Partial-Order Planning

31 December 2020 18:53

Quick Recap

You should understand:

1. The idea of an agent that can reason
2. The idea of a KB and an agent performing inference
3. How we adapt that into something that makes plans
4. How plans can do things differently to search
5. Why we might prefer planning over search
6. What the issues are with planning
7. Situation calculus and what it's used for
8. STRIPS and what it can do

Closed-world assumption - most planners assume that if state descriptions do not mention a positive literal, we can assume it to be false - this can be dangerous
Goals - conjunctions of literals, may contain variables
Planner - a system you can ask for a sequence of actions that make the goal true if executed

Operators comprise three components

- **action** e.g. buy(x) - buy an x
- **precondition** e.g. At(p), Sells(p, x) - we're at p and p sells x
- **effect** e.g. Have(x) - thus we now have x

The goal is to gradually move from incomplete and vague plan to complete and correct plans

Key Terminology

- **Partial plan** – an incomplete plan that we consider ways of expanding, until we come up with a plan that solves the problem
- **Operator schema** – an operator that takes variables – basically a function.
- **Least commitment** – one should only make choices about things that you currently care about, leaving other choices to be worked out later
- **Partial order** – A planner that can represent plans in which some steps are ordered with respect to each other, and some are unordered, is called a partial order planner.
- **Total order** – a plan that has all the steps in some order – there is no ambiguity about when the steps will take place.
- **Linearisation** – a totally ordered plan that is derived from a partially ordered plan is called a linearisation of the plan.
- **Fully instantiated plans** – a plan in which every variable is bound to a constant
- **Causal links** – a causal link is written as "Si achieves c for Sj". They serve to record the purpose of steps in the plan – here the purpose of Si is to achieve the precondition c of Sj

Plan

A plan is formally a data structure consisting of;

1. A set of plan steps – each step is an operator to the problem
2. A set of step ordering constraints – each ordering constraint is in the form Step I before Step j, meaning that step I has to take place before step j, although not necessarily directly before.
3. A set of variable binding constraints – each variable constraint is of the form v = x where v is a variable in some step, and x is either a constant or another variable
4. A set of causal links – as described above

Outline of POP

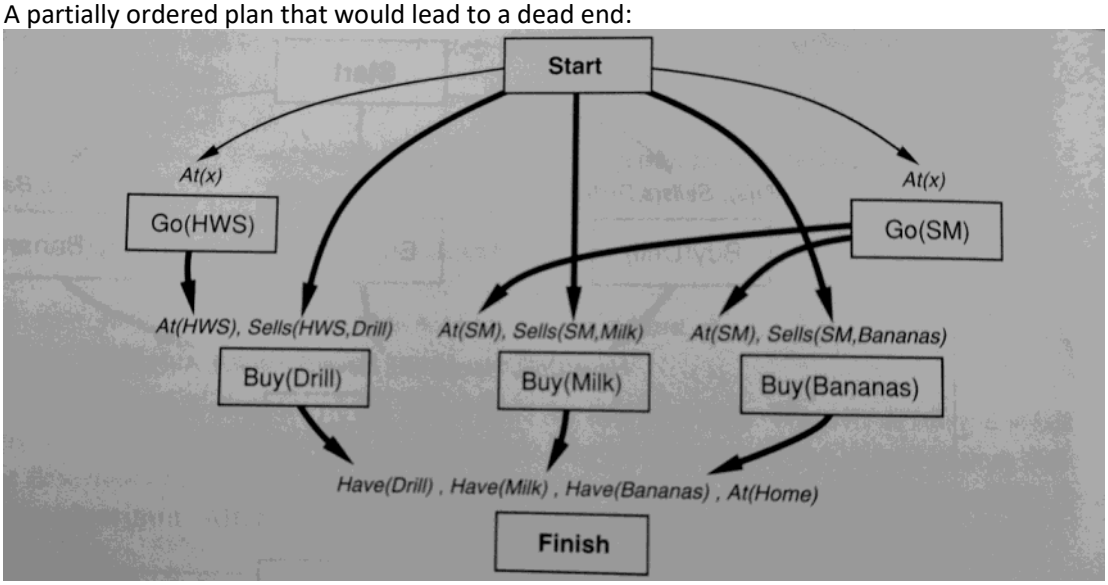
We sketch an outline for a partial-order regression planner that searches through plan space.

The planner starts with an initial plan representing the start and finish steps, and on each iteration, adds one more step. If this leads to an inconsistent plan, it backtracks and tries another branch of the search space

To keep the search focused, the planner only considers adding steps that serve to achieve a precondition that has not yet been achieved.

If you have two conditions that contradict each other – e.g. go(HardwareStore) and go(superMarket) which both require at(Home), then you reach a dead end. There is no way to go(superMarket) if we're at(hardwareStore) if the precondition for go(superMarket) is at(home). Thus, we have a flawed plan.

A partially ordered plan that would lead to a dead end:



Interestingly, the planner could notice this partial plan is flawed without wasting a lot of time. The key is that the causal links in a partial plan are protected links. A causal link is protected by ensuring that threats (steps that might delete or clobber the protected condition) are ordered to come before or after the new step.

If we place the clobbering step that threatens a condition prior to the step, then we call it a demotion
If we place the step that threatens the condition c after the causal link that protects that condition, we call it a promotion

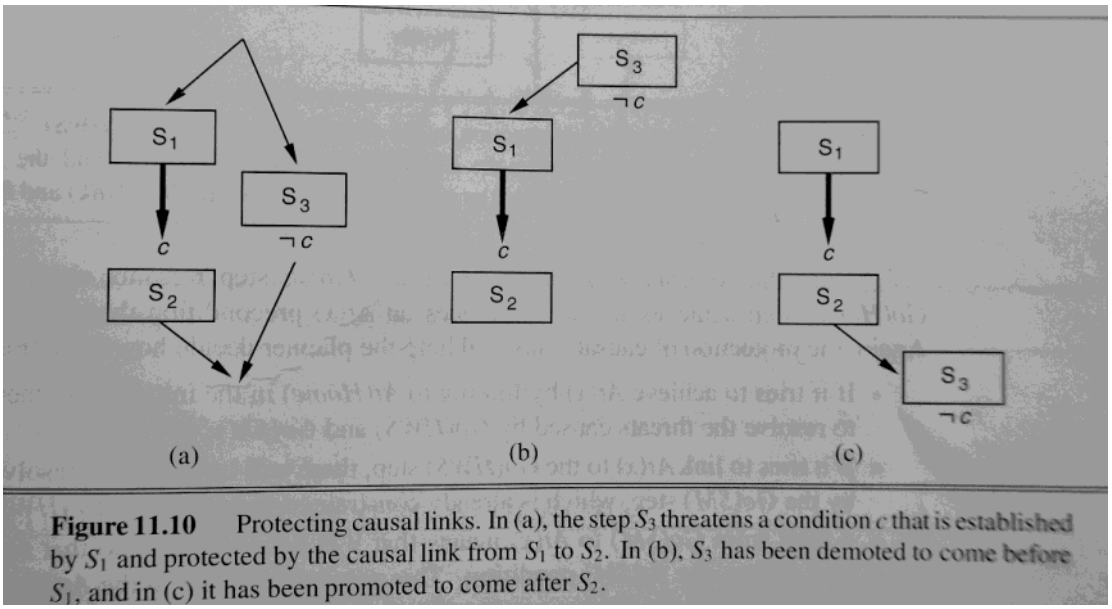


Figure 11.10 Protecting causal links. In (a), the step S_3 threatens a condition c that is established by S_1 and protected by the causal link from S_1 to S_2. In (b), S_3 has been demoted to come before S_1, and in (c) it has been promoted to come after S_2.

Here we have an example of causal link protection in a shopping plan:

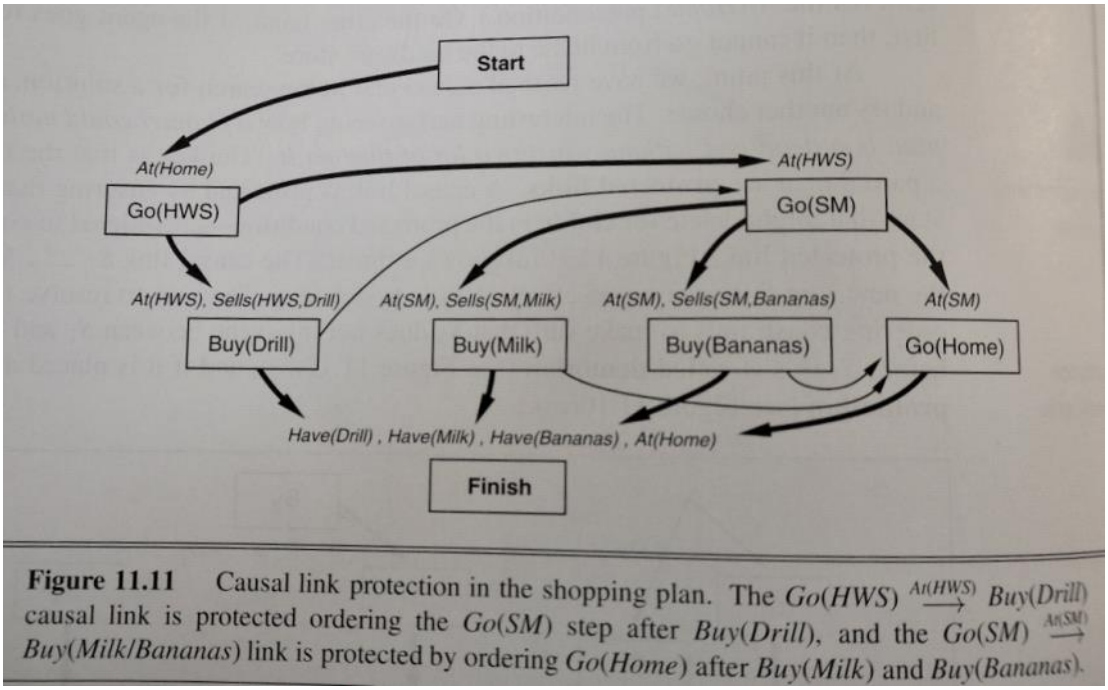
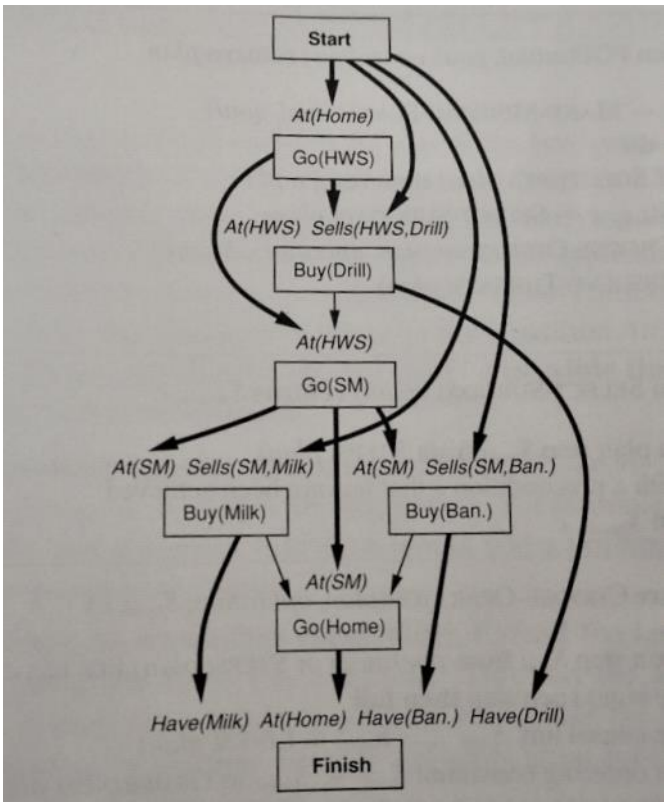


Figure 11.11 Causal link protection in the shopping plan. The $Go(HWS) \xrightarrow{At(HWS)} Buy(Drill)$ causal link is protected ordering the $Go(SM)$ step after $Buy(Drill)$, and the $Go(SM) \xrightarrow{At(SM)} Buy(Milk/Bananas)$ link is protected by ordering $Go(Home)$ after $Buy(Milk)$ and $Buy(Bananas)$.

This is what a finished solution to the shopping problem might look like. Note that the result is almost a totally ordered plan, the only ambiguity being that Buy(Milk) and Buy(bananas) can come in any order.



Our partial order planner can now take a problem that would require thousands of search states for a problem-solving approach, and solve it with only a few search states. Moreover, the least commitment nature of the planner means it only needs to search at all in the places where subplans interact with each other. Finally, the causal links allow the planner to recognise when to abandon a doomed plan without wasting a lot of time expanding irrelevant parts of the plan.

POP Algorithm

```
function POP(initial,goal,operators)
    returns a 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 plan step and precondition (S_need, c)
  pick step S_need from STEPS(plan)
    with precondition c that has not been achieved
  return S_need, c
```

```
function CHOOSE-OPERATOR(plan,operators, S_need,c)
  pick step S_add from operators or STEPS(plan)
      that has effect c
  if no such step then fail
  add causal link S_add --c--> S_need to LINKS(plan)
  add ordering constraint S_add ≺ S_need to ORDERINGS(plan)
  if S_add 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
      Demotion: Add S_threat ≺ S_i to ORDERINGS(plan)
      Promotion: Add S_j ≺ S_threat to ORDERINGS(plan)
  if not CONSISTENT(plan) then fail
```

POP starts with a minimal partial plan, and at each step extends the plan by achieving a precondition c of a step S_need.

It does this by *choosing some operator*, either from the existing steps of the plan or form the pool of operators, that achieves the precondition. *It records the causal link for the newly achieved precondition, and then resolves any threats to causal links.*

The new step may *threaten an existing causal link* or an existing step may threaten the new causal link. If at any point the algorithm fails to find a relevant operator or resolve a threat, it backtracks to a previous choice point. An important subtlety is that the selection of a step and precondition in select-subgoal is not a candidate for backtracking.

The reason is that every precondition needs to be considered eventually, and the handling of preconditions is commutative: Handling c1 and then c2 leads to exactly the same set of possible plans as handling c2 and then c1. So we can just pick a precondition and move ahead without worrying about backtracking. The pick we make effects only the speed, and not the possibility of finding a solution.

Notice that we start at the goal and move backwards until we have a solution. Thus, POP is a sound and complete regression planner. Every plan it returns is a solution.

Problems with POP and STRIPS

STRIPS cannot express:

Hierarchical plans

- we often want to specify plans at different levels of detail.
- We want to have several levels before reaching executable actions.
- This makes computation manageable and resulting plan understandable – analogous to high level and low level machine code.
- This allows human specification of abstract partial plans to guide the planner because we often want to give the planner guidance.

Complex conditions

- Operations have variables, but we have no quantification
- STRIPS use of variables is limited – we cannot express that pick(bag) also causes all objects in the bag to be lifted
- Operators are unconditional - we cannot express actions having different effects according to conditions

Solution – conditional effects: avoids premature commitment – have effect when condition

Time

- In situation calculus, time is discrete, and actions occur instantaneously
- We need to represent that actions take time, may only be applicable at certain times, and that the goal may have a deadline

Resources

- Real problems have limited resources – financial time, quantity of materials, machinery available
- Actions have a cost that we need a way to represent
- Actions descriptions need to represent the requirements of performing the action
- Planner needs to take cost into consideration

Solution: extend STRIPS and modify POP accordingly

1. **Hierarchical decomposition** can be added to STRIPS in the form of non-primitive operators. We can then modify the planner to replace non-primitives with decomposition. We do this by adding abstract operators
2. **Abstract operators** that can be decomposed into steps
3. Decompositions are predetermined and stored in a library of plans – works best when there are several possible decompositions

Broadening Operator Descriptions

We want to make operator descriptions more expressive to widen the applicability of POP

1. Conditional effects – avoids premature commitment - e.g. have effect WHEN
2. Allow negated goals – ability to call CHOOSE-OPERATOR with !p instead of p
3. Disjunctive preconditions – allow SELECT-SUBGOAL to make a nondeterministic choice between disjuncts - use principle of least commitment
4. Disjunctive effects – introduces nondeterminism – may be able to address with coercion
5. Universally quantified preconditions – instead of clear(b) we can do "for all blocks, remove blocks from b
6. Resource constraints – add numeric-value measures such as money(200) or fuel(20) - a measure fluent
7. Temporal constraints – we can treat time as a resource – remember we can do things concurrently so time is the max of the jobs not the sum

With resource constraints, we want to plan for scarce resources first, delaying choice of causal links where possible. We can check for failure then without a finished plan, so we don't have to try all operators.

Conditional Planning

So far we have assumed that the world is fully observable, static and deterministic – we also assumed that action descriptions are correct and complete. Unfortunately, the real world is not like this – typically have to deal with both incomplete and incorrect information.

In the real world we have incomplete information – we don't know the preconditions of an operator, and we don't know the effects of an operator e.g. inflate(tire) might cause inflated(tire), slowhiss(tire), burst(tire), breakPump(tire) etc

We can also have incorrect information. We might think we have a spare tire, then find we don't.

We can never finish listing all the required preconditions and possible conditional outcomes of actions in the real world

Conditional Planning

- Plan to obtain information
- Subplan for each contingency that could happen
- Expensive because it plans for many unlikely cases

Replanning

Action Monitoring:

- Preconditions of next action note met

Plan monitoring:

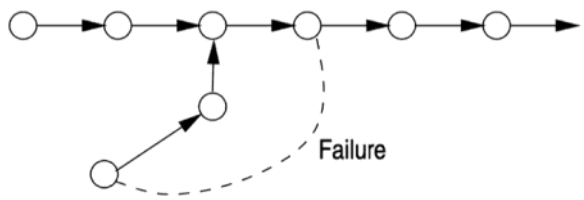
- Effects of an action might not be as predicted
- Failure = preconditions of remaining plan are not met
- Preconditions = causal links at current time
- Check current preconditions with perceived state

In each case, we fail and need to replan.

There are 2 types of plan failure –

1. **bounded indeterminacy**: when unexpected effects of actions can be enumerated – we can use conditional planning to deal with this
2. **Unbounded indeterminacy**: set of possible outcomes is too large to enumerate – we can plan for (at most) a limited number of contingencies and must replace when things go wrong – e.g. driving

Replanning



A naive approach would be to replan from scratch, but this is inefficient and we might get stuck in a cycle of a bad plan

A better approach would be to plan to get back on track by connecting the old plan as show above
Ideally we introduce learning so the agent doesn't loop forever

Alternative Solutions to Replanning

Coercion – force the world into a particular state to reduce uncertainty

Abstraction – ignore details of a problem that may be unknown
Aggregation – Treat a large number of objects which have individual uncertainty as a single, aggregate practicable object

Reactive Planning

Abandon domain-independent planning and use domain specific knowledge - the agent has procedural knowledge – library of partial plans representing a collection of behaviours