PLÁNOVÁNÍ A HRY – CV 4

STATE-SPACE SEARCH

Jan Hrnčíř

Based on slides by Štěpán Kopřiva

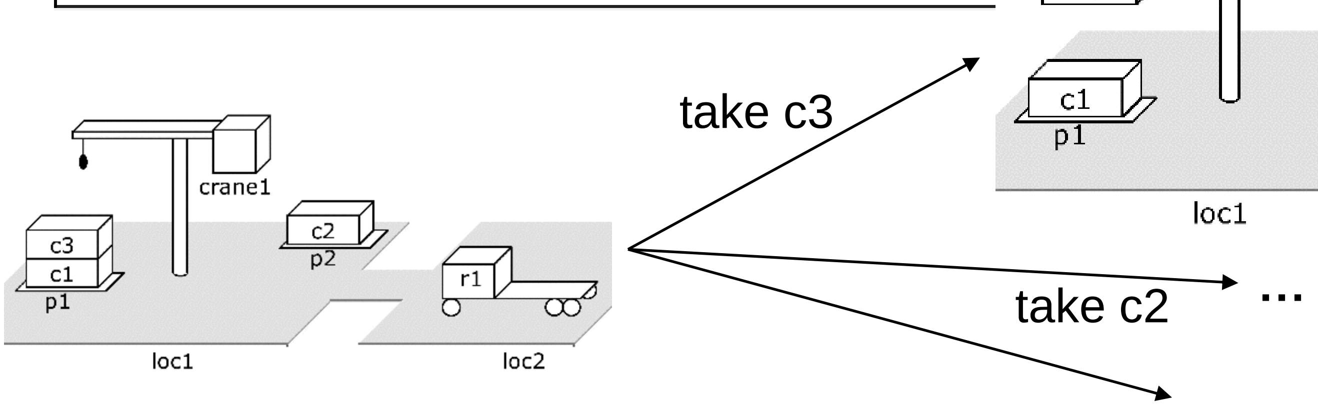
State-space Search

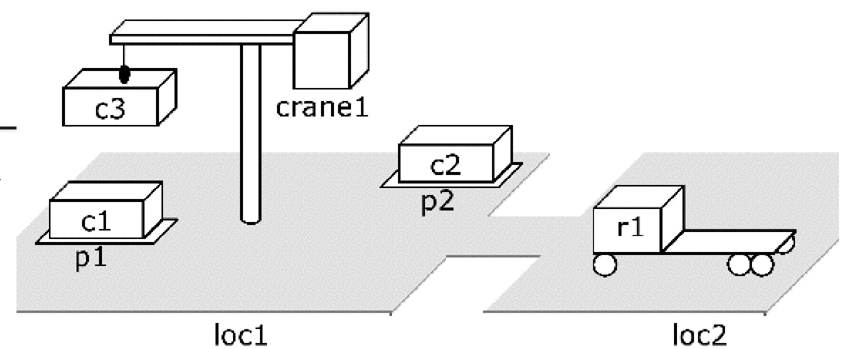
- **Forward Search**
- **Backward Search**
- **Heuristic Search**

Forward search

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 $a \in E$
    $s \leftarrow \gamma(s, a)$
    $\pi \leftarrow \pi.a$



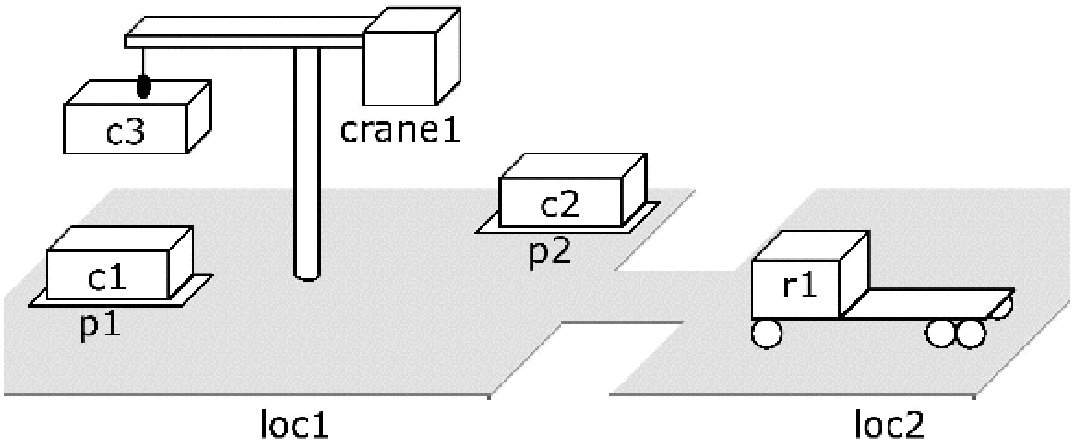


Forward Search Properties

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

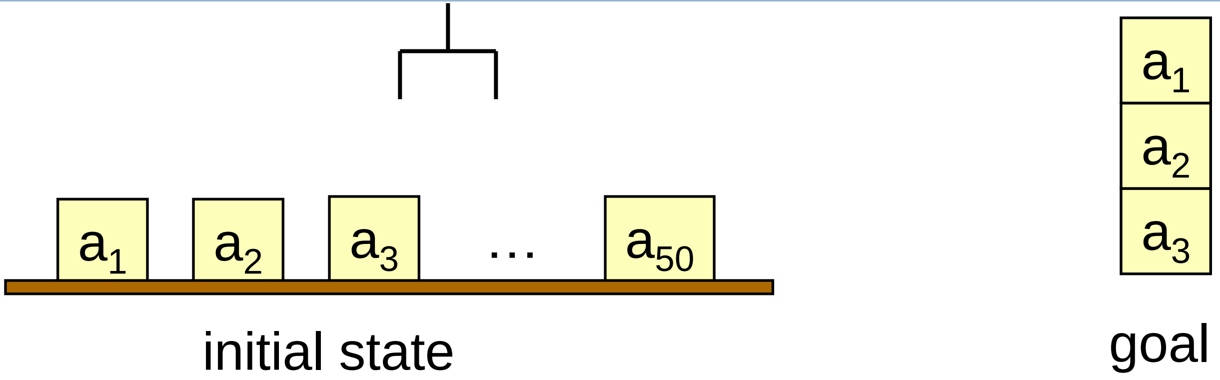
Task 1: DWR, find one finite and one infinite trace

- s_0:



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

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

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
  - $g = \{\text{on}(b1,b2), \text{on}(b2,b3)\}$
  - $a = \text{stack}(b1,b2)$
- What is $\gamma^{-1}(g,a)$?

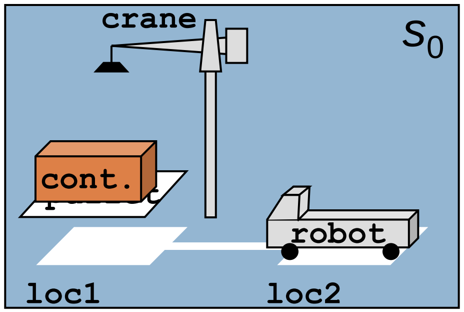
Backward Search

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 $a \in A$
        $\pi \leftarrow a.\pi$
        $g \leftarrow \gamma^{-1}(g, a)$

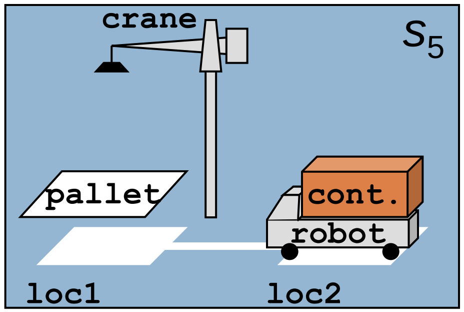
Task 2: DWR, backward search

- Solve the problem by the backward-search, trace the algorithm.
- **Actions:** load(crane, loc, cont, r), take(crane, loc, cont, pallet, pile), move(r, from, to)

initial state:



goal state:



Task 2: DWR, backward search

- Solve the problem by the backward-search, trace the algorithm.
- **Actions:** load(crane, loc, cont, r), take(crane, loc, cont, pallet, pile), move(r, from, to)

initial state:

plan =

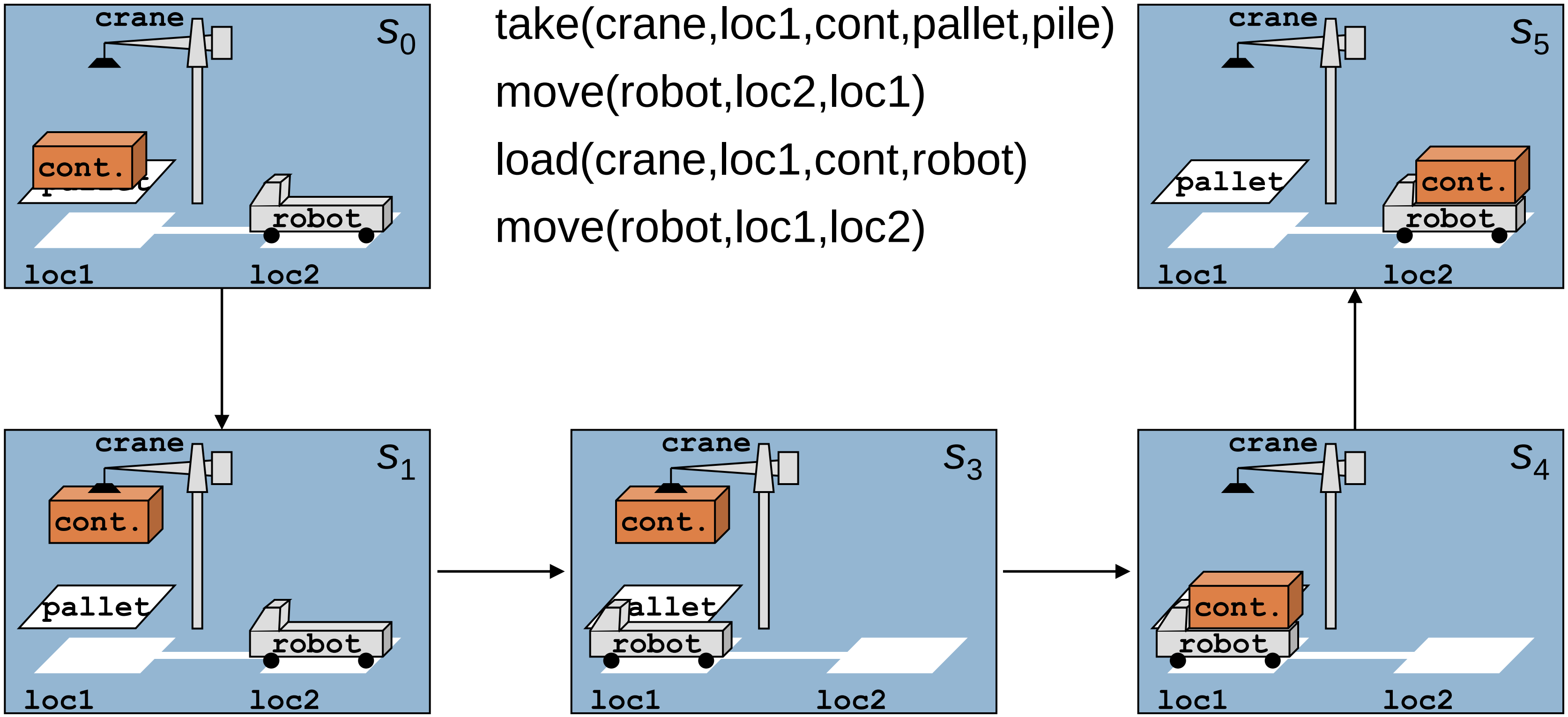
take(crane,loc1,cont,pallet,pile)

move(robot,loc2,loc1)

load(crane,loc1,cont,robot)

move(robot,loc1,loc2)

goal state:

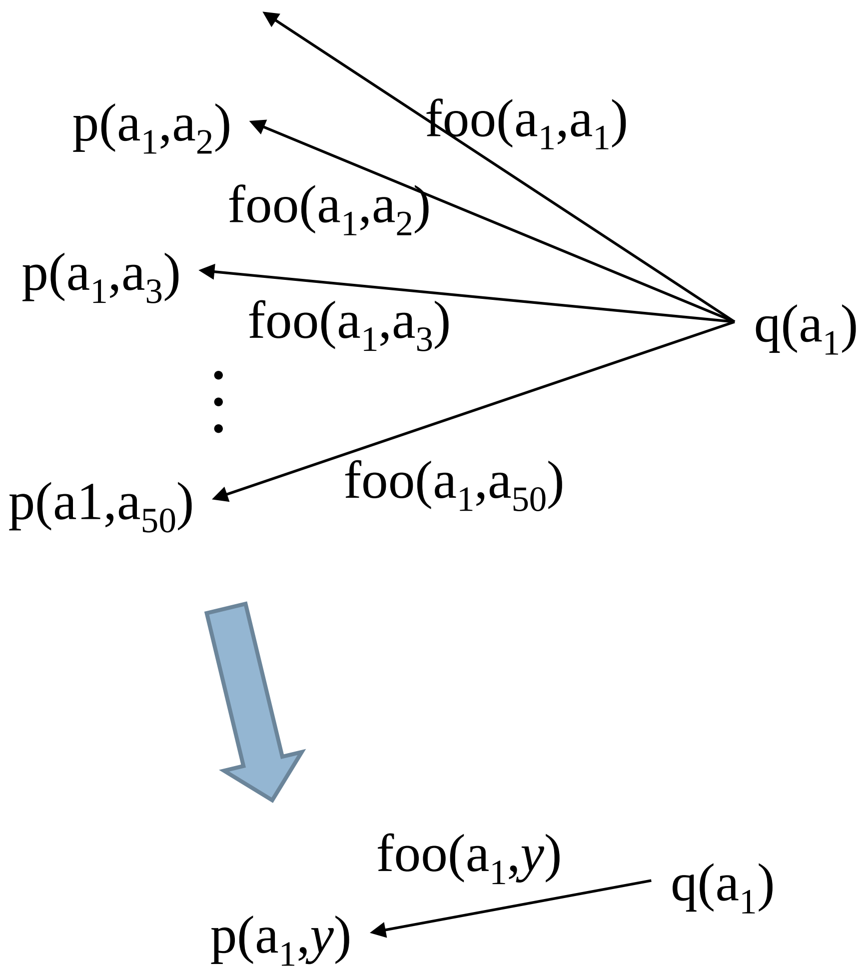


Lifting I.

- Backward search can *also* have a very large branching factor
  - E.g., an operator *o* that is relevant for *g* may have many ground instances a_1, a_2, …, a_n such that each a_i's input state might be unreachable from the initial state
- Can reduce the branching factor of backward search if we *partially* instantiate the operators
  - this is called *lifting*
- Basic Idea: Delay grounding of operators until necessary in order to bind variables with those required to realize goal or subgoal

Lifting II.

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



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))
```

Heuristic search

Local heuristic search: Hill climbing

Hill-climbing

$\sigma :=$ make-root-node(init())
forever:
    if is-goal(state(σ)):
        return extract-solution(σ)
    $\Sigma' := \{$ make-node$(\sigma, o, s) \mid \langle o, s\rangle \in$ succ(state(σ)) $\}$
    $\sigma :=$ an element of Σ' minimizing h (random tie breaking)

Enforced hill-climbing: procedure improve

```
def improve(σ0):
    queue := new fifo-queue
    queue.push-back(σ0)
    closed := ∅
    while not queue.empty():
        σ = queue.pop-front()
        if state(σ) ∉ closed:
            closed := closed ∪ {state(σ)}
            if h(σ) < h(σ0):
                return σ
            for each ⟨o, s⟩ ∈ succ(state(σ)):
                σ' := make-node(σ, o, s)
                queue.push-back(σ')
    fail
```

Enforced hill-climbing

```
σ := make-root-node(init())
while not is-goal(state(σ)):
    σ := improve(σ)
return extract-solution(σ)
```

Systematic heuristic search: Greedy best-first search

Greedy best-first search (with duplicate detection)

open := **new** min-heap ordered by $(\sigma \mapsto h(\sigma))$
open.insert(make-root-node(init()))
closed := $\emptyset$
while not *open*.empty():
    σ = *open*.pop-min()
    if *state*$(\sigma) \notin$ *closed*:
        closed := *closed* $\cup$ {*state*(σ)}
        if is-goal(state(σ)):
            return extract-solution(σ)
        for each $\langle o, s\rangle \in$ succ(*state*(σ)):
            σ' := make-node(σ, o, s)
            if $h(\sigma') < \infty$:
                open.insert(σ')
return unsolvable