6 ADVERSARIAL SEARCH

In which we examine the problems that arise when we try to plan ahead in a world where other agents are planning against us.

6.1 GAMES

Chapter 2 introduced **multiagent environments,** in which any given agent will need to consider the actions of other agents and how they affect its own welfare. The unpredictability of these other agents can introduce many possible **contingencies** into the agent's problem-solving process, as discussed in Chapter **3.** The distinction between **cooperative** and **competitive** multiagent environments was also introduced in Chapter 2. Competitive environments, in which the agents' goals are in conflict, give rise to **adversarial search** problems—often known as **games.**

GAMES

Mathematical **game theory,** a branch of economics, views any multiagent environment as a game provided that the impact of each agent on the others is "significant," regardless of whether the agents are cooperative or competitive.[1] In AI, .'games" are usually of a rather specialized kind—what game theorists call deterministic, turn-taking, two-player, **zero-sum games** of **perfect information.** In our terminology, this means deterministic, fully observable environments in which there are two agents whose actions must alternate and in which the utility values at the end of the game are always equal and opposite. For example, if one player wins a game of chess (+1), the other player necessarily loses (–1). It is this opposition between the agents' utility functions that makes the situation adversarial. We will consider multiplayer games, non-zero-sum games, and stochastic games briefly in this chapter, but will delay discussion of game theory proper until Chapter 17.

ZERO-SUM GAMES

PERFECT INFORMATION

Games have engaged the intellectual faculties of humans—sometimes to an alarming degree—for as long as civilization has existed. For AI researchers, the abstract nature of games makes them an appealing subject for study. The state of a game is easy to represent, and agents are usually restricted to a small number of actions whose outcomes are defined by

[1] Environments with very many agents are best viewed as **economies** rather than games.

different drink, and a different pet. Given the following facts, the question to answer is "Where does the zebra live, and in which house do they drink water?"

The Englishman lives in the red house.
The Spaniard owns the dog.
The Norwegian lives in the first house on the left.
Kools are smoked in the yellow house.
The man who smokes Chesterfields lives in the house next to the man with the fox.
The Norwegian lives next to the blue house.
The Winston smoker owns snails.
The Lucky Strike smoker drinks orange juice.
The Ukrainian drinks tea.
The Japanese smokes Parliaments.
Kools are smoked in the house next to the house where the horse is kept.
Coffee is drunk in the green house.
The Green house is immediately to the right (your right) of the ivory house.
Milk is drunk in the middle house.

Discuss different representations of this problem as a CSP. Why would one prefer one representation over another?

b. As a constraint satisfaction problem. Should the variables be words or letters?

Which formulation do you think will be better? Why?

5.5 Give precise formulations for each of the following as constraint satisfaction problems:

FLOOR-PLANNING

a. Rectilinear floor-planning: find nonoverlapping places in a large rectangle for a number of smaller rectangles.

CLASS SCHEDULING

b. **Class** scheduling: There is a fixed number of professors and classrooms, a list of classes to be offered, and a list of possible time slots for classes. Each professor has a set of classes that he or she can teach.

5.6 Solve the cryptarithmetic problem in Figure 5.2 by hand, using backtracking, forward checking, and the MRV and least-constraining-value heuristics.

5.7 Figure ***5.5*** tests out various algorithms on the n-queens problem. Try these same algorithms on map-coloring problems generated randomly as follows: scatter n points on the unit square; selecting a point X at random, connect X by a straight line to the nearest point Y such that X is not already connected to Y and the line crosses no other line; repeat the previous step until no more connections are possible. Construct the performance table for the largest n you can manage, using both d = **3** and d = 4 colors. Comment on your results.

5.8 Use the AC-3 algorithm to show that arc consistency is able to detect the inconsistency of the partial assignment { WA = *red,* V = blue) for the problem shown in Figure 5.1.

5.9 What is the worst-case complexity of running AC-3 on a tree-structured CSP?

5.10 AC-3 puts back on the queue every arc (XI,,X_i) whenever any value is deleted from the domain of X_i, even if each value of XI, is consistent with several remaining values of X_i. Suppose that, for every arc (XI,,Xi), we keep track of the number of remaining values of X_i that are consistent with each value of XI,. Explain how to update these numbers efficiently and hence show that arc consistency can be enforced in total time $O(n^2d^2)$.

5 . 1 Show how a single ternary constraint such as "A + B = C" can be turned into three binary constraints by using an auxiliary variable. You may assume finite domains. (Hint: consider a new variable that takes on values which are pairs of other values, and consider constraints such as "X is the first element of the pair Y.") Next, show how constraints with more than three variables can be treated similarly. Finally, show how unary constraints can be eliminated by altering the domains of variables. This completes the demonstration that any CSP can be transformed into a CSP with only binary constraints.

5.12 Suppose that a graph is known to have a cycle cutset of no more than k nodes. Describe a simple algorithm for finding a minimal cycle cutset whose runtime is not much more than $O(n^k)$ for a CSP with n variables. Search the literature for methods for finding approximately minimal cycle cutsets in time that is polynomial in the size of the cutset. Does the existence of such algorithms make the cycle cutset method practical?

5.13 Consider the following logic puzzle: In five houses, each with a different color, live 5 persons of different nationalities, each of whom prefer a different brand of cigarette, a its constraint graph. The notion of tree width was introduced by the graph theorists Robertson and Seymour (1986). Dechter and Pearl (1987, 1989), building on the work of Freuder, applied the same notion (which they called induced width) to constraint satisfaction problems and developed the tree decomposition approach sketched in Section 5.4. Drawing on this work and on results from database theory, Gottlob *et al.* (1999a, 1999b) developed a notion, hypertree **width,** that is based on the characterization of the CSP as a hypergraph. In addition to showing that any CSP with hypertree width w can be solved in time $O(n^{w+1} \log n)$, they also showed that hypertree width subsumes all previously defined measures of "width" in the sense that there are cases where the hypertree width is bounded and the other measures are unbounded.

There are several good surveys of CSP techniques, including those by Kumar (1992), Dechter and Frost (1999), and Bartak (2001); and the encyclopedia articles by Dechter (1992) and Mackworth (1992). Pearson and Jeavons (1997) survey tractable classes of CSPs, covering both structural decomposition methods and methods that rely on properties of the domains or constraints themselves. Kondrak and van Beek (1997) give an analytical survey of backtracking search algorithms, and Bacchus and van Run (1995) give a more empirical survey. The texts by Tsang (1993) and by Marriott and Stuckey (1998) go into much more depth than has been possible in this chapter. Several interesting applications are described in the collection edited by Freuder and Mackworth (1994). Papers on constraint satisfaction appear regularly in *Artificial Intelligence* and in the specialist journal, *Constraints.* The primary conference venue is the International Conference on Principles and Practice of Constraint Programming, often called CP.

EXERCISES

5.1 Define in your own words the terms constraint satisfaction problem, constraint, backtracking search, arc consistency, backjumping and min-conflicts.

5.2 How many solutions are there for the map-coloring problem in Figure 5.1?

5.3 Explain why it is a good heuristic to choose the variable that is *most* constrained, but the value that is *least* constraining in a CSP search.

5.4 Consider the problem of constructing (not solving) crossword puzzles:[5] fitting words into a rectangular grid. The grid, which is given as part of the problem, specifies which squares are blank and which are shaded. Assume that a list of words (i.e., a dictionary) is provided and that the task is to fill in the blank squares using any subset of the list. Formulate this problem precisely in two ways:

a. As a general search problem. Choose an appropriate search algorithm, and specify a heuristic function, if you think one is needed. Is it better to fill in blanks one letter at a time or one word at a time?

[5] Ginsberg *et al.* (1990) discuss several methods for constructing crossword puzzles. Littman *et al.* (1999) tackle the harder problem of solving them.

assignment—an algorithm later called MAC by Sabin and Freuder (1994). The latter paper provides somewhat convincing evidence that, on harder CSPs, full arc consistency checlung pays off. Freuder (1978, 1982) investigated the notion of k-consistency and its relationship to the complexity of solving CSPs. Apt (1999) describes a generic algorithmic framework within which consistency propagation algorithms can be analyzed.

Special methods for handling higher-order constraints have been developed primarily within the context of **constraint logic programming.** Marriott and Stuckey (1998) provide excellent coverage of research in this area. The *Alldiff* constraint was studied by Regin (1994). Bounds constraints were incorporated into constraint logic programming by Van Hentenryck *et al.* (1998).

The basic backjumping method is due to John Gaschnig (1977, 1979). Kondrak and van Beek (1997) showed that this algorithm is essentially subsumed by forward checking. Conflict-directed backjumping was devised by Prosser (1993). The most general and powerful form of intelligent backtracking was actually developed very early on by Stallman and Sussman (1977). Their technique of **dependency-directed backtracking** led to the development of **truth maintenance systems** (Doyle, 1979), which we will discuss in Section 10.8. The connection between the two areas is analyzed by de Kleer (1989).

DEPENDENCY-DIRECTED BACKTRACKING

The work of Stallman and Sussman also introduced the idea of **constraint recording,** in which partial results obtained by search can be saved and reused later in the search. The idea was introduced formally into backtracking search by Dechter (1990a). **Backmarking** (Gaschnig, 1979) is a particularly simple method in which consistent and inconsistent pairwise assignments are saved and used to avoid rechecking constraints. Backmarking can be combined with conflict-directed backjumping; Kondrak and van Beek (1997) present a hybrid algorithm that provably subsumes either method taken separately. The method of **dynamic backtracking** (Ginsberg, 1993) retains successful partial assignments from later subsets of variables when backtracking over an earlier choice that does not invalidate the later success.

CONSTRAINT RECORDING

BACKMARKING

DYNAMIC BACKTRACKING

Local search in constraint satisfaction problems was popularized by the work of Kirkpatrick *et al.* (1983) on **simulated annealing** (see Chapter 4), which is widely used for scheduling problems. The *min-conflicts* heuristic was first proposed by Gu (1989) and was developed independently by Minton *et* al. (1992). Sosic and Gu (1994) showed how it could be applied to solve the 3,000,000 queens problem in less than a minute. The astounding success of local search using min-conflicts on the n-queens problem led to a reappraisal of the nature and prevalence of "easy" and "hard problems. Peter Cheeseman *et al.* (1991) explored the difficulty of randomly generated CSPs and discovered that almost all such problems either are trivially easy or have no solutions. Only if the parameters of the problem generator are set in a certain narrow range, within which roughly half of the problems are solvable, do we find "hard" problem instances. We discuss this phenomenon further in Chapter 7.

Work relating the structure and complexity of CSPs originates with Freuder (1985), who showed that search on arc-consistent trees works without any backtracking. A similar result, with extensions to acyclic hypergraphs, was developed in the database community (Beeri *et* al., 1983). Since those papers were published, there has been a great deal of progress in developing more general results relating the complexity of solving a CSP to the structure of

Bibliographical and Historical Notes

The earliest work related to constraint satisfaction dealt largely with numerical constraints. Equational constraints with integer domains were studied by the Indian mathematician Brahmagupta in the seventh century; they are often called **Diophantine equations,** after the Greek mathematician Diophantus (c. 200–284), who actually considered the domain of positive rationals. Systematic methods for solving linear equations by variable elimination were studied by Gauss (1829); the solution of linear inequality constraints goes back to Fourier (1827).

DIOPHANTINE EQUATIONS

Finite-domain constraint satisfaction problems also have a long history. For example, **graph coloring** (of which map coloring is a special case) is an old problem in mathematics. According to Biggs *et al.* (1986), the four-color conjecture (that every planar graph can be colored with four or fewer colors) was first made by Francis Guthrie, a student of De Morgan, in 1852. It resisted solution—despite several published claims to the contrary—until a proof was devised, with the aid of a computer, by Appel and Haken (1977).

GRAPH COLORING

Specific classes of constraint satisfaction problems occur throughout the history of computer science. One of the most influential early examples was the SKETCHPAD system (Sutherland, 1963), which solved geometric constraints in diagrams and was the forerunner of modern drawing programs and CAD tools. The identification of CSPs as a *general* class is due to Ugo Montanari (1974). The reduction of higher-order CSPs to purely binary CSPs with auxiliary variables (see Exercise 5.11) is due originally to the 19th-century logician Charles Sanders Peirce. It was introduced into the CSP literature by Dechter (1990b) and was elaborated by Bacchus and van Beek (1998). CSPs with preferences among solutions are studied widely in the optimization literature; see Bistarelli *et al.* (1997) for a generalization of the CSP framework to allow for preferences. The bucket-elimination algorithm (Dechter, 1999) can also be applied to optimization problems.

Backtracking search for constraint satisfaction is due to Bitner and Reingold (1975), although they trace the basic algorithm back to the 19th century. Bitner and Reingold also introduced the MRV heuristic, which they called the *most-constrained-variable* heuristic. Brelaz (1979) used the degree heuristic as a tie-breaker after applying the MRV heuristic. The resulting algorithm, despite its simplicity, is still the best method for k-coloring arbitrary graphs. Haralick and Elliot (1980) proposed the *least-constraining-value heuristic.*

Constraint propagation methods were popularized by Waltz's (1975) success on polyhedral line-labeling problems for computer vision. Waltz showed that, in many problems, propagation completely eliminates the need for backtracking. Montanari (1974) introduced the notion of constraint networks and propagation by path consistency. Alan Mackworth (1977) proposed the AC-3 algorithm for enforcing arc consistency as well as the general idea of combining backtracking with some degree of consistency enforcement. AC-4, a more efficient arc consistency algorithm, was developed by Mohr and Henderson (1986). Soon after Mackworth's paper appeared, researchers began experimenting with the tradeoff between the cost of consistency enforcement and the benefits in terms of search reduction. Haralick and Elliot (1980) favored the minimal forward checking algorithm described by McGregor (1979), whereas Gaschnig (1979) suggested full arc consistency checking after each variable a global solution as follows. First, we view each subproblem as a "mega-variable" whose domain is the set of all solutions for the subproblem. For example, the leftmost subproblems in Figure 5.12 is a map-coloring problem with three variables and hence has six solutions—one is $\{WA = red, SA = blue, NT = green)$. Then, we solve the constraints connecting the subproblems using the efficient algorithm for trees given earlier. The constraints between subproblems simply insist that the subproblem solutions agree on their shared variables. For example, given the solution $\{WA = red, SA = blue, NT = green\}$ for the first subproblem, the only consistent solution for the next subproblem is $\{SA = blue, NT = green, Q = red)$.

TREE WIDTH

A given constraint graph admits many tree decompositions; in choosing a decomposition, the aim is to make the subproblems as small as possible. The **tree width** of a tree decomposition of a graph is one less than the size of the largest subproblem; the tree width of the graph itself is defined to be the minimum tree width among all its tree decompositions. If a graph has tree width w, and we are given the corresponding tree decomposition, then the problem can be solved in $O(nd^{w+1})$ time. Hence, *CSPs with constraint graphs of bounded tree width are solvable in polynomial time.* Unfortunately, finding the decomposition with minimal tree width is NP-hard, but there are heuristic methods that work well in practice.

5.5 Summary

- **Constraint satisfaction problems** (or CSPs) consist of variables with constraints on them. Many important real-world problems can be described as CSPs. The structure of a CSP can be represented by its **constraint graph.**
- **Backtracking search,** a form of depth-first search, is commonly used for solving CSPs.
- The **minimum remaining values** and **degree** heuristics are domain-independent methods for deciding which variable to choose next in a backtracking search. The **least-constraining-value** heuristic helps in ordering the variable values.
- By propagating the consequences of the partial assignments that it constructs, the backtracking algorithm can reduce greatly the branching factor of the problem. **Forward checking** is the simplest method for doing this. **Arc consistency** enforcement is a more powerful technique, but can be more expensive to run.
- Backtracking occurs when no legal assignment can be found for a variable. **Conflict-directed backjumping** backtracks directly to the source of the problem.
- Local search using the **min-conflicts** heuristic has been applied to constraint satisfaction problems with great success.
- The complexity of solving a CSP is strongly related to the structure of its constraint graph. Tree-structured problems can be solved in linear time. **Cutset conditioning** can reduce a general CSP to a tree-structured one and is very efficient if a small cutset can be found. **Tree decomposition** techniques transform the CSP into a tree of subproblems and are efficient if the **tree width** of the constraint graph is small.

In the worst case, however, c can be as large as $(n-2)$. Finding the *smallest* cycle **cutset** is NP-hard, but several efficient approximation algorithms are known for this task. The overall algorithmic approach is called **cutset** conditioning; we will see it again in Chapter 14, where it is used for reasoning about probabilities.

CUTSET CONDITIONING

TREE DECOMPOSITION

The second approach is based on constructing a tree decomposition of the constraint graph into a set of connected subproblems. Each subproblem is solved independently, and the resulting solutions are then combined. Like most divide-and-conquer algorithms, this works well if no subproblem is too large. Figure 5.12 shows a tree decomposition of the map-coloring problem into five subproblems. A tree decomposition must satisfy the following three requirements:

- Every variable in the original problem appears in at least one of the subproblems.
- If two variables are connected by a constraint in the original problem, they must appear together (along with the constraint) in at least one of the subproblems.
- If a variable appears in two subproblems in the tree, it must appear in every subproblem along the path connecting those subproblems.

The first two conditions ensure that all the variables and constraints are represented in the decomposition. The third condition seems rather technical, but simply reflects the constraint that any given variable must have the same value in every subproblem in which it appears; the links joining subproblems in the tree enforce this constraint. For example, SA appears in all four of the connected subproblems in Figure 5.12. You can verify from Figure 5.11 that this decomposition makes sense.

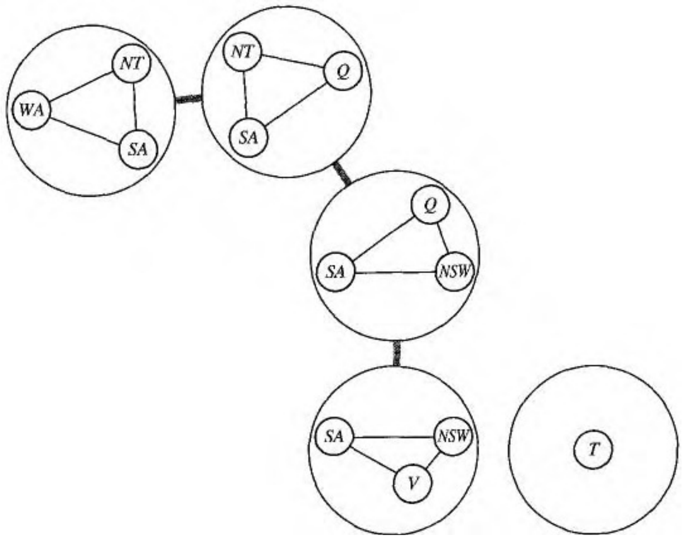


Figure 5.12 A tree decomposition of the constraint graph in Figure 5.11(a).

We solve each subproblem independently; if any one has no solution, we know the entire problem has no solution. If we can solve all the subproblems, then we attempt to construct

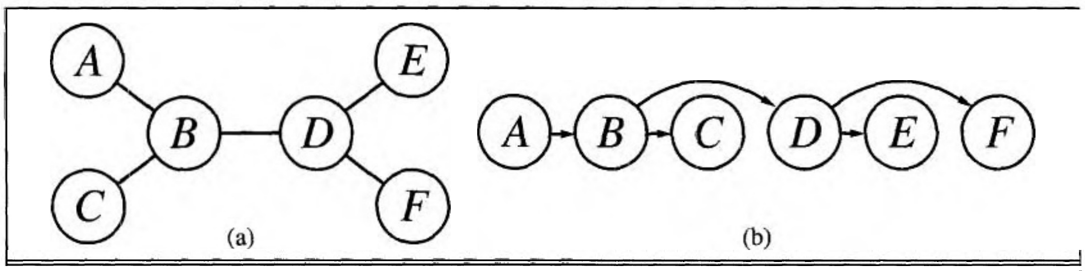


Figure 5.10 (a) The constraint graph of a tree-structured CSP. (b) A linear ordering of the variables consistent with the tree with A as the root.

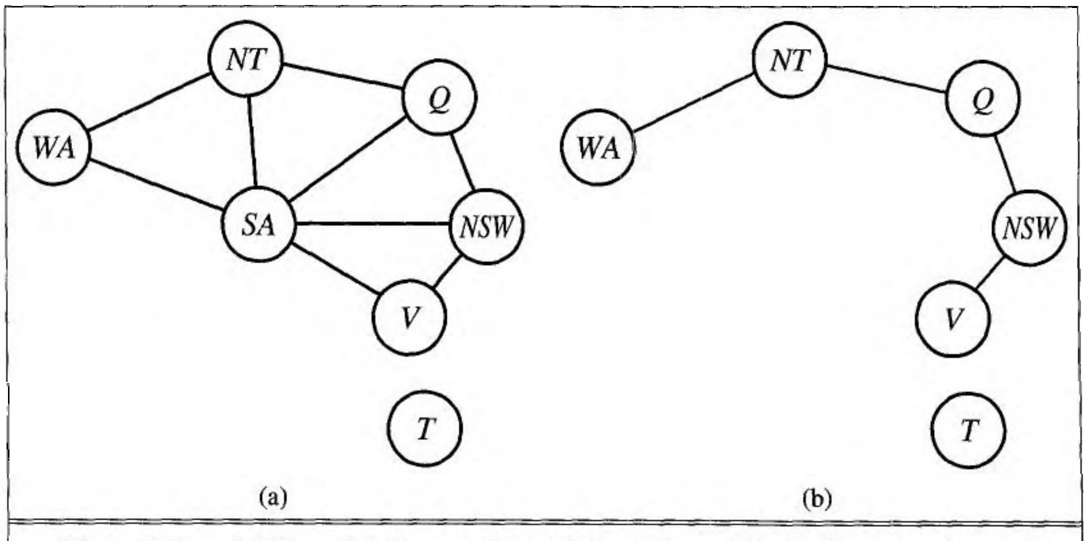


Figure 5.11 (a) The original constraint graph from Figure 5.1. (b) The constraint graph after the removal of SA.

Now, any solution for the CSP after SA and its constraints are removed will be consistent with the value chosen for SA. (This works for binary CSPs; the situation is more complicatecl with higher-order constraints.) Therefore, we can solve the remaining tree with the algorithm given above and thus solve the whole problem. Of course, in the general case (as opposed to map coloring) the value chosen for SA could be the wrong one, so we would need to try each of them. The general algorithm is as follows:

1. Choose a subset S from VARIABLES[csp] such that the constraint graph becomes a tree after removal of S. S is called a **cycle cutset**. CYCLE CUTSET
2. For each possible assignment to the variables in S that satisfies all constraints on S,
   (a) remove from the domains of the remaining variables any values that are inconsistent with the assignment for S, and
   (b) If the remaining CSP has a solution, return it together with the assignment for S.

If the cycle cutset has size c, then the total runtime is $O(d^c \cdot (n - c)d^2)$. If the graph is "nearly a tree" then c will be small and the savings over straight backtracking will be huge.

structure, one fact stands out: Tasmania is not connected to the mainland.[3] Intuitively, it is obvious that coloring Tasmania and coloring the mainland are independent subproblems — any solution for the mainland combined with any solution for Tasmania yields a solution for the whole map. Independence can be ascertained simply by looking for connected components of the constraint graph. Each component corresponds to a subproblem CSP_i. If assignment S_i is a solution of CSP,, then $\bigcup_i S_i$ is a solution of $\bigcup_i$ CSP,. Why is this important? Consider the following: suppose each CSP, has c variables from the total of n variables, where c is a constant. Then there are n/c subproblems, each of which takes at most d^c work to solve. Hence, the total work is $O(d^c n/c)$, which is *linear* in n; without the decomposition, the total work is $O(d^n)$, which is exponential in n. Let's make this more concrete: dividing a Boolean CSP with $n = 80$ into four subproblems with $c = 20$ reduces the worst-case solution time from the lifetime of the universe down to less than a second.

INDEPENDENT SUBPROBLEMS

CONNECTED COMPONENTS

Completely independent subproblems are delicious, then, but rare. In most cases, the subproblems of a CSP are connected. The simplest case is when the constraint graph forms a tree: any two variables are connected by at most one path. Figure 5.10(a) shows a schematic example.[4] We will show that *any tree-structured CSP can be solved in time linear in the number of variables.* The algorithm has the following steps:

1. Choose any variable as the root of the tree, and order the variables from the root to the leaves in such a way that every node's parent in the tree precedes it in the ordering. (See Figure 5.10(b).) Label the variables $X_1, \ldots, X_n$ in order. Now, every variable except the root has exactly one parent variable.
2. For j from n down to 2, apply arc consistency to the arc (X_i, X_j), where X_i is the parent of X_j, removing values from DOMAIN$[X_i]$ as necessary.
3. For j from 1 to n, assign any value for X_j consistent with the value assigned for X_i, where X_i is the parent of X_j.

There are two key points to note. First, after step 2 the CSP is directionally arc-consistent, so the assignment of values in step 3 requires no backtracking. (See the discussion of k-consistency on page 147.) Second, by applying the arc-consistency checks in reverse order in step 2, the algorithm ensures that any deleted values cannot endanger the consistency of arcs that have been processed already. The complete algorithm runs in time $O(nd^2)$.

Now that we have an efficient algorithm for trees, we can consider whether more general constraint graphs can be *reduced* to trees somehow. There are two primary ways to do this, one based on removing nodes and one based on collapsing nodes together.

The first approach involves assigning values to some variables so that the remaining variables form a tree. Consider the constraint graph for Australia, shown again in Figure 5.11(a). If we could delete South Australia, the graph would become a tree, as in (b). Fortunately, we can do this (in the graph, not the continent) by fixing a value for SA and deleting from the domains of the other variables any values that are inconsistent with the value chosen for SA.

[3] A careful cartographer or patriotic Tasmanian might object that Tasmania should not be colored the same as its nearest mainland neighbor, to avoid the impression that it might be part of that state.

[4] Sadly, very few regions of the world, with the possible exception of Sulawesi, have tree-structured maps.

```
function MIN-CONFLICTS(csp, max-steps) returns a solution or failure
  inputs: csp, a constraint satisfaction problem
          max-steps, the number of steps allowed before giving up

  current ← an initial complete assignment for csp
  for i = 1 to max-steps do
      if current is a solution for csp then return current
      var ← a randomly chosen, conflicted variable from VARIABLES[csp]
      value ← the value v for var that minimizes CONFLICTS(var, v, current, csp)
      set var = value in current
  return failure
```

Figure 5.8 The *MIN-CONFLICTS* algorithm for solving CSPs by local search. The initial state may be chosen randomly or by a greedy assignment process that chooses a minimal-conflict value for each variable in turn. The *CONFLICTS* function counts the number of constraints violated by a particular value, given the rest of the current assignment.

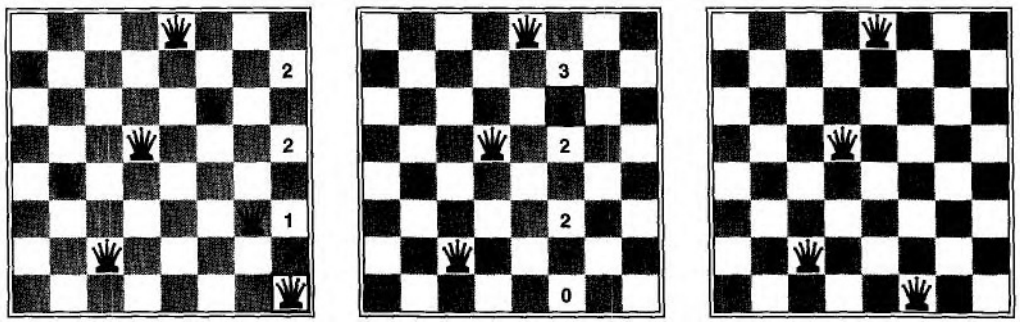


Figure 5.9 A two-step solution for an 8-queens problem using min-conflicts. At each stage, a queen is chosen for reassignment in its colunm. The number of conflicts (in this case, the number of attacking queens) is shown in each square. The algorithm moves the queen to the min-conflict square, breaking ties randomly.

constraints usually requires much more time and might find a solution with many changes from the current schedule.

5.4 THE STRUCTURE OF PROBLEMS

In this section, we examine ways in which the *structure* of the problem, as represented by the constraint graph, can be used to find solutions quickly. Most of the approaches here are very general and are applicable to other problems besides CSPs, for example probabilistic reasoning. After all, the only way we can possibly hope to deal with the real world is to decompose it into many subproblems. Looking again at Figure 5.1(b) with a view to identifying problem to the most recent variable X_i in conf (X_j), and set

$$\text{conf}(X_i) \leftarrow \text{conf}(X_i) \cup conf(X_j) - \{X_i\} .$$

Conflict-directed backjumping takes us back to the right point in the search tree, but doesn't prevent us from making the same mistakes in another branch of the tree. **Constraint learning** actually modifies the CSP by adding a new constraint that is induced from these conflicts.

CONSTRAINT LEARNING

5.3 LOCAL SEARCH FOR CONSTRAINT SATISFACTION PROBLEMS

Local-search algorithms (see Section 4.3) turn out to be very effective in solving many CSPs. They use a complete-state formulation: the initial state assigns a value to every variable, and the successor function usually works by changing the value of one variable at a time. For example, in the 8-queens problem, the initial state might be a random configuration of 8 queens in 8 columns, and the successor function picks one queen and considers moving it elsewhere in its column. Another possibility would be start with the 8 queens, one per column in a permutation of the 8 rows, and to generate a successor by having two queens swap rows.[2] We have actually already seen an example of local search for CSP solving: the application of hill climbing to the n-queens problem (page 112). The application of WALKSAT (page 223) to solve satisfiability problems, which are a special case of CSPs, is another.

In choosing a new value for a variable, the most obvious heuristic is to select the value that results in the minimum number of conflicts with other variables—the **min-conflicts** heuristic. The algorithm is shown in Figure 5.8 and its application to an 8-queens problem is diagrammed in Figure 5.9 and quantified in Figure 5.5.

MIN-CONFLICTS

Min-conflicts is surprisingly effective for many CSPs, particularly when given a reasonable initial state. Its performance is shown in the last column of Figure 5.5. Amazingly, on the n-queens problem, if you don't count the initial placement of queens, the runtime of min-conflicts is roughly independent of problem size. It solves even the million-queens problem in an average of 50 steps (after the initial assignment). This remarkable observation was the stimulus leading to a great deal of research in the 1990s on local search and the distinction between easy and hard problems, which we take up in Chapter 7. Roughly speaking, n-queens is easy for local search because solutions are densely distributed throughout the state space. Min-conflicts also works well for hard problems. For example, it has been used to schedule observations for the Hubble Space Telescope, reducing the time taken to schedule a week of observations from three weeks (!) to around 10 minutes.

Another advantage of local search is that it can be used in an online setting when the problem changes. This is particularly important in scheduling problems. A week's airline schedule may involve thousands of flights and tens of thousands of personnel assignments, but bad weather at one airport can render the schedule infeasible. We would like to repair the schedule with a minimum number of changes. This can be easily done with a local search algorithm starting from the current schedule. A backtracking search with the new set of

[2] Local search can easily be extended to CSPs with objective functions. In that case, all the techniques for hill climbing and simulated annealing can be applied to optimize the objective function.

backtracks to the *most recent* variable in the conflict set; in this case, backjumping would jump over Tasmania and try a new value for V. This is easily implemented by modifying BACKTRACKING-SEARCH so that it accumulates the conflict set while checking for a legal value to assign. If no legal value is found, it should return the most recent element of the conflict set along with the failure indicator.

The sharp-eyed reader will have noticed that forward checking can supply the conflict set with no extra work: whenever forward checking based on an assignment to X deletes a value frorn Y's domain, it should add X to Y's conflict set. Also, every time the last value is deleted from Y's domain, the variables in the conflict set of Y are added to the conflict set of X. Then, when we get to Y, we know immediately where to backtrack if needed.

The eagle-eyed reader will have noticed something odd: backjumping occurs when every value in a domain is in conflict with the current assignment; but forward checking detects this event and prevents the search from ever reaching such a node! In fact, it can be shown that *every branch pruned by backjumping is also pruned by forward checking.* Hence, simple backjumping is redundant in a forward-checking search or, indeed, in a search that uses stronger consistency checking, such as MAC.

Despite the observations of the preceding paragraph, the idea behind backjumping remains a good one: to backtrack based on the reasons for failure. Backjumping notices failure when a variable's domain becomes empty, but in many cases a branch is doomed long before this occurs. Consider again the partial assignment { *WA* = red, NSW = red) (which, from our earlier discussion, is inconsistent). Suppose we try T = red next and then assign NT, Q, V, SA. We know that no assignment can work for these last four variables, so eventually we run out of values to try at NT. Now, the question is, where to backtrack? Backjumping cannot work, because NT *does* have values consistent with the preceding assigned variables — NT doesn't have a complete conflict set of preceding variables that caused it to fail. We know, however, that the four variables NT, Q, V, and SA, *taken together,* failed because of a set of preceding variables, which must be those variables which directly conflict with the four. This leads to a deeper notion of the conflict set for a variable such as NT: it is that set of preceding variables that caused NT, *together with any subsequent variables,* to have no consistent solution. In this case, the set is *WA* and NSW, so the algorithm should backtrack to NSW and skip over Tasmania. A backjumping algorithm that uses conflict sets defined in this way is called **conflict-directed backjumping.**

CONFLICT-DIRECTED BACKJUMPING

We must now explain how these new conflict sets are computed. The method is in fact very simple. The "terminal" failure of a branch of the search always occurs because a variable's domain becomes empty; that variable has a standard conflict set. In our example, SA fails, and its conflict set is (say) $\{WA, NT, Q\}$. We backjump to Q, and Q *absorbs* the conflict set from SA (minus Q itself, of course) into its own direct conflict set, which is {NT, NSW); the new conflict set is { WA, NT, NSW}. That is, there is no solution from Q onwards, given the preceding assignment to { WA, NT, NSW). Therefore, we backtrack to NT, the most recent of these. NT absorbs {WA, NT, NSW) – {NT) into its own direct conflict set { WA}, giving { WA, NSW) (as stated in the previous paragraph). Now the algorithm backjumps to NSW, as we would hope. To summarize: let X_j be the current variable, and let $conf(X_j)$ be its conflict set. If every possible value for X_j fails, backjump tive than applying arc consistency to an equivalent set of binary constraints.

RESOURCE CONSTRAINT

Perhaps the most important higher-order constraint is the resource constraint, sometimes called the *atmost* constraint. For example, let $PA_1, \ldots, PA_4$ denote the numbers of personnel assigned to each of four tasks. The constraint that no more than 10 personnel are assigned in total is written as $atmost(10, PA_1, PA_2, PA_3, PA_4)$. An inconsistency can be detected simply by checking the sum of the minimum values of the current domains; for example, if each variable has the domain $\{3, 4, 5, 6\}$, the *atmost* constraint cannot be satisfied. We can also enforce consistency by deleting the maximum value of any domain if it is not consistent with the minimum values of the other domains. Thus, if each variable in our example has the domain $\{2, 3, 4, 5, 6\}$, the values 5 and 6 can be deleted from each domain.

For large resource-limited problems with integer values—such as logistical problems involving moving thousands of people in hundreds of vehicles—it is usually not possible to represent the domain of each variable as a large set of integers and gradually reduce that set by consistency checking methods. Instead, domains are represented by upper and lower bounds and are managed by bounds propagation. For example, let's suppose there are two flights, 271 and 272, for which the planes have capacities 165 and 385, respectively. The initial domains for the numbers of passengers on each flight are then

$$Flight271 \in [0, 165] \quad \text{and} \quad Flight272 \in [0, 385] \,.$$

Now suppose we have the additional constraint that the two flights together must carry 420 people: $Flight271 + Flight272 \in [420, 420]$. Propagating bounds constraints, we reduce the domains to

$$Flight271 \in [35, 165] \quad \text{and} \quad Flight272 \in [255, 385] \,.$$

We say that a CSP is bounds-consistent if for every variable X, and for both the lower bound and upper bound values of X, there exists some value of Y that satisfies the constraint between X and Y, for every variable Y. This kind of bounds propagation is widely used in practical constraint problems.

BOUNDS PROPAGATION

Intelligent backtracking: looking backward

The BACKTRACKING-SEARCH algorithm in Figure 5.3 has a very simple policy for what to do when a branch of the search fails: back up to the preceding variable and try a different value for it. This is called chronological backtracking, because the *most recent* decision point is revisited. In this subsection, we will see that there are much better ways.

CHRONOLOGICAL BACKTRACKING

Consider what happens when we apply simple backtracking in Figure 5.1 with a fixed variable ordering Q, NSW, V, T, SA, WA, NT. Suppose we have generated the partial assignment $\{Q = red, NSW = green, V = blue, T = red\}$. When we try the next variable, SA, we see that every value violates a constraint. We back up to T and try a new color for Tasmania! Obviously this is silly—recoloring Tasmania cannot resolve the problem with South Australia.

A more intelligent approach to backtracking is to go all the way back to one of the set of variables that *caused the failure.* This set is called the conflict set; here, the conflict set for SA is $\{Q, NSW, V\}$. In general, the conflict set for variable X is the set of previously assigned variables that are connected to X by constraints. The backjumping method

CONFLICT SET

BACKJUMPING

K-CONSISTENCY

sistency. Stronger forms of propagation can be defined using the notion called k-consistency. A CSP is k-consistent if, for any set of $k - 1$ variables and for any consistent assignment to those variables, a consistent value can always be assigned to any kth variable. For example, 1-consistency means that each individual variable by itself is consistent; this is also called node consistency. 2-consistency is the same as arc consistency. 3-consistency means that any pair of adjacent variables can always be extended to a third neighboring variable; this is also called path consistency.

NODE CONSISTENCY

PATH CONSISTENCY

STRONGLY K-CONSISTENT

A graph is strongly k-consistent if it is k-consistent and is also (k − 1)-consistent, (k − 2)-consistent,... all the way down to 1-consistent. Now suppose we have a CSP problem with n nodes and make it strongly n-consistent (i.e., strongly k-consistent for k = n). We can then solve the problem with no backtracking. First, we choose a consistent value for X_1. We are then guaranteed to be able to choose a value for X_2 because the graph is 2-consistent, for X_3 because it is 3-consistent, and so on. For each variable X_i, we need only search through the d values in the domain to find a value consistent with $X_1, \ldots, X_{i-1}$. We are guaranteed to find a solution in time $O(nd)$. Of course, there is no free lunch: any algorithm for establishing n-consistency must take time exponential in n in the worst case.

There is a broad middle ground between n-consistency and arc consistency: running stronger consistency checks will take more time, but will have a greater effect in reducing the branching factor and detecting inconsistent partial assignments. It is possible to calculate the smallest value k such that running k-consistency ensures that the problem can be solved without backtracking (see Section 5.4), but this is often impractical. In practice, determining the appropriate level of consistency checking is mostly an empirical science.

Handling special constraints

Certain types of constraints occur frequently in real problems and can be handled using special-purpose algorithms that are more efficient than the general-purpose methods described so far. For example, the *Alldiff* constraint says that all the variables involved must have distinct values (as in the cryptarithmetic problem). One simple form of inconsistency detection for *Alldiff* constraints works as follows: if there are m variables involved in the constraint, and if they have n possible distinct values altogether, and m $>$ n, then the constraint cannot be satisfied.

This leads to the following simple algorithm: First, remove any variable in the constraint that has a singleton domain, and delete that variable's value from the domains of the remaining variables. Repeat as long as there are singleton variables. If at any point an empty domain is produced or there are more variables than domain values left, then an inconsistency has been detected.

We can use this method to detect the inconsistency in the partial assignment { WA = red, NSW = red) for Figure 5.1. Notice that the variables SA, NT, and Q are effectively connected by an *Alldiff* constraint because each pair must be a different color. After applying AC-3 with the partial assignment, the domain of each variable is reduced to {green, blue). That is, we have three variables and only two colors, so the *Alldiff* constraint is violated. Thus, a simple consistency procedure for a higher-order constraint is sometimes more effec-

```
function AC-3( csp) returns the CSP, possibly with reduced domains
  inputs: csp, a binary CSP with variables $\{X_1, X_2, \ldots, X_n\}$
  local variables: queue, a queue of arcs, initially all the arcs in csp

  while queue is not empty do
    $(X_i, X_j) \leftarrow$ REMOVE-FIRST(queue)
    if REMOVE-INCONSISTENT-VALUES($X_i$, $X_j$) then
      for each $X_k$ in NEIGHBORS[$X_i$] - $\{X_j\}$ do
        add $(X_k, X_i)$ to queue

function REMOVE-INCONSISTENT-VALUES($X_i$, $X_j$) returns true iff we remove a value
  removed $\leftarrow$ false
  for each x in DOMAIN[$X_i$] do
    if no value $y$ in DOMAIN[$X_j$] allows $(x,y)$ to satisfy the constraint between $X_i$ and $X_j$
      then delete x from DOMAIN[$X_i$]; removed $\leftarrow$ true
  return removed
```

Figure 5.7 The arc consistency algorithm AC-3. After applying AC-3, either every arc is arc-consistent, or some variable has an empty domain, indicating that the CSP cannot be made arc-consistent (and thus the CSP cannot be solved). The name "AC-3" was used by the algorithm's inventor (Mackworth, 1977) because it's the third version developed in the paper.

tency that is not detected by pure forward checking.

Arc consistency checking can be applied either as a preprocessing step before the beginning of the search process, or as a propagation step (like forward checking) after every assignment during search. (The latter algorithm is sometimes called MAC, for *Maintaining Arc Consistency.)* In either case, the process must be applied repeatedly until no more inconsistencies remain. This is because, whenever a value is deleted from some variable's domain to remove an arc inconsistency, a new arc inconsistency could arise in arcs pointing to that variable. The full algorithm for arc consistency, AC-3, uses a queue to keep track of the arcs that need to be checked for inconsistency. (See Figure 5.7.) Each arc (X_i, X_j) in turn is removed from the agenda and checked; if any values need to be deleted from the domain of X_i, then every arc (X_k, X_i) pointing to X_i must be reinserted on the queue for checking. The complexity of arc consistency checking can be analyzed as follows: a binary CSP has at most $O(n^2)$ arcs; each arc (X_k, X_i) can be inserted on the agenda only d times, because X_i has at most d values to delete; checking consistency of an arc can be done in $O(d^2)$ time; so the total worst-case time is $O(n^2d^3)$. Although this is substantially more expensive than forward checking, the extra cost is usually worthwhile.'

Because CSPs include 3SAT as a special case, we do not expect to find a polynomial-time algorithm that can decide whether a given CSP is consistent. Hence, we deduce that arc consistency does not reveal every possible inconsistency. For example, in Figure 5.1, the partial assignment { WA = red, NSW = red) is inconsistent, but AC-3 will not find the incon-

[1] The AC-4 algorithm, due to Mohr and Henderson (1986), runs in $O(n^2d^2)$. See Exercise 5.10.

	WA	*NT*	*Q*	*NSW*	*V*	*SA*	*T*
Initial domains	R G B	R G B	R G B	R G B	R G B	R G B	R G B
After *WA=red*	R	G B	R G B	R G B	R G B	G B	R G B
After *Q=green*	8	B	G	R B	R G B	B	R G B
After *V=blue*	R	B	[illegible]	R	B		R G B

Figure 5.6 The progress of a map-coloring search with forward checking. $WA = red$ is assigned first; then forward checking deletes *red* from the domains of the neighboring variables NT and *SA*. After Q = *green, green* is deleted from the domains of NT, *SA,* and *NS*W. After V = *blue, blue* is deleted from the domains of NSW and SA, leaving SA with no legal values.

MRV heuristic needs to do its job.) A second point to notice is that, after V = *blue,* the domain of SA is empty. Hence, forward checking has detected that the partial assignment { *WA* = *red,* Q = *green,V* = *blue)* is inconsistent with the constraints of the problem, and the algorithm will therefore backtrack immediately.

Constraint propagation

Although forward checking detects many inconsistencies, it does not detect all of them. For example, consider the third row of Figure 5.6. It shows that when WA is *red* and Q is *green,* both N T and *SA* are forced to be blue. But they are ,adjacent and so cannot have the same value. Forward checking does not detect this as an inconsistency, because it does not look far enough ahead. **Constraint propagation** is the general term for propagating the implications of a constraint on one variable onto other variables; in this case we need to propagate from WA and Q onto N T and SA, (as was done by forward checking) and then onto the constraint between N T and SA to detect the inconsistency. And we want to do this fast: it is no good reducing the amount of search if we spend more time propagating constraints than we would have spent doing a simple search.

CONSTRAINT PROPAGATION

ARC CONSISTENCY

The idea of **arc consistency** provides a fast method of constraint propagation that is substantially stronger than forward checking. Here, "arc" refers to a *directed* arc in the constraint graph, such as the arc from SA to NSW. Given the current domains of SA and NSW, the arc is consistent if, for *every* value x of SA, there is *some* value y of NSW that is consistent with x. In the third row of Figure 5.6, the current domains of SA and NSW are *{blue)* and *{red, blue)* respectively. For SA = *blue,* there is a consistent assignment for NSW, namely, NSW = *red;* therefore, the arc from SA to *NSW* is consistent. On the other hand, the reverse arc from NSW to SA is not consistent: for the assignment NSW = *blue,* there is no consistent assignment for SA. The arc can be made consistent by deleting the value *blue* from the domain of *NS*W.

We can also apply arc consistency to the arc from SA to NT at the same stage in the search process. The third row of the table in Figure 5.6 shows that both variables have the domain *{blue)*. The result is that *blue* must be deleted from the domain of SA, leaving the domain empty. Thus, applying arc consistency has resulted in early detection of an inconsis-

The second column of Figure 5.5, labeled BT+MRV, shows the performance of this heuristic. The performance is 3 to 3,000 times better than simple backtracking, depending on the problem. Note that our performance measure ignores the extra cost of computing the heuristic values; the next subsection describes a method that makes this cost manageable.

DEGREE HEURISTIC

The MRV heuristic doesn't help at all in choosing the first region to color in Australia, because initially every region has three legal colors. In this case, the **degree heuristic** comes in handy. It attempts to reduce the branching factor on future choices by selecting the variable that is involved in the largest number of constraints on other unassigned variables. In Figure 5.1, SA is the variable with highest degree, 5; the other variables have degree 2 or 3, except for T, which has 0. In fact, once SA is chosen, applying the degree heuristic solves the problem without any false steps—you can choose any consistent color at each choice point and still arrive at a solution with no backtracking. The minimum remaining values heuristic is usually a more powerful guide, but the degree heuristic can be useful as a tie-breaker.

LEAST-CONSTRAINING-VALUE

Once a variable has been selected, the algorithm must decide on the order in which to examine its values. For this, the **least-constraining-value** heuristic can be effective in some cases. It prefers the value that rules out the fewest choices for the neighboring variables in the constraint graph. For example, suppose that in Figure 5.1 we have generated the partial assignment with WA = red and NT = green, and that our next choice is for Q. Blue would be a bad choice, because it eliminates the last legal value left for Q's neighbor, SA. The least-constraining-value heuristic therefore prefers red to blue. In general, the heuristic is trying to leave the maximum flexibility for subsequent variable assignments. Of course, if we are trying to find all the solutions to a problem, not just the first one, then the ordering does not matter because we have to consider every value anyway. The same holds if there are no solutions to the problem.

Propagating information through constraints

So far our search algorithm considers the constraints on a variable only at the time that the variable is chosen by SELECT-UNASSIGNED-VARIABLE. But by looking at some of the constraints earlier in the search, or even before the search has started, we can drastically reduce the search space.

Forward checking

FORWARD CHECKING

One way to make better use of constraints during search is called **forward checking.** Whenever **a** variable X is assigned, the forward checking process looks at each unassigned variable Y that is connected to X by a constraint and deletes from Y's domain any value that is inconsistent with the value chosen for X. Figure 5.6 shows the progress of a map-coloring search with forward checking. There are two important points to notice about this example. First, notice that after assigning WA = red and Q = green, the domains of NT and SA are reduced to a single value; we have eliminated branching on these variables altogether by propagating information from WA and Q. The MRV heuristic, which is an obvious partner for forward checking, would automatically select *SA* and NT next. (Indeed, we can view forward checking as an efficient way to incrementally compute the information that the

Problem	Backtracking	BT+MRV	Forward Checking	FC+MRV	Min-Conflicts
USA	(> 1,000K)	(> 1,000K)	2K	60	64
n-Queens	(>40,000K)	13,500K	(> 40,000K)	817K	4K
Zebra	3,859K	1K	35K	0.5K	2K
Random 1	415K	3K	26K	2K	
Random 2	942K	27K	77K	15K	

Figure 5.5 Comparison of various CSP algorithms on various problems. The algorithms from left to right, are simple backtracking, backtracking with the MRV heuristic, forward checking, forward checking with MRV, and minimum conflicts local search. Listed in each cell is the median number of consistency checks (over five runs) required to solve the problem; note that all entries except the two in the upper right are in thousands (K). Numbers in parentheses mean that no answer was found in the allotted number of checks. The first problem is finding a 4-coloring for the 50 states of the United States of America. The remaining problems are taken from Bacchus and van Run (1995), Table 1. The second problem counts the total number of checks required to solve all n-Queens problems for n from 2 to 50. The third is the "Zebra Puzzle," as described in Exercise 5.13. The last two are artificial random problems. (Min-conflicts was not run on these.) The results suggest that forward checking with the MRV heuristic is better on all these problems than the other backtracking algorithms, but not always better than min-conflicts local search.

edge. Instead, we find general-purpose methods that address the following questions:

1. Which variable should be assigned next, and in what order should its values be tried?
2. What are the implications of the current variable assignments for the other unassigned variables?
3. When a path fails—that is, a state is reached in which a variable has no legal values—can the search avoid repeating this failure in subsequent paths?

The subsections that follow answer each of these questions in turn.

Variable and value ordering

The backtracking algorithm contains the line

$var \leftarrow$ SELECT-UNASSIGNED-VARIABLE(VARIABLES[*csp*], *assignment, csp*).

By default, SELECT-UNASSIGNED-VARIABLE simply selects the next unassigned variable in the order given by the list VARIABLES[*csp*]. This static variable ordering seldom results in the most efficient search. For example, after the assignments for $WA = red$ and $NT = green$, there is only one possible value for SA, so it makes sense to assign $SA = blue$ next rather than assigning Q. In fact, after SA is assigned, the choices for Q, NSW, and V are all forced. This intuitive idea—choosing the variable with the fewest "legal" values—is called the **minimum remaining values (MRV)** heuristic. It also has been called the "most constrained variable" or "fail-first" heuristic, the latter because it picks a variable that is most likely to cause a failure soon, thereby pruning the search tree. If there is a variable X with zero legal values remaining, the MRV heuristic will select X and failure will be detected immediately—avoiding pointless searches through other variables which always will fail when X is finally selected.

MINIMUM REMAINING VALUES

```
function BACKTRACKING-SEARCH(csp) returns a solution, or failure
  return RECURSIVE-BACKTRACKING({ }, csp)

function RECURSIVE-BACKTRACKING(assignment, csp) returns a solution, or failure
  if assignment is complete then return assignment
  var ← SELECT-UNASSIGNED-VARIABLE(VARIABLES[csp], assignment, csp)
  for each value in ORDER-DOMAIN-VALUES(var, assignment, csp) do
     if value is consistent with assignment according to CONSTRAINTS[csp] then
        add {var = value} to assignment
        result ← RECURSIVE-BACKTRACKING(assignment, csp)
        if result $\neq$ failure then return result
        remove {var = value} from assignment
  return failure
```

Figure 5.3 A simple backtracking algorithm for constraint satisfaction problems. The algorithm is modeled on the recursive depth-first search of Chapter 3. The functions SELECT-UNASSIGNED-VARIABLE and ORDER-DOMAIN-VALUES can be used to implement the general-purpose heuristics discussed in the text.

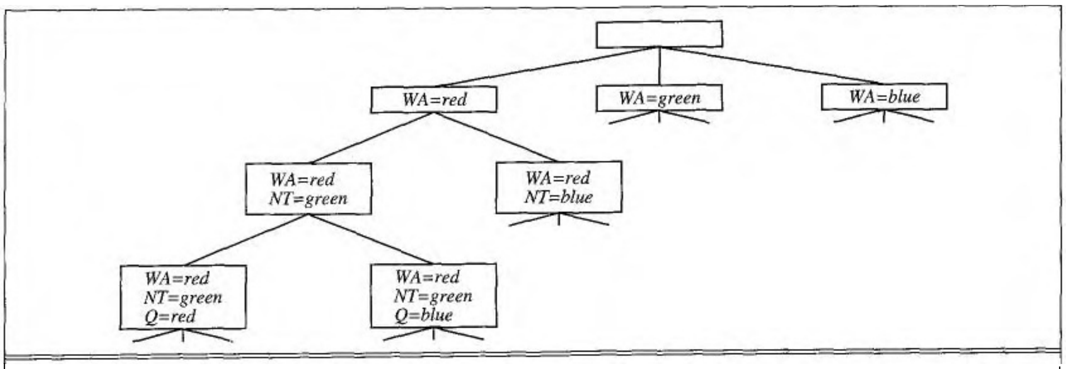


Figure 5.4 Part of the search tree generated by simple backtracking for the map-coloring problem in Figure 5.1.

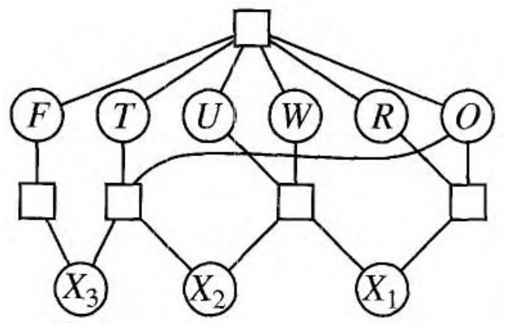
incremental successor generation described on page 76. Also, it extends the current assignment to generate a successor, rather than copying it. Because the representation of CSPs is standardized, there is no need to supply BACKTRACKING-SEARCH with a domain-specific initial state, successor function, or goal test. Part of the search tree for the Australia problem is shown in Figure 5.4, where we have assigned variables in the order WA, NT, Q, ...

Plain backtracking is an uninformed algorithm in the terminology of Chapter **3,** so we do not expect it to be very effective for large problems. The results for some sample problems are shown in the first column of Figure 5.5 and confirm our expectations.

In Chapter 4 we remedied the poor performance of uninformed search algorithms by supplying them with domain-specific heuristic functions derived from our knowledge of the problem. It turns out that we can solve CSPs efficiently without such domain-specific knowl-

$$\begin{array}{r} T\ W\ O \\ +\ T\ W\ O \\ \hline F\ O\ U\ R \end{array}$$

(a)



(b)

Figure 5.2 (a) A cryptarithmetic problem. Each letter stands for a distinct digit; the aim is to find a substitution of digits for letters such that the resulting sum is arithmetically correct, with the added restriction that no leading zeroes are alllowed. (b) The constraint hypergraph for the cryptarithmetic problem, showing the *Alldiff* constraint as well as the column addition constraints. Each constraint is a square box connected to the variables it constrains.

mization search methods, either path-based or local. We do not discuss such CSPs further in this chapter, but we provide some pointers in the bibliographical notes section.

5.2 BACKTRACKING SEARCH FOR CSPS

The preceding section gave a formulation of CSPs as search problems. Using this formulation, any of the search algorithms from Chapters 3 and 4 can solve CSPs. Suppose we apply breadth-first search to the generic CSP problem formulation given in the preceding section. We quickly notice something terrible: the branching factor at the top level is nd, because any of d values can be assigned to any of n variables. At the next level, the branching factor is $(n-1)d$, and so on for n levels. We generate a tree with $n! \cdot d^n$ leaves, even though there are only d^n possible complete assignments!

COMMUTATIVITY

Our seemingly reasonable but naïve problem formulation has ignored a crucial property common to all CSPs: **commutativity.** A problem is commutative if the order of application of any given set of actions has no effect on the outcome. This is the case for CSPs because, when assigning values to variables, we reach the same partial assignment, regardless of order. Therefore, *all* CSP *search algorithms generate successors by considering possible assignments for only a* single *variable at each node in the search tree.* For example, at the root node of a search tree for coloring the map of Australia, we might have a choice between $SA = red$, $SA = green$, and $SA = blue$, but we would never choose between $SA = red$ and $WA = blue$. With this restriction, the number of leaves is d^n, as we would hope.

BACKTRACKING SEARCH

The term **backtracking search** is used for a depth-first search that chooses values for one variable at a time and backtracks when a variable has no legal values left to assign. The algorithm is shown in Figure 5.3. Notice that it uses, in effect, the one-at-a-time method of of experiments on the Hubble Space Telescope requires very precise timing of observations; the start and finish of each observation and maneuver are continuous-valued variables that must obey a variety of astronomical, precedence, and power constraints. The best-known category of continuous-domain CSPs is that of **linear programming** problems, where constraints must be linear inequalities forming a convex region. Linear programming problems can be solved in time polynomial in the number of variables. Problems with different types of constraints and objective functions have also been studied—quadratic programming, second-order conic programming, and so on.

LINEAR PROGRAMMING

In addition to examining the types of variables that can appear in CSPs, it is useful to look at the types of constraints. The simplest type is the **unary constraint,** which restricts the value of a single variable. For example, it could be the case that South Australians actively dislike the color green. Every unary constraint can be eliminated simply by preprocessing the domain of the corresponding variable to remove any value that violates the constraint. A **binary constraint** relates two variables. For example, $SA \neq NSW$ is a binary constraint. A binary CSP is one with only binary constraints; it can be represented as a constraint graph, as in Figure 5.1(b).

UNARY CONSTRAINT

BINARY CONSTRAINT

Higher-order constraints involve three or more variables. A familiar example is provided by **cryptarithmetic** puzzles. (See Figure 5.2(a).) It is usual to insist that each letter in a cryptarithmetic puzzle represent a different digit. For the case in Figure 5.2(a)), this would be represented as the six-variable constraint $Alldiff(F, T, U, W, R, O)$. Alternatively, it can be represented by a collection of binary constraints such as $F \neq T$. The addition constraints on the four columns of the puzzle also involve several variables and can be written as

CRYPTARITHMETIC

$$
\begin{aligned}
&O + O = R + 10 \cdot X_1\\
&X_1 + W + W = U + 10 \cdot X_2\\
&X_2 + T + T = O + 10 \cdot X_3\\
&X_3 = F
\end{aligned}
$$

where X_1, X_2, and X_3 are **auxiliary variables** representing the digit (0 or 1) carried over into the next column. Higher-order constraints can be represented in a **constraint hypergraph,** such as the one shown in Figure 5.2(b). The sharp-eyed reader will have noticed that the *Alldiff* constraint can be broken down into binary constraints—$F \neq T$, F # U, and so on. In fact, as Exercise 5.11 asks you to prove, every higher-order, finite-domain constraint can be reduced to a set of binary constraints if enough auxiliary variables are introduced. Because of this, we will deal only with binary constraints in this chapter.

AUXILIARY VARIABLES

CONSTRAINT HYPERGRAPH

The constraints we have described so far have all been **absolute** constraints, violation of which rules out a potential solution. Many real-world CSPs include **preference** constraints indicating which solutions are preferred. For example, in a university timetabling problem, Prof. X might prefer teaching in the morning whereas Prof. Y prefers teaching in the afternoon. A timetable that has Prof. X teaching at 2 p.m. would still be a solution (unless Prof. X happens to be the department chair), but would not be an optimal one. Preference constraints can often be encoded as costs on individual variable assignments—for example, assigning an afternoon slot for Prof. X costs 2 points against the overall objective function, whereas a morning slot costs 1. With this formulation, CSPs with preferences can be solved using opti-

PREFERENCE

It is fairly easy to see that a CSP can be given an **incremental formulation** as a standard search problem as follows:

◇ **Initial state:** the empty assignment $\{\}$, in which all variables are unassigned.

◇ **Successor function:** a value can be assigned to any unassigned variable, provided that it does not conflict with previously assigned variables.

◇ **Goal test:** the current assignment is complete.

◇ **Path cost:** a constant cost (e.g., 1) for every step.

Every solution must be a complete assignment and therefore appears at depth n if there are n variables. Furthermore, the search tree extends only to depth n. For these reasons, depth-first search algorithms are popular for CSPs. (See Section 5.2.) It is also the case that *the path* by *which a solution is reached is irrelevant.* Hence, we can also use a **complete-state formulation,** in which every state is a complete assignment that might or might not satisfy the constraints. Local search methods work well for this formulation. (See Section 5.3.)

FINITE DOMAINS

The simplest kind of CSP involves variables that are **discrete** and have **finite domains.** Map-coloring problems are of this kind. The 8-queens problem described in Chapter 3 can also be viewed as a finite-domain CSP, where the variables $Q_1, \ldots, Q_8$ are the positions of each queen in columns $1, \ldots, 8$ and each variable has the domain $\{1, 2, 3, 4, 5, 6, 7, 8\}$. If the maximum domain size of any variable in a CSP is d, then the number of possible complete assignments is $O(d^n)$—that is, exponential in the number of variables. Finite-domain CSPs include **Boolean CSPs,** whose variables can be either *true* or *false.* Boolean CSPs include as special cases some NP-complete problems, such as 3SAT. (See Chapter 7.) In the worst case, therefore, we cannot expect to solve finite-domain CSPs in less than exponential time. In most practical applications, however, general-purpose CSP algorithms can solve problems *orders of magnitude* larger than those solvable via the general-purpose search algorithms that we saw in Chapter 3.

BOOLEAN CSPS

INFINITE DOMAINS

Discrete variables can also have **infinite domains**—for example, the set of integers or the set of strings. For example, when scheduling construction jobs onto a calendar, each job's start date is a variable and the possible values are integer numbers of days from the current date. With infinite domains, it is no longer possible to describe constraints by enumerating all allowed combinations of values. Instead, a **constraint language** must be used. For example, if Job_1, which takes five days, must precede Job_3, then we would need a constraint language of algebraic inequalities such as $StartJob_1 + 5 \leq StartJob_3$. It is also no longer possible to solve such constraints by enumerating all possible assignments, because there are infinitely many of them. Special solution algorithms (which we will not discuss here) exist for **linear constraints** on integer variables—that is, constraints, such as the one just given, in which each variable appears only in linear form. It can be shown that no algorithm exists for solving general **nonlinear constraints** on integer variables. In some cases, we can reduce integer constraint problems to finite-domain problems simply by bounding the values of all the variables. For example, in a scheduling problem, we can set an upper bound equal to the total length of all the jobs to be scheduled.

CONSTRAINT LANGUAGE

LINEAR CONSTRAINTS

NONLINEAR CONSTRAINTS

CONTINUOUS DOMAINS

Constraint satisfaction problems with **continuous domains** are very common in the real world and are widely studied in the field of operations research. For example, the scheduling

So what does all this mean? Suppose that, having tired of Romania, we are looking at a map of Australia showing each of its states and territories, as in Figure 5.1(a), and that we are given the task of coloring each region either red, green, or blue in such a way that no neighboring regions have the same color. To formulate this as a CSP, we define the variables to be the regions: *WA,* NT, Q, *NSW, V, SA,* and *T*. The domain of each variable is the set *{red, green, blue).* The constraints require neighboring regions to have distinct colors; for example, the allowable combinations for *WA* and *NT* are the pairs

$$\{(red,green),(red,blue),(green,red),(green,blue),(blue,red),(blue,green))\,.$$

(The constraint can also be represented more succinctly as the inequality $WA \neq NT$, provided the constraint satisfaction algorithm has some way to evaluate such expressions.) There are many possible solutions, such as

$$\{WA = red, NT = green, Q = red, NSW = green, V = red, SA = blue, T = red\}.$$

CONSTRAINT GRAPH It is helpful to visualize a CSP as a constraint graph, as shown in Figure 5.1(b). The nodes of the graph correspond to variables of the problem and the arcs correspond to constraints.

Treating a problem as a CSP confers several important benefits. Because the representation of states in a CSP conforms to a standard pattern—that is, a set of variables with assigned values—the successor function and goal test can be written in a generic way that applies to all CSPs. Furthermore, we can develop effective, generic heuristics that require no additional, domain-specific expertise. Finally, the structure of the constraint graph can be used to simplify the solution process, in some cases giving an exponential reduction in complexity. The CSP representation is the first, and simplest, in a series of representation schemes that will be developed throughout the book.

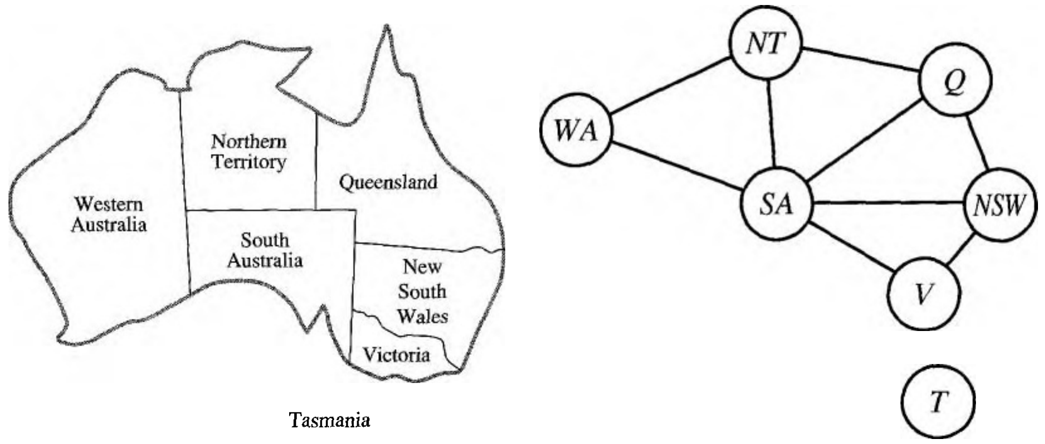


Figure 5.1 (a) The principal states and territories of Australia. Coloring this map can be viewed as a constraint satisfaction problem. The goal is to assign colors to each region so that no neighboring regions have the same color. (b) The map-coloring problem represented as a constraint graph.

5 CONSTRAINT SATISFACTION PROBLEMS

In which we see how treating states as more than just little black boxes leads to the invention of a range of powerful new search methods and a deeper understanding of problem structure and complexity.

Chapters 3 and 4 explored the idea that problems can be solved by searching in a space of **states.** These states can be evaluated by domain-specific heuristics and tested to see whether they are goal states. From the point of view of the search algorithm, however, each state is a **black box** with no discernible internal structure. It is represented by an arbitrary data structure that can be accessed only by the *problem-,specific* routines —the successor function, heuristic function, and goal test.

BLACK BOX

This chapter examines **constraint satisfaction problems**, whose states and goal test conform to a standard, structured, and very simple **representation** (Section 5.1). Search algorithms can be defined that take advantage of the structure of states and use *general-purpose* rather than *problem-specific* heuristics to enable the solution of large problems (Sections 5.2–5.3). Perhaps most importantly, the standard representation of the goal test reveals the structure of the problem itself (Section 5.4). This leads to methods for problem decomposition and to an understanding of the intimate connection between the structure of a problem and the difficulty of solving it.

REPRESENTATION

5.1 CONSTRAINT SATISFACTION PROBLEMS

CONSTRAINT SATISFACTION PROBLEM

VARIABLES

CONSTRAINTS

DOMAIN

VALUES

ASSIGNMENT

CONSISTENT

OBJECTIVE FUNCTION

Formally speaking, a **constraint satisfaction problem** (or **CSP)** is defined by a set of **variables,** $X_1, X_2, \ldots, X_n$, and a set of **constraints,** $C_1, C_2, \ldots, C_m$. Each variable X_i has a nonempty **domain** D_i of possible **values.** Each constraint C_i involves some subset of the variables and specifies the allowable combinations of values for that subset. A state of the problem is defined by an **assignment** of values to some or all of the variables, $\{X_i = v_i, X_j = v_j, \ldots)$. An assignment that does not violate any constraints is called a **consistent** or legal assignment. A complete assignment is one in which every variable is mentioned, and a **solution** to a CSP is a complete assignment that satisfies all the constraints. Some CSPs also require a solution that maximizes an **objective function.**

four actions Up, *Down*, *Left*, Right have their usual effects unless blocked by a wall. The agent does *not* know where the internal walls are. In any given state, the agent perceives the set of legal actions; it can also tell whether the state is one it has visited before or a new state.

a. Explain how this online search problem can be viewed as an offline search in belief state space, where the initial belief state includes all possible environment configurations. How large is the initial belief state? How large is the space of belief states?

b. How many distinct percepts are possible in the initial state?

c. Describe the first few branches of a contingency plan for this problem. How large (roughly) is the complete plan?

Notice that this contingency plan is a solution for *every possible environment* fitting the given description. Therefore, interleaving of search and execution is not strictly necessary even in unknown environments.

4.15 In this exercise, we will explore the use of local search methods to solve TSPs of the type defined in Exercise 4.8.

a. Devise a hill-climbing approach to solve TSPs. Compare the results with optimal solutions obtained via the A* algorithm with the MST heuristic (Exercise 4.8).

b. Devise a genetic algorithm approach to the traveling salesperson problem. Compare results to the other approaches. You may want to consult Larrañaga *et al.* (1999) for some suggestions for representations.

4.16 Generate a large number of 8-puzzle and 8-queens instances and solve them (where possible) by hill climbing (steepest-ascent and first-choice variants), hill climbing with random restart, and simulated annealing. Measure the search cost and percentage of solved problems and graph these against the optimal solution cost. Comment on your results.

4.17 In this exercise, we will examine hill climbing in the context of robot navigation, using the environment in Figure 3.22 as an example.

a. Repeat Exercise 3.16 using hill climbing. Does your agent ever get stuck in a local minimum? Is it *possible* for it to get stuck with convex obstacles?

b. Construct a nonconvex polygonal environment in which the agent gets stuck.

c. Modify the hill-climbing algorithm so that, instead of doing a depth-1 search to decide where to go next, it does a depth-k search. It should find the best k-step path and do one step along it, and then repeat the process.

d. Is there some k for which the new algorithm is guaranteed to escape from local minima?

e. Explain how LRTA* enables the agent to escape from local minima in this case.

4.18 Compare the performance of A* and RBFS on a set of randomly generated problems in the 8-puzzle (with Manhattan distance) and TSP (with MST—see Exercise 4.8) domains. Discuss your results. What happens to the performance of RBFS when a small random number is added to the heuristic values in the 8-puzzle domain?

4.6 Invent a heuristic function for the 8-puzzle that sometirnes overestimates, and show how it can lead to a suboptimal solution on a particular problem. (You can use a computer to help if you want.) Prove that, if h never overestimates by more than c, A* using h returns a solution whose cost exceeds that of the optimal solution by no more than c.

4.7 Prove that if a heuristic is consistent, it must be admissible. Construct an admissible heuristic that is not consistent.

4.8 The traveling salesperson problem (TSP) can be solved via the minimum spanning tree (MST) heuristic, which is used to estimate the cost of completing a tour, given that a partial tour has already been constructed. The MST cost of a set of cities is the smallest sum of the link costs of any tree that connects all the cities.

a. Show how this heuristic can be derived from a relaxed version of the TSP.

b. Show that the MST heuristic dominates straight-:linedistance.

c. Write a problem generator for instances of the TSP where cities are represented by random points in the unit square.

d. Find an efficient algorithm in the literature for constructing the MST, and use it with an admissible search algorithm to solve instances of the TSP.

4.9 On page 108, we defined the relaxation of the 8-puzzle in which a tile can move from square A to square B if B is blank. The exact solution of this problem defines **Gaschnig's heuristic** (Gaschnig, 1979). Explain why Gaschnig's heuristic is at least as accurate as h_1 (misplaced tiles), and show cases where it is more accurate than both h_1 and h_2 (Manhattan distance). Can you suggest a way to calculate Gaschnig's heuristic efficiently?

4.10 We gave two simple heuristics for the 8-puzzle:: Manhattan distance and misplaced tiles. Several heuristics in the literature purport to improve on this—see, for example, Nilsson (1971), Mostow and Prieditis (1989), and Hansson *et al.* (1992). Test these claims by implementing the heuristics and comparing the performance of the resulting algorithms.

4.11 Give the name of the algorithm that results from each of the following special cases:

a. Local beam search with $k = 1$.

b. Local beam search with one initial state and no limit on the number of states retained.

c. Simulated annealing with T = 0 at all times (and omitting the termination test).

d. Genetic algorithm with population size N = 1.

4.12 Sometimes there is no good evaluation function for a problem, but there is a good comparison method: a way to tell whether one node is better than another, without assigning numerical values to either. Show that this is enough to do a best-first search. Is there an analog of A*?

4.13 Relate the time complexity of LRTA* to its space complexity.

4.14 Suppose that an agent is in a **3** x **3** maze environment like the one shown in Figure 4.18. The agent knows that its initial location is (1,1), that the goal is at (3,3), and that the a small competitive ratio is achievable with square obstacles, but with general rectangular obstacles no bounded ratio can be achieved. (See Figure 4.19.)

REAL-TIME SEARCH

The LRTA* algorithm was developed by Korf (1990) as part of an investigation into **real-time search** for environments in which the agent must act after searching for only a fixed amount of time (a much more common situation in two-player games). LRTA* is in fact a special case of reinforcement learning algorithms for stochastic environments (Barto *et* al., 1995). Its policy of optimism under uncertainty—always head for the closest unvisited state--can result in an exploration pattern that is less efficient in the uninformed case than simple depth-first search (Koenig, 2000). Dasgupta *et al.* (1994) show that online iterative deepening search is optimally efficient for finding a goal in a uniform tree with no heuristic information. Several informed variants on the LRTA* theme have been developed with different methods for searching and updating within the known portion of the graph (Pemberton and Korf, 1992). As yet, there is no good understanding of how to find goals with optimal efficiency when using heuristic information.

PARALLEL SEARCH

The topic of **parallel search** algorithms was not covered in the chapter, partly because it requires a lengthy discussion of parallel computer architectures. Parallel search is becoming an important topic in both AI and theoretical computer science. A brief introduction to the AI literature can be found in Mahanti and Daniels (1993).

EXERCISES

4.1 Trace the operation of A* search applied to the problem of getting to Bucharest from Lugoj using the straight-line distance heuristic. That is, show the sequence of nodes that the algorithm will consider and the f, g, and h score for each node.

4.2 The **heuristic path algorithm** is a best-first search in which the objective function is $f(n) = (2 - w)g(n) + wh(n)$. For what values of w is this algorithm guaranteed to be optimal? (You may assume that h is admissible.) What kind of search does this perform when w = 0? When w = 1? When w = 2?

4.3 Prove each of the following statements:

a. Breadth-first search is a special case of uniform-cost search.

b. Breadth-first search, depth-first search, and uniform-cost search are special cases of best-first search.

c. Uniform-cost search is a special case of A* search.

4.4 Devise a state space in which A* using GRAPH-SEARCH returns a suboptimal solution with an $h(n)$ function that is admissible but inconsistent.

4.5 We saw on page 96 that the straight-line distance heuristic leads greedy best-first search astray on the problem of going from Iasi to Fagaras. However, the heuristic is perfect on the opposite problem: going from Fagaras to Iasi. Are there problems for which the heuristic is misleading in both directions?

as a method to expand our understanding of adaptation, biological or otherwise (Holland, 1995). The **artificial life** movement (Langton, 1995) takes this idea one step further, viewing the products of genetic algorithms as *organisms* rather than solutions to problems. Work in this field by Hinton and Nowlan (1987) and Ackley and Littman (1991) has done much to clarify the implications of the Baldwin effect. For general background on evolution, we strongly recommend Smith and Szathmáry (1999).

ARTIFICIAL LIFE

Most comparisons of genetic algorithms to other approaches (especially stochastic hill-climbing) have found that the genetic algorithms are slower to converge (O'Reilly and Oppacher, 1994; Mitchell *et al.,* 1996; Juels and Wattenberg, 1996; Baluja, 1997). Such findings are not universally popular within the GA community, but recent attempts within that community to understand population-based search as an approximate form of Bayesian learning (see Chapter 20) might help to close the gap between the field and its critics (Pelikan *et al.,* 1999). The theory of **quadratic dynamical systems** may also explain the performance of GAs (Rabani *et al.*, 1998). See Lohn *et al.* (2001) for an example of GAs applied to antenna design, and Larrañaga *et al.* (1999) for an application to the traveling salesperson problem.

GENETIC PROGRAMMING

The field of **genetic programming** is closely related to genetic algorithms. The principal difference is that the representations that are mutated and combined are programs rather than bit strings. The programs are represented in the form of expression trees; the expressions can be in a standard language such as Lisp or can be specialty designed to represent circuits, robot controllers, and so on. Crossover involves splicing together subtrees rather than substrings. This form of mutation guarantees that the offspring are well-formed expressions, which would not be the case if programs were manipulated as strings.

Recent interest in genetic programming was spurred by John Koza's work (Koza, 1992, 1994), but it goes back at least to early experiments with machine code by Friedberg (1958) and with finite-state automata by Fogel *et al.* (1966). As with genetic algorithms, there is debate about the effectiveness of the technique. Koza *et al.* (1999) describe a variety of experiments on the automated design of circuit devices using genetic programming.

The journals *Evolutionary Computation* and *IEEE Transactions on Evolutionary Computation* cover genetic algorithms and genetic programming; articles are also found in *Complex Systems, Adaptive Behavior,* and *Artificial Life*. 'The main conferences are the *International Conference on Genetic Algorithms* and the *Conference on Genetic Programming,* recently merged to form the *Genetic and Evolutionary Computation Conference.* The texts by Melanie Mitchell (1996) and David Fogel (2000) give good overviews of the field.

Algorithms for exploring unknown state spaces have been of interest for many centuries. Depth-first search in a maze can be implemented by keeping one's left hand on the wall; loops can be avoided by marking each junction. Depth-first search fails with irreversible actions; the more general problem of exploring of **Eulerian graphs** (i.e., graphs in which each node has equal numbers of incoming and outgoing edges) was solved by an algorithm due to Hierholzer (1873). The first thorough algorithmic study of the exploration problem for arbitrary graphs was carried out by Deng and Papadimitriou (1990), who developed a completely general algorithm, but showed that no bounded competitive ratio is possible for exploring a general graph. Papadimitriou and Yannakakis (1991) examined the question of finding paths to a goal in geometric path-planning environments (where all actions are reversible). They showed that

EULERIANGRAPHS

Local-search techniques have a long history in mathematics and computer science. Indeed, the Newton–Raphson method (Newton, 1671; Raphson, 1690) can be seen as a very efficient local-search method for continuous spaces in which gradient information is available. Brent (1973) is a classic reference for optimization algorithms that do not require such information. Beam search, which we have presented as a local-search algorithm, originated as a bounded-width variant of dynamic programming for speech recognition in the HARPY system (Lowerre, 1976). A related algorithm is analyzed in depth by Pearl (1984, Ch. 5).

The topic of local search has been reinvigorated in recent years by surprisingly good results for large constraint satisfaction problems such as n-queens (Minton *et* al., 1992) and logical reasoning (Selman *et* al., 1992) and by the incorporation of randomness, multiple simultaneous searches, and other improvements. This renaissance of what Christos Papadimitriou has called "New Age" algorithms has also sparked increased interest among theoretical computer scientists (Koutsoupias and Papadimitriou, 1992; Aldous and Vazirani, 1994).

TABU SEARCH

In the field of operations research, a variant of hill climbing called **tabu search** has gained popularity (Glover, 1989; Glover and Laguna, 1997). Drawing on models of limited short-term memory in humans, this algorithm maintains a tabu list of k previously visited states that cannot be revisited; as well as improving efficiency when searching graphs, this can allow the algorithm to escape from some local minima. Another useful improvement on hill climbing is the STAGE algorithm (Boyan and Moore, 1998). The idea is to use the local maxima found by random-restart hill climbing to get an idea of the overall shape of the landscape. The algorithm fits a smooth surface to the set of local maxima and then calculates the global maximum of that surface analytically. This becomes the new restart point. The algorithm has been shown to work in practice on hard problems. (Gomes *et* al., 1998) showed that the run time distributions of systematic backtracking algorithms often have a **heavy-tailed distribution,** which means that the probability of a very long run time is more than would be predicted if the run times were normally distributed. This provides a theoretical justification for random restarts.

HEAVY-TAILED DISTRIBUTION

Simulated annealing was first described by Kirkpatrick *et* al. (1983), who borrowed directly from the **Metropolis algorithm** (which is used to simulate complex systems in physics (Metropolis *et al.,* 1953) and was supposedly invented at a Los Alamos dinner party). Simulated annealing is now a field in itself, with hundreds of papers published every year.

Finding optimal solutions in continuous spaces is the subject matter of several fields, including **optimization theory, optimal control theory,** and the **calculus of variations.** Suitable (and practical) entry points are provided by Press *et* al. (2002) and Bishop (1995). **Linear programming** (LP) was one of the first applications of computers; the **simplex algorithm** (Wood and Dantzig, 1949; Dantzig, 1949) is still used despite worst-case exponential complexity. Karmarkar (1984) developed a practical polynomial-time algorithm for LP.

EVOLUTION STRATEGIES

Work by Sewall Wright (1931) on the concept of a **fitness landscape** was an important precursor to the development of genetic algorithms. In the 1950s, several statisticians, including Box (1957) and Friedman (1959), used evolutionary techniques for optimization problems, but it wasn't until Rechenberg (1965, 1973) introduced **evolution strategies** to solve optimization problems for airfoils that the approach gained popularity. In the 1960s and 1970s, John Holland (1975) championed genetic algorithms, both as a useful tool and

A* arid other state-space search algorithms are closely related to the *branch-and-bound* techniques that are widely used in operations research (Lawler and Wood, 1966). The relationships between state-space search and branch-and-bound have been investigated in depth (Kumar and Kanal, 1983; Nau *et al.,* 1984; Kumar *et al.*, 1988). Martelli and Montanari (1978) demonstrate a connection between dynamic programming (see Chapter 17) and certain types of state-space search. Kumar and Kanal (1988) attempt a "grand unification" of heuristic search, dynamic programming, and branch-and-bound techniques under the name of CDP—the "composite decision process."

Because computers in the late 1950s and early 1960s had at most a few thousand words of main memory, mernory-bounded heuristic search was an early research topic. The Graph Traverser (Doran and Michie, 1966), one of the earliest search programs, commits to an operator after searching best first up to the memory limit. IDA* (Korf, 1985a, 1985b) was the first widely used optimal, memory-bounded, heuristic search algorithm, and a large number of variants have been developed. An analysis of the efficiency of IDA* and of its difficulties with real-valued heuristics appears in Patrick *et al.* (1992).

RBFS (Korf, 1991, 1993) is actually somewhat more complicated than the algorithm shown in Figure 4.5, which is closer to an independently developed algorithm called **iterative expansion,** or IE (Russell, 1992). RBFS uses a lower bound as well as the upper bound; the two algorithms behave identically with admissible heuristics, but RBFS expands nodes in best-first order even with an inadmissible heuristic. The idea of keeping track of the best alternative path appeared earlier in Bratko's (1986) elegant Yrolog implementation of A* and in the DTA* algorithm (Russell and Wefald, 1991). The latter work also discusses metalevel state spaces and metalevel learning.

ITERATIVE EXPANSION

The MA* algorithm appeared in Chakrabarti *et al.* (1989). SMA*, or Simplified MA*, emerged from an attempt:to implement MA* as a comparison algorithm for IE (Russell, 1992). Kaindl and Khorsand (1994) have applied SMA* to produce a bidirectional search algorithm that is substantially faster than previous algorithms. Korf and Zhang (2000) describe a divide-and-conquer approach, and Zhou and Hansen (2002) introduce memory-bounded A* graph search. Korf (1995) surveys memory-bounded search techniques.

The idea that admissible heuristics can be derived by problem relaxation appears in the seminal paper by Held and Karp (1970), who used the the minimum-spanning-tree heuristic to solve the TSP. (See Exercise 4.8.)

The automation of the relaxation process was implemented successfully by Prieditis (1993), building on earlier work with Mostow (Mostow and Prieditis, 1989). The use of pattern databases to derive admissible heuristics is due to Gasser (1995) and Culberson and Schaeffer (1998); disjoint pattern databases are described by Korf and Felner (2002). The probabilistic interpretation of heuristics was investigated in depth by Pearl (1984) and Hansson and Mayer (1989).

By far the most comprehensive source on heuristics and heuristic search algorithms is Pearl's (1984) *Heuristics* text. This book provides especially good coverage of the wide variety of offshoots and variations of A*, including rigorous proofs of their formal properties. Kanal and Kumar (1988) present an anthology of important articles on heuristic search. New results on search algorithms appear regularly in the journal *Artificial Intelligence.*

- **A genetic algorithm** is a stochastic hill-climbing search in which a large population of states is maintained. New states are generated by **mutation** and by **crossover,** which combines pairs of states from the population.
- **Exploration problems** arise when the agent has no idea about the states and actions of its environment. For safely explorable environments, **online search** agents can build a map and find a goal if one exists. Updating heuristic estimates from experience provides an effective method to escape from local minima.

BIBLIOGRAPHICAL AND HISTORICAL NOTES

The use of heuristic information in problem solving appears in an early paper by Simon and Newell (1958), but the phrase "heuristic search" and the use of heuristic functions that estimate the distance to the goal came somewhat later (Newell and Ernst, 1965; Lin, 1965). Doran and Michie (1966) conducted extensive experimental studies of heuristic search as applied to a number of problems, especially the 8-puzzle and the 15-puzzle. Although Doran and Michie carried out theoretical analyses of path length and "penetrance" (the ratio of path length to the total number of nodes examined so far) in heuristic search, they appear to have ignored the information provided by current path length. The A* algorithm, incorporating the current path length into heuristic search, was developed by Hart, Nilsson, and Raphael (1968), with some later corrections (Hart *et al.*, 1972). Dechter and Pearl (1985) demonstrated the optimal efficiency of A*.

The original A* paper introduced the consistency condition on heuristic functions. The monotone condition was introduced by Pohl (1977) as a simpler replacement, but Pearl (1984) showed that the two were equivalent. A number of algorithms predating A* used the equivalent of open and closed lists; these include breadth-first, depth-first, and uniform-cost search (Bellman, 1957; Dijkstra, 1959). Bellman's work in particular showed the importance of additive path costs in simplifying optimization algorithms.

Pohl (1970, 1977) pioneered the study of the relationship between the error in heuristic functions and the time complexity of A*. The proof that A* runs in linear time if the error in the heuristic function is bounded by a constant can be found in Pohl (1977) and in Gaschnig (1979). Pearl (1984) strengthened this result to allow a logarithmic growth in the error. The "effective branching factor" measure of the efficiency of heuristic search was proposed by Nilsson (1971).

There are many variations on the A* algorithm. Pohl (1973) proposed the use of ***dynamic weighting,*** which uses a weighted sum $f_w(n) = w_g g(n) + w_h h(n)$ of the current path length and the heuristic function as an evaluation function, rather than the simple sum $f(n) = g(n) + h(n)$ used in A*. The weights w_g and w_h are adjusted dynamically as the search progresses. Pohl's algorithm can be shown to be ϵ-admissible—that is, guaranteed to find solutions within a factor $1 + \epsilon$ of the optimal solution—where ϵ is a parameter supplied to the algorithm. The same property is exhibited by the A^*_ϵ algorithm (Pearl, 1984), which can select any node from the fringe provided its f-cost is within a factor $1 + \epsilon$ of the lowest-f-cost fringe node. The selection can be done so as to minimize search cost.

converge to *exact* values for every state, provided that the agent explores the state space in the right way. Once exact values are known, optimal decisions can be taken simply by moving to the highest-valued successor—that is, pure hill climbing is then an optimal strategy.

If you followed our suggestion to trace the behavior of ONLINE-DFS-AGENT in the environment of Figure 4.18, you will have noticed that the agent is not very bright. For example, after it has seen that the Up action goes from (1,1) to (1,2), the agent still has no idea that the *Down* action goes back to (1,1), or that the *Up* action also goes from (2,1) to (2,2), from (2,2) to (2,3), and so on. In general, we would like the agent to learn that Up increases the y-coordinate unless there is a wall in the way, that Down reduces it, and so on. For this to happen, we need two things. First, we need a formal and explicitly manipulable representation for these kinds of general rules; so far, we have hidden the information inside the black box called the successor function. Part III is devoted to this issue. Second, we need algorithms that can construct suitable general rules from the specific observations made by the agent. These are covered in Chapter 18.

4.6 SUMMARY

This chapter has examined the application of **heuristics** to reduce search costs. We have looked at a number of algorithms that use heuristics and found that optimality comes at a stiff price in terms of search cost, even with good heuristics.

- **Best-first search** is just GRAPH-SEARCH where the minimum-cost unexpanded nodes (according to some measure) are selected for expansion. Best-first algorithms typically use a **heuristic** function $h(n)$ that estimates the cost of a solution from n.
- **Greedy best-first search** expands nodes with minimal $h(n)$. It is not optimal, but is often efficient.
- **A* search** expands nodes with minimal $f(n) = g(n) + h(n)$. A* is complete and optimal, provided that we guarantee that $h(n)$ is admissible (for TREE-SEARCH) or consistent (for GRAPH-SEARCH). The space complexity of A* is still prohibitive.
- The performance of heuristic search algorithms depends on the quality of the heuristic function. Good heuristics can sometimes be constructed by relaxing the problem definition, by precomputing solution costs for subproblems in a pattern database, or by learning from experience with the problem class.
- RBFS and SMA* are robust, optimal search algorithms that use limited amounts of memory; given enough time, they can solve problems that A* cannot solve because it runs out of memory.
- Local search methods such as **hill climbing** operate on complete-state formulations, keeping only a small number of nodes in memory. Several stochastic algorithms have been developed, including **simulated annealing,** which returns optimal solutions when given an appropriate cooling schedule. Many local search methods can also be used to solve problems in continuous spaces.

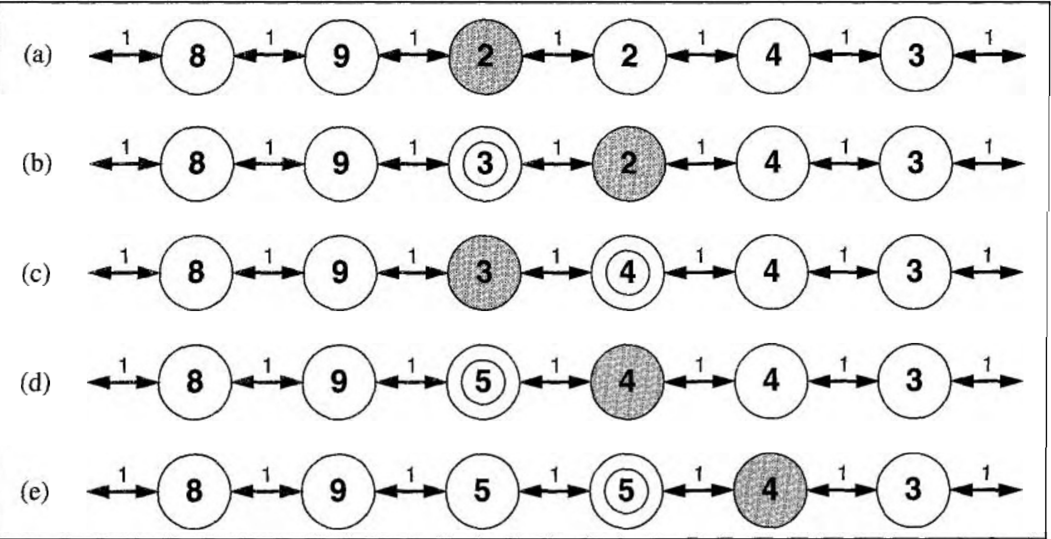


Figure 4.22 Five iterations of *LRTA** on a one-dimensional state space. Each state is labeled with $H(s)$, the current cost estimate to reach a goal, and each arc is labeled with its step cost. The shaded state marks the location of the agent, and the updated values at each iteration are circled.

```
function LRTA*-AGENT($s'$) returns an action
  inputs: $s'$, a percept that identifies the current state
  static: $result$, a table, indexed by action and state, initially empty
          $H$, a table of cost estimates indexed by state, initially empty
          $s$, $a$, the previous state and action, initially null

  if GOAL-TEST($s'$) then return stop
  if $s'$ is a new state (not in $H$) then $H[s'] \leftarrow h(s')$
  unless $s$ is null
     $result[a, s] \leftarrow s'$
     $H[s] \leftarrow \min_{b \in \text{ACTIONS}(s)}$ LRTA*-COST($s, b, result[b, s], H$)
  a ← an action $b$ in ACTIONS($s'$) that minimizes LRTA*-COST($s', b, result[b, s'], H$)
  $s \leftarrow s'$
  return $a$

function LRTA*-COST($s$, a, $s'$, $H$) returns a cost estimate
  if $s'$ is undefined then return $h(s)$
  else return $c(s, a, s') + H[s']$
```

Figure 4.23 LRTA*-AGENT selects an action according to the values of neighboring states, which are updated as the agent moves about the state space.

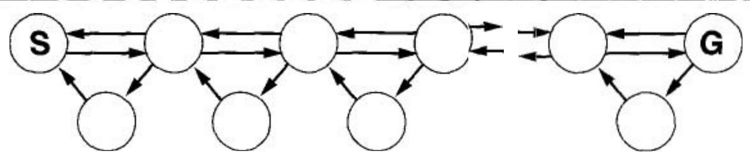


Figure 4.21 An environment in which a random walk will take exponentially many steps to find the goal.

estimate $h(s)$ and is updated as the agent gains experience in the state space. Figure 4.22 shows a simple example in a one-dimensional state space. In (a), the agent seems to be stuck in a flat local minimum at the shaded state. Rather than staying where it is, the agent should follow what seems to be the best path to the goal based on the current cost estimates for its neighbors. The estimated cost to reach the goal through a neighbor s is the cost to get to s plus the estimated cost to get to a goal from there—that is, $c(s, a, s) + H(s')$. In the example, there are two actions with estimated costs $1 + 9$ and $1 + 2$, so it seems best to move right. Now, it is clear that the cost estimate of 2 for the shaded state was overly optimistic. Since the best move cost 1 and led to a state that is at least 2 steps from a goal, the shaded state must be at least 3 steps from a goal, so its H should be updated accordingly, as shown in Figure 4.22(b). Continuing this process, the agent will move back and forth twice more, updating H each time and "flatteningout" the local minimum until it escapes to the right.

LRTA*

An agent implementing this scheme, which is called learning real-time A* (LRTA*),is shown in Figure 4.23. Like ONLINE-DFS-AGENT, it builds a map of the environment using the *result* table. It updates the cost estimate for the state it has just left and then chooses the "apparently best" move according to its current cost estimates. One important detail is that actions that have not yet been tried in a state s are always assumed to lead immediately to the goal with the least possible cost, namely $h(s)$. This **optimism under uncertainty** encourages the agent to explore new, possibly promising paths.

OPTIMISM UNDER UNCERTAINTY

An LRTA* agent is guaranteed to find a goal in any finite, safely explorable environment. Unlike A*, however, it is not complete for infinite state spaces—there are cases where it can be led infinitely astray. It can explore an environment of n states in $O(n^2)$ steps in the worst case, but often does much better. The LRTA* agent is just one of a large family of online agents that can be defined by specifying the action selection rule and the update rule in different ways. We will discuss this family, which was developed originally for stochastic environments, in Chapter 21.

Learning in online search

The initial ignorance of online search agents provides several opportunities for learning. First, the agents learn a "map" of the environment—more precisely, the outcome of each action in each state—simply by recording each of their experiences. (Notice that the assumption of deterministic environments means that one experience is enough for each action.) Second, the local search agents acquire more accurate estimates of the value of each state by using local updating rules, as in LRTA*. In Chapter 21 we will see that these updates eventually

```
function ONLINE-DFS-AGENT($s'$) returns an action
  inputs: s', a percept that identifies the current state
  static: $result$, a table, indexed by action and state, initially empty
          $unexplored$, a table that lists, for each visited state, the actions not yet tried
          $unbacktracked$, a table that lists, for each visited state, the backtracks not yet tried
          $s$, $a$, the previous state and action, initially null

  if GOAL-TEST($s'$) then return $stop$
  if $s'$ is a new state then $unexplored[s'] \leftarrow$ ACTIONS($s'$)
  if $s$ is not null then do
      $result[a, s] \leftarrow s'$
      add $s$ to the front of $unbacktracked[s']$
  if $unexplored[s]$ is empty then
      if $unbacktracked[s]$ is empty then return $stop$
      else a $\leftarrow$ an action $b$ such that $result[b, s]$ = POP($unbacktracked[s']$)
  else a $\leftarrow$ POP($unexplored[s']$)
  $s \leftarrow s'$
  return $a$
```

Figure 4.20 An online search agent that uses depth-first exploration. The agent is applicable only in bidirected search spaces.

Online local search

Like depth-first search, **hill-climbing search** has the property of locality in its node expansions. In fact, because it keeps just one current state in memory, hill-climbing search is *already* an online search algorithm! Unfortunately, it is not very useful in its simplest form because it leaves the agent sitting at local maxima with nowhere to go. Moreover, random restarts cannot be used, because the agent cannot transport itself to a new state.

RANDOM WALK

Instead of random restarts, one might consider using a **random walk** to explore the environment. A random walk simply selects at random one of the available actions from the current state; preference can be given to actions that have not yet been tried. It is easy to prove that a random walk will *eventually* find a goal or complete its exploration, provided that the space is finite.[15] On the other hand, the process can be very slow. Figure 4.21 shows an environment in which a random walk will take exponentially many steps to find the goal, because, at each step, backward progress is twice as likely as forward progress. The example is contrived, of course, but there are many real-world state spaces whose topology causes these kinds of "traps" for random walks.

Augmenting hill climbing with *memory* rather than randomness turns out to be a more effective approach. The basic idea is to store a "current best estimate" $H(s)$ of the cost to reach the goal from each state that has been visited. $H(s)$ starts out being just the heuristic

[15] The infinite case is much more tricky. Random walks are complete on infinite one-dimensional and two dimensional grids, but not on three-dimensional grids! In the latter case, the probability that the walk ever returns to the starting point is only about 0.3405. (See Hughes, 1995, for a general introduction.)

Dead ends are a real difficulty for robot exploration—staircases, ramps, cliffs, and all kinds of natural terrain present opportunities for irreversible actions. To make progress, we will simply assume that the state space is safely explorable — that is, some goal state is reachable from every reachable state. State spaces with reversible actions, such as mazes and 8-puzzles, can be viewed as undirected graphs and are clearly safely explorable.

SAFELY EXPLORABLE

Even in safely explorable environments, no bounded competitive ratio can be guaranteed if there are paths of unbounded cost. This is easy to show in environments with irreversible actions, but in fact it remains true for the reversible case as well, as Figure 4.19(b) shows. For this reason, it is common to describe the performance of online search algorithms in terms of the size of the entire state space rather than just the depth of the shallowest goal.

Online search agents

After each action, an online agent receives a percept telling it what state it has reached; from this information, it can augment its map of the environment. The current map is used to decide where to go next. This interleaving of planning and action means that online search algorithms are quite different from the offline search algorithms we have seen previously. For example, offline algorithms such as A* have the ability to expand a node in one part of the space and then immediately expand a node in another part of the space, because node expansion involves simulated rather than real actions. An online algorithm, on the other hand, can expand only a node that it physically occupies. To avoid traveling all the way across the tree to expand the next node, it seems better to expand nodes in a *local* order. Depth-first search has exactly this property, because (except when backtracking) the next node expanded is a child of the previous node expanded.

An online depth-first search agent is shown in Figure 4.20. This agent stores its map in a table, $result[a, s]$, that records the state resulting from executing action a in state s. Whenever an action from the current state has not been explored, the agent tries that action. The difficulty comes when the agent has tried all the actions in a state. In offline depth-first search, the state is simply dropped from the queue; in an online search, the agent has to backtrack physically. In depth-first search, this means going back to the state from which the agent entered the current state most recently. That is achieved by keeping a table that lists, for each state, the predecessor states to which the agent has not yet backtracked. If the agent has run out of states to which it can backtrack, then its search is complete.

We recommend that the reader trace through the progress of ONLINE-DFS-AGENT when applied to the maze given in Figure 4.18. It is fairly easy to see that the agent will, in the worst case, end up traversing every link in the state space exactly twice. For exploration, this is optimal; for finding a goal, on the other hand, the agent's competitive ratio could be arbitrarily bad if it goes off on a long excursion when there is a goal right next to the initial state. An online variant of iterative deepening solves this problem; for an environment that is a uniform tree, the competitive ratio of such an agent is a small constant.

Because of its method of backtracking, ONLINE-DFS-AGENT works only in state spaces where the actions are reversible. There are slightly more complex algorithms that work in general state spaces, but no such algorithm has a bounded competitive ratio.

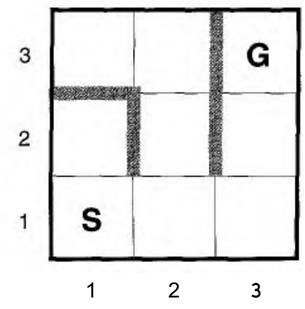


Figure 4.18 A simple maze problem. The agent starts at S and must reach G, but knows nothing of the environment.

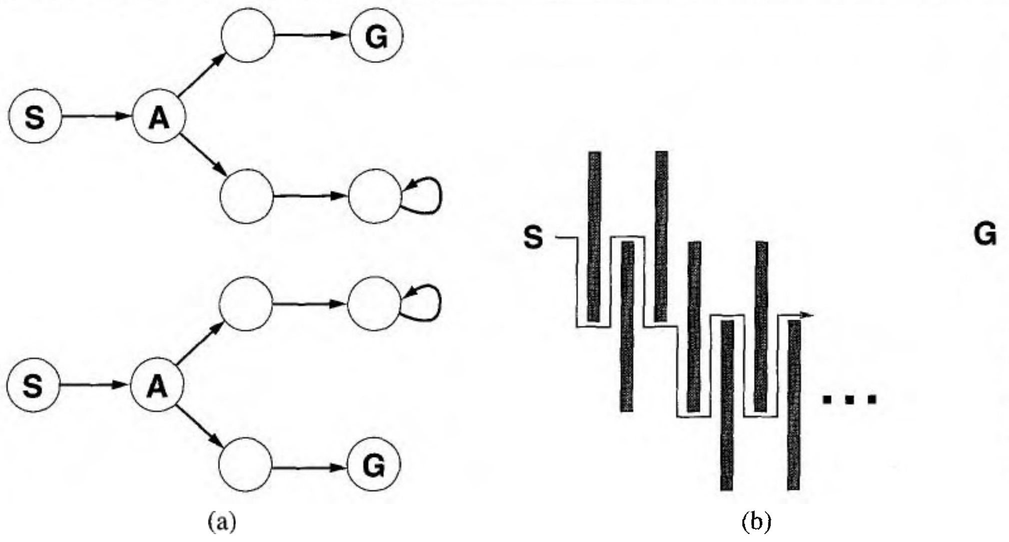


Figure 4.19 (a) Two state spaces that might lead an online search agent into a dead end. Any given agent will fail in at least one of these spaces. (b) A two-dimensional environment that can cause an online search agent to follow an arbitrarily inefficient route to the goal. Whichever choice the agent makes, the adversary blocks that route with another long, thin wall, so that the path followed is much longer than the best possible path.

Perhaps you find the term "accidentally" unconvincing — after all, there might be an algorithm that happens not to take the dead-end path as it explores. Our claim, to be more precise, is that no *algorithm can avoid dead ends in all state spaces.* Consider the two dead-end state spaces in Figure 4.19(a). To an online search algorithm that has visited states *S* and A, the two state spaces look *identical,* so it must make the same decision in both. Therefore, it will fail in one of them. This is an example of an **adversary argument** — we can imagine an adversary that constructs the state space while the agent explores it and can put the goals and dead ends wherever it likes.

ADVERSARY ARGUMENT

have to come up with an exponentially large contingency plan that considers all possible happenings, while an online search need only consider what actually does happen. For example, a chess playing agent is well-advised to make its first move long before it has figured out the complete course of the game.

EXPLORATION PROBLEM

Online search is a *necessary* idea for an **exploration** problem, where the states and actions are unknown to the agent. An agent in this state of Ignorance must use its actions as experiments to determine what to do next, and hence must interleave computation and action.

The canonical example of online search is a robot that is placed in a new building and must explore it to build a map that it can use for getting from A to B. Methods for escaping from labyrinths—required knowledge for aspiring heroes of antiquity—are also examples of online search algorithms. Spatial exploration is not the only form of exploration, however. Consider a newborn baby: it has many possible actions, but knows the outcomes of none of them, and it has experienced only a few of the possible states that it can reach. The baby's gradual discovery of how the world works is, in part, an online search process.

Online search problems

An online search problem can be solved only by an agent executing actions, rather than by a purely computational process. We will assume that the agent knows just the following:

- ACTIONS(s), which returns a list of actions allowed in state s;
- The step-cost function $c(s, a, s')$—note that this cannot be used until the agent knows that s' is the outcome; and
- GOAL-TEST(s).

Note in particular that the agent cannot access the successors of a state except by actually trying all the actions in that state. For example, in the maze problem shown in Figure 4.18, the agent does not know that going Up from (1,1) leads to (1,2); nor, having done that, does it know that going *Down* will take it back to (1,1). This degree of ignorance can be reduced in some applications—for example, a robot explorer might know how its movement actions work and be ignorant only of the locations of obstacles.

We wrll assume that the agent can always recognize a state that it has visited before, and we will assume that the actions are deterministic. (These last two assumptions are relaxed in Chapter 17.) Finally, the agent might have access to an admissible heuristic function $h(s)$ that estimates the distance from the current state to a goal state. For example, in Figure 4.18, the agent might know the location of the goal and be able to use the Manhattan distance heuristic.

Typically, the agent's objective is to reach a goal state while minimizing cost. (Another possible objective is simply to explore the entire environment.) The cost is the total path cost of the path that the agent actually travels. It is common to compare this cost with the path cost of the path the agent would follow *if* it knew the search space in advance—that is, the actual shortest path (or shortest complete exploration). In the language of online algorithms, this is called the competitive ratio; we would like it to be as small as possible.

COMPETITIVE RATIO

Although this sounds like a reasonable request, it is easy to see that the best achievable competitive ratio is infinite in some cases. For example,, if some actions are irreversible, the online search might accidentally reach a dead-end state from which no goal state is reachable.

current state. There are several schools of thought about how the new direction should be chosen at this point.

NEWTON–RAPHSON

For many problems, the most effective algorithm is the venerable **Newton–Raphson** method (Newton, 1671; Raphson, 1690). This is a general technique for finding roots of functions—that is, solving equations of the form $g(x)=0$. It works by computing a new estimate for the root x according to Newton's formula

$$x \leftarrow x - g(x)/g'(x) \,.$$

To find a maximum or minimum of f, we need to find $\mathbf{x}$ such that the gradient is zero (i.e., $\nabla f(\mathbf{x}) = 0$). Thus $g(x)$ in Newton's formula becomes $\nabla f(\mathbf{x})$, and the update equation can be written in matrix–vector form as

$$\mathbf{x} \leftarrow \mathbf{x} - \mathbf{H}_f^{-1}(\mathbf{x})\nabla f(\mathbf{x}) \,,$$

HESSIAN

where $\mathbf{H}_f(\mathbf{x})$ is the **Hessian** matrix of second derivatives, whose elements H_{ij} are given by $\partial^2 f/\partial x_i \partial x_j$. Since the Hessian has n^2 entries, Newton–Raphson becomes expensive in high-dimensional spaces, and many approximations have been developed.

Local search methods suffer from local maxima, ridges, and plateaux in continuous state spaces just as much as in discrete spaces. Random restarts and simulated annealing can be used and are often helpful. High-dimensional continuous spaces are, however, big places in which it is easy to get lost.

CONSTRAINED OPTIMIZATION

A final topic with which a passing acquaintance is useful is **constrained optimization.** An optimization problem is constrained if solutions must satisfy some hard constraints on the values of each variable. For example, in our airport-siting problem, we might constrain sites to be inside Romania and on dry land (rather than in the middle of lakes). The difficulty of constrained optimization problems depends on the nature of the constraints and the objective function. The best-known category is that of **linear programming** problems, in which constraints must be linear inequalities forming a convex region and the objective function is also linear. Linear programming problems can be solved in time polynomial in the number of variables. Problems with different types of constraints and objective functions have also been studied—quadratic programming, second-order conic programming, and so on.

LINEAR PROGRAMMING

4.5 Online Search Agents and Unknown Environments

OFFLINE SEARCH

ONLINE SEARCH

So far we have concentrated on agents that use **offline search** algorithms. They compute a complete solution before setting foot in the real world (see Figure 3.1), and then execute the solution without recourse to their percepts. In contrast, an **online search**[14] agent operates by **interleaving** computation and action: first it takes an action, then it observes the environment and computes the next action. Online search is a good idea in dynamic or semidynamic domains—domains where there is a penalty for sitting around and computing too long. Online search is an even better idea for stochastic domains. In general, an offline search would

[14] The term "online" is commonly used in computer science to refer to algorithms that must process input data as they are received, rather than waiting for the entire input data set to become available.

several places in the book, including the chapters on learning, vision, and robotics. In short, anything that deals with the real world.

Let us begin with an example. Suppose we want to place three new airports anywhere in Romania, such that the sum of squared distances from each city on the map (Figure 3.2) to its nearest airport is minimized. Then the state space is defined by the coordinates of the airports: (x_1, y_1), (x_2, y_2), and (x_3, y_3). This is a *six-dimensional* space; we also say that states are defined by six **variables.** (In general, states are defined by an n-dimensional vector of variables, **x.)** Moving around in this space corresponds to moving one or more of the airports on the map. The objective function $f(x_1, y_1, x_2, y_2, x_3, y_3)$ is relatively easy to compute for any particular state once we compute the closest cities, but rather tricky to write down in general.

One way to avoid continuous problems is simply to **discretize** the neighborhood of each state. For example, we can move only one airport at a time in either the x or y direction by a fixed amount ± 6. With 6 variables, this gives 12 possible successors for each state. We can then apply any of the local search algorithms described previously. One can also apply stochastic hill climbing and simulated annealing directly, without discretizing the space. These algorithms choose successors randomly, which can be done by generating random vectors of length 6.

GRADIENT

There are many methods that attempt to use the **gradient** of the landscape to find a maximum. The gradient of the objective function is a vector ∇f that gives the magnitude and direction of the steepest slope. For our problem, we have

$$\nabla f = \left(\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial y_1}, \frac{\partial f}{\partial x_2}, \frac{\partial f}{\partial y_2}, \frac{\partial f}{\partial x_3}, \frac{\partial f}{\partial y_3}\right) .$$

In some cases, we can find a maximum by solving the equation $\nabla f = 0$. (This could be done, for example, if we were placing just one airport; the solution is the arithmetic mean of all the cities' coordinates.) In many cases, however, this equation cannot be solved in closed form. For example, with three airports, the expression for the gradient depends on what cities are closest to each airport in the current state. This means we can compute the gradient *locally* but not *globally.* Even so, we can still perform steepest-ascent hill climbing by updating the current state via the formula

$$\mathbf{x} \leftarrow \mathbf{x} + \alpha \nabla f(\mathbf{x}) ,$$

where a is a small constant. In other cases, the objective function might not be available in a differentiable form at all—for example, the value of a particular set of airport locations may be determined by running some large-scale economic simulation package. In those cases, a so-,called **empirical gradient** can be determined by evaluating the response to small increments and decrements in each coordinate. Empirical gradient search is the same as steepest-ascent hill climbing in a discretized version of the state space.

EMPIRICAL GRADIENT

Hidden beneath the phrase "a is a small constant" lies a huge variety of methods for adjusting a . The basic problem is that, if a is too small, too many steps are needed; if a is too large, the search could overshoot the maximum. The technique of **line search** tries to overcome this dilemma by extending the current gradient direction—usually by repeatedly doubling a—until f starts to decrease again. The point at which this occurs becomes the new

LINE SEARCH

EVOLUTION AND SEARCH

The theory of evolution was developed in Charles Darwin's *On the Origin of Species by Means of Natural Selection* (1859). The central idea is simple: variations (known as mutations) occur in reproduction and will be preserved in successive generations approximately in proportion to their effect on reproductive fitness.

Darwin's theory was developed with no knowledge of how the traits of organisms can be inherited and modified. The probabilistic laws governing these processes were first identified by Gregor Mendel (1866), a monk who experimented with sweet peas using what he called artificial fertilization. Much later, Watson and Crick (1953) identified the structure of the DNA molecule and its alphabet, AGTC (adenine, guanine, thymine, cytosine). In the standard model, variation occurs both by point mutations in the letter sequence and by "crossover" (in which the DNA of an offspring is generated by combining long sections of DNA from each parent).

The analogy to local search algorithms has already been described; the principal difference between stochastic beam search and evolution is the use of *sexual* reproduction, wherein successors are generated from *multiple* organisms rather than just one. The actual mechanisms of evolution are, however, far richer than most genetic algorithms allow. For example, mutations can involve reversals, duplications, and movement of large chunks of DNA; some viruses borrow DNA from one organism and insert it in another; and there are transposable genes that do nothing but copy themselves many thousands of times within the genome. There are even genes that poison cells from potential mates that do not carry the gene, thereby increasing their chances of replication. Most important is the fact that the *genes themselves encode the mechanisms* whereby the genome is reproduced and translated into an organism. In genetic algorithms, those mechanisms are a separate program that is not represented within the strings being manipulated.

Darwinian evolution might well seem to be an inefficient mechanism, having generated blindly some 10^{45} or so organisms without improving its search heuristics one iota. Fifty years before Darwin, however, the otherwise great French naturalist Jean Lamarck (1809) proposed a theory of evolution whereby traits *acquired* by *adaptation during an organism's lifetime* would be passed on to its offspring. Such a process would be effective, but does not seem to occur in nature. Much later, James Baldwin (1896) proposed a superficially similar theory: that behavior learned during an organism's lifetime could accelerate the rate of evolution. Unlike Lamarck's, Baldwin's theory is entirely consistent with Darwinian evolution, because it relies on selection pressures operating on individuals that have found local optima among the set of possible behaviors allowed by their genetic makeup. Modern computer simulations confirm that the "Baldwin effect" is real, provided that "ordinary" evolution can create organisms whose internal performance measure is somehow correlated with actual fitness.

```
function GENETIC-ALGORITHM(population, FITNESS-FN) returns an individual
  inputs: population, a set of individuals
          FITNESS-FN, a function that measures the fitness of an individual

  repeat
     new-population ← empty set
     loop for $i$ from 1 to SIZE(population) do
        $x$ ← RANDOM-SELECTION(population, FITNESS-FN)
        $y$ ← RANDOM-SELECTION(population, FITNESS-FN)
        child ← REPRODUCE($x$, $y$)
        if (small random probability) then child ← MUTATE(child)
        add child to new-population
     population ← new-population
  until some individual is fit enough, or enough time has elapsed
  return the best individual in population, according to FITNESS-FN
```

```
function REPRODUCE($x$, $y$) returns an individual
  inputs: $x$, $y$, parent individuals

  $n$ ← LENGTH($x$)
  $c$ ← random number from 1 to $n$
  return APPEND(SUBSTRING($x$, 1, $c$), SUBSTRING($y$, $c+1$, $n$))
```

Figure 4.17 A genetic algorithm. The algorithm is the same as the one diagrammed in Figure 4.15, with one variation: in this more popular version, each mating of two parents produces only one offspring, not two.

In practice, genetic algorithms have had a widespread impact on optimization problems, such as circuit layout and job-shop scheduling. At present, it is not clear whether the appeal of genetic algorithms arises from their performance or from their æsthetically pleasing origins in the theory of evolution. Much work remains to be done to identify the conditions under which genetic algorithms perform well.

4.4 LOCAL SEARCH IN CONTINUOUS SPACES

In Chapter 2, we explained the distinction between discrete and continuous environments, pointing out that most real-world environments are continuous. Yet none of the algorithms we have described can handle continuous state spaces—the successor function would in most cases return infinitely many states! This section provides a *very brief* introduction to some local search techniques for finding optimal solutions in continuous spaces. The literature on this topic is vast; many of the basic techniques originated in the 17th century, after the development of calculus by Newton and Leibniz.[13] We will find uses for these techniques at

[13] A basic knowledge of multivariate calculus and vector arithmetic is useful when one is reading this section.

CROSSOVER

pair to be mated, a **crossover** point is randomly chosen from the positions in the string. In Figure 4.15 the crossover points are after the third digit in the first pair and after the fifth digit in the second pair.[12]

In (d), the offspring themselves are created by crossing over the parent strings at the crossover point. For example, the first child of the first pair gets the first three digits from the first parent and the remaining digits from the second parent, whereas the second child gets the first three digits from the second parent and the rest from the first parent. The 8-queens states involved in this reproduction step are shown in Figure 4.16. The example illustrates the fact that, when two parent states are quite different, the crossover operation can produce a state that is a long way from either parent state. It is often the case that the population is quite diverse early on in the process, so crossover (like simulated annealing) frequently takes large steps in the state space early in the search process and smaller steps later on when most individuals are quite similar.

MUTATION

Finally, in (e), each location is subject to random **mutation** with a small independent probability. One digit was mutated in the first, third, and fourth offspring. In the 8-queens problem, this corresponds to choosing a queen at random and moving it to a random square in its column. Figure 4.17 describes an algorithm that implements all these steps.

Like stochastic beam search, genetic algorithms combine an uphill tendency with random exploration and exchange of information among parallel search threads. The primary advantage, if any, of genetic algorithms comes from the crossover operation. Yet it can be shown mathematically that, if the positions of the genetic code is permuted initially in a random order, crossover conveys no advantage. Intuitively, the advantage comes from the ability of crossover to combine large blocks of letters that have evolved independently to perform useful functions, thus raising the level of granularity at which the search operates. For example, it could be that putting the first three queens in positions 2, 4, and 6 (where they do not attack each other) constitutes a useful block that can be combined with other blocks to construct a solution.

SCHEMA

The theory of genetic algorithms explains how this works using the idea of a **schema,** which is a substring in which some of the positions can be left unspecified. For example, the schema 246***** describes all 8-queens states in which the first three queens are in positions 2, 4, and 6 respectively. Strings that match the schema (such as 24613578) are called **instances** of the schema. It can be shown that, if the average fitness of the instances of a schema is above the mean, then the number of instances of the schema within the population will grow over time. Clearly, this effect is unlikely to be significant if adjacent bits are totally unrelated to each other, because then there will be few contiguous blocks that provide a consistent benefit. Genetic algorithms work best when schemas correspond to meaningful components of a solution. For example, if the string is a representation of an antenna, then the schemas may represent components of the antenna, such as reflectors and deflectors. A good component is likely to be good in a variety of different designs. This suggests that successful use of genetic algorithms requires careful engineering of the representation.

[12] It is here that the encoding matters. If a 24-bit encoding is used instead of 8 digits, then the crossover point has a 213 chance of being in the middle of a digit, which results in an essentially arbitrary mutation of that digit.

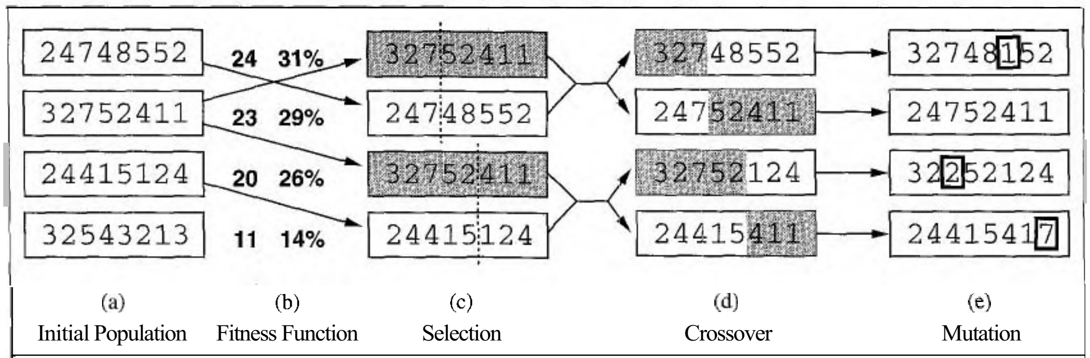


Figure 4.15 The genetic algorithm. The initial population in (a) is ranked by the fitness function in (b), resulting in pairs for mating in (c). They produce offspring in (d), which are subject to mutation in (e).

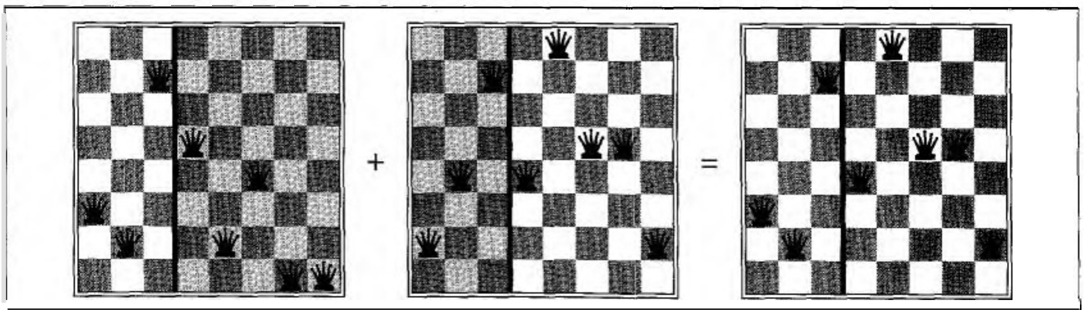


Figure 4.16 The 8-queens states corresponding to the first two parents in Figure 4.15(c) and the first offspring in Figure 4.15(d). The shaded columns are lost in the crossover step and the unshaded columns are retained.

commonly, a string of 0s and 1s. For example, an 8-queens state must specify the positions of 8 queens, each in a column of 8 squares, and so requires $8 \times \log_2 8 = 24$ bits. Alternatively, the state could be represented as 8 digits, each in the range from 1 to 8. (We will see later that the two encodings behave differently.) Figure 4.15(a) shows a population of four 8-digit strings representing 8-queens states.

The production of the next generation of states is shown in Figure 4.15(b)–(e). In (b), each state is rated by the evaluation function or (in GA terminology) the **fitness function.** A fitness function should return higher values for better states, so, for the 8-queens problem we use the number of *nonattacking* pairs of queens, which has a value of 28 for a solution. The values of the four states are 24, 23, 20, and 11. In this particular variant of the genetic algorithm, the probability of being chosen for reproducing is directly proportional to the fitness score, and the percentages are shown next to the raw scores.

FITNESS FUNCTION

In (c), a random choice of two pairs is selected for reproduction, in accordance with the probabilities in (b). Notice that one individual is selected twice and one not at all.[11] For each

[11] There are many variants of this selection rule. The method of **culling,** in which all individuals below a given threshold are discarded, can be shown to converge faster than the random version (Baum *et al.,* 1995).

```
function SIMULATED-ANNEALING(problem, schedule) returns a solution state
  inputs: problem, a problem
          schedule, a mapping from time to "temperature"
  local variables: current, a node
                   next, a node
                   T, a "temperature" controlling the probability of downward steps

  current ← MAKE-NODE(INITIAL-STATE[problem])
  for t ← 1 to $\infty$ do
     $T$ ← schedule[t]
     if $T = 0$ then return current
     next ← a randomly selected successor of current
     $\Delta E$ ← VALUE[next] – VALUE[current]
     if $\Delta E > 0$ then current ← next
     else current ← next only with probability $e^{\Delta E/T}$
```

Figure 4.14 The simulated annealing search algorithm, a version of stochastic hill climbing where some downhill moves are allowed. Downhill moves are accepted readily early in the annealing schedule and then less often as time goes on. The *schedule* input determines the value of T as a function of time.

the others. In *a local beam search, useful information is passed among the k parallel search threads.* For example, if one state generates several good successors and the other $k - 1$ states all generate bad successors, then the effect is that the first state says to the others, "Come over here, the grass is greener!" The algorithm quickly abandons unfruitful searches and moves its resources to where the most progress is being made.

In its simplest form, local beam search can suffer from a lack of diversity among the k states—they can quickly become concentrated in a small region of the state space, making the search little more than an expensive version of hill climbing. A variant called **stochastic beam search,** analogous to stochastic hill climbing, helps to alleviate this problem. Instead of choosing the best k from the the pool of candidate successors, stochastic beam search chooses k successors at random, with the probability of choosing a given successor being an increasing function of its value. Stochastic beam search bears some resemblance to the process of natural selection, whereby the "successors" (offspring) of a "state" (organism) populate the next generation according to its "value" (fitness).

STOCHASTIC BEAM SEARCH

Genetic algorithms

GENETIC ALGORITHM

A genetic algorithm (or **GA)** is a variant of stochastic beam search in which successor states are generated by combining *two* parent states, rather than by modifying a single state. The analogy to natural selection is the same as in stochastic beam search, except now we are dealing with sexual rather than asexual reproduction.

POPULATION

Like beam search, GAs begin with a set of k randomly generated states, called the **population.** Each state, or **individual,** is represented as a string over a finite alphabet—most

Simulated annealing search

A hill-climbing algorithm that ***never*** makes "downhill" moves towards states with lower value (or higher cost) is guaranteed to be incomplete, because it can get stuck on a local maximum. In contrast, a purely random walk—that is, moving to a successor chosen uniformly at random from the set of successors—is complete, but extremely inefficient. Therefore, it seems reasonable to try to combine hill climbing with a random walk in some way that yields both efficiency and completeness. **Simulated annealing** is such an algorithm. In metallurgy, **annealing** is the process used to temper or harden metals and glass by heating them to a high temperature and then gradually cooling them, thus alllowing the material to coalesce into a low-energy crystalline state. To understand simulated annealing, let's switch our point of view from hill climbing to **gradient descent** (i.e., minimizing cost) and imagine the task of getting a ping-pong ball into the deepest crevice in a bumpy surface. If we just let the ball roll, it will come to rest at a local minimum. If we shake the surface, we can bounce the ball out of the local minimum. The trick is to shake just hard enough to bounce the ball out of local minima, but not hard enough to dislodge it from the global minimum. The simulated-annealing solution is to start by shaking hard (i.e., at a high temperature) and then gradually reduce the intensity of the shaking (i.e., lower the temperature).

SIMULATED ANNEALING

GRADIENTDESCENT

The innermost loop of the simulated-annealing algorithm (Figure 4.14) is quite similar to hill climbing. Instead of picking the *best* move, however, it picks a ***random*** move. If the move improves the situation, it is always accepted. Otherwise, the algorithm accepts the move with some probability less than 1. The probability decreases exponentially with the "badness" of the move—the amount ΔE by which the evaluation is worsened. The probability also decreases as the "temperature" T goes down: "bad moves are more likely to be allowed at the start when temperature is high, and they become more unlikely as T decreases. One can prove that if the schedule lowers T slowly enough, the algorithm will find a global optimum with probability approaching 1.

Simulated annealing was first used extensively to solve VLSI layout problems in the early 1980s. It has been applied widely to factory scheduling and other large-scale optimization tasks. In Exercise 4.16, you are asked to compare its performance to that of random-restart hill climbing on the n-queens puzzle.

Local beam search

LOCAL BEAM SEARCH

Keeping just one node in memory might seem to be an extreme reaction to the problem of memory limitations. The **local beam search** algorithm[10] keeps track of k states rather than just one. It begins with k randomly generated states. At each step, all the successors of all k states are generated. If any one is a goal, the algorithm halts. Otherwise, it selects the k best successors from the complete list and repeats.

At first sight, a local beam search with k states might seem to be nothing more than running k random restarts in parallel instead of in sequence. In fact, the two algorithms are quite different. In a random-restart search, each search process runs independently of

[10] Local beam search is an adaptation of **beam search,** which is a path-based algorithm.

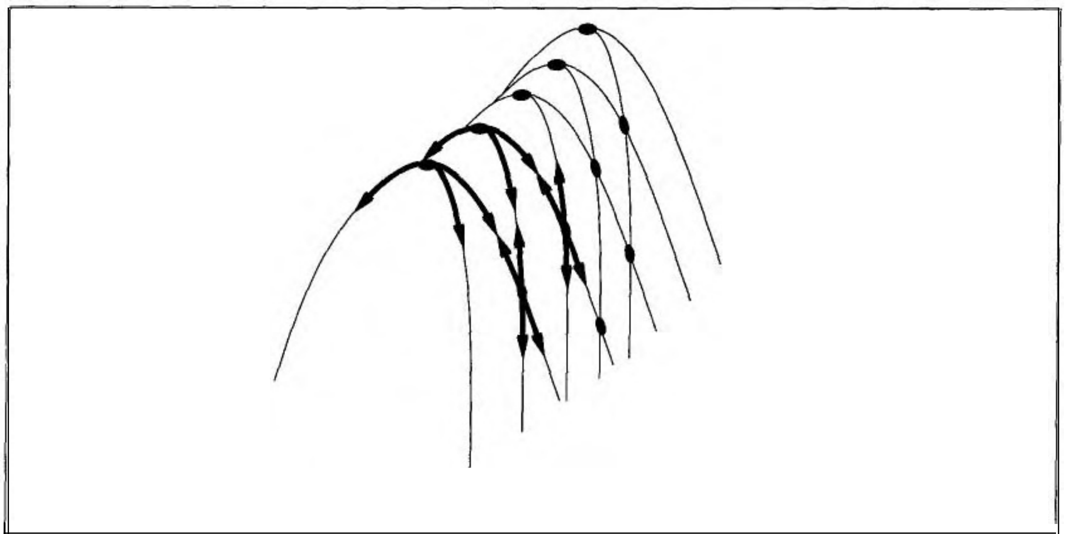


Figure 4.13 Illustration of why ridges cause difficulties for hill-climbing. The grid of states (dark circles) is superimposed on a ridge rising from left to right, creating a sequence of local maxima that are not directly connected to each other. From each local maximum, all the available actions point downhill.

RANDOM-RESTART HILL CLIMBING

climbing adopts the well known adage, "If at first you don't succeed, try, try again." It conducts a series of hill-climbing searches from randomly generated initial states,[8] stopping when a goal is found. It is complete with probability approaching 1, for the trivial reason that it will eventually generate a goal state as the initial state. If each hill-climbing search has a probability p of success, then the expected number of restarts required is $1/p$. For 8-queens instances with no sideways moves allowed, $p \approx 0.14$, so we need roughly 7 iterations to find a goal (6 failures and 1 success). The expected number of steps is the cost of one successful iteration plus $(1-p)/p$ times the cost of failure, or roughly 22 steps. When we allow sideways moves, $1/0.94 \approx 1.06$ iterations are needed on average and $(1 \times 21)+(0.06/0.94) \times 64 \approx 25$ steps. For 8-queens, then, random-restart hill climbing is very effective indeed. Even for three million queens, the approach can find solutions in under a minute.[9]

The success of hill climbing depends very much on the shape of the state-space landscape: if there are few local maxima and plateaux, random-restart hill climbing will find a good solution very quickly. On the other hand, many real problems have a landscape that looks more like a family of porcupines on a flat floor, with miniature porcupines living on the tip of each porcupine needle, ad *infinitum*. NP-hard problems typically have an exponential number of local maxima to get stuck on. Despite this, a reasonably good local maximum can often be found after a small number of restarts.

[8] Generating a ***random*** state from an implicitly specified state space can be a hard problem in itself.

[9] Luby *et al.* (1993) prove that it is best, in some cases, to restart a randomized search algorithm after a particular, fixed amount of time and that this can be much more efficient than letting each search continue indefinitely. Disallowing or limiting the number of sideways moves is an example of this.

GREEDY LOCAL SEARCH

Hill climbing is sometimes called **greedy local search** because it grabs a good neighbor state without thinking ahead about where to go next. Although greed is considered one of the seven deadly sins, it turns out that greedy algorithms often perform quite well. Hill climbing often makes very rapid progress towards a solution, because it is usually quite easy to improve a bad state. For example, from the state in Figure 4.12(a), it takes just five steps to reach the state in Figure 4.12(b), which has $h = 1$ and is very nearly a solution. Unfortunately, hill climbing often gets stuck for the following reasons:

◇ **Local maxima:** a local maximum is a peak that is higher than each of its neighboring states, but lower than the global maximum. Hill-climbing algorithms that reach the vicinity of a local maximum will be drawn upwards towards the peak, but will then be stuck with nowhere else to go. Figure 4.10 illusfrates the problem schematically. More concretely, the state in Figure 4.12(b) is in fact a local maximum (i.e., a local minimum for the cost h); every move of a single queen makes the situation worse.

◇ **Ridges:** a ridge is shown in Figure 4.13. Ridges result in a sequence of local maxima that is very difficult for greedy algorithms to navigate.

SHOULDER

◇ **Plateaux:** a plateau is an area of the state space landscape where the evaluation function is flat. It can be a flat local maximum, from which no uphill exit exists, or a **shoulder,** from which it is possible to make progress. (See Figure 4.10.) A hill-climbing search might be unable to find its way off the plateau.

In each case, the algorithm reaches a point at which no progress is being made. Starting from a randomly generated 8-queens state, steepest-ascent hill climbing gets stuck 86% of the time, solving only 14% of problem instances. It works quickly, taking just 4 steps on average when it succeeds and 3 when it gets stuck—not bad for a state space with $8^8 \approx 17$ million states.

SIDEWAYS MOVE

The algorithm in Figure 4.11 halts if it reaches a plateau where the best successor has the same value as the current state. Might it not be a good idea to keep going—to allow a **sideways move** in the hope that the plateau is really a shoulder, as shown in Figure 4.10? The answer is usually yes, but we must take care. If we always allow sideways moves when there are no uphill moves, an infinite loop will occur whenever the algorithm reaches a flat local maximum that is not a shoulder. One common solution is to put a limit on the number of consecutive sidleways moves allowed. For example, we could allow up to, say, 100 consecutive sideways moves in the 8-queens problem. This raises the percentage of problem instances solved by hill-climbing from 14% to 94%. Success comes at a cost: the algorithm averages roughly 21 steps for each successful instance and 64 for each failure.

STOCHASTIC HILL CLIMBING

FIRST-CHOICE HILL CLIMBING

Many variants of hill-climbing have been invented. **Stochastic hill climbing** chooses at random from among the uphill moves; the probability of selection can vary with the steepness of the uphill move. This usually converges more slowly than steepest ascent, but in some state landscapes it finds better solutions. **First-choice hill climbing** implements stochastic hill climbing by generating successors randomly until one is generated that is better than the current state. This is a good strategy when a state has many (e.g., thousands) of successors. Exercise 4.16 asks you to investigate.

The hill-climbing algorithms described so far are incomplete—they often fail to find a goal when one exists because they can get stuck on local maxima. **Random-restart hill**

```
function HILL-CLIMBING(problem) returns a state that is a local maximum
  inputs: problem, a problem
  local variables: current, a node
                   neighbor, a node

  current ← MAKE-NODE(INITIAL-STATE[problem])
  loop do
     neighbor ← a highest-valued successor of current
     if VALUE[neighbor] ≤ VALUE[current] then return STATE[current]
     current ← neighbor
```

Figure 4.11 The hill-climbing search algorithm **(steepest ascent** version), which is the most basic local search technique. At each step the current node is replaced by the best neighbor; in this version, that means the neighbor with the highest VALUE, but if a heuristic cost estimate h is used, we would find the neighbor with the lowest h.

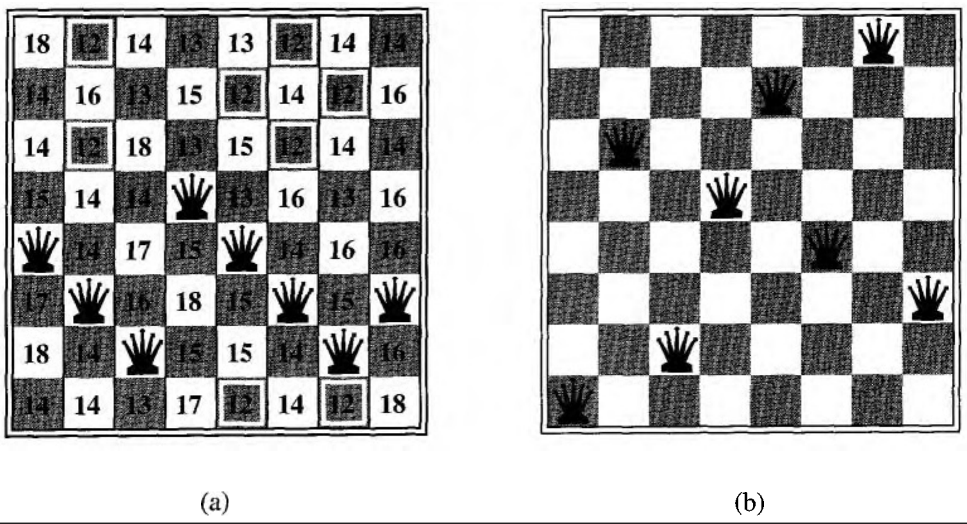


Figure 4.12 (a) An 8-queens state with heuristic cost estimate h = 17, showing the value of h for each possible successor obtained by moving a queen within its column. The best moves are marked. (b) A local minimum in the 8-queens state space; the state has h = 1 but every successor has a higher cost.

8 x 7 = 56 successors). The heuristic cost function h is the number of pairs of queens that are attacking each other, either directly or indirectly. The global minimum of this function is zero, which occurs only at perfect solutions. Figure 4.12(a) shows a state with h = 17. The figure also shows the values of all its successors, with the best successors having h = 12. Hill-climbing algorithms typically choose randomly among the set of best successors, if there is more than one.

Chapter 3. For example, nature provides an objective function—reproductive fitness—that Darwinian evolution could be seen as attempting to optimize, but there is no "goal test" and no "path cost" for this problem.

STATE SPACE LANDSCAPE

To understand local search, we will find it very useful to consider the **state space landscape** (as in Figure 4.10). A landscape has both "location" (defined by the state) and "elevation" (defined by the value of the heuristic cost function or objective function). If elevation corresponds to cost, then the aim is to find the lowest valley—a **global minimum;** if elevation corresponds to an objective function, then the aim is to find the highest peak—a **global maximum,.** (You can convert from one to the other just by inserting a minus sign.) Local search algorithms explore this landscape. A **complete,** local search algorithm always finds a goal if one exists; an **optimal** algorithm always finds a global minimum/maximum.

GLOBAL MINIMUM

GLOBAL MAXIMUM

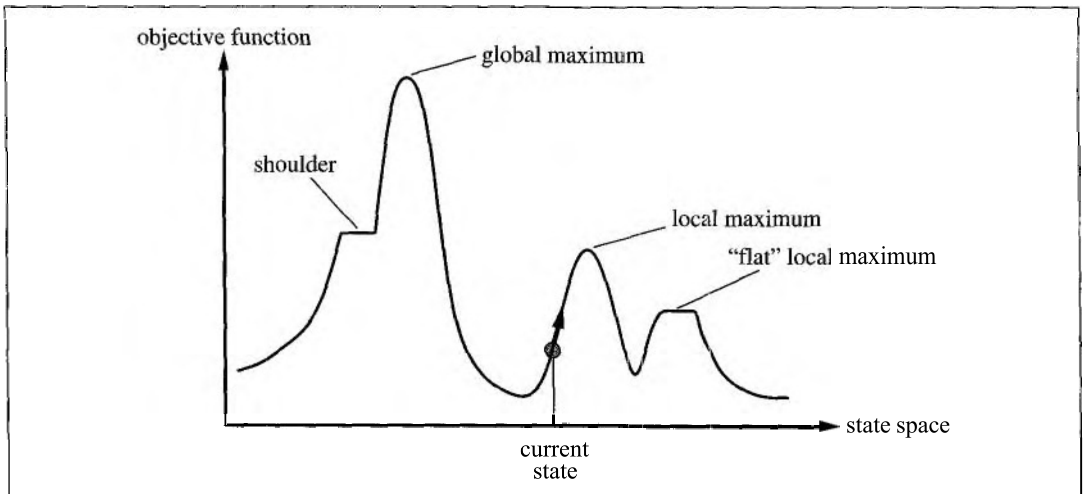


Figure 4.10 A one-dimensional state space landscape in which elevation corresponds to the objective function. The aim is to find the global maximum. Hill-climbing search modifies the current state to try to improve it, as shown by the arrow. The various topographic features are defined in the text.

Hill-climbing search

HILL-CLIMBING

The **hill-climbing** search algorithm is shown in Figure 4.11. It is simply a loop that continually moves in the direction of increasing value—that is, uphill. It terminates when it reaches a "peak" where no neighbor has a higher value. The algorithm does not maintain a search tree, so the current node data structure need only record the state and its objective function value. Hill-climbing does not look ahead beyond the immediate neighbors of the current state. This resembles trying to find the top of Mount Everest in a thick fog while suffering from amnesia.

To illustrate hill-climbing, we will use the **8-queens problem** introduced on page 66. Local-search algorithms typically use a **complete-state formulation,** where each state has 8 queens on the board, one per column. The successor function returns all possible states generated by moving a single queen to another square in the same column (so each state has amples from which $h(n)$ can be learned. Each example consists of a state from the solution path and the actual cost of the solution from that point. From these examples, an **inductive learning** algorithm can be used to construct a function $h(n)$ that can (with luck) predict solution costs for other states that arise during search. Techniques for doing just this using neural nets, decision trees, and other methods are demonstrated in Chapter 18. (The reinforcement learning methods described in Chapter 21 are also applicable.)

FEATURES

Inductive learning methods work best when supplied with **features** of a state that are relevant to its evaluation, rather than with just the raw state description. For example, the feature "number of misplaced tiles" might be helpful in predicting the actual distance of a state from the goal. Let's call this feature $x_1(n)$. We could take 100 randomly generated 8-puzzle configurations and gather statistics on their actual solution costs. We might find that when $x_1(n)$ is 5, the average solution cost is around 14, and so on. Given these data, the value of x_1 can be used to predict $h(n)$. Of course, we can use several features. A second feature $x_2(n)$ might be "number of pairs of adjacent tiles that are also adjacent in the goal state." How should $x_1(n)$ and $x_2(n)$ be combined to predict $h(n)$? A common approach is to use a linear combination:

$$h(n) = c_1 x_1(n) + c_2 x_2(n) .$$

The constants c_1 and c_2 are adjusted to give the best fit to the actual data on solution costs. Presumably, c_1 should be positive and c_2 should be negative.

4.3 LOCAL SEARCH ALGORITHMS AND OPTIMIZATION PROBLEMS

The search algorithms that we have seen so far are designed to explore search spaces systematically. This systematicity is achieved by keeping one or more paths in memory and by recording which alternatives have been explored at each point along the path and which have not. When a goal is found, the *path* to that goal also constitutes a solution to the problem.

In many problems, however, the path to the goal is irrelevant. For example, in the 8-queens problem (see page 66), what matters is the final configuration of queens, not the order in which they are added. This class of problems includes many important applications such as integrated-circuit design, factory-floor layout, job-shop scheduling, automatic programming, telecommunications network optimization, vehicle routing, and portfolio management.

LOCAL SEARCH

CURRENT STATE

If the path to the goal does not matter, we might consider a different class of algorithms, ones that do not worry about paths at all. **Local search** algorithms operate using a single **current state** (rather than multiple paths) and generally move only to neighbors of that state. Typically, the paths followed by the search are not retained. Although local search algorithms are not systematic, they have two key advantages: (1) they use very little memory—usually a constant amount; and (2) they can often find reasonable solutions in large or infinite (continuous) state spaces for which systematic algorithms are unsuitable.

OPTIMIZATION PROBLEMS

OBJECTIVE FUNCTION

In addition to finding goals, local search algorithms are useful for solving pure **optimization problems,** in which the aim is to find the best state according to an **objective function.** Many optimization problems do not fit the "standard" search model introduced in

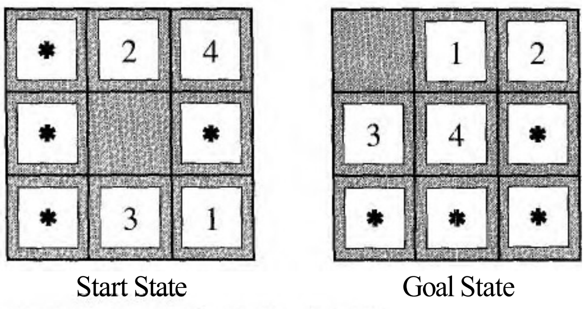


Figure 4.9 A subproblem of the 8-puzzle instance given in Figure 4.7. The task is to get tiles 1, 2, 3, and 4 into their correct positions, without worrying about what happens to the other tiles.

The choice of 1-2-3-4 is fairly arbitrary; we could also construct databases for 5-6-7-8, and for 2-4-6-8, and so on. Each database yields an admissible heuristic, and these heuristics can be combined, as explained earlier, by taking the maximum value. A combined heuristic of this kind is much more accurate than the Manhattan distance; the number of nodes generated when solving random 15-puzzles can be reduced by a factor of 1000.

One might wonder whether the heuristics obtained from the 1-2-3-4 database and the 5-6-7-8 could be added, since the two subproblems seem not to overlap. Would this still give an admissible heuristic? The answer is no, because the solutions of the 1-2-3-4 subproblem and the 5-6-7-8 subproblem for a given state will almost certainly share some moves—it is unlikely that 1-2-3-4 can be moved into place without touching 5-6-7-8, and vice versa. But what if we don't count those moves? That is, we record niot the total cost of solving the 1-2-3-4 subproblem, but just the number of moves involving 1-2-3-4. Then it is easy to see that the sum of the two costs is still a lower bound on the cost of solving the entire problem. This is the idea behind **disjoint pattern databases.** Using such databases, it is possible to solve random 15-puzzles in a few milliseconds—the numbes of nodes generated is reduced by a factor of 10,000 compared with using Manhattan distance. For 24-puzzles, a speedup of roughly a million can be obtained.

DISJOINT PATTERN DATABASES

Disjoint pattern databases work for sliding-tile puzzles because the problem can be divided up in such a way that each move affects only one subproblem—because only one tile is moved at a time. For a problem such as Rubik's cube, this kind of subdivision cannot be done because each move affects 8 or 9 of the 26 cubies. Currently, it is not clear how to define disjoint databases for such problems.

Learning heuristics from experience

A heuristic function $h(n)$ is supposed to estimate the cost of a solution beginning from the state at node n. How could an agent construct such a function? One solution was given in the preceding section—namely, to devise relaxed problems for which an optimal solution can be found easily. Another solution is to learn from experience. "Experience" here means solving lots of 8-puzzles, for instance. Each optimal solution to an 8-puzzle problem provides exwe can generate three relaxed problems by removing one or both of the conditions:

(a) A tile can move from square A to square B if A is adjacent to B.
(b) A tile can move from square A to square B if B is blank.
(c) A tile can move from square A to square B.

From (a), we can derive h_2 (Manhattan distance). The reasoning is that h_2 would be the proper score if we moved each tile in turn to its destination. The heuristic derived from (b) is discussed in Exercise 4.9. From (c), we can derive h_1 (misplaced tiles), because it would be the proper score if tiles could move to their intended destination in one step. Notice that it is crucial that the relaxed problems generated by this technique can be solved essentially ***without search,*** because the relaxed rules allow the problem to be decomposed into eight independent subproblems. If the relaxed problem is hard to solve, then the values of the corresponding heuristic will be expensive to obtain.[7]

A program called ABSOLVER can generate heuristics automatically from problem definitions, using the "relaxed problem" method and various other techniques (Prieditis, 1993). ABSOLVER generated a new heuristic for the 8-puzzle better than any preexisting heuristic and found the first useful heuristic for the famous Rubik's cube puzzle.

One problem with generating new heuristic functions is that one often fails to get one "clearly best" heuristic. If a collection of admissible heuristics $h_1 \ldots h_m$ is available for a problem, and none of them dominates any of the others, which should we choose? As it turns out, we need not make a choice. We can have the best of all worlds, by defining

$$h(n) = \max\{h_1(n), \ldots, h_m(n)\} .$$

This composite heuristic uses whichever function is most accurate on the node in question. Because the component heuristics are admissible, h is admissible; it is also easy to prove that h is consistent. Furthermore, h dominates all of its component heuristics.

SUBPROBLEM

Admissible heuristics can also be derived from the solution cost of a **subproblem** of a given problem. For example, Figure 4.9 shows a subproblem of the 8-puzzle instance in Figure 4.7. The subproblem involves getting tiles 1, 2, 3, 4 into their correct positions. Clearly, the cost of the optimal solution of this subproblem is a lower bound on the cost of the complete problem. It turns out to be substantially more accurate than Manhattan distance in some cases.

PATTERN DATABASES

The idea behind **pattern databases** is to store these exact solution costs for every possible subproblem instance—in our example, every possible configuration of the four tiles and the blank. (Notice that the locations of the other four tiles are irrelevant for the purposes of solving the subproblem, but moves of those tiles do count towards the cost.) Then, we compute an admissible heuristic h_{DB} for each complete state encountered during a search simply by looking up the corresponding subproblem configuration in the database. The database itself is constructed by searching backwards from the goal state and recording the cost of each new pattern encountered; the expense of this search is amortized over many subsequent problem instances.

[7] Note that a perfect heuristic can be obtained simply by allowing h to run a full breadth-first search "on the sly." Thus, there is a tradeoff between accuracy and computation time for heuristic functions.

	Search Cost			Effective Branching Factor		
d	IDS	A*(h_1)	A*(h_2)	IDS	A*(h_1)	A*(h_2)
2	10	6	6	2.45	1.79	1.79
4	112	13	12	2.87	1.48	1.45
6	680	20	18	2.73	1.34	1.30
8	6384	39	25	2.80	1.33	1.24
10	47127	93	39	2.79	1.38	1.22
12	3644035	227	73	2.78	1.42	1.24
14	–	539	113	–	1.44	1.23
16	–	1301	211	–	1.45	1.25
18	–	3056	363	–	1.46	1.26
20	–	7276	676	–	1.47	1.27
22	–	18094	1219	–	1.48	1.28
24	–	39135	1641	–	1.48	1.26

Figure 4.8 Comparison of the search costs and effective branching factors for the ITERATIVE-DEEPENING-SEARCH and A* algorithms with h_1, h_2. Data are averaged over 100 instances of the &puzzle, for various solution lengths.

Inventing admissible heuristic functions

We have seen that both h_1 (misplaced tiles) and h_2 (Manhattan distance) are fairly good heuristics for the 8-puzzle and that h_2 is better. How might one have come up with h_2? Is it possible for a computer to invent such a heuristic mechanically?

h_1 and h_2 are estimates of the remaining path length for the 8-puzzle, but they are also perfectly accurate path lengths for *simplified* versions of the puzzle. If the rules of the puzzle were changed so that a tile could move anywhere, instead of just to the adjacent empty square, then h_1 would give the exact number of steps in the shortest solution. Similarly, if a tile could move one square in any direction, even onto an occupied square, then h_2 would give the exact number of steps in the shortest solution. A problem with fewer restrictions on the actions is called a **relaxed problem.** *The cost of an optimal solution to a relaxedproblem is an admissible heuristic for the original problem.* The heuristic is admissible because the optimal solution in the original problem is, by definition, also a solution in the relaxed problem and therefore must be at least as expensive as the optimal solution in the relaxed problem. Because the derived heuristic is an exact cost for the relaxed problem, it must obey the triangle inequality and is therefore **consistent** (see page 99).

RELAXED PROBLEM

If a problem definition is written down in a formal language, it is possible to construct relaxed problems automatically.[6] For example, if the 8-puzzle actions are described as

A tile can move from square A to square B if
 A is horizontally or vertically adjacent to B **and** B is blank,

[6] In Chapters 8 and 11, we will describe formal languages suitable for this task; with formal descriptions that can be manipulated, the construction of relaxed problems can be automated. For now, we will use English.

- h_1 = the number of misplaced tiles. For Figure 4.7, all of the eight tiles are out of position, so the start state would have $h_1 = 8$. h_1 is an admissible heuristic, because it is clear that any tile that is out of place must be moved at least once.
- h_2 = the sum of the distances of the tiles from their goal positions. Because tiles cannot move along diagonals, the distance we will count is the sum of the horizontal and vertical distances. This is sometimes called the **city block distance** or **Manhattan distance.** h_2 is also admissible, because all any move can do is move one tile one step closer to the goal. Tiles 1 to 8 in the start state give a Manhattan distance of

MANHATTAN DISTANCE

$$h_2 = 3+1+2+2+2+3+3+2 = 18 .$$

As we would hope, neither of these overestimates the true solution cost, which is 26.

The effect of heuristic accuracy on performance

EFFECTIVE BRANCHING FACTOR

One way to characterize the quality of a heuristic is the **effective branching factor b^*.** If the total number of nodes generated by A^* for a particular problem is N, and the solution depth is d, then b^* is the branching factor that a uniform tree of depth d would have to have in order to contain N + *1* nodes. Thus,

$$N+1 = 1+b^*+(b^*)^2+\ldots+(b^*)^d .$$

For example, if A^* finds a solution at depth 5 using 52 nodes, then the effective branching factor is 1.92. The effective branching factor can vary across problem instances, but usually it is fairly constant for sufficiently hard problems. Therefore, experimental measurements of b^* on a small set of problems can provide a good guide to the heuristic's overall usefulness. A well-designed heuristic would have a value of b^* close to 1, allowing fairly large problems to be solved.

To test the heuristic functions h_1 and h_2, we generated 1200 random problems with solution lengths from 2 to 24 (100 for each even number) and solved them with iterative deepening search and with A^* tree search using both h_1 and h_2. Figure 4.8 gives the average number of nodes generated by each strategy and the effective branching factor. The results suggest that h_2 is better than h_1, and is far better than using iterative deepening search. On our solutions with length 14, A^* with h_2 is 30,000 times more efficient than uninformed iterative deepening search.

One might ask whether h_2 is ***always*** better than h_1. The answer is yes. It is easy to see from the definitions of the two heuristics that, for any node n, $h_2(n) \geq h_1(n)$. We thus say that h_2 **dominates** h_1. Domination translates directly into efficiency: A* using h_2 will never expand more nodes than A^* using h_1 (except possibly for some nodes with $f(n) = C^*$). The argument is simple. Recall the observation on page 100 that every node with $f(n) < C^*$ will surely be expanded. This is the same as saying that every node with $h(n) < C^* - g(n)$ will surely be expanded. But because h_2 is at least as big as h_1 for all nodes, every node that is surely expanded by A^* search with h_2 will also surely be expanded with h_1, and h_1 might also cause other nodes to be expanded as well. Hence, it is always better to use a heuristic function with higher values, provided it does not overestimate and that the computation time for the heuristic is not too large.

DOMINATION

space is a computation step that alters the internal state; for example, each computation step in A* expands a leaf node and adds its successors to the tree. Thus, Figure 4.3, which shows a sequence of larger and larger search trees, can be seen as depicting a path in the metalevel state space where each state on the path is an object-level search tree.

METALEVEL LEARNING

Now, the path in Figure 4.3 has five steps, including one step, the expansion of Fagaras, that is not especially helpful. For harder problems, there will be many such missteps, and a **metalevel learning** algorithm can learn from these experiences to avoid exploring unpromising subtrees. The techniques used for this kind of learning are described in Chapter 21. The goal of learning is to minimize the **total cost** of problem solving, trading off computational expense and path cost.

4.2 Heuristic Functions

In this section, we will look at heuristics for the 8-puzzle, in order to shed light on the nature of heuristics in general.

The 8-puzzle was one of the earliest heuristic search problems. As mentioned in Section 3.2, the object of the puzzle is to slide the tiles horizontally or vertically into the empty space until the configuration matches the goal configuration (Figure 4.7).

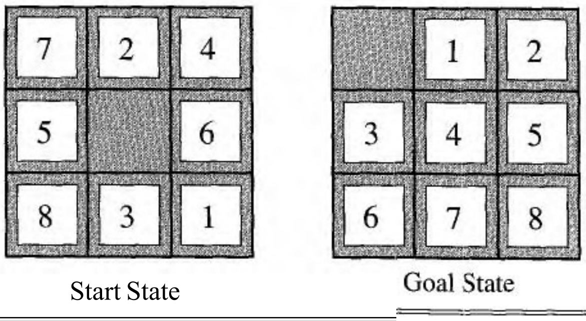


Figure 4.7 A typical instance of the 8-puzzle. The solution is 26 steps long.

The average solution cost for a randomly generated %-puzzleinstance is about 22 steps. The branching factor is about 3. (When the empty tile is in the middle, there are four possible moves; when it is in a corner there are two; and when it is along an edge there are three.) This means that an exhaustive search to depth 22 would look at about $3^{22} \approx 3.1 \times 10^{10}$ states. By keeping track of repeated states, we could cut this down by a factor of about 170,000, because there are only $9!/2 = 181{,}440$ distinct states that are reachable. (See Exercise 3.4.) This is a manageable number, but the corresponding number for the 15-puzzle is roughly 10^{13}, so the next order of business is to find a good heuristic function. If we want to find the shortest solutions by using A*, we need a heuristic function that never overestimates the number of steps to the goal. There is a long history of such heuristics for the 15-puzzle; here are two commonly-used candidates:

is—well—simpler. SMA* proceeds just like A*, expanding the best leaf until memory is full. At this point, it cannot add a new node to the search tree without dropping an old one. SMA* always drops the *worst* leaf node—the one with the highest f-value. Like RBFS, SMA* then backs up the value of the forgotten node to its parent. In this way, the ancestor of a forgotten subtree knows the quality of the best path in that subtree. With this information, SMA* regenerates the subtree only when *all otherpaths* have been shown to look worse than the path it has forgotten. Another way of saying this is that, if all the descendants of a node n are forgotten, then we will not know which way to go from n, but we will still have an idea of how worthwhile it is to go anywhere from n.

The complete algorithm is too complicated to reproduce here,[5] but there is one subtlety worth mentioning. We said that SMA* expands the best leaf and deletes the worst leaf. What if *all* the leaf nodes have the same f-value? Then the algorithm might select the same node for deletion and expansion. SMA* solves this problem by expanding the *newest* best leaf and deleting the *oldest* worst leaf. These can be the same node only if there is only one leaf; in that case, the current search tree must be a single path from root to leaf that fills all of memory. If the leaf is not a goal node, then *even if it is on an optimal solution path,* that solution is not reachable with the available memory. Therefore, the node can be discarded exactly as if it had no successors.

SMA* is complete if there is any reachable solution—that is, if d, the depth of the shallowest goal node, is less than the memory size (expressed in nodes). It is optimal if any optimal solution is reachable; otherwise it returns the best reachable solution. In practical terms, SMA* might well be the best general-purpose algorithm for finding optimal solutions, particularly when the state space is a graph, step costs are not uniform, and node generation is expensive compared to the additional overhead of maintaining the open and closed lists.

THRASHING

On very hard problems, however, it will often be the case that SMA* is forced to switch back and forth continually between a set of candidate solution paths, only a small subset of which can fit in memory. (This resembles the problem of **thrashing** in disk paging systems.) Then the extra time required for repeated regeneration of the same nodes means that problems that would be practically solvable by A*, given unlimited memory, become intractable for SMA*. That is to say, *memory limitations can make a problem intractable from the point of view of computation time.* Although there is no theory to explain the tradeoff between time and memory, it seems that this is an inescapable problem. The only way out is to drop the optimality requirement.

Learning to search better

METALEVEL STATE SPACE

OBJECT-LEVEL STATE SPACE

We have presented several fixed strategies—breadth-first, greedy best-first, and so on—that have been designed by computer scientists. Could an agent *learn* how to search better? The answer is yes, and the method rests on an important concept called the **metalevel state space.** Each state in a metalevel state space captures the internal (computational) state of a program that is searching in an **object-level state space** such as Romania. For example, the internal state of the A* algorithm consists of the current search tree. Each action in the metalevel state

[5] A rough sketch appeared in the first edition of this book.

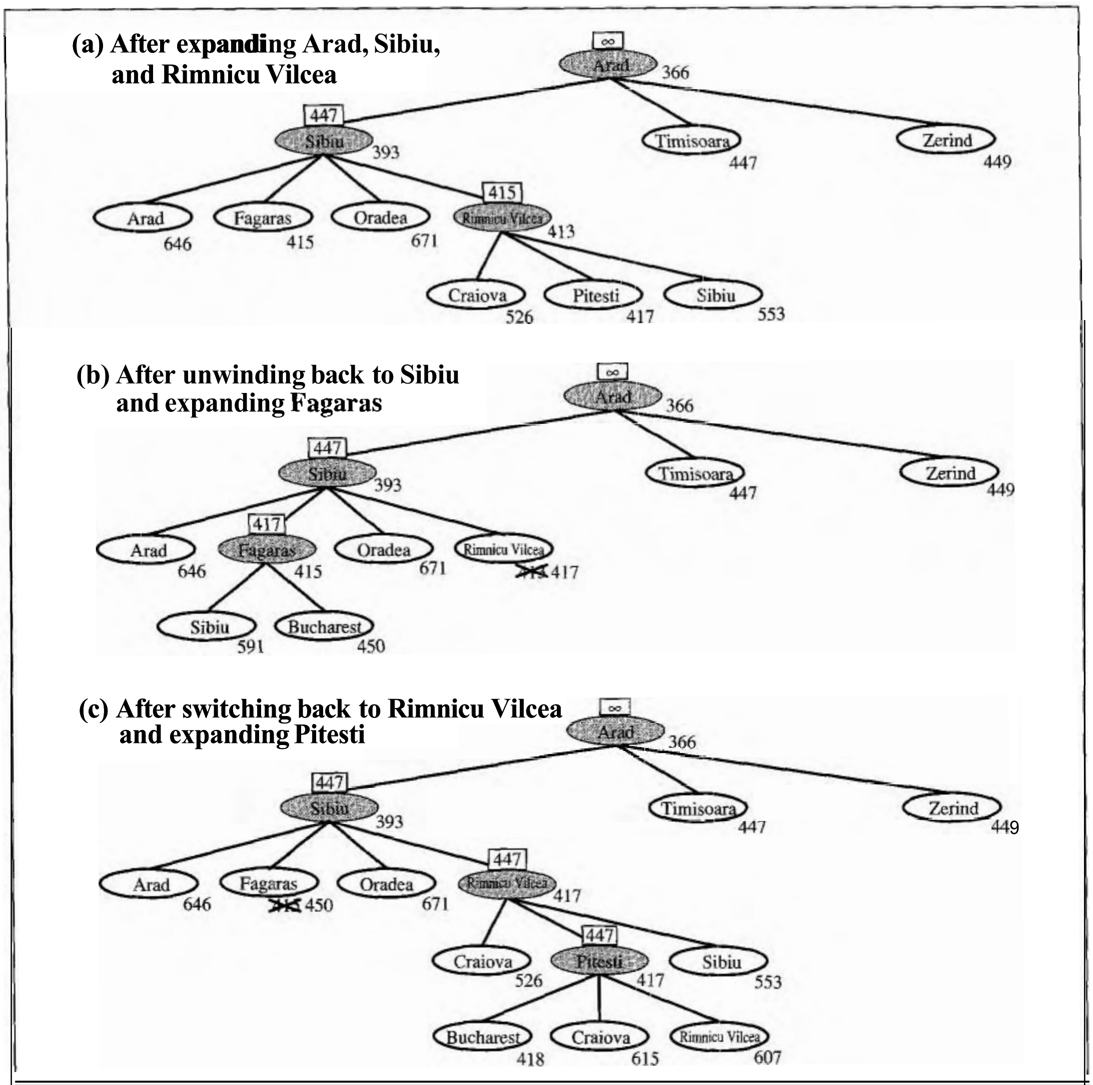


Figure 4.6 Stages in an RBFS search for the shortest route to Bucharest. The f-limit value for each recursive call is shown on top of each current node. (a) The path via Rimnicu Vilcea is followed until the current best leaf (Pitesti) has a value that is worse than the best alternative path (Fagaras). (b) The recursion unwinds and the best leaf value of the forgotten subtree (417) is backed up to Rimnicu Vilcea; then Fagaras is expanded, revealing a best leaf value of 450. (c) The recursion unwinds and the best leaf value of the forgotten subtree (450) is backed up to Fagaras; then Rirnnicu Vilcea is expanded. This time, because the best alternative path (through Timisoara) costs at least 447, the expansion continues to Bucharest.

but it uses only linear space: even if more memory were available, RBFS has no way to make use of it.

MA'
SMA'

It seems sensible, therefore, to use all available memory. Two algorithms that do this are MA* (memory-bounded A*) and SMA* (simplified MA*). We will describe SMA*, which

```
function RECURSIVE-BEST-FIRST-SEARCH(problem) returns a solution, or failure
    RBFS(problem, MAKE-NODE(INITIAL-STATE[problem]), $\infty$)

function RBFS(problem, node, f-limit) returns a solution, or failure and a new f-cost limit
    if GOAL-TEST[problem](STATE[node]) then return node
    successors $\leftarrow$ EXPAND(node, problem)
    if successors is empty then return failure, $\infty$
    for each s in successors do
        $f[s] \leftarrow \max(g(s) + h(s), f[node])$
    repeat
        best $\leftarrow$ the lowest f-value node in successors
        if f[best] > f-limit then return failure, f[best]
        alternative $\leftarrow$ the second-lowest f-value among successors
        result, f[best] $\leftarrow$ RBFS(problem, best, min(f-limit, alternative))
        if result $\neq$ failure then return result
```

Figure 4.5 The algorithm for recursive best-first search.

than continuing indefinitely down the current path, it keeps track of the f-value of the best alternative path available from any ancestor of the current node. If the current node exceeds this limit, the recursion unwinds back to the alternative path. As the recursion unwinds, RBFS replaces the f-value of each node along the path with the best f-value of its children. In this way, RBFS remembers the f-value of the best leaf in the forgotten subtree and can therefore decide whether it's worth reexpanding the subtree at some later time. Figure 4.6 shows how RBFS reaches Bucharest.

RBFS is somewhat more efficient than IDA*, but still suffers from excessive node regeneration. In the example in Figure 4.6, RBFS first follows the path via Rimnicu Vilcea, then "changes its mind" and tries Fagaras, and then changes its mind back again. These mind changes occur because every time the current best path is extended, there is a good chance that its f-value will increase—h is usually less optimistic for nodes closer to the goal. When this happens, particularly in large search spaces, the second-best path might become the best path, so the search has to backtrack to follow it. Each mind change corresponds to an iteration of IDA*, and could require many reexpansions of forgotten nodes to recreate the best path and extend it one more node.

Like A*, RBFS is an optimal algorithm if the heuristic function $h(n)$ is admissible. Its space complexity is linear in the depth of the deepest optimal solution, but its time complexity is rather difficult to characterize: it depends both on the accuracy of the heuristic function and on how often the best path changes as nodes are expanded. Both IDA* and RBFS are subject to the potentially exponential increase in complexity associated with searching on graphs (see Section 3.5), because they cannot check for repeated states other than those on the current path. Thus, they may explore the same state many times.

IDA* and RBFS suffer from using *too* little memory. Between iterations, IDA* retains only a single number: the current f-cost limit. RBFS retains more information in memory, while still guaranteeing optimality. The concept of pruning—eliminating possibilities from consideration without having to examine them—is important for many areas of AI.

OPTIMALLY EFFICIENT

One final observation is that among optimal algorithms of this type—algorithms that extend search paths from the root—A* is **optimally efficient** for any given heuristic function. That is, no other optimal algorithm is guaranteed to expand fewer nodes than A* (except possibly through tie-breaking among nodes with $f(n) = C^*$). This is because any algorithm that *does not* expand all nodes with $f(n) < C^*$ runs the risk of missing the optimal solution.

That A* search is complete, optimal, and optimally efficient among all such algorithms is rather satisfying. Unfortunately, it does not mean that A* is the answer to all our searching needs. The catch is that, for most problems, the number of nodes within the goal contour search space is still exponential in the length of the solution. Although the proof of the result is beyond the scope of this book, it has been shown that exponential growth will occur unless the error in the heuristic function grows no faster than the logarithm of the actual path cost. In mathematical notation, the condition for subexponential growth is that

$$|h(n) - h^*(n)| \leq O(\log h^*(n)) ,$$

where $h^*(n)$ is the *true* cost of getting from n to the goal. For almost all heuristics in practical use, the error is at least proportional to the path cost, and the resulting exponential growth eventually overtakes any computer. For this reason, it is often impractical to insist on finding an optimal solution. One can use variants of A* that find suboptimal solutions quickly, or one can sometimes design heuristics that are more accurate, but not strictly admissible. In any case, the use of a good heuristic still provides enormous savings compared to the use of an uninformed search. In Section 4.2, we will look at the question of designing good heuristics.

Computation time is not, however, A*'s main drawback. Because it keeps all generated nodes in memory (as do all GRAPH-SEARCH algorithms), A* usually runs out of space long before it runs out of time. For this reason, A* is not practical for many large-scale problems. Recently developed algorithms have overcome the space problem without sacrificing optimality or completeness, at a small cost in execution time. These are discussed next.

Memory-bounded heuristic search

The simplest way to reduce memory requirements for A* is to adapt the idea of iterative deepening to the heuristic search context, resulting in the iterative-deepening A" (IDA*) algorithm. The main difference between IDA* and standard iterative deepening is that the cutoff used is the f-cost ($g + h$) rather than the depth; at each iteration, the cutoff value is the smallest f-cost of any node that exceeded the cutoff on the previous iteration. IDA* is practical for many problems with unit step costs and avoids the substantial overhead associated with keeping a sorted queue of nodes. Unfortunately, it suffers from the same difficulties with real-valued costs as does the iterative version of uniform-cost search described in Exercise 3.11. This section briefly examines two more recent memory-bounded algorithms, called RBFS and MA*.

RECURSIVE BEST-FIRST SEARCH

Recursive best-first search (RBFS) is a simple recursive algorithm that attempts to mimic the operation of standard best-first search, but using only linear space. The algorithm is shown in Figure 4.5. Its structure is similar to that of a recursive depth-first search, but rather

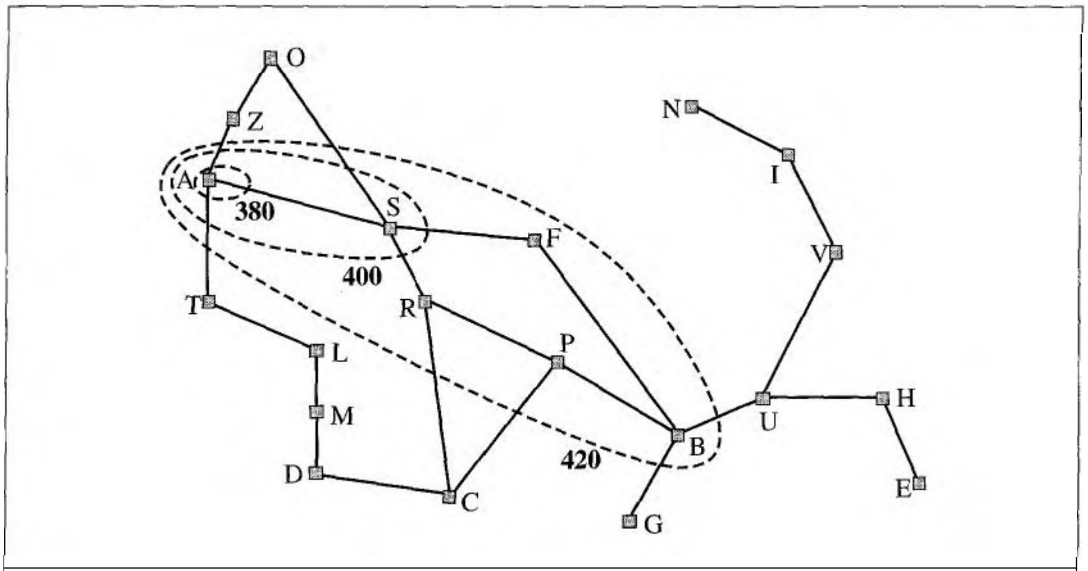


Figure 4.4 Map of Romania showing contours at $f = 380$, $f = 400$ and $f = 420$, with Arad as the start state. Nodes inside a given contour have f-costs less than or equal to the contour value.

CONTOURS

The fact that f-costs are nondecreasing along any path also means that we can draw **contours** in the state space, just like the contours in a topographic map. Figure 4.4 shows an example. Inside the contour labeled 400, all nodes have $f(n)$ less than or equal to 400, and so on. Then, because A* expands the fringe node of lowest f-cost, we can see that an A* search fans out from the start node, adding nodes in concentric bands of increasing f-cost.

With uniform-cost search (A* search using $h(n) = 0$), the bands will be "circular" around the start state. With more accurate heuristics, the bands will stretch toward the goal state and become more narrowly focused around the optimal path. If C^* is the cost of the optimal solution path, then we can say the following:

- A* expands all nodes with $f(n) < C^*$.
- A* might then expand some of the nodes right on the "goal contour" (where $f(n) = C^*$) before selecting a goal node.

Intuitively, it is obvious that the first solution found must be an optimal one, because goal nodes in all subsequent contours will have higher f-cost, and thus higher g-cost (because all goal nodes have $h(n) = 0$). Intuitively, it is also obvious that A* search is complete. As we add bands of increasing f, we must eventually reach a band where f is equal to the cost of the path to a goal state.[4]

Notice that A* expands no nodes with $f(n) > C^*$—for example, Timisoara is not expanded in Figure 4.3 even though it is a child of the root. We say that the subtree below Timisoara is **pruned;** because h_{SLD} is admissible, the algorithm can safely ignore this subtree

PRUNING

[4] Completeness requires that there be only finitely many nodes with cost less than or equal to C^*, a condition that is true if all step costs exceed some finite ϵ and if b is finite.

suboptimal goal node G_2 appears on the fringe, and let the cost of the optimal solution be C^*. Then, because G_2 is suboptimal and because $h(G_2) = 0$ (true for any goal node), we know

$$f(G_2) = g(G_2) + h(G_2) = g(G_2) > C^* .$$

Now consider a fringe node n that is on an optimal solution path—for example, Pitesti in the example of the preceding paragraph. (There must always be such a node if a solution exists.) If $h(n)$ does not overestimate the cost of completing the solution path, then we know that

$$f(n) = g(n) + h(n) \le C^* .$$

Now we have shown that $f(n) \le C^* < f(G_2)$, so G_2 will not be expanded and A* must return an optimal solution.

If we use the GRAPH-SEARCH algorithm of Figure 3.19 instead of TREE-SEARCH, then this proof breaks down. Suboptimal solutions can be returned because GRAPH-SEARCH can discard the optimal path to a repeated state if it is not the first one generated. (See Exercise 4.4.) There are two ways to fix this problem. The first solution is to extend GRAPH-SEARCH so that it discards the more expensive of any two paths found to the same node. (See the discussion in Section 3.5.) The extra bookkeeping is messy, but it does guarantee optimality. The second solution is to ensure that the optimal path to any repeated state is always the first one followed—as is the case with uniform-cost search. This property holds if we impose an extra requirement on $h(n)$, namely the requirement of **consistency** (also called **monotonicity**). A heuristic $h(n)$ is consistent if, for every node n and every successor n' of n generated by any action a, the estimated cost of reaching the goal from n is no greater than the step cost of getting to n' plus the estimated cost of reaching the goal from n':

CONSISTENCY

MONOTONICITY

$$h(n) \le c(n, a, n') + h(n') .$$

TRIANGLE INEQUALITY

This is a form of the general **triangle inequality,** which stipulates that each side of a triangle cannot be longer than the sum of the other two sides. Here, the triangle is formed by n, n', and the goal closest to n. It is fairly easy to show (Exercise 4.7) that every consistent heuristic is also admissible. The most important consequence of consistency is the following: *A* using* GRAPH-SEARCH *is optimal if $h(n)$ is consistent.*

Although consistency is a stricter requirement than admissibility, one has to work quite hard to concoct heuristics that are admissible but not consistent. All the admissible heuristics we discuss in this chapter are also consistent. Consider, for example, h_{SLD}. We know that the general triangle inequality is satisfied when each side is measured by the straight-line distance, and that the straight-line distance between n and n' is no greater than $c(n, a, n')$. Hence, h_{SLD} is a consistent heuristic.

Another important consequence of consistency is the following: *If $h(n)$ is consistent, then the values* off *(n)along any path are nondecueasing.* The proof follows directly from the definition of consistency. Suppose n' is a successor of n; then $g(n') = g(n) + c(n, a, n')$ for some a, and we have

$$f(n') = g(n') + h(n') = g(n) + c(n, a, n') + h(n') \ge g(n) + h(n) = f(n).$$

It follows that the sequence of nodes expanded by A* using GRAPH-SEARCH is in nondecreasing order of $f(n)$. Hence, the first goal node selected for expansion must be an optimal solution, since all later nodes will be at least as expensive.

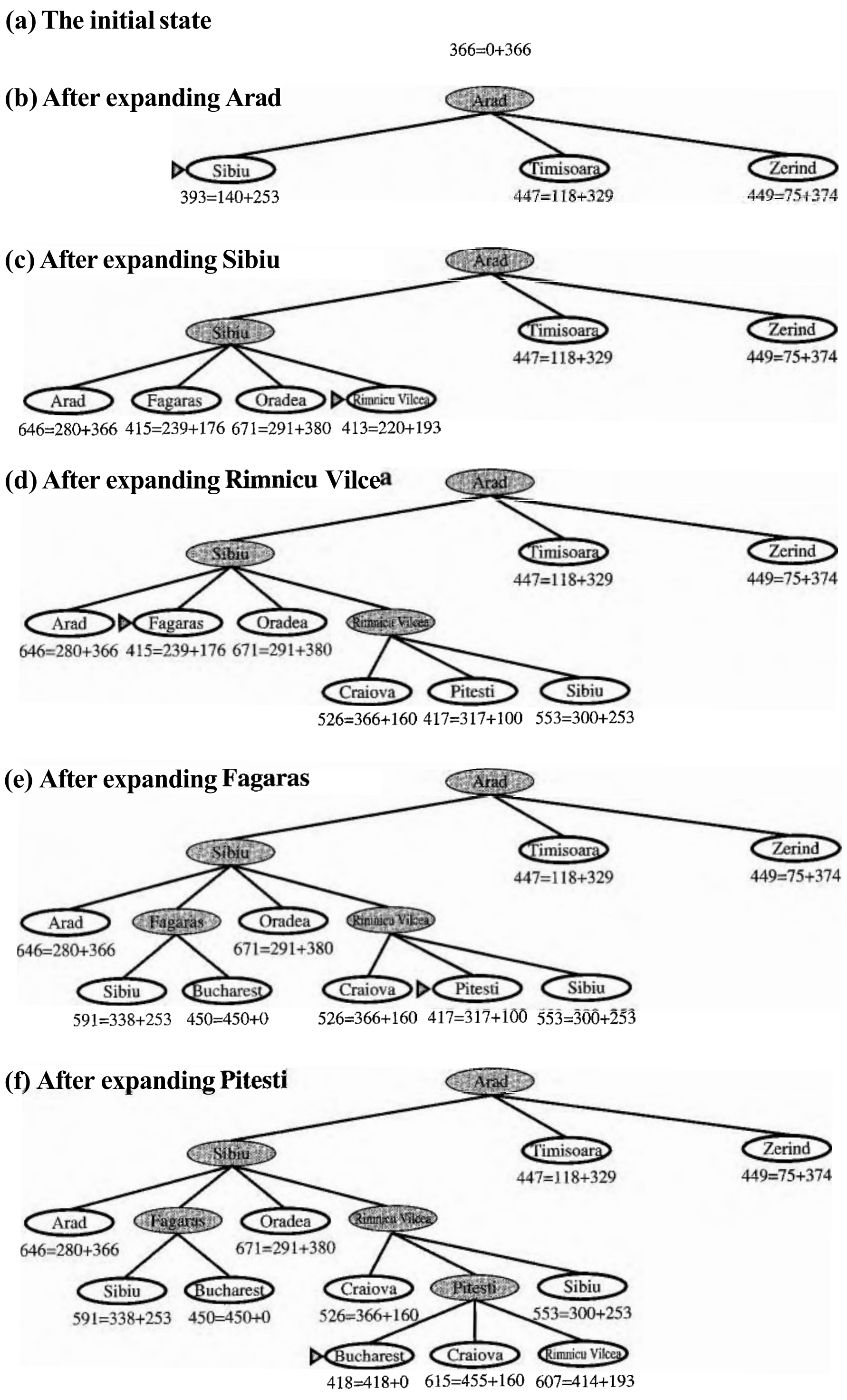


Figure 4.3 Stages in an A* search for Bucharest. Nodes are labeled with $f = g + h$. The h values are the straight-line distances to Bucharest taken from Figure 4.1.

to Fagaras, but it is a dead end. The solution is to go first to Vaslui—a step that is actually farther from the goal according to the heuristic—and then to continue to Urziceni, Bucharest, and Fagaras. In this case, then, the heuristic causes unnecessary nodes to be expanded. Furthermore, if we are not careful to detect repeated states, the solution will never be found—the search will oscillate between Neamt and Iasi.

Greedy best-first search resembles depth-first search in the way it prefers to follow a single path all the way to the goal, but will back up when it hits a dead end. It suffers from the same defects as depth-first search—it is not optimal, and it is incomplete (because it can start down an infinite path and never return to try other possibilities). The worst-case time and space complexity is $O(b^m)$, where m is the maximum depth of the search space. With a good heuristic function, however, the complexity can be reduced substantially. The amount of the reduction depends on the particular problem and on the quality of the heuristic.

A* search: Minimizing the total estimated solution cost

A* SEARCH

The most widely-known form of best-first search is called **A*** search (pronounced "A-star search"). It evaluates nodes by combining $g(n)$, the cost to reach the node, and $h(n)$, the cost to get from the node to the goal:

$$f(n) = g(n) + h(n)$$

Since $g(n)$ gives the path cost from the start node to node n, and $h(n)$ is the estirnated cost of the cheapest path from n to the goal, we have

$$f(n) = \text{estimated cost of the cheapest solution through n}$$

Thus, if we are trying to find the cheapest solution, a reasonable thing to try first is the node with the lowest value of $g(n) + h(n)$. It turns out that this strategy is more than just reasonable: provided that the heuristic function $h(n)$ satisfies certain conditions, A* search is both complete and optimal.

ADMISSIBLE HEURISTIC

The optimality of A* is straightforward to analyze if it is used with TREE-SEARCH. In this case, A* is optimal if $h(n)$ is an admissible heuristic—that is, provided that $h(n)$ ***never overestimates*** the cost to reach the goal. Admissible heuristics are by nature optimistic, because they think the cost of solving the problem is less than it actually is. Since $g(n)$ is the exact cost to reach n, we have as immediate consequence that $f(n)$ never overestimates the true cost of a solution through n.

An obvious example of an admissible heuristic is the straight-line distance h_{SLD} that we used in getting to Bucharest. Straight-line distance is admissible because the shortest path between any two points is a straight line, so the straight line cannot be an overestimate. In Figure 4.3, we show the progress of an A* tree search for Bucharest. The values of g are computed from the step costs in Figure 3.2, and the values of h_{SLD} are given in Figure 4.1. Notice in particular that Bucharest first appears on the fringe at step (e), but it is not selected for expansion because its f-cost (450) is higher than that of Pitesti (417). Another way to say this is that there ***might*** be a solution through Pitesti whose cost is as low as 417, so the algorithm will not settle for a solution that costs 450. From this example, we can extract a general proof that A* ***using*** TREE-SEARCH ***is optimal if*** $h(n)$ ***is admissible.*** Suppose a

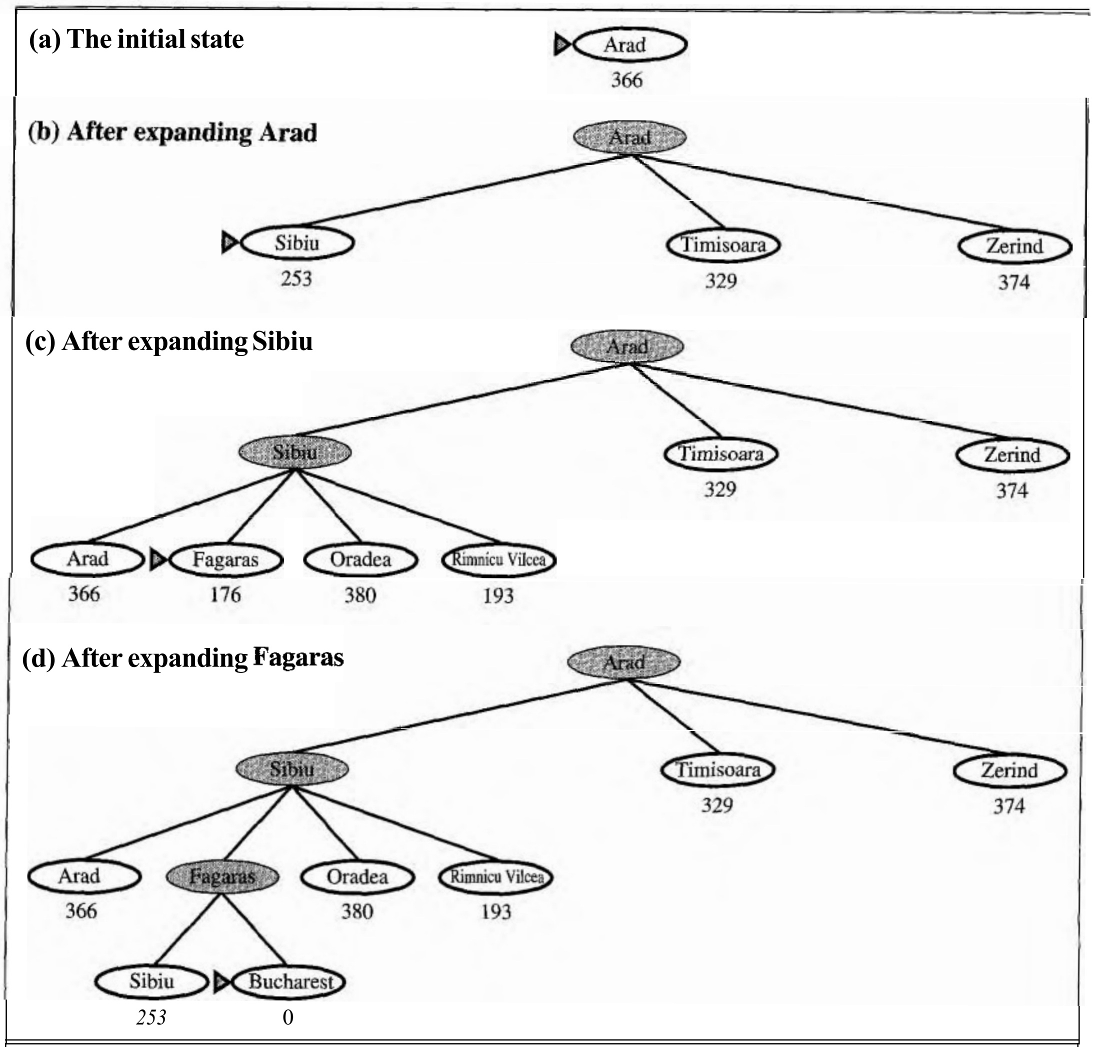


Figure 4.2 Stages in a greedy best-first search for Bucharest using the straight-line distance heuristic h_{SLD}. Nodes are labeled with their h-values.

Figure 4.2 shows the progress of a greedy best-first search using h_{SLD} to find a path from Arad to Bucharest. The first node to be expanded from Arad will be Sibiu, because it is closer to Bucharest than either Zerind or Timisoara. The next node to be expanded will be Fagaras, because it is closest. Fagaras in turn generates Bucharest, which is the goal. For this particular problem, greedy best-first search using h_{SLD} finds a solution without ever expanding a node that is not on the solution path; hence, its search cost is minimal. It is not optimal, however: the path via Sibiu and Fagaras to Bucharest is 32 kilometers longer than the path through Rimnicu Vilcea and Pitesti. This shows why the algorithm is called "greedy"—at each step it tries to get as close to the goal as it can.

Minimizing $h(n)$ is susceptible to false starts. Consider the problem of getting from Iasi to Fagaras. The heuristic suggests that Neamt be expanded first, because it is closest the goal. All we can do is choose the node that ***appears*** to be best according to the evaluation function. If the evaluation function is exactly accurate, then this will indeed be the best node; in reality, the evaluation function will sometimes be off, and can lead the search astray. Nevertheless, we will stick with the name "best-first search," because "seemingly-best-first search" is a little awkward.

HEURISTIC FUNCTION

There is a whole family of BEST-FIRST-SEARCH algorithms with different evaluation functions.' A key component of these algorithms is a heuristic **function**,[2] denoted $h(n)$:

$$h(n) = \text{ estimated cost of the cheapest path from node } n \text{ to a goal node.}$$

For example, in Romania, one might estimate the cost of the cheapest path from Arad to Bucharest via the straight-line distance from Arad to Bucharest.

Heuristic functions are the most common form in which additional knowledge of the problem is imparted to the search algorithm. We will study heuristics in more depth in Section 4.2. For now, we will consider them to be arbitrary problem-specific functions, with one constraint: if n is a goal node, then $h(n) = 0$. The remainder of this section covers two ways to use heuristic information to guide search.

Greedy best-first search

GREEDY BEST-FIRST SEARCH

Greedy best-first search[3] tries to expand the node that is closest to the goal, on the grounds that this is likely to lead to a solution quickly. Thus, it evaluates nodes by using just the heuristic function: $f(n) = h(n)$.

STRAIGHT-LINE DISTANCE

Let us see how this works for route-finding problems in Romania, using the **straight-line** distance heuristic, which we will call h_{SLD}. If the goal is Bucharest, we will need to know the straight-line distances to Bucharest, which are shown in Figure 4.1. For example, $h_{SLD}(In(Arad)) = 366$. Notice that the values of h_{SLD} cannot be computed from the problem description itself. Moreover, it takes a certain amount of experience to know that h_{SLD} is correlated with actual road distances and is, therefore, a useful heuristic.

Arad	366	**Mehadia**	241
Bucharest	0	**Neamt**	234
Craiova	160	**Oradea**	380
Drobeta	242	**Pitesti**	100
Eforie	161	**Rimnicu Vilcea**	193
Fagaras	176	**Sibiu**	253
Giurgiu	77	**Timisoara**	329
Hirsova	151	**Urziceni**	80
Iasi	226	**Vaslui**	199
Lugoj	244	**Zerind**	374

Figure 4.1 Values of h_{SLD}—straight-line distances to Bucharest.

[1] Exercise 4.3 asks you to show that this family includes several familiar uninformed algorithms.

[2] heuristic function $h(n)$ takes a node as input, but it depends only on the *state* at that node.

[3] Our first edition called this **greedy search;** other authors have called it **best-first search.** Our more general usage of the latter term follows Pearl (1984).

4 INFORMED SEARCH AND EXPLORATION

In which we see how information about the state space can prevent algorithms from blundering about in the dark.

Chapter 3 showed that uninformed search strategies can find solutions to problems by systematically generating new states and testing them against the goal. Unfortunately, these strategies are incredibly inefficient in most cases. This chapter shows how an informed search strategy—one that uses problem-specific knowledge—can find solutions more efficiently. Section 4.1 describes informed versions of the algorithms in Chapter **3,** and Section 4.2 explains how the necessary problem-specific information can be obtained. Sections 4.3 and 4.4 cover algorithms that perform purely **local search** in the state space, evaluating and modifying one or more current states rather than systematically exploring paths from an initial state. These algorithms are suitable for problems in which the path cost is irrelevant and all that matters is the solution state itself. The family of local-search algorithms includes methods inspired by statistical physics **(simulated annealing)** and evolutionary biology **(genetic algorithms).** Finally, Section 4.5 investigates **online search,** in which the agent is faced with a state space that is completely unknown.

4.1 INFORMED (HEURISTIC) SEARCH STRATEGIES

INFORMED SEARCH

This section shows how an **informed search** strategy--one that uses problem-specific knowledge beyond the definition of the problem itself—can find solutions more efficiently than an uninformed strategy.

BEST-FIRST SEARCH

EVALUATION FUNCTION

The general approach we will consider is called **best-first search.** Best-first search is an instance of the general TREE-SEARCH or GRAPH-SEARCH algorithm in which a node is selected for expansion based on an **evaluation function,** $f(n)$. Traditionally, the node with the *lowest* evaluation is selected for expansion, because the evaluation measures distance to the goal. Best-first search can be implemented within our general search framework via a priority queue, a data structure that will maintain the fringe in ascending order of f-values.

The name "best-first search" is a venerable but inaccurate one. After all, if we could *really* expand the best node first, it would not be a search at all; it would be a straight march to

e. Now suppose that there are locations from which the view is identical. (For example, suppose the world is a grid with square obstacles.) What kind of problem does the agent now face? What do solutions look like?

3.17 On page 62, we said that we would not consider problems with negative path costs. In this exercise, we explore this in more depth.

a. Suppose that actions can have arbitrarily large negative costs; explain why this possibility would force any optimal algorithm to explore the entire state space.

b. Does it help if we insist that step costs must be greater than or equal to some negative constant *c*? Consider both trees and graphs.

c. Suppose that there is a set of operators that form a loop, so that executing the set in some order results in no net change to the state. If all of these operators have negative cost, what does this imply about the optimal behavior for an agent in such an environment?

d. One can easily imagine operators with high negative cost, even in domains such as route finding. For example, some stretches of road might have such beautiful scenery as to far outweigh the normal costs in terms of time and fuel. Explain, in precise terms, within the context of state-space search, why humans do not drive round scenic loops indefinitely, and explain how to define the state space and operators for route finding so that artificial agents can also avoid looping.

e. Can you think of a real domain in which step costs are such as to cause looping?

3.18 Consider the sensorless, two-location vacuum world under Murphy's Law. Draw the belief state space reachable from the initial belief state $\{1,2,3,4,5,6,7,8\}$, and explain why the problem is unsolvable. Show also that if the world is fully observable then there is a solution sequence for each possible initial state.

3.19 Consider the vacuum-world problem defined in Figure 2.2.

a. Which of the algorithms defined in this chapter would be appropriate for this problem? Should the algorithm check for repeated states?

b. Apply your chosen algorithm to compute an optimal sequence of actions for a 3×3 world whose initial state has dirt in the three top squares and the agent in the center.

c. Construct a search agent for the vacuum world, and evaluate its performance in a set of 3×3 worlds with probability 0.2 of dirt in each square. Include the search cost as well as path cost in the performance measure, using a reasonable exchange rate.

d. Compare your best search agent with a simple randomized reflex agent that sucks if there is dirt and otherwise moves randomly.

e. Consider what would happen if the world were enlarged to $n \times n$. How does the performance of the search agent and of the reflex agent vary with n?