DIPLOMARBEIT

XCS in dynamischen Multiagenten-Überwachungsszenarien

von

Clemens Lode

Institut für Angewandte Informatik
und Formale Beschreibungsverfahren
Universität Karlsruhe (TH)

Referent: Prof. Dr. Hartmut Schmeck
Betreuer: Dipl. Wi.-Ing. Urban Richter

Karlsruhe, 31.03.2009

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Programmverzeichnis

Kapitel 1

Einleitung und Motivation

Für diese Arbeit wird ein Überwachungsszenario untersucht, das ein spezielles Räuber-Beute-Szenario [MC94] darstellt. Ein solches Szenario besteht aus zwei Gruppen sich bewegender Agenten, einem Feld auf dem sie sich bewegen können und Hindernissen verschiedener Art und Eigenschaft. Ziel der einen Gruppe („Räuber“) ist es, Mitglieder der anderen Gruppe („Beute“) einzufangen, während diese versuchen den Räubern auszuweichen. Bei einem Überwachungsszenario soll weniger das Einfangen selbst, als eine Überwachung der Beute als Ziel gesetzt werden. Eine Untersuchung dieses Szenarios ist interessant, da es zum einen ein Testfeld für verschiedene Algorithmen als auch eine offensichtliche Parallele zur Realität gibt. Insbesondere für das Forschungsgebiet lernender Agenten bietet es viele Ansatzpunkte.

Ein aktuelles Forschungsgebiet aus dem Bereich der *learning classifier systems* (LCS) stellen die sogenannten *eXtended Classifier Systems* (XCS) dar. Grundsätzlich entspricht XCS einem LCS, d.h. eine Reihe von Regeln (sogenannte *classifier*), bestehend jeweils aus einer Kondition und einer Aktion, zusammen mit einem Mechanismus, das versucht, eine gegebene Aufgabe zu lernen. Die Regeln werden mittels *reinforcement learning* schrittweise bewertet und an eine Umwelt angepasst. Bei der Frage nach dem Zeitpunkt der Bewertung gibt es bei den verwendeten Algorithmen bei XCS sogenannte *single step* und *multi step* Verfahren. Welches der Verfahren eingesetzt wird, hängt von der Problemstellung ab. Hauptaugenmerk dieser Arbeit ist das *multi step* Verfahren, bei dem über eine schrittweise Weitergabe von Bewertungen versucht wird, eine Aufgabenstellung zu lösen, über das keine globale Information verfügbar ist.

Bisherige Anwendungen von XCS haben sich hauptsächlich auf statische Szenarien mit nur einem Agenten oder mit mehreren Agenten mit zentraler Steuerung und Kommunikation beschränkt. Diese Arbeit konzentriert sich dagegen auf das Problem, ob und wie es gelingen kann, XCS so zu modifizieren, damit es sogenannte Überwachungsszenarien besser besteht, als Agenten mit zufälliger Bewegung.

Primäres Problem hierbei ist, dass die Agenten zum einen nur lokale Information besitzen und zum anderen ein solches Szenario aufgrund der Bewegung des Zielobjekts und der anderen Agenten dynamisch ist. Dadurch sind die Bedingungen, die für die Anwendung von *single step* oder ein *multi step* Verfahren vorausgesetzt sind, nicht erfüllt.

Die Zahl der möglichen Anpassungen, insbesondere was das Szenario, die XCS Parameter und Anpassungen an die XCS Implementierung betrifft, sind unüberschaubar groß. Sie bedürfen primär einer theoretischen Basis, welche noch nicht weit fortgeschritten ist. Ziel dieser Arbeit ist es deshalb, insbesondere anhand empirischer Studien zu untersuchen, welche Anpassungen speziell für das Überwachungsszenario erfolgsversprechend sind.

Wesentliche Schwerpunkte der Untersuchung sind Szenarien ohne lernende Agenten, die Analyse der Bewertungsfunktion, die Bestimmung einer geeigneten Auswahlart für Aktionen der Agenten und die Bestimmung von passenden Parameter. Auf diesen Untersuchungsergebnissen aufbauend wird dann ein neuer Algorithmus entwickelt („SXCS“) und in mehreren Tests mit der aus der Literatur bekannten Standardimplementierung von XCS in verschiedenen Szenarien verglichen. Außerdem wird ein Ansatz für Kommunikation entwickelt und diskutiert, der jedoch letztlich keine Verbesserungen im betrachteten Szenario erbrachte.

Nachfolgend stellt Kapitel 1.1 den aktuellen Stand der Wissenschaft dar und grenzt diese Arbeit von anderen ab. Insbesondere ist dort das Ergebnis, dass sich das Problem von den normalerweise verwendeten Problemstellungen, die über das *single step* bzw. *multi step* Verfahren mehr oder weniger lösbar sind, unterscheidet. In Kapitel 1.2 wird dann eine Übersicht über den weiteren Aufbau dieser Arbeit gegeben und Kapitel 1.3 fasst die wesentlichen Ergebnisse der Arbeit zusammen.

1.1 Stand der Wissenschaft

It's so wonderful to see a great, new, crucial achievement which is not mine!
Ayn Rand

Das auf Genauigkeit der auf *classifier* basierende XCS wurde zuerst in [Wil95] beschrieben und stellt eine wesentliche Erweiterung von LCS dar. Neben neuer Mechanismen zur Generierung neuer *classifier*, insbesondere im Bereich bei der Anwendung des genetischen Operators, gibt es im Vergleich zum LCS vor allem innerhalb der Funktion zur Bewertung der *classifier* Unterschiede. Während die Bewertung bei LCS direkt aus der Differenz zwischen erwarteter und tatsächlicher Bewertung berechnet wird, wird sie bei XCS auf Basis einer speziellen *accuracy* Funktion berechnet. Eine ausführliche Beschreibung zu XCS findet sich in [But06b].

Die in der Literatur besprochenen Implementierungen und Varianten von XCS beschäftigen sich meist mit Szenarien mit statischer Umgebung: Häufiger Gegenstand der Untersuchung sind insbesondere relativ einfache Probleme wie das 6-Multiplexer oder das Maze1 Problem [But06b] [Wil95] [Wil98]. Die Probleme sind Vertreter aus der Klasse der XCS *single step* bzw. *multi step* Problemen, welche im Folgenden in Kapitel 1.1.1 bzw. Kapitel 1.1.2 angesprochen werden.

Die in dieser Arbeit verwendete Implementierung entspricht im Wesentlichen der Standardimplementierung des *multi step* Verfahrens von [But00]. Die algorithmische Beschreibung des Algorithmus findet sich in [BW01], wo auch näher auf die Unterscheidung von *single step* und *multi step* Verfahren eingegangen wird. Eine Besonderheit stellt allerdings die Problemdefinition dar, auf die Kapitel 1.1.3 eingeht und das die Unterschiede zwischen dem Überwachungsszenario und den XCS Standardproblemen darstellt. Abschließend werden in Kapitel 1.1.4 exemplarisch einige Arbeiten besprochen, die das Thema dieser Arbeit schneiden, aber nur jeweils Teilaspekte behandeln.

1.1.1 Beschreibung und Beispiel für das *single step* Verfahren

Im einfachsten Fall des *single step* Verfahrens, erfolgt die Bewertung einzelner Regeln, also der Bestimmung eines jeweils neuen *fitness* Werts, sofort nach Aufruf jeder einzelnen Regel, während im sogenannten *multi step* Verfahren mehrere aufeinanderfolgende Regeln

erst dann bewertet werden, sobald ein Ziel erreicht wurde.

Ein klassisches Beispiel für den Test des *single step* Verfahrens ist das 6-Multiplexer Problem [But06b], bei dem das XCS einen Multiplexer simulieren soll, der bei der Eingabe von 2 Adressbits und 4 Datenbits das korrekte Datenbit liefert. Sind beispielsweise die 2 Adressbits auf „10“ und die 4 Datenbits auf „1101“ gesetzt, so soll das dritte Datenbit, also „0“, zurückgegeben werden. Im Gegensatz zum Überwachungsszenario kann also über die Qualität eines XCS direkt bei jedem Schritt entschieden werden. Abbildung 1.1 zeigt eine Darstellung des Problems.

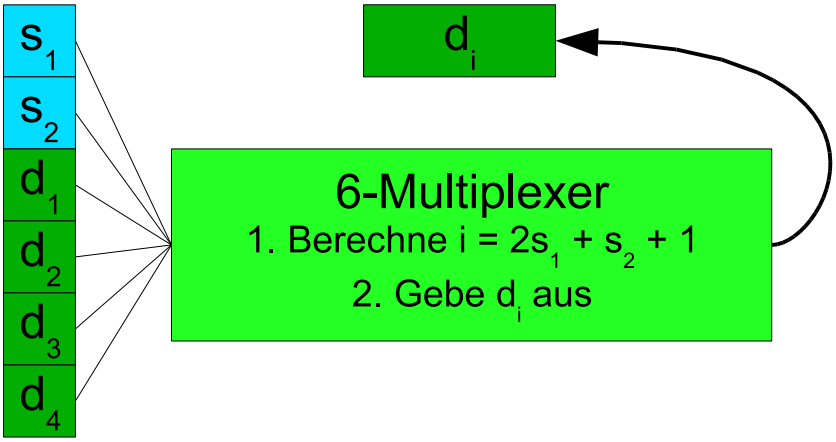

Abbildung 1.1: Schematische Darstellung des 6-Multiplexer Problems (s_1 und s_2 bezeichnen die Adressbits, d_1, d_2, d_3 und d_4 bezeichnen die Datenbits)

1.1.2 Beschreibung und Beispiel für das *multi step* Verfahren

Ein klassisches Beispiel für *multi step* Verfahren ist das *Maze N* Problem, bei dem durch ein Labyrinth auf dem kürzesten Weg mit N Schritten gegangen werden muss. Am Ziel angekommen wird der zuletzt aktivierte *classifier* positiv bewertet und das Problem neu gestartet. Bei den Wiederholungen erhält jede Regel einen Teil der Bewertung des folgenden *classifier*. Dadurch bewertet man eine ganze Kette von *classifier* und nähert sich der optimalen Wahrscheinlichkeitsverteilung an. Diese Verteilung sagt aus, welche der Regeln in welchem Maß am Lösungsweg beteiligt sind. Das (sehr einfache) Szenario in Abbildung 1.2 verdeutlicht dies.

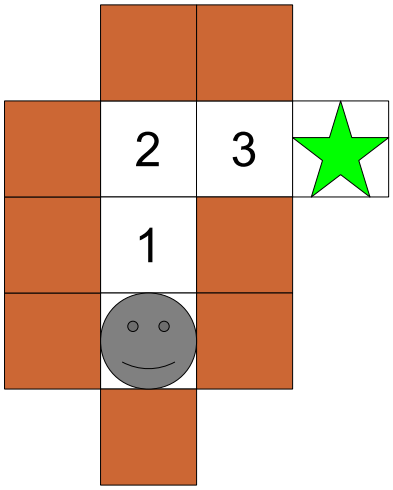

Abbildung 1.2: In der Darstellung eines einführenden Beispiels zum XCS *multi step* Verfahren entspricht der Stern dem Zielpunkt, das Gesicht dem Agenten, die roten Feldern den Hindernissen und der Rest den freien Feldern.

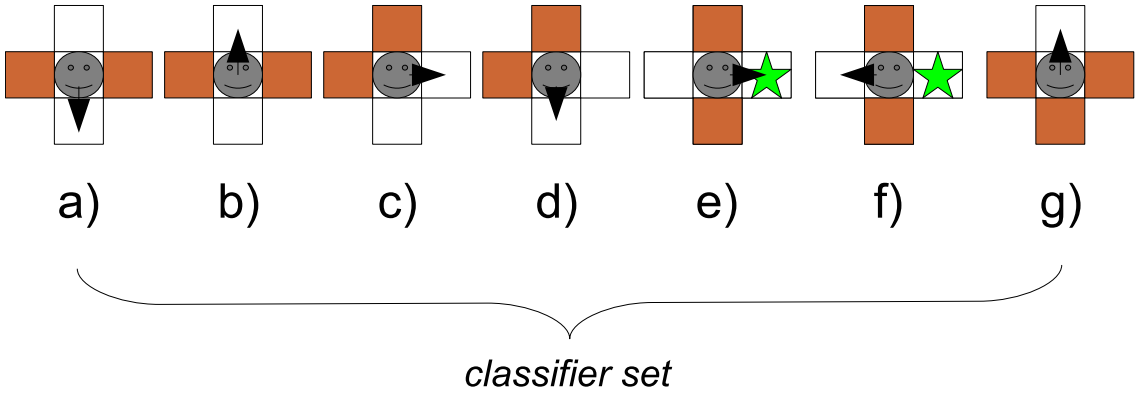

Abbildung 1.3: Vereinfachte Darstellung eines *classifier set* für das Beispiel zum XCS *multi step* Verfahren

Die zum Agenten zugehörigen *classifier* (das sogenannte *classifier set*) sind in Abbildung 1.3 dargestellt, wobei die vier angrenzenden Felder für jeden *classifier* jeweils die Konfiguration der Kondition darstellen und der Pfeil die Aktion.

Der Verlauf gestaltet sich in mehreren Abschnitten:

1. Alle *classifier* in jedem Schritt zufällig gewählt.
2. Der *classifier* e) erhält eine positive Bewertung.
3. Der *classifer* c) einen von *classifier* e) weitergegebene positive Bewertung.
4. Der *classifier* e) auf Position 3 wird mit höherer Wahrscheinlichkeit als *classifier* f) gewählt.
5. ...
6. Die Berechnung ist abgeschlossen wenn sich für *classifier* b), c), e) und g) ein ausreichend großer Wert eingestellt hat und keine wesentlichen Veränderungen mehr auftreten.

1.1.3 Problemdefinition

Das für das Überwachungsszenario am ehesten einsetzbare Verfahren ist das *multi step* Verfahren, da eine ganze Reihe von Schritten (Weg) durchgeführt werden muss und diese mangels globaler Sicht auf das Szenario nicht mit dem *single step* Verfahren berechnet werden können.

Das *multi step* Verfahren kann hier das Problem lösen, einen möglichst kurzen Weg von einem Start- zu einem Zielpunkt zu finden. Dabei wird die Umgebung als statisch angesehen. Wird das Ziel gefunden, wird das Problem neu gestartet und das Verfahren versucht in einem neuen Durchlauf einen noch besseren Weg zu finden. Außerdem wird zwischen unterschiedlichen Lernphasen unterschieden, wobei die Qualität des Algorithmus nur in bestimmten Zeitabschnitten gemessen wird (siehe Kapitel 3.5.

Im Fall des Überwachungsszenarios soll aber kein Zielpunkt oder Weg gefunden, sondern über die Zeit hinweg ein bestimmtes Verhalten, nämlich die Überwachung des Zielobjekts, erreicht werden. Deshalb stellt sich die Frage, wie das Problem hier definiert werden soll. Ein Neustart des Problems ist von der Aufgabenstellung her ausgeschlos-

sen und es gibt keinen festen Zielpunkt. Durch die Bewegung der anderen Agenten und des Zielobjekts verändert sich außerdem die Umwelt in jedem Schritt, ein Lernen durch Wiederholung von ähnlichen Bewegungsabläufen wie bei XCS ist deswegen schwieriger. Außerdem wird auch die Qualität nicht nur während bestimmten Phasen, sondern während des ganzen Laufs gemessen.

Nachfolgend wird herausgearbeitet, inwieweit es in der Literatur Arbeiten gibt, die ähnliche Probleme ansprechen bzw. inwiefern sich diese Arbeit von anderen Arbeiten abgrenzt.

1.1.4 Arbeiten über XCS Standardproblemen

Häufiger Gegenstand der Untersuchung in der Literatur ist insbesondere die Anwendung von XCS auf die oben in Kapitel 1.1.1 und Kapitel 1.1.2 besprochenen relativ einfachen Problemtypen. Es gibt zahlreiche Arbeiten, die sich mit verschiedenen Modifikationen für spezielle Szenarien, Optimierung der Parameter oder Anwendung auf schwierigere Probleme beschäftigen. Im Folgenden werden exemplarisch drei Arbeiten vorgestellt um dann eine Abgrenzung zu dieser Arbeit herauszustellen.

In [Wil95] wird das XCS *single step* Verfahren beim *11-Multiplexer* Problem (3 Adressbits, 8 Datenbits) und das XCS *multi step* Verfahren beim *Woods2* Problem getestet. Beim *Woods2* Problem (siehe Abbildung 1.4) ist das Ziel für den Agenten Futter auf dem Feld zu finden. Dabei kann er sich anhand unterschiedlicher Hindernisse und den freien Feldern orientieren und sich in die acht angrenzenden Felder bewegen. In beiden genannten Problemen konnte der XCS Algorithmus das Problem erfolgreich durch Generalisierung lösen.

Weiter verfeinert und getestet wird der XCS Algorithmus in [Wil98]. Hier werden Generalisierungsoperatoren (Zusammenfassung von *classifier* und eine Beschränkung des genetischen Operators auf eine kleinere Menge von *classifier*) für XCS eingeführt, welche in den betrachteten Szenarien (*6-Multiplexer*, *11-Multiplexer*, *20-Multiplexer* und *Woods2* Problem) zu deutlich besseren Ergebnissen führten.

Mit schwierigeren Problemen beschäftigt sich beispielsweise die Arbeit [BGL05]. Fra-

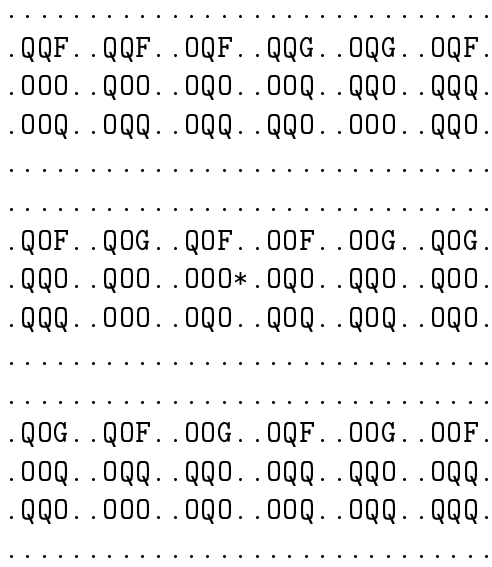

Abbildung 1.4: Darstellung einer Konfiguration des *Woods2* Problems, „.“ entspricht leerem Feld, „*“ dem Agenten, „F“ und „G“ dem Futter und „O“ und „Q“ den Hindernissen (siehe [Wil95])

gestellung hier ist, wie man u.a. die Probleme *Maze6* und *Woods14* (siehe Abbildung 1.5) mit XCS besser in den Griff bekommen kann. Ziel beider Szenarien ist, dass ein Agent, der an einem zufälligen freien Feld startet, das Futter („F“) findet. Unter „schwieriger“ wird hier aber lediglich eine größere Anzahl von Schritten zum Zielpunkt verstanden, also nicht etwa ein dynamisches Szenario mit z.B. einem sich bewegendem Zielpunkt. Zur Lösung wird ein Gradientenabstieg in XCS implementiert, wodurch besseren Ergebnisse in den besagten Problemen erreicht werden. Beim Gradientenabstieg werden Aktualisierungen einzelner *classifier* zusätzlich anhand ihrer Genauigkeit gewichtet, es handelt sich also um eine Erweiterung der Aktualisierungsfunktion.

Wie in Kapitel 1.1.3 auflistet und später in Kapitel 2 diskutiert, gibt es signifikante Unterschiede zwischen den Standardszenarien und dem in dieser Arbeit untersuchten Überwachungsszenarien. Deshalb werden im Folgenden Arbeiten aufgelistet, die sich mehr auf den Anwendungsfall und insbesondere auf dynamische Szenarien beziehen.

```
TTTTTTTTT   TTTTTTTTTTTTTT
T.....TFT   TT...TTTT.TT.T
T..T.TT.T   T.TTT.TT.T.T.T
T.T.....T   T.TTT.T.TTT.TT
T...TT..T   TFTTT.TT.TTTTT
T.T.T..TT   TTTTTT..TTTTTT
T.T.....T   TTTTTTTTTTTTTT
T.....T.T
TTTTTTTTT
```

Abbildung 1.5: Darstellung einer Konfiguration des *Maze6* und des *Woods14* Problems, „.“ entspricht einem leeren Feld, „F“ dem Futter und „T“ einem Hindernis (siehe [BGL05])

1.1.5 Arbeiten zu dynamischen *single step* Problemen

Vielversprechend war der Titel der Arbeit [LWB08], „Generation of Rule-based Adaptive Strategies for a Collaborative Virtual Simulation Environment“. In der Arbeit wurde das XCSlib [Lan] mit einem Open Source Echtzeitstrategiespiel verknüpft und bei jedem Schritt des Spiels wurde die aktuelle Situation mit dem *classifier set* verglichen und sich für eine Aktion entschieden. Ziel war es, eine Reihe von Gebäuden und Einheiten zu errichten, wofür es einer bestimmten Abfolge bedarf (z.B. zuerst das Haupthaus, dann die Arbeiter). Leider wird in der Arbeit nicht diskutiert, auf was sich der kollaborative Anteil bezog, da nicht mehrere Agenten benutzt worden sind. Auch zeigten dort Testläufe mit dem *multi step* Verfahren keine Anzeichen, dass ein Lernen stattfand, weshalb sich auf das *single step* Verfahren konzentriert wurde. Deshalb können, trotz einer ähnlichen Dynamik wie beim Überwachungsszenario, die Ergebnisse und Herangehensweisen nicht mit dieser Arbeit verglichen werden.

Eine weitere Arbeit in dieser Richtung [HFA02] beschreibt das „El Farol“ Bar Problem (EFBP) in Verbindung mit XCS und einem Multiagentensystem. Im EFBP geht es um eine Bar und eine Anzahl von Personen. Jede Person kann entscheiden, ob sie die Bar besucht oder nicht. Entscheiden sich zuviele Personen für einen Besuch, dann gibt es für keine Person eine positive Bewertung. Besucht eine Person von sich aus die Bar nicht, gibt es (für diese Person) ebenfalls keine positive Bewertung. In der Arbeit wurde eine Methode benutzt („MAXCS“), um (in Verbindung mit XCS) kooperativ die Bewertungen zwischen den Personen zu verteilen und die Ergebnisse mit egoistisch handelnden Personen verglichen. Als Ergebnis wurde eine Emergenz festgestellt, d.h. die Agenten ko-

operierten miteinander und die Aufgabe konnte dadurch optimal gelöst werden. Zwar ist auch dies ein dynamisches Szenario, die Vergleichbarkeit ist aber sehr eingeschränkt, da die Personen jeweils das Verhalten der anderen Personen in der letzten Woche kennen, und somit globale Information besitzen, weshalb es sich bei dem EFBP ebenfalls um ein *single step* Problem handelt.

1.1.6 Arbeiten zur Auswahlart von *classifier*

Ein in der Arbeit wesentlicher Punkt ist die Auswahlart der *classifier*, also ob eher explorative Verhalten (die sogenannte *explore* Phase), gefördert oder ob eher jeweils nur der vielversprechendste *classifier* ausgewählt (die *exploit* Phase) bzw. inwiefern zwischen diesen beiden Phasen hin- und hergeschaltet werden soll.

In der Standardimplementierung von XCS wird in jeder Probleminstanz entweder zufällig oder in jedem Schritt anhand eines Parameters zwischen der *explore* und *exploit* Phase gewechselt. Die Motivation von [MSB05] ist, dass sich durch vermehrtes Wissen des XCS nach einigen absolvierten Problemen die Balance zwischen beiden Phasen ändert. Als Idee wird vorgeschlagen, anstatt über Zufall und manuelle Tests des Parameters, diese Balance während eines Laufs automatisch anhand von Statistiken anzupassen. Im betrachteten Szenario wurde eine deutliche Verringerung der nötigen *explore* Phasen ohne Qualitätseinbußen und somit auch eine deutlich niedrigere Laufzeit erreicht.

Zwar wird die Idee aus der Arbeit, die Auswahlart dynamisch während eines Laufs anhand der ermittelten Statistiken zu wechseln, aufgegriffen; allerdings konnte die Erweiterung nicht übernommen werden, da sie sich wieder auf *multi step* Probleme bezieht. Stattdessen wurde ein eigener Algorithmus implementiert, der auf Basis der Sichtbarkeit des Zielobjekts entscheidet, wann zwischen den beiden Phasen gewechselt wird.

1.1.7 Arbeiten zu dynamischen Multiagentensystemen

Eine dieser Arbeit am nächsten kommende Problemstellung wird in [ITS05] vorgestellt. Dabei wird ein vereinfachtes Fußballspiel simuliert, bei dem ein bis zwei Agenten versuchen müssen, einen Ball auf der jeweils gegenüberliegenden Seite aus dem Spielfeld zu

befördern. Ähnlich wie in dieser Arbeit haben die Agenten Sensoren in die vier verschiedenen Richtungen und können sich ebenfalls in diese Richtungen bewegen. Außerdem ist das Szenario dynamisch, d.h. es gibt andere, sich bewegende Objekte. Unterschiede zu der hier verwendeten Problemstellung sind allerdings zum einen, dass sobald das Ziel erreicht wurde das Problem neu gestartet wird und zum anderen der Schwierigkeitsgrad. Zwar bewegt sich der Ball, jedoch nur dann, wenn ihn ein Agent anstösst. Dies sorgt für eine geringere Dynamik, als ein sich andauernd bewegendes Zielobjekt. Desweiteren ist der einzelne Gegenspieler in der anderen Mannschaft ein sich zufällig bewegender Agent und es gibt keine Hindernisse die es zu umgehen gilt.

Bezüglich des Aspekts des Multiagentensystems wird dort außerdem versucht, die Bewertung unter den (zwei) Agenten aufzuteilen. Die Aufteilung läuft nach dem sogenannten *profit sharing* Schema wie es in der Arbeit [KM94] vorgestellt wurde. Dabei erhält der der n-te Agent in einer vorher festgelegten Gruppe von Agenten die Bewertung $f_n = \frac{1}{M} f_{n-1}$ und der erste Agent die maximale Bewertung. In der oben vorgestellten Arbeit bei der Simulation eines Fußballspiels wird beispielsweise der Agent, der den Ball aus dem Spielfeld befördert hat mit 1,0 und der andere mit 0,9 bewertet. Diese Form der Verteilung des *reward* Werts ist im hier besprochenen Überwachungsszenario direkt leider nicht anwendbar, es werden hier aber alternative Ansätze angesprochen.

In der Literatur gibt es noch eine Reihe weiterer Multiagentensysteme die mit Kommunikation und einem XCS arbeiten. Die primäre Abgrenzung zu diesen Arbeiten ist allerdings, dass sie sich komplexer Kommunikation bedienen und damit u.a. globale, von den Agenten geteilte *classifier set* Listen anlegen, einzelne *classifier* untereinander austauschen, die Agenten in Gruppen zentral organisieren und einzelnen Agenten Rollen verteilen oder direkt zentral steuern. Beispielsweise wird in der Arbeit [TTS01] ein Organisationsmodell („Organizational-learning oriented Classifier System“, OCS) vorgestellt, welches hier aufgrund der Komplexität nur unzureichend dargestellt werden kann. Im Kern der Arbeit wird versucht, mittels einem gemeinsamen, geteilten Speicher Rollen für die Agenten zu verteilen. Der Schwerpunkt liegt also in der tatsächlichen Organisation unter den Agenten, während sich im Folgenden mehr auf die individuellen Agenten, die unabhängig voneinander eine gemeinsame Aufgabe lösen sollen, konzentriert wird.

Angesprochen wird in der Arbeit der Verweis auf die LCS Varianten mit Speicher,

deren Ziel es ist, Probleme zu lösen, die keine Markow-Kette darstellen. Eine Markow-Kette ist ein stochastischer Prozess, der gedächtnislos ist (das ist die sogenannte Markow-Eigenschaft), d.h. bei dem „die zukünftige Entwicklung des Prozesses nur von dem zuletzt beobachteten Zustand abhängt und von der sonstigen Vorgeschichte unabhängig ist“ [WS08]. Ein solches Problem wird in dieser Arbeit behandelt, es wird allerdings ein anderer Weg als in beispielsweise [LW99] gegangen. Dort kann eine Aktion eines Agenten sowohl äußere (z.B. Bewegung) als auch innere Auswirkungen (Veränderung des inneren Zustands) haben, wodurch eine Art Speicher realisierbar ist und somit die Markow-Eigenschaft wieder hergestellt werden kann. In dieser Arbeit werden dagegen die aktivierten *classifier* gespeichert bis ein Ziel erreicht wird.

1.1.8 Implementierungen von XCS

Die wesentlichen zwei Implementierungen zu XCS sind zum einen XCSlib [Lan], welche eine XCS Funktionsbibliothek in C++ zur Verfügung stellt, und zum anderen die ursprüngliche Implementierung [But00] welche eine kompakte Umsetzung in Java zur Verfügung stellt. Zur Simulation wurden keine der beiden Implementierungen verwendet, sondern eine eigene Java Umsetzung des XCS Algorithmus programmiert. Dies geschah zum einen, um die Funktionsweise von XCS voll zu verstehen und zum anderen, weil von Anfang an klar war, dass Modifikationen am Algorithmus vorgenommen werden sollen. Im Wesentlichen sind die Implementierungen identisch, die Unterschiede sind in den jeweiligen Kapiteln, insbesondere in Kapitel 4 erklärt und teilweise im Anhang (siehe Kapitel A) als Quelltext dargestellt.

1.2 Aufbau der Arbeit

Kapitel 1.1 stellt den gegenwärtigen Stand der Forschung dar, insbesondere in Bereichen, die sich mit dem Thema dieser Arbeit schneiden. Kapitel 2 geht dann auf das verwendete Szenario, die Eigenschaften der Objekte und vor allem die Eigenschaften der Agenten und des Zielobjekts ein. Schließlich wird erläutert, wie die Simulation auf dem beschriebenen Szenario ablaufen soll. In Kapitel 3 werden dann die wichtigsten Teile des XCS vorgestellt, insbesondere die sogenannten *classifier*, die Verarbeitung von Sensordaten, der allgemeine

Ablauf und die XCS Parameter. Vorbereitend für die Entwicklung neuer XCS Varianten sind insbesondere Kapitel 3.4, bei dem es um die Konstruktion einer passenden *reward* Funktion für die beschriebenen Szenarien geht, und Kapitel 3.6, bei dem es um die Frage geht, wann sich ein Agent für welche Aktion entscheiden soll, zu nennen. Darauf aufbauend bespricht Kapitel 4 Anpassungen und Verbesserungen des XCS Algorithmus. Speziell für das vorgestellte Szenario wird desweiteren eine selbstentwickelte XCS Variante (SXCS) vorgestellt und dann durch die Erweiterung der Möglichkeit zur Kommunikation zwischen den Agenten weiterentwickelt.

Der Schwerpunkt der Arbeit beschreibt Kapitel 5: Hier werden alle diskutierten Algorithmen in den vorgestellten Szenarien getestet und analysiert. Abschluss bildet die Zusammenfassung und der Ausblick in Kapitel 6. Im Anhang A findet sich dann noch eine Anzahl der wichtigsten Quelltexte der Algorithmen, die in dieser Arbeit vorgestellt wurden.

1.3 Wesentliche Erkenntnisse der Arbeit

Wesentliche Erkenntnisse aus dieser Arbeit werden sein:

- Die Bewertungsfunktion kann anhand einer Nachbildung einer gut funktionierenden Heuristik konstruiert werden (siehe Kapitel 3.4.3).
- Durch Hinzufügen einer Form von Speicher (siehe Kapitel 4.3) kann SXCS die betrachtete Aufgabe wesentlich besser als XCS lösen (siehe Kapitel 5).
- Eine Variation der Lernrate kann je nach Szenario sinnvoll sein (siehe Kapitel 5.5.7 und Kapitel 5.7.1).
- Die Agenten mit XCS und SXCS haben deutliche Probleme mit Szenarien mit vielen Hindernissen (siehe Kapitel 5.5.8).
- Ein dynamischer Wechsel der Auswahlart für Aktionen während eines Laufs kann sinnvoll sein, um die Zahl der blockierten Bewegungen zu verringern und das Zielobjekt besser verfolgen zu können (siehe Kapitel 5.7).

- Sowohl XCS als auch SXCS können (für sich zufällig bewegende Agenten) schwierige Szenarien (siehe Kapitel 2.2.4) mit markanten Hindernispunkten meistern (siehe Kapitel 5.7.1) und SXCS kann diese sogar besser als die intelligente Heuristik lösen (siehe Kapitel 5.7.2).
- Die vorgestellte Variante des verzögerten SXCS Algorithmus DSXCS bietet Raum für Verbesserung (siehe Kapitel 4.4).
- Die versuchte Implementierung von Kommunikation führte nicht zum Erfolg, zum einen wegen geringer Möglichkeiten zur Kooperation, zum anderen wegen zu einfach umgesetztem Algorithmus (siehe Kapitel 5.8 und 5.9).

Kapitel 2

Beschreibung des Szenarios

[Man] must know what he is and where he is—i.e., he must know his own nature and the nature of the universe in which he acts...
Ayn Rand

Im Folgenden werden Algorithmen in einem Szenario getestet, in dem mehrere Agenten ein sich bewegendes Zielobjekt überwachen sollen. Dies wird im folgenden als Überwachungsszenario bezeichnet. Die **Qualität** eines Algorithmus in einem solchen Szenario wird über den Anteil an der Gesamtzeit bewertet, in der er mit Hilfe der Agenten das Zielobjekt überwachen konnte (siehe Kapitel 2.6.4). Läuft der Test eines Algorithmus beispielsweise über 20.000 Zeiteinheiten und konnten Agenten das Zielobjekt in 4.000 Zeiteinheiten überwachen, ergäbe sich eine Qualität von 20%.

Als Umfeld wird ein quadratischer Torus verwendet, der aus quadratischen Feldern besteht. Für jedes bewegliche Objekt auf einem Feld des Torus gilt, dass es sich in einem Schritt nur auf eines der vier Nachbarfelder bewegen kann. Eine Ausnahme stellt hier das Zielobjekt dar, welches mehrere Bewegungen in einem Schritt durchführen kann (näheres dazu im Kapitel 2.5).

Die Felder können entweder leer oder durch ein Objekt besetzt sein. Besetzte Felder können nicht betreten werden, eine Bewegung auf ein solches Feld schlägt ohne weitere Konsequenzen fehl.

Es gibt drei verschiedene Arten von Objekten: Unbewegliche Hindernisse, ein zu überwachendes Zielobjekt und Agenten. Sowohl das Zielobjekt als auch die Agenten bewegen

sich jeweils anhand eines bestimmten Algorithmus und bestimmter Sensordaten. Eine nähere Beschreibung der Agenten findet sich in Kapitel 2.3.4, während die Eigenschaften des Zielobjekts in Kapitel 2.5 beschrieben werden.

Ziel dieses Kapitels ist es, Kapitel 5.1 vorzubereiten, in dem anhand von Tests herausgefunden werden soll, welche der hier vorgestellten Szenarien brauchbare Ergebnisse liefern können, um zum einen das gestellte Problem an sich, als auch die jeweils erforderlichen Eigenschaften besser zu verstehen.

Eine separate Beschäftigung mit diesen - relativ einfachen - Szenarien ist notwendig, um zum einen das selbstentwickelte Simulationsprogramm zu testen und zum anderen vergleichbare Ergebnisse zu erhalten. Ein Rückgriff auf die Literatur war deshalb nicht möglich. Insbesondere gibt es keine Arbeiten in Bezug auf XCS mit einer solchen Problemstellung. Auch beziehen sich die meisten Arbeiten in dieser Richtung auf relativ einfache Szenarien, die nur in der Weglänge skalieren, wie z.B. das in der Einleitung in Kapitel 1.1.2 erwähnte *Maze N* Problem, bei dem durch ein Labyrinth auf dem kürzesten Weg mit N Schritten gegangen werden muss. Das hier besprochene Szenario ist schwieriger, dafür aber etwas näher an der Realität.

Zwar entspricht das Standardszenario bei XCS einer Anzahl von Feldern, einem Agenten, Hindernissen und einem Ziel; es fehlen jedoch Arbeiten, die folgende Kriterien berücksichtigen:

Sichtbarkeit Die Sichtweite beschränkte sich in der Literatur meist auf angrenzende Felder.

Kollaboration : Meist war nur ein einzelner Agenten Gegenstand der Untersuchung.

Dynamik : Meist gab es eine feste Zielposition.

Messung der durchschnittlichen Qualität : Meist ging es um die Anzahl der Schritte zum Ziel) gemeinsam in einem Szenario betrachtet werden.

Beispiele für diese Kriterien wurden in der Einleitung in Kapitel 1.1 diskutiert. Im Folgenden wird nun zuerst auf die einzelnen Punkte eingegangen. In Kapitel 2.1 wird definiert, was unter einem dynamisch kollaborativem Szenario verstanden wird und in Kapitel 2.2

werden eine Reihe von Startkonfigurationen für den Torus vorgestellt. Anschließend werden in Kapitel 2.3 die Eigenschaften der Objekte diskutiert. Was die Bewegungen der Agenten bzw. des Zielobjekts betrifft, werden sie in Kapitel 2.4 bzw. in Kapitel 2.5 vorgestellt. Abschluss bildet der Beschreibung des Szenarios bietet dann Kapitel 2.6, in dem Statistiken und Parameter besprochen werden, und Kapitel 2.7, in dem die Reihenfolge diskutiert wird, in dem die einzelnen Elemente des Szenarios zusammenarbeiten.

2.1 Dynamische, kollaborative Szenarien

Schwerpunkt der Gestaltung der Szenarien ist hier Kollaboration. Kollaboration ist in der Literatur ein eher unscharf definierter Begriff. Nach einem der Standardwerke zu Multiagentensystemen [Wei00a] ist Kollaboration im Allgemeinen definiert als Zusammenarbeit zwischen den Agenten und bezieht sich oft auf Kooperation auf hohem Niveau und einer gemeinsamen Sicht auf die Problemstellung. Kooperation wiederum bezieht sich auf eine Koordination von an Zusammenarbeit interessierten Agenten bei der der Erfolg der einzelnen Beteiligten vom Gesamterfolg aller Agenten abhängt. Koordination ist der Prozess, den Zustand zu erreichen, bei dem die Agenten sich gegenseitig ergänzen anstatt sich gegenseitig zu blockieren oder unnötig Arbeiten doppelt erledigen.

In dieser Arbeit wird Kollaboration primär über die Aufgabenstellung gegeben sein, d.h. die (optimale Lösung der) Aufgabe kann nur mit Hilfe mehrere Agenten gemeinsam gelöst werden und es ist offen, ob die Agenten entsprechende Koordination erlernen können. Koordination kann über das Erkennen anderer Agenten erreicht werden, indem aktiv anderen Agenten ausgewichen wird um den Anteil des überwachten Gebiets zu erhöhen. Während sich die Agenten zwar eine Sicht auf dieselbe Problemstellung teilen, gibt es jedoch (ohne Kommunikation) keinen separaten Mechanismus zur Beteiligung am Erfolg anderer.

Eine erfolgreiche Überwachung ist deswegen so definiert, dass sich ein beliebiger Agent in Überwachungsreichweite des Zielobjekts befindet. Da diese Aufgabe auch ein einzelner Agent erfüllen kann, sofern die Geschwindigkeit des Zielobjekts kleiner oder gleich der Geschwindigkeit des Agenten ist, werden in späteren Tests, insbesondere in Kapitel 5 beim Vergleich unterschiedlicher XCS Varianten und im Kapitel 5.5 beim Vergleich un-

terschiedlicher XCS Parameter, unterschiedliche Geschwindigkeiten analysiert.

Bewegt sich das Zielobjekt zu schnell, werden die Agenten Schwierigkeiten haben, einen Bezug zwischen ihren Sensordaten und den eigenen Aktionen zu erkennen. Bewegt es sich zu langsam, wird das Problem sehr einfach, eine einzelne Regel („Bewege dich auf das Ziel zu„) würde zur Lösung dann schon genügen.

Unter einem dynamischen Szenario wird in [Wei00b] verstanden, dass es, im Gegensatz zu statischen Szenarien, weitere Prozesse neben dem einzelnen Agenten gibt, die die Umwelt ändert. Da zum einen die sich auf dem Feld bewegenden Agenten sowohl Bewegungs- als auch Sichthindernisse darstellen und zum anderen sich das Zielobjekt unabhängig von den Agenten bewegen kann, fallen die hier betrachteten Szenarien alle in die Kategorie „dynamisch“.

2.2 Konfigurationen des Torus

Beschrieben werden im Folgenden verschiedene Szenarien mit unterschiedlichen Werten für die Anzahl der Agenten, Größe des Torus sowie Art und Geschwindigkeit des Zielobjekts. Wesentliche Merkmale jedes Szenarios sind die Menge und die Verteilung der Hindernisse, der Startposition des Zielobjekts und die Startposition der Agenten.

In den folgenden Abbildungen repräsentieren rote Felder jeweils Hindernisse, weiße Felder Agenten und das grüne Feld das Zielobjekt. Außerdem sind die Sicht- und Überwachungsreichweiten aus Kapitel 2.3.1 dargestellt. Sie haben jeweils eine Gestalt, ähnlich der eines Viertels eines Kreisabschnitts mit dem jeweiligen Agenten im Mittelpunkt. In den Abbildungen ist der Bereich, der durch die Überwachungsreichweite abgedeckt wird, grau dargestellt, und der restliche Bereich, der zusätzlich noch durch die Sichtweite abgedeckt wird, blau.

2.2.1 Leeres Szenario ohne Hindernisse

Abbildung 2.1 zeigt ein Szenario ohne Hindernisse und mit zufälliger Verteilung der Agenten und zufälliger Position des Zielobjekts. Im leeren Szenario wird das Verhalten der Agenten in einem Torus ohne Hindernisse untersucht werden.

Eine Untersuchung dieses Szenarios erlaubt es dem Agenten die in Kapitel 2.3 besprochenen Sensoren, die für Hindernisse zuständig sind, zu ignorieren, was die Komplexität der Sensordaten deutlich verringert. Auch gibt es (mit Ausnahme der Agenten und des Zielobjekts selbst) keine Objekte die die Sicht versperren oder den Weg blockieren.

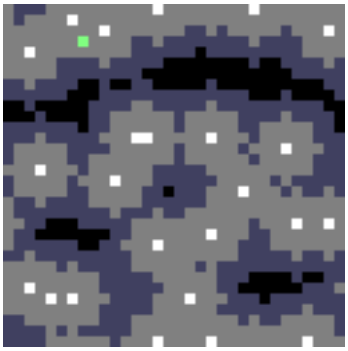

Abbildung 2.1: „Leeres Szenario“ ohne Hindernisse

2.2.2 Szenario mit zufällig verteilten Hindernissen

Für das Szenario mit zufällig verteilten Hindernissen sind zwei Parameter prägend: Der erste Parameter (Hindernissanteil λ_h) bestimmt den Prozentsatz an Hindernissen an der Gesamtzahl der Felder des Torus, der zweite Parameter (Verknüpfungsfaktor λ_p) beeinflusst die Wahrscheinlichkeit, dass zwei Hindernisse nebeneinander gesetzt werden.

Bei der Erstellung des Szenarios bestimmt λ_p die Wahrscheinlichkeit für jedes einzelne angrenzende freie Feld, dass beim Verteilen der Hindernisse nach dem Setzen eines Hindernisses dort sofort ein weiteres Hindernis gesetzt wird. $\lambda_p = 0{,}0$ ergäbe somit eine völlig zufällig verteilte Menge an Hindernissen, während ein Wert von 1,0 eine oder mehrere stark zusammenhängende Strukturen schafft. Wird der Prozentsatz an Hindernissen λ_h auf 0,0 gesetzt, dann entspricht diesem dem oben erwähnten leeren Szenario. Ein Wert

von 1,0 würde eine völlige Abdeckung des Torus bedeuten und wäre für einen Test somit unbrauchbar. Hier werden nur geringe Werte bis 0,4 betrachtet werden, wobei in Tests sich später auf Werte bis 0,2 beschränkt wird, da bei großem Hindernissanteil die lokalen Entscheidungen einzelner Agenten zu wichtig werden, da das Zielobjekt sich eher in einem kleinen Bereich aufhält.

In Abbildung 2.2 und Abbildung 2.3 werden Beispiele für zufällige Szenarien mit $\lambda_h = 0{,}1$ bzw. 0,2 und $\lambda_p = 0{,}5$ bzw. 0,99 dargestellt.

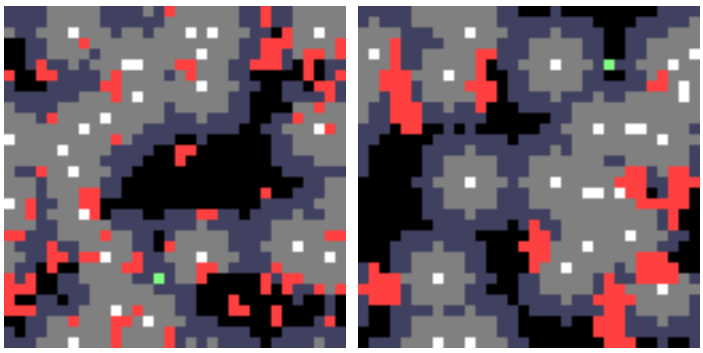

Abbildung 2.2: Szenario mit zufällig verteilten Hindernissen mit Hindernissanteil $\lambda_h = 0{,}1$ und Verknüpfungsfaktor $\lambda_p = 0{,}5$ bzw. 0,99

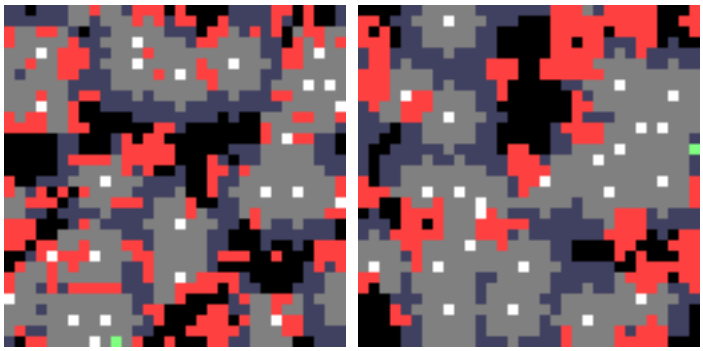

Abbildung 2.3: Szenario mit zufällig verteilten Hindernissen mit Hindernissanteil $\lambda_h = 0{,}2$ und Verknüpfungsfaktor $\lambda_p = 0{,}5$ bzw. 0,99

2.2.3 Säulenszenario

In diesem Szenario werden regelmäßig, mit jeweils 7 Feldern Zwischenraum zueinander, Hindernisse auf dem Torus verteilt. Der Zweck der Untersuchung dieses Szenarios ist,

wie beim leeren Szenario die Anzahl der blockierten Bewegungen und Sichtlinien zu minimieren, gleichzeitig aber genügend Hindernisse auf dem Feld zu verteilen, so dass sich Agenten daran orientieren können.

Das Zielobjekt startet an zufälliger Position, die Agenten starten mit möglichst großem Abstand zum Zielobjekt. Abbildung 2.4 zeigt ein Beispiel für den Startzustand eines solchen Szenarios, bei der das Zielobjekt sich in der Mitte und die Agenten am Rand befinden.

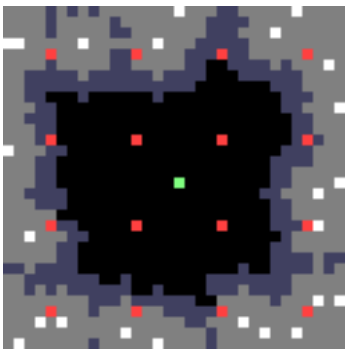

Abbildung 2.4: Startzustand des Säulenszenarios mit regelmäßig angeordneten Hindernissen und zufälliger Verteilung von Agenten mit möglichst großem Abstand zum Zielobjekt

2.2.4 Schwieriges Szenario

Hier wird der Torus an der rechten Seite vollständig durch Hindernisse blockiert, um den Torus zu halbieren. Alle Agenten starten zufällig verteilt am linken Rand, das Zielobjekt startet auf der rechten Seite.

In regelmäßigen Abständen mit 7 Feldern Zwischenraum befinden sich Hindernisse in vertikale Reihung mit Öffnungen von jeweils 2 Feldern Breite abwechselnd im oberen und unteren Viertel. Für dieses Szenario wird immer das Zielobjekt benutzt, das seine Richtung beibehält (siehe Kapitel 2.5.5). Abbildung 2.5 zeigt die Startkonfiguration eines solchen Szenarien.

Der Zweck der Untersuchung dieses Szenarios ist, zu testen, inwieweit die Agenten durch die Öffnungen zum Ziel finden können. Ohne Orientierung an den Öffnungen und anderen Agenten ist es sehr schwierig, sich durch das Szenario zu bewegen. Die später

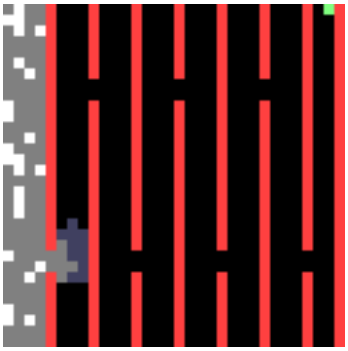

Abbildung 2.5: Schwieriges Szenario mit fester, wallartiger Verteilung von Hindernissen, in regelmäßigen Abständen und mit Öffnungen, mit den Agenten mit zufälligem Startpunkt am linken Rand und mit dem Zielobjekt mit festem Startpunkt rechts oben

besprochenen Tests in Kapitel 5.7.1 werden zeigen, dass dieses Szenario besonders schwierig für sich zufällig bewegende Agenten ist und welche Algorithmen auf welche Weise das Problem besser lösen können.

2.3 Eigenschaften der Objekte

Die sich auf dem Torus befindlichen Objekte haben verschiedene Eigenschaften, mit denen sie sich auf ihm bewegen oder andere Objekte wahrnehmen können. Neben den Hindernissen, die sich nicht bewegen können und nur das Feld blockieren gibt es die Agenten und das Zielobjekt, welche eine Anzahl visueller, binärer Sensoren mit begrenzter Reichweite besitzen. Jeder Sensor kann nur feststellen, ob sich in seinem Sichtbereich ein Objekt eines bestimmten Typs befindet (1) oder nicht (0). Die Sensoren sind jeweils in eine bestimmte Richtung ausgerichtet, wobei andere Objekte die Sicht blockieren können.

Die zur Bestimmung des Sichtbereichs nötigen Sichtlinien werden durch einen einfachen Bresenham-Algorithmus [Bre65] bestimmt. Aufgrund der großen Anzahl von Sichtbarkeitsprüfungen während eines Laufs werden in der Implementierung die Sichtlinien zu Beginn vorberechnet, so dass nur noch für jedes Feld in Sichtweite geprüft werden muss, ob sich auf der gespeicherten Sichtlinie ein Hindernis befindet.

Im Folgenden wird die Sichtbarkeit von Objekten wird in Kapitel 2.3.1 besprochen. In Kapitel 2.3.2 werden dann Sensordatenpaare besprochen, die jeweils aus zwei Sensoren mit gleicher Ausrichtung bestehen und denselben Objekttyp erkennen. Alle Sensoren, die nur gemeinsam haben, dass sie denselben Typ von Objekt erkennen, werden in einer Gruppe zusammengefasst. Der Aufbau eines ganzen, aus solchen Gruppen bestehenden Sensordatensatzes, bespricht Kapitel 2.3.3. Die restlichen Eigenschaften der Agenten und des Zielobjekts beschreibt dann schließlich Kapitel 2.3.4.

2.3.1 Sichtbarkeit von Objekten

Die Parameter *sight range* und *reward range* bestimmen, bis zu welcher Distanz andere Objekte von einem Objekt als „gesehen“ bzw. „überwacht“ gelten, sofern die Sicht durch andere Objekte nicht versperrt ist. Der Parameter *reward range* ist relevant für die Bewertung der Qualität des Algorithmus (siehe Kapitel 2.6.4), während der Parameter *sight range* immer größer als *reward range* gewählt wird, damit die Agenten das Ziel leichter erkennen zu können. Über die Sensoren kann ein Agent bzw. das Zielobjekt feststellen, in welcher der beiden Reichweiten sich Objekte befinden oder ob keine Objekte in Sicht sind. Falls nicht anders angegeben, wird *sight range* auf 5 und *reward range* auf 2 gesetzt.

2.3.2 Aufbau eines Sensordatenpaars

Ein Datenpaar besteht aus zwei Sensoren, die denselben Typ von Objekt erkennen, in dieselbe Richtung ausgerichtet sind und sich nur in ihrer Sichtweite unterscheiden. Dadurch kann der Agent rudimentär die Entfernung zu anderen Objekten feststellen. Die Sichtweite des ersten Sensors eines Paares wird über den Parameter *sight range* bestimmt, die Sichtweite des zweiten Sensors über den Parameter *reward range* (siehe auch Kapitel 2.3.1). Da *sight range* > *reward range* gilt, ist der überwachte Bereich eine Teilmenge des sichtbaren Bereichs. In Abbildung 2.6 sind alle Sichtreichweiten (heller und dunkler Bereich) und Überwachungsreichweiten (heller Bereich) für die einzelnen Richtungen dargestellt.

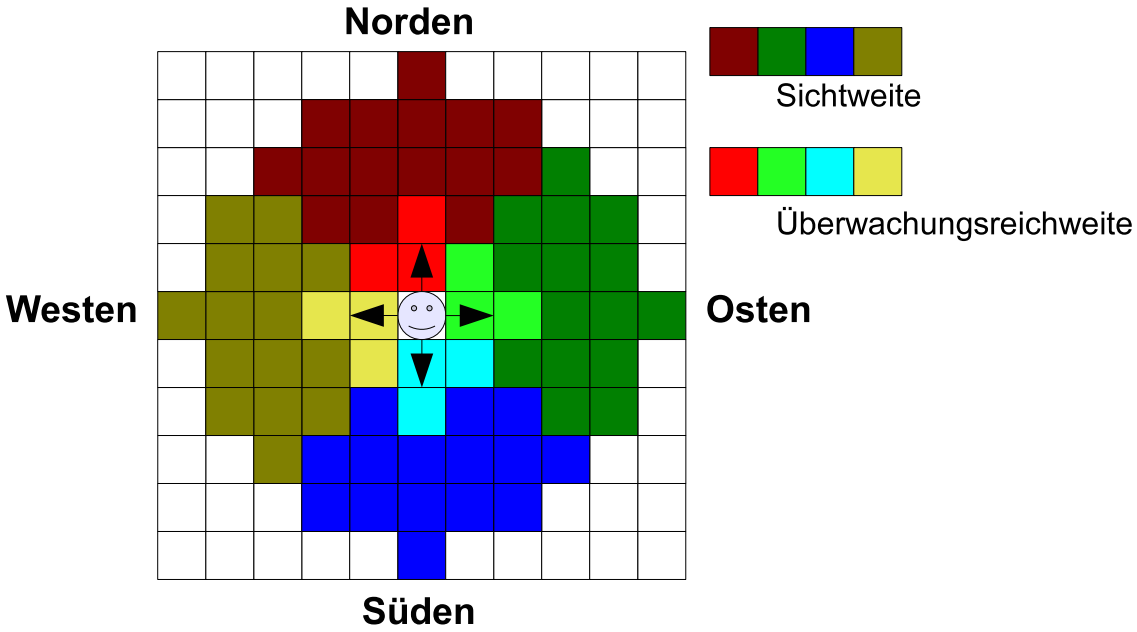

Abbildung 2.6: Sicht- (5,0, dunkler Bereich) und Überwachungsreichweite (2,0, heller Bereich) eines Agenten, jeweils für die einzelnen Richtungen

Sei $r(O_1, O_2)$ die Distanz zwischen dem Objekt O_1, das die Sensordaten erfasst und dem nächstliegenden Objekt O_2 des Typs, den der Sensor wahrnehmen kann, dann ergeben sich folgende Fälle:

1. (0/0): $r(O_1, O_2) >$ *sight range* (kein passendes Objekt in Sichtweite)
2. (1/0): *reward range* $< r(O_1, O_2) \leq$ *sight range* (Objekt in Sichtweite)
3. (1/1): $r(O_1, O_2) \leq$ *reward range* (Objekt in Sicht- und Überwachungsreichweite)
4. (0/1): *reward range* $\geq r(O_1, O_2) >$ *sight range* (Fall kann nicht auftreten, da *reward range* $<$ *sight range*)

2.3.3 Aufbau eines Sensordatensatzes

In einem Sensordatensatz sind jeweils acht Sensoren zu jeweils einer Gruppe zusammengefasst, welche wiederum jeweils in vier Richtungen mit jeweils einem Sensorenpaar aufgeteilt ist. Abbildung 2.7 stellt den allgemeinen Aufbau eines kompletten Sensordatensatzes dar, der aus den drei Gruppen der Zielobjektsensoren (z), der Agentensensoren (a) und der Hinernisssensoren (h) besteht:

$$\text{Sensordatensatz } s = \underbrace{(z_{s_N} z_{r_N})(z_{s_O} z_{r_O})(z_{s_S} z_{r_S})(z_{s_W} z_{r_W})}_{Erste\ Gruppe\ (Zielobjekt)}$$
$$\underbrace{(a_{s_N} a_{r_N})(a_{s_O} a_{r_O})(a_{s_S} a_{r_S})(a_{s_W} a_{r_W})}_{Zweite\ Gruppe\ (Agenten)} \underbrace{(h_{s_N} h_{r_N})(h_{s_O} h_{r_O})(h_{s_S} h_{r_S})(h_{s_W} h_{r_W})}_{Dritte\ Gruppe\ (Hindernisse)}$$

Abbildung 2.7: Sensordatensatzes, eingeteilt in mehrere Gruppen und Sensorpaare

Befindet sich beispielsweise das Zielobjekt außerhalb der Überwachungsreichweite aber innerhalb der Sichtweite im Norden, befinden sich im Süden ein oder mehrere Agenten in Überwachungsreichweite und befinden sich im Westen und Osten ebenfalls in Überwachungsreichweite des Agenten befindliche Hindernisse, dann ergibt sich ein Sensordatensatz $s_{Beispiel}$ wie in Abbildung 2.8 dargestellt.

$$s_{Beispiel} = (1,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,1,1,0,0,1,1)$$

Abbildung 2.8: Beispiel für einen Sensordatensatz mit dem Zielobjekt im Norden, ein oder mehreren Agenten im Süden und Hindernissen im Westen und Osten

2.3.4 Eigenschaften der Agenten und des Zielobjekts

Ein Agent kann in jedem Schritt zwischen vier verschiedenen Aktionen wählen, die den vier Himmelsrichtungen entsprechen. Darüber hinaus kann sich das Zielobjekt jedoch je nach Szenarioparameter auch mehrere Schritte bewegen, was in Kapitel 2.5 erläutert wird.

Da ein Multiagentensystem auf einem diskreten Torus betrachtet wird, werden alle Agenten nacheinander in der Art abgearbeitet, so dass jeder Agent die aktuellen Sensordaten (siehe Kapitel 2.3) aus der Umgebung holt und auf deren Basis die nächste Aktion bestimmt.

Wurden alle Aktionen bestimmt, können die Agenten in zufälliger Reihenfolge versuchen, sie auszuführen. Ungültige Aktionen, d.h. der Versuch sich auf ein besetztes Feld zu bewegen, schlagen fehl und der Agent führt in diesem Schritt keine Aktion aus, wird aber auch nicht bestraft. Eine detaillierte Beschreibung der Bewegung, im Kontext anderer Agenten und Programmteile, legt Kapitel 2.7 dar.

Auf dem Torus bewegt sich neben den Agenten auch das Zielobjekt. Es kann, wie die Agenten auch, unterschiedlichen Bewegungsarten folgen, besitzt aber außerdem noch eine bestimmte Geschwindigkeit (siehe Kapitel 2.5). Neben der Größe des Torus und den Hindernissen tragen diese Eigenschaften des Zielobjekts wesentlich zur Schwierigkeit eines Szenarios bei, da diese die Aufenthaltswahrscheinlichkeiten des Zielobjekts unter Einbeziehung des Zustands des letzten Schritts bestimmen. Beispielsweise gibt es bei einem Ziel mit zufälligem Sprung (siehe Kapitel 2.5.1) keine Verbindung zwischen den Positionen des Zielobjekts zweier aufeinanderfolgender Zeiteinheiten. Dadurch wird es sehr schwierig, mittels lokaler Heuristiken eine hohe Qualität zu erreichen (siehe Kapitel 5.1.1).

Diese Form der Bewegung wird auch nur zur allgemeinen, vorbereitenden Analyse dienen, während einfache Bewegungen, wie die zufällige Bewegung (Kapitel 2.5.2) bzw. die Bewegung mit einfacher Richtungsänderung (Kapitel 2.5.3) die später tiefer untersuchten Bewegungsarten darstellen. Danach wird das sich intelligent verhaltende Zielobjekt besprochen, was ebenfalls ein zentraler Punkt der späteren Analyse (in Kapitel 5.2.2) sein wird. Am Ende wird noch ein nur für das schwierige Szenario benutzte Zielobjekt vorgestellt, das nur in dieselbe Richtung läuft (siehe Kapitel 2.5.5).

2.4 Grundsätzliche Algorithmen der Agenten

Im folgenden werden Algorithmen besprochen, die auf einfachen Heuristiken basieren. Dadurch wird es einfacher werden, die Qualität der lernenden Algorithmen einordnen zu können. Wesentliches Merkmal im Vergleich zu auf XCS basierenden Algorithmen ist, dass sie statische, handgeschriebene Regeln benutzen und den Erfolg oder Misserfolg ihrer Aktionen ignorieren, d.h., dass sie nicht lernen und ihre Regeln während eines Laufs nicht anpassen.

Der Algorithmus mit zufälliger Bewegung in Kapitel 2.4.1 dient zum Vergleich, Algorithmen mit schlechterer Qualität können verworfen werden. Der Algorithmus mit einfacher Heuristik in Kapitel 2.4.2 läuft einfach auf das Ziel zu, wenn es in Sicht ist und dient als Vergleich für rein lokale Strategien, während der Algorithmus mit intelligenter Heuristik in Kapitel 2.4.3 zusätzlich noch versucht, Abstand zu anderen Agenten zu halten.

2.4.1 Algorithmus mit zufälliger Bewegung

Bei diesem Algorithmus wird in jedem Schritt eine zufällige Aktion ausgeführt. Jegliche Sensordaten werden dabei ignoriert. Programm A.6 zeigt den zugehörigen Quelltext.

2.4.2 Algorithmus mit einfacher Heuristik

Bei diesem Algorithmus wird in jedem Schritt geprüft, ob sich das Zielobjekt in Sicht befindet. Ist dies der Fall, dann bewegt sich ein Agent mit dieser Heuristik auf das Zielobjekt zu, andernfalls führt er eine zufällige Aktion aus. Abbildung 2.9 zeigt eine Beispielsituation bei der sich das Zielobjekt (Stern) im Süden befindet, der Agent mit einfacher Heuristik die anderen Agenten ignoriert und sich auf das Ziel bewegen möchte. Programm A.7 zeigt den zugehörigen Quelltext.

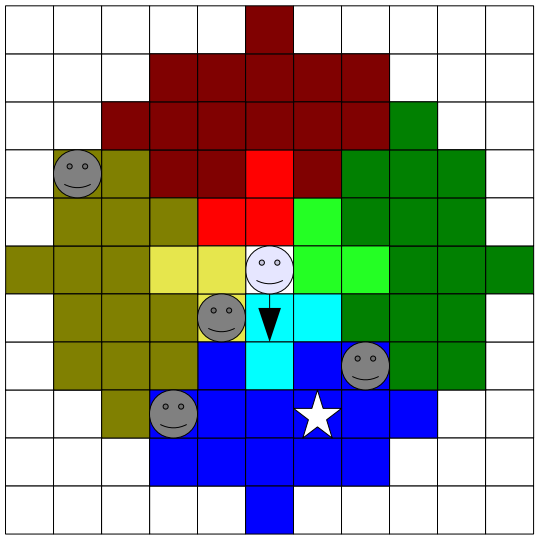

Abbildung 2.9: Agent mit einfacher Heuristik: Sofern es sichtbar ist bewegt sich der Agent auf das Zielobjekt zu.

2.4.3 Algorithmus mit intelligenter Heuristik

Ist das Zielobjekt in Sicht, verhält sich diese Heuristik wie die einfache Heuristik. Andernfalls wird versucht, anderen Agenten auszuweichen, um ein möglichst breit gestreutes Netz aus Agenten aufzubauen. In der Implementierung heißt das, dass unter allen Richtungen, in denen kein anderer Agent gesichtet wurde, eine Richtung zufällig ausgewählt wird. Falls alle Richtungen belegt oder alle frei sind, wird dagegen aus allen Richtungen eine zufällig ausgewählt. In Abbildung 2.10 ist das Zielobjekt nicht im Sichtbereich des Agenten, somit wählt dieser eine Richtung, in der die Sensoren keine Agenten anzeigen, in diesem Fall Norden. Programm A.8 zeigt den zugehörigen Quelltext.

2.5 Typen von Zielobjekten

Neben den Agenten kann auch das Zielobjekt anhand von Sensordaten über die eigene Aktion entscheiden und entspricht somit im Wesentlichen entspricht einem Agenten. Außerdem kann sich das Zielobjekt aber in einem Schritt u.U. um mehr als ein Feld bewegen, was von der durch das Szenario festgelegte Geschwindigkeit des Zielobjekts abhängt. Die Geschwindigkeit kann auch gebrochene Werte annehmen, wobei dann der gebrochene Rest die Wahrscheinlichkeit angibt, einen weiteren Schritt durchzuführen. Beispielsweise würde

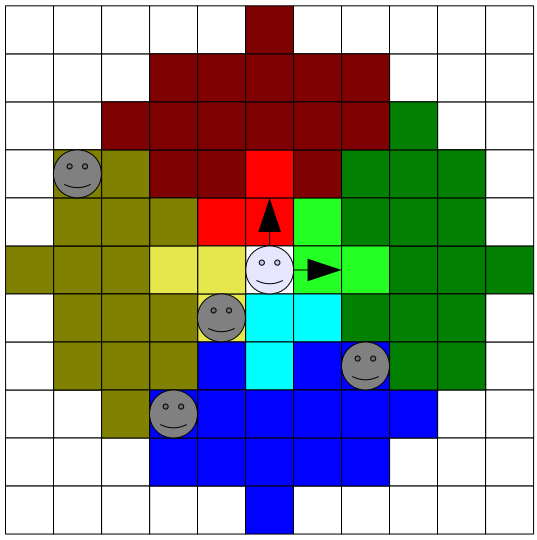

Abbildung 2.10: Agent mit intelligenter Heuristik: Falls das Zielobjekt nicht sichtbar ist, bewegt sich der Agent von anderen Agenten weg.

eine Geschwindigkeit 1,4 in 40% der Fälle zu zwei Schritten und in 60% der Fälle zu einem einzigen Schritt führen. Die Auswertung der Bewegungsgeschwindigkeit wird relevant in Kapitel 2.7, bei der Reihenfolge der Ausführung der Aktionen der Objekte.

Falls dem Algorithmus kein freies Feld zur Verfügung steht, gilt für alle Bewegungen des Zielobjekts, dass ein freies Feld in der Nähe ausgewählt zufällig ausgewählt wird und das Zielobjekt dorthin springt. Dies ist einem Neustart einer Probleminstanz (siehe Kapitel 2.6.2) ähnlich. Dieser Ablauf ist notwendig, um eine Verfälschung des Ergebnisses zu verhindern, welche eintreten kann, wenn Agenten oder unbewegliche Hindernisse alle vier Bewegungsrichtungen des Zielobjekts blockieren.

Zu beachten ist hier, dass auch der Sprung selbst eine Verfälschung darstellen kann, insbesondere wenn in einem Durchlauf viele Sprünge durchgeführt werden. In diesem Fall sollte man deshalb das Ergebnis verwerfen und z.B. andere *random seed* Werte oder einen anderen Algorithmus benutzen. Sofern nicht anders angegeben, ist der Anteil solcher Sprünge jeweils unter 0,1% und wird ignoriert.

2.5.1 Typ „Zufälliger Sprung“

Ein Zielobjekt dieses Typs springt zu einem zufälligen Feld auf dem Torus. Ist das Feld besetzt, wird der Sprung wiederholt, bis ein freies Feld gefunden wurde. Mit dieser Einstellung kann die Abdeckung des Algorithmus geprüft werden, d.h. inwieweit die Agenten jeweils außerhalb der Überwachungsreichweite anderer Agenten bleiben. Eine Anpassung an die Bewegung des Zielobjekts ist hier wenig hilfreich, da ein Agent nicht einmal davon ausgehen kann, dass sich das Zielobjekt in der Nähe seiner Position der letzten Zeiteinheit befindet.

2.5.2 Typ „Zufällige Bewegung“

Ein Zielobjekt dieses Typs verhält sich so wie ein Agent mit dem Algorithmus mit zufälliger Bewegung (siehe Kapitel 2.4.1). Sind alle möglichen Felder belegt, wird, wie oben beschrieben, auf ein zufälliges Feld gesprungen.

2.5.3 Typ „Einfache Richtungsänderung“

Dieser Typ eines Zielobjekts zieht zunächst nur diejenigen Richtungen in Betracht, in denen sich direkt angrenzend kein Hindernis befindet. Diese Erweiterung der Fähigkeiten der Sensoren wird gewählt, damit das Zielobjekt nicht längere Zeit an Hindernissen hängen bleibt. Anschließend verwirft er die Richtung, die der im letzten Schritt gewählten entgegengesetzt ist. Von den verbleibenden maximal drei Richtungen wird schließlich eine zufällig ausgewählt. Sind alle drei Richtungen versperrt, wird in die entgegengesetzte Richtung zurückgegangen, sind alle vier Richtungen versperrt, wird, wie oben beschrieben, auf ein zufälliges Feld gesprungen.

In Abbildung 2.11 sind alle Felder grau markiert, die das Zielobjekt innerhalb von zwei Schritten erreichen kann, nachdem es sich einmal nach Norden bewegt hat.

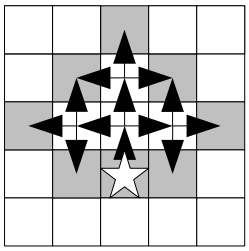

Abbildung 2.11: Zielobjekt macht pro Schritt maximal eine Richtungsänderung.

2.5.4 Typ „Intelligentes Verhalten“

Neben der Eigenschaft, Hindernissen auszuweichen, besitzt dieses Zielobjekt die Eigenschaft, Agenten zu erkennen und ihnen auszuweichen. Das Zielobjekt versucht also bei der Auswahl der Aktion möglichst die Aktion zu wählen, bei der es außerhalb der Sichtweite der Agenten bleibt. Dazu werden alle Richtungen gestrichen, in denen ein Agent sich innerhalb der Überwachungsreichweite befindet. Außerdem werden von den verbleibenden Richtungen mit 50%-iger Wahrscheinlichkeit diejenigen Richtungen gestrichen, in denen sich ein Agent in Sichtweite befindet. Falls alle Richtungen gestrichen worden sind, bewegt sich das Zielobjekt zufällig. Falls alle Richtungen blockiert sind, springt es, wie in den anderen Varianten auch, auf ein zufälliges Feld in der Nähe.

In Abbildung 2.12 wird die Richtung Süden gestrichen, da sich dort ein Agent in Überwachungsreichweite befindet. Die Richtungen Westen und Norden werden jeweils mit 50%-iger Wahrscheinlichkeit gestrichen, da sich dort Agenten in Sichtweite befinden. Nur Richtung Osten wird als Möglichkeit sicher übrig bleiben.

2.5.5 Typ „Beibehaltung der Richtung“

Ein Zielobjekt dieses Typs versucht immer in Richtung Norden zu gehen. Ist das Zielfeld blockiert, wählt es ein angrenzendes Feld im Westen oder Osten zufällig aus, das nicht besetzt ist. Anzumerken ist, dass dies eine zusätzliche Fähigkeit darstellt, d.h., das Zielobjekt kann feststellen, ob sich direkt angrenzend ein Hindernis im Norden befindet. Im Unterschied dazu können Agenten, was die Distanz betrifft, keine Informationen darüber besitzen.

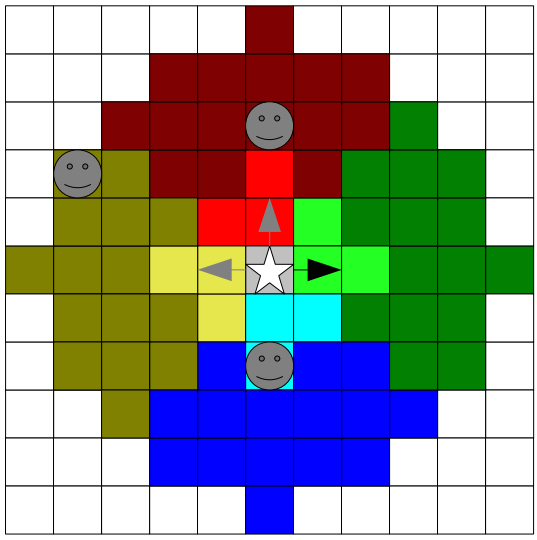

Abbildung 2.12: Ein sich intelligent verhaltendes Zielobjekt weicht Agenten aus.

In Abbildung 2.13 sind drei Situationen dargestellt: Ein wiederholtes Hin- und Herlaufen neben den Hindernissen, der Gang links um die Hindernisse herum und der Gang rechts um die Hindernisse herum.

Diese Art von Zielobjekt wird im schwierigen Szenario benutzt, um den Bereich, den das Zielobjekt überquert, möglichst gering zu halten, aber es auch nicht stehen zu lassen.

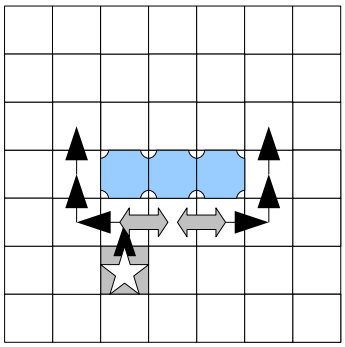

Abbildung 2.13: Bewegungsform „Beibehaltung der Richtung“: Zielobjekt bewegt sich immer nach Norden, wenn dies möglich.

2.6 Parameter und Statistiken einer Simulation

Kombiniert man nun das Szenario, die Agenten und das Zielobjekt, kann man eine Simulation laufen lassen. Im Folgenden werden nun wesentliche Elemente besprochen, die den Ablauf einer solchen Simulation betreffen. In Kapitel 2.6.1 werden allgemeine Parameter wie Experiment- und Schrittzahl angesprochen. Kapitel 2.6.2 schließt dann mit der Beschreibung einer Probleminstanz an, die in einem Experiment aufgerufen werden. In Kapitel 2.6.3 wird dann die Formel zur Berechnung der Abdeckung vorgestellt und in Kapitel 2.6.4 wird schließlich auf die Bestimmung der Qualität eines Algorithmus in einem Szenario eingegangen.

2.6.1 Allgemeines

Die Statistiken werden jeweils über einen Lauf von 10 Experimenten mit jeweils 10 Probleminstanzen (siehe Kapitel 2.6.2) ermittelt und gemittelt. Abbildung 2.14 zeigt, dass 10 Experimente genügen. Dabei wurde jeweils die Varianz von 10 Durchläufen mit aufsteigender Anzahl von Experimenten im Säulenszenario mit sich zufällig bewegenden Agenten berechnet und jeder der 10 Durchläufe wurde mit einem zufälligen *random seed* Wert gestartet. Ab einem Wert von etwa 8 Experimenten fällt die Varianz unter 1%, was für diese Arbeit als ausreichend erscheint.

2.6.2 Definition einer Probleminstanz

Eine einzelne Probleminstanz entspricht hier einem Torus mit einer bestimmten Anfangsbelegung mit bestimmten Objekten und bestimmten Parametern zur Sichtbarkeit, auf dem die Simulation über eine bestimmte Anzahl von Schritten abläuft. Die Anfangsbelegung des Torus ist über einen *random seed* Wert bestimmt. Dieser wird für jede Probleminstanz mit einem neuen Wert initialisiert, der sich aus der Nummer des Experiments und der Nummer der Probleminstanz berechnet. Die Probleminstanzen sind also untereinander unterschiedlich, jedoch mit anderen Testdurchläufen mit einer anderen Konfiguration vergleichbar. Soweit nicht anders angegeben, werden hier Probleminstanzen der Größe 16x16 Felder betrachtet und laufen über 2.000 Schritte. In den Tabellen angegebenen Werte sind jeweils auf zwei Stellen gerundet.

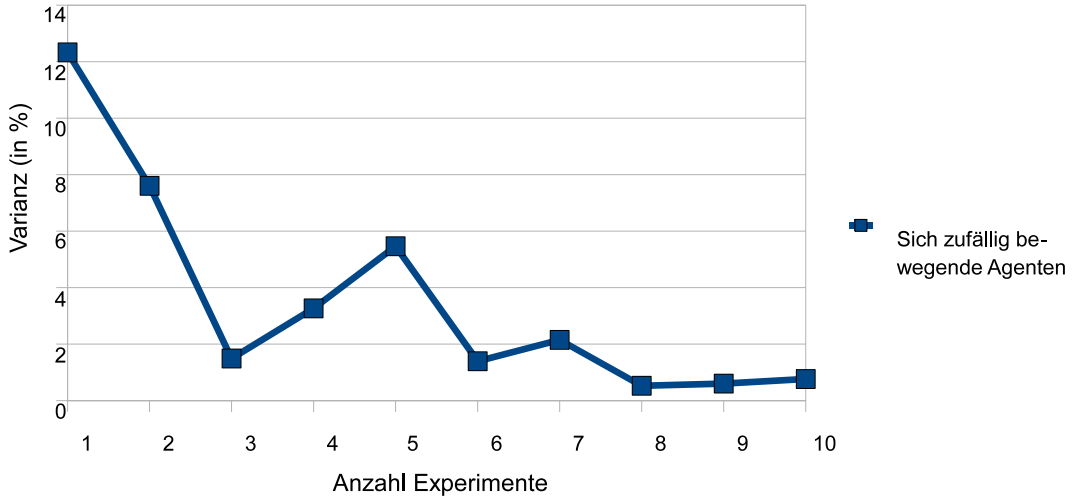

Abbildung 2.14: Varianz der Testergebnisse bei unterschiedlicher Anzahl von Experimenten (Säulenszenario, Zielobjekt mit einfacher Richtungsänderung und Geschwindigkeit 2, 8 Agenten mit zufälliger Bewegung)

Während eines Testlaufs wird eine ganze Reihe von statistischen Merkmalen erfasst. Wesentliches Merkmal zum Vergleich der Algorithmen ist der Wert der Qualität (siehe Kapitel 2.6.4), weitere Merkmale dienen zur Erklärung, warum z.B. ein Algorithmus bei einem Durchlauf schlechte Ergebnisse lieferte, oder dienten zum Testen und Finden von Fehlern oder Schwächen des Simulationsprogramms.

Die wesentlichen Merkmale sind:

1. Anteil von Sprüngen des Zielobjekts (siehe Kapitel 2.5), Durchläufe mit hohen Werten sollten verworfen werden,
2. Anteil von blockierten Bewegungen der Agenten,
3. Halbzeitqualität (siehe Kapitel 2.6.4) (größere Unterschiede zur ermittelten Qualität deuten auf Potential mit höherer Schrittzahl hin) sowie
4. Abdeckung (siehe Kapitel 2.6.3) sowie

2.6.3 Abdeckung

Die theoretisch maximal mögliche Anzahl an Feldern, die die Agenten innerhalb ihrer Überwachungsreichweite zu einem Zeitpunkt abdecken können, entspricht der Zahl der Agenten multipliziert mit der Zahl der Felder, die ein Agent in seiner Überwachungsreichweite haben kann. Ist dieser Wert größer als die Gesamtzahl aller freien Felder, wird stattdessen dieser Wert benutzt. Teilt man nun die Anzahl der momentan tatsächlich überwachten Felder durch die eben ermittelte maximal mögliche Anzahl an überwachten Felder, dann erhält man die Abdeckung.

Bezeichne n die Anzahl der Agenten, f die Anzahl der freien Felder, u die Anzahl der Felder die ein Agent zu einem Zeitpunkt maximal gleichzeitig überwachen kann und T die momentan überwachten Felder, dann berechnet sich die Abdeckung $a_n(T)$ wie folgt:

$$a_n(T) = \frac{T}{min(nu, f)}$$

2.6.4 Qualität eines Algorithmus

Die Messung der Qualität eines Algorithmus bezieht sich in dieser Arbeit auf die Betrachtung des Verhaltens des Algorithmus über einen ganzen Testlauf hinweg. Messungen finden also nicht nur z.B. am Ende oder nur während bestimmten Probleminstanzen, wie dies bei der Standardimplementierung von XCS der Fall ist.

Für die Darstellung der Qualität des Algorithmus werden die einzelnen Probleminstanzen nacheinander aufgereiht, sodass sich z.B. bei 500 Schritten pro Probleminstanz und 10 Problemen pro Experiment eine Gesamtzahl von 5.000 Schritten ergibt. Für jeden einzelnen Schritt wird die Qualität dann über das Mittel aller Experimente berechnet.

Zur Darstellung des zeitlichen Verlaufs der Qualität kann es aufgrund der relativ niedrigen Anzahl von Experimenten pro Durchlauf außerdem sinnvoll sein, den gleitenden Durchschnitt über mehrere Schritte anstatt die mittlere Qualität zu einem Zeitpunkt darzustellen. Dies erlaubt eine übersichtlichere Darstellung.

In den in dieser Arbeit verwendeten Darstellung des zeitlichen Verlaufs der Qualität wird zu Beginn jeweils der Durchschnitt aller bisherigen Schritte und nach dem max_s-ten

Schritt jeweils der Durchschnitt über die letzten max_s Schritte dargestellt werden.

Desweiteren wird von einer Halbzeitqualität die Rede sein, hierbei werden bei einem Probleminstanz jeweils nur die letzten $\frac{max_s}{2}$ Schritte für die Gesamtqualität berücksichtigt. Damit kann man beispielsweise untersuchen, ob sich der jeweilige Algorithmus nach dieser Zahl von Schritten bereits an das Problem angepasst hat oder ob eine Erhöhung der Schrittzahl die Qualität verbessern könnte. Die Betrachtung dieser Statistik hat insbesondere beim Testen es sehr einfach gemacht, festzustellen, ob ein Algorithmus noch Potential hat, d.h. ob eine Erhöhung der Schrittzahl die Qualität weiter steigern würde.

- Sei Hilfsvariable $q_{eps} = 1$ wenn sich das Zielobjekt in der Überwachungsreichweite eines beliebigen Agenten während Experiment e in Probleminstanz p im Schritt s befindet und $q_{eps} = 0$ sonst (siehe auch Kapitel 3.4).
- Bestimme max_e die Anzahl der durchgeführten Experimente,
- max_p die Anzahl der Probleminstanzen pro Experiment und
- max_s die Anzahl der Schritte pro Experiment.

Dann berechnet sich die Qualität q_i für einen Schritt i wie folgt:

$$q_i = \frac{\sum_{e=0}^{max_e} q_{ei}}{max_e} \text{ , mit } q_{ei} = q_{e(\frac{i}{max_s})(i \bmod max_s)}$$

Die Gesamtqualität q berechnet sich dann mit:

$$q = \frac{\sum_{i=0}^{max_p max_s} q_i}{max_p max_s}$$

2.7 Ablauf der Simulation

Die Simulation selbst läuft in ineinander geschachtelten Schleifen ab. Jede Konfiguration, die in den abgedruckten Programmen in Kapitel A jeweils über die globale Variable *Configuration* angesprochen wird, wird über eine Reihe von Experimenten getestet (10 soweit nicht anders angegeben). Für einen Test wird die Funktion *doOneMultiStepExperiment()* (siehe Programm A.1) mit der aktuellen Nummer des Experiments als Parameter

aufgerufen. In der Funktion wird ein neuer *random seed* Wert initialisiert, der Torus auf den Startzustand gesetzt und schließlich die eigentliche Probleminstanz mit der Funktion *doOneMultiStepProblem()* aufgerufen, welche in Programm A.2 abgebildet ist. Dort werden in einer Schleife alle Schritte durchlaufen und jeweils die Objekte abgearbeitet.

In welcher Reihenfolge dies geschieht, wird im Folgenden geklärt. Zusammenfassend ist zu sagen, dass zuerst die aktuelle Qualität und die aktuellen Sensordaten bestimmt werden. Daraus ermittelt jeder Agent die Bewertung für den letzten Schritt und bestimmt eine neue Aktion. Haben Agenten und das Zielobjekt diese Schritte abgeschlossen, werden ihre ermittelten Aktionen in zufälliger Reihenfolge ausgeführt.

In Kapitel 2.7.1 wird zuerst der äußere Ablauf einer einzelnen Probleminstanz besprochen. Da sich Zielobjekt und Agenten in unterschiedlichen Geschwindigkeiten bewegen können, wird in Kapitel 2.7.2 die Reihenfolge der Bewegungen in der Simulation diskutiert. Kapitel 2.7.3 geht dann auf die Frage ein, wann die Qualität gemessen werden soll und Kapitel 2.7.4 geht auf die Frage ein, wann der *base reward* bestimmt wird. Abschluss macht dann eine Zusammenfassung des Ablaufs einer Simulation in Kapitel 2.7.5.

2.7.1 Berechnung einer Probleminstanz

Bei der Berechnung einer einzelnen Probleminstanz in der Funktion *doOneMultiStepProblem()* stellt sich die Frage nach der Genauigkeit und der Reihenfolge der Abarbeitung, da die Simulation nicht parallel, sondern schrittweise auf einem diskreten Torus abläuft. Dies kann dazu führen, dass je nach Position in der Liste abzuarbeitender Agenten die Informationen über die Umgebung unterschiedlich alt sind. Die Frage ist deshalb, in welcher Reihenfolge folgende Punkte ausgeführt werden: Ermittlung und Auswertung der Sensordaten, Bewegung der Agenten, Bewertung der Agenten und Messung der Qualität.

Da eine Aktion auf Basis der Sensordaten ausgewählt wird, ist die erste Restriktion, dass eine Aktion nach der Verarbeitung der Sensordaten stattfinden muss. Da außerdem Aktionen bewertet werden sollen, also jeweils der Zustand nach der Bewegung mit dem gewünschten Zustand verglichen werden soll, ist die zweite Restriktion, dass die Bewertung einer Aktion nach dessen Ausführung stattfinden muss.

Unter diesen Restriktionen ergeben sich folgende zwei Möglichkeiten:

1. Für alle Agenten werden erst einmal die neuen Sensordaten erfasst und sich für eine Aktion entschieden. Sind alle Agenten abgearbeitet, werden die Aktionen ausgeführt.

2. Die Agenten werden nacheinander abgearbeitet, es werden jeweils neue Sensordaten erfasst, sich für eine neue Aktion entschieden und diese sofort ausgeführt.

Beim zweiten Punkt ergibt sich der Umstand, dass später abgearbeitete Agenten aktuellere Informationen über die Umwelt besitzen. Insbesondere schlägt bei den später ausgeführten Agenten eine Aktion mit geringerer Wahrscheinlichkeit fehl, sofern Hindernisse in der Bewegung beachtet werden. Umgekehrt können früh ausgeführte Agenten eher gute Positionen besetzen. Da dies ein schwierig zu beurteilenden Faktor in die Untersuchungen einbringt, wird für die Simulation die erste Möglichkeit gewählt, jeder Agent entscheidet seine Aktion also auf Basis von Sensordaten vom selben, gemeinsamen Zeitpunkt.

2.7.2 Reihenfolge bei unterschiedlichen Geschwindigkeiten

Bezüglich der Bewegung ergibt sich eine weitere Frage, nämlich wie unterschiedliche Bewegungsgeschwindigkeiten behandelt werden sollen. Zwar haben alle Agenten eine Einheitsgeschwindigkeit von einem Feld pro Zeiteinheit, jedoch kann sich das Zielobjekt je nach Szenario gleich eine ganze Anzahl von Feldern bewegen (siehe auch Kapitel 2.5).

Die Entscheidung fiel hier auf eine zufällige Verteilung. Kann sich das Zielobjekt um n Schritte bewegen, so wird seine Bewegung in n Einzelschritte unterteilt, die nacheinander mit zufälligen Abständen (d.h. Bewegungen anderer Agenten) ausgeführt werden.

Schließlich ist zu klären, wie das Zielobjekt diese weiteren Schritte festlegen soll. Hier wird ein Sonderfall eingeführt, sodass das Zielobjekt in einer Zeiteinheit mehrmals (n-mal) neue Sensordaten erfassen und sich für eine neue Aktion entscheiden kann.

2.7.3 Messung der Qualität

Bei der Analyse der gerade betrachteten Reihenfolge stellt sich die Frage, wann man die Qualität des Algorithmus messen sollte. Eine Messung nach der Bewegung des Zielobjekts würde bedeuten, dass sich dieses vor der Messung noch optimal positionieren kann. Eine Messung vor der Bewegung bedeutet, dass die Agenten sich relativ zum Zielobjekt noch optimal positionieren können.

Eine Messung nach der Bewegung des Zielobjekts bedeutet, dass ein Agent ein Zielobjekt, das sich gerade aus der Überwachungsreichweite herausbewegt, nicht mehr überwachen kann und somit eine tendenziell niedrigere Qualität gemessen wird. Dies gilt insbesondere bei sich intelligent verhaltenden Zielobjekten.

Da ein wesentlicher Bestandteil dieser Arbeit die Kollaboration anstatt dem dauernden Verfolgen des Zielobjekts ist, wird ein Bewertungskriterium sein, inwieweit der Einfluss des Zielobjekts minimiert werden kann. Realistisch gesehen, findet die Bewegung des Zielobjekts gleichzeitig mit allen anderen Agenten statt. Die Qualität wird somit nach der Bewegung des Zielobjekts gemessen.

2.7.4 Reihenfolge der Ermittlung des *base reward* Werts

Hat man sich für die Art entschieden, wie die Qualität des Algorithmus bewertet wird, kann man damit fortfahren, sich zu überlegen, wie der einzelne Agent aus der lokalen Sichtweise heraus bestimmt, wie gut sein Verhalten war.

Die bisher vorgestellten Agenten in Kapitel 2.4 sind nicht lernfähig, d.h. es gibt keine Rückkopplung zwischen erfassten Sensordaten und den Heuristiken. Die Agenten, die im Kapitel 3 vorgestellt werden, besitzen dagegen eine solche Rückkopplung. Deshalb stellt sich die Frage, wann geprüft werden soll, ob das Zielobjekt in Überwachungsreichweite ist und wann sich somit ein sogenannter *base reward* Wert ergeben soll. Wesentliche Punkte hierbei sind, dass der Algorithmus sich anhand der Sensordaten selbst bewertet und pro Schritt die Sensordaten nur einmal erhoben werden. Letzteres folgt aus der Standardimplementierung von XCS, wo der *base reward* Wert jeweils genau einer Aktion zugeordnet ist.

Aus Standardimplementierung wird außerdem übernommen, dass der *base reward* Wert von binärer Natur („Zielobjekt in Überwachungsreichweite“ oder „Zielobjekt nicht in Überwachungsreichweite“) ist. Deshalb werden Zwischenzustände für diesen Wert, die sich aus der mehrfachen Bewegung des Zielobjekts ergeben könnten (z.B. „War zwei von drei Schritten in Überwachungsreichweite“ $\Rightarrow \frac{2}{3}$ *base reward*), ausgeschlossen.

Für den *base reward* Wert ergibt sich somit entweder die Möglichkeit, die einzelnen *base reward* Werte jeweils direkt nach der Ausführung einer einzelnen Aktion oder nach Ausführung aller Aktionen der Agenten und des Zielobjekts zu ermitteln. Werden die *base reward* Werte sofort ermittelt, dann bezieht sich der Wert auf die veralteten Sensordaten vor der Aktion, die Aktion selbst würde bei der Ermittlung des *base reward* Werts also ignoriert werden. Würden die Werte erst nach Ausführung aller Aktionen bestimmt, müsste man bis zum nächsten Schritt warten, bis neue Sensordaten ermittelt worden wären.

2.7.5 Zusammenfassung des Simulationsablaufs

Im Folgenden ist der Ablauf aller Agenten (inklusive des Zielobjekts) dargestellt. Anzumerken ist, dass für das Zielobjekt zu Beginn in Schritt 2 und 3 nur der erste Schritt berechnet wird. Falls die Geschwindigkeit des Zielobjekts größer als 1 ist, werden in Schritt 5 nach der Ausführung der Aktion direkt neue Sensordaten erfasst und eine neue Aktion berechnet (siehe Kapitel 2.7).

1. Bestimmung der aktuellen **Qualität**,
2. Erfassung der **Sensordaten** aller Agenten und des Zielobjekts,
3. Bestimmung der jeweiligen ***base reward* Werte** für die einzelnen Objekte für den letzten Schritt (bezieht sich auf lernende Agenten),
4. Aktualisierung der Regeln anhand des *base reward* Werts (bezieht sich auf lernende Agenten) sowie
5. **Wahl der Aktion** anhand der Regeln des jeweiligen Agenten bzw. Zielobjekts.
6. **Ausführung der Aktion** aller (in zufälliger Reihenfolge, Zielobjekt wiederholt u.U. Schritte 2 und 3 zwischen zwei eigenen Bewegungen).

Kapitel 3

XCS

If everyone is thinking alike, then somebody isn't thinking.
George S. Patton

Im vorangegangenen Kapitel wurde das Szenario, die Simulation und Agenten mit Heuristiken besprochen. Nun wird es um lernende Agenten gehen, die jeweils ein unabhängiges, sogenannten *eXtended Classifier System* (XCS), besitzen, welches einem speziellen *learning classifier system* (LCS) entspricht. Ein LCS ist ein evolutionäres Lernsystem, das aus einer Reihe von *classifier* Regeln besteht, die zusammen ein sogenanntes *classifier set* bilden (siehe Kapitel 3.1). Eine allgemeine Einführung in LCS findet sich z.B. in [But06a], für eine umfassende Beschreibung von XCS wird auf [But06b] verwiesen.

Im Wesentlichen besteht ein XCS aus folgenden Elementen:

1. Einer Menge aus Regeln, sogenannte *classifier* (siehe Kapitel 3.1), die zusammen ein *classifier set* bilden,
2. einem Mechanismus zur Auswahl einer Aktion aus dem *classifier set* (siehe Kapitel 3.5),
3. einem Mechanismus zur Zusammenfassung aller *classifier* aus dem *classifier set* mit gleicher Aktion zu einer *action set* Liste,
4. einem Mechanismus zur Evolution der *classifier* (mittels genetischer Operatoren, siehe Kapitel 3.3.4) sowie
5. einem Mechanismus zur Bewertung der *classifier* (mittels *reinforcement learning*, siehe Kapitel 3.4).

Während die ersten drei Punkte bei allen hier vorgestellten XCS Varianten identisch sind, gibt es wesentliche Unterschiede bei der Bewertung der *classifier*. Diese werden gesondert in Kapitel 4 im Einzelnen besprochen. Im Folgenden werden nun Punkt 1, 2 und 3 näher betrachtet. Im Folgenden konzentrieren sich die Ausführungen auf den für das Verständnis der in Kapitel 4 vorgestellten XCS Varianten relevanten Teil.

3.1 Classifier

Ein *classifier* besteht aus einer Anzahl von Variablen, die anhand der in Kapitel 5.5 aufgelisteten Werte initialisiert werden. Wesentliche Teile sind der *condition* Vektor (Kapitel 3.1.1) und der *action* Wert (Kapitel 3.1.2). Alle restlichen Variablen dienen zur Berechnung der Wahrscheinlichkeit, mit welcher der *classifier* ausgewählt wird.

3.1.1 Der *condition* Vektor

Der *condition* Vektor gibt die Kondition an, in welchen Situationen der zugehörige *classifier* ausgewählt werden kann, d.h. welche Sensordatensätze der jeweilige *classifier* erkennt. Der Aufbau des Vektors (siehe Abbildung 3.1) entspricht dem Vektor der über die Sensoren erstellt wird (siehe Kapitel 2.3.3). Eine wesentliche Erweiterung des *condition* Vektors stellen sogenannte Platzhalter dar, die es dem *condition* Vektor erlauben, mehrere verschiedene Sensordatensätze zu erkennen (siehe Kapitel 3.2).

$$\underbrace{z_{s_N} z_{r_N} z_{s_O} z_{r_O} z_{s_S} z_{r_S} z_{s_W} z_{r_W}}_{\textit{Erste Gruppe (Zielobjekt)}} \underbrace{a_{s_N} a_{r_N} a_{s_O} a_{r_O} a_{s_S} a_{r_S} a_{s_W} a_{r_W}}_{\textit{Zweite Gruppe (Agenten)}} \underbrace{h_{s_N} h_{r_N} h_{s_O} h_{r_O} h_{s_S} h_{r_S} h_{s_W} h_{r_W}}_{\textit{Dritte Gruppe (Hindernisse)}}$$

Abbildung 3.1: Einteilung des *condition* Vektors in drei Gruppen

3.1.2 Der *action* Wert

Wird ein *classifier* ausgewählt, wird eine bestimmte Aktion ausgeführt, die durch den *action* Wert determiniert ist. Im Rahmen dieser Arbeit entsprechen diese Aktionsmöglichkeiten den vier Bewegungsrichtungen, die in Kapitel 2.3.4 besprochen wurden.

3.1.3 Der *fitness* Wert

Der *fitness* Wert soll die Genauigkeit des *classifier* repräsentieren und wird über die Zeit hinweg sukzessive an die beobachteten *reward* Werte angepasst. Hier erstreckt sich der Wertebereich zwischen 0,0 (minimale Genauigkeit) und 1,0 (maximale Genauigkeit). Insbesondere eines der ersten Werke zu XCS [Wil95] beschäftigte sich mit diesem Aspekt der Genauigkeit.

3.1.4 Der *reward prediction* Wert

Der *reward prediction* Wert des *classifier* stellt die Höhe des *reward* Werts dar, von dem der *classifier* erwartet, dass er ihn bei der nächsten Bewertung erhalten wird.

3.1.5 Der *reward prediction error* Wert

Der *reward prediction error* Wert soll die durchschnittliche Differenz zwischen dem *reward prediction* Wert und dem tatsächlichen *reward* Wert repräsentieren. U.a. auf Basis dieses Werts wird der *fitness* Wert des *classifier* angepasst.

3.1.6 Der *experience* Wert

Mit dem *experience* Wert des *classifier* wird die Anzahl repräsentiert, wie oft ein *classifier* aktualisiert wurde, also wieviel Erfahrung der *classifier* sammeln konnte. Im Wesentlichen dient dieser Wert als Entscheidungshilfe, ob auf die anderen Werte des *classifier* vertraut werden kann bzw. ob der *classifier* als „unerfahren“ gilt und somit z.B. bei Löschung und Subsummation gesondert behandelt werden muss.

3.1.7 Der *numerosity* Wert

Der *numerosity* Wert gibt an, wieviele andere, sogenannte *micro classifier* sich in dem jeweiligen *classifier* befinden. Durch Subsummation (siehe Kapitel 3.2.2 und 3.3.4) können *classifier* eine Rolle als *macro classifier* spielen, d.h. *classifier* die andere *classifier*

in sich beinhalten. Bezüglich der Implementierung sei Kapitel A.4 zu erwähnen, da dort, verglichen mit der originalen Implementierung, einige Änderungen vorgenommen wurden.

3.2 Vergleich des *condition* Vektors mit Sensordaten

Ein Vergleich zwischen zwei Vektoren ist über den Vergleich ihrer jeweiligen Einzelelemente gegeben. Hier werden nun die Zustände der Einzelelemente im *condition* Vektors erweitert und besprochen, wie dann ein Vergleich durchgeführt wird.

Neben den zu den Sensordaten korrespondierenden Werten 0 und 1 soll es noch einen dritten Zustand als Teil des *condition* Vektors geben, den Platzhalter „#“. Dieser zeigt an, dass beim Vergleich zwischen dem *condition* Vektor und den Sensordaten diese Stelle ignoriert wird. Eine Stelle im *condition* Vektor mit Platzhalter gilt dann also als äquivalent zur korrespondierenden Stelle in den Sensordaten, egal ob sie mit 0 oder 1 belegt ist. Ein Vektor, der ausschließlich aus Platzhaltern besteht, würde somit bei der Auswahl immer in Betracht gezogen werden, da er auf alle möglichen Kombinationen der Sensordaten passt. Umgekehrt können dadurch bei der Auswahl der *classifier* mehrere *classifier* auf einen gegebenen Sensordatenvektor passen. Diese bilden dann die sogenannte *match set* Liste, aus welcher dann der eigentliche *classifier* ausgewählt wird (siehe Kapitel 3.5).

Im Folgenden wird zum einen untersucht, welche Sensordatensätze ein *condition* Vektor erkennt (siehe Kapitel 3.2.1), und zum anderen, auf welche Weise man ähnliche *classifier* zusammenlegen kann (siehe Kapitel 3.2.2).

3.2.1 Erkennung von Sensordatenpaaren

Beim Vergleich der Sensordaten und Daten aus dem *condition* Vektor werden immer jeweils zwei Paare herangezogen. In Kapitel 2.3 wurde erwähnt, dass der Fall (0/1) in den Sensordaten nicht auftreten kann. Damit der *condition* Vektor nicht von vornherein ungültige Werte annehmen kann, wird durch Einführung einer Redundanz ein Datenpaar (0/1) im *condition* Vektor äquivalent zum Datenpaar (1/1) und somit auch das (0/#) äquivalent zu (#/#) sein, also beide Datenpaare die gleichen Sensordatenpaare erkennen.

Es ergeben sich also folgende Fälle:

1. Sensordatenpaar (0/0) wird erkannt von (0/0), (#, 0), (0, #), (#, #),
2. Sensordatenpaar (1/0) wird erkannt von (1/0), (#, 0), (1, #), (#, #) und
3. Sensordatenpaar (1/1) wird erkannt von (1/1), (#, 1), (1, #), (#, #), (0/1), (0/#).

Beispielsweise würden folgende Sensordaten von den folgenden *condition* Vektoren erkannt:

```
Sensordaten:
(Zielobjekt in Sicht im Norden, Agent in Sicht im Süden,
Hindernisse im Westen und Osten)
10 00 00 00 . 00 00 11 00 . 00 11 00 11

Beispiele für erkennende condition Vektoren:
10 00 00 00 . ## ## ## ## . 00 ## ## ##
## ## ## ## . ## ## #1 00 . 00 11 ## ##
#0 ## ## ## . ## ## 01 ## . ## 11 ## 11
```

3.2.2 Subsummation von *classifier*

Die Benutzung von den oben erwähnten Platzhaltern (Kapitel 3.2) erlaubt es dem XCS, mehrere *classifier* zusammenzulegen. Dadurch sinkt die Gesamtzahl der *classifier* und somit müssen Erfahrungen, die ein XCS Agent sammelt, nicht unbedingt mehrfach gemacht werden. Implizit wird dabei angenommen, dass es Situationen gibt, in denen der Gewinn, der durch Unterscheidung von zwei verschiedenen Sensordatensätzen erbracht werden kann, geringer ist, als die Ersparnis, die durch das Zusammenlegen beider *classifier* entsteht.

Besitzt ein *classifier* sowohl einen genügend großen *experience* Wert als auch einen ausreichend kleinen *reward prediction error* Wert, so kann er als sogenannter *subsumer* auftreten. Ein *subsumer* ersetzt andere *classifier* in derselben *action set* Liste, sofern der

von dem jeweiligen *classifier* gesamte abgedeckte Sensordatenbereich eine Teilmenge des von dem *subsumer* abgedeckten Bereichs ist. Der *subsumer* besitzt also an allen Stellen des *condition* Vektors entweder denselben Wert wie der zu subsummierende *classifier* oder einen Platzhalter.

3.3 Ablauf eines XCS

Ein XCS läuft wie folgt ab:

1. Vervollständigung der *classifier* Liste (*covering*, siehe Kapitel 3.3.1),
2. Auswahl auf die Sensordaten passenden *classifier* (*match set* Liste, siehe Kapitel 3.3.2),
3. Bestimmung der Auswahlart und Auswahl der Aktion (*explore*/*exploit*, siehe Kapitel 3.5) und
4. Erstellung der zur Aktion zugehörigen Liste von *classifier* (*action set* Liste, siehe Kapitel 3.3.3).

Im Folgenden werden die einzelnen Punkte besprochen.

3.3.1 Abdeckung aller Aktionen durch *covering*

Das *covering* untersucht die Menge aller *classifier* aus der letzten *match set* Liste (siehe Kapitel 3.3.2), ob für jede mögliche Aktion jeweils mindestens ein *classifier* vorhanden ist. Ist dies nicht der Fall, wird für jede fehlende Aktion ein neuer *classifier*, dessen *condition* Vektor auf den letzten Sensordatensatz passt, erstellt und in die Population eingefügt. So wird sichergestellt, dass alle Situationen und Aktionen abgedeckt sind. Ist die Populationsgröße N zu niedrig, kommt es zum *trashing*, d.h. es werden andauernd neue *classifier* erstellt; gleichzeitig müssen aber (brauchbare) alte *classifier* gelöscht werden. In Kapitel 5.5.1 in Abbildung 5.7 sieht man beispielsweise, dass dies im dortigen Szenario mindestens bis zu einer Größe von 256 regelmäßig passiert.

3.3.2 Die *match set* Liste

In der *match set* Liste werden jeweils alle *classifier* gespeichert, die den letzten Sensordatensatz erkannt haben. Sie entspricht dem *predictionArray* in der originalen Implementierung von XCS in [But00]. Dort werden außerdem Vorberechnungen zur Auswahl der nächsten Aktion durchgeführt und die Ergebnisse gespeichert, die insbesondere in Kapitel 3.5 von Bedeutung sind, die sogenannten *predictionFitnessProductSum* Werte.

3.3.3 Die *action set* Liste

In einer *action set* Liste werden jeweils alle *classifier* gespeichert, die zu diesem Zeitpunkt denselben *action* Wert besitzen wie der für die Bewegung bestimmte *classifier*. In der Standardimplementierung von XCS wird jeweils nur die letzte *action set* Liste gespeichert, während in SXCS eine ganze Reihe (bis zu *maxStackSize* Stück, siehe Kapitel 4.3) gespeichert werden.

3.3.4 Genetische Operatoren

Es werden aus der jeweiligen *action set* Liste zwei *classifier* („Eltern“) zufällig ausgewählt und zwei neue *classifier* („Kinder“) aus ihnen gebildet und in die Population eingefügt. Dabei wird mittels *two-point crossover* ein neuer *condition* Vektor generiert und der *action* Wert auf den der Eltern gesetzt. Da sie aus derselben *action set* Liste stammen, ist der Wert beider Eltern identisch. Die restlichen Werte werden standardmäßig wie in Kapitel 5.5 aufgelistet initialisiert. Wird versucht, Kinder, deren *action* Wert und *condition* Vektor identisch mit existierenden *classifier* ist, in die Population einzufügen, werden sie stattdessen subsummiert.

Da die Sensoren und somit auch der *condition* Vektor aus drei in sich geschlossenen Gruppen bestehen, werden im Unterschied zur Standardimplementierung beim *crossing over* zwei feste Stellen benutzt, die die Gruppe für das Zielobjekt, die Gruppe für Agenten und die Gruppe für feste Hindernisse voneinander trennen.

Bezeichne (z_1, a_1, h_1) bzw. (z_2, a_2, h_2) jeweils die drei Gruppen (siehe Kapitel 3.1.1) des

condition Vektors des ersten bzw. zweiten ausgewählten Elternteils, dann können für die drei Gruppen der *condition* Vektoren (z_{1k}, a_{1k}, h_{1k}) und (z_{2k}, a_{2k}, h_{2k}) der beiden Kinder folgende Kombinationen auftreten: $[(z_{1k}, a_{1k}, h_{1k}), (z_{2k}, a_{2k}, h_{2k})] = \{[(z_1, a_1, h_1), (z_2, a_2, h_2)], [(z_2, a_1, h_1), (z_1, a_2, h_2)], [(z_1, a_2, h_1), (z_2, a_1, h_2)], [(z_2, a_2, h_1), (z_1, a_1, h_2)]\}$

3.4 Bewertung der Aktionen (*base reward*)

Damit die Agenten lernen können, muss an einer Stelle eine Bewertung mit Hilfe einer sogenannten *reward* Funktion geschehen. XCS ist darauf ausgelegt, dass es eine komplette, genaue und möglichst allgemeine Darstellung einer solchen *reward* Funktion darstellt. Die *reward* Funktion läuft lokal auf jedem Agenten ab und die Bewertung, die der Agent berechnet, wird also auf Basis der eigenen Sensordaten gebildet. Die Bewertung wird im Folgenden als *base reward* Wert bezeichnet.

In Kapitel 3.4.1 wird zuerst die Bewertung mit dem *single step* Verfahren erläutert. Ein wesentlicher Bestandteil ist hier die globale Information. Entsprechend wird in Kapitel 3.4.2 die Bewertung im *multi step* Verfahren behandelt, bei begrenzter lokaler Information wird dort über mehrere Probleminstanzen hinweg eine globale Information aufgebaut. Im Gegensatz dazu steht die Bewertung bei einem Überwachungsszenario (siehe Kapitel 3.4.3). Die Frage ist hier, anhand welcher Richtlinie man eine *reward* Funktion überhaupt aufstellen soll.

3.4.1 Bewertung beim *single step* Verfahren

Bei einer Problemstellung, die mit dem *single step* Verfahren gelöst werden kann, entspricht die optimale Darstellung der *reward* Funktion durch das XCS gleichzeitig auch der Lösung des eigentlichen Problems. Beispielsweise prüft im in der Einleitung (Kapitel 1.1.1) erwähnten *6-Multiplexer* Problem die zugehörige *reward* Funktion, ob das XCS aus den 4 Datenbits anhand der 2 Steuerbits das richtige Datenbit gewählt hat, ob also das XCS so wie ein *6-Multiplexer* funktioniert. Wesentliche Voraussetzung für das *single step* Verfahren ist, dass der Agent globale Information besitzt, also in einem Schritt möglichst alle Informationen zur Lösung des Problems zur Verfügung hat, um die eigene Ausgabe zu bewerten.

3.4.2 Bewertung beim *multi step* Verfahren

Bei komplexeren Problemen, bei denen ein Agent nur lokale Informationen zur Verfügung hat, bezieht die *reward* Funktion nur eine Teilinformation der Welt in die Bewertung ein. Beispiels beim *Maze N* Problem (siehe Kapitel 1.1.2) fließen in die Entscheidung nur die angrenzenden Felder ein und die *reward* Funktion bewertet einen *classifier* „1“ beim letzten Schritt auf das Ziel und ansonsten „0“. Deshalb ist die optimale Darstellung der *reward* Funktion bei XCS in der *multi step* Variante die, dass der Aufbau eines Gesamtbilds über die Weitergabe mittels des jeweiligen *maxPrediction* Werts geschieht. Auf Basis dessen wird ein vereinfachter Gesamtweg gebildet und somit zumindest teilweise das Problem in ein *single step* Problem überführt. Vereinfacht ist er, weil Situationen und Aktionen in den *classifier set* Listen gespeichert werden und nicht Aktionsreihenfolgen und / oder Positionsangaben zusammen mit der auszuführenden Aktion.

3.4.3 Bewertung bei einem Überwachungsszenario

Der gleiche Gedankengang wie im vorangegangenen Abschnitt muss beim Überwachungsszenario ausgeführt werden. Die Darstellung des Gesamtproblems aus lokaler Information, die standardmäßig beim *multi step* Verfahren bei XCS verwendet wird, kann man zwar für das Szenario übernehmen (siehe Kapitel 4.2); die Ergebnisse, wie sie später in Kapitel 5 gezeigt werden, sind dann aber oft nicht viel besser als ein sich zufällig bewegender Agent.

Eine Alternative ist, den *base reward* Werts nicht sukzessive weiterzugeben, sondern bei jedem Auftreten eines positiven *base reward* Werts direkt alle bisherigen Aktionen (seit dem letzten Auftreten) absteigend mit dem Wert zu aktualisieren. Diese Idee wird in dann in Kapitel 4 vorgestellt und in Kapitel 5 signifikant bessere Ergebnisse erbringen.

Offen bleibt die Frage, wie die *reward* Funktion beim Überwachungsszenario aussieht. Letztlich hat ein Agent die freie Wahl, wie der *base reward* Wert aus den Sensordaten berechnet wird. Für die 24 Binärsensoren ergeben sich bis zu 2^{24} wahrnehmbare Situationen. Tatsächlich sind es weniger, da es nur ein Zielobjekt gibt und bestimmte Situationen nicht auftreten können. Weiterhin könnte jeder Situation ein individueller *base reward*

Wert zugewiesen werden, was entsprechend viele Möglichkeiten für die *reward* Funktion ergäbe. Mangels Speicherkapazität fällt eine solche Darstellung der *reward* Funktion aber von vornherein weg.

Zur Beantwortung der Frage ist deshalb eher die Betrachtung des globalen Ziels von Bedeutung. Eine Möglichkeit ist, sich anhand der in Kapitel 2.4 vorgestellten Heuristiken zu orientieren. Auch wenn diese Heuristiken nicht direkt mit einem *base reward* Wert arbeiten, bewerten sie doch jede einzelne Situation als positiv oder als negativ. Erkennt beispielsweise ein Agent mit intelligenter Heuristik andere Agenten im Sichtbereich, wird er die Richtungen, in denen sich Agenten befinden, als negativ bewerten. Umgekehrt wird er freie Richtungen als positiv bewerten. Betrachtet man die Heuristiken unter diesem Gesichtspunkt näher, dann führt dies zu folgenden Erkenntnissen:

Algorithmus mit zufälliger Bewegung Dieser Algorithmus bewertet alle Situationen identisch, er benutzt also eine konstante *reward* Funktion.

Einfache Heuristik Dieser Algorithmus bewertet jede Situation, in der das Zielobjekt in Sicht ist, als positiv.

Intelligente Heuristik Dieser Algorithmus bewertet ebenfalls jede Situation mit einem Zielobjekt in Sicht als positiv. Zusätzlich versucht ein Agent mit dieser Heuristik sich aus der Sicht anderer Agenten zu bewegen. Er bewertet also im Grunde Situationen besser, in denen mehr Richtungen frei von Agenten sind.

Auf Basis dieser Informationen und den Testergebnissen in Kapitel 5.1 ist es nun möglich, sich für eine wahrscheinlich erfolgreiche *reward* Funktion zu entscheiden.

Verwendet man die *reward* Funktion des Agenten mit zufälliger Bewegung für XCS, dann würden entsprechend die Aktionen zufällig bewertet werden. Das ergäbe, dass sich der Agent, ähnlich wie ein Agent mit zufälliger Bewegung, bewegen und eine entsprechend niedrige Qualität erzielen würde. Dieser Algorithmus soll also nicht als Vorbild dienen.

Bei den Tests ist dagegen der Agent mit intelligenter Heuristik eindeutig im Vorteil; allerdings ist es schwierig, oben beschriebene *reward* Funktion zu modellieren, da im Rahmen dieser Arbeit lediglich binäre Ausgabewerte möglich sein sollen. Um mehrere

unterschiedliche Situationen zu differenzieren, wären aber mehrere Zustände Voraussetzung. Beispielsweise könnte der *base reward* Wert anhand der Anzahl der sich in der Nähe befindlichen Agenten bestimmt werden um so eine Abstufung erreichen zu können.

Treffen kann man nur die Unterscheidung, ob gar keine Agenten in Sicht sind oder ob mindestens ein Agent in Sicht ist. Dadurch belohnt man weniger den Weg zum Ziel (d.h. die Anzahl der Agenten in Sicht verringern bzw. sich von Agenten weg zu bewegen) sondern mehr das Ziel, dass durch die intelligente Heuristik letztlich erreicht wird, selbst (keine Agenten mehr in Sicht). Trotzdem stellt dies nur eine teilweise Umsetzung um und die Ergebnisse die die intelligente Heuristik erreicht stellen somit ein oberes Limit für die mögliche Qualität (ohne Einbeziehung von Hindernissen) dar.

Dagegen lässt sich der Teil der intelligenten Heuristik einfach modellieren, der der einfachen Heuristik entspricht. Deren *reward* Funktion kann direkt übernommen werden und der *base reward* Wert wird auf „1“ gesetzt, wenn das Zielobjekt in Sicht ist. Ist das Ziel nicht in Sicht, dann wird die oben beschriebene Regel benutzt, befinden sich keine Agenten in Sicht, wird der *base reward* Wert trotz fehlendem Zielobjekts auf „1“ gesetzt, sonst auf „0“.

Bezüglich der Erweiterung mit der Überprüfung, ob Agenten in Sicht sind, haben sich in Tests nur minimale Unterschiede ergeben. Beispielsweise ergab sich (auf dem Säulenszenario mit 8 Agenten mit SXCS und einem Zielobjekt mit einfacher Richtungsänderung und Geschwindigkeit 2) eine Qualität von 39,15% im Vergleich zur originalen Implementierung von 32,20% bei 500 Schritten bzw. 36,28% zu 35,75% bei 2.000 Schritten.

Es ist allerdings denkbar, dass bestimmte Szenarien, z.B. mit vielen Hindernissen oder sehr wenigen Agenten, speziell diese Art der Bewertung problematisch machen. Beispielsweise erreicht mit nur einem Agenten mit SXCS im Säulenszenario mit einem Zielagenten mit einfacher Richtungsänderung und Geschwindigkeit 2 bei 2.000 Schritten die Variante ohne dieser Erweiterung eine Qualität von 7,21% während die Variante mit der Erweiterung nur 4,55% erreicht. Bei einer größeren Anzahl von Agenten liegt die Variante mit der Erweiterung jedoch vorne und wird in den Tests in dieser Arbeit verwendet werden.

In der Implementierung wird die mit der Bewertung zusammenhängende Funktion

checkRewardPoints() (siehe Programm A.9) in Programm A.11 (Zeile 15) bzw. in Programm A.14 (Zeile 15) aufgerufen.

Es ergeben sich also folgende Möglichkeiten:

- Zielobjekt nicht in Sicht, mindestens ein Agent in Sicht $\Rightarrow$ *base reward* $= 0$,
- Zielobjekt nicht in Sicht, kein Agent in Sicht $\Rightarrow$ *base reward* $= 1$ und
- Zielobjekt in Sicht $\Rightarrow$ *base reward* $= 1$.

3.5 Auswahlart der *classifier*

In der Standardimplementierung von XCS wird in jeder Probleminstanz entweder zufällig oder in jedem Schritt anhand eines Parameters zwischen der *explore* und *exploit* Phase gewechselt. Da dies im Überwachungsszenario in dieser Art nicht möglich ist, werden in diesem Kapitel mehrere Auswahlarten und insbesondere der Wechsel zwischen den Auswahlarten vorgestellt und diskutiert.

Eine Auswahl von *classifier* ist notwendig, da ein Sensordatensatz von mehreren *classifier* erkannt werden kann und in jedem Schritt somit mehrere passende *classifier* samt Aktionen ausgewählt werden können (siehe Kapitel 3.2). Deshalb stellt sich die Frage, welche der Aktionen im jeweiligen Schritt ausgeführt werden soll. Als Basis der Entscheidung hat ein Agent zum einen die Sensordaten und zum anderen die eigene *classifier set* Liste zur Verfügung.

In XCS wird dazu die zur jeweiligen Sensordatensatz passenden *match set* Liste in vier (Anzahl der möglichen Aktionen) Gruppen entsprechend des *action* Werts des jeweiligen *classifier* aufgeteilt. Danach werden alle Produkte aus den *fitness* und *reward prediction* Werten der *classifier* aus der jeweiligen Gruppe aufaddiert und durch die Summe der *fitness* Werte der *classifier* der jeweiligen Gruppe geteilt. Dieser Wert wird im Folgenden *predictionFitnessProductSum* genannt.

In der ursprünglichen Implementierung [But00] werden folgende Arten beschrieben, wie eine der vier Aktionen ausgewählt werden kann:

random selection Zufällige Auswahl einer Aktion (siehe Kapitel 3.5.1)

best selection Auswahl der Aktion mit dem höchsten *predictionFitnessProductSum* Wert der jeweiligen Gruppe (siehe Kapitel 3.5.2)

roulette wheel selection Zufällige Auswahl einer Aktion, Wahrscheinlichkeit abhängig vom *predictionFitnessProductSum* Wert der jeweiligen Gruppe (siehe Kapitel 3.5.3)

Im Folgenden werden nun diese Auswahlarten vorgestellt sowie eine weitere Auswahlart, die Auswahlart *tournament selection* (siehe Kapitel 3.5.4), aus der Literatur bespro-

chen. Im nächsten Abschnitt (Kapitel 3.6) wird dann der Wechsel zwischen diesen Auswahlarten untersucht. Neben der einfachen Auswahlart *best selection* scheint insbesondere der dynamische Wechsel zwischen den Auswahlarten erfolgsversprechend zu sein.

3.5.1 Auswahlart *random selection*

Prägende Idee für diese Auswahlart in einem statischen Szenario ist, das XCS möglichst vielen verschiedenen Situationen auszusetzen. Da in einem statischen Szenario die Zielposition sowie die Hindernisse fest sind, ist es wichtig, durch *random selection* dem XCS einen gewissen Spielraum zu geben.

Im Vergleich zu standardmäßig statischen Szenario ist dagegen es bei einem dynamischen Überwachungsszenario (siehe Kapitel 2.1) weder nötig noch hilfreich, die Auswahlart *random selection* zu nutzen, da

- zum einen, aufgrund ständiger Bewegung anderer Agenten und des Zielobjekts und fehlendem Neustart beim Erreichen des Ziels, das Problem dynamisch ist und die Agenten mit vielen verschiedenen Situationen konfrontiert werden und
- zum anderen, da es für ein erfolgreiches Bestehen in einem Überwachungsszenario wichtig ist, dass ein Agent über eine längere Zeit hinweg eine hohe Qualität liefert, also stetig gute Entscheidungen trifft.

Eine zufällige Auswahl scheint also wenig zielführend zu sein, weshalb im folgenden Auswahlarten besprochen werden, die auf die bisherige Qualität der in Frage kommenden *classifier* eingehen.

3.5.2 Auswahlart *best selection*

Bei der Auswahlart *best selection* wird nur die Aktion mit dem höchsten *predictionFitnessProductSum* Wert ausgewählt. Die Verwendung dieser Auswahlart kann u.U. zu langen Folgen gleicher Aktionen führen, sofern sich die Umwelt nicht ändert. Beispielsweise würde ein Agent damit in einer Sackgasse andauernd gegen eine Wand laufen.

Zur Verfolgung von einem Zielobjekt erscheint ein kompromissloses Verhalten sinnvoll, wie die Ergebnisse der Heuristiken zeigen (siehe beispielsweise die Qualitäten in Tabelle 5.4 der Fall mit einfacher Richtungsänderung). Allerdings wechseln hierbei die Heuristiken zwischen zwei verschiedenen Phasen. Ist das Zielobjekt nicht in Sicht, bewegen sie sich zufällig auf dem Torus. Gerade bei Szenarien mit Hindernissen kann ein solcher Wechsel sinnvoll sein, um die oben genannte Gefahr der großen Anzahl an blockierten Bewegungen zu vermeiden.

Auch sind die in dieser Arbeit verwendeten Sensorfähigkeiten beschränkt, der Agent weiß nicht, wo genau sich das Zielobjekt befindet, selbst wenn es in Sicht ist. Dementsprechend könnte es sein, dass eine optimale Verhaltensstrategie Entscheidungen eher auf Basis von Wahrscheinlichkeitsverteilungen treffen sollte.

3.5.3 Auswahlart *roulette wheel selection*

Bei dieser Auswahlart bestimmt der *predictionFitnessProductSum* Wert, relativ zu den anderen *predictionFitnessProductSum* Werten, die Wahrscheinlichkeit, ausgewählt zu werden. Wie die Auswahlart *best selection* erscheint diese Auswahlart also grundsätzlich sinnvoll, da sie anhand bisheriger Erfahrungen des Agenten versucht, Entscheidungen zu treffen.

Allerdings kann eine auf Proportionen basierte Auswahlart wie *roulette wheel selection* problematisch sein, wenn sich die *predictionFitnessProductSum* Werte einander ähneln und somit die Aktionen mit nahezu gleicher Wahrscheinlichkeit gewählt werden. Dies kann aufgrund der stochastischen Natur und der unscharfen Sensorinformationen gegeben sein, weshalb diese Auswahlart eher der Auswahlart *random selection* als *best selection* ähnelt.

3.5.4 Auswahlart *tournament selection*

Zusätzlich zu den oben erwähnten drei Möglichkeiten wurde in [MVBG03] eine weitere Auswahlart vorgestellt und in Bezug auf XCS diskutiert. Sie wurde dort mit *tournament selection* bezeichnet. Als Vorteile gegenüber den herkömmlichen Auswahlarten werden u.a. geringerer Selektionsdruck, höhere Effizienz und geringerer Einfluss von Störungen

genannt. Durch die Anpassung der Turniergröße ergibt sich außerdem eine flexible Anpassungsmöglichkeit.

In den dort vorgestellten Experimenten mit einem *single step* Problem wurden signifikante Vorteile dieser, auf proportionaler Selektion beruhender, Auswahlart gefunden. Daher soll sie auch im Zusammenhang mit Überwachungsszenarien getestet werden.

Allerdings findet in der Literatur die Auswahl auf Basis von einzelnen *classifier* statt. Dagegen wird in dieser Arbeit wie auch in der Standardimplementierung von XCS in [But00] alle *classifier* in nach *action* Werten eingeteilten Gruppen sortiert und deren *reward prediction* und *fitness* Werte zusammengenommen. Deshalb wird hier eine Implementierung der Auswahlart *tournament selection* gewählt, die näher am ursprünglichen Algorithmus aus dem Bereich der genetischen Algorithmen liegt [MMGG95].

Charakteristisch für diese Auswahlart ist, dass

1. k Elemente aus einer Menge zufällig ausgewählt werden,
2. nach ihrem zugehörigen Wert sortiert werden und
3. absteigend mit Wahrscheinlichkeit p das jeweilige Element gewählt wird.

   $\Rightarrow$ Das beste Element wird mit Wahrscheinlichkeit p, das zweitbeste mit Wahrscheinlichkeit $(1-p)p$, das drittbeste mit Wahrscheinlichkeit $(1-p)^2p$ und das letzte (bei $k=4$) mit Wahrscheinlichkeit $(1-p)^3$ gewählt.

In dem hier besprochenen Fall enthalten die Mengen immer vier (Anzahl der Aktionen) Elemente. Diese entsprechen jeweils den berechneten *predictionFitnessProductSum* Werten. Zur Vereinfachung der Tests wird k auf den Maximalwert gesetzt, damit alle Aktionen zumindest eine geringe Wahrscheinlichkeit besitzen, ausgewählt zu werden.

Im Grunde ist diese Auswahlart deckungsgleich mit der *roulette wheel selection*, allerdings ohne dem Problem, dass die Auswahlwahrscheinlichkeit aufgrund ähnlicher Produkte sich ebenfalls ähneln. Außerdem ist die Darstellung selbst sehr flexibel, beispielsweise wäre *tournament selection* mit $p = 1{,}0$ und $k = 4$ identisch mit *best selection* und mit

$p = 1{,}0$ und $k = 1$ wäre es identisch mit *random selection*. Diese Form der Auswahl, bei geeigneter Wahl von k und p, scheint also sehr vielversprechend zu sein.

Wie in Abbildung 3.2 zu sehen, wird für Werte $p < 0{,}5$ das schlechteste Element nicht mehr mit geringster Wahrscheinlichkeit gewählt. Außerdem verhält sich die Auswahlart im Bereich zwischen 0,2 und 0,4 ähnlich der Auswahlart *random selection*. Die Verwendung der Auswahlart *tournament selection* soll sich hier deshalb auf Werte $p \geq 0{,}6$ beschränken.

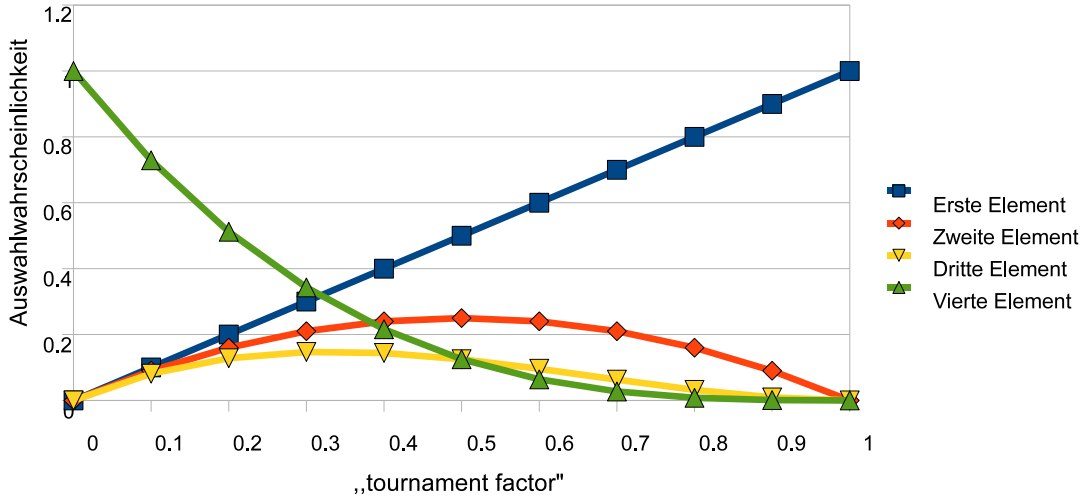

Abbildung 3.2: Darstellung der Auswahlfunktion der Auswahlart *tournament selection*

Bei der Implementierung ist darauf zu achten, dass bei der Sortierung Einträge mit gleichem Produkt aus den *fitness* und *reward prediction* Werten in zufälliger Reihenfolge aufgeführt werden. Andernfalls würden insbesondere am Anfang alle Agenten in dieselbe Richtung laufen, da alle *predictionFitnessProductSum* Werte identisch sind.

3.6 Abwechselnde *explore/exploit* Phasen

In der Standardimplementierung von XCS wird zwischen verschiedenen Auswahlarten gewechselt. Die Auswahlarten werden hierzu in zwei Gruppen geteilt, in die *explore* Phase und in die *exploit* Phase.

In der *exploit* Phase soll der jeweilige *predictionFitnessProductSum* Wert stärker als in der *explore* Phase gewichtet werden. Beispielsweise könnte man für die *exploit* Phase die Auswahlart *best selection* und für die *explore* Phase die Auswahlart *roulette wheel selection* festsetzen.

Wesentlicher Leitgedanke ist es, mit Hilfe der *explore* Phasen den Suchraum besser erforschen zu können, dann aber zur eigentlichen Problemlösung in der *exploit* Phase möglichst direkt auf das Ziel zuzugehen um *classifier* stärker zu belohnen, die am kürzesten Weg beteiligt sind.

Die Wahl der Auswahlart in Kapitel 3.3 für *classifier* in Punkt (3) kann auf verschiedene Weise erfolgen. In der Standardimplementierung von XCS wird nach jedem Erreichen des Ziels zwischen *exploit* und *explore* entweder umgeschalten oder zufällig mit einer bestimmten Wahrscheinlichkeit eine Auswahlart ermittelt. Es werden also abwechselnd ganze Probleminstanzen entweder nur in der *exploit* oder nur in der *explore* Phase berechnet. Dies erscheint sinnvoll für die erwähnten Standardprobleme, da nach Erreichen des Ziels eine neue Probleminstanz gestartet wird und die Entscheidungen, die während der Lösung einer Probleminstanz getroffen wurden, keine externen Auswirkungen auf die folgenden Probleminstanzen haben, die Probleminstanzen also nicht miteinander zusammenhängen.

Bei dem hier vorgestellten Überwachungsszenario kann dagegen nicht neu gestartet werden, es gibt keine „Trockenübung“. Die Qualität eines Algorithmus soll deshalb davon abhängen, wie gut sich der Algorithmus während der gesamten Berechnung, inklusive der Lernphasen, verhält. Es ist nicht möglich bei diesem Szenario zwischen *exploit* und *explore* Phasen in dem Sinne zu differenzieren, wie dies bei XCS geschieht. Dort wird die Qualität nur während der *exploit* Phase gemessen.

Desweiteren greift auch die Implementierung der Idee einer reinen *explore* Phase beim Überwachungsszenario nicht, da das Szenario nicht statisch, sondern dynamisch ist. Ein zufälliges Herumlaufen kann, im Vergleich zur gewichteten Auswahl der Aktionen, dazu führen, dass der Agent mit bestimmten Situationen mit deutlich niedrigerer Wahrscheinlichkeit konfrontiert wird, da der Agent sich in Hindernissen verfängt oder das Zielobjekt (z.B. mit „Intelligentem Verhalten“ aus Kapitel 2.5.4) ihm andauernd ausweicht. Daher

erscheint es sinnvoll, weitere Formen des Wechsels zwischen diesen Phasen zu untersuchen.

Bei der Standardimplementierung für den statischen Fall ist allerdings das Erreichen eines positiven *base reward* Werts äquivalent mit einem Neustart der Probleminstanz. Während dort das gesamte Szenario (alle Agenten, Hindernisse und das Zielobjekt) auf den Startzustand zurückgesetzt werden, läuft das Überwachungsszenario weiter.

Als erweiterten Ansatz wird deshalb eine neue Problemdefinition gelten, bei der nicht das Erreichen eines positiven *base reward* Werts (also ein Neustart der Probleminstanz) einen Phasenwechsel auslöst, sondern stattdessen eine *Änderung* des *base reward* Werts ausschlaggebend ist und einen Wechsel zwischen der *explore* und *exploit* Phase auslöst. Bei einer anfänglichen *explore* Phase würde dann immer in die *exploit* Phase gewechselt werden, wenn das Zielobjekt in Sicht ist bzw. wenn keine Agenten in Sicht sind.

Es werden nun also folgende Varianten des Wechsels zwischen den Phasen näher betrachtet:

1. Andauernde *explore* Phase;
2. andauernde *exploit* Phase;
3. abwechselnd *explore* und *exploit* Phase (bei Änderung des *base reward*, beginnend mit *explore*) sowie
4. in jedem Schritt zufällig entweder *explore* oder *exploit* Phase (50% Wahrscheinlichkeit jeweils)

Hervorzuheben ist, dass die Varianten 3 und 4 angewendet auf die nicht an Überwachungsszenarien angepasste Standardimplementierung des *multi step* XCS Verfahrens keinen Unterschied machen würden. Dies liegt daran, dass beim Erreichen eines positiven *base reward* Werts sowieso eine neue Probleminstanz gestartet wird, die *explore* und *exploit* Phasen separat betrachtet werden können und zwischen den Probleminstanzen zwischen der *exploit* und *explore* Phase gewechselt wird. Variante (4.) wird in der Arbeit nicht weiter beachtet, da kein Grund erkennbar ist, weshalb ein andauernder, zufälliger Wechsel einen Vorteil erbringen könnte.

Kapitel 4

XCS Varianten

I'm working to improve my methods, and every hour I save is an hour added to my life.
Ayn Rand

Nach der allgemeinen Einführung in XCS im letzten Kapitel wird hier nun die eigentliche Frage der Arbeit beantwortet, nämlich wie sich eine Umsetzung von XCS auf einem Überwachungsszenario gestaltet und welche konkreten Änderungen dafür am Algorithmus notwendig sind. Hierzu war es notwendig, die XCS Implementierung vollständig nachzuvollziehen. Dadurch war es möglich, für jeden Bestandteil entscheiden zu können, welche Rolle er bezüglich eines solchen Szenarien spielt.

Dazu werden zuerst allgemeine Anpassungen des Algorithmus und der Implementierung besprochen (siehe Kapitel 4.1) um dann auf die konkreten Veränderungen der einzelnen XCS Varianten einzugehen. Zum einen wird der XCS Algorithmus selbst in Kapitel 4.2 vorgestellt, dort wird insbesondere die Behandlung des Neustarts einer Probleminstanz diskutiert. Zum anderen wird eine an Überwachungsszenarien angepasste Variante (SXCS) vorgestellt, welche unter dem Gesichtspunkt des Problems einer kontinuierlichen Überwachung eines Zielobjekts entwickelt wird. Abschließend wird in Kapitel 4.4 eine erweiterte SXCS Variante (DSXCS) vorgestellt, die es explizit erlaubt, *reward* Werte erst mit einiger Verzögerung den jeweiligen *action set* Listen zuweist. Sie soll lediglich als Ausblick für weitere Verbesserungen dienen. Schließlich stellt das Kapitel 4.5 eine Variante mit Kommunikation mit anderen Agenten vor, die allerdings nur als Ausblick zu sehen ist.

4.1 Allgemeine Anpassungen von XCS

Eine Anzahl allgemeiner Änderungen an der Implementierung und am Algorithmus waren notwendig, um XCS in einem Überwachungsszenario laufen zu lassen. Unter anderen sind dies:

- Die Berechnung der Summe der *numerosity* Werte wurden neu organisiert und ein Fehler bei der Aktualisierung des *numerosity* Werts in der Implementierung korrigiert (siehe Kapitel A.4).
- Der genetische Operator verwendet hier zwei feste, anstatt zufällige Schnittpunkte für das *two point crossover* (siehe Kapitel 3.3.4).
- Die Qualität des Algorithmus wird zu jedem Zeitpunkt und nicht nur in der *exploit* Phase gemessen, da ein fortlaufendes Problem und kein statisches Szenario betrachtet wird (siehe Kapitel 3.6).
- Mehrere XCS Parameter wurden angepasst (siehe Kapitel 5.5).
- Das Erreichen des Ziels wurde für das Überwachungsszenario neu verfasst, wie auch der Neustart von Probleminstanzen neu geregelt wurde (siehe Kapitel 3.4).
- Die Reihenfolge bei der Bewertung, Entscheidung und der Aktion in einem Multiagentensystem auf einem diskreten Torus musste überdacht werden (siehe Kapitel 3.3).

4.2 XCS *multi step* Verfahren

Idee dieses Verfahrens ist, dass der *reward* Wert, den eine Aktion (bzw. der jeweils zugehörigen *action set* Liste und die dortigen *classifier*) erhält, vom erwarteten *reward* Wert der folgenden Aktion abhängen soll. Das wird dadurch erreicht, dass *classifier* jeweils ein Schritt lang gespeichert werden und dann einen Teil der *reward prediction* Werte der jeweils nächsten *match set* Liste weitergegeben wird. Dadurch ist es möglich, dass ein beim Ziel vergebener *base reward* Wert über eine ganze Kette von Schritten durchgereicht wird.

Da bei der Standardimplementierung von XCS die Probleminstanz beim Erreichen eines positiven *base reward* Werts jeweils neu gestartet wird, benötigt diese Weitergabe

ein ganze Anzahl von Probleminstanzen. Dabei gilt die Annahme, dass durch mehrfache Wiederholung des Lernprozesses sich ein Regelsatz ergibt, mit dem das Ziel mit höherer Wahrscheinlichkeit bzw. mit einer geringeren Schrittzahl gefunden wird.

Dies entspricht dem aus [BW01] bekannten XCS *multi step* Verfahren. Der wesentliche Unterschied zur Implementierung in dieser Arbeit ist, dass das Szenario bei einem positiven *base reward* Wert nicht neu gestartet wird. Algorithmisch ist die Implementierung ansonsten identisch. Dies zeigt sich in Programm A.11 (Zeilen 22-27). Zwar wird hier die *action set* Liste gelöscht, das Szenario selbst läuft aber weiter. In der originalen Implementierung in [But00] wird an dieser Stelle im Algorithmus die aktuelle Probleminstanz abgebrochen (in *XCS.java* in der Funktion *doOneMultiStepProblemExploit()* bzw. *doOneMultiStepProblemExplore()*). Liegt kein positiver *base reward* Wert vor, so wird lediglich der für diesen Schritt erwartete *reward* Wert (nämlich der *maxPrediction* Wert) an die letzte *action set* Liste gegeben.

In den Programmen A.12 und A.13 finden sich, neben Anpassungen an den Simulator, keine wesentlichen Änderungen. In Programm A.12 wird der ermittelte *base reward* Wert zusammen mit dem ermittelten *maxPrediction* Wert an die Aktualisierungsfunktion der jeweiligen *action set* Liste weitergegeben. Im Programm A.13 wird dann eine Aktion daraus ausgewählt und entsprechende *match set* und *action set* Listen erstellt.

4.3 XCS Variante für Überwachungsszenarien (SXCS)

Die Hypothese bei der Aufstellung dieser XCS Variante ist im Grunde dieselbe wie beim XCS *multi step* Verfahren selbst, nämlich dass die Kombination mehrerer Aktionen zum Ziel führt. Beim *multi step* Verfahren besteht die wesentliche Verbindung zwischen den *action set* Listen jeweils nur zwischen zwei direkt aufeinanderfolgenden *action set* Listen über den *maxPrediction* Wert. Dadurch kann in einer statischen Umgebung über mehrere (identische) Probleme hinweg eine optimale Einstellung (des *fitness* und *reward prediction* Werts) für die *classifier* gefunden werden, sofern das damit zusammenhängende Problem eine Markow-Kette darstellt [BW01], also bei dem Problem „die zukünftige Entwicklung des Prozesses nur von dem zuletzt beobachteten Zustand abhängt und von der sonstigen Vorgeschichte unabhängig ist“ [WS08].

Die Idee für SXCS ist nun, eine ähnliche Implementierung zu wählen, wie beispielsweise in [LW99] vorgestellt. Dort kann eine Aktion eines Agenten sowohl äußere als auch innere Auswirkungen haben, wodurch eine Art Speicher realisierbar ist und somit in die Markow-Eigenschaft teilweise wiederhergestellt werden kann. Für SXCS soll dagegen direkt die aktivierten *action set* Listen gespeichert werden, bis ein sogenanntes Ereignis (siehe Kapitel 4.3.2) auftritt.

Bei der hier besprochenen SXCS Variante (*Supervising eXtended Classifier System*) wird in Kapitel 4.3.1 zuerst die Umsetzung dieser Idee diskutiert werden. Sie baut auf sogenannten „Ereignissen“ auf, die mit einer Änderung des *base reward* Werts einhergehen (siehe Kapitel 4.3.2). Die Implementierung selbst wird dann in Kapitel 4.3.4 vorgestellt.

4.3.1 Umsetzung von SXCS

Bei der SXCS Variante soll die Verbindung zwischen den *action set* Listen direkt durch die zeitliche Nähe zur Vergabe des *base reward* Werts gegeben sein. Es wird in jedem Schritt die jeweilige *action set* Liste gespeichert und aufgehoben, bis ein neues Ereignis (siehe Kapitel 4.3.2) eintritt und dann in Abhängigkeit des Alters mit einem entsprechenden *reward* Wert aktualisiert.

In der Standardimplementierung von XCS wurde der *reward* Wert bei der Weitergabe jeweils mit einem Faktor γ multipliziert. Dieses Verhalten soll ungefähr nachgebildet werden, allerdings mit einer Funktion die sich über ein Ereignis erstreckt und einen Nullpunkt besitzt. Dies wird beispielsweise durch die quadratische Funktion erfüllt.

Bezeichne $r(a)$ den *reward* Wert für die *action set* Liste mit Alter a, bei quadratischer Verteilung des *base reward* Werts ergibt sich dann:

$$r(a) = \begin{cases} \frac{a^2}{\text{size}(action\ set)^2} & \text{falls } base\ reward = 1 \\ \frac{(1-a)^2}{\text{size}(action\ set)^2} & \text{falls } base\ reward = 0 \end{cases}$$

Der Vergleich der linearen Vergabe, der quadratischen Vergabe und der Vergabe bei XCS über 10 Schritte ist in Abbildung 4.1 dargestellt.

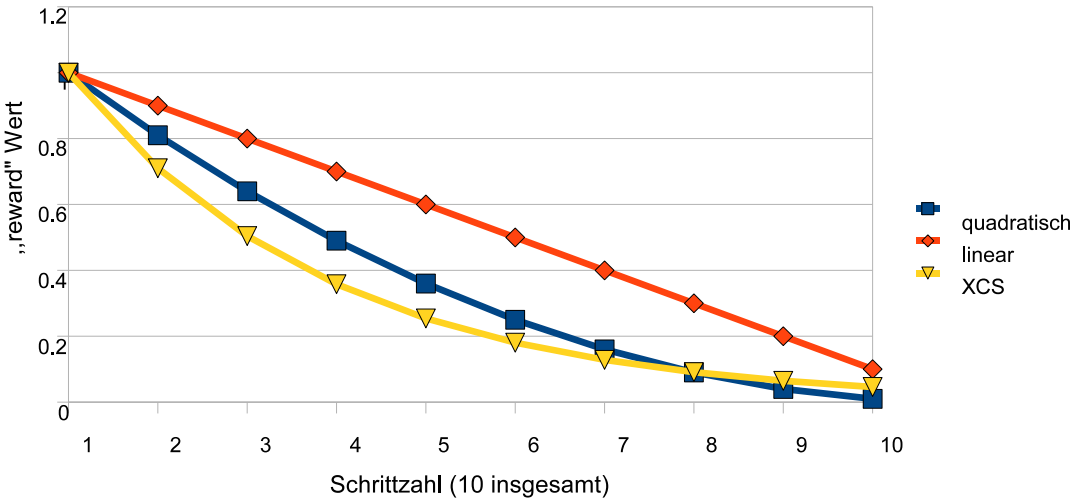

Abbildung 4.1: Vergleich verschiedener Arten der Weitergabe des *reward* Werts

In Tests ergab sich für die quadratische Verteilung im Vergleich zur linearen Verteilung des *base reward* Werts nur minimale Unterschiede. In Tests wurden keine wesentlichen Unterschiede zwischen der linearen und quadratischen Verteilung festgestellt, weitere Untersuchungen und womöglich eine Modifikation der Verteilungsfunktion wäre ein möglicher Ansatzpunkt für weitere Forschungen.

In Abbildung 4.2 ist dies dargestellt. Weitere Grafiken beschränken sich auf die lineare Verteilung des *base reward* Werts, während in den Simulationen die quadratische Vergabe des *base reward* Werts benutzt wird.

4.3.2 Ereignisse

In XCS wird lediglich die jeweils letzte *action set* Liste aus dem vorherigen Schritt gespeichert. In der neuen Implementierung werden dagegen eine ganze Anzahl (bis zum Wert *maxStackSize*) von *action set* Listen gespeichert. Die Speicherung erlaubt zum einen eine Vorverarbeitung des *reward* Werts anhand der vergangenen Schritte und auf Basis einer größeren Zahl von *action set* Listen und zum anderen die zeitliche Relativierung einer *action set* Liste zu einem Ereignis. Die *classifier* werden dann jeweils rückwirkend anhand

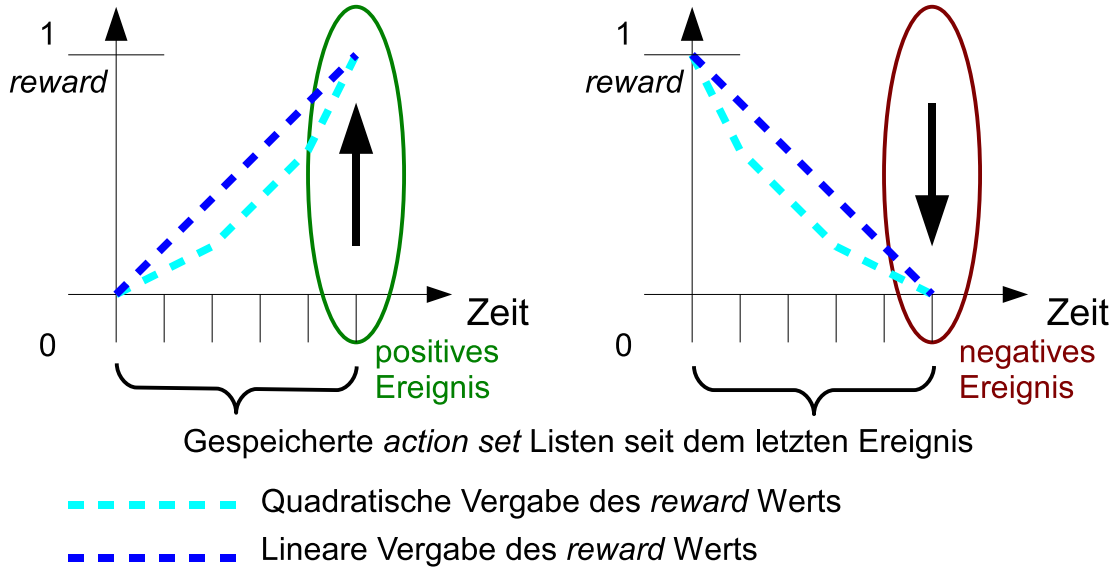

Abbildung 4.2: Schematische Darstellung der (quadratischen) Verteilung des *reward* Werts an gespeicherte *action set* Listen bei einem positiven bzw. negativen Ereignis

des jeweiligen *reward* Werts aktualisiert, sobald bestimmte Bedingungen eingetreten sind.

Von einem positiven bzw. negativen Ereignis spricht man, wenn sich der *base reward* Wert im Vergleich zum vorangegangenen Schritt verändert hat (siehe Abbildung 4.3). Mit der in Kapitel 3.4.3 vorgestellten *reward* Funktion hieße das, dass sich entweder das Zielobjekt gerade in Sichtweite bzw. aus ihr heraus bewegt hat oder, dass der jeweilige Agent die Sicht zu anderen Agenten verloren hat bzw. sich gerade erst wieder in Sicht eines anderen Agenten bewegt.

Bei der Benutzung eines solchen Stacks entsteht eine Zeitverzögerung, d.h. die *classifier* erhalten jeweils Informationen, die bis zu *maxStackSize* Schritten zu alt sein können. Tritt beim Stack ein Überlauf auf, gab es also *maxStackSize* Schritte lang keine Änderung des *base reward* Werts mehr, dann wird abgebrochen und die $\frac{maxStackSize}{2}$ ältesten Einträge werden vom Stack genommen. Alle diese Einträge werden dabei vorher mit einem *reward* Wert aktualisiert, der diesem *base reward* Wert entspricht.

Abbildung 4.4 zeigt die Bewertung bei einem solchen neutralen Ereignis, bei dem nach einem Überlauf die erste Hälfte mit 1 bewertet wurde. Außerdem ist dort der maximale

Fehler dargestellt, welcher eintreten würde, wenn direkt beim Schritt nach dem Abbruch eine Änderung des *base reward* Werts auftritt. Im dargestellten Fall würde sich also der *base reward* Wert beim aktuellen Zeitpunkt auf 0 verändern.

Ein Ereignis tritt also auf bei:

- Änderung des *base reward* Werts von 0 auf 1 (Zielobjekt war im letzten Schritt nicht in Überwachungsreichweite bzw. der letzte Agent hat die Sicht im letzten Schritt verlassen) ⇒ **positives Ereignis**,
- Änderung des *base reward* Werts von 1 auf 0 (Zielobjekt war im letzten Schritt in Überwachungsreichweite bzw. ein Agent hat gerade die Sicht betreten) ⇒ **negatives Ereignis**,
- Überlauf des Stacks (kein positives oder negatives Ereignis in den letzten *maxStackSize* Schritten), Zielobjekt ist in Überwachungsreichweite bzw. keine Agenten sind in Sicht ⇒ **neutrales Ereignis** (mit *base reward* = 1) oder bei
- Überlauf des Stacks (kein positives oder negatives Ereignis in den letzten *maxStackSize* Schritten), Zielobjekt ist nicht in Überwachungsreichweite bzw. mindestens ein Agent ist in Sicht ⇒ **neutrales Ereignis** (mit *base reward* = 0).

4.3.3 Größe des Stacks (*maxStackSize*)

Offen bleibt die Frage nach der Größe des Stacks. Mangels theoretischem Fundament muss man zwischen den drei wirkenden Faktoren einen Kompromiss finden. Große Werte für *maxStackSize* können zu folgenden Schwierigkeiten führen:

- Durch die Verwendung eines Stacks gibt es eine Verzögerung zu Beginn der Probleminstanz und insbesondere zu Beginn eines Experiments (also bei einer wesentlichen Änderung des Szenarien). Es kann u.U. bis zu $\frac{maxStackSize}{2}$ Schritte dauern, bis das erste Mal ein *classifier* aktualisiert wird.
- Ein zu großer Stack führt womöglich dazu, dass Aktionen positiv (oder negativ) bewertet werden, die an der Erreichung des *base reward* Werts nicht oder nur unbedeutend beteiligt waren, vor allem, wenn es sich um kurze lokale Entscheidungen handelt.

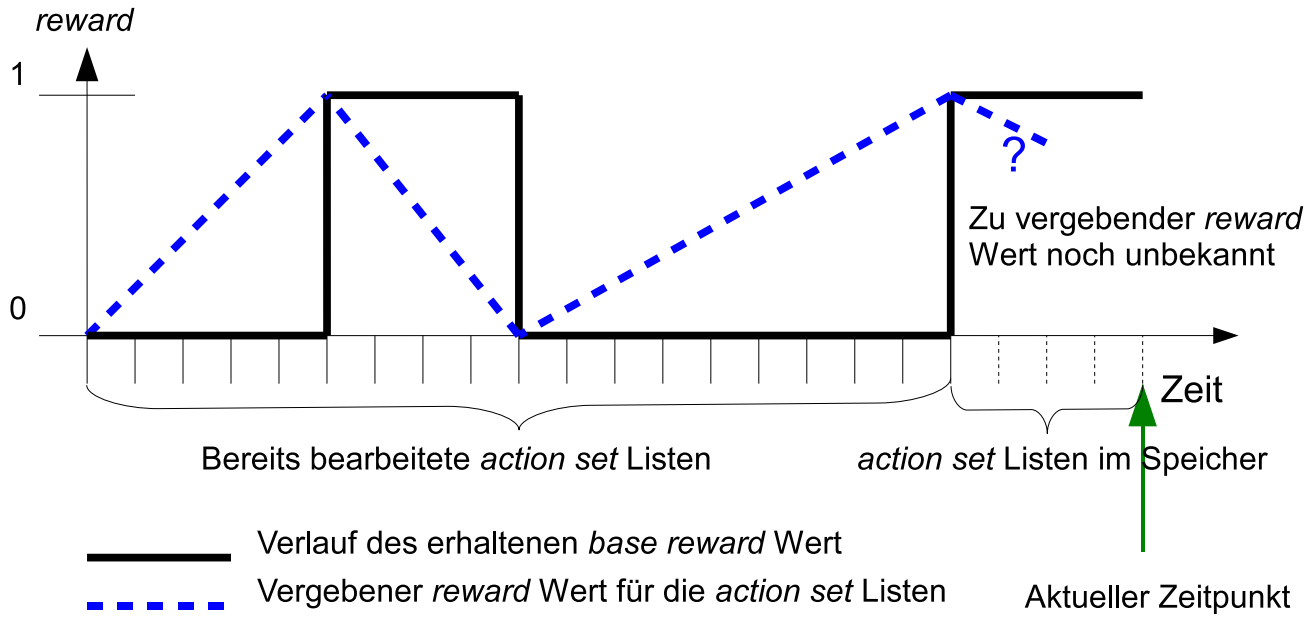

Abbildung 4.3: Schematische Darstellung der zeitlichen Verteilung des *reward* Werts an *action set* Listen nach mehreren positiven und negativen Ereignissen und der Speicherung der letzten *action set* Liste

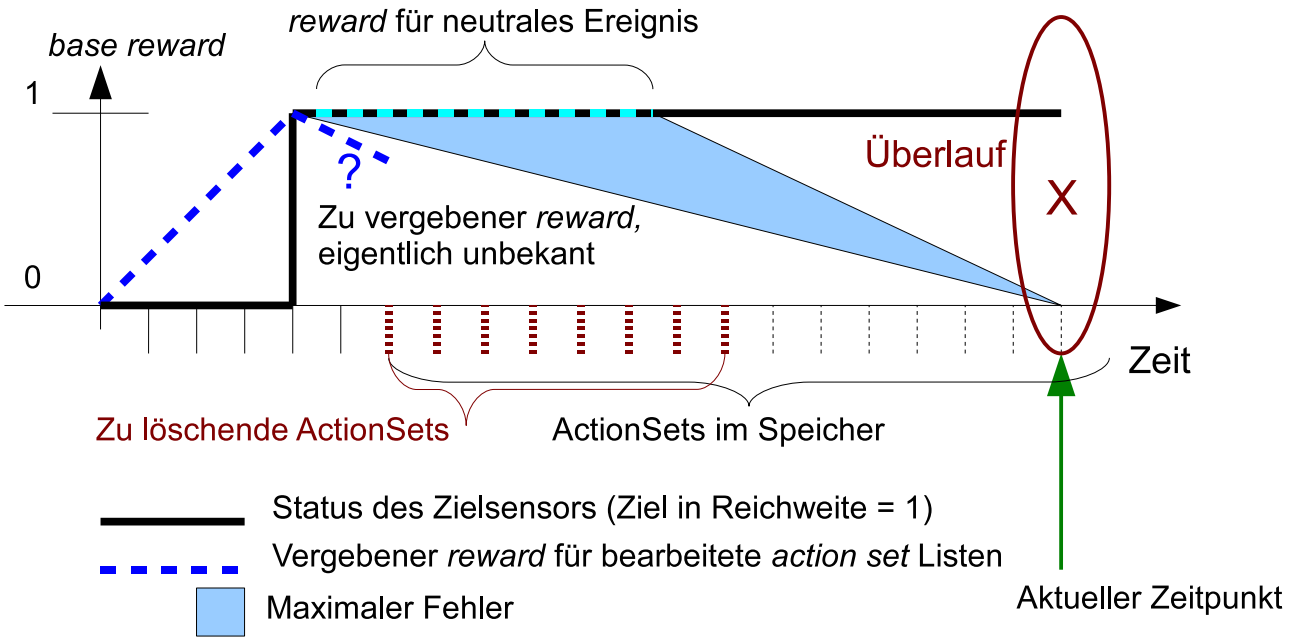

Abbildung 4.4: Schematische Darstellung der Bewertung von *action set* Listen bei einem neutralen Ereignis (mit *base reward* = 1)

Umgekehrt, wählt man den Stack zu klein, dann kann es zu folgenden Problemen kommen:

- Durch die geringere Größe des Stacks kann es zu einem Überlauf und so zu dem in Kapitel 4.3.2 beschriebenen Fehler bei der Bewertung kommen.
- Zum Erreichen des Ziels können eine ganze Reihe von Schritten notwendig sein. Passen sie nicht alle in den Stack, dann werden sie verworfen. Bei XCS wird dies durch Weitergabe der *reward* Werte von *action set* zu *action set* vermieden.

Der optimale Wert für *maxStackSize* muss also einen Kompromiss zwischen diesen Faktoren darstellen und ist somit abhängig vom Szenario. Optimal wäre eine Anpassung des *maxStackSize* Werts während des Durchlaufs. Die Untersuchung hier wird sich auf eine empirische Ermittlung eines ausreichend guten Werts für die betrachteten Szenarien beschränken.

Wie Abbildung 4.5 zeigt, spielt der Wert eher eine geringe Rolle, die erreichten Qualitäten unterscheiden sich im betrachteten Wertebereich kaum voneinander. Nur im Umfeld von $maxStackSize = 8$ sind deutliche Unterschiede sichtbar. Für den Fall mit der Torusgröße von 32x32 in Abbildung 4.6 sieht man dagegen nur eine leichte Steigerung für größere Werte. Auch das schwierige Szenario scheint hier eher von größeren Werten zu profitieren. Dies unterstreicht, dass der optimale Wert vom Szenario (bzw. der durchschnittlichen Zeit bis zum Erreichen des nächsten positiven *base reward* Werts) abhängt, dies bedarf jedoch weiterer Forschung. Für diese Arbeit wird im weiteren Verlauf im Allgemeinen ein Wert für ***maxStackSize* von** 8 und in Kapitel 5.7.1 wird in einem Fall 32 benutzt.

4.3.4 Implementierung von SXCS

Im Wesentlichen entspricht die Implementierung von SXCS der bekannten Implementierung von XCS (siehe Kapitel 4.2). Als Unterschiede sind festzuhalten:

- In der Funktion *calculateReward()* in Programm A.14 bei der Berechnung des *reward* Werts wird zwischen zwei Fällen unterschieden. Zum einen gibt es die Behandlung negativer und positiver Ereignisse (Zeile 19-23) und zum anderen die Behandlung

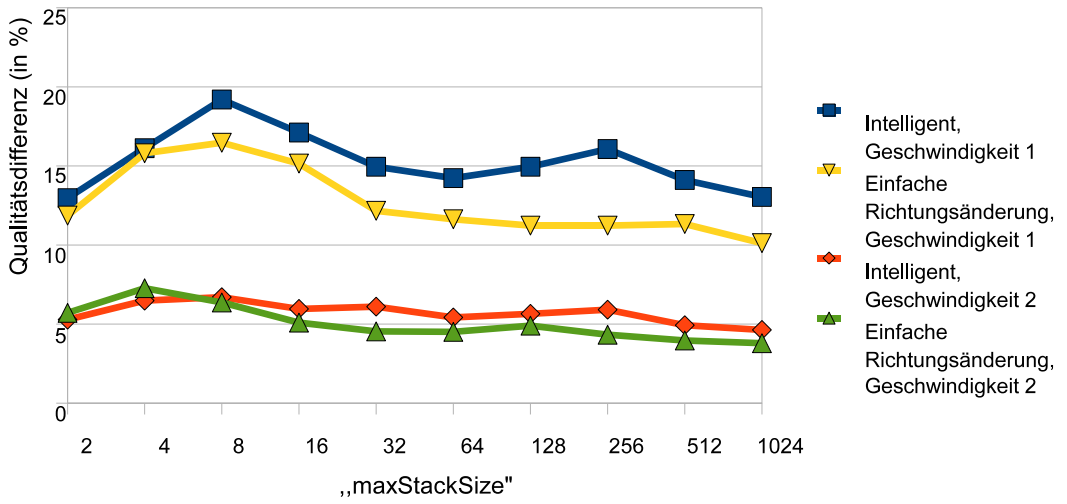

Abbildung 4.5: Vergleich verschiedener Werte für *maxStackSize* (Säulenszenario, SXCS Agenten, *tournament selection*, $p = 0{,}84$))

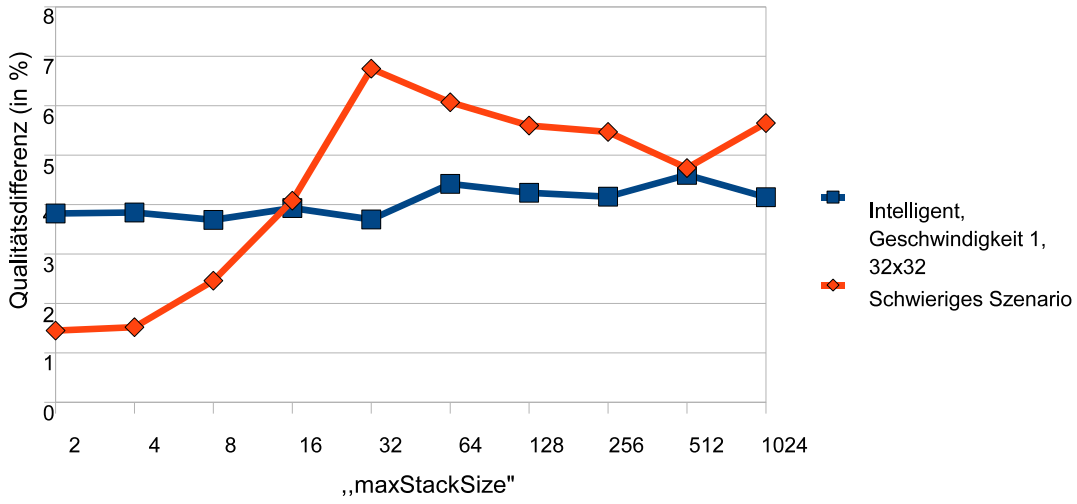

Abbildung 4.6: Vergleich verschiedener Werte für *maxStackSize* (spezielle Szenarien, SXCS Agenten, *tournament selection*, $p = 0{,}84$)

des Überlaufs des Stacks (Zeile 29-35), während bei der Implementierung von XCS in Programm A.11 in fast jedem Schritt unabhängig von Ereignissen eine Aktualisierung stattfindet.

- In der Funktion *collectReward()* in Programm A.15 werden nicht nur die aktuelle bzw. letzte *action set* Liste aktualisiert, sondern eine ganze Reihe aus dem gespeicherten Stack. Insbesondere werden dort die auf- bzw. absteigenden *reward* Werte nach einem positiven bzw. negativen Ereignis berechnet (Zeile 31-33). Bei der Berechnung der nächsten Aktion hingegen (Funktion *calculateNextMove()* in Programm A.16) wurde lediglich die Behandlung des Stacks hinzugefügt (Zeile 40-44).

4.4 SXCS Variante mit verzögerter *reward* (DSXCS)

Die Struktur der besprochenen XCS Variante SXCS erlaubt, ermittelte *reward* Werte erst bis zu *maxStackSize* Schritte später an die *classifier* weiterzugeben. Während diese Verzögerung dort lediglich als notwendiges Übel gesehen wurde, kann dies auch wesentliche Vorteile erbringen. Erst einmal kann eine Verzögerung dazu führen, dass ein ermittelter *reward* Wert nicht sofort eine Auswirkung auf die Bewegungsart des jeweiligen Agenten hat, der Agent also noch auf Basis seiner alten Konfiguration seiner *classifier set* Liste handelt. Würde ein Agent beispielsweise zu Beginn nach Westen gehen und dort das Zielobjekt sehen, dann würden in den nächsten Schritten nach Westen gerichtete Schritte besonders bevorzugt, obwohl in den meisten Fällen speziell zu Beginn ein eher explorativ es Verhalten von Vorteil ist. Zweiter und wesentlich wichtigerer Punkt ist, dass aufgrund der Verzögerung zum Zeitpunkt der Vergabe des *reward* Werts eine viel größere Anzahl an Werten zur Verfügung steht (die *reward* Werte der letzten *maxStackSize* Schritte), anhand derer entschieden werden kann, mit welchem Wert die einzelnen *classifier* aktualisiert werden sollen.

Offen bleibt die Frage nach einer Funktion, die dies vorteilhaft ausnutzt. Zwar konnte im Rahmen dieser Arbeit noch keine solche Funktion gefunden werden, prinzipiell erscheint es aber sehr wahrscheinlich, dass eine existiert. Für DSXCS ohne eine solche Funktion, also mit einer reinen Verzögerung, wird später in Kapitel 5.8 gezeigt, dass es von der Qualität her ähnlich SXCS ist.

Im Folgenden wird zwar, neben dem Aufbau einer solchen, speziell auf Verzögerungen ausgelegte, SXCS Variante DSXCS (*Delayed Surveillance eXtended Classifier System*), auf zwei solche Funktionen eingegangen, wie später in Kapitel 5.8 aber gezeigt wird, erreichen diese aber keine besseren sondern eher schlechtere Werte für ihre Qualität. Durch die am Ende im Ausblick (Kapitel 6.1) besprochenen möglichen Erweiterungen der hier besprochenen Modellierung könnten die Ansätze sich jedoch als nutzbringend erweisen, hier ist weitere Forschung nötig.

4.4.1 Aktualisierungsfaktor

Als eine mögliche Erweiterung bietet sich an, Ereignisse bzw. deren *reward* Werte mit einem Faktor zu erweitern, der bei der Aktualisierung der *action set* Listen eine Rolle spielt. Hier kann der Faktor entweder dazu dienen, mit der Lernrate β multipliziert die Rolle von β zu ersetzen und somit eine geringeren Einfluss des *reward* Werts zu erreichen, oder der *reward* Wert kann direkt mit dem Faktor multipliziert werden und somit den Faktor als eine Art Strafe verwenden. Ein Aktualisierungsfaktor von beispielsweise 1,0 hätte also keine Änderungen zur Folge, ein Faktor von 0,5 würde dagegen die Lernrate für den zugehörigen *reward* Wert bzw. den *reward* Wert selbst halbieren.

Motivation zur Einführung des Faktors war, dass man bestimmten *reward* Werten eine gewisse Genauigkeit zuweist um bestimmte Ereignisse weniger einflussreich zu machen bzw. um sie überhaupt zu bewerten. Beispielsweise könnte man durch eine Erweiterung mit kommunikativen Fähigkeiten dadurch externe Ereignisse (siehe Kapitel 4.5.2) einen niedrigeren Wert zuweisen. Im Folgenden wird der Aktualisierungsfaktor mit „*factor*“ bezeichnet.

4.4.2 Zeitgleiche Ereignisse

Durch die Struktur von DSXCS ist es desweiteren möglich, mehrere zeitgleiche Ereignisse bzw. deren zugehörige *reward* und *factor* Werte sinnvoll zu verarbeiten. Zeitgleiche Ereignisse können zwar im bisher verwendeten System nicht auftreten, aber eine Erweiterung des Systems durch eine Form der Kommunikation würde es möglich machen, dass z.B. ein Agent den eigenen *reward* Wert, die eigenen Sensordaten oder andere Informationen an

einen anderen Agenten als externes Ereignis weitergibt (siehe Kapitel 4.5.2).

Während bei SXCS das Auftreten mehrerer gleichzeitiger *reward* Werte dazu geführt hätte, dass die *classifier* der jeweiligen *action set* Listen nacheinander mit den jeweiligen Werten aktualisiert worden wären, kann sich ein auf DSXCS basierender Agent alle bisher gesammelten *reward* und *factor* Werte erst einmal näher ansehen und beispielsweise nur den höchsten *reward* Wert für die Aktualisierung benutzen. Auch hier ist wieder offen, wie die Funktion aussieht, die einen Wert auswählt bzw. berechnet.

4.4.3 Implementierung

Die Funktion *calculateReward()* ist identisch mit der in Kapitel 4.3.4 besprochenen Funktion (Programm A.14) bei der SXCS Variante ohne verzögerter Bewertung. Ebenso ist die Funktion *calculateNextMove()* (siehe Programm A.18) fast identisch mit der dort besprochenen Funktion, nur bei der Behandlung des Stacks wird hier beim Überlauf der erste Eintrag nicht einfach gelöscht, sondern mit der Funktion *processReward()* zuerst noch verarbeitet. In der Verarbeitung werden alle bisher gespeicherten *reward* und *factor* Werte berücksichtigt, um jeweils den *reward* und *factor* Wert für den ersten Eintrags zu berechnen und die *classifier* der zugehörigen *action set* Liste zu aktualisieren.

Zur Berechnung selbst gibt es verschiedene Möglichkeiten. Die einfachste Variante ist in Programm A.19 dargestellt, dort werden einfach alle zeitgleich abgespeicherten Werte nacheinander auf das *action set* angewendet. Eine Erweiterung ist in Programm A.20 dargestellt. Dort werden jeweils nur die Einträge mit dem größten Produkt aus den *reward* und *factor* Werten für die Aktualisierung benutzt. Noch komplexere Berechnungen sind denkbar, beispielsweise könnten alle Werte aller Einträge der *historicActionSet* Liste mit in die Berechnung einer jeden einzelnen Aktualisierung der *action set* Listen miteinbezogen werden.

4.5 DSXCS Variante mit Kommunikation

Wann immer ein *base reward* Wert an einen Agenten verteilt wird, kann es sinnvoll sein, diesen Wert an andere Agenten weiterzugeben. Dies wurde z.B. in einem ähnlichen Szenario in [ITS05] festgestellt, bei dem zwei auf XCS basierende Agenten gegen bis zu zwei anderen (zufälligen) Agenten eine vereinfachte Form des Fußballs spielen. Dabei erhielt bei Erreichen des Ziels (Beförderung des Balls aus dem Spielfeld) der Agent, der den Ball zuletzt berührt hat, den vollen *reward* Wert und der andere Agent 90% dieses Werts.

Hier werden nun zwei Formen der Weitergabe des *reward* Werts vorgestellt, zum einen die Kommunikationsvariante in der alle Agenten ihre *reward* Werte teilen (siehe Kapitel 4.5.3) und zum anderen eine Kommunikationsvariante bei der der *reward* Wert anhand ähnlicher in den *classifier sets* gespeicherter Verhaltensweisen verteilt wird (siehe Kapitel 4.5.4. Zuvor wird auf die Kommunikationsrestriktionen in Kapitel 4.5.1 und die Behandlung externer Ereignisse in Kapitel 4.5.2 eingegangen.

Wie später ausführlich in Kapitel 5.9 dargestellt wird, blieb dieser Ansatz erfolgreich. Über mögliche Gründe wird dort diskutiert.

4.5.1 Kommunikationsreichweite

Geht man davon aus, dass die Kommunikationsreichweite zumindest ausreichend groß ist um nahe Agenten zu kontaktieren, so kann man argumentieren, dass man dadurch ein Kommunikationsnetzwerk aufbauen kann, in dem jeder Agent jeden anderen Agenten - mit einer gewissen Zeitverzögerung - erreichen kann. Bei ausreichender Agentenzahl relativ zur freien Fläche fallen dadurch wahrscheinlich nur vereinzelte Agenten aus dem Netz, abhängig vom Szenario. Stehen die Agenten nicht indirekt andauernd miteinander in Kontakt (mit anderen Agenten als Proxy), sondern muss die Information zum Teil durch aktive Bewegungen der Agenten transportiert werden, tritt eine Zeitverzögerung auf. Auch kann die benötigte Bandbreite die verfügbare übersteigen, was ebenfalls zusätzliche Zeit benötigen kann. Der Einfachheit halber wird deshalb angenommen, dass zwar globale Kommunikation zur Verfügung steht, jedoch diese u.U. zeitverzögert stattfindet und wir nur geringe Mengen an Information weitergeben können. In diesem Falle soll ein Agent in jedem Schritt maximal lediglich einen Aktualisierungsfaktor (siehe Kapitel 4.4.1) und

einen *reward* Wert an alle anderen Agenten weitergeben können. Da dieser Algorithmus auf dem DSXCS Algorithmus aufbaut, sollen hier auch Ereignisse auftreten können und immer eine ganze Reihe von *action set* Listen bewertet werden. Somit muss außerdem noch ein Start- und Endzeitpunkt übermittelt werden. Dies wäre beispielsweise bei einem neutralen Ereignis $\frac{maxStackSize}{2}$ und *maxStackSize* oder bei einem positiven Ereignis mit 5 Schritten seit dem letzten Ereignis 0 als Startzeitpunkt und 4 als Endzeitpunkt.

4.5.2 Externe Ereignisse

Jeder ermittelte *reward* Wert kann an alle anderen Agenten weitergegeben werden. Wie ein solches sogenanntes „externes Ereignis“ dann von diesen Agenten aufgefasst wird, hängt von der jeweiligen Kommunikationsvariante ab. Diese werden in den folgenden Kapitel besprochen.

Durch eine gemeinsame Schnittstelle erhält jeder Agent den *reward* Wert zusammen mit dem Aktualisierungsfaktor. Dabei ergibt sich das Problem, dass sich *reward* überschneiden können, da jeder *reward* Wert sich rückwirkend auf die vergangenen *action set* Listen auswirken kann. Auch können mehrere externe *reward* Werte eintreffen, als auch ein eigener positiver lokaler *reward* Wert aufgetreten sein. Würden die *reward* Werte nach ihrer Eingangsreihenfolge abgearbeitet werden, kann es passieren, dass dieselbe *action set* Liste sowohl mit einem hohen als auch einem niedrigen *reward* Wert aktualisiert wird. Da das globale Ziel ist, das Zielobjekt durch *irgendeinen* Agenten zu überwachen, ist es in jedem einzelnen Zeitschritt nur relevant, dass ein *einzelner* Agent einen hohen *reward* Wert produziert bzw. weitergibt um die eigene Aktion als zielführend zu bewerten. Dies wird für die Entwicklung der Funktion, die für die Verarbeitung in Kapitel 4.4.2 verantwortlich ist, maßgeblich sein.

Befindet sich das Ziel beispielsweise gerade in Überwachungsreichweite mehrerer Agenten und verliert ein anderer Agent das Ziel aus der Sicht, sollte der Agent (und alle anderen Agenten), der das Ziel in Sicht hat, deswegen nicht bestraft werden, da das globale Ziel ja weiterhin erfüllt wurde. Ein gespeicherter höherer *reward* Wert soll also einen gespeicherten niedrigeren *reward* Wert verdrängen.

4.5.3 Einzelne Kommunikationsgruppe

Mit dieser Variante wird der Aktualisierungsfaktor fest auf 1,0 gesetzt und es werden alle *reward* Werte in gleicher Weise weitergegeben. Dadurch wird zwischen den Agenten nicht diskriminiert, was letztlich bedeutet, dass zwar zum einen diejenigen Agenten korrekt mit einem externen *reward* Wert belohnt werden, die sich zielführend verhalten, aber zum anderen eben auch diejenigen, die es nicht tun. Deren *classifier* werden somit zu einem gewissen Grad zufällig bewertet, denn es fehlt die Verbindung zwischen *classifier*, Handlung und der Bewertung. Idee der Variante ist, dass durch die Zusammenschaltung der *reward* Werte eine Art großes gemeinsames Sensorsystem entsteht.

4.5.4 Egoistische Kommunikationsgruppe

Die Idee für diese Art der Gruppenbildung ist, dass unterschiedliche Agenten unterschiedlich stark am Erfolg der anderen Agenten beteiligt sind. Ohne Kommunikation wird jeder Agent versuchen, selbst das Zielobjekt möglichst in die eigene Überwachungsreichweite zu bekommen, anstatt die Arbeit mit anderen Agenten zu teilen, also z.B. das Gebiet des Torus möglichst großräumig abzudecken, wie es der in Kapitel 2.4.3 vorgestellte Agent mit intelligenter Heuristik in mehreren Tests u.a. in Kapitel 5.2.2 demonstriert hat.

Diese Variante berechnet erst einmal für jeden Agenten einen „Egoismusfaktor“, indem grob die Wahrscheinlichkeit ermittelt wird, dass ein Agent, wenn sich ein anderer Agent in Sicht befindet, sich in diese Richtung bewegt. „*Egoismus*“-Faktor, weil ein großer Faktor bedeutet, dass der Agent eher einen kleinen Abstand zu anderen Agenten bevorzugt, also wahrscheinlich eher auf eigene Faust versuchen wird, das Zielobjekt in Sicht zu bekommen, anstatt ein möglichst großes Gebiet abzudecken.

Auf Basis dieses Faktors kann man dann eine Ähnlichkeit zwischen verschiedenen Agenten hinsichtlich des Verhaltens gegenüber anderen Agenten und damit den Aktualisierungsfaktor bestimmen. Die Hypothese hier ist, dass Agenten mit ähnlichem Egoismusfaktor auch eine ähnliche *classifier set* Liste besitzen. Besitzen mehrere Agenten eine ähnliche Liste, so bilden sie eine mehr oder weniger homogene Gruppe, die sich gegenseitig *reward* Werte mit hohem Aktualisierungsfaktor senden, während Agenten mit anderem Egoismusfaktor leer ausgehen.

Der Vorteil gegenüber Verfahren, die die *classifier set* Listen direkt vergleichen, liegt darin, dass der Kommunikationsaufwand hier nur minimal ist. Neben dem *reward* Wert muss lediglich der Egoismusfaktor übertragen und pro Zeitschritt nur einmal berechnet werden.

Die Berechnung des Faktors ist in Programm A.21 dargestellt. Bei der Berechnung des Aktualisierungsfaktors bietet es sich (wie oben beschrieben) nun an, entweder den eigenen Egoismusfaktor direkt zu verwenden, oder die Differenz des eigenen Egoismusfaktors mit des Faktors des Agenten, der das externe Ereignis ausgelöst hat. In beiden Fällen wird der Wert von 1,0 abgezogen. Der Egoismusfaktor selbst bestimmen sich aus dem (mit jeweils dem Produkt aus den jeweiligen *fitness* und *reward predicton* Werten gewichtet) Anteil aller *classifier*, die sich auf andere Agenten zu bewegen, sofern sie in Sicht sind. Somit ist der daraus berechnete Aktualisierungsfaktor umso größer, je ähnlicher die Agenten in ihrem Abstandsverhalten gegenüber anderen Agenten sind bzw. je eher der Agent anderen Agenten ausweicht.

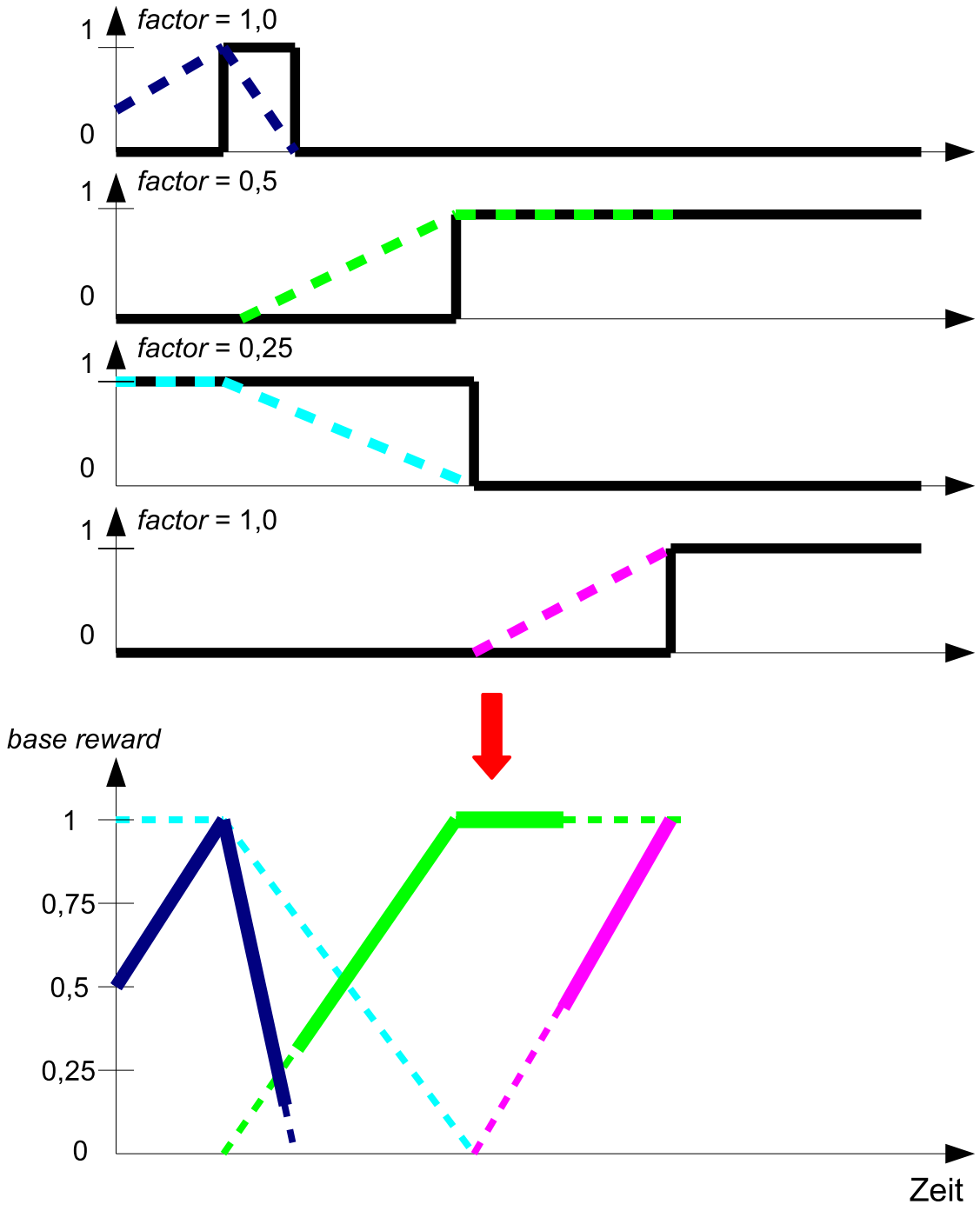

Abbildung 4.7: Beispielhafte Darstellung der Kombinierung interner und externer *reward* Werte

Kapitel 5

Analysen und Experimente

An inventor is a man who asks 'Why?' of the universe and lets nothing stand between the answer and his mind.
John Galt

In diesem Kapitel werden Tests der in Kapitel 4 und Kapitel 2.4 vorgestellten Algorithmen gesammelt dargestellt und analysiert. Ziel des Kapitels ist es, die bisher gemachten Aussagen, die nicht durch die Literatur abgedeckt sind, anhand empirischer Tests zu beweisen.

Zu Beginn werden in Kapitel 5.1 anfängliche Tests durchgeführt, bei denen ausschließlich die grundsätzlichen Algorithmen (Agent mit zufälliger Bewegung, Agent mit einfacher Heuristik und Agent mit intelligenter Heuristik) aus Kapitel 2.4 betrachtet werden. Außerdem werden dort die in Kapitel 2.2 vorgestellten Szenarien beleuchtet.

Alle drei Punkte werden zusammen mit Kapitel 5.5, in dem die XCS Parameter beschrieben und analysiert werden, als Vorbereitung für eine Analyse der XCS Varianten in Kapitel 5 dienen. Damit können gezielt die erfolgversprechenden Konfigurationen getestet werden. Insbesondere werden die drei Heuristiken auch dazu dienen, untere und obere Grenzen zu setzen. Erreicht eine XCS Variante nicht die Qualität des Algorithmus mit zufälliger Bewegung dann ist jegliche „Optimierung“ an den Parametern wenig hilfreich, denn man kann nicht unterscheiden, ob die Verbesserung von z.B. Lernen oder von zufälligerem Verhalten rührt.

Anzumerken ist außerdem, dass mit „XCS“ die XCS Variante aus Kapitel 4.2 mit den

allgemeinen Anpassungen aus Kapitel 4.1 und den Parametereinstellungen aus Kapitel 5.5 gemeint ist. Vereinzelt wird in diesem Kapitel auch von „Qualitätsdifferenz“ die Rede sein. Hierbei ist die Differenz zur Qualität des Algorithmus mit zufälliger Bewegung gemeint. Zweck der Benutzung der Differenz ist, leichter die Lerneffekte zwischen verschiedenen Szenarien darzustellen. Hinzuweisen ist inbesondere noch auf Kapitel 2.6, dort werden die einzelnen Statistiken erklärt, die hier aufgeführt werden.

Insgesamt kann dieses Kapitel allerdings nur einen kleinen Einblick geben. Die Anzahl der Parameterkombinationen sind sehr groß und können hier nur auszugsweise dargestellt werden.

5.1 Erste Analyse der Agenten ohne XCS

In diesem Abschnitt werden erste Analysen bezüglich der verwendeten Szenarien anhand des Algorithmus zufälliger Bewegung (siehe Kapitel 2.4.1), des Algorithmus mit einfacher Heuristik (siehe Kapitel 2.4.2) und des Algorithmus mit intelligenter Heuristik (siehe Kapitel 2.4.3) angefertigt. Insbesondere sollen hier aus den vorgestellten Szenarien solche gewählt werden, in welchen

- die einfache und die intelligente Heuristiken eine Qualität im Mittelfeld erreichen,
- die Berechnungszeit möglichst gering ist,
- der Anteil der Sprünge des Zielobjekts möglichst gering ist,
- der Anteil der blockierten Bewegungen der Agenten möglichst gering ist,
- ein signifikanter Unterschied zwischen der einfachen und intelligenten Heuristik besteht und
- der zufällige Algorithmus eine möglichst niedrige Qualität erreicht.

Die Ergebnisse aus der Analyse dienen als Grundlage für die vergleichende Betrachtung der Agenten mit XCS Algorithmen in Kapitel 5. Insbesondere werden sie Anhaltspunkte dafür geben, welche Szenarien welche Eigenschaften der Algorithmen testen.

5.1.1 Zielobjekt mit zufälligem Sprung (leeres Szenario)

Springt das Zielobjekt in jedem Schritt auf ein zufälliges Feld (siehe Kapitel 2.5.1), dann fehlt die Relation zwischen der Position in diesem Schritt zur Position im letzten Schritt. Für die Agenten besteht also keine Möglichkeit, Sensordaten über das Zielobjekt auszunutzen, um einen Vorteil im nächsten Schritt zu erlangen. Die Untersuchungen im Folgenden zeigen, dass sich mit dieser Form der Bewegung des Zielobjekts die Qualität des jeweiligen Algorithmus (fast) nur durch die Abdeckung des Torus durch die Agenten bestimmt ist.

Betrachtet man das Szenario ohne Hindernisse gibt sich ein klares Bild (siehe Tabelle 5.1), die intelligente Heuristik ist etwas besser als der des zufälligen Agenten und der einfachen Heuristik. Ein möglichst weiträumiges Verteilen auf dem Torus führt zum Erfolg. Dies zeigt sich auch in einem hohen Wert der Abdeckung, denn genau das wird mit dem völlig zufällig springenden Agenten getestet. Ebenfalls ist die Zahl der blockierten Bewegungen deutlich niedriger, was sich auch mit der Haltung des Abstands erklären lässt.

Die einfache Heuristik schneidet dagegen etwas schlechter als eine zufällige Bewegung ab. Zwar ist die Zahl der blockierten Bewegungen geringer, was sich dadurch erklären lässt, dass die einfache Heuristik zumindest an einem Punkt eine Sichtbarkeitsprüfung für die Richtung durchführt, in der sie sich bewegen möchte (nämlich wenn das Zielobjekt in Sicht ist), andererseits ist die Abdeckung etwas geringer. Ursache dafür ist wahrscheinlich, dass, wenn mehrere Agenten das Zielobjekt in Sichtweite haben, alle sich auf das Zielobjekt bewegen. Dadurch wird die zufällige Verteilung der Agenten auf dem Spielfeld gestört, was letztlich zu einer niedrigeren Abdeckung des Torus führt.

Bezüglich der Anzahl der Agenten ergeben sich keine Besonderheiten, die Verhältnisse zwischen den Qualitäten bleibt ähnlich. Nur mit steigender Agentenzahl steigt die Zahl der blockierten Bewegungen (aufgrund größerer Anzahl von blockierten Feldern), während die Abdeckung sinkt (aufgrund sich überlappender Überwachungsreichweiten).

5.1.2 Zielobjekt mit zufälligem Sprung (Säulenszenario)

Für das Zielobjekt treffen hier dieselben Überlegungen, wie auch schon in Kapitel 5.1.1 erwähnt, zu. Auch ergeben sich im Säulenszenario (siehe Tabelle 5.2) erwartungsgemäß

Tabelle 5.1: Zufällige Sprünge des Zielobjekts im leeren Szenario ohne Hindernisse

Algorithmus	Agentenzahl	Blockierte Bewegungen	Abdeckung	Qualität
Zufällige Bewegung	8	2,82%	73,78%	32,36%
Einfache Heuristik	8	2,79%	73,22%	32,10%
Intelligente Heuristik	8	0,64%	81,26%	35,91%
Zufällige Bewegung	12	4,32%	69,55%	44,75%
Einfache Heuristik	12	4,19%	68,88%	43,86%
Intelligente Heuristik	12	1,49%	77,60%	49,49%
Zufällige Bewegung	16	5,82%	64,28%	54,55%
Einfache Heuristik	16	5,66%	63,65%	53,99%
Intelligente Heuristik	16	2,85%	71,44%	60,73%

ähnliche Werte wie im Fall des leeren Szenarios ohne Hindernisse (siehe Tabelle 5.1). Durch geringere Sicht und höhere Zahl an blockierten Bewegungen ergibt sich jeweils eine geringere Abdeckung und auch jeweils eine geringere Qualität. Auch hier ergeben sich keine Besonderheiten bezüglich der Agenten.

Um die Anzahl blockierter Bewegungen gering zu halten, sich überlappende Überwachungsreichweiten zu vermeiden und die Szenarien möglichst schwer zu halten (d.h. niedrige Qualität des Algorithmus mit zufälliger Bewegung) werden die Tests sich im Folgenden deshalb auf den Fall mit **8 Agenten** beschränken.

Tabelle 5.2: Zufällige Sprünge des Zielobjekts in einem Säulenszenario

Algorithmus	Agentenzahl	Blockierte Bewegungen	Abdeckung	Qualität
Zufällige Bewegung	8	4,45%	72,11%	32,13%
Einfache Heuristik	8	4,08%	71,70%	31,99%
Intelligente Heuristik	8	2,34%	79,61%	35,29%
Zufällige Bewegung	12	5,93%	67,72%	44,44%
Einfache Heuristik	12	5,67%	67,23%	43,81%
Intelligente Heuristik	12	3,62%	75,86%	49,34%
Zufällige Bewegung	16	7,62%	62,53%	54,26%
Einfache Heuristik	16	7,23%	62,00%	53,58%
Intelligente Heuristik	16	5,18%	69,91%	60,43%

5.1.3 Zielobjekt mit zufälligem Sprung (Zufällig verteilte Hindernisse)

Auch hier gelten wieder dieselben Überlegungen für das Zielobjekt, die in Kapitel 5.1.1 gemacht wurden. Und auch für alle Einstellungen von λ_h und λ_p (siehe Kapitel 2.2.2) stellt sich ebenfalls ein eindeutiges Bild dar (siehe Tabelle 5.3). So liegt die intelligente Heuristik liegt wieder vorne, gefolgt wieder von der einfachen Heuristik und der zufälligen Bewegung. Im Fall mit vielen Hindernissen ($\lambda_h = 0{,}2$) liegt die einfache Heuristik trotz höherer Abdeckung hinter der zufälligen Bewegung. Dies ist wohl auf einen Zufall zurückzuführen, ändert man den *random seed* Wert oder erhöht man die Anzahl der Experimente von 10 auf 30 ergibt sich wieder die oben genannte Reihenfolge.

Tabelle 5.3: Zufällige Sprünge des Zielobjekts in einem Szenario mit Hindernissen

Algorithmus	λ_h	λ_p	Blockierte Bewegungen	Abdeckung	Qualität
Zufällige Bewegung	0,2	0,99	12,44%	62,50%	34,54%
Einfache Heuristik	0,2	0,99	10,04%	63,02%	34,48%
Intelligente Heuristik	0,2	0,99	12,71%	68,22%	37,89%
Zufällige Bewegung	0,1	0,99	7,58%	68,33%	32,81%
Einfache Heuristik	0,1	0,99	6,15%	68,49%	33,36%
Intelligente Heuristik	0,1	0,99	6,50%	74,81%	36,29%
Zufällige Bewegung	0,1	0,5	10,12%	66,01%	32,03%
Einfache Heuristik	0,1	0,5	8,57%	66,52%	32,38%
Intelligente Heuristik	0,1	0,5	9,29%	72,63%	35,12%

Kommt das Zielobjekt in Sicht, so weist der Agent mit einfacher Heuristik eine geringere Zahl an blockierten Bewegungen als der Agent mit zufälliger Bewegung auf. Das kann man damit begründen, dass er davon ausgehen kann, dass sich in dieser Richtung wahrscheinlich kein Hindernis befindet, da die Sicht nicht blockiert ist. Im Gegensatz dazu beachtet der Agent mit zufälliger Bewegung Hindernisse überhaupt nicht, läuft erwartungsgemäß öfters gegen solche und bleibt deswegen wiederholt stehen. Wie die Ergebnisse zeigen ist der Unterschied zwischen beiden Agenten ist besonders hoch in Szenarien mit größerem Anteil an Hindernissen.

Im Vergleich zur einfachen Heuristik und im Gegensatz zum Säulenszenario scheint insbesondere die intelligente Heuristik Probleme mit den Hindernissen zu haben (viele

blockierte Bewegungen). Da Hindernisse in der Heuristik nicht beachtet werden, bewirkt die Strategie der maximalen Ausbreitung der Agenten wahrscheinlich ein „Drücken“ dieser Agenten gegen die Hindernisse, da sich von anderen Agenten wegbewegt wird anstatt zufällig zu laufen.

Schließlich ist zu sehen, dass die Agenten in einem Szenario mit höherem Verknüpfungsfaktor (der Fall mit $\lambda_h = 0{,}1$ und $\lambda_p = 0{,}99$ im Vergleich zum Fall mit $\lambda_h = 0{,}1$ und $\lambda_p = 0{,}5$) besser abschneiden. Dies liegt daran, dass Szenarien mit hohem Verknüpfungsfaktor bedeuten, dass viele Hindernisse zusammenhängend einen großen Block bilden und somit dem Szenario ohne Hindernisse ähnlich sind, da es eher größere zusammenhängende Flächen gibt.

Insgesamt ist zu sagen, dass es diese Form der Bewegung des Zielobjekts genau die Eigenschaft der intelligenten Heuristik testet, sich auf dem Feld zu verbreiten. Abbildung 5.1 stellt dies nochmal grafisch dar, das Verhältnis zwischen Abdeckung und Qualität verhält sich in jedem der weiter oben betrachteten Szenarien (jeweils für den Fall mit 8 Agenten) gleich.

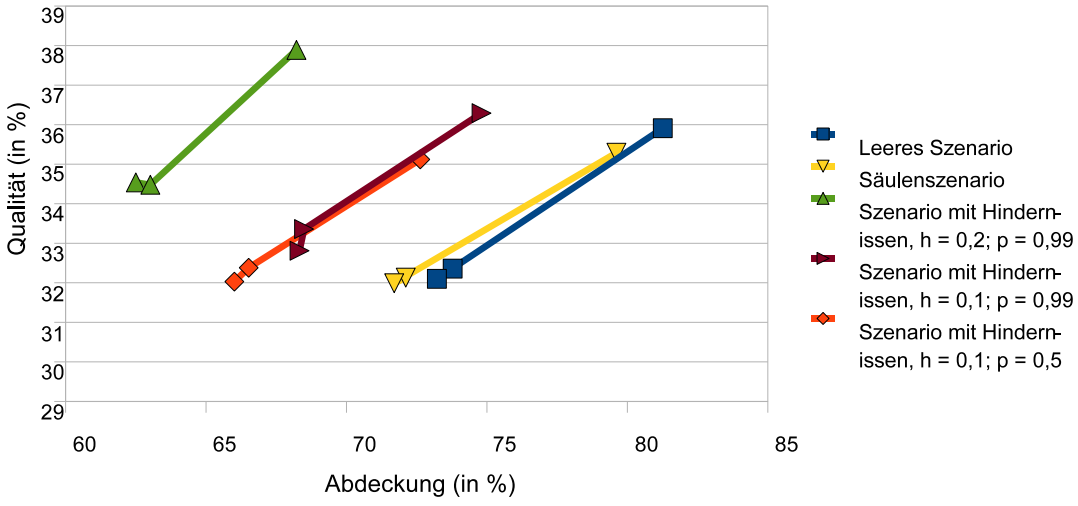

Abbildung 5.1: Zusammenhang zwischen der Abdeckung und der Qualität eines Algorithmus, getestet in verschiedenen Szenarien

Da die Agenten mit einfacher Heuristik sich gegenüber den sich zufällig bewegenden

Agenten nicht durchsetzen konnte, könnte der Test mit einem Zielobjekt mit zufälligem Sprung dazu dienen, diese Eigenschaften auch bei lernenden Agenten zu testen. Da das Zielobjekt dort jedoch häufig durch Sprung in die Sichtweite eines Agenten für diesen zufällige Ereignisse auslöst, ist zu erwarten, dass dadurch das Lernen stark gestört wird.

5.1.4 Zielobjekt mit zufälliger Bewegung bzw. einfacher Richtungsänderung

Als einfachster Test eines sich bewegenden Agenten ist die zufällige Bewegung (siehe Kapitel 2.5.2). Nun wird untersucht, ob dieser Art der Bewegung für die Tests sinnvoll ist oder ob eine andere Form der Bewegung für das Zielobjekt im Rahmen der Tests günstiger erscheint.

Am nächsten kommt der Bewegungstyp mit einfacher Richtungsänderung (siehe Kapitel 2.5.3). Gemeinsam haben beide Bewegungstypen, dass der jetzige Ort des Zielobjekts maximal zwei Felder (die maximale Geschwindigkeit des Zielobjekts in den Tests) vom Ort der vorangegangenen Zeiteinheit entfernt ist. Im Gegensatz zum zuvor besprochenen zufälligen Sprung ist hier das lokale Einfangen eher von Relevanz. Der Ort, an dem sich das Zielobjekt im nächsten Schritt befinden wird, ist zumindest von der aktuellen Position abhängig, wenn das Zielobjekt auch schneller sein, kann als andere Agenten. Hingegen unterscheiden sie sich eindeutig im Bewegungsmuster des Zielobjekts. Dieses kehrt mit zufälliger Bewegung nach 2 Schritten mit Wahrscheinlichkeit von $\frac{1}{4}$ auf das ursprüngliche Feld zurück.

Außerdem bezieht der Bewegungstyp mit einfacher Richtungsänderung Hindernisse in die Entscheidung über die nächste Aktion mit ein. Dies führt ebenfalls zu einer deutlich geringeren Anzahl von blockierten Bewegungen. Wie die Anzahl der Sprünge des Zielagenten in Tabellen 5.4 und 5.5 zeigen, ist es den Agenten beim Zielobjekt mit zufälliger Bewegung deutlich öfters gelungen, ihn in seiner Bewegung zu blockieren. Wie auch an der Qualität abzulesen ergibt sich dadurch ein deutlich leichteres Szenario für beide Heuristiken, während es kaum Unterschiede in der Qualität bei der zufälligen Bewegung der Agenten ergibt.

Da die Differenz der Qualität zwischen der einfachen und intelligenten Heuristik beim **Zielobjekt mit einfacher Richtungsänderung** größer, der Anteil der Sprünge des Zielobjekts kleiner, der Anteil der blockierten Bewegungen der Agenten kleiner und das Problem für den Algorithmus mit zufälliger Bewegung schwieriger ist, soll im Weiteren das Zielobjekt mit zufälliger Bewegung nicht mehr verwendet werden.

In den Tabellen bezieht sich der Eintrag „Sprünge“ auf den Anteil vom Zielobjekt durchgeführter Sprünge, „Blockiert“ auf den Anteil blockierter Bewegungen des Agenten und „Zufällig bewegend“ bzw. „Einfache Richtungsänderung“ auf das Zielobjekt.

Tabelle 5.4: Vergleich von Zielobjekt mit zufälliger Bewegung und einfacher Richtungsänderung (zufälliges Szenario mit $\lambda_h = 0{,}1$, $\lambda_p = 0{,}99$)

Algorithmus	Sprünge	Blockiert	Abdeckung	Qualität
Zufällig bewegend				
Zufällige Bewegung	0,01%	7,49%	66,63%	33,96%
Einfache Heuristik	0,41%	11,51%	59,72%	79,99%
Intelligente Heuristik	0,36%	10,76%	65,87%	81,50%
Einfache Richtungsänderung				
Zufällige Bewegung	0,00%	7,54%	68,31%	31,66%
Einfache Heuristik	0,06%	8,68%	62,31%	57,95%
Intelligente Heuristik	0,08%	8,57%	68,28%	61,72%

Tabelle 5.5: Vergleich von Zielobjekt mit zufälliger Bewegung und einfacher Richtungsänderung (Säulenszenario)

Algorithmus	Sprünge	Blockiert	Abdeckung	Qualität
Zufällig bewegend				
Zufällige Bewegung	0,00%	4,34%	72,27%	31,80%
Einfache Heuristik	0,07%	8,77%	62,87%	78,34%
Intelligente Heuristik	0,04%	6,40%	69,98%	80,54%
Einfache Richtungsänderung				
Zufällige Bewegung	0,00%	4,30%	72,28%	29,17%
Einfache Heuristik	0,01%	6,29%	65,80%	56,19%
Intelligente Heuristik	0,01%	4,58%	72,44%	60,41%

Vergleicht man die Ergebnisse für **das Säulenzenario** aus Tabelle 5.5 mit den Ergebnissen für das leere Szenario aus Tabelle 5.6 ergeben sich kaum Unterschiede. Das leere Szenario soll im Folgenden also nicht weiter für Tests herangezogen werden.

Tabelle 5.6: Vergleich von Zielobjekt mit zufälliger Bewegung und einfacher Richtungsänderung (leeres Szenario ohne Hindernisse)

Algorithmus	Sprünge	Blockiert	Abdeckung	Qualität
Zufällig bewegend				
Zufällige Bewegung	0,00%	2,71%	73,85%	32,57%
Einfache Heuristik	0,06%	11,51%	63,65%	79,97%
Intelligente Heuristik	0,02%	4,71%	71,15%	81,59%
Einfache Richtungsänderung				
Zufällige Bewegung	0,00%	2,75%	73,81%	30,99%
Einfache Heuristik	0,01%	4,98%	66,61%	58,38%
Intelligente Heuristik	0,01%	2,93%	73,37%	62,48%

5.2 Auswirkung der Geschwindigkeit des Zielobjekts

Lässt keine der beiden Parteien, Agenten und das Zielobjekt, Sensordaten über die jeweils andere Partei in die Entscheidung über die nächste Aktion mit einfließen, so spielt das Verhältnis der Geschwindigkeiten beider Parteien langfristig keine Rolle (sofern beide eine Geschwindigkeit größer 0 besitzen). Dies hat man im letzten Kapitel 5.1.4 daran gesehen, dass bei Agenten mit zufälliger Bewegung sich beim Vergleich zwischen beiden Bewegungstypen kaum Unterschiede in der Qualität auftreten, während dies bei der einfachen und intelligenten Heuristik der Fall war. Im Folgenden werden nun also die Fälle untersucht, bei denen mindestens einer der Parteien die andere Partei mit in die Überlegung miteinbezieht.

5.2.1 Zielobjekt mit einfacher Richtungsänderung

In Abbildung 5.2 sind die Testergebnisse für einen Test auf dem Säulenszenario dargestellt, bei dem sich das Zielobjekt mit einfacher Richtungsänderung bewegt. Es ist keine

Korrelation zwischen der Geschwindigkeit und der Qualität des Algorithmus mit zufälliger Bewegung festzustellen, nur bei Geschwindigkeit 0,0 scheint es ein deutlich besseres Ergebnis zu geben. Das lässt sich aber durch die Anfangskonfiguration erklären, beim Säulenszenario startet das Zielobjekt in der Mitte mit maximalem Abstand zu den Hindernissen, ist also immer optimal in Sicht.

Da ein sich zufällig bewegender Agent nichts über die Umwelt wissen muss, stellt dessen erreichte Qualität also eine Untergrenze dar, ein Agent muss mehr als diesen Wert erreichen, damit man sagen kann, dass er etwas gelernt hat.

Bei den Testergebnissen für die einfache und die intelligente Heuristik sind dagegen folgende drei Punkte anzumerken:

- Es existiert eine Korrelation zwischen Qualität und Geschwindigkeit,
- es gibt einen Knick bei Geschwindigkeit 1,0 und
- es ist ein fast stetiger Anstieg der Differenz zwischen der einfachen und der intelligenten Heuristik zu verzeichnen.

Der Knick lässt sich dadurch erklären, dass es dem Zielobjekt oberhalb dieser Geschwindigkeit möglich ist, eventuelle Verfolger abzuschütteln. Der Anstieg der Differenz lässt sich dadurch erklären, dass die Abdeckung des Gebiets eine immer größere Rolle spielt, als die Verfolgung des Zielobjekts.

Da die Heuristiken das obere Limit angeben und so gebaut sind, dass sie sich immer für die jeweilige in ihren Augen beste Aktion entscheiden, wohingegen die auf XCS basierenden Varianten dies beispielsweise mit der Auswahlart *tournament selection* mit $p = 0{,}84$ in bestenfalls 84% der Fälle tun (siehe Kapitel 5.5.8). Es ist deshalb anzunehmen, dass in Verbindung mit XCS auch niedrigere Geschwindigkeiten betrachtet werden können, ohne Agenten zu erhalten, die ausschließlich auf Verfolgung aus sind.

Im Szenario mit zufällig verteilten Hindernissen (siehe Abbildung 5.3) gelten obige Punkte nur eingeschränkt, insbesondere ist der Knick schwierig auszumachen.

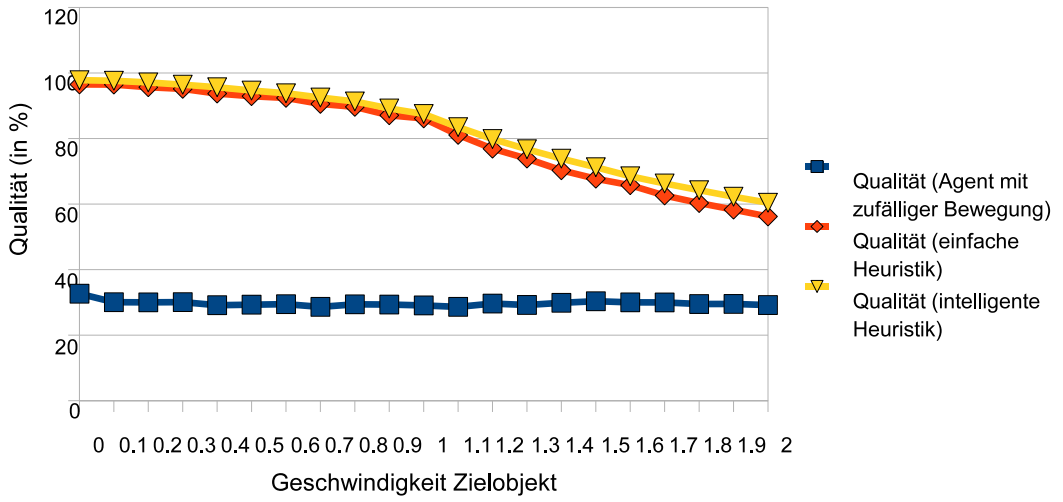

Abbildung 5.2: Auswirkung der Zielgeschwindigkeit auf Agenten mit bestimmten Heuristiken (Zielobjekt mit einfacher Richtungsänderung, Säulenszenario)

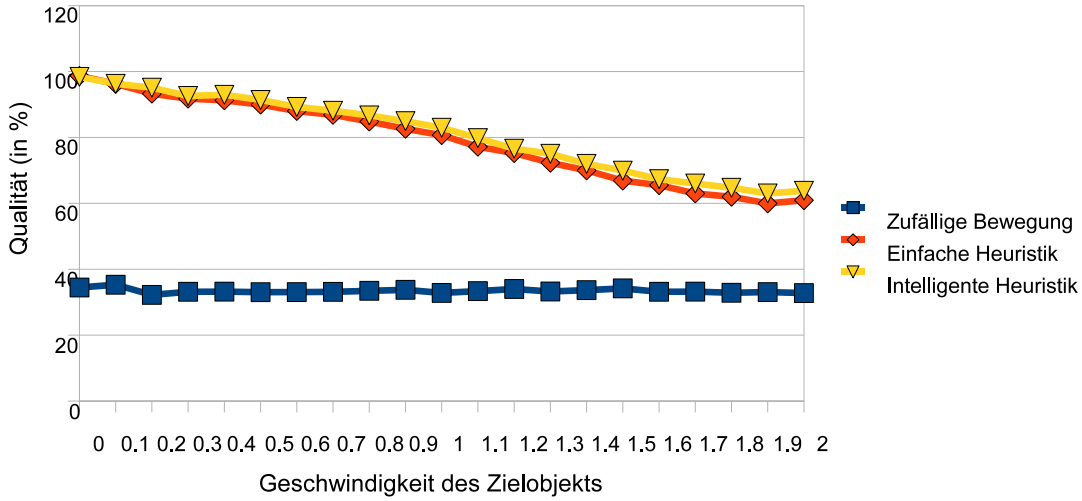

Abbildung 5.3: Auswirkung der Zielgeschwindigkeit auf Agenten mit bestimmten Heuristiken (Zielobjekt mit einfacher Richtungsänderung, Szenario mit zufällig verteilten Hindernissen, $\lambda_h = 0{,}2$, $\lambda_p = 0{,}99$)

5.2.2 Zielobjekt mit intelligenter Bewegung

In Abbildung 5.4 und Abbildung 5.5 werden im Säulenszenario bzw. Szenario mit zufällig verteilten Hindernissen wieder die Heuristiken bei unterschiedlichen Geschwindigkeiten des Zielobjekts verglichen. Beim Säulenszenario ist wieder der Knick wie beim Fall mit Zielobjekt mit einfacher Richtungsänderung (siehe Kapitel 5.2.1) zu beobachten.

Anzumerken ist hier, dass bei Agenten mit zufälliger Bewegung ein stetiger Abfall der Qualität zu verzeichnen ist, das Zielobjekt einem sich zufällig bewegenden Agenten also mit steigender Geschwindigkeit immer etwas besser ausweichen kann.

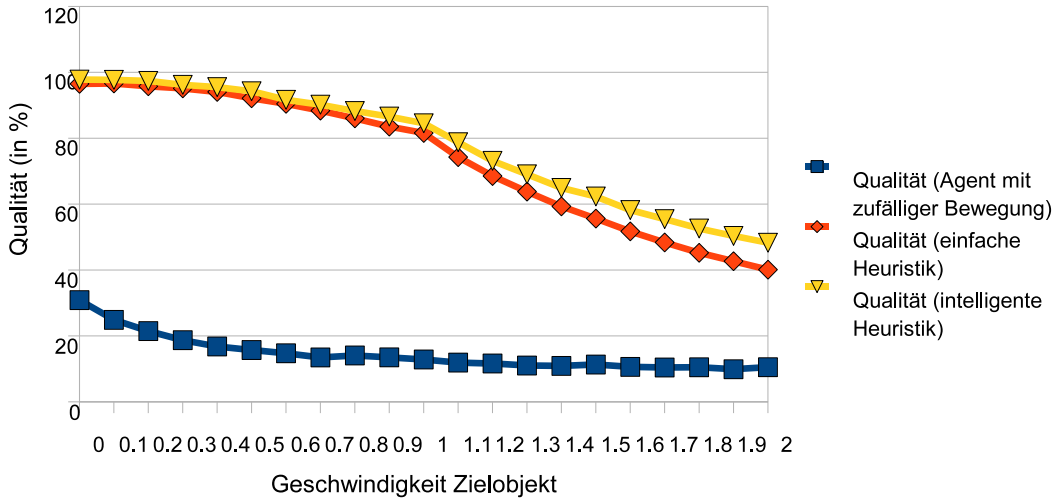

Abbildung 5.4: Auswirkung der Zielgeschwindigkeit (intelligentes Zielobjekt, Säulenszenario) auf Agenten mit Heuristik

Aufgrund der hohen Qualität bei niedrigen Geschwindigkeiten und aufgrund des höheren Bedarfs an Kollaboration, sollen in Tests in dieser Arbeit, soweit nicht anders angegeben, nur Zielobjekte mit einer **Geschwindigkeit 2** getestet werden. Insbesondere vermeidet dies ein einfaches, stetiges Verfolgen des Zielobjekts.

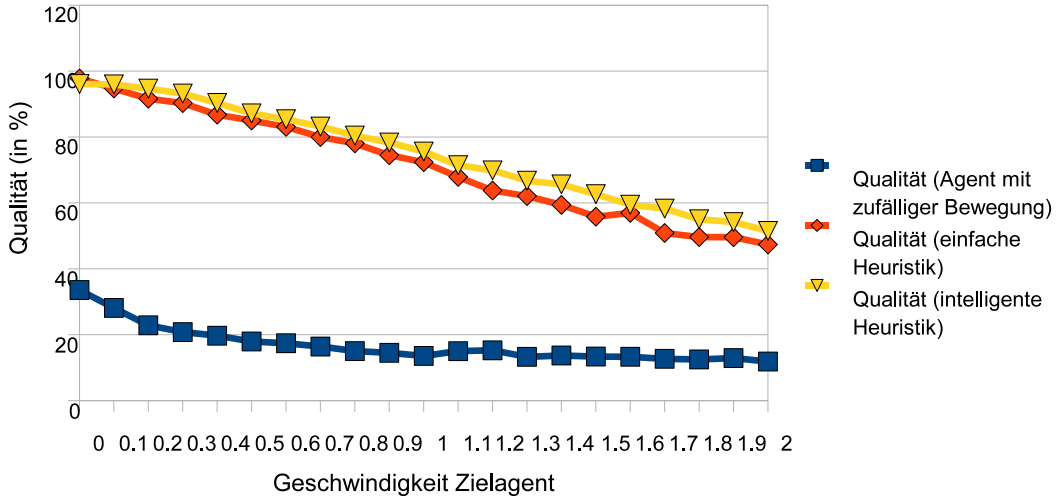

Abbildung 5.5: Auswirkung der Zielgeschwindigkeit (intelligentes Zielobjekt, Szenario mit zufällig verteilten Hindernissen, $\lambda_h = 0{,}2$, $\lambda_p = 0{,}99$) auf Agenten mit Heuristik

5.3 Heuristiken im schwierigen Szenario

Für verschiedene Anzahl von Schritten sind für die drei Agententypen in Abbildung 5.6 die jeweiligen Qualitäten aufgeführt. Wie man beim Vergleich zwischen zufälliger Bewegung und einfacher Heuristik sehen kann, ist es nicht nur entscheidend, in den letzten Bereich am rechten Rand des Szenarios vorzudringen, sondern auch, dort den Agenten zu verfolgen und in diesem Bereich zu bleiben. Deutlich zeigen sich hier die Vorzüge der intelligenten Heuristik, durch das Bestreben, Agenten auszuweichen, hat es dieser Algorithmus leichter, durch die Öffnungen in von Agenten unbesetzte Bereiche vorzudringen.

Der Unterschied zwischen einfacher und intelligenter Heuristik zeigt auch, dass in diesem Szenario ein deutlich größeres Lernpotential, was die Einbeziehung von wahrgenommenen Agentenpositionen betrifft, für Agenten besteht. Wie später in Kapitel 5.7.1 gezeigt wird, können in diesem Szenario unter anderem deshalb auf XCS basierte Agenten ihre Vorteile besonders gut ausspielen und erreichen sogar bessere Ergebnisse als die intelligente Heuristik sofern sie genug Zeit zum Lernen erhalten.

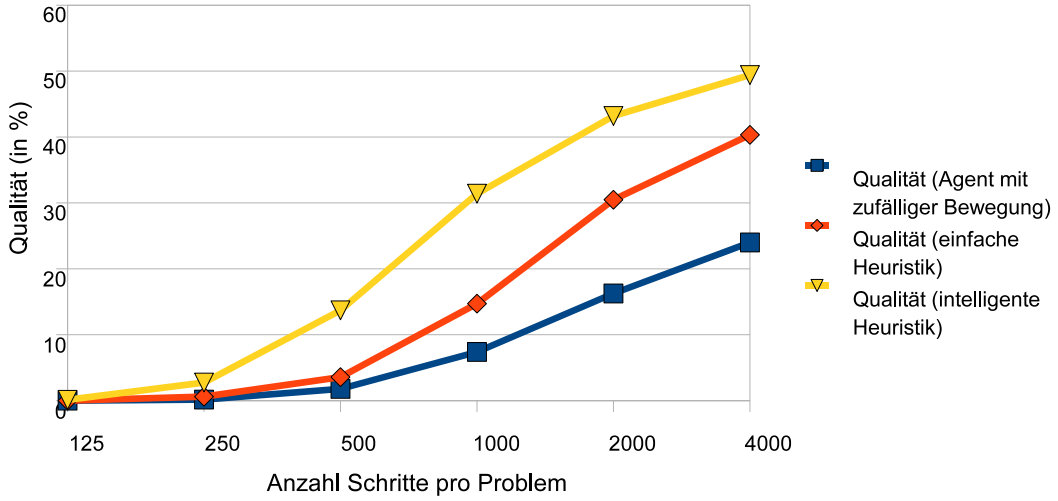

Abbildung 5.6: Auswirkung der Anzahl der Schritte (schwieriges Szenario, ohne Richtungsänderung) auf Qualität von Agenten mit Heuristik

5.4 Zusammenfassung der Tests mit Heuristiken

Folgende Ergebnisse konnten in diesem Kapitel gewonnen werden:

- Um die Anzahl blockierter Bewegungen gering zu halten, sich überlappende Überwachungsreichweiten zu vermeiden und die Szenarien möglichst schwer zu halten wird sich im Folgenden auf den Fall mit 8 Agenten beschränkt.

- Kapitel 5.1.4 hat gezeigt, dass die einfachste Implementierung eines Zielobjekts, das Zielobjekt mit zufälliger Bewegung, eher nicht für Tests benutzt werden sollte. Die Einfachheit des Algorithmus rechtfertigt nicht die höhere Zahl der Sprünge und blockierten Bewegungen. In weiteren Tests werden deswegen immer nur Zielobjekte mit einfacher Richtungsänderung getestet.

- Bei der Analyse der Geschwindigkeit des Zielobjekts in Kapitel 5.2 ergab sich bei einer Geschwindigkeit von 1 ein Knick, ab dem kollaboratives Verhalten gegenüber sturem Verfolgen an Bedeutung gewann. Für weitere Tests wird deshalb eine Geschwindigkeit von 2 verwendet.

- Beim Test des schwierigen Szenarien in Kapitel 5.3 wurde zum einen festgestellt, dass ein einfaches Verfolgen des Zielobjekts nicht zum Ziel führen kann. Agenten mit intelligenter Heuristik konnten trotzdem Erfolge zeigen, da sie sich gegenseitig auf frei Felder durch die Öffnungen hindurch drängen.

- Der Vergleich von Agenten intelligenten Heuristik mit Agenten mit zufälliger Bewegung Aufschluss darüber geben, wieviel und welche Aspekte ein Agent in einem solchen Szenario überhaupt lernen kann. Große Unterschiede zwischen intelligenter und einfacher Heuristik weisen beispielsweise darauf hin, dass die Verteilung auf dem Torus wichtiger ist, als das Hinterherlaufen. Dies sieht man insbesondere am Extrembeispiel des Zielobjekts mit zufälligem Sprung in Kapitel 5.1.1.

5.5 Beschreibung und Analyse der XCS Parameter

Motivation der Untersuchung der Parameter waren zum einen die vom Standard abweichenden Überwachungsszenarien und zum anderen der neuartige Algorithmus SXCS. Primäres Ziel dieses Kapitels ist es, darzustellen, dass die im Vergleich zu XCS bessere Qualität nicht von bestimmten Parametereinstellungen sondern vom Algorithmus selbst herrührt.

Anzumerken sei, dass alle Tests jeweils mit den in Tabelle 5.7 angegebenen Parameterwerten durchgeführt wurden und bei jedem Test jeweils nur der zu untersuchende Wert verändert wurde. Mit dem Ziel, synchronisierte und vergleichbare Daten zu haben, wurden die Tests in mehreren Etappen durchgeführt. Damit entsprechen die hier aufgeführten Testergebnisse nur den endgültigen Ergebnissen. Eine umfangreiche Untersuchung der Parameter war notwendig, da keine Vergleichsarbeiten mit der hier verwendeten Problemstellung existieren. Nicht auszuschließen ist bei diesem Vorgehen allerdings, dass es mehrdimensionale Kombinationen von Parameterwerten gibt, bei denen wesentliche bessere Ergebnisse erzielt werden, wobei dies bei den Tests nicht aufgefallen ist.

Liegt beim Test die erreichte Qualität unter der des zufälligen Algorithmus, ist beim Vergleich der Parameterwerte Vorsicht geboten. Die Ursache für die Verbesserung kann sein, dass der Algorithmus nicht besser lernt, sondern sich umgekehrt eher wie der zufällige Algorithmus verhält. Deswegen ist stets der Vergleich mit der Qualität des sich zufällig bewegenden Algorithmus durchzuführen.

Die Einstellungen der XCS Parameter der durchgeführten Experimente entsprechen weitgehend den Vorschlägen in [BW01] („Commonly Used Parameter Settings“). Eine Auflistung findet sich in Tabelle 5.7. Nachstehend werden Parameter besprochen, die entweder in der Empfehlung offen gelassen sind, also klar vom jeweiligen Szenario abhängen, und solche, bei denen von der Empfehlung abgewichen wurde. Es wurden viele weitere Veränderungen getestet, in vielen Fällen war die Standardeinstellung jedoch passend. Abschluss der Besprechung der Parameter bietet Kapitel 5.5.9 in dem eine komplette Übersicht über die wichtigsten Parameterwerte aufgelistet wird.

5.5.1 Parameter *max population N*

Der Wert von *max population N* bezeichnet die maximalen Größe der *classifier set* Liste. Nach [BW01] sollte N so groß gewählt werden, dass *covering* nur zu Beginn eines Durchlaufs stattfindet, also die Anzahl der neu erstellten *classifier* gegen Null geht. In Abbildung 5.7 ist dies für das angegebene Säulenszenario ab einer Populationsgröße von etwa 512 erfüllt.

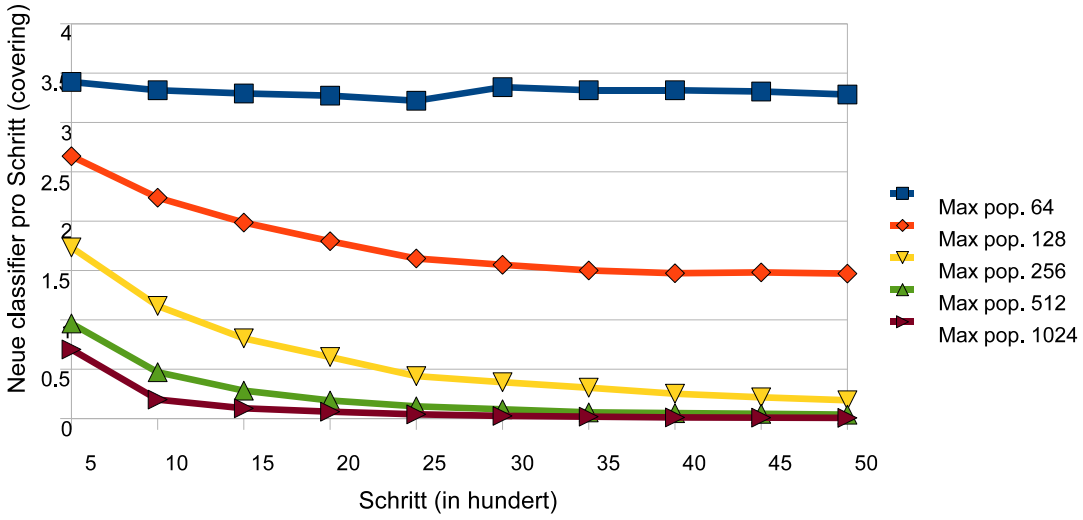

Abbildung 5.7: Auswirkung der maximalen Populationsgröße auf die Anzahl der durch *covering* neu erstellten *classifier* (**Säulenszenario**, Zielobjekt mit einfacher Richtungsänderung, Geschwindigkeit 1, Agenten mit SXCS, *tournament selection*, $p = 0{,}84$)

Bei der Wahl eines geeigneten Werts für N spielen außerdem die Konvergenzgeschwindigkeit und die Laufzeit eine Rolle. Einen allgemein besten Wert für N gibt es nicht, denn er hängt insbesondere von der durch das Szenario und der durch die Länge des *condition* Vektors gegebenen Möglichkeiten ab.

Die Zahl der Möglichkeiten wird wesentlich durch die Anzahl von verschiedenen *classifier* bestimmt, die durch die *covering* Funktion während eines bestimmten Laufs konstruiert werden können. Würde man beispielsweise weitere Zielobjekte auf das Feld setzen, könnten eine Reihe weiterer Situationen auftreten. So könnten z.B. Zielobjekte in zwei unterschiedlichen Richtungen in Sichtweite eines Agenten kommen.

Beim Szenario ohne Hindernisse fällt dagegen eine ganze Anzahl von Möglichkeiten heraus, was man in Abbildung 5.8 als Vergleich sehen kann. Selbiges stellt man im schwieriges Szenario fest (siehe Abbildung 5.9), hier gibt es zwar wesentlich mehr Hindernisse, jedoch sind sie immer in Sicht. Auch ist das Zielobjekt in relativ wenigen unterschiedlichen Situationen in Sicht. Beides verringert die Zahl der möglichen Situationen wieder.

Im Szenario mit zufällig verteilten Hindernissen ($\lambda_h = 0{,}2$ und $\lambda_p = 0{,}99$) gilt dies jedoch nicht. Zwar ergeben sich aufgrund der groben Auflösung der Sensoren dort im Vergleich zum Säulenszenario kaum neue Situationen, die durch *covering* abgedeckt werden müssen (siehe Abbildung 5.10), jedoch existieren auch genug hindernisfreie Freiräume.

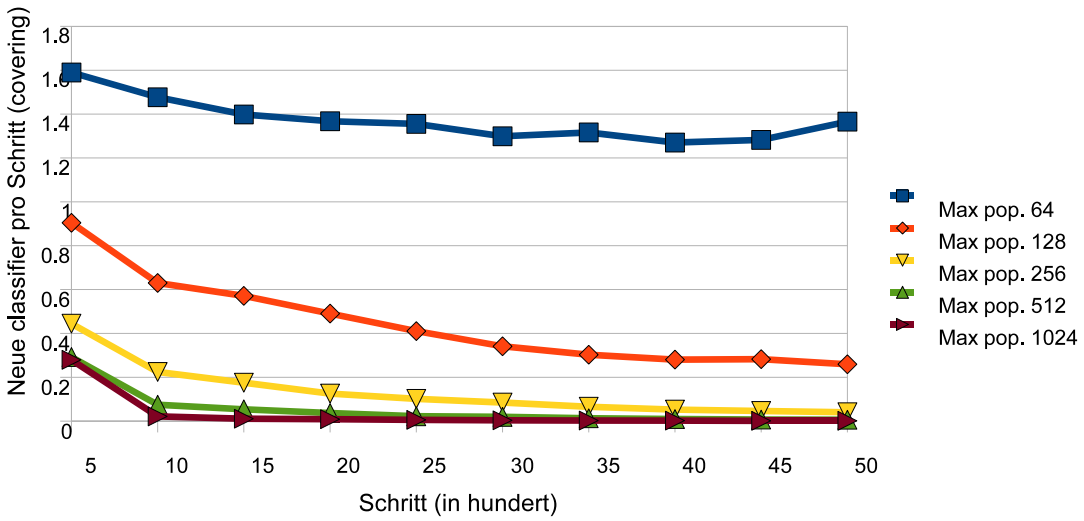

Abbildung 5.8: Auswirkung der maximalen Populationsgröße auf die Anzahl der *classifier* die durch *covering* neu erstellt werden (**leeres Szenario ohne Hindernisse**, Zielobjekt mit einfacher Richtungsänderung, Geschwindigkeit 1, Agenten mit SXCS, *tournament selection*, $p = 0{,}84$)

5.5.2 Overhead durch Populationsgröße

Für den Overhead (d.h. die Zeit, die 8 Agenten mit zufälliger Bewegung benötigen) ergab sich eine mittlere Laufzeit von 2,02s pro Experiment bei 500 Schritten (bzw. 8,34s bei 2.000 Schritten). Dieser Wert wird von den Messwerten abgezogen und dort jeweils als „Zeitdifferenz“ bezeichnet (siehe Abbildung 5.11). Da im Wert noch der Overhead des

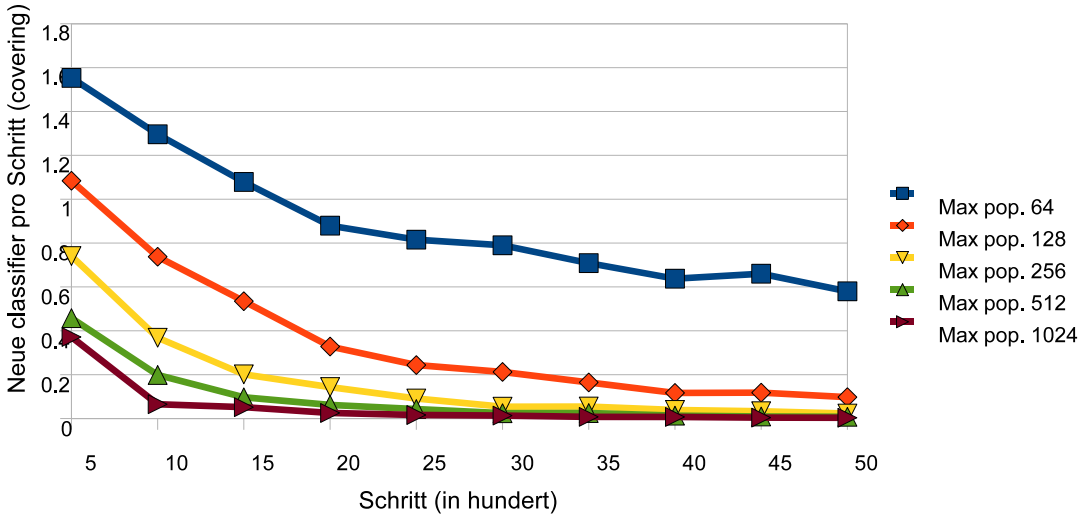

Abbildung 5.9: Auswirkung der maximalen Populationsgröße auf die Anzahl der *classifier* die durch *covering* neu erstellt werden (**schwieriges Szenario**, Zielobjekt mit einfacher Richtungsänderung, Geschwindigkeit 1, Agenten mit SXCS, *tournament selection*, $p = 0{,}84$)

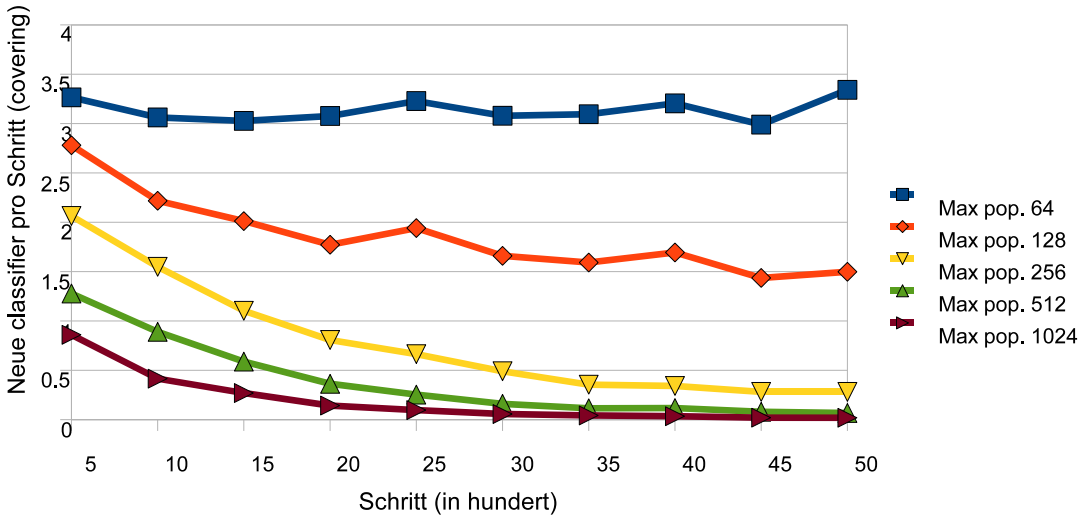

Abbildung 5.10: Auswirkung der maximalen Populationsgröße auf die Anzahl der *classifier* die durch *covering* neu erstellt werden (**Szenario mit zufällig verteilten Hindernissen**, $\lambda_h = 0{,}2$, $\lambda_p = 0{,}99$, Zielobjekt mit einfacher Richtungsänderung, Geschwindigkeit 1, Agenten mit SXCS, *tournament selection*, $p = 0{,}84$)

XCS Algorithmus selbst enthalten ist, fällt der Wert erst relativ stark (bis etwa $N = 256$ und konvergiert dann langsam (siehe Abbildung 5.12).

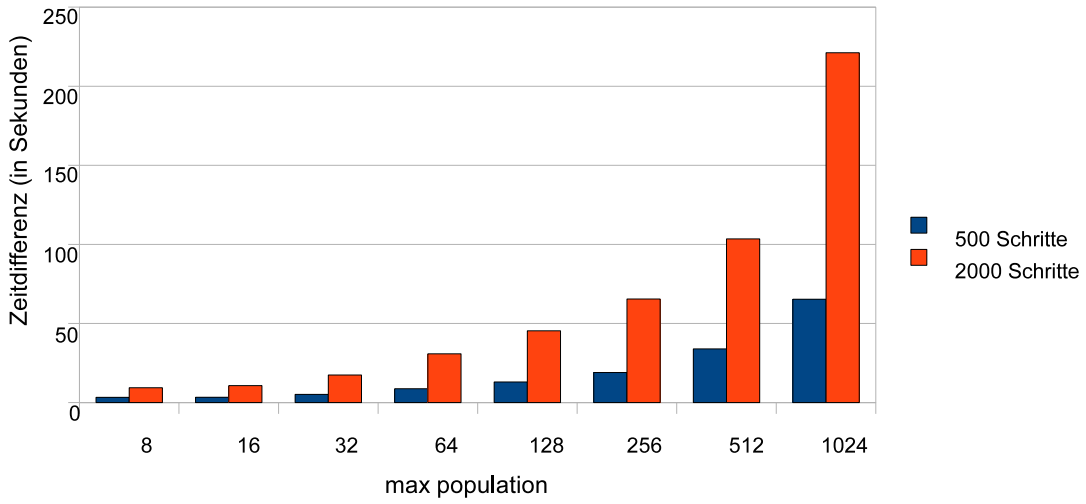

Abbildung 5.11: Darstellung der Auswirkung des Parameters *max population* N auf die Laufzeit (Säulenszenario, Zielobjekt mit einfacher Richtungsänderung, Geschwindigkeit 1, Agenten mit SXCS, *tournament selection*, $p = 0{,}84$)

Wie im vorherigen Abschnitt gesehen, sind die wichtigsten *classifier* mit Populationsgröße 512 (bzw. 256 im leeren und im schwierigen Szenario) bereits abgedeckt und eine Erhöhung der Populationsgröße schlägt sich negativ auf die Laufzeit nieder. Da den Agenten das Szenario unbekannt ist, wird für alle Szenarien deshalb derselbe Wert benutzt. Alles in allem scheint somit $N = 512$ die beste Parametereinstellung zu sein, was Laufzeit und *covering* betrifft.

Die Tests liefen auf einem T7500 mit 2,2 GHz in einem einzelnen Thread. Als Vergleich hierzu wurde auch der Einfluss der Torusgröße auf die Laufzeit betrachtet. Dabei wurden die Ergebnisse des einfachsten Falls (ein Torus mit der Größe 12x12, 8 SXCS Agenten, einem Zielobjekt mit einfacher Richtungsänderung und Geschwindigkeit 1) von den anderen Fällen (Sekunden und 144 Felder) abgezogen.

Wie in Abbildung 5.13 zu sehen, ist der Einfluss auf die Laufzeit im getesteten Bereich (16x16 bis 64x64) linear mit der Anzahl der Felder (mit Ausnahme des Ausreißers

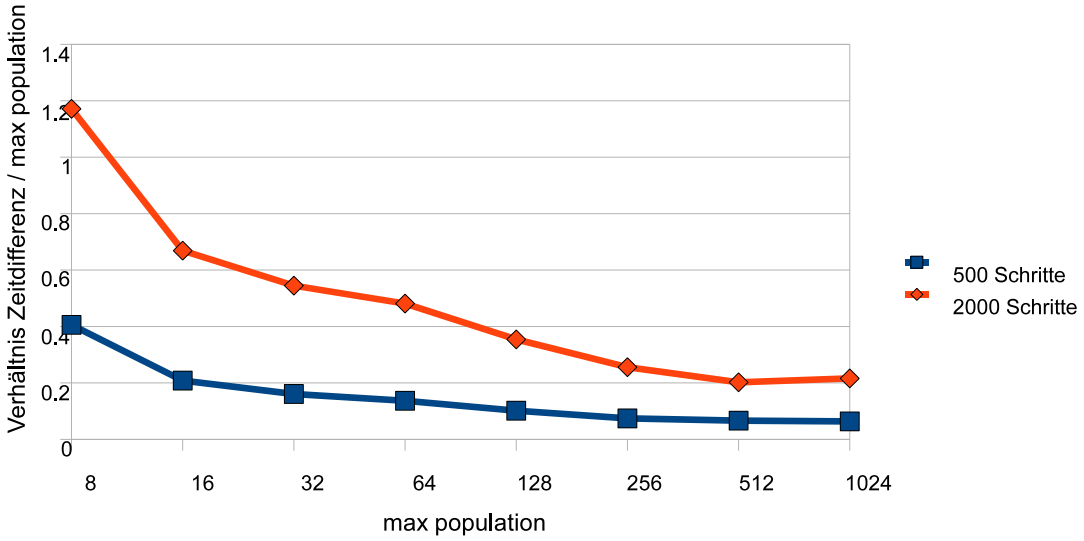

Abbildung 5.12: Darstellung der Auswirkung des Parameters *max population* N auf das Verhältnis der Laufzeit zu N (Säulenszenario, Zielobjekt mit einfacher Richtungsänderung, Geschwindigkeit 1, Agenten mit SXCS, *tournament selection*, $p = 0{,}84$)

bei 16x16) und ohne wesentliche Bedeutung.

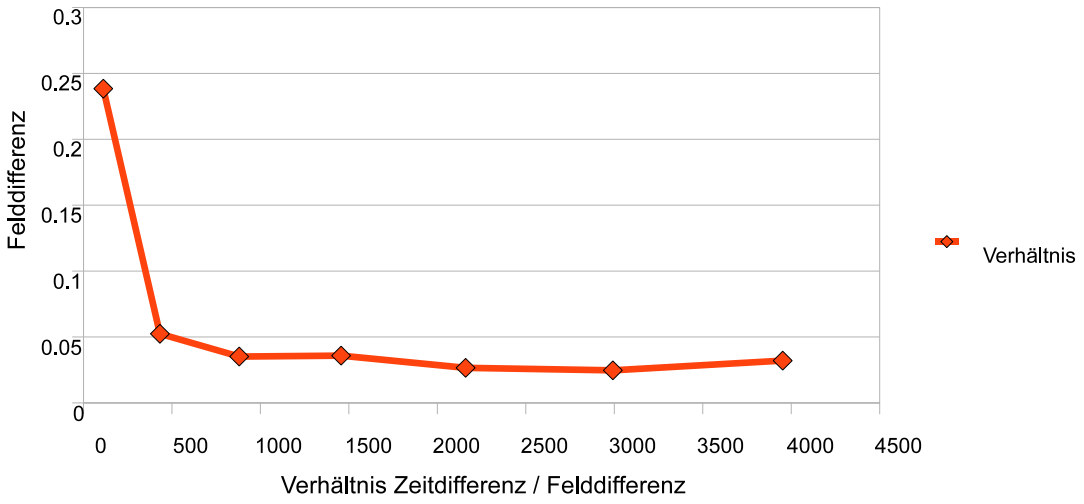

Abbildung 5.13: Darstellung der Auswirkung der Torusgröße auf die Laufzeit (Säulenszenario, Zielobjekt mit einfacher Richtungsänderung, Geschwindigkeit 1, SXCS Agenten, *tournament selection*, $p = 0{,}84$)

5.5.3 Zufällige Initialisierung der *classifier set* Liste

Normalerweise werden XCS Systeme mit leeren *classifier set* Listen initialisiert. Als Option wird jedoch auch eine zufällige Initialisierung erwähnt [But06b], bei der zu Beginn die *classifier set* Liste mit mehreren *classifier* mit zufälligen *action* Werten und *condition* Vektoren gefüllt wird. Dort wird aber auch angemerkt, dass sich beide Varianten in ihrer Qualität nur wenig unterscheiden. Da zum einen ein gewisser Zeitaufwand nötig ist, die Liste zu füllen und zum anderen nicht sichergestellt ist, dass die generierten *classifier* in dem jeweiligen Szenario überhaupt aktiviert werden können, scheint es sinnvoll zu sein, mit einer leeren *classifier set* Liste zu starten.

Dies bestätigen auch Tests, in denen man die Anzahl durch *covering* neu erstellter *classifier* mit zufällige Initialisierung der *classifier set* Liste (Abbildung 5.14) mit der Anzahl ohne Initialisierung (Abbildung 5.7) verglichen hat. Hier erkennt man, dass durchgehend etwas mehr *classifier* generiert werden. Dies sieht man auch beim direkten Vergleich der insgesamt erstellten *classifier* in Abbildung 5.15.

Von der Qualität her erreicht die Variante mit initialisierter Liste minimal schlechtere Werte. Die Begründung ist, dass in der Liste auch eine Reihe von *condition* Vektoren generiert werden, die nie auftreten. Diese müssen dann vom XCS erst erkannt und entfernt werden. Aus diesen Gründen werden alle Agenten mit leerer Liste starten.

5.5.4 Parameter *accuracy equality* ϵ_0

Der Parameter ϵ_0 gibt an, unter welchem *reward prediction error* Wert ein *classifier* als exakt gilt (und als *subsumer* auftreten kann, siehe Kapitel 3.2.2) und wie stark dieser Wert in die Berechnung der *fitness* einfließt.

In der Literatur [BW01] wird als Regel genannt, dass der Wert auf etwa 1% des Maximalwerts des *base reward* Werts (ρ) gesetzt werden soll. Dieser ist beliebig wählbar und hat lediglich ästhetische Auswirkungen. Somit wird dieser auf 1,0 und ϵ_0 auf 0,01 gesetzt.

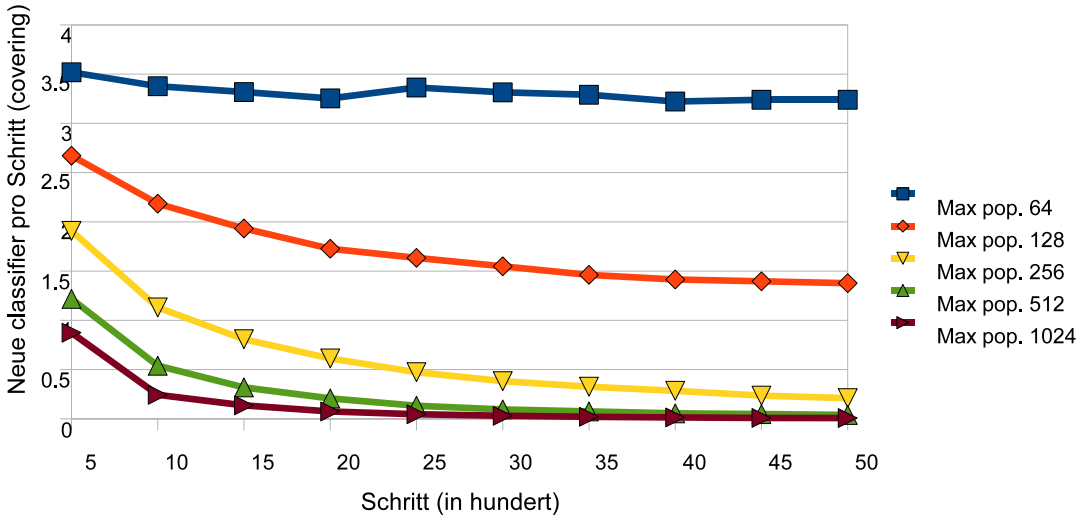

Abbildung 5.14: Auswirkung der maximalen Populationsgröße auf die Anzahl der *classifier*, die durch *covering* neu erstellt werden (Säulenszenario, Zielobjekt mit einfacher Richtungsänderung, Geschwindigkeit 1, Agenten mit SXCS, mit Initialisierung der *classifier set* Liste, *tournament selection*, $p = 0{,}84$))

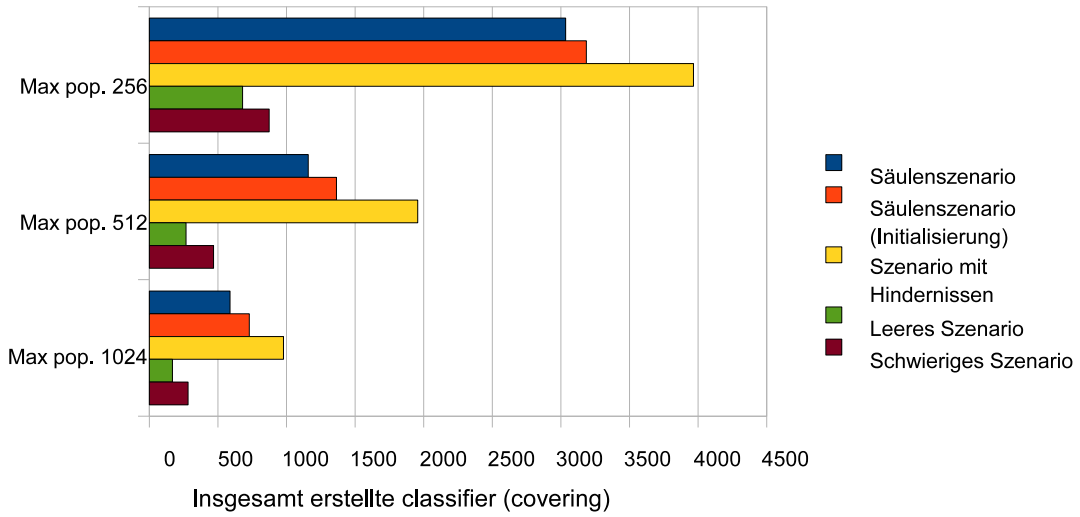

Abbildung 5.15: Auswirkung der maximalen Populationsgröße auf die Anzahl der *classifier*, die durch *covering* neu erstellt werden (Säulenszenario, Zielobjekt mit einfacher Richtungsänderung, Geschwindigkeit 1, Agenten mit SXCS, ohne Initialisierung der *classifier set* Liste, *tournament selection*, $p = 0{,}84$))

5.5.5 Parameter *GA threshold* θ_{GA}

Der *GA threshold* Parameter gibt an, ab wie groß der zeitliche Abstand zwischen zwei Aufrufen des genetischen Operators ist. Als Vergleichswerte werden hierbei zum einen der aktuelle Zeitpunkt und zum anderen der Durchschnittswert der Erstellungszeitpunkte der *classifier* aus dem jeweiligen *action set* benutzt. Die in der Einleitung erwähnte (siehe Kapitel 1.1.7), vergleichbare Arbeit [ITS05] benutzt einen deutlich größeren Wert für θ_{GA} (10.000) als andere Arbeiten aus der Literatur. Dies wird damit begründet, dass dort eine wesentlich größere Menge an Schritten zum Ziel benötigt als üblich. Allerdings liefen die dortigen Tests auch über eine wesentlich größere Zahl von Schritten (500.000).

Tests (siehe Abbildung 5.16) zeigten keinen Vorteil in diesem Zusammenhang, weshalb der Standardwert am oberen Ende des vorgeschlagenen Bereichs $\theta_{\mathrm{GA}} = 50{,}0$ benutzt wird. Zwar wurden mit $\theta_{\mathrm{GA}} = 25{,}0$ etwas bessere Werte erzielt, dies ist jedoch erst in der letzten Testetappe zum Vorschein gekommen und konnte aus Zeitgründen nicht in alle anderen Tests wieder integriert werden. Soweit nicht anders angegeben wird also der Wert 50,0 benutzt.

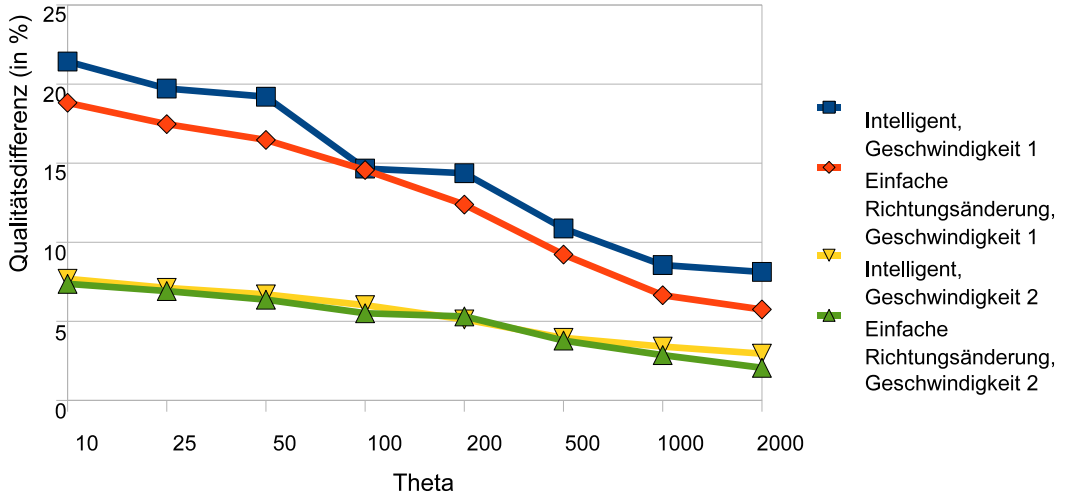

Abbildung 5.16: Vergleich verschiedener Werte für θ_{GA} (Säulenszenario, SXCS Agenten, *tournament selection*, $p = 0{,}84$)

5.5.6 Parameter *reward prediction discount* γ

In der Literatur in [BW01] wird ein Standardwert von $\gamma = 0{,}71$ genannt, es seien je nach Szenario aber auch größere und kleinere Werte möglich. Ein höherer Wert für γ bedeutet, dass die Höhe des Werts, der über *maxPrediction* weitergegeben wird, mit zeitlichem Abstand zur ursprünglichen Bewertung mit einem *base reward* Wert, weniger schnell abfällt, wodurch eine längere Verkettung von *base reward* Werten möglich ist. Umgekehrt führen zu hohe Werte für γ zu einer positiven Bewertung von *classifier* die am Erfolg womöglich gar nicht beteiligt waren, was sich negativ auf die Qualität auswirken kann, da diese etwas Falsches lernen.

Abbildung 5.17 zeigt einen Vergleich der Qualität bei unterschiedlichen Werten für γ beim XCS Algorithmus im Säulenszenario. „Intelligent“ und „Einfache Richtungsänderung“ beziehen sich jeweils auf die Eigenschaften des Zielobjekts. Ein konkretes Muster ist nicht zu erkennen, nur im Fall von $\gamma = 1{,}0$ gibt es einen deutlichen Abfall der Qualität. Das erscheint auch logisch, da dies bei XCS bedeuten würde, dass jede *action set* Liste alle *reward* Werten in voller Stärke sukzessive an alle weitergibt. Da sonst keine besonderen Fluktuationen zu beobachten sind, wird, wie vorgeschlagen, hier jeweils $\gamma = 0{,}71$ verwendet wird.

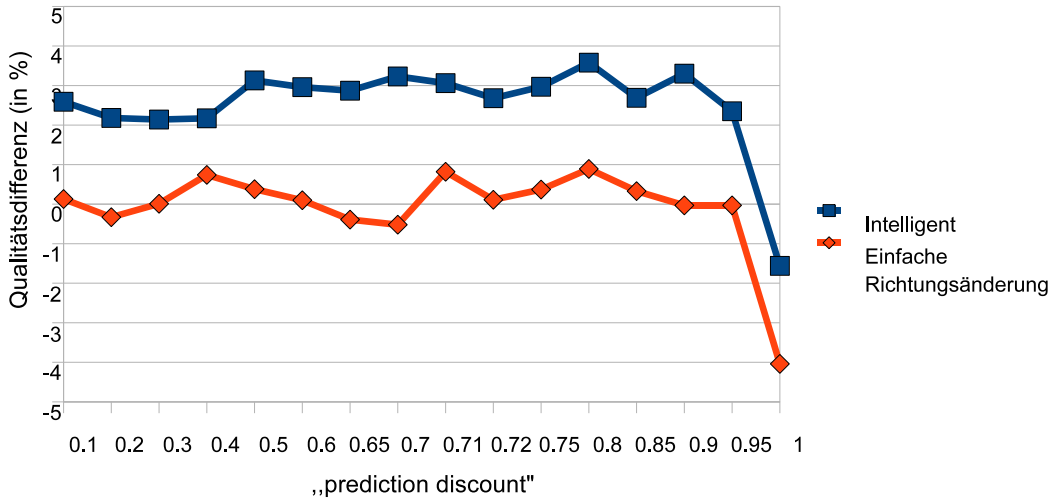

Abbildung 5.17: Auswirkung verschiedener *reward prediction discount* γ Werte auf die Qualität (Säulenszenario, Agenten mit XCS, *best selection*)

Die Beobachtung ist ein Hinweis darauf, dass die Weitergabe des *base reward* Werts grundsätzlich erfolgen sollte, jedoch sonst keinen Einfluss hat, also wahrscheinlich keine zusammenhängenden Ketten von Aktionen gebildet werden. Diese Überlegung unterstreicht auch die Überlegungen die zur SXCS Variante gemacht wurden (siehe Kapitel 4.3): Im Überwachungsszenario kann das Finden von „Wegen“ auf diese Weise (wie beim einführenden Beispiel zum *multi step* Verfahren in Kapitel 1.1.2) nicht zum Erfolg führen, da dem Szenario die Markow-Eigenschaft fehlt. Mit einer Erweiterung der Sensoren, prägnanteren Strukturen der Hindernisse und Aufenthaltswahrscheinlichkeiten des Zielobjekts auf dem Torus mit hoher Varianz würde dies vielleicht besser aussehen, der einzige Ansatz, dessen Erfolg in dieser Arbeit gezeigt wird, ist allerdings SXCS.

5.5.7 Parameter Lernrate β

Die Lernrate bestimmt u.a., wie stark ein ermittelter *reward* Wert den *reward prediction*, *reward prediction error* und *fitness* Wert bei jeder Aktualisierung beeinflusst.

In [ITS05] wurde ein unter dem Standardwert liegender Wert von $\beta = 0{,}02$ benutzt. Als Begründung wurde angegeben, dass dynamische Multiagentensysteme in vielerlei Hinsicht nur über Wahrscheinlichkeiten beschrieben werden können und somit der Lernvorgang eher vorsichtig vonstatten gehen sollte.

Vergleichende Tests (siehe Abbildung 5.18) zeigen für SXCS einen Optimalwert zwischen 0,01 und 0,1 an, größere Werte scheinen den Lernprozess deutlich zu schaden. Interessanterweise steigt für XCS bei großen Werten für β die Qualität deutlich an. Woher dieser Anstieg kommt, müsste näher untersucht werden. Da aber große Werte für β die Vergleichbarkeit in Frage stellen, soll sich hier auf niedrige Werte beschränkt werden.

Später in Kapitel 5.7.1 wird auch noch die Lernrate im schwierigen Szenario untersucht, bei dem sich höhere Lernraten als vorteilhaft herausstellen.

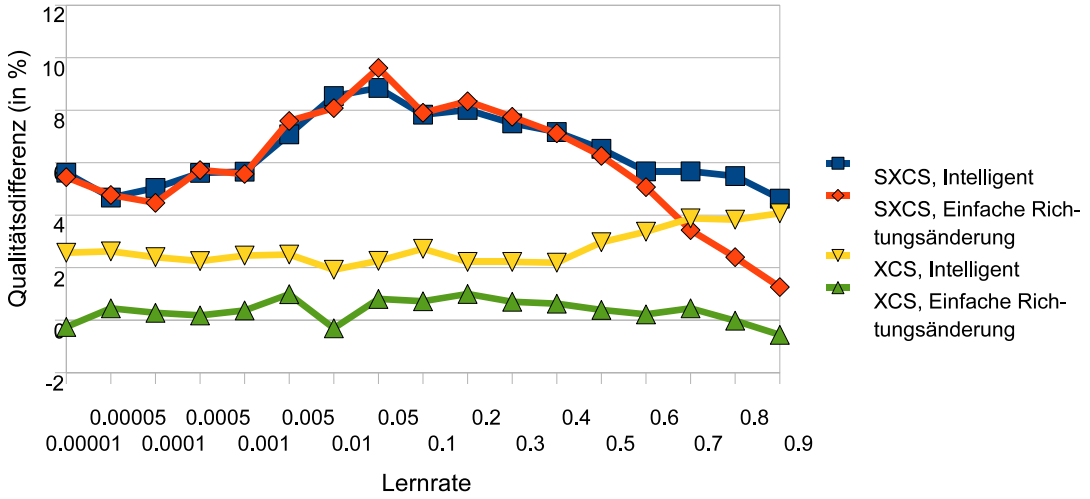

Abbildung 5.18: Auswirkung des Parameters Lernrate β auf die Qualität (Säulenszenario, Agenten mit XCS und SXCS Algorithmus, *best selection*)

5.5.8 Parameter *tournament factor p*

Beim Vergleich zwischen *best selection* (in einer andauernden *exploit* Phase) und einer abwechselnden *explore/exploit* Phase (mit *tournament selection* in der *explore* Phase und *best selection* in der *exploit* Phase), zeigt Abbildung 5.19 deutlich, dass im Säulenszenario die Auswahlart *best selection* (entspricht *tournament selection* mit $p = 1{,}0$) die bessere Wahl für SXCS ist.

Für das Szenario mit Hindernissen (mit $\lambda_p = 0{,}99$ und $\lambda_h = 0{,}2$) sieht die Sache jedoch anders aus. Hier spielen die Zahl der blockierten Bewegungen mit eine wesentliche Rolle. In den Tests stieg der Anteil der blockierten Bewegungen von etwa 40% (bei $p = 0{,}6$) auf bis über 70% ($p = 1{,}0$).

Ähnliches kann man in Abbildung 5.20 beim XCS Agenten feststellen. Hier erstreckt sich der Anteil der blockierten Bewegungen ebenfalls bis über 70% für $p = 1{,}0$, wenn auch für $p = 0{,}6$ der Wert etwas unter 30% lag. Hier ist möglicherweise eine explorativere Variante von Vorteil, dies wird in Kapitel 5.7 näher untersucht.

Für den *tournament factor* selbst konnte hier kein Optimalwert gefunden werden, je nach Szenario muss eine eigene Auswahlart gefunden werden. In einigen Szenarien wird

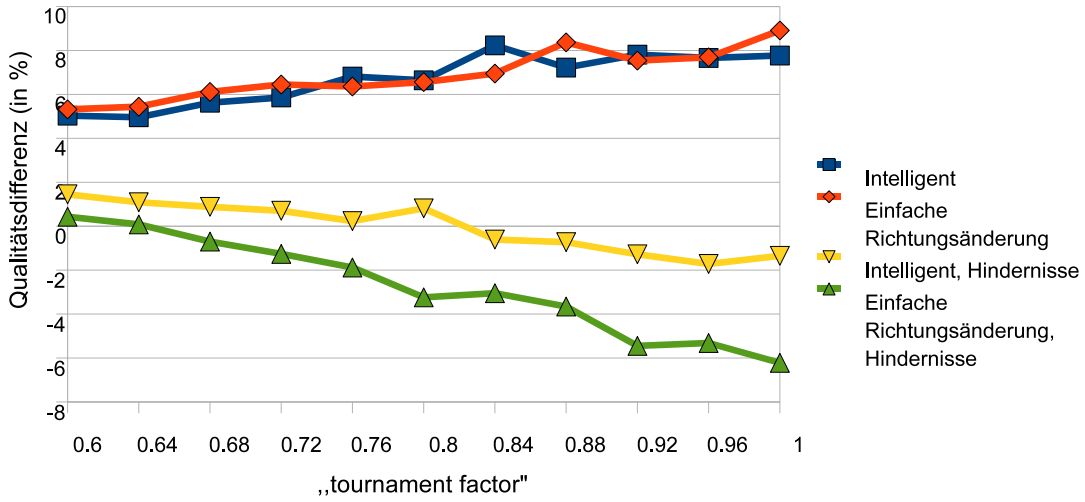

Abbildung 5.19: Vergleich verschiedener *tournament factor* Werte mit *best selection* (Säulenszenario und Szenario mit zufällig verteilten Hindernissen mit ($\lambda_p = 0{,}99$ und $\lambda_h = 0{,}2$), **SXCS Agenten**)

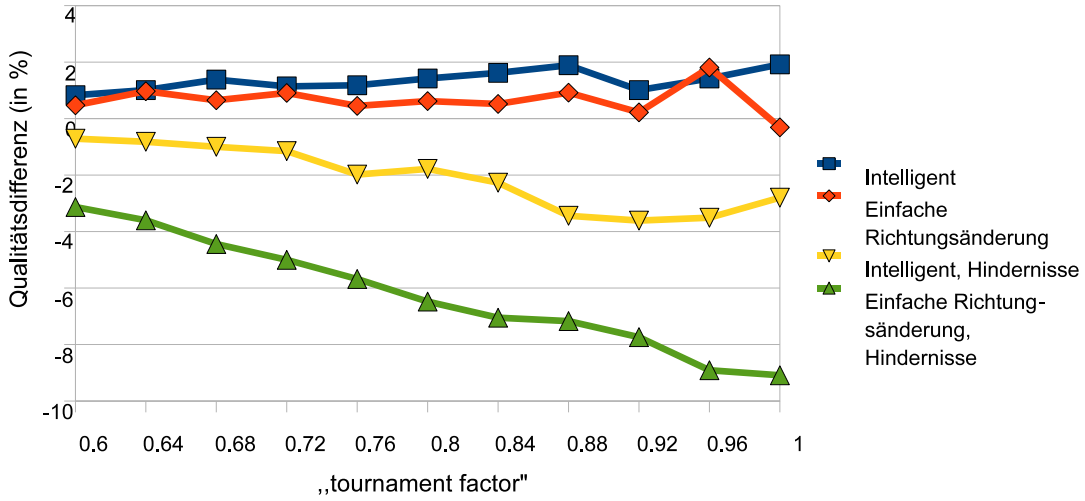

Abbildung 5.20: Vergleich verschiedener *tournament factor* Werte mit *best selection* (Säulenszenario und Szenario mit zufällig verteilten Hindernissen mit ($\lambda_p = 0{,}99$ und $\lambda_h = 0{,}2$), **XCS Agenten**)

ein Wert von $p = 0{,}84$ verwendet werden, welcher sich erfahrungsgemäß als brauchbar für Vergleiche herausgestellt hat, auch wenn er nicht optimal ist.

5.5.9 Übersicht über alle Parameterwerte

In Tabelle 5.7 findet sich die Aufstellung der wesentlichen Parameter für die Simulation. In der dritten Spalte sind jeweils die Standardwerte aus der Literatur [BW01] angegeben, nähere Erläuterungen finden sich in den jeweils aufgeführten Verweisen zu den einzelnen Kapiteln.

Tabelle 5.7: Verwendete Parameter (soweit nicht anders angegeben) und Standardparameter

Parameter	Wert	Standardwert [BW01]
max population N	**512** (siehe Kapitel 5.5.1)	[*je nach Szenario*]
max value ρ	**1,0** (siehe Kapitel 5.5.4)	[10.000]
fraction mean fitness δ	0,1	[0,1]
deletion threshold θ_{del}	20,0	[$\sim$ 20,0]
subsumption threshold θ_{sub}	20,0	[20,0+]
covering # probability $P_{\#}$	0,33	[$\sim$ 0,33]
GA threshold θ_{GA}	50 (siehe Kapitel 5.5.5)	[25-50]
mutation probability μ	0,05	[0,01-0,05]
prediction error reduction	0,25	[0,25]
fitness reduction	0,1	[0,1]
reward prediction init p_i	0,01	[$\sim$ 0]
prediction error init ϵ_i	0,0	[0,0]
fitness init F_i	0,01	[0,01]
condition vector	**leer** (siehe Kapitel 5.5.3)	[*zufällig oder leer*]
numerosity	1	[1]
experience	0	[0]
accuracy equality ϵ_0	**0,01** (siehe Kapitel 5.5.4)	[*1% des größten Werts*]
accuracy calculation α	0,1	[0,1]
accuracy power ν	5,0	[5,0]
reward prediction discount γ	0,71	[0,71]
learning rate β	**0,01 - 0,2** (siehe Kapitel 5.5.7)	[0,1-0,2]
exploration probability	0,5 (siehe Kapitel 3.2.2)	[$\sim$ 0,5]
tournament factor	0,84 (siehe Kapitel 5.5.8)	[-]

5.6 Unterschiedliche Geschwindigkeiten des Zielobjekts

In Kapitel 5.2 wurde dargestellt, dass bis zu einer Geschwindigkeit von 1 die auf Heuristiken basierenden Agenten das Zielobjekt lediglich andauernd verfolgt haben. Bei größeren Geschwindigkeiten wurde ein deutlicher Abfall der Qualität bemerkt, das Zielobjekt konnte den Agenten also öfters entkommen. Hier werden nun die gleichen Tests für lernende Agenten durchgeführt.

In Abbildung 5.21 ist der Vergleich zwischen XCS und SXCS bezüglich der Qualitätsdifferenzen bei verschiedenen Geschwindigkeiten des Zielobjekts dargestellt. Neben der von SXCS erreichten im Vergleich zu XCS deutlich höheren Qualitätsdifferenz ist hier wie bei den Heuristiken wieder ein Knick zu sehen. Bis zu einer Geschwindigkeit von etwa 0,7 bleibt die Qualitätsifferenz ungefähr auf einem Niveau um dann abzufallen. Dies zeigt an, dass ein gewisser Teil der erreichten Qualität durch eine Strategie der Verfolgung erreicht wurde.

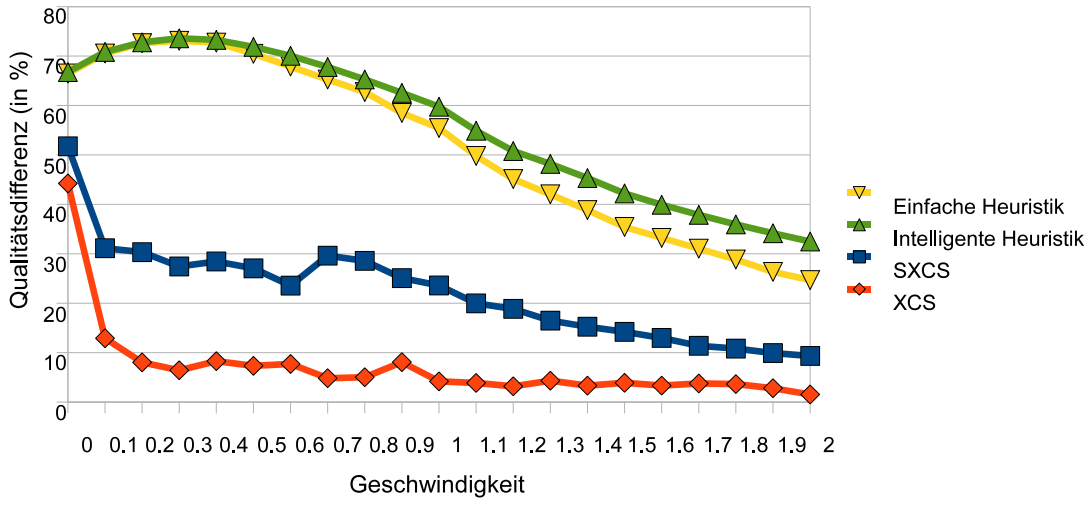

Abbildung 5.21: Vergleich der Qualitäten von XCS und SXCS bezüglich der Geschwindigkeit des Zielobjekts (intelligentes Zielobjekt, *best selection*)

5.7 Tests im Szenario mit zufällig verteilten Hindernissen

Im vorherigen Abschnitt wurde festgestellt, dass Szenarien mit zufällig verteilten Hindernissen (mit $\lambda_h > 0$) für XCS und SXCS eine Herausforderung darstellt. Einfache Tests mit andauernder *exploit* Phase lässt beide Varianten an dem sehr hohen Anteil an blockierten Bewegungen scheitern. Neben einer Erweiterung der Sensorfähigkeiten und einer Anpassung der *reward* Funktion scheint hier nur ein Wechsel der Auswahlart zur *roulette wheel selection* bzw. der *tournament selection* mit niedrigerem Wert für p Abhilfe zu schaffen. Eine Erhöhung der maximalen Populationsgröße bringt keine Verbesserung, da die Zahl der neu erstellten *classifier* in einem solchen Szenario nicht viel höher ist als im Säulenszenario (siehe Kapitel 5.5.1).

Ein Testlauf mit *roulette wheel selection* erbringt für obigen Fall mit Hindernissen und intelligentem Zielobjekt einen ähnlich hohen Wert für die Qualitätsdifferenz (etwa 2,0%). Eine Verringerung des *tournament factor* Werts mit obiger Konfiguration führt also zu einer Annäherung der Auswahlart *tournament selection* an die Auswahlart *roulette wheel selection*.

Desweiteren ist aufgrund der großen Anzahl blockierter Sichtlinien davon auszugehen, dass die Agenten relativ häufig keine anderen Agenten in Sicht bekommen, ein *base reward* Wert von 0 wird also eher die Seltenheit als die Regel. Zusätzlich verringert sich dadurch (aus der lokalen Sicht eines Agenten gesehen) die Dynamik des Systems, was die Wiederholung gleicher Aktionen weiter fördert. Um dem entgegenzuwirken, wird hier die in Kapitel 3.6 erwähnte Auswahlart mit abwechselnder *explore/exploit* Phase ausprobiert.

Setzt man in der *explore* Phase die Auswahlart *roulette wheel selection* und in der *exploit* Phase die Auswahlart *tournament selection* führt dies zu den Ergebnissen in Abbildung 5.22. XCS erreicht auch mit dieser Auswahlart keinen wesentlichen Vorteil gegenüber dem Algorithmus mit zufälliger Bewegung, während beim SXCS Algorithmus speziell bei $p = 1{,}0$, also beim Äquivalent zur Auswahlart mit abwechselnder *roulette selection* und *best selection*, ein Ausschlag beim sich intelligent verhaltenden Zielobjekt zu sehen ist. Das Zielobjekt mit einfacher Richtungsänderung bleibt aber auch hier eine zu schwierige Hürde.

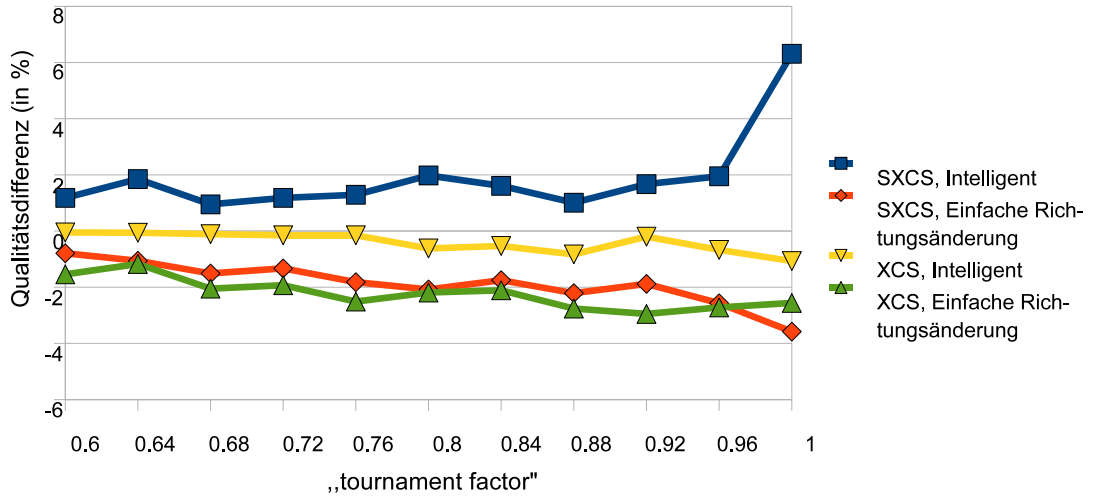

Abbildung 5.22: Vergleich verschiedener *tournament factor* Werte bei abwechselnder *explore/exploit* Phase (Szenario mit zufällig verteilten Hindernissen mit ($\lambda_p = 0{,}99$ und $\lambda_h = 0{,}2$, SXCS Agenten)

Von den blockierten Bewegungen her konnte mit dieser Auswahlart einerseits die Zahl der blockierten Bewegungen reduziert (auf etwa 34%) bei XCS und andererseits weiterhin die Möglichkeit offen gelassen, das Ziel zu verfolgen, wenn es in Sicht ist.

Insgesamt ist also zusammenzufassen, dass für ein Szenario mit wenigen Hindernissen die Auswahlart *best selection* die beste Wahl ist und in Szenarien mit vielen Hindernissen ein Wechsel der *roulette wheel selection* und *best selection* Auswahlart die beste Wahl ist.

Desweiteren war zu sehen, dass hier eindeutig das Szenario mit einem Zielobjekt mit einfacher Richtungsänderung zu schwierig für die betrachteten Agenten. Zwar erreichen die Agenten höhere Werte als im Fall mit intelligenter Heuristik, jedoch ist auch die Qualität des zufälligen Algorithmus deutlich höher. Insbesondere besitzen beide Zielobjekttypen die Fähigkeit, Hindernisse zu erkennen, d.h. sie bleiben deutlich weniger oft stehen und können somit schneller aus der Reichweite der Agenten fliehen.

Würde man, wie im Ausblick in Kapitel 6.1.1 erwähnt, die Sensorfähigkeiten der Agenten und die *reward* Funktion auf Hindernisse erweitern, würden sich in diesem Szenario womöglich Verbesserungen ergeben. Dann wäre es auch möglich, im Szenario mit vielen

Hindernissen die Auswahlart *best selection* zu verwenden. Hierzu müssten z.B. Strafen für Aktionen verteilt werden. Da die *reward* Funktion eigentlich nur Situationen und nicht Aktionen bewertet, ist eine Analyse der bisher gespeicherten *action set* Listen nötig. Zeigt der in einer *action set* Liste gespeicherte Sensorstatus ein Hindernis in unmittelbarer Nähe an und zeigt die gespeicherte Aktion in die Richtung des Hindernis, könnte dies mit einem *base reward* Wert von 0 bewertet werden. Wie bei jeder zusätzlichen Heuristik muss man sich aber die Frage stellen, wie allgemeingültig Agenten mit solchen Modifikationen dann noch agieren können und ob dadurch nicht optimale Lösungen wegfallen.

5.7.1 XCS, SXCS und DSXCS im schwierigen Szenario

Im schwierigen Szenario wurde in Kapitel 5.3 gezeigt, dass hier sich zufällige bewegende Agenten wie auch Agenten mit einfacher Heuristik versagen. Auch wurde argumentiert, dass Agenten mit intelligenter Heuristik nur deshalb Erfolg haben, weil sie sich gegenseitig durch die Öffnungen „drängen“. Hier werden nun lernende Agenten ihre Fähigkeiten unter Beweis stellen. Der wesentliche Vorteil der lernenden Agenten in diesem Szenario ist, dass sie ihr Gelerntes über die, wie bisher, 10 Probleminstanzen behalten können und somit direkt auf den letzten Abschnitt durch die Öffnungen laufen können, sofern sie das Richtige gelernt haben.

Auch wird sich hier wieder das Zielobjekt nur in einer Linie bewegen (siehe Kapitel 2.5.5), es ist also im Grunde kein Überwachungsszenario im eigentlichen Sinne. Wenn ein Agent den letzten Abschnitt auf der rechten Seite erreicht, ist das Problem im Grunde schon gelöst. Primär soll, im Vergleich z.B. zum Säulenszenario, auch gezeigt werden, dass SXCS bei einer solchen Problemstellung im Vergleich zu XCS nicht versagt, also durch die Abänderungen nicht die Eigenschaften verliert, die XCS in einfacheren, statischen Problemen zeigt.

Zuerst wird mit der einfachen Auswahlart *roulette wheel selection* die Lernrate in Abbildung 5.23 getestet. Hier sieht man mehrere Eigenschaften:

- SXCS und DSXCS haben in etwa denselben Verlauf.
- XCS liegt unter der Qualität von SXCS.

- XCS erreicht etwas stabilere Ergebnisse, sichtbar an der glatteren Kurve.
- Für SXCS und DSXCS ändert sich im betrachteten Bereich für etwa $\beta > 0{,}2$ nichts mehr, für XCS für etwa $\beta > 0{,}7$.

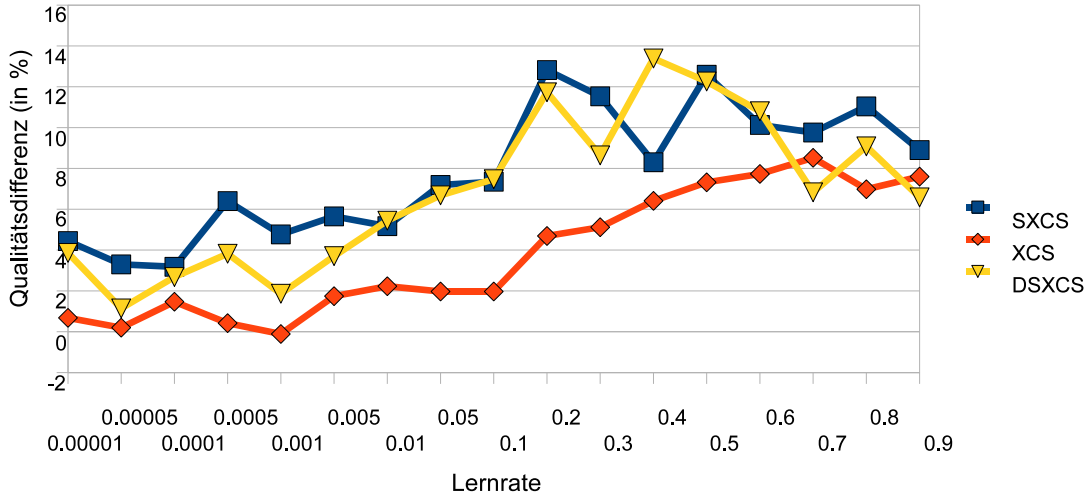

Abbildung 5.23: Auswirkung der Lernrate auf die Qualität (schwieriges Szenario, Agenten mit XCS, SXCS und DSXCS, *roulette wheel selection*)

Zwar erreicht XCS ähnliche Ergebnisse wie SXCS, allerdings nur durch über eine hohe Lernrate. Wie in Kapitel 5.5.7 gesehen, führt dies aber zu deutlich schlechteren Ergebnissen in anderen Szenarien, weshalb insgesamt gesagt werden muss, dass für die benutzte Auswahlart XCS auch hier unterliegt. Insgesamt erscheint 0,2 als passender Wert für die Lernrate, auch im Hinblick auf die Standardwerte in der Literatur (siehe Kapitel 5.5.9) und bezüglich der Vergleichbarkeit mit Ergebnissen in anderen Szenarien.

5.7.2 SXCS mit intelligenter Heuristik im schwierigen Szenario

Betrachtet man die Ergebnisse der Heuristiken im schwierigen Szenario (siehe Kapitel 5.3), stellt man fest, dass diese etwas niedriger sind. SXCS hat im betrachteten Szenario in Kapitel 5.7.1 etwa 29,24%, während die intelligente Heuristik dort einen Wert von etwa 40,63% erreicht. Im Folgenden wird nun geprüft werden, ob anhand der besprochenen Methoden eine Verbesserung erzielt werden kann.

Anstatt *roulette wheel selection* soll nun verstärkt die Verfahren der *exploit* Phase angewendet werden. Hierzu wurden eine Reihe von Experimenten durchgeführt, die erfolgreichsten Ergebnisse erbrachte zuerst eine abwechselnde *explore/exploit* Phase mit *roulette wheel selection* und *best selection* mit 36,22% für SXCS (mit $\beta = 0{,}2$), während sich der XCS Wert nur minimal verbesserte.

Noch weiter konnte die Qualität durch die Verwendung einer abwechselnden *explore/exploit* Phase (mit *tournament selection* in der *explore* Phase und *best selection* in der *exploit* Phase) gesteigert werden (siehe Abbildung 5.24, die Tests für SXCS wurden hier zur Sicherheit über 20 Experimente durchgeführt). Der hier ermittelte Optimalwert für $p = 0{,}84$ liegt nun mit 43,75% (bei $\beta = 0{,}2$) deutlich über dem Wert der Heuristik. Damit ist gezeigt, dass die Quelle der Steigerung aus zusätzlicher Erlernung der Hindernisse rührt, eine Eigenschaft, die der intelligenten Heuristik fehlt.

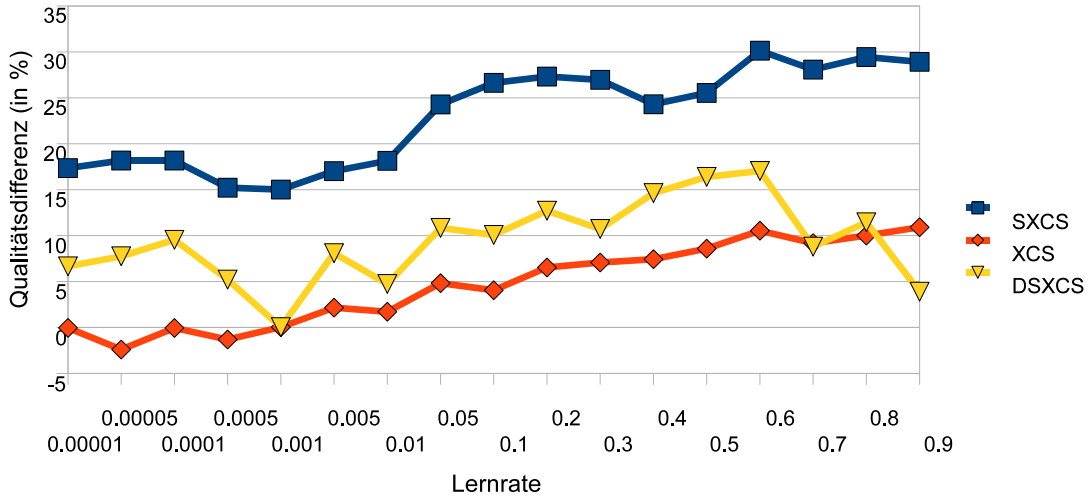

Abbildung 5.24: Auswirkung der Lernrate auf die Qualität (schwieriges Szenario, Agenten mit XCS und SXCS, mit abwechselnder *explore/exploit* Phase)

Als letztes wird eine Variation der Anzahl der Probleminstanzen in Verbindung mit dem schwierigen Szenario betrachtet. Es wird Testlauf durchgeführt, bei dem insgesamt jeweils genau 8.000 Schritte durchgeführt werden. Die 8.000 Schritte werden allerdings jeweils auf eine unterschiedliche Anzahl von Problemeninstanzen verteilt, d.h. es wird eine Probleminstanz mit 8.000 Schritten, 2 Probleminstanzen mit jeweils 4.000 Schritten, 4

Probleminstanzen mit jeweils 2.000 Schritten usw. getestet. Neben den oben ermittelten Werten für die Lernrate $\beta = 0{,}2$ und dem Standardwert *tournament factor* $p = 0{,}84$ ist außerdem (wie in Kapitel 4.3.3 untersucht) die Größe des Stacks von 32 statt 8 benutzt worden, was zu einer weiteren Erhöhung der Qualität führte.

In Abbildung 5.25 ist ein Vergleich mit der intelligenten Heuristik dargestellt. Hier sieht man, dass ab 8 Probleminstanzen mit jeweils 500 Schritten der SXCS Algorithmus etwas besser abschneidet als der intelligente Algorithmus. SXCS kann die geringere Schrittzahl durch eine erhöhte Anzahl von Probleminstanzen also durch Erlernen eines Weges durch das Szenario kompensieren. Auch sieht man, dass SXCS bei nur einer Problemzahl eine deutlich niedrigere Qualität erreicht. Dies rührt daher, dass der Weg zum Ziel bis dahin noch nicht gefunden wurde und erst gelernt werden muss, dann aber bis 8 Probleminstanzen mit der intelligenten Heuristik mithält und dann aufgrund der niedrigen Zahl von Schritten abfällt. Dagegen bedarf XCS ganzer 8 Probleminstanzen um das Maximum zu erreichen, die Lerngeschwindigkeit ist also deutlich geringer.

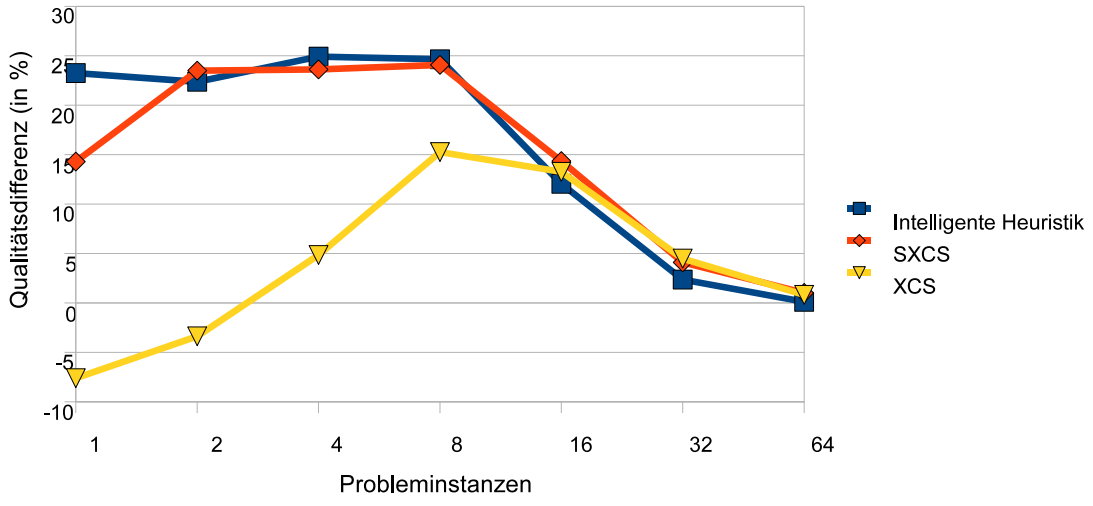

Abbildung 5.25: Qualität bei unterschiedlicher Anzahl von Problemen bei gleichbleibender Gesamtzeit (schwieriges Szenario)

5.8 Vergleich SXCS mit DSXCS

Wie in Tabelle 5.8 zu sehen, erreicht DSXCS im Säulenszenario mit und ohne Kommunikation nicht die Qualität von SXCS. Anzumerken ist allerdings, dass die egoistische Kommunikationsgruppe einen etwas höheren Wert als die einzelne Kommunikationsgruppe erreicht, was vielleicht bedeutet, dass die Annahme, dass die indiskriminierende Verteilung des *reward* Werts an alle Agenten nachteilig sein könnte. Desweiteren fällt deutlich der Unterschied in der Varianz der Punkte auf, d.h. im Gegensatz zu SXCS ähnelt sich der Anteil, den jeder Agent am Gesamtergebnis beiträgt, bei den Varianten mit Kommunikation stärker. Dies wiederum könnte auf eine Form kollaborativer Zusammenarbeit deuten. Der geringe Anteil der Varianz von XCS lässt sich durch das niedrige Ergebnis erklären.

Tabelle 5.8: Vergleich von SXCS mit den DSXCS Varianten (Säulenszenario, *best selection*)

Algorithmus	Varianz Punkte	Abdeckung	Qualität
XCS	53,96	69,95%	12,41%
SXCS	78,51	70,50%	19,03%
DSXCS (ohne Kommunikation)	72,85	70,33%	16,96%
Einzelne Kommunikationsgruppe	49,73	68,45%	14,91%
Egoistische Kommunikationsgruppe	47,70	68,39%	15,30%

Eine Betrachtung des gleitenden Durchschnitts führt auch zu keinen neuen Erkenntnissen (siehe Abbildung 5.26). Die zwei Varianten mit Kommunikation haben einen sehr ähnlichen Verlauf, DSXCS liegt etwas unter SXCS und SXCS erreicht zu jedem Zeitpunkt den höchsten Wert. Anzumerken sei hier nur die Lernkurve der dargestellten Algorithmen.

5.9 Bewertung Kommunikation

Bei keiner der zwei in Kapitel 4.5.3 und Kapitel 4.5.4 vorgestellten Kommunikationsarten konnten in den betrachteten Szenarios Vorteile, was die Qualität betrifft, gegenüber dem SXCS Algorithmus ohne Kommunikation festgestellt werden. Einzig eine etwas höhere Qualität der einzelnen Kommunikationsgruppe gegenüber der DSXCS Variante ohne Kommunikation konnte festgestellt werden.

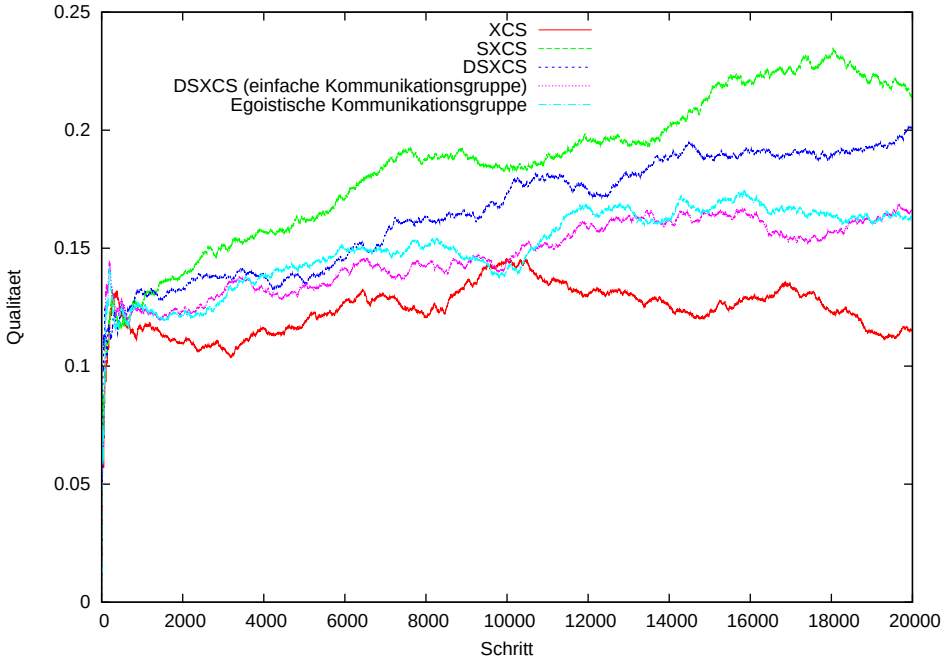

Abbildung 5.26: Vergleich des Verlaufs des gleitenden Durchschnitts der Qualität von SXCS und DSXCS (Säulenszenario, *best selection*)

Kommunikation war allerdings nicht der Hauptschwerpunkt dieser Arbeit, die Erweiterung bot sich primär an, da sie einfach zu implementieren war. Es ist vor allem in der Theorie mehr Aufwand nötig um Kommunikation in diesem Zusammenhang zu bewerten. Hinsichtlich der Anzahl möglicher Erweiterungen, die im Ausblick in Kapitel 6.1 aufgelistet sind, erscheint es sinnvoll, den Punkt der Kommunikation in Verbindung mit auf SXCS basierenden Agenten erst wieder aufzugreifen, nachdem diese Punkte untersucht wurden.

Die Untersuchung in Kapitel 5.8 konnte nicht klären, wo das Problem bei dieser Umsetzung von Kommunikation lag. Aufgrund zeitlicher Begrenzung der Arbeit sind die Untersuchungen aber noch nicht abgeschlossen gewesen. Es können deshalb nur eine Reihe Vermutungen angegeben werden:

- Die Weitergabe des *reward* Werts basierte auf der Idee, nicht beteiligte Agenten an einem positiven *base reward* zu beteiligen, da sie trotzdem ein Gebiet überwachen.
- Die Implementierung über den separaten Faktor war nicht zielführend. Möglicher-

weise muss eine bessere Funktion bei der Auswertung der gespeicherten *action set* Listen gefunden werden (siehe Kapitel 4.4).

- Die Implementierung sorgte bei der Kombination der *reward* Werte bei der egoistischen Kommunikationsgruppe für einen unstetige Verlauf der neuen Verteilung des *reward* Werts unter den *action set* Listen (siehe letzter Graph in Abbildung 4.7).
- Die Erwartungen (in Kapitel 4.5.4 an die Agenten, dass sie Gruppen auf Basis des Aktualisierungsfaktors bilden, war zu hoch gegriffen. Möglicherweise muss der Ansatz der Übermittlung der *reward* Werte mittels Aktualisierungsfaktors überdacht werden.
- Die indiskriminierende Verteilung des *reward* Werts an alle Agenten sorgte bei der einzelnen Kommunikationsgruppe (siehe Kapitel 4.5.3) für die Bewertung von Agenten, die nicht zur der Lösung des Problems beigetragen haben.
- Der relativ geringe Wert für die Größe des Stacks bei SXCS könnte ein Indiz darauf sein, dass nur relativ kurze Wege gelernt werden müssen, während bei der Kommunikation das Augenmerk mehr auf Belohnung längerfristiger Positionierung lag.

5.10 Zusammenfassung der Ergebnisse

In den Tests erreichten die in Kapitel 5.1 getesteten Algorithmen mit einfacher und mit intelligenter Heuristik deutlich bessere Werte als die betrachteten XCS Varianten. Dies liegt erst einmal daran, dass das XCS andauernd versucht, mittels genetischer Operatoren neue *classifier* zu generieren, was die Auswahl der Regeln stören kann. Auch versuchen die lernenden Agenten andauernd, Korrelationen zu finden, welche nicht unbedingt vorhanden sind. Tritt beispielsweise ein Hindernis gleichzeitig mit dem Zielobjekt auf, so wird das Hindernis (im übertragenen Sinn) als „gut“ empfunden, obwohl das Zielobjekt selbst die Hindernisse überhaupt nicht in die Entscheidung miteinbezieht, in welche Richtung es laufen soll.

Schafft man es den in dieser Arbeit vorgestellten SXCS Algorithmus weiter zu verbessern, ist es durchaus denkbar, dass ähnliche Qualitäten erreicht werden können. Insbesondere die im Ausblick in Kapitel 6.1 diskutierten Erweiterungen bieten sich dazu an.

Außerdem wurde beim Test in Szenarien mit vielen Hindernissen in Kapitel 5.7 festgestellt, dass ein hoher Anteil von *exploit* Phasen (bzw. ein großer Wert für den *tournament factor p*) dazu führen kann, dass einzelne Agenten andauernd in Richtung eines Hindernis laufen. Da dies u.U. zu über 70% blockierten Bewegungen (und entsprechend niedriger Qualität) führt, haben die Agenten es also offensichtlich nicht geschafft, Sensorinformationen über Hindernisse sinnvoll zu verarbeiten und zu erlernen.

Ausnahme hierfür bieten die Ergebnisse aus dem schwierigen Szenario, der Weg zum Ziel war hier wesentlicher Bestandteil der Lösung, die Öffnungen wurden erkannt und ausgenutzt.

Als Gründe kann man mehrere Punkte anbringen. Sensorinformationen alleine können es nicht sein, denn die betrachteten Heuristiken schaffen es schließlich völlig ohne Information über die Hindernisse. Aber hier liegt auch schon das, für menschliche Beobachter auf den ersten Blick vielleicht etwas seltsam erscheinende, Problem. Ein wesentlicher Punkt ist, dass die Agenten lernen müssen, Hindernisse zu ignorieren. Beim gleichzeitigen Auftreten von Hindernissen und dem Zielobjekt ziehen sie u.U. fehlerhafte Schlüsse, da sich das Zielobjekt (fast) unabhängig von den Hindernissen bewegt. Würde sich das Zielobjekt beispielsweise andauernd um eine ganz bestimmte Hinderniskonfiguration herumbewegen, hätten auf XCS basierende Agenten wohl einen gewissen Vorteil.

Kapitel 6

Zusammenfassung, Ergebnis und Ausblick

People do not like to think. If one thinks, one must reach conclusions. Conclusions are not always pleasant.
Helen Keller

Wesentliche Erkenntnisse aus dieser Arbeit sind:

- Die Bewertungsfunktion kann anhand einer Nachbildung einer gut funktionierenden Heuristik konstruiert werden (siehe Kapitel 3.4.3).
- Durch Hinzufügen einer Form von Speicher (siehe Kapitel 4.3) kann SXCS die betrachtete Aufgabe wesentlich besser als XCS lösen (siehe Kapitel 5).
- Eine Variation der Lernrate kann je nach Szenario sinnvoll sein (siehe Kapitel 5.5.7 und Kapitel 5.7.1).
- Die Agenten mit XCS und SXCS haben deutliche Probleme mit Szenarien mit vielen Hindernissen (siehe Kapitel 5.5.8).
- Ein dynamischer Wechsel der Auswahlart für Aktionen während eines Laufs kann sinnvoll sein, um die Zahl der blockierten Bewegungen zu verringern und das Zielobjekt besser verfolgen zu können (siehe Kapitel 5.7).
- Sowohl XCS als auch SXCS können (für sich zufällig bewegende Agenten) schwierige Szenarien (siehe Kapitel 2.2.4) mit markanten Hindernispunkten meistern (siehe Kapitel 5.7.1) und SXCS kann diese sogar besser als die intelligente Heuristik lösen (siehe Kapitel 5.7.2).

- Die vorgestellte Variante des verzögerten SXCS Algorithmus DSXCS bietet Raum für Verbesserung (siehe Kapitel 4.4).

- Die versuchte Implementierung von Kommunikation führte nicht zum Erfolg, zum einen wegen geringer Möglichkeiten zur Kooperation, zum anderen wegen zu einfach umgesetztem Algorithmus (siehe Kapitel 5.8 und 5.9).

Das wesentliche Ergebnis ist, dass die Implementierung des XCS auf Überwachungsszenarien ausgeweitet werden kann, ohne wesentliche Veränderungen am Algorithmus vorzunehmen. Die alleinige Anpassung des XCS *multi step* Verfahrens, dass eine neue Probleminstanz gestartet wird, wann immer sich das Ziel in Überwachungsreichweite befand, führte aber nicht zum Erfolg. Die Ergebnisse waren kaum besser als ein sich zufällig bewegender Agent.

Erst eine Untersuchung der Natur von Überwachungsszenarien, insbesondere die Feststellung, dass bei ihnen nicht die Markow-Eigenschaft gilt und somit ein Speicher sinnvoll sein könnte, führte weiter zur Entwicklung des SXCS Algorithmus.

Zuvor musste jedoch erst die Bewertungsfunktion analysiert werden. Dazu wurden die aus XCS bekannten *single step* und *multi step* Verfahren näher betrachtet und drei einfache, im Überwachungsszenario relativ erfolgreiche, Heuristiken untersucht. Dadurch konnte eine Bewertungsfunktion gewonnen werden, die abhängig von den Sensordaten einen positiven *base reward* vergab, wenn das Zielobjekt oder keine Agenten in Sicht waren.

Zusammen mit der betrachteten Implementierung von XCS im Überwachungsszenario konnte mit der Bewertungsfunktion die funktionsfähige Variante SXCS implementiert und getestet werden. Ergebnis hier war, dass SXCS in den betrachteten Szenarien XCS deutlich überlegen ist.

Ausnahme bildete das Szenario mit Hindernissen, hier musste aufgrund zahlreicher blockierter Bewegungen die Auswahlart näher untersucht werden. Resultat der Überlegungen war dann ein dynamischer Wechsel der Auswahlart für die Aktionen, mit dem bei einer Mischung der *roulette wheel selection* und *best selection* etwas bessere Ergebnisse erzielt werden konnten.

Eine Untersuchung des schwierigen Szenarios ergab, dass SXCS auch hier gegenüber XCS deutlich bessere Ergebnisse erzielt. Insbesondere wurde auch die intelligente Heuristik deutlich geschlagen, was beweist, dass Hindernisstrukturen bei den durchgeführten Tests von den SXCS Agenten gelernt wurden.

Schließlich wurde intensiv ein Ansatz für Kommunikation diskutiert. Hauptidee war, dass Agenten ihre Aktionen nicht negativ bewerten sollten, wenn andere Agenten das Ziel in Sicht haben. Die tatsächliche Umsetzung konnte in Tests allerdings nicht überzeugen. Hier ist noch viel Raum für weitere Untersuchungen gegeben.

6.1 Ausblick

Mit dieser Arbeit wurde ein neues Gebiet von Problemfeldern in Verbindung mit dem XCS Algorithmus eröffnet. Dadurch bieten sich zahlreiche mögliche Fortsetzungen der Untersuchung, welche nachstehend kurz diskutiert werden.

6.1.1 Verbesserung der Sensoren

In dieser Arbeit wurde ein einfaches Sensorenmodell verwendet, das zwar höhere Sichtweiten ermöglicht, dafür aber sehr ungenaue Informationen liefert. Zu einem wesentlich besseren Ergebnis könnte die Verwendung von einer größeren Anzahl von Sensoren bzw. rationale Eingabewerten führen. Dies würde die Möglichkeit eröffnen, den Abstand zu anderen Agenten je nach Szenario genauer zu regeln. Eine einführende Arbeit bezüglich rationalen Eingabewerten und XCS findet sich z.B. in [Wil00]. Alternativ böte sich an, einfach weitere Binärsensoren einzugliedern um so mehrwertige Sensoren zu erhalten.

Mit dem Erweitern der Sensoren würde sich auch die Tür für eine Optimierung der *reward* Funktion für den *base reward* Wert öffnen (siehe Kapitel 3.4), wodurch sich auch der Lernerfolg für das globale Problem verbessern könnte.

Desweiteren könnten Szenarien untersucht werden, bei denen es, durch die besondere Positionierung von Hindernissen, markante Stellen auf dem Feld gibt, an denen sich die

Agenten orientieren können. Beispielsweise sind dann auch komplexere Bewegungen des Zielobjekts denkbar (z.B. Pendel- oder Kreisbewegungen), welche die Agenten durch kluge Positionierung an markanten Stellen erlernen könnten.

6.1.2 Verwendung einer mehrwertigen *reward* Funktion

Es ist aus den Analysen in Kapitel 5.1 bekannt, dass ein Agent mit intelligenter Heuristik in den betrachteten Szenarien sehr gut abschneidet; die *reward* Funktion wurde allerdings von der einfachen Heuristik übernommen. Der Grund dafür wurde in Kapitel 3.4 ausführlich diskutiert, zur Darstellung bedarf es einer mehrwertigen *reward* Funktion. Eine Erweiterung des Algorithmus in diese Richtung erscheint deshalb sinnvoll.

6.1.3 Untersuchung der Theorie

Genauer untersucht werden muss die mathematische Grundlage des verwendeten Ansatzes vom in Kapitel 4.3 besprochenen XCS Variante SXCS. Zwar wurden in dieser Arbeit einige Eigenschaften untersucht und festgestellt, jedoch fehlt die theoretische Begründung, weshalb diese Form der Verteilung des *reward* Werts auf *action set* Listen in zeitlichem Zusammenhang in diesen Szenarien deutlich besser abschneidet als die von XCS. Womöglich ist hierzu eine Untersuchung einzelner Agenten in einem einfacheren Szenario zielführend. Insbesondere würde ein einfacheres Szenario auch die benötigte Rechenzeit verkürzen und somit die Untersuchung wesentlich verkürzen.

6.1.4 Untersuchung der *classifier*

Eine tiefergehende Analyse der *classifier set* Listen selbst könnte ebenfalls neue Erkenntnisse liefern. Hier könnte man feststellen, was für Strategien die Agenten tatsächlich lernen. Zwar wird vom Programm die Liste vorsortiert, echte Strategien lassen sich aber nur bei der Analyse mehrerer *classifier* gleichzeitig erkennen, weshalb im Rahmen dieser Arbeit die Ausgabe dieser Listen lediglich zur Fehleranalyse der Simulation benutzt wurde.

Es ist anzunehmen, dass die jeweilige Umwelt, also die anderen Agenten, eine wesentliche Rolle spielen. Bei stichprobenartiger Untersuchung wurden beispielsweise Strategien gefunden, bei denen die Agenten bewusst dem Zielobjekt aus dem Weg gehen. Eine solche Strategie ist auf dem ersten Blick sinnlos. Im Kontext von egoistisch handelnden Agenten, eines Szenarien ohne Hindernisse und im Hinblick auf die eingebaute Belohnung bei der Erreichung eines Gebiets ohne Agenten (siehe Kapitel 3.4.3), erscheint dies aber in einem ganz anderen Licht.

6.1.5 Erhöhung des Bedarfs an Kollaboration

Die in dieser Arbeit verwendeten Szenarien konnten nur teilweise die Kollaboration zwischen den Agenten in den Vordergrund stellen. Ein einfaches Verfolgen, also eine lokale Strategie, führte bereits zum Erfolg. Aufgezeigt wird dies insbesondere beim Vergleich zwischen der einfachen mit der intelligenten Heuristik bei unterschiedlichen Geschwindigkeiten des Zielobjekts in Szenarien mit relativ wenigen Hindernissen (siehe Kapitel 5.2.2), obwohl sich das Zielobjekt in diesem Fall intelligent verhalten hat.

Ein Ansatz wurde in Kapitel 3.4 erwähnt, eine Abkehr von einem binären zu einem mehrwertigen *base reward* Wert, um die *reward* Funktion der intelligenten Heuristik noch etwas besser abbilden zu können. Auch wäre es sowohl was Kollaboration als auch z.B. Kommunikation betrifft, sinnvoll, zuerst einen neuen Agenten mit einer Heuristik zu entwickeln, der das jeweilige Szenario optimal löst. Darauf aufbauend könnte man dann wieder z.B. die Bewertungsfunktion anpassen.

Will man tiefer auf Kollaboration eingehen, drängt sich auch die Lösungsidee auf, die Problemstellung an sich zu ändern. Beispielsweise könnte man das Zielobjekt nur dann als überwacht einzustufen, wenn es gleichzeitig von mehreren Agenten oder von mehreren Seiten beobachtet wird.

6.1.6 Rotation des *condition* Vektors

Ursprünglich wurde das Szenario auf Basis von Rotation konzipiert. Die Annahme war, dass wenn ein Agent, eine für einen Satz an Sensordaten optimale *classifier set* Liste

gefunden hat, die *classifier set* Liste auch für Sensordaten eines um 90, 180 und 270 Grad gedrehten Szenarien (mit entsprechend 90, 180 und 270 Grad gedrehter Aktion des jeweiligen *classifier*) optimal sei. Aufgrund der deutlichen Komplexitätssteigerung des Programms, der niedrigeren Laufzeit und mangels konkreter Qualitätssteigerungen gegenüber dem Ansatz ohne Rotation wurde diese Idee jedoch verworfen.

Möglicherweise könnte man durch Hinzunahme eines weiteren Bits im *condition* Vektor, das bestimmt, ob dieser *classifier* gleichzeitig auch die drei rotierten Szenarien erkennen kann, die Leistung des Systems verbessern. Dies hätte jedoch weitere Untersuchungen erfordert, welche über den Rahmen dieser Arbeit hinausgegangen wären.

6.1.7 Anpassungsfähigkeit von SXCS

XCS im Allgemeinen besitzt die Eigenschaft zu lernen. Interessant wäre eine Untersuchung, ob und wie sich die in dieser Arbeit diskutierten Varianten sich an neue Szenarien anpassen können. Zwar wurde bei der Untersuchung der Szenarien mit zufälligen Hindernissen gezeigt, dass sie es schaffen, mit veränderten Hindernispositionen zurechtzukommen, es wurde jedoch nicht untersucht, wie schnell eine solche Anpassung erfolgt. Beispielsweise könnte man 10 Probleminstanzen das Säulenszenario testen lassen und dann mit derselben Population 10 Probleminstanzen mit dem schwierigen Szenario. Anschließend wäre eine Betrachtung der Zahl der neu erstellten *classifier*, des gleitenden Durchschnitts der Qualität und der Vergleich zu den Qualitäten bei separatem Testens beider Szenarien interessant.

6.1.8 Wechsel zwischen *explore*/*exploit* Phasen

Kapitel 3.6 stellte mehrere Arten des Wechsels zwischen der *explore* und *exploit* Phase vor. In der Literatur gibt es beispielsweise in [HR05] einen Ansatz, um die Wahrscheinlichkeit, in eine *explore* Phase zu wechseln bzw. in dieser Phase zu bleiben, während eines Durchlaufs mittels einer intelligenten Methode anzupassen.

Wie dies bei Überwachungsszenarien ausgenutzt werden könnte, ist noch nicht ganz verstanden. Die Ergebnisse bezüglich des Wechsels bei der Änderung des *base reward*

Werts in Kapitel 5.7 deuten darauf hin, dass ein striktes Verfolgen des Zielobjekts und ein eher zufälliges Herumlaufen im Szenario mit Hindernissen von Vorteil sein kann.

6.1.9 Anpassung des *maxStackSize* Werts

Wie in Kapitel 4.3.3 erwähnt, kann der *maxStackSize* Wert stark vom jeweiligen Szenario abhängen, wenn wichtige Entscheidungen weit zurückliegen. Mangels eines theoretischen Fundaments muss man zwischen den folgenden drei wirkenden Faktoren einen Kompromiss finden:

- Erstens gibt es die Verzögerung zu Beginn einer Probleminstanz und insbesondere zu Beginn eines Experiments, es kann u.U. bis zu $\frac{maxStackSize}{2}$ Schritte dauern, bis das erste Mal ein *classifier* aktualisiert wird.
- Auch werden bei einem großen *maxStackSize* Wert womöglich Aktionen positiv (oder negativ) bewertet, die an der Situation nicht beteiligt waren. Insbesondere gilt dies, wenn es sich um kurze lokale Entscheidungen handelt.
- Umgekehrt, wählt man den Stack zu klein, kann es sein, dass ein Überlauf und somit u.U. ein gewisser Fehler auftritt. Der Wert *maxStackSize* stellt also einen Kompromiss zwischen Zeitverzögerung bzw. Reaktionsgeschwindigkeit und Genauigkeit dar.

Bei der Besprechung von Ereignissen in Verbindung mit SXCS in Kapitel 4.3.2 hat man gesehen, dass sich die optimalen Werte für das Säulenszenario und das schwierige Szenario stark unterscheiden. Um sich die Anpassung mittels Testläufen an das jeweilige Szenario zu sparen, wäre es für den Algorithmus sinnvoll, eine Methode zu entwickeln, mit der sich der *maxStackSize* Wert während des Laufs an das jeweilige Szenario anpassen kann. Zur Entwicklung des Algorithmus müsste der Algorithmus allerdings in der Theorie erst näher analysiert werden.

6.1.10 Lernendes Zielobjekt

Sicher interessant ist auch der umgekehrte Ansatz, bei dem die Rollen von Agent und Zielobjekt, was die Bestimmung der Qualität betrifft, vertauscht werden. Dann wäre das

Zielobjekt das Objekt, das lernt und den Agenten ausweichen muss. Bei dieser Problemstellung fällt zwar der kollaborative Aspekt weg, es hat aber den Vorteil, mit derselben Simulation diese neue Probleminstanz untersuchen zu können. Für die Implementierung muss lediglich die *reward* Funktion entsprechend abgeändert werden, damit beispielsweise das Ausweichen von Agenten positiv bewertet wird. Bis auf die in Kapitel 2.5 erwähnte Sprungeigenschaft ist ein lernendes Zielobjekt ansonsten praktisch identisch mit der XCS bzw. SXCS Implementierung. Bei anfänglichen Tests konnten keine besonderen Erkenntnisse gewonnen werden, deshalb wurde der Ansatz nicht weiter verfolgt.

6.1.11 Verbesserung der DSXCS *reward* Funktion

Bei der angesprochenen XCS Variante DSXCS (siehe Kapitel 4.4) wurden eine Reihe von Ansatzmöglichkeiten besprochen, wie die *reward* Funktion verbessert werden könnte. Blickt man in die Literatur, so wird der grundsätzliche Ansatz bestätigt. Beispielsweise wird in [TTS01] erläutert, dass Szenarien, die keine Markow-Kette darstellen und bei denen somit frühere Sensordaten relevant für die Entscheidung in der Gegenwart sind, mittels Speicher besser gelöst werden können. Eine Arbeit [Lan98] zu diesem Thema wurde in der Einleitung in Kapitel 1.1.7 erwähnt.

6.2 Vorgehen und verwendete Hilfsmittel und Software

Begonnen wurde mit der YCS Implementierung [Bul03]. Sie ist in der Literatur wenig vertreten, die Implementierung bot aber einen guten Einstieg in das Thema, da sie sich auf das Wesentliche eines LCS beschränkte und nur wenige Optimierungen im Quelltext enthielt.

Das so gewonnene Wissen war Basis und Voraussetzung für das Verstehen und Nachvollziehen der XCS Implementierung. Insbesondere die Optimierungen und der etwas unsaubere Programmierstil in der Standardimplementierung bereiteten Probleme.

Aufgrund der Komplexität der Fragestellung beschränkte sich die Untersuchung zuerst auf den Fall, in dem lokale Informationen ohne zentrale Steuerung und ohne Kommuni-

kation auftreten. Dies machte die Verwendung komplexerer Simulationssysteme unnötig. Auch erschien die Einarbeitungszeit in Multiagenten Frameworks wie z.B. Repast [Rep] erschien zu hoch, wie auch die unbekannten Risiken, was Geschwindigkeit, Kompatibilität und Speicherverbrauch betraf, weshalb letztlich ein eigenes Simulationsprogramm entwickelt wurde.

Das Simulationsprogramm samt zugehöriger Oberfläche zur Erstellung von neuen Test-Jobs wurde in Java mit Hilfe von NetBeans IDE 6.5 [NB6] selbst entwickelt und gestaltet.

Der Quelltext ist auf der beiliegenden DVD, über [Lod09] oder alternativ per E-Mail unter clemens@lode.de verfügbar.

Für die Verlaufsgraphen wurde GnuPlot 4.2.4 [ea] benutzt, die Darstellungen der jeweiligen Konfiguration des Torus (siehe Kapitel 2) wurden im Programm mittels Gif89Encoder [Ell00] erstellt. Weitere Graphen und Darstellungen wurden in OpenOffice.org Impress und OpenOffice.org Calc [OO0] angefertigt.

Besonders hilfreich für die Programmierung der Simulation, der Testumgebung, des Konfigurationsmanagements und der Oberfläche waren hierzu Erfahrungen aus der Studienarbeit zum Thema *estimation of distribution algorithms* am Institut für Angewandte Informatik und Formale Beschreibungsverfahren (AIFB) der Universität Karlsruhe (TH) bei Dr. Jürgen Branke wie auch einer halbjährigen Arbeit zum Thema Zellularautomaten am Institut für Algorithmen und Kognitive Systeme der Universität Karlsruhe (TH) bei Herrn Dr. rer. nat. Thomas Worsch.

Wesentlicher Bestandteil der Konfigurationsoberfläche ist insbesondere eine Automatisierung der Erstellung von Konfigurationsdateien und Batchdateien für ein Einzelsystem bzw. für JoSchKA [Bon06] zum Testen einer ganzen Reihe von Szenarien und GnuPlot Skripts. Die Automatisierung war aufgrund der tausenden zu testender Szenarien und Parametereinstellungen entscheidend zur Durchführung dieser Arbeit.

Dieses Dokument schließlich wurde mittels dem LaTeXEditor LEd 0.5263 [SD] erstellt und mittels MiKTeX 2.7 [MTX] kompiliert.

6.3 Beschreibung des Konfigurationsprogramms

In Abbildung 6.1 ist ein Screenshot des gesamten Konfigurationsprogramms (um 90 Grad gedreht) abgebildet. Auf der rechten Seite sind die Ergebnisse aller bisherigen Läufe in einer Datenbank angeordnet, auf der linken Seite befindet sich das Konfigurationsmenü, um neue Testläufe zusammenzustellen. Dabei kann man mehrere Konfigurationen nacheinander eingeben, mittels des *Save* Knopfs speichern und schließlich mittels des *Package* Knopfs alle gespeicherten Konfigurationen zu einem Testpaket zusammenschnüren. Das Testpaket kann dann entweder lokal als Batchdatei oder mit Hilfe des gemeinsam generierten Skripts bei JoSchKa am AIFB hochgeladen und dort automatisch abgearbeitet werden. Die Ergebnisse werden für jedes Experiment in ein Verzeichnis geschrieben, aus der das Programm wiederum bei Betätigung des *Update* Knopfs einen Eintrag in der Datenbank generiert und darstellt.

Das Simulationsprogramm selbst wird mit einem oder mehreren Dateinamen als Parameter aufgerufen. Die zugehörigen Dateien enthalten die Konfigurationsdaten für jeweils einen Durchlauf und entsprechen im Aufbau dem Format, welches das Konfigurationsprogramm erstellt.

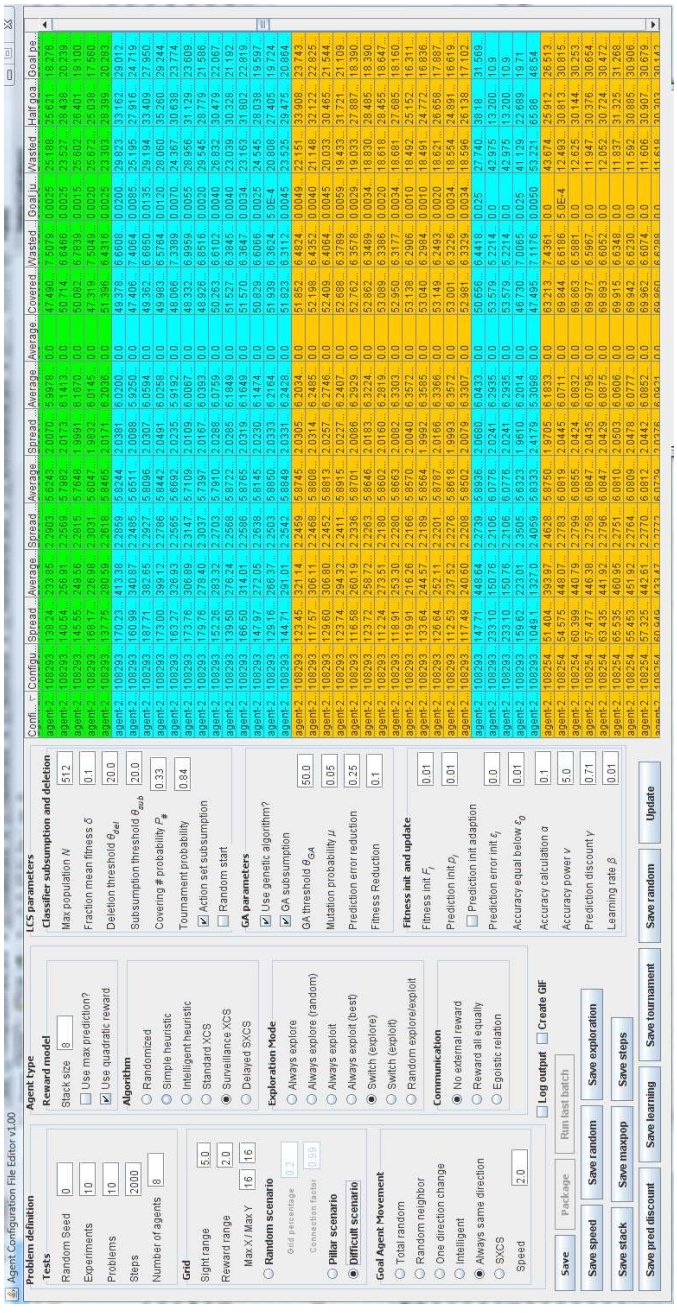

Abbildung 6.1: Screenshot des Konfigurationsprogramms (Gesamtübersicht)

Anhang A

Implementierung

Im Folgenden sind die Kernbestandteile aus dem Quelltext des Simulators aufgelistet. Sie dienen primär zum Verständnis der Idee und sind nur bedingt lauffähig. Der vollständige Quelltext ist auf der beiliegenden DVD bzw. unter [Lod09] verfügbar.

A.1 Implementierung eines Problemablaufs

In der Schleife der Funktion zur Berechnung eines Experiments (Programm A.1) wird die Funktion zur Berechnung einer Probleminstanz (*doOneMultiStepProblem()* in Programm A.2) aufgerufen. Dort wird in einer weiteren Schleife über die Anzahl der maximalen Schritte die Sicht aktualisiert (*updateSight()*), die Qualität bestimmt (*updateStatistics()*), die neuen Sensordaten und die nächste Aktion ermittelt (*calculateAgents()*, siehe Programm A.3), der *reward* Wert ermittelt (*rewardAgents()*, siehe Programm A.4) und schließlich werden die Objekte bewegt (*moveAgents()*, siehe Programm A.5). Die konkrete Umsetzung der dort aufgerufenen Funktionen (insbesondere *calculateNextMove()* und *calculateReward()*) wird im Kapitel 4 erläutert (bzw. in Kapitel 2.3.4, was die Heuristiken betrifft, wobei *calculateReward()* dort keine Rolle spielt und eine leere Funktion aufgerufen wird).

```
1  /**
2   * Führt eine Anzahl von Probleminstanzen aus.
3   * @param experiment_nr Nummer des auszuführenden Experiments
4   */
5    public void doOneMultiStepExperiment(final int experiment_nr) {
6      int currentTimestep = 0;
7
8    /**
9     * Durchlaufe Anzahl von Probleminstanzen für die selbe Population.
10    */
11     for (int i = 0; i < Configuration.getNumberOfProblems(); i++) {
12
13     /**
14      * Initialisierung des neuen "Random Seed" Wert
15      */
16       Misc.initSeed(Configuration.getRandomSeed() +
17         experiment_nr * Configuration.getNumberOfProblems() + i);
18
19     /**
20      * Erstellt einen neuen Torus und verteilt Agenten und
21      *    das Zielobjekt neu.
22      */
23       BaseAgent.grid.resetState();
24
25     /**
26      * Führe Probleminstanz aus und aktualisiere aktuellen Zeitschritt.
27      */
28       currentTimestep = doOneMultiStepProblem(currentTimestep);
29     }
30   }
```

Programm A.1: Zentrale Schleife für einzelne Experimente

```
1  /**
2   * Führt eine Anzahl von Schritten auf dem aktuellen Torus aus.
3   * @param stepCounter Der aktuelle Zeitschritt
4   * @return Der Zeitschritt nach der Ausführung
5   */
6    private int doOneMultiStepProblem(final int stepCounter) {
7    /**
8     * Zeitpunkt bis zu dem das Problem ausgeführt wird
9     */
10     int steps_next_problem =
11       Configuration.getNumberOfSteps() + stepCounter;
12     for (int currentTimestep = stepCounter;
13          currentTimestep < steps_next_problem; currentTimestep++) {
14
15     /**
16      * Ermittle die Sichtbarkeit und erhebe Statistiken.
17      */
18       BaseAgent.grid.updateSight();
19       BaseAgent.grid.updateStatistics(currentTimestep);
20
21     /**
22      * Ermittle neue Sensordaten und berechne Aktionen der Agenten.
23      */
24       calculateAgents(currentTimestep);
25
26     /**
27      * Ermittle den reward Wert für alle Agenten (nach dem ersten Schritt).
28      */
29       if(currentTimestep > stepCounter) {
30         rewardAgents(currentTimestep);
31       }
32
33     /**
34      * Führe zuvor berechnete Aktionen aus.
35      */
36       moveAgents();
37     }
38
39     /**
40      * Abschließende Ermittlung des reward Werts
41      */
42     BaseAgent.grid.updateSight();
43     rewardAgents(steps_next_problem);
44     return steps_next_problem;
45   }
```

Programm A.2: Zentrale Schleife für einzelne Probleminstanzen

```
1  /**
2   * Berechnet die Aktionen und führt sie in zufälliger Reihenfolge aus.
3   * @param gaTimestep Der aktuelle Zeitschritt
4   */
5    private void calculateAgents(final long gaTimestep) {
6
7     /**
8      * Ermittle Sensordaten und bestimme die nächste Bewegung.
9      */
10      for(BaseAgent a : agentList) {
11        a.aquireNewSensorData();
12        a.calculateNextMove(gaTimestep);
13      }
14      BaseAgent.goalAgent.aquireNewSensorData();
15      BaseAgent.goalAgent.calculateNextMove(gaTimestep);
16    }
```

Programm A.3: Zentrale Bearbeitung (Sensordaten und Berechnung der neuen Aktion) aller Agenten und des Zielobjekts innerhalb einer Probleminstanz

```
1  /**
2   * Rufe die Verarbeitungsfunktion für den reward Wert aller Agenten auf.
3   */
4    private void rewardAgents(final long gaTimestep) {
5      for(BaseAgent a : agentList) {
6        a.calculateReward(gaTimestep);
7      }
8      BaseAgent.goalAgent.calculateReward(gaTimestep);
9    }
```

Programm A.4: Zentrale Bearbeitung (Verteilung des *reward* Werts) aller Agenten und des Zielobjekts innerhalb einer Probleminstanz

```
1  /**
2   * Berechnet die Aktionen und führt sie in zufälliger Reihenfolge aus.
3   * @param gaTimestep Der aktuelle Zeitschritt
4   */
5    private void moveAgents(final long gaTimestep) {
6    /**
7     * Erstelle Ausführungsliste für alle Objekte (Zielobjekt mehrfach).
8     */
9      int goal_speed = Configuration.getGoalAgentMovementSpeed();
10     ArrayList<BaseAgent> random_list =
11       new ArrayList<BaseAgent>(agentList.size() + goal_speed);
12
13     random_list.addAll(agentList);
14     for(int i = 0; i < goal_speed; i++) {
15       random_list.add(BaseAgent.goalAgent);
16     }
17
18   /**
19    * Führe die ermittelten Aktionen in zufälliger Reihenfolge aus.
20    * Das Zielobjekt kann dabei mehrfach ausgeführt werden.
21    */
22     int[] array = Misc.getRandomArray(random_list.size());
23     for(int i = 0; i < array.length; i++) {
24       BaseAgent a = random_list.get(array[i]);
25       a.doNextMove();
26       if(a.isGoalAgent() && goal_speed > 1) {
27         goal_speed--;
28         a.aquireNewSensorData();
29         a.calculateNextMove(gaTimestep);
30         a.calculateReward(gaTimestep);
31       }
32     }
33   }
```

Programm A.5: Zentrale Bearbeitung (Ausführung der Bewegung) aller Agenten und des Zielobjekts innerhalb einer Probleminstanz

A.2 Typen von Agentenbewegungen

```
/**
 * Berechne die nächste Aktion (zufälliger Algorithmus).
 */
  private void calculateNextMove() {
  /**
   * Wähle eine zufällige Richtung als nächste Aktion.
   */
    calculatedAction = Misc.nextInt(Action.MAX_DIRECTIONS);
  }
```

Programm A.6: Berechnung der nächsten Aktion bei der Benutzung des Algorithmus mit zufälliger Bewegung

```
/**
 * Berechne die nächste Aktion (einfache Heuristik).
 */
  private void calculateNextMove() {
  /**
   * Ermittlung der Informationen der Gruppe der Sensoren, die auf
   * das Zielobjekt ausgerichtet sind
   */
    boolean[] goal_sensor = lastState.getSensorGoal();
    calculatedAction = -1;
    for(int i = 0; i < Action.MAX_DIRECTIONS; i++) {
    /**
     * Zielagent in Sicht in dieser Richtung?
     */
      if(goal_sensor[2*i]) {
        calculatedAction = i;
        break;
      }
    }

  /**
   * Sonst wähle zufällige Richtung als nächste Aktion.
   */
    if(calculatedAction == -1) {
      calculatedAction = Misc.nextInt(Action.MAX_DIRECTIONS);
    }

  }
```

Programm A.7: Berechnung der nächsten Aktion bei der Benutzung der einfachen Heuristik

```
1  /**
2   * Berechne nächste Aktion (intelligente Heuristik).
3   */
4  private void calculateNextMove() {
5    /**
6     * Ermittlung der Informationen der Gruppe der Sensoren, die auf
7     * das Zielobjekt ausgerichtet sind
8     */
9      boolean[] goal_sensor = lastState.getSensorGoal();
10
11     calculatedAction = -1;
12     for(int i = 0; i < Action.MAX_DIRECTIONS; i++) {
13     /**
14      * Zielagent in Sicht in dieser Richtung?
15      */
16       if(goal_sensor[2*i]) {
17         calculatedAction = i;
18         break;
19       }
20     }
21
22   /**
23    * Zielobjekt nicht in Sicht? Dann bewege von Agenten weg.
24    */
25     if(calculatedAction == -1) {
26       calculatedAction = Misc.nextInt(Action.MAX_DIRECTIONS);
27
28       boolean[] agent_sensors = lastState.getSensorAgent();
29       boolean one_free = false;
30       for(int i = 0; i < Action.MAX_DIRECTIONS; i++) {
31         if(!agent_sensors[2*i]) {
32           one_free = true;
33           break;
34         }
35       }
36
37       if(one_free) {
38         while(agent_sensors[2*calculatedAction]) {
39           calculatedAction = Misc.nextInt(Action.MAX_DIRECTIONS);
40         }
41       }
42     }
43   }
```

Programm A.8: Berechnung der nächsten Aktion bei der Benutzung der intelligenten Heuristik

A.3 Bewertungsfunktion

```
1
2  /**
3   * Berechnet den base reward Wert und gibt ihn zurück.
4   * @return Den base reward Wert
5   */
6    public boolean checkRewardPoints() {
7      if(lastState == null) {
8        return false;
9      }
10   /**
11    * Prüft die eigenen Sensordaten.
12    */
13     boolean[] sensor_agent = lastState.getSensorAgent();
14     boolean[] sensor_goal = lastState.getSensorGoal();
15
16   /**
17    * Ziel in Sicht? Dann gebe positiven base reward Wert zurück.
18    */
19     for(int i = 0; i < Action.MAX_DIRECTIONS; i++) {
20       if((sensor_goal[2*i])) {
21         return true;
22       }
23     }
24
25   /**
26    * Kein Zielobjekt, aber Agent in Sicht?
27    * Dann gebe negativen base reward Wert zurück.
28    */
29     for(int i = 0; i < Action.MAX_DIRECTIONS; i++) {
30       if(sensor_agent[2*i]) {
31         return false;
32       }
33     }
34
35   /**
36    * Kein Zielobjekt und kein Agent in Sicht?
37    * Dann gebe positiven base reward Wert zurück.
38    */
39     return true;
40   }
```

Programm A.9: Bewertungsfunktion für die XCS Varianten

A.4 Korrigierte *addNumerosity()* Funktion

Durch die Benutzung von *macro classifier* ergibt sich das programmiertechnische Problem, dass man nicht mehr direkt weiß, wieviele *micro classifier* sich in einer Population befinden, bei jeder Benutzung des Werts der Populationsgröße müssten die *numerosity* Werte aller *classifier* jedes Mal addiert werden. In der Standardimplementierung [But00] ist die Behandlung des *numerosity* Werts deswegen stark optimiert, jedes *classifier set* trägt eine temporäre Variable *numerositySum* mit sich, in der die aktuelle Summe gespeichert ist. Die Aktualisierung ist jedoch zum einen mangelhaft umgesetzt, zum anderen auf die Verwendung von einer einzelnen *action set* Liste optimiert. Dagegen wurde die für diese Arbeit erstellte Implementierung jeweils mit bis über 100 *action set* Listen programmiert, denen ein *classifier* Mitglied sein kann. Deswegen wurde die Optimierung entfernt und durch eine dezentrale Verwaltung mit einem *Observer* ersetzt, jede Änderung des *numerosity* Wertes hat also die Änderung aller *action set* Listen zur Folge, in welcher der *classifier* Mitglied ist.

Wird also ein *micro classifier* entfernt, dann wird lediglich die Änderungsfunktion des jeweiligen *classifier* aufgerufen, der dann wiederum den *numerositySum* Wert der jeweiligen Eltern anpasst. Dies macht einige Optimierungen rückgängig, erspart aber sehr viele Umstände, den *numerositySum* der Eltern immer auf den aktuellen Stand zu halten und einzelne *classifier* zu löschen.

Positiver Nebeneffekt durch die verbesserte Struktur ist der vereinfachte Zufgriff auf die Menge der *action set* Listen, denen ein *classifier* angehört, hierfür wurde aber im Rahmen dieser Arbeit keine Verwendung gefunden.

Die Standardimplementierung weist ein weiteres Problem auf. So enthält der *fitness* Wert eines *classifier* als Optimierung bereits den *numerosity* Wert als Faktor, während bei der Aktualisierung des *numerosity* Werts der *fitness* Wert nicht aktualisiert wurde. Das hat zur Folge, dass theoretisch der *fitness* Wert eines *classifier* fast den *max population* Wert annehmen kann, wenn ein *classifier* mit *numerosity* und *fitness* Wert in der Höhe von *max population* auf einen *numerosity* Wert von 1,0 reduziert wird. Dies betrifft die Funktion *public void addNumerosity(int num)* der Klasse *XClassifier* in der Datei *XClassifier.java*. Die Korrektur besteht darin, den *fitness* Wert mit dem Quotienten aus dem

neuen durch den alten *numerosity* Wert zu multiplizieren. Die korrigierte Fassung ist in Programm A.10 dargestellt.

Möglicherweise kann man diesen Fehler durch Veränderung der Parameter oder längere Laufzeiten kompensieren. Es ergibt jedoch keinen Sinn, dass beim Subsummieren bzw. Löschen eines *micro classifier* der *fitness* Wert verändert wird. In Tests haben sich nur minimale Unterschiede ergeben. Beispielsweise ergab sich (auf dem Säulenszenario mit 8 Agenten mit SXCS und einem Zielobjekt mit einfacher Richtungsänderung und Geschwindigkeit 2) eine Qualität von 32,43% im Vergleich zur originalen Implementierung von 32,20% bei 500 Schritten bzw. 36,28% zu 35,75% bei 2.000 Schritten. Der Fehler scheint sich also minimal negativ auf die Qualität auszuwirken. Problematisch wird es aber auf theoretischer Ebene, wenn Modifikationen von XCS darauf aufbauen, dass der *fitness* Wert für jeden *micro classifier* immer kleiner gleich 1,0 ist. Dies kann u.U. zu größeren Fehlern führen.

Alles in allem betrachtet soll im Rahmen dieser Arbeit die korrigierte Fassung benutzt werden.

```
1  /**
2   * Erhöht oder erniedrigt den numerosity Wert des classifier.
3   * @param num Addiere num (kann negativ sein) zum numerosity Wert.
4   */
5    public void addNumerosity(int num) {
6      int old_num = numerosity;
7
8      numerosity += num;
9
10   /**
11    * Korrektur des fitness Werts
12    */
13     if(old_num > 0) {
14       fitness = fitness * (double)numerosity / (double)old_num;
15     } else {
16       fitness = Configuration.getFitnessInitialization();
17     }
18
19   /**
20    * Aktualisierung der Eltern
21    */
22     for (ClassifierSet p : parents) {
23       p.changeNumerositySum(num);
24       if (numerosity == 0) {
25         p.removeClassifier(this);
26       }
27     }
28   }
```

Programm A.10: Korrigierte Version der *addNumerosity()* Funktion

A.5 Implementierung des XCS *multi step* Verfahrens

```
1  /**
2   * Diese Funktion wird in jedem Schritt aufgerufen um den aktuellen
3   * reward Wert zu bestimmen, den besten Wert der ermittelten match set
4   * Liste weiterzugeben und, bei aktuell positivem reward Wert, die
5   * aktuelle action set Liste positiv zu bewerten.
6   *
7   * @param gaTimestep Der aktuelle Zeitschritt
8   */
9
10   public void calculateReward(final long gaTimestep) {
11   /**
12    * checkRewardPoints() liefert "wahr" wenn sich das Zielobjekt in
13    * Sichtweite oder sich keine Agenten in Sichtweite befindet.
14    */
15     boolean reward = checkRewardPoints();
16
17     if(prevActionSet != null){
18       collectReward(lastReward, lastMatchSet.getBestValue(), false);
19       prevActionSet.evolutionaryAlgorithm(classifierSet, gaTimestep);
20     }
21
22     if(reward) {
23       collectReward(reward, 0.0, true);
24       lastActionSet.evolutionaryAlgorithm(classifierSet, gaTimestep);
25       prevActionSet = null;
26       return;
27     }
28     prevActionSet = lastActionSet;
29     lastReward = reward;
30   }
```

Programm A.11: Erstes Kernstück des Standard XCS *multi step* Verfahrens (*calculateReward()*, Bestimmung und Verarbeitung des *reward* Werts anhand der Sensordaten), angepasst an ein dynamisches Überwachungsszenario

```
1  /**
2   * Diese Funktion verarbeitet den übergebenen reward Wert und
3   * gibt ihn an die zugehörigen action set Listen weiter.
4   *
5   * @param reward Wahr wenn das Zielobjekt in Sicht war.
6   * @param best_value Bester Wert des vorangegangenen action set Listen
7   * @param is_event Wahr wenn diese Funktion wegen eines Ereignisses,
8   *           d.h. einem positiven reward Wert, aufgerufen wurde.
9   */
10
11   public void collectReward(final boolean reward,
12                   final double best_value, final boolean is_event) {
13
14     double corrected_reward = reward ? 1.0 : 0.0;
15
16    /**
17     * Falls der reward Wert von einem Ereignis rührt, dann aktualisiere
18     * die aktuelle action set Liste und lösche die vorherige.
19     */
20      if(is_event) {
21        if(lastActionSet != null) {
22          lastActionSet.updateReward(corrected_reward, best_value, 1.0);
23          prevActionSet = null;
24        }
25      }
26
27    /**
28     * Kein Ereignis, also nur die letzte action set Liste aktualisieren.
29     */
30      else
31      {
32        if(prevActionSet != null) {
33          prevActionSet.updateReward(corrected_reward, best_value, 1.0);
34        }
35      }
36   }
```

Programm A.12: Zweites Kernstück des XCS *multi step* Verfahrens (*collectReward()*, Verteilung des *reward* Werts auf die *action set* Listen), angepasst an ein dynamisches Überwachungsszenario

```
1  /**
2   * Bestimmt die zum letzten bekannten Status passenden classifier und
3   * wählt aus dieser Menge eine Aktion. Außerdem wird die aktuelle
4   * action set Liste mithilfe der gewählten Aktion ermittelt.
5   *
6   * @param gaTimestep Der aktuelle Zeitschritt
7   */
8
9    public void calculateNextMove(final long gaTimestep) {
10
11  /**
12   * Überdecke das classifierSet mit zum Status passenden classifier
13   * welche insgesamt alle möglichen Aktionen abdecken.
14   */
15     classifierSet.coverAllValidActions(
16                       lastState, getPosition(), gaTimestep);
17
18  /**
19   * Bestimme alle zum Status passenden classifier.
20   */
21     lastMatchSet = new AppliedClassifierSet(lastState, classifierSet);
22
23  /**
24   * Entscheide auf welche Weise die Aktion ausgewählt werden soll.
25   */
26     lastExplore = checkIfExplore(lastState.getSensorGoalAgent(),
27                                              lastExplore, gaTimestep);
28
29  /**
30   * Wähle eine Aktion und bestimme die zugehörige action set Liste.
31   */
32     calculatedAction = lastMatchSet.chooseAbsoluteDirection(lastExplore);
33     lastActionSet = new ActionClassifierSet(lastState, lastMatchSet,
34                                                    calculatedAction);
35   }
```

Programm A.13: Drittes Kernstück des XCS *multi step* Verfahrens (*calculateNextMove()*, Auswahl der nächsten Aktion und Ermittlung der zugehörigen *action set* Liste), angepasst an ein dynamisches Überwachungsszenario

A.6 Implementierung des SXCS Verfahrens

```
/**
 * Diese Funktion wird in jedem Schritt aufgerufen um den aktuellen
 * reward Wert zu bestimmen und positive, negative und neutrale
 * Ereignisse zu verarbeiten.
 *
 * @param gaTimestep Der aktuelle Zeitschritt
 */

  public void calculateReward(final long gaTimestep) {
  /**
   * checkRewardPoints() liefert "wahr" wenn sich das Zielobjekt in
   * Sichtweite oder sich keine Agenten in Sichtweite befindet.
   */
    boolean reward = checkRewardPoints();

  /**
   * Positives oder negatives Ereignis?
   */
    if (reward != lastReward) {
      int start_index = historicActionSet.size() - 1;
      collectReward(start_index, actionSetSize, reward, true);
      actionSetSize = 0;
    }
    else

  /**
   * Neutrales Ereignis?
   */
    if(actionSetSize >= Configuration.getMaxStackSize())
    {
      int start_index = Configuration.getMaxStackSize() / 2;
      int length = actionSetSize - start_index;
      collectReward(start_index, length, reward, false);
      actionSetSize = start_index;
    }

    lastReward = reward;
  }
```

Programm A.14: Erstes Kernstück des SXCS-Algorithmus (*calculateReward()*, Bestimmung des *reward* Werts anhand der Sensordaten)

```
1  /**
2   * Diese Funktion verarbeitet den übergebenen base reward Wert und
3   * gibt ihn an die zugehörigen action set Listen weiter. Es wird
4   * kein maxPrediction Wert berechnet oder weitergegeben.
5   *
6   * @param start_index Index in der historicActionSet Liste der als
7   *           erstes aktualisiert werden soll.
8   * @param action_set_size Anzahl der action set Listen, die aktualisiert
9   *           werden sollen.
10  * @param reward Wahr wenn der Zielobjekt in Sicht oder keine Agenten
11  *           in Sicht waren.
12  * @param is_event Wahr wenn diese Funktion wegen eines positiven oder
13  *           negativen Ereignisses aufgerufen wurde.
14  */
15
16   public void collectReward(final int start_index,
17                 final int action_set_size, final boolean reward,
18                 final boolean is_event) {
19     double corrected_reward = reward ? 1.0 : 0.0;
20
21   /**
22    * Aktualisiere eine ganze Anzahl von action set Listen
23    */
24     for(int i = 0; i < action_set_size; i++) {
25
26   /**
27    * Benutze aufsteigenden bzw. absteigenden reward Wert bei einem
28    * positiven bzw. negativen Ereignis.
29    */
30       if(is_event) {
31         corrected_reward = reward ?
32           calculateReward(i, action_set_size) :
33           calculateReward(action_set_size - i, action_set_size);
34       }
35   /**
36    * Aktualisiere die action set Liste mit dem bestimmten reward Wert.
37    */
38       ActionClassifierSet action_classifier_set =
39         historicActionSet.get(start_index - i);
40       action_classifier_set.updateReward(corrected_reward, 0.0, 1.0);
41     }
42   }
```

Programm A.15: Zweites Kernstück des SXCS-Algorithmus (*collectReward()* - Verteilung des *reward* Werts auf die *action set* Listen)

```
1  /**
2   * Bestimmt die zum letzten bekannten Status passenden classifier und
3   * wählt aus dieser Menge eine Aktion. Außerdem wird die aktuelle
4   * action set Liste mithilfe der gewählten Aktion ermittelt.
5   * Im Vergleich zum originalen multi step Verfahren wird am Schluss noch
6   * die ermittelte action set Liste gespeichert.
7   *
8   * @param gaTimestep Der aktuelle Zeitschritt
9   */
10
11   public void calculateNextMove(final long gaTimestep) {
12
13  /**
14   * Überdecke die classifier set Liste mit zum Status passenden
15   * classifier welche insgesamt alle möglichen Aktionen abdecken.
16   */
17     classifierSet.coverAllValidActions(
18                     lastState, getPosition(), gaTimestep);
19
20  /**
21   * Bestimme alle zum Status passenden classifier.
22   */
23     lastMatchSet = new AppliedClassifierSet(lastState, classifierSet);
24
25  /**
26   * Entscheide auf welche Weise die Aktion ausgewählt werden soll,
27   * wähle eine Aktion und bestimme die zugehöriges action set Liste.
28   */
29     lastExplore = checkIfExplore(lastState.getSensorGoalAgent(),
30                                          lastExplore, gaTimestep);
31
32     calculatedAction = lastMatchSet.chooseAbsoluteDirection(lastExplore);
33     lastActionSet = new ActionClassifierSet(lastState, lastMatchSet,
34                                                  calculatedAction);
35
36  /**
37   * Speichere die action set Liste, erhöhe die Größe des Stacks und
38   * und passe den Stack bei einem Überlauf an.
39   */
40     actionSetSize++;
41     historicActionSet.addLast(lastActionSet);
42     if (historicActionSet.size() > Configuration.getMaxStackSize()) {
43       historicActionSet.removeFirst();
44     }
45   }
```

Programm A.16: Drittes Kernstück des SXCS-Algorithmus (*calculateNextMove()* - Auswahl der nächsten Aktion und Ermittlung und Speicherung der zugehörigen *action set* Liste)

A.7 Implementierung des DSXCS Algorithmus

```
1  /**
2   * Diese Funktion verarbeitet den übergebenen base reward Wert und
3   * gibt ihn an die zugehörigen action set Listen weiter. Es wird
4   * kein maxPrediction Wert berechnet oder weitergegeben.
5   *
6   * @param start_index Index in der historicActionSet Liste der als
7   *           erstes aktualisiert werden soll.
8   * @param action_set_size Anzahl der action set Listen, die aktualisiert
9   *           werden sollen.
10  * @param reward Wahr wenn der Zielobjekt in Sicht oder keine Agenten
11  *           in Sicht waren.
12  * @param factor Der Aktualisierungsfaktor
13  * @param is_event Wahr wenn diese Funktion wegen eines positiven oder
14  *           negativen Ereignisses aufgerufen wurde.
15  */
16   public void collectReward(final int start_index,
17                 final int action_set_size, final boolean reward,
18                 final double factor, final boolean is_event) {
19
20     double corrected_reward = reward ? 1.0 : 0.0;
21
22   /**
23    * Aktualisiere eine ganze Anzahl von Einträgen in der
24    * historicActionSet Liste.
25    */
26     for(int i = 0; i < action_set_size; i++) {
27
28     /**
29      * Benutze aufsteigenden bzw. absteigenden reward Wert bei
30      * einem positiven bzw. negativen Ereignis.
31      */
32       if(is_event) {
33         corrected_reward = reward ?
34           calculateReward(i, action_set_size) :
35           calculateReward(action_set_size - i, action_set_size);
36       }
37     /**
38      * Füge den ermittelten reward Wert zur historicActionSet Liste.
39      */
40       historicActionSet.get(start_index - i).
41         addReward(corrected_reward, factor);
42     }
43   }
```

Programm A.17: Erstes Kernstück des verzögerten SXCS Algorithmus DSXCS (*collectReward()*, Bewertung der *action set* Listen)

```
1
2  /**
3   * Der erste Teil der Funktion ist identisch mit der calculateNextMove()
4   * Funktion der SXCS Variante ohne Kommunikation. Der Zusatz ist, dass beim
5   * Überlauf die in der historicActionSet Liste gespeicherte reward Werte
6   * verarbeitet werden.
7   *
8   * @param gaTimestep Der aktuelle Zeitschritt
9   */
10
11   public void calculateNextMove(final long gaTimestep) {
12
13   // ...
14
15   /**
16    * historicActionSet Liste voll? Dann verarbeite den dortigen reward Wert.
17    */
18      if (historicActionSet.size() > Configuration.getMaxStackSize()) {
19        historicActionSet.pop().processReward();
20     }
21   }
```

Programm A.18: Zweites Kernstück des verzögerten SXCS Algorithmus DSXCS (*calculateNextMove()*, Auswahl der nächsten Aktion und Ermittlung der zugehörigen *action set* Liste)

```
1  /**
2   * Zentrale Routine der historicActionSet Liste zur Verarbeitung aller
3   * eingegangenen reward Werte bis zu diesem Punkt.
4   */
5
6   public void processReward() {
7
8     for(RewardHelper r : reward) {
9     /**
10     * Aktualisiere den Eintrag mit den entsprechenden Werten.
11     */
12      actionClassifierSet.updateReward(r.reward, 0.0, r.factor);
13    }
14  }
```

Programm A.19: Drittes Kernstück des verzögerten SXCS Algorithmus DSXCS (Verarbeitung des jeweiligen reward Werts, *processReward()*)

```
1  /**
2   * Verbesserte zentrale Routine der historicActionSet Liste zur
3   * Verarbeitung aller eingegangenen reward Wert bis zu diesem Punkt.
4   */
5
6    public void processReward() {
7
8      double max_value = 0.0;
9      double max_reward = 0.0;
10   /**
11    * Finde das größte reward / factor Paar.
12    */
13     for(RewardHelper r : reward) {
14
15       if(r.reward * r.factor > max_value) {
16         max_value = r.reward * r.factor;
17         max_reward = r.reward;
18       }
19     }
20     /**
21      * Aktualisiere den Eintrag mit dem ermittelten Werten.
22      */
23     actionClassifierSet.updateReward(max_reward, 0.0, 1.0);
24   }
```

Programm A.20: Verbesserte Variante des dritten Kernstück des verzögerten SXCS Algorithmus DSXCS (Verarbeitung des jeweiligen reward Werts, *processReward()*)

A.8 Implementierung des egoistischen *reward*

```
1  /**
2   * Ähnlichkeit dieser classifier set Liste zu der übergebenen Liste im
3   * Hinblick auf die Wahrscheinlichkeit, auf andere Agenten zuzugehen
4   * @param other Die andere Liste mit der verglichen werden soll
5   * @return Grad der Ähnlichkeit bzw. der Kommunikationsfaktor (0,0 - 1,0)
6   */
7    public double checkEgoisticDegreeOfRelationship(
8                       final MainClassifierSet other) {
9      double ego_factor = getEgoisticFactor() - other.getEgoisticFactor();
10     if(ego_factor == 0.0) {
11       return 0.0;
12     } else {
13       return 1.0 - ego_factor * ego_factor;
14     }
15   }
16
17   public double getEgoisticFactor() throws Exception {
18     double factor = 0.0;
19     double pred_sum = 0.0;
20     for(Classifier c : getClassifiers()) {
21       if(!c.isPossibleSubsumer()) {
22         continue;
23       }
24       factor += c.getEgoFactor();
25       pred_sum += c.getFitness() * c.getPrediction();
26     }
27     if(pred_sum > 0.0) {
28       factor /= pred_sum;
29     } else {
30       factor = 0.0;
31     }
32     return factor;
33   }
```

Programm A.21: "Egoistische Relation", Algorithmus zur Bestimmung des Kommunikationsfaktors basierend auf dem erwarteten Verhalten des Agenten gegenüber anderen Agenten

Literaturverzeichnis

[BGL05] M. V. Butz, D. E. Goldberg, and P. L. Lanzi. Gradient descent methods in learning classifier systems: improving xcs performance in multistep problems. *IEEE Transactions on Evolutionary Computation*, 9(5):452–473, Oct. 2005.

[Bon06] Matthias Bonn. Joschka job manager 4.0.3161.17992, 2006. Available from: `http://www.aifb.uni-karlsruhe.de/EffAlg/mbo/joschka/index.html`.

[Bre65] J.E. Bresenham. Algorithm for computer control of a digital plotter. *IBM Systems J.*, 4(1):25–30, 1965.

[Bul03] Larry Bull. A simple accuracy-based learning classifier system. Technical report, Learning Classifier Systems Group Technical Report UWELCSG03-005, 2003. Available from: `http://www2.cmp.uea.ac.uk/~it/ycs/ycs.pdf`.

[But00] Martin V. Butz. Xcs classifier system in java, 2000. Available from: `http://www.illigal.uiuc.edu/pub/papers/IlliGALs/2000027.ps.Z`.

[But06a] Martin V. Butz. *Simple Learning Classifier Systems*, chapter 4, pages 31–50. Springer, 2006.

[But06b] Martin V. Butz. *The XCS Classifier System*, chapter 4, pages 51–64. Springer, 2006.

[BW01] Martin V. Butz and Stewart W. Wilson. An algorithmic description of XCS. *Lecture Notes in Computer Science*, 1996:253–272, 2001.

[ea] Thomas Williams et al. Gnuplot 4.2.4. Available from: `http://www.gnuplot.info/`.

[Ell00] J. M. G. Elliott. Gif89encoder 0.90 beta, Jul. 2000. Available from: `http://jmge.net/java/gifenc/`.

[HFA02] Luis Miramontes Hercog, Terence C. Fogarty, and London Se Aa. Social simulation using a multi-agent model based on classifier systems: The emergence of vacillating behaviour in the „el farol“ bar problem. In *Proceedings of the International Workshop in Learning Classifier Systems 2001*. Springer-Verlag, 2002.

[HR05] Ali Hamzeh and Adel Rahmani. Intelligent exploration method for xcs. In *GECCO '05: Proceedings of the 2005 workshops on Genetic and evolutionary computation*, pages 100–102, New York, NY, USA, 2005. ACM.

[ITS05] Hiroyasu Inoue, Keiki Takadama, and Katsunori Shimohara. Exploring xcs in multiagent environments. In *GECCO '05: Proceedings of the 2005 workshops on Genetic and evolutionary computation*, pages 109–111, New York, NY, USA, 2005. ACM.

[KM94] S. Kobayashi K. Miyazaki, M. Yamamura. On the rationality of profit sharing in reinforcement learning. In *Proceedings of the 3rd International Conference on Fuzzy Logic, Neural Nets and Soft Computing*, pages 285–288, 1994.

[Lan] P. L. Lanzi. The xcs library. Available from: `http://xcslib.sourceforge.net`.

[Lan98] Pier Luca Lanzi. Adding memory to XCS. In *Proceedings of the IEEE Conference on Evolutionary Computation (ICEC98)*. IEEE Press, 1998. Available from: `citeseer.ist.psu.edu/393845.html`.

[Lod09] Clemens Lode. Agentsimulator 1.0, 2009. Available from: `http://www.clawsoftware.de/AgentSimulator10.zip`.

[LW99] Pier Luca Lanzi and Stewart W. Wilson. Optimal classifier system performance in non-markovian environments. Technical Report 99.36, Milan, Italy, 1999. Available from: `citeseer.ist.psu.edu/lanzi99optimal.html`.

[LWB08] A. Lujan, R. Werner, and A. Boukerche. Generation of rule-based adaptive strategies for a collaborative virtual simulation environment. In *Proc. IEEE International Workshop on Haptic Audio visual Environments and Games HAVE 2008*, pages 59–64, 18–19 Oct. 2008.

[MC94] Geoffrey F. Miller and Dave Cliff. Co-evolution of pursuit and evasion i: Biological and game-theoretic foundations. Technical Report CSRP311, August 1994.

[MMGG95] Brad L. Miller, Brad L. Miller, David E. Goldberg, and David E. Goldberg. Genetic algorithms, tournament selection, and the effects of noise. *Complex Systems*, 9:193–212, 1995.

[MSB05] Alex McMahon, Dan Scott, and Will Browne. An autonomous explore/exploit strategy. In *GECCO '05: Proceedings of the 2005 workshops on Genetic and evolutionary computation*, pages 103–108, New York, NY, USA, 2005. ACM.

[MTX] Miktex 2.7. Available from: `http://www.miktex.org/`.

[MVBG03] K. Sastry M. V. Butz and D. E. Goldberg. Tournament selection: Stable fitness pressure in xcs. In *Lecture Notes in Computer Science*, pages 1857––1869, 2003.

[NB6] Netbeans ide 6.5. Available from: `http://www.netbeans.org`.

[OO0] Openoffice.org. Available from: `http://www.openoffice.org`.

[Rep] Repast agent simulation toolkit. Available from: `http://repast.sourceforge.net/`.

[SD] Adam Skórczynski and Sebastian Deorowicz. Latex editor led. Available from: `http://www.latexeditor.org/`.

[TTS01] Keiki Takadama, Takao Terano, and Katsunori Shimohara. Learning classifier systems meet multiagent environments. In *IWLCS '00: Revised Papers from the Third International Workshop on Advances in Learning Classifier Systems*, pages 192–212, London, UK, 2001. Springer-Verlag.

[Wei00a] G. Weiss. *Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence*, pages 586–587. The MIT Press, July 2000. "Collaboration".

[Wei00b] G. Weiss. *Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence*, page 30. The MIT Press, July 2000. "Static vs. Dynamic".

[Wil95] Stewart W. Wilson. Classifier fitness based on accuracy. *Evolutionary Computation*, 2(3):149–175, 1995.

[Wil98] Stewart W. Wilson. Generalization in the xcs classifier system. In *Genetic Programming 1998: Proceedings of the Third Annual Conference*, pages 665–674. Morgan Kaufmann, 1998.

[Wil00] Stewart W. Wilson. Get real! xcs with continuous-valued inputs. In *Learning Classifier Systems, From Foundations to Applications*, pages 209–222, London, UK, 2000. Springer-Verlag.

[WS08] K.-H. Waldmann and U. M. Stocker. *Stochastische Modelle*, chapter 2, page 11. Springer, 2008.

Erklärung

Ich versichere hiermit wahrheitsgemäß, die Arbeit bis auf die dem Aufgabensteller bereits bekannte Hilfe selbständig angefertigt, alle benutzten Hilfsmittel vollständig und genau angegeben und alles kenntlich gemacht zu haben, was aus Arbeiten anderer unverändert oder mit Abänderungen entnommen wurde.

Karlsruhe, 31. März 2009,

Clemens Lode