Jan-Uriel Lorbeer

Entwicklung von Reinforcement-Learning-Agenten für Strategie-Gesellschaftsspiele

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung

im Studiengang Technische 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. Zhen Ru Dai

Abgegeben am 09.11.2012

Jan-Uriel Lorbeer

Thema der Bachelorthesis
Entwicklung von Reinforcement-Learning-Agenten für Strategie-Gesellschaftsspiele

Stichworte
Künstliche Intelligenz, Maschinelles Lernen, Bestärkendes Lernen

Kurzzusammenfassung
Es wird die Entwicklung von reinforcement learning Agenten untersucht. Dies geschieht am Beispiel eines Strategie-Gesellschaftsspiels. Es wird die Programmiersprache C++ und das Qt-Framework verwendet. Dabei werden insbesondere Konzeptionierung und Realisierung von Agenten mit den reinforcement learning Algorithmen Monte Carlo, Q-Learning, Sarsa und Sarsa(λ) untersucht. In einem abschließenden Vergleich werden die Eignung und die Effizienz der verschiedenen Algorithmen für das gewählte Beispiel gegenüber gestellt. Hierbei zeigt sich, dass die komplexeren Aglorithmen der TD-Aglorithmen die besten Ergebnisse erzielen.

Jan-Uriel Lorbeer

Title of the paper
Entwicklung von Reinforcement-Learning-Agenten für Strategie-Gesellschaftsspiele

Keywords
artifical intelligence, machine learning, reinforcement learning

Abstract
The development of reinforcement learning agents is under investigation. For this the example of a strategy parlor game is used. The programming language C++ and the Qt framework is applied. In particular, conception and realization of agents with the reinforcement learning algorithms Monte Carlo, Q-learning, Sarsa and Sarsa(λ) are investigated. In a final comparison the suitability and the efficiency of the different algorithms are being compared. The more complex algorithms among the TD algorithms achieved the best results.

Inhaltsverzeichnis

1 Einleitung **1**
1.1 Aufbau der Arbeit . . . 2

2 Grundlagen **3**
2.1 Strategie-Gesellschaftsspiele . . . 3
2.1.1 Merkmale . . . 3
2.2 Verwendete Spielumgebung . . . 4
2.2.1 Spielfeld . . . 4
2.2.2 Spielregeln . . . 5
2.3 Maschinelles Lernen . . . 7
2.4 Reinforcement Learning . . . 7
2.4.1 Markov-Entscheidungsprozess MDP . . . 8
2.4.2 Elemente des Reinforcement Learning . . . 9
2.4.3 Funktionsweise des Reinforcement Learning . . . 10
2.4.4 Reinforcement Learning Algorithmen . . . 11

3 Analyse **15**
3.1 Spielanalyse . . . 15
3.2 Zustandsraumanalyse . . . 15
3.3 Modellierung der Zustände . . . 17
3.4 MDP Analyse . . . 18

4 Entwicklung **19**
4.1 Konzept . . . 19
4.1.1 Umwelt . . . 20
4.1.2 Agent . . . 21
4.1.3 Kommunikation . . . 22
4.1.4 Konzeptüberblick . . . 24
4.1.5 Agenten mit Umweltmodell . . . 24
4.2 Realisierung . . . 28

4.2.1 Allgemeine Realisierungsaspekte 28
4.2.2 Implementierung der Umwelt 28
4.2.2.1 Erläuterung der übernommenen Elemente 29
4.2.2.2 Umsetzung des Spielleiters 30
4.2.2.3 GUI der Umwelt 32
4.2.3 Implementierung der Agenten 33
4.2.3.1 Allgemeiner Aufbau der Agenten 33
4.2.3.2 Umsetzung der Reinforcment Learing Algorithmen . . 35
4.2.3.3 GUI der Agenten 40

5 Auswertung **42**
5.1 Implementationsaufwand 42
5.2 Lernverhalten 43
5.3 Spielerfolg 45
5.4 Auswertende Betrachtung 46

6 Fazit und Ausblick **48**

Literaturverzeichnis **50**

A Software Elemente **51**

B Benutzte Hilfsmittel **62**

C Übersicht über Inhalt der CD **63**

Abbildungsverzeichnis

2.1 Spielfeld nur mit festen Quadraten 4
2.2 Komplettes Spielfeld ohne Spielfiguren und Ziele 5
2.3 Komplettes Spiel mit Aktionsmöglichkeiten 6
2.4 Agent Umwelt Schema 8
2.5 Gridworld: Vergleich des Lernerfolg von Sarsa zu Sarsa(λ) 14

4.1 Agent Umwelt Schema 19
4.2 Komponentendiagramm der Umweltsoftware 20
4.3 Komponentendiagramm der Agentensoftware 22
4.4 Sequenzdiagramm der Kommunikation mit 2 Agenten 23
4.5 Komponentendiagramm Gesamtprojekt 24
4.6 Komponentendiagramm Agent mit Modell 25
4.7 Gridworld Dyna-Q Agent 26
4.8 Klassendiagramm: Umwelt 29
4.9 Aktivitätsdiagramm des Gamemasters 30
4.10 Aktivitätsdiagramm der Episodenaktivität 31
4.11 GUI der Umwelt 32
4.12 Klassendiagramm: Agenten 33
4.13 Aktivitätsdiagramm der Agentenhauptroutine 34
4.14 GUI der Agenten 40

5.1 Lernverhalten einzeln spielender Agenten 43
5.2 Speicherverbrauchs- (links) und Ausführungsgeschwindigkeitstest (rechts) 44
5.3 Mehrspieler-Test gegen Zufallsagenten 45
5.4 Mehrspieler-Test jeder gegen jeden 46

A.1 Code: Gekürzter run loop der Gamemaster Klasse 55
A.2 Code: Episodenablauf der Gamemaster Klasse 56
A.3 Code: run loop der Agenten 57
A.4 Code: Greedy-Suche 58
A.5 Code: ε-Greedy-Suche 59

Abkürzungen

ε-Greedy-Suche ... Version der Greedy-Suche, bei der auch zufällige Aktionen ausgewählt werden
λ ... Nachlassrate bei spurlernenden Algorithmen
α ... Lernfaktor, der bei TD-Algorithmen bestimmt, wie stark neue Erfahrungen sich auswirken
ε ... die Wahrscheinlichkeit mit der bei der ε-Greedy-Suche eine zufällige Aktion gewählt wird
γ ... Skontorate mit der TD-Algorithmen zukünftige Belohnungen abzinsen
Agent ... Computerprogramm, das zu autonomen Verhalten fähig ist
Belohnung ... Von der Umwelt wiedergegebene Bewertung
CPU ... Central Processing Unit
DDR3 ... Double Data Rate - DDR-Arbeitsspeicher in der 3. Generation
distribution Modell Wahrscheinlichkeiten abbildendes Modell
Framework ... Ein Programmiergerüst, hauptsächlich bei objektorientierter Softwareentwicklung
Greedy-Suche ... Aktionsauswahl Methode, bei der nur die am besten bewerteten Züge gewählt werden
Gridworld ... Eine in quadratischen Feldern angeordnete Testumwelt
GUI ... Graphical User Interface ; grafische Benutzeroberfläche
IPC ... inter-process communication
MDP ... Markov decision process
Modell ... Agenten interne Abbildung der Umwelt
Q-Wert ... Wert eines Zustandsaktionspaares
Qt-Framework ... Ein auf den Qt-Bibliotheken basierendes Framework
RAM ... Random-Access Memory/ Direktzugriffsspeicher, Arbeitsspeicher
RL ... reinforcement learning
sample Modell ... deterministisches Modell
SSD-Festplatten .. Solid-State-Drive Festplatte
Strategie ... Bestimmt das Verhalten von RL-Agenten
Umwelt ... Umgebung in der ein Agent agiert

Kapitel 1

Einleitung

Mit Anbruch des Informationszeitalters wuchsen die Möglichkeiten deutlich über die Erzeugung einfacher Maschinen zur Unterstützung manueller Arbeit hinaus. So war es nur eine Frage der Zeit, bis Hilfsmitteln geschaffen wurden, die bei der Erfüllung einer Aufgabe nicht nur unterstützen, sondern diese unter Zuhilfenahme von Informationstechniken erledigen.

Dabei ist ein Programm für eine solche Aufgabe ein erster Schritt. Erledigt dieses Programm seine Aufgabe eigenständig, so wird es zum sogenannten Agenten([LC01] S. 19). Dies ist eine sehr offene Definition, da eine engere Definition in der Fachwelt, selbst für einen einzeln arbeitenden Agenten, nicht konsensfähig scheint (vgl. [SLB09] S. VIII).

Ein Agent soll also eigenständig handeln, was eine gewisse Art von Intelligenz bedingt. Da die zu erfüllenden Aufgaben komplex sein können und ein Programmdesigner nicht alle Möglichkeiten eines Agenten im Vorhinein festlegen kann und die Möglichkeiten eventuell nicht einmal bekannt sind, bedeutet dies: Ein Agent muss eigenständig hinzulernen. Man spricht hierbei auch vom „Maschinellen Lernen“.

Das Feld des Maschinellen Lernens wird unterteilt in Überwachtes Lernen (supervised learning), Unüberwachtes Lernen (unsupervised learning) und Bestärkendes Lernen (reinforcement learning). Im Verlauf dieser Arbeit wird Reinforcement Learning im Vordergrund stehen. Auf die anderen Lernverfahren wird nur am Rande eingegangen.

Der Begriff Reinforcement Learning ist nicht neu, sondern stammt aus den frühen Tagen von Kybernetik und der Arbeit in Statistik, Psychologie, Neurowissenschaften und Informatik.(vgl. [KLM96] S. 1).

Bei Reinforcement Learning versucht der Agent sich selbst mit Hilfe von Aktionen und Sensoren das richtige Verhalten beizubringen. Allgemein ist Reinforcement Learning jedoch nicht gekennzeichnet durch Lernmethoden, also Art der Aneignung von Wissen, sondern durch die Charakterisierung von Lernproblemen. (vgl. [SB05] S. 15).

Diese Art von Lernverfahren findet auch Anwendung in Programmen für intelligenten Regelungsmaschinen oder in ersten Ansätzen von intelligenten Produktionsanlagen.

Ein Agent arbeitet immer auf ein Ziel zu, welches je nach Kontext sein könnte: Gewinne das Spiel, fahre das Auto so schnell wie möglich oder finde die beste Route und so weiter. (vgl. [CFLM08] S. 203).

Agenten kommen hierbei in einem äußeren Rahmen, hier Umwelt genannt, zum Einsatz. Für die Entwicklung solcher Agenten wird im Verlauf dieser Arbeit das Beispiel von Spielern in einem Strategie-Gesellschaftsspiel herangezogen.

Diese Arbeit baut auf einer früheren Projektarbeit auf, die an der HAW-Hamburg vom Verfasser und Johannes Reidle durchgeführt wurde. Bei dem Projekt wurde ein ähnliches Spiel entwickelt und erste Ansätze für lernende Agenten geschaffen. In dieser Arbeit werden wesentliche Aspekte der Projektarbeit erneuter und die Ansätze in einem erweiterten Rahmen verfolgt. Dabei sollen alle nötigen Schritte absolviert werden, damit der Agent das Spielen erfolgreich lernen kann. Dies beinhaltet auch die Analyse der Umwelt, Bestimmung deren Schnittstellen zum Agenten als auch die Auswahl und Implementierung von mehreren Reinforcement Learning Algorithmen. Das Ziel ist, möglichst erfolgreich spielende Agenten zu schaffen, und deren Verhalten miteinander zu vergleichen. Im Vergleich stehen der Spielerfolg, die Lerngeschwindigkeit und der Ressourcenverbrauch der Agenten.

1.1 Aufbau der Arbeit

In Kapitel 2 werden zunächst die Grundlagen des bearbeiteten Themas dargelegt. Dazu zählen sowohl Erklärungen über Strategie-Gesellschaftsspiele und deren Eigenschaften als auch die Grundlagen des Maschinellen Lernens. Insbesondere wird auf das reinforcement learning und seiner Algorithmen eingegangen.

Das Spiel wird in Kapitel 3 in Hinsicht auf reinforcement learning analysiert. Speziell die Spielzustände, deren Anzahl und wie sich diese modellieren lassen wird behandelt.

Kapitel 4 beschäftigt sich zunächst mit der Kozeptionierung der Agenten, der Umwelt und der Verknüpfung der beiden. Anschließend wird auf den Aufbau der Software und deren Umsetzung eingegangen. Insbesondere wird hierbei die Implementation der reinforcement learning Algorithmen gezeigt.

Im vorletzten Kapitel 5 werden die Ergebnisse aus Testes mit den entwickelten Agenten und die Beschaffenheit dieser Tests erläutert. Das Verhalten der Agenten wird dabei verglichen und bewertet.

Im letzten Kapitel 6 wird die Arbeit zusammen gefasst und abschließende Folgerungen gezogen.

Kapitel 2

Grundlagen

2.1 Strategie-Gesellschaftsspiele

Strategie- und Gesellschaftsspiele bezeichnen jeweils eine Gruppe von Spielen.

Als *Strategiespiele* werden für gewöhnlich Spiele bezeichnet, die sich durch taktisches Vorgehen oder das Planen von Strategien und deren Durchführung auszeichnen. Langfristige Strategien erstrecken sich meist über einen größeren Zeitraum bzw. mehrere Züge bis hin zum gesamten Spiel, während taktische Entscheidungen oft situationsabhängig sind und sich auf den derzeitigen Zug und gegebenenfalls die nahe Zukunft beschränken. In Strategiespielen treten Spieler vorwiegend im Wettstreit oder direkt gegeneinander an, um ein bestimmtes Ziel zu erreichen.

Gesellschaftsspiele beschreiben ein weites Feld an Spielen. Allen Spielen dieses Typs ist jedoch gemein, dass sich mindestens zwei, für gewöhnlich jedoch mehr, Spieler daran beteiligen und miteinander direkt oder indirekt interagieren.

Strategie-Gesellschaftsspiele sind alle jene Spiele, die sowohl in die Gruppe der Strategiespiele als auch in die der Gesellschaftsspiele fallen.

2.1.1 Merkmale

Allen Strategie-Gesellschaftsspielen ist gemein, dass sie über *Spielregeln* verfügen, die einen festen Anfang und ein eindeutiges Ende haben. Das Ende wird dabei meist durch das Erreichen des Spielziels durch einen Spieler herbeigeführt. Somit ist ein *episodisches Spielen* eines der Hauptmerkmale dieser Spielart. Die Spielregeln setzten des Weiteren den Ablauf dieser Episoden fest.

Ein weiterer Aspekt bei Strategie-Gesellschaftsspielen ist, dass ein Spielzug sich aus mehreren *Aktionen* pro Zug zusammensetzten kann, wie z.B. bei dem Spiel, die Siedler von Catan (TM), bei dem zunächst Ressourcen aufgenommen, anschließend Handel betrieben und als letztes Gebäude errichtet werden (vgl. [Teu95]). Charakteristisch für Strategiespiele ist dabei, dass eine solche Aktion meist keine oder höchstens eine geringe Zufallskomponente enthält und so die direkten Konsequenzen absehbar sind. Aktionen verändern das Spiel innerhalb der erlaubten Spielregeln. Durch die Teilnahme mehrerer Spieler, die alle Einfluss auf das Spiel nehmen, wird selbst ein Spiel ohne Zufallskomponenten für den einzelnen Spieler nicht mehr vollständig vorhersehbar. Aus einem gewählten Zug eines Spielers, in einem bestimmten Zustand des Spiels, folgt nicht

immer der erwartete Zustand zum Beginn seines nächsten Zuges, wodurch eine nicht deterministische Spielersicht zustande kommt.

Viele Strategie-Gesellschaftsspiele verfügen des Weiteren über einen Mechanismus um verschiedene Startzustände zu erzeugen. Dergleichen kann zu Beginn des Spiels über eine Zufallsverteilung durch Karten oder Spielsteine erfolgen.

2.2 Verwendete Spielumgebung

Um die charakteristischen Merkmale in einem geeigneten Maße umzusetzen wird ein fiktives Brettspiel betrachtet. Hierbei handelt es sich um einen Irrgarten, in dem jeder Spieler versucht, durch Verändern des Irrgartens und Bewegen seiner Spielfigur alle seine zugeteilten Ziele als Erstes zu erreichen.

Das Irrgartenspiel ist episodisch und es werden mehrere Aktionen pro Zug ausgeführt. Eine Aktion hat keine Zufallskomponente, die Spielerzahl beträgt 2-4 Spieler und es weist eine Vielzahl von Startzuständen des Spielfeldes auf.

2.2.1 Spielfeld

Das Spielfeld besteht aus quadratischen Irrgartenteilen, die in einem 5x5 Raster angeordnet sind. Sowohl die Eck- und die mittleren Seitenquadrate als auch das Zentralquadrat sind fest und stellen 90°-Kurven, Geraden und eine Kreuzung dar (siehe Abbildung 2.1).

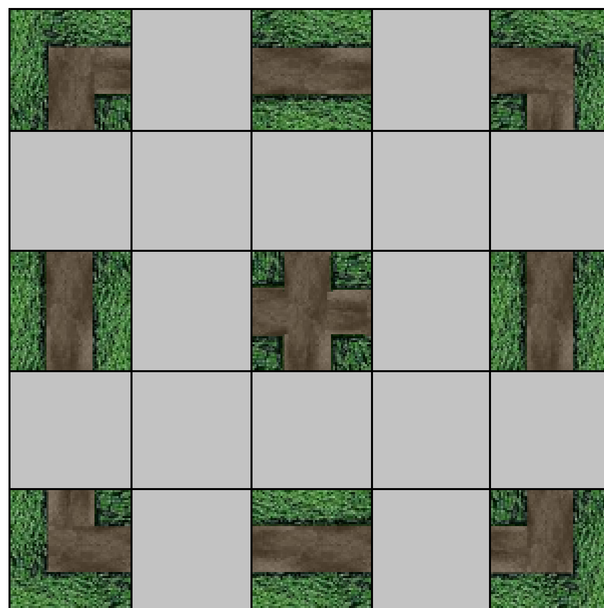

Abbildung 2.1: Spielfeld nur mit festen Quadraten

Die restlichen Quadrate des 5x5 Musters werden durch 90°-Kurven und Geraden gefüllt.

Ergänzt wird das Spielfeld durch ein 26. Quadrat, das im Verlauf des Spieles, eingeschoben wird, um den Irrgarten zu verändern (siehe Abbildung 2.2).

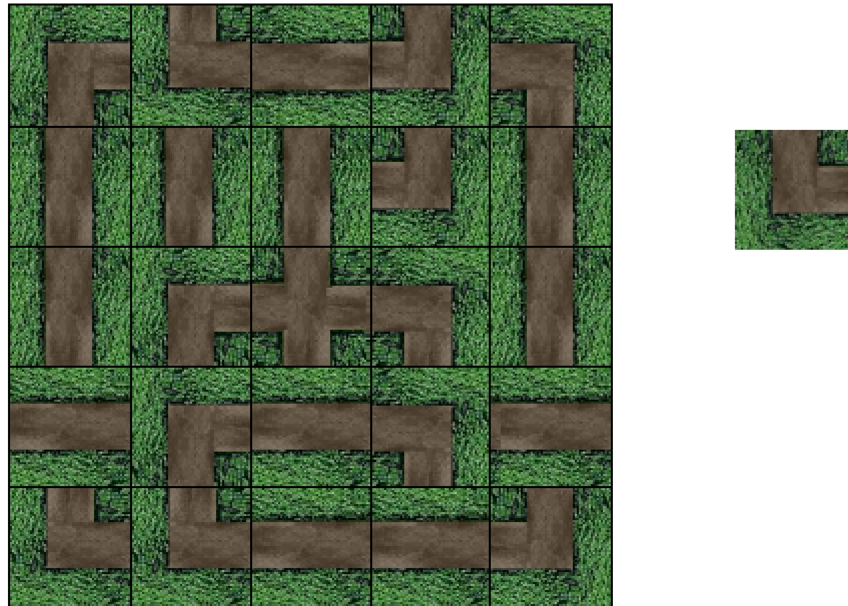

Abbildung 2.2: Komplettes Spielfeld ohne Spielfiguren und Ziele

Sowohl die Ecken als auch einige der beweglichen Quadrate weisen Zielmarkierungen auf, die zum Erreichen des Spielziels nötig sind.

2.2.2 Spielregeln

Spielerzahl: 2-4

Spielbeginn

Zu Beginn werden alle leeren Flächen mit beweglichen Irrgartenquadraten in zufälliger Ausrichtung und Reihenfolge aufgefüllt. Anschließend werden alle teilnehmenden Spielfiguren auf den mittleren Seitenquadraten platziert, wobei jede Spielfigur ein anderes Quadrat belegt. Jedem Spieler wird nun eine Folge von bis zu drei zufällig gewählten Zielmarkierungen (in Abbildung 2.3 als T01,T02 etc. zu erkennen) zugeteilt, wobei eine Folge jedes Ziel maximal einmal enthält. Jeder Spieler kennt nur sein derzeitiges Ziel, zu Beginn also nur das erste Ziel der Folge. Kein Spieler hat Informationen über die Ziele der anderen Spieler.

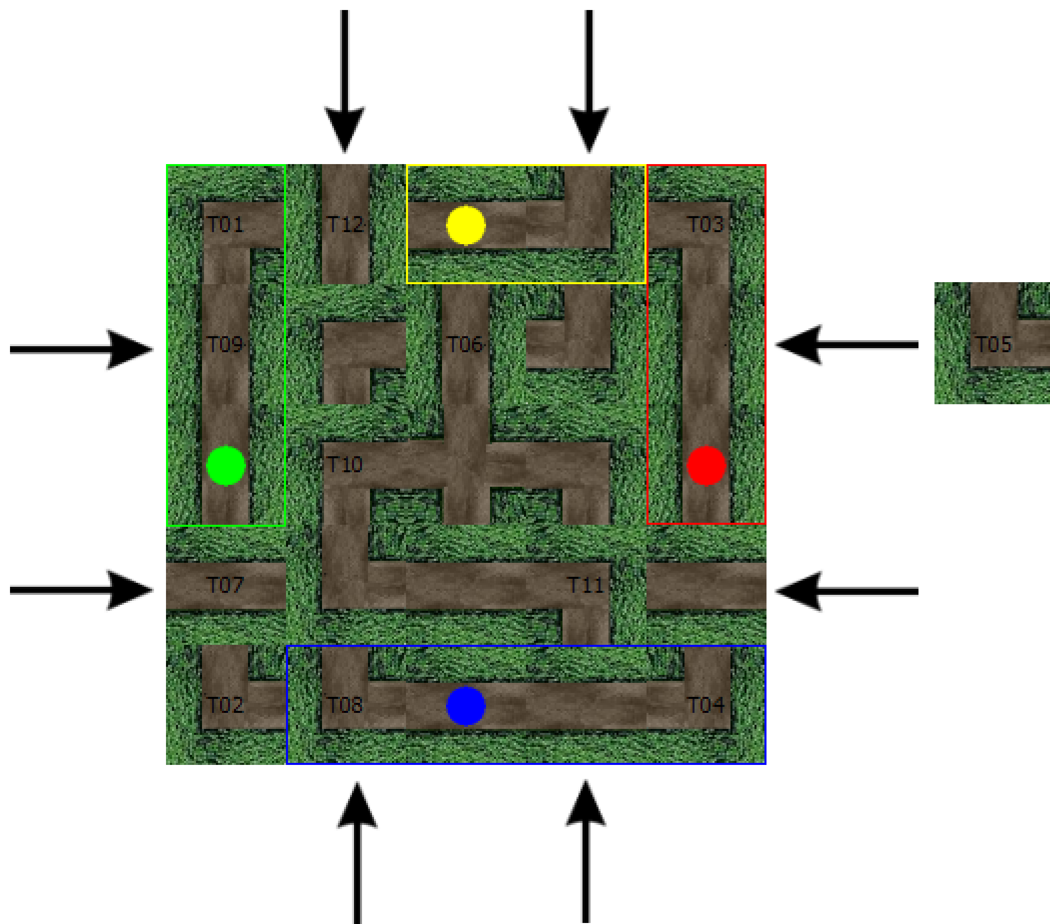

Abbildung 2.3: Komplettes Spiel mit Aktionsmöglichkeiten

Spielverlauf

Die Spieler müssen nacheinander einen Zug durchführen. Ein Zug besteht aus zwei Aktionen:

1. aus dem Einschieben des 26. Quadrats in eine der beweglichen Spalten oder Zeilen (in Abbildung 2.3 mit Pfeilen markiert). Die Ausrichtung des einzuschiebenden Quadrats ist dabei beliebig wählbar. Das gegenüberliegende Quadrat, dass dadurch herausgeschoben wird, ist das neue 26. Quadrat. Es darf nicht entgegen der Richtung geschoben werden, die vom letzten Spieler geschoben wurde.

2. aus dem Bewegen der Spielfigur auf eines der nun erreichbaren Quadrate. Es kann auch gewählt werden sich nicht zu bewegen. Als erreichbar gelten alle Quadrate, die von der Position der Spielfigur aus einen ununterbrochenen Pfad aufweisen (in Abbildung 2.3 in der Farbe des jeweiligen Spielers markiert). Eine Bewegung darf jedoch nicht auf einem Quadrat enden auf dem sich die Spielfigur eines anderen Spielers befindet. Passieren ist jedoch möglich.

Hat ein Spieler in seinem Zug sein derzeitiges Ziel erreicht, fährt er, sofern vorhanden, mit seinem nächstes Ziel fort.

Spielende

Das Spiel endet sobald ein Spieler alle ihm zugeteilten Zielmarkierungen mit seiner Spielfigur erreicht hat. Die Reihenfolge der Ziele ist dabei einzuhalten. Der Spieler, der das Spiel auf diese Weise beendet, hat gewonnen.

2.3 Maschinelles Lernen

In den folgenden Kapiteln werden grundlegende Begrifflichkeiten des Maschinellen Lernens und der Aufbau von reinforcement learning erläutert. Dies geschieht auf Basis der Werke von Richard S. Sutton [SB05] und Ethem Alpaydin [Alp08].

Unter dem Begriff des *Maschinellen Lernens* vereinen sich Methoden zur Generierung von Wissen. Maschinelles Lernen kann in drei Bereiche aufgeteilt werden. Hierbei handelt es sich um *überwachtes Lernen* (engl. *supervised learning*), *unüberwachtes Lernen*(engl. *unsupervised learning*) und *bestärkendes Lernen* (engl. *reinforcement learning*).

Auf dem *reinforcement learning* liegt das Hauptaugenmerk dieser Arbeit, im folgenden Abschnitt 2.4 wird genauer darauf eingegangen. Die anderen Bereiche sollen jedoch nicht unerwähnt bleiben.

Das *überwachte Lernen* zeichnet sich dadurch aus, dass erlernt wird, Eingabewerte auf Ausgabewerte abzubilden. Hierbei sind die korrekten Ausgabewerte bekannt. Ein bekanntes Beispiel für überwachtes Lernen ist die Schrifterkennung. Dabei wird dem lernenden Programm zu jedem Buchstaben eine Reihe von unterschiedlich geschriebenen Buchstaben gegeben. Die Eingabewerte sind in diesem Fall die geschriebenen Buchstaben, die Ausgabewerte die zugehörigen Buchstaben des Alphabets. Wurde erfolgreich gelernt, hat das Programm die Kriterien der verschiedenen Buchstaben erkannt. Es kann fortan mit einer hohen Wahrscheinlichkeit, auch bisher nicht bekannte, geschriebene Buchstaben dem Alphabet zuordnen.

Beim *unüberwachten Lernen* sind die Ausgabewerte nicht im Vorfeld bekannt. Das Ziel dieser Methode ist, in einer gegebenen Menge von Eingabewerten, Strukturen und Regelmäßigkeiten zu erkennen. Dies kann etwa durch *Clusteranalyse* geschehen, bei der nach Gruppierungen bzw. Häufungen in den Eingabedaten gesucht wird.

2.4 Reinforcement Learning

Bei reinforcement learning (RL) handelt es sich um einen Teilbereich des Maschinellen Lernens. Auch hier wird künstlich Wissen generiert, wobei das Erreichen von möglichst hohen *Belohnungen* durch Bewerten von *Zuständen* oder *Zustandsaktionspaaren* im Vordergrund steht.

Das moderne Feld des reinforcement learning bildete sich in den 1980er Jahren durch die Verbindung von mehreren, voneinander unabhängigen Forschungssträngen. Hierbei beschäftigt sich einer dieser Forschungssträngen, dessen Ursprung in der Psychologie liegt, mit *„trial and error"*-Lernen. Ein anderer beschäftigt sich mit dem Problem der optimalen Steuerung von Systemen mit Hilfe von Wertfunktionen und dynamischer Programmierung.

Basierend auf dieser Verbindung wurden, und werden weiterhin, eine Vielzahl von Ansätzen, Methoden und Algorithmen für die Lösung verschiedenster reinforcement learning Problematiken entwickelt.

Das beim reinforcement learning, der *Agent* genannte, lernende Programm interagiert bzw. reagiert, um zu lernen, mit der *Umwelt*. Die Umwelt ist die Umgebung, in der sich der Agent befindet. Als klassisches Beispiel dient ein Roboter, der in ein rechtwinkliges Labyrinth gesetzt wird und den Ausgang suchen soll. Die Umwelt ist in diesem Fall das gesamte Labyrinth mit allem, somit auch dem Roboter, darin. Der Agent steuert den Roboter. Die Umwelt befindet sich zu jedem Zeitpunkt in einem bestimmten *Zustand*, dieser wird dem Agenten mitgeteilt. Dieser Zustand kann die Position des Roboters sein, die er mit Hilfe seiner Sensoren ermittelt hat.

Der Agent verfügt über eine bestimmte Menge von *Aktionen*, die er ausführen kann. Das Durchführen von Aktionen hat Einfluss auf die Umwelt und verändert deren Zustand. Der Roboter kann sich als Aktion in jede Richtung bewegen. Durch seine Positionsänderung ändert sich auch der Zustand der Umgebung. Für jede Aktion erhält der Agent eine Rückmeldung von der Umwelt in Form einer *Belohnung*. Diese Belohnung ist numerischer Natur und der Agent versucht die Summe aller Belohnungen zu maximieren.

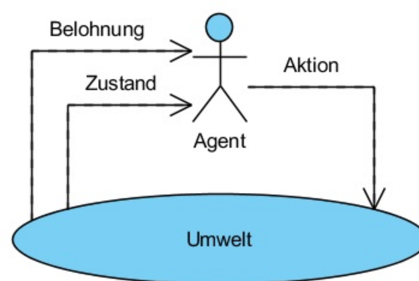

Abbildung 2.4: Agent Umwelt Schema

Zur Erfüllung der Aufgabe, hier das Erreichen des Ausgangs, sind für gewöhnlich mehr als nur eine Aktion nötig und so fallen die meisten Belohnungen vor Erfüllung der Aufgabe neutral aus. Negative Belohnungen sind ebenfalls möglich und stellen oft Wegkosten dar.

2.4.1 Markov-Entscheidungsprozess MDP

Bei der *Markov Eigenschaft* handelt es sich um eine Eigenschaft eines Zustandsübergangs der Umwelt. Diese Eigenschaft liegt vor, wenn der beobachtete Zustand alle relevanten Informationen enthält, sowohl über den Zustand als auch über die Sequenz von Positionen, die zu ihm geführt haben. So ist zum Beispiel die Komposition der Spielsteine bei klassischen Brettspielen wie Dame, Schach oder Mühle, ein Zustand welcher die Markov-Eigenschaft unterstützt.

Geht man von einem finiten Zustandsraum aus, lässt sich aus einem Pfad an Zuständen und Aktionen, die Wahrscheinlichkeit P, dass der nächste Zustand s' und dessen Belohnung r ist, bestimmen. Zum Zeitpunkt t gilt also in Anbetracht der Vergangenheit: (vgl. [SB05] Kap. 3.5)

$$Pr\{s_{t+1} = s', r_{t+1} = r | s_t, a_t, r_t, s_{t-1}, a_{t-1}, ..., r_1, s_0, a_0\}$$

Mit: Zustand s_i, Aktion a_i und Belohnung r_i zum Zeitpunkt i.

Ist die Markov Eigenschaft erfüllt, so hängt die Wahrscheinlichkeit P nur vom Zustand und der Aktion zum Zeitpunkt t ab und es gilt:

$$Pr\{s_{t+1} = s', r_{t+1} = r | s_t, a_t, r_t, s_{t-1}, a_{t-1}, ..., r_1, s_0, a_0\} = Pr\{s_{t+1} = s', r_{t+1} = r | s_t, a_t\}$$

Der *Markov-Entscheidungsprozess* (engl. *Markov decision process*, MDP) ist eine Modellierung für Optimierungsprobleme. Der MDP zeichnet sich durch diskrete Zeitschritte und die Einhaltung der Markov Eigenschaft aus. (vgl. [Alp08] Kap 16.3)

Eine reinforcement learning Aufgabe, bei der diese Eigenschaften erfüllt sind, nennt sich Markovscher Entscheidungsprozess.

2.4.2 Elemente des Reinforcement Learning

Abgesehen von den Basisbausteinen (*Agent, Umwelt, Zustand, Aktion* und *Belohnung*) wird reinforcement learning durch drei bzw. vier Hauptelemente geprägt: der *Strategie*, der *Belohnungsfunktion*, der *Wertfunktion* und ggf. einem *Modell der Umwelt*.

Unter der *Strategie* oder auch *Taktik* (engl. *policy*) versteht sich das Verhalten des Agenten. Eine Strategie π definiert im allgemeinen Fall die Wahrscheinlichkeit, mit der in einem Zustand s_t eine bestimmte Aktion a_t gewählt wird: $P(\{(s_t, a_t)\}) : \pi$. Im deterministischen Fall ist π eine Abbildung der wahrgenommenen Zustände der Umwelt auf Aktionen: $\pi : S \to A$.

Die *Belohnungsfunktion* (engl. *reward function*) ist die Abbildung jedes wahrgenommen Zustandes oder Zustandsaktionspaares auf eine numerische Belohnung. Sie bestimmt, wie erstrebenswert der jeweilige Zustand ist. Dadurch wird auch das Ziel des gegebenen Problems festgelegt, da der Agent einzig versucht diese Belohnung zu maximieren. Aufgrund dessen darf der Agent nicht die Möglichkeit besitzen, die Belohnungsfunktion zu ändern. Er könnte sonst sein Ziel ändern, wodurch er der Lösung des Problems nicht weiter dienlich sein könnte.

Um Langzeiterfolge zu gewährleisten, wird die *Wertfunktion* (engl. *value function* V^π) herangezogen. Sie spezifiziert, im Gegensatz zur Belohnungsfunktion, die die direkten Belohnungen der Zustände bestimmt, wie erstrebenswert ein Zustand auf lange Sicht ist. Ein Zustand mit geringer direkter Belohnung kann ein Trittstein zu einem Zustand mit hoher Belohnung sein und ist somit trotz geringer Belohnung sehr erstrebenswert. Das Gegenteil kann ebenfalls der Fall sein, ein Zustand mit hoher Belohnung kann nur noch Zustände mit geringer Belohnung zur Folge haben und somit, trotz hoher Belohnung, möglicherweise wenig erstrebenswert sein.

Ein *Modell der Umwelt* (engl. *model of the enviroment*) ([SB05] Kap. 1.3) ist eine relativ neue, optionale Erweiterung. Dabei handelt es sich um ein Modell, das die Umwelt

simuliert, und so mögliche Folgen von gewählten Aktionen voraussagt. Dies kann sich zB. im Rahmen eines deterministischen Eigenabbildes des Spiels gestalten. Ein solches Modell kann sowohl dem Agenten mitgegeben werden, als auch vom Agenten selbst durch Lernen gebildet werden. Bei vielen reinforcement learning Problemen kann es als eine sinnvolle Ergänzung zur Problemlösung beitragen.

2.4.3 Funktionsweise des Reinforcement Learning

Reinforcement learning kann sowohl auf *kontinuierliche* als auch auf *episodische* Probleme angewendet werden. Da Strategie-Gesellschaftsspiele ausschließlich episodisch sind, beschränken sich die folgenden Erläuterungen auf die episodische Anwendung von reinforcement learning. In dem Roboter Beispiel erstreckt sich eine Episode über das Hineinsetzten des Roboters in das Labyrinth bis hin zu dem Auffinden des Ausgangs durch den Roboter.

Zu Beginn verfügt der Agent noch über keinerlei Erfahrungen. Um diese anzusammeln, beginnt er damit zufällig ausgewählte Züge auszuführen, gemäß dem trial-and-error-Prinzip. Für diese Aktionen erhält er von der Umwelt Belohnungen, gemäß der Belohnungsfunktion. Mithilfe dieser Belohnungen aktualisiert er seine Wertfunktion, ordnet also den durchlaufenen Zuständen bzw. Zustands-Aktionspaaren Werte zu. Dies ist auch bekannt als *exploring* (dt. erkunden), da neue Möglichkeiten aufgetan werden. Nun folgt der Agent seiner Strategie und nutzt, sofern Erfahrungen vorhanden, die gespeicherten Werte um Entscheidungen zu treffen.

Eine mögliche Art der Strategieentwicklung ist die *Greedy-Suche* (dt. gierige Suche). Hierbei wird immer, wenn die Wahl besteht, die Aktion gewählt, die den höchsten Wert erzielt. Dieses Verhalten wird *exploiting* (dt. ausnutzen) genannt. In dem Roboter Beispiel kann dies bedeuten, dass sobald ein Weg gefunden wurde, er diesen immer wieder ausnutzt, da die bekannten Zustände über einen höheren Wert verfügen als die unbekannten. Dem Agenten droht also, nichts mehr dazu zu lernen.

Eine mögliche Herangehensweise an dieses *eploration-exploitation* Problem ist die *ϵ-Greedy-Suche.* Dabei wird mit der Wahrscheinlichkeit ε, mit$0 \leq ε \leq 1$, eine zufällige Aktion ausgewählt, um neue Erfahrungen zu sammeln. Mit der Wahrscheinlichkeit von 1-ε wird eine Greedy-Aktion gemäß der Strategie ausgewählt. Wird $ε = 0$, entspricht sie der Greedy-Suche, mit $ε = 1$ werden nur zufällige Aktionen ausgewählt. Die Wahl des ε ist somit von entscheidender Bedeutung. Ein höheres ε bedeutet schnelleres Lernen, beeinträchtigt jedoch den Agenten, sobald weitestgehend fertig gelernt wurde, und schränkt seine maximale Effektivität ein, vice versa mit einem niedrigen ε. Um die Effektivität zu erhöhen, kann ein sich über die Zeit verringerndes ε gewählt werden. Wird die ε-Greedy Suche angewandt, ist die Wahrscheinlichkeit jeder Aktion a_i in allen Zuständen s_j größer als 0. Ist dies der Fall, spricht man von einer *ϵ-soft* Strategie.

Herangehensweisen, bei denen die gespeicherten Werte nur die Wahrscheinlichkeit erhöhen, mit der ein Zug ausgewählt wird, sind ebenfalls geläufig. Ein simples Beispiel ist, dass bei zwei möglichen Aktion $a_1$und a_2 in einem Zustand s mit Wertigkeiten Q_1= Q(s,a_1)und Q_2= Q(s,a_2) die Aktion $a_1$bzw. a_2 mit der Wahrscheinlichkeit $\frac{Q_1}{Q_1+Q_2}$ bzw. $\frac{Q_2}{Q_1+Q_2}$ ausgewählt wird. Gewöhnlich fällt die Wahrscheinlichkeitsbestimmung bei dieser Art von Herangehensweise jedoch komplexer aus.

2.4.4 Reinforcement Learning Algorithmen

Für die Bildung der Wertfunktion und dadurch der Strategie wurden eine Reihe von *reinforcement learning Algorithmen* entwickelt. Sie regeln, wie mit Belohnungen umgegangen wird, und welche Werte dadurch aktualisiert werden sollen. Somit bestimmen sie entscheidend das Verhalten und den Erfolg des Agenten. Vier der bekanntesten Algorithmen sind der *Monte Carlo*, der *Q-Learning*, der *Sarsa* und der *Sarsa(λ)* Algorithmus. Q-Learning, Sarsa und Sarsa(λ) sind Algorithmen, die mit *temporaler Differenz (TD)* arbeiten. Das heißt, dass die Differenz zwischen dem momentanen Wert eines Zustands bzw. Zustandsaktionspaares und dem abgezinsten Wert des Folgezustands zur Aktualisierung der Werte genutzt wird. Alle diese TD-Algorithmen ist gemein, dass ein *Lernfaktor* α, mit$0 \leq \alpha \leq 1$, bestimmt, wie stark sich neue Erfahrungen auf das bisherige Wissen auswirken. Je höher α desto stärker fällt eine neue Erfahrung ins Gewicht. Bei $\alpha = 0$ werden neue Erfahrungen ignoriert, es wird also nicht hinzugelernt, ist $\alpha = 1$, so ersetzt die neue Erfahrung für gewöhnlich das vorherige Wissen komplett. Alle vier sind auf reinforcement learning Probleme anwendbar bei denen ein *nicht komplettes Wissen* des Systems vorliegt. Die folgenden Erklärungen beschränken sich auf die Anwendung mit Zustandsaktionspaaren und deren Wertigkeiten, den sogenannten *Q-Werten* [Q(Zustand,Aktion)].

Monte Carlo

Der *Monte Carlo* Algorithmus ist auf die Anwendung auf *episodische* Probleme beschränkt. Dabei wird eine Episode gemäß der Strategie durchlaufen, bis der Endzustand erreicht wird. Alle durchlaufenen Zustandsaktionspaare werden gespeichert und für sie Wertelisten angelegt. Bei dem *first visit* Monte Carlo Algorithmus wird bei jedem ersten Erreichen eines Zustandsaktionspaares die folgende Belohnung der Werteliste hinzugefügt. Wird innerhalb einer Episode ein Paar erneut erreicht, so wird dies nicht weiter beachtet. Beim *every visit* Monte Carlo werden die Belohnungen jedes Erreichens eines Paares in die Werteliste aufgenommen. Nach Abschluss der Episode wird aus jeder Werteliste ein Mittelwert gebildet. Q-Werte der Zustandsaktionspaare werden anschließend in Anbetracht des ermittelten Wertes aktualisiert.

In Pseudocode, unter der Anwendung von ε-Greedy-Suche, einer Strategie π:

Beispielcode 2.1 Monte Carlo: Pseudocode (vgl. [SB05] Abb. 5.6)

Initialisiere, für alle $s \in S, a \in A(s)$:
    $Q(s, a) \leftarrow$willkürlich
    $\pi(s) \leftarrow$willkürlich
    $Returns(s, a) \leftarrow$leere Liste
while(true){
    a) Generiere eine Episode mit π
    b) Für jedes Zustandsaktionspaar (s, a) aus der Episode:
        $R \leftarrow$Belohnung folgend des ersten Erreichens von (s, a)
        Füge R zu $Returns(s, a)$ hinzu
        $Q(s, a) \leftarrow$arithmetische Mittel($Returns(s, a)$)
    c) Für jedes s aus der Episode
        $a* \leftarrow max_a Q(s, a)$
        Für alle $a \in A(s)$:

$$\pi(s, a) \leftarrow \begin{cases} 1 - \varepsilon + \frac{\varepsilon}{|A(s)|} & a = a* \\ \frac{\varepsilon}{|A(s)|} & a \neq a* \end{cases}$$

}

Q-Learning

Bei dem Q-Learning Algorithmus wird zur Bildung eines Q-Wertes eine Schätzung aus dem nächsten Zeitschritt genutzt. Außer der direkten Belohnung wird der Q-Wert der Aktion mit dem höchsten Q-Wert des Folgezustandes in die Berechnung miteinbezogen. Dieser Wert wird mit einer *Skontorate* γ, $0 \leq \gamma < 1$, versehen, die festlegt, wie wichtig zukünftige Belohnungen sind. Bei einer Skontorate von 0 werden zukünftige Belohnungen ignoriert, für ein γ das gegen 1, strebt der Agent eine hohe Langzeitbelohnung an. Bei Q-Learning handelt es sich um einen *strategiefreien* (engl. *off-policy*) Algorithmus, da der Schätzwert der Folgeaktion ohne Nutzung der Strategie bestimmt wird.

In Pseudocode, unter der Anwendung von Greedy-Suche, einer Strategie π, einem Lernfaktor α und einer Skontorate γ:

Beispielcode 2.2 Q-Learning: Pseudocode (vgl. [Alp08] listing 16.3)

Initialisiere, für alle $s \in S, a \in A(s)$:
    $Q(s, a) \leftarrow$willkürlich
Für alle Episoden:
    Initialisiere s
    while(s nicht finaler Zustand){
        a) Wähle a mittels der Strategie aus Q
        b) Führe Aktion a aus
        c) beobachte Belohnung r und nächsten Zustand s'
        d) aktualisiere $Q(s, a)$:
            $Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma * max_{a'} Q(s', a') - Q(s, a)]$
        e) $s \leftarrow s'$
    }

Sarsa

Der *Sarsa* Algorithmus ähnelt stark dem Q-Learning-Algorithmus, jedoch ist Sarsa eine *strategiebasierte* (engl. *on-policy*) Methode. Hierbei wird der Q-Wert der gemäß Strategie bestimmten Aktion des folgenden Zustandes in die Wertbildung miteinbezogen. Wie beim Q-Learning wird dieser Wert mit einer Skontorate γ verrechnet.

In Pseudocode, unter der Anwendung von Greedy-Suche, einer Strategie π, einem Lernfaktor α und einer Skontorate γ:

Beispielcode 2.3 Sarsa : Pseudocode (vgl. [Alp08] listing 16.4)

Initialisiere, für alle $s \in S, a \in A(s)$:
    $Q(s,a) \leftarrow$willkürlich
Für alle Episoden:
    Initialisiere s
    Wähle a mittels der Strategie aus Q
    while(s nicht finaler Zustand){
        a) Führe Aktion a aus
        b) beobachte Belohnung r und nächsten Zustand s'
        c) Wähle a' mittels der Strategie aus Q
        d) aktualisiere $Q(s,a)$:
            $Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma^* Q(s',a') - Q(s,a)]$
        e) $s \leftarrow s'$,$a \leftarrow a'$
    }

Sarsa(λ)

Sarsa(λ) ist eine Weiterentwicklung des Sarsa Algorithmus und bezieht nicht nur einen Schritt sondern den gesamten Pfad mit ein, um Q-Werte zu bilden. Dabei bestimmt der Wert λ, mit $0 \leq \lambda \leq 1$, wie stark sich eine Wertanpassung eines Q-Wertes auf die anderen Q-Werte des Pfades auswirkt. Ein vergangener, x Schritte entfernter, Q-Wert würde eine Wertanpassung in der Stärke von λ^{x-1} erfahren. Wird $\lambda = 0$ gewählt wird nur der vorherige Schritt beeinflusst, somit entspricht Sarsa(λ=0) dem Sarsa Algorithmus. Bei $\lambda = 1$ reduziert lediglich die Skontorate γ die Wertänderung über die Schritte hinweg.

In Pseudocode, unter der Anwendung von Greedy-Suche, einer Strategie π, einem Lernfaktor α, einer Skontorate γ und einem λ:

Beispielcode 2.4 Sarsa(λ) : Pseudocode (vgl. [Alp08] listing 16.5)

```
Initialisiere Q(s,a) beliebig, e(s,a) = 0, alle s, a
Initialisiere, für alle s ∈ S, a ∈ A(s):
    Q(s, a) ←willkürlich
    e(s, a) ← 0
Für alle Episoden:
    Initialisiere s
    Wähle a mittels der Strategie aus Q
    while(s nicht finaler Zustand){
        a) Führe Aktion a aus
        b) beobachte Belohnung r und nächsten Zustand s′
        c) Wähle a′ mittels der Strategie aus Q
        d) δ ← r + γQ(s′, a′) − Q(s, a)
        e) e(s, a) ← 1
        f) Für alle (s, a):
            Q(s, a) ← Q(s, a) + αδe(s, a)
            e(s, a) = γλe(s, a)
        g) s ← s′,a ← a′
    }
```

Ein einfaches 8x10 Gridworld Beispiel veranschaulicht, wie sich Sarsa(λ) im Vergleich zu Sarsa auswirkt. Nur das Feld, das mit dem Stern gekennzeichnet ist, hält eine Belohnung bereit. Jede Bewegung von einem Feld zu einem anderen ist eine Aktion und löst einen Zustandswechsel aus. Der gewählte Pfad ist ein mögliches Verhalten eines Agenten. Die Größen der Pfeile in den rechten beiden Abbildungen zeigen, die Größenordnung der Wert der Zustandsaktionspaare nach dem einmaligen Durchlaufen.

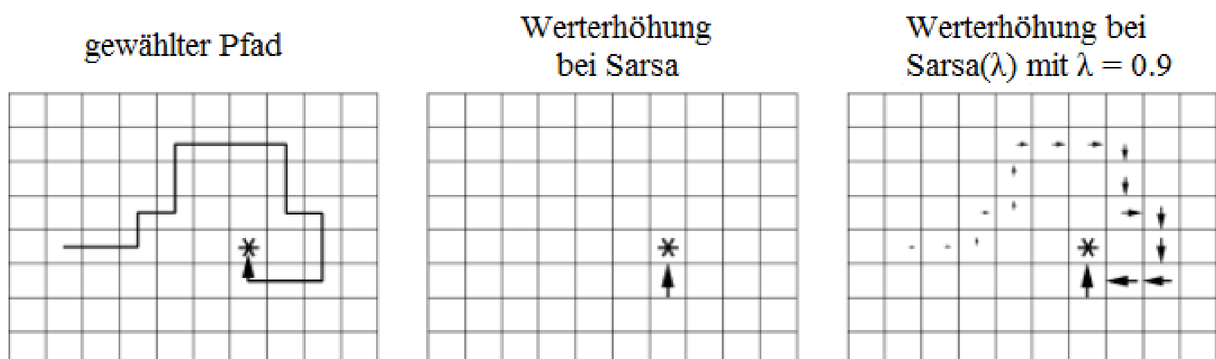

Abbildung 2.5: Gridworld: Vergleich des Lernerfolg von Sarsa zu Sarsa(λ)

(vgl. [SB05] Abb. 7.12)

Kapitel 3

Analyse

3.1 Spielanalyse

Bei dem Irrgartenspiel ist jedem Spieler das gesamte Spielbrett und sämtliche Positionen der Spieler und Ziele bekannt. Welches Ziel die jeweiligen Spieler derzeit verfolgen jedoch nicht, abgesehen vom eigenen Ziel. Bekannt ist ebenfalls die Zahl der restlichen Ziele aller Spieler, hingegen nicht wie diese Ziele Lauten. Durch die unbekannten Elemente zählt das Irrgartenspiel also zu den Spielen mit *nicht vollständiger* bzw. *perfekter Information*, im Gegensatz zu Schach zum Beispiel.

Es verfügt außerdem über alle Eigenschaften eines *German Game*, das sich durch die vollständige Teilnahme aller Spieler bis zum Ende des Spiels, eine kurze Spieldauer, geringe Zufallsfaktoren und relativ simple Regeln auszeichnet.

Außer den Aktionen selbst, die in jedem Zug erneut verfügbar sind, verfügt das Spiel über keinerlei verbrauchbare Ressourcen. Jeder Zug jedes Spielers ist, abgesehen vom Zustand des Spieles, unabhängig von früheren Zügen und anderen Spielern, wodurch keine direkte Interaktion der Spieler stattfindet. Da nur das eigene Ziel bekannt ist, fällt ein gezieltes Blockieren der Mitspieler schwer. Ebenfalls ist das Planen von Zügen für den Zeitraum nach Erreichen eines Ziels nur bedingt möglich, da das Folgeziel bis zum Erreichen dieses Ziels unbekannt bleibt.

Der Mittelpunkt des Spielbretts zeigt als Einziges vier Weg an Stelle von zweien und ist zentral gelegen. Dies könnte sich als strategisch wertvoll zeigen, da potentiell die Anzahl an mögliche Aktionen im nächsten Zug steigt.

3.2 Zustandsraumanalyse

Die Größe des Zustandsraumes hängt direkt von der Modellierung der Zustände ab. Die komplette Information eines Spielzustandes ist dabei ein erster Anhaltspunkt. Im Falle des Irrgartenspiels wäre dies zunächst die Spielfeldkonstellation, also Position und Ausrichtung sämtlicher Irrgartenquadrate sowie die Positionen aller Ziele und Spielfiguren. Hinzu kommen die Zielfolgen aller Spieler.

Da das Spiel jedoch verdeckte Elemente aufweist stehen diese Informationen einem Spieler nicht vollständig zur Verfügung. Der Spieler hat keine Informationen über die verdeckten Elemente, beim Irrgartenspiel sind dies die Ziele der Mitspieler und die

eigenen, nachfolgenden Ziele. Aus der Sicht eines Spielers können eigentlich verschiedene Spielzustände so gleich aussehen. Bei dem beobachteten Zustand fallen also die verdeckten Elemente weg.

Bei alleiniger Betrachtung des Spielfeldes, ohne Figuren oder Ziele, sind von dem 5x5 Feld neun der Quadrate fest, und 16 plus das 26. Quadrat zum Schieben, beweglich. Die festen Quadrate nehmen in jeder Spielkonstellation dieselbe Position und Ausrichtung ein, womit sie für den Zustand unerheblich sind. Die beweglichen Quadrate weisen für ihre Position eine Vielzahl von Permutationen auf. Die 17 Quadrate setzten sich aus acht Geraden und neun 90°-Kurven zusammen.

Die genaue Anzahl lässt sich mit der Formel für Permutationen mit Wiederholung für 2 Klassen bestimmen.

$$P_Q = \frac{n!}{k_1! * k_2!} = \frac{17!}{8! * 9!} = 24310 \lessapprox 2^{15}$$

Mit n gleich der gesamt Anzahl der Quadrate und k_i gleich der Anzahl der Geraden bzw. der 90°-Kurven.

Durch die Ausrichtungen der Quadrate ergibt sich, dass es für jede Permutation eine Anzahl von Kombinationen gibt. Jede der Geraden kann eine von zwei Ausrichtungen annehmen und jede 90°-Kurve eine von vier.

$$K_q = 2^{k_1} * 4^{k_2} = 2^8 * 4^9 = 2^8 * (2^9 * 2^9) = 2^{26}$$

Das 17. Extraquadrat besitzt jedoch keine Ausrichtung und kann sowohl eine Gerade als auch eine 90°-Kurve sein. Dies reduziert die Kombinationsmöglichkeiten um eine Zweierpotenz bei einer Gerade und um zwei Zweierpotenzen bei einer 90°-Kurve.

$$K_Q = p_1 * K_{Q1} + p_2 * K_{Q2} = \frac{8}{17} * 2^{25} + \frac{9}{17} * 2^{24} = 2^{24} * (2 * \frac{8}{17} + \frac{9}{17}) = 2^{24} * 1,470588 \lessapprox 2^{24} * 1,5$$

Mit p_i gleich der Wahrscheinlichkeit, dass das Extraquadrat eine Gerade bzw. 90°-Kurve ist und K_{Qi} gleich der Anzahl an Kombinationsmöglichkeiten für den gegebenen Fall.

Die Anzahl an Möglichkeiten für Ziele und Spielfiguren wird mit der Gleichung zur Berechnung der Anzahl der Variationen bestimmt. Bei dem Irrgartenspiel existieren zwölf Zielmarkierungen, wovon sich vier auf festen Quadraten befinden und somit in jedem Zustand dieselbe Position aufweisen. Dadurch nehmen sie keinen Einfluss auf den Zustandsraum. Die restlichen acht Ziele befinden sich auf beweglichen Quadraten, wodurch sie eine der 16 Positionen des beweglichen Raums einnehmen können oder sich auf dem Extraquadrat befinden. Auf jedem Quadrat ist maximal ein Ziel verzeichnet wodurch sich die Anzahl an Variationsmöglichkeiten wie folgend bestimmen lässt.

$$V_Z = \frac{n!}{(n-k)!} = \frac{17!}{(17-8)!} = \frac{17!}{9!} = 980179200 \lessapprox 2^{30}$$

Mit n gleich der Anzahl an möglichen Positionen und k gleich der Anzahl an Zielen.

Mit 25 möglichen Positionen für Spielfiguren und maximaler Spielerzahl ergibt sich mir der gleichen Formel die Anzahl der Variationsmöglichkeiten für die Spielfiguren.

$$V_S = \frac{n!}{(n-k)!} = \frac{25!}{(25-4)!} = \frac{25!}{21!} = 303600 \lessapprox 2^{19}$$

Mit n gleich der Anzahl an möglichen Positionen und k gleich der Anzahl an Spielern.

Die letzten Informationen, die jedem Spieler vorliegen, sind die bereits erreichten Ziele aller Spieler. Dabei kann es sich pro Spieler um bis zu zwei handeln, da beim Erreichen des dritten das Spiel beendet ist. Die Mächtigkeit dieser Menge ergibt sich wie folgt.

$$E_Z = S_z * [o + Z_{anz} * (o + Z_{anz-1}) = 4 * [1 + 12 * (1 + 11)] = 580 \lessapprox 2^{10}$$

Mit S_Z gleich der Spielerzahl, Z_{anz} gleich der Anzahl an Zielen und o gleich eins für die Möglichkeiten bei denen noch kein erstes bzw. zweites Ziel erreicht wurde.

Aus den so errechneten Werten ergibt sich, bei Erfassung aller aus Spielersicht vorhandenen Informationen eines Zustandes, ein Zustandsraum der Größe R.

$$R = P_Q * K_Q * V_Z * V_S * E_Z = 2^{96,414} \lessapprox 2^{97}$$

Somit handelt es sich um einen finiten Zustandsraum von diskreten Zuständen.

3.3 Modellierung der Zustände

Für den Lern- und Spielerfolg der Agenten ist die Modellierung der Zustände entscheidend. Ist der Zustand zu detailliert, wächst der Zustandsraum stark an, wodurch nicht nur die Lerngeschwindigkeit sondern auch der Wert des Gelernten sinkt. Nahezu gleiche Zustände werden nicht wiedererkannt, da sie sich in eher unwichtigen Details unterscheiden. Werden hingegen wichtige Informationen weggelassen, beschränkt dies die Chancen des Agenten auf gute Züge, und somit seinen Spielerfolg bis hin zur Nutzlosigkeit des Agenten bei zu stark beschränkten Zuständen.

Im Abschnitt 3.2 sind die Auswirkungen auf die Mächtigkeit des Zustandsraumes durch die verschiedenen Informationen analysiert. Die Konstellation des Spielbrettes ist eine essentielle Information und muss daher in einer Form Teil der Zustände sein. Hierzu zählt sowohl Position als auch Ausrichtung der Quadrate.

Die Position der Zielmarkierungen ist einerseits entscheidend, da dessen Erreichen der Kernpunkt ist, andererseits spielen die keinem Spieler zugeteilten Zielmarkierungen während eines Spieles keine Rolle. Ob eine Zielmarkierung einem Spieler zugeteilt wurde, und welchem Spieler, bleibt den Spielern weitestgehend verborgen. Die Variationsmöglichkeiten der Ziele sind unterdessen enorm und würden den Zustandsraum stark vergrößern. Es bietet sich an, nur die Position des eigenen Zieles zu modellieren, was eine Vereinfachung des Zustandsraumes bei sehr geringem Informationsverlust bewirkt.

$$V'_Z = m = 21 \lessapprox 2^5$$

m als Anzahl an möglichen Positionen für Ziele.

Die Verfolgung der von Mitspielern absolvierten Ziele wird ohne Informationen über die Positionen der Ziele hinfällig. Die Nutzbarkeit dieser Informationen beschränkt

sich ohnehin darauf, in wenigen speziellen Fällen einen andern Mitspieler geringfügig erfolgreicher zu blockieren.

Ebenfalls eine wichtige Information verbirgt sich in den Positionen der Spielfiguren. Die Position der eigenen, um den Weg zum Ziel zu finden, und die Positionen der anderen, um die durch diese Spieler blockierten Quadrate zu kennen. Da über die Mitspieler nicht bekannt ist, welche Ziele sie verfolgen, ist die Unterscheidung, welcher Spieler genau wo steht, nicht unbedingt nötig, welche Quadrate durch andere Spieler blockiert sind hingegen schon.

$$V'_S = m * \left(\frac{n'!}{k'! * (n' - k')!} \right) = 25 * (\frac{24!}{3! * 21!}) = 50600 \lessapprox 2^{16}$$

Mit m gleich der Anzahl an Möglichkeiten für die Position der eigenen Spielfigur und n' gleich der Anzahl an verbleibenden Möglichkeiten der Mitspieler und k' der Anzahl der Mitspieler.

Für die gewählte Modellierung ergibt sich folgende Größe für den Zustandsraum.

$$P_Q * K_Q * V'_Z * V'_S = 2^{59.576} \lessapprox 2^{60}$$

Die so modellierten Zustände setzten sich aus 16 Werten für die Brettkonstellation, vier Werten für die Spielerpositionen und einem für die Position des derzeitigen Zieles zusammen.

3.4 MDP Analyse

Aus dem Abschnitt 3.2 geht hervor, dass es sich um einen finiten Zustandsraum mit einer finiten Anzahl von möglichen Aktionen handelt. Somit ist eine *infinites MDP* auszuschließen.

Ausgehend von einem *finiten MDP* wird die Spielmechanik untersucht. Jeder Spielzug besteht aus zwei Phasen, die jeweils aus einer Spielbrettkonstellation mit einer einzelnen Aktion zu einer neuen Konstellation führen. Das Spiel lässt sich somit leicht in *diskrete Zeitabschnitte* unterteilen. Jeder der beiden möglichen Aktionstypen folgt einem streng deterministischen Ablauf. Das Spiel selbst ist somit mehr als hinreichend für ein MDP.

Wichtig sind jedoch die für den Agenten sichtbaren Abläufe. Die Einschiebeaktion läuft innerhalb des Zuges eines Agenten ab, und ist vom Agenten vollständig beobachtbar. Die Bewegungsaktion endet in einem, durch die anderen Mitspieler beeinflussten, beschränkt vorhersehbaren Zustand, der jedoch nicht von vergangenen Zuständen oder Aktionen abhängt. Es gibt jedoch eine Ausnahme, wenn ein Ziel erreicht und ein neues bestimmt wird. Die Wahl des neuen Ziels wird durch verdeckte Karten und bereits erreichte Ziele beinflusst. Diese Informationen stehen dem Agenten beim Beobachten des Zustandes nicht zur Verfügung, wodurch sich die Markov-Eigenschaft nicht vollständig einhalten lässt. Es liegt ein *teilweise beobachtbarer Markov-Entscheidungsprozess* vor, der, in diesem Fall, einem gewöhnlichen Markov-Entscheidungsprozess nahezu gleich kommt, da jede Episode über zwei oder drei Ziele sich in zwei bzw. drei Teilepisoden aufteilen liese. Jede dieser Teilepisoden erfüllt, da kein Zielwechsel eintritt, die Markov-Eigenschaft.

Kapitel 4

Entwicklung

Im Kapitel 4 wird in dem Abschnitt 4.1 zunächst ein Konzept für die Softwareelemente dargestellt. Dies betrifft sowohl den Agenten als auch die Umgebung. Hierfür werden die Grundlagen aus dem Kapitel 2 und die gewonnen Informationen aus dem Kapiteln 3 verwendet. Im Anschluss wird im Abschnitt 4.2 darauf eingegangen, wie die geplanten Elemente für das Irrgartenspiel umgesetzt wurden.

4.1 Konzept

Dieser Abschnitt beschäftigt sich zu Beginn mit der allgemeinen Kozeptionierung und geht dann anschließend genauer auf drei Teilbereiche ein, den Agenten, die Umwelt und die Kommunikation zwischen Agent und Umwelt.

Der Agent und die Umwelt sind, wie in Abschnitt 2.4 erklärt und Abbild 4.1 zu erkennen, nicht direkt sondern nur über wenige Kommunikationswege verbunden. Eine vollständige Trennung ist somit auch während der Entwicklungsarbeit angebracht. Für die Umwelt als auch für den Agenten werden jeweils ein Projekt angelegt und separat entwickelt. Allein bei der Kommunikationsschnittstelle muss dabei auf Kompatibilität geachtet werden.

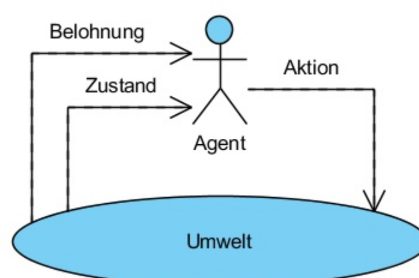

Abbildung 4.1: Agent Umwelt Schema

Bei Strategie-Gesellschaftsspielen fällt der Aktionsraum für gewöhnlich relativ groß aus, wobei nur ein geringer Teil der Aktionen zu einem Zeitpunkt, den Spielregeln nach, gestattet sind. Bei dem Irrgartenspiel z.B. gibt es 25 verschiedene Bewegungsaktionen (eine für jedes Zielfeld), von denen immer nur wenige ausführbar sind, da nur erreichbare Felder angesteuert werden dürfen. Bei komplexeren Strategie-Gesellschaftsspielen

wird, das Verhältnis aller existierender Aktionen zu den durch die Spielregeln erlaubten Aktionen tendenziell noch größer. Um einen frequenten Nachrichtenwechsel zu vermeiden, bei dem die Umwelt dem Agenten immer wieder mitteilt, dass er eine regelwidrige Aktion gewählt hat und eine andere wählen soll, wird die Umwelt dem Agenten vor seinem Zug eine Liste aller seiner regelkonformen Aktionen übermitteln. Dies spart ebenfalls Speicher, da der Agent eine Vielzahl von Zustandsaktionspaaren nicht speichern muss, die ohnehin zu keinem Spielzug geführt hätten.

4.1.1 Umwelt

Der Kernteil der Umwelt ist eine Instanz des Strategie-Gesellschaftsspiels, die durch eine Darstellung mit einer grafische Benutzeroberfläche GUI erweitert werden kann.

Für das Irrgartenspiel ist die Umwelt, wie in Kapitel 1 bereits erwähnt, schon teilweise vorhanden. Zu den bereits vorhandenen Elementen gehören die grundlegenden Spielfunktionen und eine GUI, die das Spielgeschehen darstellt.

Die GUI wird lediglich um wenige Elemente erweitert, um das Spielgeschehen besser darzustellen. Es ist eine Spielinstanz vorhanden, die Spielbrett und die Spieler verwaltet und auf die Einhaltung der Spielregeln achtet. Bei diesen Elementen sind ebenfalls einige kleinere Änderungen vorzunehmen, so dass die Spielregeln des Irrgartenspiels verwirklicht werden.

Um das Spiel für die Agenten nutzbar zu machen, wird eine weitere Programmebene über die Spielinstanz gesetzt. Ein Spielleiter soll als Kommunikator, Überwacher und Kritiker dienen. Für den Spielleiter ist eine einzelne Threadklasse vorgesehen, da die Umwelt sequenziell vorgeht.

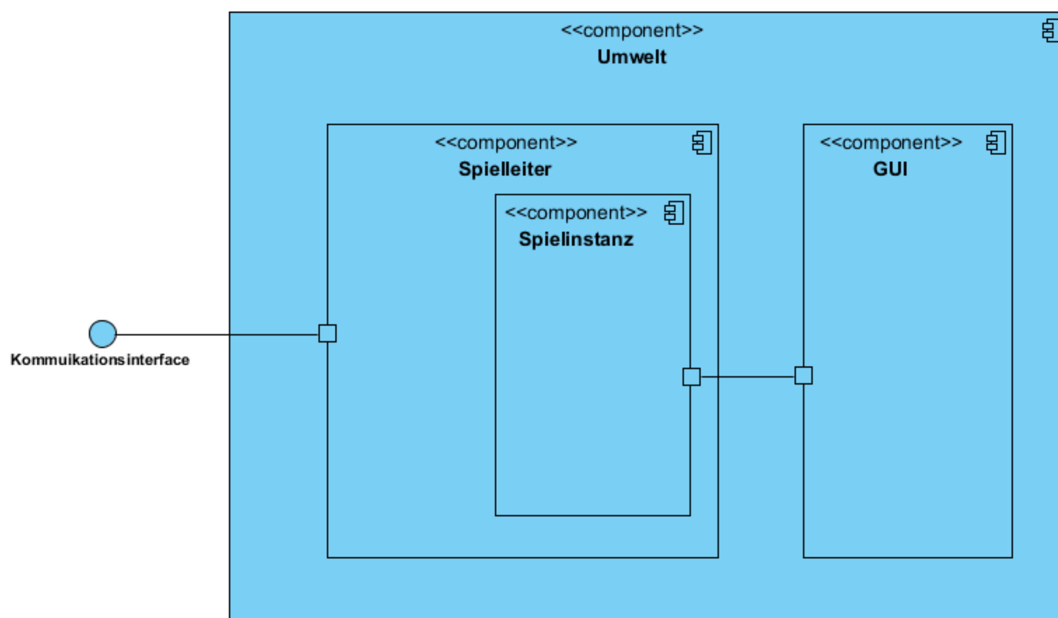

Abbildung 4.2: Komponentendiagramm der Umweltsoftware

Kommunikator: Um die Kommunikation zu ermöglichen, wird der Spielleiter die Kommunikationsschnittstellen der Umwelt bereitstellen. In seinem *run loop* wird zunächst die Interprozesskommunikation IPC initialisiert und anschließend auf die Teil-

nahme einer ausreichenden Anzahl von Spielern gewartet. Ist die Verbindungsaufbauphase abgeschlossen, regelt der Spielleiter die Kommunikation mit den Agenten. Dies geschieht gemäß dem Kommunikationskonzept aus Abschnitt 4.1.3.

Überwacher: Um den Spielablauf zu sichern, übernimmt der Spielleiter eine Reihe von Aufgaben. Der Spielleiter wird:

- für jeden Agenten vor seinem Spielzug eine Liste mit seinen möglichen Aktionen ermitteln und dem Agenten mitteilen.
- die von den Agenten gewählten Aktionen an das Spiel weiterleiten.
- unzulässige Aktionen durch die Spielinstanz erkennen.
- das Spielende registrieren und allen Agenten mitteilen.
- das Spiel für eine neue Episode vorbereiten.
- eine Logdatei über das Spiel und über den Kommunikationsverlauf anlegen.

Kritiker: Der Spielleiter wird die Züge der Agenten bewerten und entsprechend Belohnungen verteilen. Bei episodischen Problemstellungen haben sich Belohnungen für den Spielsieg bewährt (Quelle). Beim Irrgartenspiel ist jedes erreichte Ziel ein wichtiger Schritt zum Sieg. Das Irrgartenspiel lässt sich, wie im Abschnitt 3.4 erwähnt, in Teilepisoden für jedes zu erreichende Ziel unterteilen. Aus diesem Grund ist eine positive Belohnung durch den Spielleiter beim Erreichen eines Ziels angebracht. Der endgültige Spielsieg wird zusätzlich mit einer erhöhten Belohnung versehen, damit sein Stellenwert bestärkt wird.

4.1.2 Agent

Für jeden reinforcement learning Algorithmus wird eine eigene Agentenklasse erstellt. Die Grundstruktur aller Agenten ist jedoch gleich. Sie wird in einer abstrakten Klasse angelegt. Jeder der Agenten wird von dieser abstrakten Klasse abgeleitet.

Teil dieser Grundstruktur, und damit bei jedem Agenten gleich, ist der Aufbau des Gedächtnisses des Agenten. Im Gedächtnis werden die Zustandsaktionspaare und ihre zugeordneten Q-Werte gespeichert. Um die Zeit gering zu halten, die für das Heraussuchen eines Q-Werts benötigt wird, wird das Gedächtnis in einer Map organisiert. Beim Irrgartenspiel werden alle Q-Werte zu Beginn mit dem Wert Null initialisiert.

Den Rahmen der Agenten bildet eine Software mit GUI. Diese ermöglicht dem Benutzter, Anzahl und Art der Agenten festzulegen. Für jeden Spieler ist die Auswahl eines Algorithmus zu treffen. Abgesehen von den in Abschnitt 2.4.4 vorgestellten reinforcement learning Algorithmen, ist ebenfalls die Wahl eines Zufallsagenten oder keines Agenten möglich.

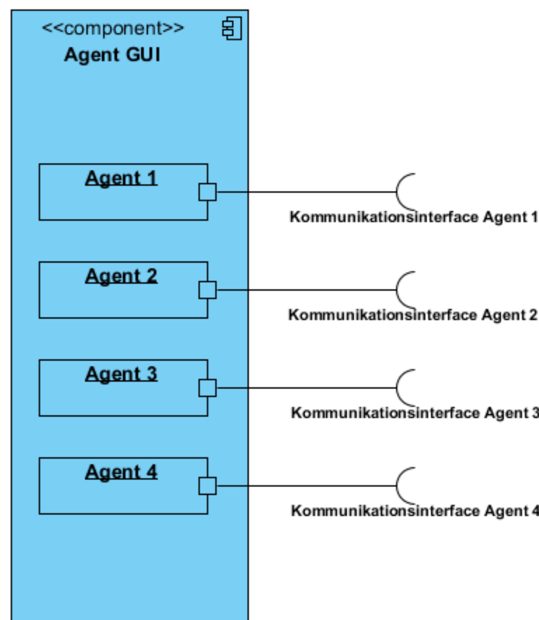

Figure 4.3: Komponentendiagramm der Agentensoftware

Da das Irrgartenspiel bis zu vier Spieler unterstützt, wird auch die GUI Anwendung für das Irrgartenspiel die Erstellung von bis zu vier Agenten unterstützen. Jeder Agent wird unabhängig von den anderen Agenten erstellt. Somit können bis zu vier Agenten, die auf unterschiedlichen Algorithmen basieren können, gleichzeitig getestet werden. Dabei sind auch die Variablen der Algorithmen für jeden Agenten über die GUI einstellbar. Dazu zählen hier die Lernrate α, die Skontorate γ, das λ des Sarsa(λ) Algorithmus und das ε für die ε-Greedy Suche.

4.1.3 Kommunikation

Da Agent und Umwelt strikt getrennt sind, wird für deren Kommunikation IPC nötig. Hierbei bietet sich ein *Client-Server-Modell* an. Die Software der Umwelt bildet einen eher passiven Teil und bietet als Dienst, die Möglichkeit zu spielen an. Die Agenten melden sich bei dem Spielleiter und nehmen diesen Dienst in Anspruch. Die Kommunikation findet hierbei direkt zwischen jedem individuellen Agenten und dem Spielleiter statt.

Für die Kommunikation wird ein kleines Set von Nachrichten definiert, die für den Spielverlauf und die Abgrenzung von Episoden benötigt werden. Die Nachrichten selbst sind Strings, deren Länge wenige Bytes nicht übertreffen.

Nachrichten Typen:

Neue Episode / new episode - NE

Ende der Episode / end of episode - EE

Zustand und Belohnung / state and reward - SR

Verfügbare Aktionen / available actions - AA

Aktion ausführen / do action - DA

Bestätigung / acknowledge - ACK

Die Abkürzungen NE, EE etc. bilden dabei den Anfang jeder Nachricht, wodurch sich die Nachrichtentypen leicht voneinander unterscheiden lassen.

Nach dem anfänglichen Verbindungsaufbau wird jeder Nachrichtenaustausch vom Spielleiter eingeleitet und besteht immer aus einer Nachricht vom Spielleiter zum Agenten und einer anschließenden Nachricht vom Agenten zum Spielleiter. Hierbei werden die Nachrichten für eine neue Episode(NE), das Ende einer Episode(EE) und Zustand und Belohnung(SR) von den Agenten mit einer Bestätigungsnachricht(ACK) quittiert.

Abgesehen von der NE, der EE und der ACK-Nachricht tragen alle Nachrichten noch weiteren Informationen. Die SR-Nachricht enthält den momentanen Zustand der Umgebung und dessen Belohnung. Die AA-Nachricht wird vom Spielleiter versandt und enthält eine Liste mit den AktionsIDs aller zurzeit vom Agenten ausführbaren Aktionen. Die vom Agenten versandte DA-Nachricht enthält die AktionsID der Aktion, die er ausführen möchte.

Die Abbildung 4.4 zeigt exemplarisch den Kommunikationsablauf anhand einer Episode, die nach einem Zug vom zweiten Spieler gewonnen wurde.

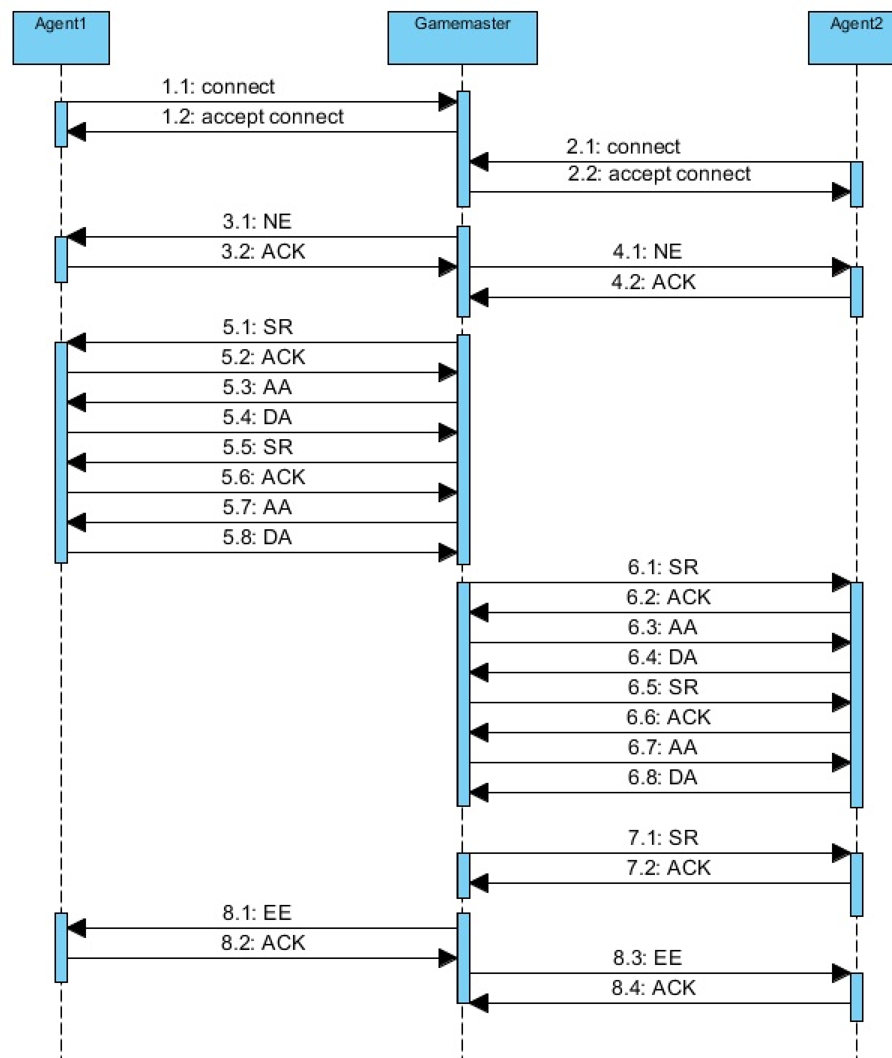

Abbildung 4.4: Sequenzdiagramm der Kommunikation mit 2 Agenten

Für die IPC werden Windows TCP Sockets verwendet, was sowohl eine verteilte Anwendung von Agenten und Umwelt als auch eine simple Implementation ermöglicht.

4.1.4 Konzeptüberblick

Für den Gesamtüberblick dient die Abbildung 4.5. Wie zu erkennen, sind Agent und Umwelt nur über eine Kommunikationsschnittstelle verbunden.

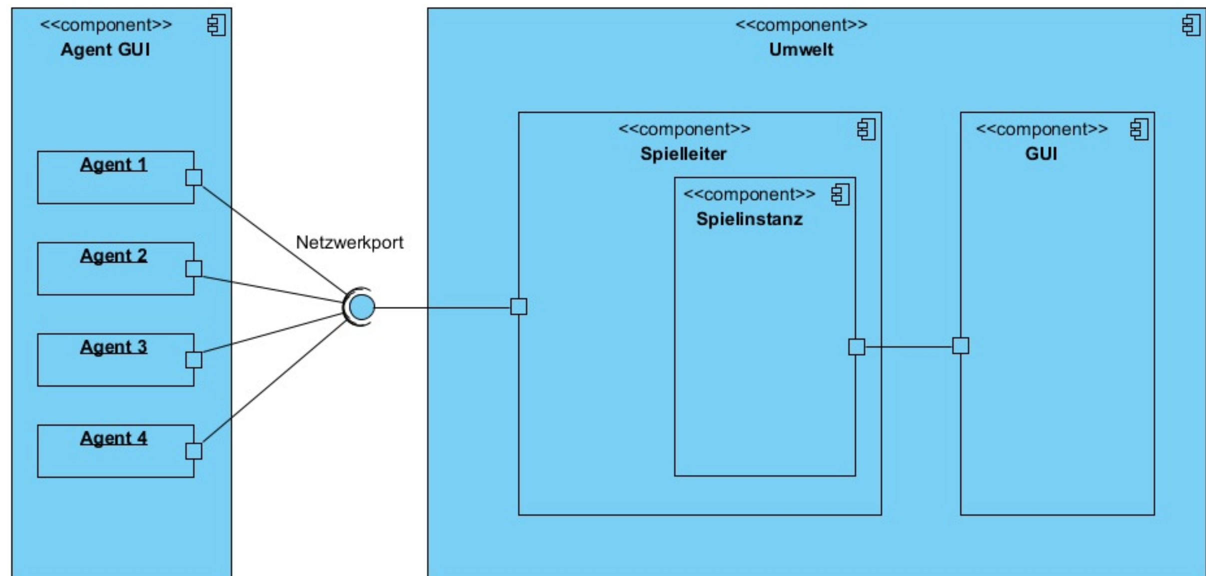

Abbildung 4.5: Komponentendiagramm Gesamtprojekt

Kommunikation zwischen dem Spielleiter und einem Agenten erfolgt immer in dem Zeitraum, in dem der Zug des jeweiligen Agenten stattfindet.

4.1.5 Agenten mit Umweltmodell

In diesem Abschnitt wird ein Konzept für einen Agenten mit Umweltmodell erstellt. Dieses Thema wird in dieser Arbeit jedoch lediglich theoretisch behandelt und dessen Konzept in Abschnitt 4.2 nicht weiter aufgegriffen. Jedoch wird sowohl der Aufbau als auch der Nutzen eines solchen Umweltmodells betrachtet. Agenten mit Umweltmodell sind ein umfangreiches Themengebiet und dieser Abschnitt stellt lediglich einen Anriss dar.

Ein Umweltmodell eines solchen Agenten kann, wie in Abschnitt 2.4.2 bereits erwähnt, dem Agenten mitgegeben oder von ihm selbst entwickelt werden. Für dieses Projekt wurde die Entscheidung getroffen, den Agenten ein Modell selbst entwickeln zu lassen. Dies lässt den Agenten auch weiterhin unabhängig von dem gewählten Spiel arbeiten. Es steht ohnehin kein Modell für das Spiel mit Mitspielern zur Verfügung.

Jeder Agent verfügt über ein eigenes Modell. Dadurch verändert sich das generelle Konzept der Agentensoftware leicht, wie in Abbildung 4.6 dargestellt.

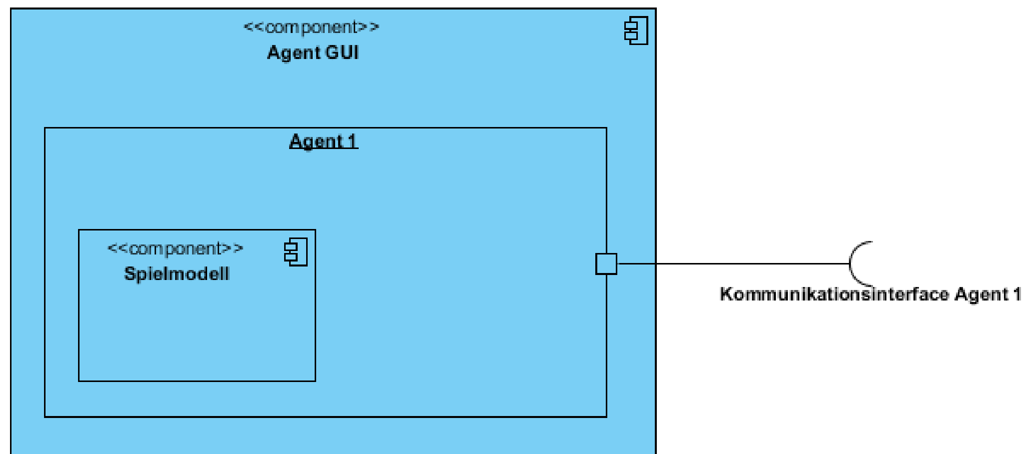

Abbildung 4.6: Komponentendiagramm Agent mit Modell

Für die Selbstentwicklung wird ein *sample* Modell gewählt. Diese Art von Modell bildet immer nur eine der möglichen Folgen einer Aktion ab. Die Betrachtung eines *distribution* Modells, bei dem alle Möglichkeiten und deren Wahrscheinlichkeiten abgebildet werden, würde den Rahmen dieses Abschnittes überschreiten. Das Spielmodell wird zusätzlich zu den normalen Funktionen des Agenten betrieben. Für das Konzept wird der Aufbau eines *Dyna*-Agenten gewählt.([SB05] Kap. 9.2 / [Sut90])

Damit der Agent ein Modell selbst entwickelt, wird er jedes Zustandsaktionspaar speichern, das durch Interaktion mit dem echten Spiel durchlaufen wird. Zu jedem dieser Zustandsaktionspaare wird der zuletzt aus diesem Paar folgende Zustand und die resultierende Belohnung gespeichert ([SB05] Kap. 9.1). Dies wird ähnlich wie das Gedächtnis des Agenten in einer Map organisiert. So entsteht ein deterministisches Eigenabbild der Umwelt.

Das so gebildete Modell wird parallel zur Interaktion mit dem realen Spiel genutzt, um *simulierte Erfahrungen* zu sammeln. Dabei wird ein zufälliger Zustand und eine zufällige Aktion ausgewählt und dem Modell übergeben. Das Modell liefert einen Folgezustand und eine Belohnung. Diese werden behandelt, so als kämen sie von der Umwelt. Durch die Anwendung des Lernalgorithmus des Agenten wird simulierte Erfahrung gewonnen, die so die Q-Werte beeinflusst. Nach jedem realen Zug wird eine feste Anzahl N mal simuliert.

In Pseudocode am Beispiel des Dyna-Q Agenten, unter der Anwendung von ε-Greedy-Suche, einem Lernfaktor α, einer Skontorate γ und Wiederholanzahl N:

Beispielcode 4.1 Dyna-Q Algorithmus (vgl. [SB05] Abb. 9.4)

Initialisiere $Q(s,a)$ und $Modell(s,a)$ für alle $s \in S, a \in A(s)$:
while(true){
  a)$s \leftarrow$derzeitiger nicht terminaler Zustand
  b)$a \leftarrow \varepsilon - Greedy(s,Q)$
  c) Führe Aktion a aus und beobachte Belohnung r und nächsten Zustand s'
  d)$Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma * max_{a'} Q(s',a') - Q(s,a)]$
  e)$Modell(s,a) \leftarrow s', r$
  f)Wiederhole N mal:
    $s \leftarrow$zufälliger bereits beobachteter Zustand
    $a \leftarrow$zufällige bereits in Zustand s gewählte Aktion
    $s', r \leftarrow Modell(s,a)$
    $Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma * max_{a'} Q(s',a') - Q(s,a)]$
}

Bei dem Dyna Agenten wird sowohl direktes Lernen, hier der Q-Learning Algorithmus (Algo. 4.1 a-d), als auch Modelllernen angewandt.(Algo. 4.1 e,f)

Unter Punkt e des Dyna-Q Algorithmus wird das Modell gelernt. In Punkt f wird simulierte Erfahrung gewonnen.

Der Nutzen eines solchen Modells kann an einem Gridworld Beispiel gezeigt werden. Bei diesem Beispiel muss ein Agent zu einem Ziel gelangen. Startposition ist hier das Quadrat, gekennzeichnet durch ein S. Das Ziel ist das Quadrat, gekennzeichnet durch ein G. Nach einer realen Episode bildet der Agent einen neuen Q-Wert für das angrenzende Feld gemäß dem Q-Learning Algorithmus. Mithilfe des Modells wird diese Erfahrung durch Simulation bei den anderen durchlaufenen Zustandsaktionspaare verbreitet.

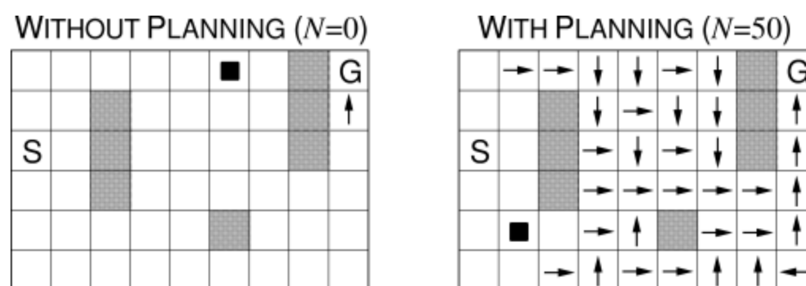

Abbildung 4.7: Gridworld Dyna-Q Agent

([SB05] Abb. 9.6)

Es ist jedoch zu beachten, dass es sich bei dem Irrgartenspiel im Gegensatz zur Gridworld nicht um ein voll deterministisches System handelt.

Das vom Dyna-Q Agenten genutzte Modell eignet sich nicht für den Monte Carlo Algorithmus, da keine kompletten Episoden durchlaufen werden.

Dadurch, dass das echte Spiel von mehreren Spielern gespielt wird, wartet jeder Agent einige Zeit auf die anderen Agenten. Diese Zeit kann mit einem Modell genutzt werden um weitere Erfahrungen anzusammeln, ohne dass der Agent dadurch langsamer agiert.

Jedoch wird das Irrgartenspiel stark von Mitspielern beeinflusst, die obendrein oft ihr Verhalten durch eigene neue Erfahrungen ändern oder gar zufällig handeln (z.B. im Fall der Teilnahme eines Zufallsagenten). Es stellt sich dadurch die Frage, wie zuverlässig ein sample Modell und die resultierende simulierte Erfahrung ist.

Da der Folgezustand einer Einschiebeaktion jedoch nicht von Mitspielern beeinflusst wird, ist das Modell für Fälle mit Einschiebeaktion sehr präzise. Hierdurch ist zu erwarten, dass die Effektivität dieser Züge durch das Modell deutlich gestärkt wird.

4.2 Realisierung

In diesem Abschnitt wird die Realisierung des in Abschnitt 4.1 vorgestellten Konzeptes beschreiben.

In Abschnitt 4.2.1 wird auf die Rahmenbedingungen, die genutzten Bibliotheken und die IPC Nachrichten eingegangen. Abschnitt 4.2.2 beschreibt anschließend die bereits vorhandenen Elemente der Umwelt und die Umsetzung des Spielleiters. Der abschließende Abschnitt 4.2.3 zeigt, wie die jeweiligen Algorithmen der Agenten umgesetzt wurden.

4.2.1 Allgemeine Realisierungsaspekte

Da im zugrundeliegenden Projekt die Programmiersprache C++ verwendet wurde, wird auch für dieses Projekt C++ eingesetzt. Für die Entwicklung wird die Entwicklungsumgebung Visual Studio 2008 [Mic]genutzt. Der bestehenden GUI liegt das Qt-Framework [Nokd] und dessen Bibliotheken zugrunde. Die Bibliotheken werden auch für die neu entwickelten Elemente verwendet. Die Visual Studio 2008er Version wies die höchste Kompatibilität zum Qt Designer plugin auf. Die GUIs wurden mit dem Qt Designer entworfen.

Die den Qt-Bibliotheken entspringende Klassen werden auch noch für viele nicht GUI Elemente verwendet. Näheres dazu kann bei den jeweiligen Erklärungen der Elemente gefunden werden.

IPC Nachrichten

Bevor auf Umwelt und Agent gesondert eingegangen wird, wird hier zunächst das von beiden genutzte Nachrichtenset beschrieben. Die Umsetztung der in Abschnitt 4.1.3 vorgestellten Nachrichten wird hier anhand von einigen Beispielnachrichten erläutert.

Die EE-, NE- und ACK-Nachrichtenstrings enthalten lediglich den Nachrichtenidentifikator.

Für die SR-, AA- und DA Nachrichten, die noch weitere Informationen tragen, wird ein Seperatorchar '~' definiert, der die einzelnen Informationen voneinander trennt.

SR-Nachricht: Identifikator ~ Zustand ~ Belohnung

```
SR~:556333:<5659<53D626G~0
```

AA-Nachricht: Identifikator ~ AktionsID ~ ... ~ AktionsID

```
AA~34~35~36~37
```

DA-Nachricht: Identifikator ~ AktionsID

```
DA~35
```

4.2.2 Implementierung der Umwelt

In der Abbildung 4.8 ist ein Klassendiagramm des Projektes, das die Umwelt modelliert, gegeben. Der Detailgrad der Klassen ist dabei beschränkt worden und zeigt die

wichtigeren Methoden und Attribute. Im Folgenden wird auf die einzelnen Klassen, deren Funktion und Umsetzung eingegangen.

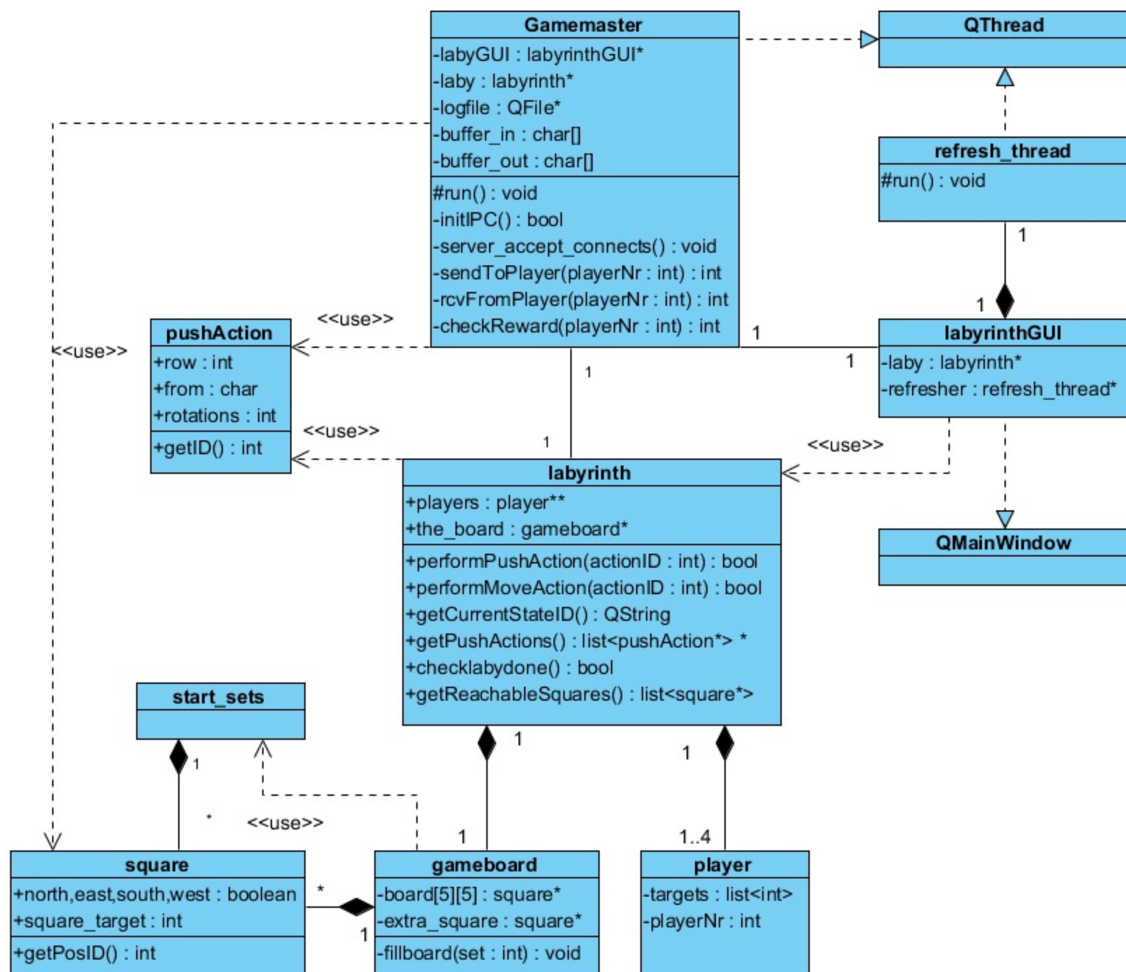

Abbildung 4.8: Klassendiagramm: Umwelt

4.2.2.1 Erläuterung der übernommenen Elemente

Bei den meisten Elementen handelt es sich um von dem früheren Projekt übernommene Klassen.

Die Klasse labyrinth stellt die Spielinstanz dar. Sie verfügt über ein Array von Spielern (player Klasse) und eine Instanz eines Spielbretts (gameboard Klasse).

Die gameboard Klasse verfügt über ein zweidimensionales Array, in dem 5x5 Quadrate (square Klasse) gespeichert sind. Sie verwaltet auch das 26. Quadrat. Art und Anzahl der Quadrate sind in der start_sets Klasse festgelegt, die zur Generierung eines Spielfelds genutzt wird. Die gameboard Klasse verfügt über eine Methode fillboard(int set), um pseudozufällige Startzustände zu erzeugen. Der Startzustand wird anhand des Übergabeparameters set erzeugt. Wird immer derselbe Wert übergeben, wird auch derselbe Startzustand erzeugt. Um einen zufälligen Startzustand zu erzeugen, wird der Funktion ein zufälliger Integerwert übergeben.

Die player Klasse enthält alle Informationen über einen Spieler, unter anderem die Spielernummer und die Liste seiner Ziele.

4.2.2.2 Umsetzung des Spielleiters

Komplett neuentwickelt ist der Spielleiter (Gamemaster Klasse). Die Gamemaster Klasse wird in diesem Kapitel fortan als der Gamemaster bezeichnet. Der Gamemaster wird von der Qt-Threadklasse QThread abgeleitet [?]. Er wird in der main() Methode erstellt und gestartet.

In Abbildung 4.9 ist der Ablauf der Hauptroutine des Gamesmasters dargestellt. Die genutzten Methoden und der veranschaulichte Code kann nach Bedarf in Anhang A gefunden werden.

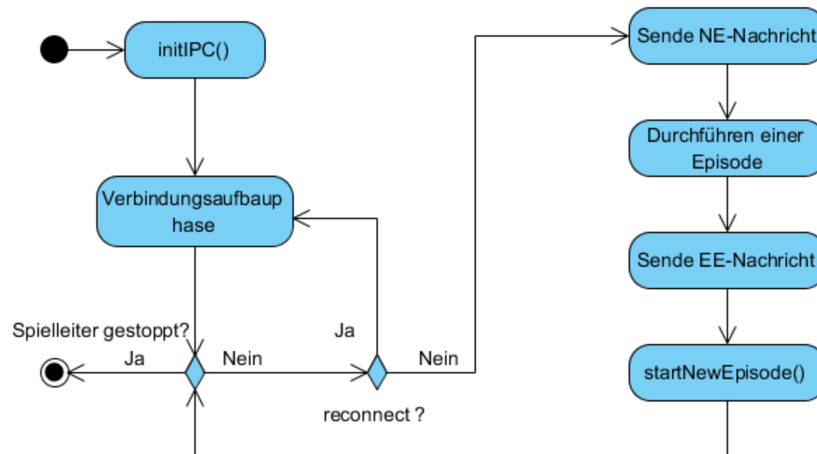

Abbildung 4.9: Aktivitätsdiagramm des Gamemasters

Beim Start des Gamemasters wird zunächst die Kommunikation mittels initIPC() initialisiert. Ist die Initialisierung abgeschlossen, so geht er in eine Verbindungsaufbauphase. Während dieser Phase akzeptiert der Gamemaster neue Verbindungen von Agenten, bis sich ausreichend Agenten für das Spiel angemeldet haben. Für ein Spiel mit z.B. vier Spielern wird gewartet bis sich vier Agenten verbunden haben.

In der Hauptschleife wird jede Episode durch eine NE-Nachricht vorbereitet und mit einer EE-Nachricht gefolgt vom Zurücksetzten des Spiels mit startNewEpisode() beendet.

Die Durchführung einer Episode wird in Abbildung 4.10 dargestellt und noch gesondert erläutert.

Tritt während einer Episode ein fataler Kommunikationsfehler auf, etwa durch den Verbindungsabbruch von einem Agenten, so wird der boolean reconnect auf true gesetzt und die laufende Episode frühzeitig beendet. Dies hat zur Folge, dass alle Verbindungen getrennt werden und der Gamemaster wieder in die Verbindungsaufbauphase geht.

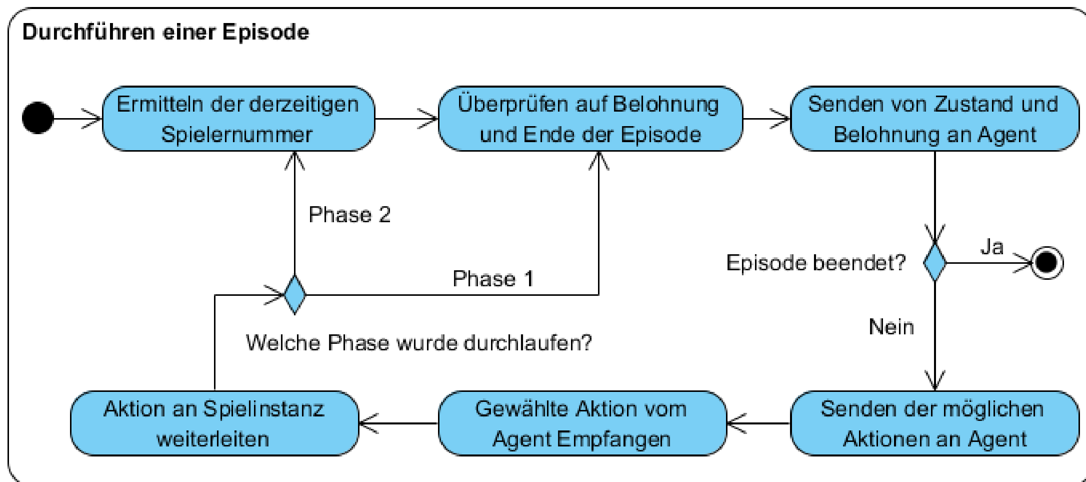

Abbildung 4.10: Aktivitätsdiagramm der Episodenaktivität

Der Episodenablauf besteht größtenteils aus einer Schleife, die den Zug eines Spielers abwickelt. Die Schleife wird solange durchlaufen, bis festgestellt wird, dass die Episode zu Ende ist.

In der Episodenschleife wird als Erstes vom Spielleiter überprüft, welcher Spieler an der Reihe ist. Anschließend folgen die beiden Phasen eines Spielzugs: Phase 1 Bewegen und Phase 2 Einschieben. Der Ablauf beider Phasen ist nahezu gleich:

1. Es wird überprüft, ob sich ein Agent eine Belohnung verdient hat und ob die Episode zu Ende ist.
2. Spielzustand und Belohnung werden an den betreffenden Agenten versandt und dessen Quittierung abgewartet.
3. Die Liste der möglichen Aktionen des Agenten wird zusammengestellt und an den Agenten gesandt.
4. Die vom Agent ausgewählte Aktion wird empfangen und an die Spielinstanz weitergeleitet.

Ist die Episode vorbei, wird noch die erhöhte Belohnung für das Gewinnen des Spiels an den Gewinner versandt.

4.2.2.3 GUI der Umwelt

Die GUI greift direkt auf die Spielinstanz zu und stellt das Spielgeschehen dar. Eine QTreadklasse refresher gibt mehrmals pro Sekunde einen Impuls an die GUI, die daraufhin den Spielzustand überprüft und abbildet. Eine hohe Bildwiederholrate ist nicht nötig, deswegen wird der Impuls lediglich alle 200 Millisekunden gesendet.

Abbildung 4.11: GUI der Umwelt

In Abbildung 4.11 ist links das Spielbrett dargestellt, um dieses herum alle Möglichkeiten, um das 26. Quadrat einzuschieben. Rechts oben befindet sich das 26. Quadrat und ein Knopf zur manuellen Bilderneuerung. Rechts unten befindet sich ein Logfeld, in dem unter anderem das Erreichen von Zielen durch Spieler mitgeteilt wird. Im unteren Bereich ist ein Siegeszähler angebracht, er kann durch betätigen des „reset” Knopfes zurückgesetzt werden.

4.2.3 Implementierung der Agenten

In diesem Abschnitt wird zunächst auf den allgemeinen Aufbau der Agenten durch die Beschreibung der agent Klasse eingegangen. Anschließend wird auf jeden Algorithmus gesondert eingegangen. Der Rahmen der Agenten und die GUI werden in Abschnitt 4.2.3.3 beschrieben.

Das Projekt der Agentensoftware setzt sich aus den in Abblildung 4.12 gezeigten Klassen zusammen.

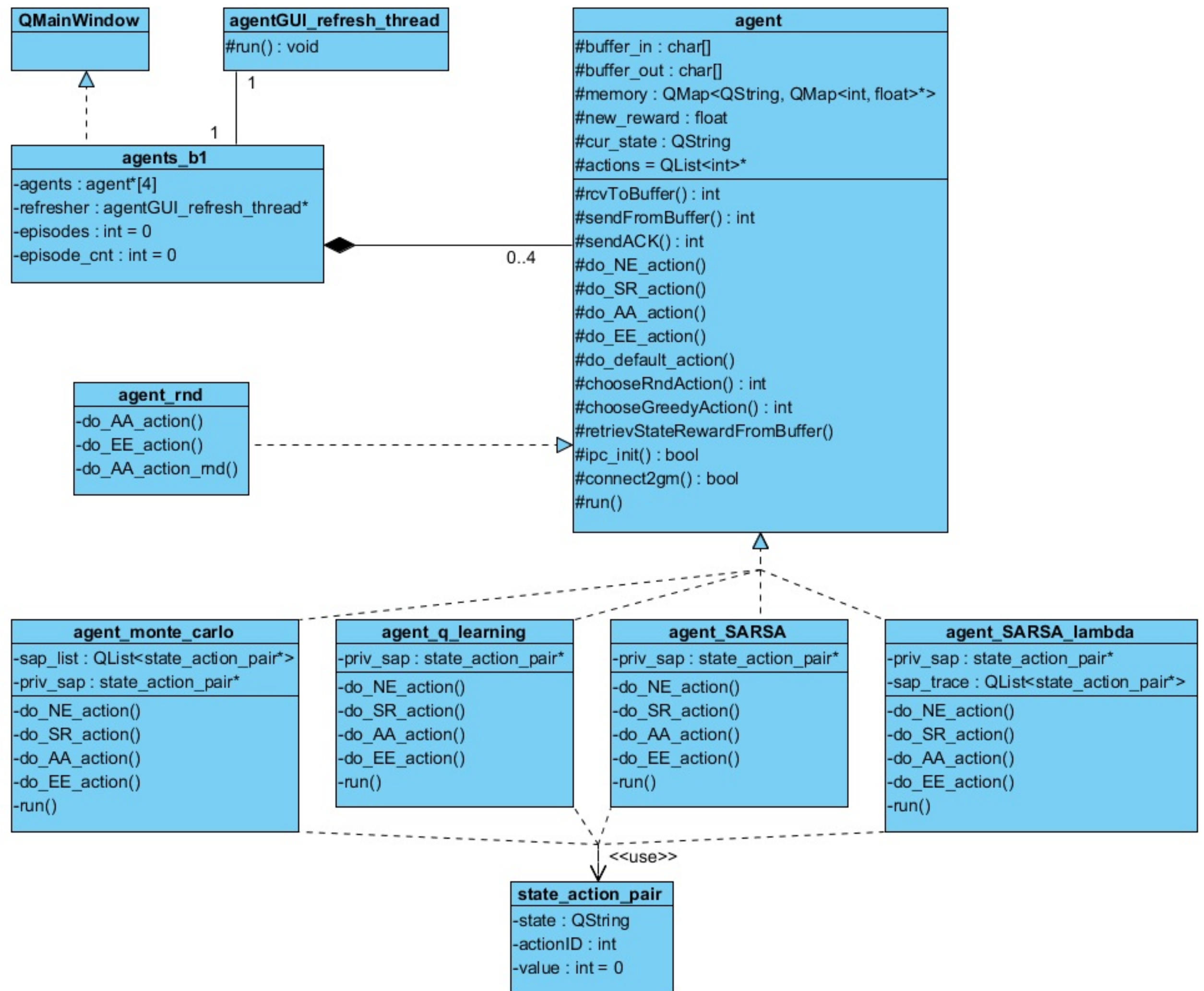

Abbildung 4.12: Klassendiagramm: Agenten

4.2.3.1 Allgemeiner Aufbau der Agenten

Die Oberklasse agent realisiert die generelle Struktur eines Agenten und ist ein QThread. Sie implementiert sowohl Methoden zur Kommunikation mit dem Spielleiter als auch Methoden für das reinforcement learning, die von allen Agenten benötigt werden. Das von allen RL-Agenten genutzte Gedächtnis wird ebenfalls hier implementiert.

Organisation des Agentengedächtnisses

Das Gedächtnis aller Agenten wird, wie in Abschnitt 4.1 bereits erwähnt, in einer Map organisiert. Die Gedächtnis Map wird mit Hilfe einer QHash, einer Hash Qt-Mapklasse

[Nokb], realisiert. Eine QHash erlaubt das schnelle Durchsuchen [Noka]. Sie ist Attribut jedes Agenten und ,wie folgt, deklariert:

```
QHash<QString, QHash<int,float>*> * memory;
```

In ihr werden Zustände einer zweiten QHash zugeordnet. Die zweite QHash ordnet AktionsIDs Q-Werte zu. Einem Zustand wird somit eine Reihe von passenden Aktionen zugeordnet, denen jeweils ein Q-Wert zugeordnet ist.

Mit der Zuweisung:

```
QList<int> bekannte_ActionIDs = memory->value(Zustand)->keys()
    ;
```

wird eine Liste über alle dem gegebenen Zustand zugeordneten AktionsIDs erlangt.

Ein Gedächtniseintrag wird ausgelesen mit:

```
float Q_Wert = memory->value(Zustand)->value(AktionsID);
```

Das Hinzufügen oder Ersetzten eines Eintrages erfolgt durch die folgende Zeile:

```
memory->value(Zustand)->insert(AktionsID,Q_Wert);
```

Funktionsweise der Agenten

Die Hauptroutine ist bei allen Agenten gleich und ist in Abbildung 4.13 dargestellt.

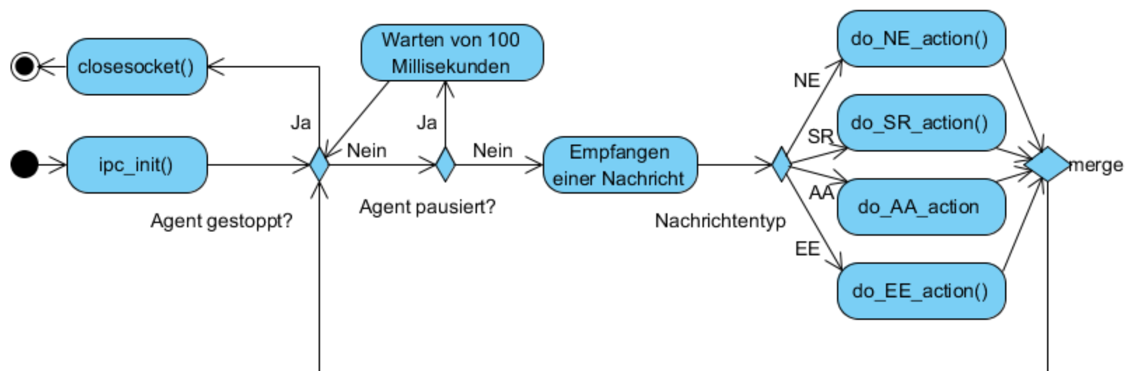

Abbildung 4.13: Aktivitätsdiagramm der Agentenhauptroutine

Zu Beginn wird, genau wie beim Spielleiter, die IPC initialisiert. Solange der Agent nicht endgültig gestoppt wird, wird die Hauptschleife ausgeführt. Zu Beginn jedes Schleifendurchlaufs wird überprüft, ob der Agent pausieren soll. Soll er dies, so wartet er in einer Schleife.

Nun folgt die Hauptaufgabe. Es wird auf eine Nachricht vom Spielleiter gewartet. Ist eine Nachricht eingetroffen, wird je nach Nachrichtentyp (NE,SR,AA oder EE) eine andere do_XX_action() Methode ausgeführt. In den do_XX_action() Methoden sind die Funktionalitäten der verschiedenen Agenten untergebracht. Jeder Algorithmus implementiert die do_XX_action() Methoden neu. Allen do_XX_action() Implementationen ist jedoch gemein, dass sie eine Antwortnachricht an den Spielleiter schicken.

4.2.3.2 Umsetzung der Reinforcment Learing Algorithmen

In diesem Abschnitt wird die Implementation der Algorithmen beschrieben. Hierzu wird nach einem Unterabschnitt zur Realisierung der ε-Greedy-Suche jeder Algorithmus in einem eigenen Unterabschnitt erläutert.

Allen ist gemein, dass ihre do_AA_action() Methode mit den folgenden 2 Aufrufen beginnt:

```
1 void agent_SARSA::do_AA_action(void){
2     /** Erstellen der Liste der verfügbaren Aktionen */
3     retrievActionsFromBuffer();
4 
5     /** Abwarten von msleeper Millisekunden, eingestellt in
         der GUI */
6     msleep(msleeper);
```

ε-Greedy-Suche

Für die Umsetzung der in Abschnitt 2.4.3 vorgestellten ε-Greedy-Suche wird die Methode chooseEpsilonGreedyAction() geschaffen. Sie wird von allen implementierten reinforcement learning Agenten genutzt.

In der Methode wird mit der Wahrscheinlichkeit ε die Methode chooseRndAction() aufgerufen, die eine zufällige AktionsID aus den vom Spielleiter gesendeten AktionsIDs auswählt und wiedergibt.

Mit der Restwahrscheinlichkeit wird die chooseGreedyAction() Methode ausgeführt. Diese überprüft, die zu dem derzeitigen Zustand cur_state im Gedächtnis gespeicherten AktionsIDs und gibt die AktionsID mit dem höchsten Q-Wert wieder, die sich auch in der vom Spielleiter gesendeten AktionsID Liste befindet. Trifft dies auf mehrere Aktionen zu, wird zufällig eine dieser Aktionen ausgewählt.

Wird keine gefunden, wird ein negativer Wert zurückgegeben und in der chooseEpsilonGreedyAction() die chooseRndAction() aufgerufen.

Monte Carlo

Das Generieren einer Episode (Algo. 2.1 Punkt a)) geschieht durch den Aufruf der do_SR_action() und der do_AA_action(). In der do_SR_action() wird, abgesehen von den Aufrufen von retrievStateRewardFromBuffer() und sendACK(), noch die Belohnung dem letzten Zustandsaktionspaar zugeordnet.

```
1 priv_sap->setValue(new_reward);
```

In der do_AA_action() wird mit chooseEpsilonGreedyAction() eine Aktion gewählt und per sendFromBuffer() an den Spielleiter gesendet.

Während der Episode wird in der do_AA_action() Methode eine Liste über die in dieser Episode erreichten Zustandsaktionspaare angelegt:

```
1 /** Merken des Zustandes s und der ausgewählten Aktion a
2     in einem state_action_pair Objekt */
3 priv_sap = new state_action_pair(cur_state, action);
```

```
4
5  /** Das zum Ersten mal erreichte Zustandsaktionspaar zur
6      Zustandsaktionspaarliste hinzufügen (Vorbereitung für
         Punkt b) )*/
7  if(!sap_inlist){
8      sap_list.push_front(priv_sap);
9  }
```

Bei dem Monte Carlo Algorithmus wird für jedes Zustandsaktionspaar eine Liste mit allen erhaltenen Bewertungen angelegt (Punkt b)). Da diese Listen lediglich dazu genutzt werden, das arithmetische Mittel für Q-Werte zu berechnen, können diese Listen mit Hilfe einer mathematische Umformung durch den Q-Wert und die eigentliche Mächtigkeit der Liste ersetzt werden. Der neue Q-Wert wird, wie in Algorithmus 4.2 3. dargestellt, berechnet. Die Punkte 1. und 2. zeigen die Äquivalenz zum arithmetischen Mittel der Liste.

Beispielcode 4.2 Äquivalenzumformung

1. $Q(s,a)_n = \frac{1}{n}\sum_{i=1}^{n} R_i$
2. $Q(s,a)_{n+1} = \frac{1}{n+1}\sum_{i=1}^{n+1} R_i = \frac{1}{n+1}(\sum_{i=1}^{n} R_i + R_{n+1}) =$

   $\frac{1}{n+1}(n * \frac{1}{n}\sum_{i=1}^{n} R_i + R_{n+1}) = \frac{1}{n+1}(n * Q(s,a)_n + R_{n+1})$
3. $Q(s,a)_{n+1} = \frac{1}{n+1}(n * Q(s,a)_n + R_{n+1})$

Mit R_i gleich der i-ten beobachteten Belohnung des Zustandsaktionspaares.

Anstelle der Listen der Belohnungen wird für jedes Zustandsaktionspaar abgesehen vom Q-Wert nur noch die Mächtigkeit gespeichert. Die Mächtigkeiten werden in einer, parallel zum Gedächtnis angelegten, QHash organisiert.

Die Bestimmung der neuen Q-Werte wird in der do_EE_action() Methode vollzogen:

```
1  /** Durchgehen der Zustandsaktionspaarliste */
2  int return_new = 0;
3  while(!sap_list.isEmpty()){
4          /** Auslesen des alten Q-Wertes Q(s,a) und der
5              entsprechenden Mächtigkeit n aus dem Gedächtnis */
6          ...
7          /** Berechnen des neuen Q-Wertes nach
               Äquivalenzalgoritumus */
8          float sum_n = q_old * n;
9          return_new += sap_list.front()->getValue();
10         float q_new = (sum_n + return_new) / (n+1);
11         n++;
12         /** speichern des neuen Q-Wertes und
13             der neuen Mächtigkeit im Gedächtnis */
14         ...
15         delete sap_list.takeFirst();
16     }
```

Schritt c) aus dem Monte Carlo Algorithmus 2.1 wird durch das Aufrufen der chooseEpsilonGreedyAction() Methode bei der Aktionsauswahl abgedeckt.

Q-Learning

Hier folgt die Erläuterung zur Implementierung des Q-Learning Algorithmus.

Die do_NE_action() und do_EE_action() Methoden senden lediglich eine ACK-Nachricht an den Spielleiter und zählen den Episodenzähler hoch.

Die Schritte a), b) und e) aus dem Algorithmus 2.2 werden bei dem Q-Learning Agenten in der do_AA_action() Methode abgehandelt. Sie werden durch die folgenden Zeilen realisiert:

```
1  /** Auswahl der Aktion a mittels Strategie ( Punkt a)) */
2  int action = chooseEpsilonGreedyAction();
3  /** Merken des Zustandes s und der ausgewählten Aktion a
4      in einem state_action_pair Objekt ( Punkt e) */
5  priv_sap = new state_action_pair(cur_state, action);
6  /** Übermitteln der Aktion an den Spielleiter ( Punkt b)) */
7  sprintf(buffer_out, "%s%s%d", DO_ACTION_MSG, SEPERATOR, action);
8  sendFromBuffer();
```

Die Schritte c) und d) werden in der do_SR_action() Methode verwirklicht. Nach dem Aufruf der retrievStateRewardFromBuffer() Methode, um die Belohnung und den Folgezustand zu erfassen (Punkt c)), wird der alte Q-Wert aus dem Gedächtnis geladen.

Anschließend wird der neue Q-Wert berechnet:

```
1  /** Aktualisieren von Q(s,a) (Punkt d) )*/
2  /** Bestimmen von max_a'Q(s',a') und Speichern in
      max_exp_value */
3  max_exp_value = 0;
4  if(memory->contains(cur_state)){
5      QList<int> known_actionIDs = memory->value(cur_state)->
           keys();
6      for(int i = 0; i < known_actionIDs.size(); i++){
7          int tmp = memory->value(cur_state)->value(
               known_actionIDs[i]);
8          if(tmp > max_exp_value){
9              max_exp_value = tmp;
10         }
11     }
12 }
13 /** Berechnen des neuen Q-Wertes Q(s,a) */
14 q_new = q_old + (alpha * (new_reward +
15                 (skonto * max_exp_value) - q_old));
```

Der so ermittelte neue Q-Wert wird im Gedächtnis gespeichert.

Sarsa

Bei der do_SR_action() wird zusätzlich zum Senden einer ACK-Nachricht lediglich noch die retrievStateRewardFromBuffer() aufgerufen. Dies verwirklicht Punkt b) des Sarsa Algorithmus (Algo. 2.3).

Der Großteil des Sarsa Algorithmus läuft in der do_AA_action() Methode des Sarsa Agenten ab:

```
1  /** Entspricht beim ersten Aufruf von do_AA_action() in einer
2      Episode: der Zeile vor der while-Schleife des Algo. */
3  /** Danach: Auswahl der Aktion a' mittels Strategie (Punkt c))
       */
4  int action = chooseEpsilonGreedyAction();
5
6  /** priv_sap ist nur NULL bei dem ersten Aufruf von
7      do_AA_action() in einer Episode */
8  if(priv_sap != NULL){
9      /** Aktualisieren von Q(s,a) (Punkt d) )*/
10     /** Auslesen von (Q(s,a)) und Q(s',a') aus dem Gedächtnis
          */
11     q_old = memory->value(priv_sap->state)->value(priv_sap->
          actionID);
12     q_next = memory->value(cur_state)->value(action);
13     /** Berechnen des neuen Q-Wertes Q(s,a) */
14     q_new = q_old + (alpha * (new_reward + (skonto * q_next) -
           q_old));
15     /** Speichern des neuen Q-Wertes im Gedächtnis */
16     memory->value(priv_sap->state)->insert(priv_sap->actionID,
          q_new);
17 }
18 /** Merken des Zustandes s' und der ausgewählten Aktion a'
19     in einem state_action_pair Objekt (Punkt e) */
20 priv_sap = new state_action_pair(cur_state, action);
```

Zuletzt wird in der do_AA_action() noch die gewählte Aktion an den Spielleiter übermittelt (Punkt a)).

Sarsa(λ)

Beim Sarsa(λ) Agenten gestalten sich die do_NE_action() und die do_SR_action() eben so wie bei dem Sarsa Agenten. Die do_SR_action() verwirklicht somit ebenso den Punkt b) des Sarsa(λ) Algorithmus (Algo. 2.4).

Da der Sarsa(λ) jedoch mehrere Schritte für die Q-Wert Berechnung miteinbezieht, wird eine Liste sap_trace über die in einer Episode besuchten Zustandsaktionspaare angelegt.

Die Punkte a), b), c) und g) sind auf gleiche Weise und an gleicher Stelle wie beim Sarsa Agenten implementiert und werden hier nicht erneut aufgezeigt. (Punkt g) entspricht dem Punkt e) im Sarsa Algorithmus)

```
1  /** priv_sap ist nur NULL bei dem ersten Aufruf von
     do_AA_action()
```

```
2      in einer Episode */
3  if(priv_sap != NULL){
4      /** Auslesen von (Q(s,a)) und Q(s',a') aus dem Gedächtnis
          */
5      q_old = memory->value(priv_sap->state)->value(priv_sap->
          actionID);
6      q_next = memory->value(cur_state)->value(action);
7      /** Berechnen von delta (Punkt d) ); */
8      delta = new_reward + (skonto * q_next) - q_old;
9      /** Bereitstellen von e(s,a) (Punkt e) ) */
10     e_sa = 1;
11     /** Aktualisieren von Q(s,a) für alle (s,a)
12         des Episodenverlaufs sap_trace(Punkt f) )*/
13     QList<state_action_pair*>::iterator iter;
14     for(iter = sap_trace.begin();
15                     iter != sap_trace.end(); iter++){
16         float q_sa = memory->value((*iter)->state)
17                         ->value((*iter)->actionID);
18         q_sa = q_sa + (alpha * delta * e_sa);
19         memory->value((*iter)->state)
20                         ->insert(((*iter)->actionID),q_sa);
21         e_sa = skonto * lambda * e_sa;
22     }
23 }
24 /** hinzufügen des Zustandaktionspaares zum Episodenverlauf */
25 sap_trace.push_front(priv_sap);
```

4.2.3.3 GUI der Agenten

Wie bei der GUI der Umwelt hat auch die GUI der Agenten eine QThreadklasse agentGUI_refresh_thread die mehrmals pro Sekunde einen Impuls an die GUI gibt, um die unter 5) dargestellten Elemente zu aktualisieren. Eine hohe Bildwiederholrate ist auch hier nicht nötig, deswegen wird der Impuls lediglich alle 200 Millisekunden gesendet.

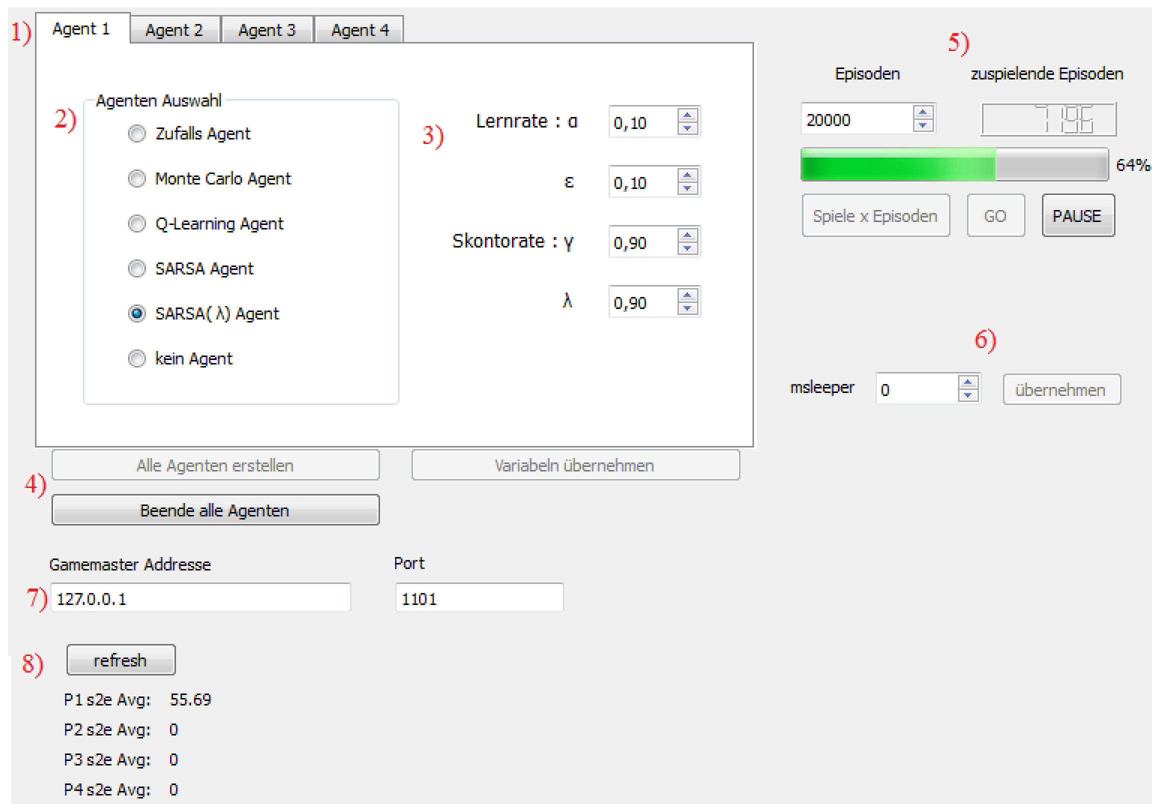

Abbildung 4.14: GUI der Agenten

Die in Abbildung 4.14 gezeigte GUI bietet im linken oberen Bereich bei 1) die Möglichkeit, für jeden der vier Agenten Einstellungen vorzunehmen. Dabei kann die Art des Agenten ausgewählt (2)) und die Parameter des Algorithmus (3)) festgelegt werden.

Darunter bei 4) befinden sich drei Knöpfe, um alle unter 1) gewählten Agenten zu erstellen, die unter 3) festgelegten Parameter auch bei laufenden Agenten zu übernehmen und alle Agenten endgültig zu beenden.

Die GUI stellt unter 5) die Steuerung für den Ablauf von Episoden bereit. In einem Eingabefeld kann die Anzahl an Episoden eingegeben werden, die alle Agenten spielen sollen. Das Betätigen des „Spiele x Episoden" Knopfs überträgt die Eingabe auf die Agenten. Mit „GO" und „PAUSE" können alle Agenten gestartet und jederzeit pausiert werden. Die restlichen Elemente in diesem Bereich stellen den Fortschritt der Agenten dar.

Pausieren die Agenten, kann bei 6) eine Zahl msleeper eingegeben werden. Die Agenten werden fortan bei jeder Aktionsauswahl msleeper Millisekunden abwarten. Dies dient dazu, um ein Spiel beobachten zu können. Beim Lernen ist hier der Wert Null einzutragen, um die Agenten auf Höchstgeschwindigkeit arbeiten zu lassen.

Unter 7) kann Adresse und Port des Spielleiters eingestellt werden, die Standardeinstellung ist das loopback-Interface.

Im unteren Bereich bei 8) wird beim Betätigen des „refresh" Knopfes die durchschnittlich pro Episode benötigten Züge je Agent angezeigt. Der Durchschnitt gilt für alle Episoden seit dem letzten betätigen des „GO" Knopfes.

Kapitel 5

Auswertung

In diesem Kapitel werden die in Kapitel 4 entwickelten Agenten untersucht. In Abschnitt 5.2 wird das Lernverhalten anhand einiger Tests analysiert, dabei werden zunächst die Testergebnisse dargelegt und anschließend erläutert.

Alle in den folgenden Test erlangten Werte sind in Beziehung zum verwendeten Testsystem zusehen. Dabei konnten, aufgrund der Limitation durch den bei den Tests verwendeten Computer, die Agenten nur bis zu einer Gedächtnisgröße von zwei Gigabyte getestet werden.

Die genaue Konfiguration des Testcomputers kann dem Anhang B entnommen werden.

5.1 Implementationsaufwand

Da alle Agenten von der agent Klasse abgeleitet wurden, stimmen sie größtenteils überein. Sie unterscheiden sich überwiegend nur in den Implementationen ihrer do_XX_action() Methoden. Deshalb werden im Folgenden ausschließlich die Unterschiede der Agenten betrachtet.

Den geringsten Implementationsaufwand benötigte der Sarsa Agent. Seine Implementation lässt sich mit wenigen Zeilen Code in der Aktionsauswahl Methode do_AA_action() realisieren.

Im Hinblick auf den Implementationsaufwand sind der Sarsa Agent und der Q-Learning Agent nahezu gleich. Der Q-Learning Agent benötigt lediglich zusätzlich noch eine Schleife um $max_{a'}Q(s', a')$ zu bestimmen.

Geringfügig mehr Aufwand benötigte der Monte Carlo Agent. Abgesehen von dem Anlegen einer Liste für die Zustandsaktionspaare der laufenden Episode und einer Map für die Mächtigkeiten war für seine Implementation eine Schleife am Ende der Episode nötig.

Der Sarsa(λ) Algorithmus stellt den komplexesten und damit etwas aufwändiger zu programmierenden hier verwendeten reinforcement learning Algorithmus dar. Seine Implementation ist aufwendiger als die der anderen Algorithmen, wenn auch nicht viel. Der Sarsa(λ) Agent implementiert das Gleiche wie auch der Sarsa Agent. Zusätzlich muss jedoch noch eine Liste für die Zustandsaktionspaare der laufenden Episode und eine Schleife, um diese Liste zu durchlaufen, erstellt werden.

5.2 Lernverhalten

Die Agenten werden in diesem Abschnitt einzeln auf ihre Lerngeschwindigkeit untersucht. Hierzu wird ein Agent jedes Algorithmus zunächst einzeln getestet. Dieser Test gestaltet sich dabei, wie folgt:

Es wird ein konstanter Startzustand mit dem Pseudozufallsgenerator für die Startzustände erzeugt, indem ein konstanter integer Wert übergeben wird. Jeder Agent wird nun als einziger Spieler das Irrgartenspiel eine Anzahl von Episoden spielen. Er muss eine Folge von zwei zufällig ausgewählten Zielen erreichen. Dieser Vorgang wird mehrere Male mit einer verschiedenen Anzahl von Episoden wiederholt. Dabei wird beobachtet, wie viel Zeit er dafür benötigt, wie viele Züge er durchschnittlich pro Episode braucht und wie viel Speicher er dabei belegt. Hierbei gilt für die Agenten $\varepsilon = 0.1$, $\alpha = 0.1$, $\gamma = 0.9$ und $\lambda = 0.9$.

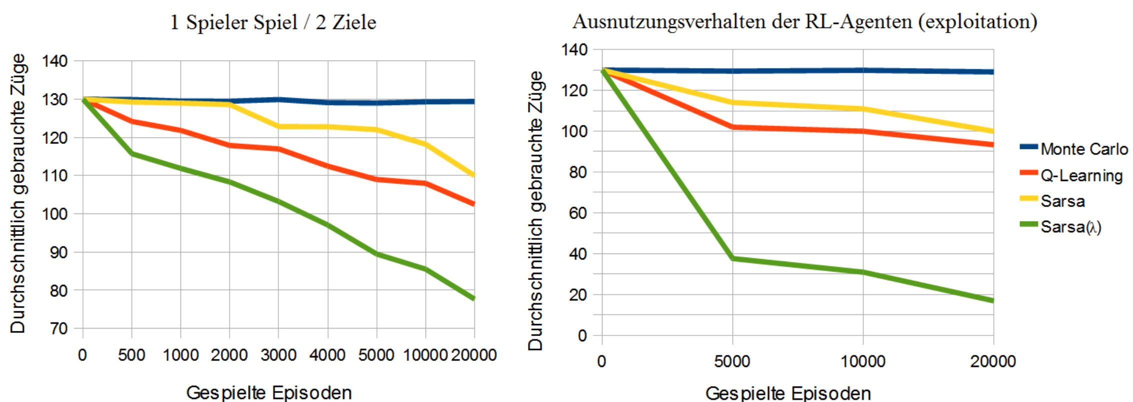

Abbildung 5.1: Lernverhalten einzeln spielender Agenten

Abbildung 5.1 zeigt links den Messverlauf über 20000 Episoden. Der Abstand der Messpunkte steigt dabei mit steigender Episodenzahl an. Jeder der Messpunkte stellt die durchschnittliche Anzahl an Zügen dar, die zum Absolvieren der Episoden benötigt wurde. Hierfür wird das arithmetische Mittel aus allen Episoden seit dem letzten Messpunkt gebildet (z.B. für den Wert bei 4000 Episoden wird der Mittelwert aus den Episoden 3001-4000 gebildet). Alle Agenten beginnen bei einem Durchschnitt von 130 Zügen, dieser Wert entspricht dem Durchschnitt eines Zufallsagenten.

In Abbildung 5.1 wurde rechts nach 5000, 10000 und 20000 Episoden eine Messung über 1000 Episoden mit $\alpha = 0$ und $\varepsilon = 0$ durchgeführt. Es wird damit die Geschwindigkeit des Agenten bei einem rein ausnutzendem Verhalten (exploitation) ohne weiteres Lernen getestet.

Wie zuerkennen ist, besteht die Tendenz, dass sich sich alle TD-Agenten dabei weiter mit steigender Episodenzahl verbessern. Der Sarsa(λ) Agent schneidet hierbei mit 78 Zügen bei dem 20000 Episoden Messpunkt am besten ab, gefolgt vom Q-Learning Agenten mit 102 und dem Sarsa Agenten mit 110. Der Monte Carlo Agent verbleibt bis zum Ende der Messung auf dem Niveau eines Zufallsagenten. Bei dem ausnutzenden Test (exploitation) zeigt sich, dass der Sarsa(λ) Agent schon sehr früh auf erheblich bessere Lösungen kommt als die anderen TD-Algorithmen, die sich jedoch ebenfalls verbessern. Der Monte Carlo Agent zeigt auch hier keine Lernerfolge.

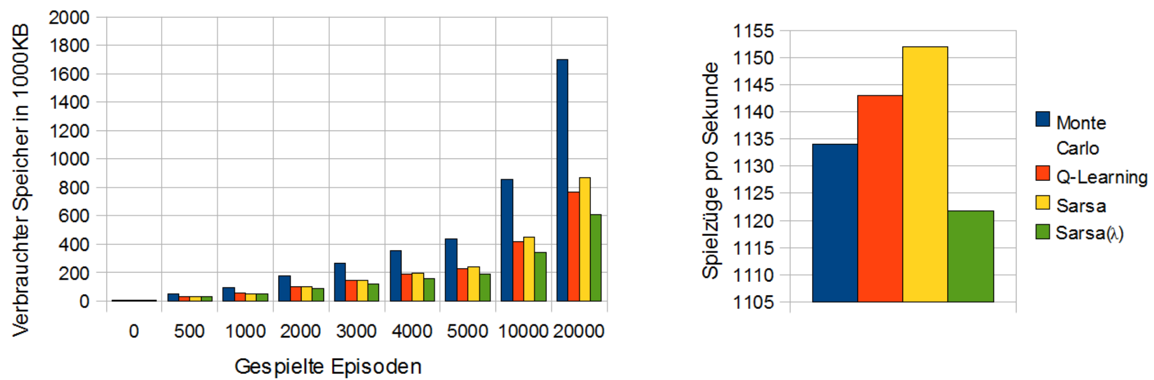

Abbildung 5.2: Speicherverbrauchs- (links) und Ausführungsgeschwindigkeitstest (rechts)

Während des zuvor beschriebenen Testablaufs wurden an allen Messpunkten ebenfalls der gesamte Speicherverbrauch der Agenten gemessen. Abbildung 5.2 zeigt links die Ergebnisse dieser Messung.

Die Agent belegen vor dem Spielbeginn etwa 6000 KB, da im Spielverlauf die Agenten nur Daten in Form ihres Gedächtnisses anhäufen, kann davon ausgegangen werden, dass der gemessene Speicherverbrauch, minus den genannten 6000 KB, dem Umfang des Gedächtnisses entspricht. Ausgenommen hiervon ist der Monte Carlo Agent, der wie in Abschnitt 4.2.3.2 beschrieben, noch eine Map über die Mächtigkeiten der ersetzten Listen hält, die zusätzlich Speicher belegt.

Nach 20000 Episoden ist zu sehen, dass die erfolgreicher spielenden Agenten weniger Speicher verbrauchen als die weniger erfolgreichen. Die erfolgreicheren bilden weniger Gedächtniseinträge, da sie öfter auf schon vorhandenes Wissen zurückgreifen. Da die Gedächtniseinträge aller Agenten gleich aufgebaut sind, lässt sich aus dem Speicherverbrauch direkt auf die Anzahl an Gedächtniseinträgen schließen. Der Monte Carlo Agent belegt erheblich mehr Speicher als die anderen Algorithmen. Dies ist zum einen auf den geringeren Spielerfolg zurückzuführen, zum anderen großteils darauf, dass die zusätzliche Map geführt werden muss.

Für die Ermittlung der Spielzüge pro Sekunde der Agenten sind immer jeweils zwei Agenten des gleichen Typs in mehreren Spielen gegeneinander angetreten. Dabei wurden die Anzahl der ausgeführten Züge gemessen und auf die Gesamtdauer bezogen. In Abbildung 5.2 sind rechts die zugehörigen Ergebnisse gezeigt.

Der Unterschied in der Ausführungsgeschwindigkeit der Agenten spiegelt direkt die Komplexität der jeweiligen Berechnungen wieder. Der Sarsa Agent stellt nur eine simple Berechnung an, der Q-Learning Agent muss zusätzlich noch $max_{a'}Q(s', a')$ bestimmen, daher haben diese die höchsten Ausführungsgeschwindigkeit. Der Monte Carlo Agent verbraucht innerhalb einer Episode kaum Zeit, führt jedoch zwischen den Episoden eine Reihe von Berechnungen und Speicherzugriffen aus. Der Sarsa(λ) Agent stellt für eine ganze Reihe von Zustandsaktionspaaren dieselbe Berechnung wie der Sarsa Agent an, was ihn zum langsamsten Agenten bei der Ausführung macht. Die Unterschiede in der Ausführungsgeschwindigkeit sind jedoch relativ gering, da die Agenten ihre Berechnungen, soweit möglich, während der Züge der anderen Spieler tätigen.

Es war des weiteren zu beobachten, dass mit zunehmender Episodenzahl die Ausführungsdauer eines Zuges sich nicht erhöht und die einer Episode zurückgeht. Dies ist darauf zurückzuführen, dass die Episoden in immer weniger Zügen absolviert werden

und durch die Organisation des Gedächtnisses in einer HashMap, diese ist im Normalfall nahezu unabhängig von der Gedächtnisgröße und bietet eine konstante Zugriffsgeschwindigkeit.

5.3 Spielerfolg

Um den Spielerfolg zu bestimmen, werden auf verschiedenen Algorithmen basierende Agenten gegeneinander antreten. Dabei wird mit ungelernten Agenten gestartet. Immer nach einer festgelegten Anzahl von Episoden wird überprüft, welcher Agent wie oft gewonnen hat.

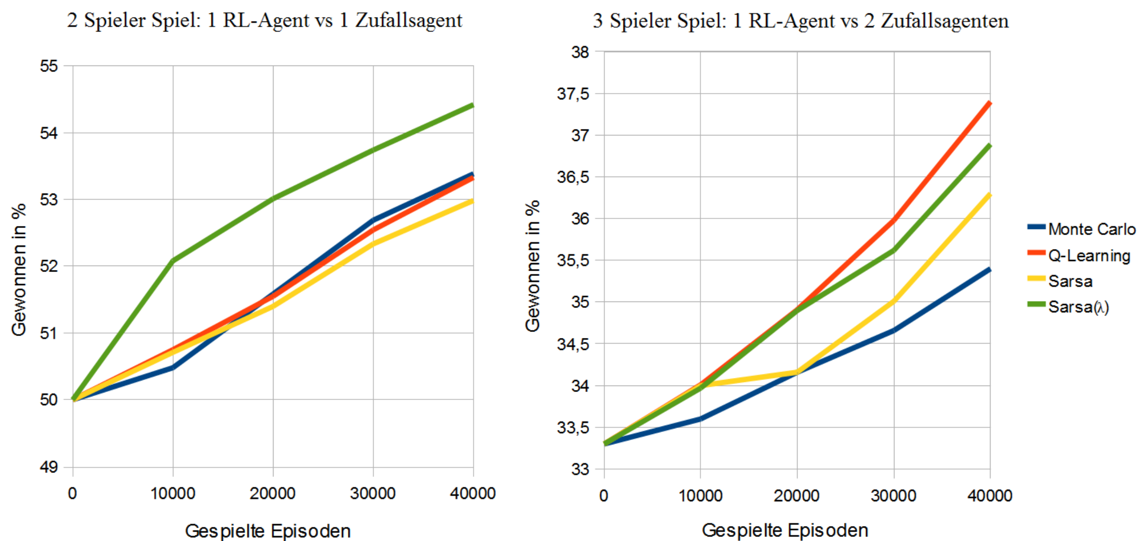

Abbildung 5.3: Mehrspieler-Test gegen Zufallsagenten

Zunächst wird jede Art von Agent in einem 2 Spieler Spiel gegen einen Zufallsagenten antreten, und anschließend in einem 3 Spieler Spiel gegen 2 Zufallsagenten. Das Spiel wird dabei so aufgebaut, wie in Abschnitt 5.2. In Abbildung 5.3 ist zu sehen, wie viele Episoden (%) die einzelnen Agenten gegen die Zufallsagenten seit dem letzten Messpunkt gewonnen haben.

Gegen einen Agenten schneidet, ähnlich wie in Abschnitt 5.2, dabei der Sarsa(λ) Agent am besten ab. Der Monte Carlo Agent gleicht sich nach einem schwächeren Start dem Q-Learning Agenten an, der abermals leicht besser abschneidet als der Sarsa Agent. Im Spiel gegen 2 Zufallsagenten schneidet jedoch der Q-Learning Agent am besten ab. Der Monte Carlo Agent fällt hinter dem Sarsa Agenten zurück.

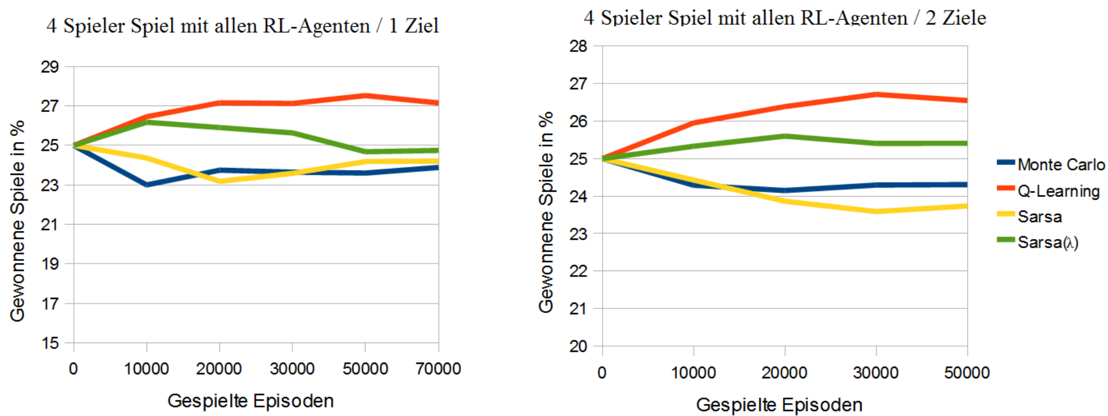

Abbildung 5.4: Mehrspieler-Test jeder gegen jeden

Um die Agenten direkt miteinander zu vergleichen, treten diese in einem 4 Spieler Spiel gegeneinander an. In Abbildung 5.4 ist dargestellt, wie viele Episoden (%) die einzelnen Agenten gegeneinander seit dem letzten Messpunkt gewonnen haben. Links wurde dafür jedem Spieler nur ein zufälliges Ziel pro Episode zugeteilt, rechts zwei Ziele.

Es ist zuerkennen, dass der Q-Learning Agent sich bei beiden Tests von den anderen Agent abhebt. Der Sarsa und der Sarsa(λ) Agent liegen dahinter. Bei weniger Zielen ist der Unterschied zwischen den beiden Agenten geringer als bei mehreren Zielen, wobei sich der Sarsa(λ) Agent durchgehend vom Sarsa Agenten absetzt. Der Monte Carlo Agent liegt bei nur einem Ziel hinter dem Sarsa Agenten, übertrifft diesen jedoch bei mehreren Zielen.

5.4 Auswertende Betrachtung

Wie in Abschnitt 5.2 deutlich wird, schneidet der Sarsa(λ) durchgehend besser ab als die anderen Agenten, dabei ist jedoch zu beachten, dass dieser erste Test eine nahezu voll deterministische Umwelt testet, da keine Mitspieler ins Spielgeschehen eingreifen. Somit kann der Sarsa(λ) Algorithmus, der immer ganze Pfade in einem lernt, seine Stärken voll ausspielen, da ein einmal gefundener Pfad immer wieder komplett und ohne Risiken genutzt werden kann.

Das für den Monte Carlo Agenten in Abschnitt 5.2 dargestellte Lernverhalten ist für den Monte Carlo Agenten nicht repräsentativ, da bei einem Spiel ohne Mitspieler jede Runde gewonnen wird und er somit immer die gleiche Belohnung erhält. Die Folge daraus ist, dass allen durchlaufenen Zustandsaktionspaaren immer die selben Werte zugeordnet werden, unabhängig von der Anzahl der benötigten Spielzügen des Agenten. Wodurch alle Züge gleich gut für ihn erscheinen.

Aus den Tests geht hervor, dass der Q-Learning Agent in allen gewählten Spielvariationen dem Sarsa Agenten überlegen ist. Da diese sich nur in der Schätzwertbestimmung des Folgezustandes unterscheiden, ist daraus zu schließen, dass für das gewählte Spiel die strategiefreie Schätzwertbestimmung sich besser eignet als die strategiebasierte.

Bei zunehmender Spielerzahl büßt der Sarsa(λ) Agent seinen Vorteil gegenüber dem Q-Learning Agenten ein. Auch wenn er bei einem einzelnen Gegenspieler noch einen deutlichen Vorteil genießt, so verliert er diesen bei abnehmendem Determinismus der

Umwelt durch Mitspieler, wie sich aus den Abbildungen 5.3 und 5.4 schließen lässt. Durch die Spielteilnahme von drei weiteren Spielern ist der Sarsa(λ) Agent im getesteten Rahmen nur noch teilweise in der Lage gelernten Pfaden zu folgen. Dadurch reduziert sich sich dessen Effektivität und nähert sich der des Sarsa Agenten mit dem Unterschied, dass der Sarsa(λ) Agent dennoch schneller Erfahrungen dazugewinnt als der Sarsa Agent und sich somit leicht von diesem abhebt.

Es erscheint als würde der Q-Learning Agent mit abnehmendem Determinismus besser werden. Wahrscheinlicher ist jedoch, dass er deshalb den Sarsa(λ) Agenten übertrifft, da dessen Effektivität mit abnehmendem Determinismus sinkt und sich dem Sarsa Agenten angleicht, der durch den Q-Learning Agent übertroffen wird.

Mit mehreren Spielern ist der Monte Carlo Agent in der Lage aus Sieg und Niederlage zu lernen und somit sein Spielverhalten zu verbessern. Je nach Testszenario ordnet sich der Monte Carlo Agent auf dem letzten Platz oder im Mittelfeld der Agenten ein. Es war während der Tests auch noch zu beobachten, dass das Lernverhalten des Monte Carlo Agenten stärker von der Auswahl der mitspielenden Agenten beeinflusst wurde. (siehe auch[MM11])

Kapitel 6

Fazit und Ausblick

Diese Arbeit dokumentiert den Entwicklungsprozess von reinforcement Learning Agenten für Strategie-Gesellschaftsspiele.

Ziel dieser Arbeit war, möglichst erfolgreich spielende Agenten zu entwickeln und deren Verhalten miteinander zu vergleichen. Für die Entwicklung wurde ein fiktives Irrgartenspiel herangezogen, das die Merkmale von Strategie-Gesellschaftsspielen in sich vereinigt.

Die Modellierung der Zustände und die damit verbundene Analyse der Umwelt stellt sich als erster und einer der wichtigsten Schritte bei der Entwicklung heraus. Die hier gewählte Modellierung beeinflussen nachhaltig den Erfolg und das Lernverhalten der Agenten.

Die strikte Trennung von Agent und Umwelt kann sowohl der Agent ohne Probleme auf andere Aufgaben angewandt werden als auch ungewollte Effekten, die durch den direkten Zugriffe des Agenten auf die Umwelt entstehen, vorgebeugt werden. Bei der Trennung ist jedoch auf die Wahl des Kommunikationskonzeptes zu achten, da dies sich zwar nicht auf den Erfolg der Agenten, wohl aber auf die Ausführungsgeschwindigkeit auswirkt. Je nach Anwendungsfall sollte auf schnellere Kommunikationsarten zurückgegriffen werden, da langsame Kommunikation die Ausführungsgeschwindigkeit aufgrund der vielen Ausführungen extrem beeinflusst. Dabei spielt vor allem die Nachrichtenlatenz eine Rolle.

Nachdem ein allgemeiner Agent bzw. ein Agenten-Framework geschaffen wurde, ist die Implementation eines einzelnen reinforcement learning Algorithmus vergleichsweise schnell und ohne größeren Aufwand möglich. Werden viele Tests ausgeführt, erweist sich die Entwicklung einer Testumgebung durch eine GUI im Hinblick auf die Durchführung der Tests als auch im Bezug auf das Auslesen der durch den Test gewonnen Daten als hilfreich.

Um den für ein Gesellschafts-Strategiespiel am besten geeigneten Agenten, aus den in dieser Arbeit vorgestellten Agenten, auszuwählen, sind mehrere Faktoren mit einzubeziehen. Die Tests deuten darauf hin, dass es darauf an kommt, wie sehr das Spiel durch die Mitspieler beeinflusst wird. Dabei gilt, dass mehr Mitspieler zu einem stärker beeinflussten Spielzustand führen. Wird das Spiel weniger beeinflusst, so erzielt der Sarsa(λ) Agent die besten Ergebnisse. Beeinflussen die Mitspieler den Spielzustand stärker, so übertrifft der Q-Learning Agent den Sarsa(λ) Agenten.

Die Ausführungsgeschwindigkeiten der verschiedenen Agenten sind bei der Agentenauswahl kein ausschlaggebender Faktoren. Durch eine gut geplante Implementation

kann ein Großteil der Berechnungen der Agenten parallel zu den Zügen anderer Spieler durchgeführt werden, womit alle Agenten in etwa die gleiche Geschwindigkeit aufweisen. Der Speicherverbrauch ist bei allen in etwa gleich groß bzw. hängt vom Spielerfolg ab.

Eine mögliche Weiterentwicklung dieser Arbeit stellt die Implementation eines Q(λ) Agenten dar. Dieser verbindet die Q-Wertberechnung des Q-Learning mit dem Pfadlernen von Sarsa(λ) und hat somit die Möglichkeit, sowohl bei geringer als auch bei hoher Beeinflussung durch die Mitspieler, bessere Ergebnisse zu erzielen als der Q-Learning und Sarsa(λ) Agent.

Darüber hinaus gibt es eine Vielzahl von Ansätzen:

Erweiterung der Speicherkapazität und Systemumgebung:

Die entwickelte Software wurde als 32-Bit-Anwendung kompiliert. Dies hatte zur Folge, dass die maximale Hauptspeichernutzung für einen einzelnen Agenten unter Windows im Allgemeinfall auf zwei Gigabyte beschränkt war. Da das Testsystem jedoch lediglich über vier Gigabyte verfügte und mehrere Agenten gleichzeitig betrieben wurden, spielte diese Grenze in dieser Arbeit noch keine Rolle. Mit mehr Speicher bietet sich an zukünftig das Irrgartenspiel bei voller Komplexität zu testen: zufällige Erstellung von Startzuständen; Zuweisung von drei zufälligen Zielen pro Episode und Spieler; bei vier Spielern.

Für das Testen auf einer stärkeren Testumgebung mit mehr Arbeitsspeicher wird die Umsetzung als 64-Bit-Anwendung nötig, wofür eine Kompilation mit den 64-Bit-Qt-Bibliotheken nötig ist. Eine andere Möglichkeit bietet die direkte Auslagerung auf eine Festplatte. Voraussetzung hierfür ist jedoch schneller Datenaustausch, wie dies u.a. bei SSD-Festplatten gegeben ist.

Einführung einer Abstraktionsebene:

Die Implementierung einer Abstraktionsebene, um ähnliche Zustände zu erkennen, um somit sowohl die Lerngeschwindigkeit zu erhöhen als auch den Speicherbedarf zu reduzieren.

Agenten mit Modell:

Ein Agent mit Modell, wie in Abschnitt 4.1.5 angerissen. Dieser könnte besonders bei dem erfolgreichen Q-Learning Agenten dazu beitraten, um gesammelte Erfahrungen schneller zu verbreiten. Ein Modell kann sich als besonders nützlich bei Strategie-Gesellschaftsspielen erweisen, bei denen die Möglichkeiten, echte Erfahrungen zu sammeln, begrenzt ist. Oder die Verwendung eines distribution Modells, wie in Abschnitt 4.1.5 erwähnt.

Literaturverzeichnis

[Alp08] ALPAYDIN, Ethem ; LINKE, Simone (Hrsg.): *Maschinelles Lernen*. München, Oldenbourg Wissenschaftsverlag GmbH, 2008. – XVIII, 440 S. – ISBN 3-486-58114-7

[Doc] DOCUMENT, Foundation: *LibreOffice - Calc* online:http://de.libreoffice.org/product/calc/. – (Abrufdatum 2012.10.20)

[Mic] MICROSOFT: *Visual Studio* online:http://msdn.microsoft.com/de-de/vstudio. – (Abrufdatum 2012.10.20)

[MM11] MARCOLINO, Leandro S. ; MATSUBARA, Hitoshi: *Multi-Agent Monte Carlo Go*. 2011 online:http://teamcore.usc.edu/people/sorianom/AAMAS2011.pdf. – (Abrufdatum 2012.10.15)

[Noka] NOKIA, Corp.: *Algorithmic Complexity* online:http://doc.qt.digia.com/qt/containers.html#algorithmic-complexity. – (Abrufdatum 2012.10.19)

[Nokb] NOKIA, Corp.: *QHash Class Reference* http://doc.qt.digia.com/qt/qhash.html. – (Abrufdatum 2012.10.19)

[Nokc] NOKIA, Corp.: *Qt Developer Network* online:http://qt-project.org/downloads. – (Abrufdatum 2012.10.20)

[Nokd] NOKIA, Corp.: *Qt Reference Documentation* online:http://doc.qt.digia.com/qt/. – (Abrufdatum 2012.10.20)

[Noke] NOKIA, Corp.: *Qt VS Add-in 1.1.9* online:http://blog.qt.digia.com/2011/03/30/qt-vs-add-in-1-1-9-released/. – (Abrufdatum 2012.10.20)

[SB05] SUTTON, Richard S. ; BARTO, Andrew G.: *Reinforcement Learning: An Introduction*. Cambridge, Mass. : MIT Press, 2005 (Adaptive computation and machine learning). – ISBN 0-262-19398-1

[Sut90] SUTTON, Richard S.: Integrated Architectures for Learning, Planning, and Reacting Based on Approximating Dynamic Programming. In: *In Proceedings of the Seventh International Conference on Machine Learning*, Morgan Kaufmann, 1990, S. 216–224

[Teu95] TEUBER, Klaus: *Siedler von Catan Basis Spielregel*. Kosmos, 1995 online:www.catan.de/files/downloads/Siedler_von_Catan-Basis-3_4-Spielregel.pdf. – (Abrufdatum 2012.10.20)

[Vis] VISUALPARADIGM, Corp.: *Visual Paradigm for UML 10* online:http://www.visual-paradigm.com/product/vpuml/. – (Abrufdatum 2012.10.20)

Anhang A

Software Elemente

Methoden Erklärung: Methoden der Spielleiter-Spielinstanz Schnittstelle

Der Gamemaster nutzt, um die Spielinstanz zu verwalten, die von der labyrinth Klasse bereitgestellten Methoden:

▷ `QString labyrinth::getGamestateID(int playerNr);`

Der von dieser Methode gelieferte QString, eine Qt Stringklasse , repräsentiert den Zustand des Spiels. Jeder char des Strings stellt ein Spielquadrate oder eine Position eines Spielers bzw. des Ziels dar.

Ein Beispiel für solch einen String ist:

```
:556333:<5659<53D626G
```

Die ersten 16 chars stehen für die beweglichen Spielquadrate, gefolgt von einem char für die Position des derzeitigen Spielers, einem für die Position seines Ziels und drei chars für die Positionen der anderen Spieler.

▷ `list<square*> * labyrinth::getReachableSquares(void);`

▷ `list<pushAction*> * labyrinth::getPushActions(void);`

Diese beiden get Methoden werden für die Bestimmung, der vom derzeitigen Spieler ausführbaren Aktionen, genutzt. Über die Aufrufe:

```
int square::getPosID(void);
```

```
int pushAction::getID(void);
```

werden die AktionsIDs erlangt, die für die Bewegungsaktionen zu den erreichbaren Spielquadraten und die möglichen Einschiebeaktionen stehen.

▷ `bool labyrinth::performPushAction(int actionID);`

▷ `bool labyrinth::performMoveAction(int actionID);`

Die perform Methoden veranlassen die Spielinstanz die Aktion, die zu der übergebenen AktionsID gehört, für den derzeitigen Spieler auszuführen. Der Rückgabewert sagt, ob die Aktion erfolgreich ausgeführt wurde. Wird etwa eine regelwidrige Aktion ausgewählt, ist der Rückgabewert gleich false.

▷
```
bool labyrinth::checklabydone(void);
```

Die Methode checklabydone() überprüft, ob einer der Spieler keine weiteren Ziele mehr hat. Ist der Rückgabewert gleich true, so ist das Spiel zu Ende.

Der Gamemaster implementiert ebenfalls Methoden für die Verwaltung:

▷
```
int Gamemaster::checkReward(int playerNr);
```

Um zu überprüft, ob sich der Spieler mit der übergebenen Spielernummer eine Belohnung verdient hat und in welcher Höhe, greift die checkReward() Methode direkt auf die Spielinstanz zu. Diese Methode ruft ebenfalls die checklabydone() Methode auf und kann so ebenfalls das Ende einer Episode feststellen. Im Fall, dass die Episode zu Ende ist, wird ein negativer Wert zurückgegeben, da in diesem Fall, eine andere Belohnung durch den Spielleiter verteilt wird.

▷
```
void Gamemaster::startNewEpisode(void);
```

Durch den Aufruf dieser Methode wird die Spielinstanz zurückgesetzt, so dass eine neue Episode gestartet werden kann. Genauer gesagt, wird die Spielinstanz gelöscht und eine neue erstellt.

Methoden Erklärung: Methoden der Kommunikationsschnittstelle des Gamemaster

Für die Realisierung der Kommunikation implementiert der Gamemaster ein char array als Eingangsbuffer (buffer_in) und eines als Ausgangsbuffer (buffer_out). Mit den Aufrufen:

▷
```
int Gamemaster::rcvFromPlayer(int playerNr);
```

▷
```
int Gamemaster::sendToPlayer(int playerNr);
```

wartet er auf eine Nachricht des Spielers dessen Spielernummer übergeben wurde und speichert sie im buffer_in bzw. sendet die Nachricht, die sich in buffer_out befindet an diesen Spieler. Der Rückgabewert ist die Anzahl an empfangen bzw. gesendeten Bytes.

In allen folgenden rcv...() und send...() Methoden des Gamemaster werden sie in Verbindung mit dem anschließenden Lesen aus buffer_in oder dem vorherigen Schreiben nach buffer_out verwendet.

Um die Kommunikation mit dem Agenten vorzubereiten und durchzuführen, implementiert der Gamemaster eine Reihe von Methoden:

▷
```
bool Gamemaster::initIPC();
```

InitIPC() erstellt den host socket für die IPC, mit dem sich später die Agenten verbinden. Ist die Initialisierung fehlgeschlagen, ist der Rückgabewert gleich false.

▷
```
void Gamemaster::server_accept_connects();
```

In der server_accept_connects() Methode akzeptiert der Gamemaster neue Verbindungen von Agenten, bis sich ausreichend Agenten für das Spiel angemeldet haben. Für ein Spiel mit z.B. vier Spielern wird gewartet bis sich viel Agenten verbunden haben.

▷
```
void Gamemaster::sendAvailableMoveActions(int playerNr);
```

▷
```
void Gamemaster::sendAvailablePushActions(int playerNr);
```

Jede dieser beiden Methoden stellt eine Liste mit verfügbaren Bewegungs- bzw. Einschiebeaktionen zusammen und schickt sie anschließend an den Spieler dessen Spielernummer übergeben wurde. Sie nutzen die von der Spielinstanz bereitgestellten Methoden getReachableSquares() bzw. getPushActions().

▷
```
void Gamemaster::sendStateReward(int playerNr, int reward)
    ;
```

In dieser send Methode wird über den Aufruf getGamestateID() der Spielzustand ermittelt und mit dem übergebenen Belohnungswert reward an den Agenten geschickt.

▷
```
int Gamemaster::rcvDoAction(int playerNr);
```

Die Methode rcvDoAction() empfängt eine Nachricht, extrahiert daraus die mitgeschickte AktionsID und gibt diese als Rückgabewert wieder.

Methoden Erklärung: Methoden der Kommunikationsschnittstelle der Agenten

▷
```
bool agent::ipc_init(void);
```

Die ipc_init() erstellt einen Socket für die IPC mit dem Spielleiter. Der Rückgabewert ist bei Erfolg ture.

▷
```
bool agent::connect2gm(void);
```

Mit dem Aufruf der connect2gm() verbindet sich der Agent mit dem Spielleiter. Der Rückgabewert ist bei Erfolg ture.

▷
```
int agent::rcvToBuffer(void);
```

▷
```
int agent::sendFromBuffer(void);
```

Durch den Aufruf dieser Methoden wird auf eine Nachricht gewartet und in den buffer_in geschrieben bzw. eine Nachricht aus dem buffer_out zu dem Spielleiter geschickt. Der Rückgabewert entspricht der Anzahl an empfangen bzw. gesendeten Bytes.

▷
```
int agent::sendACK(void);
```

Die sendACK() Methode schreibt die ACK-Nachricht in den buffer_out und ruft anschließend die sendFromBuffer() auf. Der Rückgabewert entspricht der Anzahl gesendeten bytes.

▷
```
int agent::getMsgType(void);
```

Der Aufruf von getMsgType() überprüft die ersten chars der Nachricht im buffer_in und gibt dem Nachrichtentyp entsprechend einen repräsentierenden integer zurück.

Methoden Erklärung: Unterstützende Agentenmethoden

Für die do_AA_action() Methoden stellt die Agent Oberklasse noch Methoden zur Verfügung:

▷
```
void agent::retrievStateRewardFromBuffer(void);
```

Diese Methode extrahiert den Zustand und die Belohnung aus einer SR-Nachricht und speichert sie in den globalen Variablen cur_state und new_reward.

▷
```
void agent::retrievActionsFromBuffer(void);
```

Diese Methode extrahiert den alle AktionsIDs aus einer AA-Nachricht und speichert sie in der globalen QList actions.

Code Erklärung: Run Methode des Gamemaster

Bei dem hier abgebildeten Code handelt es sich um eine, zugunsten der Übersicht gekürzte, Fassung. Der Bereich zwischen „Beginn einer Episode" und „Ende einer Episode" wird im Anschluss gesondert in Abbildung 4.10 dargestellt.

Zeilen mit '...' repräsentieren heraus gekürzte Codefragmente, sie enthalten zumeist variablen Deklarationen, Sicherheitsabfragen und Logeinträge oder ähnliches. NoP entspricht der Spielerzahl und bei csocks[] handelt es sich um ein Array das die clientsockets für die Kommunikation enthält.

```
1  void Gamemaster::run(){
2      ...
3      initIPC();
4      ...
5      server_accept_connects();
6      ...
7      /** Hauptschleife */
8      while(!stopped){
9          if(reconnect){
10             for(int i = 0; i < NoP; i++){
11                 free(csocks[i]);
12             }
13             server_accept_connects();
14             reconnect = false;
15         }
16         ...
17         /** Sende NE-Nachricht an alle Spieler */
18         for(int p = 1; p < NoP+1; p++){
19             sprintf(buffer_out,"%s", NEW_EPISODE_MSG);
20             sendToPlayer(p);
21             rcvFromPlayer(p);
22             ...
23         }
24         /** Beginn einer Episode */
25         ...
26         /** Ende einer Episode */
27         /** Sende EE-Nachricht an alle Spieler */
28         for(int p = 1; p < NoP+1; p++){
29             sprintf(buffer_out,"%s", END_EPISODE_MSG);
30             sendToPlayer(p);
31             rcvFromPlayer(p);
32             ...
33         }
34         ...
35         startNewEpisode();
36     }
37 }
```

Abbildung A.1: Code: Gekürzter run loop der Gamemaster Klasse

Code: Episoden Code des Gamemaster

```
/** Beginn einer Episode */
while(!endOfEpisode){
    playerNr = laby->getCurrendPlayerNr();
    ...
    /** Phase 1: Bewegung */
    reward = checkReward(playerNr);
    if(reward < 0){
        endOfEpisode = true; break;
    }
    ...
    sendStateReward(playerNr,reward);
    /** Empfangen des ACK */
    rcvFromPlayer(playerNr);
    ...
    sendAvailablePushActions(playerNr);
    do_actionID = rcvDoAction(playerNr);
    ...
    laby->performPushAction(do_actionID);
    ...
    /** Phase 2: Einschieben */
    reward = checkReward(playerNr);
    if(reward < 0){
        endOfEpisode = true; break;
    }
    ...
    sendStateReward(playerNr,reward);
    /** Empfangen des ACK */
    rcvFromPlayer(playerNr);
    ...
    sendAvailableMoveActions(playerNr);
    do_actionID = rcvDoAction(playerNr);
    ...
    laby->performMoveAction(do_actionID);
    ...
    /** Überprüfen auf Spielende */
    if(laby->checklabydone()){
        endOfEpisode = true; break;
    }
    ...
    laby->incrementPlayer();
    ...
}
...
reward = TARGET_REACHED_REWARD*2;
//send final state +reward
sendStateReward(playerNr,reward);
/** Empfangen des ACK */
rcvFromPlayer(playerNr);
...
/** Ende einer Episode */
```

Code: Run Methode aller Agenten

```
void agent::run(){
    if(!ipc_init()){
        closesocket(hsock);
        return;
    }
    while(!stopped){
        while(paused || episode_cnt >= episodes){
            if(stopped){
                closesocket(hsock);
                return;
            }
            this->msleep(100);
        }
        int msg_type = 0;
        rcvToBuffer();
        msg_type = getMsgType();
        switch(msg_type){
            case NE_ACTION : {
                do_NE_action(); break;
            }
            case SR_ACTION : {
                do_SR_action(); break;
            }
            case AA_ACTION : {
                do_AA_action(); break;
            }
            case EE_ACTION : {
                do_EE_action(); break;
            }
            default :
                do_default_action(); break;
        }
    }
}
```

Abbildung A.3: Code: run loop der Agenten

Code: Greedy-Suche

In der ersten Schleife (Z.10-25) wird eine Liste namens greedy erstellt, in der sich alle Aktionen, die sich den höchsten Q-Wert teilen und ausführbar sind, befinden. Anschließend wird die actions Liste, in der sich alle verfügbaren Aktionen befinden, durch die greedy Liste ersetzt (Z.26-31) und aus ihr zufällig eine Aktion ausgewählt (Z. 35).

```
1  int agent::chooseGreedyAction(){
2      /** default Rückgabewert festlegen */
3      int action = -1;
4      float max_exp_value = -1;
5      if(memory->contains(cur_state)){
6          QList<int> greedy;
7          /** Liste der bekannten Aktionen erstellen */
8          QList<int> known_actionIDs = memory->value(cur_state)
               ->keys();
9          /** Aktionsliste durchgehen */
10         for(int i = 0; i < known_actionIDs.size(); i++){
11             /** Überprüfen ob Aktion höheren Q-Wert hat */
12             int tmp = memory->value(cur_state)->value(
                   known_actionIDs[i]);
13             if(tmp >= max_exp_value){
14                 /** überprüfen ob Aktion auch in der Liste
15                     der verfügbaren Aktionen enthalten ist*/
16                 if (!(actions->indexOf(known_actionIDs[i]) < 0)
                       ){
17                     if(tmp > max_exp_value){
18                         greedy.clear();
19                         max_exp_value = tmp;
20                         action = known_actionIDs[i];
21                     }
22                     greedy.push_front(known_actionIDs[i]);
23                 }
24             }
25         }
26         if(action >= 0){
27             actions->clear();
28             for(int i = 0; i < greedy.size(); i++){
29                 actions->push_front(greedy[i]);
30             }
31         }
32
33
34     }
35     action = chooseRndAction();
36     return action;
37 }
```

Abbildung A.4: Code: Greedy-Suche

Code: ε-Greedy-Suche

```
1  int agent::chooseEpsilonGreedyAction(){
2      int action = -1;
3      /** epsilon-greedy */
4      if(rand() % 100 < (epsilon*100)){
5          action = chooseRndAction();
6      }else{
7          action = chooseGreedyAction();
8      }
9      if(action < 0){
10         action = chooseRndAction();
11     }
12     return action;
13 }
```

Abbildung A.5: Code: ε-Greedy-Suche

Code Erklärung: Q-Learning

Die Schritte a),b) und e) aus dem Algorithmus 2.2 werden bei dem Q-Learning Agenten in der do_AA_action() Methode abgehandelt. Sie werden durch die folgenden Zeilen realisiert:

```
1  /** Auswahl der Aktion a mittels Strategie (Algo. Punkt a)) */
2  int action = chooseEpsilonGreedyAction();
3
4  /** Merken des Zustandes s und der ausgewählten Aktion a
5      in einem state_action_pair Objekt (Algo. Punkt e) */
6  priv_sap = new state_action_pair(cur_state, action);
7
8  /** Übermitteln der Aktion an den Spielleiter (Algo. Punkt b))
       */
9  sprintf(buffer_out,"%s%s%d",DO_ACTION_MSG,SEPERATOR, action);
10 sendFromBuffer();
```

Die Schritte c) und d) werden in der do_SR_action() Methode verwirklicht. Die folgenden Codefragmente realisieren dies.

```
1  /** Beobachten der Belohnung r gespeichert in Variable
      new_reward
2      und des Folgezustandes s' gespeichert in Vraiable
          cur_state
3      ( Punkt c) ) */
4  retrievStateRewardFromBuffer();
5
6  /** Aktualisieren von Q(s,a) (Punkt d) )*/
7  /** Auslesen des alten Q-Wertes aus dem Gedächtnis (Q(s,a)) */
8  q_old = memory->value(priv_sap->state)->value(priv_sap->
      actionID);
9
```

```
10 /** Bestimmen von max_a'Q(s',a') und Speichern in
      max_exp_value */
11 max_exp_value = 0;
12 if(memory->contains(cur_state)){
13     QList<int> known_actionIDs = memory->value(cur_state)->
           keys();
14     for(int i = 0; i < known_actionIDs.size(); i++){
15         int tmp = memory->value(cur_state)->value(
               known_actionIDs[i]);
16         if(tmp > max_exp_value){
17             max_exp_value = tmp;
18         }
19     }
20 }
21 /** Berechnen des neuen Q-Wertes Q(s,a) */
22 q_new = q_old + (alpha * (new_reward + (skonto * max_exp_value
      ) - q_old));
23 
24 /** speichern des neuen Q-Wertes im Gedächtnis */
25 memory->value(priv_sap->state)->insert(priv_sap->actionID,
      q_new);
```

Code Erklärung: Sarsa

Der Großteil des Sarsa Algorithmus läuft in der do_AA_action() Methode des Sarsa Agenten ab:

```
1 /** Entspricht beim ersten Aufruf von do_AA_action() in einer
     Episode:
2     der Zeile vor der while-Schleife des Algo. */
3 /** Danach: Auswahl der Aktion a' mittels Strategie (Algo.
     Punkt c)) */
4 int action = chooseEpsilonGreedyAction();
5 
6 /** priv_sap ist nur NULL bei dem ersten Aufruf von
     do_AA_action()
7     in einer Episode */
8 if(priv_sap != NULL){
9     /** Aktualisieren von Q(s,a) (Punkt d) )*/
10    /** Auslesen des alten Q-Wertes aus dem Gedächtnis (Q(s,a)
         ) */
11    q_old = memory->value(priv_sap->state)->value(priv_sap->
         actionID);
12 
13    /** Auslesen von Q(s',a') aus dem Gedächtnis*/
14    q_next = memory->value(cur_state)->value(action);
15 
16    /** Berechnen des neuen Q-Wertes Q(s,a) */
17    q_new = q_old + (alpha * (new_reward + (skonto * q_next) -
           q_old));
```

```
18
19      /** speichern des neuen Q-Wertes im Gedächtnis */
20      memory->value(priv_sap->state)->insert(priv_sap->actionID,
           q_new);
21  }
22  /** Merken des Zustandes s' und der ausgewählten Aktion a'
23      in einem state_action_pair Objekt (Algo. Punkt e) */
24  priv_sap = new state_action_pair(cur_state, action);
25
26  /** Übermitteln der Aktion an den Spielleiter (Algo. Punkt a))
       */
27  sprintf(buffer_out,"%s%s%d",DO_ACTION_MSG,SEPERATOR,action);
28  sendFromBuffer();
```

Code Erklärung: Sarsa(λ)

```
1   /** priv_sap ist nur NULL bei dem ersten Aufruf von
       do_AA_action()
2       in einer Episode */
3   if(priv_sap != NULL){
4       /** Auslesen des alten Q-Wertes aus dem Gedächtnis (Q(s,a)
           ) */
5       q_old = memory->value(priv_sap->state)->value(priv_sap->
           actionID);
6
7       /** Auslesen von Q(s',a') aus dem Gedächtnis*/
8       q_next = memory->value(cur_state)->value(action);
9
10      /** Berechnen von delta (Punkt d) ); */
11      delta = new_reward + (skonto * q_next) - q_old;
12
13      /** Bereitstellen von e(s,a) (Punkt e) ) */
14      e_sa = 1;
15
16      /** Aktualisieren von Q(s,a) für alle (s,a)
17          des Episodenverlaufs sap_trace(Punkt f) )*/
18      QList<state_action_pair*>::iterator iter;
19      for(iter = sap_trace.begin(); iter != sap_trace.end();
           iter++){
20          float q_sa =
21              memory->value((*iter)->state)->value((*iter)->
                   actionID);
22          q_sa = q_sa + (alpha * delta * e_sa);
23          memory->value((*iter)->state)->insert(((*iter)->
               actionID),q_sa);
24          e_sa = skonto * lambda * e_sa;
25      }
26  }
27  /** hinzufügen des Zustandaktionspaares zum Episodenverlauf */
28  sap_trace.push_front(priv_sap);
```

Anhang B

Benutzte Hilfsmittel

Die Entwicklungsplattform:

Betriebssystem: Windows 7 Professional 64bit

CPU: AMD Phenom II X4 955 (3,20 GHz)

RAM: DDR3 12800 CL7 2x2GB

Die Entwicklungstools:

Microsoft Visual Studio 2008 [Mic]

Visual Studio Qt addin 1.1.9 [Nokc]

Qt-Framework 4.7.4 [Noke]

Die Designtools:

Für die Planungsdiagramme aus dem Kapitel 4 wurde Visual Paradigm for UML 10.0 verwendet[Vis].

Für die Auswertung in Kapitel 5 wurde libreoffice calc verwendet[Doc].

Anhang C

Übersicht über Inhalt der CD

- ▷ Bachelorarbeit als PDF : Bachelorarbeit EvRLfSG
- ▷ Source-Code im Ordner „Source-Code”
  - Source-Code der Umweltsoftware in Source-Code/Umwelt_src
  - Source-Code der Agentensoftware in Source-Code/Agent_src

Versicherung über die Selbständigkeit

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen habe ich unter Angabe der Quellen kenntlich gemacht.

______________________	______________________
Ort, Datum	Unterschrift