MCS 380/590 Intro To Artificial Intelligence
Dr. William C. Regli
Drexel University Fall 2000
Homework Assignment Solutions #6
Artificial Intelligence (Russell / Norvig)

11.1 The operators we need are the four from page 346

Op (ACTION: $RightShoe$, PRECOND: $RightSockOn$, EFFECT: $RightShoeOn$)
Op (ACTION: $RightSock$, EFFECT: $RightSockOn$)
Op (ACTION: $LeftShoe$, PRECOND: $LeftSockOn$, EFFECT: $LeftShoeOn$)
Op (ACTION: $LeftSock$, EFFECT: $LeftSockOn$)

plus the following two new ones (no preconditions):

Op (ACTION: Hat, EFFECT: $HatOn$)
Op (ACTION: $Coat$, EFFECT: $CoatOn$)

On page 348 it states there are 6 total-order plans for this problem. Each of these plans has four steps and thus five arrow links The next step, Hat could got at any of these five locations, giving us 6 x 5 = 30 total order plans, each with five steps and six links. Then the final step, $Coat$, can go in any one of these 6 positions, giving us 30 x 6 = 180 possible linearizations of total ordered plans.

```
              /----->Coat-----------\
             /                       \
            /--->RghtSock--->RghtShoe>\
start -                                ->finish
            \--->LeftSock--->LeftShoe>/
             \                       /
              \----->Hat------------/
```

11.3 Only (g), interleaved v non-interleaved affects completeness: if you don't consider interleaved plans ... then you won't find all solutions. Depending on the details of the algorithm, a strict debugging planner might also be incomplete, if you don't include enough debugging operations. (But then it is always possible to build an incomplete planner by making a buggy version of any approach.) In terms of efficiency, we note that much of the current work in planning involves plan-space, regressive, least commitment, partial order planners. In general, it is hard to comment on the efficiency effects of any one factor in isolation.

11.3a situation space v plan space (p 345-346)

"Situation space" means the planner searches through he space of situations or states of the world, trying to find a totally-ordered plan through this space from the start to the goal. (However, it does not mean that planning must always proceed in that order; we could start at the start state, or the goal state, or do bi-directional search.) "Plan space" means the search is through the space of possible (partially ordered) plans. The planner tries to find a plan that achieves the goal by modifying or extending an incomplete plan.

11.3b progressive v regressive (p 345,365)

"Progressive" means the planner starts in the start state and tries to move towards the goal. "Regressive" means starting in the goal state and moving backwards. The choice of which to use depends on the number of applicable operators going forward vs. the number of relevant operators going backwards. The idea is to minimize the branching factor.

11.3c refinement v debugging (p 345)

"Refinement" planners start with a most-general initial partial plan, and extend it by adding new steps and constraints, and by backtracking to previous choice points. "Debugging" planners not only extend partial plans,; they also modify plans by deleting and reordering steps and/or constraints. They may or may not backtrack.

11.3d least commitment v more commitment (p 346)

This is somewhat vague, all encompassing term, but the idea is that a "least commitment" planner can represent abstract plans w/o having to make certain low-level implementation choices. For example, being able to represent that two steps are part of the plan w/o saying which come first, or being able to say that a $Go(x, Supermarket)$ step is part of the plan w/o deciding what the origin (x) is for the trip to the supermarket.

11.3e bound v unbound variables (p 357-358)

It is a mistake in the text when they talk about "instantiated" v "uninstantiated" in the text. Either way, it is a kind of least/more commitment issue, having to do with whether we can deal w/unbound or uninstantiated variables such as the x in $Go(x, Supermarket)$, or whether the planner insists that we make a concrete choice for x when we add the step.

11.3f total order v partial order (p 346)

Total-order planners represent a plan as a sequence of steps; partial-order planners represent a plan as a set of steps with some constraints about which steps have to occur before which others.

11.3g interleaved v non-interleaved (p 363)

An "non-interleaved" planner insists that when there are two sub-goals, the steps to achieve one sub-goal must come either all before or all after the steps to achieve the other. An "interleaved" planner has no such requirement.

11.3h unambiguous preconditions v ambiguous preconditions

this is a distinction that occurs in the planning literature. The idea is that some planners allow causal links that are ambiguous: they may or may not achieve the desire condition, depending on details of what other steps are taken. The TWEAK planner was ambiguous in this way. All the planners covered in the text have unambiguous preconditions.

11.3i systematic v unsystematic:

again, this is a distinction that is in the literature but not in the text. A systematic planner is one that never considers the same plan twice. A non-systematic planner has a search strategy that allows it to arrive at the same plan twice (usually by following different paths in the plan-space). The "S" in the "SNLP" planner stands for systematic.

11.4 STRIPS

You can use the strips operators on p359-360. This initial state is:

$On(B, Table \wedge On(C, A) \wedge On(A, Table) \wedge Clear(B) \wedge Clear(C)$

The goal state is:

$On(A, B) \wedge On(B, C)$

The reason why it is an anomaly for a non-interleaved planner for the following reasons. Their are two sub-goals; suppose we decide to work on $On(A, B)$ first. We can clear C off of A and then move A on to B. But then there is no way to achieve $On(B, C)$ w/o undoing the work we have done. Similarly, if we work on the sub-goal $On(B, C)$ first we can immediately achieve it in one step, but then we have to undo it to get A on B.

Now we'll show how things work out with an interleaved planner such as POP. Since $On(A, B)$ isn't true in the initial state, there is only one way to achieve it: $Move(A, x, B)$, for some x. Similarly, we also need a $Move(B, x', C)$ step, for some x'. Now let's look at the $Move(A, x, b)$ step. We need to achieve its precondition $Clear(A)$. We could do that either with $Move(b, A, y)$ or with $MoveToTable(b, A)$. Let's assume we choose the latter. Now if we bind b to C, then all of the preconditions for the step $MoveToTable(C, A)$ are true in the initial state, and we can add causal links to them. We then notice that there is a threat: the MOve (B, x', C) step threatens the $Clear(C)$ condition that is required by the $MoveToTable$ step. We can resolve the threat by ordering $Move(B, x', C)$ after the $MoveToTable$ step. Finally, notice that all the preconditions for $Move(B, x', C)$ are true in the initial state. Thus, we have a complete plan with all the preconditions satisfied. It turns out there is a well-ordering of the three steps:

$MoveToTable(C, A)$
$Move(B, Table, C)$
$Move(A, Table, B)$

11.5 Time Machine

Time only has an affect on causal links. When we say step S_1 achieves precondition c of step S_2 (ie., $S_1 \xrightarrow{c} S_2$), we also add the temporal ordering constraint $S_1 \prec S_2$. Now suppose the time machine gives us an operator for sending an object x back in time to time t:

Op (ACTION: $SendBackInTimeTo(x, t)$, PRECOND: $At(TimeMachine, l, Now) \wedge At(x, l, Now)$, EFFECT: $At(x, l, t)$))

Note the necessity of a time, t argument on predicates such as At, and the introduction of the time Skolem constant, Now. One change to the planner would be handling this. The other change would be to eliminate the $S_1 \prec S_2$ link for steps involving $SendBackInTimeTo$. This is likely to be a controversial exercise w/many different answers, based on the many definitions of what a time machine exactly does.

12.1 It is important to understand what constitutes a correct implementation of an operator (p373). It must be a consistent plan where all preconditions and effects are accounted for. So the first thing that needs to be done for this problem is to decide on the preconditions and effects of the high-level operators. For $ObtainPermit$, assume the precondition is owning land, and the effect is having a permit for that piece of land. For $HireBuilder$, the precondition is having the ability to pay, and the effect is having a signed contract in hand.

One possible decomposition for $ObtainPermit$ is the three step sequence $GetPermitForm$, $FillOutForm$, and $GetFormApproved$. There is a causal link with the condition $HaveForm$ b/w the first two, and one with the condition $HaveCompletedForm$ b/w the last two. Finally the $GetFormApproved$ step has the effect $HavePermit$. This is a valid decomposition.

You get the idea.

12.2 One simple answer to this is to give the conditions that are preconditions to the high-level operator a "high" criticality level, and the others ' "low" level. This is sufficient to mimic the effects of one level of hierarchical decomposition.

12.3 Consider the problem of building two adjacent walls of the house. Mostly these sub-plans are independent, but they must share the step of putting a common post at the corner of the two walls. If that step was not shared, we would end up with an extra beam, and two unattached walls.

Note that tasks are often decomposed specifically so as to minimize the amount of step sharing. For example, one could decompose the house building task into sub-tasks such as "walls" and "floors." However, real contractors don't do it that way. Instead they have "rough walls" and "rough floors" steps, followed by a "finishing" step.

12.4 obvious, use your imagination.

12.7 If we just wanted to solve the specific problem of grabbing three oranges, it would be easy: define operators for $Grab3Oranges$ and $Carry3OrangesHome$. The hard part is making it more general. We would like to have operators $Grab(o)$ for grabbing an object, $PutIn(o, c)$ for putting an object in a container, $Go(x, y)$, for going from one location to another, and $Pay(o, s)$ for paying for an object at a store.

We could start with the $Carry$ operator from p383 as a model, but note that we don't want to have to decide whether to Go or to $Carry$; we just want to Go and have the carrying happen automatically. We can do that with a Go operator that transports all objects that are grabbed, or are inside containers that are grabbed:

Op (ACTION: $Go(x, y)$, PRECOND: $At(Agent, x)$, EFFECT: $At(Agent, y), \neg At(Agent, x) \wedge$ $\forall\, i\ Item(i) \Rightarrow (At(i, y) \wedge \neg At(i, x))$**when**$Grabbed(Agent, i) \forall\, c\ Container(c) \Rightarrow$ $\forall\, i\ \Rightarrow (At(i, y)\ \wedge\ \neg At(i, x))$**when**$Grabbed(Agent, c)\ \wedge\ In(i, c)))$

Now we need the $Grab$ and $PutIn$ operators:

Op (ACTION: $Grab(o)$, PRECOND: $At(Agent, x)\ \wedge\ At(o, x)$, EFFECT: $Grabbed(Agent, o)))$

Op (ACTION: $PutIn(o, c)$, PRECOND: $Grabbed(Agent, o) \wedge Grabbed(agent, c) \wedge$ $Container(c)$, EFFECT: $In(o, c)\ \wedge\ \neg Grabbed(Agent, o)))$

Finally, we need the Pay operator, which is just the Buy operator of page 387:

Op (ACTION: $Pay(o, store$, EFFECT: $Have(o)\ \wedge\ Cash \leftarrow Cash - Price(o, store)))$

12.10 We need one operator, $Assign$, which assigns the value in the source register (or variable if you prefer, but the term "register" makes it clearer that we are dealing with a physical location) sr to the destination register dr:

Op (ACTION: $Assign(dr, sr)$, PRECOND: $Register(dr) \wedge Register(sr) \wedge$ $Value(dr, dv) \wedge Value(sr, sv)$, EFFECT: $Value(dr, sv) \wedge \neg Value(dr, dv)))$

Now suppose we start in an initial state with:

$Register(R_1\ \wedge\ R_2\ \wedge\ Value(R_1, R_2)\ \wedge\ Value(R_1, V_1)\ \wedge\ Value(R_2, V_2)$

and we have the goal:

$Value(R_1, V_2) \wedge Value(R_2, V_1)$

Unfortunately, there is no way to solve this as is. We either need to add an explicit $Register(R_3)$ condition to the initial state we need a way to create new registers. That could be done with an operator for allocating a new register:

Op (ACTION: $Allocate(r)$, EFFECT: $Register(r)$)

Then the following sequence of steps constitutes a valid plan:

$Allocate(R_3)$
$Assign(R_3, R_1)$
$Assign(R_1, R_2)$
$Assign(R_2, R_1)$