DIPLOMARBEIT

XCS in dynamischen Multiagenten-Überwachungsszenarien ohne globale Kommunikation

von

Clemens Lode

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

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

Karlsruhe, 30.03.2009

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Kapitel 1

Einführung

Ein aktuelles Forschungsgebiet aus dem Bereich der *learning classifier systems* (LCS) stellen die sogenannten *accuracy based* LCS (XCS) dar. In der Basis entspricht XCS einem LCS, d.h. eine Reihe von Regeln, bestehend jeweils aus einer Kondition und einer Aktion, 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 soll das *multi step* Verfahren sein, bei dem die Bewertung (der *reward* der Regeln erst nach einigen Schritten verfügbar ist und an zurückliegende Regeln sukzessive weitergeleitet wird um möglichst alle beteiligten Regeln an dem *reward* zu beteiligen.
Bisherige Anwendungen haben sich hauptsächlich auf statische Szenarien mit nur einem XCS oder mit mehreren Agenten mit globaler Organisation und Kommunikation beschränkt. Diese Arbeit hat sich auf die Problemstellung konzentriert, wie man XCS modifizieren sollte, damit es ein dynamisches Überwachungsszenario, mit sich bewegendem Zielobjekt und mehreren Agenten, im Vergleich zu zufälliger Bewegung möglichst gut bestehen.
Neben der Anpassung der Implementation, damit XCS für eine solche Problemstellung

anwendbar ist, wurden weitere Modifikationen durchgeführt, die in einigen Fällen zu deutlich besseren Ergebnissen als die der Standardimplementation führten.
Außerdem wurde untersucht, wie eine einfache Kommunikation ohne globale Steuereinheit stattfinden kann, um das Ergebnis weiter zu verbessern. Im Wesentlichen war dazu eine weitere Anpassung von XCS vonnöten, so dass die Implementierung auch mit (durch die Kommunikation) zeitverzögerten und externen *rewards* arbeiten konnte. Wesentliche Schlußfolgerung ist, dass sich unterschiedliche Szenarien unterschiedlich gut für Kommunikation eignen, dass Kommunikation Möglichkeiten zur Anpassung bietet um mit einer variablen, unbekannten Feldgröße besser zurecht zu kommen und, dass es Szenarien gibt, in denen Kommunikation signifikante Vorteile erbringt.
Erfolgversprechende Ansatzpunkte für weitere Forschung gibt es im Bereich der mathematischen Begründung, warum die Implementierung Vorteile erbringt, im Ausbau der Untersuchung von Kommunikation zwischen den Agenten in Verbindung mit XCS und in der Anwendung der gefundenen Ergebnisse in anderen Problemstellungen ähnlicher Natur.

Kapitel 2

Szenario

Im Wesentlichen sollen die hier besprochenen Algorithmen in einem Überwachungsszenario getestet werden, d.h. die Qualität eines Algorithmus wird anhand des Anteils der Zeit bewertet, die er das Zielobjekt überwachen konnte, relativ zur Gesamtzeit.
Verwendetes Umfeld wird ein quadratischer Torus sein, der aus quadratischen Feldern besteht.
Jedes bewegliche Objekt auf einem Feld kann sich in einem Zeitschritt nur auf eines der vier Nachbarfelder bewegen (mit Ausnahme des Zielobjekts, welches mehrere Bewegungen in einem Zeitschritt durchführen kann).
Die Felder können entweder leer oder durch ein Objekt besetzt sein. Besetzte Felder können nicht betreten werden, eine Bewegung auf ein solches Feld schlägt 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 anhand eines jeweils bestimmten Algorithmus und bestimmter Sensordaten.
Bis Kapitel 11 wird ausschließlich der Fall ohne Kommunikation betrachtet, d.h. die Agenten können untereinander keine Informationen austauschen und müssen sich alleine auf ihre Sensordaten verlassen.

2.1 Probleminstanzen

Eine einzelne Probleminstanz entspricht einem Torus mit einer (abhängig vom verwendeten Random-Seed-Wert) bestimmten Anfangsbelegung mit bestimmten Objekten und bestimmten Parametern zur Sichtbarkeit. Soweit nicht anders angegeben sollen hier Prombleninstanzen der Größe 16x16 Felder betrachtet werden, insbesondere beziehen sich die Ergebnisse der Tests auf diesen Fall.
Jedes Problem soll sich, sofern nicht anders angegeben, über 500 Zeitschritte ziehen. Ein einzelnes Experiment entspricht dem Test einer Anzahl von Probleminstanzen die jeweils mit einer Reihe von *random seed* Werten initialisiert werden. In einem Durchlauf werden mehrere Experimente (jeweils mit unterschiedlichen Reihen an *random seed* Werten) durchgeführt. und über 10 Probleme, gemittelt über 10 Experimente

2.2 Sichtbarkeit

Der Parameter *sight range* bzw. *reward range* einer Probleminstanz 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 2.4) und wird immer kleiner als der *sight range* Wert gewählt. Über die Sensoren kann ein Agent feststellen, ob sich Objekte in welcher der beiden Reichweiten befindet.

2.3 Kollaboration

Wesentliches Hauptaugenmerk der Gestaltung der Szenarien soll Kollaboration sein, d.h. die Aufgabe soll mit Hilfe mehrerer Agenten gemeinsam gelöst werden. Eine erfolgreiche Überwachung soll deswegen so definiert sein, dass sich ein beliebiger Agent in Überwachungsreichweite des Ziels befindet. Um sicherzustellen, dass diese Aufgabe nicht auch ein einzelner Agent erfüllen kann, soll sich das Zielobjekt schneller fortbewegen als die einzelnen Agenten.
Würde sich das Zielobjekt genauso langsam oder langsamer als die Agenten bewegen, dann wäre das Problem sehr simpel, da das Zielobjekt Schwierigkeiten hätte, Agenten abzuschütteln, deren einzige Strategie es ist, sich in Richtung des Zielobjekts zu bewegen.

2.4 Qualität

Qualität eines Algorithmus zu einem Problem wird anhand des Anteils der Zeit berechnet, die er das Zielobjekt während des Problems überwachen (d.h. das Zielobjekt innerhalb einer Distanz von höchstens *reward range* halten) konnte, relativ zur Gesamtzeit.
Qualität eines Algorithmus zu einer Anzahl von Problemen (also einem Experiment) wird Anhand des Gesamtanteil der Zeit berechnet, die er das Zielobjekt während aller Probleme überwachen konnte, relativ zur Gesamtzeit aller Probleme.
Qualität eines Algorithmus entspricht dem Durchschnitt der Qualitäten des Algorithmus mehrerer Experimente.
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.
Halbzeitqualität eines Algorithmus zu einer Anzahl von Problemen entspricht dem Anteil

der Zeit, die der Algorithmus das Zielobjekt während jeweils der zweiten Hälfte des Problems überwachen konnte, relativ zur halben Gesamtzeit aller Probleme.
Die Halbzeitqualität eines Algorithmus entspricht dem Durchschnitt aller Halbzeitqualitäten des Algorithmus mehrerer Experimente.
Ein Vergleich der Qualität mit der Halbzeitqualität eines Algorithmus ermöglicht einen Einblick, wie gut sich der Algorithmus verhält, nachdem er sich auf das Problem bereits eine Zeit lang einstellen konnte.

2.5 Dynamik

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, den diese Arbeit von vielen anderen unterscheidet, Gegenstand der Untersuchung in der Literatur sind eher statische Probleme wie z.B. 6-Multiplexer Problem und Maze1 (z.B. in [10]) bzw. Maze5, Maze6, Woods14 (in [7]) oder Probleme bei denen die Agenten globale Information besitzen. Eine nähere Diskussion zur Literatur gibt es in Kapitel 7.

2.6 Startkonfigurationen des Torus

Getestet wurden eine Reihe von Szenarien (in Verbindung mit unterschiedlichen Werten für die Anzahl der Agenten, Größe des Torus und Art und Geschwindigkeit des Zielobjekts). Wesentliche Rolle spielt hier die Verteilung der Hindernisse.
In den folgenden Abbildungen repräsentieren rote Felder jeweils Hindernisse, weiße Felder

jeweils Agenten und das grüne Feld jeweils das Zielobjekt. In Abbildung (2.1) ist ein Szenario ohne Hindernisse und mit zufälliger Verteilung der Agenten und zufälliger Position des Zielobjekts dargestellt.

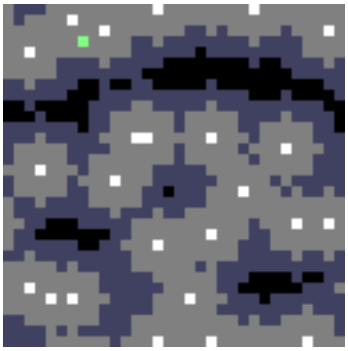

Abbildung 2.1: "Leeres Szenario" ohne Hindernisse

2.6.1 Zufälliges Szenario

Zwei Parameter bestimmen das Aussehen des zufälligen Szenarios, zum einen der Prozentsatz an Hindernissen an der Gesamtzahl der Felder des Torus (Hindernissanteil λ_h), zum anderen der Grad inwieweit die Hindernisse zusammenhängen (Verknüpfungsfaktor λ_p.
Bei der Erstellung des Szenarios bestimmt λ_p die Wahrscheinlichkeit für jedes einzelne angrenzende freie Feld, dass beim Verteilen der Hindernisse nach dem Setzen eines Hindernisses dort sofort ein weiteres Hindernis gesetzt wird. $\lambda_p = 0.0$ ergäbe somit eine völlig zufällig verteilte Menge an Hindernissen, während ein Wert von 1.0 eine oder mehrere stark zusammenhängende Strukturen schafft.
Wird der Prozentsatz an Hindernissen λ_h auf 0.0 gesetzt, dann werden Hindernisse keine Hindernisse gesetzt.

In Abbildungen (2.2), (2.3), (2.4) und (2.5) werden Beispiele für zufällige Szenarien

gegeben mit $\lambda_h = 0.05$, 0.1, 0.2 bzw. 0.4 und $\lambda_p = 0.01$, 0.5 bzw. 0.99.

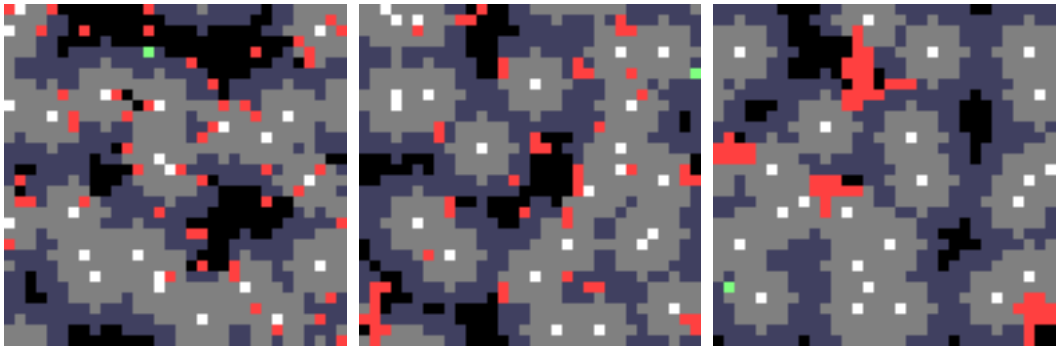

Abbildung 2.2: "Zufälliges Szenario" mit $\lambda_h = 0.05$ und $\lambda_p = 0.01$, 0.5 bzw. 0.99.

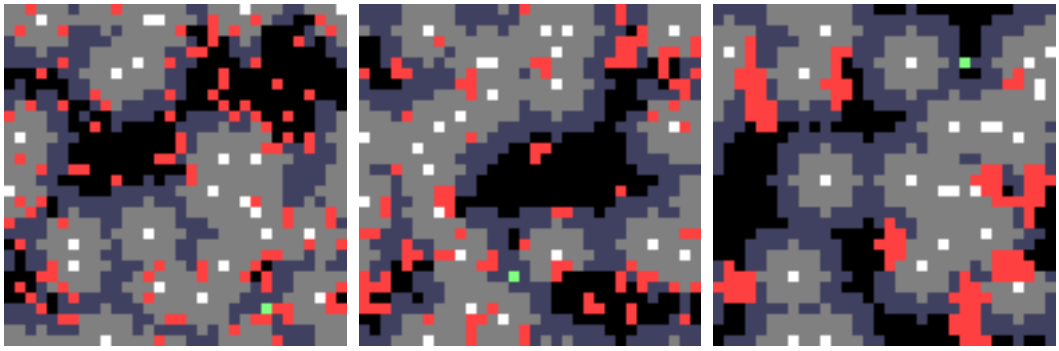

Abbildung 2.3: "Zufälliges Szenario" $\lambda_h = 0.1$ und $\lambda_p = 0.01$, 0.5 bzw. 0.99.

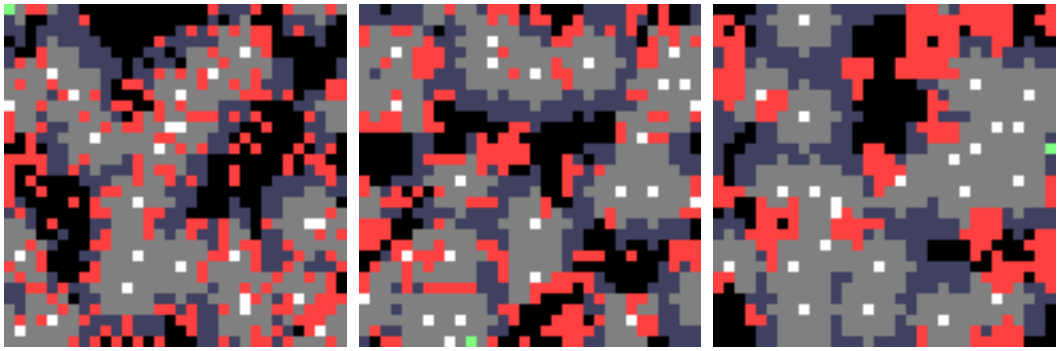

Abbildung 2.4: "Zufälliges Szenario" mit $\lambda_h = 0.2$ und $\lambda_p = 0.01$, 0.5 bzw. 0.99.

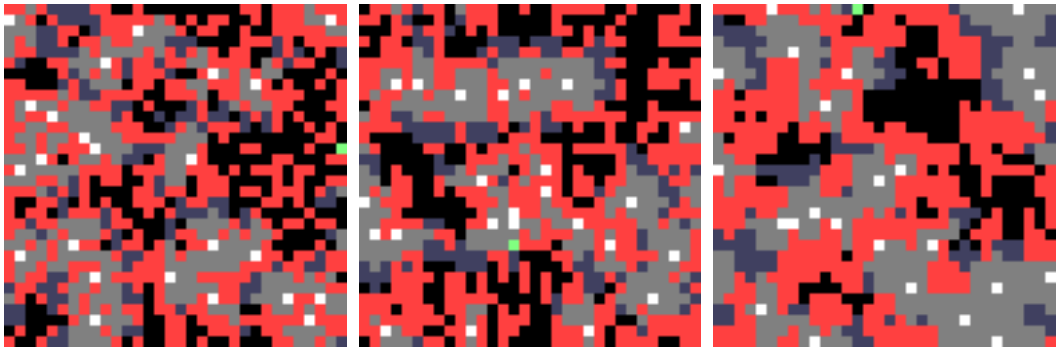

Abbildung 2.5: "Zufälliges Szenario" mit $\lambda_h = 0.4$ und $\lambda_p = 0.01$, 0.5 bzw. 0.99.

2.6.2 "Säulen Szenario"

Hier werden mit jeweils 7 Feldern Zwischenraum zueinander regelmäßig Hindernisse verteilt. Idee ist, dass die Agenten eine kleine Orientierungshilfe besitzen aber gleichzeitig möglichst wenig Hindernisse verteilt werden. Das Zielobjekt startet an zufälliger Position, die Agenten starten mit möglichst großem Abstand zum Zielobjekt. Abbildung 2.6 zeigt ein Beispiel bei der das Zielobjekt sich in der Mitte und die Agenten am Rand befinden.

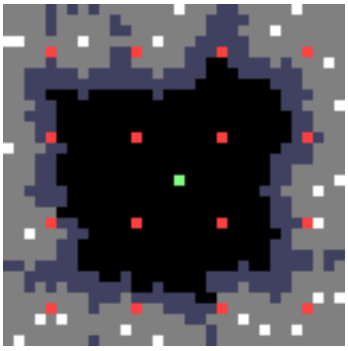

Abbildung 2.6: "Säulen Szenario" mit regelmäßig angeordneten Hindernissen und zufälliger Verteilung von Agenten mit möglichst großem Abstand zum Zielobjekt

2.6.3 "Kreuz Szenario"

Hier gibt eine horizontale Reihe aus Hindernissen halber Gesamtbreite welche durch eine vertikale Reihe aus Hindernissen halber Gesamthöhe in der Mitte geschnitten wird. Agen-

ten und das Zielobjekt werden zufällig verteilt. Abbildung 2.7 zeigt ein Beispiel für ein solches Szenario.

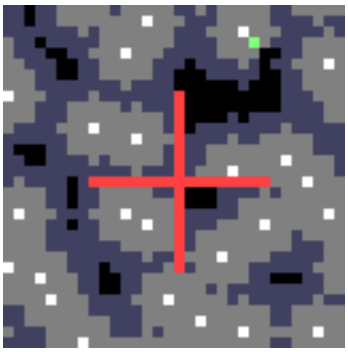

Abbildung 2.7: "Kreuz Szenario" mit kreuzförmiger Anordnung der Hindernisse und zufälliger Verteilung der Agenten und des Zielobjekts

2.6.4 "Raum Szenario"

Auf dem Torus wird ein Rechteck der halben Gesamthöhe und -breite des Torus erstellt, welches im Norden eine Öffnung von 4 Feldern Breite aufweist. Der Zielagent startet in der Mitte des Raums, alle Agenten starten mit maximaler Distanz zu den Hindernissen an zufälliger Position. Abbildung 2.8 zeigt ein Beispiel für eine Startkonfiguration eines solchen Szenarios.

2.6.5 "Schwieriges Szenario"

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

In regelmäßigen Abständen (7 Felder Zwischenraum) befindet sich eine vertikale Reihe von Hindernissen mit Öffnungen von 4 Feldern Breite abwechselnd im oberen Viertel und

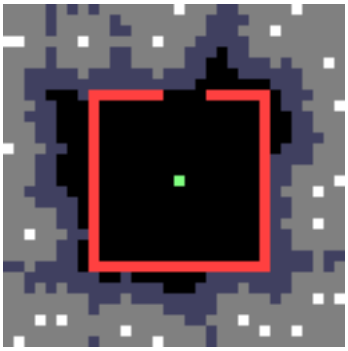

Abbildung 2.8: “Raum Szenario” mit quaderförmiger Anordnung der Hindernisse mit Öffnung im Norden, zufälliger Verteilung der Agenten am Rand und zu Beginn im Zentrum startendem

dem unteren Viertel.

Idee dieses Szenarios ist zu testen, inwieweit die Agenten durch die Öffnungen zum Ziel finden können. Ohne Orientierung an den Öffnungen und anderen Agenten ist es sehr schwierig, sich durch das Szenario zu bewegen. Abbildung 2.9 zeigt die Startkonfiguration des Szenarios.

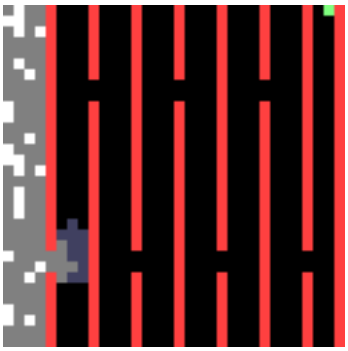

Abbildung 2.9: “Schwieriges Szenario” mit fester, wallartiger Verteilung von Hindernissen in regelmäßigen Abständen und mit Öffnungen, mit den Agenten mit zufälligem Startpunkt am linken Rand und dem Zielagenten mit festem Startpunkt rechts oben

2.6.6 “Irrgarten Szenario”

Der Code zur Generierung der Hindernisse stammt aus [5]. In den “Gängen” des Irrgartens herrscht jeweils eine Breite von 2 Feldern. Die anfängliche Verteilung der Agenten und des Zielobjekts geschieht zufällig. Abbildung 2.10 zeigt ein Beispiel für eine Startkonfiguration eines Irrgartens.

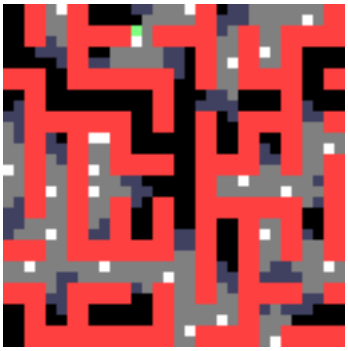

Abbildung 2.10: “Irrgarten Szenario” mit einer Anordnung von Hindernissen in der Art, dass es nur wenige Pfade gibt.

Kapitel 3

Agenten

3.1 Sensoren

Jeder Agent besitzt 3 Gruppen mit jeweils 8 Sensoren. Alle Sensoren sind visuelle, binäre Sensoren mit begrenzter Reichweite und können nur feststellen, ob sich in ihrem Sichtbereich ein entsprechendes Objekt befindet oder nicht. Andere Objekte blockieren die Sicht, Sichtlinien werden durch einen einfachen Bresenham-Algorithmus bestimmt.
Jede Gruppe von Sensoren nimmt einen anderen Typ von Objekt wahr. Die erste Gruppe nimmt das Zielobjekt, die zweite Gruppe andere Agenten und die dritte Gruppe Hindernisse wahr.
Sensoren sind jeweils bestimmte Richtungen ausgerichtet (Norden, Osten, Süden und Westen, wobei auf den Abbildungen in dieser Arbeit Norden immer oben ist) und wird auf "wahr" (1) gesetzt, wenn sich in dem Sichtbereich des Sensors ein entsprechendes Objekt befindet.
Die 8 Sensoren in einer Gruppe sind in 4 Richtungen mit jeweils einem Sensorenpaar aufgeteilt. Ein Sensorenpaar besteht aus zwei Sensoren mit unterschiedlich großer Sichtweite, mit der der Agent also rudimentär die Entfernung zu anderen Objekten feststellen kann.

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

TODO darauf Bezug nehmen

```
TODO Beispiele für Sensoren
a) 10 00 00 00 . 00 00 00 00 . 00 00 00 00
b) 00 00 00 00 . 00 00 00 00 . 00 00 00 00
c) 00 00 00 00 . 11 00 10 00 . 00 00 00 00
d) 00 00 00 00 . 00 00 00 00 . 00 00 00 00
e) 00 00 00 00 . 00 00 00 00 . 00 00 00 00
```

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 2.2). Allgemein soll *sight range = 5.0* und *reward range = 2.0* betragen, der überwachte Bereich ist also eine Teilmenge des sichtbaren Bereichs. Anzumerken sei hier, dass deshalb ein Sensorenpaar (01) nicht auftreten kann.
Sei $r(O1, O2)$ die Distanz zwischen dem Objekt, das die Sensordaten erfasst und dem nächstliegenden Objekt des Typs, den der Sensor wahrnehmen kann, dann gibt es folgende Fälle:

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

In Abbildung 3.1 sind alle Sichtkegel (dunkler und heller Bereich) und Überwachungsreichweiten (heller Bereich) für die einzelnen Richtungen dargestellt.

TODO von 3-5 auf 2-5 umändern

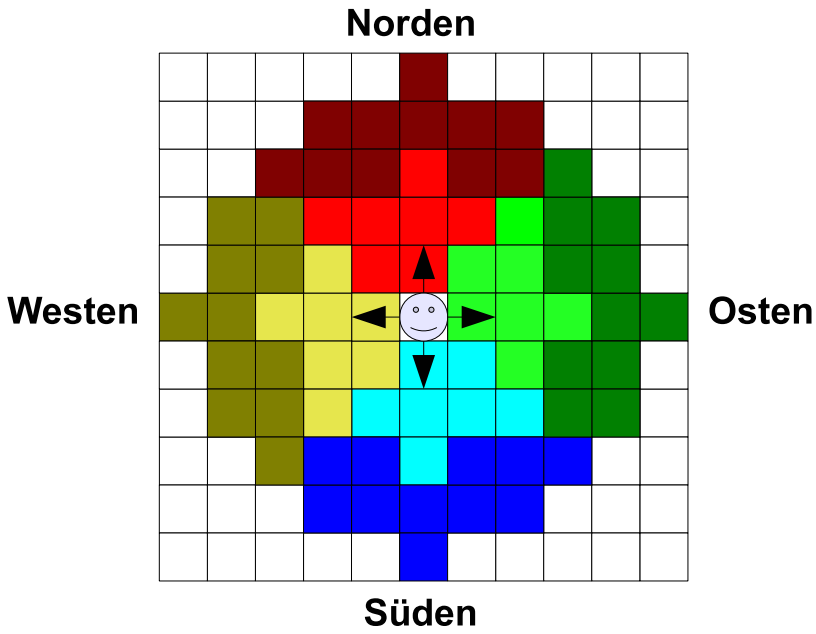

Abbildung 3.1: Sicht- (2.0) und Überwachungsreichweite (5.0) eines Agenten, jeweils für die einzelnen Richtungen

3.2 Fähigkeiten

Jeder Agent kann in jedem Schritt zwischen vier verschiedenen Aktionen wählen, die den vier Richtungen (Norden, Osten, Süden, Westen), bei der der Agent sich nicht bewegt, entsprechen. Agenten können pro Zeiteinheit genau einen Schritt durchführen. Das Zielobjekt kann je nach Szenarioparameter mehrere Schritte ausführen.

3.3 Ablauf der Bewegung

Alle Agenten werden nacheinander in der Art abgearbeitet, dass der jeweilige Agent die aktuellen Sensordaten aus der Umgebung holt und anhand dieser die nächste Aktion bestimmt. 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 nicht weiter bestraft. Eine detaillierte Beschreibung der Bewegung im Kontext anderer Agenten und Programmteile wird in Kapitel 5.2 gegeben.

3.4 Grundsätzliche Algorithmen der Agenten

Neben denjenigen Algorithmen, die auf LCS basieren und in Kapitel 7 besprochen werden, gibt es folgende Grundtypen, die dazu dienen, die Qualität der anderen Algorithmen einzuordnen. Wesentliches Merkmal im Vergleich zu auf LCS basierenden Algorithmen ist, dass sie statische, handgeschriebene Regeln benutzen und den Erfolg oder Misserfolg ihrer Aktionen ignorieren, d.h. ihre Regeln nicht anpassen. Die in Kapitel 5.6 erwähnte, im Programm aufgerufenen Funktion *calculateReward()* soll für die hier aufgelisteten Algorithmen also einer leeren Funktion entsprechen, während hier jeweils die Implementierung der *calculateNextMove()* Funktion aufgeführt werden soll.

3.4.1 "Zufälliger Algorithmus"

Bei diesem Algorithmus wird in jedem Schritt wird eine zufällige Aktion ausgeführt. Abbildung (3.2) zeigt eine Beispielsituation, bei der der Agent jegliche Sensordaten ignoriert und eine Aktion zufällig auswählen wird. Programm 3.4.1 zeigt den zugehörigen Quelltext.

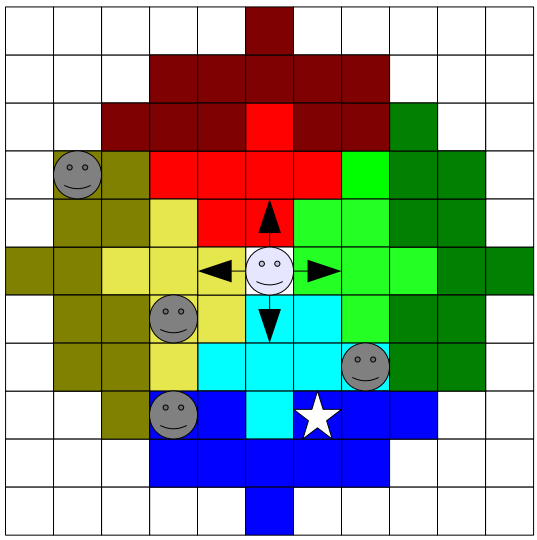

Abbildung 3.2: Agent bewegt sich in eine zufällige Richtung (oder bleibt stehen)

Programm 1 Berechnung der nächsten Bewegung beim zufälligen Algorithmus

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

3.4.2 Einfache 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 (3.3) 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 3.4.2 zeigt den zugehörigen Quelltext.

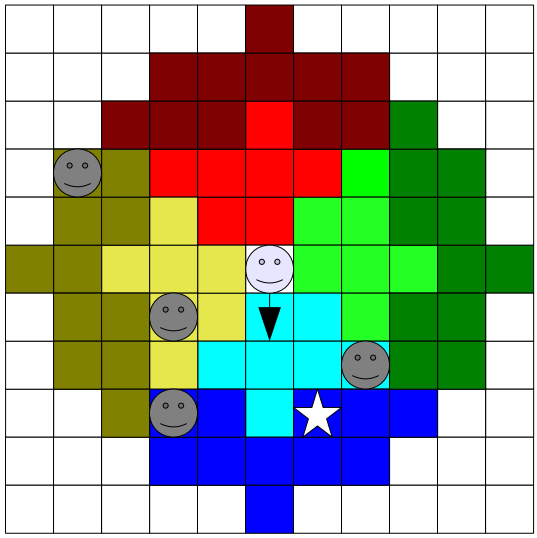

Abbildung 3.3: Agent mit einfacher Heuristik: Sofern es sichtbar ist bewegt sich der Agent auf das Zielobjekt zu.

3.4.3 Intelligente Heuristik

Ist der Zielobjekt in Sicht, verhält sich diese Heuristik wie die einfache Heuristik. Ist das Zielobjekt dagegen nicht in Sicht, wird versucht, anderen Agenten auszuweichen, um ein möglichst breit gestreutes Netz aus Agenten aufzubauen. In der Implementation heißt das, dass unter allen Richtungen, in denen kein anderer Agent gesichtet wurde, eine Richtung zufällig ausgewählt wird und falls alle Richtungen belegt (oder alle frei) sind, wird aus allen Richtungen eine zufällig ausgewählt wird. In Abbildung (3.4) sieht der Agent das

Programm 2 Berechnung der nächsten Bewegung im Fall der einfachen Heuristik

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

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

  }
```

Zielobjekt nicht und wählt deswegen eine Richtung, in der die Sensoren keine Agenten anzeigt, in diesem Fall Norden. Programm 3.4.3 zeigt den zugehörigen Quelltext.

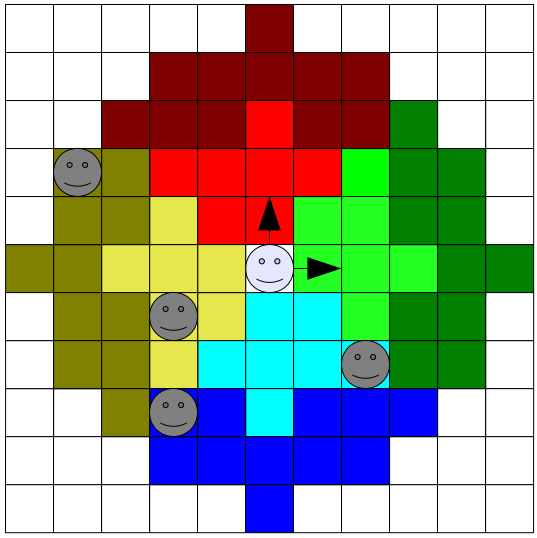

Abbildung 3.4: Agent mit intelligenter Heuristik: Falls das Zielobjekt nicht sichtbar ist bewegt sich der Agent von anderen Agenten weg.

Programm 3 Berechnung der nächsten Bewegung im Fall der intelligenten Heuristik

```
/**
 * Berechne nächste Aktion (intelligente Heuristik)
 */
private void calculateNextMove() {
  /**
   * Holt sich die Informationen der Gruppe der Sensoren, die auf
   * das Zielobjekt ausgerichtet sind
   */
    boolean[] goal_sensor = lastState.getSensorGoal();

    calculatedAction = -1;
    for(int i = 0; i < Action.MAX_DIRECTIONS; i++) {
    /**
     * Zielagent in Sicht in dieser Richtung?
     */
      if(goal_sensor[2*i]) {
        calculatedAction = i;
        break;
      }
    }

  /**
   * Zielobjekt nicht in Sicht? Dann bewege von Agenten weg
   */
    if(calculatedAction == -1) {
      calculatedAction = Misc.nextInt(Action.MAX_DIRECTIONS);

      boolean[] agent_sensors = lastState.getSensorAgent();
      boolean one_free = false;
      for(int i = 0; i < Action.MAX_DIRECTIONS; i++) {
        if(!agent_sensors[2*i]) {
          one_free = true;
          break;
        }
      }

      if(one_free) {
        while(agent_sensors[2*calculatedAction]) {
          calculatedAction = Misc.nextInt(Action.MAX_DIRECTIONS);
        }
      }
    }
  }
```

Kapitel 4

Das Zielobjekt

Die Typen von Zielobjekten werden zum einen über ihre Geschwindigkeit und zum anderen über ihre Bewegungsart definiert. Neben der Größe des Torus und den Hindernissen trägt der Typ des Zielobjekts wesentlich zur Schwierigkeit eines Szenarios bei, da dieser die Aufenthaltswahrscheinlichkeiten des Zielobjekts unter Einbeziehung des Zustands des letzten Zeitschritts bestimmt. Die Schwierigkeit bestimmt sich über die Summe der erwarteten Aufenthaltswahrscheinlichkeiten in nicht überwachten Feldern geteilt durch die Summe der Aufenthaltswahrscheinlichkeiten in überwachten Feldern.

4.1 Basiseigenschaften

Im wesentlichen entspricht ein Zielobjekt einem Agenten, d.h. das Zielobjekt kann sich bewegen und besitzt Sensoren.

TODO Zusammenhang

Gemeinsam haben alle Arten von Bewegungen des Zielobjekts, dass, wenn dem Algorithmus kein freies Feld zur Verfügung steht, ein zufälliges, freies Feld in der Nähe ausgewählt und dorthin gesprungen wird. Dies kommt einem Neustart gleich und ist notwendig um eine Verfälschung des Ergebnisses zu verhindern, das daher rühren kann, dass

ein oder mehrere Agenten (zusammen mit eventuellen Hindernissen) alle vier Bewegungsrichtungen des Zielobjekts blockieren. Andererseits ist auch der Sprung eine Verfälschung, falls bei einem Durchlauf eine ganze Anzahl von Sprüngen durchgeführt worden sein, sollte man deshalb das Ergebnis verwerfen und z.B. andere Random-Seed Werte oder einen anderen Algorithmus benutzen. Sofern nicht anders angegeben ist die Zahl solcher Sprünge jeweils unter 0.1% und wird ignoriert.

Das Zielobjekt kann sich außerdem in einem Schritt u.U. mehr als um ein Feld bewegen. Festgelegt wird die Bewegungsgeschwindigkeit über das Szenario (TODO Verweis?) und kann auch gebrochene Werte annehmen, wobei 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 Schritt führen. Die Auswertung der Bewegungsgeschwindigkeit ist relevant in Kapitel 5.2.

4.2 Typen von Zielobjekten

4.2.1 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.
Die Aufenthaltswahrscheinlichkeit für jedes freie Feld ist hierbei $\frac{1}{n}$, wobei n die Anzahl der freien Felder entspricht.

4.2.2 Typ "Zufällige Bewegung"

Wie "Zufälliger Algorithmus" bei Agenten (3.4.1). Sind alle möglichen Felder belegt, wird wie oben beschrieben auf ein zufälliges Feld gesprungen.
Diesen Fall außen vor gelassen beträgt die Aufenthaltswahrscheinlichkeit für die 4 angrenzenden Felder jeweils $\frac{1}{4}$.

4.2.3 Typ "Einfache Richtungsänderung"

Mit dieser Einstellung wird die der letzten Richtung entgegengesetzten Richtung aus der Menge der Auswahlmöglichkeiten entfernt und von den verbleibenden drei Richtungen eine zufällig ausgewählt. Sind alle drei Richtungen versperrt, wird stehengeblieben.
War die letzte Aktion nicht eine Bewegungsrichtung, sondern die Aktion "Stehenbleiben", so wird eine zufällige Richtung ausgewählt. Sind alle Richtungen versperrt, wird auch hier wieder auf ein zufälliges Feld gesprungen. Die Aufenthaltswahrscheinlichkeit beträgt im Fall ohne angrenzende Hindernisse für die Felder vor, links und rechts (relativ zur Bewegungsrichtung im vergangenen Zeitschritt) also $\frac{1}{3}$. In Abbildung 4.1 ist eine Beispielsituation zu sehen, bei der der Zielagent sich zuletzt nach Norden bewegt hat und nun zwischen Norden, Westen und Osten auswählen kann.

4.2.4 Typ "Beibehaltung der Richtung"

Der Zielobjekt versucht, immer Richtung Norden zu gehen. Ist das Zielfeld blockiert, wählt er 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.
Sind auch die Felder im Westen und Osten belegt, springt er auf ein zufälliges freies Feld. Schafft es der Zielobjekt innerhalb von einer bestimmten Zahl (Breite des Spielfelds) von

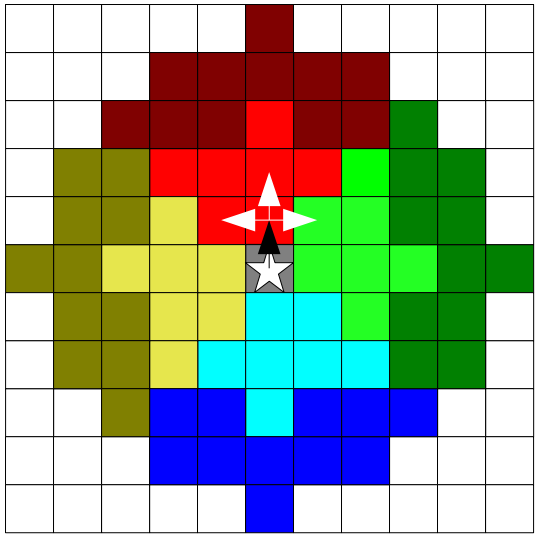

Abbildung 4.1: Zielobjekt macht pro Schritt maximal eine Richtungsänderung

Schritten nicht, einen weiteren Schritt nach Norden zu gehen, wird ebenfalls gesprungen, um ein "festhängen" an einem Hindernis zu vermeiden. Ohne Hindernisse ergibt sich also eine Aufenthaltswahscheinlichkeit von 1.0 im darüberliegenden Feld im Norden.
In Abbildung 4.2 sind drei Situationen dargestellt, zum einen ein wiederholtes hin- und herlaufen unter den Hindernissen, der Weg links um die Hindernisse herum und der Weg rechts um die Hindernisse herum.

4.2.5 Typ "Intelligent (Open)"

Das Zielobjekt versucht bei der Auswahl der Aktion möglichst die Aktion zu wählen, bei der es außerhalb der Sichtweite der Agenten bleibt, es werden also alle Richtungen gestrichen, in denen ein anderer Agent gesichtet wird. Von den verbleibenden Richtungen werden außerdem mit 20% Wahrscheinlichkeit alle Richtungen gestrichen, in denen sich ein Hindernis 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.

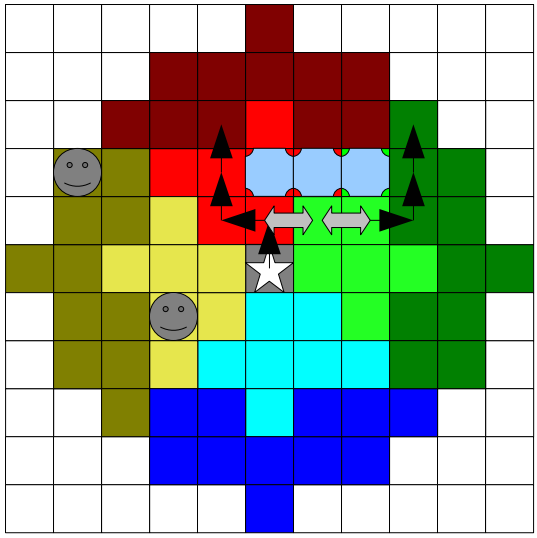

Abbildung 4.2: Bewegungsform “Beibehaltung der Richtung”: Zielobjekt bewegt sich, wenn möglich, immer nach Norden

In Abbildung 4.3 wird die Richtung Westen gestrichen, da sich dort ein Agent befindet. Im Norden und Osten befinden sich Hindernisse, diese Richtungen werden jeweils mit 20% gestrichen, während die Richtung Süden mit Sicherheit als Auswahlmöglichkeit übrig bleibt. Die Aufenthaltswahrscheinlichkeit für Norden und Osten wären also jeweils $\frac{8*8}{3*10*10} + \frac{2*8}{2*10*10} = \frac{88}{300}$ und $\frac{8*8}{3*10*10} + \frac{2*2*8}{2*10*10} + \frac{2*2}{10*10} = \frac{124}{300}$ für den Süden.

4.2.6 Typ “Intelligent (Hide)”

Das Zielobjekt vermeidet Agenten wie bei “Intelligent (Open)”, streicht aber statt Richtungen mit Hindernissen Richtungen ohne Hindernisse mit 20% Wahrscheinlichkeit, tendiert also eher dazu, auf Hindernisse zuzugehen. Idee ist, dass es sich dadurch möglicherweise den Blicken der Agenten entziehen kann.
Betrachten wir in Abbildung 4.4 die selbe Situation wie bei “Intelligent (Open)” so sind die Aufenthaltswahrscheinlichkeiten von Norden und Osten mit denen von Süden vertauscht.

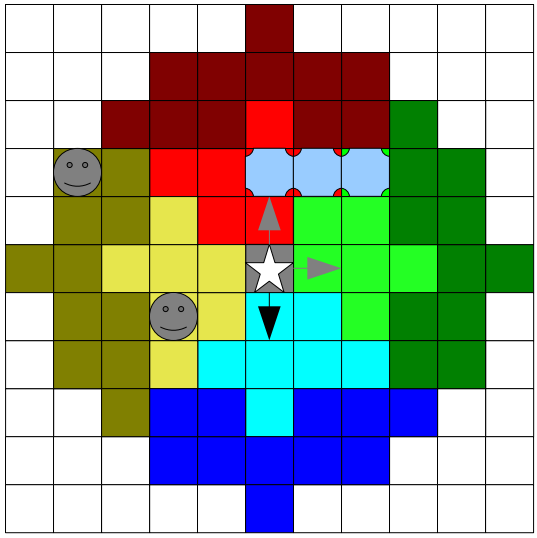

Abbildung 4.3: Zielobjekt bewegt sich mit bestimmter Wahrscheinlichkeit von Agenten und größerer Wahrscheinlichkeit von Hindernissen weg

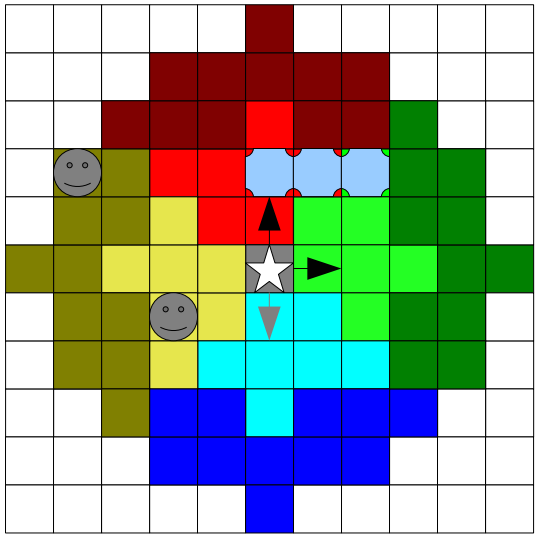

Abbildung 4.4: Zielobjekt bewegt sich von Agenten weg und mit bestimmter Wahrscheinlichkeit auf Hindernisse zu

4.2.7 Typ "SXCS"

Dieser Typ ist eine Implementierung für das Zielobjekt, das auf der SXCS Implementierung in Kapitel 9 basiert. Einziger Unterschied ist in der Art, wie das SXCS die eigenen Aktionen bewertet. Während das dort beschriebene SXCS die Nähe zum Zielobjekt belohnt, soll hier das Zielobjekt die Situationen positiv bewerten, bei denen sich keine Agenten in Überwachungsreichweite befinden. Eine genaue Beschreibung folgt im Kapitel 9, hier sollte die Idee nur der Vollständigkeit halber erwähnt werden.

TODO Pendelbewegung? TODO Fester Pfad? TODO Bezeichnungen englisch/deutsch

Kapitel 5

Ablauf der Simulation

5.1 Hauptschleife

In der Hauptschleife (Programm 5.1) wird ein Experiment mit vorgegebener Konfiguration ("*Configuration*") durchgeführt. Dabei werden eine Anzahl von Problemen abgearbeitet, bei denen jeweils der Torus auf den Startzustand gesetzt, das eigentliche Problem berechnet und ein neuer *random seed* Wert gesetzt wird.

5.2 Reihenfolge der Ausführung (*doOneMultiStepProblem()*)

TODO vielleicht erst noch eine allgemeine Übersicht geben

Für die Berechnung eines einzelnen Problems ("*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

Programm 4 Zentrale Schleife für einzelne Experimente

```
/**
 * Führt eine Anzahl von Problemen aus
 * @param experiment_nr Nummer des auszuführenden Experiments
 */
  public void doOneMultiStepExperiment(int experiment_nr) {
    int currentTimestep = 0;

  /**
   * number of problems for the same population
   */
    for (int i = 0; i < Configuration.getNumberOfProblems(); i++) {

    /**
     * creates a new grid and deploys agents and goal at random positions
     */
      BaseAgent.grid.resetState();

    /**
     * Führe Problem aus und aktualisiere aktuellen Zeitschritt
     */
      currentTimestep = doOneMultiStepProblem(currentTimestep);

    /**
     * Initialisiere neuen "Random Seed" Wert
     */
      Misc.initSeed(Configuration.getRandomSeed() +
        experiment_nr * Configuration.getNumberOfProblems() + 1 + i);
    }
  }
```

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. Und da 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.
Ansonsten gibt es folgende 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 und sich sofort für eine neue Aktion entschieden.

Bei der ersten Möglichkeit haben alle Agenten die Sensordaten vom Beginn der Zeiteinheit, während bei der zweiten Möglichkeit später verarbeitete Agenten bereits die Aktionen der bereits berechneten Agenten miteinbeziehen können. Umgekehrt können dann frühere Agenten bessere Positionen früher besetzen. Da aufgrund der primitiven Sensoren nicht davon auszugehen ist, dass Agenten beginnende Bewegungen (und somit deren jeweilige Zielposition) anderer Agenten einbeziehen können, soll jeder Agent von den Sensorinformationen zu Beginn der Zeiteinheit ausgehen.
Wenn sich mehrere Agenten auf dasselbe Feld bewegen wollen, dann spielt die Reihenfolge der Ausführung der Aktionen eine Rolle. Wird die Liste der Agenten einfach linear abgearbeitet, können Agenten mit niedriger Position in der Liste die Aktion auf Basis jüngerer

Sensordaten fällen. Dies kann dazu führen, dass Aktionen von Agenten mit höherer Position in der Liste eher fehlschlagen, da das als frei angenommene Feld nun bereits besetzt ist. Da es keinen Grund gibt, Agenten mit niedrigerer Position zu bevorteilen, werden die Aktionen der Agenten in zufälliger Reihenfolge abgearbeitet.
Bezüglich der Bewegung ergibt sich hierbei eine weitere Frage, nämlich wie unterschiedliche Bewegungsgeschwindigkeiten behandelt werden sollen, da alle Agenten eine Einheitsgeschwindigkeit von maximal einem Feld pro Zeiteinheit haben, während sich das Zielobjekt je nach Szenario gleich eine ganze Anzahl von Feldern bewegen kann (siehe auch Kapitel 4.1). 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, so dass das Zielobjekt in einer Zeiteinheit mehrmals (n-mal) neue Sensordaten erfassen und sich für eine neue Aktion entscheiden kann.

5.3 Messung der Qualität

Eine konkrete Antwort kann man auf diese zwei Fragen nicht geben, sie hängt davon ab, was man denn nun eigentlich erreichen möchte, also auf welche Weise die Qualität des Algorithmus bewertet wird. Der naheliegendste Messzeitpunkt ist, nachdem sich alle Agenten bewegt haben. Da wir die Agenten und das Zielobjekt in einem Durchlauf gemeinsam nacheinander bewegen, stellt sich die Frage nicht, ob wir womöglich vor der Bewegung des Zielobjekts die Qualität messen sollen. Eine Messung nach der Bewegung des Zielobjekts würde diesem erlauben, sich vor jeder Messung optimal zu positionieren, was in einer geringeren Qualität für den Algorithmus resultiert, da sich das Zielobjekt aus der Überwachungsreichweite anderer Agenten hinausbewegen kann. Letztlich ist es eine

Frage der Problemstellung, denn eine Messung nach Bewegung des Zielobjekts bedeutet letztlich, dass ein Agent einen gerade aus seiner Überwachungsreichweite heraus laufenden Zielobjekts in diesem Schritt nicht mehr überwachen kann.
Da ein wesentlicher Bestandteil die Kooperation (und somit die Abdeckung des Torus anstatt dem Verfolgen des Zielobjekts) sein soll, soll ein Bewertungskriterium sein, inwieweit der Einfluss des Zielobjekts minimiert werden soll. Auch findet, wenn man vom realistischen Fall ausgeht, die Bewegung des Zielobjekts gleichzeitig mit allen anderen Agenten statt. Die Qualität wird somit nach der Bewegung des Zielobjekts gemessen. Die Überlegung unterstreicht auch nochmal, dass es besser ist, das Zielobjekt insgesamt wie einen normalen (aber sich mehrmals bewegenden) Agenten zu behandeln.

5.4 Reihenfolge der Ermittlung des *base reward*

Keine der bisher vorgestellten Varianten machen Gebrauch von einem sogenannten *base reward*, d.h.

Schließlich bleibt die Frage danach, wann geprüft werden soll, ob das Zielobjekt in Überwachungsreichweite ist, und wann sich somit ein *reward* ergeben soll. Wesentliche Punkte hierbei sind, dass der Algorithmus sich anhand der Sensordaten selbst bewertet und pro Zeitschritt die Sensordaten nur einmal erhoben werden. Letzteres folgt aus der Auslegung von XCS, der in der Standardimplementation darauf ausgelegt ist, dass der Reward jeweils genau einer Aktion zugeordnet ist. Daraus ergibt sich auch, dass der Reward von binärer Natur ist ("Zielobjekt in Überwachungsreichweite" oder "Zielobjekt nicht in Überwachungsreichweite"), weshalb Zwischenzustände für den Reward, der sich aus der mehrfachen Bewegung des Zielagenten ergeben könnte, ausgeschlossen werden soll (z.B. "War zwei von drei Schritten in der Überwachungsreichweite" $\Rightarrow \frac{2}{3}$ Reward). Insbesondere würde dies eine mehrfache Erhebung der Sensordaten erfordern.

TODO neue rewarderhebung... TODO Rewarderhebung für normale Agenten irrelevant, evtl teilen

Für den Reward gibt es somit folgende Möglichkeiten:

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

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

Werden die *reward* Werte sofort ermittelt (Punkt 1), dann bezieht sich der Wert auf die veralteten Sensordaten vor der Aktion, die Aktion selbst würde bei der Ermittlung des *reward* Werts also ignoriert werden. Bei Punkt 2 müsste man bis zum neuen Zeitschritt warten, bis neue Sensordaten ermittelt wurden.

5.5 Zusammenfassung

Zusammenfassend sieht der Ablauf aller Agenten (inklusive des Zielobjekts) also wie folgt aus:

1. Bestimmen der aktuellen **Qualität**
2. Erfassung aller **Sensordaten**
3. Bestimmung der jeweiligen ***reward* Werte** für die einzelnen Objekte für den letzten Schritt
4. **Wahl der Aktion** anhand der Regeln des jeweiligen Agenten
5. **Ausführung der Aktion** (in zufälliger Reihenfolge, das Zielobjekt wiederholt Schritte 1 und 2 nach der Ausführung der Aktion)

5.6 Implementierung eines Problemablaufs

In der Schleife der Funktion zur Berechnung eines Experiments (Programm 5.1) wird die Funktion zur Berechnung des Problems (*doOneMultiStepProblem()*, Programm 5.6) 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 5.6), der *reward* Wert ermittelt (*rewardAgents()*, siehe Programm 5.6) und schließlich die Objekte bewegt (*moveAgents()*, siehe Programm 5.6). Die konkrete Umsetzung der dort aufgerufenen Funktionen (insbesondere *calculateNextMove()* und *calculateReward()*) wird im Kapitel 9 erläutert (bzw. in Kapitel 3 was die Heuristiken betrifft, wobei *calculateReward()* dort keine Rolle spielt und eine leere Funktion aufgerufen wird).

Programm 5 Zentrale Schleife für einzelne Probleme

```
/**
 * Führt eine Anzahl von Schritten auf dem aktuellen Torus aus
 * @param stepCounter Aktuelle Zeitschritt
 * @return Der Zeitschritt nach der Ausführung
 */
 private int doOneMultiStepProblem(int stepCounter) {
 /**
  * Zeitpunkt bis zu dem das Problem ausgeführt wird
  */
   int steps_next_problem =
     Configuration.getNumberOfSteps() + stepCounter;
   for (int currentTimestep = stepCounter;
        currentTimestep < steps_next_problem; currentTimestep++) {

   /**
    * Ermittle die Sichtbarkeit und erhebe Statistiken
    */
     BaseAgent.grid.updateSight();
     BaseAgent.grid.updateStatistics(currentTimestep);

   /**
    * Ermittle neue Sensordaten und berechne Aktionen der Agenten
    */
     calculateAgents(currentTimestep);

   /**
    * Ermittle den Reward für alle Agenten (nach dem ersten Schritt)
    */
     if(currentTimestep > stepCounter) {
       rewardAgents(currentTimestep);
     }

   /**
    * Führe zuvor berechnete Aktionen aus
    */
     moveAgents();
   }

   /**
    * Abschließende Ermittlung des Rewards
    */
   BaseAgent.grid.updateSight();
   rewardAgents(steps_next_problem);
   return steps_next_problem;
 }
```

Programm 6 Zentrale Bearbeitung (Sensordaten und Berechnung der neuen Aktion) aller Agenten und des Zielobjekts innerhalb eines Problems

```
/**
 * Berechnet die Aktionen und führt sie in zufälliger Reihenfolge aus
 * @param gaTimestep der aktuelle Zeitschritt
 */
  private void calculateAgents(final long gaTimestep) {

  /**
   * Ermittle Sensordaten und bestimme nächste Bewegung
   */
    for(BaseAgent a : agentList) {
      a.aquireNewSensorData();
      a.calculateNextMove(gaTimestep);
    }
    BaseAgent.goalAgent.aquireNewSensorData();
    BaseAgent.goalAgent.calculateNextMove(gaTimestep);
  }
```

Programm 7 Zentrale Bearbeitung (Verteilung des Rewards) aller Agenten und des Zielobjekts innerhalb eines Problems

```
/**
 * Rewards all agents
 */
  private void rewardAgents(final long gaTimestep) {
    for(BaseAgent a : agentList) {
      a.calculateReward(gaTimestep);
    }
    BaseAgent.goalAgent.calculateReward(gaTimestep);
  }
```

Programm 8 Zentrale Bearbeitung (Ausführung der Bewegung) aller Agenten und des Zielobjekts innerhalb eines Problems

```
private void moveAgents(long gaTimestep) {
/**
 * Erstelle Ausführungsliste für alle Objekte (Zielobjekt mehrfach)
 */
  int goal_speed = Configuration.getGoalAgentMovementSpeed();
  ArrayList<BaseAgent> random_list =
    new ArrayList<BaseAgent>(agentList.size() + goal_speed);

  random_list.addAll(agentList);
  for(int i = 0; i < goal_speed; i++) {
    random_list.add(BaseAgent.goalAgent);
  }

/**
 * Führe die ermittelten Aktionen in zufälliger Reihenfolge aus
 *   (Zielobjekt kann mehrfach ausgeführt werden).
 */
  int[] array = Misc.getRandomArray(random_list.size());
  for(int i = 0; i < array.length; i++) {
    BaseAgent a = random_list.get(array[i]);
    a.doNextMove();
    if(a.isGoalAgent() && goal_speed > 1) {
      goal_speed--;
      a.aquireNewSensorData();
      a.calculateNextMove(gaTimestep);
      a.calculateReward(gaTimestep);
    }
  }
}
```

Kapitel 6

Erste Analyse der Agenten ohne LCS

In diesem Kapitel sollen erste Analysen bezüglich der verwendeten Szenarien anhand des "zufälligen Algorithmus" 3.4.1, des Algorithmus mit "einfacher Heuristik" 3.4.2 und des Algorithmus mit "intelligenter Heuristik" 3.4.3 angefertigt werden. Die Ergebnisse aus der Analyse werden eine Grundlage für die vergleichende Betrachtung der Agenten mit LCS Algorithmen in Kapitel 10 dienen, insbesondere werden sie Anhaltspunkte dafür geben, welche Szenarien welche Eigenschaften der Algorithmen testen.

TODO: Ziel: Schwere Szenarien finden (schwierig für zufälligen, leicht für einfache heuristik)

6.1 Statistische Merkmale

Da keiner der hier vorgestellten Algorithmen lernt und somit statische Regeln besitzt, ist es nicht notwendig, die Qualitäten der Algorithmen bei verschiedener Anzahl von Zeitschritten zu betrachten und zu vergleichen, die Zahl der Zeitschritte wird somit auf 500 festgesetzt. Außerdem sollen in den Statistiken die Werte jeweils über einen Lauf von 10 Experimenten mit jeweils 10 Problemen ermittelt und gemittelt werden. Alle Werte sind auf 2 Stellen nach dem Komma gerundet. TODO

6.1.1 Qualität

6.1.2 Abdeckung

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

6.1.3 Blockierte Bewegungen

Der Wert der blockierten Bewegungen entspricht dem Anteil an Bewegungen, die insgesamt (Zielagent und Bewegungen gegen vom Zielagenten besetzte Felder ausgeschlossen) blockiert waren und somit nicht ausgeführt wurden. Der Wert ist ein

6.2 “Total Random” Zielobjekt

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

6.2.1 Ohne Hindernisse

Ohne Hindernisse gibt sich ein klares Bild (siehe 6.1), die intelligente Heuristik ist etwas besser als der des zufälligen Agenten und der einfachen Heuristik. Ein möglichst

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

Die einfache Heuristik schneidet dagegen etwas schlechter als eine zufällige Bewegung ab. Zwar ist die Zahl der blockierten Bewegungen geringer, was sich dadurch erklären lässt, dass die einfache Heuristik zumindest an einem Punkt eine Sichtbarkeitsüberprüfung für die Richtung durchführt, in der sie sich bewegen möchte (nämlich wenn das Zielobjekt in Sicht ist), andererseits ist die Abdeckung etwas geringer. Dies kommt daher, dass, wenn mehrere Agenten das Zielobjekt in der selben Richtung in Sichtweite haben, mehrere Agenten sich in die selbe Richtung bewegen. Dies beeinträchtigt die zufällige Verteilung der Agenten auf dem Spielfeld und führt somit auch zu einer niedrigeren Abdeckung des Torus.

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

Tabelle 6.1: "Total Random" ohne Hindernisse

Algorithmus	Agentenzahl	Blockierte Bewegungen	Abdeckung	Qualität
Zufällige Bewegung	8	2.84%	73.74%	32.43%
Einfache Heuristik	8	2.81%	73.21%	32.12%
Intelligente Heuristik	8	0.63%	81.26%	36.01%
Zufällige Bewegung	12	4.34%	69.49%	44.81%
Einfache Heuristik	12	4.17%	68.89%	43.90%
Intelligente Heuristik	12	1.50%	77.57%	49.56%
Zufällige Bewegung	16	5.81%	64.27%	54.57%
Einfache Heuristik	16	5.67%	63.62%	54.05%
Intelligente Heuristik	16	2.85%	71.43%	60.73%

6.2.2 Säulenszenario

6.2.3 Zufällig verteilte Hindernisse

Hier ergeben sich bei allen Einstellungen für λ_h und λ_p (siehe Kapitel 2.6.1) ebenfalls ein klares Bild (siehe 6.2), die intelligente Heuristik liegt wieder vorne, gefolgt wieder von der einfachen Heuristik und der zufälligen Bewegung. Die einfache Heuristik schneidet minimal besser ab als die zufällige Bewegung, die Zahl der blockierten Bewegungen scheint hier stärker ins Gewicht zu fallen. Insbesondere die intelligente Heuristik scheint Probleme mit den Hindernissen zu haben. Da Hindernisse in der Heuristik nicht beachtet werden, erzeugt die maximale Ausbreitung der Agenten einen Bewegungsdruck gegen sie.

Tabelle 6.2: "Total Random" mit Hindernisse (12 Agenten)

Algorithmus	λ_h	λ_p	Blockierte Bewegungen	Abdeckung	Qualität
Zufällige Bewegung	0.2	0.99	14.25%	57.93%	46.89%
Einfache Heuristik	0.2	0.99	11.96%	57.94%	46.99%
Intelligente Heuristik	0.2	0.99	17.76%	63.17%	51.39%
Zufällige Bewegung	0.1	0.99	9.11%	64.04%	45.75%
Einfache Heuristik	0.1	0.99	7.76%	63.82%	45.34%
Intelligente Heuristik	0.1	0.99	9.68%	70.50%	50.40%
Zufällige Bewegung	0.1	0.5	11.89%	61.82%	44.62%
Einfache Heuristik	0.1	0.5	10.27%	62.20%	44.32%
Intelligente Heuristik	0.1	0.5	12.21%	68.96%	49.16%

Agenten. Der wesentliche zweite Faktor ist hier, dass der einfache Agent, wenn er das Zielobjekt in Sicht hat, davon ausgehen kann, dass sich in dieser Richtung wahrscheinlich kein Hindernis befindet, während der zufällige Agent Hindernisse überhaupt nicht beachtet, somit öfters gegen ein Hindernis läuft und letztlich öfters stehen bleibt. Der Unterschied zwischen beiden Agenten ist besonders hoch in Szenarien mit größerem Anteil an Hindernissen.

Ansonsten liegt der intelligente Agent wieder eindeutig vorne, beherrscht aber besonders gut Szenarien mit hohem "Verknüpfungsfaktor" (1.0) der geringem Anteil an Hindernissen (0.1), bei denen er bis zu etwa 15% über dem Ergebnis des einfachen Agenten liegt.
Dies liegt daran, dass Szenarien mit hohem "Verknüpfungsfaktors" bedeuten, dass alle Hindernisse zusammenhängend einen großen Block bilden und somit dem Szenario ohne Hindernissen ähnlich sind, auf dem dieser Agent ja besonders gut abschneidet. In zerklüftete Szenarien hat der Algorithmus dagegen Schwierigkeiten um andere Agenten überhaupt zu Gesicht bekommen, der Vorteil der Verteilung fällt also zu einem Teil weg.

Dies bestätigt auch ein Durchlauf bei dem Behinderungen der Sicht durch Hindernisse deaktiviert sind. Hierbei erreicht der intelligente Agent im Szenario (0.4, 0.1) statt TODO evtl weg

TODO

6.3 "Zufälliger Nachbar" und "Einfache Richtungsänderung"

Wesentlicher Punkt bei beiden Bewegungstypen (siehe 4.2.2, 4.2.3) ist, dass der jetzige Ort des Zielobjekts maximal zwei Felder (die Standardgeschwindigkeit des Zielobjekts in den Tests) vom Ort in der vorangegangenen Zeiteinheit entfernt ist. Somit ist ein lokales Einfangen eher von Relevanz, wenn auch das Zielobjekt grundsätzlich schneller als andere Agenten ist.
Wesentlicher Unterschied zwischen beiden Bewegungstypen ist, dass das Zielobjekt mit Bewegungstyp "zufälliger Nachbar" bei einer Bewegungsgeschwindigkeit von 2 mit einer Wahrscheinlichkeit von $\frac{1}{4}$ auf das ursprüngliche Feld zurückkehrt, innerhalb eines Zeitschritts also stehenbleibt. Wie die Ergebnisse in Tabellen 6.4 und 6.5 zeigen (TODO

vielleicht noch näher darauf eingehen), ergibt sich dadurch ein leichteres Szenario. Ein mitunter stehenbleibender Agent kann mittels Heuristiken leichter überwacht werden, während es keine signifikante Veränderung bei der zufälligen Bewegung ergibt. In weiteren Tests soll deswegen immer nur die Bewegungsform "Einfache Richtungsänderung" getestet werden.

Tabelle 6.3: Vergleich "Zufälliger Nachbar" und "Einfache Richtungsänderung" (12 Agenten, ohne Hindernisse)

Algorithmus	Blockierte Bewegungen	Abdeckung	Qualität
"Zufälliger Nachbar"			
Zufällige Bewegung	4.26%	69.41%	45.75%
Einfache Heuristik	8.24%	61.77%	80.32%
Intelligente Heuristik	5.20%	70.09%	84.20%
"Einfache Richtungsänderung"			
Zufällige Bewegung	4.23%	69.63%	48.79%
Einfache Heuristik	7.20%	62.71%	69.78%
Intelligente Heuristik	4.07%	71.24%	74.53%

Tabelle 6.4: Vergleich "Zufälliger Nachbar" und "Einfache Richtungsänderung" (12 Agenten, zufälliges Szenario mit $\lambda_h = 0.2$, $\lambda_p = 0.99$)

Algorithmus	Blockierte Bewegungen	Abdeckung	Qualität
"Zufälliger Nachbar"			
Zufällige Bewegung	14.56%	57.80%	47.32%
Einfache Heuristik	18.29%	51.22%	85.92%
Intelligente Heuristik	22.33%	57.06%	88.31%
"Einfache Richtungsänderung"			
Zufällige Bewegung	14.60%	57.78%	47.92%
Einfache Heuristik	17.11%	52.38%	78.39%
Intelligente Heuristik	21.54%	57.76%	82.31%

Tabelle 6.5: Vergleich "Zufälliger Nachbar" und "Einfache Richtungsänderung" (12 Agenten, Säulenszenario)

Algorithmus	Blockierte Bewegungen	Abdeckung	Qualität
"Zufälliger Nachbar"			
Zufällige Bewegung	5.94%	67.75%	43.37%
Einfache Heuristik	10.41%	59.85%	85.61%
Intelligente Heuristik	7.82%	68.10%	88.98%
"Einfache Richtungsänderung"			
Zufällige Bewegung	6.02%	67.60%	45.83%
Einfache Heuristik	9.54%	60.34%	76.05%
Intelligente Heuristik	6.90%	68.75%	81.28%

6.4 "Intelligent Open" und "Intelligent Hide"

6.6 6.7 10.2

TODO: Erläuterung!

Zu beachten sei, dass im Fall von "Intelligent Hide" eine relativ große Nummer an Sprüngen des Zielobjekts (siehe Kapitel 4.1) stattgefunden hat, was die Ergebnisse etwas verzerrt, die Zahl hält sich aber noch in Grenzen (bis zu ca. 0.5% im Fall der einfachen und intelligenten Heuristik im Fall mit vielen Hindernissen).

Tabelle 6.6: Vergleich "Intelligent Open" und "Intelligent Hide" (8 Agenten, ohne Hindernisse)

Algorithmus	Abdeckung	Qualität
"Intelligent Open"		
Zufällige Bewegung	74.15%	11.32%
Einfache Heuristik	60.90%	82.86%
Intelligente Heuristik	69.62%	85.74%
"Intelligent Hide"		
Zufällige Bewegung	74.13%	12.26%
Einfache Heuristik	69.43%	55.31%
Intelligente Heuristik	74.87%	64.41%

Tabelle 6.7: Vergleich "Intelligent Open" und "Intelligent Hide" (8 Agenten, zufälliges Szenario mit $\lambda_h = 0.2$, $\lambda_p = 0.99$)

Algorithmus	Abdeckung	Qualität
"Intelligent Open"		
Zufällige Bewegung	62.54%	13.37%
Einfache Heuristik	52.23%	84.33%
Intelligente Heuristik	56.92%	85.12%
"Intelligent Hide"		
Zufällige Bewegung	62.52%	13.10%
Einfache Heuristik	50.17%	90.32%
Intelligente Heuristik	56.94%	90.45%

Tabelle 6.8: Vergleich "Intelligent (Open)" und "Intelligent (Hide)" (8 Agenten, Säulenszenario)

Algorithmus	Abdeckung	Qualität
"Intelligent (Open)"		
Zufällige Bewegung	72.55%	11.58%
Einfache Heuristik	57.19%	85.58%
Intelligente Heuristik	64.26%	91.18%
"Intelligent (Hide)"		
Zufällige Bewegung	72.56%	11.78%
Einfache Heuristik	58.45%	80.98%
Intelligente Heuristik	65.65%	86.38%

6.5 Always Same Direction

TODO

leeres Szenario

6.6 LCS

Wird weiter unten besprochen.

6.7 Zusammenfassung

Wie wir gesehen haben gibt es also Szenarien in denen Abdeckung kaum eine Rolle spielt und lokale Entscheidungen eine wesentliche Rolle spielen. Dies wird es erleichtern, geeignete Szenarien im Kapitel 11 “Kommunikation” zu finden.

Kapitel 7

LCS

7.1 Einführung

Jeder Agent besitzt ein sogennantes *XCS Classifier System* welches einem speziellen *learning classifier system* (LCS) entspricht. Ein LCS ist ein evolutionäres Lernsystem, das aus einer Reihe von *classifier* Regeln besteht (siehe Kapitel 7.2). Eine allgemeine Einführung in LCS findet sich z.B. in [11].

Eine wesentliche Erweiterung des LCS ist das sogenannte "accuracy-based classifier system XCS", zuerst beschrieben in [12]. Neben neuer Mechanismen zur Generierung neuer *classifier* (insbesondere im Bereich bei der Anwendung des genetischen Operators) ist im Vergleich zum einfachen LCS der wesentliche Unterschied, auf welche Weise der *fitness* Wert berechnet wird. Während der *fitness* Wert beim einfachen LCS lediglich auf dem *reward prediction error* Wert basierte, basiert bei XCS der *fitness* Wert auf der Genauigkeit der jeweiligen Regel. Eine genaue Beschreibung findet sich in [10].

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 (z.B. in [10]), bei dem mit 2 Adressbits und 4 Datenbits das den Adressbits entsprechende ausgegeben werden soll. Hier ist offensichtlich, dass nach der Klassifizierung sofort bestimmbar ist, ob sie ein korrektes Ergebnis geliefert hat.
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 wiederholt. Bei den Wiederholungen erhält jede Regel einen Teil der Bewertung des folgenden *classifiers*. 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.

Die in dieser Arbeit verwendete Implementierung entspricht im Wesentlichen der Standardimplementation des Multistep-Verfahrens von [3] (mit der algorithmischen Beschreibung des Algorithmus in [1]), eine Besonderheit stellt allerdings die Problemdefinition dar, da es kein festes Ziel gibt und somit auch keinen Neustart des Problems.

Die meisten Implementationen und Varianten von XCS beschäftigen sich mit Szenarios, bei denen das Ziel in einer statischen Umgebung gefunden werden muss. Häufiger Gegenstand der Untersuchung in der Literatur sind insbesondere relativ einfache Probleme 6-Multiplexer Problem und Maze1 (z.B. in [10] [15] [16]), während XCS mit Problemen größerer Schrittzahl zwischen Start und Ziel Probleme hat [14] [13]. Zwar gibt es Ansätze um auch schwierigere Probleme besser in den Griff zu bekommen (z.B. Maze5, Maze6,

Woods14 in [7]), indem ein Gradientenabstieg in XCS implementiert wurde. Ein konkreter Bezug zu einem dynamischen Überwachungsszenario konnte jedoch in keiner dieser Arbeiten gefunden werden.

Bezüglich Multiagentensystemen und XCS gibt es hauptsächlich Arbeiten, die auf auf zentraler Steuerung bzw. *OCS* [8] basieren, also im Gegensatz zum Gegenstand dieser Arbeit auf eine übergeordnete Organisationseinheit bzw. auf globale Regeln oder globalem Regeltausch zwischen den Agenten zurückgreifen.
Arbeiten bezüglich Multiagentensysteme in Verbindung mit LCS im Allgemeinen finden sich z.B. in [21], wobei es auch dort zentrale Agenten gibt, mit deren Hilfe die Zusammenarbeit koordiniert werden soll, während in dieser Arbeit alle Agenten die selbe Rolle spielen sollen.

Vielversprechend war der Titel der Arbeit [18], "Generation of Rule-based Adaptive Strategies for a Collaborative Virtual Simulation Environment". Leider wird in der Arbeit nicht diskutiert, auf was sich der kollaborative Anteil bezog, da nicht mehrere Agenten benutzt worden sind. Auch konnte dort jeder einzelne Schritt mittels einer *reward* Funktion bewertet werden, da es globale Information gab. Dies vereinfacht ein solches Problem deutlich und macht einen Vergleich schwierig.

Eine weitere Arbeit in dieser Richtung ([9]) beschreibt das "El Farol" Bar Problem, welches dort mit Hilfe eines Multiagenten XCS System erfolgreich gelöst wurde. Die Vergleichbarkeit ist hier auch eingeschränkt, da es sich bei dem EFBP um ein "Single-Step" Problem handelt.

Eine der dieser Arbeit (bezüglich Multiagentensysteme) am nächsten kommende Pro-

blemstellung wurde in [22] vorgestellt. Dort wurde der *base reward* unter den (zwei) Agenten aufgeteilt, es fand also eine Kommunikation des *reward* Werts statt. Wie das Ergebnis in Verbindung mit den Ergebnissen dieser Arbeit interpretiert werden kann, wird in 11 diskutiert.

In [20] wurde gezeigt, dass bei der Weitergabe des *base reward* Gruppenbildung von entscheidender Wichtigkeit ist. Nach bestimmten Kriterien werden Agenten in Gruppen zusammengefasst und der *base reward* anstatt an alle, jeweils nur an die jeweiligen Gruppenmitgliedern weitergegeben. Dies bestätigen auch Tests in Kapitel 11 bei der sich Agenten mit ähnelnden (was das Verhalten gegenüber anderen Agenten betrifft) *classifier sets* in Gruppen zusammengefasst wurden und zum Teil bessere Ergebnisse erzielt werden konnten als ohne Kommunikation.

7.2 Übersicht

TODO base reward und reward unterscheiden

Ein XCS ist ein regelbasiertes evolutionäres Lernsystem, das im Wesentlichen aus folgenden Elementen besteht:

1. Einer Menge an Regeln, sogenannte *classifier* (siehe 7.3
2. Ein Mechanismus zur Auswahl der *classifier* (siehe 8.9
3. Einem Mechanismus zur Evolution der *classifier* (mittels genetischer Operatoren, siehe 7.6)
4. Eine Mechanismus zur Bewertung der *classifier* (mittels *reinforcement learning*, siehe 7.8)

Während die ersten drei Punkten bei allen hier vorgestellten XCS Varianten identisch sind, gibt es wesentliche Unterschiede bei der Bewertung der *classifier*. Diese werden gesondert in Kapitel 9 im Einzelnen besprochen. Im Folgenden sollen nun die ersten drei Punkte näher betrachtet werden.

7.3 Classifier

Ein *classifier* besteht aus einem *condition* Vektor, einem *action*, einem *fitness*, einem *reward prediction*, einem *reward prediction error*, einem *experience* und einem *actionSetSize* Wert. Initialisiert werden diese Teile wie in 8 aufgelistet.

7.3.1 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 3.2 besprochen wurden.

7.3.2 Der *fitness* Wert

Der *fitness* Wert soll die allgemeine Genauigkeit des *classifier* repräsentieren und wird über die Zeit hinweg sukzessive an die beobachteten *reward* Werte angepasst. Der Wertebereich verläuft zwischen 0.0 und 1.0 (maximale Genauigkeit). Insbesondere eines der frühesten Werke zu XCS [2] beschäftigte sich mit diesem Aspekt der Genauigkeit.

7.3.3 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.

7.3.4 Der *reward prediction error* Wert

Der *reward prediction error* Wert soll die Genauigkeit des *classifier* bzgl. des *reward prediction* Werts (durchschnittliche Differenz zwischen *reward prediction* und tatsächlichem *reward*) repräsentieren. U.a. auf Basis dieses Werts wird der *fitness* Wert des *classifier* angepasst.

7.3.5 Der *condition* Vektor

Der *condition* Vektor gibt die Kondition an, in welcher Situation der zugehörige *classifier* ausgewählt werden kann, d.h. welche Sensordaten von dem jeweiligen *classifier* erkannt werden. Alle *classifier* die einen Sensordatensatz erkennen bilden das sogenannte *match-Set*. Der Aufbau des Vektors entspricht dem Vektor der über die Sensoren erstellt wird (siehe 3.1).

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

7.4 Platzhalter im *condition* Vektor

Neben den zu den Sensordaten korrespondierenden Werten 0 und 1 soll es noch einen dritten Zustand, den Platzhalter "#", geben, der anzeigen soll, dass beim Vergleich zwischen Kondition und Sensordaten diese Stelle ignoriert werden soll. Eine Stelle im *condition* Vektor mit Platzhalter gilt 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.

Beim Vergleich der Sensordaten und Daten aus dem *condition* Vektor werden immer jeweils zwei Paare verglichen. In 3.1 wurde erwähnt, dass der Fall (0/1) in den Sensordaten

nicht auftreten kann, weswegen (um die Aufgabe nicht unnötig zu erschweren) ein Datenpaar (0/1) im *condition* Vektor äquivalent zum Datenpaar (1/1) sein soll, es damit also eine gewisse Redundanz gibt.

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

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

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


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

7.5 Subsummation

Die Benutzung von Platzhaltern erlaubt es dem LCS mehrere *classifiers* zu subsummieren, wodurch die Gesamtzahl der *classifier* sinkt und somit Erfahrungen, die ein LCS Agent sammelt, nicht unbedingt mehrfach gemacht werden müssen. Die dahinter stehende Annahme ist, dass es Situationen gibt, in denen der Gewinn der durch Unterscheidung zwischen zwei verschiedenen Sensordatensätzen geringer ist als die Ersparnis durch das

Zusammenlegen beider *classifiers*, d.h. dem Ignorieren der Unterschiede.

Besitzt ein *classifier* zum einen einen genügend großen *experience* Wert und ausreichend kleinen *reward prediction error* Wert, so kann er als sogenannter *subsumer* auftreten. Andere *classifiers* 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 den selben Wert wie der zu subsummierende *classifier* oder einen Platzhalter besitzt.

7.6 Genetische Operatoren

Es werden aus den jeweiligen *actionSets* 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 dem selben *actionSet* stammen, ist der Wert beider Eltern identisch). Die restlichen Werte werden standardmäßig wie in Kapitel 8 aufgelistet initialisiert. Werden Kinder in die Population eingefügt, deren *action* Wert und *condition* Vektor identisch mit existierenden *classifiers* 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 7.3.5) des *condition* Vektors des ersten bzw. zweiten ausgewählten Elternteils, dann können für die drei Grup-

pen der *condition* Vektoren (z_{1k}, a_{1k}, h_{1k}) und (z_{2k}, a_{2k}, h_{2k}) der beiden Kinder folgende Kombinationen auftreten:

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

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

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

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

7.7 Der *numerosity* Wert

Durch Subsummation (siehe 7.5 und 7.6) 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. Durch die Benutzung von *macro classifiers* ergibt sich allerdings das programmiertechnische Problem, dass man nicht mehr direkt weiß, wieviele *micro classifiers* sich in einer Population befinden, bei jeder Benutzung des Werts der Populationsgröße müssten die *numerosity* Werte aller *classifiers* jedes Mal addiert werden. In der Standardimplementierung [3] 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 einem einzelnen *actionSet* optimiert, während die hier verwendete Implementierung jeweils mit bis über 100 *actionSets* programmiert wurde, denen ein *classifier* Mitglied sein kann. Deswegen wurde die Optimierung entfernt und durch eine dezentrale Verwaltung mit einem *Observer* ersetzt, jede Änderung des *numerosity* Wertes hat also die Änderung aller *actionSets* zur Folge, in der der *classifier* Mitglied ist.

Wird also z.B. ein *micro classifier* entfernt, dann wird lediglich die Änderungsfunktion des *classifiers* aufgerufen, der dann wiederum den *numerositySum* Wert der jeweiligen Eltern anpasst. Dies macht einige Optimierungen rückgängig, erspart aber sehr viel Umstände, den *numerositySum* der Eltern immer auf den aktuellen Stand zu halten und einzelne *classifiers* zu löschen.
Positiver Nebeneffekt durch die verbesserte Struktur, dass man dadurch leicht auf die Menge der *actionSets* zugreifen kann, denen ein *classifier* angehört, hierfür wurde aber keine Verwendung gefunden.

Ein weiteres Problem der Standardimplementierung ist, dass der *fitness* Wert eines *classifiers* als Optimierung bereits den *numerosity* Wert als Faktor enthält, während bei der Aktualisierung des *numerosity* Werts der *fitness* Wert nicht aktualisiert wurde. Das hat zur Folge, dass theoretisch *fitness* Werte von *classifiers* fast den *max population* Wert annehmen kann, wenn ein *classifier* mit *numerosity* und *fitness* Wert in der Höhe von *max population* auf einen *numerosity* Wert von 1 reduziert wird.
Dies betrifft die Funktion *public void addNumerosity(int num)* der Klasse *XClassifier* in der Datei *XClassifier.java*. Die korrigierte Fassung ist in 7.7 gelistet, ein Vergleich der Qualität, mit und ohne Korrektur, ist in **??** dargestellt.

TODO Vergleich! TODO wenig Unterschied sensoragent, evtl wieder raus

7.8 Bewertung

Programm 9 Korrigierte Version der *addNumerosity()* Funktion

```
/**
 * Adds to the numerosity of the classifier.
 * @param num The added numerosity (can be negative!).
 */
  public void addNumerosity(int num) {
    int old_num = numerosity;

    numerosity += num;

  /**
   * Korrektur der fitness
   */
    fitness = fitness * (double)numerosity / (double)old_num;

  /**
   * Aktualisierung der Eltern
   */
    for (ClassifierSet p : parents) {
      p.changeNumerositySum(num);
      if (numerosity == 0) {
        p.removeClassifier(this);
      }
    }
  }
```

Programm 10 Bestimmung des *base rewards* für Agenten

```
/**
 * @return true Falls das Zielobjekt von diesem Agenten überwacht wird
 *   und kein anderer Agent in dieser Richtung in
 *   Überwachungsreichweite steht
 */
  public boolean checkRewardPoints() {
    boolean[] sensor_agent = lastState.getSensorAgent();
    boolean[] sensor_goal = lastState.getSensorGoal();

    for(int i = 0; i < Action.MAX_DIRECTIONS; i++) {
      if((sensor_goal[2*i]) && (!sensor_agent[2*i+1])) {
        return true;
      }
    }

    return false;
  }
```

Kapitel 8

Parameter

Die Einstellungen der XCS Parameter der durchgeführten Experimente entsprechen weitgehend den Vorschlägen in [1] ("Commonly Used Parameter Settings"). Eine Auflistung findet sich in Tabelle 8.2. Im Folgenden sollen Parameter besprochen werden, die entweder in der Empfehlung offen gelassen sind, also klar vom jeweiligen Szenario abhängen, und solche, bei denen von der Empfehlung abgewichen wurde.
Mitunter führen andere Parametereinstellungen auch zu wesentlich besseren Ergebnissen. Dies muss man aber vorsichtig bewerten, wenn die erreichte Qualität unter der des zufälligen Algorithmus liegt, da eine Auswirkung sein kann, dass der Algorithmus nicht besser lernt, sondern sich umgekehrt eher wie der zufällige Algorithmus verhält. Ein Vergleich mit der Qualität des zufälligen Algorithmus wird deswegen jeweils immer angegeben.
Anzumerken sei, dass alle Tests jeweils mit den in Tabelle 8.2 angegebenen Parameterwerten durchgeführt wurde und bei jedem Test jeweils nur der zu untersuchende Wert verändert wurde. Um synchronisierte und vergleichbare Daten zu haben, wurden die Tests deshalb in mehreren Etappen durchgeführt, die angegebenen Testergebnisse entsprechen jeweils den endgültigen Ergebnissen.

8.1 Parameter *max population N*

Der Wert von *max population* bezeichnet die maximalen Größe der *classifier set* Liste. Ein größerer Wert verlängert die Laufzeit linear (siehe 8.2), ein kleinerer Wert erhöht die Konkurrenz zwischen den *classifiers*. Größere Werte erlauben eine bessere Anpassung, da weniger *classifier* während eines Laufs gelöscht werden müssen und mehr Plätze zur Speicherung der Erfahrungen zur Verfügung steht. Auf der anderen Seite werden mehr Schritte benötigt um die für die jeweiligen *classifier* ausreichend Erfahrung zu sammeln (siehe 8.3).

Für den Overhead (Benutzung des zufälligen Algorithmus ohne XCS) ergab sich eine mittlere Laufzeit von 1.77 Sekunden pro Experiment bei 500 Schritten (bzw. 6.65 Sekunden bei 2000 Schritten), was die anfängliche Stagnierung bis $N = 32$ erklärt. Die Tests liefen auf einem T7500, 2.2 GHz in einem einzelnen Thread.

In den Tests wird $N = 128$ gesetzt, was als ausreichender Kompromiss zwischen den erwähnten Faktoren erscheint.

und über 10 Probleme, gemittelt über 10 Experimente

8.1

8.2 Maximalwert *reward*

Der Wert der bei der Bewertung als *reward* vergeben wird hat lediglich ästhetische Auswirkungen und wurde auf 1.0 gesetzt. In der Standardimplementation von XCS (siehe 9.3) ist der maximale *reward* äquivalent mit dem Maximalwert von ρ, da das Problem bei jedem positiven *reward* Wert neugestartet wird, also entweder der *reward* Wert aus dem letzten Schritt also immer 0 ist oder *maxPrediction* auf 0 gesetzt wurde, und $\rho = reward + \gamma\ maxPrediction$ gilt.

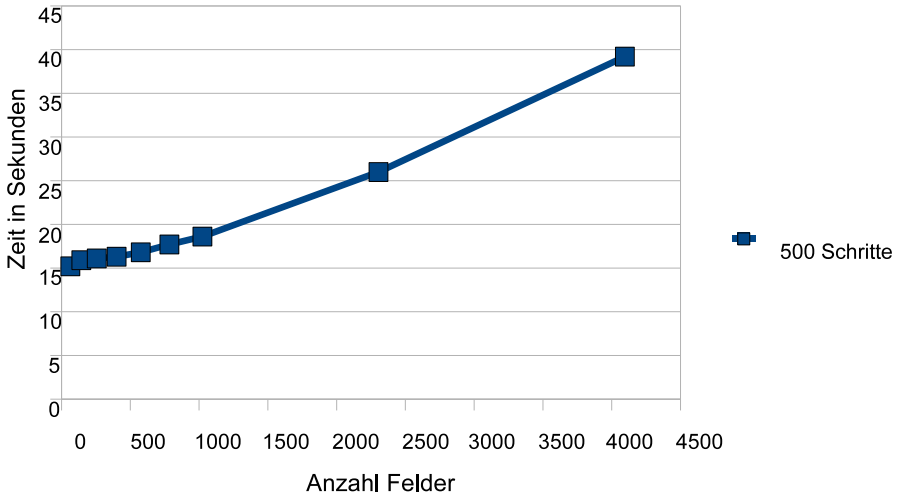

Abbildung 8.1: Darstellung der Auswirkung der Torusgröße auf die Laufzeit im leeren Szenario, zufälliger Bewegung des Zielobjekts, 8 sich zufällig bewegenden Agenten

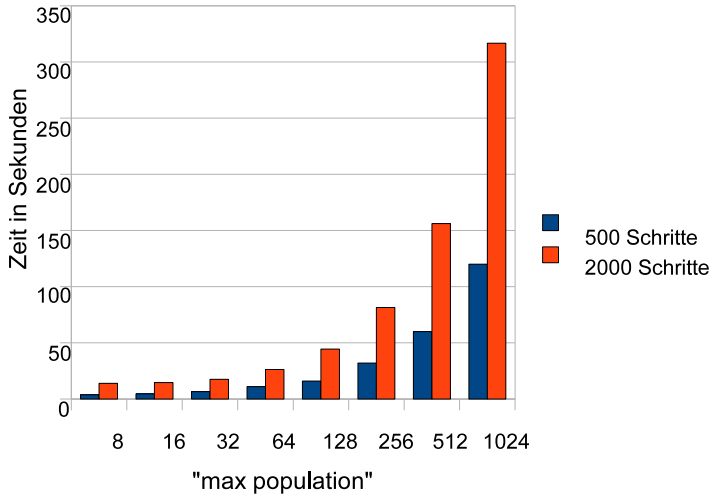

Abbildung 8.2: Darstellung der Auswirkung des Parameters *max population N* auf die Laufzeit im leeren Szenario, zufälliger Bewegung des Zielobjekts, 8 Agenten mit LCS Algorithmus

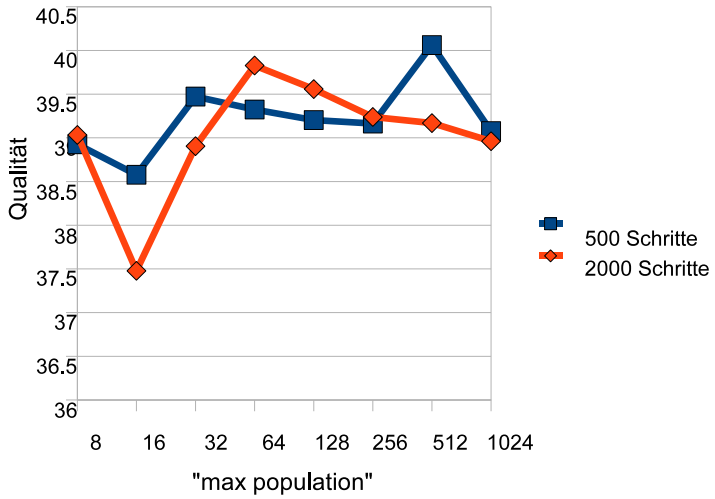

Abbildung 8.3: Darstellung der Auswirkung des Parameters *max population* N auf die Qualität im leeren Szenario, zufälliger Bewegung des Zielobjekts, 8 Agenten mit LCS Algorithmus

In den hier vorgestellten XCS Varianten wird dagegen der *reward* Wert absteigend, zusammen mit dem *maxPrediction* Wert, an frühere *actionSet* Listen verteilt, ρ kann also größer als 1.0 werden. In diesem Bereich ist noch Bedarf an theoretischer Forschung, in Tests haben sich Werte bis 3.0 ergeben, welche aber vom jeweiligen Szenario abhängen. Wird das Zielobjekt (z.B. wegen Hindernissen oder großen Torusdimensionen) eher selten gesehen, fällt der Wert geringer aus.

8.3 Parameter *accuracy equality* ϵ_0

Der Parameter ϵ_0 gibt an, unter welchem Wert zwei *accuracy* Werte als gleich gelten sollen. Dies ist insbesondere bei der *subsummation* Funktion und der Berechnung des *accuracy* Werts von Bedeutung. In der Literatur [1] wird als Regel genannt, dass der Wert auf etwa 1% des Maximalwerts von ρ gesetzt werden soll, den der erwartete Reward annehmen kann. Aufgrund der Überlegungen in 8.2 wird ϵ_0 für die neuen XCS Varianten

auf 0.02 gesetzt, während es für die Standardimplementation von XCS auf 0.01 gesetzt wird. Ein Testdurchlauf auf dem Säulenszenario (siehe Abbildung 8.4) ergibt aber, dass der Parameter keine besondere Auswirkung hat, weshalb der Wert auf 0.01 belassen wird.

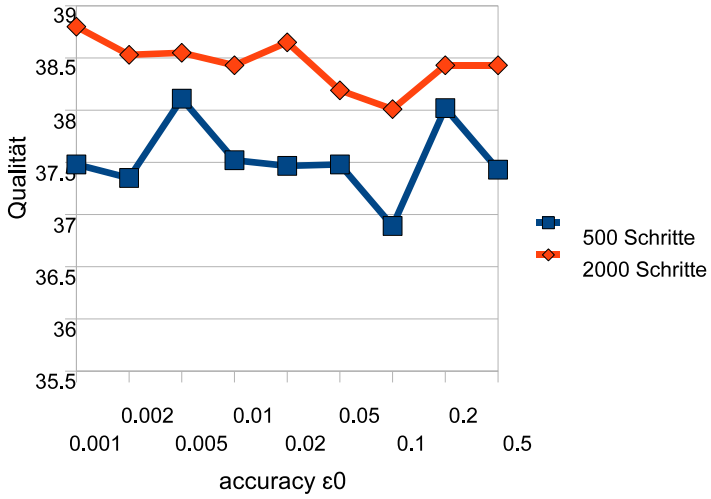

Abbildung 8.4: Auswirkung des Parameters *accuracy equality* ϵ_0 auf die Qualität im Säulenszenario, zufälliger Bewegung des Zielobjekts, 8 Agenten mit SXCS Algorithmus

8.4 Parameter *reward prediction discount* γ

TODO Abschnitt entfernen Auch für den Wert *reward prediction discount* γ hat sich ein etwas höherer Wert als sinnvoll erwiesen, als standardmäßig benutzt wird. Laut [1] hängt der Wert auch vom verwendeten Szenario ab. Ein höherer Wert für γ bedeutet, dass die Höhe des Werts, der über *maxPrediction* weitergegeben wird, mit zeitlichem Abstand zur ursprünglichen Bewertung mit einem *reward*, weniger schnell abfällt, wodurch eine längere Verkettung von *reward* Werten möglich ist. Umgekehrt führen zu hohe Werte für γ zu der positiven Bewertung von *classifiers* die am Erfolg gar nicht beteiligt waren, was sich negativ auf die Qualität auswirken kann.

Tabelle **??** zeigt einen Vergleich der Qualität mit dem Standardwert $\gamma = 0.71$ und

dem für die in dieser Arbeit verwendeten Testszenarien gewählten Wert $\gamma = 0.95$.

TODO 0.71 lassen

TODOTabelle prediction discount

8.5 Parameter Lernrate β

Für die Lernrate β hat sich ein etwas niedrigerer als in der Literatur angegebener Wert (0.01) als erfolgreich erwiesen. Die Lernrate bestimmt, wie stark ein ermittelter *reward* Wert den *reward prediction*, *reward prediction error*, *fitness* und *action set size* Wert pro Aktualisierung beeinflusst.TODO Auch dieser Parameter ist szenariospezifisch, über die konkrete Begründung kann nur spekuliert werden, die Schwierigkeit des Szenarios TODO

Vergleichende Tests (siehe Abbildung 8.5 mit niedrigerem bzw. höherem Wert haben zu einer etwas schlechteren Qualität geführt. TODO

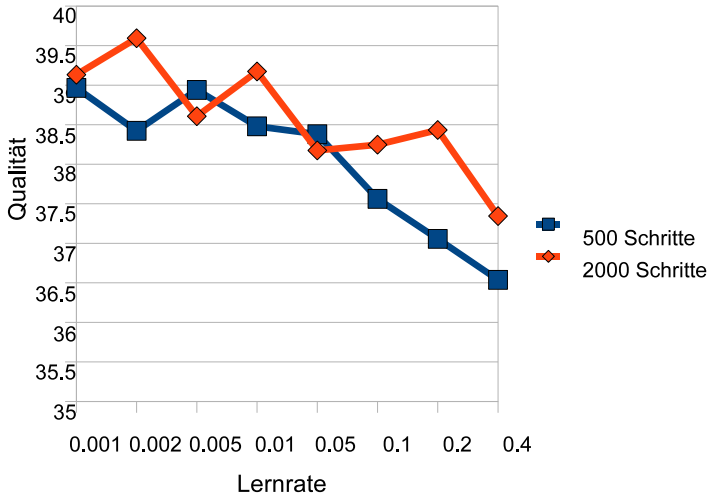

Abbildung 8.5: Auswirkung des Parameters *learning rate* β auf die Qualität im Säulenszenario, zufälliger Bewegung des Zielobjekts, 8 Agenten mit SXCS Algorithmus

8.6 Parameter *reward prediction init* p_i

In der Literatur werden Werte nahe Null bzw. 1% von ρ als Initialisierung für den *reward prediction* Wert eines *classifiers* angegeben. Wählt man einen Wert, der näher am Durchschnitt der *reward prediction* Werte der *classifier* liegt, die sich in den besten Lösungen am Ende eines Testdurchlaufs befinden, so ist zu erwarten, dass die Anzahl der benötigten Aktualisierungen des *reward prediction* Werts geringer ausfällt, das System also schneller konvergiert. Diese Überlegung wird bestätigt durch entsprechende Tests (siehe 8.6).
Wir setzen somit für SXCS den Parameter auf $p_i = 1.0$.
TODO Standardverfahren?

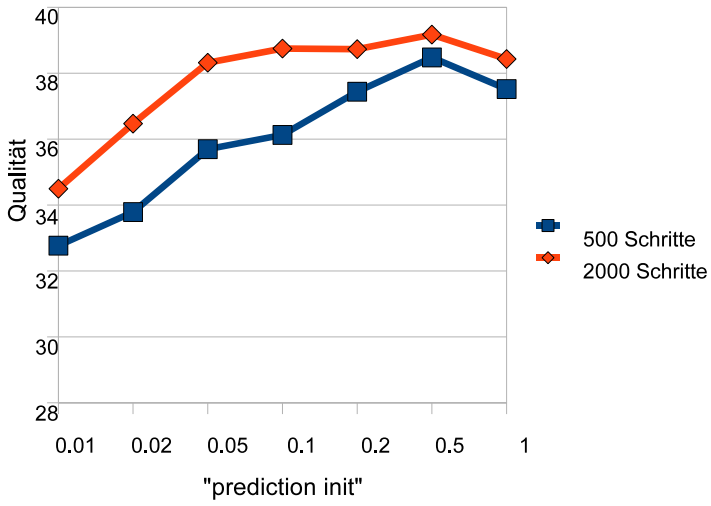

Abbildung 8.6: Darstellung Auswirkung des Parameters *reward prediction init* p_i auf die Qualität im Säulenszenario, zufälliger Bewegung des Zielobjekts, 8 Agenten mit SXCS Algorithmus

8.7 Zufällige Initialisierung

Normalerweise werden XCS Systeme mit leeren *classifier set* Listen initialisiert, als Option wird jedoch auch eine zufällige Initialisierung erwähnt ([10]), bei der zu Beginn die

classifier set Liste mit mehreren *classifiers* mit zufälligen *action* Werten und *condition* Vektoren gefüllt wird.

Tests haben gezeigt (siehe Tabelle 8.1), dass dadurch minimal bessere Ergebnisse erzielt werden, allerdings nur in Szenarien mit ausreichender Schrittzahl (> 100). Dies lässt sich darauf zurückführen, dass anfänglich gefüllte *classifier set* Listen die *matchSet* Listen relativ groß lassen werden, somit die Auswirkungen anfänglichen Lernens geringer ausfallen und sich die Agenten eher wie sich zufällig bewegende Agenten verhalten. Da hier durchgeführten Tests über 500 bzw. 2000 Schritte laufen, sollen somit die *classifier set* Listen mit zufällig generierten *classifiers* gefüllt werden.

Tabelle 8.1: Vergleichende Tests für den den Start mit und ohne zufällig gefüllten *classifier set* Listen

Algorithmus	Agentenzahl	Schrittzahl	Abdeckung	Qualität
Zufälliger Agent	8	500	60.64%	61.54%
Multistep	8	500	60.64%	61.54%
LCS	8	500	60.64%	61.54%
NewLCS	8	500	60.64%	61.54%
Zufälliges Szenario	8	500	60.64%	61.54%
Säulenszenario	8	100	1.0%	1.0%
LCS Ohne Drehung	8	2000	1.0%	1.0%
LCS Mit Drehung	8	100	1.0%	1.0%
LCS Mit Drehung	8	500	1.0%	1.0%
LCS Mit Drehung	8	2000	1.0%	1.0%
Zufälliger Agent	12	500	76.03%	76.59%
Einfacher Agent	12	500	67.30%	96.86%
Intelligenter Agent	12	500	86.85%	95.08%
LCS Ohne Drehung	12	100	1.0%	1.0%
LCS Ohne Drehung	12	500	1.0%	1.0%
LCS Ohne Drehung	12	2000	1.0%	1.0%
LCS Mit Drehung	12	100	1.0%	1.0%
LCS Mit Drehung	12	500	1.0%	1.0%
LCS Mit Drehung	12	2000	1.0%	1.0%

8.8 Übersicht über alle Parameterwerte

Tabelle 8.2: Verwendete Parameter (soweit nicht anders angegeben) und Standardparameter, TODO englisch/deutsch

Parameter	Wert	Standardwert [1]
Max population N	**128** (siehe 8.1)	
Max value ρ	**1.0** (siehe 8.2)	[10000]
Fraction mean fitness δ	0.1	[0.1]
Deletion threshold θ_{del}	20.0	[$\sim$ 20.0]
Subsumption threshold θ_{sub}	20.0	[20.0+]
Covering # probability $P_{\#}$	0.5	[$\sim$ 0.33]
GAthreshold θ_{GA}	25.0	[25-50]
Mutation probability μ	0.05	[0.01-0.05]
Prediction error reduction	0.25	[0.25]
Fitness reduction	0.1	[0.1]
Reward prediction init p_i	**0.5, 1.0** (siehe 8.6)	[$\sim$ 0]
Prediction error init ϵ_i	0.0	[0.0]
Fitness init F_i	0.01	[0.01]
Random init	**ja**	[ja oder nein]
Numerosity	1	[1]
Experience	0	[0]
Accuracy equality ϵ_0	**0.05**	[*1% des größten Werts*]
Accuracy calculation α	0.1	[0.1]
Accuracy power ν	5.0	[5.0]
Reward prediction discount γ	0.71	[0.71]
Learning rate β	**0.01** (siehe 8.5)	[0.1-0.2]
exploration probability	0.5 (siehe 7.5)	[$\sim$ 0.5]

8.9 Auswahlart der *classifier*

In jedem Zeitschritt gilt es zu entscheiden, welche Bewegung ein Agent ausführen soll. Als Basis der Entscheidung hat ein Agent zum einen die Sensordaten und zum anderen das eigene *classifier set* zur Verfügung. Da ein Sensordatensatz von mehreren *classifiers* erkannt werden kann (siehe 7.4), stellt sich die Frage, welchen *classifier* (und den dazugehörigen *action* Wert der die Bewegung bestimmt) man aus dem gebildeten *matchSet* auswählen soll. In der ursprünglichen Implementierung [3] wurden folgende Auswahlarten benutzt:

1. *random selection* : Zufällige Auswahl eines *classifiers*
2. *roulette wheel selection* : Zufällige Auswahl eines *classifier*, mit Wahrscheinlichkeit abhängig vom Produkt seines *fitness* und *reward prediction* Werts
3. *best selection* : Auswahl des *classifiers* mit dem höchsten Produkt aus seinen *fitness* und *reward prediction* Werten

Bei einem dynamischen Überwachungsszenario ist es im Vergleich zu standardmäßigen statischen Szenarien weder nötig noch hilfreich *random selection* zu nutzen. Die Idee für diese Auswahlart in einem statischen Szenario ist, dass man möchte, dass das XCS möglichst vielen verschiedenen Situationen ausgesetzt ist. Da in einem statischen Szenario Start- und Zielposition wie auch die Hindernisse fest sind, ist es wichtig, durch *random selection* dem XCS einen gewissen Spielraum zu geben.
Bei einem dynamischen Szenario ergibt sich dieses Problem nicht, andere Agenten und das Zielobjekt sind in stetiger Bewegung, der eigene Startpunkt ist nicht fixiert und das Problem wird bei Erreichen des Ziels nicht neugestartet. Aufgrund der Natur der Aufgabenstellung ist es in einem Überwachungsszenario außerdem wichtig, dass das XCS über eine längere Zeit hinweg eine gute Leistung liefert, also stetig gute Entscheidungen trifft,

eine zufällige Auswahl scheint also wenig hilfreich zu sein.
Außerdem wird in XCS zwischen den verschiedenen Auswahlarten hin und her geschalten. Die Auswahlarten werden in zwei Gruppen geteilt, in die sogenannte *explore* Phase und in die *exploit* Phase. In der *exploit* Phase soll bevorzugt eine Auswahlart ausgeführt werden, die das Produkt aus den Werten *fitness* und *reward prediction* möglichst stark gewichten, beispielsweise wäre *best selection* ein Kandidat für die *exploit* Phase, während z.B. *random selection* ein Kandidat für die *explore* Phase wäre.

Dies bestätigen später auch Tests TODO Tests

8.9.1 Auswahlart *tournament selection*

Zu den oben erwähnten drei Möglichkeiten wurde in [6] eine weitere vorgestellt und in Bezug auf XCS diskutiert, die sogenannte *tournament selection*. Als Vorteile werden geringerer Selektionsdruck, höhere Effizienz, geringerer Einfluss von Störungen, wie auch Flexibilität der Anpassung über zwei Parameter, k und p, genannt. Dabei werden k *classifier* aus dem *matchSet* zufällig ausgewählt, nach ihrem Produkt aus den jeweiligen *fitness* und *reward prediction* Werten sortiert und absteigend mit Wahrscheinlichkeit p der jeweilige *classifier* ausgewählt (d.h. der erste mit p, der zweite mit $(1.0 - p) * p$, der dritte mit $(1.0 - p)(1.0 - p) * p$ usw.). Mit $p = 1.0$ und $k = n$ (wobei n der Größe des *matchSets* entspricht) wäre *tournament selection* identisch mit *best selection* und mit $k = 1$ wäre es identisch mit *random selection*.
Bei der Entscheidung, welche Auswahlart jeweils für die *explore* und welche für die *exploit* Phase benutzt werden soll, ergeben sich also zwei Möglichkeiten:

1. *roulette wheel selection* : Zufällige Auswahl eines *classifier*, mit Wahrscheinlichkeit abhängig vom Produkt seines *fitness* und *reward prediction* Werts

2. *tournament selection* : Zufällige Wahl von k *classifiers* und daraus Wahl des jeweils

besten *classifiers* mit Wahrscheinlichkeit p, Wahl des zweitbesten mit Wahrscheinlichkeit $(1.0 - p) * p$ usw. TODO

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

In der Standardimplementierung von XCS wird zwischen jedem Problem zwischen der *explore* und der *exploit* Phase hin und hergeschaltet. Idee ist, dass man mit Hilfe der *explore* Phasen den Suchraum besser erforschen kann, dann aber zur eigentlichen Problemlösung in der *exploit* Phase möglichst direkt auf das Ziel zugeht.
Bei der Standardimplementierung für den statischen Fall ist allerdings das Erreichen eines positiven *base rewards* äquivalent mit einem Neustart des Problems. Während in der Standardimplementierung beim Neustart des Problems das gesamte Szenario (alle Agenten, Hindernisse und das Zielobjekt) auf den Startzustand zurückgesetzt wird, läuft das Überwachungsszenario weiter. Als erweiterten Ansatz soll nun deshalb eine neue Problemdefinition gelten, dass nicht das Erreichen eines positiven *base rewards* einen Phasenwechsel auslöst, sondern eine Änderung des *base rewards*, so dass mit anfänglicher *explore* Phase immer dann in die *exploit* Phase gewechselt wird, wenn das Zielobjekt in Sicht ist (bzw. umgekehrt, wenn mit der *exploit* Phase begonnen wird). Als Vergleich soll der andauernde, zufällige Wechsel zwischen der *explore* und *exploit* Phase, eine andauernde *exploit* und andauernde *explore* Phase dienen. Es sollen nun also folgende Arten des Wechsel zwischen den Phasen untersucht werden:

1. Andauernde *explore* Phase
2. Andauernde *exploit* Phase
3. Abwechselnd *explore* und *exploit* Phase (bei positivem *base reward*)
4. Abwechselnd *explore* und *exploit* Phase (bei Änderung des *base reward*, beginnend mit *explore*)

5. Abwechselnd *explore* und *exploit* Phase (bei Änderung des *base reward*, beginnend mit *exploit*)

6. In jedem Schritt zufällig entweder *explore* oder *exploit* Phase (50% Wahrscheinlichkeit jeweils)

1. Vergleich Random Explore, Roulette Wheel 2. Always Explore und Switch(exploit) schnell ausschliessen

4 verschiedene Wechsel, 2 verschiedene explore/exploit Dinger, mehrere Parametereinstellungen (p), k auf Maximum

=¿ 16-32 Tests

No exploration =¿ viele ungültige Bewegungen, nicht "wegkommen" von Hindernis / stehenbleiben?

TODO SEHR WICHTIG BEI SICH WENIG BEWEGENDEN ZIELEN

Die Wahl der Auswahlart für *classifier* in Punkt (3) (in Kapitel 9.2) kann auf verschiedene Weise erfolgen. In der Standardimplementierung von XCS wird zwischen "exploit" und "explore" nach jedem Erreichen des Ziels entweder umgeschalten oder zufällig mit einer bestimmten Wahrscheinlichkeit eine Auswahlart ermittelt. Es werden also abwechselnd ganze Probleme im "exploit" und "explore" Modus berechnet. Dies erscheint sinnvoll für die erwähnten Standardprobleme, da nach Erreichen des Ziels ein neues Problem gestartet wird und die Entscheidungen die während der Lösung eines Problems getroffen werden keine Auswirkungen auf die folgenden Probleme hat, die Probleme also nicht miteinander zusammenhängen.

Bei dem hier vorgestellten Überwachungsszenario kann 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 zu differenzieren, wie dies in den Standardszenarien bei XCS der Fall ist, bei denen 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 ihm andauernd ausweicht. Aus diesen Gründen erscheint es sinnvoll, weitere Formen des Wechsels zwischen diesen Phasen zu untersuchen:

Möglichkeit (3.) und (4.) entspricht dem Fall in der Standardimplementierung von XCS. Dabei wird bei jedem Erreichen eines positiven Rewards zwischen “explore” und “exploit” hin und hergeschaltet, was in der Standardimplementierung dem Beginn eines neuen Problems entspricht.

TODO Umschalten bei reward, Code evtl.

TODOTESTS

TODO SWITCH EXPLORE/EXPLOIT + NEW LCS sehr gut

Kapitel 9

XCS Varianten

9.1 Einführung

TODO!!

Ziel der Arbeit war es, wie man den XCS Algorithmus auf ein Überwachungsszenario anwenden kann. Notwendig dafür war es, die XCS Implementierung vollständig nachzuvollziehen, um für jeden Bestandteil entscheiden zu können, welche Rolle es bezüglich eines solchen Szenarios spielt. Für die Tests wurde nicht auf bestehende Pakete (z.B. XCSlib [17]) zurückgegriffen, wenn auch der Quelltext von [3] Modell stand.

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

Der hier entwickelte Algorithmus muss primär nicht einen Weg zum Ziel erkennen, sondern eine möglichst optimale (und auch an andere Agenten angepasste) Verhaltensstrategie finden.

In Kapitel 8 wurden mögliche Optimierungen zu den Parametern vorgestellt, in Kapitel 9.2 wurde diskutiert, in welcher Reihenfolge bei einem Multiagentensystem auf einem

diskreten Torus die einzelnen Teile ausgeführt werden sollen.

Besonders die Verwaltung der Numerosity und die Verwendung des maxPrediction bereitete

Das Multistepverfahren baut darauf auf, dass die Qualität der Agenten sich sukzessive mit jeder Probleminstanz verbessert, der Reward eben an immer weiter vom Ziel entfernte Aktionen TODO weitergereicht wird.

Da sich das Ziel schneller bewegt, kann eine einfache Verfolgungsstrategie nicht zum Erfolg führen. Eine einfache Implementation mit einem simplen Agenten der auf das Ziel zugeht, wenn es in Sicht ist und sich sonst wie ein sich zufällig bewegender Agent verhält, schneidet grundsätzlich schlechter ab.

TODO!

9.2 Ablauf eines XCS

1. Vervollständigung der *classifier* Liste (*covering*, siehe **??**)
2. Auswahl auf die Sensordaten passender *classifier* (*matching*, siehe 9.2.1)
3. Bestimmung der Auswahlart der Aktion (*explore/exploit*, siehe Kapitel 8.9)
4. Auswahl der Aktion TODO
5. Erstellung des zur Aktion zugehörigen Liste von *classifiers* (*actionSet*, siehe 9.2.2)

   , so dass es in der Liste *classifiers* deren

   TODO Bei der Auswahl einer Aktion werden alle *classifier* mit *condition* Vektoren gesucht, die auf die aktuellen Sensordaten passen. Diese bilden dann das *matchSet*.
6. Im nächsten Schritt wählen wir einen *classifier* aus diesem *matchset* aus und speichern dessen Aktion.

7. Schließlich bilden wir anhand des MatchSets und der gewählten Aktion das ActionSet

9.2.1 Variable *lastMatchSet*

In der *lastMatchSet* Variable werden jeweils alle *classifier* gespeichert, die den letzten Sensordatenvektor erkannt haben. Sie entspricht dem *predictionArray* in der originalen Implementierung von XCS, dort werden nämlich außerdem Vorberechnungen zur Auswahl des nächsten *classifier* für die Bewegung durchgeführt und die Ergebnisse gespeichert.

9.2.2 Variable *actionSet*

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

9.3 Standard XCS Multistepverfahren

Idee dieses Verfahrens ist, dass der *reward* Wert, den eine Aktion (bzw. das jeweils zugehörige *actionSet* und die dortigen *classifier*) erhält, vom erwarteten *reward* Wert der folgenden Aktion abhängt. Somit wird, rückführend vom letzten Schritt auf das Ziel, der *reward* Wert schrittweise an vorgehende Aktionen verteilt, mit der Annahme, dass dann, durch mehrfache Wiederholung des Lernprozesses, mit dem sich dadurch ergebenen Regelsatz mit höherer Wahrscheinlichkeit das Ziel gefunden wird.

Kern des Verfahrens ist die Vergabe des *base rewards*. Wird das Ziel erreicht, d.h.

erhält der Algorithmus einen positiven *base reward* Wert, so wird der *reward* 1.0 an das letzte *actionSet* gegeben. Liegt kein positiver *base reward* Wert vor, so wird lediglich der für diesen Schritt erwartete *reward* Wert (nämlich der *maxPrediction* Wert) an das letzte *actionSet* gegeben.

Als Vergleich wurde das bekannte Verfahren [3] fast unverändert übernommen. Der wesentliche Unterschied ist, dass das Szenario bei einem positiven *base reward* nicht neugestartet wird, algorithmisch ist die Implementierung ansonsten identisch. Außerdem, wie schon in Kapitel 8.9.2 erwähnt, soll die Qualität des Algorithmus nicht nur in der *exploit* Phase gemessen werden, da ein fortlaufendes Problem und kein statisches Szenario betrachtet wird. Schließlich gibt es, neben den Parametereinstellungen im Kapitel 8, feste Schnittpunkte für das *two point crossover* beim genetischen Operator (siehe 7.6).

9.4 XCS Variante für Überwachungsszenarien (SXCS)

Die Hypothese bei der Aufstellung dieser Variante des XCS-Algorithmus ist im Grunde dieselbe wie beim XCS-Multistepverfahren, nämlich dass die Kombination mehrerer Aktionen zum Ziel führt. Beim Multistepverfahren besteht die wesentliche Verbindung zwischen den *actionSet* Listen jeweils nur zwischen zwei direkt aufeinanderfolgenden *actionSets* über den *maxPrediction* Wert. In einer statischen Umgebung kann dadurch über mehrere (identische) Probleme hinweg eine optimale Einstellung (der *fitness* und der *reward prediction* Wert) für die *classifier* gefunden werden.
Bei der veränderten XCS Variante SXCS soll die Verbindung zwischen den *actionSets* zusätzlich direkt durch die zeitliche Nähe zum Ziel gegeben sein. Es wird in jedem Schritt

Programm 11 Erstes Kernstück des Standard XCS Multistepverfahrens (*calculateReward()*, Bestimmung des Rewards anhand der Sensordaten), angepasst an ein dynamisches Überwachungsszenario

```
/**
 * Diese Funktion wird in jedem Schritt aufgerufen um den aktuellen
 * Reward zu bestimmen, den besten Wert des ermittelten MatchSets
 * weiterzugeben und, bei aktuell positivem Reward, das aktuelle
 * ActionSet zu belohnen.
 *
 * @param gaTimestep Der aktuelle Zeitschritt
 */

  public void calculateReward(final long gaTimestep) {
  /**
   * checkRewardPoints liefert "wahr" wenn sich der Zielagent in
   * Überwachungsreichweite befindet
   */
    boolean reward = checkRewardPoints();

    if(prevActionSet != null){
      collectReward(lastReward, lastMatchSet.getBestValue(), false);
      prevActionSet.evolutionaryAlgorithm(classifierSet, gaTimestep);
    }

    if(reward) {
      collectReward(reward, 0.0, true);
      lastActionSet.evolutionaryAlgorithm(classifierSet, gaTimestep);
      prevActionSet = null;
      return;
    }
    prevActionSet = lastActionSet;
    lastReward = reward;
  }
```

Programm 12 Zweites Kernstück des Multistepverfahrens (*collectReward()* - Verteilung des Rewards auf die ActionSets), angepasst an ein dynamisches Überwachungsszenario

```
/**
 * Diese Funktion verarbeitet den übergebenen Reward und gibt ihn an die
 * zugehörigen ActionSets weiter.
 *
 * @param reward Wahr wenn der Zielagent in Sicht war.
 * @param best_value Bester Wert des vorangegangenen ActionSets
 * @param is_event Wahr wenn diese Funktion wegen eines Ereignisses, d.h.
 *        einem positiven Reward, aufgerufen wurde
 */

 public void collectReward(boolean reward,
               double best_value, boolean is_event) {
   double corrected_reward = reward ? 1.0 : 0.0;

 /**
  * Falls der Reward von einem Ereignis rührt, aktualisiere das
  * aktuelle ActionSet und lösche das vorherige
  */
   if(is_event) {
     if(lastActionSet != null) {
       lastActionSet.updateReward(corrected_reward, best_value, factor);
       prevActionSet = null;
     }
   }

 /**
  * Kein Ereignis, also nur das letzte ActionSet aktualisieren
  */
   else
   {
     if(prevActionSet != null) {
       prevActionSet.updateReward(corrected_reward, best_value, factor);
     }
   }
 }
```

Programm 13 Drittes Kernstück des Multistepverfahrens (*calculateNextMove()* - Auswahl der nächsten Aktion und Ermittlung des zugehörigen ActionSets), angepasst an ein dynamisches Überwachungsszenario

```
/**
 * Bestimmt die zum letzten bekannten Status passenden Classifier und
 * wählt aus dieser Menge eine Aktion. Außerdem wird das aktuelle
 * ActionClassifierSet mithilfe der gewählten Aktion ermittelt.
 *
 * @param gaTimestep Der aktuelle Zeitschritt
 */

  public void calculateNextMove(long gaTimestep) {

 /**
  * Überdecke das classifierSet mit zum Status passenden Classifiern
  * welche insgesamt alle möglichen Aktionen abdecken.
  */
    classifierSet.coverAllValidActions(
                    lastState, getPosition(), gaTimestep);

 /**
  * Bestimme alle zum Status passenden Classifier.
  */
    lastMatchSet = new AppliedClassifierSet(lastState, classifierSet);

 /**
  * Entscheide auf welche Weise die Aktion ausgewählt werden soll.
  */
    lastExplore = checkIfExplore(lastState.getSensorGoalAgent(),
                                          lastExplore, gaTimestep);

 /**
  * Wähle Aktion und bestimme zugehöriges ActionSet
  */
    calculatedAction = lastMatchSet.chooseAbsoluteDirection(lastExplore);
    lastActionSet = new ActionClassifierSet(lastState, lastMatchSet,
                                                        calculatedAction);
  }
```

das jeweilige *actionSet* gespeichert und aufgehoben, bis ein neues Ereignis (siehe Kapitel 9.4.1) eintritt und dann in Abhängigkeit des Alters mit einem entsprechenden *reward* Wert aktualisiert.
$r(a)$ bezeichnet den *reward* Wert für das *actionSet* mit Alter a.

Bei linearer Vergabe des *reward*:

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

bzw. bei quadratischer Vergabe des *reward*:

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

In Tests ergab sich für die quadratische Vergabe des *reward* ein minimal besseres Ergebnis (TODO zeigen), weitere Grafiken werden auf die lineare Vergabe des *reward* beschränkt sein um eine verständliche Darstellung zu ermöglichen, während in den Simulationen die quadratische Vergabe des *reward* benutzt wird.

9.4.1 Ereignisse

In XCS wird lediglich das jeweils letzte ActionSet aus dem vorherigen Zeitschritt gespeichert, in der neuen Implementierung werden dagegen eine ganze Anzahl (bis zu "maxStackSize") von ActionSets gespeichert. Die Speicherung erlaubt zum einen eine Vorverarbeitung des Rewards anhand der vergangenen Zeitschritte und auf Basis einer größeren Zahl von ActionSets und zum anderen die zeitliche Relativierung eines ActionSets zu einem Ereignis. Die Classifier wird dann jeweils rückwirkend anhand des Rewards aktualisiert sobald bestimmte Bedingungen eingetreten sind.

Von einem positiven bzw. negativen Ereignis spricht man, wenn sich der Reward im

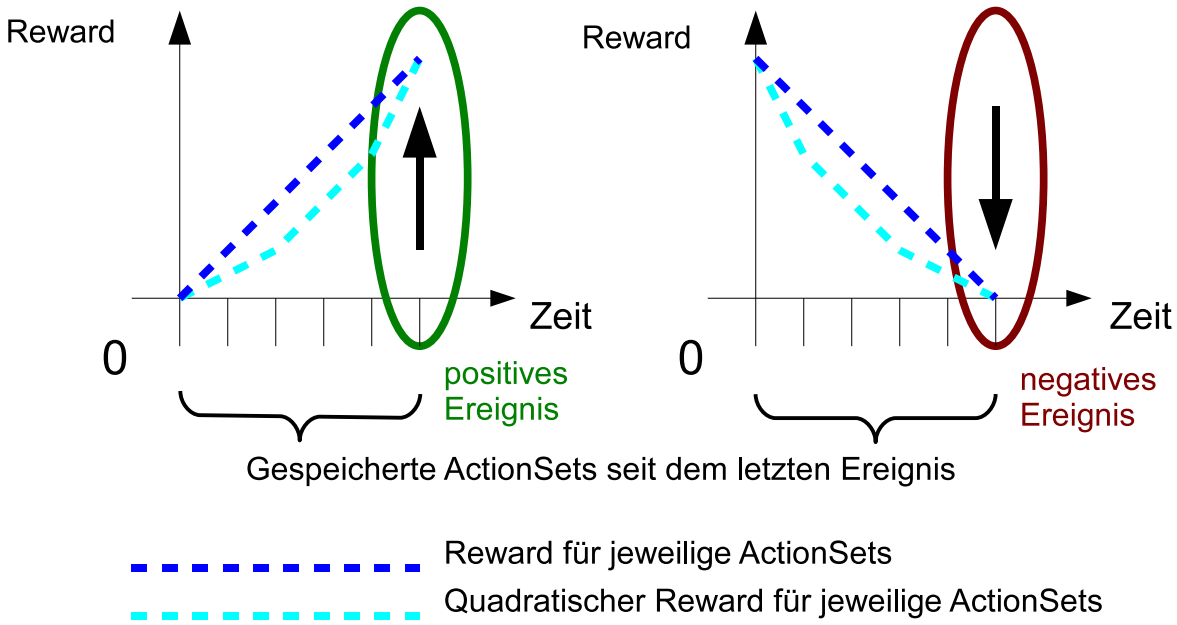

Abbildung 9.1: Schematische Darstellung der (quadratischen) Rewardverteilung an gespeicherte ActionSets bei einem positiven bzw. negativen Ereignis

Vergleich zum vorangegangenen Zeitschritt verändert hat, also wenn der Zielagent sich in Übertragungsreichweite bzw. aus ihr heraus bewegt hat (siehe (9.2)).

Bei der Benutzung eines solchen Stacks entsteht eine Zeitverzögerung, d.h. die Classifier besitzen jeweils Information die bis zu "maxStackSize" Schritte zu alt sind. Wählen wir den Stack zu groß, nimmt die Konvergenzgeschwindigkeit und Reaktionsfähigkeit des Systems zu stark ab, wählen wir ihn zu klein, kann es sein, dass wir einen Überlauf bekommen, also "maxStackSize" Schritte lang keine Rewardänderung aufgetreten ist. Im letzteren Fall brechen wir deswegen ab, bewerten die ActionSets der ersten Hälfte des Stacks (also die $\frac{maxStackSize}{2}$ ältesten Einträge) mit dem damals vergebenem konstanten Reward (welcher dem aktuellen Reward entspricht, es ist ja keine Rewardänderung eingetreten) und nehmen sie vom Stack (siehe (9.3)). Anschließend wird normal weiter verfahren bis der Stack wieder voll ist bzw. bis eine Rewardänderung auftritt. Das Szenario mit dem maximalen Fehler wäre das, bei dem ein Schritt nach des Abbruchs eine

Rewardänderung auftritt. “maxStackSize” stellt also einen Kompromiss zwischen Zeitverzögerung bzw. Reaktionsgeschwindigkeit und Genauigkeit dar.

Ein Ereignis tritt auf, wenn:

1. Positive Rewardänderung (Zielagent war im letzten Zeitschritt nicht in Überwachungsreichweite) ⇒ positives Ereignis (mit reward = 1)
2. Negative Rewardänderung (Zielagent war im letzten Zeitschritt in Überwachungsreichweite) ⇒ negatives Ereignis (mit reward = 0)
3. Überlauf des Stacks (keine Rewardänderung in den letzten “maxStackSize” Schritten), Zielagent ist in Überwachungsreichweite ⇒ neutrales Ereignis (mit reward = 1)
4. Überlauf des Stacks (keine Rewardänderung in den letzten “maxStackSize” Schritten), Zielagent ist nicht in Überwachungsreichweite ⇒ neutrales Ereignis (mit reward = 0)

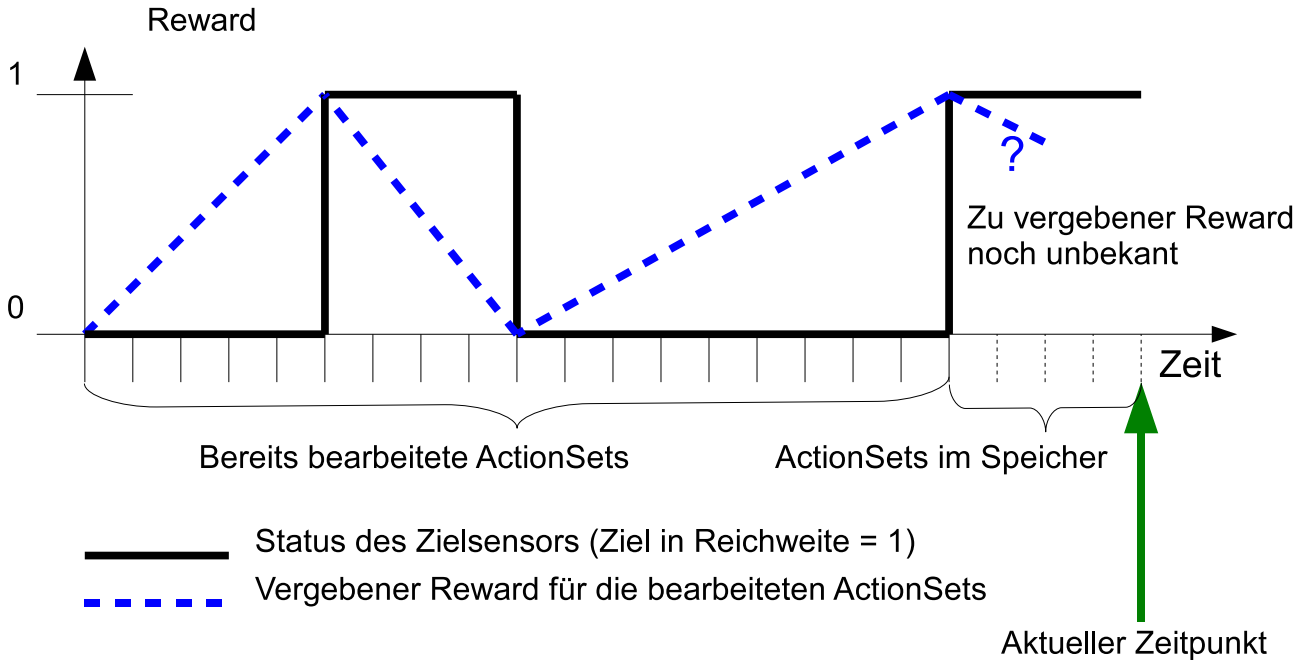

Abbildung 9.2: Schematische Darstellung der zeitlichen Rewardverteilung an ActionSets nach mehreren positiven und negativen Ereignissen und der Speicherung der letzten ActionSets

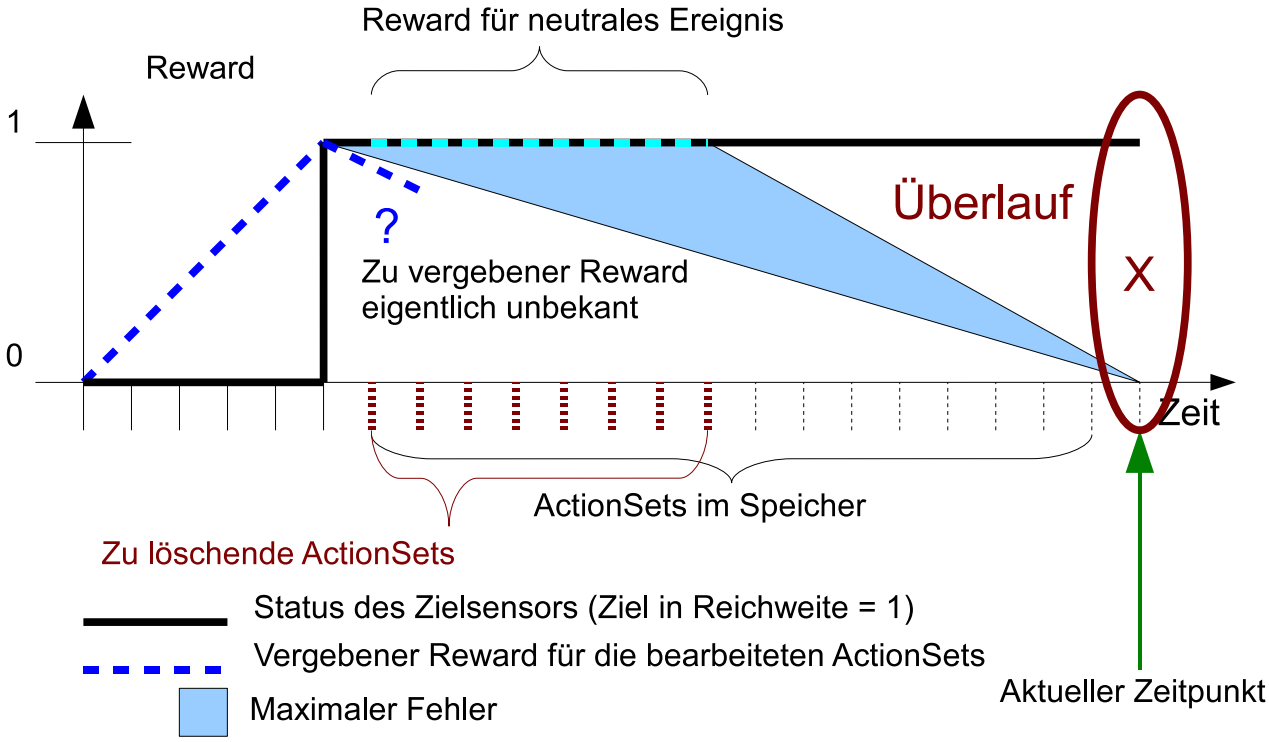

Abbildung 9.3: Schematische Darstellung der Rewardverteilung an ActionSets bei einem neutralen Ereignis

9.4.2 Implementierung von SXCS

TODO Erläuterung

Programm 14 Erstes Kernstück des SXCS-Algorithmus (*calculateReward()*, Bestimmung des Rewards anhand der Sensordaten)

```
/**
 * Diese Funktion wird in jedem Schritt aufgerufen um den aktuellen
 * Reward zu bestimmen und positive, negative und neutrale Ereignisse
 * den besten Wert des ermittelten MatchSets weiterzugeben und, bei
 * aktuell positivem Reward, das aktuelle ActionSet zu belohnen.
 *
 * @param gaTimestep Der aktuelle Zeitschritt
 */

  public void calculateReward(final long gaTimestep) {
  /**
   * checkRewardPoints liefert "wahr" wenn sich der Zielagent in
   * Überwachungsreichweite befindet
   */
    boolean reward = checkRewardPoints();

    if (reward != lastReward) {
      int start_index = historicActionSet.size() - 1;
      collectReward(start_index, actionSetSize, reward, 1.0, true);
      actionSetSize = 0;
    }
    else

    if(actionSetSize >= Configuration.getMaxStackSize())
    {
      int start_index = Configuration.getMaxStackSize() / 2;
      int length = actionSetSize - start_index;
      collectReward(start_index, length, reward, 1.0, false);
      actionSetSize = start_index;
    }

    lastReward = reward;
  }
```

Programm 15 Zweites Kernstück des SXCS-Algorithmus (*collectReward()* - Verteilung des Rewards auf die ActionSets)

```
/**
 * Diese Funktion verarbeitet den übergebenen Reward und gibt ihn an die
 * zugehörigen ActionSets weiter.
 *
 * @param reward Wahr wenn der Zielagent in Sicht war.
 * @param best_value Bester Wert des vorangegangenen ActionSets
 * @param is_event Wahr wenn diese Funktion wegen eines Ereignisses, d.h.
 *        einem positiven Reward, aufgerufen wurde
 */

 public void collectReward(
              boolean reward, double best_value, boolean is_event) {
   double corrected_reward = reward ? 1.0 : 0.0;
 /**
  * Wenn es kein Event ist, dann gebe den Reward weiter wie beim
  * Multistepverfahren
  */
   double max_prediction = is_event ? 0.0 :
     historicActionSet.get(start_index+1).getMatchSet().getBestValue();

 /**
  * Aktualisiere eine ganze Anzahl von ActionSets
  */
   for(int i = 0; i < action_set_size; i++) {

 /**
  * Benutze aufsteigenden bzw. absteigenden Reward bei einem positiven
  * bzw. negativen Ereignis
  */
     if(is_event) {
       corrected_reward = reward ?
         calculateReward(i, action_set_size) :
         calculateReward(action_set_size - i, action_set_size);
     }
 /**
  * Aktualisiere das ActionSet mit dem bestimmten Reward und
  * gebe bei allen anderen ActionSets den Reward weiter wie
  * beim Multistepverfahren
  */
     ActionClassifierSet action_classifier_set =
       historicActionSet.get(start_index - i);
     action_classifier_set.updateReward(
       corrected_reward, max_prediction, factor);

     max_prediction =
       action_classifier_set.getMatchSet().getBestValue();
   }
```

Programm 16 Drittes Kernstück des SXCS-Algorithmus (*calculateNextMove()* - Auswahl der nächsten Aktion und Ermittlung und Speicherung des zugehörigen ActionSets)

```
/**
 * Bestimmt die zum letzten bekannten Status passenden Classifier und
 * wählt aus dieser Menge eine Aktion. Außerdem wird das aktuelle
 * ActionClassifierSet mithilfe der gewählten Aktion ermittelt.
 * Im Vergleich zur originalen Multistepversion wird am Schluß noch
 * das ermittelte ActionSet gespeichert.
 *
 * @param gaTimestep Der aktuelle Zeitschritt
 */

  public void calculateNextMove(long gaTimestep) {

 /**
  * Überdecke das classifierSet mit zum Status passenden Classifiern
  * welche insgesamt alle möglichen Aktionen abdecken.
  */
    classifierSet.coverAllValidActions(
                    lastState, getPosition(), gaTimestep);

 /**
  * Bestimme alle zum Status passenden Classifier.
  */
    lastMatchSet = new AppliedClassifierSet(lastState, classifierSet);

 /**
  * Entscheide auf welche Weise die Aktion ausgewählt werden soll,
  * wähle Aktion und bestimme zugehöriges ActionSet
  */
    lastExplore = checkIfExplore(lastState.getSensorGoalAgent(),
                                          lastExplore, gaTimestep);

    calculatedAction = lastMatchSet.chooseAbsoluteDirection(lastExplore);
    lastActionSet = new ActionClassifierSet(lastState, lastMatchSet,
                                                    calculatedAction);

 /**
  * Speichere das ActionSet und passe den Stack bei einem Überlauf an
  */
    actionSetSize++;
    historicActionSet.addLast(lastActionSet);
    if (historicActionSet.size() > Configuration.getMaxStackSize()) {
      historicActionSet.removeFirst();
    }
  }
```

9.4.3 Zielobjekt mit SXCS

Wie bereits in Kapitel 4.2.7 erwähnt, soll hier eine Implementierung von SXCS für das Zielobjekt diskutiert werden. Bis auf die Funktion *checkRewardPoints()* (siehe Programm 7.8) ist die Implementierung für das Zielobjekt identisch. Die abgeänderte Version ist in Programm 9.4.3 aufgelistet.

Programm 17 Bestimmung des *base rewards* für Agenten

```
/**
 * @return true Falls das Zielobjekt von keinem Agenten überwacht wird
 */
@Override
public boolean checkRewardPoints() {
  boolean[] sensor_agent = lastState.getSensorAgent();

  for(int i = 0; i < Action.MAX_DIRECTIONS; i++) {
    if(sensor_agent[2*i+1]) {
      return false;
    }
  }

  return true;
}
```

Kapitel 10

Analyse SXCS

Tabelle 10.1: Vergleich “Intelligent (Open)” und “Intelligent (Hide)” (8 Agenten, Säulenszenario)

Algorithmus	Abdeckung	Qualität
“Intelligent (Open)”		
Zufällige Bewegung	72.55%	11.58%
XCS	71.35%	13.98%
SXCS	72.10%	13.50%
“Intelligent (Hide)”		
Zufällige Bewegung	72.56%	11.78%
XCS	71.33%	14.27%
SXCS	72.05%	13.90%

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

Geschwindigkeit 2 problematisch, Geschwindigkeit 1 ok?

TODO classifier ausgeben

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


#0######.###0#0##.#0#0###0-S : [Fi: 0.38] [Ex: 00450.0] [Pr: 0.74] [PE: 0.38]
```

Tabelle 10.2: Vergleich “Intelligent (Open)” und “Intelligent (Hide)” (8 Agenten, Säulenszenario)

Algorithmus	Abdeckung	Qualität
“Intelligent (Open)”		
Zufällige Bewegung	72.55%	11.58%
XCS	71.35%	13.98%
SXCS	72.10%	13.50%
“Intelligent (Hide)”		
Zufällige Bewegung	72.56%	11.78%
XCS	71.33%	14.27%
SXCS	72.05%	13.90%

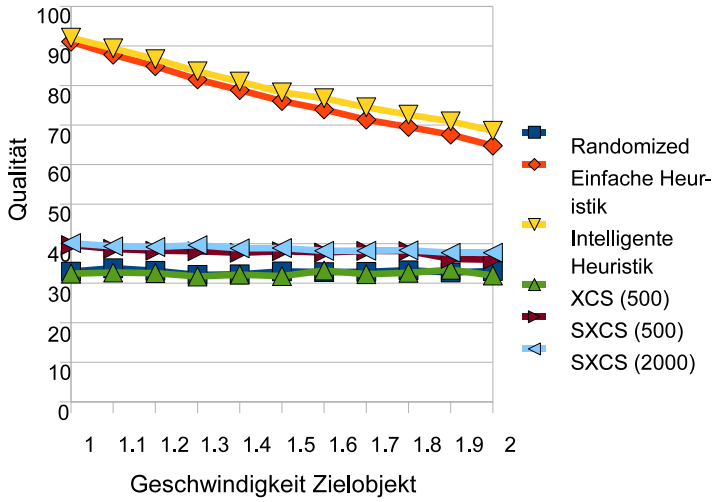

Abbildung 10.1: Vergleich der Qualitäten verschiedener Algorithmen bezüglich der Geschwindigkeit des Zielobjekts

. . . .

TODO

10.1 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.

SXCS sehr gut bei NO DIRECTION CHANGE und speed 1!

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

10.2 Standard XCS Multistepverfahren

10.2.1 SXCS und Heuristiken

erst multistep... mit random vergleichen

In allen Tests erreichten die Heuristiken deutlich bessere Ergebnisse. Diesen Nachteil hat sich LCS in diesen Szenarien durch deutlich überlegene Flexibilität erkauft 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 LCS Agenten unmöglich ein solches globales Ziel erreichen können, es ist also kein emergentes Verhalten zu beobachten. Dies ist dadurch zu begründen, dass bei der Berechnung des Rewards keine Information außer der eigenen, lokalen Information

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

TODO statistical value:Error in predictions!

10.2.2 Vergleich Multistep / LCS

Szenarien, Parameter.

10.2.3 Test der verschiedenen Exploration-Modi

Prediction Error sehr hoch, da dynamisches

Kapitel 11

Kommunikation

Einführung, Kommunikationsbeschränkungen (nur Reward weitergeben)

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

reward all equally besser als reward none

11.1 Realistischer Fall mit Kommunikationsrestriktionen

Bisher wurde der Fall betrachtet, dass Kommunikation mit beliebiger Reichweite stattfinden kann. Dies ist natürlich kein realistisches Szenario. Geht man jedoch davon aus, dass die Kommunikationsreichweite zumindest ausreichend groß ist um nahe Agenten zu kontaktieren, so kann man argumentieren, dass man dadurch ein Kommunikationsnetzwerk aufbauen kann, in dem jeder Agent jeden anderen Agenten - mit einer gewissen Zeitverzögerung - erreichen kann. Bei ausreichender Agentenzahl relativ zur freien Fläche fallen dadurch nur vereinzelte Agenten aus dem Netz, was der Effektivität der Agentengruppe erwartungsgemäß nur geringfügig schadet (TODO zeigen?) Stehen die Agenten nicht indirekt andauernd miteinander in Kontakt (mit anderen Agenten als Proxy), son-

dern muss die Information zum Teil durch aktive Bewegungen der Agenten transportiert werden, tritt eine Zeitverzögerung auf. Auch kann die benötigte Bandbreite die verfügbare übersteigen, was ebenfalls Zeit benötigt. Im realistischen Fall ist also davon auszugehen, dass jede Kommunikation erst mit einer gewissen Verzögerung ausgeführt wird, weshalb für Kommunikation nur der zuvor besprochene verzögerte LCS Algorithmus in Frage kommt.

pg. 286 Zentralisierung der Daten

TODO bei Faktorberechnung Ranking

11.2 Lösungen aus der Literatur

Da wir ein Multiagentensystem betrachten, stellt sich natürlich die Frage nach der Kommunikation. In der Literatur gibt es Multiagentensysteme die auf Learning Classifier Systemen aufbauen, wie z.B. TODO Literatur. Alle Ansätze in der Literatur erlauben jedoch globale Kommunikation, z.T. Gibt es globale Classifier auf die alle Agenten zurückgreifen können, z.T. gibt es globale Steuerung.

Verteilung des rewards an alle - soccer

TODO Einordnen In [20] gezeigt, Gruppenbildung (rationality, grade 2 confusion)

soccer!

[8] OCS, centralized control system

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

11.3 SXCS Variante mit verzögerter Reward (DSXCS)

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

Realistischer Fall

Drei Werte weitergeben... Egoismus Faktor, Reward und Timestamp

Der wesentliche Unterschied zur ersten XCS Variante SXCS ist, dass jeglicher ermittelter *reward* Wert und der jeweils zugehörige Faktor lediglich erst einmal zusammen mit den jeweiligen *actionSets* in einer Liste (*historicActionSet* TODO Bezeichnung) gespeichert werden und in jedem Schritt immer nur die *classifiers* des *actionSets* des ältesten Eintrags in der *historicActionSet* Liste aktualisiert wird. Somit haben wir also eine zeitlich beliebig verzögerbare Aktualisierungsfunktion, welche uns erlaubt, mehrere gleichzeitig stattgefundene (aber erst verzögert eintreffende, wegen z.B. Kommunikationsschwierigkeiten) Ereignisse zusammen auszuwerten. Dies ist eine wesentliche Voraussetzung für Kommunikation zwischen den Agenten. TODO

Wann immer ein *base reward* Wert an einen Agenten verteilt wird, kann es sinnvoll sein, diesen *base reward* an andere Agenten weiterzugeben. Dies wurde z.B. in einem ähnlichen Szenario in [22] festgestellt, bei dem zwei auf XCS basierende Agenten gegen bis zu zwei anderen (zufälligen) Agenten eine vereinfachte Form des Fußballs spielen. Das in dieser Arbeite besprochene Szenario ist wesentlich komplexer, was d

Die Funktion *calculateReward()* ist identisch mit der in Kapitel 9.4.2 besprochenen Funktion bei der SXCS Variante ohne verzögerten *reward*.

In der Funktion *processReward()* werden die gespeicherten *reward* und *factor* ausgewertet. In der Implementation in Programm 11.3 werden einfach alle nacheinander auf das

action set angewendet, während in der verbesserten Version in Programm 11.3 nur der *reward* Wert aus dem Paar mit dem größten Produkt aus den *reward* und *factor* Werten für die Aktualisierung benutzt wird. In beiden Implementationen werden außerdem Einträge mit sowohl einem *reward* als auch *factor* Wert von 1.0 ignoriert, sie wurden bereits in Programm 11.3 ausgewertet.

11.4 Ablauf

TODO wann weitergabe des rewards

Jeder Reward, der aus einem normalen Ereignis generiert wird, wird unter Umständen an alle anderen Agenten weitergegeben. Wie ein solches sogenanntes "externes Ereignis" von diesen Agenten aufgefasst wird, hängt von der jeweiligen Kommunikationsvariante ab, die in (11.5) besprochen werden.

Durch eine gemeinsame Schnittstelle erhält jeder Agent den Reward zusammen mit dem Kommunikationsfaktor. Dabei ergibt sich das Problem, dass sich Rewards überschneiden können, da jeder Reward sich rückwirkend auf die vergangenen ActionClassifierSets auswirken kann. Auch können mehrere externe Rewards eintreffen als auch ein eigener lokaler Reward aufgetreten sein. Würden die Rewards nach ihrer Eingangsreihenfolge abgearbeitet werden, kann es passieren, dass das selbe ActionClassifierSet sowohl mit einem hohen als auch einem niedrigen Reward aktualisiert wird. Da das globale Ziel ist, den Zielagenten durch *irgendeinen* Agenten zu überwachen, ist es in jedem einzelnen Zeitschritt nur relevant, dass ein *einzelner* Agent einen hohen Reward produziert bzw. weitergibt um die eigene Aktion als zielführend zu bewerten.

Befindet sich das Ziel beispielsweise gerade in Überwachungsreichweite mehrerer Agenten und verliert ein anderer Agent das Ziel aus der Sicht, sollte der Agent (und alle anderen Agenten), der das Ziel in Sicht hat, deswegen nicht bestraft werden, da das globale Ziel ja weiterhin erfüllt wurde.

Programm 18 Zweites Kernstück des verzögerten SXCS-Algorithmus (*collectReward()* - Verteilung des Rewards auf die ActionSets)

```
/**
 * Diese Funktion verarbeitet den übergebenen Reward und gibt ihn an die
 * zugehörigen ActionSets weiter. Wesentlicher Unterschied zum LCS ohne
 * Verzögerung ist, dass maxPrediction erst bei der endgültigen
 * Verarbeitung des historicActionSets ermittelt wird.
 *
 * @param reward Wahr wenn der Zielagent in Sicht war.
 * @param best_value Bester Wert des vorangegangenen actionSets
 * @param is_event Wahr wenn diese Funktion wegen eines Ereignisses, d.h.
 *        einem positiven Reward, aufgerufen wurde
 */

 public void collectReward(
              boolean reward, double best_value, boolean is_event) {
   double corrected_reward = reward ? 1.0 : 0.0;

 /**
  * Aktualisiere eine ganze Anzahl von Einträgen im historicActionSet
  */
    for(int i = 0; i < action_set_size; i++) {

 /**
  * Benutze aufsteigenden bzw. absteigenden Reward bei einem positiven
  * bzw. negativen Ereignis
  */
      if(is_event) {
        corrected_reward = reward ?
          calculateReward(i, action_set_size) :
          calculateReward(action_set_size - i, action_set_size);
      } else {
        if(corrected_reward == 1.0 && factor == 1.0) {
          historicActionSet.get(start_index - i).
            rewardPrematurely(
              historicActionSet.get(start_index - i + 1).getBestValue());
        }
      }

 /**
  * Füge den ermittelten Reward zum historicActionSet
  */
      historicActionSet.get(start_index - i).
        addReward(corrected_reward, factor);

   }
```

Programm 19 Auszug aus dem dritten Kernstück des verzögerten SXCS-Algorithmus (*calculateNextMove()*)

```
/**
 * Der erste Teil der Funktion ist identisch mit dem calculateNextMove
 * der SXCS Variante ohne Kommunikation. Der Zusatz ist, dass beim
 * Überlauf die im HistoricActionSet gespeicherte Rewards verarbeitet
 * werden
 */

  public void calculateNextMove(long gaTimestep) {

  // ...

  /**
   * HistoryActionSet voll? Dann verarbeite den dort gespeicherten Reward
   */
     if (historicActionSet.size() > Configuration.getMaxStackSize()) {
      HistoryActionClassifierSet first = historicActionSet.pop();
      last.processReward(historicActionSet.getFirst().getBestValue());
    }
  }
```

Programm 20 Auszug aus dem vierten Kernstück des verzögerten SXCS-Algorithmus (Verarbeitung des Rewards, *processReward()* TODO)

```
/**
 * Zentrale Routine des HistoryActionSets zur Verarbeitung aller
 * eingegangenen Rewards bis zu diesem Punkt.
 */

  public void processReward(double max_prediction) {

  /**
   * Finde das größte reward / factor Paar TODO Verbessern
   */
    for(RewardHelper r : reward) {
    /**
     * Dieser Eintrag wurde schon in collectReward() verwertet
     */
      if(r.reward == 1.0 && r.factor == 1.0) {
        continue;
      }
    /**
     * Aktualisiere den Eintrag mit den entsprechenden Werten und dem
     * übergebenen maxPrediction Wert
     */
      actionClassifierSet.updateReward(r.reward, max_prediction, r.factor);
    }
  }
```

Programm 21 Verbesserte Variante Auszug aus dem vierten Kernstück des verzögerten SXCS-Algorithmus (Verarbeitung des Rewards, *processReward()* TODO)

```
/**
 * Zentrale Routine des HistoryActionSets zur Verarbeitung aller
 * eingegangenen Rewards bis zu diesem Punkt.
 */

  public void processReward(double max_prediction) {

    double max_value = 0.0;
    double max_reward = 0.0;
  /**
   * Finde das größte reward / factor Paar TODO Verbessern
   */
    for(RewardHelper r : reward) {
    /**
     * Dieser Eintrag wurde schon in collectReward() verwertet
     */
      if(r.reward == 1.0 && r.factor == 1.0) {
        return;
      }

      if(r.reward * r.factor > max_value) {
        max_value = r.reward * r.factor;
        max_reward = r.reward;
      }
    }
    /**
     * Aktualisiere den Eintrag mit dem ermittelten Wert und dem
     * übergebenen maxPrediction Wert
     */
    actionClassifierSet.updateReward(max_reward, max_prediction, 1.0);
  }
```

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

Gebe keinen Reward an andere Agenten weiter. Es ist nicht relevant, ob ein Agent das Ziel aus den Augen verliert oder nicht, es ist nur relevant, ob der Zielagent weiterhin von anderen Agenten beobachtet wird. Ein Sonderfall ist, wenn im vorherigen Schritt der Zielagent nicht in Sichtweite eines anderen Agenten stand, also in diesem Schritt auf einmal mehrere Agenten den Zielagenten sehen können. In diesem Fall gibt nur der erste Agent den Reward weiter und setzt ein Flag.

Ziel verschwindet aus Sicht War der Zielagent von keinem anderen Agenten in Sicht, dann hat sich der Zielagent hiermit aus der Sichtweite aller Agenten bewegt. Somit haben alle Agenten versagt und der negative Reward wird weitergegeben.

Selbiges wenn das Ziel in Sicht kommt und von keinem anderen Agenten in Sicht ist. Die Agenten waren offensichtlich erfolgreich und können belohnt werden.

TODOTODOTODOTODO Ist kein Event aufgetreten und leeren wir die Hälfte des Stacks ist es nicht sinnvoll, einen 0-Reward weiterzugeben, da zwangsläufig immer mehrere Agenten eine längere Zeit den Zielagenten nicht sehen, selbst wenn sie sich optimal verteilen / bewegen. TODO

Dies zeigt auch der Test: TODO

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

11.5 Kommunikationsvarianten

Allen hier vorgestellten Kommunikationsvarianten ist gemeinsam, dass sie einen Kommunikationsfaktor berechnen, nach denen sie den externen Reward, den ihnen ein anderer Agent übermittelt hat, bewerten. Der Kommunikationsfaktor gewichtet alle Verwendungen des Parameters β (welcher die Lernrate bestimmt). Ein Faktor von 1.0 hieße, dass der

externe Reward wie ein normaler Reward behandelt wird, ein Faktor von 0.0 hieße, dass externe Rewards deaktiviert sein sollen. Die Idee ist, dass unterschiedliche Agenten unterschiedlich stark am Erfolg des anderen Agenten beteiligt sind, da ohne Kommunikation jeder Agent versuchen wird, selbst den Zielagenten möglichst in die eigene Überwachungsreichweite zu bekommen, anstatt mit anderen Agenten zu kooperieren, also das Gebiet des Grids möglichst großräumig abzudecken.

Gruppenbildung

11.5.1 Einzelne Gruppe

Mit dieser Variante wird der Kommunikationsfaktor fest auf 1.0 gesetzt und es werden alle Rewards in gleicher Weise weitergegeben. Dadurch wird zwischen den Agenten nicht diskriminiert, was letztlich bedeutet, dass zwar zum einen diejenigen Agenten korrekt mit einem externen Reward belohnt werden, die sich zielführend verhalten, aber zum anderen eben auch diejenigen, die es nicht tun. Deren Classifier werden somit zu einem gewissen Grad zufällig bewertet, denn es fehlt die Verbindung zwischen Classifier und Reward.

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

In Tests (TODO) haben sich dennoch in bestimmten Fällen mit “Reward all equally” deutlich bessere Ergebnisse gezeigt als im Fall ohne Kommunikation. Dies ist wahrscheinlich darauf zurückzuführen, dass in diesen Fällen die Kartengröße und Geschwindigkeit des Zielagenten relativ zur Sichtweite und Lerngeschwindigkeit zu groß war, die Agenten also annahmen, dass ihr Verhalten schlecht ist, weil sie den Zielagenten relativ selten in Sicht bekamen. Eine Weitergabe des Rewards an alle Agenten kann hier also zu einer Verbesserung führen, dabei ist der Punkt aber nicht, dass Informationen ausgetauscht werden, sondern, dass obiges Verhältnis zugunsten der Sichtweite gedreht wird. Für die Auswahl

geeigneter Tests sollten die Szenario-Parameter also möglichst so gewählt werden, dass "Reward all equally" keinen signifikanten Vorteil gegenüber "No external reward" bringt. Blickt man auf diesen Sachverhalt aus einer etwas anderen Perspektive ist es auch einleuchtend. Es scheint offensichtlich, dass es relevant ist, ob das Spielfeld z.B. 100x100 oder nur 10x10 Felder groß ist, wenn es darum geht, das Verhalten über die Zeit hinweg zu bewerten. In den Algorithmus für die Kommunikation bzw. für die Rewardvergabe müsste man deshalb einen weiteren (festen) Faktor einbauen, der zu Beginn in Abhängigkeit von Größe des zu überwachenden Feldes berechnet wird. Dies soll aber nicht Teil der Arbeit werden. TODO

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

11.5.2 Gruppenbildung über Ähnlichkeit der *classifier* der Agenten

Eine dritte Implementation vergleicht die Classifier jeweils beider bei der Kommunikation beteiligten Agenten direkt. Alle Classifier des Agenten, der den Reward weitergibt, die ausreichend Erfahrung gesammelt haben und ausreichend genau ist (*experience* und geringes *predictionError*, Bedingung ist identisch mit *isPossibleSubsumer*), werden mit einem identischen Classifier (d.h. mit gleicher *condition* und gleicher *action*) verglichen. Die Differenz der Produkte aus *fitness* und *prediction* geteilt durch den größeren *prediction*-Wert der beiden Classifier stellt hier den Faktor dar.

pSet1 sei eine Teilmenge (bestehend aus Classifiern, deren *experience* größer als *thetaSubsumer* und dessen *predictionError* kleiner als epsilon0 ist) des ClassifierSets des Agenten, der den Reward vergibt. *pSet2* sei die gleiche Teilmenge, allerdings des Agen-

ten, der den Reward empfängt.

Nun werden Paare identischer Classifier aus *pSet1* und *pSet2* gebildet. Gibt es mehrere Kandidaten für den selben Classifier aus *pSet1*, wird der mit dem ähnlichsten Produkt aus *fitness* und *prediction* gewählt. Die Differenz zwischen den beiden Classifiern eines jeden Paares wird anhand ihres *prediction*-Werts auf einen Wert zwischen 0.0 und 1.0 skaliert und aufaddiert. Die resultierende Summe wird schließlich durch die Anzahl der Paare dividiert und man erhält den Kommunikationsfaktor.

Ein wesentlicher Nachteil hierbei ist natürlich, dass Classifier-Daten direkt übertragen werden müssen, was bei großer *maxPopulation* u.U. einen hohen Kommunikations- und Speicheraufwand darstellt.

Umgekehrt könnte man diese Funktion auch noch ausbauen und weitergehende Vergleiche ausführen, auch unter Einbeziehung der restlichen Classifier beider ClassifierSets, deren *condition* bzw. *action* nicht übereinstimmen. Beispielsweise kann man alle möglichen Zustände der Sensordaten durchgehen, die passenden Classifier beider ClassifierSets auswählen und die Wahrscheinlichkeiten für gewählte Aktionen vergleichen, was aber hinsichtlich der benötigten Rechenzeit nur für kleine Mengen von Sensordaten sinnvoll erscheint.

TODO Format!

11.5.3 Gruppenbildung über Ähnlichkeit des Verhaltens der Agenten

Eine weitere Variante berechnet erst einmal für jeden Agenten einen “Egoismus-Faktor”, indem grob die Wahrscheinlichkeit ermittelt wird, dass ein Agent, wenn sich ein anderer Agent in Sicht befindet, sich in diese Richtung bewegt. “Egoismus”-Faktor, weil ein großer Faktor bedeutet, dass der Agent eher einen kleinen Abstand zu anderen Agenten bevorzugt, also wahrscheinlich eher auf eigene Faust versucht, den Zielagenten in Sicht zu

Programm 22 "Simple relation", Algorithmus zur Bestimmung des Kommunikationsfaktors basierend auf *prediction* und *fitness* der ClassifierSets

```
/**
 * Relation of this classifier set (the active agent classifier set,
 * e.g. the set that received a reward) to another classifier set
 * @param other The other set we want to compare with
 * @return degree of relationship (0.0 - 1.0)
 */
public double checkDegreeOfRelationship(MainClassifierSet other) {
    double degree = 0.0;
    int size = 0;
    ArrayList<Classifier> matched = new ArrayList<Classifier>();

    for (Classifier c : getClassifiers()) {
        if(!c.isPossibleSubsumer()) {
            continue;
        }

        Classifier cl = other.getBestIdenticalClassifier(matched, c);
        if (cl != null) {
            matched.add(cl);

            double div = c.getPrediction();
            if(cl.getPrediction() > div) {
                div = cl.getPrediction();
            }
            if(div != 0.0) {
                double difference =
                  1.0 - Math.abs(
                    c.getFitness() * c.getPrediction() -
                    cl.getFitness() * cl.getPrediction()) / div;
                if(difference > 1.0) {
                    difference = 1.0;
                } else
                if(difference < 0.0) {
                    difference = 0.0;
                }
                degree += difference;
            }
        }
        size++;
    }

    if(size == 0) {
        return 0.0;
    }

    return degree / (double)size;
  }
```

bekommen anstatt ein möglichst großes Gebiet abzudecken.

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

Damit gäbe es einen Druck in Richtung eines bestimmten Egoismus-Faktors. TODO

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

Die Variante müsste also zum einen in

schlecht abschneiden TODO

11.6 Bewertung Kommunikation:

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

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

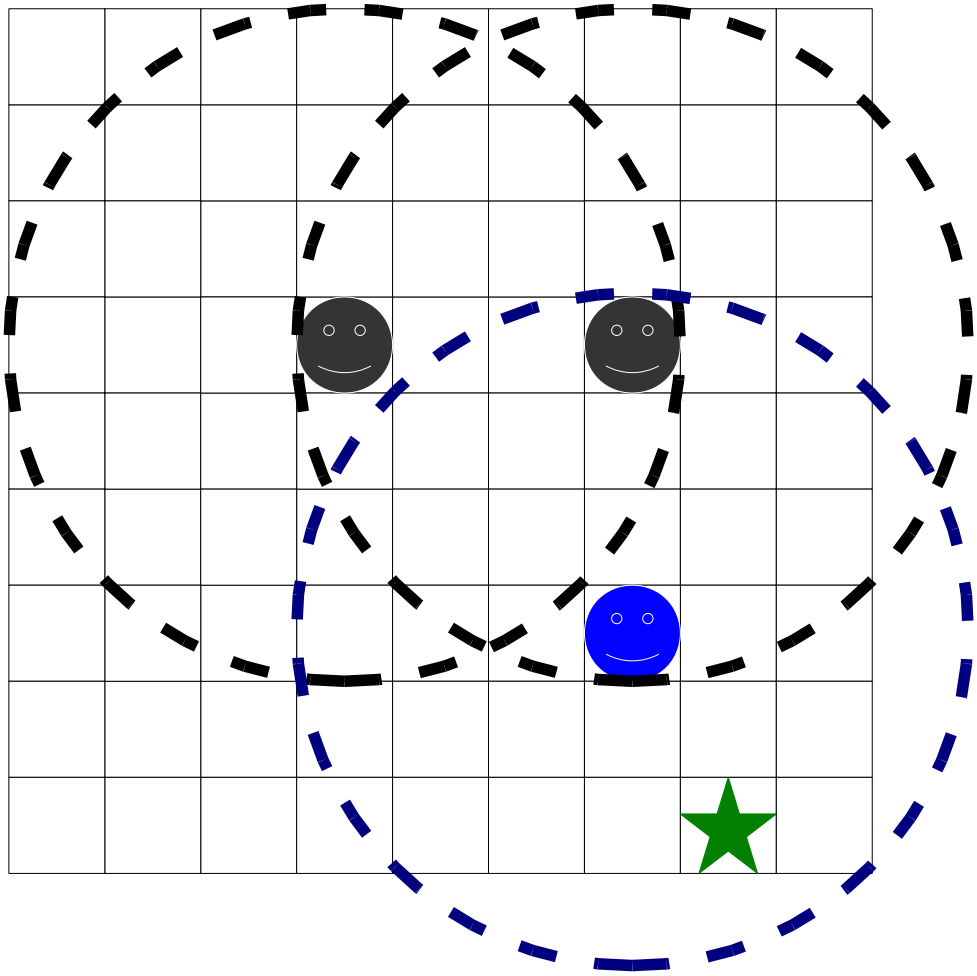

Abbildung 11.1: Schematische Darstellung der Rewardverteilung an ActionSets bei einem neutralen Ereignis

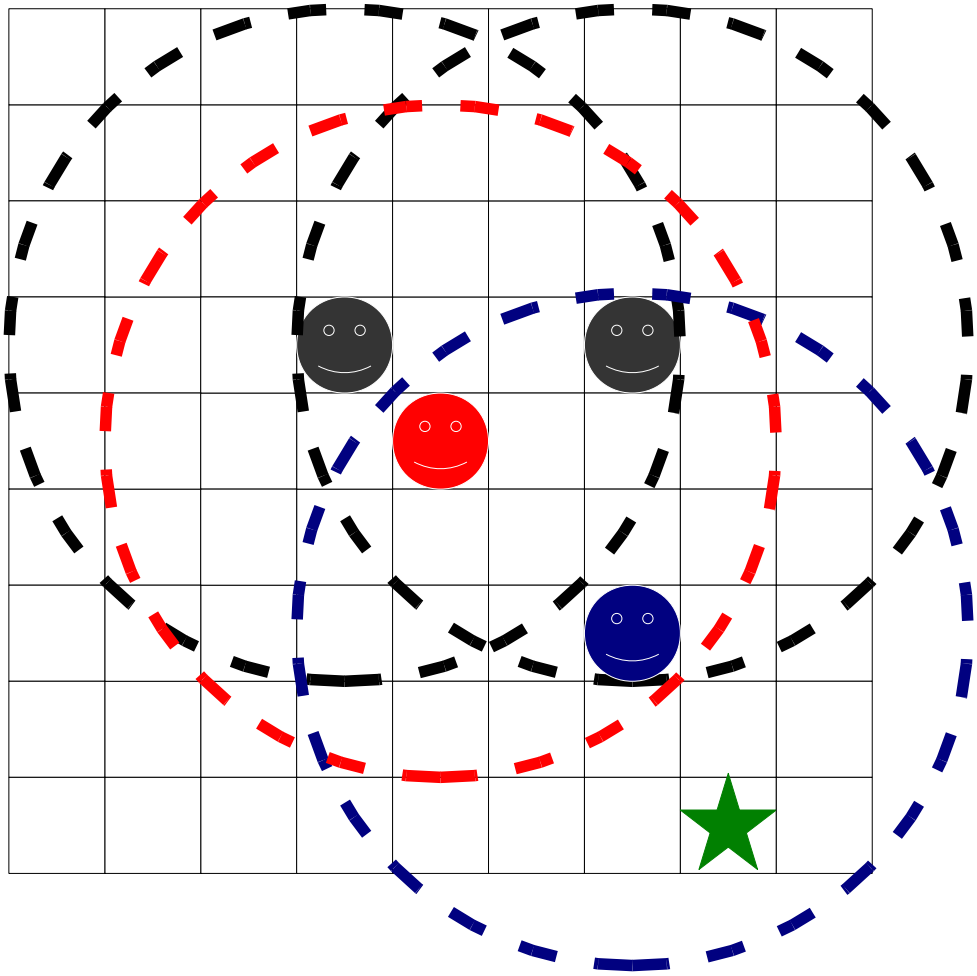

Abbildung 11.2: Schematische Darstellung der Rewardverteilung an ActionSets bei einem neutralen Ereignis

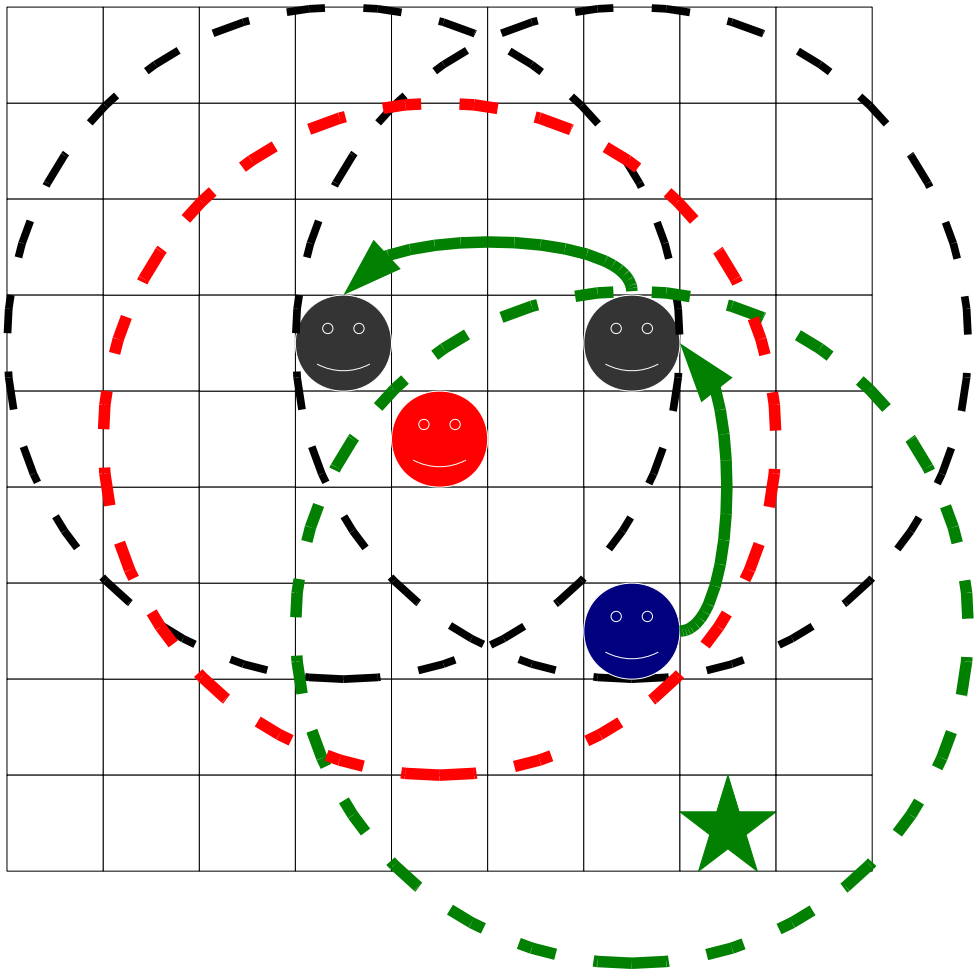

Abbildung 11.3: Schematische Darstellung der Rewardverteilung an ActionSets bei einem neutralen Ereignis

Programm 23 "Egoistic relation", Algorithmus zur Bestimmung des Kommunikationsfaktors basierend auf dem Verhalten des Agenten gegenüber anderen Agenten

```
/**
 * Relation of this classifier set (the active agent classifier set,
 * e.g. the set that received a reward) to another classifier set
 * @param other The other set we want to compare with
 * @return degree of relationship (0.0 - 1.0)
 */
public double checkEgoisticDegreeOfRelationship(
                final MainClassifierSet other) {
    double ego_factor =
             getEgoisticFactor() - other.getEgoisticFactor();
    if(ego_factor == 0.0) {
        return 0.0;
    }
    return 1.0 - ego_factor * ego_factor;
}

public double getEgoisticFactor() throws Exception {
    double factor = 0.0;
    double pred_sum = 0.0;
    for(Classifier c : getClassifiers()) {
        if(!c.isPossibleSubsumer()) {
            continue;
        }
        factor += c.getEgoFactor();
        pred_sum += c.getFitness() * c.getPrediction();
    }
    if(pred_sum > 0.0) {
        factor /= pred_sum;
    } else {
        factor = 0.0;
    }
    return factor;
}
```

11.6.1 Vergleich TODO

Old LCS Agent New LCS Agent

Multistep LCS Agent Dieser Algorithmus stellt eine Implementation des Standard XCS Algorithmus dar. Unterschied zur Standardimplementation ist, dass die Probleminstanz bei Erreichen des temporären Ziels (d.h. den Zielagenten in Sicht zu bekommen) nicht tatsächlich neugestartet wird. Events, wie bei den neuen LCS Implementationen gibt es nicht, ist das Ziel in Sicht wird Reward 1.0 weitergegeben.

Single LCS Agent

Mehrere LCS Agenten ("Old LCS Agent") teilen sich ein gemeinsames ClassifierSet, das sie entsprechend updaten. Entspricht dem Extremfall der Kommunikation Sight range/Kommunikationsrange

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

TODOGrafiken

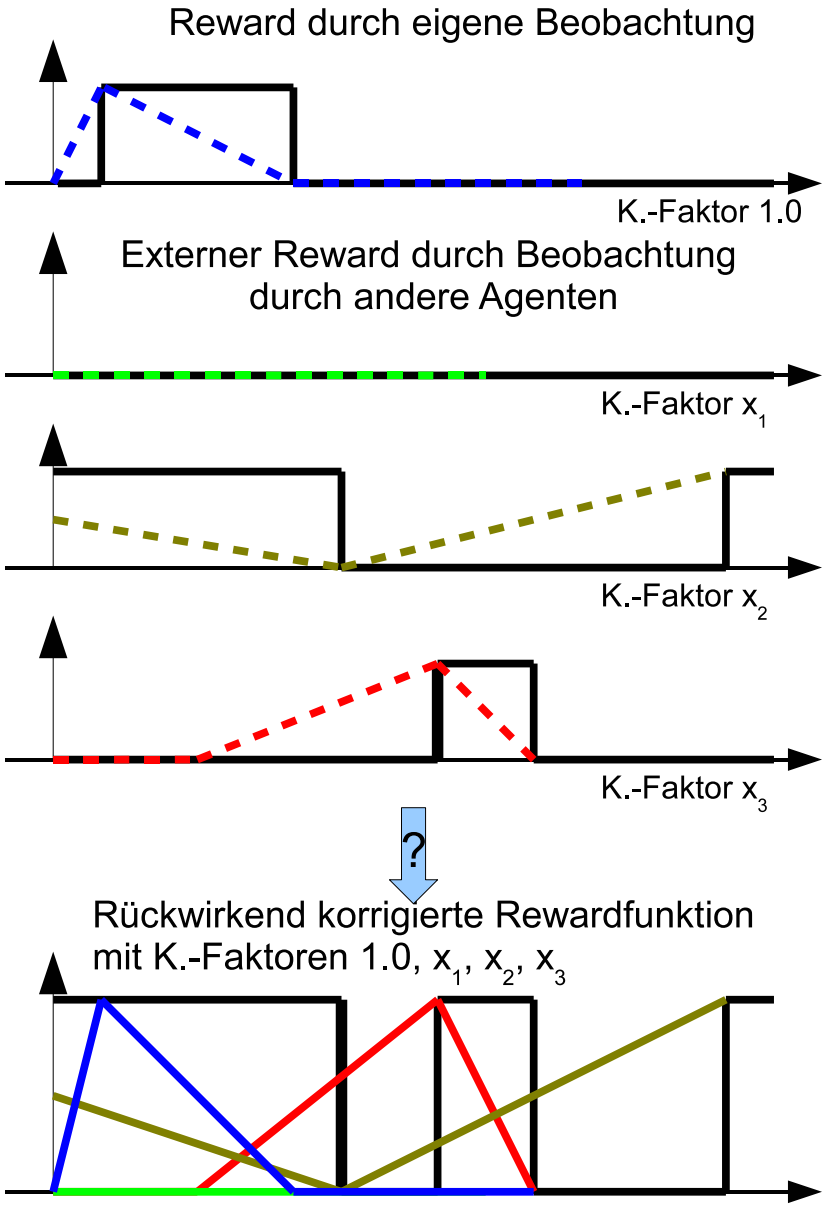

Abbildung 11.4: Beispielhafte Darstellung der Kombinierung interner und externer Rewards

Kapitel 12

Zusammenfassung, Ergebnis und Ausblick

12.1 Zusammenfassung

Zu Beginn wurde auf die Szenariodefinition und die Fähigkeiten der Agenten eingegangen. Anhand von Beispielen heuristischer Agenten wurden einige Grundeigenschaften der präsentierten Szenarien als Vorbereitung für die Analyse der Learning Classifier Systeme bestimmt. Nach der Einführung in LCS, der Beschreibung des Standardverfahren XCS und der angepassten Implementierung für Überwachungsszenarios konnten dann umfangreiche Tests ausgeführt werden. Danach wurde durch die Hinzunahme von der Möglichkeit zur Kommunikation eine angepasste Implementierung für verzögerten Reward definiert auf Basis dessen dann mehrere Varianten für die Weitergabe des Rewards vorgestellt, analysiert und verglichen wurden.

12.2 Ergebnis

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 des Rewards 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.

12.3 Ausblick

Weitere Untersuchungen sind nötig um zu bestimmen, inwiefern Kommunikation, beispielsweise mit einer größeren Zahl an besseren Sensoren, zu einem besseren Ergebnis führen kann. TODO

Vom theoretischen Standpunkt ist noch zu klären, warum genau der zeitliche Abstand zum Erfolg geführt hat und wo die Grenzen hierfür liegen.

Erschwerung, mehr Kollaboration TODO aus verschiedenen Richtungen betrachten? Mehrere Agenten notwendig?

Kapitel 13

Verwendete Hilfsmittel und Software

Zu Beginn stellte sich die Frage, welche Software zu benutzen ist, da es sich um ein recht komplexe Problemstellung handelt. Begonnen habe ich mit der YCS Implementierung von TODO. Sie ist in der Literatur wenig vertreten, die Implementierung bot aber einen guten Einstieg in das Thema, da sie sich auf das Wesentliche beschränkte und keine Optimierungen enthielt.

Der nächste Schritt war zu entscheiden, auf welchem System die Agenten simuliert werden sollen. Unter einer Reihe von vorhandenen Implementierungen entschied ich mich für eine eigene Implementation. Wesentlicher Grund war die Unerfahrenheit mit den Lösungen (und der damit verbundenen Einarbeitungszeit) wie auch Überlegungen bzgl. der Geschwindigkeit, dem Speicherverbrauch und der Kompatibilität. TODO

Das Programm und die zugehörige Oberfläche zum Erstellen von Test-Jobs wurden in Netbeans 6.5 programmiert.

Grafiken wurden mittels GnuPlot erstellt.

Grafiken der Grid-Konfiguration wurden im Programm mittels GifEncode TODO erste * @version 0.90 beta (15-Jul-2000) * @author J. M. G. Elliott (tep@jmge.net)

Wesentlicher Bestandteil der Konfigurationsoberfläche war auch eine Automatisierung

der Erstellung von Konfigurationsdateien, Batchdateien (für ein Einzelsystem und für JoSchKA) zum Testen einer ganzen Reihe von Szenarien und auch GnuPlot Skripts.

Speicherverbrauch

Speicherung der Agentenpositionen und des Grids verbrauchen fast keinen Speicher TODO Wesentlicher Faktor waren die LCS Systeme mit ihren ClassifierSets TODO

OpenOffice

LEd Latex

13.1 Beschreibung des Konfigurationsprogramms

Abbildung 13.1: Screenshot des Konfigurationsprogramms

Literaturverzeichnis

[1] Butz, M. & Wilson, S.W.: *An Algorithmic Description of XCS*, 2001. In P-L. Lanzi, W. Stolzmann & S.W. Wilson (eds) Advances in Learning Classifier Systems: IWLCS 2000. Springer, pp253-272.

[2] Wilson, S.W.: *Classifier Fitness Based on Accuracy*, 1995 Evolutionary Computation 3(2): 149-175

[3] Butz, M.: *XCSJava 1.0: An implementation of the XCS classifier system in Java* IlliGAL Report No. 2000027, June, 2000 `http://www.illigal.uiuc.edu/pub/papers/IlliGALs/2000027.ps.Z`

[4] Larry Bull: *A Simple Accuracy-Based Learning Classifier System*, `http://www2.cmp.uea.ac.uk/~it/ycs/ycs.pdf`

[5] Carol Hamer, *J2ME Games With MIDP2*, Apress, 2004, ISBN 1-590-59382-0 `http://www.java-tips.org/java-me-tips/midp/how-to-create-a-maze-game-in-j2me-3.html`

[6] M. V. Butz, K. Sastry, and D. E. Goldberg: *"Tournament selection: Stable fitness pressure in XCS,"* in Lecture Notes in Computer Science, E. Cantú-Paz, J. A. Foster, K. Deb, D. Davis, R. Roy, U.-M. O'Reilly, H.-G. Beyer, R. Standish, G. Kendall, S. Wilson, M. Harman, J. Wegener, D. Dasgupta, M. A. Potter, A. C.

Schultz, K. Dowsland, N. Jonoska, and J. Miller, Eds. Chicago, IL, Jul. 12–16, 2003, vol. 2724, Proc. Genetic and Evol. Comput., pp. 1857–1869.

[7] Martin V. Butz, David E. Goldberg, Pier Luca Lanzi, *Gradient descent methods in learning classifier systems: Improving XCS performance in multistep problems* IEEE Transaction on Evolutionary Computation, 9(5):452–473, October 2005.

[8] Keiki Takadama, Koichiro Hajiri, Tatsuya Nomura, Michio Okada, Shinichi Nakasuka, Katsunori Shimohara *Learning model for adaptive behaviors as an organized group of swarm robots* In Artif Life Robotics (1998) 2 : 123-128, ISAROB 1998

[9] Luis Miramontes Hercog, Terence C. Fogarty, 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*, Proceedings of the International Workshop in Learning Classifier Systems 2001, 2002, Springer-Verlag

[10] Martin V. Butz *The XCS Classifier System* In Studies in Fuzziness and Soft Computing, Springer, 2006, pp51-64. ISBN 978-3-540-25379-2

[11] Martin V. Butz *Simple Learning Classifier Systems* In Studies in Fuzziness and Soft Computing, Springer, 2006, pp51-64. ISBN 978-3-540-25379-2

[12] Stewart W. Wilson: *Classifier Fitness Based on Accuracy.* Evolutionary Computation 3(2): 149-175 (1995)

[13] 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 Proc. Genetic Evol. Comput. Conf., Eds., 1999b, pp. 337–344.

[14] A. M. Barry *"The stability of long action chains in XCS,"*, Soft Comput.—A Fusion Foundations, Methodologies, Applicat., vol. 6, no. 3–4, pp. 183–199, 2002.

[15] *"Classifier fitness based on accuracy,"* Evol. Comput., vol. 3, no. 2, pp. 149–175, 1995

[16] J. R. Koza,W. Banzhaf, K. Chellapilla, K. Deb, M. Dorigo, D. B. Fogel, M. H. Garzon, D. E. Goldberg, H. Iba, and R. Riolo *"Generalization in the XCS classifier system,"* in Proc. 3rd Ann. Conf. Genetic Program., , Eds., 1998, pp. 665–674.

[17] P. L. Lanzi *The XCS library* `http://xcslib.sourceforge.net`

[18] Alejandro Lujan, Richard Werner, Azzedine Boukerche *Generation of Rule-based Adaptive Strategies for a Collaborative Virtual Simulation Environment* PARADISE Research Laboratory, University of Ottawa, HAVE 2008 – IEEE International Workshop on Haptic Audio Visual Environments and their Applications, Ottawa – Canada, 18-19 October 2008

[19] Pier Luca Lanzi, Daniele Loiacono, Stewart W. Wilson, David E. Goldberg: *XCS with Computed Prediction in Continuous Multistep Environments* In Proceedings of the IEEE Congress on Evolutionary Computation – CEC-2005, pages 2032–2039, Edinburgh, UK, September 2005. IEEE.

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

[21] Toufik Benouhiba, Jean-Marc Nigro *An evidential cooperative multi-agent system* Laboratoire ISTIT, CNRS FRE 2732, Universite´ de Technologie de Troyes, Troyes, France

[22] Hiroyasu Inoue, Keiki Takadama, Katsunori Shimohara *Exploring XCS in Multiagent Environments* GECCO 05 June 25.29, 2005, Washington, DC, USA.

Anhang A

Statistical significance tests

This is the first appendix

Anhang B

Implementation

The second appendix...

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