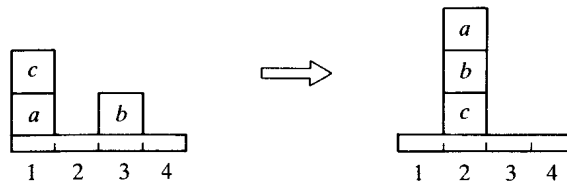


Chapter 17

Planning Basics

With the *planning* process, we think about the states, and the effects of actions, and put into a sequence of available actions to achieve a given effect.

Below shows an example of planning. It can be solved following the state space approach, but now we will see how to represent, and solve, this problem through explicit planning.



The key observation is that an action will change the current state of the “world”, thus causing it to move over to another state. For example, after taking the action of eating a lunch, the state of our stomach will transition from “I am hungry” to “I am stuffed.” .

On the other hand, an action will not change everything, it only changes part of a world. In the above case, technically, it only changes the content of our stomach.

Thus, a good representation should take care of this *local effect*. For example, we can represent a state as a list of relationships that are currently true.

Represent states

For example, for the blocks world, we can use the following two relations: `on(Block, Object)`, and `clear(Object)`.

While the former one is clear, the latter says that there is nothing on `Object`. This clearance relation is relevant, since before we move a block, it must have nothing on it.

Given the following situation,

```
c
a  b
-----
1 2 3 4
```

we can have the following representation:

```
[clear(2), clear(4), clear(b), clear(c),
on(a, 1), on(b, 3), on(c, a)].
```

Represent actions

Each available action is defined in terms of its precondition and its effects.

An action is defined as a three-piece package: 1) *precondition*: it must be satisfied before this action can take place; 2) *add-list*: it represents a collection of relations that the action is expected to establish; and 3) *delete-list*: it is just the other side of the coin, i.e., a bunch of relations to be falsified as a consequence of this action.

Technically, a precondition can be represented by `can(Action,Cond)`, meaning `Action` can happen if `Cond` is true.

As a result of taking an action, we have to do `adds(Action,AddRelS)`, and `deletes(Action,DelRelS)`, where `AddRelS` and `DelRelS` are lists of relations to be added or deleted, respectively.

For the block world, the only action is

```
move(Block, From, To),
```

namely, a block `Block` is to be moved from the position `From` to the position `To`.

For the block world, we can specify this action as follows:

```
can(move(Block,From,To),
    [clear(Block),clear(To),on(Block,From)]) :-
    is_block(Block),object(To), object(From),
    To\==Block, From\==To,Block\==From.

adds( move(X,From,To), [on(X,To),clear(From)]).
deletes(move(X,From,To), [on(X,From),clear(To)]).

object(X) :-place(X)
;
block(X).
```

The initial configuration of the blocks world can be represented as follows:

```
block(a).  
block(b).  
block(c).
```

```
place(1).  
place(2).  
place(3).  
place(4).
```

```
state1([clear(2),clear(4),clear(b),clear(c),  
        on(a, 1),on(b, 3),on(c, a)]).
```

Finally, we also have to state a goal as a list of relations: [on(a, b), on(b, c)].

A definition of this nature not only gives the initial configuration, but, together with that of actions, actually specify a state space. Thus, such a definition is also called a *planning space*.

Homework: 17.1, 17.2.

How to get there?

Given the initial state for the blocks world, let the goal be `on(a,b)`. The planner has to find a plan, i.e., a sequence of actions, to achieve that goal. The question is *how*?

A typical planner might work as follows:

1. Find an action that *achieves* `on(a, b)`. Since this is not in the current state, it can only be added on with an action `move(a, From, b)` for a `From` position.
2. To enable this action, we have to ensure its precondition is met. Thus, we check the associated `can` relation and find out the conditions are `[clear(a), clear(b), on(a, From)]`. We then notice that only the first one is not true in the initial state. Now, we will try to get this piece done.

3. Again, we look at the adds relation and find out that to clear the block a, the action should be `move(Block, a, To)`, and its associated precondition is `[clear(Block), clear(To), on(Block, a)]`. They are satisfied in the initial state if we set `Block` to be `c` and `To` to `2`, or `4`.

4. Now, we can execute `move(c, a, 2)` to get into a new state by deleting anything this action deletes, and adding in anything it adds. This new state turns out to be the following:

`[clear(a), clear(b), clear(c), clear(4),
on(a,1), on(b,3), on(c,2)]`.

5. At this moment, the action expected in step 2, i.e., `move(a,1,b)`, can be executed, which achieves the final goal.

Thus, the plan is `[move(c, a, 2), move(a, 1, b)]`.

Mean-ends analysis

This style of planning is generally referred to as *means-ends analysis*, namely, find out the actions that meet the goal. In the previous case, a plan is found without any backtracking. This is certainly not true in more complicated situation. In reality, we often see combinatorial complexity, and serious search in planning.

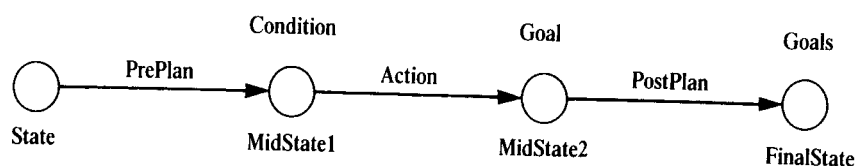
Below collects the general principles of planning via means-ends analysis: Start from a state `State`, to solve a list of goals `Goals`, all of which are true in `FinalState`, do the following:

1. If all the `Goals` are true in `State`, then `State` is the same as `FinalState`. Otherwise do the following:
2. Pick a goal `Goal` in `Goals` and find an action `Action` that will add this goal to a state.

3. To apply `Action`, we have to satisfy its precondition `Condition`, which requires a `PrePlan` which will take us from `State` to `MidState1`, where `Condition` is met.

4. We now can apply `Action` to `MidState1`, which brings us to another state `MidState2`, where `Goal` is true.

5. we now have to find a plan `PostPlan`, which is to solve the other goals of `Goals` in `MidState2`, to make them all true in `FinalState`.



Let's check out the program `planblock.pl`

Homework: Exercise 17.3.

Goal protection

The block world is quite complex because it is quite flexible: In each step, there are several choices for the planner to make as which goal we should satisfy next, and, in this case, where the next block should be placed. It seems that all of them make sense in terms of the means-ends principle.

For example, if we start with the original configuration of the block world, and set the goal to be `[on(a,b),on(b,c)]`, and run the current planner, we will get a plan back, consisting of 7 steps: `move(b,3,c),move(b,c,3),move(c,a,2),move(a,1,b),move(a,b,1),move(b,3,c),move(a,1,b)`; while the shortest possible plan takes only 3 steps.(?)

Look at the plan itself. What do you see?

Where is the problem?

The reason for the unnecessary steps is that the planner pursues different goals at different stages, which might be in conflict with each other.

For example, out of the two goals, it picks up `on(b, c)` to resolve first, thus, `move(b,3,c)`.

Now, it has to resolve `on(a, b)`, which can be achieved by `move(a, From, b)`.

To do that, we have to satisfy its precondition, including `clear(a)`. Before doing that, we have to clear off `c`. Thus, `move(b,c,3)`. now, `on(b, c)` no long holds.

In other words, to achieve the second goal, the planner has destroyed the first goal.

So, we have `clear(c)`, so we can do `move(c, a, 2)`. Once it is done, we have `clear(a)` so that the next move `move(a,1,b)` is possible. Now, the second goal, `on(a, b)` is achieved.

We again have to restore the first part, i.e., `on(b, c)`. To do that, we have to do `move(a, b, 1)` to ensure `clear(b)`, which again destroys the first goal `on(a, b)`.

Then, the planner does `move(b,3,c)` to reestablish `on(b,c)`.

Now, believe or not, we have to restore the second goal again.

But, this time, we are lucky in the sense that we can do it w/o destroying the second.

Waste of time

What happened is that that, after easily achieving the second goal, i.e., `on(b,c)`, it immediately destroys it to achieve the first goal, `on(a, b)`. Once `on(a, b)` is established, it is destroyed again to reestablish `on(b,c)`! We are lucky for this example that this plan actually ends somewhere.

It is not hard to imagine such hectic, unorganized, behavior will lead to total failure in some cases, when getting into an infinite loop.

For example, given the goal `[clear(2),clear(3)]`, it will get into the following infinite loop, i.e., `move(b,3,2)` to clear 3, `move(b, 2,3)` to clear 2, then `move(b,3,2)` to clear 3, then again `move(b, 2,3)` to clear 2, Each of the step achieves one goal, and the next one destroys it. Thus, nothing is really achieved at all.

What to do?

It is clear that a better planner should preserve, or protect, those already achieved goals. This can be done by collecting those achieved goals, and, whenever an action is taken, check and make sure that those achieved goals will not be destroyed.

Given such a plan, the troublesome goal of `[clear(2),clear(3)]`, will be solved by `move(b,3,2)`, then `move(b,2,4)`.

The protecting code

```
plan(InitialState,Goals,Plan,FinalState):-
    plan(InitialState,Goals,[],Plan,FinalState).

plan(State,Goals,_,[],State):-
    satisfied(State,Goals).
plan(State,Goals,Protected,Plan,FS):-
    conc(PrePlan,[Action|PostP],Plan),
    select(State,Goals,Goal),
    achieves(Action,Goal),
    can(Action,Condition),
    preserves(Action,Protected),
    plan(State,Condition,Protected,PrePlan,MidS1),
    apply(MidS1,Action,MidS2),
    plan(MidS2,Goals,[Goal|Protected],PostP,FS).

%nothing in the goals is deleted
preserves(Action,Goals):-
    delete(Action,Relations),
    not((member(Goal,Relations),
        member(Goal,Goals))).
```

Let's run planProtect.pl to check it out.

Further improvement

The strategies we have obtained for planning so far only use basic search ideas such as DFS. Thus, they are generally inefficient. To do a better job, we might want to bring in some better search strategies, such as the best-first search as we discussed before.

Thus we have to provide a *successor* relation between states, some *goal* nodes via a `goal(Node)` relation, a heuristic function via `h(Node, Estimate)`, and a start node.

Homework: Run the `planBestFirstSearch.pl` to find out solution for 1) `[on(a, b), on(b,c)]`, 2) `[clear(2), clear(3)]`, and 3) some goal you find interesting.