1988

Artificial Intelligence through Prolog by Neil C. Rowe

Rowe, Neil C.

Prentice-Hall

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

Search

It's time for a change of pace. In this and the next two chapters, we'll cover a new class of artificial intelligence techniques. These are good for reasoning about events in time and plans of action. Technically, "search" | REFERENCE 1|. .FS | REFERENCE 1| Also called *heuristic search*, but that's misleading because you don't need heuristics to do it, and anyway the term "heuristic" is overused and much abused in artificial intelligence. .FE We'll need a new vocabulary to talk about search problems--terms like "state", "operator", "search strategy", "evaluation function", and "branching factor".

Search problems appear in many application areas. Search relates to rule-based systems because they involve actions occurring over time. In fact, search is an abstraction of the concept of a control structure. So we'll be looking at rule-based systems in a new way.

Changing worlds

So far the facts in Prolog databases have rarely changed. We added facts through caching in forward chaining, but those facts were logically derivable from the other facts and rules all along. We rarely removed facts from a Prolog database--only in implementing a few special programs.

But Prolog databases usually model a situation in the world, and the world can change. In fact, the result of running an artificial intelligence program may be to recommend actions in the world, which are then taken, and then reflected in changes to the facts true about the world. Often an artificial intelligence program is useless unless it does change something in the world. For instance, an automobile diagnosis program is not much help unless you can actually fix a car by its recommendations.

In many important applications, a problem can be solved only by a series of actions, not just one. Then the problem must track the changing situation in the world. Changing the facts appropriately can be complicated for such problems, and we need some new tricks.

States

A situation in the world described by some facts is a *state*. State is a fundamental concept in many areas of science and engineering. For instance, in physics a state is a set of measurable physical parameters fully describing some part of the universe; in simulation models of economics, a state is the set of values assigned to model parameters; and in computer systems programming, a processing state is a set of values currently in registers, buffers, and stacks. So a state is a "snapshot" of a process in time.

A state can be instantaneous or it can last for quite a while. In artificial intelligence (and computer science too) we only concern ourselves with non-instantaneous states, but where state changes (*branches* or *transitions*) are instantaneous. That is, state changes are *discrete* and not *continuous*. Loading a computer register is an example of a discrete state change. *Search* is the study of states and their transitions.

A state must include everything important about a situation in one bundle | REFERENCE 2|. .FS | REFERENCE 2| At least the states in this book. To make things easier, we deliberately ignore a class of searches performed using and-or-not lattices, for which states represent pieces of a problem. See other books

under the topic of "searching and-or trees". .FE That means everything necessary to reason about it--and in artificial intelligence, that usually means a list of facts. These may be a lot of facts, and often we'll want to reason about many states, as when we try to figure how to achieve some desirable state. So two tricks often simplify state descriptions. First, we can keep only the facts relevant to the very specific problem we're working on. For instance, if our problem is to fix a car when we know there is something wrong with the electrical system, we can ignore facts about the state of the engine (whether it is cold or warm, whether the timing is accurate, whether the oil has been changed recently, etc.) since they don't matter. Second, we can compress the fact representation by collapsing several predicates into one, like the "relational database" predicates of Section 2.9. For instance, if ten screws hold the battery onto the chassis of a car, we can write one fact with ten arguments where argument number K is the word "on" or the word "off" reflecting the status of screw number K.

Three examples

Let's consider some examples of search problems. First, consider the problem of getting by car from one intersection to another in a city. There are different streets that you can take and different turns you can make. To complicate matters, some streets are one-way, and some turns are illegal. Also, you will probably want a short route rather than just any route. Considering this a search problem, a state s ithe position of a car in a city. We can represent this as the name of the intersection, since intersections are the only places where decisions must be made, and therefore the only locations that matter to reasoning. Solving a search problem means finding a series of states ending in some desired state, so the solution will be a sequence of intersections a car will traverse in order.

As a second example of search, consider the problem of car repair. Even if you know what's wrong, repair is rarely as simple as yanking the old part out and sticking the new part in. Usually you must unloosen fasteners (like screws and nuts) first, and move other parts out of the way to get to the part you want. You may need to run some tests on the suspect part to see if it is indeed faulty. Then you must reassemble the whole mess, doing the reverse of the previous steps in the reverse order--and sometimes exact reversal is impossible, as when removing a cover is easy but refastening requires careful hole alignment. You must also worry about the difficulty of various actions, if you want to get the job done efficiently; a sequence of actions that take half as much effort as another is better. For this problem, states are situations of partial disassembly (or assembly) of the car.

As a third example, consider the problem of smarter forward chaining for a rule-based expert system. Section 6.2 presented a simple general-purpose approach to forward chaining, using simple ideas like putting facts in a priority list in the order we find them (the focus-of-attention strategy). These ideas work for a wide class of applications. But for a specific application, a specific conflict-resolution policy may be better. For instance, we can keep the facts sorted by priorities of human experts (the experts' prestored hunches of relative importance), by our own analysis of the rules (like how many new conclusions a fact can lead to), by statistics on past runs of the rule-based system, or by dynamic features of the reasoning (like whether a certain fact has been proved yet). In general, intelligent forward chaining by intelligent fact sorting is a search problem with many options, each representing a fact to pursue, each leading to a different set of facts to choose from next, and so on. So a state for this problem is a set of unpursued (and perhaps also pursued) facts.

We'll use these examples subsequently in the chapter. They were picked to symbolize three important applications of search: optimization problems, human and robot task planning, and expert systems.

Operators

Our three examples involve many states. They can involve even more branches, ways directly from one state to another. It helps to group branches into categories called *operators*. Operators often correspond to verbs of English, the names of real-world actions.

Consider the three examples:

1. For finding routes in a city, the operators are possible actions at an intersection, since states correspond to intersections. So the operators are "go straight ahead", "turn right", "turn left", "turn into the first street to your right", "turn into the second street to your right", etc.

2. For car repair, the operators are much more numerous, including every possible action that can be done on the car: unfasten, unscrew, remove objects, manipulate objects, replace, align, fasten, screw, and so on.

3. For smarter forward chaining for a rule-based system, there seems to be only one operator: remove a fact from the set of unexamined facts and pursue its implications.

Be careful though, because operators are not well-defined for many problems. We could, for instance, consider each screwing-in for every single screw as a separate operator; or we could consider each pursuit of a fact as a separate operator. But if operators get too specific, they lose their chief advantage of abstraction over many state transitions.

Preconditions and *postconditions* are frequently associated with operators. Preconditions for an operator are things that must be true before the operator can be applied, writable as predicate expressions. For instance, a precondition to the operator of removing the battery from a car is that all the screws fastening the battery to the chassis are removed. Analogously, postconditions are conditions that must be true after an operator has been applied. For instance, a postcondition of removing the battery from a car is that the car can't start when the ignition switch is turned. Preconditions are useful because they suggest what is necessary to apply an operator; postconditions are useful because they summarize the effect of an operator.

Search as graph traversal

The main reason for identifying states and operators is to let us plan in advance a solution to a search problem. By reasoning hypothetically about operator sequences, we can avoid blundering about in the real world. Planning can also ensure that we find the *best* solution to a problem, not just any solution.

For such advance planning, we can consider a search problem as a task of traversing a directed-graph data structure. Semantic networks (see Section 2.7) are one kind of directed graph important in artificial intelligence, but there are several others too. Graphs consist of nodes (small named circles) and directed edges (named arrows) connecting nodes; we can equate nodes with states, the names on directed edges with operator names, and edges themselves with branches or operator applications. In a search problem, we are given a *starting* state and one or more finishing or *goal* states. On the search graph this means finding a path or traversal between a start node and one of a set of goal nodes. (Here we broaden the term "goal" beyond its meaning for rule-based systems, in which a "goal" is a predicate expression we want to prove; in search, a "goal" is any desired final state.)

Figure 9-1 shows an example route-planning problem. As we said, city-route planning is a search problem for which the states are intersections (intersections are the only places where we must make choices). Then a search graph can be drawn as in Figure 9-2. There is just one goal state.

Figures 9-3 and 9-4 show similar example search graphs for the auto repair and the smarter forward chaining problems. The latter is interesting because we don't know the goal state in advance: we just keep searching until there are no more facts to pursue, and that's a goal state. If we knew what a goal state was in advance, we wouldn't need to do any work. An important class of search problems similarly defines goal states indirectly, in that they only can recognize one when they see one.

The simplest search strategies: depth-first and breadth-first

There are many control structures for search; these *search strategies* are summarized in Figure 9-5. The two simplest are depth-first search and breadth-first search.

With depth-first search, the start state is chosen (*visited*) to begin, then some *successor* (a state that we can reach by a single branch or state transition) of the start state, then some successor of *that* state, then some successor of that, and so on until we reach a goal state. Usually the choice among successors of a state isn't arbitrary, but made by "heuristics", which we'll explain in Section 9.7. If depth-first search reaches a state S without successors, or if all the successors of a state S have been chosen (visited) and a goal state has not yet been found, then it "backs up". That means it goes to the immediately previous state or *predecessor*--formally, the state P whose successor was S originally. Depth-first search then takes the next-suggested successor choice of P. So "backing up" is just like Prolog backtracking.

An example will make this clearer (see Figure 9-6). The circled letters are states, and as before, the arrows are branches (operator applications). Suppose S is the start state and G is the only goal state. Suppose that vertical height of the state on the page is used to rank states when a choice is necessary, with higher states having higher rank. Depth-first search will first visit S, then A, then D. But D has no successors, so we must back up to A and try its second successor, E. But this doesn't have any successors either, so we back up to A again. But now we've tried all the successors of A and haven't found the goal state G, so we must back up to S.

Now S has a second successor, B. But B has no successors, so we back up to S again and choose its third successor, C. C has one successor, F. The first successor of F is H, and the first of H is J. J doesn't have any successors, so we back up to H and try its second successor. And that's G, the only goal state. So we're done. The solution path to the goal is S, C, F, H, and G.

One problem with depth-first search is that it works fine when search graphs are trees or lattices, but can get stuck in an infinite loop on other graphs (see Appendix C for definitions of these terms). This is because depth-first search can travel around a cycle in the graph forever. One fix is to keep a list of states previously visited, and never permit search to return to any of them. We will do this in our depth-first program in the next chapter, but this check can require a lot of time since there may be lots of previous states, and states with long, complicated descriptions.

Breadth-first search does not have this danger of infinite loops. The idea is to consider states in order of increasing number of branches (*level*) from the start state. So we first check all the immediate successors of the start state, then all the immediate successors of these, then all the immediate successors of those, and so on until we find a goal state. For each level, we order states in some way as with depth-first search. For

Figure 9-6, S is on level 0; A, B, and C are on level 1; D, E, and F, level 2; H and I, level 3; and J, G, and K, level 4. So breadth-first search, assuming the previously-used vertical ordering among same-level states, will consider in order S, A, B, C, D, E, F, H, I, J, and G--and then stop because it's reached the goal state. But the solution path it found was the same as depth-first's.

Breadth-first search is guaranteed to find a goal state if a path to one exists, unlike depth-first, but it may take a while. Loops in the search graph will cause inefficiency (useless extra paths being considered) instead of infinite processing loops. If that inefficiency is bothersome, we can use the same trick in depth-first of storing a list of previously visited states and checking against it. Notice that by working by level, any later path we find to a state can be no shorter than the first path found to it, so the first path found to a state can be considered the best.

Depth-first and breadth-first search often occur in disguise in artificial intelligence and computer science. The backward chaining, forward chaining, and rule-cycle hybrid chaining algorithms of Chapter 6 were really kinds of depth-first search, so they can be called *depth-first control structures*. Depth-first control structures are common in computer applications because of their intimate connection to stacks, an easy-to-implement data structure. Though we didn't discuss it in Chapter 6, we could do breadth-first backward chaining as a variant control structure, or breadth-first forward chaining. So be careful not to confuse the backward/forward chaining distinction with the depth-first/breadth-first search distinction--they're quite different things.

Heuristics

Depth-first and breadth-first search are easy to implement but often inefficient for hard problems. A top concern of artificial intelligence research has been finding better search strategies. Many of these better strategies are related to depth-first and breadth-first search. Two common variants are search using heuristics and search using evaluation functions.

Heuristics are any nonnumeric advice about what order to try the successors of a state for further search | REFERENCE 3|. .FS | REFERENCE 3| Some authors say heuristics can be based on numeric calculations, but we'll call such things *evaluation functions* to prevent confusion. .FE So their effects are "local": they give advice for specific successor choices, not about the whole search strategy. Since both depth-first and breadth-first must make such choices, both can use heuristics, as well as the other search strategies we'll discuss in this chapter. Heuristics are like gardeners because they *prune* (eliminate) branches. Heuristics are a generalization of meta-rules (see Section 6.7), rules about rules.

Usually heuristics are not guaranteed--that is, they represent reasonable advice about how to proceed, and may be wrong. But if they're wrong a lot, there's no point in using them. Heuristics need not give a unique recommendation of what's best in every situation. At worst, we can choose at random among multiple recommendations. What if that's not a good idea? "Search" me.

Here are example heuristics:

> --For city route planning, never turn right twice in a row, since this tends to make you go back in the direction you came from.
>
> --For city route planning, turn whenever you find you've left city limits.
>
> --For car repair, never remove a part unless it is near another part you think is faulty.

--For car repair, take out small parts first.

--For smarter forward chaining, pursue facts that occur together with other known facts on the right side of a rule.

--For smarter forward chaining, pursue facts that led to useful conclusions on the ten most recent runs.

Heuristics are not all equally valuable. Compare:

1. For car repair, do little jobs first.

2. For car repair, first take out small parts attached to objects you want to fix.

3. For car repair, first take out screws attached to objects you want to fix.

4. For car repair, if you must fix the alternator, take out its mounting screws first.

5. For car repair, if you must fix the alternator, take out its Right Front Mounting Screw first.

These heuristics span a spectrum from general to specific. Really general advice like #1 is hard to apply (it's hard to decide what's a "little job" and what isn't) and wrong in many cases (it would recommend equally the removal of any screw in the car, most of which are irrelevant). Such heuristics are "proverbs", like "Haste makes waste"; they sound nice, but they're nearly useless. At the other extreme, heuristic #4 and #5 is too specific to be helpful: it just applies to one action involving one screw in a car, and its effect could be had by just putting conditions on the "fix-alternator" operator. And if we tried to use heuristics like #5 in a search problem, we'd need a lot of them--probably many more than the number of operators--so search efficiency would probably decrease, not increase as is the purpose of heuristics. Our best bets are heuristics like #2 and #3 that compromise between extremes; probably #3 is better because "small part" is hard to define.

Evaluation functions

A big difficulty with heuristics is disagreements between them, situations for which two heuristics make contradictory recommendations. If one heuristic is known to be better than the other, then it should have priority, but the two can seem equally good. If such difficulties arise frequently, it's better to rate states numerically. A method for calculating such numbers is called an evaluation function. By convention the values of evaluation functions are nonnegative numbers such that the smaller the number, the better the associated state; and goal states have an evaluation function value of zero. Evaluation functions can be in any form and can use any available information about a state; they can also use a description of the goal states. (However, it's desirable that evaluation functions be "smooth"; that is, if they're calculated using numbers, they shouldn't ever jump abruptly in value when the numbers vary slightly.)

Some example evaluation functions:

--For city route planning, take the straight-line ("as the crow flies") distance between an intersection and the goal intersection (that is, prefer the successor state closest to the goal state along a straight line).

--For city route planning, take the straight-line distance to the goal plus one tenth of the number of streets crossed by that straight line (this helps you avoid stop signs and traffic lights).

--For car repair, take the number of parts removed from the car plus the number of faulty parts in the car (this will increase in the first half of the solution, but will decrease in the last half, and should be kept small anyway).

--For smarter forward chaining, take the number of right-side expressions in rules minus the number of facts (proved and given) in some state. (This isn't likely to approach zero, but it does guide search helpfully.)

Evaluation functions make possible two new search strategies, summarized on the third and fourth rows in Figure 9-5. The evaluation-function variant of depth-first search is called *hill-climbing* (or sometimes *discrete optimization*) search; the evaluation-function variant of breadth-first search is called *best-first* search. However, best-first usually has an additional twist beyond breadth-first: the best-evaluation (lowest-evaluation) state of those *anywhere* in the search graph, of those whose successors have not yet been found, is picked, not just a state at the same level as the last state. So best-first search usually "jumps around" a lot in the search graph, always picking the minimum-evaluation state of all the unvisited states it knows about.

To illustrate best-first search, take the search graph of Figure 9-6 and assume the evaluation function values shown as circled numbers in Figure 9-7: S:12, A:7, B:8, C:8, D:6, E:4, F:7, H:4, I:5, J:2, G:0, and K:1. (Ignore the numbers in *squares* and the numbers beside the arrows for now.) As before, assume S is the starting state and G is the only goal state. Best-first search would start with S, and would find and evaluate its successors A, B, and C, to discover that A is the minimum-evaluation one. So A is picked next, and its successors D and E evaluated. The states not yet examined to find successors (*visited*) are D, E, B, and C, with evaluation function values 6, 4, 8, and 8 respectively. E is the best, but it has no successors; D is the second best, but it has no successors.

We've got a tie between the two remaining unexamined states, B and C. In such cases, heuristics can decide (and perhaps for near-ties too). Assuming the vertical ordering heuristic used with the depth-first and breadth-first examples in Section 9.6, B should be picked next. But it has no successors, so C must be picked. It has one successor, F. F has two successors, H and I, with evaluation function values 4 and 5. The 4 is better, so H is better, and is picked next. H has successors J, G, and K with evaluation function values 2, 0, and 1. But G is a goal state, so we can stop. In summary, best-first search examined the states in the order S, A, E, D, B, C, F, H, G. That's different from both depth-first and breadth-first.

Cost functions

Evaluation function values are not the only numbers helpful in search. For some problems, we don't want just *any* path to a goal state, but a good or best path. We could assign costs to each operator, total up these costs along solution paths, and rate the paths. This requires a nonnegative *cost function* measuring something like the difficulty by the sums of going from one state to another.

Some example cost functions:

--For city route planning, the total length in meters of a path;

--For city route planning, the number of intersections in a path (since intersections slow you

down);

--For car repair, the time in minutes needed to do a sequence of actions;

--For car repair, the amount of energy in calories needed to do a sequence of actions;

--For smarter forward chaining, the computer time in seconds required to pursue facts in a particular order;

--For smarter forward chaining, the amount of main-memory storage in bytes required to pursue facts in a particular order.

Cost functions and evaluation functions are easy to confuse. Remember that evaluation functions refer to the future, cost functions to the past. That is, evaluation functions guess how close a state is to a goal state, while cost functions measure how far a state is from the start state. So cost functions are more concrete than evaluation functions. A cost function often suggests an associated evaluation function, not the other way around, because (as we will see) it is useful to have them in the same units. So if a cost function measures in meters, its evaluation function should too.

Optimal-path search

A search that must find the lowest-cost (*optimal*) path to a goal state, instead of just any path, needs a different search strategy from those so far considered. If we have a cost function but no good evaluation function and no good heuristics, we can use *branch-and-bound* search (see Figure 9-5). It's like best-first search but using costs instead: it always finds successors of the state whose path has lowest total cost from the start state. Such a strategy may "jump around" among states as best-first search does, but it has a nice property: the first path to the goal that it finds is guaranteed to be the lowest-cost path to the goal.

If we have both cost and evaluation functions, we can use an *A* search* strategy (that's pronounced "A-star"). The idea is to sum the cost and evaluation function value for a state to get a measure of overall worth, and use these numbers to select states in a best-first search algorithm instead of just the evaluation function values. (This sum makes most sense when the cost function and evaluation function are in the same units.) So A* search is sort of a hybrid of best-first (using an evaluation function) and branch-and-bound search (using a cost function), incorporating information from both, and often giving better performance than both. As with branch-and-bound search, a certain guarantee applies to a solution found by A* search: if the evaluation function value for any state S is always no more than the subsequently found cost from S to the goal, then the first path to the goal found by A* search is the lowest-cost path to a goal. But A* is often still a good search strategy even when the guarantee doesn't hold.

Suppose we use the A* strategy instead of best-first on the search graph of Figure 9-7. Recall that the numbers in circles are the evaluation function values of states. Suppose the costs for path segments are shown by the numbers next to the arrows: 4 for A to S, 1 for A to D, 2 for A to 3, 1 for S to B, 2 for S to C, 1 for C to F, 3 for F to H, 1 for F to I, 1 for H to J, 2 for H to G, and 2 for H to K. The criterion number for a state is now the sum of the evaluation function value and the costs along the path leading to that state. For instance, the criterion number for state H is |4 + 3 + 1 + 2 = 10|. The other criterion values are: S:12, A:11, D:11, E:10, B:9, C:10, F:10, I:9, J:9, G:8, and K:9; these numbers appear inside squares in Figure 9-7. A* search will work like best-first but use the numbers in the squares instead of the numbers in the circles, and it will visit the states in the order S, B, C, F, I, H, and G. (The evaluation function values are not a lower bound on the

cost to the goal, but that doesn't matter when there's only one path to the goal.)

A route-finding example

To better illustrate the differences between search programs, we show results from a program for one of our three standard examples, finding city routes. Figure 9-8 shows a portion Monterey, California, USA for which we stored street-intersection coordinates. The problem was to go from the the point marked "start" to point marked "goal". The evaluation function was the straight-line distance to the goal (computed from approximate coordinates of each intersection), and the cost function was the distance along the route (computed by summing straight-line distances between successive intersections). Shown on the map are the results of three search strategies: breadth-first, best-first, and A* search. As you can see, the paths are different: A* finds the shortest path, breadth-first minimizes the number of intersections, and best-first tries to keep moving towards the goal. (Though not shown on the map, depth-first search wanders around stupidly.)

Special cases of search

Certain tricks sometimes make search problems easier. One trick often possible is reversing the search, working backward from goal states to start states. For instance:

> --for city route planning, find a path from the destination to the start;
>
> --for car repair, solve the first half of the job by reasoning from the part P that must be fixed, deciding what other part P2 needs to be removed to get to P, what other part P3 must be removed to get to P2, and so on.

This does require you to know beforehand all goal states, something not possible for the smarter forward chaining problem and other important problems, in which finding a goal state is the whole point of the search (though backward chaining does represent something like a reverse of forward chaining). If you know more than one goal state, you can either try reverse search with each in turn, or make all the goal states the starting set of "unexamined" states for those strategies that use them.

Backward search generally requires different *reverse* operators from the forward search operators. For instance in city route planning, the reverse of a right-turn operator is a "backward left-turn" operator. So the solution to the backward search can be different from the solution to the forward search even if the same heuristics, evaluation function, and/or cost function are used. Fortunately in many search problems, the backward operators *are* identical to the forward operators, as in route planning on a larger scale, planning of routes between cities.

Whenever backward search is good, an even better idea is parallel forward and backward search on the same problem--*bidirectional search*. Run both searches independently (either on two processors or time-sharing on a single processor), and stop whenever they meet (or reach the same state). As illustration, compare the top and middle diagrams in Figure 9-9. Bidirectional search often finds a solution much faster than either forward or backward search alone, because the number of search states usually increases quickly as we move farther from the starting point, and with two searches we're searching half as deep. But it does have a serious danger: if we use a poor search strategy for both searches, like best-first with a poor evaluation function or, often, or depth-first search, the two searches may "bypass" one another instead of meeting in the middle.

Another trick we can often use to make search easier is *decomposition*, breaking the search problem into several simpler sub-searches, each of which can be solved independently. This usually means finding an intermediate state through which the search must go | REFERENCE 4|. .FS | REFERENCE 4| A starting state usually has several branches from it. But each branch isn't considered a "decomposition" of the problem--the whole search problem isn't made any simpler by looking at it this way, as it usually is by decomposition around intermediate states. .FE Then we solve two separate search problems: getting from the start to the intermediate state, and getting from the intermediate state to a goal state. As illustration, compare the top and bottom diagrams in Figure 9-9. You can also decompose a problem into more than two pieces, if you can figure out more than one intermediate state that the search must go through.

For city route planning, if the goal is a place on the other side of a river, and only one bridge crosses the river, you must use that bridge. So you can simplify the problem by decomposition into two subproblems: getting from your start to the bridge, and getting from the bridge to the goal. For car repair, a good decomposition is into three pieces: getting the faulty part out, fixing the faulty part, and then reassembling the car. Decomposability of search depends on the problem, but whenever it's possible it's usually a good idea because shallow search problems can be a lot easier than deep search problems.

Another useful feature of a search problem is *monotonicity*. If whenever an operator can be applied from a state S, that same operator can be applied from any state reachable from S (by a sequence of successors), then the search problem is monotonic. Often we don't need to be so careful in choosing an operator in a monotonic search problem, because we can can get much the same effect by applying the overlooked operator later, and hence we don't need terribly good evaluation functions and heuristics. But monotonic search problems aren't common. Only the third of our three standard examples is monotonic, the smarter forward chaining problem (since a postponed pursuit of some fact F is always doable later). Many searches for proving things are monotonic, since provable things usually don't stop being provable later (what is called *monotonic logic*). One word of warning: monotonicity depends on the operator definitions, and different operators can be defined for the same search problem, so a search problem monotonic for one set of operators may not be for another.

How hard is a search problem?

It helps to know the difficulty of a search problem before tackling it so you can allocate time and space resources. Sometimes what seems an easy problem can be enormously difficult, or a problem very similar to a really difficult one can be easy. So people try to estimate the number of states that need to be studied to solve a search problem. Two methods can do this: bounding the size of the search space, and calculating the average *branching factor* or *fanout* of successive states.

The first method takes an upper bound on the number of states that need to be examined to solve a problem as the number of possible states in the entire problem. This set of all possible states is called the *search space*, and what's needed is the size of the search space. (Don't confuse the term "search space" with the amount of memory needed to search: "space" is used abstractly here, the way mathematicians use "vector space".) For some problems, the size of the search space is easy to see from the description of the problem, as for city route planning in which it's the number of intersections in the city, something we can count. For other problems, we can use what mathematicians call *combinatorial methods*. For instance, for car repair we can describe the condition of every part in the car as either in the car and faulty, in the car and OK, out of the car and faulty, or out of the car and OK. So if there are 1000 parts in the car, the size of the search space is 4 to the 1000th power--a lot! That means heuristics or an evaluation function are necessary to solve this problem,

because we'll never succeed if we try operators at random.

The size-of-the-search-space method of estimating search difficulty can't always estimate the size of the search space so easily, as in smarter forward chaining when we can't tell in advance which or even how many facts we'll prove. Also, the method calculates only an upper bound on difficulty. Alternatively, we can reason how the number of states increases with each level of search, and figure how many levels deep we'll go. The number of successors (usually, previously unvisited successors) of a state is called its *branching factor*. (Caution: don't confuse the branching factor with the number of *operators* that can generate successors, a smaller number.) If the branching factor doesn't differ much between states, we can speak of an *average branching factor* for the whole search problem. Then we can estimate the number of states at level K in the search graph as |B sup K|, B the average branching factor. Notice that this exponential function gets large very fast as K increases--the so-called *combinatorial explosion*. So if we can estimate the level of a goal (the number of states from the starting state to it), we can estimate the number of states that will be visited by breadth-first search, summing up the estimate for the number of states at each level. This is also a good approximation for best-first search, branch-and-bound search, and A* search, when evaluation and cost functions aren't very helpful in guiding search.

As an example, the average branching factor for the city route problem is around three, since there are usually three branches (directions of travel) at an intersection: go straight, turn right, and turn left. Suppose we know we are about 10 blocks from where we want to go. Then for a breadth-first search there are approximately three states at level 1, nine states at level 2, 27 at level 3, and so on up to |3 sup 10| for level 10. So the total number of states of level 10 or less is approximately the sum of a geometric series:

```
1 + 3 + 9 + 27 + ...   =   ( 3 sup 11 - 1 ) / ( 3 - 1 ) = ( 3 sup 11 - 1 ) / 2   =   88573
```

In general, in a search problem with an average branching factor of B, the number of states up to and including those at level K is |(B sup {K+1} - 1) / (B - 1)|. If B is large (say 5 or more), this can be approximated by |B sup K|, the number of states at the deepest level. These formulas help quantify the advantages of bidirectional search and search decomposition. If you divide a problem whose solution is |R| states long into two approximately even halves, each half is about |R / 2| states long. If the branching factor B is large, and the same in both directions, the number of states examined in the two half-problems is much less than the number of states examined without decomposition or bidirectional search because:

```
B sup { R / 2 } + B sup {R / 2 }   "is much less than"   B sup R
```

because, dividing both sides by |B sup {R / 2}|:

```
2   "is much less than"   B sup {R / 2}
```

The effect of heuristics and evaluation functions is to rule out certain successors for particular states in preference to other successors. So in effect, heuristics and evaluation functions decrease the average branching factor. This means we can get farther down into the search graph faster. So the usefulness of heuristics and evaluation functions can be quantified as a ratio of average branching factors with and without them.

Backward chaining versus forward chaining (*)

Analysis of search-problem difficulty lets us quantify for the first time differences between the control structures discussed in Chapter 6. Take for instance this rule-based system:

```
t :- a, b.
t :- c.
u :- not(a), c.
u :- a, d.
v :- b, e.
```

And further assume the facts **a**, **e**, and **d** only are true, and given in that order.

Backward chaining will try the first three rules, which fail, and then the fourth, which succeeds. Basic backward chaining without caching involves eight total queries: top-level predicates **t** and **u**, **a** (which succeeds), **b** (which fails), **c** (which fails), **a** again (which succeeds), **a** again, and finally **d** (which succeeds). The time to do this is approximately proportional to the number of queries made because each query requires an index lookup of nearly-equal time.

Pure forward chaining will match fact **a** to expressions in the first and fourth rules, then match **e** to an expression in the last. Finally, it matches **d** in the fourth rule, and **u** is proved. So four matches are made. The time to do this is approximately proportional to the number of matches because each requires an index lookup of nearly-equal time. Now we just compare eight queries to four matches to decide whether backward or forward chaining is better for this very specific situation. Though this is a comparison of different units ("apples with oranges") we can usually conduct experiments to equate both queries and matches to time units.

The main problem with this analysis is that it's hard to generalize: we may not know which or how many facts are going to be true in a situation, and there are many possible situations. Simulations are one approach. But we can also use probabilities in mathematical analysis, as we'll now show.

Let's consider backward chaining first, still using the previous example. Suppose that potential facts **a**, **b**, **c**, **d**, and **e** have independent probabilities P of truth. Then the probability of the first rule **t :- a, b.** succeeding is |P sup 2|, the probability of the second succeeding is |P|, the third |(1 - P) P|, and the fourth and fifth |P sup 2|. The expected number of queries generated when the rules are in this order is a long, tricky formula because we don't need to fully evaluate a rule to have it fail:

```
3 P sup 2 + 3 ( 1 - P ) P + 4 P ( 1 - P ) P + 8 P ( 1 - P ) sup 2 P

    + 10 ( 1 - P ) P ( 1 - P ) P + 10 P ( 1 - P ) sup 3
+ 9 ( 1 - P ) sup 3 + 10 ( 1 - P ) P ( 1 - P ) sup 2
```

We got this formula by reasoning about the search lattice, which was actually more like a decision lattice, for the situations possible (see Figure 9-10). Here are the rules again:

```
t :- a, b.
t :- c.
u :- not(a), c.
u :- a, d.
v :- b, e.
```

The first term in the formula corresponds to success of the first rule, terms 2 and 3 to the success of the second, no terms to the third (there's no way the third rule can succeed given the second fails), term 4 to the success of the fourth rule, term 5 to the success of the fifth, and the remaining terms to the three different cases in which all the rules fail. To get this formula, we separately considered cases with **a** true and with **a** false.

We can reason similarly about forward chaining when facts are always taken in an order and each has the same probability P of occurrence (see Figure 9-11). Suppose the order is **a** before **b**, **b** before **c**, **c** before **d**, and **d** before **e**. The formula for the number of fact-rule matches is:

```
1 P ( 1 - P ) + 3 P sup 2 + 3 P sup 2 ( 1 - P ) + 3 P sup 2 ( 1 - P ) sup 2
+ 4 P sup 2 ( 1 - P ) sup 3 +  3 P ( 1 - P ) sup 4
+ 4 P sup 3 ( 1 - P ) sup 2

+ 4 P sup 2 ( 1 - P ) sup 3 + 3 P sup 2 ( 1 - P ) sup 3
+ 3 P ( 1 - P ) sup 4 + 3 P sup 2 ( 1 - P ) sup 3 + 2 P ( 1 - P ) sup 4
+ 2 P ( 1 - P ) sup 4 + 1 ( 1 - P ) sup 5
```

Knowing the ratio of the cost of a backward-chaining query the cost of a forward-chaining match, we can compare the two formulas for a specific value of P to decide whether backward or forward chaining is better.

Using probabilities in search (*)

Since probabilities are numbers, they can guide hill-climbing, best-first, branch-and-bound, and A* searches. For instance, the probability that a state is on a path to a goal can be used as an evaluation function. Or in search during forward chaining, facts can be ranked for selection by one million minus the reciprocal of their a priori probability, so unusual facts are followed up first. In backward chaining using rules with uncertainty like those in Chapter 8, rules can be selected by the reciprocal of their rule strength (the rule strengths being the probability of their conclusion under the most favorable conditions.)

Another example: visual edge-finding as search (*)

Computer vision is an important subarea of artificial intelligence. It's complicated, and its methods are often quite specialized, since two-dimensional and three-dimensional information seems at first to be quite different from predicate. But it also exploits many of the general-purpose techniques described in this book, including search.

One role search plays is in edge finding and edge following, as a kind of "constructive" search that actually builds something as it performs a search. Most computer vision systems start from a digitized image, a two-dimensional array representing the brightness of dots in a television picture. The picture is first "cleaned up" and "smoothed out" to make it easier to analyze, using some mathematical tricks. Any remaining sharp contrasts between adjacent parts in the picture are important--contrasts in brightness, color, and the "grain" or *texture* of small adjacent regions. These contrasts can be used to make a line drawing (a drawing consisting only of lines) of the picture, where lines correspond to boundaries of high contrast between regions of mostly-homogeneous characteristics; the lines are called *edges*. So a line drawing can be a kind of data compression of the original picture. Line drawings provide a basis for most visual analysis techniques, techniques that try to figure out what the picture is showing.

But edge-finding isn't as easy as it may seem. The problem is that for various reasons, things in the real world that you'd think would make edges don't. Consider Figure 9-12. Different surfaces may coincidentally be the same brightness and color along part of their edge, or an edge may lie in shadow, or glares and reflections may cover the edge, or the resolution of the picture may be insufficient to pick up small details of edges. So usually only some edges and parts of edges in a picture can be recognized, meaning line drawings with gaps. Human beings can easily fill in these gaps, because they have strong expectations about what they will see and their vision automatically fills in details. But computers must be taught how.

A good first step is to quantify the "edgeness" of each dot in the picture. Several mathematical formulas can be used, but we'll show here a simple one that only examines the brightness of its cell and its immediate neighbors. (Edges between regions of different color can be found by looking at brightness of the picture viewed through colored filters.) Suppose we represent the picture as a two-dimensional array |b (i , j)| of numbers representing light intensities. Then the *magnitude of the gradient* for each dot is defined as:

```
g ( i , j )   =   sqrt { ( b ( i+1 , j ) - b ( i-1 , j ) ) sup 2 +
( b ( i , j+1 ) - b ( i , j-1 ) ) sup 2 }
```

This is a measure of "edgeness" for every dot. The larger this number is, the more the brightnesses around some dot in the picture differ among themselves.

Now we're ready to formulate edge finding as a search problem. A state can be represented as a two-dimensional bit array |e (i , j) | (that is, an array of things with Boolean or true/false values) with an entry for every dot in the picture. A "true" means that in the corresponding picture, the dot lies on an edge; in the starting state, every dot is marked "false". A branch between states changes a single element from "false" to "true", meaning we've decided an edge is there. There's only one operator: mark |e (i , j)| as "true" for some i and some j. Figure 9-13 gives an example interpretation problem. The upper part shows an array of edgeness measures |g (i , j)|, and the lower part shows a reasonable assignment of edges |e (i , j)| to the corresponding cells.

However, there is no well-defined goal for this problem. It's more like an optimization problem in operations research. We want to maximize the number of "good" edges identified, so it sounds a bit like a best-first search with an evaluation function. Several things can be considered in an evaluation function:

> --the number of cells with edgeness more than some number C, of those for which |e (i , j)| is false (this measures the completeness of the edge assignments);
>
> --the total number of approximately straight line segments formed by the dots marked "true" (this measures the "elegance" of the edge assignments);
>
> --the sum of the average curvatures for all such approximately-straight line segments (this measures straightness of segments);
>
> --the number of true-marked cells with exactly two true-marked immediate neighbors (this measures narrowness of edges);
>
> --the negative of the average edgeness measure of all true-marked cells (this measures the selectivity of true-marking).

Whenever several different measures like these describe progress in a search problem, a good idea is to take a weighted average of these, and make that the evaluation function. (Or if all the numbers are positive, take the product.) That seems good here, but we would need to experiment to find the proper weightings.

Despite the lack of a goal per se, it helps to invent one to prevent working forever. A simple but effective criterion is to stop at state S if no successor state under consideration is better than D worse than the evaluation of S, for some fixed constant D.

Notice how this search problem differs from city-route planning: the edge finder will rarely mark adjacent

cells in succession. In fact, it's likely to jump around considerably, since a good heuristic is to mark the cell with the highest "edgeness" among those not yet marked "true". With that heuristic, we're unlikely to mark adjacent cells in succession. This nonlocality of processing appears in many different vision applications, and reflects the concurrency apparently present in much human vision.

.SH Keywords:

search
state
branch
operator
precondition
postcondition
starting state
goal state
search strategy
depth-first search
breadth-first search
hill-climbing search
best-first search
heuristic
pruning
evaluation function
cost function
optimal-path search
branch-and-bound search
A search*
backward search
bidirectional search
decomposability
monotonicity
monotonic logic
search space
branching factor
combinatorial explosion

Exercises

9-1. (E) Explain why none of the following are states as we define the term for search.

(a) All the important events that happen in an organization in one day.

(b) The current temperature of a room.

(c) The mental condition of a person's mind.

9-2. (A,E) Explain how using inheritance to find a value of the property of some object is a kind of search.

9-3. The search graphs in this chapter have all been *planar*--that is, none of the branches crossed. Is this necessarily true of all search graphs?

9-4. (a) Modify the description of the algorithm for pure forward chaining, given in Section 6.2, to work in a

breadth-first way.

(b) Explain how a breadth-first backward chaining might work.

9-5. Consider the problem of driving by car from one place in a city to another. Assume you do not know the way. Consider this as a search problem with three operators:

A: travel to the next intersection and keep going straight, if legal
B: travel to the next intersection and turn right, if legal
C: travel to the next intersection and turn left, if legal

(a) Suppose you are driving a car without a map. Which control strategy is better, depth-first or breadth-first? Assume you can use simple heuristics (like "turn around if you're in the country") to stay near the goal.

(b) Suppose you planned a route with a map, and the map was slightly wrong (for instance, a street is closed for repairs). How could you prepare to better handle such occurrences?

9-6. Consider the complete search graph for some problem shown in Figure 9-14. Suppose state "a" is the starting state, and the states shown are the only possible states (and none of the states shown is the goal). Numbers written next to states represent the evaluation function at those states, and numbers written next to branches represent the cost function on those branches. (Some of this information may be irrelevant.)

(a) Which is the fourth state whose successors are attempted to be found by a depth-first search using the heuristic that states whose names are vowels (a, e, i, o, or u) are preferred to states whose names are not?

(b) Which is the fourth state whose successors are attempted to be found by a best-first search?

9-7. (a) For the search graphs of Figures 9-6 and 9-7 we imposed a heuristic that states be ordered when necessary by vertical height on the page from top to bottom. Explain how this can be considered either a heuristic or an evaluation function.

(b) An evaluation function is just a way to assign nonnegative numbers to states (not necessarily a continuous function on any variable). Explain how an evaluation function can always be constructed to duplicate the effect of any set of heuristics for a search problem.

(c) Now consider the reverse direction. In a search problem, can you always go from an evaluation function for that problem to a finite set of (nonnumeric) heuristics that have the same meaning?

9-8. (E) (For people who know something about music) Explain how harmonizing a melody is a kind of search. What are the states and operators? What are some good heuristics? Why is backing up to previous states necessary?

9-9. (a) Consider A* search in which the evaluation function is zero for every state. What is the name for this search?

(b) Consider A* search in which the cost function is zero for every state. What is the name for this search?

(c) Consider A* search in which the evaluation function is zero for every state, and the cost function is the length of the path to the state. What is the name for this search, besides the name for part (a)?

9-10. (A) Suppose for some search problem for which you want to use the A* search you have found an evaluation function that never overestimates the cost to a goal state by more than K units. How can you get a guaranteed-optimal solution from A* search?

9-11. (R,A) Suppose for a search problem there are three operators, Op1, Op2, and Op3. Suppose in the starting state you can apply any of the three. Then suppose if the first operator was not Op3 you can apply a different operator for the second action than the first operator. Assume no other operator applications are possible. No goal is given, so a search must eventually explore every possible state.

(a) Suppose you do breadth-first search using the heuristic that Op1 branches are preferred to Op2 branches, and Op2 to Op3. Draw the state diagram, and label the states in the order in which you try to find their successors. Use labels a, b, c, d, e, f, g, and h.

(b) Suppose you do best-first search for which the evaluation function is

6 after Op1 then Op2
4 after Op1 then Op3
9 after Op2 then Op1
11 after Op2 then Op3
8 after Op1
7 after Op2
5 after Op3
10 for the starting state

List the states in the order in which you try to find their successors.

(c) Suppose you do A* search for which the evaluation function in part (b) and the cost function is

2 for Op1
5 for Op2
9 for Op3

List the states in the order in which you try to find their successors.

9-12. (H) Suppose we have computer terminals we wish to move to different floors of a three-floor building. Suppose at the start:

1. one terminal on the third floor belongs on the second floor;
2. one terminal on the second floor belongs on the first floor;
3. two terminals on the first floor belong on the third floor;
4. another terminal on the first floor belongs on the second floor.

In the starting state, the elevator is on the first floor. In the goal state, each terminal is on the floor where it belongs. Assume there are two operators:

A: take one terminal from floor X to floor Y, X different from Y;
B: take two terminals from floor X to floor Y, X different from Y;

Suppose the cost function is the sum over all steps in the solution of a number that is 1 for trips between

adjacent floors, and 1.2 otherwise.

(a) Give a heuristic (nonnumeric reason) for choosing branches during search.

(b) Give a lower-bound evaluation function for use with A*.

(c) Would bidirectional search be a good idea for this and similar problems? Why?

(d) Approximate the size of the search space for T terminals and F floors.

(e) Draw the state graph after the first three states have had their successors found, in the solution of the given problem using A*. Use the evaluation function you gave, but not the heuristic. Don't allow returns to previous states. Draw the evaluation plus the cost of a state next to it. If ties arise, use a heuristic to choose, and say what heuristic you're using. Hint: use a compact notation for the state, so you don't have to write a lot.

9-13. (A) Consider this problem:

> A farmer wants to get a lion, a fox, a goose, and some corn across a river. There is a boat, but the farmer can only take one passenger in addition to himself on each trip, or else both the goose and the corn, or both the fox and the corn. The corn cannot be left with the goose because the goose will eat the corn; the fox cannot be left with the goose because the fox will eat the goose; and the lion cannot be left with the fox because the lion will eat the fox. How does everything get across the river? Assume animals do not wander off when left alone.

(a) What is the search space?

(b) Give the starting and ending states.

(c) Give the operators.

(d) Draw the first two levels of the search graph. That's two besides the starting state.

(e) What is the average branching factor for these four levels? Disregard branches back to previous states.

(f) Give an upper bound on the size of the search space.

(g) Is this problem decomposable about an intermediate state?

9-14. Consider the two tasks of solving jigsaw puzzles and solving integration problems symbolically. For each, answer the following.

(a) What is the search space?

(b) What are the operators?

(c) What is the starting state?

(d) What are the final states?

(e) Is the task decomposable (breakable into subproblems that can be solved independently) about an intermediate state?

(f) Are the operators monotonic (applicable at any time, if applicable once)?

(g) Is one solution needed or the best solution?

(h) What is the initial branching factor?

(i) How (approximately) does the branching factor vary as the task proceeds?

9-15. (R,A) Consider the problem of designing electrical connections between places on the surface of a two-dimensional integrated circuit. Group the surface into square regions with a square grid. Consider this a search problem (a single search problem, not a group of search problems) in which there is only one operator: coat with metal the grid cell [X,Y] where X and Y are integers representing Cartesian coordinates. No grid cells have any metal coating in the starting state. In the goal state, electrical connections exist between each pair of a specified list of cell pairs; "electrical connection" means an unbroken sequence of adjacent coated grid cells linking a cell pair, while at the same time not linking them to any other cell pair. So for instance the goal might be to connect [10,20] to [25,29], and simultaneously [3,9] to [44,18], but not [10,20] to [3,9]. It's desired to find the coating pattern that uses the least amount of metal in achieving the goal connections.

(a) What is the search space?

(b) Is bidirectional search a good idea? Why?

(c) How does the branching factor vary as forward search proceeds?

(d) Give a heuristic for limiting forward search in this problem.

9-16. Consider the allocation of variables to registers done by an optimizing compiler for a programming language. Consider this as a search with just one operator: assign (allocate) occurrence N of variable V on line L of the program to be in register R. (Assume that variables must be in registers to use them in programs.) Since computers have a small fixed number of registers, and a poor allocation requires a lot of unnecessary instructions for moving data between registers and main memory, it's important to choose a good allocation if you want a compiler that generates the fastest possible code. For this problem, assume you do want to generate the fastest possible code. And assume this means the code with the fewest number of instructions, to make this simpler.

Notice that you can calculate speed for partial allocations of variables, not just complete allocations. That is, for all unbroken segments of code that mention register-assigned variables, you can count the number of instructions in the segments. Those numbers for those code segments can't change as new allocations of other variables are made, because one register allocation can't affect another.

(a) Which is the best search strategy?

(i) depth-first search
(ii) breadth-first search
(iii) best-first search
(iv) A* search

(b) Which is a heuristic for this problem? (only one answer is correct)

(i) "Count the number of machine instructions generated by a solution."
(ii) "Allocate registers to real-number variables first."
(iii) "Prefer forward chaining to backward chaining when there are few facts and many conclusions."
(iv) "Don't figure a register allocation of the same variable occurrence in a particular line twice."

(c) How does the branching factor change as the search proceeds (that is, with level in the search)? (choose only one answer)

(i) it decreases
(ii) it decreases or stays the same, depending on the state
(iii) it stays the same
(iv) it increases

9-17. Consider the task of proving a theorem from a set of postulates and assumptions. Suppose postulates, assumptions, and theorems are all either in the form of "A implies B" or just "A" where A and B are simple logical statements about the world. Suppose there is a way of taking any two postulates, assumptions, or previously derived theorems to get a conclusion, if any. For instance, if one statement says "A implies B" and another statement says "A", then we get a conclusion "B"; if one statement says "A implies B" and another statement says "B implies C", then we get a conclusion "A implies C".

(a) What is the search space?

(b) What is the branching factor, as a function of the number of steps taken?

(c) Give a search heuristic.

(d) Give a rough evaluation function for guiding search.

(e) Is the search monotonic?

(f) Is the search decomposable about an intermediate state?

(g) Is this a better problem for heuristics or for evaluation functions? Explain.

(h) Assuming we had a good approximate evaluation function, would the A* algorithm work well here? Explain.

(i) Would best-first search be a good idea here?

(j) Would bidirectional search be a good idea here?

9-18. Search has some surprising applications to numeric calculations. Consider the search for a numeric formula that approximates the value of some *target* variable from mathematical operations on a set of *study* variables. To test the accuracy of our approximation formula, we are given a set of *data points*, Prolog facts of the form

```
data_point([<study-variable-value-1>,<study-variable-value-2>, ... ],
  <target-variable-value>).
```

That is, the study variable values for a data point are stored in a list. Now think of this search problem as one of creating new list entries whose values match more closely the target variable values. To get these new study variables, we will do simple arithmetic operations on the old study variables. For example, take these data points:

```
data_point([1,2],6).
data_point([2,5],14).
data_point([3,2],10).
```

Then if we take the sum of the two study variables we can get a third study variable:

```
data_point([1,2,3],6).
data_point([2,5,7],14.)
data-point([3,2,5],10.)
```

and if we double those new values we can get

```
data_point([1,2,3,6],6).
data_point([2,5,7,14],14).
data_point([3,2,5,10],10).
```

and we have "explained" the value of the target variable as twice the sum of the original two study variables. In general, assume the following arithmetic operations are permissible on study variables:

> --multiplying by a constant or adding a constant to values of some study variable;
> --taking the negatives, squares, square roots, logarithms to the base 2 or powers of 2 of the values of some study variable;
> --taking the sum, difference, product, or quotient of the corresponding values for two different study variables of the same data point for all data points.

(a) Considered as a search problem, what is the search space?

(b) What are the operators?

(c) What is the branching factor from any state?

(d) For the following starting data points, draw the first two levels (not counting the starting state as a level) of the search graph for the portion including only the "squaring" operation (that is, multiplying values by themselves).

```
data_point([1,2],1).
data_point([2,0],16).
data_point([3,1],81).
```

(e) Give an evaluation function for this problem (the problem in general, not just the data points in part (e)).

(f) If the variable values represent experimental measurements, it will be difficult for a formula to match the target variable exactly. What then would be a reasonable goal condition?

(g) Give a general-purpose heuristic (nonnumeric way) for choosing branches well for this general problem (for any set of data points, not just those in part (d)).

(h) Is bidirectional search a good idea for this problem? Why?

(i) Is A* search a good idea for this problem? Why?

(j) Explain why preconditions are necessary for some of the operators.

(k) Is the problem decomposable about an intermediate state? Why?

(l) Professional statisticians often do this sort of analysis. They claim they don't use search techniques very often. If so, how (in artificial intelligence terminology) must they be solving these problems?

9-19. (E) (a) Explain how playing chess or checkers requires search.

(b) How is this kind of search fundamentally different from those that we have considered in this chapter?

Go to book index