DIPLOMARBEIT

XCS in dynamischen Multiagenten-Überwachungsszenarios ohne globaler Kommunikation

von

Clemens Lode

Institut für Angewandte Informatik
und Formale Beschreibungsverfahren
Universität Karlsruhe

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

Karlsruhe, 30.03.2009

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Kapitel 1

Einführung

Die Diplomarbeit bestand im Wesentlichen aus drei Teilen: Ausarbeitung Szenarien, Sensorik

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 nur auf eines der vier Nachbarfelder bewegen oder stehenbleiben. Die Felder können entweder leer oder durch ein Objekt besetzt sein. Besetzte Felder können nicht betreten werden. Es gibt drei verschiedene Arten von Objekten, unbewegliche Hindernisse, ein zu überwachende Zielagent und Agenten. Zielagent und Agent bewegen sich anhand eines bestimmten Algorithmus anhand bestimmter Sensordaten.

TODO Auflösen

2.1 Problem-Instanzen

Eine einzelne Probleminstanz entspricht einem Torus mit einer (abhängig vom verwendeten Random-Seed-Wert) bestimmten Anfangsbelegung mit bestimmten Objekten und bestimmten Parametern zur Sichtbarkeit. Die Parameter bestimmen, ob und wie Objekte

andere Objekte die Sicht versperren und bis zu welcher Distanz der Zielagent von einem Agenten als “überwacht” gilt, sofern die Sicht durch andere Objekte nicht versperrt ist.

Ein Experiment entspricht dem Test einer Anzahl von Probleminstanzen mit einer Reihe von Random-Seed-Werten. In einem Durchlauf werden mehrere Experimente (jeweils mit unterschiedlichen Random-Seed-Wert-Reihen) durchgeführt.

2.2 Qualität

Problem-Qualität eines Algorithmus zu einem Problem wird anhand des Anteils der Zeit berechnet, die er das Zielobjekt während des Problems überwachen konnte, relativ zur Gesamtzeit. Experiment-Qualität eines Algorithmus zu einer Anzahl von Problemen (einem Experiment) wird Anhand des Gesamtanteilder Zeit berechnet, die er das Zielobjekt während aller Probleme überwachen konnte, relativ zur Gesamtzeit aller Probleme. Qualität eines Algorithmus entspricht dem Durchschnitt aller Experimentqualitäten des Algorithmus.

Halbzeit-Problem-Qualität 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. Halbzeit-Experiment-Qualität 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 Halbzeit-Experiment-Qualitäten eines Algorithmus.

Kapitel 3

Agenten

3.1 Sensoren

Jeder Agent besitzt 3 Gruppen mit jeweils 4 Binärsensoren. Alle Sensoren sind visuelle Sensoren mit begrenzter Reichweite. Je nach Parameter des Szenarios wird die Sicht durch andere Objekten blockiert. Sichtlinien werden durch einen einfachen Bresenham-Linienalgorithmus bestimmt. Jede Gruppe von Sensoren nimmt einen anderen Typ von Objekt wahr. Die erste Gruppe nimmt den Zielagenten, die zweite andere Agenten und die dritte Hindernisse. Ein Sensor ist jeweils in eine bestimmte Richtung ausgerichtet (Norden, Osten, Süden, Westen) und wird auf "wahr" gesetzt, wenn sich in dem von der Sichtweite bestimmten Kegel ein entsprechendes Objekt befindet.

3.2 Fähigkeiten

Jeder Agent kann in jedem Schritt zwischen 5 verschiedenen Aktionen wählen, die den vier Richtungen plus einer Aktion, bei der der Agent sich nicht bewegt, entsprechen. Agenten können pro Zeiteinheit genau einen Schritt durchführen. Der Zielagent kann je nach Szenario mehrere Schritte ausführen. Könnte der Zielagent ebenfalls nur einen

Schritt ausführen, wäre das Problem sehr simpel, da der Zielagent Schwierigkeiten hätte, Agenten abzuschütteln, deren einzige Strategie es ist, sich in Richtung des Zielagenten zu bewegen.

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 wird in Kapitel 13 geliefert.

3.4 Grundsätzliche Algorithmen der Agenten

Neben denjenigen Algorithmen die auf LCS basieren und weiter untern 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 Regeln benutzen und den Erfolg oder Misserfolg ihrer Aktionen ignorieren.

3.4.1 “Randomized”

In jedem Schritt wird eine zufällige Aktion ausgeführt.

3.4.2 “Simple AI Agent”

Ist der Zielagent in Sichtweite, bewegt sich dieser Agent auf das Ziel zu. Ist das Ziel nicht in Sichtweite, führt er eine zufällige Aktion aus.

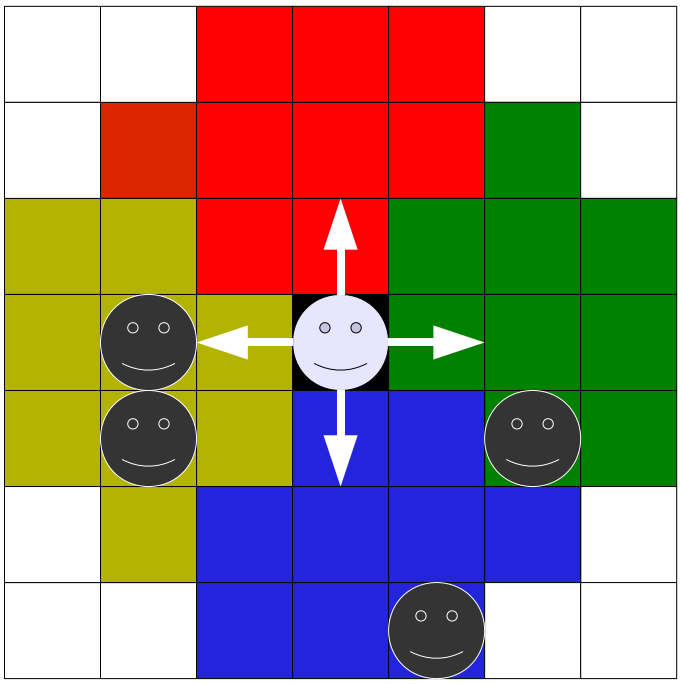

Abbildung 3.1: Agent bewegt sich in eine zufällige Richtung (oder bleibt stehen)

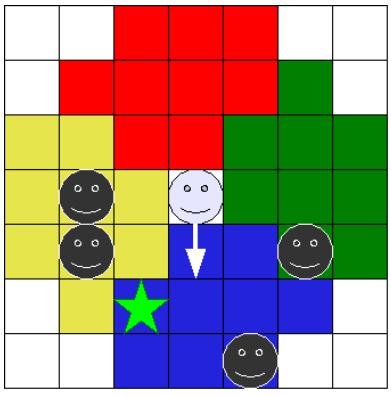

Abbildung 3.2: Agent bewegt sich auf Zielagent zu (sofern sichtbar)

3.4.3 "Intelligent AI Agent"

Ist der Zielagent in Sicht, verhält sich dieser Algorithmus wie "Simple AI Agent". Ist der Zielagent 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 geortet wurde, eine zufällig ausgewählt wird. Falls alle Richtungen belegt (oder alle frei) sind, aus allen Richtungen eine zufällig ausgewählt wird.

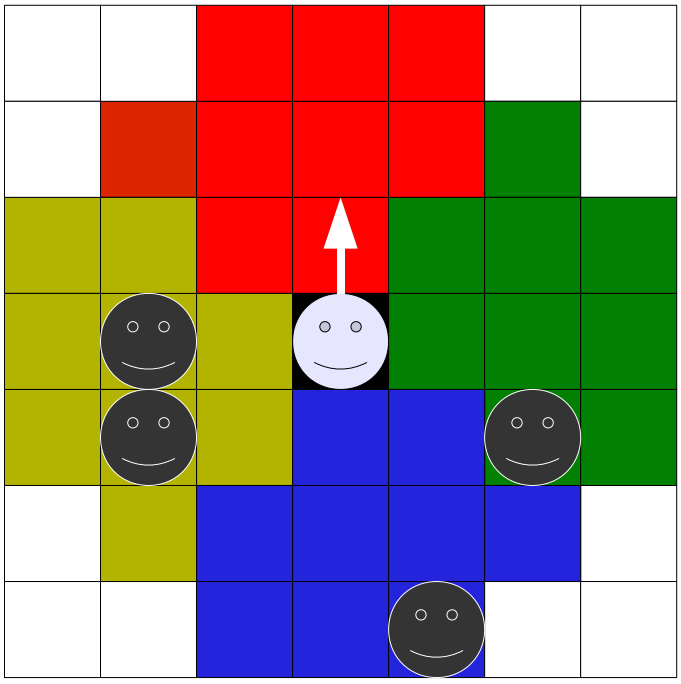

Abbildung 3.3: Agent bewegt sich von anderen Agenten weg (falls Zielagent nicht sichtbar)

3.5 Grundsätzliche Zielagententypen

Die Art der Bewegung des Zielagenten trägt grundsätzlich zur Schwierigkeit eines Szenarios bei. Gemeinsam haben alle Arten von Bewegungen des Zielagenten, dass, wenn kein freies Feld zur Verfügung steht, springt der Zielagent auf ein zufälliges freies Feld

auf dem Grid springt. Dies kommt einem Neustart gleich und ist notwendig um einer Verfälschung des Ergebnisses zu verhindern, das dadurch rühren kann, dass ein oder mehrere Agenten (zusammen mit eventuellen Hindernissen) bis zum Ende des Problems alle vier Bewegungsrichtungen des Zielagenten blockieren.

3.5.1 "Random Movement"

Wie "Randomized". Sind alle möglichen Felder belegt, wird wie oben beschrieben auf ein zufälliges Feld gesprungen.

3.5.2 "Total Random"

Der Zielagent springt zu einem zufälligen (freien) Feld auf dem Grid. 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 Zielagenten ist hier wenig hilfreich, ein Agent kann nicht einmal davon ausgehen, dass sich der Zielagent in der Nähe seiner Position der letzten Zeiteinheit befindet.

3.5.3 "One direction change"

Mit dieser Einstellung wird die der letzten Richtung entgegengesetzten Richtung aus der Menge der Auswahlmöglichkeiten entfernt und von den verbleibenden drei Richtungen (plus der Aktion "Stehenbleiben") 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 wie bei "Random Movement" auf ein zufälliges Feld gesprungen.

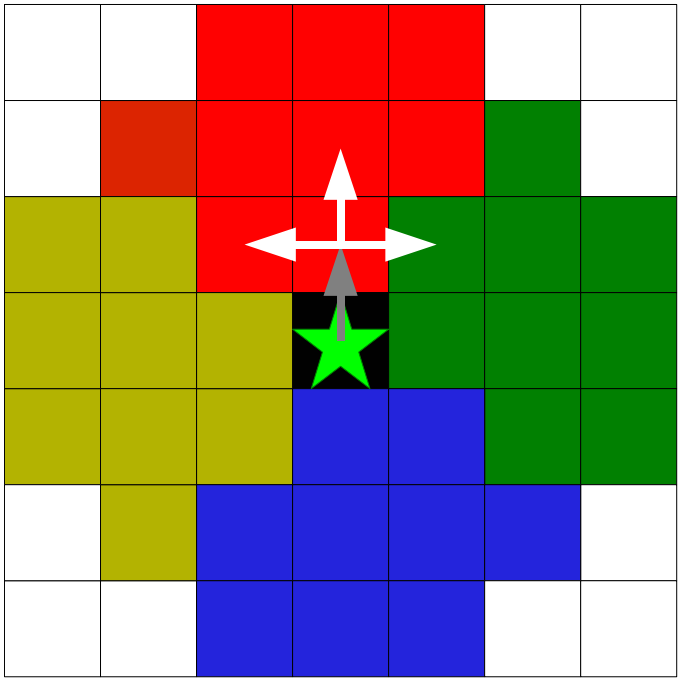

Abbildung 3.4: Zielagent macht pro Schritt maximal eine Richtungsänderung

3.5.4 “Always same direction”

Der Zielagent versucht immer Richtung Norden zu gehen. Ist das Zielfeld blockiert, wählt er ein zufälliges, angrenzendes, freies Feld im Westen oder Osten. Sind auch diese belegt, springt er wie oben auf ein zufälliges freies Feld. Schafft es der Zielagent innerhalb von einer bestimmten Zahl (Breite des Spielfelds) von Schritten nicht, einen weiteren Schritt nach Norden zu gehen, wird ebenfalls gesprungen um ein “Festhängen” an einem Hindernis zu vermeiden.

3.5.5 “Intelligent (Open)”

Der Zielagent versucht andere Agenten zu vermeiden. Bei der Wahl der Richtung werden alle Richtungen gestrichen, in denen sich ein anderer Agent befindet. Von den verbleibenden Richtungen werden mit 20% Wahrscheinlichkeit alle Richtungen gestrichen, in denen sich ein Hindernis befindet. Sind alle Richtungen gestrichen worden, bewegt der Zielagent

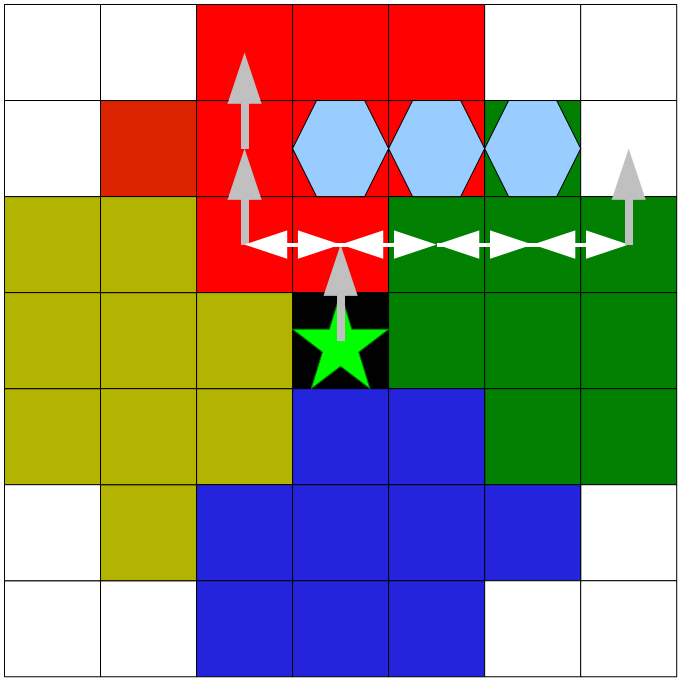

Abbildung 3.5: Zielagent bewegt sich, wenn möglich, immer nach Norden

sich zufällig. Sind alle Richtungen blockiert, springt er wie in den anderen Einstellungen auch.

3.5.6 "Intelligent (Hide)"

Der Zielagent vermeidet andere Agenten wie bei "Intelligent (Open)", streicht aber statt Richtungen mit Hindernissen Richtungen ohne Hindernisse mit 20% Wahrscheinlichkeit.

3.5.7 "LCS"

Eine LCS Implementierung die der der Implementierung eines LCS Agenten entspricht. Das Ziel des Zielagenten ist es hier aber, möglichst nicht andere Agenten zu überwachen (bzw. umgekehrt sich von anderen nicht überwachen zu lassen, d.h. nicht in die Überwachungsreichweite anderer Agenten zu geraten). Eine genaue Beschreibung folgt weiter unten.

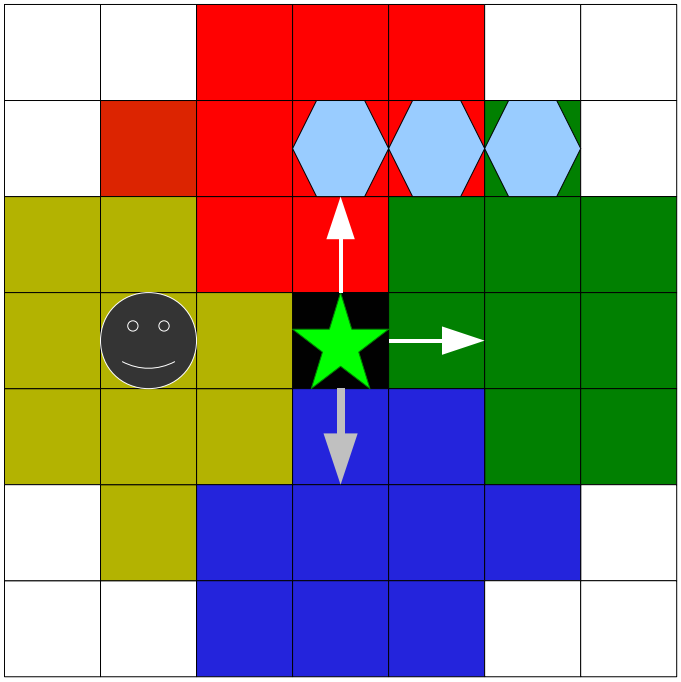

Abbildung 3.6: Zielagent bewegt sich von Agenten und mit bestimmter Wahrscheinlichkeit von Hindernissen weg

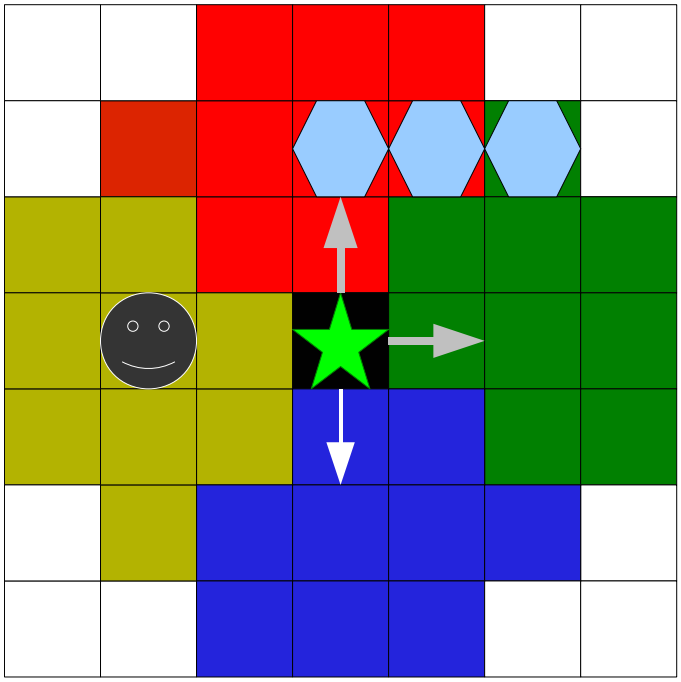

Abbildung 3.7: Zielagent bewegt sich von Agenten weg und mit bestimmter Wahrscheinlichkeit auf Hindernissen zu

Kapitel 4

Szenarien

Getestet werden eine Reihe von Szenarien (in Verbindung mit unterschiedlichen Werte für die Anzahl der Agenten, Größe des Spielfelds und Art und Geschwindigkeit der Zielagentenbewegung). Wesentliche Rolle spielt hier die Verteilung der Hindernisse. In den Darstellungen repräsentieren rote Felder Hindernisse, weiße Felder Agenten und das grüne Feld jeweils den Zielagenten. In Abbildung (4.1) ist ein Szenario ohne Hindernissen, mit zufälliger Verteilung der Agenten und zufälliger Position des Zielagenten dargestellt.

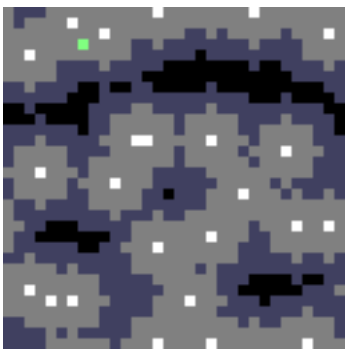

Abbildung 4.1: “Leeres Szenario” ohne Hindernisse

4.1 Zufälliges Szenario

Zwei Parameter bestimmen, wie das zufällige Szenario auszusehen hat, zum einen der Prozentsatz an Hindernissen an der Gesamtzahl der Felder, zum anderen der Grad inwieweit die Hindernisse zusammenhängen. Dieser Grad bestimmt nach Setzen eines Hindernisses die Wahrscheinlichkeit für jedes einzelne angrenzende freie Feld, dass dort sofort ein weiteres Hindernis gesetzt wird. Ein Wert von 0.0 ergibt somit ein völlig zufällig verteilte Menge an Hindernissen während ein Wert nahe 1.0 eine oder mehrere stark zusammenhängende Strukturen schafft. Wird der Prozentsatz an Hindernissen auf 0.0 gesetzt, dann werden Hindernisse vollständig deaktiviert. Als Optimierung werden in diesem Fall auch alle Sensorinformationen diesbezüglich deaktiviert. In Abbildung (4.2), (4.3), (4.4) und (4.5) werden Beispiele für zufällige Szenarien gegeben mit einem Anteil an Hindernissen von 5%, 10%, 20% bzw. 40% und jeweils einem Verknüpfungsfaktor von 1%, 50% bzw. 99%.

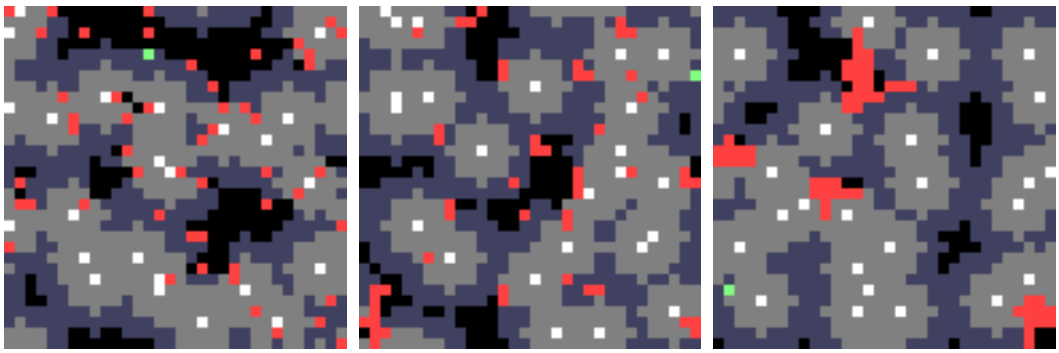

Abbildung 4.2: “Zufälliges Szenario” mit 5% zufällig verteilten Hindernissen und aufsteigendem Verknüpfungsfaktor, 0.01, 0.5 und 0.99

4.2 “Säulen Szenario”

Hier werden mit jeweils 7 Feldern Zwischenraum zwischen den Hindernissen (und mindestens 3 Feldern Zwischenraum zwischen Rand und den Hindernissen) Hindernisse verteilt. Idee ist, dass die Agenten eine kleine Orientierungshilfe besitzen aber gleichzeitig

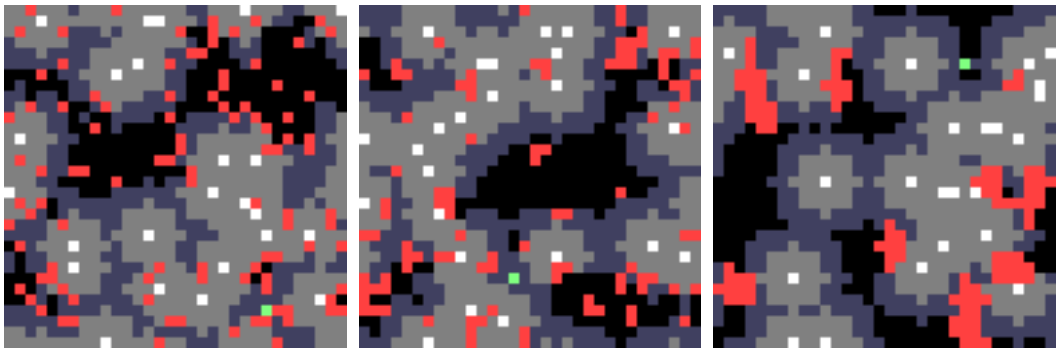

Abbildung 4.3: "Zufälliges Szenario" mit 10% zufällig verteilten Hindernissen und aufsteigendem Verknüpfungsfaktor, 0.01, 0.5 und 0.99

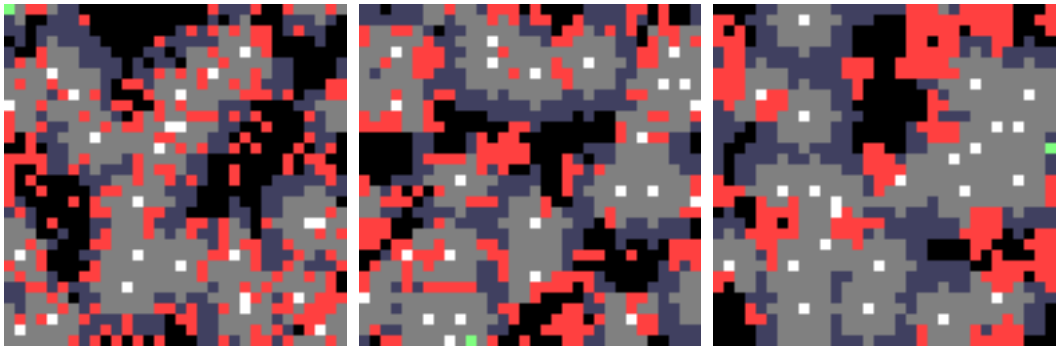

Abbildung 4.4: "Zufälliges Szenario" mit 20% zufällig verteilten Hindernissen und aufsteigendem Verknüpfungsfaktor, 0.01, 0.5 und 0.99

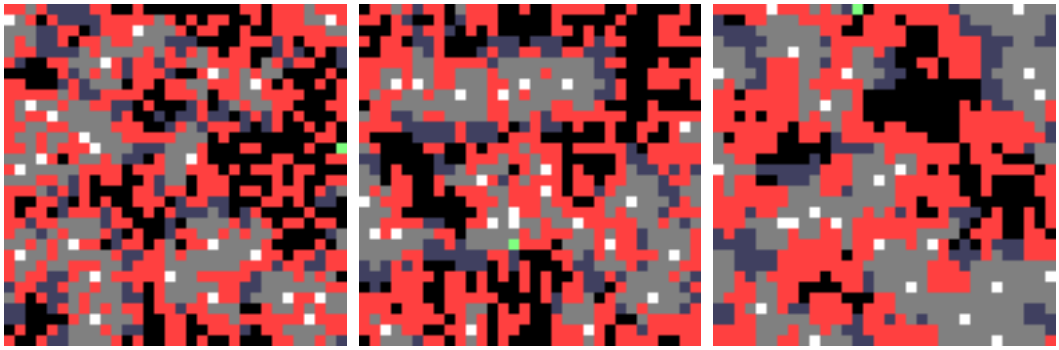

Abbildung 4.5: "Zufälliges Szenario" mit 40% zufällig verteilten Hindernissen und aufsteigendem Verknüpfungsfaktor, 0.01, 0.5 und 0.99

möglichst wenig Hindernisse verteilt werden. TODO Der Zielagent startet in der Mitte.

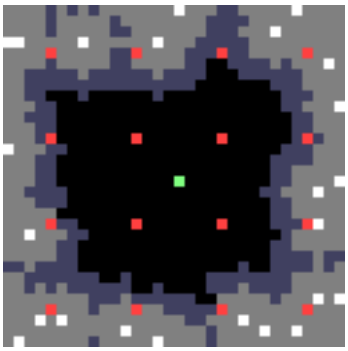

Abbildung 4.6: “Säulen Szenario” mit regelmäßig angeordneten Hindernissen, zufälliger Verteilung von Agenten am Rand und fester Startposition des Zielagenten im Zentrum

4.3 “Kreuz Szenario”

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

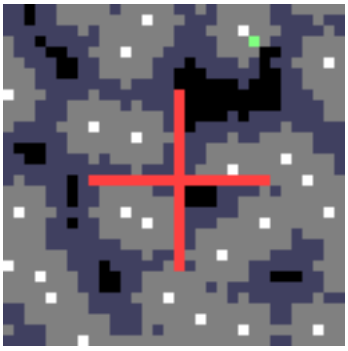

Abbildung 4.7: “Kreuz Szenario” mit zentrierter, kreuzförmiger Anordnung der Hindernisse, zufälliger Verteilung der Agenten und dem Zielagenten

4.4 "Raum Szenario"

In der Mitte des Grids wird ein Rechteck der halben Gesamthöhe und -breite des Grids erstellt, welches eine Öffnung von 4 Feldern Breite aufweist. Der Zielagent startet wie im Pillar Scenario in der Mitte, alle Agenten starten am Rand des Grids.

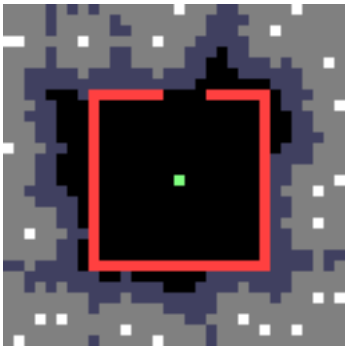

Abbildung 4.8: "Raum Szenario" mit zentrierter, quaderförmiger Anordnung der Hindernisse mit Öffnung oben, zufälliger Verteilung der Agenten am Rand und zu Beginn im Zentrum startendem Zielagenten

4.5 "Schwieriges Szenario"

Hier wird das Grid zum einen an der rechten Seite vollständig durch Hindernisse blockiert um den Torus zu halbieren. Alle Agenten starten 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 dem unteren Viertel.

4.6 "Irrgarten Szenario"

Der Code zur Generierung stammt aus [3].

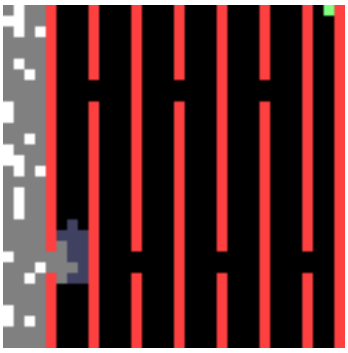

Abbildung 4.9: “Schwieriges Szenario” mit fester, wallartiger Verteilung von Hindernissen in regelmäßigen Abständen und mit Öffnungen, mit den Agenten mit zufälligem Startpunkt am linken Rand und dem Zielagenten mit festem Startpunkt rechts oben

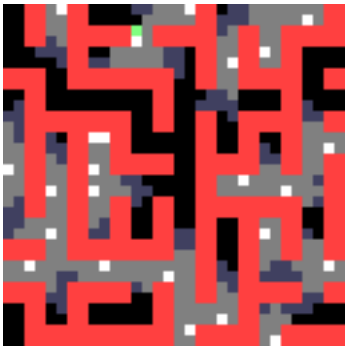

Abbildung 4.10: “Irrgarten Szenario” mit einer Anordnung von Hindernissen in der Art, dass es nur wenige Pfade gibt

Kapitel 5

Ablauf der Simulation

Bei Simulationen am Computer stellt sich sofort die Frage nach der Genauigkeit. Die Agenten werden bei dieser Simulation nacheinander abgearbeitet und bewegen sich auf einem diskreten Grid. Dies kann u.U. dazu führen, dass je nach Position in der Liste abzuarbeitender Agenten die Informationen über die Umgebung unterschiedlich alt sind. Die große Frage ist deshalb, in welcher Reihenfolge Sensordaten ermittelt, ausgewertet, Agenten bewegt, intern sich selbst bewertet und global die Qualität gemessen wird. Einzige Restriktionen sind, dass eine Aktion nach der Verarbeitung der Sensordaten stattfinden muss und eine Bewertung einer Aktion nach dessen Ausführung stattfinden muss. Ansonsten gibt es folgende Möglichkeiten:

5.1 Reihenfolge der Ausführung

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 Zielposition) anderer Agenten einbeziehen können, soll jeder Agent von den Sensorinformationen zu Beginn der Zeiteinheit ausgehen. Die Reihenfolge der Ausführung der Aktionen spielt eine Rolle, wenn mehrere Agenten sich auf das selbe Feld bewegen wollen. Arbeiten wir die Liste unserer Agenten einfach linear ab, haben vorne stehende Agenten eine höhere Wahrscheinlichkeit, dass ihre Aktion ausgeführt wird. Da es keine Veranlassung gibt, ihnen diesen Vorteil zu geben, werden Aktionen in zufälliger Reihenfolge abgearbeitet. Bezüglich der Bewegung ergibt sich hierbei eine weitere Frage, nämlich wie unterschiedliche Bewegungsgeschwindigkeiten behandelt werden sollen. Alle Agenten haben eine Einheitsgeschwindigkeit von maximal einem Feld pro Zeiteinheit, während sich der Zielagent je nach Szenario gleich eine ganze Anzahl von Feldern bewegen kann. Auch hier habe ich mich für eine zufällige Verteilung entschieden. Kann sich der Zielagent 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 der Zielagent diese weiteren Schritte festlegen soll. Hier soll ein Sonderfall eingeführt, so dass der Zielagent in einer Zeiteinheit mehrmals (n-mal) neue Sensordaten erfassen und sich für eine neue Aktion entscheiden kann.

5.2 Reihenfolge Rewardverteilung

Schließlich bleibt die Frage danach, wann geprüft werden soll, ob der Zielagent in Sicht ist, und wann somit der Reward verteilt wird. Da XCS in der Standardimplementation darauf ausgelegt ist, den Reward jeweils genau einer Aktion zuzuordnen, sollte der Reward

nicht bei jeder einzelnen Bewegung des Zielagenten überprüft und verteilt werden, sondern nur einmal pro Zeiteinheit. Außerdem soll der Einfachheit halber der Reward auch von binärer Natur sein ("Zielagent in Überwachungsreichweite" oder "Zielagent nicht in Überwachungsreichweite"), weshalb Zwischenzustände für den Reward (z.B. "War zwei von drei Zeitschritten in der Überwachungsreichweite" =¿ 2/3 Reward) ausgeschlossen werden sollen. Für den Reward gibt es somit folgende Möglichkeiten:

1. Rewards werden direkt nach der Ausführung einer einzelnen Aktion vergeben
2. Rewards werden nach Ausführung aller Aktionen der Agenten vergeben
3. Rewards werden nach Ausführung aller Aktionen des Zielagenten vergeben

Werden die Rewards sofort vergeben (Punkt 1), dann werden sich später noch wegbewegende bzw. sich später noch hin-bewegende Agenten nicht beachtet. Da die Agenten in zufälliger Reihenfolge abgearbeitet werden, würde das bedeuten, dass die Bewegung der restlichen (zufälligen) Anzahl von Agenten in den Reward nicht miteinbezogen wird. Selbiges gilt für Punkt 3. Auch der Zielagent kann Reward erhalten. Hierbei gibt es ebenfalls drei Möglichkeiten:

1. Reward direkt nach Ausführung des letzten Schritts
2. Reward nach Ausführung aller Agenten

Eine konkrete Antwort kann man auf diese zwei Fragen nicht geben, sie hängt nämlich 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 Agenten und Zielagenten in einem Durchlauf gemeinsam bewegen, stellt sich die Frage nicht, ob wir womöglich vor der Bewegung des Zielagenten die Qualität messen sollen. Eine Messung nach der Bewegung des Zielagenten würde diesem erlauben, sich vor jeder Messung optimal zu positionieren, was in einer

geringeren Qualität für den Algorithmus resultiert, da sich der Zielagent aus der Überwachungsreichweite anderer Agenten hinausbewegen kann. Letztlich ist es eine Frage der Problemstellung, denn eine Messung nach Bewegung des Zielagenten bedeutet letztlich, dass ein Agent, einen gerade aus seiner Überwachungsreichweite heraus laufenden Zielagenten in diesem Schritt nicht mehr überwachen kann. Da ein wesentlicher Bestandteil die Kooperation (und somit die Abdeckung des Grids anstatt dem Verfolgen des Zielagenten) sein soll, soll ein Bewertungskriterium sein, inwieweit der Einfluss des Zielagenten minimiert werden soll. Auch findet, wenn wir vom realistischen Fall ausgehen, die Bewegung des Zielagenten gleichzeitig mit allen anderen Agenten statt. Die Qualität wird somit nach der Bewegung des Zielagenten gemessen. Die Überlegung unterstreicht auch nochmal, dass es besser ist, den Zielagenten insgesamt wie einen normalen (aber sich mehrmals bewegenden) Agenten zu behandeln. Was den Zeitpunkt des Rewards betrifft, lautet die Hypothese, dass wir ein besseres Ergebnis erreichen, wenn man den Reward anhand der selben Momentaufnahme verteilt, anhand der wir auch die Qualität testen, d.h. dass Punkt 2 Punkt 1 überlegen ist. Dies bestätigt sich in Tests siehe TODO.

5.3 Zusammenfassung

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

1. Erfassung aller Sensordaten
2. Wahl der Aktion anhand der Regeln des jeweiligen Agenten
3. Ausführung der Aktion (in zufälliger Reihenfolge, der Zielagent führt nach dem ersten Schritt außerdem Schritt 1. und 2. für alle weiteren Schritte nochmals durch)
4. Bestimmung des Rewards
5. Bestimmen der Qualität dieser Zeiteinheit

Kapitel 6

Erste Analyse der Agenten ohne LCS

In diesem Kapitel sollen erste Analysen bezüglich der verwendeten Szenarien anhand des zufälligen Algorithmus, des Algorithmus mit einfacher Intelligenz ("Simple AI Agent") und des Algorithmus mit komplexerer Regeln ("Intelligent AI agent") angefertigt werden. Die Ergebnisse aus der Analyse werden eine Grundlage für die vergleichende Betrachtung der Agenten mit LCS Algorithmen dienen.

6.1 Total Random Goal Agent Movement

In allen Szenarien mit dieser Form der Bewegung des Zielagenten kommt es nur darauf an, dass die Agenten einen möglichst großen Bereich des Grids abdecken. In allen Standardszenarios zeigt sich, dass der "Simple AI Agent" sich wie erwartet nicht wesentlich vom zufälligen Agenten unterscheidet.

6.1.1 Ohne Hindernisse

Ohne Hindernisse gibt sich ein klares Bild. Das Ergebnis der einfachen KI ist etwas schlechter als der des zufälligen Agenten, da sich immer wenn mehrere Agenten den Zielagenten

in der selben Richtung in Sichtweite haben, sich mehrere Agenten in die selbe Richtung bewegen. Dies beeinträchtigt die zufällige Verteilung der Agenten auf dem Spielfeld und führt somit auch zu einer niedrigeren Abdeckung des Grids. Der intelligente Agent liegt hier sehr deutlich vorne, ein möglichst weiträumiges Verteilen auf dem Feld führt zum Erfolg, denn genau das wird mit dem völlig zufällig springenden Agenten getestet.

6.1.2 Zufällig verteilte Hindernisse

Hier ergeben sich bei allen Einstellungen des “Verknüpfungsfaktors” und ”Obstacle Factors” ebenfalls ein klares Bild, der intelligente Agent liegt wieder vorne, dann kommt allerdings schon der einfache Agent mit bis zu 10% zum zufälligen Agenten. Der wesentliche zweite Faktor ist hier, dass der einfache Agent, wenn er den Zielagenten 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

6.2 Random Neighbor und One Direction Change

Wesentlicher Punkt bei beiden Szenarien ist, dass der jetzige Ort des Zielagenten maximal zwei Felder (die Standardgeschwindigkeit des Zielagenten in den Tests) vom Ort in der vorangegangenen Zeiteinheit entfernt ist. Somit ist ein lokales Einfangen eher von Relevanz, wenn auch der Zielagent grundsätzlich schneller als andere Agenten ist.

Dementsprechend ist der einfache Agent bei einem Hindernis-Anteil von 0.0 bis 0.1 besser als alle anderen Agenten und dementsprechend ist bei allen Tests der zufällige Agent weit abgeschlagen. Ab einem Anteil von 0.2 liegt jedoch der intelligente Agent vorne. Dies liegt schlicht an der Zahl der Agenten relativ zur hindernisfreien Fläche, da sich die Agenten in möglichst großem Abstand zueinander positionieren.

Im Falle des “One Direction Change” bewegt sich der Agent im Grunde nur etwas schneller, da er es vermeidet, auf das ursprüngliche Feld zurückzukehren. TODO? Vielleicht sogar Random Neighbor raus...

6.3 Intelligent Open

6.4 Intelligent Hide

TODO:Beide gleiche Ergebnisse?Source prüfen

6.5 Always Same Direction

TODO

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 “Kommunikation” zu finden.

TODO Anpassung LCS an unterschiedliche Sichtreichweiten?

Kapitel 7

LCS

7.1 Einführung

Jeder Agent besitzt ein Learning Classifier System, welches auf dem von Butz TODO XCS basiert. Die Implementierung entspricht im Wesentlichen der Standardimplementation von Butz 2000, eine Besonderheit stellt allerdings die Problemdefinition dar. Keine der gegenwärtigen Implementationen und Varianten von XCS beschäftigen sich mit dynamischen Überwachungsszenarios sondern mit Szenarios, bei denen das Ziel in einer statischen Umgebung gefunden werden muss.

Im XCS-Multistepverfahren (TODOLiteratur) läuft ein Problem so lange bis ein positiver Reward aufgetreten ist und startet dann das Szenario neu. Bei einem Überwachungsszenario mit kontinuierlichem Reward ist das Multistepverfahren nicht anwendbar, das Szenario kann nicht neu gestartet werden, da sich die Agenten während des Laufs anpassen sollen. Ziel ist hier ja nicht, einen bestimmten Weg zu einem festen Ziel zu finden (wie z.B. bei WOODS TODO), sondern eine bestimmte Regelmenge zu erlernen, mit der eine möglichst gute, dauerhafte Überwachung stattfinden kann.

In der hier verwendeten Implementierung läuft das Problem deshalb einfach weiter.

Wesentlicher Unterschied zu XCS wird deshalb die Behandlung des Rewards sein.

In der Literatur (TODO) fehlen Arbeiten, in denen ein solches Szenario (ohne globale Steuerungseinheit oder Regeltausch) in Verbindung mit dem XCS behandelt wird. Diese Arbeit soll diese Lücke schließen und die Basis für weitere Arbeiten in dieser Richtung liefern.

7.2 Classifier

Ein LCS beinhaltet eine Reihe von Classifiern. Ein einzelner Classifier besteht im Wesentlichen aus fünf Teilen:

7.2.1 Fitness

Der Fitness Wert soll die allgemeine Genauigkeit des Classifiers repräsentieren und wird über die Zeit hinweg sukzessive an die beobachteten Rewards angepasst. TODO Wilson. Der Wertebereich verläuft zwischen 0.0 und 1.0 (maximale Genauigkeit).

7.2.2 Prediction

Der "Prediction"-Wert des Classifiers stellt die Höhe des Rewards dar, von dem der Classifier vermutet, dass er ihn bei der nächsten Vergabe des Rewards erhalten wird. Auch dieser Wert wird stetig angepasst.

7.2.3 Prediction Error

Der „Prediction-Error"-Wert soll die Genauigkeit des Classifiers bzgl. der Reward-Prediction (durchschnittliche Differenz zwischen Prediction und tatsächlichem Reward) repräsentieren. U.a. auf Basis dieses Werts wird der Fitness-Wert des Classifiers angepasst.

7.2.4 Aktion

Wird ein Classifier ausgewählt, wird eine bestimmte Aktion ausgeführt. In unserem Szenario entsprechen die Aktionsmöglichkeiten die der anderen Agenten, also 4 Bewegungsrichtungen plus eine “nichts-tun”-Aktion.

7.2.5 Kondition

Die Kondition gibt an, bei welchem Sensor-Input dieser Classifier ausgewählt werden kann.

7.3 Initialisierung

Aus dem Bereich naturanalogen Algorithmen weiss man (TODO Quelle), dass es beim Absuchen des Lösungsraums vorteilhaft sein kann nicht mit Null initialisierten Genotypen zu starten sondern mit einer zufälligen Population. Die Idee ist, dass dadurch die Zahl der Schritte zum Optimum verringert werden können (TODO genauer etvtl. Zitat). Beim XCS liegt die Sache nicht so einfach. Tests (TODO) haben gezeigt, dass die Qualität weder kurz noch langfristig besser ist. Der Grund dafür liegt darin, dass die Classifier zueinander bezüglich ihrer Wahrscheinlich in Konkurrenz stehen, aufgerufen und gelöscht zu werden. Viele der zufällig erstellten Classifier müssen also erst einmal beseitigt werden. Eine Beseitigung passiert mit einer Wahrscheinlichkeit abhängig von der Numerosity, der Fitness und der Prediction. Ein zufällig generierter Classifier mit hoher Fitness und Prediction dessen Condition in dem aktuellen Szenario selten auftritt (also die Fitness nur schwer verringert werden kann) benötigt lange, bis es aktualisiert wird und wird mit nur geringer Wahrscheinlichkeit gelöscht. Erst mit steigender Erfahrung des LCS und somit steigender Numerosity können solche Classifier eher verdrängt werden, in dieser Zeit können die entsprechenden Classifier aber bereits mittels Covering erstellt worden sein und somit

den womöglich existierenden Vorteil eines zufälligen Starts einholen. Bleibt also nur die Möglichkeit das ClassifierSet mit Classifiern mit zufälliger Kondition und zufälliger Aktion zu füllen, während man Fitness, Prediction und Prediction Error auf den Standardwerten eines neuen Classifiers belässt. Tests haben gezeigt (TODO), dass dadurch minimal bessere Ergebnisse erzielt werden, allerdings nur in Szenarien mit größerer Schrittzahl (> 100). Dies lässt sich darauf zurückführen, dass zum einen das anfänglich gefüllte ClassifierSet die MatchSets relativ groß werden lässt und die erstellten Classifier den Algorithmus anfänglich stören und zum anderen, dass TODO Begründung? $\Rightarrow$ Starten mit Classifiern mit zufälliger Kondition und Aktion aber Startinitialisierung der restlichen Werten.

7.4 Sensoren und Matching

In der hier verwendeten Implementierung gibt es zwar eine gewisse Vorverarbeitung der Sensordaten, im Wesentlichen müssen aber Kondition und Sensordaten übereinstimmen, damit der jeweilige Classifier ausgewählt wird. Konkret besteht die Kondition ebenfalls aus einem 9-stelligen Vektor, der allerdings nicht nur binäre sondern trinäre Werte besitzen kann.

7.4.1 Wildcards

Neben den zu den Sensordaten korrespondierenden Werten 0 und 1 gibt es noch einen dritten "dont-care"-Zustand "#", der anzeigen soll, dass beim Vergleich zwischen Kondition und Sensordaten diese Stelle ignoriert werden soll. Eine aus nur "dont-care" Werten bestehende Kondition würde somit bei der Auswahl immer in Betracht gezogen werden, da er auf alle Sensor-Inputs passt.

Beispiel: Kondition 1.#010.1#01 matched Sensordaten 1.0010.1001, 1.1010.1001, 1.0010.1101 und 1.1010.1101.

Die Benutzung von Wildcards erlaubt es dem LCS mehrere Classifier zu subsummieren, wodurch die Gesamtzahl der Classifier sinkt und somit Erfahrungen, die ein LCS Agent sammelt, nicht unbedingt doppelt gemacht werden müssen. Die dahinter stehende Annahme ist, dass es Situationen gibt, in denen ein Wert aus dem Sensor-Input für die Qualität der Entscheidung weniger entscheidend sein kann, als die Ersparnis durch das Zusammenlegen beider Classifier, d.h. dem Ignorieren dieses Inputs.

7.4.2 Matching von Sensordaten mit Classifiern

Beim Vergleich mit Sensordaten wir ebenfalls mit einem Vektor wie bei B erglichen. Entscheidend beim Vergleich ist hier aber nicht, dass beide Vektoren identisch sind, sondern, dass der Classifier "matched". Ein Element des Bedingungsvektors kann 3 Zustände einnehmen. 0, 1 und #. # beinhaltet beide Zustände 0 und 1.

Den dritten verwendeten Vergleich zwischen Bedingungsvektoren gibt es bei der Subsumation (der Kinder zu ihren Eltern und des gesamten ActionSets siehe (7.5.5). Ein Classifier subsumiert einen anderen Classifier, wenn er ihn beinhaltet, aber nicht identisch mit ihm ist, also allgemeiner ist.

7.4.3 Drehungen

Ein Classifier besteht aus dem Bedingungsvektor

$$(gx_0x_1x_2x_3)$$

(bzw.

$$(gx_0x_1x_2x_3y_0y_1y_2y_3)$$

für den Fall mit Hindernissen) und der Aktion a.

Eine wesentliche Vereinfachung, die angenommen wird, ist, dass angenommen wird,

dass eine Aktion in einer anderen, in 90 Grad Schritten gedrehten Umwelt, gleiche Güte besitzt. In einer statischen Umgebung ist dies nicht unbedingt der Fall, ohne diese Vereinfachung könnte sich ein Agent einfacher zurechtfinden, wie TODO (keine Drehung, 1 Problem pro Experiment) diese Testläufe zeigen. Da wir uns aber auf den dynamischen Fall konzentrieren und durch diese Vereinfachung eine signifikante Verkleinerung des Suchraums erreichen, benutzen wir die Vereinfachung. Im Algorithmus betrifft dies primär den Vergleich von Classifiern untereinander und mit dem Sensorstatus.

7.4.4 Äquivalenz von Classifiern

Ein Classifier A ist identisch mit einem Classifier B wenn gilt:

1. $A_g = B_g$
2. Falls $A_g = 0$:
   (a) Es gibt ein i für das gilt:

$$(A_{x_{0+i \bmod 4}} A_{x_{1+i \bmod 4}} A_{x_{2+i \bmod 4}} A_{x_{3+i \bmod 4}}) = (B_{x_0} B_{x_1} B_{x_2} B_{x_3})$$

   (b) Mit Hindernissen muss für dieses i außerdem gelten:
      i. $(A_{y_{0+i \bmod 4}} A_{y_{1+i \bmod 4}} A_{y_{2+i \bmod 4}} A_{y_{3+i \bmod 4}}) = (B_{y_0} B_{y_1} B_{y_2} B_{y_3})$
      ii. Falls $A_a = \text{NO\_ACTION}$:

$$A_a = B_a = \text{NO\_ACTION}$$

      iii. Falls $A_a \neq \text{NO\_ACTION}$:

$$(A_a + i) \bmod 4 = B_a$$

3. Falls $A_g = 1$:

   (a) Identische Aktion:

   $$A_a = B_a$$

   (b) Identische Agentensensoren:

   $$(A_{x_0} A_{x_1} A_{x_2} A_{x_3}) = (B_{x_0} B_{x_1} B_{x_2} B_{x_3})$$

   (c) Mit Hindernissen muss außerdem:

   $$(A_{y_0} A_{y_1} A_{y_2} A_{y_3}) = (B_{y_0} B_{y_1} B_{y_2} B_{y_3})$$

Die Gleichheit zwischen Vektoren gilt, wenn paarweise Gleichheit zwischen den Elementen besteht, also $A_{x_i} = B_{x_i}$ für $i = 0 \ldots 3$ und $A_{y_i} = B_{y_i}$ für $i = 0 \ldots 3$ im Fall mit Hindernissen.

7.4.5 Test Drehungen

Mit aktivierter Optimierung der Drehungen wird in bestimmten Szenarien bis zu einer bestimmten Schrittzahl ein besseres Ergebnis erreicht, d.h. die Konvergenzgeschwindigkeit wird erhöht. In längeren Durchläufen mit größerer Schrittzahl werden jedoch bessere Ergebnisse ohne der Optimierung erreicht. Der Zugewinn folgt aus einer durch die zusätzliche lokale Information die in den Classifiern gespeichert werden kann.

In einer zukünftigen Implementierung sollte überlegt werden, wie bei räumlichen Aufgaben für Classifier ein LCS sowohl drehbare als auch nicht-drehbare Classifier speichern kann. Dies kann in einer Form des erwähnten Wildcard-Systems bewerkstelligt werden. Dadurch und zusammen mit der Subsummation könnten beide Vorteile vereint werden

und sowohl lokale, als auch allgemeine Information gesammelt werden. TODOvielleicht doch noch rein?

7.5 Implementation

Für die Auswahl der tatsächlich auszuführenden Aktion müssen dann alle (genotypischen) Classifier in AppliedClassifier umgewandelt werden, die jeweils genau auf eine bestimmte Aktion verweisen. Sichtwe

Zur Durchführung und Forschung war es notwendig, den kompletten XCS Algorithmus nachvollziehen zu können. TODO

Besonders die Verwaltung der Numerosity und die Verwendung des maxPrediction TODO

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.

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

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.

7.5.1 ActionSets

7.5.2 AppliedClassifierSet

7.5.3 Numerosity

TODO Beschreibung In der originalen Implementierung von Butz 2000 TODO war die Behandlung der Numerosity stark optimiert auf den Fall des einmaligen Rewards ohne Protokollierung der bisherigen ActionSets. Nach einer missglückten ersten Implementierung – der Wert der numerositySum eines ClassifierSets stimmte nicht mehr mit der Summe der numerosity-Werte der enthaltenen Classifiers überein – entschloss ich mich den entsprechenden Code neuzuschreiben. Hierbei wurde jedem Classifier eine Liste der Eltern, d.h. der jeweiligen ActionSets, zugewiesen. Wird ein Microclassifier entfernt, wird dann lediglich die Änderungsfunktion der Numerosity des Classifiers aufgerufen, der dann wiederum die NumerositySum der jeweiligen Eltern anpasst. Dies macht einige Optimierungen rückgängig, erspart aber sehr viel Umstände, die NumerositySum immer auf den aktuellen Stand zu halten. Positiver Nebeneffekt ist, dass man dadurch leicht auf die Menge der ActionSets zugreifen kann, denen ein Classifier angehört. Inwiefern das tatsächlich von Nutzen sein kann ist offen. TODO Kernstück des LCS Agenten:

7.5.4 Covering

Die Implementation entspricht im Wesentlichen dem Original, es wurde aber im Hinblick auf eine klare Code-Struktur eine Optimierung entfernt und Code zur Behandlung von Drehungen hinzugefügt. Im Original wird die Erstellung des MatchSets gleichzeitig mit dem Covering ausgeführt, wodurch möglicherweise Zeit gespart wird, während in dieser Implementierung es in zwei Funktionen aufgeteilt wurde. TODOBEschreibung Bezüglich der Drehung musste eine Schleife eingefügt werden, die die abgedeckten Aktionen mit allen, inklusive durch Drehung entstandenen, Aktionen eines Classifiers aktualisiert. Ein

einzelner Classifier kann also mehrere Aktionen abdecken, beispielsweise kann “0-0000-0000-¿0” bei der Sensoreingabe “0-0000-0000” alle vier Aktionen bereits abdecken. TODO besseres Beispiel.

In meiner ersten Version hatte ich alle ungültigen Aktionen von vornherein für das Covern ausgeschlossen. Eine ungültige Aktion ist beispielsweise ein Laufen gegen ein Hindernis oder einen Agenten. Alleine dadurch verbesserte sich die Leistung um etwa 0.5%. Das lässt sich darauf zurückführen, dass die Sensoren eines Agenten eigentlich nur feststellen können, ob ein anderer Agent in Sichtweite ist, nicht aber in welcher Entfernung. Für die eigentlichen Ergebnisse wurde die dafür verantwortliche Methode wieder entfernt.

Außerdem ist ein Fehler der originalen XCS Implementation behoben. Wenn neue Classifier beim Covering hinzugefügt werden, wird ihre anfängliche ActionSetSize auf die numerositySum des matchSets gesetzt. Einen Grund dafür gibt es nicht, in meiner Implementation setze ich deshalb den Wert auf die anfängliche Größe des actionSets. Beim Aufruf des Covering-Algorithmus weiss man schon, wie groß

TODO nein!

7.5.5 Subsummation

Ein Problem ist hier natürlich die Sicherstellung, dass Informationen nicht verloren gehen. Andere Arbeiten befassen sich mit der Untersuchung von der Benutzung von Wildcards. In der hier verwendeten Implementation übernehme ich unverändert die Implementation aus der Literatur.

7.5.6 Evolutionärer Algorithmus

Der genetische Algorithmus wurde im Wesentlichen nicht verändert. Da aber die Binärsensoren eng zusammenhängen, werden beim Crossing Over zwei feste Stellen für Crossing Over benutzt. Die Stellen trennen somit die 3 Gene, Zielagent, Agenten und feste Hinder-

nisse.

Im Test erbrachte die Benutzung des Algorithmus wenig Unterschied. TODO Erklärung

7.6 Ablauf eines LCS

1. Bei der Auswahl einer Aktion werden zuerst einmal alle Classifier mit denjenigen Konditionen gesucht, die auf die aktuellen Sensordaten passen. Diese bilden dann das MatchSet.

2. Im nächsten Schritt wählen wir einen Classifier aus diesem MatchSet aus und speichern dessen Aktion.

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

7.6.1 Exploration und Exploitation

Die Auswahl in Punkt (2) kann auf verschiedene Weise erfolgen. In XCS gibt es drei Möglichkeiten der Auswahl, wobei man zusätzlich noch während eines Experiments zwischen den Möglichkeiten wechseln kann.

1. “exploit”: Es wird immer ein Classifier mit dem höchsten Produkt aus fitness $*$ prediction gewählt

2. “explore”: Es wird mit Hilfe einer Roulette-Auswahl, welche anhand des Produkts aus fitess $*$ prediction geordnet ist, ein Classifier ausgewählt

3. “random-explore”: Es wird ein zufälliger Classifier gewählt, unabhängig von fitness oder prediction.

In der XCS Implementierung sind alle drei Möglichkeiten zu finden, standardmäßig ist jedoch Möglichkeit 2 zugunsten von Möglichkeit 3 deaktiviert. Bei einem dynamischen Überwachungsszenario ist es im Vergleich zu standardmäßigen statischen Szenarien weder nötig noch hilfreich "random-explore" zu nutzen. Die Idee für "random-explore" in einem statischen Szenario ist, dass man vermeiden möchte, dass das LCS immer wieder die selben Entscheidungen trifft und somit immer wieder den selben Umweltreizen ausgesetzt ist, was wiederum zu immer wieder gleichen Entscheidungen führt usw. Bei einem dynamischen Szenario ergibt sich das Problem nicht, andere Agenten und das Ziel sind in stetiger Bewegung und der eigene Startpunkt ist nicht fixiert. Das gewichtete "explore" oder gar nur "exploit" sollten deshalb ausreichen, wenn nicht sogar wesentlich besser abschneiden.

7.6.2 Wechsel zwischen Exploration und Exploitation

In der Standardimplementierung von XCS wird zwischen "exploit" und "random-explore" nach jedem Erreichen des Ziels umgeschalten.

Möglichkeit (3.) 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.

Hierbei gibt es mehrere verschiedene Möglichkeiten:

1. Immer "exploit"
2. Immer "explore"
3. Abwechselnd "explore" und "exploit"
4. Zufällig entweder "explore" oder "exploit" (50% Wahrscheinlichkeit jeweils)
5. Erste Hälfte eines jeden Problems nur "explore", dann nur "exploit"

6. Wie (4.), während des Problems allerdings eine lineare Abnahme der Wahrscheinlichkeit von “explore” und eine lineare Zunahme der Wahrscheinlichkeit von “exploit”

7. “exploit” wenn Ziel in Sichtweite, “explore” sonst

TODO

Während es in der

TODO Vergleich weiter unten

Kapitel 8

LCS Varianten

8.1 Multistepverfahren

TODO Als Vergleich wurde das bekannte Verfahren fast unverändert übernommen. Wie weiter oben erwähnt wird das Szenario bei einem positiven Reward aber nicht neugestartet.

Idee ist, dass der Reward, den eine Aktion (bzw. das jeweils zugehörige ActionSet) erhält, vom erwarteten Reward der folgenden Aktion abhängt. Somit wird, rückführend vom Ziel, der Reward schrittweise an vorgehende Aktionen verteilt und somit das Ziel schneller gefunden

TODO Quelle

reward = 0 ? Gebe maxPrediction des nächsten Zugs reward = 1 ? Gebe maxPrediction 0, reward = 1

In jedem Schritt wird das vorherige ActionSet durch maxPrediction bzw. reward belohnt TODO

Program 1 Erstes Kernstück des 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);
      return;
    }
    prevActionSet = lastActionSet;
    lastReward = reward;
  }
```

Program 2 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;

  /**
   * Ereignis, 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);
      }
    }
  }
```

Program 3 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);
  }
```

8.2 LCS Variante ohne Kommunikation

Die Hypothese bei der Aufstellung dieser Variante des XCS-Algorithmus ist im Grunde die selbe wie beim einfachen Multistepverfahren, nämlich dass die Kombination mehrerer Aktionen zum Ziel führt. Beim Multistepverfahren besteht die wesentliche Verbindung zwischen den ActionSets jeweils zwischen zwei direkt aufeinanderfolgenden ActionSets. In einer statischen Umgebung kann dadurch über mehrere (identische) Probleme hinweg eine optimale Einstellung (fitness, prediction) für die Classifier gefunden werden. Bei der veränderten LCS Variante ist die Verbindung zwischen den ActionSets direkt die zeitliche Nähe zum Ziel. ActionSets von jedem Schritt seit dem Erreichen des letzten Ziels werden gespeichert bis das Ziel erreicht wurde und dann in Abhängigkeit des Alters mit absteigendem bzw. aufsteigendem Reward bewertet.

$r(a)$ bezeichnet den Reward für das ActionSet mit Alter a.

Bei linearer Rewardvergabe:

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

bzw. bei quadratischer Rewardvergabe:

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

In Tests ergab sich für die quadratische Rewardvergabe ein minimal besseres Ergebnis (TODO zeigen), im weiteren Verlauf, insbesondere in den Grafiken, werde ich mich auf die lineare Rewardvergabe beschränken um eine verständliche Darstellung zu ermöglichen.

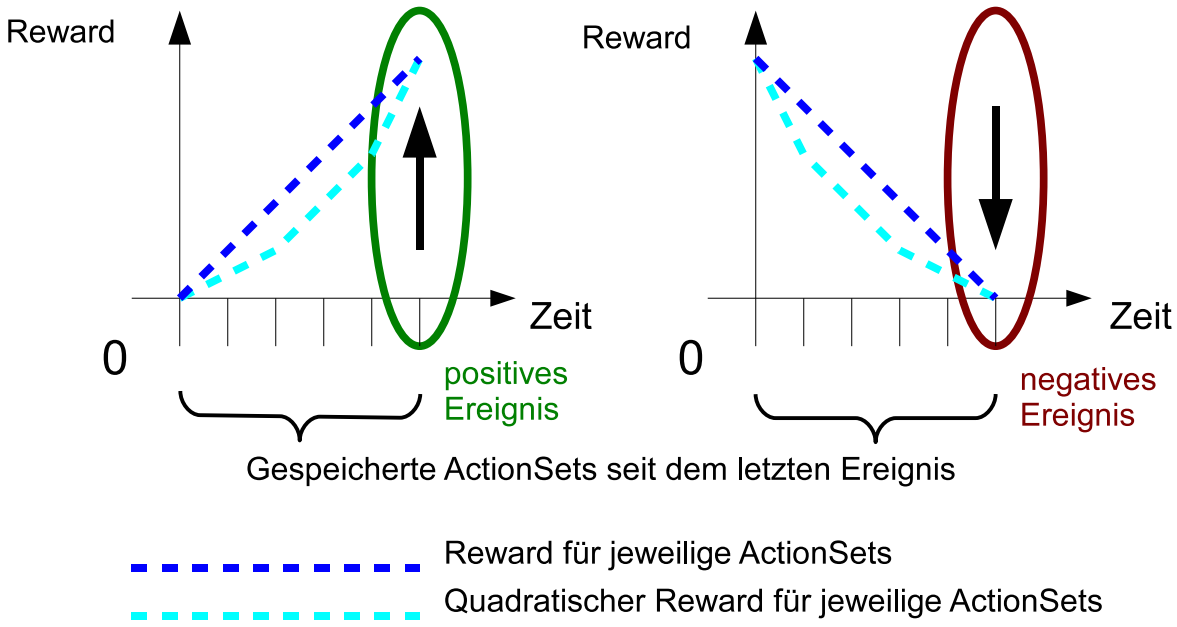

Abbildung 8.1: Schematische Darstellung der (quadratischen) Rewardverteilung an gespeicherte ActionSets bei einem positiven bzw. negativen Ereignis

8.3 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 Vergleich zum vorangegangenen Zeitschritt verändert hat, also wenn der Zielagent sich in Übertragungsreichweite bzw. aus ihr heraus bewegt hat (siehe (8.2)).

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 (8.3)). 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)

8.3.1 Test der verschiedenen Exploration-Modi

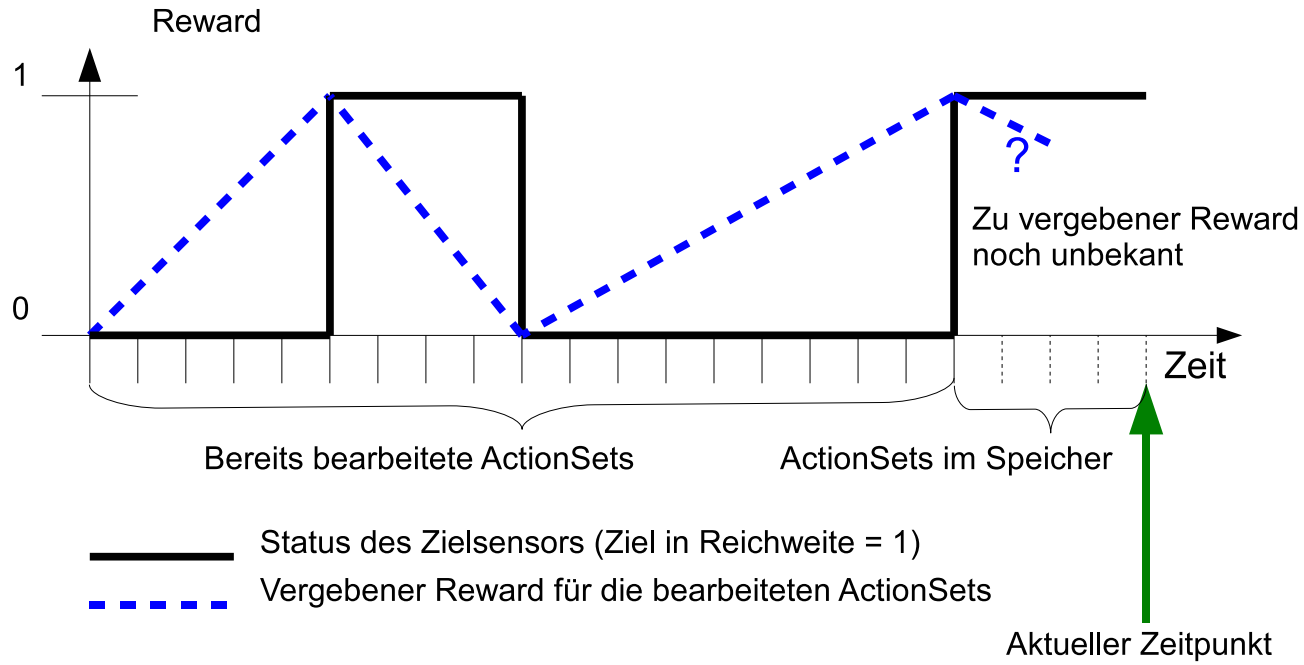

Abbildung 8.2: Schematische Darstellung der zeitlichen Rewardverteilung an ActionSets nach mehreren positiven und negativen Ereignissen und der Speicherung der letzten ActionSets

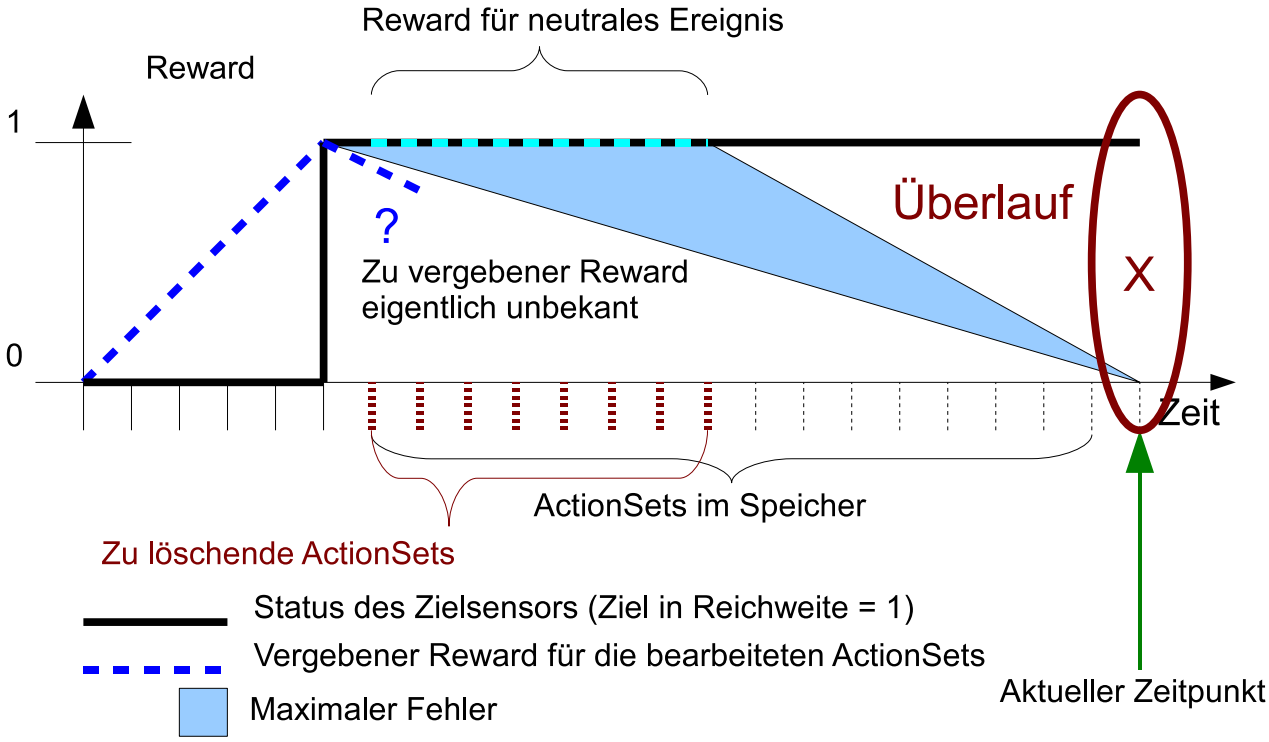

Abbildung 8.3: Schematische Darstellung der Rewardverteilung an ActionSets bei einem neutralen Ereignis

Program 4 Erstes Kernstück des LCS-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;
  }
```

Program 5 Zweites Kernstück des LCS-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();
   }
```

Program 6 Drittes Kernstück des LCS-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();
    }
  }
```

Kapitel 9

Analyse LCS und Multistep

9.0.2 Vergleich Multistep / LCS

Szenarien, Parameter.

Kapitel 10

Verzögertes LCS

10.1 Verzögerter Reward

Der wesentliche Unterschied zur ersten LCS Variante ist, dass jeglicher Reward und die zugehörigen Faktoren lediglich zusammen mit den ActionSets in einem HistoricActionSet gespeichert werden und in jedem Schritt immer nur das allerletzte ActionSet mit dem entsprechenden Reward aktualisiert wird. TODO genauer calculateReward identisch

Program 7 Zweites Kernstück des verzögerten LCS-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);
       }

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

    }
```

Program 8 Auszug aus dem dritten Kernstück des verzögerten LCS-Algorithmus (calculateNextMove)

```
/**
 * Der erste Teil der Funktion ist identisch mit dem calculateNextMove
 * der LCS 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();
      double max_prediction = historicActionSet.getFirst().getBestValue();
      last.processReward(max_prediction);
    }
  }
```

Program 9 Auszug aus dem vierten Kernstück des verzögerten LCS-Algorithmus (Verarbeitung des Rewards)

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

  public void processReward(double max_prediction) {

    double max = 0.0;
    double max_factor = 0.0;
  /**
   * Finde das größte reward / factor Paar TODO Verbessern
   */
    for(RewardHelper r : reward) {
      if(r.reward >= max && r.factor >= max_factor) {
        max = r.reward;
        max_factor = r.factor;
      }
    }
  /**
   * Aktualisiere den Reward mit den ermittelten Werten und dem
   * übergebenen maxPrediction Wert
   */
    actionClassifierSet.updateReward(max, max_prediction, max_factor);
  }
```

Kapitel 11

LCS mit Kommunikation

11.1 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.

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

11.2 Ablauf

Jeder Reward, der aus einem normalen Event generiert wird, wird unter Umständen an alle anderen Agenten weitergegeben. Wie ein solcher sogenannter "externer Reward" von diesen Agenten aufgefasst wird, hängt von der jeweiligen Kommunikationsvariante ab, die weiter unten besprochen werden.

Durch eine gemeinsame Schnittstelle erhält jeder Agent den Reward zusammen mit dem Factor. 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, 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 einen 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.

11.2.1 Fälle

In (**??**) wurden 4 verschiedene mögliche Situationen für einen einzelnen Agenten dargestellt. In Verbindung mit externen Rewards ergeben sich einige neue Situationen, nämlich ob sich der Zielagent in Überwachungsreichweite anderer Agenten befindet und wie dies im letzten Zeitschritt ausgesehen hat. Zusammen mit den ursprünglichen vier Möglichkeiten ergeben sich folgende 16 Möglichkeiten: Hier die erweiterte Übersicht, welche Arten von Events im Rahmen eines Multiagentensszenarios auftreten können:

-
-
-

TODO nochmal überlegen...

Ziel befindet sich von anderen Agenten in Sicht: Time-out (Ziel in Sicht) Time-out (Ziel nicht in Sicht)

Ziel kommt in Sicht Ziel verschwindet aus Sicht

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 befindet sich von anderen Agenten nicht in Sicht: Time-out (Ziel in Sicht) Time-out (Ziel nicht in Sicht)

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

11.3 Kommunikationsvarianten

Allen hier vorgestellten Kommunikationsvarianten ist gemeinsam, dass sie einen Faktor berechnen, nach denen sie den externen Reward, den ihnen ein anderer Agent übermittelt hat, bewerten. Der Faktor beeinflusst alle Verwendungen des Parameters beta (welcher die Lernrate bestimmt) der mit dem Faktor gewichtet wird. 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. 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 Überwachungsreichweite zu bekommen anstatt zu kooperieren, also das Gebiet des Grids möglichst großräumig abzudecken.

11.3.1 No external reward

Mit dieser Variante wird der Faktor fest auf 0.0 gesetzt ist Kommunikation deaktiviert.

11.3.2 Reward all equally

Mit dieser Variante wird der Faktor 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, es fehlt die Verbindung zwischen Classifier und Reward. In Tests haben sich dennoch in bestimmten Fällen deutlich bessere Ergebnisse gezeigt als im Fall ohne Kommunikation. Dies ist wahrscheinlich darauf zurückzuführen, dass in dem Fall die Kartengröße und Zielagentengeschwindigkeit 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 zu einer Verbesserung führen, dabei ist der Punkt aber nicht, dass Informationen ausgetauscht werden, sondern, dass obiges Verhältnis gedreht wird. TODO genauer

Für die Auswahl geeigneter Tests sollten die Szenario-Parameter also möglichst so gewählt werden, dass "Reward all equally" keinen signifikanten Vorteil bringt.

11.3.3 Egoism factor

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 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 Kommunikationsaufwand ist hier nur minimal größer, neben dem Reward muss der Egoismus-Faktor übertragen 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

11.3.4 Simple relation

Eine dritte Implementation vergleicht die Classifiers direkt. Alle Classifier des Agenten, der den Reward weitergibt, die ausreichend Erfahrung gesammelt haben und ausreichend genau ist (Experience und geringes PredictionError, 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 des ClassifierSets des Agenten, der den Reward vergibt, bestehend aus Classifiern, deren experience größer als thetaSubsumer und dessen predictionError kleiner als epsilon0 ist. pSet2 ist die gleiche Teilmenge, allerdings des Agenten, 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.

Nachteil: Übertragung von Classifier-Daten notwendig

Weitere Implementationen sind denkbar, bei denen komplexere Vergleiche und Analysen durchgeführt werden. TODO

11.4 Realistischer Fall mit Kommunikationsrestriktionen

Wann immer ein Reward an einen Agenten verteilt wird, kann es sinnvoll sein, diesen Reward an andere Agenten weiterzugeben. Bisher wurde der Fall betrachtet, dass Kommunikation mit beliebiger Reichweite stattfinden kann. Dies ist natürlich kein realistisches Szenario. Bedenkt man jedoch, dass die Kommunikationsreichweite zumindest ausreichend groß ist um nahe Agenten zu erreichen, so kann man argumentieren, dass man dadurch ein Kommunikationsnetzwerk aufbauen kann, in dem jeder Agent jeden anderen Agenten erreichen kann. Bei ausreichender Agentenzahl relativ zur freien Fläche fallen dadurch nur vereinzelte Agenten aus dem Netz, was der Effektivität der Agentengruppe nur geringfügig schadet. Stehen die Agenten nicht indirekt andauernd miteinander in Kontakt (mit anderen Agenten als Proxy), sondern muß die Information zum Teil durch aktive Bewegungen der Agenten transportiert werden, tritt eine Zeitverzögerung auf. Auch kann die benötigte Bandbreite die verfügbare übersteigen, was ebenfalls Zeit benötigt. Im realistischen Fall ist also davon auszugehen, dass jede Kommunikation erst mit einer gewissen Verzögerung ausgeführt wird.

TODO lag einführen...

11.5 Weitergabe des Rewards

11.6 Bewertung Kommunikation:

Die Vorteile, die man durch Kommunikation erzielen kann, hängt stark durch das 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?

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

Kapitel 12

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

12.1 Beschreibung des Konfigurationsprogramms

Abbildung 12.1: 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] Larry Bull: *A Simple Accuracy-Based Learning Classifier System*, `http://www2.cmp.uea.ac.uk/~it/ycs/ycs.pdf`

[3] 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`

Kapitel 13

Zusammenfassung, Ergebnis und Ausblick

13.1 Zusammenfassung

This is a summary section

13.2 Ergebnis

13.3 Ausblick

THE END...

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