Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Masterarbeit

David Olszowka

Reinforcement Learning mit Prädikaten am Beispiel von 4-Gewinnt

Fakultät Technik und Informatik
Studiendepartment Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

David Olszowka

Reinforcement Learning mit Prädikaten am Beispiel von 4-Gewinnt

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

Eingereicht am: 4. August 2016

David Olszowka

Thema der Arbeit
Reinforcement Learning mit Prädikaten
am Beispiel von 4-Gewinnt

Stichworte
Reinforcement Learning, CARCASS, 4-Gewinnt, RRL, Zustandsabstraktion, prädikatenlogische Zustandsbeschreibung

Kurzzusammenfassung
Diese Arbeit beschäftigt sich mit der Entwicklung und Auswertung eines Abstraktionsalgorithmus, der automatisch ähnliche Zustände zu abstrakten Zuständen zusammenfasst. Der hier entwickelte Agent verwendet eine prädikatenlogische Zustandsbeschreibung, um das Spiel 4-Gewinnt zu spielen, und verbindet das CARCASS-Konzept mit dem entwickelten Abstraktionsalgorithmus.

David Olszowka

Title of the paper
Reinforcement Learning with predicates
using the example of Connect Four

Keywords
reinforcement learning, CARCASS, connect four, RRL, state abstraction, predicate state description

Abstract
This thesis focuses on the development and evaluation of an abstraction algorithm, which enables agents to aggregate similar states into abstract states. In order to do so, the agents, which are described here, use conjunctions of predicates as state description and combine the abstraction algorithm with the CARCASS concept. To evaluate the agents' performance, they play the game Connect Four against each other.

Inhaltsverzeichnis

1 Einleitung

1.1 Reinforcement Learning

Reinforcement-Learning (RL) bezeichnet eine Gruppe von Lernmethoden aus dem Forschungsbereich des maschinellen Lernens[16][9]. RL-Algorithmen modellieren das zu lösende Problem durch eine Einteilung in Agent und Umwelt. Der Agent soll lernen, im aktuellen Zustand aus einer Liste von möglichen Aktionen die beste Aktion (in Hinblick auf das formulierte Ziel) auszuwählen. Die Umwelt teilt dem Agenten mit, in welchem Zustand er sich befindet und vergibt für die gewählte Aktion des Agenten eine Belohnung, wenn sie zielführend war, oder eine Bestrafung, wenn sie kontraproduktiv war.

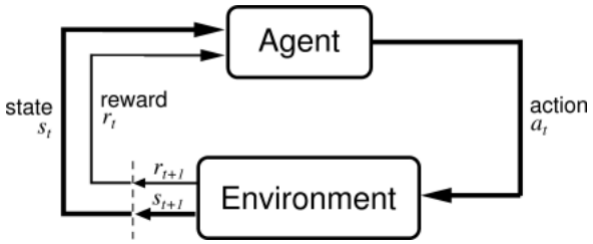

Abbildung 1.1: Interaktion zwischen Agent und Umwelt[16, S. 71]

Diese Interaktion zwischen Agent und Umwelt wird in Abbildung 1.1 veranschaulicht. Wird eine zu lernende Aufgabe auf ein solches Entscheidungsproblem zurückgeführt, dann wird diese Modellierung auch als *Markov-Decision-Process* bezeichnet (MDP)[13][5]. Um sicherzustellen, dass ein Agent die Aufgabe lernen kann, muss für den MDP die Markov-Eigenschaft erfüllt sein, die besagt, dass der Agent seine Entscheidung ausschließlich aufgrund des aktuellen Zustands treffen darf.

RL-Agenten eignen sich gut für episodische Entscheidungsprobleme, die ein definiertes Ende haben (Beispiel: Brettspiele) und nicht episodische Entscheidungsprobleme, die keinen definierten Zielzustand besitzen (Beispiel: ein Spurhalteassistent fürs Auto).

Die Grundlage der RL-Algorithmen bilden die Bellman-Gleichungen[3]:

$$V^*(s) \leftarrow \max_{a \in A} R(s,a) + \gamma \sum_{s' \in S} T(s,a,s') \cdot V^*(s') \tag{1.1}$$

$$Q^*(s,a) \leftarrow R(s,a) + \gamma \sum_{s' \in S} T(s,a,s') \cdot \max_{a' \in A} Q^*(s',a') \tag{1.2}$$

Dabei können die Agenten entweder eine V-Funktion lernen, die jedem Zustand s eine Bewertung zuweist, oder sie lernen eine Q-Funktion, die jeder Aktion a in jedem Zustand s eine Bewertung zuweist. Sind die Transitionswahrscheinlichkeiten T für alle Zustandsübergänge bekannt, oder soll der Agent nur eine gegebene Strategie bewerten, dann reicht die Verwendung der V-Funktion aus. $V(s)$ ist die erwartete, aufsummierte Belohnung, die der Agent vom Zustand s bis zum Zielzustand erhält. Ist T nicht bekannt, dann muss der Agent eine Q-Funktion lernen. $Q(s,a)$ ist dabei die erwartete, aufsummierte Belohnung, die der Agent vom Zustand s bis zum Zielzustand erhält, wenn er im Zustand s die Aktion a wählt. Die Funktionen bilden somit ein Maß für die Entfernung eines Zustands zum Zielzustand.

Die Gleichung 1.2 besagt, dass man den optimalen Q-Wert (Q^*) für eine Aktion a in einem Zustand s berechnen kann, indem man für alle möglichen Nachfolgezustände s' den Q^*-Wert der am besten bewerteten Aktion a' des Nachfolgezustands mit der Wahrscheinlichkeit, dass man in diesen Zustand kommt ($T(s,a,s')$), gewichtet aufsummiert und zu der Belohnung der gewählten Aktion ($R(s,a)$) addiert. Der Faktor γ ist dabei ein Abschwächungsfaktor, der verhindern soll, dass bei nicht episodischen Aufgaben die aufsummierte Belohnung unendlich hoch wird. Es gilt: $0 \leq \gamma \leq 1$.

Hat man Q^* errechnet, dann kann die optimale Strategie π^* daraus abgeleitet werden, indem der Agent in jedem Zustand die Aktion mit dem höchsten Q-Wert wählt. Die Bellman-Gleichungen werden von den RL-Algorithmen iterativ gelöst. Die Algorithmen unterscheiden sich hauptsächlich darin, wie sie die Gleichungen lösen und welche Informationen vorausgesetzt werden. Bei Algorithmen aus der dynamischen Programmierung werden die Transitionswahrscheinlichkeiten T und die Belohnungsfunktion R als bekannt vorausgesetzt, um die Gleichungen iterativ zu lösen. Beim *Temporal-Difference-Learning* sind die Funktionen T und R für den Agenten unbekannt und werden aus der Interaktion mit der Umwelt geschätzt.

1.2 Problemstellung

Die Modellierung von Aufgaben als MDP ist für die meisten Probleme zwar sehr einfach, führt aber häufig zu Problemen bei der praktischen Umsetzung. Im Rahmen dieser Arbeit werden die Agenten untereinander verglichen, indem sie das Spiel 4-Gewinnt gegeneinander spielen. Ziel des Spiels ist es, aus den eigenen Spielsteinen eine horizontale, diagonale oder vertikale Kette zu bilden, die aus vier oder mehr Steinen besteht, während man gleichzeitig verhindern muss, dass der Gegner eine solche Kette aus seinen Steinen bilden kann. Die Spieler werfen abwechselnd ihre Steine in die Spalten des Spielfelds. Dabei fallen die Steine auf das unterste freie Feld der gewählten Spalte.

Will man das Spiel 4-Gewinnt als MDP modellieren, dann muss für jedes Zustands-Aktions-Paar ein Q-Wert gelernt werden. Das Spielfeld besteht aus sieben Spalten, die jeweils sechs Felder hoch sind. Jedes Feld hat die möglichen Belegungen: $\{rot, gelb, leer\}$. In jedem Zug kann der Spieler in eine der sieben Spalten seinen Spielstein werfen. Das ergibt eine Höchstgrenze von $7 \cdot 3^{6 \cdot 7} \approx 7{,}7 \cdot 10^{20} \approx 2^{69}$ Q-Werten, die der Agent lernen müsste. Diese Schätzung enthält zwar viele nach den Spielregeln illegale Zustände, gibt aber die Größenordnung des Zustandsraums gut wieder. Der Zustandsraum wächst exponentiell mit der Anzahl der relevanten Eigenschaften. Das Problem ist auch bekannt unter dem Namen Zustandsraumexplosion oder *Curse-of-Dimensionality* [6]. Hinzu kommt, dass beim *Temporal-Difference-Learning* jeder Zustand viele Male vom Agenten besucht werden muss, damit die Q-Werte ausreichend nah an Q^* herankommen.

Ein weiteres Problem bei der Modellierung von 4-Gewinnt ist das sogenannte *Temporal-Credit-Assignment-Problem*. Es beschreibt die Problematik eine gute Belohnungsfunktion R für jede Problemstellung zu definieren, die jedem Zustands-Aktions-Paar einen skalaren Wert zuordnet, welcher von dem Agenten als Maß für die Wichtigkeit der Aktion verwendet werden kann. Beim Beispiel von 4-Gewinnt ist es schwer zu beurteilen, ob die ersten Züge gut, oder schlecht sind. Ist R ungünstig gewählt und der Agent erhält für einen guten Zug eine zu geringe Belohnung, dann kann das dazu führen, dass der Agent die optimale Strategie gar nicht lernen kann. Aus diesem Grund kann nur für den letzten Zug eine korrekte Bewertung für Sieg, Remis oder Niederlage vergeben werden.

Das daraus resultierende Problem wird bei der Betrachtung der nachfolgenden Update-Formel für *Q-Learning*[17][18] deutlich. Die Formel ist aus der Bellman-Gleichung (Gleichung 1.2) abgeleitet und gibt an, wie $Q(s,a)$ bei dem Zustandsübergang $s \xrightarrow{a} s'$ angepasst werden muss.

$$\begin{aligned} Q_{t+1}(s,a) &= Q_t(s,a) + \alpha \cdot \delta \\ \delta &= R(s,a) + \gamma \cdot \max_{a' \in A} Q_t(s',a') - Q_t(s,a) \end{aligned} \tag{1.3}$$

α steht für die Lernrate($0 \leq \alpha \leq 1$) und beeinflusst, wie stark neue Lernerfahrungen gewichtet werden. δ wird auch als *TD-Error* bezeichnet und ist der Korrekturwert, der zu dem aktuellen Q-Wert addiert werden muss, damit sich dieser dem optimalen Q-Wert nähert. γ kann für episodische Anwendungen (eine Episode entspricht einem Spiel) gleich 1 gesetzt werden. Ist nun $R(s,a) = 0$, weil kein Bewertungsmaß für den Zug vorhanden ist, dann reduziert sich die Update-Formel zu:

$$Q_{t+1}(s,a) = Q_t(s,a) + \alpha \cdot \left(\max_{a' \in A} Q_t(s',a') - Q_t(s,a) \right)$$

Es wird lediglich die Differenz zwischen dem aktuellen Q-Wert und dem besten Q-Wert des Nachfolgezustands durch α abgeschwächt auf den aktuellen Q-Wert addiert. Erhält der Agent dementsprechend erst nach n Zügen eine Belohnung fürs Gewinnen, muss er im nächsten Spiel die ersten $n-1$ Züge genauso spielen, damit die Belohnung für das gewonnene Spiel zum vorletzten Q-Wert propagiert wird. Damit die Belohnung beim Q-Wert des Startzustands ankommt, sind n gleiche Spiele nötig. Das führt dazu, dass der Agent sehr viele Spiele spielen muss, um eine gute Strategie zu lernen. Je mehr Zustände der Zustandsraum hat, desto stärker schränkt dieses Problem die Lerngeschwindigkeit ein.

Algorithmen, wie $Q(\lambda)$[17] oder $SARSA(\lambda)$[16] können das Problem etwas reduzieren, indem sich die Belohnung in jedem Schritt zusätzlich auf alle Vorgängerzustände auswirkt. Das grundlegende Problem ist jedoch, dass der Agent zu viele Zustände voneinander unterscheiden muss, da er über keine Abstraktionsmöglichkeit verfügt. In vielen Aufgabenstellungen gibt es eine Menge strukturell ähnlicher Zustände, die sich die gleiche optimale Aktion teilen. Da die Zustände in einem MDP für den Agenten nur eine Identität (und keine Eigenschaften) besitzen, kann der Agent die Zustände nicht nach struktureller Ähnlichkeit zusammenfassen und muss so für jeden einzelnen Zustand lernen, welches Verhalten optimal ist. Der Agent nutzt die in der Umwelt vorhandene Information über den Zustand nicht aus und zieht somit nicht den optimalen Nutzen aus seiner Lernerfahrung. Abstrahiert man jedoch vom konkreten Zustand und betrachtet Mengen von ähnlichen Zuständen, dann kann der Agent für eine Menge von

Zuständen die optimale Aktion gleichzeitig lernen und reduziert so die Anzahl der separat betrachteten Zustände. Dadurch kann der Agent mehr Erfahrung aus jeder Interaktion mit der Umwelt ziehen.

1.3 Zielsetzung

Die Zielsetzung dieser Arbeit ist es, einen lernenden Agenten für das Spiel 4-Gewinnt zu implementieren, der das *CARCASS*-Konzept[11] umsetzt und auf Zustandsmengen lernt, anstatt auf einzelnen Zuständen. Wie durch *CARCASS* vorgegeben, verwendet der Agent eine Entscheidungsliste mit prädikatenlogischen Zustandsbeschreibungen zur Definition der Zustandsmengen. Über das *CARCASS*-Konzept hinaus soll der Agent diese Entscheidungsliste durch die gesammelte Erfahrung selbstständig erweitern und so die Zustandsmengen immer weiter partitionieren, bis eine ausreichend detaillierte Modellierung der Aufgabe erreicht ist. Auf diese Weise soll die benötigte Anzahl an Zuständen auf ein notwendiges Minimum reduziert werden.

Um die Auswirkungen des Abstraktionsalgorithmus zu messen, wird der Agent gegen deterministische Strategien und lernende Agenten getestet. Dabei wird die Entwicklung der benötigten Zustände, der Gewinnrate und die Simulationsgeschwindigkeit ausgewertet.

Aufgrund der Abstraktion von konkreten Zuständen sollte der hier entwickelte Agent weniger Zustände als *Q-Learning* benötigen, um eine ähnlich gute Strategie zu repräsentieren. Da der Agent für weniger Zustände eine Aktion lernen muss, sollte er nach einer deutlich geringeren Anzahl an Episoden, eine gute Strategie lernen.
Die Abstraktion von (relevanten) Zustandsdetails kann jedoch auch dazu führen, dass der Agent keine optimale Strategie lernen kann und so auf lange Sicht von *Q-Learning* geschlagen wird. Da die Zustandsmodellierung mittels Prädikaten und der damit verbundenen Unifikation rechnerisch aufwendiger ist, als das Nachschlagen eines Q-Werts in einer Tabelle, ist zu erwarten, dass die Simulationsgeschwindigkeit abnimmt.

2 Umsetzung

2.1 Lernalgorithmus

Alle lernenden Agenten, die im Rahmen dieser Arbeit implementiert wurden, benutzen den Lernalgorithmus *Q-Learning*. Dieser Algorithmus gehört zum *Temporal-Difference-Learning* und ist aus der generellen *TD-Updateregel* aus Gleichung 2.1 abgeleitet.

$$\begin{aligned} Q_{t+1}(s,a) &= Q_t(s,a) + \alpha \cdot \delta \\ \delta &= R(s,a) + \gamma \cdot Q_t(s',a') - Q_t(s,a) \end{aligned} \tag{2.1}$$

Q-Learning interpretiert a' dabei als die am besten bewertete Aktion des Nachfolgezustands s'. Die Aktion a' kann somit von der Aktion abweichen, die im Nachfolgezustand s' tatsächlich gewählt wird. Das macht den Algorithmus zu einem *Off-Policy*-Algorithmus, denn die gewählte Aktion hat keinen Einfluss auf den Q-Wert aus dem Vorgängerzustand (s,a).

Der *On-Policy*-Algorithmus *SARSA*[14][15] wählt die tatsächlich im Nachfolgezustand s' gewählte Aktion als a'. Dadurch ist der Lernerfolg von *SARSA* abhängig von der gewählten Explorationsstrategie. Die Bewertung der Aktion im aktuellen Zustand wird durch die Wahl einer schlecht bewerteten Aktion im Nachfolgezustand verschlechtert, obwohl eine besser bewertete Aktion im Nachfolgezustand möglich wäre. Das hat zur Folge, dass eine schlechte Explorationsstrategie das Lernen einer guten Strategie bei *SARSA* verlangsamen, oder sogar gänzlich verhindern kann.
Q-Learning hat dieses Problem nicht und lernt unabhängig von der gewählten Explorationsstrategie eine optimale Strategie[18][4]. Eine schlechte Explorationsstrategie führt lediglich dazu, dass der Agent länger lernen muss, um die Strategie zu erlernen.

2.1.1 Explorationsstrategie

Damit *Q-Learning* die optimale Strategie lernen kann, muss die Explorationsstrategie sicherstellen, dass nach unendlich vielen Zustandsübergängen jede Aktion in jedem Zustand unendlich oft ausgewählt wurde [18]. Eine einfache Möglichkeit, dies sicherzustellen bietet die *ϵ-Greedy-Strategie*. Hierbei beträgt die Wahrscheinlichkeit, die beste Aktion auszuwählen $1 - \epsilon$ und mit einer Wahrscheinlichkeit von ϵ wird eine beliebige andere Aktion gewählt.

In dieser Arbeit verwenden alle Agenten $\epsilon = 0{,}001$. Bei etwa 20 Zügen pro Spiel führt dies im Durchschnitt zu einer explorierten Aktion alle 50 Spiele. Die geringe Explorationsrate stellt sicher, dass die Ergebnisse nicht durch eine zu häufige Exploration verschlechtert werden. Im Vorfeld dieser Arbeit wurden für die Bestimmung der Parameter eine Reihe von Tests durchgeführt, in denen unter anderem *Q-Learning*-Agenten mit unterschiedlicher Explorationsrate gegeneinander gespielt haben. Die Tests haben ergeben, dass eine höhere Explorationsrate vor allem dazu führt, dass *Q-Learning* mehr Zustände lernt und mehr Spiele spielen muss, um optimal zu spielen.

2.1.2 Lernrate

Bei *Q-Learning* muss die Lernrate α mit der Anzahl der Episoden monoton gegen 0 gehen, um sicherzustellen, dass die gelernte Strategie zur optimalen Strategie konvergiert.[9][11] Um diese Bedingung zu erfüllen, wird die Lernrate α für eine Episode e nach folgender Formel berechnet:

$$\alpha(e) = \alpha_0 \cdot 0{,}5^{\frac{e}{e_2}} \tag{2.2}$$

Der Parameter α_0 steht für die Lernrate in Episode 0. Der Parameter e_2 gibt an, nach wie vielen Episoden $\alpha = \frac{1}{2}\alpha_0$ gilt. Die Parameter sind frei wählbar. Für die Agenten in dieser Arbeit wurde $a_0 = 0{,}5$ gewählt, um anfangs relativ schnell neue Erfahrungen zu lernen. Da die

Gesamtanzahl der simulierten Episoden aus Zeitgründen[1] zwischen 10^6 und 10^7 liegt, wurde $e_2 = 3 \cdot 10^5$ gewählt. Dadurch geht α in genau diesem Bereich stark gegen 0.

$$\alpha(10^6) \approx 0{,}05$$
$$\alpha(10^7) \approx 5 \cdot 10^{-11}$$

2.2 Namensschema

Um die verschiedenen Variationen von Agenten im Nachfolgenden eindeutig zu benennen, wird hier das Namensschema erklärt, welches so auch in der Implementierung eingehalten wurde.

Alle Agenten enden mit dem Suffix *AI*(Artificial Intelligence), um sie in der Anwendung von menschlichen Spielern zu unterscheiden. Beispiele sind *Simple**AI***, *QLearning**AI***, *Random**AI***.

Alle Agenten, die ein Lernverhalten implementieren, enthalten das Suffix *LearningAI* (oder abgekürzt: *L*). Beispiele sind: *Q**LearningAI***, *PredicateQ**LearningAI***, sowie deren Kurzformen: *Q**L*** und *PQ**L***.

Lernende Agenten enthalten das Infix *Q* (Q-Learning), *CQ* (CARCASS mit Q-Learning) oder *TQ* (Baumstruktur mit Q-Learning), welches den zugrunde liegenden Lernalgorithmus angibt. Die Präfixe vor dem Lernalgorithmus beschreiben entweder die Zustandsmodellierung oder das charakteristische Spielverhalten. Ist kein Präfix im Namen enthalten, handelt es sich um die klassische Implementierung von *Q-Learning* mit Zeichenketten als Zustandsidentifikator (siehe Abschnitt 2.4.1) und Zahlen als Aktionen.

[1] Die im Vorfeld durchgeführten Tests haben gezeigt, dass für 10^7 Episoden bereits über 10 Stunden Rechenzeit notwendig sind. Für die aufwendigeren Agenten, die später in dieser Arbeit erklärt werden, sogar über 100 Stunden. 10^8 Episoden wären also im zeitlichen Rahmen dieser Arbeit nicht mehr auswertbar gewesen.

2.3 Architektur

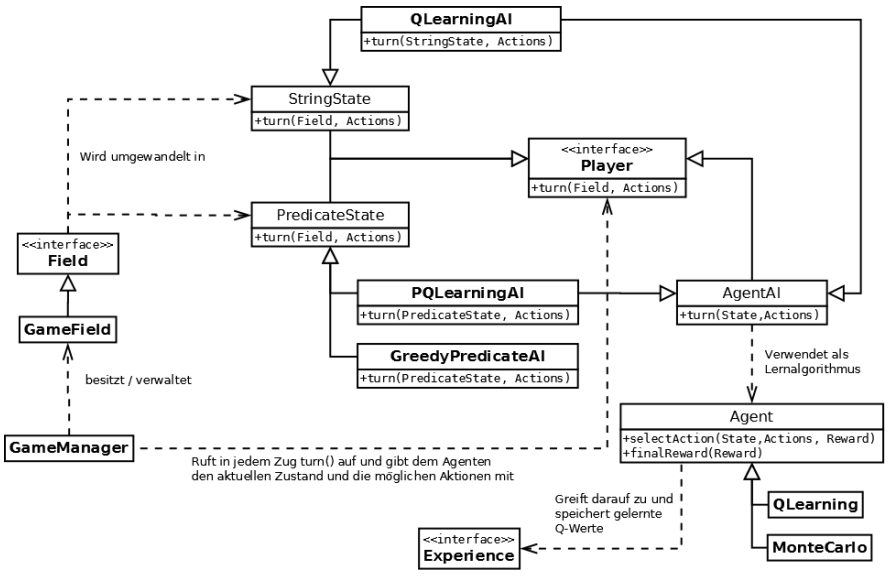

Abbildung 2.1: Eine grobe Übersicht der wichtigsten Klassen und deren Interaktionen

Für die Umsetzung des Projekts wurde die Programmiersprache Scala gewählt. Zur Darstellung der GUI und des Spielfelds wurde *Scala-Swing* verwendet und zum Speichern und Laden von Konfigurationen die Bibliothek *Spray-JSON*. Hinzu kommt die Bibliothek *log4j2*, die für das Logging zuständig ist. Der Source-Code zu diesem Projekt ist online verfügbar[2].

Abbildung 2.1 bietet einen Überblick über den Zusammenhang der wichtigsten Klassen. Der *GameManager* enthält die gesamte Spiellogik des Spiels 4-Gewinnt. Die Klasse ist dafür zuständig, die gespeicherte Konfiguration zu laden und dem Anwender die Oberfläche zu präsentieren. Beim Spielstart muss der *GameManager* die Agenten laden (wenn gespeicherte Lerndaten vorhanden sind) oder erstellen, das Spielfeld initialisieren und den Agenten die Lernparameter $(\alpha_0, e_2, \epsilon, \gamma)$ mitteilen.

Anschließend ruft er bei jedem Agenten abwechselnd die turn()-Methode auf, damit der Agent entscheiden kann, in welche Spalte er seinen Spielstein setzen will. Nach jedem Zug kontrolliert er, ob das Spiel vorbei ist und informiert die Agenten gegebenenfalls über den Ausgang des Spiels.

[2] https://gitlab.com/lSoleyl/connect4rl

Die Lerndaten werden in Abbildung 2.1 durch das *Experience*-Interface dargestellt. Die *Experience*-Implementierungen bieten den Agenten Zugriff auf die Q-Wert-Tabelle und sind zuständig dafür, die Lerndaten in eine Datei zu speichern und von dieser wieder zu laden. Für die einzelnen Zustandsmodellierungen (Zeichenketten, Prädikate) gibt es eigene Implementierungen des *Experience*-Interfaces.

Die Erfahrung wurde gezielt von dem eigentlichen Agenten getrennt, welcher durch das *Agent*-Interface repräsentiert wird. Die Überlegung dahinter war, dass die Lerndaten keine Eigenschaft eines Agenten sind. Stattdessen benutzt der Agent konkrete Lerndaten und trifft auf Basis dieser Lerndaten Entscheidungen. Diese Trennung führt dazu, dass die Lernerfahrung, die ein Agent benutzt, von mehreren Agenten gleichzeitig verwendet werden kann. So kann ein Agent gegen sich selbst trainieren, indem zwei Agenten gegeneinander spielen, die auf den gleichen Lerndaten arbeiten.

Das *Agent*-Interface repräsentiert alle implementierten Lernalgorithmen der Anwendung. Zur Veranschaulichung sind hier *QLearning* und *MonteCarlo* dargestellt. Prinzipiell kann die Anwendung beliebig um weitere Lernalgorithmen erweitert werden. Die Implementierung der Algorithmen wurde von den Agenten selbst getrennt, um nicht den gleichen Algorithmus mit minimalen Anpassungen (an die Zustandsbeschreibung) mehrfach zu implementieren. So wird *QLearning* sowohl von *QLearningAI*, als auch *PredicateQLearningAI* als zugrunde liegender Lernalgorithmus verwendet. Der einzige Unterschied der beiden Implementierungen ist die Darstellung des Zustands. Das *Agent*-Interface ist generisch und kann für alle möglichen Zustands- und Aktionstypen verwendet werden. Das Interface wurde nach dem generellen Agenten-Schema aus Abbildung 1.1 entworfen.

Listing 2.1: Der Hauptbestandteil des *Agent*-Interfaces

```
trait Agent[S,A,V] {
  def selectAction(state:S, reward:Double, possibleActions:List[A]):A
  def finalReward(reward:Double)
}
```

Alle lernenden Algorithmen erben von der abstrakten Klasse *AgentAI*. Diese Klasse implementiert das *Player*-Interface, indem *turn()* durch einen entsprechenden Aufruf von *selectAction()* des Agenten implementiert wird. Durch die starke Trennung der Zuständigkeiten lassen sich die lernenden Agenten so aus den einzelnen Klassen ”zusammenstecken” ohne konkret etwas zu implementieren. Die gesamte Implementierung der Klasse *QLearningAI* ist beispielsweise im Listing 2.2 ungekürzt gezeigt.

Dabei ist *StringExperience* eine Implementierung des *Experience*-Interfaces, mit Zeichenketten als Zuständen (wird in Abschnitt 2.4.1 erklärt). Die Klasse *StringState* wandelt *Field*, welches der Agent vom *GameManager* erhält, in eine Zustandsbeschreibung durch Zeichenketten um. *HeaderV1* ist eine Erweiterung für *Experience*. Diese speichert in den Lerndaten zusätzlich, wie viele Spiele der Agent bisher gespielt hat, und zu welchem Agententypen die Lerndaten gehören.

Listing 2.2: *QLearningAI* wird durch "zusammenstecken" der Grundbestandteile implementiert

```
case class QLearningAI(experience:StringExperience with HeaderV1)
                      extends AgentAI(new QLearning(experience))
                      with StringState
```

2.4 Zustandsmodellierung

2.4.1 Durch Zeichenketten

Klassische RL-Algorithmen treffen keine Annahmen über die Beschaffenheit des Zustands und betrachten den Zustand lediglich als Identifikator, um ihn von anderen Zuständen zu unterscheiden. Die Darstellung des Zustands als Zeichenkette ist deswegen eingeführt worden, um den Speicherplatz für die Q-Wert-Tabelle des Agenten zu minimieren und den Identitätsvergleich von Zuständen zu beschleunigen. Würde der Agent den Zustand als 2-dimensionales Array speichern mit Elementen aus (0 = leer, 1 = gelb, 2 = rot), so müsste der Agent für einen Zustand $6 \cdot 7 = 42$ Zahlen speichern. Bei dem Vergleich von zwei Zuständen wären entsprechend 42 einzelne Vergleiche notwendig.

Während mehrerer Testläufe wurde beobachtet, dass die Spiele durchschnittlich nach 22 Zügen beendet wurden. In den meisten Fällen blieb über die Hälfte der Felder während des gesamten Spiels frei. Beschreibt man den Zustand durch die Zugfolge (mit Spaltennummern 1-7), die notwendig ist, um den Zustand von einem leeren Feld aus zu erreichen (beginnend beim gelben Spieler), so reduziert man die Zustandsbeschreibung auf weniger als die Hälfte.

Der Zustand aus Abbildung 2.2 lässt sich auf diese Weise durch 6 Zahlen ausdrücken. Das Beispiel zeigt jedoch auch ein Problem bei der Modellierung von Zuständen durch Zugfolgen. Der gleiche Zustand lässt sich durch viele unterschiedliche Zugfolgen erreichen. Das führt

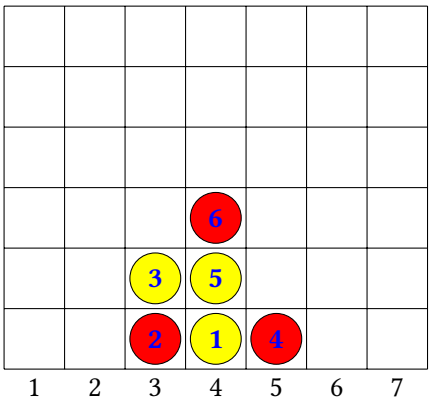

Abbildung 2.2: Beispielzustand mit der Zustandsbeschreibung $4, 3, 3, 5, 4, 4$. Auch die Zugfolge $4, 3, 4, 4, 3, 5$ führt zum gleichen Zustand. Erst die Festlegung, die Steine von unten nach oben und dann von links nach rechts zu legen, ordnet diesem Zustand eindeutig die Beschreibung $4, 3, 3, 5, 4, 4$ zu.

dazu, dass es für jede einzelne Spielfeldkonfiguration mehrere Zustände gibt, die alle separat gelernt werden. Dies würde den Zustandsraum von 4-Gewinnt weiter vergrößern und das Lernen verlangsamen.

Durch die Zugfolge wurde mehr Information in die Zustandsbeschreibung aufgenommen, als der Agent benötigt. Zusätzlich zum Zustand wurde die Historie des Spiels in den Zustand aufgenommen. Da 4-Gewinnt die Markov-Eigenschaft erfüllt, muss die Historie für den Lernprozess nicht in dem Zustand modelliert werden. Das Problem lässt sich auflösen, indem eine eindeutige Reihenfolge festgelegt wird, in welcher die Zugfolge aus dem Feld rekonstruiert wird. In diesem Fall wurde festgelegt, dass die Spielsteine von unten nach oben und dann von links nach rechts in die Zugfolge aufgenommen werden. Um die Zustände einfacher miteinander zu vergleichen, wird die Zugfolge intern durch eine Zeichenkette, die aus Spaltennummern besteht, dargestellt. Damit erhält der Agent eine kompakte Zustandsbeschreibung, die den Zustandsraum nicht vergrößert.

2.4.2 Durch Prädikate

Das *CARCASS*-Konzept gibt vor, dass die Zustände durch Konjunktionen von Prädikaten beschrieben werden. Analog zur Zustandsbeschreibung durch Features, steht jedes Prädikat für eine bestimmte Eigenschaft, die im aktuellen Zustand erfüllt ist. Im Gegensatz zu Features

haben Prädikate Argumente. Die Argumente des Prädikats stellen die Verbindung zwischen der aggregierten Zustandsmenge und den möglichen Aktionen her. Seien beispielsweise $p/1$ und $q/1$ beliebige Prädikate, dann gilt $p(X) \wedge q(Y)$ in den Zuständen:

$$s_1 = p(1) \wedge q(2)$$
$$s_2 = p(2) \wedge q(2)$$
$$s_3 = p(1) \wedge q(5)$$

Dadurch definieren die Prädikate einen "abstrakten Zustand" $S_1 = p(X) \wedge q(Y)$, welcher die Zustandsmenge $\{s_1, s_2, s_3\}$ zu einem Zustand zusammenfasst. Die möglichen Aktionen in diesem abstrakten Zustand werden ebenfalls durch Prädikate dargestellt, die von einer oder mehreren Variablen abhängen können. Beispiele wären Aktionen, wie $put\_on(X, Y)$ oder $select(X)$. Bei der Definition der Zustandsprädikate und Aktionen muss darauf geachtet werden, dass die Aktionen für die Zustände $\{s_1, s_2, s_3\}$ jeweils eine ähnliche Bedeutung haben. Sind diese Voraussetzungen erfüllt, kann *Q-Learning* angewendet werden, um *Q*-Werte für die Aktionen dieser abstrakten Zustände zu lernen.

Es lässt sich zeigen, dass der Agent zwar immer zu einer Strategie konvergiert, diese ist jedoch nur dann optimal, wenn sich die aggregierten Zustände eines abstrakten Zustands ausreichend ähneln. Dies wird damit begründet, dass das Lernen von Q-Werten für Zustandsmengen einer Mittelwertbildung über die Q-Werte dieser Zustandsmenge entspricht.[11, S.259]
Werden nun zwei Zustände zu einem abstrakten Zustand zusammengefasst, obwohl deren Aktionen gänzlich unterschiedlich bewertet sein müssten, dann kann der Agent für diesen abstrakten Zustand keine optimale Aktion lernen, da die fälschlich aggregierten Zustände nun einen einzigen Zustand bilden.

Bis auf *QLearningAI* verwenden die in dieser Arbeit implementierten Agenten eine Zustandsbeschreibung, die aus einer Konjunktion der folgenden Prädikate besteht. Die Prädikate $1k/1, 2k/1, 3k/1, 4k/1$ sind für einen Zustand erfüllt, wenn der Spieler eine Kette der Länge $1, 2, 3, 4$ legen kann, indem er seinen Spielstein in die Spalte wirft, die als Argument an das entsprechende Prädikat gebunden ist. Die Prädikate $1b/1, 2b/1, 3b/1, 4b/1$ sind für einen Zustand erfüllt, wenn der Spieler eine gegnerische Kette der Länge $1, 2, 3, 4$ verhindern kann, indem er seinen Spielstein in die Spalte wirft, die als Argument an das entsprechende Prädikat gebunden ist. Da ein einzelner Stein bereits eine Kette der Länge 1 darstellt, würde jeder Zustand für jede Spalte, für die keine anderen Prädikate gelten, die Prädikate $1k/1$ und $1b/1$ enthalten. Dies

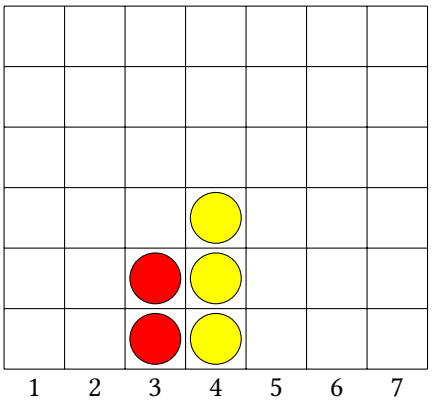

Abbildung 2.3: Beispiel für eine 4-Gewinnt Spielsituation

würde dazu führen, dass die Zustandsbeschreibung für jeden Zustand viele Prädikate enthält, die strategisch nicht relevant sind. Die Prädikate für die Ketten der Länge 1 und 2 werden deshalb nur dann in die Zustandsbeschreibung aufgenommen, wenn weniger als drei Prädikate der Länge 3 oder 4 im Zustand enthalten sind. Bei 4-Gewinnt lassen sich auch Ketten der Länge 5, 6 und 7 legen. Da alle Ketten mit einer Länge ≥ 4 im Spiel strategisch gleichbedeutend sind, werden diese nicht untereinander unterschieden und durch die Prädikate $4k/1$ und $4b/1$ abgebildet.

Der Zustand aus Abbildung 2.3 würde für den roten Spieler wie folgt beschrieben werden:

$$s = 2k(2) \wedge 3k(3) \wedge 2b(3) \wedge 4b(4) \wedge 2b(5) \tag{2.3}$$

Der rote Spieler kann hier zum Beispiel einen Stein in Spalte 4 werfen, um eine Viererkette des Gegners zu verhindern ($4b(4)$) oder in Spalte 3 werfen, um gleichzeitig eine Dreierkette zu legen und eine Zweierkette des Gegners zu verhindern ($3k(3) \wedge 2b(3)$).

Für die optimale Strategie für 4-Gewinnt aus [1] werden Zustandseigenschaften betrachtet wie: nützliche und nutzlose Bedrohungen, gerade und ungerade Bedrohungen und der Zugzwang. Diese recht komplizierten Zustandseigenschaften wurden nicht in die Zustandsmodellierung aufgenommen. Hierdurch soll geprüft werden, ob der Agent die fehlenden strategischen Details der Zustandsbeschreibung durch Lernen ausgleichen kann und ähnlich gut spielen kann, wie eine Strategie, die diese Details miteinbezieht. Zudem entspricht die Modellierung der Zustandsbeschreibung in etwa dem, wie menschliche Spieler das Spielfeld betrachten. Diese prüfen hauptsächlich, in welchen Spalten sie Ketten verlängern oder blockieren können.

2.5 GreedyPredicateAI (Greedy-Strategie)

Dieser Agent spielt nach einer sinnvollen, deterministischen Strategie und dient als Referenzagent an dem der Lernerfolg der lernenden Agenten gemessen wird. Da die Strategie deterministisch ist, sind Tests gegen diesen Agenten aussagekräftiger, als Tests gegen eine Zufallsstrategie. Zudem spielt dieser Agent deutlich besser, als eine Zufallsstrategie und stellt somit eine größere Herausforderung für die Agenten dar.

Der Agent betrachtet das Spielfeld durch die in Abschnitt 2.4.2 beschriebene prädikatenlogische Zustandsmodellierung. Die Prädikate des aktuellen Zustands werden nach der Wichtigkeit sortiert und nur die wichtigsten Prädikate im weiteren Schritt betrachtet. Die Wichtigkeit der Prädikate wurde hierfür festgelegt durch die folgende Reihenfolge: $4k > 4b > 3k > 3b > 2k > 2b$. Im nachfolgenden Beispiel betrachtet der Agent im Zustand S nur die Menge R.

$$S = 3k(1) \wedge 3k(4) \wedge 3b(1) \wedge 2k(3) \wedge 2b(4) \tag{2.4}$$

$$\Rightarrow R = \{3k(1), 3k(4)\} \tag{2.5}$$

Jedes Prädikat aus der Menge R bindet eine Aktion, die für den Agenten zu folgender Zustandsänderung führt: Eine Zweierkette wird zu einer Dreierkette verlängert. Da beide Prädikate in dieser Hinsicht das gleiche Ergebnis haben, wird zwischen den beiden Prädikaten entschieden, indem die Aktion gewählt wird, die am nächsten zur Mitte des Spielfelds liegt. Die Mitte wird deshalb bevorzugt, da die Spielfeldmitte mehr Möglichkeiten bietet, Ketten in verschiedene Richtungen zu legen, als der Spielfeldrand. In dem eben genannten Beispiel würde sich der Agent für die Aktion $A = 4$ entscheiden.
Sollte die Zustandsbeschreibung keine Prädikate enthalten, was im Startzustand der Fall ist, so wählt der Agent von allen möglichen Aktionen in diesem Zustand die Aktion, die am nächsten zur Mitte liegt.

2.6 PredicateQLearningAI (PQL)

Dieser Agent erhält von der Umwelt die gleiche Zustandsbeschreibung, wie *GreedyPredicateAI* und lernt in diesem Zustandsraum mittels *Q-Learning*. Der Agent soll sicherstellen, dass durch Lernen eine bessere Strategie in diesem Zustandsraum gefunden werden kann, als die vorgegebene *Greedy-Strategie*. Wenn keine bessere Strategie gelernt werden kann, dann

folgt daraus, dass die verwendete Zustandsmodellierung strategisch relevante Details nicht abbildet und die Beschreibung um zusätzliche Prädikate ergänzt werden muss, um gut spielen zu können.

Um von den konkreten Zuständen zu den abstrakten Zuständen zu kommen, ersetzt dieser Agent lediglich die Argumente der Prädikate durch Variablen und lernt für die möglichen Aktionen in diesem abgeleiteten Zustand Q-Werte. Am nachfolgenden Beispiel ist kurz veranschaulicht, wie der Agent aus einem konkreten Zustand einen abstrakten Zustand ableitet.

$$\begin{aligned} s &= 3k(1) \wedge 3k(2) \wedge 3b(1) \wedge 2k(4) \\ \Rightarrow S &= 3k(A) \wedge 3k(B) \wedge 3b(A) \wedge 2k(C) \end{aligned} \tag{2.6}$$

Dieser Zustand S würde nun alle Zustände aggregieren, die sich auf diese Weise ableiten lassen und der Agent würde Q-Werte für die Aktionen $\{A, B, C\}$ lernen. Da Zustände möglich sind, in denen keine Prädikate gelten und somit keine Aktion anbieten, gibt es in jedem Zustand zusätzlich die Aktion $?$. Wird diese Aktion ausgewählt, wählt der Agent aus allen Spalten, die nicht an ein Prädikat gebunden sind, eine zufällige Spalte aus.

Ehe der Agent einen neuen Zustand ableitet, prüft er zuerst, ob es sich bereits um einen bekannten Zustand handelt. Dies prüft der Agent, indem er seine Liste von abgeleiteten Zuständen der Reihe nach durchgeht und nach dem ersten abgeleiteten Zustand sucht, der durch korrektes Einsetzen von Werten für die Variablen exakt mit dem aktuellen Zustand übereinstimmt. Kann der Agent einen solchen Zustand finden, lernt der Agent für diesen Zustand Q-Werte, anstatt einen neuen Zustand abzuleiten.

Dieses Zustandsmatching kann als eine striktere Form der Unifikation angesehen werden. Bei der Unifikation werden die Variablen der zu testenden Aussage (Bezogen auf das Beispiel 2.6 ist dies der abgeleitete Zustand S) so belegt, dass die Prädikate dieser Aussage eine Teilmenge der Prädikate der Wissensbasis (hier der konkrete Zustand s) bilden. Bezogen auf die Menge der Prädikate gilt für die Unifikation $S \subseteq s$. Das hier angewandte Matching verlangt zusätzlich, dass alle Prädikate der Wissensbasis s auch in S vorkommen und somit $S = s$ gelten muss. Dadurch wird sichergestellt, dass der Agent für alle strukturell unterschiedlichen Zustände,

unterschiedliche abstrakte Zustände ableitet und lernt. Dies wird kurz an dem folgenden Beispiel veranschaulicht.

$$\begin{aligned} S &= 2k(A) \\ s_1 &= 2k(1) \\ s_2 &= 3k(2) \wedge 2k(3) \\ s_3 &= 4k(4) \wedge 3b(1) \wedge 2k(5) \end{aligned} \tag{2.7}$$

Das Beispiel 2.7 zeigt, wieso der Agent keine Unifikation für die Zustandssuche nutzen kann. Hat in diesem Beispiel der Agent den abstrakten Zustand S bereits abgeleitet, würde dieser abstrakte Zustand S die Zustände s_1, s_2, s_3 abdecken und der Agent würde für diese Zustände keinen neuen abstrakten Zustand ableiten. Dennoch unterscheiden sich die dargestellten Spielsituationen stark voneinander. Damit der Agent diese Zustände voneinander unterscheiden und für jeden der Zustände ein anderes Verhalten lernen kann, muss der Agent für die Identifizierung des aktuellen Zustands dieses Matching verwenden.

2.6.1 Anpassung der Zustandsbeschreibung

Das Ziel dieses Agent war es sicherzustellen, dass durch Lernen eine bessere Strategie gefunden werden kann, als die *Greedy-Strategie*. Die ursprüngliche Zustandsmodellierung enthielt nur die Prädikate $4k, 4b, 3k, 3b$, die Prädikate der Länge 2 wurden nicht als strategisch relevant angesehen und nur dann in die Zustansbeschreibung aufgenommen, wenn sie keine Prädikate der Länge ≥ 3 enthielt.

Mit dieser Zustandsmodellierung war es *PQL* jedoch nicht möglich, zuverlässig gegen die *Greedy-Strategie* zu gewinnen. Ein regulärer *Q-Learning* Agent (*QLearningAI*) konnte im Gegensatz dazu nach kurzer Lernphase in 100% der Spiele gegen die *Greedy-Strategie* gewinnen. Die Ursache konnte zum einen sein, dass durch die Zustandsbeschreibung mehrere unterschiedliche Zustände zum gleichen abstrakten Zustand zusammengefasst werden, obwohl diese jeweils unterschiedliche Aktionen erfordern. Zum anderen könnte das Problem auch dadurch verursacht werden, dass die optimale Aktion in einem bestimmten Zustand für den Agenten nicht existiert, da sie von keinem Prädikat als Argument gebunden wird. Der Agent könnte eine solche Aktion nur über die ?-Aktion zufällig auswählen. In Abbildung 2.4 und 2.5 sind zwei Strategien abgebildet, die *QLearningAI* gelernt hat, um gegen die *Greedy-Strategie* zu gewinnen.

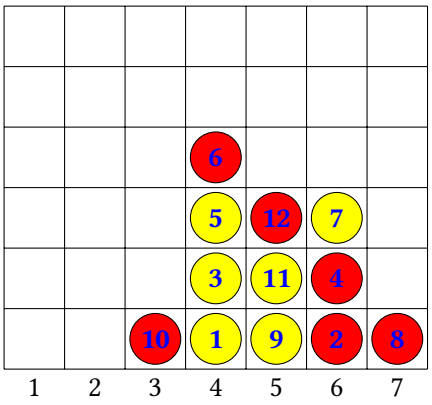

Abbildung 2.4: Spielsituation, in welcher die *GreedyPredicateAI*(gelb) gegen *QLearningAI*(rot) verliert. *GreedyPredicateAI* spielte den ersten Spielzug.

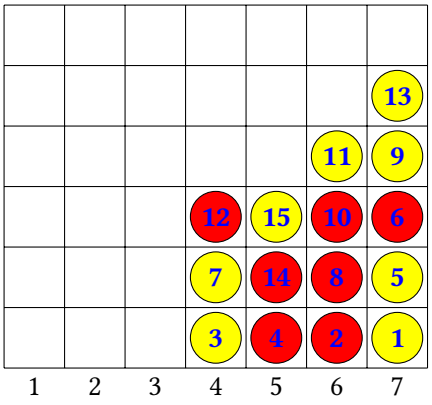

Abbildung 2.5: Spielsituation, in welcher die *GreedyPredicateAI*(rot) gegen *QLearningAI*(gelb) verliert. Diesmal spielte *QLearningAI* den ersten Spielzug.

Betrachtet man die Spielzüge aus Abbildung 2.4, dann sieht man, dass *PQL* im zweiten Schritt mit $s_2 = 2b(3) \wedge 2b(4) \wedge 2b(5)$ keine Möglichkeit hat, die korrekte Aktion 6 zu wählen, da sie an kein Prädikat gebunden ist. In Abbildung 2.5 sieht man, dass *PQL* auch keine Möglichkeit hat, im ersten Zug mit $s_1 = \emptyset$ die korrekte Aktion 7 zu wählen. Des Weiteren kann der Agent im siebten Schritt mit $s_7 = 3b(6)$ nicht die optimale Aktion 4 wählen und im neunten Schritt mit $s_9 = 3k(4) \wedge 3b(6)$ nicht die optimale Aktion 7 wählen.

Die Probleme wurden durch die folgende Erweiterung des Zustandsraums behoben: Falls der aktuelle Zustand weniger als **drei** Prädikate der Länge ≥ 3 enthält, werden zusätzlich die Prädikate $2k, 2b$ und $1k$ in die Zustandsbeschreibung aufgenommen. Mit dieser Erweiterung

war *PQL* in der Lage, eine Strategie zu lernen, die in 100% der Fälle gegen die *Greedy-Strategie* gewonnen hat.

2.6.2 Performance

Da die Zustände im Vergleich zu *QLearningAI* keine reinen Identifikatoren darstellen, musste die zugrunde liegende Datenstruktur für die Q-Wert-Tabelle angepasst werden. Bisher konnte eine *Map* verwendet werden, die einen schnellen Zugriff auf die Q-Werte zu einem Zustand lieferte und nur *Q*-Werte für die Zustände abspeicherte, die tatsächlich besucht wurden. Da bei diesem Agenten für alle abstrakten Zustände geprüft werden muss, ob sie mit dem konkreten Zustand gematched werden können, wurde die *Map* durch eine Liste ersetzt.

Sowohl die *Map*, als auch die Liste waren von der Performance nicht optimal und erreichten eine Simulationszeit von durchschnittlich $130\frac{Spiele}{s}$ (im Spiel gegen *QLearningAI*). Im Vergleich dazu erreicht *QLearningAI* $3500\frac{Spiele}{s}$. Um die Simulationsgeschwindigkeit zu verbessern, wurde statt der Liste eine *Queue* verwendet, da Zustände, die häufig besucht werden, meist am Anfang des Spiels entdeckt werden, wie zum Beispiel der Startzustand. Dies erhöhte die durchschnittliche Simulationsgeschwindigkeit auf $200\frac{Spiele}{s}$.

Die letzte Optimierung zielt darauf ab, die am häufigsten besuchten Zustände, immer am Anfang der *Queue* zu halten und somit die Anzahl der notwendigen Vergleiche für das Matching zu minimieren. Dazu wird mitgezählt, wie oft ein Matching fehlgeschlagen ist und wie oft ein Zustand besucht wurde. Bei $100\,000$ fehlgeschlagenen Matchings werden die Zustände in der *Queue* danach sortiert, wie oft sie besucht wurden. Der Wert $100\,000$ wurde experimentell ermittelt und hat die größte Verbesserung bewirkt. Mit dieser letzten Optimierung erreicht *PQL* eine akzeptable Simulationsgeschwindigkeit von $892\frac{Spiele}{s}$.

2.7 DynamicCARCASSQLearningAI (DCQL)

Dieser Agent implementiert im Gegensatz zu *PQLearningAI* einen vollständigen *CARCASS*, welcher zusätzlich um einen im Rahmen dieser Arbeit entwickelten Abstraktionsalgorithmus erweitert wurde. Das *CARCASS*-Konzept sieht zusätzlich zur Darstellung von Zuständen durch Konjunktionen von Prädikaten vor, dass der Agent eine Entscheidungsliste mit abstrakten

Zuständen hält, die den Zustandsraum vollständig partitioniert. Diese Entscheidungsliste soll vom Designer vorgegeben werden und kann zum Beispiel wie folgt aussehen:

$$S_1 : 4k(X) \Longrightarrow \{X, ?\}$$
$$S_2 : 4b(X) \Longrightarrow \{X, ?\}$$
$$S_3 : 3k(X) \wedge 3b(Y) \Longrightarrow \{X, Y, ?\}$$
$$S_4 : true \Longrightarrow \{?\}$$

Kommt der Agent in einen Zustand s, dann werden die Prädikate von s wie eine Prolog-Wissensbasis betrachtet. Der Agent versucht die abstrakten Zustände der Entscheidungsliste, der Reihe nach mit den Prädikaten von s zu unifizieren. Der erste abstrakte Zustand, der sich mit s unifizieren lässt, ist der Zustand, in dem sich der Agent momentan befindet. Für die Aktionen dieses Zustands aktualisiert der Agent in diesem Schritt die Q-Werte. Der letzte Zustand der Entscheidungsliste ist ein sogenannter *Catch-All-State* und muss immer $true \Longrightarrow \{?\}$ sein, um sicher zu stellen, dass jeder Zustand durch einen abstrakten Zustand abgedeckt wird.

Durch die Unifikation kann ein *CARCASS*-Agent von der Struktur eines Zustands abstrahieren und nur die relevanten Prädikate berücksichtigen. So werden beispielsweise die Zustände $s_1 = 4k(2) \wedge 2k(1), s_2 = 4k(3), s_3 = 4k(4) \wedge 4b(3)$ alle zu einem abstrakten Zustand S_1 aggregiert, da $4k(X)$ in all diesen Zuständen gilt. Diese Zustandsbeschreibung aggregiert alle Zustände, in denen eine Viererkette gelegt werden kann zu einem abstrakten Zustand S_1. Im Gegensatz dazu hat *PQLearningAI* durch das reine Ersetzen von Variablen ausschließlich von konkreten Spalten des Zustands abstrahiert.

Ein Problem der fest vorgegebenen Zustandsliste im *CARCASS* ist, dass der Agent einerseits relativ langsam lernt, wenn sie zu detailliert ist und zu viele Zustände enthält. Nimmt der Entwickler jeden denkbaren Zustand in die Zustandsliste auf, dann findet keine Abstraktion statt, da jeder abstrakte Zustand der Zustandsliste genau einem konkreten Zustand im Zustandsraum entspricht. Der Agent müsste dadurch mehr Q-Werte lernen, als nötig. Andererseits kann der Agent keine gute Strategie lernen, wenn sie nicht ausreichend detailliert ist und zu wenige Zustände enthält. Im zweiten Fall werden mehrere Zustände zu einem abstrakten Zustand aggregiert, zwischen denen der Agent unterscheiden müsste, um optimal zu spielen. Um dieses Problem zu lösen, wurde im Rahmen dieser Arbeit ein Algorithmus entwickelt, der die Zustandsliste selbstständig erweitert, bis die Aufgabe ausreichend detailliert durch die Zustandsliste abgebildet wird.

Ausgangspunkt für den Algorithmus ist eine Zustandsliste mit wenigen Zuständen. Im ersten Schritt muss entschieden werden, woran der Agent erkennt, dass ein neuer Zustand in die Zustandsliste eingefügt werden muss und an welcher Stelle. Die Idee dazu stammt aus Algorithmen für konstruktive neuronale Netze (auch *CNN*). *CNNs* sind plastische neuronale Netze, die mit einer minimalen Struktur beginnen und diese so lange modifizieren, bis die zu lernende Abbildung ausreichend genau gelernt wurde. Der Vorteil dieser Netze ist es, dass sich der Entwickler im Vorfeld keine Gedanken über eine angemessene Netzstruktur machen muss. Zu kleine Netze können die Aufgabe nicht lernen und zu große Netze tendieren zum *Overfitting*, sie spezialisieren sich also zu stark auf die Trainingsdaten. Zwei Ansätze aus diesem Bereich sind zum einen *Dynamic-Node-Creation*[2] und *Meiosis*[8]. *Dynamic-Node-Creation* fügt Neuronen in ein *Hidden-Layer* ein, wenn der Gesamtfehler des Netzes über einem festgelegten Schwellwert liegt. *Meiosis* ersetzt Neuronen durch zwei neue Neuronen, wenn deren Kantengewichte stark schwanken.

In Analogie zu den schwankenden Kantengewichten aus den *CNNs* betrachtet dieser Algorithmus die Schwankung des *TD-Errors* (δ aus Formel 1.3). Schwankt δ für ein bestimmtes Zustands-Aktions-Paar, dann folgt daraus, dass der Agent für die gleiche Aktion in dem gleichen Zustand manchmal belohnt und manchmal bestraft wird. Dieser Zustand aggregiert somit Zustände, die unterschiedlich behandelt werden müssten, damit der Agent effektiv spielen kann. Der Agent summiert also alle δ-Werte für jedes Zustands-Aktions-Paar betragsmäßig auf und sobald ein Wert einen festgelegten Schwellwert überschreitet, wird versucht, einen neuen Zustand in die Zustandsliste einzufügen.
Um diesen neuen Zustand zu bestimmen, sammelt der Agent für alle abstrakten Zustands-Aktions-Paare auf, in welchem konkreten Zustand sich der Agent befunden hat, welche Aktion tatsächlich gewählt wurde und welchen *TD-Error* (δ) der Agent dafür erhalten hat. In Tabelle 2.1 ist dies an einem Beispiel für das Zustands-Aktions-Paar $(true, ?)$ veranschaulicht.

Tabelle 2.1: Beispieldaten, die für das Zustands-Aktionspaar $(true, ?)$ gesammelt wurden. Aus diesen Informationen leitet der Algorithmus einen neuen abstrakten Zustand ab, der in die Zustandsliste eingefügt wird.

Zustand	*Aktion*	δ	*prim. Prädikate*	*sek. Prädikate*
$3k(2) \wedge 3b(1)$	3	12	$\emptyset$	$\emptyset$
$3b(2) \wedge 2k(5)$	2	20	$3b(X)$	$2k(X+3)$
$3b(7) \wedge 2k(2)$	7	-15	$3b(X)$	$2k(X-5)$
$3b(3) \wedge 2k(6)$	3	10	$3b(X)$	$2k(X+3)$
$3k(2) \wedge 2k(2)$	2	20	$3k(X) \wedge 2k(X)$	$\emptyset$

Für jeden Zustand in der Liste werden zunächst die primären Prädikate bestimmt. Das sind alle Prädikate des Zustands, deren Argument mit der gewählten Aktion übereinstimmt. Hat ein Eintrag keine primären Prädikate (1.Zeile der Tabelle), dann ist es nicht möglich, die gewählte Aktion mit einem der Prädikate der Zustandsbeschreibung in Verbindung zu bringen, was zur Folge hat, dass der Agent die Aktion nicht lernen kann. Der Agent kann ausschließlich zwischen den Aktionen wählen, die von den Prädikaten vorgegeben werden. Zustände, aus denen sich keine primären Prädikate ableiten lassen, werden deswegen nicht weiter betrachtet. Anschließend werden die sekundären Prädikate bestimmt, indem die Argumente der restlichen Prädikate relativ zur gewählten Aktion angegeben werden. Diese relativen Werte werden später zu *Constraints* für die Variablen dieser Prädikate.

Nun werden die Einträge nach den primären Prädikaten gruppiert und diese Gruppen werden dann nach dem betragsmäßig aufsummierten δ absteigend sortiert. Innerhalb dieser Gruppen werden die Einträge nochmals nach den sekundären Prädikaten gruppiert und diese werden wiederum nach dem betragsmäßig aufsummierten δ absteigend sortiert. Nach dieser mehrstufigen Sortierung ist die Kombination aus primären und sekundären Prädikaten, die den größten Einfluss auf den schwankenden *TD-Error* hatte, am Anfang. Tabelle 2.2 zeigt die Beispieldaten aus Tabelle 2.1 wie eben beschrieben sortiert. Die Tabelle zeigt zusätzlich die daraus abgeleiteten neuen abstrakten Zustände für die Zustandsliste.

Tabelle 2.2: Gruppierte und sortierte Daten aus Tabelle 2.1, sowie der daraus abgeleitete abstrakte Zustand. $\sum |\delta_{prim}|$ steht für das aufsummierte $|\delta|$ bezüglich der primären Prädikate und $\sum |\delta_{sek}|$ für das aufsummierte $|\delta|$ der Kombination aus primären und sekundären Prädikaten.

$\sum \lvert\delta_{prim}\rvert$	$\sum \lvert\delta_{sek}\rvert$	*prim. Prädikate*	*sek. Prädikate*	*abgeleiteter Zustand*
45	30	$3b(X)$	$2k(X+3)$	$3b(X) \wedge 2k(A) \lvert A = X+3$
45	15	$3b(X)$	$2k(X-5)$	$3b(X) \wedge 2k(A) \lvert A = X-5$
20	20	$3k(X) \wedge 2k(X)$	$\emptyset$	$3k(X) \wedge 2k(X)$

Der erste abgeleitete Zustand aus dieser Liste, der noch nicht Teil der Entscheidungsliste ist, wird in die Entscheidungsliste vor dem Zustand eingefügt der in diesem Schritt betrachtet wurde. In diesem Beispiel war das der Zustand $true$. Wenn die bisherige Zustandsliste nur diesen Zustand enthielt, dann würde sie nach diesem Ableitungsschritt wie folgt aussehen:

$$S_1 : 3b(X) \wedge 2k(A) | A = X+3 \Longrightarrow \{X, A, ?\}$$
$$S_2 : true \Longrightarrow \{?\}$$

Abschließend werden die bisher aufsummierten δ-Werte für alle Zustands-Aktions-Paare auf 0 zurückgesetzt, da deren Werte durch die Veränderung der Zustandsliste die Schwankungen nicht mehr korrekt wiedergeben. Durch diesen Schritt wurde die Menge an Zuständen reduziert, die von S_2 aggregiert wird. Bevor S_1 in die Entscheidungsliste aufgenommen wurde, hat S_2 alle Zustände des Zustandsraums aggregiert. Nach dem Einfügen von S_1 werden aber alle Zustände, für die $3b(X) \wedge 2k(A)|A = X + 3$ gilt, von S_1 aggregiert und alle verbleibenden von S_2. Da S_1 die Zustände aggregiert, die bei S_2 zu hohen Schwankungen von δ geführt haben, wird durch diesen Schritt auch effektiv die δ-Schwankung von S_2 reduziert. Je mehr neue Zustände aus S_2 abgeleitet werden, desto weiter sinkt die δ-Schwankung, die gleichzeitig das Auswahlkriterium dafür darstellt, aus welchem Zustand ein neuer Zustand abgeleitet werden muss. Das führt dazu, dass der Algorithmus nach einigen Iterationen keine neuen Zustände mehr ableitet, weil die δ-Schwankung den Schwellwert für kein Zustands-Aktions-Paar mehr überschreitet. Die minimale Zustandsliste, die noch gutes Lernen ermöglicht, steht dann fest.

Es gibt grundsätzlich zwei mögliche Fehlerfälle in welchen der vorgeschlagene Algorithmus keine neuen Zustände ableiten kann. Der erste Fehlerfall tritt ein, wenn die Tabelle nur Einträge enthält, zu denen sich keine primären Prädikate finden lassen. Da diese Einträge keine Verbindung zwischen gewählter Aktion und Zustandsbeschreibung zulassen, kann daraus kein neuer Zustand abgeleitet werden. Dies passiert nur dann, wenn in allen Einträgen die Aktion ? gewählt wurde und sich die daraus resultierende Aktion an kein Prädikat der Zustandsbeschreibung binden lässt. Einerseits kann dies ein Hinweis darauf sein, dass die Zustandsbeschreibung erweitert werden sollte, sodass die gewählte Aktion durch ein neues Prädikat abgebildet wird. Andererseits kann die Ursache dafür sein, dass der Abstraktionsalgorithmus durch einen zu niedrigen Schwellwert zu häufig ausgelöst wird und nur wenige Einträge für das Zustands-Aktions-Paar vorhanden sind.
Der zweite Fehlerfall tritt ein, wenn zwar neue Zustände ableitbar wären, diese aber alle bereits Teil der Entscheidungsliste sind. In dem Fall wird ebenfalls kein neuer Zustand abgeleitet. Auch dieser Fall kann durch einen zu niedrigen Schwellwert verursacht werden. Ansonsten deutet dieser Fehler darauf hin, dass der abstrakte Zustand, aus welchem sich kein neuer Zustand ableiten ließ, mehrere unterschiedliche Zustände zu einem Zustand aggregiert. Das bedeutet, dass in einem Zustand der Entscheidungsliste eine Aktion gleichzeitig gut und schlecht bewertet wird und somit der δ-Wert schwankt. Das Problem ließe sich dadurch lösen, dass versucht wird, die unterschiedlichen Zustände, die fälschlicher Weise aggregiert wurden zu identifizieren und Prädikate in die Zustandsbeschreibung einzuführen, mit deren Hilfe der Agent die Zustände voneinander unterscheiden kann.

Um dem Agenten etwas Wissen über die Umwelt mitzugeben, wird er mit folgender Zustandsliste initialisiert:

$$S_1 : 4k(X) \Longrightarrow \{X, ?\}$$
$$S_2 : 4b(X) \Longrightarrow \{X, ?\}$$
$$S_3 : true \Longrightarrow \{?\}$$

2.7.1 Parameterbestimmung

Der eben beschriebene Abstraktionsalgorithmus lässt sich an einigen Stellen anpassen und das Lernverhalten dadurch verändern. Einer dieser Parameter ist der Schwellwert ab welchem die betragsmäßig aufsummierten δ-Werte den Ableitungsschritt auslösen. Dieser Schwellwert wird im folgenden als δ_{limit} bezeichnet. Unabhängig von der konkreten Problemstellung in welcher der Agent lernen soll, muss δ_{limit} von den verteilten Belohnungen abhängen. Je weiter die minimale Belohnung r_{min} und die höchste Belohnung r_{max} auseinander liegen, desto mehr kann ein Q-Wert während der Lernphase schwanken. Um diese Differenz zu berücksichtigen wurde der Parameter wie folgt bestimmt: $\delta_{limit} = f \cdot |r_{max} - r_{min}|$. Um den bestmöglichen Wert für den Faktor f zu bestimmen, haben *DCQL* und *GreedyPredicateAI* für die Werte $1, 2, 5, 10, 20$ jeweils 10 mal 100 000 Spiele gegeneinander gespielt und der Mittelwert der Gewinnrate und der gelernten Zustände wurde auf einem Graphen abgetragen.

In Abbildung 2.6 wird erkennbar, dass es keinen eindeutig optimalen Wert gibt. Stattdessen ergibt sich ein Trade-Off zwischen schnell lernen, gut zu spielen und mit möglichst wenig Zuständen gut spielen. Je kleiner der Wert für f ist, desto häufiger werden neue Zustände abgeleitet und desto schneller hat der Agent eine ausreichend große Zustandsliste, um gegen die Greedy-Strategie zuverlässig zu gewinnen. Je höher der Wert für f ist, desto seltener werden neue Zustände abgeleitet und desto mehr Tabelleneinträge hat der Agent im Ableitungsschritt zur Verfügung, da der Zustand häufiger besucht werden musste. Für die weiteren Tests wurde $f = 10$ gewählt. Nur $f = 1$ kommt deutlich schneller zu einer Gewinnrate von 100%, benötigt dafür jedoch das dreifache an Zuständen.

Eine weitere Möglichkeit, den Algorithmus zu variieren bietet das Sortierkriterium bei der mehrstufigen Sortierung der Tabelleneinträge. Ein weiteres denkbares Kriterium ist die Häufigkeit der Elemente, nach denen sortiert wird, um so eine Zustandsbeschreibung abzuleiten,

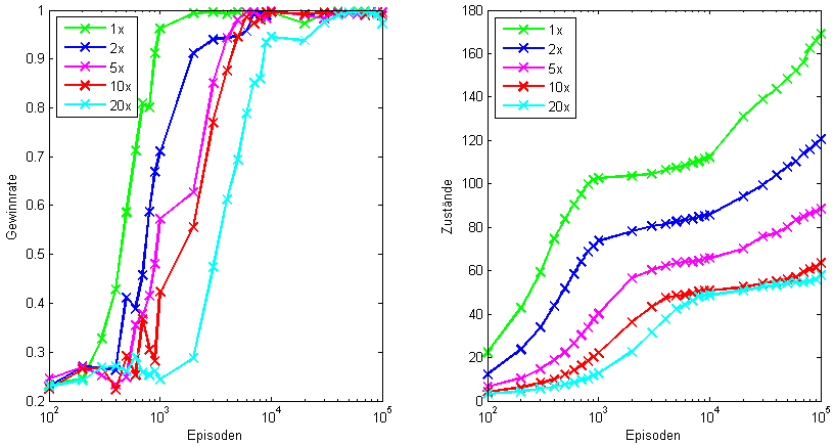

Abbildung 2.6: Auswirkung der verschiedenen δ_{limit}-Werte auf das Lernverhalten. Die Graphen zeigen den Mittelwert aus 10 mal 100 000 Spielen pro Parameterwert.

die möglichst viele Zustände abdeckt. Dadurch reduziert sich in jedem Ableitungsschritt die Anzahl der Zustände, die vom ursprünglichen Zustand aggregiert wird und damit auch die δ-Schwankung und der Algorithmus würde ebenfalls zu einer (minimalen) Zustandsliste kommen, in der keine neuen Zustände mehr abgeleitet werden.

Ebenfalls denkbar ist einen Schritt vor der Sortierung, alle Einträge mit negativem δ zu streichen und nur nach positiven δ-Werte zu sortieren, um so einen Zustand abzuleiten, der hauptsächlich positive Belohnungen erhält, während der ursprüngliche Zustand die Zustände aggregiert, die hauptsächlich negative Belohnungen erhalten. Auch dies sollte langfristig zu einer Reduzierung der δ-Schwankung führen und den Abstraktionsalgorithmus enden lassen.
Ebenso denkbar ist es, die Sortierreihenfolge umzukehren. In Abbildung 2.7 und 2.8 sieht man die Auswirkung der verschiedenen Sortierkriterien auf das Lernverhalten. Die Graphen repräsentieren wieder den Durchschnitt aus 10 mal 100 000 Spielen gegen die Greedy-Strategie.

In Abbildung 2.7 kann man auf den ersten Blick erkennen, dass die Sortierung in aufsteigender Reihenfolge (gekennzeichnet durch ein $+$ am Ende) zu einem drastischen Anstieg in der Anzahl der Zustände führt. Dieses Ergebnis ergibt Sinn, wenn man berücksichtigt, wie der Abstraktionsschritt arbeitet. Durch eine aufsteigende Sortierung der Einträge werden neue Zustände zuerst aus den Einträgen abgeleitet, die den geringsten Einfluss auf die δ-Schwankung hatten. Das führt dazu, dass der neu abgeleitete Zustand die δ-Schwankung nur minimal reduziert und der Ableitungsschritt aufgrund der verbliebenen hohen δ-Schwankung nach wenigen Episoden für diesen Zustand erneut durchgeführt wird. Die Sortierung in aufsteigender Reihenfolge

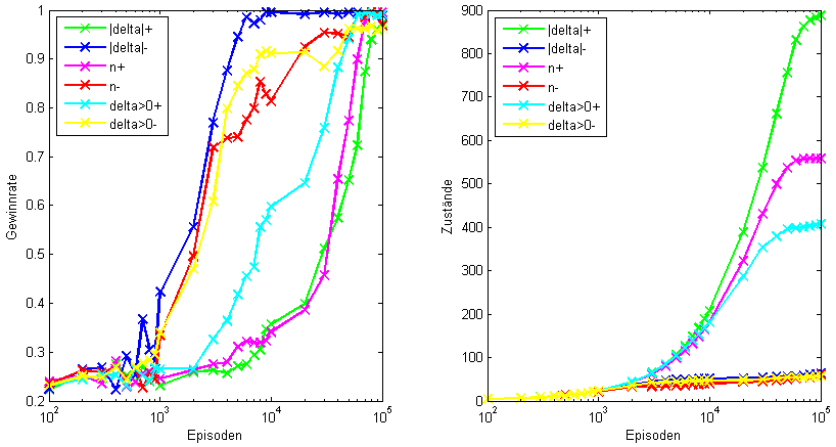

Abbildung 2.7: Die Auswirkung verschiedener Sortierkriterien auf das Lernverhalten. Die Graphen zeigen den Mittelwert aus 10 mal 100 000 Spielen pro Kriterium. Ein $+$ am Ende der Bezeichnung steht für aufsteigend sortierte Werte und ein $-$ für absteigend sortierte Werte.

widerspricht somit der Grundidee das Ableitungsschrittes, die δ-Schwankung möglichst schnell zu reduzieren.

Die Rate mit der neue Zustände erlernt werden, hat einen direkten Einfluss auf den Lernerfolg des Agenten. Kommt der Agent in einen neu erlernten Zustand, dann sind die Q-Werte für jede Aktion dieses Zustands gleich bewertet und der Agent muss jede Aktion mehrfach ausprobieren, um entscheiden zu können, welche Aktion in diesem Zustand zielführend ist. Lernt der Agent viele neue Zustände in kurzer Zeit, dann ist die Wahrscheinlichkeit dafür, in einen neuen Zustand zu kommen sehr hoch und der Agent muss häufig unbekannte Aktionen ausprobieren. Lernt der Agent hingegen nur langsam neue Zustände, dann kommt er nur selten in neue Zustände und kann im Normalfall bereits gelerntes Wissen anwenden.
Die Sortierung in aufsteigender Reihenfolge führt also zu einem Anstieg der gelernten Zustände und dadurch auch zu einem insgesamt schlechteren Lernverhalten.

Abbildung 2.8 zeigt ausschließlich Graphen für die Sortierung nach den Kriterien in absteigender Reihenfolge. Dadurch lassen sich die Unterschiede besser erkennen. Von den Sortierkriterien in absteigender Reihenfolge liefert $|\delta|$, also das betragsmäßige Aufsummieren der δ-Werte, die besten Ergebnisse. Der Agent lernt damit bereits nach 6 000 Episoden eine optimale Strategie, was bei n (Häufigkeit der Einträge) erst nach 60 000 Episoden der Fall ist. Mit $\delta > 0$ (nur

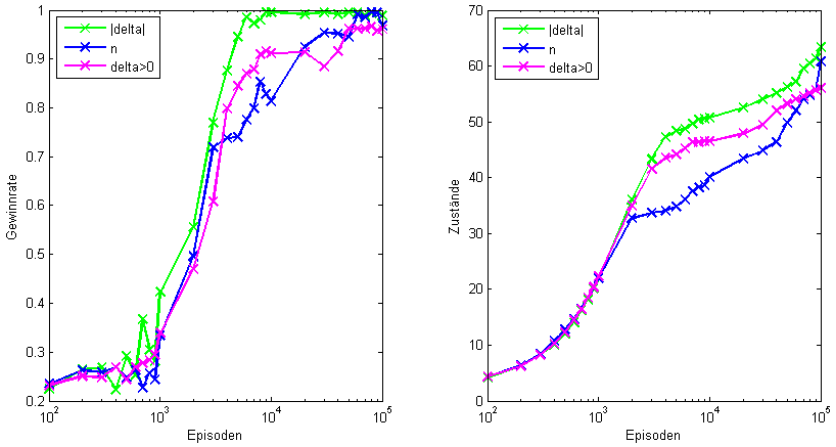

Abbildung 2.8: Die Auswirkung verschiedener Sortierkriterien auf das Lernverhalten. Die Graphen zeigen den Mittelwert aus 10 mal 100 000 Spielen pro Kriterium. Hier nur die Kriterien in absteigender Sortierreihenfolge

positive Einträge) erreicht der Agent dagegen innerhalb der 100 000 Episoden keine Gewinnrate über 96%.

Trotz der Unterschiede lernt der Agent für alle Kriterien nach den 100 000 Episoden nur 55 bis 65 Zustände. Mit dem $|\delta|$-Kriterium gewinnt der Agent nach nur 6 000 Episoden zuverlässig gegen die Greedy-Strategie und hat zu dem Zeitpunkt nur 48 Zustände. Dass die Agenten noch weiterhin neue Zustände lernen und die Gewinnrate nie genau 100% erreicht, liegt an der gewählten Explorationsrate von $\epsilon = 0{,}001$, die dazu führt, dass der Agent im Schnitt alle 1 000 Züge (≈ 50 Spiele) einen nicht optimalen Spielzug ausprobiert, um möglicherweise eine effektivere Strategie zu entdecken.

3 Ergebnisse

3.1 Vergleich aller Agenten gegen GreedyPredicateAI

Um zu beurteilen, welche Vor- und Nachteile der implementierte Agent im Vergleich zu den bisherigen Implementationen hat, wurden die Agenten *DCQL* (Abschnitt 2.7), *PQL* (Abschnitt 2.6) und ein normaler *Q-Learning*-Agent im Spiel gegen die Greedy-Strategie getestet. Die Ergebnisse sind in Abbildung 3.1 dargestellt.

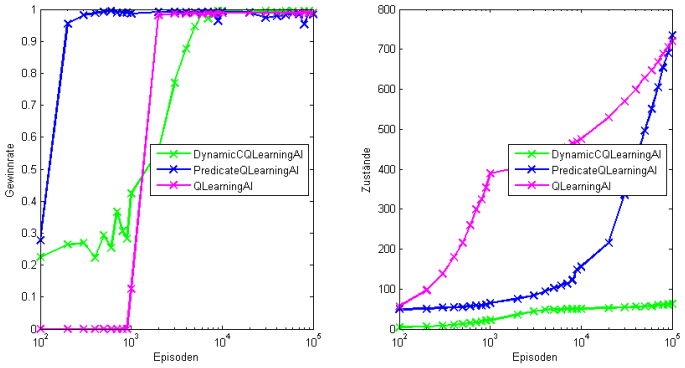

Abbildung 3.1: Vergleich der Agenten im Spiel gegen die Greedy-Strategie. Der Graph zeigt den Mittelwert aus 10 mal 100 000 Spielen.

Dieser Test zeigt, dass der Agent *DCQL* gegen eine deterministische Strategie deutlich langsamer lernt, als ein einfacher *Q-Learning*-Agent. Da *PQL* am schnellsten zu einer optimalen Strategie kommt, liegt dies jedoch nicht an der Modellierung der Zustände durch Prädikate, sondern an dem Abstraktionsalgorithmus, der erst eine Mindestmenge an Erfahrung sammeln muss, ehe der erste neue Zustand abgeleitet wird. *DCQL* benötigt wie erwartet die wenigsten Zustände, um eine optimale Strategie zu erlernen und lernt auch durch spätere Exploration nicht so schnell neue Zustände.

3.2 Direktvergleich mit QLearningAI

Um festzustellen, wie gut der Agent mit lernenden Gegnern zurecht kommt, wurde der Agent zum einen gegen *Q-Learning* getestet und zum anderen gegen *PQL* (Abschnitt 3.3). Abbildung 3.2 zeigt jeweils das Lernverhalten von *DCQL* und *PQL* im Spiel gegen einen *Q-Learning*-Agenten. Die Abbildung zeigt zusätzlich einen Graphen für $DCQL_{f1}$, hierbei handelt es sich um *DCQL* mit dem Schwellwertparameter $f = 1$ (Abschnitt 2.7.1). Der Agent wurde zusätzlich getestet, da er im Vergleich gegen die Greedy-Strategie zwar mehr Zustände gelernt hat als $DCQL_{f10}$, dafür schneller eine gute Strategie gelernt hat. Die Agenten haben jeweils 10 000 000 Spiele gegen einen *Q-Learning*-Agenten gespielt, sodass *Q-Learning* ausreichend Zeit hat, eine erfolgreiche Strategie gegen die Agenten zu lernen.

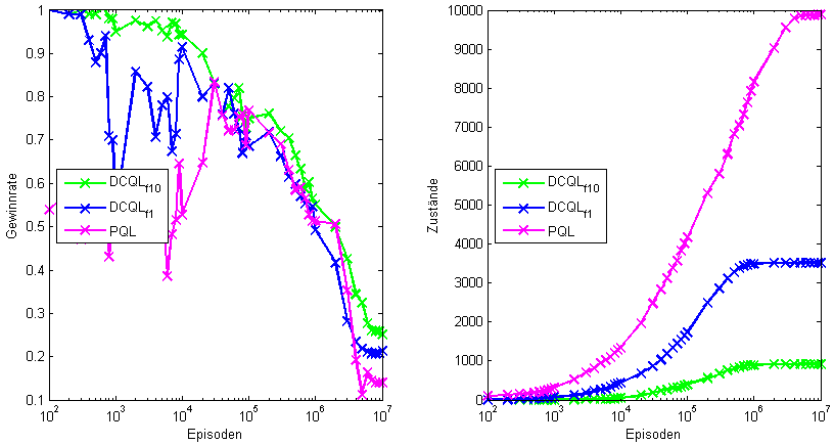

Abbildung 3.2: Vergleich der Agenten im Spiel gegen *Q-Learning*. $DCQL_{f10}$ ist hier der normale *DCQL*-Agent. Jeder Agent hat einmal 10 000 000 Spiele gegen *Q-Learning* gespielt.

Alle Agenten spielen ungefähr die ersten 1 000 000 Spiele besser als *Q-Learning*, da ihnen ihr kleinerer Zustandsraum schnelleres Lernen ermöglicht. Gleichzeitig gehen durch den kleineren Zustandsraum auch Informationen über das Spiel verloren, was dazu führt, dass sie keine optimale Strategie lernen können und *Q-Learning* schließlich zuverlässig besser spielt. Was aus der Abbildung nicht hervorgeht ist, dass *Q-Learning* für das Spiel gegen *PQL* und *DCQL* etwa 1 200 000 Zustände benötigt hat und gegen $DCQL_{f1}$ etwa 800 000. Im Vergleich dazu hat *PQL* etwa 9 900 Zustände benötigt und *DCQL* nur 900 ($DCQL_{f1}$ benötigt hingegen 3 500 Zustände).

Die Zustandsmodellierung mit Prädikaten bietet den Agenten somit die Möglichkeit, mit weniger Zuständen und nach weniger Spielen eine akzeptable Strategie zu lernen und lange Zeit besser zu spielen als *Q-Learning*. Dadurch, dass *DCQL* nicht mit einer leeren Zustandsliste beginnt, besitzt der Agent zu Anfang mehr Informationen über das Spiel und kann schneller zu einer guten Strategie gegen *Q-Learning* kommen als *PQL*. Gleichzeitig sorgt der Abstraktionsmechanismus dafür, dass der Agent langfristig mit weniger Zuständen auskommt als *PQL*. Hinzu kommt, dass *DCQL* mit den wenigen Zuständen nach 10 000 000 Spielen mit 25% immer noch eine höhere Gewinnrate erzielt als *PQL* mit 14%. Die Ursache für die bessere Spielweise von *DCQL* ist vermutlich die Verwendung der *Contraints*, die es *DCQL* ermöglichen, Zustände genauer voneinander zu unterscheiden.

$DCQL_{f1}$ scheint im Vergleich zu *DCQL* mit dem lernenden Gegner nicht so gut klarzukommen, wie mit der Greedy-Strategie. $DCQL_{f1}$ spielt am Anfang schlechter, als *DCQL* und spielt auch langfristig nicht besser. Nach 10 000 000 Spielen erreicht der Agent eine Gewinnrate von 21% und benötigt dafür fast sechs mal so viele Zustände wie *DCQL*.
Die erhöhte Anzahl an Zuständen hat dazu geführt, dass $DCQL_{f1}$ sehr langsam ist (bezogen auf die Simulationsgeschwindigkeit). Bei *DCQL* wirkt sich eine große Zustandsliste deutlich stärker auf die Geschwindigkeit aus, als bei *PQL*. *PQL* hat in diesem Beispiel zwar mehr Zustände gelernt und damit eine größere Zustansliste, als jeder der beiden *DCQL*-Agenten und das Matching ist ähnlich aufwendig, wie die Unifikation, jedoch kann *PQL* seine Zustandsliste umsortieren (Abschnitt 2.6.2). *PQL* sortiert als Optimierungsschritt seine Zustandsliste regelmäßig nach den am häufigsten besuchten Zuständen. Da die Zustandsliste von *PQL* keine Zustandsaggregation über die Reihenfolge der Zustände definiert, ist diese Umsortierung ohne weiteres möglich. Dank dieser Umsortierung erreicht die Simulation von *PQL* gegen *Q-Learning* eine Geschwindigkeit von etwa $892 \frac{Spiele}{s}$. Bei *DCQL* hingegen wird über die Reihenfolge der abstrakten Zustände, die Zustandsaggregation definiert. Würde beispielsweise der abstrakte Zustand $true \Longrightarrow \{?\}$ an die erste Stelle der Zustandsliste sortiert werden, dann würde dies die Zustandsaggregation verändern und dieser abstrakte Zustand würde alle Zustände des Zustandsraums aggregieren.

Im Spiel gegen *Q-Learning* erreichte die Simulation mit *DCQL* eine durchschnittliche Geschwindigkeit von $159{,}12 \frac{Spiele}{s}$, während die Simulation mit $DCQL_{f1}$ nur $39{,}1 \frac{Spiele}{s}$ erreicht hat. Dieser Faktor von ≈ 4 lässt sich durch einen Blick auf die Positionen innerhalb der Zustandsliste der am häufigsten besuchten Zustände für die beiden Agenten erklären. Die Daten sind in

Tabelle 1 und 2 im Anhang abgebildet. Anhand dieser Werte kann man die durchschnittliche Position (gewichteter Mittelwert) der am häufigsten besuchten Zustände abschätzen.

$$\varnothing pos_{f1} = \frac{\sum p \cdot pos}{\sum p} = \frac{802{,}047}{51\%} = 1572{,}64 \tag{3.1}$$

$$\varnothing pos_{f10} = \frac{\sum p \cdot pos}{\sum p} = \frac{155{,}223}{41{,}6\%} = 373{,}13 \tag{3.2}$$

$$\frac{\varnothing pos_{f1}}{\varnothing pos_{f10}} = \frac{1572{,}64}{373{,}13} = 4{,}21 \tag{3.3}$$

Bei der Entscheidungsliste von *DCQL* entspricht die Position eines Zustands innerhalb der Zustandsliste gleichzeitig der Anzahl der Unifikationen, die durchgeführt werden, ehe der Zustand erkannt wird. Dies resultiert daraus, dass der Agent die Zustände der Zustandsliste der Reihe nach durchgeht und versucht, sie mit dem aktuellen Zustand zu unifizieren. Aus Gleichung 3.3 geht somit direkt hervor, dass $DCQL_{f1}$ im Durchschnitt in jedem Zustand etwa viermal so viele Unifikationen durchführt, wie $DCQL_{f10}$. Daraus resultiert auch die viermal langsamere Geschwindigkeit des Agenten.

3.2.1 DynamicTreeQLearningAI (DTQL)

Aus den bisherigen Ergebnissen geht hervor, dass der Agent vergleichsweise langsam spielt und potenziell verbessert werden kann, wenn der Agent weniger Schritte benötigt, um die häufig besuchten Zustände zu erkennen. Dieses Problem lässt sich durch eine Baumdarstellung der Zustandsliste reduzieren. Um das Konzept der Zustandsliste auf eine Baumstruktur zu übertragen, würde jeder Zustand einem Knoten im Baum entsprechen und jeder abgeleitete Zustand würde als Kindknoten eingetragen werden. Tabelle 3.1 zeigt die sechs Zustände, von denen im Spiel gegen *Q-Learning* die meisten neuen Zustände abgeleitet wurden.

Tabelle 3.1: Eine Auflistung der Zustände, aus denen die meisten neuen Zustände abgeleitet wurden bei einem Spiel gegen *Q-Learning* mit 1 000 000 Spielen. Insgesamt hat der Agent zu dem Zeitpunkt 727 Zustände gelernt.

Zustand	*davon abgeleitet*	*Anteil*	$\sum$
$true$	149	20%	20%
$4b(X)$	124	17%	38%
$2k(X) \wedge 2b(X)$	119	16%	54%
$3k(X) \wedge 2k(X) \wedge 2b(X)$	59	8%	62%
$2k(X) \wedge 2b(X) \wedge 2k(A) \wedge 2b(A) \vert A = X - 1$	50	7%	69%
$4b(X) \wedge 3k(A) \wedge 3k(B) \vert A = X - 3, B = X - 2$	25	3%	72%

Abbildung 3.3 zeigt, wie eine Baumstruktur für diese Zustände und die davon abgeleiteten Zustände aussehen würde. Um festzustellen, in welchem Zustand sich der Agent momentan befindet, würde er am Wurzelknoten anfangen und zuerst beim aktuellen Knoten selbst prüfen, ob sich dieser mit dem konkreten Zustand unifizieren lässt. Der Agent würde anschließend den tiefsten Knoten im Baum suchen, der sich mit dem konkreten Zustand unifizieren lässt. Dabei werden alle Kindknoten eines nicht unifizierbaren Knotens nicht weiter beachtet. Auf diese Weise kann die Zustandsliste durch eine äquivalente Baumstruktur ersetzt werden, welche die gleichen Zustandsaggregation definiert, dafür jedoch eine unterschiedliche Anzahl an Unifikationen benötigt.

Am Beispiel des Baums aus Abbildung 3.3 bedeutet dies, dass der Agent 150 Unifikationen durchführen muss, um festzustellen, dass er sich in dem Zustand $true \Longrightarrow ?$ befindet. In einer äquivalenten Zustandsliste mit 527 Zuständen wären 527 Unifikationen notwendig, da dieser Zustand immer am Ende der Zustandsliste liegt. An dem Baum lässt sich ablesen, dass 150 Unifikationen das *Worst-Case*-Verhalten darstellen, was deutlich weniger ist, als das *Average-Case*-Verhalten der Zustandsliste mit $\frac{n}{2} = \frac{527}{2} = 263{,}5$ Unifikationen.

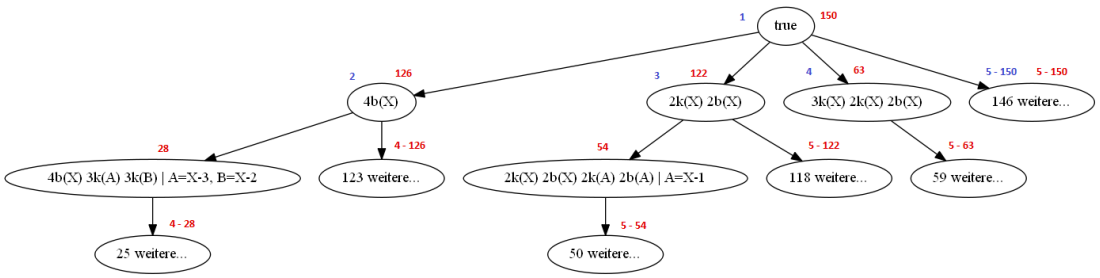

Abbildung 3.3: Darstellung der sechs Zustände aus Tabelle 3.1 als Ableitungsbaum. Die blauen Zahlen verdeutlichen die Reihenfolge in welcher der Agent versuchen würde die Zustände zu unifizieren, für den Fall, dass sich nur $true$ unifizieren lässt. Die roten Zahlen geben für jeden Zustand an, wie viele Unifikationen der Agent jeweils durchführen muss, um zu erkennen, dass er sich in dem Zustand befindet. Dieser Baum stellt 527 Zustände dar.

DCQL auf einer solchen Baumstruktur lernen zu lassen, sollte einen deutlichen Performancegewinn erzielen. Tatsächlich führt dies allein zu einer Steigerung der Simulationsgeschwindigkeit auf $455 \frac{Spiele}{s}$ ohne dabei das Lernverhalten zu beeinflussen. Dies entspricht einer Steigerung der Simulationsgeschwindigkeit um 185%.

Beweis für Äquivalenz von Zustandsliste und Zustandsbaum.
Seien S_1, S_2, S_3 von S_0 abgeleitete Zustände.

1. Dann werden sie in der Zustandsliste in der Reihenfolge, in der sie abgeleitet wurden vor dem Zustand S_0 angeordnet.
2. Bevor der Agent prüft, ob sich der Zustand S_0 unifizieren lässt, muss er sicherstellen, dass sich keiner der Zustände S_1, S_2, S_3 unifizieren lässt. (In der Reihenfolge, in der sie abgeleitet wurden)

Auf den Baum übertragen: Alle abgeleiteten Zustände S_1, S_2, S_3 werden in der Ableitungsreihenfolge als Kindknoten von S_0 eingetragen (links nach rechts). Der Agent weiß, dass er sich in Zustand S_0 befindet, wenn sich S_0, aber kein davon abgeleiteter Zustand unifizieren lässt.

3. Da S_1 von S_0 abgeleitet ist, muss S_1 eine echte Teilmenge der Zustände unifizieren, die sich mit S_0 unifizieren lassen.
4. Somit gilt: $\forall S : unifizierbar(S, S_1) \implies unifizierbar(S, S_0)$.
5. Also auch die Kontraposition: $\forall S : \neg unifizierbar(S, S_0) \implies \neg unifizierbar(S, S_1)$.

6. Lässt sich ein Zustand S somit nicht mit S_0 unifizieren, dann kann er auch mit keinem davon abgeleiteten Zustand unifiziert werden.

Auf den Baum übertragen: Lässt sich ein Knoten im Baum nicht mit einem Zustand S unifizieren, dann lassen sich auch dessen Unterknoten nicht mit S unifizieren und müssen nicht weiter beachtet werden. □

An den Unifikationsschritten aus Abbildung 3.3 lässt sich erkennen, dass der Agent weniger Schritte benötigt, wenn die häufig besuchten Zustände möglichst weit links im Baum angeordnet sind. Daraus folgt die Überlegung die Kindknoten eines Zustands regelmäßig nach der Anzahl der erfolgreichen Unifikationen neu anzuordnen. Dazu merkt sich der Agent für jeden Knoten des Baums, wie oft versucht wurde, den Knoten mit einem Zustand zu unifizieren und wie oft sich der Knoten mit dem Zustand unifizieren ließ. Nach $limit = 50\,000$ versuchten Unifikationen werden alle direkten Kindknoten des betroffenen Knotens nach der Anzahl der erfolgreichen Unifikationen absteigend sortiert. Anschließend werden die Zähler zurückgesetzt. Der Grenzwert von $limit = 50\,000$ brachte von mehreren getesteten Werten die höchste Verbesserung der Simulationsgeschwindigkeit. Setzt man $limit = \infty$, dann entspricht das *DTQL* ohne Sortierung der Kindknoten. Mit $limit = 50\,000$ erreicht der Agent eine Simulationsgeschwindigkeit von $540 \frac{Spiele}{s}$, was gegenüber dem nicht sortierten Baum einem relativ geringen Performancegewinn von 18% entspricht.

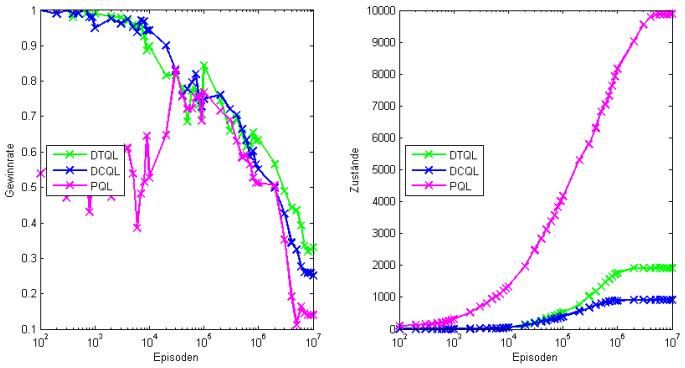

Abbildung 3.4: Vergleich der Agenten im Spiel gegen *Q-Learning*. Hier ist zusätzlich das Lernverhalten von *DTQL* abgebildet. Jeder Agent hat einmal 10 000 000 Spiele gegen *Q-Learning* gespielt.

In Abbildung 3.4 ist zusätzlich zu den bisherigen Agenten der in diesem Abschnitt beschriebene Agent *DTQL* abgebildet. Man erkennt, dass der Agent ein anderes Lernverhalten zeigt und deutlich mehr Zustände lernt, als *DCQL*. Die Ursache für diese Unterschiede ist die hinzugekommene Umsortierung der Knoten. Hierdurch werden einige Zustände mehrfach von unterschiedlichen Knoten abgeleitet, was ohne Sortierung nicht der Fall war. Dies wird in Abbildung 3.5 veranschaulicht. Hier wird der Zustand $3k(X) \wedge 3b(X)$ zunächst aus dem Zustand $3b(X)$ abgeleitet und nach dem Umsortieren nochmals aus dem Zustand $3k(X)$.

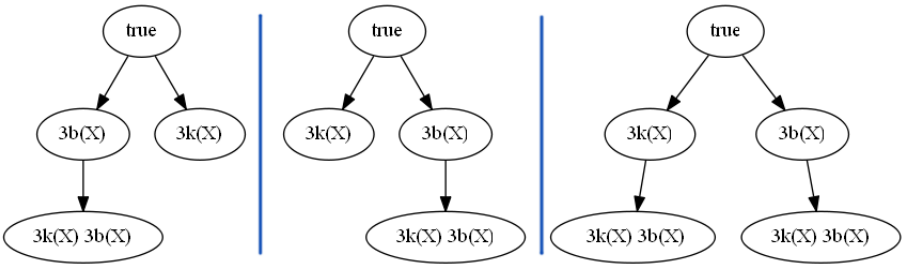

Abbildung 3.5: Durch das Umsortieren der Knoten, wurde der Zustand $3k(X) \wedge 3b(X)$ mehrfach abgeleitet.

Das Problem lässt sich wiederum dadurch lösen, dass vor dem Einfügen eines neuen Zustands der gesamte Baum danach durchsucht wird, ob dieser Zustand bereits vorhanden ist. Nur falls er nicht existiert, wird er eingefügt, anderenfalls wird der ursprüngliche Zustand referenziert. Welche Auswirkungen diese Änderung auf das Lernverhalten hat, wurde im Rahmen dieser Arbeit nicht mehr untersucht. Es ist zu erwarten, dass die Zustandsmenge im Vergleich zu dem unsortierten Baum nicht so schnell anwächst, da einige Zustände mehrfach abgeleitet werden, ohne effektiv die Zustandsmenge zu erhöhen. Welche Auswirkung dies auf das Lernverhalten hat, ist hingegen schwer abzuschätzen.

Vergleichbare Ansätze

Vergleichbare Ansätze sind die Tree-Learner *UTree*[10] und *RRL-TG*[7]
UTree modelliert die Umgebung zwar durch Features anstatt durch Prädikate, lernt aber genau wie der hier vorgestellte Agent, die Zustände als Konjunktion von Zustandseigenschaften. Im Gegensatz zu *DTQL* wird in *UTree* die Konjunktion über die Baumstruktur abgebildet und nicht innerhalb der Zustände. Jeder Knoten des Baums stellt eine Zustandseigenschaft des Zustands dar und auf dem Weg zum Blattknoten muss der Zustand jede Zustandseigenschaft der darüber liegenden Knoten erfüllen. Das besondere an *UTree* ist die Art, wie neue Knoten

in den Zustandsbaum aufgenommen werden. Der Agent fügt zunächst eine sogenannte *Fringe* ein. Dieser Knoten stellt ein Hypothese dar und wird erst zu einem richtigen Knoten, wenn der Agent anhand von Statistiken entscheidet, dass dieser Knoten eine sinnvolle Partitionierung des Zustandsraums darstellt. Anderenfalls wird sie wieder verworfen.

RRL-TG is hingegen auf das Lernen mit prädikatenlogischer Zustandsmodellierung ausgelegt. Auch *RRL-TG* bildet die Konjunktion der Prädikate über die Baumstruktur ab, wobei sich die Knoten entlang eines Pfades gleiche Variablen teilen können. *RRL-TG* erwartet von dem Entwickler eine Vorgabe von sogenannten *rmode*-Deklarationen. Diese geben vor, wie oft innerhalb eines Pfades ein bestimmtes Prädikat vorkommen darf und ob sich das Prädikat die Variablen mit einem Eltern- oder Kindknoten teilen kann. Diese Deklarationen sollen beim Ableiten neuer Kindknoten die möglichen Prädikate einschränken, damit der Agent nicht zu viele Möglichkeiten testen muss.
Der Agent sammelt statistische Werte über jeden einzelnen Knoten und fügt nur dann neue Kindknoten ein, wenn dadurch die Varianz der Q-Werte deutlich reduziert wird.

3.3 Direktvergleich mit PQLearningAI

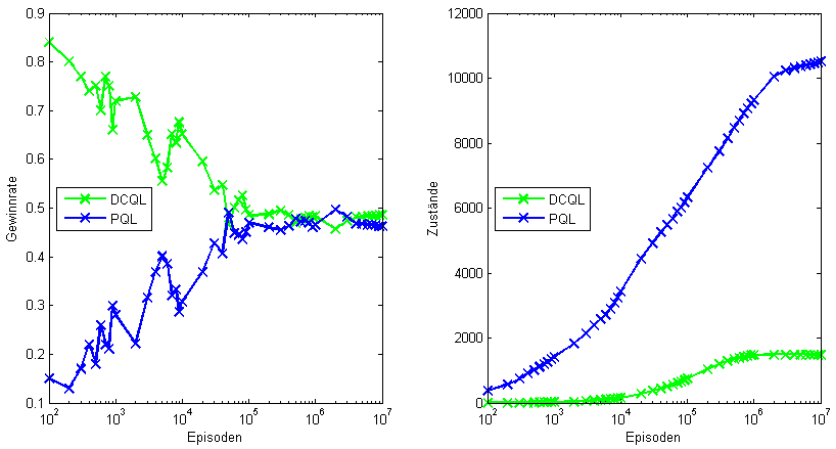

Abbildung 3.6: Direkter Vergleich von *DCQL* und *PQL* über 10 000 000 Spiele gegeneinander.

Abbildung 3.6 zeigt das Lernverhalten von *DCQL* und *PQL* über 10 000 000 Spiele im Spiel gegeneinander. Wie bereits im Spiel gegen *Q-Learning*, spielt auch hier *DCQL* in den ersten 100 000 Spielen besser als *PQL*, was an dem anfangs deutlich kleinerem Zustandsraum liegt. *DCQL* kann dadurch für wenige Zustände Q-Werte immer weiter optimieren, während *PQL*

viele neue Zustände lernt, in denen der Agent noch keine Erfahrung gesammelt hat. Nach etwa 100 000 Spielen, sind beide Agenten ähnlich gut und gewinnen jeweils in etwa 50% der Fälle.

Dieses Ergebnis ist etwas überraschend, da *DCQL* mithilfe der *Constraints* für die Variablen, die abstrakten Zustände genauer beschreiben kann. So kann *DCQL* prinzipiell auch zwischen zwei Zuständen unterscheiden, die sich ausschließlich in den *Constraints* unterscheiden, was für *PQL* nicht möglich ist. Das Beispiel aus Gleichung 3.4 legt nahe, dass der Zustand S_1 besser ist, als der Zustand S_2, da im ersten Zustand die Prädikate näher beieinander sind, was dem Agenten in diesem Spielfeldbereich bessere Gewinnchancen bietet. Beim Zustand S_2 sind die Prädikate hingegen auf die beiden Spielfeldränder verteilt.

$$\begin{aligned} S_1 &= 3k(X) \wedge 2k(A)|A = X + 1 \\ S_2 &= 3k(X) \wedge 2k(A)|A = X + 6 \end{aligned} \tag{3.4}$$

Daraus, dass beide Agenten nach einer bestimmten Lernphase gleich gut gegeneinander spielen, kann man schließen, dass die Unterscheidung von Zuständen anhand ihrer *Constraints* hier keinen strategisch relevanten Vorteil bringt. Wie jedoch in Abschnitt 3.2 gezeigt wurde, spielt *DCQL* gegen *Q-Learning* langfristig etwas besser als *PQL*. Die Ursache für die etwas bessere Spielweise gegen *Q-Learning* sind wahrscheinlich die *Constraints*, die jedoch im direkten Vergleich gegen *PQL* keinen deutlichen Vorteil bringen.
Auch hier benötigt *DCQL* deutlich weniger Zustände als *PQL*, um gleich gut zu spielen. *PQL* benötigt hierfür 10 519 Zustände, während *DCQL* nur 1 481 Zustände lernt.

3.3.1 Vergleich mit erweitertem Zustandsraum

Aus Rücksicht auf die starke Zustandszunahme von *PQL* und die damit verbundene erhöhte Simulationsdauer, wurde die Zustandsbeschreibung so weit wie möglich eingeschränkt, indem unwichtige Prädikate nicht in die Zustandsbeschreibung aufgenommen wurden. Dies ist in Abschnitt 2.6.1 beschrieben. Um zu prüfen, ob diese ausgelassenen Prädikate vielleicht doch eine strategische Relevanz haben, wurde ein *DCQL*-Agent mit vollständigem Zustandsraum ($DCQL_{full}$) gegen *PQL* getestet.

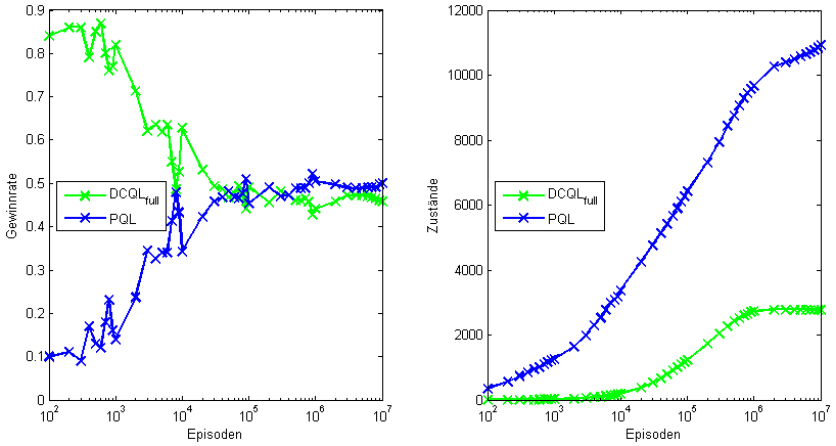

Abbildung 3.7: Direkter Vergleich von $DCQL_{full}$ und *PQL* über 10 000 000 Spiele gegeneinander.

In Abbildung 3.7 sieht man, dass er erweiterte Zustandsraum für den Agenten jedoch keinen Vorteil darstellt. Auch hier spielen die Agenten auf lange Sicht in etwa gleich gut.
Der erweiterte Zustandsraum wirkt sich hauptsächlich auf die Menge der gelernten Zustände aus. *PQL* benötigt mit 10 915 nur 400 Zustände mehr, während $DCQL_{full}$ mit 2 773 Zuständen 87% mehr Zustände benötigt, als ohne die zusätzlichen Prädikate. Man kann hieraus schlussfolgern, dass die zusätzlichen Prädikate wirklich keine strategische Relevanz für das Spiel haben und somit nicht beachtet werden müssen.

4 Fazit

4.1 Zusammenfassung der Ergebnisse

Die Auswertungen zeigen, dass stärker abstrahierende Agenten weniger Zustände benötigen, um eine gute Strategie zu repräsentieren, und diese nach weniger Episoden gelernt haben. Die Lerndaten der stark abstrahierenden Agenten nehmen trotz der komplexeren Datenstruktur weniger Speicherplatz ein, als die Lerndaten von *Q-Learning*. Werden die *Traces*[1] der abstrahierenden Agenten entfernt (sie spielen für die Strategie selbst keine Rolle), dann nehmen die Lerndaten von *DCQL* nach 10 000 000 Episoden gegen *Q-Learning* nur etwa 100 Kilobyte ein, während die Lerndaten von *Q-Learning* bei etwa 100 Megabyte liegen. Die Lerndaten von *PQL* liegen mit etwa 1 Megabyte zwischen denen von *DCQL* und *Q-Learning*. Die Nachteile der abstrahierenden Agenten liegen in dem erhöhten Rechenaufwand, um eine Entscheidung zu treffen. Zudem können sie aufgrund ihrer abstrakten Modellierung des Zustandsraums nicht den gesamten Zustandsraum abbilden und keine optimale Strategie lernen.

Die hier vorgestellten Agenten sind somit vor allem dann sinnvoll, wenn Episoden sehr teuer sind, weil beispielsweise keine Simulation möglich ist und der Agent in der realen Welt Erfahrung sammeln muss. In solchen Szenarien muss der Agent bereits nach sehr wenigen Episoden eine gute Strategie lernen.

Tabelle 4.1: Kurze Zusammenfassung der Daten aus Abschnitt 3.2. Die benötigten Zustände und die Datenmenge sind dabei auf die nächste Zehnerpotenz gerundet.

Agent	*Benötigte Zustände*	*Datenmenge (KB)*	$\frac{Episoden}{sec}$
QL	10^6	10^5	3500
PQL	10^4	10^3	900
DTQL	$2 \cdot 10^3$	$2 \cdot 10^2$	500
DCQL	10^3	10^2	160

[1] Hiermit sind die Daten gemeint, die DCQL sammelt, um neue Zustände daraus abzuleiten. (Abschnitt 2.7)

4.2 Übertragbarkeit auf andere Anwendungen

Da der hier entwickelte Abstraktionsalgorithmus genau auf die Aufgabenstellung von 4-Gewinnt und die vorgestellte Zustandsmodellierung zugeschnitten ist, stellt sich die Frage, ob sich dieser Algorithmus auf andere Problemstellungen übertragen lässt und welche Voraussetzungen dafür erfüllt sein müssen.

Damit dieser Algorithmus funktionieren kann, müssen die Zustände durch Prädikate ausgedrückt werden, deren Argumente den möglichen Aktionen dieses Zustands entsprechen. In dieser Arbeit waren die Aktionen natürliche Zahlen, wodurch sie über die Differenz zueinander in Beziehung gesetzt werden konnten. Diese Eigenschaft wurde für die Herleitung der *Constraints* genutzt. Dies ist keine notwendige Voraussetzung für die Aktionen. Lassen sich die Aktionen beispielsweise nur ordnen, dann können sich die *Contraints* auf die Ordnung der Aktionen beziehen. (Bsp: $S_1 = p(X) \wedge q(Y) | X > Y$)
Denkbar wäre auch, die Aktionen über eigene Prädikate in Beziehung zueinander zu setzen. (Bsp: $S_2 = p(X) \wedge q(Y) | ist\_benachbart(X, Y)$)
Lässt sich gar keine Beziehung zwischen den Aktionen eines Zustands herstellen, dann können die Contraints auch weggelassen werden. Aus der Auswertung von *DCQL* gegen *PQL* (Abschnitt 3.3) sieht man, dass ein Agent ohne *Contraints* ähnlich gut spielen kann.

Lässt sich für eine Problemstellung eine prädikatenlogische Zustandsmodellierung angeben, sodass die Argumente der Prädikate den möglichen Aktionen entsprechen, dann lässt sich der in dieser Arbeit vorgestellte Algorithmus darauf anwenden.

4.3 Ausblick

Im Rahmen dieser Arbeit wurden die Agenten *DCQL* und dessen Erweiterung *DTQL* entwickelt und darauf hin optimiert, nach möglichst wenigen Episoden mit möglichst wenigen Zuständen eine gute Strategie zu lernen und trotzdem vom rechnerischen Aufwand in einem akzeptablen Rahmen zu bleiben.

Es wurden noch nicht alle möglichen Verbesserungen umgesetzt und ausgewertet. Denkbar wäre, regelmäßig die selten besuchten Zustände aus der Zustandsliste (bzw. Zustandsbaum) zu entfernen und so nur Q-Werte für die Zustände zu führen, die der Agent auch tatsächlich benö-

tigt. Das würde die Zahl der Zustände weiter reduzieren und damit die benötigte Datenmenge zur Darstellung einer Strategie reduzieren und die Simulationsgeschwindigkeit erhöhen.

Darüber hinaus könnte man die am häufigsten besuchten Zustände mit Q-Werten eines trainierten Agenten als Expertenwissen für einen neuen Agenten nutzen und prüfen, ob dieser neue Agent mit diesem Expertenwissen eine bessere Strategie lernen kann oder die gleiche Strategie mit weniger Zuständen ausdrücken kann.
Eine weitere Möglichkeit die Abstraktion des Agenten flexibler zu machen, wäre die Betrachtung von Teilmengen von Zustandsbeschreibungen. Im hier vorgestellten Abstraktionsalgorithmus werden im Ableitungsschritt die Zustände aus den *Traces* nach dem aufsummierten $|\delta|$ sortiert (Abschnitt 2.7). Der Abstraktionsalgorithmus wäre flexibler, wenn er auch Teilmenengen der Zustandsbeschreibung berücksichtigen würde. Seien beispielsweise die folgenden *Traces* gegeben:

Tabelle 4.2: Beispiel-*Traces*, für die der Abstraktionsalgorithmus den Zustand $S_1 = 3b(X) \wedge 2k(A) \wedge 2b(A)|A = X + 3$ ableiten würde.

Zustand	*Aktion*	δ	*prim. Prädikate*	*sek. Prädikate*
$3b(1) \wedge 2k(4) \wedge 2k(5)$	1	10	$3b(X)$	$2k(X+3) \wedge 2k(X+4)$
$3b(2) \wedge 2k(5) \wedge 2b(5)$	2	20	$3b(X)$	$2k(X+3) \wedge 2b(X+3)$
$3b(3) \wedge 2k(6) \wedge 2k(7)$	7	10	$3b(X)$	$2k(X+3) \wedge 2k(X+4)$
$3b(3) \wedge 2k(6) \wedge 2b(6)$	3	10	$3b(X)$	$2k(X+3) \wedge 2b(X+3)$

Für die *Traces* aus Tabelle 4.2 würde der Agent den Zustand
$S_1 = 3b(X) \wedge 2k(A) \wedge 2b(A)|A = X + 3$ ableiten, der zwei *Traces* abdeckt mit einem $\sum |\delta| = 30$. Mit einem Blick auf die Tabelle erkennt man jedoch, dass alle *Traces* durch den Zustand $S_2 = 3b(X) \wedge 2k(A)|A = X + 3$ abgedeckt wären mit einem $\sum |\delta| = 50$. Um den Zustand S_2 abzuleiten, müsste der Algorithmus zusätzlich zu den primären und sekundären Prädikaten auch alle ihre Teilmengen nach $\sum |\delta|$ sortieren. Der Abstraktionsalgorithmus könnte den Zustandsraum dann in weniger Ableitungsschritten stärker partitionieren und das könnte dazu führen, dass der Agent nach weniger Ableitungsschritten eine bessere Strategie lernt.

Als Ergebnis dieser Arbeit lässt sich festhalten, dass sich ein Abstraktionsalgorithmus entwickeln lässt, der zusammen mit dem *CARCASS*-Konzept eine sinnvolle Alternative zu regulärem *Q-Learning* darstellt, falls eine Strategie schnell gelernt werden muss. Zudem wurden mehrere denkbare Verbesserungen aufgezeigt, um den Algorithmus weiter zu verbessern.

Anhang

Tabelle 1: Die 15 am häufigsten besuchten Zustände des Agenten $DCQL_{f1}$ bei einem Spiel gegen *Q-Learning* über 100 000 Spiele. *p* steht für prozentuale Häufigkeit, mit welcher der Zustand besucht wurde und *pos* für die Position dieses Zustands in der Zustandsliste. Die letzte Zeile enthält die aufsummierten Werte.

p	*pos*	$p \cdot pos$	*Zustand*
4,8%	212	10,176	$3k(X) \wedge 3b(X) \wedge 4b(A) \wedge 3k(A) \wedge 3k(B) \wedge 3b(B) \vert A = X + 1, ...$
4,0%	224	8,96	$4b(X) \wedge 3k(X) \wedge 3k(A) \wedge 3b(A) \wedge 3b(B) \vert A = X - 1, B = X + 1$
3,4%	895	30,43	$4b(X)$
3,3%	900	29,7	$1k(X) \wedge 1k(C) \wedge 1k(A) \wedge 1k(F) \wedge 1k(E) \wedge 1k(B) \wedge 1k(D) \vert ...$
3,3%	920	30,36	$1k(X) \wedge 2b(C) \wedge 2b(D) \wedge 1k(A) \wedge 1k(F) \wedge 1k(E) \wedge 1k(B) \vert ...$
3,3%	919	30,327	$2k(X) \wedge 2b(X) \wedge 2b(B) \wedge 1k(D) \wedge 1k(C) \wedge 1k(A) \wedge 1k(F) \wedge ...$
3,3%	3406	112,398	$1k(X) \wedge 2k(C) \wedge 2k(A) \wedge 2b(B) \wedge 2b(D) \wedge 1k(F) \wedge 1k(E) \vert ...$
3,2%	915	29,28	$3b(X) \wedge 2k(E) \wedge 2k(C) \wedge 2k(B) \wedge 2k(D) \wedge 2b(D) \wedge 1k(A) \wedge ...$
3,2%	2964	94,848	$2k(X) \wedge 2k(E) \wedge 2k(D) \wedge 2k(A) \wedge 2k(B) \wedge 2k(C) \wedge 2b(B) \wedge ...$
3,2%	2541	81,312	$3k(X) \wedge 3k(C) \wedge 2k(F) \wedge 2k(D) \wedge 2k(A) \wedge 2k(B) \wedge 2b(B) \wedge ...$
3,2%	2531	80,992	$2k(X) \wedge 3k(C) \wedge 2k(A) \wedge 2k(E) \wedge 2k(D) \wedge 2k(B) \wedge 2b(A) \wedge ...$
3,2%	2532	81,024	$2k(X) \wedge 3k(C) \wedge 3b(D) \wedge 2k(A) \wedge 2k(F) \wedge 2k(E) \wedge 2k(B) \wedge ...$
3,2%	2778	88,896	$3k(X) \wedge 3k(A) \wedge 3k(B) \vert A = X + 1, B = X + 4$
3,2%	2777	88,864	$3k(X) \wedge 3k(A) \wedge 3k(B) \wedge 3k(C) \vert A = X - 2, B = X - 1, ...$
3,2%	140	4,48	$3k(X) \wedge 4b(A) \wedge 3k(B) \wedge 3k(C) \wedge 3b(D) \vert A = X + 4, ...$
51%		802,047	

Tabelle 2: Die 15 am häufigsten besuchten Zustände des Agenten *DCQL* ($f = 10$) bei einem Spiel gegen *Q-Learning* über 100 000 Spiele. *p* steht für prozentuale Häufigkeit, mit welcher der Zustand besucht wurde und *pos* für die Position dieses Zustands in der Zustandsliste. Die letzte Zeile enthält die aufsummierten Werte.

p	*pos*	$p \cdot pos$	*Zustand*
6,4%	260	16,64	$1k(X) \wedge 2k(C) \wedge 2b(C) \wedge 2b(D) \wedge 1k(A) \wedge 1k(F) \wedge 1k(E) \wedge ...$
3,2%	301	9,632	$1k(X) \wedge 1k(A) \wedge 1k(F) \wedge 1k(E) \wedge 1k(B) \wedge 1k(C) \wedge 1k(D) \vert ...$
3,2%	261	8,352	$1k(X) \wedge 2b(C) \wedge 2b(D) \wedge 1k(A) \wedge 1k(F) \wedge 1k(E) \wedge 1k(B) \vert ...$
3,2%	686	21,952	$2k(X) \wedge 3b(E) \wedge 2k(A) \wedge 2k(B) \wedge 2b(C) \wedge 1k(F) \wedge 1k(D) \vert ...$
3,2%	11	0,352	$4b(X) \wedge 3k(B) \wedge 2k(E) \wedge 2k(C) \wedge 2k(A) \wedge 2k(F) \wedge 2b(D) \vert ...$
3,2%	537	17,184	$1k(X) \wedge 2k(B) \wedge 2b(B) \wedge 1k(C) \wedge 1k(A) \wedge 1k(E) \wedge 1k(D) \vert ...$
3,2%	527	16,864	$2k(X) \wedge 3k(B) \wedge 3b(F) \wedge 2k(E) \wedge 2k(C) \wedge 2k(D) \wedge 2k(A) \wedge ...$
3,2%	315	10,08	$1k(X) \wedge 2k(A) \wedge 2k(B) \wedge 2k(C) \wedge 2b(B) \wedge 2b(C) \wedge 1k(E) \wedge ...$
2,8%	307	8,596	$2k(X) \wedge 2b(X) \wedge 3b(B) \wedge 2k(C) \wedge 2k(A) \wedge 2k(D) \wedge 2k(B) \wedge ...$
2,3%	309	7,107	$2k(X) \wedge 2b(X) \wedge 3b(F) \wedge 2k(E) \wedge 2k(B) \wedge 2k(C) \wedge 2k(D) \wedge ...$
1,8%	617	11,106	$3b(X) \wedge 2k(X) \wedge 2b(X)$
1,6%	627	10,032	$2k(X) \wedge 2b(X) \wedge 2k(A) \wedge 2b(A) \vert A = X - 1$
1,5%	638	9,57	$3k(X) \wedge 3k(A) \wedge 3b(B) \vert A = X + 1, B = X + 4$
1,4%	172	2,408	$3k(X) \wedge 3k(A) \wedge 3k(B) \wedge 3b(C) \vert A = X - 3, B = X - 2, C = X + 2$
1,4%	382	5,348	$2k(X) \wedge 3b(C) \wedge 2k(F) \wedge 2k(D) \wedge 2k(A) \wedge 2k(B) \wedge 2k(C) \wedge ...$
41,6%		155,223	

Abbildungsverzeichnis

Tabellenverzeichnis

Literaturverzeichnis

[1] Allis, Louis V.: *A knowledge-based approach of connect-four.* Vrije Universiteit, Subfaculteit Wiskunde en Informatica, 1988

[2] Ash, Timur: Dynamic node creation in backpropagation networks. In: *Connection Science* 1 (1989), Nr. 4, S. 365–375

[3] Bellman, Richard: Dynamic Programming. In: *Princeton University Press.* (1957), S. 151

[4] Bertsekas, Dimitri P. ; Tsitsiklis, John N.: *Neuro-Dynamic Programming.* 1st. Athena Scientific, 1996. – ISBN 1886529108

[5] Boutilier, Craig: *Knowledge Representation for Stochastic Decision Processes.* S. 111–152. In: Wooldridge, Michael J. (Hrsg.) ; Veloso, Manuela (Hrsg.): *Artificial Intelligence Today: Recent Trends and Developments.* Berlin, Heidelberg : Springer Berlin Heidelberg, 1999. – URL http://dx.doi.org/10.1007/3-540-48317-9_5. – ISBN 978-3-540-48317-5

[6] Cobo, Luis C. ; Isbell, Charles L. ; Thomaz, Andrea L.: Object Focused Q-learning for Autonomous Agents. In: *Proceedings of the 2013 International Conference on Autonomous Agents and Multi-agent Systems.* Richland, SC : International Foundation for Autonomous Agents and Multiagent Systems, 2013 (AAMAS '13), S. 1061–1068. – URL http://dl.acm.org/citation.cfm?id=2484920.2485087. – ISBN 978-1-4503-1993-5

[7] Driessens, Kurt: *Relational reinforcement learning*, Katholieke Univeriteit Leuven, Dissertation, 2004

[8] Hanson, Stephen J.: Meiosis networks. In: *Advances in neural information processing systems*, 1990, S. 533–541

[9] Kaelbling, Leslie P. ; Littman, Michael L. ; Moore, Andrew W.: Reinforcement learning: A survey. In: *Journal of artificial intelligence research* (1996), S. 237–285

[10] McCallum, Andrew K.: *Reinforcement learning with selective perception and hidden state*, University of Rochester, Dissertation, 1995

[11] Otterlo, Martijn van: *The Logic of Adaptive Behavior: Knowledge Representation and Algorithms for Adaptive Sequential Decision Making under Uncertainty in First-Order and Relational Domains*. Bd. 192. IOS Press, 2009

[12] Precup, Doina: Eligibility traces for off-policy policy evaluation. In: *Computer Science Department Faculty Publication Series* (2000), S. 80

[13] Puterman, Martin L.: *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. 1st. New York, NY, USA : John Wiley & Sons, Inc., 1994. – ISBN 0471619779

[14] Rummery, Gavin A. ; Niranjan, M.: On-Line Q-Learning Using Connectionist Systems / Cambridge University, Engineering Department. 1994. – Forschungsbericht. CUED/F-INFENG/TR 166

[15] Sutton, Richard S.: Generalization in Reinforcement Learning: Successful Examples Using Sparse Coarse Coding. In: Touretzky, D. S. (Hrsg.) ; Hasselmo, M. E. (Hrsg.): *Advances in Neural Information Processing Systems 8*, MIT Press, 1996, S. 1038–1044. – URL http://papers.nips.cc/paper/1109-generalization-in-reinforcement-learning-successful-examples-using-sparse-coarse-coding.pdf

[16] Sutton, Richard S. ; Barto, Andrew G.: *Introduction to Reinforcement Learning*. 1st. Cambridge, MA, USA : MIT Press, 1998. – ISBN 0262193981

[17] Watkins, Christopher J. C. H.: *Learning from delayed rewards*. Cambridge, England, University of Cambridge, Dissertation, 1989

[18] Watkins, Christopher J. ; Dayan, Peter: Q-learning. In: *Machine learning* 8 (1992), Nr. 3-4, S. 279–292

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

Hamburg, 4. August 2016 David Olszowka