Planning

Chapter 11.1-11.3

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

1

Overview

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

2

Planning problem

- Find a **sequence of actions** that achieves a given **goal** when executed from a given **initial world state**. I.e., 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

3

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

4

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

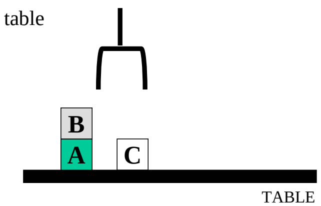
The **blocks world** is a micro-world that consists of a table, a set of blocks and a robot hand.

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)



This is meant to be a very simple model!

Major approaches

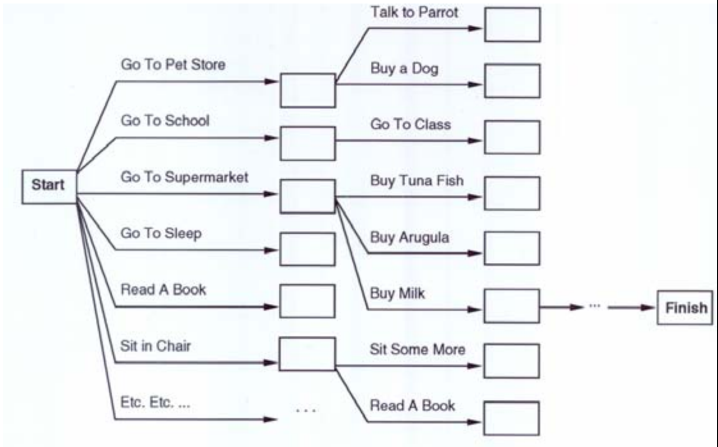
- Planning as search?
- GPS / STRIPS
- **Situation calculus**
- **Partial order planning**
- Hierarchical decomposition (HTN planning)
- **Planning with constraints (SATplan, Graphplan)**
- Reactive planning

Planning as Search?

- **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

  These are black boxes; we can't look inside to select actions that might be useful
- **Plan representation:** an 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.”



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 a best action to do next
  - A table of differences identifies 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
- Introduced a 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 situation characterizing the world state, will lead to a desired result
- This is how the “neats” approach the problem

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))

  ∧ Have(Milk, Result'(p,S_0))

  ∧ Have(Bananas, Result'(p,S_0))

  ∧ 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)]

13

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!

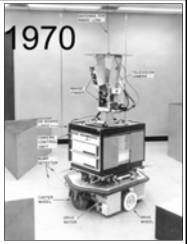
14

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
- Since planning is a ubiquitous task for an intelligent agent, it's reasonable to develop a special purpose subsystem for it.

15

Strips planning representation



Shakey the robot

- 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) ...

- 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

16

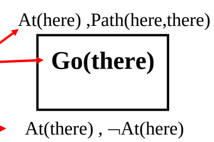
Operator/action representation

- 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

- 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 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]).

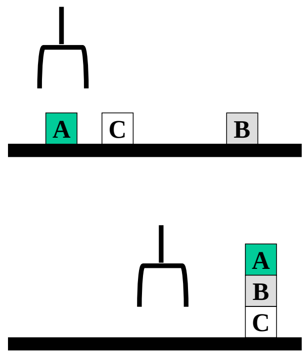
STRIPS planning

- 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 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

```
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)])
```

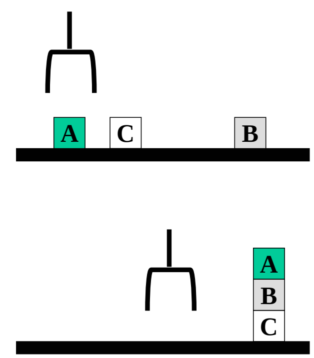

STRIPS

```
% 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)

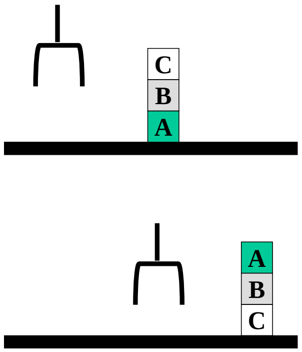


A plan:
- pickup(a)
- stack(a,b)
- unstack(a,b)
- putdown(a)
- pickup(b)
- stack(b,c)
- pickup(a)
- stack(a,b)

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)

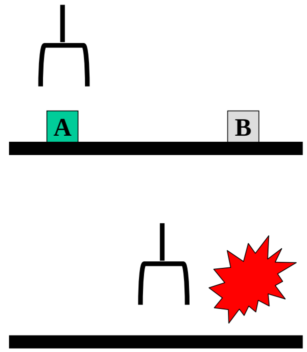


Plan:
- unstack(c,b)
- putdown(c)
- unstack(b,a)
- putdown(b)
- pickup(b)
- stack(b,a)
- unstack(b,a)
- putdown(b)
- pickup(a)
- stack(a,b)
- unstack(a,b)
- putdown(a)
- pickup(b)
- stack(b,c)
- pickup(a)
- stack(a,b)

Yet Another BW planning problem

Initial state:
- ontable(a)
- ontable(b)
- clear(a)
- clear(b)
- handempty

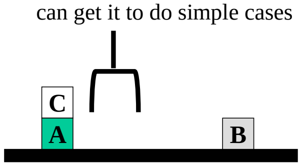
Goal:
- on(a,b)
- on(b,a)



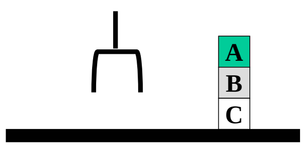
Plan:
??

Goal interaction

- Simple planning algorithms assume that 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 (via unstack(C,A), stack(A,B)) is undone when solving the second 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 could not handle this, although minor modifications can get it to do simple cases



Initial state

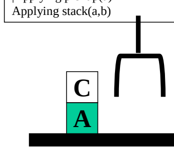


Goal state

Sussman Anomaly

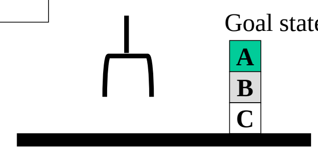
```
Achieve on(a,b) via stack(a,b) with preconds: [holding(a),clear(b)]
|Achieve holding(a) via pickup(a) with preconds: [ontable(a),clear(a),handempty]
||Achieve clear(a) via unstack(_1584,a) with preconds:
[on(_1584,a),clear(_1584),handempty]
||Applying unstack(c,a)
||Achieve handempty via putdown(_2691)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



Sussman Anomaly

- Classic Strips assumed that once a goal had been satisfied it would stay satisfied.
- Our 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? -- a notion of "protecting" a subgoal so that it isn't undone by some later step.

29

State-space planning

- STRIPS searches thru 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
  - Usually results in a high branching factor
- A **regression planner** searches backward from the goal
  - OK if operators have enough information to go both ways
  - Ideally this leads to reduced branching –you are only considering things that are relevant to the goal
  - Handling a conjunction of goals is difficult (e.g., STRIPS)

30

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 …]

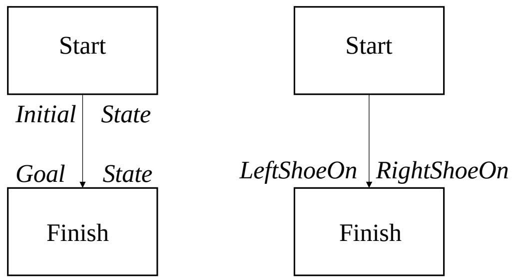
31

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 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

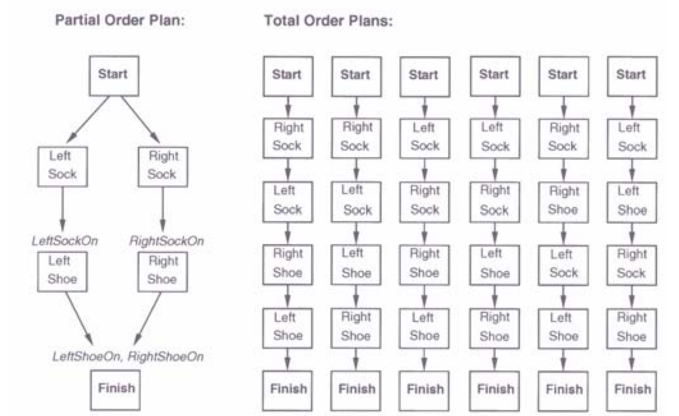
32

A simple graphical notation



(a) (b)

Partial Order Plan vs. Total Order Plan



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

Least commitment

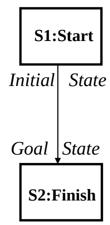
- Non-linear planners embody the principle of **least commitment**
  - only choose actions, orderings, and variable bindings absolutely necessary, leaving other decisions till later
  - avoids early commitment to decisions that don't really matter
- A linear planner always chooses to add a plan step 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** $\{S_1, S_2, S_3, S_4 \ldots\}$
    - Steps have operator descriptions, preconditions and post-conditions
  - (2) A set of **causal links** $\{ \ldots (S_i,C,S_j) \ldots\}$
    - Meaning: 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) (makes C false), then (3) contains either $S_k<S_i$ or $S_j<S_k$

The initial plan

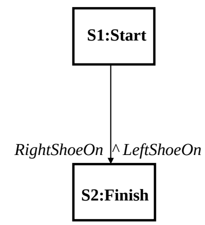
Every plan starts the same way



Trivial example

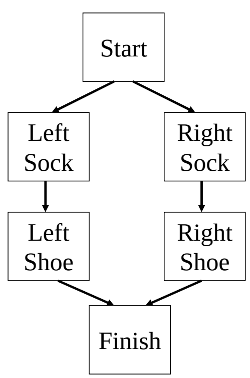
Operators:

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

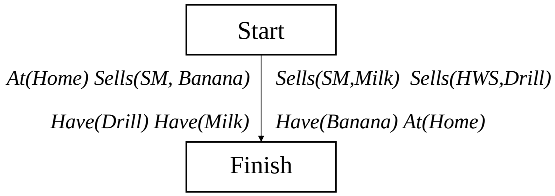


POP constraints and search heuristics

- 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 $S_1 \rightarrow S_2$ that **protect** condition *c*:
  - Never add an intervening step S_3 that violates *c*
  - If a parallel action **threatens** *c* (i.e., has 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**)

Partial-order planning example

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

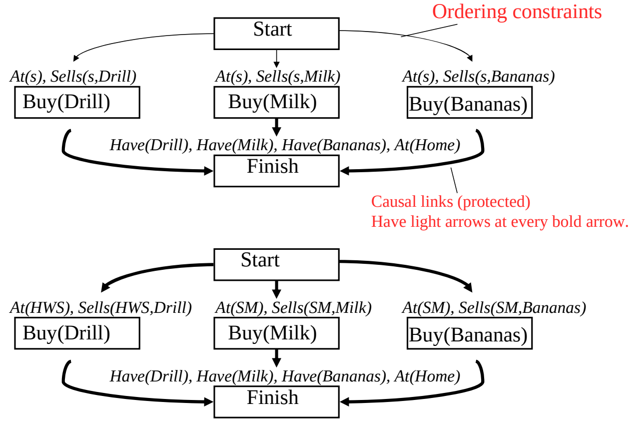


41

```
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 --c--> S_need to LINKS(plan)
  add the ordering constraint S_add ≺ S_need to ORDERINGS(plan)
  if S_add is a newly added step from operators then
     add S_add to STEPS(plan)
     add Start ≺ S_add ≺ Finish to ORDERINGS(plan)
-----------------------------------------------------------------
procedure RESOLVE-THREATS(plan)

  for each S_threat that threatens a link S_i --c--> S_j in LINKS(plan) do
     choose either
        Promotion: Add S_threat ≺ S_i to ORDERINGS(plan)
        Demotion: Add S_j ≺ S_threat to ORDERINGS(plan)
     if not CONSISTENT(plan) then fail
  end
```

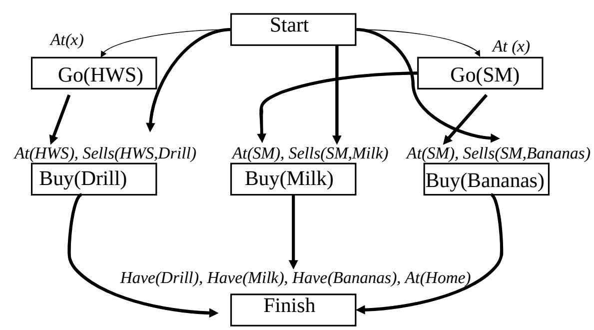
42

Planning



43

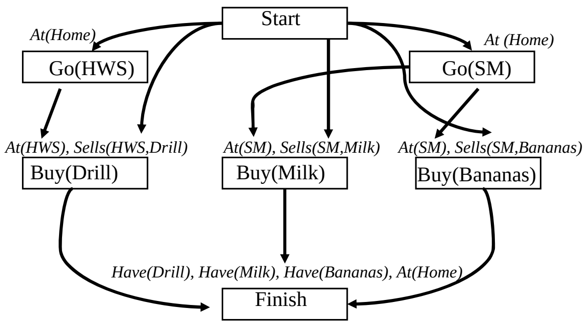
Planning



44

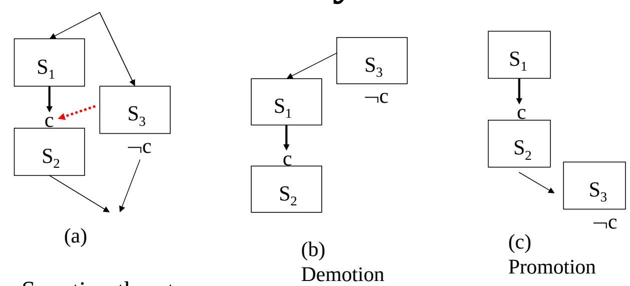
Planning

Impasse → must backtrack & make another choice



45

How to identify a dead end?

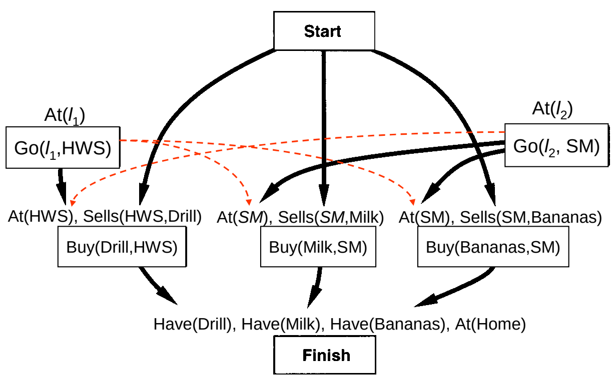


The S_3 action threatens the c precondition of S_2 if S_3 neither precedes nor follows S_2 and S_3 has an effect that negates c.

Resolving a threat

46

Consider the threats

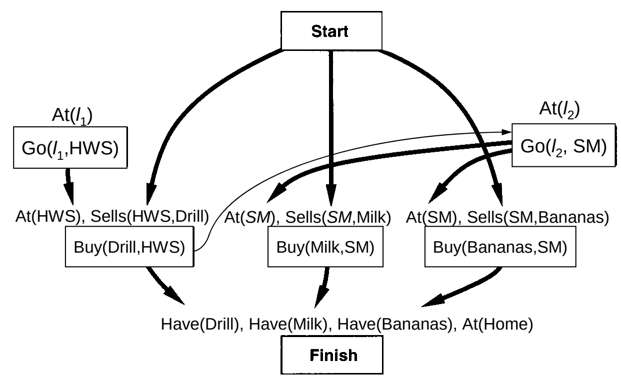


47

Resolve a threat

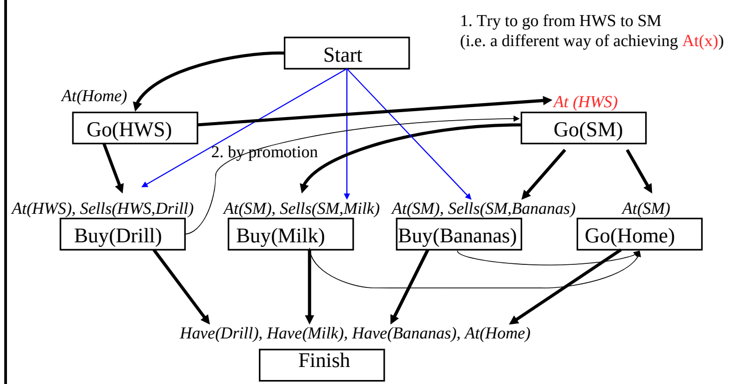
To resolve the third threat, make Buy(Drill) precede Go(SM)

This resolves all three threats

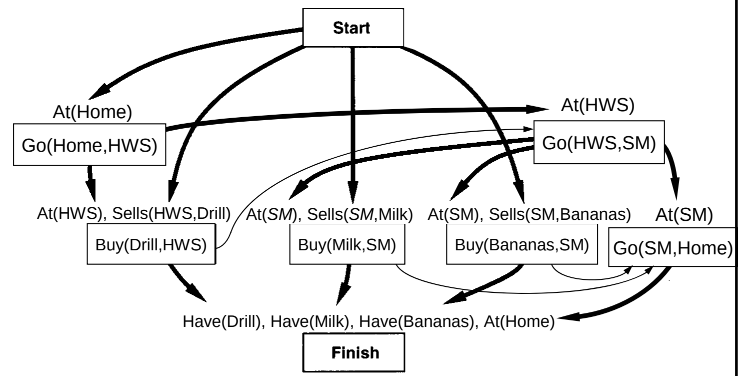


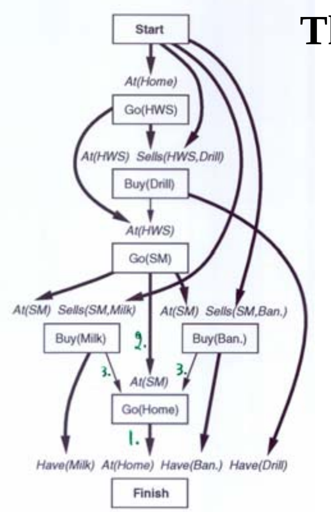
48

Planning



Final Plan





The final plan

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