Fachhochschule Hamburg
Fachbereich Elektrotechnik und Informatik

von Jan Schellenberger und
Heiko Holst

Studiengang Softwaretechnik

Studienarbeit

Planungsalgorithmen

Dezember 1996

Inhalt

1. Einleitung

An der Fachhochschule Hamburg wurde im Herbst 1995 das Projekt ICS (Integrating Cognitive Systems) ins Leben gerufen. Ziel dieses Projektes ist, daß ein Roboter sich in einer bestimmten Domäne (Arbeitswelt) bewegt und vorbestimmte Aufgaben erfüllen kann. Zur Steuerung besitzt der Roboter Sensoren und Aktoren, die es ihm ermöglichen, die Aufgaben in der Domäne zu erfüllen.

Die Reihenfolge der vom Roboter auszuführenden Aktionen soll durch ein Planungsprogramm bestimmt werden. Die Domäne des Robotors ist vergleichbar mit der Klötzchenwelt, die eine typische Domäne für Planungsprogramme darstellt.

Im Folgenden werden nichtlineare Planungsprogramme betrachtet, um ein neues, nichtlineares Planungsprogramm zu erstellen. Anschließend werden die Funktionsweisen von linearen und nichtlinearen Planungsprogrammen dargestellt und auf einige bestimmte, nichtlineare Planungsprogramme näher eingegangen.

Bevor jedoch das Planungsproblem angegangen wurde, mußte die Domäne der Klötzchenwelt ausreichend erläutert werden.

Hieraus wurden bestimmte Schlußfolgerungen gezogen, die zur Erstellung des Planungsprogrammes PLATZ führten, welches in PROLOG implementiert wurde. Die Besonderheiten dieses Planungsprogrammes werden eingehend erläutert. Der Programmcode ist im Anhang aufgeführt.

2. Lösungsalgorithmen

2.1 Allgemeine Funktionsarten von Lösungsalgorithmen

Die Aufgabe eines Lösungsalgorithmus ist es, nach Bestimmung von Domäne, Ausgangs- und Zielzustand, einen Lösungsplan zu erstellen. Der Lösungsplan (LP) besteht aus einer Liste von Aktionen. Wird dieser Lösungsplan auf den Ausgangszustand angewandt, so ist der Zielzustand erreicht. Während der Ausführung des Lösungsplanes können beliebig viele Zwischenzustände (ZS) erreicht werden.

2.1.1 Lineare Lösungsalgorithmen

Ein linearer Lösungsalgorithmus überführt ein Objekte aus einem Ausgangszustande mittels einer Aktion in einen nachfolgenden Zustand. In dem neuen Zustand wird <u>irgendein</u> Objekt (also das gleiche oder ein anderes Objekt) durch eine Aktion beeinflußt. Es entsteht ein weiterer Zustand.
Diese Zustandsüberführung setzt sich solange fort, bis der gewünschte Zielzustand erreicht wurde.

Verbesserte Lösungsalgorithmen können Kombination von Aktionen finden, die zur Lösung des Planungsproblemes beitragen, einschließlich der nichtlinearen Aktionsfolgen. Hierbei besteht der Lösungsplan aus mehreren Plänen.

2.1.2 Nichtlineare Lösungsalgorithmen

Die nichtlinearen Lösungsalgorithmen suchen Aktionen, die die Eigenschaften <u>eines</u> Objektes aus dem Ausgangszustand so verändern, daß sie den Eigenschaften des gleichen Objektes im Zielzustand entsprechen. Erst wenn eine Regeln keine Aktion erlaubt, weil ein anderes Objekt die Eigenschaft festlegt, wird das ‘störende’ Objekt betrachtet. Die Eigenschaft des Objektes wird so verändert, daß die geforderte Eigenschaften des ersten Objektes erreicht werden können (d.h. Regeln erlauben jetzt die notwendige Aktion). Aus den Regeln, die zum Erfüllen der Eigenschaften notwendig sind, lassen sich bestimmte Teilpläne erstellen. Teilpläne beinhalten eine bestimmte Aktionsreihenfolge, die einzuhalten ist. Die Reihnenfolge der Ausführung von Teilplänen selber ist nur durch Ausschlußkriterien der angewendeten Regeln zueinander zu erkennen.

2.2 Bestimmte nichtlineare Lösungsalgorithmen

2.2.1 NOAH (Nets of Action Hierarchies)

Das System besteht bei Noah [Sacerdoti75] aus einem prozeduralen Netz, welches aus Knoten mit Informationen und Zeigern auf andere Knoten besteht. Die Zeiger verweisen jeweils auf die Vorgänger und auf die Nachfolger eines Knotens. Am Anfang gibt es einen einzigen Knoten, indem der Zielzustand beschrieben wird.
Knoten können einen unterschiedlichen Typ haben:

- GOAL: Knoten, die ein Ziel repräsentieren
- PHANTOM: Knoten, die Ziele beschreiben, welche zum betreffenden Zeitpunkt bereits erreicht wurden
- SPLIT: Knoten dieses Typs besitzen nur einen Vorgängerknoten, aber mehrere Nachfolgerknoten
- JOIN: Knoten dieses Typs besitzen nur einen Nachfolgerknoten, aber mehrere Vorgängerknoten

Ferner besitzt jeder Knoten einen Codeteil, der bei Evaluierung die Aktionen des Knotens simuliert. Jeder Knoten wird bei NOAH in ein detaillierteres prozedurales Netz aufgespalten und es werden weitere Knoten produziert. Der Algorithmus, nachdem dieses geschieht, sieht wie folgt aus:

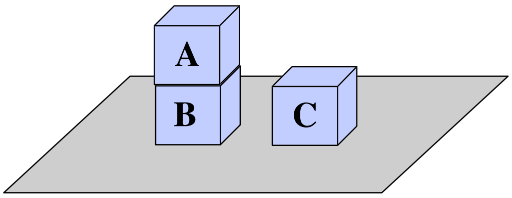
1. Der detaillierteste Plan wird simuliert und dadurch ein neuer noch detaillierter Plan erzeugt.
2. Der neue Plan wird bearbeitet, indem kritische Punkte bzw. redundante Operationen entfernt werden und eine Neustrukturierung stattfindet.
3. gehe zu 1.

Dieses wird solange fortgesetzt bis kein detaillierter Plan mehr gefunden werden konnte.
Probleme treten bei NOAH zum Beispiel dann auf, wenn bei zwei parallelen Teilzielen eine Aktion des einen Teilziels einen Ausdruck löscht, der Vorbedingung für das andere Teilziel war. Das System muß in so einem Fall das Löschen der Vorbedingung verhindern. Außerdem könnten bei mehreren parallelen Teilzielen auch dieselben Vorbedingungen vorhanden sein. Auch dieses muß vom System erkannt und gleiche Vorbedingungen durch eine einzige Vorbedingung ersetzt werden.
Diese und andere kritischen Punkte bzw. redundante Operationen werden im Punkt 2 des Algorithmus verhindert.

Ein Beispiel für NOAH:

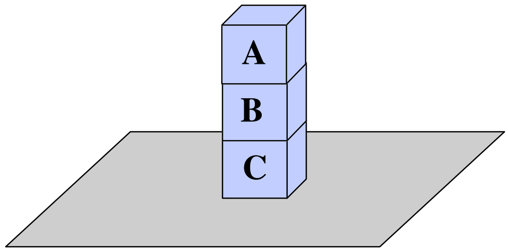
Das vorliegende System bedient sich der Klötzchenwelt. Jeder Goal-Knoten besitzt eine Liste von Unterfunktionen, welche in diesem Fall „frei“ oder „Stelle auf“ sein können. Die genauere Erklärung der Klötzchenwelt erfolgt in Kapitel 3.
Zunächst werden der Ausgangs- und der Zielzustand beschrieben.

Ausgangszustand:



auf(A, B)
frei(A)
frei(C)

Zielzustand:



und(auf(A, B),
auf(B, C))
frei(A)

An diesem Beispiel ist besonders interressant, daß im Ausgangszustand bereits ein Teilziel existiert, welches auch ein Teilziel des Zielzustandes ist (auf (A, B)). Um den Zielzustand zu erreichen, muß dieses Teilziel wieder zerstört werden.

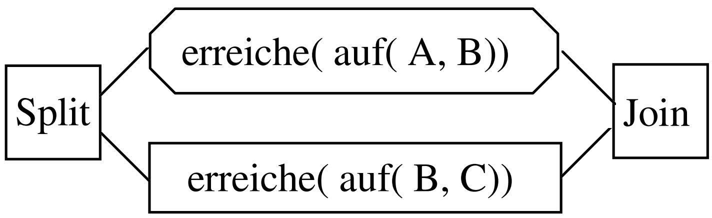
Phantomknoten werden bei der Auswertung durch ein abgerundetes Rechteck und Goals durch ein Rechteck dargestellt.

Das System startet mit einem einzigen Goal, welches den gesamten Zielzustand beschreibt.

1. Phase: erreiche(und(auf(A, B), auf(B, C)))

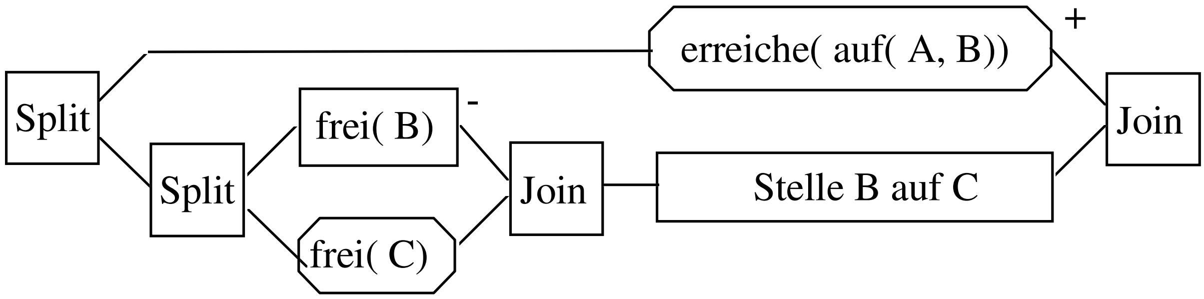
In der zweiten Phasen wird gemäß dem NOAH-Algorithmus ein detaillierterer Plan erstellt.
Die Konjunktion wird aufgespalten, so daß beide Teile unabhängig voneinander erreichbar sind.

2. Phase:



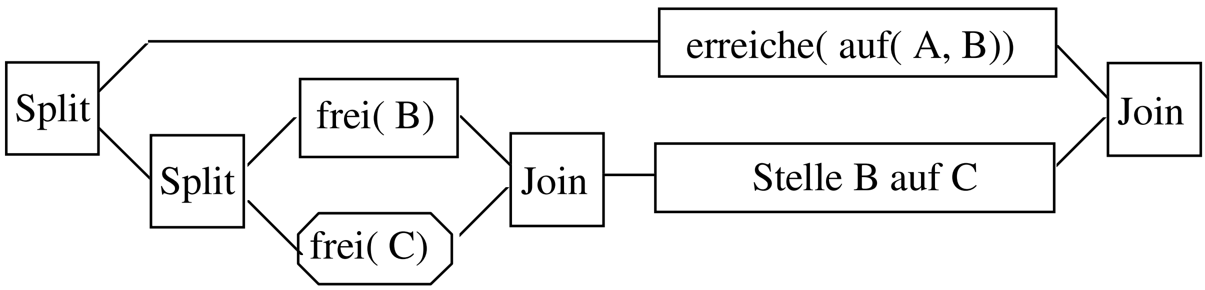
Da auf(A, B) bereits im Ausangszustand erreicht wurde, handelt es sich hierbei erstmal um einen Phantom-Knoten, der in der nächsten Phase nicht weiter ausgewertet werden muß. Für den zweiten Knoten wird nun ein detaillierter Plan gesucht, indem man ihn in drei Unterknoten aufspaltet, die aus der Liste der Unterfunktionen entnommen wurden.

3. Phase:
(Vor der Auswertung)



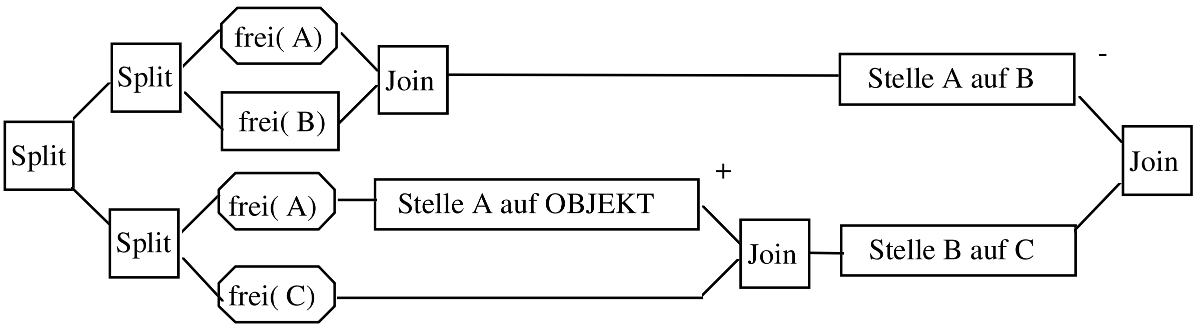
An dieser Stelle bemerkt das System, daß ein Knoten eine Vorbedingung für ein Teilziel auslöscht. Die Vorbedingung ist in diesem Fall „auf(A, B)“ und der widersprechende Knoten „frei(B)“. Der Knoten, der zuerst erreicht werden muß, wurde mit einem „-“ und der zeitlich danach kommende Knoten mit einem „+“ gekennzeichnet. Denn bevor A auf B gesetzt werden kann, muß B, genau wie A, frei sein. Um diesen Konflikt zu lösen, wird der Phantom-Knoten in einen Goal-Knoten umgewandelt und es ergibt sich folgender Plan.

3. Phase:
(Nach der Auswertung)



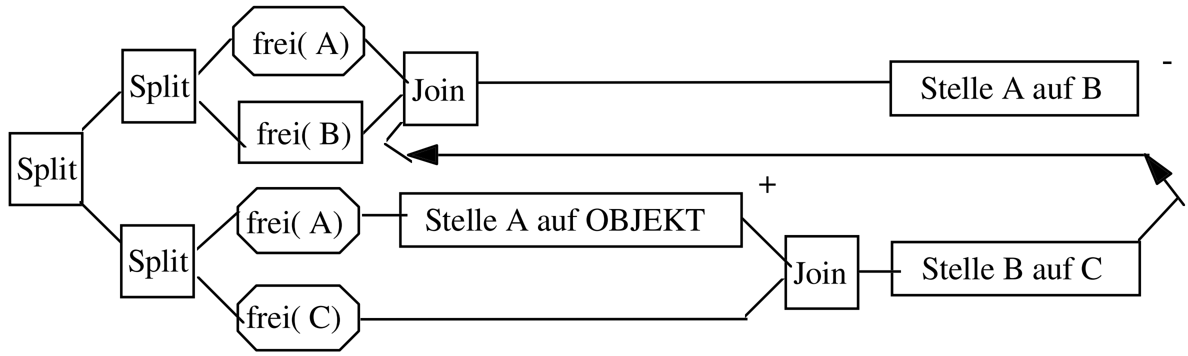
Jetzt können alle Knoten bis auf den Phantom-Knoten „frei(C)“, der schon durch den Ausgangszustand erreicht wird, neu evaluiert werden. „frei(B)“ wird nur erfüllt, wenn vorher A weggestellt wurde und das kann nur geschehen, wenn A vorher frei war. „erreiche(auf(A, B))“ wird nur erfüllt, wenn man A auf B stellt und das kann nur geschehen, wenn A und B vorher frei waren. Es ergibt sich:

4. Phase:
(Vor der Auswertung)



Auch hier ergibt sich wieder ein Konflikt, weil „Stelle A auf B“ dazu führt, daß B nicht frei ist, während „Stelle A auf OBJEKT“ dazu führt, daß B frei ist. Es muß eine zeitliche Reihenfolge erwirkt werden:

4. Phase:
(Neustrukturierung)

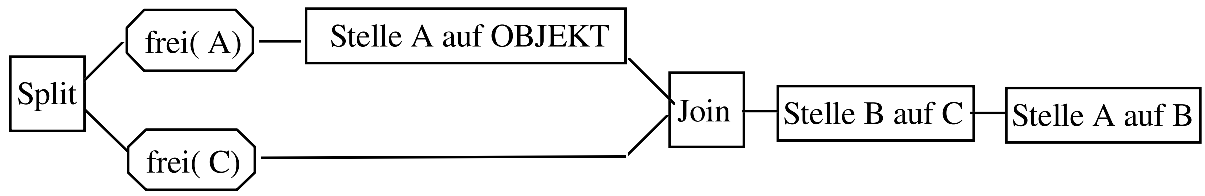


Durch den Pfeil und die Vorzeichen wird deutlich gemacht, daß „Stelle A auf OBJEKT“ zeitlich vor „Stelle A auf B“ durchgeführt werden muß.

Außerdem ist der Knoten „frei(B)“ redundant, da „Stelle A auf OBJEKT“ schon dazu führt, daß B frei wird. Genau wie der Knoten „frei(A)“, der zweimal aufgetaucht.

Nach der Eliminierung des Konfliktes und der redundanten Knoten ergibt sich:

4. Phase:
(Nach der Neustrukturierung)



Da „frei(A)“ und „frei(C)“ bereits durch den Ausgangszustand erreicht werden, lautet der endgültige Plan:

Stelle A auf OBJEKT
Stelle B auf C
Stelle A auf B

2.2.2 NONLIN

NONLIN beginnt wie NOAH mit einem einzigen Knoten. Auch der einfache Entwicklungszyklus ist sehr ähnlich:

1. Erweitere einen Knoten unter Verwendung der Expansionsvorschriften .
2. Verbessere alle Aktionen.
3. Beginne solange wieder bei 1. bis sich keine Knoten mehr erweitern lassen.

Jeder Knoten beinhaltet folgende Komponente :

nodenum	eine Nummer
nodetype	Ziel, Aktion oder Phantom
pattern	um ein Expansionsschema zu suchen
prenodes	eine Liste von direkt davor verbundenen Knoten
succnodes	eine Liste von direkt danach zu verbindenen Knoten
nodectxt	ein Kontext, der die Effekte des Knoten enthält
parentnode	der Knoten wurde durch die Expansion seines parentnode eingefügt
nodemark	ein temporärer Markierer, um Relationen dieses Knotens während der Expansion zu speichern

Die Expandierung eines Knotens hängt von seinem Type (nodetype) ab:

Zielknoten	Falls es eine Expandierung gibt, die ‘pattern’ wahr oder falsch macht, wird der Zielknoten durch die Expandierung ersetzt.
Phantomknoten	Ein Phantomknoten ist ein Zielknoten, der bereits erreicht worden ist. Es soll damit veranschaulicht werden, daß dieser Knoten nicht weiter expandiert werden muß.
Aktionsknoten	Im Aktionsknoten steht explizit ein Befehl, der etwas tut oder bewirkt. Es werden auch keine Annahmen gemacht, ob ‘pattern’ dadurch falsch oder wahr wird.

Bei NONLIN [Tate77] ist es auch möglich, einmal erreichte Ziele wieder rückgänig zu machen. NONLIN unterstützt einen einfachen interaktiven Planungsalgorithmus, da es den Anwender während des Prozesses nach Informationen fragt, die dem System fehlen.

2.2.3 TWEAK

Der von David Chapman entwickelte Algorithmus faßt in der Vergangenheit entwickelte Verfahren zusammen und gilt deswegen als korrekt und vollständig. TWEAK erweist sich jedoch als in der Anwendung nicht praktikabel, da es nur eingeschränkte Darstellungsmöglichkeiten für Aktionen und Weltzustände bietet.

Mittels Breitensuche wird bei TWEAK [Geibel92] ein Ziel ausgewählt, welches das System durch Einfügen neuer Schritte zu erfüllen versucht. Auch hier wird also nach dem Prinzip der Zielreduktion gearbeitet (goal reduction).

Die Vorbedingungen der neuen Aktionen sind zu erfüllende Unterziele, auf die der Algorithmus wiederum angewendet wird (constraint posting).

Algorithmus (ausgehend von einem leeren Plan):

1. Die Zeitrelation T wird verfeinert.
2. Die Äquivalenzrelation wird spezifiziert.
3. Neue Aktionen werden so ausgewählt, daß das Ziel erreicht wird.

Die Bedingungen werden schrittweise an die Reihenfolge der Aktionen und an die Variablenbindungen festgelegt. Sollten sich die Bedingungen nicht mehr verschärfen lassen, setzt der Algorithmus zurück (Backtracking).

Ein Plan P wird definiert durch P = (A, T, R), wobei A die Aktionen, T die partielle Ordnung und R eine Äquivalenzrelation sind. Während der Berchnungsphase kann ein Plan auf drei Weisen unvollständig und unfertig sein:

a) nicht alle Variablen müssen an Werte gebunden sein,
b) es können Aktionen fehlen oder
c) T ist eine partielle Ordnung.

Begonnen wird mit einem Plan, der nur ‘init’ enthält. Zur Lösung des Planungsproblemes wird eine Prozedur ‘erfülle.Ziel’ nacheinander auf jedes Zielliteral angewendet und falls diese Prozedur dann ‘Sackgasse’ zurückliefert, so erfolgt ein Zurücksetzen zur letzten relevanten Auswahlstelle innerhalb von ‘erfülle.Ziel’. Also wurde in ‘erfülle.Ziel’ entweder die falsche Aktion unter zwei möglichen ausgewählt oder es muß ein anderer Zweig innerhalb der indeterministischen oder-Auswahl gewählt werden.

Durch das Einfügen neuer Schritte entstehen dann neue unerreichte Ziele, auf welche wiederum ‘erfülle.Ziel’ angewendet wird bis alle Ziele erfüllt sind.

Wenn es einen Plan gibt, findet TWEAK einen Plan und terminiert. Falls es keinen Plan gibt, hält TWEAK mit einer Fehleranzeige an oder er terminiert nicht. Dieses kann eintreten, wenn man zum Beispiel zwei Aktionsschemata a und b hat, wobei a die Vorbedingung -g und die Nachbedingungen g und ¬h hat, während b die Vorbedingung ¬h und die Nachbedingungen h und ¬g hat. Wenn man dann von der Anfangsituation (¬g, ¬h) und dem Ziel (g, h) ausgeht, gerät TWEAK in eine Endlosschleife.

TWEAK hat den Vorteil, auf einer mathematisch präzisen Beschreibung zu beruhen und ordnet deshalb viele der bekannten Planungsalgorithmen, die vorher mehr intuitiven Charakter hatten, in einen formalen Rahmen ein. Die präzise Darstellungsweise führt dazu, daß sich theoretische Aussagen über das Planungsproblem im Allgemeinen machen lassen. Um aber in der Praxis einsetzbare Planungssysteme gewinnen zu können, muß man gerade von allgemeinen Planungsansätzen wegkommen. Nur gebietsspezifische Methoden können anscheinend eine ausreichende Effizienz gewährleisten.

3. Domäne Klötzchenwelt

3.1 Aufbau der Klötzchenwelt

Im Folgenden wird die Domäne Klötzchenwelt dargestellt. Diese Domäne wird oft als anschauliches Beispiel zur Dokumentation von Planungsalgorithmen herangezogen. Bestimmt wird diese Domäne von Fakten, Eigenschaften, Relationen und Regeln. Die Zusammenhänge von Eigenschaften und Relationen sind in den Zuständen enthalten. Wenn bestimmte Regeln in einem Zustand zutreffen, dann werden Aktionen ausgeführt. Diese Aktionen überführen den gegebenen Zustand in einen nachfolgenden Zustand.

3.2 Begriffsdefinitionen der Klötzchenwelt

Neben der Definition der Domäne Klötzchenwelt sollen jetzt noch zusätzliche Begriffe erklärt werden:

Ort	:	(p, q, r,...,z) Objekte der Domäne, die nicht bewegbar sind
Block	:	(A, B, C,..., O) Aktive Objekte der Domäne, werden durch Aktionen bewegt
Position	:	(A, B, C,..., O) U (p, q, r,...., z) Die Menge aller Blöcke und Orte
Zustand	:	Eigenschaften und Relationen der Blöcke und Orte
Aktion	:	Eine Aktion überführt ein Zustand in den Nachfolgezustand (z.B. ‘A auf q’)
Aktionsfolge	:	Mehrere Aktionen, die nacheinander ausgeführt werden
Oberblock	:	Der oberste Block einer Position
freie Orte	:	Alle Orte bis auf den Zielort und den Standort
Ausgangszustand	:	Ausgangslage des Planers
Zielzustand	:	Ziel des Planers
Teilziel	:	Ein Element des Zielzustandes
Fehlende Teilzustände	:	Alle noch nicht erreichten Teilziele
Zugblock	:	Oberblock eines Teilzieles
Zielort	:	Ort eines Teilzieles

Standort	:	Ort eines Zugblockes
freie Blöcke	:	Blöcke, auf denen keine weiteren Blöcke stehen

3.3 Umsetzung der Klötzchenwelt im PLATZ-Algorithmus

Die Implementation des Lösungsalgorithmus PLATZ erfolgte in PROLOG, da die Regelstrukturen von Planungsalgorithmen leicht in die PROLOG-Regelbasis übernommen werden können.
Die Implementation in anderen Programmiersprachen, die so ähnliche Regelstrukturen aufweisen, wie zum Beispiel LISP, ist ebenfalls einfach möglich.

Fakten :

Die Objekte der Klötzchenwelt werden durch Fakten festgelegt.

```
block( a).  block( b). block( c). block( d).
ort( p).    ort( q).   ort( r).   ort( s).
```

Objekte vom Typ Block können nicht gleichzeitig vom Typ Ort sein

Neben den Objekten werden noch einige Testfälle durch Fakten festgelegt (siehe bei Zustände)

Eigenschaften von Objekten :

Block

Block α ist frei, wenn kein weiterer Block auf Block α steht.
Dieses wird festgestellt, indem die Prozedure `oberblock` mit *Position = Block α* aufgerufen wird und die Rückgabe *Oberblock = Block α* ist.
Die *Höhe* gibt dabei die Anzahl, der auf der *Position* stehenden *Blöcken* an, bezogen auf einen *Zustand* (*Zaehler* dient als Merker).

```
oberblock( Oberblock, Position, Zustand, Hoehe, Zaehler)
```

Ort

Ort α ist frei, wenn kein Block auf Ort α steht.
Dieses wird festgestellt, indem die Prozedure `oberblock` mit *Position = Ort α* aufgerufen wird und die Rückgabe *Oberblock = Ort α* ist und *Höhe = 0* (*Zaehler* dient als Merker).

```
oberblock( Oberblock, Position, Zustand, Hoehe, Zaehler)
```

Relationen von Objekten :

Block

Block α steht auf Block β.
Um festzustellen, ob *Block* α auf *Block* β steht, wird geprüft, ob diese *auf*-Relation in dem gegebenen *Zustand* vorhanden ist.

```
member( auf( Block α, Block β), Zustand)
```

Block

Der Standort Block α ist Ort β.
Ein *StandOrt* ist der Ort unter einem *Block*, bezogen auf einen *Zustand* (*Position* dient als Merker).

```
standort( Block, Position, Zustand, StandOrt)
```

Zustände :

Die Zustände ergeben sich aus bestimmt Relationen und Eigenschaften der Objekte (Ort und Block). Immer wenn die gleichen Relationen und Eigenschaften für die gleichen Objekte gelten, ist ein äquivalenter Zustand erreicht.

Ausgangs- und Endzustände werden durch Fakten vorgegeben :

```
ausgangszustand( test4, [ auf( c, q), auf( b, a), auf( a, p) ] )
zielzustand( test4,     [ auf( a, b), auf( b, s), auf( c, r) ] ).
```

Dies bedeutet, daß im Ausgangszustand Block c auf Ort q, Block b auf Block a und Block a auf Ort p steht. Im Zielzustand soll Block a auf Block b, Block b auf Ort s und Block c auf Ort r stehen.

Von den Fakten des Ausgangszustandes werden durch Änderung von Relationen der Objekte zueinander oder durch Änderung von Eigenschaften der Objekte neue Zustände abgeleitet. Dies geschieht solange, bis der Zielzustand erreicht worden ist.

Aktionen :

Aktionen werden nur von Blöcken ausgeführt. Hierbei wird die Relation des Blockes verändert. Der Block wird aus seiner vorheriger Beziehung gelöscht und in einer neuen Beziehung eingefügt (Zustandswechsel).

Regeln :

Regeln gelten nur für die Aktionen von Blöcken.
Die Regeln geben an, welche Überführungsaktionen möglich sind. In der Klötzchenwelt können nur die Blöcke Aktionen ausführen. Eine Aktion wird ausgeführt, wenn eine bestimmte Regel zutrifft.
Es gelten die Regeln :

- Setze Block α nur auf Block β, wenn Block α und Block β frei sind
- Setze Block α nur auf Ort β, wenn Block α und Ort β frei sind

3.4 Konsistenzanforderungen des PLATZ-Algorithmus

Neben der allgemeinen Beschreibung der Domäne Klötzchenwelt müssen noch die folgenden Konsistenzbedingungen gelten :

- Jeder Block kann nur auf einem andern Block oder einem Ort stehen
- Auf einem Ort kann nur ein Block stehen
- Blöcke und Orte können bei einer Zustandsüberführung weder entstehen noch gelöscht werden
- Jeder Ort und jeder Block darf nur einmal vorhanden sein
- Eine Zustandsüberführung bezieht sich auf ein Objekt und seine Relation
- Blöcke können keine Orte und Orte keine Blöcke sein
- Ausgangs- und Zielzustand müssen bekannt sein
- Die Berechnungsfunktion für das nächste Teilziel muß optimal sein

4. PLATZ-Algorithmus

4.1 Einführung PLATZ-Algorithmus (Planung abhängigier Teilziele)

Der PLATZ-Algorithmus wurde für allgemeine Planungsprobleme konzipiert, die sich aus mehreren Teilzielen zusammensetzen lassen. Einzelne Teilziele können dabei je nach Abhängigkeit sowohl aufgelöst werden, als auch unauflösbar sein.

Der erstellte Lösungsplan besteht aus einem oder mehreren linearen Plänen, aus denen sich eine nichtlineare Zugfolge konstruieren läßt. Das Programm ist domänenunabhängig, hat aber domänen-spezifische Schnittstellen und die Regeln einer Domäne sind dabei deklarativ in dem Programmcode des Planers umgesetzt.

In diesem Fall wurde die Domäne Klötzchenwelt (siehe 3.) benutzt. Diese Domäne wird oft als anschauliches Beispiel zur Dokumentation von Planungsalgorithmen benutzt, da sie einerseits leicht vertändlich ist, aber andererseits wichtige Planungskriterien beinhaltet.

PLATZ versucht jedes einzelne Teilziel, welches zum Zielzustand führt, nach dem Best-First-Prinzip [Bratko88] zu erreichen. Die Idee dabei ist, die Suche immer vom meistversprechendsten Knoten der Anwärtermenge fortzusetzen. Im Gegensatz zur Breitensuche, die für die Erweiterung immer einen kürzesten Anwärterpfad sucht (das ist der flachste Endknoten der Suche), verfeinert die Best-First-Suche dieses Verfahren durch Berechnung einer heuristischen Bewertung für jeden Anwärter. Eine solche Bewertung eines Knotens gibt an, wie vielversprechend ein Knoten in bezug auf das Erreichen eines Zielknotens ist. Dann wird der für die Erweiterung gemäß dieser Bewertung beste Anwärter ausgewählt.

Ausgangszustand und Zielzustand müssen bekannt sein. Es werden zunächst alle unverletzbaren Teilziele vom Zielzustand abgeleitet, die unbedingt als erstes erreicht werden müssen (domänen-abhängig). Diese Teilziele werden durch eine Funktion, die die Anzahl der Überführungsschritte zählt, bewertet und nach dem Greedy-Verfahren ausgewählt. Die Bewertungsfunktion sollte dabei genau sein.

Um das ausgewählte Teilziel zu erreichern, wird nun eine serielle Aktionsfolge ausgeführt, indem nach der ersten Aktion wieder ein rekursiver Aufruf stattfindet.

Sobald eines der unverletzbaren Teilziele erreicht wurde, wird es gesperrt, indem es in die Liste der erreichten Teilziele überführt wird und anschließend nicht mehr aufgelöst werden kann.

4.2 PLATZ-Algorithmus

Als erstes werden im Hauptprogramm Ausgangszustand und Zielzustand zugewiesen. Fünf Testfälle sind bereits vorher erstellt worden und sind erweiterbar. Für eine neue Domäne müssen neue Zustände zugewiesen werden. Die Liste der erreichten Teilziele ist zunächst leer.

Anschließend wird der Zielzustand nach bereits erreichten Teilzielen im Ausgangszustand durchsucht. Falls diese Teilziele laut Zielzustand als erstes erreicht werden müssen, überführt man sie bereits schon hier in die Liste der erreichten Teilziele. Alle anderen erreichten Teilziele des Ausgangszustandes können wieder zerstört werden.

Jetzt wird der `Plangenerator` aufgerufen, der die als nächstes zu erreichenden Teilziele mittels des Zielzustandes ermittelt. Diese Liste der Teilziele gibt der `Plangenerator` an den `Zuggenerator` weiter.

Der `Zuggenerator` bestimmt das nächste Teilziel und alle Züge, die unabhängig voneinander möglich wären, um dieses Teilziel zu erreichen. Er entscheidet also, was noch gezogen werden muß. Die Funktion `Oberblock` gibt dabei die Aktion zurück, welche beim Erreichen eines Teilzieles unbedingt als erstes ausgeführt werden muß. Auch diese Funktion muß bei einer anderen Domäne zugewiesen werden. Bei der Auswahl der Züge werden die Objekte des Teilziels gesperrt, damit das aktuelle Teilziel nicht verschlechtert wird. Die vom `Zuggenerator` erstellte Liste gibt der `Zuggenerator` an den `Plangenerator` zurück.

Nun wird der `Zugexekutor` vom `Plangenerator` mit der Liste der möglichen Züge aufgerufen und vollzieht den ersten Zug (Zustandswechsel). Es erfolgt ein rekursiver Aufruf des `Plangenerators`.

Nach Abarbeitung aller Zugmöglichkeiten (Backtracking-Punkte) wurde eine Liste mit allen Aktionen gefunden, die den gegebenen Zustand in einen nachfolgenden Zustand überführt.

Dies geschieht solange, bis alle Teilziele erreicht wurden und der Lösungsplan gefunden wurde.

4.3 Schnittstellen der Domäne

- Ausgangs- und Zielzustand

```
ausgangszustand( test1, [ auf( a, r), auf( b, a), auf( c, b) ] )
zielzustand( test1,     [ auf( c, r), auf( b, c), auf( a, b) ] )
```

Ausgangs- und Zielzustand der Domäne werden durch zwei Listen beschrieben.

- Anzahl Überführungsschritte

```
anzahlUeberfuehrungsschritte( AZ, [auf( Block, ZielPosition)],
[Bewertung])
```

Um vom aktuellen Zustand (AZ) ausgehend ein bestimmtes Teilziel zu erreichen (in der Blockswelt: [auf(Block, ZielPosition)]), benötigt man eine bestimmte Anzahl von Überführungsschritten.

- Standort

```
standort( Block, Position, AZ, StandOrt)
```

Zu jedem Block im aktuellen Zustand (AZ) gibt es genau einen Standort, auf dem der unterste Block steht.

- Teilzieldomäne

```
teilZieleDomaene( auf( _, Ort), ort( Ort))
```

Hier wird ein Teilziel der Domäne spezifiziert. In diesem Fall ist ein Teilziel der Domäne Blochsworld, daß ein Objekt auf einem Ort steht.

- von_Objekt

```
von_Objekt( ZZ, ETZ, Teilziel)
```

Von_Objekt berechnet mittels der Liste aller Teilziele (ZZ) und der Liste aller bereits erreichten Teilziele (ETZ) das nächstmögliche Teilziel.

- Oberblock

```
oberblock( Oberblock, Position, Zustand, Hoehe, Zaehler)
```

Die Funktion Oberblock ermittelt die nächste Aktion, um ein Teilziel zu erreichen. Außerdem wird die Höhe des Turms berechnet, indem jeder Funktionsaufruf mitgezählt wird.

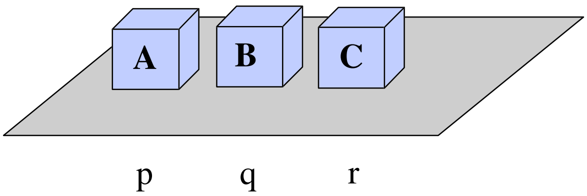
4.4 Erstellung von optimalen Plänen durch Berechnung der Zuganzahl

In einer Klötzchenwelt sei B die Anzahl der Klötze und O die Anzahl der möglichen Orte. Für die möglichen Züge Z ergibt sich:

$$Z = (O-1) * O = O^2 - O,$$

falls jeder Ort mit mindestens einem Klotz besetzt ist.

<u>Beispiel:</u>



O = 3 => $Z = 3^2-3$ => Es gibt 6 verschiedene Zugmöglichkeiten:
„A nach B“, „A nach C“, „B nach A“, „B nach C“, „C nach A“ und „D nach B“.

Um offensichtlich unsinnige Züge zu verhindern, sucht man nach Bewertungsmöglichkeiten, die es einen schon vorab erlauben, bestimmte Züge auszuschließen.

Sei „A auf X“ irgendein Teilziel zum Erreichen eines Zieles, wobei X sowohl ein Block als auch ein Ort sein kann. Der Zug „A auf X“ kann nur ausgeführt werden,.wenn:

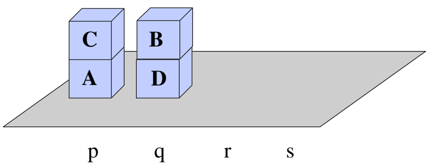
1. A frei ist, d.h. kein weiterer Block auf A steht und
2. X frei ist.

Falls A oder B nicht frei sind, müssen zunächst die Oberblöcke von A und X entfernt werden. Dabei benötigt man für jeden Oberblock einen Zug. Somit ergibt sich für die Anzahl der Züge um den Zug „A auf X“ auszuführen:

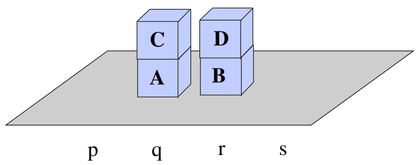
Anzahl Züge „A nach X“ = Anzahl Oberblöcke auf A + Anzahl Oberblöcke auf X + 1

<u>Beispiel:</u>

Anfangssituation :



Zielsituation :

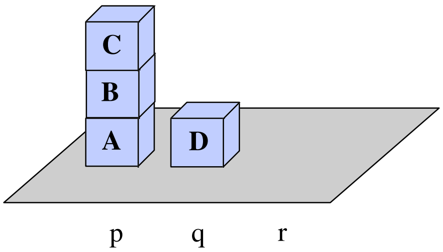


Anzahl Züge „A nach q“	=	Anzahl Oberblöcke auf A + Anzahl Oberblöcke auf q + 1
	=	1 + 2 + 1 = 4
Anzahl Züge „B nach r“	=	Anzahl Oberblöcke auf B + Anzahl Oberblöcke auf r + 1
	=	0 + 0 + 1 = 1

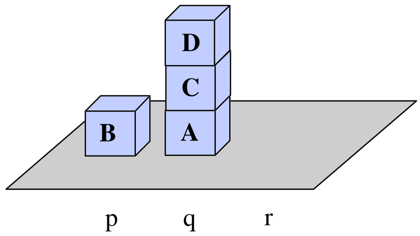
Ein Sonderfall tritt dann auf, wenn sich ein Block bereits auf seinem Zielort befindet und zwischen Block und Ort noch mindestens ein anderer Block steht.

<u>Beispiel:</u>

Anfangssituation :



Zielsituation :



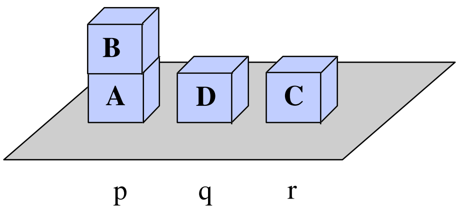
Anzahl Züge „A nach q“ = Anzahl Oberblöcke auf A + Anzahl Oberblöcke auf q + 1
= 2 + 1 + 1 = 4

Anzahl Züge „B nach p“ = Anzahl Oberblöcke auf B + Anzahl Oberblöcke auf r + 1
= 1 + 3 + 1 = 5

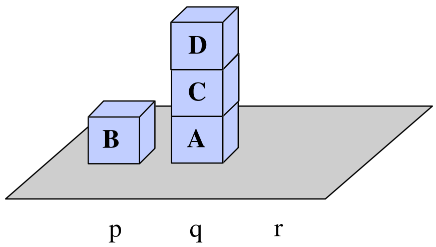
Das Teilziel „A nach q“ scheint also schneller erreichbar zu sein. Aber nachdem man den Oberblock von A verstellt hat, ergibt sich:

<u>Beispiel:</u>

Anfangssituation :



Zielsituation :



Anzahl Züge „A nach q“ = Anzahl Oberblöcke auf A + Anzahl Oberblöcke auf q + 1
= 1 + 1 + 1 = 3

Anzahl Züge „B nach p“ = Anzahl Oberblöcke auf B + Anzahl Oberblöcke auf r + 1
= 0 + 2 + 1 = 3

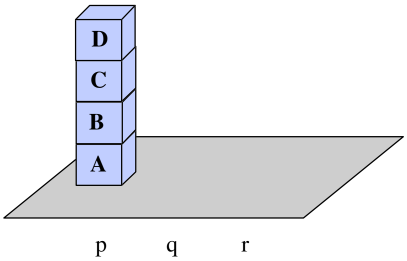
Das Bewegen eines Oberblockes verringert die Zuganzahl in diesem Fall also um zwei, da sowohl ein Oberblock des Blockes als auch ein Oberblock des Ortes bewegt wurde. Dieser Spezialfall muß man bei Berechnung der Zuganzahl immer berücksichtigen.

Es ist natürlich zu beachten, daß die so errechnete Anzahl von Zügen nur die minimale Zuganzahl zum Erreichen eines Teilzieles ist und es noch mehrere Möglichkeiten geben kann, dieses Teilziel mit mehr oder gleichvielen Zügen zu erreichen. Dabei muß verhindert werden, daß man die Oberblöcke des zu ziehenden Blockes auf den Zielort oder auf Oberblöcke des Zielortes stellt. Für die Oberblöcke des Zielortes gilt natürlich entsprechendes.

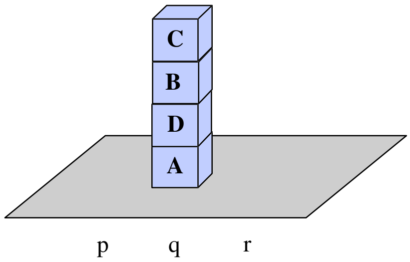
Ferner ist es ganz entscheidend, ob man von einer endlichen Menge von Orten ausgeht oder eine unendliche Anzahl von Orten zur Verfügung hat (table). Bei einer endlichen Menge kann es vorkommen, daß beim Ziehen von Blöcken (um ein Teilziel zu erreichen) andere später zu erreichende Teilziele verschlechtert werden.

<u>Beispiel:</u>

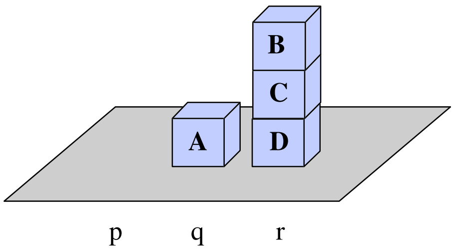
Ausgangssituation :



Zielsituation :

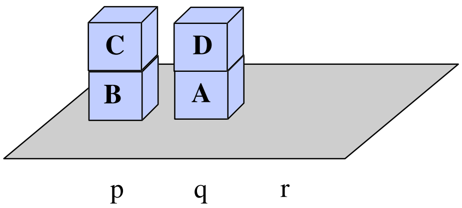


Das erste Teilziel „A nach q“ wird am schnellsten erreicht, indem man einfach die drei Oberblöcke nacheinander nach r zieht und erhält dann:



Dafür benötigt man vier Züge.

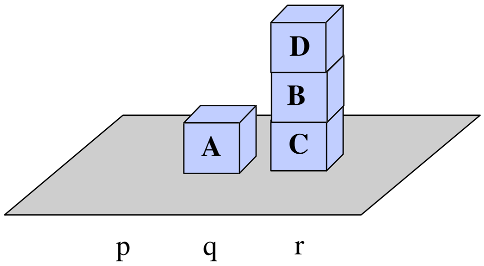
Um das nächste Teilziel „D auf A“ muß man entsprechend C und B nach p setzen und es ergibt sich:



Dafür benötigt man drei Züge.

Auch für die Teilziele „B nach D“ und „C nach B“ benötigt man noch drei Züge und man kommt auf eine Gesamtanzahl von zehn Zügen.

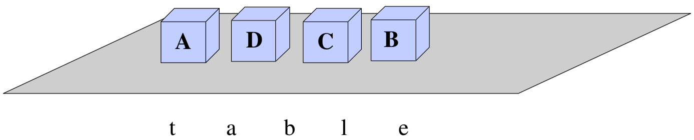
Hätte man jedoch von Anfang an alle Teilziele in Betracht gezogen, so würde die Zugfolge ganz anders ausgesehen. Zuerst wird D nach q und erst jetzt dann nach r gezogen wenn C und B bereits dort sind:



Wobei man allerdings für das erste Teilziel „A nach q“ bereits fünf Züge benötigt hat. Aber alle weiteren Teilziele „D nach A“, „B nach D“ und „C nach B“ lassen sich jetzt durch einen Zug erreichen und man erhält eine Zuganzahl von acht.

Hier wird deutlich, warum nichtlineare Planung stets alle Teilziele betrachten sollte, denn die benötigte Zuganzahl ist dadurch um 1/5 gesunken. Im Endeffekt ist ein Planer um so effizienter, je weiter er vorausdenken kann.

Bei einem table (eine Menge von unendlich vielen Orten) ist es jedoch unmöglich andere Teilziele durch Wegziehen von Blöcken zu verschlechtern, da Blöcke nicht übereinander gestellt werden. Für das erste Teilziels „A auf table“ benötigt man dann sogar null Schritte, da A bereits auf dem table steht und für das Teilziel „D nach A“ zieht man:



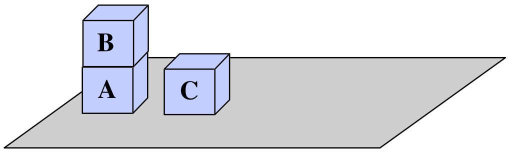
Man kommt auf eine Gesamtanzahl von sechs Zügen.

In der Praxis hat man natürlich nur einen begrenzten Raum zur Verfügung. Trotzdem wird bei den meisten nichtlinearen Planern von einem ‘table’ ausgegangen, was zu ganz anderen Ergebnissen führen kann.

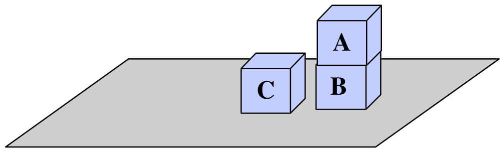
4.5 Ein Beispiel aus der Klötzchenwelt

In der Domäne der Klötzchenwelt (siehe 3.) werden als Beispiel folgende Ausgangs- und Zielzustände angenommen :

Ausgangszustand : Block B steht auf Block A, Block C steht auf Ort q,
Block A steht auf Ort p



Zielzustand : Block A steht auf Block B, Block B steht auf Ort s,
Block C steht auf Ort r



Aus Definition von AZ und ZZ ist zu erkennen, daß hierbei zwei Ziele erreicht werden sollen.

Diese sind : 1. Block A auf B UND Block B auf Ort s
2. Block C auf Ort r

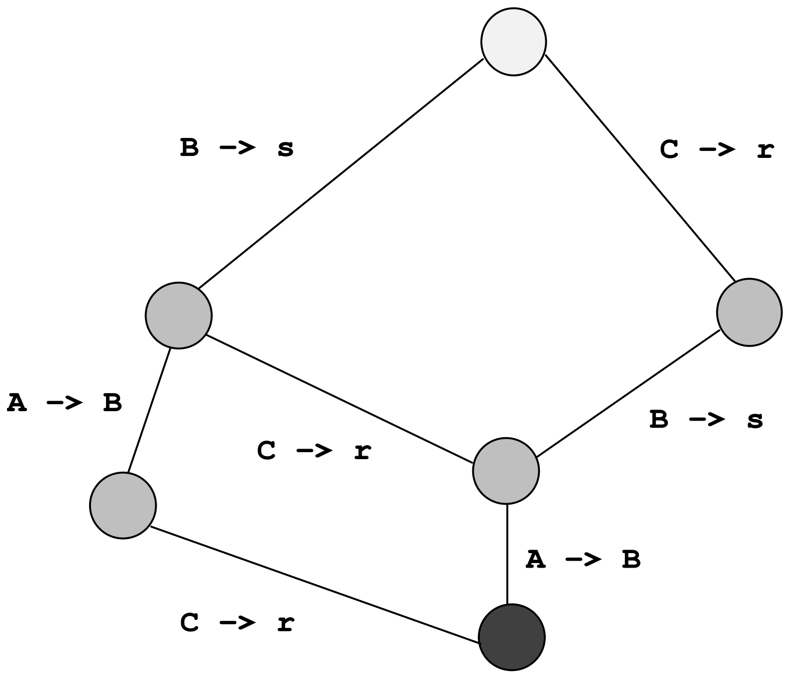
Die beide Teilziele können durch Aktionen erreicht werden, die nicht miteinander im Konflikt stehen (d.h., daß Aktionen auf Objekte eines Teilzieles keine Auswirkungen auf die Eigenschaften des anderen Objektes haben).

Der Programmablauf vom Lösungsalgorithmus PLATZ ist nichtlinear. Diese Nichtlinearität ist gegeben durch das Wiedererreichen der Backtracking-Punkte, nachdem ein Lösungsplan gefunden wurde. Alle Zwischenzustände stellen so einen Backtracking-Punkt dar.

Zur grafischen Darstellung des Zustandswechsels wird der Zustand-Überführungsbaum verwendet. Der Zustand-Überführungsbaum zeigt anschaulich, daß es

- genau einen Ausgangszustand gibt
- genau eine Zielzustand gibt
- verschiedene Aktionen für Zustandswechsel gibt (bedingt durch die geltenden Regeln)
- beliebig viele Zwischenzustände gibt

```
Ausgangszustand : [auf(a, p),auf(b, a),auf(c, q)]
Zielzustand     : [auf(a, b),auf(b, s),auf(c, r)]
```

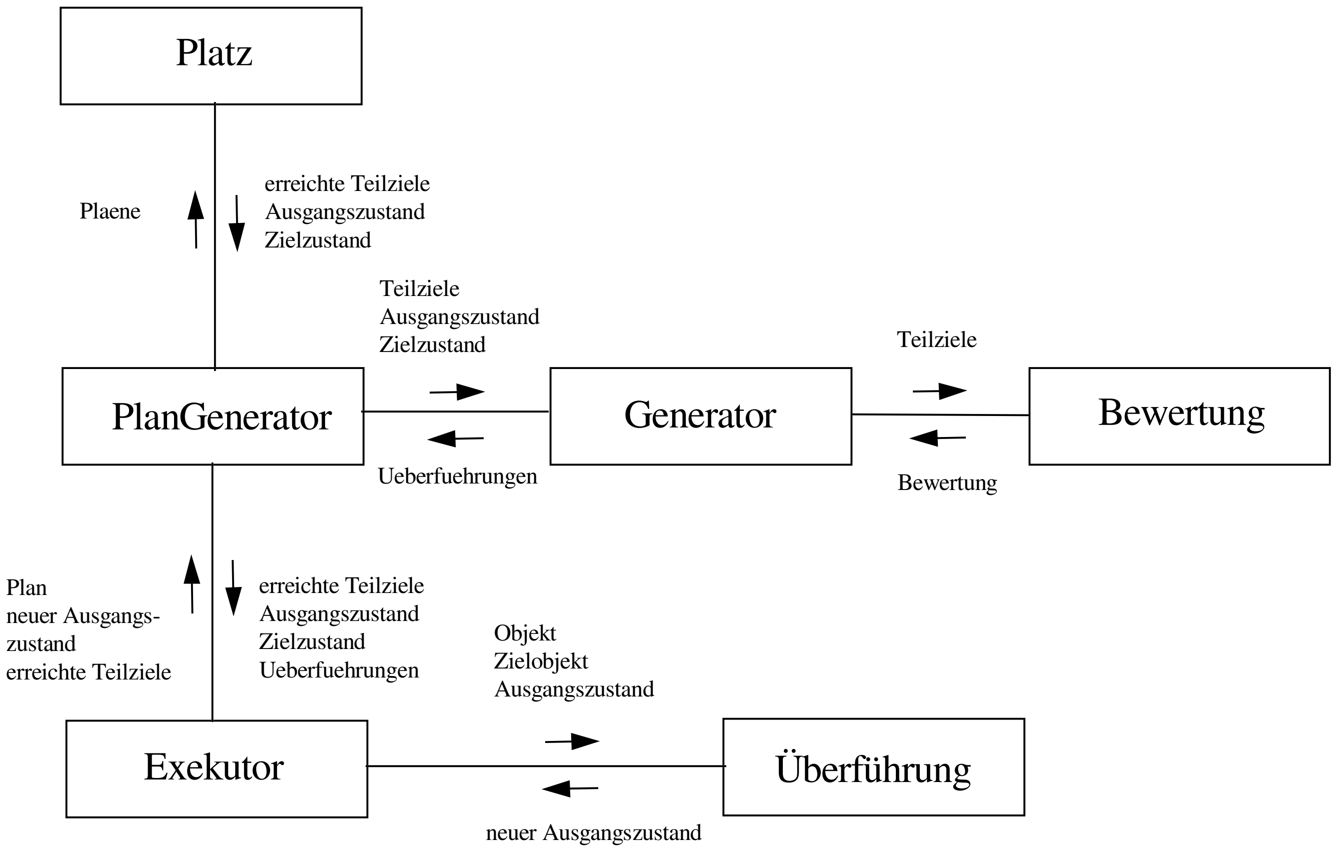


Darstellung des Zustandwechsels mit Hilfe des Zustand-Überführungsbaumes

Lösungsplan = [[C -> r, B -> s, A -> B],
[B -> s, C -> r , A -> B],
[B -> s, A -> B, C -> r]]

Die Teilziele werden durch nichtlineare Aktionen erreicht.

4.6 Datenfluß PLATZ-Algorithmus



5. Kritische Projektbetrachtung

Bei der Erstellung eines Planungsprogrammes sollte darauf geachtet werden, daß bestehende Lösungsalgorithmen domänenunabhängig betrachtet werden. Dies gilt insbesondere für nichtlineare Planer. Bei einer starken Fixierung auf eine bestimmte Domäne wird die nichtlineare Funktionalität durch domänenspezifisches Wissen ersetzt.
Dieses ist bei der Erstellung des Programmes PLATZ geschehen. Das erstellte Programm besitzt domänenspezifische Funktionen, und da die Klötzchenwelt bei fast allen der uns vorliegenden nichtlinearen Planungsprogramme die zu betrachtende Domäne war, ist der PLATZ-Algorithmus auch für diese konzipiert worden. Dazu gehören die Unverletztbarkeit des ausgewählten Teilzieles, die aktuelle Bewertungsfunktion, sowie die Vorsortierung nach verletzbaren und unverletztbaren Teilzielen.
Erst im späteren Projektverlauf konnte durch die Einführung von Domänenschnittstellen eine gewisse Abstraktion von der Klötzchenwelt erreicht werden. Die Schnittstellen und das Domänenwissen machen deshalb einen Hauptteil des Programmcodes aus.

Der PLATZ-Algorithmus läßt sich in die Kategorie der linearen Algorithmen einordnen, da er weder vorrausschauen noch Ziele erweitern kann. Trotzdem liefert PLATZ nichtlineare Lösungspläne, die Überschneidungspunkte besitzen. Er ist nicht in der Lage, Ergebnisse zusammenzufassen, sondern gibt Pläne, die sich nur in einem ganz kleinen Teil unterscheiden, immer in voller Länge aus.
Um dieses zu verbessern, müßte eine neue Funktion geschrieben werden, die am Ende nochmal alle Lösungspläne analysiert und auf Gemeinsamkeiten untersucht. Eine Ausgabe könnte in Form eines Lösungsbaumes erfolgen, der dann die gemeinsamen Knoten verschiedener Lösungspläne und alternative Erreichbarkeitsmöglichkeiten anschaulich darstellt.

Um ein späteres Fortsetzten diese Projektes zu unterstützen, haben wir hier einige Punkte aufgezählt, die helfen sollen, die Arten der nichtlinearen- und der linearen Algorithmen besser zu unterscheiden.

Lineare Lösungsalgorithmen kennzeichnen

- definierte Ausgangs- und Endzustände,
- ein Lösungsplan, der ohne Merkfunktion ausgeführt werden kann,
- die Möglichkeit, bei einer Unterbrechung der Programmausführung Aussagen über den jeweiligen Zustand der Berechnung zu machen und
- Zustände und Zustandsüberführungen.

Nichtlineare Lösungsalgorithmen kennzeichnen

- evtl. nur Teilkenntnis über Ausgangs- und Endzustand,
- ein Lösungsplan, bei dem Merkfunktionen wichtig sind,
- unbestimmte Positionen (z.B. Objekt X) und
- daß bei Programmunterbrechung nichts über den derzeitigen Zustand ausgesagt werden kann.

6. Anhang

6.1 Pseudocode PLATZ-Algorithmus

```
PROGRAMM  platz
BEGIN
   IF Ausgangszustand <> Zielzustand THEN
   BEGIN
         Ermittlung erreichter Teilziele
         planGenerator( Ausgangszustand, Zielzustand, erreichte Teilziele)
   END
END
```

```
PROCEDURE planGenerator
BEGIN
   Teilziele = Teilziele - erreichten Teilziele
   Berechnung der Überführungsschritte pro Teilziel

   WHILE Überführungen <> []
   BEGIN
     generator(  Ausgangszustand, Zielzustand, Teilziele)
     Überführungen = Überführungen + Überführung
   END

   WHILE Ausgangszustand <> Zielzustand
   BEGIN
     exekutor( Ausgangszustand, Zielzustand, erreichte Teilziele,
                   Überführungen)
     Pläne = Pläne + Plan
   END
END
```

```
PROCEDURE generator
BEGIN
   Bewertung aller Teilziele
   Bestimmung der Überführung
END



PROCEDURE exekutor
BEGIN
   überführung( Ausgangszustand, Objekt, ZielObjekt)
   Ermittlung erreichter Teilziele
   planGenerator( Ausgangszustand, Zielzustand, erreichte Teilziele)
END


PROCEDURE überführung
BEGIN
   alte Relation des Objektes im Ausgangszustand löschen
   neue Relation des Objekt bezüglich des Zielobjektes
    im Ausgangszustand einfügen
END
```

6.2 Programmcode Lösungsalgorithmus PLATZ

```
/* *******************************************************************

                P L A T Z A L G O R I T H M U S                  */

platz( _) :-
       writeall( ['PLATZ-Algorithmus  - - -  H I L F E  - - - ',nl,nl]),
       writeall( ['Erstellung eines nichtlineare Loesungsplanes',nl]),
       writeall( ['durch Berechnung abhaengiger Teilziele',nl,nl]),
       writeall( ['Es sind fünf Testfaelle vordefiniert.',nl]),
       writeall( ['Diese koennen direkt aufgerufen werden.',nl]),
       writeall( ['Beispiel : "platz( test3, Z)"',nl,nl]),
       writeall( ['Durch setzen von "write_level" wird die Ausgabe']),
       writeall( [' eingeschraenkt',nl]),
       writeall( ['   0 ... Es erfolgt keine Ausgabe',nl]),
       writeall( ['   1 ... Die Teilplaene werden angezeigt',nl]),
       writeall( ['   2 ... Ausgangs- u. Endzustand + Namen',nl]),
       writeall( ['   3 ... Aktuelle- u. erreichte Zustaende',nl]),
       writeall( ['   4 ... Die Ausweichmoeglichkeiten werden
angezeigt',nl]),
       writeall( ['   5 ... Die Aktionen werden angezeigt',nl]),
       writeall( ['   6 ... Die Ueberfuehrungsfunktionen werden
angezeigt',nl]),
       writeall( ['   7 ... Es wird nach jedem Zustandswechsel
angehalten',nl,nl]),
       writeall( ['Die Einschraenkungen gelten ab dem Aufruf',nl]),
       writeall( ['von "platz_set". ( z.B. "platz_set( test4, Z, 5)")']), !.

platz_set( Name, Plaene) :-
       platz( Name, Plaene), !.

platz_set( Name, Plaene, Write_level) :-
       retract( write_level( _)),
       assert( write_level( Write_level)),
       platz( Name, Plaene), !.

platz( Name, Plaene) :-
            /* Programmstart mit platz( testX, Plaene)              */
      ausgangszustand( Name, Ausgangszustand),
            /* Zuordnung des Anfangs-                               */
      zielzustand( Name, Zielzustand),
            /* und Zielzustandes von testX                          */

      sort( Ausgangszustand, SortAusgangszustand),
      sort( Zielzustand, SortZielzustand),

      writeall( [6,'Ausgangszustand : ',SortAusgangszustand,nl]),
      writeall( [6,'Zielzustand     : ',SortZielzustand,nl,nl,nl]),

      findall( TZ, (member( TZ, Ausgangszustand), member( TZ,
                    Zielzustand),
                   TZ = teilZieleDomaene
                 ), ETZ),
            /* Ermittlung der schon erreichten Teilziele            */

      planGenerator( Ausgangszustand, Zielzustand, ETZ, [], KPlaene),

      writeall( [2,nl,'Ausgangszustand : ',SortAusgangszustand]),
      writeall( [2,nl,'Zielzustand     : ',SortZielzustand,nl,nl]),

      writeall( [2,'                    PLATZ von H. Holst', nl]),
      writeall( [2,'                          und J. Schellenberger',nl]),
      klammernweg( KPlaene, Plaene),
      !.
```

```
/* --> planGenerator <-----------------------------------------------

pre   : aktueller Zustand (AZ),
        fehlende Teilziele (FTZ)
post  : alle Teilziele wurden erreicht (FTZ=[] und AZ=Zielzustand)
eff   :

*/

planGenerator( AZ, ZZ, _, Plan, Plan) :-
            /* Es wurde ein Plan gefunden */

     findall( Element, element_in( AZ, ZZ, Element), Elemente),
     Elemente = [],
            /* kein Element aus AZ ist nicht in ZZ ( AZ = ZZ )      */

     writeall([1,nl,'Ein Teilplan       : ',Plan,nl, nl]),
     !.

            /* Ein Plan wurde erstellt, zurück
               zum nächsten Backtrackingpunkt                       */

planGenerator( AZ, ZZ, ETZ, TempSortTeilziele, Plaene) :-
            /* AZ = aktueller Zustand                               */
            /* FTZ = fehlende Teilziele                             */
     teilziele( ZZ, ETZ, Teilziele),
            /* Welche Blöcke stehen im FTZ auf einem                */
            /* Ort --> Teilziele                                    */

     anzahlUeberfuehrungsschritte( AZ, Teilziele, UnSortTeilziele  ),
            /* 'wieviele Züge sind zum Erreichen des
                Teilzieles notwendig ?'                             */
            /* Diese Funktion ist nur notwendig, wenn die Möglich-
               keiten der Folgezustände beschränkt werden soll      */

     sort( UnSortTeilziele, AlleSortTeilziele),
            /*   Die Teilziele, die am schnellsten erreicht
                 werden können, stehen vorne                        */

     ausgabeAnzahlUeberfuehrungsschritte( AlleSortTeilziele),

     findall( Ueberfuehrung,
            generator(  AZ, ZZ, Ueberfuehrung, AlleSortTeilziele),
             UnSortUeberfuehrungen),
            /* Alle Züge des Zuggenerators werden gesammelt         */

     sort( UnSortUeberfuehrungen, Ueberfuehrungen),

     writeall( [6,'Ueberfuehrungen    : ',Ueberfuehrungen,nl]),
            /* Die Züge werden angezeigt                            */

     findall( Plan, exekutor( AZ, ZZ, ETZ, TempSortTeilziele,
                        Ueberfuehrungen, Plan),
            Plaene).
            /* Die Züge werden gezogen                              */

/* --> Ende PlanGenerator <------------------------------------------ */
```

```
/* --> generator <--------------------------------------------------

pre   :   aktueller Zustand (AZ),
            fehlende Teilziele (FTZ)
post  :   ein Teilziel wurde ausgewählt
            Liste aller möglichen Züge wurde erstellt
eff   :

*/

generator( AZ, ZZ, Ueberfuehrung, AlleSortTeilziele) :-
     beste_Bewertung( BesteBewertung, AlleSortTeilziele),
     member( EinTeilziel, AlleSortTeilziele),

     EinTeilziel = ( BesteBewertung, Objekt, Ziel),

            /* Bestimmung des Blockes und des
               Zielortes für ein Teilziel                        */

     moeglicheFolgeZustaende( AZ, ZZ, Ziel, Objekt,
                        MoeglicheFolgeZustaende),
            /* Orte, auf die gezogen werden darf                 */

     writeall( [4,'Teilziel           : ', Objekt, ' nach ', Ziel,nl ]),
     writeall( [4,'Frei ist           : ', MoeglicheFolgeZustaende,nl]),

     member( Folgezustand, MoeglicheFolgeZustaende),

     aktionsbestimmung( AZ, Ueberfuehrung, Folgezustand, Objekt, Ziel).

/* --> Ende generator-------------------------------------------------- */


/* --> exekutor <-----------------------------------------------

pre   : ein Teilziel wurde ausgewählt
        Liste aller möglichen Züge wurde erstellt
post  : Zustandswechsel ist erfolgt
eff   : rekursiver Aufruf vom Plangenerator

*/

exekutor( AZ, ZZ, ETZ, TempSortTeilziele, Ueberfuehrungen, Plan) :-
     member( [ueberfuehrung( Objekt, ZielObjekt),
            ziel( _, Ziel)], Ueberfuehrungen),
            /* Eine bestimmte Überführung auswählen              */

     writeall( [6,'Ueberfuehrung      : ',Objekt,' nach ',ZielObjekt,nl]),

     ueberfuehrung( AZ, NeuerAZ, Objekt, ZielObjekt),
            /* Zustandsaktualisierung                            */

     erreichte_Teilziele( ETZ, NeueETZ, Objekt,
                        ZielObjekt, Ziel),

     sort( NeuerAZ, SortAZ),
     sort( NeueETZ, SortNeueETZ),

     writeall( [3,'Akt. Zustand       : ',SortAZ,nl]),
     writeall( [3,'Err. Teilziele     : ',SortNeueETZ,nl]),

     writeall( [7,get,nl]),                          /* get( _), */
            /* warten auf Tastendruck                            */

     concat( Objekt, '->', Objekt2),
```

```
     concat( Objekt2, ZielObjekt, ATempSortTeilziele),

     flatten( [TempSortTeilziele, ATempSortTeilziele],
            NTempSortTeilziele),

     planGenerator( NeuerAZ, ZZ, NeueETZ, NTempSortTeilziele, Plan).
            /* rekursiver Funktionsaufruf                           */


/* --> Ende exekutor<---------------------------------------------------- */



beste_Bewertung( BesteBewertung, [( BesteBewertung, _, _)|_]).
            /* Bestimmung der kürzeste Überführungsanzahl           */



/* -- > ueberfuehrung <--------------------------------------------------

   pre  : -
   post : -
   eff  : AZ wurde in NeuerAZ überführt
        FTZ wurde in NeuerFTZ überführt
*/


ueberfuehrung( AZ, NeuerAZ, Objekt, ZielPosition) :-
     delel( AZ, NZ1, auf( Objekt, _)),
     NeuerAZ = [auf( Objekt, ZielPosition)| NZ1].


/* -- > teilziele <-----------------------------------------------------

   pre  : Die Zielliste des Endzustandes
   post : Eine Liste von Teilzielen
   eff  : Alle Blöcke, die auf einem Ort stehen, werden aus dem End-
            Zustand ermittelt
*/

teilziele( ZZ, FTZ, Teilziele) :-
      findall( Objekt, von_Objekt( ZZ, FTZ, Objekt), Teilziele).

/* -- > Ende teilziele <-------------------------------------------- */


/* -- > erreichte_Teilziele <--------------------------------------

eff  :  fügt ein eben erreichtes Teilziel in die Liste
         der bereits erreichten Teilziele ein

*/

erreichte_Teilziele( ETZ, NeueETZ, Objekt, ZielObjekt, ZielObjekt) :-
     NeueETZ = [auf( Objekt, ZielObjekt)| ETZ], !.

erreichte_Teilziele( ETZ, ETZ, _, _, _).

/* -- > Ende erreichte_Teilziele <-------------------------------- */
```

```
/* -- > element_in <---------------------------------------------------

eff  : Gibt ein Element zurück, dass im AZ, aber nicht im ETZ ist

*/

element_in( AZ, ETZ, Element) :-
     member( auf( A, B), AZ),
     not member( auf( A, B), ETZ),
     Element = auf( A, B).


/* ****************************************************************** */
/* ****************** E N D E   P L A T Z  ************************** */
/* ****************************************************************** */

/* ****************************************************************** */
/* ****************** DEFINITION DER DOMÄNE ************************** */
/* ****************************************************************** */

/*                     K L Ö T Z C H E N W E L T                      */



block( a).  block( b). block( c). block( d).
ort( p).    ort( q).   ort( r).   ort( s).

alleorte( [p,q,r,s]).

            /* Definition einiger Testfälle                            */

ausgangszustand( test1, [ auf( a, r), auf( b, a), auf( c, b) ] ) :-
  retract( alleorte( _)),        /* Umsortierung der Blöcke auf        */
  asserta( alleorte( [p,q,r])).  /* einem Ort                          */

ausgangszustand( test2, [ auf( c, p), auf( b, q), auf( a, b) ] ) :-
  retract( alleorte( _)),        /* Zerstörung bereits erreichter      */
  asserta( alleorte( [p,q,r])).  /* Teilziele                          */

ausgangszustand( test3, [ auf( b, p), auf( a, b), auf( c, q), auf( d, c) ] )
:-
  retract( alleorte( _)),       /* Überführungen mit gegenseitiger     */
  asserta( alleorte( [p,q,r])). /* Beeinflussung                       */

ausgangszustand( test4, [ auf( c, q), auf( b, a), auf( a, p) ] ) :-
  retract( alleorte( _)),         /* Überführungen ohne gegenseitige   */
  asserta( alleorte( [p,q,r,s])). /* Beeinflussung                     */

ausgangszustand( test5, [ auf( a, p), auf( b, a), auf( c, b), auf( d, c) ] )
:-
  retract( alleorte( _)),        /* Zerstörung bereits erreichter      */
  asserta( alleorte( [p,q,r])).  /* Teilziele                          */


zielzustand( test1,     [ auf( c, r), auf( b, c), auf( a, b) ] ).
zielzustand( test2,     [ auf( c, q), auf( b, c), auf( a, b) ] ).
zielzustand( test3,     [ auf( c, p), auf( d, c), auf( b, q), auf( a, b) ] ).
zielzustand( test4,     [ auf( a, b), auf( b, s), auf( c, r) ] ).
zielzustand( test5,     [ auf( b, p), auf( a, q), auf( c, b), auf( d, a) ] ).

teilZieleDomaene( auf( _, Ort), ort( Ort)).
            /* Ein 'Teilziel' wird für die Domäne bestimmt             */
```

```
/* --> anzahlUeberfuehrungsschritte <--------------------------------------

   pre  : -
   post : -
   eff  : Berechnet die Züge zum Erreichen eines Teilzieles

*/

anzahlUeberfuehrungsschritte( AZ, [auf( Block, ZielPosition)],
                              [Bewertung])  :-
      wert( ZielPosition, AZ, Anz1, 1),
            /* Bewertung des letzten                           */
      wert( Block, AZ, Anz2, Anz1),
            /* Teilzieles                                      */
      wertkorrektur( Anz2, Anz3, Block, ZielPosition, AZ),
      Bewertung = ( Anz3, Block, ZielPosition), !.



anzahlUeberfuehrungsschritte( AZ, [auf( Block, ZielPosition)|S],
[Bewertung|R]) :-
     wert( ZielPosition, AZ, Anz1, 1),
            /* zählt die Blöcke auf dem ZielPosition           */
     wert( Block, AZ, Anz2, Anz1),
            /* zählt die Blöcke auf dem Block                  */
     wertkorrektur( Anz2, Anz3, Block, ZielPosition, AZ),
     Bewertung = ( Anz3, Block, ZielPosition),
     anzahlUeberfuehrungsschritte( AZ, S, R).

/* --> Ende anzahlUeberfuehrungsschritte <------------------------------ */

/* --> wertkorrektur <-----------------------------------------------------

   pre  : -
   post : -
   eff  : Korrigiert die Anzahl der notwendigen Züge wenn der
        Standort schon erricht wurde
*/

wertkorrektur( Anz2, Anz3, Block, ZielPosition, AZ) :-
     standort( Block, _, AZ, Standort),
     standort( ZielPosition, _, AZ, Standort),
     oberblock( _, Block, AZ, Korrektur, 0),
     Anz3 is Anz2 - Korrektur, !.

wertkorrektur( Anz3, Anz3, _, _, _).

/* --> Ende wertkorrektur -----<----------------------------------------- */

ausgabeAnzahlUeberfuehrungsschritte( AlleSortTeilziele) :-
      writeall( [3,'Zuganzahl          : ', AlleSortTeilziele,nl]).

/* --> standort <-----------------------------------------------

pre  : -
post : -
eff  : Gibt den Orte zurück, auf dem ein Block steht

*/

standort( Block, Position, AZ, StandOrt) :-
      member( auf( Block, Position), AZ),
            /* Position ist kein Ort                           */
     not ort( Position),
     standort( Position, _, AZ, StandOrt), !.
```

```
standort( Block, Position, AZ, StandOrt) :-
     member( auf( Block, Position), AZ),
            /* Position ist ein Ort                              */
     ort( Position),
     StandOrt = Position, !.
standort( Ort, _, _, Ort).

/* --> Ende standort <-------------------------------------------------- */


/* --> wert <------------------------------------------------------------

   pre  : -
   post : -
   eff  : Zählt Blöcke auf einer Position

*/

wert( Position, AZ, Anz, Zaehler) :-
     member( auf( Block, Position), AZ),
     NeuZaehler is Zaehler +1,
     wert( Block, AZ, Anz, NeuZaehler), !.

wert( _, _, Anz, Zaehler) :-
      Anz is Zaehler.

/* --> Ende wert <------------------------------------------------------ */

/* -- > von_Objekt <----------------------------------------------------

pre  :
post : ein Teilziel
eff  : bestimmt von einem Ort ausgehend das nächstmögliche Teilziel
*/

von_Objekt( ZZ, ETZ, Teilziel) :-
     member( auf( _, Ort), ZZ),
     ort( Ort),
     oberblock( EPosition, Ort, ETZ, _, 0),
     member( auf( Position, EPosition), ZZ),
     Teilziel = ( auf( Position, EPosition)).

/* --> Ende von_Objekt <------------------------------------------------- */

/* -- > oberblock <-------------------------------------------------

   pre  : Position und Zustand
   post : Oberblock
   eff  : Gibt den Oberblock einer Position zurück und die
          Anzahl der Blöcke auf der Position
*/

oberblock( Oberblock, Position, Zustand, Hoehe, Zaehler) :-
     member( auf( Block, Position), Zustand),
            /* Wenn auf einer Position (Block oder               */
      not member( auf( _, Block), Zustand),
            /* Ort) ein Block steht und auf diesem               */
     Oberblock = Block,
     Hoehe is Zaehler +1,
     !.
            /* kein weiterer, dann ist dieser                    */
            /* Block der Oberblock                               */
```

```
oberblock( Oberblock, Position, Zustand, Hoehe, Zaehler) :-
     member( auf( Block, Position), Zustand),
            /* Block ist noch nicht der Oberblock                   */
     Zaehler2 is Zaehler +1,
     oberblock( Oberblock, Block, Zustand, Hoehe, Zaehler2),
     !.

oberblock( Position, Position, Zustand, Hoehe, Zaehler) :-
            /* Wenn auf einer Position kein Block steht,
               dann ist der Oberblock die Position                  */
      not member( auf( _, Position), Zustand),
      Hoehe is Zaehler.


/* -- > Ende oberblock <-------------------------------------------------- */


/* --> moeglicheFolgeZustaende <------------------------------------------------

pre  : Zielort und Objekt
post : -
eff  : Gibt alle Orte zurück, die nicht am Teilziel beteiligt sind

*/

moeglicheFolgeZustaende( AZ, ZZ, ZielPosition, Objekt,
                         MoeglicheFolgeZustaende) :-
      standort( Objekt, _, AZ, Standort_Objekt),
      standort( ZielPosition, _, ZZ, Standort_ZielObjekt),

      alleorte( AlleOrte),
      delel( AlleOrte, OrteOhneZielOrt, Standort_ZielObjekt),
      delel( OrteOhneZielOrt, MoeglicheFolgeZustaende, Standort_Objekt).


/* -- > aktionsbestimmung <-------------------------------------------------

   pre  : -
   post : -
   eff  : Bestimmung des Blockes und der Position
*/

aktionsbestimmung( AZ, Ueberfuehrung, _, Objekt, ZielPosition)  :-
      oberblock( Objekt, Objekt, AZ, _, 0),
            /* Block ist frei                                       */

      oberblock( ZielPosition, ZielPosition, AZ, _, 0),
            /* ZielPosition ist frei                                */
     Ueberfuehrung = [ueberfuehrung( Objekt, ZielPosition),
             ziel( Objekt,ZielPosition)],
     writeall( [5,'Aktion             : Ein Teilziel wird erreicht !',nl]),
     !.

aktionsbestimmung( AZ, Ueberfuehrung, Folgezustand, Objekt, ZielPosition) :-
      ( (  oberblock( Block, Objekt, AZ, _, 0) ,
        Block \== Objekt
      )  ;
      (
        oberblock( Block, ZielPosition, AZ, _, 0)),
        Block \== ZielPosition
      )  ,
      oberblock( Position, Folgezustand, AZ, _, 0),
      Ueberfuehrung = [ueberfuehrung( Block, Position),
                       ziel( Objekt,ZielPosition)],
      writeall( [5,'Aktion             : Ein Oberblock wird verlegt ',nl]).
```

```
/* -- > Ende aktionsbestimmung <-------------------------------------- */


/* ********************************************************************** */
/* ****** ENDE DER DEFINITION DER DOMÄNE `KLÖTZCHENWELT' *************** */
/* ********************************************************************** */



/* -- > writeall <---------- Hilfsfunktion -------------------------

   eff  : Gibt alle Elemente einer Liste aus
   call : writall( LISTE)
*/

write_level( 6).

writeall( []) :- !.

writeall( [A|X]) :-
      integer( A),
      write_level( WL),
      not( A > WL),
      writeall( X), !.

writeall( [A|_]) :-
      integer( A),
      write_level( WL),
      A > WL, !.


writeall( [nl|X]) :-
      nl,
      writeall( X), !.


writeall( [get|X]) :-
      get( _),
      writeall( X), !.

writeall( [A|X]) :-
      write( A),
      writeall( X) ,!.

/* --> Ende writeall<--------------------------------------------- */


/* -- > delel <---------- Hilfsfunktion -------------------------

   eff  : löscht aus einer Liste ein Element
   call : delel( LISTE, LISTE, ELEMENT)
*/

delel( [], [], _) :- !.

delel( [C|X], B, C) :-
      delel( X, B, C), !.

delel( [A|X], [A|Y], C) :-
      delel( X, Y, C), !.
```

```
/* -- > klammernweg <---------- Hilfsfunktion -------------------------

   eff  : löscht Klammern aus einer Liste
   call : klammernweg( LISTE, LISTE)
*/


kW( [A|_], RG) :-
  flatten( A, RG),
  A = RG.

kW( [A|_], RG) :-
  kW( A, RG).

kW( [_|B], RG) :-
  kW( B, RG).

klammernweg( Plaene, NeuePlaene) :-
  findall( NeuPlan, kW( Plaene, NeuPlan), NeuePlaene).


/* -------------------------------------------------------------- */



/* ---------------> P R O G R A M M E N D E <------------------- */
```

6.3 Ausgabebeispiel Test4

```
?- platz( test4, Z).


Ausgangszustand : [auf(a, p),auf(b, a),auf(c, q)]
Zielzustand     : [auf(a, b),auf(b, s),auf(c, r)]

Zuganzahl          : [(1, b, s),(1, c, r)]
Teilziel           : b nach s
Frei ist           : [q,r]
Aktion             : Ein Teilziel wird erreicht !
Aktion             : Ein Teilziel wird erreicht !
Teilziel           : c nach r
Frei ist           : [p,s]
Aktion             : Ein Teilziel wird erreicht !
Aktion             : Ein Teilziel wird erreicht !
Ueberfuehrungen    : [[ueberfuehrung(b, s),ziel(b, s)],[ueberfuehrung(c,
r),ziel(c, r)]]
Ueberfuehrung      : b nach s
Akt. Zustand       : [auf(a, p),auf(b, s),auf(c, q)]
Err. Teilziele     : [auf(b, s)]
Zuganzahl          : [(1, a, b),(1, c, r)]
Teilziel           : a nach b
Frei ist           : [q,r]
Aktion             : Ein Teilziel wird erreicht !
Aktion             : Ein Teilziel wird erreicht !
Teilziel           : c nach r
Frei ist           : [p,s]
Aktion             : Ein Teilziel wird erreicht !
Aktion             : Ein Teilziel wird erreicht !
Ueberfuehrungen    : [[ueberfuehrung(a, b),ziel(a, b)],[ueberfuehrung(c,
r),ziel(c, r)]]
Ueberfuehrung      : a nach b
Akt. Zustand       : [auf(a, b),auf(b, s),auf(c, q)]
Err. Teilziele     : [auf(a, b),auf(b, s)]
Zuganzahl          : [(1, c, r)]
Teilziel           : c nach r
Frei ist           : [p,s]
Aktion             : Ein Teilziel wird erreicht !
Aktion             : Ein Teilziel wird erreicht !
Ueberfuehrungen    : [[ueberfuehrung(c, r),ziel(c, r)]]
Ueberfuehrung      : c nach r
Akt. Zustand       : [auf(a, b),auf(b, s),auf(c, r)]
Err. Teilziele     : [auf(a, b),auf(b, s),auf(c, r)]

Ein Teilplan       : [b->s,a->b,c->r]


Ueberfuehrung      : c nach r
Akt. Zustand       : [auf(a, p),auf(b, s),auf(c, r)]
Err. Teilziele     : [auf(b, s),auf(c, r)]
Zuganzahl          : [(1, a, b)]
Teilziel           : a nach b
Frei ist           : [q,r]
Aktion             : Ein Teilziel wird erreicht !
Aktion             : Ein Teilziel wird erreicht !
Ueberfuehrungen    : [[ueberfuehrung(a, b),ziel(a, b)]]
Ueberfuehrung      : a nach b
Akt. Zustand       : [auf(a, b),auf(b, s),auf(c, r)]
Err. Teilziele     : [auf(a, b),auf(b, s),auf(c, r)]

Ein Teilplan       : [b->s,c->r,a->b]
```

```
Ueberfuehrung      : c nach r
Akt. Zustand       : [auf(a, p),auf(b, a),auf(c, r)]
Err. Teilziele     : [auf(c, r)]
Zuganzahl          : [(1, b, s)]
Teilziel           : b nach s
Frei ist           : [q,r]
Aktion             : Ein Teilziel wird erreicht !
Aktion             : Ein Teilziel wird erreicht !
Ueberfuehrungen    : [[ueberfuehrung(b, s),ziel(b, s)]]
Ueberfuehrung      : b nach s
Akt. Zustand       : [auf(a, p),auf(b, s),auf(c, r)]
Err. Teilziele     : [auf(b, s),auf(c, r)]
Zuganzahl          : [(1, a, b)]
Teilziel           : a nach b
Frei ist           : [q,r]
Aktion             : Ein Teilziel wird erreicht !
Aktion             : Ein Teilziel wird erreicht !
Ueberfuehrungen    : [[ueberfuehrung(a, b),ziel(a, b)]]
Ueberfuehrung      : a nach b
Akt. Zustand       : [auf(a, b),auf(b, s),auf(c, r)]
Err. Teilziele     : [auf(a, b),auf(b, s),auf(c, r)]

Ein Teilplan       : [c->r,b->s,a->b]


Ausgangszustand : [auf(a, p),auf(b, a),auf(c, q)]
Zielzustand     : [auf(a, b),auf(b, s),auf(c, r)]
                       PLATZ von H. Holst
                             und J. Schellenberger

Z = [['b->s','a->b','c->r'],['b->s','c->r','a->b'],['c->r','b->s','a->b']]
```

6.4 Literaturverzeichnis

[Chapman85]
Dave Chapman : Planning for Conjunctive goals
MIT TR 802, Cambrige (Aus : Reading in Plans / S. 537)

[Kluzniak75]
Kluzniak und Szapowicz : APIC Studies in Data Processing, No. 24 (Aus: RiP /S. 140)

[Waldiger75]
R. Waldinger : Achieving Several Gols Simultaneously
Artifical Intellegance Center / Standfort Research Center (Aus: RiP /S. 118)

[Sacerdoti75]
Earl D. Sacerdoti : The NONLIN Nature Of Plans (Aus: RiP /S. 162)

[Tate77]
Austin Tate : Gernerating Projekt Network (Aus : RiP /S. 291)

[Geibel92]
Peter Geibel : Planen für konjunktive Ziele

[Bratko88]
Ivan Bratko : Programming for Artifical Intelligence, 2. Edition

[Allen90]
James Allen, James Hendler, Austin Tate : Readings in Plans
1990 ISBN 1-55860-130-9