Finish Planning; Knowledge Representation

CS156

Chris Pollett

Nov 10, 2014

Outline

- GraphPlan
- Knowledge Representation

Introduction

- On Wednesday, we were discussing how to estimate the number of steps/actions needed to come up with a plan.
- The idea was using such a heuristic a problem solving agent could do something like $A^\star$ search to look for a state that implied the goal state.
- One of the means for doing this kind of estimation was to in polynomial time create a planning graph for the planning problem.
- We start today by looking at an algorithm, GraphPlan, that extracts from the planning graph, a plan to solve a planning problem.
- We then switch from talking about planning to an area of AI known as knowledge representation.

GraphPlan

The GraphPlan algorithm is as follows:

```
function GraphPlan(problem) returns solution or failure:
    graph := Initial-Planning-Graph(problem)
    goals := Conjuncts(problem, Goal)
    nogoods := empty hash table
    for t= 0 to infty do:
        if goals all present and non-mutex in S_t of graph then
            solution := Extract-Solution(graph, goals, NumLevels(graph), nogoods)
            if solution != failure then return solution
        if graph  and nogoods have not changed since t-1 then return failure
        graph := Expand-Graph(graph, problem)
```

- Initial-Planning-Graph just computes and returns the S_0 layer of the planning graph.
- Conjuncts takes the Goal for the problems and makes a list of fluents
- Expand-Graph computes A_{i-1} and S_i from S_{i-1} including mutex relations.
- We will describe algorithms for Extract-Solution on the next slide.

Two Approaches to Extract Solution

There are two common approaches to computing extract solution:

1. If all the goals are present and non-mutex, one can take the current graph and formulate it as a CSP. The variables for this CSP are the actions at each level, the values for each variable are either **in** or **out** of the plan, and the constraints are the mutexes and the need to satisfy each goal and precondition. One can then use CSP algorithms to solve this problem.
2. Another way to code Extract-Solution is as a backward search problem, where each state in the search contains a pointer to a level in the planning graph and a set of unsatisfied goals. This search problem can be defined as follows:
   - *The initial state is the last level of the planning graph S_n, along with the set of goals from the planning problem.*
   - *The actions available at level S_i are to select any conflict-free subset of the actions in A_{i-1} whose effects cover the goals in the state. The resulting state has level S_{i-1} and has as its set of goals the preconditions for the selected set of actions. Here "conflict free" means a set of actions such that no two of them are mutex and no two of their preconditions are mutex.*
   - *The goal is to reach a state at level S_0 such that all the goals are satisfied.*
   - *The cost of each action is 1.*

GraphPlan - Spare Tire Example

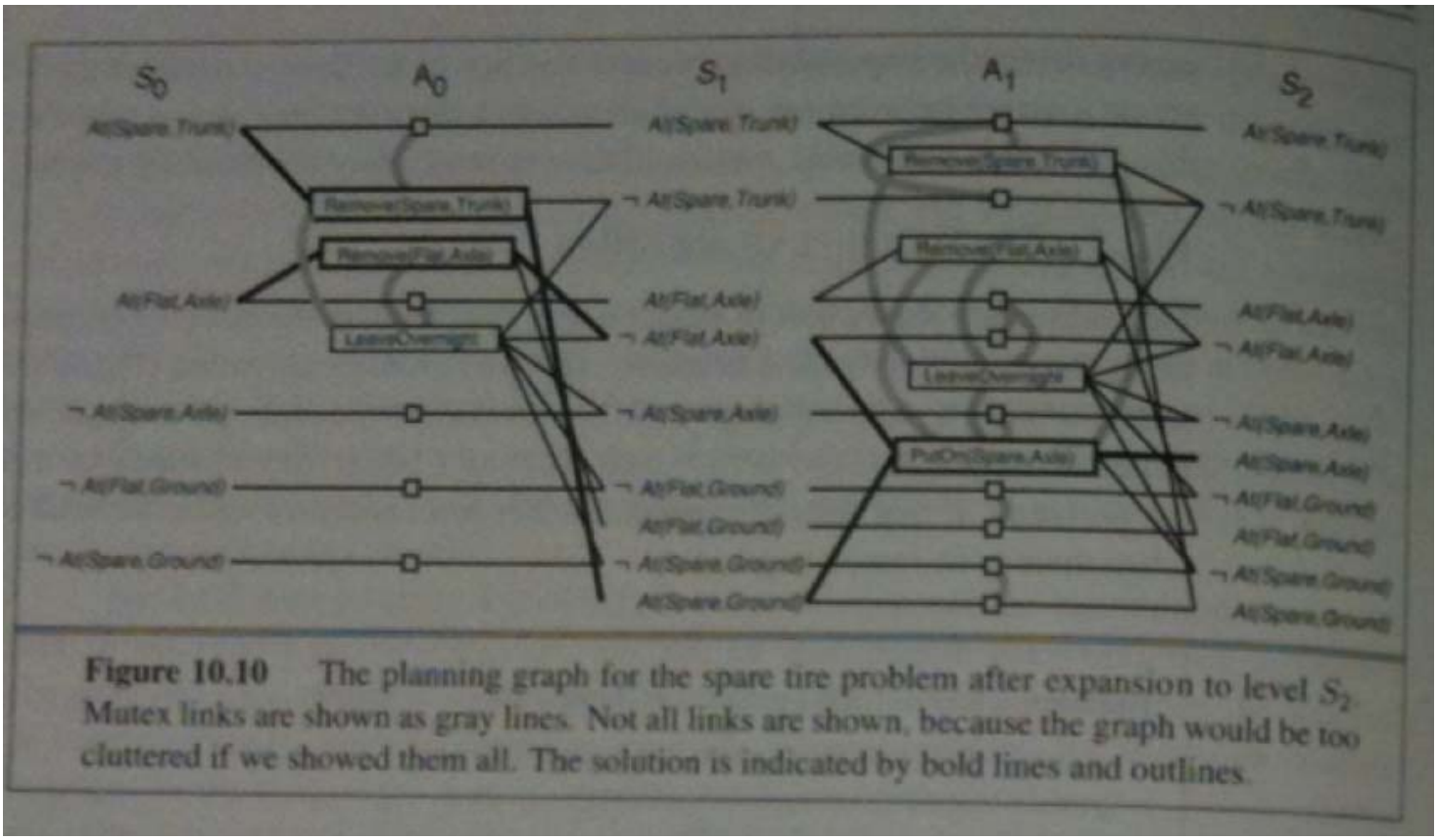


Figure 10.10 The planning graph for the spare tire problem after expansion to level S_2. Mutex links are shown as gray lines. Not all links are shown, because the graph would be too cluttered if we showed them all. The solution is indicated by bold lines and outlines.

- Notice the goal $At(Spare, Axle)$ is not present in in S_0 so we would call Expand-Graph adding into A_0 the three actions whose preconditions exist at level S_0. Using these actions we get S_1.
- Again, $At(Spare, Axle)$ is still not present in S_1, so Expand-Graph is called again, adding A_1 and S_2.
- At this point we might be able to call Extract-Solution.
- Before we continue, let's classify the kinds of mutexes which we have in our graph:
  - ***Inconsistent effects***: *Remove(Spare, Trunk) is mutex with LeaveOvernight because one has the effect* $At(Spare, Ground)$ *and the other its negation.*
  - ***Interference***: $Remove(Flat, Axle)$ *is mutex with LeaveOvernight because one has the precondition* $At(Flat, Axle)$ *and the other has its negation as an effect. i.e., if we imagine trying to write these operation in serial order, if we did* $LeaveOvernight$ *first, we couldn't do* $Remove(Flat, Axle)$ *, on the other hand, if we did* $Remove(Flat, Axle)$ *first, then* $LeaveOvernight$ *has no effect.*
  - ***Competing needs***: $PutOn(Spare, Axle)$ *is mutex with* $Remove(Flat, Axle)$ *because one has* $At(Flat, Axle)$ *as a precondition and the other has its negation.*
  - ***Inconsistent support***: $At(Spare, Axle)$ *is mutex with* $At(Flat, Axle)$ *in* S_2

because the only way of achieving $At(Spare, Axle)$ *is by* $PutOn(Spare, Axle)$ *and that is mutex with the persistence action that is only way of achieving* $At(Flat, Axle)$ *. Thus, the mutex relations relations detect the immediate conflict that arises from trying to put two objects in the same space at the same time*

Backward Search and Spare Tire Problem

- If we used backward search to try to solve the spare tire problem, then we start at S_2 with the goal $At(Spare, Axle)$.
- The only choice for achieving this goal set is $PutOn(Spare, Axle)$. That brings us to a search state at S_1 with goals $At(Spare, Ground)$ and $\neg At(Flat, Axle)$.
- The former can be achieved only by $Remove(Spare, Trunk)$, and the latter by either $Remove(Flat, Axle)$ or $LeaveOvernight$.
- $LeaveOvernight$ is mutex with $Remove(Spare, Trunk)$, so we choose $Remove(Spare, Trunk)$ and $Remove(Flat, Axle)$
- This brings us to a search of state S_0 with the goals $At(Spare, Trunk)$ and $At(Flat, Axle)$. As both of these are present in S_0 we have found a solution: $Remove(Spare, Trunk)$, $Remove(Flat, Axle)$, and $PutOn(Spare, Axle)$.
- In the case where Extract-Solution fails to find a solution for a set of goals at a given level, we record the pair (level, goals) pairs as a **no-good**. If Extract-Solution is called again with the same level and goals, we just immediately return failure.

GraphPlan Heuristics

- We know planning is PSPACE-complete and making the planning graph is in polynomial time.
- So the backward search can still be hard, so heuristics are still needed for choosing among actions.
- People often use a greedy algorithm based on the level cost of literals.
- For any set of goals, we proceed in the following order:
  1. *Pick first the literal with the highest level cost.*
  2. *To achieve that literal, prefer actions whose sum of the level costs of its preconditions is smallest.*
- Although we won't show it in class, one can show that the GraphPlan Algorithm terminates.

Quiz

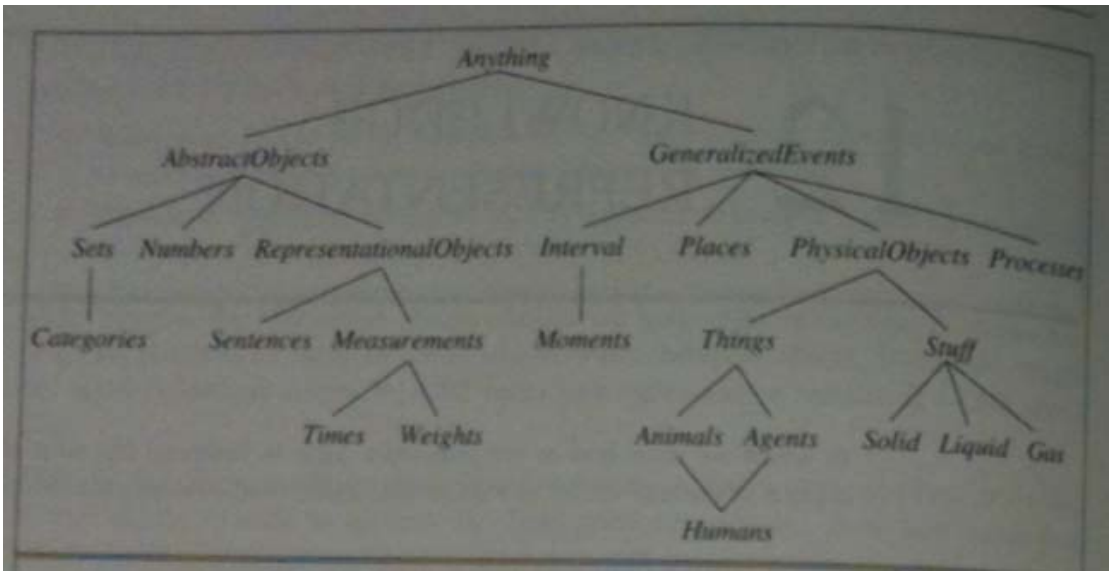
Which of the following is true?

1. Goals in PDDL are not allowed to have negated atoms.
2. The unique names assumption in PDDL says that two constants with different names refer to different elements of any model of the given problem.
3. The ignore preconditions heuristic drops all effects from actions in a PDDL problem.

Knowledge Representation

- So far we have looked at the technology needed for knowledge-based agents: the syntax, semantics, proof theory, and the implementation of agents that use these logics.
- We now turn to the question of what content to put into an agent's knowledge base -- how to represent facts about the world.
- For "toy" domains like the Wumpus world, the choice of how to represent things is less important, many choices will work.
- For complex domains such as shopping on the Internet, driving a car in traffic, it is important to have flexible enough representations.
- The concepts that people commonly try to represent are: Events, Time, Physical Objects, and Beliefs.
- How we choose to represent these abstract objects is sometimes called **ontological engineering**.

Coming up with Ontologies



- One approach to coming up with an ontology is to take an OO-like approach
- We describe general properties of things like a "Physical Object" or a computer "Window"
- Then we allow people to subclass this to get more specific concepts like "SpreadSheetWindow"
- From this we get a hierarchy like in the image above.
- This kind of ontology is called an **upper ontology**.
- We express these ontologies in first-order logic. However, one problem we will encounter is that generalizations have exceptions which are often hard to express in logic.
- For example, although "tomatoes are red" is usually true one can have green tomatoes.
- So we will want to look at systems that can handle this.

Famous Ontologies

- Most ontologies that have been created and have been successful are for a particular domain.
- For example, one might come up with an ontology that lets one describe circuits, people can write their circuits in this ontology, then theorem provers can be used to prove correctness facts about a given circuit (or else find a bug).
- So the field of knowledge engineering has many different ontologies, one might ask if one can come up with a common language for describing everything?
- There have been several famous attempts to do this and each of them has their uses:
    1. *Cyc which was developed by Lenat and described in Lenat and Guha (1990). A group of logicians by hand built this system.*
    2. *DBPedia which was made by trying to extract categories, values, attributes in a semi-automated fashion for Wikipedia. (Bizer, et al 2007)*
    3. *TextRunner, which has built by parsing text documents from the web and trying to extract information from them. (Banko and Etzioni 2008)*
    4. *OpenMind, which was built by volunteers who propose facts in English (Singh, et al 2002; Chklovski and Gil 2005.)*