ersten Merkmals, eine bessere heuristische Bewertung erhalten, als Spielsituation (b). Die Kreuzspielsteine sind in beiden Spielsituationen zwar gleich positioniert, aber der Kreisspielstein stört in Spielsituation (b) die Siegesformation des Kreuzspielers. Aus dem selben Grund ist Spielsituation (c) ”wertvoller” oder ”nützlicher” als Spielsituation (d). In Spielsituation (c) ist eine mögliche Siegesformation des Kreuzspielers (mit bereits 2 Kreuzfiguren) ungestört. In Spielsituation (d) ist die diagonale Siegesformation des Kreuzspielers bereits gestört und somit unbrauchbar, hinsichtlich einer größeren Gewinnchance.

Das zweite Merkmal der Tic Tac Toe Heuristik ist die Beachtung der ungestörten Möglichkeiten für Siegesformationen. Eine Formationsmöglichkeit wird gefährlicher bzw. attraktiver, je mehr gleiche Spielfiguren sich bereits in dieser befinden. Wir stellen bei diesem Merkmal gegenüber, wie viele, vom Gegenspieler nicht gestörte Formationsmöglichkeiten einem Spieler zur Verfügung stehen. Gleichzeitig berücksichtigen wir, wie viele ungestörte Formationsmöglichkeiten der Gegenspieler hat.

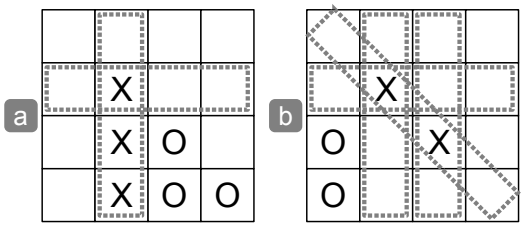


Abbildung 3.4 Tic Tac Toe Formationsmöglichkeiten.

Betrachten wir die ungestörten möglichen Siegesformationen in Abbildung 3.4. In Spielsituation (a) hat der Kreuzspieler 2 Möglichkeiten eine Siegesformation zu erreichen. In der vertikalen Formation sind bereits 3 Kreuze und in der diagonalen 1 Kreuz vorhanden. Die vertikal mögliche Siegesformation ist wesentlich hochwertiger, als die diagonal mögliche Siegesformation, weil diese bereits mehr Kreuzfiguren enthält. In Spielsituation (b) verfügt der Kreuzspieler über 4 mögliche ungestörte Siegesformationen, wobei die diagonal mögliche Siegesformation attraktiver sein sollte, als die anderen 3 möglichen Formationen. Die möglichen Siegesformationen des Gegenspielers sollen ebenfalls berücksichtigt werden.

3.4.2 Reversi Heuristik

Für die Erstellung der Reversi Heuristik verwenden wir bereits existierendes strategisches Wissen (siehe Sammlung von Reversi Strategien [Mac15]).

Das erste wichtige Merkmal für unsere Reversi Heuristik ist die Mobilität. Das Merkmal der Mobilität wird in zwei Merkmale aufgeteilt. Ein Merkmal für die aktuelle Mobilität und ein zweites Merkmal für die mögliche Mobilität. Mit aktueller Mobilität ist die Anzahl aller möglichen Spielzüge in einem aktuellen Spielzustand gemeint. Die Anzahl der Spielsteine am Ende des Spiels ist zwar entscheidend, aber in den Spielzügen bevor das Spiel endet, ist der Spieler im Vorteil, der mehr Zugmöglichkeiten hat.

Abbildung 3.5 zeigt eine Spielsituation in der Spieler Weiß nahezu alle Spielsteine kontrolliert. Spieler Schwarz ist jedoch der einzige Spieler der noch über Mobilität verfügt, d.h. er kann noch Spielzüge ausführen. In den nachfolgenden 4 Spielzügen $(0,0) \rightarrow (0,7) \rightarrow (7,0) \rightarrow (7,7)$ gewinnt der Schwarze Spieler die Partie.

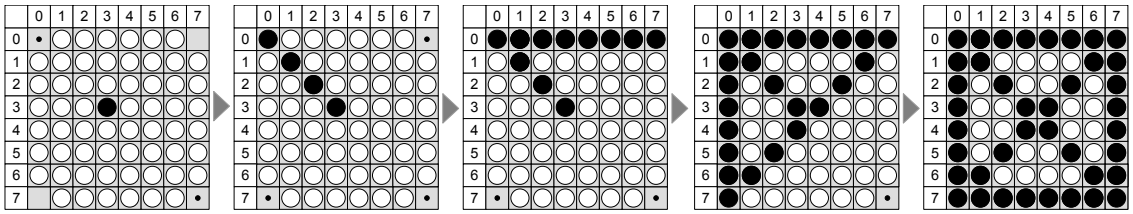


Abbildung 3.5 Reversi Merkmal der aktuellen Mobilität.

Die mögliche Mobilität beachtet alle freien Spielfelder, die an gegnerische Spielsteine Angrenzen. In Abbildung 3.6 (a) ist Spieler Weiß am Zug. Die aktuelle Mobilität ist in dieser Spielsituation identisch mit der möglichen Mobilität, denn es gibt nur 3 freie Spielfelder die an schwarze Spielsteine angrenzen. Bei der ersten Spielsituation in Abbildung 3.5 ist die aktuelle Mobilität gleich 2 und die mögliche Mobilität gleich 4.

Das zweite wichtige Merkmal für unsere Reversi Heuristik ist die Bewertung der Eckspielfelder und der Randspielfelder. In Abbildung 3.6 (b) sind bestimmte Spielfelder mit Buchstaben gekennzeichnet, diese Buchstaben repräsentieren Spielfelder mit bestimmten Eigenschaften. Alle X-Spielfelder (eng. x-squares) sollten unbedingt vermieden werden, denn sie bieten dem Gegner (fast immer) die Möglichkeit eine der 4 Ecken zu besetzen. Es existieren auch Strategien, die die vier Ecken generell, aus Gründen eingeschränkter Mobilität, vermeiden.

Das Ziel unserer Reversi Heuristik soll das Besetzen dieser 4 Ecken sein und gleichzeitig das Verhindern, dass der Gegenspieler diese Ecken besetzt. Genau wie die X-Spielfelder, bieten die C-Spielfelder direkten Zugang zu den 4 Ecken des Spielbretts. Die C-Spielfelder sollen daher ebenfalls vermieden werden. A- und B-Spielfelder sind zu bevorzugen und können besetzt werden. In Abbildung 3.6 (c) sind die einzelnen Bewertungen der Positionen veranschaulicht. Diese numerischen Bewertungen beziehen sich auf das gesamte Reversi Spielbrett, weil dieses symmetrisch ist.

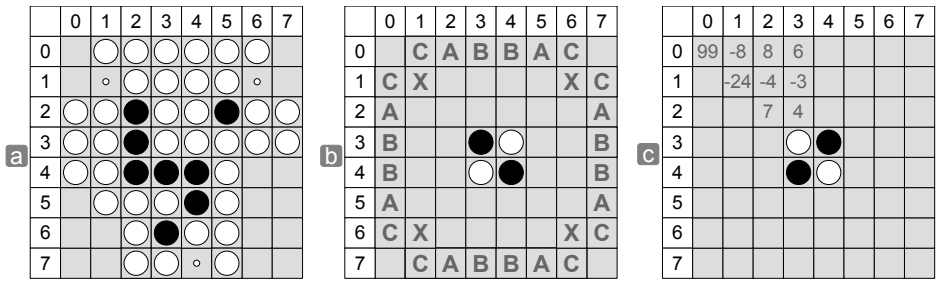


Abbildung 3.6 Merkmale einer Reversi Heuristik.

Kapitel 4

Realisierung des TD-Q-Agenten (Verstärkendes Lernen)

”Beim bestärkenden Lernen ist der Lerner ein entscheidungstreffender Agent, der in einer Umgebung Handlungen ausführt und Belohnung (oder Bestrafung) für seine Aktionen beim Versuch, das Problem zu lösen, erfährt. Nach einer Menge an Versuch-und-Irrtum-Durchläufen sollte er die beste Vorgehensweise lernen, welche der Sequenz an Aktionen entspricht, durch welche die Gesamtbelohnung maximiert wird. [Alp08, S. 397]”

”Ein zentrales Problem beim Verbessern der Stellungsbewertung aufgrund von gewonnenen oder verlorenen Partien ist heute bekannt unter dem Namen Credit Assignment. Man hat zwar am Ende des Spiels eine Bewertung des ganzen Spiels, aber keine Bewertung der einzelnen Züge. Der Agent macht also viele Aktionen und erhält erst am Ende ein positives oder negatives Feedback. Wie soll er nun den vielen Aktionen in der Vergangenheit dieses Feedback zuordnen? Und wie soll er dann seine Aktionen gegebenenfalls verbessern? Mit diesen Fragen beschäftigt sich das spannende junge Gebiet des Lernens durch Verstärkung (engl. reinforcement learning) [Ert13, S. 120].”

Wir wollen in dieser Arbeit ein Lernverfahren untersuchen und implementieren. Dieses Lernverfahren soll die Strategiespiele Tic Tac Toe und Reversi lernen. Nach den beiden oberen Definitionen, kann ein Agent, der verstärkendes Lernen anwendet, eine annähernd optimalen Strategie, für eine ihm unbekannte Umgebung, lernen. Wir erklären daher in diesem Kapitel verschiedene verstärkende Lernverfahren. Die Wert-Iteration mit der Bellman-Gelichung (siehe Abschnitt 4.3.1), die temporale Differenz (siehe Abschnitt 4.3.2) und das Q-Lernen (siehe Abschnitt 4.3.3).

Das Q-Lernen soll in dieser Arbeit angewendet und untersucht werden. Die Wert-Iteration, ist ein wichtiges Lernverfahren im verstärkenden Lernen. Sie ermöglicht es eine optimale Strategie für Strategiespiele zu lernen. Wert-Iteration hat auch Eigenschaften die im Temporalen Differenz Lernen wiederzufinden sind. Versteht man die Wert-Iteration fällt es leichter das Temporale Differenz Lernen zu verstehen. Das Q-Lernen ist eine Variante des Temporalen Differenz Lernens und wird daher auch als TD-Q-Lernen bezeichnet.

Bevor wir jedoch die einzelnen Lernverfahren erläutern, müssen wir noch definieren, wie wir die beiden Strategiespiele Tic Tac Toe und Reversi modellieren werden (siehe Abschnitt 4.1) und wie eigentlich eine optimale Strategie aussehen wird (siehe Abschnitt 4.2).

4.1 Markov-Entscheidungsprozess (MEP)

Der Markov-Entscheidungsprozess (MEP) oder MDP (engl. Markov decision process) nach Russell und Norvig [RN12, S. 752 ff.] ist ein sequentielles Entscheidungsproblem für eine vollständige beobachtbare, stochastische Umgebung mit einem Markov-Übergangsmodell und additiven Gewinnen. Der MEP besteht aus einem Satz von Zuständen (mit einem Anfangszustand s_0), einem Satz Actions(s) von Aktionen in jedem Zustand, einem Übergangsmodell P(s' | s, a) und einer Gewinnfunktion R(s).

Ganz ähnlich definiert Wolfgang Ertel die Markov-Entscheidungsprozesse [Ert13, S. 291]. Seine Agenten bzw. die Strategien der Agenten verwenden für die Bestimmung des nächsten Zustandes s_{t+1} nur Informationen über den aktuellen Zustand s_t und nicht über die Vorgeschichte. Dies ist gerechtfertigt, wenn die Belohnung einer Aktion nur von aktuellem Zustand und aktueller Aktion abhängt.

(sodass die Lernverfahren auf die Darstellung der Strategiespiele anwendbar sind)
Um darzustellen wie die Strategiespiele exakt auszusehen haben, verwenden wir einen Markov-Entscheidungsprozess, um die Strategiespiele zu modellieren. Die Eigenschaften eines Markov-Entscheidungsprozesses werden auch die Eigenschaften der Strategiespiele. Die Lernverfahren sind nur auf die Eigenschaften des Markov-Entscheidungsprozesses anwendbar. Wir modellieren nachfolgend die Eigenschaften eines MEP auf die beiden Strategiespiele Tic Tac Toe und Reversi.

4.1.1 Modellierung der Eigenschaften eines MEP auf Tic Tac Toe und Reversi

Tic Tac Toe und Reversi sind **sequentielle Entscheidungsprobleme**, denn die einzelnen Züge werden nicht direkt Belohnt, erst am Spielende wird ein Gewinner und ein Verlierer verkündet. Der Agent erhält dafür eine verspätete Verstärkung, die er auf die Spielzugsequenz aufteilen muss (siehe Abschnitt 4.3.2).

Wolfgang Ertel erklärt vollständig beobachtbare Spiele wie folgt (vlg. [Ert13, S. 114]): Schach, Reversi, Tic Tac Toe, 4-Gewinnt und Dame sind **vollständig beobachtbare** Spiele, denn jeder Spieler kennt immer den kompletten Spielzustand. Vollständig beobachtbare Spiele werden auch als Spiele mit vollständiger Information bezeichnet. Viele Kartenspiele wie zum Beispiel Skat, sind nur teilweise beobachtbar, denn der Spieler kennt die Karten des Gegners nicht oder nur teilweise.

Aus den Ausführungen von Wolfgang Ertel (vgl. [Ert13, S. 114]) ist folgende Beschreibung stochastischer Übergänge abzuleiten:
Ein **stochastischer Übergang** ist nur in einer nicht deterministischen Umgebung möglich. Tic Tac Toe und Reversi sind deterministische Strategiespiele, d.h. jeder Nachfolgezustand ist eindeutig definiert. Eine Aktionssequenz führt also immer zum selben Ergebnis. Es finden keine stochastischen Übergänge in Tic Tac Toe und Reversi statt. Backgammon ist ein nichtdeterministisches Strategiespiel, in diesem werden stochastische Übergänge durch ein Würfelergebnis bestimmt, es ist also vorher nicht eindeutig, welcher Nachfolgezustand durch eine Aktion eintreten wird.

Sinngemäße Definition des Übergangsmodells [RN12, S. 753]:
Das **Übergangsmodell** beschreibt das Ergebnis jeder Aktion in jedem Zustand. Ist das Ergebnis stochastisch, bezeichnet P(s'|s, a) die Wahrscheinlichkeit, den Zustand s' zu erreichen, wenn die Aktion a im Zustand s ausgeführt wird. Handelt es sich um einen Markov-Übergang, dann ist die Wahrscheinlichkeit s' von s zu erreichen, nur von s abhängig und nicht vom Verlauf der vorherigen Zustände.

Ein stochastisches Übergangsmodell für die Wahrscheinlichkeiten der Zustandsübergänge ist für Tic Tac Toe und Reversi nicht sinnvoll, da beide Spiele nicht vom Zufall abhängen (keine stochastischen Übergänge) und für jede Aktion in jedem Zustand nur ein einziger Zustandsübergang möglich ist.

Additive Gewinne, nach Russell und Norvig [RN12, S. 756], bestimmen über das zukunftsbezogene Verhalten des Agenten. Verwendet der Agent additive Gewinne, dann bedeutet das für den Agenten, jeder Nutzen eines Zustandes in einer gewählten Zustandsfolge ist gleich wertvoll. Zudem ist die Summe der Zustandsnutzen endlich, deshalb auch Modell des endlichen Horizonts. Der Nutzen einer Zustandsfolge ist wie folgt definiert:

$$U_h([s_0, s_1, s_2...]) = R(s_0) + R(s_1) + R(s_2) + ...$$

Spiele die unter Umständen nicht immer einen Endzustand erreichen, haben keinen endlichen Horizont, sondern einen unendlichen Horizont. Für diese Spiele ist ein Modell mit einem endlichen Horizont unangemessen, denn wir wissen nicht, wie lang die Lebensdauer des Agenten ist (vgl. [KLM96, S. 250]).

Tic Tac Toe und Reversi terminieren immer, nach einer endlichen Anzahl von Aktionen, in einem Endzustand. Wir können daher für beide Spiele das Modell des endlichen Horizonts bzw. additive Gewinne verwenden.

Verminderte Gewinne, nach Russell und Norvig [RN12, S. 756], unterscheiden sich von den additiven Gewinnen durch einen Vermeidungsfaktor γ. Der Verminderungsfaktor schwächt Zustände in der Zukunft immer weiter ab, d.h. je weiter ein Zustand in der Zukunft liegt, desto mehr wird er abgeschwächt. Der Nutzen für den ersten Zustand der Zustandsfolge wird nicht abgeschwächt. Ist γ gleich 1, sind die verminderten Gewinne gleich den additiven Gewinnen. Die additiven Gewinne sind also ein Sonderfall der verminderten Gewinne.

$$U_h([s_0, s_1, s_2...]) = R(s_0) + \gamma R(s_1) + \gamma^2 R(s_2) + ...$$

Wichtiges Fazit daraus:
Der Nutzen einer gegebenen Zustandsfolge ist die Summe der verminderten Gewinne, die während der Folge erhalten werden (vgl. [RN12, S. 757]).

4.2 Optimale Taktiken

Wir definieren in diesem Abschnitt, nach Russell und Norvig [RN12, S. 757 f.], was eine optimale Taktik ist: Eine Taktik (Strategie) beeinflusst das Verhalten des Agenten, d.h. sie empfiehlt welche Aktion der Agent in jedem Zustand ausführen soll. Aus Tradition wird beim verstärkenden Lernen eine Taktik mit dem Symbol π gekennzeichnet. Die Abbildung der Zustände auf Aktionen ist folgendermaßen definiert $\pi : S \rightarrow A$ (vgl. [Ert13, S. 290]) oder $\pi(s) = a$. Abhängig von den Dimensionen der Umgebung existieren unterschiedlich viele Taktiken. Eine optimale Taktik wird bestimmt durch den erwarteten Nutzen bei Ausführung der Taktik π beginnend in einem Startzustand s:

$$U^{\pi}(s) = E\left[\sum_{t=0}^{\infty} \gamma^t R(S_t)\right]. \tag{4.1}$$

Eine optimale Taktik hat im Vergleich zu allen anderen möglichen Taktiken einen gleich hohen oder höheren erwarteten Nutzen. Eine solche optimale Taktik wird gekennzeichnet durch π_s^*:

$$\pi_s^* = \underset{\pi}{\operatorname{argmax}}\, U^{\pi}(s). \tag{4.2}$$

Es ist möglich, dass mehrere optimale Taktiken für ein Problem existieren. Russell und Norvig erklären, dass für eine optimale Strategie π_s^*, auch π^* geschrieben werden kann, denn wenn Taktik π_a^* optimal beim Beginn in a und Taktik π_b^* optimal beim Start in b sind und sie einen dritten Zustand c erreichen, gibt es keinen vernünftigen Grund, dass sie untereinander oder mit π_c^* nicht übereinkommen.

Mit diesen Definitionen ist der wahre Nutzen eines Zustands einfach $U^{\pi^*}(s)-$ d.h. die erwartete Summe verminderter Gewinne, wenn der Agent eine optimale Taktik ausführt. Wir schreiben dies als U(s). Russell und Norvig unterstreichen den Sachverhalt, dass die Funktionen U(s) und R(s) ganz unterschiedliche Quantitäten sind, denn R(s) gibt den "kurzfristigen" Gewinn, sich in s zu befinden an, wohingegen U(s) den "langfristigen" Gesamtgewinn ab s angibt.

4.3 Verstärkende Lernverfahren

Verstärkendes Lernen (eng. reinforcement Learning) ist eine Lernkategorie des maschinellen Lernens. Problemstellungen des verstärkenden Lernens sind, u.a. das lernen von Strategiespielen, wie Schach, Reversi, Dame oder Backgammon. Der theoretische verstärkend lernende Lösungsansatz dieser Probleme ist wie folgt: ein Agent soll ein ihm unbekanntes Strategiespiel lernen (das Strategiespiel ist die unbekannte Umgebung), für einen Spielzug (Aktion) in einer Spielsituation (Zustand) erhält der Agent eine numerische Belohnung oder Bestrafung (Verstärkung), mittels dieser Verstärkung soll der Agent ein optimales Verhalten in der ihm unbekannte Umgebung erlernen.

Es gibt im verstärkenden Lernen mehrere Lernverfahren, die wir nachfolgend erläutern (Abschnitt 4.3.1 bis Abschnitt 4.3.3).

4.3.1 Dynamische Programmierung und Wert-Iteration

Verwenden wir bereits vorhandenes Wissen über Strategien und speichern dieses in Zwischenergebnisse über Teile von Strategien, dann bezeichnen wir diese Vorgehensweise zur Lösung von Optimierungsproblemen als dynamische Programmierung. Diese Vorgehensweise wurde bereits 1957 von Richard Bellman beschrieben [Ert13, S. 293].

Im Abschnitt optimale Taktiken haben wir gezeigt, dass der Nutzen U, in einem Zustand s, unter Beachtung einer Strategie π, berechnet werden kann, aus der Summe aller abgeschwächten Belohnungen, für jeden besuchten Zustand, in einem Zeitintervall von $t = 0$ bis ∞. Dementsprechend gibt eine optimale Taktik $\pi^*(s)$ für jeden Zustand s den Nachfolgezustand mit dem größtmöglichen erwarteten Nutzen an (siehe [RN12, S. 759]):

$$\pi^*(s) = \underset{a \in A(s)}{\operatorname{argmax}} \sum_{s'} P(s'|s,a)U(s'). \tag{4.3}$$

"Daraus folgt, dass es eine direkte Beziehung zwischen dem Nutzen eines Zustandes und dem Nutzen seiner Nachbarn gibt. Der Nutzen eines Zustandes ist der unmittelbare Gewinn für diesen Zustand plus dem erwarteten verminderten Gewinn des nächsten Zustandes, vorausgesetzt, der Agent wählt die optimale Aktion. Das bedeutet, der Nutzen eines Zustandes ist gegeben durch:

$$U(s) = R(s) + \gamma \max_{a \in A(s)} \sum_{s'} P(s'|s,a)U(s'). \tag{4.4}$$

Diese Gleichung wird als Bellman-Gleichung bezeichnet, nach Richard Bellman(1957). Die Nutzen der Zustände - durch Gleichung 4.1 als die erwarteten Nutzen nachfolgender Zustandsfolgen definiert - sind Lösungen der Menge der Bellman-Gleichungen. [RN12, S. 759]"

Die aus der Bellman-Gleichung formulierbare rekursive Aktualisierungsregel, auch die Bellman-Aktualisierung genannt, ist Hauptbestandteil des Wert-Iteration Algorithmus. Wolfgang Ertel notiert diese Aktualisierungsregel wie folgt [Ert13, S. 294]:

$$\hat{V}(s) = \max_a [r(s,a) + \gamma \hat{V}(\delta(s,a))]. \tag{4.5}$$

Dahingegen notieren Russell und Norvig die Bellman-Aktualisierung etwas anders [RN12, S. 760]:

$$U_{i+1}(s) \leftarrow R(s) + \gamma \max_{a \in A(s)} \sum_{s'} P(s'|s,a) U_i(s'). \tag{4.6}$$

Betrachten wir jetzt die Äquivalenzen der beiden Gleichungen. Ertel bezeichnet den Iterationsschritt in einem Zustand s als $\hat{V}(s)$ und Russell und Norvig definieren den Nutzwert für den Zustand s bei der i-ten Iteration als $U_i(s)$ und den Iterationsschritt bezeichnen sie als U_{i+1}. Die Gewinnfunktionen R(s) und r(s,a) sind leicht unterschiedlich. Funktion R(s) gibt den direkten Gewinn in einem Zustand s an und Funktion r(s,a) den Gewinn für eine Aktion, die im Zustand s ausgeführt wird. Die Funktionen $\max_a$ und $\max_{a \in A(s)}$ berechnen die Aktion a mit dem höchsten erwarteten Nutzen. Das stochastische Modell der Welt wird durch die Funktionen $\delta(s,a)$ und $P(s'|s,a)$ beschrieben. Beide Funktionen bilden die Wahrscheinlichkeit ab, dass ein Zustand s' erreicht wird, wenn eine Aktion a in Zustand s ausgeführt wird.

Den wahren Nutzen haben wir definiert als die erwartete Summe verminderter Gewinne. Die Verminderung wird in beiden Gleichungen durch den Abschwächungsfaktor γ notiert. Die erwartete Summe verminderter Gewinne ist die Summe aller Iterationsschritte bis zur Konvergenz beider Gleichungen. Der rekursive Funktionsaufruf in der Aktualisierungsregel von Wolfgang Ertel $\hat{V}(\delta(s,a))$ übergibt dem nächsten Iterationsschritt den Zustand s', der zu einer von δ bzw. von der Umgebung festgelegten Wahrscheinlichkeit eintrifft. In der Aktualisierungsgleichung von Russell und Norvig wird dies durch die Kombination des stochastischen Modells P(s' | s,a) und dem rekursiven Funktionsaufruf $U_i(s')$ realisiert.

Ein Lernverfahren (z.B. die adaptive dynamische Programmierung), welches die Wert-Iteration nutzt, wird im Rahmen dieser Arbeit jedoch nicht implementiert. Die dynamische Programmierung und die Wert-Iteration sind sehr wichtige Ansätze für Lernverfahren und die nachfolgenden Lernverfahren sind teilweise sehr eng mit Lernverfahren verwandt, die Wert-Iteration verwenden.

4.3.2 Lernen mit temporaler Differenz (TD-Lernen)

Bei dieser Lernmethode werden die Nutzen der beobachteten Zustände an die beobachteten Übergänge angepasst, sodass sie mit den Bedingungsgleichungen (siehe Bellman-Gleichung) übereinstimmen.

"Allgemeiner können wir sagen, wenn ein Übergang vom Zustand s in den Zustand s' stattfindet, wenden wir die folgende Aktualisierung mit $U^{\pi}(s)$ an:

$$U^{\pi}(s) \leftarrow U^{\pi}(s) + \alpha(R(s) + \gamma U^{\pi}(s') - U^{\pi}(s)). \tag{4.7}$$

Hier ist α der Lernratenparameter. Weil diese Aktualisierungsregel die Differenz der Nutzen aufeinanderfolgender Zustände verwendet, wird sie auch häufig als TD-Gleichung (Temporale Differenz) bezeichnet. [RN12, S. 966 f.]"

Der Lernratenparameter α gibt an, wie stark neue Nutzwerte die derzeitige Bewertungsfunktion anpassen können.

4.3.3 Q-Lernen (TD-Q-Lernen)

Wir beschreiben das Q-Lernen anhand der Ausführungen von Russell und Norvig (vgl. [RN12, S. 973 f.]):

Das Q-Lernen ist eine Variante des TD-Lernens und wird auch als TD-Q-Lernen bezeichnet. Die Aufgabe des TD-Q-Lernens ist eine optimale Strategie zu entwickeln. Das TD-Q-Lernen lernt nicht, wie bei einer Wert-Iteration, eine wahre Nutzenfunktion U(s), sondern eine Q-Funktion. Eine Q-Funktion ist eine Abbildung von Zustands/ Aktions-Paaren auf Nutzwerte. Q-Werte sind wie folgt mit Nutzwerten verknüpf [RN12, S. 974]:

$$U(s) = \max_{a} Q(s, a). \tag{4.8}$$

"Ein TD-Q-Agent der eine Q-Funktion lernt, braucht weder für das Lernen noch die Aktionsauswahl ein Modell der Form P(s' | s,a). Aus diesem Grund sagt man auch, das Q-Lernen ist eine modellfreie Methode. [RN12, S. 974]"

Dahingegen ist eine Nutzenfunktion U(s) abhängig von den abgeschwächten Nutzwerten aller nachfolgenden Zustände.

Die Aktualisierungsgleichung für TD-Q-Lernen lautet [RN12, S. 974]:

$$Q(s,a) \leftarrow Q(s,a) + \alpha(R(s) + \gamma \max_{a'} Q(s',a') - Q(s,a)). \tag{4.9}$$

Diese Aktualisierungsgleichung für TD-Q-Lernen wird Kernstück unserer Implementierung für den TD-Q-Agenten.

Was ist jedoch der Unterschied zwischen einer Belohnungsfunktion r(s, a) und einer Q-Funktion Q(s, a)? Die Funktion r(s, a) ist von der Umgebung definiert und kann vom Agenten nicht beeinflusst werden. Sollte diese Funktion dem Agenten eine numerische Verstärkung von -0,5 zuweisen, dann kann der Agent dies nicht ändern. Der Agent soll versuchen die Zusammenhänge der Zustands/Aktions-Paare zu lernen und Entscheidungen basierend auf seinen Lernerfahrungen zu treffen. Dies bezeichnen wir dann als Q-Lernen. Die vom Agenten gelernten zusammenhänge werden in Q-Werten gespeichert. Folglich wird in Q(s, a) oder Q[s, a] die gelernte Erfahrung des Agenten, für ein Zustand/Aktions-Paar, gespeichert.

Kapitel 5

Anforderungsdefinition und Modellierung

In diesem Kapitel definieren wir die funktionalen Anforderungen des Softwareprojektes für die ausgesuchten Strategiespiele, den Zufallsagenten, den Heuristik Agenten, den TD-Q-Agenten und die Agententests. Anschließend werden wir die funktionalen Anforderungen und die Zusammenhänge in einem Klassendiagramm veranschaulichen.

Bisher war geplant, für Reversi genau wie für Tic Tac Toe, das TD-Q-Lernen zu implementieren. Anhand der Tests der unterschiedlich großen Tic Tac Toe Spielfelder, werden wir ableiten können, dass bereits eine kleine Veränderung der Dimension des Spielfelds eine enorme Rechenzeitverlängerung der Lernphasen bedeutet. Wir können daraus schlussfolgern, dass das TD-Q-Lernen, mit tabellarischer Q-Funktionsrepräsentation, für Reversi nicht geeignet ist. Diese Aussage wird in der Auswertung bewiesen, deshalb werden wir den TD-Q-Agenten für Reversi nicht mehr implementieren.

5.1 Funktionale Anforderungen

Im nachfolgenden Abschnitt definieren wir die funktionalen Anforderungen der Software. Wir bestimmten, welche Funktionalitäten die Strategiespiele und die Agenten mindestens haben müssen und wie die Agenten getestet werden sollen. Wir definieren die Funktionalitäten, um den Funktionsbereich der Software einzugrenzen und einen Überblick zu verschaffen.

5.1.1 Tic Tac Toe Spielumgebung

class TicTacToe:
Die Spielumgebung soll die in Abschnitt 2.1 definierten Tic Tac Toe Spielregeln implementieren. Die Tic Tac Toe Spielumgebung repräsentiert eine Testumgebung für die Agenten. Der Zufallsagent wird in dieser Umgebung gegen den TicTacToe-Heuristik Agenten antreten. Der TD-Q-Agent soll zuerst diese Umgebung erkunden und lernen, sich in der Umgebung zurecht zu finden, d.h. der TD-Q-Agent soll eine TicTacToe-Siegesstrategie entwickeln.

makeMove(position):
Die Funktion soll Koordinaten erhalten. Die Koordinaten definiert exakt, auf welches Spielfeld eine Spielfigur gesetzt werden soll. Die Funktion soll diesen Spielzug, sollte dieser Regelkonform sein, ausführen.

undoMove():
Die Funktion soll den letzten durchgeführten Spielzug revidieren.

getPossibleMoves(): return list
Die Funktion soll eine Liste von Koordinaten liefern. In dieser Liste sind nur mögliche und regelkonforme Spielzüge (Koordinaten) enthalten.

getPlayerToMove(): return str
Die Funktion soll einen String zurückgeben. Dieser String repräsentiert den Spieler der aktuell einen Spielzug ausführen soll. Der String ”X” ist die Repräsentation des Kreuzspielers. Der String ”O” ist die äquivalente Repräsentation des Kreisspielers.

isTerminal: return bool
Die Funktion soll True zurück liefern, wenn der aktuelle Zustand der Umgebung ein Endzustand (Terminalzustand) ist, andernfalls liefert die Funktion ein False.

getReward: return float
Die Funktion soll eine nummerische Belohnung liefern. Die Belohnung soll abhängig sein vom aktuellen Spielzustand.

5.1.2 Reversi Spielumgebung

class Reversi:
Die Spielumgebung soll die in Abschnitt 2.2 definierten Reversi Spielregeln implementieren. Die Reversi Spielumgebung repräsentiert eine Testumgebung für die Agenten. Der Zufallsagent wird in dieser Umgebung gegen den Reversi-Heuristik Agenten antreten. Der TD-Q-Agent soll zuerst diese Umgebung erkunden und lernen, sich in der Umgebung zurecht zu finden, d.h. der TD-Q-Agent soll eine Reversi-Siegesstrategie entwickeln.

makeMove(position):
Die Funktion soll Koordinaten erhalten. Die Koordinaten definiert exakt, auf welches Spielfeld eine Spielfigur gesetzt werden soll. Die Funktion soll diesen Spielzug, sollte dieser Regelkonform sein, ausführen.

undoMove():
Die Funktion soll den letzten durchgeführten Spielzug revidieren.

getPossibleMoves(): return list
Die Funktion soll eine Liste von Koordinaten liefern. In dieser Liste sind nur mögliche und regelkonforme Spielzüge (Koordinaten) enthalten.

getPlayerToMove(): return str
Die Funktion soll einen String zurückgeben. Dieser String repräsentiert den Spieler der aktuell einen Spielzug ausführen soll. Der String ”B” ist die Repräsentation des schwarzen (black) Spielers. Der String ”W” ist die äquivalente Repräsentation des weißen (white) Spielers.

isTerminal: return bool
Die Funktion soll True zurück liefern, wenn der aktuelle Zustand der Umgebung ein Endzustand (Terminalzustand) ist, andernfalls liefert die Funktion ein False.

getReward: return float
Die Funktion soll eine nummerische Belohnung liefern. Die Belohnung soll abhängig sein vom aktuellen Spielzustand.

5.1.3 Agent des Zufalls

class RandomAgent:
Der Agent des Zufalls soll den Spieler symbolisieren, der keine Strategie hat und nicht lernt. Er soll seine Entscheidungen vollkommen zufällig treffen. In Kapitel 7 Validierung werden wir diesen Agenten, als Gegenspieler für die Heuristik Agenten und die lernenden TD-Q-Agenten einsetzen.

suggestRandomTicTacToeAction(ticTacToeState): return tuple
Diese Funktion soll eine Tic Tac Toe Spielsituation übergeben bekommen, d.h. eine Instanz der TicTacToe-Klasse. Die Funktion soll eine zufällige, aber zulässige Aktion zurückgeben.

suggestRandomReversiAction(reversiState): return tuple
Diese Funktion soll eine Reversi Spielsituation übergeben bekommen, d.h. eine Instanz der Reversi-Klasse. Die Funktion soll eine zufällige, aber zulässige Aktion zurückgeben.

5.1.4 Tic Tac Toe Heuristik Agent

class TicTacToeHeuristicSearchAgent:
Der Agent soll die in Abschnitt 3.4.1 erstellte Tic Tac Toe Heuristik und eine 2-Spielzüge vorausschauende Alpha-Beta Suche verwenden. Dieser Agent soll einen fortgeschrittenen Spielgegner repräsentiert, d.h. wir müssen mittels Testspielen gegen den Zufallsagenten zeigen, dass der Tic Tac Toe Heuristik Agent verhältnismäßig oft gewinnt. Dieser Agent soll in Tic Tac Toe Testspielen gegen den TD-Q-Agenten antreten. Die Ergebnisse sollen dabei helfen, die Leistungsfähigkeit und Grenzen des TD-Q-Lernens, hinsichtlich des Lernens von Tic Tac Toe, zu beurteilen.

suggestAction(ticTacToeState): return tuple
Diese Funktion soll eine Tic Tac Toe Spielsituation (eine Instanz der TicTacToe-Klasse) übergeben bekommen. Die Funktion soll, abhängig von der erhaltenen Spielsituation, eine Aktion vorschlagen. Die Aktion soll mittels der Tic Tac Toe Heuristik und einer 2-Zug Vorausschau und Alpha-Beta-Suche ermittelt werden.

5.1.5 Reversi Heuristik Agent

class ReversiHeuristicSearchAgent:
Der Agent soll die in Abschnitt 3.4.2 erstellte Reversi Heuristik und eine 2-Spielzüge vorausschauende Alpha-Beta Suche verwenden. Dieser Agent soll einen fortgeschrittenen Spielgegner repräsentieren, d.h. wir müssen mittels Testspielen gegen den Zufallsagenten zeigen, dass der Reversi-Heuristik Agent verhältnismäßig oft gewinnt.

suggestAction(reversiState): return tuple
Diese Funktion soll eine Reversi Spielsituation (eine Instanz der Reversi-Klasse) übergeben bekommen. Die Funktion soll, abhängig von der erhaltenen Spielsituation, eine Aktion vorschlagen. Die Aktion soll mittels der Reversi Heuristik und einer 2-Zug Vorausschau und Alpha-Beta-Suche ermittelt werden.

5.1.6 Tic Tac Toe TD-Q-Agent

class TicTacToeTDQLearningAgent:
Der Agent soll mittels des in Abschnitt 4.3.3 behandelten TD-Q-Lernens, eine Siegesstrategie für das Strategiespiel 9 und 16 Spielfelder Tic Tac Toe entwickeln.

learnTicTacToeInXGames(amountOfGames):
Die Funktion soll den Lernmodus des Agenten realisieren. Der Eingabeparameter legt die Anzahl der Trainingsspiele fest. Die Lernerfahrungen während dieser Trainingsspiele, sollen in einer SQLite Datenbank gespeichert werden. Für jede der 6 Trainingsphase (3 für 9 und 3 für 16 Spielfelder Tic Tac Toe) wird eine eigene Datenbank erstellt.

suggestAction(ticTacToeState): return tuple
Die Funktion soll eine Tic Tac Toe Spielsituation übergeben bekommen. Ausgehend von der Eingangsspielsituation, ist nur eine bestimmte Anzahl von Aktionen möglich. Abhängig von seinen Erfahrungen (SQL-Datenbank - Q-Funktion) und dem gegebenen Spielzustand, soll der Agent die mögliche Aktion mit dem höchsten gelernten Q-Wert zurückgeben.

5.1.7 Agententests in 9 Spielfelder Tic Tac Toe

class TestAgentsIn9FieldTicTacToe100Testgames:
Wir testen in dieser Testklasse 3 gelernte TD-Q-Strategien (100, 1.000 und 10.000 Trainingsspiele) für das 9 Spielfelder Tic Tac Toe. Die Funktionen unterscheiden zwischen beginnendem TD-Q-Agent oder beginnendem Gegenspieler (Zufallsagent oder Heuristik Agent). In jeder Funktion der Testklasse werden genau 100 Testspiele durchgeführt.

5.1.8 Agententests in 16 Spielfelder Tic Tac Toe

class TestAgentsIn16FieldTicTacToe100Testgames:
Wir testen in dieser Testklasse 3 gelernte TD-Q-Strategien (100, 1.000 und 10.000 Trainingsspiele) für das 16 Spielfelder Tic Tac Toe. Die Funktionen unterscheiden zwischen beginnendem TD-Q-Agent oder beginnendem Gegenspieler (Zufallsagent oder Heuristik Agent). In jeder Funktion der Testklasse werden genau 100 Testspiele durchgeführt.

Beispielfunktion → testTDQ100AgainstFirstMoveRandomAgent():
In dieser Funktion spielen der TD-Q-Agent (100 Trainingsspiele Strategie) und der Zufallsagent gegeneinander. Der Zufallsagent beginnt das Spiel.

5.2 Modellierung

Die Zusammenhänge der Anforderungen sind im nachfolgenden Klassendiagramm (siehe Abbildung 5.1) dargestellt.

In diesem Klassendiagramm sind die im Abschnitt 5.1 beschriebenen Funktionen vollständig wiedergegeben.

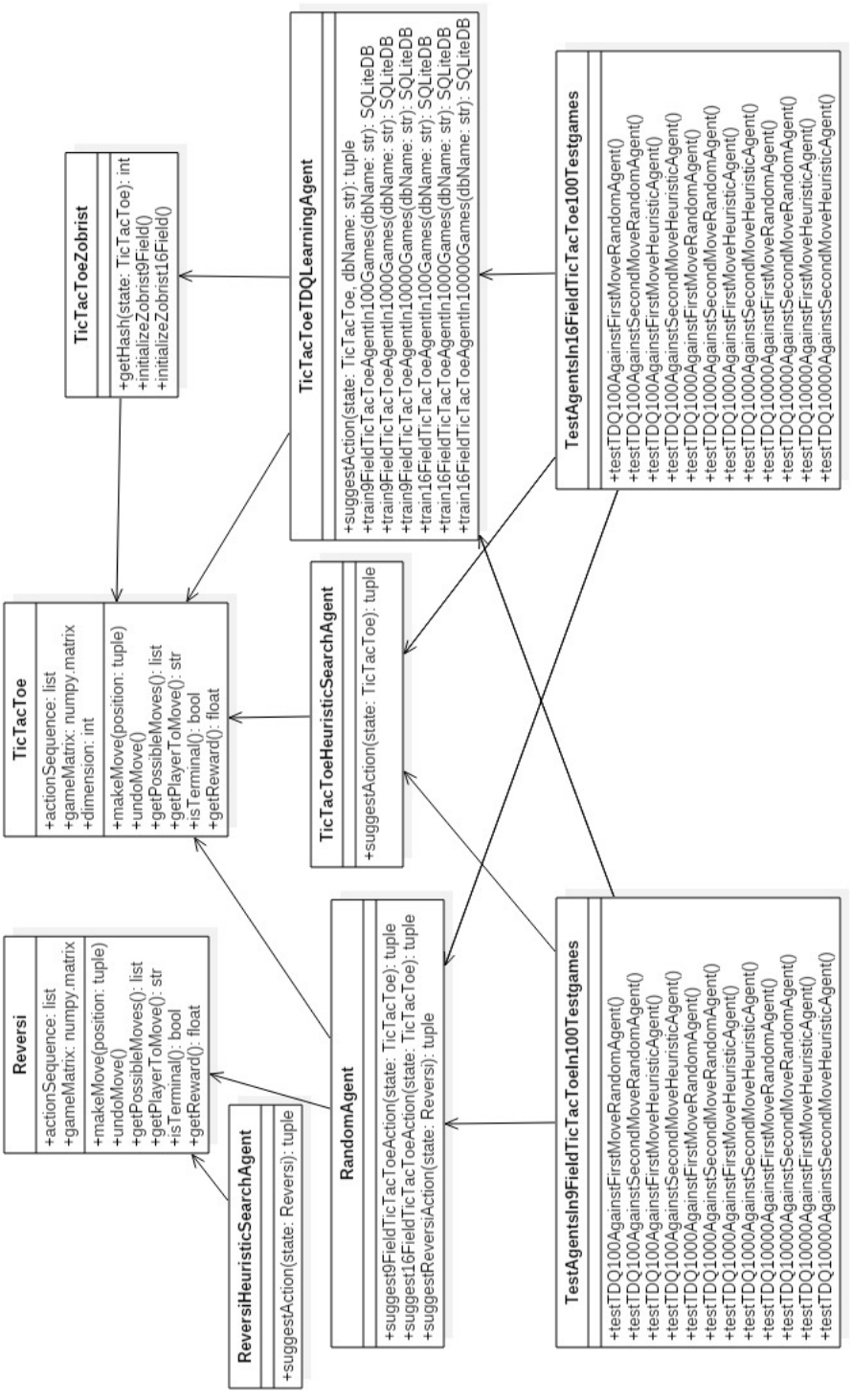


Abbildung 5.1 Klassendiagramm der Software

Kapitel 6

Algorithmen und Implementierung

Wir werden in diesem Kapitel den Algorithmus des Heuristik Agenten und des TD-Q-Agenten betrachten. Nachfolgend haben wir die wichtigsten Algorithmen für die Implementierung der Agenten dargestellt und erläutert. Die Gesamtheit aller für die Implementierung nötigen Algorithmen und der Tests findet man auf der beiliegenden Software-CD.

Für die Konvergenz des TD-Q-Lernens ist die Theorie des Erkundens und Verwertens (eng. exploration and exploitation) elementar wichtig. Am Ende des Kapitels werden wird dazu näheres erklären.

Weiterhin beschreiben wir am Ende des Kapitels, wie wir einen Spielzustand in einen einzigartigen Zahlenwert (Hash-Wert) umwandeln können, d.h. jede unterschiedliche Spielsituation wird durch einen anderen Zahlenwert referenziert. Wir benötigen diese einzigartige Referenzierung für das speichern und aktualisieren der Q-Werte in einer Datenbank.

Die von mir implementierten Algorithmen sind in der Programmiersprache Python realisiert. Python ist eine Hochsprache, die sehr leicht erlernbar ist und leicht von Menschen gelesen werden kann. Sie ähnelt einer Darstellung in Pseudocode. Python wird außerdem sehr gerne im Bereich des maschinellen Lernens eingesetzt. Wir verwenden in dieser Arbeit die Python Version 2.7.13.

6.1 Iterative-Alpha-Beta-Suche

Die Alpha-Beta-Suche und die iterativ vertiefende Tiefensuche des Abschnitts waren bereits Thema im Abschnitt 3. Es folgt eine Beschreibung der Implementierung dieser beiden Algorithmen.

Sehen wir uns das Codebeispiel aus Abbildung 6.1 genauer an. Der Eingabeparameter der Funktion ist ein Zustand (eine Spielsituation) der Spielumgebung. Basierend auf diesem Zustand expandiert die Funktion einen Suchbaum mit der maximalen Tiefe 2. Ziel der Funktion ist es, ein minimales oder ein maximales Ergebnis innerhalb der Suchtiefe 2 zu finden. Muss der Kreuzspieler seinen Spielzug ausführen, dann wird ein maximales Ergebnis gesucht und muss der Kreisspieler seinen Spielzug ausführen, wird ein minimales Ergebnis gesucht. Zurückgegeben wird ein entsprechendes Aktionstupel der Form (x,y) oder der Form (Koordinate 1, Koordinate 2).

Das Codebeispiel bezieht sich auf die Tic Tac Toe Implementierung der iterative vertiefenden Alpha-Beta-Suche. Die Reversi Implementierung ist fast identisch. Einzig der Vergleich in Zeile 8 ist unterschiedlich. Würde dieser Vergleich ausgelagert werden, wären die Algorithmen identisch, denn die Funktionalitäten der Strategiespielumgebungen sind sehr ähnlich definiert.

```
1  def alphaBetaIterativeDeepeningSearch(state):
2    listOfActionUtilities = []
3    actionList = actions(state)
4    for action in actionList:
5      state.makeMove(action)
6      listOfActionUtilities.append(maxValue(
         state, -sys.maxint, sys.maxint, 0, 2))
7      state.undoMove()
8    if state.getPlayerToMove() == 'X':
9      bestActionIndex = argmax(listOfActionUtilities)
10   else:
11     bestActionIndex = argmin(listOfActionUtilities)
12   return actionList[bestActionIndex]
```

Abbildung 6.1 Codeauszug der Iterativen-Alpha-Beta-Suche.

Die in der Funktion alphaBetaIterativeDeepeningSearch(state) verwendete Funktion maxValue(state, alpha, beta, depth, depthBound), realisiert die rekursive Exploration des Suchbaums (siehe Abbildung 6.2). Die Funktion liefert den Ergebniswert eines Spielzustands, dieser wird durch die Bewertungsfunktion evaluate(state) bestimmt. Die Rekursion entsteht durch den Aufruf der Funktion minValue(state, alpha, beta, depth +1, depthBound). Der Eingabeparameter "depth + 1" bedeutet eine Erhöhung der aktuellen Tiefe. Der Eingabeparameter "depthBound" speichert die maximale Tiefe, die von der aktuellen Tiefe nicht überschritten werden darf. Die beiden Funktionen minValue(...) und maxValue(...) unterscheiden sich nur in ihren letzten drei Codezeilen und in dem gegenseitigen Aufrufen der jeweils anderen Funktion. Die letzten drei Codezeilen von minValue(...) bewirken: Liefere Alpha zurück, wenn $Beta <= Alpha$ ist, andernfalls gib Beta zurück.

Die beiden Codebeispiele (Abbildung 6.1 und Abbildung 6.2) sind abgeleitet vom Alpha-Beta Algorithmus [RN12, S. 214 f.] und dem Algorithmus der iterativ vertiefenden Tiefensuche [RN12, S. 124].

```
1  def maxValue(state, alpha, beta, depth, depthBound):
2    if cutoffTest(state, depth, depthBound):
3      return evaluate(state)
4    for a in actions(state):
5      state.makeMove(a)
6      alpha = max(alpha, minValue(
         state, alpha, beta, depth + 1, depthBound))
7      state.undoMove()
8      if alpha >= beta:
9        return beta
10   return alpha
```

Abbildung 6.2 Iteratives Suchen des maximalen Ergebnisses.

6.2 TD-Q-Lernen

Das Thema in diesem Kapitel ist die Implementierung des TD-Q-Lernens und die Erläuterung des dafür benötigten Algorithmus.

```
1  def Q–Lernen(s', r', α, γ):
2    if istTerminalzustand(s):
3      Q[s, None] ← r'
4    if s ist nicht None:
5      inkrementiere N_sa[s, a]
6      Q[s, a] ← Q[s, a] + α(N_sa{s, a])
                 * (r + γ max_a' Q[s', a'] − Q[s, a])
7    s, a, r ← s', argmax_a', f(Q[s', a'], N_sa), r'
8    return a
```

Abbildung 6.3 Algorithmus des TD-Q-Lernens vgl. [RN12, S. 974]

Der Q-Lernen Algorithmus (Abbildung 6.3) verwendet einige persistente (d.h. beständige oder dauerhafte) Variablen. Persistent deshalb, weil sie die einzelnen Funktionsaufrufe bzw. Iterationen des Algorithmus überdauern:

- **Q** ist eine Tabelle mit Aktionswerten, indiziert nach Zustand und Aktion. Der Aufruf Q[s, a] liefert z.B. einen Aktionswert (Q-Wert) für eine Aktion a in einem Zustand s. Zu Beginn des Lernprozesses sind alle Werte dieser Tabelle leer. Die Abbildung von Zustand/Aktionspaaren auf Nutzenwerte wird als Q-Funktion bezeichnet. Für die Realisierung einer solchen Q-Tabelle bzw. Q-Funktion implementieren wir diverse SQLite Datenbank Funktionen, um diese Datenbankfunktionen zu verwirklichen benutzen wir benutzen wir das Python Paket ”sqlite3”. Die Datenbank Funktionen erstellen und aktualisieren Q-Werte, der Zugriff auf diese Q-Werte erfolgt wie bereits beschrieben durch Zustand/Aktionspaare.

- $\mathbf{N}_{sa}$ ist eine Tabelle mit Häufigkeiten für Zustand/Aktions-Paare. Diese ist wie Q anfangs leer. Jedes mal wenn ein Zustand/Aktions-Paar durchlaufen wird, welches bereits durchlaufen wurde, dann wird der Tabelleneintrag N_{sa}[s, a] inkrementiert d.h. um den Wert 1 erhöht. Anstatt diese Statistik über bereits besuchte Zustand/Aktionspaare zu führen, verwenden wir eine geeignete Explorationsfunktion f (später im aktuellen Abschnitt erklärt).

- **s** ist der vorhergehende Spielzustand (eine Instanz der Strategiespielumgebungen), anfangs leer. Berücksichtigen wir den zeitlichen Aspekt, dann wäre s zu einem Zeitpunkt t geschrieben s_t und ein darauffolgender Spielzustand wäre $s_t + 1$. Der direkt auf s folgende Spielzustand wird auch als s' (s Prime) bezeichnet.

- **a** ist die vorhergehende Aktion (ein Positionstupel der Spielmatrix), anfangs leer. Wird die Aktion a im Zustand s ausgeführt, dann wird der Zustand s' bzw. s_{t+1} erreicht. Eine Aktion die in s' ausgeführt werden kann, bezeichnen wir als a' oder a_{t+1}.

- **r** ist die Belohnung, die dem Agenten von der Umgebung zugeteilt wird, anfangs leer, wenn der Agent eine Aktion a in einem Zustand s ausführt (die Funktion der Strategiespielumgebung getReward() liefert diesen Wert). Wir können eine Funktion r(s, a) definieren. Die Funktion r(s, a) wird für die meisten Spielzustände $s \in S$ den Wert 0 liefern. Für Endzustände der jeweiligen Strategiespiele wird die Funktion r(s, a) andere Werte liefern. Ist r die Belohnung dafür, die Aktion a in Zustand s auszuführen, dann ist r' (r Prime) die Belohnung dafür, Nachfolgeaktion a' in Nachfolgezustand s' auszuführen.

Der Q-Lernen Algorithmus (Abbildung 6.3) bekommt folgende Eingabeparameter übergeben:

- **s'** ist der aktuelle Spielzustand und gleichzusetzen mit der aktuellen Wahrnehmung des Agenten. Wie bereits erklärt ist s' der Nachfolgezustand von s.

- **r'** ist das Belohnungssignal, welches der Agent erhält, wenn er eine Aktion a' im Zustand s' ausführt.

- **α** bestimmt über die Lernrate des Algorithmus. Der Wert von α ist in der Regel zwischen 0 und 1. Eine hohe Lernrate (α nahe 1) bedeutet, dass die Aktualisierung des Q-Werts stärker ist. Bei einer niedrigen Lernrate ist die Aktualisierung schwächer. Der Ausdruck $\alpha(N_{sa}[s, a])$ im TD-Q-Lernen Algorithmus bedeutet: Aktualisiere Q-Werte für neue noch unbekannte Zustand/Aktionspaare mehr (wenig Vertrauen in den Q-Wert) und aktualisiere den Q-Werten von bereits öfter besuchten Zustand/Aktionspaaren weniger (mehr Vertrauen in den Q-Wert).

- γ ist der Abschwächungsfaktor (eng. discounting factor). Im fachlichen Umfeld des verstärkenden Lernens wird dieser Abschwächungsfaktor bei Modellen mit unendlichen Horizont verwendet. Endet eine Aktionssequenz in einem Markov-Entscheidungsprozess nicht, dann ist diese unendlich. Um Probleme dieser Klasse trotzdem handhaben zu können, wird für die Berechnung des erwarteten Nutzens $U^{\pi}(s)$ eines Zustands s der Abschwächungsfaktor verwendet. Da sowohl Tic Tac Toe als auch Reversi, nach einer maximalen Anzahl von Aktionen, immer in einem Endzustand terminieren, werden wir den Abschwächungsfaktor gleich 1 setzen. Ein Abschwächungsfaktor von 1 bedeutet, dass Belohnungen in der Zukunft genau so wertvoll sind, wie unmittelbare Belohnungen.

Erkunden und Verwenden

Russell und Norvig schreiben sinngemäß vgl. [RN12, S. 974]: Die Statistik N_{sa} kann weggelassen werden, wenn eine angemessene Explorationsstrategie f verwendet wird. Mit einer angemessenen Explorationsstrateige meinen sie, ein zufälliges Agieren, für einen bestimmten Anteil an Schritten, wobei dieser Anteil mit der Zeit geringer wird.

Dies ist auch die Empfehlung von Wolfgang Ertel [Ert13, S. 303]: "Es empfiehlt sich eine Kombination aus Erkunden und Verwerten mit einem hohen Erkundungsanteil am Anfang, der dann im Laufe der Zeit immer weiter reduziert wird."

Wir benötigen demnach eine Explorationsfunktion f(state), die einen Zustand als Eingabeparameter bekommt und eine, von diesem Spielzustand aus, mögliche Aktion zurück liefert.

Die Funktion explorationStrategy(state, randomFactor) aus Abbildung 6.4 realisiert eine solche Explorationsstrategie. Sie berechnet eine auszuführende Aktion. Die Aktion wird zu einer bestimmten Wahrscheinlichkeit zufällig ausgewählt. Diese Wahrscheinlichkeit wird durch zwei Faktoren beeinflusst.

Der erste Faktor ist die Anzahl der bereits durchgeführten Trainingsspiele des TD-Q Agenten. Dieser Faktor wird durch den Eingabeparameter "randomFactor" dargestellt. Alle 100 Trianingsspiele wird dieser Faktor um 1 erhöht, d.h. alle 100 Trainingsspiele sinkt die Wahrscheinlichkeit eine zufällige Aktion auszuwählen.

```
1  def explorationStrategy(state, randomFactor):
2    if not state.isTerminal():
3      depth = state.countOfGameTokensInGame()
4      if randint(0, randomFactor * depth) ==
           randint(0, randomFactor * depth) and
           depth < (2 * state.dimension()):
5        moves = state.getPossibleMoves()
6        return moves[randint(0, (len(moves) - 1)]
7      else:
8        return suggestAction(state)
9    else:
10     return None
```

Abbildung 6.4 Die implementierte Explorationsstrategie.

Der zweite Faktor der die Wahrscheinlichkeit beeinflusst, ist die aktuelle Tiefe des Spielbaums. Die aktuelle Tiefe des Spielbaums wird von der Funktion countOfGameTokensInGame() ermittelt, denn die Tiefe des Spielbaums ist gleichzusetzen mit den bereits gesetzten Spielfiguren. Für Knoten des Spielbaums die sich näher am Wurzelknoten befinden, wird mit einer größeren Wahrscheinlichkeit, eine zufällige Aktion ausgewählt. Sollte der Fall eintreten, dass keine Zufällige Aktion ausgewählt werden soll, dann wird die Funktion auggestAction(state) aufgerufen. Diese Funktion liefert die Aktion mit dem maximalen Q-Wert zurück (die von Wolfgang Ertel beschriebene "Ausnutzung"). Ist der übergebene Spielzustand bereits ein Endzustand, dann wird der Wert "None" zurück gegeben.

Zobrist Hash

"Wenn ein Computerprogramm einen Gegenstand in einer großen Tabelle speichert, muss die Tabelle zwangsläufig durchsucht werden, um den Gegenstand wiederzuverwenden bzw. zu referenzieren. Dies gilt solange, bis eine Tabellendresse aus dem Gegenstand selbst, in systematischer weise, berechnet werden kann. Eine Funktion die Gegenstände in Adressen umwandelt ist ein Hash-Algorithmus, und die daraus resultierende Tabelle ist eine Hashtabelle [Zob70, S. 3]."

Zobrist-Hashing ermöglicht es, Spielzustände eindeutig als Zahlenwerte zu definieren. Berechnen wir den Zobrist-Hash eines Spielzustandes, dann ist dieser immer gleich, selbst wenn der Spielzustand durch verschiedene Aktionssequenzen repräsentiert werden kann. Das Zobrist-Hashverfahren ist sehr wichtig für unseren TD-Q-Agenten, weil die Spielsituationen, als Zobrist-Hash, in der Q-Wertetabelle eingetragen werden können.

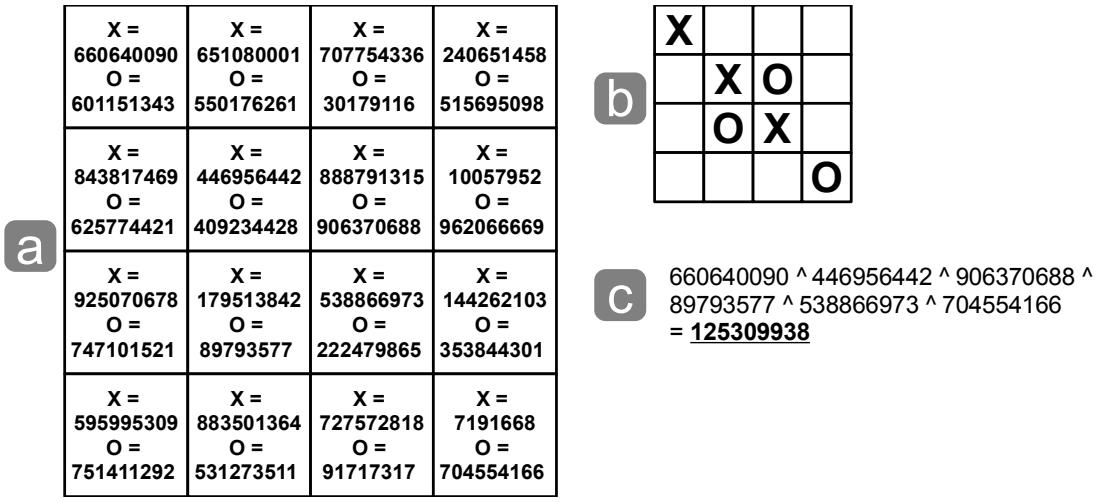


Abbildung 6.5 Zobrist Hashing von Spielzuständen.

In Abbildung 6.5 wird das Zobrist Hash Verfahren auf einen bestimmten Spielzustand (b) angewendet. Schritt 1: (a) wir weisen jedem Spielfeld zwei zufällige ganzzahlige Werte zu, im Bereich von 0 bis maximal 1×10^9. Einen zufälligen Wert für den Kreuzspielstein an dieser Position und einen für den Kreisspielstein. Das 4x4 Tic Tac Toe Spielbrett sollte insgesamt 32 verschiedene Werte erhalten. Spielsituation (b) soll in einen Zobrist-Hash umgewandelt werden.

Der Zobrist-Hash berechnet sich wie folgt (c), ist die aktuelle Position mit einem Kreuzspielstein oder einem Kreisspielstein besetzt, dann wähle den entsprechenden Wert aus der Werttabelle (a). Dies wiederhole für jedes besetzte Spielfeld. Wir verknüpfen die bestimmten Werte, mittels eines exklusiven bzw. bitweisen Oder (XOR). Das Ergebnis ist eine Adresse, die exakt den Spielzustand (b) referenziert.

Kapitel 7

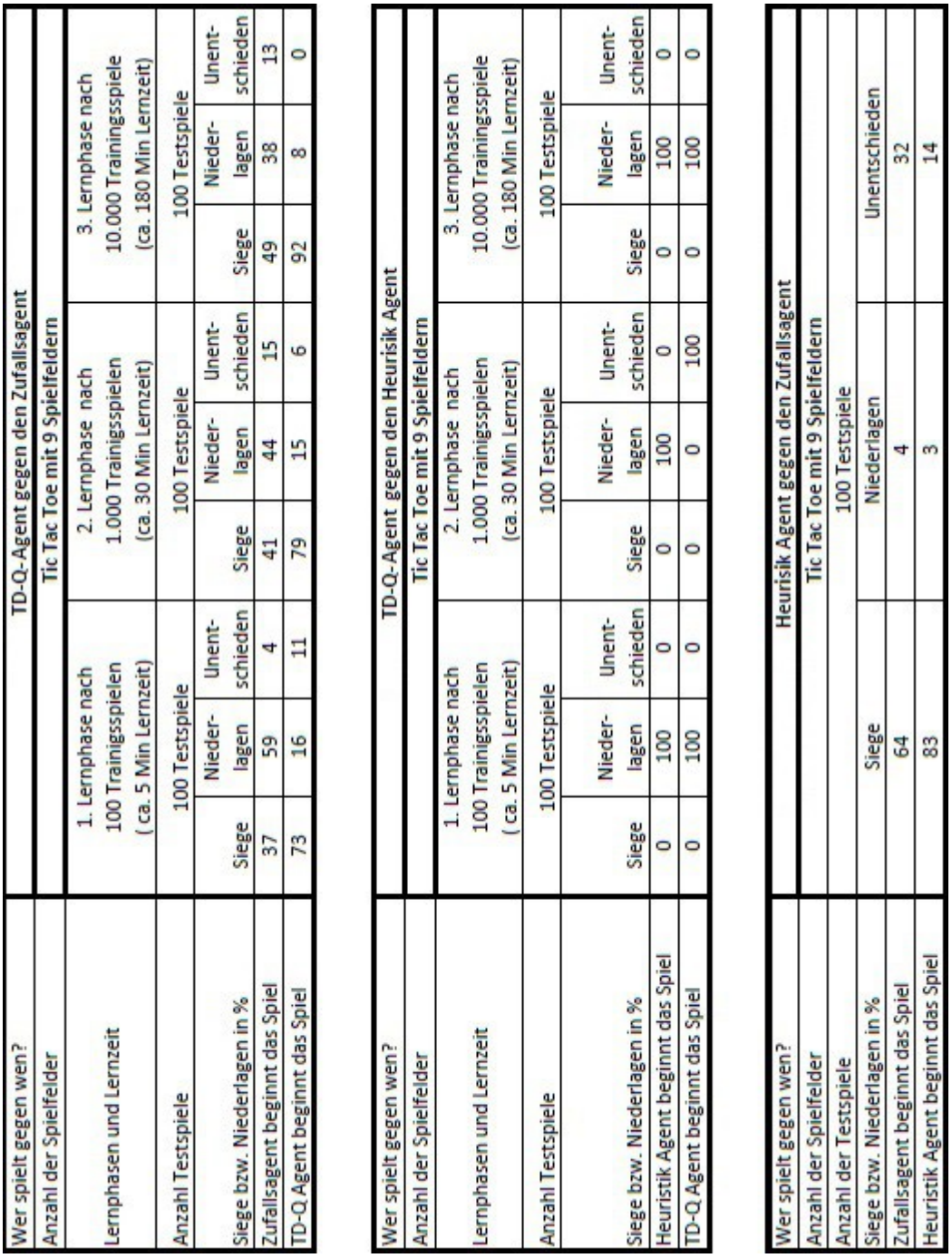
Testergebnisse (Validierung)

In diesem Kapitel testen wir die Lernfähigkeit und die Lerndauer des TD-Q Agenten, in der Strategiespielumgebung 9 und 16 Spielfelder Tic Tac Toe. Wir testen ebenso die Spielstärke des vorausschauenden Tic Tac Toe Heuristik Agenten. Der TD-Q-Agent und der Heuristik Agent spielen genau 100 Testspiele gegeneinander und 100 Testspiele gegen einen Zufallsagenten.

Der TD-Q Agent wird 3 verschiedene Lernphasen ausführen. Die Lernphasen unterscheiden sich in der Anzahl der Trainingsspiele. Untersucht werden vom TD-Q Agenten gelernte Strategien, die in 100 Trainingsspielen (Lernphase 1), in 1.000 Trainingsspielen (Lernphase 2) und in 10.000 Trainingsspielen (Lernphase 3), gegen sich selbst trainiert werden. Jede dieser Lernphasen wird für das 9 Spielfelder Tic Tac Toe und das 16 Spielfelder Tic Tac Toe ausgeführt.

Im nächsten Kapitel 8 werden wir erklären warum wir den TD-Q-Agenten keine Strategie für Reversi lernen lassen. Die Ergebnisse der Tests für das 9 Spielfelder Tic Tac Toe und das 16 Spielfelder Tic Tac Toe werden diese Erklärung belegen.

Die für das Testen der Agenten implementierten Strategiespielumgebungen wurden mit dem Python Paket ”unittest” getestet. Die Tests der Tic Tac Toe Strategiespielumgebung befinden sich in der Datei ”TestTicTacToe.py” und die Tests der Reversi Strategiespielumgebung befinden sich in der Datei ”TestReversi.py”. In diesen Dateien sind die wichtigsten Funktionalitäten der Strategiespielumgebungen ausgeführt und mit ”asserts” getestet. Alle Tests der Strategiespielumgebungen waren positiv und könnten bei Bedarf wiederholt werden.

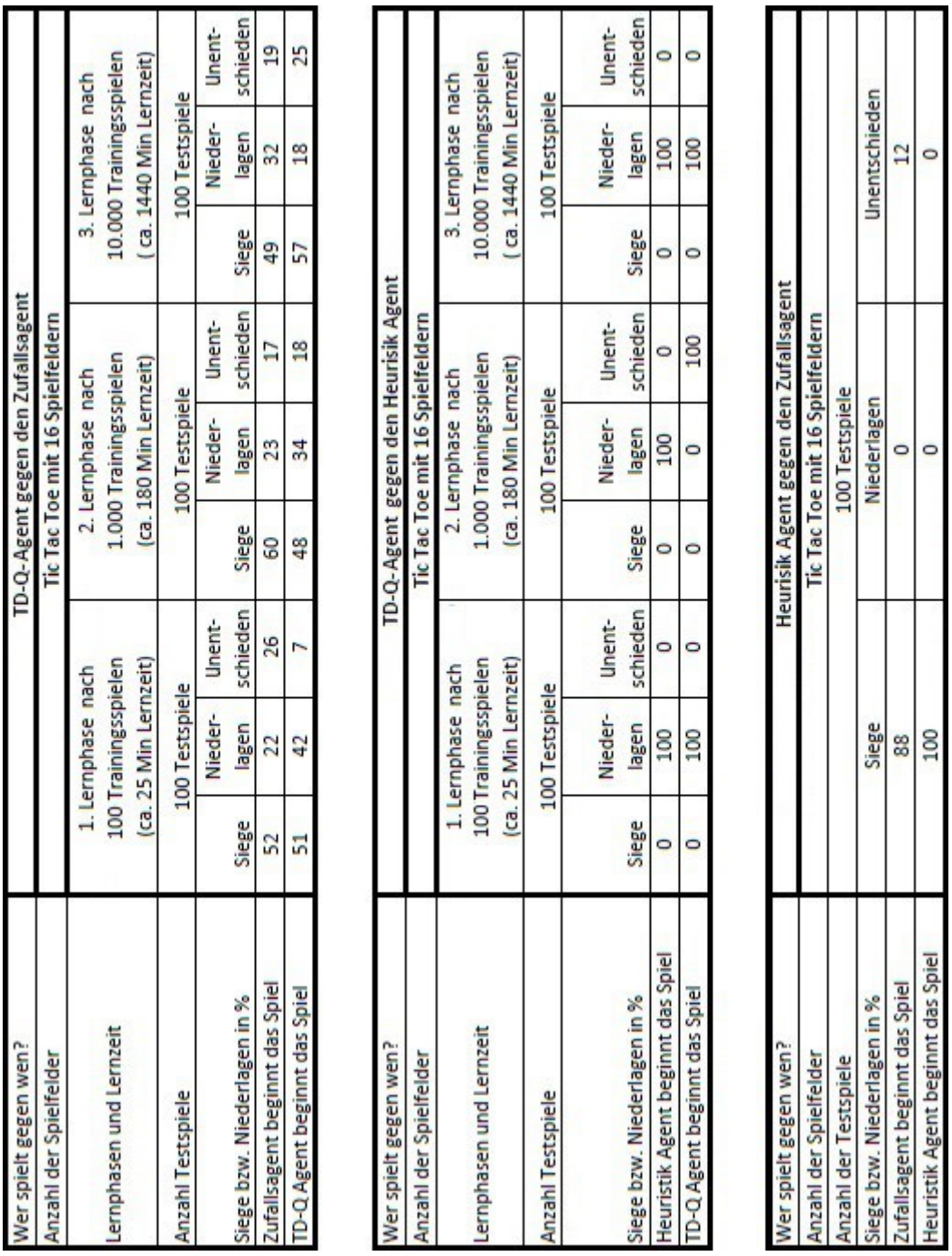


Wer spielt gegen wen?	TD-Q-Agent gegen den Zufallsagent									
Anzahl der Spielfelder	Tic Tac Toe mit 9 Spielfeldern									
Lernphasen und Lernzeit	1. Lernphase nach 100 Trainigsspielen (ca. 5 Min Lernzeit)			2. Lernphase nach 1.000 Trainigsspielen (ca. 30 Min Lernzeit)			3. Lernphase nach 10.000 Trainingsspiele (ca. 180 Min Lernzeit)			
Anzahl Testspiele	100 Testspiele			100 Testspiele			100 Testspiele			
Siege bzw. Niederlagen in %	Siege	Nieder-lagen	Unent-schieden	Siege	Nieder-lagen	Unent-schieden	Siege	Nieder-lagen	Unent-schieden	
Zufallsagent beginnt das Spiel	37	59	4	41	44	15	49	38	13	
TD-Q Agent beginnt das Spiel	73	16	11	79	15	6	92	8	0	

Wer spielt gegen wen?	TD-Q-Agent gegen den Heurisik Agent									
Anzahl der Spielfelder	Tic Tac Toe mit 9 Spielfeldern									
Lernphasen und Lernzeit	1. Lernphase nach 100 Trainigsspielen (ca. 5 Min Lernzeit)			2. Lernphase nach 1.000 Trainigsspielen (ca. 30 Min Lernzeit)			3. Lernphase nach 10.000 Trainingsspiele (ca. 180 Min Lernzeit)			
Anzahl Testspiele	100 Testspiele			100 Testspiele			100 Testspiele			
Siege bzw. Niederlagen in %	Siege	Nieder-lagen	Unent-schieden	Siege	Nieder-lagen	Unent-schieden	Siege	Nieder-lagen	Unent-schieden	
Heuristik Agent beginnt das Spiel	0	100	0	0	100	0	0	100	0	
TD-Q Agent beginnt das Spiel	0	100	0	0	0	100	0	100	0	

Wer spielt gegen wen?	Heurisik Agent gegen den Zufallsagent		
Anzahl der Spielfelder	Tic Tac Toe mit 9 Spielfeldern		
Anzahl der Testspiele	100 Testspiele		
Siege bzw. Niederlagen in %	Siege	Niederlagen	Unentschieden
Zufallsagent beginnt das Spiel	64	4	32
Heuristik Agent beginnt das Spiel	83	3	14

Abbildung 7.1 Die Testergebnisse für das 9 Spielfelder Tic Tac Toe.



Wer spielt gegen wen?	TD-Q-Agent gegen den Zufallsagent									
Anzahl der Spielfelder	Tic Tac Toe mit 16 Spielfeldern									
Lernphasen und Lernzeit	1. Lernphase nach 100 Trainingsspielen (ca. 25 Min Lernzeit)			2. Lernphase nach 1.000 Trainingsspielen (ca. 180 Min Lernzeit)			3. Lernphase nach 10.000 Trainingsspielen (ca. 1440 Min Lernzeit)			
Anzahl Testspiele	100 Testspiele			100 Testspiele			100 Testspiele			
Siege bzw. Niederlagen in %	Siege	Nieder-lagen	Unent-schieden	Siege	Nieder-lagen	Unent-schieden	Siege	Nieder-lagen	Unent-schieden	
Zufallsagent beginnt das Spiel	52	22	26	60	23	17	49	32	19	
TD-Q Agent beginnt das Spiel	51	42	7	48	34	18	57	18	25	

Wer spielt gegen wen?	TD-Q-Agent gegen den Heurisik Agent									
Anzahl der Spielfelder	Tic Tac Toe mit 16 Spielfeldern									
Lernphasen und Lernzeit	1. Lernphase nach 100 Trainingsspielen (ca. 25 Min Lernzeit)			2. Lernphase nach 1.000 Trainingsspielen (ca. 180 Min Lernzeit)			3. Lernphase nach 10.000 Trainingsspielen (ca. 1440 Min Lernzeit)			
Anzahl Testspiele	100 Testspiele			100 Testspiele			100 Testspiele			
Siege bzw. Niederlagen in %	Siege	Nieder-lagen	Unent-schieden	Siege	Nieder-lagen	Unent-schieden	Siege	Nieder-lagen	Unent-schieden	
Heuristik Agent beginnt das Spiel	0	100	0	0	100	0	0	100	0	
TD-Q Agent beginnt das Spiel	0	100	0	0	0	100	0	100	0	

Wer spielt gegen wen?	Heurisik Agent gegen den Zufallsagent		
Anzahl der Spielfelder	Tic Tac Toe mit 16 Spielfeldern		
Anzahl der Testspiele	100 Testspiele		
Siege bzw. Niederlagen in %	Siege	Niederlagen	Unentschieden
Zufallsagent beginnt das Spiel	88	0	12
Heuristik Agent beginnt das Spiel	100	0	0

Abbildung 7.2 Die Testergebnisse für das 16 Spielfelder Tic Tac Toe.

Kapitel 8

Auswertung

In diesem Kapitel wollen wir die Testergebnisse des vorherigen Kapitels 7 auswerten. Die Ergebnisse der Testphase sollen uns dabei helfen die Leistungsfähigkeit und die Grenzen des TD-Q-Lernens zu beurteilen. Wir werden ein, in der Literatur bereits bekanntes Problem dieses Lernverfahrens mit den Testergebnissen belegen und Lösungsmöglichkeiten besprechen. Diese Lösungsmöglichkeiten sind aus der Literatur entnommen und sie stellen herausragende Erfolge auf dem Gebiet des verstärkenden Lernens dar.

8.1 TD-Q-Lernen - Leistung und Grenzen

Die Testergebnisse ermöglichen uns, über die Konvergenz und über die benötigte Rechenzeit der gelernten Strategien, in den verschiedenen Testphasen, Aussagen zu treffen. Die Konvergenz bezieht sich auf eine Annäherung der vom TD-Q-Agenten gelernten Strategie an eine unbekannte optimale Strategie.

Wolfgang Ertel schreibt über die allgemeine Konvergenz des Q-Lernens (TD-Q-Lernens) (vgl. [Ert13, S. 299]):

Das Q-Lernen konvergiert für ein konkretes Beispiel und allgemein, zu einer optimalen Strategie, wenn jedes Zustands-Aktions-Paar unendlich oft besucht wird. Konkret konvergiert der Wert $\hat{Q}_n(s,a)$ für alle Werte von s und a gegen $Q(s,a)$ für $n \to \infty$, mit n gleich der Anzahl der Aktualisierungen des Q-Werts ($\hat{Q}_n(s,a)$).

8.1.1 TD-Q-Lernen Leistungsfähigkeit (Konvergenz)

In diesem Abschnitt analysieren wir die Testergebnisse der einzelnen Lernphasen, für 9 und 16 Spielfelder Tic Tac Toe. Wir beurteilen die Leistung der einzelnen gelernten Strategien, aufgrund der Testspiele gegen den Zufallsagenten und den Heuristik Agenten. Wir versuchen, die oben zitierte Aussage von Wolfgang Ertel, über die Konvergenz, mit unseren Testergebnissen zu untermauern.

Tic Tac Toe - Lernen gegen Zufall - 9 Spielfelder

Zufallsagent beginnt das Spiel:
Die in 100 Trainingsspielen gelernte Strategie des TD-Q-Agenten hat schlechtere Gewinnquoten (37 Siege), als die in 1.000 Trainingsspielen gelernte Strategie. Die in 10.000 Trainingsspielen gelernte Strategie, hat die besten Gewinnquoten (49 Siege). Die Steigerung der Gewinnquoten der einzelnen Lernphasen ist hier noch recht klein.

TD-Q-Agent beginnt das Spiel:
Die in 100 Trainingsspielen gelernte Strategie des TD-Q-Agenten hat schlechtere Gewinnquoten (73 Siege), als die in 1.000 Trainingsspielen gelernte Strategie. Die in 10.000 Trainingsspielen gelernte Strategie, hat die besten Gewinnquoten (92 Siege). Wir können aus den Testergebnissen ableiten, dass die Leistungsfähigkeit ansteigt, je mehr Trainingsspiele der TD-Q-Agent durchführen konnte. Die Testergebnisse der Lernphasen zeigen sehr gut die Auswirkungen, die die Anzahl der Trainingsspiele auf die Konvergenz der gelernten Strategie hat.

Obwohl es praktisch nicht möglich ist, unendlich viele Trainingsspiele durchzuführen, konnten wir zeigen, dass die gelernten Strategien eher zu einer optimalen Strategie konvergieren, je mehr Trainingsspiele durchgeführt werden. Die oben zitierte Aussage von Wolfgang Ertel, über die Konvergenz, trifft für diese Testergebnisse demnach annähern zu.

Tic Tac Toe - Lernen gegen Zufall - 16 Spielfelder

Beim 16 Spielfelder Tic Tac Toe sind die Aktions- und Zustandsräume wesentlich umfangreicher, als beim 9 Spielfelder Tic Tac Toe. Den Testergebnissen können wir entnehmen, dass beim 16 Spielfelder Tic Tac TOe keine eindeutige Steigerung der Gewinnquote mit ansteigender Trainingsspielanzahl erkennbar ist, egal welcher der beiden Agenten das Spiel beginnt. Auf Grund der erhöhten Komplexität, sind

mit hoher Wahrscheinlichkeit wesentlich mehr Trainingsspiele erforderlich, um auch hier eine annähernde Konvergenz der gelernten Strategie zu erreichen.
In Abschnitt 8.1.2 klären wir noch ausführlich, warum die Testergebnisse der einzelnen Lernphasen, für das 16 Spielfelder Tic Tac Toe, keine eindeutige Leistungssteigerung aufzeigen, obwohl die Trainingsspiele in jeder Lernphase um Faktor 10 erhöht wurden.

Tic Tac Toe - Lernen gegen Heuristik - 9 und 16 Spielfelder

Die Testspiele der gelernten TD-Q-Strategien gegen den vorausschauenden Heuristik Agent sind insofern eindeutig ausgefallen, dass der Heuristik Agent bis auf 2 gelernte Strategien, immer die bessere Strategie hatte und gewinnt. In zwei Lernphasen geht das Spiel unentschieden aus.

In jeder der 6 Lernphasen (3 Lernphasen 9 Spielfelder und 3 Lernphasen 16 Spielfelder) wird eine unterschiedliche Strategie gelernt. In den Lernphasen wurden 6 verschiedene Strategien gelernt.

In den einzelnen Testphasen haben die beiden Agenten, je nachdem wer beginnt, in den jeweils 100 Testspielen das gleiche Spiel gespielt. Der Grund dafür ist, nach Abschluss der Lernphasen entstehen feste nicht veränderbare Strategien. Diese entscheiden für den selben Ausgangszustand immer mit der selben Aktion, das Gleiche gilt auch für den Heuristik Agenten. Insgesamt wurden in den 1.200 Testspielen nur 12 unterschiedliche Spiele gespielt. Eine Erkenntnis der Auswertung der Testergebnisse ist deshalb auch, dass es ausreichend gewesen wäre, wenn in jeder Testphase maximal 2 Testspiele durchgeführt worden wären. Einmal hätte der Heuristik Agent das Spiel begonnen und einmal der TD-Q-Agent.

Folglich ist die 2. Hypothese aus dem Abschnitt 1.4 bestätigt.

Tic Tac Toe - Heuristik gegen Zufall - 9 und 16 Spielfelder

Die Testergebnisse des vorausschauenden Heuristik Agenten gegen den Zufallsagenten sind eindeutig. Der kleinste Erfolg des Heuristik Agenten, mit 64% Gewinnquote bestätigt bereits die 1. Hypothese aus Abschnitt 1.4. Die größte Gewinnquote des von uns implementierten Tic Tac Toe vorausschauenden Heuristik Agenten ist 100%, d.h. von 100 Testspielen in einem 16 Spielfelder Tic Tac Toe, wenn der Heuristik Agent das Spiel beginnt, gewinnt der Heuristik Agent alle 100 Testspiele.

8.1.2 TD-Q-Lernen Grenzen (Fluch der Dimensionalität)

"Trotz der Erfolge in den letzten Jahren bleibt das Lernen durch Verstärkung ein sehr attraktives Forschungsgebiet der KI, nicht zuletzt deshalb, weil auch die besten heute bekannten Lernalgorithmen bei hochdimensionalen Zustands- und Aktionsräumen wegen ihrer gigantischen Rechenzeit immer noch nicht praktisch anwendbar sind. [Ert13, S. 305]"

Wolfgang Ertel beschreibt in diesem Zitat den Fluch der Dimensionalität, als großes Problem der heutigen Lernalgorithmen. Auch Russell und Norvig erwähnen dieses Problem:

"Bisher sind wir davon ausgegangen, dass die Nutzenfunktionen und die von den Agenten gelernten Q-Funktionen in tabellarischer Form mit einem Ausgabewert für jedes Eingabetupel vorliegen. Ein solcher Ansatz funktioniert ausreichend gut für kleine Zustandsräume, aber die Zeit bis zur Konvergenz und (für ADP) die Zeit pro Iteration steigt mit wachsendem Raum rapide an. [RN12, S. 975]"

Stellen wir uns einen Suchbaum vor (ähnlich der Abbildung 3.1 aus dem Abschnitt Minimax-Suche), der alle Aktionen in jedem Zustand eines 9 Spielfelder Tic Tac Toe Spiels abbildet. Das Spielbrett eines 9 Spielfelder Tic Tac Toe's ist 3 mal 3 Spielfelder groß, die Dimension des 9 Spielfelder Tic Tac Toe's ist also gleich 3. Der Verzweigungsfaktor des Suchbaums ist direkt nach dem Wurzelknoten gleich 9. Erhöhen wir die Tiefe des Suchbaums um 1, dann verringert sich der Verzweigungsfaktor ebenfalls um 1 (gilt für Reversi und Tic Tac Toe). Der Verzweigungsfaktor symbolisiert den Aktionsraum, der in einem bestimmten Spielzustand vorhanden ist. Jeder Knoten (auch Blattknoten und Wurzelknoten) ist ein möglicher Spielzustand. Die Gesamtheit aller Spielzustände bildet den Zustandsraum ab.

Wenn wir die redundanten Spielzustände nicht ausschließen und den Fakt nicht beachten, dass nicht jeder Baumpfad eine maximale Länge von 9 hat (9 Spielfelder Tic Tac Toe kann schon in einer Tiefe von 5 terminieren), dann hat der Spielbaum geschätzt 362.880 Spielzustände ($9 * 8 * 7 * 6 * 5 * 4 * 3 * 2$ entspricht 9 Fakultät!). Erhöhen wir die Dimension des Tic Tac Toe Spiels auf 4, dann erhalten wir das 16 Spielfelder Tic Tac Toe. Berechnen wir die Spielzustände für das 16 Spielfelder Tic Tac Toe unter den gleichen Voraussetzungen wie bei dem 9 Spielfelder Tic Tac Toe, dann erhalten wir ungefähr 20.922.789.888.000 Spielzustände. Die Anzahl der Berechneten Spielzustände entspricht einer groben Schätzung und soll zur Veranschaulichung der Dimensionen dienen.

Diese Anzahl der Spielzustände entsprechen nicht der tatsächlichen Anzahl, sie sollen hier nur als grobe Schätzung und zur Veranschaulichung dienen (keine Garantie der Korrektheit der Werte).

Wir können aus diesen beiden Spielzustandsschätzungen schlussfolgern, dass eine Erhöhung der Dimension einen enormen Effekt auf die Vergrößerung des Zustands- und Aktionsraums hat. Vermutlich meint Wolfgang Ertel genau dieses Verhältnis, wenn er von Fluch der Dimensionalität spricht.

Rechenzeit der Lernphasen

Wir können die in der Testphase ermittelten Rechenzeitwerte der einzelnen Lernphasen verwenden, um den Fluch der Dimensionalität aufzuzeigen.

Rechenzeit des TD-Q-Lernens für 9 Spielfelder Tic Tac Toe:

- 100 Trainingsspiele ungefähr 5 Minuten
- 1.000 Trainingsspiele ungefähr 25 Minuten
- 10.000 Trainingsspiele ungefähr 180 Minuten

Rechenzeit des TD-Q-Lernens für 16 Spielfelder Tic Tac Toe:

- 100 Trainingsspiele ungefähr 25 Minuten
- 1.000 Trainingssiele ungefähr 180 Minuten
- 10.000 Trainingsspiele ungefähr 1440 Minuten

Die einzelnen Testergebnisse zeigen deutlich eine Rechenzeiterhöhung der Lernphasen des 16 Spielfelder Tic Tac Toe's im Gegensatz zum 9 Spielfelder Tic Tac Toe, d.h. mit steigender Dimension erhöht sich die Rechenzeit für das TD-Q-Lernen in unserem Beispiel bereits um 800%. Wir schlussfolgern daher: 10.000 Trainingsspiele für das Strategiespiel Reversi, würden mehrere Monate oder Jahre in Anspruch nehmen, bei Verwendung einer tabellarischen Darstellung der Q-Funktion.

Die bisherigen Ergebnisse dieser Arbeit bestätigen somit die nachfolgende Aussagen von Wolfgang Ertel:

"Die weltbesten Schachcomputer arbeiten bis heute immer noch ohne Lernverfahren. Dafür gibt es zwei Gründe. Einerseits benötigen die bis heute entwickelten Verfahren zum Lernen durch Verstärkung bei großen Zustandsräumen noch sehr viel

Rechenzeit. Andererseits sind aber die manuell erstellten Heuristiken der Hochleistungsschachcomputer schon sehr stark optimiert. Das heißt, dass nur ein sehr gutes Lernverfahren noch zu Verbesserungen führen kann. [Ert13, S. 120]"

8.2 Lösungen für das Dimensionalitätsproblem

Im letzten Abschnitt dieser Arbeit werden wir auf die sehr guten Lernverfahren eingehen, die Wolfgang Ertel in seiner Aussage zuvor erwähnt hat. Die beiden vielleicht bisher erfolgreichsten lernenden Programme. Das Dame-Spiel von Arthur L. Samuel aus dem Jahre 1955 und TD-Gammon von Gerald Tesauro aus dem Jahre 1992.

8.2.1 Samuels-Dame-Spiel

"Arthur L. Samuel schrieb 1955 ein Programm, dass Dame spielen konnte und mit einem einfachen Lernverfahren seine Parameter verbessern konnte. Sein Programm hatte dabei jedoch Zugriff auf eine große Zahl von archivierten Spielen, bei denen jeder einzelne Zug von Experten bewertet war (Überwachtes Lernen zur Unterstützung des verstärkenden Lernens). Damit verbesserte das Programm seine Bewertungsfunktion. Um eine noch weitere Verbesserung zu erreichen, ließ Samuel sein Programm gegen sich selbst spielen. Das Credit Assignment löste er auf einfache Weise. Für jede einzelne Stellung während eines Spiels vergleicht er die Bewertung durch die Funktion B(s) mit der durch Alpha-Beta-Pruning berechneten Bewertung und verändert B(s) entsprechend. 1961 besiegte sein Dame-Programm den viertbesten Damespieler der USA. Mit dieser bahnbrechenden Arbeit war Samuel seiner Zeit um fast dreißig Jahre voraus. [Ert13, S. 120 f.]"

Wir werden nachfolgend den Unterschied zwischen dem in dieser Arbeit implementierten Lernverfahren und dem von Samuel implementierten Lernverfahren erläutern (vgl. [RN12, S. 976]):

Russell und Norvig beschreiben das Lernverfahren von Samuel als **Funktionsannäherung**. Der Unterschied zwischen Samuels Lernalgorithmus und dem von uns implementierten TD-Q-Algorithmus ist, dass Samuel eine andere Darstellung als eine Suchtabelle (die Tabelle die wir mit Q(s,a) gekennzeichnet hatten) für die Q-Funktion verwendet.

Arthur Samuels Dame Spiel verwendete also eine parametrisierte Bewertungsfunktion in der Form:

$$\hat{U}_\theta(s) = \theta_1 f_1(s) + \theta_2 f_2(s) + ... + \theta_n f_n(s),$$

um die Q-Funktion darzustellen. Samuel verwendet für das Lernen der einzelnen Parameter ($\theta = \theta_1, ...\theta_n$) der Bewertungsfunktion $\hat{U}_\theta(s)$ eine abgewandelte Form, des in dieser Arbeit implementierten TD-Q-Lernens (vgl. [RN12, S. 981]). Dieses Vorgehen ermöglichte es Samuels Lernalgorithmus die Q-Funktion, statt in einer 10^{40} Tabelle, durch etwa $n = 20$ Parameter zu charakterisieren. Dies ist gegenüber der Tabelle (unsere Darstellung der Q-Funktion) die unser TD-Q Agent in dieser Arbeit lernt, eine riesige Komprimierung.

8.2.2 TD-Gammon

"Das TD-Lernen zusammen mit einem Backpropagation-Netz mit 40 bis 80 verdeckten Neuronen wurde sehr erfolgreich angewendet in TD-Gammon, einem Programm zum Spielen von Backgammon, programmiert vom Entwickler Gerald Tesauro im Jahr 1992. Die einzige direkte Belohnung für das Programm ist das Ergebnis am Ende eines Spiels. Eine optimierte Version des Programms mit einer 2-Züge-Vorausschau wurde mit 1,5 Millionen Spielen gegen sich selbst trainiert. Es besiegte damit Weltklassespieler und spielt so gut wie die drei besten menschlichen Spieler. [Ert13, S. 304]"
Aus dem oberen Zitat von Wolfgang Ertel und den Ausführungen von Russell und Norvig [RN12, S. 982] zum Thema TD-Gammon können wir folgendes entnehmen:

Gerald Tesauro gelang es demnach ein verstärkendes Lernverfahren zu entwickeln, welches tatsächlich fähig war, menschliche Weltklassespieler zu besiegen. Sein Lernverfahren war dem von Samuel implementierten Lernverfahren sehr ähnlich, denn er verwendete ebenfalls das TD-Lernen, eine Funktionsannäherung und einen Selbsttrainingsmodus. In einem Selbstspielmodus spielt das Lernverfahren gegen sich selbst, um seine gelernte Strategie zu verbessern. Der besondere Unterschied der beiden Lernverfahren ist, die Darstellung der Bewertungsfunktion $\hat{U}_\theta(s)$. Gerald Tesauro stellte die Bewertungsfunktion als ein vollständig verknüpftes neuronales Netz, mit einer einzigen verborgenen Schicht mit 40 Knoten dar.

Die erfolgreichen Anwendungen von Arthur L. Samuel und Gerald Tesauro haben gezeigt, dass es verstärkende Lernverfahren gibt, die das Lernen einer annähernd optimalen Strategie, für Dame und Backgammon, ermöglichen.

Literatur

[Alp08] Ethem Alpaydin. *Maschinelles Lernen*. 1. Aufl. Oldenbourg, 2008.

[Ert13] Wolfgang Ertel. *Grundkurs Künstliche Intelligenz: Eine prakmatische Einführung*. 3. Aufl. Springer, 2013.

[Har12] Peter Harrington. *Machine Learning: IN ACTION*. 1. Aufl. Manning, 2012.

[KLM96] Leslie Pack Kaelblig, Michael L. Littman und Andrew W. Moore. „Reinforcement Learning: A Survey“. In: *Jornal of Artificial Intelligence Research* 4 (1996), S. 237–285.

[Mac15] Steve MacGuire. *Strategy Guide for Reversi and Reversed Reversi*. 2015. URL: `http://www.samsoft.org.uk/reversi/strategy.htm` (besucht am 14. 03. 2017).

[Nea96] Joachim Neander. *Computer schlägt Kasparow*. 1996. URL: `https://www.welt.de/print-welt/article652666/Computer-schlaegt-Kasparow.html` (besucht am 14. 03. 2017).

[RN12] Stuart Russell und Peter Norvig. *Künstliche Intelligenz: Ein moderner Ansatz*. 3. Aufl. Pearson, 2012.

[SB12] Richard S. Sutton und Andrew G. Barto. *Reinforcement Learning: An Introduction*. 2. Aufl. MIT Press, 2012.

[Wey77] Willy Weyer. *Schach als Sport - Beitrag des Abendlandes*. 1977. URL: `http://www.schachbund.de/schach-als-sport.html` (besucht am 14. 03. 2017).

[Zob70] Albert L. Zobrist. „A new hashing method with application for game playing“. In: *Technical Report* 88 (1970), S. 1–12.

Eigenständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Bachelorarbeit selbstständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel verfasst habe. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt.