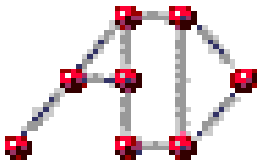
Studienarbeit

Implementation eines Planers zur symbolischen Exploration mit binären Entscheidungsdiagrammen

Malte Helmert



Studienarbeit
am Institut für Informatik
der Universität Freiburg

18.12.1999

Betreuer:

Prof. Dr. Th. Ottmann
Dr. St. Edelkamp

Inhaltsverzeichnis

1 Einleitung

1.1 Aufgabenstellung

Planung ist ein Kernbereich der Künstlichen Intelligenz und dient der Lösung von Problemen, die in einem allgemeinen, problemunabhängigen Formalismus gegeben sind. Neuere Erfolge [8, 15] in der symbolischen Exploration von Problemen in der Modellprüfung und von Einpersonenspielen legen die Verallgemeinerung des auf die Datenstruktur der binären Entscheidungsdiagramme aufbauenden Suchansatzes in der Planung nahe.

In dieser Studienarbeit wird die Konzeption und Implementation eines vollwertigen Planungssystems vorgestellt. Insbesondere werden die folgenden drei Phasen im Planungsprozess realisiert und dokumentiert:

- Einlesen und Parsen der Problemspezifikation aus einer Datei
- Inferenz einer effizienten binären Zustandskodierung aus der so gewonnenen Information
- Aufbau der Übergangsfunktion und symbolische Exploration

1.2 Gliederung

Im folgenden wird zunächst die Problemstellung angesprochen und motiviert, warum BDDs zur Lösung von Planungsproblemen sinnvoll sein können. Dazu müssen zunächst einige Schwierigkeiten überwunden werden; insbesondere muß die Eingabe für den Planer in eine kompakte und gut handhabbare binäre Kodierung überführt werden. Dieses Thema bildet den algorithmischen Kern der Arbeit. Abgeschlossen wird der algorithmische Teil durch die Anbindung des eigentlichen Explorationsalgorithmus und die Vorstellung eines Verfahren zur Extraktion der Lösung aus den erzeugten Zustandsmengen.

Anschließend werden einige der dabei beobachteten empirischen Ergebnisse diskutiert und zu den Fähigkeiten anderer Planungssysteme in Beziehung gesetzt sowie ein Ausblick auf mögliche Erweiterungen des hier vorgestellten Ansatzes geboten.

Alle hier vorgestellten Verfahren wurden im Rahmen dieser Studienarbeit in C++ implementiert. Der abschließende Teil beschreibt die interne Struktur des Planungssystems und geht auf dessen Bedienung ein.

1.3 Klassische Handlungsplanung

Das Problem der Handlungsplanung besteht grob gesagt darin, eine Folge von Operationen zu finden, die einen gegebenen Startzustand in einen bestimmten Endzustand überführen [16]. Der dabei verwendete Formalismus ist recht allgemein, so daß Planungsprobleme sehr verschiedenartig sein können. Klassische Beispiele sind Logistik- und Packprobleme, aber auch viele typische Suchprobleme können als Planungsprobleme ausgedrückt werden.

Abbildung 1 zeigt ein Beispiel für die Spezifikation eines Planungsproblems in der sogenannten *STRIPS*-Notation (für *STanford Research Institute Planning System*) [9]. Es handelt sich um das bekannte Problem der Türme von Hanoi, hier gegeben durch drei Scheiben `lit`, `med` und `big` und drei Stäben `l`, `m` und `r`.

```
(define (domain towers-of-hanoi)
  (:predicates (smaller ?d1 ?d2) (on ?d1 ?d2) (clear ?d))
  (:action move :parameters (?move ?from ?to)
      :precondition (and (smaller ?move ?to) (on ?move ?from)
                         (clear ?move) (clear ?to))
      :effect (and (clear ?from) (on ?move ?to)
                   (not (clear ?to)) (not (on ?move ?from)))))
(define (problem hanoi-3) (:domain towers-of-hanoi)
  (:objects l m r lit med big)
  (:init (smaller lit med) (smaller lit big) (smaller med big)
         (smaller lit l)   (smaller lit m)   (smaller lit r)
         (smaller med l)   (smaller med m)   (smaller med r)
         (smaller big l)   (smaller big m)   (smaller big r)
         (on big l)        (on med big)      (on lit med)
         (clear lit)       (clear m)         (clear r))
  (:goal (and (on lit med) (on med big)     (on big r))))
```

Abbildung 1: Die Türme von Hanoi als STRIPS-Files

Für eine formale Definition des Planungsproblems müssen nun zunächst einige Begriffe eingeführt werden:

- Ein *Objekt* ist ein eindeutiger Bezeichner, der für eine Entität in der modellierten Domäne steht, in obigem Problem etwa die drei Scheiben `lit`, `med` und `big`.

- Ein *Prädikat* ist ein eindeutiger Bezeichner, der zur Modellierung allgemeiner Eigenschaften dient, die Objekten zukommen können, etwa die Prädikate `smaller` und `clear` in obigem Beispiel. Jedem Prädikat p ist eine Stelligkeit $\sigma(p) \in \mathbb{N}_0$ zugeordnet. So ist z.B. $\sigma(\texttt{smaller}) = 2$, $\sigma(\texttt{clear}) = 1$.

- Ein n-stelliges Prädikat und ein n-Tupel von Objekten bilden zusammen einen *Fakt* (oder auch ein instantiiertes Prädikat). Ein Fakt modelliert eine Eigenschaft, die einem Objekt oder einem Tupel von Objekten zukommt. So wird durch den Fakt `(smaller, (med, big))` in obigem Beispiel ausgedrückt, daß die Scheibe `med` kleiner ist als die Scheibe `big`. Wir werden Fakten im folgenden nach dem Schema `smaller(med, big)` notieren.

- Eine Menge von Fakten bildet einen *Zustand* des Planungsproblems. Der momentane Zustand der Planungswelt ist durch die Menge der derzeit gültigen Fakten eindeutig beschrieben. Ein Beispiel ist der oben angegebene *Startzustand* `init`. Die *Endzustände* des Problems sind alle Zustände, die die in `goal` spezifizierten Fakten als Teilmenge enthalten.

- Ein *Operator* ist ein 3-Tupel (V, A, D) von *Vorbedingungen*, *Add-Effekten* und *Del-Effekten*, wobei V, A und D jeweils Mengen von Fakten sind. Ein Operator ist auf einem Zustand Z *anwendbar*, wenn die Vorbedingungen erfüllt sind, d.h. $V \subseteq Z$ gilt. Er überführt Z in den Zustand $Z' = (Z \cup A) \setminus D$, wobei gefordert wird, daß A und D disjunkt sind.

- Operatoren werden durch *Operator-Schemata* oder *Aktionen* definiert. So lassen sich etwa aus der obigen Aktion `move` durch Einsetzen beliebiger

Objekte für die freien Variablen `?move`, `?from` und `?to` z.B. die Operatoren `move(big, l, m)` und `move(lit, med, big)` erzeugen. Für letzteren Operator gilt beispielsweise $V =$ { `smaller(lit, big)`, `on(lit, med)`, `clear(lit)`, `clear(med)` }, $A =$ { `clear(med)`, `on(lit, big)` } und $D =$ { `clear(big)`, `on(lit, med)` }.

Mit diesen Begriffen läßt sich das Problem nun wie folgt formulieren: Gegeben seien eine Menge P von Prädikaten, eine Menge A von Aktionen (über der Prädikatsmenge), eine Menge O von Objekten, der Startzustand Z_0 sowie der Endzustand Z_e (jeweils Zustände über P und O). Gesucht ist eine Folge von Operatoren (gemäß den Schemata aus A), die den Zustand Z_0 in einen Zustand Z'_e mit $Z_e \subseteq Z'_e$ überführt, sofern eine solche Folge überhaupt existiert. Wenn keine solche Folge existiert, dann soll der Algorithmus dies feststellen und eine entsprechende Ausgabe erzeugen.

1.4 Handlungsplanung als Erfüllbarkeitsproblem

Im Laufe der Zeit hat es zum Teil sehr unterschiedliche Ansätze zur Lösung von Planungsproblemen gegeben [1, 6, 14]. Einer der klassischen Ansätze besteht darin, das Problem in ein Erfüllbarkeitsproblem zu transformieren und mit einem geeigneten Algorithmus eine erfüllende Belegung zu bestimmen, aus der sich dann wiederum die erforderlichen Operationen zur Lösung des Planungsproblems ableiten lassen [11]. Dabei soll die Gültigkeit des Fakts f nach t Zeitschritten durch die aussagenlogische Variable $x_{f,t}$ formalisiert werden. Im folgenden bezeichnet F die Menge aller Fakten, Z_0 den Startzustand, Z_e den Endzustand und Op die Menge aller Operatoren.

- Die Gültigkeit des Startzustands im ersten Zeitschritt wird durch folgende Konjunktion sichergestellt:

$$start = \bigwedge_{f \in Z_0} x_{f,0} \wedge \bigwedge_{f \in F \setminus Z_0} \neg x_{f,0}.$$

- Das Erreichen des Endzustands im Zeitschritt t formalisiert die Konjunktion

$$end(t) = \bigwedge_{f \in Z_e} x_{f,t}.$$

- Die Anwendung eines Operators $o = (V, A, D)$ zum Zeitpunkt t läßt sich formalisieren als

$$apply(o,t) = \bigwedge_{f \in V} x_{f,t} \wedge \bigwedge_{f \in A} x_{f,t+1} \wedge \bigwedge_{f \in D} \neg x_{f,t+1} \wedge \bigwedge_{f \in F \setminus (A \cup D)} x_{f,t} \leftrightarrow x_{f,t+1}.$$

Aus diesen Bestandteilen läßt sich dann das Gesamtproblem zusammensetzen:

$$Problem(n) = start \wedge end(n) \wedge \bigwedge_{0 \le t < n} \bigvee_{o \in Op} apply(o,t).$$

Kann man nun eine erfüllende Belegung für $Problem(n)$ finden, und sei n_0 das kleinste n, so daß dies möglich ist, so kann man aus den Variablen $x_{f,t}$ für $t \in \{0, 1, \ldots, n_0\}$ eine Folge von Zuständen extrahieren, die den Startzustand in

einen Endzustand transformieren. Für die so gefundene Zustandssequenz ist es nicht weiter schwierig, eine korrespondierende Operatorsequenz zu bestimmen, die die Lösung des Problems darstellt. Die so gefundene Lösung ist optimal in dem Sinne, daß sie aus einer minimalen Anzahl an Operatoren besteht.

Ein Problem des SAT-basierten Ansatzes besteht darin, daß sowohl die Formel selbst als auch die Anzahl der verwendeten aussagenlogischen Variablen bereits bei vergleichsweise einfachen Problemen sehr groß werden können. Dadurch, daß aus den Operatorschemata sehr viele gleichartige Operationen erzeugt werden, entsteht eine Vielzahl ähnlicher Teilformeln. Wünschenswert wäre demnach eine Repräsentation in einer Datenstruktur, die diese inhärenten Symmetrien ausnutzen kann.

1.5 Boolesche Funktionen und BDDs

Aus diesem Grunde liegt es nahe, sich im Zusammenhang mit Planungsproblemen mit *binären Entscheidungsdiagrammen* (binary decision diagrams, BDDs) zu beschäftigen. BDDs bieten eine Möglichkeit, boolesche Funktionen (speziell Funktionen des Typs $f : \mathbb{B}^n \to \mathbb{B}$) kompakt zu repräsentieren und in effizienter Weise wesentliche Operationen (etwa Konjunktion, Disjunktion oder Existenzquantifizierung bzgl. bestimmter Variablen) auf ihnen durchzuführen. Wie BDDs intern strukturiert sind und wie die auf ihnen operierenden Algorithmen funktionieren, wird in dieser Arbeit nicht diskutiert; BDDs werden hier als *black box* betrachtet und lediglich als Vehikel zur Implementation des Planungsalgorithmus benutzt [4].

Wir konzentrieren uns hier vielmehr auf die Frage, wie man BDDs benutzen kann, um eine Belegung der obigen booleschen Formel zu bestimmen. Zunächst lassen sich Zustandsmengen in einer Planungswelt ohne weiteres als boolesche Funktionen darstellen lassen, indem man sie durch ihre charakteristische Funktion repräsentiert, d.h. die boolesche Funktion, die auf einen gegebenen binär kodierten Zustand angewendet genau dann den Wert "Wahr" annimmt, wenn dieser Zustand in der jeweiligen Zustandsmenge liegt. Ebenso lassen sich Relationen über Zuständen, also Mengen von Tupeln von Zuständen, über ihre charakteristische Funktion repräsentieren.

Damit kann man nun die drei wesentlichen Bestandteile der obigen Formel: Startzustand, Endzustand und Übergang von einem Zeitschritt zum nächsten (die *Transitionsrelation*) als BDDs repräsentieren. Durch Berechnung des *relationalen Produkts* einer Zustandsmenge mit der Transitionsrelation kann man nun die Menge der Nachfolgezustände berechnen. Im Kontext der obigen Formel bedeutet das also: Aus der Menge der konsistenten Belegungen der Variablen $x_{f,t}$ (die gültigen Zustände zum Zeitpunkt t) wird die Menge der konsistenten Belegungen der Variablen $x_{f,t+1}$ (der gültigen Zustände zum Zeitpunkt $t+1$) berechnet. Sobald eine dieser Mengen einen Endzustand enthält, ist eine erfüllende Belegung gefunden. Diese muß anschließend noch aus den BDDs extrahiert werden; das dafür verwendete Verfahren wird später noch vorgestellt. Mit anderen Worten wird also eine Breitensuche über dem Zustandsraum durchgeführt. Dabei ist hervorzuheben, daß durch die Verwendung von BDDs der Übergang von einem Niveau zum nächsten in einem einzigen Schritt und nicht *Zustand für Zustand* geschieht.

1.6 Von STRIPS-Files zur Binärkodierung

Leider ist das Verfahren in der bislang beschriebenen Form noch zu einfach, um größere Planungsprobleme zu lösen. Kodiert man jeden Fakt in einer eigenen booleschen Variable (die sogenannte *naive Kodierung*), kommt man leicht auf BDDs mit fünfstelliger Variablenzahl und wird bei der Exploration unweigerlich an der Komplexität des Problems scheitern.

Außerdem ist nicht klar, wie man aus der Beschreibung des Planungsproblems in STRIPS-Notation zu einer Kodierung durch boolesche Variablen kommen kann und wie man die für den Aufbau der Transitionsrelation notwendigen Operatoren bestimmen kann (alle denkbaren Operatoren zu betrachten, ist aufgrund der kombinatorischen Explosion normalerweise nicht machbar, und der weitaus größte Teil von ihnen kann ohnehin aufgrund unerfüllbarer Vorbedingungen unberücksichtigt bleiben).

Diese offenen Fragen bilden das eigentliche Problem beim Planen mithilfe von binären Entscheidungsdiagrammen, und in den folgenden Abschnitten soll es darum gehen, dieses Problem zu lösen [7].

2 Die Algorithmen

2.1 Parsing der STRIPS-Files

Der erste Schritt bei der Verarbeitung des Problems besteht darin, die Eingabedateien einzulesen und in geeigneten Datenstrukturen abzulegen. In unserem Fall wurden als Eingabeformat PDDL-Dateien verwendet. Das PDDL-Format ist das heute allgemein akzeptierte Eingabeformat für Planungsprobleme. Es gibt verschiedene Ebenen der Ausdrucksmächtigkeit innerhalb von PDDL. Wir haben uns auf die Teilmenge von PDDL beschränkt, die im Wesentlichen dem STRIPS-Formalismus entspricht und daher auf Erweiterungen wie etwa negierte Vorbedingungen verzichtet, auch wenn einige dieser Erweiterungen problemlos in den vorliegenden Rahmen integriert werden könnten.

Probleme in PDDL werden immer durch zwei Dateien spezifiziert:

- Eine Domänen-Datei, die Prädikate und Aktionen definiert, sowie
- eine Problem-Datei, die Objekte, Start- und Endzustand definiert.

Diese Aufteilung ermöglicht es, für ähnliche Probleme (etwa Varianten der Türme von Hanoi mit unterschiedlich vielen Scheiben) dieselbe Domänen-Datei zu verwenden und nur die Problem-Datei zu variieren. Das Beispiel aus Abbildung 1 zeigt sowohl die Domänen- als auch die Problemdatei; letztere beginnt ab der Zeile `define (problem hanoi-3)`.

Eine erste Version des Planers verwendete die Programme *flex* und *bison* (Varianten der Unix-Programme *lex* und *yacc*), um die Daten einzulesen, aber dieser Ansatz wurde zugunsten einer manuellen Implementation verworfen, da sich zum einen PDDL-Files nur unzureichend durch kontextfreie Grammatiken beschreiben lassen und zum anderen die von *flex* und *bison* erzeugten Programmteile schlecht in eine objektorientierte Umgebung integriert werden können.

Die Arbeit in dieser Phase besteht darin, die Daten in einer geeigneten Struktur abzulegen. Dafür werden im Wesentlichen Vektoren verwendet, die von der *C++ Standard Template Library* (*STL*) zur Verfügung gestellt werden.

Anschließend kann auf alle Prädikate, Aktionen und Objekte über ihren Namen oder einen ganzzahligen Schlüssel zugegriffen werden. Auch den verschiedenen möglichen Fakten werden Schlüssel zugewiesen, so daß Zustände entweder als Listen von ganzen Zahlen gespeichert werden (was sich für Zustände mit wenigen Fakten empfiehlt) oder auch durch Bitvektoren repräsentiert werden können, in denen ein gesetztes Bit an Position i bedeutet, daß der Fakt mit dem Index i diesem Zustand angehört (man sagt: in diesem Zustand wahr ist). Diese Repräsentation ist bei Zuständen empfehlenswert, in denen sehr viele der möglichen Fakten auftreten.

Das eigentliche Parsing läßt sich problemlos in linearer Zeit in der Länge der Eingabe bewältigen. Der Aufbau der angesprochenen Datenstrukturen zum Zugriff auf die verschiedenen Entitäten über ihren Namen benötigt insgesamt Zeit $O(n \log n)$ (wobei n die Anzahl der dargestellten Entitäten bezeichnet), da balancierte Binärbäume verwendet werden. Die in dieser Phase aufgebauten Datenstrukturen reflektieren die Eingabe selbst und benötigen daher linearen Platz in der Größe der Eingabe.

2.2 Erkennen konstanter Prädikate

Nach diesen vorbereitenden Maßnahmen beginnt die eigentliche Verarbeitung der Daten mit der Erkennung konstanter Prädikate. Ein *konstantes Prädikat* ist ein Prädikat, das niemals in einer Effekt-Liste (A oder D) eines Operators auftritt. Derartige Prädikate sind in Planungsproblemem sehr häufig. Sie dienen zur Modellierung statischer Informationen, in obigem Beispiel etwa der Tatsache, daß die kleine Scheibe `lit` kleiner ist als die mittelgroße Scheibe `med`. Weiterhin werden konstante Prädikate oft als Ersatz für Typen verwendet, die es im reinen STRIPS-Formalismus nicht gibt. Beispiele dafür sind die Prädikate `PACKAGE`, `TRUCK` und `LOCATION` im Logistik-Beispiel aus Abbildung 2, das uns (gemeinsam mit der zugehörigen Problemdefinition aus Abbildung 3) eine Weile begleiten wird.

```
(define (domain easy-logistics)
  (:predicates (PACKAGE ?p) (TRUCK ?t) (LOCATION ?loc)
               (at ?x ?loc) (in ?p ?truck))
  (:action LOAD    :parameters (?p ?truck ?loc)
     :precondition (and (PACKAGE ?p) (TRUCK ?truck) (LOCATION ?loc)
                        (at ?truck ?loc) (at ?p ?loc))
     :effect       (and (not (at ?p ?loc)) (in ?p ?truck)))
  (:action UNLOAD  :parameters (?p ?truck ?loc)
     :precondition (and (PACKAGE ?p) (TRUCK ?truck) (LOCATION ?loc)
                        (at ?truck ?loc) (in ?p ?truck))
     :effect       (and (not (in ?p ?truck)) (at ?p ?loc)))
  (:action DRIVE   :parameters (?truck ?from ?to)
     :precondition (and (TRUCK ?truck) (LOCATION ?from) (LOCATION ?to)
                        (at ?truck ?from))
     :effect       (and (not (at ?truck ?from)) (at ?truck ?to))))
```

Abbildung 2: Eine einfache Logistik-Domäne

Da unser Ziel die Erzeugung einer effizienten Zustandskodierung ist, ist es wichtig, konstante Prädikate zu erkennen. Diese enthalten offenbar keine dyna-

```
(define (problem easy-logistics-10)
   (:domain easy-logistics)
   (:objects package1 ... package10 truck1 ... truck10 loc1 ... loc10)
   (:init (PACKAGE package1) ... (PACKAGE package10)
          (TRUCK truck1) ... (TRUCK truck10)
          (LOCATION loc1) ... (LOCATION loc10)
          (at truck1 loc1) ... (at truck10 loc10)
          (at package1 loc1) ... (at package10 loc10))
   (:goal (and (at package1 loc10) (at package2 loc10) ...
               (at package9 loc10) (at package10 loc10))))
```

Abbildung 3: Ein einfaches Logistik-Problem

mische Information und müssen daher nicht mitkodiert werden. Der Algorithmus zum Erkennen konstanter Prädikate ist sehr einfach: Wir betrachten für jede Aktion die Effektlisten und markieren alle dort auftretenden Prädikate. Alle am Ende noch unmarkierten Prädikate sind konstant.

Auf dieselbe Weise wäre es möglich, Prädikate zu erkennen, deren Fakten entweder nur gelöscht oder nur hinzugefügt werden können (sogenannte *Einweg-Prädikate*, z.B. das `fuelled`-Prädikat in der *rocket world*, denn hier können Raketen nicht mehr betankt werden, nachdem sie einmal ihren Sprit verbraucht haben). Wir verzichten darauf, diese Informationen zu nutzen, da eine spätere Phase, die Exploration des Faktraums, detailliertere Resultate derselben Art liefert.

Das Erkennen konstanter Prädikate erfordert es nur, jeden Effekt jeder Aktion einmal zu betrachten und ist damit in linearer Zeit in der Anzahl der Effekte zu bewältigen. Der zusätzliche Platzbedarf für die Speicherung der gewonnenen Informationen ist linear in der Anzahl der Aktionen.

2.3 Erkennen balancierter Prädikate

Eine *naive Kodierung* eines Zustands könnte darin bestehen, für jeden einzelnen Fakt mit einem Bit zu speichern, ob er in diesem Zustand wahr ist, oder nicht.

Betrachtet man das obige Logistik-Beispiel, so zählt man 30 Objekte (jeweils zehn Pakete, Lastwagen und Orte), drei einstellige Prädikate (`PACKAGE`, `TRUCK`, `LOCATION`) sowie zwei zweistellige Prädikate (`at`, `in`). Für jedes einstellige Prädikat wären 30 Bits zur Kodierung nötig, für jedes zweistellige Prädikat $30 \cdot 30 = 900$ Bits, insgesamt also $3 \cdot 30 + 2 \cdot 900 = 1890$ Bits, eine viel zu große Zahl, die auch durch Weglassen der konstanten Prädikate nur auf 1800 verringert werden kann.

Es geht wesentlich besser. Dem menschlichen Beobachter wird schnell auffallen, daß es z.B. keineswegs nötig ist, 30 Bits für die diversen Varianten von `at(package1, x)` zu verwenden: Wenn das Paket an einem bestimmten Ort ist, so kann es nicht gleichzeitig an einem anderen Ort sein. Es würde also reichen, einen Schlüssel zu kodieren, der den Aufenthaltsort des Pakets kennzeichnet, wofür $\lceil \log_2 30 \rceil = 5$ Bits ausreichend wären.

2.3.1 Balancierte Prädikate

Die Frage ist also, wie eine derartige Information aus der Beschreibung der Domäne automatisch abgeleitet werden kann. Leider ist dies nicht möglich, denn es handelt sich hierbei um eine Eigenschaft, die nicht nur von der Domäne, sondern auch vom Startzustand abhängt: Enthielte der Startzustand etwa sowohl die Fakten `at(package1, loc1)` und `at(package1, loc2)`, dann könnte man diese beiden Aufenthaltsorte nicht ohne weiteres in einer einzelnen Zahl kodieren.

Man könnte jedoch zumindest versuchen zu zeigen, daß die Aktionen in der Domäne garantieren, daß die *Zahl* der Aufenthaltsorte eines gegebenen Objekts niemals steigt, d.h. formaler, daß für jedes Objekt $o \in O$ gilt, daß $at_2(o, Z) := |\{(\texttt{at}\ o\ x) \in Z \mid x \in O\}|$ durch Anwendung eines beliebigen Operators im Zustand Z niemals steigt. In diesem Fall sprechen wir davon, daß `at` *balanciert im zweiten Parameter* ist. Allgemeiner nennen wir das n-stellige Prädikat `pred` *balanciert im k-ten Parameter*, wenn für beliebige Objekte o_i der Ausdruck $pred_k(o_1, \ldots, o_{k-1}, o_{k+1}, \ldots, o_n, Z)$ bei Anwendung eines Operators im Zustand Z niemals steigt, sich also $|\{(\texttt{pred}(o_1, \ldots o_{k-1}, x, o_{k+1}, \ldots o_n) \in Z \mid x \in O\}|$ beim Zustandsübergang nicht erhöht.

Was nützt das nun für die Kodierung? Falls bekannt ist, daß `at` im zweiten Parameter balanciert ist, dann muß für jedes Objekt o einer der folgenden drei Fälle gelten (Z_0 bezeichnet den Startzustand):

- $at_2(o, Z_0) = 0$: In diesem Fall muß über den Aufenthaltsort von o überhaupt keine Information gespeichert werden.
- $at_2(o, Z_0) = 1$: Dann kann o nur an maximal einem Ort auf einmal sein, d.h. wir kommen mit $\lceil \log_2(|O|+1) \rceil$ Bits aus (Die Addition der 1 dient dazu, auch den Fall repräsentieren zu können, daß das Objekt schließlich an *keinem* Ort mehr ist, denn die Zahl der Aufenthaltsorte darf ja durchaus sinken, nur eben niemals steigen).
- $at_2(o, Z_0) > 1$: In diesem Fall begnügen wir uns mit einer naiven Kodierung.

Es stellt sich nun die Frage, wie man im vorliegenden Beispiel zeigen kann, daß `at` balanciert ist. Betrachtet man die Aktionen genauer, stellt man fest, daß dies gar nicht der Fall ist, denn es gibt zwei Aktionen, die die Balance verändern: bei `LOAD`-Operatoren kann sie dekrementiert, bei `UNLOAD`-Operatoren inkrementiert werden. Es fällt jedoch auf, daß, wann immer $at_2(o)$ steigt, $in_2(o)$ entsprechend sinkt, und umgekehrt. Würde man also diese beiden Prädikate zu einem neuen Prädikat `at+in` zusammenfassen, so ergäbe sich ein balanciertes Prädikat, denn $(at + in)_2 = at_2 + in_2$ bleibt unter jedem Operator invariant, wie man leicht anhand der Definition der Aktionen nachvollziehen kann.

2.3.2 Der Algorithmus

Wir sind nun in der Lage, den Algorithmus zum Überprüfen der Balance von $pred_i$ für ein gegebenes Prädikat `pred` und einen Parameter $i \in \{1, \ldots, \sigma(pred)\}$ wie folgt zu beschreiben: Für jedes Aktionsschema und jeden Add-Effekt a überprüfen wir, ob dieser sich auf das Prädikat `pred` bezieht. Falls ja, suchen wir

einen zugehörigen Del-Effekt d, d.h. einen solchen Del-Effekt, der sich gleichfalls auf *pred* bezieht und dieselbe Argumentliste hat wie a, abgesehen vom i-ten Argument, das abweichend sein darf (und normalerweise sein wird). Finden wir zu jedem Add-Effekt einen passenden Del-Effekt, dann gleichen sich die Effekte gegenseitig aus und das Prädikat ist balanciert.

Gibt es jedoch einen Add-Effekt ohne Partner, dann durchsuchen wir die Liste der Del-Effekte nach einem beliebigen Effekt mit einer passenden Argumentliste - dieses Mal muß es sich dabei nicht um einen Effekt mit demselben Prädikat handeln. Gibt es keinen solchen Partner, dann kann keine Balance gefunden werden. Finden wir aber einen Partner, der sich auf das Prädikat `other` bezieht, dann rufen wir den Algorithmus rekursiv mit dem zusammengefaßten Prädikat `pred+other` auf. Der Ausdruck "passende Argumentliste" impliziert dabei *nicht*, daß `other` seine Argumente auch in derselben Reihenfolge nehmen muß wie `pred`, was den Algorithmus etwas verkompliziert, da gegebenenfalls verschiedene Permutationen der Argumentliste von `other` betrachtet werden müssen.

Da das i-te Argument von `pred` nicht in der Argumentliste des Partners auftauchen muß, ist es sogar möglich, ein Prädikat mit einem Prädikat zu vereinigen, dessen Stelligkeit um 1 geringer ist. Dies ist ein Spezialfall, den wir hier nicht näher ausführen möchten. Er tritt aber durchaus häufig auf, etwa in der *Gripper*-Domäne, wo ein Gripper entweder einen beliebigen Ball tragen kann (`carry ?ball ?gripper`) oder derzeit frei ist (`free ?gripper`), wobei immer eine der beiden Situationen zutreffen muß, d.h. `carry+free` ist balanciert.

Es kann beim Vereinigen von Prädikaten natürlich auch mehrere unterschiedliche Kandidaten geben. In diesem Fall werden alle sich ergebenden Möglichkeiten verfolgt. Obwohl die Rekursionstiefe theoretisch nur durch die Gesamtzahl der Prädikate beschränkt ist, ist uns kein praktisches Beispiel bekannt, indem es nötig ist, mehr als zwei Prädikate zu vereinigen, um Balance zu erreichen (sofern überhaupt Balance vorliegt).

Führt man diesen Algorithmus nun für alle Prädikate `pred` und alle möglichen i durch, so erhält man am Ende eine Menge von (ggfs. vereinigten) Prädikaten, die balanciert sind.

2.3.3 Beispiel und Komplexität

Im vorliegenden Beispiel ergibt sich das erwartete Resultat, daß `at+in` im zweiten Parameter balanciert ist. Betrachtet man den Startzustand, so stellt man mit obiger Fallunterscheidung fest, daß nur die Aufenthaltsorte der Pakete und Trucks gespeichert werden müssen. Für jedes Objekt gibt es 30 verschiedene Ausprägungen von `at`-Fakten und 30 verschiedene Ausprägungen von `in`-Fakten sowie den (in dieser Domäne allerdings unmöglichen) Fall, daß keiner dieser 60 Fakten gilt, also reichen insgesamt $\lceil \log_2 61 \rceil = 6$ Bits für jedes Paket und jeden Truck aus. Damit ergibt sich insgesamt eine Kodierungsgröße von $(10+10) \cdot 6 = 120$ Bits. Ein achtbares Resultat im Vergleich zu der vorherigen Größe von 1800 Bits, aber es geht noch kompakter, wie sich im folgenden Abschnitt herausstellen wird.

Vorher aber noch einige Worte zur Komplexität des vorgestellten Algorithmus. Bei ungünstigen Eingaben könnte theoretisch jede Teilmenge der Prädikatsmenge, abgesehen von konstanten Prädikaten, für eine Überprüfung in Frage kommen, weswegen exponentielle Zeit in der Zahl der Prädikate notwendig sein

kann.

Dies ist jedoch aus zwei Gründen unkritisch: zum einen tritt dieser ungünstige Fall bei allen betrachteten Benchmarkproblemen nicht auf – eine Rekursionstiefe von zwei wurde nie überschritten –, zum anderen läßt sich bei der geringen Zahl nicht-konstanter Prädikate, wie sie in Handlungsplanungsproblemen auftreten, ein exponentieller Aufwand durchaus akzeptieren.

Abgesehen von dem Platz zur Speicherung der ermittelten Informationen besteht kein nennenswerter zusätzlicher Speicherbedarf; er ist linear in der maximalen Anzahl an Add-Effekten der Aktionen der Domäne (für die Speicherung der Effekte, zu denen später ein Partner gefunden werden muß) zuzüglich der Anzahl der Prädikate (für die Rekursion).

2.4 Exploration des Faktraums

In den meisten Fällen ist es nicht notwendig, zum Kodieren der balancierten Prädikate den vollen Wertebereich zuzulassen. Beispielsweise kann ein Paket sich nur an einem Ort oder in einem Truck befinden, aber nicht in einem anderen Paket, in einem Ort usw.

Dazu eine Definition: Wenn ein Fakt im Startzustand enthalten ist oder es eine anwendbare Folge von Operatoren gibt, die diesen Fakt erzeugt, dann bezeichnen wir ihn als *erreichbar*, ansonsten als *unerreichbar*. Ein Operator heißt entsprechend *erreichbar*, wenn alle seine Vorbedingungen erreichbar sind.

Bei vielen Fakten kann man durch einfaches Betrachten der Operatorschemata auf die Unerreichbarkeit schließen. So kann ein `LOAD`-Operator Aktionen beispielsweise Objekte nur in Trucks verladen, wie man an den Vorbedingungen leicht erkennen kann. Diese Art der Analyse reicht aber bei komplexeren Beispielen nicht mehr aus.

Außerdem benötigen wir in einem späteren Stadium für die Konstruktion der Transitionsrelation eine Liste aller erreichbaren Operatoren.

2.4.1 Naive Exploration

Sowohl erreichbare Operatoren als auch erreichbare Fakten lassen sich über eine Breitensuche vom Startzustand ausgehend ermitteln: Zu Beginn des Algorithmus ist nur der Startzustand erreichbar. In einem Iterationsschritt werden alle möglichen Operatoren instantiiert. Diejenigen davon, deren Vorbedingungen erfüllt sind, werden als erreichbar markiert, ebenso die Fakten aus ihren Add-Effekt-Menge. Solange noch weitere Fakten erzeugt werden können, wird dieser Vorgang iteriert.

Der Algorithmus bricht ab, weil aufgrund der endlichen Zahl möglicher Fakten irgendwann ein Fixpunkt erreicht sein muß. Jedoch ist die naive Exploration sehr zeitaufwendig, denn die Zahl der theoretisch denkbaren Operatoren wächst bei großen Problemen sehr schnell, obwohl nur ein winziger Bruchteil der Operatoren tatsächlich erreichbar ist. Als Beispiel hierfür möge das Problem `Mprime-14` aus der AIPS'98-Suite[1] herangezogen werden, bei dem es ca. $3 \cdot 10^{13}$ theoretisch denkbare Operatoren gibt, von denen nur ca. 120.000 erreichbar sind.

[1] `http://ftp.cs.yale.edu/pub/mcdermott/aipscomp-results.html`

2.4.2 Fakt-basierte Exploration

Besser wäre daher ein Ansatz, der unnötige Instantiierungen von nicht erreichbaren Operatoren vermeidet. Dafür verwenden wir einen fakt-zentrierten Ansatz. Die zentrale Datenstruktur hierbei ist eine Schlange, in der die Fakten, die als nächstes als erreichbar markiert werden sollten, gespeichert werden. Zu Beginn enthält diese Schlange genau die Fakten des Startzustandes. Ein Verarbeitungsschritt besteht nun darin, den obersten Fakt aus dieser Schlange zu entnehmen, als erreichbar zu markieren und alle dadurch erreichbar gewordenen Operatoren als erreichbar zu markieren und zu instantiieren, d.h. die Fakten aus ihrer Add-Effekt-Menge in die Schlange aufzunehmen, falls sie bisher noch nicht als erreichbar markiert sind und auch noch nicht in der Schlange gespeichert sind. Dieser Vorgang wird wiederholt, bis die Schlange leer ist.

Die entscheidende Beobachtung besteht darin, daß es bei der Instantiierung der Operatoren ausreicht, solche Operatoren zu betrachten, in deren Vorbedingungen der neue Fakt auftritt, denn alle anderen zu diesem Zeitpunkt erreichbaren Operatoren müssen nach Konstruktion des Algorithmus schon vorher betrachtet worden sein. Dadurch wird kein Operator mehr als einmal betrachtet, denn jeder Operator, der instantiiert wird, enthält eine Vorbedingung, die im vorherigen Verarbeitungsschritt noch nicht erreichbar war und kann daher nicht bereits früher instantiiert worden sein.

Es bleibt die Frage, wie man in effizienter Weise die Menge der Operatoren bestimmen kann, die in einem bestimmten Verarbeitungsschritt betrachtet werden müssen. Zunächst ist klar, daß nur Operatoren in Frage kommen, die den aktuell betrachteten Fakt als Vorbedingung beinhalten. Operatorschemata, die das entsprechende Prädikat nicht enthalten, scheiden von vorneherein aus, und bei den in Frage kommenden Schemata ist es möglich, bestimmte Parameter von vorneherein an bestimmte Werte zu binden.

Dazu ein Beispiel: Betrachten wir das bekannte Logistik-Problem, der aktuelle Fakt sei `in(package4, truck2)`. Damit können die Schemata `LOAD` und `DRIVE` komplett ignoriert werden, denn in beiden tritt das Prädikat `in` nicht als Vorbedingung auf. Es bleibt nur noch das Schema `UNLOAD` mit den Parametern `?p`, `?truck` und `?loc`. Damit `in(package4, truck2)` eine Vorbedingung ist, muß `?p` = `package4` und `?truck` = `truck2` gelten, es bleibt also nur noch `?loc` als einzig frei wählbarer Parameter.

2.4.3 Optimierungen

Es gibt noch zwei weitere Optimierungen, die verwendet werden, um unnötige Instantiierungen zu vermeiden. Zum einen wird nach jeder Belegung eines Parameters überprüft, ob der Operator erreichbar sein kann, nicht erst, nachdem alle Parameter belegt wurden. Sollte also etwa bereits durch die Belegung `?p` = `package4`, `?truck` = `truck2` klar sein, daß eine der Vorbedingungen noch nicht erreichbar ist, kann auch der Operator nicht erreichbar sein, unabhängig davon, wie `?loc` belegt werden sollte, und die Suche nach einer Instantiierung mit einer solchen Belegung wird sofort abgebrochen.

Zweitens merkt sich der Algorithmus für jedes Prädikat `pred`, jede Position $i \in \{1, \ldots, \sigma(p)\}$ und jedes Objekt o, ob es bereits einen erreichbaren Fakt von `pred` gibt, dessen i-tes Objekt o ist, d.h. für jedes Prädikat werden alle *Projektionen* der Menge der erreichbaren Fakten auf die i-te Komponente gespeichert.

Damit kann der Bereich, der für die Belegung der noch offenen Parameter in Frage kommt, weiter eingeschränkt werden. In unserem Beispiel tritt der Parameter `?loc` bei den Vorbedingungen `LOCATION ?loc` und `at ?truck ?loc` auf. Wenn bisher die einzigen erreichbaren Fakten des Prädikats `LOCATION` die Fakten `LOCATION(loc1)`, `LOCATION(loc2)`, `LOCATION(loc3)`, `LOCATION(loc5)` und `LOCATION(loc7)` und die erreichbaren Fakten von `at` etwa `at(package1, loc1)`, `at(truck2, loc2)`, `at(truck3, loc3)` und `at(truck2, loc4)` sind, dann sind die entsprechenden Projektionen die beiden Mengen {`loc1`, `loc2`, `loc3`, `loc5`, `loc7`} und {`loc1`, `loc2`, `loc3`, `loc4`}, so daß nur Elemente aus deren Schnittmenge, also aus {`loc1`, `loc2`, `loc3`} als mögliche Belegungen für `?loc` in Frage kommen.

Die Projektionen können leicht über Bitvektoren in konstanter Zeit auf dem neuesten Stand gehalten und abgefragt werden. Der Schnitt zweier Projektionen läßt sich über bitweise Und-Verknüpfung zweier Bitleisten ebenfalls sehr schnell berechnen (in einer Zeit, die zwar linear in der Anzahl der Objekte ist, aber durch die Einfachheit der Operationen effektiv sehr niedrig ist).

Damit brauchen wir im vorliegenden Fall also nur noch drei verschiedene Belegungen zu betrachten. Trotz aller Optimierungen kommt man aber nicht umhin, auch hin und wieder bisher unerreichbare Operatoren zu konstruieren, im vorliegenden Fall etwa bei der Belegung `?loc` = `loc1`, da `at(truck2, loc1)` noch nicht erreichbar ist. Die Zahl der unnötig betrachteten Operatoren ist jedoch gegenüber dem nicht optimierten Algorithmus drastisch reduziert.

2.4.4 Analyse

Dennoch bleibt die Laufzeit dieser Phase kritisch, denn sie orientiert sich nicht an der Zahl der Prädikate oder an einer anderen Größe, die linear in der Größe der Eingabe ist, sondern an der Anzahl der Fakten, die exponentiell in der maximalen Stelligkeit der Prädikate ist.

Die Zahl der Fakten ist durch $|P| \cdot |O|^{\sigma_{\max}}$ nach oben beschränkt, wobei P die Menge der nicht-konstanten Prädikate, O die Objektmenge und $\sigma_{\max}$ die größte Stelligkeit aller Prädikate aus P bezeichnet.

Die Gesamtzahl der Fakten ist eine obere Schranke für die Anzahl der Fakten, die jemals in der verarbeitenden Schlange landen, und damit für die Anzahl der durchgeführten Schritte in dieser Phase. Die Laufzeit eines Schrittes kann jedoch sehr groß sein, wenn es komplizierte Aktionen mit vielen Parametern gibt, insbesondere wenn diese viele mehrstellige Prädikate als Vorbedingungen haben, so daß das Betrachten der Projektionen die Zahl der möglichen Instantiierungen nicht wirkungsvoll genug einschränken kann.

Dies ist zum Glück nur ein theoretisches Problem, denn wie bereits erwähnt gibt es Planungsprobleme mit ca. $3 \cdot 10^{13}$ theoretisch denkbaren Aktionsinstantiierungen, und auch bei diesen Problemgrößen kommt der Algorithmus noch in wenigen Sekunden zum Ziel.

Es ist jedoch problemlos möglich, Eingaben zu konstruieren, bei denen die Zahl der zu betrachtenden Instantiierungen nicht wirkungsvoll eingeschränkt werden kann, so daß die Zahl der zu betrachtenden Instantiierungen in einem einzelnen Schritt von der Größenordnung $\Theta(|A| \cdot |O|^{p_{\max}})$ ist, wobei A die Menge der Aktionen, O wiederum die Menge der Objekte und $p_{\max}$ die maximale Anzahl an Parametern einer Aktion ist, so daß die Gesamtlaufzeit als Produkt dieser Größe mit der Anzahl der Fakten von der Größenordnung $O(|P| \cdot |A| \cdot |O|^{\sigma_{\max}+p_{\max}})$

ist, was bei typischen Werten von $p_{\max}$ von ca. 5 schon schnell nicht mehr handhabbar wird.

Der Platzbedarf dieser Phase ist zunächst linear in der Zahl der erreichbaren Fakten und Operatoren, denn die Ergebnisse müssen gespeichert werden. Außerdem werden noch $|P| \cdot |O| \cdot \sigma_{\max}$ Bits für die Speicherung der Projektionen benötigt. Alle anderen Platzerfordernisse dieser Phase werden von diesen beiden Größen dominiert und können daher vernachlässigt werden. In der Praxis ist dieser Platzverbrauch tragbar.

Was ist nun in unserem Beispiel das Ergebnis der Exploration? Zum einen erhält man eine Liste aller anwendbaren Operatoren, die später noch benötigt wird. Zum anderen ermittelt der Algorithmus, daß Trucks nur an Orten, Pakete nur in Trucks oder an Orten befindlich sein können. Damit benötigt man pro Truck nur noch $\log_2(10+1) = 4$, pro Paket nur noch $\lceil \log_2(10+10+1) \rceil = 5$, insgesamt also 90 Bits gegenüber 120 Bits vor dem Explorationsschritt. Damit ergibt sich ein gutes Endergebnis, das auch mit einer Handkodierung konkurrieren kann.

2.5 Generierung der Zustandskodierung

Im vorliegenden Beispiel ist die Erzeugung der Zustandskodierung nach Exploration des Faktraums eine einfache Angelegenheit: Durch die Balance von `at+in` ist klar, daß die erreichbaren Fakten in Teilmengen sich gegenseitig ausschließenden Fakten (wie etwa `at(package1, loc1)` und `in(package1, truck1)`) zerfallen. Solche Teilmengen bezeichnen wir als *Faktgruppen*. Da die Fakten in einer Faktgruppe sich gegenseitig ausschließen, können sie gemeinsam mit $\lceil \log_2(|G|+1) \rceil$ Bits kodiert werden.

Auch bei anderen Problemen wird durch die Exploration des Faktraums zunächst die Menge der erreichbaren Fakten eingeschränkt, durch das Wissen über balancierte Prädikate lassen sich dann Faktgruppen bilden. Fakten, die in keine Faktgruppe fallen, weil die zugehörigen Prädikate in keiner Balance-Beziehung stehen, werden naiv mit 1 Bit pro Fakt kodiert (Ein Beispiel dafür ist das `locked`-Prädikat in der *Grid*-Domäne: Alle Türen können unabhängig voneinander offen oder geschlossen sein, es wird daher für jede Tür ein eigenes Bit benötigt).

Unklar ist jedoch, was zu tun ist, wenn die verschiedenen Faktgruppen nicht disjunkt sind. Ein Beispiel dafür findet sich in der Gripper-Domäne mit den drei Prädikaten `carry`, `free` und `at`. Das Prädikat `carry` gibt an, welcher Roboterarm gerade welchen Ball trägt, `free`, ob ein bestimmter Roboterarm gerade keinen Ball trägt, und `at`, in welchem Raum sich ein gegebener Ball gerade befindet. Es gelten folgende zwei Balancen:

- Ein Ball ist stets in genau einem Raum oder wird von genau einem Roboterarm getragen, d.h. `at+carry` ist balanciert.
- Ein Roboterarm trägt stets genau einen Ball oder ist frei, d.h. `carry+free` ist balanciert.

Es ergeben sich also hier Überschneidungen. So könnte eine Faktgruppe die Fakten {`carry(arm1, ball1)`, `carry(arm1, ball2)`, `free(arm1)`} enthalten, eine zweite Gruppe {`at(room1, ball1)`, `at(room2, ball1)`, `carry(arm1,`

`ball1)`, `carry(arm2, ball1)`} und eine dritte Gruppe {`at(room1, ball2)`, `at(room2, ball2)`, `carry(arm1, ball2)`, `carry(arm2, ball2)`}.

Kodiert man zunächst alle Informationen, die sich aus der Balance von `carry+free` ergeben, dann müssen bei den Gruppen aus `at+carry` nur noch die `at`-Fakten kodiert werden, da die anderen bereits kodiert wurden; bei einer umgekehrten Reihenfolge müßten von `carry+free` nur noch die `free`-Fakten kodiert werden.

Da nicht ohne Weiteres klar ist, welche Reihenfolge die beste ist – das hängt von der Anzahl der Räume, Bälle und Tragearme ab – und es außerdem in der Regel nur wenige in Frage kommende Alternativen gibt, geht der Kodierer so vor, daß er alle möglichen Reihenfolgen systematisch durchprobiert und schließlich diejenige wählt, bei der sich die kleinste Kodierung ergibt. Um den Vorgang etwas zu beschleunigen (was eigentlich nicht nötig ist, da dies keine zeitkritische Phase ist), merkt sich der Algorithmus immer das Minimum der Kodierungslängen der bisher verwirklichten Alternativen und bricht einen neuen Kodierungsversuch ab, sobald klar wird, daß dieses bisherige Optimum nicht mehr verbessert werden kann.

2.6 Aufbau der Transitionsrelation

Damit ist der Aufbau der Zustandskodierung abgeschlossen. In den abschließenden drei Phasen geht es nun darum, die eigentliche Suche vorzubereiten, durchzuführen und den gesuchten Plan aus den bei der Suche erzeugten Zustandsmengen zu extrahieren.

Wie bereits in der Einleitung kurz angeklungen, werden zur Exploration drei BDDs benötigt:

- Die Kodierung der aktuellen Zustandsmenge (zu Beginn also einfach eine Kodierung des Startzustands) als BDD,
- die Kodierung der Endzustandsmenge sowie
- die Kodierung der Transitionsrelation, also der Menge der Zustands-Tupel (Z, Z'), die der Anwendung von Operatoren entsprechen.

Nachdem eine Zustandskodierung bestimmt wurde, ist es trivial, die ersten beiden BDDs zu erzeugen. Das BDD der Transitionsrelation ist die Vereinigung (logische Oder-Verknüpfung) der Transitions-BDDs der einzelnen Operatoren. Durch die vorherige Exploration des Faktraums sind wir in der Lage, uns auf diejenigen Operatoren zu beschränken, die wir als erreichbar markiert haben, was für die praktische Durchführbarkeit des Algorithmus entscheidend ist.

Ein einzelner Operator $op = (P, A, D)$ entspricht der Relation

$$R_{op} = \{(Z, Z') | P \subseteq Z, A \subseteq Z', D \cap Z' = \emptyset, Z \setminus (A \cup D) = Z' \setminus (A \cup D)\}$$

Die charakteristische Funktion von R_{op} ist also durch den folgenden logischen Ausdruck festgelegt:

$$\begin{aligned} \forall f \in F \quad & (f \in P \rightarrow f \in Z) \wedge (f \in A \rightarrow f \in Z') \wedge (f \in D \rightarrow f \notin Z') \wedge \\ & (f \notin (A \cup D) \rightarrow (f \in Z \leftrightarrow f \in Z')). \end{aligned}$$

F bezeichnet dabei die Menge aller erreichbaren Fakten. Dieser logische Ausdruck läßt sich direkt in eine BDD-Darstellung transformieren.

Nachdem die BDDs der einzelnen Operatoren erzeugt wurden, müssen diese nur noch durch logische Oder-Verknüpfung zu einem Gesamt-BDD zusammengesetzt werden. Dazu werden zunächst jeweils zwei BDDs von einzelnen Operatoren verschmolzen, wodurch BDDs entstehen, die jeweils zwei Operatoren kodieren. Von diesen werden wiederum jeweils zwei verschmolzen, usf., bis am Ende nur noch ein einzelnes großes BDD übrigbleibt, das die komplette Transitionsrelation kodiert.

Diese Phase ist von der Laufzeit her durchaus kritisch. Die zur Konstruktion des BDDs zu einem einzelnen Operator benötigte Zeit ist linear in der Größe der Zustandskodierung. Die Zeit, die für die Oder-Verknüpfung zweier BDDs notwendig ist, ist theoretisch nur durch das Produkt der Größen (Knotenzahl) der beiden beteiligten BDDs beschränkt. Durch die dem Problem inhärente Struktur (sehr viele Operatoren stammen aus demselben Operatorschema) sinken Laufzeit und Platzbedarf bedeutend, ohne daß sich dieses Verhalten theoretisch quantifizieren ließe.

Auch über den Platzbedarf des resultierenden BDDs kann man nur aussagen, daß er theoretisch sehr groß werden kann (es gelten dieselben Abschätzungen wie für die Laufzeit), normalerweise aber nicht wird.

Die Variablenordnung im verwendeten BDD hat einen entscheidenden Einfluß auf Zeit- und Speicherbedarf; leider ist das Problem, eine optimale Variablenordnung zu bestimmen, aber selbst NP-schwierig [2] und damit, falls $P \neq NP$, nicht in polynomialer Zeit zu lösen. Bei großen Problemen kann der Algorithmus in dieser Phase an Laufzeit- oder Speicherplatzbeschränkungen scheitern. Wenn die Zahl der erreichbaren Operatoren in den fünfstelligen Bereich kommt, beginnt der Aufbau der Transitionsfunktion zu schwierig zu werden.

2.7 BDD-Exploration

Nach dem Aufbau der Transitionsrelation kann nun die eigentliche BDD-Exploration beginnen. Abbildung 4 zeigt den benutzten Algorithmus in Pseudocode.

```
function explore(start, end, transition: BDD): BDD array
  S[0] := start
  explored := 0
  i := 0
  while (S[i] & end) = 0 do
    S[i+1] := apply(S[i], transition)
    if S[i+1] is subset of Explored then
      output "no solution"
      return []
    explored := explored | S[i+1]
    i := i + 1
  return S
```

Abbildung 4: Der Algorithmus zur BDD-Exploration

Es wird also solange mittels `apply` (dem relationalen Produkt, einer Funktion aus dem BDD-Paket) von der Zustandsmenge `S[i]` zur Zustandsmen-

ge `S[i+1]` übergegangen, bis entweder ein Endzustand gefunden wurde, d.h. `current` und `end` nicht mehr disjunkte Mengen repräsentieren, oder aber der komplette Zustandsraum durchsucht wurde, d.h. die Menge `explored` einen Fixpunkt erreicht hat.

Als Ausgabe liefert der Algorithmus ein leeres Array, falls keine Lösung existiert und ansonsten die BDDs, die die Zustandsmengen auf den verschiedenen Niveaus repräsentieren.

Es wurde weiterhin auch eine bidirektionale Variante dieses Algorithmus implementiert, die hier aber nicht näher besprochen werden soll.

Diese Phase dominiert Zeit- und Platzbedarf des Planungssystems. Gleichzeitig kann man über ihren Aufwand leider am wenigsten aussagen, da die wesentliche Verarbeitung innerhalb des BDD-Pakets geschieht. Die Laufzeit der Exploration wird dominiert von der Berechung des relationalen Produkts, was ein NP-schwieriges Problem ist und dementsprechend in der verwendeten Implementation in ungünstigen Fällen exponentielle Zeit benötigt. Auch der Platzbedarf der BDDs kann exponentiell anwachsen, so daß diese Phase sowohl an Zeit- als auch an Speicherbeschränkungen scheitern kann.

2.8 Lösungsextraktion

Sollte die Exploration ergeben haben, daß ein Plan existiert, dann besteht die letzte zu erledigende Aufgabe nun darin, aus den ermittelten BDDs den gefundenen Plan zu extrahieren.

Dazu beginnt man damit, die letzte Zustandsmenge `S[n]` mit der Menge der Endzustände `goal` zu schneiden und einen beliebigen Zustand aus der Schnittmenge zu extrahieren (der Schnitt ist nicht leer, da der Algorithmus sonst nicht abgebrochen hätte). Damit hat man nun einen in n Schritten erreichbaren Endzustand Z_n ermittelt.

Über die inverse Transitionsrelation $T^{-1} = \{(Z', Z) | (Z, Z') \in T\}$ ermittelt man nun alle Zustände, von denen aus Z_n durch Anwendung eines einzelnen Operators erreicht werden kann. Offenbar muß mindestens einer dieser Zustände in $n-1$ Schritten erreichbar sein, also in der Schnittmenge mit `S[n-1]` liegen. Es wird ein beliebiger solcher Zustand ausgewählt und mit Z_{n-1} bezeichnet.

Das Verfahren wird iteriert, bis schließlich eine Folge $(Z_0, Z_1, \ldots, Z_{n-1}, Z_n)$ von Zuständen bestimmt wurde, so daß für alle $i \in \{0, \ldots, n-1\}$ ein Operator-BDD op_i existiert, so daß (Z_i, Z_{i+1}) in op_i kodiert ist.

Der letzte Schritt besteht nun darin, die richtigen Operatoren zu bestimmen, indem einfach der Reihe nach für jeden Zustandsübergang alle möglichen Operator-BDDs überprüft werden.

Schließlich wird der so erzeugte Plan ausgegeben.

In bezug auf die Komplexität ist diese Phase unkritisch. Für die Rückwartsschritte zur Extraktion der Zustandsfolge wird zwar wieder das relationale Produkt benötigt, aber da die Zustandsmenge jeweils nur ein Element hat, ist das hier nicht schwierig, der Aufwand liegt in $O(m \cdot t)$, wobei m die Lösungslänge, t die Größe (Anzahl der BDD-Knoten) der Transitionsrelation bezeichnet.

Wurde die Zustandsfolge erzeugt, muß man im schlechtesten Fall noch einmal für jeden Schritt in der Lösung alle erreichbaren Operatoren daraufhin untersuchen, ob sie für den jeweiligen Zustandsübergang in Frage kommen. Eine solche Anfrage benötigt lineare Zeit in der Größe der Zustandskodierung, so daß insgesamt für diesen Teil ein Zeitbedarf von $O(m \cdot |Op| \cdot c)$ notwendig ist, wobei

m wiederum die Länge der Lösung, Op die Anzahl der Operatoren und c die Größe der Zustandskodierung bezeichnet (c kann sehr grob durch die Anzahl der Fakten nach oben abgeschätzt werden).

Dies ist nicht zeitkritisch und gewiß auch nicht platzkritisch: Für die Zustandsfolge wird Platz benötigt, der linear in dem Produkt aus Kodierungsgröße und Lösungslänge ist.

3 Empirische Ergebnisse

Die klassische Handlungsplanung ist ein PSPACE-vollständiges Problem [5], es können also alle Probleme, die unter einem geeigneten Berechnungsmodell (etwa einer Mehrband-Turingmaschine) mit polynomialem Platzbedarf gelöst werden können, auf dieses Problem zurückgeführt werden. Daher ist es nicht überraschend, daß es nicht gelang, für die vorliegenden Algorithmen günstige obere Schranken für die Laufzeit anzugeben.

In der Tat sind Algorithmen, die mit Planungsproblemen umgehen, schwer zu analysieren, denn Worst-Case-Abschätzungen liefern in der Regel unrealistische Ergebnisse, denn durch ihre "natürliche Struktur" sind praktische Probleme in aller Regel wesentlich einfacher zu handhaben als eine theoretisch denkbare schlechteste Eingabe derselben Größe. Eine Analyse im average case ist ebenso wenig sinnvoll oder durchführbar, denn es ist unklar, über was für eine Menge von möglichen Eingaben hier gemittelt werden sollte.

Daher soll das Schwergewicht bei diesen Untersuchungen hier nicht bei theoretischen Resultaten liegen, sondern bei empirisch ermittelten Werten aus praktischen Problemen. Dies ist ohnehin die einzig sinnvolle Möglichkeit zum Vergleich verschiedener Planungsalgorithmen, da auch über die Performance anderer Planungssysteme kaum verwertbare theoretische Resultate vorliegen.

3.1 AIPScomp'98

Im Rahmen der AIPS'98 fand ein Wettbewerb für Programme zur klassischen Handlungsplanung, genannt *AIPScomp'98*, statt. Wir haben die dabei verwendeten STRIPS-Probleme zur Grundlage unserer Experimente gemacht.

Dabei ergab sich, daß bei keinem der 155 Probleme die Phasen vor dem Aufbau der Transitionsfunktion zeitlich problematisch waren. Im schlechtesten Fall wurden 8 Sekunden (auf einer Sun Ultra Sparc Station) verbraucht, nur 7 der 155 Probleme benötigten überhaupt mehr als 1 Sekunde für diese Phase.

Der Aufbau der Transitionsfunktion und die eigentliche BDD-Exploration waren dann in vielen Fällen aber nicht in akzeptabler Zeit zu bewältigen. Bei einer Laufzeitbeschränkung von zehn Minuten waren wir in der Lage, 30 der 30 Probleme in der Domäne *Movie*, 20 der 20 Probleme in der Domäne *Gripper*, 4 der 35 Probleme in der Domäne *Logistics*, 10 der 35 Probleme in der Domäne *Mprime*, 9 der 30 Probleme in der Domäne *Mystery* und 1 der 5 Probleme in der Domäne *Grid* zu lösen. Dies gibt eine Gesamtzahl von 74 gelösten Problemen. Zum Vergleich: die Teilnehmer am AIPS'98-Wettbewerb lösten zwischen 71 und 91 Problemen. Dabei gelang es nur dem Planungssystem *HSP* [3], das einen heuristischen Suchansatz verfolgt, mehr als 72 Probleme zu lösen. Allerdings liefert *HSP* vergleichsweise lange Lösungen, im Gegensatz zum vorliegenden

Problem	Plan-länge	Zeit (in sec)	Kodie-rung	Fakten (zu kodieren)	Operatoren (erreichbar)
Movie 1-28	7	0,22	7 Bits	160 (7)	809 (162)
Movie 1-29	7	0,23	7 Bits	165 (7)	834 (167)
Movie 1-30	7	0,23	7 Bits	170 (7)	859 (172)
Gripper 1-18	113	32,39	79 Bits	3.738 (156)	149.940 (460)
Gripper 1-19	119	262,18	83 Bits	4.092 (164)	172.304 (484)
Gripper 1-20	125	418,95	87 Bits	4.462 (172)	196.788 (508)
Logistics 1-01	26	343,70	42 Bits	3.264 (144)	1.212.416 (727)
Logistics 1-05	22	65,11	35 Bits	5.805 (151)	3.816.336 (699)
Logistics 2-02	20	43,99	28 Bits	1.449 (80)	240.786 (341)
Mystery 1-01	5	0,39	28 Bits	3.192 (58)	12.252.303 (269)
Mystery 1-07	∞ †	2,86	82 Bits	12.558 (181)	392.073.696 (521)
Mystery 1-27	5	9,49	63 Bits	7.788 (152)	117.406.179 (2.280)
MPrime 1-07	5	74,94	126 Bits	12.558 (352)	$\approx 231 \cdot 10^9$ (3.291)
MPrime 1-11	7	15,79	61 Bits	4.862 (131)	$\approx 8 \cdot 10^9$ (3.189)
MPrime 1-28	7	4,57	41 Bits	4.152 (90)	$\approx 5 \cdot 10^9$ (2.544)
Grid 2-01	14	23,01	67 Bits	6.043 (276)	2.144.340 (4.295)

Tabelle 1: Experimentelle Ergebnisse

System, das aufgrund des Breitensuche-Ansatzes immer einen kürzestmöglichen Plan findet.

Tabelle 1 zeigt für einige charakteristische Probleme die Länge der Lösung, die zu deren Auffindung benötigte Zeit, die Größe der erzeugten Zustandskodierung, die Gesamtzahl aller Fakten, die Zahl der zu kodierenden Fakten, die Gesamtzahl aller Operatoren und die Zahl der als erreichbar markierten Operatoren.

Die Abbildungen 5, 6, 7, 8 und 9 zeigen den Zusammenhang zwischen Größe der Zustandskodierung und Handhabbarkeit des Problems durch das Planungssystem (die wenig interessante Domäne *Movie* wurde nicht berücksichtigt).

3.2 Bewertung und Ausblick

Zunächst einmal kann festgehalten werden, daß es möglich ist, mit symbolischen Explorationstechniken auf Basis von BDDs ein Planungssystem zu implementieren, das von seiner Leistungsfähigkeit dem heutigen Stand der Technik entspricht. Die Erzeugung der Zustandskodierung läßt sich effizient genug verwirklichen, um auch bei extrem schwierigen Problemen noch handhabbar zu bleiben. Wenn der Planer scheitert, dann liegt es meistens an der BDD-Explorationsphase selbst.

Daher wäre es sinnvoll, hier über mögliche Verbesserungen nachzudenken, für die es noch viel Raum gibt. Eine einfache Breitensuche kann nicht das Maß aller Dinge sein. Bereits der Übergang von der unidirektionalen zur bidirektionalen Suche liefert bereits bei vielen Domänen eine spürbare Verbesserung. Als weitere Optimierung könnte man eine Vereinfachung der Suchfront anstreben, etwa durch *forward set simplification* oder Anwendung des *Restrict-*

†Beim Problem `Mystery 1-07` konnte das Planungssystem beweisen, daß kein Plan zur Lösung des Problems existiert.

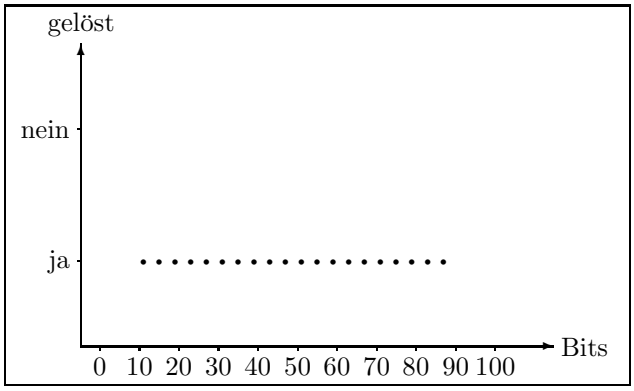


Abbildung 5: Kodierungsgröße vs. Lösbarkeit in *Gripper*

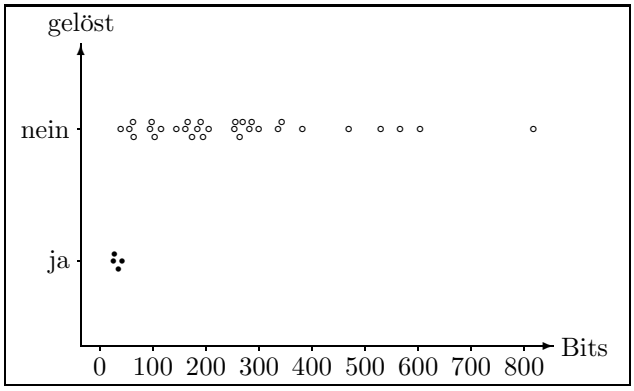


Abbildung 6: Kodierungsgröße vs. Lösbarkeit in *Logistics*

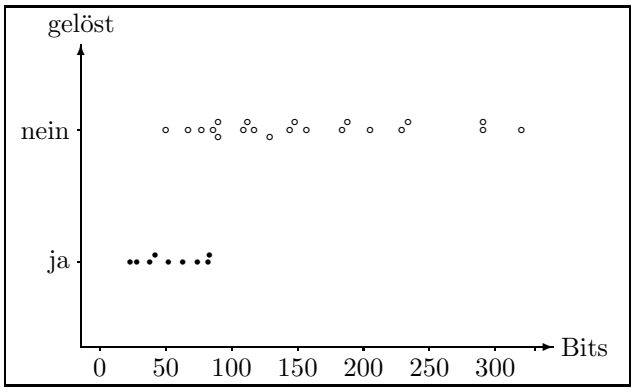


Abbildung 7: Kodierungsgröße vs. Lösbarkeit in *Mystery*

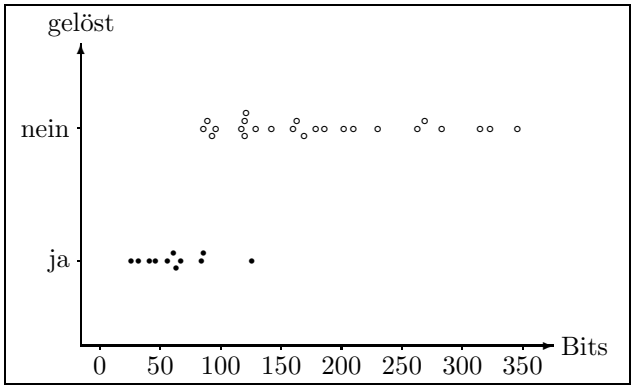


Abbildung 8: Kodierungsgröße vs. Lösbarkeit in *Mystery Prime*

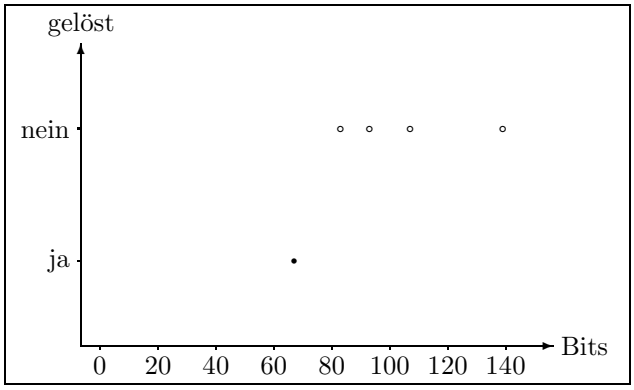


Abbildung 9: Kodierungsgröße vs. Lösbarkeit in *Grid*

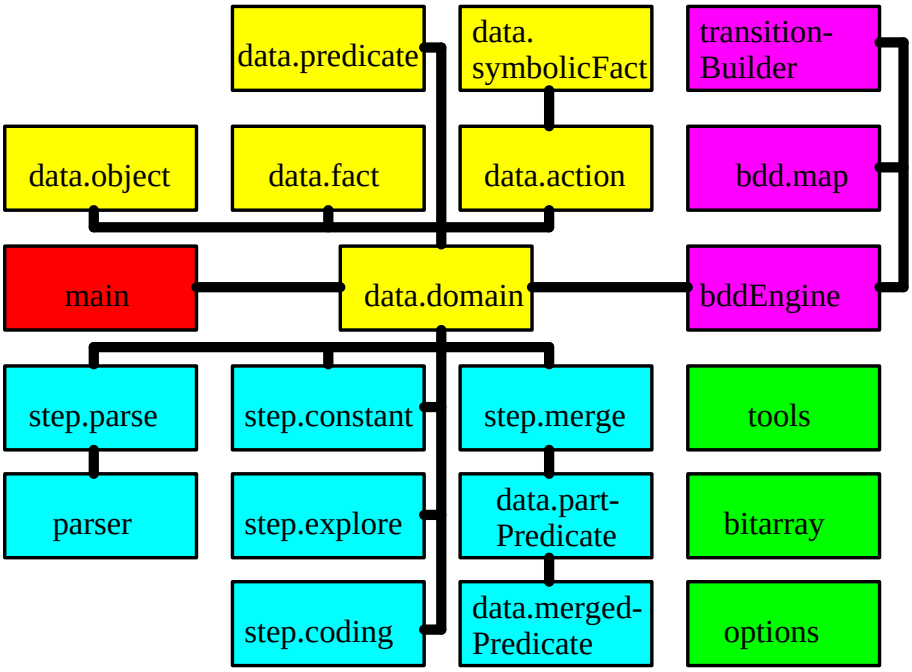


Abbildung 10: Die Architektur des Planungssystems

Operators. Um die Bildung des relationalen Produkts zu vereinfachen, könnte man auch die Transitionsfunktion in mehrere Teile aufspalten, für die dann separate Übergänge vorgenommen werden können, anstatt wie im hier beschriebenen Ansatz alle Operatoren in einem einzelnen BDD zu vereinigen.

Schließlich liegt es nahe, eine Kombination des symbolischen Suchansatzes mit heuristischen Strategien anzustreben, wofür etwa die Algorithmen *BDDA** oder *Pure BDDA** verwendet werden können [8]. Experimente haben gezeigt, daß diese Algorithmen einen weiteren großen Schritt nach vorne bringen können. Insbesondere die Ergebnisse bei der Domäne *Logistics* konnten massiv verbessert werden – wenn auch teilweise auf Kosten der Optimalität der Lösung.

Auch andere Erweiterungen und Verbesserungen wären denkbar, etwa die Erkennung und Behandlung von Symmetrien in der Eingabe oder das Eliminieren unnötiger Objekte und Operatoren, wie andere Planungssysteme (etwa *STAN* [10] und *IPP* [12]) sie teilweise vornehmen, um die Suchphase zu erleichtern – es gibt hier ein großes Potential, das bislang kaum ausgeschöpft ist. Auch eine Ausweitung auf einen ausdrucksmächtigeren Eingabeformalismus, etwa *ADL* (*Abstract Description Language*)[13] mit negierten Vorbedingungen, bedingten Effekten und quantifizierten Effekten, ist denkbar.

4 Der Planer

4.1 Architektur des Planungssystems

Abbildung 10 stellt die grobe Architektur des Planungssystems dar.

Die einzelnen Teile haben dabei folgende Funktionen:

- `main.cc`: Das Hauptprogramm. Es ist dafür verantwortlich, die Komman-

dozeile auszulesen, das Planungssystem zu starten und bei Bedarf eine Fehlermeldung auszugeben.

- `data.domain.cc`: Das Herzstück des Planungssystems. Von hier aus werden die Datenstrukturen des Planungssystems verwaltet und die einzelnen Phasen nacheinander aufgerufen.
- `data.object.cc`: Verantwortlich für die Behandlung von Objekten des Planungsproblems, z.B. Routinen zur Umwandlung zur Ermittlung des Namens eines Objekts aus seiner internen Kodierung.
- `data.predicate.cc`: Verantwortlich für die Behandlung von Prädikaten des Planungsproblems. In der zugehörigen Klasse `Predicate` werden z.B. die Projektionen der Prädikate verwaltet.
- `data.fact.cc`: Verantwortlich für die Behandlung von Fakten des Planungsproblems. Die Klasse `Fact` enthält beispielsweise Methoden zur Umwandlung von Fakten von textueller Darstellung in einen ganzzahligen Code und zurück.
- `data.action.cc`: Verantwortlich für die Behandlung von Aktionen des Planungsproblems. In der Klasse `Action` wird auch über die erreichbaren Operator-Instantiierungen Buch geführt.
- `data.symbolicFact.cc`: Wird vor allem von der Klasse `Action` benutzt, um sog. *symbolische*, d.h. parametrisierte Fakten zu verwalten, wie sie in den Vorbedingungs- und Effektlisten der Operatorschema auftauchen.
- `step.parse.cc`: Hier werden die Eingabedateien eingelesen und die Datenstrukturen entsprechend aufgebaut. Dafür wird `parser.cc`, ein einfacher Parser für *LISP*-Dateien, verwendet.
- `step.constant.cc`: Hier werden konstante Prädikate erkannt.
- `step.merge.cc`: Hier werden balancierte Prädikate ermittelt und Prädikate vereinigt, falls das für das Sicherstellen der Balance notwendig ist. Die Dateien `data.mergedPredicate.cc` und `data.partPredicate.cc` enthalten alle Methoden zur Verwaltung vereinigter Prädikate.
- `step.explore.cc`: Hier ist der im Abschnitt über die Exploration des Faktraums beschriebene Algorithmus implementiert.
- `step.coding.cc`: Hier ist der im Abschnitt über die Erzeugung der Zustandskodierung beschriebene Algorithmus implementiert.
- `bddEngine.cc`: Dies ist der zentrale Teil derjenigen Komponente des Systems, die sich mit BDDs befaßt. Hier wird insbesondere die BDD-Exploration durchgeführt. Die Erzeugung der Transitionsrelation wird an die Datei `transitionBuilder.cc`, die Extraktion des Plans teilweise an die Datei `bdd.map.cc` delegiert.
- `tools.cc`: Einige Hilfsroutinen, die an verschiedenen Stellen im Programm benötigt werden (Abhängigkeiten der Dateien des Moduls “Utility” sind im Schaubild nicht explizit angegeben). Im einzelnen sind dies Routinen zur Fehlerbehandlung, Zeitmessung, Verwaltung von Tupeln und Permutationen sowie einige mathematische Routinen.

- `bitarray.cc`: Eine Klasse zur Verwaltung von Bitvektoren.
- `option.cc`: Dient der Verwaltung der Kommandozeilenoptionen.

4.2 Verwendung und Aufrufparameter

Der Befehl zum Starten des Planungssystems ist

`planer <Optionen> <Domänendatei> <Problemdatei>`.

Jeder der drei Parameter kann auch weggelassen werden. Wird nur eine Datei angegeben, dann wird diese als Problemdatei betrachtet und als Domänendatei die Datei `domain.pddl` im aktuellen Verzeichnis verwendet. Wird gar keine Datei angegeben, dann wird die Problemdatei `problem.pddl` und die Domänendatei `domain.pddl` verwendet.

Gültige Optionen sind:

- `-?`, `-h` (*help*): Zeigt eine kurze Anleitung an. Das Planungssystem wird nicht gestartet.
- `-p` (*preprocess*): Es werden nur die Schritte bis zur Erzeugen der Zustandskodierung durchgeführt. Danach wird das Programm beendet.
- `-t` (*transition*): Es werden nur das preprocessing und der Aufbau der Transitionsfunktion ausgeführt, aber nicht exploriert.
- `-u` (*unidirectional*): Die BDD-Exploration wird unidirektional (wie beschrieben) durchgeführt. Standardmäßig wird bidirektional exploriert.
- `-s` (*silent mode*): Veranlaßt das Planungssystem, weniger Ausgaben zu tätigen.
- `-n` (*normal mode*): Veranlaßt das Programm, die normalen Ausgaben auszugeben. Ist nur in Verbindung mit spezialisierten Versionen von `-s`, `-v` oder `-d` sinnvoll (siehe weiter unten).
- `-v` (*verbose output*): Veranlaßt das Programm, umfangreichere Ausgaben zu tätigen. Beispielsweise wird in jedem BDD-Explorationsschritt die verbrauchte Zeit und die Zahl der im BDD kodierten Zustände ausgegeben.
- `-d` (*debug output*): Veranlaßt das Programm, die maximale Menge an Informationen auszugeben. Beispielsweise wird bei der Exploration des Faktraums der Status der Schlange protokolliert.

Den letzten vier Optionen können beliebige Zeichen aus `pcmeobts` angehängt werden, um die Wirkung der Option auf bestimmte Phasen zu beschränken (wird darauf verzichtet, wirkt die Option auf alle Phasen).

Dabei steht `p` für *parsing*, `c` für *constant predicates*, `m` für *merging*, `e` für *exploring (fact space)*, `o` für *coding*, `b` für *BDD package handling*, `t` für *transition function creation* und `s` für *BDD search*, also die Phasen in der Reihenfolge, in der sie hier besprochen wurden (abgesehen von dem zusätzlichen Schalter `b` und davon, daß die Planerzeugung nicht beeinflußt werden kann – hier werden unabhängig vom Modus immer dieselben Ausgaben getätigt).

Beispiel: `planer -u -s -vo gripper.pddl gripper.prob.10.pddl` startet das Planungssystem mit der Domänendatei `gripper.pddl` und der Problemdatei `gripper.prob.10.pddl`. Es wird unidirektional gesucht (`-u`), es werden

nur minimale Ausgaben getätigt (`-s`), abgesehen von der Codierungsphase, bei der erweiterte Ausgaben getätigt werden (`-vo`).

Literatur

[1] A. Blum and M. Furst. Fast planning through planning graph analysis. *Artificial Intelligence*, 90(1–2):281–300, 1997.

[2] B. Bollig and I. Wegener. Improving the variable ordering of OBDDs is NP-complete. *IEEE Transactions on Computers*, 45(9):993–1002, 1996.

[3] B. Bonet, G. Loerincs, and H. Geffner. A robust and fast action selection mechanism for planning. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI-97)*, pages 714–719. AAAI Press, 1997.

[4] R. E. Bryant. Symbolic manipulation of boolean functions using a graphical representation. In H. Ofek and L. A. O'Neill, editors, *Proceedings of the 22nd ACM/IEEE Conference on Design Automation (DAC 1985)*, pages 688–694, 1985.

[5] T. Bylander. The computational complexity of propositional STRIPS planning. *Artificial Intelligence*, 69(1–2):165–204, 1994.

[6] J. G. Carbonell, J. Blythe, O. Etzioni, Y. Gil, R. Joseph, D. Kahn, C. Knoblock, S. Minton, A. Pérez, S. Reilly, M. Veloso, and X. Wang. Prodigy 4.0: The manual and tutorial. Technical Report CMU-CS-92-150, Computer Science Department, Carnegie-Mellon University, 1992.

[7] S. Edelkamp and M. Helmert. Exhibiting knowledge in planning problems to minimize state encoding length. In M. Fox and S. Biundo, editors, *Recent Advances in AI Planning. 5th European Conference on Planning (ECP'99)*, volume 1809 of *Lecture Notes in Artificial Intelligence*, pages 135–147, New York, 1999. Springer-Verlag.

[8] S. Edelkamp and F. Reffel. OBDDs in heuristic search. In O. Herzog and A. Günter, editors, *Proceedings of the 22nd Annual German Conference on Artificial Intelligence (KI 1998)*, volume 1504 of *Lecture Notes in Computer Science*, pages 81–92. Springer-Verlag, 1998.

[9] R. E. Fikes and N. J. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2:189–208, 1971.

[10] M. Fox and D. Long. The automatic inference of state invariants in TIM. *Journal of Artificial Intelligence Research*, 9:367–421, 1998.

[11] H. Kautz and B. Selman. Pushing the envelope: Planning, propositional logic, and stochastic search. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI-96)*, pages 1194–1201. AAAI Press, 1996.

[12] J. Koehler, B. Nebel, J. Hoffmann, and Y. Dimopoulos. Extending planning graphs to an ADL subset. In S. Steel and R. Alami, editors, *Recent Advances in AI Planning. 4th European Conference on Planning (ECP'97)*, volume 1348 of *Lecture Notes in Artificial Intelligence*, pages 273–285, New York, 1997. Springer-Verlag.

[13] E. P. D. Pednault. ADL: Exploring the middle ground between STRIPS and the situation calculus. In R. J. Brachman, H. J. Levesque, and R. Reiter, editors, *Proceedings of the First International Conference on Principles of Knowledge Representation and Reasoning (KR'89)*, pages 324–332. Morgan Kaufmann, 1989.

[14] J. S. Penberthy and D. S. Weld. UCPOP: A sound, complete, partial order planner for ADL. In B. Nebel, C. Rich, and W. Swartout, editors, *Proceedings of the Third International Conference on Principles of Knowledge Representation and Reasoning (KR'92)*, pages 103–114. Morgan Kaufmann, 1992.

[15] F. Reffel and S. Edelkamp. Error detection with directed symbolic model checking. In J. M. Wing, J. Woodcock, and J. Davies, editors, *Proceedings of the World Congress on Formal Methods in the Development of Computing Systems (FM 1999)*, volume 1708 of *Lecture Notes in Computer Science*, pages 195–211. Springer-Verlag, 1999.

[16] S. Russell and P. Norvig. *Artificial Intelligence — A Modern Approach*. Prentice Hall, 1995.