1988

Artificial Intelligence through Prolog by Neil C. Rowe

Rowe, Neil C.

Prentice-Hall

http://hdl.handle.net/10945/36984

Abstraction in search

When people solve without computers the search problems discussed in the last two chapters, they only use the techniques so far described as a last resort. Instead, they try to reason about abstractions, what's known as *hierarchical reasoning*.

Abstraction is simplifying your model of the world so you can reason more easily about it. Intermediate predicates for rule-based systems (see Section 7.5) are one form of abstraction, since they summarize a category of data and provide a handle on that summarization. Abstraction is essential in organizing and managing large numbers of facts, as we will see in the next chapter. Abstraction is also a way to simplify large search problems. Search abstraction works best when a problem is decomposable, and exploits the preconditions and postconditions of operators.

Means-ends analysis

The classic technique for solving search problems by abstraction is means-ends analysis. It applies to those decomposable search problems (see Section 9.12) for which clear "major operators" or "recommended operators" on the solution path can be identified in advance. To specify the these operators for specific situations, means-ends analysis needs something like a table. Usually these tables refer to the difference between the current state and the goal state, and are thus called *difference tables*.

Difference tables describe operators in terms of their preconditions and postconditions (see Section 9.4). So they're akin to the **successor** definitions of Chapter 10, but with one big difference: difference tables only *recommend* an operator appropriate to a state and a goal, with no concern if the operator can actually be applied to the state. Difference tables provide a way of decomposing a problem into three simpler subproblems: a subproblem of getting from the current state to a state in which we can apply the recommended operator, a subproblem of applying the operator, and a subproblem of going from there to the goal state. Formally, a search from state S to some goal state G is decomposed into:

1. satisfying the preconditions (prerequisites) for some recommended operator O when starting from S, by going to state S2;
2. applying operator O to S2 to get to state S3 (by postconditions);
3. going from S3 to the goal state G.

The first and third steps are search problems themselves, possibly requiring additional decompositions of their own by the difference table. So means-ends analysis is recursive search. Means-ends analysis is also hierarchical reasoning because we start with big wholes and gradually reason down to little details. The ends truly justify the means.

Complete and correct specification in the difference table of preconditions and postconditions of operators is essential for means-ends analysis. In particular, you must carefully distinguish preconditions from the looser conditions recommending operators. Both could be mixed together in the **successor** rules of Chapter 10, but that won't work here.

A simple example

Means-ends analysis is useful for many human "planning" activities. Here's a simple example of planning for an office worker. Suppose we have a difference table of three rules (see Figure 11-1), written informally as:

> If in your current state you are hungry, and in your goal state you are not hungry, then either the "visit_cafeteria" or "visit_vending_machine" operator is recommended.
>
> If in your current state you do not have change, and if in your goal state you have change, then the "visit_your_office" operator or the "visit_secretary" operator is recommended.
>
> If in your current state you do not know where something is, and in your goal state you do know, then either the "visit_your_office", "visit_secretary", or "visit_colleague" operator is recommended.

We need preconditions and postconditions for the preceding operators:

> Preconditions of visit_cafeteria: you know where the cafeteria is.
> Postconditions of visit_cafeteria: you are no longer hungry.
> Preconditions of visit_vending_machine: you know where the vending machine is and you have change.
> Postconditions of visit_vending_machine: you are no longer hungry.
> Preconditions of visit_your_office: none (we assume everyone knows where his or her office is).
> Postconditions of visit_your_office: you have change and you know where everything is (since you can look it up in, say, the phone directory).
> Preconditions of visit_secretary: you know where the secretary is and the secretary is available.
> Postconditions of visit_secretary: you have change, and you know where everything is.
> Preconditions of visit_colleague: none.
> Postconditions of visit_colleague: you know where everything is.

Since we will implement difference tables in Prolog, we'll follow the usual top-to-bottom and left-to-right conventions about order of action. So we will try difference-table rules in the order listed, try operator alternatives in the order listed, and try to satisfy preconditions and postconditions in the order listed. So the order of rows and columns in Figure 11-1 is significant.

Let's illustrate means-ends analysis for the situation in which you are hungry, you have no change, a secretary is available, and you are new and don't know where anything is besides your own office; and your goal is any state in which you are not hungry. (Other things can be true of your final state besides being not hungry, but hunger relief is what you insist on.)

> 1. The first rule succeeds, so you want to visit the cafeteria.
>
> 2. This has the precondition that you know where the cafeteria is, and since you don't, you must recursively call on means-ends analysis to find a way to change that discrepancy (difference).
>
> 3. Only the third rule can handle this kind of a difference between a state and a goal, so we try to apply the first operator recommended, visit_your_office. (That is, you will look through the telephone directory to locate the cafeteria.)
>
> 4. There are no preconditions for visit_your_office (everybody knows where their office is), so it

can be applied without further recursion. And the state after visit_your_office satisfies the preconditions of visit_cafeteria as you hoped--no postconditions will interfere.

5. The postcondition of visit_cafeteria is that you are no longer hungry, and the goal of the problem was to make this become true. So we're done. So the operator sequence found to solve the problem is visit_your_office and then visit_cafeteria.

To show the importance of difference-table order, suppose we switch the order of visit_cafeteria and visit_vending_machine in the first rule, switch the order of visit_your_office and visit_secretary in the second rule, put visit_colleague first in the third rule, and switch the second and third rules, so we have (see Figure 11-2):

If in your current state you are hungry and in your goal state you are not hungry, then either the "visit_vending_machine" or "visit_cafeteria" operator is recommended.

If in your current state you do not know where something is, and in your goal state you do know, then either the "visit_colleague", "visit_your_office", or "visit_secretary" operator is recommended.

If in your current state you do not have change and if in your goal state you have change, then the "visit_secretary" operator or the "visit_your_office" operator is recommended.

Let's redo the problem in which you are hungry, you need change, a secretary is available, and you don't know anything, and you want to get to a state in which you are no longer hungry. Means-ends analysis finds a different, but equally valid, solution:

1. The first rule succeeds, so it is recommended that you first try to visit a vending machine. This has two preconditions: that you know where one is and you have change.

2. We recursively call on means-ends analysis to satisfy these preconditions. The starting state is the original starting state, and the new goal is any state in which you know where a vending machine is and you have change.

3. The first rule won't help this difference, but the second rule will, the one that recommends visiting a colleague (to ask where a vending machine is). This has no preconditions, so you can directly apply it.

4. After applying the postconditions of visit_colleague, we must still get to a state in which we have change; remember, visit_vending_machine had two preconditions. (All my colleagues are poor and never have any change--or so they tell me.) So we need a recursive "postcondition search" to accomplish this, from the state in which we are hungry, we need change, a secretary is available, and we know everything. For this difference the third rule is recommended, and the recommended operator is visit_secretary.

5. This has two preconditions: that you know where the secretary is and that secretary is available. Both conditions hold in the current state (the first condition since the beginning, and the second condition was just accomplished by visit_colleague). So we can directly apply visit_secretary.

6. Now taking into account the postconditions of visit_secretary, we're in a state in which we are hungry, we have change, a secretary is available, and we know where everything is. This satisfies all the conditions of the visit_vending_machine operator, so we apply it. Examining its postconditions, we see we've reached a goal state.

7. So the overall operator sequence is to visit your colleague to find out where a vending machine is and where the secretary is, visit the secretary to get change, and then visit a vending machine.

This example didn't require backtracking, but you may need it when your first choice for a rule or operator doesn't pan out, just as with depth-first search. A choice doesn't pan out when no way can be found to satisfy the preconditions or to handle the postconditions. However, the more "intelligent" nature of means-ends analysis compared to the other search methods means that it backtracks a lot less than they do on nicely decomposable problems.

This example of means-ends analysis is highly simplified. If we wanted to realistically model the way people solve daily problems like the preceding, we would need a lot more operators, preconditions, and postconditions. Nevertheless, many psychologists argue that something like means-ends analysis underlies much human activity.

Partial state description

In the preceding example, we described goal states by giving a few facts that must be true in them. For instance, we said the overall goal was a state in which you are no longer hungry. This *partial state description* is very common with means-ends analysis, because it's hard to define a single **goalreached** predicate when each precondition decomposition has a different goal.

Since means-ends analysis is driven by differences between states, we must be careful to define this difference for partial state descriptions. Basically, we can go through the list of facts describing the goal G and "cross out" facts also true for current state S. Here's the definition:

```
difference([],S,[]).
difference([P|G],S,G2) :- singlemember(P,S), !, difference(G,S,G2).
difference([P|G],S,[P|G2]) :- difference(G,S,G2).
```

(Predicate **singlemember** was defined in Section 10.7 and checks only once whether an item is a member of a list; the **!** (cut symbol) was explained in Section 10.7 too.) Then we write difference-table rules to refer to these differences.

So for instance:

```
?-difference
  ([on(b,c),on(e,f),on(a,e)],[on(a,b),on(b,c),on(d,e),on(e,f)],D).
D=[on(a,e)]
```

Implementation of means-ends analysis

With these details out of the way, implementation of means-ends analysis is straightforward. We represent the difference table with rules and/or facts defining

```
recommended(<difference>,<operator>)
```

where **<difference>** is some result (binding of the third argument) of the **difference** predicate defined in the last section, and **<operator>** is the name of a recommended operator. Generally, each operator will have at least one **recommended** definition.

Preconditions are best expressed as two-argument facts. The first argument can be the operator, and the second argument the list of associated preconditions, predicate expressions that must be present in a state description for the operator to be used in that state:

```
precondition(<operator>,[<predexpr1>,<predexpr2>,...]).
```

But postconditions are tricky. An operator can not only make certain conditions true, but can make certain other true conditions become false--otherwise, we couldn't undo anything. So we'll distinguish two kinds of postconditions: predicate expressions added by an operator, and predicate expressions deleted.

```
addpostcondition(<operator>,[<predexpr1>,<predexpr2>...]).
deletepostcondition(<operator>,[<predexpr1>,<predexpr2>...]).
```

By the way, operators can have arguments just as in Chapter 9, and these arguments can be referenced in the precondition and postcondition lists.

That covers the format of the problem-dependent part of means-ends analysis; now for the problem-independent part. We'll use *double recursion*, which sounds like a horrible disease but just means a self-referencing program that self-references twice. Our top-level rule will be a recursive **means_ends** predicate of four arguments:

> **State**, a complete list of facts true in the starting state (an input);
> **Goal**, a list of those facts that must be true in the goal state, a partial description of the goal state (an input);
> **Oplist**, the of operators that will get you from the start to the goal (an output);
> **Goalstate**, the complete list of facts true in the goal state (an output).

Here's the program:

```
/* Problem-independent means-ends analysis */
means_ends(State,Goal,[],State) :- difference(Goal,State,[]), !.
means_ends(State,Goal,Oplist,Goalstate) :-
  difference(Goal,State,D),
  recommended(Dsub,Operator), subset(Dsub,D),
  precondition(Operator,Prelist),
  means_ends(State,Prelist,Preoplist,Prestate),
  deletepostcondition(Operator,Deletepostlist),
  deleteitems(Deletepostlist,Prestate,Prestate2),
  addpostcondition(Operator,Addpostlist),
  union(Addpostlist,Prestate2,Postlist),
  means_ends(Postlist,Goal,Postoplist,Goalstate),
  append(Preoplist,[Operator|Postoplist],Oplist).
```

This recursive program has a single basis step (the first line), which says we don't need any operators to solve a problem in which the starting state includes all the goal facts. The rest of the lines are a single induction step, which you'll note has two recursive calls: the first for the preconditions, the second for the

postconditions. The lines say to first compute the list of facts different between **State** and **Goal**, and find a recommended operator for some subset of those facts. Look up the preconditions of the operator and recursively call **means_ends** to figure how to satisfy them. Then look up the deletion postconditions and delete them from the final state resulting from the precondition recursion; look up the addition postconditions, and add them to that state. Now we have the state after application of the original recommended operator. So recursively call **means_ends** to figure how to get from here to the original goal. The final operator list for the whole problem is the appending together of the precondition-recursion operator list, the recommended operator, and the postcondition-recursion operator list.

The **means_ends** program requires five utility functions: **singlemember**, **subset**, **append**, **union**, and **deleteitems**. The **singlemember** (defined in Section 10.7) checks whether an item is a member of a list, while preventing backtracking. The **subset** (defined in Section 5.7) says whether every item in its list first argument is also a member of its list second argument. The **append** (defined in Section 5.6) glues two lists together to create a new list. The **union** is like the **append** except it removes duplicates from the result. The **deleteitems** removes all members of one list from another list, and uses the **delete** predicate of Section 5.6. To improve efficiency, cut symbols **!** appear in some of these definitions.

```
/* Utility functions for means-ends analysis */
singlemember(X,[X|L]) :- !.
singlemember(X,[Y|L]) :- singlemember(X,L).


subset([],L).
subset([X|L],L2) :- singlemember(X,L2), subset(L,L2).


append([],L,L).
append([X|L],L2,[X|L3]) :- append(L,L2,L3).


union([],L,L).
union([X|L1],L2,L3) :- singlemember(X,L2), !, union(L1,L2,L3).
union([X|L1],L2,[X|L3]) :- union(L1,L2,L3).


deleteitems([],L,L).
deleteitems([X|L],L2,L3) :- delete(X,L2,L4), deleteitems(L,L4,L3).


delete(X,[],[]).
delete(X,[X|L],M) :- !, delete(X,L,M).
delete(X,[Y|L],[Y|M]) :- delete(X,L,M).
```

To make the means-ends program more user-friendly, it's helpful to put in some error checking. This goes after the two original definitions of **means_ends**.

```
means_ends(State,Goal,Oplist,Goalstate) :-
  not(difference(Goal,State,D)),
  write('Bug found: no difference defined between '),
  write(Goal), write(' and '), write(State), nl, !, fail.
means_ends(State,Goal,Oplist,Goalstate) :-
  difference(Goal,State,D), recommended(Dsub,Operator),
  subset(Dsub,D), not(precondition(Operator,Prelist)),
  write('Bug found: no preconditions given for operator '),
```

```
  write(Operator), nl, !, fail.
means_ends(State,Goal,Oplist,Goalstate) :-
  difference(Goal,State,D),
   recommended(Dsub,Operator), subset(Dsub,D),
  not(deletepostcondition(Operator,Deletepostlist)),
  write('Bug found: no deletepostconditions given for operator '),
  write(Operator), nl, !, fail.
means_ends(State,Goal,Oplist,Goalstate) :-
  difference(Goal,State,D),
  recommended(Dsub,Operator), subset(Dsub,D),
  not(addpostcondition(Operator,Addpostlist)),
  write('Bug found: no addpostconditions given for operator '),
  write(Operator), nl, !, fail.
```

A harder example: flashlight repair

Car repair is often a decomposable search problem and thus appropriate for means-ends analysis, but it's too complicated to use as an example here. So let's consider repair of a flashlight, the kind shown in Figure 11-3. This flashlight has two batteries inside a case, attached to a light bulb. To reach the batteries you must disassemble the case, and to reach the light bulb you must disassemble the top as well as disassembling the case. We'll use the following facts to describe states in flashlight repair (which mean what they suggest):

```
defective(batteries).
ok(batteries).
defective(light).
ok(light).
open(case).
closed(case).
unbroken(case).
broken(case).
open(top).
closed(top).
inside(batteries).
outside(batteries).
```

We'll assume the following operators: replace_batteries, replace_light, disassemble_case, assemble_case, disassemble_top, assemble_top, turn_over_case, and smash_case.

Now let's apply our means-ends program to the flashlight-repair problem. First we need a difference table (summarized in Figure 11-4), stating recommended operators with facts of the predicate name **recommended**:

```
recommended([ok(batteries)],replace_batteries).
recommended([ok(light)],replace_light).
recommended([open(case)],disassemble_case).
recommended([open(case)],smash_case).
recommended([open(top)],disassemble_top).
recommended([open(top)],smash_case).
recommended([closed(case)],assemble_case).
recommended([closed(top)],assemble_top).
recommended([outside(batteries)],turn_over_case).
recommended([outside(batteries)],smash_case).
```

As you may recall, the first argument to **recommended** is part of the difference of the goal from the current

state, and the second argument is the recommended operator. So the first argument is what we want to become true, what isn't true in the starting state. So read the first fact as "If you want to get the batteries to be OK because they aren't now, try the replace_batteries operator." All these recommendations refer to single-fact desires, but that's only because it's hard to do two things at once with one action on a flashlight; other kinds of repair are different.

The order of the **recommended** rules is important, because they'll be considered by the Prolog interpreter in that order, and the first one applying to the given state and goal will be the first one used. So we've put in front those for the most important operators, the "replace_batteries" and "replace_light" operators. Several situations suggest the "smash_case" operator besides another operator, but we want to make "smash_case" a last resort, and hence put the line mentioning it after the line for the alternative. Now for the preconditions:

```
precondition(replace_batteries,
  [open(case),outside(batteries),unbroken(case)]).
precondition(replace_light,[open(top)]).
precondition(disassemble_case,[closed(case)]).
precondition(assemble_case,
  [open(case),closed(top),unbroken(case)]).
precondition(disassemble_top,[open(case),closed(top)]).
precondition(assemble_top,[open(top)]).
precondition(turn_over_case,[open(case)]).
precondition(smash_case,[]).
```

Notice we don't need to list everything that must be true for an operator to be applied, only the major conditions that imply everything else. For instance, the precondition for "replace_light" that the top be open requires that the case be open first, but we will find this in looking up the preconditions to the disassemble_top operator anyway. So means-ends analysis lets us put preconditions in the places that make our programs clearest.

Now for the postconditions. Often the facts deleted by postconditions (deletepostconditions) are just the preconditions, and often the facts added by postconditions (addpostconditions) are just the opposites of all the deletepostconditions. So postconditions are often easy to write. But this isn't true when operators have "side effects", like the "smash_case" operator that does several things simultaneously--and that's one of the reasons it's not a very good operator.

```
deletepostcondition(replace_batteries,
  [outside(batteries),defective(batteries)]).
deletepostcondition(replace_light,[defective(light)]).
deletepostcondition(disassemble_case,[closed(case)]).
deletepostcondition(assemble_case,[open(case)]).
deletepostcondition(disassemble_top,[closed(top)]).
deletepostcondition(assemble_top,[open(top)]).
deletepostcondition(turn_over_case,[inside(batteries)]).
deletepostcondition(smash_case,
  [unbroken(case),closed(case),closed(top),inside(batteries)]).


addpostcondition(replace_batteries,
  [inside(batteries),ok(batteries)]).
addpostcondition(replace_light,[ok(light)]).
addpostcondition(disassemble_case,[open(case)]).
addpostcondition(assemble_case,[closed(case)]).
addpostcondition(disassemble_top,[open(top)]).
```

```
addpostcondition(assemble_top,[closed(top)]).
addpostcondition(turn_over_case,[outside(batteries)]).
addpostcondition(smash_case,
  [broken(case),open(case),open(top),outside(batteries)]).
```

Note that even if some **deletepostcondition** facts are not true for a state, no harm is done--those facts will just be ignored--because the **delete** predicate and hence the **deleteitems** predicate always succeed, even when they don't delete anything.

Running the flashlight program

Let's run our flashlight and means-ends analysis code for the situation in which the batteries are defective and we want to get the flashlight working again. The correct solution is shown in Figure 11-5. Let's assume the flashlight is assembled and unbroken when we start out, so the starting state is

```
[closed(case),closed(top),inside(batteries),defective(batteries),
  ok(light),unbroken(case)].
```

The goal is any state in which these facts are true (i.e., this partial goal description holds):

```
[ok(batteries),closed(case),closed(top)].
```

(A flashlight isn't much good unless it's put back together again, hence the last two conditions.) To do means-ends analysis, we supply the preceding two lists as inputs (the first two arguments) to **means_ends**, leaving the last two arguments unbound:

```
?- means_ends([closed(case),closed(top),inside(batteries),
  defective(batteries),ok(light),unbroken(case)],
  [ok(batteries),closed(case),closed(top)], Operators, Final_state).
```

What will happen is summarized in Figures 11-6 (an overview) and 11-7 (the details). The four arguments to **means_ends** (**State, Goal, Oplist, and Goalstate**) are listed for each call, plus the local variable **Operator**. The calls are drawn like the predicate hierarchies of Figures 10-5, 10-6, and 10-7. **Oplist** and **Goalstate** are the outputs bound as a result of reasoning, so the values shown for them are their final bindings.

In more detail, here's what Figure 11-7 recounts. Fasten your seat belts.

1. The **difference** function is applied to the original goal and the starting state. The last two facts in the goal cancel out, leaving only **[ok(batteries)]**. For this difference, "replace_batteries" is the recommended operator. The problem can then be recursively divided into three parts: satisfying the preconditions of replace_batteries, applying it, and going from there to the original goal.

2. The three preconditions of replace_batteries are that case is open, the batteries are outside the case, and the case is unbroken. So we recursively call the predicate **means_ends** with first argument the original starting state, and second argument (partial goal description) the precondition list

   ```
   [open(case),outside(batteries),unbroken(case)].
   ```

3. The difference between this new goal and the original starting state is

```
[open(case),outside(batteries)].
```

For this, either the "disassemble_case", "smash_case", or "turn_over_case" operators are recommended. We always try the first recommendation first. That's disassemble_case, and its only precondition is that the case be closed.

4. So we do a third level of recursive call with the same starting state and the goal

```
[closed(case)].
```

But this is already true in the starting state. So the difference is **[]**, and no work must be done to solve this recursion. We succeed with the first **means_ends** rule (the basis step), setting the third argument (the operator list) to **[]** and the fourth argument (the final state) to the starting state. Such trivial search problems are indicated in Figure 11-7 by the words "nothing needed".

5. Now we return to the second level of recursion where we are trying to apply the recommended "disassemble_case" operator. We just discovered that no work is needed to satisfy preconditions, so we just figure out the consequences of disassemble_case. The facts to be deleted (the deletepostconditions) are **[closed(case)]**, and the facts to be added (the addpostconditions) are **[open(case)]**. So the state after applying disassemble_case (the **Postlist** variable) is

```
[open(case),closed(top),inside(batteries),defective(batteries),
  ok(light),unbroken(case)]
```

6. Now we recurse (to the third level) to find how, from this new state, we can satisfy the second-level partial goal description

```
[open(case),outside(batteries),unbroken(case)].
```

The difference is

```
[outside(batteries)].
```

so now only one fact is now different. For this difference, two operators are recommended: turn_over_case and smash_case, in that order. We try the first one first.

7. Turn_over_case has precondition

```
[open(case)].
```

which is already true. So the precondition recursion for turn_over_case succeeds trivially. Using the deletepostconditions and addpostconditions, the new state after turn_over_case is

```
[outside(batteries),open(case),closed(top),
  defective(batteries),ok(light),unbroken(case)]
```

We now compare this to the second-level goal of

```
[open(case),outside(batteries),unbroken(case)].
```

and the second-level goal is satisfied. So a postcondition recursion for turn_over_case succeeds trivially. The third level of **means_ends** recursion binds its third argument (answer operator list)

to **[turn_over_case]**, and its fourth argument (final state) to the just-mentioned state.

8. We return to the second level, and can complete its job with third argument (answer operator list)

```
[disassemble_case,turn_over_case].
```

so we can bind the fourth argument (final state) to the same state list as previously.

9. We finally return to the first level, where were have satisfied the preconditions of replace_batteries. We now figure out the result of replace_batteries. The deletepostcondition is

```
[outside(batteries),defective(batteries)].
```

and the addpostcondition is

```
[inside(batteries),ok(batteries)].
```

so the new state is

```
[inside(batteries),ok(batteries),open(case),
  closed(top),ok(light),unbroken(case)]
```

10. We now must determine how to go from this new state to the original (first-level) goal of

```
[ok(batteries),closed(case),closed(top)].
```

We recursively call **means_ends** to do this. The only difference is **[closed(case)]**, for which the recommended operator is assemble_case. The preconditions of assemble_case are satisfied by the last-mentioned state, so we just work out the postconditions of the operator. This gives the new state

```
[closed(case),inside(batteries),ok(batteries),
  closed(top),ok(light),unbroken(case)]
```

This state matches the goal, so no postcondition recursion is necessary at the second level. So the postcondition recursion for replace_batteries (the top level) finishes with third argument (operator list) being **[assemble_case]** and fourth argument (new state) the preceding state.

11. To summarize, we have for the replace_batteries operator a precondition-satisfying operator list **[disassemble_case,turn_over_case]** and a postcondition-handling operator list of **[assemble_case]**. So the complete operator list for this problem is

```
[disassemble_case,turn_over_case,replace_batteries,assemble_case].
```

and the final state is what we just found,

```
[closed(case),inside(batteries),ok(batteries),
  closed(top),ok(light),unbroken(case)]
```

You can see that means-ends analysis is a little tricky, but it all makes sense when you follow through the steps carefully. Curiously, the operator sequence found seems "obvious" to people. This is another one of those situations mentioned at the beginning of Chapter 2: things that seem obvious at first, but aren't so

obvious when you study them. Getting computers to understand language and interpret pictures are also surprisingly difficult problems.

It's interesting to see what happens if for step 3 we try smash_case instead of disassemble_case to get the case open. The smash_case operator doesn't have any prerequisites, but it has the serious postcondition (actually, addpostcondition) that the case is broken. None of the operators can replace a **broken(case)** fact by the **unbroken(case)** we need for the goal state, so further progress is impossible. So if we ever selected smash_case, we would fail and need to backtrack.

Means-ends versus other search methods

The flashlight example illustrates the advantages of means-ends analysis over the classical search methods of Chapters 9 and 10. Consider what those earlier methods would do on the same problem. They would have considered every action possible in the starting state, then every next action, and so on until they reached the goal. Such a search is basically "local" in not having an overall plan of attack, just moving like an ant one step at a time. Heuristics, evaluation functions, and cost functions are just tools to make better local decisions about which operators are most likely to lead to the goal.

Means-ends analysis, on the other hand, reasons top-down from abstractions, gradually filling in details of a solution. It's "global" since it always has an overall idea of what it's trying to do. It uses recursion to decompose a problem in an intelligent way, a way more like how people solve problems. It does require that you identify explicit and complete preconditions and postconditions for each operator; when this is impossible, means-ends won't work.

Another advantage of means-ends analysis is its explainability. Search results in a list of operators to apply in the real world. When this list is long, it's hard to understand and follow, leading to mistakes. But if you can group ("chunk") operators as means-ends analysis does, like saying that a sequence of four operators accomplishes some single overall goal, then a long string is more comprehensible. We'll talk more about explanations in Chapter l5.

Modeling real-word uncertainty (*)

Means-ends analysis plans a solution to (operator sequence for) a search problem; actually using its solution in the real world is another thing altogether. Difficulties occur when assumptions on which the plan is based are violated. For instance, you may assume the flashlight batteries are defective, but discover when you've replaced them that it really was the light that was defective. Or while you were assembling the case, the batteries may fall out. With cases (pun intended) like this we don't need to start over, but we can replan from the intermediate state; that is, do a new means-ends analysis with this intermediate state as the starting state, and goal the same. Do this anytime you find inconsistency between the real-world state and the state anticipated by planning at some point. So even though means-ends analysis is top-down reasoning from abstract goals, you can handle mistakes and imperfections in the world.

Procedural nets (*)

All the search methods in this and the last two chapters find a sequence of states or operators to solve a problem, on the assumption you can do only one action at a time. But this may be false: you may have a multiarm robot, or you may have several people to help you, or actions not requiring supervision can be

started simultaneously. As we discussed in Section 10.12, concurrency can speed up search.

Concurrency in an operator sequence can be expressed with the *PERT chart*, a kind of lattice often used in operations research. Figure 11-8 gives an example, one way of replacing both the batteries and light in a flashlight. Each box represents an operator; an operator to the right of another must wait for the first to finish, and vertically parallel operator sequences can be done concurrently. Much work in operations research has studied good ways to design and use such PERT charts efficiently.

PERT charts are often used in artificial intelligence under the name of *procedural nets*. The algorithms used are similar to means-ends analysis but result in a procedural net, by noting multiple differences between a state and the goal that can be eliminated simultaneously by parallel activities. But a major headache with concurrency is possible conflict between simultaneous activities. The postconditions of one concurrent operator might undo the preconditions for another, for instance, requiring that the operators be done in sequence instead of in parallel. So several different conflict-handling approaches have been proposed for procedural nets. They're too complicated to explain in detail here, but involve intersection checks on preconditions and postconditions of operators appearing in parallel; when conflicts are found, the operators are rearranged into a sequence. This may involve removing part or all of the concurrency, or finding a new concurrency. A rule-based system can summarize such fixes.

.SH Keywords:

means-ends analysis
difference table
recursive decomposition
double recursion
hierarchical reasoning
abstraction
planning
partial state description
procedural net

Exercises

11-1. (A) Newspapers often run *household hints* like:

> To remove egg from plastic, use grapefruit juice.
> Wrap sandwiches in lettuce to keep them fresher.
> To make a cauldron bubble faster, use some eye of newt.

What part of means-ends analysis do these correspond to?

11-2. (E) As described, means-ends analysis decomposes a problem into three parts. Explain why a useful variant might be to decompose a problem into five parts. Generally speaking, how could such a variant be implemented in Prolog?

11-3. You are a character in a British murder mystery. You have endured enough of Sir Reginald Finch-Stratton. Not only is he an unbearable egotist, but his bad advice has caused you to lose your life savings on unsound business ventures. Now you hear of his sordid liaison with your daughter, an affair that has left her a broken woman. Clearly the man deserves to die.

(a) Set this up for the means-ends analysis program. Assume his room is next to yours and he keeps the door locked. Assume the operators are to stab him with a knife, to pick up the knife, to hide the knife, to go from a room to the corridor or vice versa, to knock on his door, and to wait until the corridor is empty. Needless to say, you don't want to be caught in this foul deed, so you don't want anyone to see you or to find the murder weapon. Assume in the starting state that you are in your room, there are people in the corridor, you are not holding the murder weapon (knife), Sir Reginald is alive, Sir Reginald is in his room, and the door is locked.

(b) Trace through informally the operation of means-ends analysis on this problem. What is the final operator sequence found?

(c) Some of the things assumed in this problem are hard to know in advance, like whether there will be people in the corridor or not. How in general can means-ends analysis be modified to handle such randomness in facts?

11-4. (R,A,P) In many flashlights the top consists of three parts instead of the two assumed in this chapter: a holder, a transparent plate, and a light bulb that screws into the holder underneath the plate. Modify the flashlight repair program to handle flashlights with these three parts, in which only the light is ever defective. You'll need to define new operators **remove_plate** and **replace_plate**; the plate must be removed before the light can be removed, and the plate must be replaced before the top can be assembled. You'll also need some new predicates to include in states.

Try it out on the same repair situation given in the chapter.

11-5. (P) Consider using the flashlight repair program for the situation in which both the light and the batteries are defective.

(a) What does the program in this chapter give as an answer operator list? (Use a computer if you have one.)

(b) The order of the **recommended** rules can make a lot of difference. Suppose you move the second **recommended** rule (the one about replacing the light) to the end of the others. What answer do you get now? Why?

(c) Suppose you instead move the first **recommended** rule (the one about replacing the batteries) to the end of the end of the others. What answer do you get now? Why?

11-6. (A,E) Auto repair manuals often suggest laying out the parts removed in order in a line on a flat surface. Assuming that people do auto repair by something close to means-ends analysis, explain in artificial-intelligence and computer-science terms why this way of laying out is a good idea.

11-7. (A) So far, preconditions and postconditions have been lists of predicate expressions "and"ed together. This seems limiting. Suppose some operator has two alternative sets of preconditions, either of which can be satisfied to make the operator usable. Give an easy way to represent such situations when using the mean-ends analysis program in this chapter.

11-8. (E) The means-ends analysis program of this chapter works in a depth-first way: it figures out how to satisfy the preconditions of a recommended operator, possibly requiring more recursions, and then it figures out how to go from the result of the operator to the goal, possibly requiring more recursions. So it's depth-first in style, with a branching factor of 0, 1, or 2 at each node (each invocation of the **means_ends**

predicate). But this is not strictly necessary.

(a) Explain how a breadth-first means-ends analysis might work.

(b) Why do you think we used the depth-first approach in this chapter?

11-9. (E) Discuss the application of means-ends analysis to improve the user-friendliness of using networks of computers. Give an example.

11-10. (E) Discuss the similarities and differences between means-ends analysis and the top-down parsing of strings of words as discussed at the end of Chapter 6.

11-11. (H) Machines now exist to synthesize pieces of DNA (deoxyribonucleic acid) under microprocessor control. Chemical reactions take place in a single test tube. The following actions are possible:

1. Add plastic beads to the test tube.
2. Add one of four similar "bases" to the tube (symbolized by A+X, C+X, G+X, and T+X).
3. Add a chemical that strips the X off bases (for instance, A+X becomes A)
4. Add a chemical that binds the left ends of bases to beads.
5. Neutralize the chemical that binds the left ends of bases to beads.
6. Add a chemical that detaches the left ends of bases (or strings of bases) from beads.

The following conditions apply:

1. The right end of a base will attach to the left end of another base, but only if there is no X on the right end of the first base.

2. If an X is removed from a base and the base is not used immediately, it will decompose into a different (useless) chemical.

For this problem, use means-ends analysis to find a plan to synthesize the chemical A+C+G (and only that chemical, no other combinations). That is, accomplish both the operations "attach C to right end of A" and "attach G to right end of C".

Go to book index