Planning and Acting in Nondeterministic Domains

So far, we've been looking at *classical planning*. To repeat, 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;
- there are no other agents in the execution environment whose actions would interfere with execution of the plan.

These are pretty unrealistic assumptions.

Non-classical planners are ones that abandon one or more of these assumptions. We are going to focus here on abandoning the second assumption. However, the techniques we look at can be very helpful in domains in which the first and third assumptions also do not pertain.

There are two main families of techniques, and which you adopt and when depends on how much indeterminacy there is in the domain.

Bounded indeterminacy: Actions can have unpredictable effects, but the possible effects are known and sufficiently few in number that they can be listed in the operator. For example, the possible outcomes of tossing a coin are *Heads* or *Tails*. To cope with bounded indeterminacy, the agent can make a plan that works for all possible outcomes. The main technique here is *conditional planning*.

Unbounded indeterminacy: Actions can have unpredictable effects, and the full set of possible effects is either unknown or too large to be enumerated in an operator. For example, many of the actions that you take when driving a car exhibit unbounded indeterminacy. The main techniques here are *execution monitoring* and *replanning*.

Of course, some domains have both kinds of indeterminacy, and so you would need to integrate both conditional planning and execution monitoring & replanning into a single system.

1 Conditional Planning

Conditional planning, also known as *contingency planning*, deals with bounded indeterminacy by allowing operators to have *disjunctive effects* and building plans that contain different branches for different eventualities. This implies also that the plan contains *sensing actions* which, when executed, enable the agent to decide which branch of the plan to follow.

Here is the specification of a coin tossing operator with disjunctive effects:

Op(ACTION: tossCoin(x),
PRECOND: haveCoin(x),
EFFECT: landsHeads(x) $\vee$ landsTails(x))

But we can also use disjunctive effects for operators that might go wrong: one set of effects will be when the operator executes correctly (e.g. the block is now no longer on top of another block, it is now in the robot's hand) and the other set of effects will be for when the operator is executed clumsily (e.g. the block is no longer on top of another block, but it isn't in the robot's hand either, since it was dropped, and so now it is on the table):

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

We can also use operators with disjunctive effects for operators that discover information about the world. For example, if we execute an operator that tries turning a door handle, we will know whether the door is locked or not:

Op(ACTION: checkDoor(x),
PRECOND: atDoor(x) $\wedge$ $\neg$knowIf(locked(x)),
EFFECT: know(locked(x)) $\vee$ know($\neg$locked(x)))

(The observant amongst you will note that the above is not strictly first-order predicate logic since it allows wffs to be arguments of predicate symbols. But, it serves our purposes here: an illustration of a disjunctive effect.)

Conditional plans contain conditional steps. We will write these using the syntax **if** *test* **then** *PlanA* **else** *PlanB*, where *test* is a Boolean function that tests the state of the world. For example, we might have a plan that comprises two steps: a step for tossing coin a, and then a conditional step:

tossCoin(a)
if landsHeads(a) **then** goRight() **else** goLeft()

By nesting conditional steps, plans become trees.

We won't look at a conditional planning algorithm. Instead, we'll content ourselves with a few further observations.

A single plan will now have multiple paths through it and, strictly, a plan is not finished unless *all* paths lead to satisfaction of the goal. In other words, a conditional plan must reach a goal state regardless of which outcomes actually occur.

But, this leads to a problem. There can be an 'explosion' of paths. (A sequence of n disjunctive operators, each having just two alternative effects, gives 2^n paths.) We cannot in fact plan for every eventuality. Instead, we must focus on the most likely eventualities and plan for these. This requires that we use probability information to constrain the planning algorithm.

2 Execution Monitoring and Replanning

Imagine that an agent has built a plan using any of the ideas that we have discussed so far (partial-order planning, hierarchical planning, conditional planning, or some mixture of these). How can it now cope with unbounded indeterminacy?

While executing the plan, the agent can also engage in *execution monitoring* to determine whether the current state of the world is as the plan says it should be. Why might the world not be in the state that the plan says it should be? Perhaps execution of the previous action did not have its expected effects because it was fumbled; or perhaps some other agent has been executing actions that interfere with our agent's. In fact, there are two kinds of execution monitoring:

- *Action monitoring:* Before executing the next step, the agent uses its sensors to check that the preconditions of that step are, indeed, true (as the plan expects them to be).
- *Plan monitoring:* Before executing the next step, the agent uses its sensors to check that the preconditions for the entire remaining plan are true (i.e. it checks all preconditions, except those that are achieved by another remaining step in the plan). (Of course, in practice, you don't want your agent to spend too much time sensing and checking preconditions. You have to check just those that are important and readily-checked.)

Replanning occurs when something goes wrong. The agent invokes the planner again to come up with a revised plan to reach the goal. Often, the most efficient way to come up with the revised plan is to *repair* the old plan, i.e. to find a way from the current, unexpected state of the world back onto the plan.

3 Continuous Planning

In execution monitoring and replanning, there is a tighter integration of planning and execution. Rather than looking further at execution monitoring and replanning, we'll look at an even more general approach, known as *continuous planning*. It has a tight integration of planning and execution. It doesn't engage in identifiably separate planning episodes. It can begin execution of partial plans, before they are complete. And it can continuously formulate new goals, which means that it's plan may never be complete.

We can readily extend POP to be a continuous planner.[1] When we extended POP for hierarchical planning, all we needed to do was add a new way of refining plans. And the same is true here. The algorithm is just a loop, and each time round the loop we pick one of the plan refinement operators of which there are now 10 (excluding decomposition of abstract operators):

Make the initial plan, i.e. the one that contains only the Start and Finish steps.
while true
{ Choose one of the following:
1. Achieve an unachieved precondition by adding a new step.
2. Achieve an unachieved precondition using an existing step.
3. Protect a link by promotion.
4. Protect a link by demotion.
5. *New goal:* Add a new goal to the Finish step.
6. *Unsupported link:* If there is a causal link Start $\xrightarrow{c}$ a, where c is no longer true in Start, then remove the link. This prevents us from executing an action whose preconditions are false.
7. *Extending a causal link:* If $s_j \xrightarrow{c} s_k$, but there is an earlier step $s_i \prec s_j$ which could also achieve condition c, without introducing a new conflict, then remove $s_j \xrightarrow{c} s_k$ and insert $s_i \xrightarrow{c} s_k$. (This refinement allows us to take advantage of serendipitous events.)
8. *Redundant action:* If an action a is the source of no causal links, then remove a from the plan. It serves no purpose.
9. *Execute an unexecuted action:* If an action a (other than Finish) has its preconditions satisfied by Start, has no actions (other than Start) ordered before it and conflicts with no causal links, then remove a from the plan and execute it.
10. *Unnecessary historical goal:* If there are no unachieved preconditions and no actions (other than Start and Finish), then we have achieved the current goal set. Remove the goals, and await new ones.
}

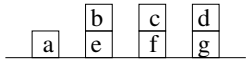
Here's a running example. To make the example more manageable, we will allow ourselves to use a *move* operator, which moves a block to another. This avoids the need to separately unstack and stack, so it makes the example less cluttered. For the same reasons, we're also going to omit all mention of the *armempty* precondition.

Here is the operator:

Op(ACTION: move(x, z),
PRECOND: clear(x) $\wedge$ clear(z) $\wedge$ on(x, y),
EFFECT: on(x, z) $\wedge$ $\neg$clear(z) $\wedge$ clear(y) $\wedge$ $\neg$on(x, y))

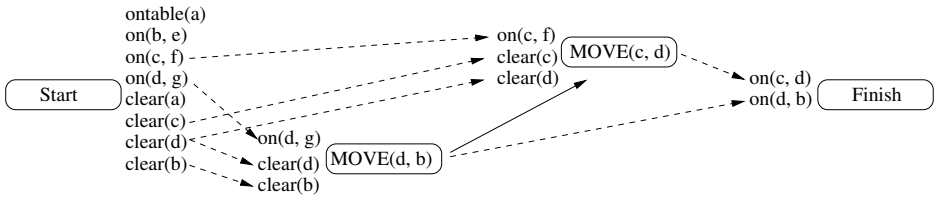
[1] This formulation of continuous planning, and the example, are based on section 12.6 in S.Russell and P.Norvig, *Artificial Intelligence: A Modern Approach* (2nd edn.), Prentice-Hall, 2003.

Suppose the start state is like this:

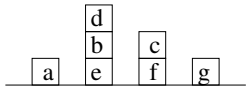


Suppose the goal is $on(c, d) \wedge on(d, b)$.

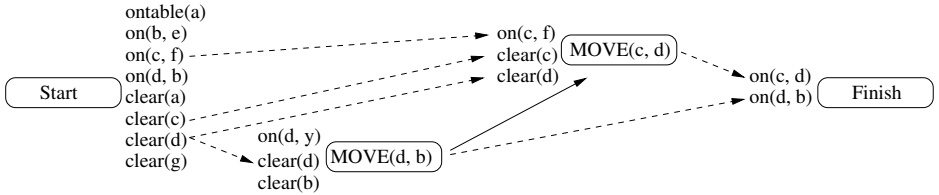
The agent starts to plan. We'll assume that it constructs the following complete plan, without doing any execution:



The agent is about to execute the first step of the plan. But before it can even choose this step, another agent intervenes! The other agent (quite helpfully in this case) moves d onto b. The world is now like this:



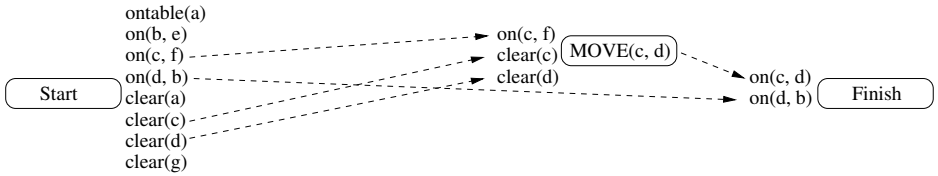
Our agent perceives this. It recognises that $clear(b)$ and $on(d, g)$ are no longer true in the current state, so it updates the effects of the Start step. This also means that causal links Start $\xrightarrow{clear(b)}$ $move(d, b)$ and Start $\xrightarrow{on(d,g)}$ $move(d, b)$ are invalid so they are removed (see *Unsupported link* in the algorithm):



The plan is now incomplete. It now has two unachieved preconditions, $on(d, y)$ and $clear(b)$.

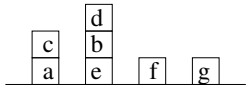
$move(d, b)$ was being used to achieve goal $on(d, b)$. But we now notice that, as a result of the 'helpful' interference of the other agent, an earlier step in the plan (in this case, Start) can achieve this goal. So we replace $move(d, b)$ $\xrightarrow{on(d,b)}$ Finish by Start $\xrightarrow{on(d,b)}$ Finish (see *Extending a causal link* in the algorithm).

And we can now remove action $move(d, b)$ from the plan entirely (see *Redundant action* in the algorithm):

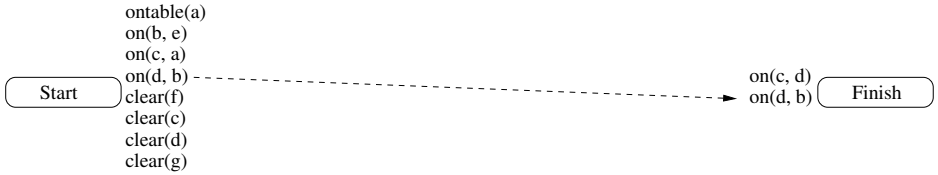


This time round the loop, the agent realises that *move*(c, d) can be executed (*Execute an unexecuted action* in the algorithm). The step will be deleted from the plan.

Unfortunately, the agent is clumsy. While executing the move of c to d, it drops c onto a, instead of d. The current state is therefore now as follows:

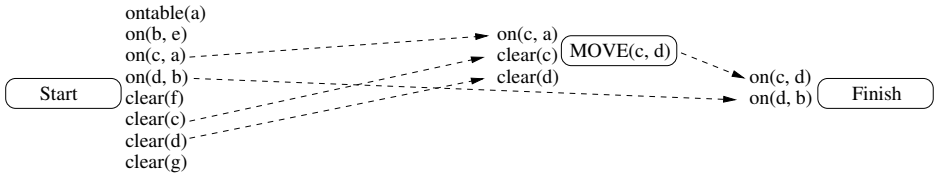


So the plan now looks like this:

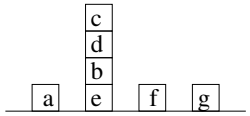


The plan is therfore still incomplete: there remains an unachieved precondition, *on*(c, d).

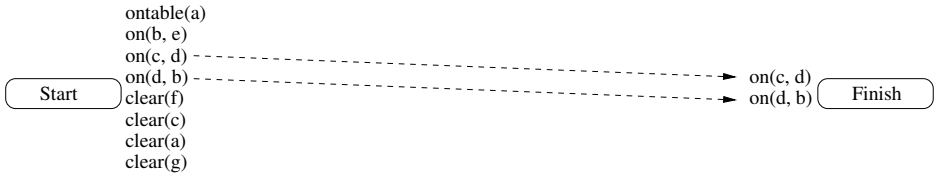
So the agent does some more planning (using goal achievement, as in normal POP), and obtains:



Again, *move*(c, d) is ready to be executed (see *Execute an unexecuted action*). Suppose this time it works correctly, so the new state of the world is:



The action is deleted from the plan, and the results of sensing the world are used to update the effects of the Start step, and these now directly achieve the preconditions of Finish:



We can now delete the two goal conditions (see *Unnecessary historical goal* in the algorithm). And we are done. Of course, in practice, new goals are always being formulated and added to the preconditions of Finish (see *New goal* in the algorithm), and so this process continues indefinitely.

The system we have just described is very flexible. And you can see how extensible it is: it is easy to add new ideas, simply by incorporating new plan refinement operators. For example, one nice idea is to bring in the idea of hierarchical decomposition. The idea might be to use abstract operators whenever possible and to defer decomposition so that it is only done immediately prior to execution time. The high level plan ensures that we have done enough thinking ahead. But deferral of decomposition means that we only come up with detailed actions when we know what the world is actually like.