Lecture slides for

Automated Planning: Theory and Practice

Chapter 4
State-Space Planning

Dana S. Nau

University of Maryland

4:56 PM February 1, 2012

Motivation

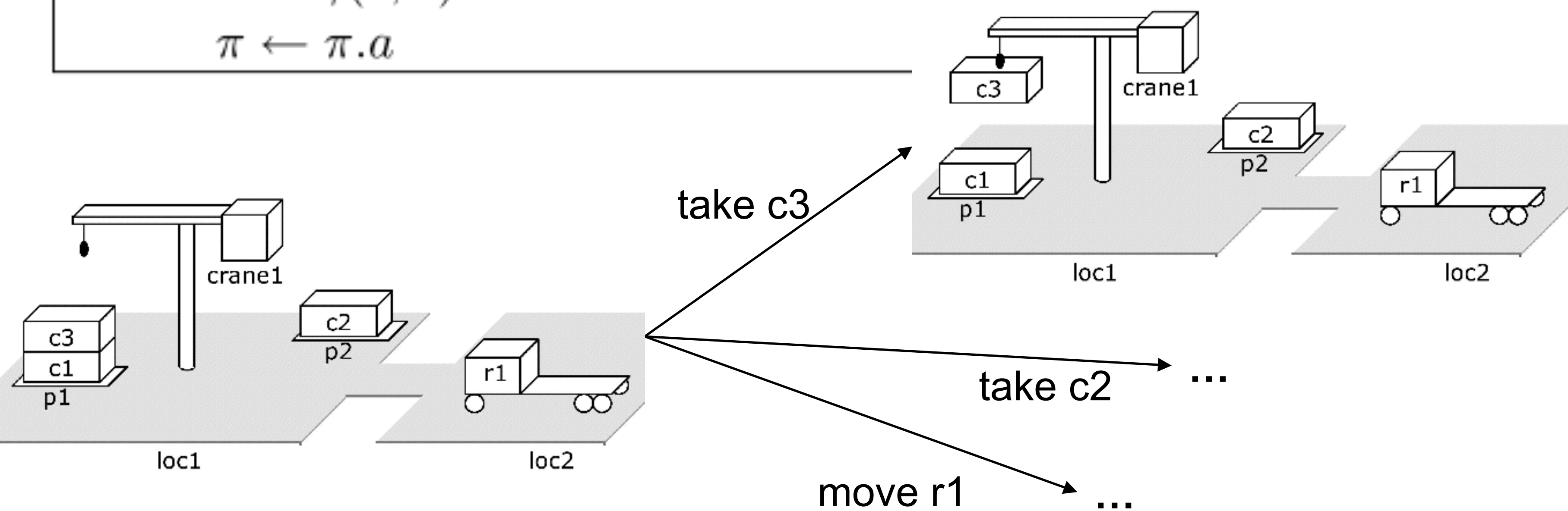
- Nearly all planning procedures are search procedures
- Different planning procedures have different search spaces
  - Two examples:
- *State-space planning*
  - Each node represents a state of the world
    - A plan is a path through the space
- *Plan-space planning*
  - Each node is a set of partially-instantiated operators, plus some constraints
    - Impose more and more constraints, until we get a plan

Outline

- State-space planning
  - Forward search
  - Backward search
  - Lifting
  - STRIPS
  - Block-stacking

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

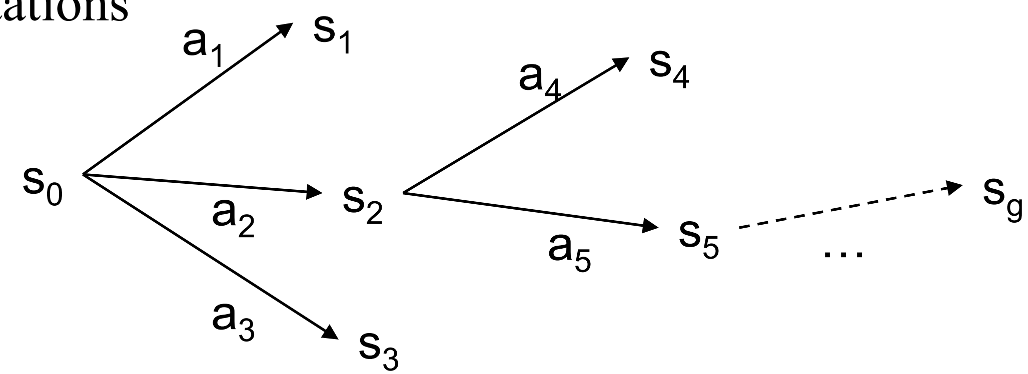


Properties

- 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.

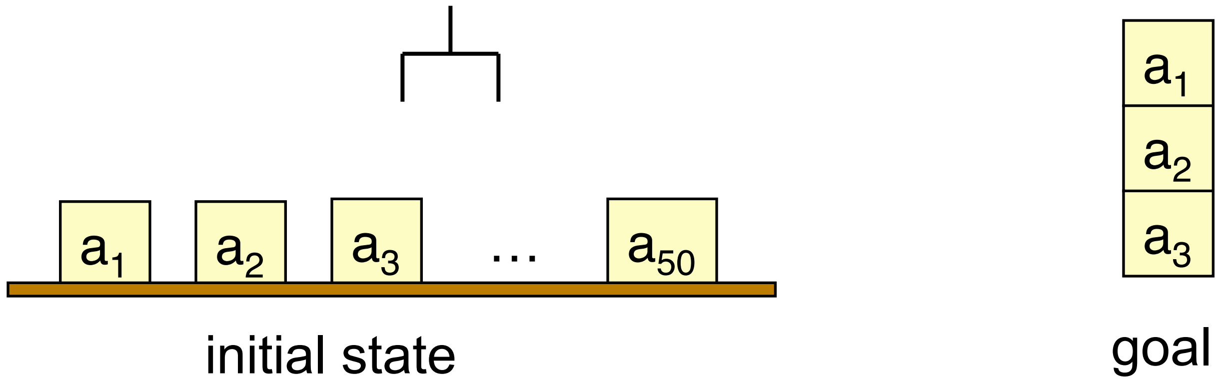
Deterministic Implementations

- 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 they usually aren't practical because they require too much memory
  - Memory requirement is exponential in the length of the solution
- In practice, more likely to use depth-first search or greedy search
  - Worst-case memory requirement is linear in the length of the solution
  - In general, sound but not complete
    - » But classical planning has only finitely many states
    - » Thus, can make depth-first search complete by doing loop-checking

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
  - See Section 4.5 (Domain-Specific State-Space Planning) and Part III (Heuristics and Control Strategies)

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(O, s_0, g)

$\pi \leftarrow$ the empty plan

loop

if s_0 satisfies g then return π

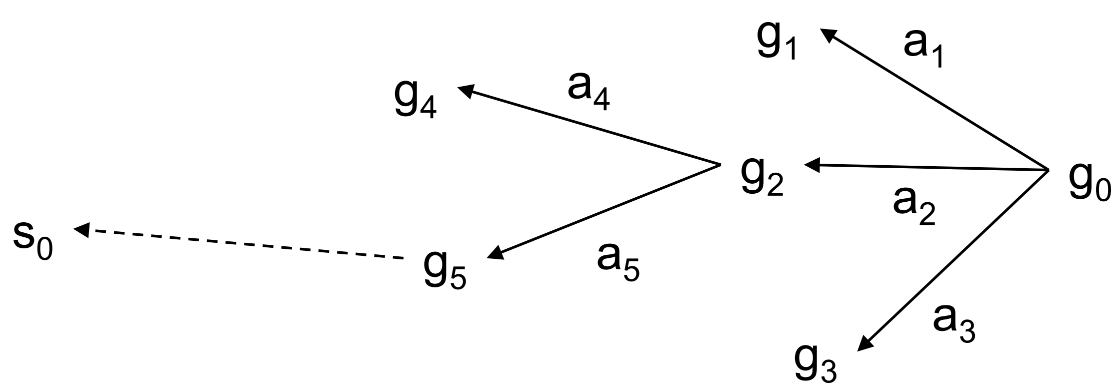
$A \leftarrow \{a | a$ is a ground instance of an operator in O and $\gamma^{-1}(g, a)$ is defined$\}$

if $A = \emptyset$ then return failure

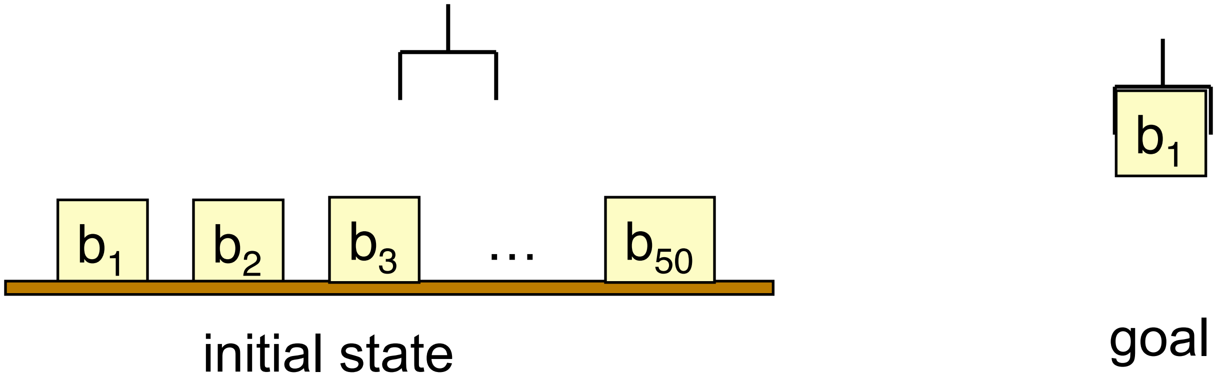
nondeterministically choose an action $a \in A$

$\pi \leftarrow a.\pi$

$g \leftarrow \gamma^{-1}(g, a)$

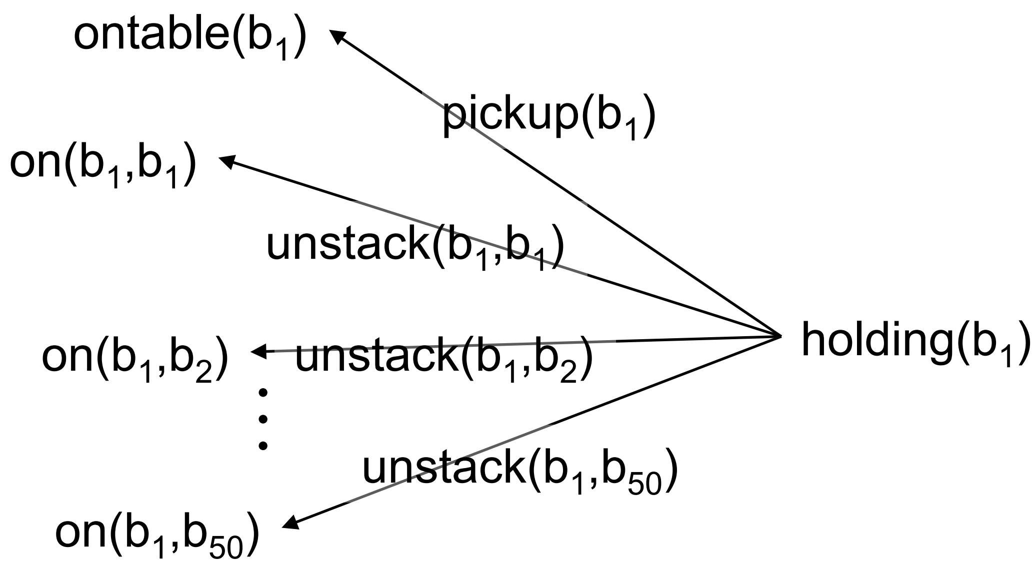


Efficiency of Backward Search

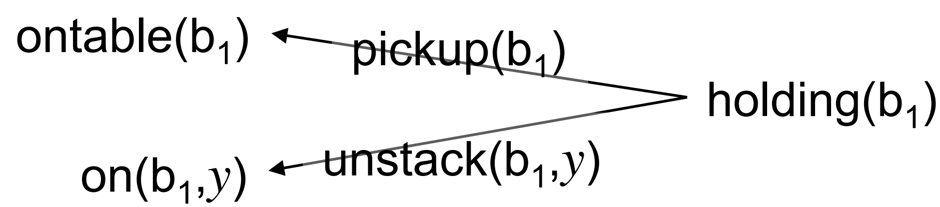


- Backward search can *also* have a very large branching factor
  - E.g., an operator o that is relevant for g may have many ground instances $a_1, a_2, \ldots, a_n$ such that each a_i's input state might be unreachable from the initial state
- As before, deterministic implementations can waste lots of time trying all of them

Lifting



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



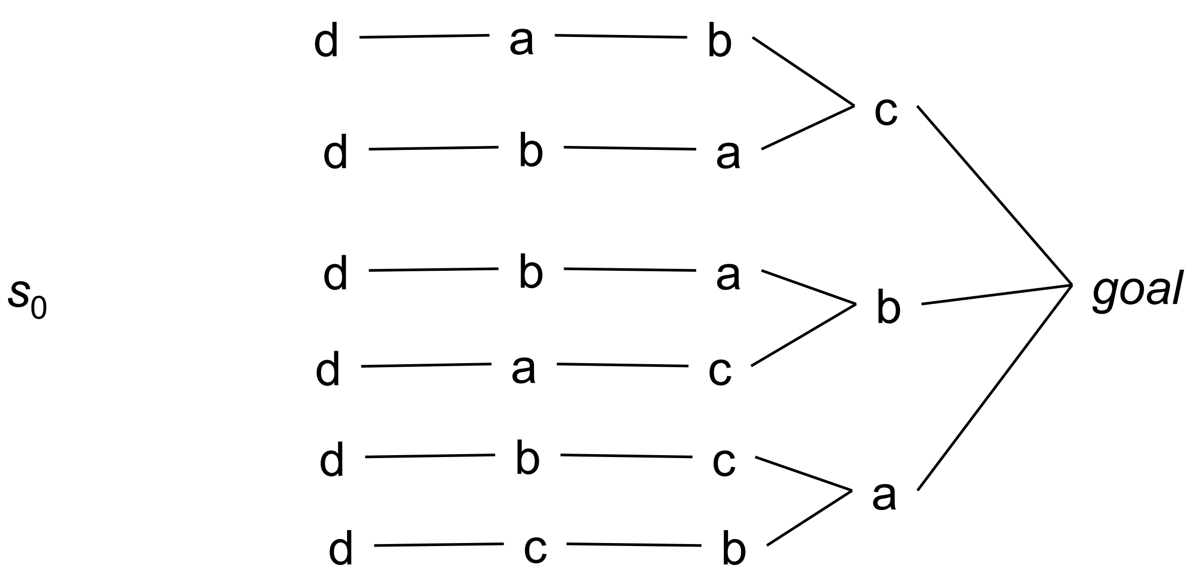
Lifted Backward Search

- 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))
```

The Search Space is Still Too Large

- Lifted-backward-search generates a smaller search space than Backward-search, but it still can be quite large
  - Suppose actions *a*, *b*, and *c* are independent, action *d* must precede all of them, and there's no path from s_0 to *d*'s input state
  - We'll try all possible orderings of *a*, *b*, and *c* before realizing there is no solution
  - More about this in Chapter 5 (Plan-Space Planning)



Pruning the Search Space

- I'll say a lot about this later, in Part III of the book
- For now, just two examples:
  - STRIPS
  - Block stacking

STRIPS

- Basic idea: given a compound goal $g = \{g_1, \mathrm{g}_1, \ldots\}$, try to solve each g_i separately
  - Works if the goals are *serializable* (can be solved in some linear order)

$\pi \leftarrow$ the empty plan

do a modified backward search from g:

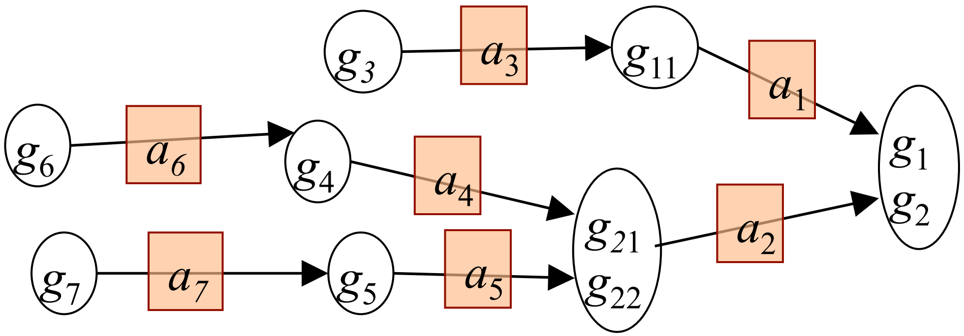
instead of $\gamma^{-1}(s,a)$, each new set of subgoals is just precond(a)

whenever you find an action that's executable in the current state,

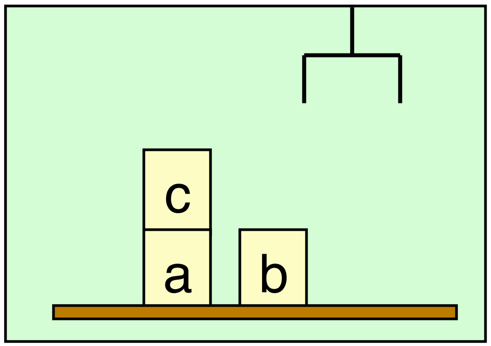
go forward on the current search path as far as possible, executing actions and appending them to π

repeat until all goals are satisfied

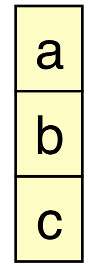
$$\pi = \langle \pi_1, \pi_2 \rangle \text{ or } \langle \pi_2, \pi_1 \rangle$$
$$\pi_2 = \langle \pi_{11}, \pi_{12}, a_2 \rangle \text{ or } \langle \pi_{12}, \pi_{11}, a_2 \rangle$$
$$\pi_{21} = \langle a_7, a_4 \rangle$$
$$\pi_{22} = \langle a_7, a_5 \rangle$$



The Sussman Anomaly



Initial state



goal

- On this problem, STRIPS can't produce an irredundant solution
  - Try it and see

The Register Assignment Problem

- Interchange the values stored in two registers
  - State-variable formulation:
    - registers r1, r2, r3

s_0: {value(r1)=3, value(r2)=5, value(r3)=0}

g: {value(r1)=5, value(r2)=3}

Operator: assign(r,v,r',v')

precond: value(r)=v, value(r')=v'

effects: value(r)=v'

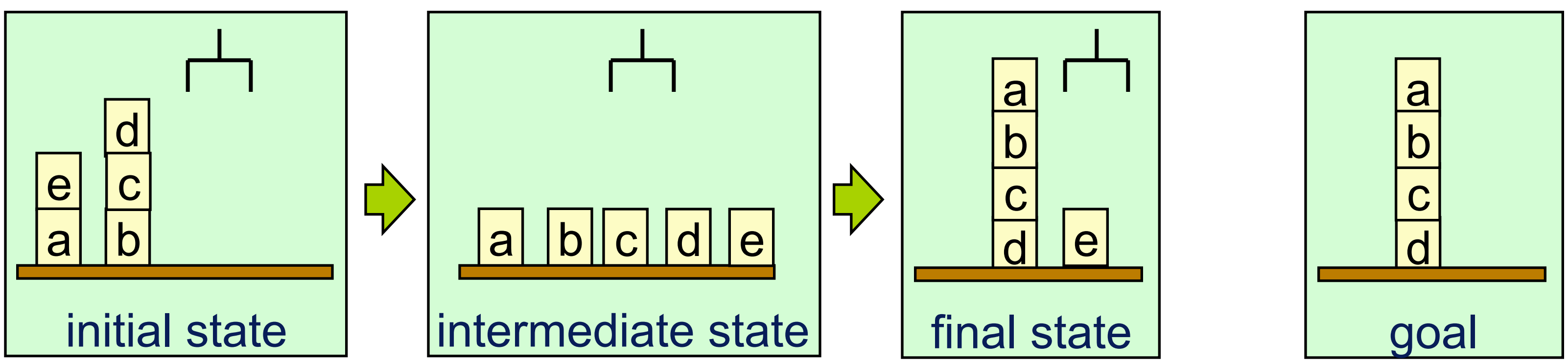
- STRIPS cannot solve this problem at all

How to Handle Problems like These?

- Several ways:
  - Use a planning algorithm other than state-space search
    - e.g., Chapters 5–8
  - Write a domain-specific algorithm
    - Example: the blocks world

Domain-Specific Knowledge

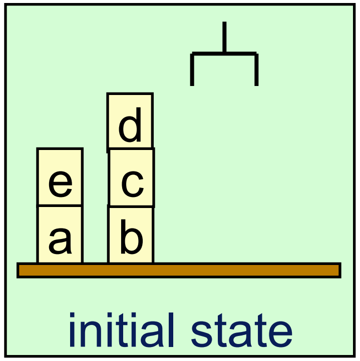
- A blocks-world planning problem $P = (O, s_0, g)$ is solvable iff s_0 and g satisfy some simple consistency conditions
  - no block can be on two other blocks at once, every block in g must also be in s_0, etc.
    - Can check these in time O(n log n)
- If P is solvable, can easily construct a solution 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 *(next slide)*

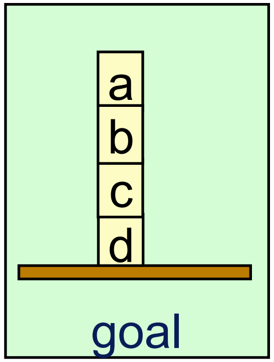


Block-Stacking Algorithm

- All of the possible situations in which a block x needs to be moved:
  - s contains ontable(x) and g contains on(x,y) - e.g., a
  - s contains on(x,y) and g contains ontable(x) - e.g., d
  - s contains on(x,y) and g contains on(x,z) for some $y \neq z$ - e.g., c
  - s contains on(x,y) and y needs to be moved - e.g., e

loop
  if there is a clear block x that 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's a clear block x that needs to be moved
    then move x to the table
  else if the goal is satisfied **then return** the plan
  else return failure
repeat





Properties of the Block-Stacking Algorithm

- Sound, complete, guaranteed to terminate

- Easily solves problems like the Sussman anomaly

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

- Often finds optimal (shortest) solutions
- But sometimes only near-optimal (Exercise 4.22 in the book)
  - Recall that PLAN LENGTH for the blocks world is NP-complete