Plánování a rozvrhování

3

Roman Barták, KTIML

roman.bartak@mff.cuni.cz
http://ktiml.mff.cuni.cz/~bartak

Na úvod

Plánovací problém P je trojice (Σ, s_0, g)

- Σ je **plánovací doména** popisující stavy a akce (přechody mezi stavy)
- s_0 je **počáteční stav**
- g charakterizuje **cílové stavy**

Množinová reprezentace problému

- **stav** je množina výroků
- **akce** je trojice množin výroků (precond,effects$^-$,effects$^+$)
  - $\text{precond} \subseteq s \rightarrow (s - \text{effects}^-) \cup \text{effects}^+$

Klasická reprezentace problému

- **stav** je množina instanciovaných atomů
- **operátor** je trojice (name, precond, effects), kde precond a effects jsou množiny literálů
- **akce** je instancí operátoru
  - $\text{precond}^+ \subseteq s \ \wedge \ \text{precond}^- \cap s = \varnothing \rightarrow (s - \text{effects}^-) \cup \text{effects}^+$

Klasická reprezentace

- **Konstanty**
  - bloky: a,b,c,d,e
- **Predikáty:**
  - **ontable(*x*)** kostka *x* je na stole
  - **on(*x*,*y*)** kostka *x* je na kostce *y*
  - **clear(*x*)** kostka *x* na sobě nic nemá
  - **holding(*x*)** robotova ruka drží kostku *x*
  - **handempty** robotova ruka nic nedrží

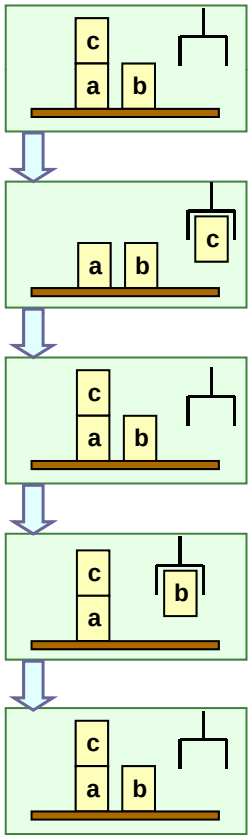
- **Akce**

unstack(*x*,*y*)
Precond: on(x,y), clear(x), handempty
Effects: ¬on(x,y), ¬clear(x), clear(y), ¬handempty, holding(x),

stack(*x*,*y*)
Precond: holding(x), clear(y)
Effects: ¬holding(x), ¬clear(y), on(x,y), clear(x), handempty

pickup(*x*)
Precond: ontable(x), clear(x), handempty
Effects: ¬ontable(x), ¬clear(x), ¬handempty, holding(x)

putdown(*x*)
Precond: holding(x)
Effects: ¬holding(x), ontable(x), clear(x), handempty



Množinová reprezentace

Výroky:
pro 5 kostek máme 36 výroků

- **ontable-a** kostka a je na stole (5x)
- **on-c-a** kostka c je na kostce a (20x)
- **clear-c** kostka c na sobě nic nemá (5x)
- **holding-d** robotova ruka drží kostku d (5x)
- **handempty** robotova ruka nic nedrží (1x)

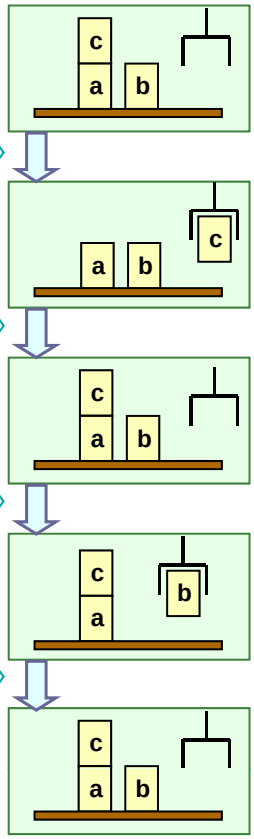
Akce
pro 5 kostek máme 50 akcí

unstack-c-a
Pre: on-c-a, clear-c, handempty
Del: on-c-a, clear-c, handempty
Add: holding-c, clear-a

stack-c-a
Pre: holding-c, clear-a
Del: holding-c, clear-a
Add: on-c-a, clear-c, handempty

pickup-b
Pre: ontable-b, clear-b, handempty
Del: ontable-b, clear-b, handempty
Add: holding-b

putdown-b
Pre: holding-b
Del: holding-b
Add: ontable-b, clear-b, handempty



Poznámky ke složitosti

- Jaká je složitost klasického plánování v klasické reprezentaci?
- **Rozhodnutelnost**

Funkční symboly	*Existence plánu*	*Plán do dané délky*
ne	ano	ano
ano	částečně	ano

Problém zastavení

- **Složitost**

Negativní efekty	*Negativní předpoklady*	*Existence plánu*	*Plán do dané délky*
ano	ano/ne	EXPSPACE-c	NEXPTIME-c
ne	ano	NEXPTIME-c	NEXPTIME-c
	ne	EXPTIME-c	NEXPTIME-c

Jak se dělá plánování?

- Téměř všechny plánovací algoritmy jsou založeny na prohledávaní.
- Jednotlivé algoritmy se liší tím, jaký prostor a jak konkrétně se prohledává.
  - **Plánování ve stavovém prostoru**
    - uzly odpovídají stavům
  - **Plánování v prostoru plánů**
    - uzly odpovídají částečně instanciovaným plánům

Plánování
ve stavovém prostoru

Plánování se stavy

- **Prohledávaný prostor odpovídá stavovému prostoru plánovacího problému.**
  - uzly odpovídají stavům
  - hrany odpovídají stavovým přechodům pomocí akcí
  - cílem je najít cestu mezi počátečním stavem a některým koncovým stavem
- **Typy prohledávání**
  - dopředné (forward search)
  - zpětné (backward search)
    - liftovaná verze
    - STRIPS
  - problémově závislé (svět kostek)
- Poznámka: algoritmy budeme uvádět pro klasickou reprezentaci.

Dopředné plánování

Začínáme v počátečním stavu a jdeme k některému stavu cílovému.

Je potřeba umět:

- rozhodnout, zda je daný **stav cílový** nebo ne
- najít množinu **akcí aplikovatelných** na daný stav
- vypočítat stav, do kterého se dostaneme **aplikací akce**

Algoritmus

Forward-search(O, s_0, g)

$s \leftarrow s_0$

$\pi \leftarrow$ the empty plan

loop

if s satisfies g then return π

$E \leftarrow \{a | a$ is a ground instance of an operator in O,

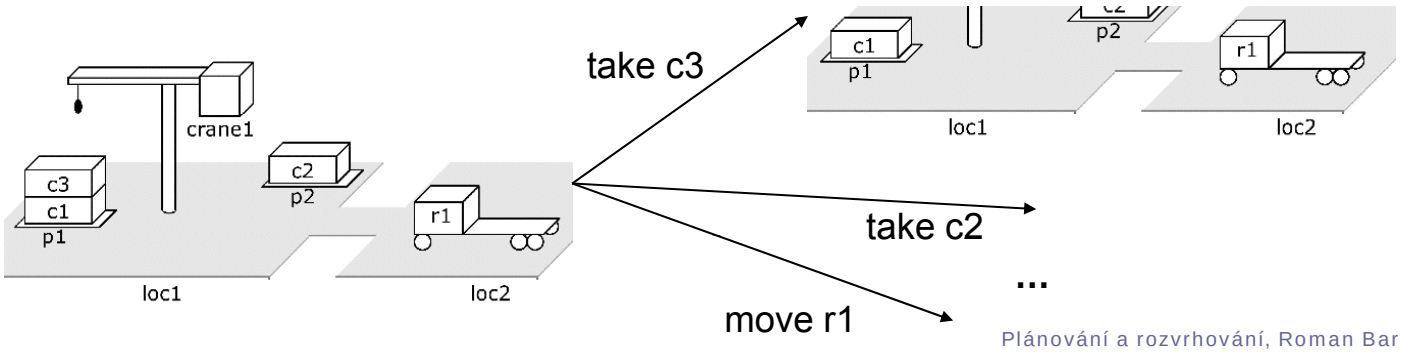
and $\text{precond}(a)$ is true in $s\}$

if $E = \emptyset$ then return failure

nondeterministically choose an action $a \in E$

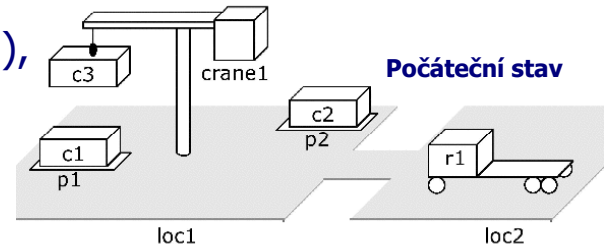
$s \leftarrow \gamma(s, a)$

$\pi \leftarrow \pi.a$



Příklad

{belong(crane1,loc1), adjacent(loc2,loc1), holding(crane1,c3), unloaded(r1), at(r1,loc2), ¬occupied(loc1), occupied(loc2),...}



Počáteční stav

move(r1,loc2,loc1)

move(r, l, m)
;; robot r moves from location l to location m
precond: adjacent(l, m), at(r, l), ¬occupied(m)
effects: at(r, m), occupied(m), ¬occupied(l), ¬at(r, l)

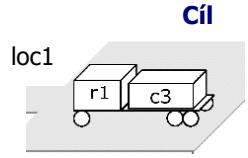
{belong(crane1,loc1), adjacent(loc2,loc1), holding(crane1,c3), unloaded(r1), at(r1,loc1), occupied(loc1), ...}

load(crane1,loc1,c3,r1)

load(k, l, c, r)
;; crane k at location l loads container c onto robot r
precond: belong(k, l), holding(k, c), at(r, l), unloaded(r)
effects: empty(k), ¬holding(k, c), loaded(r, c), ¬unloaded(r)

{belong(crane1,loc1), adjacent(loc2,loc1), empty(crane1), loaded(r1,c3), at(r1,loc1), occupied(loc1), ...}

Cíl = {at(r1,loc1),loaded(r1,c3)}



Vlastnosti

Procedura dopředného plánování je korektní.

- Pokud vrátí nějaký plán, potom je řešením.
- Stačí si uvědomit, že s = $\gamma(s_0,\pi)$.

Procedura dopředného plánování je úplná.

- Pokud existuje plán, potom alespoň jedna z větví nedeterminismu ho najde.
- indukcí podle délky plánu
- v každém kroku má algoritmus šanci zvolit správnou akci (pokud v předchozích krocích volil akce z plánu)

Determinismus

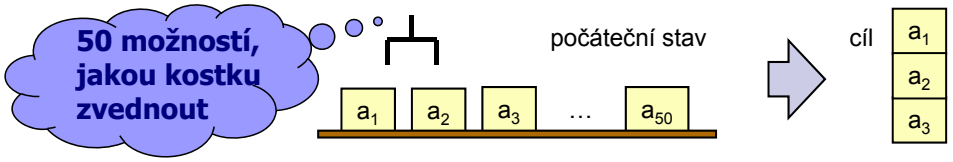
Algoritmus dopředného prohledávání můžeme implementovat deterministicky:

- **prohledávání do šířky** (breadth-first search)
  - korektní, úplné, ale paměťově náročné
- **prohledávání do hloubky** (depth-first search)
  - korektní, úplnost lze zařídit kontrolou cyklů (stav se na cestě neopakuje)
- **uspořádané prohledávání** (best-first search)
  - korektní, úplné, ale paměťově náročné
- **hladové prohledávání** (greedy search)
  - korektní a neúplné

Větvení

V čem je problém prohledávání dopředu?

Vysoký větvící faktor – počet možností k výběru



- To vadí u determinismu, který může ztrácet čas zkoušením irelevantních akcí.

Řešení:

- **heuristika** doporučující výběr akce
- **ořezání** prohledávaného prostoru
  - Např. pokud plány π_1 a π_2 dosáhly stejného stavu, potom víme, že také plány $\pi_1\pi_3$ a $\pi_2\pi_3$ dosáhnou stejného stavu. Delší z plánů π_1 a π_2 tedy nemusíme dále rozvíjet.
    Je potřeba si pamatovat všechny navštívené stavy ☹.

Zpětné plánování

Začínáme s cílem (pozor to není stav, ale reprezentace množiny stavů!) a jdeme přes podcíle k počátečnímu stavu.

Je potřeba umět:

- zjistit, zda daný **stav odpovídá cíli**
- pro daný cíl najít **relevantní akce**
- **vypočítat podcíl** umožňující aplikovat danou akci

Trochu opakování

Akce a je relevantní pro cíl g právě když:

- akce přispívá do g: $g \cap effects(a) \neq \emptyset$
- efekty akce nejsou v konfliktu s g:
  - $g^- \cap effects^+(a) = \emptyset$
  - $g^+ \cap effects^-(a) = \emptyset$

Regresní (zpětná) množina cíle g pro (relevantní) akci a:
$\gamma^{-1}(g,a) = (g - effects(a)) \cup precond(a)$

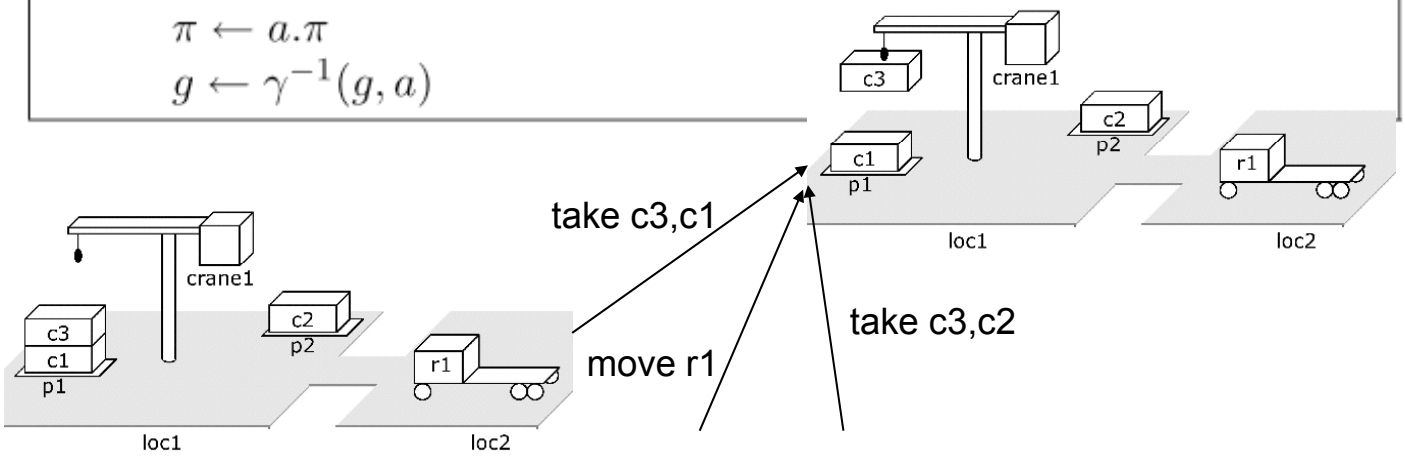
Příklad:

cíl: **{on(a,b), on(b,c)}**
akce **stack(a,b)** je relevantní
její „zpětnou aplikací" dostaneme nový cíl:
{holding(a), clear(b), on(b,c)}

stack(*x*,*y*)
Precond: holding(*x*), clear(*y*)
Effects: ~holding(*x*), ~clear(*y*),
on(*x*,*y*), clear(*x*), handempty

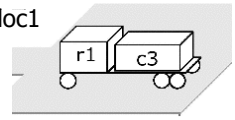
Algoritmus

Backward-search(O, s_0, g)
  $\pi \leftarrow$ the empty plan
  loop
    if s_0 satisfies g then return π
    $A \leftarrow \{a | a$ is a ground instance of an operator in O
      and $\gamma^{-1}(g, a)$ is defined$\}$
    if $A = \emptyset$ then return failure
    nondeterministically choose an action $a \in A$
    $\pi \leftarrow a.\pi$
    $g \leftarrow \gamma^{-1}(g, a)$



Příklad

Cíl = {at(r1,loc1),loaded(r1,c3)}



load(crane1,loc1,c3,r1)

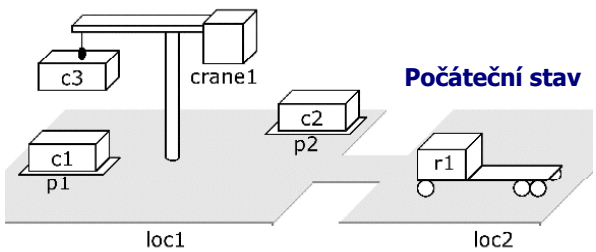
load(k, l, c, r)
  ;; crane k at location l loads container c onto robot r
  precond: belong(k, l), holding(k, c), at(r, l), unloaded(r)
  effects: empty(k), $\neg$ holding(k, c), loaded(r, c), $\neg$ unloaded(r)

{at(r1,loc1), belong(crane1,loc1),
holding(crane1,c3), unloaded(r1)}

move(r1,loc2,loc1)

move(r, l, m)
  ;; robot r moves from location l to location m
  precond: adjacent(l, m), at(r, l), $\neg$ occupied(m)
  effects: at(r, m), occupied(m), $\neg$ occupied(l), $\neg$ at(r, l)

{belong(crane1,loc1), holding(crane1,c3),
unloaded(r1),
adjacent(loc2,loc1),
at(r1,loc2),
¬ occupied(loc1)}



Počáteční stav

Vlastnosti

- Procedura zpětného prohledávání je **korektní a úplná**.
- Můžeme ji implementovat **deterministicky**.
  - Pro úplnost potřebujeme detekci cyklů.
    - Je-li $(g_1,\ldots,g_k)$ posloupnost cílů, potom pokud $\exists i<k\ g_i \subseteq g_k$ pak můžeme prohledávání této cesty ukončit.
- **Větvení**
  - může být menší než u prohledávání dopředu (zacílení)
  - pořád ale zbytečně velké
    - Chceme-li, aby byl robot v pozici loc51, do které existují cesty z loc1,…,loc50, dostaneme 50 možných podcílů. Nám je ale pro splnění cíle jedno, odkud robot přijel!
    - Částečné instanciování akcí (místo úplného) dále zmenší velikost větvení, tzv. **liftování** (lifting).

Liftovaná verze

Lifted-backward-search(O, s_0, g)
  $\pi \leftarrow$ the empty plan
  loop
    if s_0 satisfies g then return π
    $A \leftarrow \{(o,\theta) | o$ is a standardization of an operator in O,
      θ is an mgu for an atom of g and an atom of effects (o),
      and $\gamma^{-1}(\theta(g),\theta(o))$ is defined$\}$
    if $A = \emptyset$ then return failure
    nondeterministically choose a pair $(o,\theta) \in A$
    $\pi \leftarrow$ the concatenation of $\theta(o)$ and $\theta(\pi)$
    $g \leftarrow \gamma^{-1}(\theta(g),\theta(o))$

- standardizace = kopie s novými proměnnými
- mgu = most general unifier (nejobecnější unifikace)
- použití volných proměnných zmenšuje větvení, ale komplikuje detekci cyklu

STRIPS

- Dosud probrané plánovací algoritmy sdílejí stejný problém - jak zlepšit efektivitu redukcí prohledávaného prostoru.

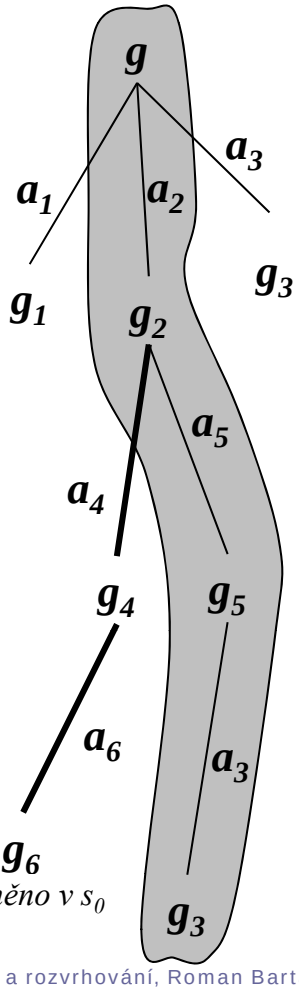
- **STRIPS algoritmus** redukuje prohledávaný prostor zpětného plánování a to takto:
  - **z podcíle řeší vždy jen část odpovídající předpokladům poslední přidané akce**
    - tj. místo $\gamma^{-1}(s,a)$ se jako nový cíl použije precond(a)
    - vede k neúplnosti algoritmu
  - **pokud aktuální stav splňuje všechny předpoklady operátoru, daný operátor se použije a tento závazek nebude rušen při backtrackingu.**

Algoritmus

- Originální algoritmus STRIPS je liftovanou verzí níže popsaného algoritmu.

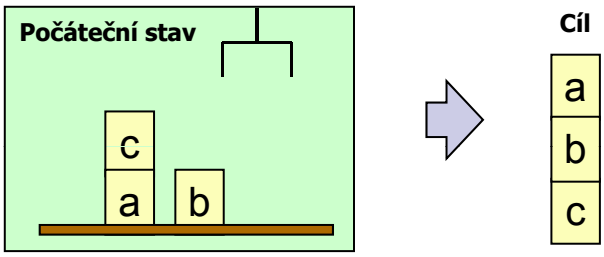
```
Ground-STRIPS(O, s, g)
    π ← the empty plan
    loop
        if s satisfies g then return π
        A ← {a | a is a ground instance of an operator in O,
                 and a is relevant for g}
        if A = ∅ then return failure
        nondeterministically choose any action a ∈ A
        π' ← Ground-STRIPS(O, s, precond(a))
        if π' = failure then return failure
        ;; if we get here, then π' achieves precond(a) from s
        s ← γ(s, π')
        ;; s now satisfies precond(a)
        s ← γ(s, a)
        π ← π.π'.a
```

$g_2 = (g$ - effects$(a_2)) \cup$ precond(a_2)
$\pi' = \langle a_6, a_4 \rangle$ je plán pro cíl precond(a_2)
$s = \gamma(\gamma(s_0,a_6),a_4)$ je stav splňující precond(a_2)



Sussmanova anomálie

- Pravděpodobně nejznámější problém, který STRIPS neumí efektivně řešit (najde pouze redundantní plány).
- **Svět kostek**



Plán vytvořený STRIPSem může vypadat takto:

- unstack(c,a),putdown(c),pickup(a),stack(a,b)
  nyní jsme splnili cíl on(a,b)
- unstack(a,b),putdown(a),pickup(b),stack(b,c)
  nyní jsme splnili cíl on(b,c), ale musíme se vrátit k on(a,b)
- pickup(a),stack(a,b)

červené akce může vyřadit

Jak na svět kostek?

- **Řešení Sussmanovy anomálie**
  - **Prolínání plánů**
    - plánování v prostoru plánů
  - **Použití doménově závislé informace**
    - Kdy má problém ve světě kostek řešení?
      - v cíli nesmí být kostky, které nejsou v počátečním stavu
      - kostka nesmí najednou stát na dvou jiných kostkách
      - ...
    - Jak to řešení najdeme?

      Celkem snadno a rychle!
      - dáme všechny kostky (samostatně) na stůl
      - postavíme požadované věže

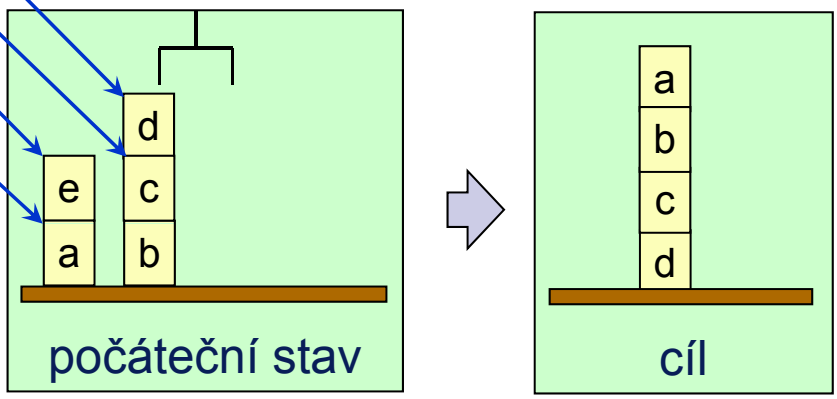
      S dalšími znalostmi to můžeme udělat ještě lépe!

Doménové znalosti
pro svět kostek

Kdy musíme pohnout kostkou *x*?

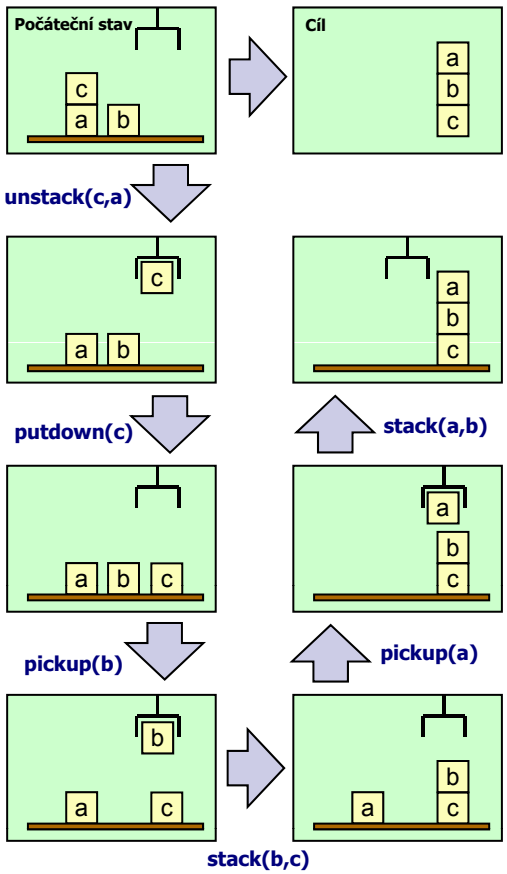
Pokud platí některé z následujícího:

- *s* obsahuje ontable(*x*) a *g* obsahuje on(*x*,*y*)
- *s* obsahuje on(*x*,*y*) a *g* obsahuje ontable(*x*)
- *s* obsahuje on(*x*,*y*) a *g* obsahuje on(*x*,*z*) pro nějaké *y*≠*z*
- *s* obsahuje on(*x*,*y*) a *y* se musí pohnout



Plánování ve světě kostek

```
Stack-containers(O, s0, g):
    if g does not satisfy the consistency conditions then
        return failure        ;; the planning problem is unsolvable
    π ← the empty plan
    s ← s0
    loop
        if s satisfies g then return π
        if there are containers b and c at the tops of their piles such that
                position(c, s) is consistent with g and on(b, c) ∈ g
        then
            append actions to π that move b to c
            s ← the result of applying these actions to s
            ;; we will never need to move b again
        else if there is a container b at the top of its pile
                such that position(b, s) is inconsistent with g
                and there is no c such that on(b, c) ∈ g
        then
            append actions to π that move b to an empty auxiliary pile
            s ← the result of applying these actions to s
            ;; we will never need to move b again
        else
            nondeterministically choose any container c such that c is
                at the top of a pile and position(c, s) is inconsistent with g
            append actions to π that move c to an empty auxiliary pallet
            s ← the result of applying these actions to s
```



Konzistence pozice s kostkou *c* znamená, že není důvod s *c* pohnout.