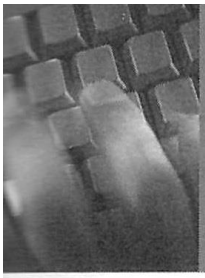

CHAPTER 2

Problem Solving Using Search

In this chapter we explore the first major thrust of artificial intelligence (AI) research, problem solving using search. We discuss how problems can be represented as states and then solved using simple brute-force search techniques or more sophisticated heuristic search methods. A Java application is developed that implements five different search algorithms.

Defining the Problem

The focus of much AI research has been on solving problems. While, in the past, some critics have argued that AI has focused on unrealistically simple or toy problems, this is certainly not the case today. In hindsight, it is not clear that the "toy problem" criticisms were justified at all. Much of the point of the AI research was to understand "how" to solve the problem, not just to get a solution. So seemingly simple problems—puzzles, games, stacking wooden blocks—were the focus of AI programs. And one of the first areas of work, general problem-solving methods, highlighted a major barrier to artificial intelligence software. How do you represent a problem so that the computer can solve it? Even before that, how do you define the problem with enough precision so that you can figure out how to represent it?

While "knowing what business you are in" is one of the elementary business maxims, for artificial intelligence it is "knowing what problem you are trying to solve." For some people, solving the problem successfully is the only goal. Why use a computer unless it can help you solve business problems (and make money)? For others, the challenge is to reproduce human problem-solving techniques or, at least, gain a better understanding of how people solve complex problems. Today, very few people would claim that search-based methods

demonstrate how our brain solves problems, but these methods have proven extremely useful and have a place in any discussion of practical AI techniques.

The first step in any problem-solving exercise is to clearly and succinctly define what it is we are trying to do. Do we want to find the best possible route for a trip or any one that will get us to our destination? Can we wait several hours or days for an answer, or do we need the best answer that can be computed in ten seconds? Someone once said that in AI the most important part of problem solving is "representation, representation, representation." In this chapter, we look at ways of representing our problem so that we can solve it using search techniques. We also examine which search techniques we should use. In the next section, we explore one of the primary problem representations, the state-space approach.

State Space

Suppose that the problem we want to solve deals with playing and winning a game. This game could be a simple one, such as tic-tac-toe, or a more complex one, such as checkers, chess, or backgammon. In any case, the key to approaching this problem with a computer is to develop a mapping from the game-space world of pieces and the geometric pattern on a board, to a data structure that captures the essence of the current game state. For tic-tac-toe, we could use an array with nine elements, or we could define a 3-by-3 matrix with 1s and 0s to denote the Xs and Os of each player.

We start with an *initial state*, which in the case of tic-tac-toe is a 3-by-3 matrix filled with spaces (or empty markers). For any state of our game board, we have a set of *operators* that can be used to modify the current state, thereby creating a new state. In our case, this is a player marking an empty space with either an X or an O. The combination of the initial state and the set of operators make up the *state space* of the problem. The sequence of states produced by the valid application of operators from the initial state is called the *path* in the state space. Now that we have the means to go from our initial state to additional valid game states, we need to be able to detect when we have reached our *goal* state. In tic-tac-toe, a goal state is when any row, column, or diagonal consists of all Xs or all Os. In this simple example, we can check the tic-tac-toe board to see if either player has won by explicitly testing for our goal condition. In more complicated problems, defining the *goal test* may be a substantial problem in itself.

In many search problems, we are not only interested in reaching a goal state, we would like to reach it with the lowest possible cost (or the maximum profit). Thus, we can compute the cost as we apply operators and transition from state to state. This *path cost* or *cost function* is usually denoted by g. Given a problem which can be represented by a set of states and operators and then solved using a search algorithm, we can compare the quality of the solution by measuring the path cost. In the case of tic-tac-toe we have a limited search space. However, for many real-world problems the search space can grow very large, so we need algorithms that can scale to handle a large search space.

Effective search algorithms must do two things: cause motion or traversal of the state space, and do so in a controlled, systematic manner. Random search may work in some problems, but, in general, we need to search in an organized, methodical way. If we have a systematic search strategy that does not use information about the problem to help direct the search, it is called a *brute-force*, *uninformed*, or *blind* search. The only difference between the different brute-force search techniques is the order in which the nodes are expanded. But even slight changes in the order can have a significant impact on the behavior of the algorithm. Search algorithms which use information about the problem, such as the cost or distance to the goal state, are called *heuristic*, *informed*, or *directed* searches. The primary advantage of heuristic search algorithms is that we can make better choices concerning which node to expand next. This substantially improves the efficiency of the search algorithms.

There are several aspects of the performance of a search algorithm that are important to recognize [Russell and Norvig 1995]. An algorithm is *optimal* if it will find the best solution from among several possible solutions. A strategy is *complete* if it guarantees that it will find a solution if one exists. The efficiency of the algorithm, in terms of *time complexity* (how long it takes to find a solution) and *space complexity* (how much memory it requires), is a major practical consideration. Having an optimal search algorithm, guaranteed to find the best solution, has little practical value if it takes hours to complete, when we only have minutes to make a decision. Similarly, having a complete algorithm that is a memory hog is useless if it runs out of memory just before it finds the solution to your problem.

Search Strategies

In this section we examine two basic search techniques used to solve problems in AI, breadth-first search and depth-first search. Later we will explore enhancements to these algorithms, but first we need to make sure we understand how these basic approaches work. We will work through several examples, using the map shown in Figure 2.1.

First, we need to define the problem we are trying to solve. In this case, it is quite simple. Given a starting point at any one of the cities on the map, can we find any other city on the map as long as there is a path from the start city to the end or goal city? At this time, we are not trying to find the shortest path or anything like that. We simply want to find whether the goal city is on the map.

Now that we have defined the problem, the next step is to decide how to represent the problem as a state space. A map like the one in Figure 2.1 can be naturally represented by a graph data structure, where the cities' names are the nodes, and the major roadways between cities are the links or edges of the graph. So, from a programming perspective, our problem is to traverse a graph data structure in a systematic way until we either find the goal city or exhaust all possibilities. Hopefully having the entire state space shown on a map will make understanding the operations of the search algorithms easier. In more complex problems, all we have is the single start state and a set of operators that are used to generate new states. The search algorithms work the same way,

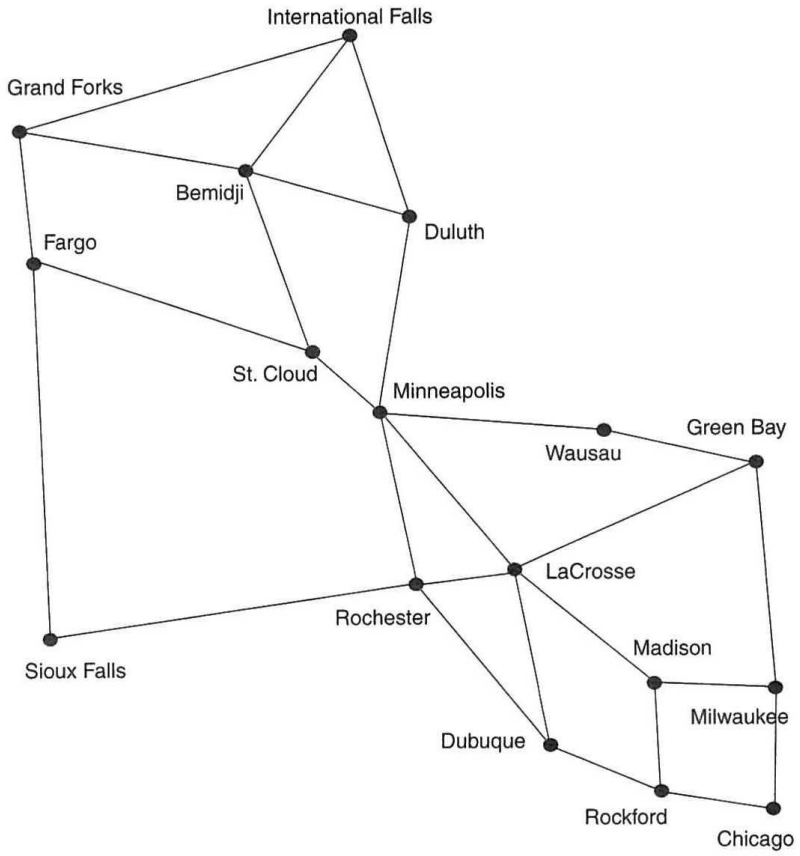

Figure 2.1 Map of midwestern U.S. cities.

but conceptually, we dynamically grow or expand the graph, instead of having it entirely specified at the start.

Breadth-First Search

The breadth-first search algorithm searches a state space by constructing a hierarchical tree structure consisting of a set of nodes and links. The algorithm defines a way to move through the tree structure, examining the values at nodes in a controlled and systematic way, so that we can find a node that offers a solution to the problem we have represented using the tree structure.

The algorithm is as follows:

1. Create a queue and add the first **SearchNode** to it.
2. Loop:
   a. If the queue is empty, quit.
   b. Remove the first **SearchNode** from the queue.

c. If the **SearchNode** contains the goal state, then exit with the **SearchNode** as the solution.

d. For each child of the current **SearchNode**, add the new **SearchNode** to the back of the queue.

The breadth-first algorithm spreads out in a uniform manner from the start node. From the start node, it looks at each node that is one edge away. Then it moves out from those nodes to all nodes two edges away from the start node. This continues until either the goal node is found or the entire tree is searched. Breadth-first search is complete; it will find a solution if one exists. But it is neither optimal in the general case (it won't find the best solution, just the first one that matches the goal state), nor does it have good time or space complexity (it grows exponentially in time and memory consumption).

Let's walk through an example to see how breadth-first search could find a city on our map. Our search begins in **Rochester**, and we want to know if we can get to **Wausau** from there. The **Rochester** node is placed on the queue in Step 1. Next we enter our search loop at Step 2. We remove **Rochester**, the first node, from the queue. **Rochester** does not contain our goal state (**Wausau**) so we expand it by taking each child node in **Rochester**, and adding them to the back of the queue. So we add **Sioux Falls**, **Minneapolis**, **LaCrosse**, and **Dubuque** to our search queue. Now we are back at the top of our loop. We remove the first node from the queue (**Sioux Falls**) and test it to see if it is our goal state. It is not, so we expand it, adding **Fargo** and **Rochester** to the end of our queue, which now contains [**Minneapolis**, **LaCrosse**, **Dubuque**, **Fargo**, **Rochester**]. We remove **Minneapolis**, the goal test fails, and we expand that node, adding **St. Cloud**, **Wausau**, **Duluth**, **LaCrosse**, and **Rochester** to the search queue, now holding [**LaCrosse**, **Dubuque**, **Fargo**, **Rochester**, **St. Cloud**, **Wausau**, **Duluth**, **LaCrosse**, **Rochester**]. We test **LaCrosse** and then expand it, adding **Minneapolis**, **Green Bay**, **Madison**, **Dubuque**, and **Rochester** to the list, which has now grown to [**Dubuque**, **Fargo**, **Rochester**, **St. Cloud**, **Wausau**, **Duluth**, **LaCrosse**, **Rochester**, **Minneapolis**, **Green Bay**, **Madison**, **Dubuque**, **Rochester**]. We remove **Dubuque** and add **Rochester**, **LaCrosse**, and **Rockford** to the search queue.

At this point, we have tested every node that is one level away in the tree from the start node (**Rochester**). Our search queue contains the following nodes: [**Fargo**, **Rochester**, **St. Cloud**, **Wausau**, **Duluth**, **LaCrosse**, **Rochester**, **Minneapolis**, **Green Bay**, **Madison**, **Dubuque**, **Rochester**, **Rochester**, **LaCrosse**, **Rockford**]. We remove **Fargo**, which is two levels away from **Rochester**, and add **Grand Forks**, **St. Cloud**, and **Sioux Falls**. Then we test and expand **Rochester** (**Rochester** to **Minneapolis** to **Rochester** is two levels away from our start). Next is **St. Cloud**; again we expand that node. Finally, we get to **Wausau**; our goal test succeeds and we declare success. Our search order was **Rochester**, **Sioux Falls**, **Minneapolis**, **LaCrosse**, **Dubuque**, **Fargo**, **Rochester**, **St. Cloud**, and **Wausau**.

Note that this trace could have been greatly simplified by keeping track of nodes that had been tested and expanded. This would have cut down on our time and space complexity, yet still would have been a complete algorithm because we would have searched every node before we stopped. However, in more realistic problems where each node is expanded using a list of operators and the states are more complex than

just strings, it is not so easy to determine that two states are identical. This explosion of nodes and states is not unusual for a breadth-first search problem.

Depth-First Search

Depth-first search is another way to systematically traverse a tree structure to find a goal or solution node. Instead of completely searching each level of the tree before going deeper, the depth-first algorithm follows a single branch of the tree down as many levels as possible until it either reaches a solution or a dead end. The algorithm follows:

1. Create a queue and add the first **SearchNode** to it.
2. Loop:
   a. If the queue is empty, quit.
   b. Remove the first **SearchNode** from the queue.
   c. If the **SearchNode** contains the goal state, then exit with the **SearchNode** as the solution.
   d. For each child of the current **SearchNode**: Add the new **SearchNode** to the front of the queue.

This algorithm is identical to the breadth-first search with the exception of the last step in the loop. The depth-first algorithm searches from the start or root node all the way down to a leaf node. If it does not find the goal node, it backtracks up the tree and searches down the next untested path until it reaches the next leaf. If you imagine a large tree, the depth-first algorithm may spend a large amount of time searching the paths on the lower left when the answer is really in the lower right. But since depth-first search is a brute-force method, it will blindly follow this search pattern until it comes across a node containing the goal state, or until it searches the entire tree. Depth-first search has lower memory requirements than breadth-first search. For problems with extremely deep or infinite search trees, depth-first search can spend all of its time searching one deep branch of the tree, looping forever as it dynamically expands nodes. For this reason, depth-first search is neither complete nor optimal.

As we did earlier, let's walk through a simple example of how depth-first search would work if we started in **Rochester** and wanted to see if we could get to **Wausau**. Starting with **Rochester**, we test and expand it, placing **Sioux Falls**, then **Minneapolis**, then **LaCrosse**, then **Dubuque** at the front of the search queue: [**Dubuque**, **LaCrosse**, **Minneapolis**, **Sioux Falls**]. We remove **Dubuque** and test it; it fails, so we expand it adding **Rochester** to the front, then **LaCrosse**, then **Rockford**. Our search queue now looks like [**Rockford**, **LaCrosse**, **Rochester**, **LaCrosse**, **Minneapolis**, **Sioux Falls**]. We remove **Rockford**, and add **Dubuque**, **Madison**, and **Chicago** to the front of the queue in that order, yielding [**Chicago**, **Madison**, **Dubuque**, **LaCrosse**, **Rochester**, **LaCrosse**, **Minneapolis**, **Sioux Falls**]. We test **Chicago**, and place **Rockford** and **Milwaukee** on the queue. We take **Milwaukee** from the front and add **Chicago**, **Madison**, and **Green Bay** to the search queue. It is now [**Green Bay**, **Madison**, **Chicago**, **Rockford**, **Chicago**, **Madison**, **Dubuque**, **LaCrosse**, **Rochester**, **LaCrosse**, **Minneapolis**, **Sioux Falls**]. We remove **Green Bay** and add **Milwaukee**, **LaCrosse**, and

Wausau to the queue in that order. Finally, **Wausau** is at the front of the queue, our goal test succeeds, and our search ends. Our search order was **Rochester, Dubuque, Rockford, Chicago, Milwaukee, Green Bay, and Wausau.**

In this example, we again did not prevent tested nodes from being added to the search queue. As a result, we had duplicate nodes on the queue. In a depth-first search, this could be disastrous. We could have easily had a cycle or loop where we tested one city, then a second, then the first again, ad infinitum. In the next section, we show our Java implementation of these search algorithms, starting with a class for the node and one for the search graph. We include tests to avoid the duplication of nodes on the search queue.

The SearchNode Class

We define search problems using a hierarchical tree or graph structure, comprised of a set of nodes and links. Our search algorithms work on the node structure defined in the **SearchNode** class shown in Figure 2.2.

The first data member in our **SearchNode** class (see Figure 2.2) is the symbolic name or *label* of the node. Next is the *state*, which can be any **Object**. The *oper* is the definition of the operation that created the state of the object. This would be used in problems such as games where the state of a parent node would be expanded into multiple child nodes and states based on the application of the set of operators.

The *links* member is a **Vector** containing references to all other **SearchNode** objects to which this node is linked. **SearchNode** objects can be connected to form any graph including tree structures.

The *depth* member is an integer that defines the distance from the start or root node in any search. The two boolean flags, *expanded* and *tested*, are used by the search algorithms to avoid getting into infinite loops. The first, *expanded*, is set whenever the node is expanded during a search. The second, *tested*, is used whenever the state of the node is tested during a search. Depending on the search algorithm, this flag may or may not be used. The *cost* member can be used to represent the current accumulated cost or any other cost measure associated with the search problem. The static member *traceTextArea* is used to display trace information in our example application.

The constructor takes two parameters, the *name* and the initial object state. The constructor initializes the *depth*, *links*, and *oper* members and sets the boolean flags to false.

Once we have created an instance of a **SearchNode**, we specify the links to other nodes. The *addLink()* method adds a link to a single **SearchNode** object. For graphs with high connectivity, the *addLinks()* method is provided to add links to an arbitrary number of nodes by passing an array of **SearchNode** objects as a parameter to the method.

We provide a method for testing whether the node is a leaf node in a tree structure (i.e., it has no children) and methods to set and get the *depth*, *oper*, *expanded*, and *tested* data members.

The *reset()* method is used to reset the node depth and the boolean flags before starting a search. The *setDisplay()* method is used in our example application to register a

```
package search;

import java.util.*;
import javax.swing.*;

public class SearchNode extends Object {
  protected String label;
  protected Object state;
  protected Object oper;
  protected Vector links;
  protected int depth;
  protected boolean expanded;
  protected boolean tested;
  protected float cost = 0;
  private static JTextArea traceTextArea;
  public static final int FRONT = 0;
  public static final int BACK = 1;
  public static final int INSERT = 2;

  SearchNode(String label, Object state) {
    this.label = label;
    this.state = state;
    depth = 0;
    links = new Vector();
    oper = null;
    expanded = false;
    tested = false;
  }

  public void addLink(SearchNode node) {
    links.addElement(node);
  }

  public void addLinks(SearchNode[] nodes) {
    for(int i = 0; i < nodes.length; i++) {
      links.addElement(nodes[i]);
    }
  }

  public boolean leaf() {
    return (links.size() == 0);
  }

  public void setDepth(int depth) {
    this.depth = depth;
  }

  public void setOperator(Object oper) {
```

Figure 2.2 The SearchNode class listing.

```
    this.oper = oper;
  }

  public void setExpanded() {
    expanded = true;
  }

  public void setExpanded(boolean state) {
    expanded = state;
  }

  public boolean isExpanded() {
    return expanded;
  }

  public void setTested() {
    tested = true;
  }

  public void setTested(boolean state) {
    tested = state;
  }

  public boolean isTested() {
    return tested;
  }

  static public void setDisplay(JTextArea textArea) {
    traceTextArea = textArea;
  }

  public Object getState() {
    return state;
  }

  public void reset() {
    depth = 0;
    expanded = false;
    tested = false;
  }

  public void trace() {
    String indent = new String();

    for(int i = 0; i < depth; i++) {
      indent += "  ";
    }
```

(continues)

Figure 2.2 Continued.

```
        traceTextArea.append(indent + "Searching " + depth + ": " + label
            + " with state = " + state + "\n");
    }

    public void expand(Vector queue, int position) {
        setExpanded();
        for(int j = 0; j < links.size(); j++) {
            SearchNode nextNode = (SearchNode) links.elementAt(j);

            if(!nextNode.tested) {
                nextNode.setTested(true);
                nextNode.setDepth(depth + 1);
                switch(position) {
                    case FRONT:
                        queue.insertElementAt(nextNode, 0);
                        break;
                    case BACK:
                        queue.addElement(nextNode);
                        break;
                    case INSERT:
                        boolean inserted = false;
                        float nextCost = nextNode.cost;

                        for(int k = 0; k < queue.size(); k++) {
                            if(nextCost < ((SearchNode) queue.elementAt(k)).cost) {
                                queue.insertElementAt(nextNode, k);
                                inserted = true;
                                break;
                            }
                        }
                        if(!inserted) {
                            queue.addElement(nextNode);
                        }
                        break;
                }
            }
        }
    }
}
```

Figure 2.2 The SearchNode class listing (Continued).

JTextArea component for displaying trace information during a search. The *trace()* method writes out an indented string indicating the depth of the node in the search tree along with its label and state. As written, it assumes that the state is a string.

The most complicated method in our **SearchNode** class is the *expand()* method, which is used by the various search algorithms to build a search tree from the initial

search graph. The parameters are a *queue* (a **Vector** instance) and a parameter specifying where the child nodes should be placed on the *queue*. The options are *front*, *back*, or *insert*, based on the current cost in the **SearchNode**.

When expanding a node, we first mark the node as expanded. We then loop over all of the nodes to which the node has links. For each node, if it has not been *tested* (actually, this means placed on the *queue*, because the node states are tested only when they are removed from the front of the *queue*), we mark it as *tested*, set its *depth* in the search tree, and then place it on the *queue* in the specified position. For FRONT, we add at position 0. For BACK we simply *addElement()*, which places it at the end of the **Vector**.

For the INSERT case, we have a loop where we compare the cost of the node we are trying to insert, *nextNode*, to the cost of the nodes on the *queue*. If *nextCost* is lower, we place the *nextNode* on the *queue*. There are two cases where the *nextNode* will not be inserted in this loop: when the *queue* is empty, and when the *nextCost* is greater than the cost of all of the nodes already on the *queue*. In this case, the boolean flag, *inserted*, will not be set and *nextNode* will be added to the end of the *queue*.

Now that we have defined the **SearchNode** class, we can talk about the **SearchGraph** class, which contains the set of **SearchNode** objects that define our problem states. **SearchGraph** is a relatively simple class, as shown in Figure 2.3. It contains only two instance variables: the *graph* that is a **Hashtable** of (*label*, **SearchNode**) pairs and a *name* which is a **String** that identifies the **SearchGraph**. The constructor takes a single **String** parameter for the **SearchGraph** name. The *reset()* method uses the enumeration of the elements in the *graph* to iterate over the **SearchNode** objects and reset each one in turn. The *put()* method takes a **SearchNode** object as a single argument, and adds it to the *graph* with the *label* as the key and the **SearchNode** object as the value.

The remainder of the methods in the **SearchGraph** class define four of the five search methods that are described in more detail in the next section.

Search Application

In this section, we develop a Java application to illustrate the behavior of five search algorithms. The Search application classes are **SearchApp**, which contains our *main()*, and **SearchFrame**, which implements our main window. They were constructed using the Borland/Inprise JBuilder 3.0 Java interactive development environment. The visual builder allows us to create the complete GUI using Java Swing components in a drag and drop style. JBuilder automatically generates the Java code for creating the GUI controls and action event handlers in the **SearchFrame** class. We then added the logic for creating the **SearchGraph** and **SearchNodes** and invoking the search algorithms. The **SearchApp** code that invokes the **SearchFrame** is not presented here. It simply instantiates the **SearchFrame** class and displays it. Likewise, a substantial amount of the code in the **SearchFrame** class deals with GUI control logic and we will not present that

```
package search;

import java.util.*;

class SearchGraph extends Object {
  String name;
  Hashtable graph = new Hashtable();

  public SearchGraph(String name) {
    this.name = name;
  }

  public Hashtable getGraph() {
    return graph;
  }

  public SearchNode getNode(String nodeName) {
    return (SearchNode) graph.get(nodeName);
  }

  void reset() {
    Enumeration enum = graph.elements();

    while(enum.hasMoreElements()) {
      SearchNode nextNode = (SearchNode) enum.nextElement();

      nextNode.reset();
    }
  }

  void put(SearchNode node) {
    graph.put(node.label, node);
  }

  public SearchNode depthFirstSearch(SearchNode initialNode,
      Object goalState) {
    Vector queue = new Vector();

    queue.addElement(initialNode);
    initialNode.setTested(true);
    while(queue.size() > 0) {
      SearchNode testNode = (SearchNode) queue.firstElement();

      queue.removeElementAt(0);
      testNode.trace();
      if(testNode.getState().equals(goalState)) {
        return testNode;
```

Figure 2.3 The SearchGraph class listing.

```
      }
      if(!testNode.isExpanded()) {
        testNode.expand(queue, SearchNode.FRONT);
      }
    }
    return null;
  }

  public SearchNode breadthFirstSearch(SearchNode initialNode,
      Object goalState) {
    Vector queue = new Vector();

    queue.addElement(initialNode);
    initialNode.setTested(true);
    while(queue.size() > 0) {
      SearchNode testNode = (SearchNode) queue.firstElement();

      queue.removeElementAt(0);
      testNode.trace();
      if(testNode.getState().equals(goalState)) {
        return testNode;
      }
      if(!testNode.isExpanded()) {
        testNode.expand(queue, SearchNode.BACK);
      }
    }
    return null;
  }

  public SearchNode depthLimitedSearch(SearchNode initialNode,
      Object goalState, int maxDepth) {
    Vector queue = new Vector();

    queue.addElement(initialNode);
    initialNode.setTested(true);
    while(queue.size() > 0) {
      SearchNode testNode = (SearchNode) queue.firstElement();

      queue.removeElementAt(0);
      testNode.trace();
      if(testNode.getState().equals(goalState)) {
        return testNode;
      }
      if(testNode.depth < maxDepth) {
        if(!testNode.isExpanded()) {
          testNode.expand(queue, SearchNode.FRONT);
        }
```

(continues)

Figure 2.3 Continued.

```
      }
    }
    return null;
  }

  public SearchNode iterDeepSearch(SearchNode startNode,
      Object goalState) {
    int maxDepth = 10;

    for(int j = 0; j < maxDepth; j++) {
      reset();
      SearchNode answer = depthLimitedSearch(startNode, goalState, j);

      if(answer != null) {
        return answer;
      }
    }
    return null;
  }

  public SearchNode bestFirstSearch(SearchNode initialNode,
      Object goalState) {
    Vector queue = new Vector();

    queue.addElement(initialNode);
    initialNode.setTested(true);
    while(queue.size() > 0) {
      SearchNode testNode = (SearchNode) queue.firstElement();

      queue.removeElementAt(0);
      testNode.trace();
      if(testNode.getState().equals(goalState)) {
        return testNode;
      }
      if(!testNode.isExpanded()) {
        testNode.expand(queue, SearchNode.INSERT);
      }
    }
    return null;
  }
}
```

Figure 2.3 The SearchGraph class listing (Continued).

material here. Instead, we will highlight only brief sections of the code that demonstrate how to use the underlying application classes. All of the classes discussed in this chapter are in the *search* package shown in Figure 2.4.

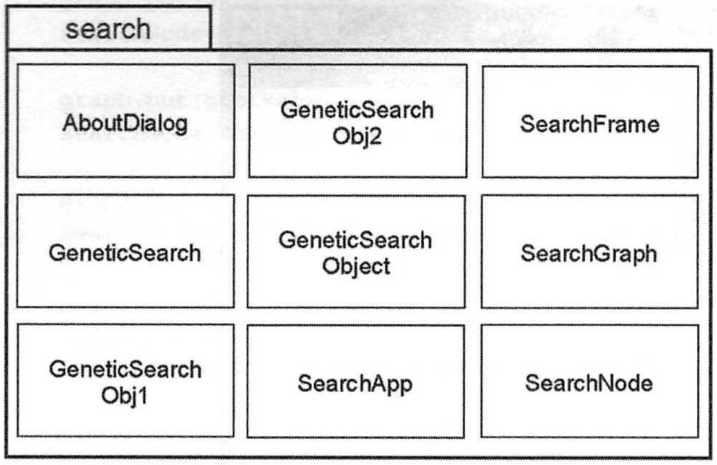

Figure 2.4 The *search* package UML diagram.

The user interface of our application is comprised of a single **JFrame**, shown in Figure 2.5. A user can select from one of five search techniques: depth-first, breadth-first, iterated-deepening, best-first, and genetic search. We have already presented the breadth-first and depth-first algorithms in some detail. We will discuss all five algorithms and their implementations in the following sections. The user can choose which algorithm to use by selecting the appropriate menu item under the **Algorithm** pull-down menu. For algorithms that use the **SearchGraph** class, the **Start node** and **Goal state** can be selected from the top two combo boxes. If the **Genetic search algorithm** is selected, the bottom four controls are enabled, allowing the user to specify the **Genetic object class name**, the **Population size**, the **Number of generations** to be processed, and the **Fitness threshold** used for early stopping. Pressing the **Start** menu item under the **File** pull-down menu will invoke the corresponding search algorithm with the specified parameters passed as arguments. As the search algorithms progress, trace information is displayed in the text area at the top of the dialog. Pressing the **Clear** menu item under the File pull-down menu will clear this area between runs, if desired.

A single static method *buildTestGraph()* is defined in the **SearchFrame** class which creates a **SearchGraph** used in our examples of the different search algorithms. This graph defines the set of midwestern U.S. cities as shown in Figure 2.1. First we instantiate the **SearchGraph** object and then create one **SearchNode** for each city. After each **SearchNode** object is created, it is added to the **SearchGraph** by using the *put()* method. Note that the node *name* and the *state* (also the city's name) are identical. This is for illustration purposes only. There is no reason why the node *label* and the *state* have to be the same. In fact, the second parameter on the **SearchNode** constructor can be any Java **Object**. So we could have an array or matrix for tic-tac-toe, or some other arbitrarily complex state-space representation, depending on the problem we are trying to solve.

```
public static SearchGraph buildTestGraph() {
  SearchGraph graph = new SearchGraph("test");
  SearchNode roch = new SearchNode("Rochester", "Rochester");
```

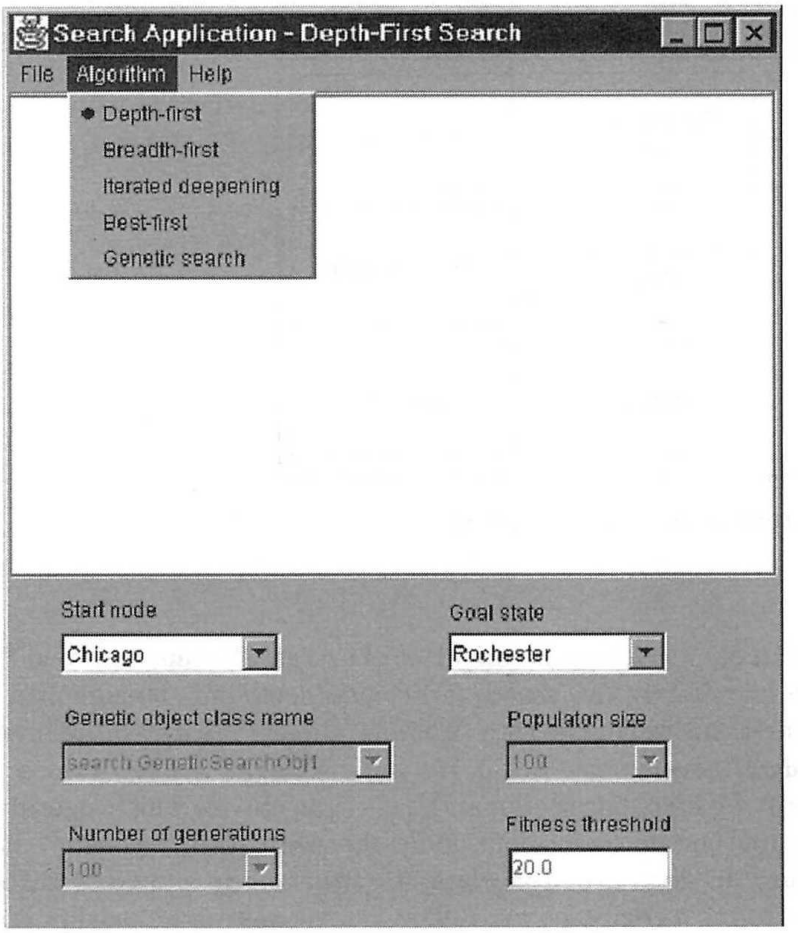

Figure 2.5 The search application window.

```
graph.put(roch);
SearchNode sfalls = new SearchNode("Sioux Falls", "Sioux Falls");

graph.put(sfalls);
SearchNode mpls = new SearchNode("Minneapolis", "Minneapolis");

graph.put(mpls);
SearchNode lacrosse = new SearchNode("LaCrosse", "LaCrosse");

graph.put(lacrosse);
SearchNode fargo = new SearchNode("Fargo", "Fargo");

graph.put(fargo);
SearchNode stcloud = new SearchNode("St.Cloud", "St.Cloud");

graph.put(stcloud);
SearchNode duluth = new SearchNode("Duluth", "Duluth");

graph.put(duluth);
SearchNode wausau = new SearchNode("Wausau", "Wausau");
```

```
graph.put(wausau);
SearchNode gforks = new SearchNode("Grand Forks", "Grand Forks");

graph.put(gforks);
SearchNode bemidji = new SearchNode("Bemidji", "Bemidji");

graph.put(bemidji);
SearchNode ifalls = new SearchNode("International Falls",
                      "International Falls");

graph.put(ifalls);
SearchNode gbay = new SearchNode("Green Bay", "Green Bay");

graph.put(gbay);
SearchNode madison = new SearchNode("Madison", "Madison");

graph.put(madison);
SearchNode dubuque = new SearchNode("Dubuque", "Dubuque");

graph.put(dubuque);
SearchNode rockford = new SearchNode("Rockford", "Rockford");

graph.put(rockford);
SearchNode chicago = new SearchNode("Chicago", "Chicago");

graph.put(chicago);
SearchNode milwaukee = new SearchNode("Milwaukee",
                         "Milwaukee");

graph.put(milwaukee);
roch.addLinks(new SearchNode[]{ mpls, lacrosse, sfalls, dubuque });
mpls.addLinks(new SearchNode[]{ duluth, stcloud, wausau });
mpls.addLinks(new SearchNode[]{ lacrosse, roch });
lacrosse.addLinks(new SearchNode[]{ madison, dubuque, roch });
lacrosse.addLinks(new SearchNode[]{ mpls, gbay });
sfalls.addLinks(new SearchNode[]{ fargo, roch });
fargo.addLinks(new SearchNode[]{ sfalls, gforks, stcloud });
gforks.addLinks(new SearchNode[]{ bemidji, fargo, ifalls });
bemidji.addLinks(new SearchNode[]{ gforks, ifalls, stcloud, duluth });
ifalls.addLinks(new SearchNode[]{ bemidji, duluth, gforks });
duluth.addLinks(new SearchNode[]{ ifalls, mpls, bemidji });
stcloud.addLinks(new SearchNode[]{ bemidji, mpls, fargo });
dubuque.addLinks(new SearchNode[]{ lacrosse, rockford });
rockford.addLinks(new SearchNode[]{ dubuque, madison, chicago });
chicago.addLinks(new SearchNode[]{ rockford, milwaukee });
milwaukee.addLinks(new SearchNode[]{ gbay, chicago });
gbay.addLinks(new SearchNode[]{ wausau, milwaukee, lacrosse });
wausau.addLinks(new SearchNode[]{ mpls, gbay });
roch.cost = 0;
sfalls.cost = 232;
mpls.cost = 90;
```

```
    lacrosse.cost = 70;
    dubuque.cost = 140;
    madison.cost = 170;
    milwaukee.cost = 230;
    rockford.cost = 210;
    chicago.cost = 280;
    stcloud.cost = 140;
    duluth.cost = 180;
    bemidji.cost = 260;
    wausau.cost = 200;
    gbay.cost = 220;
    fargo.cost = 280;
    gforks.cost = 340;
    return graph;
}
```

After the **SearchNode** objects are created, we define the connectivity between nodes using the *addLinks()* method. Next, we set the *cost* of each node as the distance from each city to Rochester, Minnesota. We use this in the best-first search example discussed later in this section. This cost value is set to a static value for this example. In most search problems, the cost would be computed as the search progresses. This is another simplification used to make the explanation clearer.

When the **Start** menu item is pressed, an **ActionEvent** is generated and the *StartMenuItem_actionPerformed()* method is invoked. The search algorithms are run on a separate thread from the GUI. We create a new thread, *runnit*, and start it. Because the **SearchFrame** implements the **Runnable** interface, the *run()* method is called. In *run()*, first we retrieve the *start* and *goal* **Strings** from the combo box controls. Next we clear all of the **SearchNode** objects in the graph by calling *reset()*. Then we call the selected search algorithm. Only one **Algorithm** radio button menu item can be selected at any time. When pressed, the menu item event handler methods set the *searchAlgorithm* data member value accordingly. The *run()* method invokes the corresponding search method based on the *searchAlgorithm* value. For the four graph-based search techniques this requires only a single method call. The genetic search algorithm, discussed in more detail later in this chapter, must retrieve the user settings from the four GUI controls first, and then invoke the search method.

```
void StartMenuItem_actionPerformed(ActionEvent e) {
  runnit = new Thread(this);
  runnit.start();
}

public void run() {
  int method = 0;
  SearchNode answer = null;
  SearchNode startNode;
  String start = (String) startComboBox.getSelectedItem();

  startNode = graph.getNode(start);
  String goal = (String) goalComboBox.getSelectedItem();
```

```
graph.reset();
switch(searchAlgorithm) {
  case DEPTH_FIRST: {
    traceTextArea.append("\n\nDepth-First Search for " + goal
        + ":\n\n");
    answer = graph.depthFirstSearch(startNode, goal);
    break;
  }
  case BREADTH_FIRST: {
    traceTextArea.append("\n\nBreadth-First Search for " + goal
        + ":\n\n");
    answer = graph.breadthFirstSearch(startNode, goal);
    break;
  }
  case ITERATED: {
    traceTextArea.append("\n\nIterated-Deepening Search for "   + goal
        + ":\n\n");
    answer = graph.iterDeepSearch(startNode, goal);
    break;
  }
  case BEST_FIRST: {
    traceTextArea.append("\n\nBest-First Search for " + goal
        + ":\n\n");
    goalComboBox.setSelectedItem("Rochester");
    answer = graph.bestFirstSearch(startNode, "Rochester");
    break;
  }
  case GENETIC_SEARCH: {
    traceTextArea.append("\n\nGenetic Search using... \n\n");
    geneticSearch.setGeneticObjectClassName(
      (String) GeneticObjClassComboBox.getSelectedItem());
    geneticSearch.setMaxNumPasses(
      ((Integer) NumGenerationsComboBox.getSelectedItem())
       .intValue());
    geneticSearch.setDebugOn(debugOn);
    geneticSearch.setPopulationSize(
      ((Integer) PopulationSizeComboBox.getSelectedItem())
       .intValue());
    try {
      geneticSearch.setFitnessThreshold(
        Double.valueOf( FitnessThresholdTextField.getText().trim())
         .doubleValue());
      } catch(Exception e) {
      geneticSearch.setFitnessThreshold(20.0);
      FitnessThresholdTextField.setText("20.0");
    }
    geneticSearch.init();
    geneticSearch.search();
    break;
  }
}
```

```
    if(searchAlgorithm != GENETIC_SEARCH) {
      if(answer == null) {
        traceTextArea.append("Could not find answer!\n");
      } else {
        traceTextArea.append("Answer found in node " + answer.label);
      }
    }
  }
```

Breadth-First Search

The Java implementation of the breadth-first search algorithm is shown in the following example. The *breadthFirstSearch()* method is part of the **SearchGraph** class. The parameters are the initial **SearchNode** and the goal state, which in our Search application is the object label. First we instantiate our queue of nodes to search, then add the *initialNode* to the queue and mark it as tested.

In a *while()* loop, we first check to see if there are any more nodes to expand. If so, we remove the first node, call *trace()* to print out a message, then test the node to see if it matches our goal state. Although in our example application we are using **Strings** to represent our goal state, this code should work for any state **Object** provided it implements the *equals()* method. If the goal test succeeds, we return with the node that matches the goal state. If the goal test fails and we haven't already expanded this node, we expand it by placing all of the nodes it has links to on the back of the queue. When the queue is empty we exit the *while()* loop.

```
public SearchNode breadthFirstSearch(SearchNode initialNode,
    Object goalState) {
  Vector queue = new Vector();

  queue.addElement(initialNode);
  initialNode.setTested(true);
  while(queue.size() > 0) {
    SearchNode testNode = (SearchNode) queue.firstElement();

    queue.removeElementAt(0);
    testNode.trace();
    if(testNode.getState().equals(goalState)) {
      return testNode;
    }
    if(!testNode.isExpanded()) {
      testNode.expand(queue, SearchNode.BACK);
    }
  }
  return null;
}
```

A breadth-first search performs a complete search of the state space. If there are multiple solutions, it will find a solution with the shortest number of steps because each node of length n is examined before the next layer of nodes is explored. In Figure 2.6 we show

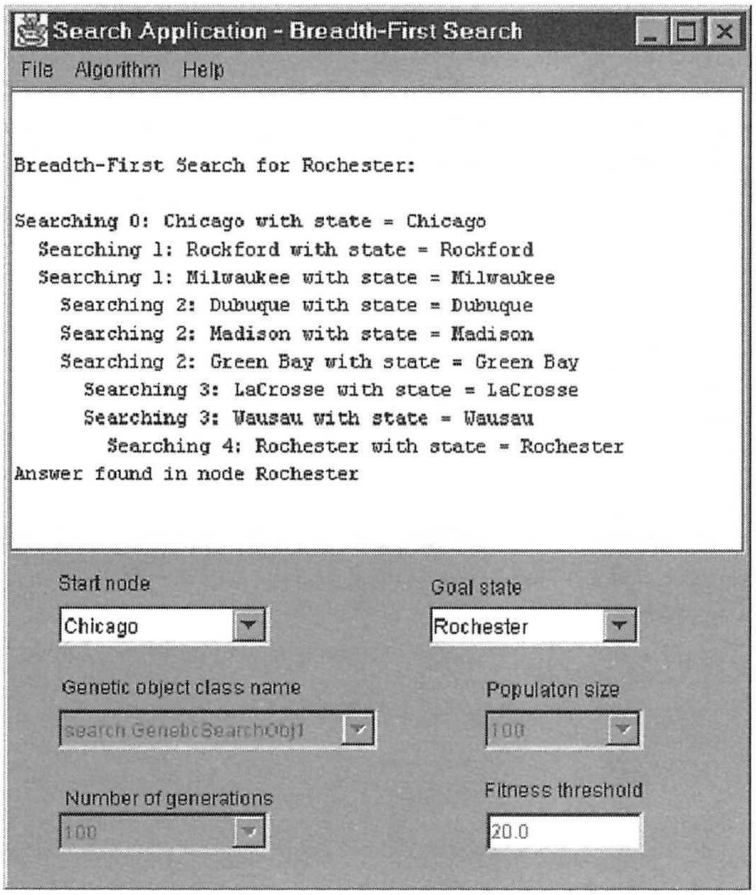

Figure 2.6 Breadth-first search algorithm example.

the output of a breadth-first search run against our *testGraph* with Chicago as the start node and Rochester as the destination. Note that the exact search order is affected by how we define the links in our *testGraph*.

Depth-First Search

The Java code for our depth-first search algorithm is similar to our breadth-first code, except the child nodes are added to the front of the queue instead of the back. This small change makes a big difference in the search behavior of the algorithm. Depth-first search requires less memory than breadth-first search but it can also spend lots of time exploring paths that lead to dead ends.

The *depthFirstSearch()* parameters are the starting **SearchNode** and the goal state. First we instantiate our *queue* of nodes to search, and add the *initialNode* to the queue and mark it as tested.

In a *while()* loop, we first check to see if there are any more nodes to expand. If so, we remove the first node, call *trace()* to print out a message, then test the node to see if it matches our goal state. Although in our example application we are using **Strings** to

represent our goal state, this code should work for any state **Object** provided it implements the *equals()* method. If the goal test succeeds, we return with the node that matched the goal state. If the goal test fails and we haven't already expanded this node, we expand it by placing all of the nodes it has links to on the front of the *queue*. When the *queue* is empty we exit the *while()* loop.

```
public SearchNode depthFirstSearch(SearchNode initialNode,
    Object goalState) {
  Vector queue = new Vector();

  queue.addElement(initialNode);
  initialNode.setTested(true);
  while(queue.size() > 0) {
    SearchNode testNode = (SearchNode) queue.firstElement();

    queue.removeElementAt(0);
    testNode.trace();
    if(testNode.getState().equals(goalState)) {
      return testNode;
    }
    if(!testNode.isExpanded()) {
      testNode.expand(queue, SearchNode.FRONT);
    }
  }
  return null;
}
```

Figure 2.7 shows depth-first search results using our search application, with Chicago as the start node and Rochester as the destination.

Improving Depth-First Search

One easy way to get the best characteristics of the depth-first search algorithm along with the advantages of the breadth-first search is to use a technique called iterative-deepening search. In this approach, we first add a slight modification to the depth-first search algorithm by adding a parameter called *maxDepth*, to limit our search to a maximum depth of the tree. Then we add a control loop in which we continually deepen our depth-first search until we find the solution.

This algorithm, like standard breadth-first search, is a complete search and will find an optimal solution. But, like the depth-first algorithm, it has much lower memory requirements. Although we are retracing ground when we increase our depth of search, this approach is still more efficient than pure breadth-first or pure unlimited depth-first search for large search spaces [Russell and Norvig 1995].

Our Java implementation of the iterated-deepening search algorithm uses two methods. The *iterDeepSearch()* method performs the outer loop, repeatedly calling *depthLimitedSearch()* with the *maxDepth* increasing by one for each successive call. The *depthLimitedSearch()* method is essentially the same as our standard depth-first search

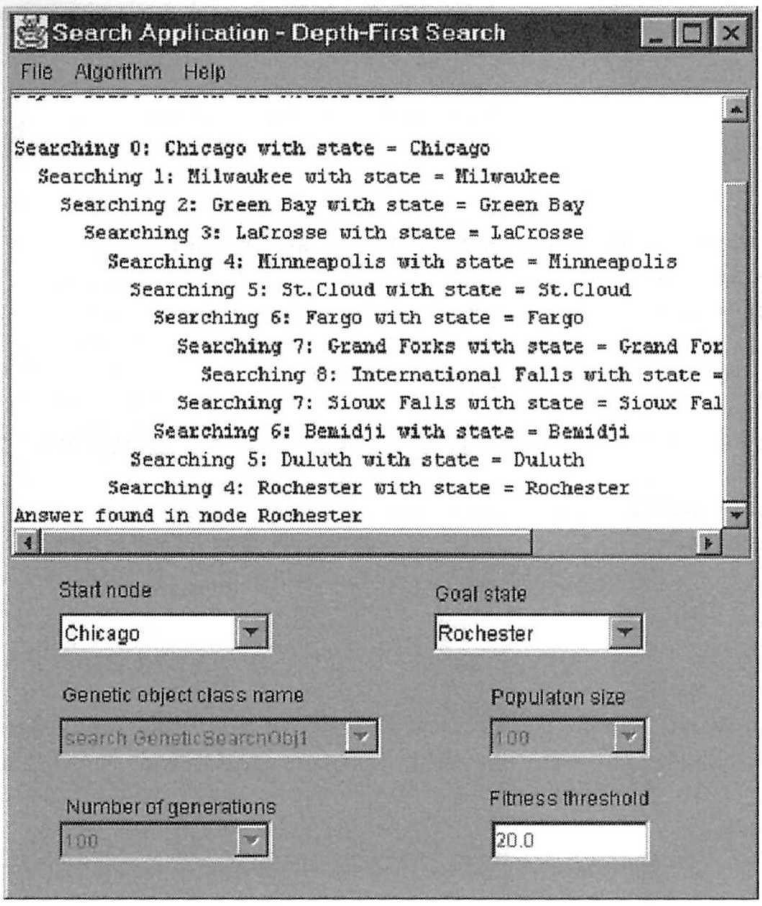

Figure 2.7 Depth-first search algorithm example.

algorithm, with the test against *maxDepth* used to short-circuit expansion of **SearchNode** once they get too deep.

```
public SearchNode depthLimitedSearch(SearchNode initialNode,
    Object goalState, int maxDepth) {
  Vector queue = new Vector();

  queue.addElement(initialNode);
  initialNode.setTested(true);
  while(queue.size() > 0) {
    SearchNode testNode = (SearchNode) queue.firstElement();

    queue.removeElementAt(0);
    testNode.trace();
    if(testNode.getState().equals(goalState)) {
      return testNode;
    }
    if(testNode.depth < maxDepth) {
      if(!testNode.isExpanded()) {
```

```
            testNode.expand(queue, SearchNode.FRONT);
          }
        }
      }
      return null;
    }

    public SearchNode iterDeepSearch(SearchNode startNode,
        Object goalState) {
      int maxDepth = 10;

      for(int j = 0; j < maxDepth; j++) {
        reset();
        SearchNode answer = depthLimitedSearch(startNode, goalState, j);

        if(answer != null) {
          return answer;
        }
      }
      return null;
    }
```

In Figure 2.8 we show a trace of a search from Chicago to Rochester using the iterated-deepening algorithm. Notice how the algorithm recovers ground as it goes to each level.

Heuristic Search

The Traveling Salesman Problem (TSP), in which a salesman makes a complete tour of the cities on his route and visits each city exactly once while traveling the shortest possible distance, is an example of a problem that has a combinatorial explosion. As such, it cannot be solved using breadth-first or depth-first search for problems of any realistic size. TSP belongs to a class of problems known as NP-hard or NP-complete. Unfortunately, there are many problems which have this form and which are essentially intractable. In these cases, finding the best possible answer is not computationally feasible, and so we have to settle for a good answer. In this section we discuss several heuristic search methods that attempt to provide a practical means for approaching these kinds of search problems.

Heuristic search methods are characterized by the sense that we have limited time and space in which to find an answer to complex problems, and so we are willing to accept a good solution. As such, we apply heuristics or rules of thumb as we search the tree to try to determine the likelihood that following one path or another is more likely to lead to a solution. Note that this is in stark contrast to the brute-force methods, which chug along merrily regardless of whether a solution is anywhere in sight.

Heuristic search methods use objective functions called (surprise!) heuristic functions to try to gauge the value of a particular node in the search tree and to estimate the value of following down any of the paths from the node. In the next sections we describe four types of heuristic search algorithms.

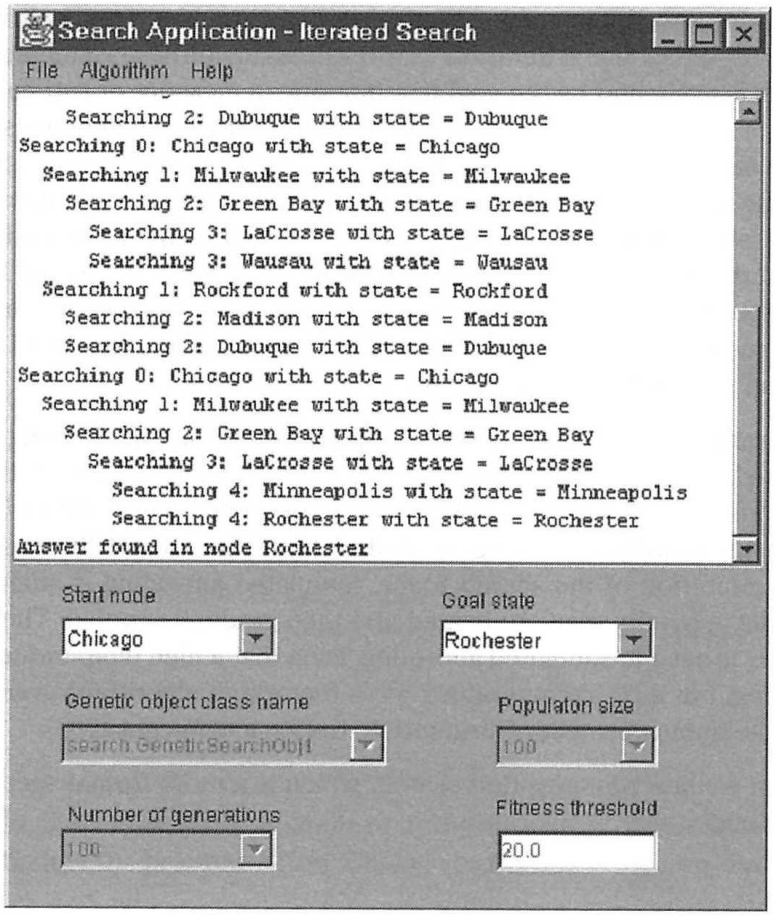

Figure 2.8 Iterated-deepening search algorithm example.

Generate-and-Test Algorithm

The generate-and-test algorithm is the most basic heuristic search function. The steps are as follows:

1. Generate a possible solution, either a new state or a path through the problem space.
2. Test to see if the new state or path is a solution by comparing it to a set of goal states.
3. If a solution has been found, return success; otherwise return to step 1.

This is a depth-first search procedure that performs an exhaustive search of the state space. If a solution is possible, the generate-and-test algorithm will find it. However, it may take an extremely long time. For small problems, generate-and-test can be an effective algorithm, but for large problems, the undirected search strategy leads to lengthy run times and is impractical. The major weakness of generate-and-test is that we get no feedback on which direction to search. We can greatly improve this algorithm by providing feedback through the use of heuristic functions.

Hill-climbing is an improved generate-and-test algorithm in which feedback from the tests is used to help direct the generation (and evaluation) of new candidate states. When a node state is evaluated by the goal test function, a measure or estimate of the distance to the goal state is also computed. One problem with a hill-climbing search in general is that the algorithm can get caught in local minima or maxima. Because we are always going in the direction of least cost, we can follow a path up to a locally good solution, while missing the globally excellent solution available just a few nodes away. Once at the top of the locally best solution, moving to any other node would lead to a node with lower goodness. Another possibility is that a plateau or flat spot exists in the problem space. Once the search algorithm gets up to this area all moves would have the same goodness and so progress would be halted.

To avoid getting trapped in sub-optimal states, variations on the hill-climbing strategy have been proposed. One is to inject noise into the evaluation function, with the initial noise level high and slowly decreasing over time. This technique, called *simulated annealing*, allows the search algorithm to go in directions that are not "best" but allow more complete exploration of the search space. Simulated annealing is analogous to annealing of metals, whereby they are heated and then gradually cooled. Thus a temperature parameter is used in simulated annealing, such that a high temperature allows continued searching, but as the temperature cools the search algorithm reverts to the more standard hill-climbing behavior [Kirkpatrick, Gelatt, and Vecchi 1983].

In the next section we describe best-first search, which is a more formal specification of a greedy, hill-climbing search algorithm.

Best-First Search

Best-first search is a systematic control strategy, combining the strengths of breadth-first and depth-first search into one algorithm. The main difference between best-first search and the brute-force search techniques is that we make use of an evaluation or heuristic function to order the **SearchNode** objects on the queue. In this way, we choose the **SearchNode** that appears to be best, before any others, regardless of their position in the tree or graph. Our implementation of best-first search uses the current value of the *cost* data member to order the **SearchNode** objects on the search queue. In our *buildTestGraph()* method, we set the cost of the nodes to be equal to the approximate straight-line distance from each city to Rochester, Minnesota.

The *bestFirstSearch()* parameters are the initial **SearchNode** and the goal state. To start, we instantiate our *queue* of nodes to search, and add the *initialNode* to the queue and mark it as tested.

In a *while()* loop, we first check to see if there are any more nodes to expand. If so, we remove the first node, call *trace()* to print out a message, then test the node to see if it matches our goal state. Although in our example application we are using **Strings** to represent our goal state, this code should work for any state **Object** provided it implements the *equals()* method. If the goal test succeeds, we return with the node that matched the goal state. If the goal test fails and we haven't already expanded this node,

we expand the node by inserting each of the nodes it has links to on the *queue* based on the cost. When the *queue* is empty we exit the *while()* loop.

```
public SearchNode bestFirstSearch(SearchNode initialNode,
    Object goalState) {
  Vector queue = new Vector();

  queue.addElement(initialNode);
  initialNode.setTested(true);
  while(queue.size() > 0) {
    SearchNode testNode = (SearchNode) queue.firstElement();

    queue.removeElementAt(0);
    testNode.trace();
    if(testNode.getState().equals(goalState)) {
      return testNode;
    }
    if(!testNode.isExpanded()) {
      testNode.expand(queue, SearchNode.INSERT);
    }
  }
  return null;
}
```

Figure 2.9 shows an example of the best-first search algorithm applied to the problem of finding the best route from Chicago to Rochester, Minnesota.

Greedy Search

Greedy search is a best-first strategy in which we try to minimize the estimated cost to reach the goal (certainly an intuitive approach!). Since we are greedy, we always expand the node that is estimated to be closest to the goal state. Unfortunately, the exact cost of reaching the goal state usually can't be computed, but we can estimate it by using a cost estimate or heuristic function *h()*. When we are examining node **n**, then $h(\mathbf{n})$ gives us the estimated cost of the cheapest path from **n**'s state to the goal state. Of course, the better an estimate *h()* gives, the better and faster we will find a solution to our problem. Greedy search has similar behavior to depth-first search. Its advantages are delivered via the use of a quality heuristic function to direct the search.

A* Search

One of the most famous search algorithms used in AI is the A* search algorithm, which combines the greedy search algorithm for efficiency with the uniform-cost search for optimality and completeness. In A* the evaluation function is computed by adding the two heuristic measures; the $h(\mathbf{n})$ cost estimate of traversing from **n** to the goal state, and $g(\mathbf{n})$ which is the known path cost from the start node to **n** into a function called $f(\mathbf{n})$.

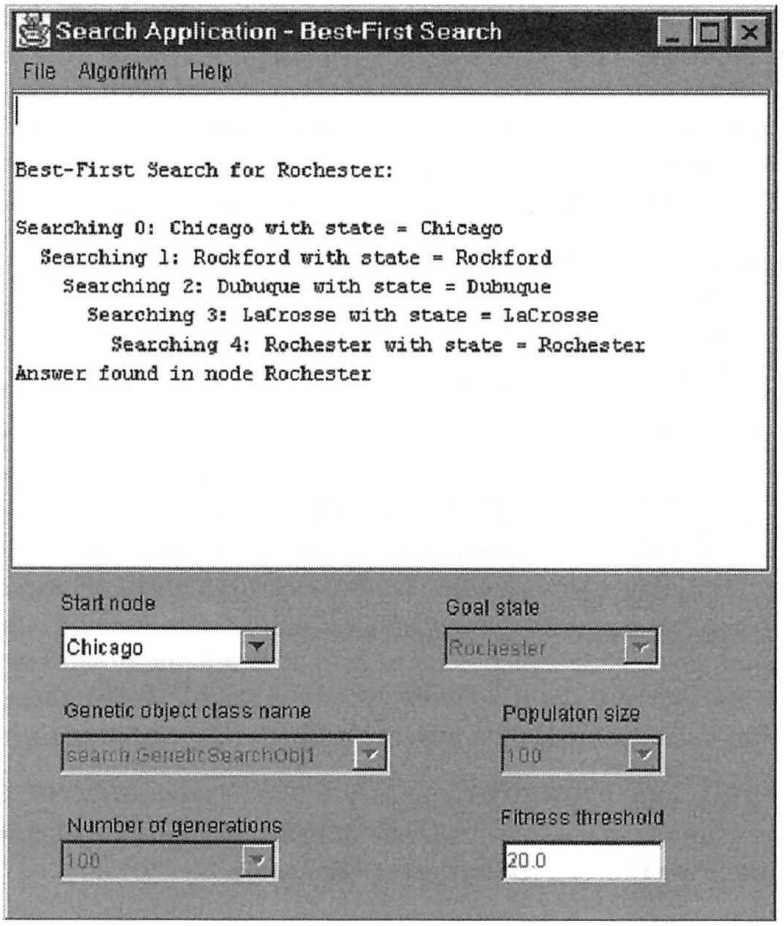

Figure 2.9 Best-first search algorithm example.

Again, there is nothing really new here from a search algorithm point of view; all we are doing is using better information (heuristics) to evaluate and order the nodes on our search queue. We know how much it cost to get where we are (node **n**) and we can guesstimate how much it will cost to reach the goal from **n**. Thus we are bringing all of the information about the problem we can to bear on directing our search. This combination of strategies turns out to provide A* with both completeness and optimality.

Constraint Satisfaction

Another approach to problem solving using search is called constraint satisfaction. All problems have some constraints that define what the acceptable solutions are. For example, if our problem is to load a delivery truck with packages, a constraint may be that the truck holds only 2000 pounds. This constraint could help us substantially reduce our search space by ignoring search paths that contain a set of items that weigh more than 2000 pounds. Constraint satisfaction search uses a set of constraints to define the space of acceptable solutions. Rather than a simple goal test, the search is complete

when we have a set of bindings of values to variables, a state where the minimum set of constraints holds.

The role of constraints is to bind variables to values or limit the range of values that they may take on, thus reducing the number of combinations we need to explore. First, an initial set of constraints is applied and propagated through the problem, subject to the dependencies that exist. For example, if we set a limit of 2000 pounds for a particular delivery truck, that may immediately remove some items from consideration. Also, once we assign an object to the truck that weighs 800 pounds, we can infer that now the limit for additional items is 1200 pounds. If a solution is not found by propagating the constraints, then search is required. This may involve backtracking or undoing an assignment or variable binding.

Means-Ends Analysis

Means-ends analysis is a problem-solving process based on detecting differences between states and then trying to reduce those differences. First used in the General Problem Solver [Newell and Simon 1963], means-ends analysis uses both forward and backward reasoning and a recursive algorithm to systematically minimize the differences between the initial and goal states.

Like any search algorithm, means-ends analysis has a set of states along with a set of operators that transform that state. However, the operators or rules for transformation do not completely specify the before and after states. The left side or antecedent of the rule only contains the subset of conditions, the *preconditions*, which must be true in order for the operator to be applied. Likewise the right-hand side of the rule identifies only those parts of the state that the operator changes. Each operator has a before and after list of states and state changes that are only a subset of the whole problem space. A simplified version of the means-ends analysis is described here [Rich and Knight 1991].

Means-Ends Analysis (Current-State, Goal-State)

1. Compare the current-state to the goal-state. If states are identical then return success.
2. Select the most important difference and reduce it by performing the following steps until success or failure:
   a. Select an operator that is applicable to the current difference. If there are no operators that can be applied, then return failure.
   b. Attempt to apply the operator to the current state by generating two temporary states, one where the operator's preconditions are true (pre-state), and one that would be the result if the operator were applied to the current state (post-state).
   c. Divide the problem into two parts, a *first*part, from the current-state to the pre-state, and a *last*part, from the post-state to the goal state. Call means-ends analysis to solve both pieces. If both are true, then return success, with the solution consisting of the *first*part, the selected operator, and the *last*part.