PLÁNOVÁNÍ A HRY - CV 3

kopriva@agents.felk.cvut.cz

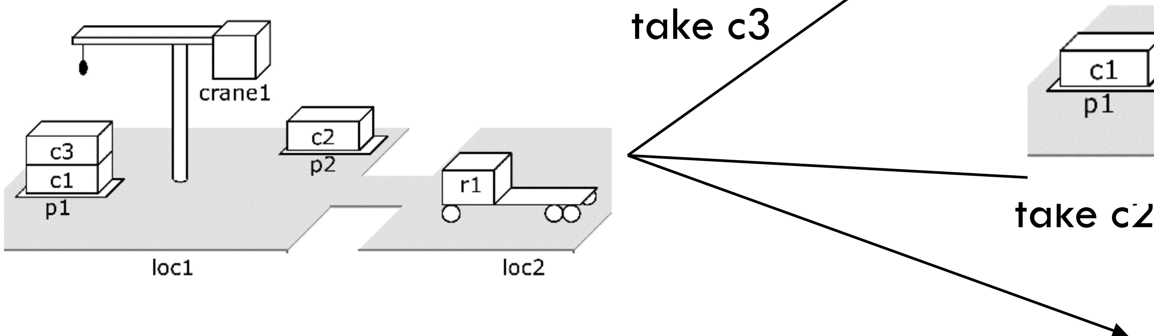
State – space Planning

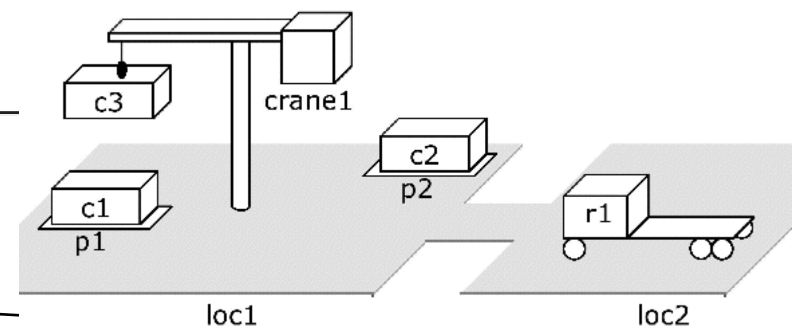
- **Forward Search**
- **Backward Search**
- **Lifting**
- **STRIPS**

Forward Search

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 an operator in O, and $\mathrm{precond}(a)$ is true in $s\}$
  - if $E = \emptyset$ then return failure
  - nondeterministically choose an action $a \in E$
  - $s \leftarrow \gamma(s, a)$
  - $\pi \leftarrow \pi.a$



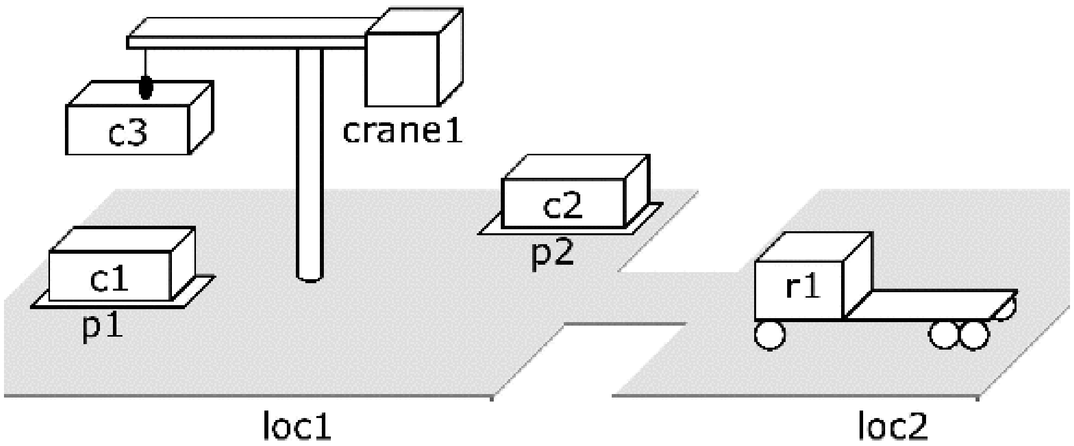


Forward Search Properties

- Forward-search is *sound*
  - for any plan returned by any of its nondeterministic traces, this plan is guaranteed to be a solution
- Forward-search also is *complete*
  - if a solution exists then at least one of Forward-search's nondeterministic traces will return a solution.

Task 1: DWR, find 1 finite and 1 infinite trace

- s_0:

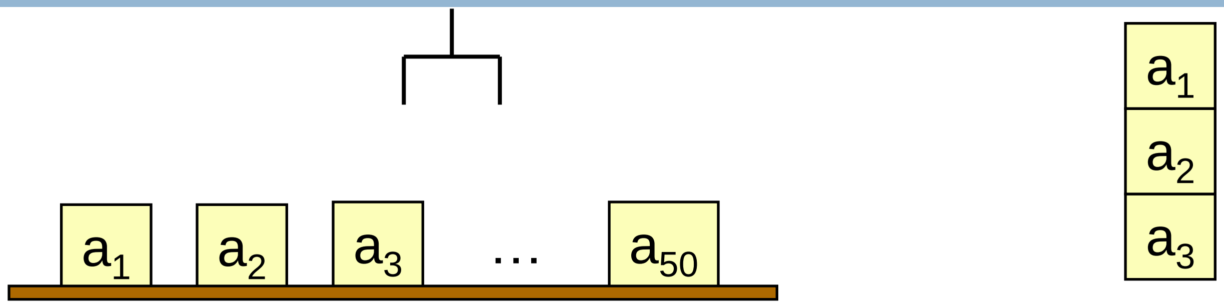


- g: {at(r1, loc1), loaded(r1, c3)}

Task 2: Interchanging variables

- Objective: Interchange the values of variables v1 and v2.
- s_0= {value(v1,3), value(v2,5), value(v3,0)}
- g = {value(v1,5), value(v2,3)}
- assign(v, w, x, y)
  - precond: value(v,x), value(w,y)
  - effects: ¬value(v,x), value(v,y)

Branching Factor of Forward Search



- Forward search can have a very large branching factor
  - E.g., many applicable actions that don't progress toward goal
- Why this is bad:
  - Deterministic implementations can waste time trying lots of irrelevant actions
- Need a good heuristic function and/or pruning procedure
- How to do pruning?

Backward Search

- For forward search, we started at the initial state and computed state transitions
  - new state = $\gamma(s,a)$
- For backward search, we start at the goal and compute inverse state transitions
  - new set of subgoals = $\gamma^{-1}(g,a)$
- To define $\gamma^{-1}(g,a)$, must first define *relevance*:
  - An action *a* is relevant for a goal *g* if
    - *a* makes at least one of *g*'s literals true
      - $g \cap \text{effects}(a) \neq \emptyset$
    - *a* does not make any of *g*'s literals false
      - $g^+ \cap \text{effects}^-(a) = \emptyset$ and $g^- \cap \text{effects}^+(a) = \emptyset$

Inverse State Transitions

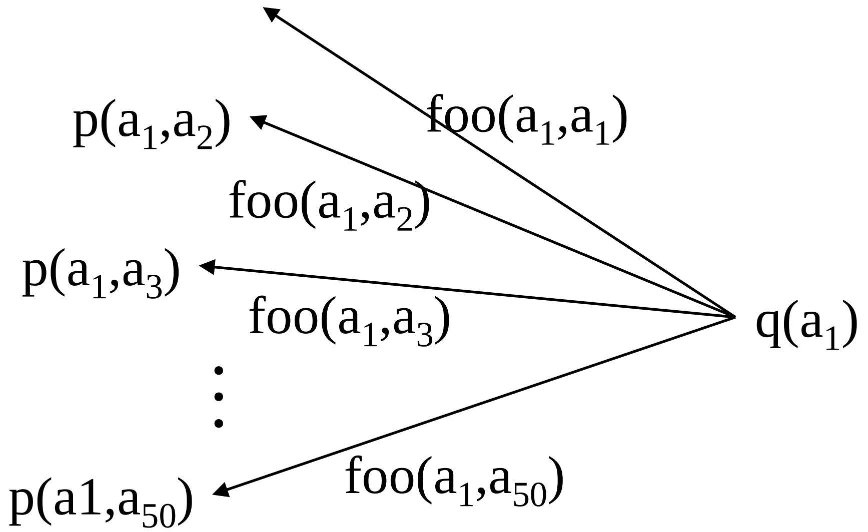
- If a is relevant for g, then
  - $\gamma^{-1}(g,a) = (g - \text{effects}(a)) \cup \text{precond}(a)$
- Otherwise $\gamma^{-1}(g,a)$ is undefined
- Example: suppose that
  - g = {on(b1,b2), on(b2,b3)}
  - a = stack(b1,b2)
- What is $\gamma^{-1}(g,a)$?

Backward Search

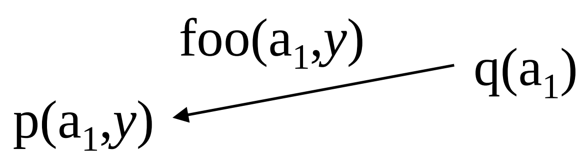
```
Backward-search(O, s_0, g)
    π ← the empty plan
    loop
        if s_0 satisfies g then return π
        A ← {a|a is a ground instance of an operator in O
                  and γ^{-1}(g, a) is defined}
        if A = ∅ then return failure
        nondeterministically choose an action a ∈ A
        π ← a.π
        g ← γ^{-1}(g, a)
```

Lifting

foo(*x*,*y*)
  precond: p(*x*,*y*)
  effects: q(*x*)



- Can reduce the branching factor of backward search if we *partially* instantiate the operators
  - this is called *lifting*



Lifted Backward Search

- More complicated than Backward-search
  - Have to keep track of what substitutions were performed
- But it has a much smaller branching factor

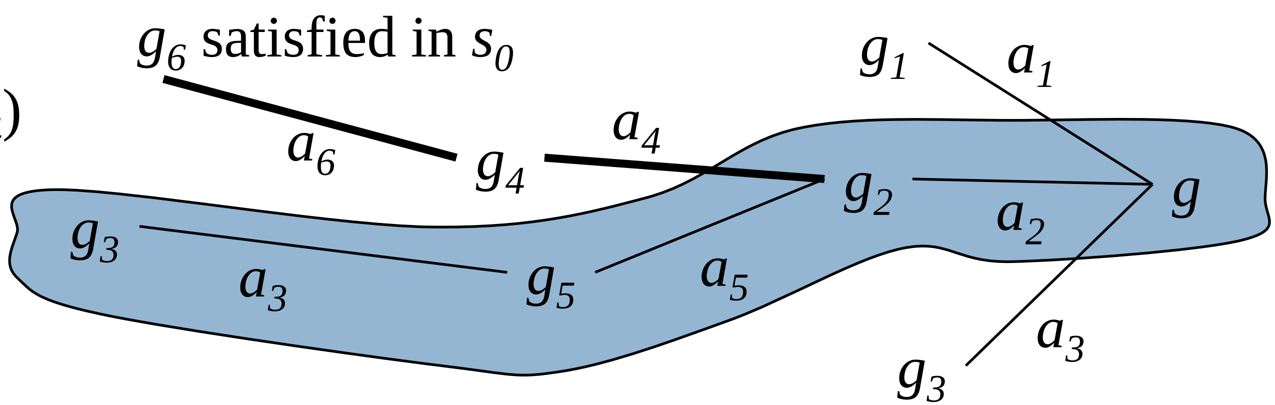
```
Lifted-backward-search(O, s_0, g)
    π ← the empty plan
    loop
        if s_0 satisfies g then return π
        A ← {(o, θ)|o is a standardization of an operator in O,
                    θ is an mgu for an atom of g and an atom of effects⁺(o),
                    and γ⁻¹(θ(g), θ(o)) is defined}
        if A = ∅ then return failure
        nondeterministically choose a pair (o, θ) ∈ A
        π ← the concatenation of θ(o) and θ(π)
        g ← γ⁻¹(θ(g), θ(o))
```

STRIPS

- π ← the empty plan
- do a modified backward search from *g*
  - instead of $\gamma^{-1}(s,a)$, each new set of subgoals is just precond(*a*)
  - whenever you find an action that's executable in the current state, then go forward on the current search path as far as possible, executing actions and appending them to π
  - repeat until all goals are satisfied

$\pi = \langle a_6, a_4 \rangle$

$s = \gamma(\gamma(s_0,a_6),a_4)$



STRIPS

```
function groundStrips(O,s,g)
    plan ← ⟨⟩
    loop
        if s.satisfies(g) then return plan
        applicables ←
            {ground instances from O relevant for g-s}
        if applicables.isEmpty() then return failure
        action ← applicables.chooseOne()
        subplan ← groundStrips(O,s,action.preconditions())
        if subplan = failure then return failure
        s ← γ(s, subplan • ⟨action⟩)
        plan ← plan • subplan • ⟨action⟩
```

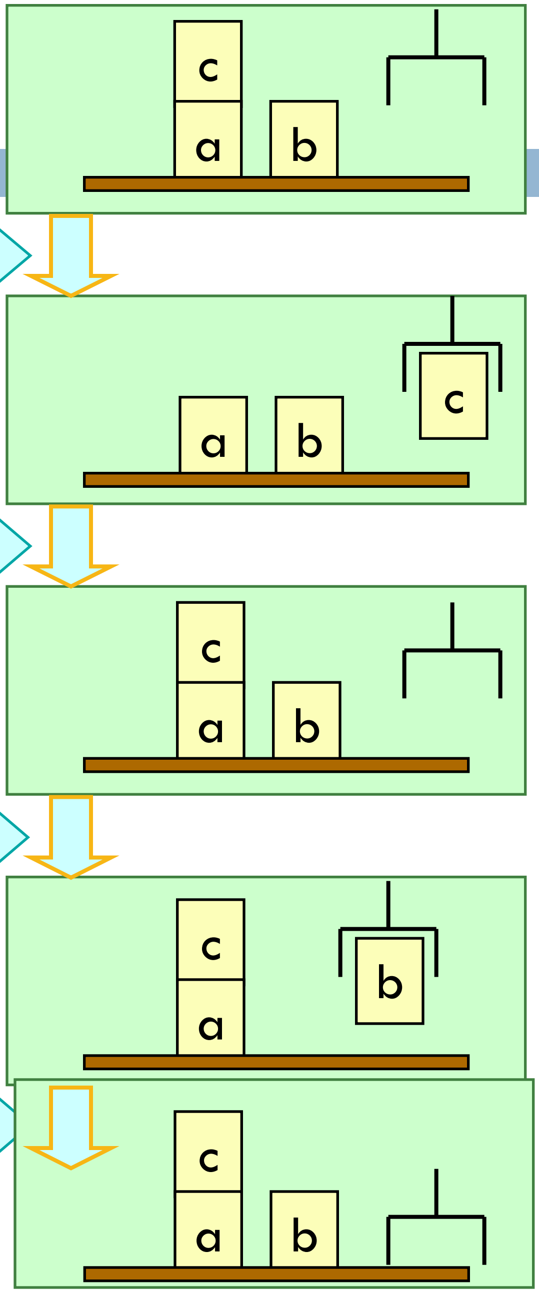
Blocks World ?

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

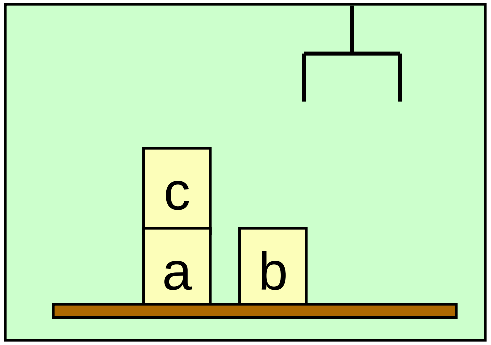
stack(x,y)
Precond: holding(x), clear(y)
Effects: ¬holding(x), ¬clear(y), on(x,y), clear(x), handempty

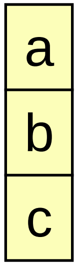
pickup(x)
Precond: ontable(x), clear(x), handempty
Effects: ¬ontable(x), ¬clear(x), ¬handempty, holding(x)

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



Sussman Anomaly





- Initial State

Goal

- Sub goals:
- 1) Put A on B
- 2) Put B on C

Interchanging Variables Repeated

- Objective: Interchange the values of variables v1 and v2.
- s_0= {value(v1,3), value(v2,5), value(v3,0)}
- g = {value(v1,5), value(v2,3)}
- assign(v, w, x, y)
  - precond: value(v,x), value(w,y)
  - effects: ¬value(v,x), value(v,y)

How to Handle Problems like These?

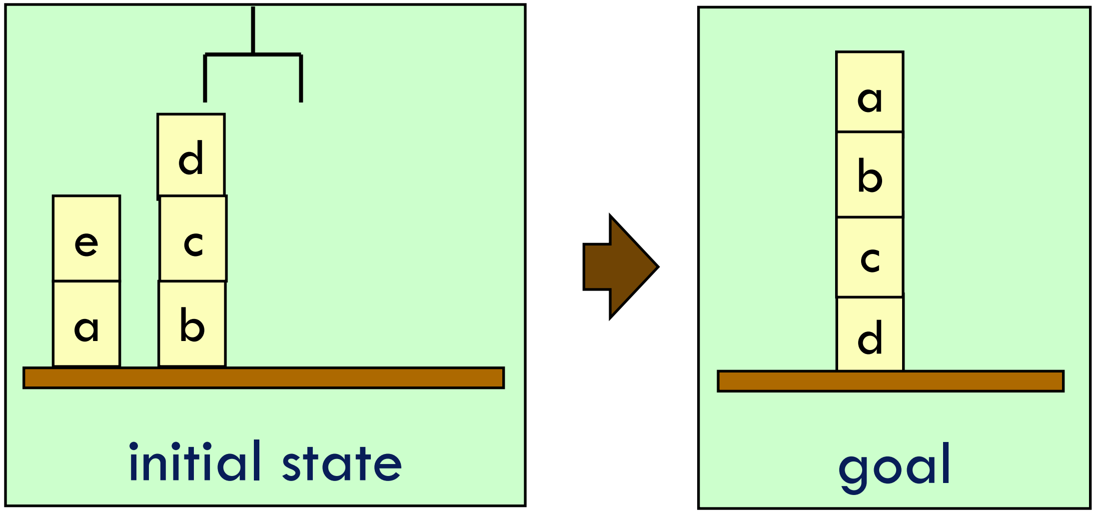
- Several ways:
  - Do something other than state-space search
  - Use forward or backward state-space search, with *domain-specific* knowledge to prune the search space
    - Can solve both problems quite easily this way
    - Example: block stacking using forward search

Domain-specific knowledge

- A blocks-world planning problem $P = (O, s_0, g)$ is solvable
  if s_0 and g satisfy some simple consistency conditions
  - g should not mention any blocks not mentioned in s_0
  - a block cannot be on two other blocks at once
- If P is solvable, can easily construct a solution of length $O(2m)$, where m is the number of blocks
  - Move all blocks to the table, then build up stacks from the bottom
    - Can do this in time $O(n)$
- With additional domain-specific knowledge can do even better …

Additional Domain-Specific Knowledge

- A block *x* needs to be moved if any of the following is true:
  - *s* contains ontable(*x*) and *g* contains on(*x*,*y*) - see a below
  - *s* contains on(*x*,*y*) and *g* contains ontable(*x*) - see d below
  - *s* contains on(*x*,*y*) and *g* contains on(*x*,*z*) for some $y \neq z$, see c below
  - *s* contains on(*x*,*y*) and *y* needs to be moved - see e below



Domain – specific Algorithm

loop
if there is a clear block *x* such that
x needs to be moved **and**
x can be moved to a place where it won't need to be moved
then move *x* to that place
else if there is a clear block *x* such that
x needs to be moved
then move *x* to the table
else if the goal is satisfied
then return the plan
else return failure
repeat

STRIPS Planning Task

- Monkey and Banana