22 Problem Spaces and Search

Chapter Objectives

Uninformed state space search strategies are revisited:
- Depth-first search
- Breadth-first search

Heuristic or best-first search is presented

Java idioms are created for implementing state-space search
- Establishing a framework
  - Interface class
    - `Solver`
    - `AbstractSolver`
  - Implements search algorithms

Chapter Contents

22.1 Abstraction and Generality in Java
22.2 Search Algorithms
22.3 Abstracting Problem States
22.4 Traversing the Solution Space
22.5 Putting the Framework to Use

22.1 Abstraction and Generality in Java

This book, like the history of programming languages in general, is a story of increasingly powerful abstraction mechanisms. Because of the complexity of the problems it addresses and the rich body of theory that defines those problems in computational terms, AI has proven an ideal vehicle for developing languages and the abstraction mechanisms that give them much of their value. Lisp advanced ideas of procedural abstraction by following a mathematical model – recursive functions – rather than the explicitly machine influenced structures of earlier languages. The result was a sophisticated approach to variable binding, data structures, function definition and other ideas that made Lisp a powerful test bed for ideas about symbolic computing and programming language design. In particular, Lisp contributed to the development of object-oriented programming through a variety of Lisp-based object languages culminating in the Common Lisp Object System (CLOS). Prolog took a more radical approach, combining unification based pattern matching, automated theorem proving, and built-in search to render procedural mechanisms almost completely implicit and allow programmers to approach programs in a declarative, constraint-based manner.

This chapter continues exploring abstraction methods, taking as its focus the object-oriented idiom, and using the Java programming language as a vehicle. It builds on Java's object-orientated semantics, employing such mechanisms as inheritance and encapsulation. It also explores the interaction between AI theory and program architecture: the problems of

rendering theories of search and representation into code in ways that honor the needs of both. In doing so, it provides a foundation for the search engines used in expert systems, cognitive models, automated reasoning tools, and similar approaches that we present later. As we develop these ideas, we urge the reader to consider the ways in which AI theories of search and representation shaped programming techniques, and the abstractions that project that theory into working computer programs.

One of the themes we have developed throughout this book concerns generality and reuse. Through the development of object-oriented programming, many of these ideas have evolved into the notion of a framework: a collection of code that furnishes general, reusable components (typically data structures, algorithms, software tools, and abstractions) for building a specific type of software. Just as a set of modular building components simplifies house construction, frameworks simplify the computational implementation of applications.

Creating useful frameworks in Java builds on several important abstraction mechanisms and design patterns. Class inheritance is the most basic of these, allowing us to specify frameworks as general classes that are specified to an application. However, class inheritance is only the starting point for the subtler problems of implementing search algorithms. Additional forms of abstractions we use include interfaces and generic collections.

Our approach follows the development of search algorithms leading to the Lisp and Prolog search shells in Chapters 4 and 14 respectively, but translates it into the unique Java idiom. We encourage the reader to reflect on the differences between these approaches and their implications for programming. Before implementing a framework for basic search, we present a brief review of the theory of search.

22.2 Search Algorithms

Search is ubiquitous in computer science, particularly in Artificial Intelligence where it is the foundation of both theoretical models of problem solving, practical applications, and general tools and languages (including Prolog). This chapter uses the theory of search and Java idioms to develop a framework for implementing search strategies. We have explored the theory of Artificial Intelligence search elsewhere (Luger 2009, Chapters 3, 4, and 6), but will review the key theoretical ideas briefly.

Both the analysis of problem structure and the implementation of problem solving algorithms depend upon modeling the structure of a problem graphically: as a state-space. The elements defining a state-space are:

> **A formal representation of possible states of a problem solution**. We can think of these as all possible steps in a solution process, including both complete solutions and partial steps toward them. An example of a state might be a configuration of a puzzle like the Rubik's cube, an arrangement of tiles in the 16-puzzle, or a complex set of logical assertions in an expert system.

Operators for generating new states from a given state. In our puzzle example, these operators are legal moves of the puzzle. In more sophisticated applications, they can be logical inferences, steps in data interpretation, or heuristic rules for expert reasoning. These operators not only generate the next states in a problem solving process, but also define the arcs or links in a state-space graph.

Some way of recognizing a goal state.

A starting state of the problem, represented as the root of the graph.

Figure 22.1 shows a portion of the state-space for the 8-puzzle, an example we will develop later in this chapter.

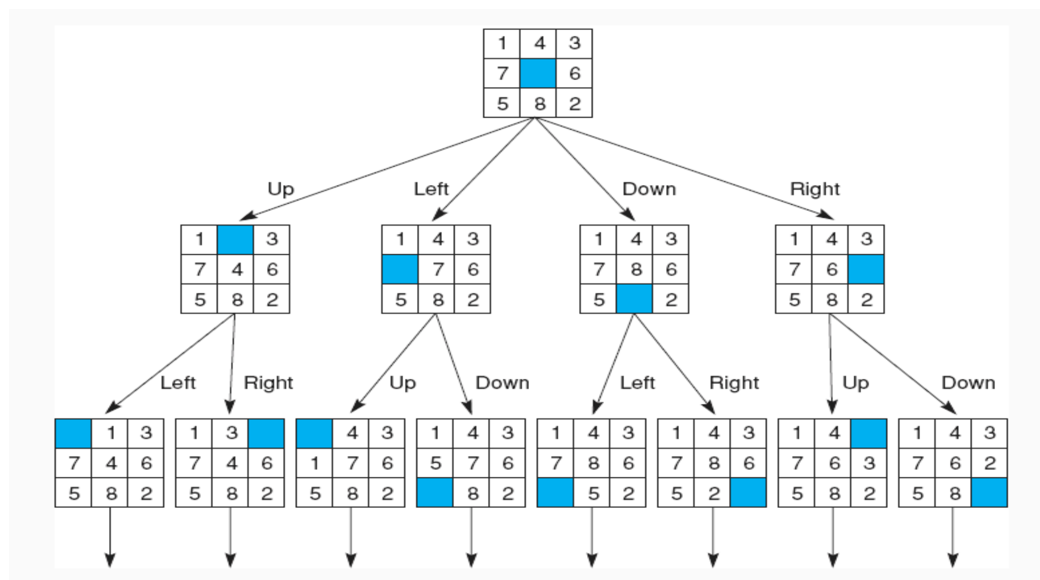

Figure 22.1. A sample state space to be searched. The goal is to have the numbers in clockwise order from the upper left hand corner. States are generated by "moving" the blank.

Our discussion begins with two basic "uninformed" algorithms: depth-first search (DFS) and breadth-first search (BFS). We call these "uninformed" because they do not use knowledge of the problem-space to guide search, but proceed in a fixed order. DFS picks a branch of the search space and follows it until it finds a goal state; if the branch ends without finding a goal, DFS "backs up" and tries another branch. In contrast, breadth-first search goes through the state space one layer at a time.

Figure 22.2 illustrates the difference between these approaches for the 8-puzzle problem, where depth-first search is given a five level depth bound. Note that although both searches find solutions, breadth-first search finds the solution that is closest to the root or start state. This is always true because BFS does not consider a solution path of distance `d` from the start until it has considered all solution paths of distance `d-1` from the start.

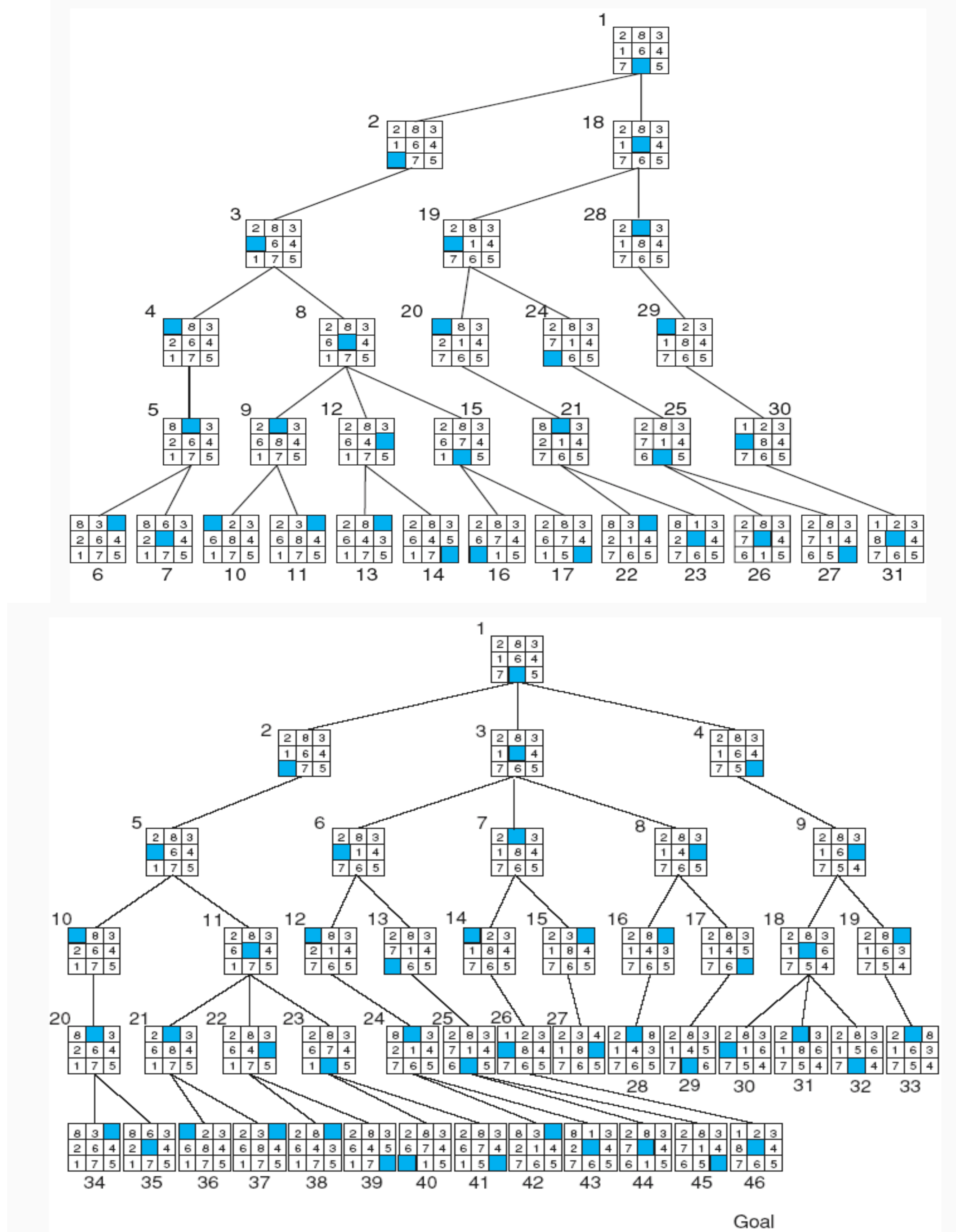

Figure 22.2. Depth-first search (a), using a depth bound of five levels, and breadth-first search (b) of the 8-puzzle. States are generated by "moving" the blank space.

These algorithms search a state-space in very different orders, but share a common general structure: both go through an iterative process of selecting a state in the graph, beginning with the start-state, quitting with success if the state is a goal, and generating its child states if it is not. What distinguishes them is how they handle the collection of child states. The following pseudo code gives a basic search implementation:

```
Search(start-state)
{
    place start-state on state-list;
    while (state-list is not empty)
    {
         state = next state on state-list;
         if(state is goal)
              finish with success;
         else
              generate all children of state and
              place those not previously visited
              on state-list;
    }
}
```

This algorithm is the general structure for all the implementations of search in this chapter: what distinguishes different search algorithms is how they manage unvisited states. DFS manages the `state-list` as a *stack*, while BFS uses a *queue*. The theoretical intuition here is that the last-in/first-out nature of a stack gives DFS its aggressive qualities of pursuing the current path. The first-in/first-out state management of a queue assures the algorithm will not move to a deeper level until all prior states have been visited. (It is worth noting that the Prolog interpreter maintains execution states on a stack, giving the language its depth-first character).

However, BFS and DFS proceed blindly through the space. The order of the states they visit depends only on the structure of the state space, and as a result DFS and/or BFS may not find a goal state efficiently (or at all). Most interesting problems involve multiple transitions from each problem state. Iterating on these multiple child states through the entire problem space often leads to an exponential growth in the number of states involved in the `state-list`. For small enough spaces, we can find a solution through uninformed search methods. Larger spaces, however, must be searched intelligently, through the use of heuristics. *Best-first search* is our third algorithm; it uses heuristics to evaluate the set of states on `state-list` in order to choose which of several possible branches in a state space to pursue next. Although there are general heuristics, such as means-ends analysis and inheritance (Luger 2009), the most powerful techniques exploit knowledge unique to the particular problem domain.

A general implementation of best-first or heuristic search computes a numerical estimate of each state's "distance" from a solution. Best-first search always tries the state that has the closest estimate of its distance to a goal. We can implement best-first search using the same general algorithm as the other state space searches presented above, however we must assign a heuristic rank to each state on the `state-list`, and maintain it as a sorted list, guaranteeing that the best ranked states will be evaluated first. `state-list` is thus handled as a priority queue.

This summary of search is not intended as a tutorial, but rather as a brief refresher of the background this chapter assumes. We refer readers who need further information on this theory to an AI text, such as Luger (2009, see Chapters 3 and 4).

21.3 Abstracting Problem States

We begin with the search pseudo-code defined above, focusing on the problem of representing states of the search space, as is typical of AI programming. This also reflects architectures we have used throughout the book: the separation of representation and control. Because of the relative simplicity of the search algorithms we implement, we approach this as separate implementations of states and search engines.

Our goal is to define an abstract representation of problem states that supports the general search algorithm and can be easily specialized through the mechanism of class inheritance. Our basic approach will be to define a class, called `State`, that specifies the methods common to all problem states and to define subclasses that add problem-specific information to it, as we see in Figure 22.3.

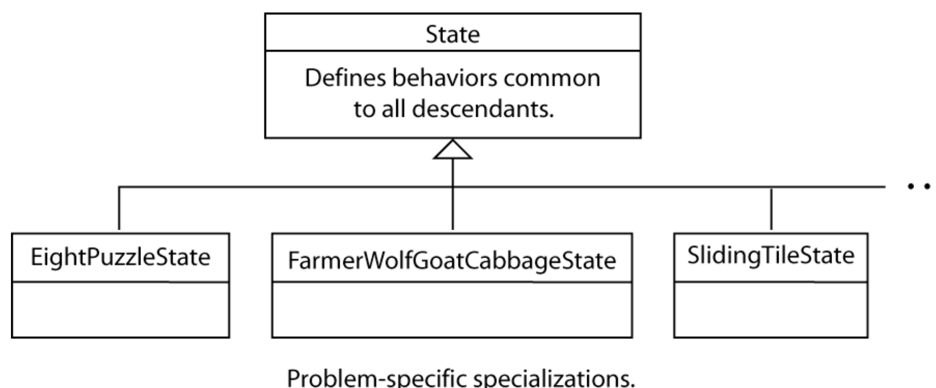

Figure 22.3. State representation for search containing problem-specific specifications.

In many cases, general class definitions like `State` will implement methods to be used (unless overridden) by descendant classes. However, this is only one aspect of inheritance; we may also define method names and the types of their arguments (called the *method signature*), without actually implementing them. This mechanism, implemented through abstract classes and methods, allows us to specify a sort of contract with future extensions to the class: we can define other methods in the framework to call these methods in instances of `State`'s descendants without knowing how they will implement them. For example, we know that all states must be able to produce next moves, determine if they are a solution, or calculate their heuristic value, even though the implementation of these is particular to specific problem states.

We can specify this as an abstract class:

```
public abstract class State
{
    public abstract Set<State> getPossibleMoves();
    public abstract boolean isSolution();
    public abstract double getHeuristic();
}
```

Note that both the methods and the class itself must be labeled `abstract`. The compiler will treat efforts to instantiate this class as errors. Abstract methods are an important concept in object-oriented programming, since the signature provides a full specification of the method's interface, allowing us to define code that uses these methods prior to their actual implementation.

Requiring the state class to inherit from the `State` base class raises a number of design issues. Java only allows inheritance from a single parent class, and the programmer may want to use the functionality of another parent class in creating the definition of the state. In addition, it seems wasteful to use a class definition if all we are defining are method signatures; generally, object-oriented programming regards classes as containing at least some state variables or implemented methods.

Java provides a mechanism, the *interface*, which addresses this problem. If all we need is a specification – essentially, a contract – for how to interact with instances of a class, we leave the implementation to future developers, we can define it as an `interface`. In addition to having a single parent class, a class can be declared to implement multiple `interface`s.

In this case, we will define `State` as an interface. An interface is like a class definition in which all methods are abstract: it carries no implementation, but is purely a contract for how the class must be implemented. In addition to the above methods, our `interface` will also define methods for getting the parent of the state, and its distance from the goal:

```
public interface State
{
    public Iterable<State> getPossibleMoves();
    public boolean isSolution();
    public double getHeuristic();
    public double getDistance();
    public State getParent();
}
```

Note in this situation the expression `Iterable<State>` returned by `getPossibleMoves()`. The expression, `Iterable<State>` is part of Java's *generics* capability, which was introduced to the language in Java 1.5. Generics use the notation `Collection-Type<Element-Type>` to specify a collection of elements of a specific type, in this case, an `Iterable` collection of objects of type `State`. In earlier versions of

Java, collections could contain any descendants of `Object`, the root of Java's class hierarchy. This prevented adequate type checking on collection elements, creating the potential for run-time type errors. Generics prevent this, allowing us to specify the type of collection elements.

Like `State`, `Iterable<State>` is an interface, rather than a class definition. `Iterable` defines an interface for a variety of classes that allow us to iterate over their members. This includes the `Set`, `List`, `Stack`, `PriorityQueue` and other collection classes. We could define this collection of child states using a specific data structure, such as a list, stack, etc. However, it is generally a bad idea to constrain later specialization of framework classes unnecessarily. Suppose the developer has good reason to collect child states in a different data structure? Once again, by using an interface to define a data type, we create a contract that will allow our search framework to implement functions that use classes, while leaving their instantiation to future programmers.

This interface is adequate to implement the search algorithms of Section 22.1, but before implementing the rest of our framework, note that two of the methods specified in the `State` interface are general enough to be defined for most, if not all, problems: `getDistance()` computes the distance from the start state, and `getParent()` returns the `parent` state in the search space. To simplify the work for future programmers, we implement a class, `AbstractState`, that provides a default implementation of these methods.

```
public abstract class AbstractState implements
         State
{
    private State parent = null;
    private double distance = 0;

    public AbstractState() {}
    public AbstractState(State parent)
    {
         this.parent = parent;
         this.distance = parent.getDistance() + 1;
    }
    public State getParent()
    {
         return parent;
    }
    public double getDistance()
    {
         return distance;
    }
}
```

Note that `AbstractState` implements the `State` interface, so classes that extend it can be used in our framework, freeing the programmer for the need to implement certain methods. It may seem that we have gone in a circle in this discussion, first defining `State` as an abstract class, then arguing that it should be an `Interface`, and now reintroducing an abstract class again. However, there are good reasons for this approach. Figure 22.4 illustrates a common Java design pattern, the use of an interface to specify an implementation "contract", with an abstract class providing default implementations of general methods.

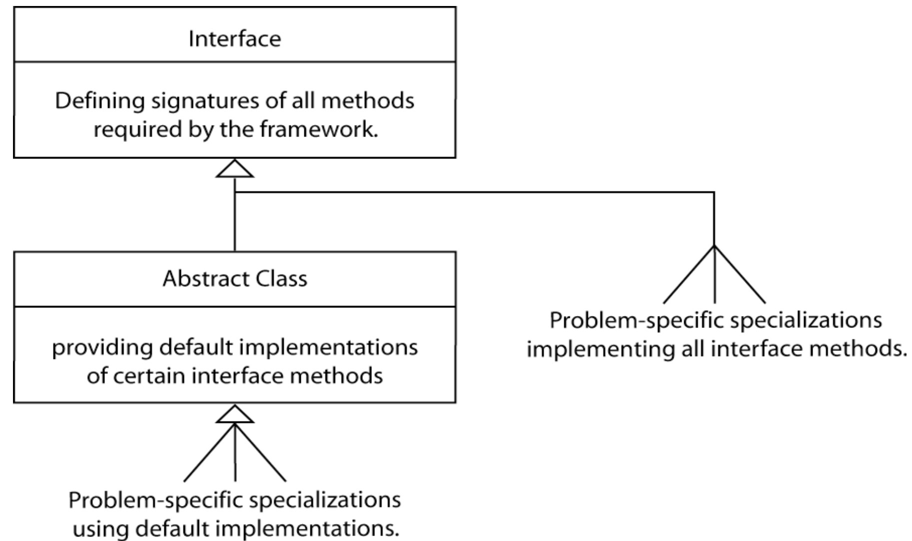

Figure 22.4. A Java design pattern: using an interface to specify a contract.

The pattern of specifying method signatures in an interface and providing default implementations of certain methods in an abstract class, is common in Java. By defining all methods required of the framework in an interface and using the interface to specify all types, we do not constrain future programmers in any way. They can bypass the abstract class entirely to address efficiency concerns, the needs of a problem that may not fit the default implementations, or simply to improve on the defaults. In many situations, however, programmers will use the abstract class implementation to simplify coding.

The next section repeats this pattern in implementing the control portion of our framework: the depth-first, breadth-first, and best-first search algorithms described earlier.

22.4 Traversing the Problem Space

Although simple, the `State` interface fully specifies the "contract" between the search framework and developers of problem-specific state representations. It gives the method signatures for testing if a state is a goal, and for generating child states. It leaves the specific representation to descendant classes. The next task is to address the implementation of search itself: defining the list of states and the mechanisms for moving

through them in search. As with `State`, we will begin with an `interface` definition:

```
public interface Solver
{
    public List<State> solve(State initialState);
}
```

Although simple, this captures a number of constraints on solvers. In addition to requiring an initial state as input, the solve method returns the list of visited states as a result. Once again, it defines the returned collection using a generic interface. A `List<E>` is a collection of ordered elements of type `E`. As with `Set<E>`, the list interface is supported by a variety of implementations.

Using the pattern of Figure 22.4, we will provide an abstract implementation of `Solver`. The code fragment below implements a general search algorithm that does not specify the management of open states:

```
private Set<State> closed = new HashSet<State>();
public List<State> solve(State initialState)
{                               //Reset closed and open lists
    closed.clear();
    clearOpen();
    addState(initialState);
    while (hasElements())
   {
         State s = nextState();
         if (s.isSolution())
              return findPath(s);
         closed.add(s);
         Iterable<State> moves =
              s.getPossibleMoves();
         for (State move : moves)
              if (!closed.contains(move))
                   addState(move);
    }
    return null;
}
```

In this method implementation, we maintain a closed list of visited states to detect loops in the search. We maintain `closed` as a `Set<State>` and implement it as a `HashSet<State>` for efficiency. We use the `Set<State> interface` since the closed list will contain no duplicates.

The `solve` method begins by clearing any states from the `closed` list, and adding the initial state to the `open` list using the `addState` method. We specify `addState` as an abstract method, along with the methods

hasElements() and nextState(). These methods allow us to add and remove states from the open list, and test if the list is empty. We specify them as abstract methods to hide the implementation of the open list, allowing the particular implementation to be defined by descendents of AbstractSolver.

The body of the method is a loop that:

Tests for remaining elements in the open list, using the abstract method hasElements();

Acquires the next state from the list using the abstract method nextState();

Tests to see if it is a solution and returns the list of visited states using the method findPath (to be defined);

Adds the state to the closed list; and

Generates child states, placing them on the open list using the abstract addState() method.

Perhaps the most significant departure from the Lisp and Prolog versions of the algorithm is the use of an iterative loop, rather than recursion to implement search. This is mainly a question of style. Like all modern languages, Java supports recursion, and it is safe to assume that the compiler will optimize tail recursion so it is as efficient as a loop. However, Java programs tend to favor iteration and we follow this style.

We have now implemented the general search algorithm. We complete the definition of the AbstractSolver class by defining the findPath method and specifying the abstract methods that its descendents must implement:

```
public abstract class AbstractSolver implements
         Solver
{
    private Set<State> closed =
          new HashSet<State>();
    public List<State> solve(State initialState)
    {                                    // As defined above
    }
    public int getVisitedStateCount()
     {
         return closed.size();
    }
    private List<State> findPath(State solution)
     {
         LinkedList<State> path =
                new LinkedList<State>();
         while (solution != null) {
              path.addFirst(solution);
              solution = solution.getParent();
```

```
        }
        return path;
    }
    protected abstract boolean hasElements();
    protected abstract State nextState();
    protected abstract void addState(State s);
    protected abstract void clearOpen();
}
```

Note once again how the abstract methods hide the implementation of the maintenance of `open` states. We can then implement the different search algorithms by creating subclasses of `AbstractSolver` and implementing these methods. Depth-first search implements them using a `stack` structure:

```
public class DepthFirstSolver extends AbstractSolver
{
    private Stack<State> stack = new Stack<State>();
    protected void addState(State s)
    {
        if (!stack.contains(s))
            stack.push(s);
    }
    protected boolean hasElements()
    {
        return !stack.empty();
    }
    protected State nextState()
    {
        return stack.pop();
    }
    protected void clearOpen()
    {
        stack.clear();
    }
}
```

Similarly, we can implement breadth-first search as a subclass of `AbstractSolver` that uses a `LinkedList` implementation:

```
public class BreadthFirstSolver extends
          AbstractSolver
{
    private Queue<State> queue =
          new LinkedList<State>();
    protected void addState(State s)
    {
```

```
            if (!queue.contains(s))
                queue.offer(s);
        }
        protected boolean hasElements()
        {
            return !queue.isEmpty();
        }
        protected State nextState()
        {
            return queue.remove();
        }
        protected void clearOpen()
        {
            queue.clear();
        }
    }
```

Finally, we can implement the best-first search algorithm by extending `AbstractSolver` and using a priority queue, PriorityQueue, to implement the `open` list:

```
    public class BestFirstSolver extends AbstractSolver
    {
        private PriorityQueue<State> queue = null;
        public BestFirstSolver()
        {
            queue = new PriorityQueue<State>(1,
                      new Comparator<State>()
                {
                public int compare(State s1, State s2)
                {
                //f(x) = distance + heuristic
                return Double.compare(
                s1.getDistance() + s1.getHeuristic(),
                s2.getDistance() + s2.getHeuristic());
                }
                });
        }
        protected void addState(State s)
        {
            if (!queue.contains(s))
                queue.offer((State)s);
        }
```

```
        protected boolean hasElements()
        {
            return !queue.isEmpty();
        }
        protected State nextState()
        {
            return queue.remove();
        }
        protected void clearOpen()
        {
            queue.clear();
        }
    }
```

In defining the `open` list as a `PriorityQueue`, this algorithm passes in a comparator for states that uses the heuristic evaluators defined in the `State` interface.

Note that both `BreadthFirstSolver` and `BestFirstSolver` define the open list using the interface `Queue<State>`, but instantiate them as `LinkedList<State>` and `PriorityQueue<State>` respectively. This suggests we could gain further code reuse by combining these definitions into a common superclass. This is left as an exercise.

22.5 Putting the Framework to Use

All that remains in order to apply these search algorithms is to define an appropriate state representation for a problem. As an example, we define a state representation for the *framer, wolf, goat and cabbage*, FWGC, problem as a subclass of `AbstractState`. (We have presented representations and generalized search strategies for the FWGC problem in both Prolog, Chapter 4 and Lisp, Chapter 13. These different language-specific approaches to the same problem can offer insight into the design patterns and idioms of each language.)

The first step in this implementation is representing problem states. A simple, and for this problem effective, way to do so is to define the location of each element of the problem. Following a common Java idiom, we will create a user-defined type for locations using the Java `enum` capability. This simplifies readability of the code. We will also create two constructors, one that creates a default starting state with everyone on the east bank, and a private constructor that can create arbitrary states to be used in generating moves:

```
    public class FarmerWolfGoatState extends
               AbstractState
    {
        enum Side
        {
            EAST { public Side getOpposite()
```

```
                 { return WEST; } },
        WEST { public Side getOpposite()
                 { return EAST; } };
        abstract public Side getOpposite();
    }
    private Side farmer = Side.EAST;
    private Side wolf = Side.EAST;
    private Side goat = Side.EAST;
    private Side cabbage = Side.EAST;
    /**
     * Constructs a new default state with everyone on the east side.
     */
    public FarmerWolfGoatState()
    {}
    /**
     * Constructs a move state from a parent state.
     */
    public FarmerWolfGoatState(
             FarmerWolfGoatState parent,
             Side farmer, Side wolf,
             Side goat, Side cabbage)
     {
        super(parent);
        this.farmer = farmer;
        this.wolf = wolf;
        this.goat = goat;
        this.cabbage = cabbage;
    }
}
```

Having settled on a representation, the next step is to define the abstract methods specified in `AbstractState`. We define `isSolution()` as a straightforward check for the goal state, i.e., if everyone is on the west bank:

```
public boolean isSolution()
{
        return farmer==Side.WEST &&
               wolf==Side.WEST &&
               goat==Side.WEST &&
               cabbage==Side.WEST;
}
```

The most complex method is `getPossibleMoves()`. To simplify this definition, we will use the `getOpposite()` method defined above, and `addIfSafe(. . .)` will add the state to the set of moves if it is legal:

```
private final void addIfSafe(Set<State> moves)
{
        boolean unsafe =
              (farmer != wolf && farmer != goat) ||
              (farmer != goat && farmer != cabbage);
        if (!unsafe)
            moves.add(this);
}
```

Although simple, these methods have some interest, particularly their use of the `final` specification. This indicates that the method will not be redefined in a subclass. They also indicate to the compiler that it can substitute the actual method code for a function call as a compiler optimization. We implement `getPossibleMoves`:

```
public Iterable<State> getPossibleMoves()
{
        Set<State> moves = new HashSet<State>();
        if (farmer==wolf)                          //Move wolf
            new FarmerWolfGoatState(
                  this,farmer.getOpposite(),
                  wolf.getOpposite(),
                  goat,
                  cabbage).addIfSafe(moves);
        if (farmer==goat)                          //Move goat
            new FarmerWolfGoatState(
                  this,farmer.getOpposite(),
                  wolf,
                  goat.getOpposite(),
                  cabbage).addIfSafe(moves);
        if (farmer==cabbage)                  //Move cabbage
            new FarmerWolfGoatState(
                  this,farmer.getOpposite(),
                  wolf,
                  goat,
                  cabbage.getOpposite()).
                              addIfSafe(moves);
        new FarmerWolfGoatState(          //Move just farmer
                  this,farmer.getOpposite(),
                  wolf,
                  goat,
                  cabbage).addIfSafe(moves);
        return moves;
}
```

Although we will leave implementation of getHeuristic() as an exercise, there are a few more details we must address. Among the methods defined in Object (the root of the Java hierarchy), are equals and hashCode. We must override the default definitions of these because two states should be considered equal if the four participants are at the same location, ignoring the move count and parent states that are also recorded in states. Simple definitions of these methods are:

```
public boolean equals(Object o)
{
        if (o==null ||
             !(o instanceof FarmerWolfGoatState))
        return false;
        FarmerWolfGoatState fwgs =
                  (FarmerWolfGoatState)o;
        return farmer  == fwgs.farmer &&
               wolf    == fwgs.wolf &&
               cabbage == fwgs.cabbage &&
               goat    == fwgs.goat;
 }
public int hashCode()
{
        return (farmer  == Side.EAST ? 1 : 0)+
               (wolf    == Side.EAST ? 2 : 0)+
               (cabbage == Side.EAST ? 4 : 0)+
               (goat    == Side.EAST ? 8 : 0);
}
```

This chapter examined a number of Java techniques and idioms. Perhaps the most important, however, is the use of interfaces and abstract classes to specify a contract for viable extensions to the basic search methods. This was essential to our approach to building reusable search methods, and will continue to be an important abstraction method throughout Part IV.

Exercises

1. Building on the code and design patterns suggested in Chapter 22, finish coding and run the complete solution of the Farmer, Wolf, Coat, and Cabbage problem. Implement depth-first, breadth-first, and best-first solutions.

2. At the end of section 22.4, we noted that the LinkedList<State> and PriorityQueue<State> used to manage the open list in BreadthFirstSolver and BestFirstSolver respectively both used the interface Queue<State>. This suggests the possibility of creating a superclass of both solvers to provide shared definitions of the open list functions. Rewrite the solver framework to include such a class. What are the advantages of doing so for the maintainability, understandability, and usefulness of the framework? The disadvantages?

3. The current solver stops when it finds the first solution. Extend it to include a `nextSolution()` method that continues the search until it finds a next solution, and a reset() method that resets the search to the beginning.

4. Use the Java framework of Section 22.5 to create depth-first, breadth-first, and best-first solutions for the Missionary and Cannibal problem.

> Three missionaries and three cannibals come to the bank of a river they wish to cross. There is a boat that will hold only two, and any of the group is able to row. If there are ever more missionaries than cannibals on any side of the river the cannibals will get converted. Devise a series of moves to get all the people across the river with no conversions.

5. Use and extend your code of problem 4 to check alternative forms of the missionary and cannibal problem—for example, when there are four missionaries and four cannibals and the boat holds only two. What if the boat can hold three? Try to generalize solutions for the whole class of missionary and cannibal problems.

6. Use the Java framework of Section 22.5 to create depth-first, breadth-first, and best-first solutions for the Water Jugs problem:

> There are two jugs, one holding 3 and the other 5 gallons of water. A number of things can be done with the jugs: they can be filled, emptied, and dumped one into the other either until the poured-into jug is full or until the poured-out-of jug is empty. Devise a sequence of actions that will produce 4 gallons of water in the larger jug. (Hint: use only integers.)