Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Fakultät Technik und Informatik
Department Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Patrick Boekhoven

Entwicklung eines Reinforcement Learning Frameworks auf Basis eines Agentensystems

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung
im Studiengang Angewandte Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer : Prof. Dr. rer. nat. Michael Neitzke
Zweitgutachter : Prof. Dr. rer. nat. Bernd Kahlbrandt

Abgegeben am 27. März 2011

Patrick Boekhoven

Thema der Bachelorarbeit
Entwicklung eines Reinforcement Learning Frameworks auf Basis eines Agentensystems.

Stichworte
Jade, Reinforcement learning, Agenten

Kurzzusammenfassung
Gegenstand dieser Bachelorarbeit ist es eine einheitliche Architektur für die Methoden im Bereich Reinforcement Learning zu finden und aus den gewonnenen Erkenntnissen ein generisches Framework auf Basis eines Agentensystems zu konstruieren. Die verschiedenen Methoden im Bereich Reinforcement Learning werden zu Anfang beschrieben. Dann wird die Architektur des Lernenden Agenten aufgegriffen um auf dessen Basis dieses Framework zu konstruieren. Darauf folgend werden die generischen Aspekte der einzelnen Methoden im Bereich des Reinforcment Learnings analysiert und es wird ein Bezug zu der Architektur des lernenden Agenten hergestellt. Es wird eine Abgrenzung vorgenommen welche Aspekte der Methoden im Bereich Reinforcement Learning umgesetzt werden sollen. Diese Abgrenzung geschieht ebenfalls für die Agenten und deren Umwelt in der diese agieren. Abschließend wird die Implementierung mit Hilfe der voran gegangenen Erkenntnisse vorgestellt.

Patrick Boekhoven

Title of the paper
Development of a reinforcement learning framework on basis of the agent system.

Keywords
Jade, Reinforcement learning, Multi-Agent System

Abstract
This Bachelor Thesis aims at finding a standartized architecture for methods in the field of reinforcement learning and to create a generic framework on basis of the agent system out of these findings. At first the different methods in the field of reinforcement learning will be described. Then the basic archiceture of the learning agent will be used to construct this framework. After that the generic aspects of the individual methods of reinforcement learning will be analyzed and put into context of the architecture of the learning agent. Subsequently it will be distinguished between the different aspects of reinforcement learning. This border will also be established between the agents and their enviroment. Finally the implementation will be presented on basis of the gained knowledge.

Inhaltsverzeichnis

1 Einleitung **8**
1.1 Motivation . . . 8
1.2 Ziel . . . 8
1.3 Aufbau der Arbeit . . . 8

2 Reinforcement Learning **10**
2.1 Agenten und Umwelt Interaktion . . . 10
2.2 Das Beispiel der Grid World . . . 11
2.3 Ziele und rewards . . . 11
2.4 Returns . . . 12
2.5 Markov Entscheidungsprozesse . . . 13
2.6 Value-Funktionen . . . 13
2.7 Explorationsverfahren . . . 14
2.8 ϵ greedy . . . 14
2.9 Boltzmann-Exploration . . . 14
2.10 Dynamische Programmierung . . . 15
2.11 Policy Evaluation . . . 15
2.12 Policy Improvement . . . 16
2.13 Policy Iteration . . . 17
2.14 On- und Off-Policy . . . 18
2.15 Monte Carlo Methoden . . . 18
2.16 First-visit Methode . . . 18
2.17 Every-visit-Methode . . . 18
2.18 On-Policy Monte Carlo Methode . . . 19
2.19 Off-Policy Monte Carlo Methode . . . 19
2.20 Temporal Difference Learning . . . 19
2.21 TD-Prediction . . . 20
2.22 Sarsa . . . 20
2.23 Sarsa-On-Policy . . . 20
2.24 Sarsa-Off-Policy . . . 22
2.25 Eligibility-Traces . . . 22
2.26 TD(λ) . . . 23

2.27 Sarsa(λ) 24
2.28 Replacing-Eligibility-Traces 25
2.29 Efficient-Eligibility-Traces 26

3 Agentensysteme **27**
3.1 Grundlagen 27
3.2 Die Autonomie der Agenten 27
3.3 3-Phasen-Zyklus 29
3.4 Umwelt des Agenten 29
3.5 Charakterisierung der Umwelt 29
3.6 Arten von Agentenprogrammen 31
3.7 Einfacher Reflex-Agent 31
3.8 Modellbasierter Reflex-Agent 31
3.9 Zielbasierter Agent 31
3.10 Nutzenbasierter Agent 32
3.11 Lernender Agent 32
3.12 Agentensystem 33

4 Generische Aspekte **34**
4.1 Generische Aspekte anhand des Umwelt Agenten Interaktion Interface 34
4.2 Das Leistungselement 35
4.3 Das Leistungselement anhand der Temporal Difference Learning und der Monte Carlo Methoden 35
4.4 Leistungselement anhand der dynamischen Programmierung 37
4.5 Das Lernelement anhand der verschiedenen Algorithmen 38
4.6 Lernelement anhand der dynamischen Programmierung 39
4.7 Lernelement anhand der TD Prediction 39
4.8 Lernelement anhand von Sarsa-On-Policy 39
4.9 Lernelement für den Sarsa-Off-Policy-Algorithmus 40
4.10 Der Problemgenerator 40
4.11 Ergebnis der Analyse 41

5 Abgrenzung des Frameworks **42**
5.1 Auswahl der Algorithmen 42
5.2 Dynamische Programmierung 42
5.3 Monte Carlo Methoden 43
5.4 Temproal Differnce Learning 44
5.5 V- beziehungsweise Q-Wert-Implementation 44
5.6 On- und Off-Policy 44
5.7 Eligibility-Traces 45
5.8 Explorationsverfahren 45

5.9 Ergebnis der Analyse 45
5.10 Beschaffenheit des Agenten 46
5.11 Ergebnis der Analyse 47
5.12 Beschaffenheit der Umwelt 47
5.13 Deterministische und stochastische Umwelt 48
5.14 Grad der Beobachtbarkeit 48
5.15 Statische und dynamische Umgebungen 48
5.16 Episodische und kontinuierliche Aufgaben 49
5.17 Diskrete und stetige Zustandsräume 49
5.18 Einzel- oder Multi-Agenten Umgebung 50
5.19 Ergebnisse der Analyse 50

6 Implementation **51**
6.1 Generelle Aspekte der Umwelt und des Agenten 51
6.2 Umwelt als Agent 51
6.3 Art des Agenten 52
6.4 Abbild der Umwelt 53
6.5 Sonstige Klasse 53
6.6 A_Situation 54
6.7 A_Aktion 54
6.8 A_Situation_Aktion 55
6.9 Aufbau der Umwelt 55
6.10 Elemente zur Kommunikation 56
6.11 Methode zur Bewertung aller Situationen 56
6.12 Element für verbotene Aktionen 57
6.13 Mittel zur Synchronisierung 57
6.14 Aufbau des Agenten 58
6.15 Das Kommunikationselement 58
6.16 Das Lernelement 58
6.17 Das Leistungselement 59
6.18 Der Problemgenerator 60
6.19 Übergabe 60
6.20 GUI 61

7 Beispielanwendungen **62**
7.1 Grid World 62
7.2 Tic Tac Toe 63
7.3 Die Siedler 63

8 Zusammenfassung **64**
8.1 Reinforcement Learning 64

8.2 Agentensysteme 64
8.3 Generische Aspekte 64
8.4 Abgrenzung des Frameworks 65
8.5 Bau der Beispielanwendungen 65
8.6 Fazit 65
8.7 Ausblick 66

Literaturverzeichnis **67**

Abbildungsverzeichnis **69**

Algorithmenverzeichnis **70**

Dies ist ein ganz kurzer Beispieltext [**?**]

Gegenstand dieser Bachelorarbeit ist es eine einheitliche Architektur für die Methoden im Bereich Reinforcement Learning zu finden und aus den gewonnenen Erkenntnissen ein generisches Framework auf Basis eines Agentensystems zu konstruieren. Die verschiedenen Methoden im Bereich Reinforcement Learning werden zu Anfang beschrieben. Dann wird die Architektur des Lernenden Agenten aufgegriffen um auf dessen Basis dieses Framework zu konstruieren. Darauf folgend werden die generischen Aspekte der einzelnen Methoden im Bereich des Reinforcment Learnings analysiert und es wird ein Bezug zu der Architektur des lernenden Agenten hergestellt. Es wird eine Abgrenzung vorgenommen welche Aspekte der Methoden im Bereich Reinforcement Learning umgesetzt werden sollen. Diese Abgrenzung geschieht ebenfalls für die Agenten und deren Umwelt in der diese agieren. Abschließend wird die Implementierung mit Hilfe der voran gegangenen Erkenntnisse vorgestellt.

This Bachelor Thesis aims at finding a standartized architecture for methods in the field of reinforcement learning and to create a generic framework on basis of the agent system out of these findings. At first the different methods in the field of reinforcement learning will be described. Then the basic archiceture of the learning agent will be used to construct this framework. After that the generic aspects of the individual methods of reinforcement learning will be analyzed and put into context of the architecture of the learning agent. sequently it will be distinguished between the different aspects of reinforcement learning. This border will also be established between the agents and their enviroment. Finally the implementation will be presented on basis of the gained knowledge. Key word Jade, Agentensystem, Reinforcment Learning

1 Einleitung

Die Arbeit befasst sich damit ein Framework für Reinforcement Learning Anwendungen zu schreiben. Dieses Framework soll auf einem Agentensystem basieren. Der einzelne Agent soll nach der Architektur eines lernenden Agenten von Stuart Russel und Peter Norvig aufgebaut werden.

1.1 Motivation

Zum Start dieser Arbeit ist kein ähnliches Projekt dieser Art bekannt. Zwar gibt es verschiedene Frameworks die für Reinforcement Learning Anwendungen ausgelegt sind, allerdings keine die auf einem eigenem Agentensystem beruht. Zudem ist keine einheitliche Architektur bekannt die es ermöglicht einzelne Algorithmen zu kapseln und direkt auszutauschen.

1.2 Ziel

Das Ziel am Ende dieser Arbeit ist es am Ende eine einheitliche Architektur vorzufinden die es einfach machen soll Anwendungen zu schreiben und dabei Algorithmen leicht auszutauschen. Der Aufwand neue Anwendungen zu schreiben soll minimiert werden. Dabei soll mit dieser Architektur gezeigt werden wie einfach es ist, mit diesem Framework einzelne Anwendungen einheitlich zu schreiben. Mehrere Anwendungen sollen mit diesem Framework am Ende realisiert werden. Die letzte implementierte Anwendung soll eine schon eher komplexere Anwendung sein, die auf einem Siedler Spiel beruht. Diese Anwendung wurde im Rahmen eines Projekts an der HAW geschrieben in welchem die Agenten ebenfalls mit Reinforcement Learning Algorithmen lernen.

1.3 Aufbau der Arbeit

Im zweiten Kapitel dieser Arbeit werden die Grundlagen im Bereich Reinforcement Learning vorgestellt. Im dritten Kapitel werden in Kurzform Software-Agenten vorgestellt und die

einzelnen Architekturen aufgegriffen. In den beiden folgenden Analyse Kapiteln werden erstens die generischen Aspekte der einzelnen Algorithmen aufgegriffen, und ein Bezug zur späteren Architektur hergestellt. Danach wird das Framework welches im Rahmen dieser Arbeit entwickelt wird eingegrenzt und klargestellt welche Aspekte aufgrund der bisherigen Erkenntnisse wie realisiert werden sollen und welche nicht. Das nächste Kapitel wird dann mit den Ergebnissen aus diesen beiden Analyse Kapiteln die Implementation aufzeigen, und wie dieses Framework aufgebaut ist. Im 7. Kapitel werden die 3 geschrieben Beispielanwendungen dann kurz erläutert. Kapitel 8 fasst die gewonnen Erkenntnisse alle noch einmal zusammen und gibt einen kleinen Ausblick.

2 Reinforcement Learning

Reinforcement Learning im allgemeinen gesprochen ist ein computergestützter Ansatz zum lernen. Hierbei lernt ein Agent durch die Wahl von Aktionen, welche Effekte in seiner Umwelt gegebenenfalls hervorzurufen. Für diese Effekte wird der Agent entweder belohnt oder bestraft. Diese Belohnung beziehungsweise Bestrafung funktioniert über *rewards* die der Agent zu jeder Situation erhält. Diese *rewards* sind direkt auf die Situation bezogen die der Agent durch seine Aktionen hervorgerufen hat. Das Ziel des Agenten ist es diesen erhaltenen *reward* zu maximieren, ohne das der Agent detaillierte Anweisungen von außen erhält. Der lernende Agent muss dabei selber herausfinden durch welche Aktionen er die Belohnungen maximieren kann. Der Agent muss hierbei selber abwägen zwischen der:

Exploration (Neue Aktionen ausprobieren um für die Zukunft zu lernen)

und der:

Exploitation (Bestehendes Wissen ausnutzen, um eine hohe Belohnung zu erzielen).

Der Agent entscheidet selbst wie er auf eine bestimmte Situation zu reagieren hat. Zu diesem Zweck hat der Agent eine bestimmte Strategie.

2.1 Agenten und Umwelt Interaktion

Beim Reinforcement Learning nimmt der Agent seine Umwelt wahr und agiert in ihr. Der Agent muss im Stande sein alleine aus den erhaltenen *rewards* und den erhaltenen Situationen eine Entscheidung zu treffen welche Aktion der Agent nun ausführen will.

Abbildung 1 veranschaulicht diese Interaktion noch einmal. Der Agent erhält zum Zeitpunkt t wie oben beschrieben die Situation S_t und einen *reward* r_t. Daraufhin wählt der Agent eine Aktion a_t aus und sendet diese Aktion an die Umwelt. Die Umwelt führt diese Aktion aus und erzeugt daraufhin eine neue Situation S_{t+1} basierend auf der gesendeten Aktion a_t. Der Agent erhält diese neue Situation s_{t+1} und zusätzlich einen positiven beziehungsweise negativen *reward* r_{t+1} als Belohnung oder Bestrafung.

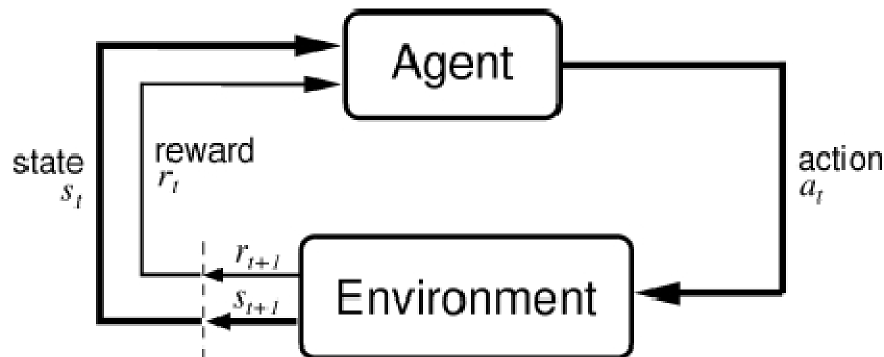

Abbildung 2.1: vgl. [SB98, Kap. 3.1]

Der Agent kann nun wenn er die in Abbildung 1 empfangenen Situationen voneinander unterscheiden kann, lernen welche Aktion ihm in einer Situation eine hohe Belohnung einbringt, oder welche Aktionen eher schlecht sind. Er trifft daraufhin die alleinige Entscheidung welche Aktion er in welcher Situation ausführt beziehungsweise wann er neue Aktionen ausprobiert oder sein bisheriges Wissen ausnutzt.

2.2 Das Beispiel der Grid World

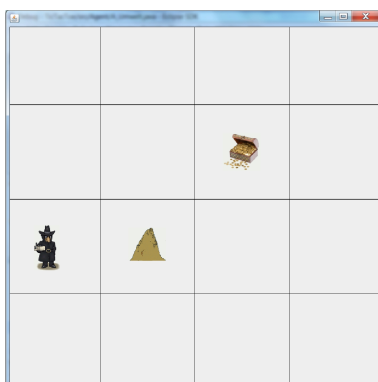

Abbildung 2.2: Die Grid World

Um das Beispiel einiger Algorithmen zu erklären wird oftmals das Beispiel der leicht verständlichen Grid World angesprochen. Dieses Beispiel basiert darauf das ein Agent sich durch eine mit quadratischen Feldern gestaltete Landschaft bewegt und lernen soll zu seinem Ziel zu kommen. Alle Situationen sind in diesem Beispiel als Koordinatensystem dargestellt. In der linken Abbildung befindet sich der Agent gerade im Zustand x:0 y:1. Sein Ziel befindet sich bei den Koordinaten x:2 und y:2.

2.3 Ziele und rewards

Ein Agent selber hat anfangs keine Ahnung was sein Ziel ist. Er bekommt lediglich einen *reward* nachdem dieser seine Aktion an die Umwelt gesendet hat. Der Agent kann so nur

langsam entdecken was sein eigentliches Ziel sein soll. Er wird versuchen auf lange Sicht die Summe der *rewards* zu maximieren, die ihm mitgeteilt werden. Den *reward* dazu zu benutzen das Ziel für den Agenten zu kennzeichnen, ist eine der wichtigsten Reinforcement Learning Aspekte. Ein spielender Schach Agent hat anfangs keine Ahnung was sein genaues Ziel ist. Erst wenn er verloren oder gewonnen hat erhält er einen *reward* von beispielsweise +100 bei einem Sieg oder -100 bei einer Niederlage. Da der Agent richtig lernen soll muss man dem Agenten die richtigen *rewards* zur Verfügung stellen. Denkbar wären auch Teilziele zu definieren, wie beispielsweise einem Schach Agenten eine Belohnung zu übergeben wenn dieser einen Großteil des Spielfeldes beherrscht. Allerdings wird laut [SB98] davor gewarnt die *rewards* auf Teilziele zu verteilen. Dies wird damit begründet das die *rewards* einzig und allein dazu dienen sollen dem Agenten mitzuteilen **was** er erreichen und nicht **wie** er es erreichen soll. Ansonsten läuft der Agent Gefahr das er nur lernt Teilziele zu erreichen aber das so das wirkliche Ziel nicht erreicht wird.

2.4 Returns

Das Ziel eines jeden Agenten ist es die *rewards* auf Dauer zu maximieren. In diesem Fall redet man hierbei von dem *Gesamt-reward*. Hier muss allerdings nun differenziert werden zwischen den *episodischen* und den *kontinuierlichen* Aufgaben. Bei einer episodischen Aufgabe wo der Agent nach jeder Episode einen Return erhält lässt sich der Return mit der folgenden endlichen Summe folgt ausdrücken:

$$R_t = r_{t+1} + r_{t+2} + r_{t+3} + ... + r_T. \tag{1.1}$$

wobei t den letzten Zeitpunkt kennzeichnet. Jede Episode endet hier in einer terminalen Situation.
Allerdings gibt es in vielen Umwelt und Agenten Szenarien keine erkennbaren Episoden. Diese laufen bis zu einem unendlichem Zeitpunkt durch. Da hier T = ∞ ist, würde der Return grenzenlos werden. Diesem Problem kann man allerdings mit einem Diskontierungsparameter entgegenwirken.

$$R_t=r_{t+1}+\gamma r_{t+2} + \gamma^2 r_{t+3} = \gamma^k r_{r+k+1}. \tag{1.2}$$

Hierbei ist γ der Diskontierungsparameter der dafür sorgt das der Gesamt-Return nicht unendlich groß wird. Wobei γ nur in dem Wertebereich $0 \leq \gamma \leq 1$ zulässig ist. Umso kleiner γ gewählt wird, umso schwächer werden die zukünftigen *rewards* gewichtet. Wenn man γ nähe 1 wählen würde, dann würden zukünftige Aktion stärker berücksichtigt werden.

2.5 Markov Entscheidungsprozesse

Eine Anwendung welche die *Markov-Eigenschaft* besitzt, hat bei jedem Situations-Übergang, Kenntnis über die kompletten Informationen über den Zustand der Welt, inklusive vergangener Entscheidungen. Eine Anwendung besitzt ebenfalls die Markov Eigenschaft wenn sie nicht alle Informationen über die Vergangenheit enthält, aber alle Informationen die relevant sind um das Eintreten der zukünftigen Wahrscheinlichkeiten zu berechnen.
Wenn S eine endliche Menge an Situationen ist, und man zur Vereinfachung annimmt das alle $a \in A$ in allen Situationen s $\in$ S durchführbar sind, dann kann man beobachten das der Agent zum Zeitpunkt t a_t ausführt das die Übergangswahrscheinlichkeit von $P^a_{ss'} = P_r\{s_{t+1} = s'|s_t = s, a_t = a\}$ und der voraussichtliche *reward* $R^a_{ss'} = E_r\{r_{t+1}|s_t = s, a_t = a, s_{t+1} = s'\}$ den der Agent erhält bekannt sind. Diese Folge realisiert einen Markov-Prozess. Zwar ist es nicht immer möglich die Markov Eigenschaft vollkommen zu erfüllen, sich dieser Eigenschaft allerdings stark anzunähern was oftmals ausreichend ist. [RLR96]

2.6 Value-Funktionen

Die Value-Funktionen sind eines der wichtigsten Kriterien beim Reinforcement Learning. Sie liefern einen Erwartungswert, für eine Situation oder ein Situations-Aktionspaar. Als Strategie bezeichnet man in diesem Fall die Aktionen die in bestimmten Situationen ausgewählt werden. Die Strategie wird von nun an als π gekennzeichnet. Eine Value Funktion ist von der Strategie abhängig und wird in Abhängigkeit von ihrer Strategie betrachtet:

$$V^\pi(s) = E_\pi\{R_t || s_t = s\} = E_\pi\left\{\sum \gamma^k r_t + k + 1 || s_t = s\right\}. \quad (1.3)$$

Die optimale Value Funktion V^* erreicht ein Agent bei der Verfolgung der optimalen Strategie π^*.

Gleiches gilt für die *Aktion-Wert-Funktion*. Diese Funktion besagt wie gut es ist die Aktion a, im Zustand s auszuwählen anhand der gewählten Strategie π.

$$Q^\pi(s,a) = E_\pi\{R_t || s_t = s, a_t = a\} = E_\pi\left\{\sum \gamma^k r_t + k + 1 || s_t = s\right\}. \quad (1.4)$$

Eine Strategie π ist besser oder gleich einer anderen Strategie π' wenn der erwartete Return größer oder gleich ist als der Return der Strategie π' für alle Situationen. Sprich $\pi \geq \pi$' wenn $V^\pi(s) \geq V^{\pi^|}(s)$ für alle s $\in$ S. Wenn also auf jeden Zustand, in der eine Aktion gewählt die die Summe des sofortigen *reward* und der Folgesituation s' maximieren nach der Formel:

$$\pi^*(s) = argmax[r + \gamma V^*(s')]. \quad (1.5)$$

kommt der Agent zu seiner optimalen Strategie.

.

2.7 Explorationsverfahren

Um das oben im ersten Abschnitt angesprochene Dilemma zwischen der Exploration und der Exploitation zu lösen werden hier verschiedene Strategien dargestellt um zwischen der Exploration und der Exploitation wählen.

2.8 ϵ greedy

Bei dem wohl bekanntesten Verfahren um zwischen Exploration und der Exploitation abzuwägen, wird dem Agenten ein konstantes ϵ übergeben. Dieses ϵ bestimmt die Verteilung zwischen Exploration und der Exploitation. Es ist eine Variable die dem Agenten im Vorfeld mitgeteilt wird, welche dann einen Wert zwischen 0,0 und 1,0 zugewiesen bekommt. 0,8 stünde dann dafür das zu 80% immer die Aktion des Agenten gewählt die für den Agenten am profitabelsten erscheint. Zu 20% wird der Agent nicht die beste Aktion auswählen. Stattdessen wird dieser eine zufällige Aktion wählen um neue Aktion auszuprobieren. Natürlich steht es dem Entwickler frei diesen Parameter mit der Zeit anzupassen. Nur ist diese Abschätzung wann und um wieviel man diesen Parameter abschwächt nicht leicht und hängt auch stark von der Umgebung des Agenten ab. In der Praxis wird wird meist ein kleines konstantes ϵ gewählt, um die ständige Exploration sicherzustellen, ohne das dabei das Finden einer optimalen Strategie verhindert wird.

2.9 Boltzmann-Exploration

Als zweite und letzte bekanntere Variante zur Abwägung zwischen Exploration und Exploitation soll noch die Boltzmann-Exploration vorgestellt werden. Bei dieser Strategie, die auch eine Zufallskomponente besitzt werden die Werte auf die V- beziehungsweise Q(s,a)-Werte bezogen. Hat der Agent 2 mögliche Aktionen zur Auswahl, Aktion a_1 und Aktion a_2 wobei a_2 einen höheren Q-Wert hat, ist die Wahrscheinlichkeit das a_2 ausgewählt wird höher. Ganz entscheidend hierbei ist so genannte Boltzmann-Temperatur t. Wird t zu hoch gewählt gewählt haben fast alle Aktionen die selbe Wahrscheinlichkeit ausgewählt zu werden. Wird die Temperatur zu niedrig gewählt dann besteht die Gefahr das nur noch die erfolgversprechendsten Aktionen ausgewählt werden. Der Algorithmus arbeitet nach der Formel[HA07,

S. 35]:

$$P(a|s) = \frac{e^{Q(s,a)/t}}{\sum_{a' \in A} e^{Q(s,a)/t}} \quad (1.6)$$

2.10 Dynamische Programmierung

Die dynamische Programmierung basiert auf der kompletten Kenntnis der Umwelt. Es wird hierzu die Markov-Eigenschaft benötigt. Laut [SB98] erhält die Dynamische Programmierung deshalb eher eine theoretische Bedeutung. Sie dient allerdings als Grundlage um die restlichen Bereiche des Reinforcement Learning zu verstehen.

Damit die folgenden Gleichungen anwendbar sind muss der Agent voraussetzen das das S und A(s) für s $\in$S endlich sind, und alle Übergangswahrscheinlichkeiten:

$$P^a_{ss'} = P_r\{s_{t+1} = s' || s_t = s, a_t = a\}. \quad (1.7)$$

bekannt sind. Außerdem ist es notwendig das die die voraussichtlichen *rewards*:

$$R^a_{ss'} = E\{r_{t+1} || a_t = a, s_t = s, s_{t+1} = s'\}. \quad (1.8)$$

ebenfalls bekannt sind, wobei s' $\in$ S eine terminale Situation darstellt.
Wenn die oben genannten Aspekte gelten, ist die Bellman Gleichung anwendbar. Diese existiert einmal als Situations-Wert- Funktion:

$$V^*(s) = max_a \sum P^a_{ss'}[R^a_{ss'} + \gamma V^*(s')]. \quad (1.9)$$

sowie als Situations-Aktions-Paar-Wertfunktion

$$Q^* = \sum^a_{ss'}[R^a_{ss'} + \gamma max_{a'} Q^*(s', a')]. \quad (1.10)$$

Zur Vertiefung können Details in [SB98, Kap. 3.8] nachgelesen werden.

2.11 Policy Evaluation

Der Algorithmus der Policy Evaluation beginnt damit das die State-Value-Funktion V^π mit einer willkürlich festen Strategie π initialisiert wird. Die darauf folgende Berechnung findet iterativ statt, indem die Funktionswerte dann approximiert werden. Der Agent initialisiert die V-Werte anfangs bei 0 und ersetzt dann den alten V-Wert durch den neu berechneten. Folgend, der komplette Algorithmus:

Der Algorithmus berechnet automatisch bis die Werte genau genug sind.

Algorithmus 1 vgl. [SB98, Kap. 4.1]

1: Als Eingabe eine beliebige π Strategie
2: Initialisierung V(s) = 0 für alle s $\in$ S
3: **repeat**
4: $\Delta \leftarrow 0$
5: **for all** s $\in$ S **do**
6: v $\leftarrow$ V(s)
7: V(s) $\leftarrow \sum_a \pi(s,a) \sum_{s'} P^a_{ss'}[R^a_{ss'} + \gamma V(s')]$
8: $\Delta \leftarrow max(\Delta, |v - V(S)|)$
9: **end for**
10: **until** $\Delta < \Theta$ (kleine positive Zahl)
11: Output V $\approx V^\pi$

2.12 Policy Improvement

Nach jedem einzelnem Durchlauf sollte der Agent sich überlegen ob es nicht günstiger ist eine andere Aktion zu wählen statt der Aktionen der vorher gewählten Strategie $\pi(s)$ zu wählen. Um herauszufinden welche Aktionen nun profitabler sind, betrachtet dieser nun die Aktions-Werte-Funktion. Nun sucht der Agent in jedem Zustand die Aktion, die die Aktions-Werte-Funktion maximiert. Wenn $Q^\pi(s,a) > V^\pi(s)$ ist, dann ist es eine andere Aktion zu wählen. Ansonsten hat der Agent schon die derzeit beste Strategie gefunden. Anhand der berechneten Werte mit der Policy Evaluation hat der Agent nun eine neue Strategie gefunden, die zumindest gleich oder sogar besser ist.

2.13 Policy Iteration

Nachdem der Agent nun eine anfängliche Bewertungsfunktion mit eine zufälligen Strategie errechnet hat, und diese nun verbessert hat, kann dieser langsam versuchen die optimale Strategie und eine optimalen Bewertungsfunktion zu finden. Mit der Policy Iteration ist dies schrittweise möglich. Der Agent wechselt immer wieder zwischen der Policy Evaluation und dem Policy Improvement. So werden die Werte und die Strategie π nach und nach besser. Der komplette Algorithmus sieht dann wie folgt aus:

Algorithmus 2 vgl. [SB98, Kap. 4.3]

1: Initialisierung
2: $V(s) \in \Re$ und $\pi(s) \in A(s)$ willkürlich für alle s $\in S$
3: Policy Evaluation
4: **repeat**
5: $\quad \Delta \leftarrow 0$
6: $\quad$ **for all** s $\in$ S **do**
7: $\quad\quad$ v $\leftarrow$ V(s)
8: $\quad\quad$ V(s) $\leftarrow \sum_{s'} P_{ss'}^{\pi(s)}[R_{ss'}^{\pi(s)} + \gamma$ V(s')]
9: $\quad\quad \Delta \leftarrow$ max(Δ, $\|v - V(s)|$)
10: $\quad$ **end for**
11: **until** $\Delta < \Theta$ (kleine positive Zahl)
12: Policy Improvement
13: *politik-stabil* $\leftarrow$ true
14: **for all** s $\in$ S **do**
15: $\quad b \leftarrow \pi(s)$
16: $\quad \pi(s) \leftarrow$ arg $max_a \sum_{s'} P_{ss'}^a [R_{ss'}^a + \gamma V(s')]$
17: $\quad$ **if** b $\neq \pi(s)$ **then**
18: $\quad\quad$ *politik-stabil* $\leftarrow$ false
19: $\quad$ **end if**
20: **end for**
21: **if** politik-stabil **then**
22: $\quad$ Gehe zu Policy Evaluation
23: **end if**

2.14 On- und Off-Policy

Bei den nächsten 2 vorgestellten Methoden Methoden wird durchgehend wieder zwischen den On- und den Off-Policy Methoden unterschieden. Der Unterschied zwischen diesen beiden Methoden liegt darin, das der Agent bei der Off-Policy 2 Strategien besitzt. Zum einen die *Verhaltens-Strategie* die für das Verhalten in der Umwelt verantwortlich ist. Die zweite Strategie ist die *Schätz-Strategie*. Diese wird von dem Agenten erlernt und nach und nach verbessert. Bei den On-Policy Methoden gibt es immer nur eine Strategie für das Verhalten und für die Schätzungen. Das Off-Policy Verfahren ist aufgrund dessen, das der Agent eine Strategie verbessert und eine andere ausführt wesentlich erfolgreicher. Denn so kann der Agent während er lernt noch weitere Aktionen ausprobieren.

2.15 Monte Carlo Methoden

Der wohl grundlegendste Unterschied zwischen den Monte Carlo Methoden und der dynamischen Programmierung ist, das bei den Monte Carlo Methoden im Vorfeld kein komplettes Modell der Umwelt bekannt sein muss. Der Agent lernt seine Umwelt erst kennen. Anfangs hat dieser kein Wissen wie sich bestimmte Aktionen auswirken und generiert erst einmal eine beliebige Entscheidungspolitik. Mit der Zeit gewinnt der Agent an Erfahrung und kann Situationen beziehungsweise Aktionen bewerten. Diese Werte setzten sich aus dem Durchschnitt aller Episoden zusammen. Dies bedeutet allerdings das die Monte Carlo Methoden auch nur auf episodische Aufgabenbereiche anwendbar sind.

2.16 First-visit Methode

Die first-vistit Methode kommt dann zum Einsatz wenn ein Zustand das erste mal besucht wird. Bei der first-visit Methode wird ein Mittelwert aus aus allen Returns einer Episode gebildet, ausgehend vom first-visit in jeder Episode.Der komplette Algorithmus lautet wie folgt:

2.17 Every-visit-Methode

Die every-visit-Methode kommt dann zum Einsatz wenn ein Zustand bereits besucht wurde. Ist dies der Fall, so wird nicht mehr der gemittelte return benutzt sondern eine eigene Upda-

Algorithmus 3 vgl. 3.3.1 [WESC]

1: Initialisiere
2: $\pi \leftarrow$zu evaluierende Politik
3: $V \leftarrow$beliebige Zustands-Wert-Funktion
4: returns(s) $\leftarrow$ leere Liste, für alle s $\in$ S
5: **while** endlos **do**
6: Generiere eine Episode unter Verwendung von π
7: **for all** Für jeden Zustand s der auf der Episode liegt **do**
8: Füge R in returns(s) ein
9: V(s) $\leftarrow$ gemittelte(returns(s))
10: **end for**
11: **end while**

teregel. Ein Beispiel hierfür wäre die konstant-α MC-Methode: α MC-Methode:

$$V^{\pi}_{neu}(s_t) = V(s_z) + \alpha[R_t - V(s_t)]. \quad (1.11)$$

[WESC, Kap 3.2]

2.18 On-Policy Monte Carlo Methode

Die On-Policy Methode arbeitet mit der Wahrscheinlichkeit von $\epsilon/|A(s)|$ das nicht die Aktion mit dem höchsten Q- beziehungsweise V-Wert gewählt wird, mit voraussichtlich höchstem Aktionswert auf diesem Pfad. So ist es möglich das alle Pfade im Grenzfall, zum Einsatz kommen für eine Politik.

2.19 Off-Policy Monte Carlo Methode

Bei der Off-Policy wird die Schätzung von dem Verhalten getrennt. Für eine Vertiefung wird hier auf [SB98, Kap 5.6] verwiesen.

2.20 Temporal Difference Learning

Temporal Difference Learning ist eine Mischung aus dynamischer Programmierung und den Monte Carlo Methoden. Hierbei wird anfangs ebenfalls kein komplettes Modell der Umgebung benötigt. Der Name Temporal Difference Learning basiert darauf das auf:

Temporal: Nach jedem Zeitschritt
Difference: die Werte verändert werden aufgrund der Differenzen
Learning: und somit eine Value-Funktion gelernt wird.

Man unterscheidet beim Temporal Difference Learning, genau wie bei den Monte Carlo Methoden, zwischen On-Policy und Off-Policy. Bei den On-Policy Methoden wird auch hier die optimierende Strategie zur Aktionswahl genutzt, während beim Off-Policy eine Strategie für die Aktionswahl verantwortlich ist und eine andere Strategie optimiert wird. Beide Verfahren werden kurz vorgestellt.

2.21 TD-Prediction

Der Agent lernt hierbei wie bei den Monte Carlo Methoden auch aus der Erfahrung, die dieser mit der Zeit sammelt. Die einfachste TD-Methode ist bekannt als TD(0).

$$V(s_t) \leftarrow V(s_t) + \alpha[r_{t+1} + \gamma V(s_{t+1)-V(s_t)}] \tag{1.12}$$

α steht hierbei für die Lernrate. Bei einem erhöhten α könnten vergangene Lernerfolge schnell verloren gehen, wohingegen ein kleines α die Gefahr verbirgt, das der Lernprozess sehr langsam verläuft.

2.22 Sarsa

Das Ziel des Sarsa-Algorithmus ist es Q* zu bestimmen indem der Agent sich langsam diesen Werten annähert. Der Sarsa-Algorithmus basiert auf der TD-Prediction. Bisher war es so, dass der Agent von Situation zu Situation gekommen ist und diese mit Werten versehen hat. Beim Sarsa-Algorithmus wird das aus der Situations-Aktions-8 Perspektive betrachtet. Die Schätzungen setzen sich bei diesem Algorithmus aus dem erhaltenen reward, der alten Schätzung und dem Nachfolgepaar zusammen. Ebenfalls ist hier auch ein Discountparameter γ vorhanden, der dafür verantwortlich ist, festzulegen wie viel der neuen Erfahrung in das aktuelle Wissen einbezogen werden soll.

2.23 Sarsa-On-Policy

Auch hierbei gilt wieder, wenn von On-Policy gesprochen wird, das die optimierte Strategie auch gleichzeitig der Aktionswahl entspricht. Der Algorithmus beginnt mit einer willkürlichen Strategie die nach und nach verbessert wird. Der Algorithmus für Sarsa-On-Policy sieht

folgendermaßen aus:

Algorithmus 4 vgl. [SB98, Kap. 6.4]

1: Initialisiere Q(s,a) beliebig
2: **for all** Episoden **do**
3: Initialisiere den aktuellen Zustand s zufällig
4: Wähle Aktion a für s gemäß der aus Q abgeleiteten Strategie.
5: **for all** jeden Schritt der Episode bis s terminal **do**
6: Führe Aktion a aus, beobachte den reward r und den Folgezustand s'
7: Wähle Aktion a für Folgezustand s' gemäß der aus Q abgeleiteten Strategie
8: $Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r + \gamma Q((s_{t+1}, a_{t+1}) - Q(s_t, a_t)]$
9: Setze: s $\leftarrow s_{t+1}$, $a \leftarrow a_{t+1}$
10: **end for**
11: **end for**

2.24 Sarsa-Off-Policy

Sarsa-Off-Policy basiert auf dem selben Prinzip wie der On-Policy Algorithmus, nur das auch hier wie bei jedem Off-Policy Algorithmus eine optimale Wertefunktion unabhängig von der Strategie berechnet wird. Daraus ergibt sich, das 2 parallele Strategien existieren. Laut [SB98, Kap. 6.5.] der wohl erfolgreichste Algorithmus:

Algorithmus 5 vgl. [SB98, Kap. 6.5]

1: Initialisiere Q(s,a) beliebig
2: **for all** Episoden **do**
3: Initialisiere den aktuellen Zustand s zufällig
4: Wähle Aktion a für s gemäß der aus Q abgeleiteten Strategie.
5: **for all** jeden Schritt der Episode bis s terminal **do**
6: Führe Aktion a aus, beobachte den reward r und den Folgezustand s'
7: Wähle Aktion a für Folgezustand s' gemäß der aus Q abgeleiteten Strategie
8: $Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma max_{a'} Q((s_{t+1}, a_{t+1}) - Q(s_t, a_t)]$
9: Setze: s $\leftarrow s_{t+1}$, $a \leftarrow a_{t+1}$
10: **end for**
11: **end for**

2.25 Eligibility-Traces

Oftmals besteht das Problem, das Belohnungen erst sehr spät oder am Ende einer Episode auftreten. In in dem Fall wird von einem *delayed reward* gesprochen. Bei dem in Abbildung 3 betrachteten Agenten einer Grid World kann man in der linken Bild seinen Weg zum Ziel nachvollziehen. Die mittlere Bild in der Abbildung zeigt das der Agent nur einen reward für seine letzte Aktion bekommen hat. Dies wäre bei allen bisher vorgestellten Algorithmen des Temporal Difference Learning der Fall. Bei dem im vorherigen Abschnitt dargestellten Sarsa Algorithmus würde der Agent die Belohnung nur für die Aktion erhalten die den Agenten zum Ziel geführt hat. Alle anderen Situations-Aktions-Paare werden dabei nicht aktualisiert.

So lernt der Agent zwar, dies allerdings nur sehr langsam. Im rechten Bild der Abbildung 3 kann man erkennen das der Agent für vergangenen Situations-Aktions-Paare eine immer weiter verminderte Belohnung erhält. So ist es dem Agenten möglich das Lernen maßgeblich zu beschleunigen. Um dieses beschleunigte Lernen zu erreichen, können laut [SB98] Eligibility-Traces verwendet werden.

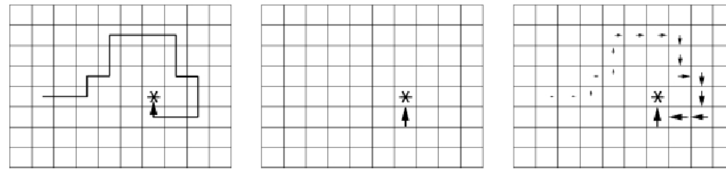

Abbildung 2.3: Quelle: [SB98, Kap. 7.5]

2.26 TD(λ)

Als erste Variante der Eligibility-Traces soll der backward view of TD(λ) betrachtet werden, da dieser sehr einfach zu verstehen ist. An dieser Stelle sei noch erwähnt, das das ein forward view existiert. Bei Vertiefungen sei hier auf [SB98, Kap 7.2] verwiesen. Die Funktion des backward views arbeitet nach dem Prinzip, das der vom Agent erhaltene reward an alle zurückliegenden Situationen beziehungsweise Situations-Aktions-Paare, die der Agent durchlaufen hat in immer weiter abgeschwächter Form weiter gibt.

Jede Situation beziehungsweise jedes Situations-Aktions-Paar besitzt einen eigenen Eligibility-Trace $e_t(s)$. Die folgende Darstellung geschieht aus der Situations-Perspektive, diese ist allerdings genauso auf Situations-Aktions-Paare anwendbar. Die Formale Definition des Backward View of TD(λ):

$$e_t(s) = \begin{cases} \gamma\lambda e_{t-1}(s) wenn\ s \neq s_t; \\ \gamma\lambda e_{t-1}(s) + 1 wenn\ s = s_t; \end{cases} \tag{1.13}$$

λ steht in diesem Fall für *Trace-Decay-Parameter*, zu deutsch Zerfallparameter. Dieser ist dafür verantwortlich, wie stark die zurückliegenden Situationen mit dem aktuellen reward aktualisiert werden. Ein hoher Wert an dieser Stelle sorgt dafür, dass der aktuelle reward einen großen Einfluss auf die zurückliegenden Situationen hat. Dieser Typ der Eligibility-Traces wird *Accumulating-Traces* genannt. Da der Eligibility-Trace bei jedem Besuch ansteigt. Wenn die Situation allerdings länger nicht besucht wird, nimmt dieser Schritt für Schritt ab wie in Abbildung 4 zu erkennen ist:

Um mit dem Eligibility-Trace alle vergangen Situationen für das aktuelle Ereignis zu belohnen, existiert der TD-Error δ. Dieser hat den Zweck, das damit die in der Vergangenheit besuchten Zustände ihre Belohnung oder Bestrafung erhalten:

$$V(s_{t+1}) \leftarrow V(s) + \alpha\delta e(s). \tag{1.14}$$

mit: $\delta = r_{t+1} + \gamma V_t(s_{t+1}) - V_t(s_t).$ (1.15)

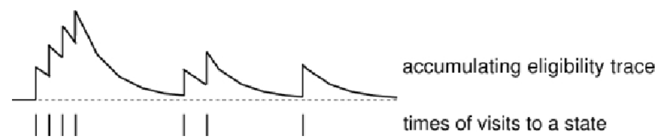

Abbildung 2.4: Quelle:[[SB98, Kap. 7.3]]

Um die Werte für alle Situationen weiterzugeben, wird dem altem Situations-Wert jeweils die abgeschwächte Belohnung hinzugefügt. Die nicht besuchten Paare erhalten in diesem Fall eine Belohnung von 0, da ihr E-Trace Wert ebenfalls 0 ist. Der komplette On-Line(beutetet dass das Update sofort stattfindet). TD(λ) Algorithmus zeigt wie nach jedem Situationsübergang die Belohnungen weitergegeben werden:

Algorithmus 6 vgl. [SB98, Kap. 7.3]

1: Initialisiere V(s) willkürlich und e(s) = 0, alle alle s $\in$ S
2: **for all** Episoden **do**
3: Initialisiere s
4: **for all** jeden Schritt der Episode bis s terminal **do**
5: a $\leftarrow$ wähle Aktion anhand π in der Situation s
6: $\delta \leftarrow r + \gamma V(s') - V(s)$
7: $e(s) \leftarrow e(s) + 1$
8: **for all** s **do**
9: V(s) $\leftarrow V(s) + \alpha\delta e(s)$
10: e(s) $\leftarrow \gamma\lambda e(s)$
11: **end for**
12: s $\leftarrow s'$
13: **end for**
14: **end for**

2.27 Sarsa(λ)

Der Sarsa(λ)-Algorithmus basiert auf dem in 1.8.3 vorgestellten Sarsa-On-Policy-Algorithmus, kombiniert mit Eligibility-Traces. Sarsa(λ) basiert im allgemeinen dem selben Prinzip, welches die TD(λ) Methode benutzt. Nur arbeitet der Sarsa(λ) Algorithmus mit Situations-Aktions-Paaren statt der Situations-Werten. Die Definition lautet:

Q_{t+1}(s,a) = $Q_t(s, a) + \alpha\delta_t e_t(s, a)$, für alle s,a

mit: $\delta = r_{t+1} + \gamma Q_t(s_{t+1}, a_{t+1} - Q_t(s_t, a_t)$

und: $e_t(s,a) = \begin{cases} \gamma\lambda e_{t-1}(s,a) & wenn\ s \neq s_t; \\ \gamma\lambda e_{t-1}(s,a) + 1 & wenn\ s = s_t; \end{cases}$ für alle s,a (1.16)

Sarsa(λ) ist ein On-Policy-Algorithmus, was bedeutet das die approximierte Strategie auch die aktuelle Strategie ist. Der komplette Algorithmus lautet:

Algorithmus 7 vgl. [SB98, Kap. 7.5]

1: Initialisiere Q(s,a) willkürlich, und e(s,a) = 0, für alle s,a
2: **for all** Episoden **do**
3: Initialisiere s,a
4: **for all** jeden Schritt der Episode bis s terminal **do**
5: Nehme Aktion a, beobachte r,s'
6: Wähle a' von s' anhand der benutzen Strategie von Q(z.B. ϵ-greedy)
7: $\delta \leftarrow r + \gamma$ Q(s',a') - Q(s,a)
8: e(s,a) $\leftarrow$ e(s,a) +1
9: **for all** s,a **do**
10: Q(s,a) $\leftarrow Q(s,a) + \alpha\delta e(s,a)$
11: e(s,a) $\leftarrow \gamma\lambda e(s,a)$
12: **end for**
13: $s \leftarrow s'; a \leftarrow a'$
14: **end for**
15: **end for**

2.28 Replacing-Eligibility-Traces

Eine Verbesserung für die Accumulating-Eligibility-Traces bieten die Replacing-Traces. Der Nachteil bei den Accumulating-Eligibility-Traces ist der das die einzelnen E-Trace Werte schnell in die Höhe wachsen können. Das Problem ist hierbei laut [SB98] das wenn der Agent durch eine falsche Aktion lange in einem Zustand bleibt oder diesen oftmals besucht, sich sein E-Trace Wert über eine gewisse Zeit stark erhöht. Erhält der Agent danach einen reward wird die Aktion die den Agenten zum Ziel geführt hat zwar belohnt, allerdings wird die vorherige Situation die einen mittlerweile viel höheren E-Trace-Wert hat viel stärker belohnt. Dies Problem lösen die Replacing-Eligibility-Traces:

$$e_t(s) = \begin{cases} \gamma\lambda e_{t-1}(s) & wenn\ s \neq s_t; \\ 1 & wenn\ s = s_t. \end{cases} \tag{1.17}$$

Die einzige Veränderung die vorgenommen wurde ist, das hier der E-Trace-Wert niemals höher als 1 werden kann.

Abbildung 5 zeigt den Akkumulierenden-Eligibility-Traces und die modifizierte Variante, den *Replacing-Eligibility-Trace*.

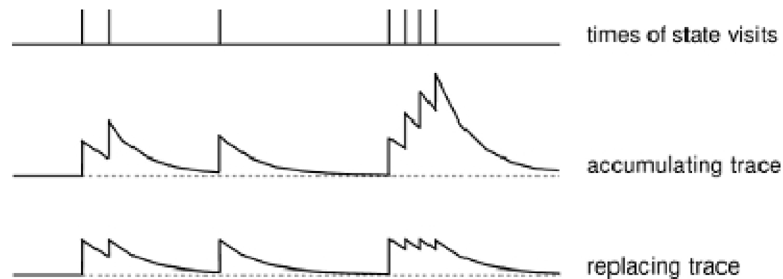

Abbildung 2.5: vgl. [SB98, Kap. 7.8]

2.29 Efficient-Eligibility-Traces

Allerdings sind auch die Accumulating-Eligibility-Traces nicht problemlos zu benutzen. Die Abbildung 6 zeigt, was passiert wenn der Agent 2 Aktionen zur Auswahl hat. Einmal eine Aktion a_r sowie eine Aktion a_w. a_r würde dafür sorgen das der Agent einen reward von +1 erhält. Die Aktion a_w führt nur dazu das der Agent in seiner derzeitigen Situation bleibt. Der Agent wählt nun mehrmals a_w aus, bevor er nun a_r wählt und die Belohnung erhält.

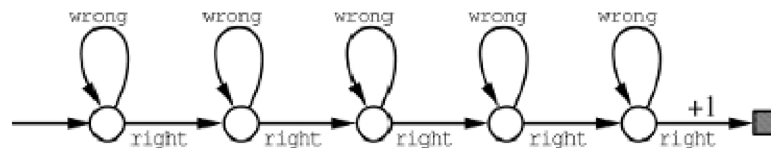

Abbildung 2.6: Quelle: [SB98, Kap. 7.8]

Dies hätte bei einem Replacing-Eligibility-Trace nun zur Folge, dass der E-Trace-Wert von e(s,wrong) höher ist als der E-Trace-Wert(s,right). Die Lösung für dieses Problem ist recht simpel. Wenn sich ein Agent in einem Zustand befindet und der Agent wählt eine Aktion, werden alle E-Traces für alle nicht gewählten Aktionen auf 0 gesetzt. Die formale Definition der Efficient-Eligibility-Traces:

$$e_t(s,a) = \begin{cases} 1 + \gamma\lambda e_{t-1}(s,a) & wenn\ s = s_t\ und\ a = a_t; \\ 0 & wenn\ s = s_t\ und\ a \neq a_t;\ ; \\ \gamma\lambda e_{t-1}(s,a) & wenn\ s \neq s_t \end{cases} \tag{1.18}$$

3 Agentensysteme

Dieses Kapitel soll in Kurzform Agenten und ihre Zusammenhänge betrachten. Anfangs soll kurz die generelle Definition eines Agentensystems erklärt werden. Woraufhin der 3-Phasen-Zyklus eines Agenten erklärt werden soll, nach dem ein Agent aufgebaut ist. Im Anschluss daran sollen kurz die Eigenschaften der Umwelt erläutert werden, in welcher sich ein Agent bewegt. In diesem Zusammenhang wird kurz erläutert nach welchen Aspekten sich laut [Bahlo07] eine Umwelt charakterisieren lässt. Ebenfalls soll die Definition eines lernenden Agenten laut [RN04] erklärt werden, da Teile der folgenden Analyse darauf aufbauen. Im letzten Abschnitt dieses Kapitels soll ein Framework für Agentensysteme ausgewählt werden und begründet werden, wieso die Wahl auf dieses Framework gefallen ist.

3.1 Grundlagen

Laut [GO03] bezeichnet der Begriff „Agent“ ein Programm welches eine gewisse Eigenständigkeit bei der Ausführung seiner Aufträge besitzt. Der Agent besitzt eine relative Abgeschlossenheit gegenüber seiner Umgebung und ein eigenständiges Verhalten. Die genaue Definition eines Agenten unterscheidet sich in der Literatur teilweise. Das ist darauf zurückzuführen da es sehr viele unterschiedliche Anforderungen an Agenten gibt. Da diese Arbeit sich darauf spezialisieren soll einzig und allein lernende Agenten zu betrachten, wird die Definition eines lernenden Agenten später noch einmal genauer betrachtet.

3.2 Die Autonomie der Agenten

Der Unterschied zwischen einem Objekt und einem Agenten wird in [WJ95] angesprochen. Der Unterschied besteht in der Autonomie sowie in der Art der Kommunikation. Die Methoden der einzelnen Objekte können von anderen Objekten aufgerufen werden. Ruft ein Objekt die Methode eines anderen Objekts auf, wird diese auch ausgeführt. Die Objekte selber haben keine Kontrolle darüber welche Methoden wann ausgeführt werden. Agenten hingegen treffen eine eigene Entscheidung ob sie eine Aktion ausführen oder nicht. Agenten können

verhandeln, Abmachungen treffen oder auch Bedingungen stellen, ob sie eine Aktion ausführen oder auch nicht. „Objects do it for free, Agents do it for Money“ heißt es laut [WJ95, S.10]. Da ein Agent keine Methoden hat und dieser seine eigenen Entscheidungen trifft, funktioniert die Kommunikation auch nicht wie zwischen Objekten. Agenten kommunizieren über Nachrichten. Diese können einzelne, festgelegte Textbausteine sein oder aber auch eigene Objekte.

3.3 3-Phasen-Zyklus

[GO03, Kap. 24.3.1] beschreibt den 3-Phasen-Zyklus eines Agenten. Ein Agent durchläuft immer wieder genau diesen Zyklus.

1.) Phase der Informationsaufnahme:
Im ersten Schritt werden hier die Informationen der Umwelt aufgenommen. Dies geschieht in Form von Nachrichten und Aufträgen.

2.) Phase der Wissensverarbeitung:
Im zweiten Schritt aktualisiert der Agent sein altes Wissen mit Hilfe der neuen Informationen. Er trifft eigenständig eine Entscheidung was zu tun ist.

3.) Phase der Aktionsausführung:
Die in Schritt 2 gewählte Aktion wird ausgeführt.

3.4 Umwelt des Agenten

Jeder Agent besitzt eine Umwelt in der dieser sich bewegt. Der Agent ist fähig Teile seiner Umwelt oder sogar die komplette Umwelt durch seine Sensoren wahrzunehmen, um die Situation in der dieser sich befindet zu charakterisieren. Der Agent agiert in seiner Umwelt. Die Umwelt allein allerdings definiert welche Aktionen möglich sind und welche nicht. Zum Beispiel kann ein Roboter nicht durch Wände laufen, was die Umwelt allerdings festlegen muss. Der Agent kann eventuell feststellen, dass eine Kollision bevorsteht. Nur trägt die Umwelt die Verantwortung dafür, das der Agent diese Wand nicht durchschreiten kann. Im nächsten Unterabschnitt wird die Umwelt noch einmal unter bestimmten Eigenschaften beleuchtet.

3.5 Charakterisierung der Umwelt

Laut [Bahlo07, Kap. 3.3] lässt sich die Umwelt in bestimmte Eigenschaften einteilen, die von großer Bedeutung für diese Arbeit sind. In Kapitel 4 soll analysiert werden welche Eigenschaften die Umwelt erfüllen kann und welche Eigenschaften in dieser Arbeit nicht realisiert werden.

- **Vollständig beobachtbar und teilweise beobachtbar**
  Eine vollständig beobachtbare Umgebung beutetet das ein Agent alle relevanten Informationen zu jedem Zeitpunkt erfassen kann. Ein Beispiel hierfür ist ein Schachspiel bei dem der Agent zu jedem Zeitpunkt alle Informationen abfragen kann. Ein Beispiel

hingegen für eine Umgebung die nur teilweise beobachtbar ist, ist ein Pokerspiel. Hier kann der Agent lediglich seine eigene Hand beobachten, und darf keinerlei Kenntnis darüber haben welche Blätter die anderen Agenten auf der Hand haben.

- **Deterministisch und Stochastisch**
  Wenn ein Agent genau festlegen kann mit welcher Aktion er in einen Folgezustand kommt, dann ist seine Umgebung deterministisch. Ist dies nicht der Fall, befindet sich der Agent in einer stochastischen Umgebung. Nimmt man als Beispiel einen Black-Jack-Agenten der eine neue Karte nimmt, hat dieser keine Kenntnis darüber in welchen Zustand er mit dieser Aktion kommt. Dies beutetet das seine Umgebung stochastisch ist. In einer einfachen Grid World, in der ein Agent im nächsten Schritt die Aktion „einen Schritt nach rechts gehen“ auswählt, weiß der Agent genau das er sich nach diesem Schritt ein Feld weiter rechts befindet. Dies ist ein Beispiel für eine deterministische Umwelt.

- **Episodisch und sequenziell**
  Eine sequentielle Umgebung beutetet dass eine Entscheidung die der Agent trifft alle weiteren Entscheidungen beeinflusst die darauf folgen. Es ist dem Agent hierbei nicht möglich seine Aktionsfolgen in Episoden zu gliedern. Bei einer episodischen Umgebung hingegen durchläuft der Agent eine Aktionsfolge, die damit allerdings irgendwann beendet ist. Damit ist diese Umwelt in einzelne Episoden aufgeteilt ist. Diese Episoden beginnen immer wieder mit ihren Initialwerten. Als Beispiel sei hier ein Agent genannt der Teile auf einem Fließband erkennt. Hat der Agent eine Tätigkeit, beispielsweise das Aussortieren eines Teiles abgeschlossen, beginnt er mit dem nächsten Teil das evtl. aussortiert werden soll. Das aussortieren des zweiten Teiles ist allerdings unabhängig davon, was beim ersten aussortieren passiert ist. Dies bedeutetet das seine Umwelt episodisch ist.

- **Statisch und dynamisch**
  Wenn sich die Umwelt eines Agenten verändert, während dieser eine Entscheidung trifft und ausführen möchte, spricht man von einer dynamischen Welt. Verändert sich die Umwelt einzig und allein aufgrund der Entscheidungen des Agenten, wird diese Umwelt statisch genannt. Ein Beispiel für eine statische Umwelt ist ein Sudoku-Agent, dessen Umgebung nur verändert wird wenn der Agent eine Aktion ausführt.

- **Diskret und stetig**
  Wenn die Menge der Zustände abzählbar ist, wie beispielsweise bei einer Grid World, befindet sich der Agent in einer diskreten Umwelt. Ein inverses Pendel dagegen besitzt eine stetige Umwelt, da die Menge an Zuständen nicht abzählbar ist.

- **Einzel-Agenten- und Multi-Agenten Umgebung**
  Einzel-Agent-Umgebung bedeutetet, das nur ein Agent in der Umgebung vorhanden

ist, beispielsweise bei einem Sokdu-Agenten. Multi-Agenten Umgebung bedeutet, das mehrere Agenten in einer Umgebung sind, die miteinander interagieren.

3.6 Arten von Agentenprogrammen

[RN04] zählt in seinem Buch die einzelnen Arten der Agenten auf. Diese Unterscheiden sich vor allem darin wie sie auf Veränderungen reagieren.

3.7 Einfacher Reflex-Agent

Bei dem wohl simpelsten Modell eines Agenten basieren die Aktionen die dieser ausführt auf einfachen Regeln. Diese hängen nur von dem aktuellen Zustand ab. Die vorherigen Aktionen haben keinen Einfluss darauf, welche Aktionen ausgeführt werden. Das Verhalten des Agenten basiert auf Wenn-Dann-Regeln die in einer Wissenstabelle abgespeichert sind.

3.8 Modellbasierter Reflex-Agent

Der modellbasiert Reflex-Agent speichert zusätzlich noch einen internen Zustand, indem frühere Auswirkungen gespeichert werden. Dieser benötigt dabei ein Modell der Welt oder zumindest einen Teil (der Welt) der ihm zeigt wie er diese wahrzunehmen hat.

3.9 Zielbasierter Agent

Der zielbasierte Agent hat zusätzlich zu den einfachen Funktionen eines modellbasierten Reflex-Agenten noch die Definition eines Zieles. Der Agent versucht Aktionen zu wählen, um dieses Ziel zu erreichen. Sollte das Ziel nicht mit dem nächsten Schritt erreichbar sein muss dieser beispielsweise planen oder nach dem Weg suchen, auf dem er dieses Ziel erreicht. Ebenfalls benötigt dieser Agent Informationen über sein Ziel und eine interne Bewertung, inwieweit er sich dem Ziel nähert.

3.10 Nutzenbasierter Agent

Damit ein Agent nicht nur das Ziel bewerten kann, sondern auch den Weg, hat diese Art Agent eine Nutzenfunktion um verschiedene Zustände mit dieser bewerten zu können. Sein Ziel ist in diesem Fall die Maximierung dieser Nutzenfunktion. Durch die vorhandene Nutzenfunktion die den Werte jeder Situation binär ausdrückt, ist es auch möglich das der Agent mehrere Ziele besitzen kann.

3.11 Lernender Agent

Laut [RN04] besitzt ein lernender Agenten ein Lern- sowie ein Leistungselement. Ersteres zielt darauf ab, mithilfe externer Kritik eine Verbesserung zu erzielen, während das Leistungselement für die Auswahl der Aktionen verantwortlich ist. Ein Lernprozess basiert dabei darauf dass der Agent Kritik von seiner Umwelt empfangen kann. Das Kritikelement übergibt das Feedback an das Lernelement, woraufhin dieses entscheidet ob das Leistungselement abgeändert werden soll oder nicht. In Abbildung 7 sind die Komponenten eines lernenden Agenten dargestellt. Ebenfalls benötigt ein lernender Agent einen Problemgenerator, der dafür sorgt das auch Aktionen ausgeführt werden, die auf den ersten Blick nicht optimal erscheinen, um bessere Aktionen in bestimmten Situationen zu erforschen.

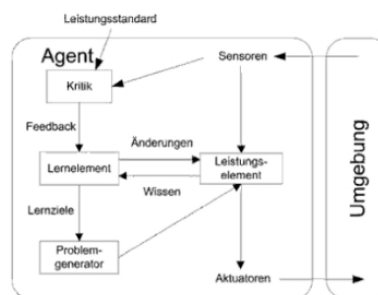

Abbildung 3.1: Quelle: [RN04, Seite 80]

Ein lernender Agent hat laut [WJ95] folgende Eigenschaften die ihn von einem normalen Software-Programm unterscheiden:

- **Situiert**: Die Bedeutung eines situierten Agenten steht dafür das dieser einen Bezug zu seiner Umwelt hat.
- **Autonom**: Ein autonomer Agent trifft seine Entscheidungen ohne direkten Einfluss eines anderen Agenten.

- **Sozial**: Soziale Agenten interagieren über einen Kommunikationskanal mittels einer festgelegten Sprache miteinander.
- **Reaktiv**: Ist der Agent reaktiv nimmt dieser seine Umwelt wahr und reagiert auf Veränderungen in ihr.
- **Proaktiv**: Proaktiv bedeutet dass ein Agent ein oder mehrere Ziele besitzt. Der Agent versucht selbständig diese(s) Ziel(e) zu erreichen.
- **Flexibel**: Der Agent ist flexibel, wenn er zum erreichen seiner Ziele auf mehrere Arten reagieren kann. Schlägt ein Versuch fehl, ist es dem Agenten möglich einen neuen Weg zu probieren, um dieses Ziel zu erreichen.
- **Robust**: Robust bedeutet, dass der Agent in der Lage sein muss Fehler zu erkennen und auf diese angemessen zu reagieren.

3.12 Agentensystem

Da es bereits mehrere Fameworks für Agentensysteme gibt, ist es sinnvoll diese Arbeit auf ein bestehendes Framework aufzusetzen. Die Wahl fiel auf Jade. Ein auf Java basierendes Framework. Die Wahl geschah aufgrund der folgenden Vorteile die Jade bietet [WE07]:

- **Interoperabilität**: Ein auf Jade basierendes System kann mit jedem anderem System kommunizieren, welches auf dem FIPA Standard basiert.
- **Anwendungsfreundlichkeit**: Die Middlewarekomponeten spielen keine Rolle bei der Benutzung. Es ist möglich, die von der API zur Verfügung gestellten Funktionen zu benutzen. Außerdem ist man nicht gezwungen, alle Teile Jades zu kennen, sondern nur die Teile zu nutzen die zur Implementierung nötig sind.
- **Uniformität und Portabilität**: Die API von Jade kann unabhängig von der verwendeten Java-Version und von dem zu Grunde liegenden Netzwerk benutzt werden.

4 Generische Aspekte

In diesem Teil der Analyse sollen

- die generischen Aspekte anhand des Interfaces in Abbildung 1 bestimmt werden. Dieser Teil der Analyse soll an den 3-Phasen-Zyklus aus Abschnitt 2.2 angelehnt werden.
- Ebenfalls soll ein Bezug zu der Definition eines lernenden Agenten von [RN04] in Abschnitt 3.6 hergestellt werden.

  In diesem Kontext sollen für die verschiedenen Algorithmen
  - das Leistungselement,
  - das Lernelement,
  - sowie der Problemgenerator

  definiert werden

Aus den in dieser Analyse dargestellten generischen Aspekten, soll es es möglich sein dieses Framework zu konstruieren. Es soll ebenfalls herauskristallisiert werden, ob sich bestimmte Punkte nicht vereinbaren lassen. Dies bedeutet, dass in dieser Arbeit auf einige Aspekte verzichtet wird. Ebenfalls wird durch das Modell eines lernenden Agenten eine Architektur geliefert, die es möglich macht die Struktur des Frameworks einfach zu erweitern. Beispielsweise soll es möglich sein, mit dem Austausch eines Lernelements und dem dazu passenden Leistungselement, einen zusätzlichen Algorithmus einzubauen, ohne das der Rest dieses Frameworks verändert werden muss.

4.1 Generische Aspekte anhand des Umwelt Agenten Interaktion Interface

In diesem Unterabschnitt soll der 3-Phasen-Zyklus aus Abschnitt 3.3 in Beziehung zu einem Reinforcement Learning Agenten gebracht werden. Es lässt sich erkennen, das in Anbetracht der Struktur aus Abbildung 1, der Agent im ersten Schritt eine Situation s_t übergeben bekommt. Ebenfalls erhält der Agent einen reward r_t. Diese Schritte lassen sich als die **Phase der Informationsaufnahme** definieren. Der Agent aktualisiert nun seine Werte intern. Er

wählt daraufhin die neue Aktion aus, die er im zukünftigen Zeitschritt ausführen möchte was die Phase der **Wissensverarbeitung** charakterisiert. Als letzten Schritt übergibt der Agent diese Aktion an die Umwelt, die diese dann ausführt. Dies ist in dem Fall die **Phase der Aktionsausführung**.

4.2 Das Leistungselement

An dieser Stelle der Analyse soll das Leistungselement für die dynamische Programmierung, sowie für das Temporal Difference Learning und die Monte Carlo Methoden aufgezeigt werden. Die letzten beiden Algorithmen werden im folgenden Unterabschnitt zusammengefasst. Dies begründet sich darin, dass sich das Leistungselement bei beiden Algorithmen nicht unterscheidet, da sowohl die Monte Carlo Methoden, als auch das Temporal Difference Learning mit den selben Datentypen arbeiten und sich nur in ihrer Art unterscheiden inwieweit sie diese Werte verändern.

4.3 Das Leistungselement anhand der Temporal Difference Learning und der Monte Carlo Methoden

Wie oben bereits erwähnt ist das Leistungselement für beide Algorithmen identisch, da beide Algorithmen die selben Informationen für die interne Verarbeitung benötigen. Allerdings muss bei den Monte Carlo Methoden und beim Temporal Difference Learning zwischen der V und der Q-Wert Implementation differenziert werden.

Für die V-Wert-Implementation besitzt der Agent eine Tabelle als Leistungselement, die alle V-Werte für die die bisher besuchten Situationen speichert. Die Tabelle muss zum Anfang nur ihren Initialzustand beinhalten. Die Tabelle könnte bei der Grid World nach ein paar Schritten folgendermaßen aussehen:

Als weiteren Aspekt muss der Agent allerdings sein Wissen darüber verwalten welche Aktion ihn zu welcher Aktion führt. Dafür muss dieser eine weitere Tabelle besitzen, die sogenannte *After-State-Tabelle*. Diese könnte für eine deterministische Welt folgendermaßen aussehen:

$s = s_t$	V
(0,0)	0,0
(1,0)	0,3
(0,1)	0,22
(1,1)	0,8
(2,2)	1

Abbildung 4.1: Leistungselement zur V-Wert-Implementation anhand der Temporal Difference Learning und der Monte Carlo Methoden

$s = s_t$	$a = a_t$	V
(0,0)	rechts	(1,0)
(1,0)	hoch	(1,1)
(1,0)	rechts	(1,1)

Abbildung 4.2: After-State-Tabelle zur V-Wert-Implementation anhand der Temproal Difference Learning und der Monte Carlo Methoden

Befindet sich der Agent in einer nicht deterministischen Welt, muss der Agent zusätzlich noch abspeichern, in wie viel Prozent der Fälle er mit einer bestimmten Aktion von der Situation s_t in die Folgesituation s_{t+1} gekommen ist. Die Eigenschaften, die der Agent beim Temproal Differnce Learning sowie bei den Monte Carlo Methoden anhand der V-Wert-Implementation für das Leistungselement besitzen muss sind die folgenden:

- Jede Situation s muss unterschieden werden können.
- Jede Situation s muss einen V-Wert besitzen.
- Eine weitere After-State-Tabelle ist erforderlich.
- Im Falle einer nicht deterministischen Welt muss diese eine Prozentangabe beinhalten, in wie viel Prozent der Fälle der Agent in die neue Situation kommt.
- Jede Situation muss eine Aktion besitzen, die in der aktuelle Situation ausgeführt wird. Dies kennzeichnet die Strategie.

Bei der Q-Wert-Implementation ändert sich im Bezug auf das Leistungselement, das nun nicht nicht mehr jede Situation unterschieden wird. Hierbei wird jede Situation, in Kombination mit jeder Aktion, als Wert unterschieden. Der Vorteil ist allerdings, dass der Agent nun keine Informationen mehr über erreichbare Situationen besitzen muss und damit keine After-State-Tabelle benötigt. Die Tabelle könnte für die Grid World dann folgendermaßen aussehen.

s_t	a_t	$Q(s,a)$
(0,0)	rechts	0
(0,0)	hoch	0,4
(0,1)	rechts	0,5
(2,2)	hoch	1

Abbildung 4.3: Leistungselement zur Q-Wert-Implementation anhand der Temproal Difference Learning und der Monte Carlo Methoden

Die Eigenschaften, die der Agent beim Temproal Differnce Learning sowie bei den Monte Carlo Methoden anhand der Q-Wert-Implementation für das Leistungselement besitzen muss sind die folgenden:

- Jede Situation S muss unterschieden werden können
- Zu jeder Situation gibt es einen Q-Wert im Bezug auf jede Aktion
- Jede Situation muss eine Aktion besitzen, die in der aktuelle Situation ausgeführt wird. Dies kennzeichnet die Strategie.

4.4 Leistungselement anhand der dynamischen Programmierung

Das Leistungselement ist bei der dynamischen Programmierung um einiges aufwändiger als bei den Monte Carlo Methoden oder beim Temproal Difference Learning. Für die dynamische Programmierung aus Abschnitt 1.7 ist es nötig, die Umwelt von Anfang an genau zu kennen, was auch die voraussichtlichen rewards, wie auch die Wahrscheinlichkeiten für alle Situationsübergänge beinhaltet. Die in Abschnitt 2.7.3 vorgestellte Policy Iteration benötigt somit eine Tabelle nach dem folgendem Schema, die die folgenden Werte enthält.

$$s = s_t,\ s' = s_{t+1}\ ,\ a = a_t,\ P^a_{ss'},\ R^a_{ss'}$$

Bei der oben dargestellten Grid World bräuchte ein Agent eine Tabelle dieser Art:

$s = s_t$	$s' = s_{t+1}$	$a = a_t$	$P^a_{ss'}$	$R^a_{ss'}$	V
(0,0)	(1,0)	rechts	1	0	0
(0,0)	(0,1)	hoch	1	0	0
(1,0)	(1,1)	rechts	1	0	0
(2,1)	(2,2)	hoch	1	1	1

Abbildung 4.4: Leistungselement der dynamischen Programmierung

Zusätzlich benötigt ein Agent noch eine Strategie, welche dieser abspeichern muss. Das Leistungselement für die dynamische Programmierung muss also folgende Eigenschaften erfüllen:

- Der Agent muss alle Situationen in Kombination mit alle möglichen Aktionen kennen und unterscheiden können.
- Der Agent muss für jede dieser Situationen einen V Wert speichern.
- Jede Situation muss eine Aktion besitzen, die in der aktuelle Situation ausgeführt wird. Dies kennzeichnet die Strategie.
- Der Agent muss alle Rewardwahrscheinlichkeiten zu jedem Zeitpunkt kennen.
- Der Agent muss alle Übergangswahrscheinlichkeiten der Situationen zu jedem Zeitpunkt kennen.

Festzuhalten bleibt an dieser Stelle das sich die Komplexität bei der dynamischen Programmierung stark erhöhen würde.

4.5 Das Lernelement anhand der verschiedenen Algorithmen

In diesem Abschnitt soll das Lernelement für jeden Algorithmus aufgezeigt werden. Da man für jeden Algorithmus einzelne Lernelemente definieren kann, hat dies den Vorteil, das diese einzeln ausgetauscht werden können. Das Lernelement ist nur dafür verantwortlich dem Leistungselement mitzuteilen, ob dieses seine Werte verändern soll und falls ja, wie. Für die bisher vorgestellten Algorithmen sind die Formeln stellvertretend für das Lernelement.

4.6 Lernelement anhand der dynamischen Programmierung

Die Betrachtung des Lernelements bei der dynamischen Programmierung wird in Bezug auf den in Abschnitt 2.7.3 vorgestellten Policy Iteration geschehen. Das Lernelement entspricht in dem Fall der Policy Evaluation einmal der Update Formel des Policy Improvement:

$V(s) \leftarrow \sum_{s'} P_{ss'}^{\pi(s)}[R_{ss'}^{\pi(s)} + \gamma V(s')]$

und einmal der Formel des Policy Improvements:

$\pi(s) \leftarrow \arg max_a \sum_{s'} P_{ss'}^{\pi(s)}[R_{ss'}^{\pi(s)} + \gamma V(s')]$.

Beide Formeln bestimmen wie das Leistungselement aktualisiert wird. Hierbei lässt sich festhalten das für alle Situationen und alle Situations-Aktionspaare die folgenden Werte bekannt sein müssen:

- Die jeweilig möglich folgenden Situationen:s'.
- Alle möglichen Aktionen a_1 bis a_n.
- Alle Wahrscheinlichkeiten von einer Situation in die nächste zu gelangen $P_{ss'}^a$.
- Sowie die Wahrscheinlichkeiten rewards zu erhalten $R_{ss'}^a$.
- Und alle V-Werte der Situationen.

4.7 Lernelement anhand der TD Prediction

Das Lernelement beim der TD Prediction ist die folgende Formel:

$$V(s_t) \leftarrow V(s_t) + \alpha[r_{t+1} + \gamma V(s_{t+1)-V(s_t)})] \quad (1.10)$$

4.8 Lernelement anhand von Sarsa-On-Policy

Das Lernelement für den Sarsa-On-Policy-Algorithmus wird durch folgende Formel ausgedrückt:

$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma Q((s_{t+1}, a_{t+1}) - Q(s_t, a_t)]$.

Festzuhalten an der Stelle ist, dass der eigentliche Algorithmus für die Algorithmen selber das Lernelement darstellt. Die benötigten Variablen der Algorithmen aus dem Bereich des

Temproal Difference Learning und der Monte Carlo Methoden benötigen sind:

- Die Werte für die Situations-Aktionspaare $Q(s_t, a_t)$ sowie $Q((s_{t+1}, a_{t+1})$
- und den gerade erhaltenen reward r_{t+1}.
- Ebenfalls werden die Konstanten α und γ benötigt.

4.9 Lernelement für den Sarsa-Off-Policy-Algorithmus

Das Lernelement für den Sarsa-Off-Policy-Algorithmus verändert sich nur geringfügig gegenüber dem Lernelement für den On-Policy-Algorithmus. Allerdings muss hier in Betracht gezogen werden das hierbei für s_{t+1} alle Situations-Aktionspaare benötigt werden:

$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma \max_a Q((s_{t+1}, a) - Q(s_t, a_t)]$.

Festzuhalten an dieser Stelle ist, dass der eigentliche Algorithmus selber das Lernelement darstellt. Die benötigten Variablen der Algorithmen aus dem Bereich des Temproal Difference Learning und der Monte Carlo Methoden sind die folgenden:

- Die Werte für die Situations-Aktionspaare $Q(s_t, a_t)$,
- sowie die Werte für die Situations-Aktionspaare $Q((s_{t+1}, a_1)$ bis $Q((s_{t+1}, a_n)$
- und den gerade erhaltenen reward r_{t+1}.
- Ebenfalls werden die Konstanten α und γ benötigt.

4.10 Der Problemgenerator

Die folgende Betrachtung der generischen Aspekte des Problemgenerators basiert auf der Annahme das nur die beiden Algorithmen des ϵ greedy Algorithmus sowie die Boltzmann Exploration betrachtet werden. Bei beiden Algorithmen ist festzuhalten das immer eine Variable existiert. Diese ist in beiden Fällen für die Wahrscheinlichkeit verantwortlich zu wie viel Prozent die scheinbar beste Aktion gewählt wird. Bei der Boltzmann Exploration wäre dies die Temperatur, während dies bei dem ϵ greedy Algorithmus das ϵ selber ist. So ist es auch beim Problemgenerator möglich die interne Berechnung zu jederzeit auszutauschen da die Schnittstelle immer gleich bleibt.

4.11 Ergebnis der Analyse

In diesem Analyse Kapitel wurde

- als erstes wurde der 3 Phasen-Zyklus eines Agenten betrachtet und auf einen Reinforcement Learning Agenten abgebildet.
- Daraufhin wurde ein lernender Agent laut [RN04] betrachtet, woraufhin die einzelnen Elemente in Bezug auf einen Agenten im Bereich Reinforcement Learning betrachtet wurden und die generischen Aspekte herauskristallisiert wurden.
- Als Ergebnis in Bezug auf das **Leistungselement** ist festzuhalten, dass die dynamische Programmierung ein komplexeres Leistungselement benötigt als die Monte Carlo Methoden und das Temporal Difference Learning. Ebenfalls wurde aufgezeigt das die Monte Carlo Methoden und das Temporal Differnce Learning das selbe Leistungselement besitzen. Das bedeutet wenn sich bei den Monte Carlo Methoden sowie beim Temproal Difference Learning jeweils das selbe Leistungselement vorhanden ist, dann reicht es für einen Wechsel des Algorithmus, lediglich das Lernelement zu tauschen.
- Daraufhin wurden kurz das **Lernelement** der einzelnen Algorithmen betrachtet. Als Ergebnis kam hierbei heraus das die Algorithmen selber das Lernelement darstellen.
- Am Ende wurden die generischen Aspekte des **Problemgenerators** betrachtet. Hier bleibt festzuhalten das der Problemgenerator bei den beiden betrachteten Algorithmen als generischer Aspekt jeweils eine Variable aufweist, die für die Wahrscheinlichkeit der Exploration verantwortlich ist.

5 Abgrenzung des Frameworks

In diesem Kapitel soll die Abgrenzung des Frameworks analysiert werden. Hier soll entschieden werden, welche Eigenschaften mit diesem Framework umgesetzt werden. Dieser Teil der Analyse soll in folgende Abschnitte unterteilt werden:

- Im ersten Unterabschnitt dieses Kapitels soll analysiert werden, welche der im ersten Abschnitt dieser Arbeit vorgestellten Algorithmen zum Einsatz kommen sollen und welche vielleicht nicht realisierbar sind. Ebenfalls stehen hier die Eigenschaften der Algorithmen zur Diskussion.
- Im folgenden Unterabschnitt sollen die Eigenschaften eines lernenden Agenten anhand von [WJ95] betrachtet werden, die in Abschnitt 3.6 erklärt wurden. Es soll analysiert werden, ob sich all diese Eigenschaften für diese Arbeit miteinander vereinbaren lassen.
- Im letzten Unterabschnitt sollen die Eigenschaften der Umwelt aus Abschnitt 3.4 betrachtet werden und analysiert werden, welche davon zu realisieren sind und welche nicht.

5.1 Auswahl der Algorithmen

Im ersten Abschnitt der Abgrenzung soll analysiert werden, welche der im ersten Abschnitt dieser Arbeit vorgestellten Algorithmen zum Einsatz kommen.

5.2 Dynamische Programmierung

Die dynamische Programmierung fordert um einiges mehr als das Temporal Difference Learning und die Monte Carlo Methoden. In Abschnitt 4.2 und 4.3 wurde aufgezeigt welche Werte bekannt sein müssen, um die Dynamische Programmierung zu implementieren. Dieser Abschnitt basierte allerdings auf einem trivialen Beispiel, in dem die Berechnung der Wahrscheinlichkeiten $P^a_{ss'}$ und $R^a_{ss'}$ recht leicht war. Nimmt man nun als Beispiel einen Black-Jack-Agenten an, dessen Situationen sich jeweils immer aus den zusammengezählten Punkte

ergeben, müsste dem Agenten $P^a_{ss'}$ schon im Vorfeld bekannt sein. Ansonsten könnte keine der Formeln zum Einsatz kommen. Ist die Situation nun so, das der Agent bei 15 Punkten steht und dieser noch eine Karte nimmt, muss in die Berechnung die Wahrscheinlichkeit mit einfließen, zu wie viel Prozent der Agent nun beispielsweise im Zustand 20 landen würde. Diese Berechnung müsste ebenfalls für alle weiteren erreichbaren Situationen durchgeführt werden. Ebenfalls müsste noch $R^a_{ss'}$ bekannt sein, sprich zu wie viel Prozent der Agent noch einen reward erhält. Dieser ist abhängig davon ob der Agent mit diesem Blatt gewinnt oder verliert. Um die genaue Prozentzahl zu erhalten würde der Agent noch eine Methode benötigen, die mit den restlichen Karten im Deck die Wahrscheinlichkeiten dazu berechnet. Dies verkompliziert die Berechnung ungemein. Hinzu kommt die Möglichkeit, das diese Berechnung extrem rechenintensiv wird. Ebenfalls wurde schon in Kapitel 2.6 erwähnt, dass der dynamischen Programmierung eine eher theoretische Bedeutung zu kommt. Hinzu kommt noch das bei jeder Anwendung die für dieses Framework die dynamische Programmierung als Algorithmus implementiert, die Umwelt von Anfang an bekannt sein müsste. Dies schränkt die Art der Anwendung, die man mit diesem Framework implementieren kann, erheblich ein. An dieser Stelle sind das genügend Gründe um die Dynamische Programmierung für dieses Framework auszuschließen.

5.3 Monte Carlo Methoden

Die Monte Carlo Methoden benötigen keine komplett bekannte Umwelt im Vorfeld und auch die Wahrscheinlichkeiten der Situationsübergänge und der Wahrscheinlichkeit vom Erhalt der rewards müssen im Vorfeld nicht bekannt sein. Ebenfalls sind bei den Monte Carlo Methoden die zur Berechnung benötigen Werte exakt die selben wie beim Temporal Difference Learning. Somit könnten die Algorithmen einfach ausgetauscht werden, während der Rest der internen Darstellung nicht verändert werden müsste. Allerdings wurde bereits erwähnt, dass die Monte Carlo Methoden nur für episodische Aufgaben benutzt werden können, was die Menge der implementierten Anwendungen extrem einschränkt. Für langwierige Episoden können die Monte Carlo Methoden außerdem extrem speicherintensiv werden, da die Zwischenwerte bis zum Ende einer Episode im Speicher gehalten werden müssen. Aus diesen Gründen erscheint es an dieser Stelle wichtiger, die Priorität auf verschiedene Algorithmen des Temproal Differnce Learning zu legen. Im Nachhinein wäre es immer noch möglich die Monte Carlo Methoden zu implementieren.

5.4 Temproal Differnce Learning

Das Temporal Differnce Learning ist das am meisten verbreitetste Lernverfahren im Bereich Reinforcement Learning. Erstens benötigten die Methoden des Temproal Differnce Learning im Vorfeld keine bekannte Umwelt wie die dynamische Programmierung und sie besitzen nicht die zwei großen Nachteile der Monte Carlo Methoden. Die Methoden des Temproal Differnce Learning sind auf kontinuierliche, sowie auf episodische Aufgaben anwendbar. Und sie sind nicht so extrem speicherintensiv, da die Werte nicht erst am Ende einer Episode aktualisiert werden sondern immer nach jedem Situationsübergang. Allerdings bieten sie alle Vorteile, die die Monte Carlo Methoden ebenfalls besitzen. An dieser Stelle der Analyse wird somit festgelegt das dieses Framework anhand der Temporal Difference Methode konstruiert wird. Die Frage die sich an dieser Stelle der Analyse stellt ist, welche Algorithmen implementiert wie werden sollen.

5.5 V- beziehungsweise Q-Wert-Implementation

Der Unterschied zwischen der V- und der Q-Wert-Implementation ist eher gering, wie im vorigen Kapitel aufgezeigt wurde. Ein sinnvolles Einsatzgebiet der V-Wert-Implementation wäre, wenn die Umwelt von Anfang an bekannt ist, während die Q-Wert-Implementation für beide Fälle sinnvoll ist. Für die V-Wert-Implementation ist noch eine weitere Tabelle nötig, die After-State-Tabelle, wie in Kapitel 4.3.1 aufgezeigt wurde. Die V-Wert-Implementation hat keinerlei Vorteile, die dieses Framework durch eine Q-Wert-Implementation nicht abdecken würde. Deshalb wird an dieser Stelle festgelegt, dass diese Arbeit sich lediglich auf die Q-Wert-Implementation beschränkt.

5.6 On- und Off-Policy

Die Unterschiede zwischen On- und Off-Policy wurden kurz im Abschnitt 2.8 angesprochen. Das Off-Policy Verfahren ist eher erfolgversprechend als das On-Policy Verfahren. Das liegt, wie in Abschnitt 2.8 bereits erwähnt, daran das während des Lernens der Agent noch weitere Aktionen ausprobieren kann und somit die Bewertungsfunktionen auch mit Aktionen aktualisiert werden können, die nicht ausprobiert wurden. So kann der Agent eine Strategie verfolgen, und dabei eine andere aktualisieren. Allerdings sind die Vorteile hier nicht allzu gravierend. Es bietet sich demnach ebenfalls an, einen On- und einen Off-Policy-Algorithmus zu implementieren. Somit wären beide Seiten abgedeckt und das Framework würde vielfältiger werden. Laut [SB98] ist der Sarsa-Off-Policy-Algorithmus der erfolgreichste Algorithmus. Außerdem bietet der Sarsa Algorithmus eine Off- und eine On-Policy Variante an. Und dieser

arbeitet mit Q-Werten. An dieser Stelle wird festegelgt das beide beide Sarsa-Algorithmen implementiert werden.

5.7 Eligibility-Traces

Auf Eligibility-Traces kann man natürlich nicht verzichten, da sie das Lernen sehr stark beschleunigen. An dieser Stelle stellt sich dann auch kaum noch die Frage welche Form der Eligibility-Traces benutzt werden. Im vorherigen Abschnitt wurde bereits festgelegt, das beide Sarsa-Algorithmen implementiert werden. In diesem Fall bietet es sich an den Sarsa(λ)-Algorithmus ebenfalls zu implementieren. Dieser kombiniert den Sarsa-Algorithmus mit Eligibility-Traces.

5.8 Explorationsverfahren

Das Explorationsverfahren ist dafür verantwortlich, zwischen der Exploration und der Exploitation abzuwägen. Es entscheidet, ob das bestehende Wissen ausgenutzt werden soll oder ob neue Wege ausprobiert werden sollen. An dieser Stelle der Analyse soll auch festgelegt werden, welche der oben vorgestellten Algorithmen für dieses Framework zum Einsatz kommen sollen oder ob es eventuell sinnvoll ist mehrere Elemente auszuwählen. Als wohl erfolgversprechendste Variante bietet sich die Boltzmann-Verteilung an. Diese enthält zwar auch eine Zufallskomponente, allerdings sind diese nicht rein zufällig, sondern richten sich nach den aktuellen Q-Werten. Allerdings ist die Auswahl der Aktionen wie bereits in Abschnitt 2.7.2 erwähnt laut [HA07] stark von der Temperatur abhängig . Die richtige Auswahl kann sehr kompliziert werden, und könnte zu eher schlechteren Ergebnissen führen. Das Ziel dieser Arbeit ist allerdings ein einfaches und leicht bedienbares Framework zu schaffen. Deshalb scheidet die Boltzmann-Exploration aus. Der ϵ-greedy Algorithmus hingegen ist leicht zu verstehen und die Werte sind leicht zu verändern. Als einziges Explorartionsverfahren wird in diesem Framework also der ϵ-greedy Algorithmus implementiert werden. Da diese Arbeit sich, wie oben erwähnt, nach dem Modell von [RN04] ausrichtet, ist es im Nachhinein immer noch möglich den Algorithmus leicht auszutauschen, da das Explorationsverfahren alleine im Problemgenerator gekapselt ist.

5.9 Ergebnis der Analyse

Im ersten Teil dieser Analyse schied die dynamische Programmierung aufgrund ihrer Komplexität und ihrer eher theoretischen Bedeutung aus. Die Monte Carlo Methoden ebenfalls,

da diese keine Vorteile aufweisen, die das Temproal Differnce Learning nicht aufweist. Stattdessen einige Nachteile wie, dass die Monte Carlo Methoden nur auf episodische Aufgaben anwendbar sind. Außerdem ist es im Nachhinein immer noch möglich diese zu implementieren. Das Temproal Differnce Learning eignet sich am ehesten für die Zwecke dieser Arbeit, da es alle Vorteile der beiden vorher genannten Methoden besitzt. Allerdings keine Nachteile. Ebenfalls wurde diese Arbeit auf die Q-Wert-Implementationen beschränkt. Diese Entscheidung fiel aus dem Grunde da die V-Wert-Implementation keinerlei Vorteile aufweist die die Q-Wert-Implementationen nicht hat. Außerdem sind die Sarsa-Algorithmen die wohl erfolgreichsten Algorithmen und alle Q-Wert-Algorithmen. Als Ergebnis wird deshalb festgehalten, das folgende Algorithmen implementiert werden.

- Sarsa-On-Policy,
- Sarsa-Off-Policy,
- Sarsa-(λ).

Die ersten beiden Sarsa-Algorithmen aufgrund dessen das sie Q-Wert Algorithmen sind. Der letzte, da dieser den Sarsa-Algorithmus mit Eligibility-Traces kombiniert, auf die in dieser Arbeit nicht verzichtet werden kann.
Als Explorationsverfahren wurde festgelegt das nur der

- ϵ-greedy-Algorithmus

implementiert wird, da dieser sehr einfach zu verstehen ist.

5.10 Beschaffenheit des Agenten

In diesen Unterabschnitt soll geklärt werden ob es in dieser Arbeit möglich ist, alle Eigenschaften eines lernenden Agenten in Anlehnung an [WJ95] zu erfüllen.

- **Situiert**: Ein Agent muss in dieser Arbeit jeder Zeit situiert sein. Da dieses Framework auf dem in Abschnitt 2.1 vorgestellten Interface basiert, ist sofort zu sehen das die Agenten einzig und allein mit ihrer Umwelt kommunizieren und diese Eigenschaft unumgänglich ist.
- **Autonom**: Der Agent wählt selbständig eine Aktion aus den ihm zur Verfügung stehenden Aktionen aus. Dies geschieht ohne das die Umwelt oder ein anderer Agent darauf Einfluss nimmt. Dies bedeutet, dass jeder Agent autonom handelt.

- **Sozial**: Die Agenten kommunizieren nur mit ihrer Umwelt. Sie kommunizieren zu keinem Zeitpunkt mit anderen Agenten. Allerdings verändern sie die Umwelt jedes mal durch ihre Aktionen die ausgeführt werden. Dies stellt ebenfalls eine Art der Kommunikation dar, auch wenn diese nur über einen Vermittler(die Umwelt) stattfindet. Somit erfüllen die Agenten auch diese Eigenschaft.
- **Reaktiv**: Da die Umwelt jeweils Situationen an den Agenten sendet und dieser daraufhin seine Werte intern aktualisiert und daraufhin eine Aktion auswählt, reagiert ein Agent immer auf Veränderungen seiner Umwelt. Was bedeutet das ein Agent in dieser Arbeit ebenfalls reaktiv ist.
- **Proaktiv**: Das Ziel der Agenten in diesem Framework wurde in Abschnitt 2.3 erklärt. Jeder Agent versucht seine rewards zu maximieren. Der Agent hat damit ein Ziel und diese Eigenschaft ist damit auch erfüllt.
- **Flexibel**: Ein Agent im Bereich Reinforcement Learning ist generell flexibel im erreichen seiner Ziele, solange er exploriert. Oder aber der Agent hat komplett approximierte Werte und der Agent weiß dieses bereits. Meist weiß der Agent dieses allerdings nicht sicher. Deshalb sollte egal welches der explorierenden Verfahren der Agent aus Kapitel 2.2 besitzt, immer eine ständige Exploration stattfinden. Dies bedeutet auch, dass das der Agent die Eigenschaft flexibel erfüllt.
- **Robust**: Diese Eigenschaft lässt sich mit den in 2.3 vorgestellten rewards realisieren. Betrachtet man als Beispiel einen Black-Jack-Agenten. Verliert dieser ein Spiel und erhält einen negativen reward, wird diese Entscheidung wahrscheinlich zukünftig nicht mehr so häufig von ihm ausgeführt werden. Somit reagiert der Agent angemessen und erkennt einen Fehler. Jeder Agent in diesem Framework ist somit ebenfalls robust.

5.11 Ergebnis der Analyse

Als Ergebnis kann man festhalten das alle Eigenschaften eines lernenden Agenten laut [WJ95] erfüllt sind.

5.12 Beschaffenheit der Umwelt

In diesem Abschnitt soll beleuchtet werden, welcher der in Abschnitt 3.4.1 angesprochenen Aspekte in dieser Arbeit wie umgesetzt werden sollen.

5.13 Deterministische und stochastische Umwelt

Sicherlich muss man sich bei diesem Framework die Frage stellen, ob deterministische, stochastische oder eventuell sogar beide Umgebungen umgesetzt werden sollen. Im Abschnitt 5.1.1 bis 5.1.3 wurde das Thema schon einmal angesprochen. Es wurde aufgezeigt das nur die Dynamische Programmierung im Vorfeld alle Wahrscheinlichkeiten kennen muss. Für die Monte Carlo Methoden und das Temporal Differnce Learning hingegen ist es nicht relevant, ob die Welt stochastisch oder deterministisch ist. Auch wurde bereits festgelegt, dass die dynamische Programmierung ausgeschlossen wird. Somit kann festgelegt werden, das deterministische sowie stochastische Aufgaben in dieser Arbeit möglich sind, da es für die in dieser Arbeit benutzte Methode des Temproal Differnce Learning nicht von Bedeutung ist.

5.14 Grad der Beobachtbarkeit

Da dieses Framework rein Software-basiert ist, ist mit Signalstörungen nicht zu rechnen. Das Agentensystem, auf dem das Framework basiert, ist dafür verantwortlich sicherzustellen, dass die an den Agenten geschickten Nachrichten oder Objekte ankommen. Anders sieht es aus wenn man die Beobachtbarkeit anhand der wirklichen Umwelt betrachtet, beispielsweise auf das Pokern bezogen. Hier darf ein Spieler niemals die Hand des anderen kennen. An dieser Stelle muss festgelegt werden, ob das Framework zu jedem Zeitpunkt entscheiden kann, welchem Agenten es welche Information über die Welt vermittelt. Oder aber ob jeder Agent eine einheitliche, komplette Situation erhält. Da Aufgaben, bei denen ein Agent es nur mit einer teilweise beobachtbaren Umwelt, einen nicht unerheblichen Teil solcher Anwendungen darstellen wird hier festgelegt, dass das Framework den Grad der Beobachtbarkeit zu jedem Zeitpunkt selber festlegen kann. Um dies zu erreichen, muss die Umwelt für jeden Agenten eine eigene Situation verwalten. Dies kann in Anbetracht gewisser Aufgaben zwar recht kompliziert werden, dennoch ist diese Art der Anwendung unumgänglich.

5.15 Statische und dynamische Umgebungen

Es wird schnell klar das dieses Framework statische, sowie dynamische Umgebungen unterstützen muss. Diese Tatsache, das beide Aufgaben unterstützt werden können, ist allerdings schon mit der Tatsache gegeben, dass die Umwelt alle Daten intern verwaltet und diese zu jedem Zeitpunkt verändern kann. Aufgrund dieser Tatsache sind statische, sowie dynamische Umgebungen implementierbar.

5.16 Episodische und kontinuierliche Aufgaben

Ebenfalls muss entschieden werden ob endliche, kontinuierliche oder beide Aufgabentypen in diesem Framework implementiert werden sollen. Die Unterschiede zwischen kontinuierlichen und episodischen Aufgaben sind recht gering. Die einzige Bedingung hierbei wäre das ein initialer Zustand abgespeichert werden müsste und es wäre notwendig das die Option eine Situation s_t als einen finalen Zustand zu deklarieren vorhanden wäre. Wobei ein finaler Zustand einzig und allein den Effekt hätte, dass die folgende Situation s_{t+1} wieder ein initaler Zustand wäre. Wenn diese Option in dieses Framework implementiert werden würde, wäre dies generisch genug um episodische als auch kontinuierliche zu implementieren. Um die rewards aus den Formeln 1.1 und 1.2 aufzulisten, denen allerdings eine eher theoretische Bedeutung zukommt, gibt es die Möglichkeit die Formel 1.2 für beide Aufgabentypen zu benutzen. Da diese für kontinuierliche wie auch für episodische Aufgaben ihre Gültigkeit hat. Die beiden Fälle lassen sich somit zusammenfassen mit der Formel:

$$R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} = \gamma^k r_{r+k+1} \tag{1.2}$$

wobei $\gamma = 1$ ist, wenn ein terminaler Zustand vorhanden ist. Ebenfalls muss noch eine kleine Änderung an dem generellen Entwurf aus Abbildung 1 vorgenommen werden. Der Agent muss ebenfalls Kenntnis darüber haben, in welcher Episode dieser sich befindet. Die Angabe ist der bei der Berechnung unerlässlich. Die Umwelt hat allerdings sicherzustellen, dass diese Verwaltung der Nummerierung der Episoden korrekt verläuft. Deshalb muss die Umwelt dem Agenten nicht nur jedes mal seinen reward r_t und seine Situation s_t mitteilen, sondern auch als weiteres Argument noch e_t . Dieses weitere Argument kennzeichnet die Episode zum Zeitpunkt t und setzt den Agenten darüber in Kenntnis, in welcher Episode dieser sich befindet. Dies ist unerlässlich um die Berechnungen sicherzustellen, beispielsweise für Eligibility-Traces die diesen Parameter zur Berechnung benötigen. Ebenfalls ist die Eigenschaft unabdingbar, da bereits festgelegt wurde, dass dieses Framework später für die Monte Carlo Methoden erweiterbar sein soll.

5.17 Diskrete und stetige Zustandsräume

Diskrete Mengen an Situationen in diesem Framework müssen natürlich darstellbar sein, da sie die einfachste Form der Menge an Situationen darstellen. Auf stetige Zustandsräume wird in dieser Arbeit verzichtet, da die Darstellung zu komplex werden würde. Als Möglichkeit, stetige Zustandsräume zu implementieren stehen beispielsweise neuronale Netze zur Verfügung.

5.18 Einzel- oder Multi-Agenten Umgebung

Eine Multi-Agenten Umgebung ist bereits vorhanden sobald mehr als ein Agent in einer Umwelt Aktionen ausführt. Es lässt sich an dieser Stelle leicht vermuten, dass der Einsatzbereich bei einer reinen Einzel-Agenten Umgebung stark beschränkt sein würde. Viele Anwendungen wie beispielsweise schon ein einfaches Tic-Tac-Toe Spiel würden eine Multi-Agenten Umgebung voraussetzen. Der Unterschied an dieser Stelle wäre das die Umwelt eine festgelegte Reihenfolge hätte in der diese Situationen, rewards und die Nummerierung in welcher Episode sich die Agenten befinden versendet und Aktionen ausführt. Den bei einigen Szenarien wie schon bei 2 trivialen Mau-Mau Agenten könnte diese Reihenfolge unter Umständen variieren. An dieser Stelle ist es unausweichlich festzulegen, dass dieses Framework eine Einzel-Agenten Umgebung sowie für eine Multi-Agenten Umgebung kompatibel sein soll.

5.19 Ergebnisse der Analyse

Im vorangegangen Kapitel wurde die Beschaffenheit der Umwelt analysiert und welche Aspekte diese enthalten soll. Das Ergebnis der Analyse ist, dass in der Umwelt in dieser Arbeit:

- deterministische und stochastische Anwendungen möglich sind.
- Ebenfalls kann der Grad der Beobachtbarkeit variieren, was bedeutetet das eine Umwelt für einen Agenten nur teilweise beobachtbar sein kann.
- Auch wird dieses Framework für endliche sowie kontinuierliche Aufgaben Aufgaben ausgelegt werden soll.
- Statische sowie dynamische Umgebungen sind für dieses Framework ebenfalls angedacht, da die Umwelt alle Änderungen alleine verwaltet und beide Optionen somit leicht realisierbar sind.
- Weiterhin wurde festgelegt das stetige Zustandsräume in dieser Arbeit aufgrund ihrer Komplexität nicht berücksichtigt werden.
- Als letzten Aspekt wurde analysiert das dieses Framework für Einzel- sowie auch Multi-Agenten Umgebungen geeignet sein wird.

6 Implementation

In diesem Kapitel soll der interne Aufbau dieses Frameworks aufgezeigt werden. Diese Anschauung soll aus der Sicht der Implementation geschehen. Ebenfalls soll am Anfang dieses Kapitels noch analysiert werden, inwieweit sich die Umwelt in diesem Framework als eigener Agent verhält. Daraufhin soll die generelle Architektur des Agenten, sowie der Umwelt festgelegt werden. Im drauffolgenden Unterabschnitt werden dann die wichtigsten Klassen kurz erläutert.

6.1 Generelle Aspekte der Umwelt und des Agenten

In diesem Abschnitt sollen die Interaktionen zwischen dem Agenten und der Umwelt geklärt werden. Im ersten Teil wird die Frage aufgeworfen werden, ob die Umwelt als eigenständiger Agent fungieren soll oder nicht.

6.2 Umwelt als Agent

In dieser Analyse soll herauskristallisiert werden, ob die Umwelt als Agent fungiert oder ob die Umwelt eher als ein normales Programm zu implementieren ist. Um dies herauszufinden sollen die wichtigsten Aspekte eines Agenten mit den Eigenschaften der Umwelt verglichen werden. Es sollen die wichtigsten Eigenschaften eines Software-Agenten betrachtet werden und analysiert werden inwieweit die Umwelt diese Aspekte erfüllt. Betrachtet werden sollen die wichtigsten Aspekte die laut [KR08] einen Software-Agenten charakterisieren. Diese sind flexibel, autonom, proaktiv, interaktiv, situiert und sozial. Ebenfalls soll noch ein Bezug zu speziellen Jade-Agenten hergestellt werden.
Folgende Punkte davon sind erfüllbar:

- Die Umwelt ist komplett **autonom** und trifft jede Entscheidung ohne das Zutun anderer Agenten.
- Die Eigenschaft **sozial**, erfüllt die Umwelt ebenfalls da sie durchgehend mit anderen Agenten in Verbindung steht.

- Ebenfalls ist die Umwelt **proaktiv**, da sie das Ziel hat Situationen zu verwalten und auszuführen. Die Umwelt handelt rational um dieses Ziel zu erreichen.
- Die **Interaktivität** ist aus dem Grund gewährleistet, dass die Umwelt selber ihre Eigenschaften verändern kann. Außerdem nimmt sie Nachrichten der Agenten entgegen und reagiert dementsprechend darauf.
- In dieser Arbeit wird das Agentensystem Jade benutzt, welches auf behaviours basiert. Diese behaviours definieren das Verhalten eines Agenten. Die Umwelt wird in diesem Framework ebenfalls mit behaviours arbeiten. Dies bietet sich gut an, um die korrekte Synchronisierung sicherzustellen. Ebenfalls sind behaviours in einem Jade-Agenten dafür geeignet die Kommunikation sicherzustellen. Deshalb wird die Umwelt intern wie ein Jade-Agent aufgebaut werden.

Folgende Punkte können nur teils oder gar nicht erfüllt werden:

- Die **Flexibilität** kann nur teilweise gewährleistet werden. Zwar reagiert die Umwelt auf Ereignisse. Allerdings sind dies nur Aktionen die von Agenten gesendet werden.
- Die Umwelt ist nicht **situiert**. Sie interagiert mit keiner anderen Umwelt sondern nur mit sich selbst.

Ergebnis der Analyse ist das die Umwelt zum Großteil viele Aspekte eines Agenten unterstützt, außerdem ist der Punkt der behaviours komplett erfüllt. Das bedeutet die Umwelt ist in diesem Framework wie ein Agent implementiert. Alle Punkte die sich allerdings auf die Umwelt beziehen, wie beispielsweise situiert, können nicht erfüllt werden, was allerdings eher daran liegt das diese Punkte sich auf eine externe Umwelt beziehen.

6.3 Art des Agenten

In diesem Abschnitt soll kurz darüber diskutiert werden, welche Architektur des Agenten in dieser Arbeit zum Einsatz kommt.

In Abschnitt 3.3 wurde bereits ein lernender Agent angesprochen. Nur sagt das nicht alles über die interne Architektur eines Agenten aus. Der Agent muss ebenfalls wissen wie er Entscheidungen trifft oder wie er Situationen bewertet. Ein Reflex-Agent scheidet an dieser Stelle aus, da dieser nur nach bestimmten Aktionen auf Regeln reagiert. Ein zielbasierter Agent hat dem Reflex-Agenten voraus, dass dieser ein Ziel hat, was er erreichen soll. Er versucht mit Hilfe dieser Ziele seine Aktionen zu wählen, um dieses Ziel zu erreichen. Allerdings muss der Agent im Stande sein, binär zwischen guten und schlechten Zuständen, unterscheiden. Was bedeutet das jeder Zustand einen Nutzwert hat. Ebenfalls muss es das Ziel des Agenten sein, diese Werte zu maximieren. Für die interne Architektur eines Agenten

wird also ein nutzenbasierten Agenten benötigt. Dieser Teil der Architektur wird sich später im Leistungselement wiederfinden.
Ergebnis der Analyse ist, dass die Architektur eines Nutzenbasierten-Agenten übernommen werden kann. Dies ist nötig um binär zwischen guten und schlechten Zuständen zu unterscheiden.

6.4 Abbild der Umwelt

Ebenfalls muss an dieser Stelle noch eine Analyse darüber folgen, ob der Agent zu jedem Zeitpunkt eine komplette Abbildung seiner Umwelt besitzt oder nicht.

Der Agent braucht zu keinem Zeitpunkt ein komplettes Abbild der Umwelt. Oftmals reicht es aus oder ist sogar wichtig, dass der Agent nur Teile seiner Umwelt kennt. Spielt der Agent beispielsweise Poker, darf der Agent gar nichts über die anderen Hände wissen. Die Umwelt alleine muss verwalten welcher Agent welche Informationen bekommt, ansonsten wären nur Szenarien zu modellieren, die einer vollständig beobachtbaren Umwelt nachempfunden sind. In Kapitel 5.4.2 wurde allerdings bereits festgelegt, dass auch eine teilweise beobachtbare Umwelt modelliert werden soll. Der Agent bekommt die Informationen allerdings als Situationen von der Umwelt gesendet. Die Umwelt kann also alleine darüber verfügen, welche Aspekte der Situation sie sendet und welche Informationen die Umwelt dem Agenten bereitstellt.

Ergebnis der Analyse ist, dass die Umwelt alleine dafür verantwortlich sein muss welche Informationen sie dem Agenten über eine Situation zur Verfügung stellt. Diese Informationen müssen auch aus dem Grund variieren, dass der Agent auch eine teilweise beobachtbare Umwelt vorfinden können muss.

6.5 Sonstige Klasse

Für dieses Framework ist es wichtig das die Agenten und auch die Umwelt die verschiedenen Situationen und Aktionen unterscheiden können. Für diesen Fall besitzt das Framework die abstrakten Klassen A_Situation und A_Aktion. Jedes konkrete Objekt beider Klassen muss durch eine ID jeweils eindeutig gekennzeichnet werden. Ebenfalls vorhanden ist eine Kombination aus den beiden Klassen mit dem Namen A_Situation_Aktion. Da diese Klasse von den Agenten, sowie von der Umwelt benutzt werden, sollen diese zuerst beleuchtet werden.

6.6 A_Situation

Die Klasse *A_Situation* ist eine abstrakte Klasse. In einer konkreten Unterklasse muss für die Situationen lediglich die Methode zur Generierung einer ID überschrieben werden und die Parameter der Situation definiert werden. Um das Beispiel einer Grid World zu beschreiben muss die konkrete Klasse der Situation lediglich die Methode generiere_ID() implementieren und die Merkmale int x, sowie int y zur Beschreibung der Situation aufweisen.

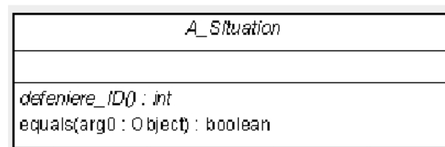

Abbildung 6.1: Die Klasse A_Situation

Die Methode generiere_ID() muss für jede Situation einen eindeutigen Wert zurückgeben. Bei der Grid World wäre dies beispielsweise eine Zahl zwischen 1 und 1000, wobei die ersten beiden Zahlen für den y Wert und die letzten beiden für den x Wert stehen. Diese Methode für die Beispielanwendung der Grid World sieht folgendermaßen aus:

```
 public int definiere_ID() {
   return x + (100 * y);
}
```

6.7 A_Aktion

Alle Aktionen die für die Agenten ausführbar sind müssen im Vorfeld als konkrete Objekte erzeugt werden. Die Aktionen sind selber über eine ID definiert. Die Klasse A_Aktion von der alle Aktionen abgeleitet sind, besitzt die abstrakte Methode A_Situation fuehre_Aus(A_Situation situation) die von der konkreten Unterklasse implementiert werden muss.
In ihr muss beschrieben werden was diese Aktion auslöst. Als Beispiel hierfür ein Auszug aus dem Beispiel der Grid World für die Aktion rechts:

```
public Situation fuehre_Aus(A_Situation situation)
Situation tmp = (Situation) situation;
```

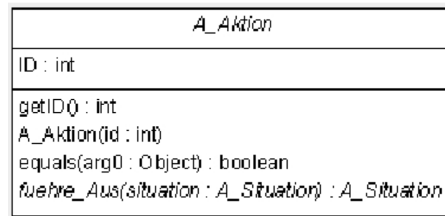

Abbildung 6.2: Die Klasse A_Aktion

```
int x = tmp.getX();
if(x < 3)
   x = x+1;
else
   x = 0
tmp.setX(x);
return tmp;
```

6.8 A_Situation_Aktion

Um Situationen in Kombination mit ihren Aktionen zu speichern, beispielsweise für die Wertetabelle des Agenten, ist es nötig ein Objekt aus beiden zu definieren. Ebenfalls kann der Agent nur so seine eigene History abspeichern. Diese Klasse besteht jeweils aus einer Situation und einer Aktion.

6.9 Aufbau der Umwelt

In diesem Kapitel soll ein kurzer Überblick darüber gegeben werden, welche Elemente die Umwelt besitzen soll. Als generelle Oberklasse für die Umwelt dient die Klasse A_Umwelt. Als wichtigste Aspekte sollen hier kurz die Elemente zur:

- Kommunikation,
- sowie das Element zum Ausführen von Aktionen

angesprochen werden, sowie die weiteren Optionen die dieses Framework bietet, welche u. a. wären:

- Methode für verbotene Aktionen,

- die Mittel zur Synchronisierung,
- und die Methode zum bewerten aller Situationen.

6.10 Elemente zur Kommunikation

Die Umwelt selber besteht aus 4 Elementen, um mit den registrierten Agenten zu kommunizieren. All diese Elemente sind eigene behaviours die miteinander synchronisiert werden müssen. Die Reihenfolge in welcher diese behaviours ausgeführt werden, muss der Benutzer im Vorfeld festlegen

Als erstes sei hier das *Kommunikationselement* genannt. Es ist dafür verantwortlich zu jedem Zeitpunkt die Aktionen der Agenten zu empfangen. Ebenfalls prüft es nach jeder empfangenen Aktion, ob es für den Agenten in diesem Moment möglich ist diese Aktion auszuführen. Ist dies nicht möglich, teilt die Umwelt dieses dem Agenten in Form einer Nachricht mit. Dies ist aus dem Grunde wichtig, da die Umwelt dafür verantwortlich ist zu prüfen, ob eine Aktion eines Agenten ausführbar ist oder nicht. Das Kommunikationselement ist als ein cyclic behaviour implementiert, der zu jedem Zeitpunkt abfragt ob neue Nachrichten eingetroffen sind.

Das Element *Sende_Reward* ist ebenfalls ein eigenes behaviour. Es berechnet den reward für einzelne Agenten, den diese für ihre aktuelle Situation erhalten würden, und teilt dem Agenten diesen in Form eines Double Objektes mit. Dieses Objekt ist ein Simple Behaviour der einmal ausgeführt wird.

Das Element *Sende_Situation* funktioniert genau wie das Element *Sende_Reward*. Es sendet ein Objekt von der abgeleiteten Klasse A_Situation an den Agenten.

Um dem Agenten mitzuteilen zu welcher Episode eine Situation und ein reward gehören gibt es noch das Element *Sende_Episode*. Es ist auf die selbe Art und Weise implementiert, wie die beiden Elemente Sende_Reward und Sende_Situation.

6.11 Methode zur Bewertung aller Situationen

Die Methode zur Bewertung aller Situationen was in den vorigen Kapitel als Kritikelement bezeichnet wurde ist dafür verantwortlich, dem Agenten mitzuteilen wie gut oder schlecht eine Situation ist. Diese Methode darf sich nicht im Agenten selber befinden, da die Kritik extern erfolgen muss. Dies ist aus dem Grunde wichtig, da die Wahrnehmung des Agenten alleine eventuell nicht ausreicht, um die Situation zu bewerten. Gegen Anfang dieses Kapitels

wurde bereits erklärt, dass der Agent nicht zwangsläufig ein komplettes Bild seiner Umwelt besitzen muss um auch eine teilweise beobachtbare Umwelt zu gewährleisten. Was das Kritikelement lediglich benötigt, ist der Name des Agenten worüber die Umwelt den Reward berechnet und zum Agenten sendet.

public abstract Double berechne_Reward(String agent)

In diesem Framework ist das Kritikelement eine abstrakte Methode, die in einer konkreten Unterklasse überschrieben werden muss. Diese Methode wird kurz bevor die Umwelt den reward sendet aufgerufen.

6.12 Element für verbotene Aktionen

Oftmals ist es für einen Agenten nicht möglich eine Aktion auszuführen. Steht ein Agent in einer Grid World beispielsweise vor einem Hindernis, kann dieser das Feld nicht erreichen. Die Umwelt ist aus dem selben Grund wie beim Kritikelement dafür verantwortlich, dieses zu beurteilen. Der Grund ist das der Agent kein komplettes Modell seiner Umwelt intern selber verwaltet. In der konkreten Unterklasse der Umwelt gibt es eine abstrakte Methode die in einer konkreten Umwelt implementiert werden muss:

public boolean aktion_Moeglich(A_Aktion aktion_tmp, String agent)

Bei jeder Aktion, die die Umwelt von einem Agenten empfängt, wird geprüft ob diese Aktion wirklich möglich ist. Sollte eine Aktion nicht möglich sein, sendet die Umwelt dem Agenten eine Nachricht, worauf dieser eine neue Aktion berechnen muss.

6.13 Mittel zur Synchronisierung

Da im Vorfeld bereits festgelegt wurde, dass die Umwelt in der Art eines Agenten implementiert wird, bedeutetet dies, dass das gesamte Verhalten der Umwelt parallel ablaufen kann. Um die Synchronisierung einfach zu halten, muss verhindert werden das bestimmte Aktionen der Umwelt parallel ausgeführt werden. Um dies sicherzustellen reicht es, wenn die Umwelt ein Sequential Behaviour benutzt. Dieses Behaviour arbeitet alle Aufträge nacheinander ab. Wenn sich nun 2 Agenten in einer Umwelt befinden, ist so sehr leicht möglich die korrekte Synchronisierung sicherzustellen. Das ist darauf zurückzuführen, dass die einzelnen Behaviours sich nach ihrer Ausführung beenden und somit die nächste Aktion ausgeführt werden kann.

6.14 Aufbau des Agenten

Als generelle Oberklasse des Agenten dient die Klasse Agent_Frame. Sie besitzt die folgenden Objekte, die der Klasse im Konstruktor übergeben werden müssen. Diese Argumente sind angelehnt an die Definition des lernenden Agenten aus Abschnitt 3.5 laut [WJ95]. Dadurch, dass diese Elemente bei jedem Start neu initialisiert werden müssen, sind sie auch jedes mal austauschbar. Kurz erklärt werden hier die folgenden Elemente:

- das Kommunikationselement
- Leistungselement
- Lernelement
- Problemgenerator

6.15 Das Kommunikationselement

Aufgrund dessen das im vorherigen Abschnitt festgelegt wurde, dass das Element zur Bewertung der Situation sich in der Umwelt befindet, musste der Entwurf des lernenden Agenten geringfügig modifiziert werden. Der Agent empfängt die Kritik von der Umwelt und hat nun kein eigenes Element mehr was diese berechnet. Der Agent muss also zwischen mehreren Typen von Nachrichten unterscheiden können die ihm gesendet werden. Einmal dem reward, der neuen Situation, sowie der Mitteilung zu welcher Episode die vorherigen Objekte gehören. Ebenfalls gibt es noch die Möglichkeit, dass die Umwelt wie oben beschrieben gewisse Aktionen nicht zulässt. Also muss der Agent auch Nachrichten in Form von Boolean Objekten verstehen. Das Kommunikationselement ist dafür verantwortlich zu unterscheiden um was für ein Objekt es sich bei einer ankommenden Nachricht handelt. Diese Klasse ist ein Cyclic Behaviour, der durchgehend abfragt, ob eine neue Nachricht, den Agenten erreicht. Hat der Agent nun reward, Episode, sowie Situation erhalten ruft dieser das Lernelement auf.

6.16 Das Lernelement

Durch die Trennung zwischen Leistungs- und Lernelement wird der interne Algorithmus gekapselt. Da das Leistungselement immer dasselbe bleibt, kann das Lernelement einfach

ausgetauscht werden. Somit kann der interne Algorithmus zur Berechnung einfach ausgetauscht werden. Um einen einzelnen Algorithmus zu implementieren, muss die abstrakte Klasse Lernelement überschrieben werden. Nach dem Aufruf des Lernelementes aktualisiert dieses das Leistungselement und ruft daraufhin den Problemgenerator auf. Im Vorfeld sind die folgenden Objekte als Lernelement implementiert:

- Sarsa-On-Policy,
- Sarsa-Off-Policy,
- Sarsa-(λ).

Diese Algorithmen wurden im Abschnitt 5.1.5 und 5.1.6 ausgewählt.

6.17 Das Leistungselement

Da im Kapitel 4 bereits die dynamische Programmierung ausgeschlossen wurde reicht es aus nur ein Leistungselement zu implementieren. Denn dieses Leistungselement ist sowohl für die Monte Carlo Methoden wie auch für die Methoden des Temporal Differnce Learning tauglich. Deshalb ist das Leistungselement auch keine abstrakte Klasse. Um die Werte zu verwalten besitzt das Leistungselement eine Hash Map, die als Tabelle fungiert. Ebenfalls benötigt es eine Methode um abzufragen, welche Aktion das Leistunsgelement nun ausführen möchte. Im Normalfall führt das Leistungselement immer die Aktion aus, die am erfolgversprechendsten ist. Allerdings muss es eine weitere Schnittstelle geben, die vom Problemgenerator angesprochen wird. Dieser muss die Möglichkeit haben, die bisher intern ausgewählte Aktion noch einmal zu verändern. Für die interne Verarbeitung ist es nötig, dass das Leistungselement mehrere Objekte verwaltet. Diese sind in der Java Notation festgehalten das dieses Framework ebenfalls nur auf Java basiert:

- **ArrayList<A_Situation> bekannte_Situationen** um eine Liste darüber zu verwalten welche Situationen dem Agenten bereits bekannt sind.
- **ArrayList<A_Situation_Aktion> histroy** Um eine interne History zur verwalten. Dies ist für Eligibility-Traces nötig. Ebenfalls würde diese History gebraucht werden wenn dieses Framework um die Monte Carlo Methoden erweitert werden soll.
- **HashMap<A_Situation_Aktion, Double> werte** In dieser Hash Map werden die Q-Werte abgespeichert.

- **HashMap<A_Situation_Aktion, Double> e_Werte** Da bereits festgelegt wurde das Eligibility-Traces in diesem Framework implementiert werden, reicht es nicht aus jedes Situations-Aktions-Paar mit einem Q-Wert zu versehen. Es ist ebenfalls nötig für jedes dieser Paare einen Eligibility-Trace-Wert zu verwalten.
- **HashMap<A_Situation_Aktion, Boolean> verbotene_Aktionen** Der Agent muss sich merken, ob bestimmte Aktionen in bestimmten Situationen ausscheiden, damit dieser die Aktionen nicht mehr auswählen darf.

6.18 Der Problemgenerator

Der Problemgenerator muss die Möglichkeit besitzen, zu wählen ob die beste Aktion gewählt werden soll oder eine zufällige Aktion ausprobiert werden soll. Der Problemgenerator verändert dafür die im Leitungselement intern gesetzte Variable *ausgewaehlte_Aktion*. Als einziges implementiertes Objekt für den Problemgenrator ist die Klasse

- *e_greedy*

vorhanden. Im Abschnitt 5.1.7 wurde dieser Algorithmus ausgewählt. Trotz allem existiert eine abstrakte Klasse Problemgenrator. Wenn im nachhinein noch weitere Explorationsverfahren implementiert werden sollten, müsste von der abstrakten Klasse Problemgenerator lediglich die Methode
public void start_Generator(Leistungselement leistungselement) überschrieben werden.

6.19 Übergabe

Die Klasse Uebergabe wird dazu genutzt, dass die Umwelt, sowie alle Agenten mit den selben Werte arbeiten. Hier werden als Objekte definiert:

- Die mögliche Aktionen.
- Die Namen der Agenten.
- Und alle Agenten werden in einer Hash Map mit einer Initalsituation ausgestattet.

Somit ist sichergestellt das Umwelt, wie auch Agent auf den selben Daten arbeiten und somit bsp. der Agent, wie auch die Umwelt beide wissen welche Aktionen in dieser Anwendung existieren.

6.20 GUI

Da die Visualisierung ein wichtiger Aspekt, ist um den Verlauf des Lernens zu verfolgen, gibt es erstens für jeden Agenten eine eigene GUI um die Lernerfolge zu verfolgen und zweitens noch eine Schnittstelle für die Umwelt. Für die GUI eines Agenten sind am wichtigsten die Situations-Aktionspaare mit ihren dazu gehörigen Werten. Der Vorteil hierbei ist sicherlich das sich bei dieser Anzeige lediglich die Beschreibungen der Situationen und der Aktionen ändern. Die Struktur bleibt allerdings immer die selbe. Man findet eine Tabelle mit Situations-Aktionspaaren und Werten vor. Um es dem Benutzer an dieser Stelle einfach zu machen, wird hier einfach die toString Methode der einzelnen Situationen, sowie Aktionen ausgegeben. So entsteht für jeden Agenten eine einzelne GUI. Über das Comparable Interface kann der Benutzer dann festlegen in welcher Reihenfolge die Werte sortiert werden sollen.

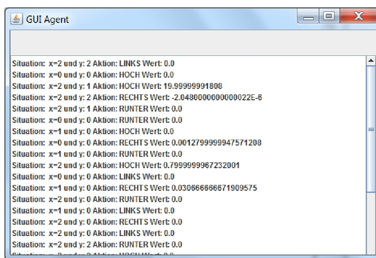

Abbildung 6.3: Die GUI eines Agenten

Ein generelle Schnittstelle für eine GUI für die Umwelt wird in diesem Framework nicht vorgegeben. Die Szenarien unterscheiden sich an bestimmte Stellen einfach zu stark. Deshalb ist es an dieser Stelle einfacher, dem Entwickler hier freie Hand zu lassen.

7 Beispielanwendungen

Zu diesem Framework sollen ebenfalls 3 Beispielanwendungen implementiert werden. Diese werden in diesem Abschnitt kurz beschrieben.

7.1 Grid World

Als erste Beispielawendung in dieser Arbeit soll das Beispiel der Grid World implementiert werden. Dieses Beispiel wurde bereits in Kapitel 2.2 erklärt. Der Agent startet hier bei den Koordinaten x=0 und y=0. Um auch noch ein Beispiel für Aktionen einzubinden die nicht möglich sind, befindet sich bei den Koordinaten x=1 und y=1 ein Berg. Die Koordinaten kann der Agent zu keinem Zeitpunkt besuchen.

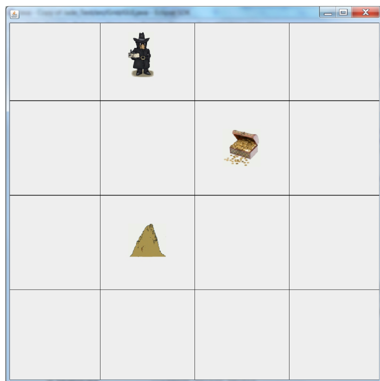

Abbildung 7.1: Die Grid World

Der Agent lernt nach einiger Zeit den optimalen Weg zum Schatz. Dieses Beispiel ist zwar recht trivial, dennoch ist diese Anwendung gut dazu geeignet die Algorithmen nachzuvollziehen und auch neue Algorithmen zu testen. Den man kann an diesem Beispiel leicht die einzelnen Berechnungen nachvollziehen. Auch die Konstruktion eigener Anwendungen wird durch dieses Beispiel leicht verständlich.

7.2 Tic Tac Toe

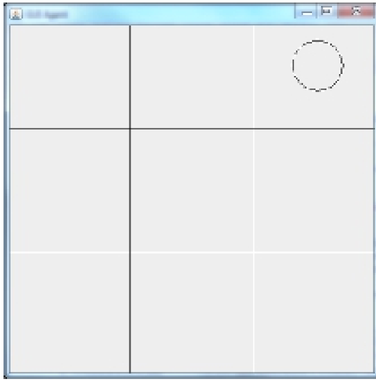

Abbildung 7.2: Beispielanwendung des Tic-Tac-Toe Spiels

Als zweite Anwendung die realisiert werden soll, wurde im Vorfeld ein Tic-Tac-Toe-Spiel festgelegt. Diese Anwendung ist zwar kaum komplexer als das Beispiel der Grid World dennoch ist es von großer Wichtigkeit, das damit gezeigt wurde das auch ein Szenario mit mehr als einem Agent realisiert werden kann. Bei diesem Spiel bekommt der erste Agent die Kreise zugewiesen und der zweite Agent die Kreuze. Gewonnen hat der Agent der es schafft 3 seiner Symbole waagerecht, senkrecht oder quer hintereinander zu postieren. Es wird jede Spielrunde gewechselt. Als erstes darf Agent 1 einen Kreis setzen danach der zweite Agent ein Kreuz.

7.3 Die Siedler

Als letzte und komplexeste Anwendung wird noch ein kleines Siedlerspiel als Beispielanwendung für diese Arbeit implementiert. Das Spiel basiert darauf das jeder Agent verschiedene Gebäude und Ressourcen besitzt. Die Ressourcen sind nötig um die Gebäude zu bauen die die Ressourcen erwirtschaften. Die Situation jedes Agenten besteht aus der Anzahl der vorhandenen Gebäude und der vorhandenen Ressourcen. Die einzige Aktion die die Agenten ausführen können ist die Aktion “Baue“. Die Agenten haben das Spiel am Ende gewonnen wenn diese eine bestimmte Anzahl Gold oder Eisen erwirtschaftet haben. Die Agenten lernen nach mehreren Spielen wie sie am schnellsten eine der beiden Ressourcen erwirtschaften. Alternativ wäre es sehr einfach hier die Siegbedingungen zu verändern, indem der Entwickler die Methode berechne_Reward() modifizieren würde. Ebenfalls wäre das Spiel sehr leicht erweiterbar. Beispielsweise wäre es möglich Soldaten einzuführen. Hierzu müsste lediglich die Situation leicht modifiziert werden und eine neue Aktion eingefügt werden.

Festzuhalten bleibt das diese Beispielanwendung sehr leicht umgesetzt werden konnte. Es war lediglich nötig die Situationen, sowie die Aktionen zu modellieren und die Bedingungen für den Sieg sowie die möglichen Aktionen zu definieren.

8 Zusammenfassung

Hier sollen noch einmal alle Erkenntnisse in Kurzform betrachtet werden und inwieweit es gelungen ist die Ziele zu erreichen. Ebenfalls soll ein kleiner Ausblick gegeben werden in welchen Punkten noch Erweiterungsbedarf bestehen könnte.

8.1 Reinforcement Learning

Im zweiten Kapitel dieser Arbeit wurden die einzelnen Grundlagen des Reinforcement Learning erklärt. Daraufhin wurden in Kurzform die einzelnen Algorithmen vorgestellt. Zum Abschluss dieses Kapitels wurden die Eligibility-Traces noch einmal aufgegriffen.

8.2 Agentensysteme

Im Kapitel 3 wurde in Kurzform erklärt was genau einen Software-Agenten charakterisiert. Daraufhin wurden die Merkmale der verschiedenen Arbeitsumgebungen eines Agenten beschrieben. Einzelne Agentenarchitekturen u.a. die eines lernenden Agenten von Stuart Russell und Peter Norvig wurden daraufhin erläutert. Am Ende dieses Kapitels wurde in knapper Form erwähnt wieso die Wahl des gewählten Agentensystems auf Jade gefallen ist.

8.3 Generische Aspekte

In diesem Analyse Kapitel wurde ein Bezug hergestellt zu der Architektur eines lernenden Agenten von Stuart Russell und Peter Norvig. Die Priorität in diesem Kapitel lag darauf, das Leistungselement sowie das Lernelement auf die einzelnen Algorithmen eines Reinforcement Learning Agenten abzubilden. Hierbei wurde schnell ersichtlich wie komplex die Datenstruktur der dynamischen Programmierung ist. Ebenfalls wurde aufgezeigt, dass die Monte Carlo Methoden und das Temporal Difference Learning auf denselben Daten arbeiten und deshalb dasselbe Leistungselement benötigen. Was zu der Erkenntnis führte, das zum

Austausch des Lernalgorithmus lediglich das Lernelement ausgetauscht werden muss. Auch die Algorithmen zu Exploration wurden auf Gemeinsamkeiten untersucht. Auch hierbei war es möglich einzelne Algorithmen zu kapseln.

8.4 Abgrenzung des Frameworks

In diesem Teil der Analyse wurden die einzelnen Algorithmen in Anbetracht der Erkenntnisse aus dem vorangegangenem Kapitel untersucht. Die dynamische Programmierung wurde an dieser Stelle aufgrund ihrer komplexen Datenstruktur und ihrer eher theoretischen Bedeutung ausgeschlossen. Die Monte Carlo Methoden ebenfalls. Allerdings wurde in diesem Kontext festgehalten, dass dieses Framework für die Monte Carlo Methoden leicht erweiterbar ist. An dieser Stelle fiel die Entscheidung für die Algorithmen, die im Vorfeld implementiert werden auf 3 Algorithmen aus dem Bereich des Temporal Difference Learning. Dies war darauf zurückzuführen, dass die Methoden des Temporal Difference Learning keinerlei Nachteile der Monte Carlo Methoden besitzen hingegen aber alle Vorteile. Als weiteren Punkt in dieser Analyse wurden die Eigenschaften eines lernenden Agenten laut [RN04] aufgegriffen und analysiert ob sich alle diese Eigenschaften miteinander vereinbaren lassen. Als Ergebnis konnte hierbei festgelegt werden das ein Agent in dieser Arbeit ausnahmslos alle diese Punkte erfüllen kann. Ebenfalls wurden die Merkmale aller Arbeitsumgebungen eines Agenten betrachtet und analysiert welche Merkmale sich davon in dieser Arbeit realisieren lassen. Hier ergab sich, dass die Umwelt in diesem Framework alle Merkmale bis auf stetige Zustandsräume erfüllen kann.

8.5 Bau der Beispielanwendungen

Der Bau der 3 Beispielanwendungen hat gezeigt wie leicht man mit diesem Framework die bestimmten Szenarien nachprogrammieren kann. Die Grid World wurde deshalb gewählt um mit dieser Anwendung die einzelnen Algorithmen leicht zu testen. Das Tic-Tac-Toe Spiel war hingegen die erste Anwendung die eine Multi-Agenten Umgebung war. Als letztes wurde ein Siedler Spiel realisiert, welches zeigen sollte das auch komplexere Anwendungen mit diesem Framework realisiert werden können.

8.6 Fazit

Als Fazit bleibt hier festzuhalten, das es gelungen ist die Architektur eines lernenden Agenten von Stuart Russell und Peter Norvig auf verschieden Reinforcement Learning Anwendungen

abzubilden. Allerdings wurden nicht alle Algorithmen implementiert und das Framework lässt in der gewählten Architektur keine dynamische Programmierung zu. Dennoch wurde dargelegt das die Architektur konform zu den Monte Carlo Methoden sowie den Methoden des Temporal Difference Learning ist. Es wurde ebenfalls dargestellt weshalb diese Methoden eine weitaus höhere Priorität haben als die dynamische Programmierung. Am Ende bleibt festzuhalten: Das Hauptziel, mittels des Aufbaus des Frameworks nach einem lernenden Agenten eine Architektur zu schaffen, die es erlaubt die einzelnen Algorithmen zu kapseln und auszutauschen, ist erfüllt worden.

8.7 Ausblick

Abschließend bleibt noch festzustellen, das sich in dieser Arbeit klar gezeigt hat, das es noch eine Menge an möglichen Erweiterungen gibt. Dies wäre einmal die Monte Carlo Methoden zu implementieren die bisher nicht realisiert sind. Auch noch möglich wäre es, weiter Algorithmen im Bereich Exploration zu implementieren. Als weiteren Aspekt, der dieses Framework qualitativ steigern würde, wäre ebenfalls noch darüber nachzudenken stetige Zustandsräume zu implementieren. Hier gäbe es verschiedene Möglichkeiten wie dies geschehen könnte. Alleine dieser Punkt würde sicherlich genug Stoff für eine weitere Arbeit dieser Art sicherstellen.

Literaturverzeichnis

[AOSE00] *WOOLDRIDGE, Michael: An Introduction to MultiAgent Systems. West Sussex, England : John Wiley Sons Ltd., 2002. ISBN 9780471496915*

[AwPO95] *AWIZEN, Michael ; Paulussen, Torsten O.: Modellierung der Kommunikationsprotokolle für agentenunterstützte Koordinationsverfahren. http://vsis-www.informatik.uni-hamburg.de/getDoc.php/publications/95/ECDPvA01.pdf*

[Bahlo07] *BAHLO, Tim: Untersuchung des Einsatzes von Multiagentensystemen für die Steuerung des Materialflusses in der innerbetrieblichen Logistik, Hamburg, Diplomica Verlag 2006 - ISBN 978-3-8366-5470-8*

[Lueke07] *Lüke, Olaf: Reinforcement Learning in hierarchischen Architekturen. http://homepages.uni-paderborn.de/olaf/dinge/bachelor/ausarbeitung.pdf*

[RLR96] *Wengerek, Thomas : Reinforcement-Lernen in der Robotik, Universität Bielefeld, 1996. ISBN 3-89601-119-7*

[GO03] *GÖRZ Günther: Handbuch der künstlichen Intelligenz, München, Oldenburg Wissenschaftsverlag GmbH, 2003. ISBN 3486´272128*

[KR08] *WERNO Sascha: Konzeption und Realisierung eines agentenbasierten Sensornetzwerks zur automatischen Zustandsermittlung von Geschäftsprozessen, GRIN Verlag, 2008. ISBN 978-3-638-02800-4*

[SB98] *SUTTON, R.S. ; Barto, A.G.: Reinforcement Learning: An Introduction. Cambridge, MA : MIT Press, 1998. ISBN 0-262-19398-1*

[Wolter07] *WOLTER, Anne: Reinforcement Learning in der Roboternavigation, Lehrstuhl Intelligente Systeme, Universität Duisburg/Essen, 2007. ISBN 978-3-639-04702-8*

[HA07] *Hans Alexander:Untersuchungen zur sicheren Exploration in Reinforcement Learning. http://ahans.de/publications/hans07diplomarbeit.pdf*

[WJ95] *WOLRIDGE, Michael; Jennings, Nick; Stuart ; Intelligent Agents: Theory and Practice; 1995*

[WE07] *Weiß Chathrin: JADE und FIPA. http://www.dfki.de/ kipp/seminar/ws0607/reports/Cathrin-Weiss-final.pdf*

[RN04] *RUSSEL, Stuart ; NORVIG, Peter: Künstliche Intelligenz: Ein moderner Ansatz. 2. München, Germany : Pearson Studium, 2004. ISBN 9783827370891*

[WESC] *WENDT, Oliver ; Schwind , Michael: Reinforcement Learning zur Lösung multidimensionaler Yield-Management-Probleme, Institut für Wirtschaftsinformatik der Universität Frankfurt. http://www.wi-frankfurt.de/publikationenNeu/ReinforcementLearningzurLoesung.pdf* bibitem[TO07]tokic *TOKIC , Michel :Optimierung des Explorationsverhaltens eines lernenden Laufroboters. Hoschule Ravensburg-Weingarten http://opus.bsz-bw.de/hsbwgt/volltexte/2008/46/pdf/projektarbeit.pdf*

Abbildungsverzeichnis

2.1 Agent Umwelt Interaktion . . . 11
2.2 Die Grid World . . . 11
2.3 Der Agent mit und ohne Eligibility-Traces . . . 23
2.4 Accumulating-Traces . . . 24
2.5 Accumulating Traces und Replacing-Eligibility-Traces im Vergleich . . . 26
2.6 Agent erst mehrmals falsche Aktion und erst danach die richtige . . . 26

3.1 Lernender Agent . . . 32

4.1 Leistungselement zur V-Wert-Implementation anhand der Temporal Difference Learning und der Monte Carlo Methoden . . . 36
4.2 After-State-Tabelle zur V-Wert-Implementation anhand der Temproal Difference Learning und der Monte Carlo Methoden . . . 36
4.3 Leistungselement zur Q-Wert-Implementation anhand der Temproal Difference Learning und der Monte Carlo Methoden . . . 37
4.4 Leistungselement der dynamischen Programmierung . . . 38

6.1 Die Klasse A_Situation . . . 54
6.2 Die Klasse A_Aktion . . . 55
6.3 Die GUI eines Agenten . . . 61

7.1 Die Grid World . . . 62
7.2 Beispielanwendung des Tic-Tac-Toe Spiels . . . 63

Algorithmenverzeichnis

1 Policy Evaluation . . . 16
2 Policy Iteration . . . 17
3 First-visit Methode . . . 19
4 Sarsa-On-Policy . . . 21
5 Sarsa-Off-Policy . . . 22
6 TD(λ) . . . 24
7 Sarsa(λ) . . . 25

Versicherung über Selbstständigkeit

Hiermit versichere ich, dass ich die vorliegende Arbeit im Sinne der Prüfungsordnung nach §24(5) ohne fremde Hilfe selbstständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 27. März 2011

Ort, Datum

Unterschrift