A.I. Planning

1 Classical Planning

We are looking at how to get an agent to ‘think ahead’. But up to now we have used analogical representations. What we’re now going to do is look at how to get our agent to ‘think ahead’ but this time using logical representations of the states of the world. The task of finding sequences of actions in a state space where the states have logical representations is called *A.I. Planning*.

In this lecture and the next two, we look at what is sometimes called *classical planning*. Classical planners make at least the following assumptions:

- The planner has complete and certain knowledge of (relevant portions of) the initial world state;
- each action will be executed infallibly; and
- the world changes only as a result of this agent’s actions (in particular, there are no other agents in the execution environment whose actions would interfere with execution of the plan).

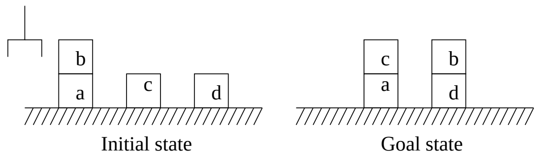
In other words, the environment is fully observable; deterministic and static with only a single agent.

After our three lectures on classical planning, we then have one lecture in which we look at *non-classical planning*. Non-classical planners abandon one or more of these assumptions.

2 The Blocks World

The blocks world is a simple world in which planning algorithms are commonly taught and demonstrated. It comprises a robot arm and a number of labelled wooden cubes, resting on a table.

Here is an example of a start state and a goal state.



The robot is capable of four actions: stacking a block that it is holding on top of another block; unstacking a block from another block; picking a block up from the table; and putting down a block that it is holding onto the table.

3 The STRIPS Language

In fact, we’re not going to allow ourselves the full expressiveness of FOPL. We’ll use a sublanguage. We do this for efficiency. The sublanguage we use is called the *STRIPS language* because it was first used in a planning system developed in the late 1960s/early 1970s called STRIPS (Stanford Research Institute Problem Solver).

Although the STRIPS language is quite a restricted language, it seems to be just about expressive enough to represent many typical planning problems, which explains why it has endured. (Recently, ADL, the Action Description Language, which is a slightly more expressive subset of FOPL, has been gaining ground and replacing the STRIPS language in A.I. planners. For simplicity, we’ll stick to the STRIPS language.)

Here is how we encode actions, states and goals in the STRIPS language.

3.1 States

Each state will be described by a conjunction of function-free ground atoms.

So here is a representation of the start state depicted earlier:

$$\begin{gathered}\text{on(b, a)} \wedge \text{ontable(a)} \wedge \text{ontable(c)}\\ \wedge\, \text{ontable(d)} \wedge \text{armempty}\\ \wedge\, \text{clear(b)} \wedge \text{clear(c)} \wedge \text{clear(d)}\end{gathered}$$

You can see we have only conjunction (no negation, no disjunction, etc.); we have ground atoms (no variables) and they are function-free (no function symbols). So, in effect, the arguments of the predicate symbols are all constant symbols.

In the STRIPS language, we make the *closed-world assumption*. This means that any statement that is not mentioned in a state representation is assumed false. For example, in the start state, because clear(a) is not mentioned, it is assumed to be false. Similarly, holding(c) and on(e, c) are assumed to be false.

3.2 Goals

The goal is also represented by a conjunction of function-free ground atoms.

So here is a representation of the goal depicted earlier:

$$\begin{gathered}\text{on(c, a)} \wedge \text{on(b, d)}\\ \wedge\, \text{ontable(a)} \wedge \text{ontable(d)}\\ \wedge\, \text{clear(c)} \wedge \text{clear(b)}\end{gathered}$$

In general, goals do not have to be fully specified. So, for example, suppose someone’s goal is simply

$$\text{on(c, b)} \wedge \text{on(b, a)}$$

Then, any state in which this wff is true is a goal state, e.g. states where additionally

- d is on c and a is on the table;
- d is on the table and a is on the table
- a is on d and d is on the table

provided c *is on* b *and* b *is on* a.

3.3 Operators

Information about each action is encoded in the form of an operator. Each operator comprises three components:

Action: A name and a parameter list.

Precondition: A conjunction of function-free atoms that need to be true for the operator to be applicable in a state. So these are atoms where the arguments can be constant symbols or they can be variables from the parameter list.

Effect: A conjunction of function-free literals that describes the way the world changes if the operator is executed. So these are atoms or *negated* atoms where the arguments can be constant symbols or they can be variables from the parameter list.

It is common to divide the *effect* into two wffs:

- the *add list* (the positive effects) — what will be newly true in the state after executing the action; and
- the *delete list* (the negative effects) — what will no longer be true in the state after executing the action.

Any variables in an operator must appear in the parameter list and are taken to be universally quantified. An operator with variables in it denotes a family of actions: when inserting an action into a plan, the variables must, at some point, be instantiated, so we know what object the action is being done to.

Here are the operators that specify the four actions of the Blocks World:

Op(	ACTION:	stack(x, y),
	PRECOND:	clear(y) $\wedge$ holding(x),
	EFFECT:	$\neg$clear(y) $\wedge$ $\neg$holding(x) $\wedge$
		armempty $\wedge$ on(x, y) $\wedge$ clear(x))
Op(	ACTION:	unstack(x, y),
	PRECOND:	on(x, y) $\wedge$ clear(x) $\wedge$ armempty,
	EFFECT:	$\neg$on(x, y) $\wedge$ $\neg$armempty $\wedge$
		$\neg$clear(x) $\wedge$ holding(x) $\wedge$ clear(y))
Op(	ACTION:	pickup(x),
	PRECOND:	clear(x) $\wedge$ ontable(x) $\wedge$ armempty,
	EFFECT:	$\neg$ontable(x) $\wedge$ $\neg$armempty $\wedge$
		$\neg$clear(x) $\wedge$ holding(x))
Op(	ACTION:	putdown(x),
	PRECOND:	holding(x),
	EFFECT:	$\neg$holding(x) $\wedge$ ontable(x) $\wedge$
		clear(x) $\wedge$ armempty)

It's worth pointing out that it's not easy to come up with good operator descriptions. There are at least three problems:

The qualification problem: It is hard to define preconditions: the circumstances under which an action is *guaranteed* to work, e.g. to pick up an object requires that it not be too heavy. Leaving things out gives danger of execution failure.

The ramification problem: It is hard to define effects: especially implicit consequences of actions, e.g. moving a container moves its contents.

The frame problem: To give a logically correct specification of an operator (one that allows the right inferences), we would have to specify not only what changes (effects) but that everything else doesn't change. In STRIPS operators, we only say what changes. It is implicit in the specialised planning algorithms we use that all else remains unchanged.

4 State-Space Planning

We can try to use our state space search algorithms for A.I. planning. The main difference is that states will have logical representations instead of analogical representations. Because the operators specify both the preconditions and the effects *declaratively*, we can easily conduct the search in either a forwards or a backwards direction, as explained further below.

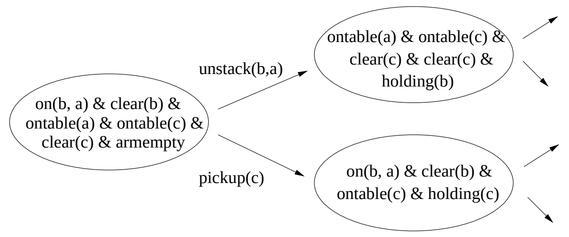
4.1 Progression Planning

Progression planners plan in a forwards direction, from the start state to the goal.

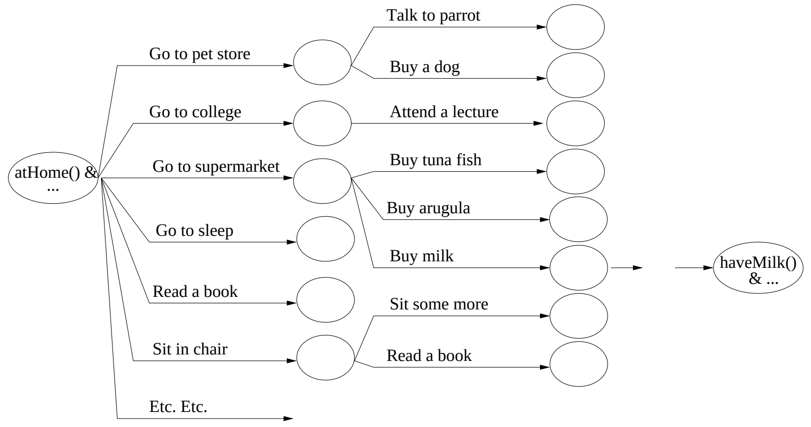
An operator is applicable to the current state if the atoms in its precondition all unify with atoms in the current state.

We generate the successors by taking the current state and

- inserting the operator's positive effects (add-list), and
- deleting the operator's negative effects (delete-list).


on(b, a) & clear(b) & ontable(a) & ontable(c) & clear(c) & armempty
unstack(b,a)
ontable(a) & ontable(c) & clear(c) & clear(b) & holding(b)
pickup(c)
on(b, a) & clear(b) & ontable(c) & holding(c)

Until the beginning of the 1990s, it was assumed that progression planning was too inefficient to be practical. The branching factor is very high: in any one state many operators will be applicable. Many of these operators, while applicable, will be irrelevant: they do nothing to help us to achieve the goal. This isn't easily illustrated using the Blocks World. Here is a diagram from a different domain that is suggestive of the possible problems:



(Based on Fig. 11.2 in S.Russell & P.Norvig: *Artificial Intelligence: A Modern Approach*, Prentice-Hall, 1995.)

Of course, a large *state space* is not necessarily a problem. What matters is the *search tree*. If we can devise a highly focused search strategy, the search tree may not be so bad.

Unfortunately, heuristics are not much help. They choose among states, not operators. So, even if we had a great heuristic, we would need to generate all successors before using the heuristic to select among the many states on the agenda. And, good heuristics are hard to devise for logical representations anyway. The most obvious heuristic is to count how many conjuncts in the goal are not yet true in the current state. (Question: do you think this is an admissible heuristic?) More accurate variations of this have also been devised but, unfortunately, these heuristic functions are themselves typically quite expensive to compute.

4.2 Regression Planning

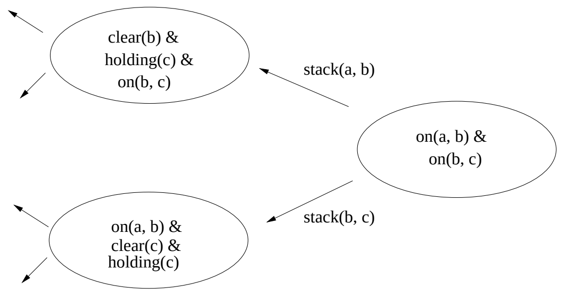
Regression planners plan in a backwards direction, from the goal to the start state, *applying operators in reverse*. At each point, we compute *predecessor states*.

For example, if the goal contains the atom *holding*(a), then we find all operators that have *holding*(x) as a positive effect.

We generate the predecessors by taking the goal and

- deleting all of the operator's positive effects, and
- inserting all of the operator's preconditions.

The predecessor now acts as a *subgoal*: it is a state that we would like to achieve. So we compute its predecessors. And so we continue until, if possible, we reach a subgoal all of whose atoms are true in the start state.



clear(b) & holding(c) & on(b, c)
stack(a, b)
on(a, b) & on(b, c)
stack(b, c)
on(a, b) & clear(c) & holding(c)

This has a major advantage: it only considers relevant operators. The only operators installed into the search tree are ones that make goal (or subgoal) atoms true. It tends therefore to give a much more focused search than progression planning does.

However, in realistic problems, the branching factor is still often very large. Heuristics remain important for focusing the search. In a similar vein to progression planning, it may be possible to base the heuristic on a count of how many atoms a predecessor has in common with the start state.

5 Problem Decomposition

Standard state-based search, whether done forwards or backwards, may still not be efficient enough. A good heuristic is essential but hard to come by. So we need to look for further ideas to improve efficiency.

One such idea is to use *problem decomposition*: to *divide-and-conquer*. We can work on bits of the problem, then combine the partial solutions to get an overall solution. The fact that goals (and subgoals) are conjunctions facilitates this. In effect, we can work on each conjunct of the goal separately.

Of course, there may be a problem with this. The partial solutions found for each part of the goal may conflict.

Problem decomposition will work best if the problem is perfectly decomposable into manageable independent sub-problems. Then there's no risk of conflict.

But the idea of problem decomposition might still be viable for most real-world problems. While real-world problems are rarely perfectly decomposable, they are often *nearly decomposable*: there is limited interaction between subproblems. So we can solve them as if they were perfectly decomposable, and then, when we combine the partial solutions, we can fix any problematic interactions. As long as these are few in number and easily fixed, we should benefit.

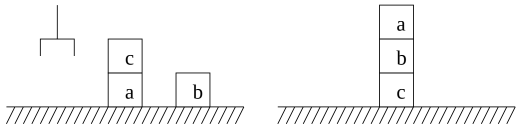
This is a very human problem-solving strategy. We tend to break problems down into subproblems, solve them largely independently and then try to stick the partial solutions together, at which point we fix any troublesome interactions. And this works well for us on most real-world problems.

As a speculation, one reason why we find puzzles challenging may be because the puzzle designer deliberately sets out to come up with a problem in which problem decomposition doesn't work. Either the puzzle is barely decomposable, or the puzzle is decomposable but not in an obvious way.

The idea of problem decomposition has gone a very long way to making state-based planning systems practical. Many of the most efficient A.I. planners now work in a forwards direction, exploiting powerful heuristics and problem decomposition. However, A.I. researchers have come up with a variety of other ideas too, and we will now look at these.

6 The Sussman Anomaly

Here is a planning problem, known as the *Sussman anomaly*, which causes problems for many simple-minded planning algorithms, especially if they use problem decomposition.



Initial state Goal state

Some planning algorithms cannot solve this at all; others can solve it, but only sub-optimally.

The goal is *on*$(a, b) \wedge$ *on*(b, c). Suppose we decompose the goal into its two conjuncts and try to solve *on*(a, b) first. The solution that a simple-minded planner might find (if it finds one at all) is:

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)

If, on the other hand, it solves *on*(b, c) first, it finds a different sub-optimal plan. (You work this one out for yourself.)

A major cause of the problem is the assumption, in simple-minded planners, that the order in which actions are found during planning should be their ordering in the plan, i.e. the order in which they will be executed. It turns out that the way to avoid problems such as the Sussman anomaly is to break the connection between the order in which actions are found during planning and the order in which they will be executed: we will allow ourselves to add actions to the plan wherever they are needed, rather than in an unbroken sequence from the start state (or in reverse from the goal).

There are many ways to achieve this. Particularly popular for a long while has been to combine the ideas of the next three sections. In the next lecture, we will see a planner that combines these ideas.

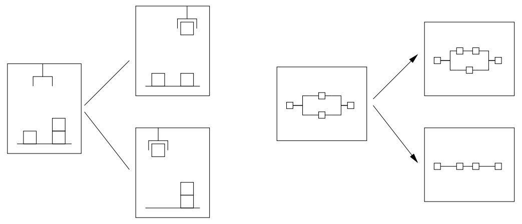
7 Plan Space versus State Space

So far we've been discussing *state space* planners: operators apply to states and goals.

An alternative is *plan space* planners: they search through the plan space. Each node is a (possibly) partial plan. The planner starts with a very simple partial plan. It repeatedly refines the partial plan (often by adding things to it) until it is no longer partial and it solves the problem.

In this kind of search, the nodes are plans, so the operators will not be the STRIPS operators; they will be *plan refinement operators*. Plan refinement operators include such things as: adding a step to the plan; or imposing an ordering constraint on the steps within the plan. They further constrain the plan. (Some people talk of *constraint-posting*.)

Here is a diagram that might illuminate this:



State Space Search Plan Space Search

(Based on Fig. 13.7 in E.Rich & K.Knight: *Artificial Intelligence* (2nd edn.), McGraw-Hill, 1991)

8 Partial-Order versus Total-Order Planning

A planner that can represent plans in which some steps are ordered w.r.t. each other but others are unordered is a *partial-order* or *non-linear* planner.

Simple-minded planners cannot represent plans in this way: even when steps do not need to be ordered, they are still ordered (the ordering coming from the order in which they were determined during planning). We call these *total-order* or *linear* planners.

Plan space planning is amenable to partial-order planning in a way that state space planning is generally not.

9 Least Commitment Planning

Partial-order planning is then usually subject to the principle of *least-commitment*: don't make any choices that you don't have to, e.g. don't order actions unless they must be ordered. Postponement of commitment means, if search needs to backtrack, then fewer decisions need to be undone.

In the next lecture, we look at POP, an example of a regression planner that carries out problem decomposition, that searches plan space rather than state space, that builds a partially-ordered plan and that operates by the principle of least-commitment.