Seminar AI Tools

SS 2007

Thema:
HTN Planning in JSHOP und JSHOP2

Sandro Castronovo
Gärtnerstr. 32
66117 Saarbrücken
sandro.castronovo@dfki.de
Matrikenummer: 2032823

Betreuer: Dr. Michael Kipp
28. April 2007

Zusammenfassung

Planungssysteme finden immer häufiger Anwendung in KI Applikationen. Die Erfahrung hat jedoch gezeigt, dass die traditionellen Planungsstrategien wie partial-order-planning oder backward-chaining in einigen praktischen Anwendungen nicht unbedingt die beste Wahl sind. Diese Arbeit beschreibt eine neue Planungsstrategie, das Hierarchical Task-Network (HTN). Im Gegensatz zu den traditionellen Planungsalgorithmen werden in einem HTN Aufgaben in der selben Reihenfolge geplant, in der sie auch später ausgeführt werden. Nach einer theoretischen Einleitung in das Planen mit HTNs zeigt diese Arbeit praktische Implementierungen eines HTN-Planers in SHOP[1] und dessen Nachfolger SHOP2 sowie einen direkten Vergleich der beiden Planer in zwei Beispieldomänen.

[1] Simple Hierarchical Ordered Planer

Inhaltsverzeichnis

1 Einleitung

Eine der neueren Planungsstrategien ist das das Planen mit Hierarchical Task Networks (HTNs). Dieser Ansatz verfolgt eine sog. total-order-Strategie, d.h. die einzelnen Schritte werden so geplant, wie sie später auch ausgeführt werden. Dies steht im Gegensatz zu den traditionellen Planungsstrategien wie z.B. partial-order-planning. Kapitel 2 stellt die genauen Unterschiede dar.
Ein HTN-Planer unterteilt komplexe Aufgaben in kleinere Aufgaben, bis primitive, direkt ausführbare tasks erreicht sind. Der Planer greift hierbei auf eine Domänenspezifikation zurück, die für jede Domäne neu geschrieben werden muss. Zudem können bestimmte Unterteilungen von Aufgaben auch gewissen Vorbedingungen unterworfen sein. Dies bedeutet zunächst einmal einen Mehraufwand, relativiert sich aber durch die Effizienz des Planens, denn ein HTN-Planer durchläuft weniger unnötige Pfade im Planungsbaum als traditionellen Planer. Je nach verwendetem Planungssystem unterscheidet sich auch der Aufwand, eine gute Domänenspezifikation zu geben.
Diese Arbeit gibt zunächst einen theoretischen Einstieg in HTNs und deren grundlegende Vorgehensweise, bevor eine konkrete Implementierung in SHOP und dessen Nachfolger SHOP2 vorgestellt wird. Anhand einer Beispieldomäne wird ein exemplarisches Planungsproblem besprochen und soll dem Leser ein Gefühl für die Vorgehensweise von SHOP bzw. von SHOP2 geben. Der Planer hat die Aufgabe von einem Punkt A zu einem Punkt B innerhalb einer Stadt zu gelangen. Es stehen verschiedene Verkehrsmittel zur Verfügung, die nur unter bestimmten Voraussetzungen benutzt werden können.
Ein Vergleich von SHOP und SHOP2 zeigt die Unterschiede in der Planungsgeschwindigkeit und gefundenen Plangrößen in der Blockweltdomäne und der Logistikdomäne.

1.1 HTN-Planer

Die grundlegenden Ideen für ein Planen mit HTNs wurden schon vor über 20 Jahren entwickelt. Um Pläne zu entwerfen, unterteilt der HTN-Planer die anstehende Aufgabe rekursiv in kleinere Aufgaben, bis primitive, nicht weiter zerlegbare Aufgaben erreicht sind, die dann direkt ausgeführt werden können. Um dies zu realisieren, benötigt jeder auf HTN basierende Planer eine Wissensdatenbank, die Regeln zur Unterteilung und Ausführung enthalten. Eine solche exemplarische Wissensdatenbank ist in Abbildung 1 zu sehen. Sie enthält Methoden, um einen Weg von A nach B zu finden und dabei von verschiedenen Verkehrsmitteln Gebrauch zu machen, die bestimmten Restriktionen unterworfen sind. So kann beispielsweise das Taxi nur bei geringer Entfernung von Start und Ziel benutzt werden. Die horizontalen Pfeile symbolisieren die total-order Strategie eines HTN.

Schon bei der Planung steht die Reihenfolge der Aktionen fest, die am Ende ausgeführt werden. Ovale Boxen symbolisieren eine Aufgabe, die sich in Unteraufgaben zerteilen lässt, rechteckige Boxen sind primitive Aufgaben, die direkt ausgeführt werden können.

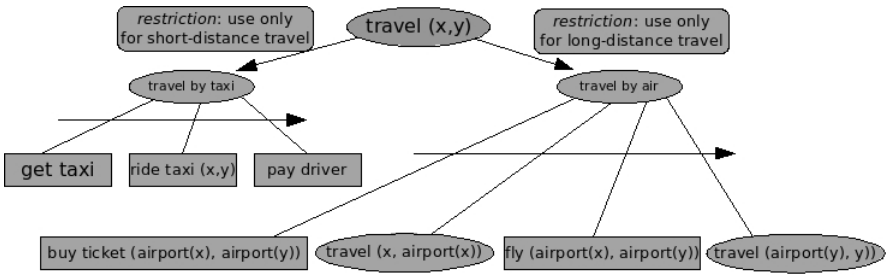


Abbildung 1: *exemplarische Wissensdatenbank*

Abbildung 2 zeigt den gefundenen Plan für das Problem **travel(Saarbrücken, Berlin)**. Der Planer beginnt an der Wurzel des Baumes und läuft aufgrund der Entfernungsrestriktion in den rechten Teilbaum, in dem er zuerst den elementaren Task **buy ticket()** ausführt und dann den komplexen Task **travel(x,airport(x))** in primitive Unteraufgaben teilt.
Hier wird die total-order Strategie deutlich: Es wird in derselben Reihenfolge geplant, in der die Aufgaben auch später im gefundenen Plan ausgeführt werden.

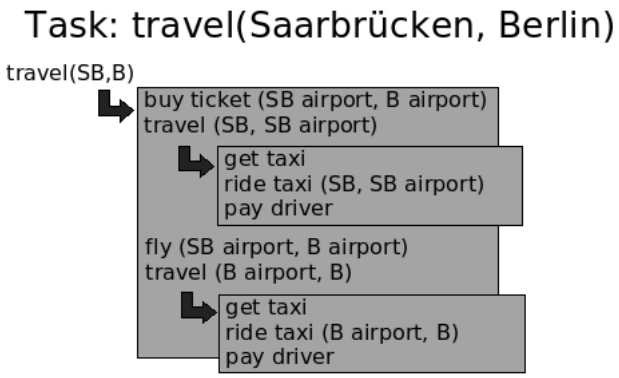


Abbildung 2: *Plan*

1.2 Beispieldomäne: Kartenspiel Bridge

Computerprogramme finden auch Anwendung als Gegner für menschliche Spieler, beispielsweise beim Schach. Dort schlagen sich Programme erstaunlich gut. Beim Kartenspiel Bridge hingegen waren menschliche Spieler den Computergegnern bisher überlegen. Das hängt damit zusammen, dass bei einem Kartenspiel wie Bridge die Karten des Gegenspielers nicht einsehbar sind und somit hat jeder Spieler nur ein Teil des Weltwissens.
Ein Baum, der alle *möglichen* Kartenverteilungen repräsentiert wäre für die Spielpraxis viel zu groß, um ihn in annehmbarer Zeit zu durchlaufen.
Ohne auf die genauen Spielregeln einzugehen, kann man dennoch sagen, dass Bridge ein Kartenspiel ist, dass auf Planen bzw. auf festgelegten Strategien beruht. In der Bridgeliteratur findet man taktische Vorgehensweisen wie z.B. finessing, ruffing oder crossruffing. Ein menschlicher Spieler kombiniert diese Vorgehensweisen nun zu Plänen, um seine Karten zu spielen. Diese Planungsnatur von Bridge macht man sich zunutze und adaptiert einen HTN Planungsalgorithmus.
In den so generierten Spielbäumen repräsentiert eine Verzweigung innerhalb des Baumes genau einen Zug in der entsprechenden Strategie. Dies hat zur Folge, dass im Gegensatz zu einem brute-force-Baum, der alle *möglichen* Spielzüge enthält, der HTN-Baum massiv ausgedünnt wird. Dieser kann dann auch während des Spiels in Echtzeit durchlaufen werden. Sollte allerdings ein Gegner von einer solchen Strategie abweichen, ist dieser Zug nicht mehr durch den HTN-Baum repräsentiert.
Ein solcher HTN-Baum enthält im *worst case* 305000 Knoten und 26000 im *average case*. Ein deutlicher Gewinn gegenüber $5.55*10^{44}$ Knoten im worst case bzw. 10^{24} im average case bei einem brute-force-Baum. Konkret implementiert wurde diese Strategie im Bridge Baron[1]

2 Vergleich zu traditionellem Planen

2.1 forward/backward chaining

Backward chaining benutzt wie viele andere traditionelle Verfahren eine zielgerichtete Methode, um einen Plan zu finden. Wie die Bezeichnung vermuten lässt, arbeitet sich der Planer hierbei rückwärts von einem Zielzustand aus zu einem zuvor definierten Anfangszustand.
Abbildung 3 zeigt das bekannte blocks-world-Problem mit Startzustand und Zielzustand. Nehmen wir an, move(X,Y) wäre ein Operator, der X nach Y transportieren kann, wobei X ein freier Block ist und Y entweder der Tisch oder ebenfalls ein freier Block. Vom Startzustand aus gibt es 36 verschiedene Möglichkeiten, diese move-Operation auszuführen, wobei aber nur eine, nämlich move(B,C) den Planer dem Ziel näher bringt. Würde

dieser nun vom Startzustand *vorwärts* planen, würden eine Menge falsche Pfade durchlaufen, bevor der richtige gefunden wird. Daher versucht der Planer den Zielzustand zu erreichen, in diesem Fall on(A,B) bzw. on(B,C). Daher wird er auf die einzigen Planungsschritte schauen, die in Frage kommen, nämlich move(A,B) und move(B,C).

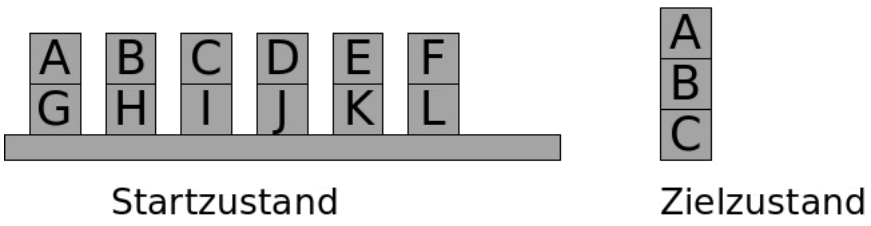


Abbildung 3: *blocks world*

- Die Anzahl der Verzweigungen auf der Suche nach einem Plan im Baum sind beim forward chaining generell höher als beim backward chaining.

- Viele Pfade im Suchraum können länger sein als der daraus resultierende Plan. Das heisst, wenn beim Planen der falsche Pfad im Baum genommen wird, kann es lange dauern, bis der richtige Pfad gefunden wird.

Im Gegensatz dazu gilt beim HTN-Planen

- Die Pläne sind im Gegensatz zum forward/backward chaining immer fast genauso lang wie die Suchpfade

- Auch wenn die Suche aufwändig sein kann, sucht der Planer den jeweils *optimalen* Plan und nicht *irgendeinen* Plan

- Ein HTN-Planer muss kein backward-chaining ausführen, um sein Ziel zu erreichen. Er führt nicht alle Planoperatoren aus, sondern nur diejenigen, die im HTN vorkommen.

2.2 Partial order planning

In einem Planungsproblem, in dem mehrere Teilziele zu erreichen sind kommt es beim *partial order planning* zur sog. Sussman anomaly. Abbildung 4 zeigt ein einfaches Problem aus der Blockwelt, bei dem es darum geht vom Startzustand (links) die Blöcke so zu stapeln, dass der Zielzustand (rechts) erreicht wird. Ein partial-order Planer wird nun zuerst das Zwischenziel on(A,B) auswählen obwohl es aber später zum Erreichen von on(B,C) wieder zerstört wird und danach wieder neu erzeugt werden muss. Wenn der Planer nicht explizite Mechanismen mitbringt, dies zu entdecken, gilt er als unvollständig. Unvollständig in dem Sinne, dass er in einigen Fällen keinen Plan findet, obwohl es einen gäbe.

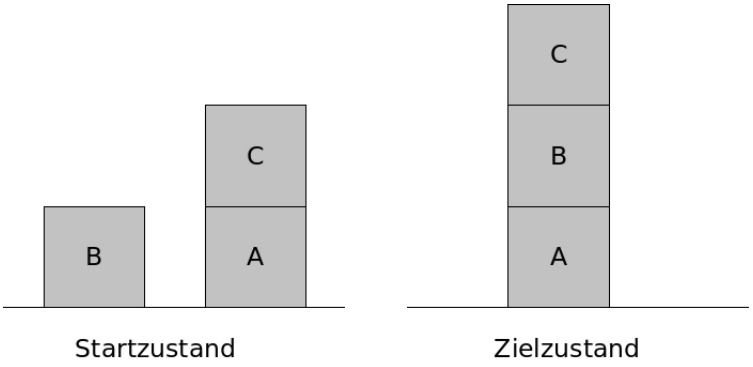


Abbildung 4: *Sussman Anomalie*

2.3 Zusammenfassung

Es bleibt also festzuhalten, dass ein Planer, der *forward chaining* einsetzt, grundsätzlich mehrere Pfade in einem Suchbaum ablaufen muss, bis er einen Plan gefunden hat. Im Gegensatz dazu entspricht beim Planen mit HTNs die Länge der Suchpfade auch meist der Länge der gefundenen Pläne. Die Sussman Anomalie zerstört Zwischenziele im Planungsprozess bei einer partial order Strategie, was bei HTNs nicht vorkommen kann.

3 SHOP: Eine HTN-Implementierung

3.1 Einleitung

SHOP (Simple Hierarchical Ordered Planer)[2] ist ein HTN Planungssystem, das Aufgaben in der selben Reihenfolge plant, in der sie auch später ausgeführt werden (vgl. Abbildung 1).

Nachdem wir uns die Syntax und Semantik von SHOP in den folgenden Kapiteln einführen, werden wir uns den Planungsalgorithmus näher betrachten. Eine exemplarische Planungsdomäne in Kapitel 3.5 zeigt danach die Vorgehensweise von SHOP bei einem einfachen Transportproblem.

3.2 Syntax und Semantik

Die Syntax von SHOP benutzt die Regeln der Prädikatenlogik und ist äußerlich an Prolog angelehnt. SHOP selbst aber ist in Lisp implementiert. Hier beispielsweise ein Menge von Hornklauseln, zuerst in Prolog Notation, dann in SHOP Notation:

```
p(f(X) :- q(X,c), r(Y,d), s(d)).
q(b,c).

(:- (p (f x) ((q ?x c) (r ?y d) (s d))))
(:- (q b c) nil)
```

Eine Domänenspezifikation von SHOP enthält *Zustände*, *Axiome*, *Operatoren* und *Methoden*.
Ein *Zustand* ist eine Menge von Atomen, eine Sammlung von Fakten wie z.B. 'Das Wetter ist schön' oder 'Ich stehe auf Position x'. Die genaue Syntax wird später besprochen.
Ein Axiom ist eine Menge von Hornklauseln und wird verwendet, um den Wahrheitswert einer Aussage zu ermitteln: 'Habe ich genug Geld um die Entfernung x mit dem Taxi zu fahren?'
Operatoren und Methoden werden dazu verwendet den Weltzustand zu verändern, wobei ein Operator im Gegensatz zu einer Methode keine Vorbedingungen enthält.
Eine Methode enthält drei Elemente:

- Den Zustand, der durch Anwendung dieser Methode erreicht werden kann.
- Eine Menge von Vorbedinungen, die erfüllt sein müssen, um die Methode anzuwenden.
- Einen Menge von Aktionen, die in derselben Reihenfolge ausgeführt werden, wie sie geplant werden. Hier wird auch die total-order Strategie von SHOP deutlich. Ein Element dieser Aktionen ist ein Operator.

Ein Operator enthält ebenfalls den Zustand, der durch dessen Anwendung erreicht werden kann. Des Weiteren eine Liste von Elementen, die aus dem Weltzustand entfernt werden müssen, sowie eine Liste von Elementen,

die dem Weltzustand hinzugefügt werden müssen, um den vom Operator spezifizierten Zustand zu erreichen.

Nachdem nun die allgemeine Intention der Elemente von SHOP eingeführt wurde, gibt der folgenden Abschnitt eine genaue Beschreibung der konkreten syntaktischen Implementierung.

Ein *Task* ist eine Liste der Form $(st_1t_2...t_n)$, wobei s der Name des Tasks ist und durch ein Tasksymbol bezeichnet ist. $t_1t_2...t_n$ sind die Terme und die Argumente des Tasks. Wir nennen den Task *primitiv*, wenn s ein primitives Task-Symbol ist. Primitive Tasksymbole beginnen mit einem Ausrufezeichen andernfalls nennen wir den Task es einen *compound task*, also einen Task, der in kleinere Aufgaben zerteilt werden kann.

Ein *Operator* hat die allgemeine Form `(:operator h D A)`, wobei `h` ein primitiver Task und der Kopf (head) des Operators ist. `D` und `A` sind *deletions* und *additions*, eine Menge von Atomen, die keine anderen Variablen enthalten als der Kopf h. Die Semantik von deletions und additions wird im nächsten Kapitel erklärt. Das folgende Beispiel zeigt einen einfachen Operator, um einen Block auf dem Tisch abzulegen:

```
(:operator (!putdown ?block))
      ((holding ?block))
((ontable ?block) (handempty)))
```

Eine *Methode* hat die allgemeine Form `(:method h C T)`, wobei h den Kopf der Methode bezeichnet und aus einen compound task besteht. C besteht aus Vorbedingungen für diese Methode und `T` ist eine Liste von Tasks bzw. beschreibt, in welche Unteraufgaben unterteilt werden muss, um h zu erreichen.
Methoden können auch mehrere Vorbedingungen und Dekompositionen enthalten:

(:method
    h
    Vorbedingung-1
    Unterteilung-1
    Vorbedingung-2
    Unterteilung-2 ...)

Die Idee dabei ist, eine if-then-else-Struktur zu realisieren. Wenn Vorbedingung-1 erfüllt ist, zerlege den zusammengesetzten Tasks h in die Unterteilung-1 usw.

Im folgenden sind zwei Methoden angegeben, die von einem Block die darauf liegenden entfernen:

```
(:method (make-clear ?y) ((clear ?y)) nil)
(:method (make-clear ?y)
((on ?x ?y))
((make-clear ?x)
(!unstack ?x ?y) (!putdown ?x))
```

Methode 1 hat die Vorbedingung *(clear ?y)*. Wenn also auf dem Block kein weiterer liegt, ist nichts zu tun. Die Liste von Task ist leer (nil). Wenn aber auf dem Block ein weiterer liegt, heben wir diesen ab und legen in daneben ab. Dies geschieht solange, bis kein weiterer Block über dem freizulegenden Block liegt und die Rekursion mit Methode 1 terminiert.

3.3 Planungsstrategie

Ein *Plan* ist eine Liste von Instanzen von Operatoren. Wenn p ein Plan ist und S ein Zustand, dann bezeichnet $p(S)$ denjenigen Zustand, der von S aus mit dem Plan p erreicht wird.

Ein *Planungsproblem* ist ein Tupel $P = (S, T, D)$, wobei S ein Zustand ist, T eine Taskliste und D eine Menge von Axiomen, Operatoren und Methoden.

Um nun den Planungsalgorithmus zu formulieren, definieren wir formal $\Pi(S, T, D)$, die Menge aller Pläne für T vom Zustand S mit Hilfe der Axiome, Operatoren und Methoden in D. Die Implementierung des Planungsalgorithmus ist dann einfach die Implementierung der formalen Definition von $\Pi(S, T, D)$.

Wenn T leer ist, enthält $\Pi(S, T, D)$ genau einen Plan: Den leeren Plan. Andernfalls bezeichnen wir mit t den ersten Task in T und mit R die verbleibenden Task in T. Wir unterscheiden drei Fälle:

1. Falls t ein primitiver Task ist und es einen elementaren Plan q für t gibt
   $\Rightarrow \Pi(S, T, D) = append(q, p) : p \in \Pi(q(S), R, D)$

2. Falls t ein primitiver Task ist und **kein** elementarer Plan p für t existiert
   $\Rightarrow \Pi(S, T, D) = \emptyset$

3. Falls t ein zusammengesetzter Task ist
   $\Rightarrow \Pi(S, T, D) = \cup\Pi(S, append(r, R), D) : r : einfache\ Reduktion\ von\ t$

3.4 Planungsalgorithmus

Wie im vorigen Kapitel erwähnt, ist der Planungsalgorithmus die Implementierung der formalen Definition von $\Pi(S, T, D)$

```
procedure find-plan(S,T,D)
  seek-plan(S,T,D,nil)
end find-plan
procedure seek-plan(S,T,D,p)
  if T=nil then return p
  t = 1. Task in T; R = verbleibende Tasks
  if t primitiver Task then
    if∃ simpler Plan q für t then
      return seek-plan(q(S),R,D,append(p,q))
    else return FAIL
  else
    for ∀ einfachen Reduktionen r für t in S
      result = seek-plan(S,append(r,R),D,p)
      if result ≠ FAIL then return result
    end for
    return FAIL
  end if
end seek-plan
```

3.5 Beispieldomäne

3.5.1 Domänenspezifikation

Tabelle 1 zeigt eine Beispieldomäne für ein einfaches Transportproblem. Die Domänenspezifikation enthält Axiome (A1, A2), Methoden (M1-M3) und Operatoren (O1-O5). Innerhalb dieses Transportproblems geht es darum, von einem bestimmten Punkt in der Stadt mittels Taxi, Bus oder einfach per Fuß an das vorgegebene Ziel zu gelangen. Die Fortbewegungsmittel sind bestimmten Restriktionen unterworfen, so kann das Taxi beispielsweise nur benutzt werden, wenn genug Geld für die Fahrt vorhanden ist. Der Fußweg kann nur eingeschlagen werden, wenn das Wetter gut und das Ziel nicht zu weit entfernt ist. Hier kommen die Vorbedingungen von Methoden wie in Kapitel 5.3 beschrieben zum Einsatz. Beim Betrachten der Planungsdomäne wird auch deutlich, dass die Aktionen in der selben Reihenfolge geplant, wie sie später tatsächlich ausgeführt werden (vgl. Kapitel 1). So spezifiziert Methode M3 beispielsweise, wie ein Taxi zu benutzen ist: Das Taxi rufen, das Taxi benutzen und im Anschluss den Fahrer bezahlen, in genau dieser Reihenfolge. Diese Methode realisiert auch eine if-then-else-Struktur wie in der Syntax zu SHOP allgemein eingeführt.

Tabelle 2 zeigt ein spezifisches Transportproblem innerhalb dieser Domäne und den Weltzustand zum Startzeitpunkt.

3.5.2 Beispieldurchlauf

Wir starten mit Methode M2 und setzen die Variable ?q auf *suburb* und ?p auf *downtown*. Der Planer überprüft die Vorbedingung von M2 mit Hilfe des Axioms A2. Diese gibt false zurück, da die Entfernung zum Ziel zu weit entfernt ist, um den Fußweg einzuschlagen. Also geht der Planer über zu Methode M3 und setzt die entsprechenden Variablen aus dem aktuellen Zustand ein und stellt beim Überprüfen der Vorbedingung von M3 fest, dass nicht genug Geld vorhanden ist, um das Taxi zu benutzen.
Da die Vorbedingung nach dem Einsetzen der aktuellen Werte für die Variablen der Busbenutzung erfüllt ist, wählt der Planer diesen Transportweg aus.

4 Der Nachfolger SHOP2

4.1 Warum SHOP2

SHOP kann Pläne sehr schnell generieren und plant effizient. Die Kehrseite der Medaille ist allerdings, dass es sehr schwierig und zeitaufwändig ist, gute Wissensdatenbanken für SHOP zu schreiben. Das vorgestellte Beispiel beschreibt nur einer sehr kleine, exemplarische Domäne. Reale Anwendungsszenarien sind dagegen wesentlich komplexer und erfordern einen erheblich höheren Aufwand.
In SHOP müssen die innerhalb der Methoden die Unteraufgaben ebenfalls der total-order-Restriktion genügen. Das heisst, die Unteraufgaben müssen in der selben Reihenfolge geplant werden, wie sie später auch ausgeführt werden. Dies macht es unmöglich, Unteraufgaben zu verschachteln. SHOP2 verfolgt nun den Ansatz, diese Restriktion zu lockern und ein partial-order-planning für Unteraufgaben von Methoden zuzulassen um in den Domänen, bei denen der Aufwand mit SHOP zu hoch ist, effizienter zu planen.[3]

4.2 Motivation

Um die Problematik der total-order-Restriktion für Methoden bei SHOP zu verdeutlichen, betrachten wir das nachfolgende Beispiel, bei dem es darum geht, ein Paket von einer Startposition zu einer Zielposition zu transportieren. Um mit SHOP dieses Problem zu lösen, benötigt der Planer eine Methode wie in Abbildung 5 (a). Um also das Paket zu befördern, gehen

Nr.	Eintrag
A1	(:- (have-taxi-fare ?distance) ((have-cash ?m) (eval (≥ ?m (+ 1.5 ?distance))))) (:- (walking-distance ?u ?v) ((weather-is good) (distance ?u ?v ?w) (eval (≤ ?w 3))) ((distance ?u ?v ?w) (eval (≤ ?w 0.5))))
M1 M2 M3	(:method (pay-driver ?fare) ((have-cash ?m) (eval (≥ ?m ?fare))) ((!set-cash ?m ,(- ?m ?fare)))) (:method (travel-to ?q) ((at ?p) (walking-distance ?p ?q)) ((!walk ?p ?q))) (:method (travel-to ?y) ((at ?x) (at-taxi-stand ?t ?x) (distance ?x ?y ?d) (have-taxi-fare ?d)) ((!hail ?t ?x) (!ride ?t ?x ?y) (pay-driver ,(+ 1.50 ?d))) ((at ?x) (bus-route ?bus ?x ?y)) ((!wait-for ?bus ?x) (pay-driver 1.00) (!ride ?bus ?x ?y)))
O1 O2 O3 O4 O5	(:operator (!hail ?vehicle ?location) () ((at ?vehicle ?location))) (:operator (!wait-for ?bus ?location) () ((at ?bus ?location))) (:operator (!ride ?vehicle ?a ?b) ((at ?a) (at ?vehicle ?a)) ((at ?b) (at ?vehicle ?b))) (:operator (!set-cash ?old ?new) ((have-cash ?old)) ((have-cash ?new))) (:operator (!walk ?here ?there) ((at ?here)) ((at ?there)))

Tabelle 1: *Beispieldomäne*

Startzustand	1.((at downtown) 2. (weather-is good) 3. (have-cash 12) 4. (distance downtown park 2) 5. (distance downtown uptown 8) 6. (distance downtown suburb 12) 7. (at-taxi-stand taxi1 downtown) 8. (bus-route bus1 downtown park) 9. (bus-route bus2 downtown uptown) 10.(bus-route bus3 downtown suburb))
Aufgabenliste	((travel-to suburb))

Tabelle 2: *Startzustand mit Transportproblem*

wir zur Startposition, nehmen das Paket auf, gehen zur Zielposition und laden es dort ab.

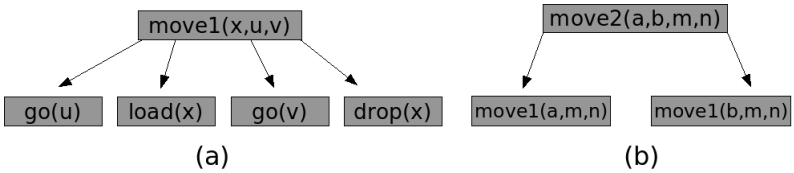


Abbildung 5: *Methoden für den Transport von Paketen*

Wenn wir nur ein Paket transportieren möchten, funktioniert das wunderbar. Nehmen wir nun an, wir möchten zwei Pakete auf einmal transportieren. Auf den ersten Blick erscheinen die Methoden in Abbildung 5 (a) und (b) ausreichend: move2 führt zweimal move1 aus. Probleme entstehen genau dann, wenn beide Pakete die identische Start- und Zielposition haben.

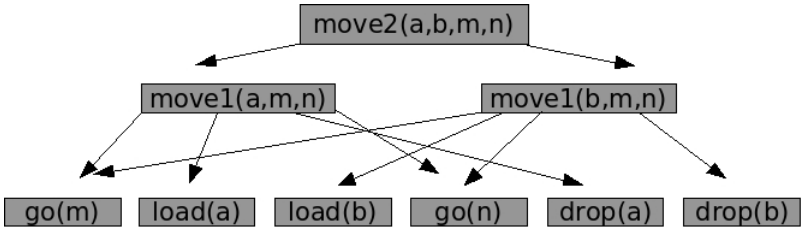


Abbildung 6: *gewünschter Plan, wenn Start- und Zielposition gleich*

Dann nämlich wäre ein Plan wünschenswert, wie in Abbildung 6 zu se-

hen. Aber der Planer wird mit den Methoden aus Abbildung 5 (a) und (b) den (ungünstigen) Plan wie in Abbildung 7 generieren. Nämlich den doppelten Weg fahren, der eigentlich nötig wäre.

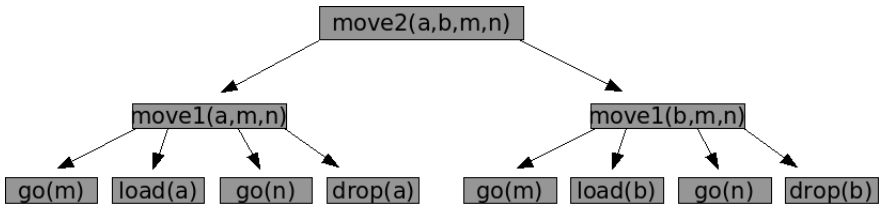


Abbildung 7: *Tatsächlich generierter Plan*

In diesem einfachen Beispiel ist es kein Problem, Methoden zu schreiben, die es dem Planer erlauben, zwei Pakete effizient zu transportieren (Abbildung 8). Daraus resultiert dann der gewünschte Plan wie in Abbildung 9 zu sehen.

Dennoch sieht man selbst an diesem einfachen Beispiel, dass man SHOP explizit die Methoden mitgeben muss, damit der Planer mit zwei Paketen effizient umgehen kann. Damit steigt die Anzahl der benötigten Methoden erheblich, was eine Fehlersuche schwieriger gestaltet. Hier handelt es sich um ein simples Beispiel. In der Praxis treten Probleme auf, die wesentlich komplexer sind.

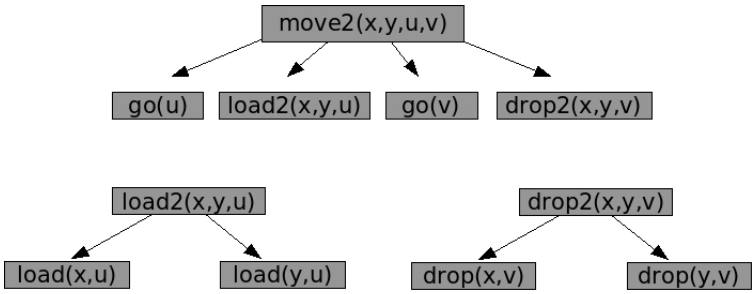


Abbildung 8: *Methoden, die den Umgang mit zwei Paketen effizienter machen*

4.3 SHOP2: Syntax

Wie auch der Vorgänger SHOP enthält die Domäne eines SHOP2-Planers **Axiome**, **Methoden** und **Operatoren**. SHOP2 ist kompatibel mit SHOP, d.h. SHOP2 kann - mit geringen syntaktischen Änderungen - Wissensdatenbanken von SHOP übernehmen und darin planen.

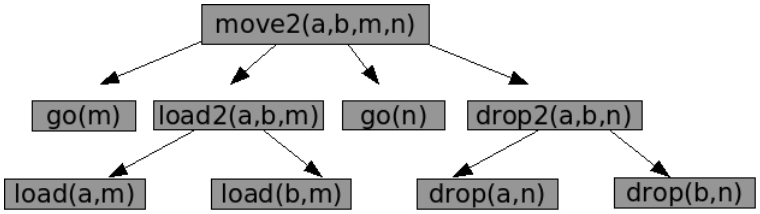


Abbildung 9: *Plan, der mit Hilfe der Methoden aus Abbildung 8 generiert wurde*

Axiome bestehen wie in SHOP aus Hornklauseln und können auch in SHOP2 nummerische Berechnungen durchführen (siehe Transportdomäne). Diese Berechnungen werden wie auch in SHOP dazu verwendet die Vorbedingungen von Methoden zu überprüfen. Die Syntax ist identisch zu SHOP.

Methoden sind in SHOP2 ähnlich wie in SHOP, mit dem Unterschied, dass diese nun Unteraufgaben generieren können, die teilweise geordnet sind (partially-orderd). Wie in SHOP ist die Syntax einer Methode `(:method h C T)`, wobei h ein nicht-elementarer Task ist (compound task). C sind die Vorbedingungen und T die Liste von Operationen, die ausgeführt werden müssen, um den Zustand h zu erreichen bzw. in welche Aufgaben diese Methode unterteilt werden muss (vgl. Kapitel 1). Wie auch in SHOP können mehrere Vorbedingungen benutzt werden, um eine if-then-else-Struktur innerhalb der Methode zu realisieren.

Operatoren unterscheiden sich von denen in SHOP. Dadurch, dass SHOP2 die total-order-Restriktion lockert und dadurch Unteraufgaben von Methoden partially-ordered sein können, kann es vorkommen, dass diese Unteraufgaben sich verschachteln. Dadurch kann es passieren, dass sich Löschungen von Atomen überschneiden weshalb der Operator in SHOP2 einen Schutzmechanismus implementiert, um dies zu vermeiden.
Zur Erinnerung: Der Operator in SHOP hatte die allgemeine Form:

```
(:operator h D A)
```

wobei h ein elementarer Task ist. h kann erreicht werden, indem die Atome in D aus dem Weltzustand entfernt werden und die Atome in A dem Weltzustand hinzugefügt.
Wir erweitern deshalb den Operator von SHOP2 um zwei Parameter:

```
(:operator h D A P C)
```

wobei P eine Liste von Atomen ist, die nicht gelöscht werden dürfen. Für die Atome in C wird der Löschschutz aufgehoben und sie sind wieder zum Entfernen freigegeben.

4.4 SHOP2: Planungsalgorithmus

Der Planungsalgorithmus von SHOP2 berücksichtigt den Schutzmechanismus, wie im vorigen Kapitel erläutert. In der nachfolgenden Darstellung bezeichnet S den aktuellen Zustand, M eine teilweise geordnete Liste von tasks und L die Liste mit den zu schützenden Atomen. Weiterhin ist zu erwähnen, dass SHOP2 nichtdeterministisch aus den tasks auswählt. Zur Erinnerung: SHOP nimmt immer den ersten task t in der Taskliste (vgl. 3.4).

procedure SHOP2(S,M,L)
  if M leer **then return** NIL **endif**
  wähle nichtdeterministisch einen task t in M, der keine Vorgänger hat
  (r,R') = reduction(S,t)
  if r = FAIL **then return** FAIL **endif**
  wähle nichtdeterministisch eine Operatorinstanz o aus, die auf r in S anwendbar ist
  S' = Der Zustand, der sich von S aus ergibt, wenn o auf r angewendet wird
  L' = Die Liste der zu schützenden Atome, die sich aus L ergibt, wenn o auf r angewendet wird
  M' = Die Menge von teilweise geordneten Tasks, die sich aus M ergeben, wenn t durch R' ersetzt
  P = SHOP2(S',M',L')
  return cons(o,P)
end SHOP2
procedure reduction(S,t)
  if t primitiver task **then return** (t,NIL)
  else if keine Methode anwendbar auf t im Zustand S **then**
    return (FAIL,NIL) **endif**
  wähle m nichtdeterministisch als eine Methode anwendbar auf t im Zustand S aus
  R = Die Dekomposition, die von m von t ergibt
  r = ein beliebiger task in R, der keine Vorgänger hat
  (r',R') = reduction(S,r)
  if r' = FAIL **then return** ¡FAIL,NIL¿ **endif**
  R" = Die Menge teilweise geordneter Unteraufgaben, die sich aus R ergibt, indem man r durch R
  return ¡r',R"¿
end reduction

5 SHOP vs. SHOP2

Um die beiden Planer miteinander vergleichen zu können, hat man sie in zwei unterschiedlichen Domänen gegeneinander antreten lassen. Dazu hat man sich die Größe der jeweiligen Wissensdatenbanken angeschaut, sowie die Zeiten, die für die Lösung von unterschiedlich komplexen Probleme benötigt wurden. Eine weiteres Kriterium für die Qualität eines Planers war in diesem Versuch die Länge des generierten Plans. Da SHOP2 auch Wissendatenbanken von SHOP verarbeiten kann, hat man als Planer SHOP2 mit beiden

Datenbanken verwendet.

Die beiden Domänen, in denen die Planer arbeiten sind bekannte Beispieldomänen der KI. Einmal die Logistikdomäne und einmal die Blockweltdomäne.

In der blocks-world-Domäne geht es darum eine bestimmte Konfiguration von Blöcken übereinander zu stapeln, wobei immer nur ein Block auf einmal bewegt werden kann und mehrere Blöcke auf einem zu erreichenden Block liegen, die es zuerst zu entfernen gilt.

Die Logistikdomäne[2] wurde entworfen um eine komplexere Domäne als blocks-world zu schaffen und es wurden zusätzliche Aspekte einer Transportlogistik verwendet. Ziel ist es, eine Fracht von einer Startposition zu einer Zielposition zu bringen, wobei verschiedene Transportwege zur Verfügung stehen (Luft, Schiene und Straße). Im Gegensatz zur blocks-world, wo nur einfache Blöcke transportiert werden müssen, gibt es hier verschiedene Frachten, wie zum Beispiel Briefe, Pakete, Stahl oder aber auch Gefahrengut, das nicht durch bestimmte Städte gefahren werden darf.

5.1 Logistikdomäne

Wie in Tabelle 2 zu sehen ist die SHOP-Wissensdatenbank für die Logistikdomäne wesentlich größer als die von SHOP2. Es war nicht möglich, die Methoden von SHOP zu vereinfachen, ohne dem Planer die Kenntnisse über die Domäne zu entziehen und ihn somit brute-force laufen zu lassen.

Getestet wurde über 110 zufällig generierte Probleme. Es mussten immer N Pakete transportiert werden, für N=10, 15, ... , 60 und für jedes N wurden zehn Probleme generiert. In jedem Problem selbst gab es nicht mehr als N/2 Städte und jede Stadt enthielt drei Start- bzw. Zielpunkte. Zusätzlich wurde vorausgesetzt, dass die Start- und Zielorte zufällig ausgewählt und verschieden voneinander waren.

Abbildung 10 zeigt die benötigte CPU-Zeit für die beiden Wissensdatenbanken in Abhängigkeit der Problemgröße. Wie man sieht, ist die SHOP2-Datenbank wesentlich schneller, als die von SHOP. Die Größe der Pläne allerdings ist bei beiden Datenbanken ungefähr gleich, wie in Abbildung 11 zu sehen ist.

5.2 Blockweltdomäne

Wie auch in der Logistikdomäne war es in der Blockweltdomäne nicht möglich, die Wissensdatenbank von SHOP zu vereinfachen ohne dem Planer die Domänenkenntnisse zu entziehen und ihn zu einem brute-force-Ansatz zu zwingen.
In der SHOP2 Datenbank wurden zusätzlich die total-order-Restriktionen

[2] http://www.cs.umd.edu/projects/plus/UMT/umt.ps

Abbildung 10: *CPU Zeiten für SHOP und SHOP2 in Abhängigkeit der Problemgröße in der Logistikdomäne. Die x-Achse zeigt die Problemgröße, die y-Achse die benötigte CPU-Zeit in Sekunden*

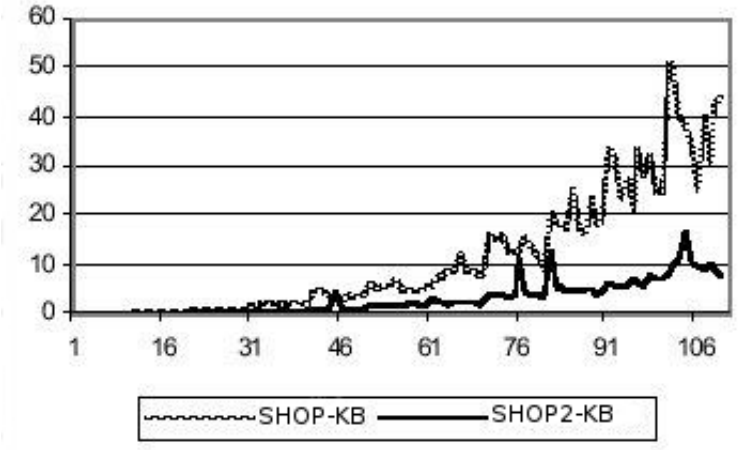


Abbildung 11: *Größe der generierten Pläne mit SHOP bzw. SHOP2 Datenbank in der Logistikdomäne. Die x-Achse zeigt die Problemgröße, die y-Achse die Größe des Plans*

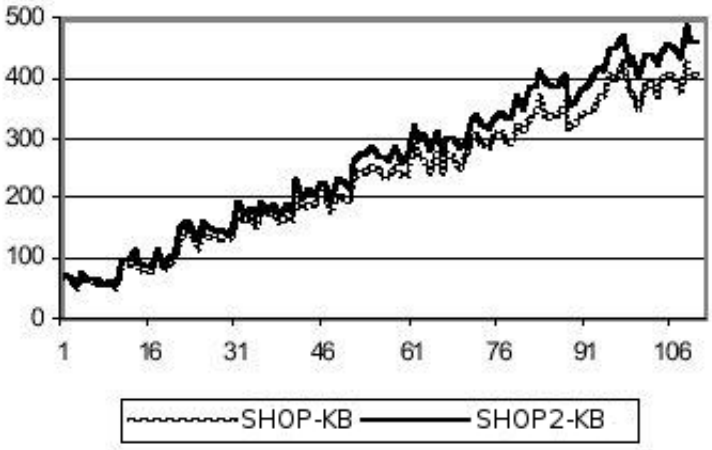


Tabelle 3: *Größen der Wissensdatenbanken in der Logistikdomäne für SHOP und SHOP2*

	SHOP	SHOP2
Methoden	50	10
Operatoren	7	7
Axiome	10	1

Tabelle 4: *Größen der Wissensdatenbanken in der Blockweltdomäne für SHOP und SHOP2*

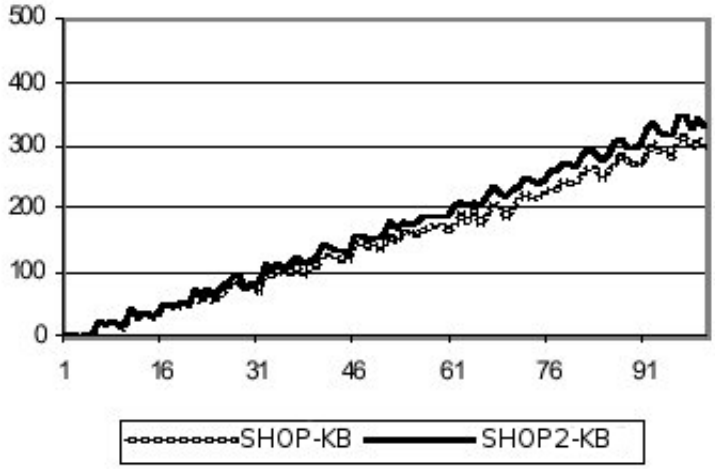
	SHOP	SHOP2
Methoden	10	13
Operatoren	7	8
Axiome	1	5

wie in den vorangegangen Kapiteln beschrieben relaxiert um ein Verschachteln der Methode zu realisieren. Die notwendigen Schutzmechanismen schlagen sich in der Größe der Wissensdatenbank nieder, wie in Tabelle drei zu sehen.
Die Planer mussten jeweils N=5,10,...100 Blöcke verrücken. Es wurden fünf Probleme für jedes N generiert, also insgesamt 100 Probleme. Die Start- bzw. Zielzustände wurden zufällig generiert.

Wie in Abbildung 12 zu sehen schwankt die benötigte CPU-Zeit von SHOP2 mit der SHOP2-Datenbank erheblich je nach Problem. Im Schnitt war SHOP2 2.4 mal langsamer als SHOP2 mit der SHOP-Datenbank. Dennoch unterscheiden sich die Größen der gefundenen Probleme kaum, wie in Abbildung 13 zu sehen. SHOP2 verarbeitet also SHOP-Datenbanken schneller als sein Vorgänger.

Abbildung 12: *CPU Zeiten für SHOP und SHOP2 in Abhängigkeit der Problemgröße in der Blockweltdomäne. Die x-Achse zeigt die Problemgröße, die y-Achse die benötigte CPU-Zeit in Sekunden*

Abbildung 13: *Größe der generierten Pläne mit SHOP bzw. SHOP2 Datenbank in der Blockweltdomäne. Die x-Achse zeigt die Problemgröße, die y-Achse die Größe des Plans*



6 Zusammenfassung

Diese Arbeit hat einen grundlegenden Einstieg in das Planen mit HTNs gegeben. Ein solcher Planer greift auf eine Wissensdatenbank zurück, die explizit für jede Domäne geschrieben, was den initialen Aufwand erhöht. Allerdings kennt der Planer zu jedem Zeitpunkt den Zustand der Welt. Ein HTN-Planer verwendet eine total-order Strategie, bei der die einzelnen Schritte des Plans in der selben Reihenfolge geplant werden, wie sie später tatsächlich ausgeführt werden. Im Gegensatz zum backward-/forward-chaining, sucht ein HTN-Planer immer nach dem *optimalen* Plan und nicht nach irgendeinem Plan. Am Beispiel des Bridge Baron haben wir gesehen, dass durch diese Einschränkung ein Durchsuchen eines Planungsbaums in Echtzeit stattfinden kann, während das für einen backward-/forward-chaining Planer nicht möglich ist. Es sei auch noch einmal erwähnt, dass ein HTN-Planer dadurch auch nicht auf einen Spielzug, der nicht in seinem Planungsbaum bzw. nicht in seiner Planungsdomäne repräsentiert ist, reagieren kann.

Es wurde die konkrete Implementierung eines HTN-Planers in SHOP und dessen Nachfolger SHOP2 besprochen sowie dessen Syntax und Semantik. Ein weiterer Punkt war die Vorgehensweise von SHOP beim Finden von Plänen. Eine Beispieldomäne sollte dies anschaulich machen.

SHOP2 wurde nötig, da die total-order-Restriktion von SHOP das Schreiben von guten Wissensdatenbanken für manche Domänen unnötig komplex gemacht hat. Deshalb hat man in SHOP2 den Unteraufgaben von Methoden ein partial-order-planning zugelassen und somit diese Restriktion relaxiert. Dies führte aber dazu, dass sich nun diese Unteraufgaben verschachteln konnten und man musste einen Schutzmechanismus einführen, um Löschungen von Atomen zu verhindern. Realisiert wurde das Ganze, in dem man die Syntax von `:method()` erweitert hat.
Durch dieses Relaxieren hat SHOP2 gegenüber SHOP im direkten Vergleich in der Transportdomäne seinen Vorteil ausspielen können, nicht jedoch in der Blockwelt-Domäne.

Literatur

[1] Dana Nau, Stephen Smith, Kutluhan Erol (1998)
Control Strategies in HTN Planning: Theory Versus Practice
In: AAAI-98/IAAI-98 Proceedings, pp. 1127-1133

[2] Dana Nau, Yue Cao, Amnon Lotem, Héctor Muñoz-Avila (1999)
SHOP: Simple Hierarchical Ordered Planer
In: IJCAI-99, pp. 968-973

[3] Yue Cao, Amnon Lotem, Steven Mitchell (2001)
Total Order Planning with Partially Ordered Subtasks
In: IJCAI-2001.