Calhoun
Institutional Archive of the Naval Postgraduate School

Calhoun: The NPS Institutional Archive

DSpace Repository

Faculty and Researchers | Faculty and Researchers' Publications

1988

Artificial Intelligence through Prolog by Neil C. Rowe

Rowe, Neil C.

Prentice-Hall

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

Downloaded from NPS Archive: Calhoun

DUDLEY KNOX LIBRARY

Calhoun is a project of the Dudley Knox Library at NPS, furthering the precepts and goals of open government and government transparency. All information contained herein has been approved for release by the NPS Public Affairs Officer.

Dudley Knox Library / Naval Postgraduate School
411 Dyer Road / 1 University Circle
Monterey, California USA 93943

http://www.nps.edu/library

Implementing search

We have postponed consideration of search implementation to this chapter to better focus on the quite different issues involved. We will present programs for several kinds of search, working from a search problem described in an abstract way. We'll also use search as a springboard to discuss some advanced features of Prolog, including set-collection and "cut" predicates, and more of the wild and wacky world of backtracking.

Defining a simple search problem

First, we need a way to describe a search problem precisely. We ask the programmer to define the following predicates, the last two optional depending on the desired search strategy:

> --**successor(<old-state>,<desired new-state>)**: rules and facts defining this predicate give all possible immediate branches or state transitions. It's a function predicate with its first argument an input, a state, and second argument an output, a successor state. Both arguments are state descriptions. We emphasize that **<new-state>** must be an *immediate* successor.
>
> --**goalreached(<state>)**: rules and facts defining this predicate give the stopping conditions for the search. The argument is an input, a state. Multiple goal states are possible.
>
> --**eval(<state>,<evaluation>)**: rules and facts defining this predicate give the evaluation function. This is a function predicate taking an input state as first argument, and binding its second argument, an output, to an estimate of how close that state is to the nearest goal state.
>
> --**cost(<state-list>,<cost>)**: rules and facts defining this predicate give the cost function. This is a function predicate taking an input list of states as first argument, and binding its second argument, an output, to some nonnegative number representing the sum of the costs along the path through those states.

Of these four, the first is generally the hardest: it must define all the operators, incorporating their necessary conditions and describing precisely how they modify a state-description list. (The **successor** definitions may also include heuristics about operator use, though Section 10.11 will show a more general way to handle heuristics.) Successor definitions turn a search problem into a rule-based system, so we can steal ideas from Chapter 7.

Then to actually start searching, we ask that you query something like

```
?- search(<starting-state>,<answer-state-list>).
```

where the first argument is an input, the starting state (written in the format handled by the successor rules), and the second argument is an output, the discovered sequence of states, usually in reverse order (that is, from the goal state back to the start state).

As an example of this Prolog-style definition of a search problem, let's return to the example of Figure 9-1, redrawn as Figure 10-1. This is a city route planning problem in which our starting location is **a** and our goal

location is **h**. States are the labeled intersections, so to define **successor** we must write a fact for every pair of intersections that connect directly:

```
successor(a,b).
successor(a,d).
successor(b,c).
successor(b,a).
successor(b,d).
successor(c,b).
successor(d,a).
successor(d,e).
successor(d,g).
successor(e,d).
successor(e,f).
successor(e,g).
successor(f,e).
successor(g,d).
successor(g,e).
successor(h,g).
```

For the **goalreached** condition we need:

```
goalreached(h).
```

And to start the program out, we will query a particular search predicate, like

```
?- depthsearch(a,Path).
```

to do depth-first search from start state **a**. (The search predicates we'll define in this chapter are **depthsearch**, **breadthsearch** (breadth-first), **bestsearch** (best-first), and **astarsearch** (A* search).) When search is done, the variable **Path** is bound to the list of intersections (states) we must travel through. We can use evaluation and cost functions to improve search if we like. A good evaluation function for this problem would be the straight-line distance to the goal (measuring on the map with a ruler), giving approximate distances for Figure 10-1 of:

```
eval(a,8).
eval(b,7).
eval(c,8).
eval(d,5).
eval(e,3).
eval(f,1).
eval(g,2).
eval(h,0).
```

The **cost** definition can be the actual distance along a road, which we can approximate from Figure 10-1 as

```
cost([X],0).
cost([X,Y|L],E) :- piece_cost(X,Y,E2), cost([Y|L],E3), E is E2 + E3.

piece_cost(a,b,3).
piece_cost(a,d,5).
piece_cost(b,c,1).
piece_cost(b,d,2).
piece_cost(d,e,2).
piece_cost(d,g,3).
piece_cost(e,f,2).
```

```
piece_cost(e,g,1).
piece_cost(g,h,2).
piece_cost(X,Y,C) :- piece_cost(Y,X,C).
```

A **cost** definition will often need recursion to handle paths of unknown length.

Defining a search problem with fact-list states

Now let's try representing a different kind of search problem, one more typical. The previous example was easy because states could be represented by letters. But often you can't name states in advance, as when you don't know what states are possible, and you must describe them by lists of facts.

Here's an example. This search concerns a very small part of car repair: putting nuts and washers onto bolts in the right way. Suppose we have two bolts. In the starting state (the top picture in Figure 10-2), nut **a**, washer **b**, and nut **c** are on the left bolt in that order, and nut **d**, washer **e**, and nut **f** are on the right bolt in that order. The goal is to get nut **c** on top of washer **e** (as for example in the bottom picture in Figure 10-2). To do this, we must work on one nut or washer at a time. We can remove one from the top of the nuts and washers on a bolt, place one on the top of the nuts and washers on a bolt, or just place one alone on the flat surface around the bolts. The bolts themselves can't be moved. The four nuts and two washers are the only parts available.

The only facts that change during the search are those about what parts (nuts or washers) rest on what other parts. Let's use **on(<part-1>,<part-2>,<bolt>)** facts for this, where **<part-1>** and **<part-2>** are codes for parts, and **<part-1>** is on the bolt named **<bolt>** directly above **<part-2>**. To keep track of parts *not* on the bolts, we'll represent them with **on** facts too, by **on(<part>,surface,none)**; parts can only rest on other parts when both are on bolts. To keep track of empty bolts, we'll use two additional permanent facts, **bolt(bolt1)** and **bolt(bolt2)**. So any state can be represented as a list of eight predicate expressions, six specifying the location of each of six parts, and two giving the names of the bolts. For instance, the starting state (the top picture in Figure 10-2) can be written

```
[on(a,b,bolt1),on(b,c,bolt1),on(c,surface,bolt1),on(d,e,bolt2),
  on(e,f,bolt2),on(f,surface,bolt2),bolt(bolt1),bolt(bolt2)]
```

In other words, like a Prolog database on its side. We could start a depth-first search by querying

```
?- depthsearch([on(a,b,bolt1),on(b,c,bolt1),on(c,surface,bolt1),
  on(d,e,bolt2),on(e,f,bolt2),on(f,surface,bolt2),
  bolt(bolt1),bolt(bolt2)],Answerpath).
```

The example final state in Figure 10-2 can be written

```
[on(a,surface,none),on(b,surface,none),on(c,e,bolt2),
  on(d,surface,none),on(e,f,bolt2),on(f,surface,bolt2),
  bolt(bolt1),bolt(bolt2)]
```

But let's say we're not so fussy about where every part ends up. Let's say we only want part **c** to be on part **e**. Then we can define the goal state by:

```
goalreached(S) :- member(on(c,e,bolt2),S).
```

That is, we can stop in state **S** whenever the facts true for **S** include **on(c,e,bolt2)**. The **member** predicate was defined in Section 5.5, and is true whenever some item is a member of some list:

```
member(X,[X|L]).
member(X,[Y|L]) :- member(X,L).
```

Successor rules like the preceding that work on fact lists usually just insert and delete facts from the list describing one state to get the list describing another. Typically, they only insert and delete a few. So to specify the effect of an operator, we need only list the fact *changes* made by an operator--everything else can be assumed constant. In problems with complicated state descriptions, it may be hard to figure which facts stay the same, because of "side effects". This is called the *frame problem* in artificial intelligence (not to be confused with the knowledge-partitioning "frames" we'll discuss in Chapter 12).

By referring to variables, just three successor rules are needed for the bolts problem, as shown in the following code. Their form is like the **append** and **delete** predicate definitions (Section 5.6). The first handles removing parts from bolts, and the second handles placing parts on bolts. (The intermediate predicate **cleartop** simplifies the rules.) The first successor rule says that if in some state the part **X** (1) is on a bolt and (2) doesn't have another part on it, then a successor state is one in which **X** is removed from the bolt and placed on the surface. Furthermore, the rule says that we can get the successor state from the old state by removing the old fact about where part **X** was (with the **delete** predicate), and adding a new fact that **X** is alone on the surface (with the stuff in brackets on the rule left side, the last thing done before the rule succeeds). The second successor rule says that if in some state the part **X** (1) doesn't have another part on it, and (2) another part **Z** doesn't have anything on it either, and (3) **Z** is on a bolt, and (4) **Z** is different from **X**, then a possible successor state has **X** placed on **Z**. And we can get a description of the new state by removing the old location of **X** and adding the new location. The third successor rule says that if a bolt is empty in some state, then we can put on it any part with a clear top.

```
successor(S,[on(X,surface,none)|S2]) :- member(on(X,Y,B),S),
  not(B=none), cleartop(X,S), delete(on(X,Y,B),S,S2).
successor(S,[on(X,Z,B2)|S2]) :- member(on(X,Y,B),S),
  cleartop(X,S), member(on(Z,W,B2),S), not(B2=none),
  cleartop(Z,S), not(X=Z), delete(on(X,Y,B),S,S2).
successor(S,[on(X,surface,B2)|S2]) :- member(on(X,Y,B),S),
  cleartop(X,S), member(bolt(B2),S), not(member(on(Z,W,B2),S)),
  delete(on(X,Y,B),S,S2).


cleartop(Part,State) :- not(member(on(X,Part,B),State)).


delete(X,[X|L],L).
delete(X,[X|L],L2) :- delete(X,L,L2).
delete(X,[Y|L],[Y|L2]) :- not(X=Y), delete(X,L,L2).
```

In the preceding, **cleartop** just checks that there's nothing resting on a part. The **delete** predicate comes from Section 5.6: it removes all occurrences of its first argument (an input) from the list that is its second argument (an input), binding the resulting list to the third argument (the output).

Implementing depth-first search

Prolog interpreters work depth-first, so it isn't hard to implement a general depth-first search facility in Prolog. As with the programs in Chapter 7, we'll divide code for a search into two files: a problem-dependent or "problem-defining" file containing, **successor**, **goalreached**, **eval**, and **cost** definitions discussed in Section 10.1, and a problem-independent file containing search machinery. Here is the problem-independent

depth-first-search file in its entirety | REFERENCE 1|: .FS | REFERENCE 1| When using all the programs in this chapter, be careful not to redefine or duplicate definitions of the predicate names used here, or you can get into serious trouble. That particularly applies to duplication of the classic list-predicate definitions **member**, **length**, and **append**. .FE

```
/* Problem-independent code for depth-first search */
depthsearch(Start,Ans) :- depthsearch2(Start,[Start],Ans).


depthsearch2(State,Statelist,Statelist) :- goalreached(State).
depthsearch2(State,Statelist,Ans) :- successor(State,Newstate),
 not(member(Newstate,Statelist)),
 depthsearch2(Newstate,[Newstate|Statelist],Ans).


member(X,[X|L]).
member(X,[Y|L]) :- member(X,L).
```

Predicate **depthsearch** is the top level. Its first argument is an input, bound to a description of the starting state, and its second argument is the output, bound when search is done to the solution path in reverse order. The **depthsearch** rule just initializes a third argument (the middle one) for the predicate **depthsearch2**.

The recursive **depthsearch2** predicate does the real work of the program. Let's first show it declaratively; the next section will explain it mostly procedurally. Its three arguments are the current state (an input), the path followed to this state (an input), and the eventual path list found (an output). The first **depthsearch2** rule says that if the current state is a goal state, bind the output (third argument) to the second argument, the list of states we went through to get here. Otherwise, the second rule says to find some successor of the current state not previously encountered on the path here (that is, avoid infinite loops), put this successor on the front of the path list, and recursively search for the goal from the new state. The **member** predicate is from the last section.

The key to this program is the **successor** predicate expression in the second **depthsearch2** rule. Backing up in a search problem means backtracking to that expression. Whenever a state has no successors, or all its successors have been tried and found not to lead to a goal, the second **depthsearch2** rule fails. Since it's the *last* **depthsearch2** rule, the program returns to where it was called--or it "backs up". If the call of **depthsearch2** was (as usually) a recursive one from the same second **depthsearch2** rule at the next highest level of recursion, backtracking goes to the **not** (which like all **not**s, can never succeed on backtracking), and then immediately to the **successor** predicate. If another successor can be found for this earlier state, it is then taken. Otherwise, *this* invocation of the rule also fails, and backing up and backtracking happens again.

A depth-first example

Let's simulate the depth-first program on an example, to illustrate its procedural interpretation. The previous city-route example is a little too complicated for a good show, so let's try the following. Suppose in some city (see Figure 10-3) that two-way streets connect intersection **a** with intersection **b**, intersection **b** to intersection **d**, and intersection **d** to intersection **e**. Suppose also that one-way streets connect intersection **b** to intersection **c**, and intersection **a** to intersection **d**. Then the following **successor** facts hold; assume they're put in this order.

```
successor(a,b).
```

```
successor(b,a).
successor(b,c).
successor(a,d).
successor(b,d).
successor(d,b).
successor(d,e).
successor(e,d).
```

Assume for this problem that the starting state is **a**, and there is only one goal state **e**. Then the problem-definition file must also include

```
goalreached(e).
```

Now let's follow what happens when we query

```
?- depthsearch(a,Answer).
```

and the problem-independent file of the last section is loaded in addition to the previous problem-dependent specification. The action is summarized in Figure 10-4.

1. The predicate **depthsearch** is called with its first argument bound to **a** and its second argument unbound. So predicate **depthsearch2** is called with first argument **a**, second argument **[a]**, and third argument unbound.

2. The goal is not reached in state **a**, so the second rule for **depthsearch2** is tried.

3. In the **successor** facts, the first successor of state **a** is **b**, and **b** is not a member of the list of previous states, **[a]**. So **depthsearch2** is recursively called with first argument **b**, second argument **[b,a]**, and third argument unbound.

4. For this second level of **depthsearch2** recursion, the state **b** is not a goal state; the first successor listed for **b** is **c**, and **c** isn't in the list of previous states, **[b,a]**. So we recursively call **depthsearch2**, now with first argument **c**, second argument **[c,b,a]**, and third argument unbound.

5. For this third level of recursion, new state **c** is not a goal state nor does it have successors, so both rules for **depthsearch2** fail. We must backtrack, "backing up" to the previous state **b** at the second level, and hence to the recursive **depthsearch2** call in the last line of the last **depthsearch2** rule.

6. The **not** fails on backtracking, as all **not**s do, so we backtrack further to the **successor** predicate in the second rule for **depthsearch2**, which chooses successors. Backtracking to here means we want a different successor for state **b** than **c**. And the only other successor of **b** is **d**. So we resume forward progress through the second **depthsearch2** rule with Newstate bound to **d**. This **d** is not in the list of previously visited states **[b,a]**, so we recursively call **depthsearch2** with first argument **d**, second argument **[d,b,a]** (**c** was removed in backtracking), and third argument unbound.

7. For this new third-level call, the new state **d** is not a goal, so we find a successor of it. Its first-listed successor is **b**, but **b** is a member of the list of previous states **[d,b,a]**, so we backtrack to find another successor.

8. The only other successor fact for **d** mentions state **e**. This isn't a member of **[d,b,a]**, so we recursively call **depthsearch2** with the **e** as first argument, **[e,d,b,a]** as second argument, and an unbound variable (still) as third argument.

9. But for this fourth level of recursion, state **e** is a goal state, and the **goalreached** predicate succeeds. So the first rule for **depthsearch2** succeeds, binding the third argument **Statelist** (finally!) to the list of states visited on the path here in reverse order, **[e,d,b,a]**. Now all other levels of **depthsearch2** recursion succeed because the recursive call was the last predicate expression in the rule (in other words, we always did tail recursion).

l0. So query variable **Answer** is bound to **[e,d,b,a]**.

Notice this is not the shortest solution to the problem, as is common with depth-first search.

Implementing breadth-first search

We can write code for problem-independent breadth-first search, to load as an alternative to the depth-first problem-independent code.

Recall from the last chapter that breadth-first search finds states level-by-level (that is, by distance in number of branches from the starting state). To do this, it must keep facts about all states found but whose successors haven't yet been found. Those states are an *agenda*; each represents further work to do. One simple way to implement breadth-first search is to make the agenda a queue (see Appendix C), a data structure for which the first thing added is always the first thing removed. We begin with a queue consisting of just the starting state, and anytime we find successors, we put them on the end of the queue. That way we are guaranteed to not try (find successors of) any states at level |N| until all states at level |N-1| have been tried.

We can implement agendas in Prolog, by facts with predicate name **agenda**, one for each unexplored (successors-not-found) state. To get the effect of a queue, we can put new states at the end of the agenda by the built-in predicate **assertz** (introduced in Section 6.1), so the first fact will always be the one that has been on the agenda the longest, and the first we'll find when we query the agenda facts. Predicate **agenda** can have two arguments: a state, and the path followed to reach it. As with depth-first search, the second argument is needed because (1) checking states against it prevents some of the possible infinite loops, and (2) its value for the goal state is the answer to the search problem.

We also should keep **oldagenda** facts. These, with the same two arguments as **agenda** facts, can record "exhausted" states, states for which we have found all successors. Checking against **oldagenda** facts before adding a new **agenda** fact prevents other infinite loops. This checking also ensures that any path we find to a state **S** has the fewest number of branches of any path to **S**, because it was found first and breadth-first works level by level.

Here is the problem-independent breadth-first search code | REFERENCE 2|, whose predicate hierarchy is given in Figure 10-5. (Those strange exclamation points "**!**" will be explained in Section 10.7, and the **bagof** in Section 10.6.) .FS | REFERENCE 2| This won't work for some Prolog dialects, those that can't handle dynamic additions of new backtracking points with **assertz**. For such implementations, we can get breadth-first search from the best-first search program **bestsearch** given later in this chapter, by including two extra lines with it:

eval(S,T) :- time(T), retract(time(T)), T2 is T+1, asserta(time(T2)).
eval(S,0) :- not(time(T)), asserta(time(1)). .FE

```
/* Problem-independent breadth-first search */
breadthsearch(Start,Ans) :- cleandatabase,
  asserta(agenda(Start,[Start])), agenda(State,Oldstates),
  find_successors(State,Oldstates,Newstate),
  goalreached(Newstate), agenda(Newstate,Ans),
  retract(agenda(Newstate,Ans)),
  asserta(oldagenda(Newstate,Ans)), measurework.


find_successors(State,Oldstates,Newstate) :-
  successor(State,Newstate), not(State = Newstate),
  not(agenda(Newstate,S)), not(oldagenda(Newstate,S)),
  assertz(agenda(Newstate,[Newstate|Oldstates])).
find_successors(State,Oldstates,Newstate) :-
  retract(agenda(State,Oldstates)),
  asserta(oldagenda(State,Oldstates)), fail.


cleandatabase :- abolish(oldagenda,2), abolish(agenda,2), !.
cleandatabase :- abolish(agenda,2), !.
cleandatabase.


measurework :- bagof([X,Y],agenda(X,Y),Aset), length(Aset,Len),
 bagof([X2,Y2],oldagenda(X2,Y2),A2set), length(A2set,Len2),
 write(Len), write(' incompletely examined state(s) and '),
 write(Len2),write(' examined state(s).'),!.
measurework :- bagof([X,Y],oldagenda(X,Y),Aset),
 length(Aset,Len), write('no incompletely examined states and '),
 write(Len), write(' examined state(s).'),!.


length([],0).
length([X|L],N) :- length(L,N2), N is N2+1.
```

The predicate **breadthsearch** starts by removing any **agenda** and **oldagenda** facts remaining from previous searches. It "seeds" or starts the agenda with a single item, the starting state. It then spends most of its time bouncing back and forth between the next three predicate expressions **agenda**, **find_successors**, and **goalreached**. The **agenda** retrieves an agenda state, **find_successors** finds a successor of it (as explained in a moment), and **goalreached** checks whether it's done. Most of the time it won't be. So most of the time it backtracks to the **find_successors** call to find another successor, or if there aren't any more, it backtracks to the **agenda** call to pick the next state on the agenda. When a goal state is found, it cleans up the agenda, and binds the answer variable (the second argument to **breadthsearch** as with **depthsearch**) to the path used to get there. Finally, it prints the size of the agenda and the oldagenda to show how much work it did.

The two **find_successors** rules are the key to the program. Function predicate **find_successors** has three arguments: the current state (an input), the path there (an input), and a successor state (an output). The right side of the first **find_successors** rule calls first on the **successor** definition, just as in the depth-first search program. If a successor is found, it is checked to be (1) not the current state, (2) not on the agenda, and (3) not on the oldagenda. Only if these tests succeed is the new successor added to the agenda. The first **find_successors** rule is repeatedly backtracked into to generate all the successors of a state; this backtracking

is forced by the usually-failing **goalreached** expression in **breadthsearch**.

The second **find_successors** rule applies whenever the first fails, or whenever no further successors can be found for a state. It removes the "exhausted" state from the agenda and adds it to the oldagenda. Then by a **fail**, it forces the top-level predicate **breadthsearch** to pick a new state to find successors of. As we explained before, the next state picked will always be the oldest remaining agenda fact because of the **assertz**.

If the agenda ever becomes empty (that is, there are no new states to be found), then the **agenda** in **breadthsearch** fails, and then the **asserta** fails (there's never a new way to assert something), and then the **cleandatabase** fails (though we can't explain why this last for several pages yet). So the interpreter would type **no**.

As an example, here is the result of running the breadth-first program on the bolts problem defined in Section 10.2. Three solution paths were found, differing only in when part **e** is removed relative to the removals of parts **a** and **b**. (Carriage returns have been added to improve readability.)

```
?- breadthsearch([on(a,b,bolt1),on(b,c,bolt1),on(c,surface,bolt1),
  on(d,e,bolt2),on(e,f,bolt2),on(f,surface,bolt2)],Answer).
81 incompletely examined state(s) and 1 examined state(s).
Answer=
[[on(c,e,bolt2),on(d,surface,none),on(b,surface,none),
  on(a,surface,none),on(e,f,bolt2),on(f,surface,bolt2)],
[on(d,surface,none),on(b,surface,none),on(a,surface,none),
  on(c,surface,bolt1),on(e,f,bolt2),on(f,surface,bolt2)],
[on(b,surface,none),on(a,surface,none),on(c,surface,bolt1),
  on(d,e,bolt2),on(e,f,bolt2),on(f,surface,bolt2)],
[on(a,surface,none),on(b,c,bolt1),on(c,surface,bolt1),
  on(d,e,bolt2),on(e,f,bolt2),on(f,surface,bolt2)],
[on(a,b,bolt1),on(b,c,bolt1),on(c,surface,bolt1),
  on(d,e,bolt2),on(e,f,bolt2),on(f,surface,bolt2)]] ;

102 incompletely examined state(s) and 2 examined state(s).
Answer=
[[on(c,e,bolt2),on(b,surface,none),on(d,surface,none),
  on(a,surface,none),on(e,f,bolt2),on(f,surface,bolt2)],
[on(b,surface,none),on(d,surface,none),on(a,surface,none),
  on(c,surface,bolt1),on(e,f,bolt2),on(f,surface,bolt2)],
[on(d,surface,none),on(a,surface,none),on(b,c,bolt1),
  on(c,surface,bolt1),on(e,f,bolt2),on(f,surface,bolt2)],
[on(a,surface,none),on(b,c,bolt1),on(c,surface,bolt1),
  on(d,e,bolt2),on(e,f,bolt2),on(f,surface,bolt2)],
[on(a,b,bolt1),on(b,c,bolt1),on(c,surface,bolt1),
  on(d,e,bolt2),on(e,f,bolt2),on(f,surface,bolt2)]] ;

152 incompletely examined state(s) and 3 examined state(s).
Answer=
[[on(c,e,bolt2),on(b,surface,none),on(a,surface,none),
  on(d,surface,none),on(e,f,bolt2),on(f,surface,bolt2)],
[on(b,surface,none),on(a,surface,none),on(d,surface,none),
  on(c,surface,bolt1),on(e,f,bolt2),on(f,surface,bolt2)],
[on(a,surface,none),on(d,surface,none),on(b,c,bolt1),
  on(c,surface,bolt1),on(e,f,bolt2),on(f,surface,bolt2)],
[on(d,surface,none),on(a,b,bolt1),on(b,c,bolt1),
```

```
  on(c,surface,bolt1),on(e,f,bolt2),on(f,surface,bolt2)],
[on(a,b,bolt1),on(b,c,bolt1),on(c,surface,bolt1),
  on(d,e,bolt2),on(e,f,bolt2),on(f,surface,bolt2)]]
```

Collecting all items that satisfy a predicate expression

A feature of the **breadthsearch** program we haven't explained is the **bagof** predicate in the **measurework** rules. This predicate we used before in Chapter 8 to implement "or-combination", and it is built-in in most Prolog dialects (though easily defined in Prolog). It collects into a list all the values for some variable that satisfy a predicate expression, much like the **forall** (defined in Section 7.12) in reverse. Predicate **bagof** takes three arguments: an input variable, an input predicate expression containing that variable, and an output list to hold all possible bindings of that variable which satisfy that expression. (Some variants of **bagof** delete duplicates in the result.)

Here's an example. Suppose we have this database:

```
boss_of(mary,tom).
boss_of(mary,dick).
boss_of(dick,harry).
boss_of(dick,ann).
```

Suppose we want a list of all people that Mary is the boss of. We can type

```
?- bagof(X,boss_of(mary,X),L)
```

and the interpreter will type

```
L=[tom,dick]
```

and **X** won't be printed because it's just a placeholder.

We can put, within the expression that is the second argument to **bagof**, variables besides the one we are collecting; the interpreter will try to bind them too. So if, for the preceding database, we type

```
?- bagof(X,boss_of(Y,X),L).
```

the interpreter will type

```
Y=mary, L=[tom,dick]
```

for its first answer. If we then type a semicolon, it will type

```
Y=dick, L=[harry,ann]
```

for its second answer.

The first argument to **bagof** can be a list. That is, we can search for a set of values satisfying a predicate instead of just one value. For instance, we can query

```
?- bagof([X,Y],boss_of(X,Y),L).
```

and receive the answer

```
L=[[mary,tom],[mary,dick],[dick,harry],[dick,ann]]
```

This query form is used in the **measurework** rules in the **breadthsearch** program.

The **bagof** predicate can be defined this way (provided you have no other predicate named **zzz** in your program):

```
bagof(X,P,L) :- asserta(zzz([])), fail.
bagof(X,P,L) :- call(P), zzz(M), retract(zzz(M)),
  asserta(zzz([X|M])), fail.
bagof(X,P,L) :- zzz(L), retract(zzz(L)).
```

The cut predicate

We still must explain those strange exclamation points (the "**!**") in the **breadthsearch** program. These are a special built-in predicate of no arguments called the *cut*, whose meaning is exclusively procedural. The cut always succeeds when the interpreter first encounters it, but has a special side effect: it prevents backtracking to it by throwing away the necessary bookkeeping information. This can improve the efficiency of Prolog programs, but it is also necessary to make some programs work properly, those for which backtracking just doesn't make sense.

Usually the cut predicate expression is last in a rule, as in **cleandatabase** and the two **measurework** rules in the **breadthsearch** program. It can be paraphrased as: "Don't ever backtrack into this rule. What's more, don't ever try another rule to satisfy the goal that this rule tried to satisfy. That goal is dead. So fail it." (Note that a cut has no effect on the next call of the rule, even a call with the same arguments as before: the prohibition of backtracking just applies to the call in which the cut was encountered.) So a cut symbol forces behavior like that of a subprocedure in a conventional programming language, in that once a subprocedure is done it can only be reentered by a different call--except that a Prolog "subprocedure" is all the rules with the same left-side predicate name, not just a single rule. So a cut at the end of a rule means you want only one solution to the query of that rule. This often is true for "existential quantification" queries in which we check existence of something of a certain type, and we don't care what. For instance, the **member** predicate from Section 5.5

```
member(X,[X|L]).
member(X,[Y|L]) :- member(X,L).
```

is often used this way, when it is queried with both arguments bound. For instance:

```
?- member(b,[a,b,c,b,e,a,f).
```

Recursion will find the first **b** in the list and the interpreter will type **yes**. If we were to type a semicolon, recursion would find the second occurrence of **b** and the interpreter would type **yes** again. But finding that second occurrence doesn't make sense in most applications; the answer to the query must be the same, after all. A backtracking **member** is useless in our depth-first and breadth-first search programs, where **member** is enclosed in a **not**, since the Prolog interpreter doesn't backtrack into **not**s. And backtracking into **member** is unnecessary whenever there can't be duplicates in a list. (But we do need a backtracking **member** in the bolts example of Section 10.2, to choose alternative parts to move by querying **member** with an unbound first argument.) A non-backtracking **member** can be obtained by just inserting a cut symbol in the backtracking **member**:

```
singlemember(X,[X|L]) :- !.
```

```
singlemember(X,[Y|L]) :- singlemember(X,L).
```

We don't need a cut symbol at the end of the second rule, because when it fails no more possibilities remain.

We don't actually need a cut symbol to define **singlemember**, for we could say equivalently:

```
singlemember(X,[X|L]).
singlemember(X,[Y|L]) :- not(X=Y), singlemember(X,L).
```

But this is slower because the extra expression **not(X=Y)** must be tested on every recursion; the cut predicate expression is done at most once for any query.

A related use of the cut predicate is to do something only once instead of repeatedly. For instance, here's the predicate from Section 5.6 that deletes all occurrences of an item **X** from a list **L**:

```
delete(X,[],[]).
delete(X,[X|L],M) :- delete(X,L,M).
delete(X,[Y|L],[Y|M]) :- not(X=Y), delete(X,L,M).
```

Suppose we want a predicate that deletes only the first item **X** from list **L**. We can remove the recursion in the second rule, remove the **not(X=Y)** in the third rule, and insert a cut symbol:

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

The cut symbol is important here, because if we just omit it like this:

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

then **deleteone** will give a correct first answer, but wrong subsequent answers on backtracking, just like the similar mistake-making **delete** discussed in Section 5.6.

The cut predicate can also be used merely to improve efficiency. Recall the definition of the maximum of a list in Section 5.5:

```
max([X],X).
max([X|L],X) :- max(L,M), X>M.
max([X|L],M) :- max(L,M), not(X>M).
```

When the first two rules fail, the computation of the maximum of a list is done twice: once in the second rule, and once in the third rule. This is wasteful. So we can define it:

```
max([X],X) :- !.
max([X|L],M) :- max(L,M), not(X>M), !.
max([X|L],X).
```

Here we've changed the order of the last two rules and removed the redundant **max** call. The cuts guarantee that if we ever backtrack into **max**, we won't start taking the third rule when the second rule was taken before, and thereby get wrong answers.

A cut predicate can be put in the middle of a rule. Then it means that backtracking is allowed to the right of

it, but that if the interpreter ever tries to backtrack to its left, both the rule and the goal that invoked it will fail unconditionally.

Nothing in life is free, so it's not surprising that the efficiency advantages of the cut predicate have the associated disadvantage of restricting multiway use (see Section 3.3) of predicate definitions. That's because the cut is a purely procedural feature of Prolog, with no declarative meaning. But if you're sure you'll only query a definition with one particular pattern of bindings, multiway use is not an issue, and cut predicates can be used freely to improve efficiency.

Iteration with the cut predicate (*)

The cut predicate provides a way to write iterative Prolog programs in a way more general than the **forall** and **doall** of Section 7.12 and **bagof** of Section 10.6. It gives a sometimes better way to repeat things than by backtracking, since backtracking doe things in reverse order, and that can be confusing or even wrong.

To get a true "do-until", or in other words to repeatedly query some predicate expression Pred until some other predicate expression **done** holds, query predicate **iterate** with argument Pred:

```
iterate(Pred) :- repeat, iterate2(Pred), done.
iterate2(Pred) :- call(Pred), !.
repeat.
repeat :- repeat.
```

Here **repeat** is a predicate that always succeeds, and what's more, always succeeds on backtracking (unlike **1=1** which always succeeds once, but fails on backtracking because it can't succeed in a new way). The **repeat** is built-in in many Prolog dialects, but it's easy to define as you see.

Predicate **iterate** will hand the expression Pred to **iterate2** for execution using the **call** predicate explained in Section 7.12. Then **iterate** checks the **done** condition, which usually fails. At this point, the cut in **iterate** is important, because it prevents backtracking into **iterate2** and Pred. So **iterate2** fails, and the interpreter returns to the **repeat**. But **repeat** always succeeds anew on backtracking (it just recurses once more as a new way to succeed), and so the interpreter returns to **iterate2**, and Pred is executed again *in the forward direction*. So the cut predicate forces the interpreter to execute Pred like within a loop in a conventional programming language: always forward. (Note that Pred can contain arguments.)

One disadvantage of the preceding is that **iterate** can never fail. This would cause an infinite loop if **done** has a bug preventing it from ever succeeding. So we might distinguish **donegood** and **donebad** conditions, both user-defined, for when the iteration should stop with success and failure respectively:

```
iterate(Pred) :- repeatcond, iterate2(Pred), donegood.
iterate2(Pred) :- call(Pred), !.
repeatcond.
repeatcond :- not(donebad), repeatcond.
```

Another kind of iteration increases a counter at each iteration, like the "FOR" construct in Pascal and the "DO" construct in Fortran which iterate for K=1 to N:

```
foriterate(Pred,N) :- asserta(counter(0)), repeat, counter(K),
  K2 is K+1, retract(counter(K)), asserta(counter(K2)),
  iterate2(Pred), K=<N.
```

```
iterate2(Pred) :- call(Pred), !.
repeat.
repeat :- repeat.
```

To access the counter at any time inside the rules invoked by calling Pred, you query predicate **counter**.

Implementing best-first search (*)

Now we can show our best-first search program. To use it you need (besides **successor** and **goalreached** definitions) a definition of a function predicate **eval** of two arguments. As we said in Section 10.1, the first argument to **eval** is an input state, and its second is an output number, a nonnegative evaluation of that state.

The best-first program keeps an agenda of states like the breadth-first program, but each agenda fact has a third argument holding the evaluation function value for the state. (It makes sense to compute this when we find the state and put it on the agenda, so we only do it once per state.) And when we select a state from an agenda with the **pick_best_state** predicate, we must take the state with the minimum evaluation function value, not just the first one on the agenda. Our best-first search program also has iteration in several places where the breadth-first search program used recursion, so it's a more efficient program. Here's the best-first search program, whose predicate hierarchy appears in Figure 10-6:

```
/* Problem-independent best-first search */
bestsearch(Start,Goalpathlist) :- cleandatabase,
  add_state(Start,[]), repeatifagenda,
  pick_best_state(State,Pathlist),
  add_successors(State,Pathlist), agenda(State,Goalpathlist,E),
  retract(agenda(State,Goalpathlist,E)), measurework.


pick_best_state(State,Pathlist) :-
  asserta(beststate(dummy,dummy,dummy)),
  agenda(S,SL,E), beststate(S2,SL2,E2), special_less_than(E,E2),
  retract(beststate(S2,SL2,E2)), asserta(beststate(S,SL,E)), fail.
pick_best_state(State,Pathlist) :- beststate(State,Pathlist,E),
  retract(beststate(State,Pathlist,E)), not(E=dummy), !.


add_successors(State,Pathlist) :- goalreached(State), !.
add_successors(State,Pathlist) :- successor(State,Newstate),
  add_state(Newstate,Pathlist), fail.
add_successors(State,Pathlist) :-
  retract(agenda(State,Pathlist,E)),
  asserta(usedstate(State)), fail.


add_state(Newstate,Pathlist) :- not(usedstate(Newstate)),
  not(agenda(Newstate,P,E)), eval(Newstate,Enew),
  asserta(agenda(Newstate,[Newstate|Pathlist],Enew)), !.
add_state(Newstate,Pathlist) :- not(eval(Newstate,Enew)),
  write('Warning: your evaluation function failed on state '),
  write(Newstate), nl, !.

/* Utility functions */
repeatifagenda.
repeatifagenda :- agenda(X,Y,Z), repeatifagenda.
```

```
special_less_than(X,dummy) :- !.
special_less_than(X,Y) :- X<Y.


cleandatabase :- checkabolish(agenda,3), checkabolish(usedstate,1),
  checkabolish(beststate,1), checkabolish(counter,1).


checkabolish(P,N) :- abolish(P,N), !.
checkabolish(P,N).


measurework :- countup(agenda(X,Y,Z),NA), countup(usedstate(S),NB),
  write(NA), write(' incompletely examined state(s) and '),
  write(NB),write(' examined state(s)'), !.


countup(P,N) :- asserta(counter(0)), call(P), counter(K),
  retract(counter(K)), K2 is K+1, asserta(counter(K2)), fail.
countup(P,N) :- counter(N), retract(counter(N)), !.
```

The top-level **bestsearch** predicate iterates by repeatedly picking a state from the agenda. It initializes and cleans up around a kernel of commands repeatedly invoked. But the iteration is done differently from the **breadthsearch** program, with the **bestsearch** bouncing among a **repeatifagenda** predicate on the left, the **pick_best_state** predicate, and an **add_successors** on the right. The **repeatifagenda** is an instance of the **repeatcond** discussed in the last section.

The **pick_best_state** chooses the minimum-evaluation state by iterating over the agenda states; the cut predicate at the end of its definition ensures that it finds only one such state per call. The **add_successors** checks whether some state **S** is a goal state (then succeeding), and otherwise adds all the acceptable successors of **S** to the agenda (then failing); the predicate **add_state** runs the necessary acceptance checks on a new state before adding it to the agenda. When **add_successors** finally succeeds, the path associated with its state argument must be the solution, so this is retrieved and bound to the answer **Goalpathlist**.

As an example, let's use best-first search on the bolts problem of Section 10.2. We need an evaluation function. Our goal is to get part **c** on part **e**, so we could take the sum of the number of parts on top of both **c** and **e**. That is, we could measure the degree of "burial" of each. But then many states have evaluation zero. Instead, let's try:

```
eval(S,0) :- goalreached(S).
eval(S,N) :- burial(c,S,N1), burial(e,S,N2), N is N1+N2+1.


burial(P,S,0) :- cleartop(P,S).
burial(P,S,N) :- member(on(X,P,B),S), burial(X,S,N2), N is N2+1.
```

Running **bestsearch** on the same starting state as before, we get the third answer for breadth-first search:

```
?- bestsearch([on(a,b,bolt1),on(b,c,bolt1),on(c,surface,bolt1),
  on(d,e,bolt2),on(e,f,bolt2),on(f,surface,bolt2)],A).
23 incompletely examined state(s) and 4 examined state(s)
A=[[on(c,e,bolt2),on(b,surface,none),on(a,surface,none),
```

```
  on(d,surface,none),on(e,f,bolt2),on(f,surface,bolt2)],
[on(b,surface,none),on(a,surface,none),on(d,surface,none),
  on(c,surface,bolt1),on(e,f,bolt2),on(f,surface,bolt2)],
[on(a,surface,none),on(d,surface,none),on(b,c,bolt1),
  on(c,surface,bolt1),on(e,f,bolt2),on(f,surface,bolt2)],
[on(d,surface,none),on(a,b,bolt1),on(b,c,bolt1),
  on(c,surface,bolt1),on(e,f,bolt2),on(f,surface,bolt2)],
[on(a,b,bolt1),on(b,c,bolt1),on(c,surface,bolt1),
  on(d,e,bolt2),on(e,f,bolt2),on(f,surface,bolt2)]]
```

But look how many fewer states were found--27 (23+4) versus 82. That's more efficient search.

Implementing A* search (*)

A* search is like best-first except we must add path cost to the evaluation function. So as we said, the user must define a **cost** function predicate with two arguments, an input path and an output holding the computed cost of that path.

Our A* program is intended to be forgiving. So it still works if you don't give it a lower-bound cost function, though you may need to type semicolons to get the right (lowest-cost) answer. Also without a lower-bound evaluation function, the first path found to any state **S** may not be lowest-cost, so if we find a lower-cost path to **S** later, we must revise the path lists of everything on the agenda mentioning **S**.

Here's the program (whose predicate hierarchy is summarized in Figure 10-7):

```
/* Problem-independent A* search code. */
/* Note: "cost" must be nonnegative.  The "eval" should be a lower */
/* bound on cost in order for the first answer found to be */
/* guaranteed optimal, but the right answer will be reached */
/* eventually otherwise.  */


astarsearch(Start,Goalpathlist) :- cleandatabase,
  add_state(Start,[]), repeatifagenda,
  pick_best_state(State,Pathlist),
  add_successors(State,Pathlist), agenda(State,Goalpathlist,C,D),
  retract(agenda(State,Goalpathlist,C,D)), measurework.


pick_best_state(State,Pathlist) :-
  asserta(beststate(dummy,dummy,dummy)),
  agenda(S,SL,C,D), beststate(S2,SL2,D2), special_less_than(D,D2),
  retract(beststate(S2,SL2,D2)), asserta(beststate(S,SL,D)), fail.
pick_best_state(State,Pathlist) :- beststate(State,Pathlist,D),
  retract(beststate(State,Pathlist,D)), not(D=dummy), !.


add_successors(State,Pathlist) :- goalreached(State), !.
add_successors(State,Pathlist) :- successor(State,Newstate),
  add_state(Newstate,Pathlist), fail.
add_successors(State,Pathlist) :-
  retract(agenda(State,Pathlist,C,D)),
  asserta(usedstate(State,C)), fail.
```

```
/* If you are sure that your evaluation function is always
/* a lower bound on the cost function, then you can delete the */
/* first rule for "agenda_check" and the first rule for */
/* "usedstate_check", and delete the entire definitions of */
/* "fix_agenda", "replace_front", and "append". */
add_state(Newstate,Pathlist) :- cost([Newstate|Pathlist],Cnew), !,
  agenda_check(Newstate,Cnew), !,
  usedstate_check(Newstate,Pathlist,Cnew), !,
  eval(Newstate,Enew), D is Enew + Cnew,
  asserta(agenda(Newstate,[Newstate|Pathlist],Cnew,D)), !.
add_state(Newstate,Pathlist) :-
  not(cost([Newstate|Pathlist],Cnew)),
  write('Warning: your cost function failed on path list '),
  write(Pathlist), nl, !.
add_state(Newstate,Pathlist) :- not(eval(Newstate,Enew)),
  write('Warning: your evaluation function failed on state '),
  write(Newstate), nl, !.


agenda_check(S,C) :- agenda(S,P2,C2,D2), C<C2,
  retract(agenda(S,P2,C2,D2)), !.
agenda_check(S,C) :- agenda(S,P2,C2,D2), !, fail.
agenda_check(S,C).


usedstate_check(S,P,C) :- usedstate(S,C2), C<C2,
  retract(usedstate(S,C2)), asserta(usedstate(S,C)), !,
  fix_agenda(S,P,C,C2).
usedstate_check(S,P,C) :- usedstate(S,C2), !, fail.
usedstate_check(S,P,C).


fix_agenda(S,P,C,OldC) :- agenda(S2,P2,C2,D2),
  replace_front(P,S,P2,Pnew), cost(Pnew,Cnew),
  Dnew is D2+C-OldC, retract(agenda(S2,P2,C2,D2)),
  asserta(agenda(S2,Pnew,Cnew,Dnew)), fail.


replace_front(P,S,P2,Pnew) :- append(P3,[S|P4],P2),
  append(P,[S|P4],Pnew), !.

/* Utility functions */
repeatifagenda.
repeatifagenda :- agenda(X,Y,Z,W), repeatifagenda.


special_less_than(X,dummy) :- !.
special_less_than(X,Y) :- X<Y.


cleandatabase :- checkabolish(agenda,4), checkabolish(usedstate,2),
  checkabolish(beststate,1), checkabolish(counter,1).


checkabolish(P,N) :- abolish(P,N), !.
checkabolish(P,N).
```

```
measurework :- countup(agenda(X,Y,C,D),NA),
  countup(usedstate(S,C),NB), write(NA),
  write(' incompletely examined state(s) and '),
  write(NB),write(' examined state(s).'), !.


countup(P,N) :- asserta(counter(0)), call(P), counter(K),
  retract(counter(K)), K2 is K+1, asserta(counter(K2)), fail.
countup(P,N) :- counter(N), retract(counter(N)), !.


append([],L,L).
append([I|L1],L2,[I|L3]) :- append(L1,L2,L3).
```

Implementing search with heuristics (*)

Heuristics define a policy for choosing things from the agenda in search. So searching with heuristics can be implemented by redefining the **pick_best_state** in the best-first program, and leaving the rest of the program unchanged. Specifically, it should pick the first state on the agenda for which no other state is preferred by heuristics:

```
pick_best_state(State,Pathlist) :-
  agenda(State,Pathlist,E), not(better_pick(State,Pathlist,E)), !.
better_pick(X,Y,E) :- agenda(X2,Y2,E2), not(X=X2), prefer(X2,Y2,E2,X,Y,E).
```

This same idea was used to implement meta-rules in Section 7.16, and a heuristic *is* a kind of meta-rule, a rule guiding usage of **successor** rules. We'll broaden the concept of a heuristic here to allow use of evaluation functions in its decision, to be fully general.

Such *extended heuristics* can be defined with a **prefer** predicate of six arguments: two groups of three where the first group represents an agenda item preferred to an agenda item represented by the second group. Each group represents the three arguments to agenda facts: state, path list, and evaluation-function value. (The last can be left unbound and ignored if numeric evaluation is hard.) So

```
prefer(X1,Y1,E1,X2,Y2,E2) :- E1 < E2.
```

implements best-first search, and

```
prefer(X1,Y1,E1,X2,Y2,E2) :- length(Y1,L1), length(Y2,L2), L1 < L2.
length([],0).
length([X|L],N) :- length(L,N2), N is N2+1.
```

implements breadth-first search, and

```
prefer(X1,Y1,E1,X2,Y2,E2) :- fail.
```

(or equivalently, no **prefer** rule at all) implements depth-first search. (Then the first agenda item will always be picked; it will always be the successor found most recently, thanks to the **asserta** in **add_state**.) These **prefer** rules can also refer to quite specific characteristics of a search problem, though then they can't be reused in other search problems as much. For instance for the bolts problem, we want part **c** on part **e**, so a heuristic would be to prefer a state in which the top of **e** is clear:

```
prefer(X1,Y1,E1,X2,Y2,E2) :- cleartop(e,X1), not(cleartop(e,X2)).
```

We can have several extended heuristics for a problem, and they will be tried in the order given. For instance we can do a best-first search with a breadth-first criterion to break ties between states with the same evaluation-function value:

```
prefer(X1,Y1,E1,X2,Y2,E2) :- E1 < E2.
prefer(X1,Y1,E,X2,Y2,E) :- length(Y1,L1), length(Y2,L2), L1 < L2.
length([],0).
length([X|L],N) :- length(L,N2), N is N2+1.
```

We could also combine breadth-first and best-first search in a different way:

```
prefer(X1,Y1,E1,X2,Y2,E2) :- Ep1 is E1+1, Ep1 < E2.
prefer(X1,Y1,E1,X2,Y2,E2) :-
  length(Y1,L1), length(Y2,L2), L1 < L2.
```

This says to prefer states according to the evaluation function if they are more than one unit apart, but otherwise pick the state closest to the start.

Sets of heuristics or extended heuristics can exhibit complex behavior; they can be an extensive rule-based system on their own. The most important heuristics (**prefer** rules) should be first, then general-purpose heuristics, and finally default heuristics. As rule-based system rules go, heuristics are powerful because they can be quite general: we can often reuse the same heuristic in many different problems.

The preceding heuristics choose among states. Other heuristics choose among branches of (operators applied to) a state, but these can often be transformed into the first kind by having them reference characteristics of the result of the desired branch.

Compilation of search (*)

Since a search problem can be seen as a rule-based system with **successor** rules, it isn't surprising that we can use compilation techniques for rule-based systems to make search faster. We can exploit all the "compiled forms" in the lower half of Figure 6-1, except virtual facts and decision lattices which require questioning (this is uncommon in search).

Caching is helpful and simple to apply. An agenda is one form of a cache, but we can extend an agenda to include entries from previous related problems. So as we solve problems, we gradually build up a library of solutions, some to full problems and some to subproblems, to make new problems easier. If this creates a large agenda, we can index or hash items for fast lookup. If this is still too slow or requires too much storage, we can cache selectively, using as criterion the amount of work needed to compute the item and perhaps statistics on how often it was used in problems. Caching criteria can also be suggested by the problem itself, as in route planning in a city for which it makes sense to cache routes between landmarks.

Partitioning or modularization is important in many areas of computer science, and search is no exception. Groups of related operators can be put into partitions, each loaded as needed. File loads can be done by a **consult** last in a **successor** rule. Partitions are essential for a system with many **successor** rules, like a general-purpose car repair system, for which quite different parts of the car require quite different repair techniques, so many operator have narrow applicability.

Parallelism can increase the speed of successor rules too. All four types of parallelism discussed in Section 6.9 can be used: partition parallelism, "and" parallelism, "or" parallelism and variable-matching parallelism.

Actual concurrency in execution is essential for speed in real-time applications. Good examples are the lattices used for high-performance speech-understanding systems, a kind of and-or-not lattice representing parallelism taken to the maximum extent. For a fixed vocabulary of words, the lattice corresponds to every possible sequence in which sounds can occur to express grammatical utterances using those words. Each node in the lattice represents a sound, and each branch represents a transition to a possible next sound. The speech to be understood is first broken into small varying-length segments representing homogeneous (constant-nature) intervals in the speech. A cost function is used that measures the "distance" any segment comes to some "ideal" sound. Speech recognition then becomes an optimal-path search problem, in which we want to find a path through a lattice that minimizes the sum of the branch costs. Such precompiled search lattices have many other real-time applications.

.SH Keywords:

frame problem
agenda of states
the ***bagof predicate***
the cut predicate
iteration

Exercises

10-1. (R,A,P) (a) Improve the depth-first search program to print out the answer path list in the forward (opposite) order. Print one state per line.

(b) Suppose we want a search answer to include names of operators used at each step in the solution. Explain how you can do this by writing the problem-dependent files in a certain way without making any changes at all to the problem-independent files.

(c) Suppose states are lists of facts in which the order of the facts doesn't matter--that is, states are described by sets of facts, not lists. Modify the depth-first search program to prevent going to "permutation" states that have the same facts as an already-examined state but in a different order.

10-2. (P) Country X and country Y are carefully negotiating an arms agreement. Each has missiles of varying strengths, each strength summarized by a single number. Each step in the negotiation does one of two things: (1) eliminates from both sides a single missile of the same numerical strength, or (2) eliminates from one side a missile of strength S and from the other side two missiles whose strength sums to S. Write a program that figures out a sequence of such negotiation steps that will eliminate all the missiles entirely from one or both of the sides, but making sure in the process that no side ever has more than one more missile than the other.

Use the depth-first program to do searching. Assume to start:

> *Country X: missile strengths 9, 11, 9, 3, 4, 5, 7, 18*
> *Country Y: missile strengths 12, 2, 16, 5, 7, 4, 20*

Type semicolons to your program to get alternative solutions.

Hints: use the ***member*** and **delete** predicates from Sections 5.5 and 5.6, and define a **twomembers** which says two things are different items from the same list.

10-3. Suppose we modify the bolts problem in Section 10.2 a bit. Suppose we have the same operators and starting state, but our goal is to get part **b** and part **c** on bolt 1, with **c** above **b** instead of vice versa.

(a) Give the **goalreached** definition.

(b) Give a corresponding **eval** definition.

(c) What answer will breadth-first search find?

10-4. (P) We didn't show the bolts problem being solved with depth-first search, and there's a good reason: it takes a long time to solve it, a thousand times more perhaps than breadth-first.

(a) Study the program to see why it's taking so much time for such a simple problem. Insert additional code to print out what it's doing periodically.

(b) One possible reason is the lack of unordered sets as opposed to lists in Prolog. Explain why this is a problem. Then explain how you would change the problem-independent code for depth-first search to eliminate this problem. Do so and see if efficiency improves.

(c) Another suggestion is to use heuristics to control the depth-first search. But the implementation of heuristics given in Section 10.11 isn't very helpful. Why not?

(d) Explain how, in place of meta-rules or heuristics, additional successor rules can be written. Do this for the bolts program and analyze the improvement in performance.

10-5. (R,A,P) Load into the Prolog interpreter the breadth-first program given in Section 10.5 and the following additional code:

```
glassesdepth(Cap1,Cap2,Goal,Ans) :- checkretract(cap1(C1)),
  checkretract(cap2(C2)), checkretract(goalvolume(G)),
  checkassert(cap1(Cap1)), checkassert(cap2(Cap2)),
  checkassert(goalvolume(Goal)), !, depthsearch([0,0],Ans).


glassesbreadth(Cap1,Cap2,Goal,Ans) :- checkretract(cap1(C1)),
  checkretract(cap2(C2)), checkretract(goalvolume(G)),
  checkassert(cap1(Cap1)), checkassert(cap2(Cap2)),
  checkassert(goalvolume(Goal)), !, breadthsearch([0,0],Ans).


goalreached([Goalvolume,V2]) :- goalvolume(Goalvolume).
goalreached([V1,Goalvolume]) :- goalvolume(Goalvolume).


successor([V1,V2],[Vsum,0]) :- V2 > 0, Vsum is V1 + V2,
  cap1(Cap1), Vsum =< Cap1.
successor([V1,V2],[0,Vsum]) :- V1 > 0, Vsum is V1 + V2,
  cap2(Cap2), Vsum =< Cap2.
successor([V1,V2],[Cap1,Vdiff]) :- V2 > 0, V1 >= 0,
  cap1(Cap1), Vdiff is V2 - ( Cap1 - V1 ), Vdiff > 0.
successor([V1,V2],[Vdiff,Cap2]) :- V1 > 0, V2 >= 0,
  cap2(Cap2), Vdiff is V1 - ( Cap2 - V2 ), Vdiff > 0.
successor([V1,V2],[Cap1,V2]) :- cap1(Cap1).
```

```
successor([V1,V2],[V1,Cap2]) :- cap2(Cap2).
successor([V1,V2],[0,V2]).
successor([V1,V2],[V1,0]).


checkassert(S) :- call(S), !.
checkassert(S) :- asserta(S).


checkretract(S) :- call(S), retract(S), fail.
checkretract(S).
```

This code defines a famous puzzle known as the water-glass problem. You have two empty glasses of capacities C1 and C2, a faucet to fill the glasses with water, and a drain. The glasses are not marked along the sides so you can't fill them a specified partial amount. The goal is to get a certain fixed quantity of liquid. To do this you can fill the glasses, empty the glasses, pour all of one glass into another, or fill one glass from the other. To run the program, type

```
?- glassesbreadth(<capacity-glass-1>,<capacity-glass-2>,<goal-amount>,<answer>).
```

for which the first three arguments are inputs, numbers you must fill in, and the last is the output.

(a) First study the situation in which you have two glasses of sizes 5 and 7. Call the predicate **glassesbreadth** with goal amounts equal to 0, 1, 2, 3, 4, 5, 6, 7, and 8, and construct a diagram of all possible states and their connections. Type the semicolon to obtain all distinct solutions. Note that **<answer>** represents a stack (i.e., it's in reverse chronological order). Describe the overall pattern in the diagram.

(b) Now use the depth-first program given in the chapter. Invoke it with

```
?- glassesdepth
  (<capacity-glass-1>,<capacity-glass-2>,<goal-amount>,<answer>).
```

Try it on all the situations considered previously, and draw its state diagram. Describe how the diagram is different from that for breadth-first search. In particular, how does the depth-first state diagram relate to the breadth-first diagram?

(c) Explain what each of the eight **successor** rules does.

(d) Modify the preceding code defining the operators so that the solutions found will have the desired quantity in the left glass (that is, the glass with the first-mentioned capacity). Test your code with the breadth-first search code.

10-6. (H,P) *Beam search* is a more efficient kind of breadth-first search, a sort of cross between breadth-first search and best-first search. Rather than putting all the successors of a given state on the agenda, it only puts the K best successors according to the evaluation function. Modify the breadth-first search program to do beam search.

10-7. (A,P) Modify the A* search program to do a branch-and-bound search.

10-8. In the city-route planning example of Section 10.1, the last line of the cost function definition seems very dangerous: it would seem to define an infinite loop. Such definitions of commutativity for predicates

always have such dangers.

(a) What advantage is gained by having such a line in the definitions?

(b) Why can the danger be ignored for this particular cost function definition used with the A* search program?

(c) Suppose we are doing route planning in which the routes are highways and the nodes are cities. If you can go from city X to city Y along route R you can also go from Y to X backward along R. So state transitions would be commutative, and we could write a single commutativity rule to state this, like the commutativity rule in the cost function definition. Why wouldn't this be a good idea, when it generally is a good idea with costs?

10-9. (R,P) (a) Consider a housekeeping robot that must clean a two-room suite (Figure 10-8). It must vacuum the floors, dust, and empty the trash baskets. Vacuuming the floor generates dust that goes into the trash basket of the room it is in. Emptying the trash baskets in each room requires a trip to the trash chute. Dusting puts dust on the floor, requiring vacuuming.

Assume the following energy costs:

--10 units to vacuum a single room (assume the robot has a built-in vacuum);
--6 units to dust a room (assume the robot has a built-in duster);
--3 units to pick up a trash basket (assume the robot can hold more than one);
--1 unit to put down a trash basket;
--3 units to travel between the two offices;
--8 units to travel between an office and the trash chute (assume there are doors to the trash chute from both rooms);
--5 units to dispose of the contents of a trash basket down the trash chute.

Use the following reasonable preconditions for actions (and no others):

--It doesn't make sense to vacuum or dust if the room isn't vacuumable or dusty respectively.
--It doesn't make sense to dust if a room is vacuumable.
--It doesn't make sense to pick up a trash basket if a room is vacuumable or dusty.
--It doesn't make sense to put down something you just picked up.

For the starting state, assume the robot is at the trash chute. Assume that both rooms need dusting. Office 1 has trash in its trash basket, but Office 2 doesn't; and the carpet in Office 1 does not need vacuuming, while the carpet in Office 2 does. The goal is to get everything dusted, every carpet vacuumed, and every basket emptied. The robot must finish at either room but not at the trash chute.

Set this problem up for the A* search program for this chapter. Give appropriate definitions, including a good evaluation function, one that gives a lower bound on the cost to a goal state. You may find you run out of space; if so, revise the program to eliminate unnecessary successors. Run the program and show your results.

Hints: make the first item in each state represent the last operator applied, or "none" for the starting state. Have the cost function use these to assign costs. You don't need **append** or **=** anywhere in your program; use the left sides of rules to get their effect. And in defining **successor**, it may improve efficiency to rule out new

states in which the facts are just permutations (ignoring the last-operator-applied item) of the facts in some other state already on the agenda or already visited, as in Exercise 10-1(c).

10-10. (A,P) Combining chemicals to react and make new chemicals is a kind of search. Find a good way to represent states and successor functions for operations including these:

> If we combine one mole of |Cl sub 2| and one mole of |H sub 2 O|, you get one mole of |H Cl O| and one mole of |H Cl|.
>
> If you combine one mole of |Ca C O sub 3| with two moles of |H Cl|, you get one mole of |Ca Cl sub 2|, one mole of |C O sub 2|, and one mole of |H sub 2 O|.
>
> If you combine two moles of |H sub 2 O sub 2| and one mole of |Mn O sub 2|, you get two moles of |H sub 2 O|, one mole of |O sub 2|, and one mole of |Mn O sub 2| (that is, manganese dioxide is only a catalyst).
>
> If you combine one mole of |H sub 2| and one mole of |Cl sub 2|, you get two moles of |H Cl|.
>
> If you combine four moles of |H Cl| with one mole of |Mn O sub 2|, you get one mole of |Mn Cl sub 2|, two moles of |H sub 2 O|, and one mole of |Cl sub 2|.

Hint: represent states by lists of lists.

(b) Write Prolog code, and try it with the depth-first program. Do the problem of making 1 mole of |Ca Cl sub 2| from a starting state consisting of 2 moles of |Cl sub 2|, 1 mole of |Mn O sub 2|, 1 mole of |Ca C O sub 3|, and 2 moles of |H sub 2 O sub 2|. That is, find a sequence of operations that could lead to a mole of |Ca Cl sub 2| being formed--not necessarily the only way it could be formed. Hint: use the **member** predicate of Section 5.5, and the **delete** predicate of Section 5.6 or the **deleteone** of Section 10.7.

(c) Describe a cost function that would be meaningful for this kind of search problem, and describe a corresponding evaluation function.

(d) The Prolog interpreter's conflict-resolution method (trying rules in the order they appear) is not good for chemical-reaction rules. Explain why, and suggest (describe) a better conflict-resolution method from ideas in Chapter 6.

10-11. Suppose you are doing a best-first search in a search space of N states.

(a) Suppose only one state in the search space satisfies the condition for a goal state. What is the smallest maximum size of the agenda during the finding of this goal state?

(b) For the same situation as part (a), what is the largest maximum size of the agenda during the finding of the goal state?

(c) Now suppose there is no goal state in the search space and we must search to confirm this. What is now the smallest maximum size of the agenda in this process of confirmation?

(d) For the situation in part (c), what is the largest maximum size of the agenda?

10-12. (A) Suppose a search space can be represented as a complete (filled-in) and balanced binary tree of 31 states on four levels (not counting the starting state as a level). Suppose there is a single goal state that does lie in the search space.

(a) Suppose we are doing a breadth-first search. What is the maximum possible size of the agenda?

(b) Suppose we are doing a best-first search. What is the maximum possible size of the agenda?

10-13. When a search problem has a solution, breadth-first search, best-first search, and A* search cannot get into an infinite loop. So it's not strictly necessary to check for previously visited states--that just improves the efficiency of search. There are also searches for which the added overhead of checking isn't worth it, and we'd like to define some criteria for recognizing such situations.

(a) Suppose for some search problem there are B successors of every state on the average, fraction P of which are previously visited (either on the agenda or the oldagenda) on the average. How many states can we expect in the search lattice up to and including level K for both visited-state pruning and no pruning at all?

(b) Suppose on the average visited-state checking requires E units of time while all the other processing required of a state (finding it and adding it to the agenda) requires C units on the average. Give an inequality for deciding when visited-state checking pays off.

(c) Now suppose that visited-state checking is not constant time but increases proportionately to the level in the search lattice; let's say the dependency is AK, K the level. What now is the inequality for deciding whether visited-state checking pays off?

10-14.(a) Consider the following program. Suppose it is only used when the first two arguments are bound (inputs) and the third argument is unbound (an output). Could a cut predicate improve efficiency? If so, under what circumstances, and where would you put it? Assume the rules are in a database in the order listed.

```
countup(I,[I|L],N) :- countup(I,L,N2), N is N2+1.
countup(I,[X|L],N) :- countup(I,L,N), not(I=X).
countup(I,[],0).
```

(b) Discuss why the cut predicate is often so tricky and hard to use.

10-15. (A,P) The Micro-Prolog dialect of Prolog has an interesting built-in feature: you can ask it to find and match the Kth rule or fact that is able to match some query. Implement this feature in the "standard" Prolog of this book, using **asserta**. Assume you are given a predicate expression P that is to be queried K times, and assume that P has no side effects so that it is OK to query the previous K-1 matches to it.

10-16. Yet another kind of iteration not yet considered occurs with the "mapcar" feature of Lisp and the vector-processing features of APL. Something similar can be provided for every two-argument function predicate in Prolog. Suppose we have a function predicate **f(X,Y)**. We would want to define a new predicate **f_of_list(L,FL)** such that **FL** is a list created by applying the **f** function to every element of the **L** list in turn. For example, suppose we had a defined predicate **double**:

```
double(X,Y) :- Y is 2 * X.
```

then querying

```
?- double_of_list([1,3,4],L).
```

will result in

```
L=[2,6,8]
```

Write a Prolog definition of **double_of_list**. Explain how the idea can be generalized to other predicates.

10-17. Implement each of the following heuristics in the way described in Section 10.11 using a six-argument **prefer** predicate. Assume they are called on by something like the best-first search program **bestsearch**.

(a) Prefer a state whose fact-list description includes an predicate named **zzxxy** of one argument.

(b) Prefer the state about which the fewest facts are true, assuming states are represented as lists of facts true in them.

(c) Prefer the state whose path list involves the fewest fact additions and deletions along its course. (Assume states are described as lists of facts true in them.)

(d) Prefer any state to one that was a successor to the last state selected.

10-18. Redo Exercise 5-12, but this time have the third argument be the *minimum* distance between two things. Allow for the possibility of cycles in the facts and more than one route between things.

10-19. (E) Suppose we defined a search problem in Prolog, and we now want to try searching backward, as perhaps when the forward search didn't work well. To make things easier, assume there is a single goal state known beforehand.

(a) Give a simple way the definition of the cost function can be modified to provide a cost function for the new backward search.

(b) Give a simple way the evaluation function can be modified to provide an evaluation function for the new problem of searching backward.

(c) Describe in general terms how the successor predicate definitions must be modified. Note that some definitions are easier to use backward than others, so explain which ones these are.

10-20. (P) The program following | REFERENCE 3| plays the game of Reversi. .FS | REFERENCE 3| This and other major programs in this book are available on tape from the publisher. .FE In this game, two players take turns placing pieces on a checkerboard; their pieces are indicated by the symbols **X** and **O**. Players place pieces so as to surround part of a row, column, or diagonal containing an unbroken string of the opponent's pieces; the pieces so surrounded are "captured" and become the opposite (surrounding player's) pieces. A player must always move so as to capture at least one opposing piece; if a player can't move, they forfeit their turn. The game is over when neither player can move; then the player with the most pieces on the board wins. Here's an example of the program output, in which first a human player makes a move, and then the computer.

```
8 X X X X X X X _
7 O O O O X O _ _
6 X O X X X O _ _
```

```
5 X O O X X O _ _
4 X O O O X O _ _
4 X O O X O O _ _
2 _ O O X X O _ _
1 _ _ O O X O _ _
  1 2 3 4 5 6 7 8
Give move:76.


8 X X X X X X X _
7 O O O O X X _ _
6 X O X X X X X _
5 X O O X X X _ _
4 X O O O X O _ _
3 X O O X O O _ _
2 _ O O X X O _ _
1 _ _ O O X O _ _
  1 2 3 4 5 6 7 8
Evaluation of move 12 is 2
Evaluation of move 74 is 193
Evaluation of move 75 is 191
Evaluation of move 86 is 144
Evaluation of move 77 is 178
Evaluation of move 87 is 266
I make move 12
```

In the program, the board is stored as an 8-item list of 8-item lists, where a 0 element means a blank board position, a 1 element means a board position occupied by the human player, and a -1 element means a board position occupied by the computer player. States are represented as 10-item lists: the player who last moved, the move they took, and the eight rows of the resulting board position.

(a) The program shown following uses a simple evaluation function. Redefine **pick_best_move** so it uses nonnumeric heuristics to select a move. That is, don't use any numbers for move selection except to look up board positions by row and column number.

(b) Now change **gamesearch** so the computer can play against itself, using your heuristics for player 1 and its evaluation function for player -1. Make sure by your heuristics that player 1 wins--improve the heuristics as necessary. Then use the heuristics for player -1, and the evaluation function for player 1, and further improve the heuristics as necessary so that player -1 wins.

(c) Now change the **pick_best_move** rules for player -1 so it looks ahead ahead one move to anticipate what the other player will do, and chooses the move leading to the best result assuming player 1 always makes the best choice available to them. Run this smarter player against the heuristics from part (b) used for player 1. Who wins?

(d) Probably player -1 won in part (c), but the comparison wasn't fair because player 1 doesn't look ahead. But they don't need to if they just have better heuristics. Demonstrate this.

```
/* Top-level routines */
go :-
  write('This program plays the game of Reversi (X is you, O is me).'),
  nl, start_state(SS), gamesearch(SS).
```

```
start_state([-1,none,Z,Z,Z,[0,0,0,1,-1,0,0,0],[0,0,0,-1,1,0,0,0],Z,Z,Z])
  :- zero_row(Z).


zero_row([0,0,0,0,0,0,0,0]).


gamesearch(State) :- goalreached(State), print_win(State), !.
gamesearch([-1,M|B]) :- not(get_move(B,1,S,NB)),
  write('You cannot move so I get another turn.'), nl, !,
  gamesearch([1,M|B]).
gamesearch([-1,OM|B]) :- print_board(B), write('Give move:'),
  repeatread(Movecode), move_decode(Movecode,Move),
  get_move(B,1,Move,NB), !, gamesearch([1,Move|NB]).
gamesearch([1,M|B]) :-  print_board(B),
  pick_best_move([1,M|B],Newstate),
  write_move(Newstate), !, gamesearch(Newstate).
gamesearch([1,M|B]) :- not(get_move(B,-1,S,NB)),
  not(get_move(B,1,S,NB)), print_win([1,M|B]), !.
gamesearch([1,M|B]) :- not(get_move(B,-1,S,NB)),
  write('I cannot move so you get another turn.'),
  nl, !, gamesearch([-1,M|B]).
gamesearch(S) :- print_win(S), !.

/* Intermediate routines */
pick_best_move(S,NS) :- asserta(ebest(none,1000000)), successor(S,S2),
  eval(S2,E2), write_eval(S2,E2), ebest(S1,E1), E2<E1,
  retract(ebest(S1,E1)), asserta(ebest(S2,E2)), fail.
pick_best_move(S,NS) :- ebest(NS,E), retract(ebest(NS,E)).


goalreached([X,Y|B]) :- not(somezero(B)).


somezero(B) :- member(R,B), member(0,R).


successor([OP,Lastmove|Oldboard],[P,M|Newboard]) :-
  opposite(OP,P), get_move(Oldboard,P,M,Newboard).


opposite(1,-1).
opposite(-1,1).


repeatread(M) :- read(M).
repeatread(M) :- write('Invalid move.  Try again:'),
  nl, repeatread(M).


move_decode(M,[X,Y]) :- X is M//10, Y is M mod 10.

/* Printing routines */
write_move([P,[X,Y]|B]) :- Move is (10*X)+Y, write('I make move '),
  write(Move), nl, !.
write_eval([P,[X,Y]|B],E) :- write('Evaluation of move '),
  Movecode is (10*X)+Y, write(Movecode), write(' is '),
  write(E), nl, !.
```

```
print_win([P,M|B]) :- print_board(B),
  countup(1,B,N1), countup(-1,B,N2), print_win2(N1,N2).
print_win2(N1,N2) :- N2<N1, write('Congratulations! You win.'), nl.
print_win2(N1,N2) :- N2>N1, write('Sorry--I win.'), nl.
print_win2(N,N) :- write('The game is a draw.'), nl.


countup(I,[],0) :- !.
countup(I,[R|B],N) :- countrow(I,R,N2), countup(I,B,N3), N is N2+N3.


countrow(I,[],0) :- !.
countrow(I,[I|L],N) :- countrow(I,L,N2), N is N2+1, !.
countrow(I,[J|L],N) :- countrow(I,L,N).


print_board(B) :- reverse(B,RB), print_board2(RB,8), !.
print_board2([],N) :- write('  1 2 3 4 5 6 7 8'), nl.
print_board2([Row|Board],N) :- write(N), print_row(Row), N2 is N-1,
  print_board2(Board,N2).
print_row([]) :- nl.
print_row([I|Row]) :- decode(I,DI), write(' '), write(DI),
  print_row(Row).


decode(0,'_').
decode(1,'X').
decode(-1,'O').

/* Move generation */
get_move(B,P,[X,Y],NB) :- get_move2(B,[X,Y],FR,BR,FI,BI),
  fix_key_row_half(P,FI,NFI), fix_key_row_half(P,BI,NBI),
  fix_rows(FR,X,P,NFR), fix_rows(BR,X,P,NBR),
  not(nochange(FI,NFI,BI,NBI,FR,NFR,BR,NBR)),
  reverse(NFI,RFI), reverse(NFR,RFR),
  append(RFI,[P|NBI],Row), append(RFR,[Row|NBR],NB).


nochange(X,X,Y,Y,A,B,C,D) :- samefront(A,B), samefront(C,D).


samefront([],[]) :- !.
samefront([X|L],[X|M]).


get_move2(B,M,FR,BR,FI,BI) :- get_move3(B,1,[],M,FR,BR,FI,BI).


get_move3([R|B],Y,FR,[X,Y],FR,B,FI,BI) :- get_move4(R,1,[],X,FI,BI).
get_move3([R|B],Y,FR,M,FRF,BR,FI,BI) :-  Y2 is Y+1,
  get_move3(B,Y2,[R|FR],M,FRF,BR,FI,BI).


get_move4([0|R],X,FI,X,FI,R).
get_move4([I|R],X,FI,XF,FIF,BI) :- X2 is X+1,
  get_move4(R,X2,[I|FI],XF,FIF,BI).
```

```
/* Board update for a particular move */
fix_key_row_half(P,R,NR) :- fix_key_row_half2(P,R,NR), !.
fix_key_row_half(P,R,R) :- !.


fix_key_row_half2(P,[],[]) :- !, fail.
fix_key_row_half2(P,[0|L],[0|L]) :- !, fail.
fix_key_row_half2(P,[P|L],[P|L]) :- !.
fix_key_row_half2(P,[OP|L],[P|L2]) :- opposite(P,OP),
  fix_key_row_half2(P,L,L2).


fix_rows([],X,P,[]) :- !.
fix_rows([R],X,P,[R]) :- !.
fix_rows(B,X,P,NB) :- Xm1 is X-1, Xp1 is X+1, opposite(P,OP),
  fix_rows2(B,OP,NB,Xm1,X,Xp1,VL,VM,VR), !.


fix_rows2(B,OP,B,dead,dead,dead,VL,VM,VR).
fix_rows2([],OP,[],XL,XM,XR,VL,VM,VR) :- topfix(OP,VL),
  topfix(OP,VM), topfix(OP,VR).
fix_rows2([R|B],OP,[NR|NB],XL,XM,XR,VL,VM,VR) :-
  fix_rowL(R,OP,1,NR,XL,XM,XR,VL,VM,VR,NXL,NXM,NXR),
  fix_rows2(B,OP,NB,NXL,NXM,NXR,VL,VM,VR).


topfix(OP,OP) :- !.
topfix(OP,V).


fix_rowL(R,OP,X,NR,dead,dead,XR,VL,VM,VR,dead,dead,NXR) :-
  fix_rowR(R,OP,X,NR,XR,VR,NXR).
fix_rowL(R,OP,X,NR,dead,XM,XR,VL,VM,VR,dead,NXM,NXR) :-
  fix_rowM(R,OP,X,NR,XM,XR,VM,VR,NXM,NXR).
fix_rowL(R,OP,X,NR,0,XM,XR,OP,VM,VR,dead,NXM,NXR) :-
  fix_rowM(R,OP,X,NR,XM,XR,VM,VR,NXM,NXR).
fix_rowL([0|R],OP,X,[0|NR],X,XM,XR,OP,VM,VR,dead,NXM,NXR)
  :- Xp1 is X+1, fix_rowM(R,OP,Xp1,NR,XM,XR,VM,VR,NXM,NXR).
fix_rowL([OP|R],OP,X,[VL|NR],X,XM,XR,VL,VM,VR,Xm1,NXM,NXR) :-
  Xp1 is X+1, Xm1 is X-1,
  fix_rowM(R,OP,Xp1,NR,XM,XR,VM,VR,NXM,NXR).
fix_rowL([P|R],OP,X,[P|NR],X,XM,XR,P,VM,VR,dead,NXM,NXR) :-
  Xp1 is X+1, opposite(OP,P),
  fix_rowM(R,OP,Xp1,NR,XM,XR,VM,VR,NXM,NXR).
fix_rowL([I|R],OP,X,[I|NR],XL,XM,XR,VL,VM,VR,NXL,NXM,NXR) :-
  Xp1 is X+1,
  fix_rowL(R,OP,Xp1,NR,XL,XM,XR,VL,VM,VR,NXL,NXM,NXR).


fix_rowM(R,OP,X,NR,dead,XR,VM,VR,dead,NXR) :-
  fix_rowR(R,OP,X,NR,XR,VR,NXR).
fix_rowM([0|R],OP,X,[0|NR],X,XR,OP,VR,dead,NXR) :-  Xp1 is X+1,
  fix_rowR(R,OP,Xp1,NR,XR,VR,NXR).
fix_rowM([OP|R],OP,X,[VM|NR],X,XR,VM,VR,X,NXR) :-  Xp1 is X+1,
  fix_rowR(R,OP,Xp1,NR,XR,VR,NXR).
fix_rowM([P|R],OP,X,[P|NR],X,XR,P,VR,dead,NXR) :-  Xp1 is X+1,
  opposite(OP,P), fix_rowR(R,OP,Xp1,NR,XR,VR,NXR).
```

```
fix_rowM([I|R],OP,X,[I|NR],XM,XR,VM,VR,NVM,NXR) :-  Xp1 is X+1,
  fix_rowM(R,OP,Xp1,NR,XM,XR,VM,VR,NVM,NXR).


fix_rowR(R,OP,X,R,dead,VR,dead).
fix_rowR(R,OP,X,R,9,OP,dead).
fix_rowR([0|R],OP,X,[0|R],X,OP,dead).
fix_rowR([OP|R],OP,X,[VR|R],X,VR,Xp1) :- Xp1 is X+1.
fix_rowR([P|R],OP,X,[P|R],X,P,dead) :- opposite(OP,P).
fix_rowR([I|R],OP,X,[I|NR],XR,VR,NXR) :- Xp1 is X+1,
  fix_rowR(R,OP,Xp1,NR,XR,VR,NXR).

/* Evaluation function */
eval([P,[X,Y],A,B,C,D,E,F,G,H],Ev) :- eval2s(A,B,Ev1),
  eval2s(H,G,Ev2), eval2m(C,Ev3), eval2m(D,Ev4),
  eval2m(E,Ev5), eval2m(F,Ev6),
  Ev is -P*(Ev1+Ev2+Ev3+Ev4+Ev5+Ev6), !.


eval2m([A,B,C,D,E,F,G,H],Ev) :-
  Ev2 is ((30*A)+(-15*B)+C+D+E+F+(-15*G)+(30*H)),
  sc(A,B,C1), sc(H,G,C2), Ev is Ev2+C1+C2, !.


sc(0,B,0).
sc(A,0,0).
sc(1,1,30).
sc(1,-1,20).
sc(-1,-1,-30).
sc(-1,1,-20).


eval2s([A,B,C,D,E,F,G,H],[I,J,K,L,M,N,O,P],Ev) :-
  Ev2 is
   (200*A)+(-50*B)+(30*C)+(30*D)+(30*E)+(30*F)+(-50*G)+(200*H),
  Ev3 is
   -((50*I)+(100*J)+(15*K)+(15*L)+(15*M)+(15*N)+(100*O)+(50*P)),
  sc2(A,I,J,B,Ev4), sc2(H,G,O,P,Ev5), sc(C,K,Ev6), sc(D,L,Ev7),
  sc(E,M,Ev8), sc(F,N,Ev9),
  Ev is Ev2+Ev3+Ev4+Ev5+Ev6+Ev7+Ev8+Ev9, !.


sc2(0,B,C,D,0).
sc2(A,B,C,D,Ev) :- sc(A,B,E1), sc(A,C,E2), sc(A,D,E3),
  Ev is 3*(E1+E2+E2+E3).

/* Utility functions */
reverse(L,RL) :- reverse2(L,[],RL), !.


reverse2([],L,L).
reverse2([X|L],L2,RL) :- reverse2(L,[X|L2],RL).


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

```
member(X,[X|L]).
member(X,[Y|L]) :- member(X,L).
```

Go to book index