Entwicklung einer Planning Engine
und
Integration in das living agents framework

Diplomarbeit

zur Erlangung des akademischen Grades
Diplom Informatiker (FH)
an der
Fachhochschule für Technik und Wirtschaft Furtwangen

Fachbereich Informatik
Studiengang Computer Networking
Wintersemester 2001/2002

Betreuer
Prof. Dr. Sabine Mahling-Ennaoui
Dr. Klaus Dorer

Vorgelegt von
Alexander Schneider

Erstellt bei der Firma living systems AG

Furtwangen, 28. Februar 2002

Danksagung

An dieser Stelle möchte ich mich bei meinen Betreuern Prof. Dr. Sabine Mahling-Ennaoui von der Fachhochschule Furtwangen und Dr. Klaus Dorer von der living systems AG für wertvolle Hinweise und Anregungen zu dieser Arbeit bedanken.

Darüber hinaus danke ich all jenen Kommilitonen, die durch Diskussionen und Ideen Denkanstöße zur vorliegenden Arbeit gaben. Dem Forschungsteam der living systems AG will ich auf diesem Weg für das sehr angenehme Arbeitsklima danken.

Besonderer Dank gilt meinen Eltern für die unermüdliche Unterstützung in vielerlei Hinsicht während des ganzen Studiums. Der Abschluss des Studiums wäre ohne sie auf diese Weise nicht möglich gewesen. Für das entgegengebrachte Verständnis und die Rücksichtnahme möchte ich mich bei meinen Freunden bedanken.

Ich erkläre hiermit an Eides statt, dass ich die vorliegende Diplomarbeit selbständig und ohne unzulässige fremde Hilfe angefertigt habe.

Die verwendeten Quellen sind vollständig zitiert.

Furtwangen, den _____

_____

Alexander Schneider
Kosbachblick 6
78147 Vöhrenbach

Abstract

Die Diplomarbeit behandelt die Entwicklung einer Planning Engine zur Handlungskontrolle von Software Agenten auf der Basis der von der Firma living systems AG entwickelten Agentenplattform. In der Diplomarbeit wird dazu auf verschiedene existierende Planungsalgorithmen und deren Speicherplatz- und Laufzeitverhalten eingegangen. Außerdem werden bereits vorhandene und weiter geplante Techniken zur Handlungskontrolle erläutert. Die Handlungsplanung wird in Bezug auf Einsatzgebiete und in Bezug auf Vor- und Nachteile der unterschiedlichen Technologien zur Handlungskontrolle beleuchtet. Schließlich wird die Implementierung der Planning Engine, sowie die Integration in die Agentenplattform erläutert und das empirische Verhalten bei der Handlungsplanung in Bezug auf Speicherbedarf und Zeitbedarf betrachtet.

Inhaltsverzeichnis

Anhang

Kapitel 1

Einführung

1.1 Aufgabenstellung

Im Rahmen der Diplomarbeit soll ein Planungssystem entwickelt werden und in das Produkt „living markets“ der living systems AG eingebunden werden.

Dazu muss man sich im Wesentlichen mit drei Dingen auseinander setzen:

a.) Planungssysteme

- Was sind Planungssysteme?
- Welche Arten von Planungssystemen gibt es?
- Wie funktionieren Planer?
- Wozu werden Planer gebraucht?

b.) Java-basierte Software-Agenten

- Was sind Software-Agenten?
- Welche Vorteile bieten Software-Agenten?
- Wie funktioniert das Agentensystem der living systems AG?

c.) Integration von Planungssystemen in eine Agentenplattform

- Inwiefern lässt sich ein Planungssystem in das Produkt integrieren?
- Welche Potenziale hat ein Planungssystem im Produkt der living systems AG?
- Welches Planungsverfahren ist am besten geeignet für die Integration?

1.2 Zielsetzung

Auf Initiative der living systems AG wird in dieser Diplomarbeit das Forschungsgebiet der Handlungsplanung im Zusammenspiel mit der bereits eingesetzten Agenten-Technologie untersucht. Die Implementierung entsprechender Komponenten respektive Erweiterungen sollen das Agenten-System „LARS“ und die implementierte „Logic-Engine“ um spezielle Planungsalgorithmen zur Planung von Handlungsweisen von Agenten bereichern. Die erarbeiteten und entwickelten Planungsalgorithmen und deren Nutzen werden im Rahmen dieser Arbeit detailliert erläutert und bewertet.

1.3 Vorgehensweise

Aufgrund der komplexen Thematik wird in der vorliegenden Arbeit erst ein Überblick über die Handlungsplanung allgemein geschaffen. Dabei wird der Sinn und Zweck der Handlungsplanung sowie die allgemeine Funktionsweise beschrieben. Außerdem werden einige weiterführende Algorithmen behandelt, die im Rahmen der Diplomarbeit jedoch nur auf theoretischer Basis erarbeitet wurden.

Im Anschluss daran werden Agenten in Bezug auf Funktionsweise und Besonderheiten gegenüber anderen Softwaretechnologien herausgearbeitet. Ein Überblick des Produktes der living systems AG wird erstellt, sowohl in Bezug auf den Aufbau der Agentenplattform, wie auch in Bezug auf die Funktionsweise, insbesondere auf die für die Diplomarbeit relevante „Logic-Engine“, die als Kontaktstelle zwischen Agenten und Planer fungiert.

Im vierten Kapitel wird die konzeptionelle Integration in das Framework der „Logic-Engine“ vorgestellt. Dabei wird die Integration in Bezug auf andere bereits implementierte Technologien und das Klassenmodell gesehen.

Zum Schluss wird die eigentliche Implementierung vorgestellt, Einblick auf die Herausforderungen bei der Implementierung gewährt und die Handlungsplanung insgesamt bewertet. Im Anhang sind Messprotokolle abgedruckt, welche einen Einblick auf den benötigten Speicher- und Zeitbedarf der Handlungsplanung ermöglichen.

1.4 Hinweise zum Lesen der Diplomarbeit

Quellenangaben zu Zitaten, sinngemäßen Wiedergaben, Abbildungen und Tabellen sind in den Fußnoten durch einen Kurzbeleg in eckigen Klammern und am Ende der Arbeit im Literaturverzeichnis respektive Abbildungs- und Tabellenverzeichnis zu finden.

Sofern es sich um Angaben zu weiterführenden Informationsquellen handelt, was insbesondere für einige Web-Adressen gilt, werden diese nicht im Literaturverzeichnis aufgeführt.

Kapitel 2

Handlungsplanung

In diesem Kapitel wird die Handlungsplanung erläutert. Dabei wird auf die Funktionsweise mit Hilfe anschaulicher Beispiele auf den praktischen Einsatz und auf weiterführende Algorithmen eingegangen.

2.1 Ziel

Das Ziel der Handlungsplanung ist einfach ausgedrückt, in einer gegebenen Umgebung ein vorgegebenes Ziel mit Hilfe der zur Verfügung stehenden Aktionen zu erreichen.

Dabei wird die Umgebung, in der man sich befindet wahrgenommen. Aufgrund dieser Wahrnehmungen und des Ziels kann entschieden werden, welche Aktionen als nächstes ausgeführt werden können. Durch die Ausführung einer Aktion ändert sich die Umgebung, wodurch sich wiederum neue Wahrnehmungen ergeben.

Ziel der Handlungsplanung ist, durch das Ausführen von Aktionen eine vorher definierte Umgebung zu erhalten. Die Umgebung soll so beeinflusst werden, dass die Wahrnehmungen, die man erhält, genau denen entsprechen, welche man vorher als Ziel definiert hat.

Würde man „planlos“ damit beginnen Aktionen auszuführen, würde man sich schnell im Kreis drehen und das Ziel niemals erreichen. Deshalb erstellt man zuvor einen Plan. Der Plan beinhaltet eine Folge ausführbarer Aktionen, welche die Umgebung so ändern, dass die zuvor als Ziel definierten Zustände erreicht werden.

Bei der Handlungsplanung geht man in der Regel von folgenden Eigenschaften aus. Die Umgebung ist:

- deterministisch: die Aktionen haben immer die erwartete Wirkung auf die Umgebung.
- statisch: die Umgebung ändert sich ausschließlich aufgrund der Aktionen des Agenten. Eine sich ändernde Umgebung könnte einen Plan ungültig machen.
- diskret: es existieren nur Wahrheitswerte, wahr oder falsch.

Durch Repräsentation der Umgebung und der möglichen Aktionen, kann jedes beliebige, lösbare Planungsproblem gelöst werden. Umgebungsabhängig ist nur die reale Ausführung des Plans. Real deshalb, weil während des Suchens eines Plans die Aktionen virtuell ausgeführt werden.

In der Literatur werden vor allem die folgenden Beispiele zur Veranschaulichung der Handlungsplanung verwendet. Durch diese Beispiele sollen der Zweck, die verschiedenen Umgebungen und die verschiedenen Methoden der Planung aufgezeigt werden.

2.1.1 Blockwelt

In der Blockwelt existieren verschiedene Blöcke, welche in irgendeiner Form aufeinander oder nebeneinander liegen. Diese sollen durch Umstapeln in einen definierten Zielzustand gebracht werden.

Der Planer erhält eine Beschreibung der Welt – welche Zustände die Blöcke einnehmen können – und der Aktionen mit ihren Vorbedingungen und Effekten. Die Zustände der Blöcke können sein: on_table_block-x, on_block-x_block-y, holding_block-x und clear_block-x. clear_block-x bedeutet, dass kein weiterer Block auf Block X liegt. holding_block-x betrifft sowohl den Block selbst, als auch den Greifer, der den Block hält. Daher gibt es noch einen weiteren Zustand, nämlich handempty, wenn der Greifer keinen Block in der Hand hält.

Die möglichen Aktionen sind: pick_up_block-x, put_down_block-x, stack_block-x_block-y und unstack_block-x_block-y. Die Vorbedingungen und Effekte ergeben sich durch die Aktionen und die möglichen Zustände, was nicht heißen soll, dass sie nicht definiert werden müssten. Hier ein Beispiel: Für ‚pick_up_block-x' muss die Hand des Greifers frei sein, der Block X muss auf dem Tisch liegen und kein anderer Block darf auf Block X liegen. Die Effekte sind: Der Greifer hat seine Hand nicht mehr frei, der Block X liegt nicht mehr auf dem Tisch, der Greifer hält Block X und Block X ist nicht mehr frei.

Propositionen:

- handempty — die Hand ist frei, sie hält keinen Block in der Hand
- holding_x — der Block x wird von der Hand gehalten
- ontable_x — der Block x befindet sich auf dem Tisch
- clear_x — der Block x ist frei, es befindet sich kein weiterer Block auf ihm
- on_x_y — der Block x liegt auf dem Block y

Aktionen:

- pickup_x — der Block x wird vom Tisch aufgehoben. *Vorbedingungen*: der Block x muss sich auf dem Tisch befinden, es darf kein weiterer Block auf ihm liegen und die Hand muss frei sein. *Effekte*: der Block x befindet sich nicht mehr auf dem Tisch, der Block x ist nicht mehr frei, die Hand ist nicht mehr frei und Block x wird gehalten
- putdown_x — der Block x wird auf den Tisch gelegt. *Vorbedingungen*: der Block x wird gehalten. *Effekte*: der Block x befindet sich auf dem Tisch, der Block x ist frei, die Hand ist frei und Block x wird nicht mehr gehalten

- stack_x_y — der Block x wird auf Block y gelegt. *Vorbedingungen*: der Block x wird gehalten und Block y ist frei (kein anderer Block befindet sich auf Block y). *Effekte*: der Block x ist frei, der Block y ist nicht mehr frei, die Hand ist frei, Block x wird nicht mehr gehalten und Block x liegt auf Block y
- unstack_x_y — der Block x wird vom Block y aufgehoben. *Vorbedingungen*: der Block x befindet sich auf Block y, Block x ist frei (kein anderer Block befindet sich auf Block x) und die Hand ist frei. *Effekte*: der Block x ist nicht mehr frei, der Block y ist frei, die Hand ist nicht mehr frei, Block x wird gehalten und Block x liegt nicht mehr auf Block y

Zusätzlich erhält der Planer noch eine Beschreibung der aktuellen Zustände der Blöcke, also des aktuellen Weltzustandes und das Ziel, welches erreicht werden soll.

Im Beispiel in Abbildung 1 liegen Block A und Block B auf dem Tisch. Block C liegt auf Block A. Ziel ist, die Blöcke so zu stapeln, dass Block C auf dem Tisch liegt, Block B auf Block C und Block A auf Block B.

Der Weltzustand zum Start sieht dann wie folgt aus:

- ontable_block-a
- ontable_block-b
- on_block-c_block-a
- handempty
- clear_block-b
- clear_block-c

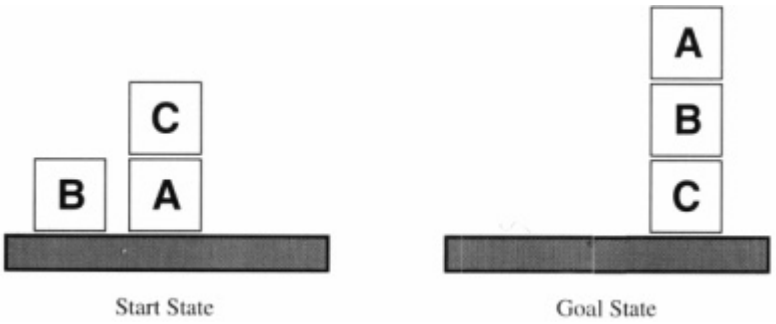

Abbildung 1: Beispiel eines Blockwelt-Problems[1]

[1] Bild aus: ‚Artificial Intelligence a Modern Approach', Kapitel 11.9, Seite 365

Um das Ziel zu erreichen müssten die folgenden Aktionen durchgeführt werden:

- unstack_block-c (Block C von einem anderen Block herunter nehmen)
- putdown_block-c (Block C auf den Tisch legen)
- pickup_block-b (Block B vom Tisch nehmen)
- stack_block-b_block-c (Block B auf Block C stapeln)
- pickup_block-a (Block A vom Tisch nehmen)
- stack_block-a_block-b (Block A auf Block B stapeln)

Der Planer sollte genau auf dieses Ergebnis kommen, wobei je nach Implementierung das Ergebnis auch länger sein kann, da nicht unbedingt die optimale Lösung gefunden wurde.

2.1.2 8-Puzzle Problem

Das 8-Puzzle Problem ist das in Abbildung 2 dargestellte Spiel, bei dem die Zahlen durch Verschieben der Plättchen in eine bestimmte Ordnung der Zahlen zu schieben sind. Solche Probleme werden in der künstlichen Intelligenz verwendet, um Algorithmen zur Lösung von komplexen Problemen zu testen.

Die Möglichkeiten die Plättchen zu verschieben sind so mannigfaltig, dass das Problem mit akzeptablem Aufwand kaum lösbar ist.

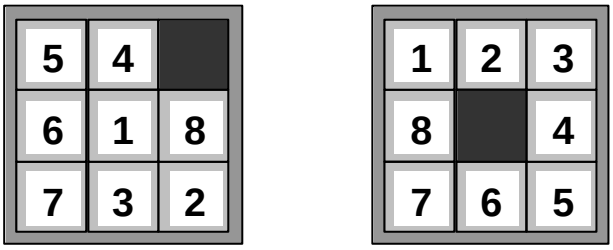

Abbildung 2: Das 8-Puzzle Problem[2]

Mit Hilfe der Handlungsplanung können alle möglichen Kombinationen von Aktionen ausgeführt werden, wodurch ein Plan gefunden werden kann, der eine Folge von Aktionen beinhaltet. Falls ein Plan gefunden wird, führt diese Folge von Aktionen zum gewünschten Ziel.

2.1.3 Missionars-Kannibalen Problem

Bei diesem Problem stehen drei Missionare und drei Kannibalen am Ufer eines Flusses, welcher überquert werden soll. Am Fluss befindet sich ein Ruderboot, welches zwei Personen zur selben Zeit aufnehmen kann. Zu keiner Zeit dürfen die Kannibalen auf einer Seite des

[2] Bild aus: ‚Einführung in die Künstliche Intelligenz', Kapitel 3.3, Seite 25

Flusses, inklusive der Person auf dem Ruderboot, in der Überzahl sein. Ansonsten würden die Missionare von den Kannibalen aufgefressen werden. Im folgenden werden die Kannibalen mit ‚x' und die Missionare mit ‚o' dargestellt. Bevor man sich die Lösung anschaut, sollte man selbst einmal versuchen, dieses Problem zu lösen.

Lösung:

```
xxx ooo
x ooo          xx    →
x ooo          ←     x        x
ooo            xx    →        x
ooo            ←     x        xx
x o            oo    →        xx
x o            ←     xo       x o
xx             oo    →        x o
xx             ←     x        ooo
x              xx    →        ooo
x              ←     x        x ooo
               xx    →        x ooo
                               xxx ooo
```

Listing 1: Missionars-Kannibalen Problem

Wie beim 8-Puzzle Problem können auch hier alle Kombinationsmöglichkeiten der Aktionen „gedacht" werden, wodurch man einen ausführbaren Plan erhalten kann. Nicht ausführbare Aktionen werden ausgeschlossen. Die Aktionen werden in „Gedanken" ausgeführt und es wird „überlegt", was für Auswirkungen die Aktionen haben. Es wird geplant, wie man ans Ziel kommt. Die gemerkten Aktionen, die zum Plan führen, können nacheinander ausgeführt werden.

2.1.4 Routenplanung

Unter Routenplanung versteht man nicht nur, den kürzesten oder schnellsten Weg von A nach B zu finden. Auch Routing in Computernetzen, Reiseberatungssysteme, Flugreisen-Planungssysteme gehören zur Routenplanung.

Ein bekanntes Beispiel der Routenplanung aus der Betriebswirtschaftslehre (BWL) und der Informatik ist das *travelling salesman*[3] Problem. Aufgabe ist hier, die bestmögliche Route für einen Handelsreisenden zu finden, der etliche Stationen (also Kunden) anfahren muss, um diesen etwas zu verkaufen.

Die BWL sieht allerdings nicht vor, Unterscheidungen zu machen, ob irgendwelche anderen Bedingungen bei der Routenfindung erfüllt sein müssen. Im Falle des Handelsreisenden wäre das noch davon abhängig, ob der Kunde zu einem bestimmten Zeitpunkt zu erreichen ist.

Dieses Szenario findet nicht nur Anwendung bei einem Handelsreisenden, sondern auch in der Logistik, Roboternavigation und Steuerung von Maschinen, wie CNC-Fräsmaschinen.

[3] Siehe: „Scheduling Problems and Travelling Salesmen: The Genetic Edge Recombination Operator"

2.2 Repräsentation

Ein grundlegendes Problem der Handlungsplanung ist die richtige Darstellung der Zustände einzelner Objekte, der Repräsentation des gesamten Weltzustandes, des Ziels und der Aktionen mit ihren Vorbedingungen und Effekten.

Die Zustände einzelner Objekte werden als Propositionen bezeichnet. Dabei kann beispielsweise die Proposition „rechter Schuh an“ wahr oder falsch sein. Die Menge aller möglichen Propositionen aller Objekte ergeben den Weltzustand.

Diese Propositionen werden auch dazu verwendet, den Zielzustand zu definieren und die Vorbedingungen und Effekte der Aktionen zu repräsentieren.

2.2.1 Weltzustand

Zunächst muss der Planer dazu in der Lage sein, die Zustände aller Objekte entsprechend abzubilden. Da in vielen Umgebungen die Objekte meistens alle Zustände einnehmen können, erfolgt für die möglichen Zustände eine vollständige Instanziierung. Das bedeutet, dass in der Beschreibung der Umgebung alle Zustände der Objekte dargestellt werden müssen. Am Beispiel der Blockwelt kann ein Block A und ein Block B die folgenden Zustände einnehmen:

- „ontable_block-a“ (Block A befindet sich auf dem Tisch)
- „ontable_block-b“ (Block B befindet sich auf dem Tisch)
- „on_block-a_block-a“ (Block A liegt auf Block A)
- „on_block-a_block-b“ (Block A liegt auf Block B)
- „on_block-b_block-a“ (Block B liegt auf Block A)
- „on_block-b_block-b“ (Block B liegt auf Block B)
- „clear_block-a“ (Block A liegt frei, kein anderer Block liegt auf ihm)
- „clear_block-b“ (Block B liegt frei, kein anderer Block liegt auf ihm)
- „holding_block-a“ (Block A wird von einem Greifer gehalten)
- „holding_block-b“ (Block B wird von einem Greifer gehalten)

Diese Aussagen müssen immer gemacht werden und können wahr oder falsch sein. Da die Objekte jeden dieser Zustände erreichen können, müssen sie dargestellt werden. Existiert noch ein Block C, so werden die Kombinationsmöglichkeiten entsprechend mehr. Auffällig ist, dass „on_block-a_block-a“ und „on_block-b_block-b“ in dieser Domäne niemals wahr werden können, da man ein Objekt nicht auf sich selbst stellen kann. Ohne Domänenkenntnisse kann der Planer das aber nicht wissen.

Dies sind allerdings noch nicht alle Zustände, welche die Welt beschreiben. Der Zustand „handempty“ (die Hand des Greifers ist frei) muss noch dargestellt werden. Dieser Zustand

bezieht sich auf kein definiertes Objekt. Daher gibt es diesen Zustand, unabhängig davon wie viele Objekte existieren, nur ein einziges Mal.

Existieren drei Objekte Block A, Block B und Block C, ergeben sich für „ontable“, „clear“ und „holding“ jeweils drei Zustandsmöglichkeiten, für „on“ ergeben sich $variable^{objekte}$ = neun Zustandsmöglichkeiten und für „handempty“ ergibt sich eine Zustandsmöglichkeit. Mit den Zustandsmöglichkeiten sind die unterschiedlichen Zustands-Objekt-Kombinationen gemeint und nicht der tatsächliche Zustand, wahr oder falsch.

In der Definitionsdatei der Umgebung wird nicht für jedes Objekt ein neuer Zustand definiert, sondern es werden Variable benutzt, die für jedes beliebige Objekt stehen können. So wird beispielsweise „ontable_block-a“ als „ontable_block-x“ definiert. Somit muss „ontable“ nicht für jedes Objekt neu definiert werden. Es reicht, genau diesen Zustand und alle Objekte zu definieren. Der Planer erstellt die vollständige Instanziierung automatisch.

Während der Erstellung der Propositionen kann den Propositionen ein Wahrheitswert zugeordnet werden. Der Wahrheitswert ist abhängig von der Definition, wie die Umgebung zu Beginn der Planung aussehen soll. Dabei ist darauf zu achten, dass nicht Zustände definiert werden, welche sich gegenseitig ausschließen. So ist es am Beispiel der Blockwelt nicht möglich, dass die Propositionen „ontable_block-a“ und „holding_block-a“ gleichzeitig wahr sind.

2.2.2 Aktionen

Aktionen beinhalten im einfachsten Fall lediglich den Namen der Aktion. Um eine Aktion ausführen zu können, müssen meistens bestimmte Zustände herrschen. Man kann beispielsweise nur mit einem Auto fahren, wenn man eines zur Verfügung hat. Somit würde die Aktion „fahre Auto“ mindestens die Bedingung „besitze Auto“ voraussetzen. Zweifelsohne hat das Autofahren den Verbrauch von Energie zur Folge.

Wie schon zu Beginn erwähnt, werden diese Vorbedingungen zur Ausführung einer Aktion wie auch die Effekte ebenfalls mit Hilfe von Propositionen dargestellt. Hier sind jedoch nicht mehr einzelne Propositionen von Bedeutung, sondern eine bestimmte Abhängigkeit zwischen diesen. Eine sinnvolle Aktion hat mindestens einen, und oft auch mehrere Effekte. Jeder einzelne Effekt entspricht genau einer Proposition. Die Aktion „stack_block-a_block-b“ beispielsweise hat zur Folge, dass Block A auf Block B liegt. Dieser Effekt entspricht der Aussage „on_block-a_block-b“, wodurch diese Proposition nach der Ausführung der Aktion wahr wird.

Etwas komplizierter wird es bei den Vorbedingungen. Am Beispiel der Blockwelt sind die Vorbedingungen zwar alle miteinander „und“-verknüpft, also alle Propositionen für die Aktion definierten Vorbedingungen müssen erfüllt sein, jedoch gibt es andere Umgebungen, wo verschiedene Zustände von Vorbedingungen dafür sorgen können, dass eine Aktion ausführbar ist. So kann beispielsweise eine tüte Milch in Supermarkt A oder in Supermarkt B gekauft werden. Somit kann die Proposition „in Supermarkt A“ oder „in Supermarkt B“ wahr sein und die Aktion ausgeführt werden. Für die Vorbedingungen kann eine boolesche

Funktion erforderlich sein. Diese Funktion muss in einer Formel dargestellt werden, und anhand dieser Formel überprüft werden, ob das Ergebnis der Formel erfüllt ist.

$$Y = A \wedge B \vee C \wedge \overline{D} \vee \overline{B \wedge C}$$

Listing 2: Beispiel einer booleschen Funktion

Die Gleichung der Vorbedingung könnte genau so aussehen, wie in diesem Beispiel, wobei die Buchstaben A, B, C, D für jede beliebige Proposition stehen können. Wenn die Vorbedingung wahr ist, ist die Aktion ausführbar. Dazu muss in obigem Beispiel die Proposition A und B wahr sein, oder die Proposition C wahr und D falsch sein, oder die Propositionen B und C falsch sein (trifft zu, wenn eine der beiden Propositionen falsch ist → Gesetz nach De Morgan).

2.2.3 Zielzustand

Das Ziel ist eine Teilmenge des Weltzustandes. Es können alle vorhandenen Objekte darin enthalten sein, oder nur ein Teil der Objekte. Im Zielzustand ist definiert, welche Propositionen wahr sein müssen und welche falsch.

Es kann für jede Proposition einen eindeutig definierten Zustand geben. Dies ist jedoch nicht erforderlich, da nicht zwingend alle Propositionen relevant sind für die Erreichung des Ziels. Daher ist der Zielzustand meistens nur eine Teilmenge aus allen möglichen Weltzuständen.

Am Beispiel der Blockwelt mit zwei Blöcken A und B könnte das Ziel sein, dass Block A auf Block B liegt. Für die Definition des Ziels würde es dann genügen, eine einzige Proposition zu definieren, nämlich „on_block-a_block-b“. Da es nur diese zwei Blöcke gibt, liegt Block B auf jeden Fall auf dem Tisch („ontable_block-b“), der Greifer ist frei („handempty“) und auf Block A liegt kein weiterer Block („clear_block-a“). Kommt ein dritter Block hinzu, wäre das Ergebnis nicht mehr eindeutig, was durchaus gewünscht sein kann. Im anderen Fall müssen zusätzliche Propositionen als Ziel definiert werden.

Die Propositionen des Zielzustands sind logisch mit „und“-verknüpft. Das heißt, dass alle im Ziel definierten Propositionen erfüllt sein müssen.

2.3 Planen

Ausgehend vom Startzustand der Propositionen kann nach einem Plan gesucht werden. Es gibt die verschiedensten Varianten, um das Ziel zu erreichen. Egal, welche dieser Algorithmen man letztendlich verwendet, das Grundprinzip ist immer dasselbe.

Um das Ziel zu erreichen, werden alle möglichen Aktionen in irgendeiner Form nacheinander virtuell ausgeführt und überprüft, ob das Ziel bereits erreicht ist. Das Ausführen der Aktionen geschieht in allen Kombinationsmöglichkeiten, in denen sie ausgeführt werden können. Dadurch entsteht ausgehend vom Start ein Suchbaum. In Abbildung 3 wird die Ausbreitung

eines solchen Suchbaumes schematisch dargestellt. In den folgenden Unterkapiteln werden verschiedene Suchverfahren zur Findung eines Plans vorgestellt.

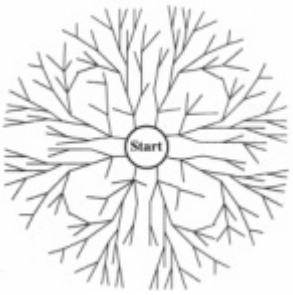

Abbildung 3: Darstellung der Ausbreitung eines Suchbaumes[4]

Insgesamt kann man das Planen in acht Schritte unterteilen. Die Schritte eins bis drei wurden bereits erläutert. Dabei handelt es sich um:

1. Propositionen vollständig instanziieren (Repräsentation des Weltzustandes)
2. Aktionen vollständig instanziieren (Repräsentation der Aktionen mit ihren Formeln für die Vorbedingungen und Effekte)
3. Propositionen mit Wahrheitswerten belegen (den aktuellen Weltzustand definieren)

Die Reihenfolge ist für die ersten drei Schritte veränderbar. Diese drei Schritte sind die Vorarbeit für das eigentliche Planen. Anschließend muss nach einem Plan gesucht werden. Ein allgemeiner Ansatz folgt:

4. Ziel erreicht? → fertig
5. Identifizieren aller ausführbaren Aktionen (für welche Aktionen sind die Vorbedingungen erfüllt). Wenn keine ausführbaren Aktionen vorhanden sind, wird dieser Zweig des Suchbaums verlassen. Ist kein weiterer undurchsuchter Zweig im Suchbaum vorhanden, existiert kein Plan, der zum Ziel führt.
6. eine Aktion aus der Liste aller ausführbaren Aktionen aussuchen
7. ausgesuchte Aktion ausführen
8. wiederhole ab 4.

Der gefundene Plan ist die Abfolge einzelner Aktionen, welche ausgeführt werden müssen, um vom Start zum gewünschten Zielzustand zu kommen. Verschiedene Wege zum Ziel können sich in der Optimalität des Weges unterscheiden. Je nach angewandtem Algorithmus kann der Weg auch über Umwege führen.

[4] Bildausschnitt aus: ‚Artificial Intelligence a Modern Approach', Kapitel 3.5, Seite 81

2.3.1 Suchverfahren

Innerhalb der Handlungsplanung gibt es verschiedene Suchstrategien. Die Wahl des Algorithmus geschieht in Abhängigkeit von der Vollständigkeit und Optimalität, welche erreicht werden soll, und in Abhängigkeit vom Speicher- und Laufzeitverhalten, das der Algorithmus aufweisen soll.

Im Gegensatz zu den Formeln, welche die Vorbedingungen und Effekte von Aktionen repräsentieren, ist der Suchbaum der während der Suche erstellt wird nicht binär. An jedem Knotenpunkt des aktuellen Zustandes der Weltbeschreibung können beliebig viele der vorhandenen Aktionen zur Ausführung geeignet sein. Der Suchbaum ergibt sich also aus den ausführbaren Aktionen an jedem Knoten im Suchbaum, wobei die Knotenpunkte erreichbaren Weltzuständen entsprechen.

Das bedeutet, dass sich der Baum in beliebige Dimensionen ausweiten kann. Infolgedessen ist der Zeitaufwand wesentlich höher als das Generieren der Propositionen und die Wahl des Algorithmus ist umso entscheidender. Deshalb sollte besonders darauf geachtet werden, einen für das vorhandene Problem guten Suchalgorithmus zu verwenden.

Im Wesentlichen wird die Bestimmung des Suchalgorithmus durch die folgenden Kriterien bestimmt:[5]

- **Vollständigkeit:** Wird auf jeden Fall eine Lösung gefunden, sofern es eine gibt?
- **Optimalität:** Wird die beste Lösung gefunden, sofern es mehrere gibt?
- **Zeitbedarf:** Wie viel Zeit darf die Lösungsfindung maximal Kosten?
- **Speicherbedarf:** Wie viel Speicherplatz wird benötigt bzw. steht zur Verfügung?

2.3.2 Breitensuche

Die Breitensuche ist ein üblicher Suchalgorithmus für Standardprobleme. Dabei wird der Suchbaum, wie der Name schon sagt, in der Breite durchsucht. Das heißt im Detail, dass zuerst alle Nachfolger des Startknotens auf einer Ebene exploriert und auf die Erreichung des Ziels hin überprüft werden. Ist das Ziel nicht erreicht, wird auf der nächsten Ebene jeder Nachfolger eines jeden Knotens exploriert und wiederum auf das Ziel untersucht. Dies wiederholt sich, bis das Ziel oder das Ende des Suchbaumes erreicht ist.

Somit erfüllt die Breitensuche das Kriterium der Vollständigkeit, da der gesamte Suchbaum durchsucht wird. Existiert eine Lösung, wird diese auch gefunden. Sind die Pfadkosten der einzelnen Knoten monoton in der Tiefe, so ist die Suche sogar optimal. Der Algorithmus findet die Lösung beim ersten Knoten im Pfad, in dem das Ziel erreicht ist. Gibt es auf einer Ebene gleich zwei Lösungen wird nur eine der beiden gefunden. Dies ist jedoch nicht weiter tragisch, denn beide Lösungen führen zum selben Ziel. Hat man jedoch mehrere Ziele als

[5] Vgl.: ‚Einführung in die Künstliche Intelligenz', Kapitel 3.5, Seite 30

Lösung und diese unterscheiden sich in der Qualität, so ist ein anderer Algorithmus anzuwenden.

Zur Verdeutlichung der Optimalität sei kurz die Planung einer Route des Routenplaners erwähnt. Die Anzahl der Knoten, also die Anzahl der zu durchfahrenden Zwischenstationen, kann zwar geringer sein, jedoch können sich die Pfadkosten durch die Entfernungen oder sonstige Eigenschaften unterscheiden. Somit kann es sein, dass nicht die optimale Lösung gefunden worden ist.

Die Breitensuche hat identische Laufzeit- und Speicherplatzkomplexität. Der Speicherplatzbedarf für einen Knoten ist so groß, wie der Speicherplatzbedarf eines Knotens multipliziert mit der Anzahl der folgenden Zweige. Das wird in der Literatur als *branching factor* (Verzweigungsfaktor) kurz „*b*“ bezeichnet. Der Speicherplatzbedarf gilt nur für einen Knoten mit all seinen Nachfolgern. Die nachfolgende Ebene kann ein Blatt oder ein weiterer Knoten sein. Erweitert man nun diesen einen Knoten auf mehrere folgende Knoten, so entsteht eine Gleichung, mit welcher der Speicherplatzbedarf bzw. Zeitbedarf berechnet werden kann. Die Gleichung lautet:

$1 + b + b^2 + b^3 + ... + b^d = \frac{b^{d+1} - 1}{b - 1} = O(b^d)$, wobei *d* (depth) der Pfadtiefe entspricht.[6]

Tiefe	Knoten	Zeitbedarf		Speicherbedarf	
0	1	1	Millisekunden	100	Bytes
2	111	0,1	Sekunden	11	kilobytes
4	11.111	11	Sekunden	1	megabytes
6	10^6	18	Minuten	111	megabytes
8	10^8	31	Stunden	11	gigabytes
10	10^{10}	128	Tage	1	terrabytes
12	10^{12}	35	Jahre	111	terrabytes
14	10^{14}	3500	Jahre	11.111	terrabytes

Tabelle 1: Zusammenhang Tiefe, Knoten, Zeit, Speicher[7]

Die Breitensuche ist schematisch in Abbildung 4 dargestellt.

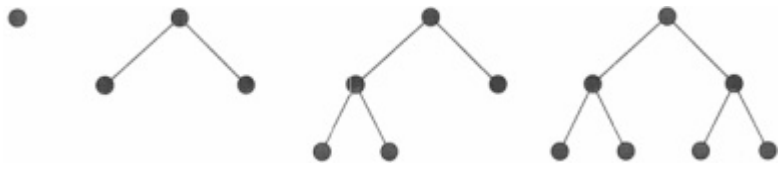

Abbildung 4: Schematische Darstellung der Breitensuche[8]

[6] Vgl.: ‚Einführung in die Künstliche Intelligenz', Kapitel 3.5, Seite 30
[7] Vgl.: ‚Artificial Intelligence a Modern Approach', Kapitel 3.5, Seite 75
[8] Bild aus: ‚Artificial Intelligence a Modern Approach', Kapitel 3.5, Seite 74

2.3.3 Kostengesteuerte Breitensuche

Die kostengesteuerte Breitensuche wird realisiert, indem immer der billigste Knoten zuerst expandiert wird. Somit wird der billigste Pfad als erstes gefunden, sofern sich die Pfadkosten nicht dahingehend ändern, dass ein zunächst sehr teurer Pfad plötzlich sehr billig wird.

Wenn sich die Pfadkosten auf den jeweiligen Pfad gleichmäßig verteilen und die Kosten eher teurer als billiger werden, kann man davon ausgehen, dass man den billigsten Pfad findet. Dazu muss man jedoch den Suchbaum von vornherein kennen und mit Sicherheit sagen können, dass sich die Pfade entsprechend verhalten.

Verhalten sich die Pfade nicht entsprechend oder kann man das Kostenverhalten der Pfade nicht vorhersagen, so ist die kostengesteuerte Breitensuche nicht geeignet. Die einzige Möglichkeit, ein optimales Ergebnis zu erreichen, besteht darin, den ganzen Baum zu durchsuchen, die Pfadkosten aufzuzeichnen und am Ende den billigsten Pfad heraus zu suchen.

2.3.4 Tiefensuche

Die Tiefensuche durchsucht die Knoten, indem sie vom ersten Knoten beginnend einen Zweig bis ans Ende des Suchbaumes exploriert. Ist das Ziel nicht erreicht, nimmt es sich den nächsten Zweig vor, bis ein Ziel gefunden wird, sofern es im Suchbaum vorhanden ist.

Die Tiefensuche ist weder vollständig noch optimal. In vielen Fällen wird mit der Tiefensuche trotzdem ein Ziel gefunden. Wenn nun aber ein Zweig besonders Tief ist, kann man bei dieser Suchstrategie „hängen bleiben“. Das bedeutet, dass der Suchalgorithmus in einen tiefen Zweig kommen kann, vergeblich nach dem Ziel suchen kann, obwohl sich die Lösung in einem sehr kurzen anderen Zweig befindet. Ein weiteres Problem ist, dass mehrere Aktionen genau so ausgeführt werden, dass sie sich gegenseitig wieder aufheben und man somit nie das Ziel erreicht. Um dies an einem trivialen Beispiel zu verdeutlichen, kann man sich des Blockwelt Beispiels bedienen. Wird dort Block D von einem Stapel genommen, auf den Tisch gelegt, dann wiederum Block D vom Tisch genommen und auf den Stapel abgelegt, auf welchem sich der Block bereits vorher befand, so kann sich die Folge der Aktionen ins Unendliche wiederholen. Dieses Szenario kann man zumindest abbrechen, indem man eine *begrenzte Tiefensuche* verwendet, welche innerhalb eines Suchzweiges einen Zähler besitzt, der nach einer vorgegebenen Tiefe den Zweig verlässt und in einem anderen Zweig weitersucht.

Das erste gefundene Ziel wird hier als Lösung präsentiert. Die Lösung ist aber im Allgemeinen nicht optimal.

Vorteil der Tiefensuche gegenüber der Breitensuche ist der geringe Speicherbedarf. Ist es sicher, dass ein Ziel gefunden wird und nur eine mögliche Lösung existiert, ist die Tiefensuche der Breitensuche vorzuziehen. Der Speicherbedarf ist so groß, wie der Speicherbedarf des tiefsten Zweiges, da ein bereits explorierter Zweig beim Zurückgehen in einen übergeordneten Knoten den Speicher nicht mehr benötigt und wieder frei gibt. Der maximale Zeitbedarf der Tiefensuche ist größer als bei der Breitensuche. Der Bedarf setzt sich nicht, wie bei der Breitensuche durch b^d (d entspricht der Pfadtiefe der Lösung) zusammen, sondern aus b^m,

wobei „*m*“ der maximalen Tiefe entspricht. Die Tiefensuche stellt sich, wie in Abbildung 5 zu sehen ist, dar.

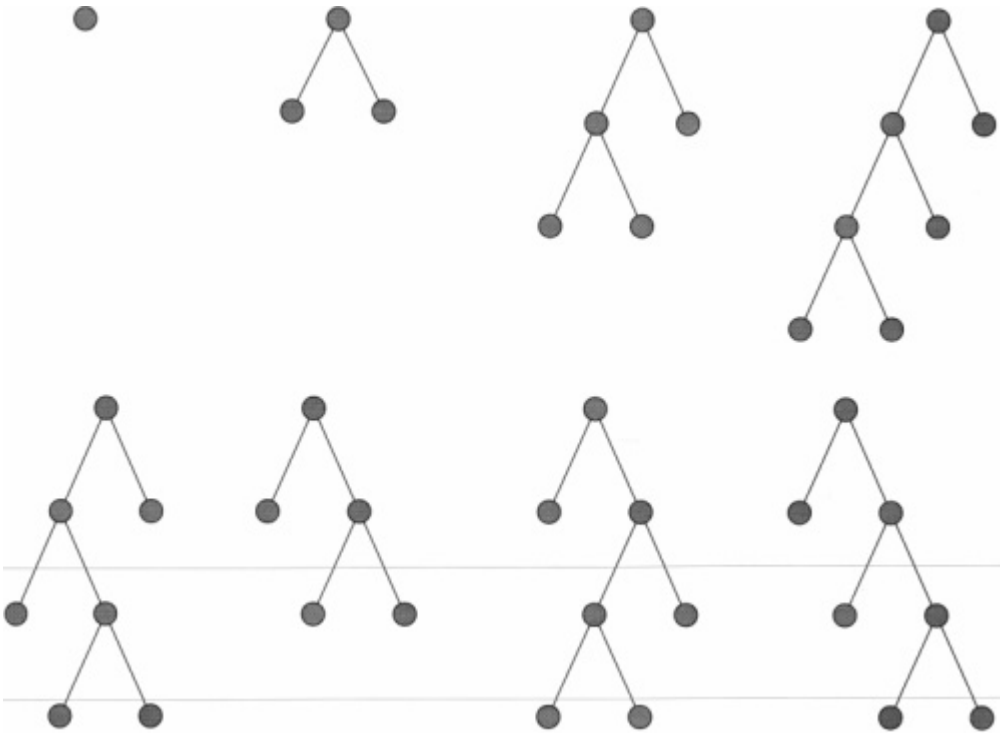

Abbildung 5: Schematische Darstellung der Tiefensuche[9]

2.3.5 Iterative Tiefensuche

Die Funktionsweise der iterativen Tiefensuche ist prinzipiell dieselbe wie die der Tiefensuche. Allerdings wird vorläufig eine maximale Tiefe festgelegt. In der Regel ist die Tiefenbeschränkung zu Beginn bei eins festgelegt.

Ist die Tiefe des Ergebnisses in etwa bekannt, oder kann man bestimmen, dass sich das Ziel mindestens in einer bestimmten Tiefe befindet, so kann die Suchzeit etwas verkürzt werden, indem die Tiefenbeschränkung auf einen höheren Startwert festgelegt wird.

Wird die Tiefenbeschränkung erreicht und das Ziel wurde nicht gefunden, so kann die Tiefenbeschränkung erhöht und der Suchbaum erneut durchsucht werden. Dadurch erreicht man eine Kombination aus Tiefensuche und Breitensuche. Es wird der erste Knoten im Suchbaum gefunden, welcher zum Ziel führt. Somit ist die Suche im Sinne der Breitensuche optimal. Der Speicherbedarf ist geringer als bei der normalen Tiefensuche, da der Suchbaum nur eine maximale Tiefe des Ergebnisses benötigt. Die iterative Tiefensuche ist auch vollständig. Ein Ziel wird immer gefunden, sofern es eines gibt.

[9] Bild aus: ‚Artificial Intelligence a Modern Approach', Kapitel 3.5, Seite 77

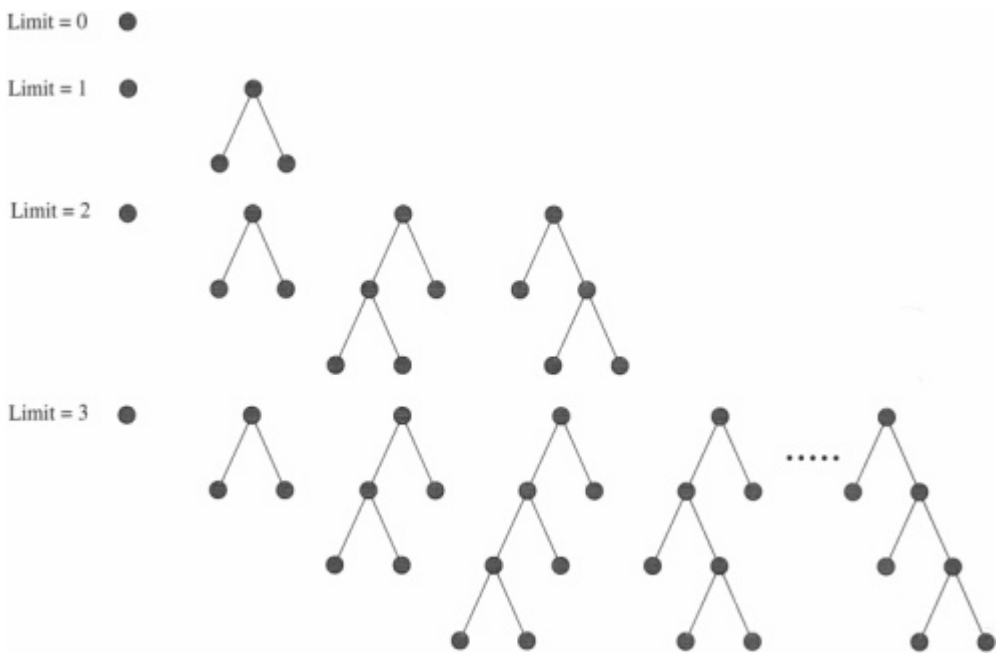

Abbildung 6: Schematische Darstellung der iterativen Tiefensuche[10]

2.3.6 Bidirektionale Suche

Die bidirektionale Suche vereint die Vorwärts- und Rückwärtssuche in einem Algorithmus. Dazu wird der Suchbaum von beiden Richtungen gleichzeitig expandiert, vom Start und vom Ziel. Da bei der Handlungsplanung sowohl der Startpunkt, als auch der Zielpunkt bekannt sind, ist dies prinzipiell möglich.

Es ist jedoch wesentlich aufwändiger, eine Rückwärtssuche zu realisieren, da nicht die Folgeknoten bekannt sein müssen, sondern der oder die Ursprungsknoten. Die Aktionen, die ausgeführt werden können, müssen alle umkehrbar sein. Wenn mehrere Ziele existieren, erweist sich die bidirektionale Suche als sehr schwierig. Es liegt dann ein so genanntes „Mehr-Zustands-Problem" vor.

Die bidirektionale Suche hat jedoch einen entscheidenden Vorteil. Der Suchbaum muss jeweils nur zur Hälfte durchsucht werden. Dadurch wird der benötigte Zeitbedarf drastisch reduziert. Der Zeitbedarf errechnet sich aus: $b^{\frac{d}{2}}$, wobei *b* wiederum der *branching factor* ist und *d* der Zieltiefe im Baum entspricht. Der Speicherplatzbedarf errechnet sich mit derselben Formel. Er ist gegenüber der Breitensuche ebenfalls erheblich geringer, jedoch etwas höher als bei der Tiefensuche.

[10] Bild aus: ‚Artificial Intelligence a Modern Approach', Kapitel 3.5, Seite 79

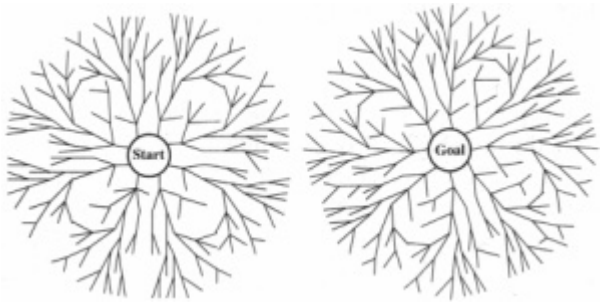

Abbildung 7: Ausbreitung des Suchbaumes bei bidirektionaler Suche[11]

2.3.7 Kostensuche

Die Kostensuche kann unterschiedliche Suchalgorithmen implementieren. Entscheidend für die Suche ist, dass Wegekosten mit eingerechnet werden, um den optimalen Pfad zu finden.

Die Kosten können sich in den unterschiedlichsten Grundlagen niederschlagen. Bei der Routenplanung kann der kürzeste, günstigste oder schnellste Weg von Bedeutung sein. Beim so genannten *travelling salesman* Problem können auch noch Faktoren hinzukommen, wie „Ist der Kunde zu einem Zeitpunkt erreichbar, …". Bei anderen Problemen spielen wiederum andere Faktoren eine Rolle für die Kostenbestimmung.

2.3.8 Heuristische Suchverfahren

Heuristische Algorithmen schließen innerhalb eines Plans von vorne herein schlechte oder nicht ausführbare Züge bzw. Aktionen aus. Sie beginnen ihre Suche mit den vielversprechendsten Aktionen. Innerhalb dieser Arbeit soll aber nicht näher auf die Funktionsweise dieser heuristischen Algorithmen eingegangen werden. Hier sei auf das Literaturverzeichnis im Anhang verwiesen.

Zur Vervollständigung sollen die Suchverfahren zumindest genannt werden. Folgende heuristische Suchverfahren werden in der Literatur erwähnt: Best-First-Suche, Greedy-Suche, A*-Suche, SMA*-Suche, IDA*-Suche und Hill-Climbing.

2.3.9 Gegenüberstellung der Suchalgorithmen

Kriterium	Breiten-suche	Kosten-suche	Tiefen-suche	Limitierte Tiefen-suche	Iterative Tiefen-suche	Bidirektionale Suche
Zeit	b^d	b^d	b^m	b^l	b^d	$b^{d/2}$
Speicher	b^d	b^d	bm	bl	Bd	$b^{d/2}$
Optimal?	Ja	Ja	Nein	Nein	Ja	Ja
Vollständig?	Ja	Ja	Nein	Ja, wenn l = d	Ja	Ja
d = tiefe des Ziels; *m* = tiefe des Suchbaums; *l* = limitierte tiefe des Suchbaums						

Tabelle 2: Gegenüberstellung Eigenschaften der Suchalgorithmen[12]

[11] Bild aus: ‚Artificial Intelligence a Modern Approach', Kapitel 3.5, Seite 81

2.4 Erweiterungen

Die KI (Künstliche Intelligenz) beschäftigt sich seit den frühen 80er Jahren mit der Handlungsplanung. Dennoch ist es bis heute nicht gelungen die großen Handlungsprobleme zufrieden stellend zu lösen.

Aufgrund des hohen Speicher- und Zeitbedarfs lassen sich Probleme, wie das *travelling salesman* Problem nur Näherungsweise lösen. Die optimale Lösung lässt sich in angemessener Zeit nur für sehr kleine Instanzen des Problems ermitteln. Daher wurden einige Strategien zur Verbesserung des Zeit- und Speicherbedarfs entwickelt. Außerdem wurde die Anwendbarkeit durch zusätzliche Funktionalität erweitert, wie beispielsweise Wahrscheinlichkeiten bei Effekten.

Propositionale Handlungsplanung ist im Allgemeinen *P-Space-vollständig* Zur Definition solcher Probleme:

- Mit „P“ bezeichnet man die Menge aller Probleme, die sich auf herkömmlichen deterministischen Rechnern in polynominaler Zeit bearbeiten lassen, wobei unter einem deterministischen Rechner ein Rechner zu verstehen ist, bei dem der Ablauf vorhersagbar ist (also jeder normale existierende Rechner)

- Mit „NP“ bezeichnet man die Menge aller Probleme, die sich auf nicht deterministischen Rechnern in polynominaler Zeit bearbeiten lassen, wobei unter einem nicht deterministischen Rechner ein Rechner zu verstehen ist, der „perfekt raten“ kann (er rät also die richtige Wahl immer auf Anhieb richtig, solch einen Rechner gibt es natürlich nicht)

- Mit „P-Space“ bezeichnet man die Menge aller Probleme, die sich auf nicht deterministischen Rechnern mit polynominalen Speicherplatz bearbeiten lassen

Jeder polynominale Algorithmus auf einem nicht deterministischen Rechner kann mit exponentiellem Aufwand auf deterministischen Rechnern simuliert werden. Ein *NP-vollständiges* Problem ist eine Teilmenge von einem *P-Space-vollständigem* Problem. Ein *P-Space-vollständiges* Problem ist daher im Allgemeinen noch komplexer als ein *NP-vollständiges* Problem. Somit lassen sich entsprechend große Planungsprobleme nach heutigen Erkenntnissen nicht effizient lösen. Einige der im folgenden beschriebenen Algorithmen versuchen zumindest, die obere Grenze der lösbaren Probleme zu erhöhen.

2.4.1 Partielle Planung

Die partielle Planung wurde 1977 von Austin Tate[13] entwickelt. Die Motivation für die Erstellung von nichtlinearen Planern ist die Flexibilität bei der Erstellung von Plänen. Beim Arbeiten mit total geordneten Plänen wird immer eine Aktion nach der anderen berechnet, bei nichtlinearen Planern ist die Reihenfolge von Aktionen zunächst nicht festgelegt. Sie kann sich bei der weiteren Berechnung eines Plans ändern.

---

[12] Vgl.: ‚Einführung in die Künstliche Intelligenz’, Kapitel 3.5, Seite 32

[13] Vgl.: ‚O-Plan: the open planning architecture’, Artificial Intelligence, 52 (1) (1991) Seiten 49-86

Dazu werden nur die wichtigsten Ordnungsbedingungen für Aktionen erfasst, d.h. es wird nur grob festgehalten, welche Aktionen unbedingt vor bzw. nach anderen ausgeführt werden müssen. Eine exakte Reihenfolge für die Aktionen gibt es aber nicht. Dieses Ordnungsverfahren wird als „Least Commitment" bezeichnet und wird als synonym für partiell geordnetes Planen gebraucht.

Ein partiell geordneter Plan sieht dann wie folgt aus. Die Kinder eines Knotens bestehen aus einem Tupel von möglichen Aktionen A, Ordnungsbedingungen O und kausalen Verknüpfungen (Links) L, wobei eine kausale Verknüpfung jeweils eine Menge von Propositionen Q enthält.

Durch die fehlende Festlegung der Ausführungsreihenfolge von Aktionen ergeben sich bei partiellen Plänen Probleme, die bei total geordneten Plänen nicht auftreten. Dies sind hauptsächlich die so genannten Threats (Bedrohungen), die dann auftreten, wenn eine Proposition für eine kausale Verknüpfung durch den Effekt einer anderen Aktion negiert werden kann und dies nicht der Ordnungsrelation widerspricht. Um einen Threat zu beseitigen gibt es die Möglichkeit der Demotion oder Promotion, was nichts anderes bedeutet als die Verschiebung der Ausführung der Aktion A vor B (Demotion) bzw. hinter B (Promotion).

Um partielles Planen zu realisieren, müssen zusätzliche Informationen über die Reihenfolge der auszuführenden Aktionen vorliegen. Dadurch kann erreicht werden, dass Aktionen welche unabhängig voneinander ausgeführt werden können, zur selben Zeit, also parallel ausführbar sind. Kann keine Parallelität errechnet werden, erhält man wieder einen linearen Plan. Sind Aktionen aber parallel ausführbar, also in ihren Vorbedingungen und Effekten teilweise unabhängig voneinander, kann parallel geplant und ausgeführt werden. Der größte Vorteil ist, dass durch diese Parallelität der Suchbaum wesentlich kleiner werden kann. Dadurch kann ein entscheidender Geschwindigkeitsvorteil verbucht werden.

2.4.2 Hierarchische Planung

Die hierarchische Planung wurde 1974 von Sacerdoti[14] entwickelt. Dabei geht es darum, komplexe Zusammenhänge erst im Großen zu sehen und dann ins Kleine herunter zu brechen.

Dabei wird versucht, große Probleme erst einmal auf einer hohen Abstraktionsebene zu lösen. Die einzelnen Schritte der Ergebnisse werden dann wieder für sich gesehen und wiederum auf einer niedrigeren Abstraktionsebene betrachtet, bis die Aktionen „atomar" sind.

Zur Erläuterung dieses Algorithmus kann das Einkaufsproblem betrachtet werden. Dabei geht es darum, zum Supermarkt zu gehen, Butter und Milch zu kaufen und wieder nach Hause zu gehen. Das hört sich trivial an, jedoch muss berücksichtigt werden, dass ein Softwareprogramm damit nicht zurechtkommen würde. Einem Softwareprogramm muss jeder einzelne Schritt bis ins letzte Detail mitgeteilt werden. Die Aktionen wären dann zum Beispiel: „gehe 100 Schritte, drehe dich nach rechts um 23°, gehe 38 Schritte, …".

[14] Vgl.: ‚Planning in a hierarchy of abstraction spaces', E. D. Sacerdoti

Man erkennt bereits, dass ein „normaler“ Planer total überfordert wäre und keine Lösung in akzeptabler Zeit finden würde. Die Anzahl der möglichen Aktionen steigt mit der Genauigkeit des Plans. Wird eine detaillierte Darstellung der Aktionen im Einkaufsbeispiel verwendet, steigen die möglichen Aktionen ins unermessliche. Die 100 Schritte, die gegangen werden sollen, können nicht in einer Aktion dargestellt werden, da Propositionen keine Mengenangabe unterstützen. Es muss dann Schritt für Schritt gegangen werden, wodurch die 100 Schritte alleine bereits 100 Aktionen beansprucht. Dadurch wird der Suchbaum so groß, dass der Speicherbedarf und der Zeitbedarf sehr stark ansteigen, wodurch ein Planen nahezu unmöglich wird.

Eine Möglichkeit, das Problem zu umgehen ist hierarchisches Planen. Dabei wird erst abstrakt betrachtet, welche „großen“ Aktionen ausgeführt werden müssen. Unter diesen „großen“ Aktionen sind die abstrakten Aktionen zu verstehen, welche keine endgültige Beschreibung der Aktionen sind, sondern mehrere Aktionen zusammenfassen. Dies wären im Einkaufs-Beispiel: „gehe zum Supermarkt, kaufe Milch, kaufe Butter, gehe nach Hause“. Nun wird jeder einzelne Vorgang wieder für sich gesehen. So kann das Kaufen der Milch für sich gesehen und abstrakt beschrieben werden. Dies könnte zum Beispiel so aussehen: „gehe an die Kühltheke, nimm eine Milchtüte“. Hinzukommen könnten bei einem solchen Szenario dann noch Dinge wie: „überprüfe Verfallsdatum“. Das wird so lange konkretisiert, bis man letztendlich die atomaren Aktionen und somit den vollständigen Plan hat. Man kann hieran erkennen, dass dieser Algorithmus nicht auf alle Probleme sinnvoll anwendbar ist.

Die großen Vorteile bei diesem Algorithmus liegen in der Schnelligkeit und des geringen Bedarfs an Speicher im Gegensatz zu anderen Algorithmen. Es kann jedoch kein optimales Ergebnis bei der Anwendung dieses Algorithmus garantiert werden.

2.4.3 Graph Planung

Die Graph Planung wurde 1995 von Blum und Furst[15] entwickelt. Diese Art der Planung ist die derzeit schnellste Art einen Plan zu finden. Dabei wird zunächst in Vorwärtsrichtung der Suchbaum expandiert. Jedoch wird nicht wie bei anderen Planern der Weltzustand nach einer Aktion betrachtet, sondern der Weltzustand aller Aktionen auf einer Ebene. Man kann sich das als Interferenz vorstellen, ausgehend vom Startzustand. Um den Startzustand werden Kreise gebildet, welche alle möglichen Aktionen und damit neuen Zustände beinhalten, die ausgehend vom Startzustand jeweils die gleiche Entfernung haben. Ist an irgend einer Stelle eine Situation erreicht, wo alle - zur Erreichung des Ziels - nötigen Propositionen wahr werden, kann von dort aus eine Rückwärtssuche vorgenommen werden, welche in den meisten Fällen zum gesuchten Plan führen. Durch dieses Szenario wird der Suchbaum stark verkleinert, da quasi alle Aktionen parallel ausgeführt werden.

Annahme: Bei der Annahme, dass 100 verschiedene Propositionen existieren, werden vom Start zunächst alle ausführbaren Aktionen ausgeführt. Das Ausführen der Aktionen hat eine Änderung in einigen Propositionen zur Folge. Auf dieser Grundlage können weitere Aktionen

[15] Vgl.: ‚Fast Planning through Planning Graph Analysis’, Artificial Intelligence

ausgeführt werden. Falls ein Ziel existiert, kommt der Planer irgendwann an eine Stelle, an der alle Propositionen wahr sind, die für das Ziel zutreffend sein müssen. Dadurch wird der komplette Suchgraph parallel und nur bis zur nötigen Tiefe expandiert. Angenommen wird, dass die Propositionen mit dem Index 7, 9, 11, 13, 27, 48, 64, 96 zur Erreichung des Ziels wahr sein müssen. Tritt dies beispielsweise in der dritten Ebene aller ausführbaren Aktionen ein, kann von dort aus ein Weg zurück zum Ausgangspunkt gesucht werden. Erreicht man den Ausgangspunkt, hat man gleich den Weg zum Ziel und damit den Plan gefunden. Aktionen, deren Propositionen ab der Position des Suchbeginns nicht wahr sind, werden erst gar nicht ausgeführt, wodurch der Suchraum letztendlich kleiner wird.

Der Suchbaum kann vorwärts einfach durchlaufen werden. Von den möglichen Zielen aus wird der Graph rückwärts durchsucht. Dabei ist zu beachten, dass rückwärts nicht Aktionsfolgen ausgeführt werden, welche sich gegenseitig ausschließen. Kommt man wieder zurück zum Ausgangspunkt, hat man den Plan gefunden. Falls kein Weg zurück zum Ausgangspunkt existiert, muss der Graph - ausgehend von den zuvor erreichten Endpunkten - noch um eine weitere Stufe expandiert werden und wiederum rückwärts zum Ausgangspunkt gesucht werden. Dies ist jedoch selten der Fall, wodurch im Vergleich zu anderen Planungsalgorithmen der Plan schnell gefunden wird.

Die Vorwärtssuche ist nicht zielgerichtet. Es wird sozusagen einfach drauf los gesucht. Wenn ein Ziel existiert, wird dieses auch gefunden. Es wird sogar der kürzeste Weg zum Ziel gefunden. Mit der Graph Planung kann ein unlösbares Problem sehr schnell lokalisiert werden. Dafür sind zwei entscheidende Faktoren ausschlaggebend:

- die Aktionen fangen an, sich zu wiederholen, liefern jedoch dieselben Ergebnisse, ohne jemals alle nötigen Zustände zu erreichen
- es wird eine Stelle erreicht, an der keine Aktionen mehr ausführbar sind, ohne die nötigen Zustände erreicht zu haben

Tritt einer dieser beiden Fälle auf, kann mit Sicherheit bestimmt werden, dass kein Plan existiert, ohne danach gesucht zu haben. Dieses Phänomen ist sehr untypisch für solche Algorithmen, da die Feststellung des Nichtexistierens eines Ziels bei den meisten anderen Algorithmen die meiste Zeit benötigt.

Bei sehr kleinen Problemen lohnt es kaum die Graphplanung einzusetzen, da der „overhead“ zu groß ist. Bei mittleren bis großen Planungsproblemen ist die Planung wesentlich schneller als bei anderen Algorithmen.

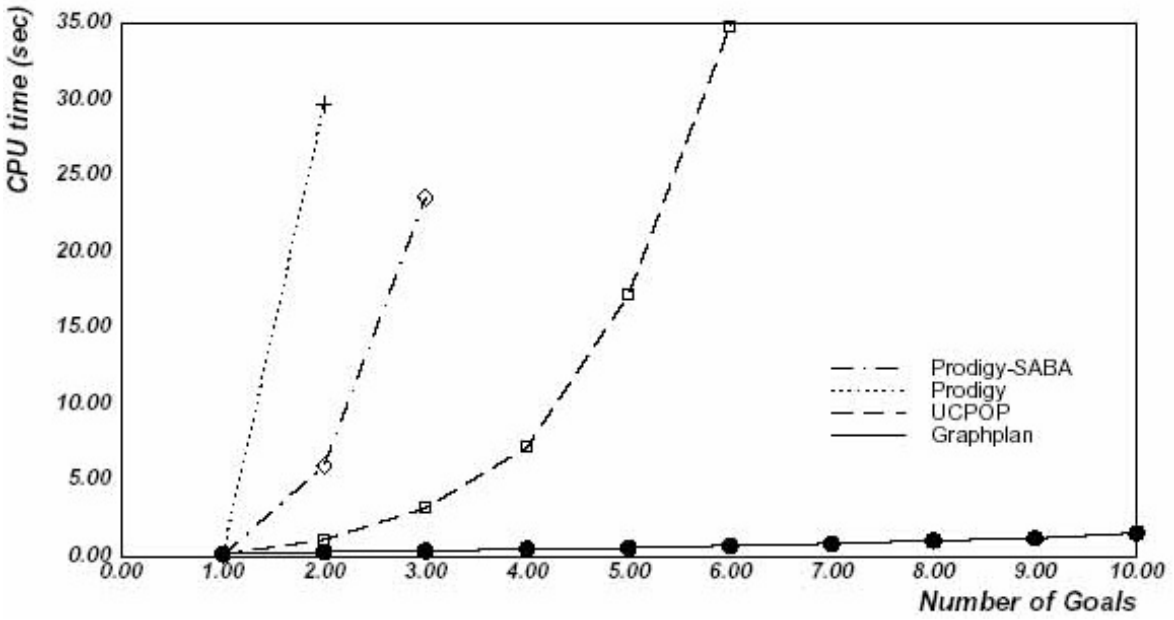

Abbildung 8: Zeitbedarf beim 2-Raketen-Problem[16]

2.4.4 Probabilistische Planung

Die probabilistische Planung wurde 1995 von Kushmerick[17] entwickelt. Im Gegensatz zu den bisher vorgestellten Erweiterungen der Handlungsplanung wurde diese Erweiterung nicht zur Geschwindigkeitsoptimierung entwickelt, sondern um erweiterte Möglichkeiten in der Handlungsplanung zu eröffnen.

Bisher wurden lediglich Planungsalgorithmen mit propositionalen Zuständen vorgestellt. Dieser Algorithmus unterstützt Wahrscheinlichkeiten bei den Effekten. Dabei können für die Effekte der Ausführung einer Aktion Wahrscheinlichkeiten angegeben werden, mit denen die Effekte eintreten. Aus diesem Wissen kann dann die „beste“ Aktion ausgeführt werden und auf diese Weise nach dem Plan gesucht werden. Nun kann es jedoch sein, dass die Aktionen mit den größten Wahrscheinlichkeiten ausgeführt werden, und doch nicht zum Ziel führen. Dann müssen entsprechend Aktionen mit geringeren Wahrscheinlichkeiten ausgeführt werden, bis ein Ziel gefunden wird, oder der gesamte Baum durchsucht ist.

Dieser Planungsalgorithmus erhebt keinen Anspruch auf Optimalität. Wenn der Baum bis zum Schluss durchsucht wird ist der Suchalgorithmus vollständig. Im Extremfall kann die Prozedur jedoch einen sehr hohen Zeitbedarf mit sich bringen.

[16] Vgl.: ‚Fast Planning Through Planning Graph Analysis’, Kapitel 4.1, Seite 11
[17] Vgl.: ‚An Algorithm for Probabilistic Planning’, University of Washington Department of Computer Science

Kapitel 3

Living Agents Framework

Das zweite Teilgebiet dieser Diplomarbeit umfasst Java-basierte Software-Agenten und das Living Agents Framework, auf welche in diesem Kapitel genauer eingegangen wird. Der Bereich der Agententechnologie ist schon seit längerer Zeit ein interessantes Forschungsgebiet an Universitäten und Unternehmen, das seine Wurzeln in der KI (Künstlichen Intelligenz) hat.

3.1 Agenten

Wie oben bereits erwähnt, haben sich im Laufe der letzten Jahre viele Forscher mit Software-Agenten befasst und in zahlreichen Publikationen versucht, dieses Paradigma zu definieren:

- Der AIMA[18] Agent (Russel und Norvig, 1995):
  "*An agent is anything that can be viewed as perceiving its environment through sensors and acting upon that environment through effectors*."
- Der MAES[19] Agent (Pattie Maes, 1995):
  "*Autonomous agents are computational systems that inhabit some complex dynamic environment, sense and act autonomously in this environment, and by doing so realize a set of goals or tasks for which they are designed*"

Unter vielen weiteren Eigenschaften sind die wesentlichen Eigenschaften von Agenten Autonomie, Kollaboration und Mobilität, also die Fähigkeit, die Umgebung zu wechseln, um z. B. im Internet auf einen anderen Server zu migrieren. Auf diese Eigenschaften wird in den nächsten Abschnitten etwas genauer eingegangen.

3.1.1 Autonomie

Autonomie ist eine wesentliche Eigenschaft von Agenten und ein wesentlicher Unterschied zu Objekten objektorientierter Software. Agenten können in einer vom Benutzer festgelegten Domäne selbständig agieren. Dabei werden grundsätzlich zwei Möglichkeiten der Verhaltensauswahl unterschieden.

Zum einen gibt es das reaktive Verhalten, welches die Fähigkeit eines Agenten beschreibt, auf die Wahrnehmungen seiner Umgebung und deren Änderung zu reagieren. Dabei geht er nach einem fest vorgegebenen Muster vor, das er vorher erlernt hat. Bei dieser Verhaltensart ist sich der Agent allerdings nicht der Auswirkungen seiner auszuführenden Aktion bewusst. Der Vorteil dieser Technik ist, dass sie einfach zu implementieren ist und schnell auf Änderungen in der Umgebung reagieren kann. Sie wird daher bevorzugt in dynamischen, d.h. sich schnell

[18] AIMA ist ein Akronym für „Artificial Intelligence: a Modern Approach", ein bemerkenswertes Essay zu Software-Agenten, die AI-Techniken anwenden.
[19] Pattie Maes arbeitet im MIT Media Lab uns ist eine Pionierin in der Erforschung von Software-Agenten

ändernden Umgebungen eingesetzt. Demgegenüber gilt Verhalten als proaktiv, wenn es vorausschauend und zielgerichtet ist. Der Agent nimmt Wahrnehmungen auf und gleicht diese mit seinem Ziel ab. Da er die Auswirkungen seiner Aktionen kennt, kann er vorausschauend darauf hin arbeiten. Eine explizite Repräsentation der Ziele des Agenten ist also eine wichtige Grundvoraussetzung bei proaktivem Verhalten. Idealerweise ist ein Agent in der Lage, sowohl reaktiv als auch proaktiv zu agieren. Diese nicht triviale Aufgabe, reaktives und proaktives Verhalten zu integrieren, ist seit der Entstehung von Agententechnologie Gegenstand der Forschung.

3.1.2 Kollaboration

Eine weitere Charaktereigenschaft von Agentensystemen ist die Interaktion zwischen den Agenten. Diese Interaktion geschieht sowohl indirekt durch die Änderung der gemeinsamen Umgebung, z. B. wenn Fußballagenten den Ball hin und her spielen, als auch durch direkte Kommunikation. Um eine reibungslose Kommunikation unterschiedlicher Agenten zu gewährleisten, wurden verschiedene Standards für Agentenkommunikation beschlossen. Die bekanntesten und weitverbreitetsten sind KQML, KIF sowie FIPA-ACL[20]. Ein weiterer wesentlicher Unterschied zu herkömmlicher Programmierung ist, dass ein Agent selbst entscheiden kann wann und mit wem er kommunizieren möchte. Kollaboration ist vor allem deshalb wichtig, weil nicht einzelne Aktionen zählen, sondern erst die Zusammenarbeit von allen Teilen zu einer erfolgreichen Bewältigung eines Problems führen. Die Lösung komplexer Aufgaben wird durch diese Zusammenarbeit erst möglich. Man kann sich das wie die Arbeit in einem großen Unternehmen vorstellen. Erst durch die erfolgreiche Zusammenarbeit der einzelnen Abteilungen kann ein entsprechendes Produkt entstehen. Genau wie in realen Unternehmungen oder Organisationen schafft diese Eigenschaft gleichzeitig die Herausforderung, die Kollaboration so zu organisieren, dass eine optimale Gesamtperformance entsteht.

3.1.3 Mobilität

Die Mobilität ist ein weiterer Schritt in Richtung transparenter, verteilter, agenten-basierter Systeme. Unter Agenten-Mobilität versteht man die Fähigkeit des Agenten, mit seinen Daten und seinem binären Code von einer Plattform zur anderen zu migrieren. Es wird also im Gegensatz zum Austausch von Nachrichten (data-shipping) sowohl der Zustand, als auch die Funktionalität des Agenten verschickt (code-shipping). Diese Mobilität kann unter anderem in der Informationssuche eingesetzt werden, wenn ein Agent von einer Plattform zur nächsten wandert, um interessante Daten zu filtern und nach der Suche mit den gesammelten Daten zum Auftraggeber zurückzukehren. Dadurch kann vermieden werden, dass alle Daten über das Netzwerk geschickt werden müssen, von denen anschließend nur ein winziger Bruchteil verwendet wird. Eine andere Möglichkeit ist die Migration auf einen alternativen Server, wenn z. B. der eigene Server heruntergefahren wird, um dort ohne merklichen Unterschied für den Benutzer ihre Funktion weiterhin auszuführen.

---

[20] Kone, M.T., Shimazu, A., und Nakajima, T. *Knowledge and Information Systems 2*, Seiten 259-284.

Ein weiterer Vorteil der Agenten-Mobilität ist ihre hohe Flexibilität. Da mobile Agenten ihre eigenen Protokolle benutzen, sind sie nicht von den verwendeten Protokollen abhängig, sie können sich sogar je nach Umgebung aus ihren zur Verfügung stehenden Protokollen das passende auswählen.

3.2 Einsatzgebiete

Agentensysteme haben mittlerweile auch in kommerziellen Systemen Fuß gefasst, und werden nicht mehr nur in isolierter Laborumgebung untersucht. Es gibt eine Reihe von möglichen Einsatzgebieten, die in den folgenden Abschnitten genauer beleuchtet werden.

3.2.1 Auktions-Plattform

Eines der älteren Einsatzgebiete von Agenten-Plattformen ist der Bereich der B2C oder B2B Auktionen, die gegen Ende der 90er Jahre aufkamen. Beispiele für diese Art von Agenten-Plattformen sind die B2C-Plattform Ebay (www.ebay.de) und die B2B Plattformen Netbid (www.netbid.de) und Portax (www.portax.de).

Anforderungen an eine solche Plattform sind unter anderem Skalierbarkeit, um bei einem entsprechend hohen Datenaufkommen das System ohne große Probleme erweitern zu können, eine entsprechende Verfügbarkeit, um für den Kunden rund um die Uhr da zu sein und eine problemlose Integration in bestehende Systeme, weil eine Trennung der Plattform von anderen Geschäftprozessen einen größeren Verwaltungsaufwand bedeuten würde. Hier können Agenten-Plattformen jeden ihrer Vorteile ausspielen. Zum einen sind sie sowohl auf kleinen, wie auch auf großen Systemen lauffähig. Durch die Möglichkeit der Verteilung kann man eine oder mehrere Plattformen auf den Servern laufen lassen. Man erhält trotz allem eine transparente Lösung, da sich die einzelnen Plattformen wie eine verhalten. Durch ihre Modularität ist eine Wartung der Plattform ohne Probleme zur Laufzeit möglich, der normale Geschäftsbetrieb wird nicht unterbrochen. Die Integration der Plattform in bestehende Systeme wird durch entsprechende Schnittstellen unterstützt. Ein weiterer Vorteil ist die Möglichkeit der Kapselung der einzelnen Prozesse, d.h. jeder Agent hat eine ganz bestimmte Aufgabe.

3.2.2 Logistik-Plattform

Logistik war schon immer eine der Schlüsselvoraussetzungen für den Handel mit tangiblen Gütern, aber erst in den letzten Jahren, mit zunehmend bedarfssynchroner Lieferung, entwickelte sich Logistik zum unverzichtbaren Bestandteil für Unternehmen, die in diesem Markt bestehen wollen. Mit dem immer stärker werdenden eCommerce-Markt und den damit verbundenen Angebots- und Bedarfsketten, ist auch die Logistik immer mehr ins Zentrum der Betrachtung gekommen. Jede eCommerce-Plattform benötigt eine entsprechende Logistik-Komponente, um die verkauften Waren zum Kunden zu bringen. Auch werden eigenständige Logistikplattformen, wie das Portivas-System der Post AG, immer interessanter vor allem im Bezug auf B2B Ökosysteme.

Aber um ein entsprechend dynamisches Logistiksystem aufzubauen, bedarf es leistungsfähiger Lösungen. Es muss vor allem Real-Time fähig sein, um die auftretenden Ereignisse schnell zu bearbeiten und so einen effizienten Logistikprozess zu gewährleisten. Gerade für diesen Bereich eignen sich Software-Agenten hervorragend, da sie ihr Umfeld permanent kontrollieren und die entsprechenden Prozesse koordinieren und organisieren können, die von der Logistik-Plattform zur Verfügung gestellt werden. Beispiele sind neben eProcurement das Management von Frachten, das Real-Time Monitoring sowie der Handel mit Frachtkapazitäten.

3.2.3 Finanz-Plattformen

Der Finanzsektor gehört zu den Wirtschaftszweigen, die sich in der jüngsten Vergangenheit rasend schnell und überaus nachhaltig verändert haben und sich auch in Zukunft weiter verändern werden. Neue Mitbewerber, neue Vertriebskanäle, die Möglichkeiten der elektronischen Kommunikation und Informationsgebung sorgen für verschärften Wettbewerb.

Einer der Finanzsektoren ist unter anderem das Trading, welches durch das Internet fundamentalen Änderungen unterworfen wurde und wird. In der Vergangenheit war Trading durch hohe Infrastrukturkosten geprägt und damit auf traditionelle Börsen und auf den institutionellen Handel begrenzt. Die Zukunft gehört einem trading anywhere, das durch die niedrigen Transaktionskosten im Internet sowie durch die Liberalisierung und damit einhergehende größere Handelsvielfalt geprägt sein wird. Hier kann man die vielfältigen Möglichkeiten einer Agenten-Plattform nutzen. Die Möglichkeiten erstrecken sich von einfachen B2B Marktplätzen bis hin zum Real-Time-Handel über sogenannte Electronic Communications Networks (ECNs). Ein eindrucksvolles Beispiel für solche Plattformen ist z.B. die prämierte Stromhandelsplattform Entras (www.entras.de).

3.3 LARS

Das Produkt „living markets“ der living systems AG besteht aus einem Kern, welcher die wesentlichen Aufgaben der gesamten Kommunikation und der Steuerung der Agenten übernimmt. Dieser Kern wird kurz LARS genannt und ist die Abkürzung für „Living Agents Runtime System“.

Das Konzept beinhaltet einen Basisagenten, welcher grundsätzliche Aufgaben übernimmt und diese bereits implementiert. Von diesem Basisagenten erben alle weiteren Agenten. So versteht beispielsweise jeder Agent eine Nachricht, die den Betreff, oder Dienst „ping“ hat. Also kann jeder Agent, der von diesem Basisagenten erbt auf die Nachricht automatisch antworten, ohne dies extra implementieren zu müssen.

Beim Start der Plattform wird nur ein einziger Agent fest eingebunden, also gestartet. Dieser Agent liest seinerseits eine Konfigurationsdatei im XML Format und startet anhand dieser Datei dann weitere Agenten. Der Agent heißt „AgentManager“ und unterstützt weitere Fähigkeiten, wie Agenten und Gruppen von Agenten zu starten, zu stoppen, zu beenden, „load-balancing“ zu betreiben und so weiter. Jeder gestartete Agent hat die Möglichkeit seine eigene Konfigurationsdatei einzulesen und auszuwerten.

3.3.1 Agenten-Kommunikation

Eine der grundlegenden Schichten der living agents Architektur ist die Kommunikationsschicht. Sie basiert auf grundlegenden Kommunikationsprotokollen, wie RMI, Sockets und einem internen, optimierten Protokoll für den XML-basierten Nachrichtenaustausch. Auch die Konfiguration der Plattform und ihrer Agenten geschieht mittels XML-Dokumenten, die eine Sammlung von Nachrichten darstellen, die beim Starten des Systems ausgelesen und sequentiell an die zuständigen Agenten gesendet werden.

Jeder Agent hat einen sogenannten „Briefkasten“, in dem die zu bearbeitenden Nachrichten abgelegt werden. Diese Nachrichten werden in regelmäßigen Abständen ausgelesen und interpretiert. Dabei reichen die Nachrichteninhalte von der einfachen Initialisierung des Agenten bis hin zu Statusabfragen oder anderer komplexen Aufgaben. Außerdem unterstützt das LARS die optionale Ver- und Entschlüsselung von Nachrichten mit einer Kombination von RSA und Blowfish Key Algorithmus, um die Vorteile beider Algorithmen optimal zu nutzen.

Auch werden verschiedene Nachrichtentypen unterstützt um eine höhere Effektivität zu erreichen. So gibt es Nachrichten an genau einen Agenten, an Agentengruppen und Broadcast-Nachrichten, die an alle Agenten einer Plattform versendet werden. Gerade die Funktionalität der Gruppenbildung ist für größere Plattformen aus Performanzgründen interessant, weil es dort oft Agenten mit ähnlichen Aufgaben gibt und diese gleichzeitig erledigt werden können.

3.3.2 System-Agenten

Die Agenten des LARS lassen sich in verschiedene Typen klassifizieren. Ein Typ sind die so genannten System-Agenten, die entsprechend ihrer Konfiguration anderen Agenten grundlegende Dienste bereitstellen. Dazu gehören unter anderem verschiedene Kommunikationsmöglichkeiten mit anderen LARS-Plattformen oder anderen externen Plattformen. Diese Funktionalitäten werden durch die zugehörigen System-Agenten gekapselt, so dass z.B. Socket-Verbindungen vom zuständigen Agenten transparent für die anderen Agenten aufgebaut werden und danach als Kommunikationskanal genutzt werden können. Um diese Kommunikation durchführen zu können, arbeiten im Hintergrund verschiedene Komponenten zusammen. Es gibt den sogenannten „Listener“, welcher den Kommunikationskanal auf ankommende Nachrichten abhört und den dazugehörigen „Messenger“ der die empfangenen Nachrichten im Briefkasten des jeweiligen Agenten ablegt. Auch in diesem Kommunikationssystem gibt es einen Router, der seinerseits durch einen Agenten gekapselt wird und die Nachrichten je nach Typ weiterleitet. Dieser Router trägt im LARS den Namen „MessageRouter“.

Verantwortlich für das Starten der verschiedenen Agenten mit der entsprechenden Konfiguration und den notwendigen Parametern beim Hochfahren der Plattform, ist der sogenannte AgentManager. Dazu liest der AgentManager beim Start der jeweiligen Agenten deren Konfigurationsdatei aus und initialisiert sie mit den entsprechenden Nachrichten. Durch den Einsatz der Java Reflection API, die auf ähnliche Weise auch bei der JavaBeans-Technologie eingesetzt wird, können zur Laufzeit relevante Methoden ausfindig gemacht und zur späteren Benutzung registriert werden. Durch diesen Vorgang wird die Speicherung redundanter

Informationen und somit auch unnötiger Fehlerquellen verhindert, da sich eine separate Konfiguration erübrigt.

Um bestimmte Aktionen zeitabhängig ausführen zu können, wie z.B. das regelmäßige Update der Börsendaten, gibt es den Timer-Agent. In größeren Systemen können bei solchen Aktionen mehrere Datenbanken und eventuell sogar mehrere Plattformen involviert sein. Die wichtige Aufgabe der Transaktionskontrolle erledigt hierbei wiederum ein System-Agent. Weitere Aufgaben von System-Agenten sind neben der Bereitstellung von gesammelten System-Informationen Dinge wie Load-Balancing und Monitoring.

3.3.3 Applikations-Agenten

Eine weitere wesentliche Rolle beim LARS übernehmen die Applikations-Agenten, die hier die Schicht der Applikations-Logik repräsentieren und für die Erledigung spezifischer Aufgaben wie das Registrieren von Benutzern oder das Eintragen von Auktions-Geboten verantwortlich sind. Hier setzt auch die Arbeit der verschiedenen Projekte an, die eventuell noch nicht vorhandene Agenten implementieren oder vorhandene Agenten an die Bedürfnisse des jeweiligen Kunden anpassen. Um diese Schicht zu realisieren, gibt es zwei Möglichkeiten. Zum einen durch die Implementierung der Logik in den Agenten selbst oder durch den Zugriff auf externe Software-Komponenten, wie beispielsweise Enterprise Java Beans oder CORBA. Um diese Aufgaben zu bewältigen, haben die Applikations-Agenten Zugriff auf die System-Agenten, um unter anderem Zugriff auf die Persistenzschicht zu erhalten.

3.3.4 Komponenten-Integration

Um ein API zu einem externen System zu implementieren, gibt es beim LARS Agenten, die unter dem Namen „Connectors“ bekannt sind. Sie führen Funktionen wie etwa die Datenkonvertierung zu XML des SAP-Dateiformates durch.

Es gibt aber auch Agenten, die über ein bestimmtes Protokoll mit externen Software-Komponenten kommunizieren, ohne die Daten selbst zu manipulieren. Sie heißen „Adapters“ und ermöglichen die Auslagerung der Geschäftlogik und der Persistenzschicht in andere Software-Komponenten.

3.4 Logic Engine

Die Logic Engine ist ein Framework, welches es ermöglicht, Wahrnehmungen aus der Umgebung aufzunehmen und Entscheidungen auf bestimmte Änderungen in der Umgebung zu treffen. Dabei ist die Auswahl der auszuführenden Aktion nicht abhängig von der Logic Engine selbst, sondern von der implementierten Technologie, welche im folgenden erläutert werden. Die Logic Engine bietet die benötigten Schnittstellen zur Aufnahme von Wahrnehmungen und Ausführung von Aktionen. Ein genauerer Einblick in die Logic Engine selbst ist in Kapitel 4 zu finden.

Innerhalb des Produktes der living systems AG existieren bereits zwei Implementierungen für autonome Handlungskontrolle, welche ebenfalls auf der „Logic Engine“ basieren. Dabei handelt es sich um die „Workflow Engine“ und die „EBN Engine“ (EBN = Extended Behavior Network).

Außer den beiden bereits implementierten Technologien ist eine weitere Technologie im Rahmen einer Diplomarbeit im Entstehen, die sogenannte „Decision Tree Engine“. Somit werden nach Abschluss der Diplomarbeiten vier Ansätze zur Handlungskontrolle bzw. zur Repräsentation von business knowledge existieren, die für unterschiedliche Umgebungen geeignet sind.

Diese vier Ansätze kann man zunächst in zwei Gruppen unterteilen. Zum einen in reaktive und zum anderen in proaktive Ansätze zur Handlungskontrolle. Reaktiv sind die Workflow Engine und die Decision Tree Engine, proaktiv sind die EBN Engine und die Planning Engine.

3.4.1 Reaktiver Ansatz

Reaktive Agenten handeln nach Ereignissen, die in der Umgebung eintreten, und nach dem vorgegebenen Wissen, wie auf bestimmte Ereignisse reagiert werden soll. Sie handeln sofort auf ein neues Ereignis, ohne sich darüber Gedanken zu machen, welche Auswirkungen diese Aktionen auf die Umgebung haben.

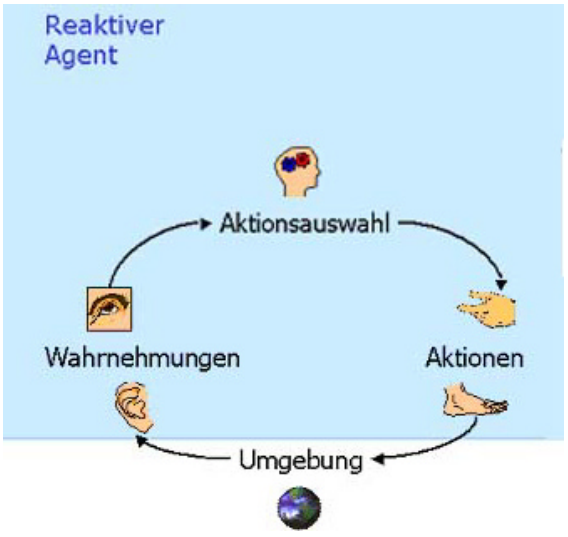

Abbildung 9: Funktionsweise eines reaktiven Agenten[21]

Der reaktive Ansatz ist für dynamische Umgebungen geeignet, die sich durch externe Einflüsse schnell ändern können. Der Agent muss schnell auf Ereignisse reagieren, er kennt seine Ziele nicht.

[21] Bild aus: Artikel von Dannegger und Dorer: *Java Magazin*. Agenten in aller Munde, Ausgabe 7, 2001

Workflow Engine
Die Workflow Engine ist mit einer einfachen Programmiersprache vergleichbar. Es gibt drei unterschiedliche Arten von Kontrollstrukturen.

- sequentielles Ausführen von Aktionen
- bedingtes Abarbeiten vordefinierter Aktionen anhand der zugrunde liegenden Zustände in der Umgebung (Beispiel: Wenn Zustand A wahr ist, wird Aktion 1 ausgeführt, andernfalls Aktion 2)
- wiederholtes Ausführen von Aktionen solange bestimmte Zustände herrschen (Beispiel: solange Zustand A wahr ist, führe Aktion 1 aus

Die verschiedenen Möglichkeiten sind miteinander kombinierbar, so dass beispielsweise innerhalb einer Schleife eine Sequenz von Aktionen ausgeführt werden kann.

Die Workflow Engine ist für dynamische Umgebungen geeignet. Da sie sich keine Gedanken darüber macht, welchen Einfluss eine Aktion auf die Umgebung hat, ist der benötigte Speicherbedarf sehr gering und die Handlung sehr schnell. Die Workflow Engine handelt programmgesteuert und nicht zielgerichtet, sie hat keine explizite Repräsentation des Ziels.

Decision Tree Engine
Bei der Decision Tree Engine werden verschiedene Faktoren zusammen genommen und aufgrund dieser Faktoren gehandelt. Wie der Name schon sagt, entscheidet sich die Engine anhand eines vorgegebenen Entscheidungsbaumes. Die Umgebung wird sozusagen ausgekundschaftet und dann entschieden, was gemacht wird.

Beispiel: Um entscheiden zu können, ob man Federball spielt, ins Schwimmbad geht oder einfach zu Hause bleibt, werden vorgegebene Attribute untersucht. Die Attribute spiegeln sich in Propositionen wie: „es ist schönes Wetter“, „es ist windig“, „es regnet“, ... Aus den zusammengesetzten Attributen, die für die Entscheidung der Handlung wichtig sind, entsteht ein Entscheidungsbaum. So ist es sowohl für Federball spielen, als auch für das Schwimmbad wichtig, dass gutes Wetter herrscht. Wenn es jedoch zum schönen Wetter noch windig ist, ist Federball spielen ungeeignet. Aufgrund der Kombination verschiedenster Zustände wird entschieden, welche Aktion ausgeführt werden soll.

Die Decision Tree Engine handelt aufgrund der Zustände in der aktuellen Umgebung. Es wird immer nur für den aktuellen Zustand entschieden. Daher kann diese Engine sofort nach einer Änderung der Umgebung reagieren. Der Speicherbedarf hält sich durch die relativ einfache Struktur in Grenzen. Es können von vornherein Zweige im Entscheidungsbaum ausgegrenzt werden, wenn durch eine unwahre Proposition die Bedingung für die Ausführung einer Aktion nicht zutrifft. Die Engine handelt nicht zielgerichtet, sie hat keine explizite Repräsentation des Ziels.

3.4.2 Proaktive Agenten

Proaktive Agenten kennen die Effekte, die ihre Aktionen auf die Umgebung haben. Somit kann ein solcher Agent bereits vor Ausführung einer Aktion überlegen, was mit der Umgebung geschieht, wenn die Aktion ausgeführt wird. Er kann daher aus allen möglichen Aktionen die „beste“ aussuchen und vorausschauend handeln.

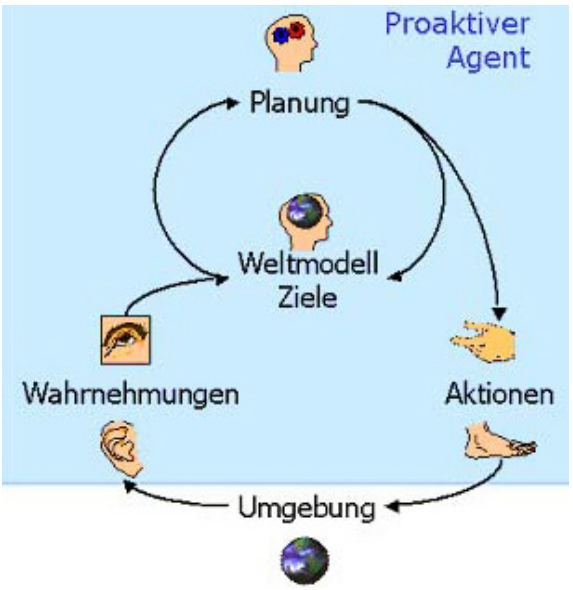

Abbildung 10: Funktionsweise eines proaktiven Agenten[22]

Der proaktive Ansatz ist für Umgebungen geeignet, die sich durch externe Einflüsse wenig oder gar nicht ändern, bzw. bei denen dem Agenten genügend Zeit bleibt zu planen.

EBN Engine

Die EBN Engine handelt direkt auf eine sich ändernde Umgebung. Sie weiß vorher, was für Auswirkungen Aktionen auf die Umgebung haben. Sie handelt zielgerichtet, hat das Ziel immer vor Augen und führt die Aktion aus, die im Moment am Besten ist, um die Ziele zu erreichen.

Ein gutes Beispiel für den Einsatz einer EBN Engine ist ein Fußballspiel. Die living systems AG nimmt regelmäßig an der Weltmeisterschaft für Fußball spielende Agenten „RoboCup“ mit Erfolg teil. Ein Fußballagent kann die Aufgabe haben, zum Tor zu stürmen und ein Tor zu schießen. Ein Gegenspieler kann beim Stürmen in die Linie des stürmenden Agenten laufen, wodurch dieser seine Richtung wechseln sollte, um einen Ballverlust zu vermeiden.

Externe Einflüsse können eine Änderung der Aktionsfolge hervorrufen. Treten keine externen Einflüsse ein, hat der Agent jedoch immer noch die Aufgabe ein Tor zu schießen. Da die Aktionen anderer Agenten nicht vorhersagbar sind, kann auch kein vollständiger Plan erstellt werden, welche Aktionen am schnellsten zum Ziel führen. Durch die dynamische Umgebung ist ein schnelles „Umdenken“ der auszuführenden Aktionen nötig.

---

[22] Bild aus: Artikel von Dannegger und Dorer: *Java Magazin*. Agenten in aller Munde, Ausgabe 7, 200

Da immer nur für eine oder einige wenige Aktionen voraus geplant wird, ist der Speicherbedarf sehr gering und die Handlungen erfolgen sofort. Aufgrund dieser Tatsachen und aufgrund der Tatsache, dass externe Einflüsse die Umgebung ändern, wird das Ziel nicht immer und eventuell nicht auf dem optimalen Weg erreicht.

Planning Engine

Die Planning Engine ist nur für Umgebungen geeignet, die sich durch externe Einflüsse gar nicht oder nur sehr langsam ändern. Da die Planning Engine umfassendes Wissen über die Umgebung besitzt, plant sie vom Beginn bis zum Ziel, sofern es einen Weg zum Ziel gibt. Es werden also keine Aktionen ausgeführt, bevor nicht der genaue Weg zum Ziel bekannt ist.

Der gesamte Weg zum Ziel muss gespeichert werden. Während der Suche nach dem Ziel muss sogar der gesamte Suchbaum gespeichert werden, wodurch der Speicherbedarf sehr hoch ist. Aufgrund der Komplexität des Suchbaums ist der Zeitbedarf ebenfalls sehr hoch im Vergleich zu anderen Technologien. Jedoch wird der optimale Weg zum Ziel gefunden, sofern es einen gibt. Die Funktionsweise der Handlungsplanung wurde in Kapitel 2 bereits umfassend beschrieben.

3.4.3 Gegenüberstellung

Ansatz	Engine	Vorteile	Nachteile
reaktiv	Workflow	dynamisch wenig Speicherbedarf	nicht optimal nicht vollständig
	Decision Tree	mittlerer Speicherbedarf	statisch nicht optimal nicht vollständig
proaktiv	EBN	dynamisch wenig Speicherbedarf zielgerichtet	nicht optimal nicht vollständig
	Planning	optimal vollständig zielgerichtet	statisch hoher Speicherbedarf zeitintensiv

Tabelle 3: Gegenüberstellung von Handlungsansätzen

Kapitel 4

Planung und LARS konzeptionell

In diesem Kapitel wird auf das konzeptionelle Zusammenspiel der Handlungsplanung und der LARS-Plattform insbesondere auf das Framework der Logic Engine eingegangen. Der Aufbau der Logic Engine, Planning Engine und die konzeptionelle Darstellung des Weltzustandes, der Aktionen und des Planens werden erläutert.

4.1 Design der Logic Engine

Wenn man sich die Integration betrachtet, fällt auf, dass die oben erwähnten Technologien Gemeinsamkeiten aufweisen. Diese Gemeinsamkeiten liegen in den Wahrnehmungen (*engl. perception*), und in den Fähigkeiten (*engl. capability*). Alle Ansätze müssen dazu in der Lage sein, Wahrnehmungen aus der Umgebung aufzunehmen und mit ihren Aktionen die Umgebung zu beeinflussen. Die wichtigste Gemeinsamkeit der Ansätze ist, dass aufgrund der Wahrnehmungen entschieden wird, wie der Agent handeln soll.

Daraus kann Standardfunktionalität in die Logic Engine positioniert werden. Agenten können Wahrnehmungen aufnehmen und diese entsprechend interpretieren. Sie können Aktionen ausführen, welche die Umgebung entsprechend beeinflussen. Dadurch können dieselben Wahrnehmungen und Aktionen für die verschiedenen *decision engines* verwendet werden. Da die Technologien für unterschiedliche Umgebungen geeignet sind, macht es wenig Sinn, genau dieselben Agenten zu verwenden. Deshalb sind die Agenten konfigurierbar, wodurch sie unabhängig von der eigentlichen Ausführung der Aktionen werden.

Die Implementierung der Wahrnehmungen und Aktionen sind letztendlich unabhängig von der verwendeten Handlungskontrolle. Allen Ansätzen ist gemeinsam, dass aufgrund der Wahrnehmung der Umgebung entschieden wird, was der Agent tun soll. Die Entscheidung kann der Agent in die Tat umsetzen, indem er Aktionen ausführt. Die Wahrnehmungen und Aktionen gelten für alle Ansätze der Handlungskontrolle gleichermaßen, unabhängig davon, wie man zu den auszuführenden Aktionen gelangt ist. Daraus ergibt sich auch ein entsprechendes Klassenmodell:

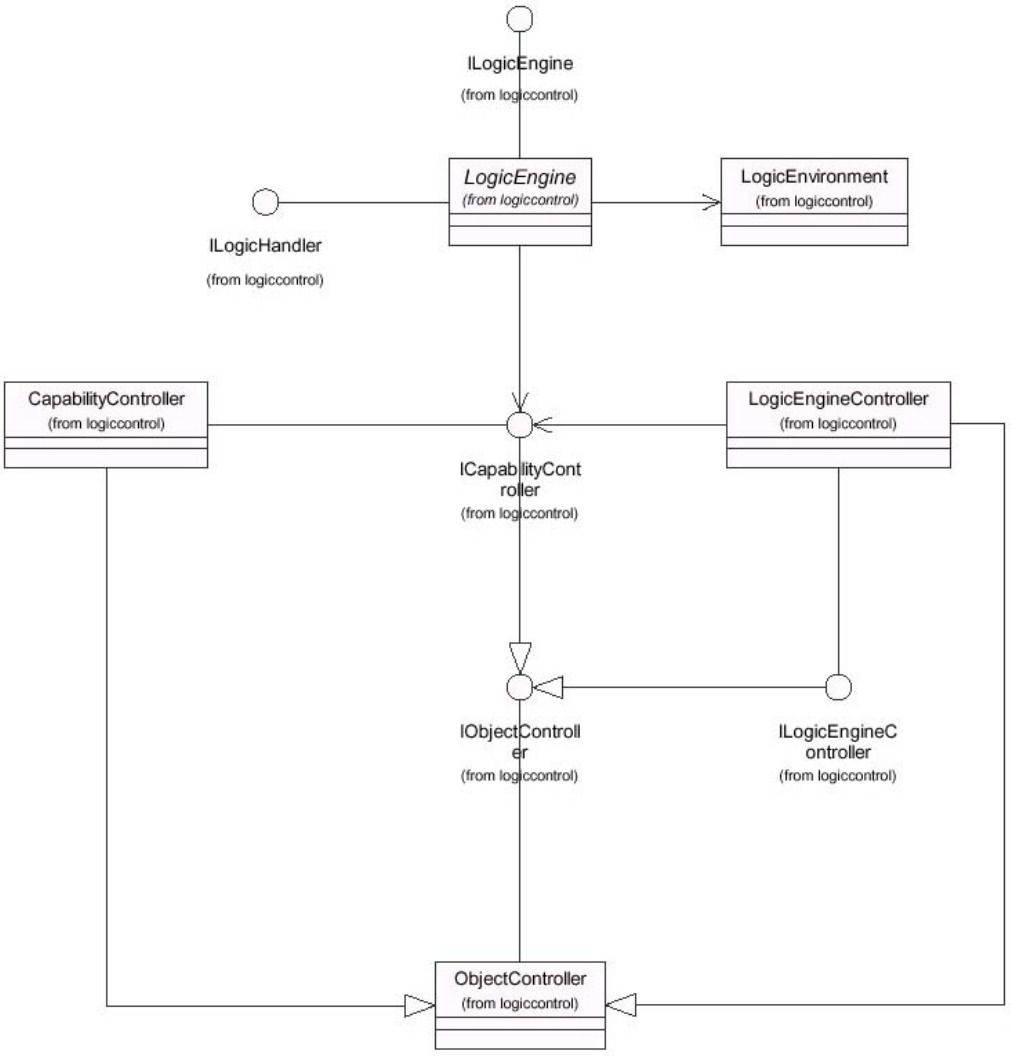

Abbildung 11: Ausschnitt des Klassenmodells der Logic Engine

Wie im Klassenmodell zu sehen ist, wird Standardfunktionalität bereits in der Logic Engine integriert. Dabei handelt es sich nicht um Implementierungen der einzelnen Aktionen, oder des Weltzustandes, sondern vielmehr um die Steuerung dieser.

Die Klasse LogicEngine implementiert aus dem Interface ILogicEngine Methoden, die für das Starten, das Stoppen, das Konfigurieren, usw. der Logik zuständig sind. Aus dem Interface ILogicHandler werden die drei Methoden zur Steuerung der Wahrnehmungen und Aktionen in der Klasse LogicEngine implementiert. Dabei handelt es sich um „performLogic“, „performAction“ und „performPerception“. Die Klasse LogicEnvironment ist für die Speicherung und den Zugriff auf sämtliche Daten einer Instanz von AgentLogic zuständig. Die Klasse ObjectController implementiert die Methoden des Interfaces IObjectController. Der ObjectController ist zuständig für die Instanziierung von Objekten und Registrierung der Methoden und dient als Anbieter der registrierten Methoden. Die Klasse CapabilityController implementiert das Interface ICapabilityController und dient als Anbieter für die Aktionen. Dies beinhaltet ebenfalls die Registrierung und Steuerung der Methoden für die Capabilities.

Die Klasse LogicEngineController verwaltet unterschiedliche logic engines, die innerhalb des Frameworks gestartet worden sind.

4.2 Integration

Eine Implementierung einer konkreten Logic Engine kann von der Klasse Logic Engine erben. Das heißt, dass Funktionen, die in der Logic Engine implementiert sind auch für davon abgeleitete Klassen zur Verfügung stehen. Wenn von der Logic Engine geerbt wird, müssen einige Funktionen noch implementiert werden. Diese Methoden sind in der Logic Engine abstrakte Methoden. Sie werden dort lediglich definiert, die eigentliche Implementierung muss in abgeleiteten Klassen vorgenommen werden. So kann die Klasse „PlanningEngine“ von der Klasse Logic Engine abgeleitet werden und die Methoden „performPerception“, „performAction“ und „performLogic“ implementieren.

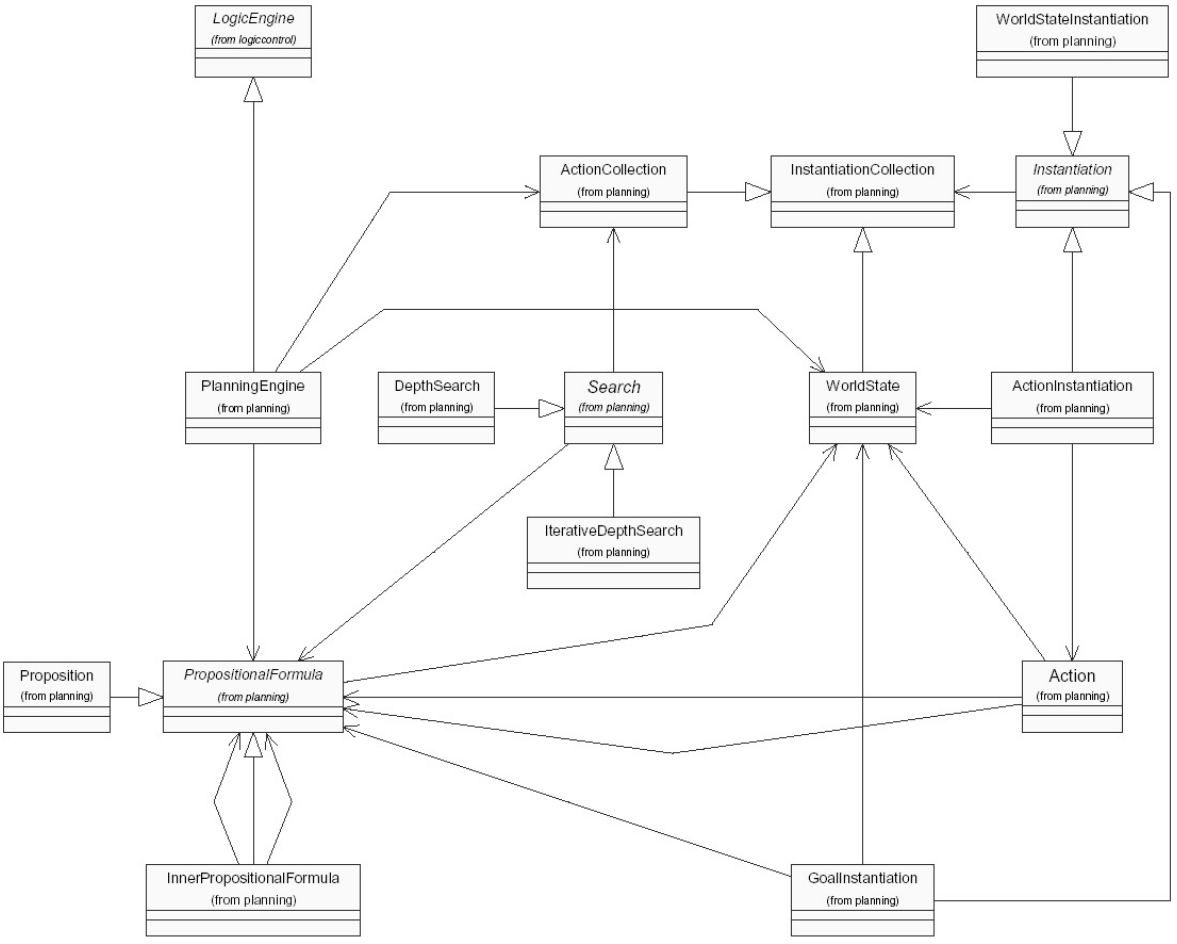

Abbildung 12: Klassenmodell der PlanningEngine

Wie im Klassendiagramm zu sehen ist, erbt die Klasse PlanningEngine von der Klasse Logic Engine. Somit muss die PlanningEngine die Methoden „performLogic“, „performAction“ und „performPerception“ implementieren.

Mit Hilfe der Methode „performLogic“ werden der PlanningEngine die Propositionen übergeben, die die Umgebung repräsentieren und die Suche nach dem Plan wird gestartet. Die PlanningEngine generiert den Weltzustand, die Aktionen mit ihren Vorbedingungen und Effekten und den Zielzustand. Mit den Zuständen aus „performPerception“ kann der Weltzustand, der gerade vorliegt, an den in der PlanningEngine generierten Weltzustand angepasst werden. Von diesem Ausgangspunkt kann geplant werden.

Wie in der PlanningEngine der Weltzustand, der Zielzustand und die Aktionen repräsentiert werden, wird in Kapitel 5 eingehend beschrieben. Der gefundene Plan kann an die Methode „performAction“ übergeben werden, wodurch die Aktionen tatsächlich ausgeführt werden.

4.2.1 Weltzustand

Der Weltzustand ist gegeben durch die möglichen Zustände der vorhandenen Objekte. Die einzelnen Zustände werden als Wahrheitswert von Propositionen repräsentiert. Diese müssen vor dem Planen entsprechend der aktuellen Situation aktualisiert werden. Dazu wird die Methode „performPerception“ verwendet. Mit Hilfe dieser Methode können die Zustände der einzelnen Propositionen mit dem aktuellen Zustand der Umgebung abgeglichen werden. Der aktuelle Weltzustand wird somit an die PlanningEngine übertragen. Sie befindet sich nun im Ausgangszustand. Aufgrund der jetzt bekannten Umgebung und der bereits bekannten Aktionen mit ihren Vorbedingungen und Effekten kann geplant werden.

4.2.2 Planen

Das Planen selbst ist unabhängig von der LogicEngine. Wann ein Plan erstellt werden soll wird aber von der LogicEngine durch Aufruf der Methode „performLogic“ festgelegt. Während des Planens werden keinerlei Änderungen der Umgebung wahrgenommen. Das Planen beeinflusst die Umgebung auch nicht. Die Aktionen mit ihren Vorbedingungen und Effekten sind hinreichend bekannt, so dass während des Planens keine Verbindung zur LogicEngine benötigt wird. So kann wie in Kapitel 2 beschrieben nach dem Weg zum Ziel, ohne andere Einflüsse zu berücksichtigen, gesucht werden.

Die Aktionen, die zur Erreichung des Ziels ausgeführt werden müssen, werden in einer Liste, welche den Plan repräsentiert, gespeichert. Diese Liste kann dazu verwendet werden, im Anschluss an das Planen, die Aktionen real auszuführen.

4.2.3 Aktionen

Um später die Aktionen ausführen zu können, müssen diese programmiert werden. Dabei gibt es keine allgemein gültige Methode, welche für alle Aktionen in beliebigen Umgebungen gilt. Für jede neue Umgebung müssen neue Aktionen programmiert werden. Das Suchen nach einem Plan selbst ist durch die Beschreibung und das virtuelle Ausführen von Aktionen Domänen-unabhängig.

Der gefundene Plan, also die Liste der nacheinander auszuführenden Aktionen, kann der Methode „performAction“ übergeben werden, damit die Aktionen real ausgeführt werden.

Dabei ist es nicht vorgesehen, die Zustände während der Ausführung des Plans nach jeder Aktion zu überprüfen. Wenn es sich um eine statische und deterministische Umgebung handelt, stimmen die erwarteten Zustände mit den realen Zuständen überein. Mit Ausnahme des zusätzlichen Zeitbedarfs spricht jedoch nichts dagegen, eine zusätzliche Kontrolle der Zustände vorzunehmen.

Kapitel 5

Implementierung

In diesem Kapitel wird die Implementierung der PlanningEngine im Detail beschrieben. Dabei wird die konkrete Repräsentation des Weltzustandes, der Aktionen mit ihren Vorbedingungen und Effekten und das Planen erläutert.

5.1 Weltzustand

Der Weltzustand wird durch einzelne Propositionen beschrieben. Diese können wahr oder falsch sein. Die Schwierigkeit bei der Implementierung des Weltzustands liegt in der Darstellung aller möglichen Kombinationsmöglichkeiten von Propositionen, die in der Welt als Zustände eintreten können.

So gibt es für eine Blockwelt mit zwei Blöcken insgesamt elf Propositionen, welche wahr oder falsch sein können.

- on_table_block-a
- on_table_block-b
- on_block-a_block-a
- on_block-a_block-b
- on_block-b_block-a
- on_block-b_block-b
- clear_block-a
- clear_block-b
- holding_block-a
- holding_block-b
- handempty

Diese Propositionen können die Zustände wahr oder falsch annehmen. In einer gültigen Umgebung wären die Zustände dann beispielsweise wie folgt definiert:

```
on_table_block-a        = true
on_table_block-b        = false
on_block-a_block-a      = false
on_block-a_block-b      = false
on_block-b_block-a      = false
on_block-b_block-b      = false
clear_block-a           = false
clear_block-b           = true
holding_block-a         = false
holding_block-b         = false
handempty               = true
```

Listing 3: Weltzustand der Blockwelt mit zwei Blöcken

Bei der Darstellung der Domäne wäre es sehr aufwändig, diese Propositionen alle einzeln zu definieren. In der Blockwelt würden bei fünf vorhandenen Objekten (Blöcke) 41 Propositionen entstehen. Die 41 Propositionen können wahr oder falsch sein. Die Propositionsvielfalt, also die Anzahl unterschiedlicher Weltzustände würde dann bei $propositions^2 = 1681$ liegen. Deshalb benutzt man für die Propositionen Variable. Durch die Verwendung von Variablen gibt es immer nur fünf Propositionen. Diese setzen sich zusammen aus:

- on_block-x_block-y
- ontable_bloxk-x
- holding_block-x
- clear_block-x
- handempty

Zusätzlich werden die vorhandenen Objekte definiert. Die Variable können in allen Kombinationsmöglichkeiten durch die Objekte ersetzt werden. Dadurch kann die vorhandene Umgebung sehr schnell mit einem zusätzlichen Block erweitert werden. Das Softwareprogramm kann durch rekursive oder iterative Funktionen, die verschiedenen Kombinationsmöglichkeiten nachbilden.

Für den Prototyp wurde eine XML-Repräsentation verwendet, welche die Propositionen mit allen Objekten darstellt:

```
<objects>
      <object>block_a</object>
      <object>block_b</object>
      <object>block_c</object>
</objects>

<propositions>
      <proposition>
            <name>on</name>
            <parameter>x-block</parameter>
            <parameter>y-block</parameter>
      </proposition>
      <proposition>
            <name>ontable</name>
            <parameter>x-block</parameter>
      </proposition>
      <proposition>
            <name>clear</name>
            <parameter>x-block</parameter>
      </proposition>
      <proposition>
            <name>handempty</name>
      </proposition>
      <proposition>
            <name>holding</name>
            <parameter>x-block</parameter>
      </proposition>
</propositions>
```

Listing 4: Beispiel der Propositionen aus der Blockwelt

Durch Hinzufügen eines weiteren Blocks zwischen den object-Tags ist der komplette Weltzustand bereits erweitert. Wenn nun die tatsächlichen Zustände der Objekte noch ausgelesen werden, wie es in Kapitel 4 beschrieben ist, hat man den aktuellen Weltzustand mit seinen Wahrheitswerten.

Nun stellt sich die Frage, wie der Weltzustand im Speicher repräsentiert wird. Während der Entwicklungsphase wurde zunächst eine *Map* benutzt. Eine *Map* besteht aus Schlüssel-Werte-Paaren, wobei der Schlüssel der Proposition und der Wert dem Zustand der Proposition entsprechen. Ausgehend vom Blockweltbeispiel könnte dann „ontable_block-a“ ein Schlüssel sein, und der Wert des Schlüssels wahr oder falsch sein. Am Prototyp stellte sich schnell heraus, dass eine solche Map zu viel Speicher und Zeit benötigt, für das Anlegen der Propositionen und den Zugriff auf diese. Im Versuch benötigte das Programm, beim Anlegen eines Weltzustandes mit ca. 280.000 verschiedenen Kombinationen von Zuständen, ca. sieben Sekunden bei der Darstellung als Map und 500 Millisekunden bei der Darstellung als Array. Dieser Zustand trat bei einer Proposition mit sechs Variablen und sieben Objekten ein. Schon bei acht Objekten in einer Proposition war der Speicherbedarf so groß, dass der Speicher im Rechner nicht mehr ausreichend war und der Rechner deshalb so sehr mit auslagern auf die Festplatte beschäftigt war, dass nach ca. zwei Stunden der Weltzustand immer noch nicht repräsentiert war. Deshalb wurden für die Repräsentation sogenannte Arrays verwendet.

Für jede Proposition kann ein neues Array vom Datentyp *boolean* angelegt werden. Abhängig davon, wie viele Variable eine Proposition besitzt und wie viele Objekte existieren, muss das Array unterschiedlich groß sein. Hat man eine Proposition, welche drei Variable besitzt und gibt es in der Domäne fünf Objekte, so ist das Array $objekte^{variable} = 125$ Speicherstellen groß. Der Datentyp boolean benötigt 1 Bit. Der verwendete Speicherplatz ist somit 125 Bit groß, während eine Map ein komplexer Datentyp (Objekt) ist und daher wesentlich mehr Speicher benötigt. Besitzt eine Proposition keine Variable, wie beispielsweise „handempty", so benötigt diese Proposition lediglich 1 Bit Speicherplatz, unabhängig davon, wie viele Objekte in der Domäne existieren.

Entscheidend für die Zuordnung der Proposition ist der Index des Arrays. Je nach Reihenfolge kann beim Blockweltbeispiel die Proposition „ontable_block-a" dem Index 0 und „ontable_block-b" dem Index 1 zugeordnet werden. Um diese Zuordnung entsprechend interpretieren zu können, muss diese extra gespeichert werden. Da man eine Liste von Objekten hat, ist die Reihenfolge immer dieselbe. Dabei genügt es, die Position der Objekte innerhalb der Objektliste zu speichern und später auszulesen. Dadurch kann die Zuordnung wieder rückgängig gemacht werden. Für das Planen genügt der Index. Es ist nicht notwendig, zu wissen, welche Proposition tatsächlich hinter einem Index steckt. Will man aber später wieder darstellen, um welche Proposition es sich handelt, so muss diese Zuordnung wieder rückgängig gemacht werden.

Die folgende Tabelle zeigt die Zusammenhänge der Arrays am Beispiel der Blockwelt mit zwei Blöcken (A, B):

0: on	0: Block A, Block A	wahr
		falsch
	1: Block A, Block B	wahr
		falsch
	2: Block B, Block A	wahr
		falsch
	3: Block B, Block B	wahr
		falsch
1: ontable	0: Block A	wahr
		falsch
	1: Block B	wahr
		falsch
2: clear	0: Block A	wahr
		falsch
	1: Block B	wahr
		falsch
3: holding	0: Block A	wahr
		falsch
	1: Block B	wahr
		falsch
4: handempty	0:	wahr
		falsch

Tabelle 4: Array Verschachtelung bei 2 Objekten

Zu Beginn brauchen die Arrays nicht explizit zugeordnet werden. Beim Anlegen der Arrays genügt es ein *boolean* Array anzulegen, welches $objekte^{variable}$ Speicherstellen zur Verfügung stellt. Die Zuordnung der Wahrheitswerte geschieht, wie in Kapitel 4 beschrieben, über die Abfrage der Zustände einzelner Propositionen. Sind die Zuordnungen gesetzt, kann der Weltzustand für die Planung verwendet werden. Während des Planens wird aufgrund der statischen Umgebung keine Änderung der Propositionen mehr berücksichtigt.

Für die Simulation der Blockwelt können diese Propositionen zu Beginn eingelesen werden. Im obigen XML-Listing (Listing 4) sind alle möglichen Propositionen dargestellt, welche die Umgebung beschreiben. Um die Menge der Objekte zu vernachlässigen, werden für die Propositionen Variable verwendet, welche bei der Benutzung durch die tatsächlichen Objekte ersetzt werden. Die Blockwelt kann nur simuliert werden, da man ansonsten Sensoren benötigen würde, welche die Zustände der Blöcke liefern und Roboterarme, welche die Position der Blöcke ändern.

Prinzipiell werden alle möglichen Propositionen in einer Umgebung dargestellt. Das heißt, dass alle Propositionen wahr werden können. Dies gilt auch, wenn nach menschlichem Ermessen ein Zustand keinen Sinn ergeben würde. So ist beispielsweise die Proposition „on_block_a_block_a“ in der Blockwelt nie wahr, da ein Block nicht auf sich selbst liegen kann. Ohne Domänenwissen kann ein Softwareprogramm an dieser Stelle nicht unterscheiden, was sinnvoll ist und was nicht. Deshalb werden alle möglichen Kombinationen dargestellt.

Weiterhin gibt es in der Beschreibung des Weltzustandes ungültige Zustände. Ungültige Zustände sind beispielsweise Zustände, wo sich die Wahrheitswerte von Propositionen gegenseitig ausschließen. Am Blockweltbeispiel gesehen, kann ein Block A nicht gleichzeitig vom Greifer in der Luft gehalten werden und der Block sich auf einem Stapel anderer Blöcke befinden, oder gar zwischen anderen Blöcken sein. Die möglichen Zustände in denen man sich befinden kann, sind extrem groß. Es gibt keine allgemein gültigen Regeln, welche ungültige Zustände von vornherein ausschließen. Man hat lediglich die Möglichkeit, gewisse Regeln mit Hilfe von XML-Beschreibungen aufzustellen, wodurch zumindest ein Teil ungültiger Zustände ausgeschlossen werden könnte. Im Fall der Blockwelt könnte eine Regel aufgestellt werden, wonach beispielsweise für ein und dasselbe Objekt die Proposition „ontable“ und „on“ nicht gleichzeitig wahr sein dürfen. In der Implementierung wurde auf eine solche Konsistenzkontrolle verzichtet, wenngleich einige prinzipielle Fehler in den Domänen-Beschreibungen abgefangen werden können. Dies gilt jedoch nicht im Bezug auf ungültige Zustände, da diese sehr schwer, wenn nicht gar unmöglich, zu erkennen sind.

5.2 Aktionen

Die Darstellung der Aktionen gestaltet sich etwas komplizierter, als die des Weltzustandes. Die Aktionen bestehen aus Namen, Vorbedingungen und Effekten. Sowohl die Vorbedingungen, wie auch die Effekte bestehen aus Propositionen, die in booleschen Formeln miteinander verknüpft werden. In Vorbedingungen ist im Gegensatz zu Effekten auch

Disjunktion, also die „oder“-Verknüpfung von Bedingungen erlaubt. So kann beispielsweise die Aktion „kaufe Milch“ die Vorbedingung „befinde dich in Supermarkt A“, oder „befinde dich in Supermarkt B“ besitzen.

Sowohl Vorbedingungen, als auch Effekte können in ihren Propositionen negiert sein. So kann die Aktion „gehe zum Supermarkt A“ die Vorbedingung „befinde dich nicht in Supermarkt A“ haben. Das Greifen eines Blocks hat den negierten Effekt „nicht handfrei“.

Die Propositionen der Vorbedingungen und Effekte entsprechen exakt denen des Weltzustandes. Um nicht für jede Aktion mit ihren Propositionen eine neue Aktion schreiben zu müssen, werden wie beim Weltzustand, ebenfalls Variable verwendet. Sonst müsste man für jeden Block ein und dieselbe Aktion extra beschreiben. Im Blockweltbeispiel gibt es insgesamt vier unterschiedliche Aktionen. Dabei handelt es sich um:

- stack
- unstack
- pickup
- putdown

Die Aktionen pickup und putdown betreffen einen Block, die Aktionen stack und unstack betreffen zwei Blöcke. Würde man keine Variable für die Objekte verwenden, müssten bei einer Blockwelt mit drei Objekten insgesamt 24 Aktionen definiert werden. Bei fünf Blöcken wären es schon 60 Aktionen. Da sich die Aktionen jedoch nur in den Objekten unterscheiden, genügt es diese genannten vier Aktionen mit Hilfe von Variablen zu definieren.

Die Vorbedingungen und Effekte unterscheiden sich in der Darstellung der Verknüpfung. Während Vorbedingungen bgisch unterschiedlich miteinander verknüpft sein können, sind die Effekte immer miteinander „und“-verknüpft.

Vorbedingungen

Für die Implementierung werden die Vorbedingungen schon als boolesche Formeln abgebildet. Dabei gibt es in der Formel immer einen linken und einen rechten Nachfolger, sofern die Anzahl der Propositionen in den Vorbedingungen größer eins ist. Dadurch entsteht in der Formel automatisch eine Klammerung. Im folgenden Listing ist ein Beispiel der Vorbedingungen für die Aktion „unstack“ aus der Blockwelt dargestellt.

```
<precondition>
      <and>
            <proposition>
                  <name>on</name>
                  <parameter>x-block</parameter>
                  <parameter>y-block</parameter>
            </proposition>
            <and>
                  <proposition>
                        <name>handempty</name>
                  </proposition>
                  <proposition>
                        <name>clear</name>
                        <parameter>x-block</parameter>
                  </proposition>
            </and>
      </and>
</precondition>
```

Listing 5: Vorbedingungen für „unstack“ aus der Blockwelt-Domänen-Datei

Durch das hier dargestellte Listing entsteht eine boolesche Formel mit folgender Gleichung:

$$Y = on\_x - block\_y - block \wedge (handempty \wedge clear\_x - block)$$

In diesem Beispiel wäre die Klammerung nicht notwendig. Man könnte die Propositionen alle auf einer Ebene darstellen. Jedoch wird durch diese hierarchische Struktur eine einfachere Darstellung der Formel im Programmcode erreicht. Die Formel ist somit weniger fehleranfällig und besser kontrollierbar. Im folgenden Bild ist die Formel der Vorbedingungen graphisch dargestellt.

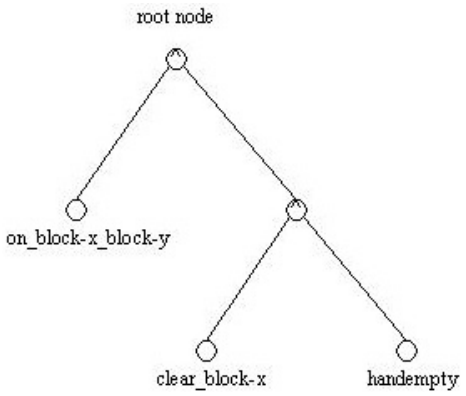

Abbildung 13: Darstellung der propositionalen Formel

Generiert wird die Formel durch Auslesen der XML-Repräsentation. Wenn nur eine Proposition für Vorbedingungen existiert, muss keine Formel generiert werden. Bei mehr als einer Proposition wird die Formel zunächst mit einem sogenannten „root“-Knoten repräsentiert, der ein logisches „und“, oder ein logisches „oder“ sein kann. Repräsentiert wird dieser Knoten durch den Namen. Der Knoten besitzt einen linken und einen rechten Nachfolger, die entweder Zweige des Knotens, oder Blätter sein können. Ein Blatt entspricht einer Proposition. Der Zweig eines Knotens hat wieder weitere Zweige oder Blätter, während ein

Blatt keine weiteren Zweige mehr besitzt. Zweige werden als Objekt der Klasse „InnerPropositionalFormula“ generiert, die Blätter durch Objekte der Klasse „Proposition“ dargestellt. Die Propositionen selbst können negiert sein, wobei die Negation durch ein einfaches Flag dargestellt wird. Mitgespeichert werden die Nummer der Proposition, welche dem Propositionstyp entspricht und der Index, der der Proposition selbst entspricht. Unter Propositionstyp sind die Propositionen zu verstehen, die denselben Namen haben. Im Blockweltbeispiel wären alle „on“-Propositionen von einem Typ, alle „ontable“-Propositionen von einem Typ, usw. Die eigentliche Proposition ist in Bezug auf die Objekte zu sehen, welche die Variable ersetzen.

Effekte

Weil alle Effekte bei der Ausführung einer Aktion die Umgebung beeinflussen, also logisch alle Propositionen mit „und“ verknüpft sind, werden diese in der XML-Beschreibung als Liste dargestellt. Die Funktionalität der propositionalen Formel ist wegen der Vorbedingungen schon vorhanden. Deshalb werden die Effekte intern ebenfalls als propositionale Formeln abgebildet. Aus der vorhandenen Liste wird eine Formel generiert, bei der die einzelnen Propositionen alle mit einem logischen „und“ verknüpft sind. In der XML-Beschreibung sieht die Darstellung der Effekte wie folgt aus:

```
<effect>
      <proposition negated="true">
            <name>clear</name>
            <parameter>x-block</parameter>
      </proposition>
      <proposition>
            <name>clear</name>
            <parameter>y-block</parameter>
      </proposition>
      <proposition negated="true">
            <name>on</name>
            <parameter>x-block</parameter>
            <parameter>y-block</parameter>
      </proposition>
      <proposition negated="true">
            <name>handempty</name>
      </proposition>
      <proposition>
            <name>holding</name>
            <parameter>x-block</parameter>
      </proposition>
</effect>
```

Listing 6: Effekte für „unstack“ aus der Blockwelt-Domänen-Datei

Im folgenden Listing wird die Aktion „pick-up“ beschrieben. Die Beschreibung besteht aus dem Namen der Aktion, den benötigten Parametern, den Vorbedingungen, um die Aktion ausführen zu können und den Effekten, die bei der Ausführung der Aktion eintreten.

```
<action>
      <name>pick-up</name>
      <parameter>x-block</parameter>
      <precondition>
            <and>
                  <proposition>
                        <name>clear</name>
                        <parameter>x-block</parameter>
                  </proposition>
                  <and>
                        <proposition>
                              <name>ontable</name>
                              <parameter>x-block</parameter>
                        </proposition>
                        <proposition>
                              <name>handempty</name>
                        </proposition>
                  </and>
            </and>
      </precondition>
      <effect>
            <proposition negated="true">
                  <name>clear</name>
                  <parameter>x-block</parameter>
            </proposition>
            <proposition negated="true">
                  <name>ontable</name>
                  <parameter>x-block</parameter>
            </proposition>
            <proposition negated="true">
                  <name>handempty</name>
            </proposition>
            <proposition>
                  <name>holding</name>
                  <parameter>x-block</parameter>
            </proposition>
      </effect>
</action>
```

Listing 7: Auszug aus einer Domänen-Datei der Blockwelt

Ausführung

Sind die Aktionen mit all ihren Vorbedingungen und Effekten generiert, können sie ausgeführt werden, sofern es die Vorbedingungen zulassen. Es ist jedoch notwendig, die Zustände vor dem Ausführen zu speichern, da eventuell Rückschritte notwendig werden, welche eine vorherige Speicherung der Zustände notwendig machen. Zum Speichern der Zustände gibt es eine Funktion, welche eine Liste der aktuellen Zustände zurück liefert. Um die Änderungen rückgängig zu machen, gibt es ebenfalls eine Funktion, welche eine Liste vorher gespeicherter Zustände erwartet. Zum Ausführen der Aktion wird die propositionale Formel, welche für die Effekte generiert wurde, automatisch rekursiv durchlaufen und der Weltzustand an die Effekte entsprechend angepasst.

Die auszuführende Aktion wird vom Planer ausgesucht, die aktuellen Weltzustände werden gespeichert und anhand der Effekte der Aktion der Weltzustand geändert. Dies jedoch alles auf virtueller Basis.

5.3 Ziel

Die Repräsentation des Ziels geschieht auf dieselbe Weise, wie die Repräsentation des Weltzustandes. Das Ziel kann sich jedoch in der Anzahl der relevanten Objekte zur Anzahl, der sich in der Welt befindlichen Objekte unterscheiden. Ein Ziel ist ein definierter Endzustand, wodurch keinerlei Variable benötigt werden, es ist lediglich eine Teilmenge aller möglichen Weltzustände. So kann ein Ziel, am Beispiel der Blockwelt, definiert sein als

- ontable_block-b
- on_block-a_block-b
- clear_block-a
- handempty,

obwohl vielleicht noch ein Block C existiert.

5.4 Planen / Suchen

Das Planen umfasst die gesamte Darstellung der Umgebung und der Aktionen, inklusive der Suche nach dem Weg zum Ziel. Beim Planen kommt es darauf an, einen geeigneten Suchalgorithmus auszuwählen und die nötigen Schritte zu realisieren, um Weltzustände zu speichern und gegebenenfalls wieder herzustellen.

Die Darstellung des Weltzustandes und die Repräsentation der Aktionen und des zu erreichenden Ziels war lediglich die Vorarbeit, jedoch bedeutender Bestandteil, für das Suchen nach dem Plan. Zum Planen gehört außer der Suche nach dem Ziel noch die Speicherung des Weges zum Ziel und der einzelnen Weltzustände um Aktionen wieder rückgängig machen zu können.

5.4.1 Überprüfung der Zielerreichung

Bevor mit dem Planen begonnen wird, wird überprüft, ob der Startzustand schon dem Ziel entspricht. Zur Überprüfung, ob das Ziel erreicht ist, wird der aktuelle Weltzustand mit dem generierten Zielzustand verglichen. Wenn alle angegebenen Propositionen, die für das Ziel definiert sind, mit den im Weltzustand befindlichen Propositionen übereinstimmen, ist das Ziel erreicht und somit der Weg zum Ziel gefunden. Das Vergleichen der Propositionen geschieht über einfachen Vergleich der Zustände. So wird jede Proposition einzeln verglichen, bis eine Proposition die Bedingung nicht erfüllt, oder alle nötigen Zustände übereinstimmen.

5.4.2 Identifizierung ausführbarer Aktionen

Ist das Ziel zu Beginn nicht erreicht, müssen die ausführbaren Aktionen identifiziert werden. Dies geschieht über den Vergleich des aktuellen Weltzustandes mit den Vorbedingungen der Aktionen. Stimmt der Weltzustand mit der Vorbedingung einer Aktion überein, also erfüllt der Weltzustand die boolesche Gleichung der Vorbedingung einer Aktion, so ist diese Aktion ausführbar. Daraus ergibt sich eine Liste mit ausführbaren Aktionen, welche auch leer sein kann. Wenn die Liste der ausführbaren Aktionen leer ist, das Ziel jedoch noch nicht erreicht ist, gibt es für den aktuellen Zweig im Suchbaum keinen Weg zum Ziel. Gibt es ausführbare Aktionen, muss der Planer eine Aktion zur Ausführung auswählen und diese dann in Gedanken ausführen.

Der Planer kennt die Effekte dieser Aktion, kann somit den Weltzustand ändern, ohne die Aktion real auszuführen. Anschließend wird der geänderte Weltzustand wieder auf die Erreichung des Ziels überprüft und gegebenenfalls ausgehend vom neuen Weltzustand wieder eine Aktion zur Ausführung ausgesucht. Dies wiederholt sich so lange, bis das Ziel gefunden ist, bzw. erkannt wird, dass kein Weg zum Ziel existiert.

5.4.3 Speicherung des Weltzustandes

Vor der Ausführung von Aktionen muss für eine spätere Wiederherstellung der alten Zustände der Weltzustand gespeichert werden. Dies geschieht durch Auslesen des aktuellen Weltzustandes und Speicherung in einem neuen Objekt. Es wird jedoch aus Effizienzgründen nicht der gesamte Weltzustand gespeichert, sondern nur die von der Aktion betroffenen Propositionen. Das bedeutet, dass die Propositionen der Effekte einer Aktion gespeichert werden, da nur diese Propositionen im Weltzustand verändert werden. Das Anlegen eines neuen Objektes ist erforderlich, da es bei Java lediglich Referenzen gibt und die Änderung eines Objektes auch die Änderung des gespeicherten Objektes zu Folge hätte, wenn nur die Referenz gespeichert werden würde.

Durch den Aufbau der propositionalen Formel gestaltet sich die Speicherung recht einfach. Dabei wird die Speicherung des linken und des rechten Nachfolgers in eine Liste eingetragen, also jeweils wieder die Speicherfunktion des linken und des rechten Zweiges aufgerufen. Dadurch wird ein rekursives Durchlaufen der Formel ausgeführt, bis jeweils das Blatt der Formel erreicht ist. An dieser Stelle wird dann der tatsächliche propositionale Wert der Proposition ausgelesen und in ein neues Boolean-Objekt gespeichert.

5.4.4 Ausführen der Aktion

Nun sind genau die Werte gespeichert, welche von der Aktion verändert werden und die Aktion kann ausgeführt werden. Die Ausführung geschieht auf genau dieselbe Weise, wie die Speicherung. Die propositionale Formel wird ebenfalls rekursiv durchlaufen, indem an jeder Stelle vom Nachfolger wieder die Ausführfunktion aufgerufen wird, bis man am Blatt, also an der Proposition angelangt ist. Diese wird dann in ihrem Wert entsprechend abgeändert.

5.4.5 Wiederherstellung des Weltzustandes

Das Wiederherstellen eines alten Zustandes geschieht auf dieselbe Weise, wie die Speicherung, nur dass die Werte jetzt nicht in einer Liste gespeichert werden, sondern die zuvor gespeicherten Werte in einer Liste übergeben werden. Die Formel wird wie beschrieben wieder rekursiv durchlaufen und die gespeicherten Werte entsprechend zurückgesetzt.

5.4.6 Suchen mit Tiefensuche

Die Wahl der Reihenfolge der auszuführenden Aktionen wird durch den verwendeten Suchalgorithmus bestimmt. Exemplarisch wird dies am Beispiel der Tiefensuche erläutert. Bei der Tiefensuche wird eine Methode aufgerufen, die sich dann selbst wieder aufruft, bis der Plan gefunden ist, bzw. feststeht, dass es keinen Weg zum Ziel gibt.

Innerhalb dieser Methode werden ausgehend vom aktuellen Weltzustand alle ausführbaren Aktionen identifiziert. Da es sich um eine Liste handelt, welche auch keine oder nur eine ausführbare Aktion beinhalten kann, wird aus dieser Liste zunächst die erste Aktion zur Ausführung ausgesucht. An dieser Stelle wird überprüft, ob das Ziel bereits erreicht ist. Wenn das Ziel noch nicht erreicht ist, werden die Propositionen des Effektes der ausgewählten Aktion gespeichert und anschließend die Aktion im Plan gespeichert und ausgeführt. Diese Methode wird jetzt wieder aufgerufen und das Spiel beginnt von neuem. Ist der Planer an eine Stelle gelangt, an der es keine ausführbare Aktion mehr gibt, so wird die letzte ausgeführte Aktion wieder aus dem Plan entfernt, die Änderungen dieser Aktion rückgängig gemacht und aus der Liste der ausführbaren Aktionen des vorigen Suchknotens die nächste Aktion ausgewählt. Damit dies funktioniert muss noch eine Tiefenbeschränkung vorgenommen werden. In der Blockwelt kann beispielsweise die erste ausführbare Aktion in der Liste die „pickup“-Aktion sein. Wenn aus dem geänderten Weltzustand die neue Liste ausführbarer Aktionen als erste Aktion „putdown“ ist, kann sich die vorige Aktion wieder aufheben. Dadurch kann eine Endlosschleife entstehen, welche durch eine Tiefenbeschränkung abgebrochen werden kann.

Wird das Ziel gefunden, befindet sich in der Liste gespeicherter Aktionen der Plan, also der Weg zum Ziel. Diese Aktionen können dann sequentiell ausgeführt werden, um das Ziel zu erreichen.

5.5 Empirische Untersuchungen

In diesem Abschnitt sind die Ergebnisse einiger Messprotokolle dargestellt. Die Messprotokolle protokollieren den Zeit- und Speicherbedarf, einer Blockwelt mit unterschiedlicher Anzahl von Blöcken und unterschiedlichen Startzuständen. Die Intention liegt nicht im Vergleich unterschiedlicher Technologien oder Suchalgorithmen der Handlungsplanung, da diese bereits weitestgehend auf Grundlage der Komplexitätstheorien untersucht wurden. Vielmehr sollen die Messprotokolle den Zusammenhang zwischen Anzahl der Objekte und Zeitbedarf beim Planen aufzeigen. Gemessen wurde der gesamte Zeitbedarf vom Anlegen der Propositionen und Aktionen, bis hin zum gefundenen Plan.

Weil es bei den Messprotokollen nicht um unterschiedlich implementierte Technologien und Suchalgorithmen geht, wurde für die Messprotokolle immer die iterative Tiefensuche verwendet. Um Einflüsse der Messung möglichst vernachlässigen zu können, wurden die Messergebnisse zunächst zwischengespeichert und erst am Ende der Messung ausgegeben. Natürlich ist die Messung mit wachsender Anzahl von Objekten zeitaufwändiger, wodurch eine solche Beeinflussung nicht total vernachlässigt werden darf. Es darf auch nicht vergessen werden, dass unterschiedliche Umgebungen in der Planung unterschiedliche Auswirkungen auf Laufzeitverhalten und Speicherbedarf haben. Um auch Einflüsse des LARS und der Logic Engine zu vermeiden, da aufgrund des „Threading"-Verhaltens der Virtuellen Maschine von Java die Prozesszeit von Messung zu Messung anders zugeordnet sein kann, wird für die Messungen der vorhandene, nicht ins LARS integrierte, Prototyp verwendet.

Die Messungen erfolgten auf einem System mit einem AMD1700XP Prozessor und 1 GB PC266 Arbeitsspeicher. Beim Start wurde der minimale Speicher für die virtuelle Maschine von Java auf 256 MB und der maximale Speicher auf 1 GB festgelegt.

Im Anhang sind die Messprotokolle im Einzelnen dargestellt.

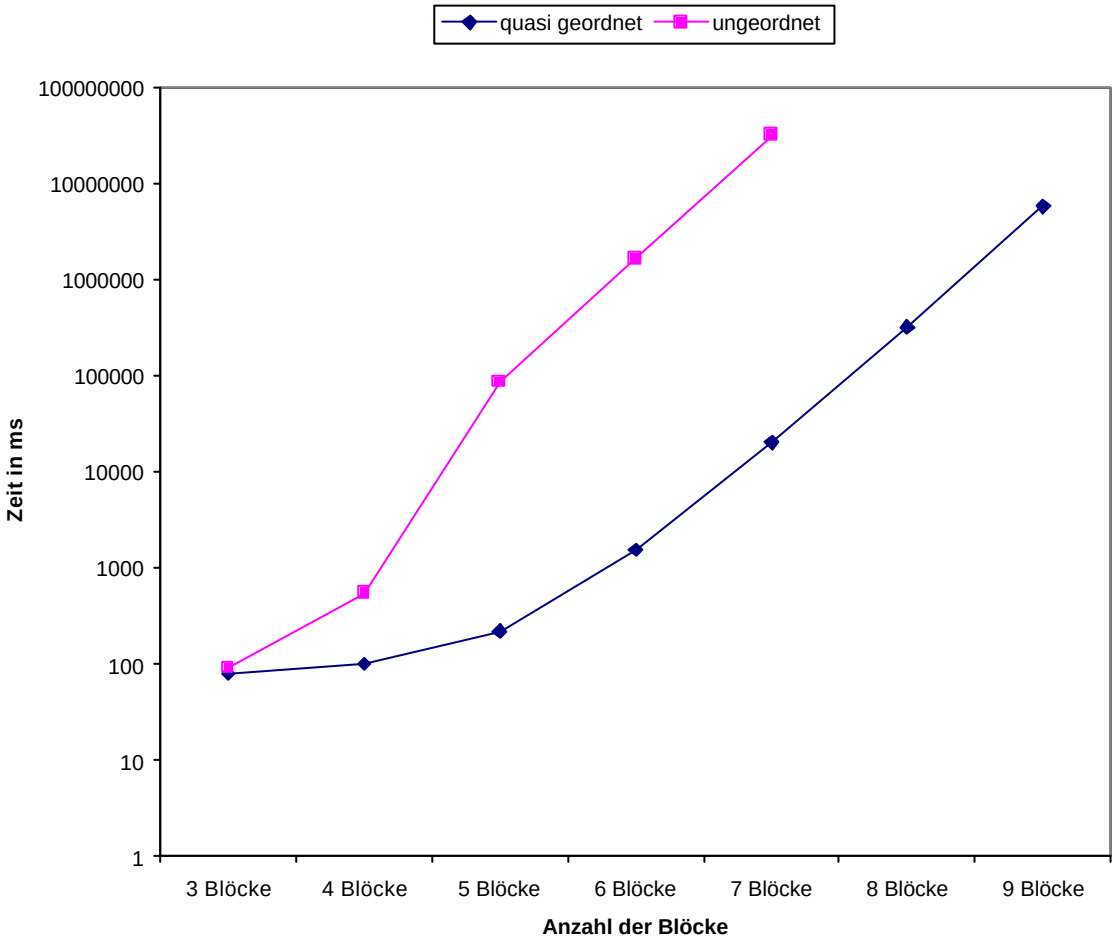

Abbildung 14: Messungen für Zeitbedarf beim Planen in logarithmischer Darstellung

Wie in der Grafik zu sehen ist, steigt der Zeitbedarf in beiden Fällen exponentiell an. Der Unterschied zwischen „quasi“ geordneten Startzuständen und ungeordneten Startzuständen liegt im Beginn des exponentiellen Anstiegs. Umgebungen mit ungeordneten Startzuständen steigen mit wenigen Blöcken stärker an. Ab ca. fünf bis sechs Blöcken verläuft die Steigerungsrate nahezu parallel, wodurch der Unterschied zwischen beiden Umgebungen mit zunehmender Anzahl an Blöcken komplexitätsbezogen vernachlässigbar ist.

Der Grund, warum ungeordnete Startzustände deutlich mehr Zeit benötigen, liegt in zwei Punkten begründet. Die Tiefensuche besitzt kein besonderes Auswahlverfahren, was die Wahl der zuerst auszuführenden Aktion betrifft. Somit werden bei den geordneten Startzuständen zufällig zuerst die geeigneteren Aktionen ausgeführt. Da die Aktionen in einer XML-Beschreibungsdatei definiert und als Sammlung gespeichert sind, haben die Aktionen in der Liste der auszuführenden Aktionen immer dieselbe Reihenfolge. Der zweite und wohl gewichtigere Grund liegt in der Anzahl der auszuführenden Aktionen. Wie den Messprotokollen zu entnehmen ist, benötigt die Blockwelt mit sieben Blöcken im Falle des geordneten Startzustandes 14 Aktionen, um zum Ziel zu gelangen, und im Falle des ungeordneten Startzustandes 20 Aktionen. Diese sechs zusätzlichen Aktionen vergrößern den Suchbaum in erheblichem Maße, was einen höheren Zeitbedarf zur Folge hat.

Bedeutende Erkenntnis der Messungen ist, dass schon nach kurzer Zeit der Zeitbedarf so stark ansteigt, dass ein Planen kaum noch sinnvoll ist. Daher müssen andere Technologien und Suchalgorithmen verwendet werden, um in angemessener Zeit ein Ergebnis zu finden.

Kapitel 6

Fazit

Die Handlungsplanung ist eine gute Möglichkeit, um in bekannten Umgebungen den idealen Weg zu einem vorgegebenen Ziel zu finden. Innerhalb der Handlungskontrolle von Softwareagenten ist sie die einzige Möglichkeit, ein optimales und vollständiges Ergebnis zu erzielen, jedoch auch die kostspieligste in Bezug auf Speicher- und Zeitbedarf.

Die Nutzung der Handlungsplanung eröffnet vollkommen neue Möglichkeiten, wenn diese auch meist sehr komplex sein können und aufgrund der Komplexität als unlösbar erscheinen. Sie kann in Zukunft mit fortschreitender Forschung und immer schneller werdenden Prozessoren eine erhebliche Ersparnis in sämtlichen Bereichen der Planung erbringen, sei es im Bereich der Logistik oder im Bereich des Managements, wie beispielsweise der Projektplanung.

Voraussetzung, um ein Planungssystem einsetzen zu können, ist eine Umgebung, die man mit Propositionen darstellen kann. Im Gegensatz zu den meisten anderen Technologien der Handlungskontrolle ist die Handlungsplanung nur für Umgebungen geeignet, die sich durch externe Einflüsse nicht, oder nur sehr langsam ändern. Dafür erhält man ein genaues Ergebnis darüber, welche Aktionen durchgeführt werden müssen, um das gewünschte Ziel zu erreichen. Aufgrund der Komplexität, des hohen Zeit- und Speicherbedarfs zur Planung findet diese Technologie heute nur wenige Anwendungen im praktischen Einsatz. Wenn Planung eingesetzt wird, ist diese meist auf spezielle Gebiete optimiert, wie beispielsweise auf die Routenplanung. Das hat speziell bei der Routenplanung zur Folge, dass nicht immer das optimale Ergebnis erreicht wird und das Planungssystem nur für diese Umgebung eingesetzt werden kann. Man darf gespannt sein, was sich in diesem Forschungsgebiet in den nächsten Jahren noch alles tut. An der Universität in Freiburg und anderen Hochschulen wurde während der letzten Jahre und wird aller Voraussicht nach noch einige Jahre in diesem Gebiet intensiv geforscht werden.

Anhang A. Tabellenverzeichnis

Anhang B. Abbildungsverzeichnis

Anhang C. Listing-Verzeichnis

Anhang D. Messprotokolle

Messung 1

Startzustand:

- ontable_block-a
- on_block-b_block-a
- on_block-c_block-b
- clear_block-c
- handempty

Zielzustand:

- ontable_block-c
- on_block-b_block-c
- on_block-a_block-b
- clear_block-a
- handempty

```
free memory at startup = 255 Mbyte
total memory at startup = 255 Mbyte
used memory at startup = 536 Kbyte
used memory after worldstate generation = 615 Kbyte
used memory after action generation = 720 Kbyte
used memory after goal generation = 731 Kbyte

max used memory during search:
196 count of executions with max 909 Kbyte of used memory.

received plan:
unstack_block_c_block_b
put-down_block_c
unstack_block_b_block_a
stack_block_b_block_c
pick-up_block_a
stack_block_a_block_b

ready in '80' ms
```

Messung 1: 3 Blöcke quasi geordnet

Messung 2

Startzustand:

- ontable_block-a
- on_block-b_block-a
- on_block-c_block-b
- on_block-d_block-c
- clear_block-d
- handempty

Zielzustand:

- ontable_block-d
- on_block-c_block-d
- on_block-b_block-c
- on_block-a_block-b
- clear_block-a
- handempty

```
free memory at startup = 255 Mbyte
total memory at startup = 255 Mbyte
used memory at startup = 541 Kbyte
used memory after worldstate generation = 621 Kbyte
used memory after action generation = 752 Kbyte
used memory after goal generation = 763 Kbyte

max used memory during search:
1302 count of executions with max 1 Mbyte of used memory.

received plan:
unstack_block_d_block_c
put-down_block_d
unstack_block_c_block_b
stack_block_c_block_d
unstack_block_b_block_a
stack_block_b_block_c
pick-up_block_a
stack_block_a_block_b

ready in '100' ms
```

Messung 2: 4 Blöcke quasi geordnet

Messung 3

Startzustand:

- ontable_block-a
- on_block-b_block-a
- on_block-c_block-b
- on_block-d_block-c
- on_block-e_block-d
- clear_block-e
- handempty

Zielzustand:

- ontable_block-e
- on_block-d_block-e
- on_block-c_block-d
- on_block-b_block-c
- on_block-a_block-b
- clear_block-a
- handempty

```
free memory at startup = 255 Mbyte
total memory at startup = 255 Mbyte
used memory at startup = 546 Kbyte
used memory after worldstate generation = 626 Kbyte
used memory after action generation = 790 Kbyte
used memory after goal generation = 801 Kbyte
max used memory during search:
10972 count of executions with max 2 Mbyte of used memory.

received plan:
unstack_block_e_block_d
put-down_block_e
unstack_block_d_block_c
stack_block_d_block_e
unstack_block_c_block_b
stack_block_c_block_d
unstack_block_b_block_a
stack_block_b_block_c
pick-up_block_a
stack_block_a_block_b

ready in '220' ms
```

Messung 3: 5 Blöcke quasi geordnet

Messung 4

Startzustand:

- ontable_block-a
- on_block-b_block-a
- on_block-c_block-b
- on_block-d_block-c
- on_block-e_block-d
- on_block-f_block-e
- clear_block-f
- handempty

Zielzustand:

- ontable_block-f
- on_block-e_block-f
- on_block-d_block-e
- on_block-c_block-d
- on_block-b_block-c
- on_block-a_block-b
- clear_block-a
- handempty

```
free memory at startup = 255 Mbyte
total memory at startup = 255 Mbyte
used memory at startup = 551 Kbyte
used memory after worldstate generation = 631 Kbyte
used memory after action generation = 835 Kbyte
used memory after goal generation = 847 Kbyte

max used memory during search:
113531 count of executions with max 2 Mbyte of used memory.

received plan:
unstack_block_f_block_e
put-down_block_f
unstack_block_e_block_d
stack_block_e_block_f
unstack_block_d_block_c
stack_block_d_block_e
unstack_block_c_block_b
stack_block_c_block_d
unstack_block_b_block_a
stack_block_b_block_c
pick-up_block_a
stack_block_a_block_b

ready in '1542' ms = 1,542 s
```

Messung 4: 6 Blöcke quasi geordnet

Messung 5

Startzustand:

- ontable_block-a
- on_block-b_block-a
- on_block-c_block-b
- on_block-d_block-c
- on_block-e_block-d
- on_block-f_block-e
- on_block-g_block-f
- clear_block-g
- handempty

Zielzustand:

- ontable_block-g
- on_block-f_block-g
- on_block-e_block-f
- on_block-d_block-e
- on_block-c_block-d
- on_block-b_block-c
- on_block-a_block-b
- clear_block-a
- handempty

```
free memory at startup = 255 Mbyte
total memory at startup = 255 Mbyte
used memory at startup = 556 Kbyte
used memory after worldstate generation = 636 Kbyte
used memory after action generation = 887 Kbyte
used memory after goal generation = 899 Kbyte

max used memory during search:
1396541 count of executions with max 2 Mbyte of used memory.

received plan:
unstack_block_g_block_f
put-down_block_g
unstack_block_f_block_e
stack_block_f_block_g
unstack_block_e_block_d
stack_block_e_block_f
unstack_block_d_block_c
stack_block_d_block_e
unstack_block_c_block_b
stack_block_c_block_d
unstack_block_b_block_a
stack_block_b_block_c
pick-up_block_a
stack_block_a_block_b

ready in '20109' ms = 20,109 s
```

Messung 5: 7 Blöcke quasi geordnet

Messung 6

Startzustand:

- ontable_block-a
- on_block-b_block-a
- on_block-c_block-b
- on_block-d_block-c
- on_block-e_block-d
- on_block-f_block-e
- on_block-g_block-f
- on_block-h_block-g
- clear_block-h
- handempty

Zielzustand:

- ontable_block-h
- on_block-g_block-h
- on_block-f_block-g
- on_block-e_block-f
- on_block-d_block-e
- on_block-c_block-d
- on_block-b_block-c
- on_block-a_block-b
- clear_block-a
- handempty

```
free memory at startup = 255 Mbyte
total memory at startup = 255 Mbyte
used memory at startup = 561 Kbyte
used memory after worldstate generation = 642 Kbyte
used memory after action generation = 946 Kbyte
used memory after goal generation = 958 Kbyte

max used memory during search:
19961355 count of executions with max 2 Mbyte of used memory.

received plan:
unstack_block_h_block_g
put-down_block_h
unstack_block_g_block_f
stack_block_g_block_h
unstack_block_f_block_e
stack_block_f_block_g
unstack_block_e_block_d
stack_block_e_block_f
unstack_block_d_block_c
stack_block_d_block_e
unstack_block_c_block_b
stack_block_c_block_d
unstack_block_b_block_a
stack_block_b_block_c
pick-up_block_a
stack_block_a_block_b

ready in '314893' ms = 5,248 min
```

Messung 6: 8 Blöcke quasi geordnet

Messung 7

Startzustand:

- ontable_block-a
- on_block-b_block-a
- on_block-c_block-b
- on_block-d_block-c
- on_block-e_block-d
- on_block-f_block-e
- on_block-g_block-f
- on_block-h_block-g
- on_block-i_block-h
- clear_block-i
- handempty

Zielzustand:

- ontable_block-i
- on_block-h_block-i
- on_block-g_block-h
- on_block-f_block-g
- on_block-e_block-f
- on_block-d_block-e
- on_block-c_block-d
- on_block-b_block-c
- on_block-a_block-b
- clear_block-a
- handempty

```
free memory at startup = 255 Mbyte
total memory at startup = 255 Mbyte
used memory at startup = 567 Kbyte
used memory after worldstate generation = 647 Kbyte
used memory after action generation = 1012 Kbyte
used memory after goal generation = 1 Mbyte

max used memory during search:
326052265 count of executions with max 3 Mbyte of used memory.

received plan:
unstack_block_i_block_h
put-down_block_i
unstack_block_h_block_g
stack_block_h_block_i
unstack_block_g_block_f
stack_block_g_block_h
unstack_block_f_block_e
stack_block_f_block_g
unstack_block_e_block_d
stack_block_e_block_f
unstack_block_d_block_c
stack_block_d_block_e
unstack_block_c_block_b
stack_block_c_block_d
unstack_block_b_block_a
stack_block_b_block_c
pick-up_block_a
stack_block_a_block_b

ready in '5674800' ms = 94,58 min
```

Messung 7: 9 Blöcke quasi geordnet

Messung 8

Startzustand:

- ontable_block-b
- on_block-c_block-b
- on_block-a_block-c
- clear_block-a
- handempty

Zielzustand:

- ontable_block-c
- on_block-b_block-c
- on_block-a_block-b
- clear_block-a
- handempty

```
free memory at startup = 255 Mbyte
total memory at startup = 255 Mbyte
used memory at startup = 536 Kbyte
used memory after worldstate generation = 615 Kbyte
used memory after action generation = 720 Kbyte
used memory after goal generation = 731 Kbyte

max used memory during search:
796 count of executions with max 1 Mbyte of used memory.

received plan:
unstack_block_a_block_c
put-down_block_a
unstack_block_c_block_b
put-down_block_c
pick-up_block_b
stack_block_b_block_c
pick-up_block_a
stack_block_a_block_b

ready in '90' ms
```

Messung 8: 3 Blöcke ungeordnet

Messung 9

Startzustand:

- ontable_block-b
- on_block-a_block-b
- on_block-d_block-a
- on_block-c_block-d
- clear_block-c
- handempty

Zielzustand:

- ontable_block-d
- on_block-c_block-d
- on_block-b_block-c
- on_block-a_block-b
- clear_block-a
- handempty

```
free memory at startup = 255 Mbyte
total memory at startup = 255 Mbyte
used memory at startup = 541 Kbyte
used memory after worldstate generation = 621 Kbyte
used memory after action generation = 752 Kbyte
used memory after goal generation = 763 Kbyte

max used memory during search:
43668 count of executions with max 2 Mbyte of used memory.

received plan:
unstack_block_c_block_d
put-down_block_c
unstack_block_d_block_a
put-down_block_d
pick-up_block_c
stack_block_c_block_d
unstack_block_a_block_b
put-down_block_a
pick-up_block_b
stack_block_b_block_c
pick-up_block_a
stack_block_a_block_b

ready in '541' ms
```

Messung 9: 4 Blöcke ungeordnet

Messung 10

Startzustand:

- ontable_block-b
- on_block-a_block-b
- on_block-e_block-a
- on_block-d_block-e
- on_block-c_block-d
- clear_block-c
- handempty

Zielzustand:

- ontable_block-e
- on_block-d_block-e
- on_block-c_block-d
- on_block-b_block-c
- on_block-a_block-b
- clear_block-a
- handempty

```
free memory at startup = 255 Mbyte
total memory at startup = 255 Mbyte
used memory at startup = 546 Kbyte
used memory after worldstate generation = 626 Kbyte
used memory after action generation = 790 Kbyte
used memory after goal generation = 801 Kbyte

max used memory during search:
7623868 count of executions with max 2 Mbyte of used memory.

received plan:
unstack_block_c_block_d
put-down_block_c
unstack_block_d_block_e
put-down_block_d
unstack_block_e_block_a
put-down_block_e
pick-up_block_d
stack_block_d_block_e
pick-up_block_c
stack_block_c_block_d
unstack_block_a_block_b
put-down_block_a
pick-up_block_b
stack_block_b_block_c
pick-up_block_a
stack_block_a_block_b

ready in '86044' ms = 1,26 min
```

Messung 10: 5 Blöcke ungeordnet

Messung 11

Startzustand:

- ontable_block-b
- on_block-a_block-b
- on_block-c_block-a
- on_block-f_block-c
- on_block-e_block-f
- on_block-d_block-e
- clear_block-d
- handempty

Zielzustand:

- ontable_block-f
- on_block-e_block-f
- on_block-d_block-e
- on_block-c_block-d
- on_block-b_block-c
- on_block-a_block-b
- clear_block-a
- handempty

```
free memory at startup = 255 Mbyte
total memory at startup = 255 Mbyte
used memory at startup = 551 Kbyte
used memory after worldstate generation = 631 Kbyte
used memory after action generation = 835 Kbyte
used memory after goal generation = 847 Kbyte

max used memory during search:
132129628 count of executions with max 2 Mbyte of used memory.

received plan:
unstack_block_d_block_e
put-down_block_d
unstack_block_e_block_f
put-down_block_e
unstack_block_f_block_c
put-down_block_f
pick-up_block_e
stack_block_e_block_f
pick-up_block_d
stack_block_d_block_e
unstack_block_c_block_a
stack_block_c_block_d
unstack_block_a_block_b
put-down_block_a
pick-up_block_b
stack_block_b_block_c
pick-up_block_a
stack_block_a_block_b

ready in '1637825' ms = 27,297 min
```

Messung 11: 6 Blöcke ungeordnet

Messung 12

Startzustand:

- ontable_block-b
- on_block-a_block-b
- on_block-c_block-a
- on_block-d_block-c
- on_block-g_block-d
- on_block-f_block-g
- on_block-e_block-f
- clear_block-e
- handempty

Zielzustand:

- ontable_block-g
- on_block-f_block-g
- on_block-e_block-f
- on_block-d_block-e
- on_block-c_block-d
- on_block-b_block-c
- on_block-a_block-b
- clear_block-a
- handempty

```
free memory at startup = 255 Mbyte
total memory at startup = 255 Mbyte
used memory at startup = 556 Kbyte
used memory after worldstate generation = 636 Kbyte
used memory after action generation = 887 Kbyte
used memory after goal generation = 899 Kbyte

max used memory during search:
-1869774806 count of executions with max 8 Mbyte of used memory.
(Datentyp int war zu klein)

received plan:
unstack_block_e_block_f
put-down_block_e
unstack_block_f_block_g
put-down_block_f
unstack_block_g_block_d
put-down_block_g
pick-up_block_f
stack_block_f_block_g
pick-up_block_e
stack_block_e_block_f
unstack_block_d_block_c
stack_block_d_block_e
unstack_block_c_block_a
stack_block_c_block_d
unstack_block_a_block_b
put-down_block_a
pick-up_block_b
stack_block_b_block_c
pick-up_block_a
stack_block_a_block_b

ready in '31758887' ms = 8 h 49 min 19 s
```

Messung 12: 7 Blöcke ungeordnet

Anhang E. Literaturverzeichnis

[Bigus und Bigus, 2001]
BIGUS, Joseph und BIGUS, Jennifer: *Intelligente Agenten mit Java programmieren – eCommerce und Informationsrecherche automatisieren.* München: Addison-Wesley, 2001.

[Blum und Furst, 1995]
BLUM, A. L. und FURST, M. L.: Fast Planning through Planning Graph Analysis. *Artificial Intelligence*, Vol. 90 (1-2), S. 281-300, 1997.

[Dilger, 2001]
DILGER, Werner: *Einführung in die künstliche Intelligenz*. Technische Universität Chemnitz: Vorlesungsskript, 2001
http://www.tu-chemnitz.de/informatik/HomePages/KI/scripts.

[Dannegger und Dorer, 2001]
DANNEGGER, C und DORER, K..: Agenten in aller Munde: Grundlagen und Hintergründe. *Java Magazin*, Ausgabe 7, 2001.

[Hertzberg, 1989]
HERTZBERG, Joachim: *Planen – Einführung in die Planerstellungsmethoden der Künstlichen Intelligenz.* Mannheim: BI-Wissenschaftlicher-Verlag, 1989.

[Jennings and Wooldridge, 2001]
JENNINGS, N. R. and WOOLDRIDGE, M.: Agent-Oriented Software Engineering. *Handbook of Agent Technology*, AAAI/MIT Press, 2001.

[Kone, Shimazu und Nakajima, 2000]
KONE, M. T., SHIMAZU, A. und NAKAJIMA, T.: *Knowledge and Information Systems 2.* The State of the Art in Agent Communication Languages, S. 259-284, 2000.

[Kushmerick, 1993]
KUSHMERICK, N.: An Algorithm for Probabilistic Planning. *Technical Report, University of Washington Department of Computer Science and Engineering*, Number 93-06-03, 1993.

[Kushmerick, 1994]
KUSHMERICK, N.: An Algorithm for Probabilistic Least-Commitment Planning. *Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI-94)*, Vol. 2, S. 1073-1078, AAAI Press/MIT Press, 1994.

[Russell und Norvig, 1995]
RUSSEL, Stuart und NORVIG, Peter: *Artificial Intelligence – A Modern Approach*. New Jersey: Prentice-Hall Inc., 1995.

[Sacerdoti, 1973]
SACERDOTI, E. D.: Planning in a hierarchy of abstraction spaces. *Advance papers of IJCAI-73*, 1973.

[Tate, 1991]
TATE, A.: O-Plan: the open planning architecture. *Artificial Intelligence*, 52 (1) S. 49-86, 1991.

[Weiß, 2001]
WEIß, G.: Agentenorientiertes Software Engineering. *Informatik Spektrum*, Band 24, Heft 2, Heidelberg: Springer-Verlag, S. 98-101, 2001.

[Whitley und Starkweather, 1989]
WHITLEY, D., STARKWEATHER, T. and FUQUAY, D'Ann: Scheduling Problems and Travelling Salesmen: The Genetic Edge Recombination Operator. *Proceedings of the 3rd International Conference on Genetic Algorithms*, S. 133-140, 1989.