DIPLOMARBEIT

XCS in dynamischen Multiagenten-Überwachungsszenarien ohne globale Kommunikation

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, 30.03.2009

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Programmverzeichnis

Kapitel 1

Einleitung und Motivation

Ein aktuelles Forschungsgebiet aus dem Bereich der *learning classifier systems* (LCS) stellen die sogenannten *eXtended Classifier System* (XCS) dar. In der Basis entspricht XCS einem LCS, d.h. eine Reihe von Regeln, bestehend jeweils aus einer Kondition und einer Aktion. Die Regeln werden mittels *reinforcement learning* schrittweise bewertet und an eine Umwelt angepasst. Die Frage nach dem Zeitpunkt der Bewertung teilt die verwendeten Algorithmen bei XCS in *single step* und *multi step* Verfahren ein. Hauptaugenmerk dieser Arbeit ist das *multi step* Verfahren, bei dem die Bewertung der *reward* Wert der Regeln erst nach einigen Schritten verfügbar ist und an zurückliegende Regeln sukzessive weitergeleitet wird, um möglichst alle beteiligten Regeln an dem *reward* Wert zu beteiligen.

Bisherige Anwendungen haben sich hauptsächlich auf statische Szenarien mit nur einem XCS oder mit mehreren Agenten mit globaler Organisation und Kommunikation beschränkt. Diese Arbeit konzentriert sich auf das Problem, ob und wie es gelingen kann, XCS so zu modifizieren, damit es dynamische Überwachungsszenarien, mit sich bewegendem Zielobjekt und mehreren Agenten, im Vergleich zu Agenten mit zufälliger Bewegung,

möglichst gut besteht.

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

Damit XCS für eine solche Problemstellung überhaupt anwendbar ist, waren einige allgemeinen Anpassungen vonnöten. Desweiteren wurden eine Reihe von zusätzlicher Verbesserungen und Modifikationen implementiert, die zu teils deutlich besseren Ergebnissen als die der Standardimplementation führten.

Außerdem wurde untersucht, wie eine einfache Kommunikation ohne globale Steuereinheit stattfinden kann, um das Ergebnis weiter zu verbessern. Im Wesentlichen war dazu eine weitere Anpassung von XCS vonnöten, so dass die Implementierung auch mit (durch die Kommunikation) zeitverzögerten Bewertungen und Bewertungen von anderen Agenten arbeiten konnte.

Wesentliche Erkenntnisse sind, nicht jedes Szenarium eignet sich gleich gut für die Kommunikation, Kommunikation bietet Möglichkeiten zur Anpassung an mit einer variablen, unbekannten Feldgröße zurecht zu kommen und es gibt Szenarien, in denen Kommunikation signifikante Vorteile erbringt. TODO

Wesentliche Schlussfolgerung ist, dass sich unterschiedliche Szenarien unterschiedlich gut für Kommunikation eignen, dass Kommunikation Möglichkeiten zur Anpassung bietet, um mit einer variablen, unbekannten Feldgröße besser zurecht zu kommen und, dass es

Szenarien gibt, in denen Kommunikation signifikante Vorteile erbringt.
TODO

Erfolgversprechende Ansatzpunkte für weitere Forschung gibt es im Bereich der mathematischen Begründung, warum die Implementierung Vorteile erbringt, im Ausbau der Untersuchung von Kommunikation zwischen den Agenten in Verbindung mit XCS und in der Anwendung der gefundenen Ergebnisse in anderen Problemstellungen ähnlicher Natur.

1.1 Stand der Wissenschaft

Das auf Genauigkeit der *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) ist im Vergleich zum LCS gibt es vor allem innerhalb der Funktion zur Berechnung der *fitness* Werte der *classifier* Unterschiede. Während der *fitness* Wert beim einfachen LCS lediglich auf dem *reward prediction error* Wert basierte, basiert bei XCS der *fitness* Wert auf der Genauigkeit der jeweiligen Regel. Eine ausführliche Beschreibung findet sich in [But06b].

1.1.1 Beispiel für das *single step* Verfahren

Im einfachsten Fall, im sogenannten *single step* Verfahren erfolgt die Bewertung einzelner *classifier*, 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 *single step* Verfahren 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“, so soll das dritte Datenbit, also „0“ zurückgeben. Im Gegensatz zum Überwachungsszenario kann also über die Qualität eines XCS direkt bei jedem Schritt entschieden werden. In Abbildung 1.1 findet sich eine schematische Darstellung des Problems.

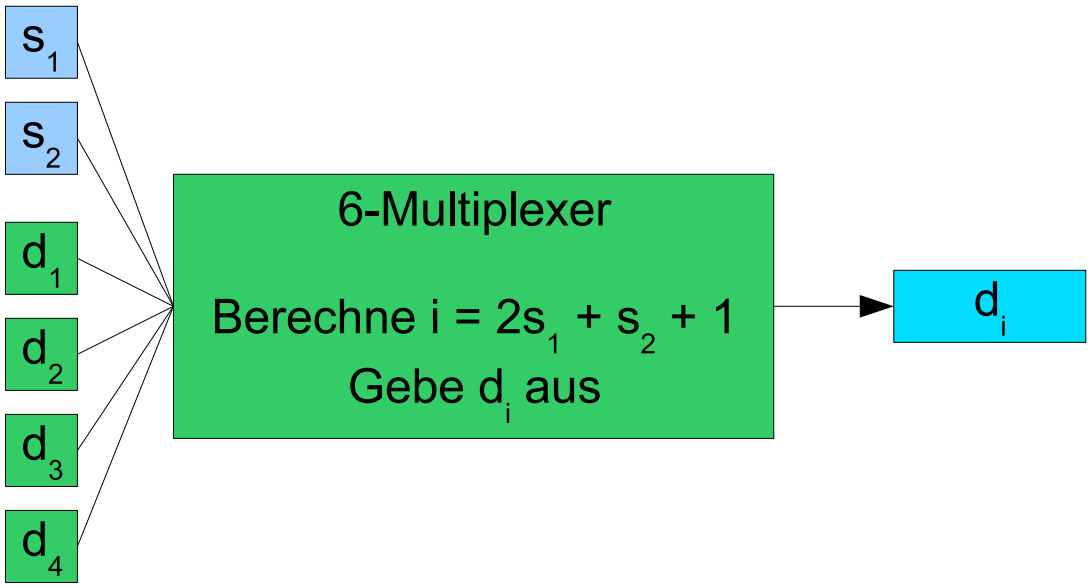

Abbildung 1.1: Schematische Darstellung des Das 6-Multiplexer Problems

1.1.2 Beispiel für das *multi step* Verfahren

Ein klassisches Beispiel für *multi step* Verfahren ist das *Maze N* Problem, bei dem durch ein Labyrinth mit dem kürzesten Weg von N Schritten gegangen werden muss. Am Ziel angekommen wird der zuletzt aktivierte *classifier* positiv bewertet und das Problem neugestartet. Bei den Wiederholungen erhält jede Regel einen Teil der Bewertung des folgenden *classifier*. Somit wird eine ganze Kette von *classifier* bewertet und sich der optimalen Wahrscheinlichkeitsverteilung angenähert, welche repräsentiert, welche der Regeln in wel-

chem Maß am Lösungsweg beteiligt sind.

Als Demonstration soll das in Abbildung 1.2 dargestellte (sehr einfache) Szenario dienen. Die zum Agenten zugehörigen *classifer* sind in Abbildung 1.3 dargestellt, wobei die 4 angrenzenden Felder für jeden *classifier* jeweils die Konfiguration der Kondition darstellt und der Pfeil die Aktion (für eine genauere Beschreibung eines *classifier* siehe Kapitel *classifier:sec*). Im ersten Durchlauf werden alle *classifier* in jedem Schritt zufällig gewählt, dann erhält *classifier* e) eine positive Bewertung. Im zweiten Durchlauf erhält dann *classifer* c) einen von *classifier* e) weitergegebene positive Bewertung und *classifier* e) auf Position 3 wird mit höherer Wahrscheinlichkeit als *classifier* f) gewählt. Das geht so lange weiter, bis sich für *classifier* b, c, e, g ein ausreichend großer Wert eingestellt hat und keine wesentlichen Veränderungen mehr auftreten.

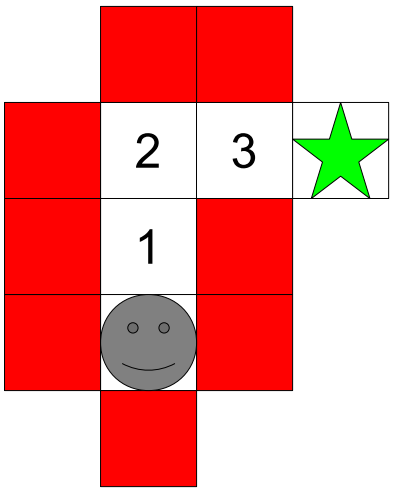

Abbildung 1.2: Einfaches Beispiel zum XCS *multi step* Verfahren

Die in dieser Arbeit verwendete Implementierung entspricht im Wesentlichen der Standardimplementation 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, die im Folgenden beschrieben werden soll.

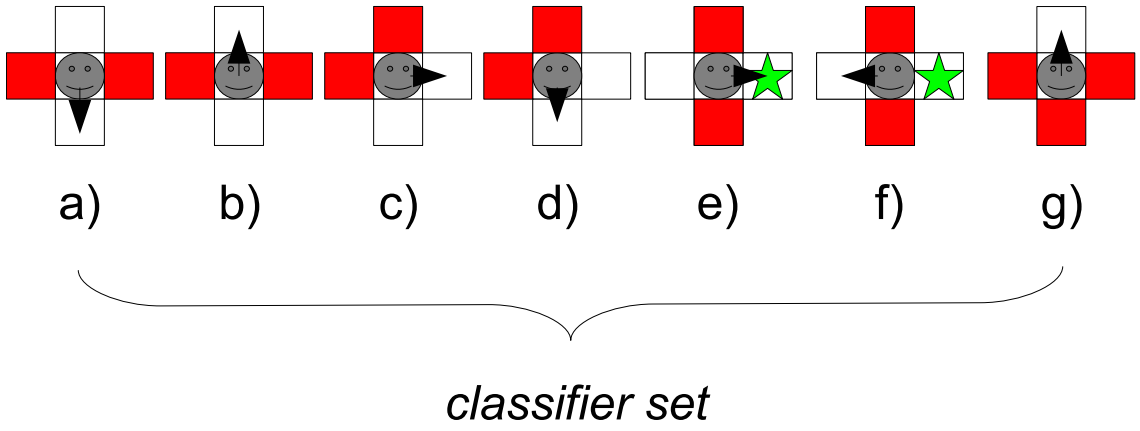

Abbildung 1.3: Vereinfachte Darstellung eines *classifier set* für das Beispiel zum XCS *multi step* Verfahren

1.1.3 Problemdefinition

Da es kein Ziel zu erreichen gibt, sondern über die Zeit hinweg ein bestimmtes Verhalten erreicht werden soll (die Überwachung des Zielobjekts), stellt sich die Frage, wie das Problem definiert werden soll. Insbesondere gibt es kein Neustart des Problems und keinen festen Start- oder Zielpunkt. Zusätzlich, durch die Bewegung der anderen Agenten und des Zielobjekts, verändert sich die Umwelt in jedem Schritt, ein Lernen durch Wiederholung gemachter Bewegungsabläufe ist deswegen deutlich schwieriger.

Die meisten Implementationen und Varianten von XCS beschäftigen sich mit derartigen Szenarien, bei denen das Ziel in einer statischen Umgebung gefunden werden muss. Häufiger Gegenstand der Untersuchung in der Literatur sind insbesondere relativ einfache Probleme 6-Multiplexer Problem und Maze1 (z.B. in [But06b] [Wil95] [Wil98]), während XCS mit Problemen größerer Schrittzahl zwischen Start und Ziel Probleme hat [Bar02] [BDE+99]. Zwar gibt es Ansätze um auch schwierigere Probleme besser in den Griff zu bekommen (z.B. Maze5, Maze6, Woods14 in [BGL05]), indem ein Gradientenabstieg in XCS implementiert wurde. Ein konkreter Bezug zu einem dynamischen Überwachungs-

szenario konnte jedoch in keiner dieser Arbeiten gefunden werden.

Bezüglich Multiagentensystemen und XCS gibt es hauptsächlich Arbeiten, die auf zentraler Steuerung bzw. *OCS* [THN+98] basieren, also im Gegensatz zum Gegenstand dieser Arbeit auf eine übergeordnete Organisationseinheit bzw. auf globale Regeln oder globalem Regeltausch zwischen den Agenten zurückgreifen.

Arbeiten bezüglich Multiagentensysteme in Verbindung mit LCS im Allgemeinen finden sich z.B. in [TB06], wobei es auch dort zentrale Agenten gibt, mit deren Hilfe die Zusammenarbeit koordiniert werden soll, während in dieser Arbeit alle Agenten dieselbe Rolle spielen sollen.

Vielversprechend war der Titel der Arbeit [LWB08], „Generation of Rule-based Adaptive Strategies for a Collaborative Virtual Simulation Environment“. Leider wird in der Arbeit nicht diskutiert, auf was sich der kollaborative Anteil bezog, da nicht mehrere Agenten benutzt worden sind. Auch konnte dort jeder einzelne Schritt mittels einer *reward* Funktion bewertet werden, da es globale Information gab. Dies vereinfacht ein solches Problem deutlich und macht einen Vergleich schwierig.

Eine weitere Arbeit in dieser Richtung [HFA02] beschreibt das „El Farol“ Bar Problem (EFBP), welches dort mit Hilfe eines Multiagenten XCS System erfolgreich gelöst wurde. Die Vergleichbarkeit ist hier auch eingeschränkt, da es sich bei dem EFBP um ein *single step* Problem handelt.

Eine der dieser Arbeit (bezüglich Multiagentensysteme) am nächsten kommende Problemstellung wurde in [ITS05] vorgestellt. Dort wurde die jeweilige Bewertung unter den

(zwei) Agenten aufgeteilt, es fand also eine Kommunikation des *reward* Werts statt. Wie das Ergebnis in Verbindung mit den Ergebnissen dieser Arbeit interpretiert werden kann, wird in Kapitel 4.4 diskutiert.

In [KM94] wurde gezeigt, dass bei der Weitergabe der Bewertung Gruppenbildung von entscheidender Wichtigkeit ist. Nach bestimmten Kriterien werden Agenten in Gruppen zusammengefasst und die Bewertung anstatt an alle, jeweils nur an die jeweiligen Gruppenmitgliedern weitergegeben. Dies bestätigen auch Tests in Kapitel 4.4, bei der sich Agenten mit ähnelnden (was das Verhalten gegenüber anderen Agenten betrifft) *classifier set* Listen in Gruppen zusammengefasst wurden und zum Teil bessere Ergebnisse erzielt werden konnten als ohne Kommunikation.

[BD03] TODO

TODO In Kapitel 4 werden dann die Implementierungen der calculateReward, calculateNextMove beschrieben TODO Limits in Long Path Learning with XCS Gamma!

TODO Anwendungen XCS!

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. Darauf aufbauend schließt Kapitel 4 an und bespricht Anpassungen wie auch 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 wesentliche Höhepunkt folgt dann in Kapitel 5 in dem alle vorgestellten Algorithmen in den vorgestellten Szenarien getestet und analysiert werden. Abschluss bildet die Zusammenfassung und der Ausblick in Kapitel 6 und im Anhang A findet sich dann noch eine Anzahl der zentralen, implementierten Quellcodes der Algorithmen, die in dieser Arbeit vorgestellt werden.

Kapitel 2

Beschreibung des Szenarios

Im Wesentlichen werden die im folgenden besprochenen 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 Überwachungsszenario wird anhand des Anteils der Zeit bewertet, in der er mit Hilfe der Agenten das Zielobjekt überwachen konnte, relativ zur Gesamtzeit (siehe Kapitel 2.6.3).

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 Zeitschritt nur auf eines der vier Nachbarfelder bewegen kann. Eine Ausnahme stellt hier Zielobjekts, welches mehrere Bewegungen in einem Zeitschritt 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, auf Kapitel 5.1 vorzubereiten, in dem anhand von Tests herausgefunden werden soll, welche der hier vorgestellten Szenarien brauchbare Ergebnisse liefern kann, um zum einen das gestellte Problem an sich, als auch die jeweils erforderlichen Eigenschaften besser verstehen zu können.

Eine separate Beschäftigung mit diesen - relativ einfachen - Szenarien war notwendig, um zum einen das eigene Simulationsprogramm zu testen und zum anderen um 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. Zwar entspricht das Standardszenario bei XCS einem Feld, einem Agenten, Hindernissen und einem Ziel, es fehlen jedoch Arbeiten, in denen 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 feste Start- und Zielpunkte) und die Messung der durchschnittlichen Qualität (meist ging es um die Anzahl der Schritte zum Ziel) gemeinsam in einem Szenario betrachtet werden.

Im Folgenden wird nun auf die einzelnen Punkte eingegangen und eine Abgrenzung zu Arbeiten in der Literatur aufgezeigt:

2.1 Dynamische, kollaborative Szenarien

Wesentliches Hauptaugenmerk der Gestaltung der Szenarien wird Kollaboration sein, d.h. die Aufgabe soll mit Hilfe mehrerer Agenten gemeinsam gelöst werden. Nach einem der Standardwerke zu Multiagentensystemen [Wei00] ist Kollaboration im Allgemeinen definiert als

„Zusammenarbeit“ und bezieht sich oft auf Kooperation auf hohem Niveau welche (die Entwicklung) ein gegenseitiges Verständnis und eine gemeinsame Sicht auf die Problemstellung, welche durch mehrere, miteinander agierende Entitäten gelöst werden soll, teilen. Manchmal werden die Begriffe Kollaboration und Kooperation auch im gleichen Sinne benutzt.

TODO keine competition coordination

cooperative state-changing rules (nicht egoistische Regeln finden) veracitz (ehrlichkeit)

Statische Umgebung: Umgebung in der nur die Agenten das Dingens verändern! rational:to behave in a waz that is suitable or een optimal for goal attainment

TODO Literatur Definition vo Kollaboration in der Literatur, Abgrenzung

Eine erfolgreiche Überwachung soll deswegen so definiert sein, dass sich ein beliebiger Agent in Überwachungsreichweite des Zielobjekts befindet. Angesichts dessen, dass diese Aufgabe auch ein einzelner Agent erfüllen kann, sofern die Geschwindigkeit des Zielobjekts kleiner oder gleich der Geschwindigkeit des Agenten ist, sollen in späteren Tests (insbesondere in Kapitel 5 beim Vergleich unterschiedlicher XCS Varianten und im Kapitel 3.5 beim Vergleich unterschiedlicher XCS Parameter) unterschiedliche Geschwindigkeiten getestet werden.

Bewegt sich das Zielobjekt zu schnell, werden die Agenten Schwierigkeiten haben, einen Bezug zwischen Sensordaten und eigener 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.

Die Szenarien fallen alle unter die Kategorie „dynamisch“. Darunter soll in diesem Zusammenhang verstanden werden, dass es kein festes Ziel gibt, das erreicht werden soll oder kann, das Zielobjekt befindet sich in stetiger Bewegung, wie auch sich andere Agenten in Bewegung befinden können.

Dies ist ein wesentlicher Gesichtspunkt, dass diese Arbeit von vielen anderen unterscheidet. Gegenstand der Untersuchung in der Literatur sind eher statische Probleme, wie z.B. das 6-Multiplexer Problem und Maze1 [But06b] bzw. Maze5, Maze6, Woods14 [BGL05].

El Fasor, Soccer

oder Probleme bei denen die Agenten globale Information besitzen

TODO

Eine nähere Diskussion zur Literatur folgt in Kapitel 3.

2.2 Konfigurationen des Torus

Getestet wurden eine Reihe von Szenarien (in Verbindung mit unterschiedlichen Werten für die Anzahl der Agenten, Größe des Torus und Art und Geschwindigkeit des Zielobjekts). Wesentliches Merkmal jedes Szenarios ist die Menge und die Verteilung der Hindernisse.

In den folgenden Abbildungen repräsentieren rote Felder jeweils Hindernisse, weiße Felder jeweils Agenten und das grüne Feld jeweils das Zielobjekt. Außerdem sind die Sicht- und Überwachungsreichweiten aus Kapitel 2.3.1 dargestellt. Sie haben jeweils eine Gestalt ähnlich der eines viertel Abschnitts einer Kreisfläche mit dem jeweiligen Agenten im Mittelpunkt. In den Abbildungen soll der Bereich, der durch die Überwachungsreich-

weite abgedeckt wird, grau dargestellt werden und der restliche Bereich, der zusätzlich noch durch die Sichtweite abgedeckt wird, blau.

2.2.1 Leeres Szenario ohne Hindernisse

In Abbildung (2.1) ist ein Szenario ohne Hindernisse und mit zufälliger Verteilung der Agenten und zufälliger Position des Zielobjekts dargestellt. Im leeren Szenario soll das Verhalten der Agenten in einem Torus ohne Hindernisse untersucht werden. Eine Untersuchung dieses Szenario erlaubt zum einen die Vereinfachung, dass Hindernisse (beispielsweise bei den in Kapitel 2.3 besprochenen Sensoren) nicht beachtet werden müssen und zum anderen, dass Sicht nicht versperrt wird und kein Agent gegen Hindernisse laufen (und stehenbleiben) muss.

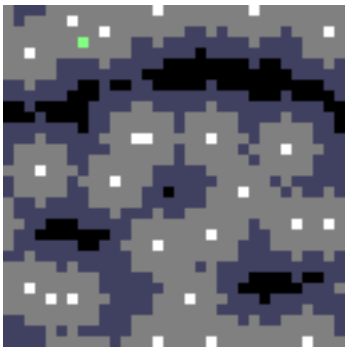

Abbildung 2.1: „Leeres Szenario“ ohne Hindernisse

2.2.2 Szenario mit zufällig verteilten Hindernissen

Für das Szenario mit zufällig verteilten Hindernissen sind zwei Parameter prägend. So bestimmt der erste Parameter (Hindernissanteil λ_h) den Prozentsatz an Hindernissen an der Gesamtzahl der Felder des Torus und der zweite Parameter (Verknüpfungsfaktor λ_p) den Grad inwieweit die Hindernisse zusammenhängen.

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 sollen nur geringe Werte bis 0,4 betrachtet werden, wobei später in Tests sich auf Werte bis 0,2 beschränkt wird, da bei großen Hindernissanteil die lokalen Entscheidungen einzelner Agenten zu wichtig werden, da das Zielobjekt sich eher in einem kleinen Bereich aufhält.

In Abbildung (2.2), Abbildung (2.3), Abbildung (2.4) und Abbildung (2.5) werden Beispiele für zufällige Szenarien mit $\lambda_h = 0{,}05$, 0,1, 0,2 bzw. 0,4 und $\lambda_p = 0{,}01$, 0,5 bzw. 0,99 dargestellt.

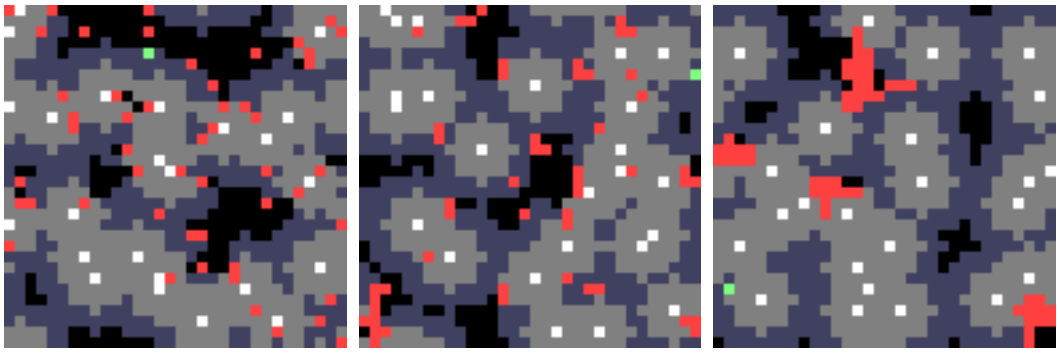

Abbildung 2.2: Szenario mit zufällig verteilten Hindernissen mit Hindernissanteil $\lambda_h = 0{,}05$ und Verknüpfungsfaktor $\lambda_p = 0{,}01$, 0,5 bzw. 0,99.

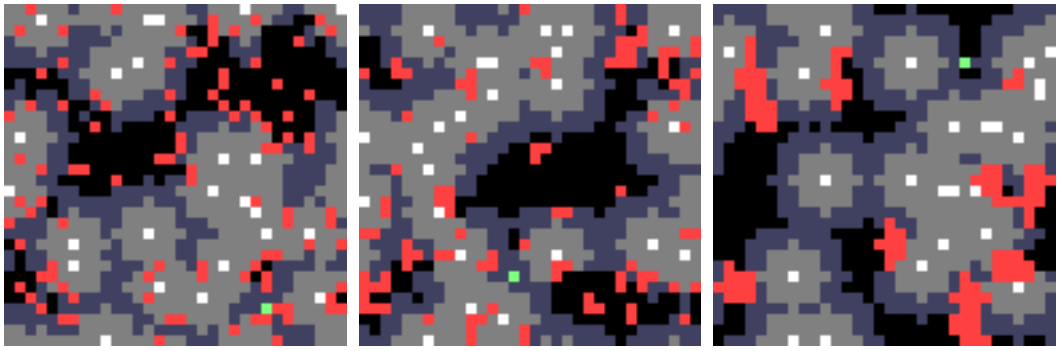

Abbildung 2.3: Szenario mit zufällig verteilten Hindernissen mit Hindernissanteil $\lambda_h = 0{,}1$ und Verknüpfungsfaktor $\lambda_p = 0{,}01$, 0,5 bzw. 0,99.

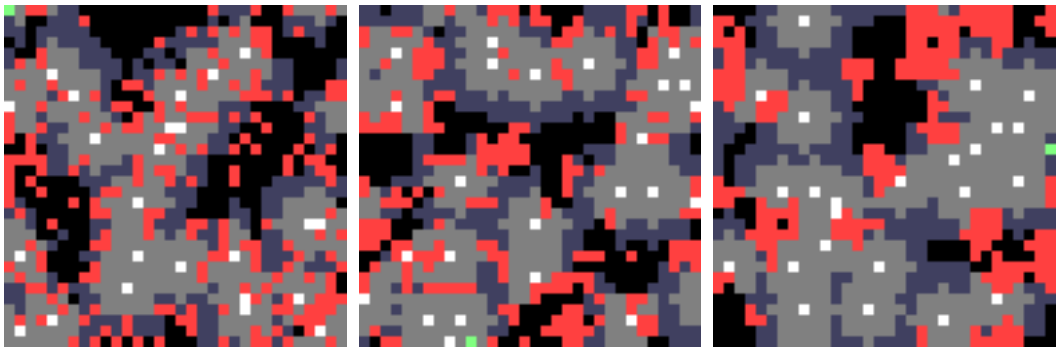

Abbildung 2.4: Szenario mit zufällig verteilten Hindernissen mit Hindernissanteil $\lambda_h = 0{,}2$ und Verknüpfungsfaktor $\lambda_p = 0{,}01$, 0,5 bzw. 0,99.

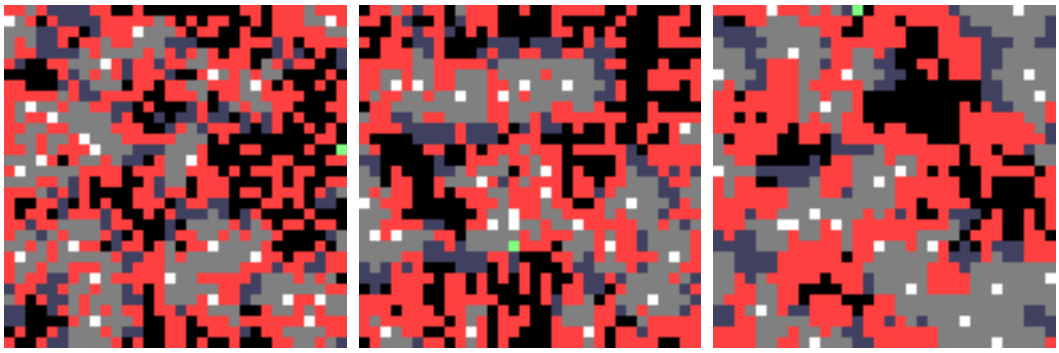

Abbildung 2.5: Szenario mit zufällig verteilten Hindernissen mit Hindernissanteil $\lambda_h = 0{,}4$ und Verknüpfungsfaktor $\lambda_p = 0{,}01$, 0,5 bzw. 0,99.

2.2.3 Säulenszenario

In diesem Szenario werden regelmäßig, mit jeweils 7 Feldern Zwischenraum zueinander, Hindernisse auf dem Torus verteilt. Tragende Idee dieses Szenarios ist es, dass die Agenten eine kleine Orientierungshilfe besitzen sollen, aber gleichzeitig möglichst wenig Hindernisse verteilt werden. Das Zielobjekt startet an zufälliger Position, die Agenten starten mit möglichst großem Abstand zum Zielobjekt.
Abbildung 2.6 zeigt ein Beispiel für den Startzustand eines solchen Szenarios, bei der das Zielobjekt sich in der Mitte und die Agenten am Rand befinden.

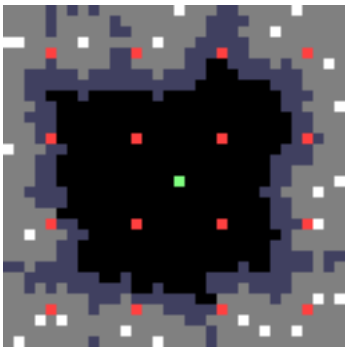

Abbildung 2.6: Startzustand des Säulen Szenarios mit regelmäßig angeordneten Hindernissen und zufälliger Verteilung von Agenten mit möglichst großem Abstand zum Zielobjekt

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 (7 Felder Zwischenraum) befindet sich eine vertikale Reihe von Hindernissen mit Öffnungen von 4 Feldern Breite abwechselnd im oberen Viertel und dem unteren Viertel.

Idee dieses Szenarios ist es, 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 besprochenen Tests in Kapitel 5.7.4 werden zeigen, dass dieses Szenario besonders schwierig für sich zufällig bewegende Agenten und Agenten mit einfacher Heuristik ist und wie Kommunikation hier von Vorteil sein kann.
Abbildung 2.7 zeigt die Startkonfiguration des Szenarios.

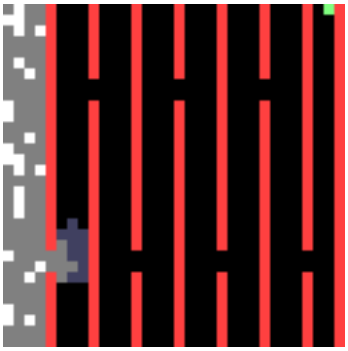

Abbildung 2.7: Schwieriges Szenario mit fester, wallartiger Verteilung von Hindernissen in regelmäßigen Abständen und mit Öffnungen, mit den Agenten mit zufälligem Startpunkt am linken Rand und mit dem Zielobjekt mit festem Startpunkt rechts oben

2.3 Eigenschaften der Objekte

Jeder Agent bzw. das Zielobjekt besitzt eine Anzahl visueller, binärer Sensoren mit begrenzter Reichweite. 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, andere Objekte blockieren die Sicht und Sichtlinien werden durch einen einfachen Bresenham-Algorithmus [Bre65] bestimmt.
Zwei Sensoren, die in dieselbe Richtung ausgerichtet sind und den selben Typ von Objekt erkennen, werden in diesem Zusammenhang ein Sensordatenpaar genannt (siehe Kapitel 2.3.2). Alle Sensoren, die nur gemeinsam haben, dass sie den selben Typ von Objekt erkennen, werden in einer Gruppe zusammengefasst und der Aufbau eines ganzen, aus solchen Gruppen bestehenden Sensordatensatzes soll in Kapitel 2.3.3 besprochen werden. Die Eigenschaften der Agenten und des Zielobjekts selbst sollen dann in Kapitel 2.3.4 beschrieben werden.

2.3.1 Sichtbarkeit von Objekten

Der Parameter *sight range* bzw. *reward range* bestimmt, 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 2.6.3) und wird immer kleiner als der *sight range* Wert gewählt. Über die Sensoren kann ein Agent bzw. das Zielobjekt feststellen, ob sich Objekte in welcher der beiden Reichweiten befinden. Falls nicht anders angegeben sollen jeweils *sight range* auf 5 und *reward range* auf 2 gesetzt werden.

2.3.2 Aufbau eines Sensordatenpaars

Ein Datenpaar besteht aus zwei Sensoren, die den selben Typ von Objekt erkennen, in dieselbe Richtung ausgerichtet sind und sich nur in ihrer Sichtweite unterscheiden, wodurch der Agent rudimentär die Entfernung zu anderen Objekten feststellen kann. 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 also eine Teilmenge des sichtbaren Bereichs. In Abbildung 2.8 sind alle Sichtreichweiten (heller und dunkler Bereich) und Überwachungsreichweiten (heller Bereich) für die einzelnen Richtungen dargestellt.

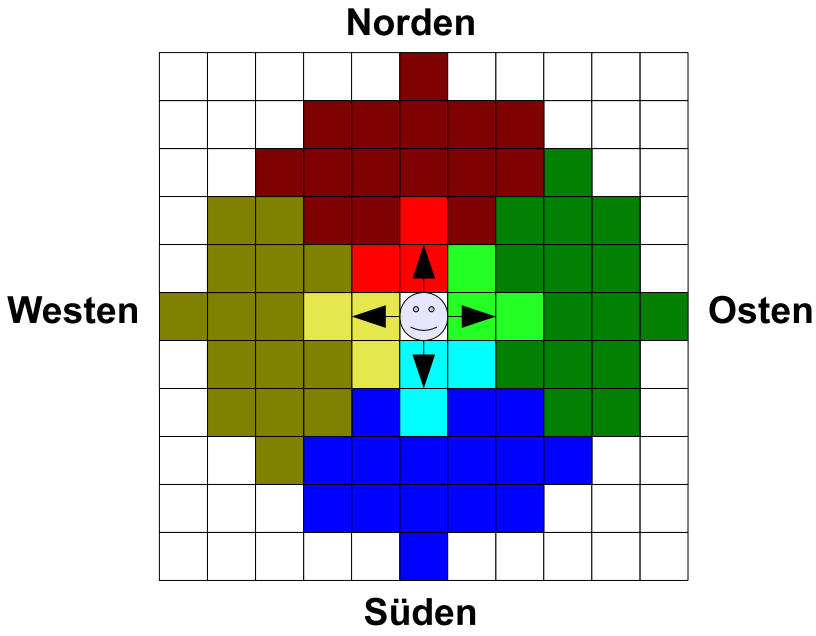

Abbildung 2.8: Sicht- (5,0, dunkler Bereich) und Überwachungsreichweite (2,0, heller Bereich) eines Agenten, jeweils für die einzelnen Richtungen

Anzumerken sei hier, dass wegen der gewählten Werte für beide Reichweiten ein Sensordatenpaar (01) nicht auftreten kann, da ein Objekt nicht gleichzeitig näher als 2,0 und

weiter als 5,0 entfernt sein kann.

Sei $r(O1, O2)$ die Distanz zwischen dem Objekt, das die Sensordaten erfasst und dem nächstliegenden Objekt 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 8 Sensoren zu jeweils einer Gruppe zusammengefasst, welche wiederum jeweils in 4 Richtungen mit jeweils einem Sensorenpaar aufgeteilt ist. Abbildung 2.9 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 2.9: Darstellung des Sensordatensatzes, eingeteilt in mehrere Gruppen und Sensorpaare

Seien beispielsweise im Norden außerhalb der Überwachungsreichweite aber in Sichtweite das Zielobjekt, im Süden ein oder mehrere Agenten in Überwachungsreichweite und im Westen und Osten sich ebenfalls in Überwachungsreichweite des Agenten befindliche Hindernisse, dann ergibt sich ein Sensordatensatz $s_{Beispiel}$ wie in Abbildung 2.10 darge-

stellt.

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

Abbildung 2.10: 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 Richtungen (Norden, Osten, Süden, Westen) 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 Feld betrachtet werden soll, werden alle Agenten werden nacheinander in der Art abgearbeitet, 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 weiter bestraft. Eine detaillierte Beschreibung der Bewegung im Kontext anderer Agenten und Programmteile wird in Kapitel 2.6.4 gegeben.

Weitere Fähigkeiten eines Agenten betreffen die Kommunikation, bis Kapitel 4.4 soll jedoch nur der Fall ohne Kommunikation betrachtet werden, d.h. die Agenten können untereinander keine Informationen austauschen und müssen sich alleine auf ihre Sensordaten verlassen.

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 dieser die Aufenthaltswahrscheinlichkeiten des Zielobjekts unter Einbeziehung des Zustands des letzten Zeitschritts bestimmt. Springt das Zielobjekt beispielsweise auf zufällige auf dem Torus (siehe Kapitel 2.5.1), dann existiert keine Verbindung zwischen den Positionen des Zielobjekts zweier aufeinanderfolgender Zeiteinheiten und Lernen wird sehr schwierig, was später in Kapitel 5.1.1 gezeigt wird. Primär soll diese Form der Bewegung 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 soll noch das sich intelligent verhaltende Zielobjekt besprochen werden, was ebenfalls ein zentraler Punkt der späteren Analyse (in Kapitel 5.2.2) sein soll. Am Ende sollen dann zwei Sonderfälle erwähnt werden, zum einen ein Zielobjekt, das nur in dieselbe Richtung läuft (Kapitel 2.5.5), welches in Kapitel 4.4 zur Untersuchung des schwierigen Szenarios benutzt werden soll.

2.4 Grundsätzliche Algorithmen der Agenten

Neben denjenigen Algorithmen, die auf XCS basieren und in Kapitel 3 besprochen werden, sollen hier einige, auf einfachen Heuristiken basierende, Algorithmen vorgestellt werden, um die Qualität der anderen Algorithmen besser 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. ihre Regeln während eines Laufs nicht anpassen.

Die in Kapitel A.1 erwähnte und dort aufgerufene Funktion *calculateReward()* soll für die hier aufgelisteten Algorithmen also jeweils der leeren Funktion entsprechen. Im Folgenden sollen also insbesondere die Implementierungen der jeweiligen *calculateNextMove()* Funktion vorgestellt werden.

2.4.1 Algorithmus mit zufälliger Bewegung

Bei diesem Algorithmus wird in jedem Schritt wird eine zufällige Aktion ausgeführt. Abbildung 2.11 zeigt eine Beispielsituation, bei der der Agent jegliche Sensordaten (die 4 Agenten und das Zielobjekt, der als Stern dargestellt ist) ignoriert und eine Aktion zufällig auswählen wird.
Programm A.6 zeigt den zugehörigen Quelltext.

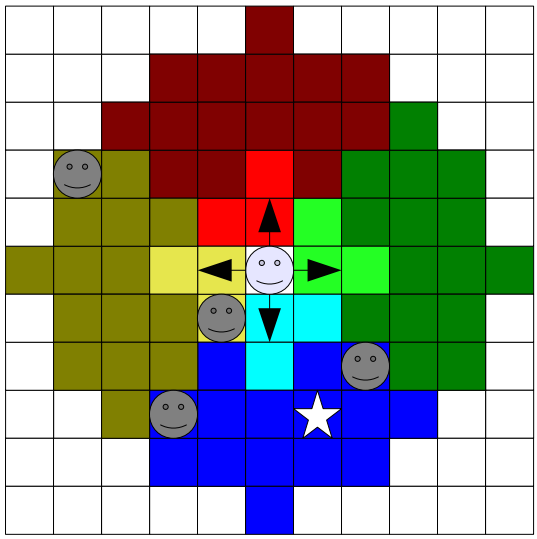

Abbildung 2.11: Agent bewegt sich in eine zufällige Richtung (oder bleibt stehen).

2.4.2 Algorithmus mit einfacher Heuristik

Ist das Zielobjekt in Sichtweite, bewegt sich ein Agent mit dieser Heuristik auf das Zielobjekt zu, ist es nicht in Sichtweite, führt er eine zufällige Aktion aus. Abbildung 2.12 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 zubewegen möchte.

Programm A.7 zeigt den zugehörigen Quelltext.

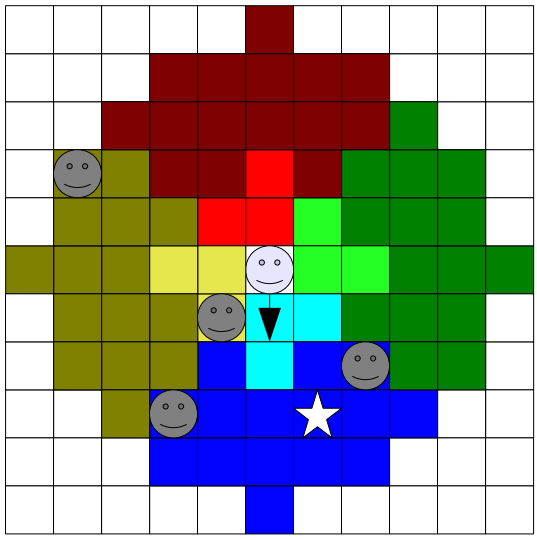

Abbildung 2.12: Agent mit einfacher Heuristik: Sofern es sichtbar ist bewegt sich der Agent auf das Zielobjekt zu.

2.4.3 Algorithmus mit intelligenter Heuristik

Ist das Zielobjekt in Sicht, verhält sich diese Heuristik wie die einfache Heuristik. Ist das Zielobjekt dagegen nicht in Sicht, wird versucht, anderen Agenten auszuweichen, um ein möglichst breit gestreutes Netz aus Agenten aufzubauen. In der Implementation heißt das, dass unter allen Richtungen, in denen kein anderer Agent gesichtet wurde, eine Richtung zufällig ausgewählt wird und falls alle Richtungen belegt (oder alle frei) sind, wird aus

allen Richtungen eine zufällig ausgewählt wird. In Abbildung 2.13 ist das Zielobjekt nicht im Sichtbereich des Agenten und dieser wählt deswegen eine Richtung, in der die Sensoren keine Agenten anzeigen, in diesem Fall Norden.
Programm A.8 zeigt den zugehörigen Quelltext.

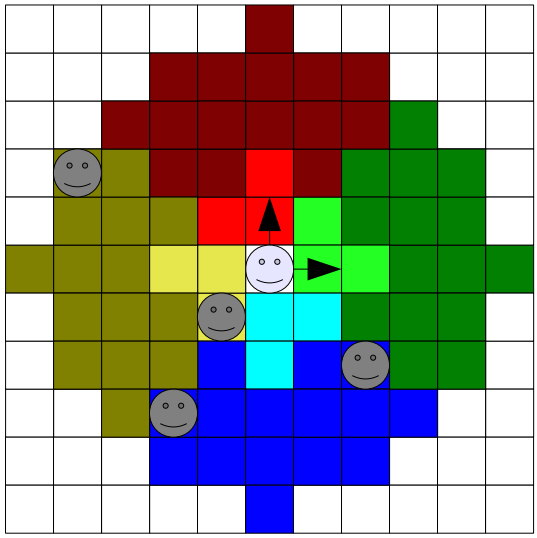

Abbildung 2.13: Agent mit intelligenter Heuristik: Falls das Zielobjekt nicht sichtbar ist, bewegt sich der Agent von anderen Agenten weg.

2.5 Typen von Zielobjekten

Im Wesentlichen entspricht ein Zielobjekt einem Agenten, d.h. das Zielobjekt kann sich bewegen und besitzt Sensoren. Außerdem kann sich das Zielobjekt in einem Schritt u.U. um mehr als ein Feld bewegen, was durch die durch das Szenario festgelegte Geschwindigkeit des Zielobjekts bestimmt ist. Der Wert der Geschwindigkeit kann auch gebrochene Werte annehmen, wobei in diesem Fall der gebrochene Rest dann die Wahrscheinlichkeit angibt, einen weiteren Schritt durchzuführen. Beispielsweise würde 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 2.6.4, bei der

Reihenfolge der Ausführung der Aktionen der Objekte.

Falls dem Algorithmus kein freies Feld zur Verfügung steht, ist es allen Bewegungen des Zielobjektes gemeinsam, dass per Zufall ein freies Feld in der Nähe ausgewählt und das Zielobjekt dorthin springt, was einem Neustart des Problems ähnlich ist. Dies ist notwendig, um eine Verfälschung des Ergebnisses zu verhindern, welche eintreten kann, wenn einer oder mehrere Agenten (eventuell zusammen mit anderen Hindernissen) alle vier Bewegungsrichtungen des Zielobjekts dauerhaft zu blockieren. Zu beachten ist hier, dass auch der Sprung selbst eine Verfälschung darstellt, insbesondere wenn in einem Durchlauf viele Sprünge durchgeführt werden. Falls dies passiert 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 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.
Jegliche Anpassung an die Bewegung des Zielobjekts ist hier wenig hilfreich, ein Agent kann nicht einmal davon ausgehen, 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 entfernt zuerst alle Richtungen, in denen sich direkt angrenzend ein Hindernis befindet. Diese Erweiterung der Fähigkeiten der Sensoren wurde gewählt, damit das Zielobjekt nicht in Hindernissen längere Zeit steckenbleibt.
Anschließend wird die Richtung entfernt, die der im letzten Schritt gewählten entgegengesetzt ist. Von den verbleibenden (bis zu) drei Richtungen wird schließlich eine zufällig ausgewählt. Sind alle drei Richtungen versperrt, wird in die entgegengesetzte Richtung zurückgegangen.
In Abbildung 2.14 sind alle Felder grau markiert, die das Zielobjekt innerhalb von zwei Schritten erreichen kann, nachdem er sich einmal nach Norden bewegt hat.

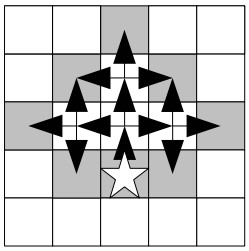

Abbildung 2.14: Zielobjekt macht pro Schritt maximal eine Richtungsänderung

2.5.4 Typ „Intelligentes Verhalten“

Hier versucht das Zielobjekt bei der Auswahl der Aktion möglichst die Aktion zu wählen, bei der es außerhalb der Sichtweite der Agenten bleibt. Dazu werden alle Richtungen ge-

strichen, in denen ein Agent sich innerhalb der Überwachungsreichweite befindet. Außerdem werden von den verbleibenden Richtungen mit 50% diejenigen Richtungen gestrichen, in denen sich ein Agent in Sichtweite befindet. Sind alle Richtungen gestrichen worden, bewegt sich das Zielobjekt zufällig. Sind alle Richtungen blockiert, springt es wie in den anderen Varianten auch auf ein zufälliges Feld in der Nähe.

In Abbildung 2.15 wird die Richtung Süden gestrichen, da sich dort ein Agent in Überwachungsreichweite befindet. Die Richtungen Westen und Norden werden jeweils mit Wahrscheinlichkeit 50% gestrichen, da sich dort Agenten in Sichtweite befinden. Nur Richtung Osten wird als Möglichkeit sicher übrig bleiben.

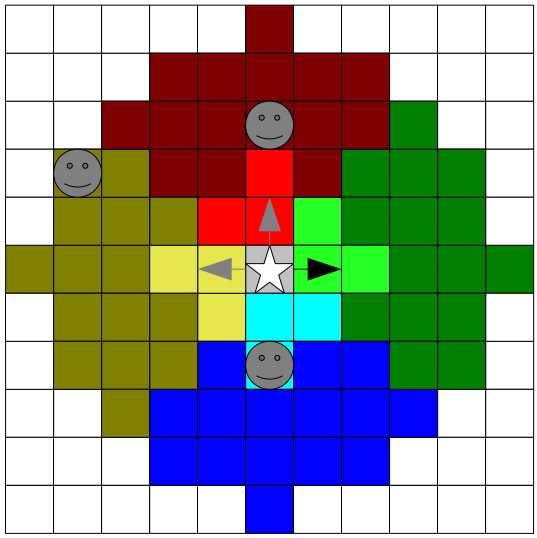

Abbildung 2.15: Sich intelligent verhaltendes Zielobjekt weicht Agenten aus.

2.5.5 Typ „Beibehaltung der Richtung“

Ein Zielobjekt dieses Typs versucht, immer Richtung Norden zu gehen. Ist das Zielfeld blockiert, wählt es ein zufälliges, angrenzendes, freies Feld im Westen oder Osten. An-

zumerken ist, dass dies zusätzliche Fähigkeiten darstellen, d.h. das Zielobjekt kann feststellen, ob sich direkt angrenzend ein Hindernis im Norden befindet, während normale Agenten, was die Distanz betrifft, keine Informationen darüber besitzen können.

In Abbildung 2.16 sind drei Situationen dargestellt, zum einen ein wiederholtes Hin- und Herlaufen unter den Hindernissen, der Weg links um die Hindernisse herum und der Weg rechts um die Hindernisse herum.

Diese Art von Zielobjekt soll insbesondere im schwierigen Szenario benutzt werden um den Bereich, den das Zielobjekt überquert, möglichst gering zu halten, aber gleichzeitig das Zielobjekt auch nicht an einer Stelle stehen zu lassen.

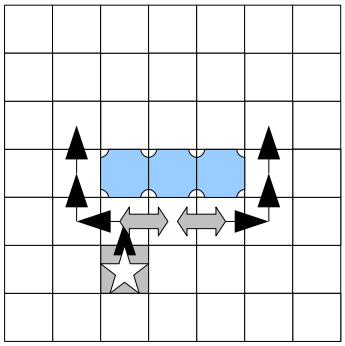

Abbildung 2.16: Bewegungsform „Beibehaltung der Richtung“: Zielobjekt bewegt sich, wenn möglich, immer nach Norden.

2.5.6 Typ „Lernendes Zielobjekt“

Ein besonderer Typ ist dieses Zielobjekt. Es kann mit Hilfe einer der in Kapitel 4 besprochenen Algorithmen lernen. Wesentlicher Unterschied zu lernenden Agenten ist, dass hier das Zielobjekt eine Aktion als positiv vermerkt, wenn sich keine Agenten in Überwachungsreichweite befinden. Eine genaue Beschreibung folgt in Kapitel 4.3.6, hier soll die

Idee nur der Vollständigkeit halber erwähnt werden.

2.6 Simulation und erfasste Statistiken

Beim Testen selbst war es durch die Verwendung der in Kapitel 2.6.3 erwähnten (und im Simulator berechneten) Halbzeitqualitäten sehr einfach, festzustellen, ob ein Algorithmus noch Potential hatte, d.h. ob eine Erhöhung der Schrittzahl die Qualität weiter steigern würde. Da (im Gegensatz zu den später in Kapitel 4 erwähnten lernenden Algorithmen) keiner der hier vorgestellten Algorithmen lernt und somit statische Regeln besitzt, ist es nicht notwendig, die Qualitäten der Algorithmen bei verschiedener Anzahl von Zeitschritten zu betrachten und zu vergleichen. Die Zahl der Zeitschritte wird somit für alle Tests, soweit nicht anders angegeben, standardmäßig auf 500 festgesetzt. Wie in Kapitel 5.1 festgestellt wird, hat die Agentenzahl selbst wenig Einfluss, die Verhältnisse zwischen den Agententypen ist ähnlich. Deshalb werden, soweit nicht anders angegeben, hier immer 8 Agenten getestet. Außerdem sollen in den Statistiken die Werte jeweils über einen Lauf von 10 Experimenten mit jeweils 10 Probleminstanzen (siehe Kapitel 2.6.1) ermittelt und gemittelt werden. 10 Experimente reichen aus um ausreichend akkurate Ergebnisse zu erhalten, wie man in Abbildung 2.17 sehen kann. Dabei wurde jeweils die Varianz von 10 Durchläufen mit aufsteigender Anzahl von Experimenten berechnet. Jeder der 10 Durchläufe wurde mit einem zufälligen *random seed* Wert gestartet.

Im Folgenden sollen nun allgemeine Eigenschaften der Simulation erläutert werden.

2.6.1 Definition einer Probleminstanz

Eine einzelne Probleminstanz entspricht hier einem Torus mit einer bestimmten Anfangsbelegung mit bestimmten Objekten und bestimmten Parametern zur Sichtbarkeit, auf dem über die erwähnte Anzahl von Schritten die Simulation abläuft. Die Anfangsbelegung des Torus ist über einen *random seed* Wert bestimmt, wobei bei jeder Probleminstanz mit einem neuen Wert initialisiert wird, der sich aus der Nummer des Experiments

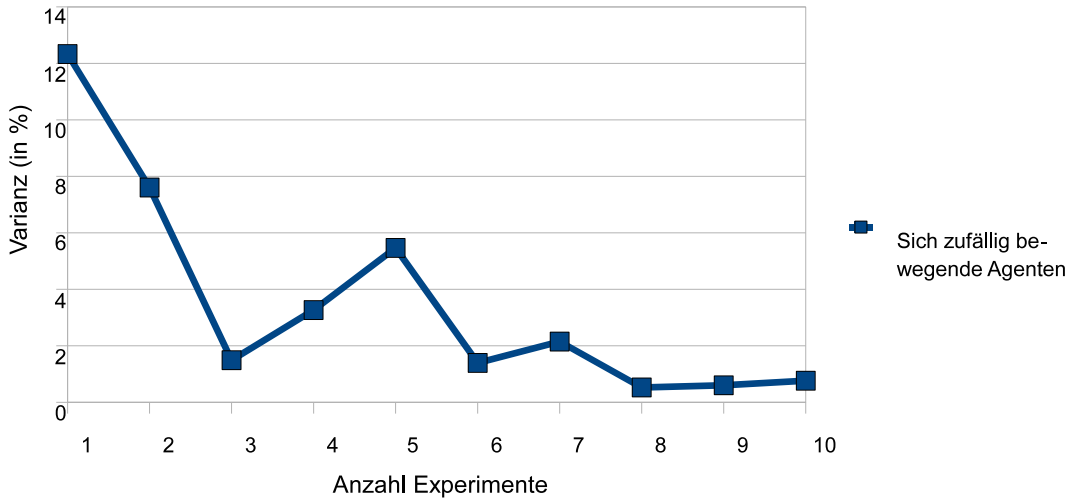

Abbildung 2.17: Varianz der Testergebnisse bei unterschiedlicher Anzahl von Experimenten (Säulenszenario, Zielobjekt mit einfacher Richtungsänderung und Geschwindigkeit 2, Agenten mit zufälliger Bewegung)

und der Nummer des Problems berechnet. Die Probleminstanzen sind also untereinander unterschiedlich, jedoch mit anderen Testdurchläufen mit einer anderen Konfiguration vergleichbar. Soweit nicht anders angegeben, sollen hier Probleminstanzen der Größe 16x16 Felder betrachtet werden, insbesondere beziehen sich die Ergebnisse der Tests auf diesen Fall. In den jeweiligen Tabellen angegebenen Werte sind auf zwei Stellen gerundet.

Während eines Testlaufs werden eine ganze Reihe von statistischen Merkmalen erfasst. Wesentliches Merkmal zum Vergleich der Algorithmen ist der Wert der Qualität (siehe Kapitel 2.6.3), weitere Merkmale dienen zur Erklärung, warum z.B. ein Algorithmus bei einem Durchlauf schlechte Ergebnisse lieferte, bzw. dienten zum Testen und Finden von Fehlern oder Schwächen des Simulationsprogramms.

Im Einzelnen sind hier zu nennen:

1. Anteil Sprünge des Zielobjekts (siehe Kapitel 2.5), Durchläufe mit hohen Werte müssten verworfen werden,
2. Anteil blockierter Bewegungen der Agenten,
3. Halbzeitqualität (siehe Kapitel 2.6.3), größere Unterschiede zur ermittelten Qualität deuten darauf hin, dass sich der Algorithmus noch nicht stabilisiert hat und das Szenario mit höherer Schrittzahl erneut durchgeführt werden sollte,
4. Abdeckung sowie
5. Varianz der individuellen Punkte, ungefähres Maß, inwieweit einzelne Agenten an der Gesamtqualität beteiligt waren.

2.6.2 Abdeckung

Die theoretisch maximal mögliche Anzahl an Felder, die die Agenten innerhalb ihrer Überwachungsreichweite zu einem Zeitpunkt haben können, entspricht der Zahl der Agenten multipliziert mit der Zahl der Felder, die ein Agent in seiner Übertragungsreichweite 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, erhält man die Abdeckung, die die Agenten momentan erreichen.

2.6.3 Qualität eines Algorithmus

Die Qualität eines Algorithmus zu einem Problem wird anhand des Anteils der Zeit berechnet, die er das Zielobjekt während des Problems überwachen (d.h. das Zielobjekt innerhalb einer Distanz von höchstens *reward range* halten) konnte, relativ zur Gesamtzeit.

Die Qualität eines Algorithmus zu einer Anzahl von Problemen (also einem Experiment) wird Anhand des Gesamtanteil der Zeit berechnet, die er das Zielobjekt während aller Probleme überwachen konnte, relativ zur Gesamtzeit aller Probleme.
Die Qualität eines Algorithmus entspricht dem Durchschnitt der Qualitäten des Algorithmus mehrerer Experimente.
Die Halbzeitqualität eines Algorithmus zu einem Problem entspricht dem Anteil der Zeit, die der Algorithmus das Zielobjekt während jeweils der zweiten Hälfte des Problems überwachen konnte, relativ zur halben Gesamtzeit.
Die Halbzeitqualität eines Algorithmus zu einer Anzahl von Problemen entspricht dem Anteil der Zeit, die der Algorithmus das Zielobjekt während jeweils der zweiten Hälfte des Problems überwachen konnte, relativ zur halben Gesamtzeit aller Probleme.
Die Halbzeitqualität eines Algorithmus entspricht dem Durchschnitt aller Halbzeitqualitäten des Algorithmus mehrerer Experimente.
Ein Vergleich der Qualität mit der Halbzeitqualität eines Algorithmus ermöglicht einen Einblick, wie gut sich der Algorithmus verhält, nachdem er sich auf das Problem bereits eine Zeit lang einstellen konnte.

2.6.4 Ablauf der Simulation

Die Simulation selbst läuft in ineinander geschachtelten Schleifen ab. Jede Konfiguration (in den abgedruckten Programmen jeweils über die globale Variable *Configuration* angesprochen) 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 das eigentliche Problem mit der Funktion *doOneMultiStepPro-*

blem() 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, soll im Folgenden geklärt werden. 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.

Bei der Berechnung eines einzelnen Problems 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 u.U. 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 Sensordaten ermittelt, ausgewertet, Agenten bewegt, intern sich selbst bewertet und global die Qualität gemessen wird.

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 Voraussetzungen 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 aus-

geführt.

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

Bei der ersten Möglichkeit haben alle Agenten die Sensordaten vom Beginn der Zeiteinheit, während bei der zweiten Möglichkeit später verarbeitete Agenten bereits die Aktionen der bereits berechneten Agenten miteinbeziehen können. Umgekehrt können dann frühere Agenten bessere Positionen früher besetzen. Da aufgrund der primitiven Sensoren nicht davon auszugehen ist, dass Agenten beginnende Bewegungen (und somit deren jeweilige Zielposition) anderer Agenten einbeziehen können, soll jeder Agent von den Sensorinformationen zu Beginn der Zeiteinheit ausgehen.

Wenn sich mehrere Agenten auf dasselbe Feld bewegen wollen, dann spielt die Reihenfolge der Ausführung der Aktionen eine Rolle. Wird die Liste der Agenten einfach linear abgearbeitet, können Agenten mit niedriger Position in der Liste die Aktion auf Basis jüngerer Sensordaten fällen. Dies kann dazu führen, dass Aktionen von Agenten mit höherer Position in der Liste eher fehlschlagen, da das als frei angenommene Feld nun bereits besetzt ist. Da es keinen Grund gibt, Agenten mit niedrigerer Position zu bevorteilen, werden die Aktionen der Agenten in zufälliger Reihenfolge abgearbeitet.

Bezüglich der Bewegung ergibt sich hierbei eine weitere Frage, nämlich wie unterschiedliche Bewegungsgeschwindigkeiten behandelt werden sollen, da alle Agenten eine Einheitsgeschwindigkeit von einem Feld pro Zeiteinheit haben, während sich das Zielobjekt je nach Szenario gleich eine ganze Anzahl von Feldern bewegen kann (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.

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

2.6.5 Messung der Qualität

Bei der Betrachtung der oben betrachteten Reihenfolge stellt sich die Frage, wann man die Qualität des Algorithmus messen sollte. Die Antwort hängt davon ab, was man denn nun eigentlich erreichen möchte. Der naheliegendste Messzeitpunkt ist, nachdem sich alle Agenten bewegt haben. Da die Agenten und das Zielobjekt in einem Durchlauf gemeinsam nacheinander bewegt werden, stellt sich die Frage nicht, ob womöglich vor der Bewegung des Zielobjekts die Qualität gemessen werden sollen. Eine Messung nach der Bewegung des Zielobjekts würde diesem erlauben, sich vor jeder Messung optimal zu positionieren, was in einer geringeren Qualität für den Algorithmus resultiert, da sich das Zielobjekt aus der Überwachungsreichweite anderer Agenten hinausbewegen kann. Letztlich ist es eine Frage der Problemstellung, denn eine Messung nach Bewegung des Zielobjekts bedeutet letztlich, dass ein Agent einen gerade aus seiner Überwachungsreichweite heraus laufenden Zielobjekts in diesem Schritt nicht mehr überwachen kann.

Da ein wesentlicher Bestandteil die Kollaboration (und somit die Abdeckung des Torus anstatt dem dauernden Verfolgen des Zielobjekts) sein soll, soll ein Bewertungskriterium sein, inwieweit der Einfluss des Zielobjekts minimiert werden soll. Auch findet, wenn man vom realistischen Fall ausgeht, die Bewegung des Zielobjekts gleichzeitig mit allen anderen Agenten statt. Die Qualität wird somit nach der Bewegung des Zielobjekts gemessen.

Die Überlegung unterstreicht auch nochmal, dass es besser ist, das Zielobjekt insgesamt wie einen normalen (aber sich mehrmals bewegenden) Agenten zu behandeln.

2.6.6 Reihenfolge der Ermittlung des *base reward*

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. Bei den bisher vorgestellten Agenten in Kapitel 2.4 haben die Agenten nicht gelernt, d.h. es gab keine Rückkopplung zwischen erfassten Sensordaten und den Regeln, nach denen die nächsten Aktionen entschieden werden. 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* ergeben soll.
Wesentliche Punkte hierbei sind, dass der Algorithmus sich anhand der Sensordaten selbst bewertet und pro Zeitschritt die Sensordaten nur einmal erhoben werden. Letzteres folgt aus der Auslegung von XCS, der in der Standardimplementation darauf ausgelegt ist, dass der *base reward* Wert jeweils genau einer Aktion zugeordnet ist. Daraus ergibt sich auch, dass der *base reward* von binärer Natur („Zielobjekt in Überwachungsreichweite“ oder „Zielobjekt nicht in Überwachungsreichweite“) ist, weshalb Zwischenzustände für diesen Wert, der sich aus der mehrfachen Bewegung des Zielobjekts ergeben könnte (z.B. „War zwei von drei Schritten in der Überwachungsreichweite“ $\Rightarrow \frac{2}{3}$ *base reward*), ausgeschlossen werden soll. Insbesondere würde dies eine mehrfache Erhebung der Sensordaten erfordern.

Für den *base reward* ergeben sich somit folgende Möglichkeiten:

1. Ermittlung der einzelnen *base reward* Werte jeweils direkt nach der Ausführung

einer einzelnen Aktion

2. Ermittlung aller *base reward* Werte nach Ausführung aller Aktionen der Agenten und des Zielobjekts

Werden die *base reward* Werte sofort ermittelt (Punkt 1), 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. Bei Punkt 2 müsste man bis zum neuen Zeitschritt warten, bis neue Sensordaten ermittelt wurden.

2.6.7 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 (wie am Ende in Kapitel 2.6.4 angemerkt) in Schritt 5 nach der Ausführung der Aktion direkt neue Sensordaten erfasst und eine neue Aktion berechnet.

Insgesamt ergibt sich also folgender Ablauf:

1. Bestimmen 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 (für lernende Agenten),
4. Aktualisierung der Regeln anhand des *base reward* Werts (für lernende Agenten),
5. **Wahl der Aktion** anhand der Regeln des jeweiligen Agenten bzw. Zielobjekts sowie
6. **Ausführung der Aktion** aller (in zufälliger Reihenfolge, Zielobjekt wiederholt u.U. Schritte 2 und 3 zwischen zwei eigenen Bewegungen).

Kapitel 3

XCS

TODO die ausführliche DIskussion der Parameter ist notwendig, da der Unterschied zum zufälligen Agenten so gering ist und keine Vergleichsarbeiten existieren!

Jeder Agent besitzt ein unabhängiges, sogenanntes *eXtended Classifier System* (XCS), 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].
Im Folgenden soll sich also auf den Teil konzentriert werden, der relevant für das Verständnis der in Kapitel 4 vorgestellten XCS Varianten ist (für eine umfassende Beschreibung von XCS soll auf [But06b] verwiesen werden).

Im Wesentlichen besteht ein XCS aus folgenden Elementen:

1. Einer Menge an 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 3.4)
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 3.3.5)
5. Eine Mechanismus zur Bewertung der *classifier* (mittels *reinforcement learning*, siehe Kapitel 3.3.4)

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 sollen nun die ersten drei der vorgestellten Punkte näher betrachtet werden und das Kapitel mit einer Diskussion und Analyse der XCS Parameter in Kapitel 3.5 abgerundet werden.

3.1 Classifier

Ein *classifier* besteht aus einer Anzahl im folgenden diskutierten Variablen die anhand der in Kapitel 3.5 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 der der *classifier* ausgewählt und dessen *action* Wert ausgeführt wird.

3.1.1 Der *condition* Vektor

Der *condition* Vektor gibt die Kondition an, in welcher Situation der zugehörige *classifier* ausgewählt werden kann, d.h. welche Sensordatensätze von dem jeweiligen *classifier* erkannt werden. 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).

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

Abbildung 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 4 Bewegungsrichtungen, die in Kapitel 2.3.4 besprochen wurden.

3.1.3 Der *fitness* Wert

Der *fitness* Wert soll die allgemeine Genauigkeit des *classifier* repräsentieren und wird über die Zeit hinweg sukzessive an die beobachteten *reward* Werte angepasst. Der Wertebereich verläuft zwischen 0.0 und 1.0 (maximale Genauigkeit). Insbesondere eines der frühesten 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 Genauigkeit des *classifier* bzgl. des *reward prediction* Werts (die durchschnittliche Differenz zwischen *reward prediction* und *reward*) repräsentieren. U.a. auf Basis dieses Werts wird der *fitness* Wert des *classifier* angepasst.

3.1.6 Der *experience* Wert

Der *experience* Wert des *classifier* repräsentiert die Anzahl, wie oft ein *classifier* aktualisiert wurde, also wieviel Erfahrung er 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

Durch Subsummation (siehe Kapitel 3.2.2 und Kapitel 3.3.5) können *classifier* eine Rolle als *macro classifier* spielen, d.h. *classifier* die andere *classifier* in sich beinhalten. Der *numerosity* Wert gibt an, wieviele andere, sogenannte *micro classifier* sich in dem jeweiligen *classifier* befinden. Was die Implementation betrifft sei Kapitel A.3 zu erwähnen, verglichen mit der originalen Implementierung wurden einige Änderungen vorgenommen.

3.2 Vergleich des *condition* Vektors mit den Sensordaten

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 soll anzeigen, dass beim Vergleich zwischen dem *condition* Vektor und den Sensordaten diese Stelle ignoriert werden soll. 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 welchem dann wie in Kapitel 3.4 beschrieben der eigentliche *classifier* ausgewählt wird.
Im Folgenden soll nun zum einen untersucht werden, 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 Sensordatenpaare

Beim Vergleich der Sensordaten und Daten aus dem *condition* Vektor werden immer jeweils zwei Paare verglichen. In Kapitel 2.3 wurde erwähnt, dass der Fall (0/1) in den Sensordaten nicht auftreten kann, weswegen (um die Aufgabe nicht unnötig zu erschweren) ein Datenpaar (0/1) im *condition* Vektor äquivalent zum Datenpaar (1/1) sein soll, es damit also eine gewisse Redundanz gibt. Daraus folgt, dass auch das Datenpaar (0/#) zu

(#/#) äquivalent ist, also beide Datenpaare die selben Sensordatenpaare erkennen. Es ergeben sich also folgende Fälle:

1. Sensorenpaar (0/0) wird erkannt von (0/0), (#, 0), (0, #), (#, #)
2. Sensorenpaar (1/0) wird erkannt von (1/0), (#, 0), (1, #), (#, #)
3. Sensorenpaar (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 im 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* zu zusammenzulegen, wodurch die Gesamtzahl der *classifier* sinkt und somit Erfahrungen, die ein XCS Agent sammelt, nicht unbedingt mehrfach gemacht werden müssen. Die dahinter stehende Annahme ist, dass es Situationen gibt, in denen der Gewinn der durch Unterscheidung zwischen zwei verschiedenen Sensordatensätzen geringer ist als die Ersparnis durch das Zusammenlegen beider *classifier*, d.h. dem Ignorieren

der Unterschiede.

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. Andere *classifier* (in derselben *action set* Liste, also mit gleichem *action* Wert) werden durch den *subsumer* ersetzt, sofern der von ihnen abgedeckte Sensordatenbereich eine Teilmenge des von dem *subsumer* abgedeckten Bereichs ist, der *subsumer* also an allen Stellen des *condition* Vektors entweder denselben Wert wie der zu subsummierende *classifier* oder einen Platzhalter besitzt.

3.3 Ablauf eines XCS

TODO Schreiben

Ein XCS läuft wie folgt ab:

1. Vervollständigung der *classifier* Liste (*covering*, siehe Kapitel 3.3.1)
2. Auswahl auf die Sensordaten passender *classifier* (*match set* Liste, siehe Kapitel 3.3.2)
3. Bestimmung der Auswahlart und Auswahl der Aktion (*explore/exploit*, siehe Kapitel 3.4)
4. Erstellung der zur Aktion zugehörigen Liste von *classifier* (*action set* Liste, siehe Kapitel 3.3.3)

3.3.1 Abdeckung aller Aktionen durch *covering*

Beim sogenannten *covering* wird die Menge aller *classifier* aus dem letzten *match set* (siehe Kapitel 3.3.2) untersucht, ob für jede mögliche Aktion jeweils mindestens ein *classifier* vorhanden ist. Ist dies nicht der Fall, wird ein neuer *classifier* mit dieser Aktion als seinen *action* Wert und einem *condition* Vektor, der 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 3.5.1 in Abbildung 3.2 sieht man beispielsweise, dass in dem dortigen Szenario mindestens bis zu einer Größe von 64 dies 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 3.4 von Bedeutung sind (die sogenannten *predictionFitnessProductSum* Werte).

3.3.3 Die *action set* Liste

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

3.3.4 Bewertung der Aktionen (*base reward*) TODO

XCS ist darauf ausgelegt, dass es eine komplette, genaue und möglichst allgemeine Darstellung einer *reward* Funktion darstellt. 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

beim oben erwähnten *6-Multiplexer* Problem prüft die *reward* Funktion, ob das XCS aus den 4 Datenbits anhand der 2 Steuerbits das richtige Datenbit gewählt hat, also ob 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 jeweilige Lösung zu bewerten.

Bei komplexeren Problemen, bei denen ein Agent nur lokale Informationen zur Verfügung hat (beispielsweise bei *Maze N* die angrenzenden Felder), liefert die *reward* Funktion nur eine Teilinformation, beispielsweise „1“ beim letzten Schritt auf das Ziel und „0“ sonst. Diese Art von Bewertung, die der Agent direkt aus den Sensordaten berechnet, soll in diesem Zusammenhang im folgenden *base reward* genannt werden.

Die Frage nun ist, wie diese Bewertung aus den Sensordaten berechnet wird. Für alle 2^{24} Situationen (tatsächlich weniger, da es nur ein Zielobjekt gibt und bestimmte Situationen nicht auftreten können), die die Sensoren der Agenten erkennen können, könnte ein eigener *base reward* Wert vergeben werden, weshalb sich die Frage nach dem Ziel des Ganzen stellt. Ziel ist, die globale Aufgabe zu erfüllen, also mithilfe der Sensordaten über die Zeit hinweg ein möglichst akkurates Gesamtbild des Problems zu bilden um auf dieser Information aufbauend möglichst gute Entscheidungen zu treffen.

Die optimale Darstellung der *reward* Funktion bei XCS in der *multi step* Variante ist zum einen eine simple Abfrage, ob die Zielposition erreicht wurde („1“) oder nicht („0“) und zum anderen der Aufbau eines Gesamtbilds über die Weitergabe mittels des jeweiligen *maxPrediction* Werts. Auf Basis dessen wird ein vereinfachter Gesamtweg gebildet (vereinfacht deshalb, weil Situationen und Aktionen in den *classifier set* Listen gespeichert werden und nicht Aktionsreihenfolgen und/oder Positionsangaben zusammen mit

der auszuführenden Aktion) und somit zumindest teilweise das Problem in ein *single step* Problem überführt.

Der selbe Gedankengang muss bei einem Überwachungsszenario ausgeführt werden. Beim zweiten Teil der Frage (Darstellung des Gesamtproblems aus lokaler Information) kann man zwar, wie es in Kapitel 4.2 getan wird, XCS einfach auf das Szenario fast ohne Änderung übertragen, die Ergebnisse, wie sie in Kapitel 5 gezeigt werden, sind dann aber nicht viel besser als ein sich zufällig bewegender Agent. Ein alternativer Weg, der dann in Kapitel 4 vorgestellt und in Kapitel 5 signifikant bessere Ergebnisse erbringt, 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.

Übrig bleibt dann nur noch die Frage nach dem ersten Teil der Frage, also wie der *base reward* selbst berechnet wird. Hierzu wurden verschiedene Ansätze ausprobiert, da aber weder ein Qualitätsgewinn festgestellt wurde, noch eine Verkomplizierung bei der Berechnung für die weitere Untersuchung hilfreich erschien, blieb es bei der einfachst möglichen Implementation. Diese bestand aus einer einfachen Abfrage, ob das Zielobjekt in Sichtweite ist oder nicht. Die Einbeziehung von Agenten und Hindernissen führte nicht zum Erfolg, vermutlich deshalb, weil die durch die Sensoren wahrgenommen Positionen zu unklar waren. Mit Sensoren, die exakte Positionen bestimmen können, wäre vorstellbar, dass z.B. kein positiver *base reward* vergeben wird, wenn andere Agenten sich näher am Zielobjekt befinden.

Was die Implementierung betrifft, wird die mit der Bewertung zusammenhängende Funktion *checkRewardPoints()* in Programm A.10 (Zeile 15) bzw. in Programm A.13

(Zeile 16) aufgerufen.

3.3.5 Genetische Operatoren

Es werden aus der jeweiligen *action set* Liste zwei *classifier* (die Eltern) zufällig ausgewählt und zwei neue *classifier* (die 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 3.5 aufgelistet initialisiert. Werden Kinder in die Population eingefügt, deren *action* Wert und *condition* Vektor identisch mit existierenden *classifier* ist, werden sie stattdessen subsummiert.
Da die Sensoren und somit auch der *condition* Vektor aus drei in sich geschlossenen Gruppen bestehen, werden im Unterschied zur Standardimplementation 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:

$$[(z_{1k}, a_{1k}, h_{1k}), (z_{2k}, a_{2k}, h_{2k})] = [(z_1, a_1, h_1), (z_2, a_2, h_2)]$$

$$[(z_{1k}, a_{1k}, h_{1k}), (z_{2k}, a_{2k}, h_{2k})] = [(z_2, a_1, h_1), (z_1, a_2, h_2)]$$

$$[(z_{1k}, a_{1k}, h_{1k}), (z_{2k}, a_{2k}, h_{2k})] = [(z_1, a_2, h_1), (z_2, a_1, h_2)]$$

$$[(z_{1k}, a_{1k}, h_{1k}), (z_{2k}, a_{2k}, h_{2k})] = [(z_2, a_2, h_1), (z_1, a_1, h_2)]$$

3.4 Auswahlart der *classifier*

In jedem Zeitschritt gilt es zu entscheiden, welche Bewegung ein Agent ausführen soll. Als Basis der Entscheidung hat ein Agent zum einen die Sensordaten und zum anderen das eigene *classifier set* zur Verfügung. 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), stellt sich die Frage, welche der Aktionen ausgeführt werden soll.

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 um dann 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 soll im folgenden *predictionFitnessProductSum* genannt werden.

In der ursprünglichen Implementierung [But00] wurden dann folgende Arten beschrieben, wie eine Aktion aus diesen vier ausgewählt werden kann:

1. *random selection* : Zufällige Auswahl einer Aktion (identisch mit zufälliger Bewegung)
2. *roulette wheel selection* : Zufällige Auswahl einer Aktion, Wahrscheinlichkeit abhängig vom *predictionFitnessProductSum* Wert der jeweiligen Gruppe
3. *best selection* : Auswahl der Aktion mit dem höchsten *predictionFitnessProductSum* Wert der jeweiligen Gruppe

Im Folgenden sollen diese Auswahlarten näher vorgestellt und außerdem noch eine weitere Auswahlart aus der Literatur besprochen werden. Im nächsten Abschnitt (Kapi-

tel 3.4.5) soll dann der Wechsel zwischen diesen Auswahlarten näher untersucht und die tatsächlichen Testergebnisse (Kapitel 5.3) zwischen den vorgestellten Varianten präsentiert werden.

3.4.1 Auswahlart *random selection*

Bei einem dynamischen Überwachungsszenario ist es im Vergleich zu standardmäßigen statischen Szenarien weder nötig noch hilfreich *random selection* zu nutzen. Die Idee für diese Auswahlart in einem statischen Szenario ist, dass man möchte, dass das XCS möglichst vielen verschiedenen Situationen ausgesetzt ist. Da in einem statischen Szenario Start- und Zielposition wie auch die Hindernisse fest sind, ist es wichtig, durch *random selection* dem XCS einen gewissen Spielraum zu geben.

Bei einem dynamischen Szenario (siehe Kapitel 2.1) ergibt sich dieses Problem nicht, andere Agenten und das Zielobjekt sind in stetiger Bewegung, der eigene Startpunkt ist nicht fixiert und das Problem wird bei Erreichen des Ziels nicht neugestartet. Aufgrund der Natur der Aufgabenstellung ist es in einem Überwachungsszenario außerdem wichtig, dass das XCS über eine längere Zeit hinweg eine gute Leistung liefert, also stetig gute Entscheidungen trifft, eine zufällige Auswahl scheint also wenig zielführend zu sein.

3.4.2 Auswahlart *best selection*

Bei der Auswahlart *best selection* wird einfach nur die Aktion mit dem höchsten *predictionFitnessProductSum* Wert ausgewählt. Die Verwendung dieser Auswahlart kann u.U. schnell in eine Sackgasse bzw. zu langen Folgen gleicher Aktionen (beispielsweise andauernd gegen eine Wand laufen) führen, sofern sich die Umwelt nicht ändert. Auf den ersten Blick scheint es zwar, dass z.B. zur Verfolgung von einem Zielobjekt ein kompromissloses

Verhalten sinnvoll ist, jedoch bedarf dies zum einen bereits guter, gelernter *classifier* und zum anderen vollständige Information. In dem in dieser Arbeit betrachteten Szenario sind die Sensordaten allerdings beschränkt, der Agent weiß nicht genau, wo sich das Zielobjekt befindet, selbst wenn es in Sicht ist. Eine optimale Verhaltensstrategie muss hier also Entscheidungen auf Basis von Wahrscheinlichkeitsverteilungen treffen, weshalb die alleinige Verwendung der Auswahlart *best selection* eher nicht in Frage kommt.

3.4.3 Auswahlart *roulette wheel selection*

Bei dieser Auswahlart bestimmt der *predictionFitnessProductSum* Wert (relativ zu den anderen *predictionFitnessProductSum* Werten) die Wahrscheinlichkeit, ausgewählt zu werden. Diese Auswahlart erscheint sinnvoll, allerdings ist speziell bei diesem Szenario davon auszugehen, dass, wie auch schon in Kapitel 3.4.2 erwähnt, es aufgrund mangelnder Sensorinformation keine eindeutig besten Aktionen gibt, weshalb sich die *reward prediction* Werte der *classifier* sich eher ähneln. Eine auf Proportionen ausgelegte Auswahlart wie *roulette wheel selection* kann deshalb dazu führen, dass es kaum Unterschiede in den Auswahlwahrscheinlichkeiten gibt, mit der eine Aktion ausgewählt wird. Diese Auswahlart ähnelt somit eher der Auswahlart *random selection* als *best selection*.

3.4.4 Auswahlart *tournament selection*

Zu den oben erwähnten drei Möglichkeiten wurde in [MVBG03] eine weitere vorgestellt und in Bezug auf XCS diskutiert, die sogenannte *tournament selection*. Als Vorteile werden 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, Auswahl-

art gefunden, weshalb sie auch hier getestet werden soll. Da dort allerdings die Auswahl auf Basis von einzelnen *classifier* stattfindet, während hier wie in der Standardimplementation von XCS in [But00] alle *classifier* in nach *action* Werten eingeteilten Gruppen sortiert und deren *prediction* und *fitness* Werte zusammengenommen werden, soll hier eine Implementation der Auswahlart *tournament selection* gewählt werden, die näher am ursprünglichen Algorithmus aus dem Bereich der genetischen Algorithmen liegt [MMGG95].

Bei dieser Auswahlart werden allgemein gesagt k Elemente aus einer Menge zufällig ausgewählt, nach ihrem zugehörigen Wert sortiert und absteigend mit Wahrscheinlichkeit p das jeweilige Element gewählt (d.h. das erste mit p, das zweite mit $(1{,}0-p)p$, das dritte mit $(1{,}0-p)^2p$ usw.).

In dem hier besprochenen Fall wären die Mengen immer der Größe 4 (Anzahl der Aktionen) und die Elemente entsprechen jeweils den berechneten *predictionFitnessProductSum* Werten. Der Einfachheit soll k auf den Maximalwert gesetzt werden, damit alle Aktionen zumindest eine geringe Wahrscheinlichkeit besitzen, ausgewählt zu werden.

Im Grunde entspricht diese Auswahlart also der *roulette wheel selection*, allerdings ohne dem Problem, dass die Auswahlwahrscheinlichkeit aufgrund ähnlicher Produkte sich ebenfalls ähneln. Diese Form der Auswahl, bei geeigneter Wahl von k und p, scheint also am vielversprechend zu sein. 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*.

Bei der Implementierung dieser Auswahlart muss man aufpassen, dass bei der Sortierung Einträge mit gleichem Produkt aus *fitness* und *reward prediction* in zufälliger

Reihenfolge aufgeführt werden. Insbesondere am Anfang kann es sonst dazu kommen, dass alle Agenten in die selbe Richtung laufen.

Bei der Bestimmung des idealen Werts für p ist es wichtig, verschiedene Szenarien und sowohl XCS als auch SXCS zu vergleichen, ansonsten ergibt ein späterer Vergleich von XCS und SXCS womöglich nur deshalb einen Vorteil für SXCS, da die Parameterwerte für XCS schlecht gewählt wurden. In den Tests in Kapitel 3.5.6 wurde der Wert 0,84 als für die hier betrachteten Szenarien optimaler Wert bestimmt.

3.4.5 Wechsel zwischen den *explore* und *exploit* Phasen

In der Standardimplementierung von XCS wird zwischen verschiedenen Auswahlarten hin und her geschalten. Die Auswahlarten werden in zwei Gruppen geteilt, in die sogenannte *explore* Phase und in die *exploit* Phase. In der *exploit* Phase soll bevorzugt eine Auswahlart ausgeführt werden, die das Produkt aus den Werten *fitness* und *reward prediction* möglichst stark gewichtet, *best selection* und *tournament selection* sind Kandidaten für die *exploit* Phase, während *random selection* und *roulette wheel selection* Kandidaten für die *explore* Phase wären. Idee ist, dass man mit Hilfe der *explore* Phasen den Suchraum besser erforschen kann, dann aber zur eigentlichen Problemlösung in der *exploit* Phase möglichst direkt auf das Ziel zugeht 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 zwischen *exploit* und *explore* nach jedem Erreichen des Ziels entweder umgeschalten oder zufällig mit einer bestimmten Wahrscheinlichkeit eine Auswahlart ermittelt. Es werden also abwechselnd ganze Probleme entweder im *exploit* oder im *explore* Modus berechnet. Dies erscheint

sinnvoll für die erwähnten Standardprobleme, da nach Erreichen des Ziels ein neues Problem gestartet wird und die Entscheidungen die während der Lösung eines Problems getroffen werden keine Auswirkungen auf die folgenden Probleme hat, die Probleme also nicht miteinander zusammenhängen.

Bei dem hier vorgestellten Überwachungsszenario kann dagegen nicht neugestartet 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 in den Standardszenarien bei XCS der Fall ist, bei denen u.a. die Qualität nur während der *exploit* Phase gemessen wird.

Desweiteren greift auch die 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. Aus diesen Gründen 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* äquivalent mit einem Neustart des Problems. Während dort beim Neustart des Problems das gesamte Szenario (alle Agenten, Hindernisse und das Zielobjekt) auf den Startzustand zurückgesetzt werden, läuft das Überwachungsszenario weiter. Als erweiterten Ansatz soll nun deshalb eine neue Problemdefinition gelten, dass nicht das Erreichen eines positiven *base rewards* (also ein Neustart des Problems) einen

Phasenwechsel auslöst, sondern eine *Änderung* des *base rewards*, so dass mit anfänglicher *explore* Phase immer dann in die *exploit* Phase gewechselt wird, wenn das Zielobjekt in Sicht ist (bzw. umgekehrt, wenn mit der *exploit* Phase begonnen wird). Als Vergleich soll der andauernde, zufällige Wechsel zwischen der *explore* und *exploit* Phase, eine andauernde *exploit* und andauernde *explore* Phase dienen. Es sollen nun also folgende Arten des Wechsel zwischen den Phasen untersucht werden:

1. Andauernde *explore* Phase
2. Andauernde *exploit* Phase
3. Abwechselnd *explore* und *exploit* Phase (bei Änderung des *base reward*, beginnend mit *explore*)
4. Abwechselnd *explore* und *exploit* Phase (bei Änderung des *base reward*, beginnend mit *exploit*)
5. In jedem Schritt zufällig entweder *explore* oder *exploit* Phase (50% Wahrscheinlichkeit jeweils)

Anzumerken sei hier, dass Punkt (3.), (4.) und (5.) in der Standardimplementierung praktisch äquivalent sind, da dort die Phasen separat betrachtet werden können und nur nach jedem Problem vertauscht werden. Dabei wird bei jedem Erreichen eines positiven *reward* zwischen *explore* und *exploit* hin und hergeschaltet, was in der Standardimplementierung dem Beginn eines neuen Problems entspricht.

3.5 Beschreibung und Analyse der XCS Parameter

Die Einstellungen der XCS Parameter der durchgeführten Experimente entsprechen weitgehend den Vorschlägen in [BW01] („Commonly Used Parameter Settings“). Eine Auflistung findet sich in Tabelle 3.1. Im Folgenden sollen Parameter besprochen werden, 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 den meisten Fällen war die Standardeinstellung jedoch passend.

Mitunter führen andere Parametereinstellungen auch zu wesentlich besseren Ergebnissen. Dies muss man aber vorsichtig bewerten, wenn die erreichte Qualität unter der des zufälligen Algorithmus liegt, da eine Auswirkung sein kann, dass der Algorithmus nicht besser lernt, sondern sich umgekehrt eher wie der zufällige Algorithmus verhält. Ein Vergleich mit der Qualität des zufälligen Algorithmus wird deswegen jeweils immer angegeben.

Anzumerken sei, dass alle Tests jeweils mit den in Tabelle 3.1 angegebenen Parameterwerten durchgeführt wurden und bei jedem Test jeweils nur der zu untersuchende Wert verändert wurde. Um synchronisierte und vergleichbare Daten zu haben, wurden die Tests deshalb in vielen Etappen durchgeführt, die angegebenen Testergebnisse entsprechen jeweils nur den endgültigen Ergebnissen.

3.5.1 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 neuerstellten *classifier* gegen Null geht. In Abbildung 3.2 ist dies für das angegebene Szenario ab einer Populationsgröße von 256 erfüllt.

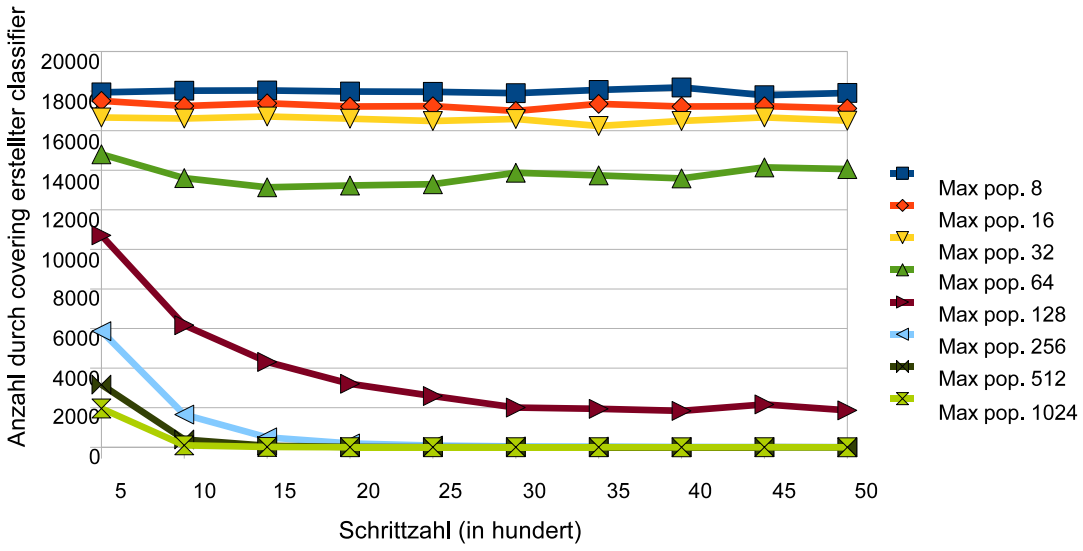

Abbildung 3.2: Auswirkung der maximalen Populationsgröße auf die Anzahl der *classifier* die durch *covering* neuerstellt werden (Säulenszenario, Zielobjekt mit einfacher Richtungsänderung, Geschwindigkeit 1, Agenten mit SXCS)

Bei der Wahl eines geeigneten Werts 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, also wieviele *classifier* mit verschiedenen *condition* Vektoren und verschiedenem *action* Wert in der *covering* Funktion konstruiert werden können. Würde man beispielsweise weitere Zielobjekte auf das Feld setzen, könnten eine Reihe weiterer Situationen auftreten, beispielsweise könnten Zielobjekte in Sicht in zwei unterschiedlichen Richtungen auftauchen. Selbiges gilt für das Szenario ohne Hindernisse, hier fällt eine ganze Anzahl von Möglichkeiten heraus, was man in Abbildung 3.3 als Vergleich sehen kann.

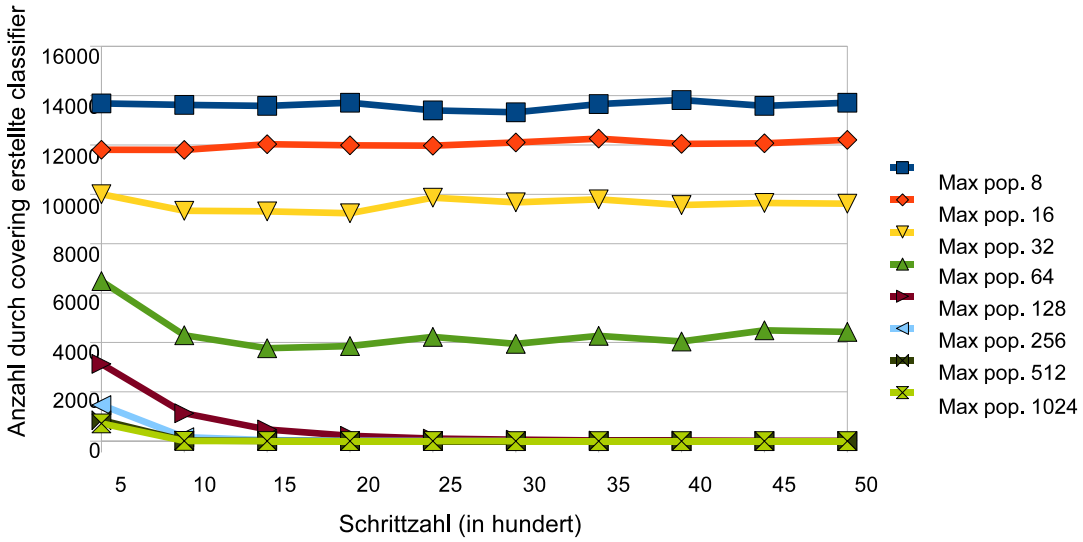

Abbildung 3.3: Auswirkung der maximalen Populationsgröße auf die Anzahl der *classifier* die durch *covering* neuerstellt werden (leeres Szenario ohne Hindernisse, Zielobjekt mit einfacher Richtungsänderung, Geschwindigkeit 1, Agenten mit SXCS)

Für den Overhead (d.h. die Zeit, die 8 Agenten mit zufälliger Bewegung benötigen) ergab sich eine mittlere Laufzeit von 1,67s pro Experiment bei 500 Schritten (bzw. 6,50s bei 2000 Schritten), was die anfängliche Stagnation bis $N = 32$ erklärt. Zieht man diesen von den Messwerten (siehe Abbildung 3.5) ab, erhält man im betrachteten Wertebereich einen nahezu linearen Verlauf (siehe Abbildung 3.6, ab $N > 128$). Der fallende Verlauf bis 128 erklärt sich durch den Overhead des XCS Algorithmus selbst.

Da also die wichtigsten *classifier* mit Populationsgröße 256 (bzw. 128 im leeren Szenario) bereits abgedeckt sind, führt eine Erhöhung der Populationsgröße nur zu einer Erhöhung der Laufzeit. Da den Agenten das Szenario unbekannt ist, soll für alle Szenarien der selbe Wert benutzt werden soll. Alles in allem scheint somit $N = 256$ die schnellste Parametereinstellung zu sein, die gleichzeitig auch ausreichend Platz für *classifier* für die Abdeckung der Möglichkeiten der betrachteten Szenarien bietet.

Die Tests liefen auf einem T7500, 2,2 GHz in einem einzelnen Thread. Als Vergleich hierzu wurde auch der Einfluss der Kartengröße auf die Laufzeit betrachtet, wie in Abbildung 3.4 zu sehen, ist der Einfluss auf die Laufzeit im getesteten Bereich (16x16 - 64x64) ohne Bedeutung.

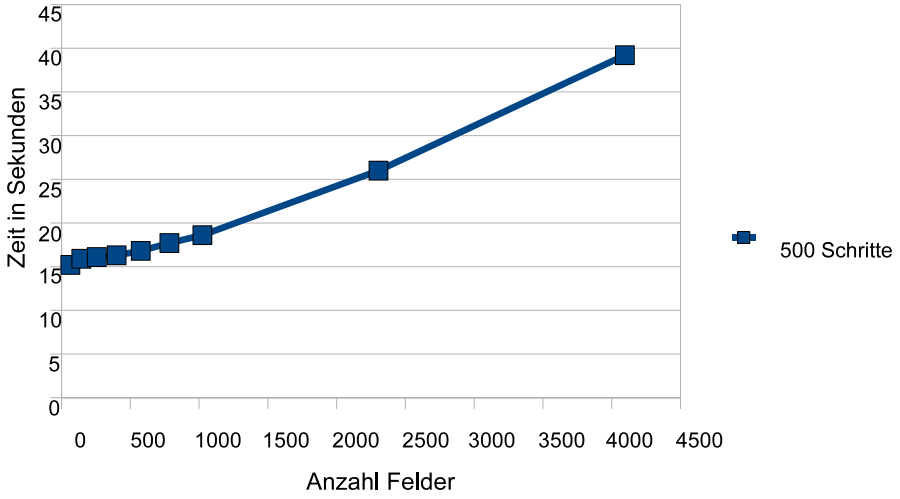

Abbildung 3.4: Darstellung der Auswirkung der Torusgröße auf die Laufzeit im leeren Szenario, zufälliger Bewegung des Zielobjekts, Geschwindigkeit 1, sich zufällig bewegenden Agenten

3.5.2 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 *classifiers* mit zufälligen *action* Werten und *condition* Vektoren gefüllt wird. Dort wird aber auch angemerkt, dass beide Varianten in ihrer Qualität sich nur wenig unterscheiden. Da zum einen 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

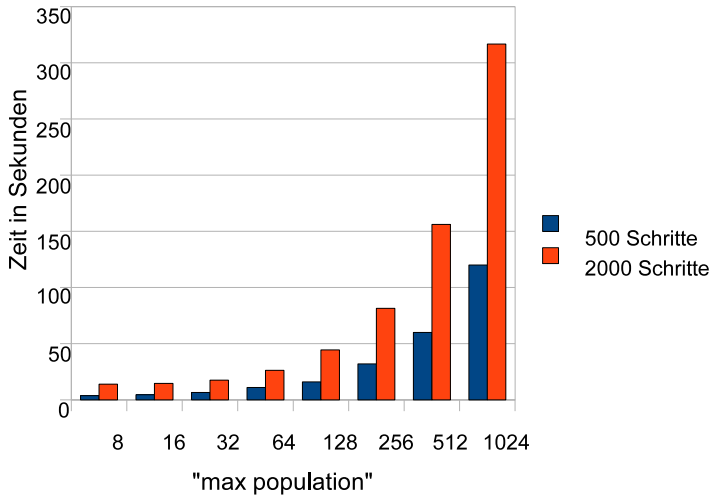

Abbildung 3.5: Darstellung der Auswirkung des Parameters *max population* N auf die Laufzeit im leeren Szenario, zufälliger Bewegung des Zielobjekts, Geschwindigkeit 1, Agenten mit SXCS Algorithmus

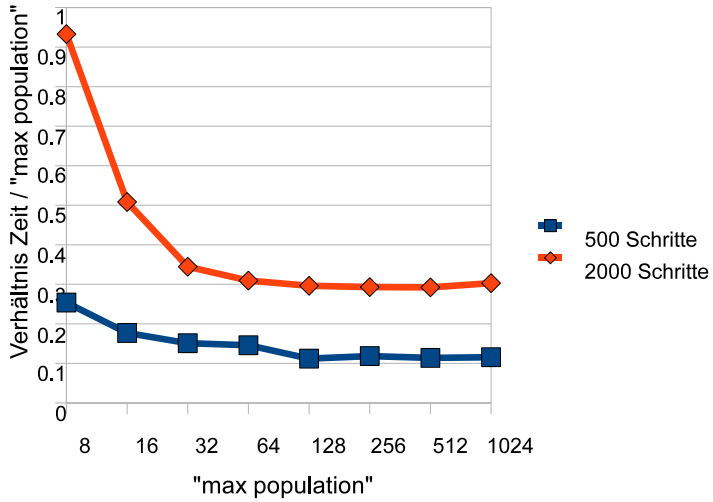

Abbildung 3.6: Darstellung der Auswirkung des Parameters *max population* N auf das Verhältnis der Laufzeit zu N im leeren Szenario, zufälliger Bewegung des Zielobjekts, Geschwindigkeit 1, Agenten mit SXCS Algorithmus

mit einer leeren *classifier set* Liste zu starten.

Dies bestätigen auch Tests, vergleicht man die Anzahl durch *covering* neu erstellter *classifier* ohne zufällige Initialisierung der *classifier set* Liste (Abbildung 3.7) mit der mit Initialisierung (Abbildung 3.2) erkennt man, dass zwar anfangs weniger neue *classifier* generiert werden müssen, umgekehrt aber einige der generierten *classifier* kaum mehr aus dem *classifier set* zu bekommen sind. Beispielsweise stagniert die Anzahl der generierten *classifier* im Fall mit vorinitialisierter *classifier set* Liste bei einer Populationsgröße von 128 bei etwa 2000 pro 500 Schritte und 8 Agenten, während sie im Fall ohne Initialisierung gegen 0 geht.

Im zweiten Fall mit vorinitialisierter Liste müssen die überflüssigen *classifier* also erst mühsam erkannt und entfernt werden, was im Grunde die Populationsgröße bis dahin verringert. Es müsste also ein größeres N benutzt werden, was wiederum die Laufzeit erhöht. Aus diesen Gründen sollen alle Agenten mit leerer Liste starten.

3.5.3 Parameter *reward prediction discount* γ

In der Literatur in [BW01] wird ein Standardwert von 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 *reward*, weniger schnell abfällt, wodurch eine längere Verkettung von *reward* Werten möglich ist. Umgekehrt führen zu hohe Werte für γ zu der positiven Bewertung von *classifiers* die am Erfolg gar nicht beteiligt waren, was sich negativ auf die Qualität auswirken kann.
Abbildung 3.8 zeigt einen Vergleich der Qualität bei unterschiedlichen Werten für γ beim XCS Algorithmus im Säulenszenario. Wie vorgeschlagen wird hier jeweils $\gamma = 0{,}71$ ver-

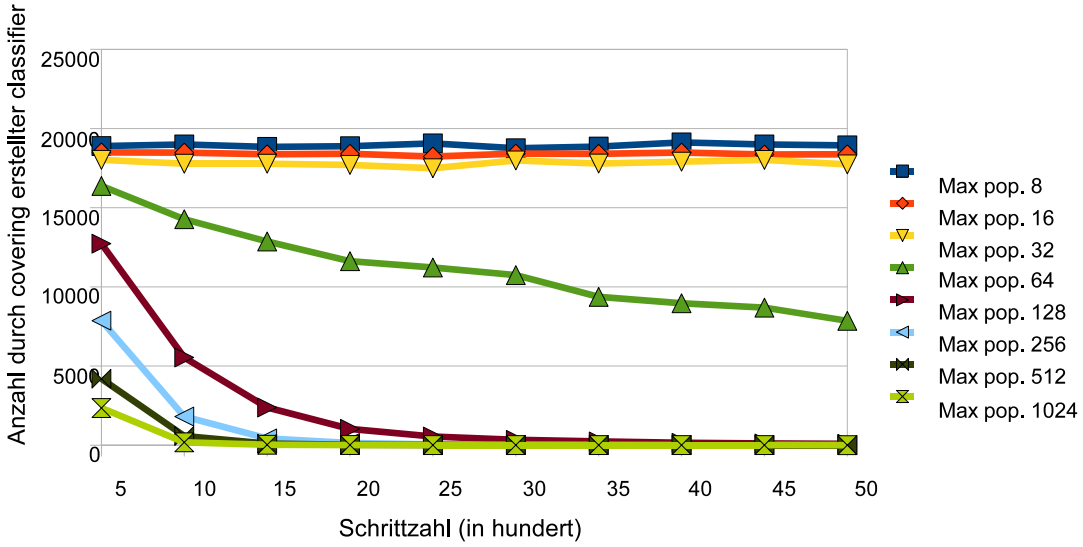

Abbildung 3.7: Auswirkung der maximalen Populationsgröße auf die Anzahl der *classifier*, die durch *covering* neuerstellt werden (Säulenszenario, Zielobjekt mit einfacher Richtungsänderung, Geschwindigkeit 1, Agenten mit SXCS, ohne Initialisierung der *classifier set* Liste)

wendet werden.

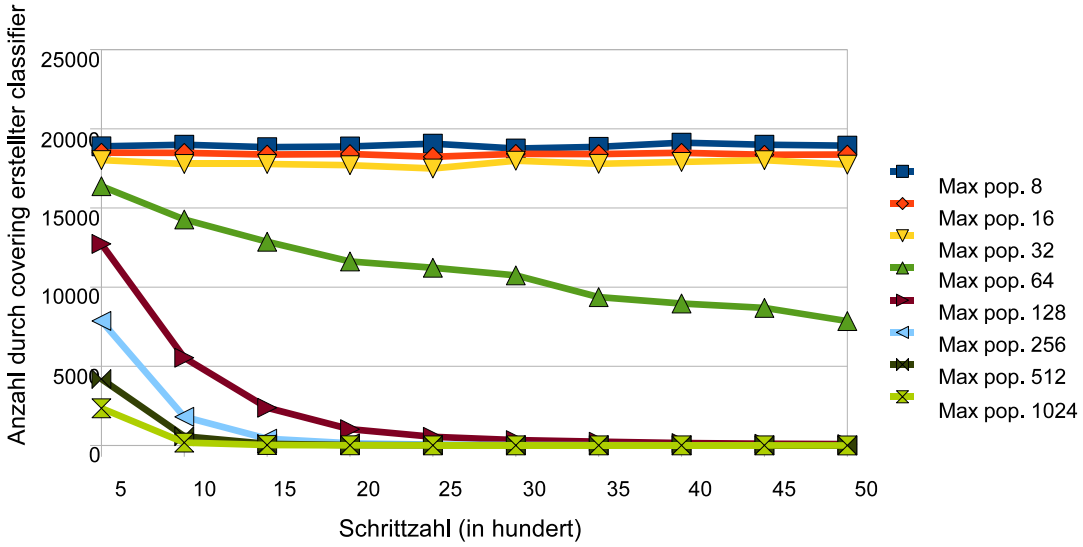

Abbildung 3.8: Auswirkung verschiedener *prediction discount* γ Werte auf die Qualität (Säulenszenario, Zielobjekt mit einfacher Richtungsänderung und Geschwindigkeit 1, Agenten mit XCS)

3.5.4 Parameter Lernrate β

Die Lernrate β hatte in den Tests kaum Auswirkungen auf die Qualität. Da eine ausreichend hohe Populationsgröße gewählt wurde (es werden nicht dauernd neue *classifier* erstellt) und die Schrittzahl groß genug war, pendelten sich die entsprechenden Werte ein. Die Lernrate bestimmt, wie stark ein ermittelter *reward* Wert den *reward prediction*, *reward prediction error*, *fitness* und *action set size* Wert bei jeder Aktualisierung beeinflusst. Vergleichende Tests (siehe Abbildung 3.9) lassen einen leichten Abwärtstrend bei größeren Werten feststellen, signifikante Unterschiede, vom zusätzlich dargestellten Extremwert bei 0,00001 abgesehen, gibt es aber keine, weshalb die Wahl im Grunde beliebig ist.

TODO! Ein Unterschied ist erst in schwierigen Szenarien zu bemerken (siehe Abbildung 3.10), dort soll 0,001 als Lernrate β gewählt werden. Der Grund für diesen Unterschied ist schwierig zu erkennen. Nötig war hierzu die Betrachung des laufenden Durchschnitts der Qualität des Algorithmus (siehe Abbildung 6.1), bei der beide Varianten im ersten Problem bis 2000 Schritte gleichauf sind, dann die Variante mit vergleichsweise großer Lernrate bei niedrigem Level stehenbleibt. Ein weiterer Test mit einer leicht modifizierten SXCS Version (die Option

3.5.5 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, welcher beliebig wählbar ist und lediglich ästhetische Auswirkungen hat. Somit wird dieser auf 1,0 gesetzt und ϵ_0 auf 0,01.

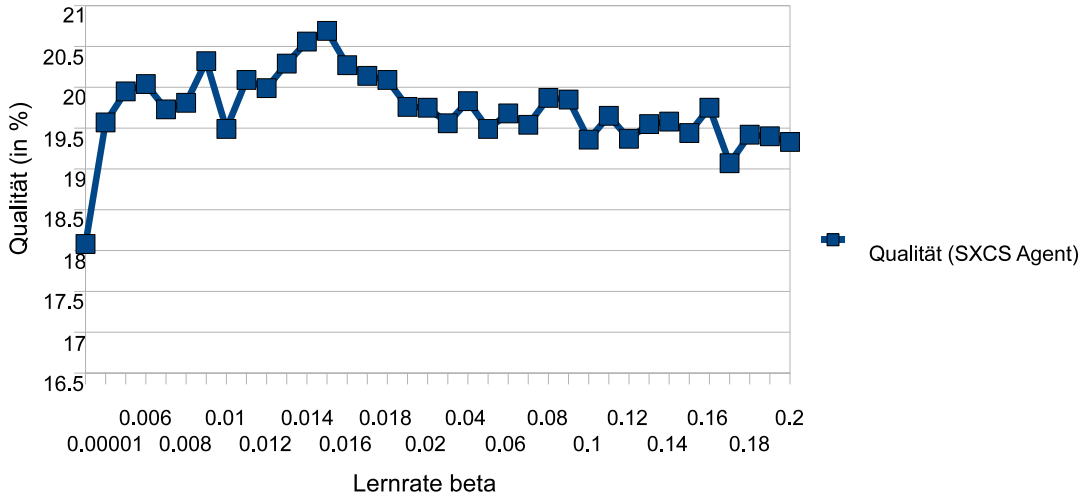

Abbildung 3.9: Auswirkung des Parameters *learning rate* β auf die Qualität im Säulenszenario, intelligente Bewegung des Zielobjekts, Agenten mit SXCS Algorithmus, 2000 Schritte

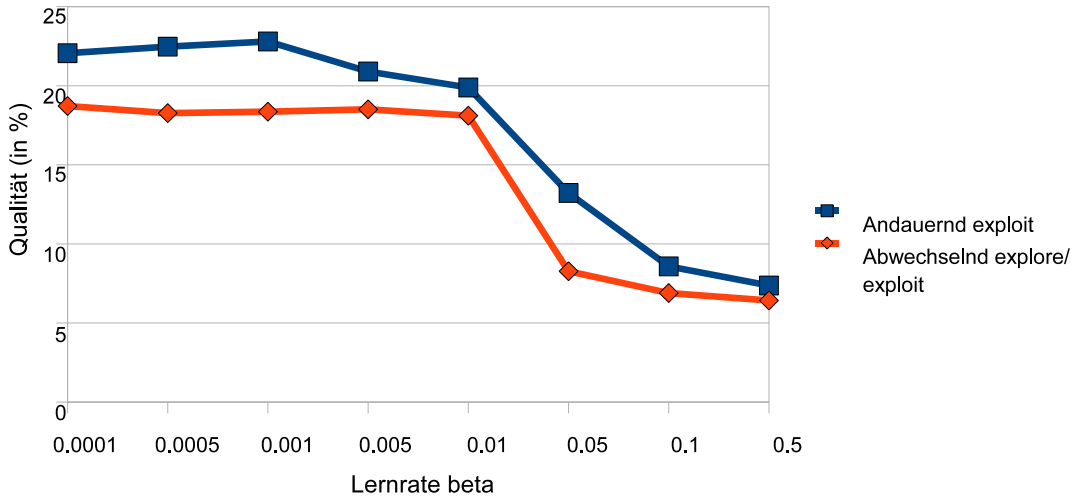

Abbildung 3.10: Auswirkung des Parameters *learning rate* β auf die Qualität im schwierigen Szenario, Bewegung des Zielobjekts ohne Richtungsänderung, Geschwindigkeit 1, Agenten mit SXCS Algorithmus, 2000 Schritte

3.5.6 Parameter *tournament factor p*

In Abbildung 3.11 und Abbildung 3.12 sind die Ergebnisse im Säulenszenario mit einem Zielobjekt mit einfacher Richtungsänderung, einmal mit Geschwindigkeit 1, das andere Mal mit Geschwindigkeit 2, dargestellt. Die Qualitätsdifferenz bezeichnet hier zur besseren Übersicht die Differenz der Qualität des Algorithmus zur Qualität eines Agenten mit zufälliger Bewegung. Was den *tournament factor p* betrifft, ist für XCS das Maximum bei 0,88 (Geschwindigkeit 1) bzw. *0,80* (Geschwindigkeit 2).

Bei SXCS ist deutlich zu sehen, dass eine andauernde *exploit* Phase bei beiden Geschwindigkeiten im Vergleich zu abwechselnden *explore/exploit* Phasen deutlich benachteiligt ist, teilweise sogar schlechter abschneidet als XCS. Die Maximalwerte bei SXCS liegen im Bereich von 0,72 bis 0,88 für Geschwindigkeit 1 und mit langsamer Steigung bei 0,92 für Geschwindigkeit 2. Ein sinnvoller Kompromiss erscheint hier deshalb 0,84 als Wert für den *tournament factor p* zu benutzen.

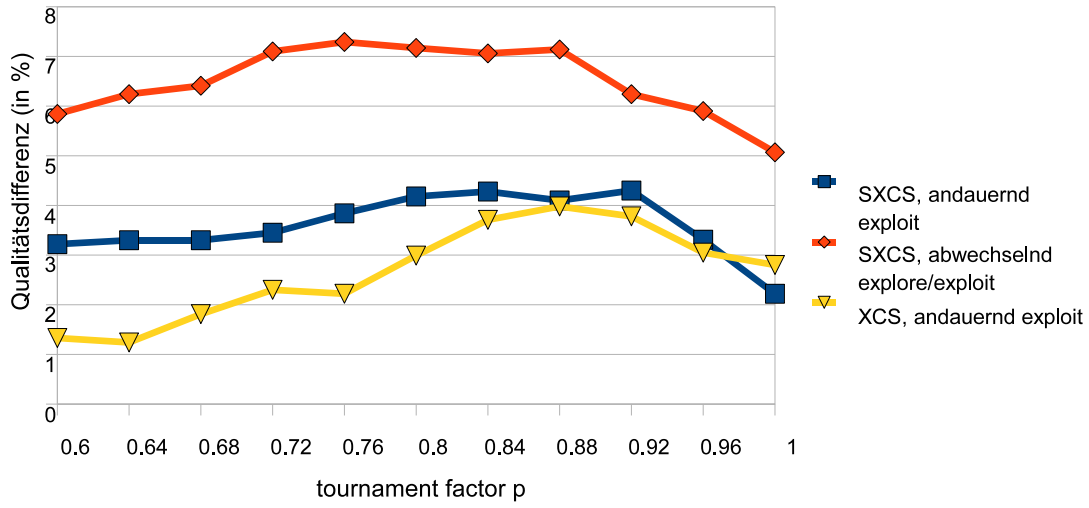

Abbildung 3.11: Vergleich verschiedener Werte p für Auswahlart *tournament selection* (Zielobjekt mit einfacher Richtungsänderung, Geschwindigkeit 2, Säulenszenario, 2000 Schritte)

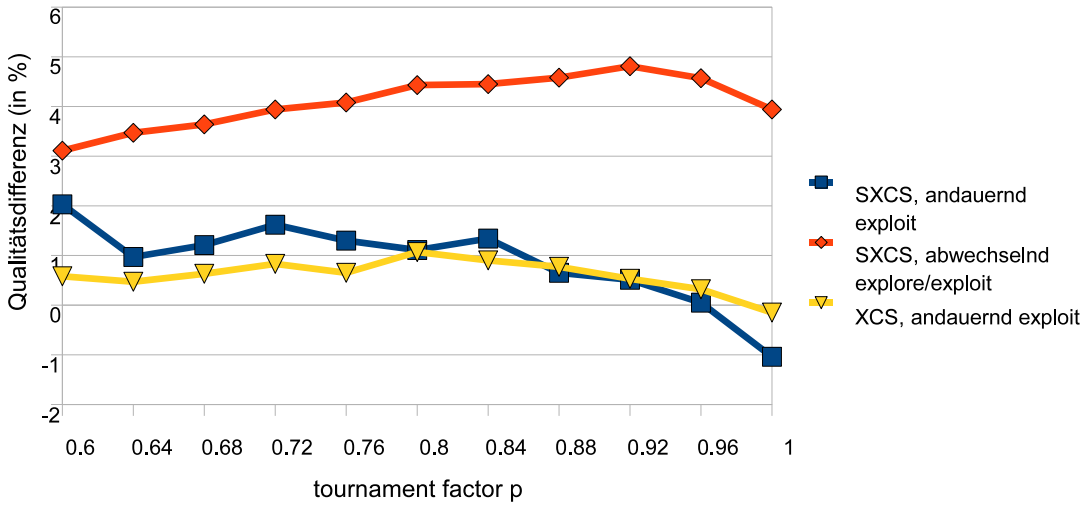

Abbildung 3.12: Vergleich verschiedener Werte p für Auswahlart *tournament selection* (Zielobjekt mit einfacher Richtungsänderung, Geschwindigkeit 2, Säulenszenario, 2000 Schritte)

Im Falle eines Zielobjekts mit intelligenter Bewegung (Abbildung 3.13 mit Geschwindigkeit 1 und Abbildung 3.14 mit Geschwindigkeit 2) fällt direkt ins Auge, dass eine abwechselnde *explore/exploit* Phase nicht vorteilhaft für einen SXCS Agenten ist. XCS erreicht ein ziemlich konstantes Ergebnis im Bereich von 0,76 bis 0,92, während SXCS mit andauernder *exploit* Phase bei einem Wert von um die 0,80 den Maximalwert besitzt.

Insgesamt bestätigt die Untersuchung also, dass $p = 0{,}84$ für diese Szenarien sinnvoll ist und somit die beste Aktion also mit $p = 84\%$ Wahrscheinlichkeit, die zweitbeste mit ca. $(1{,}0 - p)p \approx 13\%$ Wahrscheinlichkeit, die drittbeste mit ca. $(1{,}0 - p)^2 p \approx 2\%$ Wahrscheinlichkeit und die schlechteste Aktion mit ca. $(1{,}0 - p)^3 p \approx 1\%$ Wahrscheinlichkeit gewählt werden.

Außerdem kann man erkennen, dass bei einem sich intelligent verhaltenden Zielobjekt eine andauernde *exploit* Phase die beste Wahl ist. Dies wird in Kapitel 5.3 relevant und dort auch näher diskutiert.

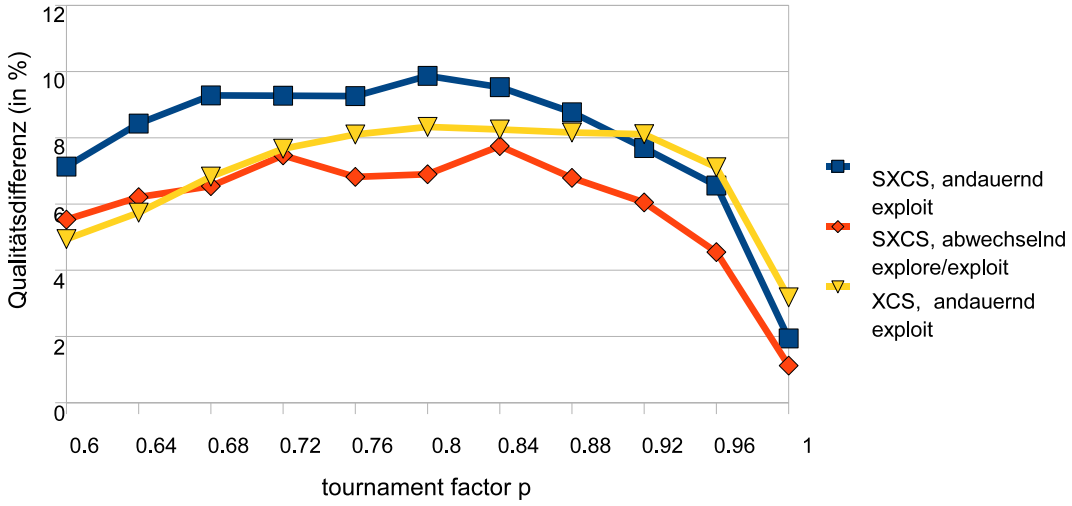

Abbildung 3.13: Vergleich verschiedener Werte p für Auswahlart *tournament selection* (intelligentes Zielobjekt, Geschwindigkeit 1, Säulenszenario, 2000 Schritte)

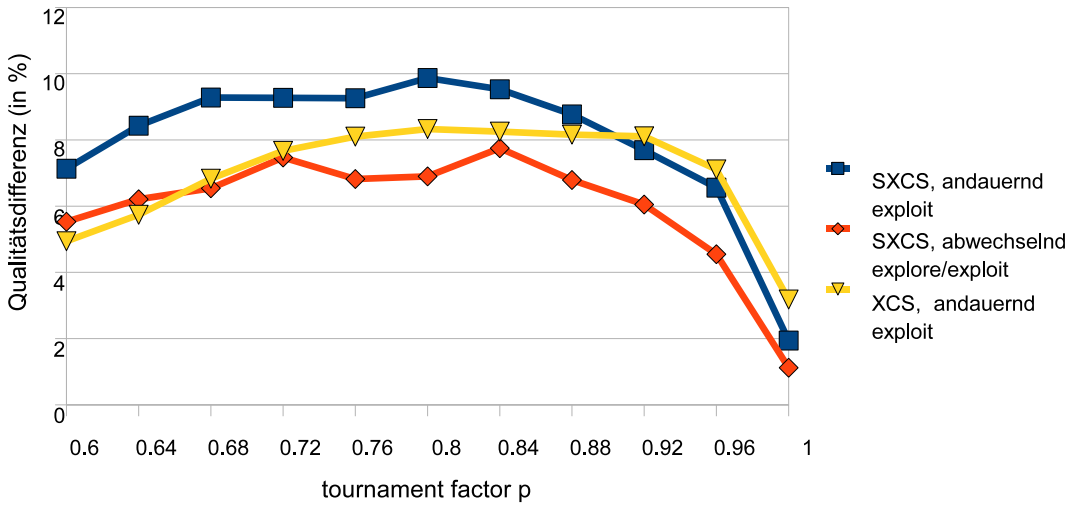

Abbildung 3.14: Vergleich verschiedener Werte p für Auswahlart *tournament selection* (intelligentes Zielobjekt, Geschwindigkeit 2, Säulenszenario, 2000 Schritte)

3.5.7 Übersicht über alle Parameterwerte

Tabelle 3.1: Verwendete Parameter (soweit nicht anders angegeben) und Standardparameter

Parameter	Wert	Standardwert (siehe [BW01])
max population N	**256** (siehe Kapitel 3.5.1)	[so, dass kein *covering* nötig]
max value ρ	**1,0** (siehe Kapitel 3.5.5)	[10000]
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}	25	[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 3.5.2)	[zufällig oder leer]
numerosity	1	[1]
experience	0	[0]
accuracy equality ϵ_0	**0,01** (siehe Kapitel 3.5.5)	[*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	**0,001 - 0,01** (siehe Kapitel 3.5.4)	[0,1-0,2]
exploration probability	0,5 (siehe Kapitel 3.2.2)	[$\sim$ 0,5]
tournament factor	0,84 (siehe Kapitel 3.5.6)	[-]

Kapitel 4

XCS Varianten

Ziel der Arbeit war es, wie man den XCS Algorithmus auf ein Überwachungsszenario anwenden kann. Notwendig dafür war es, die XCS Implementierung vollständig nachzuvollziehen, um für jeden Bestandteil entscheiden zu können, welche Rolle er bezüglich eines solchen Szenarios spielt. Für die Tests wurde nicht auf bestehende Pakete (z.B. XCSlib [Lan]) zurückgegriffen, wenn auch der Quelltext von [But00] Modell stand und aus ihm große Teile entnommen wurden.

Im Vordergrund stand zum einen die grundsätzliche Frage, ob XCS in einem solchen Szenario überhaupt besser als ein Algorithmus sein kann, der sich rein zufällig verhält, und wie mögliche Ansätze aussehen können, den Algorithmus zu verbessern.

Zuerst sollen allgemeine Anpassungen des Algorithmus und der Implementation besprochen werden (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 eines Problems diskutiert. Zum anderen wird eine an Überwachungsszenarios angepasste Variante,

der sogenannte SXCS Algorithmus, vorgestellt werden. Dieser Algorithmus wurde unter dem Gesichtspunkt des Problems einer kontinuierlichen Überwachung eines Zielobjekts entwickelt, also nicht, wie viele der Standardprobleme beim originalen XCS *multi step* Verfahren, einen Weg durch ein Labyrinth zu einem Ziel finden. Schließlich soll in Kapitel 4.4 eine Variante mit verzögerter Aktualisierung des *reward* Werts in den *action set* Listen und eine darauf aufbauende Variante mit Kommunikation mit anderen Agenten vorgestellt werden (der sogenannte DSXCS Algorithmus, siehe Kapitel 4.4).

4.1 Allgemeine Anpassungen

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

1. Die Berechnung der Summe der *numerosity* Werte wurde vollständig neuorganisiert wie auch ein Fehler bei der Aktualisierung des *numerosity* Werts in der Implementierung korrigiert (siehe Kapitel A.3)

2. Der genetischer Operator wird hier zwei feste anstatt zufällige Schnittpunkte für das *two point crossover* verwenden (siehe Kapitel 3.3.5).

3. Die Qualität des Algorithmus wird nicht nur in der *exploit* Phase gemessen werden, da ein fortlaufendes Problem und kein statisches Szenario betrachtet wird (siehe Kapitel 3.4.5).

4. Mehrere XCS Parameter wurden angepasst (siehe Kapitel 3.5).

5. Das Erreichen des Ziels wurde für das Überwachungsszenario neu verfasst wie auch der Neustart von Probleminstanzen neu geregelt wurde (siehe Kapitel 3.3.4).

6. Die Reihenfolge bei der Bewertung, Entscheidung und 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. Somit wird, rückführend vom letzten Schritt auf das Ziel, der *reward* Wert schrittweise (mit jeder neuen Probleminstanz) an vorgehende Aktionen verteilt. Die Annahme ist, dass dann, durch mehrfache Wiederholung des Lernprozesses, mit dem sich dadurch ergebenen Regelsatz mit höherer Wahrscheinlichkeit das Ziel gefunden wird.

Dies entspricht dem aus [BW01] bekannten XCS *multi step* Verfahren. Der wesentliche Unterschied der Implementierung in dieser Arbeit ist, dass das Szenario bei einem positiven *base reward* nicht neugestartet wird, algorithmisch ist die Implementierung ansonsten identisch. Dies zeigt sich in Programm A.10 (Zeilen 22-27), zwar wird 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.11 und A.12 finden sich, neben Anpassungen an den Simulator, keine wesentlichen Änderungen. In Programm A.11 wird der ermittelte *base reward* zusammen mit dem ermittelten *maxPrediction* Wert an die Aktualisierungsfunktion der

jeweiligen *action set* Liste weitergegeben und in Programm A.12 wird eine Aktion 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. In einer statischen Umgebung kann dadurch über mehrere (identische) Probleme hinweg eine optimale Einstellung (des *fitness* und *reward prediction* Werts) für die *classifier* gefunden werden.

Bei der hier besprochenen SXCS Variante (*Supervising eXtended Classifier System*) soll in Kapitel 4.3.1 zuerst die Umsetzung dieser Idee diskutiert. Insbesondere baut sie auf sogenannten Ereignissen auf, die mit einer Änderung des *base reward* Werts einhergehen, welche in Kapitel 4.3.2 erklärt werden. Die Implementierung selbst wird dann in Kapitel 4.3.5 vorgestellt.

4.3.1 Umsetzung von SXCS

Bei SXCS Variante soll die Verbindung zwischen den *action set* Listen direkt durch die zeitliche Nähe zur Vergabe des *base reward* 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.

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

$$r(a) = \begin{cases} \frac{a}{\text{size(actionSet)}} & \text{, falls base reward} = 1 \\ \frac{1-a}{\text{size(actionSet)}} & \text{, falls base reward} = 0 \end{cases}$$

bzw. bei quadratischer Verteilung des *base reward*:

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

Die schematische Abbildung 4.1 demonstriert diesen Sachverhalt nochmals anschaulich. In Tests ergab sich für die quadratische Verteilung des *base reward* ein minimal besseres Ergebnis, weitere Grafiken werden auf die lineare Verteilung des *base reward* beschränkt sein, um eine verständliche Darstellung zu ermöglichen, während in den Simulationen die quadratische Vergabe des *base reward* benutzt wird.

4.3.2 Ereignisse

In XCS wird lediglich das jeweils letzte *action set* Liste aus dem vorherigen Zeitschritt 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* anhand der vergangenen Zeitschritte 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 des jeweiligen *reward* Werts aktualisiert, sobald bestimmte Bedingungen eingetreten sind.

Von einem positiven bzw. negativen Ereignis spricht man, wenn sich der *base reward*

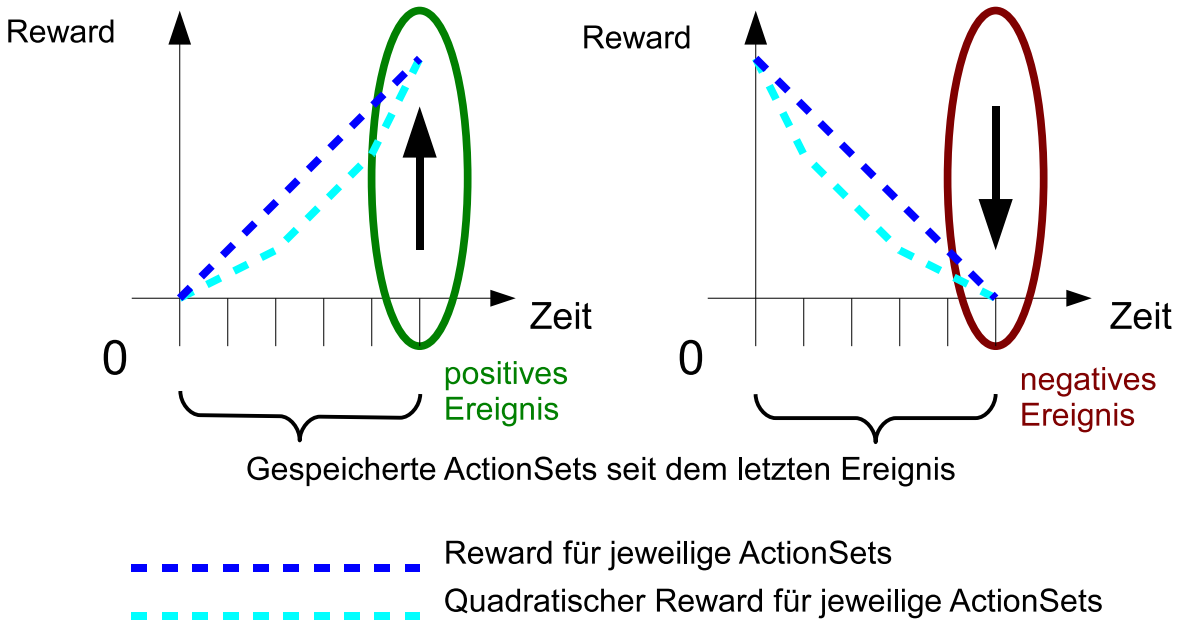

Abbildung 4.1: Schematische Darstellung der (quadratischen) Verteilung des *reward* an gespeicherte *action set* Listen bei einem positiven bzw. negativen Ereignis

im Vergleich zum vorangegangenen Zeitschritt verändert hat, also wenn das Zielobjekt sich in Überwachungsreichweite bzw. aus ihr heraus bewegt hat (siehe Abbildung 4.2).

Bei der Benutzung eines solchen Stacks entsteht eine Zeitverzögerung, d.h. die *classifier* erhalten jeweils Information, die bis zu *maxStackSize* Schritte zu alt sein kann. Tritt beim Stack ein Überlauf ein, gab es also *maxStackSize* Schritte lang keine Änderung des *base reward* Werts, wird abgebrochen und die $\frac{maxStackSize}{2}$ ältesten Einträge vom Stack genommen.

Alle diese Einträge werden vorher dabei mit diesem *base reward* Wert aktualisiert. Abbildung 4.3 zeigt die Bewertung bei einem solchen neutralen Ereignis, bei dem nach 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

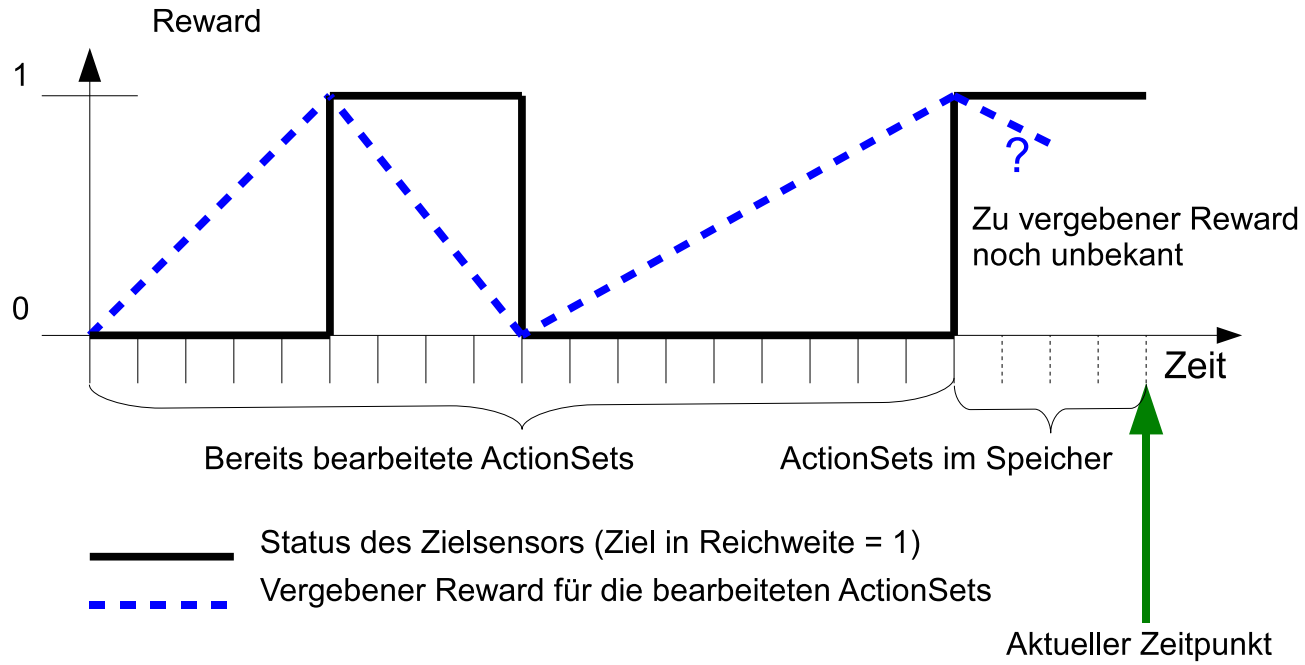

Abbildung 4.2: Schematische Darstellung der zeitlichen Verteilung des *reward* an *action set* Listen nach mehreren positiven und negativen Ereignissen und der Speicherung der letzten *action set* Liste

Änderung des *base reward* Werts auftritt, im dargestellten Fall also der *base reward* sich beim aktuellen Zeitpunkt auf 0 verändern würde.

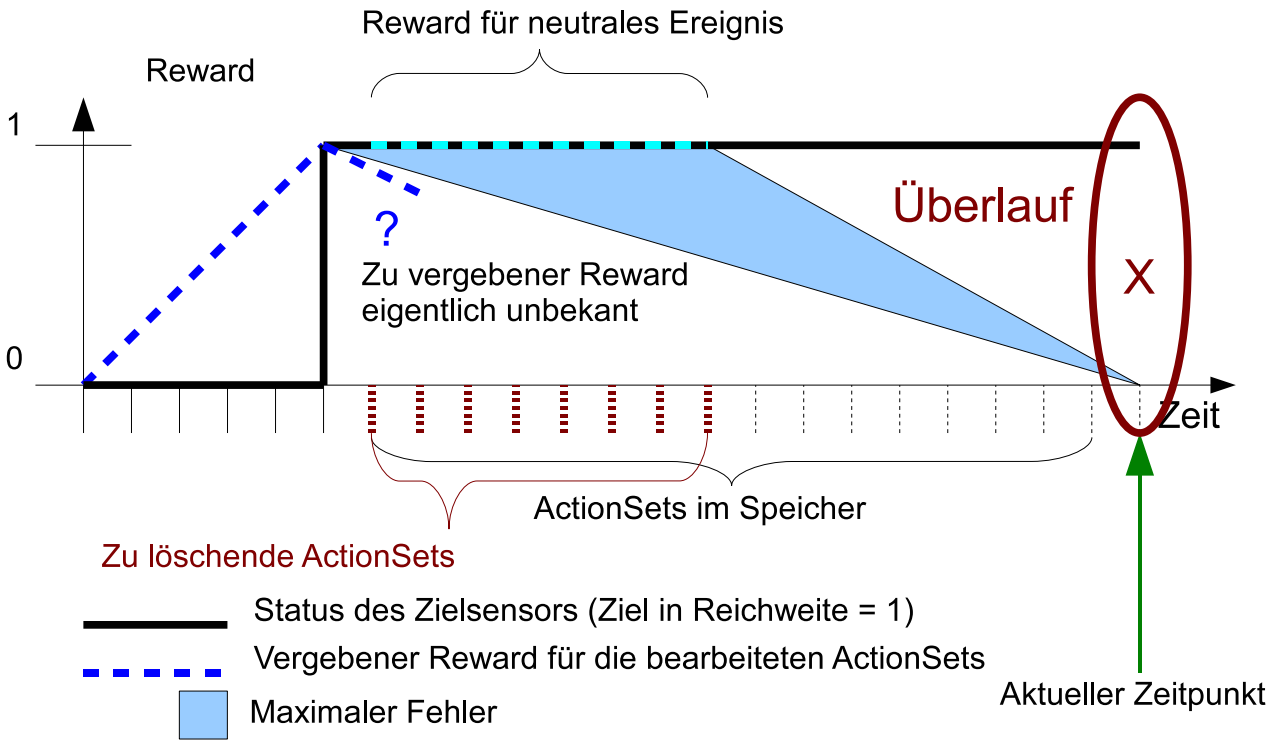

Abbildung 4.3: Schematische Darstellung der Bewertung von *action set* Listen bei einem neutralen Ereignis (mit *base reward* = 1)

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. Erstens gibt es die Verzögerung zu Beginn eines Problems 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, vor allem

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.

Wie Abbildung 4.4 zeigt, ist dies bei größerer Schrittzahl (2000 Schritte) aber vernachlässigbar, die erreichten Qualitäten unterscheiden sich im betrachteten Wertebereich kaum voneinander. Es gibt bei geringen Werten einen kleinen Anstieg, außerdem einen kleinen Abfall beim schwierigen Szenario. Da während der Entwicklung die meisten Tests mit dem Wert 128 durchgeführt wurden, wird dieser Wert belassen. Nur für das schwierige Szenario ist womöglich ein Wert von 64 vorzuziehen.

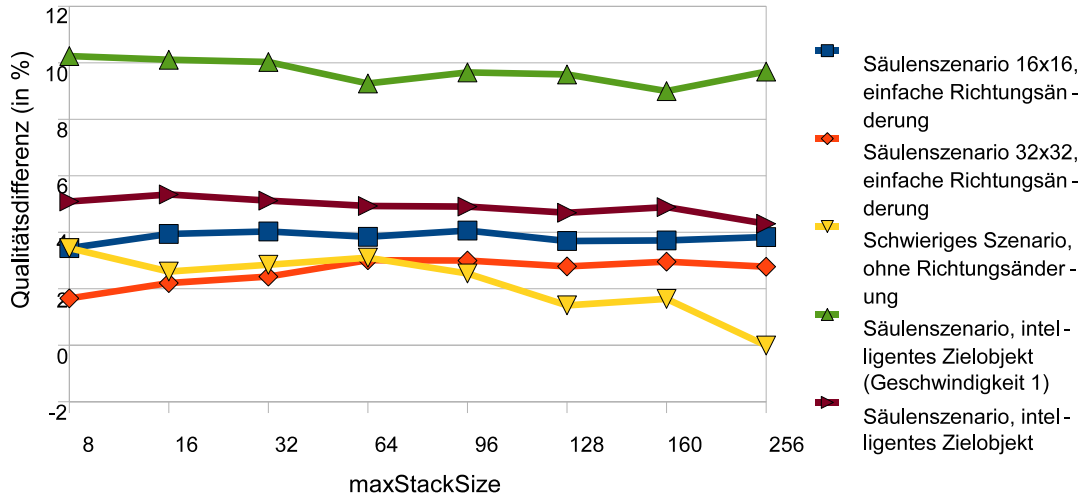

Abbildung 4.4: Vergleich verschiedener Werte für *maxStackSize* (2000 Schritte, SXCS Agenten)

4.3.4 Zusammenfassung der Ereignisse

Zusammengefasst ergeben sich also folgende Ereignisse:

Ein Ereignis tritt auf, wenn:

1. Änderung des *base reward* Werts von 0 auf 1 (Zielobjekt war im letzten Zeitschritt nicht in Überwachungsreichweite) ⇒ positives Ereignis
2. Änderung des *base reward* Werts von 1 auf 0 (Zielobjekt war im letzten Zeitschritt in Überwachungsreichweite) ⇒ negatives Ereignis
3. Überlauf des Stacks (kein positives oder negatives Ereignis in den letzten *maxStackSize* Schritten), Zielobjekt ist in Überwachungsreichweite ⇒ neutrales Ereignis (mit *base reward* = 1)
4. Überlauf des Stacks (kein positives oder negatives Ereignis in den letzten *maxStackSize* Schritten), Zielobjekt ist nicht in Überwachungsreichweite ⇒ neutrales Ereignis (mit *base reward* = 0)

4.3.5 Implementierung von SXCS

Im Wesentlichen entspricht die Implementierung von SXCS der bekannten Implementierung von XCS (siehe Kapitel 4.2). Unterschiede gibt es erstens bei Berechnung des *reward* Werts in der Funktion *calculateReward()* in Programm A.13, bei der zwischen zwei Fällen unterschieden wird. Zum einen gibt es die Behandlung negativer und positiver Ereignisse (Zeile 17-21) und zum anderen die Behandlung des Überlaufs des Stacks (Zeile 24-30), während bei der Implementierung von XCS in Programm A.10 in fast jedem Schritt unabhängig von Ereignissen eine Aktualisierung stattfindet. Zweitens gibt es Unterschiede in der Funktion *collectReward()* in Programm A.14. Dort 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.15) wurde lediglich die Behandlung des Stacks hinzugefügt (Zeile 39-43).

4.3.6 Zielobjekt mit XCS und SXCS

Wie bereits in Kapitel 2.5.6 erwähnt, soll hier eine Implementierung von XCS und SXCS für das Zielobjekt diskutiert werden. Der Grund für die Untersuchung liegt mehr darin, eine weitere Anwendungsmöglichkeit aufzuzeigen und XCS und SXCS nochmals zu vergleichen, anstatt konkrete neue Erkenntnisse zu gewinnen. Insbesondere handelt es sich hierbei nicht mehr um ein kollaboratives Multiagentensystem, da es zum einen nur ein einziges Zielobjekt im Szenario gibt und zum anderen die Aufgabe, anderen Agenten auszuweichen, durch Zusammenarbeit nicht besser gelöst werden kann (mal von hochentwickelten Verhaltensweisen abgesehen, bei der ein Zielobjekt das andere dadurch „rettet“, dass es einen Agent weglocken kann). Die Ergebnisse der Analyse folgt in Kapitel 5.5, auch hier ist der Ansatz von SXCS gegenüber dem von XCS überlegen.

Was die Implementierung betrifft ist sie, bis auf die Funktion *checkRewardPoints()*, für das Zielobjekt fast identisch. Dort wird ein positiver *base reward* Wert dann vergeben, wenn kein Agent in Sicht ist (während bei den Agenten ein positiver *base reward* Wert dann vergeben wurde, wenn das Zielobjekt in Sicht ist). Die einzige zweite Änderung ist in der Funktion *calculateNextMove()* (siehe Programm A.12 (XCS) bzw. Programm A.15 (SXCS)), bei der die zusätzliche Sprungeigenschaft des Zielobjekts hinzugefügt ist (siehe Kapitel 2.5).

4.4 SXCS Variante mit Kommunikation

Da ein Multiagentensystem betrachtet wird, stellt sich natürlich die Frage nach der Kommunikation. In der Literatur gibt es Multiagentensysteme, die auf Learning Classifier Systemen aufbauen, wie z.B. TODO Literatur. Alle Ansätze in der Literatur erlauben

jedoch globale Kommunikation, z.T. gibt es globale *classifier* auf die alle Agenten zurückgreifen können, z.T. gibt es globale Steuerung. TODO Verteilung des rewards an alle - soccer TODO Einordnen

soccer!

[THN+98] OCS, centralized control system

In dieser Arbeit soll ein Szenario ohne globale Steuerung oder globale *classifier*, also mit der Restriktion einer begrenzten, lokalen Kommunikation. Geht man davon aus, dass über die Zeit hinweg jeder Agent indirekt mit jedem anderen Agenten in Kontakt treten kann, Nachrichten also mit Zeitverzögerung weitergeleitet werden können, ist eine Form der globalen, wenn auch zeitverzögerten, Kommunikation möglich. TODO Eine spezielle Implementierung für diesen Fall werde ich weiter unten besprechen TODO

Bisher wurde der Fall betrachtet, dass Kommunikation mit beliebiger Reichweite stattfinden kann. Dies ist natürlich kein realistisches Szenario. Geht man jedoch 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 benötigt. Der Einfachheit halber soll deswegen angenommen werden, dass wir zwar globale Kommunikation zur Verfügung haben, jedoch diese u.U. zeitverzögert stattfindet und wir nur geringe Mengen an Information weitergeben können. In diesem Falle sollen ein Agent in jedem Schritt maximal lediglich einen sogenannten Kommunikationsfaktor und einen *reward* Wert an

alle anderen Agenten weitergeben können. Da dieser Algorithmus auf dem SXCS Algorithmus aufbauen soll, 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.

Insgesamt wird dafür also eine SXCS Variante benötigt, die mit zeitlich verzögerter Aktualisierung arbeiten kann.

weshalb für Kommunikation der zuvor besprochene verzögerte SXCS Algorithmus (DSXCS) in Frage kommt.

Hauptaugenmerk hier soll sein, dass es überhaupt Vorteile bringen kann, den *reward* weiterzugeben

TODO SWITCH EXPLORE/EXPLOIT + NEW LCS sehr gut

Einführung, Kommunikationsbeschränkungen (nur Reward weitergeben)

Vergleich Agentenzahl (1, 2, 3, 4, 5, 6, 7, 8)

reward all equally besser als reward none Unterscheidung interner und externer reward

Realistischer Fall mit Kommunikationsrestriktionen

pg. 286 Zentralisierung der Daten

TODO bei Faktorberechnung Ranking

Lösungen aus der Literatur

4.4.1 SXCS Variante mit verzögerter Reward (DSXCS)

Die Funktion *calculateReward()* ist identisch mit der in Kapitel 4.3.5 besprochenen Funktion (Programm A.13) bei der SXCS Variante ohne verzögerte Bewertung. Ebenso ist die

Funktion *calculateNextMove()* (siehe Programm A.17) fast identisch mit der dort besprochenen Funktion, nur bei der Behandlung des Stacks wird hier beim Überlauf der Eintrag nicht einfach gelöscht, sondern mit der Funktion *processReward()* zuerst noch verarbeitet. In der Funktion *processReward()* werden die gespeicherten *reward* und *factor* Werte ausgewertet. In der Implementation in Programm A.18 werden einfach alle nacheinander auf das *action set* angewendet, während in der verbesserten Version in Programm A.19 nur der *reward* Wert aus dem Paar mit dem größten Produkt aus den *reward* und *factor* Werten für die Aktualisierung benutzt wird. In beiden Implementationen werden außerdem Einträge mit sowohl einem *reward* als auch *factor* Wert von 1,0 ignoriert, sie wurden bereits in Programm A.16 ausgewertet.

Eine hilfreiche Voraussetzung für Kommunikation ist, wenn die dadurch möglicherweise entstehende Verzögerung vom jeweiligen Algorithmus unterstützt wird. Während weiter oben

Der wesentliche Unterschied zur ersten XCS Variante SXCS ist, dass jeglicher ermittelter *reward* Wert und der jeweils zugehörige Faktor lediglich erst einmal zusammen mit den jeweiligen *action set* Listein einer Liste (*historicActionSet* Liste) gespeichert werden und in jedem Schritt immer nur die *classifier* der *action set* Liste des ältesten Eintrags in der *historicActionSet* Liste aktualisiert werden. Somit ergibt sich also eine zeitlich beliebig verzögerbare Aktualisierungsfunktion, welche uns erlaubt, mehrere gleichzeitig stattgefundene (aber erst verzögert eintreffende, wegen z.B. Kommunikationsschwierigkeiten) Ereignisse zusammen auszuwerten. Dies ist macht die Auswertung der eingehenden *reward* Werte und Kommunikationsfaktoren wesentlich einfacher, da alle gemeinsam betrachtet werden können, anstatt dass sie sofort bei Eingang verarbeitet werden müssen.

Wann immer ein *base reward* Wert an einen Agenten verteilt wird, kann es sinnvoll sein, diesen *base reward* 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. Das in dieser Arbeite besprochene Szenario ist wesentlich komplexer, was d

Jeder Reward, der aus einem normalen Ereignis generiert wird, wird unter Umständen an alle anderen Agenten weitergegeben. Wie ein solches sogenanntes „externes Ereignis“ von diesen Agenten aufgefasst wird, hängt von den jeweiligen Kommunikationsvarianten ab, die weiter unten besprochen werden.

Durch eine gemeinsame Schnittstelle erhält jeder Agent den *reward* zusammen mit dem Kommunikationsfaktor. Dabei ergibt sich das Problem, dass sich *reward* überschneiden können, da jeder *reward* 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* 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 produziert bzw. weitergibt um die eigene Aktion als zielführend zu bewerten.

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.

TODO überlegen ob das noch Sinn macht, inwieweit das erklärt werden musws

Dies zeigt auch der Test: TODO

Ist kein Event aufgetreten und liegt ein 1-Reward vor, dann stellt sich die Frage, ob bereits andere Agenten diesen Reward weitergereicht haben. Befinden sich andere Agenten in Reichweite soll nur ein Agent den Reward weiterreichen. TODO Test

Abbildung 4.5

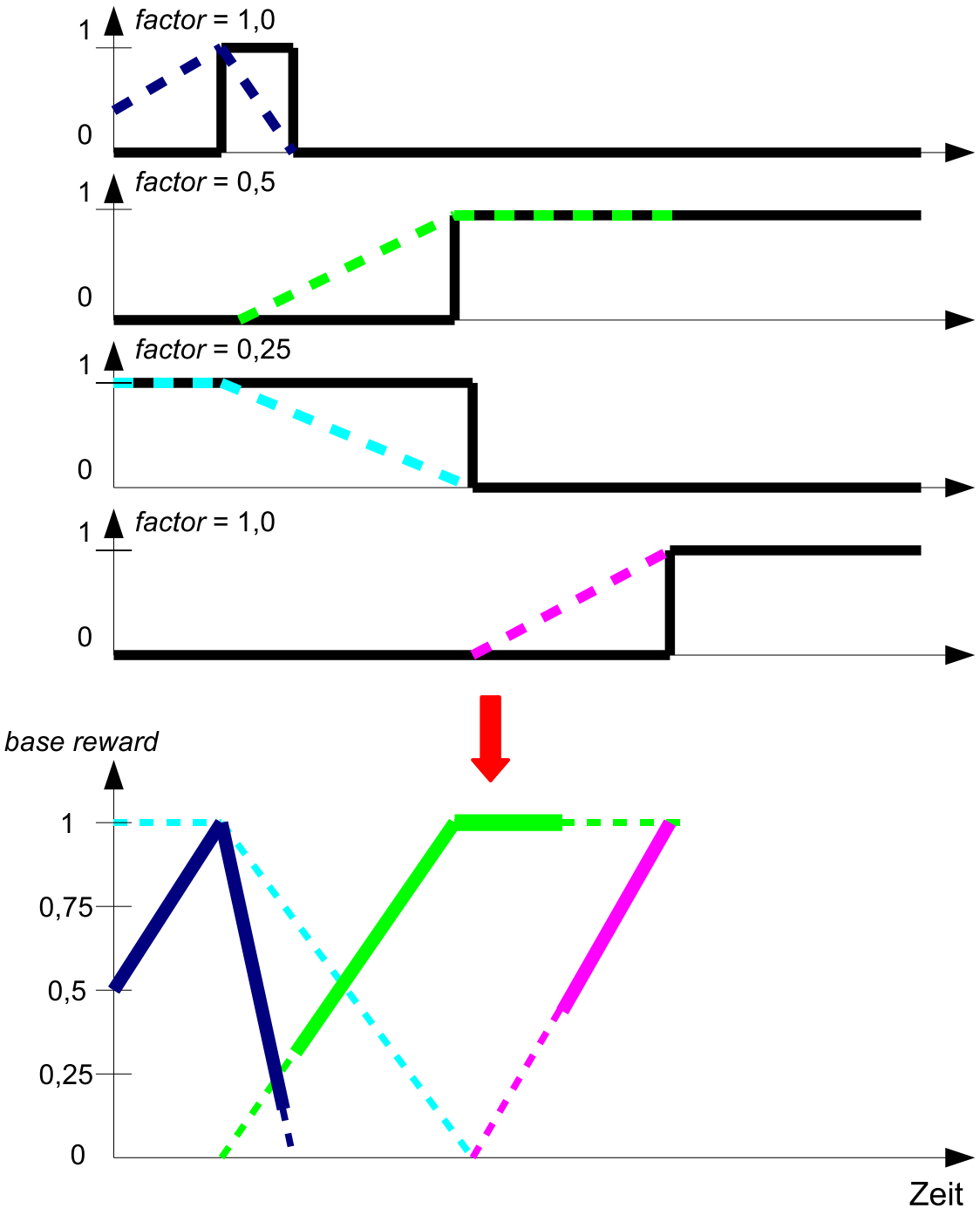

Abbildung 4.5: Beispielhafte Darstellung der Kombinierung interner und externer Rewards

Allen hier vorgestellten Kommunikationsvarianten ist gemeinsam, dass sie einen Kommunikationsfaktor berechnen, nach denen sie den externen Reward, den ihnen ein anderer Agent übermittelt hat, bewerten. Der Kommunikationsfaktor gewichtet alle Verwendungen des Parameters β (welcher die Lernrate bestimmt). Ein Faktor von 1.0 hieße, dass der externe *reward* Wert wie ein normaler *reward* Wert behandelt wird, ein Faktor von 0,0 hieße, dass externe *reward* Werte deaktiviert sein sollen.

Die Idee ist, dass unterschiedliche Agenten unterschiedlich stark am Erfolg des anderen Agenten beteiligt sind, da ohne Kommunikation jeder Agent versuchen würde, 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.

Hier sollen nun zwei Formen der Weitergabe des *reward* Werts vorgestellt werden, zum einen die Kommunikationsvariante in der alle Agenten ihre *reward* Werte teilen (siehe Kapitel 4.4.2) und zum anderen eine Kommunikationsvariante bei der der *reward* Wert anhand ähnlicher in den *classifier sets* gespeicherter Verhaltensweisen verteilt wird (siehe Kapitel 4.4.3.

4.4.2 Kommunikationsvariante „Einzelne Gruppe“

Mit dieser Variante wird der Kommunikationsfaktor 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.

Letztlich ist eine Zusammenlegung der Rewards im Grunde mit einer Zusammenlegung aller Sensoren zu vergleichen, Tatsächlich nur ein einzelner Agent?

In Tests (TODO) haben sich dennoch in bestimmten Fällen mit "Reward all equally" deutlich bessere Ergebnisse gezeigt als im Fall ohne Kommunikation. Dies ist wahrscheinlich darauf zurückzuführen, dass in diesen Fällen die Kartengröße und Geschwindigkeit des Zielobjekts relativ zur Sichtweite und Lerngeschwindigkeit zu groß war, die Agenten also annahmen, dass ihr Verhalten schlecht ist, weil sie den Zielobjekts relativ selten in Sicht bekamen. Eine Weitergabe des Rewards an alle Agenten kann hier also zu einer Verbesserung führen, dabei ist der Punkt aber nicht, dass Informationen ausgetauscht werden, sondern, dass obiges Verhältnis zugunsten der Sichtweite gedreht wird. Für die Auswahl geeigneter Tests sollten die Szenarioparameter also möglichst so gewählt werden, dass "Reward all equally" keinen signifikanten Vorteil gegenüber "No external reward" bringt. Blickt man auf diesen Sachverhalt aus einer etwas anderen Perspektive ist es auch einleuchtend. Es scheint offensichtlich, dass es relevant ist, ob das Spielfeld z.B. 100x100 oder nur 10x10 Felder groß ist, wenn es darum geht, das Verhalten über die Zeit hinweg zu bewerten. In den Algorithmus für die Kommunikation bzw. für die Rewardvergabe müsste man deshalb einen weiteren (festen) Faktor einbauen, der zu Beginn in Abhängigkeit von Größe des zu überwachenden Feldes berechnet wird. Dies soll aber nicht Teil der Arbeit werden. TODO

TODO Idee: Verteilt man den Reward an alle Agenten mit gleichem Faktor heisst das letztlich, dass jeder Agent in jedem Zeitschritt den selben Rewardwert erhält. Dann bildet das System der Agenten im Grunde als gemeinsames System von Agenten mit

gemeinsamen Sensoren und gemeinsame, ClassifierSet TODO

4.4.3 Gruppenbildung über Ähnlichkeit des Verhaltens der Agenten

In [KM94] wurde gezeigt, dass Gruppenbildung (rationality, grade 2 confusion)

Eine weitere 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 versucht, das Zielobjekt in Sicht zu bekommen, anstatt ein möglichst großes Gebiet abzudecken.

Die Hypothese ist, dass Agenten mit ähnlichem Egoismusfaktor auch einen ähnlichen Classifiersatz besitzen und der Reward nicht an alle Agenten gleichmäßig weitergegeben wird, sondern bevorzugt an ähnliche Agenten.

Damit gäbe es einen Druck in Richtung eines bestimmten Egoismusfaktors. TODO

Der Vorteil gegenüber den anderen Verfahren liegt darin, dass der Kommunikationsaufwand hier nur minimal ist, neben dem *reward* muss lediglich der Egoismus Faktor übertragen und pro Zeitschritt nur einmal berechnet werden.
Ein Problem dieser Variante kann sein, dass der Ansatz das Problem selbst schon löst, indem er kooperatives Verhalten belohnt, unabhängig davon, ob Kooperation für das Problem sinnvoll ist.

Die Variante müsste also zum einen in

schlecht abschneiden TODO

Die offensichtliche Unstetigkeit der in dieser Weise verarbeiteten *reward* Werte gibt Raum für Verbesserungen.

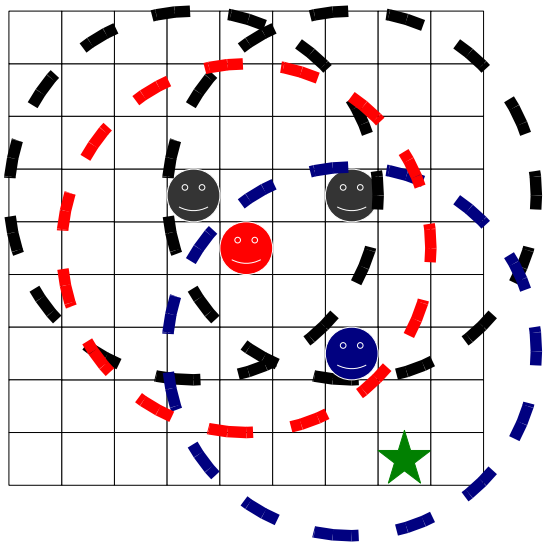

Abbildung 4.6: Schematische Darstellung der Bewertung von action set Listen bei einem neutralen Ereignis

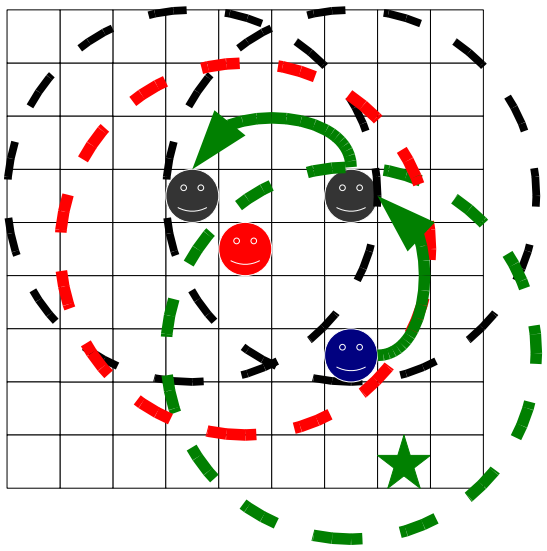

Abbildung 4.7: Schematische Darstellung der Bewertung von action set Listen bei einem neutralen Ereignis

Die Berechnung des Faktors ist in Programm A.20 dargestellt. Für jede *classifier set* Liste wird ein sogenannter Egoismusfaktor bestimmt, die Differenz beider Egoismusfaktoren wird dann im Quadrat von 1,0 abgezogen und als Kommunikationsfaktor zurückgegeben. Die Egoismusfaktoren 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 zubewegen, sofern sie in Sicht sind. Somit ist der Kommunikationsfaktor umso größer, je ähnlicher die Agenten in ihrem Abstandsverhalten gegenüber anderen Agenten sind.

Kapitel 5

Analyse SXCS

5.1 Erste Analyse der Agenten ohne XCS

In diesem Abschnitt sollen 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 werden. Die Ergebnisse aus der Analyse werden eine Grundlage für die vergleichende Betrachtung der Agenten mit XCS Algorithmen in Kapitel 5 dienen, insbesondere werden sie Anhaltspunkte dafür geben, welche Szenarien welche Eigenschaften der Algorithmen testen. Außerdem kann 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.

TODO erwähnen, dass 8 Agenten ok sind

5.1.1 Zielobjekt mit zufälligem Sprung

Im folgenden sollen alle TODO

In allen Szenarien mit dieser Form der Bewegung des Zielobjekts kommt es nur darauf an, dass die Agenten einen möglichst großen Bereich des Torus abdecken.

5.1.2 Im leeren Szenario ohne Hindernisse

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, was sich auch in einem hohen Wert der Abdeckung zeigt, denn genau das wird mit dem völlig zufällig springenden Agenten getestet. Auch 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 Sichtbarkeitsüberprüfung für die Richtung durchführt, in der sie sich bewegen möchte (nämlich wenn das Zielobjekt in Sicht ist), andererseits ist die Abdeckung etwas geringer. Dies kommt daher, dass, wenn mehrere Agenten das Zielobjekt in derselben Richtung in Sichtweite haben, mehrere Agenten sich in dieselbe Richtung bewegen. Dies beeinträchtigt die zufällige Verteilung der Agenten auf dem Spielfeld und führt somit auch zu einer niedrigeren Abdeckung des Torus.

Bezüglich der Anzahl der Agenten ergeben sich keine Besonderheiten, mit steigender Agentenzahl steigt die Zahl der blockierten Bewegungen (aufgrund größerer Anzahl von blockierten Feldern), während die Abdeckung sinkt (aufgrund sich überlappender Über-

wachungsreichweiten).

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%

5.1.3 Säulenszenario

Für das Säulenszenario (siehe Tabelle 5.2) ergeben sich erwartungsgemäß 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, im Folgenden werden sich die Tests deshalb auf den Fall mit **8 Agenten** beschränken.

5.1.4 Zufällig verteilte Hindernisse

Hier ergibt sich für alle Einstellungen für λ_h und λ_p (siehe Kapitel 2.2.2) ebenfalls ein eindeutiges Bild (siehe Tabelle 5.3), 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

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%

wieder oben genannte Reihenfolge.

Dass der einfache Agent, wenn er das Zielobjekt in Sicht hat, eine geringere Zahl an blockierten Bewegungen als der zufällige Agent aufweist, lässt sich damit begründen, dass er davon ausgehen kann, dass sich in dieser Richtung wahrscheinlich eher kein Hindernis befindet (da die Sicht nicht blockiert ist), während der zufällige Agent Hindernisse überhaupt nicht beachtet, somit öfters gegen ein Hindernis läuft und letztlich öfters stehen bleibt. Der Unterschied zwischen beiden Agenten ist besonders hoch in Szenarien mit größerem Anteil an Hindernissen.

Im Vergleich zur einfachen Heuristik 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, dass die Agenten gegen die Hindernisse gedrückt werden (andere Agenten sind bei hohem Verknüpfungsfaktor eher in einem Bereich ohne Hindernisse).

Schließlich ist zu sehen, dass die Agenten in einem Szenario mit höherem Verknüpfungs-

faktor (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 keine der Szenarien mit zufälligem Sprung des Zielobjekts sich als zu lernende Aufgabe lohnt, der Unterschied zwischen der zufälligen Bewegung und der intelligenten Heuristik ist zu gering, die Aufgabe somit zu schwierig und soll in Verbindung mit XCS nicht weiter betrachtet werden.

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

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%

5.1.5 Zielobjekt mit zufälliger Bewegung bzw. einfacher Richtungsänderung

Wesentlicher Punkt bei beiden Bewegungstypen (siehe Kapitel 2.5.2 und Kapitel 2.5.3) ist, dass der jetzige Ort des Zielobjekts maximal zwei Felder (die maximale Geschwindigkeit des Zielobjekts in den Tests) vom Ort in der vorangegangenen Zeiteinheit entfernt

ist. Somit ist ein lokales Einfangen eher von Relevanz, der Ort an dem sich das Zielobjekt im nächsten Zeitschritt befinden wird, ist zumindest vom aktuellen Ort abhängig, wenn das Zielobjekt auch schneller sein kann als andere Agenten.

Wesentlicher Unterschied zwischen beiden Bewegungstypen ist, dass das Zielobjekt mit zufälliger Bewegung nach 2 Schritten mit Wahrscheinlichkeit von $\frac{1}{4}$ auf das ursprüngliche Feld zurückkehrt, also stehenbleibt. Wie die Ergebnisse in Tabellen 5.5 und 5.6 zeigen, ergibt sich dadurch ein leichteres Szenario. Ein mitunter stehengebliebener Agent kann mittels Heuristiken leichter überwacht werden, während es keine signifikante Veränderung bei der zufälligen Bewegung ergibt. In weiteren Tests soll deswegen immer nur Zielobjekten mit einfacher Richtungsänderung getestet 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 (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%

Tabelle 5.5: 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 5.6: 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%

5.2 Auswirkung der Geschwindigkeit des Zielobjekts

Angesichts der Ergebnisse in den zwei vorangegangenen Kapiteln, ist zu erwarten, dass die Geschwindigkeit des Zielobjekts bei der Qualität des Agenten mit zufälliger Bewegung keine Rolle spielt, da weder das Zielobjekt noch die Agenten Informationen über ihre Umgebung benutzen um sich für ein Verhalten zu entscheiden.

TODO subsections zusammenfassen

5.2.1 Zielobjekt mit einfacher Richtungsänderung

In Abbildung 5.1 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 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.
Der Algorithmus mit zufälliger Bewegung stellt also eine Untergrenze dar, ein Agent muss mehr als diesen Wert erreichen, damit man sagen kann, dass er etwas gelernt hat.

In Abbildung 5.2 sind dagegen die Testergebnisse (im selben Szenario) für die einfache und die intelligente Heuristik zu sehen. Im Wesentlichen sind drei Punkte anzumerken, erstens existiert eine Korrelation zwischen Qualität und Geschwindigkeit, zweitens gibt es einen Knick bei Geschwindigkeit 1 und drittens ist ein fast stetiger Anstieg der Differenz zwischen der einfachen und der intelligenten Heuristik zu verzeichnen. Der Knick lässt sich dadurch erklären, dass es ab dieser Geschwindigkeit möglich ist, dass das Zielobjekt Verfolger abschütteln kann, der Anstieg der Differenz lässt sich dadurch erklären, dass es Abdeckung des Gebiets eine immer größere Rolle spielt, als die Verfolgung des Zielobjekts.

5.2.2 Zielobjekt mit intelligenter Bewegung

In Abbildung 5.3 und Abbildung 5.4 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. Im Fall mit TODO

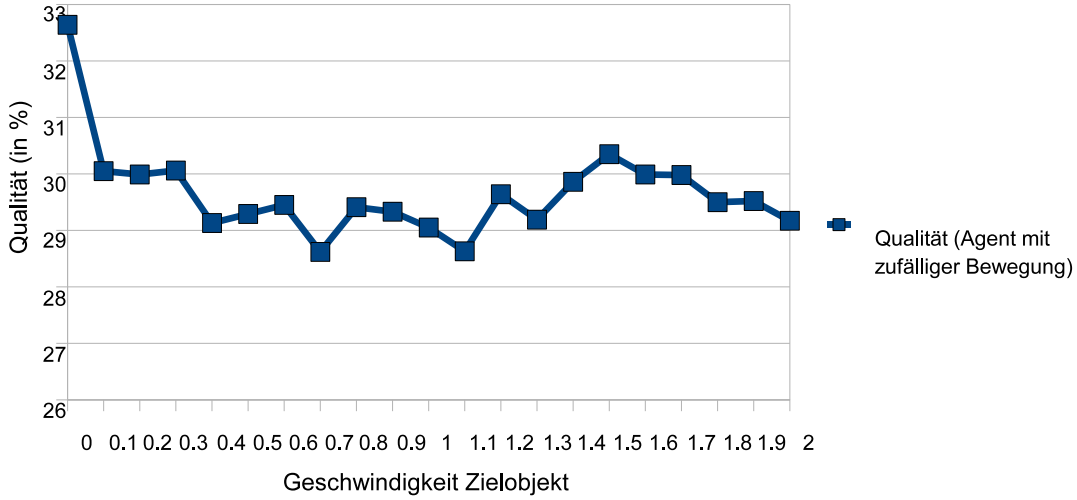

Abbildung 5.1: Auswirkung der Zielgeschwindigkeit auf Agenten mit zufälliger Bewegung (Zielobjekt mit einfacher Richtungsänderung, Säulenszenario)

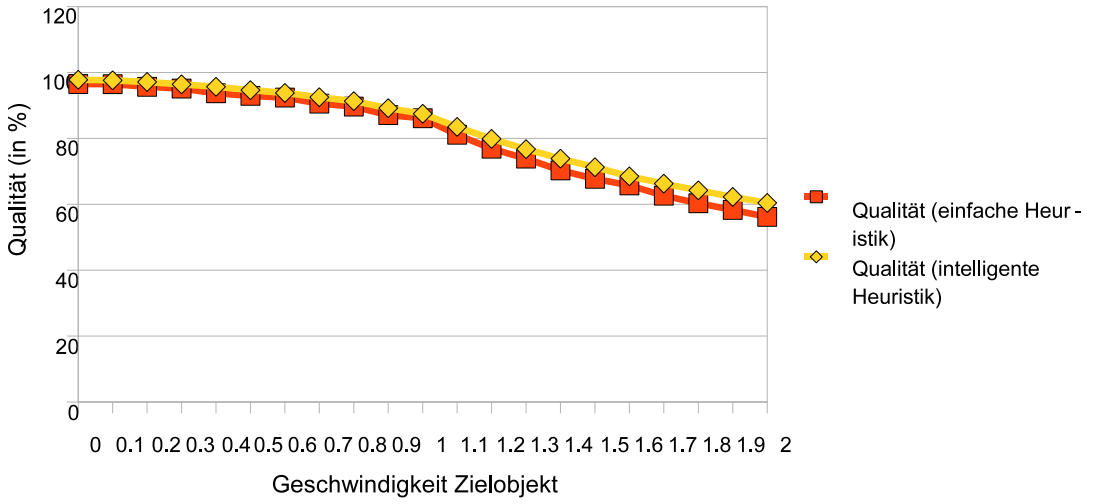

Abbildung 5.2: Auswirkung der Zielgeschwindigkeit auf Agenten mit bestimmten Heuristiken (Zielobjekt mit einfacher Richtungsänderung, Säulenszenario)

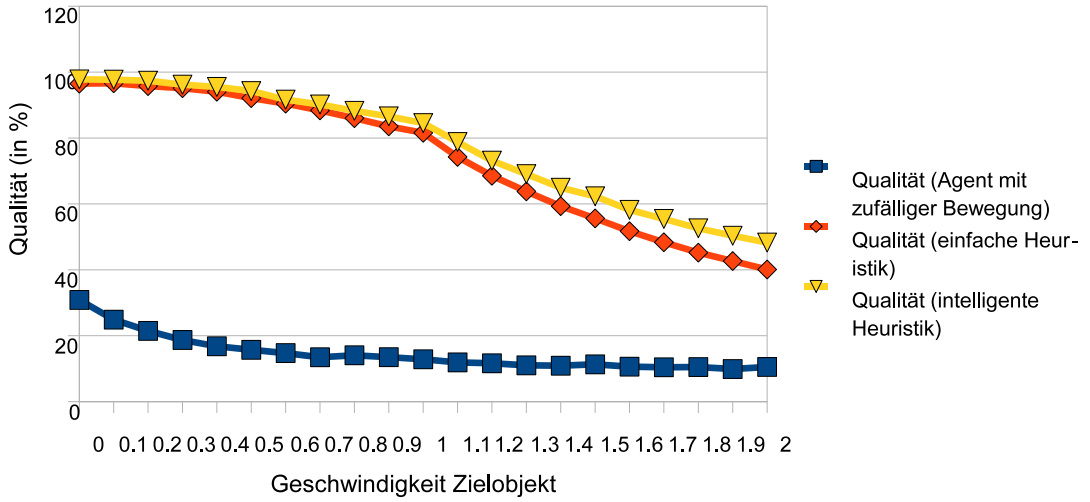

Abbildung 5.3: Auswirkung der Zielgeschwindigkeit (intelligentes Zielobjekt) auf Agenten mit Heuristik

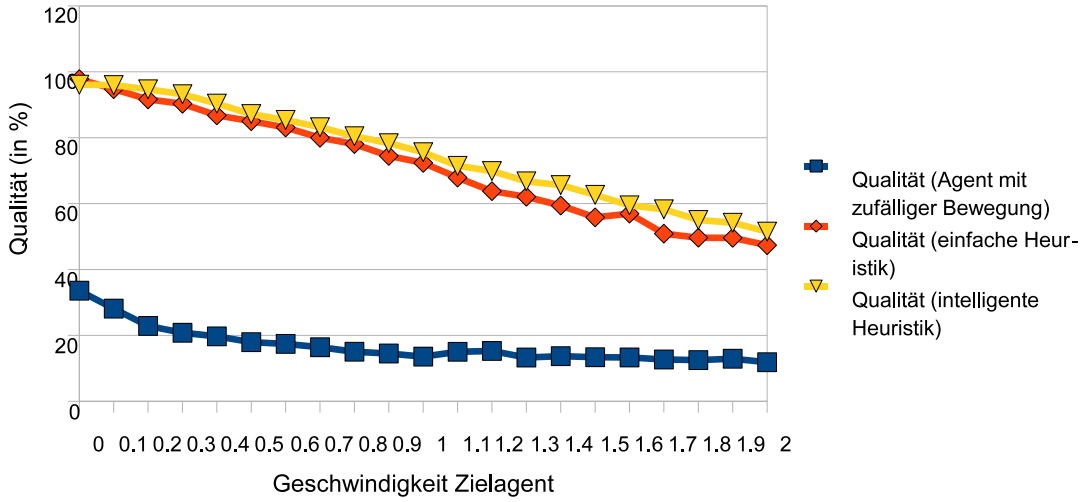

Abbildung 5.4: Auswirkung der Zielgeschwindigkeit (intelligentes Zielobjekt, Szenario mit zufällig verteilten Hindernissen, $\lambda_h = 0{,}2$, $\lambda_p = 0{,}99$) auf Agenten mit Heuristik

TODO: Erläuterung!

5.2.3 Schwieriges Szenario

Für das sogenannte schwierige Szenario aus Kapitel 2.2.4 erscheint nur der in Kapitel 2.5.5 vorgestellte Typ von Zielobjekt mit Beibehaltung der Richtung sinnvoll, da das Ziel für die Agenten sein soll, bis in den letzten Abschnitt vorzudringen und dem Zielobjekt nicht schon auf halbem Weg zu begegnen.
Für verschiedene Anzahl von Schritten sind für die drei Agententypen in Abbildung 5.5 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 5.7.4 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.

5.2.4 Zusammenfassung

Wie man sehen konnte, existieren also Szenarien in denen Abdeckung kaum eine Rolle spielt und lokale Entscheidungen eine wesentliche Rolle spielen. Dies wird es erleichtern, geeignete Szenarien im Kapitel 4.4 zu finden.

TODO

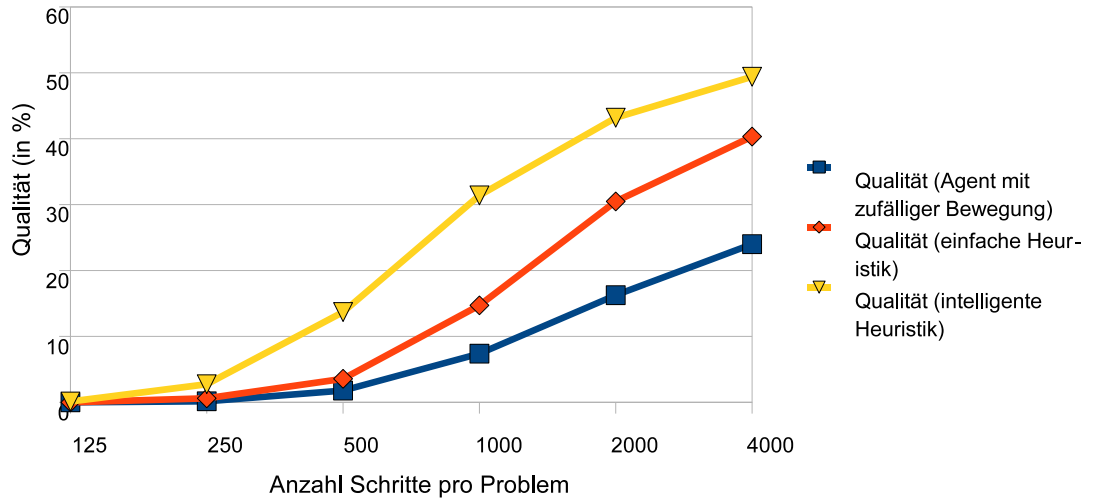

Abbildung 5.5: Auswirkung der Anzahl der Schritte (schwieriges Szenario, Geschwindigkeit 2, ohne Richtungsänderung) auf Qualität von Agenten mit Heuristik

5.3 Test der verschiedenen XCS Auswahlarten

In Tabelle 5.7 kann man die bisherigen Vermutungen sehr gut erkennen. Die Auswahlarten *random selection* und *roulette wheel selection* sind für sich alleine kaum brauchbar, das Ergebnis ist nicht besser als des sich zufällig bewegenden Agenten, für den in Kapitel 5.2.1 festgestellt wurde, dass unabhängig von der Geschwindigkeit des Zielobjekts die Qualität bei etwa 30% liegt.
Die Auswahlart *best selection* sorgt gar für über 40% blockierte Bewegungen und einer deutlich schlechteren Abdeckung. Für die *exploit* Phase scheint nur *tournament selection* deutlich bessere Ergebnisse zu liefern, wenn auch mit relativ hoher Zahl blockierter Bewegungen. Da die *roulette wheel* Auswahlart etwas bessere Ergebnisse liefert, soll sie für die *explore* Phase benutzt werden.

Für den Wechsel zwischen der *explore* und *exploit* Phase sieht man bei zufälligem Wechsel, dass die statistischen Werte zwischen denen der *roulette wheel* und *tournament selection* Auswahlart liegen, stellt angesichts der minimalen Steigerung zur Qualität von

der *roulette wheel* Auswahlart also kein signifikante Verbesserung dar. Wechselt man bei einer Änderung des *base reward* Werts und startet in der *explore* Phase ergibt sich ein deutlich schlechteres Ergebnis, der Algorithmus scheint sich also genau falsch zu verhalten. Umgekehrt, startet man in der *exploit* Phase, ergibt sich dagegen ein deutlich besseres Ergebnis. Über die Gründe kann man spekulieren. Da die meisten Ergebnisse der Tests eine Qualität von unter 50% erreichten, 8 Agenten benutzt wurden und in Tests sich gezeigt hat, dass deren Abdeckung etwa 70% betrug (also im Durchschnitt etwa 30% des maximal überwachbaren Gebiets verschwendet war), ist anzunehmen, dass der einzelne Agent das Zielobjekt sehr selten in Sichtweite bekam. Der Wechsel zur *explore* Phase erlaubt also anscheinend einen Ausgleich zwischen der unverhältnismäßig langen Zeit, in der das Zielobjekt nicht in Sicht ist und der Zeit, in der das Zielobjekt in Sicht ist. Vermutlich würde dieser Vorteil bei einer größeren Anzahl von Zielobjekten verschwinden.

Desweiteren ist zum Vergleich wichtig, wie die Situation beim XCS Algorithmus hinsichtlich der Auswahlverfahren ist. Die Ergebnisse in Tabelle 5.8 demonstrieren, dass hier nur die reine *exploit* Phase einen positiven Effekt, relativ zum Agenten mit zufälliger Bewegung, bringt.

Insgesamt soll also im Weiteren für nur die *tournament selection* und der Wechsel zwischen *tournament selection* und *roulette wheel selection* als Auswahlart benutzt werden.

TODO neu!?

Tabelle 5.9

TODO mit obigen Tabellen vergleichen!

TODO ges2, int 1 und 2

XCS + switch wirkungslos!

TODO discuss Kapitel 3.4.4 exploit+intelligent besser als switch explore/exploit

Tabelle 5.7: Vergleich der verschiedenen Auswahlarten (Zielobjekt mit einfacher Richtungsänderung, Säulenszenario, Geschwindigkeit 1, Agenten mit SXCS Algorithmus)

Auswahlart	Blockierte Bewegungen	Abdeckung	Qualität
Agent mit zufälliger Bewegung	4,46%	72,12%	29,05%
roulette wheel selection	4,54%	72,10%	30,30%
random selection	4,34%	72,21%	28,50%
tournament selection	11,21%	70,20%	**33,39%**
best selection	41,16%	63,64%	29,22%
Zufällig *explore/exploit*	6,29%	71,18%	30,58%
Abwechselnd, zuerst *explore*	5,63%	71,37%	26,30%
Abwechselnd, zuerst *exploit*	9,28%	70,40%	**35,36%**

Tabelle 5.8: Vergleich der verschiedenen Auswahlarten (Zielobjekt mit einfacher Richtungsänderung, Säulenszenario, Geschwindigkeit 1, Agenten mit XCS Algorithmus)

Auswahlart	Blockierte Bewegungen	Abdeckung	Qualität
Agent mit zufälliger Bewegung	4,46%	72,12%	29,05%
roulette wheel selection	4,54%	72,10%	30,30%
random selection	4,34%	72,21%	28,50%
tournament selection	11,21%	70,20%	**33,39%**
best selection	41,16%	63,64%	29,22%
Zufällig *explore/exploit*	6,29%	71,18%	30,58%
Abwechselnd, zuerst *explore*	5,63%	71,37%	26,30%
Abwechselnd, zuerst *exploit*	9,28%	70,40%	**35,36%**

5.4 Auswirkung unterschiedlicher Geschwindigkeiten des Zielobjekts

In Abbildung 5.6 ist ein Vergleich der unterschiedlicher Geschwindigkeiten des Zielobjekts dargestellt. XCS (mit 500 Schritten) macht bei keiner Geschwindigkeit Lernfortschritte, die Qualität pendelt zwischen 31,69% und 33,40%, also in etwa identisch mit der zufälligen Bewegung. Die SXCS Implementierung scheint dagegen die geringere Geschwindigkeit ausgenutzt zu haben und ist dadurch in der Lage das Zielobjekt besser zu verfolgen. Mit 500 Schritten ist die Qualität abnehmend von 39,64% (Geschwindigkeit 1,0) bis 35,96%

Tabelle 5.9: Vergleich der verschiedenen Auswahlarten (Zielobjekt mit einfacher Richtungsänderung, Säulenszenario, **Geschwindigkeit 2**, Agenten mit SXCS Algorithmus)

Auswahlart	Blockierte Bewegungen	Abdeckung	Qualität
Agent mit zufälliger Bewegung	4,46%	72,12%	29,05%
roulette wheel selection	4,54%	72,10%	30,30%
random selection	4,34%	72,21%	28,50%
tournament selection	11,21%	70,20%	**33,39%**
best selection	41,16%	63,64%	29,22%
Zufällig *explore/exploit*	6,29%	71,18%	30,58%
Abwechselnd, zuerst *explore*	5,63%	71,37%	26,30%
Abwechselnd, zuerst *exploit*	9,28%	70,40%	**35,36%**

(Geschwindigkeit 2.0), im Fall mit 2000 Schritten erhöht sich dieser Bereich leicht auf 40,15% bis 37,71%.

Auch bei den Heuristiken zeichnet sich ein klares Bild ab, bei niedrigen Geschwindigkeiten ist die Ausbreitung der Agenten auf dem Feld (intelligente Heuristik) weniger wichtig als die konstante Verfolgung des Zielobjekts, während bei höheren Geschwindigkeiten die Verteilung auf dem Feld wichtiger wird.

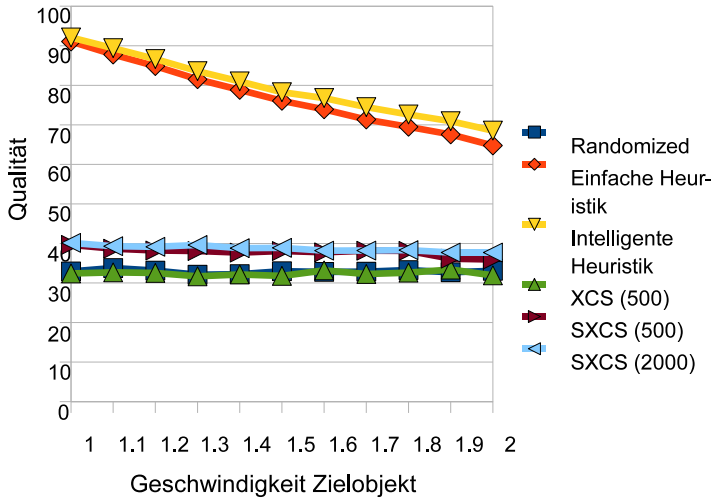

Abbildung 5.6: Vergleich der Qualitäten verschiedener Algorithmen bezüglich der Geschwindigkeit des Zielobjekts

```
Bester Agent nach 20000 Schritten
(Zielgeschwindigkeit 2.0, SXCS, 2000 Schritte)

#0######.###0#0##.#0#0###0-S : Fi: 38\% Ex: 450 Pr: 0,74 PE: 38\%
....
TODO
```

5.5 Zielobjekt mit XCS und SXCS

Der in Kapitel 4.3.6 besprochene Ansatz, einen Zielobjekt mit einem XCS bzw. SXCS auszustatten, soll hier nun getestet werden. Dabei sollen zuerst Agenten mit den besprochenen statischen Heuristiken gegen diesen Zielobjekt antreten, abschließend soll - hauptsächlich aus Neugier - ein solches Zielobjekt gegen ebenfalls mit SXCS ausgestatteten Agenten antreten, hier soll insbesondere der Verlauf über die Zeit interessieren, da die Qualität TODO

Wichtig: niedrige Lerngeschwindigkeit! evtl viele Probleme testen

5.6 Zusammenfassung der bisherigen Erkenntnisse

Algorithmen mit Ergebnissen die unter dem des zufälligen Algorithmus liegt, sind unbrauchbar und nicht vergleichbar. “Verbesserungen”, die die Qualität des Algorithmus näher an das Ergebnis des zufälligen Algorithmus bringen, sind in Wirklichkeit Veränderungen, die den Algorithmus eher zufällige Entscheidungen treffen lassen, und keine tatsächlichen Lernerfolge.

5.7 Von den Agenten nicht schaffbare Szenarien TODO Titel

Hindernisse 2-99

TODO im schwierigen Szenario, Intelligent speed 2... weder XCS noch SXCS schaffen es. ?

Schwieriges Szenario: Populationsgrößen mit und ohne zufälliger Initialisierung (covered actions)? Lernrate XCS/SXCS, 0,01-0,1

auch mit Geschwindigkeit 0,1 oder so testen - pillar, 1 direction change, speed 1, max pred + GA, 0,01 prediction init, prediction init adaption - nur gering gg random

- pillar, intelligent, low speed, maxpred aus, GA, SWITCH EXPLOIT, SEHR GUT!

Säulenszenario random, simple, intelligent, xcs, sxcs 500, sxcs 2000

DSXCS Konvergenzgeschwindigkeit vergleichen! Wenige Schritte schwieriges Szenario DSXCS besser mit geringem max stack size... TODO max stack size test für dsxcs

SXCS sehr gut bei NO DIRECTION CHANGE und speed 1!

nicht geschafft: Pillar, one direction change, speed 2, XCS besser... weil zufälliger

TODO auch sich langsam bewegende analysieren! Und auch stehenbleibende : z.B. im Raumszenario.

Geschwindigkeit 2 problematisch, Geschwindigkeit 1 ok?

TODO classifier ausgeben

SXCS sehr gut bei NO DIRECTION CHANGE und speed 1!

nicht geschafft: Pillar, one direction change, speed 2, XCS ...besser... weil zufälliger

deutlich schneller als intelligenter weil der nicht lernt

5.7.1 SXCS und Heuristiken

In den Tests erreichten die Heuristiken deutlich bessere Ergebnisse. Diesen Nachteil hat sich XCS in diesen Szenarien durch deutlich überlegene Flexibilität erkauft TODO

Ein Großteil der eingehenden Informationen ist für die Auswertung nicht relevant und lokale Information ist zu ungenau. Bei einer komplexeren Implementierung mit Distanzen

Insbesondere der Vergleich mit dem intelligenten Agenten, der anderen Agenten ausweicht, zeigt, dass die SXCS Agenten unmöglich ein solches globales Ziel erreichen können, es ist also kein emergentes Verhalten zu beobachten. Dies ist dadurch zu begründen, dass bei der Berechnung des Rewards keine Information außer der eigenen, lokalen Information

der Abstand zu anderen Agenten nicht Teil der Berechnung des Rewards ist, noch gibt keine eingebaute Heuristik. Man könnte zwar

5.7.2 Test Kommunikation

5.7.3 Bewertung Kommunikation:

Die Vorteile, die man durch Kommunikation erzielen kann, hängt stark von dem Szenario ab. Beispielsweise in dem Fall, bei dem zufällige Agenten bereits fast 100% Abdeckung erreichen, also so viele Agenten auf dem Feld sind, dass der Gewinn durch Absprache minimal ist. Auch ist, weil nur mit Binärsensoren gearbeitet wird, die Sensorik gestört, wenn sich sehr viele Agenten auf dem Feld befinden, weil die Sensoren sehr oft gesetzt sind und somit wenig Aussagekraft haben. Erweiterungen wie zusätzliche Sensoren die die genauen Abstände bestimmen, würde hier wahrscheinlich klarere Ergebnisse liefern.

Umgekehrt ist der Einfluss bei sehr wenigen Agenten gering. TODO Vergleich unterschiedliche Agentenanzahl, unterschiedliche Kommunikationsmittel Vergleich mit LCS?

5.7.4 XCS im schwierigen Szenario

Im schwierigen Szenario wurde in Kapitel 5.2.3 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 sollen nun lernende Agenten ihre Fähigkeiten unter Beweis stellen. Der wesentliche Vorteil der lernenden Agenten hier ist, dass sie ihr gelerntes über die, wie bisher, 10 Probleme behalten können und somit, sofern sie das Richtige gelernt haben, direkt auf den letzten Abschnitt durch die Öffnungen laufen können.
Auch soll 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 das letzte Feld erreicht, ist das Problem im Grunde schon gelöst. Die Hauptschwierigkeit ist, das letzte Feld zu erreichen.

XCS 125 - 4000

LCS Agenten schneiden auch ohne Kommunikation (bei ausreichender Anzahl von Schritten) immer besser ab als zufällige Agenten.

TODO

Anzumerken war, dass

TODO explore/exploit, nur bei explore lernen, erwähnen dass immer gelernt werden muss

TODO oberes Limit der Genauigkeit, 84%, da ja geswitched wird.

Geringen Unterschied ansprechen, mit zufälligem Algorithmus argumentieren, vielleicht ein BEispiel rechnen wo Qualität des zufälligen abgezogen wird!

Kapitel 6

Zusammenfassung, Ergebnis und Ausblick

Zu Beginn wurde auf die Szenariodefinition und die Fähigkeiten der Agenten eingegangen. Anhand von Beispielen heuristischer Agenten wurden einige Grundeigenschaften der präsentierten Szenarien als Vorbereitung für die Analyse der Learning Classifier Systeme bestimmt. Nach der Einführung in LCS, der Beschreibung des Standardverfahren XCS und der angepassten Implementierung für Überwachungsszenarios konnten dann umfangreiche Tests ausgeführt werden.

von der Möglichkeit zur Kommunikation eine angepasste Implementierung für verzögerten Reward definiert auf Basis dessen dann mehrere Varianten für die Weitergabe des Rewards vorgestellt, analysiert und verglichen wurden.

6.1 Ergebnis TODO

Es wurde gezeigt, welche Änderungen und Anpassungen am XCS Algorithmus durchgeführt werden müssen, um ihn auf dem Überwachungsszenario laufen zu lassen. Insbesondere die Problemdefinition spielte eine Rolle, da sie den wesentlichen Unterschied zu

den üblichen statischen Szenarien bildete.

empfindlich gegenüber Parameteränderungen

Das wesentliche Ergebnis ist, dass die Implementierung des XCS auf Überwachungsszenarios ausgeweitet werden kann ohne wesentliche Veränderungen am Algorithmus vorzunehmen. Während sich die Qualität der resultierenden Agenten im Allgemeinen über dem zufälligen Agenten befindet, ist die Effizienz der Implementierung, im Vergleich zu einfachen Heuristiken, sehr gering. Mit der verwendeten Implementierung hat XCS Probleme, eine optimale Regelmenge zu finden bzw. zu halten. Eine Regel wie z.B. „laufe auf das Ziel zu, wenn es in Sicht ist“, ist als Heuristik sehr erfolgreich, bei dauerhafter Überwachung ohne Kommunikation läuft es aber eher auf ein Verfolgungsszenario hinaus. Aufgrund andauerndem Lernens TODO

Die alleinige Anpassung des XCS Multistepverfahrens, dass ein neues Problem gestartet wird, wann immer sich das Ziel in Überwachungsreichweite befand führte nicht zum Erfolg, die Ergebnisse waren nicht besser als ein sich zufällig bewegender Agent.

Erst durch Verknüpfung der Bewertung (dem *base reward*) mit dem zeitlichen Abstand zu einer Änderung des Zustands führte zu deutlich besseren Ergebnissen.

TODO Desweiteren wurde untersucht, inwiefern sich der Austausch an minimaler Information unter den Agenten, ohne zentrale Steuerung oder globalem Regeltausch, auf die Qualität auswirkt. Zwar gab es vereinzelt positive Effekte, diese waren jedoch auf andere Faktoren zurückzuführen.

6.2 Ausblick und verworfene Ansätze

Mit dieser Arbeit wurde versucht, ein neues Gebiet von Problemfeldern in Verbindung mit dem XCS Algorithmus zu öffnen. Dies erlaubt eine ganze Reihe von fortführenden Untersuchungen, die im Folgenden kurz diskutiert werden sollen. Außerdem sollen Ansätze vorgestellt werden, die ausprobiert, aber während der Entwicklung wieder verworfen wurden. Möglicherweise würden diese Ansätze durch eine andere Herangehensweise zum Ziel führen, sie wurden im Rahmen dieser Arbeit aber nicht weiter verfolgt.

6.2.1 Ausweitung der Sensoren

In dieser Arbeit wurde ein sehr einfaches Sensorenmodell verwendet, das zwar höhere Sichtweiten liefert, dafür aber sehr ungenaue Informationen liefert. Zu einem wesentlich besseren Ergebnis könnte die Verwendung von einer größeren Anzahl von Sensoren bzw. rationale Eingabewerten führen. Das würde beispielsweise die Möglichkeit erlauben, 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]. Mehrwertige Sensoren könnte man einfach durch Hinzunahme von weiteren Binärsensoren bewerkstelligen.

Zusammen mit der Erweiterung der Sensoren könnte auch eine bessere *reward* Funktion für den *base reward* entwickelt werden (siehe Kapitel 3.3.4), wodurch das globale Problem womöglich besser gelernt werden könnte.

6.2.2 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. Womöglich ist hierzu eine Untersuchung einzelner Agenten in einem einfacheren Szenario zielführend.

6.2.3 Erhöhung des Bedarfs an Kollaboration

Die in dieser Arbeit verwendeten Szenarien konnten nur unzureichend die Kollaboration zwischen den Agenten in den Vordergrund stellen. Ein einfaches Verfolgen, also eine lokale Strategie, führte eher zum Erfolg. Dies zeigt insbesondere der Vergleich der einfachen mit der intelligenten Heuristik bei unterschiedlichen Geschwindigkeiten des Zielobjekts auf Szenarien mit relativ wenigen Hindernissen (siehe Kapitel 5.2.2), obwohl sich das Zielobjekt in diesem Fall intelligent verhalten hat.
Eine weitere Idee in dieser Richtung wäre die Änderung der Problemstellung an sich, dass also z.B. das Zielobjekt erst dann als überwacht gilt, wenn es von mehreren Seiten beobachtet wird.

6.2.4 Rotation des *condition* Vektors

Ursprünglich wurde das Szenario auf Basis von Rotation konzipiert. Die Annahme war, dass ein Agent, der für einen Satz an Sensordaten eine optimales *classifier set* gefunden hat, dieses *classifier set* auch für Sensordaten eines um 90, 180 und 270 Grad gedrehten Szenarios (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 fallengelassen. 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 bedurfte aber weiterer Untersuchung und ging am eigentlichen Thema dieser Arbeit vorbei.

6.2.5 Abnehmende Wahrscheinlichkeit der *explore* Phase

In Kapitel 3.4.5 wurden mehrere Arten des Wechsels zwischen der *explore* und *exploit* Phase vorgestellt. 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 angepasst wird.
Wie dies bei Überwachungsszenarios ausgenutzt werden könnte ist noch unklar, die Ergebnisse bezüglich des Wechsels bei der Änderung des *reward* Werts in Kapitel 5.3 scheinen daraufhin zu deuten, dass dadurch ein Ausgleich geschaffen werden kann, zwischen selten erlebten und häufig erlebten Situationen. Womöglich wäre hier auch eine Anpassung des Wechsels zwischen der *explore* und *exploit* Phase anhand der Daten in der *classifier set* Liste sinnvoll. Beispielsweise könnte man in Situationen in der das *match set* nur *classifier* mit insgesamt niedrigem *experience* Wert aufweisen, eher in die *explore* Phase wechseln. Weitere Untersuchungen wären hier angebracht.

6.2.6 Gesonderte Behandlung von neutralen Ereignissen

Beträgt bei einem neutralen Ereignis (siehe Kapitel 4.3.2) der *base reward* Wert 0 so wurde ausprobiert, dann die vom Stack genommenen Werte einfach zu verwerfen. Idee war, dass ein Agent, der es längere Zeit nicht schafft, das Zielobjekt in Sichtweite zu bekommen, nicht unbedingt falsch, was das globale Ziel betrifft, gehandelt haben muss, da er trotzdem ein Gebiet überwacht, das vom Zielobjekt passiert hätte können. Die Annahme war, dass (nach der erwarteten Verteilung der zukünftigen Positionen des Zielobjekts)

selbst bei einer optimalen Verteilung der Agenten, einige Agenten das Zielobjekt nie in Sicht bekommen und deshalb durch ein neutrales Ereignis nicht bestraft werden sollen. In den verwendeten Szenarien hat dies zu keinem Erfolg geführt, im Gegenteil insbesondere im schwierigen Szenario hat dies dazu geführt, dass ein deutlich schlechteres Ergebnis erreicht wurde. Dies lässt sich dadurch erklären, dass mit gesonderter Behandlung Agenten, die etwas falsches gelernt haben, dies (mangels Kontakt zum Zielobjekt) sehr schwierig wieder verlernen. Dies sieht man an der Darstellung des gleitenden Durchschnitts der Qualität in Abbildung 6.1. Das erste Problem wird von allen drei Varianten problemlos gemeistert, dann gibt es einen Kontakt zum Zielobjekt, der aber (bei einem hohen Wert von β von 0,1) so stark ist, dass der jeweilige Agent im nächsten Problem kein Ziel mehr findet. Mit geringerer Lernrate β lässt sich das zwar lösen, sorgt aber für eine geringere Konvergenzgeschwindigkeit.
Fazit ist, dass eine Sonderbehandlung für den Fall mit neutralem Ereignis ohne positiven *base reward* zwar wegen oben genannter Gründe wichtig scheint, das „Vergessen“ aber auch wertvoll sein kann.

6.2.7 Anpassung des *maxStackSize* Werts

Bei den Ereignissen 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 für den Algorithmus sinnvoll, dass eine Methode entwickelt wird, mit der sich der *maxStackSize* Wert während des Laufs an das jeweilige Szenario anpassen kann.

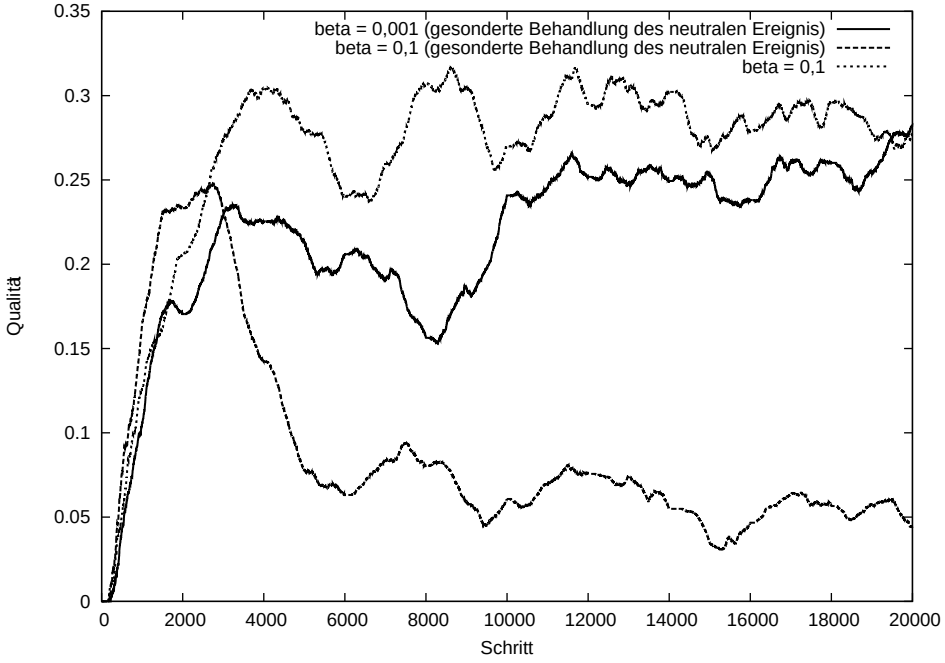

Abbildung 6.1: Auswirkung des Parameters *learning rate* β auf den gleitenden Durchschnitt der Qualität im schwierigen Szenario, Bewegung des Zielobjekts ohne Richtungsänderung, Geschwindigkeit 1, 8 Agenten mit SXCS Algorithmus, 2000 Schritte

6.2.8 Lernendes Zielobjekt

Sicher interessant ist auch der umgekehrte Ansatz, der in Kapitel 4.3.6 erwähnt wurde, dass das Zielobjekt das Objekt ist, das lernt und den Agenten ausweichen muss. Bei dieser Problemstellung fällt zwar der kollaborative Aspekt weg, der Vorteil ist jedoch, dass die selbe Simulation verwendet werden kann um dieses Problem zu untersuchen.

6.2.9 Weitere Berechnungsmethoden für den Kommunkationsfaktor

Im Bereich der Kommunikation wurde neben in dieser Arbeit besprochenen „egoistischen Relation“ (siehe Kapitel 4.4.3) auch weitere Verfahren ausprobiert, mit welchen versucht wurde, gleichartige Gruppen zu finden. Hier wurden ganze *classifier set* Listen unterschiedlicher Agenten miteinander auf Ähnlichkeit geprüft um daraus einen Faktor zu berechnen, der (wie bei der „egoistischen Relation“) Einfluss auf die Weitergabe des *reward* Werts haben sollte. Der dadurch deutlich erhöhte Kommunikations- und Berechnungsaufwand lag jedoch in keinem Verhältnis zu eventuell beobachteten Qualitätsverbesserungen, im Gegenteil wurden eher Qualitätsverschlechterungen beobachtet. Die Ergebnisse mit dem Test der „egoistischen Relation“ zeigen jedoch, dass hier zumindest etwas Potential stecken könnte und für bestimmte Szenarien die zwei Grundideen, dass sich die Agenten zum einen an die Größe des Szenarios anpassen und zum anderen der *reward* Wert möglichst nur an sich ähnlich verhaltende Agenten weitergegeben wird, nicht ganz falsch sein können. Genauere, insbesondere theoretische, Untersuchungen sind hier nötig.

6.2.10 Verworfene Szenarien

Was die Szenarien selbst betrifft, wurden ebenfalls mehrere verworfen, da bei ihnen keine zusätzlichen Beobachtungen gemacht bzw. nur unbedeutende Teilaspekte betrachtet werden konnten. Unter anderem sind dies ein Labyrinth, bei denen die Agenten wahrscheinlich an den mangelnden Fähigkeiten der Sensoren scheiterten, ein vereinfachtes „schwieriges Szenario“ mit einem „Raum“ mit einer Öffnung in der Mitte, welches sich als zu einfach zu lösen herausstellte und ein Szenario mit einem Kreuz bestehend aus Hindernissen in der Mitte, welches keine bedeutend anderen Ergebnisse lieferte als das Szenario mit zufällig verteilten Hindernissen.

6.3 Vorgehen und verwendete Hilfsmittel und Software

Zu Beginn stellte sich die Frage, welche Software zu benutzen ist, da es sich um ein recht komplexe Problemstellung handelt. 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 enthielt.
Auf Basis des dadurch gewonnenen Wissens war es dann leichter, die XCS Implementierung zu verstehen und nachvollziehen zu können. Insbesondere die Optimierungen und der etwas unsaubere Programmierstil in der Standardimplementierung bereiteten Probleme.

Anhand des Studiums der Literatur war klar, dass in der Richtung der Überwachungsszenarien es wenig Arbeiten, die sich damit beschäftigten, wie die XCS Implementierung umzusetzen sei. Ein Rückgriff auf bestehende Bibliotheken war deshalb nicht möglich,

ursprünglich geplante Untersuchungen komplexerer Systeme wie zentrale Steuerung, Austausch von Regeln etc. wurden gestrichen und es wurde sich auf den einfachen Fall, lokale Information ohne zentrale Steuerung mit höchstens minimaler Kommunikation beschränkt. Dies machte die Verwendung komplexerer Simulationssysteme unnötig, die Einarbeitungszeit in Multiagenten Frameworks wie z.B. Repast [Rep] erschien zu hoch, wie auch die Risiken, was Geschwindigkeit, Kompatibilität und Speicherverbrauch betraf, unbekannt waren, weshalb ein eigenes Simulationsprogramm entwickelt wurde.

Das Simulationsprogramm samt zugehöriger Oberfläche [Lod09] zur Erstellung von neuen Test-Jobs wurde in Java mit Hilfe von NetBeans IDE 6.5 [NB6] selbst entwickelt und gestaltet.

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

6.4 Beschreibung des Konfigurationsprogramms

In Abbildung 6.2 ist ein Screenshot des gesamten Konfigurationsprogramms abgebildet. Auf der rechten Seite befinden sich die Ergebnisse aller bisherigen Läufe in einer Datenbank, auf der linken Seite (siehe Abbildung 6.3) 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. Ergebnisse werden für jedes Experiment in ein Verzeichnis geschrieben, aus der das Programm wiederum automatisch einen Eintrag in der Datenbank generiert, welche mittels der Betätigung des *Update* Knopfs aktualisiert wird.

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

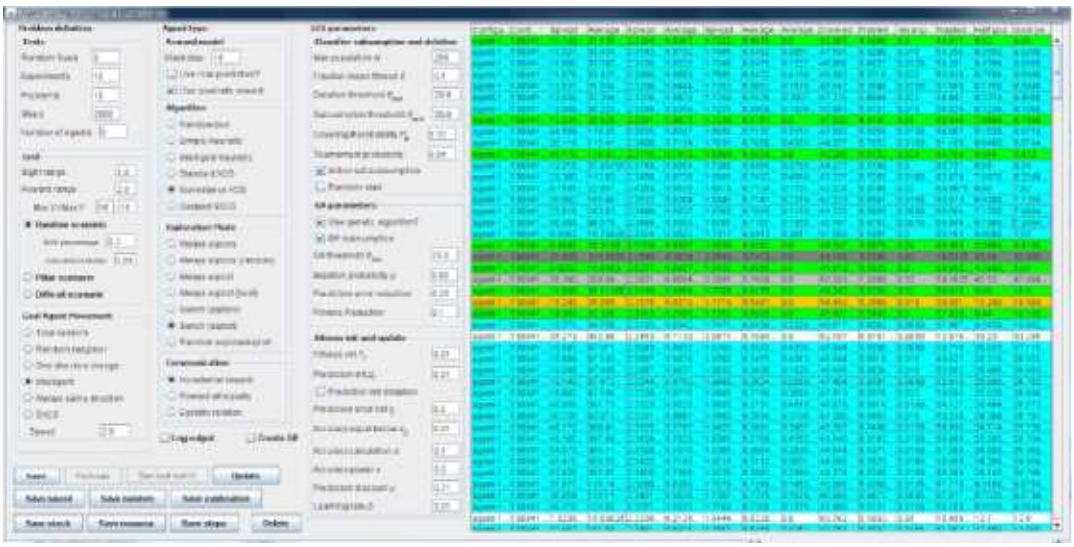

Abbildung 6.2: Screenshot des Konfigurationsprogramms (Gesamtübersicht)

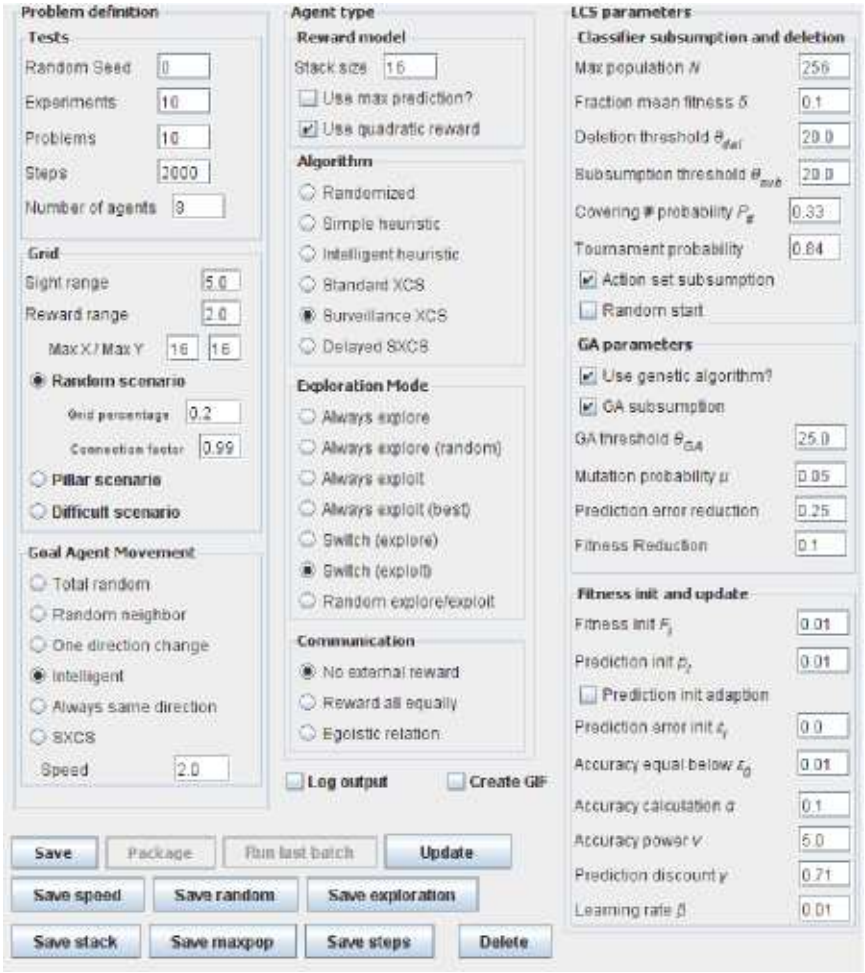

Abbildung 6.3: Screenshot des Konfigurationsprogramms (Konfigurationsbereich)

Anhang A

Implementation

A.1 Implementierung eines Problemablaufs

In der Schleife der Funktion zur Berechnung eines Experiments (Programm A.1) wird die Funktion zur Berechnung des Problems (*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 Problemen aus
3   * @param experiment_nr Nummer des auszuführenden Experiments
4   */
5    public void doOneMultiStepExperiment(int experiment_nr) {
6      int currentTimestep = 0;
7
8    /**
9     * number of problems for the same 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 Problem 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(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 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 Rewards
41      */
42     BaseAgent.grid.updateSight();
43     rewardAgents(steps_next_problem);
44     return steps_next_problem;
45   }
```

Programm A.2: Zentrale Schleife für einzelne Probleme

```
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 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 eines Problems

```
1  /**
2   * Verteilt den Reward an alle Agenten
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 eines Problems

```
1  /**
2   * Berechnet die Aktionen und führt sie in zufälliger Reihenfolge aus
3   * @param gaTimestep der aktuelle Zeitschritt
4   */
5    private void moveAgents(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    *   (Zielobjekt kann 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 eines Problems

A.2 Typen von Agentenbewegungen

```
1  /**
2   * Berechne nächste Aktion (zufälliger Algorithmus)
3   */
4    private void calculateNextMove() {
5    /**
6     * Wähle zufällige Richtung als nächste Aktion
7     */
8      calculatedAction = Misc.nextInt(Action.MAX_DIRECTIONS);
9    }
```

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

```
1  /**
2   * Berechne nächste Aktion (einfache Heuristik)
3   */
4    private void calculateNextMove() {
5    /**
6     * Holt sich die Informationen der Gruppe der Sensoren, die auf
7     * das Zielobjekt ausgerichtet sind
8     */
9      boolean[] goal_sensor = lastState.getSensorGoal();
10     calculatedAction = -1;
11     for(int i = 0; i < Action.MAX_DIRECTIONS; i++) {
12     /**
13      * Zielagent in Sicht in dieser Richtung?
14      */
15       if(goal_sensor[2*i]) {
16         calculatedAction = i;
17         break;
18       }
19     }
20
21   /**
22    * Sonst wähle zufällige Richtung als nächste Aktion
23    */
24     if(calculatedAction == -1) {
25       calculatedAction = Misc.nextInt(Action.MAX_DIRECTIONS);
26     }
27
28   }
```

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     * Holt sich die 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 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, während die hier verwendete Implementierung jeweils mit bis über 100 *action set* Listen programmiert wurde, 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 der der *classifier* Mitglied ist.

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

Positiver Nebeneffekt durch die verbesserte Struktur ist, dass man dadurch leicht auf die Menge der *action set* Listen zugreifen kann, denen ein *classifier* angehört, hierfür wurde aber im Rahmen dieser Arbeit keine Verwendung gefunden.

Ein weiteres Problem der Standardimplementierung ist, dass der *fitness* Wert eines

classifiers als Optimierung bereits den *numerosity* Wert als Faktor enthält, während bei der Aktualisierung des *numerosity* Werts der *fitness* Wert nicht aktualisiert wurde. Das hat zur Folge, dass theoretisch *fitness* Werte von *classifiers* 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.9 dargestellt.

Möglicherweise kann man diesen Fehler durch Veränderung der Parameter oder längere Laufzeiten kompensieren, logisch betrachtet macht es aber 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) eine Qualität von 39,15% im Vergleich zur originalen Implementierung von 39,95% bei 500 Schritten bzw. 35,42% zu 35,01% bei 2000 Schritten. Der Fehler scheint sich also eher langfristig auszuwirken, wenn auch der Unterschied so klein ist, dass man ihn vernachlässigen kann. Problematisch wird es, wenn Modifikationen von XCS darauf aufbauen, dass der *fitness* Wert für jeden *micro classifier* immer kleiner gleich *1.0* ist.

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

```
1  /**
2   * Erhöht oder erniedrigt den numerosity Wert des classifier
3   * @param num Der zur numerosity zu addierende Wert (kann negativ sein).
4   */
5    public void addNumerosity(int num) {
6      int old_num = numerosity;
7
8      numerosity += num;
9
10   /**
11    * Korrektur der fitness
12    */
13     if(old_num > 0) {
14       fitness = fitness * (double)numerosity / (double)old_num;
15     } else {
16       fitness = Configuration.
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.9: Korrigierte Version der *addNumerosity()* Funktion

A.4 Implementierung XCS Multistepverfahrens

```
1  /**
2   * Diese Funktion wird in jedem Schritt aufgerufen um den aktuellen
3   * reward zu bestimmen, den besten Wert der ermittelten match set Liste
4   * weiterzugeben und, bei aktuell positivem reward, die aktuelle
5   * action set Liste zu belohnen.
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 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.10: Erstes Kernstück des Standard XCS Multistepverfahrens (*calculateReward()*, Bestimmung und Verarbeitung des *reward* Werts anhand der Sensordaten), angepasst an ein dynamisches Überwachungsszenario, bei positivem *reward* Wert wird nicht abgebrochen

```
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(boolean reward,
12                 double best_value, boolean is_event) {
13     double corrected_reward = reward ? 1.0 : 0.0;
14
15   /**
16    * Falls der reward Wert von einem Ereignis rührt, aktualisiere die
17    * aktuelle action set Liste und lösche das vorherige
18    */
19     if(is_event) {
20       if(lastActionSet != null) {
21         lastActionSet.updateReward(corrected_reward, best_value, factor);
22         prevActionSet = null;
23       }
24     }
25
26   /**
27    * Kein Ereignis, also nur die letzte action set Liste aktualisieren
28    */
29     else
30     {
31       if(prevActionSet != null) {
32         prevActionSet.updateReward(corrected_reward, best_value, factor);
33       }
34     }
35   }
```

Programm A.11: 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(long gaTimestep) {
10
11  /**
12   * Überdecke das classifierSet mit zum Status passenden Classifiern
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 Aktion und bestimme zugehörige action set Liste
31   */
32     calculatedAction = lastMatchSet.chooseAbsoluteDirection(lastExplore);
33     lastActionSet = new ActionClassifierSet(lastState, lastMatchSet,
34                                                      calculatedAction);
35   }
```

Programm A.12: 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.5 Implementierung des SXCS Verfahrens

```
1  /**
2   * Diese Funktion wird in jedem Schritt aufgerufen um den aktuellen
3   * reward Wert zu bestimmen und positive, negative und neutrale
4   * Ereignisse den besten Wert der ermittelten match set Liste
5   * weiterzugeben und, bei aktuell positivem reward Wert, die
6   * aktuelle action set Liste zu belohnen.
7   *
8   * @param gaTimestep Der aktuelle Zeitschritt
9   */
10
11   public void calculateReward(final long gaTimestep) {
12   /**
13    * checkRewardPoints() liefert "wahr" wenn sich das Zielobjekt in
14    * Sichtweite befindet
15    */
16     boolean reward = checkRewardPoints();
17
18     if (reward != lastReward) {
19       int start_index = historicActionSet.size() - 1;
20       collectReward(start_index, actionSetSize, reward, 1.0, true);
21       actionSetSize = 0;
22     }
23     else
24
25     if(actionSetSize >= Configuration.getMaxStackSize())
26     {
27       int start_index = Configuration.getMaxStackSize() / 2;
28       int length = actionSetSize - start_index;
29       collectReward(start_index, length, reward, 1.0, false);
30       actionSetSize = start_index;
31     }
32
33     lastReward = reward;
34   }
```

Programm A.13: Erstes Kernstück des SXCS-Algorithmus (*calculateReward()*, Bestimmung des *reward* Werts anhand der Sensordaten)

```
1  /**
2   * Diese Funktion verarbeitet den übergebenen reward und gibt ihn an
3   * die zugehörigen action set Listen weiter.
4   *
5   * @param reward Wahr wenn der 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(int start_index, int action_set_size,
12                  boolean reward, double best_value, boolean is_event) {
13      double corrected_reward = reward ? 1.0 : 0.0;
14    /**
15     * Keine Weitergabe des reward Werts wie bei XCS
16     */
17      double max_prediction = 0.0;
18
19    /**
20     * Aktualisiere eine ganze Anzahl von action set Listen
21     */
22      for(int i = 0; i < action_set_size; i++) {
23
24    /**
25     * Benutze aufsteigenden bzw. absteigenden reward Wert bei einem
26     * positiven bzw. negativen Ereignis
27     */
28        if(is_event) {
29          corrected_reward = reward ?
30            calculateReward(i, action_set_size) :
31            calculateReward(action_set_size - i,
32  action_set_size);
33        }
34    /**
35     * Aktualisiere die action set Liste mit dem bestimmten reward Wert
36     * und gebe bei allen anderen action set Listen den reward Wert
37     * weiter wie beim multi step Verfahren
38     */
39        ActionClassifierSet action_classifier_set =
40          historicActionSet.get(start_index - i);
41        action_classifier_set.updateReward(
42          corrected_reward, max_prediction, factor);
43      }
44    }
```

Programm A.14: 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(long gaTimestep) {
12
13  /**
14   * Überdecke das classifierSet mit zum Status passenden Classifiern
15   * 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 Aktion und bestimme 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.15: Drittes Kernstück des SXCS-Algorithmus (*calculateNextMove()* - Auswahl der nächsten Aktion und Ermittlung und Speicherung der zugehörigen *action set* Liste)

A.6 Implementation des DSXCS Algorithmus

```
1  /**
2   * Diese Funktion verarbeitet den übergebenen reward Wert und gibt ihn an
3   * die zugehörigen action set Listen weiter. Wesentlicher Unterschied zum
4   * SXCS Algorithmus ist, dass der maxPrediction Wert erst bei der
5   * endgültigen Verarbeitung des historicActionSet Listen ermittelt wird.
6   *
7   * @param reward Wahr wenn das Zielobjekt in Sicht war.
8   * @param best_value Bester Wert des vorangegangenen action set Listen
9   * @param is_event Wahr wenn diese Funktion wegen eines Ereignisses,
10  *           d.h. einem positiven reward Wert, aufgerufen wurde
11  */
12
13   public void collectReward(int start_index, int action_set_size,
14                 boolean reward, double best_value, boolean is_event) {
15     double corrected_reward = reward ? 1.0 : 0.0;
16
17   /**
18    * Aktualisiere eine ganze Anzahl von Einträgen im historicActionSet
19    */
20      for(int i = 0; i < action_set_size; i++) {
21
22   /**
23    * Benutze aufsteigenden bzw. absteigenden reward Wert bei
24    * einem positiven bzw. negativen Ereignis
25    */
26        if(is_event) {
27          corrected_reward = reward ?
28            calculateReward(i, action_set_size) :
29            calculateReward(action_set_size - i, action_set_size);
30        } else {
31          if(corrected_reward == 1.0 && factor == 1.0) {
32            historicActionSet.get(start_index - i).rewardPrematurely();
33          }
34        }
35
36   /**
37    * Füge den ermittelten reward Wert zur historicActionSet Liste
38    */
39        historicActionSet.get(start_index - i).
40          addReward(corrected_reward, factor);
41
42     }
43   }
```

Programm A.16: 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
9    public void calculateNextMove(long gaTimestep) {
10
11   // ...
12
13   /**
14    * historicActionSet voll? Dann verarbeite den dortigen reward Wert
15    */
16      if (historicActionSet.size() > Configuration.getMaxStackSize()) {
17        historicActionSet.pop().processReward();
18     }
19   }
```

Programm A.17: Auszug aus dem zweiten Kernstück des verzögerten SXCS Algorithmus DSXCS (*calculateNextMove()*)

```
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     * Finde das größte reward / factor Paar
9     */
10     for(RewardHelper r : reward) {
11     /**
12      * Dieser Eintrag wurde schon in collectReward() verwertet
13      */
14       if(r.reward == 1.0 && r.factor == 1.0) {
15         continue;
16       }
17     /**
18      * Aktualisiere den Eintrag mit den entsprechenden Werten
19      */
20       actionClassifierSet.updateReward(r.reward, 0.0, r.factor);
21     }
22   }
```

Programm A.18: Drittes Kernstück des verzögerten SXCS Algorithmus DSXCS (Verarbeitung des jeweiligen reward Werts, *processReward()*)

```
1  /**
2   * Zentrale Routine der historicActionSet Liste zur Verarbeitung aller
3   * 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      * Dieser Eintrag wurde schon in collectReward() verwertet
16      */
17       if(r.reward == 1.0 && r.factor == 1.0) {
18         return;
19       }
20
21       if(r.reward * r.factor > max_value) {
22         max_value = r.reward * r.factor;
23         max_reward = r.reward;
24       }
25     }
26     /**
27      * Aktualisiere den Eintrag mit dem ermittelten Werten
28      */
29     actionClassifierSet.updateReward(max_reward, 0.0, 1.0);
30   }
```

Programm A.19: Verbesserte Variante des dritten Kernstück des verzögerten SXCS Algorithmus DSXCS (Verarbeitung des reward Werts, *processReward()*)

A.7 Implementation 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.20: “Egoistische Relation“, Algorithmus zur Bestimmung des Kommunikationsfaktors basierend auf dem erwarteten Verhalten des Agenten gegenüber anderen Agenten

Literaturverzeichnis

[Bar02] A. Barry. The stability of long action chains in xcs, 2002.

[BD03] Alwyn Barry and Claverton Down. Limits in long path learning with xcs. In *Proc. GECCO 2003, Genetic and Evolutionary Computation Conference*, pages 1832–1843. Springer-Verlag, 2003.

[BDE+99] W. Banzhaf, J. Daida, A. E. Eiben, M. H. Garzon, V. Honavar, M. Jakiela, and R. E. Smith. Extending the representation of classifier conditions, part i: From binary to messy coding. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 337–344. Morgan Kaufmann, 1999.

[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`.

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

[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.

[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.

[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/`.

[TB06] J.-M. Nigro T. Benouhiba. An evidential cooperative multi-agent system. *Expert Systems with Applications*, 30(2):255–264, 2006.

[THN+98] K. Takadama, K. Hajiri, T. Nomura, M. Okada, S. Nakasuka, and K. Shimohara. Learning model for adaptive behaviors as an organized group of swarm robots. *Artificial Life and Robotics*, 2(3):123–128, 1998.

[Wei00] G. Weiss. *Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence*. The MIT Press, July 2000. "Collaboration".

[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.

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, 30. März 2009,

Clemens Lode