4 INFORMED SEARCH METHODS

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. It also shows how optimization problems can be solved.

4.1 BEST-FIRST SEARCH

EVALUATION FUNCTION

BEST-FIRST SEARCH

In Chapter 3, we found several ways to apply knowledge to the process of formulating a problem in terms of states and operators. Once we are given a well-defined problem, however, our options are more limited. If we plan to use the GENERAL-SEARCH algorithm from Chapter 3, then the only place where knowledge can be applied is in the queuing function, which determines the node to expand next. Usually, the knowledge to make this determination is provided by an **evaluation function** that returns a number purporting to describe the desirability (or lack thereof) of expanding the node. When the nodes are ordered so that the one with the best evaluation is expanded first, the resulting strategy is called **best-first search**. It can be implemented directly with GENERAL-SEARCH, as shown in Figure 4.1.

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 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 omniscient, 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.

Just as there is a whole family of GENERAL-SEARCH algorithms with different ordering functions, there is also a whole family of BEST-FIRST-SEARCH algorithms with different evaluation

```
function BEST-FIRST-SEARCH(problem, EVAL-FN) returns a solution sequence
  inputs: problem, a problem
          Eval-Fn, an evaluation function

  Queueing-Fn — a function that orders nodes by EVAL-FN
  return GENERAL-SEARCH(problem, Queueing-Fn)
```

Figure 4.1 An implementation of best-first search using the general search algorithm.

functions. Because they aim to find low-cost solutions, these algorithms typically use some estimated measure of the cost of the solution and try to minimize it. We have already seen one such measure: the use of the path cost g to decide which path to extend. This measure, however, does not direct search *toward the goal. In order to focus the search, the measure must incorporate some estimate of the cost of the path from a state to the closest goal state.* We look at two basic approaches. The first tries to expand the node closest to the goal. The second tries to expand the node on the least-cost solution path.

Minimize estimated cost to reach a goal: Greedy search

One of the simplest best-first search strategies is to minimize the estimated cost to reach the goal. That is, the node whose state is judged to be closest to the goal state is always expanded first. For most problems, the cost of reaching the goal from a particular state can be estimated but cannot be determined exactly. A function that calculates such cost estimates is called a **heuristic function**, and is usually denoted by the letter h:

HEURISTIC FUNCTION

$h(n)$ = estimated cost of the cheapest path from the state at node n to a goal state.

GREEDY SEARCH

A best-first search that uses h to select the next node to expand is called **greedy search**, for reasons that will become clear. Given a heuristic function h, the code for greedy search is just the following:

```
function GREEDY-SEARCH(problem) returns a solution or failure
  return BEST-FIRST-SEARCH(problem, h)
```

Formally speaking, h can be any function at all. We will require only that $h(n) = 0$ if n is a goal.

To get an idea of what a heuristic function looks like, we need to choose a particular problem, because heuristic functions are problem-specific. Let us return to the route-finding problem from Arad to Bucharest. The map for that problem is repeated in Figure 4.2.

STRAIGHT-LINE DISTANCE

A good heuristic function for route-finding problems like this is the **straight-line distance** to the goal. That is,

$h_{SLD}(n)$ = straight-line distance between n and the goal location.

HISTORY OF "HEURISTIC"

> By now the space aliens had mastered my own language, but they still made simple mistakes like using "hermeneutic" when they meant "heuristic."
> — a Louisiana factory worker in Woody Allen's *The UFO Menace*

The word "heuristic" is derived from the Greek verb *heuriskein*, meaning "to find" or "to discover." Archimedes is said to have run naked down the street shouting "*Heureka*" (I have found it) after discovering the principle of flotation in his bath. Later generations converted this to Eureka.

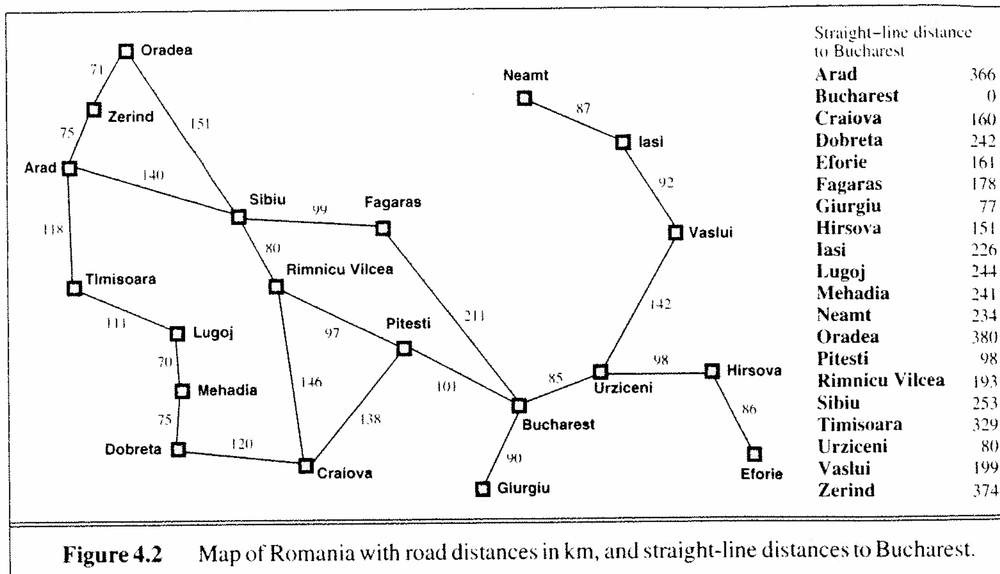
The technical meaning of "heuristic" has undergone several changes in the history of AI. In 1957, George Polya wrote an influential book called *How to Solve It* that used "heuristic" to refer to the study of methods for discovering and inventing problem-solving techniques, particularly for the problem of coming up with mathematical proofs. Such methods had often been deemed not amenable to explication.

Some people use heuristic as the opposite of algorithmic. For example, Newell, Shaw, and Simon stated in 1963, "A process that may solve a given problem, but offers no guarantees of doing so, is called a heuristic for that problem." But note that there is nothing random or nondeterministic about a heuristic search algorithm: it proceeds by algorithmic steps toward its result. In some cases, there is no guarantee how long the search will take, and in some cases, the quality of the solution is not guaranteed either. Nonetheless, it is important to distinguish between "nonalgorithmic" and "not precisely characterizable."

Heuristic techniques dominated early applications of artificial intelligence. The first "expert systems" laboratory, started by Ed Feigenbaum, Bruce Buchanan, and Joshua Lederberg at Stanford University, was called the Heuristic Programming Project (HPP). Heuristics were viewed as "rules of thumb" that domain experts could use to generate good solutions without exhaustive search. Heuristics were initially incorporated directly into the structure of programs, but this proved too inflexible when a large number of heuristics were needed. Gradually, systems were designed that could accept heuristic information expressed as "rules," and rule-based systems were born.

Currently, heuristic is most often used as an adjective, referring to any technique that improves the average-case performance on a problem-solving task, but does not necessarily improve the worst-case performance. In the specific area of search algorithms, it refers to a function that provides an estimate of solution cost.

A good article on heuristics (and one on hermeneutics!) appears in the *Encyclopedia of AI* (Shapiro, 1992).



Straight-line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Figure 4.2 Map of Romania with road distances in km, and straight-line distances to Bucharest.

Notice that we can only calculate the values of h_{SLD} if we know the map coordinates of the cities in Romania. Furthermore, h_{SLD} is only useful because a road from A to B usually tends to head in more or less the right direction. This is the sort of extra information that allows heuristics to help in reducing search cost.

Figure 4.3 shows the progress of a greedy search to find a path from Arad to Bucharest. With the straight-line-distance heuristic, 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, the heuristic leads to minimal search cost: it finds a solution without ever expanding a node that is not on the solution path. However, it is not perfectly optimal: the path it found via Sibiu and Fagaras to Bucharest is 32 kilometers longer than the path through Rimnicu Vilcea and Pitesti. This path was not found because Fagaras is closer to Bucharest in straight-line distance than Rimnicu Vilcea, so it was expanded first. The strategy prefers to take the biggest bite possible out of the remaining cost to reach the goal, without worrying about whether this will be best in the long run—hence the name "greedy search." Although greed is considered one of the seven deadly sins, it turns out that greedy algorithms often perform quite well. They tend to find solutions quickly, although as shown in this example, they do not always find the optimal solutions: that would take a more careful analysis of the long-term options, not just the immediate best choice.

Greedy search is susceptible to false starts. Consider the problem of getting from Iasi to Fagaras. The heuristic suggests that Neamt be expanded first, 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. Hence, in this case, 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.

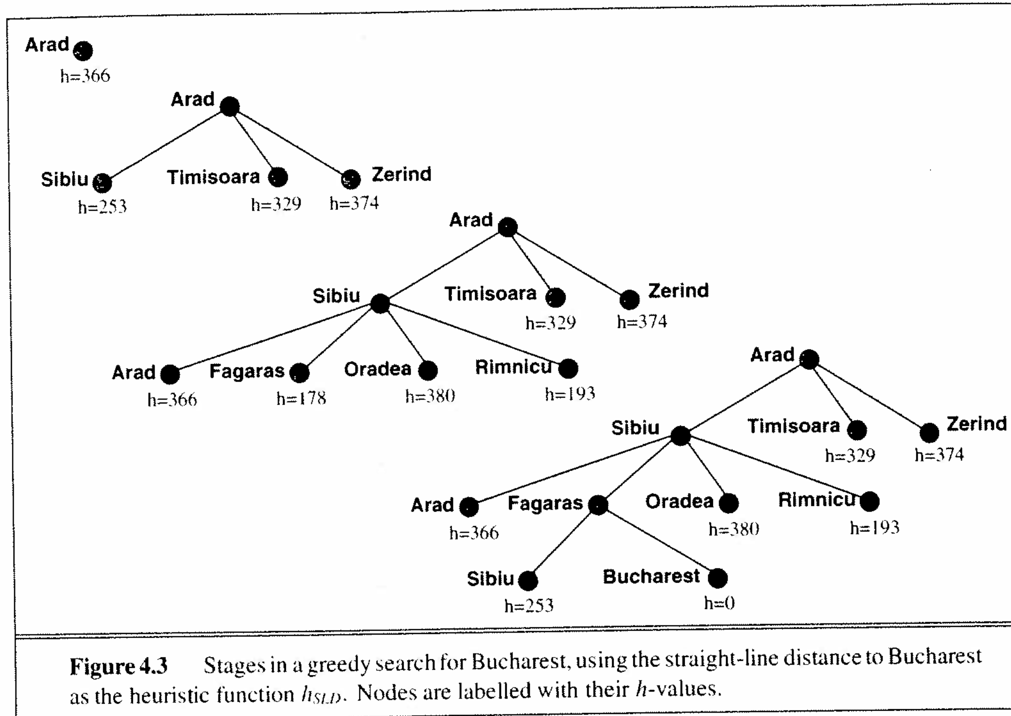


Figure 4.3 Stages in a greedy search for Bucharest, using the straight-line distance to Bucharest as the heuristic function h_{SLD}. Nodes are labelled with their h-values.

Greedy 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 complexity for greedy search is $O(b^m)$, where m is the maximum depth of the search space. Because greedy search retains all nodes in memory, its space complexity is the same as its time complexity. With a good heuristic function, the space and time complexity can be reduced substantially. The amount of the reduction depends on the particular problem and quality of the h function.

Minimizing the total path cost: A* search

Greedy search minimizes the estimated cost to the goal, $h(n)$, and thereby cuts the search cost considerably. Unfortunately, it is neither optimal nor complete. Uniform-cost search, on the other hand, minimizes the cost of the path so far, $g(n)$; it is optimal and complete, but can be very inefficient. It would be nice if we could combine these two strategies to get the advantages of both. Fortunately, we can do exactly that, combining the two evaluation functions simply by summing them:

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

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

$f(n)$ = 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 f. The pleasant thing about this strategy is that it is more than just reasonable. We can actually prove that it is complete and optimal, given a simple restriction on the h function.

ADMISSIBLE HEURISTIC

The restriction is to choose an h function that *never overestimates* the cost to reach the goal. Such an h is called an **admissible heuristic**. Admissible heuristics are by nature optimistic, because they think the cost of solving the problem is less than it actually is. This optimism transfers to the f function as well: *If h is admissible, $f(n)$ never overestimates the actual cost of the best solution through n.* Best-first search using f as the evaluation function and an admissible h function is known as **A* search**

A* SEARCH

function A*-SEARCH(*problem*) **returns** a solution or failure
  return BEST-FIRST-SEARCH(*problem*, $g + h$)

Perhaps the most 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. In Figure 4.4, we show the first few steps of an A* search for Bucharest using the h_{SLD} heuristic. Notice that the A* search prefers to expand from Rimnicu Vilcea rather than from Fagaras. Even though Fagaras is closer to Bucharest, the path taken to get to Fagaras is not as *efficient* in getting close to Bucharest as the path taken to get to Rimnicu. The reader may wish to continue this example to see what happens next.

The behavior of A* search

Before we prove the completeness and optimality of A*, it will be useful to present an intuitive picture of how it works. A picture is not a substitute for a proof, but it is often easier to remember and can be used to generate the proof on demand. First, a preliminary observation: if you examine the search trees in Figure 4.4, you will notice an interesting phenomenon. Along any path from the root, the f-cost never decreases. This is no accident. It holds true for almost all admissible heuristics. A heuristic for which it holds is said to exhibit **monotonicity**.[1]

MONOTONICITY

If the heuristic is one of those odd ones that is not monotonic, it turns out we can make a minor correction that restores monotonicity. Let us consider two nodes n and n', where n is the parent of n'. Now suppose, for example, that $g(n) = 3$ and $h(n) = 4$. Then $f(n) = g(n)+h(n) = 7$—that is, we know that the true cost of a solution path through n is at least 7. Suppose also that $g(n') = 4$ and $h(n') = 2$, so that $f(n') = 6$. Clearly, this is an example of a nonmonotonic heuristic. Fortunately, from the fact that *any path through n' is also a path through n*, we can see that the value of 6 is meaningless, because we already know the true cost is at least 7. Thus, we should

[1] It can be proved (Pearl, 1984) that a heuristic is monotonic if and only if it obeys the triangle inequality. The triangle inequality says that two sides of a triangle cannot add up to less than the third side (see Exercise 4.7). Of course, straight-line distance obeys the triangle inequality and is therefore monotonic.

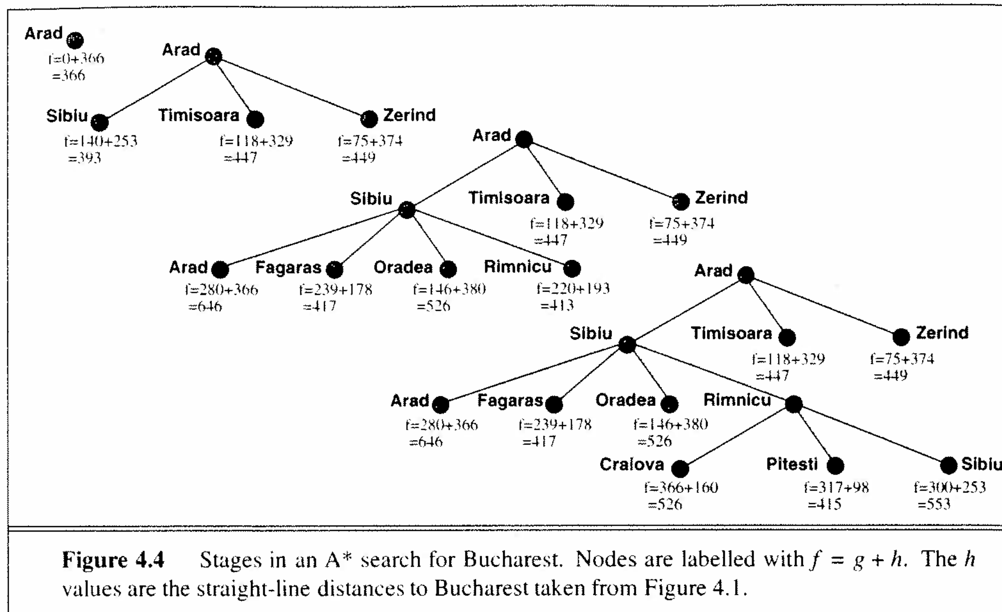


Figure 4.4 Stages in an A* search for Bucharest. Nodes are labelled with $f = g + h$. The h values are the straight-line distances to Bucharest taken from Figure 4.1.

check, each time we generate a new node, to see if its f-cost is less than its parent's f-cost; if it is, we use the parent's f-cost instead:

$$f(n') = max(f(n), g(n') + h(n')).$$

In this way, we ignore the misleading values that may occur with a nonmonotonic heuristic. This equation is called the **pathmax** equation. If we use it, then f will always be nondecreasing along any path from the root, provided h is admissible.

PATHMAX

The purpose of making this observation is to legitimize a certain picture of what A* does. If f never decreases along any path out from the root, we can conceptually draw **contours** in the state space. Figure 4.5 shows an example. Inside the contour labelled 400, all nodes have $f(n)$ less than or equal to 400, and so on. Then, because A* expands the leaf node of lowest f, we can see that an A* search fans out from the start node, adding nodes in concentric bands of increasing f-cost.

CONTOURS

With uniform-cost search (A* search using $h = 0$), 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 we define f^* to be the cost of the optimal solution path, then we can say the following:

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

Intuitively, it is obvious that the first solution found must be the optimal one, because nodes in all subsequent contours will have higher f-cost, and thus higher g-cost (because all goal states have $h(n) = 0$). Intuitively, it is also obvious that A* search is complete. As we add bands of

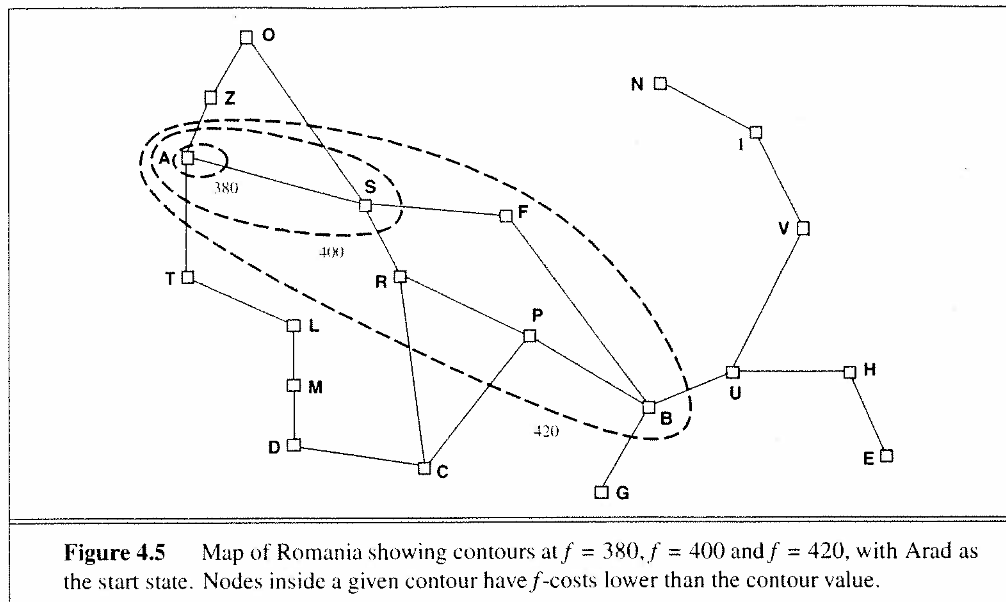


Figure 4.5 Map of Romania showing contours at $f = 380$, $f = 400$ and $f = 420$, with Arad as the start state. Nodes inside a given contour have f-costs lower than the contour value.

increasing f, we must eventually reach a band where f is equal to the cost of the path to a goal state. We will turn these intuitions into proofs in the next subsection.

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*. We can explain this as follows: any algorithm that *does not* expand all nodes in the contours between the root and the goal contour runs the risk of missing the optimal solution. A long and detailed proof of this result appears in Dechter and Pearl (1985).

Proof of the optimality of A*

Let G be an optimal goal state, with path cost f^*. Let G_2 be a suboptimal goal state, that is, a goal state with path cost $g(G_2) > f^*$. The situation we imagine is that A* has selected G_2 from the queue. Because G_2 is a goal state, this would terminate the search with a suboptimal solution (Figure 4.6). We will show that this is not possible.

Consider a node n that is currently a leaf node on an optimal path to G (there must be some such node, unless the path has been completely expanded—in which case the algorithm would have returned G). Because h is admissible, we must have

$$f^* \geq f(n).$$

Furthermore, if n is not chosen for expansion over G_2, we must have

$$f(n) \geq f(G_2).$$

Combining these two together, we get

$$f^* \geq f(G_2).$$

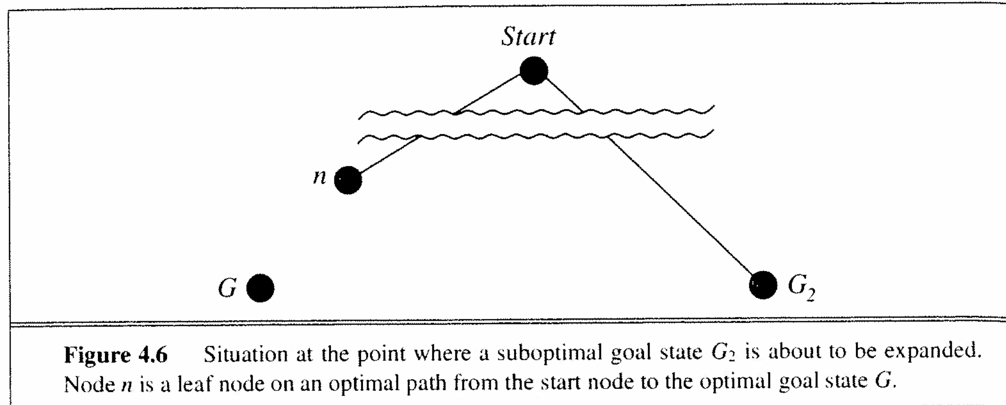


Figure 4.6 Situation at the point where a suboptimal goal state G_2 is about to be expanded. Node n is a leaf node on an optimal path from the start node to the optimal goal state G.

But because G_2 is a goal state, we have $h(G_2) = 0$; hence $f(G_2) = g(G_2)$. Thus, we have proved, from our assumptions, that

$$f^* \geq g(G_2).$$

This contradicts the assumption that G_2 is suboptimal, so it must be the case that A* never selects a suboptimal goal for expansion. Hence, because it only returns a solution after selecting it for expansion, A* is an optimal algorithm.

Proof of the completeness of A*

We said before that because A* expands nodes in order of increasing f, it must eventually expand to reach a goal state. This is true, of course, unless there are infinitely many nodes with $f(n) < f^*$. The only way there could be an infinite number of nodes is either (a) there is a node with an infinite branching factor, or (b) there is a path with a finite path cost but an infinite number of nodes along it.[2]

LOCALLY FINITE GRAPHS

Thus, the correct statement is that A* is complete on **locally finite graphs** (graphs with a finite branching factor) provided there is some positive constant δ such that every operator costs at least δ.

Complexity of A*

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

[2] Zeno's paradox, which purports to show that a rock thrown at a tree will never reach it, provides an example that violates condition (b). The paradox is created by imagining that the trajectory is divided into a series of phases, each of which covers half the remaining distance to the tree; this yields an infinite sequence of steps with a finite total cost.

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. Of course, the use of a good heuristic still provides enormous savings compared to an uninformed search. In the next section, 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, A* usually runs out of space long before it runs out of time. Recently developed algorithms have overcome the space problem without sacrificing optimality or completeness. These are discussed in Section 4.3.

4.2 HEURISTIC FUNCTIONS

So far we have seen just one example of a heuristic: straight-line distance for route-finding problems. In this section, we will look at heuristics for the 8-puzzle. This will shed light on the nature of heuristics in general.

The 8-puzzle was one of the earliest heuristic search problems. As mentioned in Section 3.3, 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).

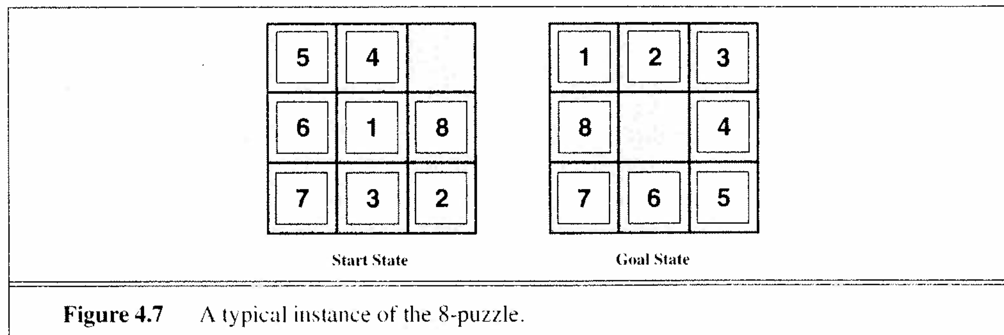


Figure 4.7 A typical instance of the 8-puzzle.

The 8-puzzle is just the right level of difficulty to be interesting. A typical solution is about 20 steps, although this of course varies depending on the initial state. 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 20 would look at about $3^{20} = 3.5 \times 10^9$ states. By keeping track of repeated states, we could cut this down drastically, because there are only $9! = 362,880$ different arrangements of 9 squares. This is still a large number of states, so the next order of business is to find a good

Artificial Intelligence
A Modern Approach

Stuart J. Russell and Peter Norvig

Contributing writers:
John F. Canny, Jitendra M. Malik, Douglas D. Edwards

Prentice Hall, Upper Saddle River, New Jersey 07458

Library of Congress Cataloging-in-Publication Data

Russell, Stuart J. (Stuart Jonathan)
Artificial intelligence : a modern approach / Stuart Russell, Peter Norvig.
p. cm.
Includes bibliographical references and index.
ISBN 0-13-103805-2
1. Artificial intelligence I. Norvig, Peter. II. Title.
Q335.R86 1995
006.3--dc20 94-36444
CIP

Publisher: Alan Apt
Production Editor: Mona Pompili
Developmental Editor: Sondra Chavez
Cover Designers: Stuart Russell and Peter Norvig
Production Coordinator: Lori Bulwin
Editorial Assistant: Shirley McGuire

© 1995 by Prentice-Hall, Inc.
A Simon & Schuster Company
Upper Saddle River, New Jersey 07458

The author and publisher of this book have used their best efforts in preparing this book. These efforts include the development, research, and testing of the theories and programs to determine their effectiveness. The author and publisher shall not be liable in any event for incidental or consequential damages in connection with, or arising out of, the furnishing, performance, or use of these programs.

All rights reserved. No part of this book may be reproduced, in any form or by any means, without permission in writing from the publisher.

Printed in the United States of America

10 9 8 7 6 5 4

ISBN 0-13-103805-2

Prentice-Hall International (UK) Limited, *London*
Prentice-Hall of Australia Pty. Limited, *Sydney*
Prentice-Hall Canada, Inc., *Toronto*
Prentice-Hall Hispanoamericana, S.A., *Mexico*
Prentice-Hall of India Private Limited, *New Delhi*
Prentice-Hall of Japan, Inc., *Tokyo*
Simon & Schuster Asia Pte. Ltd., *Singapore*
Editora Prentice-Hall do Brasil, Ltda., *Rio de Janeiro*