Lecture slides for

Automated Planning: Theory and Practice

Chapter 4
State-Space Planning

Dana S. Nau
University of Maryland

5:05 PM September 16, 2013

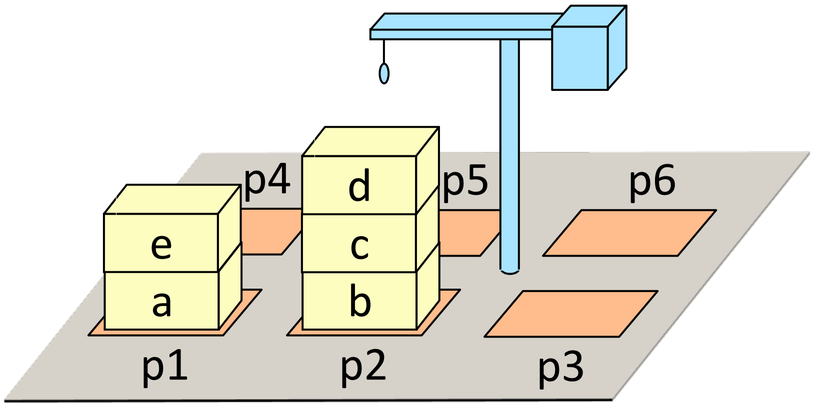
Motivation and Outline

- Nearly all planning procedures are search procedures
- Different planning procedures have different search spaces
  - This chapter: **state-space planning**
    - Each node represents a state of the world
      - A plan is a path through the space

- **Outline**
  - Example: container-stacking problems
  - Forward search
  - Backward search
  - Lifting

Container-Stacking Problems

- Another simplified version of DWR
  - One location, one crane
  - *k* stackable containers
  - At least *k* pallets
    - locations to stack containers
- Objects:
  - *Containers* = {a,b,c,…} or {c1,c2,…}
  - *Pallets* = {p1,p2,p3, …}
  - *Positions* = *Containers* ∪ *Pallets*
  - *Booleans* = {T, F}
- State variables:
  - pos(*c*) for each *c* ∈ *Containers*
    - Dom(pos(*c*)) = *Positions*
  - clear(*z*) for each *z* ∈ *Positions*
    - Dom(clear(*c*)) = *Booleans*



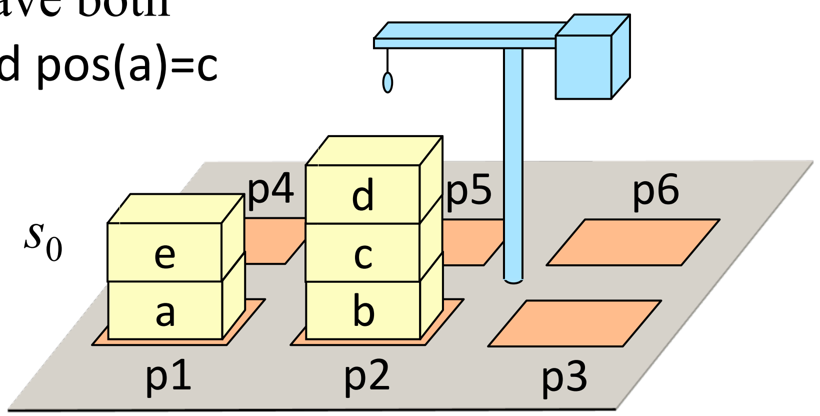
- Example state:
  - clear(a) = clear(b) = clear(c) = F
  - clear(d) = clear(e) = T
  - clear(p1) = clear(p1) = F
  - clear(p2) = clear(p4) = clear(p5) = clear(p6) = T
  - pos(a) = p1, pos(b) = p2, pos(c) = b, pos(d) = c, pos(e) = a

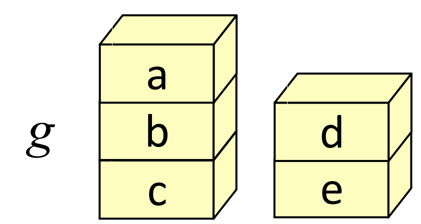
Container-Stacking Problems

- One class of action: move

  move(c: *Containers*, y: *Positions*, z: *Positions*$-\{y\}$)
  Pre: pos(c)=y, clear(c)=T, clear(z)=T
  Eff: pos(c)$\leftarrow z$, clear(y)$\leftarrow$T, clear(z)$\leftarrow$F

- Initial state s_0: arbitrary configuration of the containers
- Goal g is a set of state-variable assignments for pos variables
  - specifies stacks of containers, but not what pallets they're on
- g must represent a set of real states of Σ
  - e.g., can't have both pos(a)=b and pos(a)=c
- Example:





Forward Search

i.e., D

Forward-search(Σ, s_0, g)

1. $\pi \leftarrow \langle\rangle$; $s \leftarrow s_0$
2. loop
3. if s satisfies g then return π
4. $A' \leftarrow \{a \in A \mid s \text{ satisfies } \mathrm{Pre}(a)\}$
5. if $A' = \varnothing$ then return failure
6. nondeterministically choose $a \in A'$
7. $s \leftarrow \gamma(s, a)$
8. $\pi \leftarrow \pi \cdot a$

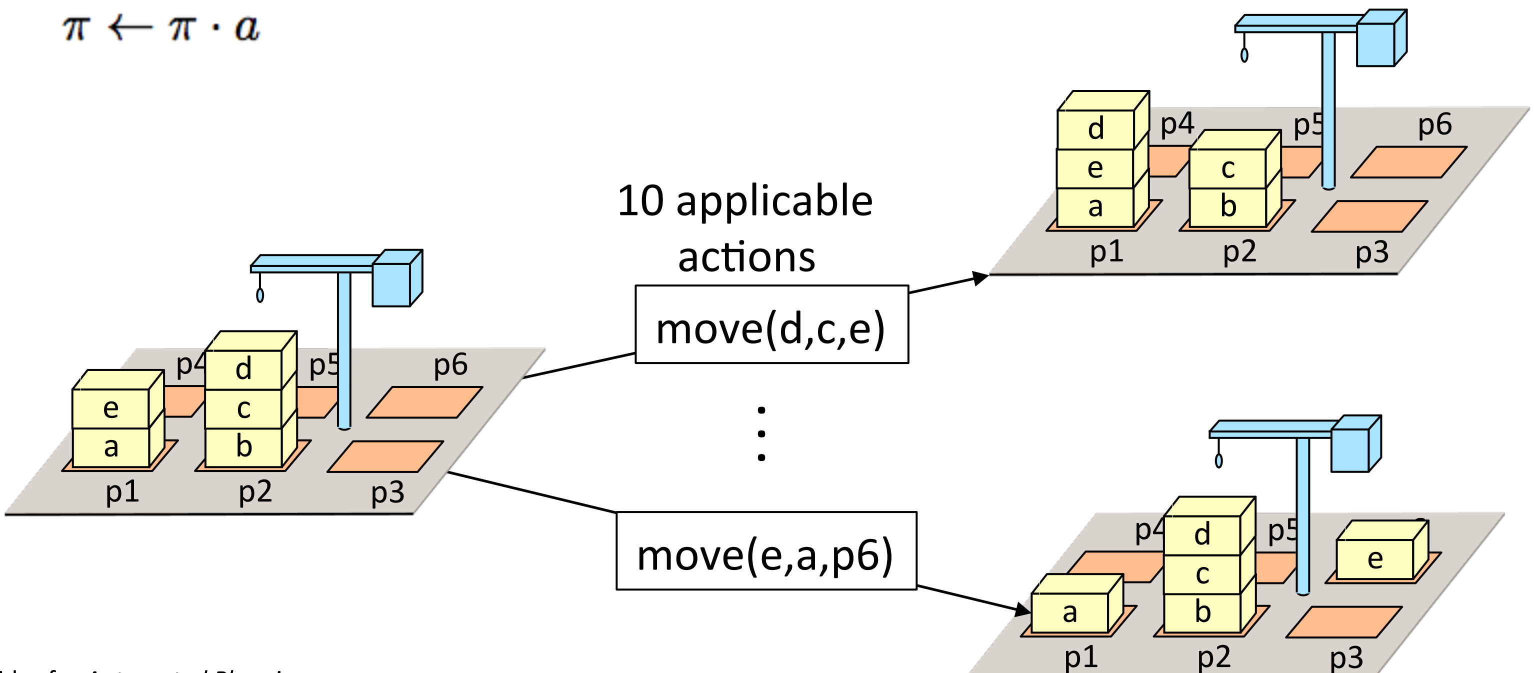
For loop checking:

- After line 1, put

  $Visited = \{s_0\}$
- After line 6, put

  if $s \in Visited$ then return failure

  $Visited \leftarrow Visited \cup \{s\}$

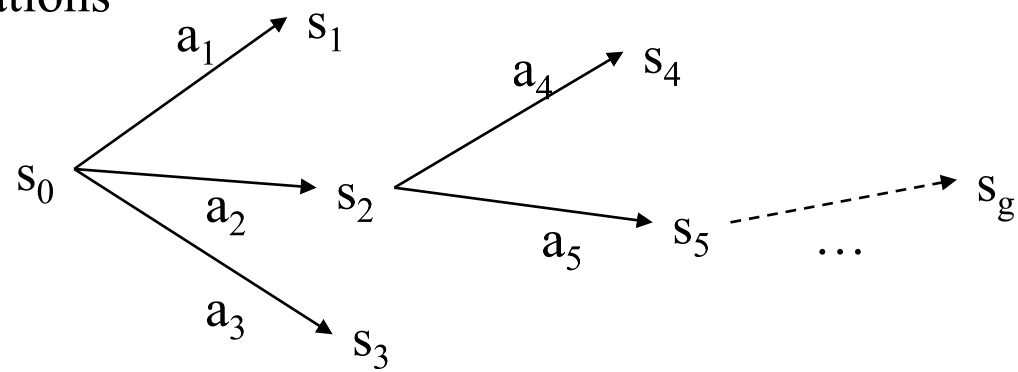


Properties

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

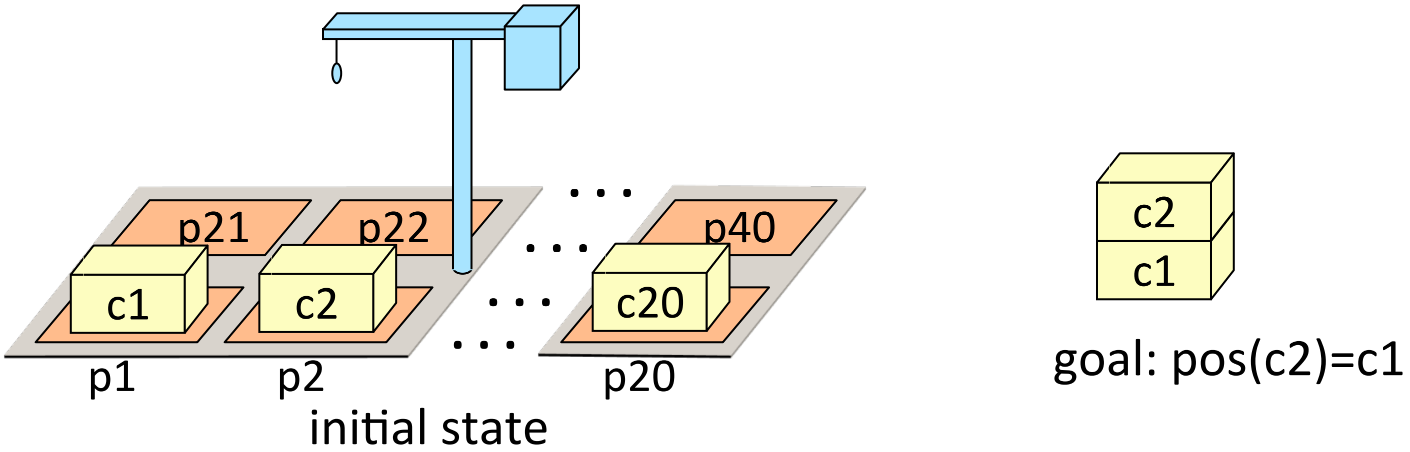
Deterministic Implementations

- Some deterministic implementations of forward search:
  - breadth-first search
  - depth-first search
  - best-first search (e.g., A*)
  - greedy search



- Breadth-first and best-first search are sound and complete
  - But often they aren't practical
  - Memory requirement is exponential in the length of the solution
- Planning algorithms often use depth-first search or greedy search
  - Worst-case memory requirement is linear in the length of the solution
  - Sound but not complete – can go down an infinite path and never return
    - But classical planning has only finitely many states
    - Thus, can make depth-first search complete by checking whether the current path contains a cycle

Branching Factor of Forward Search



- Forward search can have a very large branching factor
  - Example: 20 containers, 39 places to move each container
    - 780 applicable actions
    - all but one are useless for reaching the goal
- Need a good heuristic function and/or pruning procedure
  - Domain-specific algorithm (later in this lecture)
  - Search Heuristics (next lecture)
    - Based loosely on Chapters 9 and 6

Backward Search

- Forward search started at the initial state and computed state transitions
  - $s' = \gamma(s,a)$
- Backward search starts at the goal and computes **inverse** state transitions
  - $g' = \gamma^{-1}(g,a)$
  - g' = properties a state s' should satisfy in order for $\gamma(s',a)$ to satisfy g

- To define $\gamma^{-1}(g,a)$, we need a to be **relevant** for achieving g
  - a could be the last action of a minimal plan that achieves g
  - definition on next slide

- If a is relevant for achieving g, then
  - state-variable notation: $\gamma^{-1}(g,a) = \text{Pre}(a) \cup (g - \text{Eff}(a))$
  - classical notation: $\gamma^{-1}(g,a) = \text{precond}(a) \cup (g - \text{effects(a)})$

- If a isn't relevant for g, then $\gamma^{-1}(g,a)$ is undefined

Relevance

- Idea: a is **relevant** for g if a could potentially be the last action of a minimal plan that achieves g
- If $g = \{g_1, \ldots, g_k\}$, then this means
  1. Eff(a) makes at least one g_i true
  2. Eff(a) doesn't make any g_i false
  3. Pre(a) doesn't require any g_i to be false *unless* Eff(a) makes g_i true

State-variable representation

- g, Pre(a), and Eff(a) are sets of state-variable assignments (x,c)

1. $\mathrm{Eff}(a) \cap g \neq \emptyset$

2. $\forall x, c, c'$, if $(x,c) \in \mathrm{Eff}(a)$ and $(x,c') \in g$ then $c = c'$

3. $\forall x, c, c'$, if $(x,c) \in \mathrm{Pre}(a)$ and $(x,c') \in g - \mathrm{Eff}(a)$ then $c = c'$

Classical representation

- g, precond(a), and effects(a) are sets of ground literals

1. $\mathrm{effects}(a) \cap g \neq \emptyset$

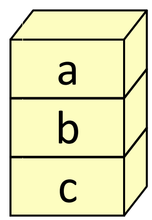
2. $\mathrm{effects}^-(a) \cap g^+ = \emptyset$; $\mathrm{effects}^+(a) \cap g^- = \emptyset$

3. $(\mathrm{precond}^-(a) - \mathrm{effects}^+(a)) \cap g^+ = \emptyset$; $(\mathrm{precond}^+(a) - \mathrm{effects}^-(a)) \cap g^- = \emptyset$

Inverse State Transitions

- If a isn't relevant for g, then $\gamma^{-1}(g,a)$ is undefined
- If a is relevant for g, then
  - state-variable notation: $\gamma^{-1}(g,a) = \text{Pre}(a) \cup (g - \text{Eff}(a))$
  - classical notation: $\gamma^{-1}(g,a) = \text{precond}(a) \cup (g - \text{effects(a)})$

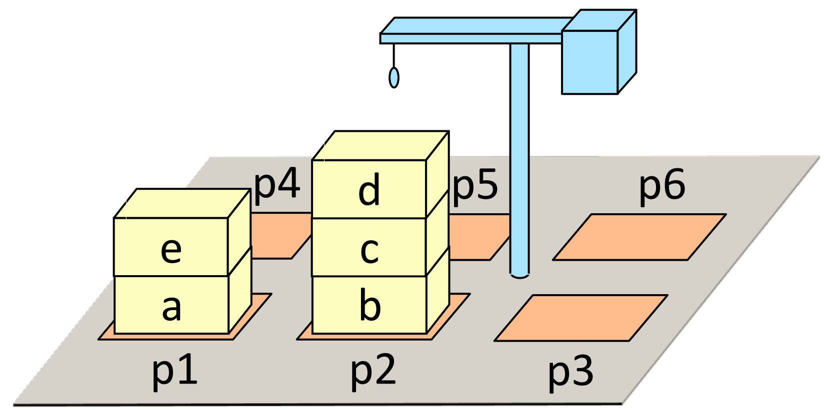
- Example:
  - $g = \{$pos(a)=b, pos(b)=c$\}$
  - a = move(a,p1,b)



g

- What is $\gamma^{-1}(g,a)$?

- What if a = move(a,p2,b) ?



move(c: *Containers*, y: *Positions*, z: *Positions*$-\{y\}$)
 Pre: pos(c)=y, clear(c)=T, clear(z)=T
 Eff: pos(c)$\leftarrow z$, clear(y)$\leftarrow$T, clear(z)$\leftarrow$F

Backward Search

Backward-search(Σ, s_0, g)

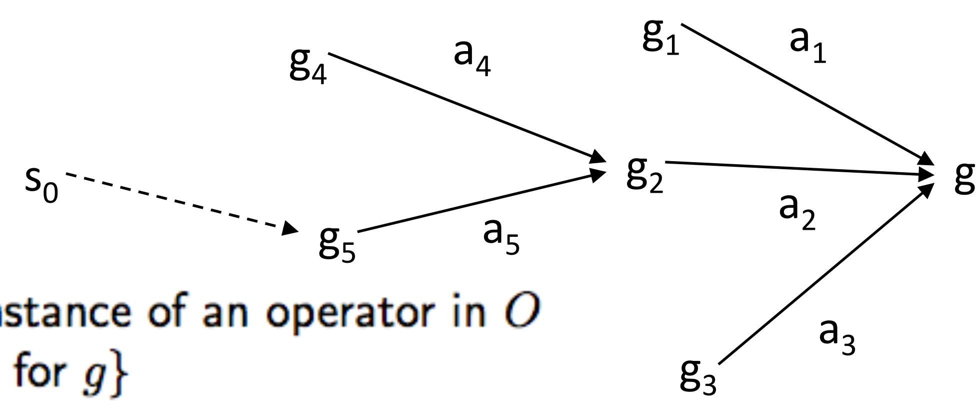
1. $\pi \leftarrow \langle\rangle;\ \ g' \leftarrow g;$
2. loop
3. if s_0 satisfies g' then return π
4. $A' \leftarrow \{a \in A \mid a \text{ is relevant for } g'\}$
5. if $A' = \varnothing$ then return failure
6. nondeterministically choose $a \in A'$
7. $g' \leftarrow \gamma^{-1}(g', a)$
8. $\pi \leftarrow a \cdot \pi$

For loop checking:

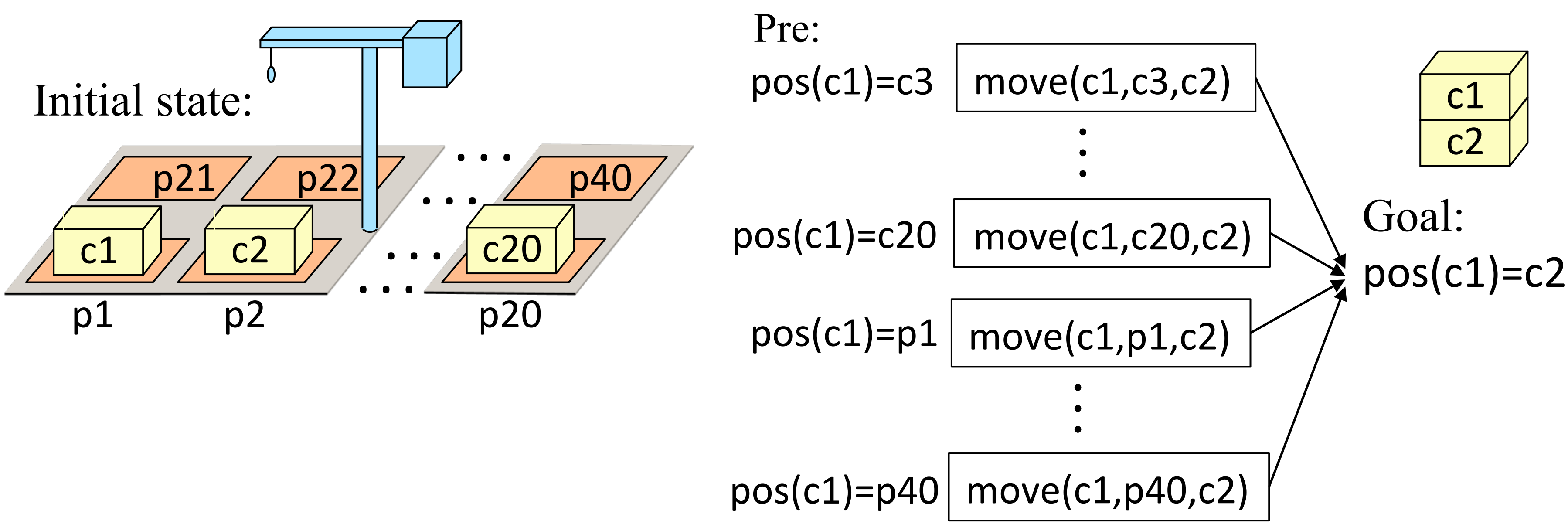
- After line 1, put
  Solved $= \{g\}$
- After line 6, put
  if $g' \in$ *Solved* then return failure
  Solved $\leftarrow$ *Solved* $\cup \{g'\}$
- More powerful:
  if $\exists g \in$ *Solved* s.t. $g \subseteq g'$ then return failure

Backward-search(O, s_0, g)

$\pi \leftarrow$ the empty plan
loop
  if s_0 satisfies g then return π
  applicable $\leftarrow \{a \mid a$ is a ground instance of an operator in O that is relevant for $g\}$
  if *applicable* $= \emptyset$ then return failure
  nondeterministically choose an action $a \in$ *applicable*
  $\pi \leftarrow a.\pi$
  $g \leftarrow \gamma^{-1}(g, a)$

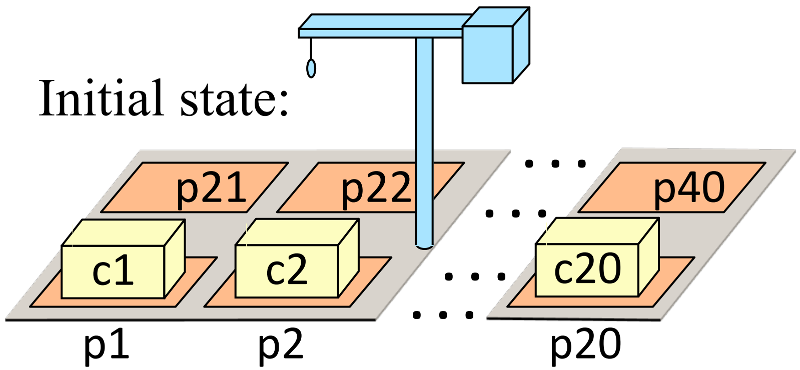


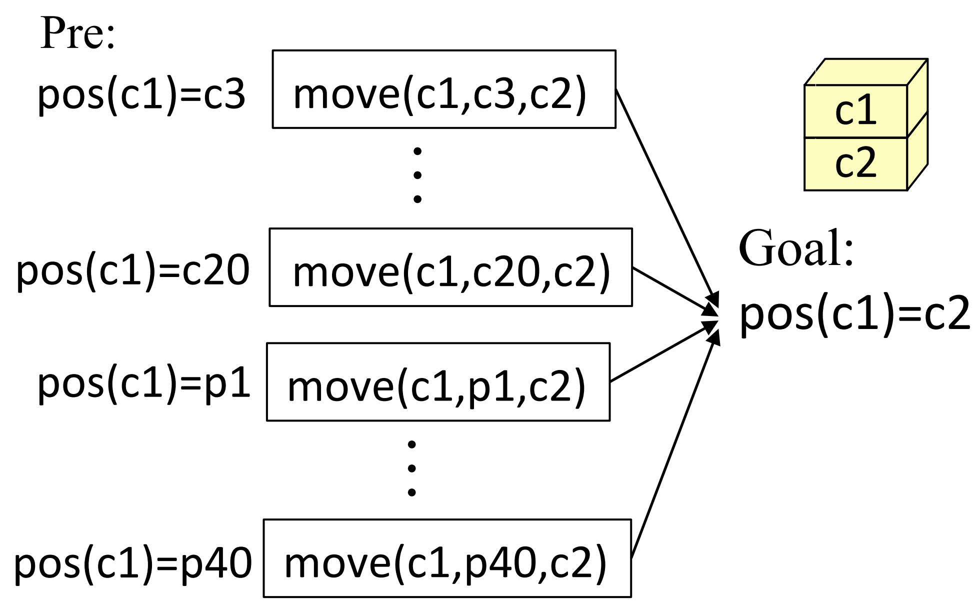
Branching Factor



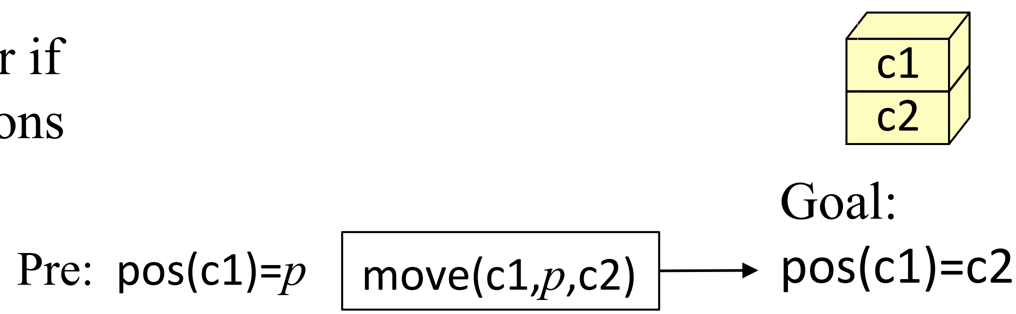
- Backward search can *also* have a very large branching factor
  - Example: g = {pos(c1)=c2}
  - 58 relevant actions
    - move c1 to c2 from 18 containers, 40 pallets
- A blind search may waste lots of time trying useless actions

Lifting





- Can reduce the branching factor if we *partially* instantiate the actions
  - this is called **lifting**



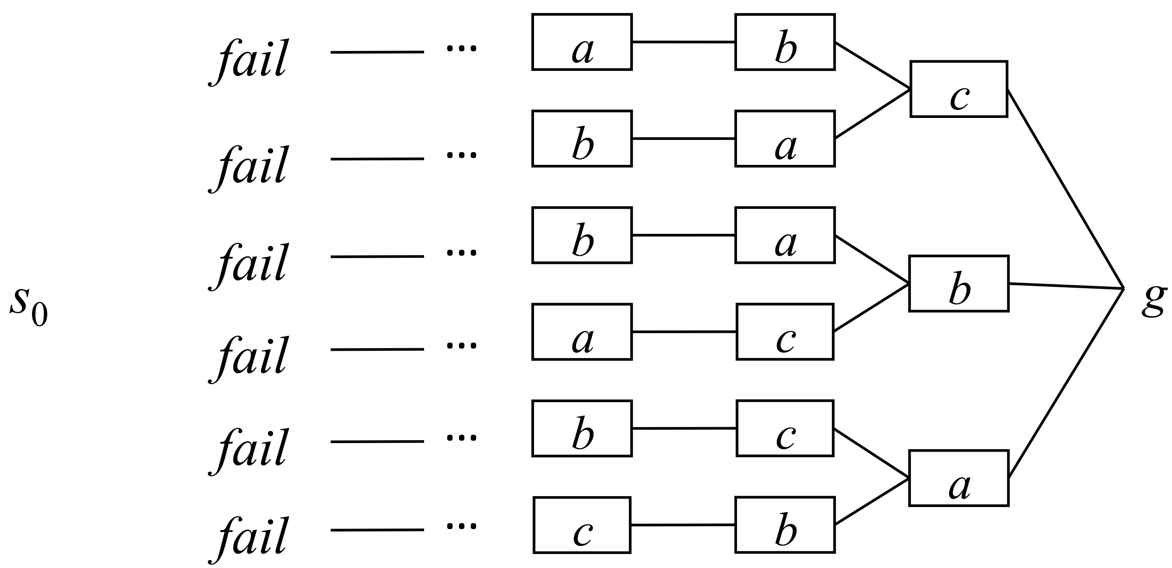
Lifted Backward Search

- Like Backward-search but more complicated
  - Have to keep track of what substitutions were performed on what parameters
  - But it has a much smaller branching factor
- Classical-planning version:

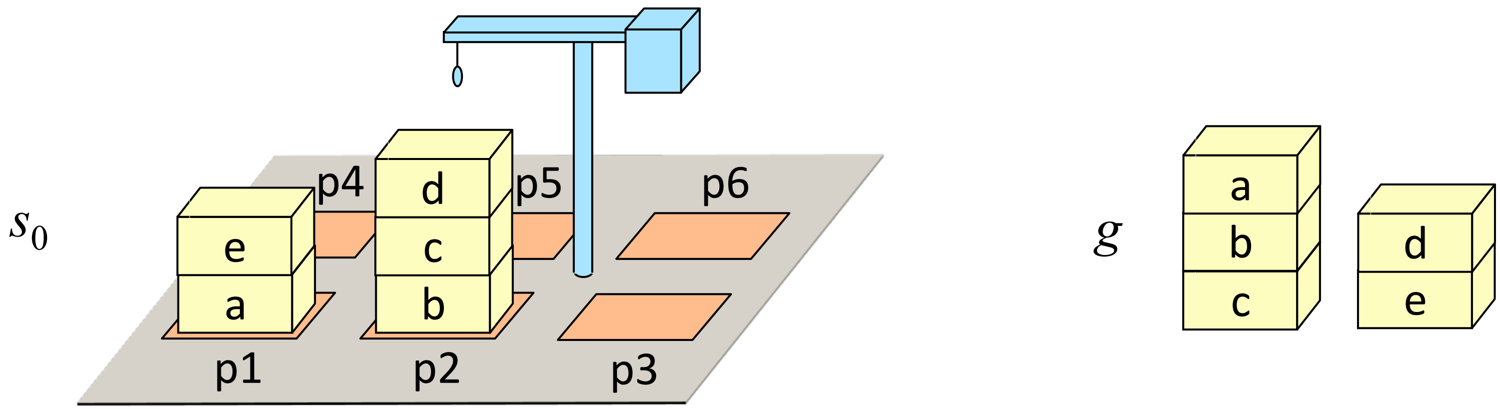
```
Lifted-backward-search(O, s_0, g)
    π ← the empty plan
    loop
        if s_0 satisfies g then return π
        A ← {(o, θ)|o is a standardization of an operator in O,
                θ is an mgu for an atom of g and an atom of effects⁺(o),
                and γ⁻¹(θ(g), θ(o)) is defined}
    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

Search Space

- Even with lifting, the search space may *still* be quite large
  - Example:
    - actions a, b, and c are independent, all are relevant for g
    - g is unreachable from s_0
    - try all possible orderings before finding there's no solution
  - This can also happen with forward search
- More about this in Chapter 5 (Plan-Space Planning)



Domain-Specific Planning Algorithms



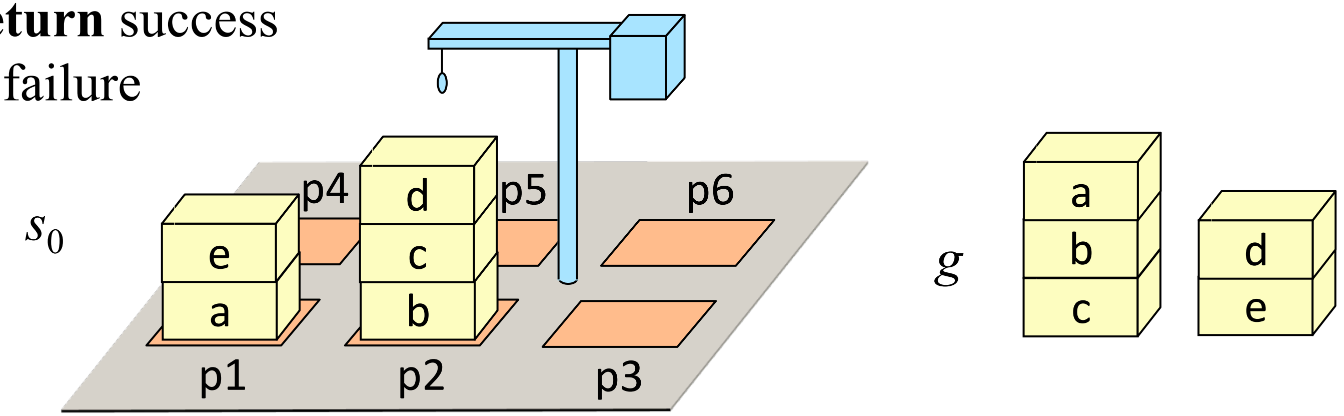
- Sometimes we can write highly efficient planning algorithms for a specific class of problems
  - Use special properties of that class

- For container-stacking problems with n containers, we can easily get a solution of length $O(n)$
  - Move all containers to pallets, then build up stacks from the bottom
- With additional domain-specific knowledge, can do even better ...

Container-Stacking Algorithm

loop
  if ∃ a clear container *c* that needs moving
    & we can move *c* to a position *d*
      where *c* won't need moving
   then move *c* to *d*
  else if ∃ a clear container *c* that needs moving
    then move *c* to any clear pallet
  else if the goal is satisfied
    then return success
  else return failure
repeat

- *c* needs moving if
  - *s* contains pos(*c*)=*d*, and *g* contains pos(*c*)=*e*, $e \neq d$
  - *s* contains pos(*c*)=*d*, and *g* contains pos(*b*)=*d*, $b \neq c$
  - *s* contains pos(*c*)=*d*, and *d* needs moving



- The algorithm generates the following sequence of actions:
  - ⟨move(e,a,p3), move(d,c,e), move(c,b,p4), move(b,p2,c), move(a,p1,b)⟩

Properties of the Algorithm

- Sound, complete, guaranteed to terminate on all container-stacking problems

- Easily solves problems like the Sussman anomaly

- Runs in time $O(n^3)$
  - Can be modified (Slaney & Thiébaux) to run in time $O(n)$

- Often finds optimal (shortest) solutions
- But sometimes only near-optimal (Exercise 4.22 in the book)
  - For container-stacking problems, PLAN-LENGTH is NP-complete