

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

17. März 2017



**Hochschule für Technik
und Wirtschaft Berlin**

University of Applied Sciences

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

Abstrakt

LOREM IPSUM

Abkürzungsverzeichnis

bzw. Beziehungsweise

eng. Englische Sprache

ID Identifikator

MDP Markov decision process

MEP Markov Entscheidungsprozess

UI User interface

vgl. Vergleich

vs. Versus, Gegenüberstellung

Abbildungsverzeichnis

| | | |
|------|--|----|
| 2.1 | Tic Tac Toe Siegesformationen. | 8 |
| 2.2 | Ausgangsspielzustand Reversi. | 10 |
| 2.3 | Spielzugmöglichkeiten Reversi. | 10 |
| 2.4 | Ein (partieller) Suchbaum vgl. [RN12, S. 208] | 13 |
| 2.5 | Ein Alpha Beta Suchbaum [RN12, S. 213]. | 14 |
| 2.6 | Verschiedene Spielzugsequenzen enden im selben Spielzustand. . . . | 17 |
| 2.7 | Zobrist Hashing von Spielzuständen. | 18 |
| 2.8 | Tic Tac Toe Eröffnungssituationen. | 20 |
| 2.9 | Tic Tac Toe Formationsmöglichkeiten. | 21 |
| 2.10 | Reversi Merkmal der aktuellen Mobilität. | 22 |
| 2.11 | Merkmale einer Reversi Heuristik. | 23 |
| 4.1 | Der Agent und die Umgebung vgl. [Ert13, S. 290]. | 32 |
| 4.2 | Tic Tac Toe und Reversi Spielzustände. | 33 |
| 4.3 | Die Projektproblematik. | 34 |
| 4.4 | Klassendiagramm der Software | 43 |
| 5.1 | Alpha-Beta iterativ vertiefende Suche | 45 |
| 5.2 | Iteratives Suchen des maximalen Ergebnisses. | 46 |
| 5.3 | TD-Q-Lernen Algorithmus vgl. [RN12, S. 974] | 47 |
| 5.4 | Die implementierte Explorationsstrategie. | 49 |
| 6.1 | Die Ergebnisse der Agententests. | 56 |

Inhaltsverzeichnis

| | |
|--|------------|
| Abkürzungsverzeichnis | iii |
| Abbildungsverzeichnis | iv |
| 1 Projektvision | 1 |
| 1.1 Zielsetzung | 2 |
| 1.2 Quantifizierung der Ziele | 2 |
| 1.3 Realisierung des Heuristik Agenten | 3 |
| 1.4 Realisierung des TD-Q lernenden Agent | 4 |
| 1.5 Hypothesen | 5 |
| 2 Strategiespiele und Spieltheorie | 6 |
| 2.1 Das Strategiespiel Tic Tac Toe | 7 |
| 2.2 Das Strategiespiel Reversi | 9 |
| 2.3 Spieltheorie | 11 |
| 2.3.1 Minimax-Suche | 12 |
| 2.3.2 Alpha-Beta-Kürzung | 13 |
| 2.3.3 Iterativ vertiefende Tiefensuche | 16 |
| 2.3.4 Übergangstabellen | 17 |
| 2.3.5 Heuristik | 19 |
| 3 Einführung in verstärkendes Lernen | 24 |
| 3.1 Markov-Entscheidungsprozess (MEP) | 25 |
| 3.1.1 Eigenschaften eines MEP | 25 |
| 3.1.2 Zusammenfassung und Anwendung | 27 |
| 3.2 Optimale Taktiken | 28 |
| 3.3 Dynamische Programmierung und Wert-Iteration | 29 |
| 3.4 Temporale Differenz Lernen (TD-Lernen) | 30 |
| 3.5 Q-Lernen (TD-Q-Lernen) | 31 |
| 4 Problemanalyse und Anforderungsdefinition | 32 |
| 4.1 Die Problematik | 32 |
| 4.2 Fallbeispiel: Ein Agent im Labyrinth | 36 |
| 4.3 Anforderungen | 37 |
| 4.3.1 Tic Tac Toe Spielumgebung | 37 |
| 4.3.2 Reversi Spielumgebung | 38 |
| 4.3.3 Agent des Zufalls | 39 |

Inhaltsverzeichnis

| | | |
|----------|--|-----------|
| 4.3.4 | Tic Tac Toe Heuristik Agent | 39 |
| 4.3.5 | Reversi Heuristik Agent | 40 |
| 4.3.6 | Tic Tac Toe TD-Q lernender Agent | 40 |
| 4.3.7 | Reversi TD-Q lernender Agent | 41 |
| 4.3.8 | Testen der Agenten | 41 |
| 4.4 | Die Anforderungen im Klassendiagramm | 42 |
| 5 | Algorithmen und Implementierung | 44 |
| 5.1 | Iterative Alpha-Beta Suche | 44 |
| 5.2 | TD-Q-Lernen | 46 |
| 6 | Validierung | 51 |
| 6.1 | Tic Tac Toe - 9 Spielfelder | 52 |
| 6.1.1 | Heuristik gegen Zufall | 52 |
| 6.1.2 | TD-Q Lernen in 100 Trainingsspielen | 52 |
| 6.1.3 | TD-Q Lernen in 1.000 Trainingsspielen | 53 |
| 6.1.4 | TD-Q Lernen in 10.000 Trainingsspielen | 53 |
| 6.2 | Tic Tac Toe - 16 Spielfelder | 54 |
| 6.2.1 | Heuristik gegen Zufall | 54 |
| 6.2.2 | TD-Q Lernen in 100 Trainingsspielen | 54 |
| 6.2.3 | TD-Q Lernen in 1.000 Trainingsspielen | 55 |
| 6.2.4 | TD-Q Lernen in 10.000 Trainingsspielen | 55 |
| 7 | Auswertung | 57 |
| 7.1 | TD-Q-Lernen - Leistung und Grenzen | 57 |
| 7.1.1 | TD-Q-Lernen Konvergenz (Leistungsfähigkeit) | 57 |
| 7.1.2 | Fluch der Dimensionalität (Grenzen des TD-Q-Lernens) | 60 |
| 7.1.3 | Zusammenfassung | 62 |
| 7.2 | Ausblick | 63 |
| 7.2.1 | Samuels-Dame-Spiel | 63 |
| 7.2.2 | TD-Gammon | 64 |

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 bestimmte Computerspiele erlernen. Wir bezeichnen die Implementierung des TD-Q-Lernens fortan als TD-Q-Agenten.

Die Anwendungsgrundlage für das TD-Q-Lernen werden die Computerspiele Tic Tac Toe und Reversi. Wir werden die Strategiespiele Tic Tac Toe mit 9 Spielfeldern, Tic Tac Toe mit 16 Spielfeldern und Reversi mit 64 Spielfeldern eigenständig implementieren.

Testkriterium für die Leistungsfähigkeit des TD-Q-Agenten werden Testspiele gegen andere Agenten mit unterschiedlichen Strategien. Wir werden daher innerhalb dieser Arbeit einen Zufallsagenten und einen vorausschauenden Heuristik Agent entwickeln und implementieren. Ein Zufallsagent wählt immer einen zufälligen Spielzug, aus einer Auswahl von möglichen Spielzügen.

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 erhöhen die Anzahl der Trainingsspiele in der zweiten Phase auf 1.000 und in der dritten Phase auf 10.000.

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.

1.3 Realisierung des Heuristik Agenten

Der vorausschauende Heuristik Agent ist ein nicht lernender Agent der eine von uns festgelegte Strategie erhält. Die Strategie besteht aus einer Stellungsbewertung (Heuristik) und einer 2 Spielzüge vorausschauenden Spielbaumsuche. Wir werden in Abschnitt ?? Spieltheorie die Theoretischen Grundlagen erklären, die wir für die Implementierung des vorausschauenden Heuristik Agenten benötigen.

In der Implementierung des vorausschauenden Heuristik Agenten ist, neben der Bewertungsfunktion, noch ein anderes Verfahren enthalten. Das Suchbaumverfahren für 2-Personenspiele. Dieses Verfahren durchsucht einen Spielbaum nach der bestmöglichen Aktion (einem Spielzug) in einem gegebenen Zustand. Ein Zustand oder Spielzustand ist eine Spielsituation bzw. eine Stellung der Spielfiguren auf dem Spielfeld.

Das Problem der Suchverfahren ist die Dimensionalität bzw. Komplexität des Ausgangsproblems. Suchbaumverfahren können für sehr einfache Probleme relativ schnell eine optimale Aktion finden. Die Größe des Suchbaums wächst exponentiell mit der Komplexität des Problems, d.h. die Laufzeit des Suchbaumverfahrens ohne Erweiterungen könnte für das Strategiespiele Tic Tac Toe nicht handhabbar sein und ist für das Strategiespiel Reversi nicht handhabbar. Wir schreiben "könnte" bei Tic Tac Toe, weil dieses noch ein recht einfacher Vertreter der Strategiespiele ist, dahingegen ist Reversi ein komplexeres Strategiespiel.

Um die Dimensionalitätsproblematik zu lösen, kombinieren wir Suchbaumverfahren mit Heuristiken, wir bezeichnen diese Kombination als heuristische Suche. Eine Heuristik berechnet eine Gewinnwahrscheinlichkeit, ausgehend von einem Spielzustand. Ein Spielzustand mit einer hohen heuristischen Bewertung ist, gegenüber einem Spielzustand mit niedriger heuristischer Bewertung, zu bevorzugen.

Das Suchbaumverfahren muss den Suchbaum, unter Verwendung einer Heuristik, nicht mehr komplett durchsuchen. Die Suche kann in einer bestimmten Suchbaumtiefe abgebrochen werden. Das Suchbaumverfahren liefert die erste Aktion einer Aktionssequenz. Eine Aktionssequenz ist eine Folge von Aktionen und beschreibt einen Pfad im Suchbaum. Die Aktionssequenz, welche von der heuristischen Suche ausgewählt wurde, repräsentiert den Spielzustand mit der maximalen Gewinnwahrscheinlichkeit.

Die Qualität dieser Gewinnschätzung ist wiederum von der maximalen Suchtiefe und der Bewertungsfunktion abhängig. Eine größere Suchtiefe resultiert in einer besseren Schätzung, weil unter Umständen mehr Spielzustände berücksichtigt werden können. Die Verwendung einer Bewertungsfunktion ist keine Garantie für eine

optimale Strategie. Verschiedene Bewertungsfunktionen können stark voneinander abweichende Gewinnschätzungen für Spielzustände berechnen.

1.4 Realisierung des TD-Q lernenden Agent

Wir stellen mehrere Lernverfahren innerhalb dieser Arbeit vor, aber wir werden nur das Q-Lernen (auch TD-Q-Lernen) implementieren und untersuchen. Das TD-Q-Lernen ist ein Lernverfahren aus dem Bereich des verstärkenden Lernens. Das TD-Q-Lernen soll es uns ermöglichen einen selbst lernenden Agenten zu programmieren. 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.

Wie realisiert das TD-Q-Lernen diesen verstärkenden Lernansatz? Das TD-Q-Lernen lernt Q-Werte für Zustand/Aktionspaare, diese Q-Werte werden bei jedem erneuten Auftreten des Zustand/Aktionspaares aktualisiert. Eine Q-Funktion ist eine Abbildung von allen möglichen Zustand/Aktionspaaren auf Q-Werte und eine Q-Funktion ist eine Möglichkeit Nutzeninformationen zu speichern [RN12, S. 974]. Nachdem der Agent eine Q-Funktion gelernt hat, kann er mittels dieser, vermeintlich optimale Aktionen auswählen. Wir schreiben "Vermeintlich", weil eine gelernte Q-Funktion nicht immer zu einer optimalen Strategie konvergiert.

Wir zeigen in dieser Arbeit praktisch, dass das TD-Q-Lernen ohne Erweiterungen, nur auf Probleme mit geringer Komplexität angewendet werden kann. Die Komplexität bzw. Dimensionalität des Ausgangsproblems ist ein Grund dafür, dass die gelernte Q-Funktion nicht immer zu einer optimalen Strategie konvergiert, ein anderer Grund ist die zeitliche Beschränkung durch die Realität, d.h. in der Realität können nicht unendlich viele Testspiele durchgeführt werden. Es wurde bereits empirisch belegt, dass das Q-Lernen, sollte jedes Zustand/Aktionspaar nahezu unendlich oft besucht und aktualisiert werden, immer zu einer optimalen Strategie konvergiert. Das Problem dabei ist, dass die Komplexität bzw. die Dimensionalität des Ausgangsproblems, ein exponentiellen Verhältnis zur Zustands- und Aktionsmenge hat.

Lernt der TD-Q Agent, z.B. innerhalb von 10.000 Testspielen eine nahezu optimale Strategie für ein Tic Tac Toe Spiel mit 3 mal 3 Dimensionen (9 Spielfelder), dann ist das TD-Q-Lernen praktisch für ein Strategiespiel bis zu dieser Dimensionalität anwendbar. Erhöhen wir die Zustands- und Aktionsdimension, z.B. bei einem 16 Spielfelder Tic Tac Toe Spiel, dann reichen selbst 1.000.000 Testspiele unter Umständen nicht mehr aus, um eine annähernd optimale Strategie zu lernen. Jede weitere Dimension erhöht außerdem die Dauer eines Trainingsspiels, d.h. für jede weitere Dimension benötigt das TD-Q-Lernverfahren erheblich mehr Testspiele, um zu einer annähernd optimalen Strategie zu konvergieren und gleichzeitig erhöht sich die Dauert jedes Testspiels für jede zusätzliche Dimension des Ausgangsproblems.

1.5 Hypothesen

1. Der Heuristik Agent wird in beiden Strategiespielen gegen den lernenden Agenten mindestens 50% 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 die in dieser Arbeit implementierte 2-Züge vorausschauende Heuristik-Suche.
3. Das TD-Q-Lernen muss möglicherweise mehr als 10.000 Trainingsspiele gegen sich selbst spielen, um eine bessere Strategie, als die nicht lernende Strategie, zu lernen.
4. Die Konvergenzgeschwindigkeit das TD-Q-Lernen zu einer optimalen Strategie, ist möglicherweise stark von der Dimensionalität des Ausgangsproblems abhängig, d.h. genau wie die uninformierten Suchbaumverfahren, ist das TD-Q-Lernen nur auf sehr einfache bzw. niedrig dimensionale Probleme anwendbar. Konvergenzgeschwindigkeit ist die Zeit, die ein Lernverfahren benötigt, bis es eine annähernd optimale Strategie entwickelt hat.
5. Das reine TD-Q-Lernen, ohne Erweiterungen, ist möglicherweise keine geeignetes Lernverfahren für das lernen eines Strategiespiels.

Strategiespiele und Spieltheorie

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.

Im Unterkapitel Spieltheorie (Abschnitt 2.3) werden uninformierte Suchbaumverfahren, deren Optimierungsmöglichkeiten, Übergangstabellen und Heuristiken erklärt.

Die Minimax-Suche (Abschnitt 2.3.1) ist ein uninformiertes Suchbaumverfahren. Die Minimax-Suche kann in Zweipersonenstrategiespielen eingesetzt werden, um eine optimale Strategie zu finden. In der Praxis ist das Verfahren jedoch nicht anwendbar. Die Suchbäume realistischer Probleme sind meist entartet bzw. zu groß für eine klassische Minimax-Suche. Die Alpha-Beta-Kürzung (Abschnitt 2.3.2) ist eine Verbesserung der Minimax-Suche, dieses Optimierungsverfahren versucht, den meist viel zu großen Suchbaum zu kürzen, ohne das Endergebnis der Minimax-Suche zu beeinflussen. Wir werden die Minimax-Suche bzw. Alpha-Beta-Suche für die Implementierung des Heuristik Agenten verwenden. Die iterativ vertiefende Tiefensuche (Abschnitt 2.3.3) ist ein uninformiertes Suchverfahren, welches Breitensuche und Tiefensuche kombiniert und bis zu einer bestimmten Suchtiefe ein bestmögliches Ergebnis sucht. Wir verwenden die iterativ vertiefende Tiefensuche, für die Verbesserung der vorausschauenden Suche des Heuristik Agenten.

Die Übergangstabellen (Abschnitt 2.3.4) beschreiben eine Möglichkeit Übergänge zu vermeiden. Übergänge sind identische Spielsituationen, die durch unterschiedliche Aktionssequenzen (eine Folge von Spielzügen) dargestellt werden können, daher scheinen die Spielsituationen unterschiedlich zu sein. Das Zobrist-Hash Verfahren beschreibt eine Möglichkeit die Spielsituationen eindeutig als Zahlwerte dar-

stellen zu können. Wir verwenden dieses Hashverfahren bzw. auch eine Art der Übergangstabellen für das TD-Q lernen.

Heuristiken bilden die stärkste Leistungsverbesserung, bezüglich der Rechenzeit, für Suchbaumverfahren. Eine Heuristik (Abschnitt 2.3.5) ist eine Bewertungsfunktion $B(s)$, welche für jeden Spielzustand (Stellung der Spielfiguren) s eine Schätzung bereitstellt. Die Schätzung gibt an, wie hoch die Gewinnchance, in einem bestimmten Spielzustand, für den Spielzug ausführenden Spieler, ist. Heuristiken ermöglichen das abbrechen der Suche innerhalb eines Suchbaumes, d.h. auch die nicht Blattknoten des Baumes können Spielergebnisse bereitstellen. Die Kombination von uninformativer Suche und Heuristiken ist nicht mehr uninformativ. Heuristiken stellen zusätzliche Spielinformationen bereit. Die Literatur bezeichnet dies daher als heuristische (informierte) Suche [Ert13, S. 105].

Wir fassen zusammen: Die Implementierung des Heuristik Agenten erhält eine iterativ vertiefende Alpha-Beta Suche, diese kann auch nicht Blattknoten des Suchbaums, mittels einer Heuristik, bewerten. Wir können also die Rechenzeit des Heuristik Agenten begrenzen, indem wir die maximale Suchtiefe auf 2 Halbzüge begrenzen. Der Heuristik Agent expandiert, wie die iterativ vertiefende Tiefensuche, zu erst alle Knoten in einer Tiefe und erhöht danach schritt für schritt (iterativ) die aktuelle Tiefenschranke, bis zu der maximal möglichen Tiefe.

2.1 Das Strategiespiel Tic Tac Toe

In diesem Abschnitt definieren wir die Regeln des Strategiespiels Tic Tac Toe. Der lernende TD-Q Agent soll für diese Umgebung eine annähernd optimale Strategie lernen. In einem späteren Abschnitt (2.3.5 Heuristik) werden wir eine Tic Tac Toe Heuristik für den Heuristik Agenten entwerfen. Wir verwenden Tic Tac Toe als Anwendungsgrundlage für unsere Agentenmodelle verwenden und wir werden die Lernfähigkeit des TD-Q Agenten messen, anhand von Tic Tac Toe Testspielen gegen die anderen beiden Agenten (siehe Abschnitt 6 Validierung).

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 3×3 Spielbrett). Innerhalb dieser Arbeit betrachten wir auch ein Tic Tac Toe Spiel mit 16 Spielfeldern (ein 4×4 Spielbrett). Der beginnende Spieler muss Kreuzspielfiguren setzen und der nachziehende Spieler Kreisspielfiguren.

Spielzüge jeder Spieler setzt abwechselnd entweder ein Kreuz oder 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 oder vier Kreise in einer bestimmten Position anzuordnen (Siegesformation). Es existieren mehrere unterschiedliche Anordnungen von Spielfiguren, die das Spiel beenden und einen Sieg herbeiführen. Bei einem 4×4 Spielfeld existieren vier vertikale, vier horizontale und zwei diagonale Anordnungen 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.

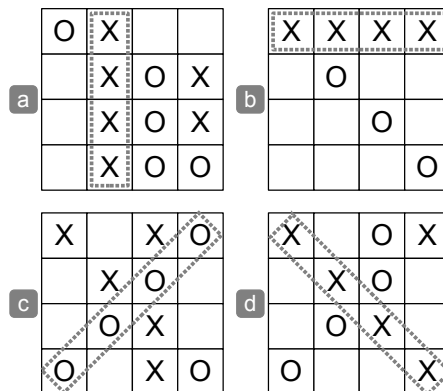


Abbildung 2.1 Tic Tac Toe Siegesformationen.

Vier mögliche Siegesformationen sind in Abbildung 2.1 dargestellt. (a) Kreuz gewinnt, mit einer vertikalen Siegesformation. (b) Kreuz gewinnt, mit einer horizontalen Siegesformation. (c) Kreis 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. Genau wie bei dem Strategiespiel Tic Tac Toe, sollen die Agenten in diesem Spiel gegeneinander antreten und der TD-Q Agent soll eine annähernd optimale Strategie für dieses Spiel lernen. Eine Reversi Heuristik wird ebenfalls in einem späteren Abschnitt entworfen. Reversi ist ein komplexeres Strategiespiel als Tic Tac Toe, d.h. die gelernte Strategie des TD-Q Agenten, benötigt viel mehr Zeit, um eine annähernd optimale Strategie für Reversi zu lernen. Zudem dauert die Berechnung der Stellungsbewertungen, des Heuristik Agenten, (vermutlich) wesentlich länger, als bei Tic Tac Toe. Das Testen der Reversi Heuristik und das Testen der gelernten Reversi Strategie, durch den TD-Q Agenten, könnte viel Rechenzeit beanspruchen. Reversi ist wesentlich komplexer als Tic Tac Toe, weil Reversi 64 Spielfelder (ein 8×8 Spielbrett), statt 16 Spielfelder hat. Das Reversi Spielbrett ist um den Faktor 4 größer, als das 4×4 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. Wie beim klassischen Tic Tac Toe behalten die Spieler während des gesamten Spiels ihre Spielsteinfarbe 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).

Spielzüge

Eine Besonderheit von Reversi ist, dass gesetzte Spielsteine ihre Farbe ändern können. Werden z.B. zwei weiße Spielsteine von zwei schwarzen eingeschlossen, dann werden die weißen Spielsteine in schwarze umgewandelt. Ein korrekter Spielzug muss immer mindestens einen gegnerischen Spielstein erobern. Weiterhin darf ein Spielstein nur dann gesetzt werden, wenn ein anderer Spielstein (Anker), in einer diagonalen, vertikalen oder horizontalen Linie, existiert. Es dürfen auch keine freien Felder zwischen dem zu setzendem Stein und dem Anker liegen. Ein Anker ist ein Spielstein mit der selben Farbe wie der zu setzende Spielstein. Ein zu setzender Spielstein kann mehrere Anker haben, aber er muss mindestens einen haben.

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| 0 | | | | | | | | |
| 1 | | | | | | | | |
| 2 | | | | | | | | |
| 3 | | | | ● | ○ | | | |
| 4 | | | | ○ | ● | | | |
| 5 | | | | | | | | |
| 6 | | | | | | | | |
| 7 | | | | | | | | |

Abbildung 2.2 Ausgangsspielzustand Reversi.

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|-----|---|---|---|---|---|---|---|---|
| 0 | ○ | | | ○ | | | • | |
| 1 | | ○ | ○ | ○ | | ○ | | |
| 2 | | ○ | ○ | ○ | ○ | | | |
| a 3 | | • | ○ | ● | ○ | ○ | • | |
| 4 | | | ○ | ○ | ○ | | | |
| 5 | | • | ○ | • | ○ | ○ | | |
| 6 | | ○ | | | | | ○ | |
| 7 | | | | | | | | • |

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|-----|---|---|---|---|---|---|---|---|
| 0 | ○ | ● | • | ○ | ● | | • | |
| 1 | | ○ | ○ | ○ | | ○ | | |
| 2 | | ○ | ○ | ○ | ○ | | • | |
| b 3 | | • | ○ | ● | ○ | ○ | • | |
| 4 | | | ○ | ○ | ○ | • | | |
| 5 | | • | ○ | • | ○ | ○ | | |
| 6 | | ○ | | | | | ○ | |
| 7 | | | | | | | | • |

Abbildung 2.3 Spielzugmöglichkeiten Reversi.

Anmerkung zu Abbildung 2.3, diese zeigt zwei möglicherweise nicht in der Praxis 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 Spielsituation mit maximal einem möglichen Anker. (b) Eine Spielsituation mit maximal 3 möglichen Ankern für die Position (3,1).

Abbildung 2.3 zeigt zwei verschiedene Reversi Spielsituationen (Schwarz ist am Zug). Die kleinen schwarzen Kreise 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 Ankersteine diagonal, horizontal und vertikal an diese Position angrenzen.

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.

2.3 Spieltheorie

In diesem Abschnitt werden wir erklären was uninformierte Suchbaumverfahren, Übergangstabellen und Heuristiken sind. Wir verwenden diese Methoden, aus der Spieltheorie, für die Implementierung unserer Agentenmodelle. Wolfgang Ertel beschreibt Spiele mit Gegenspieler wie folgt [Ert13, S. 114]:

Schach, Vier gewinnt, Dame, Tic Tac Toe und Reversi sind strategische Spiele für zwei Personen, die gegeneinander antreten, um nach den Regeln des Spiels, den Gegenspieler zu besiegen. Diese Spiele sind deterministisch, weil das Spiel nicht vom Zufall abhängt und der gleiche Spielzug führt, bei gleichem Ausgangszustand, immer zum selben Spielergebnis. Ein nichtdeterministisches Spiel mit Gegenspieler ist z.B. Backgammon, denn Würfelergebnisse und somit der Zufall sind Bestandteil des Spiels. Die eben genannten Strategiespiele sind alle vollständig beobachtbar (wir verfügen über vollständige Information). Sie sind vollständig überschaubar, weil zu jedem Zeitpunkt des Spiels, das Spielfeld und alle Spielzüge einsehbar sind. Ein nicht vollständig beobachtbares Spiel ist, z.B. Poker oder andere Kartenspiele. Bei einem Poker Spiel sind die gegnerischen Karten und die Karten im Spieldeck unbekannt.

“In der künstlichen Intelligenz haben die gebräuchlichsten Spiele in der Regel eine spezielle Natur - die Spieltheoretiker sprechen von deterministischen Zwei-Personen-Nullsummenspielen mit vollständiger Information, bei denen zwei Spieler abwechselnd agieren (wie zum Beispiel Schach) [RN12, S. 206].“ Die von Russell und Norvig erwähnten Eigenschaften treffen auch auf unsere Strategiespiele zu. Reversi und Tic Tac Toe sind deterministische und vollständig überschaubare Nullsummenspiele, d.h. wir müssen zufällig auftretende Zustandsübergänge und unbekannte Informationen, in unseren Heuristiken und Lernverfahren, nicht berücksichtigen. Nullsummenspiel bedeutet: gewinnt ein Spieler eine Partie, dann verliert der Gegenspieler automatisch in gleicher Höhe.

2.3.1 Minimax-Suche

Wir werden die von Russell und Norvig beschriebene Minimax-Suche, für die Implementierung des vorausschauenden Heuristik Agenten verwenden, daher werden wir die Minimax-Suche genauer erklären.

“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 2.5 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 sein Kreuzspielstein in die obere linke Ecke setzen, daraus ergeben sich neue Zustandsmöglichkeiten. Spieler MIN 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.

“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.].” Wolfgang Ertel beschreibt in diesem Zitat das Verhältnis von Problemkomplexität und Suchbaumgröße. Genau dieses Problem wird auch das TD-Q-Lernen stark beeinflussen. Das TD-Q-Lernen exploriert zwar keinen Suchbaum, jedoch steigen

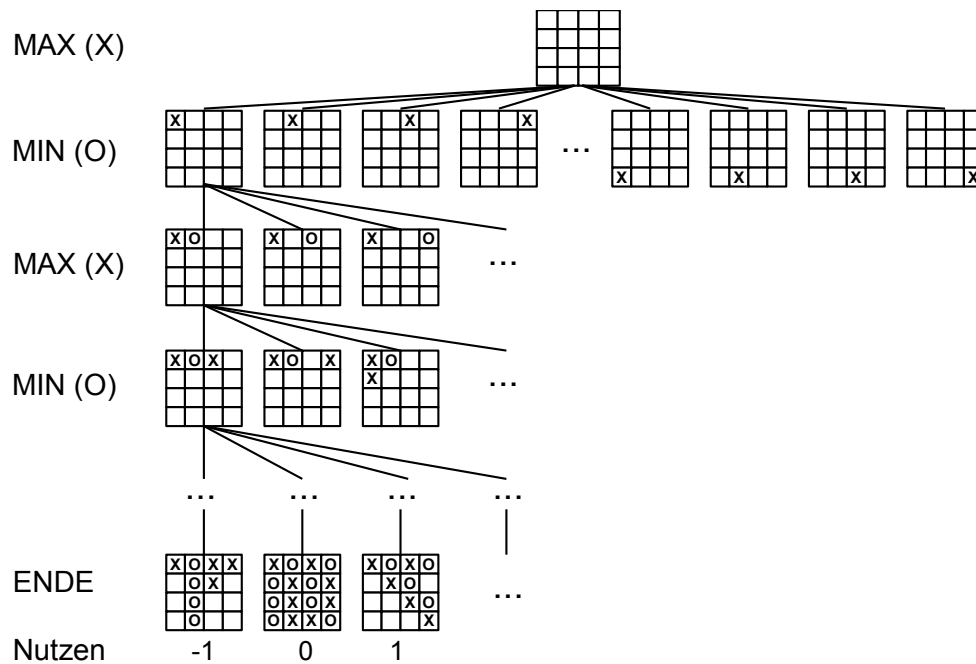


Abbildung 2.4 Ein (partieller) Suchbaum vgl. [RN12, S. 208]

die nötigen Lerndurchläufe, bis zur Konvergenz zu einer optimalen Strategie, exponentiell mit der Komplexität des Problems. Wir werden im Kapitel Validierung praktisch zeigen, welche Auswirkungen verschieden komplexe Probleme, auf das TD-Q-Lernen, haben. Im Kapitel Auswertung werden wir mögliche Lösungsmöglichkeiten vorstellen.

2.3.2 Alpha-Beta-Kürzung

Im vorherigen Abschnitt haben wir erklärt was genau die Minimax-Suche ist und warum diese praktisch nicht anwendbar ist (siehe [Ert13, S. 114 f.]). Eine Möglichkeit die Rechenzeit der Minimax-Suche zu verbessern, ist das Kürzen oder Beschneiden des Suchbaums (eng. Pruning). Wir verwenden diese Minimax-Optimierung für die Implementierung eines vorausschauenden Heuristik Agenten.

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 Maximum Knoten, der bisher bei der Traversierung (Erkundung oder das Durchlaufen) des Suchbaums gefunden wurde. In Beta wird für jeden Minimum Knoten der bisher kleinste gefundene Wert gespeichert. Ist Beta an einem Minimum Knoten kleiner oder

gleich Alpha ($Beta \leq Alpha$), so kann die Suche unterhalb von diesem Minimum Knoten abgebrochen werden. Ist Alpha an einem Maximum Knoten größer oder gleich Beta ($Alpha \geq Beta$), so kann die Suche unterhalb von diesem Maximum Knoten abgebrochen werden.

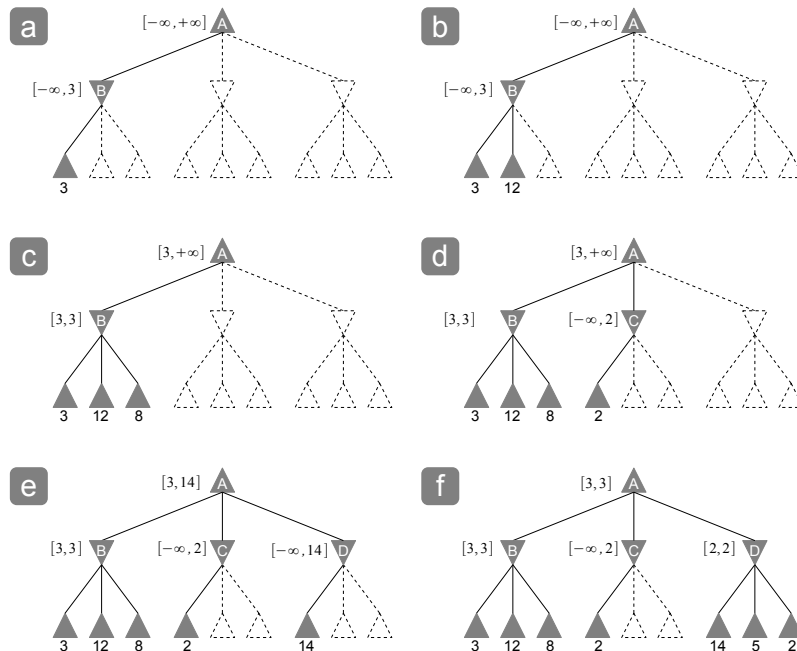


Abbildung 2.5 Ein Alpha Beta Suchbaum [RN12, S. 213].

Verdeutlichen wir das Alpha-Beta-Pruning an Hand eines Beispiels (Abbildung 2.5). 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) Minimum Knoten B findet einen Nutzwert 3, da dieser Wert der bisher kleinste gefundene Wert ist wird er in Beta gespeichert.

(b) Der Minimum Knoten 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 Minimum Knoten B ignoriert und

Beta bleibt unverändert.

(c) Minimum Knoten 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. Minimum Knoten B hat alle seine nachfolgenden Knoten exploriert. Maximum Knoten A wird vom Minimum Knoten B maximal den Nutzwert 3 erhalten, somit ergibt sich für den Maximum Knoten A, dass dieser mindestens den Nutzwert 3 erreichen kann.

(d) Ein weiterer Minimum Knoten ist C. Der erste Blattknoten von C liefert einen Nutzwert von 2, weil dieser Wert der erste gefundene Wert unterhalb des Minimum Knotens C ist, wird er in Beta gespeichert. C wird Maximum Knoten A maximal einen Nutzwert 2 liefern. A wiederum kann durch Minimum Knoten 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 Minimum Knoten 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 Minimum Knoten wird exploriert. Der erste Blattknoten unterhalb des Minimum Knoten D liefert den Nutzwert 14. Dieser Wert wäre für Maximum Knoten 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) Minimum Knoten 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 Minimum Knoten D jedoch nicht abgeschnitten, weil der Nutzwert 2 erst im zuletzt explorierten Knoten gefunden wurde. Theoretisch könnten zwei Pfade unterhalb des Minimum Knoten D abgeschnitten werden, wenn der Blattknoten mit dem Nutzwert 2 zuerst exploriert worden wäre.

Praktisch ist die Alpha-Beta Kürzung ebenfalls nicht anwendbar. Selbst wenn ein großer Teil des Suchbaums abgeschnitten werden kann, ohne das Ergebnis zu beeinflussen, dann ist eine Exploration des Suchbaums immer noch, aus Gründen zu großer Rechenzeit, zu teuer. Wir werden daher noch die iterativ vertiefende Tiefen-

suche und Heuristiken behandeln.

2.3.3 Iterativ vertiefende Tiefsuche

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

Die iterativ vertiefende Tiefsuche (eng. Iterative Deepening) ist ein kombinatorisches bzw. uninformatiertes Suchbaumverfahren und kombiniert die Breitensuche mit der Tiefsuche. Die Strategien der uninformatierten 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. Die Reihenfolge der Suche ist entscheidend für die Unterscheidung der einzelnen uninformatierten Suchverfahren.

Die Breitensuche expandiert (erweitert oder vergrößert) zu erst alle Nachfolger (Knoten eines Suchbaums) die in derselben Tiefe liegen, beginnend mit dem Wurzelknoten. Sind alle Nachfolger einer Tiefe expandiert, dann werden deren Nachfolger nacheinander expandiert. Diesen Schritt wiederholt die Breitensuche bis ein gesuchtes Ergebnis gefunden wird oder der Suchbaum vollständig exploriert (erkundet) ist.

Die Tiefsuche expandiert zuerst die tiefsten Knoten des Suchbaums (Depth-first). Erreicht die Tiefsuche einen Endknoten der nicht dem gesuchten Ergebnis entspricht, dann werden die alternativen Knoten des letzten expandierten Knotens, der sich eine Tiefenebene höher befindet, expandiert.

Kombinieren wir diese beiden uninformatierten Suchverfahren miteinander und mit einer Grenze für die Suchtiefe, erhalten wir die iterative Tiefsuche. Diese expandiert zuerst die Nachfolger des Wurzelknotens der Suchtiefe 1. Sind alle Knoten auf dieser Ebene expandiert, dann wird die Schranke für die aktuelle Suchtiefe um 1 erhöht (Iteration) und die Knoten der Suchtiefe 2 werden expandiert. Diese Schritte wiederholt die Tiefsuche 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 Tiefsuche 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.

2.3.4 Übergangstabellen

Eine Übergangstabelle (eng. transition table) ist eine Tabelle in der Spielsituationen mit verschiedenen Attributen gespeichert werden (vgl. [RN12, S. 215 f.]). Übergänge sind der Grund dafür, dass der gleiche Spielzustand durch unterschiedliche Spielzugsequenzen auftritt (siehe Abbildung 2.6).

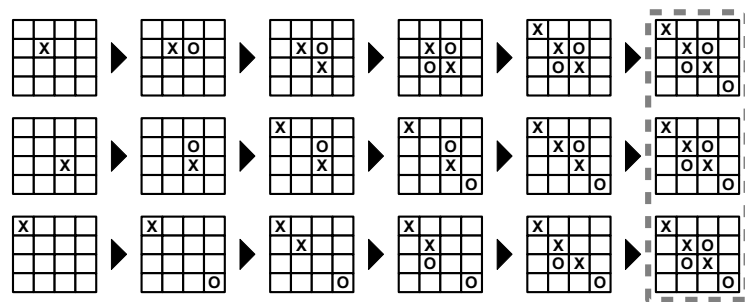


Abbildung 2.6 Verschiedene Spielzugsequenzen enden im selben Spielzustand.

Übergänge innerhalb des Suchbaums verursachen Redundanzen. Für jede dieser Redundanzen wird eine erneute Suche durchgeführt, falls diese nicht durch Alpha-Beta-Kürzung abgeschnitten werden. Sollten diese Übergänge vermieden werden können, dann würde sich die Rechenzeit der Suchverfahren weiter verringern, weil weniger Spielzustände durchsucht bzw. expandiert werden müssen.

Wir können uns das TD-Q-Lernen, als eine Art Übergangstabelle vorstellen. Die Tabelle würde 3 Spalten haben. Die erste Spalte beinhaltet den Spielzustand, die zweite Spalte die dazugehörige Aktion und die dritte Spalte enthält den Q-Wert (Nutzwert), d.h. Zustand/Aktionspaare werden auf Q-Werte abgebildet. Diese Tabelle wird nicht unbedingt geführt um Redundanzen zu vermeiden, sondern eher um Nutzeninformationen zu speichern.

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 lernenden Agenten, weil die Spielsituationen, als Zobrist-Hash, in der Q-Wertetabelle eingetragen werden können.

a

| | | | |
|--------------------------------------|--------------------------------------|--------------------------------------|--------------------------------------|
| X = 660640090 O = 601151343 | X = 651080001 O = 550176261 | X = 707754336 O = 30179116 | X = 240651458 O = 515695098 |
| X = 843817469 O = 625774421 | X = 446956442 O = 409234428 | X = 888791315 O = 906370688 | X = 10057952 O = 962066669 |
| X = 925070678 O = 747101521 | X = 179513842 O = 89793577 | X = 538866973 O = 222479865 | X = 144262103 O = 353844301 |
| X = 595995309 O = 751411292 | X = 883501364 O = 531273511 | X = 727572818 O = 91717317 | X = 7191668 O = 704554166 |

b

| | | | |
|---|---|---|---|
| X | | | |
| | X | O | |
| | O | X | |
| | | | O |

c

$$\begin{aligned}
 &660640090 \wedge 446956442 \wedge 906370688 \wedge \\
 &89793577 \wedge 538866973 \wedge 704554166 \\
 &= \underline{\underline{125309938}}
 \end{aligned}$$

Abbildung 2.7 Zobrist Hashing von Spielzuständen.

In Abbildung 2.7 wird das Zobrist Hash Verfahren auf den redundanten Spielzustand aus Abbildung 2.6 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.

2.3.5 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 oder 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. Eine schlechte Stellungsbewertung (Heuristik), kann schlechte Spielzüge verursachen oder fatale Spielzüge des Gegners übersehen.

Die Verwendung einer Heuristik ermöglicht es, nicht Blattknoten eines Spielbaumes zu bewerten, somit können Suchverfahren die Suche in einer bestimmten Tiefe abbrechen und das bisher beste bisher gefundene Ergebnis zurückliefern. Aus Gründen der starken Rechenzeitverbesserung implementieren wir Heuristiken für den nicht lernenden Agenten.

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

$$B(s) = a_1 \times \text{Material} + a_2 \times \text{Bauernstruktur} + a_3 \times \text{Königssicherheit} \\ + a_4 \times \text{Springer im Zentrum} + a_5 \times \text{Läufer Diagonalabdeckung} + \dots,$$

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

$$\text{Material} = \text{Material}(\text{eigenes Team}) - \text{Material}(\text{Gegner})$$

$$\text{Material}(\text{Team}) = \text{Anzahl Bauern}(\text{Team}) \times 100 + \text{Anzahl Springer}(\text{Team}) \times 300 \\ + \text{Anzahl Läufer}(\text{Team}) \times 300 + \text{Anzahl Türme}(\text{Team}) \times 500 \\ + \text{Anzahl Damen}(\text{Team}) \times 900$$

Wolfgang Ertel schreibt sinngemäß [Ert13, S. 118]: Diese Schach Heuristik ist entstanden aus der Zusammenarbeit von Schachexperten und Wissensingenieuren. Die Schachexperten verfügen über Wissen und Erfahrungen bezüglich des Schach-

spiels, 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.

Formal definieren Russell und Norvig Bewertungsfunktionen [RN12, S. 218]:

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

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

Tic Tac Toe Heuristik

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

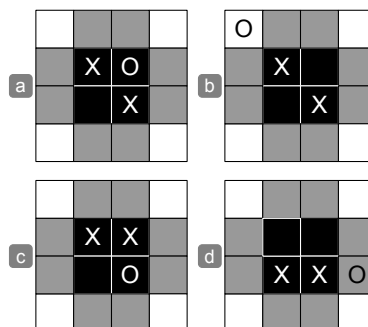


Abbildung 2.8 Tic Tac Toe Eröffnungssituationen.

Abbildung 2.8 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 und wie viele ungestörte Formationsmöglichkeiten der Gegenspieler hat.

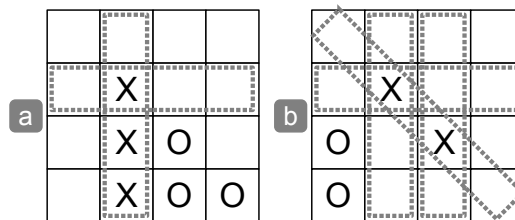


Abbildung 2.9 Tic Tac Toe Formationsmöglichkeiten.

Betrachten wir die ungestörten möglichen Siegesformationen in Abbildung 2.9. 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 höherwertiger, 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.

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 2.10 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.

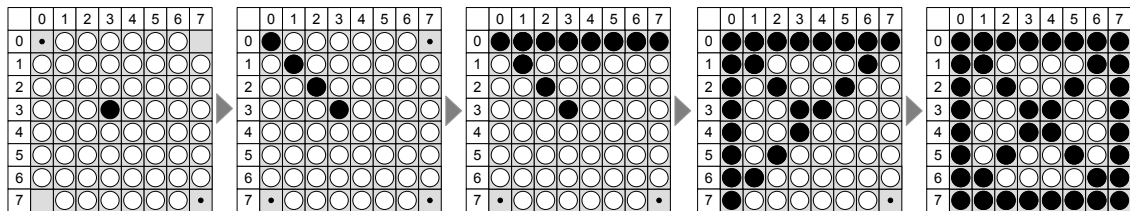


Abbildung 2.10 Reversi Merkmal der aktuellen Mobilität.

Die mögliche Mobilität beachtet alle freien Spielfelder die an gegnerische Spielsteine angrenzen. In Abbildung 2.11 (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 2.10 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 2.11 (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 2.11 (c) sind die einzelnen Bewertungen der Positionen veranschaulicht. Diese numerischen Bewertungen beziehen sich auf das gesamte Reversi Spielbrett, weil dieses Symmetrisch ist.

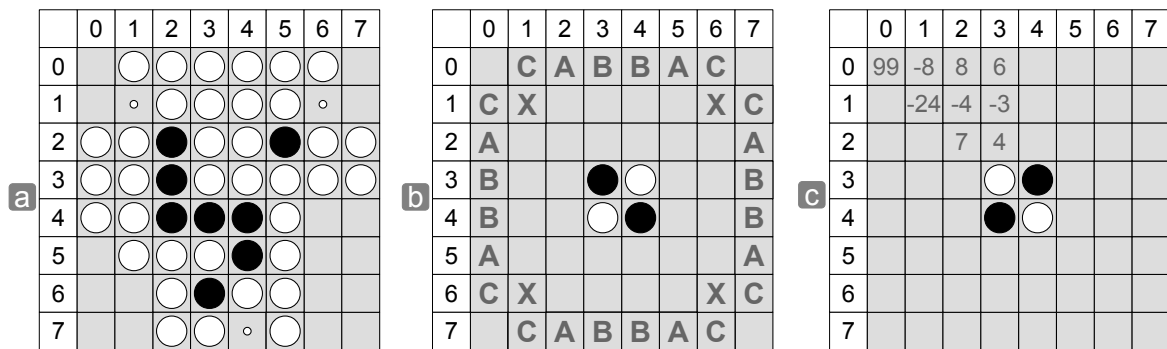


Abbildung 2.11 Merkmale einer Reversi Heuristik.

Einführung in 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 dieses Lernverfahren soll die Strategiespiele Reversi und Tic Tac Toe 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-Gleichung (siehe Abschnitt 3.3), die Temporale Differenz (siehe Abschnitt 3.4) und das Q-Lernen (siehe Abschnitt 3.5). Das Q-Lernen oder auch TD-Q-Lernen soll in dieser Arbeit angewendet und untersucht werden, die Wert-Iteration wird nur erklärt aber nicht implementiert. Die Temporale Differenz wiederum ist eine Grundlage für das Q-Lernen bzw. das Q-Lernen ist eine Variante des TD-Lernens.

Bevor wir jedoch die einzelnen Lernverfahren erläutern werden, müssen wir noch

definieren wie wir die beiden Strategiespiele Reversi und Tic Tac Toe modellieren wollen (siehe Abschnitt 3.1) und wie eigentlich eine optimale Strategie definiert ist (siehe Abschnitt 3.2).

3.1 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.

Wir verwenden einen Markov-Entscheidungsprozess, um den Agenten zu modellieren, d.h. die Eigenschaften eines Markov-Entscheidungsprozesses sind auch die Eigenschaften des lernenden Agenten. Wir modellieren nachfolgend die Eigenschaften eines MEP auf die beiden Strategiespiele Tic Tac Toe und Reversi.

3.1.1 Eigenschaften eines MEP

„Ein **sequentielles Entscheidungsproblem** für eine vollständig beobachtbare, stochastische Umgebung mit einem Markov-Übergangsmodell und additiven Gewinnen heißt Markov-Entscheidungsprozess oder MEP. [RN12, S. 752]“ Bei diesen Problemen ist dem Agenten der direkte Nutzen des Aktionsergebnisses nicht bekannt, erst nach einer Folge von Aktionen (am ende des Spiels) wird dem Agenten eine Belohnung zugeteilt (vgl. [Alp08, S. 397]).

Dies trifft auch auf die Strategiespiele Tic Tac Toe und Reversi zu, denn bei diesen Spielen erhält der Agent keine direkte Belohnung nach den einzelnen Spielzügen. Erst am Ende eines Spiels entsteht ein Sieg, eine Niederlage oder ein Unentschieden

und der Agent erhält dem entsprechen eine verspätete Belohnung.

Wolfgang Ertel erklärt vollständig beobachtbare Spiele wie folgt (vgl. [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. Reversi und Tic Tac Toe sind deterministische Strategiespiele, d.h. jeder Nachfolgezustand ist eindeutig definiert, eine Aktionssequenz führt also immer zum selben Ergebnis. Es finden keine Stochastischen Übergänge in Reversi und Tic Tac Toe 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 Reversi und Tic Tac Toe 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:

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

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]).

Reversi und Tic Tac Toe 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 Vermeidungsfaktor 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.

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

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

3.1.2 Zusammenfassung und Anwendung

Reversi und Tic Tac Toe 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 3.4). Wie bereits erwähnt sind die beiden Strategiespiele vollständig beobachtbar und nicht stochastisch, somit sind sie deterministisch. Ein stochastisches Übergangsmodell für die Wahrscheinlichkeiten der Zustandsübergänge ist für Reversi und Tic Tac Toe nicht sinnvoll, da beide Spiele nicht vom Zufall abhängen und für jede Aktion in jedem Zustand nur ein einziger Zustandsübergang möglich ist. Wir werden in dieser Arbeit ausschließlich additive Gewinne für Reversi und Tic Tac Toe verwenden, denn diese beiden Strategiespiele terminieren immer, nach einer endlichen Anzahl von Aktionen, in einem Endzustand.

3.2 Optimale Taktiken

Wir definieren in diesem Abschnitt, nach Russell und Norvig [RN12, S. 757 f.], was eine optimale Taktik ist:

Eine Taktik oder 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 :

$$U^\pi(s) = E \left[\sum_{t=0}^{\infty} \gamma^t R(S_t) \right]. \quad (3.1)$$

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^* :

$$\pi_s^* = \operatorname{argmax}_{\pi} U^\pi(s). \quad (3.2)$$

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.

3.3 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]. Verfahren welche kein Wissen über bereits vorhandene Strategien verwendet sind z.B. Minimax-Suche, Alpha-Beta-Suche und Iterativ vertiefende Tiefensuche.

Im vorherigen Abschnitt haben wir gezeigt (mittels der Ausführungen von Russell und Norvig), 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 ∞ (siehe 3.2 Optimale Taktiken, Gleichung für den erwarteten Nutzen 3.1). Dementsprechend gibt eine optimale Taktik $\pi^*(s)$ für jeden Zustand s den Nachfolgezustand mit dem größtmöglichen erwarteten Nutzen an:

$$\pi^*(s) = \operatorname{argmax}_{a \in A(s)} \sum_{s'} P(s'|s, a) U(s'). \quad (3.3)$$

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:

$$U(s) = R(s) + \gamma \max_{a \in A(s)} \sum_{s'} P(s'|s, a) U(s'). \quad (3.4)$$

Diese Gleichung wird als Bellman-Gleichung bezeichnet, nach Richard Bellman(1957). Die Nutzen der Zustände - durch Gleichung 3.1 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]:

$$\hat{V}(s) = \max_a [r(s, a) + \gamma \hat{V}(\delta(s, a))]. \quad (3.5)$$

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

$$U_{i+1}(s) \leftarrow R(s) + \gamma \max_{a \in A(s)} \sum_{s'} P(s'|s, a) U_i(s'). \quad (3.6)$$

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.

3.4 Temporale Differenz Lernen (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 [RN12, S. 966 f.]:

$$U^\pi(s) \leftarrow U^\pi(s) + \alpha(R(s) + \gamma U^\pi(s') - U^\pi(s)). \quad (3.7)$$

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

3.5 Q-Lernen (TD-Q-Lernen)

Das Q-Lernen ist eine Variante des TD-Lernens und wird auch als TD-Q-Lernen bezeichnet. Die Aufgabe des TD-Q-Lernenden Agenten ist eine optimale Strategie zu entwickeln, er 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 verknüpft [RN12, S. 973]:

$$U(s) = \max_a Q(s, a). \quad (3.8)$$

Eine Nutzenfunktion $U(s)$ ist abhängig von den abgeschwächten Nutzwerten aller nachfolgenden Zustände. Ein TD-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].

$$Q(s, a) \leftarrow Q(s, a) + \alpha(R(s) + \gamma \max_{a'} Q(s', a') - Q(s, a)). \quad (3.9)$$

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.

Problemanalyse und Anforderungsdefinition

» *Das Spiel ist die höchste Form der Forschung.* «
(Albert Einstein)

In Abschnitt 4.1 wird die Aufgabenstellung genauer analysiert, die spieltheoretischen Verfahren und die Verfahren des maschinellen Lernens aus dem Grundlagenkapitel werden auf die beiden Strategiespiele Reversi und Tic Tac Toe angewendet. Abschnitt 4.3 definiert die Anforderungen, die der Softwareprototyp erfüllen sollte. Die Anforderungen beziehen sich auf die Problematik und die Grundlagen.



Abbildung 4.1 Der Agent und die Umgebung vgl. [Ert13, S. 290].

4.1 Die Problematik

Das Thema der Arbeit ist Untersuchung der Lernfähigkeit verschiedener Verfahren am Beispiel von Computerspielen. Bevor die Lernfähigkeit der Verfahren untersucht werden kann, müssen wir die Computerspiele festlegen und analysieren. Wie bereits erwähnt werden wir die Lernfähigkeit der Verfahren am Beispiel der

Strategiespiele Reversi und Tic Tac Toe untersuchen (siehe Abbildung 4.2). Eine genaue Beschreibung der Spielregeln, der Siegesbedingungen und möglicher Strategien bezüglich Heuristiken, wird in Kapitel ?? Modellierung und Entwurf erfolgen. Die zentrale Frage ist: wie kann ein Programm lernen ein Computerspiel erfolgreich zu spielen?

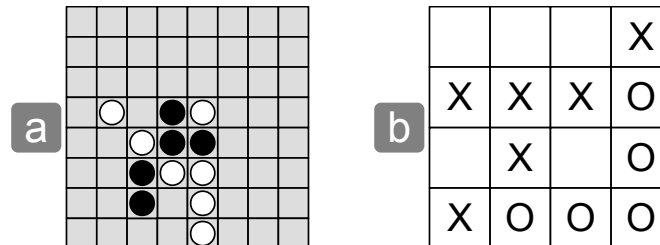


Abbildung 4.2 Tic Tac Toe und Reversi Spielzustände.

Die Spieltheorie aus Abschnitt ?? liefert gleich mehrere Ansätze diese Frage zu beantworten. Die kombinatorische Suche (Abschnitt 2.3.1 Minimax) probiert einfach alle Möglichkeiten aus und liefert die beste gefundene Möglichkeit zurück. Die reine kombinatorische Minimax Suche ist praktisch jedoch nicht anwendbar, da, wie bereits im Abschnitt Minimax beschrieben wurde, die Anzahl der Kombinationsmöglichkeiten mit der Komplexität des Ausgangsproblems exponentiell ansteigt.

Selbst mit einer Kürzung von ganzen Unterbäumen des Suchbaums, ist die Rechenzeit für realistische Probleme nicht handhabbar (siehe Abschnitt 2.3.2 Alpha-Beta-Kürzung). Das Kürzen des Suchbaums kann unter Umständen mit einer iterativ vertiefenden Tiefensuche verbessert werden (siehe Abschnitt 2.3.3 Iterativ vertiefende Tiefensuche). Die iterativ vertiefende Suche könnte Züge, z.B. in einer Tiefe von 2, sortieren. Vielversprechende Spielzüge könnten zu erst ausprobiert werden und das Alpha-Beta Verfahren könnte einen größeren Teil des Suchbaums kürzen.

Eine weitere Möglichkeit die Suche nach dem optimalen Spielzug in jeder Spielsituation zu verbessern, ist das Vermeiden von Übergängen. Ein Übergang oder Transition ist ein Spielzustand der mehrfach, an verschiedenen Stellen, in einem Suchbaum auftreten kann. Übergangstabellen und Transitions sind ausführlich in Abschnitt 2.3.4 Übergangstabellen erläutert. Eine Vermeidung dieser Übergänge könnte eine weitere Rechenzeitverringerung bewirken.

Die Heuristik oder Bewertungsfunktion ist wohl der wichtigste Leistungsfaktor aus den Verfahren der Spieltheorie (siehe Abschnitt 2.3.5 Heuristik). Eine Heuristik kann jeden Knoten des Suchbaums bewerten (Nutzwert) und nicht nur die Blätter. Dies ermöglicht es die Suche nach einer bestimmten Zeitspanne oder iterierten Tiefe abubrechen und den Spielzustand mit der besten Bewertung zurück zu geben.

Die Vorteile einer Heuristik sind die Limitierung der Zeit, die für eine Suche benötigt wird und dass das bisher beste gefundene Ergebnis zurück gegeben wird. Der große Nachteil einer Bewertungsfunktion ist, ein ermittelter Nutzwert für einen Spielzustand kann falsch sein. Die Qualität einer Heuristik ist also ausschlaggebend für die Spielstärke des Programms. Heuristiken sind zudem stark abhängig von ihrer Spielgrundlage, d.h. sowohl Reversi als auch Tic Tac Toe benötigen eigene Heuristiken, die individuell den Nutzen der Stellungen bewerten.

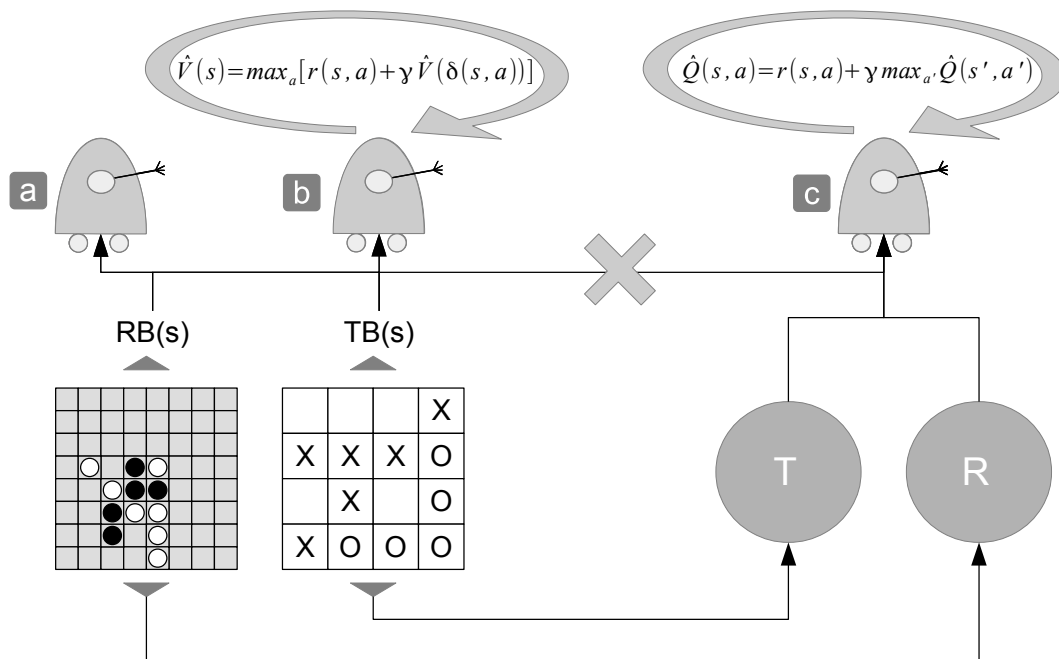


Abbildung 4.3 Die Projektproblematik.

Die drei Agenten aus Abbildung 4.3 repräsentieren drei Programme, die mit unterschiedlichen Verfahren, die selbe Aufgabe lösen. Die Aufgabe lautet: für jeden Spielzustand der Strategiespiele Reversi und Tic Tac Toe sollen die Agenten (a), (b) und (c) einen optimalen Spielzug (eine Aktion) vorschlagen, d.h. die Agen-

ten müssen eine optimale Strategie anwenden. Eine optimale Strategie verwendet für jeden Möglichen Spielzustand immer den bestmöglichen Spielzug.

Agent (a) ist ein nicht lernender Agent, der Verfahren aus der Spieltheorie anwendet und dem die Bewertungsfunktionen $RB(s)$ und $TB(s)$ zur Verfügung stehen. $RB(s)$ ist die Reversi Bewertungsfunktion, mit einem Reversi Spielzustand s als Eingabeparameter. $TB(s)$ ist die Tic Tac Toe Bewertungsfunktion, mit einem Tic Tac Toe Spielzustand s als Eingabeparameter.

Agent (b) ist ein lernender Agent, der wie der Agenten (a) über die Bewertungsfunktionen $RB(s)$ und $TB(s)$ verfügt. Dieser Agent soll die Gewichtungen (Faktoren a_x) der Bewertungsfunktionen mittels Spielerfahrung lernen. Er spielt eine festgelegte Anzahl von Spielen z.B. gegen sich selbst und verwendet eine Aktualisierungsfunktion $\hat{V}(s, a)$. Diese Funktion wurde bereits in Abschnitt ?? Wert-Iteration und Dynamische Programmierung behandelt. Zusammengefasst ist diese Funktion eine Iterationsvorschrift, die mit der Bellman-Rekursionsgleichung die Gewichtungen der Heuristik so lange anpasst, bis diese sich nicht weiter verändern lassen und gegen eine optimale Strategie konvergieren.

Agent (c) ist ein lernender Agent, der im Gegensatz zu den Agenten (a) und (b) über kein Modell der Spielwelt verfügt, also ist ihm nicht bekannt in welchen Spielzustand ihn seine Aktionen führen. Er erhält auch keine Bewertungsfunktionen $RB(s)$ und $TB(s)$ für Spielzustände s . Alles was der Agent erhält sind zwei Menge T und R . Menge T enthält alle möglichen Aktionen die der Agent in einer bestimmten Tic Tac Toe Spielsituation ausführen darf, äquivalent dazu enthält Menge R alle möglichen Aktionen die der Agent in einer bestimmten Reversi Spielsituation ausführen darf. Nach einer bestimmten Anzahl von Aktionen erhält der Agent verspätet unterschiedliche Belohnungen für einen Sieg, eine Niederlage oder ein Unentschieden des Spiels. Der Agent verwendet das Lernverfahren $\hat{Q}(s, a)$ aus Abschnitt ??, um eine optimale Strategie zu lernen.

Sind alle Agenten implementiert, können wir beginnen die Lernfähigkeit der Agenten zu vergleichen. Wie werden die beiden lernenden Agenten gegen den nicht lernenden Agenten abschneiden? Welcher Agent wird der, der am meisten gewinnt und welcher Agent wird am meisten Verlieren? Wie viel Zeit benötigen die Agenten für die Berechnung der optimalen Strategie für die beiden Strategiespiele (Training)? Diese Fragen werden in Kapitel ?? Auswertung beantwortet.

4.2 Fallbeispiel: Ein Agent im Labyrinth

Nehmen wir an es existiere folgender Agent, er kann vier Aktionen ausführen, bewege dich nach oben, unten, rechts oder links und er wird in einem ihm unbekannten Labyrinth ausgesetzt. Das Labyrinth ist die Umgebung und die Zustände der Umgebung verändern sich durch die Aktionen des Agenten (Übergänge), das heißt verändert der Agent seine Position innerhalb des Labyrinths, dann wechselt er von einem Ausgangszustand s , durch eine Aktion a , in einen neuen Zustand s' . Eine Aktion a ist immer Element der Menge aller möglichen Aktionen A . Welche Aktionen in einem bestimmten Zustand s möglich sind, wird durch die Funktion $A(s)$ oder $ACTIONS(s)$ bestimmt. Die Menge aller möglichen Zustände einer Umgebung bezeichnen wir als S , d.h. jeder einzelne Zustand s ist Element von S .

Der Agent lernt also, dass die Aktion 'bewege dich nach oben' den Ausgangszustand s in einen neuen Zustand s' transformiert. In einer deterministischen Umgebung wird der neue Zustand s' durch die Übergangsfunktion $\delta(s, a)$ bestimmt, sprich führt der Agent eine Aktion a in einem Zustand s aus, dann wird er definitiv den Zustand s' erreichen. Ist die Umgebung nicht deterministisch, dann verändert sich die Übergangsfunktion in $P(s' | s, a)$, d.h. die Umgebung bestimmt die Wahrscheinlichkeit P mit der s' erreicht werden kann, wenn im Zustand s die Aktion a ausgeführt wird. Die Funktion P kann auch ein deterministisches Modell der Welt darstellen, wenn die Wahrscheinlichkeit jedes Zustandsübergangs 100% ist, also wenn nur ein Zustandsübergang pro Zustand/Aktions-Paar möglich ist.

Führt der Agent die Aktion 'bewege dich nach oben' aus, dann ist jedoch die Zustandsveränderung abhängig von der individuellen Umgebung in der sich der Agent befindet, d.h. die Übergangsfunktion δ oder $P(s' | s, a)$, wird von der Umgebung festgelegt und nicht vom Agenten. Diese Übergangsfunktionen werden auch als Modelle der Umgebung bzw. der Welt bezeichnet. In einem Labyrinth kann der Agent nicht immer alle seiner vier Aktionen ausführen, denn er ist umringt von Mauern die seinen Aktionsradius beschränken. Würde er trotzdem eine dieser Aktion ausführen, dann verändert sich der Zustand der Umgebung nicht, denn der Agent würde sprichwörtlich "gegen die Wand laufen".

Nach einer endlichen Sequenz von Aktionen (Zustandsfolge oder Umgebungsverlauf) gelingt es dem Agenten den Ausgang des Labyrinths zu erreichen und er erhält eine numerische Belohnung (eng. reward), die Belohnung kann auch als Verstärkung (eng. reinforcement) oder Gewinn bezeichnet werden. Eine Gewinnfunktion $R(s)$ gibt die direkte Belohnung an, die der Agent erhält wenn er einen Zustand s erreicht.

Die Aktionen die der Agent bei Erreichen des Ausgangs ausgeführt hat, werden im ersten Versuch und vielleicht in den nachfolgenden Versuchen wahrscheinlich nicht optimal sein. Nicht optimal in dem Sinne, dass die Aktionssequenz nicht die kürzeste sein wird. Der Agent kann den Wert für die numerische Belohnung maximieren, indem er eine optimale Strategie entwickelt, die den kürzesten Pfad findet. Eine optimale Strategie die für jeden möglichen Zustand eindeutig definiert, welche Aktion er durchführen muss, um so wenig wie möglich Aktionen zu verwenden und den Ausgang zu erreichen. Eine genauere Beschreibung der optimalen Strategie wird in den nachfolgenden Abschnitten gegeben.

4.3 Anforderungen

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

4.3.1 Tic Tac Toe Spielumgebung

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-Lernende Agent soll zuerst diese Umgebung erkunden und lernen sich in der Umgebung zurecht zu finden, d.h. der TD-Q-Lernende 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 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 numerische Belohnung liefern. Die Belohnung soll abhängig sein vom aktuellen Spielzustand.

4.3.2 Reversi Spielumgebung

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-Lernende Agent soll zuerst diese Umgebung erkunden und lernen sich in der Umgebung zurecht zu finden, d.h. der TD-Q-Lernende 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 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 numerische Belohnung liefern. Die Belohnung soll abhängig sein vom aktuellen Spielzustand.

4.3.3 Agent des Zufalls

Der Agent des Zufalls soll den schlechtesten Spieler symbolisieren. Er soll seine Entscheidungen vollkommen zufällig treffen. In Kapitel 6 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.

4.3.4 Tic Tac Toe Heuristik Agent

Der Agent soll die in Abschnitt 2.3.5 erstellte Tic Tac Toe Heuristik und eine 2-Spielzüge vorausschauende Alpha-Beta Suche verwenden (siehe Abschnitt 2.3.3 und 2.3.2). 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 dem Lernen von Tic Tac Toe, zu beurteilen.

suggestAction(ticTacToeState): return tuple

Diese Funktion soll eine Tic Tac Toe Spielsituation übergeben bekommen. Die Funktion soll, abhängig von der erhaltenen Spielsituation, eine Aktion vorschlagen. Die

Aktion soll mittels der TicTacToe-Heuristik und einer 2-Zug Vorausschau und Alpha-Beta-Suche ermittelt werden.

4.3.5 Reversi Heuristik Agent

Der Agent soll die in Abschnitt 2.3.5 erstellte Reversi Heuristik und eine 2-Spielzüge vorausschauende Alpha-Beta Suche verwenden (siehe Abschnitt 2.3.3 und 2.3.2). Dieser Agent soll einen fortgeschrittenen Spielgegner repräsentiert, d.h. wir müssen mittels Testspielen gegen den Zufallsagenten zeigen, dass der Reversi-Heuristik Agent verhältnismäßig oft gewinnt. Dieser Agent soll in Reversi Testspielen gegen den TD-Q-Agenten antreten. Die Ergebnisse sollen dabei helfen, die Leistungsfähigkeit und Grenzen des TD-Q-Lernens, hinsichtlich dem Lernen von Reversi, zu beurteilen.

suggestAction(reversiState): return tuple

Diese Funktion soll eine Reversi Spielsituation 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.

4.3.6 Tic Tac Toe TD-Q lernender Agent

Der Agent soll, mittels des in Abschnitt ?? behandelten TD-Q-Lernens, eine Siegesstrategie für das Strategiespiel Tic Tac Toe entwickeln. Testspiele gegen den Zufallsagenten und den Tic Tac Toe Heuristik Agenten, sollen eine Untersuchung der Leistungsfähigkeit und der Grenzen des TD-Q-Lernens ermöglichen.

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.

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 und dem gegebenen Spielzustand, soll der Agent die mögliche Aktion mit dem höchsten gelernten Q-Wert zurückgeben.

4.3.7 Reversi TD-Q lernender Agent

Der Agent soll, mittels des in Abschnitt ?? behandelten TD-Q-Lernens, eine Siegesstrategie für das Strategiespiel Reversi entwickeln. Testspiele gegen den Zufallsagenten und den Reversi Heuristik Agenten, sollen eine Untersuchung der Leistungsfähigkeit und der Grenzen des TD-Q-Lernens ermöglichen.

learnReversiInXGames(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.

suggestAction(reversiState): return tuple

Die Funktion soll eine Reversi Spielsituation übergeben bekommen. Ausgehend von der Eingangsspielsituation, ist nur eine bestimmte Anzahl von Aktionen möglich. Abhängig von seinen Erfahrungen und dem gegebenen Spielzustand, soll der Agent die mögliche Aktion mit dem höchsten gelernten Q-Wert zurückgeben.

4.3.8 Testen der Agenten

Eine Testumgebung, in der alle Agenten gegeneinander Spielen. Der Zufallsagent soll in 100 Testspielen gegen den Tic Tac Toe Heuristik Agenten, den Reversi Heuristik Agenten, den Tic Tac Toe TD-Q-Lernen Agenten und den Reversi TD-Q-Lernen Agenten antreten. Der Tic Tac Toe Heuristik Agent soll 100 Testspiele gegen die drei Lernstadien des Tic Tac Toe TD-Q-Lernen Agenten spielen. Im ersten Lernstadium soll der TD-Q-Lernen Agent, in 100 Trainingsspielen gegen sich selbst, eine Strategie entwickeln. Im zweiten Lernstadium sollen es 1000 und im dritten Lernstadium 10000 Trainingsspiele sein. Äquivalent gilt dies auch für den Reversi TD-Q-Lernen Agenten und den Reversi Heuristik Agenten. Selbstverständlich spielen die Reversi Agenten, in der Reversi Spielumgebung und die Tic Tac Toe Agenten, in der Tic Tac Toe Spielumgebung.

4.4 Die Anforderungen im Klassendiagramm

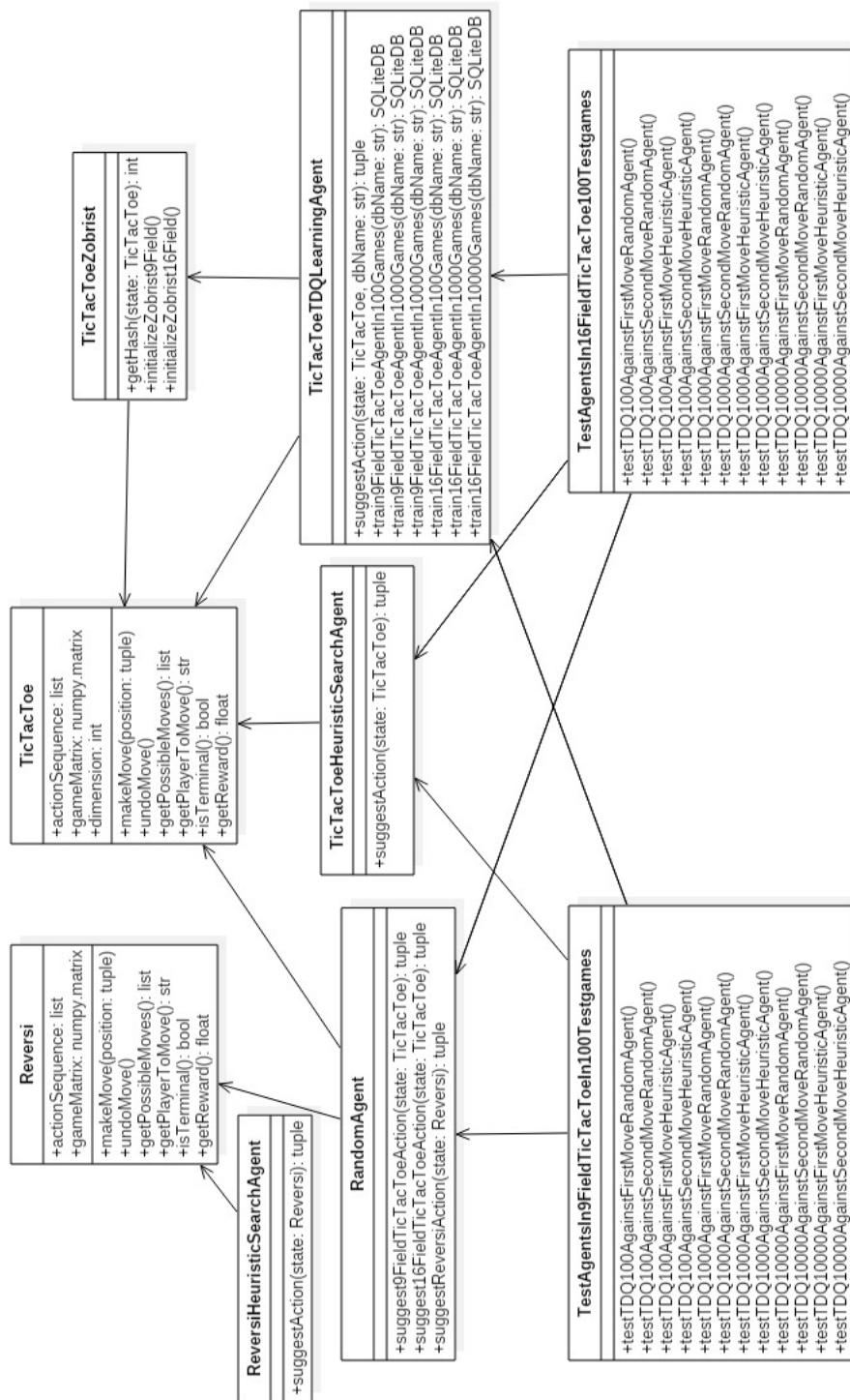


Abbildung 4.4 Klassendiagramm der Software

Algorithmen und Implementierung

Wir werden in diesem Kapitel den Algorithmus des vorausschauenden und des lernenden Agenten betrachten. Der Algorithmus des vorausschauenden Agenten ist eine Kombination aus 2 Algorithmen. Der Alpha-Beta Algorithmus [RN12, S. 212 ff.] und der Algorithmus der iterativ vertiefende Tiefensuche [RN12, S. 124 ff.]. Der vorausschauende Algorithmus benötigt zusätzlich noch eine Heuristik, um die iterativ vertiefende Suche in einer bestimmten Suchtiefe abbrechen zu können. Wir haben im Abschnitt 2.3.5 bereits Heuristiken für Reversi und Tic Tac Toe entworfen. Der Algorithmus des lernenden Agenten ist das TD-Q-Lernen [RN12, S. 973 ff.]. Das bisher noch nicht erwähnte Verhältnis von Exploration und Ausnutzung wird am ende des Kapitels erklärt.

Die Strategiespielumgebungen Reversi und Tic Tac Toe sind kommentiert und getestet. Die Implementierung dieser Strategiespielumgebungen wird in diesem Kapitel nicht erklärt. Sollte trotzdem Interesse an diesen Implementierungen bestehen, dann können diese direkt in der Programmlogik des Prototypen nachgesehen werden. 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. Ich verwenden in dieser Arbeit die Python Version 2.7.13.

5.1 Iterative Alpha-Beta Suche

Die Alpha-Beta Suche war bereits Thema des Abschnitts 2.3.2 und die iterativ vertiefende Tiefensuche des Abschnitts 2.3.3. Es folgt eine Beschreibung der Implementierung dieser beiden Algorithmen.

Sehen wir uns das Codebeispiel aus Abbildung 5.1 genauer an. Der Eingabepara-

meter 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 (Koordinate 1, Koordinate 2).

Das Codebeispiel bezieht sich auf die Tic Tac Toe Implementierung der iterative vertiefenden Alpha-Beta Suche, aber die Reversi Implementierung ist fast identisch. Einzig der Vergleich in Zeile 8 ist unterschiedlich. Würden dieser Vergleich ausgelagert werden, wären die Algorithmen identisch, denn die Funktionalitäten der Strategiespielumgebungen sind sehr ähnlich definiert (siehe Abschnitt 4.3).

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

Abbildung 5.1 Alpha-Beta iterativ vertiefende Suche

Die in der Funktion `alphaBetaIterativeDeepeningSearch(state)` verwendete Funktion `maxValue(state, alpha, beta, depth, depthBound)`, realisiert die rekursive Exploration des Suchbaums (siehe Abbildung 5.2). 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 $Beta \leq Alpha$ ist, andernfalls gib Beta zurück.

Die beiden Codebeispiele (Abbildung 5.1 und Abbildung 5.2) 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(
7             state , alpha , beta , depth + 1 , depthBound))
7         state.undoMove()
8         if alpha >= beta:
9             return beta
10    return alpha
```

Abbildung 5.2 Iteratives Suchen des maximalen Ergebnisses.

5.2 TD-Q-Lernen

Im Kapitel 3 Einführung in verstärkendes Lernen, speziell in den Abschnitten 3.4 Temporale Differenz Lernen und 3.5 Q-Lernen wurde bereits erklärt, was TD-Lernen und Q-Lernen ist. Das Thema in diesem Kapitel ist die Implementierung des TD-Q-Lernens und die Erläuterung des dafür benötigten Algorithmus.

Kurzfassung Abschnitt 3.4 Temporale Differenz Lernen: Temporale Differenz Lernen (TD-Lernen) passt die Nutzen der beobachteten Zustände an die beobachteten Übergänge an. Die Aktualisierungsregel (Gleichung 3.7) des TD-Lernens verwendet die Differenz der Nutzen aufeinanderfolgender Zustände $U\pi(s') - U\pi(s)$, daher die Bezeichnung Temporale Differenz Lernen.

Kurzfassung Abschnitt 3.5 Q-Lernen: Eine alternative TD-Methode ist das Q-Lernen, dass statt Nutzen eine Aktion/Nutzen Repräsentation lernt. Mit der Notation $Q(s,a)$ bezeichnen die Literatur den Wert der Ausführung von Aktion a im Zustand s .

```

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 Nsa[s, a]
6          Q[s, a] ← Q[s, a] + α(Nsa{s, a})
                        * (r + γmaxa' Q[s', a'] - Q[s, a])
7      s, a, r ← s', argmaxa' f(Q[s', a'], Nsa), r'
8      return a

```

Abbildung 5.3 TD-Q-Lernen Algorithmus vgl. [RN12, S. 974]

Der Q-Lernen Algorithmus (Abbildung 5.3) 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.
- **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 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 5.3) 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 ist 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 $U^\pi(s)$ (siehe ?? Optimale Taktiken Gleichung für den erwarteten Nutzen 3.1) 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 [RN12, S. 974] Die Statistik N_{sa} kann weggelassen werden, wenn eine angemessene Explorationsstrategie f verwendet wird. Mit einer angemessenen Explorationsstrategie 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 $f(\text{state})$, die einen Zustand als Eingabeparameter bekommt und eine, von diesem Spielzustand aus, mögliche Aktion zurück liefert.

```

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 5.4 Die implementierte Explorationsstrategie.

Die Funktion `explorationStrategy(state, randomFactor)` aus Abbildung 5.4 realisiert eine solche Explorationsstrategie. Sie berechnet eine auszuführende Aktion. Die Aktion wird zu einer bestimmten Wahrscheinlichkeit zufällig ausgewählt. Die 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 Trainingsspiele wird dieser Faktor um 1 erhöht, d.h. alle 100 Trainingsspiele sinkt die Wahrscheinlichkeit eine zufällige Aktion auszuwählen. 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.

Validierung

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

Der TD-Q Agent wird 3 verschiedenen 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, gegen sich selbst, trainiert wurden. Jede dieser Lernphasen wird für das 9 Spielfelder Tic Tac Toe und das 16 Spielfelder Tic Tac Toe ausgeführt.

Im nächsten Kapitel 7 werden wir erklären warum wir den TD-Q Lernenden 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 sind die wichtigsten Funktionalitäten der Strategiespielumgebungen ausgeführt und mit "asserts" getestet. Alle Tests der Strategiespielumgebungen waren positiv und könnten bei bedarf wiederholt werden.

6.1 Tic Tac Toe - 9 Spielfelder

In diesem Abschnitt testen wir den vorausschauenden Heuristik Agenten und den TD-Q Agenten gegen einen Zufallsagenten und gegeneinander, in einem 9 Spielfelder Tic Tac Toe.

6.1.1 Heuristik gegen Zufall

Der vorausschauende Heuristik Agent gewinnt mit einer 64% Gewinnquote, wenn der Zufallsagent beginnt und mit einer 83% Gewinnquote, wenn er selbst beginnt.

Der Zufallsagent beginnt:

Der vorausschauende Heuristik Agent gewinnt 64 Testspiele, der Zufallsagent gewinnt 4 Testspiele und es werden 32 unentschieden erspielt.

Der Heuristik Agent beginnt:

Der vorausschauende Heuristik Agent gewinnt 83 Testspiele, der Zufallsagent gewinnt 3 Testspiele und es werden 14 unentschieden erspielt.

6.1.2 TD-Q Lernen in 100 Trainingsspielen

- Lernzeit: ungefähr 5 Minuten
- Siege gegen den beginnenden Zufallsagenten: 37
- Niederlagen gegen den beginnenden Zufallsagenten: 59
- Siege gegen den nachziehenden Zufallsagenten: 73
- Niederlagen gegen den nachziehenden Zufallsagenten: 16
- Siege gegen den beginnenden Heuristik Agenten: 0
- Niederlagen gegen den beginnenden Heuristik Agenten: 100
- Siege gegen den nachziehenden Heuristik Agenten: 0
- Niederlagen gegen den nachziehenden Heuristik Agenten: 100

6.1.3 TD-Q Lernen in 1.000 Trainingsspielen

- Lernzeit: ungefähr 30 Minuten
- Siege gegen den beginnenden Zufallsagenten: 41
- Niederlagen gegen den beginnenden Zufallsagenten: 44
- Siege gegen den nachziehenden Zufallsagenten: 79
- Niederlagen gegen den nachziehenden Zufallsagenten: 15
- Siege gegen den beginnenden Heuristik Agenten: 0
- Niederlagen gegen den beginnenden Heuristik Agenten: 100
- Siege gegen den nachziehenden Heuristik Agenten: 0
- Niederlagen gegen den nachziehenden Heuristik Agenten: 0

6.1.4 TD-Q Lernen in 10.000 Trainingsspielen

- Lernzeit: ungefähr 180 Minuten
- Siege gegen den beginnenden Zufallsagenten: 49
- Niederlagen gegen den beginnenden Zufallsagenten: 38
- Siege gegen den nachziehenden Zufallsagenten: 92
- Niederlagen gegen den nachziehenden Zufallsagenten: 8
- Siege gegen den beginnenden Heuristik Agenten: 0
- Niederlagen gegen den beginnenden Heuristik Agenten: 100
- Siege gegen den nachziehenden Heuristik Agenten: 0
- Niederlagen gegen den nachziehenden Heuristik Agenten: 100

6.2 Tic Tac Toe - 16 Spielfelder

In diesem Abschnitt testen wir den vorausschauenden Heuristik Agenten und den TD-Q Agenten gegen einen Zufallsagenten und gegeneinander, in einem 16 Spielfelder Tic Tac Toe.

6.2.1 Heuristik gegen Zufall

Der vorausschauende Heuristik Agent gewinnt mit einer 88% Gewinnquote, wenn der Zufallsagent beginnt und mit einer 100% Gewinnquote, wenn er selbst beginnt.

Der Zufallsagent beginnt:

Der vorausschauende Heuristik Agent gewinnt 88 Testspiele, der Zufallsagent gewinnt 0 Testspiele und es werden 12 unentschieden erspielt.

Der Heuristik Agent beginnt:

Der vorausschauende Heuristik Agent gewinnt 100 Testspiele, der Zufallsagent gewinnt 0 Testspiele und es werden 0 unentschieden erspielt.

6.2.2 TD-Q Lernen in 100 Trainingsspielen

- Lernzeit: ungefähr 25 Minuten
- Siege gegen den beginnenden Zufallsagenten: 52
- Niederlagen gegen den beginnenden Zufallsagenten: 22
- Siege gegen den nachziehenden Zufallsagenten: 51
- Niederlagen gegen den nachziehenden Zufallsagenten: 42
- Siege gegen den beginnenden Heuristik Agenten: 0
- Niederlagen gegen den beginnenden Heuristik Agenten: 100
- Siege gegen den nachziehenden Heuristik Agenten: 0
- Niederlagen gegen den nachziehenden Heuristik Agenten: 100

6.2.3 TD-Q Lernen in 1.000 Trainingsspielen

- Lernzeit: ungefähr 180 Minuten
- Siege gegen den beginnenden Zufallsagenten: 60
- Niederlagen gegen den beginnenden Zufallsagenten: 23
- Siege gegen den nachziehenden Zufallsagenten: 48
- Niederlagen gegen den nachziehenden Zufallsagenten: 34
- Siege gegen den beginnenden Heuristik Agenten: 0
- Niederlagen gegen den beginnenden Heuristik Agenten: 100
- Siege gegen den nachziehenden Heuristik Agenten: 0
- Niederlagen gegen den nachziehenden Heuristik Agenten: 0

6.2.4 TD-Q Lernen in 10.000 Trainingsspielen

- Lernzeit: ungefähr 1440 Minuten
- Siege gegen den beginnenden Zufallsagenten: 49
- Niederlagen gegen den beginnenden Zufallsagenten: 32
- Siege gegen den nachziehenden Zufallsagenten: 57
- Niederlagen gegen den nachziehenden Zufallsagenten: 18
- Siege gegen den beginnenden Heuristik Agenten: 0
- Niederlagen gegen den beginnenden Heuristik Agenten: 100
- Siege gegen den nachziehenden Heuristik Agenten: 0
- Niederlagen gegen den nachziehenden Heuristik Agenten: 100

| Lernphase | TD-Q Lernender Agent gegen den Zufallsagenten | | | | | | | | | |
|---|---|--|--|----------------|-------------|--|---|---|----------------|-------------|
| | Tic Tac Toe mit 9 Spielfeldern | | | | | Tic Tac Toe mit 16 Spielfeldern | | | | |
| | 1. Lernphase nach 100 Trainingsspielen (ca. 5 Min Lernzeit) | 2. Lernphase nach 1.000 Trainingsspielen (ca. 30 Min Lernzeit) | 3. Lernphase nach 10.000 Trainingsspielen (ca. 180 Min Lernzeit) | 100 Testspiele | | 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) | 100 Testspiele | |
| Lernphase | Siege | Siege | Siege | unentschieden | Niederlagen | Siege | Siege | Siege | unentschieden | Niederlagen |
| | 37 | 41 | 49 | 15 | 38 | 52 | 60 | 49 | 17 | 32 |
| | 73 | 79 | 92 | 6 | 8 | 51 | 48 | 57 | 18 | 19 |
| Lernphase | Siege | Siege | Siege | unentschieden | Niederlagen | Siege | Siege | Siege | unentschieden | Niederlagen |
| | 59 | 44 | 49 | 15 | 38 | 22 | 23 | 49 | 17 | 32 |
| | 16 | 15 | 92 | 6 | 8 | 42 | 34 | 57 | 18 | 25 |
| TD-Q Lernender Agent gegen den Heuristik Agenten | | | | | | | | | | |
| Lernphase | Tic Tac Toe mit 9 Spielfeldern | | | | | Tic Tac Toe mit 16 Spielfeldern | | | | |
| | 1. Lernphase nach 100 Trainingsspielen (ca. 5 Min Lernzeit) | 2. Lernphase nach 1.000 Trainingsspielen (ca. 30 Min Lernzeit) | 3. Lernphase nach 10.000 Trainingsspielen (ca. 180 Min Lernzeit) | 100 Testspiele | | 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) | 100 Testspiele | |
| | Siege | Siege | Siege | unentschieden | Niederlagen | Siege | Siege | Siege | unentschieden | Niederlagen |
| Lernphase | 0 | 0 | 0 | 0 | 100 | 0 | 0 | 0 | 0 | 100 |
| | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 |
| | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Vorausschauende, nichtlernende Heuristik Agenten gegen den Zufallsagenten | | | | | | | | | | |
| Lernphase | Tic Tac Toe mit 9 Spielfeldern | | | | | Tic Tac Toe mit 16 Spielfeldern | | | | |
| | 1. Lernphase nach 100 Trainingsspielen (ca. 5 Min Lernzeit) | 2. Lernphase nach 1.000 Trainingsspielen (ca. 30 Min Lernzeit) | 3. Lernphase nach 10.000 Trainingsspielen (ca. 180 Min Lernzeit) | 100 Testspiele | | 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) | 100 Testspiele | |
| | Siege | Siege | Siege | unentschieden | Niederlagen | Siege | Siege | Siege | unentschieden | Niederlagen |
| Lernphase | 64 | 4 | 32 | 0 | 100 | 88 | 0 | 0 | 0 | 12 |
| | 83 | 3 | 14 | 0 | 100 | 100 | 0 | 0 | 0 | 0 |

Auswertung

In diesem Kapitel wollen wir die Testergebnisse des vorherigen Kapitels 6 auswerten. Die Ergebnisse der Testphase sollen uns dabei helfen die Leistungsfähigkeit und die Grenzen des TD-Q-Lernens zu beurteilen (siehe Abschnitt 7.1). Wir werden ein, in der Literatur bereits bekanntes, Problem dieses Lernverfahrens mit den Testergebnissen belegen und Lösungsmöglichkeiten besprechen. Diese Lösungsmöglichkeiten sind ebenfalls aus der Literatur entnommen und sie stellen herausragende Erfolge auf dem Gebiet des verstärkenden Lernens dar (siehe Abschnitt 7.2).

7.1 TD-Q-Lernen - Leistung und Grenzen

In diesem Abschnitt wollen wir Bezug auf die Testergebnisse des vorherigen Kapitels 6 Validierung nehmen. Die Testergebnisse sollen Aussagen über die Konvergenz und über die benötigte Rechenzeit der gelernten Strategien in den verschiedenen Testphasen ermöglichen. 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 $n \rightarrow \infty$, mit n gleich der Anzahl der Aktualisierungen des Q-Werts ($\hat{Q}_n(s, a)$).

7.1.1 TD-Q-Lernen Konvergenz (Leistungsfähigkeit)

In diesem Abschnitt analysieren wir die Testergebnisse der einzelnen Lernphasen, für 9 und 16 Spielfelder Tic Tac Toe. Wir beurteilen die Leistung der einzelnen ge-

lernten Strategien, aufgrund der Testspiele gegen den Zufallsagenten und den Heuristik Agenten. Wir wollen versuchen die oben zitierte Aussage von Wolfgang Ertel, über die Konvergenz, mit unseren Testergebnissen zu untermauern.

Die Testergebnisse des vorausschauenden Heuristik Agenten gegen den Zufallsagenten sind eindeutig. Die kleinste Gewinnquote des Heuristik Agenten erfüllt, mit 64% Gewinnchance, das Testkriterium (siehe Abschnitt ??). 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 Zufallsagent nachzieht, gewinnt der Heuristik Agent alle 100 Testspiele.

Die Testspiele der gelernten Strategien gegen den vorausschauenden Heuristik Agent sind sehr einseitig ausgefallen. Jede gelernte Strategie, egal ob für das 9 oder 16 Spielfelder Tic Tac Toe, spielt gegen die nicht gelernte Strategie des vorausschauenden Heuristik Agenten immer das selbe Spiel. Der vorausschauende Heuristik Agent führt ebenfalls immer die selben Spielzüge aus. Die folge daraus ist, beide Agenten spielen in 100 Testspielen immer die exakt gleiche Partie mit dem selben Spielergebnis. Die gelernten Strategien haben es zudem nicht geschafft, innerhalb von maximale 10.000 Trainingsspielen, eine Strategie zu lernen, die den vorausschauenden Heuristik Agenten besiegt.

Die einzige Ausnahme war die in 1.000 Testspielen gelernte Strategie, diese erreichte 100 unentschieden gegen den nachziehenden Heuristik Agent (jeweils im 9 und 16 Spielfelder Tic Tac Toe). Alle 100 Testspiele wurden von beiden Agenten exakt gleich gespielt. Für die Tests wäre es vermutlich interessanter einen weiteren Trainingsmodus für den TD-Q-Agenten zu implementieren. Ein Trainingsmodus der nicht gegen sich selbst spielt, sondern gegen Andere Agenten. Die Testergebnisse gegen den Heuristik Agenten würden dann wahrscheinlich auch variieren, dies ist jedoch kein Ziel dieser Arbeit.

Die Testergebnisse in Spielen gegen den Zufallsagenten sind ebenfalls kritisch zu betrachten, denn sie hängen offensichtlich vom Zufall ab. Wir können die Testergebnisse demnach nur berücksichtigen, wenn diese eindeutig sind, d.h. die gelernten Strategien müssten hohe Gewinnquoten und niedrige Verlustquoten erreichen. Wir versuchen trotzdem die Leistungsfähigkeit des TD-Q-Lernens, über die Testergebnisse, gegen den Zufallsagenten, herzuleiten.

Tic Tac Toe - 9 Spielfelder - Lernen gegen Zufall

Testergebnisse des TD-Q Agenten gegen den nachziehenden Zufallsagenten:

Nach Abschluss der Lernphase 1 gewinnt er 73 und verliert 16 Testspiele.
Nach Abschluss der Lernphase 2 gewinnt er 79 und verliert 15 Testspiele.
Nach Abschluss der Lernphase 3 gewinnt er 92 und verliert 8 Testspiele.

Testergebnisse des TD-Q Agenten gegen den beginnenden Zufallsagenten:
Nach Abschluss der Lernphase 1 gewinnt er 37 und verliert 59 Testspiele.
Nach Abschluss der Lernphase 2 gewinnt er 41 und verliert 44 Testspiele.
Nach Abschluss der Lernphase 3 gewinnt er 49 und verliert 38 Testspiele.

Die in 10.000 Trainingsspielen gelernte Strategie des TD-Q Agenten hat deutlich bessere Gewinn- und Verlustquoten als die in 100 Trainingsspielen gelernte Strategie. Die in 1.000 Trainingsspielen gelernte Strategie hat bessere Gewinn- und Verlustquoten als die in 100 Trainingsspielen gelernte Strategie, aber schlechtere Gewinn- und Verlustquoten als die in 10.000 Trainingsspielen gelernte Strategie.

Wir können aus den oberen Testergebnissen ableiten, dass die Leistungsfähigkeit (also die Spielstärke hinsichtlich einer höheren Gewinnquote und einer Niedrigen Verlustquote) ansteigt, je mehr Trainingsspiele der TD-Q-Lernende Agent durchführen konnte. Die gelernten Strategien nach 10.000 Trainingsspielen erbrachten die höchste Gewinnquote (92%) für ein 9 Spielfelder Tic Tac Toe, wenn der TD-Q Agent gegen den nachziehenden Zufallsagenten spielt. 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 konvertieren, 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 - 16 Spielfelder - Lernen gegen Zufall

Testergebnisse des TD-Q Agenten gegen den nachziehenden Zufallsagenten:
Nach Abschluss der Lernphase 1 gewinnt er 51 und verliert 42 Testspiele.
Nach Abschluss der Lernphase 2 gewinnt er 48 und verliert 34 Testspiele.
Nach Abschluss der Lernphase 3 gewinnt er 57 und verliert 18 Testspiele.

Testergebnisse des TD-Q Agenten gegen den beginnenden Zufallsagenten:
Nach Abschluss der Lernphase 1 gewinnt er 52 und verliert 22 Testspiele.
Nach Abschluss der Lernphase 2 gewinnt er 60 und verliert 23 Testspiele.

Nach Abschluss der Lernphase 3 gewinnt er 49 und verliert 32 Testspiele.

Wir können aus den oberen Testergebnissen ablesen, dass die Leistungsfähigkeit der gelernten Strategie nicht unbedingt ansteigt, je mehr Trainingsspiele das TD-Q-Lernverfahren absolviert. Beginnt der TD-Q Agent, dann ergibt sich, für die in 1.000 Trainingsspielen gelernte Strategie, eine Siegesquote von 60% und eine Verlustquote von 23%. Die in 10.000 Trainingsspielen gelernte Strategie erreicht nur eine Gewinnquote von 49% und eine Verlustquote von 32%. Beginnt wiederum der Zufallsagent, dann ergibt sich für die in 10.000 Trainingsspielen gelernte Strategie eine Siegesquote von 57% und eine Verlustquote von 18%. Die in 1.000 Trainingsspielen gelernte Strategie erreicht nur eine Gewinnquote von 48% und eine Verlustquote von 18%.

Die Testergebnisse für das 16 Spielfelder Tic Tac Toe zeigen keine deutliche Verbesserung der Strategien abhängig von der Anzahl der durchgeführten Trainingsspiele des TD-Q Agenten. Wir wollen nachfolgend klären, 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.

7.1.2 Fluch der Dimensionalität (Grenzen des TD-Q-Lernens)

”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 2.5 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 Toes 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 das 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 362880 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 20922789888000 Spielzustände. 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 Trainingsspiele ungefähr 180 Minuten
- 10.000 Trainingsspiele ungefähr 1440 Minuten

Die einzelnen Testergebnisse zeigen deutlich eine Rechenzeiterhöhung der Lernphasen des 9 Spielfelder Tic Tac Toe's im Gegensatz zum 16 Spielfelder Tic Tac Toe, d.h. mit steigender Dimension erhöht sich die Rechenzeit für das TD-Q-Lernen erheblich. Der TD-Q-Agent benötigt 180 Minuten um eine 9 Spielfelder Tic Tac Toe Strategie, in 10.000 Trainingsspielen, zu lernen. Dahingegen benötigt er bereits 1440 Minuten um eine 16 Spielfelder Tic Tac Toe Strategie, in ebenfalls 10.000 Trainingsspielen, zu lernen.

7.1.3 Zusammenfassung

Die Auswertung hat bereits ergeben, dass die gelernten Strategien, für das 16 Spielfelder Tic Tac Toe, nicht so Leistungsfähig sind, wie die gelernten Strategien für das 9 Spielfelder Tic Tac Toe. Grund dafür ist der enorm erhöhte Spielzustandsraum des 16 Spielfelder Tic Tac Toe's gegenüber dem 9 Spielfelder Tic Tac Toe. In 10.000 Trainingsspielen werden die ungefähr 362880 geschätzten Spielzustände des 9 Spielfelder Tic Tac Toe's wesentlich öfter besucht als in einem 16 Spielfelder Tic Tac Toe mit ungefähr 20922789888000 geschätzten Spielzuständen.

Wir haben bereits die allgemeine Konvergenzbedingung des Q-Lernens (TD-Q-Lernen) von Wolfgang Ertel zitiert. Demnach konvergiert die gelernte Strategie nur zu einer optimalen Strategie, wenn jedes Zustands-Aktions-Paar unendlich oft besucht wird. Da der geschätzte Zustandsraum des 9 Spielfelder Tic Tac Toe's viel kleiner ist, als der geschätzte 16 Spielfelder Tic Tac Toe Zustandsraum, konvergiert die gelernte Strategie annähernd für das 9 Spielfelder Tic Tac Toe in 10.000 Trainingsspielen, aber nicht für das 16 Spielfelder Tic Tac Toe. Daraus folgt: 10.000 Trainingsspiele reichen nicht aus, um die einzelnen Spielzustände des 16 Spielfelder Tic Tac Toe's für eine Konvergenz ausreichend oft zu besuchen.

Aus diesen Gründen untersuchen wir die Lernfähigkeit und Grenzen des TD-Q-Lernens nur für 9 und 16 Spielfelder Tic Tac Toe Spiele und nicht für das 64 Spielfelder Reversi. 10.000 Trainingsspiele für das Reversi Strategiespiel würden Tage oder Wochen an Rechenzeit benötigen und letztlich würden 10.000 Trainingsspiele für den Reversi Zustandsraum nicht ausreichen, um eine annähernd optimale Reversi Strategie zu lernen.

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]”

7.2 Ausblick

Im letzten Abschnitt dieser Arbeit wollen wir auf die sehr guten Lernverfahren eingehen, die Wolfgang Ertel in seiner Aussage gerade eben 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.

7.2.1 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 dein 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 wollen 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:

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

um die Q-Funktion darzustellen. Samuel verwendet für das Lernen der einzelnen Parameter ($\theta = \theta_1, \dots, \theta_n$) 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.

7.2.2 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 feststellen:

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 praktische Einführung*. 3. Aufl. Springer, 2013.
- [Har12] Peter Harrington. *Machine Learning: IN ACTION*. 1. Aufl. Manning, 2012.
- [KLM96] Leslie Pack Kaelbling, Michael L. Littman und Andrew W. Moore. „Reinforcement Learning: A Survey“. In: *Journal 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.