Today's lecture: Planning

Last time we talked about logic. It's important since it is used for a special type of search problems called **planning**.

Planning: a set of actions, each has pre-conditions. We want to choose the set of actions whose ultimate effect achieves a certain goal.

The path planning problem for instance, which we solved with search/A*, can be solved with a plan: we can encode the actions that move from one node to the next. The precondition is that we're at the start node, and the effect is that we're in the next city.

In Wumpus world we have different types of actions: we can pick up gold, fire an arrow etc.

The way to encode pre and post conditions can affect how complex of a problem can be solved using planning. In the Wumpus word, for instance:

- Pre-condition: we're at square [1,1]
- We can ask: $ASK\left(KB, \exists s\ Holding(Gold, s)\right)$ – in what situation will I be holding the gold?

The result: $PlanResult(p, s)$ is the result of executing p in s. The solution is a sequence of action. Planning system are designed to do that efficiently.

Today's topics

- Search vs. planning
- STRIPS – representation language for representing plans.
- Partial order planning – POP

Search vs. planning

Consider the task: get milk, banana and cordless drill.

If we try to solve this as a search problem, the branching factor is huge. In the start state we may be at home, and we need to go out and buy those things. Every potential destination is a branch and eventually there will be very little goal leafs. Therefore using search is infeasible.

<u>Planning as search</u>

$PlanResult(p, s)$ is the situation resulting from executing p in s:

$PlanResult([], s) = s$

$PlanResult(a[p], s) = PlanResult\left(p, Result(a, s)\right)$

- Initial state: $At(home, s_0) \land \neg Have(milk, s_0) \land \ldots$
- Actions: detailed in the slides.
- Query: $s = PlanResult(p, s_0) \land at(home, s) \land have(milk, s) \land \ldots$
- Solution: $p = [go(supermarket), buy(milk), buy(bananas), go(hardware\ store), \ldots]$

Problems: Hard to apply heuristics, unconstrained branching.

Problems with search

- Overwhelmed by irrelevant actions
- Can't count on user to supply heuristics, hard to find good heuristics
- All comes down to representation

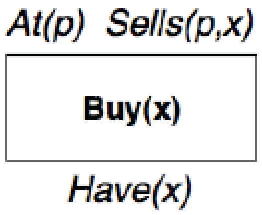
Planning representations (STRIPS)

- Use logic to make problems tractable
- States: We represent states as a list of all propositions we know are true or false at that time
- Goals: some specified states (e.g. $rich \wedge famous$) – states where these are satisfied
- Action schema:
  - Action name and parameter, e.g. $fly(p, from, to)$
  - Preconditions: what must be true before the action is executed, e.g. $at(p, from)$
  - Effect: how state changes when action is executed, e.g. $at(p, to) \wedge \neg at(p, from)$

STRIPS example

- Action: $buy(x)$
- Precondition: $at(p) \wedge sells(p, x) \wedge \neg have(x)$
- Effect: $have(x)$

Example representation:



Preconditions above the box, action in the box, effect under the box.

STRIPS exercise

Initial state description:

$$at(monkey, b) \wedge at(banana, b) \wedge at(box, c) \wedge height(monkey, low) \wedge height(box, low) \wedge$$
$$height(banana, high) \wedge (pushable, box) \wedge climable(box)$$

The complete answer is in the slides.

In classical STRIPS representation there's no quantification, so the goal state example in the solution slide is for advanced planning representation.

State space vs. plan space

In search, the node includes the state and the history that lead to that state. E.g. in 8-puzzle it represents the moves that got you to this state.

Open condition: a pre-condition of a step not yet fulfilled.

Unlike standard search where we have a sense of a feasible plan that may not achieve our goal, in a planning space we have a vague plan that is goal-oriented. In the planning space we may know how to get from x to y and from y z to w, and the vagueness is getting from y to z.

Partially ordered plans (POP)

Consider putting on shoes, which requires putting on left and right sock and left and right shoe.

There are many optimal plans that achieve the goal, lots of optimal combinations – in a search space the branches will explode. In a planning space, this is avoided and results with greater efficiency.

- **Consistent plan**: plan with no cycles in ordering constraints and no conflict with casual links. For instance, an "8" cycle.
- **Solution**: consistent plan with no open preconditions.

<u>POP as search</u>

- Initially the plan contains start and finish states, all preconditions in finish are open preconditions.
- Successor function picks op[en precondition p on an action B and generates a successor plan for every consistent way to choose action A that achieves p
    - Causal link $a \to B, p$ and ordering constraint $A < B$ are added to the plan
    - Resolve conflicts between new link and existing actions and new action A and all causal links. Resolutions can be applied by adding ordering constraints.
- Goal test checks if solution (no open preconditions)

The shoes example:

- Preconditions for FINISH: $on(left - shoe) \wedge on(right - show)$
- We start with putting on the left shoe, so in the meanwhile $on(right - shoe)$ is an open precondition. To put on the left show, we have a precondition of putting the left sock on, which has no preconditions, so that is achieved by the start state.
- Since the effect and preconditions are not conflicted with any previous actions we did, right show and left shoe handling can be in different branches.

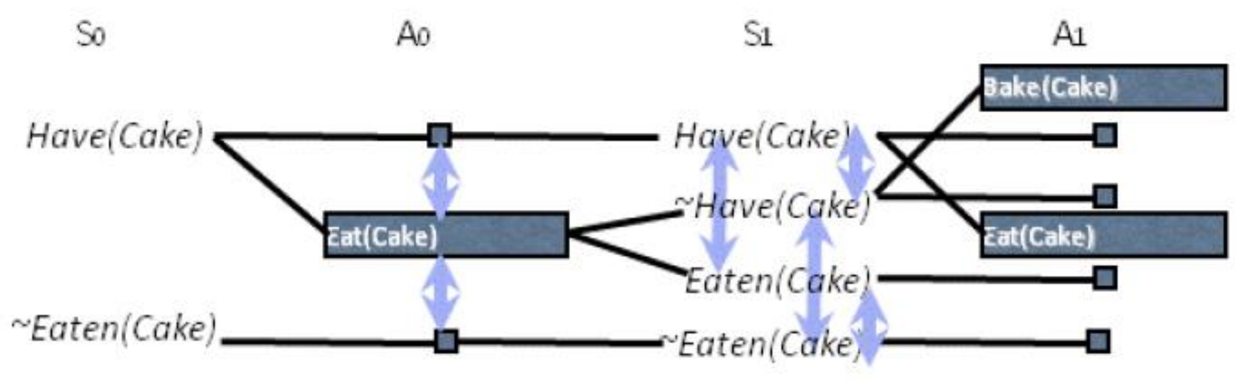
<u>Heuristics for POP</u>

- The action that satisfies the maximum preconditions
- Heuristic: number of open preconditions
- Most constrained variable heuristic – pick open condition that can be satisfied in the fewest number of ways

Planning graphs

- "level" graphs: sequence of levels where level 0 contains the initial state.
- The levels alternate between literals and actions. The first one is the preconditions.

We add exclusion edges if they lead to conflicted states, for instance doing nothing leads to still having the cake, and eating the cake leads to not having the cake – so there's an exclusion edge between those two actions (eat(cake) and NOP).



Note that have(cake) and eaten(cake) are mutually exclusive.

There's always a NOP from every node in the propositional levels.

In an action level an edge exists if the effects of one negates the preconditions of the other. For instance, since bake(cake) leads to have(cake), which is a precondition of eat(cake), bake(cake) and eat(cake) are mutex.

In a proposition level, an edge exists between two literals if the actions they source from are mutex.

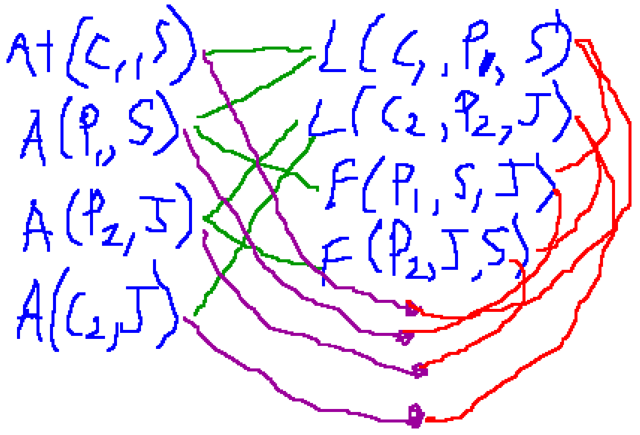
Graphplan: completeness

How do we know if/when there is no feasible plan?

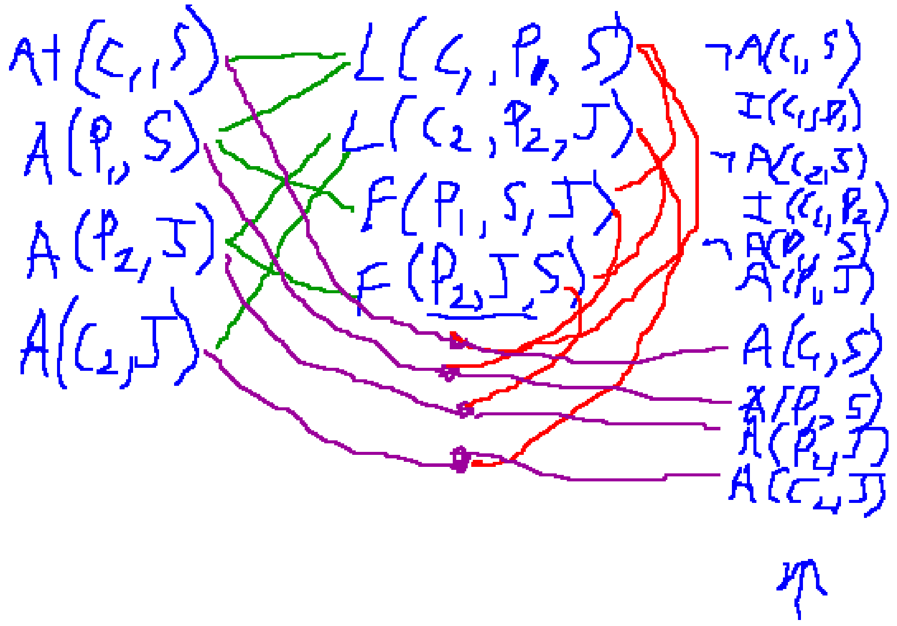
- Definition: a planning graph **levels off** if adding a new level does not change the propositions or exclusions between propositions.
- **Theorem**: if the plan is infeasible, then eventually the graph will level off
- **Theorem**: if the graph has leveled off and we expand the graph without discovering/memorizing any new infeasible goals, then there is no feasible plan. Basically means: we didn't learn anything new by performing the backward chaining, so there's no feasible solution.

The airplane cargos example solution

Step 1: red are mutex edges

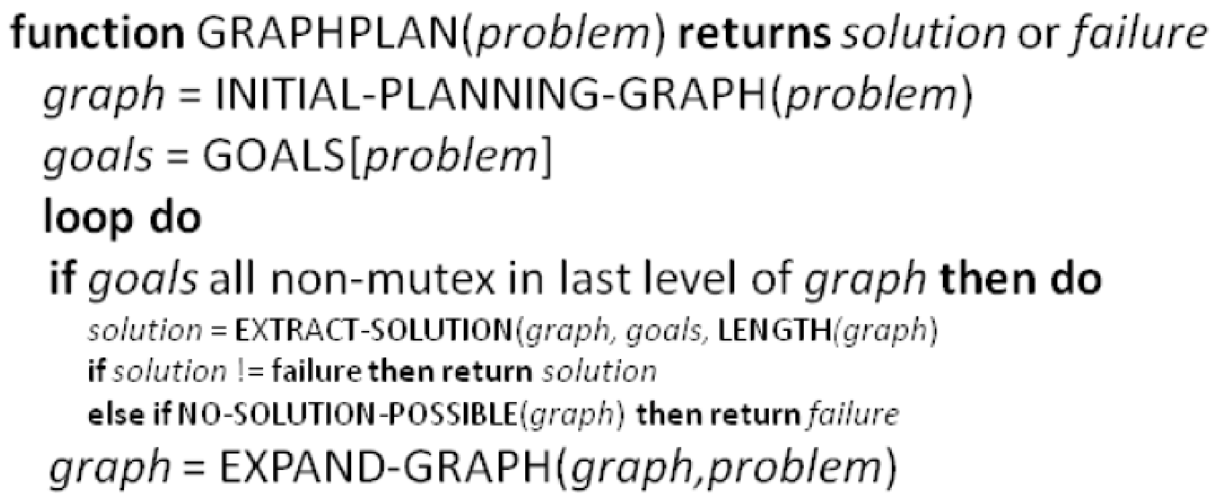


Step 2: every proposition that is true is written in the new proposition level



Step 3: …

<u>GraphPlan Algorithm</u>



```
function GRAPHPLAN(problem) returns solution or failure
  graph = INITIAL-PLANNING-GRAPH(problem)
  goals = GOALS[problem]
  loop do
  if goals all non-mutex in last level of graph then do
     solution = EXTRACT-SOLUTION(graph, goals, LENGTH(graph))
     if solution != failure then return solution
     else if NO-SOLUTION-POSSIBLE(graph) then return failure
  graph = EXPAND-GRAPH(graph,problem)
```