Hierarchical Planning

One of the main ways of coping with complexity is hierarchical abstraction. You can see this idea in operation from military command structures to well-written software. In an abstraction hierarchy, a problem or task at one level is reduced to a *small* number of subproblems or tasks at the next level. All going well, the cost of arranging subproblems or subtasks so that they achieve the desired effects of their parent problem or task will be small. This is the computational benefit of abstraction.

In this lecture, we look at ways of bringing more abstraction into A.I. planning.

1 Hierarchical Approximation

Hierarchical approximation is an early form of hierarchical planning.

In hierarchical approximation, we assign *criticality levels* to STRIPS preconditions. These are simply numbers which signify how critical a precondition is: critical ones will usually be ones that are harder to achieve. For example, we might assign criticality levels as follows:

Op(	ACTION:	stack(x, y),
	PRECOND:	1: holding(x)
		2: clear(y)
	EFFECT:	$\neg$clear(y) $\wedge$ $\neg$holding(x) $\wedge$
		armempty $\wedge$ on(x, y) $\wedge$ clear(x))

Op(	ACTION:	unstack(x, y),
	PRECOND:	0: armempty
		2: clear(x)
		3: on(x, y)
	EFFECT:	$\neg$on(x, y) $\wedge$ $\neg$armempty $\wedge$
		$\neg$clear(x) $\wedge$ holding(x) $\wedge$ clear(y))

We revise the POP algorithm to take these numbers into account. We arrange the search so that we first build a plan in which only preconditions of highest criticality are achieved. Only once this has been built do we successively deal with preconditions of lower criticality levels.

In terms of our example, we would first build a plan in which the only unachieved preconditions that we solve would be the ones whose criticality level is 3. Of course, the result is not necessarily a complete plan: some (even many) of the preconditions (all those with criticality < 3) will remain unachieved. Once this has been done, we then tackle all preconditions at level 2. And so on, until a complete plan has been built.

If you look back at the description of POP, you will see that nothing was said about the order in which unachieved preconditions would be achieved. All we said was: since all unachieved preconditions must eventually be achieved, they are not to be treated as alternatives. Now, using criticality levels, we have a way of deciding the order in which to tackle them.

This is a quite crude approach to providing control information to the planning algorithm, and it suffers the limitation that assignment of static criticality numbers does not reflect the fact that criticality might depend on context.

2 Hierarchical Decomposition

Of far greater significance, however, is another form of hierarchical planning, known as *hierarchical decomposition*.

In hierarchical decomposition, there are two kinds of actions and, correspondingly, two kinds of operators.

- *Primitive actions* are ones the robot can directly execute. They are specified using 'normal' STRIPS operators.
- *Abstract actions* are ones that are too high-level to be immediately executed; they need to be decomposed into lower-level actions. They are specified using *abstract operators*. Abstract operators are like macros. Each abstract operator, as well as specifying preconditions and effects, also specifies how it can be decomposed into a precompiled plan of lower-level steps. These lower-level steps might themselves be abstract actions, in which case they will require further decomposition. Or they might be primitive actions.

In planning, the process of decomposition will continue until only primitive actions remain in the plan.

The resulting plan, with all its levels of decomposition, is sometimes called a *hierarchical task network* or *HTN*. Planners that generate plans purely by decomposition are called HTN planners.

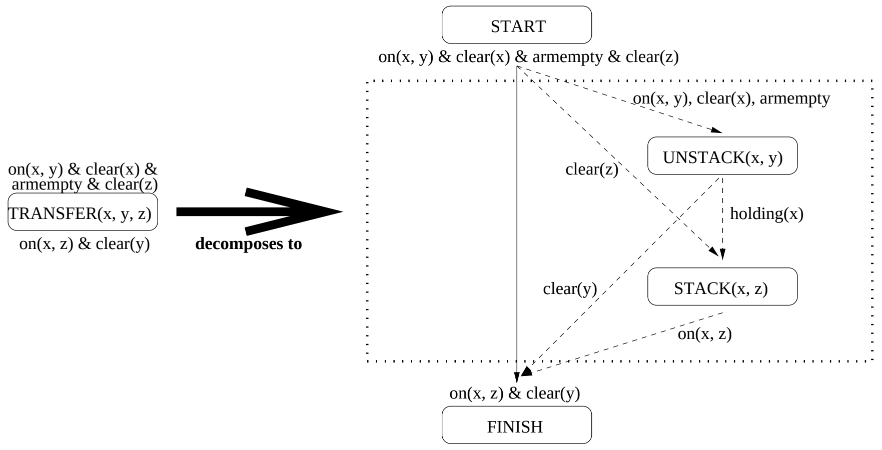
However, we'll present some of the details of a planner that seamlessly integrates partial-order planning (based on POP from the previous lecture) and hierarchical decomposition (based on HTN planning).

2.1 Abstract Operators

First, we look at how to represent abstract actions using abstract operators.

Recall that our robot is capable of executing four (primitive) actions: stacking, unstacking, picking up and putting down. Consider the action of *transferring* a block from one non-empty stack of blocks to another. This is not an action that the robot can directly execute. In terms of primitive actions, it involves unstacking the block and stacking it again at its destination. This is not a killer example, but it does allow us to see what a (simple) abstract operator would look like.

Here it is pictorially:



Here it is more formally:

```
Op( ACTION:     transfer($x, y, z$),
    PRECOND:    on($x, y$) $\wedge$ clear($x$) $\wedge$ clear($z$) $\wedge$ armempty,
    PRIMARY
    EFFECT:     on($x, z$),
    SECONDARY
    EFFECT:     clear($y$),
    DECOMP:     Plan( STEPS: {  S1: Op( ACTION:  Start,
                                        EFFECT:  on($x, y$) $\wedge$ clear($x$) $\wedge$
                                                 clear($z$) $\wedge$ armempty),
                                S2: 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($y$)),
                                S3: Op( ACTION:  stack($x, z$),
                                        PRECOND: clear($z$) $\wedge$ holding($x$),
                                        EFFECT:  $\neg$clear($z$) $\wedge$ $\neg$holding($x$) $\wedge$
                                                 armempty $\wedge$ on($x, z$) $\wedge$ clear($x$)),
                                S4: Op( ACTION:  Finish,
                                        PRECOND: on($x, z$) $\wedge$ clear($y$))},
                  ORDERINGS: {  $S1 \prec S2, S1 \prec S3, S1 \prec S4, S2 \prec S3, S2 \prec S4, S3 \prec S4$},
                  LINKS: {      $S1 \xrightarrow{on(x,y)} S2, S1 \xrightarrow{clear(x)} S2, S1 \xrightarrow{armempty} S2, S1 \xrightarrow{clear(z)} S3,$
                                $S2 \xrightarrow{holding(x)} S3, S2 \xrightarrow{clear(y)} S4, S3 \xrightarrow{on(x,z)} S4$}))
```

So we have an abstract action, *transfer*, for which there is a precompiled plan (what this action decomposes into).

- The preconditions of the *transfer* operator are the *external preconditions* of its decomposition. That is, for any operator in the decomposition, external preconditions are ones that the plan itself does not make true: within the decomposed plan, it is only the Start operator that makes them true.
- The *external effects* of the *transfer* operator are the effects of all the operators in the plan that are not negated by other later operators. But we want to distinguish two kinds of effect:
  - *Primary effects:* These are the external effects that you would use this abstract operator for. If your goal was to get x onto z, then you might use this operator.
  - *Secondary effects:* These are additional, incidental external effects of the plan. If your goal was to clear a block, you probably wouldn't want to do it by using this operator. There are easier ways to clear a block.
- The decomposition of the *transfer* operator is a POP plan. Let me repeat two points which are true in general, but not illustrated by this example.
  - First, note that the POP plan may itself contain further abstract actions, which would then themselves need to be decomposed. In this example, it only contains primitive actions.
  - Second, POP plans are, of course, in general partially-ordered. This example is, in fact, totally ordered.

In fact, I've simplified!

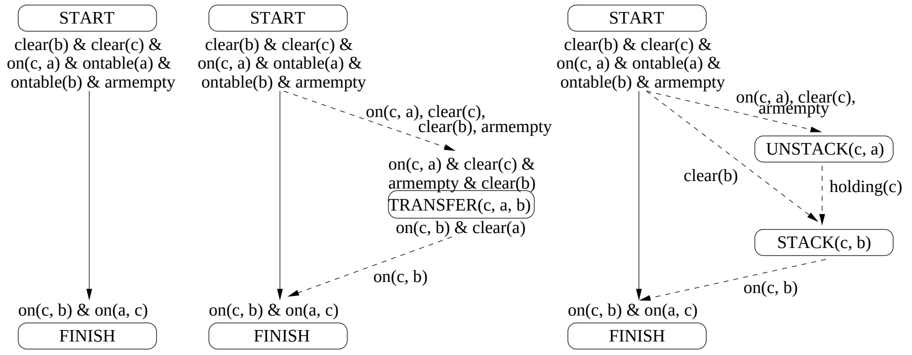
It is possible to have an operator with more than one decomposition. For example, you might have an abstract action called *travelCorkDublin()*. There could be several precompiled plans for achieving this, involving different modes of transport, for example. The different decompositions might each have different external preconditions and external effects. For example, one decomposition might involve rail travel, for which the preconditions might be to be in Cork and to have 50 euro; another might involve flying, for which the preconditions might be to be in Cork, to have 150 euro and to have less than 20kg of luggage. The precondition of the abstract operator as a whole would be the *intersection* of the external preconditions of each decomposition. Similarly, the effects of the abstract operator would be the intersection of the external effects of each decomposition.

2.2 Modifying POP for Abstract Operators

In high-level terms, modifying POP is easy. We simply extend the list of plan refinement operators. Where previously, there were four ways to refine a plan, now there are five, as summarised below:

```
Make the initial plan, i.e. the one that contains only the Start and Finish steps.
while the plan is not a solution plan
{   Choose one of the following:
    1. Achieve an unachieved precondition by adding a new step
       (using either a primitive or abstract operator);
    2. Achieve an unachieved precondition using an existing step
       (whether it be primitive or abstract);
    3. Protect a link by promotion;
    4. Protect a link by demotion;
    5. Choose an abstract action in the plan, choose one of its decompositions
       and replace the abstract action with the decomposition
       (suitably instantiated through unification and suitably hooked up with
       new ordering and causal links).;
}
```

So, imagine that we are faced with the same planning problem that we had in the previous lecture. This is shown leftmost. Maybe we then choose to use the abstract *transfer* operator to move c to b. This is shown in the centre. And then we decompose the *transfer* operator, i.e. we replace the single step by its precompiled plan. This is sketched rightmost:


START
clear(b) & clear(c) & on(c, a) & ontable(a) & ontable(b) & armempty
on(c, b) & on(a, c)
FINISH
START
clear(b) & clear(c) & on(c, a) & ontable(a) & ontable(b) & armempty
on(c, a), clear(c), clear(b), armempty
on(c, a) & clear(c) & armempty & clear(b)
TRANSFER(c, a, b)
on(c, b) & clear(a)
on(c, b)
on(c, b) & on(a, c)
FINISH
START
clear(b) & clear(c) & on(c, a) & ontable(a) & ontable(b) & armempty
on(c, a), clear(c), armempty
UNSTACK(c, a)
clear(b)
holding(c)
STACK(c, b)
on(c, b)
on(c, b) & on(a, c)
FINISH

There are some subtleties. And here's an overview of them:

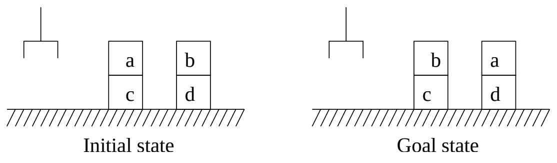
- Suppose you are replacing an abstract operator by its decomposition. Suppose that one of the steps in the decomposition is the same as one of the steps in the rest of the plan. For example, suppose the decomposition (after unification and instantiation) contains a step *unstack*(c, a) and and suppose that the rest of the plan contains this step already. Now you have a choice (and this implies search). It may be possible to *reuse* the existing step, instead of inserting a second *unstack*(c, a) step.

- How do you hook up ordering constraints? Suppose $S \prec a$, where a was the original abstract operator. The simplest solution is to insert $S \prec s$ for every s in the decomposition. This can be achieved by replacing every constraint of the form $\text{Start} \prec s$ in the decomposition by $S \prec s$. A similar story applies to constraints of the form $a \prec S$: this time you replace $s \prec \text{Finish}$ by $s \prec S$. In fact, what I have said here sometimes leads to more ordering constraints than are strictly necessary. It might be that a step had to precede the abstract operator, but does not need to precede every step within the decomposition. Some work has gone on to try to deal with this.
- How do you hook up the causal constraints? Suppose $S \xrightarrow{c} a$ was a causal link in the original plan, where a again is an abstract operator. Add $S \xrightarrow{c} s$ wherever there is a link $\text{Start} \xrightarrow{c} s$ in the decomposition. A similar story applies to constraints of the form $a \xrightarrow{c} S$: this time you replace $s \xrightarrow{c} \text{Finish}$ by $s \xrightarrow{c} S$.
- There is now the potential for new conflicts to have arisen. So you need to check for clobbering between the newly-added steps of the decomposition and the rest of the plan. Resolve any threats by promotion or demotion, as usual.

3 Closing Remarks

HTN Planning is very successful in practice, especially in systems with human users. So the integration of partial-order planning and HTN planning is very promising.

There is also the obvious potential to integrate learning. After a plan has been constructed, it makes sense to create an abstract operator from that plan and store it for future use. For example, suppose you have built a plan to solve the following problem:



Your plan probably involves unstacking a from c, then putting a on the table, unstacking b from d, then stacking b on c, then picking a up from the table and stacking it on d. What you want to do is *generalise* this plan, essentially by consistently substituting variables for constants so that it now refers to unstacking u from v, putting u on the table, etc, etc.

The resulting abstract operator can then be used in future plans whenever *swapping* is needed.

(I can't stop myself from mentioning that case-based reasoning is relevant too. Remember CBR is a way of allowing reuse of previous problem-solving episodes. But there's no time to say more.)

Exercise (Based on past exam question)

Assume that a household robot has the following repertoire of STRIPS-style operators:

Op(ACTION:	buyFish,
EFFECT:	haveFish)
Op(ACTION:	washHands,
EFFECT:	handsClean)
Op(ACTION:	layTable,
PRECOND:	handsClean,
EFFECT:	tableLaid)
Op(ACTION:	bakePotatoes,
PRECOND:	havePotatoes ∧ ovenHot,
EFFECT:	¬havePotatoes ∧ haveBakedPotatoes)
Op(ACTION:	heatOven,
EFFECT:	ovenHot)
Op(ACTION:	filletFish,
PRECOND:	haveFish ∧ handsClean,
EFFECT:	¬haveFish ∧ ¬handsClean ∧ haveBonedFish)
Op(ACTION:	bakeFish,
PRECOND:	ovenHot ∧ haveBonedFish,
EFFECT:	¬haveBonedFish ∧ haveBakedFish)

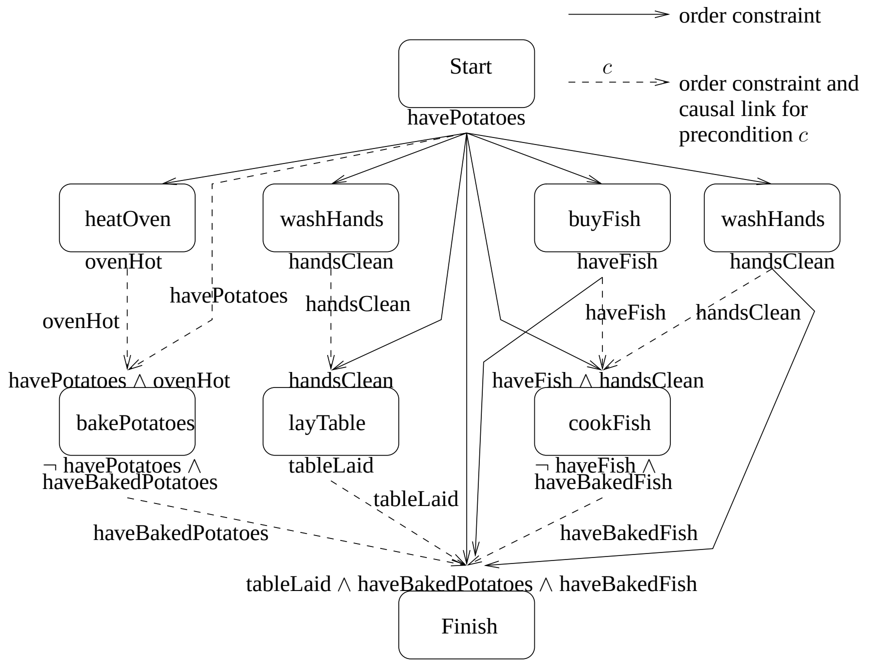
There is also an *abstract operator*, cookFish. Here is its full specification, including its decomposition:

Op(ACTION:	cookFish,		
PRECOND:	haveFish ∧ handsClean,		
EFFECT:	¬haveFish ∧ haveBakedFish,		
DECOMP:	Plan(STEPS: { S_1 : Op(	ACTION:	Start,
		EFFECT:	haveFish ∧ handsClean),
	S_2 : Op(	ACTION:	filletFish,
		PRECOND:	haveFish ∧ handsClean,
		EFFECT:	¬haveFish ∧ ¬handsClean ∧ haveBonedFish),
	S_3 : Op(	ACTION:	heatOven,
		EFFECT:	ovenHot),
	S_4 : Op(	ACTION:	bakeFish,
		PRECOND:	ovenHot ∧ haveBonedFish,
		EFFECT:	¬haveBonedFish ∧ haveBakedFish),
	S_5 : Op(	ACTION:	Finish,
		PRECOND:	haveBakedFish)},
	ORDERINGS: {	$S_1 \prec S_2$, $S_1 \prec S_3$, $S_1 \prec S_4$, $S_1 \prec S_5$, $S_2 \prec S_4$, $S_2 \prec S_5$, $S_3 \prec S_4$, $S_3 \prec S_5$, $S_4 \prec S_5$},	
	LINKS: {	$S_1 \xrightarrow{\text{haveFish}} S_2$, $S_1 \xrightarrow{\text{handsClean}} S_2$, $S_2 \xrightarrow{\text{haveBonedFish}} S_4$, $S_3 \xrightarrow{\text{ovenHot}} S_4$, $S_4 \xrightarrow{\text{haveBakedFish}} S_5$}))	

The diagram shows an incomplete plan that could be built by the POP planner.

- Redraw this plan, replacing the cookFish operator by its decomposition. You will need to incorporate the steps of the decomposition into the plan and install appropriate ordering constraints and causal links. (**Hint:** In the exam, students lost marks by not taking into account all the subtleties covered in section 2.2.)

- Explain **in detail** what you have done, especially highlighting any choices that you faced, the decisions you made, and the reasons for your decisions. (**Hint:** In the exam, students lost marks by not writing enough.)



order constraint
c
order constraint and causal link for precondition c
Start
havePotatoes
heatOven
washHands
buyFish
washHands
ovenHot
handsClean
haveFish
handsClean
ovenHot
havePotatoes
handsClean
haveFish
handsClean
havePotatoes ∧ ovenHot
handsClean
haveFish ∧ handsClean
bakePotatoes
layTable
cookFish
¬ havePotatoes ∧ haveBakedPotatoes
tableLaid
¬ haveFish ∧ haveBakedFish
tableLaid
haveBakedPotatoes
haveBakedFish
tableLaid ∧ haveBakedPotatoes ∧ haveBakedFish
Finish