Planning

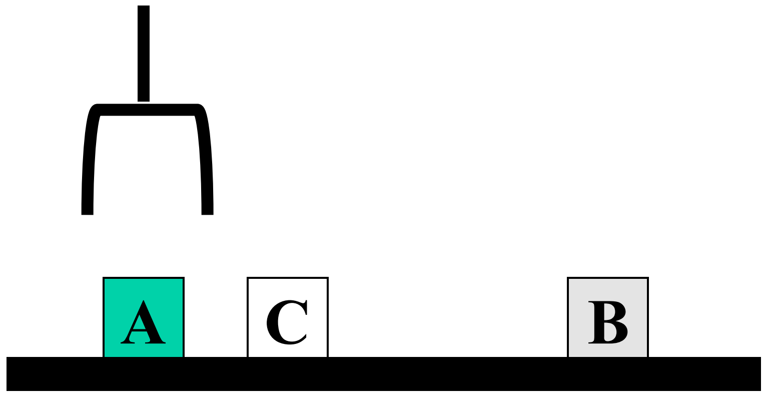
Chapter 11.1-11.3

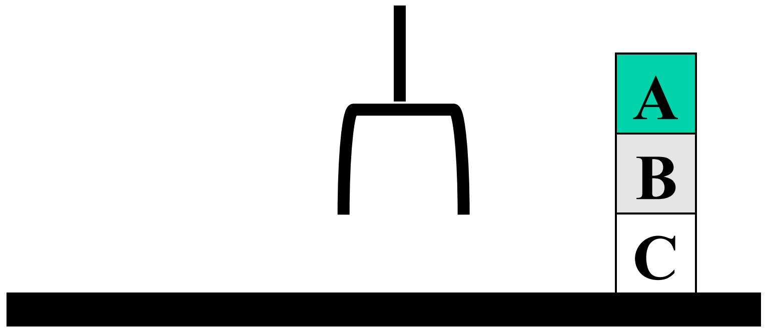
Some material adopted from notes
by Andreas Geyer-Schulz
and Chuck Dyer

Overview

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

Blocks World Planning





Blocks world

The **blocks world** is a micro-world consisting 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 uses a logic notation:

ontable(b)	ontable(d)
on(c,d)	holding(a)
clear(b)	clear(c)

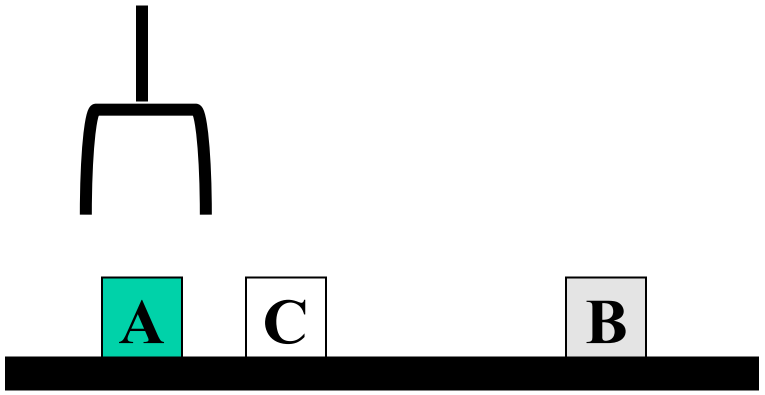
Typical BW planning problem

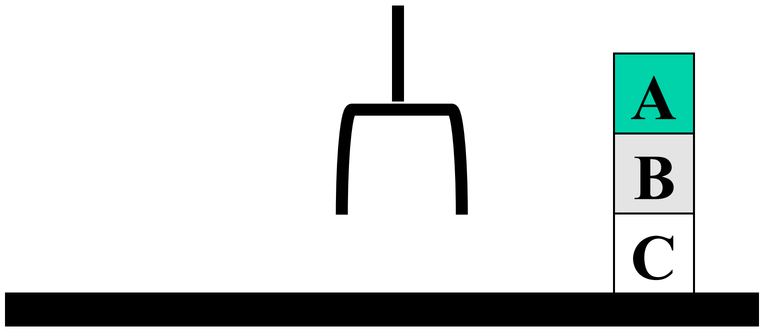
Initial state:

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

Goal:

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





Typical BW planning problem

Initial state:

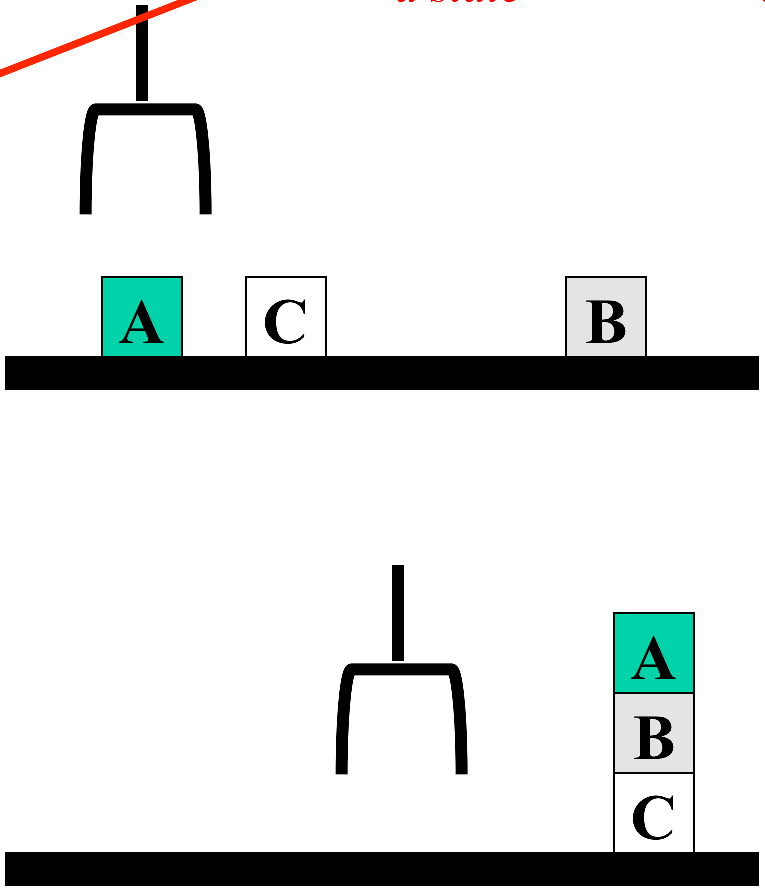
- clear(a)
- clear(b)
- clear(c)
- ontable(a)
- ontable(b)
- ontable(c)
- handempty

Goal state:

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

assertions describing a state

atomic robot actions



Plan:

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

Planning problem

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

  compute plan as sequence of operator instances that when executed in initial state changes it to goal state
- States usually specified as conjunction of conditions, e.g. $ontable(a) \land on(b, a)$

Planning vs. problem solving

- Planning and problem solving methods can often solve similar problems
- Planning is more powerful and efficient 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)
- Sub-goals can be planned independently, reducing the complexity of the planning problem

Typical assumptions

- Atomic time: Each action is indivisible
- No concurrent actions allowed, but actions need not be ordered w.r.t each other in the plan
- Deterministic actions: action results completely determined — no uncertainty in their effects
- Agent is the sole cause of change in the world
- Agent is omniscient with complete knowledge of the state of the world
- Closed world assumption where everything known to be true in the world is included in the state description and anything not listed is false

Blocks world

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

Some domain constraints:

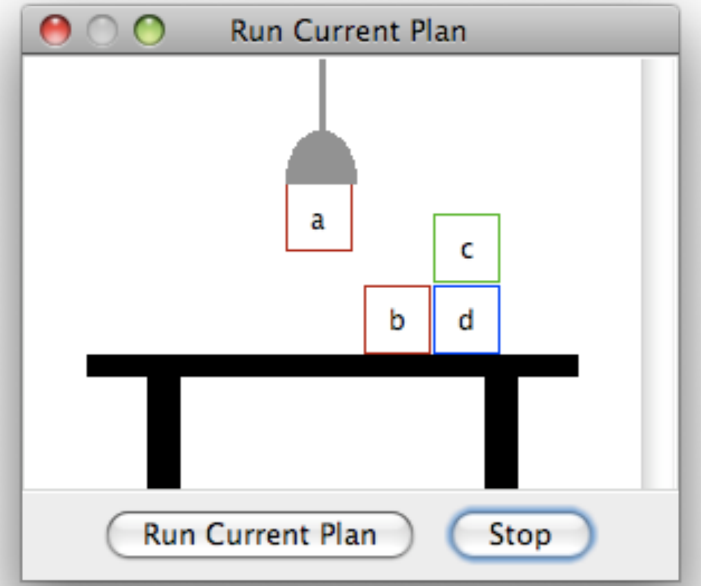
- Only one block can be on another block
- Any number of blocks can be on the table
- The hand can only hold one block

Typical representation:

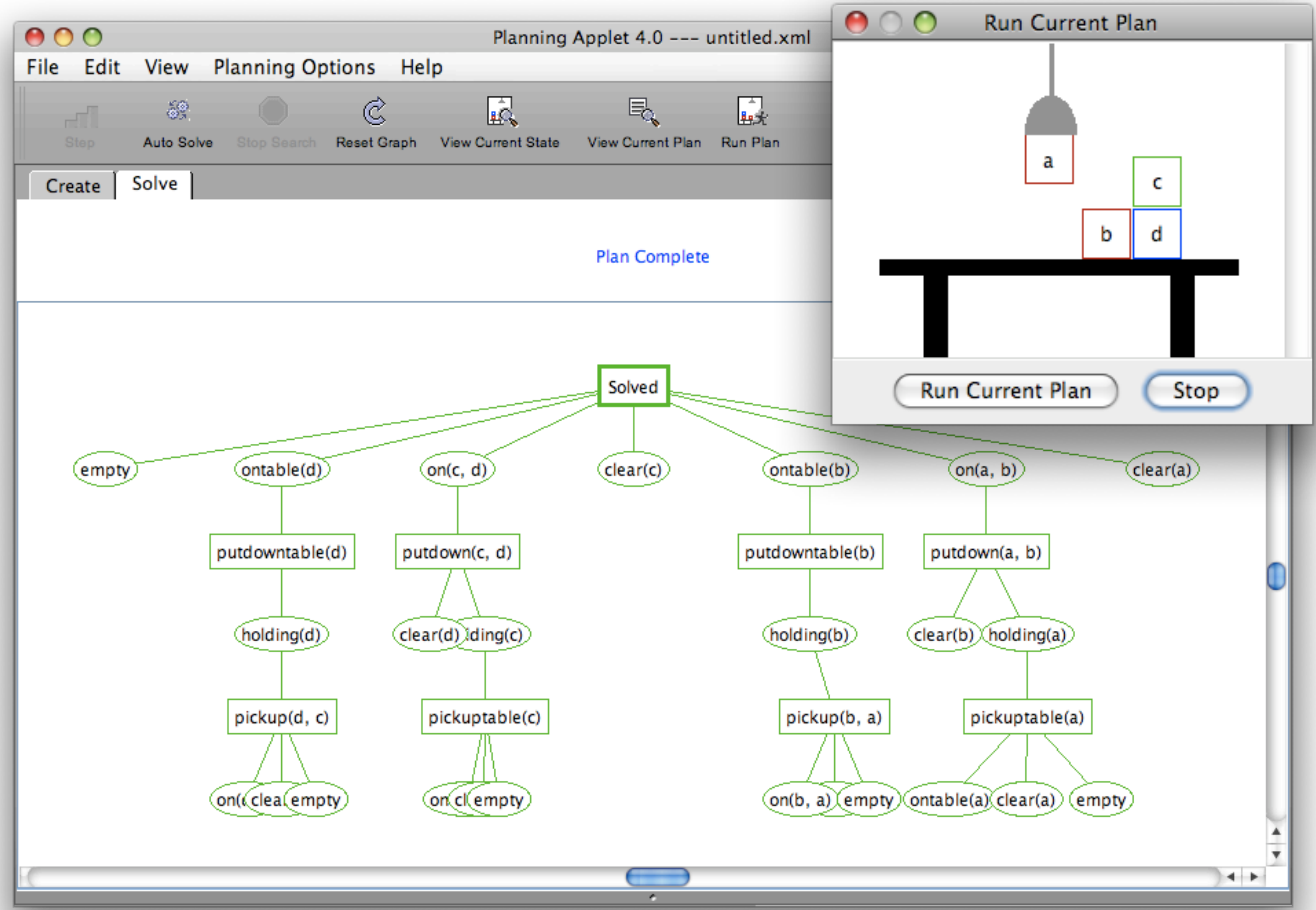
ontable(b) ontable(d)

on(c,d) holding(a)

clear(b) clear(c)



Meant to be a simple model!



Try demo at http://aispace.org/planning/

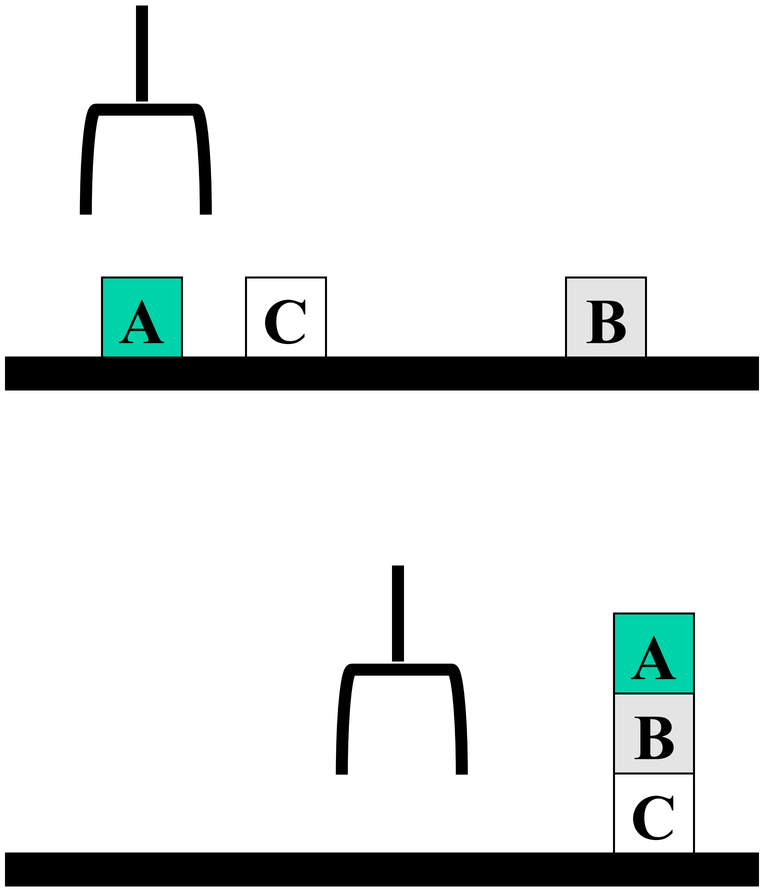
Typical BW planning problem

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)

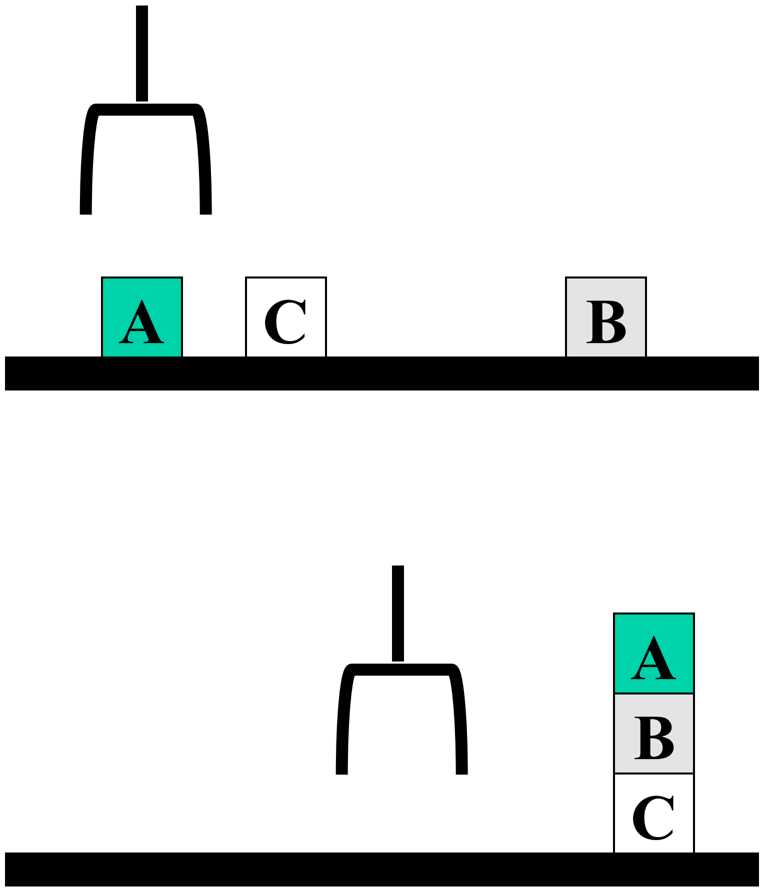
Another BW planning problem

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)

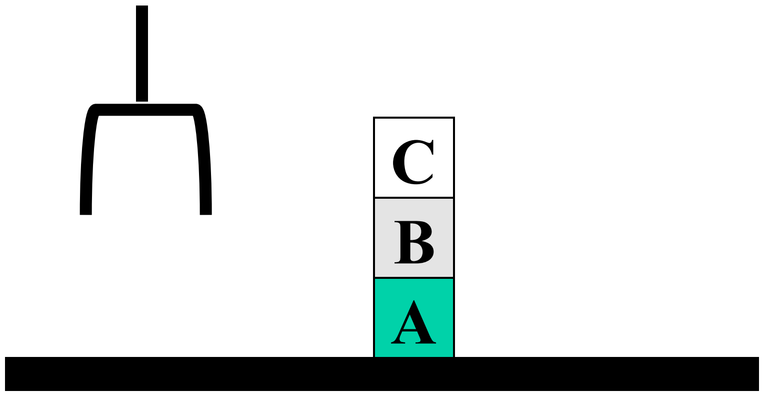
Yet Another BW planning problem

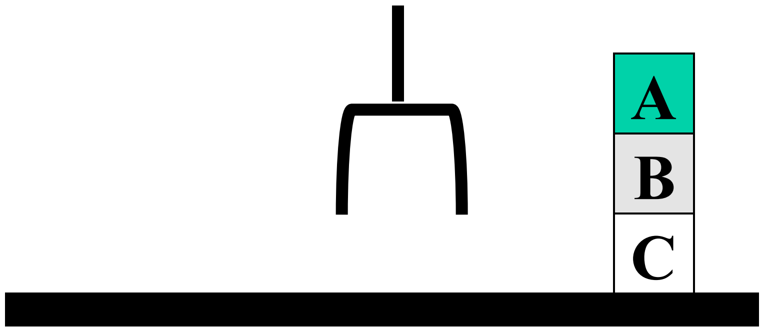
Initial state:

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

Goal:

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





Plan:

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

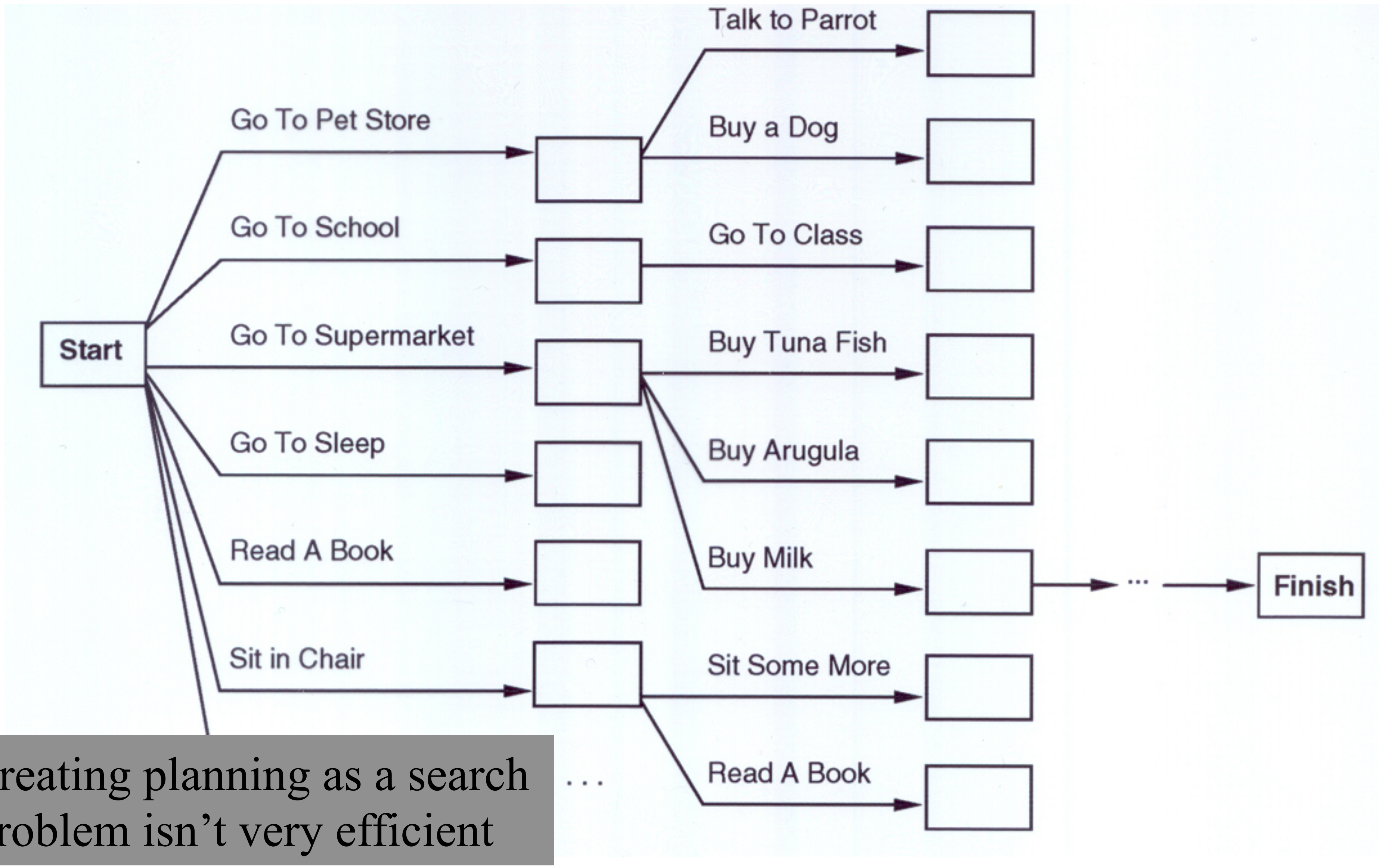
Major approaches

- Planning as search
- GPS / STRIPS
- Situation calculus
- Partial order planning
- Hierarchical decomposition (HTN planning)
- Planning with constraints (SATplan, Graphplan)
- Reactive planning

Planning as Search

- Can think of planning as a search problem
- **Actions:** generate successor states
- **States:** completely described & only used for successor generation, heuristic fn. evaluation & goal testing
- **Goals:** represented as a goal test and using a heuristic function
- **Plan representation:** unbroken sequences of actions forward from initial states or backward from goal state

"Get a quart of milk, a bunch of bananas and a variable-speed cordless drill."



Treating planning as a search problem isn't very efficient

General Problem Solver

- The General Problem Solver (GPS) system was an early planner (Newell, Shaw, and Simon, 1957)
- GPS generated actions that reduced the difference between some state and a goal state
- GPS used *Means-Ends Analysis*
  - Compare given to desired states; select best action to do next
  - Table of differences identifies actions to reduce types of differences
- GPS was a state space planner: operated in domain of state space problems specified by initial state, some goal states, and set of operations
- Introduced general way to use domain knowledge to select most promising action to take next

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 initial situation leads to desired result
- This is how the "neats" approach the problem

Situation calculus

- **Initial state**: 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** describe how world changes as a result of 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 situation resulting from executing action a in situation s
- Action sequences also useful: Result'(l,s) is 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 situation satisfying the goal:

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

  ∧ Have(Milk, Result'(p,S_0))

  ∧ Have(Bananas, Result'(p,S_0))

  ∧ Have(Drill, Result'(p,S_0))

- We 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

- 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

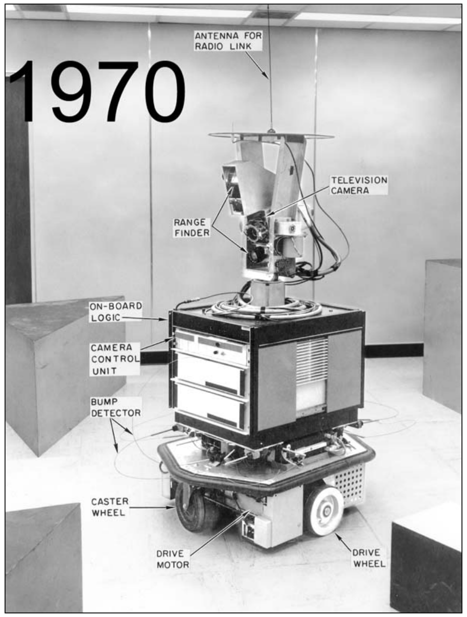
- Fine in theory, but problem solving (search) is exponential in worst case
- Resolution theorem proving only finds *a* proof (plan), not necessarily a good plan
- So, restrict language and use special-purpose algorithm (a planner) rather than general theorem prover
- Planning is a common task for intelligent agents, so it's reasonable to have a special subsystem for it

Strips planning representation

- Classic approach first used in the **STRIPS** (Stanford Research Institute Problem Solver) planner
- A State is a conjunction of ground literals

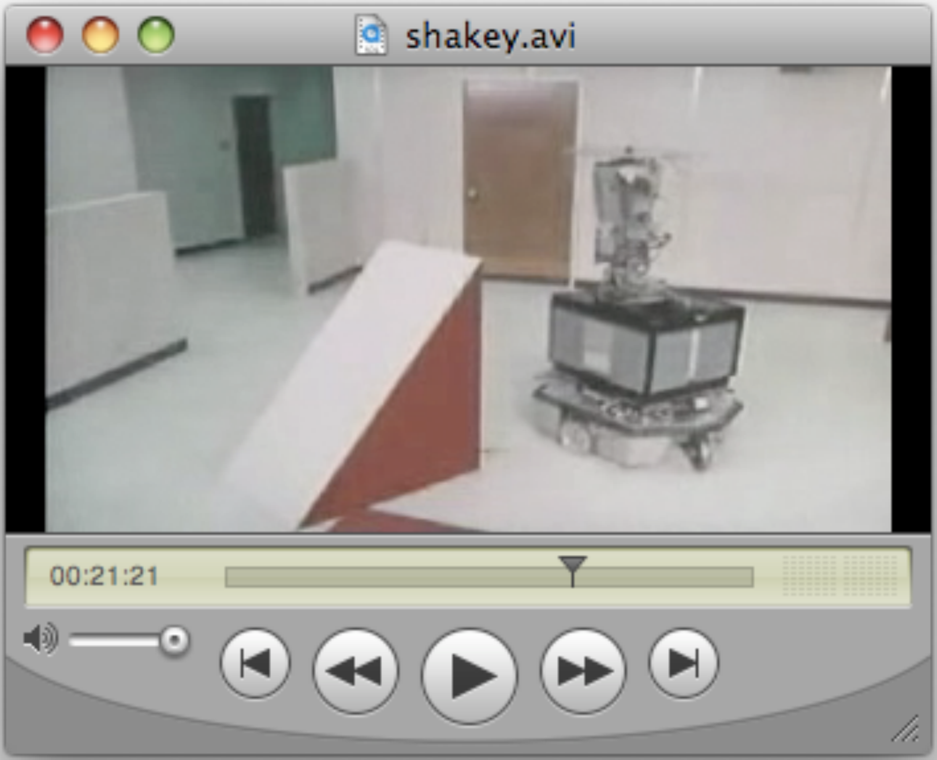
  at(Home) ∧ ¬have(Milk) ∧ ¬have(bananas) ...
- Goals are conjunctions of literals, but may have variables, assumed to be existentially quantified

  at(?x) ∧ have(Milk) ∧ have(bananas) ...
- Need not fully specify state
  - Non-specified conditions either don't-care or assumed false
  - Represent many cases in small storage
  - May only represent changes in state rather than entire situation
- Unlike theorem prover, not seeking whether goal is true, but is there a sequence of actions to attain it



Shakey the robot

Shakey video circa 1969



https://youtu.be/qXdn6ynwpiI

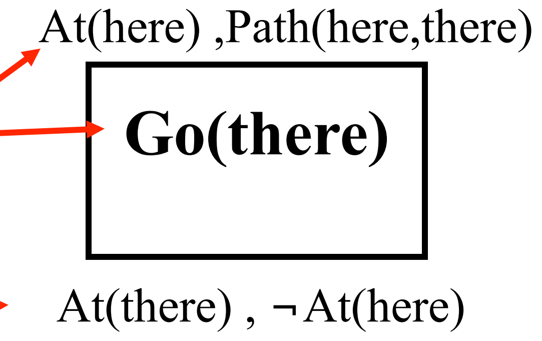
Operator/action representation

- Operators contain three components:
  - **Action description**
  - **Precondition** - conjunction of positive literals
  - **Effect** - conjunction of positive or negative literals describing 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 operator is applied; effects are true immediately after

Blocks world operators

- 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 represented by
  - list of *preconditions*
  - list of new facts to be added (*add-effects*)
  - list of facts to be removed (*delete-effects*)
  - optionally, set of (simple) *variable constraints*
- For example stack(X,Y):

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 (Prolog)

```
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** - push down stack of goals to be solved, with current goal on top
- If current goal not satisfied by present state, find operator that adds it and push operator and its preconditions (subgoals) on stack
- When a current goal is satisfied, POP from stack
- When an operator is on top stack, record the application of that operator on the plan sequence and use the operator's add and delete lists to update the current state

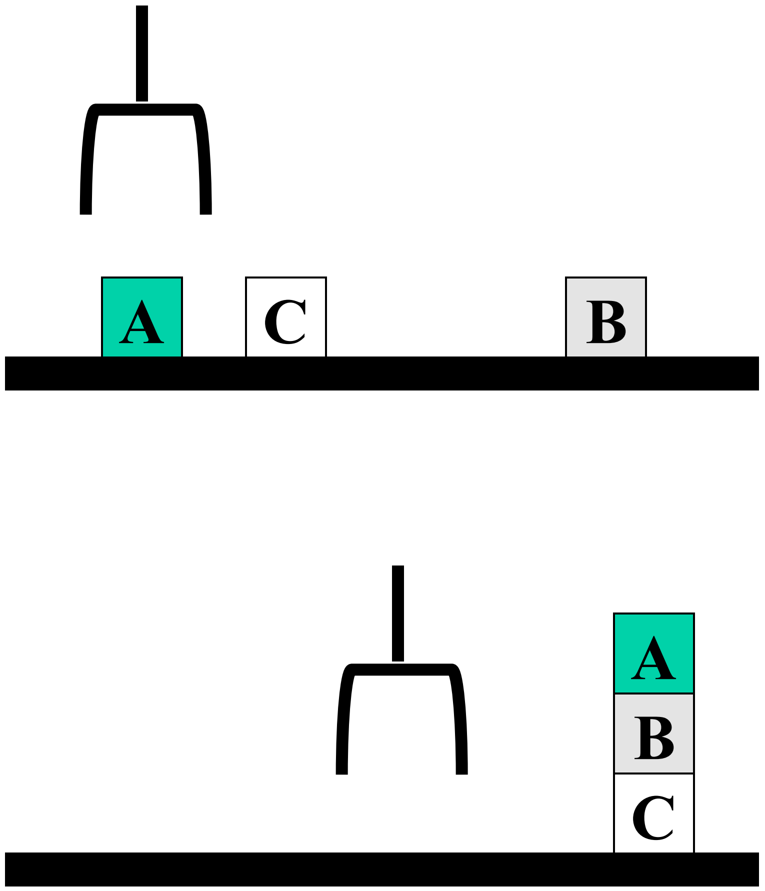
Typical BW planning problem

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)

Trace (Prolog)

```
strips([on(b,c),on(a,b),ontable(c)],[clear(a),clear(b),clear(c),ontable(a),ontable(b),ontable(c),handempty],[])
Achieve on(b,c) via stack(b,c) with preconds: [holding(b),clear(c)]
    strips([holding(b),clear(c)],[clear(a),clear(b),clear(c),ontable(a),ontable(b),ontable(c),handempty],[])
    Achieve holding(b) via pickup(b) with preconds: [ontable(b),clear(b),handempty]
      strips([ontable(b),clear(b),handempty],[clear(a),clear(b),clear(c),ontable(a),ontable(b),ontable(c),handempty],[])
    Applying pickup(b)
    strips([holding(b),clear(c)],[clear(a),clear(c),holding(b),ontable(a),ontable(c)],[pickup(b)])
Applying stack(b,c)
strips([on(b,c),on(a,b),ontable(c)],[handempty,clear(a),clear(b),ontable(a),ontable(c),on(b,c)],[stack(b,c),pickup(b)])
Achieve on(a,b) via stack(a,b) with preconds: [holding(a),clear(b)]
    strips([holding(a),clear(b)],[handempty,clear(a),clear(b),ontable(a),ontable(c),on(b,c)],[stack(b,c),pickup(b)])
    Achieve holding(a) via pickup(a) with preconds: [ontable(a),clear(a),handempty]
      strips([ontable(a),clear(a),handempty],[handempty,clear(a),clear(b),ontable(a),ontable(c),on(b,c)],
      [stack(b,c),pickup(b)])
    Applying pickup(a)
    strips([holding(a),clear(b)],[clear(b),holding(a),ontable(c),on(b,c)],[pickup(a),stack(b,c),pickup(b)])
Applying stack(a,b)
strips([on(b,c),on(a,b),ontable(c)],[handempty,clear(a),ontable(c),on(a,b),on(b,c)],
  [stack(a,b),pickup(a),stack(b,c),pickup(b)])
```

Strips in Prolog

```
% strips(+Goals, +InitState, -Plan)
strips(Goal, InitState, Plan):-
  strips(Goal, InitState, [],  _, RevPlan),
  reverse(RevPlan, Plan).

% strips(+Goals,+State,+Plan,-NewState, NewPlan )
% Finished if each goal in Goals is true
% in current State.
strips(Goals, State, Plan, State, Plan) :-
  subset(Goals,State).
```

```
strips(Goals, State, Plan, NewState, NewPlan):-
  % Goal is an unsatisfied goal.
  member(Goal, Goals),
  (\+ member(Goal, State)),
  % Op is an Operator with Goal as a result.
  operator(Op, Preconditions, Adds, Deletes,_),
  member(Goal,Adds),
  % Achieve the preconditions
  strips(Preconditions, State, Plan, TmpState1,
    TmpPlan1),
  % Apply the Operator
  diff(TmpState1, Deletes, TmpState2),
  union(Adds, TmpState2, TmpState3).
  % Continue planning.
  strips(GoalList, TmpState3, [Op|TmpPlan1],
    NewState, NewPlan).
```

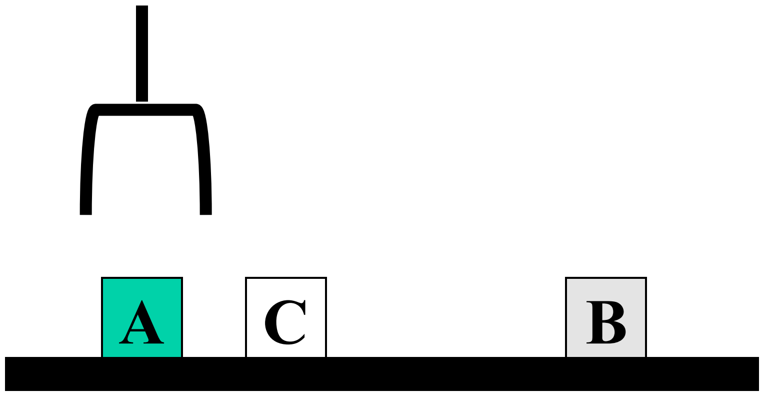
Another BW planning problem

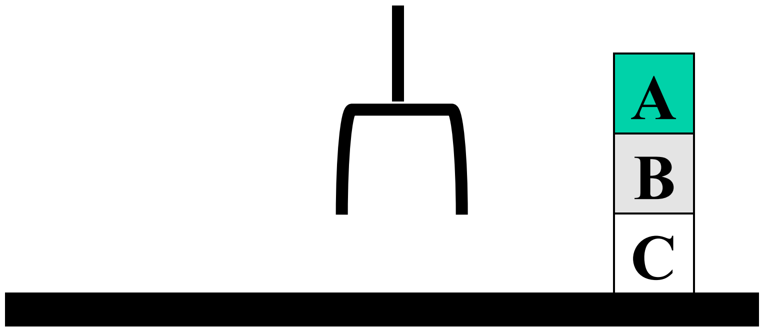
Initial state:

- 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)

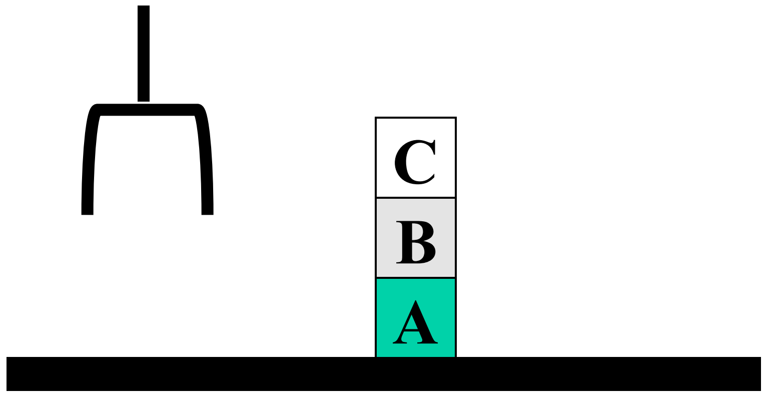
Yet Another BW planning problem

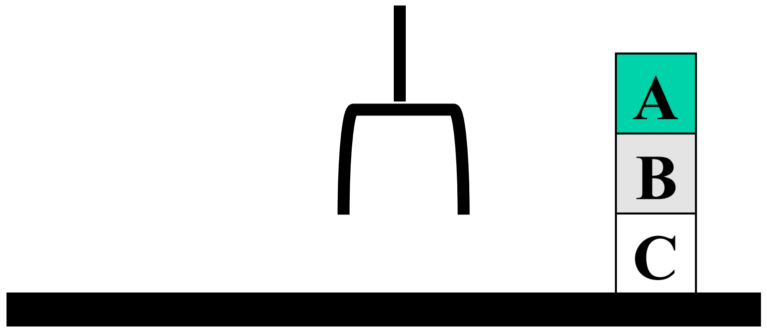
Initial state:

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

Goal:

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





Plan:

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

Yet Another BW planning problem

Initial state:

ontable(a)

ontable(b)

clear(a)

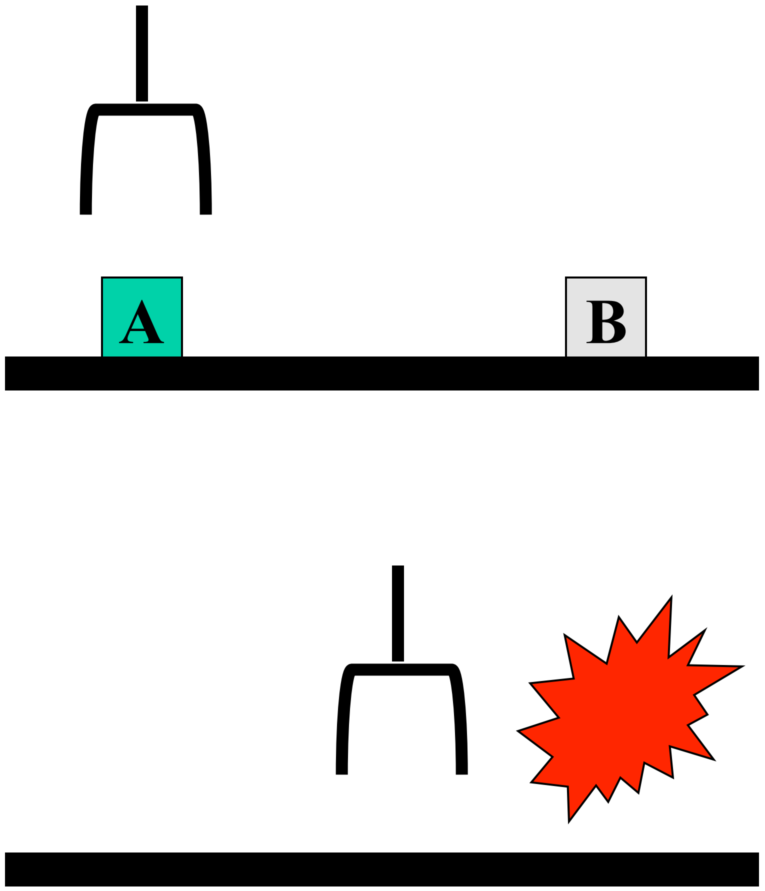
clear(b)

handempty

Goal:

on(a,b)

on(b,a)

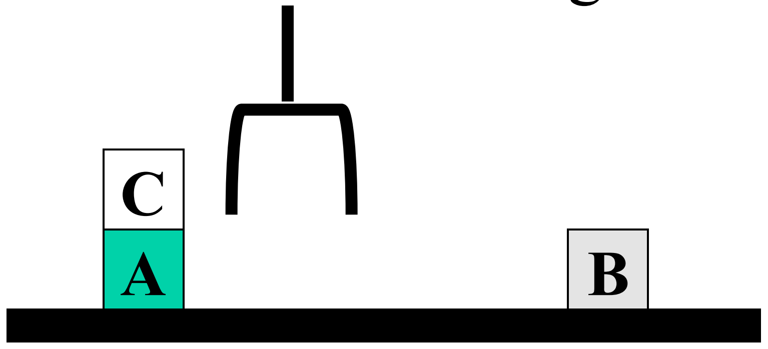


Plan:

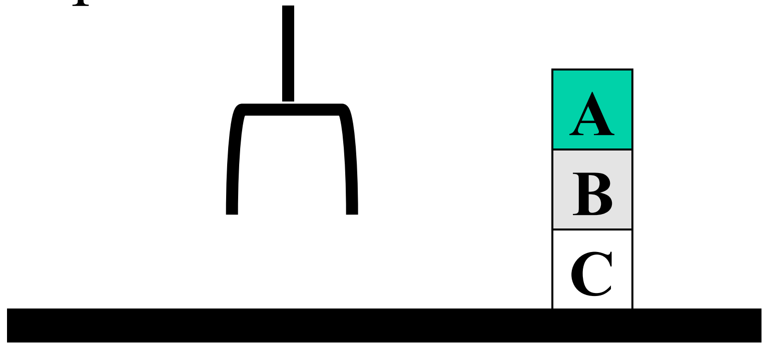
??

Goal interaction

- Simple planning algorithms assume independent sub-goals
  - Solve each separately and concatenate the solutions
- The "Sussman Anomaly" is the classic example of the goal interaction problem:
  - Solving on(A,B) first (via unstack(C,A), stack(A,B)) is undone when solving 2nd goal on(B,C) (via unstack(A,B), stack(B,C))
  - Solving on(B,C) first will be undone when solving on(A,B)
- Classic STRIPS couldn't handle this, although minor modifications can get it to do simple cases



Initial state

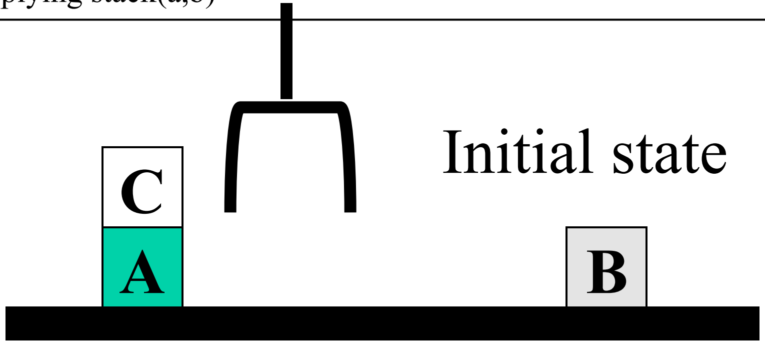


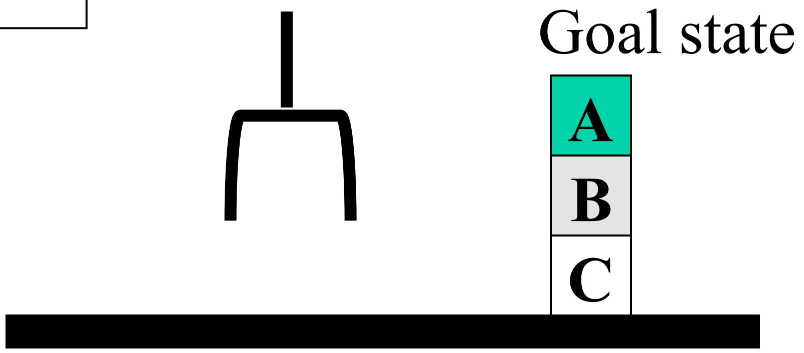
Goal state

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)
```





Sussman Anomaly

- Classic Strips assumed that once a goal had been satisfied it would stay satisfied
- Simple Prolog version selects any currently unsatisfied goal to tackle at each iteration
- This can handle this problem, at the expense of looping for other problems
- What's needed? -- notion of "protecting" a sub-goal so that it's not undone by later step

State-space planning

- STRIPS searches thru a space of situations (where you are, what you have, etc.)
  - Plan is a solution found by "searching" through situations to get to goal
- **Progression planners** search forward from initial state to goal state
  - Usually results in a high branching factor
- **Regression planners** search backward from goal
  - OK if operators have enough information to go both ways
  - Can reduce branching: you're only considering things relevant to goal
  - Handling a conjunction of goals is difficult (e.g., STRIPS)

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 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

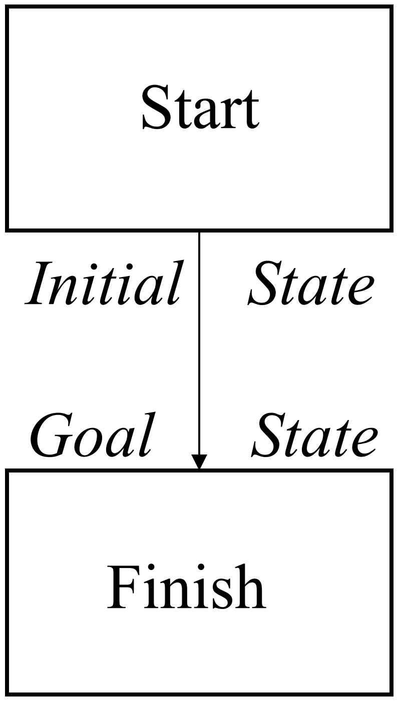
- **Linear planners** build plans as **totally ordered sequences** of steps
- **Non-linear planners (aka partial-order planners)** build plans as sets of steps with temporal constraints
  - constraints like $S1<S2$ if step S1 must come before S2
- One **refines** a partially ordered plan (POP) by either:
  - **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

Some example domains

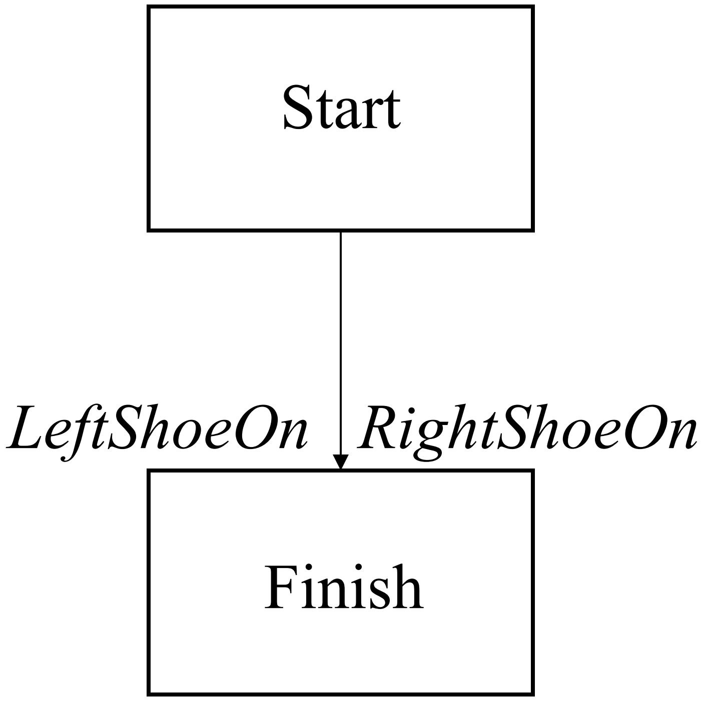
We’ll use some simple problems with a real world flavor to illustrate planning problems and algorithms

- Putting on your socks and shoes in the morning
  - Actions like put-on-left-sock, put-on-right-shoe
- Planning a shopping trip involving buying several kinds of items
  - Actions like go(X), buy(Y)

A simple graphical notation

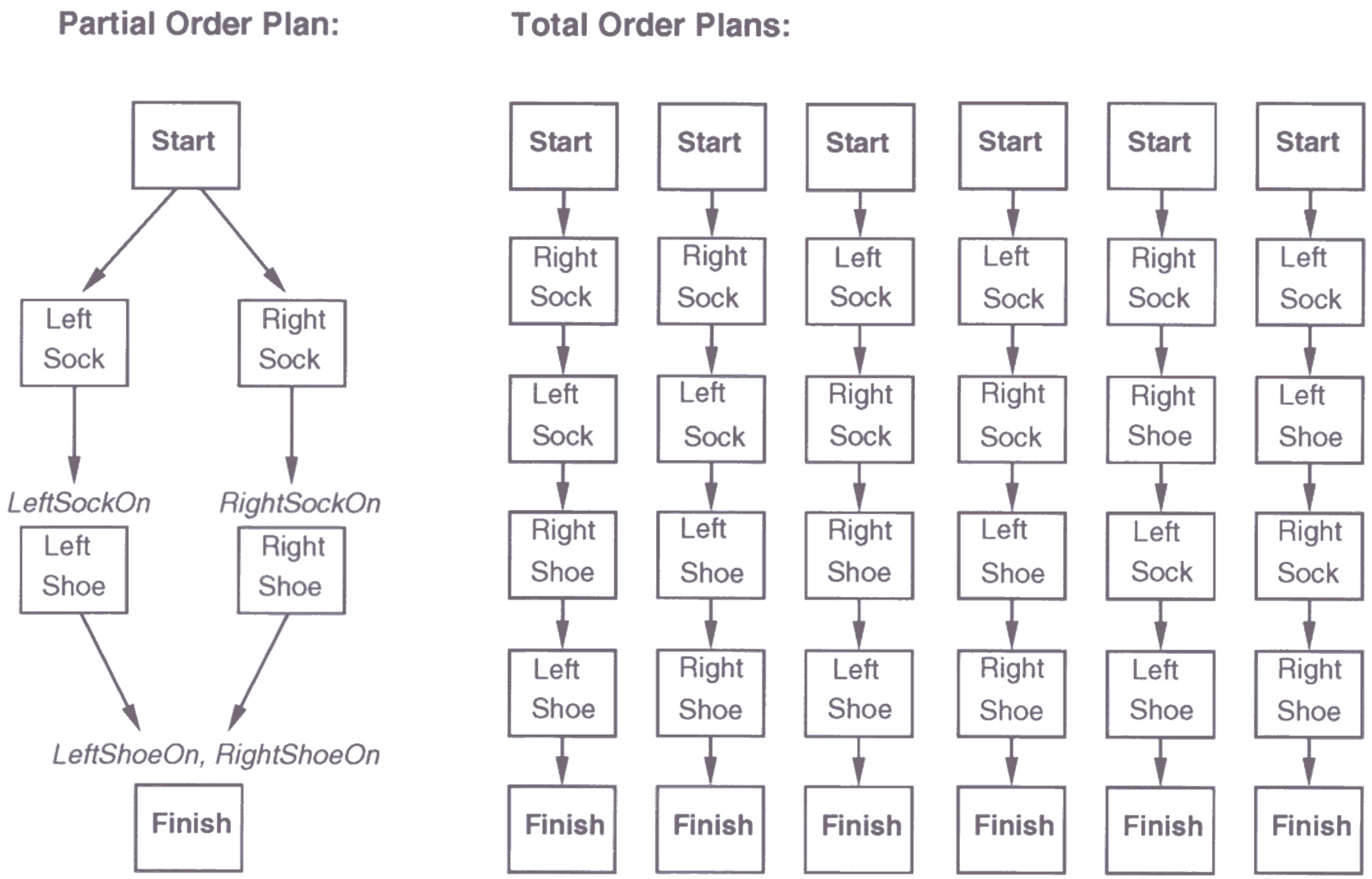


(a)



(b)

Partial Order Plan vs. Total Order Plan


Partial Order Plan:
Start
Left Sock
Right Sock
LeftSockOn
RightSockOn
Left Shoe
Right Shoe
LeftShoeOn, RightShoeOn
Finish
Total Order Plans:
Start → Right Sock → Left Sock → Right Shoe → Left Shoe → Finish
Start → Right Sock → Left Sock → Left Shoe → Right Shoe → Finish
Start → Left Sock → Right Sock → Right Shoe → Left Shoe → Finish
Start → Left Sock → Right Sock → Left Shoe → Right Shoe → Finish
Start → Right Sock → Right Shoe → Left Sock → Left Shoe → Finish
Start → Left Sock → Left Shoe → Right Sock → Right Shoe → Finish

The space of POPs is smaller than TOPs and hence involve less search

Least commitment

- Non-linear planners embody the principle of **least commitment**
  - only choose actions, orderings & variable bindings absolutely necessary, postponing other decisions
  - avoids early commitment to decisions that don’t really matter
- Linear planners always choose to add a plan step in a particular place in the sequence
- Non-linear planners choose to add a step and possibly some temporal constraints

Non-linear plan

A non-linear plan consists of

(1) A set of **steps** $\{S_1, S_2, S_3, S_4 \ldots\}$

Steps have operator descriptions, preconditions & post-conditions

(2) A set of **causal links** $\{ \ldots (S_i, C, S_j) \ldots \}$

Purpose of step S_i is to achieve precondition C of step S_j

(3) A set of **ordering constraints** $\{ \ldots S_i < S_j \ldots \}$

Step S_i must come before step S_j

Non-linear plan

A non-linear plan consists of

(1) A set of **steps** $\{S_1, S_2, S_3, S_4 \ldots\}$

Steps have operator descriptions, preconditions & post-conditions

(2) A set of **causal links** $\{ \ldots (S_i,C,S_j) \ldots\}$

Purpose of step S_i is to achieve precondition C of step S_j

(3) A set of **ordering constraints** $\{ \ldots S_i<S_j \ldots \}$

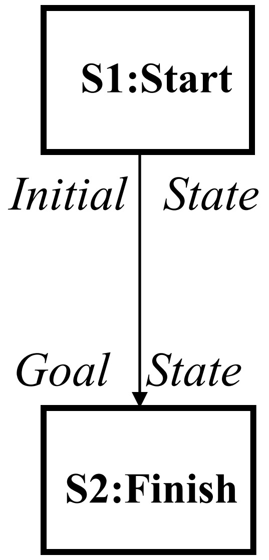
Step S_i must come before step S_j

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) (i.e., 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

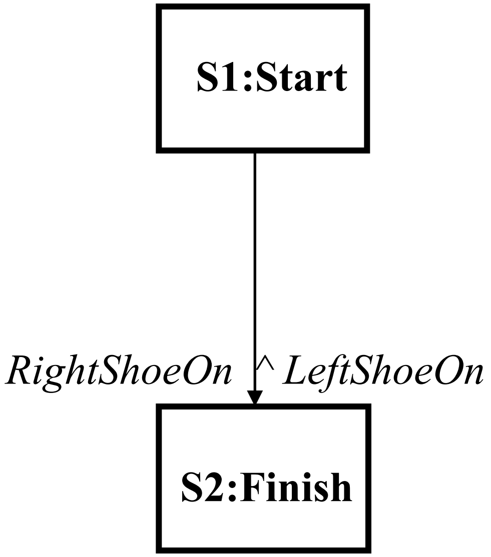
Operators:

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)],

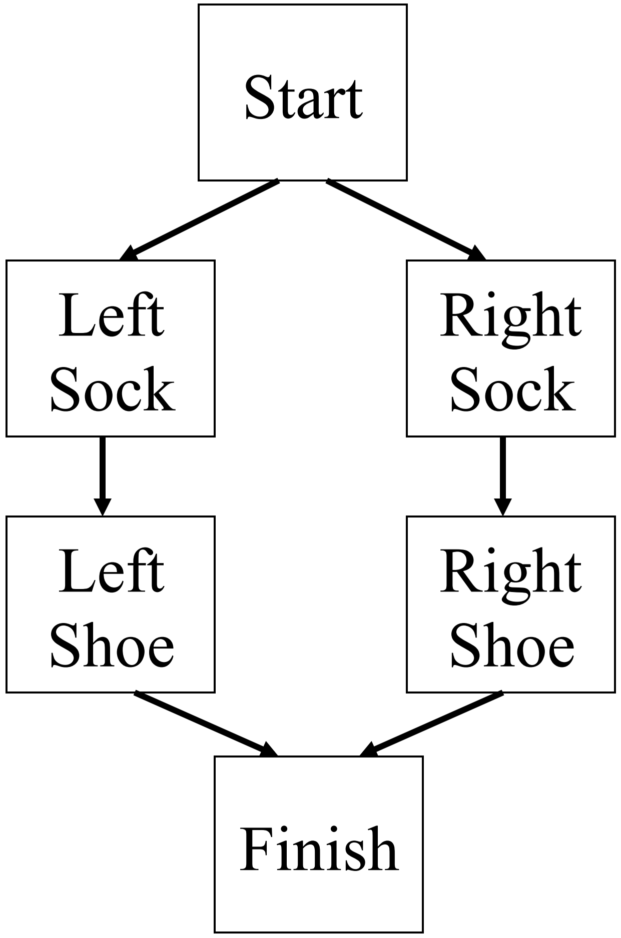
S2:[Op(Action:Finish,

Pre: RightShoeOn^LeftShoeOn)]}

Links: {}

Orderings: {S1<S2}

Solution

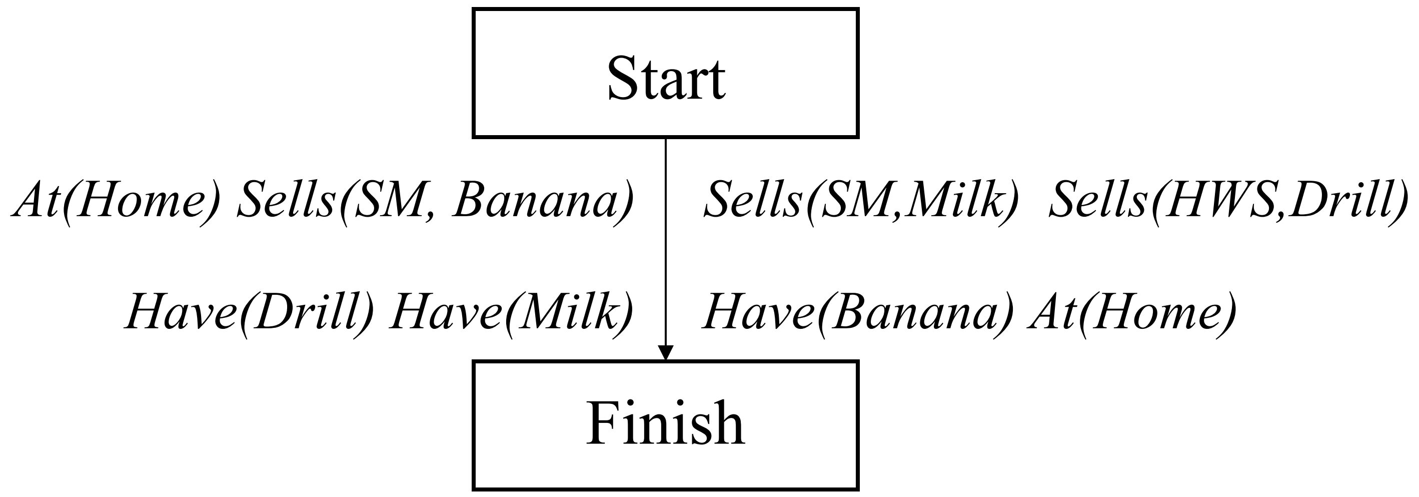


POP constraints and search heuristics

- 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** condition *c*:
  - Never add an intervening step S_3 that violates *c*
  - If a parallel action **threatens** *c* (i.e., has effect of negating or **clobbering** *c*), resolve threat by adding ordering links:
    - Order S_3 before S_1 (**demotion**)
    - Order S_3 after S_2 (**promotion**)

Partial-order planning example

- **Initially:** at home; SM sells bananas, milk; HWS sells drills
- **Goal:** Have milk, bananas, and a drill



```
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
```

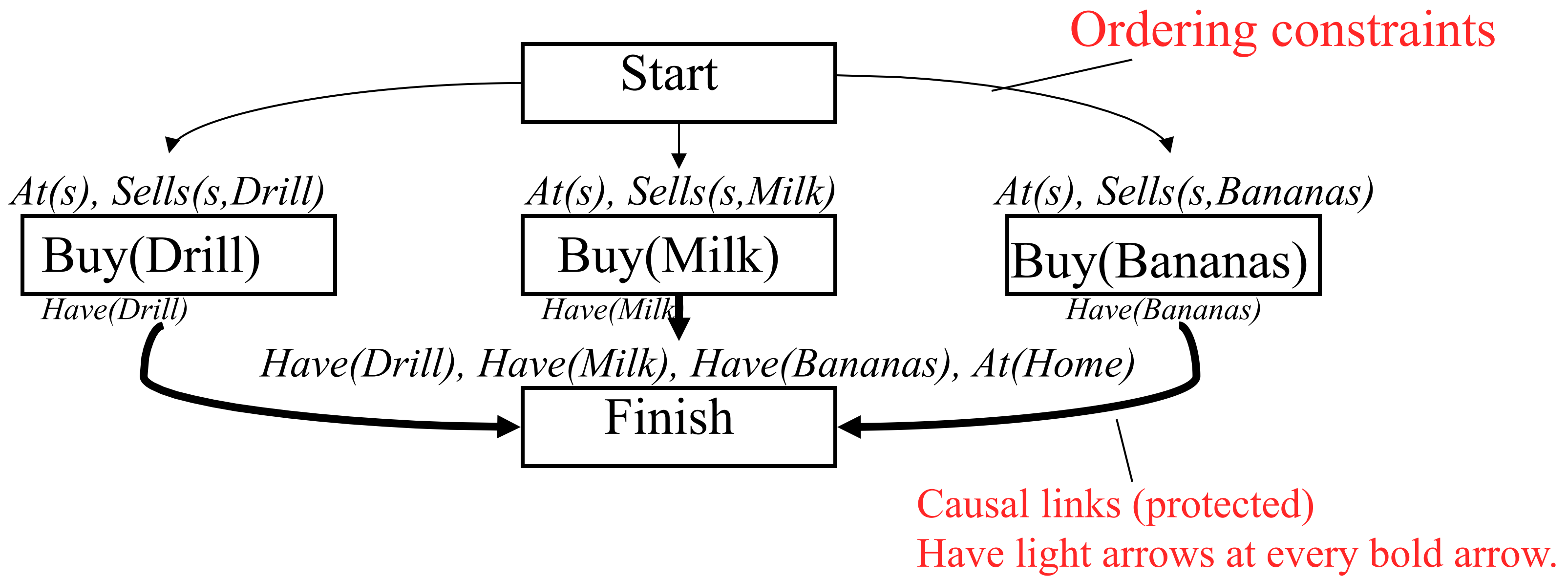
```
procedure CHOOSE-OPERATOR(plan, operators, $S_{need}$, c)

  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)
```

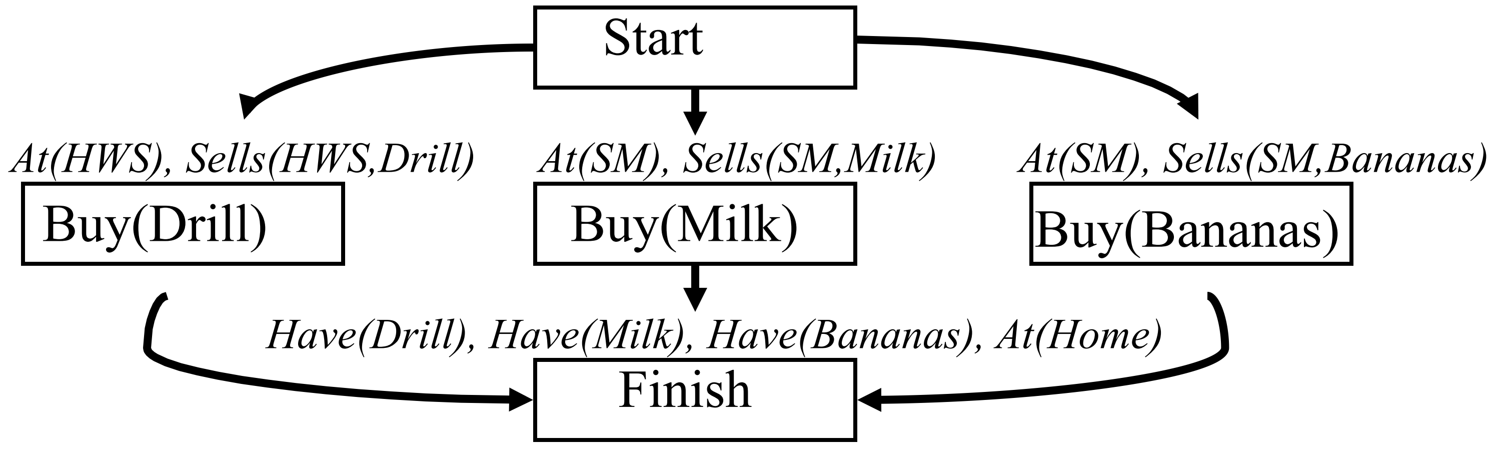
```
procedure RESOLVE-THREATS(plan)

  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
```

Planning

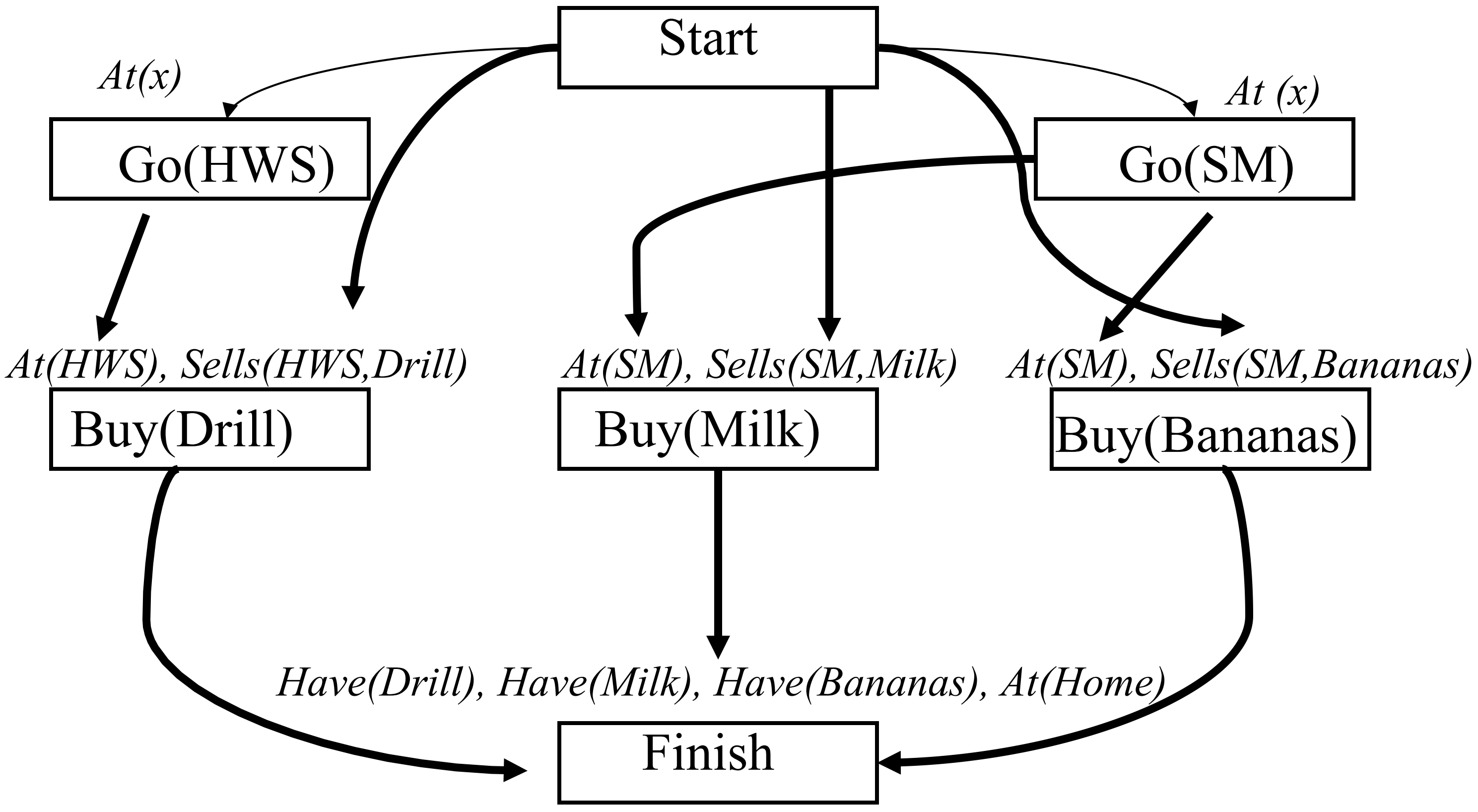


Ordering constraints
Start
At(s), Sells(s,Drill)
At(s), Sells(s,Milk)
At(s), Sells(s,Bananas)
Buy(Drill)
Buy(Milk)
Buy(Bananas)
Have(Drill)
Have(Milk)
Have(Bananas)
Have(Drill), Have(Milk), Have(Bananas), At(Home)
Finish
Causal links (protected)
Have light arrows at every bold arrow.



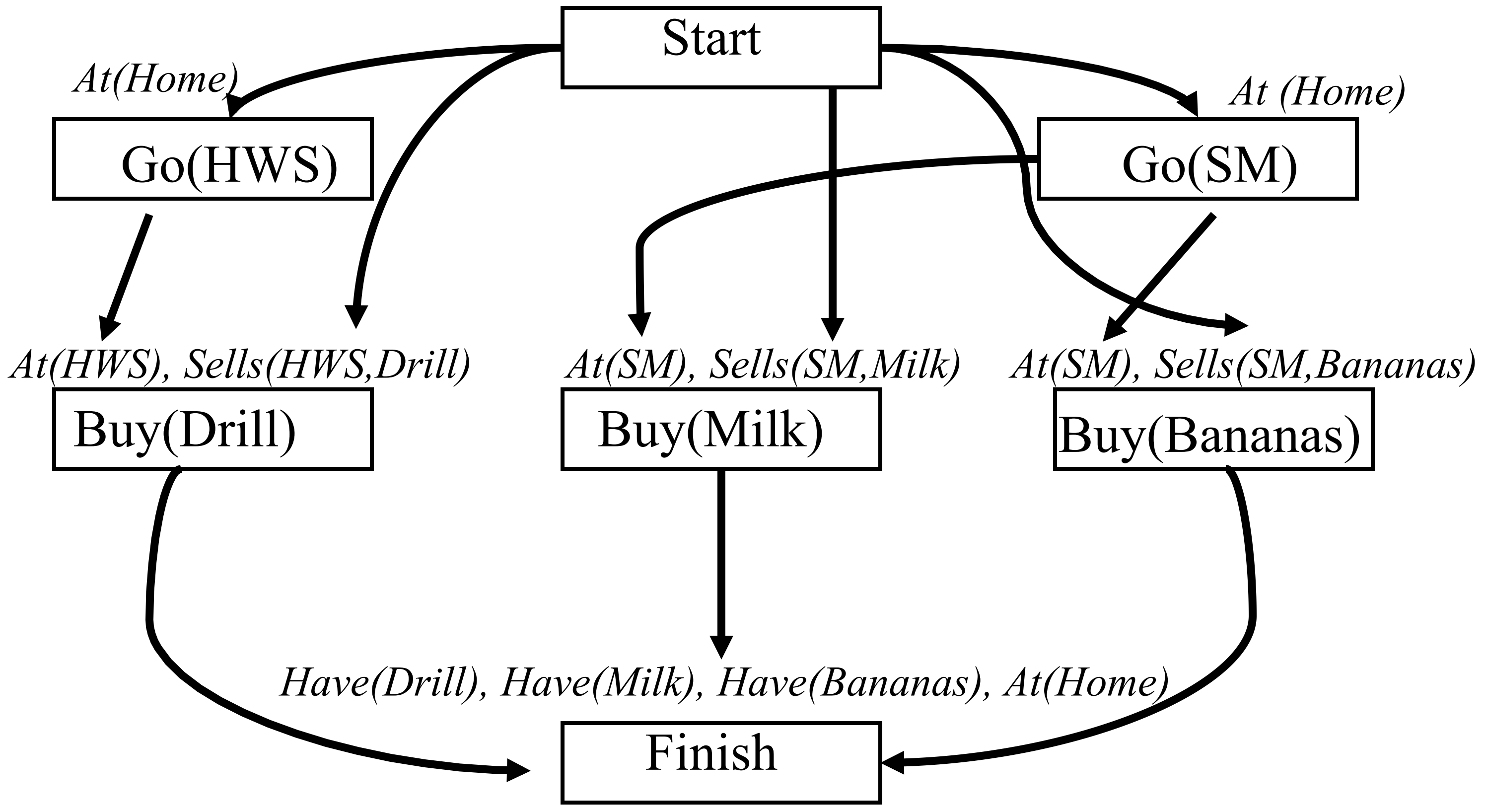
Start
At(HWS), Sells(HWS,Drill)
At(SM), Sells(SM,Milk)
At(SM), Sells(SM,Bananas)
Buy(Drill)
Buy(Milk)
Buy(Bananas)
Have(Drill), Have(Milk), Have(Bananas), At(Home)
Finish

Planning

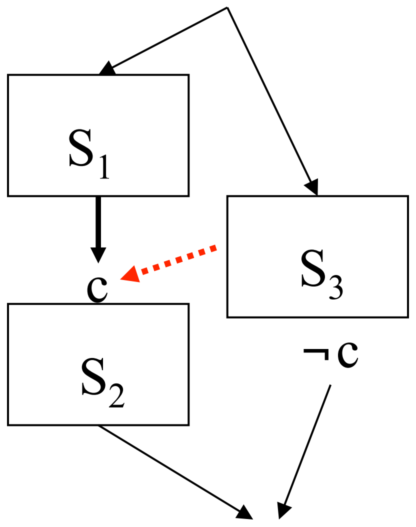

Start
At(x)
At (x)
Go(HWS)
Go(SM)
At(HWS), Sells(HWS,Drill)
At(SM), Sells(SM,Milk)
At(SM), Sells(SM,Bananas)
Buy(Drill)
Buy(Milk)
Buy(Bananas)
Have(Drill), Have(Milk), Have(Bananas), At(Home)
Finish

Planning

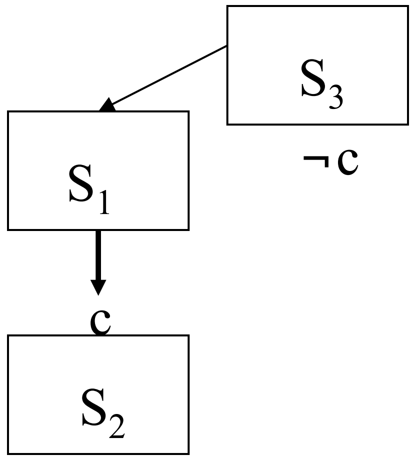
Impasse → must backtrack & make another choice


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

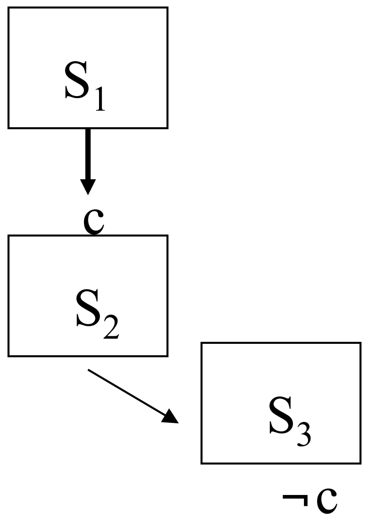
How to identify a dead end?



(a)



(b)
Demotion

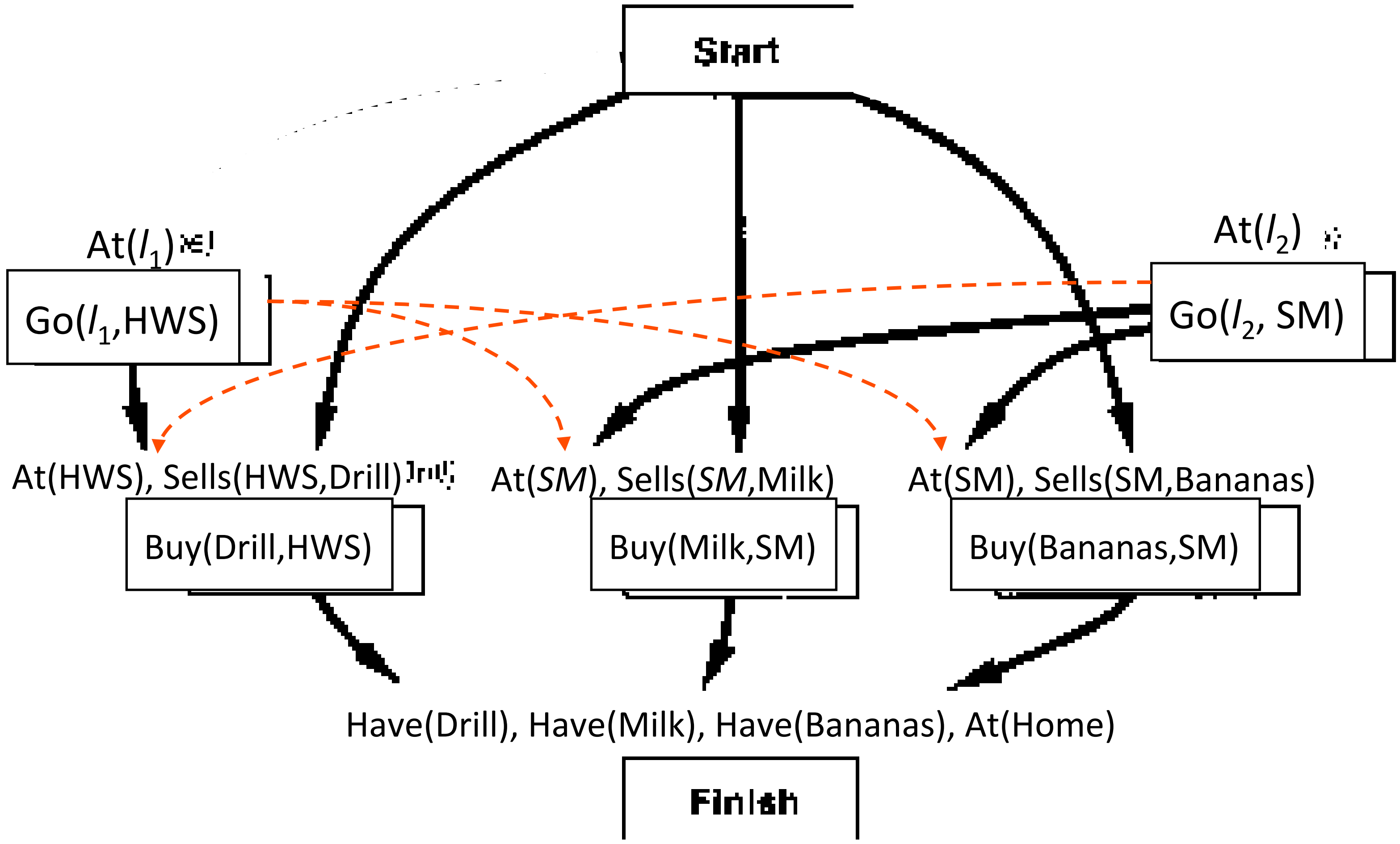


(c)
Promotion

The S_3 action threatens the c precondition of S_2 if S_3 neither precedes nor follows S_2 and S_3 has an effect that negates c.

Resolving a threat

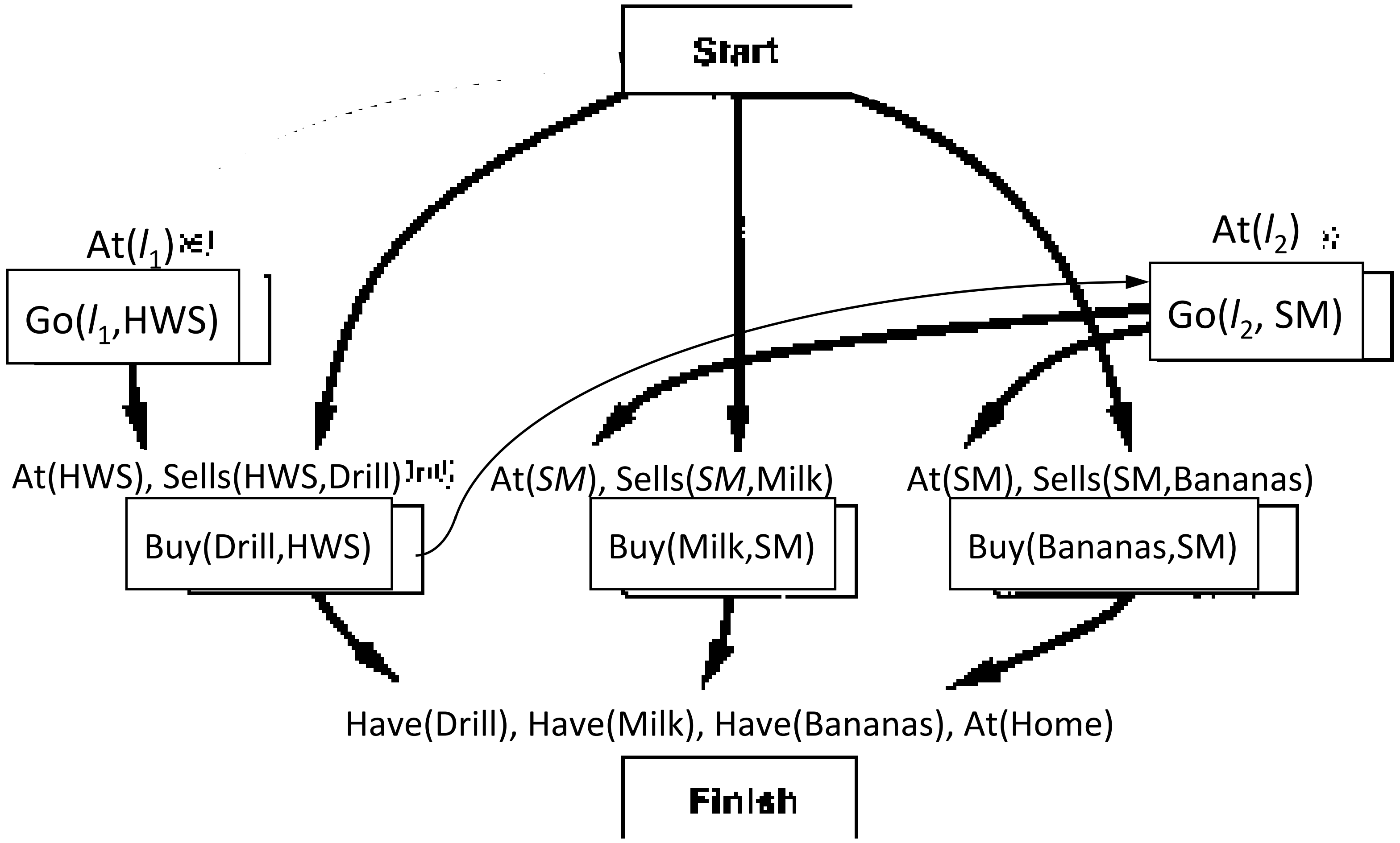
Consider the threats


Start
At(l_1)
Go(l_1,HWS)
At(l_2)
Go(l_2, SM)
At(HWS), Sells(HWS,Drill)
At(*SM*), Sells(*SM*,Milk)
At(SM), Sells(SM,Bananas)
Buy(Drill,HWS)
Buy(Milk,SM)
Buy(Bananas,SM)
Have(Drill), Have(Milk), Have(Bananas), At(Home)
Finish

Resolve a threat

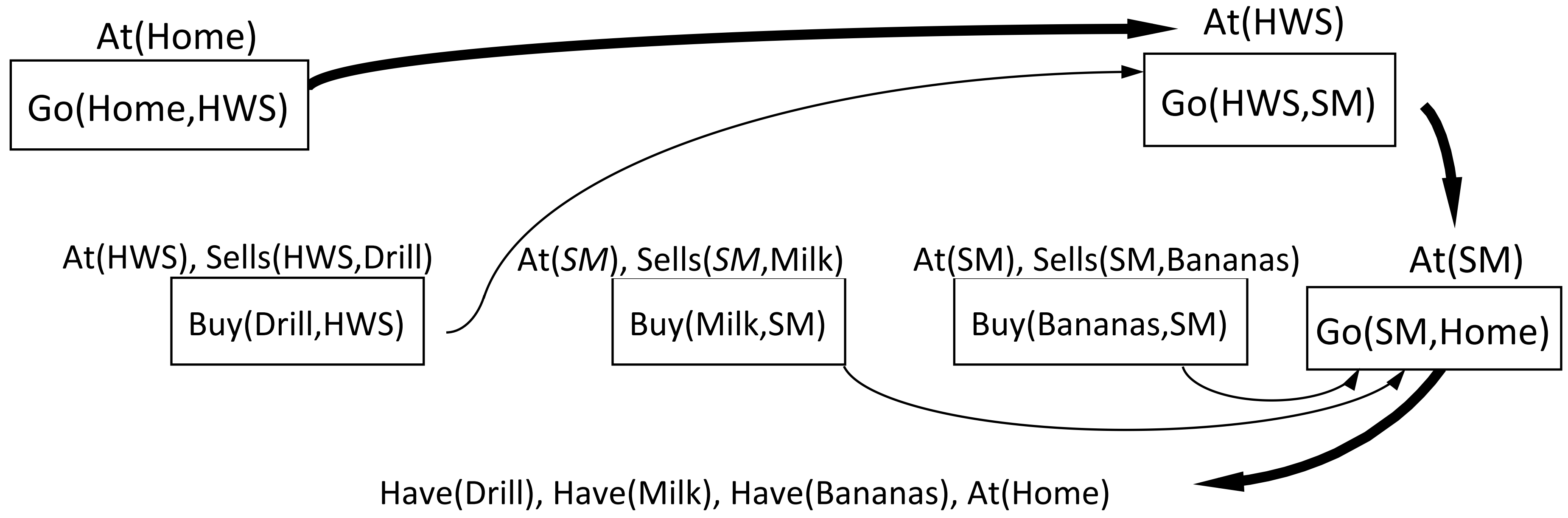
To resolve the third threat, make Buy(Drill) precede Go(SM)

This resolves all three threats

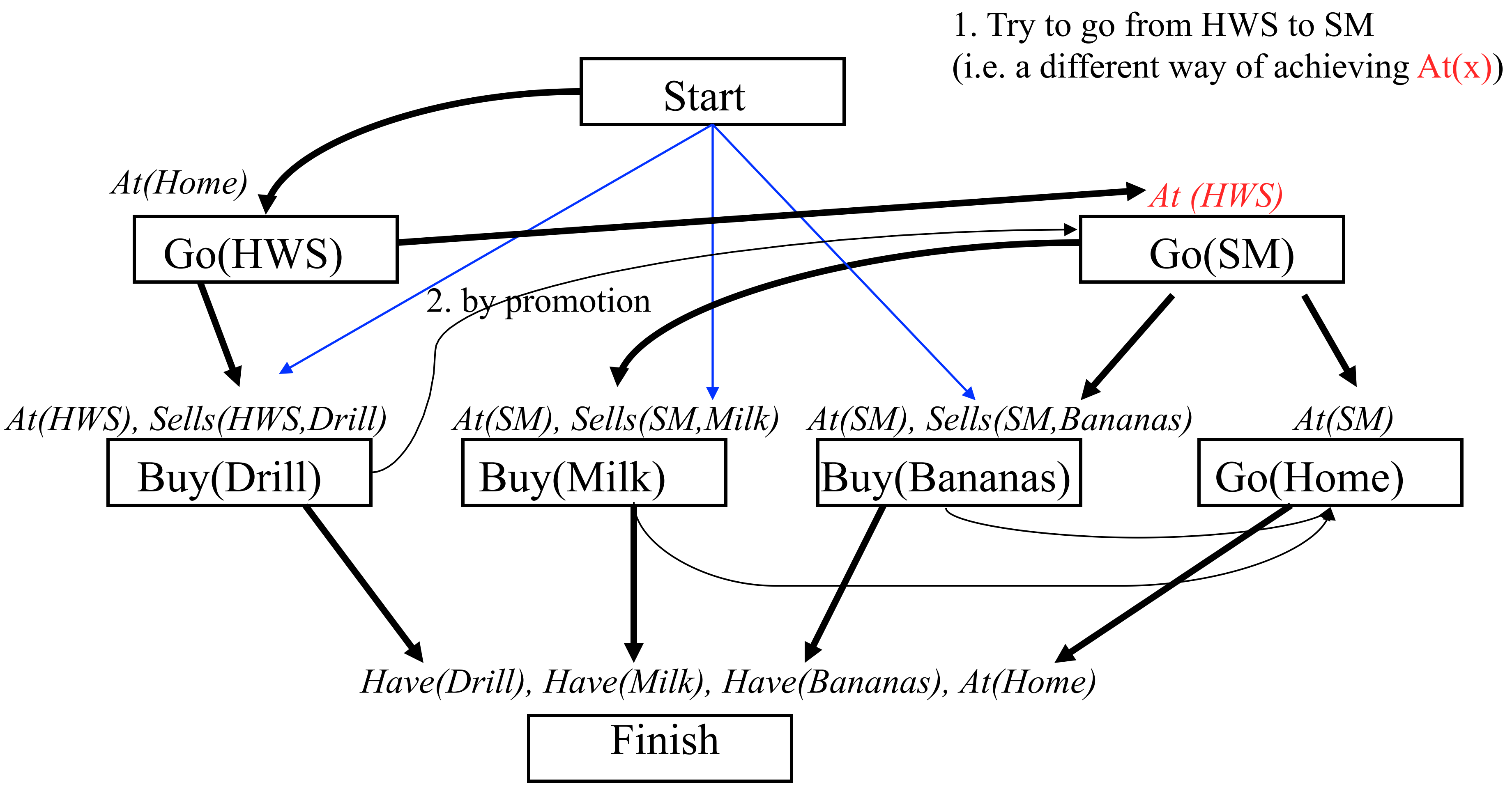

Start
At(l_1)
Go(l_1,HWS)
At(l_2)
Go(l_2, SM)
At(HWS), Sells(HWS,Drill)
At(*SM*), Sells(*SM*,Milk)
At(SM), Sells(SM,Bananas)
Buy(Drill,HWS)
Buy(Milk,SM)
Buy(Bananas,SM)
Have(Drill), Have(Milk), Have(Bananas), At(Home)
Finish

Go Home

Establish At(l_3) with l_3=SM

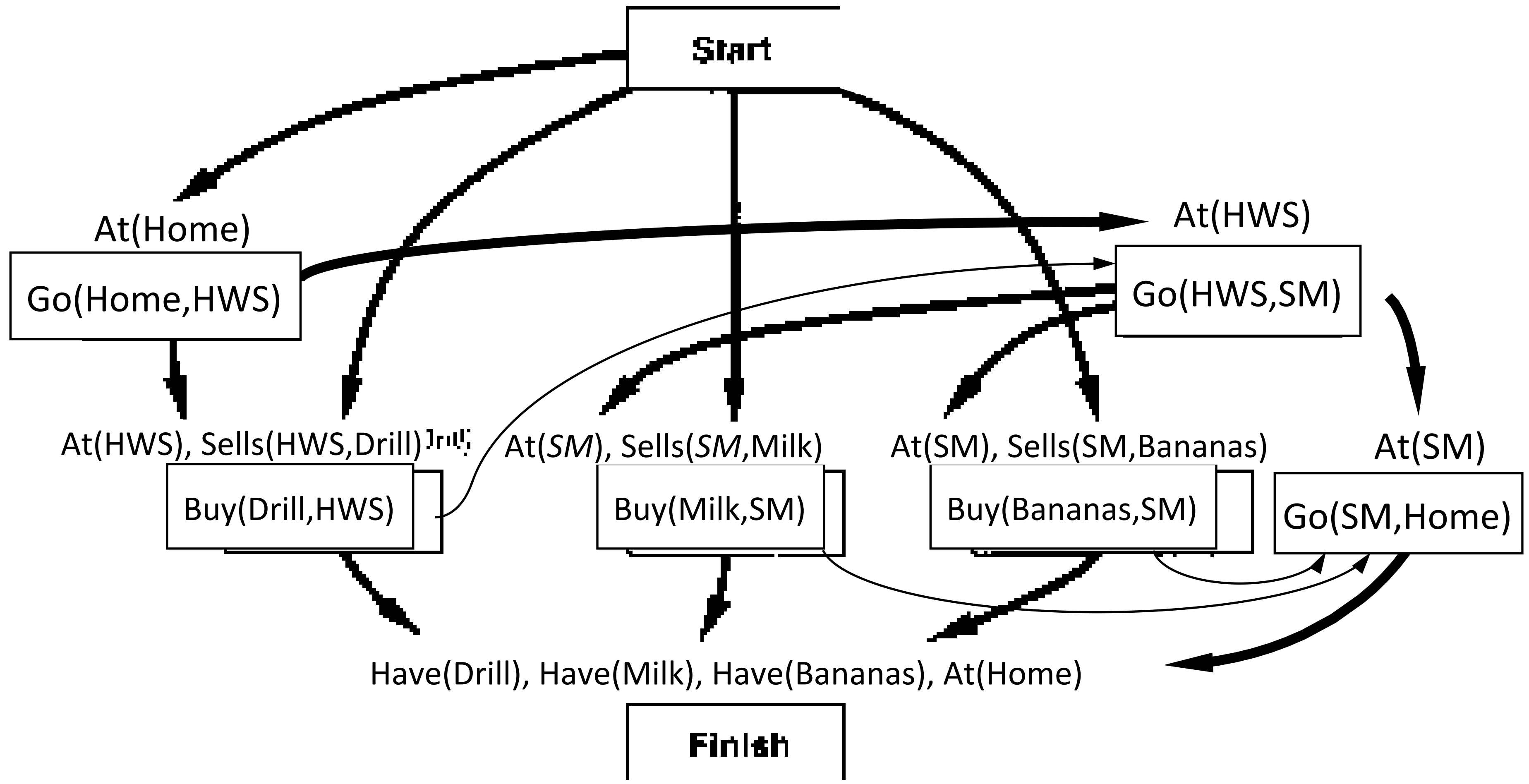

At(Home)
Go(Home,HWS)
At(HWS)
Go(HWS,SM)
At(HWS), Sells(HWS,Drill)
Buy(Drill,HWS)
At(*SM*), Sells(*SM*,Milk)
Buy(Milk,SM)
At(SM), Sells(SM,Bananas)
Buy(Bananas,SM)
At(SM)
Go(SM,Home)
Have(Drill), Have(Milk), Have(Bananas), At(Home)

Planning

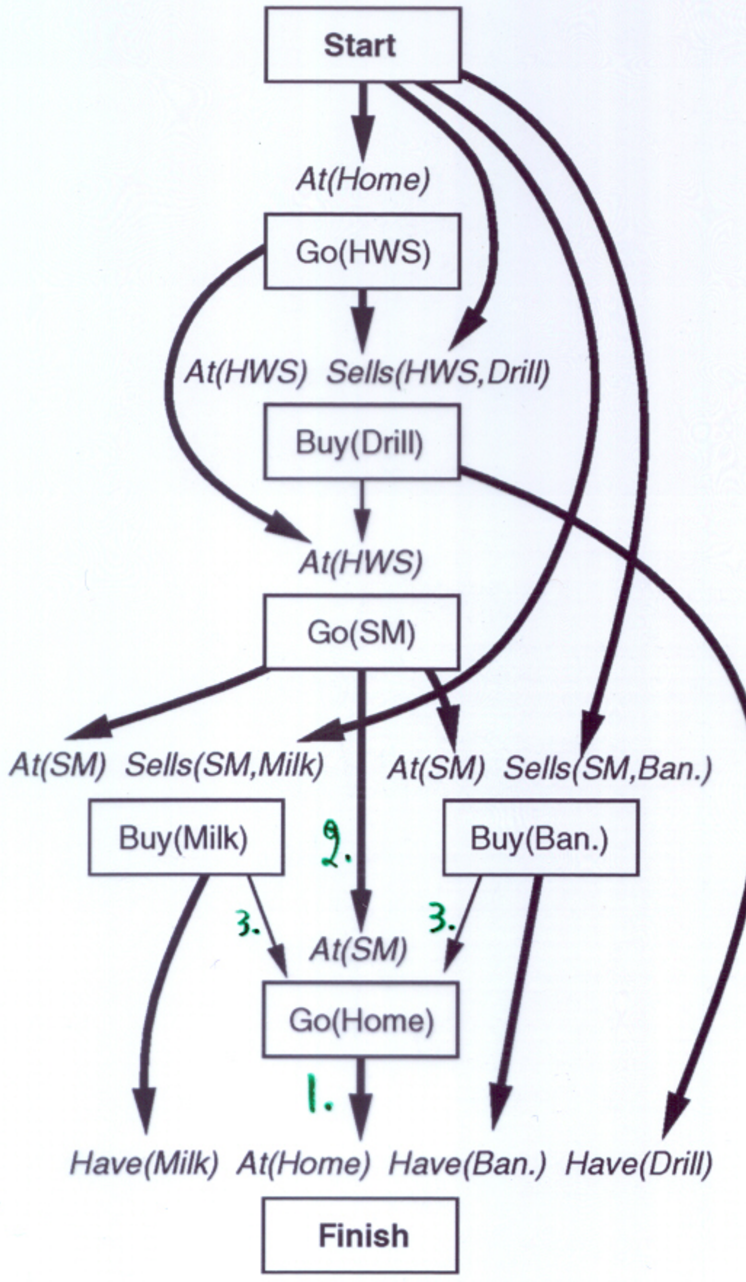


1. Try to go from HWS to SM
(i.e. a different way of achieving At(x))
Start
At(Home)
Go(HWS)
At (HWS)
Go(SM)
2. by promotion
At(HWS), Sells(HWS,Drill)
Buy(Drill)
At(SM), Sells(SM,Milk)
Buy(Milk)
At(SM), Sells(SM,Bananas)
Buy(Bananas)
At(SM)
Go(Home)
Have(Drill), Have(Milk), Have(Bananas), At(Home)
Finish

Final Plan


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

The final plan



If 2 would try At(HWS) or At(Home), threats could not be resolved.

Real-world planning domains

- Real-world domains are complex and don’t satisfy the assumptions of STRIPS or partial-order planning methods
- Some of the characteristics we may need to handle:
  - – Modeling and reasoning about resources
  - – Representing and reasoning about time
  - – Planning at different levels of abstractions

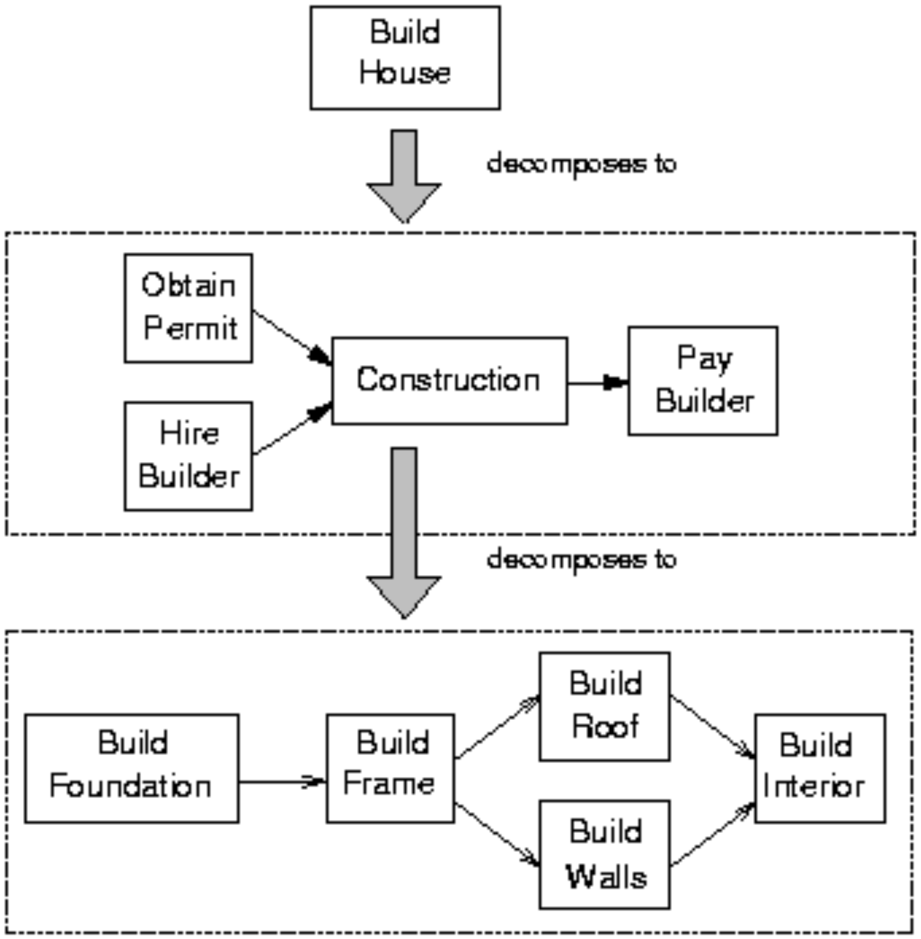
    } Scheduling
  - – Conditional outcomes of actions
  - – Uncertain outcomes of actions

    } Planning under uncertainty
  - – Exogenous events
  - – Incremental plan development } HTN planning
  - – Dynamic real-time re-planning

Hierarchical decomposition

- Hierarchical decomposition, or hierarchical task network (**HTN**) planning, uses **abstract operators** to **incrementally** decompose a planning problem from a **high-level goal** statement to a **primitive plan network**
- **Primitive operators** represent actions that are **executable**, and can appear in the final plan
- **Non-primitive operators** represent **goals** (equivalently, **abstract actions**) that require further decomposition (or *operationalization*) to be executed
- There is no "right" set of primitive actions: One agent's goals are another agent' s actions!

HTN planning: example



HTN operator: Example

```
OPERATOR decompose
PURPOSE: Construction
CONSTRAINTS:
    Length (Frame) <= Length (Foundation),
    Strength (Foundation) > Wt(Frame) + Wt(Roof)
        + Wt(Walls) + Wt(Interior) + Wt(Contents)
PLOT: Build (Foundation)
      Build (Frame)
      PARALLEL
            Build (Roof)
            Build (Walls)
      END PARALLEL
      Build (Interior)
```

HTN operator representation

- Russell & Norvig explicitly represent causal links; these can also be computed dynamically by using a model of preconditions and effects
- Dynamically computing causal links means that actions from one operator can safely be interleaved with other operators, and subactions can safely be removed or replaced during plan repair
- Russell & Norvig's representation only includes variable bindings, but more generally we can introduce a wide array of variable constraints

Truth criterion

- Determining if a formula is true at a particular point in a partially ordered plan is, in general, NP-hard
- Intuition: there are exponentially many ways to **linearize** a partially ordered plan
- Worst case: if there are N actions unordered with respect to each other, there are N! linearizations
- Ensuring soundness of truth criterion requires checking formula under all possible linearizations
- Use heuristic methods instead to make planning feasible
- Check later to ensure no constraints are violated

Truth criterion in HTN planners

- Heuristic: prove that there is *one* possible ordering of the actions that makes the formula true – but don't insert ordering links to enforce that order
- Such a proof is efficient
  - Suppose you have an action A1 with a precondition P
  - Find an action A2 that achieves P (A2 could be initial world state)
  - Make sure there is no action *necessarily* between A2 and A1 that negates P
- Applying this heuristic for all preconditions in the plan can result in infeasible plans

Increasing expressivity

- Conditional effects
  - Instead of having different operators for different conditions, use a single operator with conditional effects
  - Move (block1, from, to) and MoveToTable (block1, from) collapse into one Move (block1, from, to):
    - Op(ACTION: Move(block1, from, to),
      PRECOND: On (block1, from) ^ Clear (block1) ^ Clear (to)
      EFFECT: On (block1, to) ^ Clear (from) ^ ~On(block1, from) ^ ~Clear(to) when to<>Table
    - There's a problem with this operator: can you spot what it is?
- Negated and disjunctive goals
- Universally quantified preconditions and effects

Reasoning about resources

- Introduce numeric variables used as *measures*
- These variables represent resource quantities, and change over the course of the plan
- Certain actions may produce (increase the quantity of) resources
- Other actions may consume (decrease the quantity of) resources
- More generally, may want different resource types
  - Continuous vs. discrete
  - Sharable vs. nonsharable
  - Reusable vs. consumable vs. self-replenishing

Other real-world planning issues

- Conditional planning
- Partial observability
- Information gathering actions
- Execution monitoring and replanning
- Continuous planning
- Multi-agent (cooperative or adversarial) planning

Planning summary

- **Planning representations**
  - Situation calculus
  - STRIPS representation: Preconditions and effects
- **Planning approaches**
  - State-space search (STRIPS, forward chaining, ….)
  - Plan-space search (partial-order planning, HTN, …)
  - *Constraint-based search (GraphPlan, SATplan, …)*
- **Search strategies**
  - Forward planning
  - Goal regression
  - Backward planning
  - Least-commitment
  - Nonlinear planning