UMELÁ INTELIGENCIA
Plánovanie

Plánovanie a rozvrhovanie

- **Plánovanie**: je daný jeden alebo viacero cieľov, vytvor postupnosť akcií vedúcich k jeho/ich splneniu
- **Rozvrhovanie (Scheduling)**: je daná množina akcií a ohraničení, prideľ zdroje a priraď časy akciám tak, aby neboli porušené ohraničenia
- Tradične, plánuje sa špecializovanými logickými metódami
- Tradične, rozvrhuje sa spĺňaním ohraničení, lineárnym programovaním alebo metódami operačného výskumu
- Avšak plánovanie a rozvrhovanie sú natoľko blízke úlohy, že sa nie vždy dajú oddeliť.

Plánovanie

- plánovanie v kontexte umelej inteligencie
- vyjadrenie problému plánovania
- potreba odhaliť spojitosti priamo medzi stavmi a akciami

  (pod)cieľ: *Mám*(*Mlieko*)

  akcia *Kúp*(*x*) s dôsledkom *Mám*(*x*)

  plánovaná akcia: *Kúp*(*Mlieko*)

  (ale napr. netreba uvažovať o

  Kúp(*Šampón-a-Vymývač-Hláv-v-Jednom*)

  Odstrihni-sa)
- stratégia „rozdeľuj a panuj“

Plánovanie

```
function JEDNODUCHÝ-PLÁNOVACÍ-AGENT(vnem) returns akcia
    static: BP, báza poznatkov (sú v nej aj opisy akcií)
            plán, plán, na začiatku NoPlan
            t, počítadlo, na začiatku 0, udáva čas
    local variables: cieľ, cieľ
                     stav, opis súčasného stavu

    PRIDAJ(BP, VYTVOR-VETU-O-VNEME(vnem, t))
    stav ← OPIS-STAVU(BP, t)
    if plán = NoPlan then
       cieľ ←ODPOVEDAJ(BP,VYTVOR-DOPYT-NA-CIEĽ(t))
       plán ← IDEÁLNY-PLÁNOVAČ(stav, cieľ, BP)
    if plán = NoPlan or plán je prázdny then akcia ← NoOp
    else
       akcia ← VYBER-PRVÚ(plán)
       plán  ← ZVYŠOK-AKCIÍ(plán)
    PRIDAJ(BP, VYTVOR-VETU-O-AKCII(akcia, t)
    t ← t + 1
return akcia
```


Reprezentácia plánovacieho problému

- reprezentácia stavov a cieľov

  stav: *NachádzamSa*(*Doma*)∧¬*Mám*(*Mlieko*) ∧ ¬*Mám*(*Rožky*) ∧ ¬*Mám*(*Noviny*) ∧ …

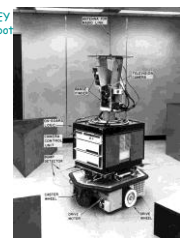
  cieľ: *NachádzamSa*(*Doma*) ∧ *Mám*(*Mlieko*) ∧ *Mám*(*Rožky*) ∧ *Mám*(*Noviny*)

  NachádzamSa(*x*) ∧ *PredávaSa*(*x, Mlieko*)
- reprezentácia akcií
  - operátor (jazyk STRIPS)
    - opis akcie
    - predpodmienka
    - účinky

  Op(AKCIA: *Choď* (*tam*),
  PREDPOD: *NachádzamSa*(*tu*) ∧ *Cesta*(*tu, tam*),
  ÚČINKY: *NachádzamSa*(*tam*)∧¬*NachádzamSa*(*tu*))
  - aplikovateľnosť operátora

rozhodovanie sa o tom, ako reprezentovať znalosti

- vyjadrovacia schopnosť vs výpočtová efektívnosť
- STRIPS: jednoduchý, ale rozumne vyjadrovací plánovací jazyk založený na výrokovom počte
- STRIPS = Stanford Research Institute Problem Solver
- Stanford Research Institute, výskum 1966-1972, behal tam legendárny robot Šejkí



Shakey - jednoduché predikáty modelu sveta

PRIMITIVE PREDICATES FOR THE ROBOT'S WORLD MODEL

Table 1: PRIMITIVE PREDICATES FOR THE ROBOT'S WORLD MODEL*

jazyk STRIPS

Each STRIPS operator must be described in some convenient way. We characterize each operator in the repertoire by three entities: an *add function*, a *delete function*, and a *precondition wff*. The meanings of these entities are straightforward. An operator is

Table 1: STRIPS OPERATORS

Reprezentácia plánovacieho problému

- priestor situácií a priestor plánov
  - progresívny a regresívny plánovač
  - least commitment
- reprezentácia plánov
  - množina krokov
  - množina ohraničení, určujúcich usporiadanie krokov $S_i \prec S_j$
  - množina ohraničení na priradenia premenným $v = x$
  - množina príčinných spojení $S_i \rightarrow^c S_j$

Reprezentácia plánovacieho problému

- reprezentácia plánov (pokr.)

Začni $\prec$ *Skonči*

Op(AKCIA:*ZačniNákup*,
ÚČINKY:*NachádzamSa(Doma)* $\wedge$
PredávaSa(Trafika, Noviny) $\wedge$
PredávaSa(PredajňaPotravín, Mlieko) $\wedge$
PredávaSa(PredajňaPotravín, Rožky))

Op(AKCIA: *SkončiNákup*,
PREDPOD: *NachádzamSa(Doma)* $\wedge$
Mám(Noviny) $\wedge$ *Mám(Mlieko)* $\wedge$ *Mám(Rožky))*

Plán(KROKY: {S_1: *Op*(AKCIA: *ZačniNákup*),
S_2: *Op*(AKCIA: *SkončiNákup*)},
USPORIADANIA:{$S_1 \prec S_2$}
PRIRADENIA: { }
SPOJENIA: { })

Reprezentácia plánovacieho problému

- reprezentácia plánov (pokr.)
  - linearizácia plánu
- riešenia
  - úplný neprotirečivý plán
    - úplnosť
      Krok S_i dosiahne predpodmienku c kroku S_j ak:
      - $S_i \prec S_j$ a $c \in$ Účinky(S_i)
      - neexistuje krok S_k taký, že $(\neg c) \in$ Účinky(S_k) a $S_i \prec S_k \prec S_j$ v nejakej linearizácii plánu.
    - neprotirečivosť
      $S_i \prec S_j$ aj $S_j \prec S_i$
      $v = A$ aj $v = B$ (A a B sú rozdielne konštanty)

Plánovanie v situačnom počte

- Reprezentácia plánovacieho problému:
  - začiatočný stav
    NachádzamSa(*Doma*, S_0) $\wedge \neg$*Mám*(*Mlieko*, S_0)
    $\wedge \neg$*Mám*(*Rožky*, S_0) $\wedge \neg$*Mám*(*Noviny*, S_0)
  - cieľový stav
    $\exists s$ *NachádzamSa*(*Doma*, *s*) $\wedge$ *Mám*(*Mlieko*, *s*)
    $\wedge$ *Mám*(*Rožky*, *s*) $\wedge$ *Mám*(*Noviny*, *s*)
  - operátory
    $\forall a \forall s$ *Mám*(*Mlieko*, *Výsledok*(*a*, *s*)) $\Leftrightarrow$
    [(*a* = *Kúp*(*Mlieko*)
    $\wedge$ *NachádzamSa*(*PredajňaPotravín*, *s*)
    $\vee$ (*Mám*(*Mlieko*, *s*) $\wedge$ *a* $\neq$ *Rozlejem*(*Mlieko*))]

$\forall s$ *Výsledok*'([ ], *s*) = *s*
$\forall s$ *Výsledok*'([*a*| *p*], *s*) = *Výsledok*'(*p*, *Výsledok*(*a*, *s*))

Plánovanie v situačnom počte

- potom cieľový test:

  $NachádzamSa(Doma, Výsledok'(p, S_0))$
  $\wedge\ Mám(Mlieko, Výsledok'(p, S_0))$
  $\wedge\ Mám(Rožky, Výsledok'(p, S_0))$
  $\wedge\ Mám(Noviny, Výsledok'(p, S_0))$

- a plán:

  p = [*Choď* (*PredajňaPotravín*), *Kúp*(*Mlieko*), *Kúp*(*Rožky*), *Choď*(*Trafika*), *Kúp*(*Noviny*), *Choď*(*Doma*)]

Algoritmus čiastočne usporadúvajúceho plánovania

```
function ČUP(stav, cieľ, operátory) returns plán

   plán ← VYTVOR-MINIMÁLNY-PLÁN(stav, cieľ)
   loop do
        if RIEŠENIE?(plán) then return plán
        S_treba, cieľ ← ZVOĽ-PODCIEĽ(plán)
        VYBER-OPERÁTOR(plán,operátory,S_treba, cieľ)
        VYRIEŠ-OHROZENIA(plán)
   end
```

```
function ZVOĽ-PODCIEĽ(plán) returns S_treba, cieľ
        vezmi krok plánu S_treba z KROKY(plán)
                s predpodmienkou c, ktorá sa ešte nedosiahla
        return S_treba, c
```


Algoritmus čiastočne usporadúvajúceho plánovania

```
procedure VYBER-OPERÁTOR(plán, operátory, S_treba, c)
   choose krok S_pridaj z operátory alebo z KROKY(plán)
                                   taký, ktorý má za účinok c

   if taký krok neexistuje then fail
   pridaj príčinné spojenie S_pridaj →^c S_treba do SPOJENIA(plán)
   pridaj usporadúvajúce spojenie S_pridaj ≺ S_treba do USPORIADANIA(plán)

   if S_pridaj je krok, ktorý sa teraz pridal z operátory then
     pridaj S_pridaj do KROKY(plán)
     pridaj Začni ≺ S_pridaj ≺ Skonči do USPORIADANIA(plán)
```


Algoritmus čiastočne usporadúvajúceho plánovania

```
procedure VYRIEŠ-OHROZENIA(plán)
   for each S_hrozí ohrozujúci spojenie S_i →^c S_j z SPOJENIA(plán)
   do
     choose buď
       Predsunutie: pridaj S_hrozí ≺ S_i do USPORIADANIA(plán)
       Odsunutie:   pridaj S_j ≺ S_hrozí do USPORIADANIA(plán)
     if not ZLUČITEĽNÝ(plán) then fail
   end
```


Algoritmus čiastočne usporadúvajúceho plánovania

- regresívny plánovač
- ohrozenia množiny na priradenia premenným:

```
procedure VYBER-OPERÁTOR(plán, operátory, S_treba, c)
     choose krok S_pridaj z operátory alebo z KROKY(plán)
          taký, ktorý má za účinok c_pridaj
          taký, že u = Unify(c, c_pridaj, PRIRADENIA(plán))

     if taký krok neexistuje then fail
     pridaj u do PRIRADENIA(plán))
     pridaj príčinné spojenie S_pridaj →^c S_treba do SPOJENIA(plán)
     pridaj usporadúvajúce spojenie S_pridaj ≺ S_treba do USPORIADANIA(plán)

     if S_pridaj je krok, ktorý sa teraz pridal z operátory then
     pridaj S_pridaj do KROKY(plán)
          pridaj Začni ≺ S_pridaj ≺ Skonči do USPORIADANIA(plán)
```


Algoritmus čiastočne usporadúvajúceho plánovania

```
procedure VYRIEŠ-OHROZENIA(plán)
   for each S_i →^c S_j in SPOJENIA(plán) do
     for each S_hrozí in KROKY(plán)  do
       for each c' in =ÚČINKY(S_hrozí) do

         if SUBST(PRIRADENIA(plán, c)) =
             SUBST(PRIRADENIA(plán), ¬c'), then
             choose buď
               Predsunutie:
                 pridaj S_hrozí ≺ S_i do USPORIADANIA(plán)
               Odsunutie:
                 pridaj S_j ≺ S_hrozí do USPORIADANIA(plán)
         if not ZLUČITEĽNÝ(plán) then fail

       end
     end
   end
```

Algoritmus čiastočne usporadúvajúceho plánovania

- rozšírenie definície dosiahnutia podmienky krokom v pláne

  Krok S_i dosiahne predpodmienku c kroku S_j ak:
  - $S_i \prec S_j$ a UNIFY(c, u) pre nejaký účinok $u \in$ ÚČINKY(S_i)
  - neexistuje krok S_k taký, že UNIFY(¬c, u) pre nejaký účinok $u \in$ ÚČINKY (S_k) a $S_i \prec S_k \prec S_j$ v nejakej linearizácii plánu

niektoré aplikácie plánovania pomocou umelej inteligencie

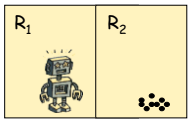
- vojenské operácie
- operácie v kontajnerových prístavoch
- konštrukčné úlohy
- autonómne riadenie družíc a iných vesmírnych lodí

plánovacie domény skutočného sveta

- domény skutočného sveta sú zložité a nemusia spĺňať predpoklady jazyka ako je STRIPS alebo metód ako je čup
- v skutočnom svete sa možno budeme musieť boriť s takýmito charakteristikami:
  - Modelovanie a usudzovanie o **zdrojoch**
  - Reprezentovanie a usudzovanie o **čase**
  - Plánovanie na rôznych úrovniach **abstrakcie**

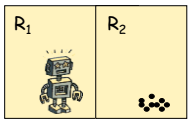
  } a.k.a. rozvrhovanie!
  - **Podmienené** výsledky akcií
  - **Neurčité** výsledky akcií
  - **Vonkajšie** (zvonku pôsobiace, exogénne) udalosti
  - **Inkrementálne navrhovanie plánu**
  - **Dynamické preplánovanie v reálnom čase**

príklad: robot vysávač



- dve miestnosti: R_1 and R_2
- robot vysávač
- prach

reprezentovanie stavu

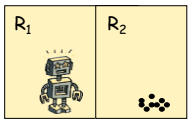


$In(Robot, R_1) \wedge Clean(R_1)$

výroky/tvrdenia, ktoré "platia" (t.j. sú pravdivé) v súčasnom stave

logická spojka "a"

reprezentovanie stavu



$In(Robot, R_1) \wedge Clean(R_1)$

- konjunkcia tvrdení
- nie je možné negované tvrdenie napr. $\neg Clean(R_2)$
- predpoklad uzavretého sveta: Každé tvrdenie, ktoré sa v danom stave neuvádza, je v tom stave nepravdivé
- nie je logická spojka "alebo" $In(Robot,R_1) \vee In(Robot,R_2)$
- nie sú premenné, napr. $\exists x\ Clean(x)$

reprezentovanie cieľa

príklad: $Clean(R_1) \wedge Clean(R_2)$

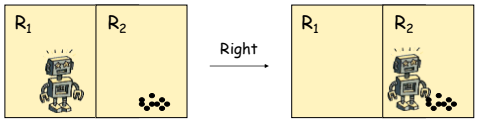
- konjunkcia tvrdení
- neobsahuje negované tvrdenie
- neobsahuje spojku "alebo"
- neobsahuje premennú

Cieľ G sa dosiahne v stave S, ak všetky tvrdenia v G (nazývané podciele) sú tiež v S

reprezentovanie akcií

Right

- Precondition = $In(Robot, R_1)$
- Delete-list = $In(Robot, R_1)$
- Add-list = $In(Robot, R_2)$



$In(Robot, R_1) \wedge Clean(R_1)$ $In(Robot, R_2) \wedge Clean(R_1)$

reprezentovanie akcií

Right

- Precondition = $In(Robot, R_1)$
- Delete-list = $In(Robot, R_1)$
- Add-list = $In(Robot, R_2)$

množiny tvrdení

rovnaký tvar ako ciele: konjunkcia tvrdení

reprezentovanie akcií

Right

- Precondition = $In(Robot, R_1)$
- Delete-list = $In(Robot, R_1)$
- Add-list = $In(Robot, R_2)$

- Akcia A je aplikovateľná v stave S, ak tvrdenia v jej predpodmienke sú všetky v S
- Aplikovaním akcie A na stav S vzniká nový stav taký, že sa vymažú z S tvrdenia uvedené v zozname na zrušenie (delete-list) a pridajú sa tvrdenia uvedené v zozname na pridanie (add-list)

ďalšie akcie

Left

- P = $In(Robot, R_2)$
- D = $In(Robot, R_2)$
- A = $In(Robot, R_1)$

Suck(R_1)

- P = $In(Robot, R_1)$
- D = ∅ [empty list]
- A = $Clean(R_1)$

Suck(R_2)

- P = $In(Robot, R_2)$
- D = ∅ [empty list]
- A = $Clean(R_2)$

ďalšie akcie

Left

- P = $In(Robot, R_2)$
- D = $In(Robot, R_2)$
- A = $In(Robot, R_1)$

Suck(r)

- P = $In(Robot, r)$
- D = ∅ [prázdny zoznam]
- A = $Clean(r)$

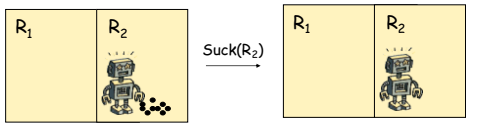
akciová schéma

jedna schéma opisuje viacero akcií, v tomto príklade:
Suck(R_1) and Suck(R_2)

Parameter, ktorý sa nainštaluje prirovnaním (matching) predpodmienky a stavu

Suck(r)
- P = In(Robot, r)
- D = ∅
- A = Clean(r)

akciová schéma



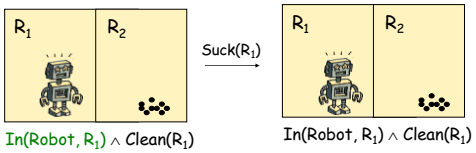
In(Robot, R_2) ∧ Clean(R_1)

In(Robot, R_2) ∧ Clean(R_1) ∧ Clean(R_2)

r ← R_2

Suck(r)
- P = In(Robot, r)
- D = ∅
- A = Clean(r)

akciová schéma



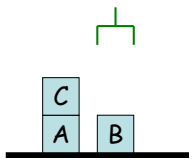
In(Robot, R_1) ∧ Clean(R_1)

In(Robot, R_1) ∧ Clean(R_1)

r ← R_1

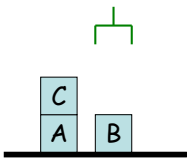
Suck(r)
- P = In(Robot, r)
- D = ∅
- A = Clean(r)

príklad: svet kociek



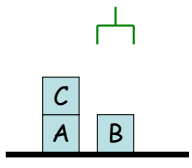
- robot dokáže ramenom pohybovať kocky na stole
- rameno nemôže držať viac než jednu kocku v jednom okamihu
- nijaké dve kocky sa nezmestia priamo na tú istú kocku
- stôl je ľubovoľne veľký

stav



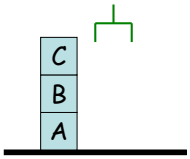
Block(A) ∧ Block(B) ∧ Block(C) ∧
On(A,stôl) ∧ On(B,stôl) ∧ On(C,A) ∧
Clear(B) ∧ Clear(C) ∧ Handempty

stav



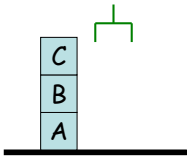
kocka(A) ∧ kocka(B) ∧ kocka(C) ∧
Na(A,stôl) ∧ Na(B,stôl) ∧ Na(C,A) ∧
čistá(B) ∧ čistá(C) ∧ prázdneRameno

cieľ



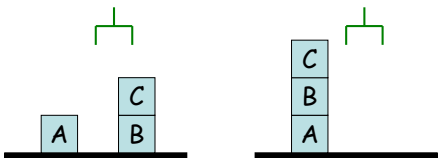
On(A,stôl) ∧ On(B,A) ∧ On(C,B) ∧ Clear(C)

cieľ



On(A,stôl) ∧ On(B,A) ∧ On(C,B) ∧ Clear(C)

cieľ



On(A,stôl) ∧ On(C,B)

akcia

Unstack(x,y)
P = Handempty∧ Block(x) ∧ Block(y) ∧ Clear(x) ∧ On(x,y)
D = Handempty, Clear(x), On(x,y)
A = Holding(x), Clear(y)

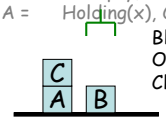
akcia

Unstack(x,y)
P = Handempty∧ Block(x) ∧ Block(y) ∧ Clear(x) ∧ On(x,y)
D = Handempty, Clear(x), On(x,y)
A = Holding(x), Clear(y)

vlastne netreba, iba ak má robot viac ramien

akcia

Unstack(x,y)
P = Handempty∧ Block(x) ∧ Block(y) ∧ Clear(x) ∧ On(x,y)
D = Handempty, Clear(x), On(x,y)
A = Holding(x), Clear(y)



Block(A) ∧ Block(B) ∧ Block(C) ∧
On(A,stôl) ∧ On(B,stôl) ∧ On(C,A) ∧
Clear(B) ∧ Clear(C) ∧ Handempty

Unstack(C,A)
P = Handempty∧ Block(C) ∧ Block(A) ∧ Clear(C) ∧ On(C,A)
D = Handempty, Clear(C), On(C,A)
A = Holding(C), Clear(A)

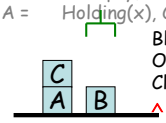
akcia

Unstack(x,y)

P = Handempty∧ Block(x) ∧ Block(y) ∧ Clear(x) ∧ On(x,y)

D = Handempty, Clear(x), On(x,y)

A = Holding(x), Clear(y)



Block(A) ∧ Block(B) ∧ Block(C) ∧ On(A,stôl) ∧ On(B,dtôl) ∧ On(C,A) ∧ Clear(B) ∧ ~~Clear(C)~~ ∧ ~~Handempty~~ ∧ Holding(C) ∧ Clear(A)

Unstack(C,A)

P = Handempty∧ Block(C) ∧ Block(A) ∧ Clear(C) ∧ On(C,A)

D = Handempty, Clear(C), On(C,A)

A = Holding(C), Clear(A)

všetky akcie

Unstack(x,y)

P = Handempty ∧ Block(x) ∧ Block(y) ∧ Clear(x) ∧ On(x,y)

D = Handempty, Clear(x), On(x,y)

A = Holding(x), Clear(y)

Stack(x,y)

P = Holding(x) ∧ Block(x) ∧ Block(y) ∧ Clear(y)

D = Clear(y), Holding(x)

A = On(x,y), Clear(x), Handempty

Pickup(x)

P = Handempty ∧ Block(x) ∧ Clear(x) ∧ On(x,Table)

D = Handempty, Clear(x), On(x,Table)

A = Holding(x)

Putdown(x)

P = Holding(x), ∧ Block(x)

D = Holding(x)

A = On(x,Table), Clear(x), Handempty

všetky akcie

Unstack(x,y)

P = Handempty ∧ Block(x) ∧ Block(y) ∧ Clear(x) ∧ On(x,y)

D = Handempty, Clear(x), On(x,y)

A = Holding(x), Clear(x)

Stack(x,y)

P = Holding(x) ∧ Block(x) ∧ Block(y) ∧ Clear(y)

D = Clear(y), Holding(x),

A = On(x,y), Clear(x), Handempty

Pickup(x)

P = Handempty ∧ Block(x) ∧ Clear(x) ∧ On(x,Table)

D = Handempty, Clear(x), On(x,Table)

A = Holding(x)

Putdown(x)

P = Holding(x), ∧ Block(x)

D = Holding(x)

A = On(x,Table), Clear(x), Handempty

vlastne netreba, iba ak má robot viac ramien

všetky akcie

Unstack(x,y)

P = Handempty ∧ Block(x) ∧ Block(y) ∧ Clear(x) ∧ On(x,y)

D = Handempty, Clear(x), On(x,y)

A = Holding(x), Clear(y)

Stack(x,y)

P = Holding(x) ∧ Block(x) ∧ Block(y) ∧ Clear(y)

D = Clear(y), Holding(x),

A = On(x,y), Clear(x), Handempty

Pickup(x)

P = Handempty ∧ Block(x) ∧ Clear(x) ∧ On(x,Table)

D = Handempty, Clear(x), On(x,TABLE)

A = Holding(x)

Putdown(x)

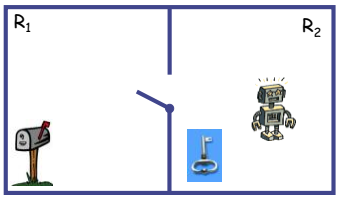
P = Holding(x), ∧ Block(x)

D = Holding(x)

A = On(x,TABLE), Clear(x), Handempty

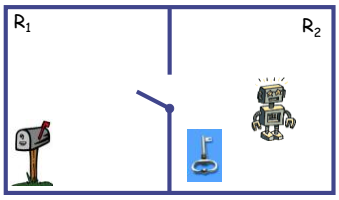
kocka sa vždy zmestí na stôl

príklad: kľúč v schránke



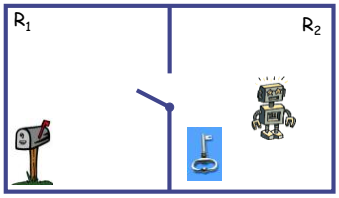
- robot musí zamknúť dvere a vložiť kľúč do schránky
- kľúč treba na zamknutie a odomknutie dverí
- keď sa raz kľúč dostane do schránky, robot ho už nedokáže dostať späť

príklad: kľúč v schránke: začiatočný stav



$In(Robot,R_2) \wedge In(Kľúč,R_2) \wedge Unlocked(Dvere)$

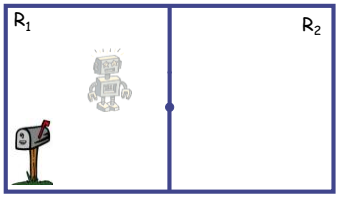
príklad: kľúč v schránke: začiatočný stav



v(Robot,R_2) ∧ v(Kľúč,R_2) ∧ odomknuté(Dvere)

49

príklad: kľúč v schránke: cieľ

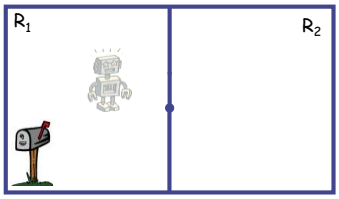


Locked(Dvere) ∧ In(Kľúč,Box)

[miesto robota sa v cieli nešpecifikuje]

50

príklad: kľúč v schránke: cieľ



zamknuté(Dvere) ∧ v(Kľúč,Schránka)

[miesto robota sa v cieli nešpecifikuje]

51

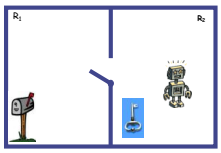
akcie

Grasp-Key-in-R_2
P = In(Robot,R_2) ∧ In(Key,R_2)
D = ∅
A = Holding(Key)

Lock-Door
P = Holding(Key)
D = ∅
A = Locked(Door)

Move-Key-from-R_2-into-R_1
P = In(Robot,R_2) ∧ Holding(Key) ∧ Unlocked(Door)
D = In(Robot,R_2), In(Key,R_2)
A = In(Robot,R_1), In(Key,R_1)

Put-Key-Into-Box
P = In(Robot,R_1) ∧ Holding(Key)
D = Holding(Key), In(Key,R_1)
A = In(Key,Box)



52

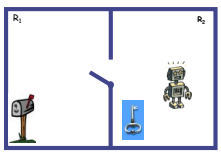
akcie

Uchop-Kľúč-v-R_2
P = v(Robot,R_2) ∧ v(Key,R_2)
D = ∅
A = Drží(Kľúč)

Zamkni-Dvere
P = Drží(Kľúč)
D = ∅
A = Zamknuté(Dvere)

Presuň-Kľúč-z-R_2-do-R_1
P = v(Robot,R_2) ∧ Drží(Kľúč) ∧ Odomknuté(Dvere)
D = v(Robot,R_2), v(Kľúč,R_2)
A = v(Robot,R_1), v(Kľúč,R_1)

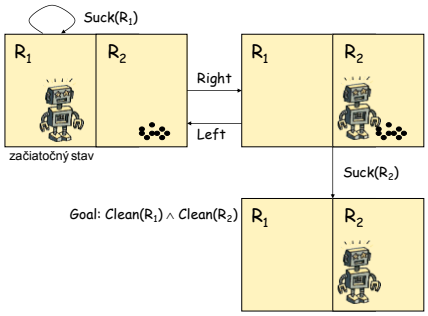
Vlož-Kľúč-do-Schránky
P = v(Robot,R_1) ∧ Drží(Kľúč)
D = Drží(Kľúč), v(Kľúč,R_1)
A = v(Key,Box)



53

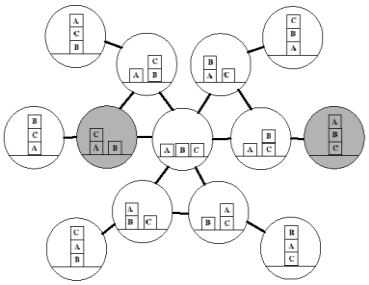
Metódy plánovania

dopredné plánovanie

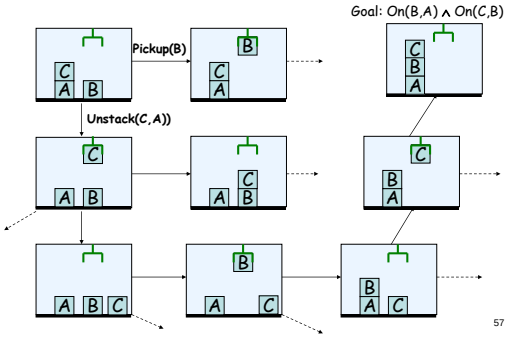


55

priestor hľadania: svet kociek



dopredné plánovanie



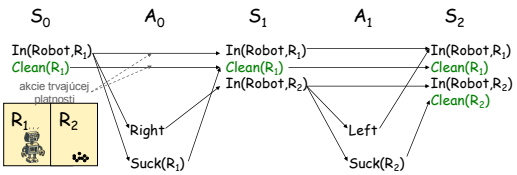
57

potreba presnej heuristiky

- Dopredné plánovanie jednoducho prehľadáva priestor stavov sveta od začiatočného k cieľovému stavu
- Predstavme si agenta s veľkou knižnicou akcií, ktorých cieľom je G, napr., G = Mať(Mlieko)
- Vo všeobecnosti sa dá v každom stave aplikovať mnoho akcií, takže faktor vetvenia je obrovský
- V ľubovoľnom stave je väčšina akcií irrelevantná z pohľadu dosiahnutia cieľa Mať(Mlieko)
- Našťastie, dá sa počítať konzistentná heuristika použitím plánovacích grafov

58

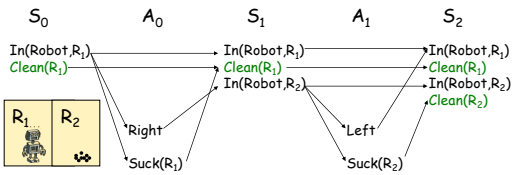
plánovací graf pre stav robota vysávača



- S_0 obsahuje tvrdenia o stave (tu, o začiatočnom stave)
- A_0 obsahuje všetky akcie, ktorých predpodmienky sa vyskytujú v S_0
- S_1 obsahuje všetky tvrdenia, ktoré boli v S_0 alebo sú v zoznamoch na pridanie akcií v A_0
- V dôsledku toho S_1 obsahuje všetky tvrdenia, ktoré by mohli byť pravdivé v stave dosiahnutom po prvej akcii
- A_1 obsahuje všetky akcie, ktoré už nie sú v A_0 , ktorých predpodmienky sú v S_1, čiže môžu sa vykonať v stave dosiahnutom po vykonaní prvej akcie. Atď...

59

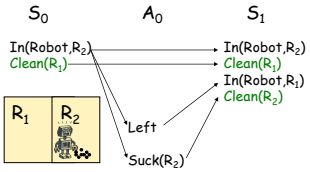
plánovací graf pre stav robota vysávača



- Hodnota i taká, že S_i obsahuje všetky cieľové tvrdenia, sa nazýva úrovňová cena (level cost) cieľa (tu i=2)
- Tak, ako sa zostrojuje plánovací graf, je to dolná hranica počtu akcií potrebných na dosiahnutie cieľa
- V tomto príklade je 2 aj skutočná dĺžka najkratšej cesty do cieľa

60

plánovací graf pre iný stav robota vysávača

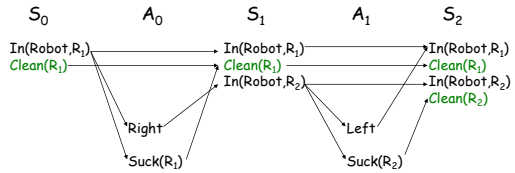


- úrovňová cena cieľa je 1 a je to opäť aj skutočná dĺžka najkratšej cesty do cieľa

použitie plánovacích grafov pri doprednom plánovaní

- Kedykoľvek sa vygeneruje nový uzol, vypočítaj plánovací graf jeho stavu [oprav/update plánovací graf pri rodičovskom uzle]
- Skonči počítanie plánovacieho grafu ak:
  - buď cieľové tvrdenia sú v množine S_i
    [vtedy i je úrovňová cena cieľa]
  - alebo ak $S_{i+1} = S_i$
    [vtedy vygenerovaný uzol nie je na ceste riešenia]
- Nastav hodnotu heuristickej funkcie h(N) pre uzol N na úrovňovú cenu cieľa pre stav reprezentovaný uzlom N
- h je konzistentná heuristika pre akcie s jednotkovou cenou
- čiže A* s použitím takejto h bude nachádzať riešenia s minimálnym počtom akcií

veľkosť plánovacieho grafu



- Nejaká akcia sa vyskytuje najviac raz
- Tvrdenie sa pridáva najviac raz a každý S_k ($k \neq i$) je ostrou nadmnožinou S_{k-1}
- Takže počet úrovní je ohraničený
  Min{počet akcií, počet tvrdení}
- Naproti tomu stavový priestor rastie exponenciálne s počtom tvrdení
- Počítanie plánovacích grafov môže ušetriť mnoho nepotrebného prehľadávania

zlepšenie plánovacieho grafu: vzájomné vylúčenia

- Cieľ (čo chceme dosiahnuť): zjemnime vyjadrenie úrovňovej ceny cieľa, aby bola presnejším odhadom počtu akcií, ktoré treba na dosiahnutie cieľa
- Metóda: rozpoznať zjavné vylúčenia medzi tvrdeniami na rovnakej úrovni
- Zvyčajne to vedie k presnejším heuristikám, ale plánovacie grafy budú väčšie a ich počítanie bude drahšie (bude dlhšie trvať)

dopredné plánovanie a jeho nedostatky

- napriek všetkým snahám, dopredné plánovanie je spojené s veľkým faktorom vetvenia
- vo všeobecnosti je omnoho menej akcií, ktoré môžu prispieť k naplneniu cieľa (sú relevantné) než akcií, ktoré sa môžu v nejakom stave vykonať
- lepšie je potom začať od cieľa
- Ako určiť, ktoré akcie sú relevantné? Ako ich použiť?
- → spätné plánovanie

akcia relevantná k cieľu

- akcia je relevantná k dosiahnutiu cieľa ak tvrdenie v jej opise.add.list sa pripodobí (match) nejakému podcieľu (čo je tiež nejaké tvrdenie)
- napr.:

Stack(B,A)

P = Holding(B) ∧ Block(B) ∧ Block(A) ∧ Clear(A)
D = Clear(A), Holding(B),
A = On(B,A), Clear(B), Handempty

je relevantná k dosiahnutiu
On(B,A)∧On(C,B)

Regresia cieľa

regresia cieľa G cez akciu A je najmenej ohraničujúca predpodmienka R[G,A] taká, že:

Ak stav S dosiahne R[G,A], tak:
1. predpodmienka akcie A je splnená v S
2. aplikovaním akcie A v stave S sa dosiahne stav, v ktorom je splnený G

príklad

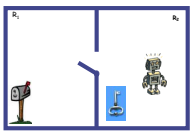
- $G = On(B,A) \wedge On(C,B)$
- **Stack(C,B)**
  P = Holding(C) ∧ Block(C) ∧ Block(B) ∧ Clear(B)
  D = Clear(B), Holding(C)
  A = On(C,B), Clear(C), Handempty
- R[G,Stack(C,B)] =
  On(B,A) ∧
  Holding(C) ∧ Block(C) ∧ Block(B) ∧ Clear(B)

príklad

- $G = On(B,A) \wedge On(C,B)$
- **Stack(C,B)**
  P = Holding(C) ∧ Block(C) ∧ Block(B) ∧ Clear(B)
  D = Clear(B), Holding(C)
  A = On(C,B), Clear(C), Handempty
- R[G,Stack(C,B)] =
  On(B,A) ∧
  Holding(C) ∧ Block(C) ∧ Block(B) ∧ Clear(B)

ďalší príklad

- G = In(key,Box) ∧ Holding(Key)
- **Put-Key-Into-Box**
  P = In(Robot,R_1) ∧ Holding(Key)
  D = Holding(Key), In(Key,R_1)
  A = In(Key,Box)



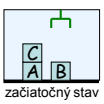
- R[G,Put-Key-Into-Box] = False
  kde False je nedosiahnuteľný cieľ
- To ale znamená, že In(key,Box) ∧ Holding(Key) sa nedá dosiahnuť vykonaním akcie **Put-Key-Into-Box**

postup počítania R[G,A]

1. ak je ľubovoľný podcieľ cieľa G v zozname na zrušenie v opise akcie A (delete list) tak return False
2. inak
   a. G' ← predpodmienka akcie A
   b. pre každý podcieľ SG cieľa G vykonaj
      ak SG nie je v zozname na pridanie v opise akcie A (add list) tak pridaj SG do G'
3. Return G'

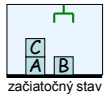
spätné plánovanie (backward planning)

On(B,A) ∧ On(C,B)



začiatočný stav

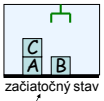
spätné plánovanie



začiatočný stav

On(B,A) ∧ On(C,B)
↓ Stack(C,B)
On(B,A) ∧ Holding(C) ∧ Clear(B)
↓ Pickup(C)
On(B,A) ∧ Clear(B) ∧ Handempty ∧ Clear(C) ∧ On(C,Table)
↓ Stack(B,A)
Clear(C) ∧ On(C,TABLE) ∧ Holding(B) ∧ Clear(A)
↓ Pickup(B)
Clear(C) ∧ On(C,Table) ∧ Clear(A) ∧ Handempty ∧ Clear(B) ∧ On(B,Table)
↓ Putdown(C)
Clear(A) ∧ Clear(B) ∧ On(B,Table) ∧ Holding(C)
↓ Unstack(C,A)
Clear(B) ∧ On(B,Table) ∧ Clear(C) ∧ Handempty ∧ On(C,A)

spätné plánovanie



začiatočný stav

On(B,A) ∧ On(C,B)
↓ Stack(C,B)
On(B,A) ∧ Holding(C) ∧ Clear(B)
↓ Pickup(C)
On(B,A) ∧ Clear(B) ∧ Handempty ∧ Clear(C) ∧ On(C,Table)
↓ Stack(B,A)
Clear(C) ∧ On(C,TABLE) ∧ Holding(B) ∧ Clear(A)
↓ Pickup(B)
Clear(C) ∧ On(C,Table) ∧ Clear(A) ∧ Handempty ∧ Clear(B) ∧ On(B,Table)
↓ Putdown(C)
Clear(A) ∧ Clear(B) ∧ On(B,Table) ∧ Holding(C)
↓ Unstack(C,A)
Clear(B) ∧ On(B,Table) ∧ Clear(C) ∧ Handempty ∧ On(C,A)

strom hľadania

- Spätné plánovanie prehľadáva priestor cieľov od pôvodného cieľa problému k cieľu, ktorý je splnený v začiatočnom stave
- Zvyčajne je omnoho menej akcií relevantných k nejakému cieľu, než je akcií, aplikovateľných v nejakom stave → menší faktor vetvenia než pri doprednom plánovaní
- Dĺžky ciest riešenia sú rovnaké

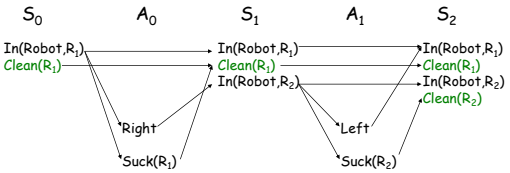
konzistentná heuristika pre spätné plánovanie

postup:

1. Pred-vypočítať plánovací graf začiatočného stavu až pokiaľ neklesne úroveň na nulu
2. pre každý uzol N, ktorý sa pridal do stromu hľadania, stanoviť hodnotu heuristickej funkcie h(N) ako hodnotu úrovňovej ceny cieľa združeného s uzlom N

Ak sa cieľ združený s uzlom N nedá splniť v nijakej množine S_k plánovacieho grafu, nedá sa vôbec dosiahnuť – useknúť ho!

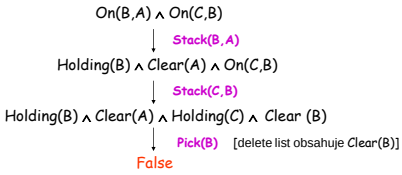
počíta sa len jeden plánovací graf



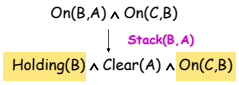
Ako spätné plánovanie rozpozná slepé vetvy?

On(B,A) ∧ On(C,B)
↓ Stack(C,B)

Ako spätné plánovanie rozpozná slepé vetvy?



Ako spätné plánovanie rozpozná slepé vetvy?



stavové ohraničenie napr.
Holding(x) → ¬(∃y)On(y,x)
by umožnilo useknúť túto vetvu skôr

Progresia: dopredné hľadanie (I -> G)

ProgWS(*stav, ciele, akcie, cesta*)
If *stav* spĺňa *ciele*, then return *cesta*
else *a* = ***choose***(*akcie*), takú, že
predpodmienky(*a*) sú splnené v *stave*
if nie je taká *a*, then return neúspech
else return
ProgWS(apply(*a*, *stav*), *ciele, akcie*,
pridaj(*cesta*, *a*))

prvé volanie: ProgWS(*IS*, *G, Akcie*, ())

Príklad dopredného hľadania

I: (on-table A) (on C A) (on-table B) (clear B) (clear C)
G: (on A B) (on B C)

- P(I, G, AkcieSvetaKociek, ())
- P(S1, G, ASK, (Unstack&Putdown(C, A))
- P(S2, G, ASK, (Unstack&Putdown(C, A), Stack(B, C))
- P(S3, G, ASK, (Unstack&Putdown(C, A), Stack(B, C), Stack(A, B))

nedeterministický výber!

G ⊆ S3 => úspech!

Regresia: spätné hľadanie (I <- G)

RegWS(*init-stav*, *súčasné-ciele, akcie, cesta*)
If *init-stav* spĺňa *súčasné-ciele*, then return *cesta*
else a =***choose*** (*akcie*), takú, že nejaký účinok *akcie* a spĺňa jeden zo *súčasných-cieľov*
If nie je taká akcia *a*, then return neúspech [nedosiahnuteľný*]
If nejaký účinok *a* je v rozpore s niektorým *súčasným-cieľom*, then return neúspech [nezlučiteľný stav]
CG' = *súčasné-ciele* – účinky(*a*) + predpodmienky(*a*)
If *súčasné-ciele* ⊂ CG', then return neúspech [zbytočné*]
RegWS(*init-stav*, CG', *akcie*, pridaj(*a,cesta*))

prvé volanie: RegWS(*IS, G, Akcie*, ())

príklad spätného hľadania

I: (on-table A) (on C A) (on-table B) (clear B) (clear C)
G: (on A B) (on B C)

- R(I, G, AkcieSvetaKociek, ())
- R(I, ((clear A) (on-table A) (clear B) (on B C)), ASK, (Stack(A, B)))
- R(I, ((clear A) (on-table A) (clear B) (clear C), (on-table B)), ASK, (Stack(B, C), Stack(A, B)))
- R(I, ((on-table A) (clear B) (clear C) (on-table B) (on C A)), ASK, (Unstack&Pushdown(C, A), Stack(B, C), Stack(A, B)))

nedeterministický výber!

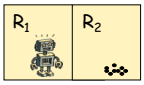
súčasné-ciele ⊆ I => úspech!

Progresia vs. Regresia

- oba algoritmy sú:
  - správne: výsledný plán je platný
  - úplné: ak existuje platný plán, nájdu ho
- nedeterministický výber => hľadanie!
  - neinformované: DFS, BFS, Iterative Deepening, ..,
  - Heuristické: A*, IDA*, …
- zložitosť: $O(b^n)$ v najhoršom prípade
  b = faktor vetvenia, n = |"choose"|
- Regresia: často menší faktor vetvenia b

Niektoré rozšírenia jazyka STRIPS

1. Negované tvrdenia v opise stavu



$In(Robot, R_1) \wedge \neg In(Robot, R_2) \wedge Clean(R_1) \wedge \neg Clean(R_2)$

Dump-Dirt(r)
P = In(Robot, r) ∧ Clean(r)
E = ¬Clean(r)

Suck(r)
P = In(Robot, r) ∧ ¬Clean(r)
E = Clean(r)

- Q v E znamená zrušiť ¬Q a pridať Q do stavu
- ¬Q v E znamená zrušiť Q a pridať ¬Q

Predpoklad otvoreného sveta: Tvrdenie v stave je pravdivé ak nie je negované, inak je nepravdivé. Pravdivosť tvrdenia, ktoré nie je v stave uvedené, je neznáma.

Metódy plánovania sa dajú relatívne ľahko rozšíriť tak, aby pracovali aj s negáciou. Len opisy stavov veľmi narastú (predstavme si, že by sa robot pohyboval v budove s 50 miestnosťami)

87

2. predikáty rovnosti/rôznosti

svet kociek:

Move(x,y,z)
P = Block(x) ∧ Block(y) ∧ Block(z) ∧ On(x,y) ∧ Clear(x) ∧ Clear(z) ∧ (x≠z)
D = On(x,y), Clear(z)
A = On(x,z), Clear(y)

Move(x,Table,z)
P = Block(x) ∧ Block(z) ∧ On(x,Table) ∧ Clear(x) ∧ Clear(z) ∧ (x≠z)
D = On(x,y), Clear(z)
A = On(x,z)

Move(x,y,Table)
P = Block(x) ∧ Block(y) ∧ On(x,y) ∧ Clear(x)
D = On(x,y)
A = On(x,Table), Clear(y)

88

2. predikáty rovnosti/rôznosti

svet kociek:

Move(x,y,z)
P = Block(x) ∧ Block(y) ∧ Block(z) ∧ On(x,y) ∧ Clear(x) ∧ Clear(z) ∧ (x≠z)
D = On(x,y), Clear(z)
A = On(x,z), Clear(y)

Move(x,Table,z)
P = Block(x) ∧ Block(z) ∧ O ∧ Clear(z) ∧ (x≠z)
D = On(x,y), Clear(z)
A = On(x,z)

Move(x,y,Table)
P = Block(x) ∧ Block(y) ∧ O
D = On(x,y)
A = On(x,Table), Clear(y)

Plánovacie metódy jednoducho vyhodnotia (x≠z) keď sú obe premenné nainštalované

To je ekvivalentné tomu, ako by sa uvažovalo, že tvrdenia (A ≠ B) , (A ≠ C) , ... sú implicitne pravdivé v každom stave

89

3. Algebraické výrazy

Dve fľaše F_1 a F_2 majú objemy 30 a 50
F_1 obsahuje 20 nejakej tekutiny
F_2 obsahuje 15 tej istej tekutiny

Stav:
$Cap(F_1,30) \wedge Cont(F_1,20) \wedge Cap(F_2, 50) \wedge Cont(F_2,15)$

Akcia prelievania (pour) obsahu jednej flašky do druhej:

Pour(f,f')
P = Cont(f,x) ∧ Cap(f',c') ∧ Cont(f',y)
D = Cont(f,x), Cont(f',y),
A = Cont(f,max{x+y-c',0}), Cont(f',min{x+y,c'})

90

3. Algebraické výrazy

Dve fľaše F_1 a F_2 majú objemy 30 a 50
F_1 obsahuje 20 nejakej tekutiny
F_2 obsahuje 15 tej sitej tekutiny

Stav:
Cap(F_1, [illegible])

Toto rozšírenie si vyžaduje, aby plánovacie metódy „vedeli" aj algebraické manipulácie

Akcia preli[illegible]

Pour(f,f')
P = Cont(f,x) ∧ Cap(f',c') ∧ Cont(f',y)
D = Cont(f,x), Cont(f',y),
A = Cont(f,max{x+y-c',0}), Cont(f',min{x+y,c'})

4. Stavové ohraničenia

h	b	
c	d	g
e	a	f

Stav:
Adj(1,2) ∧ Adj(2,1) ∧ ... ∧ Adj(8,9) ∧ Adj(9,8) ∧
At(h,1) ∧ At(b,2) ∧ At(c,4) ∧ ... ∧ At(f,9) ∧ Empty(3)

Move(x,y)
P = At(x,y) ∧ Empty(z) ∧ Adj(y,z)
D = At(x,y), Empty(z)
A = At(x,z), Empty(y)

4. Stavové ohraničenia

h	b	
c	d	g
e	a	f

Stav:
Adj(1,2) ∧ ~~Adj(2,1) ∧~~ ... ∧ Adj(8,9) ∧ ~~Adj(9,8) ∧~~
At(h,1) ∧ At(b,2) ∧ At(c,4) ∧ ... ∧ At(f,9) ∧ Empty(3)

Stavové ohraničenie:
Adj(x,y) → Adj(y,x)

Move(x,y)
P = At(x,y) ∧ Empty(z) ∧ Adj(y,z)
D = At(x,y), Empty(z)
A = At(x,z), Empty(y)

zložitejšie stavové ohraničenia v FOL

svet kociek:

(∀x)[Block(x) ∧ ¬(∃y)On(y,x) ∧ ¬Holding(x)] → Clear(x)

(∀x)[Block(x) ∧ Clear(x)] → ¬(∃y)On(y,x) ∧ ¬Holding(x)

Handempty ↔ ¬(∃x)Holding(x)

takéto tvrdenia by veľmi zjednodušili opisy akcií

Stavové ohraničenia si vyžadujú plánovacie metódy, ktoré „vedia" usudzovať, aby rozhodli, či sú ciele dosiahnuté a ohraničenia splnené