

# **DIPLOMARBEIT**

## **XCS in dynamischen Multiagenten-Überwachungsszenarien ohne 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

<b>1</b>	<b>Einleitung und Motivation</b>	<b>1</b>
1.1	Stand der Wissenschaft . . . . .	3
1.1.1	Beschreibung und Beispiel für das <i>single step</i> Verfahren . . . . .	3
1.1.2	Beschreibung und Beispiel für das <i>multi step</i> Verfahren . . . . .	4
1.1.3	Problemdefinition . . . . .	6
1.2	Aufbau der Arbeit . . . . .	9
<b>2</b>	<b>Beschreibung des Szenarios</b>	<b>11</b>
2.1	Dynamische, kollaborative Szenarien . . . . .	13
2.2	Konfigurationen des Torus . . . . .	14
2.2.1	Leeres Szenario ohne Hindernisse . . . . .	15
2.2.2	Szenario mit zufällig verteilten Hindernissen . . . . .	15
2.2.3	Säulenszenario . . . . .	17
2.2.4	Schwieriges Szenario . . . . .	18
2.3	Eigenschaften der Objekte . . . . .	19
2.3.1	Sichtbarkeit von Objekten . . . . .	19
2.3.2	Aufbau eines Sensordatenpaares . . . . .	20
2.3.3	Aufbau eines Sensordatensatzes . . . . .	21
2.3.4	Eigenschaften der Agenten und des Zielobjekts . . . . .	22

2.4	Grundsätzliche Algorithmen der Agenten . . . . .	23
2.4.1	Algorithmus mit zufälliger Bewegung . . . . .	24
2.4.2	Algorithmus mit einfacher Heuristik . . . . .	24
2.4.3	Algorithmus mit intelligenter Heuristik . . . . .	25
2.5	Typen von Zielobjekten . . . . .	26
2.5.1	Typ „Zufälliger Sprung“ . . . . .	27
2.5.2	Typ „Zufällige Bewegung“ . . . . .	27
2.5.3	Typ „Einfache Richtungsänderung“ . . . . .	28
2.5.4	Typ „Intelligentes Verhalten“ . . . . .	28
2.5.5	Typ „Beibehaltung der Richtung“ . . . . .	29
2.6	Simulation und erfasste Statistiken . . . . .	31
2.6.1	Definition einer Probleminstanz . . . . .	31
2.6.2	Abdeckung . . . . .	33
2.6.3	Qualität eines Algorithmus . . . . .	33
2.6.4	Ablauf der Simulation . . . . .	35
2.6.5	Reihenfolge bei unterschiedlichen Bewegungsgeschwindigkeiten . . .	36
2.6.6	Messung der Qualität . . . . .	37
2.6.7	Reihenfolge der Ermittlung des <i>base reward</i> . . . . .	38
2.6.8	Zusammenfassung des Simulationsablaufs . . . . .	39
<b>3</b>	<b>XCS</b>	<b>41</b>
3.1	Classifier . . . . .	42
3.1.1	Der <i>condition</i> Vektor . . . . .	42
3.1.2	Der <i>action</i> Wert . . . . .	43
3.1.3	Der <i>fitness</i> Wert . . . . .	43
3.1.4	Der <i>reward prediction</i> Wert . . . . .	43
3.1.5	Der <i>reward prediction error</i> Wert . . . . .	43

3.1.6	Der <i>experience</i> Wert . . . . .	44
3.1.7	Der <i>numerosity</i> Wert . . . . .	44
3.2	Vergleich des <i>condition</i> Vektors mit Sensordaten . . . . .	44
3.2.1	Erkennung von Sensordatenpaaren . . . . .	45
3.2.2	Subsummation von <i>classifier</i> . . . . .	46
3.3	Ablauf eines XCS . . . . .	47
3.3.1	Abdeckung aller Aktionen durch <i>covering</i> . . . . .	47
3.3.2	Die <i>match set</i> Liste . . . . .	47
3.3.3	Die <i>action set</i> Liste . . . . .	48
3.3.4	Genetische Operatoren . . . . .	48
3.4	Bewertung der Aktionen ( <i>base reward</i> ) . . . . .	49
3.4.1	Bewertung beim <i>single step</i> Verfahren . . . . .	49
3.4.2	Bewertung beim <i>multi step</i> Verfahren . . . . .	50
3.4.3	Bewertung bei einem Überwachungsszenario . . . . .	50
3.5	Auswahlart der <i>classifier</i> . . . . .	53
3.5.1	Auswahlart <i>random selection</i> . . . . .	54
3.5.2	Auswahlart <i>best selection</i> . . . . .	54
3.5.3	Auswahlart <i>roulette wheel selection</i> . . . . .	55
3.5.4	Auswahlart <i>tournament selection</i> . . . . .	55
3.5.5	Wechsel zwischen den <i>explore</i> und <i>exploit</i> Phasen . . . . .	57
3.6	Beschreibung und Analyse der XCS Parameter . . . . .	60
3.6.1	Parameter <i>max population N</i> . . . . .	60
3.6.2	Zufällige Initialisierung der <i>classifier set</i> Liste . . . . .	63
3.6.3	Parameter <i>reward prediction discount <math>\gamma</math></i> . . . . .	65
3.6.4	Parameter Lernrate $\beta$ . . . . .	67
3.6.5	Parameter <i>accuracy equality <math>\epsilon_0</math></i> . . . . .	68

3.6.6	Parameter <i>tournament factor</i> $p$ . . . . .	70
3.6.7	Übersicht über alle Parameterwerte . . . . .	73
<b>4</b>	<b>XCS Varianten</b>	<b>75</b>
4.1	Allgemeine Anpassungen . . . . .	76
4.2	XCS <i>multi step</i> Verfahren . . . . .	77
4.3	XCS Variante für Überwachungsszenarien (SXCS) . . . . .	78
4.3.1	Umsetzung von SXCS . . . . .	78
4.3.2	Ereignisse . . . . .	80
4.3.3	Größe des Stacks ( <i>maxStackSize</i> ) . . . . .	82
4.3.4	Zusammenfassung der Ereignisse . . . . .	83
4.3.5	Implementierung von SXCS . . . . .	83
<b>5</b>	<b>Analysen und Experimente</b>	<b>85</b>
5.1	Erste Analyse der Agenten ohne XCS . . . . .	86
5.1.1	Zielobjekt mit zufälligem Sprung (leeres Szenario) . . . . .	86
5.1.2	Zielobjekt mit zufälligem Sprung (Säulenszenario) . . . . .	87
5.1.3	Zielobjekt mit zufälligem Sprung (Zufällig verteilte Hindernisse) . . . . .	88
5.1.4	Zielobjekt mit zufälliger Bewegung bzw. einfacher Richtungsänderung . . . . .	91
5.2	Auswirkung der Geschwindigkeit des Zielobjekts . . . . .	92
5.2.1	Zielobjekt mit einfacher Richtungsänderung . . . . .	93
5.2.2	Zielobjekt mit intelligenter Bewegung . . . . .	94
5.3	Szenario mit zufällig verteilten Hindernissen . . . . .	96
5.4	Schwieriges Szenario . . . . .	96
5.5	Zusammenfassung der Tests mit Heuristiken . . . . .	98
5.6	Test der verschiedenen XCS Auswahlarten . . . . .	100
5.7	Unterschiedliche Geschwindigkeiten des Zielobjekts . . . . .	104

5.8	Einschränkung der Ergebnisse für Szenario mit zufällig verteilte Hindernisse	105
5.9	Vergleich XCS und SXCS . . . . .	106
5.9.1	XCS im schwierigen Szenario . . . . .	106
5.10	DSXCS . . . . .	110
5.11	Beobachtete Strategien der Agenten TODOS . . . . .	110
5.12	Zusammenfassung der bisherigen Erkenntnisse . . . . .	111
<b>6</b>	<b>Zusammenfassung, Ergebnis und Ausblick</b>	<b>113</b>
6.1	Ergebnis TODO . . . . .	114
6.2	Ausblick und verworfene Ansätze . . . . .	115
6.2.1	Ausweitung der Sensoren . . . . .	115
6.2.2	Verwendung einer mehrwertigen <i>reward</i> Funktion . . . . .	116
6.2.3	Untersuchung der Theorie . . . . .	116
6.2.4	Untersuchung der <i>classifier</i> . . . . .	116
6.2.5	Erhöhung des Bedarfs an Kollaboration . . . . .	117
6.2.6	Rotation des <i>condition</i> Vektors . . . . .	117
6.2.7	Abnehmende Wahrscheinlichkeit der <i>explore</i> Phase . . . . .	118
6.2.8	Gesonderte Behandlung von neutralen Ereignissen . . . . .	118
6.2.9	Anpassung des <i>maxStackSize</i> Werts . . . . .	119
6.2.10	Lernendes Zielobjekt . . . . .	121
6.2.11	Verworfene Szenarien . . . . .	121
6.3	Vorgehen und verwendete Hilfsmittel und Software . . . . .	122
6.4	Beschreibung des Konfigurationsprogramms . . . . .	124
<b>A</b>	<b>Implementation</b>	<b>127</b>
A.1	Implementierung eines Problemablaufs . . . . .	127
A.2	Typen von Agentenbewegungen . . . . .	132

A.3 Korrigierte <i>addNumerosity()</i> Funktion . . . . .	134
A.4 Implementierung XCS Multistepverfahrens . . . . .	137
A.5 Implementierung des SXCS Verfahrens . . . . .	140



# Abbildungsverzeichnis

1.1	Schematische Darstellung des 6-Multiplexer Problems . . . . .	4
1.2	Einführendes Beispiel zum XCS <i>multi step</i> Verfahren . . . . .	5
1.3	Vereinfachte Darstellung eines <i>classifier set</i> für das Beispiel zum XCS <i>multi step</i> Verfahren . . . . .	6
2.1	„Leeres Szenario“ ohne Hindernisse . . . . .	15
2.2	Szenario mit zufällig verteilten Hindernissen mit $\lambda_h = 0,1$ . . . . .	16
2.3	Szenario mit zufällig verteilten Hindernissen mit $\lambda_h = 0,2$ . . . . .	17
2.4	Startzustand des Säulen Szenarios . . . . .	17
2.5	Schwieriges Szenario . . . . .	18
2.6	Sicht- und Überwachungsreichweite eines Agenten . . . . .	20
2.7	Darstellung des Sensordatensatzes . . . . .	21
2.8	Beispiel für einen Sensordatensatz . . . . .	22
2.9	Sich zufällig bewegender Agent . . . . .	24
2.10	Agent mit einfacher Heuristik . . . . .	25
2.11	Agent mit intelligenter Heuristik . . . . .	26
2.12	Zielobjekt mit maximal einer Richtungsänderung . . . . .	28
2.13	Ein sich intelligent verhaltendes Zielobjekt weicht Agenten aus. . . . .	29
2.14	Bewegungsform „Beibehaltung der Richtung“: Zielobjekt das sich, wenn möglich, immer nach Norden bewegt . . . . .	30

2.15	Varianz der Testergebnisse bei unterschiedlicher Anzahl von Experimenten	32
3.1	Einteilung des <i>condition</i> Vektors . . . . .	42
3.2	Auswirkung der maximalen Populationsgröße auf die Anzahl der <i>classifier</i> die durch <i>covering</i> neu erstellt werden (Säulenszenario) . . . . .	61
3.3	Auswirkung der maximalen Populationsgröße auf die Anzahl der <i>classifier</i> die durch <i>covering</i> neu erstellt werden (leeres Szenario) . . . . .	62
3.4	Auswirkung der Torusgröße auf die Laufzeit (leeres Szenario) . . . . .	63
3.5	Auswirkung des Parameters <i>max population N</i> auf Laufzeit (leeres Szenario)	64
3.6	Verhältnis Laufzeit zu <i>max population N</i> (leeres Szenario) . . . . .	64
3.7	Auswirkung der maximalen Populationsgröße auf die Anzahl der <i>classifier</i> , die durch <i>covering</i> neu erstellt werden (Säulenszenario, ohne Initialisierung der <i>classifier set</i> Liste) . . . . .	66
3.8	Auswirkung verschiedener <i>reward prediction discount <math>\gamma</math></i> Werte auf die Qua- lität . . . . .	67
3.9	Auswirkung des Parameters <i>learning rate <math>\beta</math></i> auf Qualität (Säulenszenario) .	68
3.10	Auswirkung des Parameters <i>learning rate <math>\beta</math></i> auf Qualität (Schwieriges Sze- nario) . . . . .	69
3.11	Zeitlicher Verlauf des gleitenden Durchschnitts der Qualität (schwieriges Szenario, SXCS und XCS mit Lernrate 0,1) . . . . .	69
3.12	Vergleich verschiedener Werte <i>p</i> für Auswahlart <i>tournament selection</i> (Ziel- objekt mit einfacher Richtungsänderung, Geschwindigkeit 1) . . . . .	70
3.13	Vergleich verschiedener Werte <i>p</i> für Auswahlart <i>tournament selection</i> (Ziel- objekt mit einfacher Richtungsänderung, Geschwindigkeit 1) . . . . .	71
3.14	Vergleich verschiedener Werte <i>p</i> für Auswahlart <i>tournament selection</i> (in- telligentes Zielobjekt, Geschwindigkeit 1) . . . . .	72

3.15 Vergleich verschiedener Werte $p$ für Auswahlart <i>tournament selection</i> (intelligentes Zielobjekt, Geschwindigkeit 2) . . . . .	72
4.1 Schematische Darstellung der Verteilung des <i>reward</i> an <i>action set</i> Listen . . . . .	79
4.2 Schematische Darstellung der zeitlichen Verteilung des <i>reward</i> an und der Speicherung von <i>action set</i> Listen . . . . .	81
4.3 Schematische Darstellung der Bewertung von <i>action set</i> Listen bei einem neutralen Ereignis . . . . .	81
4.4 Vergleich verschiedener Werte für <i>maxStackSize</i> . . . . .	83
5.1 Zusammenhang zwischen der Abdeckung und der Qualität eines Algorithmus . . . . .	91
5.2 Auswirkung der Zielgeschwindigkeit (Zielobjekt mit einfacher Richtungsänderung, Säulenszenario) auf Agenten mit zufälliger Bewegung . . . . .	95
5.3 Auswirkung der Zielgeschwindigkeit (Zielobjekt mit einfacher Richtungsänderung, Säulenszenario) auf Agenten mit bestimmten Heuristiken . . . . .	95
5.4 Auswirkung der Zielgeschwindigkeit (intelligentes Zielobjekt, Säulenszenario) auf Agenten mit Heuristiken . . . . .	96
5.5 Auswirkung der Zielgeschwindigkeit (intelligentes Zielobjekt, Szenario mit zufällig verteilten Hindernissen, $\lambda_h = 0,2$ , $\lambda_p = 0,99$ ) auf Agenten mit Heuristik . . . . .	97
5.6 Auswirkung der Anzahl der Schritte (schwieriges Szenario, Geschwindigkeit 2, ohne Richtungsänderung) auf Qualität von Agenten mit Heuristik . . . . .	98
5.7 Vergleich der Qualitäten verschiedener Auswahlarten für XCS . . . . .	103
5.8 Vergleich der Qualitäten verschiedener Algorithmen bezüglich der Geschwindigkeit des Zielobjekts . . . . .	104
5.9 Auswirkung der Anzahl der Schritte (schwieriges Szenario, Geschwindigkeit 2, ohne Richtungsänderung) auf Qualität von Agenten mit XCS und SXCS . . . . .	108

5.10	Auswirkung der Anzahl der Schritte (schwieriges Szenario, Geschwindigkeit 2, ohne Richtungsänderung) auf Qualität von Agenten mit XCS und SXCS	108
5.11	Qualität bei unterschiedlicher Anzahl von Problemen bei gleichbleibender Gesamtzeit (schwieriges Szenario) . . . . .	109
5.12	Verlauf des gleitenden Durchschnitts der Qualität (schwieriges Szenario) . .	110
6.1	Auswirkung des Parameters <i>learning rate</i> $\beta$ auf den gleitenden Durch- schnitt der Qualität (Schwieriges Szenario) . . . . .	120
6.2	Screenshot des Konfigurationsprogramms (Gesamtübersicht) . . . . .	125
6.3	Screenshot des Konfigurationsprogramms (Konfigurationsbereich) . . . . .	126

# Tabellenverzeichnis

3.1	Verwendete Parameter (soweit nicht anders angegeben) und Standardparameter . . . . .	74
5.1	Zufällige Sprünge des Zielobjekts im leeren Szenario ohne Hindernisse . . .	88
5.2	Zufällige Sprünge des Zielobjekts in einem Säulenszenario . . . . .	88
5.3	Zufällige Sprünge des Zielobjekts in einem Szenario mit Hindernissen . . .	90
5.4	Vergleich von Zielobjekt mit zufälliger Bewegung und einfacher Richtungsänderung (leeres Szenario ohne Hindernisse) . . . . .	92
5.5	Vergleich von Zielobjekt mit zufälliger Bewegung und einfacher Richtungsänderung (zufälliges Szenario mit $\lambda_h = 0,1$ , $\lambda_p = 0,99$ ) . . . . .	93
5.6	Vergleich von Zielobjekt mit zufälliger Bewegung und einfacher Richtungsänderung (Säulenszenario) . . . . .	93
5.7	Vergleich der verschiedenen Auswahlarten (Zielobjekt mit einfacher Richtungsänderung, Säulenszenario, Geschwindigkeit 1, Agenten mit SXCS Algorithmus) . . . . .	101
5.8	Verschiedene Szenarien mit intelligentem Zielobjekt, Geschwindigkeit 1, 2000 Schritten, andauernder <i>exploit</i> Phase . . . . .	106
5.9	Verschiedene Szenarien mit Zielobjekt mit einfacher Richtungsänderung, Geschwindigkeit 1 und 2000 Schritten . . . . .	107



# Programmverzeichnis

A.1	Zentrale Schleife für einzelne Experimente . . . . .	128
A.2	Zentrale Schleife für einzelne Probleme . . . . .	129
A.3	Zentrale Bearbeitung (Sensordaten und Berechnung der neuen Aktion) aller Agenten und des Zielobjekts innerhalb eines Problems . . . . .	130
A.4	Zentrale Bearbeitung (Verteilung des <i>reward</i> Werts) aller Agenten und des Zielobjekts innerhalb eines Problems . . . . .	130
A.5	Zentrale Bearbeitung (Ausführung der Bewegung) aller Agenten und des Zielobjekts innerhalb eines Problems . . . . .	131
A.6	Berechnung der nächsten Aktion bei der Benutzung des Algorithmus mit zufälliger Bewegung . . . . .	132
A.7	Berechnung der nächsten Aktion bei der Benutzung der einfachen Heuristik	132
A.8	Berechnung der nächsten Aktion bei der Benutzung der intelligenten Heuristik	133
A.9	Korrigierte Version der <i>addNumerosity()</i> Funktion . . . . .	136
A.10	Erstes Kernstück des Standard XCS Multistepverfahrens ( <i>calculateReward()</i> , Bestimmung und Verarbeitung des <i>reward</i> Werts anhand der Sensordaten), angepasst an ein dynamisches Überwachungsszenario, bei positivem <i>reward</i> Wert wird nicht abgebrochen . . . . .	137

A.11 Zweites Kernstück des XCS <i>multi step</i> Verfahrens ( <i>collectReward()</i> - Verteilung des <i>reward</i> Werts auf die <i>action set</i> Listen), angepasst an ein dynamisches Überwachungsszenario . . . . .	138
A.12 Drittes Kernstück des XCS <i>multi step</i> Verfahrens ( <i>calculateNextMove()</i> , Auswahl der nächsten Aktion und Ermittlung der zugehörigen <i>action set</i> Liste), angepasst an ein dynamisches Überwachungsszenario . . . . .	139
A.13 Erstes Kernstück des SXCS-Algorithmus ( <i>calculateReward()</i> , Bestimmung des <i>reward</i> Werts anhand der Sensordaten) . . . . .	140
A.14 Zweites Kernstück des SXCS-Algorithmus ( <i>collectReward()</i> - Verteilung des <i>reward</i> Werts auf die <i>action set</i> Listen) . . . . .	141
A.15 Drittes Kernstück des SXCS-Algorithmus ( <i>calculateNextMove()</i> - Auswahl der nächsten Aktion und Ermittlung und Speicherung der zugehörigen <i>action set</i> Liste) . . . . .	142



# 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 von XCS haben sich hauptsächlich auf statische Szenarien mit nur einem Agenten 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 Überwachungsszenarien besser besteht als Agenten mit zufälliger Bewegung. In einem Überwachungsszenario ist die Aufgabe,

dass die sich dort befindlichen Agenten so positionieren, dass ein sich bewegendes Zielobjekt möglichst in jedem Schritt in unmittelbarer Nähe eines Agenten befindet. Da die Agenten nur lokale Information besitzen und ein solches Szenario aufgrund der Bewegung des Zielobjekts und der anderen Agenten dynamisch ist, lässt sich das Problem nicht ohne weiteres auf ein *single step* noch auf ein *multi step* Problem reduzieren.

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, insbesondere anhand empirischer Studien zu untersuchen, welche Anpassungen speziell für das Überwachungsszenario erfolgsversprechend sind.

Wesentliche Schwerpunkte der Untersuchung ist die Untersuchung des Szenarios ohne lernende Agenten, die Analyse der Bewertungsfunktion, die Bestimmung einer geeigneten Auswahlart für Aktionen der Agenten und die Bestimmung optimaler Parameter. Auf Basis dessen wird ein neuer Algorithmus entwickelt („SXCS“) und in mehreren Tests ausführlich mit der bekannten Standardimplementation verglichen.

Wesentliche Erkenntnisse aus dieser Arbeit sind:

- Die Bewertungsfunktion kann anhand einer Nachbildung einer gut funktionierenden Heuristik konstruiert werden.
- Ein Wechsel der Auswahlart für Aktionen während eines Laufs kann sinnvoll sein, um seltene Situationen besser zu lernen.
- Der hier vorgestellte Algorithmus erreicht in fast allen Szenarien ein deutlich besseres

Ergebnis als die Standardimplementation.

Desweiteren bietet die Arbeit zahlreiche Ansatzmöglichkeiten für spätere Arbeiten.

## 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) gibt es im Vergleich zum LCS vor allem innerhalb der Funktion zur Berechnung der *fitness* Werte der *classifier* Unterschiede. Während der *fitness* Wert beim einfachen LCS lediglich auf der Differenz zwischen erwarteter und tatsächlicher Bewertung basierte, wird bei XCS der *fitness* Wert auf Basis einer speziellen *accuracy* Funktion berechnet. Eine ausführliche Beschreibung findet sich in [But06b].

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

### 1.1.1 Beschreibung und 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.

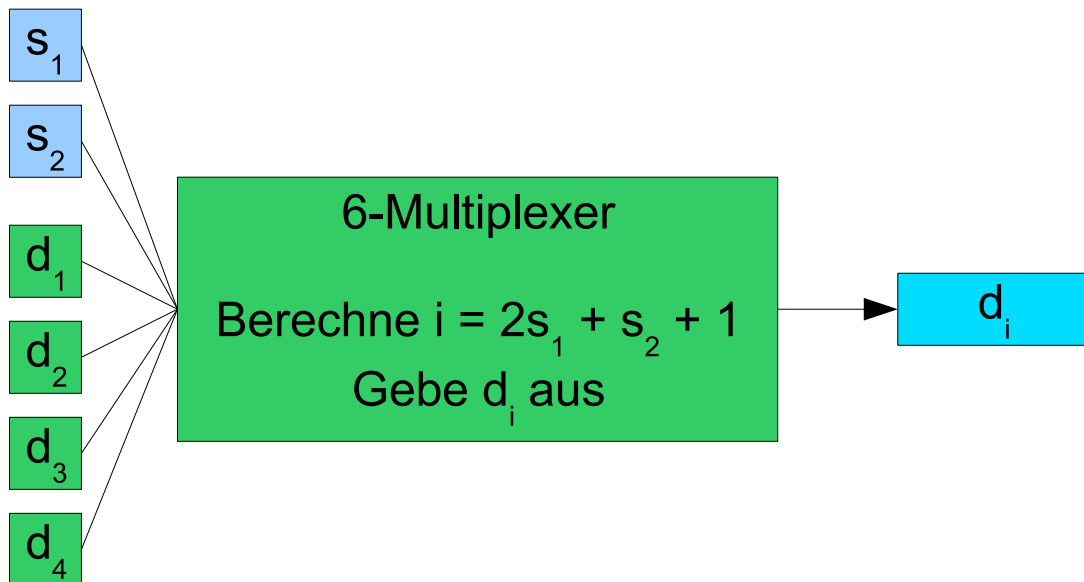


Abbildung 1.1: Schematische Darstellung des Das 6-Multiplexer Problems

### 1.1.2 Beschreibung und Beispiel für das *multi step* Verfahren

Ein klassisches Beispiel für *multi step* Verfahren ist das *Maze N* Problem, bei dem durch ein Labyrinth mit dem kürzesten Weg von  $N$  Schritten gegangen werden muss. Am Ziel angekommen wird der zuletzt aktivierte *classifier* positiv bewertet und das Problem neu gestartet. Bei den Wiederholungen erhält jede Regel einen Teil der Bewertung des folgen-

den *classifier*. Somit wird eine ganze Kette von *classifier* bewertet und sich der optimalen Wahrscheinlichkeitsverteilung angenähert, welche repräsentiert, welche der Regeln in welchem Maß am Lösungsweg beteiligt sind.

Als Demonstration soll das in Abbildung 1.2 dargestellte (sehr einfache) Szenario dienen. Die zum Agenten zugehörigen *classifier* 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 *classifier* 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.

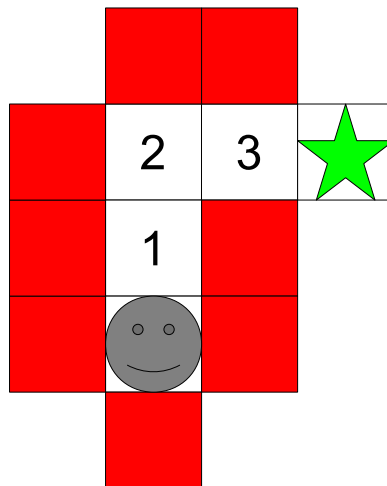


Abbildung 1.2: Einfaches Beispiel zum XCS *multi step* Verfahren

Die in dieser Arbeit verwendete Implementierung entspricht im Wesentlichen der Standardimplementierung des *multi step* Verfahrens von [But00]. Die algorithmische Beschrei-

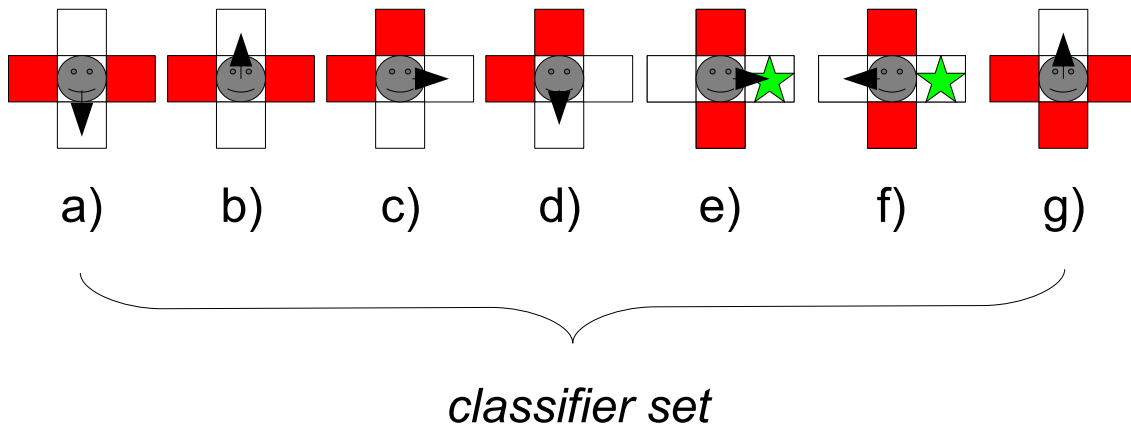


Abbildung 1.3: Vereinfachte Darstellung eines *classifier set* für das Beispiel zum XCS *multi step* Verfahren

bung 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.

### 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 [But06b] [Wil95] [Wil98], während XCS mit

Problemen größerer Schrittzahl zwischen Start und Ziel Probleme hat [BDE<sup>+</sup>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 Überwachungsszenario konnte jedoch in keiner dieser Arbeiten gefunden werden.

TODO ausführlicher!

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

Eine weitere Arbeit in dieser Richtung [HFA02] beschreibt das „El Farol“ Bar Problem

(EFBP) in Verbindung mit XCS und einem Multiagentensystem. Im EFBP geht es um eine Bar und eine Anzahl von Personen, jede Person kann entscheiden, ob sie die Bar besucht oder nicht. Entscheiden sich zuviele Personen für einen Besuch, dann gibt es *reward* für keine Person. Besucht eine Person von sich aus die Bar nicht, gibt es ebenfalls keinen *reward*. In der Arbeit wurde eine Methode benutzt („MAXCS“), um (in Verbindung mit XCS) kooperativ den *reward* zwischen den Personen zu verteilen und die Ergebnisse mit egoistisch handelnden Personen verglichen. Als Ergebnis wurde eine Emergenz festgestellt, d.h. die Agenten kooperierten miteinander und die Aufgabe konnte optimal gelöst werden. Auch hier wieder ist es ein dynamisches Szenario, die Vergleichbarkeit ist aber sehr eingeschränkt, da es sich bei dem EFBP ebenfalls um ein *single step* Problem handelt.

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

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, mit „schwieriger“ sind aber Probleme gemeint, die mit dem *multi step* Verfahren gel

. Ziel war es aber hier nicht, mit einer Dynamik zurechtzukommen, sondern lediglich Probleme mit längeren Wegen

Bei einem Gradientenabstieg wird ein

Ein konkreter Bezug zu einem dynamischen Überwachungsszenario konnte jedoch in keiner dieser Arbeiten gefunden werden.

Kapitel 3.4, eingehen auf reward Funktion



## 1.2 Aufbau der Arbeit

Kapitel 1.1 stellt den gegenwärtigen Stand der Forschung dar, insbesondere in Bereichen, die sich mit dem Thema dieser Arbeit schneiden. Kapitel 2 geht dann auf das verwendete Szenario, die Eigenschaften der Objekte und vor allem die Eigenschaften der Agenten und des Zielobjekts ein. Schließlich wird erläutert, wie die Simulation auf dem beschriebenen Szenario ablaufen soll. In Kapitel 3 werden dann die wichtigsten Teile des XCS vorgestellt, insbesondere die sogenannten *classifier*, die Verarbeitung von Sensordaten, der allgemeine Ablauf und die XCS Parameter. Vorbereitend für die Entwicklung neuer XCS Varianten sind insbesondere Kapitel 3.4, bei dem es um die Konstruktion einer passenden *reward* Funktion für die beschriebenen Szenarien geht, und Kapitel 3.5.5, bei dem es um die Frage geht, wann sich ein Agent für welche Aktion entscheiden soll, zu nennen. 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. Der wesentliche Höhepunkt folgt dann in Kapitel 5 in dem alle diskutierten 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 Gesamtzeit bewertet, in der er mit Hilfe der Agenten das Zielobjekt überwachen konnte (siehe Kapitel 2.6.3). Läuft der Test eines Algorithmus beispielsweise über 20000 Zeiteinheiten und konnten Agenten das Zielobjekt in 4000 Zeiteinheiten das Ziel überwachen, ergäbe sich eine Qualität von 20%.

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

Die Felder können entweder leer oder durch ein Objekt besetzt sein. Besetzte Felder

können nicht betreten werden, eine Bewegung auf ein solches Feld schlägt ohne weitere Konsequenzen fehl.

Es gibt drei verschiedene Arten von Objekten: Unbewegliche Hindernisse, ein zu überwachendes Zielobjekt und Agenten. Sowohl das Zielobjekt als auch die Agenten bewegen sich jeweils anhand eines bestimmten Algorithmus und bestimmter Sensordaten. Eine nähere Beschreibung der Agenten findet sich in Kapitel 2.3.4, während die Eigenschaften des Zielobjekts in Kapitel 2.5 beschrieben werden.

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

Eine separate Beschäftigung mit diesen - relativ einfachen - Szenarien 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 einer Anzahl von Feldern, einem Agenten, Hindernissen und einem Ziel. Es fehlen jedoch Arbeiten TODO, 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 TODO „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 way that is suitable or even optimal for goal attainment

TODO Literatur Definition von 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 3.6 beim Vergleich unterschiedlicher XCS Parameter) unterschiedliche Geschwindigkeiten getestet werden.

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

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, welcher 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 Fazor, 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 sowie 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 Überwachungsreichweite 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 die Sicht nicht versperrt wird und kein Agent gegen Hindernisse laufen (und stehenbleiben) muss.

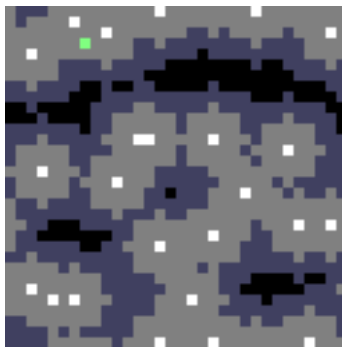


Abbildung 2.1: „Leeres Szenario“ ohne Hindernisse

### 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  $\lambda_h$ ) den Prozentsatz an Hindernissen an

der Gesamtzahl der Felder des Torus und der zweite Parameter (Verknüpfungsfaktor  $\lambda_p$ ) den Grad inwieweit die Hindernisse zusammenhängen.

Bei der Erstellung des Szenarios bestimmt  $\lambda_p$  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  $\lambda_h$  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 in Tests sich später auf Werte bis 0,2 beschränkt wird, da bei großem Hindernissanteil die lokalen Entscheidungen einzelner Agenten zu wichtig werden, da das Zielobjekt sich eher in einem kleinen Bereich aufhält.

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

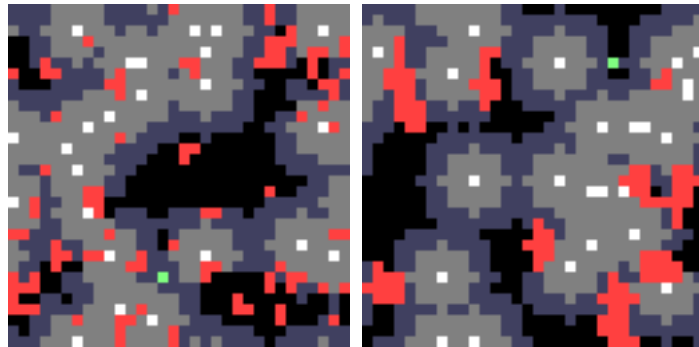


Abbildung 2.2: Szenario mit zufällig verteilten Hindernissen mit Hindernissanteil  $\lambda_h = 0,1$  und Verknüpfungsfaktor  $\lambda_p = 0,5$  bzw. 0,99.



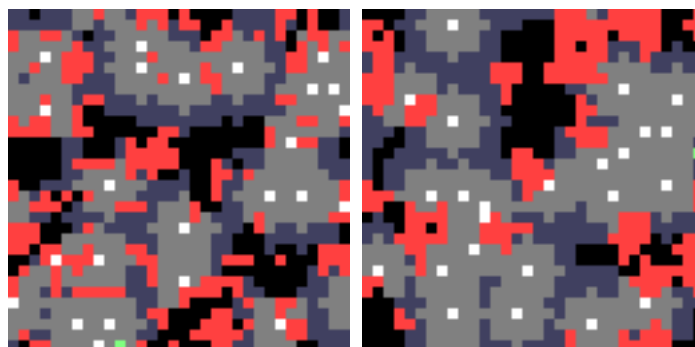


Abbildung 2.3: Szenario mit zufällig verteilten Hindernissen mit Hindernissanteil  $\lambda_h = 0,2$  und Verknüpfungsfaktor  $\lambda_p = 0,5$  bzw.  $0,99$ .

### 2.2.3 Säulenszenario

In diesem Szenario werden regelmäßig, mit jeweils 7 Feldern Zwischenraum zueinander, Hindernisse auf dem Torus verteilt. 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 2.4 zeigt ein Beispiel für den Startzustand eines solchen Szenarios, bei der das Zielobjekt sich in der Mitte und die Agenten am Rand befinden.

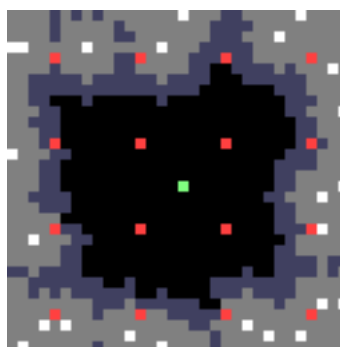


Abbildung 2.4: Startzustand des Säulen Szenarios mit regelmäßig angeordneten Hindernissen und zufälliger Verteilung von Agenten mit möglichst großem Abstand zum Zielobjekt

### 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) befinden sich vertikale Reihen von Hindernissen mit hindernissfreien Öffnungen von jeweils 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 5.9.1 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 2.5 zeigt die Startkonfiguration des Szenarios.

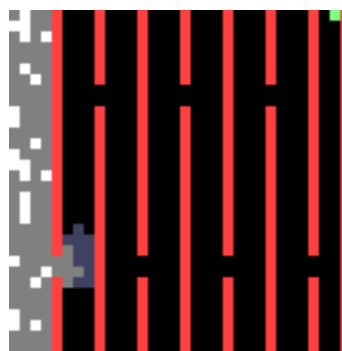


Abbildung 2.5: Schwieriges Szenario mit fester, wallartiger Verteilung von Hindernissen, in regelmäßigen Abständen und mit Öffnungen, mit den Agenten mit zufälligem Startpunkt am linken Rand und mit dem Zielobjekt mit festem Startpunkt rechts oben

## 2.3 Eigenschaften der Objekte

Jeder Agent bzw. das Zielobjekt besitzt eine Anzahl visueller, binärer Sensoren mit begrenzter Reichweite. Jeder Sensor ist beschränkt darauf, feststellen zu können, ob sich in seinem Sichtbereich ein Objekt eines bestimmten Typs befindet (1) oder nicht (0). Die Sensoren sind jeweils in eine bestimmte Richtung ausgerichtet, wobei andere Objekte die Sicht blockieren. Die dazu nötigen 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. 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 werden dann in Kapitel 2.3.4 beschrieben.

### 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, in welcher der beiden Reichweiten sich Objekte befinden (oder ob keine Objekte in Sicht sind). Falls nicht anders angegeben, werden jeweils *sight range* auf 5 und *reward range* auf 2 gesetzt.

### 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 eine Teilmenge des sichtbaren Bereichs. In Abbildung 2.6 sind alle Sichtreichweiten (heller und dunkler Bereich) und Überwachungsreichweiten (heller Bereich) für die einzelnen Richtungen dargestellt.

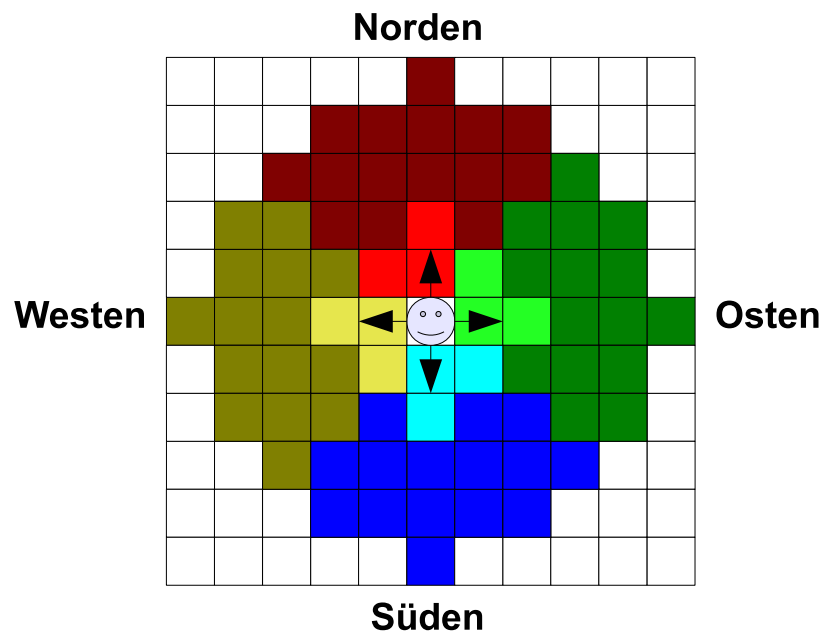


Abbildung 2.6: Sicht- (5,0, dunkler Bereich) und Überwachungsreichweite (2,0, heller Bereich) eines Agenten, jeweils für die einzelnen Richtungen

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

### 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 2.7 stellt den allgemeinen Aufbau eines kompletten Sensordatensatzes dar, der aus den drei Gruppen der Zielobjektsensoren (z), der Agentensensoren (a) und der Hinernissensoren (h) besteht:

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

Abbildung 2.7: Darstellung des Sensordatensatzes, eingeteilt in mehrere Gruppen und Sensorpaare

Befindet sich beispielsweise das Zielobjekt außerhalb der Überwachungsreichweite aber innerhalb der Sichtweite im Norden, befinden sich im Süden ein oder mehrere Agenten in Überwachungsreichweite und befinden sich im Westen und Osten ebenfalls in Überwachungsreichweite des Agenten befindliche Hindernisse, dann ergibt sich ein Sensordaten-

satz  $s_{\text{Beispiel}}$  wie in Abbildung 2.8 dargestellt.

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

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

### 2.3.4 Eigenschaften der Agenten und des Zielobjekts

Ein Agent kann in jedem Schritt zwischen vier verschiedenen Aktionen wählen, die den vier 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 nacheinander in der Art abgearbeitet, so dass jeder Agent die aktuellen Sensordaten (siehe Kapitel 2.3) aus der Umgebung holt und auf deren Basis die nächste Aktion bestimmt.

Wurden alle Aktionen bestimmt, können die Agenten in zufälliger Reihenfolge versuchen sie auszuführen. Ungültige Aktionen, d.h. der Versuch sich auf ein besetztes Feld zu bewegen, schlagen fehl und der Agent führt in diesem Schritt keine Aktion aus, wird aber auch nicht weiter bestraft. Eine detaillierte Beschreibung der Bewegung, im Kontext anderer Agenten und Programmteile, wird in Kapitel 2.6.4 gegeben.

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 Schritts bestimmt. Springt das Zielobjekt beispielsweise auf zufällige Felder auf dem Torus (siehe Kapitel 2.5.1), dann existiert keine Verbindung zwischen den Positionen des Zielobjekts zweier aufeinanderfolgender Zeiteinheiten und das 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 wird der Sonderfall erwähnt für ein Zielobjekt, das nur in dieselbe Richtung läuft (siehe Kapitel 2.5.5). Es soll ausschließlich in Verbindung mit dem schwierigen Szenario getestet werden.

## 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 werden also insbesondere die Implementierungen der jeweiligen *calculateNext-*

*Move()* Funktion vorgestellt.

### 2.4.1 Algorithmus mit zufälliger Bewegung

Bei diesem Algorithmus wird in jedem Schritt eine zufällige Aktion ausgeführt. Abbildung 2.9 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.

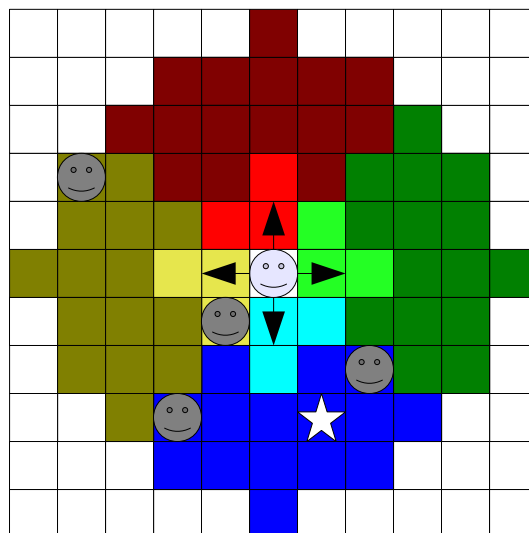


Abbildung 2.9: Agent bewegt sich in eine zufällige Richtung (oder bleibt stehen).

### 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 2.10 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.

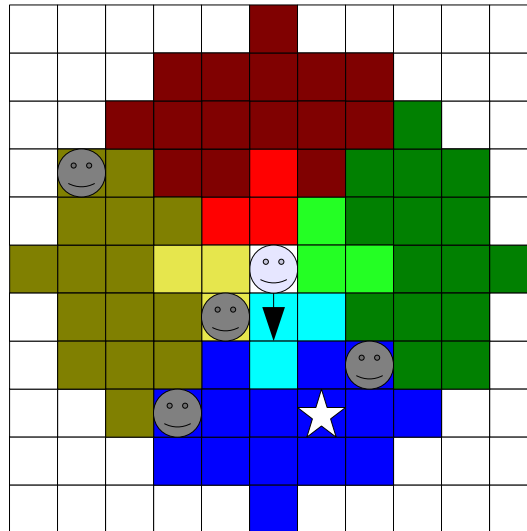


Abbildung 2.10: Agent mit einfacher Heuristik: Sofern es sichtbar ist bewegt sich der Agent auf das Zielobjekt zu.

### 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. In [Abbildung 2.11](#) 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.

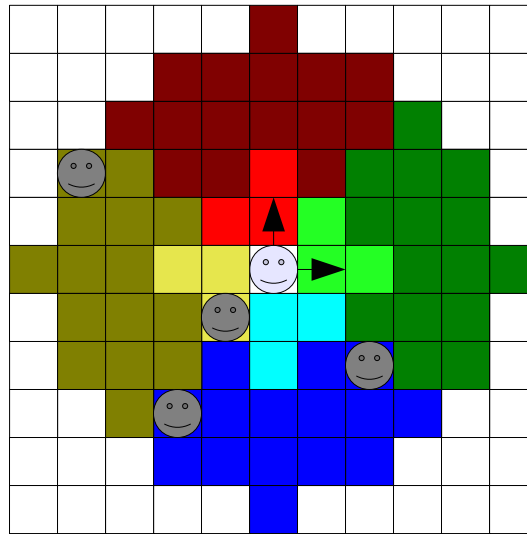


Abbildung 2.11: Agent mit intelligenter Heuristik: Falls das Zielobjekt nicht sichtbar ist, bewegt sich der Agent von anderen Agenten weg.

## 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 2.12 sind alle Felder grau markiert, die das Zielobjekt innerhalb von zwei Schritten erreichen kann, nachdem er sich einmal nach Norden bewegt hat.

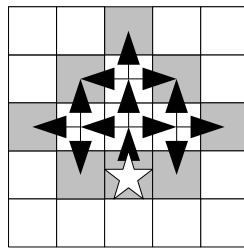


Abbildung 2.12: Zielobjekt macht pro Schritt maximal eine Richtungsänderung

### 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 gestrichen, 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 2.13 wird die Richtung Süden gestrichen, da sich dort ein Agent in Überwachungsreichweite befindet. Die Richtungen Westen und Norden werden jeweils mit 50%iger Wahrscheinlichkeit gestrichen, da sich dort Agenten in Sichtweite befinden. Nur Richtung Osten wird als Möglichkeit sicher übrig bleiben.

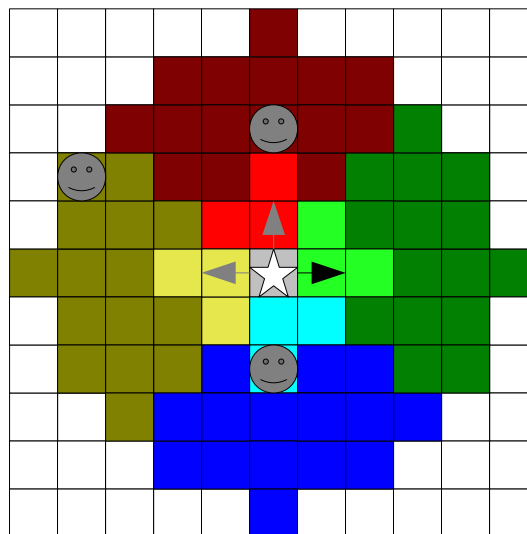


Abbildung 2.13: Ein sich intelligent verhaltendes Zielobjekt weicht Agenten aus.

### 2.5.5 Typ „Beibehaltung der Richtung“

Ein Zielobjekt dieses Typs versucht, immer in Richtung Norden zu gehen. Ist das Ziel-feld blockiert, wählt es ein zufälliges, angrenzendes, freies Feld im Westen oder Osten. Anzumerken 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 2.14 sind drei Situationen dargestellt, zum einen ein wiederholtes Hin- und Herlaufen unter den Hindernissen, der Weg links um die Hindernisse herum und der



## 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 Schritten zu betrachten und zu vergleichen. Die Zahl der Schritte 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 2.15 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. Vereinzelt wurde zur besseren Übersicht statt der Qualität die Qualitätsdifferenz angegeben. Sie bezeichnet jeweils die Differenz der Qualität des jeweils betrachteten Algorithmus zur Qualität eines Agenten mit zufälliger Bewegung.

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

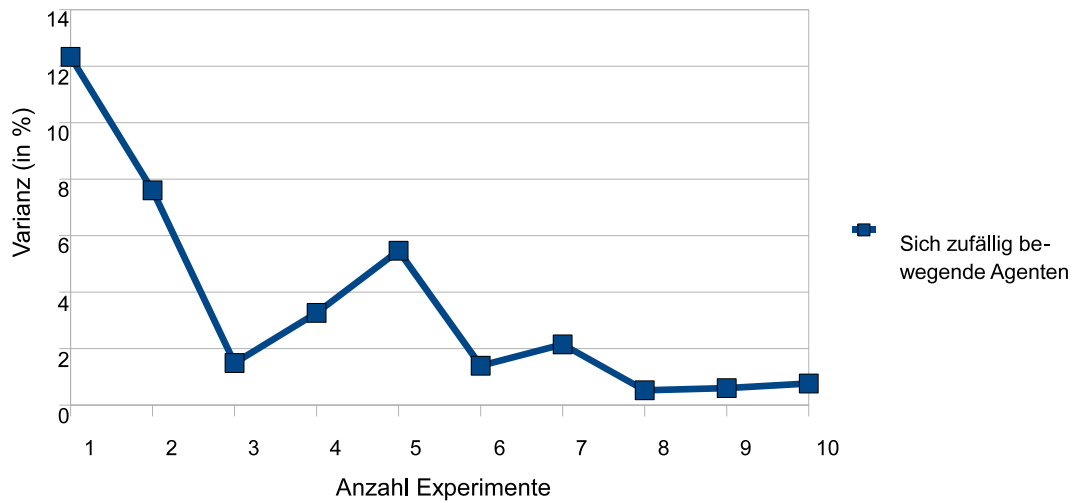


Abbildung 2.15: Varianz der Testergebnisse bei unterschiedlicher Anzahl von Experimenten (Säulenszenario, Zielobjekt mit einfacher Richtungsänderung und Geschwindigkeit 2, Agenten mit zufälliger Bewegung)

dem über die erwähnte Anzahl von Schritten die Simulation abläuft. Die Anfangsbelegung des Torus ist über einen *random seed* Wert bestimmt, wobei bei jeder Problemistanz mit einem neuen Wert initialisiert wird, der sich aus der Nummer des Experiments und der Nummer des Problems berechnet. Die Problemistanzen sind also untereinander unterschiedlich, jedoch mit anderen Testdurchläufen mit einer anderen Konfiguration vergleichbar. Soweit nicht anders angegeben, sollen hier Problemistanzen 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 wird 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. dienen 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 Werten 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 Feldern, 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 bezieht sich in dieser Arbeit auf die Betrachtung des Verhaltens des Algorithmus über einen ganzen Testlauf hinweg, es wird also nicht z.B. nur am Ende gemessen oder (wie bei XCS) nur bei bestimmten Problemen.

Ein Testlauf mit einer bestimmten Konfiguration wird über mehrere Experimente durchgeführt, wobei in jedem Experiment mehrere Probleme ausgeführt werden.

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

Die Qualität eines Algorithmus zu einer Anzahl von Problemen (also einem Experiment) wird anhand des Anteils der Gesamtzeit aller Probleme berechnet, die er das Zielobjekt während aller Probleme überwachen konnte.

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.

TODO vielleicht noch fett machen die einzelnen Wörter...

### 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 *doOneMultiStepProblem()* aufgerufen, welche in Programm A.2 abgebildet ist. Dort werden in einer Schleife alle Schritte durchlaufen und jeweils die Objekte abgearbeitet.

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

Beim zweiten Punkt ergibt sich der Umstand, dass später abgearbeitete Agenten aktuellere Informationen über die Umwelt besitzen, insbesondere schlägt bei den später ausgeführten Agenten eine Aktion mit geringerer Wahrscheinlichkeit fehl, sofern Hindernisse in der Bewegung beachtet werden. Umgekehrt können früh ausgeführte Agenten eher gute Positionen besitzen. Da diese zwei Umstände ein schwierig zu beurteilenden Faktor in die Untersuchungen einbringen, wird für die Simulation Punkt 1 gewählt, jeder Agent entscheidet seine Aktion also auf Basis von Sensordaten vom selben, gemeinsamen Zeitpunkt.

### **2.6.5 Reihenfolge bei unterschiedlichen Bewegungsgeschwindigkeiten**

Bezüglich der Bewegung ergibt sich eine weitere Frage, nämlich wie unterschiedliche Bewegungsgeschwindigkeiten behandelt werden sollen. Zwar haben alle Agenten eine Einheitsgeschwindigkeit von einem Feld pro Zeiteinheit, jedoch kann sich das Zielobjekt je

nach Szenario gleich eine ganze Anzahl von Feldern bewegen (siehe auch Kapitel 2.5).

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

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.

### 2.6.6 Messung der Qualität

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

Da ein wesentlicher Bestandteil dieser Arbeit die Kollaboration (und somit die Abdeckung des Torus anstatt dem dauernden Verfolgen des Zielobjekts) ist, soll ein Bewertungskriterium sein, inwieweit der Einfluss des Zielobjekts minimiert werden kann. 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 bewegendem) Agenten zu behandeln.

### 2.6.7 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 Schritt 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 Überwachungsreichweite“  $\Rightarrow \frac{2}{3}$  *base reward*), ausgeschlossen werden soll. Insbesondere würde dies eine mehrfache Erhebung der Sensordaten erfordern.

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

TODO

### 2.6.8 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 (bezieht sich auf lernende Agenten),
4. Aktualisierung der Regeln anhand des *base reward* Werts (bezieht sich auf 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

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

Während die ersten drei Punkte bei allen hier vorgestellten XCS Varianten identisch sind, gibt es wesentliche Unterschiede bei der Bewertung der *classifier*. Diese werden gesondert in Kapitel 4 im Einzelnen besprochen. Im Folgenden werden nun Punkt 1, 2 und 3 näher betrachtet und das Kapitel mit einer Diskussion und Analyse der XCS Parameter in Kapitel 3.6 abgerundet.

## 3.1 Classifier

Ein *classifier* besteht aus einer Anzahl im folgenden diskutierten Variablen die anhand der in Kapitel 3.6 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 welchem der *classifier* ausgewählt und dessen *action* Wert ausgeführt wird.

### 3.1.1 Der *condition* Vektor

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

$$\underbrace{z_{sN} z_{rN} z_{sO} z_{rO} z_{sS} z_{rS} z_{sW} z_{rW}}_{\text{Erste Gruppe (Zielobjekt)}} \underbrace{a_{sN} a_{rN} a_{sO} a_{rO} a_{sS} a_{rS} a_{sW} a_{rW}}_{\text{Zweite Gruppe (Agenten)}} \underbrace{h_{sN} h_{rN} h_{sO} h_{rO} h_{sS} h_{rS} h_{sW} h_{rW}}_{\text{Dritte Gruppe (Hindernisse)}}$$

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. Hier erstreckt sich der Wertebereich zwischen 0,0 und 1,0 (maximale Genauigkeit). Insbesondere eines der ersten Werke zu XCS [Wil95] beschäftigte sich mit diesem Aspekt der Genauigkeit.

### 3.1.4 Der *reward prediction* Wert

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

### 3.1.5 Der *reward prediction error* Wert

Der *reward prediction error* Wert soll die 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

Mit dem *experience* Wert des *classifier* wird die Anzahl repräsentiert, 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 3.3.4) 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 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 3.5 beschrieben, der eigentliche *classifier* ausgewählt wird.

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

### 3.2.1 Erkennung von Sensordatenpaaren

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

Es ergeben sich also folgende Fälle:

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

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

Sensordaten:

(Zielobjekt in Sicht im Norden, Agent in Sicht im Süden,  
Hindernisse im Westen und Osten)

10 00 00 00 . 00 00 11 00 . 00 11 00 11

Beispiele für erkennende condition Vektoren:

```
10 00 00 00 . ## ## ## ## . 00 ## ## ##
## ## ## ## . ## ## #1 00 . 00 11 ## ##
#0 ## ## ## . ## ## 01 ## . ## 11 ## 11
```

### 3.2.2 Subsummation von *classifier*

Die Benutzung von den oben erwähnten Platzhaltern (Kapitel 3.2) erlaubt es dem XCS mehrere *classifier* zusammenzulegen, 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 von zwei verschiedenen Sensordatensätzen erbracht werden kann, geringer ist, als die Ersparnis, die durch das Zusammenlegen beider *classifier* entsteht.

Besitzt ein *classifier* sowohl einen genügend großen *experience* Wert als auch einen ausreichend kleinen *reward prediction error* Wert, so kann er als sogenannter *subsumer* auftreten. 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

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 3.5),
4. Erstellung der zur Aktion zugehörigen Liste von *classifier* (*action set* Liste, siehe Kapitel 3.3.3);

Im Folgenden werden die einzelnen Punkte besprochen:

### 3.3.1 Abdeckung aller Aktionen durch *covering*

Das *covering* untersucht die Menge aller *classifier* aus dem letzten *match set* (siehe Kapitel 3.3.2), 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 3.6.1 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 3.5 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 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 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 3.6 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 Bewertung der Aktionen (*base reward*)

XCS ist darauf ausgelegt, dass es eine komplette, genaue und möglichst allgemeine Darstellung einer *reward* Funktion darstellt. Die *reward* Funktion läuft lokal auf jedem Agenten ab und die Bewertung, die der Agent berechnet, wird also auf Basis der eigenen Sensordaten gebildet. Die Bewertung wird in diesem Zusammenhang im Folgenden als *base reward* Wert bezeichnet.

#### 3.4.1 Bewertung beim *single step* Verfahren

Bei einer Problemstellung, die mit dem *single step* Verfahren gelöst werden kann, entspricht die optimale Darstellung der *reward* Funktion durch das XCS gleichzeitig auch der Lösung des eigentlichen Problems. Beispielsweise 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 eigene Ausgabe zu bewerten.

### 3.4.2 Bewertung beim *multi step* Verfahren

Bei komplexeren Problemen, bei denen ein Agent nur lokale Informationen zur Verfügung hat (beispielsweise bei *Maze N* die angrenzenden Felder), bezieht die *reward* Funktion nur eine Teilinformation der Welt in die Bewertung ein, beispielsweise „1“ beim letzten Schritt auf das Ziel und „0“ sonst. Deshalb ist die optimale Darstellung der *reward* Funktion bei XCS in der *multi step* Variante die, dass der Aufbau eines Gesamtbilds über die Weitergabe mittels des jeweiligen *maxPrediction* Werts geschieht. Auf Basis dessen wird ein vereinfachter Gesamtweg gebildet (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.

### 3.4.3 Bewertung bei einem Überwachungsszenario

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

tionen (seit dem letzten Auftreten) absteigend mit dem Wert zu aktualisieren.

Offen bleibt die Frage, wie die *reward* Funktion beim Überwachungsszenario aussieht. Letztlich hat ein Agent freie Wahl, wie der *base reward* Wert aus den Sensordaten berechnet wird. Für die 24 Binärsensoren ergeben sich bis zu  $2^{24}$  Situationen, die wahrgenommen werden können (tatsächlich sind es weniger, da es nur ein Zielobjekt gibt und bestimmte Situationen nicht auftreten können) und jeder Situation könnte ein individueller *base reward* zugewiesen werden, was entsprechend viele Möglichkeiten für die *reward* Funktion ergibt.

Zur Beantwortung der Frage ist es notwendig, zu betrachten, wie das globale Ziel erreicht werden kann. Eine Möglichkeit ist, sich anhand der in Kapitel 2.4 vorgestellten Heuristiken zu orientieren. Auch wenn diese Heuristiken nicht direkt mit einem *base reward* arbeiten, bewerten sie doch in jeder Situation eine bestimmte Situation als gut oder schlecht. Erkennt beispielsweise ein Agent mit intelligenter Heuristik Agenten im Sichtbereich, wird er die jeweiligen Richtungen als schlecht bewerten, während er die freien Richtungen als gut bewertet.

Betrachtet man die Heuristiken näher führt dies zu folgenden Erkenntnissen:

- Agent mit zufälliger Bewegung: Dieser Algorithmus bewertet alle Situationen identisch, er benutzt also eine konstante *reward* Funktion.
- Agent mit einfacher Heuristik: Bei diesem Algorithmus wird jede Situation als gut bewertet, in der das Zielobjekt in Sicht ist.
- Agent mit intelligenter Heuristik: Hier werden ebenfalls alle Situationen mit einem Zielobjekt in Sicht als gut bewertet. Zusätzlich versucht ein Agent mit dieser

Heuristik sich aus der Sicht anderer Agenten zu bewegen, bewertet im Grunde als Situationen besser, in denen mehr Richtungen frei von Agenten sind.

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

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

Bei den Tests siegt eindeutig der Agent mit intelligenter Heuristik. Allerdings ist es schwierig, oben beschriebene *reward* Funktion zu modellieren, da in dem Rahmen dieser Arbeit lediglich binäre Ausgabewerte möglich sind. Um mehrere unterschiedliche Situationen zu differenzieren (Anzahl der Richtungen in denen kein Agent in Sicht ist), wären aber mehrere Zustände Voraussetzung. Somit kann die intelligente Heuristik nicht als Modell stehen und stellt nur ein oberes Limit für die mögliche Qualität (ohne Einbeziehung von Hindernissen) dar.

Bleibt nur die einfache Heuristik, deren *reward* Funktion deshalb für die hier besprochenen XCS Varianten benutzt wird, der *base reward* wird auf „1“ gesetzt, wenn das Zielobjekt in Sicht ist, sonst auf „0“.

Was die Implementierung selbst betrifft, wird die mit der Bewertung zusammenhängende Funktion *checkRewardPoints()* in Programm A.10 (Zeile 15) bzw. in Programm A.13 (Zeile 16) aufgerufen.

## 3.5 Auswahlart der *classifier*

In jedem Schritt 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. Danach werden alle Produkte aus den *fitness* und *reward prediction* Werten der *classifier* aus der jeweiligen Gruppe aufaddiert und durch die Summe der *fitness* Werte der *classifier* der jeweiligen Gruppe geteilt. Dieser Wert soll im folgenden *predictionFitnessProductSum* genannt werden.

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

***random selection*** Zufällige Auswahl einer Aktion (identisch mit zufälliger Bewegung),

***roulette wheel selection*** Zufällige Auswahl einer Aktion, Wahrscheinlichkeit abhängig vom *predictionFitnessProductSum* Wert der jeweiligen Gruppe,

***best selection*** Auswahl der Aktion mit dem höchsten *predictionFitnessProductSum* Wert der jeweiligen Gruppe;

Im Folgenden werden nun diese Auswahlarten näher vorgestellt und außerdem noch eine weitere Auswahlart aus der Literatur besprochen. Im nächsten Abschnitt (Kapitel 3.5.5) wird dann der Wechsel zwischen diesen Auswahlarten näher untersucht und

abschließend die tatsächlichen Testergebnisse (Kapitel 5.6) zwischen den vorgestellten Varianten präsentiert.

### 3.5.1 Auswahlart *random selection*

Prägende Idee für diese Auswahlart in einem statischen Szenario ist, das XCS möglichst vielen verschiedenen Situationen auszusetzen. Da in einem statischen Szenario 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 Überwachungsszenario (siehe Kapitel 2.1) ist es im Vergleich zu standardmäßigen statischen Szenarien dagegen weder nötig noch hilfreich *random selection* zu nutzen, da sich oben genanntes Problem nicht ergibt. Zum einen ist, aufgrund ständiger Bewegung anderer Agenten und des Zielobjekts, nicht fixiertem Startpunkt und fehlendem Neustart beim Erreichen des Ziels (positivem *base reward* Wert), das Problem dynamisch und die Agenten werden mit vielen verschiedenen Situationen konfrontiert. Zum anderen ist es für ein erfolgreiches Bestehen in einem Überwachungsszenario 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.

### 3.5.2 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, wo genau sich das Zielobjekt befindet, selbst wenn es in Sicht ist. Dementsprechend muss eine optimale Verhaltensstrategie Entscheidungen auf Basis von Wahrscheinlichkeitsverteilungen treffen, weshalb die alleinige Verwendung der Auswahlart *best selection* eher nicht in Frage kommt.

### 3.5.3 Auswahlart *roulette wheel selection*

Bei dieser Auswahlart bestimmt der *predictionFitnessProductSum* Wert (relativ zu den anderen *predictionFitnessProductSum* Werten) die Wahrscheinlichkeit, ausgewählt zu werden. Diese Auswahlart erscheint sinnvoll, allerdings ist speziell bei diesem Szenario davon auszugehen, dass, wie auch schon in Kapitel 3.5.2 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*.

### 3.5.4 Auswahlart *tournament selection*

Zusätzlich zu den oben erwähnten drei Möglichkeiten wurde in [MVBG03] eine weitere vorgestellt und in Bezug auf XCS diskutiert. Sie wurde dort mit *tournament selection* bezeichnet und als Vorteile gegenüber den herkömmlichen Auswahlarten werden u.a. geringerer Selektionsdruck, höhere Effizienz und geringerer Einfluss von Störungen genannt. Durch die Anpassung der Turniergröße ergibt sich außerdem eine flexible Anpassungsmög-

lichkeit. In den dort vorgestellten Experimenten mit einem *single step* Problem wurden signifikante Vorteile dieser, auf proportionaler Selektion beruhender, Auswahlart 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 *reward prediction* und *fitness* Werte zusammengekommen werden, soll hier eine Implementation der Auswahlart *tournament selection* gewählt werden, die näher am ursprünglichen Algorithmus aus dem Bereich der genetischen Algorithmen liegt [MMGG95].

Charakteristisch für diese Auswahlart ist, dass

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

⇒ Das beste Element wird mit Wahrscheinlichkeit  $p$ , das zweitbeste mit Wahrscheinlichkeit  $(1,0 - p)p$ , das drittbeste mit Wahrscheinlichkeit  $(1,0 - p)^2p$  usw. gewählt.

In dem hier besprochenen Fall enthalten die Mengen immer 4 Elemente (Anzahl der Aktionen) und diese 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 ist diese Auswahlart deckungsgleich mit der *roulette wheel selection*, allerdings ohne dem Problem, dass die Auswahlwahrscheinlichkeit aufgrund ähnlicher Produkte sich ebenfalls ähneln. Außerdem ist die Darstellung selbst sehr flexibel, beispielsweise wäre *tournament selection* mit  $p = 1,0$  und  $k = 4$  identisch mit *best selection* und mit  $p = 1,0$  und  $k = 1$  wäre es identisch mit *random selection*. Diese Form der Auswahl, bei



geeigneter Wahl von  $k$  und  $p$ , scheint also sehr vielversprechend zu sein.

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

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 3.6.6 wurde der Wert 0,84 als für die hier betrachteten Szenarien optimaler Wert bestimmt.

### 3.5.5 Wechsel zwischen den *explore* und *exploit* Phasen

In der Standardimplementierung von XCS wird zwischen verschiedenen Auswahlarten gewechselt. Die Auswahlarten werden hierzu in zwei Gruppen geteilt, in die *explore* Phase und in die *exploit* Phase. In der *exploit* Phase soll 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. Wesentlicher Leitgedanke ist es, mit Hilfe der *explore* Phasen den Suchraum besser erforschen zu können, dann aber zur eigentlichen die konkrete Problemlösung in der *exploit* Phase möglichst direkt auf das Ziel zuzugehen um *classifier* stärker zu belohnen, die am kürzesten Weg beteiligt sind.

Die Wahl der Auswahlart in Kapitel 3.3 für *classifier* in Punkt (3) kann auf verschiedene Weise erfolgen. In der Standardimplementierung von XCS wird zwischen *exploit* und *explore* nach jedem Erreichen des Ziels entweder umgeschaltet oder zufällig mit einer bestimmten Wahrscheinlichkeit eine Auswahlart ermittelt. Es werden also abwechselnd ganze Probleme entweder nur in der *exploit* oder nur in der *explore* Phase 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, bei der nicht das Erreichen eines positiven *base rewards* (also ein Neustart des Problems) einen Phasenwechsel auslöst, sondern stattdessen eine *Änderung* des *base reward* Werts ausschlaggebend ist und einen Wechsel zwischen der *explore* und *exploit* Phase auslöst. Bei einer anfänglichen *explore* Phase würde dann immer in die *exploit* Phase gewechselt werden, wenn das Zielobjekt in Sicht ist (bzw. 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 Wechsels 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*), und sowie
5. in jedem Schritt zufällig entweder *explore* oder *exploit* Phase (50% Wahrscheinlichkeit jeweils)

Hervorzuheben ist, dass die Varianten (3.), (4.) und (5.) angewendet auf die Standardimplementierung des *multi step* XCS Verfahrens keinen Unterschied machen würden. Dies liegt daran, dass beim Erreichen eines positiven *base reward* Werts sowieso ein neues Problem gestartet wird, die *explore* und *exploit* Phasen separat betrachtet werden können und zwischen den Problemen zwischen der *exploit* und *explore* Phase gewechselt wird.

## 3.6 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. Nachstehend werden Parameter besprochen, die entweder in der Empfehlung offen gelassen sind, also klar vom jeweiligen Szenario abhängen, und solche, bei denen von der Empfehlung abgewichen wurde. Es wurden viele weitere Veränderungen getestet, in den meisten Fällen war die Standardeinstellung jedoch passend.

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

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. Mit dem Ziel, synchronisierte und vergleichbare Daten zu haben, wurden die Tests in mehreren Etappen durchgeführt. Damit entsprechen die hier aufgeführten Testergebnisse nur den endgültigen Ergebnissen. Der hohe Aufwand dieses Kapitels war notwendig, da der Unterschied zwischen den Qualitäten der verschiedenen Algorithmen eher gering war und keine Vergleichsarbeiten mit der hier verwendeten Problemstellung existieren.

### 3.6.1 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.

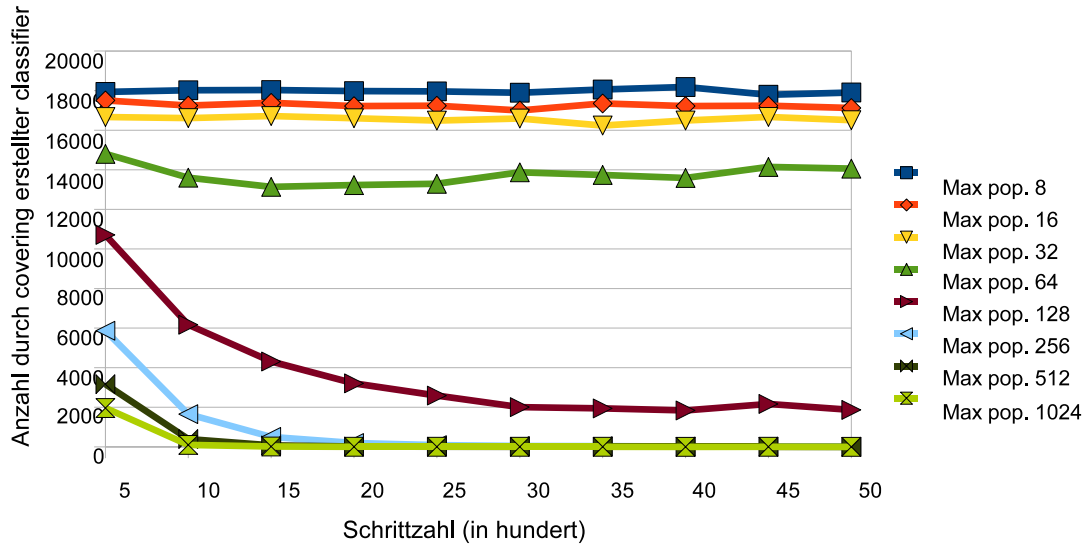


Abbildung 3.2: Auswirkung der maximalen Populationsgröße auf die Anzahl der *classifier* die durch *covering* neuerstellt werden (Säulenszenario, Zielobjekt mit einfacher Richtungsänderung, Geschwindigkeit 1, Agenten mit SXCS)

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. Wesentlichen Einfluss auf die Zahl der Möglichkeiten ist, 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. So könnten z.B. Zielobjekte in zwei unterschiedlichen Richtungen in Sichtweite eines Agenten kommen. 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.

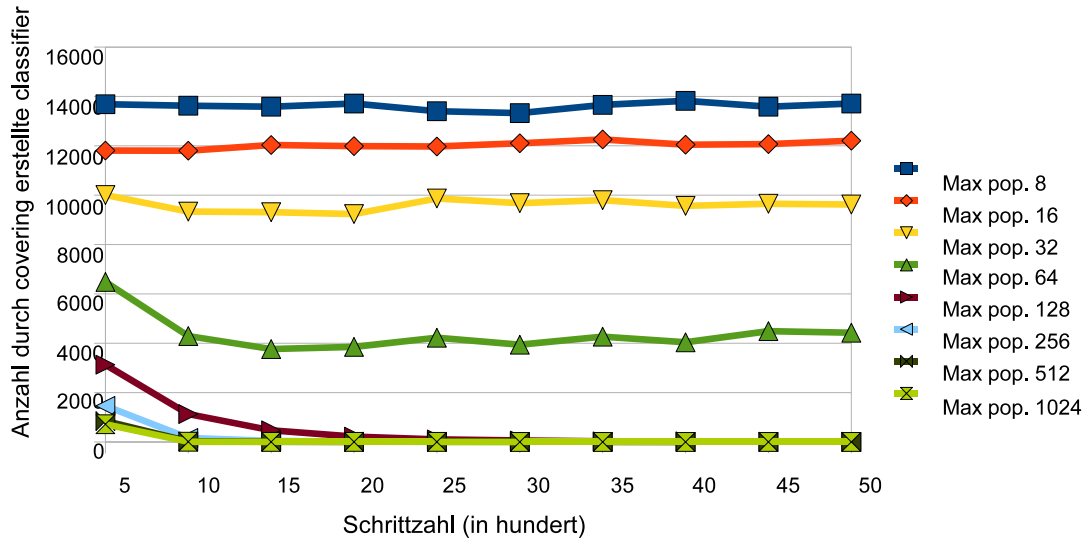


Abbildung 3.3: Auswirkung der maximalen Populationsgröße auf die Anzahl der *classifier* die durch *covering* neuerstellt werden (leeres Szenario ohne Hindernisse, Zielobjekt mit einfacher Richtungsänderung, Geschwindigkeit 1, Agenten mit SXCS)

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.

Die wichtigsten *classifier* sind mit Populationsgröße 256 (bzw. 128 im leeren Szenario) bereits abgedeckt. Eine Erhöhung der Populationsgröße führt daher lediglich 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 Torusgröße auf die Laufzeit betrachtet, wie in Abbildung 3.4 zu sehen, ist der Einfluss auf die Laufzeit im getesteten Bereich (16x16 bis 64x64) ohne Bedeutung.

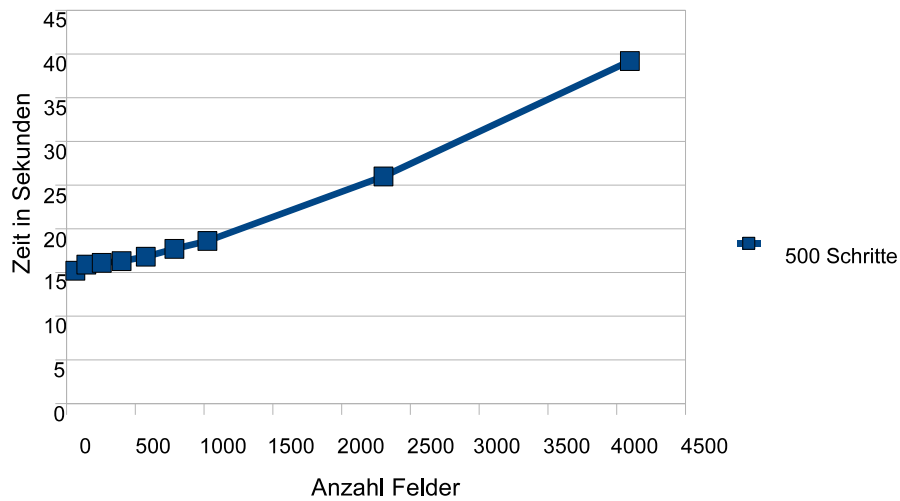


Abbildung 3.4: Darstellung der Auswirkung der Torusgröße auf die Laufzeit im leeren Szenario, zufälliger Bewegung des Zielobjekts, Geschwindigkeit 1, sich zufällig bewegenden Agenten

### 3.6.2 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 *classifier* mit zufälligen *action* Werten und *condition* Vektoren gefüllt wird. Dort wird aber auch angemerkt, dass sich beide Varianten in ihrer Qualität nur wenig unterscheiden. Da zum einen gewisser Zeitaufwand nötig ist, die Liste zu füllen und zum anderen nicht sichergestellt ist, dass die generierten *classifier* in dem jeweiligen Szenario überhaupt aktiviert werden können, scheint es sinnvoll zu sein mit

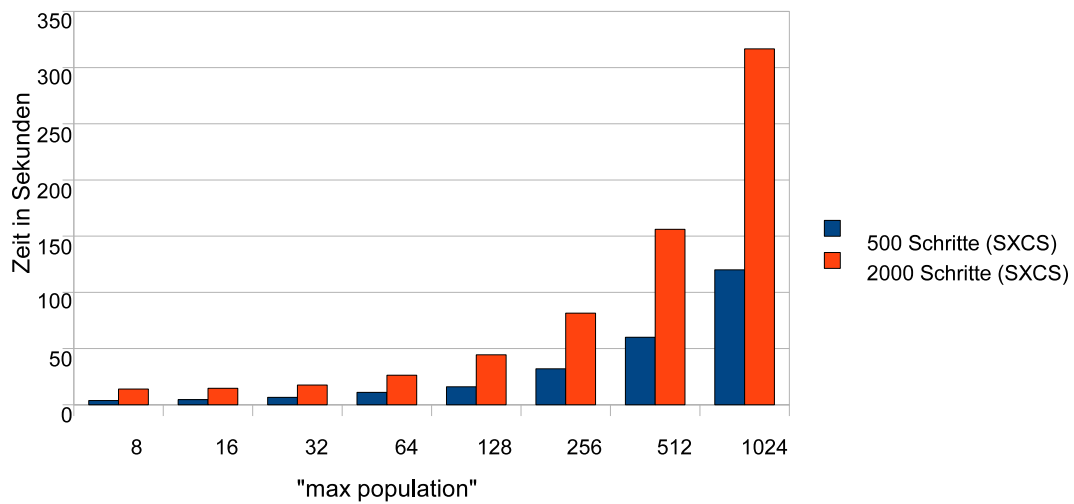


Abbildung 3.5: Darstellung der Auswirkung des Parameters *max population*  $N$  auf die Laufzeit im leeren Szenario, zufälliger Bewegung des Zielobjekts, Geschwindigkeit 1, Agenten mit SXCS Algorithmus

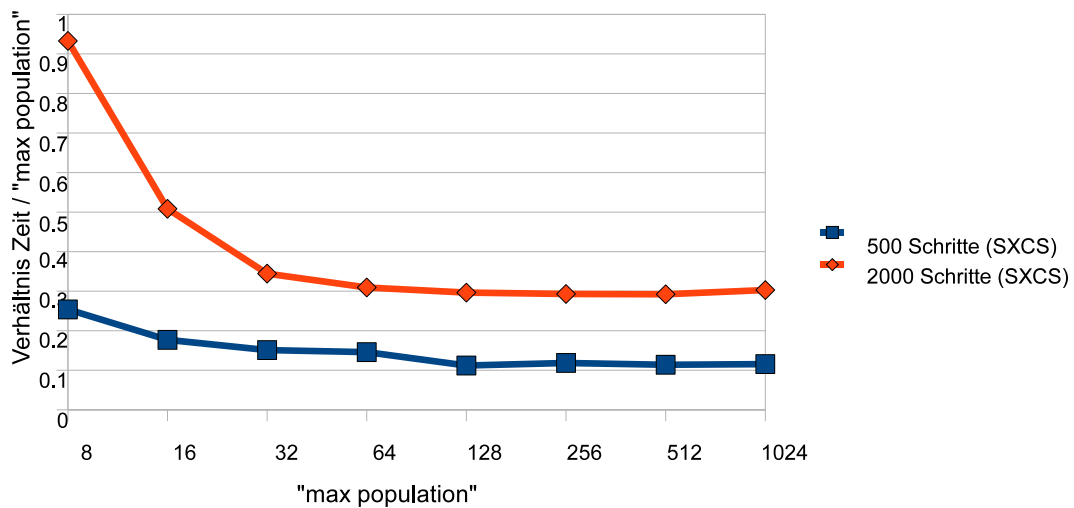


Abbildung 3.6: Darstellung der Auswirkung des Parameters *max population*  $N$  auf das Verhältnis der Laufzeit zu  $N$  im leeren Szenario, zufälliger Bewegung des Zielobjekts, Geschwindigkeit 1, Agenten mit SXCS Algorithmus



einer leeren *classifier set* Liste zu starten.

Dies bestätigen auch Tests, in denen man die Anzahl durch *covering* neu erstellter *classifier* ohne zufällige Initialisierung der *classifier set* Liste (Abbildung 3.7) mit der Anzahl mit Initialisierung (Abbildung 3.2) verglichen hat. Hier 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.

Entsprechend müssen im zweiten Fall mit vorinitialisierter Liste die überflüssigen *classifier* 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.

### 3.6.3 Parameter *reward prediction discount* $\gamma$

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  $\gamma$  bedeutet, dass die Höhe des Werts, der über *maxPrediction* weitergegeben wird, mit zeitlichem Abstand zur ursprünglichen Bewertung mit einem *base reward*, weniger schnell abfällt, wodurch eine längere Verkettung von *base reward* Werten möglich ist. Umgekehrt führen zu hohe Werte für  $\gamma$  zu der positiven Bewertung von *classifier* 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  $\gamma$

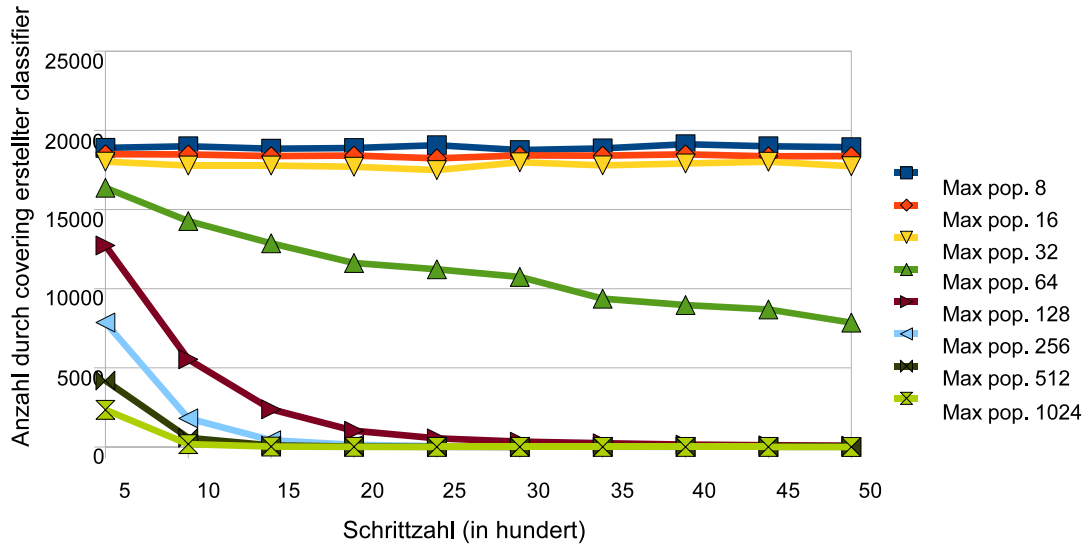


Abbildung 3.7: Auswirkung der maximalen Populationsgröße auf die Anzahl der *classifier*, die durch *covering* neu erstellt werden (Säulenszenario, Zielobjekt mit einfacher Richtungsänderung, Geschwindigkeit 1, Agenten mit SXCS, ohne Initialisierung der *classifier set* Liste)

beim XCS Algorithmus im Säulenszenario. „Intelligent“, „Einfache Richtungsänderung“ und „Geschwindigkeit“ beziehen sich jeweils auf die Eigenschaften des Zielobjekts. Ein konkretes Muster ist nicht zu erkennen und die Fluktuationen sind minimal, weshalb wie vorgeschlagen hier jeweils  $\gamma = 0,71$  verwendet wird. Diese Tatsache ist ein Hinweis darauf, dass die Weitergabe des *base reward* grundsätzlich erfolgen sollte (Abfall der Qualitätsdifferenz auf 0), jedoch sonst keinen Einfluss hat, also wahrscheinlich keine zusammenhängenden Aktions-Ketten gebildet werden. Diese Tatsache unterstreicht auch, dass im Überwachungsszenario das Finden von „Wegen“ (wie beim einführenden Beispiel zum *multi step* Verfahren in Kapitel 1.1.2) nicht zum Erfolg führt. Mit einer Erweiterung der Sensoren, prägnanteren Hindernisstrukturen und Aufenthaltswahrscheinlichkeiten des Zielobjekts auf dem Torus mit hoher Varianz würde dies wahrscheinlich anders aussehen.

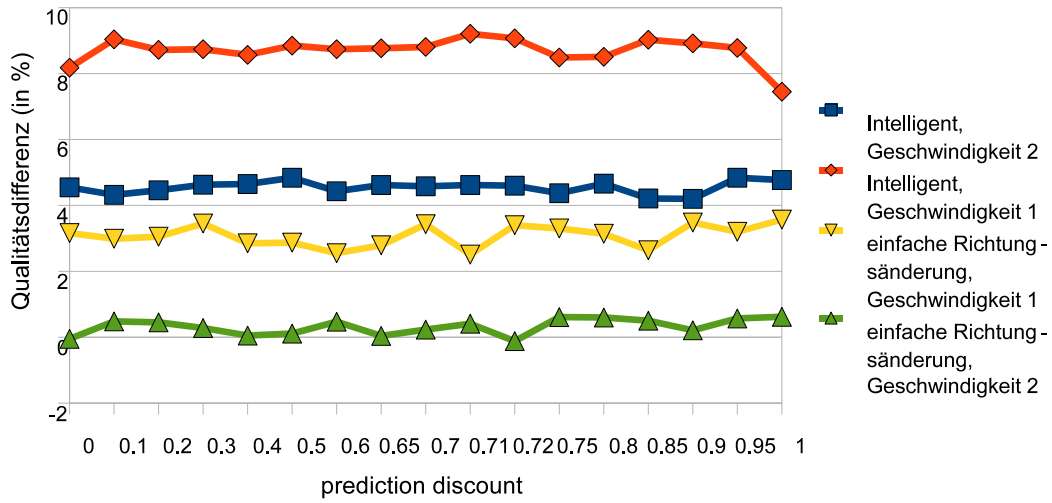


Abbildung 3.8: Auswirkung verschiedener *reward prediction discount*  $\gamma$  Werte auf die Qualität (Säulenszenario, Agenten mit XCS, 2000 Schritte)

### 3.6.4 Parameter Lernrate $\beta$

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

Vergleichende Tests (siehe Abbildung 3.9) lassen ab etwa  $\beta = 0,01$  einen leichten Abwärtstrend bei größeren Werten feststellen, bei kleineren Werten für  $\beta$  gibt es jedoch keine signifikante Unterschiede. Nur im schwierigen Szenario ist ein Unterschied zu bemerken (siehe Abbildung 3.10), dort soll 0,1 als Lernrate  $\beta$  gewählt werden. XCS hat hier Probleme das Gelernte zu behalten, verlernt also, Öffnungen zu finden. SXCS profitiert umgekehrt von einer großen Lernrate. Durch die direkte Verknüpfung der *classifier* wird der Weg durch die Öffnungen beim Erreichen des Ziels belohnt. Weiter Aufschluss gibt die Betrachtung des laufenden Durchschnitts der Qualität des Algorithmus (siehe Abbil-

dung 3.11). SXCS scheint mit  $\beta = 0,1$  einen stabilen Wert erreicht zu haben, während SXCS mit  $\beta = 0,001$  nicht viel besser abschneidet als der Agent mit zufälliger Bewegung. XCS scheint dagegen zu Beginn mit SXCS mit  $\beta = 0,1$  mithalten zu können. Ab etwa der Hälfte fällt es jedoch sogar unter die Qualität des Algorithmus mit zufälliger Bewegung, scheint also konkret etwas falsches gelernt zu haben. TODO, neues Bild mit random

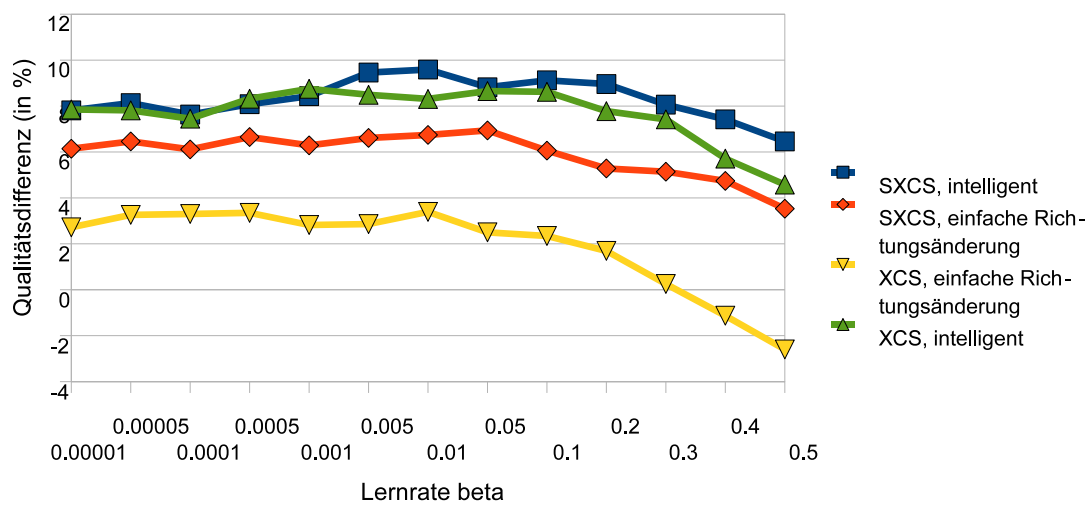


Abbildung 3.9: Auswirkung des Parameters *learning rate*  $\beta$  auf die Qualität im Säulenszenario, intelligente Bewegung des Zielobjekts, Geschwindigkeit 1, Agenten mit SXCS Algorithmus, 2000 Schritte

### 3.6.5 Parameter *accuracy equality* $\epsilon_0$

Der Parameter  $\epsilon_0$  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 ( $\rho$ ) gesetzt werden soll, welcher beliebig wählbar ist und lediglich ästhetische Auswirkungen hat. Somit wird dieser auf 1,0 gesetzt und  $\epsilon_0$  auf 0,01.

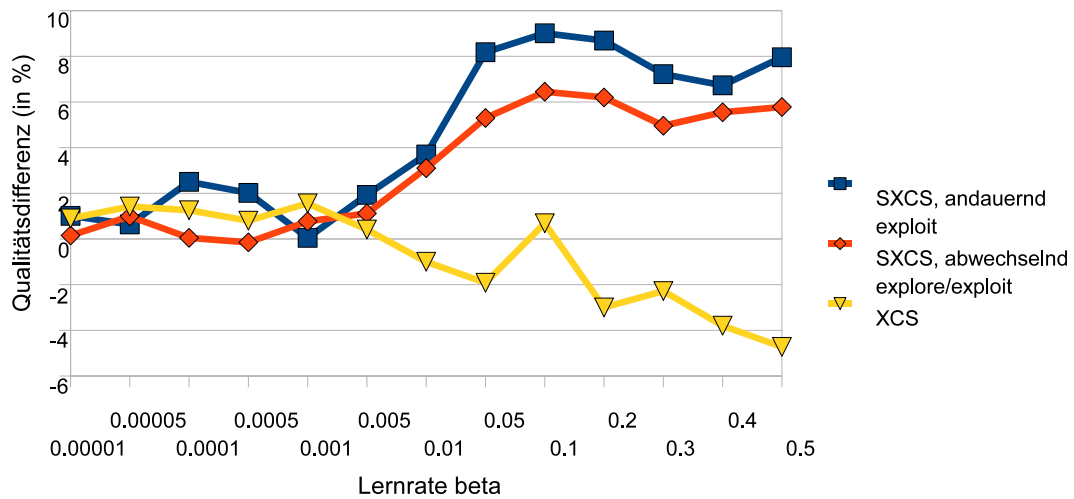


Abbildung 3.10: Auswirkung des Parameters *learning rate*  $\beta$  auf die Qualität im schwierigen Szenario, Bewegung des Zielobjekts ohne Richtungsänderung, Geschwindigkeit 2, Agenten mit SXCS Algorithmus, 2000 Schritte

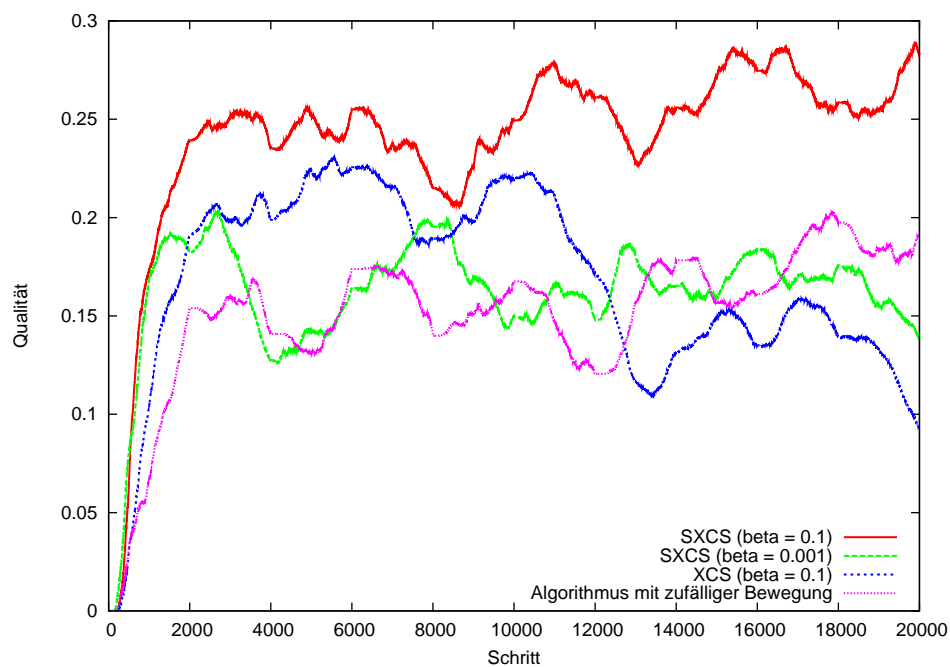


Abbildung 3.11: Zeitlicher Verlauf des gleitenden Durchschnitts der Qualität (schwieriges Szenario, Bewegung des Zielobjekts ohne Richtungsänderung, Geschwindigkeit 2, SXCS und XCS mit Lernrate 0,1, 2000 Schritte)

### 3.6.6 Parameter *tournament factor* $p$

In Abbildung 3.12 und Abbildung 3.13 sind die Ergebnisse im Säulenszenario mit einem Zielobjekt mit einfacher Richtungsänderung, einmal mit Geschwindigkeit 1, das andere Mal mit Geschwindigkeit 2, dargestellt. 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.

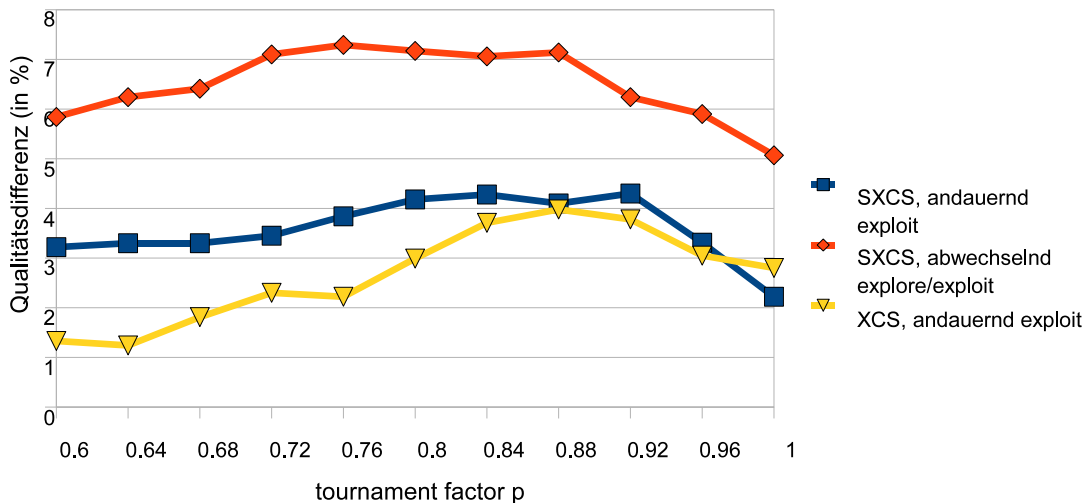


Abbildung 3.12: Vergleich verschiedener Werte  $p$  für Auswahlart *tournament selection* (Zielobjekt mit einfacher Richtungsänderung, Geschwindigkeit 2, Säulenszenario, 2000 Schritte)

Im Falle eines Zielobjekts mit intelligenter Bewegung (Abbildung 3.14 mit Geschwindigkeit 1 und Abbildung 3.15 mit Geschwindigkeit 2) fällt direkt ins Auge, dass eine

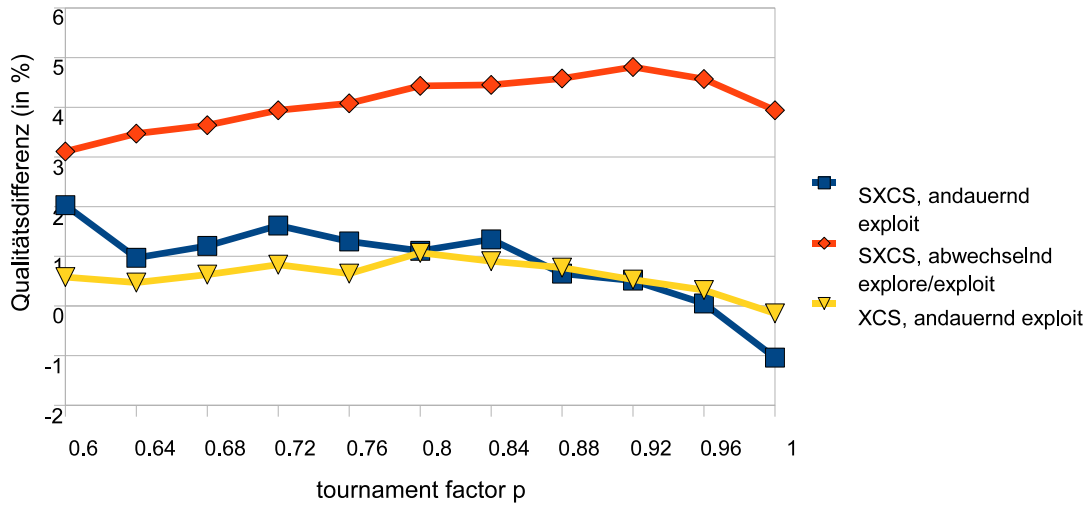


Abbildung 3.13: Vergleich verschiedener Werte  $p$  für Auswahlart *tournament selection* (Zielobjekt mit einfacher Richtungsänderung, Geschwindigkeit 2, Säulenszenario, 2000 Schritte)

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. Dass beim XCS Algorithmus im Fall mit Geschwindigkeit 2 bei einem Wert von  $p = 1.0$ , also der *best selection* Auswahlart, ein etwas besseres Ergebnis erzielt wird, erstaunt. Da der Wert während des Laufs aber nicht angepasst wird, besteht keine Möglichkeit, dies auszunutzen und gleichzeitig bei allen anderen Tests gut abzuschneiden.

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

Außerdem kann man erkennen, dass bei einem sich intelligent verhaltenden Zielobjekt

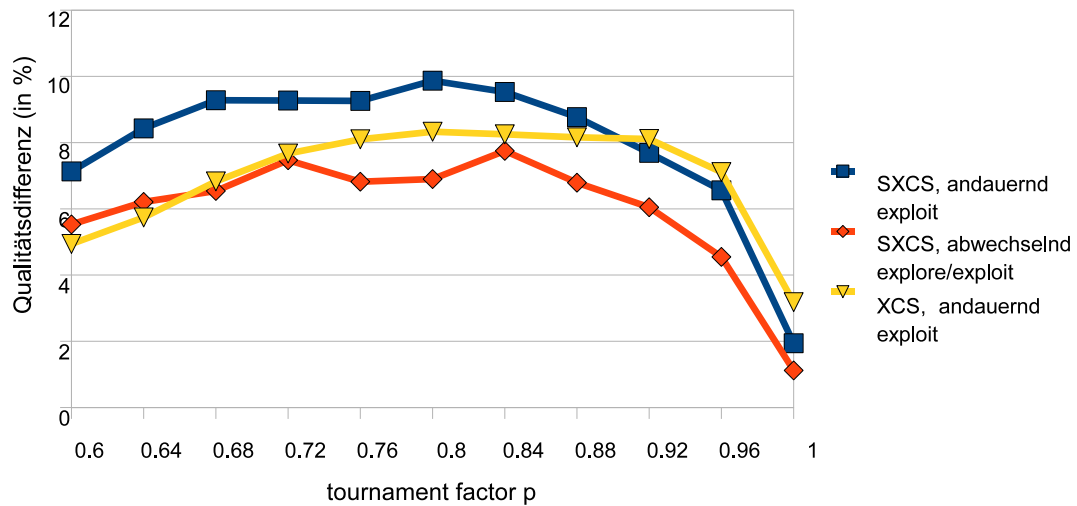


Abbildung 3.14: Vergleich verschiedener Werte  $p$  für Auswahlart *tournament selection* (intelligentes Zielobjekt, Geschwindigkeit 1, Säulenszenario, 2000 Schritte)

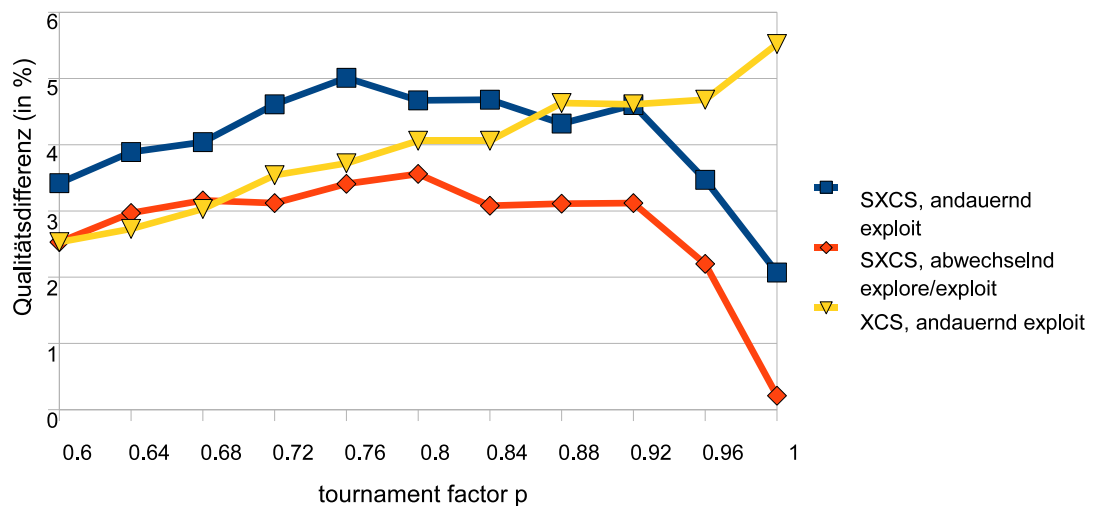


Abbildung 3.15: Vergleich verschiedener Werte  $p$  für Auswahlart *tournament selection* (intelligentes Zielobjekt, Geschwindigkeit 2, Säulenszenario, 2000 Schritte)



eine andauernde *exploit* Phase die beste Wahl ist. Dies wird in Kapitel 5.6 relevant und dort auch näher diskutiert.

Anzumerken ist weiterhin, dass, weil nicht eine andauernde *exploit* Phase mit *best selection* gewählt wurde, die hier benutzten auf XCS basierenden Agenten nicht die Qualität von Heuristiken erreichen können. Kann beispielsweise die einfache Heuristik fast problemlos ein Zielobjekt, was sich mit einer Geschwindigkeit von 1,0 bewegt, unendlich lange verfolgen, bricht ein XCS Algorithmus diese Verfolgung in jedem Schritt mit einer Wahrscheinlichkeit von mindestens  $100\% - 84\% = 16\%$  ab.

### 3.6.7 Übersicht über alle Parameterwerte

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

Parameter	Wert	Standardwert [BW01]
max population $N$	<b>256</b> (siehe Kapitel 3.6.1)	[je nach Szenario]
max value $\rho$	<b>1,0</b> (siehe Kapitel 3.6.5)	[10000]
fraction mean fitness $\delta$	0,1	[0,1]
deletion threshold $\theta_{\text{del}}$	20,0	[ $\sim$ 20,0]
subsumption threshold $\theta_{\text{sub}}$	20,0	[20,0+]
covering # probability $P_{\#}$	0,33	[ $\sim$ 0,33]
GA threshold $\theta_{\text{GA}}$	25	[25-50]
mutation probability $\mu$	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 $\epsilon_i$	0,0	[0,0]
fitness init $F_i$	0,01	[0,01]
condition vector	<b>leer</b> (siehe Kapitel 3.6.2)	[zufällig oder leer]
numerosity	1	[1]
experience	0	[0]
accuracy equality $\epsilon_0$	<b>0,01</b> (siehe Kapitel 3.6.5)	[1% des größten Werts]
accuracy calculation $\alpha$	0,1	[0,1]
accuracy power $\nu$	5,0	[5,0]
reward prediction discount $\gamma$	0,71	[0,71]
learning rate $\beta$	<b>0,001 - 0,01</b> (siehe Kapitel 3.6.4)	[0,1-0,2]
exploration probability	0,5 (siehe Kapitel 3.2.2)	[ $\sim$ 0,5]
tournament factor	0,84 (siehe Kapitel 3.6.6)	[-]

# Kapitel 4

## XCS Varianten

Ziel der Arbeit ist es, herauszufinden, 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 stehen zwei grundsätzliche Fragen:

- zum einen, ob XCS für die Lösung solcher Szenarien besser eignet als ein sich rein zufällig verhaltender Algorithmus und
- zum anderen, wie mögliche Verbesserungsansätze für den Algorithmus aussehen könnten.

Zuerst werden allgemeine Anpassungen des Algorithmus und der Implementation besprochen (siehe Kapitel 4.1) um dann auf die konkreten Veränderungen der einzelnen XCS Varianten einzugehen. Zum einen wird der XCS Algorithmus selbst in Kapitel 4.2 vorgestellt, dort wird insbesondere die Behandlung des Neustarts eines Problems diskutiert. Zum anderen wird eine an Überwachungsszenarios angepasste Variante, der sogenannte

SXCS Algorithmus, vorgestellt. 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.

## 4.1 Allgemeine Anpassungen

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

- Vollständig neu organisierte 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).
- Der genetische Operator wird hier zwei feste, anstatt zufällige Schnittpunkte für das *two point crossover* verwenden (siehe Kapitel 3.3.4).
- 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 3.5.5).
- Mehrere XCS Parameter wurden angepasst (siehe Kapitel 3.6).
- Das Erreichen des Ziels wurde für das Überwachungsszenario neu verfasst, wie auch der Neustart von Probleminstanzen neu geregelt wurde (siehe Kapitel 3.4).
- Die Reihenfolge bei der Bewertung, Entscheidung und 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. Dabei gilt die Annahme, dass durch mehrfache Wiederholung des Lernprozesses sich ein Regelsatz ergibt, mit dem das Ziel mit höherer Wahrscheinlichkeit 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 hier die *action set* Liste gelöscht, das Szenario selbst läuft aber weiter. In der originalen Implementierung in [But00] wird an dieser Stelle im Algorithmus die aktuelle Probleminstanz abgebrochen (in *XCS.java* in der Funktion *doOneMultiStepProblemExploit()* bzw. *doOneMultiStepProblemExplore()*). Liegt kein positiver *base reward* Wert vor, so wird lediglich der für diesen Schritt erwartete *reward* Wert (nämlich der *maxPrediction* Wert) an die letzte *action set* Liste gegeben.

In den Programmen A.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*) wird in Kapitel 4.3.1 zuerst die Umsetzung dieser Idee diskutiert werden. 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:

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

bzw. bei quadratischer Verteilung des *base reward*:

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

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.

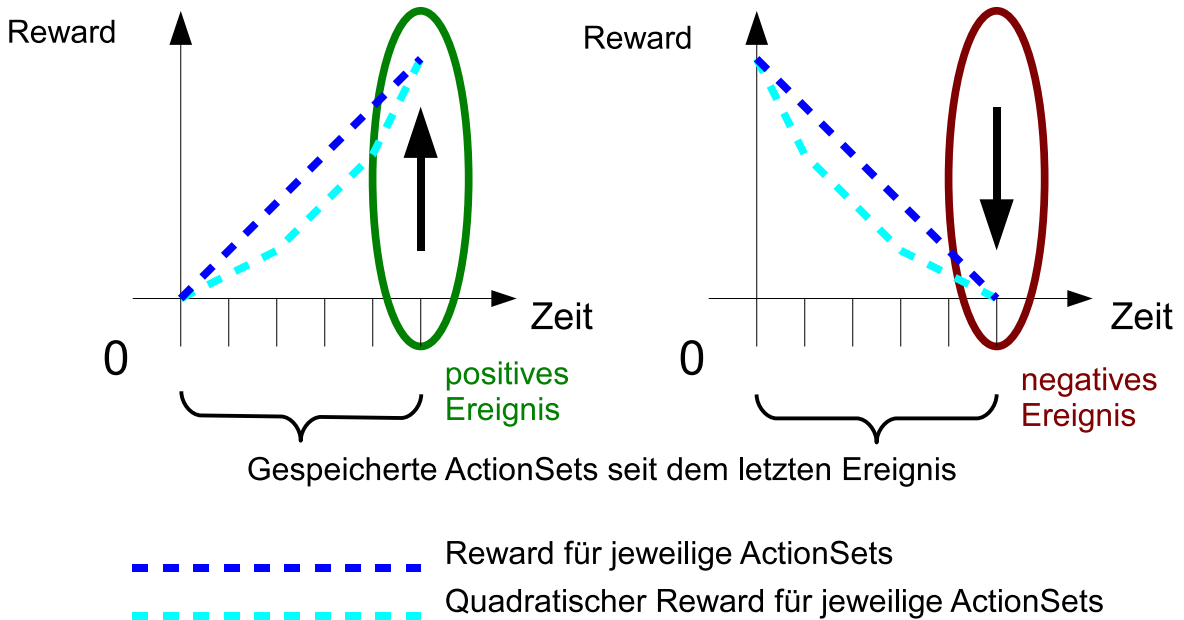


Abbildung 4.1: Schematische Darstellung der (quadratischen) Verteilung des *reward* an gespeicherte *action set* Listen bei einem positiven bzw. negativen Ereignis

### 4.3.2 Ereignisse

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

Von einem positiven bzw. negativen Ereignis spricht man, wenn sich der *base reward* im Vergleich zum vorangegangenen Schritt 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) dann wird abgebrochen und die  $\frac{\text{maxStackSize}}{2}$  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 Änderung des *base reward* Werts auftritt, im dargestellten Fall also der *base reward* sich beim aktuellen Zeitpunkt auf 0 verändern würde.



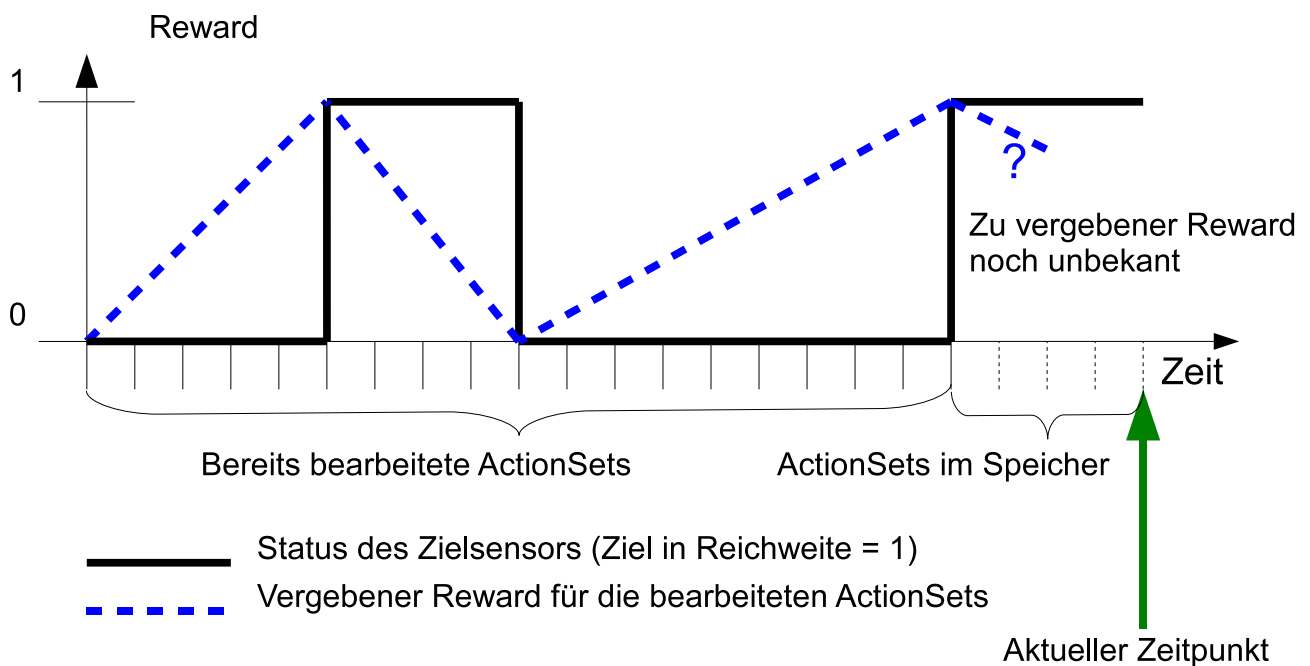


Abbildung 4.2: Schematische Darstellung der zeitlichen Verteilung des *reward* an *action set* Listen nach mehreren positiven und negativen Ereignissen und der Speicherung der letzten *action set* Liste

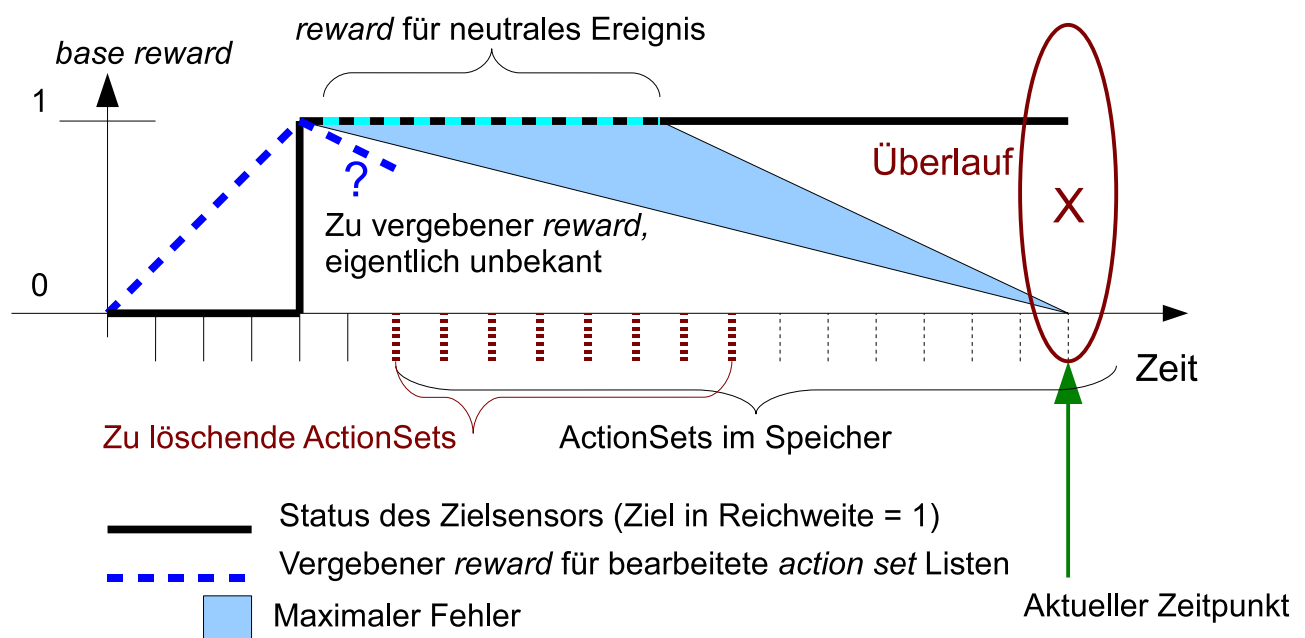


Abbildung 4.3: Schematische Darstellung der Bewertung von *action set* Listen bei einem neutralen Ereignis (mit *base reward* = 1)

### 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  $\frac{\text{maxStackSize}}{2}$  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.

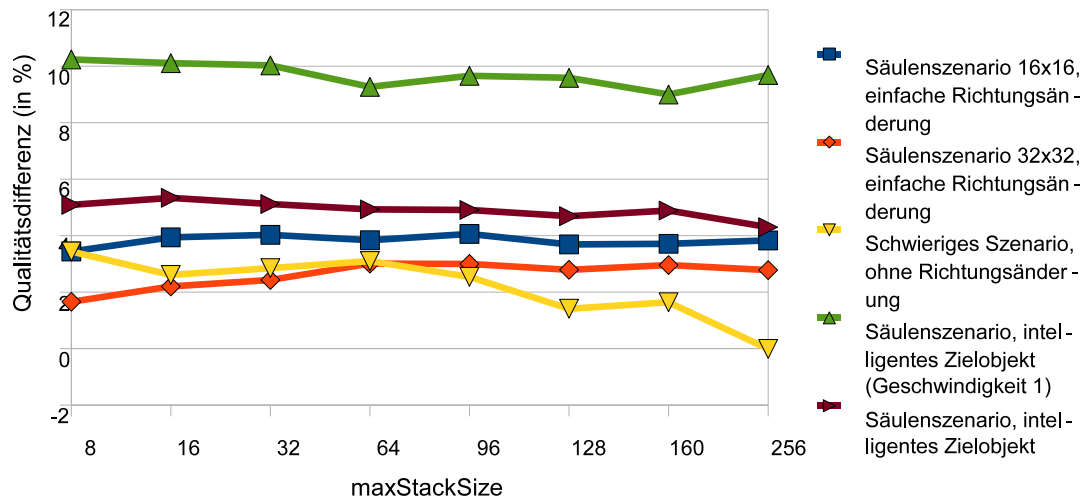


Abbildung 4.4: Vergleich verschiedener Werte für *maxStackSize* (2000 Schritte, SXCS Agenten)

#### 4.3.4 Zusammenfassung der Ereignisse

Ein Ereignis tritt auf, wenn:

- Änderung des *base reward* Werts von 0 auf 1 (Zielobjekt war im letzten Schritt nicht in Überwachungsreichweite)  $\Rightarrow$  positives Ereignis
- Änderung des *base reward* Werts von 1 auf 0 (Zielobjekt war im letzten Schritt in Überwachungsreichweite)  $\Rightarrow$  negatives Ereignis
- Überlauf des Stacks (kein positives oder negatives Ereignis in den letzten *maxStackSize* Schritten), Zielobjekt ist in Überwachungsreichweite  $\Rightarrow$  neutrales Ereignis (mit *base reward* = 1)
- Überlauf des Stacks (kein positives oder negatives Ereignis in den letzten *maxStackSize* Schritten), Zielobjekt ist nicht in Überwachungsreichweite  $\Rightarrow$  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). Als Unterschiede sind festzuhalten:

- In der Funktion *calculateReward()* in Programm A.13 bei der Berechnung des *reward* Werts wird zwischen zwei Fällen unterschieden. 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.
- In der Funktion *collectReward()* in Programm A.14 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).

# Kapitel 5

## Analysen und Experimente

In diesem Kapitel werden die Tests gesammelt dargestellt und analysiert. Ziel des Kapitels ist es, die bisher gemachten Aussagen, die nicht durch die Literatur abgedeckt sind, anhand empirischer Tests zu beweisen. Zu Beginn werden in Kapitel 5.1 anfängliche Tests durchgeführt, bei denen ausschließlich die grundsätzlichen Algorithmen (Agent mit zufälliger Bewegung, Agent mit einfacher Heuristik und Agent mit intelligenter Heuristik) aus Kapitel 2.4 betrachtet werden. Außerdem werden dort die in Kapitel 2.2 vorgestellten Szenarien beleuchtet.

Beides soll als Vorbereitung für eine Analyse der XCS Varianten in Kapitel 5 dienen, damit gezielt die erfolgversprechenden Konfigurationen getestet werden können. Anzumerken ist, dass mit „XCS“ die XCS Variante aus Kapitel 4.2 mit den allgemeinen Anpassungen aus Kapitel 4.1 und den Parametereinstellungen aus Kapitel 3.6 gemeint ist. Vereinzelt wird in diesem Kapitel auch von „Qualitätsdifferenz“ die Rede sein. Hierbei ist die Differenz zur Qualität des Algorithmus mit zufälliger Bewegung gemeint. Zweck der Benutzung der Differenz ist, leichter die Lerneffekte zwischen verschiedenen Szenarien darzustellen.

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

### 5.1.1 Zielobjekt mit zufälligem Sprung (leeres Szenario)

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

Betrachtet man das Szenario ohne Hindernisse gibt sich ein klares Bild (siehe Tabelle 5.1), die intelligente Heuristik ist etwas besser als der des zufälligen Agenten und der

einfachen Heuristik. Ein möglichst weiträumiges Verteilen auf dem Torus führt zum Erfolg. Dies zeigt sich auch in einem hohen Wert der Abdeckung, denn genau das wird mit dem völlig zufällig springenden Agenten getestet. Ebenfalls ist die Zahl der blockierten Bewegungen deutlich niedriger, was sich auch mit der Haltung des Abstands erklären lässt.

Die einfache Heuristik schneidet dagegen etwas schlechter als eine zufällige Bewegung ab. Zwar ist die Zahl der blockierten Bewegungen geringer, was sich dadurch erklären lässt, dass die einfache Heuristik zumindest an einem Punkt eine Sichtbarkeitsüberprüfung für die Richtung durchführt, in der sie sich bewegen möchte (nämlich wenn das Zielobjekt in Sicht ist), andererseits ist die Abdeckung etwas geringer.

Ursache dafür ist wahrscheinlich, dass wenn mehrere Agenten das Zielobjekt in Sichtweite haben, alle sich auf das Zielobjekt bewegen. Dadurch wird die zufällige Verteilung der Agenten auf dem Spielfeld gestört, was letztlich zu einer niedrigeren Abdeckung des Torus führt.

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

### 5.1.2 Zielobjekt mit zufälligem Sprung (Säulenszenario)

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

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%

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.

Tabelle 5.2: Zufällige Sprünge des Zielobjekts in einem Säulenszenario

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

### 5.1.3 Zielobjekt mit zufälligem Sprung (Zufällig verteilte Hindernisse)

Auch hier gelten wieder die selben Überlegungen für das Zielobjekt, die in Kapitel 5.1.1 gemacht wurden. Und auch für alle Einstellungen von  $\lambda_h$  und  $\lambda_p$  (siehe Kapitel 2.2.2)



stellt sich ebenfalls ein eindeutiges Bild dar (siehe Tabelle 5.3). So liegt die intelligente Heuristik wieder vorne, gefolgt wieder von der einfachen Heuristik und der zufälligen Bewegung. Im Fall mit vielen Hindernissen ( $\lambda_h = 0,2$ ) liegt die einfache Heuristik trotz höherer Abdeckung hinter der zufälligen Bewegung. Dies ist wohl auf einen Zufall zurückzuführen, ändert man den *random seed* Wert oder erhöht man die Anzahl der Experimente von 10 auf 30 ergibt sich wieder die oben genannte Reihenfolge.

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

Im Vergleich zur einfachen Heuristik und im Gegensatz zum Säulenszenario scheint insbesondere die intelligente Heuristik Probleme mit den Hindernissen zu haben (viele blockierte Bewegungen). Da Hindernisse in der Heuristik nicht beachtet werden, bewirkt die Strategie der maximalen Ausbreitung der Agenten wahrscheinlich ein „Drücken“ dieser Agenten gegen die Hindernisse, da sich von anderen Agenten wegbewegt wird anstatt zufällig zu laufen.

Schließlich ist zu sehen, dass die Agenten in einem Szenario mit höherem Verknüpfungsfaktor (der Fall mit  $\lambda_h = 0,1$  und  $\lambda_p = 0,99$  im Vergleich zum Fall mit  $\lambda_h = 0,1$  und  $\lambda_p = 0,5$ ) besser abschneiden. Dies liegt daran, dass Szenarien mit hohem Verknüpfungs-

faktor bedeuten, dass viele Hindernisse zusammenhängend einen großen Block bilden und somit dem Szenario ohne Hindernisse ähnlich sind, da es eher größere zusammenhängende Flächen gibt.

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

Algorithmus	$\lambda_h$	$\lambda_p$	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%

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

Da die einfache Heuristik sich gegenüber den sich zufällig bewegendenden Agenten nicht durchsetzen konnte und in Kapitel 3.4 entschieden wurde, nicht den intelligenten Agenten in der Bewertungsfunktion nachzumodellieren, ist es offensichtlich sinnlos, diese Form der Bewegung des Zielobjekts in Verbindung mit XCS zu betrachten. Es soll also eine Verbindung zwischen der Position des Zielobjekts in diesem Schritt mit der Position im letzten Schritt geben, was zum nächsten Kapitel führt.

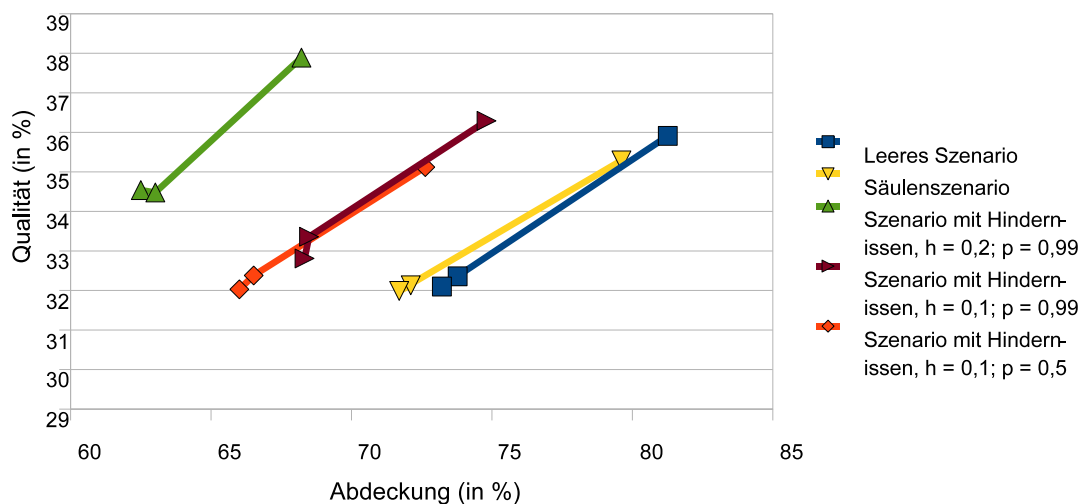


Abbildung 5.1: Zusammenhang zwischen der Abdeckung und der Qualität eines Algorithmus, getestet in verschiedenen Szenarien

#### 5.1.4 Zielobjekt mit zufälliger Bewegung bzw. einfacher Richtungsänderung

Gemeinsam ist beiden Bewegungstypen (siehe Kapitel 2.5.2 und Kapitel 2.5.3), 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 Schritt befinden wird, ist zumindest vom aktuellen Ort abhängig, wenn das Zielobjekt auch schneller sein kann als andere Agenten.

Hingegen unterscheiden sie sich eindeutig im Bewegungsmuster des Zielobjekts. Dieses kehrt mit zufälliger Bewegung nach 2 Schritten mit Wahrscheinlichkeit von  $\frac{1}{4}$  auf das ursprüngliche Feld zurück, bleibt also stehen. Außerdem bezieht der Bewegungstyp mit einfacher Richtungsänderung Hindernisse in die Entscheidung über die nächste Aktion mit ein. Dies führt ebenfalls zu einer deutlich geringeren Anzahl von blockierten Bewegungen. Wie die Anzahl der Sprünge des Zielagenten in Tabellen 5.5 und 5.6 zeigen, ist

es den Agenten beim Zielobjekt mit zufälliger Bewegung deutlich öfters gelungen, ihn in seiner Bewegung zu blockieren. Wie auch an der Qualität abzulesen ergibt sich dadurch ein deutlich leichteres Szenario für beide Heuristiken, während es kaum Unterschiede in der Qualität bei der zufälligen Bewegung der Agenten ergibt.

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%

## 5.2 Auswirkung der Geschwindigkeit des Zielobjekts

Lässt keine der beiden Parteien, Agenten und das Zielobjekt, Sensordaten über jeweils die andere Partei in die Entscheidung über die nächste Aktion mit einfließen, so spielt das Verhältnis der Geschwindigkeiten beider Parteien langfristig keine Rolle (sofern beide eine Geschwindigkeit größer 0 besitzen). Dies hat man im letzten Kapitel 5.1.4 daran gesehen, dass bei Agenten mit zufälliger Bewegung sich beim Vergleich zwischen beider

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%

Bewegungstypen kaum Unterschiede in der Qualität auftreten, während dies bei der einfachen und intelligenten Heuristik der Fall war. Im Folgenden werden nun also die Fälle untersucht, bei der mindestens einer der Parteien die andere Partei mit in die Überlegung miteinbezieht.

### 5.2.1 Zielobjekt mit einfacher Richtungsänderung

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

Korrelation zwischen der Geschwindigkeit und der Qualität des Algorithmus mit zufälliger Bewegung festzustellen, nur bei Geschwindigkeit 0 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 5.3 sind dagegen die Testergebnisse (im selben Szenario) für die einfache und die intelligente Heuristik zu sehen. Im Wesentlichen sind drei Punkte anzumerken:

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

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

### 5.2.2 Zielobjekt mit intelligenter Bewegung

In Abbildung 5.4 und Abbildung 5.5 werden im Säulenszenario bzw. Szenario mit zufällig verteilten Hindernissen wieder die Heuristiken bei unterschiedlichen Geschwindigkeiten des Zielobjekts verglichen. Beim Säulenszenario ist wieder der Knick wie beim Fall mit

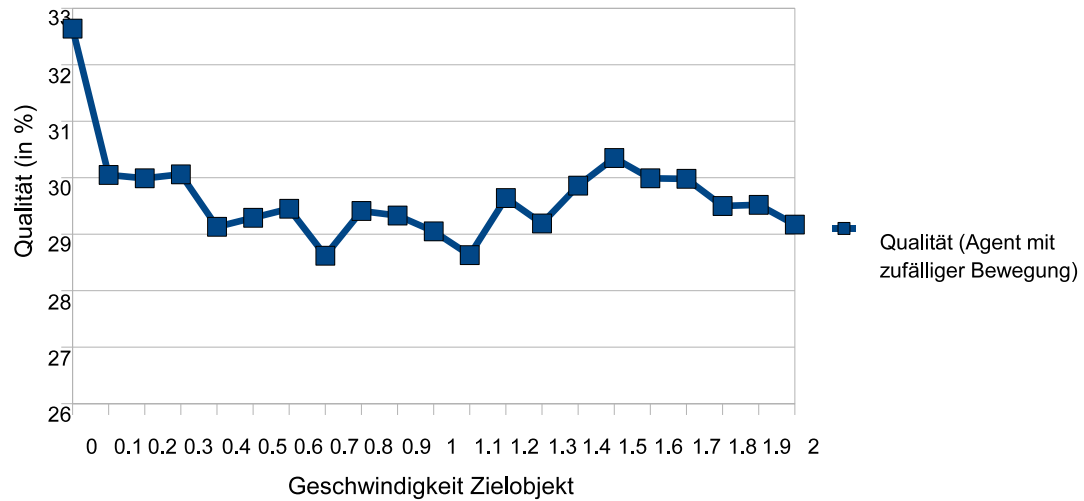


Abbildung 5.2: Auswirkung der Zielgeschwindigkeit auf Agenten mit zufälliger Bewegung (Zielobjekt mit einfacher Richtungsänderung, Säulenszenario)

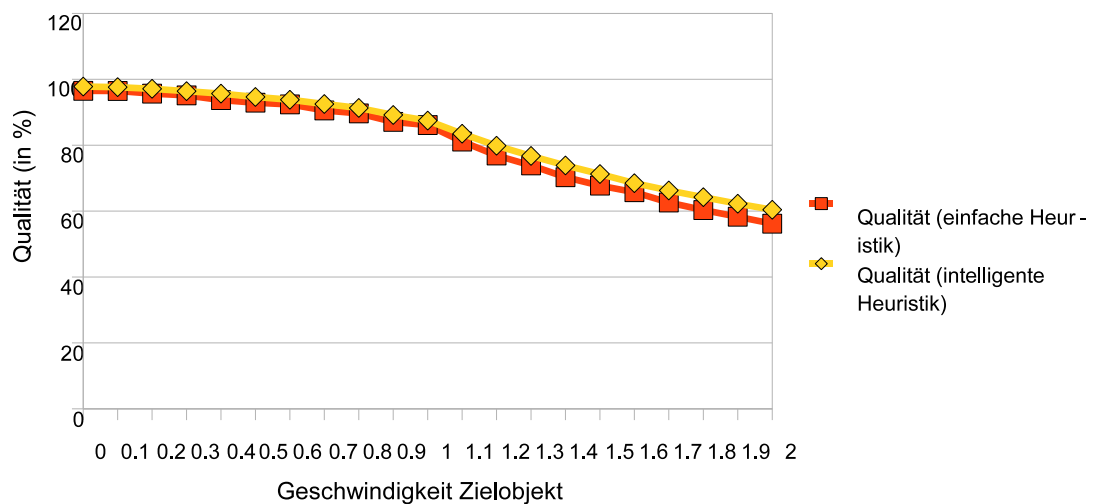


Abbildung 5.3: Auswirkung der Zielgeschwindigkeit auf Agenten mit bestimmten Heuristiken (Zielobjekt mit einfacher Richtungsänderung, Säulenszenario)

Zielobjekt mit einfacher Richtungsänderung (siehe Kapitel 5.2.1) zu beobachten.

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

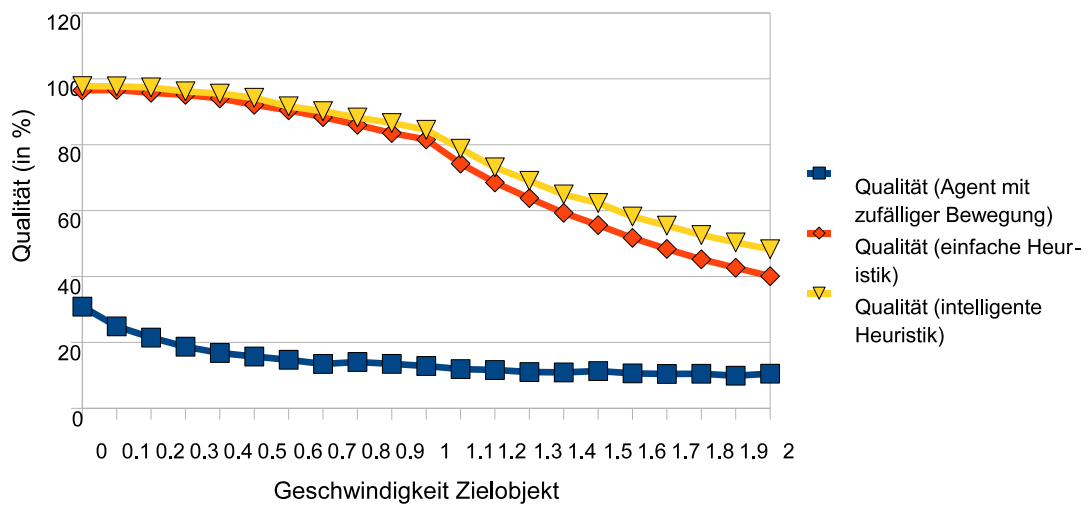


Abbildung 5.4: Auswirkung der Zielgeschwindigkeit (intelligentes Zielobjekt) auf Agenten mit Heuristik

### 5.3 Szenario mit zufällig verteilten Hindernissen

TODO!

### 5.4 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 des Torus vorzudringen und dem Ziel-



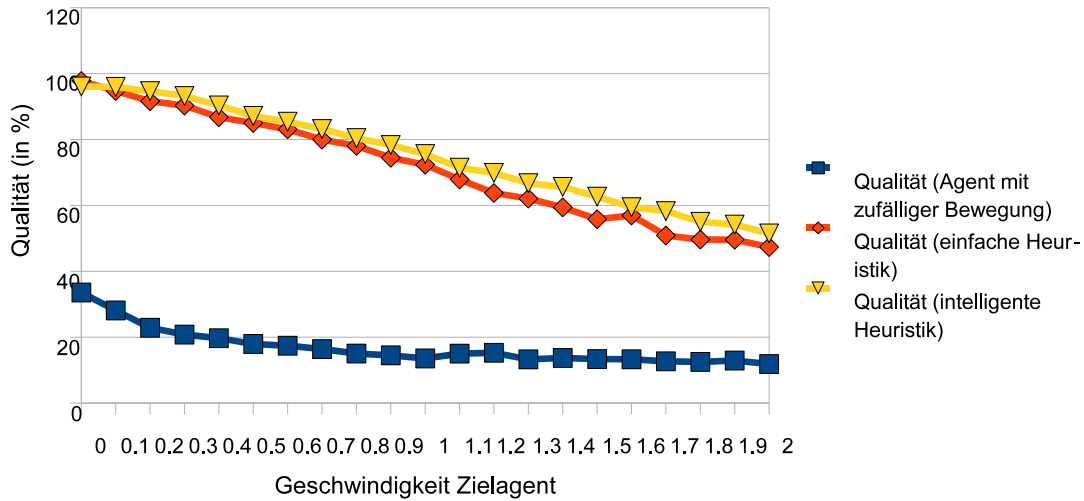


Abbildung 5.5: Auswirkung der Zielgeschwindigkeit (intelligentes Zielobjekt, Szenario mit zufällig verteilten Hindernissen,  $\lambda_h = 0,2$ ,  $\lambda_p = 0,99$ ) auf Agenten mit Heuristik

objekt nicht schon auf halbem Weg zu begegnen.

Für verschiedene Anzahl von Schritten sind für die drei Agententypen in Abbildung 5.6 die jeweiligen Qualitäten aufgeführt. Wie man beim Vergleich zwischen zufälliger Bewegung und einfacher Heuristik sehen kann, ist es nicht nur entscheidend, in den letzten Bereich am rechten Rand des Szenarios vorzudringen, sondern auch, dort den Agenten zu verfolgen und in diesem Bereich zu bleiben. Deutlich zeigen sich hier die Vorzüge der intelligenten Heuristik, durch das Bestreben, Agenten auszuweichen, hat es dieser Algorithmus leichter, durch die Öffnungen in von Agenten unbesetzte Bereiche vorzudringen. Der Unterschied zwischen einfacher und intelligenter Heuristik zeigt auch, dass in diesem Szenario ein deutlich größeres Lernpotential, was die Einbeziehung von wahrgenommenen Agentenpositionen betrifft, für Agenten besteht. Wie später in Kapitel 5.9.1 gezeigt wird, können in diesem Szenario unter anderem deshalb auf XCS basierte Agenten ihre Vorteile besonders gut ausspielen und erreichen sogar bessere Ergebnisse als die intelligente Heuristik.

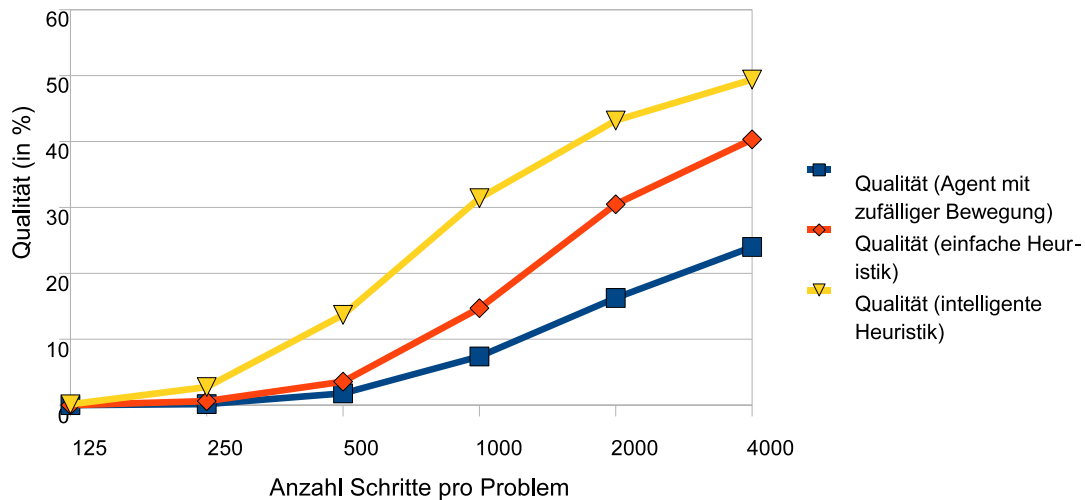


Abbildung 5.6: Auswirkung der Anzahl der Schritte (schwieriges Szenario, Geschwindigkeit 2, ohne Richtungsänderung) auf Qualität von Agenten mit Heuristik

## 5.5 Zusammenfassung der Tests mit Heuristiken

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

Bei der Analyse der Geschwindigkeit des Zielobjekts in Kapitel 5.2 ergab sich bei einer Geschwindigkeit von 1 ein Knick, ab dem kollaboratives Verhalten gegenüber sturem Verfolgen an Bedeutung gewann. Da die Heuristiken das obere Limit angeben und so gebaut sind, dass sie sich immer für die jeweilige in ihren Augen beste Aktion entscheiden, wohingegen die auf XCS basierenden Varianten dies in bestenfalls 84% der Fälle tun (siehe Kapitel 3.6.6). Es ist deshalb anzunehmen, dass in Verbindung mit XCS auch niedrigere Geschwindigkeiten betrachtet werden können, ohne Agenten zu erhalten, die ausschließlich auf Verfolgung aus sind.

Beim Test des schwierigen Szenarios in Kapitel 5.4 wurde zum einen festgestellt, dass ein einfaches Verfolgen des Zielobjekts nicht zum Ziel führen kann. Desweiteren wurde beobachtet, dass, obwohl Hindernisse nicht beachtet wurden, Agenten mit intelligenter Heuristik Erfolge zeigen, da sie sich gegenseitig auf frei Felder (durch die Öffnungen hindurch) drängen und so schneller zum Ziel finden können. Das Szenario bietet also viele Möglichkeiten des Lernens, wenn es auch, aufgrund recht spezieller Bewegung des Zielobjekts in einem kleinen Bereich, eher den Standardszenarien von XCS ähnelt.

## 5.6 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 beim zufälligem Wechsel, dass die statistischen Werte zwischen denen der *roulette wheel* und *tournament selection* Auswahlart liegen. Angesichts der minimalen Steigerung gegenüber der Qualität der *roulette wheel* Auswahlart, stellt diese Form der Auswahlart also keine signifikante Verbesserung dar.

Wechselt man bei einer Änderung des *base reward* Werts und startet in der *explore* Phase, dann ergibt sich dagegen 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. 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.

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%
<i>roulette wheel selection</i>	4,54%	72,10%	30,30%
<i>random selection</i>	4,34%	72,21%	28,50%
<b><i>tournament selection</i></b>	11,21%	70,20%	<b>33,39%</b>
<i>best selection</i>	41,16%	63,64%	29,22%
Zufällig <i>explore/exploit</i>	6,29%	71,18%	30,58%
Abwechselnd, zuerst <i>explore</i>	5,63%	71,37%	26,30%
<b>Abwechselnd, zuerst <i>exploit</i></b>	9,28%	70,40%	<b>35,36%</b>

Desweiteren ist zum Vergleich wichtig, wie die Situation beim XCS Algorithmus hinsichtlich der Auswahlverfahren ist. Die vergleichende Ergebnisse in Abbildung 5.7 demonstrieren gleich mehrere Punkte:

- XCS hat deutliche Probleme mit einem Zielobjekt mit einfacher Richtungsänderung bei höheren Geschwindigkeiten
- *random selection* führt wie erwartet dazu, dass sich der jeweilige Agent keine bessere Qualität als ein sich zufällig bewegendes Agent erreicht.
- *roulette wheel selection* führt ebenfalls zu keinem besseren Ergebnis.
- Andauernde *exploit* Phase erreicht beim XCS Agenten für *tournament selection* und *best selection* etwa die gleichen Werte.
- *best selection* erreicht beim SXCS Agenten deutlich schlechtere Werte.

- Mit abwechselnder *explore/exploit* Phase (beginnend mit *explore*) werden keine besonders guten Ergebnisse erreicht, im Fall mit Zielobjekt mit einfacher Richtungsänderung ist diese Auswahl sogar deutlich schlechter als der Agent mit zufälliger Bewegung.
- Mit abwechselnder *explore/exploit* Phase (beginnend mit *exploit*) werden beim SXCS Agenten im Schnitt die besten Ergebnisse erzielt.

höhere Werte für exploit bei intelligent und Geschwindigkeit 1

keine besonders guten Ergebnisse erreicht, im Fall mit Zielobjekt mit einfacher Richtungsänderung ist diese Auswahl sogar deutlich schlechter als der Agent mit zufälliger Bewegung.

, dass XCS von keiner Auswahlart besonders profitiert, nur eine andauernde *exploit* Phase mit *tournament selection* erreicht einen positiven Effekt relativ zum Agenten mit zufälliger Bewegung.

TODO hohe Hindernisszahl?

Insgesamt ergeben sich also folgende Ergebnisse:

- Für die *exploit* Phase wird die Auswahlart *tournament selection* benutzt,
- Für die *explore* Phase wird die Auswahlart *roulette selection* benutzt,
- XCS benutzt eine andauernde *exploit* Phase,
- SXCS benutzt abwechselnd eine *explore* und eine *exploit* Phase und
- SXCS benutzt ebenfalls eine andauernde *exploit* Phase bei einem intelligenten Zielobjekt.

Im Folgenden wird der Zusatz „(*exploit*)“ für die Auswahlart mit andauernder *exploit* Phase und der Zusatz „(*explore/exploit*)“ für die Auswahlart mit abwechselnder *explore* und *exploit* Phase (Beginn mit der *exploit* Phase) stehen.

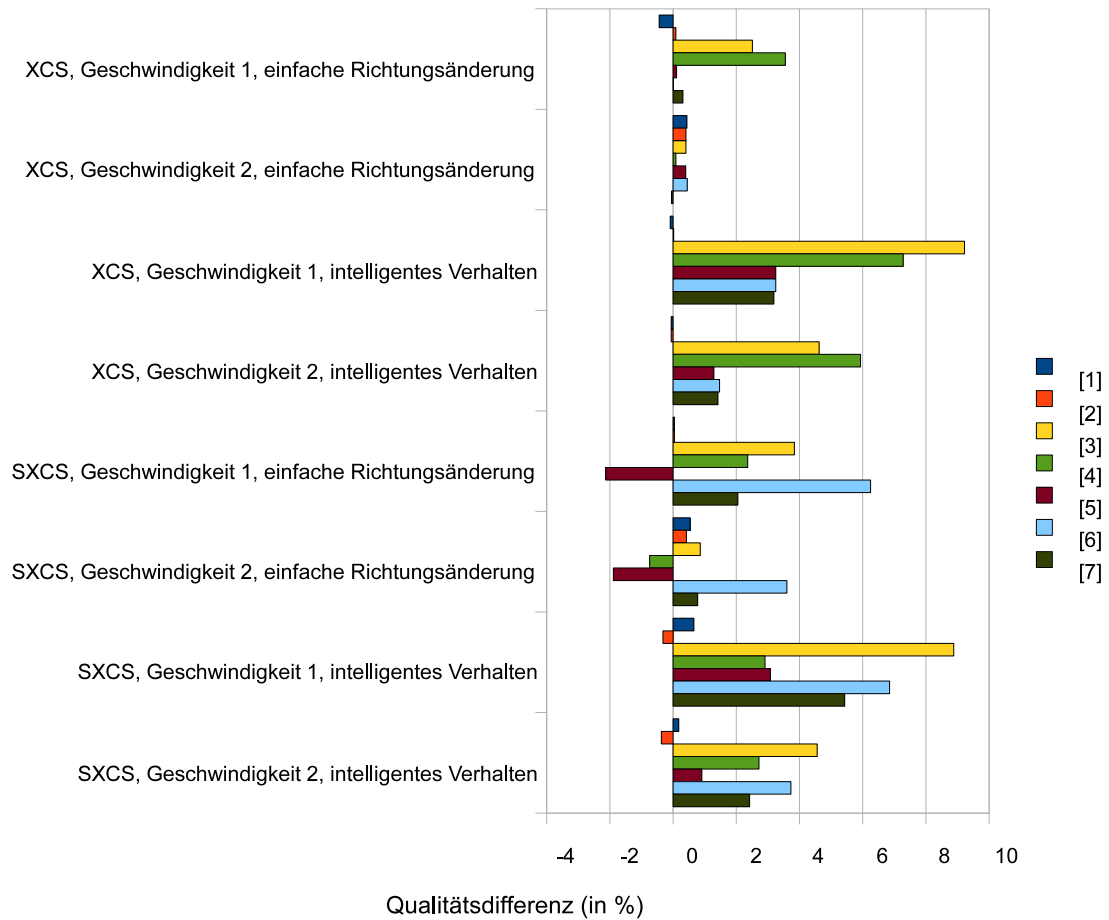


Abbildung 5.7: Vergleich der Qualitäten verschiedener Auswahlarten für **XCS**, von links nach rechts [1] *roulette wheel selection*, [2] *random selection*, [3] *tournament selection*, [4] *best selection*, [5] abwechselnd, zuerst *explore*, [6] abwechselnd, zuerst *exploit* und [7] zufällig *explore/exploit*

## 5.7 Unterschiedliche Geschwindigkeiten des Zielobjekts

In Abbildung 5.8 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% (Geschwindigkeit 2,0), im Fall mit 2000 Schritten erhöht sich dieser Bereich leicht auf 40,15% bis 37,71%.

TODO Kapitel 5.2

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.

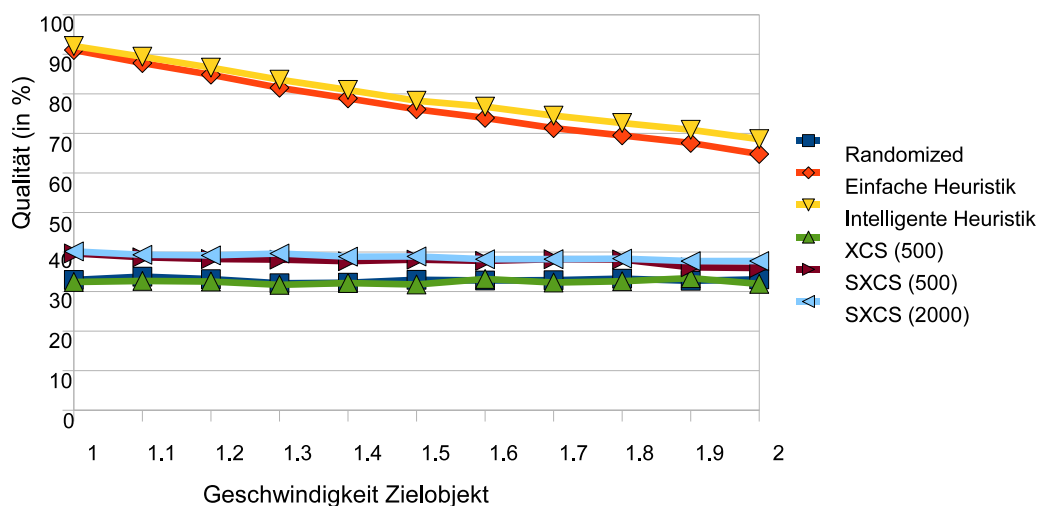


Abbildung 5.8: Vergleich der Qualitäten verschiedener Algorithmen bezüglich der Geschwindigkeit des Zielobjekts



## 5.8 Einschränkung der Ergebnisse für Szenario mit zufällig verteilte Hindernisse

Nicht schaffbare Szenarien, d.h. Szenarien in denen die Qualität des Algorithmus die des Algorithmus mit zufälliger Bewegung nicht übersteigt, gibt es mehrere. Insbesondere das zufällige Szenario scheint den XCS Algorithmen Schwierigkeiten zu bereiten (siehe Tabelle 5.8). Angesichts der (im Vergleich zum zufälligen Algorithmus) sehr hohen Anzahl an blockierten Bewegungen (bei allen Szenarien mit  $\lambda_h > 0$ ) kann man davon ausgehen, dass sie Probleme haben, sich in einer Umgebung mit einem so hohen Hindernissanteil zurechtzufinden, insbesondere auch weil bei einem Problemneustart ein neues zufälliges Szenario generiert wird.

Neben einer Erweiterung der Sensorfähigkeiten und einer Anpassung der *reward* Funktion schafft hier Abhilfe nur ein Wechsel der Auswahlart zur *roulette wheel selection* bzw. der *tournament selection* mit niedrigerem Wert für  $p$ . Eine Erhöhung der maximalen Populationsgröße erbrachte dagegen keine Besserung bzw. die Zahl der neuerstellten *classifier* ging auch mit dem Wert 256 als Populationsgröße gegen Null (dieser Zusammenhang wird in Kapitel 3.6.1 näher diskutiert).

Ohne nochmal auf alle möglichen Szenarien mit zufälliger Verteilung von Hindernissen einzugehen, soll hier TODO exemplarisch zum einen ein Vergleich mit *roulette wheel selection* dargestellt sein, wie auch eine Gegenüberstellung von verschiedenen Werten für  $p$  und  $\lambda_h$ , jeweils für XCS und SXCS.

TODO roulette wheel selection + random TODO p, lambda XCS TODO p, lambda SXCS

## 5.9 Vergleich XCS und SXCS

Tabelle 5.8 Tabelle 5.9

TODO neu, cover actions prüfen!

Tabelle 5.8: Verschiedene Szenarien mit intelligentem Zielobjekt, Geschwindigkeit 1, 2000 Schritten, andauernder *exploit* Phase

Algorithmus	Blockierte	Abdeckung	Qualitätsdifferenz
$\lambda_h = 0,2$ und $\lambda_p = 0,99$			
XCS	56,73%	46,07%	9,97%
SXCS	65,68%	44,34%	10,05%
$\lambda_h = 0,1$ und $\lambda_p = 0,99$			
XCS	38,69%	56,86%	14,25%
SXCS	47,74%	54,21%	13,50%
$\lambda_h = 0,0$			
XCS	5,52%	72,32%	24,06%
SXCS	5,86%	72,47%	24,22%
$\lambda_h = 0,1$ und $\lambda_p = 0,5$			
XCS	41,93%	56,98%	13,59%
SXCS	49,43%	54,92%	14,09%
$\lambda_h = 0,2$ und $\lambda_p = 0,5$			
XCS	63,48%	44,27%	8,57%
SXCS	72,23%	42,42%	8,80%

### 5.9.1 XCS im schwierigen Szenario

Im schwierigen Szenario wurde in Kapitel 5.4 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 gelern-

Tabelle 5.9: Verschiedene Szenarien mit Zielobjekt mit einfacher Richtungsänderung, Geschwindigkeit 1 und 2000 Schritten

Algorithmus	Blockiert	Abdeckung	Qualität
$\lambda_h = 0,2$ und $\lambda_p = 0,99$			
XCS	56,19%	46,50%	23,63%
SXCS <i>exploit</i>	65,86%	43,80%	22,97%
SXCS <i>explore/exploit</i>	54,95%	47,16%	27,20%
$\lambda_h = 0,1$ und $\lambda_p = 0,99$			
XCS	37,20%	57,26%	27,33%
SXCS <i>exploit</i>	47,56%	53,95%	26,13%
SXCS <i>explore/exploit</i>	36,35%	57,76%	31,45%
$\lambda_h = 0,0$			
XCS	5,78%	72,09%	34,42%
SXCS <i>exploit</i>	6,12%	72,28%	34,44%
SXCS <i>explore/exploit</i>	4,82%	72,22%	36,95%
$\lambda_h = 0,1$ und $\lambda_p = 0,5$			
XCS	41,40%	56,89%	27,20%
SXCS <i>exploit</i>	48,95%	54,89%	27,34%
SXCS <i>explore/exploit</i>	38,22%	57,95%	31,29%
$\lambda_h = 0,2$ und $\lambda_p = 0,5$			
XCS	63,80%	44,10%	21,70%
SXCS <i>exploit</i>	73,08%	41,83%	20,75%
SXCS <i>explore/exploit</i>	62,69%	44,18%	25,09%

tes 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.

TODO Abbildung 5.10

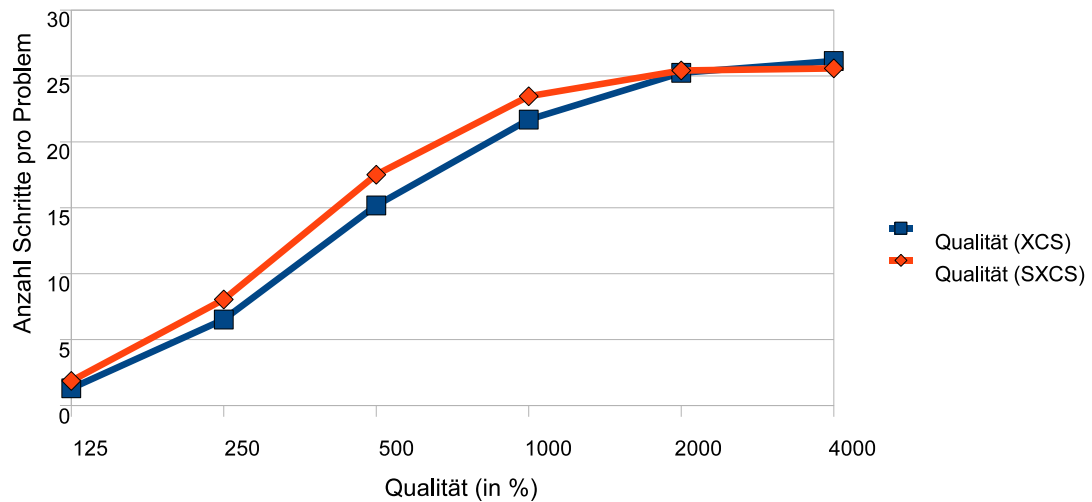


Abbildung 5.9: Auswirkung der Anzahl der Schritte (schwieriges Szenario, Geschwindigkeit 2, ohne Richtungsänderung) auf Qualität von Agenten mit XCS und SXCS

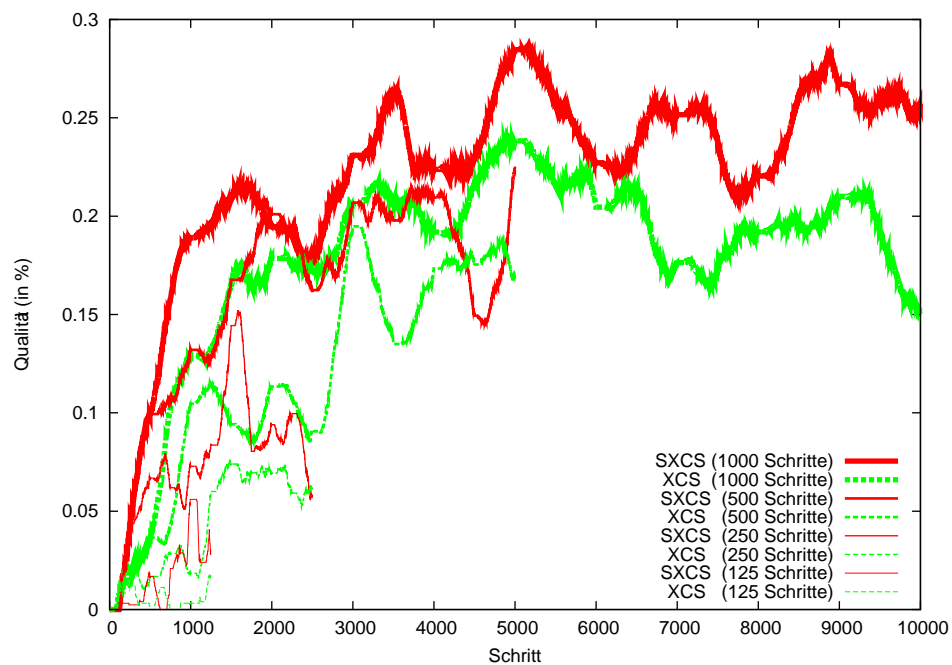


Abbildung 5.10: Auswirkung der Anzahl der Schritte (schwieriges Szenario, Geschwindigkeit 2, ohne Richtungsänderung) auf Qualität von Agenten mit XCS und SXCS

In Abbildung 5.11 ist ein Testlauf abgebildet, bei dem insgesamt jeweils genau 8000 Schritte durchgeführt werden, aber diese Schrittzahl auf eine unterschiedliche Anzahl von Problemen verteilt wird. D.h. es wird ein Problem mit 8000 Schritten, 2 Probleme mit jeweils 4000 Schritten, 4 Probleme mit jeweils 2000 Schritten usw. getestet. Hier sieht man, dass ab 16 Problemen mit jeweils 500 Schritten der SXCS Algorithmus besser abschneidet als der intelligente Algorithmus. SXCS kann die geringere Schrittzahl durch eine erhöhte Problemzahl also durch Erlernen eines Weges durch das Szenario kompensieren.

TODO Abbildung 5.12

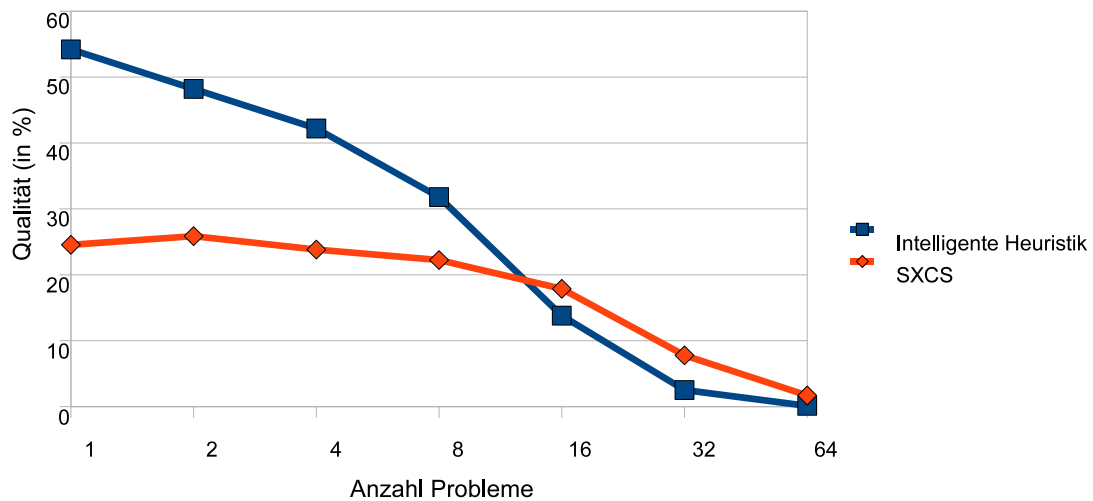


Abbildung 5.11: Qualität bei unterschiedlicher Anzahl von Problemen bei gleichbleibender Gesamtzeit (schwieriges Szenario)

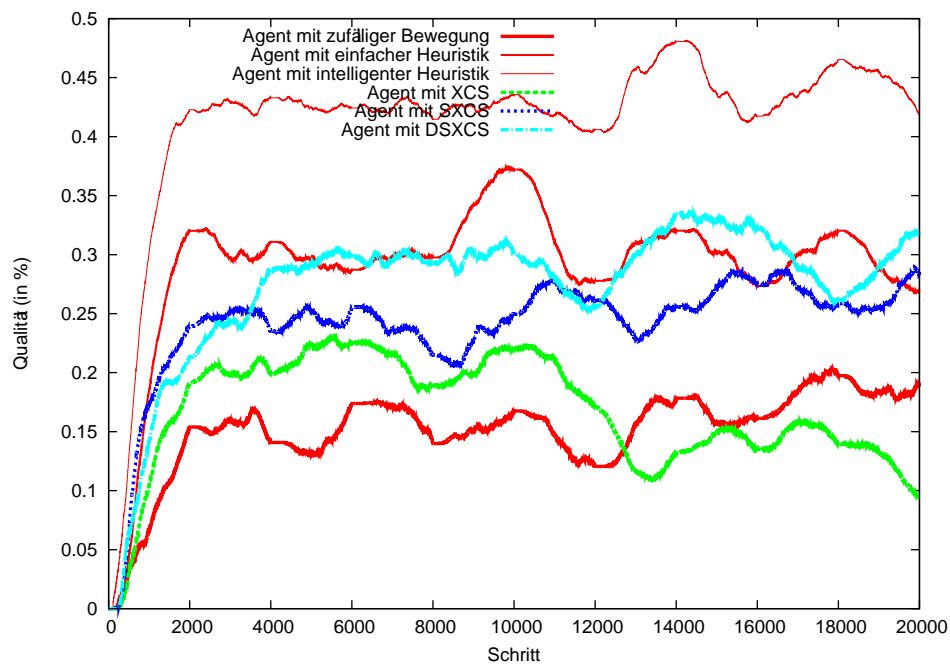


Abbildung 5.12: Verlauf des gleitenden Durchschnitts der Qualität (schwieriges Szenario) TODO

## 5.10 DSXCS

## 5.11 Beobachtete Strategien der Agenten TODOS

Analysiert man stichprobenweise die *classifier set* Listen der besten Agenten, findet man eine Reihe von wiederkehrenden Strategien in den *classifier* verschlüsselt. Wie immer sind diese Strategien im Kontext zu sehen, für sich alleine genommen mögen sie wenig Sinn ergeben, im Rahmen eines Multiagentensystems ist aber zu beachten, dass sich die Regeln im Umfeld anderer Agenten entwickelt haben.

Im Fall mit einem intelligenten Zielobjekt sind insbesondere die Regeln aufgefallen, bei denen der Agent in Richtung von TODO

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
```

Prediction Wert bei XCS fast immer identisch! max Prediction!

## 5.12 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.

TODO mehr, stärker auf Ergebnisse eingehen!

In den Tests erreichten die Heuristiken deutlich bessere Ergebnisse. Diesen Nachteil hat sich XCS in diesen Szenarien durch deutlich überlegene Flexibilität erkaufte 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 es keine eingebaute Heuristik. Man könnte zwar

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

etwas schneller zu Beginn Später schlechter, weil die falschen Aktionen belohnt werden (unter dem Gesichtspunkt der Überwachung konstruiert!)

- Verzögertes SXCS wieder rein greift die Idee des ursprünglichen XCS auf, dass eine Weile lang gesucht wird, allerdings ohne dass die classifier upgedated werden. und dann in einem Schlag upgedated

- Richtiges Haushalten ist gefragt

- kom erst dann sinnvoll, wenn lokal perfekt!

- lernen pausieren...

- da gelingt es nicht unbedingt eine Verbindung zu setzen... es müssten konkrete Be-  
fehlssätze belohnt werden, in denen mit Agenten interagiert wird...

- Gesamtpunktzahl eines Individuums hernehmen? DSXCS



# 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 herausgefiltert und bestimmt. Die Einführung in LCS, der Beschreibung des Standardverfahren XCS und der angepassten Implementierung für Überwachungsszenarios ermöglichen das Ausführen umfangreicher Tests. TODO

TODO irgendwo noch unterbringen, dass action set sizes gleich sind!

TODO oberes Limit der Genauigkeit, 84%, da ja geswitched wird. TODO erweitern, dass schlechte Ergebnisse nur geliefert werden konnten (relativ zum intelligenten Agent)w Geringen Unterschied ansprechen, mit zufälligem Algorithmus argumentieren, vielleicht ein Beispiel rechnen wo Qualität des zufälligen abgezogen wird!

## 6.1 Ergebnis TODO

Zentrales Ergebnis: Die Implementierung des XCS auf Überwachungsszenarios ist ohne wesentliche Veränderung möglich. TODO

Es wurde gezeigt, welche Änderungen und Anpassungen am XCS Algorithmus durchzuführen sind, will man ihn in einem Überwachungsszenario einsetzen. Insbesondere die Problemdefinition Bewertung! Kapitel 3.4

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 ein neues Gebiet von Problemfeldern in Verbindung mit dem XCS Algorithmus eröffnet. Dies erlaubt eine ganze Reihe von fortführenden Untersuchungen, die nachstehend kurz diskutiert werden. Zusätzlich stellen die nachfolgenden Ausführungen ausprobierte Lösungsansätze vor, die jedoch im Verlauf der Testphasen wieder verworfen und nicht weiter verfolgt wurden.

### 6.2.1 Ausweitung der Sensoren

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

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

### 6.2.2 Verwendung einer mehrwertigen *reward* Funktion

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

### 6.2.3 Untersuchung der Theorie

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

### 6.2.4 Untersuchung der *classifier*

Anekdotenweise wurden stichprobenartig *classifier* in Kapitel 5.11 untersucht. Dabei wurde per Hand und Augenmaß durch eine vorsortierte Liste an *classifier* von erfolgreichen Agenten geblättert und interessante herausgefischt. Dies diente lediglich zur Demonstration und um ein Gefühl zu erhalten, wie die Regelsätze der *classifier* aussehen. Es ist denkbar, dass man Methoden entwickeln könnte, um konkrete (vereinfachte) Strategien in der Form „Falls im Osten kein Hindernis und Zielobjekt nicht in Sicht, gehe nach Osten“ daraus abzuleiten.

### 6.2.5 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. Aufgezeigt wird dies insbesondere beim Vergleich zwischen 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. Ein Ansatz wurde in Kapitel 3.4 erwähnt, eine Abkehr von einem binären zu einem mehrwertigen *base reward* um die *reward* Funktion der intelligenten Heuristik besser abbilden zu können.

Wollte man sich dem Problemkreis der Kollaboration nähern, drängt sich auch die Lösungsidee auf, die Problemstellung an sich zu ändern z.B. in der Form, das Zielobjekt als überwacht einzustufen, wenn es gleichzeitig von mehreren Agenten oder von mehreren Seiten beobachtet wird.

### 6.2.6 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 ver-

bessern. Dies würde jedoch weitere Untersuchungen erfordern, welche über den Rahmen dieser Arbeit hinausgehen würden.

### 6.2.7 Abnehmende Wahrscheinlichkeit der *explore* Phase

Kapitel 3.5.5 stellte mehrere Arten des Wechsels zwischen der *explore* und *exploit* Phase vor. In der Literatur gibt es beispielsweise in [HR05] einen Ansatz, um die Wahrscheinlichkeit, in eine *explore* Phase zu wechseln bzw. in dieser Phase zu bleiben, während eines Durchlaufs mittels einer intelligenten Methode angepasst wird.

Wie dies bei Überwachungsszenarios ausgenutzt werden könnte ist noch unklar. Die Ergebnisse, bezüglich des Wechsels bei der Änderung des *base reward* Werts in Kapitel 5.6, deuten darauf hin, dass ein Ausgleich geschaffen werden kann, indem jeweils bei selten erlebten und häufig erlebten Situationen unterschiedliche Auswahlarten benutzt werden. Eine solche Anpassung könnte aber auch z.B. durch Analyse der aktuellen *classifier set* Liste erfolgen, was womöglich ein flexibleres Verhalten ermöglicht. Beispielsweise könnte man in Situationen, in der das *match set* nur *classifier* mit insgesamt niedrigem *experience* Wert aufweist, eher in die *explore* Phase wechseln. Weitere Untersuchungen in diesem Bereich wären angebracht.

### 6.2.8 Gesonderte Behandlung von neutralen Ereignissen

Beträgt bei einem neutralen Ereignis (siehe Kapitel 4.3.2) der *base reward* Wert 0, so wurde ausprobiert, die vom Stack genommenen Werte einfach zu verwerfen. Leitgedanke hierbei war es, das Verhalten eines Agenten nicht negativ zu bewerten, auch wenn er es für längere Zeit nicht schafft, das Zielobjekt in Sichtweite zu bekommen.

TODO!

Dies begründet sich damit, dass er trotzdem ein Gebiet überwacht hat, in dem die Aufenthaltswahrscheinlichkeit des Zielobjekts größer 0 war und somit, was das globale Problem betrifft, eigentlich zielführend gehandelt hat. 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 erkennt 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  $\beta$  von 0,1) so stark ist, dass der jeweilige Agent im nächsten Problem kein Ziel mehr findet. Mit geringerer Lernrate  $\beta$  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.

### 6.2.9 Anpassung des *maxStackSize* Werts

Bei der Besprechung von „Ereignissen“ in Verbindung mit SXCS in Kapitel 4.3.2 hat man gesehen, dass sich die optimalen Werte für das Säulenszenario und das schwierige Szenario stark unterscheiden. Um sich die Anpassung mittels Testläufen an das jeweilige Szenario

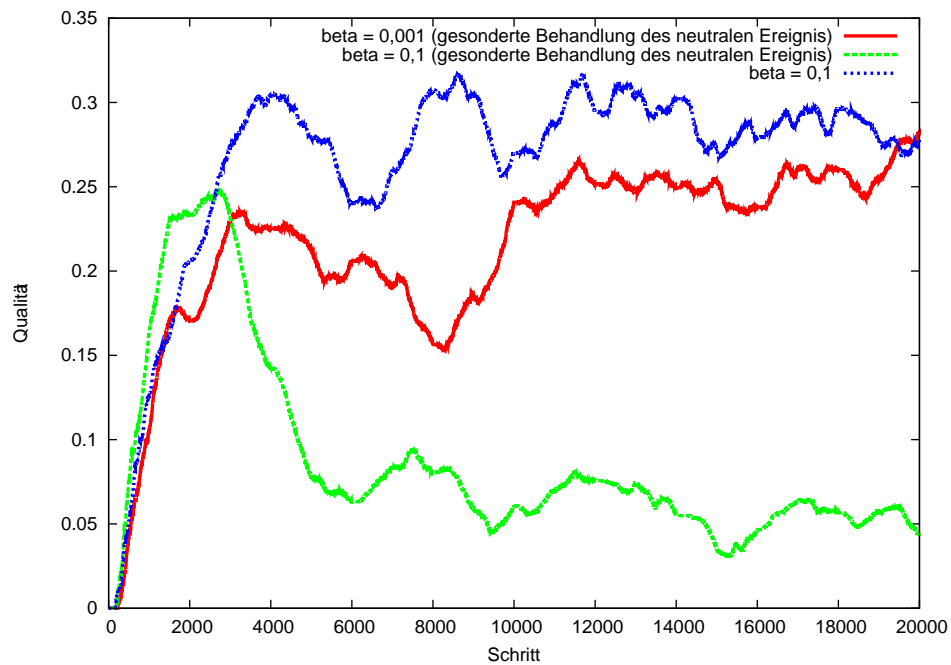


Abbildung 6.1: Auswirkung des Parameters *learning rate*  $\beta$  auf den gleitenden Durchschnitt der Qualität im schwierigen Szenario, Bewegung des Zielobjekts ohne Richtungsänderung, Geschwindigkeit 1, 8 Agenten mit SXCS Algorithmus, 2000 Schritte



zu sparen, wäre es für den Algorithmus sinnvoll, eine Methode zu entwickeln, mit der sich der *maxStackSize* Wert während des Laufs an das jeweilige Szenario anpassen kann. Wie dies konkret auszusehen hat, ist noch offen. Zuvor müsste der Algorithmus selbst in der Theorie näher analysiert werden.

### 6.2.10 Lernendes Zielobjekt

Sicher interessant ist auch der umgekehrte Ansatz, bei dem die Rollen von Agent und Zielobjekt, was die Bestimmung der Qualität betrifft, vertauscht wird. Dann wäre das Zielobjekt das Objekt, das lernt und den Agenten ausweichen muss. Bei dieser Problemstellung fällt zwar der kollaborative Aspekt weg, es hat aber den Vorteil, mit derselben Simulation dieses neue Problem untersuchen zu können. Die Implementierung ist sehr simpel, es muss lediglich die *reward* Funktion entsprechend abgeändert werden, damit beispielsweise das Ausweichen von Agenten positiv bewertet wird. Bis auf die in Kapitel 2.5 erwähnte Sprungeigenschaft ist ein lernendes Zielobjekt ansonsten praktisch identisch mit der XCS bzw. SXCS Implementierung. Bei anfänglichen Tests konnten keine besonderen Erkenntnisse gewonnen werden und der Ansatz wurde deshalb nicht weiter verfolgt.

### 6.2.11 Verworfen Szenarien

Szenarien, die keine zusätzlichen Beobachtungen erbrachten, bzw. nur das Betrachten unbedeutender Teilaspekte ermöglichten, wurden verworfen.

So sind u.a. zu nennen,

- das Szenario „Labyrinth“, bei denen die Agenten wegen der Agenten wahrscheinlich an den mangelnden Fähigkeiten der Sensoren scheiterten,
- das Szenario „Raum“, was ein vereinfachtes „schwieriges Szenario“ mit einem „Raum“ mit einer Öffnung in der Mitte darstellte, welches sich als zu einfach zu lösen her-

ausstellte und

- das Szenario „Kreuz“, bei dem die Hindernisse in der Mitte des Raums ein Kreuz bilden, jedoch keine bedeutend anderen Ergebnisse lieferte als Szenarien mit zufällig verteilten Hindernissen.

### 6.3 Vorgehen und verwendete Hilfsmittel und Software

Zu Beginn stellte sich die Frage, welche Software zu benutzen ist, da bereits frühzeitig die komplexe Problemstellung absehbar war. 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.

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

Leider bietet die Fachliteratur nur ein begrenztes Angebot an Beschreibung von Arbeiten dar, die sich mit der konkreten XCS Implementierung und deren Umsetzung beschäftigen. Insbesondere im Hinblick auf den nichtklassischen Fall des Überwachungsszenarios musste fast bei 0 begonnen werden. Ein Rückgriff auf bestehende Bibliotheken war deshalb nicht möglich, ursprünglich geplante Untersuchungen komplexerer Systeme wie zentrale Steuerung, Austausch von Regeln etc. waren deswegen zu streichen.

Konsequenterweise beschränkten sich die Untersuchungen auf den Fall, in dem lokale Informationen ohne zentrale Steuerung und ohne Kommunikation auftreten. Dies

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

Das Simulationsprogramm samt zugehöriger Oberfläche [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 [El100] erstellt. Weitere Graphen und Darstellungen wurden OpenOffice.org Impress und OpenOffice.org Calc [OO0] angefertigt.

Besonders hilfreich für die Programmierung der Simulation, der Testumgebung, des Konfigurationsmanagements und der Oberfläche waren hierzu Erfahrungen aus der 3-monatigen Studienarbeit zum Thema *estimation of distribution algorithms* am Institut für Angewandte Informatik und Formale Beschreibungsverfahren (AIFB) der Universität Karlsruhe (TH) bei Dr. Jürgen Branke wie auch einer halbjährigen Arbeit zum Thema Zellularautomaten am Institut für Algorithmen und Kognitive Systeme der Universität Karlsruhe (TH) bei Herrn Dr. rer. nat. Thomas Worsch.

Wesentlicher Bestandteil der Konfigurationsoberfläche 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 L<sup>A</sup>T<sub>E</sub>X Editor 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 sind die Ergebnisse aller bisherigen Läufe in einer Datenbank angeordnet, 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.

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, welches das Konfigurationsprogramm erstellt.

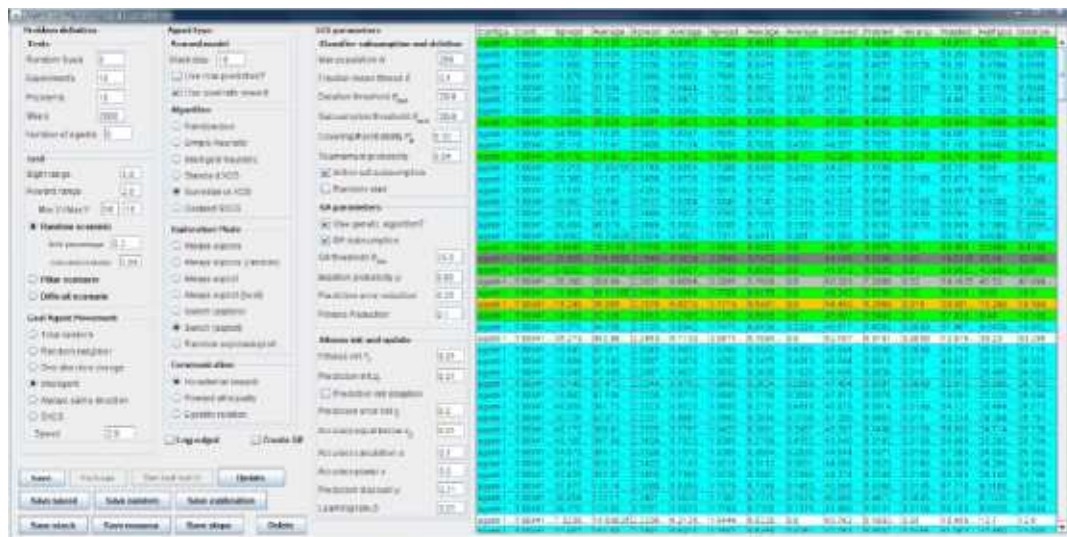


Abbildung 6.2: Screenshot des Konfigurationsprogramms (Gesamtübersicht)

The screenshot displays a configuration window with the following sections and settings:

- Problem definition**
  - Tests
    - Random Seed: 0
    - Experiments: 10
    - Problems: 10
    - Steps: 2000
    - Number of agents: 9
  - Grid
    - Sight range: 5.0
    - Reward range: 2.0
    - Max X / Max Y: 16 / 16
    - ☒ Random scenario
      - Grid percentage: 0.2
      - Connection factor: 0.99
    - ☐ Pillar scenario
    - ☐ Difficult scenario
  - Goal Agent Movement
    - ☐ Total random
    - ☐ Random neighbor
    - ☐ One direction change
    - ☒ Intelligent
    - ☐ Always same direction
    - ☐ SXCS
    - Speed: 2.0
- Agent type**
  - Reward model
    - Stack size: 16
    - ☐ Use max prediction?
    - ☒ Use quadratic reward
  - Algorithm
    - ☐ Randomized
    - ☐ Simple heuristic
    - ☐ Intelligent heuristic
    - ☐ Standard XCS
    - ☒ Surveillance XCS
    - ☐ Delayed SXCS
  - Exploration Mode
    - ☐ Always explore
    - ☐ Always explore (random)
    - ☐ Always exploit
    - ☐ Always exploit (best)
    - ☐ Switch (explore)
    - ☒ Switch (exploit)
    - ☐ Random explore/exploit
  - Communication
    - ☒ No external reward
    - ☐ Reward all equally
    - ☐ Egoistic relation
- LCS parameters**
  - Classifier subsumption and deletion
    - Max population  $N$ : 256
    - Fraction mean fitness  $\delta$ : 0.1
    - Deletion threshold  $\theta_{del}$ : 20.0
    - Subsumption threshold  $\theta_{sub}$ : 20.0
    - Covering # probability  $P_g$ : 0.33
    - Tournament probability: 0.64
    - ☒ Action set subsumption
    - ☐ Random start
  - GA parameters
    - ☒ Use genetic algorithm?
    - ☒ GA subsumption
    - GA threshold  $\theta_{GA}$ : 25.0
    - Mutation probability  $\mu$ : 0.05
    - Prediction error reduction: 0.25
    - Fitness Reduction: 0.1
  - Fitness init and update
    - Fitness init  $F_i$ : 0.01
    - Prediction init  $p_i$ : 0.01
    - ☐ Prediction init adaption
    - Prediction error init  $e_i$ : 0.0
    - Accuracy equal below  $\epsilon_0$ : 0.01
    - Accuracy calculation  $\alpha$ : 0.1
    - Accuracy power  $\nu$ : 5.0
    - Prediction discount  $\gamma$ : 0.71
    - Learning rate  $\beta$ : 0.01

Buttons at the bottom: Save, Package, Run last batch, Update, Save speed, Save random, Save exploration, Save stack, Save maxpop, Save steps, Delete.

Abbildung 6.3: Screenshot des Konfigurationsprogramms (Konfigurationsbereich)

# Anhang A

## Implementation

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

### A.1 Implementierung eines Problemablaufs

In der Schleife der Funktion zur Berechnung eines Experiments (Programm [A.1](#)) wird die Funktion zur Berechnung 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.acquireNewSensorData();
12          a.calculateNextMove(gaTimestep);
13      }
14      BaseAgent.goalAgent.acquireNewSensorData();
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.acquireNewSensorData();
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. Dagegen wurde die für diese Arbeit erstellte Implementierung jeweils mit bis über 100 *action set* Listen programmiert, denen ein *classifier* Mitglied sein kann. Deswegen wurde die Optimierung entfernt und durch eine dezentrale Verwaltung mit einem *Observer* ersetzt, jede Änderung des *numerosity* Wertes hat also die Änderung aller *action set* Listen zur Folge, in der der *classifier* Mitglied ist.

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

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

Die Standardimplementierung weist ein weiteres Problem auf. So enthält der *fitness*

Wert eines *classifier* als Optimierung bereits den *numerosity* Wert als Faktor, während bei der Aktualisierung des *numerosity* Werts der *fitness* Wert nicht aktualisiert wurde. Das hat zur Folge, dass theoretisch der *fitness* Wert eines *classifier* fast den *max population* Wert annehmen kann, wenn ein *classifier* mit *numerosity* und *fitness* Wert in der Höhe von *max population* auf einen *numerosity* Wert von 1,0 reduziert wird. Dies betrifft die Funktion `public void addNumerosity(int num)` der Klasse *XClassifier* in der Datei *XClassifier.java*. Die Korrektur besteht darin, den *fitness* Wert mit dem Quotienten aus dem neuen durch den alten *numerosity* Wert zu multiplizieren. Die korrigierte Fassung ist in Programm A.9 dargestellt.

Möglicherweise kann man diesen Fehler durch Veränderung der Parameter oder längere Laufzeiten kompensieren. Es ergibt jedoch keinen Sinn, dass beim Subsummieren bzw. Löschen eines *micro classifier* der *fitness* Wert verändert wird. In Tests haben sich nur minimale Unterschiede ergeben. Beispielsweise ergab sich (auf dem Säulenszenario mit 8 Agenten mit SXCS und einem Zielobjekt mit einfacher Richtungsänderung) 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.fitness;
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 base reward Wert und
3   * gibt ihn an die zugehörigen action set Listen weiter. Es wird
4   * kein maxPrediction Wert berechnet oder weitergegeben.
5   *
6   * @param reward Wahr wenn der Zielobjekt in Sicht war.
7   * @param best_value Bester Wert des vorangegangenen action set Listen
8   * @param is_event Wahr wenn diese Funktion wegen eines Ereignisses,
9   *                d.h. einem positiven reward Wert, aufgerufen wurde
10  */
11
12  public void collectReward(int start_index, int action_set_size,
13                          boolean reward, double best_value, boolean is_event) {
14      double corrected_reward = reward ? 1.0 : 0.0;
15
16      /**
17       * Aktualisiere eine ganze Anzahl von action set Listen
18       */
19      for(int i = 0; i < action_set_size; i++) {
20
21          /**
22           * Benutze aufsteigenden bzw. absteigenden reward Wert bei einem
23           * positiven bzw. negativen Ereignis
24           */
25          if(is_event) {
26              corrected_reward = reward ?
27                  calculateReward(i, action_set_size) :
28                  calculateReward(action_set_size - i, action_set_size);
29          }
30
31          /**
32           * Aktualisiere die action set Liste mit dem bestimmten reward Wert
33           * und gebe bei allen anderen action set Listen den reward Wert
34           * weiter wie beim multi step Verfahren
35           */
36          ActionClassifierSet action_classifier_set =
37              historicActionSet.get(start_index - i);
38          action_classifier_set.updateReward(corrected_reward, 0.0, 1.0);
39      }

```

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 die classifier set Liste mit zum Status passenden
15  * classifier welche insgesamt alle möglichen Aktionen abdecken.
16  */
17      classifierSet.coverAllValidActions(
18          lastState, getPosition(), gaTimestep);
19
20  /**
21  * Bestimme alle zum Status passenden classifier.
22  */
23      lastMatchSet = new AppliedClassifierSet(lastState, classifierSet);
24
25  /**
26  * Entscheide auf welche Weise die Aktion ausgewählt werden soll,
27  * wähle 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)

# Literaturverzeichnis

- [BDE<sup>+</sup>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.
- [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<sup>+</sup>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