State-space and Plan-space Planning Algorithms

based on Dana S. Nau, University of Maryland,
revised and presented by Michal Pechoucek, CTU in Prague

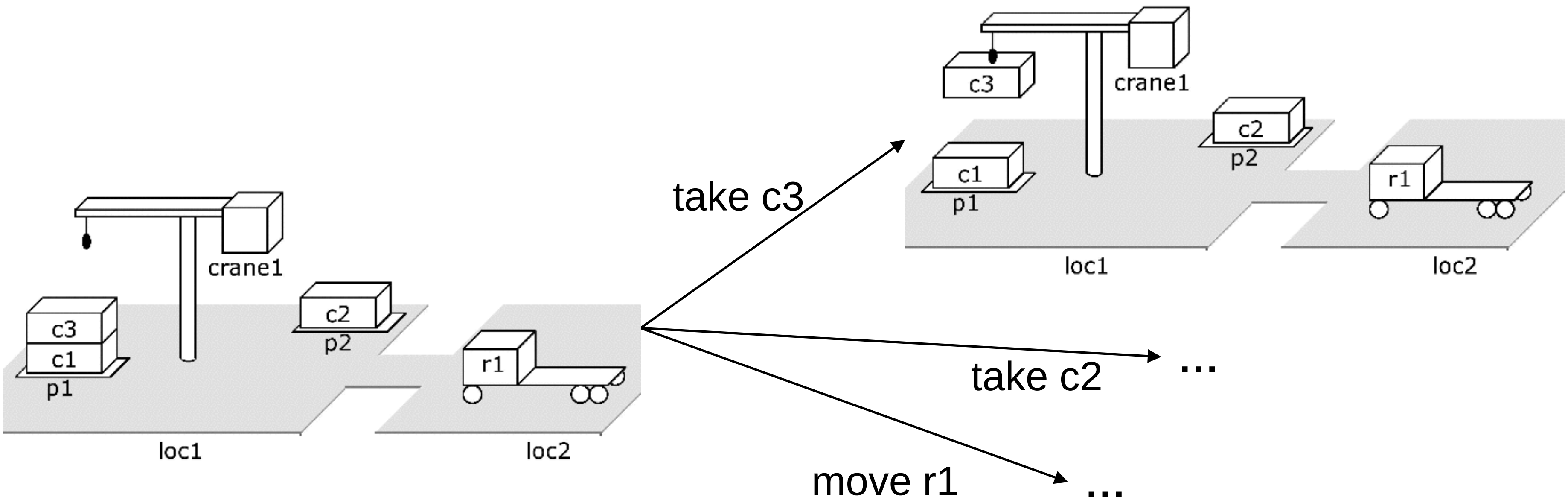
Motivation

- 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

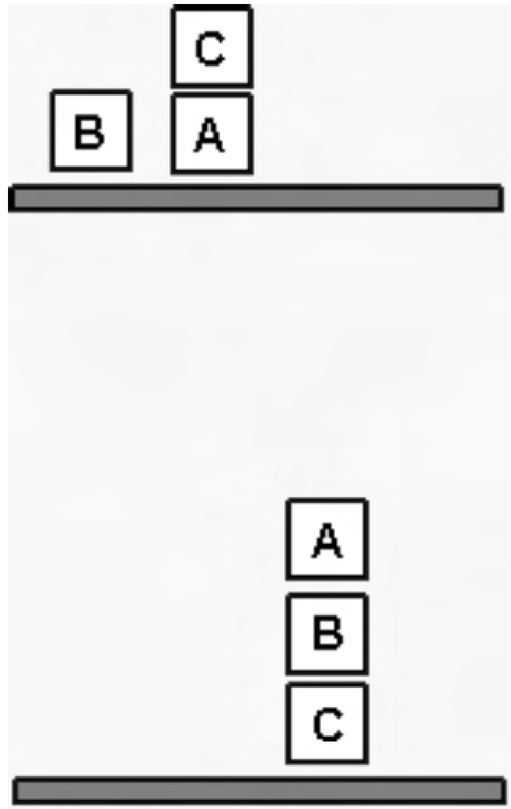
Forward Search



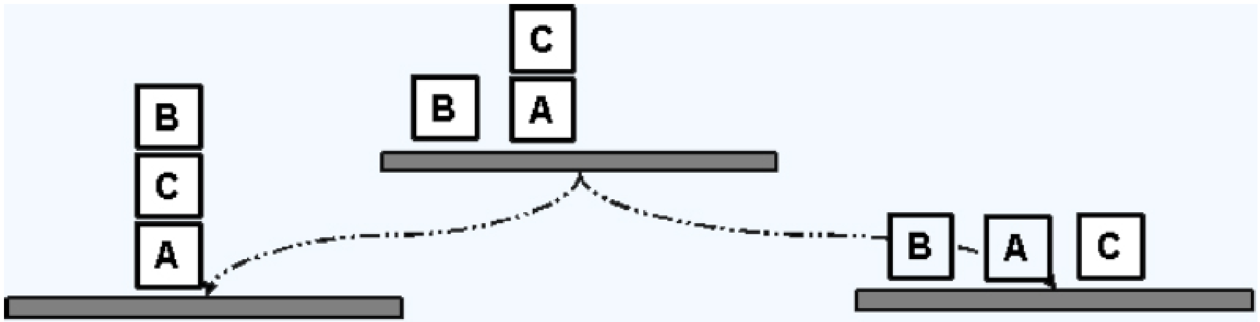
Forward Search

```
Forward-search(O, s₀, g)
    s ← s₀
    π ← the empty plan
    loop
        if s satisfies g then return π
        E ← {a|a is a ground instance an operator in O,
                  and precond(a) is true in s}
        if E = ∅ then return failure
        nondeterministically choose an action a ∈ E
        s ← γ(s, a)
        π ← π.a
```

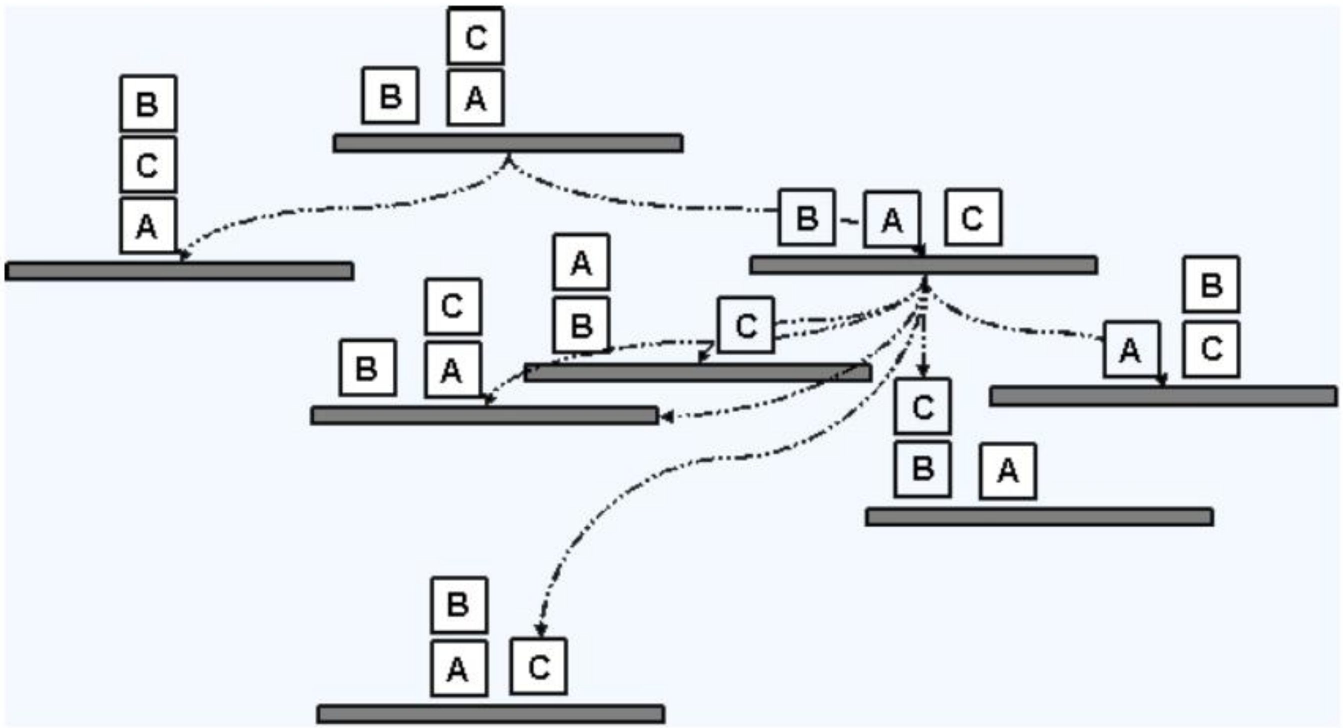
Forward Search



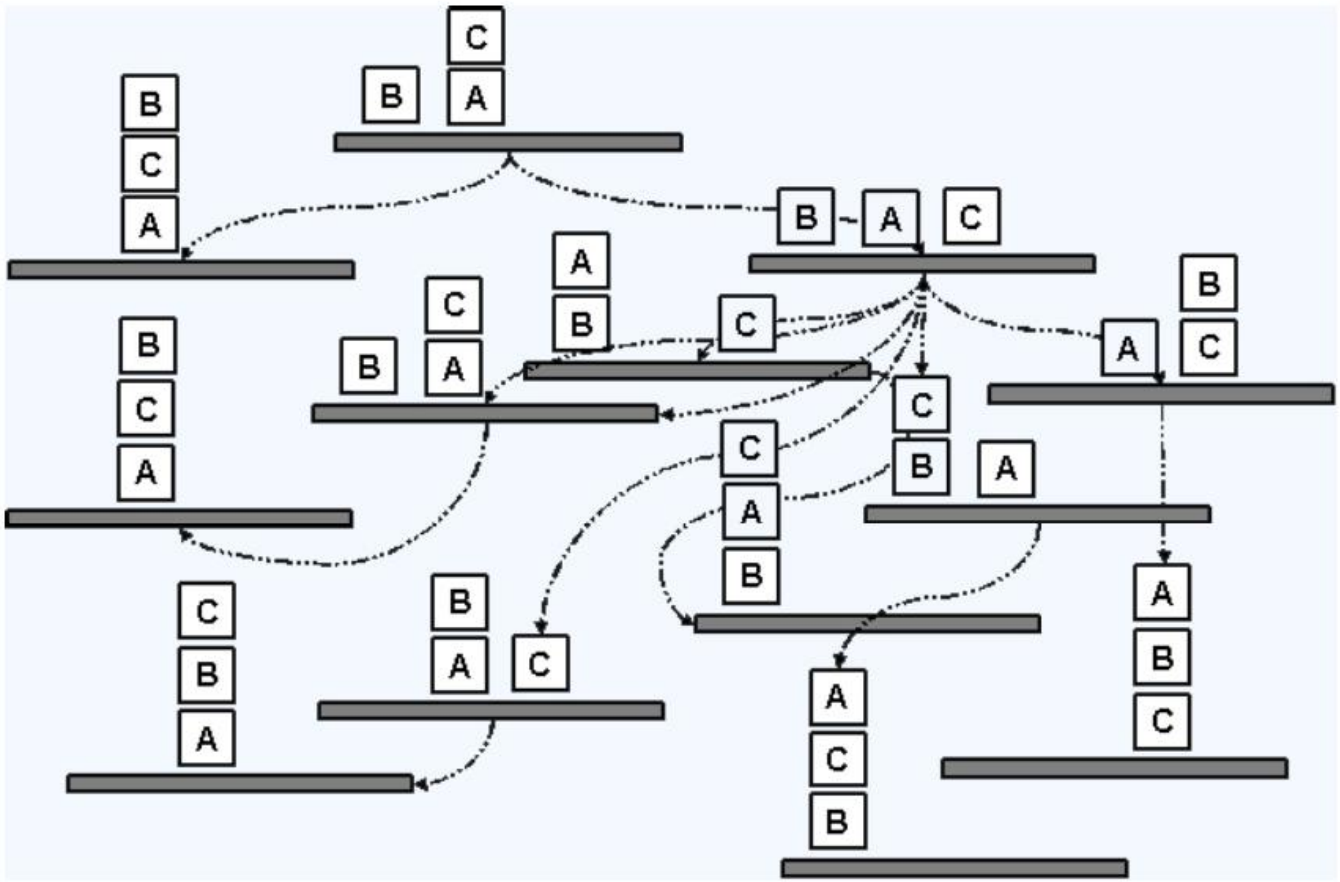
Forward Search



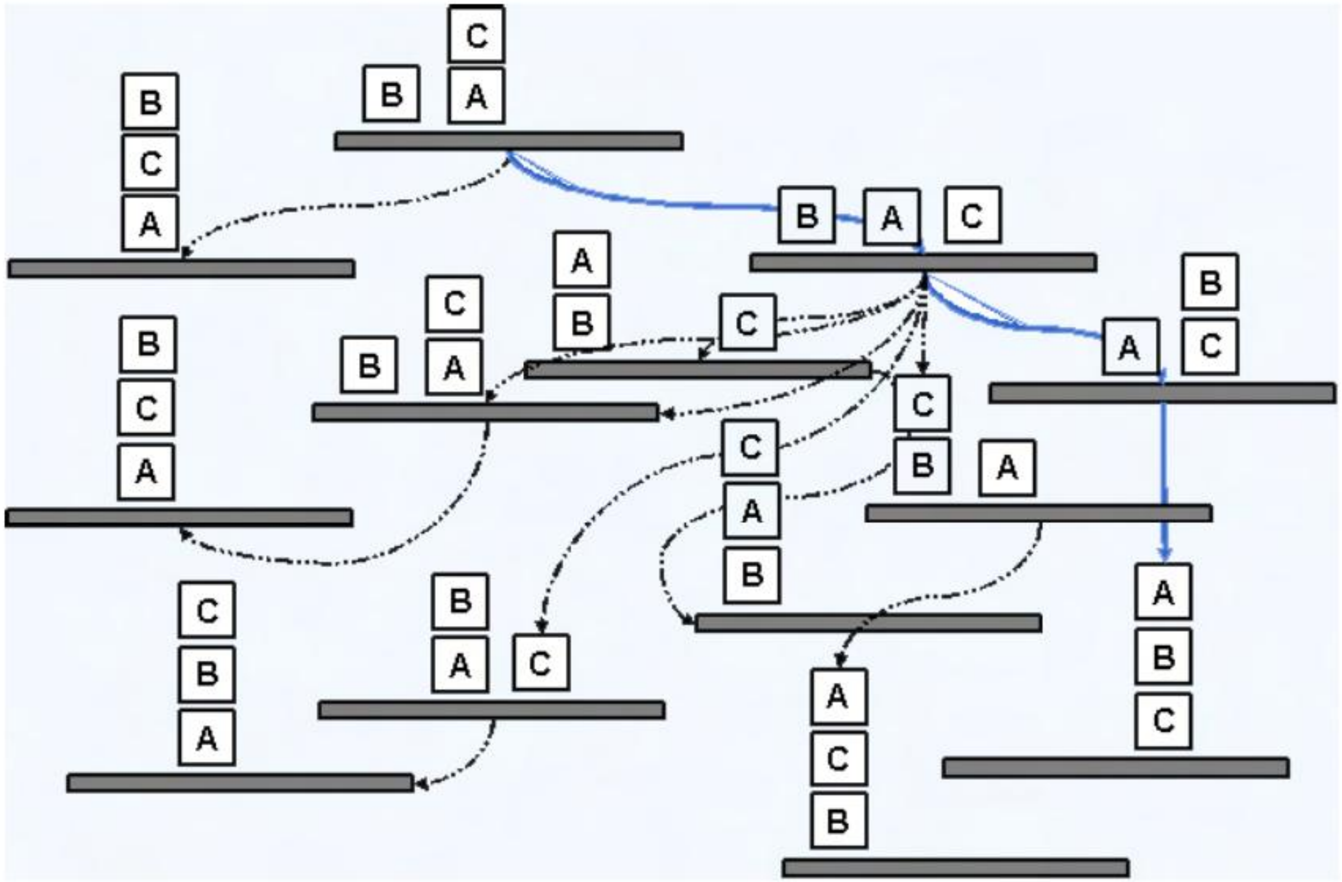
Forward Search



Forward Search



Forward Search

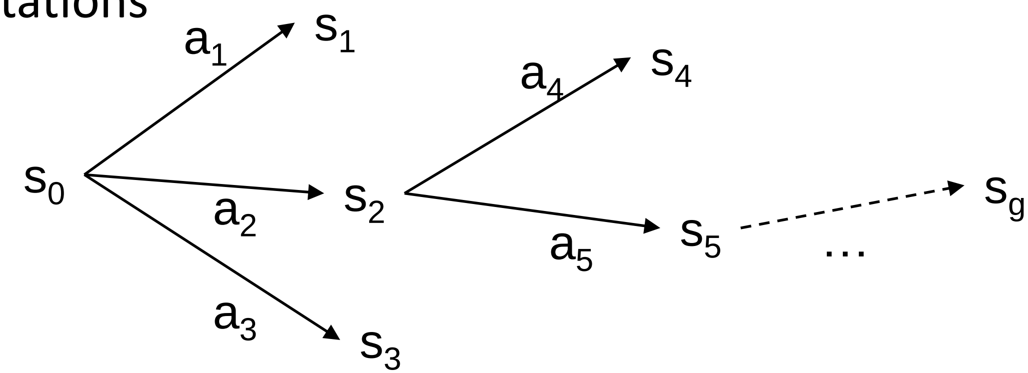


Properties

- 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.

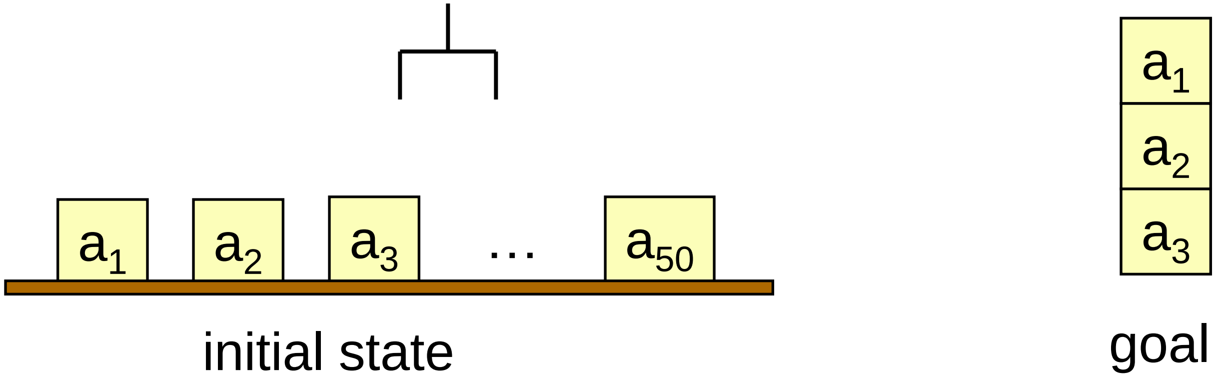
Deterministic Implementations

– 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

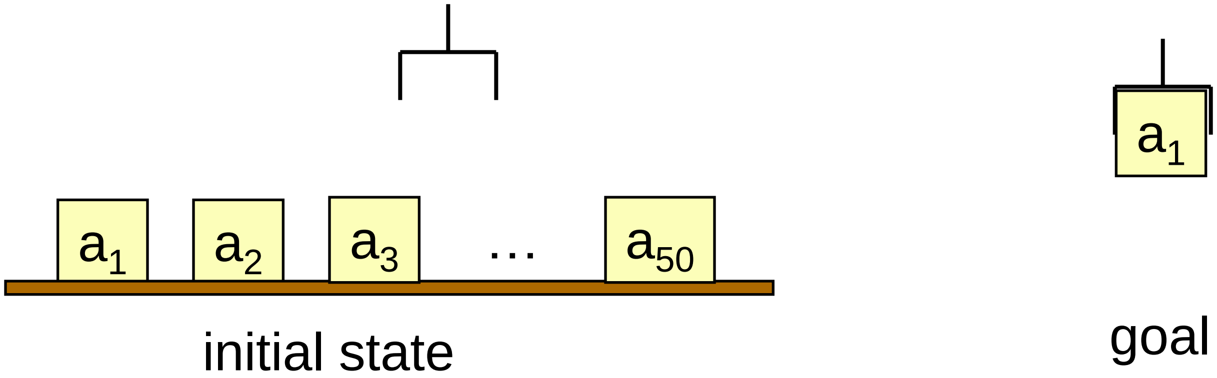
- If a is relevant for g, then
  - » $\gamma^{-1}(g,a) = (g - \text{effects(a)}) \cup \text{precond}(a)$
- Otherwise $\gamma^{-1}(g,a)$ is undefined

- Example: suppose that
  - » g = {on(b1,b2), on(b2,b3)}
  - » a = stack(b1,b2)
- What is $\gamma^{-1}(g,a)$?

Backward Search

```
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)
```

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, …, 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
- Backward-search is *sound and complete*

Pruning the Search Space

- Lifting
- STRIPS
- Block stacking

Lifted Backward Search

- We can reduce the branching factor if we *partially* instantiate the operators
  - this is called *lifting*
- More complicated than Backward-search (keeps track of what substitutions were performed), but it has a much smaller branching factor

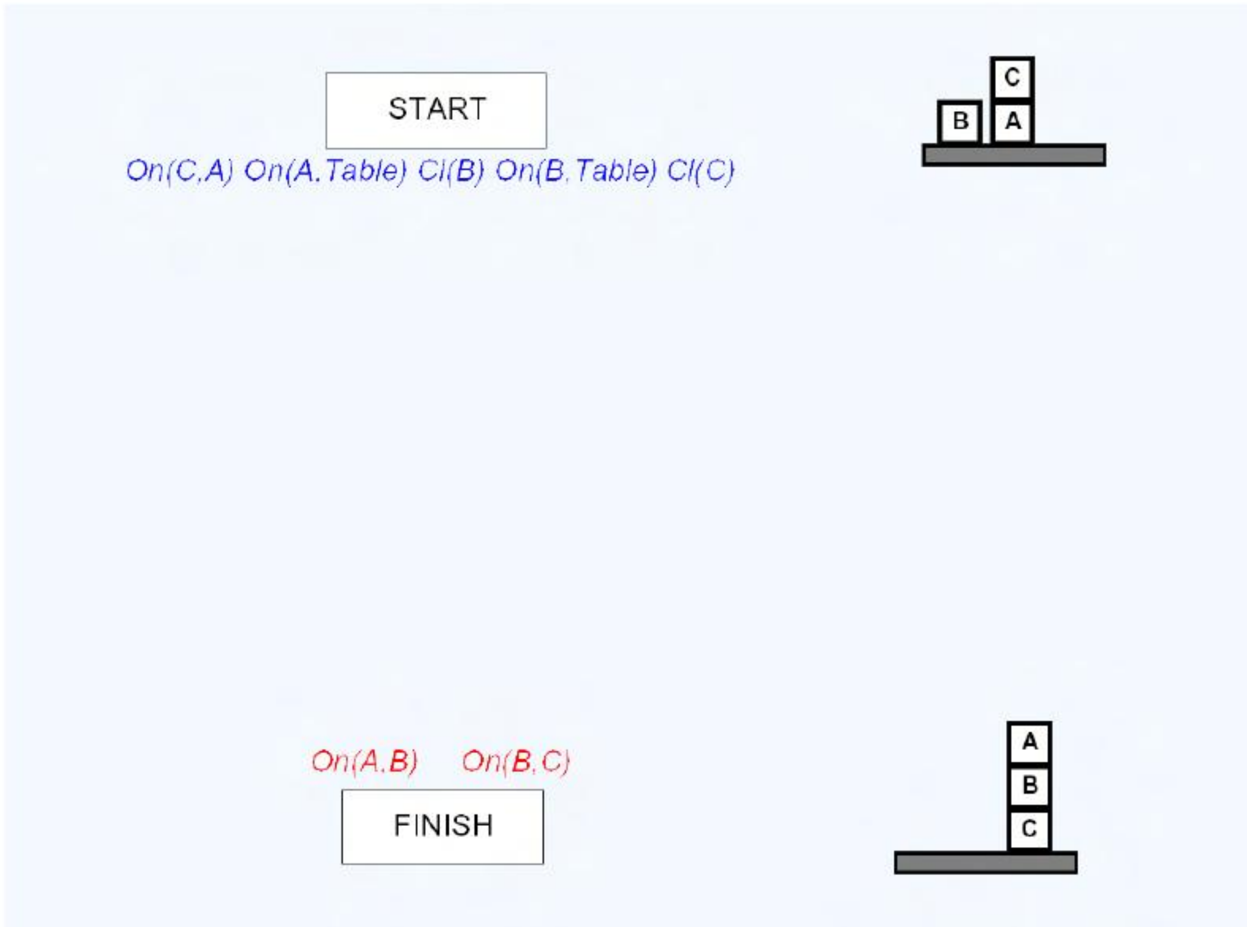
```
Lifted-backward-search(O, s_0, g)
  π ← the empty plan
  loop
    if s_0 satisfies g then return π
    A ← {(o, θ) | o is a standardization of an operator in O,
              θ is an mgu for an atom of g and an atom of effects⁺(o),
              and γ⁻¹(θ(g), θ(o)) is defined}
    if A = ∅ then return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

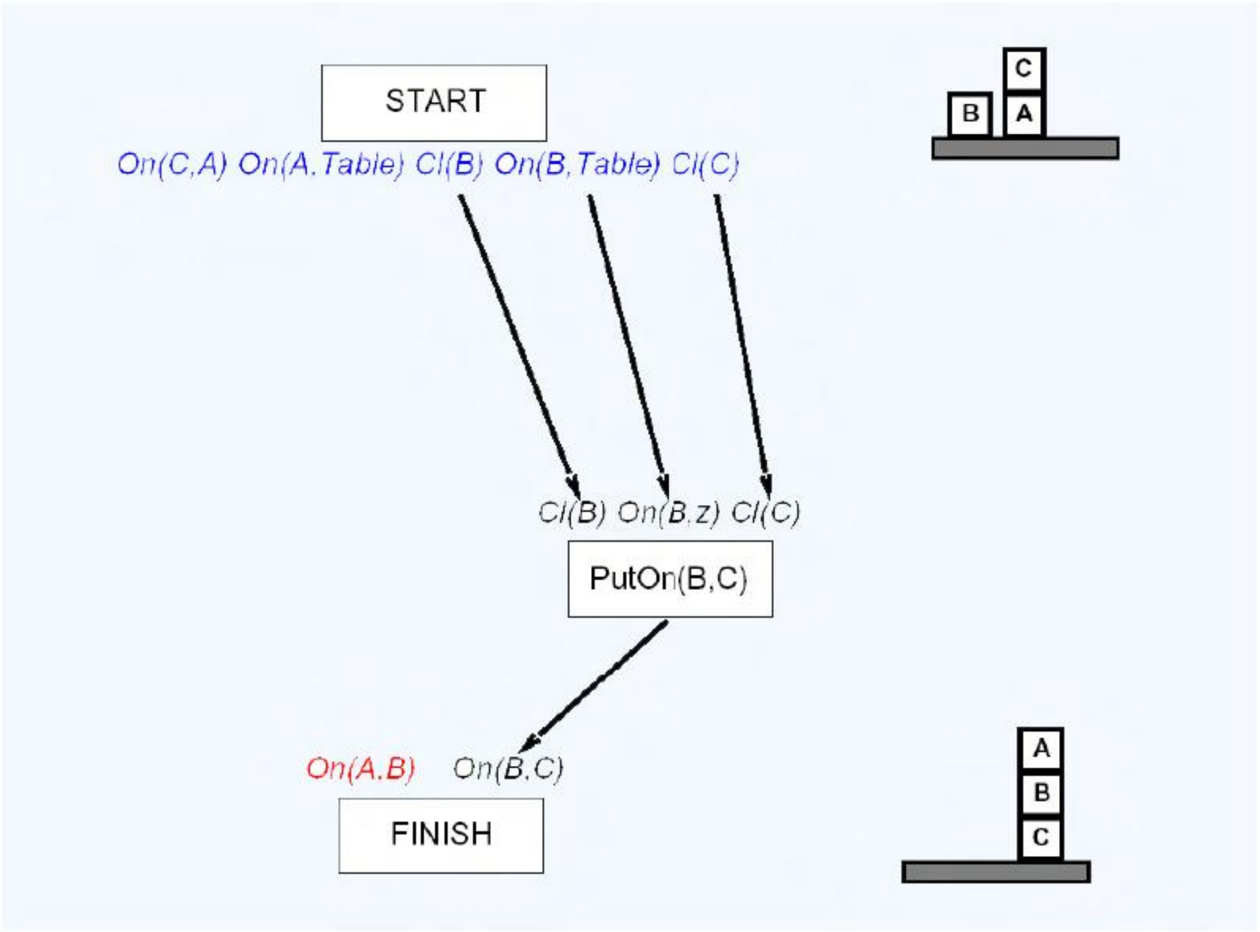
- $\pi \leftarrow$ the empty plan
- do a modified backward search from *g*
  - » instead of $\gamma^{-1}(s,a)$, each new set of subgoals is just precond(*a*)
  - » choose one of them to achieve
  - » If it is not already achieved
    - choose an action that makes the goal true
    - achieve the preconditions of the action
    - carry out the action
  - » achieve the rest of the goals.

- The STRIPS algorithm, as presented, is unsound.
- Achieving one subgoal may undo already achieved subgoals.

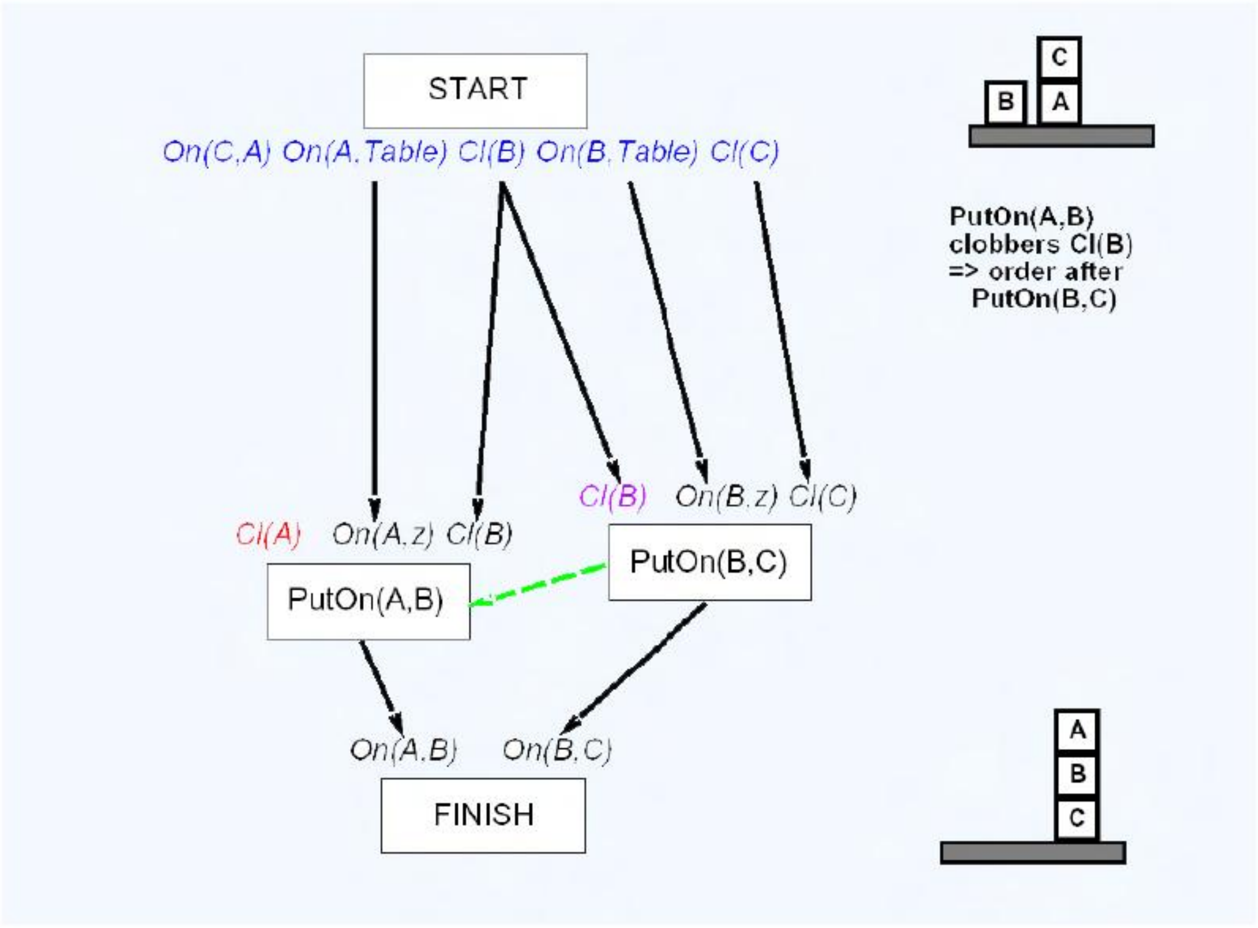
Example – Sussman Anomaly



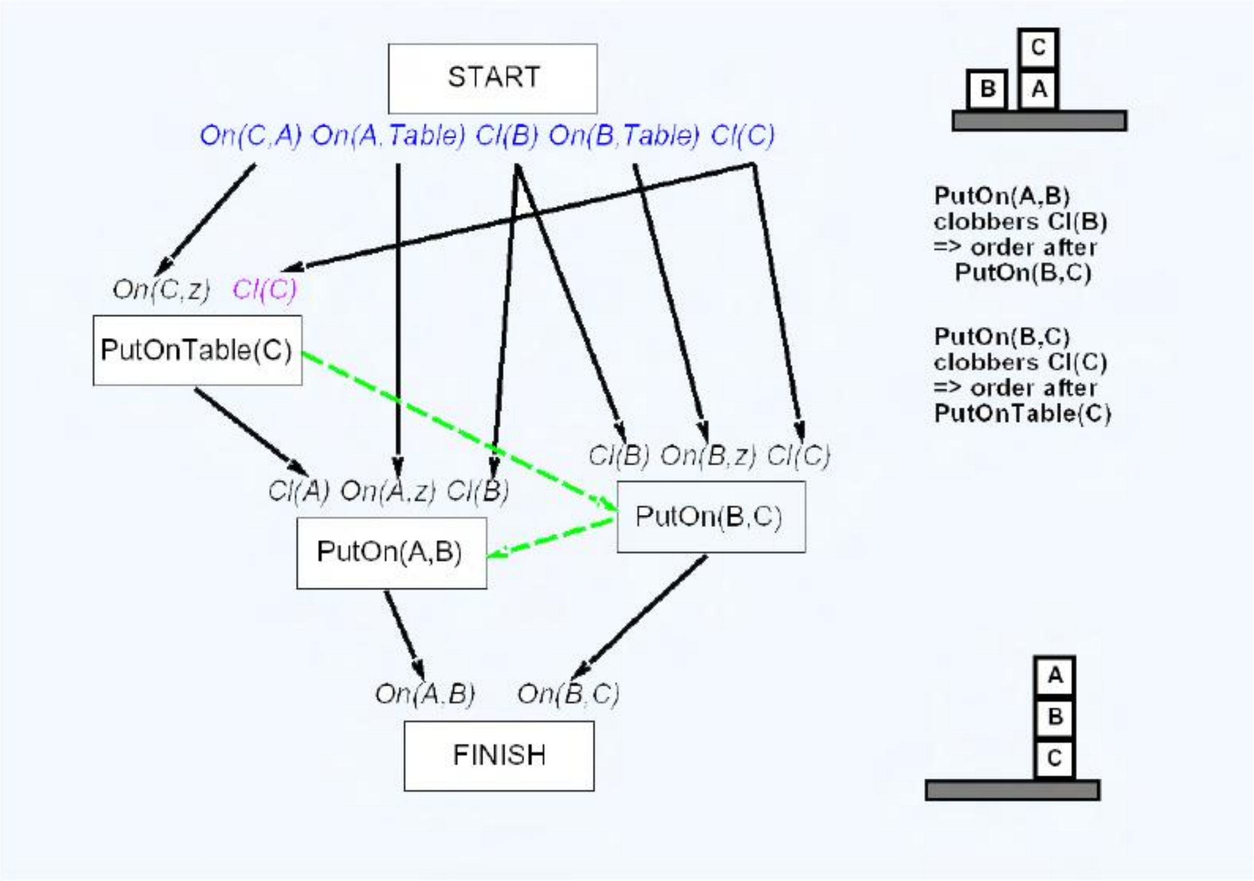
Example – Sussman Anomaly



Example – Sussman Anomaly



Example – Sussman Anomaly

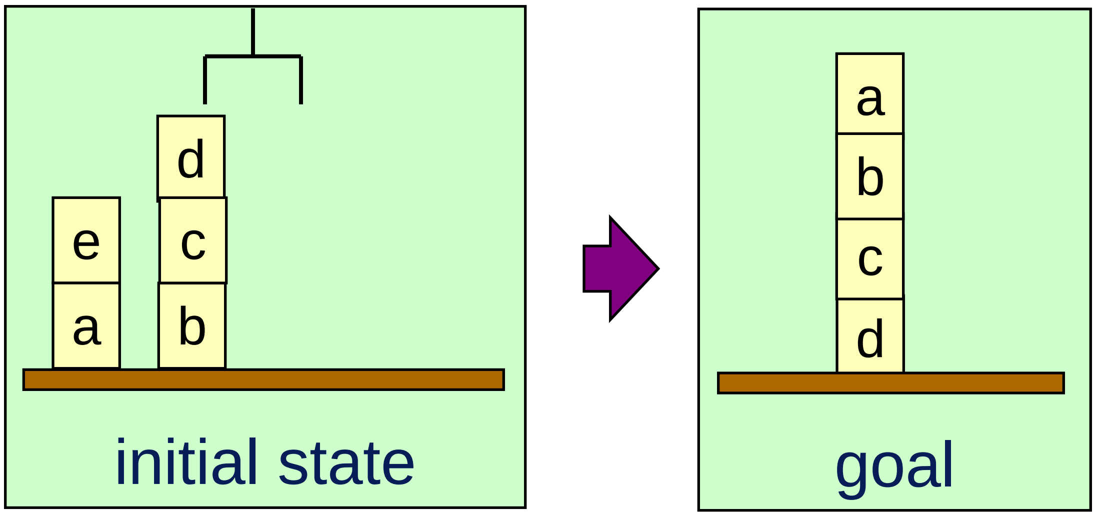


How to Handle Problems like These?

- How to make STRIPS sound?
  - *Protect subgoals* so that, once achieved, until they are needed, they cannot be undone.
    - Protecting subgoals makes STRIPS incomplete.
  - *Reachieve subgoals* that have been undone.
    - Reachieving subgoals finds longer plans than necessary.
  - Use *domain-specific* knowledge to prune the search space
    - Can solve both problems quite easily this way
    - Example: block stacking using forward search
  - Use methods for *causal links thread resolution*

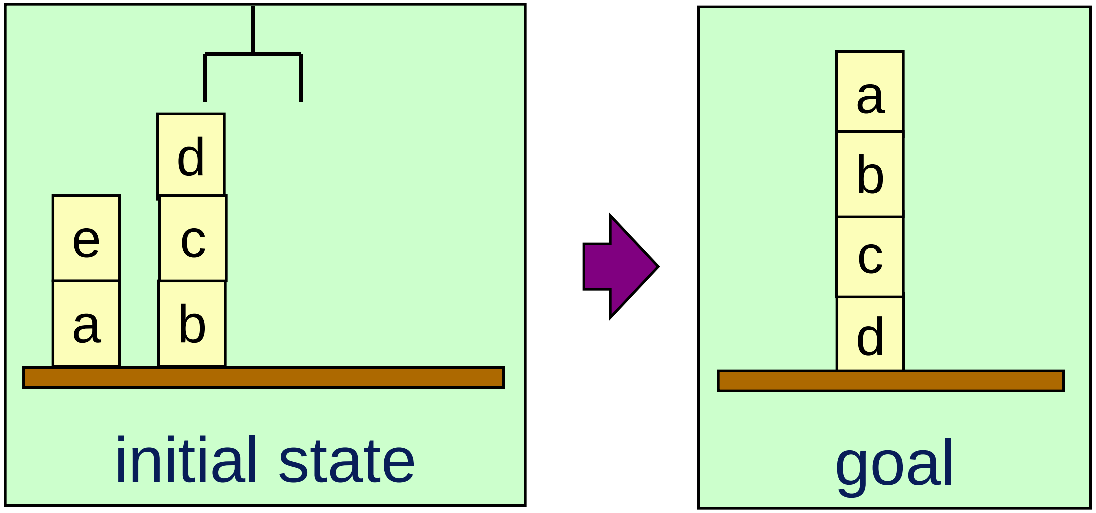
Additional Domain-Specific Knowledge

- 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*≠*z*
    - see c below
  - » *s* contains on(*x*,*y*) and *y* needs to be moved - see e below



Domain-Specific *Block Stacking* 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



Easily Solves the Sussman Anomaly

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

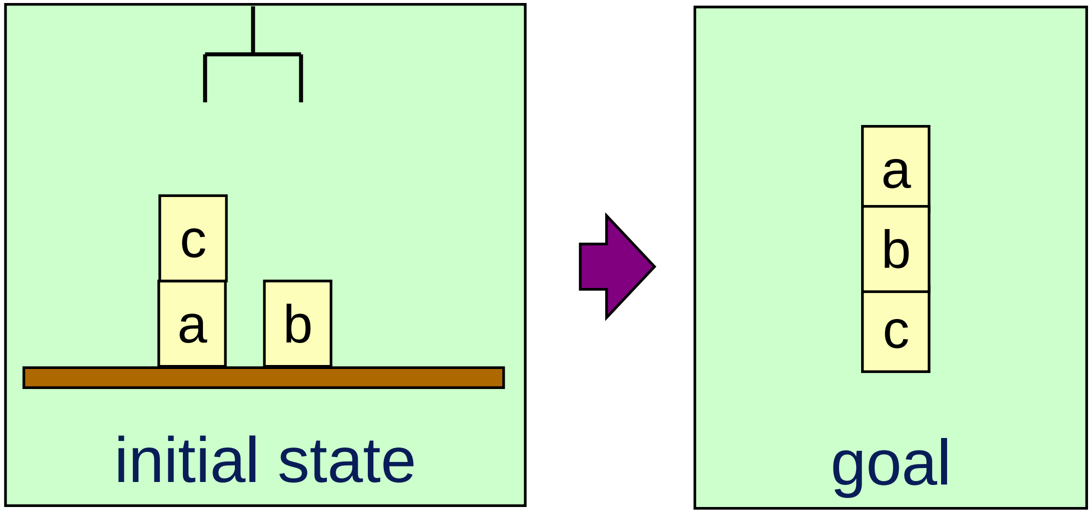
then move *x* to the table

else if the goal is satisfied

then return the plan

else return failure

repeat

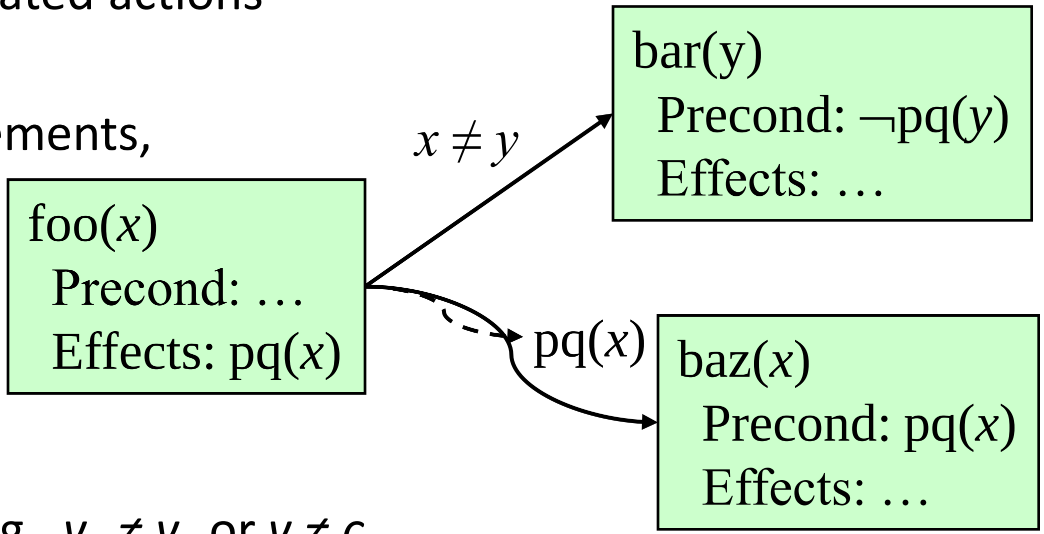


Properties

- The block-stacking algorithm:
  - Sound, complete, guaranteed to terminate
  - 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

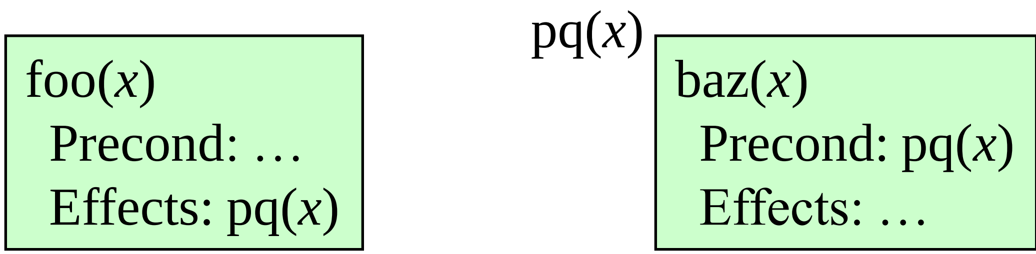
Plan Space Planning (PSP)

- Backward search from the goal
- Each node of the search space is a *partial plan*
    - A set of partially-instantiated actions
    - A set of constraints
    - » Make more and more refinements, until we have a solution
- Types of **constraints**:
    - » *precedence constraint*: *a* must precede *b*
    - » *binding constraints*:
        - inequality constraints, e.g., $v_1 \neq v_2$ or $v \neq c$
        - equality constraints (e.g., $v_1 = v_2$ or $v = c$) or substitutions
    - » *causal link*:
        - use action *a* to establish the precondition *p* needed by action *b*
- How to tell we have a solution: no more *flaws* in the plan
    - » Will discuss flaws and how to resolve them

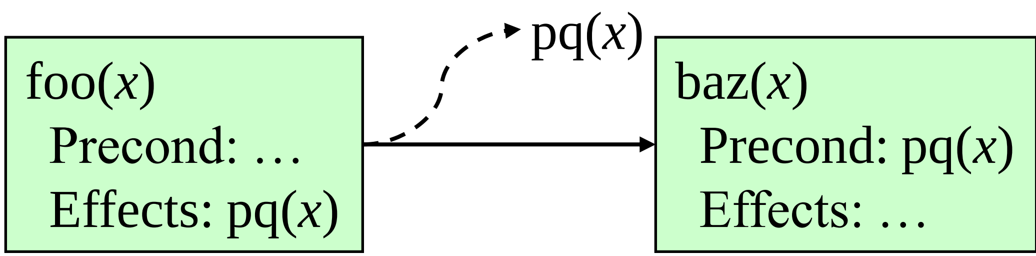


Flaws: 1. Open Goals

- Open goal:
  - » An action a has a precondition p that we haven't decided how to establish



- Resolving the flaw:
  - » Find an action b
    - – (either already in the plan, or insert it)
  - » that can be used to establish p
    - – can precede a and produce p
  - » Instantiate variables
  - » Create a causal link



Flaws: 2. Causal Link Threats

- **Causal Link Formally:** due to the properties of the ordering relation:

  $$\forall \alpha_1, \alpha_2 \in \pi : \exists x : x \in \mathtt{pre}(\alpha_2) \wedge x \in \mathtt{eff}(\alpha_1) \Leftrightarrow \alpha_1 \prec \alpha_2$$

  we introduce causal link as satisfiability relation among operators

  $$\alpha_1 \underset{x}{\rightarrow} \alpha_2, \text{ where } x \in \mathtt{eff}(\alpha_1) \;\wedge\; x \in \mathtt{pre}(\alpha_2) \;\wedge\; \alpha_1 \prec \alpha_2$$

  to be read as 1 achieves x for 2 the fact x is that true allows carrying out 2 provided that 1 has been already achieved

- **Causal link threat:**

  negative thread of causal link: $\alpha_1 \prec \alpha_2, \alpha_2 \prec \alpha_3$ and $\alpha_1 \underset{q}{\rightarrow} \alpha_3$ are consistent in a plan and there is an effect $q \in (\mathtt{eff}\ \alpha_2)$ so that $\neg q \in (\mathtt{pre}\ \alpha_3)$

  positive causal thread is defined similarly

- **Causal link threat resolution:**
  additional ordering – *demotion* $\alpha_3 \prec \alpha_2$ or *promotion* $\alpha_2 \prec \alpha_1$ or constrain variable binding preventing the threat

The PSP Procedure

PSP(π)

$flaws \leftarrow$ OpenGoals(π) $\cup$ Threats(π)
if $flaws = \emptyset$ then return(π)
select any flaw $\phi \in flaws$
$resolvers \leftarrow$ Resolve(ϕ, π)
if $resolvers = \emptyset$ then return(failure)
nondeterministically choose a resolver $\rho \in resolvers$
$\pi' \leftarrow$ Refine(ρ, π)
return(PSP(π'))

end

– PSP is both sound and complete

Example

- Similar (but not identical) to an example in Russell and Norvig's *Artificial Intelligence: A Modern Approach* (1st edition)
- Operators:
  - **Start**

    Precond: none

    Effects: At(Home), sells(HWS,Drill), Sells(SM,Milk), Sells(SM,Banana)
  - **Finish**

    Precond: Have(Drill), Have(Milk), Have(Banana), At(Home)
  - **Go(*l*,*m*)**

    Precond: At(*l*)

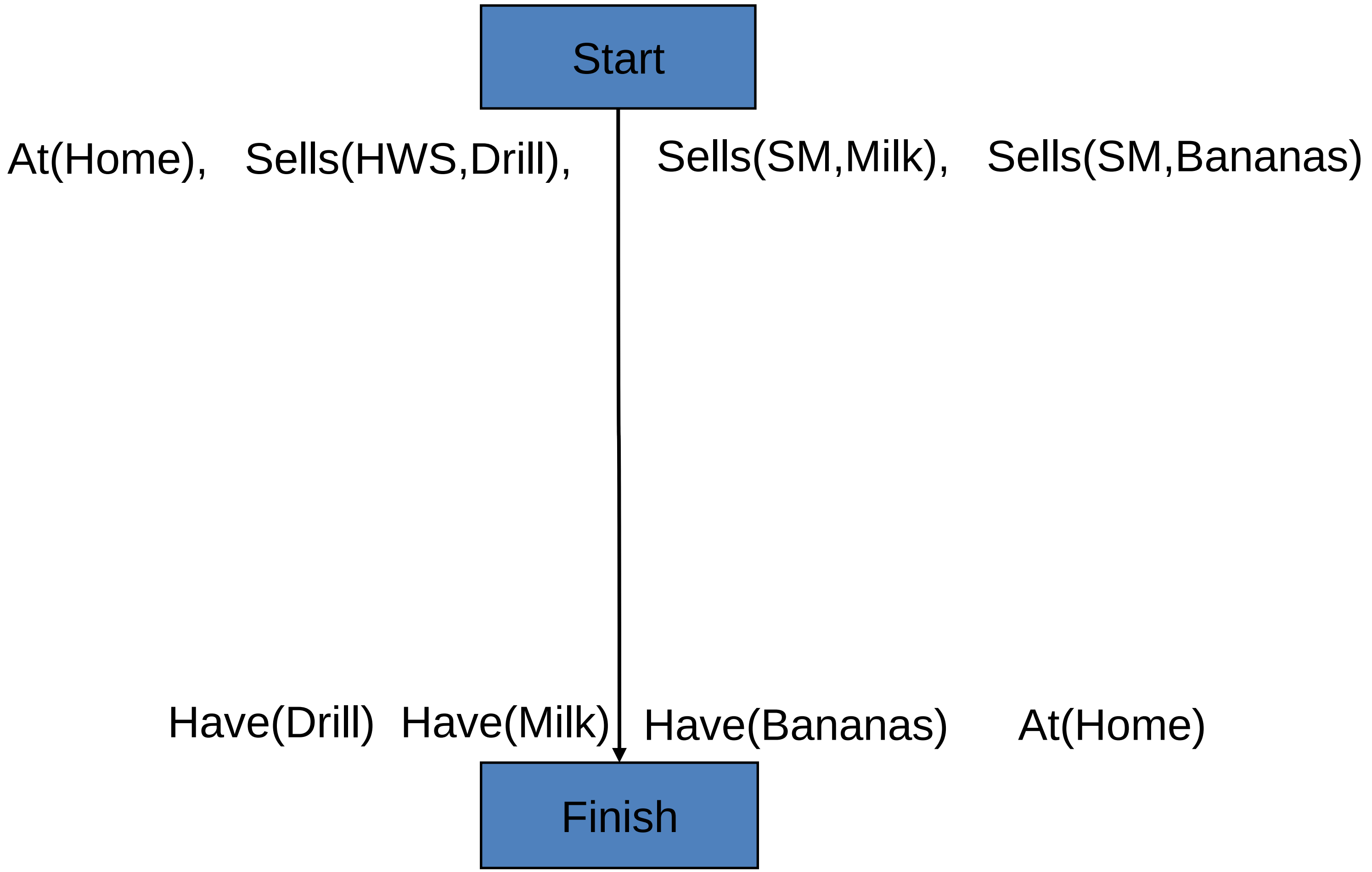
    Effects: At(*m*), ¬At(*l*)
  - **Buy(*p*,*s*)**

    Precond: At(*s*), Sells(*s*,*p*)

    Effects: Have(*p*)

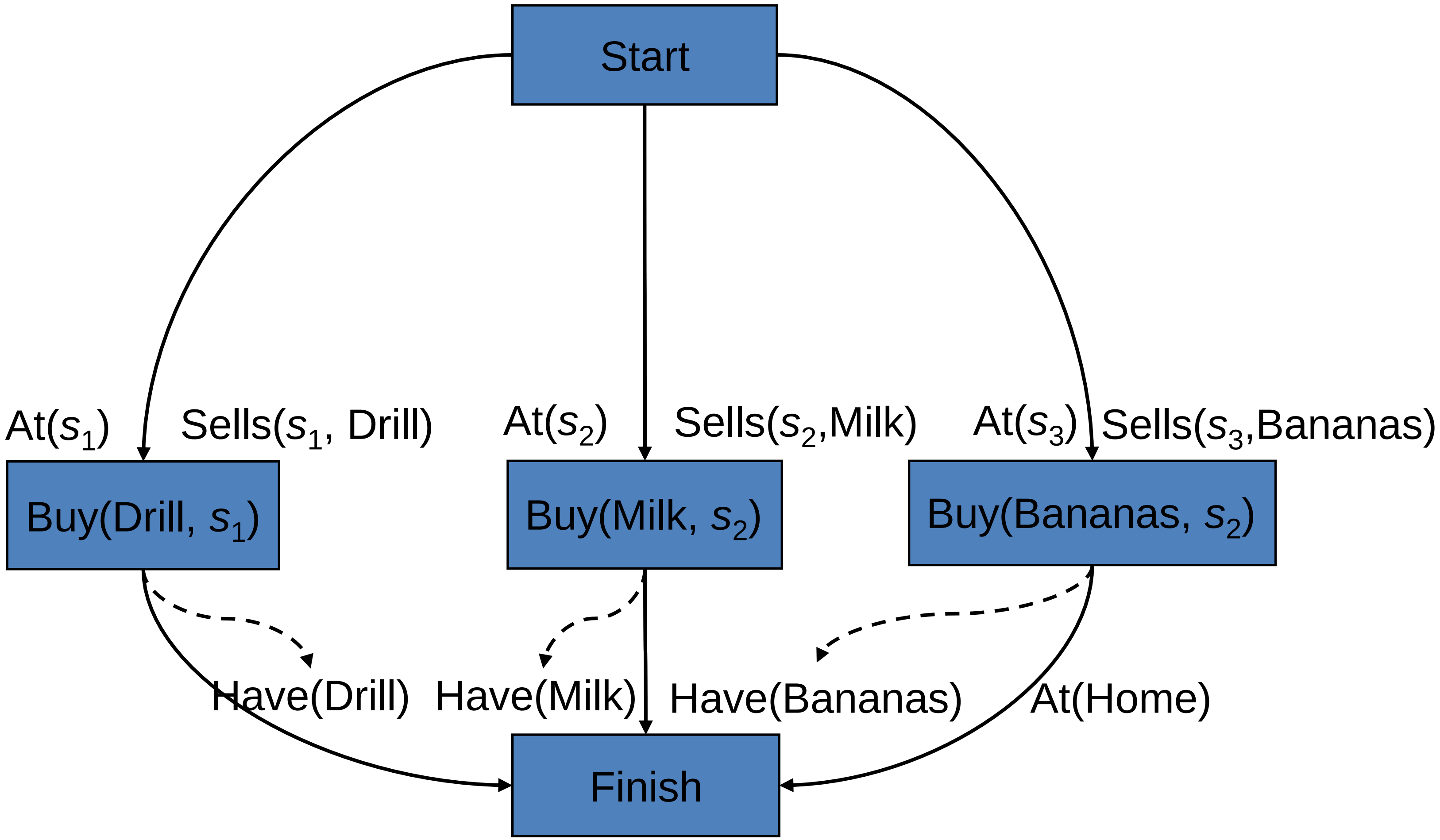
Example (continued)

– Initial plan



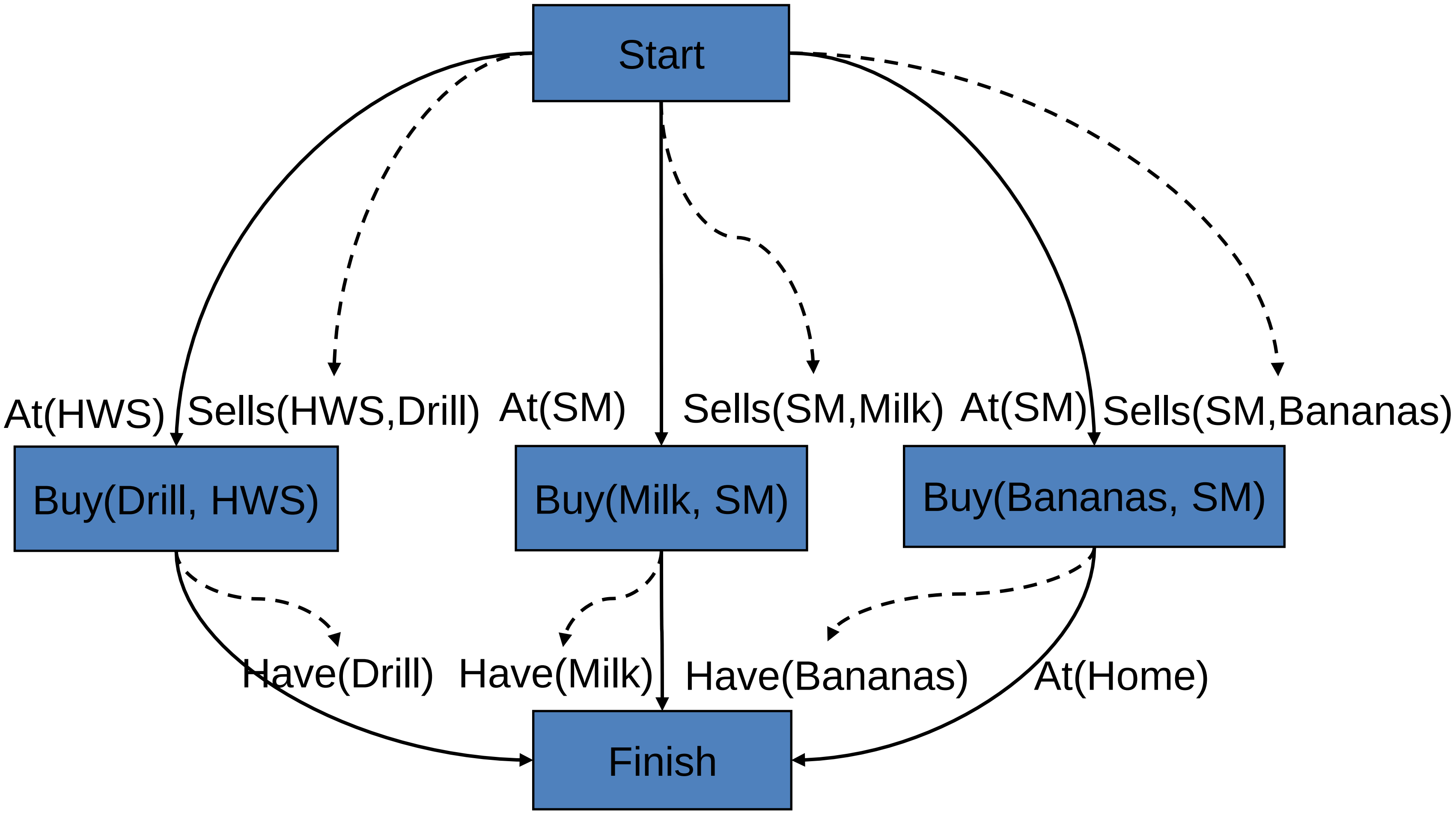
Example (continued)

– The only possible ways to establish the Have preconditions



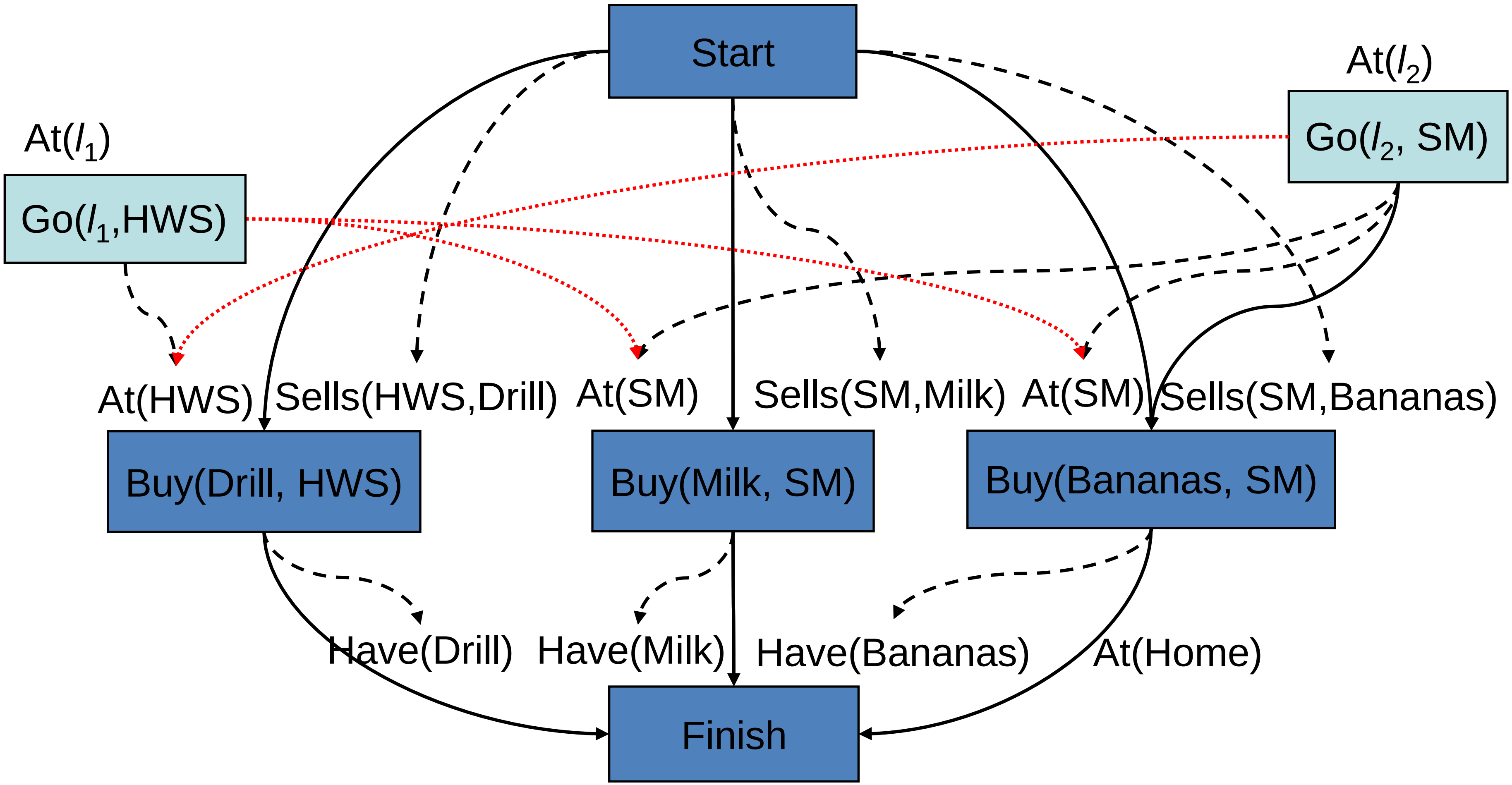
Example (continued)

– The only possible ways to establish the Sells preconditions



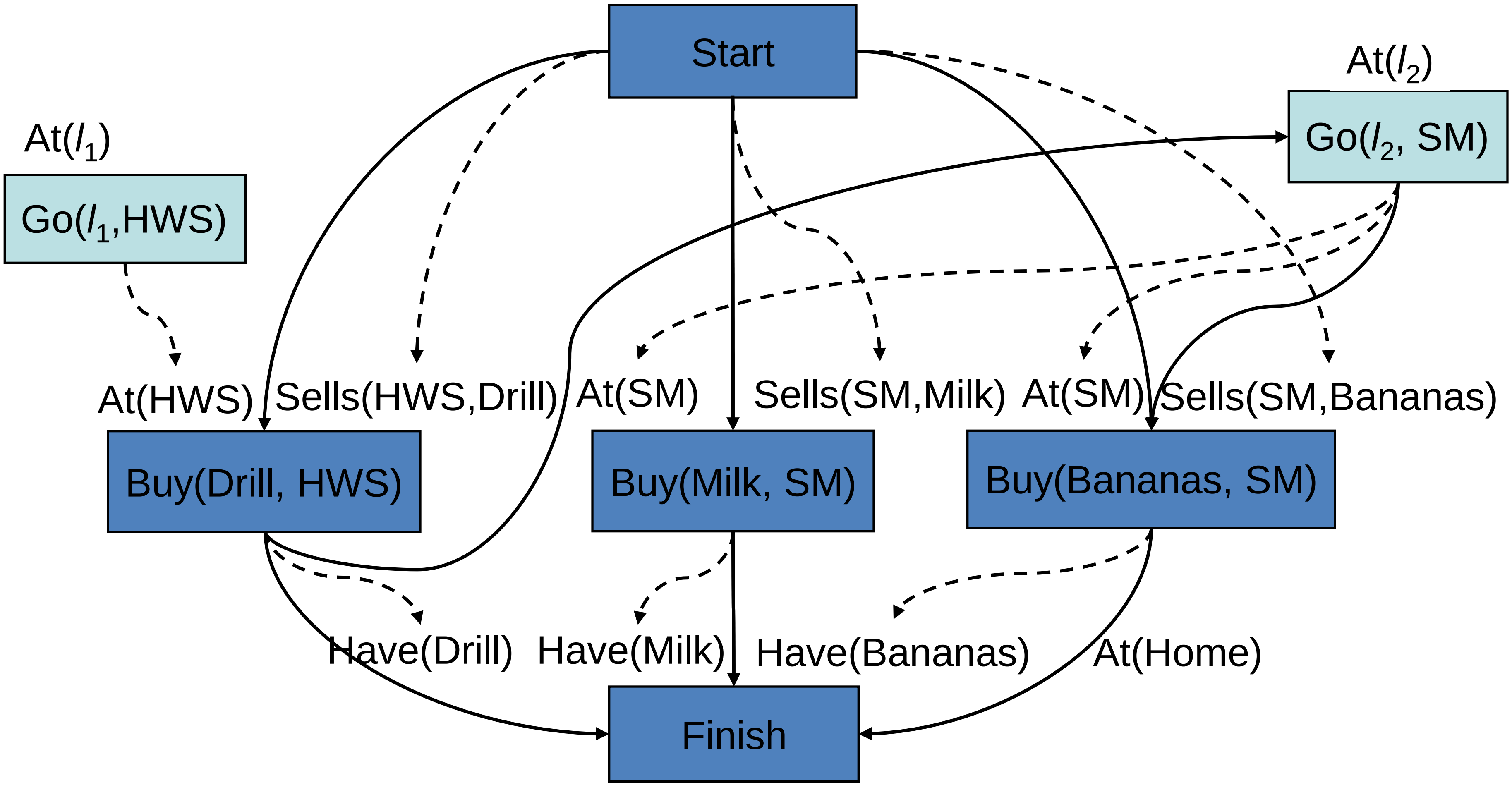
Example (continued)

- The only ways to establish At(HWS) and At(SM)
  - » Note the threats



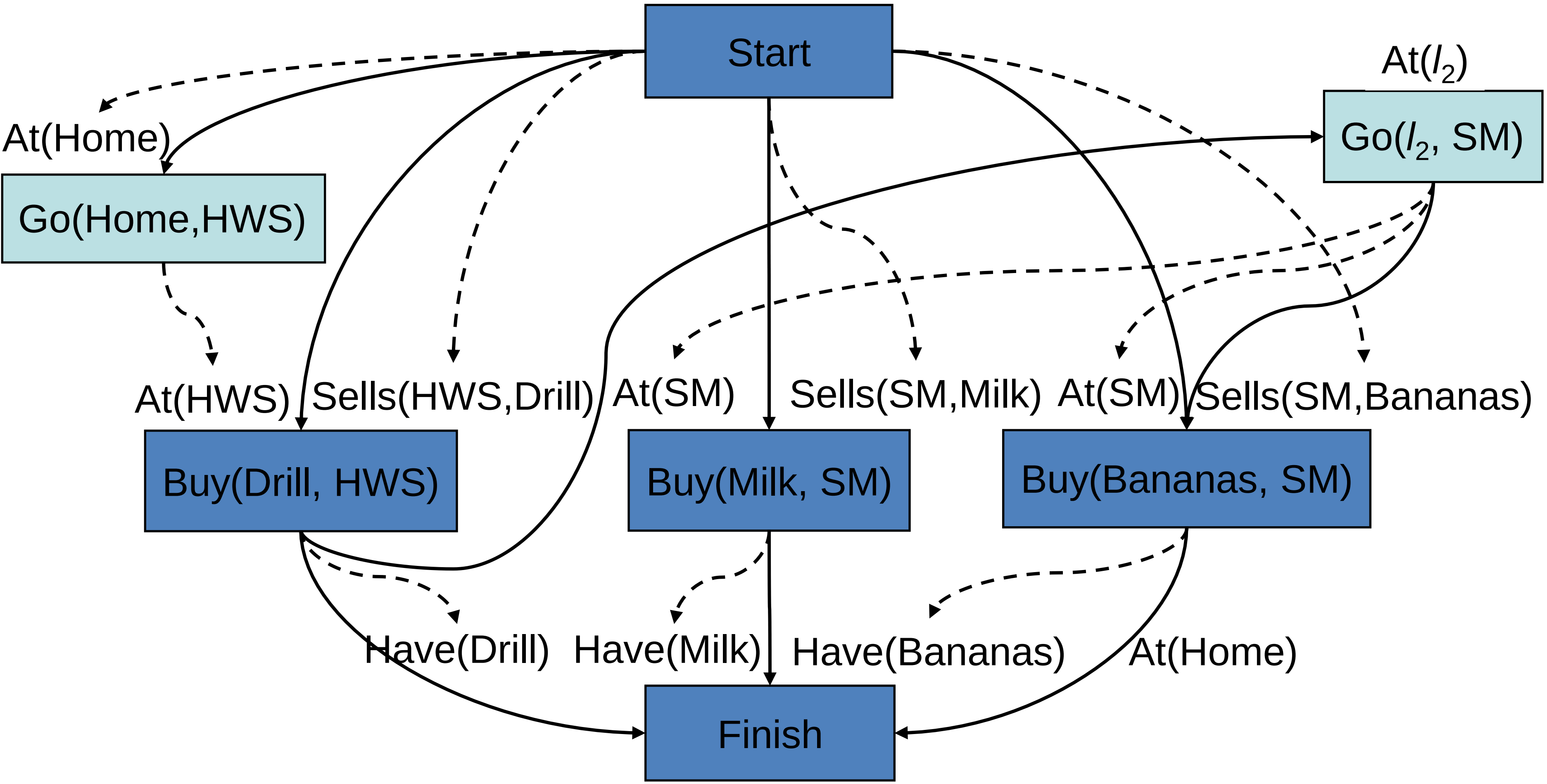
Example (continued)

- To resolve the threat to At(s_1), make Buy(Drill) precede Go(SM)
  - » This resolves all three threats



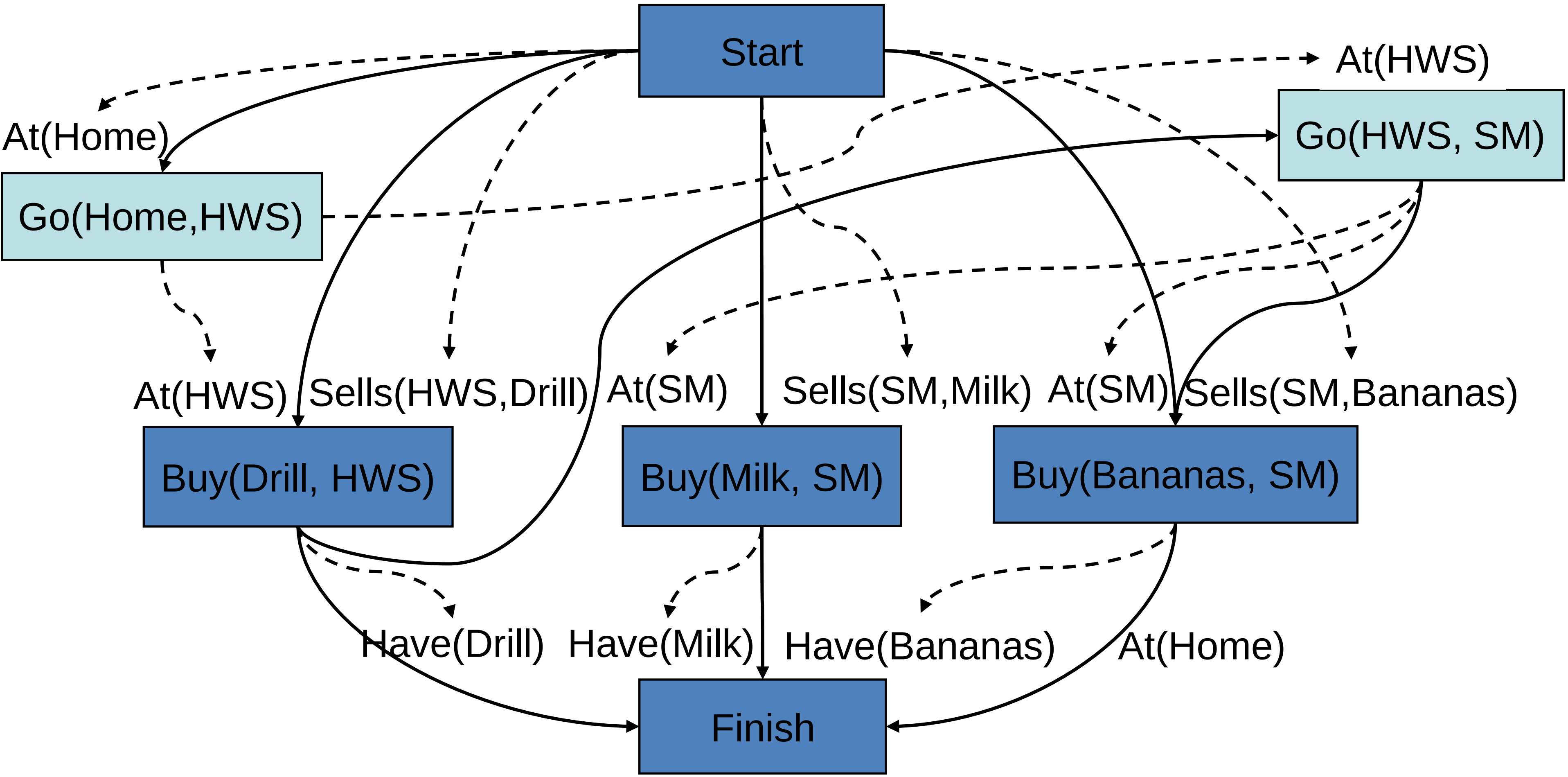
Example (continued)

– Establish At(l_1) with l_1=Home



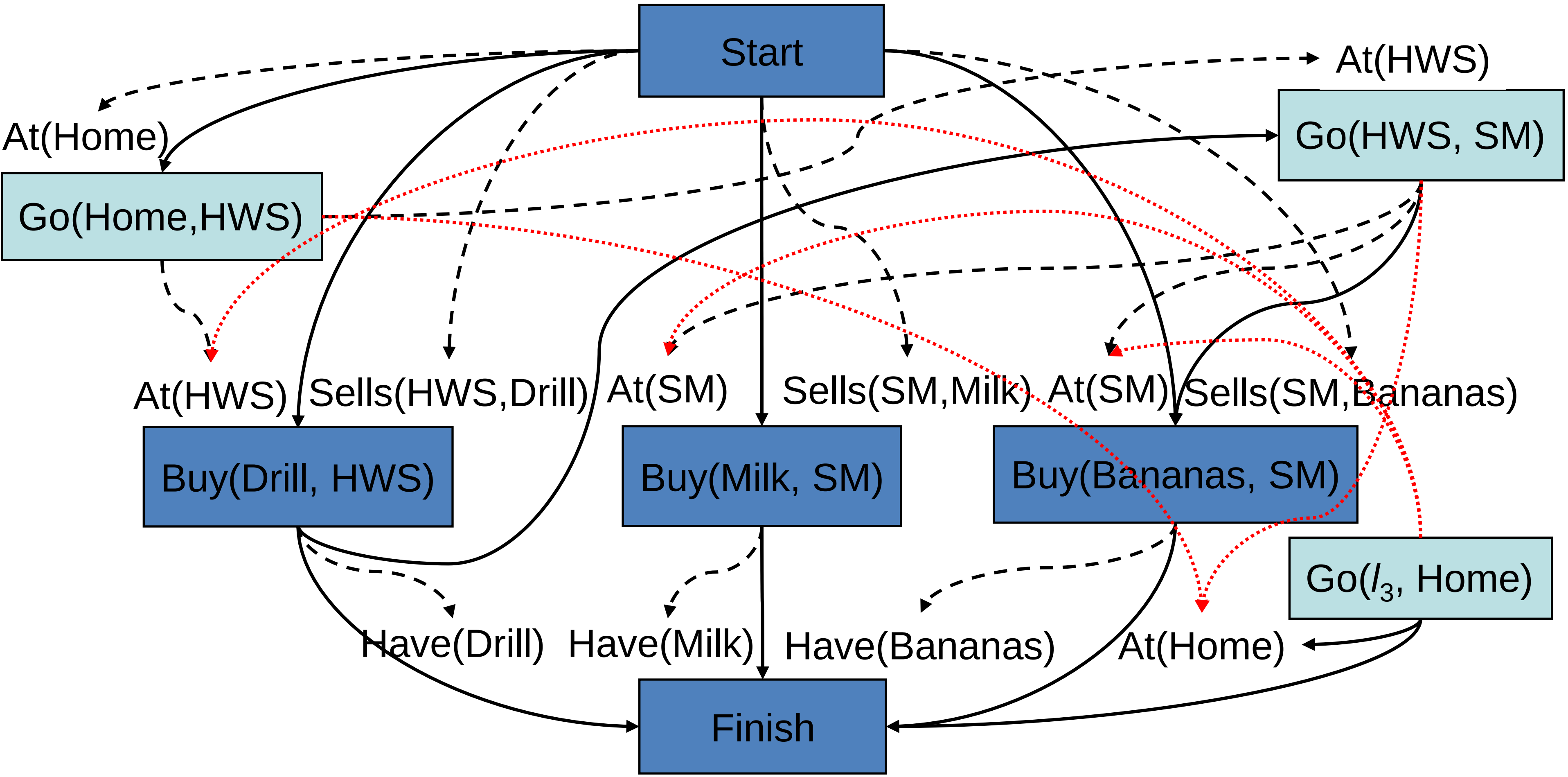
Example (continued)

– Establish At(l_2) with l_2= HWS



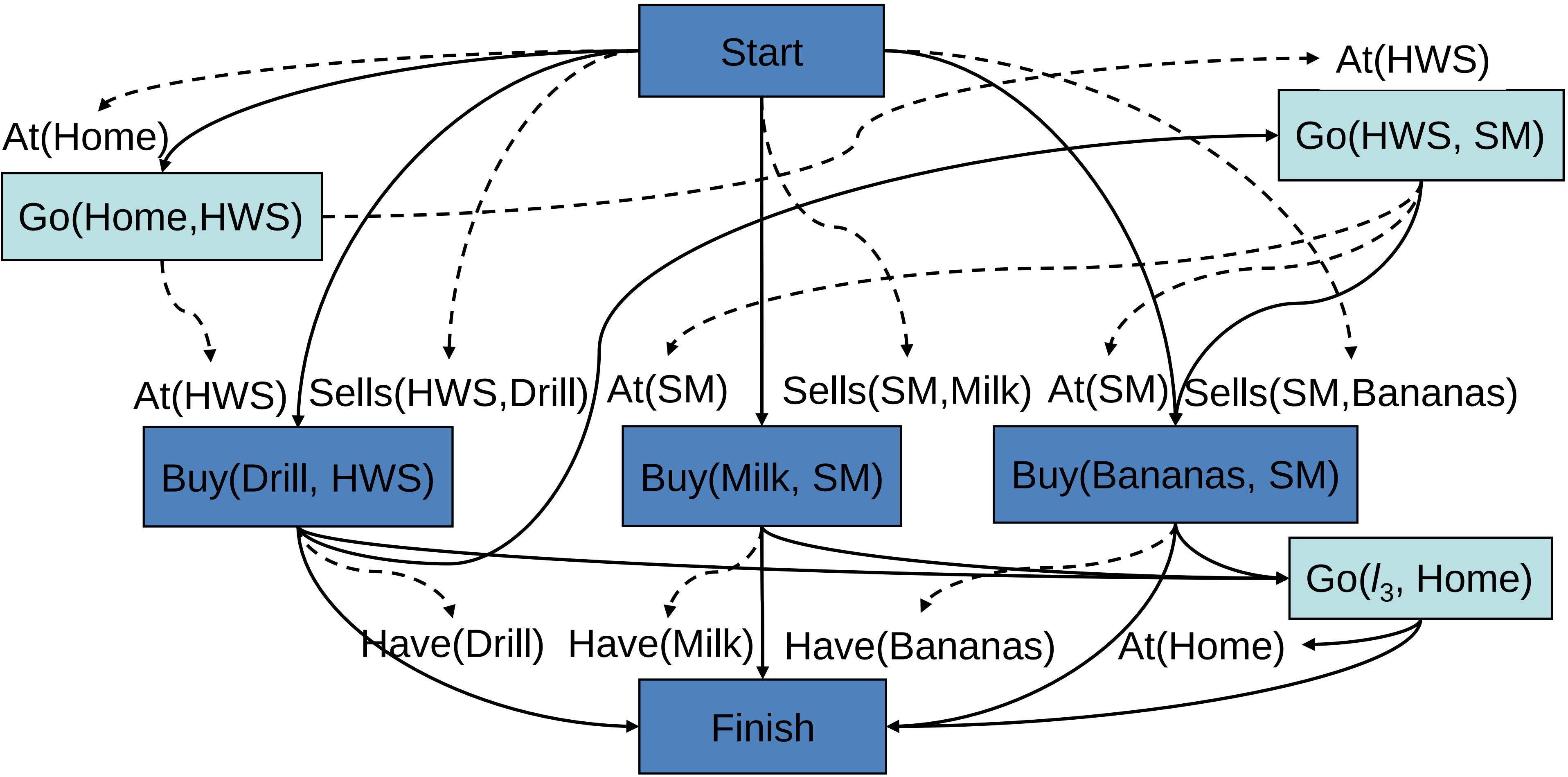
Example (continued)

- Establish At(Home) for Finish
  - » This creates a bunch of threats



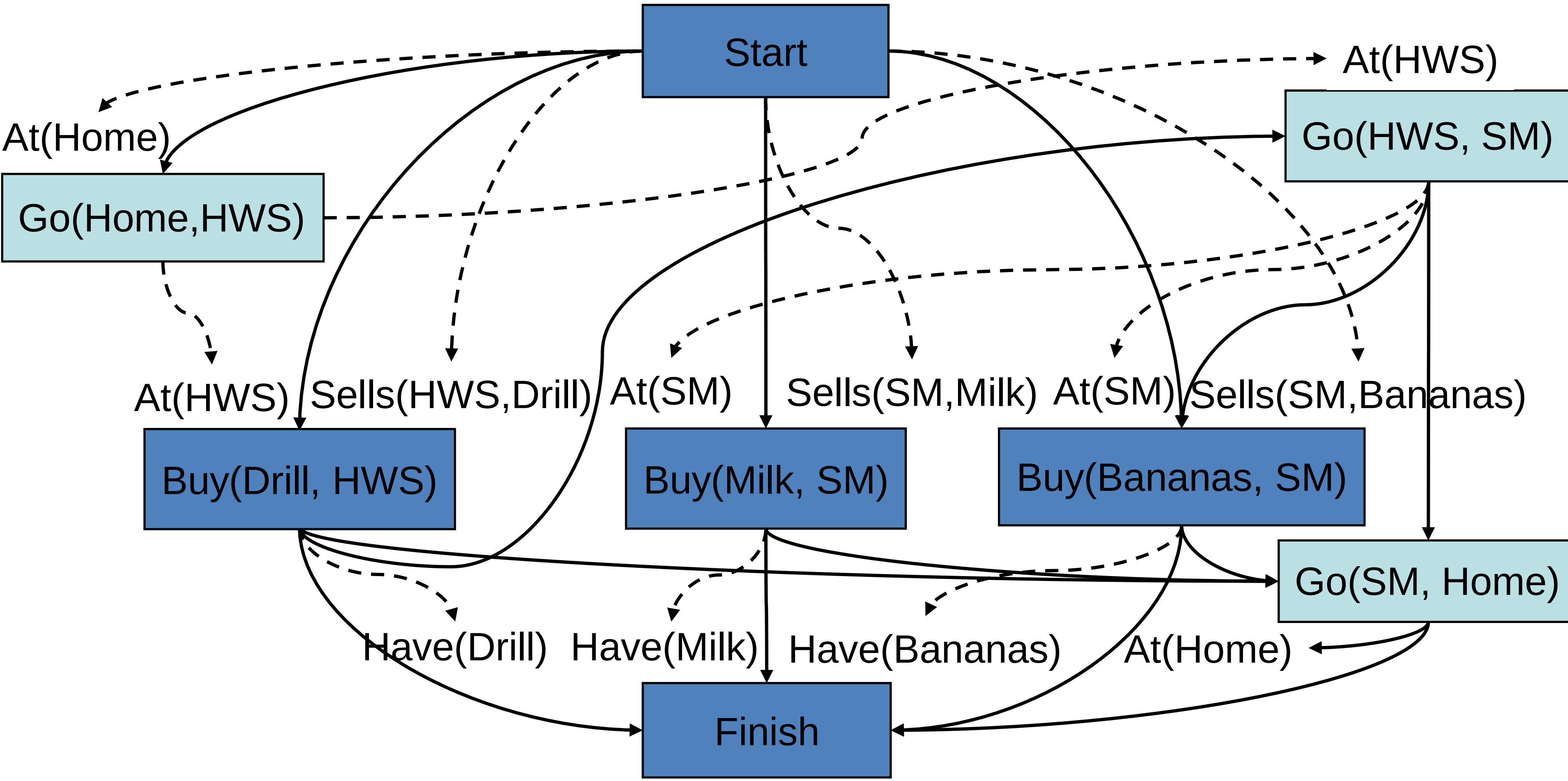
Example (continued)

– Constrain Go(l_3,Home) to remove threats to At(SM)
    » This also removes the other threats



Final Plan

– Establish At(l_3) with l_3=SM



Comments

- PSP doesn't commit to orderings and instantiations until necessary
- Problem: how to prune infinitely long paths?
  - » Loop detection is based on recognizing states we've seen before
  - » In a partially ordered plan, we don't know the states

- Can we prune if we see the same *action* more than once?

… — go(b,a) — go(a,b) – go(b,a) — at(a)

- No. Sometimes we might need the same action several times in different states of the world.

TOPLAN – known nonlinear planner

$\texttt{initialize:}\ \mathbf{\Pi} \leftarrow \{\{s_{\texttt{goal}}\}\}, \mathbf{S} \leftarrow \{s_{\texttt{goal}}\}$

$\texttt{toplan}(s_0, \Pi, S)\texttt{:}$

$\texttt{if}\ \exists s_n \in \mathbf{S}, \pi_n \in \mathbf{\Pi} : s_{\texttt{goal}} = s_n\ \texttt{then return}(\pi_n)$

$\texttt{if}\ \mathbf{S} = \{\}\ \texttt{return failure}$

$\texttt{else}\ \underline{\texttt{remove}}\ s_i\ \texttt{from}\ \mathbf{S}\ \text{and}\ \underline{\texttt{remove}}\ \pi_i\ \texttt{from}\ \mathbf{\Pi}$

$A \leftarrow \{\alpha|_{\texttt{eff}(\alpha)\in s_i}\}$

$S \leftarrow \{s|_{\forall\alpha\in A:\ \text{successor}(\alpha,s)=s_i}\}$

$\Pi \leftarrow \{\pi|_{\forall\alpha\in A:\ \pi=\alpha\cup\pi_i}\}$

$\texttt{return(}\texttt{toplan}(s_0, \underline{\text{append}}(\mathbf{\Pi}, \Pi)\texttt{,}\underline{\text{append}}(\mathbf{S}, S)\texttt{))}$

POPLAN – known nonlinear planner

initialize: $\Pi \leftarrow \{\texttt{actions}, \{s_0 \prec s_{\text{goal}}\}, \{\}, \{\text{pre}(s_{\text{goal}})\}\}$

poplan(Π):

if complete(Π) then return(Π)

if $\exists p$ of action $\beta \in$ open_goals(Π) and $\exists\alpha$ that achieves p

than append($\Pi, \{\{\alpha \underset{p}{\rightarrow} \beta\}, \{\alpha \prec \beta\}\}$) and remove($\beta$, open_goals($\Pi$))

else return(fail)

if there is a causal link $\alpha_1 \underset{x}{\rightarrow} \alpha_2$ threatened by α_3

then do either

Promotion: return poplan($\Pi \uplus \{\alpha_3 \prec \alpha_1\}$) or

Demotion: return poplan($\Pi \uplus \{\alpha_2 \prec \alpha_3\}$) or

else return poplan(Π)