CMSC 671
Fall 2010

Class #12/13 – Wednesday, October 13/
Monday, October 18

Some material adapted from slides by
Jean-Claude Latombe / Lise Getoor

Planning

Chapter 10

Some material adopted from notes
by Andreas Geyer-Schulz
and Chuck Dyer

Today’s class

- What is planning?
- Approaches to planning
  - GPS / STRIPS
  - Situation calculus formalism [revisited]
  - Partial-order planning
  - Graph-based planning
  - Satisfiability planning

Planning problem

- Find a **sequence of actions** that achieves a given **goal** when executed from a given **initial world state**. That is, given
  - a set of operator descriptions (defining the possible primitive actions by the agent),
  - an initial state description, and
  - a goal state description or predicate,

  compute a plan, which is
  - a sequence of operator instances, such that executing them in the initial state will change the world to a state satisfying the goal-state description.
- Goals are usually specified as a conjunction of goals to be achieved

Planning vs. problem solving

- Planning and problem solving methods can often solve the same sorts of problems
- Planning is more powerful because of the representations and methods used
- States, goals, and actions are decomposed into sets of sentences (usually in first-order logic)
- Search often proceeds through ***plan space*** rather than ***state space*** (though there are also state-space planners)
- Subgoals can be planned independently, reducing the complexity of the planning problem

Typical assumptions

- **Atomic time**: Each action is indivisible
- **No concurrent actions** are allowed (though actions do not need to be ordered with respect to each other in the plan)
- **Deterministic actions**: The result of actions are completely determined—there is no uncertainty in their effects
- Agent is the **sole cause of change** in the world
- Agent is **omniscient**: Has complete knowledge of the state of the world
- **Closed world assumption**: everything known to be true in the world is included in the state description. Anything not listed is false.

Blocks world

The **blocks world** is a micro-world that consists of a table, a set of blocks and a robot hand.

Some domain constraints:

- Only one block can be on another block
- Any number of blocks can be on the table
- The hand can only hold one block

Typical representation:

ontable(a)

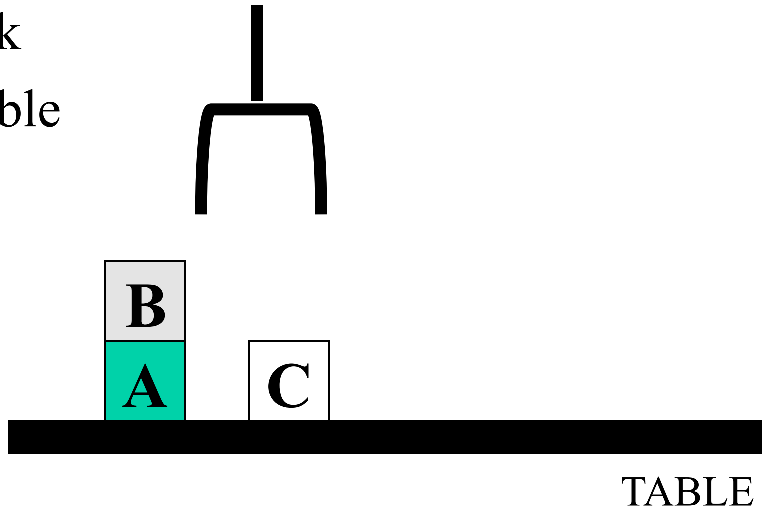
ontable(c)

on(b,a)

handempty

clear(b)

clear(c)



Major approaches

- GPS / STRIPS
- Situation calculus
- Partial-order planning
- Planning with constraints (SATplan, Graphplan)

- Hierarchical decomposition (HTN planning)
- Reactive planning

General Problem Solver

- The General Problem Solver (GPS) system was an early planner (Newell, Shaw, and Simon)
- GPS generated actions that reduced the difference between some state and a goal state
- GPS used Means-Ends Analysis
  - Compare what is given or known with what is desired and select a reasonable thing to do next
  - Use a table of differences to identify procedures to reduce types of differences
- GPS was a state space planner: it operated in the domain of state space problems specified by an initial state, some goal states, and a set of operations

Situation calculus planning

- Intuition: Represent the planning problem using first-order logic
  - Situation calculus lets us reason about changes in the world
  - Use theorem proving to “prove” that a particular sequence of actions, when applied to the situation characterizing the world state, will lead to a desired result

Situation calculus

- **Initial state**: a logical sentence about (situation) S_0

  $At(Home, S_0) \wedge \neg Have(Milk, S_0) \wedge \neg Have(Bananas, S_0) \wedge \neg Have(Drill, S_0)$

- **Goal state**:

  $(\exists s)\ At(Home,s) \wedge Have(Milk,s) \wedge Have(Bananas,s) \wedge Have(Drill,s)$

- **Operators** are descriptions of how the world changes as a result of the agent's actions:

  $\forall(a,s)\ Have(Milk,Result(a,s)) \Leftrightarrow$

  $((a=Buy(Milk) \wedge At(Grocery,s)) \vee (Have(Milk, s) \wedge a \neq Drop(Milk)))$

- Result(a,s) names the situation resulting from executing action a in situation s.
- Action sequences are also useful: Result'(l,s) is the result of executing the list of actions (l) starting in s:

  $(\forall s)\ Result'([],s) = s$

  $(\forall a,p,s)\ Result'([a|p]s) = Result'(p,Result(a,s))$

Situation calculus II

- A solution is a plan that when applied to the initial state yields a situation satisfying the goal query:

  $At(Home, Result'(p,S_0))$

  $\wedge$ $Have(Milk, Result'(p,S_0))$

  $\wedge$ $Have(Bananas, Result'(p,S_0))$

  $\wedge$ $Have(Drill, Result'(p,S_0))$

- Thus we would expect a plan (i.e., variable assignment through unification) such as:

  p = [Go(Grocery), Buy(Milk), Buy(Bananas), Go(HardwareStore), Buy(Drill), Go(Home)]

Situation calculus: Blocks world

- Here's an example of a situation calculus rule for the blocks world:
  - Clear (X, Result(A,S)) ↔
    [Clear (X, S) ∧
      (¬(A=Stack(Y,X) ∨ A=Pickup(X))
      ∨ (A=Stack(Y,X) ∧ ¬(holding(Y,S))
      ∨ (A=Pickup(X) ∧ ¬(handempty(S) ∧ ontable(X,S) ∧ clear(X,S))))]
    ∨ [A=Stack(X,Y) ∧ holding(X,S) ∧ clear(Y,S)]
    ∨ [A=Unstack(Y,X) ∧ on(Y,X,S) ∧ clear(Y,S) ∧ handempty(S)]
    ∨ [A=Putdown(X) ∧ holding(X,S)]
- English translation: A block is clear if (a) in the previous state it was clear and we didn't pick it up or stack something on it successfully, or (b) we stacked it on something else successfully, or (c) something was on it that we unstacked successfully, or (d) we were holding it and we put it down.
- Whew!!! There's gotta be a better way!

Situation calculus planning: Analysis

- This is fine in theory, but remember that problem solving (search) is exponential in the worst case
- Also, resolution theorem proving only finds *a* proof (plan), not necessarily a good plan
- So we restrict the language and use a special-purpose algorithm (a planner) rather than general theorem prover

Basic representations for planning

- Classic approach first used in the STRIPS planner circa 1970
- States represented as a conjunction of ground literals
  - at(Home) $\wedge$ $\neg$have(Milk) $\wedge$ $\neg$have(bananas) ...
- Goals are conjunctions of literals, but may have variables which are assumed to be existentially quantified
  - at(?x) $\wedge$ have(Milk) $\wedge$ have(bananas) ...
- Do not need to fully specify state
  - Non-specified either don't-care or assumed false
  - Represent many cases in small storage
  - Often only represent changes in state rather than entire situation
- Unlike theorem prover, not seeking whether the goal is true, but is there a sequence of actions to attain it

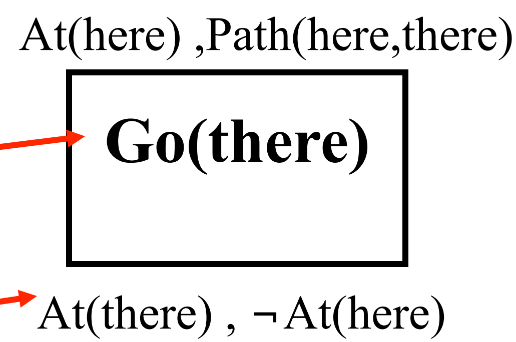
Operator/action representation

- Operators contain three components:
  - **Action description**
  - **Precondition** - conjunction of positive literals
  - **Effect** - conjunction of positive or negative literals which describe how situation changes when operator is applied
- Example:

  Op[Action: Go(there),

  Precond: At(here) ∧ Path(here,there),

  Effect: At(there) ∧ ¬At(here)]



- All variables are universally quantified
- Situation variables are implicit
  - Preconditions must be true in the state immediately before an operator is applied; effects are true immediately after

Blocks world operators

- Here are the classic basic operations for the blocks world:
  - stack(X,Y): put block X on block Y
  - unstack(X,Y): remove block X from block Y
  - pickup(X): pickup block X
  - putdown(X): put block X on the table
- Each action will be represented by:
  - a list of preconditions
  - a list of new facts to be added (add-effects)
  - a list of facts to be removed (delete-effects)
  - optionally, a set of (simple) variable constraints
- For example:

preconditions(stack(X,Y), [holding(X), clear(Y)])
deletes(stack(X,Y), [holding(X), clear(Y)]).
adds(stack(X,Y), [handempty, on(X,Y), clear(X)])
constraints(stack(X,Y), [X≠Y, Y≠table, X≠table])

Blocks world operators II

operator(stack(X,Y),
Precond [holding(X), clear(Y)],
Add [handempty, on(X,Y), clear(X)],
Delete [holding(X), clear(Y)],
Constr [X≠Y, Y≠table, X≠table]).

operator(unstack(X,Y),
[on(X,Y), clear(X), handempty],
[holding(X), clear(Y)],
[handempty, clear(X), on(X,Y)],
[X≠Y, Y≠table, X≠table]).

operator(pickup(X),
[ontable(X), clear(X), handempty],
[holding(X)],
[ontable(X), clear(X), handempty],
[X≠table]).

operator(putdown(X),
[holding(X)],
[ontable(X), handempty, clear(X)],
[holding(X)],
[X≠table]).

STRIPS planning

- STRIPS maintains two additional data structures:
  - **State List** - all currently true predicates.
  - **Goal Stack** - a push-down stack of goals to be solved, with current goal on top of stack.
- If current goal is not satisfied by present state, examine add lists of operators, and push operator and preconditions list on stack. (Subgoals)
- When a current goal is satisfied, POP it from stack.
- When an operator is on top of the stack, record the application of that operator in the plan sequence and use the operator's add and delete lists to update the current state.

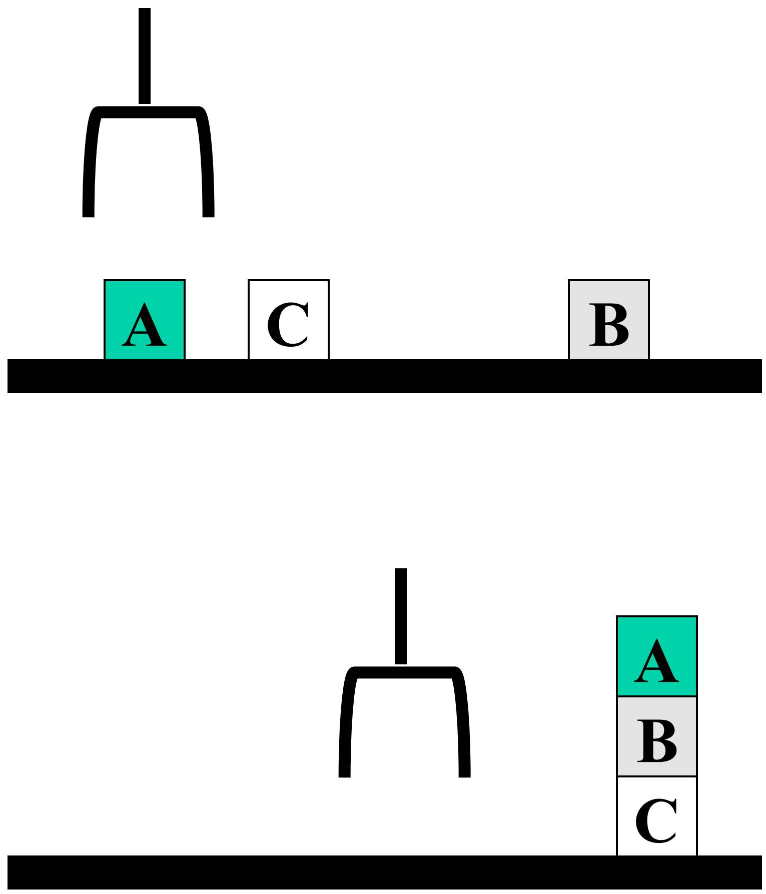
Typical BW planning problem

Initial state:

- clear(a)
- clear(b)
- clear(c)
- ontable(a)
- ontable(b)
- ontable(c)
- handempty

Goal:

- on(b,c)
- on(a,b)
- ontable(c)



A plan:

- pickup(b)
- stack(b,c)
- pickup(a)
- stack(a,b)

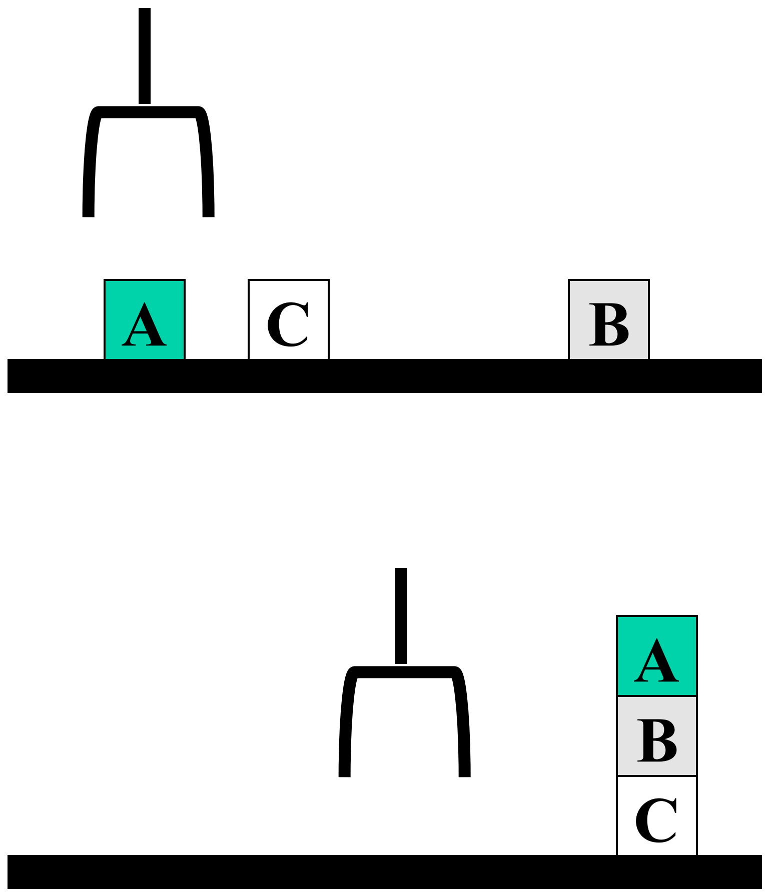
Another BW planning problem

Initial state:

- clear(a)
- clear(b)
- clear(c)
- ontable(a)
- ontable(b)
- ontable(c)
- handempty

Goal:

- on(a,b)
- on(b,c)
- ontable(c)

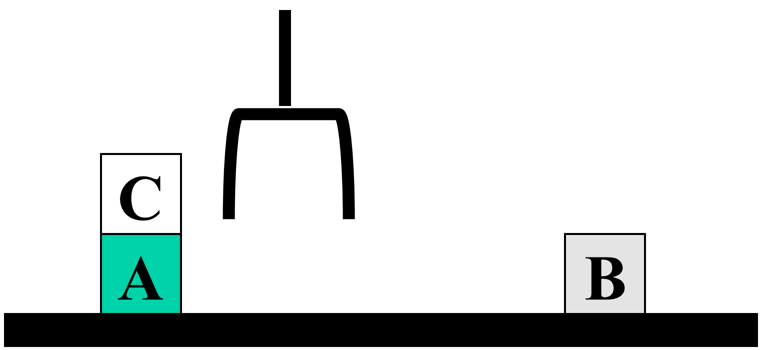


A plan:

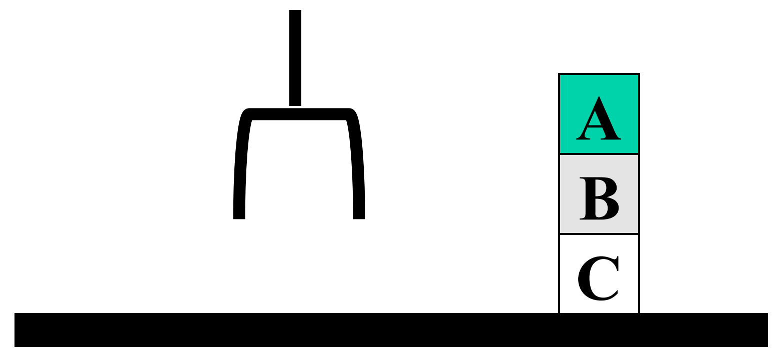
- pickup(a)
- stack(a,b)
- unstack(a,b)
- putdown(a)
- pickup(b)
- stack(b,c)
- pickup(a)
- stack(a,b)

Goal interaction

- Simple planning algorithms assume that the goals to be achieved are independent
  - Each can be solved separately and then the solutions concatenated
- This planning problem, called the "Sussman Anomaly," is the classic example of the goal interaction problem:
  - Solving on(A,B) first (by doing unstack(C,A), stack(A,B) will be undone when solving the second goal on(B,C) (by doing unstack(A,B), stack(B,C)).
  - Solving on(B,C) first will be undone when solving on(A,B)
- Classic STRIPS could not handle this, although minor modifications can get it to do simple cases



Initial state

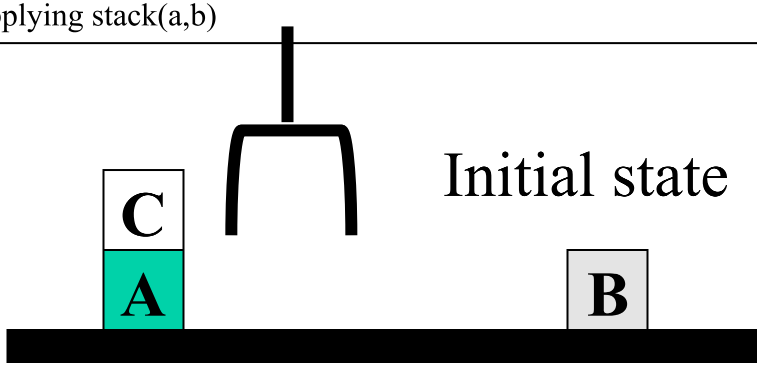


Goal state

Sussman Anomaly

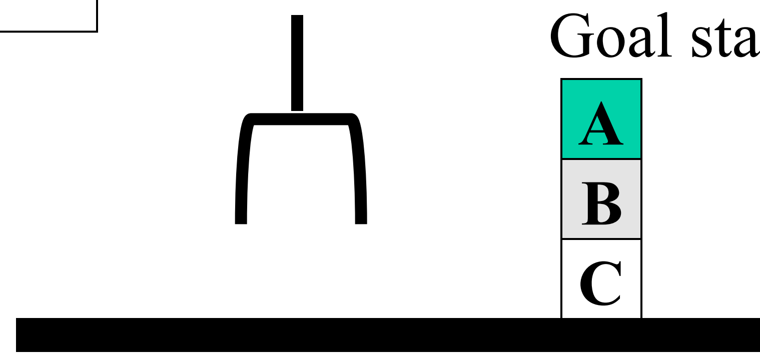
```
Achieve on(a,b) via stack(a,b) with preconds: [holding(a),clear(b)]
|Achieve holding(a) via pickup(a) with preconds: [ontable(a),clear(a),handempty]
||Achieve clear(a) via unstack(_1584,a) with preconds:
[on(_1584,a),clear(_1584),handempty]
||Applying unstack(c,a)
||Achieve handempty via putdown(_2691) with preconds: [holding(_2691)]
||Applying putdown(c)
|Applying pickup(a)
Applying stack(a,b)
Achieve on(b,c) via stack(b,c) with preconds: [holding(b),clear(c)]
|Achieve holding(b) via pickup(b) with preconds: [ontable(b),clear(b),handempty]
||Achieve clear(b) via unstack(_5625,b) with preconds:
[on(_5625,b),clear(_5625),handempty]
||Applying unstack(a,b)
||Achieve handempty via putdown(_6648) with preconds: [holding(_6648)]
||Applying putdown(a)
|Applying pickup(b)
Applying stack(b,c)
Achieve on(a,b) via stack(a,b) with preconds: [holding(a),clear(b)]
|Achieve holding(a) via pickup(a) with preconds: [ontable(a),clear(a),handempty]
|Applying pickup(a)
Applying stack(a,b)
```

```
From
[clear(b),clear(c),ontable(a),ontable(b),on(
c,a),handempty]
  To [on(a,b),on(b,c),ontable(c)]
  Do:
      unstack(c,a)
      putdown(c)
      pickup(a)
      stack(a,b)
      unstack(a,b)
      putdown(a)
      pickup(b)
      stack(b,c)
      pickup(a)
      stack(a,b)
```



Initial state

Goal state



State-space planning

- We initially have a space of situations (where you are, what you have, etc.)
- The plan is a solution found by "searching" through the situations to get to the goal
- A **progression planner** searches forward from initial state to goal state
- A **regression planner** searches backward from the goal
  - This works if operators have enough information to go both ways
  - Ideally this leads to reduced branching: the planner is only considering things that are relevant to the goal

Planning heuristics

- Just as with search, we need an **admissible** heuristic that we can apply to planning states
  - Estimate of the distance (number of actions) to the goal
- Planning typically uses **relaxation** to create heuristics
  - Ignore all or selected preconditions
  - Ignore delete lists (movement towards goal is never undone)
  - Use state abstraction (group together "similar" states and treat them as though they are identical) – e.g., ignore fluents
  - Assume subgoal independence (use max cost; or if subgoals actually are independent, can sum the costs)
  - Use pattern databases to store exact solution costs of recurring subproblems

Plan-space planning

- An alternative is to **search through the space of *plans***, rather than situations.
- Start from a **partial plan** which is expanded and refined until a complete plan that solves the problem is generated.
- **Refinement operators** add constraints to the partial plan and modification operators for other changes.
- We can still use STRIPS-style operators:

  Op(ACTION: RightShoe, PRECOND: RightSockOn, EFFECT: RightShoeOn)

  Op(ACTION: RightSock, EFFECT: RightSockOn)

  Op(ACTION: LeftShoe, PRECOND: LeftSockOn, EFFECT: LeftShoeOn)

  Op(ACTION: LeftSock, EFFECT: leftSockOn)

could result in a partial plan of

  [RightShoe, LeftShoe]

Partial-order planning

- A **linear planner** builds a plan as a **totally ordered sequence** of plan steps
- A **non-linear planner (aka partial-order planner)** builds up a plan as a set of steps with some temporal constraints
  - constraints of the form S1<S2 if step S1 must comes before S2.
- One **refines** a partially ordered plan (POP) by either:
  - **adding a new plan step**, or
  - **adding a new constraint** to the steps already in the plan.
- A POP can be **linearized** (converted to a totally ordered plan) by topological sorting

Least commitment

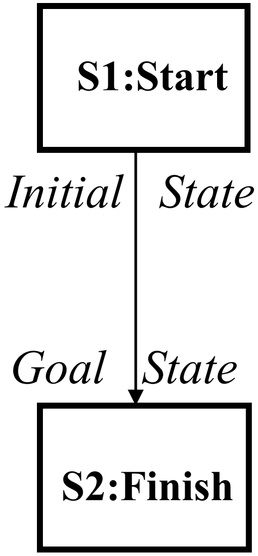
- Non-linear planners embody the principle of **least commitment**
  - only choose actions, orderings, and variable bindings that are absolutely necessary, leaving other decisions till later
  - avoids early commitment to decisions that don't really matter
- A linear planner always chooses to add a plan step in a particular place in the sequence
- A non-linear planner chooses to add a step and possibly some temporal constraints

Non-linear plan

- A non-linear plan consists of
  - (1) A set of **steps** $\{S_1, S_2, S_3, S_4 \ldots\}$
    - Each step has an operator description, preconditions and post-conditions
  - (2) A set of **causal links** $\{ \ldots (S_i,C,S_j) \ldots\}$
    - Meaning a purpose of step S_i is to achieve precondition C of step S_j
  - (3) A set of **ordering constraints** $\{ \ldots S_i<S_j \ldots \}$
    - if step S_i must come before step S_j
- A non-linear plan is **complete** iff
  - Every step mentioned in (2) and (3) is in (1)
  - If S_j has prerequisite C, then there exists a causal link in (2) of the form (S_i,C,S_j) for some S_i
  - If (S_i,C,S_j) is in (2) and step S_k is in (1), and S_k threatens (S_i,C,S_j) (makes C false), then (3) contains either $S_k<S_i$ or $S_j<S_k$

The initial plan

Every plan starts the same way



Trivial example

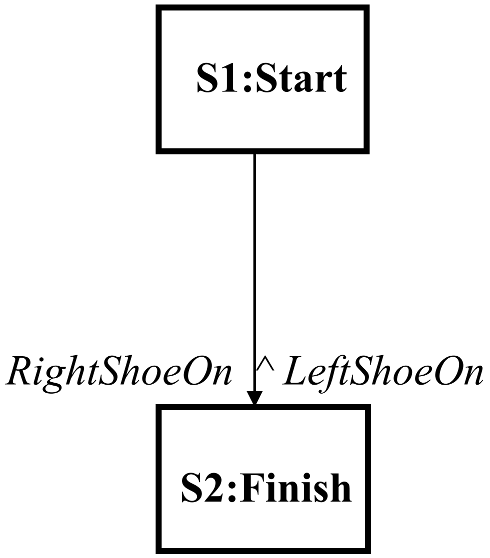
Operators:

Op(ACTION: RightShoe, PRECOND: RightSockOn, EFFECT: RightShoeOn)

Op(ACTION: RightSock, EFFECT: RightSockOn)

Op(ACTION: LeftShoe, PRECOND: LeftSockOn, EFFECT: LeftShoeOn)

Op(ACTION: LeftSock, EFFECT: leftSockOn)



Steps: {S1:[Op(Action:Start)],

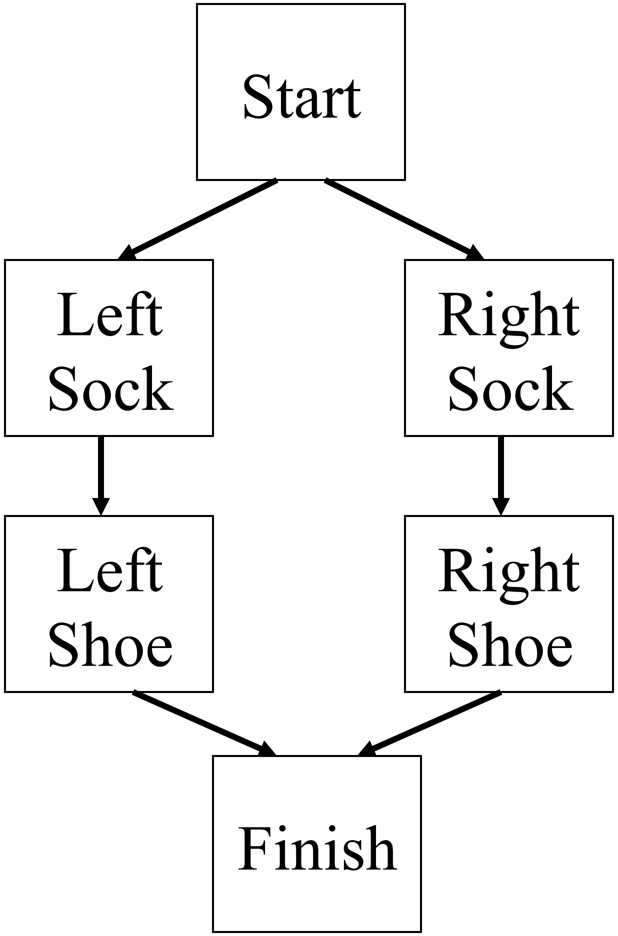
S2:[Op(Action:Finish,

Pre: RightShoeOn^LeftShoeOn)]}

Links: {}

Orderings: {S1<S2}

Solution



POP constraints and search heuristics

- Only add steps that achieve a currently unachieved precondition
- Use a least-commitment approach:
  - Don't order steps unless they need to be ordered
- Honor causal links $S_1 \xrightarrow{c} S_2$ that **protect** a condition *c*:
  - Never add an intervening step S_3 that violates *c*
  - If a parallel action **threatens** *c* (i.e., has the effect of negating or **clobbering** *c*), resolve that threat by adding ordering links:
    - Order S_3 before S_1 (**demotion**)
    - Order S_3 after S_2 (**promotion**)

function POP(*initial*, *goal*, *operators*) **returns** *plan*

plan ← MAKE-MINIMAL-PLAN(*initial*, *goal*)
loop do
  if SOLUTION?(*plan*) **then return** *plan*
  S_{need}, c ← SELECT-SUBGOAL(*plan*)
  CHOOSE-OPERATOR(*plan*, *operators*, S_{need}, c)
  RESOLVE-THREATS(*plan*)
end

---

function SELECT-SUBGOAL(*plan*) **returns** S_{need}, c

pick a plan step S_{need} from STEPS(*plan*)
  with a precondition c that has not been achieved
return S_{need}, c

---

procedure CHOOSE-OPERATOR(*plan*, *operators*, S_{need}, c)

choose a step S_{add} from *operators* or STEPS(*plan*) that has c as an effect
if there is no such step **then fail**
add the causal link $S_{add} \xrightarrow{c} S_{need}$ to LINKS(*plan*)
add the ordering constraint $S_{add} \prec S_{need}$ to ORDERINGS(*plan*)
if S_{add} is a newly added step from *operators* **then**
  add S_{add} to STEPS(*plan*)
  add $Start \prec S_{add} \prec Finish$ to ORDERINGS(*plan*)

---

procedure RESOLVE-THREATS(*plan*)

for each S_{threat} that threatens a link $S_i \xrightarrow{c} S_j$ in LINKS(*plan*) **do**
  choose either
    Promotion: Add $S_{threat} \prec S_i$ to ORDERINGS(*plan*)
    Demotion: Add $S_j \prec S_{threat}$ to ORDERINGS(*plan*)
  if not CONSISTENT(*plan*) **then fail**
end

Partial-order planning algorithm

- Create a START node with the initial state as its effects
- Create a GOAL node with the goal as its preconditions
- Create an ordering link from START to GOAL
- While there are unsatisfied preconditions:
  - Choose a precondition to satisfy
  - Choose an existing action or insert a new action whose effect satisfies the precondition
    - (If no such action, backtrack!)
  - Insert a causal link from the chosen action's effect to the precondition
  - Resolve any new threats
    - (If not possible, backtrack!)

Partial-order planning example

- Goal: Have milk, bananas, and a drill

  Have(Milk) $\wedge$ Have(Bananas) $\wedge$ Have(Drill)

- Operators:

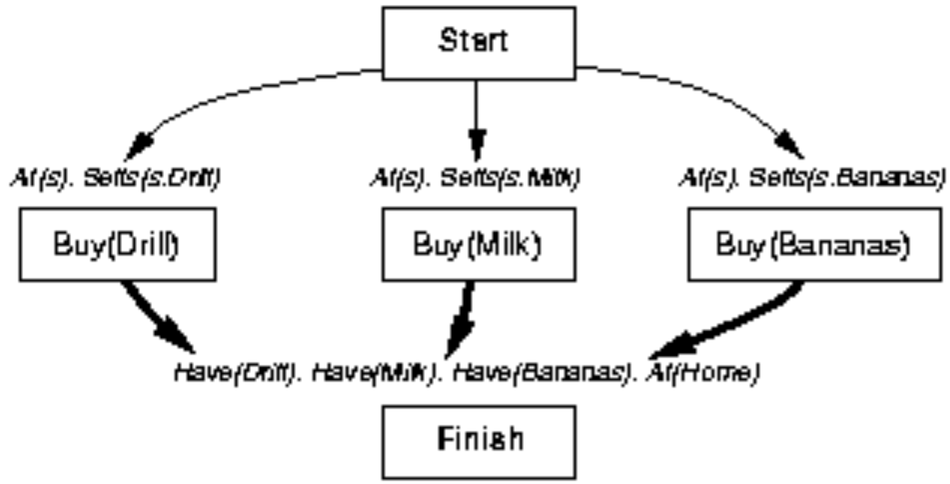
  Op(ACTION: Buy(Item), PRECOND: At(Store) $\wedge$ Sells(Store,Item),
  EFFECT: Have(Item))

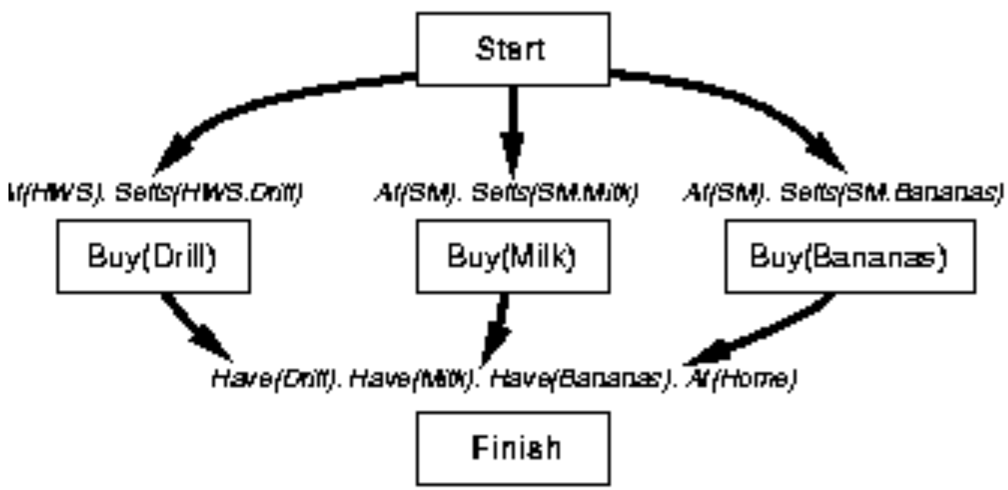
  Op(ACTION: Go(Dest), PRECOND: At(Source),
  EFFECT: At(Dest) $\wedge$ ~At(Source))

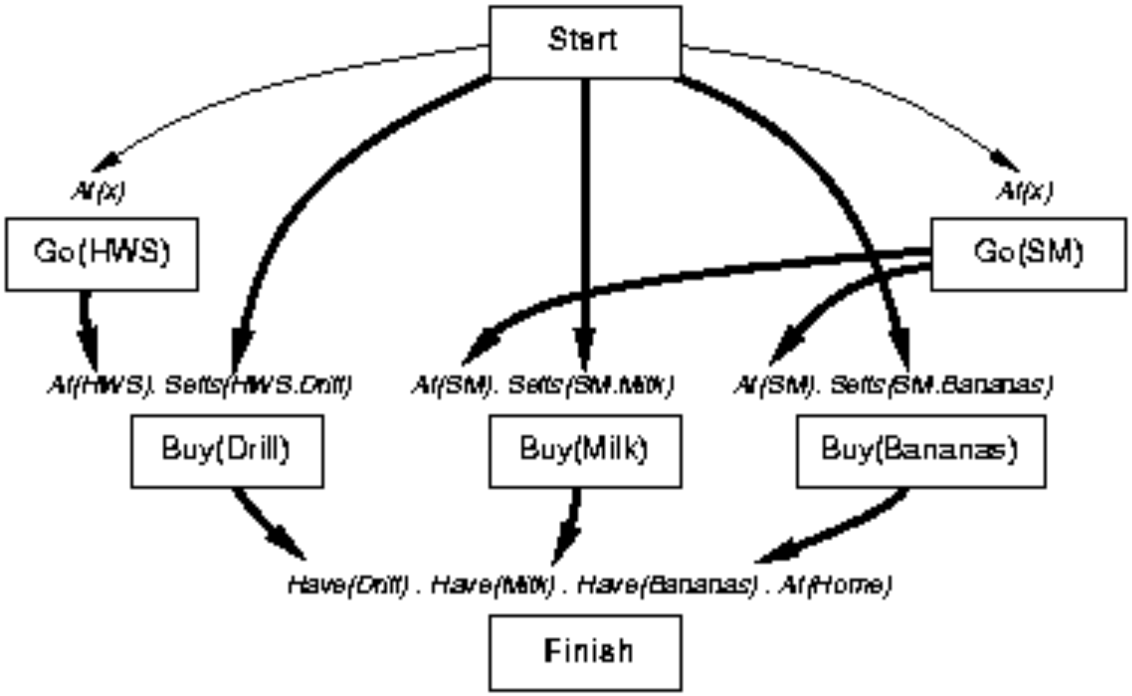
- Initial state:

  At(Home) $\wedge$ Sells(SM, Milk) $\wedge$ Sells(SM, Bananas) $\wedge$ Sells(HW, Drill)

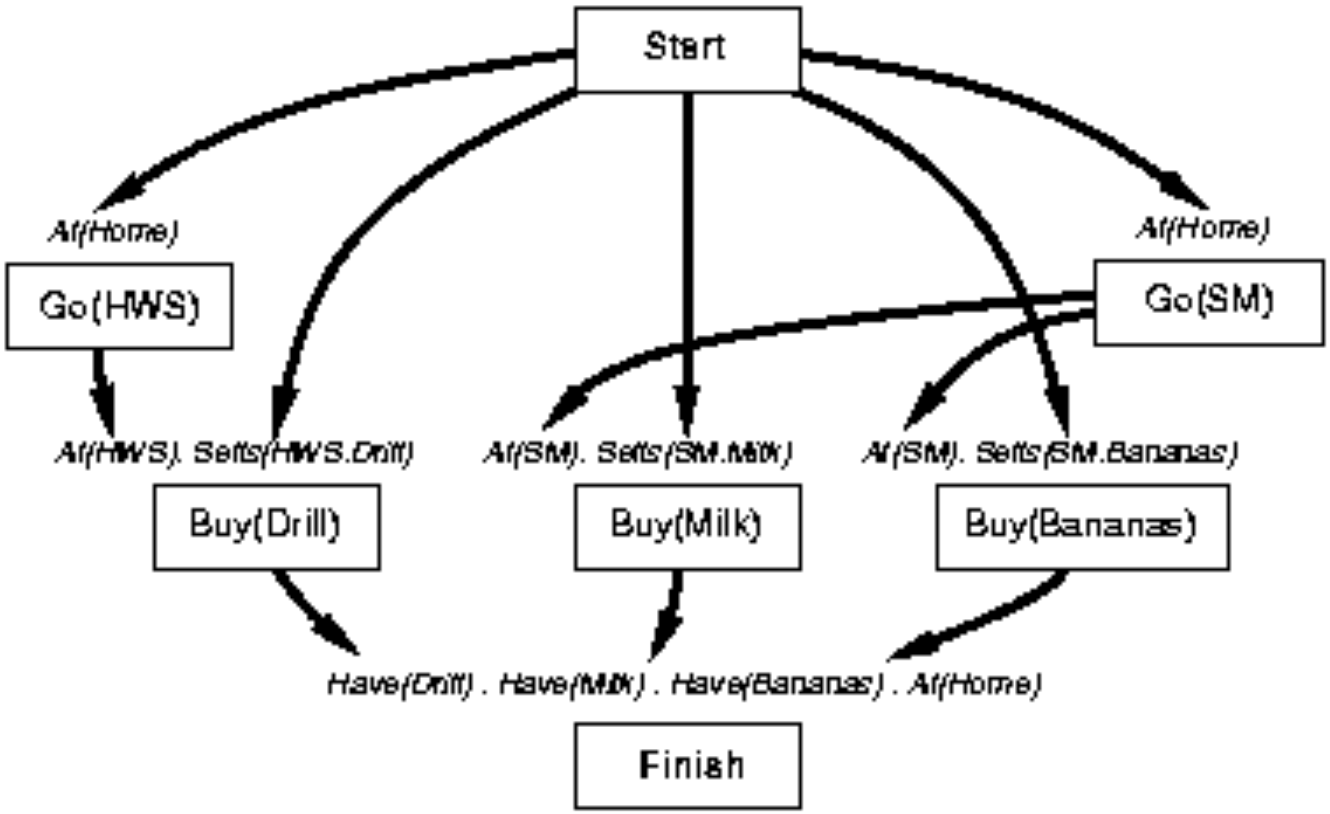

Start
At(Home) Sells(SM,Banana) Sells(SM,Milk) Sells(HWS,Drill)
Have(Drill) Have(Milk) Have(Banana) At(Home)
Finish


Start
At(s), Sells(s,Drill)
At(s), Sells(s,Milk)
At(s), Sells(s,Bananas)
Buy(Drill)
Buy(Milk)
Buy(Bananas)
Have(Drill), Have(Milk), Have(Bananas), At(Home)
Finish

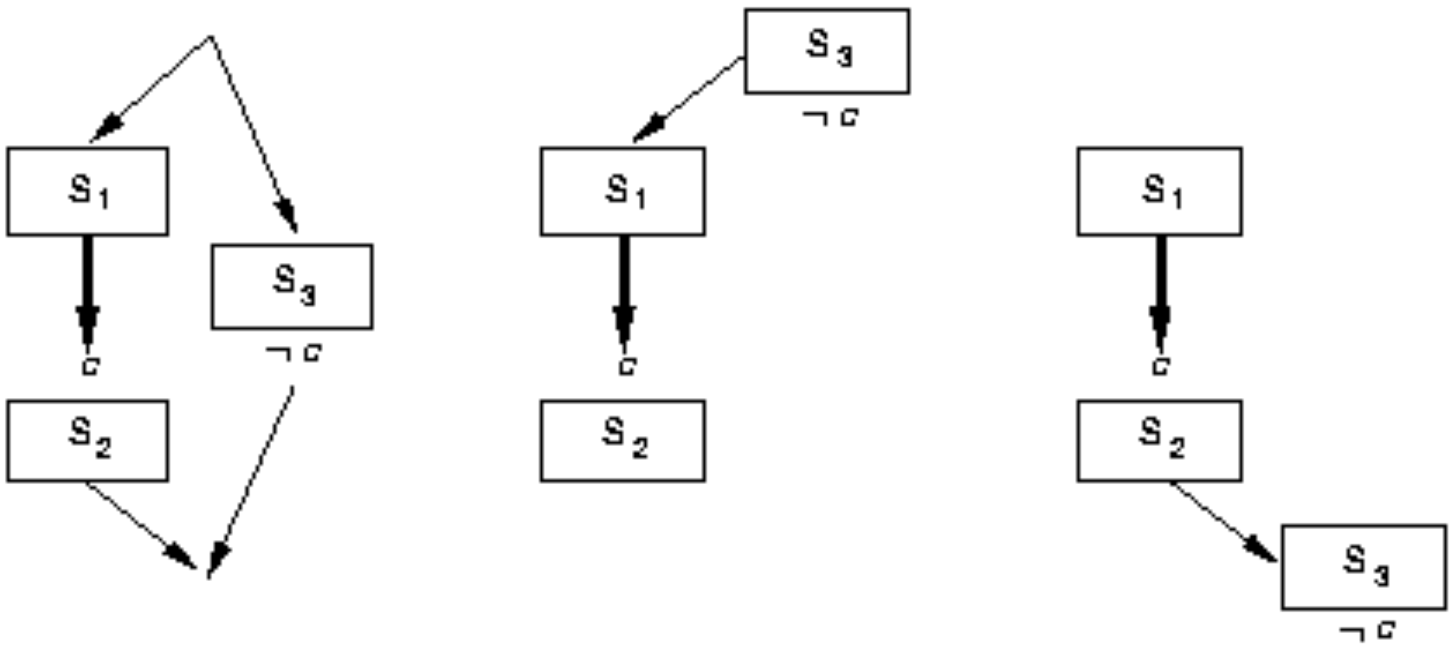

Start
t(HWS), Sells(HWS,Drill)
At(SM), Sells(SM,Milk)
At(SM), Sells(SM,Bananas)
Buy(Drill)
Buy(Milk)
Buy(Bananas)
Have(Drill), Have(Milk), Have(Bananas), At(Home)
Finish



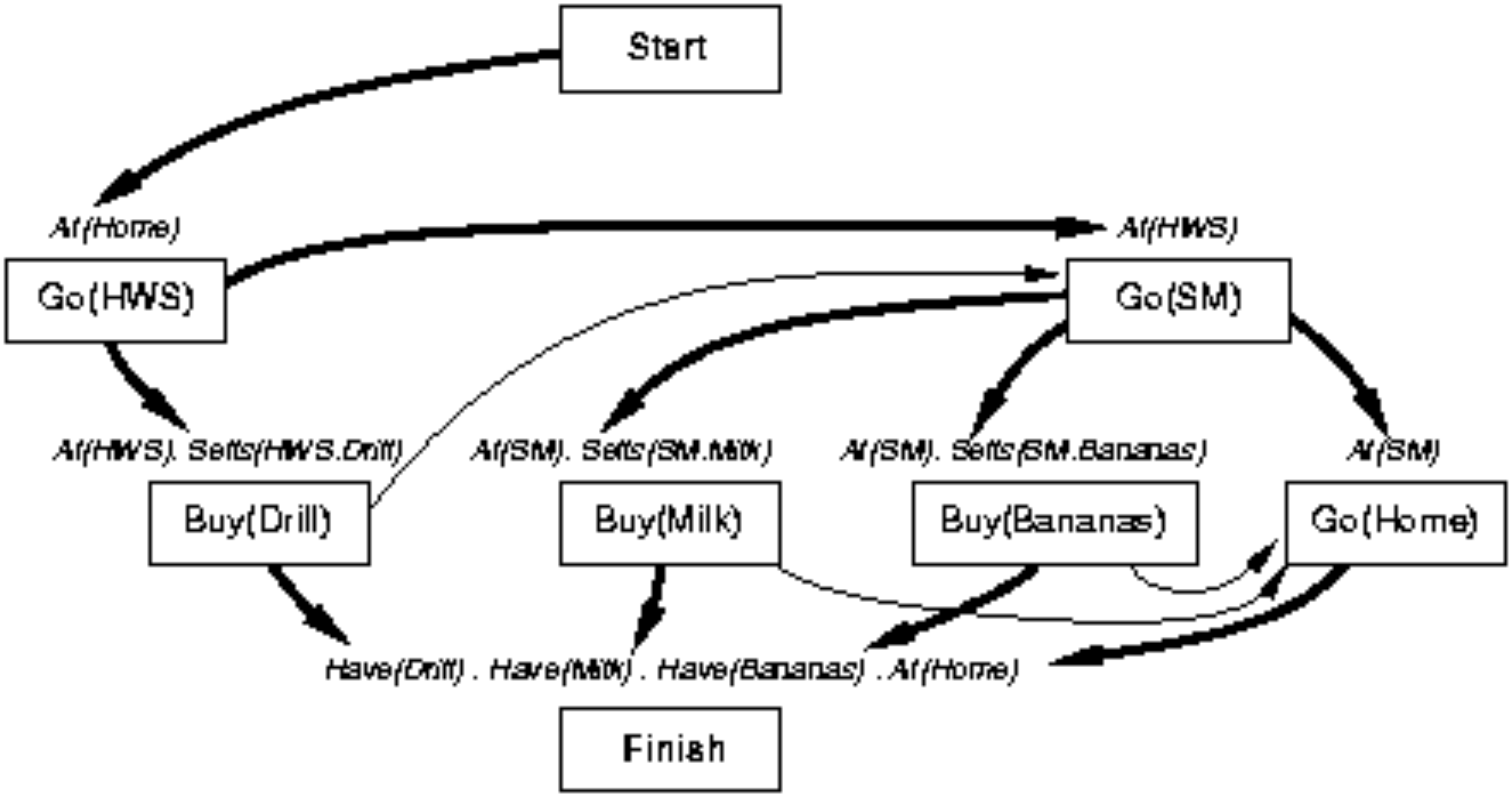
Start
At(x)
At(x)
Go(HWS)
Go(SM)
At(HWS). Sells(HWS.Drill)
At(SM). Sells(SM.Milk)
At(SM). Sells(SM.Bananas)
Buy(Drill)
Buy(Milk)
Buy(Bananas)
Have(Drill) . Have(Milk) . Have(Bananas) . At(Home)
Finish

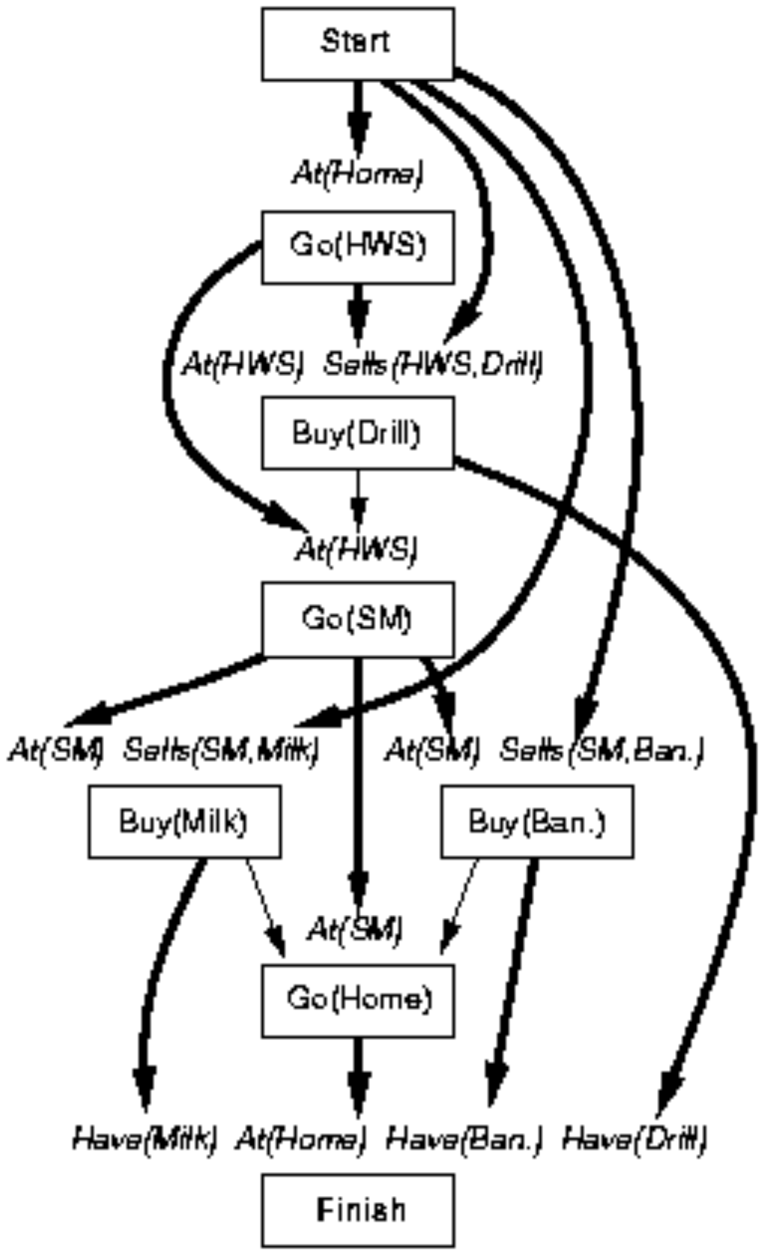

Start
At(Home)
Go(HWS)
At(Home)
Go(SM)
At(HWS), Sells(HWS,Drill)
At(SM), Sells(SM,Milk)
At(SM), Sells(SM,Bananas)
Buy(Drill)
Buy(Milk)
Buy(Bananas)
Have(Drill) , Have(Milk) , Have(Bananas) , At(Home)
Finish

Resolving threats



(a) Threat (b) Demotion (c) Promotion


Start
At(Home)
At(HWS)
Go(HWS)
Go(SM)
At(HWS), Sells(HWS,Drill)
At(SM), Sells(SM,Milk)
At(SM), Sells(SM,Bananas)
At(SM)
Buy(Drill)
Buy(Milk)
Buy(Bananas)
Go(Home)
Have(Drill) . Have(Milk) . Have(Bananas) . At(Home)
Finish



Start
At(Home)
Go(HWS)
At(HWS)
Sells(HWS,Drill)
Buy(Drill)
At(HWS)
Go(SM)
At(SM)
Sells(SM,Milk)
At(SM)
Sells(SM,Ban.)
Buy(Milk)
Buy(Ban.)
At(SM)
Go(Home)
Have(Milk)
At(Home)
Have(Ban.)
Have(Drill)
Finish

GraphPlan

GraphPlan: Basic idea

- Construct a graph that encodes constraints on possible plans
- Use this "planning graph" to constrain search for a valid plan
- Planning graph can be built for each problem in a relatively short time

Planning graph

- Directed, leveled graph with alternating layers of nodes
- Odd layers ("**state levels**") represent candidate propositions that could possibly hold at step *i*
- Even layers ("**action levels**") represent candidate actions that could possibly be executed at step *i*, including maintenance actions [do nothing]
- **Arcs** represent preconditions, adds and deletes
- We can only execute one real action at any step, but the data structure keeps track of **all actions and states that are *possible***

GraphPlan properties

- STRIPS operators: conjunctive preconditions, no conditional or universal effects, no negations
  - Planning problem must be convertible to propositional representation
  - Can't handle continuous variables, temporal constraints, …
  - Problem size grows exponentially
- Finds "shortest" plans (by some definition)
- Sound, complete, and will terminate with failure if there is no plan

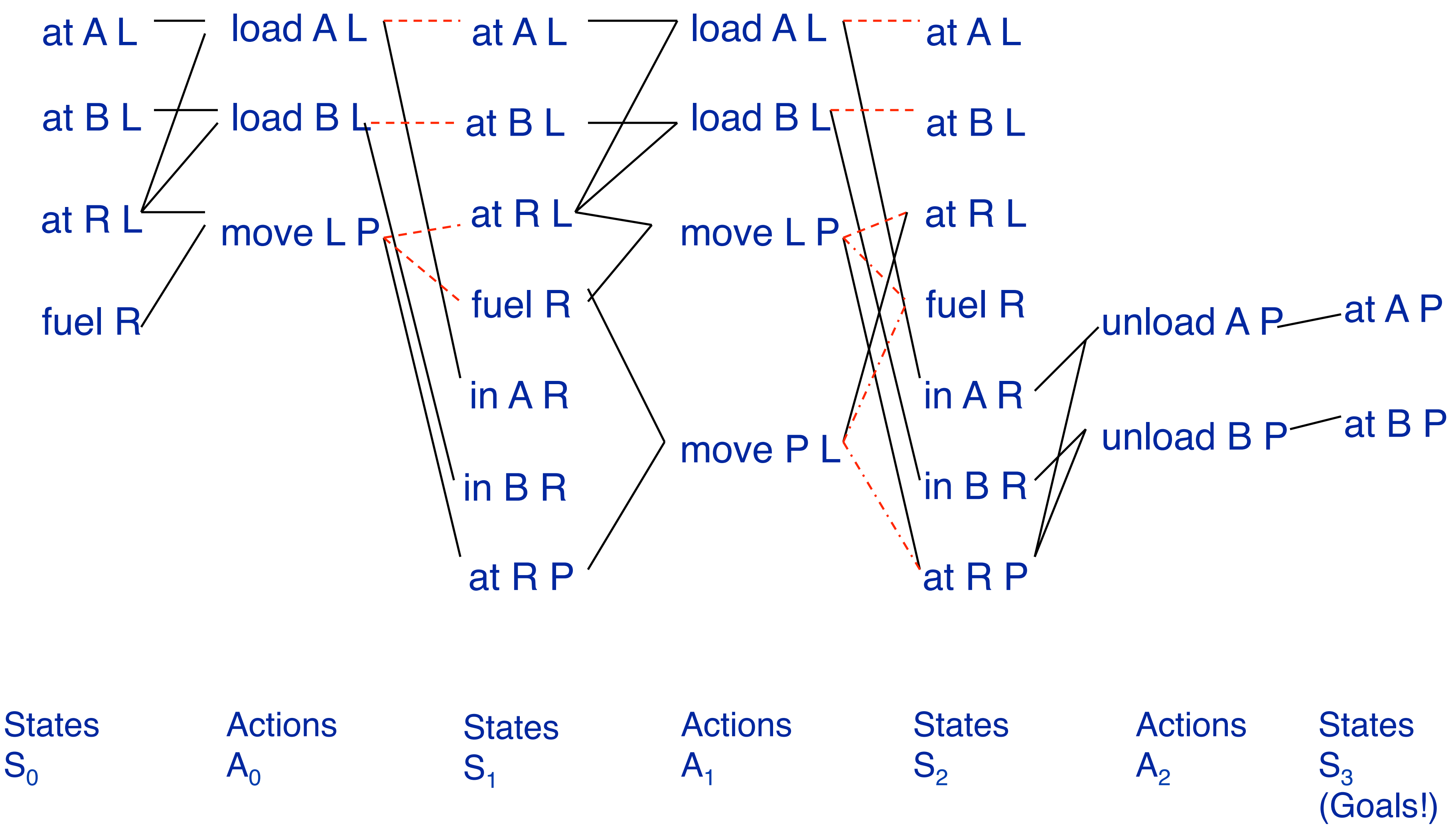
What actions and what literals?

- Add an action in level A_i if ***all*** of its preconditions are present in level S_i
- Add a literal in level S_i if it is the effect of ***some*** action in level A_{i-1} (*including no-ops*)
- Level S_0 has all of the literals from the initial state

Simple domain

- Literals:
  - at X Y — X is at location Y
  - fuel R — rocket R has fuel
  - in X R — X is in rocket R
- Actions:
  - load X L — load X (onto R) at location L (X and R must be at L)
  - unload X L — unload X (from R) at location L (R must be at L)
  - move X Y — move rocket R from X to Y (R must be at X and have fuel)
- Graph representation:
  - Solid black lines: preconditions/effects
  - Dotted red lines: negated preconditions/effects

Example planning graph



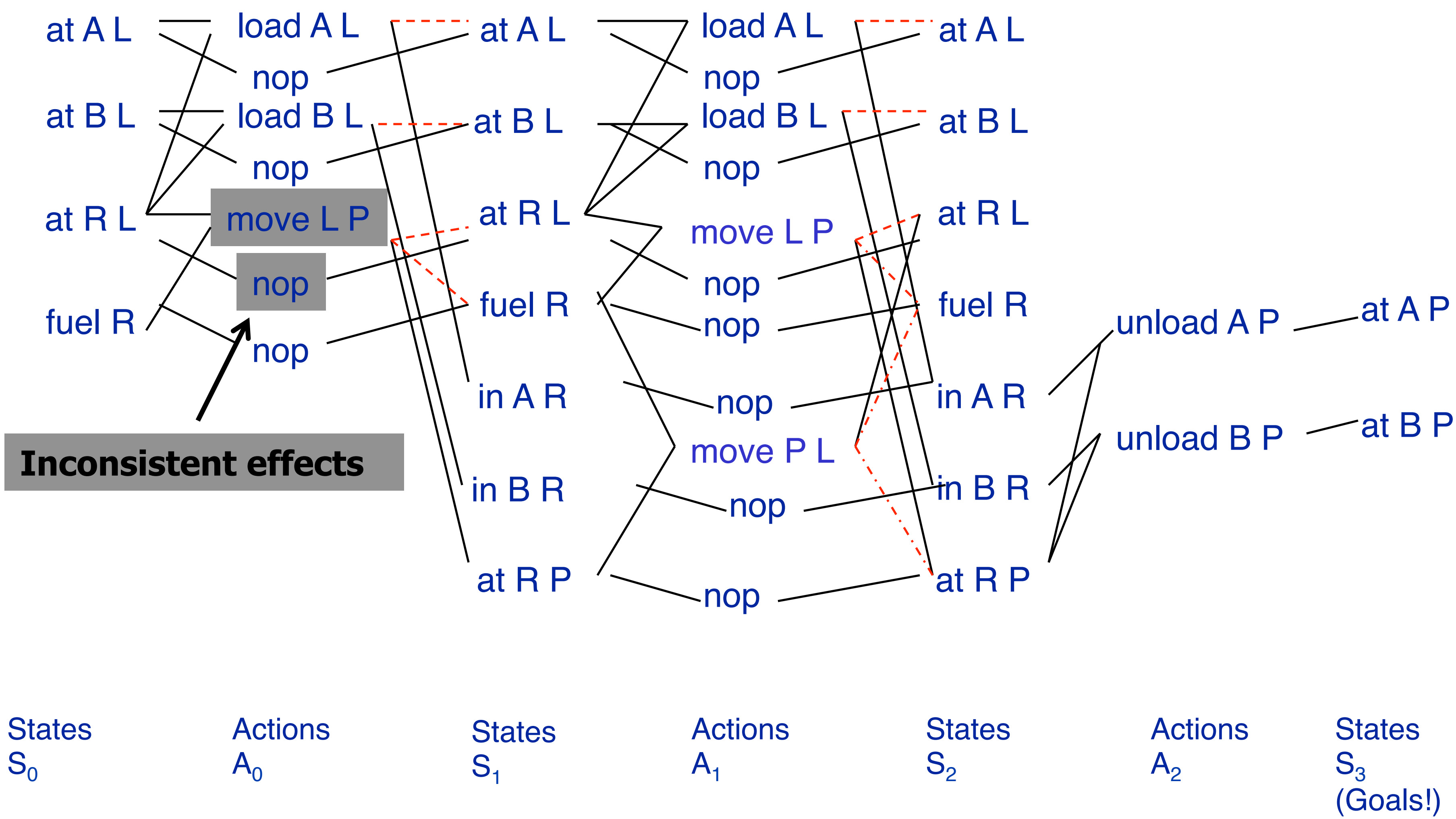
Valid plans

- A **valid** plan is a planning graph in which:
  - Actions at the same level don't interfere (delete each other's preconditions or add effects)
  - Each action's preconditions are true at that point in the plan
  - Goals are satisfied at the end of the plan

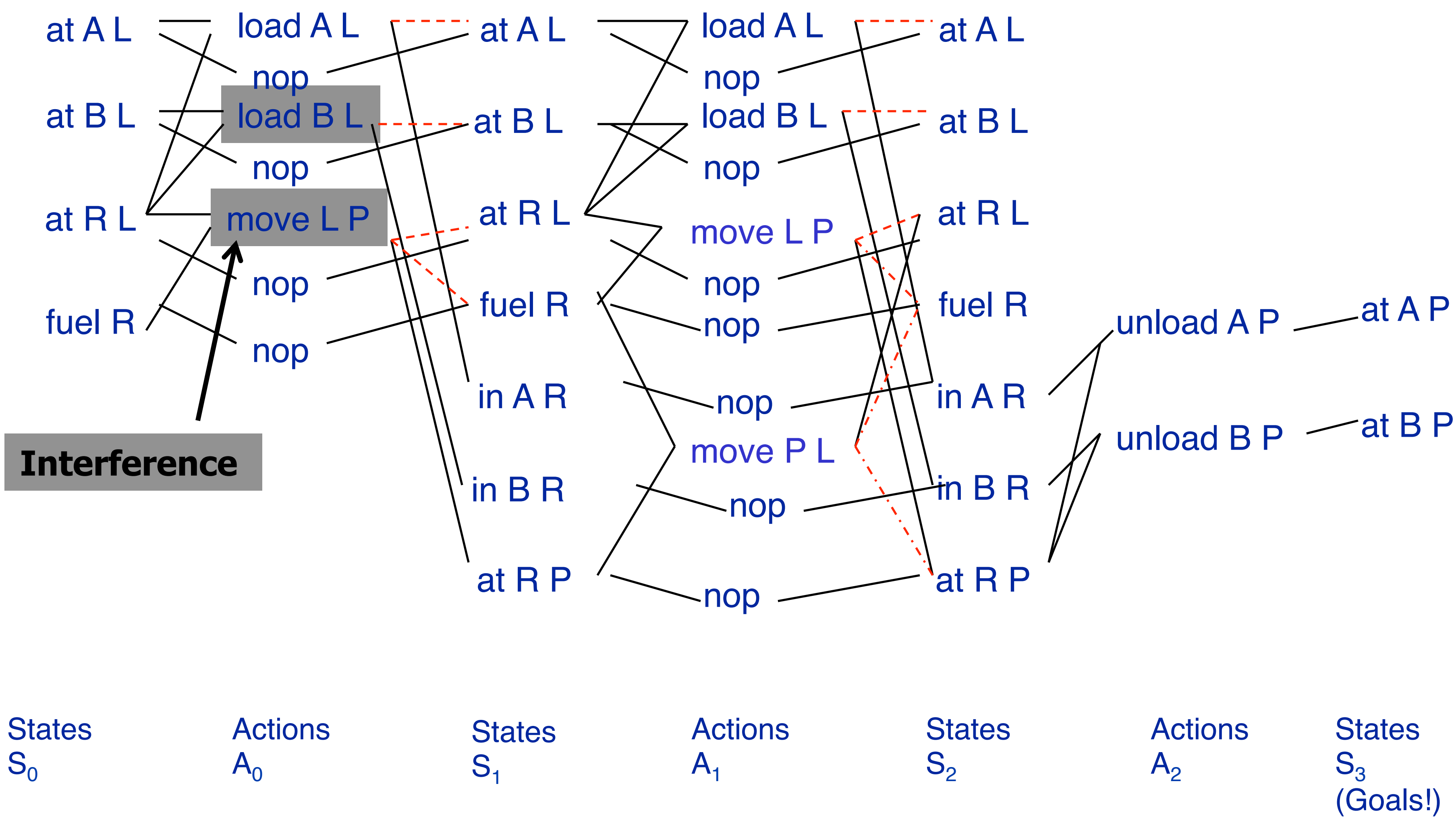
Exclusion relations (mutexes)

- Two actions (or literals) are **mutually exclusive ("mutex")** at step *i* if no valid plan could contain both actions at that step
- Can quickly find and mark *some* mutexes:
  - **Inconsistent effects:** Two actions whose effects are mutex with each other
  - **Interference**: Two actions that interfere (the effect of one negates the precondition of another) are mutex
  - **Competing needs**: Two actions are mutex if any of their preconditions are mutex with each other
  - **Inconsistent support**: Two literals are mutex if all ways of creating them both are mutex

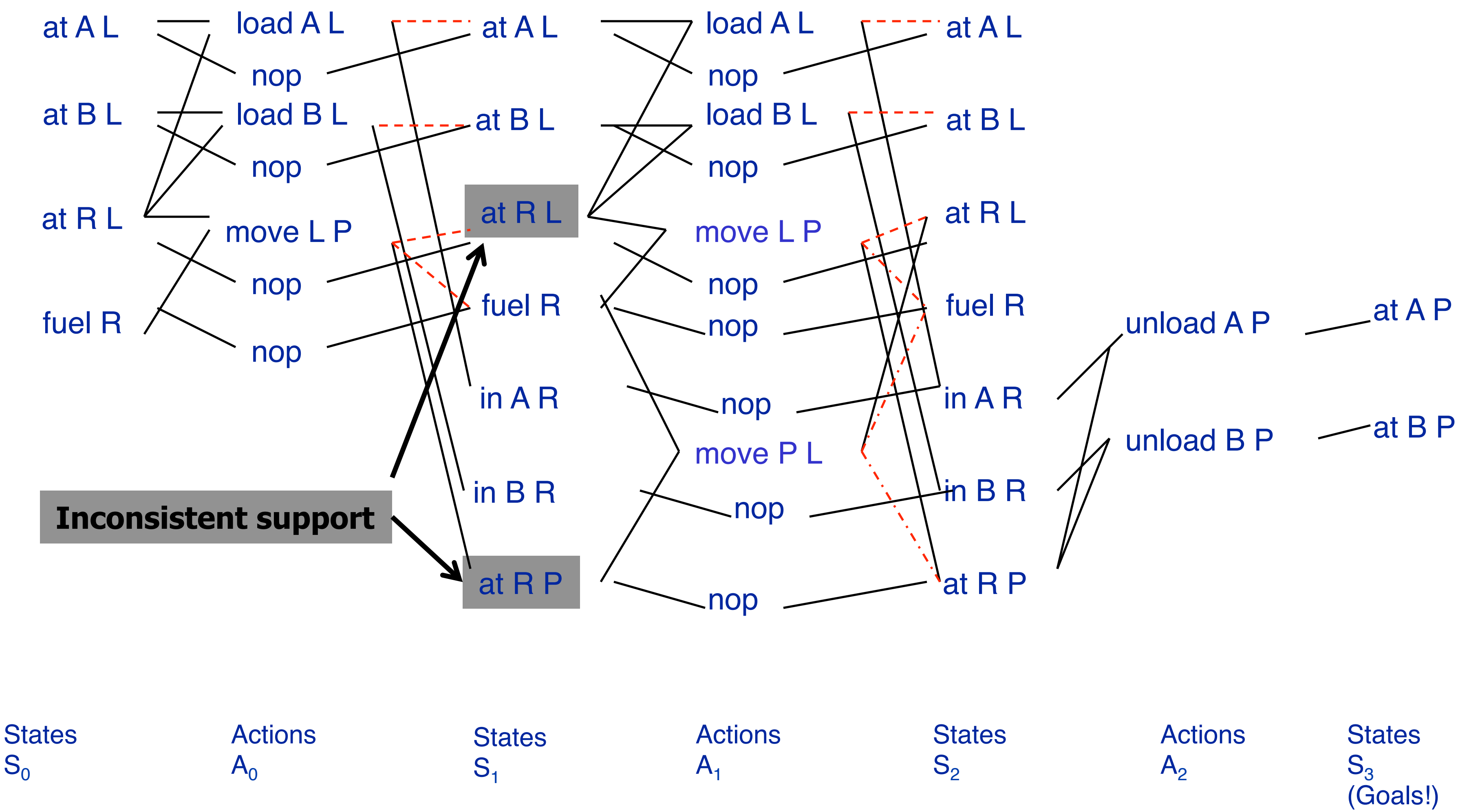
Example: Mutex constraints



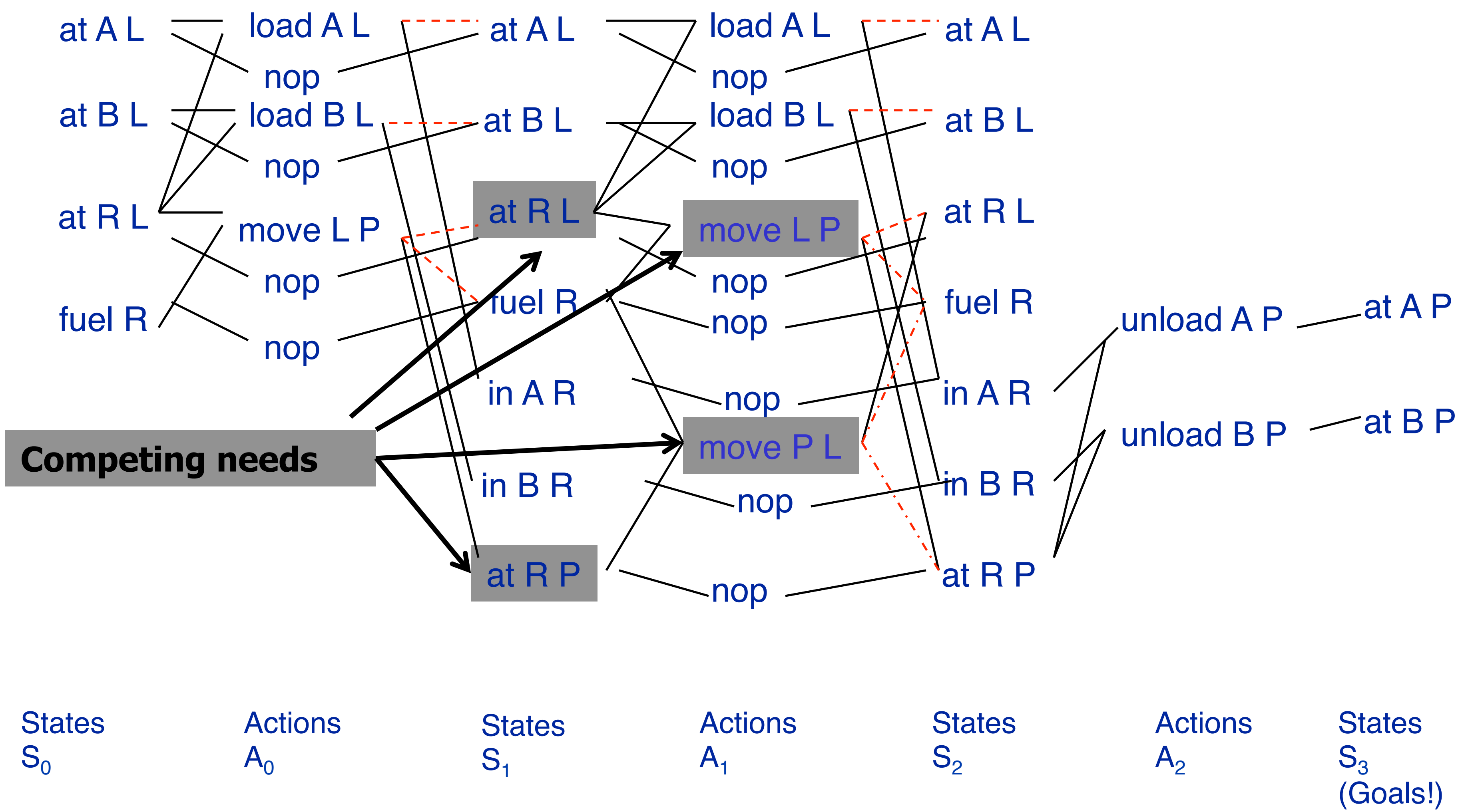
Example: Mutex constraints



Example: Mutex constraints



Example: Mutex constraints



Extending the planning graph

- **Action level A_i:**
  - Include all instantiations of all actions (including maintains (no-ops)) that have all of their **preconditions satisfied** at level S_i, with no two being mutex
  - Mark as mutex all **action-maintain (nop) pairs** that are incompatible
  - Mark as mutex all **action-action pairs** that have competing needs
- **State level S_{i+1}:**
  - Generate all propositions that are the **effect of some action** at level A_i
  - Mark as mutex all pairs of propositions that can only be generated by **mutex action pairs**

Basic GraphPlan algorithm

- **Grow** the planning graph (PG) until all goals are reachable and none are pairwise mutex. (If PG levels off [reaches a steady state] first, fail)
- **Search** the PG for a **valid plan**
- If none found, **add a level** to the PG and try again

Creating the planning graph is usually fast

- Theorem:

  The size of the t-level planning graph and the time to create it are polynomial in:
  - t (number of levels),
  - n (number of objects),
  - m (number of operators), and
  - p (number of propositions in the initial state)

Searching for a plan

- Backward chain on the planning graph
- Complete all goals at one level before going back
- At level *i*, pick a non-mutex subset of actions that achieve the goals at level *i+1*. The preconditions of these actions become the goals at level *i*
  - Various heuristics can be used for choosing which actions to select
- Build the action subset by iterating over goals, choosing an action that has the goal as an effect. Use an action that was already selected if possible. Do forward checking on remaining goals.

SATPlan
(chapter 7.7.4)

SATPlan

- Formulate the planning problem as a CSP
- Assume that the plan has k actions
- Create a binary variable for each possible action a:
  - Action(a,i) (TRUE if action a is used at step i)
- Create variables for each proposition that can hold at different points in time:
  - Proposition(p,i) (TRUE if proposition p holds at step i)

Constraints

- Only one action can be executed at each time step (XOR constraints)
- Constraints describing effects of actions
- Persistence: if an action does not change a proposition p, then p’s value remains unchanged
- A proposition is true at step i only if some action (possibly a maintain action) made it true
- Constraints for initial state and goal state

Now apply our favorite CSP solver!

Still more variations…

- FF (Fast-Forward):
  - Forward-chaining state space planning using relaxation-based heuristic and *many* other heuristics and "tweaks"
- Blackbox:

STRIPS-based plan representation

↓

Planning graph

↓

CNF representation

↓

CSP/SAT solver

↓

CSP solution

↓

Plan

Whew!