SHOP

(Simple Hierarchical Ordered Planner)

Planning with Ordered Task Decomposition

Irina Schens, 07.01.2003

Gliederung

SHOP (Simple Hierarchical Ordered Planner)

- Domänen - unabhängiger Algorithmus für eine geordnete Aufgabenzerlegung
- Arbeitet im Prinzip wie ein HTN - Planer
- Kennt den gesamten Weltzustand in jedem Planungsschritt
- Besonderheit: akzeptiert nur total geordnete Tasknetze

Einschub: HTN -Planer

- **Hierarchical Task Networks** (Erol & Hendler & Nau 1994)
- **Tasks** entsprechen sowohl Zielen als auch abstrakten Operatoren
- **primitive tasks:** (STRIPS) - Operatoren - sind ausführbar
- **goal tasks:** Ziele
- **compound tasks:** komplexe Zielspezifikationen

Einschub: HTN -Planer

- **task networks**: Graphen
- **Knoten**: tasks
- **Kanten**: *constraints*: Beziehungen zwischen tasks, Variablenbedingungen
- **Methoden**: non-primitive tasks + task networks

HTN Planen:
Ausgehend von einem *initialen* Task-Netz werden mit Hilfe geeigneter Methoden alle Tasks solange verfeinert, bis das Task-Netz nur noch primitive Tasks enthält sodass alle Constraints erfüllt sind.

SHOP (Simple Hierarchical Ordered Planner)

- SHOP ist korrekt und vollständig
- **Theorem1** *(Korrektheit von SHOP)*
  Angenommen, dass einer der Pfade von SHOP(S,T,D) einen Plan P zurückgibt. Dann löst der Plan P das Planungsproblem (S,T,D).
- **Theorem2** *(Vollständigkeit von SHOP)*
  Angenommen, dass das Planungsproblem (S,T,D) lösbar ist. Dann gibt wenigstens einer der Pfade von SHOP(S,T,D) einen Plan zurück.
- Die Methode ist mächtig genug
  um sie in komplexen Real-Welt Problemen zu nutzen

Syntax und Semantik

- **Symbole:**
  - **Variablensymbole**
    Lisp - Symbole, die mit einem Fragezeichen beginnen, so wie `?x` oder `?hello-there`
  - **primitive task - Symbole**
    Lisp - Symbole, die mit einem Ausrufezeichen beginnen, so wie `!unstack` oder `!putdown`
  - **Konstanten, Funktionssymbole, predicate - Symbole oder compound - Symbole**
    Lisp - Symbole, die mit keinem Ausrufezeichen oder Fragezeichen beginnen, so wie `block1`

Syntax und Semantik

- **Logische Ausdrücke:**
  - **Term**
    eine Variable, Konstante oder eine Liste der Form (f t1 t2 ...tn)
    /f - Funktionssymbol und jedes ti - Term/
  - **Atom**
    eine Liste der Form(p t1 t2 ... tn)
    /p - Prädikatensymbol und jedes ti - Term/, z.B. (on block2 ?x)
  - **conjunct**
    eine Liste von Literalen (l1 l2 ...ln),
    z.B. **((ontable block1) (clear block1))**
    in Prolog - Notation:
    ontable(block1), clear(block1)

  Ein Ausdruck ist ein ***ground*** - Ausdruck, wenn er keine variablen Symbole enthält

Syntax und Semantik

- **Logische Ausdrücke (2)**
  - **Substitution**
    eine Liste der Form ((x1 . t1) (x2 . t2) ... (xk . tk))
    /xi - Variable und jedes ti - Term/,
    z.B. **((?b . block1)(?y . (f ?x)))**
    in Prolog - Notation:
    {block1 /B, f(X)/Y}
  - **Substitution instance e^u**
    (Ein Ausdruck e^u der aus dem Ausdruck *e*
    durch die simultane Substitution *(xi . ti)* entsteht)

Input und Output

- **Input:**
  - **State:** ein Satz von Grundatomen
  - **Task List:** eine lineare Liste von Aufgaben
  - **Domain:** Methoden, Operatoren, Axiome
- **Output:** ein oder mehrere Pläne
  - je nachdem, was wir SHOP zu suchen auffordern, kann er zurückgeben:
    - den ersten Plan, den er findet
    - alle möglichen Pläne
    - einen Plan mit den niedrigsten Kosten
    - alle Pläne mit den niedrigsten Kosten
    - usw.

Elemente des Inputs

- **State:** eine Liste von Grundatomen (in Lisp Notation)
  Zwei Zustände sind gleich, wenn sie dieselben Atome enthalten. (unabhängig von der Erscheinungsreihenfolge)
  z.B. ***((at home)(have-cash 50.43)(distance home downtown 10))***
- **Unifier:**
  Wenn *d* und *e* zwei Ausdrücke sind und es eine Substitution *u* gibt, so dass $d^u=e^u$, dann sind *d* und *e* **unifiable** und *u* ein **unifier**.
- **Satisfier:**
  Sei *S* ein Zustand, *X* eine Axiomenliste und *C* ein conjunct.
  S erfüllt *C*, wenn es eine Substitution *u* gibt, so dass $S \cup X\ C^u$ ergibt, dann ist *u* ein **satisfier**.

Elemente des Inputs

- **Task List:** eine lineare Liste von auszuführenden Aufgaben
  z.B. ***((travel home downtown)(buy book))***

  SHOP besitzt zusätzlich zu den oben beschriebenen logischen Symbolen zwei andere Symbole:
  - **primitive task - Symbole**
    (mit einem Ausrufezeichen beginnende Lisp - Symbole)
  - **non -primitive task - Symbole**
    (Lisp - Symbole ohne Ausrufezeichen)

  Task ist eine Liste der Form: **(s t1 t2 .. tn)**
  /s - Task-Symbol und jedes ti -Term/
  Eine Aufgabe ist **primitive task** oder **non -primitive task**, je nachdem ob s primitiv oder non -primitiv ist.

Elemente des Inputs

- **Methode:** ein Ausdruck der Form m = **(: method h C ´T)**
  - h -(**head)**: eine zusammengesetzte Aufgabe;
  - C-(**precondition**): ein *conjunct;*
  - T -(**tail)**: eine *Task List*

Bedeutung:

Wenn *C* erfüllt ist, dann wird *h* durch das Ausführen der Aufgaben in *T* in der gegebenen Reihenfolge gelöst.

Ausführlicher:

Sei *t* ein Aufgabenatom und *S* ein Zustand.
Angenommen, dass *u* ein *unifier* für *h* und *t* sowie *v* ein *satisfier* für C^u in *S* ist. Dann ist die Methodeninstanz $(m^u)^v$ auf *t* in *S* anwendbar und das Ergebnis ist eine Aufgabenliste $r=(T^u)^v$.
Die Aufgabenliste ***r*** ist eine **einfache Reduktion von *t* in *S***.

Elemente des Inputs

Bsp1: **m=(: method (move-block-to-table ?x)**
((on ?x ?y) (clear ?x))
`((!unstack ?x ?y) (!putdown ?x)))

S = ((on a b) (ontable b) (clear a) (handempty));

t = (move-block-to-table a);

u = ((?x . a));
v = ((?y . b));

Dann: $(m^u)^v$ = (:method (move-block-to-table a)
((on a b) (clear a))
'((!unstack a b) (!putdown a));

***r* = ((!unstack a b) (!putdown a));**

Elemente des Inputs

- **Jedes Axiom:** eine Horn Klausel

ein Ausdruck der Form **(:-a C1 C2 C3 ... Cn)**

/ a (head) - ein Atom und (C1 C2 ... Cn) - tail)

Bedeutung: a ist true wenn C1 true ist oder
wenn C1 false, aber C2 true, oder C1 und C2 false, aber C3 true usw./

z.B.: **(:- (p (f ?x))**
((q ?x c) (r (g ?y) d) (s d)))

in Prolog - Notation:

```
p(f(x)) :-
        q(x, c), r(g(y),d),s(d).
```

Elemente des Inputs

- **Operator:** ein Ausdruck der Form **(: operator h D A c)**

h (head)	- primitive Aufgabe
D	- delete-Liste
A	- add-Liste
c	- Kosten

- wie ein STRIP - Operator, aber ohne Vorbedingungen
- führt eine primitive Aufgabe aus

Aus dem Zustand *S* entsteht nach dem Ausführen von o^u (oder h^u) ein neuer Zustand:

result (S, h^u) = result(S, o^u) = (S - D^u) ∪ A^u

Output

Wenn *t* eine primitive Aufgabe ist und es einen mgu *u* für *t* und *h* (Operator - head) gibt (h^u ist ground) , dann ist die Operator - Instanz o^u auf *t* anwendbar und $\mathbf{h^u}$ ist **ein einfacher Plan** für t.

- **Plan:** eine Liste der Form **(p1 c1 p2 c2 ... pn cn)**

  Jedes *pi* und jedes *ci* sind der Kopf und die Kosten der Operator - Instanz *oi.*

  Wenn P = (p1 p2 ... pn) ein Plan ist und S - ein Zustand, dann ist das Ergebnis der Anwendung von P in S:

  result(*S*,*P*) = result(result(…(result(*S*,*p*1),*p*2),…),*p*n)

Output

Bsp2: ***S*** **= ((on a b) (ontable b) (clear a) (handempty));**

o **= (:operator (!unstack ?x ?y)**
((clear ?x) (on ?x ?y) (handempty))
((holding ?x) (clear ?y)));

o' **= (:operator (!putdown ?block)**
((holding ?block))
((ontable ?block) (clear ?block) (handempty)));

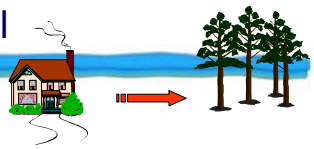
u **= ((?x . a) (?y . b));**
u' **= ((?block . a));**

P **= ((!unstack a b) (!putdown a)).**

Dann

head(*o*)^u	**= (!unstack a b);**
head(*o'*)$^{u'}$	**= (!putdown a);**
result(*S*,*o*^{u})	**= ((ontable b) (clear b) (holding a));**
result(*S*,*P*)	**= result (result (*S*,*o*^{u}) , (*o'*)$^{u'}$)**
	= ((ontable b) (clear b) (ontable a) (clear a) (handempty))

Einfaches Beispiel



- Aufgabenliste: ((travel home park))
- Startzustand: ((at home) (cash 20) (distance home park 8))
- Methoden: (Aufgaben, Vorbedingungen, Unteraufgaben)
  - (: method (travel ?x ?y)
    ((at x) (walking-distance ?x ?y)) `((!walk ?x ?y)) 1)
  - (: method (travel ?x ?y)
    ((at ?x)(have - taxi - fare ?x ?y))
    `((!call -taxi ?x)(!ride ?x ?y)(!pay - driver ?x ?y)) 1)
- Axiome:
  - (:- (walking-dist ?x ?y)((distance ?x ?y ?d)(eval(<=?d 5))))
  - (:- (have-taxi-fare ?x ?y)
    ((have-cash ?c)(distance ?x ?y ?d)(eval(>= ?c(+1.50 ?d))))
- Operatoren: (Aufgabe, delete-Liste, add-Liste)
  (: operator (!walk ?x ?y)((at ?x))((at ?y)))
  ...

Der SHOP Algorithmus

procedure SHOP (state *S,* task-list *T,* domain *D*)
1. **if** *T* = nil **then return** nil
2. *t* = the first task in *T*
3. *U* = the remaining tasks in *T*
4. **if** *t* is primitive and there is a simple plan for *t* **then**
5. nondeterministically choose a simple plan *p* for *t*
6. *P* = SHOP(result(*S,p*),*U*,*D*)
7. **if** *P* = FAIL **then return** FAIL **endif**
8. **return** cons(*p*,*P*)
9. **else if** *t* is non-primitive and there is a simple reduction of *t* in *S* **then**
10. nondeterministically choose any simple reduction *R* of *t* in *S*
11. **return** SHOP(*S*,append(*R,U*),*D*)
12. **else**
13. **return** FAIL
14. **endif**

end SHOP

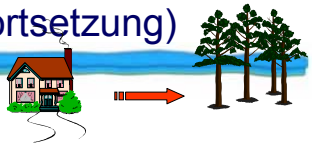
state S; task list T=($\mathbf{t_1}$,t_2,...)
operator instance **o**

state o(S) = result(*S,p)* ;
task list T=(t_2, …)

task list T=($\mathbf{t_1}$,t_2,…)
method instance **m**

task list T=($\mathbf{u_1,…,u_k}$,t_2,…)

Einfaches Beispiel (Fortsetzung)



Startzustand:

(at home)
(cash 20)
(distance home park 8)

(travel home park)

Precond:
(at home)
(walking-distance Home park)
Succeed
Fail (distance > 5)

Precond:
(at home)
(have-taxi-fare home park)
Succeed
Succeed (we have 20E, and the fare is only 9.50E)

(!call-taxi home)
(!ride home park)
(!pay-driver home park)
(!walk home park)

Endzustand:
(at park)
(cash 10.50)
(distance home park 8)

Erweiterungen von SHOP

Die Implementation von SHOP erfordert eine Erweiterung der Syntax und Semantik. Diese wird im Kontext einer *Transport - Planungsdomäne* erklärt.

Domänenbeschreibung:

**Ein Agent möchte von einem Ort der Stadt zum anderen gelangen.
Dazu stehen ihm drei verschiedene Transportmittel zur Verfügung:
Taxi, Bus und zu Fuß.**

**Um das Taxi zu benutzen, muss es erst am Taxistand gerufen werden.
Dann wird die Strecke im Taxi gefahren und im Anschluss der Fahrer bezahlt
und zwar 1,50E plus 1E für jeden gefahrenen Kilometer.**

**Um mit dem Bus zu fahren, wartet man an der Bushaltestelle.
Dann muss der Bus erst herangewunken werden und die Fahrkarte
im Voraus für 1E erworben werden. Dann wird die Strecke zurückgelegt.**

**Zu Fuß läuft man einfach die Strecke, aber der Agent läuft bei gutem Wetter
freiwillig maximal 3 Kilometer, bei schlechtem nur einen halben.**

Erweiterungen von SHOP

Obwohl die obengenannte Domäne ziemlich einfach ist, haben die meisten KI - Planungssysteme große Schwierigkeiten (wegen der numerischen Berechnungen).

Im Gegensatz dazu kann die *erweiterte* SHOP-Version sie ziemlich leicht darstellen.

Erweiterungen:

- Der *tail* von einem Axiom oder die *Methodenvorbedingungen* können Atome der Form ***(eval e)*** enthalten, wobei *e* der auszuwertende Ausdruck ist.

z.B. *A*1 = (:- (have-taxi-fare ?dist)
((have-cash ?m)
(eval (>= ?m (+ 1.5 ?dist)))))

*M*1 = (:method (pay-driver ?fare)
((have-cash ?m)
(eval (>= ?m ?fare)))
`((!set-cash ?m ,(- ?m ?fare))))

Erweiterungen von SHOP

- Axiome können mehrere tails enthalten, um diese in „if - then else“ zu benutzen.

  Das Axiom **(:- *h t1 t2 t3 … tn*)** sagt aus, dass
  - h wahr ist, wenn t1 wahr ist
    (wobei die Fälle *t*2, …, *t*n werden nicht ausgeführt);
  - sonst, -//-, wenn t1 falsch ist, aber t2 wahr
    (wobei die Fälle *t*3, …, *t*n werden nicht ausgeführt);
  - sonst, -//-, wenn t1 und t2 falsch sind, aber t3 wahr
    (wobei die Fälle *t*4, …, *t*n werden nicht ausgeführt);
    ...
  - sonst, -//- wenn *t*1, …, *t*n-1 falsch sind, aber tn wahr

z.B. ***A*2 = (:- (walking-distance ?u ?v)**
((weather-is 'good)
(distance ?u ?v ?w)
(eval (<= ?w 3)))
((distance ?u ?v ?w)
(eval (<= ?w 0.5))))

Erweiterungen von SHOP

- Eine Methode kann mehrere Vorbedingungen und *tails* enthalten, um diese in den „if - then - else“ Ausdrücken zu nutzen.

  Die Methode **(: method *h c*1 *t*1 c2 *t*2 c3 *t*3 … *cn tn*)** sagt aus,

  - dass h zu t1 reduziert wird, wenn c1 wahr ist,
  - oder zu t2, wenn c1 falsch und c2 wahr sind,
  - oder t3 wenn c1 und c2 falsch und c3 wahr, ...
  - oder zu tn, wenn *c*1,…,*cn*–1 falsch sind und cn wahr.

Erweiterungen von SHOP

- Eine Methode kann mehrere Vorbedingungen und *tails* enthalten, um diese in den „if - then - else“ Ausdrücken zu nutzen.

z.B. ***M*3 = (:method (travel-to ?y)**
(:first
(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)))

Erweiterungen von SHOP

- Wenn das erste Element der Methodenvorbedingung oder *tail* des Axioms **first** ist, dann wird SHOP nicht für alle *satisfier* durchgeführt, sondern nach dem ersten gefundenen *satisfier* zurückgegeben.

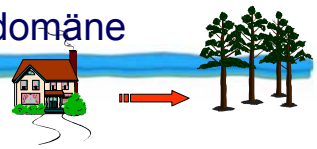
z.B. ***M*3 = (:method (travel-to ?y)**
(:first
(at ?x)
(at-taxi-stand ?t ?x)
(distance ?x ?y ?d)
(have-taxi-fare ?d))

`((!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)))

Transport - Planungsdomäne



- Startzustand: **((at downtown) (weather-is 'good) (have-cash 12)**
  (distance downtown park 2) nil)
  (distance downtown uptown 8)
  (distance downtown suburb 12)
  (at-taxi-stand taxi1 downtown)
  (at-taxi-stand taxi2 downtown)
  (bus-route bus1 downtown park)
  (bus-route bus2 downtown uptown)
  (bus-route bus3 downtown suburb))
- Methoden: (Aufgaben, Vorbedingungen, Unteraufgaben)
  - **(: 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)))

Transport - Planungsdomäne

- Methoden: (Aufgaben, Vorbedingungen, Unteraufgaben)

```
(: method (travel-to ?y)
        (:first     (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)) )
```

Transport - Planungsdomäne

- Axiome:

```
(:- (have-taxi-fare ?dist)
        ( (have-cash ?m)
          (eval (>= ?m (+ 1.5 ?dist))) ) )

(:- (walking-distance ?u ?v)
        ( (weather-is 'good)
          (distance ?u ?v ?w)
          (eval (<= ?w 3)) )

        ( (distance ?u ?v ?w)
          (eval (<= ?w 0.5)) ) )
```

- Operatoren: (Aufgabe, delete - Liste, add - Liste)

```
(:operator (!wait-for ?bus ?location)
                ()
                ((at ?bus ?location)) )

(:operator (!hail ?vehicle ?location)
        ()
        ((at ?vehicle ?location)) )
```

Transport - Planungsdomäne

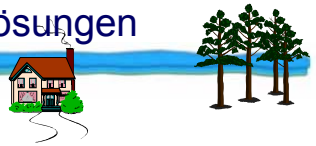
- Operatoren: (Aufgabe, delete - Liste, add - Liste)

```
(: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)))
```

Planungsprobleme und Lösungen

Problem:	**Lösung:**
1. Gehe in den Park bei gutem Wetter ohne Geld	**1. ((!WALK DOWNTOWN PARK))**
2. Gehe in den Park bei schlechtem Wetter ohne Geld	**2. None (can't afford a taxi or bus, and it's too far to walk).**
3. Gehe in den Park bei gutem Wetter mit 12E	**1. ((!WALK DOWNTOWN PARK)) 2. ((!HAIL TAXI1 DOWNTOWN) (!RIDE TAXI1 DOWNTOWN PARK) (!SET-CASH 12 8.5))**
4. Gehe in den Park bei gutem Wetter mit 80E	**1. ((!WALK DOWNTOWN PARK)) 2. ((!HAIL TAXI1 DOWNTOWN) (!RIDE TAXI1 DOWNTOWN PARK) (!SET-CASH 80 76.5))**

Planungsprobleme und Lösungen



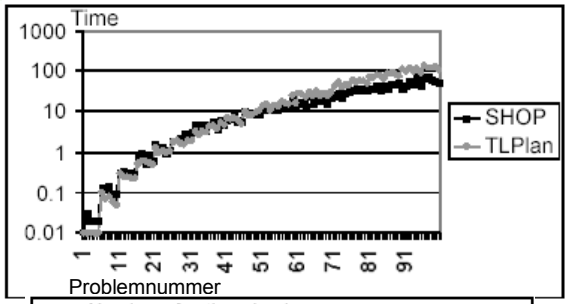
Problem:	Lösung:
5. Gehe nach Uptown bei gutem Wetter ohne Geld	None (can't afford a taxi or bus, and it's too far to walk).
6. Gehe nach Uptown bei gutem Wetter mit 12E	1. ((!HAIL TAXI1 DOWNTOWN) (!RIDE TAXI1 DOWNTOWN UPTOWN) (!SET-CASH 12 2.5))
7. Gehe nach Uptown bei gutem Wetter mit 80E	1. ((!HAIL TAXI1 DOWNTOWN) (!RIDE TAXI1 DOWNTOWN UPTOWN) (!SET-CASH 80 70.5))
8. Gehe in den Vorort bei gutem Wetter mit 12E	1. ((!WAIT- FOR BUS3 DOWNTOWN) (!SET-CASH 12 11.0) (!RIDE BUS3 DOWNTOWN SUBURB))

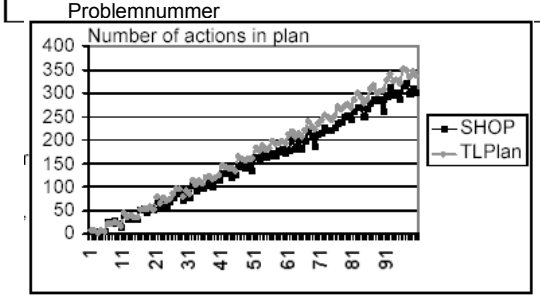
Experimentelle Untersuchungen

- Experimenteller Vergleich
  - **Planungsdomänen:**
    - Blockwelt - Domäne
    - Logistik - Domäne
    - UM Translog - Domäne
  - **Planungssysteme:**
    - Blackbox - Graphplan plus satisfiability
    - IPP - Graphplan plus ADL planning language
    - Tlplan - Vorwärtssuche mit den Regeln der modalen Logik
    - UMCP (Universal Method Composition Planner) - „klassischer" HTN - Planer
    - SHOP - System for Ordered Task Decomposition

Experimentelle Untersuchungen

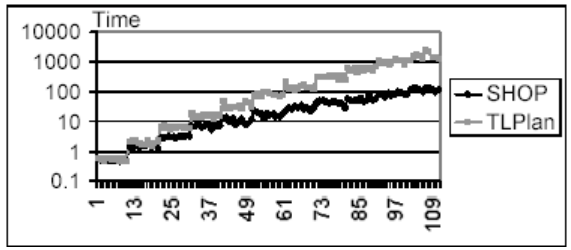
- **Block - Welt**
  - **100 zufällig generierte Probleme**
  - **167 - MHz Sun Ultra mit 64MB RAM**
  - **IPP und Blackbox konnten einige Planungsprobleme nicht llösen**

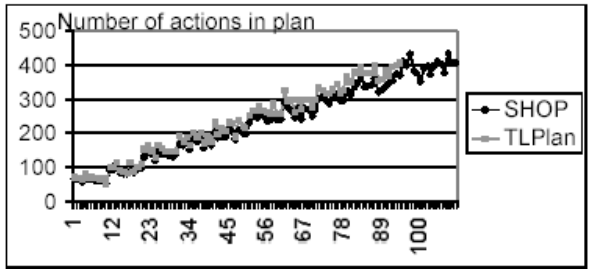




Experimentelle Untersuchungen

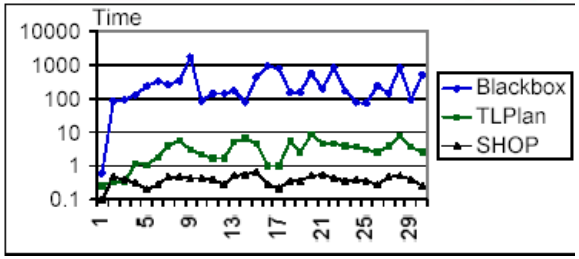
- **Logistik - Domäne**
  - **110 zufällig generierte Probleme**
  - **167 - MHz Sun Ultra mit 64MB RAM**
  - **IPP und Blackbox konnten einige Planungsprobleme nicht llösen**





Experimentelle Untersuchungen

- **Logistik - Domäne**
  - **Blackbox auf einem Satz von 30 Problemen**
  - **SHOP und Tlplan auf der gleichen Maschine**
  - **Blackbox auf einer schnelleren Maschine, mit 8 GB RAM**

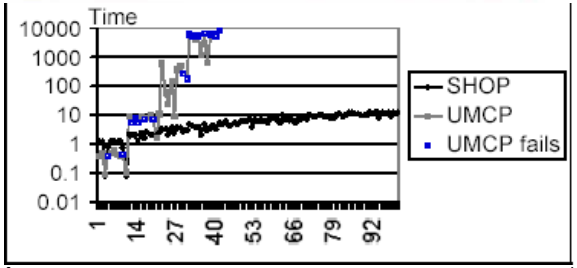


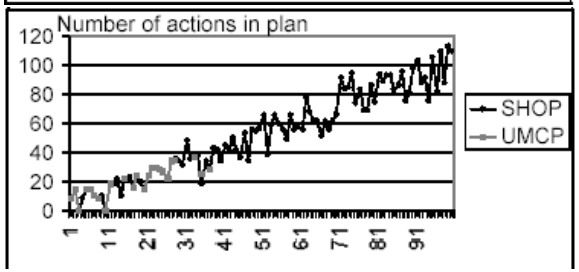
Experimentelle Untersuchungen

- **UM Translog - Domäne**
  - **HTN Planungsdomäne** [Andrews et al.,1995]
    - Inspiriert von der logistischen Domäne, ist aber wesentlich größer
    - Verschiedene Arten von Paketen, Transportmitteln und Prozeduren der Paketbehandlung
    - Besteht aus 43 Operatoren und 66 Methoden
  - **SHOP vs. UMCP**
    - UMCP [Erol,1994] - „klassischer“ HTN - Planer
  - **Tests für andere Planer konnten nicht durchgeführt werden**
    - Es war schwierig einige HTN - Konstruktionen in die Aktionen-basierte Darstellung zu transformieren

Experimentelle Untersuchungen

- **UM Translog - Domäne**
  - **100 zufällig generierte Probleme**
  - **167 - MHz Sun Ultra mit 64MB RAM**





Zusammenfassung

- Ordered Task Decomposition ist eine Adaption von der HTN - Planung
- **SHOP**: Domänen- unabhängiger Planungsalgorithmus
  - korrekt und vollständig in einer großen Klasse von Problemen
  - mächtig genug, um in den komplexen Anwendungen genutzt zu werden:
    - **NEO** (**N**oncombatant **E**vacuation **O**perations) -militärische Evakuierung
    - **HICAP** (**Hi**erarchical **I**nteractive **C**ase-based **A**rchitecture for **P**lanning) -ein Autorensystem für NEO

Zusammenfassung

- Viele Jahre war unter den KI Planungsforschern eine Meinung verbreitet, daß die Vorwärtssuche keine gute Art war, Pläne zu generieren.
  - Neue Ergebnisse widerlegen diese Behauptung. (SHOP, Tlplan, FF)
- *Entwicklung von SHOP:*
  - SHOP - die Domänen- unabhängige Implementation
  - JSHOP - benutzt den gleichen Algorithmus wie SHOP
    - geschrieben in Java
  - M-SHOP - generalisiert den SHOP - Algorithmus
    - partielle Ordnung in den Task List möglich
  - SHOP2 - generalisiert SHOP und M-SHOP
    - partielle Ordnung in den Task List und in den Unteraufgaben möglich

Quellen/Literatur

- [Nau et al., 2000] D.S. Nau, Y. Cao, A.Lotem and H. Munoz - Avila. SHOP and M-SHOP: Planning with Ordered Task Decomposition. Tech.Report CS TR 4157, University of Maryland: College Park, MD, 2000. <http://www.cs.umd.edu/~nau/papers/shop-ijcai99.pdf>
- [Nau et al., 2000] D.S. Nau, Y. Cao, A.Lotem and H. Munoz - Avila. Total-Order Planning with Partially Ordered Subtasks. Seventeenth International Joint Conference on Artificial Intelligence (IJCAI -2001), Seattle, August 2001
- Dana S. Nau Ordered Task Decomposition: Theory and Practice Institute for Systems Research, University of Maryland, College Park, MD
- J.Hertzenberg Planen: Einführung in die Planerstellungsmethoden der Künstlichen Intelligenz; Wien; Zürich: BI- Wiss .- Verl., 1989