Chapter 3

States and Searching

> *Have you ever watched a crab on the shore crawling backward in search of the Atlantic Ocean, and missing? That's the way the mind of man operates.*
>
> – H. L. Mencken (1880–1956)

The previous chapter discussed how an agent perceives and acts, but not how its goals affect its actions. An agent could be programmed to act in the world to achieve a fixed set of goals, but then it may not adapt to changing goals and so would not be intelligent. Alternatively, an agent could reason about its abilities and its goals to determine what to do. This chapter shows how the problem of an agent deciding what to do can be cast as the problem of searching to find a path in a graph, and it presents a number of ways that such problems can be solved on a computer. As Mencken suggests in the quote above, the mind uses search to solve problems, although not always successfully.

3.1 Problem Solving as Search

In the simplest case of an agent reasoning about what it should do, the agent has a state-based model of the world, with no uncertainty and with goals to achieve. This is either a flat (non-hierarchical) representation or a single level of a hierarchy. The agent can determine how to achieve its goals by searching in its representation of the world state space for a way to get from its current state to a goal state. It can find a sequence of actions that will achieve its goal before it has to act in the world.

This problem can be abstracted to the mathematical problem of finding a path from a start node to a goal node in a directed graph. Many other problems can also be mapped to this abstraction, so it is worthwhile to consider

this level of abstraction. Most of this chapter explores various algorithms for finding such paths.

This notion of search is computation inside the agent. It is different from searching in the world, when it may have to act in the world, for example, an agent searching for its keys, lifting up cushions, and so on. It is also different from searching the web, which involves searching for information. Searching in this chapter means searching in an internal representation for a path to a goal.

The idea of search is straightforward: the agent constructs a set of potential partial solutions to a problem that can be checked to see if they truly are solutions or if they could lead to solutions. Search proceeds by repeatedly selecting a partial solution, stopping if it is a path to a goal, and otherwise extending it by one more arc in all possible ways.

Search underlies much of artificial intelligence. When an agent is given a problem, it is usually given only a description that lets it recognize a solution, not an algorithm to solve it. It has to search for a solution. The existence of NP-complete problems (page 170), with efficient means to recognize answers but no efficient methods for finding them, indicates that searching is, in many cases, a necessary part of solving problems.

It is often believed that humans are able to use intuition to jump to solutions to difficult problems. However, humans do not tend to solve general problems; instead they solve specific instances about which they may know much more than the underlying search space. Problems in which little structure exists or in which the structure cannot be related to the physical world are very difficult for humans to solve. The existence of public key encryption codes, where the search space is clear and the test for a solution is given – for which humans nevertheless have no hope of solving and computers cannot solve in a realistic time frame – demonstrates the difficulty of search.

The difficulty of search and the fact that humans are able to solve some search problems efficiently suggests that computer agents should exploit knowledge about special cases to guide them to a solution. This extra knowledge beyond the search space is **heuristic knowledge**. This chapter considers one kind of heuristic knowledge in the form of an estimate of the cost from a node to a goal.

3.2 State Spaces

One general formulation of intelligent action is in terms of **state space**. A **state** contains all of the information necessary to predict the effects of an action and to determine if it is a goal state. State-space searching assumes that

- the agent has perfect knowledge of the state space and can observe what state it is in (i.e., there is full observability);
- the agent has a set of actions that have known deterministic effects;

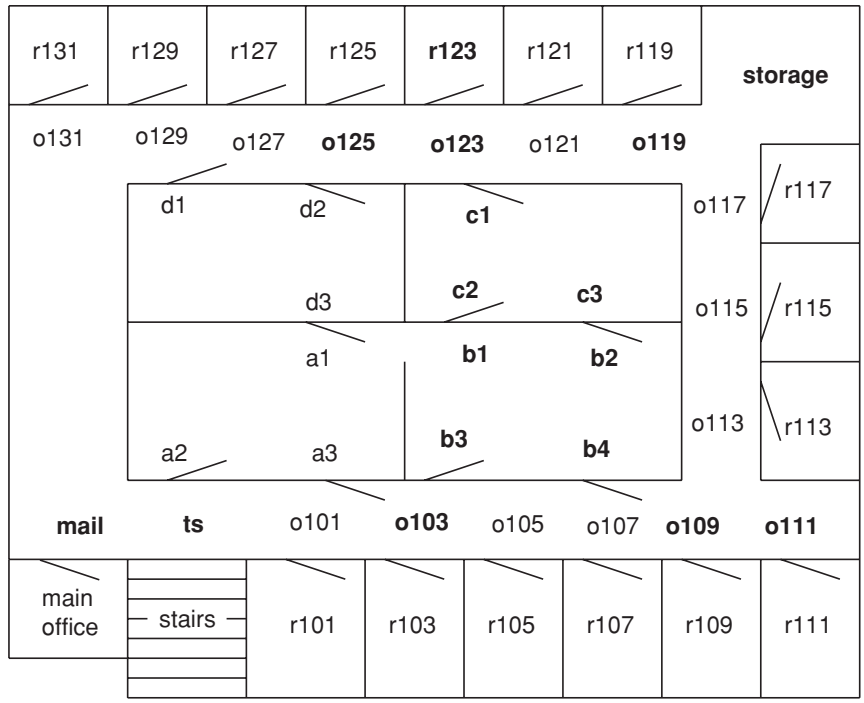

Figure 3.1: The delivery robot domain with interesting locations labeled

- some states are goal states, the agent wants to reach one of these goal states, and the agent can recognize a goal state; and
- a **solution** is a sequence of actions that will get the agent from its current state to a goal state.

Example 3.1 Consider the robot delivery domain and the task of finding a path from one location to another in Figure 3.1. This can be modeled as a state-space search problem, where the states are locations. Assume that the agent can use a lower-level controller to carry out the high-level action of getting from one location to a neighboring location. Thus, at this level of abstraction, the actions can involve deterministic traveling between neighboring locations.

An example problem is where the robot is outside room $r103$, at position $o103$, and the goal is to get to room $r123$. A solution is a sequence of actions that will get the robot to room $r123$.

Example 3.2 In a more complicated example, the delivery robot may have a number of parcels to deliver to various locations. In this case, the state may consist of the location of the robot, the parcels the robot is carrying, and the locations of the other parcels. The possible actions may be for the robot to move, to pick up parcels that are at the same location as the robot, or to put down whatever parcels it is carrying. A goal state may be one in which some specified

parcels are at their desired locations. There may be many goal states because we may not care where the robot is or where some of the other parcels are.

Notice that this representation has ignored many details, for example, how the robot is carrying the parcels (which may affect whether it can carry other parcels), the battery level of the robot, whether the parcels are fragile or damaged, and the color of the floor. By not having these as part of the state space, we assume that these details are not relevant to the problem at hand.

Example 3.3 In a tutoring system, a state may consist of the set of topics that the student knows. The action may be teaching a particular lesson, and the result of a teaching action may be that the student knows the topic of the lesson as long as the student knows the topics that are prerequisites for the lesson being taught. The aim is for the student to know some particular set of topics.

If the effect of teaching also depends on the aptitude of the student, this detail must be part of the state space, too. We do not have to model what the student is carrying if that does not affect the result of actions or whether the goal is achieved.

A **state-space problem** consists of

- a set of states;
- a distinguished set of states called the **start states**;
- a set of actions available to the agent in each state;
- an **action function** that, given a state and an action, returns a new state;
- a set of goal states, often specified as a Boolean function, $goal(s)$, that is true when s is a goal state; and
- a criterion that specifies the quality of an acceptable solution. For example, any sequence of actions that gets the agent to the goal state may be acceptable, or there may be costs associated with actions and the agent may be required to find a sequence that has minimal total cost. This is called an **optimal** solution. Alternatively, it may be satisfied with any solution that is within 10% of optimal.

This framework is extended in subsequent chapters to include cases where an agent can exploit the internal features of the states, where the state is not fully observable (e.g., the robot does not know where the parcels are, or the teacher does not know the aptitude of the student), where the actions are stochastic (e.g., the robot may overshoot, or the student perhaps does not learn a topic that is taught), and where complex preferences exist in terms of rewards and punishments, not just goal states.

3.3 Graph Searching

In this chapter, we abstract the general mechanism of searching and present it in terms of searching for paths in directed graphs. To solve a problem, first define the underlying search space and then apply a search algorithm to that

search space. Many problem-solving tasks can be transformed into the problem of finding a path in a graph. Searching in graphs provides an appropriate level of abstraction within which to study simple problem solving independent of a particular domain.

A (directed) graph consists of a set of nodes and a set of directed arcs between nodes. The idea is to find a path along these arcs from a start node to a goal node.

The abstraction is necessary because there may be more than one way to represent a problem as a graph. Whereas the examples in this chapter are in terms of state-space searching, where nodes represent states and arcs represent actions, future chapters consider different ways to represent problems as graphs to search.

3.3.1 Formalizing Graph Searching

A directed **graph** consists of

- a set N of **nodes** and
- a set A of ordered pairs of nodes called **arcs**.

In this definition, a node can be anything. All this definition does is constrain arcs to be ordered pairs of nodes. There can be infinitely many nodes and arcs. We do not assume that the graph is represented explicitly; we require only a procedure that can generate nodes and arcs as needed.

The arc $\langle n_1, n_2 \rangle$ is an **outgoing arc** from n_1 and an **incoming arc** to n_2.

A node n_2 is a **neighbor** of n_1 if there is an arc from n_1 to n_2; that is, if $\langle n_1, n_2 \rangle \in A$. Note that being a neighbor does not imply symmetry; just because n_2 is a neighbor of n_1 does not mean that n_1 is necessarily a neighbor of n_2. Arcs may be **labeled**, for example, with the action that will take the agent from one state to another.

A **path** from node s to node g is a sequence of nodes $\langle n_0, n_1, \ldots, n_k \rangle$ such that $s = n_0$, $g = n_k$, and $\langle n_{i-1}, n_i \rangle \in A$; that is, there is an arc from n_{i-1} to n_i for each i. Sometimes it is useful to view a path as the sequence of arcs, $\langle n_o, n_1 \rangle$, $\langle n_1, n_2 \rangle, \ldots, \langle n_{k-1}, n_k \rangle$, or a sequence of labels of these arcs.

A **cycle** is a nonempty path such that the end node is the same as the start node – that is, a cycle is a path $\langle n_0, n_1, \ldots, n_k \rangle$ such that $n_0 = n_k$ and $k \neq 0$. A directed graph without any cycles is called a **directed acyclic graph** (**DAG**). This should probably be an **acyclic directed graph**, because it is a directed graph that happens to be acyclic, not an acyclic graph that happens to be directed, but DAG sounds better than ADG!

A **tree** is a DAG where there is one node with no incoming arcs and every other node has exactly one incoming arc. The node with no incoming arcs is called the **root** of the tree and nodes with no outgoing arcs are called **leaves**.

To encode problems as graphs, one set of nodes is referred to as the **start nodes** and another set is called the **goal nodes**. A **solution** is a path from a start node to a goal node.

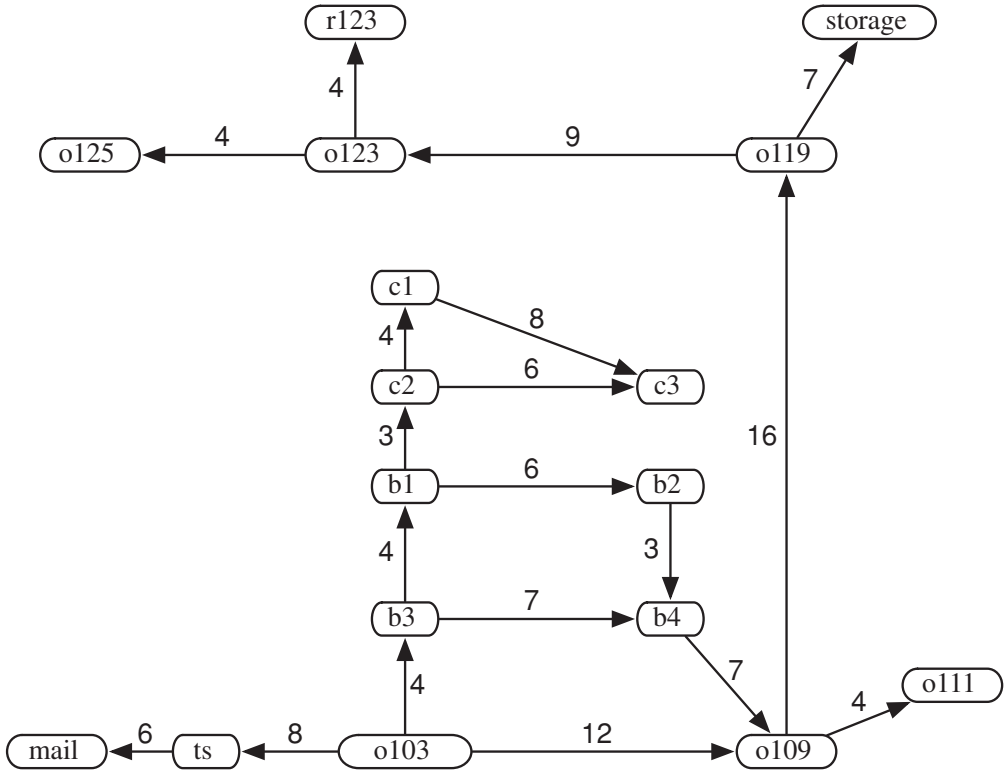

Figure 3.2: A graph with arc costs for the delivery robot domain

Sometimes there is a **cost** – a positive number – associated with arcs. We write the cost of arc $\langle n_i, n_j \rangle$ as $cost(\langle n_i, n_j \rangle)$. The costs of arcs induces a cost of paths.

Given a path $p = \langle n_0, n_1, \ldots, n_k \rangle$, the cost of path p is the sum of the costs of the arcs in the path:

$$cost(p) = cost(\langle n_0, n_1 \rangle) + \cdots + cost(\langle n_{k-1}, n_k \rangle)$$

An **optimal solution** is one of the least-cost solutions; that is, it is a path p from a start node to a goal node such that there is no path p' from a start node to a goal node where $cost(p') < cost(p)$.

Example 3.4 Consider the problem of the delivery robot finding a path from location $o103$ to location $r123$ in the domain depicted in Figure 3.1 (page 73). In this figure, the interesting locations are named. For simplicity, we consider only the locations written in bold and we initially limit the directions that the robot can travel. Figure 3.2 shows the resulting graph where the nodes represent locations and the arcs represent possible single steps between locations. In this figure, each arc is shown with the associated cost of getting from one location to the next.

In this graph, the nodes are $N = \{mail, ts, o103, b3, o109, \ldots\}$ and the arcs are $A = \{\langle ts, mail\rangle, \langle o103, ts\rangle, \langle o103, b3\rangle, \langle o103, o109\rangle, \ldots\}$. Node $o125$ has no neighbors. Node ts has one neighbor, namely $mail$. Node $o103$ has three neighbors, namely ts, $b3$, and $o109$.

There are three paths from $o103$ to $r123$:

$\langle o103, o109, o119, o123, r123\rangle$
$\langle o103, b3, b4, o109, o119, o123, r123\rangle$
$\langle o103, b3, b1, b2, b4, o109, o119, o123, r123\rangle$

If $o103$ were a start node and $r123$ were a goal node, each of these three paths would be a solution to the graph-searching problem.

In many problems the search graph is not given explicitly; it is dynamically constructed as needed. All that is required for the search algorithms that follow is a way to generate the neighbors of a node and to determine if a node is a goal node.

The **forward branching factor** of a node is the number of arcs leaving the node. The **backward branching factor** of a node is the number of arcs entering the node. These factors provide measures of the complexity of graphs. When we discuss the time and space complexity of the search algorithms, we assume that the branching factors are bounded from above by a constant.

Example 3.5 In the graph of Figure 3.2, the forward branching factor of node $o103$ is three; there are three arcs coming out of node $o103$. The backward branching factor of node $o103$ is zero; there are no arcs coming into node $o103$. The forward branching factor of $mail$ is zero and the backward branching factor of $mail$ is one. The forward branching factor of node $b3$ is two and the backward branching factor of $b3$ is one.

The branching factor is important because it is a key component in the size of the graph. If the forward branching factor for each node is b, and the graph is a tree, there are b^n nodes that are n arcs away from any node.

3.4 A Generic Searching Algorithm

This section describes a generic algorithm to search for a solution path in a graph. The algorithm is independent of any particular search strategy and any particular graph.

The intuitive idea behind the generic search algorithm, given a graph, a set of start nodes, and a set of goal nodes, is to incrementally explore paths from the start nodes. This is done by maintaining a **frontier** (or **fringe**) of paths from the start node that have been explored. The frontier contains all of the paths that could form initial segments of paths from a start node to a goal node. (See Figure 3.3 (on the next page), where the frontier is the set of paths to the gray shaded nodes.) Initially, the frontier contains trivial paths containing no

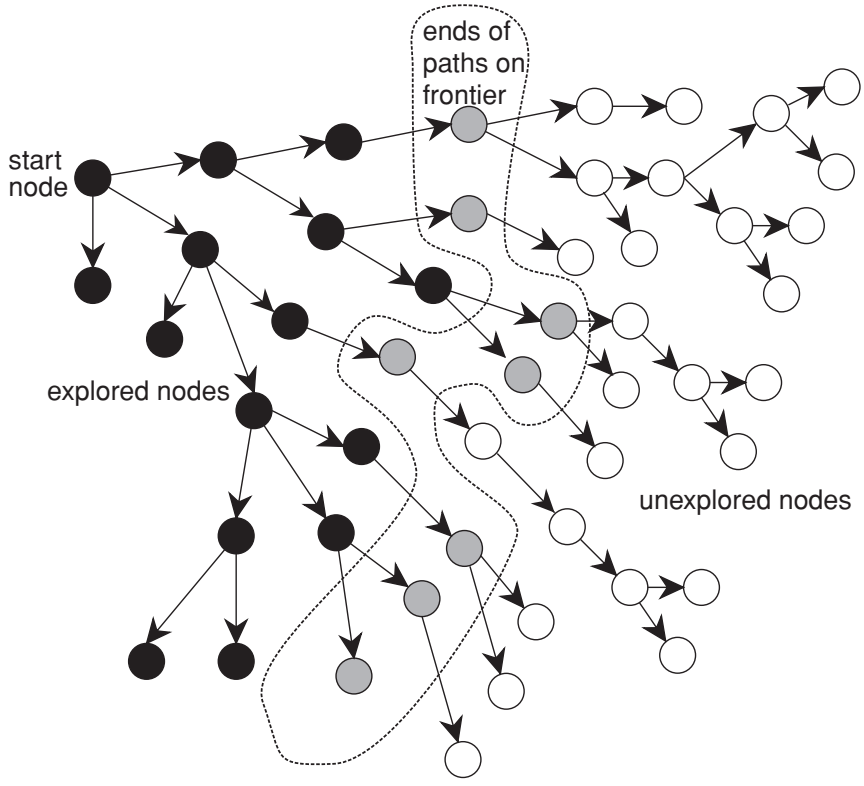

Figure 3.3: Problem solving by graph searching

arcs from the start nodes. As the search proceeds, the frontier expands into the unexplored nodes until a goal node is encountered. To expand the frontier, the searcher selects and removes a path from the frontier, extends the path with each arc leaving the last node, and adds these new paths to the frontier. A search strategy defines which element of the frontier is selected at each step.

The generic search algorithm is shown in Figure 3.4. Initially, the frontier is the set of empty paths from start nodes. At each step, the algorithm advances the frontier by removing a path $\langle s_0, \ldots, s_k \rangle$ from the frontier. If $goal(s_k)$ is true (i.e., s_k is a goal node), it has found a solution and returns the path that was found, namely $\langle s_0, \ldots, s_k \rangle$. Otherwise, the path is extended by one more arc by finding the neighbors of s_k. For every neighbor s of s_k, the path $\langle s_0, \ldots, s_k, s \rangle$ is added to the frontier. This step is known as **expanding** the node s_k.

This algorithm has a few features that should be noted:

- The selection of a path at line 13 is non-deterministic. The choice of path that is selected can affect the efficiency; see the box on page 170 for more details on our use of "select". A particular search strategy will determine which path is selected.
- It is useful to think of the *return* at line 15 as a temporary return; another path to a goal can be searched for by continuing to line 16.

```
1: procedure Search(G, S, goal)
2:     Inputs
3:         G: graph with nodes N and arcs A
4:         S: set of start nodes
5:         goal: Boolean function of states
6:     Output
7:         path from a member of S to a node for which goal is true
8:         or ⊥ if there are no solution paths
9:     Local
10:        Frontier: set of paths
11:    Frontier ← {⟨s⟩ : s ∈ S}
12:    while Frontier ≠ {} do
13:        select and remove ⟨s_0, ..., s_k⟩ from Frontier
14:        if goal(s_k) then
15:            return ⟨s_0, ..., s_k⟩
16:        Frontier ← Frontier ∪ {⟨s_0, ..., s_k, s⟩ : ⟨s_k, s⟩ ∈ A}
17:    return ⊥
```

Figure 3.4: Generic graph searching algorithm

- If the procedure returns $\bot$, no solutions exist (or there are no remaining solutions if the proof has been retried).
- The algorithm only tests if a path ends in a goal node *after* the path has been selected from the frontier, not when it is added to the frontier. There are two main reasons for this. Sometimes a very costly arc exists from a node on the frontier to a goal node. The search should not always return the path with this arc, because a lower-cost solution may exist. This is crucial when the least-cost path is required. The second reason is that it may be expensive to determine whether a node is a goal node.

If the path chosen does not end at a goal node and the node at the end has no neighbors, extending the path means removing the path. This outcome is reasonable because this path could not be part of a path from a start node to a goal node.

3.5 Uninformed Search Strategies

A problem determines the graph and the goal but not which path to select from the frontier. This is the job of a search strategy. A search strategy specifies which paths are selected from the frontier. Different strategies are obtained by modifying how the selection of paths in the frontier is implemented.

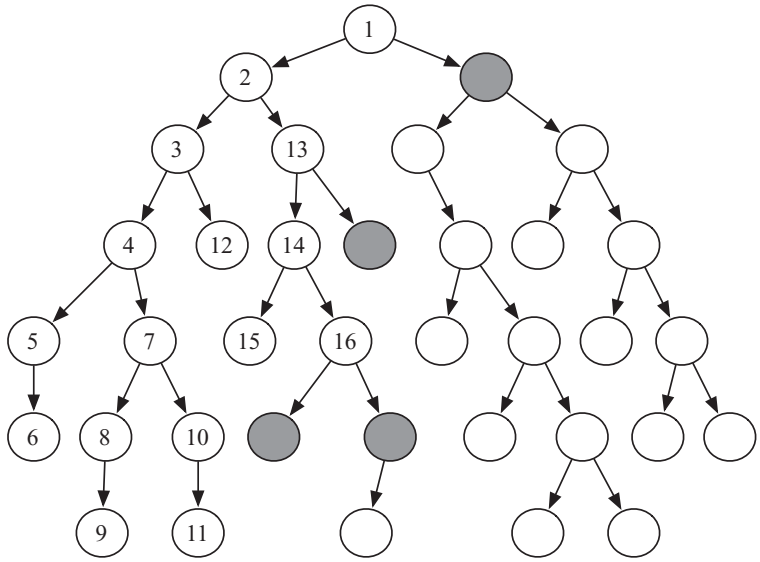

Figure 3.5: The order nodes are expanded in depth-first search

This section presents three **uninformed search strategies** that do not take into account the location of the goal. Intuitively, these algorithms ignore where they are going until they find a goal and report success.

3.5.1 Depth-First Search

The first strategy is **depth-first search**. In depth-first search, the frontier acts like a last-in first-out **stack**. The elements are added to the stack one at a time. The one selected and taken off the frontier at any time is the last element that was added.

Example 3.6 Consider the tree-shaped graph in Figure 3.5. Suppose the start node is the root of the tree (the node at the top) and the nodes are ordered from left to right so that the leftmost neighbor is added to the stack last. In depth-first search, the order in which the nodes are expanded does not depend on the location of the goals. The first sixteen nodes expanded are numbered in order of expansion in Figure 3.5. The shaded nodes are the nodes at the ends of the paths on the frontier after the first sixteen steps.

Notice how the first six nodes expanded are all in a single path. The sixth node has no neighbors. Thus, the next node that is expanded is a child of the lowest ancestor of this node that has unexpanded children.

Implementing the frontier as a stack results in paths being pursued in a depth-first manner – searching one path to its completion before trying an alternative path. This method is said to involve **backtracking**: The algorithm selects a first alternative at each node, and it *backtracks* to the next alternative

when it has pursued all of the paths from the first selection. Some paths may be infinite when the graph has cycles or infinitely many nodes, in which case a depth-first search may never stop.

This algorithm does not specify the order in which the neighbors are added to the stack that represents the frontier. The efficiency of the algorithm is sensitive to this ordering.

Example 3.7 Consider depth-first search from $o103$ in the graph given in Figure 3.2. The only goal node is $r123$. In this example, the frontier is shown as a list of paths with the top of the stack at the beginning of the list.

Initially, the frontier contains the trivial path $\langle o103 \rangle$.

At the next stage, the frontier contains the following paths:

$$[\langle o103, ts \rangle , \langle o103, b3 \rangle , \langle o103, o109 \rangle].$$

Next, the path $\langle o103, ts \rangle$ is selected because it is at the top of the stack. It is removed from the frontier and replaced by extending it by one arc, resulting in the frontier

$$[\langle o103, ts, mail \rangle , \langle o103, b3 \rangle , \langle o103, o109 \rangle].$$

Next, the first path $\langle o103, ts, mail \rangle$ is removed from the frontier and is replaced by the set of paths that extend it by one arc, which is the empty set because *mail* has no neighbors. Thus, the resulting frontier is

$$[\langle o103, b3 \rangle , \langle o103, o109 \rangle].$$

At this stage, the path $\langle o103, b3 \rangle$ is the top of the stack. Notice what has happened: depth-first search has pursued all paths from *ts* and, when all of these paths were exhausted (there was only one), it backtracked to the next element of the stack. Next, $\langle o103, b3 \rangle$ is selected and is replaced in the frontier by the paths that extend it by one arc, resulting in the frontier

$$[\langle o103, b3, b1 \rangle , \langle o103, b3, b4 \rangle , \langle o103, o109 \rangle].$$

Then $\langle o103, b3, b1 \rangle$ is selected from the frontier and is replaced by all one-arc extensions, resulting in the frontier

$$[\langle o103, b3, b1, c2 \rangle , \langle o103, b3, b1, b2 \rangle , \langle o103, b3, b4 \rangle , \langle o103, o109 \rangle].$$

Now the first path is selected from the frontier and is extended by one arc, resulting in the frontier

$$[\langle o103, b3, b1, c2, c3 \rangle , \langle o103, b3, b1, c2, c1 \rangle , \langle o103, b3, b1, b2 \rangle , \langle o103, b3, b4 \rangle , \langle o103, o109 \rangle].$$

Node $c3$ has no neighbors, and thus the search "backtracks" to the last alternative that has not been pursued, namely to the path to $c1$.

Suppose $\langle n_0, \ldots, n_k \rangle$ is the selected path in the frontier. Then every other element of the frontier is of the form $\langle n_0, \ldots, n_i, m \rangle$, for some index $i < n$ and some node m that is a neighbor of n_i; that is, it follows the selected path for a number of arcs and then has exactly one extra node.

To understand the complexity (see the box on page 83) of depth-first search, consider an analogy using family trees, where the neighbors of a node correspond to its children in the tree. At the root of the tree is a start node. A branch down this tree corresponds to a path from a start node. Consider the node at the end of path at the top of the frontier. The other elements of the frontier correspond to children of ancestors of that node – the "uncles," "great uncles," and so on. If the branching factor is b and the first element of the list has length n, there can be at most $n \times (b-1)$ other elements of the frontier. These elements correspond to the $b-1$ alternative paths from each node. Thus, for depth-first search, the space used is linear in the depth of the path length from the start to a node.

If there is a solution on the first branch searched, then the time complexity is linear in the length of the path; it considers only those elements on the path, along with their siblings. The worst-case complexity is infinite. Depth-first search can get trapped on infinite branches and never find a solution, even if one exists, for infinite graphs or for graphs with loops. If the graph is a finite tree, with the forward branching factor bounded by b and depth n, the worst-case complexity is $O(b^n)$.

Example 3.8 Consider a modification of the delivery graph, in which the agent has much more freedom in moving between locations. The new graph is presented in Figure 3.6 (page 84). An infinite path leads from *ts* to *mail*, back to *ts*, back to *mail*, and so forth. As presented, depth-first search follows this path forever, never considering alternative paths from *b*3 or *o*109. The frontiers for the first five iterations of the path-finding search algorithm using depth-first search are

$$[\langle o103\rangle]$$
$$[\langle o103, ts\rangle, \langle o103, b3\rangle, \langle o103, o109\rangle]$$
$$[\langle o103, ts, mail\rangle, \langle o103, ts, o103\rangle, \langle o103, b3\rangle, \langle o103, o109\rangle]$$
$$[\langle o103, ts, mail, ts\rangle, \langle o103, ts, o103\rangle, \langle o103, b3\rangle, \langle o103, o109\rangle]$$
$$[\langle o103, ts, mail, ts, mail\rangle, \langle o103, ts, mail, ts, o103\rangle, \langle o103, ts, o103\rangle, \langle o103, b3\rangle, \langle o103, o109\rangle]$$

Depth-first search can be improved by not considering paths with cycles (page 93).

Because depth-first search is sensitive to the order in which the neighbors are added to the frontier, care must be taken to do it sensibly. This ordering can be done statically (so that the order of the neighbors is fixed) or dynamically (where the ordering of the neighbors depends on the goal).

Depth-first search is appropriate when either

- space is restricted;
- many solutions exist, perhaps with long path lengths, particularly for the case where nearly all paths lead to a solution; or
- the order of the neighbors of a node are added to the stack can be tuned so that solutions are found on the first try.

Comparing Algorithms

Algorithms (including search algorithms) can be compared on

- the time taken,
- the space used, and
- the quality or accuracy of the results.

The time taken, space used, and accuracy of an algorithm are a function of the inputs to the algorithm. Computer scientists talk about the **asymptotic complexity** of algorithms, which specifies how the time or space grows with the input size of the algorithm. An algorithm has time (or space) complexity $O(f(n))$ – read "big-oh of $f(n)$" – for input size n, where $f(n)$ is some function of n, if there exist constants n_0 and k such that the time, or space, of the algorithm is less than $k \times f(n)$ for all $n > n_0$. The most common types of functions are exponential functions such as 2^n, 3^n, or 1.015^n; polynomial functions such as n^5, n^2, n, or $n^{1/2}$; and logarithmic functions, $\log n$. In general, exponential algorithms get worse more quickly than polynomial algorithms which, in turn, are worse than logarithmic algorithms.

An algorithm has time or space complexity $\Omega(f(n))$ for input size n if there exist constants n_0 and k such that the time or space of the algorithm is greater than $k \times f(n)$ for all $n > n_0$. An algorithm has time or space complexity $\Theta(n)$ if it has complexity $O(n)$ and $\Omega(n)$. Typically, you cannot give an $\Theta(f(n))$ complexity on an algorithm, because most algorithms take different times for different inputs. Thus, when comparing algorithms, one has to specify the class of problems that will be considered.

Algorithm A is better than B, using a measure of either time, space, or accuracy, could mean:

- the worst case of A is better than the worst case of B; or
- A works better in practice, or the average case of A is better than the average case of B, where you average over typical problems; or
- you characterize the class of problems for which A is better than B, so that which algorithm is better depends on the problem; or
- for every problem, A is better than B.

The worst-case asymptotic complexity is often the easiest to show, but it is usually the least useful. Characterizing the class of problems for which one algorithm is better than another is usually the most useful, if it is easy to determine which class a given problem is in. Unfortunately, this characterization is usually very difficult.

Characterizing when one algorithm is better than the other can be done either theoretically using mathematics or empirically by building implementations. Theorems are only as valid as the assumptions on which they are based. Similarly, empirical investigations are only as good as the suite of test cases and the actual implementations of the algorithms. It is easy to disprove a conjecture that one algorithm is better than another for some class of problems by showing a counterexample, but it is much more difficult to prove such a conjecture.

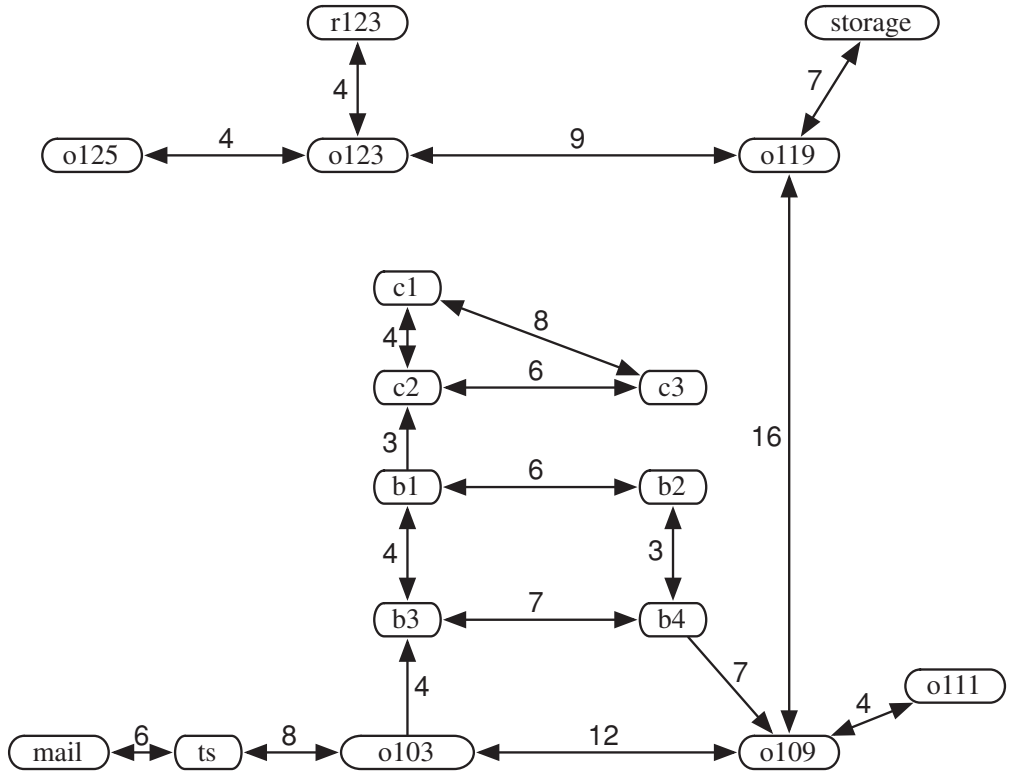

Figure 3.6: A graph, with cycles, for the delivery robot domain. Edges of the form $X \longleftrightarrow Y$ means there is an arc from X to Y and an arc from Y to X. That is, $\langle X, Y \rangle \in A$ and $\langle Y, X \rangle \in A$.

It is a poor method when

- it is possible to get caught in infinite paths; this occurs when the graph is infinite or when there are cycles in the graph; or
- solutions exist at shallow depth, because in this case the search may look at many long paths before finding the short solutions.

Depth-first search is the basis for a number of other algorithms, such as iterative deepening (page 95).

3.5.2 Breadth-First Search

In **breadth-first search** the frontier is implemented as a FIFO (first-in, first-out) queue. Thus, the path that is selected from the frontier is the one that was added earliest.

This approach implies that the paths from the start node are generated in order of the number of arcs in the path. One of the paths with the fewest arcs is selected at each stage.

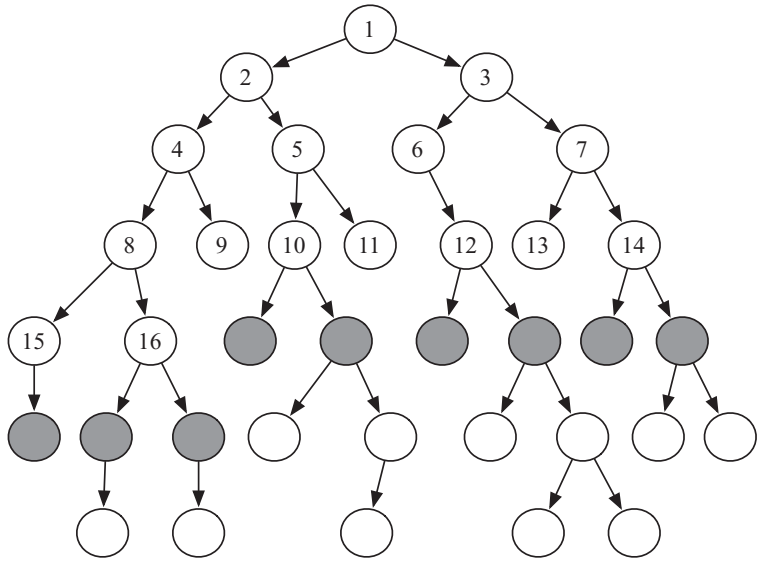

Figure 3.7: The order in which nodes are expanded in breadth-first search

Example 3.9 Consider the tree-shaped graph in Figure 3.7. Suppose the start node is the node at the top. In breadth-first search, as in depth-first search, the order in which the nodes are expanded does not depend on the location of the goal. The first sixteen nodes expanded are numbered in order of expansion in the figure. The shaded nodes are the nodes at the ends of the paths of the frontier after the first sixteen steps.

Example 3.10 Consider breadth-first search from $o103$ in the graph given in Figure 3.2 (page 76). The only goal node is $r123$. Initially, the frontier is $[\langle o103\rangle]$. This is extended by $o103$'s neighbors, making the frontier $[\langle o103, ts\rangle, \langle o103, b3\rangle, \langle o103, o109\rangle]$. These are the nodes one arc away from $o013$. The next three paths chosen are $\langle o103, ts\rangle$, $\langle o103, b3\rangle$, and $\langle o103, o109\rangle$, at which stage the frontier contains

$$[\langle o103, ts, mail\rangle, \langle o103, b3, b1\rangle, \langle o103, b3, b4\rangle, \langle o103, o109, o111\rangle, \langle o103, o109, o119\rangle].$$

These are the paths containing two arcs and starting at $o103$. These five paths are the next elements of the frontier chosen, at which stage the frontier contains the paths of three arcs away from $o103$, namely,

$$[\langle o103, b3, b1, c2\rangle, \langle o103, b3, b1, b2\rangle, \langle o103, b3, b4, o109\rangle, \langle o103, o109, o119, storage\rangle, \langle o103, o109, o119, o123\rangle].$$

Note how each of the paths on the frontier has approximately the same number of arcs. For breadth-first search, the number of arcs in the paths on the frontier always differs by, at most, one.

Suppose the branching factor of the search is b. If the first path of the frontier contains n arcs, there are at least b^{n-1} elements of the frontier. All of these paths contain n or $n+1$ arcs. Thus, both space and time complexities are exponential in the number of arcs of the path to a goal with the fewest arcs. This method is guaranteed, however, to find a solution if one exists and will find a solution with the fewest arcs.

Breadth-first search is useful when

- space is not a problem;
- you want to find the solution containing the fewest arcs;
- few solutions may exist, and at least one has a short path length; and
- infinite paths may exist, because it explores all of the search space, even with infinite paths.

It is a poor method when all solutions have a long path length or there is some heuristic knowledge available. It is not used very often because of its space complexity.

3.5.3 Lowest-Cost-First Search

When a non-unit cost is associated with arcs, we often want to find the solution that minimizes the total cost of the path. For example, for a delivery robot, costs may be distances and we may want a solution that gives the minimum total distance. Costs for a delivery robot may be resources required by the robot to carry out the action represented by the arc. The cost for a tutoring system may be the time and effort required by the students. In each of these cases, the searcher should try to minimize the total cost of the path found to reach the goal.

The search algorithms considered thus far are not guaranteed to find the minimum-cost paths; they have not used the arc cost information at all. Breadth-first search finds a solution with the fewest arcs first, but the distribution of arc costs may be such that a path of fewest arcs is not one of minimal cost.

The simplest search method that is guaranteed to find a minimum cost path is similar to breadth-first search; however, instead of expanding a path with the fewest number of arcs, it selects a path with the minimum cost. This is implemented by treating the frontier as a priority queue ordered by the *cost* function (page 76).

Example 3.11 Consider a lowest-cost-first search from $o103$ in the graph given in Figure 3.2 (page 76). The only goal node is $r123$. In this example, paths are denoted by the end node of the path. A subscript shows the cost of the path.

Initially, the frontier is $[o103_0]$. At the next stage it is $[b3_4, ts_8, o109_{12}]$. The path to $b3$ is selected, with the resulting frontier

$$[b1_8, ts_8, b4_{11}, o109_{12}].$$

The path to $b1$ is then selected, resulting in frontier

$$[ts_8, c2_{11}, b4_{11}, o109_{12}, b2_{14}].$$

Then the path to ts is selected, and the resulting frontier is

$$[c2_{11}, b4_{11}, o109_{12}, mail_{14}, b2_{14}].$$

Then $c2$ is selected, and so forth. Note how the lowest-cost-first search grows many paths incrementally, always expanding the path with lowest cost.

If the costs of the arcs are bounded below by a positive constant and the branching factor is finite, the lowest-cost-first search is guaranteed to find an optimal solution – a solution with lowest path cost – if a solution exists. Moreover, the first path to a goal that is found is a path with least cost. Such a solution is optimal, because the algorithm generates paths from the start in order of path cost. If a better path existed than the first solution found, it would have been selected from the frontier earlier.

The bounded arc cost is used to guarantee the lowest-cost search will find an optimal solution. Without such a bound there can be infinite paths with a finite cost. For example, there could be nodes $n_0, n_1, n_2, \ldots$ with an arc $\langle n_{i-1}, n_i \rangle$ for each $i > 0$ with cost $1/2^i$. Infinitely many paths of the form $\langle n_0, n_1, n_2, \ldots, n_k \rangle$ exist, all of which have a cost of less than 1. If there is an arc from n_0 to a goal node with a cost greater than or equal to 1, it will never be selected. This is the basis of Zeno's paradoxes that Aristotle wrote about more than 2,300 years ago.

Like breadth-first search, lowest-cost-first search is typically exponential in both space and time. It generates *all* paths from the start that have a cost less than the cost of the solution.

3.6 Heuristic Search

All of the search methods in the preceding section are uninformed in that they did not take into account the goal. They do not use any information about where they are trying to get to unless they happen to stumble on a goal. One form of heuristic information about which nodes seem the most promising is a heuristic function $h(n)$, which takes a node n and returns a non-negative real number that is an estimate of the path cost from node n to a goal node. The function $h(n)$ is an *underestimate* if $h(n)$ is less than or equal to the actual cost of a lowest-cost path from node n to a goal.

The heuristic function is a way to inform the search about the direction to a goal. It provides an informed way to guess which neighbor of a node will lead to a goal.

There is nothing magical about a heuristic function. It must use only information that can be readily obtained about a node. Typically a trade-off exists between the amount of work it takes to derive a heuristic value for a node and

how accurately the heuristic value of a node measures the actual path cost from the node to a goal.

A standard way to derive a heuristic function is to solve a simpler problem and to use the actual cost in the simplified problem as the heuristic function of the original problem.

Example 3.12 For the graph of Figure 3.2 (page 76), the straight-line distance in the world between the node and the goal position can be used as the heuristic function.

The examples that follow assume the following heuristic function:

$$\begin{array}{rclrclrcl} h(mail) & = & 26 & h(ts) & = & 23 & h(o103) & = & 21 \\ h(o109) & = & 24 & h(o111) & = & 27 & h(o119) & = & 11 \\ h(o123) & = & 4 & h(o125) & = & 6 & h(r123) & = & 0 \\ h(b1) & = & 13 & h(b2) & = & 15 & h(b3) & = & 17 \\ h(b4) & = & 18 & h(c1) & = & 6 & h(c2) & = & 10 \\ h(c3) & = & 12 & h(storage) & = & 12 & & & \end{array}$$

This h function is an underestimate because the h value is less than or equal to the exact cost of a lowest-cost path from the node to a goal. It is the exact cost for node $o123$. It is very much an underestimate for node $b1$, which seems to be close, but there is only a long route to the goal. It is very misleading for $c1$, which also seems close to the goal, but no path exists from that node to the goal.

Example 3.13 Consider the delivery robot of Example 3.2 (page 73), where the state space includes the parcels to be delivered. Suppose the cost function is the total distance traveled by the robot to deliver all of the parcels. One possible heuristic function is the largest distance of a parcel from its destination. If the robot could only carry one parcel, a possible heuristic function is the sum of the distances that the parcels must be carried. If the robot could carry multiple parcels at once, this may not be an underestimate of the actual cost.

The h function can be extended to be applicable to (non-empty) paths. The heuristic value of a path is the heuristic value of the node at the end of the path. That is:

$$h(\langle n_o, \ldots, n_k \rangle) = h(n_k)$$

A simple use of a heuristic function is to order the neighbors that are added to the stack representing the frontier in depth-first search. The neighbors can be added to the frontier so that the best neighbor is selected first. This is known as **heuristic depth-first search**. This search chooses the locally best path, but it explores all paths from the selected path before it selects another path. Although it is often used, it suffers from the problems of depth-fist search.

Another way to use a heuristic function is to always select a path on the frontier with the lowest heuristic value. This is called **best-first search**. It

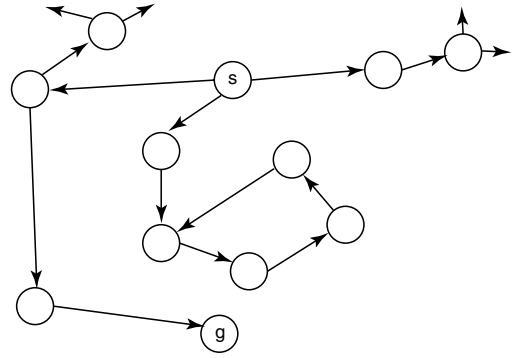

Figure 3.8: A graph that is bad for best-first search

usually does not work very well; it can follow paths that look promising because they are close to the goal, but the costs of the paths may keep increasing.

Example 3.14 Consider the graph shown in Figure 3.8, where the cost of an arc is its length. The aim is to find the shortest path from s to g. Suppose the Euclidean distance to the goal g is used as the heuristic function. A heuristic depth-first search will select the node below s and will never terminate. Similarly, because all of the nodes below s look good, a best-first search will cycle between them, never trying an alternate route from s.

3.6.1 A^* Search

A^* **search** is a combination of lowest-cost-first and best-first searches that considers both path cost and heuristic information in its selection of which path to expand. For each path on the frontier, A^* uses an estimate of the total path cost from a start node to a goal node constrained to start along that path. It uses $cost(p)$, the cost of the path found, as well as the heuristic function $h(p)$, the estimated path cost from the end of p to the goal.

For any path p on the frontier, define $f(p) = cost(p) + h(p)$. This is an estimate of the total path cost to follow path p then go to a goal node.

If n is the node at the end of path p, this can be depicted as follows:

$$\underbrace{\underbrace{start \xrightarrow{\text{actual}} n}_{cost(p)} \underbrace{\xrightarrow{\text{estimate}} goal}_{h(p)}}_{f(p)}$$

If $h(n)$ is an underestimate of the path costs from node n to a goal node, then $f(p)$ is an underestimate of a path cost of going from a start node to a goal node via p.

A^* is implemented by treating the frontier as a priority queue ordered by $f(p)$.

Example 3.15 Consider using A^* search in Example 3.4 (page 76) using the heuristic function of Figure 3.12 (page 88). In this example, the paths on the frontier are shown using the final node of the path, subscripted with the f-value of the path. The frontier is initially $[o103_{21}]$, because $h(o103) = 21$ and the cost of the path is zero. It is replaced by its neighbors, forming the frontier

$$[b3_{21}, ts_{31}, o109_{36}].$$

The first element represents the path $\langle o103, b3\rangle$; its f-value is $f(\langle o103, b3\rangle) = cost(\langle o103, b3\rangle) + h(b3) = 4 + 17 = 21$. Next $b3$ is selected and replaced by its neighbors, forming the frontier

$$[b1_{21}, b4_{29}, ts_{31}, o109_{36}].$$

Then the path to $b1$ is selected and replaced by its neighbors, forming the frontier

$$[c2_{21}, b4_{29}, b2_{29}, ts_{31}, o109_{36}].$$

Then the path to $c2$ is selected and replaced by its neighbors, forming

$$[c1_{21}, b4_{29}, b2_{29}, c3_{29}, ts_{31}, o109_{36}].$$

Up to this stage, the search has been continually exploring what seems to be the direct path to the goal. Next the path to $c1$ is selected and is replaced by its neighbors, forming the frontier

$$[b4_{29}, b2_{29}, c3_{29}, ts_{31}, c3_{35}, o109_{36}].$$

At this stage, there are two different paths to the node $c3$ on the queue. The path to $c3$ that does not go through $c1$ has a lower f-value than the one that does. Later (page 93), we consider the situation when one of these paths can be pruned.

There are two paths with the same f-value. The algorithm does not specify which is selected. Suppose the path to $b4$ is selected next and is replaced by its neighbors, forming

$$[b2_{29}, c3_{29}, ts_{31}, c3_{35}, o109_{36}, o109_{42}].$$

Then the path to $b2$ is selected and replaced by its neighbors, which is the empty set, forming

$$[c3_{29}, ts_{31}, c3_{35}, b4_{35}, o109_{36}, o109_{42}].$$

Then the path to $c3$ is removed and has no neighbors; thus, the new frontier is

$$[ts_{31}, c3_{35}, b4_{35}, o109_{36}, o109_{42}].$$

Note how A^* pursues many different paths from the start.

A lowest-cost path is eventually found. The algorithm is forced to try many different paths, because several of them temporarily seemed to have the lowest cost. It still does better than either lowest-cost-first search or best-first search.

Example 3.16 Consider Figure 3.8 (page 89), which was a problematic graph for the other heuristic methods. Although it initially searches down from s because of the heuristic function, eventually the cost of the path becomes so large that it picks the node on an actual optimal path.

The property that A^* always finds an optimal path, if one exists, and that the first path found to a goal is optimal is called the **admissibility** of A^*. Admissibility means that, even when the search space is infinite, if solutions exist, a solution will be found and the first path found will be an optimal solution – a lowest-cost path from a start node to a goal node.

Proposition 3.1. *(A^* admissibility): If there is a solution, A^* always finds a solution, and the first solution found is an optimal solution, if*

- *the branching factor is finite (each node has only a finite number of neighbors),*
- *arc costs are greater than some $\epsilon > 0$, and*
- *$h(n)$ is a lower bound on the actual minimum cost of the lowest-cost path from n to a goal node.*

Proof. **Part A:** A solution will be found. If the arc costs are all greater than some $\epsilon > 0$, eventually, for all paths p in the frontier, $cost(p)$ will exceed any finite number and, thus, will exceed a solution cost if one exists (at depth in the search tree no greater than m/ϵ, where m is the solution cost). Because the branching factor is finite, only a finite number of nodes must be expanded before the search tree could get to this size, but the A^* search would have found a solution by then.
Part B: The first path to a goal selected is an optimal path. The f-value for any node on an optimal solution path is less than or equal to the f-value of an optimal solution. This is because h is an underestimate of the actual cost from a node to a goal. Thus, the f-value of a node on an optimal solution path is less than the f-value for any non-optimal solution. Thus, a non-optimal solution can never be chosen while a node exists on the frontier that leads to an optimal solution (because an element with minimum f-value is chosen at each step). So, before it can select a non-optimal solution, it will have to pick all of the nodes on an optimal path, including each of the optimal solutions. □

It should be noted that the admissibility of A^* does not ensure that every intermediate node selected from the frontier is on an optimal path from the start node to a goal node. Admissibility relieves the algorithm from worrying about cycles and ensures that the first solution found will be optimal. It does not ensure that the algorithm will not change its mind about which partial path is the best while it is searching.

To see how the heuristic function improves the efficiency of A^*, suppose c is the cost of a shortest path from a start node to a goal node. A^*, with an admissible heuristic, expands every path from a start node in the set

$$\{p : cost(p) + h(p) < c\}$$

Strategy	Selection from Frontier	Halts?	Space
Depth-first	Last node added	No	Linear
Breadth-first	First node added	Yes	Exponential
Best-first	Globally minimal $h(p)$	No	Exponential
Lowest-cost-first	Minimal $cost(p)$	Yes	Exponential
A^*	Minimal $cost(p) + h(p)$	Yes	Exponential

"Halts?" means "Is the method guaranteed to halt if there is a path to a goal on a (possibly infinite) graph with a finite number of neighbors for each node and where the arc costs have a positive lower bound?" Those search strategies where the answer is "Yes" have worst-case time complexity which increases exponentially with the size of the path length. Those algorithms that are not guaranteed to halt have infinite worst-case time complexity.

Space refers to the space complexity, which is either "Linear" in the path length or "Exponential" in the path length.

Figure 3.9: Summary of search strategies

and some of the paths in the set

$$\{p : cost(p) + h(p) = c\}.$$

Improving h affects the efficiency of A^* if it reduces the size of the first of these sets.

3.6.2 Summary of Search Strategies

The table in Figure 3.9 gives a summary of the searching strategies presented so far.

The depth-first methods are linear in space with respect to the path lengths explored but are not guaranteed to find a solution if one exists. Breadth-first, lowest-cost-first, and A^* may be exponential in both space and time, but they are guaranteed to find a solution if one exists, even if the graph is infinite (as long as there are finite branching factors and positive non-trivial arc costs).

Lowest-cost-first and A^* searches are guaranteed to find the least-cost solution as the first solution found.

3.7 More Sophisticated Search

A number of refinements can be made to the preceding strategies. First, we present two methods that are applicable when there are cycles in the graph; one checks explicitly for cycles, whereas the other method checks for multiple paths to a node. Next, we present iterative deepening and depth-first branch-and-bound searches, which are general methods that are guaranteed to find

a solution (even an optimal solution), like breadth-first search or A^* search, but using the space advantages of depth-first search. We present problem-reduction methods to break down a search problem into a number of smaller search problems, each of which may be much easier to solve. Finally, we show how dynamic programming can be used for path finding and for constructing heuristic functions.

3.7.1 Cycle Checking

It is possible for a graph representing a search space to include cycles. For example, in the robot delivery domain of Figure 3.6 (page 84), the robot can go back and forth between nodes $o103$ and $o109$. Some of the aforementioned search methods can get trapped in cycles, continuously repeating the cycle and never finding an answer even in finite graphs. The other methods can loop though cycles, but eventually they still find a solution.

The simplest method of pruning the search tree, while guaranteeing that a solution will be found in a finite graph, is to ensure that the algorithm does not consider neighbors that are already on the path from the start. A **cycle check** or **loop check** checks for paths where the last node already appears on the path from the start node to that node. With a cycle check, only the paths $\langle s_0, \ldots, s_k, s \rangle$, where $s \notin \{s_0, \ldots, s_k\}$, are added to the frontier at line 16 of Figure 3.4 (page 79). Alternatively, the check can be made after a node is selected; paths with a cycle can be thrown away.

The computational complexity of a cycle check depends on the search method being used. For depth-first methods, where the graph is explicitly stored, the overhead can be as low as a constant factor; a bit can be added to each node in the graph that is assigned a value of 1 when the node is expanded, and assigned a value of 0 on backtracking. A search algorithm can avoid cycles by never expanding a node with its bit set to 1. This approach works because depth-first search maintains a single current path. The elements on the frontier are alternative branches from this path. Even if the graph is generated dynamically, as long as an efficient indexing structure is used for the nodes on the current path, a cycle check can be done efficiently.

For the search strategies that maintain multiple paths – namely, all of those with exponential space in Figure 3.9 – a cycle check takes time linear in the length of the path being searched. These algorithms cannot do better than searching up the partial path being considered, checking to ensure they do not add a node that already appears in the path.

3.7.2 Multiple-Path Pruning

There is often more than one path to a node. If only one path is required, a search algorithm can prune from the frontier any path that leads to a node to which it has already found a path.

Multiple-path pruning can be implemented by keeping a **closed list** of nodes that have been expanded. When a path is selected at line 13 of Figure 3.4 (page 79), if its last node is in the closed list, the path can be discarded. Otherwise, its last node is added to the closed list, and the algorithm proceeds as before.

This approach does not necessarily guarantee that the shortest path is not discarded. Something more sophisticated may have to be done to guarantee that an optimal solution is found. To ensure that the search algorithm can still find a lowest-cost path to a goal, one of the following can be done:

- Make sure that the first path found to any node is a lowest-cost path to that node, then prune all subsequent paths found to that node, as discussed earlier.
- If the search algorithm finds a lower-cost path to a node than one already found, it can remove all paths that used the higher-cost path to the node (because these cannot be on an optimal solution). That is, if there is a path p on the frontier $\langle s, \ldots, n, \ldots, m\rangle$, and a path p' to n is found that is shorter than the portion of the path from s to n in p, then p can be removed from the frontier.
- Whenever the search finds a lower-cost path to a node than a path to that already found, it can incorporate a new initial section on the paths that have extended the initial path. Thus, if there is a path $p = \langle s, \ldots, n, \ldots, m\rangle$ on the frontier, and a path p' to n is found that is shorter than the portion of p from s to n, then p' can replace the initial part of p to n.

The first of these alternatives allows the use of the closed list without losing the ability to find an optimal path. The others require more sophisticated algorithms.

In lowest-cost-first search, the first path found to a node (i.e., when the node is selected from the frontier) is the least-cost path to the node. Pruning subsequent paths to that node cannot remove a lower-cost path to that node, and thus pruning subsequent paths to each node still enables an optimal solution to be found.

As described earlier, A^* (page 89) does not guarantee that when a path to a node is selected for the first time it is the lowest cost path to that node. Note that the admissibility theorem (page 91) guarantees this for every path to a *goal* node but not for every path. To see when pruning subsequent paths to a node can remove the optimal solution, suppose the algorithm has selected a path p to node n for expansion, but there exists a lower-cost path to node n, which it has not found yet. Then there must be a path p' on the frontier that is the initial part of the lower-cost path. Suppose path p' ends at node n'. It must be that $f(p) \leq f(p')$, because p was selected before p'. This means that

$$cost(p) + h(n) \leq cost(p') + h(n').$$

If the path to n via p' has a lower cost than the path p,

$$cost(p') + d(n', n) < cost(p),$$

where $d(n', n)$ is the actual cost of the shortest path from node n' to n. From these two equations, we can derive

$$d(n', n) < cost(p) - cost(p') \leq h(p') - h(p) = h(n') - h(n).$$

Thus, we can ensure that the first path found to any node is the lowest-cost path if $|h(n') - h(n)| \leq d(n', n)$ for any two nodes n and n'. The **monotone restriction** on h is that $|h(n') - h(n)| \leq d(n', n)$ for any two nodes n and n'. That is, the difference in the heuristic values for two nodes must be less than or equal to the actual cost of the lowest-cost path between the nodes. It is applicable to, for example, the heuristic function of Euclidean distance (the straight-line distance in an n-dimensional Euclidean space) between two points when the cost function is distance. It is also typically applicable when the heuristic function is a solution to a simplified problem that has shorter solutions.

With the monotone restriction, the f-values on the frontier are monotonically non-decreasing. That is, when the frontier is expanded, the f-values do not get smaller. Thus, with the monotone restriction, subsequent paths to any node can be pruned in A^* search.

Multiple-path pruning subsumes a cycle check, because a cycle is another path to a node and is therefore pruned. Multiple-path pruning can be done in constant time, if the graph is explicitly stored, by setting a bit on each node to which a path has been found. It can be done in logarithmic time (in the number of nodes expanded, as long as it is indexed appropriately), if the graph is dynamically generated, by storing the closed list of all of the nodes that have been expanded. Multiple-path pruning is preferred over cycle checking for breadth-first methods where virtually all of the nodes considered have to be stored anyway. For depth-first search strategies, however, the algorithm does not otherwise have to store all of the nodes already considered. Storing them makes the method exponential in space. Therefore, cycle checking is preferred over multiple-path checking for depth-first methods.

3.7.3 Iterative Deepening

So far, none of the methods discussed have been ideal; the only ones that guarantee that a path will be found require exponential space (see Figure 3.9 (page 92)). One way to combine the space efficiency of depth-first search with the optimality of breadth-first methods is to use **iterative deepening**. The idea is to recompute the elements of the frontier rather than storing them. Each recomputation can be a depth-first search, which thus uses less space.

Consider making a breadth-first search into an iterative deepening search. This is carried out by having a depth-first searcher, which searches only to a limited depth. It can first do a depth-first search to depth 1 by building paths of length 1 in a depth-first manner. Then it can build paths to depth 2, then depth 3, and so on. It can throw away all of the previous computation each time and start again. Eventually it will find a solution if one exists, and, as it is

enumerating paths in order, the path with the fewest arcs will always be found first.

When implementing an iterative deepening search, you have to distinguish between

- failure because the depth bound was reached and
- failure that does not involve reaching the depth bound.

In the first case, the search must be retried with a larger depth bound. In the second case, it is a waste of time to try again with a larger depth bound, because no path exists no matter what the depth. We say that failure due to reaching the depth bound is **failing unnaturally**, and failure without reaching the depth bound is **failing naturally**.

An implementation of iterative-deepening search, *IdSearch*, is presented in Figure 3.10. The local procedure *dbsearch* implements a depth-bounded depth-first search (using recursion to keep the stack) that places a limit on the length of the paths for which it is searching. It uses a depth-first search to find all paths of length $k + b$, where k is the path length of the given path from the start and b is a non-negative integer. The iterative-deepening searcher calls this for increasing depth bounds. This program finds the paths to goal nodes in the same order as does the breadth-first search. As in the generic graph searching algorithm, to find more solutions after the *return* on line 22, the search can continue from line 23.

The iterative-deepening search fails whenever the breadth-first search would fail. When asked for multiple answers, it only returns each successful path once, even though it may be rediscovered in subsequent iterations. Halting is achieved by keeping track of when increasing the bound could help find an answer:

- The depth bound is increased if the depth bound search was truncated by reaching the depth bound. In this case, the search failed *unnaturally*. The search failed *naturally* if the search did not prune any paths due to the depth bound. In this case, the program can stop and report no (more) paths.
- The search only reports a solution path if that path would not have been reported in the previous iteration. Thus, it only reports paths whose length is the depth bound.

The obvious problem with iterative deepening is the wasted computation that occurs at each step. This, however, may not be as bad as one might think, particularly if the branching factor is high. Consider the running time of the algorithm. Assume a constant branching factor of $b > 1$. Consider the search where the bound is k. At depth k, there are b^k nodes; each of these has been generated once. The nodes at depth $k - 1$ have been generated twice, those at depth $k - 2$ have been generated three times, and so on, and the nodes at depth 1 have been generated k times. Thus, the total number of nodes

```
1: procedure IdSearch(G, s, goal)
2:     Inputs
3:         G: graph with nodes N and arcs A
4:         s: set of start nodes
5:         goal: Boolean function on states
6:     Output
7:         path from s to a node for which goal is true
8:         or ⊥ if there is no such path
9:     Local
10:        natural_failure: Boolean
11:        bound: integer
12:        procedure dbsearch(⟨n_0, ..., n_k⟩, b)
13:            Inputs
14:                ⟨n_0, ..., n_k⟩: path
15:                b: integer, b ≥ 0
16:            Output
17:                path to goal of length k + b
18:            if b > 0 then
19:                for each arc ⟨n_k, n⟩ ∈ A do
20:                    dbsearch(⟨n_0, ..., n_k, n⟩, b − 1)
21:            else if goal(n_k) then
22:                return ⟨n_0, ..., n_k⟩
23:            else if n_k has any neighbors then
24:                natural_failure := false
25:    bound := 0
26:    repeat
27:        natural_failure := true
28:        dbsearch({⟨s⟩ : s ∈ S}, bound)
29:        bound := bound + 1
30:    until natural_failure
31:    return ⊥
```

Figure 3.10: Iterative deepening search

generated is

$$
\begin{aligned}
& b^k + 2b^{k-1} + 3b^{k-2} + \cdots + kb \\
&= b^k(1 + 2b^{-1} + 3b^{-2} + \cdots + kb^{1-k}) \\
&\leq b^k \left(\sum_{i=1}^{\infty} ib^{(1-i)} \right) \\
&= b^k \left(\frac{b}{b-1} \right)^2 .
\end{aligned}
$$

There is a constant overhead $(b/(b-1))^2$ times the cost of generating the nodes at depth n. When $b = 2$ there is an overhead factor of 4, and when $b = 3$ there is an overhead of 2.25 over generating the frontier. This algorithm is $O(b^k)$ and there cannot be an asymptotically better uninformed search strategy. Note that, if the branching factor is close to 1, this analysis does not work (because then the denominator would be close to zero); see Exercise 3.9 (page 109).

Iterative deepening can also be applied to an A^* search. **Iterative deepening A^*** (IDA*) performs repeated depth-bounded depth-first searches. Instead of the bound being on the number of arcs in the path, it is a bound on the value of $f(n)$. The threshold starts at the value of $f(s)$, where s is the starting node with minimal h-value. IDA* then carries out a depth-first depth-bounded search but never expands a node with a higher f-value than the current bound. If the depth-bounded search fails *unnaturally*, the next bound is the minimum of the f-values that exceeded the previous bound. IDA* thus checks the same nodes as A^* but recomputes them with a depth-first search instead of storing them.

3.7.4 Branch and Bound

Depth-first **branch-and-bound** search is a way to combine the space saving of depth-first search with heuristic information. It is particularly applicable when many paths to a goal exist and we want an optimal path. As in A^* search, we assume that $h(n)$ is less than or equal to the cost of a lowest-cost path from n to a goal node.

The idea of a branch-and-bound search is to maintain the lowest-cost path to a goal found so far, and its cost. Suppose this cost is *bound*. If the search encounters a path p such that $cost(p) + h(p) \geq bound$, path p can be pruned. If a non-pruned path to a goal is found, it must be better than the previous best path. This new solution is remembered and *bound* is set to the cost of this new solution. It then keeps searching for a better solution.

Branch-and-bound search generates a sequence of ever-improving solutions. Once it has found a solution, it can keep improving it.

Branch-and-bound search is typically used with depth-first search, where the space saving of the depth-first search can be achieved. It can be implemented similarly to depth-bounded search, but where the bound is in terms of path cost and reduces as shorter paths are found. The algorithm remembers the lowest-cost path found and returns this path when the search finishes.

The algorithm is shown in Figure 3.11. The internal procedure *cbsearch*, for cost-bounded search, uses the global variables to provide information to the main procedure.

Initially, *bound* can be set to infinity, but it is often useful to set it to an overestimate, $bound_0$, of the path cost of an optimal solution. This algorithm

```
1: procedure DFBranchAndBound(G, s, goal, h, bound_0)
2:     Inputs
3:         G: graph with nodes N and arcs A
4:         s: start node
5:         goal: Boolean function on nodes
6:         h: heuristic function on nodes
7:         bound_0: initial depth bound (can be ∞ if not specified)
8:     Output
9:         a least-cost path from s to a goal node if there is a solution with cost
   less than bound_0
10:        or ⊥ if there is no solution with cost less than bound_0
11:    Local
12:        best_path: path or ⊥
13:        bound: non-negative real
14:        procedure cbsearch(⟨n_0, ..., n_k⟩)
15:            if cost(⟨n_0, ..., n_k⟩) + h(n_k) < bound then
16:                if goal(n_k) then
17:                    best_path := ⟨n_0, ..., n_k⟩
18:                    bound := cost(⟨n_0, ..., n_k⟩)
19:                else
20:                    for each arc ⟨n_k, n⟩ ∈ A do
21:                        cbsearch(⟨n_0, ..., n_k, n⟩)
22:    best_path := ⊥
23:    bound := bound_0
24:    cbsearch(⟨s⟩)
25:    return best_path
```

Figure 3.11: Depth-first branch-and-bound search

will return an optimal solution – a least-cost path from the start node to a goal node – if there is a solution with cost less than the initial bound $bound_0$.

If the initial bound is slightly above the cost of a lowest-cost path, this algorithm can find an optimal path expanding no more arcs than A^* search. This happens when the initial bound is such that the algorithm prunes any path that has a higher cost than a lowest-cost path; once it has found a path to the goal, it only explores paths whose the f-value is lower than the path found. These are exactly the paths that A^* explores when it finds one solution.

If it returns $\bot$ when $bound_0 = \infty$, there are no solutions. If it returns $\bot$ when $bound_0$ is some finite value, it means no solution exists with cost less than $bound_0$. This algorithm can be combined with iterative deepening to increase the bound until either a solution is found or it can be shown there is no solution. See Exercise 3.13 (page 109).

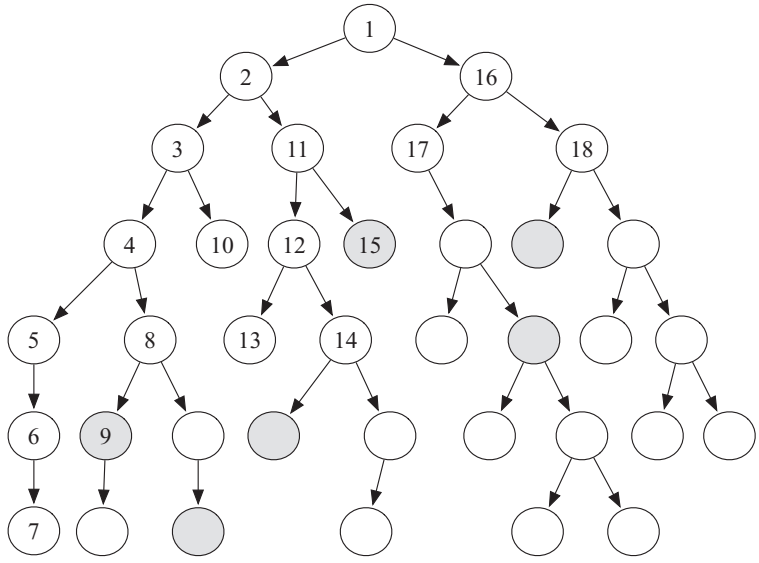

Figure 3.12: The nodes expanded in depth-first branch-and-bound search

Example 3.17 Consider the tree-shaped graph in Figure 3.12. The goal nodes are shaded. Suppose that each arc has length 1, and there is no heuristic information (i.e., $h(n) = 0$ for each node n). In the algorithm, suppose $depth_0 = \infty$ and the depth-first search always chooses the leftmost child first. This figure shows the order in which the nodes are checked to determine if they are a goal node. The nodes that are not numbered are not checked for being a goal node.

The subtree under the node numbered "5" does not have a goal and is explored fully (or up to depth $depth_0$ if it had a finite value). The ninth node checked is a goal node. It has a path cost of 5, and so the bound is set to 5. From then on, only paths with a length of less than 5 are checked for being a solution. The fifteenth node checked is also a goal. It has a path cost of 3, and so the bound is reduced to 3. There are no other goal nodes found, and so the path to the node labeled 15 is returned. It is an optimal path. There is another optimal path that is pruned; the algorithm never checks the children of the node labeled with 18.

If there was heuristic information, it could be used to prune parts of the search space, as in A^* search.

3.7.5 Direction of Search

The size of the search space of the generic search algorithm on finite uniform graphs is b^k, where b is the branching factor and k is the path length. Anything that can be done to reduce k or b can potentially give great savings.

The abstract definition of the graph-searching method of problem solving is symmetric in the sense that the algorithm can either begin with a start node and

search forward for a goal node or begin with a goal node and search backward for a start node in the inverse graph. Note that in many applications the goal is determined implicitly by a Boolean function that returns *true* when a goal is found, and not explicitly as a set of nodes, so backward search may not be possible.

For those cases where the goal nodes are explicit, it may be more efficient to search in one direction than in the other. The size of the search space is exponential in the branching factor. It is typically the case that forward and backward searches have different branching factors. A general principle is to search forward or backward, depending on which has the smaller branching factor.

The following sections consider some ways in which search efficiency can be improved beyond this for many search spaces.

Bidirectional Search

The idea of a bidirectional search is to reduce the search time by searching forward from the start and backward from the goal simultaneously. When the two search frontiers intersect, the algorithm can reconstruct a single path that extends from the start state through the frontier intersection to the goal.

A new problem arises during a bidirectional search, namely ensuring that the two search frontiers actually meet. For example, a depth-first search in both directions is not likely to work well because its small search frontiers are likely to pass each other by. Breadth-first search in both directions would be guaranteed to meet.

A combination of depth-first search in one direction and breadth-first search in the other would guarantee the required intersection of the search frontiers, but the choice of which to apply in which direction may be difficult. The decision depends on the cost of saving the breadth-first frontier and searching it to check when the depth-first method will intersect one of its elements.

There are situations where a bidirectional search can result in substantial savings. For example, if the forward and backward branching factors of the search space are both b, and the goal is at depth k, then breadth-first search will take time proportional to b^k, whereas a symmetric bidirectional search will take time proportional to $2b^{k/2}$. This is an exponential savings in time, even though the time complexity is still exponential. Note that this complexity analysis assumes that finding the intersection of frontiers is free, which may not be a valid assumption for many applications (see Exercise 3.10 (page 109)).

Island-Driven Search

One of the ways that search may be made more efficient is to identify a limited number of places where the forward search and backward search can meet. For example, in searching for a path from two rooms on different floors, it may be appropriate to constrain the search to first go to the elevator on one level,

then to the elevator on the goal level. Intuitively, these designated positions are **islands** in the search graph, which are constrained to be on a solution path from the start node to a goal node.

When islands are specified, an agent can decompose the search problem into several search problems, for example, one from the initial room to the elevator, one from the elevator on one level to the elevator on the other level, and one from the elevator to the destination room. This reduces the search space by having three simpler problems to solve. Having smaller problems helps to reduce the combinatorial explosion of large searches and is an example of how extra knowledge about a problem can be used to improve efficiency of search.

To find a path between s and g using islands:

1. Identify a set of islands $i_0, ..., i_k$;
2. Find paths from s to i_0, from i_{j-1} to i_j for each j from 1 to k, and from i_k to g.

Each of these searching problems should be correspondingly simpler than the general problem and therefore easier to solve.

The identification of islands is extra knowledge which may be beyond that which is in the graph. The use of inappropriate islands may make the problem more difficult (or even impossible to solve). It may also be possible to identify an alternate decomposition of the problem by choosing a different set of islands and search through the space of possible islands. Whether this works in practice depends on the details of the problem.

Searching in a Hierarchy of Abstractions

The notion of islands can be used to define problem-solving strategies that work at multiple levels of detail or multiple levels of abstraction.

The idea of searching in a hierarchy of abstractions first involves abstracting the problem, leaving out as many details as possible. A partial solution to a problem may be found – one that requires further details to be worked out. For example, the problem of getting from one room to another requires the use of many instances of turning, but an agent would like to reason about the problem at a level of abstraction where the details of the actual steering are omitted. It is expected that an appropriate abstraction solves the problem in broad strokes, leaving only minor problems to be solved. The route planning problem for the delivery robot is too difficult to solve by searching without leaving out details until it must consider them.

One way this can be implemented is to generalize island-driven search to search over possible islands. Once a solution is found at the island level, subproblems can be solved recursively in the same manner. Information that is found at the lower level can inform higher levels that some potential solution does not work as well as expected. The higher level can then use that information to replan. This process typically does not result in a guaranteed optimal solution because it only considers some of the high-level decompositions.

Searching in a hierarchy of abstractions depends very heavily on *how* one decomposes and *abstracts* the problem to be solved. Once the problems are abstracted and decomposed, any of the search methods can be used to solve them. It is not easy, however, to recognize useful abstractions and problem decompositions.

3.7.6 Dynamic Programming

Dynamic programming is a general method for optimization that involves storing partial solutions to problems, so that a solution that has already been found can be retrieved rather than being recomputed. Dynamic programming algorithms are used throughout AI.

Dynamic programming can be used for finding paths in graphs. Intuitively, **dynamic programming** for graph searching can be seen as constructing the *perfect* heuristic function so that A^*, even if it keeps only one element of the frontier, is guaranteed to find a solution. This cost-to-goal function represents the exact cost of a minimal-cost path from each node to the goal.

A **policy** is a specification of which arc to take from each node. The cost-to-goal function can be computed offline and can be used to build an optimal policy. Online, an agent can use this policy to determine what to do at each point.

Let $cost\_to\_goal(n)$ be the actual cost of a lowest-cost path from node n to a goal; $cost\_to\_goal(n)$ can be defined as

$$cost\_to\_goal(n) = \begin{cases} 0 & \text{if } is\_goal(n), \\ \min_{\langle n,m\rangle \in A}(cost(\langle n,m\rangle) + cost\_to\_goal(m)) & \text{otherwise.} \end{cases}$$

The general idea is to start at the goal and build a table of the $cost\_to\_goal(n)$ value for each node. This can be done by carrying out a lowest-cost-first search, with multiple-path pruning, from the goal nodes in the *inverse graph*, which is the graph with all arcs reversed. Rather than having a goal to search for, the dynamic programming algorithm records the *cost_to_goal* values for each node found. It uses the inverse graph to compute the costs from each node to the goal and not the costs from the goal to each node. In essence, dynamic programming works backward from the goal by trying to build the lowest-cost paths to the goal from each node in the graph.

For a particular goal, once the *cost_to_goal* value for each node has been recorded, an agent can use the *cost_to_goal* value to determine the next arc on an optimal path. From node n it should go to a neighbor m that minimizes $cost(\langle n,m\rangle) + cost\_to\_goal(m)$. Following this policy will take the agent from any node to a goal along a lowest-cost path. Given *cost_to_goal*, determining which arc is optimal takes constant time with respect to the size of the graph, assuming a bounded number of neighbors for each node. Dynamic programming takes time and space linear in the size of the graph to build the *cost_to_goal* table.

Dynamic programming is useful when

- the goal nodes are explicit (the previous methods only assumed a function that recognizes goal nodes);
- a lowest-cost path is needed;
- the graph is finite and small enough to be able to store the *cost_to_goal* value for each node;
- the goal does not change very often; and
- the policy is used a number of times for each goal, so that the cost of generating the *cost_to_goal* values can be amortized over many instances of the problem.

The main problems with dynamic programming are that

- it only works when the graph is finite and the table can be made small enough to fit into memory,
- an agent must recompute a policy for each different goal, and
- the time and space required is linear in the size of the graph, where the graph size for finite graphs is typically exponential in the path length.

Example 3.18 For the graph given in Figure 3.2 (page 76), the cost from $r123$ to the goal is 0; thus,

$$cost\_to\_goal(r123) = 0.$$

Continuing with a lowest-cost-first search from $r123$:

$$cost\_to\_goal(o123) = 4$$
$$cost\_to\_goal(o119) = 13$$
$$cost\_to\_goal(o109) = 29$$
$$cost\_to\_goal(b4) = 36$$
$$cost\_to\_goal(b2) = 39$$
$$cost\_to\_goal(o103) = 41$$
$$cost\_to\_goal(b3) = 43$$
$$cost\_to\_goal(b1) = 45$$

At this stage the backward search halts. Two things can be noticed here. First, if a node does not have a *cost_to_goal* value, then no path to the goal exists from that node. Second, an agent can quickly determine the next arc on a lowest-cost path to the goal for any node. For example, if the agent is at $o103$, to determine a lowest-cost path to $r123$ it compares $4 + 43$ (the cost of going via $b3$) with $12 + 29$ (the cost of going straight to $o109$) and can quickly determine to go to $o109$.

When building the *cost_to_goal* function, the searcher has implicitly determined which neighbor leads to the goal. Instead of determining at run time which neighbor is on an optimal path, it can store this information.

Optimality of the A^* algorithm

A search algorithm is **optimal** if no other search algorithm uses less time or space or expands fewer nodes, both with a guarantee of solution quality. The optimal search algorithm would be one that picks the correct node at each choice. However, this specification is not effective because we cannot directly implement it. Whether such an algorithm is possible is an open question (as to whether $P = NP$). There does, however, seem to be a statement that can be proved.

Optimality of A^*: Among search algorithms that only use arc costs and a heuristic estimate of the cost from a node to a goal, no algorithm expands fewer nodes than A^* and guarantees to find a lowest-cost path.

Proof sketch: Given only the information about the arc costs and the heuristic information, unless the algorithm has expanded each path p, where $f(p)$ is less than the cost of an optimal path, it does not know whether p leads to a lower-cost path. More formally, suppose an algorithm A' found a path for a problem P where some path p was not expanded such that $f(p)$ was less than the solution found. Suppose there was another problem P', which was the same as P, except that there really was a path via p with cost $f(p)$. The algorithm A' cannot tell P' from P, because it did not expand the path p, so it would report the same solution for P' as for P, but the solution found for P would not be optimal for P' because the solution found has a higher cost than the path via p. Therefore, an algorithm is not guaranteed to find a lowest-cost path unless it explores all paths with f-values less than the value of an optimal path; that is, it must explore all the paths that A^* explores.

Counterexample: Although this proof seems reasonable, there are algorithms that explore fewer nodes. Consider an algorithm that does a forward A^*-like search and a backward dynamic programming search, where the steps are interleaved in some way (e.g., by alternating between the forward steps and the backward steps). The backward search builds a table of $cost\_to\_goal(n)$ values of the actual discovered cost from n to a goal, and it maintains a bound b, where it has explored all paths of cost less than b to a goal. The forward search uses a priority queue on $cost(p) + c(n)$, where n is the node at the end of the path p, and $c(n)$ is $cost\_to\_goal(n)$ if it has been computed; otherwise, $c(n)$ is $max(h(n), b)$. The intuition is that, if a path exists from the end of path p to a goal node, either it uses a path that has been discovered by the backward search or it uses a path that costs at least b. This algorithm is guaranteed to find a lowest-cost path and often expands fewer nodes than A^* (see Exercise 3.11 (page 109)).

Conclusion: Having a counterexample would seem to mean that the optimality of A^* is false. However, the proof does seem to have some appeal and perhaps it should not be dismissed outright. A^* is not optimal out of the class of all algorithms, but the proof seems right for the class of algorithms that only do forward search. See Exercise 3.12 (page 109).

Dynamic programming can be used to construct heuristics for A^* and branch-and-bound searches. One way to build a heuristic function is to simplify the problem (e.g., by leaving out some details) until the simplified problem has a small enough state space. Dynamic programming can be used to find the optimal path length to a goal in the simplified problem. This information can then be used as a heuristic for the original problem.

3.8 Review

The following are the main points you should have learned from this chapter:

- Many problems can be abstracted as the problem of finding paths in graphs.
- Breadth-first and depth-first searches can find paths in graphs without any extra knowledge beyond the graph.
- A^* search can use a heuristic function that estimates the cost from a node to a goal. If this estimate underestimates the actual cost, A^* is guaranteed to find a least-cost path first.
- Iterative deepening and depth-first branch-and-bound searches can be used to find least-cost paths with less memory than methods such as A^*, which store multiple paths.
- When graphs are small, dynamic programming can be used to record the actual cost of a least-cost path from each node to the goal, which can be used to find the next arc in an optimal path.

3.9 References and Further Reading

There is a lot of information on search techniques in the literature of operations research, computer science, and AI. Search was seen early on as one of the foundations of AI. The AI literature emphasizes heuristic search.

Basic search algorithms are discussed in Nilsson [1971]. For a detailed analysis of heuristic search see Pearl [1984]. The A^* algorithm was developed by Hart, Nilsson, and Raphael [1968].

Depth-first iterative deepening is described in Korf [1985].

Branch-and-bound search was developed in the operations research community and is described in Lawler and Wood [1966].

Dynamic programming is a general algorithm that will be used as a dual to search algorithms in other parts of the book. The specific algorithm presented here was invented by Dijkstra [1959]. See Cormen, Leiserson, Rivest, and Stein [2001] for more details on the general class of dynamic programming algorithms.

The idea of using dynamic programming as a source of heuristics for A^* search was proposed by Culberson and Schaeffer [1998] and further developed by Felner, Korf, and Hanan [2004].

Minsky [1961] discussed islands and problem reduction.

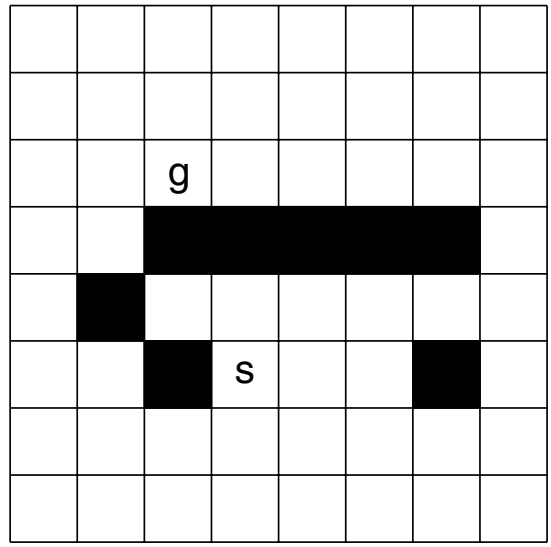

Figure 3.13: A grid-searching problem

3.10 Exercises

Exercise 3.1 Comment on the following quote: "One of the main goals of AI should be to build general heuristics that can be used for any graph-searching problem."

Exercise 3.2 Which of the path-finding search procedures are fair in the sense that any element on the frontier will eventually be chosen? Consider this for question finite graphs without loops, finite graphs with loops, and infinite graphs (with finite branching factors).

Exercise 3.3 Consider the problem of finding a path in the grid shown in Figure 3.13 from the position s to the position g. A piece can move on the grid horizontally and vertically, one square at a time. No step may be made into a forbidden shaded area.

(a) On the grid shown in Figure 3.13, number the nodes expanded (in order) for a depth-first search from s to g, given that the order of the operators is up, left, right, then down. Assume there is a cycle check.

(b) For the same grid, number the nodes expanded, in order, for a best-first search from s to g. Manhattan distance should be used as the evaluation function. The Manhattan distance between two points is the distance in the x-direction plus the distance in the y-direction. It corresponds to the distance traveled along city streets arranged in a grid. Assume multiple-path pruning. What is the first path found?

(c) On the same grid, number the nodes expanded, in order, for a heuristic depth-first search from s to g, given Manhattan distance as the evaluation function. Assume a cycle check. What is the path found?

(d) Number the nodes in order for an A^* search, with multiple-path pruning, for the same graph. What is the path found?

(e) Show how to solve the same problem using dynamic programming. Give the *dist* value for each node, and show which path is found.

(f) Based on this experience, discuss which algorithms are best suited for this problem.

(g) Suppose that the graph extended infinitely in all directions. That is, there is no boundary, but s, g, and the blocks are in the same positions relative to each other. Which methods would no longer find a path? Which would be the best method, and why?

Exercise 3.4 This question investigates using graph searching to design video presentations. Suppose there exists a database of video segments, together with their length in seconds and the topics covered, set up as follows:

Segment	Length	Topics covered
seg0	10	[welcome]
seg1	30	[skiing, views]
seg2	50	[welcome, artificial_intelligence, robots]
seg3	40	[graphics, dragons]
seg4	50	[skiing, robots]

Suppose we represent a node as a pair:

$$\langle To\_Cover, Segs \rangle ,$$

where *Segs* is a list of segments that must be in the presentation, and *To_Cover* is a list of topics that also must be covered. Assume that none of the segments in *Segs* cover any of the topics in *To_Cover*.

The neighbors of a node are obtained by first selecting a topic from *To_Cover*. There is a neighbor for each segment that covers the selected topic. [Part of this exercise is to think about the exact structure of these neighbors.]

For example, given the aforementioned database of segments, the neighbors of the node $\langle [welcome, robots], [] \rangle$, assuming that *welcome* was selected, are $\langle [], [seg2] \rangle$ and $\langle [robots], [seg0] \rangle$.

Thus, each arc adds exactly one segment but can cover one or more topics. Suppose that the cost of the arc is equal to the time of the segment added.

The goal is to design a presentation that covers all of the topics in *MustCover*. The starting node is $\langle MustCover, [] \rangle$, and the goal nodes are of the form $\langle [], Presentation \rangle$. The cost of the path from a start node to a goal node is the time of the presentation. Thus, an optimal presentation is a shortest presentation that covers all of the topics in *MustCover*.

(a) Suppose that the goal is to cover the topics $[welcome, skiing, robots]$. Suppose the algorithm always select the leftmost topic to find the neighbors for each node. Draw the search space expanded for a lowest-cost-first search until the first solution is found. This should show all nodes expanded, which node is a goal node, and the frontier when the goal was found.

(b) Give a non-trivial heuristic function h that is an underestimate of the real cost. [Note that $h(n) = 0$ for all n is the trivial heuristic function.] Does it satisfy the monotone restriction for a heuristic function?

Exercise 3.5 Draw two different graphs, indicating start and goal nodes, for which forward search is better in one and backward search is better in the other.

Exercise 3.6 Implement iterative-deepening A^*. This should be based on the iterative deepening searcher of Figure 3.10 (page 97).

Exercise 3.7 Suppose that, rather than finding an optimal path from the start to a goal, we wanted a path with a cost not more than, say, 10% greater than the least-cost path. Suggest an alternative to an iterative-deepening A^* search that would guarantee this outcome. Why might this be advantageous to iterative-deepening A^* search?

Exercise 3.8 How can depth-first branch-and-bound be modified to find a path with a cost that is not more than, say 10% greater than the least-cost path. How does this algorithm compare to the variant of A^* from the previous question?

Exercise 3.9 The overhead for iterative deepening with $b - 1$ on the denominator (page 97) is not a good approximation when $b \approx 1$. Give a better estimate of the complexity of iterative deepening when $b = 1$. What is the complexity of the other methods given in this chapter? Suggest a way that iterative deepening can have a lower overhead when the branching factor is close to 1.

Exercise 3.10 Bidirectional search must be able to determine when the frontiers intersect. For each of the following pairs of searches specify how to determine when the frontiers intersect:

(a) Breadth-first search and depth-bounded depth-first search.

(b) Iterative deepening search and depth-bounded depth-first search.

(c) A^* and depth-bounded depth-bounded search.

(d) A^* and A^*.

Exercise 3.11 Consider the algorithm sketched in the counterexample of the box on page 105:

(a) When can the algorithm stop? (Hint: it does not have to wait until the forward search finds a path to a goal).

(b) What data structures should be kept?

(c) Specify the algorithm in full.

(d) Show that it finds the optimal path.

(e) Give an example where it expands (many) fewer nodes than A^*.

Exercise 3.12 Give a statement of the optimality of A^* that specifies the class of algorithms for which A^* is optimal. Give the formal proof.

Exercise 3.13 The depth-first branch and bound of Figure 3.11 (page 99) is like a depth-bounded search in that it only finds a solution if there is a solution with

cost less than *bound*. Show how this can be combined with an iterative deepening search to increase the depth bound if there is no solution for a particular depth bound. This algorithm must return $\perp$ in a finite graph if there is no solution. The algorithm should allow the bound to be incremented by an arbitrary amount and still return an optimal (least-cost) solution when there is a solution.