Study Notes for AI (CS 440/520)
Lecture 9: Classical Planning

Corresponding Book Chapters: 10.1-10.2
Note: These notes provide only a short summary and some highlights of the material covered in the corresponding lecture based on notes collected from students. Make sure you check the corresponding chapters. Please report any mistakes in or any other issues with the notes to the instructor.

The task of coming up with a sequence of actions that will achieve a goal is called planning. We have seen two examples of planning agents so far: the search-based problem-solving agents and logical planning agents. Here we consider environments that are fully observable, deterministic, finite, static (change happens only when the agent acts), and discrete (in time, action, objects, and effects). These are called classical planning environments. In contrast, non-classical planning is for partially observable or stochastic environments and involves a different set of algorithms and agent designs.

1 A Language for Describing Planning Problems

The representation of planning problems - states, actions, and goals - should make it possible for planning algorithms to take advantage of the logical structure of the problem. The key is to find a language that is expressive enough to describe a wide variety of problems, but restrictive enough to allow efficient algorithms to operate over it.

The Planning Domain Definition Language (PDDL) is an attempt to standardize Artificial Intelligence (AI) planning languages. It was first developed by Drew McDermott and his colleagues in 1998 (inspired by STRIPS: the STanford Research Institute Problem Solver) mainly to make the 1998/2000 International Planning Competition (IPC) possible, and then evolved with each competition.

Representation of states: Planners decompose the world into logical conditions and represent a state as a conjunction of positive literals. For example, $At(Plane1, Melbourne) \wedge At(Plane2, Sydney)$ might represent a state in a package delivery problem. The literals must be ground (i.e., not variables) and function-free. Literals such as $At(x, y)$ or $At(Father(Fred), Sydney)$ are not allowed. The closed-world assumption is used, meaning that any conditions that are not mentioned in a state are assumed false.

Representation of goals: A goal is a partially specified state, represented as a conjunction of positive ground literals, such as $Rich \wedge Famous$ or $At(P2, Tahiti)$. A propositional state s satisfies a goal g if s contains all the atoms in g (and possibly others). For example, the state $Rich \wedge Famous \wedge Miserable$ satisfies the goal $RichFamous$.

Representation of actions: An action is specified in terms of the preconditions that must hold before it can be executed and the effects that ensue when it is executed. For example, an action for flying a plane from one location to another is:

$Init(At(C_1, SFO) \wedge At(C_2, JFK) \wedge At(P_1, SFO) \wedge At(P_2, JFK)$
$\wedge Cargo(C_1) \wedge Cargo(C_2) \wedge Plane(P_1) \wedge Plane(P_2)$
$\wedge Airport(JFK) \wedge Airport(SFO))$
$Goal(At(C_1, JFK) \wedge At(C_2, SFO))$
$Action(Load(c, p, a),$
PRECOND: $At(c, a) \wedge At(p, a) \wedge Cargo(c) \wedge Plane(p) \wedge Airport(a)$
EFFECT: $\neg At(c, a) \wedge In(c, p))$
$Action(Unload(c, p, a),$
PRECOND: $In(c, p) \wedge At(p, a) \wedge Cargo(c) \wedge Plane(p) \wedge Airport(a)$
EFFECT: $At(c, a) \wedge \neg In(c, p))$
$Action(Fly(p, from, to),$
PRECOND: $At(p, from) \wedge Plane(p) \wedge Airport(from) \wedge Airport(to)$
EFFECT: $\neg At(p, from) \wedge At(p, to))$

Figure 1: A specification of a problem involving transportation of air cargo between airports.

$$\begin{aligned} Action(\ & Fly(p, from, to), \\ & PRECOND: \quad At(p, from) \wedge Plane(p) \wedge Airport(from) \wedge Airport(to), \\ & EFFECT: \quad \neg At(p, from) \wedge At(p, to)\ ) \end{aligned}$$

The above definition represents a number of different actions that can be derived by instantiating the variables p, $from$, and to to different constants.

- The action name and parameter list - for example, $Fly(p, from, to)$ - serves to identify the action.
- The precondition is a conjunction of function-free positive literals stating what must be true in a state before the action can be executed. Any variables in the precondition must also appear in the action's parameter list.
- The effect is a conjunction of function-free literals describing how the state changes when the action is executed. A positive literal P in the effect is asserted to be true in the state resulting from the action, whereas a negative literal $\neg P$ is asserted to be false. Variables in the effect must also appear in the action's parameter list.

An action is applicable in any state that satisfies the precondition; otherwise, the action has no effect. Establishing applicability will involve a substitution for the variables in the precondition. For example, suppose the current state is described by:

$$At(P1, JFK) \wedge At(P2, SFO) \wedge Plane(P1) \wedge Plane(P2) \wedge Airport(JFK) \wedge Airport(SFO)$$

This state satisfies the precondition:

$$At(p, from) \wedge Plane(p) \wedge Airport(from) \wedge Airport(to)$$

with substitution $p/P1, from/JFK, to/SFO$ (among others - see Exercise 11.2). Thus, the concrete action $Fly(P1, JFK, SFO)$ is applicable.
Starting in state s, the result of executing an applicable action a is a state s' that is the same as s except that any positive literal P in the effect of a is added to s' and any negative literal $\neg P$ is removed from s'. Thus, after $Fly(P1, JFK, SFO)$, the current state becomes:

$$At(P1, SFO) \wedge At(P2, SFO) \wedge Plane(P1) \wedge Plane(P2) \wedge Airport(JFK) \wedge Airport(SFO)$$

Every literal not mentioned in the effect remains unchanged. The solution for a planning problem is just an action sequence that, when executed in the initial state, results in a state that satisfies the goal.

1.1 Example: Air Cargo Transport

Figure 1 shows an air cargo transport problem involving loading and unloading cargo onto and off of planes and flying it from place to place. The problem can be defined with three actions: $Load$, $Unload$, and Fly. The actions affect two predicates: $In(c, p)$ means that cargo c is inside plane p, and $At(x, a)$ means that object x (either plane or cargo) is at airport a. Note that cargo is not At anywhere when it is In a plane, so At really means "available for use at a given location." It takes some experience with action definitions to handle such details consistently. The following plan is a solution to the problem:

$$[Load(C1, P1, SFO), Fly(P1, SFO, JFK), Load(C2, P2, JFK), Fly(P2, JFK, SFO)].$$

1.2 Example: The blocks world

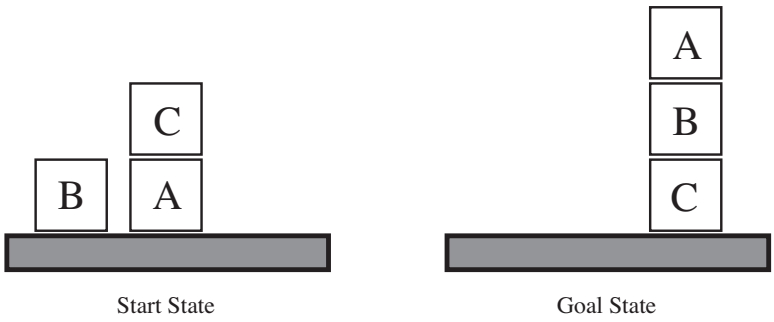

Figure 2: An example of the Blocks World problem.

One of the most famous planning domains is known as the blocks world. This domain consists of a set of cube-shaped blocks sitting on a table. The blocks can be stacked, but only one block can fit directly on top of another. A robot arm can pick up a block and move it to another position, either on the table or on top of another block. The arm can pick up only one block at a time, so it cannot pick up a block that has another one on it. The goal will always be to build one or more stacks of blocks, specified in terms of what blocks are on top of what other blocks. For example, a goal might be to get block A on B and block C on D.

We will use $On(b, x)$ to indicate that block b is on x, where x is either another block or the table. The action for moving block b from the top of x to the top of y will be $Move(b, x, y)$. Now, one of the preconditions on moving b is that no other block be on it. In first-order logic, this would be $\nexists\, x\ On(x, b)$ or, alternatively, $\forall\, x\ \neg On(x, b)$. We can introduce, however, a new predicate, $Clear(x)$, that is true when nothing is on x.

The action $Move$ moves a block b from x to y if both b and y are clear. After the move is made, x is clear but y is not. A formal description of $Move$ is the following:

$$\begin{aligned} &Action(Move(b, x, y),\\ &\quad PRECOND: \quad On(b, x) \wedge Clear(b) \wedge Clear(y),\\ &\quad EFFECT: \quad On(b, y) \wedge Clear(x) \wedge \neg On(b, x) \wedge \neg Clear(y)) \end{aligned}$$

Unfortunately, this action does not maintain $Clear$ properly when x or y is the table. When $x = Table$, this action has the effect $Clear(Table)$, but the table should not become clear, and when $y = Table$, it has the precondition

$Clear(Table)$, but the table does not have to be clear to move a block onto it. To fix this, we do two things. First, we introduce another action to move a block b from x to the table:

$$\begin{aligned} &Action(MoveToTable(b, x),\\ &\qquad PRECOND: \quad On(b, x) \wedge Clear(b)),\\ &\qquad EFFECT: \quad On(b, Table) \wedge Clear(x) \wedge \neg On(b, x)). \end{aligned}$$

Second, we take the interpretation of Clear(b) to be there is a clear space on b to hold a block. Under this interpretation, Clear(Table) will always be true. The only problem is that nothing prevents the planner from using Move(b, x, Table) instead of MoveToTable(b, x). We could live with this problemit will lead to a larger-than-necessary search space, but will not lead to incorrect answersor we could introduce the predicate Block and add Block (b) Block(y) to the precondition of Move.

Finally, there is the problem of spurious actions such as Move(B, C, C), which should be a no-op, but which has contradictory effects. It is common to ignore such problems, because they seldom cause incorrect plans to be produced. The correct approach is add in- equality preconditions as shown in Figure 3.

$$\begin{aligned} &Init(On(A, Table) \wedge On(B, Table) \wedge On(C, Table)\\ &\quad \wedge Block(A) \wedge Block(B) \wedge Block(C)\\ &\quad \wedge Clear(A) \wedge Clear(B) \wedge Clear(C))\\ &Goal(On(A, B) \wedge On(B, C))\\ &Action(Move(b, x, y),\\ &\quad \text{PRECOND: } On(b, x) \wedge Clear(b) \wedge Clear(y) \wedge Block(b) \wedge\\ &\qquad (b \neq x) \wedge (b \neq y) \wedge (x \neq y),\\ &\quad \text{EFFECT: } On(b, y) \wedge Clear(x) \wedge \neg On(b, x) \wedge \neg Clear(y))\\ &Action(MoveToTable(b, x),\\ &\quad \text{PRECOND: } On(b, x) \wedge Clear(b) \wedge Block(b) \wedge (b \neq x),\\ &\quad \text{EFFECT: } On(b, Table) \wedge Clear(x) \wedge \neg On(b, x)) \end{aligned}$$

Figure 3: A planning problem in the blocks world: building a three-block tower. One solution is the sequence $[Move(B, Table, C), Move(A, Table, B)]$.

2 Planning with State-Space Search

A straightforward approach to solve planning challenges is to use state-space search. Because the action descriptions in a planning problem specify both preconditions and effects, it is possible to search in either direction: either forward from the initial state or backward from the goal, as shown in Figure 4. We can also use the explicit action and goal representations to derive effective heuristics automatically.

2.1 Forward state search

It is sometimes called progression planning, because it moves in the forward direction. We start in the problems initial state, considering sequences of actions until we find a sequence that reaches a goal state. The formulation of planning problems as state-space search problems is as follows:

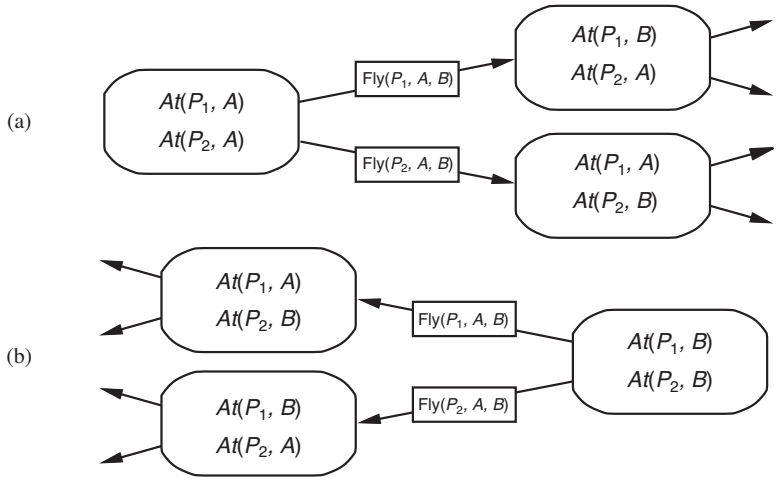

Figure 4: Two approaches searching for a plan. (a) Forward (progression) state-space search, starting in the initial state and using the problem's actions to search forward for the goal state. (b) Backward (regression) state-space search: a search process starting at the goal state(s) and using the inverse of the actions to search backward for the initial state.

- The initial state of the search is the initial state from the planning problem. In general, each state will be a set of positive ground literals; literals not appearing are false.
- The actions that are applicable to a state are all those whose preconditions are satisfied. The successor state resulting from an action is generated by adding the positive effect literals and deleting the negative effect literals. Note that a single successor function works for all planning problems - a consequence of using an explicit action representation.
- The goal test checks whether the state satisfies the goal of the planning problem.
- The step cost of each action is typically 1. It is possible to allow different costs for different actions.

Recall that, in the absence of function symbols, the state space of a planning problem is finite. Therefore, any graph search algorithm that is completefor example, A^*will be a complete planning algorithm.

From the earliest days of planning research (around 1961) until recently (around 1998) it was assumed that forward state-space search was too inefficient to be practical.

First, forward search does not address the irrelevant action problem - all applicable actions are considered from each state. Second, the approach quickly bogs down without a good heuristic. Consider an air cargo problem with 10 airports, where each airport has 5 planes and 20 pieces of cargo. The goal is to move all the cargo at airport A to airport B. There is a simple solution to the problem: load the 20 pieces of cargo into one of the planes at A, fly the plane to B, and unload the cargo. But finding the solution can be difficult because the average branching factor is huge: each of the 50 planes can fly to 9 other airports, and each of the 200 packages can be either unloaded (if it is loaded), or loaded into any plane at its airport (if it is unloaded). On average, lets say there are about 2000 possible actions, so the search tree up to the depth of the obvious solution has about 2000^{41} nodes. It is clear that a very accurate heuristic will be needed to make this kind of search efficient. We will discuss some possible heuristics after looking at backward search.

2.2 Backward State-space Search

Backward search can be difficult to implement when the goal states are described by a set of constraints rather than being listed explicitly. In particular, it is not always obvious how to generate a description of the possible predecessors of the set of goal states.

The main advantage of backward search is that it allows us to consider only relevant actions. An action is relevant to a conjunctive goal if it achieves one of the conjuncts of the goal. For example, the goal in our 10-airport air cargo problem is to have 20 pieces of cargo at airport B, or more precisely

$$At(C1, B) \wedge At(C2, B) \wedge ... \wedge At(C20, B)$$

Now consider the conjunct $At(C1, B)$. Working backwards, we can seek actions that have this as an effect. There is only one: $Unload(C1, p, B)$, where plane p is unspecified.

Notice that there are many irrelevant actions that can also lead to a goal state. For example, we can fly an empty plane from JFK to SFO; this action reaches a goal state from a predecessor state in which the plane is at JFK and all the goal conjuncts are satisfied. A backward search that allows irrelevant actions will still be complete, but it will be much less efficient. If a solution exists, it will be found by a backward search that allows only relevant actions. The restriction to relevant actions means that backward search often has a much lower branching factor than forward search. For example, our air cargo problem has about 1000 actions leading forward from the initial state, but only 20 actions working backward from the goal.

Searching backwards is sometimes called regression planning. The principal question in regression planning is this: what are the states from which applying a given action leads to the goal? Computing the description of these states is called regressing the goal through the action.

Given definitions of relevance and consistency, we can describe the general process of constructing predecessors for backward search. Given a goal description G, let A be an action that is relevant and consistent. The corresponding predecessor is as follows:

- Any positive effects of A that appear in G are deleted.
- Each precondition literal of A is added, unless it already appears.

Any of the standard search algorithms can be used to carry out the search. Termination occurs when a predecessor description is generated that is satisfied by the initial state of the planning problem. In the first-order case, satisfaction might require a substitution for variables in the predecessor description. For example, the predecessor description in the preceding paragraph is satisfied by the initial state

$$In(C1, P12) \wedge At(P12, B) \wedge At(C2, B) \wedge ... \wedge At(C20, B)$$

with substitution $p/P12$. The substitution must be applied to the actions leading from the state to the goal, producing the solution $[Unload(C1, P12, B)]$.

2.3 Heuristics for State-space Search

Neither forward nor backward search is efficient without a good heuristic function (although it is easier to apply heuristics to forward search). The basic idea is to look at the effects of the actions and at the goals that must be achieved and to guess how many actions are needed to achieve all the goals. Finding the exact number is NP hard, but it is possible to find reasonable estimates most of the time without too much computation. We might also be

able to derive an admissible heuristic - one that does not overestimate. This could be used with A search to find optimal solutions.

The first is to derive a relaxed problem from the given problem specification. The optimal solution cost for the relaxed problem - which we hope is very easy to solve - gives an admissible heuristic for the original problem. The second approach is to pretend that a pure divide-and-conquer algorithm will work. This is called the subgoal independence assumption: the cost of solving a conjunction of subgoals is approximated by the sum of the costs of solving each subgoal independently. The subgoal independence assumption can be optimistic or pessimistic. It is optimistic when there are negative interactions between the sub-plans for each subgoal - for example, when an action in one subplan deletes a goal achieved by another subplan. It is pessimistic, and therefore inadmissible, when subplans contain redundant actions - for instance, two actions that could be replaced by a single action in the merged plan.

Since explicit representations of preconditions and effects are available, the process will work by modifying those representations. The simplest idea is to relax the problem by removing all preconditions from the actions. Then every action will always be applicable, and any literal can be achieved in one step (if there is an applicable action - if not, the goal is impossible). This almost implies that the number of steps required to solve a conjunction of goals is the number of unsatisfied goals - almost but not quite, because (1) there may be two actions, each of which deletes the goal literal achieved by the other, and (2) some action may achieve multiple goals. If we combine our relaxed problem with the subgoal independence assumption, both of these issues are assumed away and the resulting heuristic is exactly the number of unsatisfied goals.

In many cases, a more accurate heuristic is obtained by considering at least the positive interactions arising from actions that achieve multiple goals. First, we relax the problem further by removing negative effects. Then, we count the minimum number of actions required such that the union of those actions' positive effects satisfies the goal. For example, consider

$$\begin{aligned} Goal(A \wedge B \wedge C) \\ Action(X, EFFECT : A \wedge P) \\ Action(Y, EFFECT : B \wedge C \wedge Q) \\ Action(Z, EFFECT : B \wedge P \wedge Q) \end{aligned}$$

The minimal set cover of the goal A, B, C is given by the actions X, Y , so the set cover heuristic returns a cost of 2. This improves on the subgoal independence assumption, which gives a heuristic value of 3. There is one minor irritation: the set cover problem is NP-hard. A simple greedy set-covering algorithm is guaranteed to return a value that is within a factor of $logn$ of the true minimum value, where n is the number of literals in the goal, and usually works much better than this in practice. Unfortunately, the greedy algorithm loses the guarantee of admissibility for the heuristic.

It is also possible to generate relaxed problems by removing negative effects without removing preconditions. That is, if an action has the effect $A \wedge \neg B$ in the original problem, it will have the effect A in the relaxed problem. This means that we need not worry about negative interactions between subplans, because no action can delete the literals achieved by another action. The solution cost of the resulting relaxed problem gives what is called the empty-delete-list heuristic. The heuristic is quite accurate, but computing it involves actually running a (simple) planning algorithm. In practice, the search in the relaxed problem is often fast enough that the cost is worthwhile.