Practical Artificial Intelligence Programming With Java

Third Edition

Mark Watson

November 11, 2008

downloaded from: lib.ommolketab.ir

Contents

List of Figures

List of Tables

Preface

I wrote this book for both professional programmers and home hobbyists who already know how to program in Java and who want to learn practical Artificial Intelligence (AI) programming and information processing techniques. I have tried to make this an enjoyable book to work through. In the style of a "cook book," the chapters can be studied in any order. Each chapter follows the same pattern: a motivation for learning a technique, some theory for the technique, and a Java example program that you can experiment with.

I have been interested in AI since reading Bertram Raphael's excellent book *Thinking Computer: Mind Inside Matter* in the early 1980s. I have also had the good fortune to work on many interesting AI projects including the development of commercial expert system tools for the Xerox LISP machines and the Apple Macintosh, development of commercial neural network tools, application of natural language and expert systems technology, medical information systems, application of AI technologies to Nintendo and PC video games, and the application of AI technologies to the financial markets.

I enjoy AI programming, and hopefully this enthusiasm will also infect the reader.

Software Licenses for example programs in this book

My example programs for chapters using Open Source Libraries are released under the same licenses as the libraries:

- Drools Expert System Demos: Apache style license
- PowerLoom Reasoning: LGPL
- Sesame Semantic Web: LGPL

The licenses for the rest of my example programs are in the directory licenses-for-book-code:

- License for commercial use: if you purchase a print version of this book or the for-fee PDF version from Lulu.com then you can use any of my code and data used in the book examples under a non-restrictive license. This book can be purchaed at http://www.lulu.com/content/4502573
- Free for non-commercial and academic use: if you use the free PDF version

of this book you can use the code and data used in the book examples free for activities that do not generate revenue.

Acknowledgements

I would like to thank Kevin Knight for writing a flexible framework for game search algorithms in *Common LISP* (Rich, Knight 1991) and for giving me permission to reuse his framework, rewritten in Java for some of the examples in Chapter 2. I have a library full of books on AI and I would like to thank the authors of all of these books for their influence on my professional life. I frequently reference books in the text that have been especially useful to me and that I recommend to my readers.

In particular, I would like to thank the authors of the following two books that have had the most influence on me:

- Stuart Russell and Peter Norvig's *Artificial Intelligence: A Modern Approach* which I consider to be the best single reference book for AI theory
- John Sowa's book *Knowledge Representation* is a resource that I frequently turn to for a holistic treatment of logic, philosophy, and knowledge representation in general

Book Editor:

Carol Watson

Thanks to the following people who found typos:

Carol Watson, James Fysh, Joshua Cranmer, Jack Marsh, Jeremy Burt, Jean-Marc Vanel

1 Introduction

There are many fine books on Artificial Intelligence (AI) and good tutorials and software on the web. This book is intended for professional programmers who either already have an interest in AI or need to use specific AI technologies at work.

The material is not intended as a complete reference for AI theory. Instead, I provide enough theoretical background to understand the example programs and to provide a launching point if you want or need to delve deeper into any of the topics covered.

1.1 Other JVM Languages

The Java language and JVM platform are very widely used so that techniques that you learn can be broadly useful. There are other JVM languages like JRuby, Clojure, Jython, and Scala that can use existing Java classes. While the examples in this book are written in Java you should have little trouble using my Java example classes and the open source libraries with these alternative JVM languages.

1.2 Why is a PDF Version of this Book Available Free on the Web?

I have written 14 books that have been published by the traditional publishers Springer-Verlag, McGraw-Hill, J. Wiley, Morgan Kaufman, Hungry Minds, MCP, and Sybex. This is my first book that I have produced and published on my own and my motivation for this change is the ability to write for smaller niche markets on topics that most interest me.

As an author I want to both earn a living writing and have many people read and enjoy my books. By offering for sale both a print version and a for-fee PDF version for purchase at http://www.lulu.com/content/4502573 I can earn some money for my efforts and also allow readers who can not afford to buy many books or may only be interested in a few chapters of this book to read the free PDF version that is available from my web site.

Please note that I do not give permission to post the free PDF version of this book on other people's web sites: I consider this to be commercial exploitation in violation of the Creative Commons License that I have chosen for this book. Having my free web books only available on my web site brings viewers to my site and helps attract customers for my consulting business. I do encourage you to copy the PDF for this book onto your own computer for local reading and it is fine to email copies of the free PDF to friends.

If you enjoy reading the no-cost PDF version of this book I would also appreciate it if you would purchase a print copy using the purchase link:

http://www.lulu.com/content/4502573

I thank you for your support.

1.3 Book Software

You can download a large ZIP file containing all code and test data used in this book from the URL:

```
http://markwatson.com/opencontent/javaai_3rd_code.zip
```

All the example code that I have written is covered by the licenses discussed in the Preface.

The code examples usually consist of reusable (non GUI) libraries and throwaway text-based test programs to solve a specific application problem; in some cases, the test code will contain a test or demonstration GUI.

1.4 Use of Java Generics and Native Types

In general I usually use Java generics and the new collection classes for almost all of my Java programming. That is also the case for the examples in this book except when using native types and arrays provides a real performance advantage (for example, in the search examples).

Since arrays must contain reifiable types they play poorly with generics so I prefer not to mix coding styles in the same code base. There are some obvious cases where not using primitive types leads to excessive object creation and boxing/unboxing. That said, I expect Java compilers, Hotspot, and the JVM in general to keep getting better and this may be a non-issue in the future.

1.5 Notes on Java Coding Styles Used in this Book

Many of the example programs do not strictly follow common Java programming idioms – this is usually done for brevity. For example, when a short example is all in one Java package I will save lines of code and programing listing space by not declaring class data private with public getters and setters; instead, I will sometimes simply use package visibility as in this example:

```
public static class Problem {
  // constants for appliance types:
  enum Appliance {REFRIGERATOR, MICROWAVE, TV, DVD};
  // constants for problem types:
  enum ProblemType {NOT_RUNNING, SMOKING, ON_FIRE,
                    MAKES_NOISE};
  // constants for environmental data:
  enum EnvironmentalDescription {CIRCUIT_BREAKER_OFF,
                                 LIGHTS_OFF_IN_ROOM};
  Appliance applianceType;
  List<ProblemType> problemTypes =
            new ArrayList<ProblemType>();
  List<EnvironmentalDescription> environmentalData =
            new ArrayList<EnvironmentalDescription>();
  // etc.
}
```

Please understand that I do not advocate this style of programming in large projects but one challenge in writing about software development is the requirement to make the examples short and easily read and understood. Many of the examples started as large code bases for my own projects that I "whittled down" to a small size to show one or two specific techniques. Forgoing the use of "getters and setters" in many of the examples is just another way to shorten the examples.

Authors of programming books are faced with a problem in formatting program snippets: limited page width. You will frequently see what would be a single line in a Java source file split over two or three lines to accommodate limited page width as seen in this example:

```
private static void
     createTestFacts(WorkingMemory workingMemory)
                                   throws Exception {
  ...
}
```

1.6 Book Summary

Chapter 1 is the introduction for this book.

Chapter 2 deals with heuristic search in two domains: two-dimensional grids (for example mazes) and graphs (defined by nodes and edges connecting nodes).

Chapter 3 covers logic, knowledge representation, and reasoning using the PowerLoom system.

Chapter 4 covers the Semantic Web. You will learn how to use RDF and RDFS data for knowledge representation and how to use the popular Sesame open source Semantic Web system.

Chapter 5 introduces you to rule-based or production systems. We will use the open source Drools system to implement simple expert systems for solving "blocks world" problems and to simulate a help desk system.

Chapter 6 gives an overview of Genetic Algorithms, provides a Java library, and solves a test problem. The chapter ends with suggestions for projects you might want to try.

Chapter 7 introduces Hopfield and Back Propagation Neural Networks. In addition to Java libraries you can use in your own projects, we will use two Swing-based Java applications to visualize how neural networks are trained.

Chapter 8 introduces you to the GPLed Weka project. Weka is a best of breed toolkit for solving a wide range of machine learning problems.

Chapter 9 covers several Statistical Natural Language Processing (NLP) techniques that I often use in my own work: processing text (tokenizing, stemming, and determining part of speech), named entity extraction from text, using the WordNet lexical database, automatically assigning tags to text, text clustering, three different approaches to spelling correction, and a short tutorial on Markov Models.

Chapter 10 provides useful techniques for gathering and using information: using the Open Calais web services for extracting semantic information from text, information discovery in relational databases, and three different approaches to indexing and searching text.

2 Search

Early AI research emphasized the optimization of search algorithms. This approach made a lot of sense because many AI tasks can be solved effectively by defining state spaces and using search algorithms to define and explore search trees in this state space. Search programs were frequently made tractable by using heuristics to limit areas of search in these search trees. This use of heuristics converts intractable problems to solvable problems by compromising the quality of solutions; this trade off of less computational complexity for less than optimal solutions has become a standard design pattern for AI programming. We will see in this chapter that we trade off memory for faster computation time and better results; often, by storing extra data we can make search time faster, and make future searches in the same search space even more efficient.

What are the limitations of search? Early on, search applied to problems like checkers and chess misled early researchers into underestimating the extreme difficulty of writing software that performs tasks in domains that require general world knowledge or deal with complex and changing environments. These types of problems usually require the understanding and then the implementation of domain specific knowledge.

In this chapter, we will use three search problem domains for studying search algorithms: path finding in a maze, path finding in a graph, and alpha-beta search in the games tic-tac-toe and chess.

2.1 Representation of Search State Space and Search Operators

We will use a single search tree representation in graph search and maze search examples in this chapter. Search trees consist of nodes that define locations in state space and links to other nodes. For some small problems, the search tree can be easily specified statically; for example, when performing search in game mazes, we can compute and save a search tree for the entire state space of the maze. For many problems, it is impossible to completely enumerate a search tree for a state space so we must define successor node search operators that for a given node produce all nodes that can be reached from the current node in one step; for example, in the

game of chess we can not possibly enumerate the search tree for all possible games of chess, so we define a successor node search operator that given a board position (represented by a node in the search tree) calculates all possible moves for either the white or black pieces. The possible chess moves are calculated by a successor node search operator and are represented by newly calculated nodes that are linked to the previous node. Note that even when it is simple to fully enumerate a search tree, as in the game maze example, we still might want to generate the search tree dynamically as we will do in this chapter).

For calculating a search tree we use a graph. We will represent graphs as node with links between some of the nodes. For solving puzzles and for game related search, we will represent positions in the search space with Java objects called nodes. Nodes contain arrays of references to both child and parent nodes. A search space using this node representation can be viewed as a **directed graph** or a **tree**. The node that has no parent nodes is the root node and all nodes that have no child nodes a called leaf nodes.

Search operators are used to move from one point in the search space to another. We deal with quantized search spaces in this chapter, but search spaces can also be continuous in some applications. Often search spaces are either very large or are infinite. In these cases, we implicitly define a search space using some algorithm for extending the space from our reference position in the space. Figure 2.1 shows representations of search space as both connected nodes in a graph and as a two-dimensional grid with arrows indicating possible movement from a reference point denoted by **R**.

When we specify a search space as a two-dimensional array, search operators will move the point of reference in the search space from a specific grid location to an adjoining grid location. For some applications, search operators are limited to moving up/down/left/right and in other applications operators can additionally move the reference location diagonally.

When we specify a search space using node representation, search operators can move the reference point down to any child node or up to the parent node. For search spaces that are represented implicitly, search operators are also responsible for determining legal child nodes, if any, from the reference point.

Note that I use different libraries for the maze and graph search examples.

2.2 Finding Paths in Mazes

The example program used in this section is MazeSearch.java in the directory src/search/maze and I assume that the reader has downloaded the entire example ZIP file for this book and placed the source files for the examples in a convenient place.

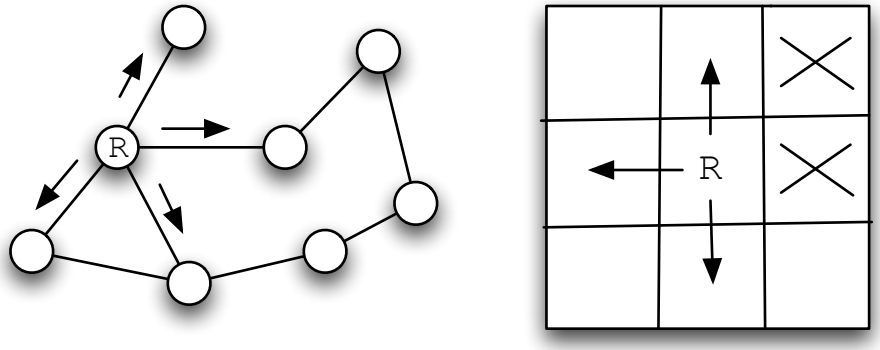

Figure 2.1: A directed graph representation is shown on the left and a two-dimensional grid (or maze) representation is shown on the right. In both representations, the letter R is used to represent the current position (or reference point) and the arrowheads indicate legal moves generated by a search operator. In the maze representation, the two grid cells marked with an X indicate that a search operator cannot generate this grid location.

Figure 2.2 shows the UML class diagram for the maze search classes: depth first and breadth first search. The abstract base class $AbstractSearchEngine$ contains common code and data that is required by both the classes $DepthFirstSearch$ and $BreadthFirstSearch$. The class $Maze$ is used to record the data for a two-dimensional maze, including which grid locations contain walls or obstacles. The class $Maze$ defines three static short integer values used to indicate obstacles, the starting location, and the ending location.

The Java class $Maze$ defines the search space. This class allocates a two-dimensional array of short integers to represent the state of any grid location in the maze. Whenever we need to store a pair of integers, we will use an instance of the standard Java class $java.awt.Dimension$, which has two integer data components: width and height. Whenever we need to store an x-y grid location, we create a new Dimension object (if required), and store the x coordinate in Dimension.width and the y coordinate in Dimension.height. As in the right-hand side of Figure 2.1, the operator for moving through the search space from given x-y coordinates allows a transition to any adjacent grid location that is empty. The Maze class also contains the x-y location for the starting location (startLoc) and goal location (goalLoc). Note that for these examples, the class Maze sets the starting location to grid coordinates 0-0 (upper left corner of the maze in the figures to follow) and the goal node in (width - 1)-(height - 1) (lower right corner in the following figures).

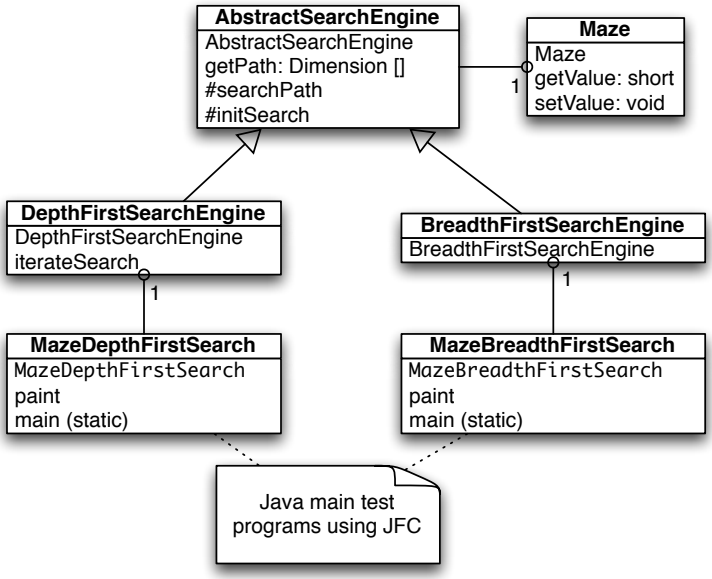

Figure 2.2: UML class diagram for the maze search Java classes

The abstract class $AbstractSearchEngine$ is the base class for both the depth first (uses a stack to store moves) search class $DepthFirstSearchEngine$ and the breadth first (uses a queue to store moves) search class $BreadthFirstSearchEngine$. We will start by looking at the common data and behavior defined in $AbstractSearchEngine$. The class constructor has two required arguments: the width and height of the maze, measured in grid cells. The constructor defines an instance of the $Maze$ class of the desired size and then calls the utility method initSearch to allocate an array $searchPath$ of $Dimension$ objects, which will be used to record the path traversed through the maze. The abstract base class also defines other utility methods:

- $equals(Dimensiond1, Dimensiond2)$ – checks to see if two arguments of type $Dimension$ are the same.
- $getPossibleMoves(Dimensionlocation)$ – returns an array of $Dimension$ objects that can be moved to from the specified location. This implements the movement operator.

Now, we will look at the depth first search procedure. The constructor for the derived class $DepthFirstSearchEngine$ calls the base class constructor and then solves the search problem by calling the method $iterateSearch$. We will look at this method in some detail. The arguments to $iterateSearch$ specify the current location and the current search depth:

```
    private void iterateSearch(Dimension loc, int depth)
```

The class variable $isSearching$ is used to halt search, avoiding more solutions, once one path to the goal is found.

```
        if (isSearching == false) return;
```

We set the maze value to the depth for display purposes only:

```
        maze.setValue(loc.width, loc.height, (short)depth);
```

Here, we use the super class $getPossibleMoves$ method to get an array of possible neighboring squares that we could move to; we then loop over the four possible moves (a null value in the array indicates an illegal move):

```
Dimension [] moves = getPossibleMoves(loc);
for (int i=0; i<4; i++) {
if (moves[i] == null) break; // out of possible moves
                             // from this location
```

Record the next move in the search path array and check to see if we are done:

```
  searchPath[depth] = moves[i];
  if (equals(moves[i], goalLoc)) {
     System.out.println("Found the goal at " +
                        moves[i].width +
                        ``, " + moves[i].height);
     isSearching = false;
     maxDepth = depth;
     return;
  } else {
```

If the next possible move is not the goal move, we recursively call the iterateSearch method again, but starting from this new location and increasing the depth counter by one:

```
    iterateSearch(moves[i], depth + 1);
    if (isSearching == false) return;
  }
```

S	2								
4	3								
5									
6	7	8	9	10	11				
					12				
18	17	16	15	14	13				
19	20								
22	21		27	28	29	30	31		
23	24	25	26				32	33	34
									G

Figure 2.3: Using depth first search to find a path in a maze finds a non-optimal solution

Figure 2.3 shows how poor a path a depth first search can find between the start and goal locations in the maze. The maze is a 10-by-10 grid. The letter S marks the starting location in the upper left corner and the goal position is marked with a G in the lower right corner of the grid. Blocked grid cells are painted light gray. The basic problem with the depth first search is that the search engine will often start searching in a bad direction, but still find a path eventually, even given a poor start. The advantage of a depth first search over a breadth first search is that the depth first search requires much less memory. We will see that possible moves for depth first search are stored on a stack (last in, first out data structure) and possible moves for a breadth first search are stored in a queue (first in, first out data structure).

The derived class $BreadthFirstSearch$ is similar to the $DepthFirstSearch$ procedure with one major difference: from a specified search location we calculate all possible moves, and make one possible trial move at a time. We use a queue data structure for storing possible moves, placing possible moves on the back of the queue as they are calculated, and pulling test moves from the front of the queue. The

effect of a breadth first search is that it "fans out" uniformly from the starting node until the goal node is found.

The class constructor for $BreadthFirstSearch$ calls the super class constructor to initialize the maze, and then uses the auxiliary method $doSearchOn2Dgrid$ for performing a breadth first search for the goal. We will look at the class $BreadthFirstSearch$ in some detail. Breadth first search uses a queue instead of a stack (depth first search) to store possible moves. The utility class $DimensionQueue$ implements a standard queue data structure that handles instances of the class $Dimension$.

The method $doSearchOn2Dgrid$ is not recursive, it uses a loop to add new search positions to the end of an instance of class $DimensionQueue$ and to remove and test new locations from the front of the queue. The two-dimensional array $allReadyVisited$ keeps us from searching the same location twice. To calculate the shortest path after the goal is found, we use the predecessor array:

```
private void doSearchOn2DGrid() {
  int width = maze.getWidth();
  int height = maze.getHeight();
  boolean alReadyVisitedFlag[][] =
          new boolean[width][height];
  Dimension predecessor[][] =
        new Dimension[width][height];
  DimensionQueue queue =
                new DimensionQueue();
  for (int i=0; i<width; i++) {
    for (int j=0; j<height; j++) {
      alReadyVisitedFlag[i][j] = false;
      predecessor[i][j] = null;
    }
  }
```

We start the search by setting the already visited flag for the starting location to true value and adding the starting location to the back of the queue:

```
  alReadyVisitedFlag[startLoc.width][startLoc.height]
     = true;
  queue.addToBackOfQueue(startLoc);
  boolean success = false;
```

This outer loop runs until either the queue is empty or the goal is found:

```
outer:
  while (queue.isEmpty() == false) {
```

We peek at the $Dimension$ object at the front of the queue (but do not remove it) and get the adjacent locations to the current position in the maze:

```
Dimension head = queue.peekAtFrontOfQueue();
Dimension [] connected =
                 getPossibleMoves(head);
```

We loop over each possible move; if the possible move is valid (i.e., not null) and if we have not already visited the possible move location, then we add the possible move to the back of the queue and set the predecessor array for the new location to the last square visited (head is the value from the front of the queue). If we find the goal, break out of the loop:

```
for (int i=0; i<4; i++) {
  if (connected[i] == null) break;
  int w = connected[i].width;
  int h = connected[i].height;
  if (alReadyVisitedFlag[w][h] == false) {
    alReadyVisitedFlag[w][h] = true;
    predecessor[w][h] = head;
    queue.addToBackOfQueue(connected[i]);
    if (equals(connected[i], goalLoc)) {
      success = true;
      break outer; // we are done
    }
  }
}
```

We have processed the location at the front of the queue (in the variable head), so remove it:

```
  queue.removeFromFrontOfQueue();
}
```

Now that we are out of the main loop, we need to use the predecessor array to get the shortest path. Note that we fill in the $searchPath$ array in reverse order, starting with the goal location:

```
maxDepth = 0;
if (success) {
  searchPath[maxDepth++] = goalLoc;
```

```
        for (int i=0; i<100; i++) {
          searchPath[maxDepth] =
             predecessor[searchPath[maxDepth - 1].
               width][searchPath[maxDepth - 1].
               height];
          maxDepth++;
          if (equals(searchPath[maxDepth - 1],
                     startLoc))
            break;  // back to starting node
        }
      }
    }
```

Figure 2.4 shows a good path solution between starting and goal nodes. Starting from the initial position, the breadth first search engine adds all possible moves to the back of a queue data structure. For each possible move added to this queue in one search cycle, all possible moves are added to the queue for each new move recorded. Visually, think of possible moves added to the queue as "fanning out" like a wave from the starting location. The breadth first search engine stops when this "wave" reaches the goal location. In general, I prefer breadth first search techniques to depth first search techniques when memory storage for the queue used in the search process is not an issue. In general, the memory requirements for performing depth first search is much less than breadth first search.

To run the two example programs from this section, change directory to src/search/-maze and type:

```
javac *.java
java MazeDepthFirstSearch
java MazeBreadthFirstSearch
```

Note that the classes $MazeDepthFirstSearch$ and $MazeBreadthFirstSearch$ are simple Java JFC applications that produced Figures 2.3 and 2.4. The interested reader can read through the source code for the GUI test programs, but we will only cover the core AI code in this book. If you are interested in the GUI test programs and you are not familiar with the Java JFC (or Swing) classes, there are several good tutorials on JFC programming at java.sun.com.

2.3 Finding Paths in Graphs

In the last section, we used both depth first and breadth first search techniques to find a path between a starting location and a goal location in a maze. Another common

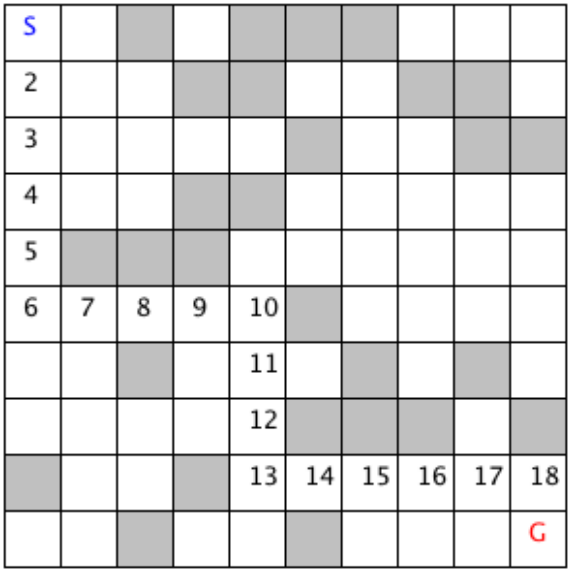

Figure 2.4: Using breadth first search in a maze to find an optimal solution

type of search space is represented by a graph. A graph is a set of nodes and links. We characterize nodes as containing the following data:

- A name and/or other data
- Zero or more links to other nodes
- A position in space (this is optional, usually for display or visualization purposes)

Links between nodes are often called edges. The algorithms used for finding paths in graphs are very similar to finding paths in a two-dimensional maze. The primary difference is the operators that allow us to move from one node to another. In the last section we saw that in a maze, an agent can move from one grid space to another if the target space is empty. For graph search, a movement operator allows movement to another node if there is a link to the target node.

Figure 2.5 shows the UML class diagram for the graph search Java classes that we will use in this section. The abstract class $AbstractGraphSearch$ class is the base class for both $DepthFirstSearch$ and $BreadthFirstSearch$. The classes $GraphDepthFirstSearch$ and $GraphBreadthFirstSearch$ and test programs also provide a Java Foundation Class (JFC) or Swing based user interface. These two test programs produced Figures 2.6 and 2.7.

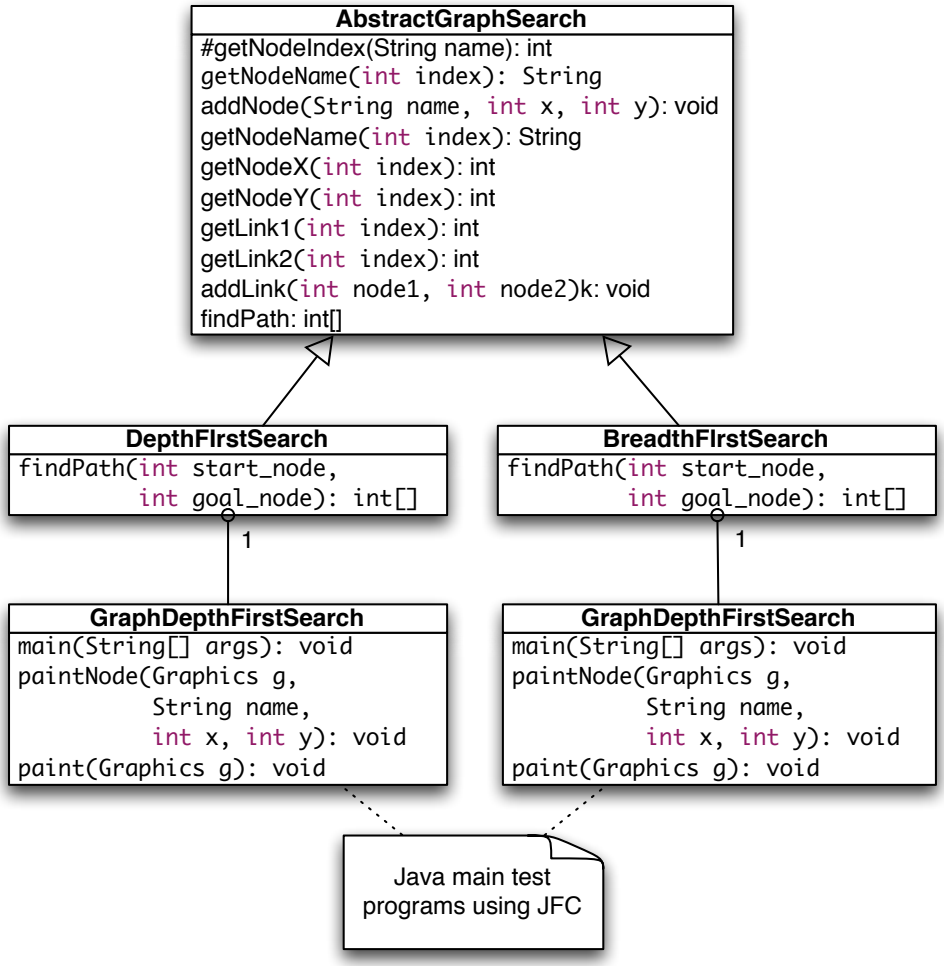

Figure 2.5: UML class diagram for the graph search classes

As seen in Figure 2.5, most of the data for the search operations (i.e., nodes, links, etc.) is defined in the abstract class $AbstractGraphSearch$. This abstract class is customized through inheritance to use a stack for storing possible moves (i.e., the array path) for depth first search and a queue for breadth first search.

The abstract class $AbstractGraphSearch$ allocates data required by both derived classes:

```
final public static int MAX = 50;
protected int [] path =
      new int[AbstractGraphSearch.MAX];
protected int num_path = 0;
// for nodes:
protected String [] nodeNames =
                      new String[MAX];
protected int [] node_x = new int[MAX];
protected int [] node_y = new int[MAX];
// for links between nodes:
protected int [] link_1 = new int[MAX];
protected int [] link_2 = new int[MAX];
protected int [] lengths = new int[MAX];
protected int numNodes = 0;
protected int numLinks = 0;
protected int goalNodeIndex = -1,
              startNodeIndex = -1;
```

The abstract base class also provides several common utility methods:

- addNode(String name, int x, int y) – adds a new node
- addLink(int n1, int n2) – adds a bidirectional link between nodes indexed by n1 and n2. Node indexes start at zero and are in the order of calling addNode.
- addLink(String n1, String n2) – adds a bidirectional link between nodes specified by their names
- getNumNodes() – returns the number of nodes
- getNumLinks() – returns the number of links
- getNodeName(int index) – returns a node's name
- getNodeX(), getNodeY() – return the coordinates of a node
- getNodeIndex(String name) – gets the index of a node, given its name

The abstract base class defines an abstract method $findPath$ that must be overridden. We will start with the derived class $DepthFirstSearch$, looking at its implementation of findPath. The $findPath$ method returns an array of node indices indicating the calculated path:

```
public int [] findPath(int start_node,
                       int goal_node) {
```

The class variable path is an array that is used for temporary storage; we set the first element to the starting node index, and call the utility method $findPathHelper$:

```
  path[0] = start_node; // the starting node
  return findPathHelper(path, 1, goal_node);
}
```

The method findPathHelper is the interesting method in this class that actually performs the depth first search; we will look at it in some detail:

The path array is used as a stack to keep track of which nodes are being visited during the search. The argument num_path is the number of locations in the path, which is also the search depth:

```
public int [] findPathHelper(int [] path,
                             int num_path,
                             int goal_node) {
```

First, re-check to see if we have reached the goal node; if we have, make a new array of the current size and copy the path into it. This new array is returned as the value of the method:

```
  if (goal_node == path[num_path - 1]) {
    int [] ret = new int[num_path];
    for (int i=0; i<num_path; i++) {
       ret[i] = path[i];
    }
    return ret;  // we are done!
  }
```

We have not found the goal node, so call the method $connected\_nodes$ to find all nodes connected to the current node that are not already on the search path (see the source code for the implementation of $connected\_nodes$):

```
int [] new_nodes = connected_nodes(path,
                                   num_path);
```

If there are still connected nodes to search, add the next possible "node to visit" to the top of the stack (variable $path$ in the program) and recursively call the method $findPathHelper$ again:

```
if (new_nodes != null) {
  for (int j=0; j<new_nodes.length; j++) {
    path[num_path] = new_nodes[j];
    int [] test = findPathHelper(new_path,
                                 num_path + 1,
                                 goal_node);
    if (test != null) {
      if (test[test.length-1] == goal_node) {
          return test;
      }
    }
  }
}
```

If we have not found the goal node, return null, instead of an array of node indices:

```
  return null;
}
```

Derived class $BreadthFirstSearch$ also must define abstract method $findPath$. This method is very similar to the breadth first search method used for finding a path in a maze: a queue is used to store possible moves. For a maze, we used a queue class that stored instances of the class Dimension, so for this problem, the queue only needs to store integer node indices. The return value of $findPath$ is an array of node indices that make up the path from the starting node to the goal.

```
public int [] findPath(int start_node,
                       int goal_node) {
```

We start by setting up a flag array $alreadyVisited$ to prevent visiting the same node twice, and allocating a predecessors array that we will use to find the shortest path once the goal is reached:

```
  // data structures for depth first search:
```

```
boolean [] alreadyVisitedFlag =
                        new boolean[numNodes];
int [] predecessor = new int[numNodes];
```

The class $IntQueue$ is a private class defined in the file BreadthFirstSearch.java; it implements a standard queue:

```
IntQueue queue = new IntQueue(numNodes + 2);
```

Before the main loop, we need to initialize the already visited and predecessor arrays, set the visited flag for the starting node to true, and add the starting node index to the back of the queue:

```
for (int i=0; i<numNodes; i++) {
  alreadyVisitedFlag[i] = false;
  predecessor[i] = -1;
}
alreadyVisitedFlag[start_node] = true;
queue.addToBackOfQueue(start_node);
```

The main loop runs until we find the goal node or the search queue is empty:

```
outer: while (queue.isEmpty() == false) {
```

We will read (without removing) the node index at the front of the queue and calculate the nodes that are connected to the current node (but not already on the visited list) using the $connected\_nodes$ method (the interested reader can see the implementation in the source code for this class):

```
  int head = queue.peekAtFrontOfQueue();
  int [] connected = connected_nodes(head);
  if (connected != null) {
```

If each node connected by a link to the current node has not already been visited, set the predecessor array and add the new node index to the back of the search queue; we stop if the goal is found:

```
  for (int i=0; i<connected.length; i++) {
     if (alreadyVisitedFlag[connected[i]] == false) {
       predecessor[connected[i]] = head;
```

```
            queue.addToBackOfQueue(connected[i]);
            if (connected[i] == goal_node) break outer;
          }
        }
        alreadyVisitedFlag[head] = true;
        queue.removeFromQueue(); // ignore return value
      }
    }
```

Now that the goal node has been found, we can build a new array of returned node indices for the calculated path using the predecessor array:

```
    int [] ret = new int[numNodes + 1];
    int count = 0;
    ret[count++] = goal_node;
    for (int i=0; i<numNodes; i++) {
      ret[count] = predecessor[ret[count - 1]];
      count++;
      if (ret[count - 1] == start_node)  break;
    }
    int [] ret2 = new int[count];
    for (int i=0; i<count; i++) {
      ret2[i] = ret[count - 1 - i];
    }
    return ret2;
  }
```

In order to run both the depth first and breadth first graph search examples, change directory to src-search-maze and type the following commands:

```
javac *.java
java GraphDepthFirstSearch
java GraphBeadthFirstSearch
```

Figure 2.6 shows the results of finding a route from node 1 to node 9 in the small test graph. Like the depth first results seen in the maze search, this path is not optimal.

Figure 2.7 shows an optimal path found using a breadth first search. As we saw in the maze search example, we find optimal solutions using breadth first search at the cost of extra memory required for the breadth first search.

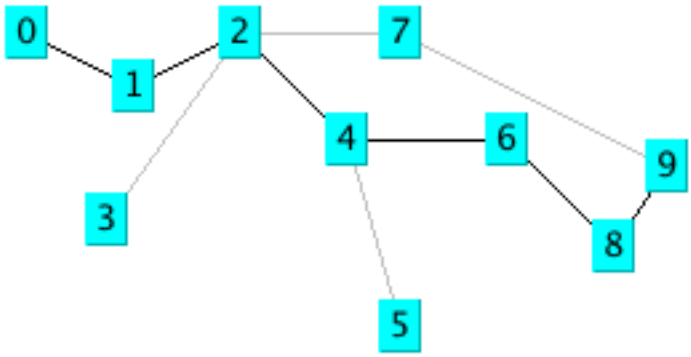

Figure 2.6: Using depth first search in a sample graph

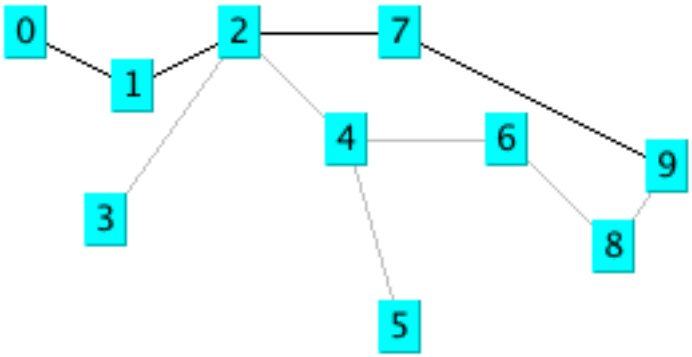

Figure 2.7: Using breadth first search in a sample graph

2.4 Adding Heuristics to Breadth First Search

We can usually make breadth first search more efficient by ordering the search order for all branches from a given position in the search space. For example, when adding new nodes from a specified reference point in the search space, we might want to add nodes to the search queue first that are "in the direction" of the goal location: in a two-dimensional search like our maze search, we might want to search connected grid cells first that were closest to the goal grid space. In this case, pre-sorting nodes (in order of closest distance to the goal) added to the breadth first search queue could have a dramatic effect on search efficiency. In the next chapter we will build a simple real-time planning system around our breadth first maze search program; this new program will use heuristics. The alpha-beta additions to breadth first search are seen in in the next section.

2.5 Search and Game Playing

Now that a computer program has won a match against the human world champion, perhaps people's expectations of AI systems will be prematurely optimistic. Game search techniques are not real AI, but rather, standard programming techniques. A better platform for doing AI research is the game of Go. There are so many possible moves in the game of Go that brute force look ahead (as is used in Chess playing programs) simply does not work.

That said, min-max type search algorithms with alpha-beta cutoff optimizations are an important programming technique and will be covered in some detail in the remainder of this chapter. We will design an abstract Java class library for implementing alpha-beta enhanced min-max search, and then use this framework to write programs to play tic-tac-toe and chess.

2.5.1 Alpha-Beta Search

The first game that we will implement will be tic-tac-toe, so we will use this simple game to explain how the min-max search (with alpha-beta cutoffs) works.

Figure 2.8 shows the possible moves generated from a tic-tac-toe position where X has made three moves and O has made two moves; it is O's turn to move. This is "level 0" in Figure 2.8. At level 0, O has four possible moves. How do we assign a fitness value to each of O's possible moves at level 0? The basic min-max search algorithm provides a simple solution to this problem: for each possible move by O in level 1, make the move and store the resulting 4 board positions. Now, at level 1, it is X's turn to move. How do we assign values to each of X's possible three moves

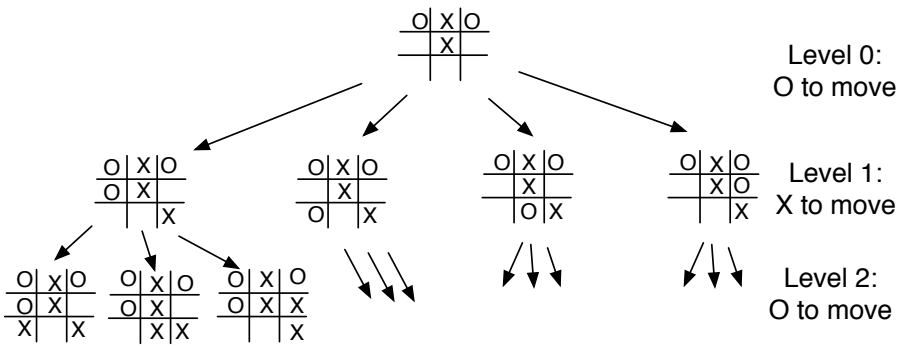

Figure 2.8: Alpha-beta algorithm applied to part of a game of tic-tac-toe

in Figure 2.8? Simple, we continue to search by making each of X's possible moves and storing each possible board position for level 2. We keep recursively applying this algorithm until we either reach a maximum search depth, or there is a win, loss, or draw detected in a generated move. We assume that there is a fitness function available that rates a given board position relative to either side. Note that the value of any board position for X is the negative of the value for O.

To make the search more efficient, we maintain values for alpha and beta for each search level. Alpha and beta determine the best possible/worst possible move available at a given level. If we reach a situation like the second position in level 2 where X has won, then we can immediately determine that O's last move in level 1 that produced this position (of allowing X an instant win) is a low valued move for O (but a high valued move for X). This allows us to immediately "prune" the search tree by ignoring all other possible positions arising from the first O move in level 1. This alpha-beta cutoff (or tree pruning) procedure can save a large percentage of search time, especially if we can set the search order at each level with "probably best" moves considered first.

While tree diagrams as seen in Figure 2.8 quickly get complicated, it is easy for a computer program to generate possible moves, calculate new possible board positions and temporarily store them, and recursively apply the same procedure to the next search level (but switching min-max "sides" in the board evaluation). We will see in the next section that it only requires about 100 lines of Java code to implement an abstract class framework for handling the details of performing an alpha-beta enhanced search. The additional game specific classes for tic-tac-toe require about an additional 150 lines of code to implement; chess requires an additional 450 lines of code.

2.5.2 A Java Framework for Search and Game Playing

The general interface for the Java classes that we will develop in this section was inspired by the Common LISP game-playing framework written by Kevin Knight and described in (Rich, Knight 1991). The abstract class GameSearch contains the code for running a two-player game and performing an alpha-beta search. This class needs to be sub-classed to provide the eight methods:

```
public abstract boolean drawnPosition(Position p)
public abstract boolean wonPosition(Position p,
                                    boolean player)
                positionEvaluation(Position p,
                                   boolean player)
public abstract void printPosition(Position p)
public abstract Position []
                possibleMoves(Position p,
                              boolean player)
public abstract Position makeMove(Position p,
                                  boolean player,
                                  Move move)
public abstract boolean reachedMaxDepth(Position p,
                                        int depth)
public abstract Move getMove()
```

The method $drawnPosition$ should return a Boolean true value if the given position evaluates to a draw situation. The method $wonPosition$ should return a true value if the input position is won for the indicated player. By convention, I use a Boolean true value to represent the computer and a Boolean false value to represent the human opponent. The method $positionEvaluation$ returns a position evaluation for a specified board position and player. Note that if we call positionEvaluation switching the player for the same board position, then the value returned is the negative of the value calculated for the opposing player. The method $possibleMoves$ returns an array of objects belonging to the class Position. In an actual game like chess, the position objects will actually belong to a chess-specific refinement of the Position class (e.g., for the chess program developed later in this chapter, the method $possibleMoves$ will return an array of $ChessPosition$ objects). The method $makeMove$ will return a new position object for a specified board position, side to move, and move. The method $reachedMaxDepth$ returns a Boolean true value if the search process has reached a satisfactory depth. For the tic-tac-toe program, the method $reachedMaxDepth$ does not return true unless either side has won the game or the board is full; for the chess program, the method $reachedMaxDepth$ returns true if the search has reached a depth of 4 half moves deep (this is not the best strategy, but it has the advantage of making the example

program short and easy to understand). The method $getMove$ returns an object of a class derived from the class $Move$ (e.g., $TicTacToeMove$ or $ChessMove$).

The $GameSearch$ class implements the following methods to perform game search:

```
protected Vector alphaBeta(int depth, Position p,
                           boolean player)
protected Vector alphaBetaHelper(int depth,
                                 Position p,
                                 boolean player,
                                 float alpha,
                                 float beta)
public void playGame(Position startingPosition,
                     boolean humanPlayFirst)
```

The method $alphaBeta$ is simple; it calls the helper method $alphaBetaHelper$ with initial search conditions; the method $alphaBetaHelper$ then calls itself recursively. The code for $alphaBeta$ is:

```
protected Vector alphaBeta(int depth,
                           Position p,
                           boolean player)  {
  Vector v = alphaBetaHelper(depth, p, player,
                             1000000.0f,
                            -1000000.0f);
  return v;
}
```

It is important to understand what is in the vector returned by the methods $alphaBeta$ and $alphaBetaHelper$. The first element is a floating point position evaluation for the point of view of the player whose turn it is to move; the remaining values are the "best move" for each side to the last search depth. As an example, if I let the tic-tac-toe program play first, it places a marker at square index 0, then I place my marker in the center of the board an index 4. At this point, to calculate the next computer move, $alphaBeta$ is called and returns the following elements in a vector:

```
next element: 0.0
next element: [-1,0,0,0,1,0,0,0,0,]
next element: [-1,1,0,0,1,0,0,0,0,]
next element: [-1,1,0,0,1,0,0,-1,0,]
next element: [-1,1,0,1,1,0,0,-1,0,]
next element: [-1,1,0,1,1,-1,0,-1,0,]
next element: [-1,1,1,1,1,-1,0,-1,0,]
```

```
 next element: [-1,1,1,1,1,-1,-1,-1,0,]
 next element: [-1,1,1,1,1,-1,-1,-1,1,]
```

Here, the alpha-beta enhanced min-max search looked all the way to the end of the game and these board positions represent what the search procedure calculated as the best moves for each side. Note that the class $TicTacToePosition$ (derived from the abstract class $Position$) has a toString method to print the board values to a string.

The same printout of the returned vector from $alphaBeta$ for the chess program is:

```
next element: 5.4
 next element:
     [4,2,3,5,9,3,2,4,7,7,1,1,1,0,1,1,1,1,7,7,
      0,0,0,0,0,0,0,0,7,7,0,0,0,1,0,0,0,0,7,7,
      0,0,0,0,0,0,0,0,7,7,0,0,0,0,-1,0,0,0,7,7,
      -1,-1,-1,-1,0,-1,-1,-1,7,7,-4,-2,-3,-5,-9,
      -3,-2,-4,]
 next element:
     [4,2,3,0,9,3,2,4,7,7,1,1,1,5,1,1,1,1,7,7,
      0,0,0,0,0,0,0,0,7,7,0,0,0,1,0,0,0,0,7,7,
      0,0,0,0,0,0,0,0,7,7,0,0,0,0,-1,0,0,0,7,7,
      -1,-1,-1,-1,0,-1,-1,-1,7,7,-4,-2,-3,-5,-9,
      -3,-2,-4,]
 next element:
     [4,2,3,0,9,3,2,4,7,7,1,1,1,5,1,1,1,1,7,7,
      0,0,0,0,0,0,0,0,7,7,0,0,0,1,0,0,0,0,7,7,
      0,0,0,0,0,0,0,0,7,7,0,0,0,0,-1,-5,0,0,7,7,
      -1,-1,-1,-1,0,-1,-1,-1,7,7,-4,-2,-3,0,-9,
      -3,-2,-4,]
 next element:
     [4,2,3,0,9,3,0,4,7,7,1,1,1,5,1,1,1,1,7,7,
      0,0,0,0,0,2,0,0,7,7,0,0,0,1,0,0,0,0,7,7,
      0,0,0,,0,0,0,0,0,7,7,0,0,0,0,-1,-5,0,0,7,7,
      -1,-1,-1,-1,0,-1,-1,-1,7,7,-4,-2,-3,0,-9,
      -3,-2,-4,]
 next element:
     [4,2,3,0,9,3,0,4,7,7,1,1,1,5,1,1,1,1,7,7,
      0,0,0,0,0,2,0,0,7,7,0,0,0,1,0,0,0,0,7,7,
      -1,0,0,0,0,0,0,0,7,7,0,0,0,0,-1,-5,0,0,7,7,
      0,-1,-1,-1,0,-1,-1,-1,7,7,-4,-2,-3,0,-9,
      -3,-2,-4,]
```

Here, the search procedure assigned the side to move (the computer) a position evaluation score of 5.4; this is an artifact of searching to a fixed depth. Notice that the board representation is different for chess, but because the $GameSearch$ class manipulates objects derived from the classes $Position$ and $Move$, the GameSearch class does not need to have any knowledge of the rules for a specific game. We will discuss the format of the chess position class $ChessPosition$ in more detail when we develop the chess program.

The classes Move and Position contain no data and methods at all. The classes Move and Position are used as placeholders for derived classes for specific games. The search methods in the abstract GameSearch class manipulate objects derived from the classes Move and Position.

Now that we have seen the debug printout of the contents of the vector returned from the methods $alphaBeta$ and $alphaBetaHelper$, it will be easier to understand how the method $alphaBetaHelper$ works. The following text shows code fragments from the $alphaBetaHelper$ method interspersed with book text:

```
protected Vector alphaBetaHelper(int depth,
                                 Position p,
                                 boolean player,
                                 float alpha,
                                 float beta) {
```

Here, we notice that the method signature is the same as for $alphaBeta$, except that we pass floating point alpha and beta values. The important point in understanding min-max search is that most of the evaluation work is done while "backing up" the search tree; that is, the search proceeds to a leaf node (a node is a leaf if the method $reachedMaxDepth$ return a Boolean true value), and then a return vector for the leaf node is created by making a new vector and setting its first element to the position evaluation of the position at the leaf node and setting the second element of the return vector to the board position at the leaf node:

```
  if (reachedMaxDepth(p, depth)) {
    Vector v = new Vector(2);
    float value = positionEvaluation(p, player);
    v.addElement(new Float(value));
    v.addElement(p);
    return v;
  }
```

If we have not reached the maximum search depth (i.e., we are not yet at a leaf node in the search tree), then we enumerate all possible moves from the current position using the method $possibleMoves$ and recursively call $alphaBetaHelper$ for each

new generated board position. In terms of Figure 2.8, at this point we are moving down to another search level (e.g., from level 1 to level 2; the level in Figure 2.8 corresponds to depth argument in $alphaBetaHelper$):

```
Vector best = new Vector();
Position [] moves = possibleMoves(p, player);
for (int i=0; i<moves.length; i++) {
  Vector v2 = alphaBetaHelper(depth + 1, moves[i],
                              !player,
                              -beta, -alpha);
  float value = -((Float)v2.elementAt(0)).floatValue();
  if (value > beta) {
    if(GameSearch.DEBUG)
      System.out.println(" ! ! ! value="+
                         value+
                         ",beta="+beta);
    beta = value;
    best = new Vector();
    best.addElement(moves[i]);
    Enumeration enum = v2.elements();
    enum.nextElement(); // skip previous value
    while (enum.hasMoreElements()) {
      Object o = enum.nextElement();
      if (o != null) best.addElement(o);
    }
  }
  /**
   * Use the alpha-beta cutoff test to abort
   * search if we found a move that proves that
   * the previous move in the move chain was dubious
   */
  if (beta >= alpha) {
    break;
  }
}
```

Notice that when we recursively call $alphaBetaHelper$, we are "flipping" the player argument to the opposite Boolean value. After calculating the best move at this depth (or level), we add it to the end of the return vector:

```
Vector v3 = new Vector();
v3.addElement(new Float(beta));
Enumeration enum = best.elements();
```

```
    while (enum.hasMoreElements()) {
      v3.addElement(enum.nextElement());
    }
    return v3;
```

When the recursive calls back up and the first call to $alphaBetaHelper$ returns a vector to the method $alphaBeta$, all of the "best" moves for each side are stored in the return vector, along with the evaluation of the board position for the side to move.

The class $GameSearch$ method $playGame$ is fairly simple; the following code fragment is a partial listing of playGame showing how to call $alphaBeta$, $getMove$, and $makeMove$:

```
  public void playGame(Position startingPosition,
                       boolean humanPlayFirst) {
    System.out.println("Your move:");
    Move move = getMove();
    startingPosition = makeMove(startingPosition,
                                HUMAN, move);
    printPosition(startingPosition);
    Vector v = alphaBeta(0, startingPosition, PROGRAM);
    startingPosition = (Position)v.elementAt(1);
    }
  }
```

The debug printout of the vector returned from the method $alphaBeta$ seen earlier in this section was printed using the following code immediately after the call to the method $alphaBeta$:

```
      Enumeration enum = v.elements();
      while (enum.hasMoreElements()) {
        System.out.println(" next element: " +
                           enum.nextElement());
      }
```

In the next few sections, we will implement a tic-tac-toe program and a chess-playing program using this Java class framework.

2.5.3 Tic-Tac-Toe Using the Alpha-Beta Search Algorithm

Using the Java class framework of $GameSearch$, $Position$, and $Move$, it is simple to write a basic tic-tac-toe program by writing three new derived classes (see Figure

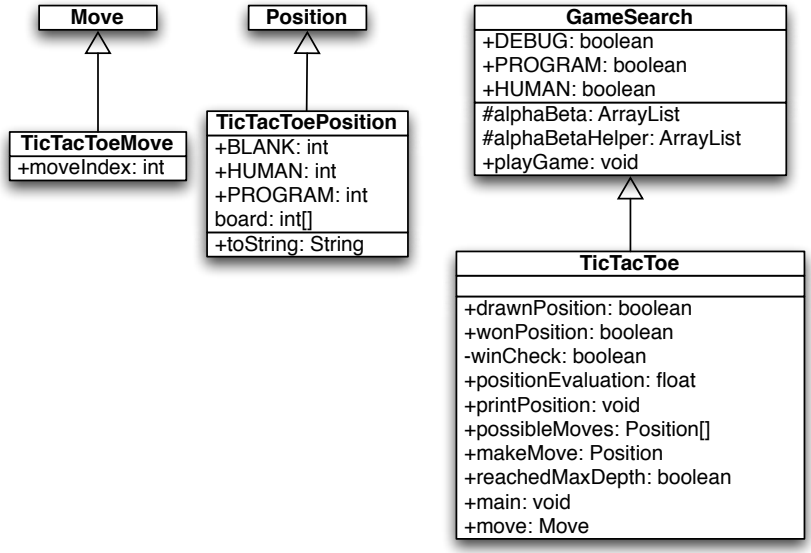

Figure 2.9: UML class diagrams for game search engine and tic-tac-toe

2.9) $TicTacToe$ (derived from $GameSearch$), $TicTacToeMove$ (derived from $Move$), and $TicTacToePosition$ (derived from $Position$).

I assume that the reader has the book example code installed and available for viewing. In this section, I will only discuss the most interesting details of the tic-tac-toe class refinements; I assume that the reader can look at the source code. We will start by looking at the refinements for the position and move classes. The $TicTacToeMove$ class is trivial, adding a single integer value to record the square index for the new move:

```
public class TicTacToeMove extends Move {
  public int moveIndex;
}
```

The board position indices are in the range of [0..8] and can be considered to be in the following order:

```
0 1 2
3 4 5
6 7 8
```

The class $TicTacToePosition$ is also simple:

```
public class TicTacToePosition extends Position {
    final static public int BLANK = 0;
    final static public int HUMAN = 1;
    final static public int PROGRAM = -1;
    int [] board = new int[9];
    public String toString() {
        StringBuffer sb = new StringBuffer("[");
        for (int i=0; i<9; i++)
          sb.append(""+board[i]+",");
        sb.append("]");
        return sb.toString();
    }
}
```

This class allocates an array of nine integers to represent the board, defines constant values for blank, human, and computer squares, and defines a toString method to print out the board representation to a string.

The $TicTacToe$ class must define the following abstract methods from the base class $GameSearch$:

```
public abstract boolean drawnPosition(Position p)
public abstract boolean wonPosition(Position p,
                                    boolean player)
public abstract float positionEvaluation(Position p,
                                         boolean player)
public abstract void printPosition(Position p)
public abstract Position [] possibleMoves(Position p,
                                          boolean player)
public abstract Position makeMove(Position p,
                                  boolean player,
                                  Move move)
public abstract boolean reachedMaxDepth(Position p,
                                        int depth)
public abstract Move getMove()
```

The implementation of these methods uses the refined classes $TicTacToeMove$ and $TicTacToePosition$. For example, consider the method $drawnPosition$ that is responsible for selecting a drawn (or tied) position:

```
  public boolean drawnPosition(Position p) {
    boolean ret = true;
    TicTacToePosition pos = (TicTacToePosition)p;
```

```
    for (int i=0; i<9; i++) {
      if (pos.board[i] == TicTacToePosition.BLANK){
        ret = false;
        break;
      }
    }
    return ret;
  }
```

The overridden methods from the $GameSearch$ base class must always cast arguments of type $Position$ and $Move$ to $TicTacToePosition$ and $TicTacToeMove$. Note that in the method $drawnPosition$, the argument of class $Position$ is cast to the class $TicTacToePosition$. A position is considered to be a draw if all of the squares are full. We will see that checks for a won position are always made before checks for a drawn position, so that the method $drawnPosition$ does not need to make a redundant check for a won position. The method $wonPosition$ is also simple; it uses a private helper method $winCheck$ to test for all possible winning patterns in tic-tac-toe. The method $positionEvaluation$ uses the following board features to assign a fitness value from the point of view of either player:

```
The number of blank squares on the board
If the position is won by either side
If the center square is taken
```

The method $positionEvaluation$ is simple, and is a good place for the interested reader to start modifying both the tic-tac-toe and chess programs:

```
    public float positionEvaluation(Position p,
                                    boolean player) {
        int count = 0;
        TicTacToePosition pos = (TicTacToePosition)p;
        for (int i=0; i<9; i++) {
            if (pos.board[i] == 0) count++;
        }
        count = 10 - count;
        // prefer the center square:
        float base = 1.0f;
        if (pos.board[4] == TicTacToePosition.HUMAN &&
            player) {
            base += 0.4f;
        }
        if (pos.board[4] == TicTacToePosition.PROGRAM &&
            !player) {
```

```
            base -= 0.4f;
        }
        float ret = (base - 1.0f);
        if (wonPosition(p, player))  {
            return base + (1.0f / count);
        }
        if (wonPosition(p, !player))  {
            return -(base + (1.0f / count));
        }
        return ret;
    }
```

The only other method that we will look at here is $possibleMoves$; the interested reader can look at the implementation of the other (very simple) methods in the source code. The method $possibleMoves$ is called with a current position, and the side to move (i.e., program or human):

```
    public Position [] possibleMoves(Position p,
                                     boolean player) {
        TicTacToePosition pos = (TicTacToePosition)p;
        int count = 0;
        for (int i=0; i<9; i++) {
            if (pos.board[i] == 0) count++;
        }
        if (count == 0) return null;
        Position [] ret = new Position[count];
        count = 0;
        for (int i=0; i<9; i++) {
            if (pos.board[i] == 0) {
                TicTacToePosition pos2 =
                           new  TicTacToePosition();
                for (int j=0; j<9; j++)
                     pos2.board[j] = pos.board[j];
                if (player) pos2.board[i] = 1;
                else pos2.board[i] = -1;
                ret[count++] = pos2;
            }
        }
        return ret;
    }
```

It is very simple to generate possible moves: every blank square is a legal move. (This method will not be as straightforward in the example chess program!)

It is simple to compile and run the example tic-tac-toe program: change directory to src-search-game and type:

```
javac *.java
java TicTacToe
```

When asked to enter moves, enter an integer between 0 and 8 for a square that is currently blank (i.e., has a zero value). The following shows this labeling of squares on the tic-tac-toe board:

```
0 1 2
3 4 5
6 7 8
```

2.5.4 Chess Using the Alpha-Beta Search Algorithm

Using the Java class framework of $GameSearch$, $Position$, and $Move$, it is reasonably easy to write a simple chess program by writing three new derived classes (see Figure 2.10) $Chess$ (derived from $GameSearch$), $ChessMove$ (derived from $Move$), and $ChessPosition$ (derived from $Position$). The chess program developed in this section is intended to be an easy to understand example of using alpha-beta min-max search; as such, it ignores several details that a fully implemented chess program would implement:

- Allow the computer to play either side (computer always plays black in this example).
- Allow en-passant pawn captures.
- Allow the player to take back a move after making a mistake.

The reader is assumed to have read the last section on implementing the tic-tac-toe game; details of refining the GameSearch, Move, and Position classes are not repeated in this section.

Figure 2.10 shows the UML class diagram for both the general purpose GameSearch framework and the classes derived to implement chess specific data and behavior.

The class ChessMove contains data for recording from and to square indices:

```
public class ChessMove extends Move {
    public int from;
    public int to;
}
```

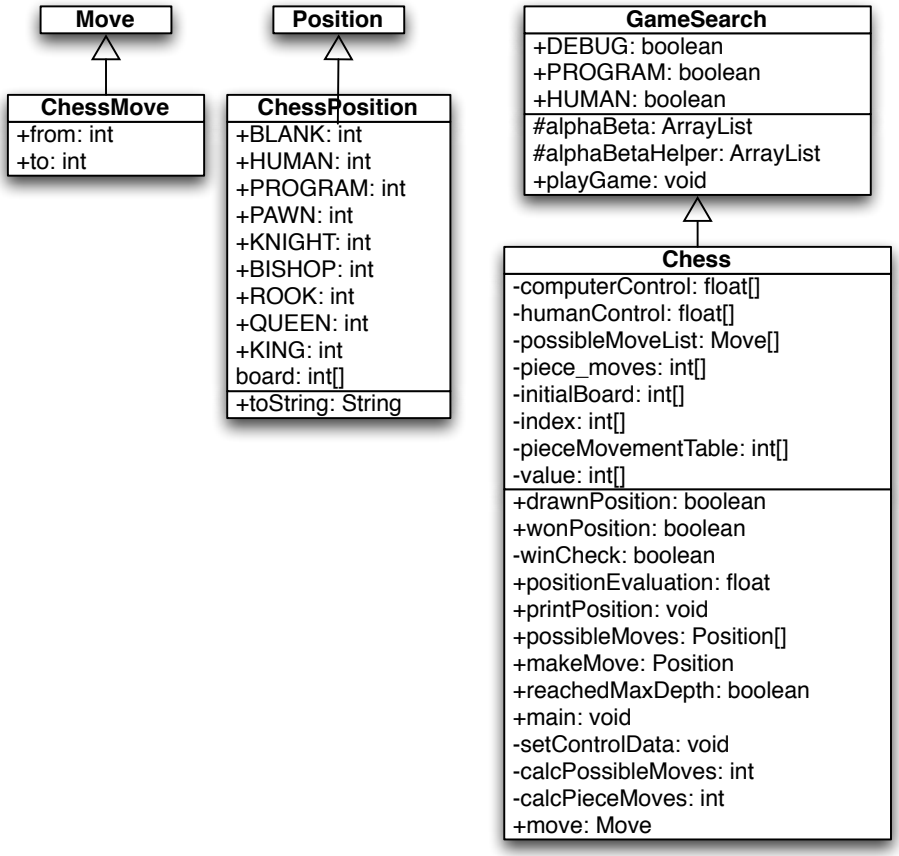

Figure 2.10: UML class diagrams for game search engine and chess

1 c4 b6 2 d4 ♗b7 Black increases the mobility of its pieces by fianchettoing the queenside bishop:

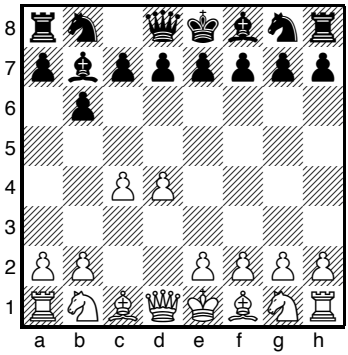

Figure 2.11: The example chess program does not contain an opening book so it plays to maximize the mobility of its pieces and maximize material advantage using a two-move lookahead. The first version of the chess program contains a few heuristics like wanting to control the center four squares.

The board is represented as an integer array with 120 elements. A chessboard only has 64 squares; the remaining board values are set to a special value of 7, which indicates an "off board" square. The initial board setup is defined statically in the Chess class and the off-board squares have a value of "7":

```
private static int [] initialBoard = {
 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7,
 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7,
 4, 2, 3, 5, 9, 3, 2, 4, 7, 7,   // white pieces
 1, 1, 1, 1, 1, 1, 1, 1, 7, 7,   // white pawns
 0, 0, 0, 0, 0, 0, 0, 0, 7, 7,   // 8 blank squares
 0, 0, 0, 0, 0, 0, 0, 0, 7, 7,   // 8 blank squares
 0, 0, 0, 0, 0, 0, 0, 0, 7, 7,   // 8 blank squares
 0, 0, 0, 0, 0, 0, 0, 0, 7, 7,   // 8 blank squares
 -1,-1,-1,-1,-1,-1,-1,-1, 7, 7,  // black pawns
 -4,-2,-3,-5,-9,-3,-2,-4, 7, 7,  // black pieces
 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7,
 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7
};
```

It is difficult to see from this listing of the board square values but in effect a regular chess board if padded on all sides with two rows and columns of "7" values.

3 ♘f3 g6 4 ♗f4 ♗g7 5 ♘c3 Black (the computer) continues to increase piece mobility and control the center squares:

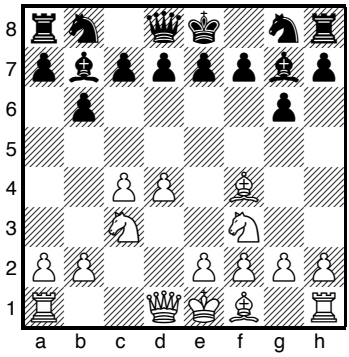

Figure 2.12: Continuing the first sample game: the computer is looking ahead two moves and no opening book is used.

We see the start of a sample chess game in Figure 2.11 and the continuation of this same game in Figure 2.12.The lookahead is limited to 2 moves (4 ply).

The class $ChessPosition$ contains data for this representation and defines constant values for playing sides and piece types:

```
public class ChessPosition extends Position {
  final static public int BLANK = 0;
  final static public int HUMAN = 1;
  final static public int PROGRAM = -1;
  final static public int PAWN = 1;
  final static public int KNIGHT = 2;
  final static public int BISHOP = 3;
  final static public int ROOK = 4;
  final static public int QUEEN = 5;
  final static public int KING = 6;
  int [] board = new int[120];
  public String toString() {
      StringBuffer sb = new StringBuffer("[");
      for (int i=22; i<100; i++) {
          sb.append(""+board[i]+",");
      }
      sb.append("]");
      return sb.toString();
  }
```

```
}
```

The class $Chess$ also defines other static data. The following array is used to encode the values assigned to each piece type (e.g., pawns are worth one point, knights and bishops are worth 3 points, etc.):

```
private static int [] value = {
  0, 1, 3, 3, 5, 9, 0, 0, 0, 12
};
```

The following array is used to codify the possible incremental moves for pieces:

```
private static int [] pieceMovementTable = {
  0, -1, 1, 10, -10, 0, -1, 1, 10, -10, -9, -11, 9,
  11, 0, 8, -8, 12, -12, 19, -19, 21, -21, 0, 10, 20,
  0, 0, 0, 0, 0, 0, 0, 0
};
```

The starting index into the pieceMovementTable array is calculated by indexing the following array with the piece type index (e.g., pawns are piece type 1, knights are piece type 2, bishops are piece type 3, rooks are piece type 4, etc.:

```
private static int [] index = {
  0, 12, 15, 10, 1, 6, 0, 0, 0, 6
};
```

When we implement the method $possibleMoves$ for the class $Chess$, we will see that except for pawn moves, all other possible piece type moves are very easy to calculate using this static data. The method $possibleMoves$ is simple because it uses a private helper method $calcPieceMoves$ to do the real work. The method $possibleMoves$ calculates all possible moves for a given board position and side to move by calling $calcPieceMove$ for each square index that references a piece for the side to move.

We need to perform similar actions for calculating possible moves and squares that are controlled by each side. In the first version of the class Chess that I wrote, I used a single method for calculating both possible move squares and controlled squares. However, the code was difficult to read, so I split this initial move generating method out into three methods:

- possibleMoves – required because this was an abstract method in GameSearch. This method calls calcPieceMoves for all squares containing pieces for the side to move, and collects all possible moves.

- calcPieceMoves – responsible to calculating pawn moves and other piece type moves for a specified square index.

- setControlData – sets the global array computerControl and humanControl. This method is similar to a combination of possibleMoves and calcPiece-Moves, but takes into effect "moves" onto squares that belong to the same side for calculating the effect of one piece guarding another. This control data is used in the board position evaluation method $positionEvaluation$.

We will discuss $calcPieceMoves$ here, and leave it as an exercise to carefully read the similar method $setControlData$ in the source code. This method places the calculated piece movement data in static storage (the array piece_moves) to avoid creating a new Java object whenever this method is called; method $calcPieceMoves$ returns an integer count of the number of items placed in the static array piece_moves. The method $calcPieceMoves$ is called with a position and a square index; first, the piece type and side are determined for the square index:

```
private int calcPieceMoves(ChessPosition pos,
                           int square_index) {
  int [] b = pos.board;
  int piece = b[square_index];
  int piece_type = piece;
  if (piece_type < 0) piece_type = -piece_type;
  int piece_index = index[piece_type];
  int move_index = pieceMovementTable[piece_index];
  if (piece < 0) side_index = -1;
  else              side_index = 1;
```

Then, a switch statement controls move generation for each type of chess piece (movement generation code is not shown – see the file Chess.java):

```
  switch (piece_type) {
  case ChessPosition.PAWN:
    break;
  case ChessPosition.KNIGHT:
  case ChessPosition.BISHOP:
  case ChessPosition.ROOK:
  case ChessPosition.KING:
  case ChessPosition.QUEEN:
    break;
  }
```

The logic for pawn moves is a little complex but the implementation is simple. We start by checking for pawn captures of pieces of the opposite color. Then check for

initial pawn moves of two squares forward, and finally, normal pawn moves of one square forward. Generated possible moves are placed in the static array piece_moves and a possible move count is incremented. The move logic for knights, bishops, rooks, queens, and kings is very simple since it is all table driven. First, we use the piece type as an index into the static array index; this value is then used as an index into the static array $pieceMovementTable$. There are two loops: an outer loop fetches the next piece movement delta from the $pieceMovementTable$ array and the inner loop applies the piece movement delta set in the outer loop until the new square index is off the board or "runs into" a piece on the same side. Note that for kings and knights, the inner loop is only executed one time per iteration through the outer loop:

```
    move_index = piece;
    if (move_index < 0) move_index = -move_index;
    move_index = index[move_index];
    //System.out.println("move_index="+move_index);
    next_square =
        square_index + pieceMovementTable[move_index];
  outer:
    while (true) {
      inner:
        while (true) {
          if (next_square > 99) break inner;
          if (next_square < 22) break inner;
          if (b[next_square] == 7) break inner;

          // check for piece on the same side:
          if (side_index < 0 && b[next_square] < 0)
             break inner;
          if (side_index >0 && b[next_square]  > 0)
             break inner;

          piece_moves[count++] = next_square;
          if (b[next_square] != 0) break inner;
          if (piece_type == ChessPosition.KNIGHT)
             break inner;
          if (piece_type == ChessPosition.KING)
             break inner;
          next_square += pieceMovementTable[move_index];
        }
        move_index += 1;
        if (pieceMovementTable[move_index] == 0)
           break outer;
        next_square = square_index +
```

1 d4 e6 2 e4 ♕h4 Black (the computer) increases the mobility of its pieces by bringing out the queen early but we will see that this soon gets black in trouble.

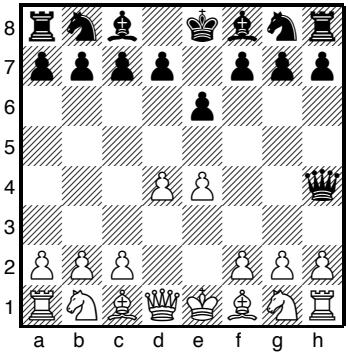

Figure 2.13: Second game with a 2 1/2 move lookahead.

```
                                pieceMovementTable[move_index];
    }
```

Figure 2.13 shows the start of a second example game. The computer was making too many trivial mistakes in the first game so here I increased the lookahead to 2 1/2 moves. Now the computer takes one to two seconds per move and plays a better game. Increasing the lookahead to 3 full moves yields a better game but then the program can take up to about ten seconds per move.

The method $setControlData$ is very similar to this method; I leave it as an exercise to the reader to read through the source code. Method $setControlData$ differs in also considering moves that protect pieces of the same color; calculated square control data is stored in the static arrays $computerControl$ and $humanControl$. This square control data is used in the method $positionEvaluation$ that assigns a numerical rating to a specified chessboard position on either the computer or human side. The following aspects of a chessboard position are used for the evaluation:

- material count (pawns count 1 point, knights and bishops 3 points, etc.)
- count of which squares are controlled by each side
- extra credit for control of the center of the board
- credit for attacked enemy pieces

Notice that the evaluation is calculated initially assuming the computer's side to move. If the position if evaluated from the human player's perspective, the evalua-

3 ♘c3 ♘f6 4 ♗d3 ♗b4 5 ♘f3 ♕h5 Black continues to develop pieces and puts pressure on the pawn on E4 but the vulnerable queen makes this a weak position for black:

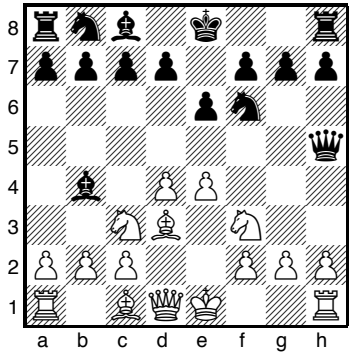

Figure 2.14: Continuing the second game with a two and a half move lookahead. We will add more heuristics to the static evaluation method to reduce the value of moving the queen early in the game.

tion value is multiplied by minus one. The implementation of *positionEvaluation* is:

```
public float positionEvaluation(Position p,
                                boolean player) {
  ChessPosition pos = (ChessPosition)p;
  int [] b = pos.board;
  float ret = 0.0f;
  // adjust for material:
  for (int i=22; i<100; i++) {
    if (b[i] != 0 && b[i] != 7)  ret += b[i];
  }

  // adjust for positional advantages:
  setControlData(pos);
  int control = 0;
  for (int i=22; i<100; i++) {
    control += humanControl[i];
    control -= computerControl[i];
  }
  // Count center squares extra:
  control += humanControl[55] - computerControl[55];
  control += humanControl[56] - computerControl[56];
  control += humanControl[65] - computerControl[65];
```

```
    control += humanControl[66] - computerControl[66];

    control /= 10.0f;
    ret += control;

    // credit for attacked pieces:
    for (int i=22; i<100; i++) {
      if (b[i] == 0 || b[i] == 7) continue;
      if (b[i] < 0) {
        if (humanControl[i] > computerControl[i]) {
            ret += 0.9f * value[-b[i]];
        }
      }
      if (b[i] > 0) {
        if (humanControl[i] < computerControl[i]) {
          ret -= 0.9f * value[b[i]];
        }
      }
    }
    // adjust if computer side to move:
    if (!player) ret = -ret;
    return ret;
  }
```

It is simple to compile and run the example chess program by changing directory to src-search-game and typing:

```
javac *.java
java Chess
```

When asked to enter moves, enter string like “d2d4” to enter a move in chess algebraic notation. Here is sample output from the program:

```
Board position:

 BR BN BB .  BK BB BN BR
 BP BP BP BP .  BP BP BP
    .     .  BP BQ    .
 .     .     .     .
    .     WP    .     .
 .     .     .  WN .
 WP WP WP .  WP WP WP WP
 WR WN WB WQ WK WB .  WR
```

```
Your move:
c2c4
```

Class.method name	% of total runtime	% in this method
Chess.main	97.7	0.0
GameSearch.playGame	96.5	0.0
GameSearch.alphaBeta	82.6	0.0
GameSearch.alphaBetaHelper	82.6	0.0
Chess.positionEvaluate	42.9	13.9
Chess.setControlData	29.1	29.1
Chess.possibleMoves	23.2	11.3
Chess.calcPossibleMoves	1.7	0.8
Chess.calcPieceMoves	1.7	0.8

Table 2.1: Runtimes by Method for Chess Program

The example chess program plays in general good moves, but its play could be greatly enhanced with an "opening book" of common chess opening move sequences. If you run the example chess program, depending on the speed of your computer and your Java runtime system, the program takes a while to move (about 5 seconds per move on my PC). Where is the time spent in the chess program? Table 2.1 shows the total runtime (i.e., time for a method and recursively all called methods) and method-only time for the most time consuming methods. Methods that show zero percent method only time used less than 0.1 percent of the time so they print as zero values.

The interested reader is encouraged to choose a simple two-player game, and using the game search class framework, implement your own game-playing program.

3 Reasoning

Reasoning is a broad topic. In this chapter we will concentrate on the use of the PowerLoom descriptive logic reasoning system. PowerLoom is available with a Java runtime and Java API – this is what I will use for the examples in this chapter. PowerLoom can also be used with JRuby. PowerLoom is available in Common Lisp and C++ versions.

Additionally, we will look briefly at different kinds of reasoning systems in Chapter 4 on the Semantic Web.

While the material in this chapter will get you started with development using a powerful reasoning system and embedding this reasoning system in Java applications, you will likely want to dig deeper and I suggest sources for further study at the end of this chapter.

PowerLoom is a newer version of the classic Loom Descriptive Logic reasoning system written at ISI. The required JAR files for PowerLoom are included in the ZIP file for this book but at some point you will probably want to download the entire PowerLoom distribution to get more examples and access to documentation; the PowerLoom web site can be found at http://www.isi.edu/isd/LOOM/PowerLoom/.

While we will look at an example of embedding the PowerLoom runtime and a PowerLoom model in a Java example program, I want to make a general comment on PowerLoom development: you will spend most of your time interactively running PowerLoom in an interactive shell that lets you type in concepts, relations, rules, and queries and immediately see the results. If you have ever programmed in Lisp, then this mode of interactive programming will be familiar to you. As seen in Figure 3.1 after interactive development you can deploy in a Java application. This style of development supports entering facts and trying rules and relations interactively and as you get things working you can paste what works into a PowerLoom source file. If you have only worked with compiled langauges like Java and C++ this development style may take a while to get used to and appreciate. As seen in Figure 3.1 the PowerLoom runtime system, with relations and rules, can be embedded in Java applications that typically clear PowerLoom data memory, assert facts from other live data sources, and then use PowerLoom for inferencing.

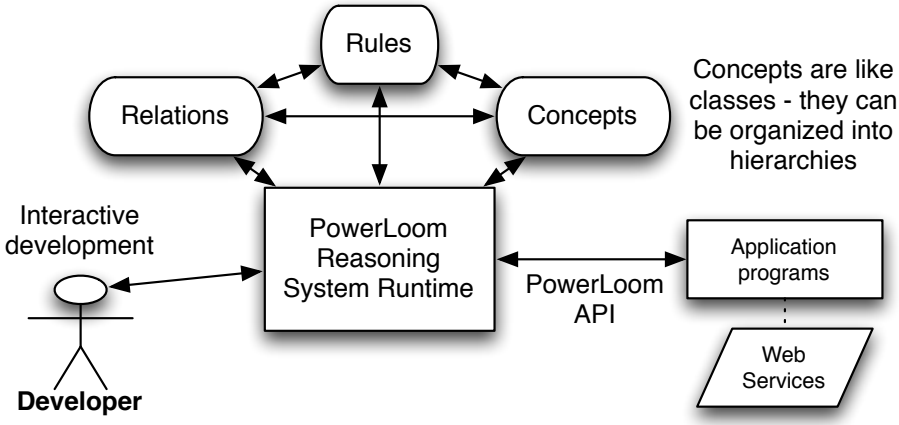

Figure 3.1: Overview of how we will use PowerLoom for development and deployment

3.1 Logic

We will look at different types of logic and reasoning systems in this section and then get into PowerLoom specific examples in Section 3.2. Logic is the basis for both Knowledge Representation and for reasoning about knowledge. We will encode knowledge using logic and see that we can then infer new facts that are not explicitly asserted.

First Order Logic was invented by the philosophers Frege and Peirce and is the most widely studied logic system. Unfortunately, full First Order Logic is not computationally tractable for most non-trivial problems so we use more restricted logics. We will use two reasoning systems in this book that support more limited logics:

- We use PowerLoom is this chapter. PowerLoom supports a combination of limited first order predicate logic and features of description logic. PowerLoom is able to classify objects, use rules to infer facts from existing facts and to perform subsumption (determining class membership of instances).
- We will use RDF Schema (RDFS) reasoning in Chapter 4. RDFS supports more limited reasoning than descriptive logic reasoners like PowerLoom and OWL Description Logic reasoners.

3.1.1 History of Logic

The Greek philosopher Aristotle studied forms of logic as part of his desire to improve the representation of knowledge. He started a study of logic and the definition of both terms (e.g., subjects, predicates, nouns, verbs) and types of logical deduction. Much later the philosopher Frege defined predicate logic (for example: All birds have feathers. Brady is a bird, therefore Brady has feathers) that forms the basis for the modern Prolog programming language.

3.1.2 Examples of Different Logic Types

Propositional logic is limited to atomic statements that can be either true or false:

```
Brady-is-a-bird
Brady-has-feathers
```

First Order Predicate Logic allows access to the structure of logic statements dealing with predicates that operate on atoms. To use a Prolog notation:

```
feathers(X) :- bird(X).
bird(brady).
```

Here "feathers" and "bird" are predicates and "brady" is an atom. The first example states that for all X, if X is a bird, then X has feathers. In the second example we state that Brady is a bird. Notice that in the Prolog notation that we are using, variables are capitalized and predicate names and literal atoms are lower case.

Here is a query that asks who has feathers:

```
?- feathers(X).
X = brady
```

In this example through inference we have determined a new fact, that Brady has feathers because we know that Brady is a bird and we have the rule (or predicate) stating that all birds have feathers. Prolog is not strictly a pure logic programming language since the order in which rules (predicates) are defined chnages the inference results. Prolog is a great language for some types of projects (I have used Prolog in both natural language processing and in planning projects). We will see that PowerLoom is considerably more flexible than Prolog but does have a steep learning curve.

Description Logic deals with descriptions of concepts and how these descriptions define the domain of concepts. In terms used in object oriented programming languages: membership in a class is determined implicitly by the description of the object and not by explicitly stating something like "Brady is a member of the bird class." Description logics divide statements into relations (historically refered to as TBox) and concepts (historically called ABox). We would say that a statement like "All birds have feathers" is stored in the TBox while a specific assertion like "Brady is a bird" is stored in the ABox.

3.2 PowerLoom Overview

PowerLoom is designed to be an expressive language for knowledge representation and reasoning. As a result, PowerLoom is not a complete reasoning system but makes tradeoffs for completeness of inferences and expressivity vs. computational efficiency. It is interesting to note that Loom and PowerLoom were designed and implemented to solve real world problems and the tradeoffs to make these problems computationally tractable have informed the design and implementation of these systems. PowerLoom does not make all posible inferences from concepts that it operates on.

The PowerLoom distribution contains two very detailed examples for representing relationships between companies and for information dealing with airplanes. These examples are more detailed than the simpler example of data from news stories used in this chapter. We will look one of these examples (business rules and relations) and after working through this chapter, I encourage you to interactively experiment with the two examples that ship with PowerLoom.

We will start by defining some terms used in PowerLoom:

- concept – the Java equivalent would be an instance of a class
- relation – specifies a link between two concepts
- function – functional mapping of one concept to another
- rule – allows new concepts to be deduced without explicitly asserting them

A relation can specify the types of concepts that a relation connects. An example will make this clear and introduce the Lisp-like syntax of PowerLoom statements:

```
;;; Concepts:
(defconcept person)
(defconcept parent (?p person))
```

```
;;; Relation:
(defrelation parent-of ((?p1 parent) (?p2 person)))
```

Here I have defined two concepts: person and parent. Note that we have a hierarchy of concept types here: the parent is a more specific concept type than the person concept. All instances that are parents are also of type person. The relation parent-of links a parent concept to a person concept.

We will learn more about basic PowerLoom functionality in the next two sections as we use PowerLoom in an interactive session and when we embed PowerLoom in a Java example program.

3.3 Running PowerLoom Interactively

We will experiment with PowerLoom concepts, relations, and rules in this section in an interactive command shell. I will introduce more examples of PowerLoom functionality for asserting instances of concepts, performing queries, loading PowerLoom source files, defining relations, using separate modules, and asking PowerLoom to explain the inference process that it used for query processing.

You can run PowerLoom using the command line interface by changing directory to the lib subdirectory from the ZIP file for this book and trying:

```
java -cp powerloom.jar:stella.jar \\
      edu.isi.powerloom.PowerLoom
```

This starts the PowerLoom standalone system and prints a prompt that includes the name of the current module. The default module name is "PL-USER". In the first example, when I enter the person concept at the interactive prompt then PowerLoom prints the result of the expression that just entered.

```
PL-USER |= (defconcept person)
|c|PERSON
PL-USER |= (defconcept parent (?p person))
|c|PARENT
PL-USER |= (defrelation parent-of
              ((?p1 parent) (?p2 person)))
|r|PARENT-OF
PL-USER |= (assert (person Ken))
|P|(PERSON KEN)
PL-USER |= (assert (person Mark))
```

```
|P|(PERSON MARK)
PL-USER |= (assert (parent-of Ken Mark))
|P|(PARENT-OF KEN MARK)
```

Now that we have entered two concepts, a test relation, and asserted a few facts, we can look at an example of PowerLoom's query language:

```
PL-USER |= (retrieve all ?p (person ?p))
There are 2 solutions:
  #1: ?P=MARK
  #2: ?P=KEN
PL-USER |= (retrieve all ?p (parent ?p))
There is 1 solution:
  #1: ?P=KEN
PL-USER |=
```

The obvious point to note from this example is that we never specified that Ken was a parent; rather, PowerLoom deduced this from the parent-of relation.

PowerLoom's command line system prompts you with the string "PL-USER —=" and you can type any definition or query. Like Lisp, PowerLoom uses a prefix notation and expressions are contained in parenthesis. PowerLoom supports a module system for partitioning concepts, relations, functions, and rules into different sets and as previously mentioned "PL-USER" is the default module. PowerLoom modules can form a hierarchy, inheriting concepts, relations, and rules from parent modules.

The subdirectory test_data contains the demo file business.plm written by Robert MacGregor that is supplied with the full PowerLoom distribution. You can load his complete example using:

```
PL-USER |= (load "../test_data/business.plm")
```

This is a good example because it demonstrates most of the available functionality of PowerLoom in a short 200 lines. When you are done reading this chapter, please take a few minutes to read through this example file since I do not list it here. There are a few things to notice in this example. Here we see a rule used to make the relation "contains" transitive:

```
(defrelation contains (
       (?l1 geographic-location)
       (?l2 geographic-location)))
```

```
(defrule transitive-contains
  (=> (and (contains ?l1 ?l2)
           (contains ?l2 ?l3))
      (contains ?l1 ?l3)))
```

The operator => means that if the first clause is true then so is the second. In English, this rule could be stated "if an instance i1 contains i2 and if instance i2 contains i3 then we can infer that i1 also contains i3." To see how this rule works in practice, we can switch to the example module "BUSINESS" and find all locations contained inside another location:

```
PL-USER |= (in-module "BUSINESS")
BUSINESS |= (retrieve all
              (?location1 ?location2)
              (contains ?location1 ?location2))
There are 15 solutions:
  #1: ?LOCATION1=SOUTHERN-US, ?LOCATION2=TEXAS
  #2: ?LOCATION1=TEXAS, ?LOCATION2=AUSTIN
  #3: ?LOCATION1=TEXAS, ?LOCATION2=DALLAS
  #4: ?LOCATION1=UNITED-STATES, ?LOCATION2=SOUTHERN-US
  #5: ?LOCATION1=GEORGIA, ?LOCATION2=ATLANTA
  #6: ?LOCATION1=EASTERN-US, ?LOCATION2=GEORGIA
  #7: ?LOCATION1=UNITED-STATES, ?LOCATION2=EASTERN-US
  #8: ?LOCATION1=SOUTHERN-US, ?LOCATION2=DALLAS
  #9: ?LOCATION1=SOUTHERN-US, ?LOCATION2=AUSTIN
  #10: ?LOCATION1=UNITED-STATES, ?LOCATION2=DALLAS
  #11: ?LOCATION1=UNITED-STATES, ?LOCATION2=TEXAS
  #12: ?LOCATION1=UNITED-STATES, ?LOCATION2=AUSTIN
  #13: ?LOCATION1=EASTERN-US, ?LOCATION2=ATLANTA
  #14: ?LOCATION1=UNITED-STATES, ?LOCATION2=GEORGIA
  #15: ?LOCATION1=UNITED-STATES, ?LOCATION2=ATLANTA
BUSINESS |=
```

Here we have fifteen solutions even though there are only seven "contains" relations asserted in the business.plm file – the other eight solutions were inferred. In addition to the "retrieve" function that finds solutions matching a query you can also use the "ask" function to determine if a specified relation is true; for example:

```
BUSINESS |= (ask (contains UNITED-STATES DALLAS))
TRUE
BUSINESS |=
```

For complex queries you can use the "why" function to see how PowerLoom solved the last query:

```
BUSINESS |= (ask (contains southern-us dallas))
TRUE
BUSINESS |= (why)
1 (CONTAINS ?location1 ?location2)
    follows by Modus Ponens
    with substitution {?l1/SOUTHERN-US, ?l3/DALLAS,
                       ?l2/TEXAS}
    since 1.1 ! (FORALL (?l1 ?l3)
                   (<= (CONTAINS ?l1 ?l3)
                       (EXISTS (?l2)
                          (AND (CONTAINS ?l1 ?l2)
                               (CONTAINS ?l2 ?l3)))))
    and   1.2 ! (CONTAINS SOUTHERN-US TEXAS)
    and   1.3 ! (CONTAINS TEXAS DALLAS)
BUSINESS |=
```

By default the explanation facility is turned off because it causes PowerLoom to run more slowly; it was turned on in the file business.plm using the statement:

```
(set-feature justifications)
```

3.4 Using the PowerLoom APIs in Java Programs

Once you interactively develop concepts, rules and relations then it is likely that you may want to use them with PowerLoom in an embedded mode, making PowerLoom a part of your application. I will get you started with a few Java example programs. The source code for this chapter is in the subdirectory src-powerloom-reasoning.

If you download the PowerLoom manual (a PDF file) from the PowerLoom web site, you will have the complete Java API documentation for the Java version of PowerLoom (there are also C++ and Common Lisp versions with separate documentation). I have found that I usually use just a small subset of the Java PowerLoom APIs and I have "wrapped" this subset in a wrapper class in the file PowerLoomUtils.java. We will use my wrapper class for the examples in the rest of this chapter.

My wrapper class has the follow public methods:

- PowerLoomUtils() – constructor initializes the Java PowerLoom runtime system.
- load(String fpath) – load a source *.plm file.
- changeModule(String workingModule) – set the current PowerLoom working module ("PL-USER" is the default module).
- assertProposition(String proposition) – asserts a new proposition; for example: "(and (company c3) (company-name c3 \"Moms Grocery\"))". Note that quotation marks are escaped with a backslash character. You can also use single quote characters like: "(and (company c3) (company-name c3 'Moms Grocery'))" because I convert single quotes in my wrapper code.
- createRelation(String relation, int arity) – create a new relation with a specified arity (number of "arguments"). For example you could create a relation "owns" with arity 2 and then assert "(owns Elaine 'Moms Grocery')" – I usually do not use this API since I prefer to place relations (with rules) in a source code file ending in the extention *.plm.
- doQuery(String query) – returns a list of results from a query. Each result in the list is itself a list.

You will always want to work in an interactive PowerLoom console for writing and debugging PowerLoom models. I built the model in test.plm (in the subdirectory test_data) interactively and we will use it here in an embedded Java example:

```
PowerLoomUtils plu = new PowerLoomUtils();
plu.load("test_data/test.plm");
plu.changeModule("BUSINESS");
plu.assertProposition(
     "(and (company c1)" +
     "     (company-name c1 \"Moms Grocery\"))");
plu.assertProposition(
     "(and (company c2)" +
      "     (company-name c2 \"IBM\"))");
plu.assertProposition(
     "(and (company c3)" +
     "     (company-name c3 \"Apple\"))");
List answers = plu.doQuery("all ?x (company ?x)");
System.out.println(answers);
// answers: [[C3], [C2], [C1]]
answers = plu.doQuery(
     "all (?x ?name)" +
     "    (and" +
     "      (company ?x)" +
```

```
     "      (company-name ?x ?name))");
System.out.println(answers);
// answers:
//   [[C3, "Apple"],
//    [C2, "IBM"],
//    [C1, "Moms Grocery"]]
plu.createRelation("CEO", 2);
plu.assertProposition(
     "(CEO \"Apple\" \"SteveJobs\")");
answers = plu.doQuery(
     "all (?x ?name ?ceo)" +
     "    (and" +
     "      (company-name ?x ?name)" +
     "      (CEO ?name ?ceo))");
System.out.println(answers);
// answers: [[C3, "Apple", "SteveJobs"]]
```

I have added the program output produced by printing the value of the list variable "answers" as comments after each System.out.println call. In the wrapper API calls that take a string argument, I broke long strings over several lines for formatting to the width of a page; you would not do this in your own programs because of the cost of the extra string concatenation.

We will not look at the implementation of the $PowerLoomUtils$ class – you can read the code if you are interested. That said, I will make a few commments on the Java PowerLoom APIs. The class PLI contains static methods for initializing the system, loading PowerLoom source files. Here are a few examples:

```
PLI.initialize();
PLI.load("test.plm", null);
PLI.sChangeModule("BUSINESS", null);
```

3.5 Suggestions for Further Study

This chapter has provided a brief introduction to PowerLoom, one of my favorite AI tools. I also showed you how to go about embedding the PowerLoom knowledge representation and reasoning systems in your Java programs. Assuming that you see benefit to further study I recommend reading through the PowerLoom manual and the presentations (PDF files) on the PowerLoom web site. As you read through this material it is best to have an interactive PowerLoom session open to try the examples as you read them.

Knowledge Representation and Logic are huge subjects and I will close out this chapter by recommending a few books that have been the most helpful to me:

- *Knowledge Representation* by John Sowa. This has always been my favorite reference for knowledge representation, logic, and ontologies.
- *Artificial Intelligence, A Modern Approach* by Stuart Russell and Peter Norvig. A very good theoretical treatment of logic and knowledge representation.
- *The Art of Prolog* by Leon Sterling and Ehud Shapiro. Prolog implements a form of predicate logic that is less expressive than the descriptive logics supported by PowerLoom and OWL (Chapter 4). That said, Prolog is very efficient and fairly easy to learn and so is sometimes a better choice. This book is one of my favorite general Prolog references.

The Prolog language is a powerful AI development tool. Both the open source SWI-Prolog and the commercial Amzi Prolog systems have good Java interfaces. I don't cover Prolog in this book but there are several very good tutorials on the web if you decide to experiment with Prolog.

We will continue Chapter 4 with our study of logic-based reasoning systems in the context of the Semantic Web.

4 Semantic Web

The Semantic Web is intended to provide a massive linked set of data for use by software systems just as the World Wide Web provides a massive collection of linked web pages for human reading and browsing. The Semantic Web is like the web in that anyone can generate any content that they want. This freedom to publish anything works for the web because we use our ability to understand natural language to interpret what we read – and often to dismiss material that based upon our own knowledge we consider to be incorrect.

The core concept for the Semantic Web is data integration and use from different sources. As we will soon see, the tools for implementing the Semantic Web are designed for encoding data and sharing data from many different sources.

There are several very good Semantic Web toolkits for the Java language and platform. I will use Sesame because it is what I often use in my own work and I believe that it is a good starting technology for your first experiments with Semantic Web technologies. This chapter provides an incomplete coverage of Semantic Web technologies and is intended merely as a gentle introduction to a few useful techniques and how to implement those techniques in Java.

Figure 4.1 shows a layered set of data models that are used to implement Semantic Web applications. To design and implement these applications we need to think in terms of physical models (storage and access of RDF, RDFS, and perhaps OWL data), logical models (how we use RDF and RDFS to define relationships between data represented as unique URIs and string literals and how we logically combine data from different sources) and conceptual modeling (higher level knowledge representation using OWL).

I am currently writing a separate book *Practical Semantic Web Programming in Java* that goes into much more detail on the use of Sesame, Jena, Protege, OwlApis, RDF/RDFS/OWL modeling, and Descriptive Logic Reasoners. This chapter is meant to get you interested in this technology but is not intended as a detailed guide.

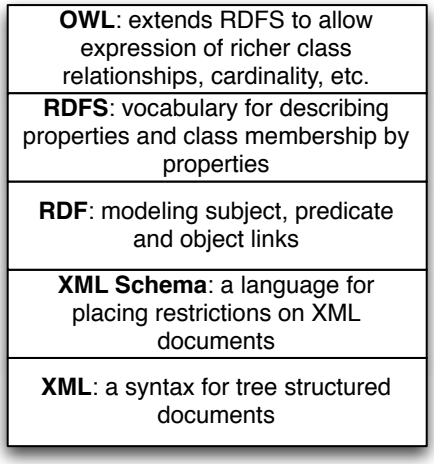

Figure 4.1: Layers of data models used in implementing Semantic Web applications

4.1 Relational Database Model Has Problems Dealing with Rapidly Changing Data Requirements

When people are first introduced to Semantic Web technologies their first reaction is often something like, "I can just do that with a database." The relational database model is an efficient way to express and work with slowly changing data models. There are some clever tools for dealing with data change requirements in the database world (ActiveRecord and migrations being a good example) but it is awkward to have end users and even developers tagging on new data attributes to relational database tables.

This same limitation also applies to object oriented programming and object modeling. Even with dynamic languages that facilitate modifying classes at runtime, the options for adding attributes to existing models is just too limiting. The same argument can be made against the use of XML constrained by conformance to either DTDs or XML Schemas. It is true that RDF and RDFS can be serialized to XML using many pre-existing XML namespaces for different knowledge sources and their schemas but it turns out that this is done in a way that does not reduce the flexibility for extending data models. XML storage is really only a serialization of RDF and many developers who are just starting to use Semantic Web technologies initially get confused trying to read XML serialization of RDF – almost like trying to read a PDF file with a plain text editor and something to be avoided.

A major goal for the rest of this chapter is convincing you that modeling data with RDF and RDFS facilitates freely extending data models and also allows fairly easy integration of data from different sources using different schemas without explicitly converting data from one schema to another for reuse.

4.2 RDF: The Universal Data Format

The Resource Description Framework (RDF) is used to encode information and the RDF Schema (RDFS) facilitates using data with different RDF encodings without the need to convert data formats.

RDF data was originally encoded as XML and intended for automated processing. In this chapter we will use two simple to read formats called ”N-Triples” and ”N3.” Sesame can be used to convert between all RDF formats so we might as well use formats that are easier to read and understand. RDF data consists of a set of triple values:

- subject
- predicate
- object

Some of my work with Semantic Web technologies deals with processing news stories, extracting semantic information from the text, and storing it in RDF. I will use this application domain for the examples in this chapter. I deal with triples like:

- subject: a URL (or URI) of a news article
- predicate: a relation like ”containsPerson”
- object: a value like ”Bill Clinton”

As previously mentioned, we will use either URIs or string literals as values for subjects and objects. We will always use URIs for the values of predicates. In any case URIs are usually preferred to string literals because they are unique. We will see an example of this preferred use but first we need to learn the N-Triple and N3 RDF formats.

In Section 4.1 I proposed the idea that RDF was more flexible than Object Modeling in programming languages, relational databases, and XML with schemas. If we can tag new attributes on the fly to existing data, how do we prevent what I might call “data chaos” as we modify existing data sources? It turns out that the solution to this problem is also the solution for encoding real semantics (or meaning) with data: we usually use unique URIs for RDF subjects, predicates, and objects, and usually

with a preference for not using string literals. I will try to make this idea more clear with some examples.

Any part of a triple (subject, predicate, or object) is either a URI or a string literal. URIs encode namespaces. For example, the containsPerson predicate in the last example could properly be written as:

```
http://knowledgebooks.com/ontology/#containsPerson
```

The first part of this URI is considered to be the namespace for (what we will use as a predicate) "containsPerson." When different RDF triples use this same predicate, this is some assurance to us that all users of this predicate subscribe to the same meaning. Furthermore, we will see in Section 4.3 we can use RDFS to state equivalency between this predicate (in the namespace http://knowledgebooks.com/ontology/) with predicates represented by different URIs used in other data sources. In an "artificial intelligence" sense, software that we write does not understand a predicate like "containsPerson" in the way that a human reader can by combining understood common meanings for the words "contains" and "person" but for many interesting and useful types of applications that is fine as long as the predicate is used consistently. We will see shortly that we can define abbreviation prefixes for namespaces which makes RDF and RDFS files shorter and easier to read.

A statement in N-Triple format consists of three URIs (or string literals – any combination) followed by a period to end the statement. While statements are often written one per line in a source file they can be broken across lines; it is the ending period which marks the end of a statement. The standard file extension for N-Triple format files is *.nt and the standard format for N3 format files is *.n3.

My preference is to use N-Triple format files as output from programs that I write to save data as RDF. I often use Sesame to convert N-Triple files to N3 if I will be reading them or even hand editing them. You will see why I prefer the N3 format when we look at an example:

```
@prefix kb:  <http://knowledgebooks.com/ontology#> .
<http://news.com/201234 /> kb:containsCountry "China"  .
```

Here we see the use of an abbreviation prefix "kb:" for the namespace for my company KnowledgeBooks.com ontologies. The first term in the RDF statement (the subject) is the URI of a news article. The second term (the predicate) is "containsCountry" in the "kb:" namespace. The last item in the statement (the object) is a string literal "China." I would describe this RDF statement in English as, "The news article at URI http://news.com/201234 mentions the country China."

This was a very simple N3 example which we will expand to show additional features of the N3 notation. As another example, suppose that this news article also

mentions the USA. Instead of adding a whole new statement like this:

```
@prefix kb:  <http://knowledgebooks.com/ontology#> .
<http://news.com/201234 /> kb:containsCountry "China"  .
<http://news.com/201234 /> kb:containsCountry "USA"  .
```

we can combine them using N3 notation. N3 allows us to collapse multiple RDF statements that share the same subject and optionally the same predicate:

```
@prefix kb:  <http://knowledgebooks.com/ontology#> .
<http://news.com/201234 /> kb:containsCountry "China" ,
                                              "USA" .
```

We can also add in additional predicates that use the same subject:

```
@prefix kb:  <http://knowledgebooks.com/ontology#> .

<http://news.com/201234 /> kb:containsCountry "China" ,
                                              "USA" .
    kb:containsOrganization "United Nations" ;
    kb:containsPerson "Ban Ki-moon" , "Gordon Brown" ,
                      "Hu Jintao" , "George W. Bush" ,
                      "Pervez Musharraf" ,
                      "Vladimir Putin" ,
                      "Mahmoud Ahmadinejad" .
```

This single N3 statement represents ten individual RDF triples. Each section defining triples with the same subject and predicate have objects separated by commas and ending with a period. Please note that whatever RDF storage system we use (we will be using Sesame) it makes no difference if we load RDF as XML, N-Triple, of N3 format files: internally subject, predicate, and object triples are stored in the same way and are used in the same way.

I promised you that the data in RDF data stores was easy to extend. As an example, let us assume that we have written software that is able to read online news articles and create RDF data that captures some of the semantics in the articles. If we extend our program to also recognize dates when the articles are published, we can simply reprocess articles and for each article add a triple to our RDF data store using a form like:

```
<http://news.com/201234 /> kb:datePublished
                           "2008-05-11" .
```

Furthermore, if we do not have dates for all news articles that is often acceptable depending on the application.

4.3 Extending RDF with RDF Schema

RDFS supports the definition of classes and properties based on set inclusion. In RDFS classes and properties are orthogonal. We will not simply be using properties to define data attributes for classes – this is different than object modeling and object oriented programming languages like Java. RDFS is encoded as RDF – the same syntax.

Because the Semantic Web is intended to be processed automatically by software systems it is encoded as RDF. There is a problem that must be solved in implementing and using the Semantic Web: everyone who publishes Semantic Web data is free to create their own RDF schemas for storing data; for example, there is usually no single standard RDF schema definition for topics like news stories and stock market data. Understanding the difficulty of integrating different data sources in different formats helps to understand the design decisions behind the Semantic Web.

We will start with an example that is an extension of the example in the last section that also uses RDFS. We add a few additional RDF statements (that are RDFS):

```
@prefix kb:  <http://knowledgebooks.com/ontology#> .
@prefix rdfs:  <http://www.w3.org/2000/01/rdf-schema#> .

kb:containsCity rdfs:subPropertyOf kb:containsPlace .
kb:containsCountry rdfs:subPropertyOf kb:containsPlace .
kb:containsState rdfs:subPropertyOf kb:containsPlace .
```

The last three lines declare that:

- The property containsCity is a subproperty of containsPlace.
- The property containsCountry is a subproperty of containsPlace.
- The property containsState is a subproperty of containsPlace.

Why is this useful? For at least two reasons:

- You can query an RDF data store for all triples that use property containsPlace and also match triples with property equal to containsCity, containsCountry, or containsState. There may not even be any triples that explicitly use the property containsPlace.

- Consider a hypothetical case where you are using two different RDF data stores that use different properties for naming cities: "cityName" and "city." You can define "cityName" to be a subproperty of "city" and then write all queries against the single property name "city." This removes the necessity to convert data from different sources to use the same Schema.

In addition to providing a vocabulary for describing properties and class membership by properties, RDFS is also used for logical inference to infer new triples, combine data from different RDF data sources, and to allow effective querying of RDF data stores. We will see examples of all of these features of RDFS when we start using the Sesame libraries in the next section to perform SPARQL queries.

4.4 The SPARQL Query Language

SPARQL is a query language used to query RDF data stores. While SPARQL may initially look like SQL, we will see that there are some important differences like support for RDFS and OWL inferencing (see Section 4.6) and graph-based instead of relational matching operations. We will cover the basics of SPARQL in this section and then see more examples in Section 4.5 when we learn how to embed Sesame in Java applications.

We will use the N3 format RDF file test_data/news.n3 for the examples in this section and in Section 4.5. This file was created automatically by spidering Reuters news stories on the news.yahoo.com web site and automatically extracting named entities from the text of the articles. We will see techniques for extracting named entities from text in Chapters 9 and 10. In this chapter we use these sample RDF files that I have created as input from another source.

You have already seen snippets of this file in Section 4.3 and I list the entire file here for reference (edited to fit line width: you may find the file news.n3 easier to read if you are at your computer and open the file in a text editor so you will not be limited to what fits on a book page):

```
@prefix kb:  <http://knowledgebooks.com/ontology#> .
@prefix rdfs:  <http://www.w3.org/2000/01/rdf-schema#> .

kb:containsCity rdfs:subPropertyOf kb:containsPlace .

kb:containsCountry rdfs:subPropertyOf kb:containsPlace .

kb:containsState rdfs:subPropertyOf kb:containsPlace .

<http://yahoo.com/20080616/usa_flooding_dc_16 />
```

```
    kb:containsCity "Burlington" , "Denver" ,
                    "St. Paul" ," Chicago" ,
                    "Quincy" , "CHICAGO" ,
                    "Iowa City" ;
    kb:containsRegion "U.S. Midwest" , "Midwest" ;
    kb:containsCountry "United States" , "Japan" ;
    kb:containsState "Minnesota" , "Illinois" ,
                     "Mississippi" , "Iowa" ;
    kb:containsOrganization "National Guard" ,
                     "U.S. Department of Agriculture" ,
                     "White House" ,
                     "Chicago Board of Trade" ,
                     "Department of Transportation" ;
    kb:containsPerson "Dena Gray-Fisher" ,
                      "Donald Miller" ,
                      "Glenn Hollander" ,
                      "Rich Feltes" ,
                      "George W. Bush" ;
    kb:containsIndustryTerm "food inflation" , "food" ,
                            "finance ministers" ,
                            "oil" .

<http://yahoo.com/78325/ts_nm/usa_politics_dc_2 />
    kb:containsCity "Washington" , "Baghdad" ,
                    "Arlington" , "Flint" ;
    kb:containsCountry "United States" ,
                       "Afghanistan" ,
                       "Iraq" ;
    kb:containsState "Illinois" , "Virginia" ,
                     "Arizona" , "Michigan" ;
    kb:containsOrganization "White House" ,
                            "Obama administration" ,
                            "Iraqi government" ;
    kb:containsPerson "David Petraeus" ,
                      "John McCain" ,
                      "Hoshiyar Zebari" ,
                      "Barack Obama" ,
                      "George W. Bush" ,
                      "Carly Fiorina" ;
    kb:containsIndustryTerm "oil prices" .

<http://yahoo.com/10944/ts_nm/worldleaders_dc_1 />
    kb:containsCity "WASHINGTON" ;
    kb:containsCountry "United States" , "Pakistan" ,
                       "Islamic Republic of Iran" ;
```

```
    kb:containsState "Maryland" ;
    kb:containsOrganization "University of Maryland" ,
                            "United Nations" ;
    kb:containsPerson "Ban Ki-moon" , "Gordon Brown" ,
                      "Hu Jintao" , "George W. Bush" ,
                      "Pervez Musharraf" ,
                      "Vladimir Putin" ,
                      "Steven Kull" ,
                      "Mahmoud Ahmadinejad" .

<http://yahoo.com/10622/global_economy_dc_4 />
    kb:containsCity "Sao Paulo" , "Kuala Lumpur" ;
    kb:containsRegion "Midwest" ;
    kb:containsCountry "United States" , "Britain" ,
                       "Saudi Arabia" , "Spain" ,
                       "Italy" , India" ,
                       ""France" , "Canada" ,
                       "Russia" , "Germany" , "China" ,
                       "Japan" , "South Korea" ;
    kb:containsOrganization "Federal Reserve Bank" ,
                            "European Union" ,
                            "European Central Bank" ,
                            "European Commission" ;
    kb:containsPerson "Lee Myung-bak" , "Rajat Nag" ,
                      "Luiz Inacio Lula da Silva" ,
                      "Jeffrey Lacker" ;
    kb:containsCompany "Development Bank Managing" ,
                       "Reuters" ,
                       "Richmond Federal Reserve Bank" ;
    kb:containsIndustryTerm "central bank" , "food" ,
                            "energy costs" ,
                            "finance ministers" ,
                            "crude oil prices" ,
                            "oil prices" ,
                            "oil shock" ,
                            "food prices" ,
                            "Finance ministers" ,
                            "Oil prices" , "oil" .
```

In the following examples, we will look at queries but not the results. Please be patient: these same queries are used in the embedded Java examples in the next section so it makes sense to only list the query return values in one place. Besides that, you will enjoy running the example programs yourself and experiment with modifying the queries.

We will start with a simple SPARQL query for subjects (news article URLs) and objects (matching countries) with the value for the predicate equal to $containsCountry$:

```
SELECT ?subject ?object
  WHERE {
    ?subject
    http://knowledgebooks.com/ontology#containsCountry>
    ?object .
  }
```

Variables in queries start with a question mark character and can have any names. We can make this query easier and reduce the chance of misspelling errors by using a namespace prefix:

```
PREFIX kb:  <http://knowledgebooks.com/ontology#>
SELECT ?subject ?object
  WHERE {
      ?subject kb:containsCountry ?object .
  }
```

We could have filtered on any other predicate, for instance $containsPlace$. Here is another example using a match against a string literal to find all articles exactly matching the text "Maryland." The following queries were copied from Java source files and were embedded as string literals so you will see quotation marks backslash escaped in these examples. If you were entering these queries into a query form you would not escape the quotation marks.

```
PREFIX kb: <http://knowledgebooks.com/ontology#>
   SELECT ?subject
   WHERE { ?subject kb:containsState \"Maryland\" . }
```

We can also match partial string literals against regular expressions:

```
PREFIX kb:
SELECT ?subject ?object
   WHERE {
     ?subject
     kb:containsOrganization
     ?object FILTER regex(?object, \"University\") .
   }
```

Prior to this last example query we only requested that the query return values for subject and predicate for triples that matched the query. However, we might want to return all triples whose subject (in this case a news article URI) is in one of the matched triples. Note that there are two matching triples, each terminated with a period:

```
PREFIX kb: <http://knowledgebooks.com/ontology#>
SELECT ?subject ?a_predicate ?an_object
   WHERE {
    ?subject
    kb:containsOrganization
    ?object FILTER regex(?object, \"University\") .

    ?subject ?a_predicate ?an_object .
   }
   DISTINCT
   ORDER BY ?a_predicate  ?an_object
   LIMIT 10
   OFFSET 5
```

When WHERE clauses contain more than one triple pattern to match, this is equivalent to a Boolean "and" operation. The DISTINCT clause removes duplicate results. The ORDER BY clause sorts the output in alphabetical order: in this case first by predicate (containsCity, containsCountry, etc.) and then by object. The LIMIT modifier limits the number of results returned and the OFFSET modifier sets the number of matching results to skip.

We are done with our quick tutorial on using the SELECT query form. There are three other query forms that I am not covering in this chapter:

- CONSTRUCT – returns a new RDF graph of query results
- ASK – returns Boolean true or false indicating if a query matches any triples
- DESCRIBE – returns a new RDF graph containing matched resources

4.5 Using Sesame

Sesame is a complete Java library for developing RDF/RDFS applications and we will use it in this chapter. Currently Sesame's support for OWL (see Section 4.6) is limited. Other Java libraries I have used that more fully support OWL are Jena, OWLAPI, and the Protege library.

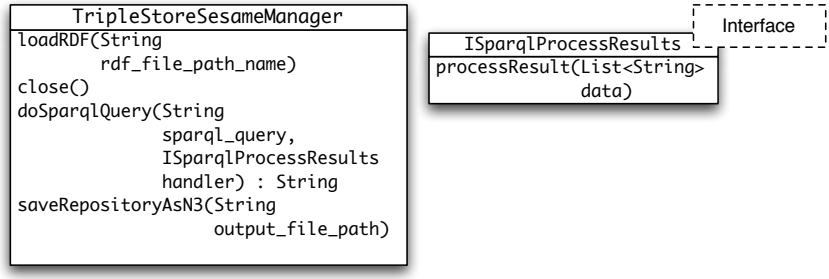

Figure 4.2: Java utility classes and interface for using Sesame

Figure 4.2 shows a UML diagram for the wrapper classes and interface that I wrote for Sesame to make it easier for you to get started. My wrapper uses an in-memory RDF repository that supports inference, loading RDF/RDFS/OWL files, and performing queries. If you decide to use Semantic Web technologies in your development you will eventually want to use the full Sesame APIs for programatically creating new RDF triples, finer control of the type of repository (options are in-memory, disk based, and database) and inferencing, and programatically using query results. That said, using my wrapper library is a good place for you to start to start experimenting.

The class constructor $TripleStoreSesameManager$ opens a new in-memory RDF triple store. I will not cover the internal implementation of the classes and interface seen in Figure 4.2 but you can read the source code in the subdirectory src-semantic-web.

We will look in some detail at an example program that uses Sesame and my wrapper library for Sesame. The source code for this example is in the file ExampleSparqlQueries.java. This example class implements the $ISparqlProcessResults$ interface:

```
public class ExampleSparqlQueries
       implements ISparqlProcessResults {
```

and does this by defining the method:

```
    public void processResult(List<String> data) {
       System.out.print("next result: ");
       for (String s : data)
           System.out.print("|"+s+"|" + "\t  ");
       System.out.println(" . ");
    }
```

that simply prints out the subject, predicate, and object of each matched triple. The class $TripleStoreSesameManager$ method

```
public String doSparqlQuery(String sparql_query,
                            ISparqlProcessResults
                            handler) {
```

calls a defined $processResult$ method once for each triple that matches a query. The $ExampleSparqlQueries$ class makes several SPARQL queries and prints the results. These queries are the example queries from the last section. Here is an example query with the program output:

```
TripleStoreSesameManager ts =
        new TripleStoreSesameManager();
ts.loadRDF("test_data/news.n3");
sparql_query =
  "PREFIX kb: <http://knowledgebooks.com/ontology#>" +
  "SELECT ?subject "+
  "WHERE { ?subject kb:containsState \"Maryland\" . }";
ts.doSparqlQuery(sparql_query, this);
```

Here is the single line of output (Sesame debug printout is not shown and the long line is split into two lines to fit the page width):

```
next result: |http://news.yahoo.com/s/nm/ \\
              20080616/ts_nm/worldleaders_trust_dc_1 /|
```

Other queries in the last section return two or three values per result; this example only returns the subject (article URL). You can run the text program implemented in the class $ExampleSparqlQueries$ to see all of the query results for the examples in the last section.

There is a lot more to RDFS than what I have covered so far in this chapter but I believe you have a sufficient introduction in order to use the example programs to experiment with using RDF and RDFS to define data and use Sesame in an imbedded mode in your java applications.

4.6 OWL: The Web Ontology Language

We have already seen a few examples of using RDFS to define sub-properties in the this chapter. The Web Ontology Language (OWL) extends the expressive power of

RDFS. We will not cover OWL programming examples in this book but this section will provide some background material. Sesame version 2.1 included in the ZIP file for this book does not support OWL DL reasoning "out of the box." When I need to use OWL DL reasoning in Java applications I use one or more of the following:

- ProtegeOwlApis – compatible with the Protege Ontology editor
- Pellet – DL reasoner
- Owlim – OWL DL reasoner compatible with some versions of Sesame
- Jena – General purpose library
- OWLAPI – a simpler API using many other libraries

OWL is more expressive than RDFS in that it supports cardinality, richer class relationships, and Descriptive Logic (DL) reasoning. OWL treats the idea of classes very differently than object oriented programming languages like Java and Smalltalk, but similar to the way PowerLoom (Chapter 3) uses concepts (PowerLoom's rough equivalent to a class). In OWL instances of a class are referred to as individuals and class membership is determined by a set of properties that allow a DL reasoner to infer class membership of an individual (this is called entailment.)

We saw an example of expressing transitive relationships when we were using PowerLoom in Section 3.3 where we defined a PowerLoom rule to express that the relation "contains" is transitive. We will now look at a similar example using OWL.

We have been using the RDF file news.n3 in previous examples and we will layer new examples by adding new triples that represent RDF, RDFS, and OWL. We saw in news.n3 the definition of three triples using rdfs:subPropertyOf properties to create a more general kb:containsPlace property:

```
kb:containsCity rdfs:subPropertyOf kb:containsPlace .
kb:containsCountry rdfs:subPropertyOf kb:containsPlace .
kb:containsState rdfs:subPropertyOf kb:containsPlace .

kb:containsPlace rdf:type owl:transitiveProperty .

kbplace:UnitedStates kb:containsState kbplace:Illinois .
kbplace:Illinois kb:containsCity kbplace:Chicago .
```

We can also infer that:

```
kbplace:UnitedStates kb:containsPlace kbplace:Chicago .
```

We can also model inverse properties in OWL. For example, here we add an inverse property kb:containedIn, adding it to the example in the last listing:

```
kb:containedIn owl:inverseOf kb:containsPlace .
```

Given an RDF container that supported extended OWL DL SPARQL queries, we can now execute SPARQL queries matching the property kb:containedIn and "match" triples in the RDF triple store that have never been asserted.

OWL DL is a very large subject and OWL is an even larger subject. From reading Chapter 3 and the very light coverage of OWL in this section, you should understand the concept of class membership not by explicitly stating that an object (or individual) is a member of a class, but rather because an individual has properties that can be used to infer class membership.

The World Wide Web Consortium has defined three versions of the OWL language that are in increasing order of complexity: OWL Lite, OWL DL, and OWL Full. OWL DL (supports Description Logic) is the most widely used (and recommended) version of OWL. OWL Full is not computationally decidable since it supports full logic, multiple class inheritance, and other things that probably make it computationally intractable for all but small problems.

4.7 Knowledge Representation and REST

A theme in this book is representing knowledge using logic, expert system rules, relational databases (supporting at the physical model level conceptual models like entity relation), and in flexible data models like RDF and RDFS (supporting higher level conceptual models in OWL).

I want to make some comments about the REST architectural style and how it is complementary to distributed knowledge representation on the web. The REST model implies that resource providers have some internal model for the storage and maintenance of data but use a possibly different representation of their internal data model to transfer their internal data to clients.

I would argue that RDF is often a good representation for resource providers to use for transferring data in their internal data formats to REST clients because of its flexibility in describing both data and relations between data. RDF is inherently a rich notation because RDFS and OWL are themselves expressed as RDF data.

I expect that conventional data sources like relational databases and conventional data-rich web sites will benefit from publishing REST style interfaces using RDF as the external representation of data. We are already seeing interesting and useful

projects that utilize a data source to publish data as RDF embedded as RDFa (an XHTML notation for embedding RDF in XHTML web pages) and I see this as a growth area for publishing information resources that are useful for both humans and software agents.

4.8 Material for Further Study

Writing Semantic Web applications in Java is a very large topic, worthy of an entire book. I have covered in this chapter what for my work have been the most useful Semantic Web techniques: storing and querying RDF and RDFS for a specific application. We will see in Chapter 10 some useful techniques for gathering semantic information from the web. Specifically, in Section 10.1 I briefly talk about entering semantic data from the Open Calais system into a Sesame RDF repository.

I have already mentioned several Java libraries that support OWL Descriptive Logic reasoning in Section 4.6. When the expressive power of RDF and RDFS become insufficient for your application you will then need to use a library supporting the OWL language and OWL Description Logic reasoning. The combination of RDF and RDFS is sufficient for many applications and using this simpler technology is the right way to get started developing semantic web applications. Because RDF and RDFS (with very few OWL features, commonly referred to as RDFS-Plus) are easier to implement and have a smaller learning curve, I believe that the adoption of OWL DL will be slow.

I concentrated on using Sesame in an embedded mode in Java applications in this chapter but another common use is as an RDF repository web service. In either case, the basic ideas of converting data to RDF, storing RDF, and allowing SPARQL queries are the same.

5 Expert Systems

We will be using the Drools Java expert system language and libraries in this chapter. Earlier editions of this book used the Jess expert system tool but due to the new more restrictive Jess licensing terms I decided to switch to Drools because it is released under the Apache 2.0 license. The primary web site for Drools is www.jboss.org/drools where you can download the source code and documentation. Both Jess and Drools are forward chaining inference engines that use the Rete algorithm and are derived from Charles Forgy's OPS5 language. One thing to keep in mind whenever you are building a system based on the Rete algorithm is that Rete scales very well to large numbers of rules but scales at $O(N^2)$ where N is the number of facts in the system. I have a long history with OPS5, porting it to Xerox Lisp Machines (1982) and the Apple Macintosh (1984) as well as building custom versions supporting multiple "worlds" of data and rule spaces. One thing that I would like to make clear: Drools is the only technology that I am covering in this book that I have not used professionally. That said, I spent some effort getting up to speed on Drools as a replacement for Jess on future projects.

While there is some interest in using packages like Drools for "business rules" to capture business process knowledge, often as embedded components in large systems, expert systems have historically been built to approach human level expertise for very specific tasks like configuring computer systems and medical diagnosis. The examples in this chapter are very simple and are intended to show you how to embed Drools in your Java applications and to show you a few tricks for using forward chaining rule-based systems. Drools is a Domain Specific Language (DSL) that attempts to provide a syntax that is easier to use than a general purpose programming language.

I do not usually recommend Java IDEs (a personal choice!) but if you already use Eclipse then I suggest that you use the Drools plugins for Eclipse (the "Eclipse Drools Workbench") which help setting up projects and understand the Drools rule language syntax.

The Eclipse Drools Workbench can automatically generate a small demo which I will go over in some detail in the next two sections. I then design and implement two simple example rule systems for solving block world type problems and for answering help desk questions.

The material in this chapter exclusively covers forward chaining production systems

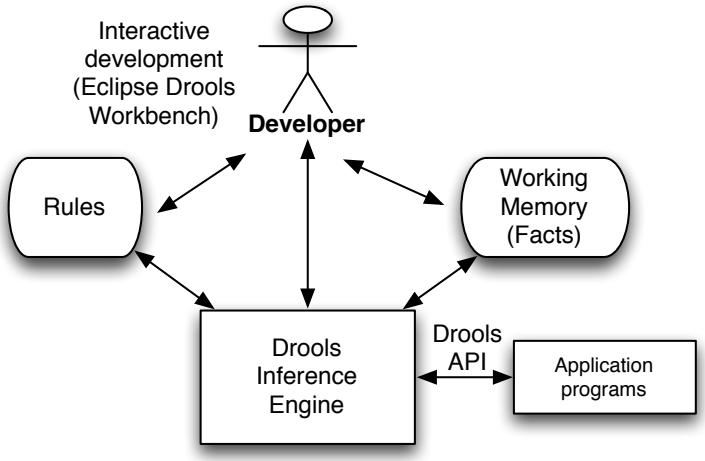

Figure 5.1: Using Drools for developing rule-based systems and then deploying them.

(also called "expert systems"). Forward chaining systems start with a set of known facts, and apply rules to work towards solving one or more goals. An alternative approach, often used in Prolog programs, is to use backward chaining. Backward chaining systems start with a final goal and attempt to work backwards towards currently known facts.

The phrase, expert systems, was almost synonymous with artificial intelligence in the early and mid 1980s. The application of expert system techniques to real problems, like configuring DEC VAX minicomputers, medical diagnosis, and evaluating seismic data for planning oil exploration had everyone very excited. Unfortunately, expert systems were over hyped and there was an eventual backlash that affected the entire field of AI. Still, the knowledge of how to write expert systems is a useful skill. This chapter contains a tutorial for using the Drools system and also shows how to use machine learning to help generate rules when training data is available.

As seen in Figure 5.1, Drools development is interactive: you will work in an environment where you can quickly add and change rules and re-run test cases. This interactive style of development is similar to using PowerLoom as we saw in Chapter 3.

5.1 Production Systems

I like to refer to expert systems by a more precise name: production systems. Productions are rules for transforming state. For example, given the three production rules:

```
a => b
b => c
c => d
```

then if a production system is initialized with the state a, the state d can be derived by applying these three production rules in order. The form of these production rules is:

```
<left-hand side>  => <right-hand side>
```

or:

```
LHS => RHS
```

Like the PowerLoom reasoning system used in Chapter 3, much of the power of a rule-based system comes from the ability to use variables so that the left-hand side (LHS) patterns can match a variety of known facts (called working memory in Drools). The values of these variables set in the LHS matching process are substituted for the variables on the right-hand side (RHS) patterns.

Production rule systems are much less expressive than Description Logic style reasoners like PowerLoom. The benefits of production rule systems is that they should have better runtime efficiency and are easier to use – a smaller learning curve. Good advice is to use production systems when they are adequate, and if not, use a more expressive knowledge representation and reasoning system like PowerLoom.

5.2 The Drools Rules Language

The basic syntax (leaving out optional components) of a Drools rule is:

```
rule "a name for the rule"
    when
        LHS
```

```
    then
        RHS
end
```

What might sample LHS and RHS statements look like? Drools rules reference POJOs ("Plain Old Java Objects") in both the LHS matching expressions and RHS actions. If you use the Eclipse Drools Workbench and create a new demo project, the Workbench will automatically create for you:

- Sample.drl – a sample rule file.
- com.sample.DroolsTest.java – defines: a simple Java POJO class $Message$ that is used in the Sample.drl rule file, a utility method for loading rules, and a main method that loads rules and creates an instance of the $Message$ class that "fires" the first rule in Sample.drl.

Even if you decide not to use the Eclipse Drools Workbench, I include these two auto-generated files in the ZIP file for this book and we will use these files to introduce both the syntax of rues and using rules and Drools in Java applications in the next section.

Here is the Sample.drl file:

```
package com.sample

import com.sample.DroolsTest.Message;

rule "Hello World"
  when
    m : Message(status == Message.HELLO,
                message : message)
  then
    System.out.println(message);
    m.setMessage("Goodbye cruel world");
    m.setStatus(Message.GOODBYE);
    update(m);
end

rule "GoodBye"
  no-loop true
  when
    m : Message(status == Message.GOODBYE,
                message : message)
  then
    System.out.println(message);
```

```
    m.setMessage(message);
end
```

This example rule file defines which Java package it has visibility in; we will see in the next section that the Java code that defines the POJO $Message$ class and code that uses these rules will be in the same Java package. This class has private data (with public accessor methods using Java Bean protocol) for attributes "status" and "message."

Another thing that might surprise you in this example is the direct calls to the static Java method $System.out.println$: this is a hint that Drools will end up compiling these rules into Java byte code. When Drools sees a reference to the class $Message$, since there are no Java import statements in this example rule file, the class $Message$ must be in the package com.sample.

On the LHS of both rules, any instance of class $Message$ that matches and thus allows the rule to "fire" sets a reference to the matched object to the local variable m that can then be used on the RHS of the rule. In the first rule, the attribute $message$ is also stored in a local variable (perhaps confusingly) also called $message$. Note that the public attribute accessor methods like $setMessage$ are used to change the state of a matched $Message$ object.

We will see later that the first step in writing a Drools based expert system is modeling (as Java classes) the data required to represent problem states. After you have defined these POJO classes you can then proceed with defining some initial test cases and start writing rules to handle the test cases. Drools development of non-trivial projects will involve an iterative process of adding new test cases, writing new rules or generalizing previously written rules, and making sure that both the original and newer test cases work.

There is a complete reference description of the Drools rule syntax on the Drools documentation wiki. The material in this chapter is tutorial in nature: new features of the rule language and how to use Drools will be introduced as needed for the examples.

5.3 Using Drools in Java Applications

We looked at the sample rules file Sample.drl in the last section which is generated automatically when creating a demo project with the Eclipse Drools Workbench. We will use the other generated file DroolsTest.java as an illustrative example in this section. The file DroolsTest.java is almost 100 lines long so I will list it in small fragments followed by an explanation of each code fragment. The first thing to note is that the Java client code is in the same package as the rules file:

```
package com.sample;

import java.io.InputStreamReader;
import java.io.Reader;

import org.drools.RuleBase;
import org.drools.RuleBaseFactory;
import org.drools.WorkingMemory;
import org.drools.compiler.PackageBuilder;
import org.drools.rule.Package;
```

This main function is an example showing how to use a rule package defined in a rule source file. We will see the definition of the utility method $readRule$ that opens a rule file and returns an instance of class $RuleBase$ shortly. After creating an instance of $RuleBase$ we create an instance of the $Message$ class and add it to open memory:

```
public class DroolsTest {

  public static final void main(String[] args) {
    try {
      RuleBase ruleBase = readRule();
      WorkingMemory workingMemory =
             ruleBase.newStatefulSession();
      Message message = new Message();
      message.setMessage(  "Hello World" );
      message.setStatus( Message.HELLO );
      workingMemory.insert( message );
      workingMemory.fireAllRules();
    } catch (Throwable t) {
      t.printStackTrace();
    }
  }
```

The main method creates a new rule base and working memory. Working memory is responsible for maintaining the "facts" in the system – in this case facts are Plain Old Java Objects (POJOs) that are maintained in a collection.

An instance of class $Message$ is created and its status is set to the constant value $Message.HELLO$. We saw in the last section how the first example rule has a condition that allows the rule to "fire" if there is any instance of class $Message$ that has its status attribute set to this value.

The method $fireAllRules$ will keep identifying rules that are eligible to fire, choosing a rule from this active set using algorithms we will discuss later, and then repeat-

ing this process until no more rules are eligible to fire. There are other $fireAllRules$ methods that have arguments for a maximum number of rules to fire and a filter to allow only some eligible rules to execute.

```
/**
 * Please note that this is the "low level" rule
 * assembly API.
 */
private static RuleBase readRule()
                        throws Exception {
  //read in the source
  Reader source =
     new InputStreamReader(
       DroolsTest.class.
               getResourceAsStream("/Sample.drl"));

  // optionally read in the DSL if you are using one:
  // Reader dsl =
  //    new InputStreamReader(
  //     DroolsTest.class.
  //            getResourceAsStream("/mylang.dsl"));
```

The method $readRule$ is a utility for reading and compiling a rule file that was generated automatically by the Eclipse Drools Workbench; in general your projects will have one or more rules files that you will load as in this example. In method $readRule$ we opened an input stream reader on the source code for the example Drools rule file Sample.drl. Drools has the ability to modify the rule syntax to create Domain Specific Languages (DSLs) that match your application or business domain. This can be very useful if you want domain experts to create rules since they can "use their own language." We will not cover custom DSLs in this chapter but the Drools documentation covers this in detail. Here is the rest of the definition of method $readRule$:

```
  // Use package builder to build up a rule package:
  PackageBuilder builder = new PackageBuilder();

  // This will parse and compile in one step:
  builder.addPackageFromDrl(source);

  // Use the following instead of above if you are
  // using a custom DSL:
  //builder.addPackageFromDrl( source, dsl );
```

```
    // get the compiled package (which is serializable)
    Package pkg = builder.getPackage();

    // add the package to a rulebase (deploy the
    // rule package).
    RuleBase ruleBase = RuleBaseFactory.newRuleBase();
    ruleBase.addPackage( pkg );
    return ruleBase;
  }
```

The $readRule$ utility method can be copied to new rule projects that are not created using the Eclipse Drools Workbench and modified as appropriate to get you started with the Java "boilerplate" required by Drools. This implementation uses Drools defaults, the most important being the "conflict resolution strategy" that defaults to first checking the most recently modified working memory POJO objects to see which rules can fire. This produces a depth first search behavior. We will modify the $readRule$ utility method later in Section 5.4 when we will need to change this default Drools reasoning behavior from depth first to breadth first search.

We will need a Plain Old Java Object (POJO) class to represent messages in the example rule set. This demo class was generated by the Eclipse Drools Workbench:

```
public static class Message {
    public static final int HELLO = 0;
    public static final int GOODBYE = 1;

    private String message;

    private int status;

    public String getMessage() {
      return this.message;
    }

    public void setMessage(String message) {
      this.message = message;
    }

    public int getStatus() {
      return this.status;
    }

    public void setStatus( int status ) {
      this.status = status;
```

```
    }
  }
}
```

You might want to review the example rules using this POJO Message class in Section 5.2. Here is the sample output from running this example code and rule set:

```
Hello World
Goodbye cruel world
```

A simple example, but it serves to introduce you the Drools rule syntax and required Java code. This is also a good example to understand because when you use the Eclipse Drools Workbench to create a new Drools rule project, it generates this example automatically as a template for you to modify and re-use.

In the next two sections I will develop two more complicated examples: solving blocks world problems and supporting a help desk system.

5.4 Example Drools Expert System: Blocks World

The example in this section solved simple "blocks world" problems; see Figures 5.2 through 5.5 for a very simple example problem.

I like this example because it introduces the topic of "conflict resolution" and (unfortunately) shows you that even solving simple problems with rule-based systems can be difficult. Because of the difficulty of developing and debugging rule-based systems, they are best for applications that have both high business value and offer an opportunity to encode business or application knowledge of experts in rules. So, the example in the next section is a more real-life example of good application of expert systems, but you will learn valuable techniques in this example. In the interest of intellectual honesty, I should say that general blocks world problems like the "Towers of Hanoi" problem and block world problems as the one in this section are usually easily solved using breadth-first search techniques.

The Java source code and Drools rule files for this example are in the files BlockWorld.drl and DroolsBockWorld.java.

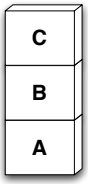

Figure 5.2: Initial state of a blocks world problem with three blocks stacked on top of each other. The goal is to move the blocks so that block C is on top of block A.

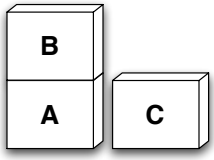

Figure 5.3: Block C has been removed from block B and placed on the table.

5.4.1 POJO Object Models for Blocks World Example

We will use the following three POJO classes (defined in the file DroolsBockWorld.java as static inner classes). The first POJO class *Block* represents the state of one block:

```
public static class Block {
  protected String name;
  protected String onTopOf;
  protected String supporting;

  public Block(String name, String onTopOf,
               String supporting) {
    this.name = name;
    this.onTopOf = onTopOf;
    this.supporting = supporting;
  }
  public String toString() {
    return "[Block_" + this.hashCode() + " " +
           name + "  on top of: " + onTopOf +
           "  supporting: " + supporting+"]";
  }
  public String getName() {
```

```
    return this.name;
  }
  public void setName(String name) {
    this.name = name;
  }

  public String getOnTopOf() {
    return this.onTopOf;
  }
  public void setOnTopOf(String onTopOf) {
    this.onTopOf = onTopOf;
  }

  public String getSupporting() {
    return this.supporting;
  }
  public void setSupporting(String supporting) {
    this.supporting = supporting;
  }
}
```

The next POJO class *OldBlockState* is used to represent previous states of blocks as they are being moved as the rules in this example "fire." We will later see rules that will not put a block into a state that it previously existed in:

```
public static class OldBlockState extends Block {
  public OldBlockState(String name,
                       String onTopOf,
                       String supporting) {
    super(name, onTopOf, supporting);
  }
  public String toString() {
    return "[OldBlockState_" + this.hashCode() +
           " " + name + "  on top of: " + onTopOf +
           "  supporting: " + supporting+"]";
  }
}
```

The next POJO class *Goal* is used to represent a goal state for the blocks that we are trying to reach:

```
public static class Goal {
```

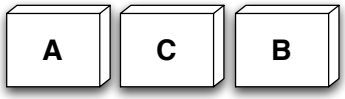

Figure 5.4: Block B has been removed from block A and placed on the table.

```
  private String supportingBlock;
  private String supportedBlock;
  public Goal(String supporting, String supported) {
    this.supportingBlock = supporting;
    this.supportedBlock = supported;
  }
  public String toString() {
    return "[Goal_" + this.hashCode() +
           " Goal: supporting block: " +
           supportingBlock +
           "  and supported block: " +
           supportedBlock +"]";
  }
  public void setSupportingBlock(
                  String supportingBlock) {
    this.supportingBlock = supportingBlock;
  }
  public String getSupportingBlock() {
    return supportingBlock;
  }
  public void setSupportedBlock(
                  String supportedBlock) {
    this.supportedBlock = supportedBlock;
  }
  public String getSupportedBlock() {
    return supportedBlock;
  }
}
```

Each block object has three string attributes: a name, the name of the block that this block is on top of, and the block that this block supports (is under). We will also define a block instance with the name “table.”

We need the POJO class OldBlockState that is a subclass of Block to avoid cycles in the reasoning process.

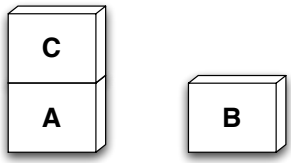

Figure 5.5: The goal is solved by placing block C on top of block A.

5.4.2 Drools Rules for Blocks World Example

We need four rules for this example and they are listed below with comments as appropriate:

```
package com.markwatson.examples.drool

import com.markwatson.examples.drool.DroolsBlockWorld
                                    .Goal;
import com.markwatson.examples.drool.DroolsBlockWorld
                                    .Block;
import com.markwatson.examples.drool.DroolsBlockWorld
                                    .OldBlockState;
```

We place the rules in the same Java package as the Java support code seen in the next section and the POJO model classes that we saw in the last section.

The first rule has no preconditions so it can always fire. We use the special rule condition "no-loop true" to let the Drools system know that we only want this rule to fire one time. This rule inserts facts into working memory for the simple problem seen in Figures 5.2 through 5.5:

```
rule "Startup Rule"
  no-loop true
  when
  then
    //insert(new Goal("C", "B"));  // test 1
    insert(new Goal("C", "A"));    // test 2
    // Block(String name, String onTopOf,
    //       String supporting)
    insert(new Block("A", "table", "B"));
    insert(new Block("B", "A", "C"));
    insert(new Block("C", "B", ""));
    insert(new Block("D", "", ""));
```

```
    insert(new Block("table", "", "A"));
end
```

The following rule looks for situations where it is possible to move a block with a few conditions:

- Find a block $block\_1$ that is on top of another block and is not itself supporting any other blocks
- Find a second block $block\_2$ that is not $block\_1$ and is not itself supporting any other blocks
- Find the block $on\_top\_of\_1$ that is under $block\_2$ and supporting $block\_1$
- Make sure that no previous block with the name in the variable $block\_2$ has already been on top of block $on\_top\_of\_2$ and supporting $block\_1$

If these conditions are met, we can remove the three matching facts and create facts for the new block positions and a new $OldBlockState$ fact in working memory. Note that the fourth LHS matching pattern is prefixed with "not" so this matches if there are no objects in working memory that match this pattern:

```
rule "Set Block On: move block_1 to block_2"
  when
    fact1 : Block(block_1 : name,
                  on_top_of_1 : onTopOf != "",
                  supporting == "")
    fact2 : Block(block_2 : name != block_1,
                  on_top_of_2 : onTopOf != "",
                  supporting == "")
    fact3 : Block(name == on_top_of_1,
                  on_top_of_3 : onTopOf,
                  supporting == block_1)
    not OldBlockState(name == block_2,
                      onTopOf == on_top_of_2,
                      supporting == block_1)
  then
    System.out.println( fact1 );
    System.out.println( fact2 );
    System.out.println( fact3 );
    retract(fact1);
    retract(fact2);
    retract(fact3);
    insert(new Block(block_1, block_2, ""));
    insert(new Block(block_2, on_top_of_2,
```

```
                    block_1));
    insert(new OldBlockState(block_2,
                          on_top_of_2, ""));
    insert(new Block(on_top_of_1,
                    on_top_of_3, ""));
    System.out.println("Moving " + block_1 +
                      " from " + on_top_of_1 +
                      " to " + block_2);
end
```

The next rule looks for opportunities to remove $block\_1$ from $block\_2$ if no other block is sitting on top of $block\_1$ (that is, $block\_1$ is clear):

```
rule "Clear Block: remove block_1 from block_2"
  when
    fact1 : Block(block_1 : name != "table",
                  on_top_of : onTopOf != "table",
                  supporting == "")
    fact2 : Block(block_2 : name,
                  on_top_of_2 : onTopOf,
                  supporting == block_1)
  then
    System.out.println( fact1 );
    System.out.println( fact2 );
    retract(fact1);
    retract(fact2);
    insert(new Block(block_1, "table", ""));
    insert(new Block(block_2, on_top_of_2, ""));
    insert(new Block("table", "", block_1));
    System.out.println("Clearing: remove " +
                      block_1 + " from " +
                      on_top_of + " to table");
end
```

The next rule checks to see if the current goal is satisfied in which case it halts the Drools engine:

```
rule "Halt on goal achieved"
  salience 500
  when
    Goal(b1 : supportingBlock, b2 : supportedBlock)
    Block(name == b1, supporting == b2)
  then
```

```
    System.out.println("Done!");
    drools.halt();
end
```

The Java code in the next section can load and run these example rules.

5.4.3 Java Code for Blocks World Example

The example in this section introduces something new: modifying the default way that Drools chooses which rules to "fire" (execute) when more than one rule is eligible to fire. This is referred to as the "conflict resolution strategy" and this phrase dates back to the original OPS5 production system. Drools by default prefers rules that are instantiated by data that is newer in working memory. This is similar to depth first search.

In the "blocks world" example in this section we will need to change the conflict resolution strategy to process rules in a first-in, first-out order which is similar to a breadth first search strategy.

First, let us define the problem that we want to solve. Consider labeled blocks sitting on a table as seen in Figures 5.2 through 5.5.

The Java code in this section is similar to what we already saw in Section 5.3 so we will just look at the differences here. To start with, in the utility method $readRule()$ we need to add a few lines of code to configure Drools to use a breadth-first instead of a depth-first reasoning strategy:

```
  private static RuleBase readRule() throws Exception {
    Reader source =
       new InputStreamReader(
        DroolsBlockWorld.class.getResourceAsStream(
                                    "/BlockWorld.drl"));
    PackageBuilder builder = new PackageBuilder();
    builder.addPackageFromDrl( source );
    Package pkg = builder.getPackage();

    // Change the default conflict resolution strategy:
    RuleBaseConfiguration rbc =
                            new RuleBaseConfiguration();
    rbc.setConflictResolver(new FifoConflictResolver());

    RuleBase ruleBase = RuleBaseFactory.newRuleBase(rbc);
    ruleBase.addPackage(pkg);
```

```
    return ruleBase;
  }
```

The Drools class $FifoConflictResolver$ is not so well named, but a first-in first-out (FIFO) strategy is like depth first search. The default conflict resolution strategy favors rules that are eligible to fire from data that has most recently changed.

Since we have already seen the definition of the Java POJO classes used in the rules in Section 5.4.1 the only remaining Java code to look at is in the static main method:

```
RuleBase ruleBase = readRule();
WorkingMemory workingMemory =
                ruleBase.newStatefulSession();
System.out.println("\nInitial Working Memory:\n\n" +
                  workingMemory.toString());
// Just fire the first setup rule:
workingMemory.fireAllRules(1);
Iterator<FactHandle> iter =
                 workingMemory.iterateFactHandles();
while (iter.hasNext()) {
  System.out.println(iter.next());
}
System.out.println("\n\n** Before firing rules...");
workingMemory.fireAllRules(20);  // limit 20 cycles
System.out.println("\n\n** After firing rules.");
System.out.println("\nFinal Working Memory:\n" +
                  workingMemory.toString());
iter = workingMemory.iterateFactHandles();
while (iter.hasNext()) {
  System.out.println(iter.next());
}
```

For making rule debugging easier I wanted to run the first "start up" rule to define the initial problem facts in working memory, and then print working memory. That is why I called $workingMemory.fireAllRules(1)$ to ask the Drools rule engine to just fire one rule. In the last example we called $workingMemory.fireAllRules()$ with no arguments so the rule engine runs forever as long as there are rules eligible to fire. After printing the facts in working memory I call the $fireAllRules(20)$ with a limit of 20 rule firings because blocks world problems can fail to terminate (at least the simple rules that I have written for this example often failed to terminate when I was debugging this example). Limiting the number of rule firings is often a good idea. The output from this example with debug output removed is:

```
Clearing: remove C from B to table
```

```
Moving B from A to C
Clearing: remove B from C to table
Moving A from table to C
Moving C from table to B
Done!
```

Note that this is not the best solution since it has unnecessary steps. If you are interested, here is the output with debug printout showing the facts that enabled each rule to fire:

```
[Block_11475926 C  on top of: B  supporting: ]
[Block_14268353 B  on top of: A  supporting: C]
Clearing: remove C from B to table
[Block_3739389 B  on top of: A  supporting: ]
[Block_15146334 C  on top of: table  supporting: ]
[Block_2968039 A  on top of: table  supporting: B]
Moving B from A to C
[Block_8039617 B  on top of: C  supporting: ]
[Block_14928573 C  on top of: table  supporting: B]
Clearing: remove B from C to table
[Block_15379373 A  on top of: table  supporting: ]
[OldBlockState_10920899 C  on top of: table  supporting: ]
[Block_4482425 table  on top of:   supporting: A]
Moving A from table to C
[Block_13646336 C  on top of: table  supporting: ]
[Block_11342677 B  on top of: table  supporting: ]
[Block_6615024 table  on top of:   supporting: C]
Moving C from table to B
Done!
```

This printout does not show the printout of all facts before and after running this example.

5.5 Example Drools Expert System: Help Desk System

Automating help desk functions can improve the quality of customer service and reduce costs. Help desk software can guide human call operators through canned explanations that can be thought of as decision trees; for example: "Customer reports that their refrigerator is not running –> Ask if the power is on and no circuit breakers are tripped. If customer reports that power source is OK –> Ask if the light

is on inside the refrigerator to determine if just the compressor motor is out...". We will see in Chapter 8 that decision trees can be learned from training data. One method of implementing a decision tree approach to help desk support would be to capture customer interactions by skilled support staff, factor operator responses into standard phrases and customer comments into standard questions, and use a machine learning package like Weka to learn the best paths through questions and answers.

We will take a different approach in our example for this section: we will assume that an expert customer service representative has provided us with use cases of common problems, what customers tend to ask for each problem, and the responses by customer service representatives. We will develop some sample Drools rules to encode this knowledge. This approach is likely to be more difficult to implement than a decision tree system but has the potential advantage that if individual rules "make sense" in general they may end up being useful in contexts beyond those anticipated by rules developers. With this greater flexibility comes a potential for less accuracy.

We will start in the next section by developing some POJO object models required for our example help desk expert system and then in the next section develop a few example rules.

5.5.1 Object Models for an Example Help Desk

We will use a single Java POJO class for this example. We want a problem type, a description of a problem, and a suggestion. A "real" help desk system might use additional classes for intermediate steps in diagnosing problems and offering advice but for this example, we will chain "problems" together. Here is an example:

```
Customer: My refrigerator is not running.
Service: I want to know if the power is on. Is the light
on inside the refrigerator?
Customer: No.
Service: Please check your circuit breaker, I will wait.
Customer: All my circuit breakers looked OK and
everything else is running in the kitchen.
Service I will schedule a service call for you.
```

We will not develop an interactive system; a dialog with a customer is assumed to be converted into facts in working memory. These facts will be represented by instances of the class $Problem$. The expert system will apply the rule base to the facts in working memory and make suggestions. Here is the Java class $Problem$ that is defined as an inner static class in the file DroolsHelpDesk.java:

```
public static class Problem {
  // Note: Drools has problems dealing with Java 5
  //       enums as match types so I use static
  //       integers here. In general, using enums
  //       is much better.
  final public static int NONE = 0;
  // appliance types:
  final public static int REFRIGERATOR = 101;
  final public static int MICROWAVE = 102;
  final public static int TV = 103;
  final public static int DVD = 104;
  // environmentalData possible values:
  final public static int CIRCUIT_BREAKER_OFF = 1002;
  final public static int LIGHTS_OFF_IN_ROOM = 1003;
  // problemType possible values:
  final public static int NOT_RUNNING = 2001;
  final public static int SMOKING = 2002;
  final public static int ON_FIRE = 2003;
  final public static int MAKES_NOISE = 2004;

  long serviceId = 0; // unique ID for all problems
                      // dealing with customer problem
  int applianceType = NONE;
  int problemType = NONE;
  int environmentalData = NONE;

  public Problem(long serviceId, int type) {
    this.serviceId = serviceId;
    this.applianceType = type;
  }

  public String toString() {
    return "[Problem: " + enumNames.get(applianceType) +
           " problem type: " + enumNames.get(problemType) +
           " environmental data: " +
           enumNames.get(environmentalData) + "]";
  }
  public long getServiceId() { return serviceId; }
  public int getEnvironmentalData() {
    return environmentalData;
  }
  public int getProblemType() {
    return problemType;
  }
  static Map<Integer, String> enumNames =
```

```
                                new HashMap<Integer, String>();
    static {
      enumNames.put(0, "NONE");
      enumNames.put(1002, "CIRCUIT_BREAKER_OFF");
      enumNames.put(1003, "LIGHTS_OFF_IN_ROOM");
      enumNames.put(2001, "NOT_RUNNING");
      enumNames.put(2002, "SMOKING");
      enumNames.put(2003, "ON_FIRE");
      enumNames.put(2004, "MAKES_NOISE");
      enumNames.put(101, "REFRIGERATOR");
      enumNames.put(102, "MICROWAVE");
      enumNames.put(103, "TV");
      enumNames.put(104, "DVD");
    }
  }
```

It is unfortunate that the current version of Drools does not work well with Java 5 enums – the $Problem$ class would have been about half as many lines of code (no need to map integers to meaningful descriptions for $toString()$) and the example would also be more type safe.

I used constant values like REFRIGERATOR and RUNNING to represent possible values for the member class attributes like $applianceType$, $problemType$, and $environmentalData$. There is obviously a tight binding from the Java POJO classes like $Problem$ to the rules that use these classes to represent objects in working memory. We will see a few example help desk rules in the next section.

5.5.2 Drools Rules for an Example Help Desk

This demo help desk system is not interactive. The Java code in the next section loads the rule set that we are about to develop and then programmatically adds test facts into working memory that simulate two help desk customer service issues. This is an important example since you will likely want to add data directly from Java code into Drools working memory.

There are several rules defined in the example file HelpDesk.drl and we will look at a few of them here. These rules are intended to be a pedantic example of both how to match attributes in Java POJO classes and to show a few more techniques for writing Drools rules.

I used to use the Lisp based OPS5 to develop expert systems and I find the combination of Java and Drools is certainly "less agile" to use. I found myself writing a rule, then editing the POJO class Problem to add constants for things that I wanted to use in the rule. With more experience, this less than interactive process might become

more comfortable for me.

As in the blocks world example, we want to place the rules file in the same package as the Java code using the rules file and import any POJO classes that we will use in working memory:

```
package com.markwatson.examples.drool
import com.markwatson.examples.drool.
                                DroolsHelpDesk.Problem;
```

The first rule sets a higher than default rule salience so it will fire before any rules with the default rule salience (a value of zero). This rule has a feature that we have not seen before: I have no matching expressions in the "when" clause. All Java Problem instances will match the left-hand side of this rule.

```
rule "Print all problems"
  salience 100
  when
    p : Problem()
  then
    System.out.println("From rule 'Print all problems': "
                       +  p);
end
```

The following rule matches an instance of the class $Problem$ in working memory that has a value of "Problem.CIRCUIT_BREAKER_OFF" for the vakue of attribute $environmentalData$. This constant has the integer value of 1002 but is is obviously more clear to use meaningful constant names:

```
rule "Reset circuit breaker"
  when
    p1 : Problem(environmentalData ==
                         Problem.CIRCUIT_BREAKER_OFF)
  then
    System.out.println("Reset circuit breaker: " + p1);
end
```

The last rule could perhaps be improved by having it only fire if any appliance was not currently running; we make this check in the next rule. Notice that in the next rule we are matching different attributes ($problemType$ and $environmentalData$) and it does not matter if these attributes match in a single working memory element or two different working memory elements:

```
rule "Check for reset circuit breaker"
  when
    p1 : Problem(problemType == Problem.NOT_RUNNING)
    Problem(environmentalData ==
            Problem.CIRCUIT_BREAKER_OFF)
  then
    System.out.println("Check for power source: " + p1 +
             ". The unit is not is not on and " +
             "the circuit breaker is tripped - check " +
             "the circuit breaker for this room.");
end
```

We will look at the Java code to use these example rules in the next section.

5.5.3 Java Code for an Example Help Desk

We will see another trick for using Drools in this example: creating working memory elements (i.e., instances of the Problem POJO class) in Java code instead of in a "startup rule" as we did for the blocks world example. The code in this section is also in the DroolsHelpDesk.java source file (as is the POJO class definition seen in Section 5.5.1).

The static main method in the $DroolsHelpDesk$ class is very similar to the main method in the blocks world example except that here we also call a new method $createTestFacts$:

```
  public static final void main(String[] args)
                            throws Exception {
    //load up the rulebase
    RuleBase ruleBase = readRule();
    WorkingMemory workingMemory =
       ruleBase.newStatefulSession();
    createTestFacts(workingMemory);

    .. same as the blocks world example ..
  }
```

We already looked at the utility method $readRule$ in Section 5.4.3 so we will just look at the new method $createTestFacts$ that creates two instance of the POJO class $Problem$ in working memory:

```
  private static void
```

```
      createTestFacts(WorkingMemory workingMemory)
         throws Exception {
    Problem p1 = new Problem(101, Problem.REFRIGERATOR);
    p1.problemType = Problem.NOT_RUNNING;
    p1.environmentalData = Problem.CIRCUIT_BREAKER_OFF;
    workingMemory.insert(p1);

    Problem p2 = new Problem(101, Problem.TV);
    p2.problemType = Problem.SMOKING;
    workingMemory.insert(p2);
  }
```

In this code we created new instances of the class $Problem$ and set desired attributes. We then use the $WorkingMemory$ method $insert$ to add the ojects to the working memory collection that Drools maintains. The output when running this example is (reformatted to fit the page width):

```
From rule 'Print all problems':
   [Problem: TV
               problem type: SMOKING
               environmental data: NONE]

From rule 'Print all problems':
   [Problem: REFRIGERATOR
               problem type: NOT_RUNNING
               environmental data: CIRCUIT_BREAKER_OFF]

Unplug appliance to prevent fire danger:
   [Problem: TV problem type: SMOKING
                         environmental data: NONE]

Check for power source:
   [Problem: REFRIGERATOR
               problem type: NOT_RUNNING
               environmental data: CIRCUIT_BREAKER_OFF]
   The unit is not is not on and the circuit breaker
   is tripped - check the circuit breaker for this room.
```

5.6 Notes on the Craft of Building Expert Systems

It may seem like rule-based expert systems have a lot of programming overhead; that is, it will seem excessively difficult to solve simple problems using production systems. However, for encoding large ill-structured problems, production systems provide a convenient notation for collecting together what would otherwise be too large a collection of unstructured data and heuristic rules (*Programming Expert Systems in Ops5: An Introduction to Rule-Based Programming*, Brownston et al. 1985). As a programming technique, writing rule-based expert systems is not for everyone. Some programmers find rule-based programming to be cumbersome, while others find it a good fit for solving some types of problems. I encourage the reader to have some fun experimenting with Drools, both with the examples in this chapter, and the many examples in the Drools distribution package and documentation.

Before starting a moderate or large expert system project, there are several steps that I recommend:

- Write a detailed description of the problem to be solved.
- Decide what structured data elements best describe the problem space.
- Try to break down the problem into separate modules of rules; if possible, try to develop and test these smaller modules independently, preferably one source file per module.
- Plan on writing specific rules that test parts of the system by initializing working memory for specific tests for the various modules; these tests will be very important when testing all of the modules together because tests that work correctly for a single module may fail when all modules are loaded due to unexpected rule interactions.

Production systems model fairly accurately the stimulus-response behavior in people. The left-hand side (LHS) terms represent environmental data that triggers a response or action represented by the right-hand side (RHS) terms in production rules. Simple stimulus-response types of production rules might be adequate for modeling simple behaviors, but our goal in writing expert systems is to encode deep knowledge and the ability to make complex decisions in a very narrow (or limited) problem domain. In order to model complex decision-making abilities, we also often need to add higher-level control functionality to expert systems. This higher level, or meta control, can be the control of which rule modules are active. We did not look at the Drools APIs for managing modules in this chapter but these APIs are covered in the Drools documentation. Hopefully, this chapter both gave you a quick-start for experimenting with Drools and enough experience to know if a rule-based system might be a good fit for your own development.

6 Genetic Algorithms

Genetic Algorithms (GAs) are computer simulations to evolve a population of chromosomes that contain at least some very fit individuals. Fitness is specified by a fitness function that rates each individual in the population.

Setting up a GA simulation is fairly easy: we need to represent (or encode) the state of a system in a chromosome that is usually implemented as a set of bits. GA is basically a search operation: searching for a good solution to a problem where the solution is a very fit chromosome. The programming technique of using GA is useful for AI systems that must adapt to changing conditions because "re-programming" can be as simple as defining a new fitness function and re-running the simulation. An advantage of GA is that the search process will not often "get stuck" in local minimum because the genetic crossover process produces radically different chromosomes in new generations while occasional mutations (flipping a random bit in a chromosome) cause small changes. Another aspect of GA is supporting the evolutionary concept of "survival of the fittest": by using the fitness function we will preferentially "breed" chromosomes with higher fitness values.

It is interesting to compare how GAs are trained with how we train neural networks (Chapter 7). We need to manually "supervise" the training process: for GAs we need to supply a fitness function and for the two neural network models used in Chapter 7 we need to supply training data with desired sample outputs for sample inputs.

6.1 Theory

GAs are typically used to search very large and possibly very high dimensional search spaces. If we want to find a solution as a single point in an N dimensional space where a fitness function has a near maximum value, then we have N parameters to encode in each chromosome. In this chapter we will be solving a simple problem that is one-dimensional so we only need to encode a single number (a floating point number for this example) in each chromosome. Using a GA toolkit, like the one developed in Section 6.2, requires two problem-specific customizations:

- Characterize the search space by a set of parameters that can be encoded in a chromosome (more on this later). GAs work with the coding of a parameter set, not the parameters themselves (*Genetic Algorithms in Search, Optimiza-*

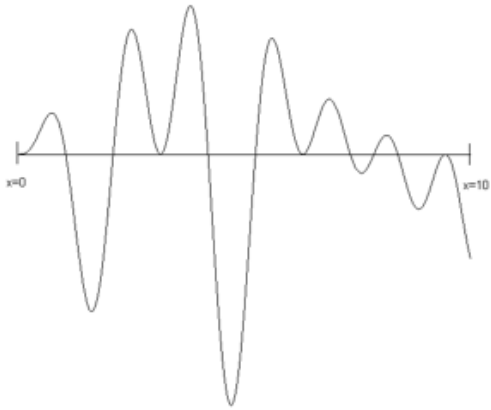

Figure 6.1: The test function evaluated over the interval [0.0, 10.0]. The maximum value of 0.56 occurs at x=3.8

tion, and Machine Learning, David Goldberg, 1989).

- Provide a numeric fitness function that allows us to rate the fitness of each chromosome in a population. We will use these fitness values to determine which chromosomes in the population are most likely to survive and reproduce using genetic crossover and mutation operations.

The GA toolkit developed in this chapter treats genes as a single bit; while you can consider a gene to be an arbitrary data structure, the approach of using single bit genes and specifying the number of genes (or bits) in a chromosome is very flexible. A population is a set of chromosomes. A generation is defined as one reproductive cycle of replacing some elements of the chromosome population with new chromosomes produced by using a genetic crossover operation followed by optionally mutating a few chromosomes in the population.

We will describe a simple example problem in this section, write a general purpose library in Section 6.2, and finish the chapter in Section 6.3 by solving the problem posed in this section.

For a sample problem, suppose that we want to find the maximum value of the function F with one independent variable x in Equation 6.1 and as seen in Figure 6.1:

$$F(x) = sin(x) * sin(0.4 * x) * sin(3 * x) \tag{6.1}$$

The problem that we want to solve is finding a good value of x to find a near to possible maximum value of $F(x)$. To be clear: we encode a floating point number

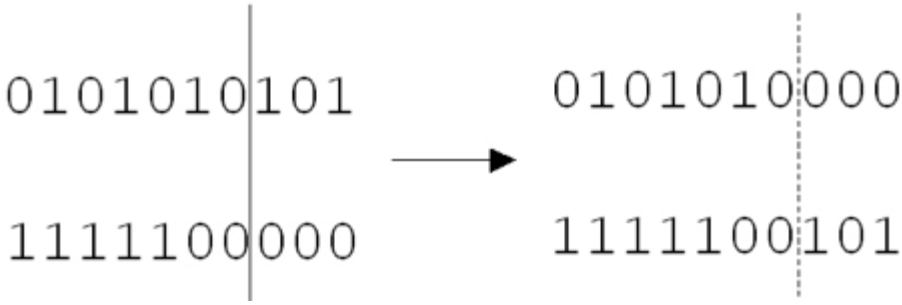

Figure 6.2: Crossover operation

as a chromosome made up of a specific number of bits so any chromosome with randomly set bits will represent some random number in the interval [0, 10]. The fitness function is simply the function in Equation 6.1.

Figure 6.2 shows an example of a crossover operation. A random chromosome bit index is chosen, and two chromosomes are "cut" at this this index and swap cut parts. The two original chromosomes in $generation_n$ are shown on the left of the figure and after the crossover operation they produce two new chromosomes in $generation_{n+1}$ shown on the right of the figure.

In addition to using crossover operations to create new chromosomes from existing chromosomes, we will also use genetic mutation: randomly flipping bits in chromosomes. A fitness function that rates the fitness value of each chromosome allows us to decide which chromosomes to discard and which to use for the next generation: we will use the most fit chromosomes in the population for producing the next generation using crossover and mutation.

We will implement a general purpose Java GA library in the next section and then solve the example problem posed in this section in Section 6.3.

6.2 Java Library for Genetic Algorithms

The full implementation of the GA library is in the Java source file Genetic.java. The following code snippets shows the method signatures defining the public API for the library; note that there are two constructors, the first using default values for the fraction of chromosomes on which to perform crossover and mutation operations and the second constructor allows setting explicit values for these parameters:

```
abstract public class Genetic {
  public Genetic(int num_genes_per_chromosome,
```

```
                int num_chromosomes)
  public Genetic(int num_genes_per_chromosome,
                int num_chromosomes,
                float crossover_fraction,
                float mutation_fraction)
```

The method *sort* is used to sort the population of chromosomes in most fit first order. The methods *getGene* and *setGene* are used to fetch and change the value of any gene (bit) in any chromosome. These methods are protected but you will probably not need to override them in derived classes.

```
  protected void sort()
  protected boolean getGene(int chromosome,
                            int gene)
  protected void setGene(int chromosome,
                         int gene, int value)
  protected void setGene(int chromosome,
                         int gene,
                         boolean value)
```

The methods *evolve*, *doCrossovers*, *doMutations*, and *doRemoveDuplicates* are utilities for running GA simulations. These methods are protected but you will probably not need to override them in derived classes.

```
  protected void evolve()
  protected void doCrossovers()
  protected void doMutations()
  protected void doRemoveDuplicates()
```

When you subclass class *Genetic* you must implement the following abstract method *calcFitness* that will determine the evolution of chromosomes during the GA simulation.

```
  // Implement the following method in sub-classes:
  abstract public void calcFitness();
}
```

The class *Chromosome* represents a bit set with a specified number of bits and a floating point fitness value.

```
class Chromosome {
  private Chromosome()
  public Chromosome(int num_genes)
  public boolean getBit(int index)
  public void setBit(int index, boolean value)
  public float getFitness()
  public void setFitness(float value)
  public boolean equals(Chromosome c)
}
```

The class $ChromosomeComparator$ implements a $Comparator$ interface and is application specific: it is used to sort a population in "best first" order:

```
class ChromosomeComparator
      implements Comparator<Chromosome> {
  public int compare(Chromosome o1,
                     Chromosome o2)
}
```

The last class $ChromosomeComparator$ is used when using the Java $Collection$ class static $sort$ method.

The class $Genetic$ is an abstract class: you must subclass it and implement the method $calcFitness$ that uses an application specific fitness function (that you must supply) to set a fitness value for each chromosome.

This GA library provides the following behavior:

- Generates an initial random population with a specified number of bits (or genes) per chromosome and a specified number of chromosomes in the population
- Ability to evaluate each chromosome based on a numeric fitness function
- Ability to create new chromosomes from the most fit chromosomes in the population using the genetic crossover and mutation operations

There are two class constructors for Genetic set up a new GA experiment by setting the number of genes (or bits) per chromosome, and the number of chromosomes in the population.

The $Genetic$ class constructors build an array of integers $rouletteWheel$ which is used to weight the most fit chromosomes in the population for choosing the parents

of crossover and mutation operations. When a chromosome is being chosen, a random integer is selected to be used as an index into the $rouletteWheel$ array; the values in the array are all integer indices into the chromosome array. More fit chromosomes are heavily weighted in favor of being chosen as parents for the crossover operations. The algorithm for the crossover operation is fairly simple; here is the implementation:

```
public void doCrossovers() {
  int num = (int)(numChromosomes * crossoverFraction);
  for (int i = num - 1; i >= 0; i--) {
    // Don't overwrite the "best" chromosome
    // from current generation:
    int c1 = 1 + (int) ((rouletteWheelSize - 1) *
                        Math.random() * 0.9999f);
    int c2 = 1 + (int) ((rouletteWheelSize - 1) *
                        Math.random() * 0.9999f);
    c1 = rouletteWheel[c1];
    c2 = rouletteWheel[c2];
    if (c1 != c2) {
      int locus = 1+(int)((numGenesPerChromosome-2) *
                          Math.random());
      for (int g = 0; g<numGenesPerChromosome; g++) {
        if (g < locus) {
          setGene(i, g, getGene(c1, g));
        } else {
          setGene(i, g, getGene(c2, g));
        }
      }
    }
  }
}
```

The method $doMutations$ is similar to $doCrossovers$: we randomly choose chromosomes from the population and for these selected chromosomes we randomly "flip" the value of one gene (a gene is a bit in our implementation):

```
public void doMutations() {
  int num = (int)(numChromosomes * mutationFraction);
  for (int i = 0; i < num; i++) {
    // Don't overwrite the "best" chromosome
    // from current generation:
    int c = 1 + (int) ((numChromosomes - 1) *
                       Math.random() * 0.99);
```

```
      int g = (int) (numGenesPerChromosome *
                     Math.random() * 0.99);
      setGene(c, g, !getGene(c, g));
    }
  }
```

We developed a general purpose library in this section for simulating populations of chromosomes that can evolve to a more "fit" population given a fitness function that ranks individual chromosomes in order of fitness. In Section 6.3 we will develop an example GA application by defining the size of a population and the fitness function defined by Equation 6.1.

6.3 Finding the Maximum Value of a Function

We will use the Java library in the last section to develop an example application to find the maximum of the function seen in Figure 6.1 which shows a plot of Equation 6.1 plotted in the interval [0, 10].

While we could find the maximum value of this function by using Newton's method (or even a simple brute force search over the range of the independent variable x), the GA method scales very well to similar problems of higher dimensionality. The GA also helps us to not find just locally optimum solutions. In this example we are working in one dimension so we only need to encode a single variable in a chromosome. As an example of a higher dimensional system, we might have products of sine waves using 20 independent variables $x1, x2, ..x20$. Still, the one-dimensional case seen in Figure 6.1 is a good example for showing you how to set up a GA simulation.

Our first task is to characterize the search space as one or more parameters. In general when we write GA applications we might need to encode several parameters in a single chromosome. For example, if a fitness function has three arguments we would encode three numbers in a singe chromosome. In this example problem, we have only one parameter, the independent variable x. We will encode the parameter x using ten bits (so we have ten 1-bit genes per chromosome). A good starting place is writing utility method for converting the 10-bit representation to a floating-point number in the range [0.0, 10.0]:

```
float geneToFloat(int chromosomeIndex) {
  int base = 1;
  float x = 0;
  for (int j=0; j<numGenesPerChromosome; j++)  {
     if (getGene(chromosomeIndex, j)) {
```

```
        x += base;
      }
      base *= 2;
    }
```

After summing up all on bits times their $base_2$ value, we need to normalize what is an integer in the range of [0,1023] to a floating point number in the approximate range of [0, 10]:

```
  x /= 102.4f;
  return x;
}
```

Note that we do not need the reverse method! We use the GA library from Section 6.2 to create a population of 10-bit chromosomes; in order to evaluate the fitness of each chromosome in a population, we only have to convert the 10-bit representation to a floating-point number for evaluation using the following fitness function (Equation 6.1):

```
private float fitness(float x) {
  return (float)(Math.sin(x) *
                 Math.sin(0.4f * x) *
                 Math.sin(3.0f * x));
}
```

Table 6.1 shows some sample random chromosomes and the floating point numbers that they encode. The first column shows the gene indices where the bit is "on," the second column shows the chromosomes as an integer number represented in binary notation, and the third column shows the floating point number that the chromosome encodes. The center column in Table 6.1 shows the bits in order where index 0 is the left-most bit, and index 9 if the right-most bit; this is the reverse of the normal order for encoding integers but the GA does not care: it works with any encoding we use. Once again, GAs work with encodings.

"On bits" in chromosome	As binary	Number encoded
2, 5, 7, 8, 9	0010010111	9.1015625
0, 1, 3, 5, 6	1101011000	1.0449219
0, 3, 5, 6, 7, 8	1001011110	4.7753906

Table 6.1: Random chromosomes and the floating point numbers that they encode

Using methods $geneToFloat$ and $fitness$ we now implement the abstract method $calcFitness$ from our GA library class $Genetic$ so the derived class $TestGenetic$

is not abstract. This method has the responsibility for calculating and setting the fitness value for every chromosome stored in an instance of class $Genetic$:

```
public void calcFitness() {
  for (int i=0; i<numChromosomes; i++) {
    float x = geneToFloat(i);
    chromosomes.get(i).setFitness(fitness(x));
  }
}
```

While it was useful to make this example more clear with a separate $geneToFloat$ method, it would have also been reasonable to simply place the formula in the method $fitness$ in the implementation of the abstract (in the base class) method $calcFitness$.

In any case we are done with coding this example: you can compile the two example Java files Genetic.java and TestGenetic.java, and run the $TestGenetic$ class to verify that the example program quickly finds a near maximum value for this function.

You can try setting different numbers of chromosomes in the population and try setting non-default crossover rates of 0.85 and a mutation rates of 0.3. We will look at a run with a small number of chromosomes in the population created with:

```
  genetic_experiment =
                 new MyGenetic(10, 20, 0.85f, 0.3f);
  int NUM_CYCLES = 500;
  for (int i=0; i<NUM_CYCLES; i++) {
    genetic_experiment.evolve();
    if ((i%(NUM_CYCLES/5))==0 || i==(NUM_CYCLES-1)) {
      System.out.println("Generation " + i);
      genetic_experiment.print();
    }
  }
```

In this experiment 85% of chromosomes will be "sliced and diced" with a crossover operation and 30% will have one of their genes changed. We specified 10 bits per chromosome and a population size of 20 chromosomes. In this example, I have run 500 evolutionary cycles. After you determine a fitness function to use, you will probably need to experiment with the size of the population and the crossover and mutation rates. Since the simulation uses random numbers (and is thus non-deterministic), you can get different results by simply rerunning the simulation. Here is example program output (with much of the output removed for brevity):

```
count of slots in roulette wheel=55
```

```
Generation 0
Fitness for chromosome 0 is 0.505, occurs at x=7.960
Fitness for chromosome 1 is 0.461, occurs at x=3.945
Fitness for chromosome 2 is 0.374, occurs at x=7.211
Fitness for chromosome 3 is 0.304, occurs at x=3.929
Fitness for chromosome 4 is 0.231, occurs at x=5.375
...
Fitness for chromosome 18 is -0.282 occurs at x=1.265
Fitness for chromosome 19 is -0.495, occurs at x=5.281
Average fitness=0.090 and best fitness for this
generation:0.505
...
Generation 499
Fitness for chromosome 0 is 0.561, occurs at x=3.812
Fitness for chromosome 1 is 0.559, occurs at x=3.703
...
```

This example is simple but is intended to be show you how to encode parameters for a problem where you want to search for values to maximize a fitness function that you specify. Using the library developed in this chapter you should be able to set up and run a GA simulation for your own applications.

7 Neural Networks

Neural networks can be used to efficiently solve many problems that are intractable or difficult using other AI programming techniques. I spent almost two years on a DARPA neural network tools advisory panel, wrote the first version of the ANSim neural network product, and have used neural networks for a wide range of application problems (radar interpretation, bomb detection, and as controllers in computer games). Mastering the use of simulated neural networks will allow you to solve many types of problems that are very difficult to solve using other methods.

Although most of this book is intended to provide practical advice (with some theoretical background) on using AI programming techniques, I cannot imagine being interested in practical AI programming without also wanting to think about the philosophy and mechanics of how the human mind works. I hope that my readers share this interest.

In this book, we have examined techniques for focused problem solving, concentrating on performing one task at a time. However, the physical structure and dynamics of the human brain is inherently parallel and distributed [*Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, Rumelhart, McClelland, etc. 1986]. We are experts at doing many things at once. For example, I simultaneously can walk, talk with my wife, keep our puppy out of cactus, and enjoy the scenery behind our house in Sedona, Arizona. AI software systems struggle to perform even narrowly defined tasks well, so how is it that we are able to simultaneously perform several complex tasks? There is no clear or certain answer to this question at this time, but certainly the distributed neural architecture of our brains is a requirement for our abilities. Unfortunately, artificial neural network simulations do not currently address "multi-tasking" (other techniques that do address this issue are multi-agent systems with some form or mediation between agents).

Also interesting is the distinction between instinctual behavior and learned behavior. Our knowledge of GAs from Chapter 6 provides a clue to how the brains of especially lower order animals can be hardwired to provide efficient instinctual behavior under the pressures of evolutionary forces (i.e., likely survival of more fit individuals). This works by using genetic algorithms to design specific neural wiring. I have used genetic algorithms to evolve recurrent neural networks for control applications. This work only had partial success but did convince me that biological genetic pressure is probably adequate to "pre-wire" some forms of behavior in natural (biological) neural networks.

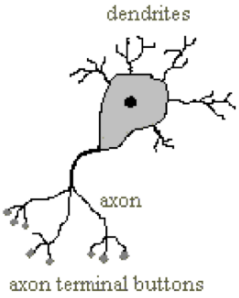

Figure 7.1: Physical structure of a neuron

While we will study supervised learning techniques in this chapter, it is possible to evolve both structure and attributes of neural networks using other types of neural network models like Adaptive Resonance Theory (ART) to autonomously learn to classify learning examples without intervention.

We will start this chapter by discussing human neuron cells and which features of real neurons that we will model. Unfortunately, we do not yet understand all of the biochemical processes that occur in neurons, but there are fairly accurate models available (web search "neuron biochemical"). Neurons are surrounded by thin hair-like structures called dendrites which serve to accept activation from other neurons. Neurons sum up activation from their dendrites and each neuron has a threshold value; if the activation summed over all incoming dendrites exceeds this threshold, then the neuron fires, spreading its activation to other neurons. Dendrites are very localized around a neuron. Output from a neuron is carried by an axon, which is thicker than dendrites and potentially much longer than dendrites in order to affect remote neurons. Figure 7.1 shows the physical structure of a neuron; in general, the neuron's axon would be much longer than is seen in Figure 7.1. The axon terminal buttons transfer activation to the dendrites of neurons that are close to the individual button. An individual neuron is connected to up to ten thousand other neurons in this way.

The activation absorbed through dendrites is summed together, but the firing of a neuron only occurs when a threshold is passed.

7.1 Hopfield Neural Networks

Hopfield neural networks implement associative (or content addressable) memory. A Hopfield network is trained using a set of patterns. After training, the network can be shown a pattern similar to one of the training inputs and it will hopefully associate the "noisy" pattern with the correct input pattern. Hopfield networks are

very different than back propagation networks (covered later in Section 7.4) because the training data only contains input examples unlike back propagation networks that are trained to associate desired output patterns with input patterns. Internally, the operation of Hopfield neural networks is very different than back propagation networks. We use Hopfield neural networks to introduce the subject of neural nets because they are very easy to simulate with a program, and they can also be very useful in practical applications.

The inputs to Hopfield networks can be any dimensionality. Hopfield networks are often shown as having a two-dimensional input field and are demonstrated recognizing characters, pictures of faces, etc. However, we will lose no generality by implementing a Hopfield neural network toolkit with one-dimensional inputs because a two-dimensional image can be represented by an equivalent one-dimensional array.

How do Hopfield networks work? A simple analogy will help. The trained connection weights in a neural network represent a high dimensional space. This space is folded and convoluted with local minima representing areas around training input patterns. For a moment, visualize this very high dimensional space as just being the three dimensional space inside a room. The floor of this room is a convoluted and curved surface. If you pick up a basketball and bounce it around the room, it will settle at a low point in this curved and convoluted floor. Now, consider that the space of input values is a two-dimensional grid a foot above the floor. For any new input, that is equivalent to a point defined in horizontal coordinates; if we drop our basketball from a position above an input grid point, the basketball will tend to roll down hill into local gravitational minima. The shape of the curved and convoluted floor is a calculated function of a set of training input vectors. After the "floor has been trained" with a set of input vectors, then the operation of dropping the basketball from an input grid point is equivalent to mapping a new input into the training example that is closest to this new input using a neural network.

A common technique in training and using neural networks is to add noise to training data and weights. In the basketball analogy, this is equivalent to "shaking the room" so that the basketball finds a good minima to settle into, and not a non-optimal local minima. We use this technique later when implementing back propagation networks. The weights of back propagation networks are also best visualized as defining a very high dimensional space with a manifold that is very convoluted near areas of local minima. These local minima are centered near the coordinates defined by each input vector.

7.2 Java Classes for Hopfield Neural Networks

The Hopfield neural network model is defined in the file Hopfield.java. Since this file only contains about 65 lines of code, we will look at the code and discuss the

algorithms for storing and recall of patterns at the same time. In a Hopfield neural network simulation, every neuron is connected to every other neuron.

Consider a pair of neurons indexed by i and j. There is a weight $W_{i,j}$ between these neurons that corresponds in the code to the array element $weight[i, j]$. We can define energy between the associations of these two neurons as:

$$energy[i, j] = -weight[i, j] * activation[i] * activation[j]$$

In the Hopfield neural network simulator, we store activations (i.e., the input values) as floating point numbers that get clamped in value to -1 (for off) or +1 (for on). In the energy equation, we consider an activation that is not clamped to a value of one to be zero. This energy is like "gravitational energy potential" using a basketball court analogy: think of a basketball court with an overlaid 2D grid, different grid cells on the floor are at different heights (representing energy levels) and as you throw a basketball on the court, the ball naturally bounces around and finally stops in a location near to the place you threw the ball, in a low grid cell in the floor – that is, it settles in a locally low energy level. Hopfield networks function in much the same way: when shown a pattern, the network attempts to settle in a local minimum energy point as defined by a previously seen training example.

When training a network with a new input, we are looking for a low energy point near the new input vector. The total energy is a sum of the above equation over all (i,j).

The class constructor allocates storage for input values, temporary storage, and a two-dimensional array to store weights:

```
public Hopfield(int numInputs) {
  this.numInputs = numInputs;
  weights = new float[numInputs][numInputs];
  inputCells = new float[numInputs];
  tempStorage = new float[numInputs];
}
```

Remember that this model is general purpose: multi-dimensional inputs can be converted to an equivalent one-dimensional array. The method $addTrainingData$ is used to store an input data array for later training. All input values get clamped to an "off" or "on" value by the utility method $adjustInput$. The utility method $truncate$ truncates floating-point values to an integer value. The utility method $deltaEnergy$ has one argument: an index into the input vector. The class variable $tempStorage$ is set during training to be the sum of a row of trained weights. So, the method $deltaEnergy$ returns a measure of the energy difference between the input vector in the current input cells and the training input examples:

```
private float deltaEnergy(int index) {
```

```
    float temp = 0.0f;
    for (int j=0; j<numInputs; j++) {
      temp += weights[index][j] * inputCells[j];
    }
    return 2.0f * temp - tempStorage[index];
  }
```

The method $train$ is used to set the two-dimensional weight array and the one-dimensional $tempStorage$ array in which each element is the sum of the corresponding row in the two-dimensional weight array:

```
  public void train() {
    for (int j=1; j<numInputs; j++) {
      for (int i=0; i<j; i++) {
        for (int n=0; n<trainingData.size(); n++) {
          float [] data =
              (float [])trainingData.elementAt(n);
          float temp1 =
            adjustInput(data[i]) * adjustInput(data[j]);
          float temp = truncate(temp1 +  weights[j][i]);
          weights[i][j] = weights[j][i] = temp;
        }
      }
    }
    for (int i=0; i<numInputs; i++) {
      tempStorage[i] = 0.0f;
      for (int j=0; j<i; j++) {
        tempStorage[i] += weights[i][j];
      }
    }
  }
```

Once the arrays $weight$ and $tempStorage$ are defined, it is simple to recall an original input pattern from a similar test pattern:

```
  public float [] recall(float [] pattern,
                         int numIterations) {
    for (int i=0; i<numInputs; i++) {
      inputCells[i] = pattern[i];
    }
    for (int ii = 0; ii<numIterations; ii++) {
      for (int i=0; i<numInputs; i++) {
        if (deltaEnergy(i) > 0.0f) {
```

```
          inputCells[i] = 1.0f;
        } else {
          inputCells[i] = 0.0f;
        }
      }
    }
    return inputCells;
  }
```

7.3 Testing the Hopfield Neural Network Class

The test program for the Hopfield neural network class is $Test\_Hopfield$. This test program defined three test input patterns, each with ten values:

```
static float [] data [] = {
  { 1,  1,  1, -1, -1, -1, -1, -1, -1, -1},
  {-1, -1, -1,  1,  1,  1, -1, -1, -1, -1},
  {-1, -1, -1, -1, -1, -1, -1,  1,  1,  1}
};
```

The following code fragment shows how to create a new instance of the $Hopfield$ class and train it to recognize these three test input patterns:

```
  test = new Hopfield(10);
  test.addTrainingData(data[0]);
  test.addTrainingData(data[1]);
  test.addTrainingData(data[2]);
  test.train();
```

The static method $helper$ is used to slightly scramble an input pattern, then test the training Hopfield neural network to see if the original pattern is re-created:

```
  helper(test, "pattern 0", data[0]);
  helper(test, "pattern 1", data[1]);
  helper(test, "pattern 2", data[2]);
```

The following listing shows an implementation of the method $helper$ (the called method pp simply formats a floating point number for printing by clamping it to zero or one). This version of the code randomly flips one test bit and we will see that the trained Hopfield network almost always correctly recognizes the original

pattern. The version of method *helper* included in the ZIP file for this book is slightly different in that two bits are randomly flipped (we will later look at sample output with both one and two bits randomly flipped).

```
private static void helper(Hopfield test, String s,
                           float [] test_data) {
  float [] dd = new float[10];
  for (int i=0; i<10; i++) {
    dd[i] = test_data[i];
  }
  int index = (int)(9.0f * (float)Math.random());
  if (dd[index] < 0.0f) dd[index] =  1.0f;
  else                  dd[index] = -1.0f;
  float [] rr = test.recall(dd, 5);
  System.out.print(s+"\nOriginal data:      ");
  for (int i = 0; i < 10; i++)
    System.out.print(pp(test_data[i]) + " ");
  System.out.print("\nRandomized data:    ");
  for (int i = 0; i < 10; i++)
    System.out.print(pp(dd[i]) + " ");
  System.out.print("\nRecognized pattern: ");
  for (int i = 0; i < 10; i++)
    System.out.print(pp(rr[i]) + " ");
  System.out.println();
}
```

The following listing shows how to run the program, and lists the example output:

```
java Test_Hopfield
pattern 0
Original data:      1 1 1 0 0 0 0 0 0 0
Randomized data:    1 1 1 0 0 0 1 0 0 0
Recognized pattern: 1 1 1 0 0 0 0 0 0 0
pattern 1
Original data:      0 0 0 1 1 1 0 0 0 0
Randomized data:    1 0 0 1 1 1 0 0 0 0
Recognized pattern: 0 0 0 1 1 1 0 0 0 0
pattern 2
Original data:      0 0 0 0 0 0 0 1 1 1
Randomized data:    0 0 0 1 0 0 0 1 1 1
Recognized pattern: 0 0 0 0 0 0 0 1 1 1
```

In this listing we see that the three sample training patterns in *Test_Hopfield.java* are re-created after scrambling the data by changing one randomly chosen value to

its opposite value. When you run the test program several times you will see occasional errors when one random bit is flipped and you will see errors occur more often with two bits flipped. Here is an example with two bits flipped per test: the first pattern is incorrectly reconstructed and the second and third patterns are reconstructed correctly:

```
pattern 0
Original data:      1 1 1 0 0 0 0 0 0 0
Randomized data:    0 1 1 0 1 0 0 0 0 0
Recognized pattern: 1 1 1 1 1 1 1 0 0 0
pattern 1
Original data:      0 0 0 1 1 1 0 0 0 0
Randomized data:    0 0 0 1 1 1 1 0 1 0
Recognized pattern: 0 0 0 1 1 1 0 0 0 0
pattern 2
Original data:      0 0 0 0 0 0 0 1 1 1
Randomized data:    0 0 0 0 0 0 1 1 0 1
Recognized pattern: 0 0 0 0 0 0 0 1 1 1
```

7.4 Back Propagation Neural Networks

The next neural network model that we will use is called back propagation, also known as back-prop and delta rule learning. In this model, neurons are organized into data structures that we call layers. Figure 7.2 shows a simple neural network with two layers; this network is shown in two different views: just the neurons organized as two one-dimensional arrays, and as two one-dimensional arrays with the connections between the neurons. In our model, there is a connection between two neurons that is characterized by a single floating-point number that we will call the connection's weight. A weight $W_{i,j}$ connects input neuron i to output neuron j. In the back propagation model, we always assume that a neuron is connected to every neuron in the previous layer.

The key thing is to be able to train a back-prop neural network. Training is performed by calculating sets of weights for connecting each layer. As we will see, we will train networks by applying input values to the input layer, allowing these values to propagate through the network using the current weight values, and calculating the errors between desired output values and the output values from propagation of input values through the network. Initially, weights are set to small random values. You will get a general idea for how this is done in this section and then we will look at Java implementation code in Section 7.5.

In Figure 7.2, we only have two neuron layers, one for the input neurons and one for the output neurons. Networks with no hidden layers are not usually useful – I

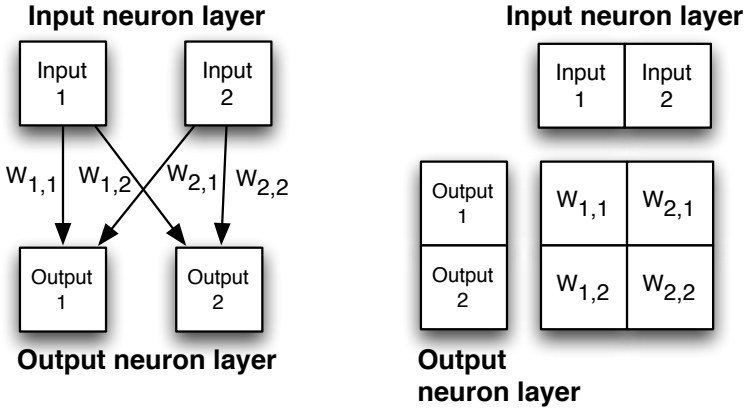

Figure 7.2: Two views of the same two-layer neural network; the view on the right shows the connection weights between the input and output layers as a two-dimensional array.

am using the network in Figure 7.2 just to demonstrate layer to layer connections through a weights array.

To calculate the activation of the first output neuron $O1$, we evaluate the sum of the products of the input neurons times the appropriate weight values; this sum is input to a $Sigmoid$ activation function (see Figure 7.3) and the result is the new activation value for $O1$. Here is the formula for the simple network in Figure 7.2:

$$O1 = Sigmoid(I1 * W[1,1] + I2 * W[2,1])$$

$$O2 = Sigmoid(I2 * W[1,2] + I2 * W[2,2])$$

Figure 7.3 shows a plot of the $Sigmoid$ function and the derivative of the sigmoid function ($SigmoidP$). We will use the derivative of the $Sigmoid$ function when training a neural network (with at least one hidden neuron layer) with classified data examples.

A neural network like the one seen in Figure 7.2 is trained by using a set of training data. For back propagation networks, training data consists of matched sets of input with matching desired output values. We want to train a network to not only produce the same outputs for training data inputs as appear in the training data, but also to generalize its pattern matching ability based on the training data to be able to match test patterns that are similar to training input patterns. A key here is to balance the size of the network against how much information it must hold. A common mistake when using back-prop networks is to use too large a network: a network that contains too many neurons and connections will simply memorize the training

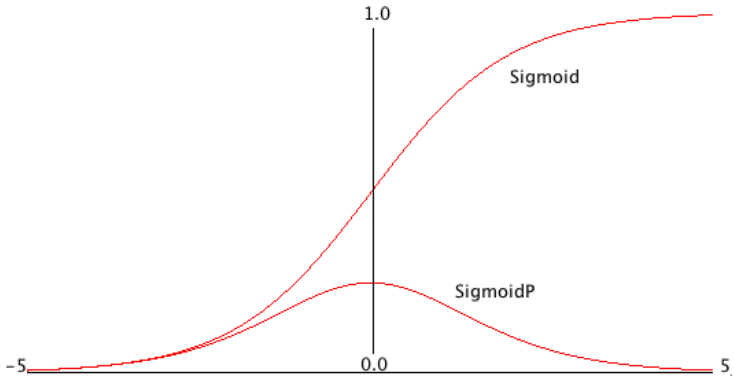

Figure 7.3: Sigmoid and derivative of the Sigmoid (SigmoidP) functions. This plot was produced by the file src-neural-networks/Graph.java.

examples, including any noise in the training data. However, if we use a smaller number of neurons with a very large number of training data examples, then we force the network to generalize, ignoring noise in the training data and learning to recognize important traits in input data while ignoring statistical noise.

How do we train a back propagation neural network given that we have a good training data set? The algorithm is quite easy; we will now walk through the simple case of a two-layer network like the one in Figure 7.2, and later in Section 7.5 we will review the algorithm in more detail when we have either one or two hidden neuron layers between the input and output layers.

In order to train the network in Figure 7.2, we repeat the following learning cycle several times:

1. Zero out temporary arrays for holding the error at each neuron. The error, starting at the output layer, is the difference between the output value for a specific output layer neuron and the calculated value from setting the input layer neuron's activation values to the input values in the current training example, and letting activation spread through the network.

2. Update the weight $W_{i,j}$ (where i is the index of an input neuron, and j is the index of an output neuron) using the formula $W_{i,j} += learning\_rate * output\_error_j * I_i$ ($learning\_rate$ is a tunable parameter) and $output\_error_j$ was calculated in step 1, and I_i is the activation of input neuron at index i.

This process is continued to either a maximum number of learning cycles or until the calculated output errors get very small. We will see later that the algorithm is similar but slightly more complicated, when we have hidden neuron layers; the difference is that we will "back propagate" output errors to the hidden layers in order to estimate errors for hidden neurons. We will cover more on this later. This type of neural

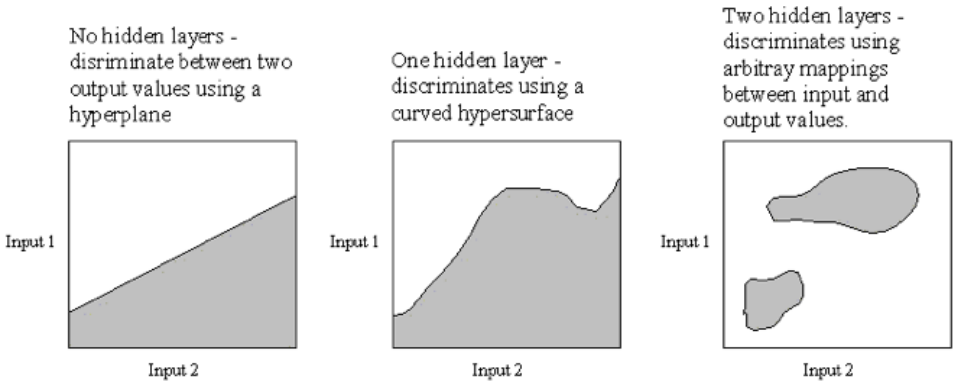

Figure 7.4: Capabilities of zero, one, and two hidden neuron layer neural networks. The grayed areas depict one of two possible output values based on two input neuron activation values. Note that this is a two-dimensional case for visualization purposes; if a network had ten input neurons instead of two, then these plots would have to be ten-dimensional instead of two-dimensional.

network is too simple to solve very many interesting problems, and in practical applications we almost always use either one additional hidden neuron layer or two additional hidden neuron layers. Figure 7.4 shows the types of problems that can be solved by zero hidden layer, one hidden layer, and two hidden layer networks.

7.5 A Java Class Library for Back Propagation

The back propagation neural network library used in this chapter was written to be easily understood and is useful for many problems. However, one thing that is not in the implementation in this section (it is added in Section 7.6) is something usually called "momentum" to speed up the training process at a cost of doubling the storage requirements for weights. Adding a "momentum" term not only makes learning faster but also increases the chances of sucessfully learning more difficult problems.

We will concentrate in this section on implementing a back-prop learning algorithm that works for both one and two hidden layer networks. As we saw in Figure 7.4 a network with two hidden layers is capable of arbitrary mappings of input to output values so there is no theoretical reason that I know of for using networks with three hidden layers.

The source directory src-neural-networks contains example programs for both back

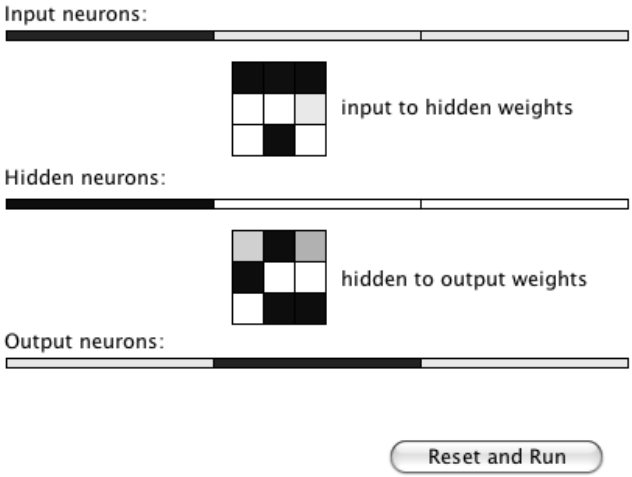

Figure 7.5: Example backpropagation neural network with one hidden layer.

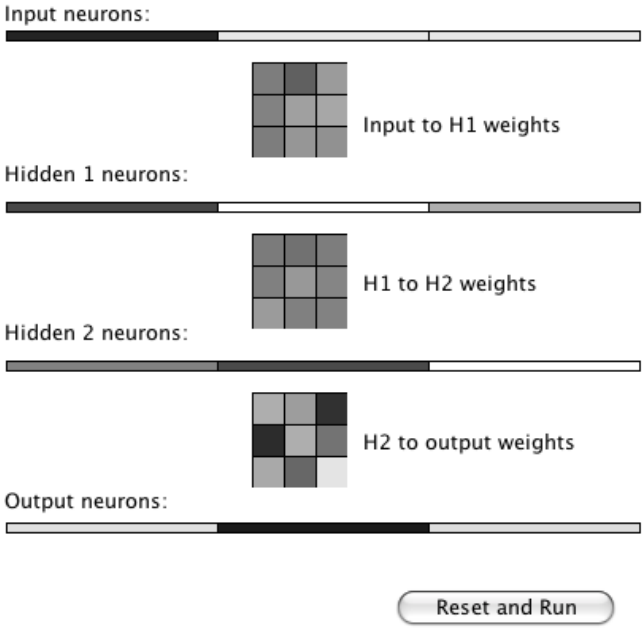

Figure 7.6: Example backpropagation neural network with two hidden layers.

propagation neural networks and Hopfield neural networks which we saw at the beginning of this chapter. The relevant files for the back propagation examples are:

- Neural_1H.java – contains a class for simulating a neural network with one hidden neuron layer
- Test_1H.java – a text-based test program for the class Neural_1H
- GUITest_1H.java – a GUI-based test program for the class Neural_1H
- Neural_2H.java – contains a class for simulating a neural network with two hidden neuron layers
- Neural_2H_momentum.java – contains a class for simulating a neural network with two hidden neuron layers and implements momentum learning (implemented in Section 7.6)
- Test_2H.java – a text-based test program for the class Neural_2H
- GUITest_2H.java – a GUI-based test program for the class Neural_2H
- GUITest_2H_momentum.java – a GUI-based test program for the class Neural_2H_momentum that uses momentum learning (implemented in Section 7.6)
- Plot1DPanel – a Java JFC graphics panel for the values of a one-dimensional array of floating point values
- Plot2DPanel – a Java JFC graphics panel for the values of a two-dimensional array of floating point values

The GUI files are for demonstration purposes only, and we will not discuss the code for these classes; if you are interested in the demo graphics code and do not know JFC Java programming, there are a few good JFC tutorials at the web site java.sun.com.

It is common to implement back-prop libraries to handle either zero, one, or two hidden layers in the same code base. At the risk of having to repeat similar code in two different classes, I decided to make the $Neural\_1H$ and $Neural\_2H$ classes distinct. I think that this makes the code a little easier for you to understand. As a practical point, you will almost always start solving a neural network problem using only one hidden layer and only progress to trying two hidden layers if you cannot train a one hidden layer network to solve the problem at-hand with sufficiently small error when tested with data that is different than the original training data. One hidden layer networks require less storage space and run faster in simulation than two hidden layer networks.

In this section we will only look at the implementation of the class $Neural\_2H$

(class $Neural\_1H$ is simpler and when you understand how $Neural\_2H$ works, the simpler class is easy to understand also). This class implements the $Serializable$ interface and contains a utility method $save$ to write a trained network to a disk file:

```
class Neural_2H implements Serializable {
```

There is a static factory method that reads a saved network file from disk and builds an instance of $Neural\_2H$ and there is a class constructor that builds a new untrained network in memory, given the number of neurons in each layer:

```
public static
       Neural_2H Factory(String serialized_file_name)
public Neural_2H(int num_in, int num_hidden1,
                 int num_hidden2,
                 int num_output)
```

An instance of $Neural\_2H$ contains training data as transient data that is not saved by method $save$.

```
    transient protected
              ArrayList inputTraining = new Vector();
    transient protected
              ArrayList outputTraining = new Vector();
```

I want the training examples to be native float arrays so I used generic $ArrayList$ containers. You will usually need to experiment with training parameters in order to solve difficult problems. The learning rate not only controls how large the weight corrections we make each learning cycle but this parameter also affects whether we can break out of local minimum. Other parameters that affect learning are the ranges of initial random weight values that are hardwired in the method $randomizeWeights()$ and the small random values that we add to weights during the training cycles; these values are set in in $slightlyRandomizeWeights()$. I usually only need to adjust the learning rate when training back-prop networks:

```
public float TRAINING_RATE = 0.5f;
```

I often decrease the learning rate during training – that is, I start with a large learning rate and gradually reduce it during training. The calculation for output neuron values given a set of inputs and the current weight values is simple. I placed the code for calculating a forward pass through the network in a separate method $forwardPass()$ because it is also used later in the method $training$:

```
public float[] recall(float[] in) {
  for (int i = 0; i < numInputs; i++)
    inputs[i] = in[i];
  forwardPass();
  float[] ret = new float[numOutputs];
  for (int i = 0; i < numOutputs; i++)
    ret[i] = outputs[i];
  return ret;
}

public void forwardPass() {
  for (int h = 0; h < numHidden1; h++) {
    hidden1[h] = 0.0f;
  }
  for (int h = 0; h < numHidden2; h++) {
    hidden2[h] = 0.0f;
  }
  for (int i = 0; i < numInputs; i++) {
    for (int h = 0; h < numHidden1; h++) {
      hidden1[h] +=
              inputs[i] * W1[i][h];
    }
  }
  for (int i = 0; i < numHidden1; i++) {
    for (int h = 0; h < numHidden2; h++) {
      hidden2[h] +=
              hidden1[i] * W2[i][h];
    }
  }
  for (int o = 0; o < numOutputs; o++)
    outputs[o] = 0.0f;
  for (int h = 0; h < numHidden2; h++) {
    for (int o = 0; o < numOutputs; o++) {
      outputs[o] +=
              sigmoid(hidden2[h]) * W3[h][o];
    }
  }
}
```

While the code for $recall$ and $forwardPass$ is almost trivial, the training code in method $train$ is more complex and we will go through it in some detail. Before we get to the code, I want to mention that there are two primary techniques for training back-prop networks. The technique that I use is to update the weight arrays after each individual training example. The other technique is to sum all output errors

over the entire training set (or part of the training set) and then calculate weight updates. In the following discussion, I am going to weave my comments on the code into the listing. The private member variable *current_example* is used to cycle through the training examples: one training example is processed each time that the *train* method is called:

```
private int current_example = 0;

public float train(ArrayList ins, ArrayList v_outs) {
```

Before starting a training cycle for one example, we zero out the arrays used to hold the output layer errors and the errors that are back propagated to the hidden layers. We also need to copy the training example input values and output values:

```
  int i, h, o;
  float error = 0.0f;
  int num_cases = ins.size();
  //for (int example=0; example<num_cases; example++) {
    // zero out error arrays:
      for (h = 0; h < numHidden1; h++)
        hidden1_errors[h] = 0.0f;
  for (h = 0; h < numHidden2; h++)
    hidden2_errors[h] = 0.0f;
  for (o = 0; o < numOutputs; o++)
    output_errors[o] = 0.0f;
  // copy the input values:
  for (i = 0; i < numInputs; i++) {
    inputs[i] = ((float[]) ins.get(current_example))[i];
  }
  // copy the output values:
  float[] outs = (float[]) v_outs.get(current_example);
```

We need to propagate the training example input values through the hidden layers to the output layers. We use the current values of the weights:

```
  forwardPass();
```

After propagating the input values to the output layer, we need to calculate the output error for each output neuron. This error is the difference between the desired output and the calculated output; this difference is multiplied by the value of the calculated

output neuron value that is first modified by the $Sigmoid$ function that we saw in Figure 7.3. The $Sigmoid$ function is to clamp the calculated output value to a reasonable range.

```
for (o = 0; o < numOutputs; o++) {
  output_errors[o] =
    (outs[o] -
        outputs[o])
        * sigmoidP(outputs[o]);
}
```

The errors for the neuron activation values in the second hidden layer (the hidden layer connected to the output layer) are estimated by summing for each hidden neuron its contribution to the errors of the output layer neurons. The thing to notice is that if the connection weight value between hidden neuron h and output neuron o is large, then hidden neuron h is contributing more to the error of output neuron o than other neurons with smaller connecting weight values:

```
for (h = 0; h < numHidden2; h++) {
  hidden2_errors[h] = 0.0f;
  for (o = 0; o < numOutputs; o++) {
    hidden2_errors[h] +=
      output_errors[o] * W3[h][o];
  }
}
```

We estimate the errors in activation energy for the first hidden layer neurons by using the estimated errors for the second hidden layers that we calculated in the last code snippet:

```
for (h = 0; h < numHidden1; h++) {
  hidden1_errors[h] = 0.0f;
  for (o = 0; o < numHidden2; o++) {
    hidden1_errors[h] +=
      hidden2_errors[o] * W2[h][o];
  }
}
```

After we have scaled estimates for the activation energy errors for both hidden layers we then want to scale the error estimates using the derivative of the sigmoid function's value of each hidden neuron's activation energy:

```
for (h = 0; h < numHidden2; h++) {
  hidden2_errors[h] =
    hidden2_errors[h] * sigmoidP(hidden2[h]);
}
for (h = 0; h < numHidden1; h++) {
  hidden1_errors[h] =
    hidden1_errors[h] * sigmoidP(hidden1[h]);
}
```

Now that we have estimates for the hidden layer neuron errors, we update the weights connecting to the output layer and each hidden layer by adding the product of the current learning rate, the estimated error of each weight's target neuron, and the value of the weight's source neuron:

```
// update the hidden2 to output weights:
  for (o = 0; o < numOutputs; o++) {
    for (h = 0; h < numHidden2; h++) {
      W3[h][o] +=
        TRAINING_RATE * output_errors[o] * hidden2[h];
      W3[h][o] = clampWeight(W3[h][o]);
    }
  }
  // update the hidden1 to hidden2 weights:
  for (o = 0; o < numHidden2; o++) {
    for (h = 0; h < numHidden1; h++) {
      W2[h][o] +=
        TRAINING_RATE * hidden2_errors[o] * hidden1[h];
      W2[h][o] = clampWeight(W2[h][o]);
    }
  }
  // update the input to hidden1 weights:
  for (h = 0; h < numHidden1; h++) {
    for (i = 0; i < numInputs; i++) {
      W1[i][h] +=
        TRAINING_RATE * hidden1_errors[h] * inputs[i];
      W1[i][h] = clampWeight(W1[i][h]);
    }
  }
  for (o = 0; o < numOutputs; o++) {
    error += Math.abs(outs[o] - outputs[o]);
  }
```

The last step in this code snippet was to calculate an average error over all output neurons for this training example. This is important so that we can track the training

status in real time. For very long running back-prop training experiments I like to be able to see this error graphed in real time to help decide when to stop a training run. This allows me to experiment with the learning rate initial value and see how fast it decays. The last thing that method $train$ needs to do is to update the training example counter so that the next example is used the next time that $train$ is called:

```
    current_example++;
    if (current_example >= num_cases)
      current_example = 0;
    return error;
  }
```

You can look at the implementation of the Swing GUI test class $GUTest\_2H$ to see how I decrease the training rate during training. I also monitor the summed error rate over all output neurons and occasionally randomize the weights if the network is not converging to a solution to the current problem.

7.6 Adding Momentum to Speed Up Back-Prop Training

We did not use a momentum term in the Java code in Section 7.5. For difficult to train problems, adding a momentum term can drastically reduce the training time at a cost of doubling the weight storage requirements. To implement momentum, we remember how much each weight was changed in the previous learning cycle and make the weight change larger if the current change in "direction" is the same as the last learning cycle. For example, if the change to weight $W_{i,j}$ had a large positive value in the last learning cycle and the calculated weight change for $W_{i,j}$ is also a large positive value in the current learning cycle, then make the current weight change even larger. Adding a "momentum" term not only makes learning faster but also increases the chances of sucessfully learning more difficult problems.

I modified two of the classes from Section 7.5 to use momentum:

- Neural_2H_momentum.java – training and recall for two hidden layer back-prop networks. The constructor has an extra argument "alpha" that is a scaling factor for how much of the previous cycle's weight change to add to the new calculated delta weight values.
- GUITest_2H_momentum.java – a GUI test application that tests the new class $Neural\_2H\_momentum$.

The code for class $Neural\_2H\_momentum$ is similar to the code for $Neural\_2H$ that we saw in the last section so here we will just look at the differences. The

class constructor now takes another parameter $alpha$ that determines how strong the momentum correction is when we modify weight values:

```
// momentum scaling term that is applied
// to last delta weight:
private float alpha = 0f;
```

While this $alpha$ term is used three times in the training code, it suffices to just look at one of these uses in detail. When we allocated the three weight arrays $W1$, $W2$, and $W3$ we also now allocate three additional arrays of corresponding same size: $W1\_last\_delta$, $W2\_last\_delta$, and $W3\_last\_delta$. These three new arrays are used to store the weight changes for use in the next training cycle. Here is the original code to update $W3$ from the last section:

```
W3[h][o] +=
  TRAINING_RATE * output_errors[o] * hidden2[h];
```

The following code snippet shows the additions required to use momentum:

```
W3[h][o] +=
  TRAINING_RATE * output_errors[o] * hidden2[h] +
  // apply the momentum term:
  alpha * W3_last_delta[h][o];
W3_last_delta[h][o] = TRAINING_RATE *
                      output_errors[o] *
                      hidden2[h];
```

I mentioned in the last section that there are two techniques for training back-prop networks: updating the weights after processing each training example or waiting to update weights until all training examples are processed. I always use the first method when I don't use momentum. In many cases it is best to use the second method when using momentum.

8 Machine Learning with Weka

Weka is a standard Java tool for performing both machine learning experiments and for embedding trained models in Java applications. I have used Weka since 1999 and it is usually my tool of choice on machine learning projects that are compatible with Weka's use of the GPL license. In addition to the material in this chapter you should visit the primary Weka web site www.cs.waikato.ac.nz/ml/weka for more examples and tutorials. Good online documentation can also be found at weka.sourceforge.net/wekadoc. Weka can be run both as a GUI application and for using a command line interface for running experiments. While the techniques of machine learning have many practical applications the example used in this chapter is simple and is mostly intended to show you the techniques for running Weka and techniques for embedding Weka in your Java applications. Full documentation of the many machine learning algorithms is outside the scope of this chapter.

In addition to data cleansing and preprocessing utilities (filters for data normalization, resampling, transformations, etc.) Weka supports most machine-learning techniques for automatically calculating classification systems. I have used the following Weka learning modules in my own work:

- Nave Bayes – uses Bayes's rule for probability of a hypothesis given evidence.
- Instance-based learner – stores all training examples and use.
- C4.5 – a learning scheme by J Ross Quinlan that calculates decision trees from training data. We will use the J48 algorithm in this chapter.

Weka can be used for both unsupervised and supervised learning. An example of unsupervised learning is processing a set of unlabeled data and automatically clustering the data into smaller sets containing similar items. We will use supervised learning as the example in this chapter: data on daily stock prices is labeled as buy, sell, or hold. We will use the J48 algorithm to automatically build a decision tree for deciding on how to process a stock, given its cost data. This example is simplistic and should not be used to actually trade stocks.

It is also possible to induce rules from training data that are equivalent to decision trees for the same training data. The learned model uses linear combinations of attribute values for classification.

We are going to use a simple example to learn how to use Weka interactively and

embedded in applications in the next two sections. Weka uses a data file format call ARFF. The following listing shows the sample ARFF input file that we will use in the next two sections:

```
@relation stock

@attribute percent_change_since_open real
@attribute percent_change_from_day_low real
@attribute percent_change_from_day_high real
@attribute action {buy, sell, hold}

@data
-0.2,0.1,-0.22,hold
-2.2,0.0,-2.5,sell
0.2,0.21,-0.01,buy
-0.22,0.12,-0.25,hold
-2.0,0.0,-2.1,sell
0.28,0.26,-0.04,buy
-0.12,0.08,-0.14,hold
-2.6,0.1,-2.6,sell
0.24,0.25,-0.03,buy
```

Here the concept of a relation is similar to a relation in PowerLoom as we saw in Chapter 3: a relation has a name and a list of attributes, each with an allowed data type. Here the relation name is "stock" and we have three attributes that have floating point (numerical) values and a fourth attribute that has an enumeration of discrete allowed values. The @data section defines data for initializing nine stock relations.

8.1 Using Weka's Interactive GUI Application

The Weka JAR file is included with the ZIP file for this book. To run the Weka GUI application, change directory to test_data and type:

```
java -cp ../lib -jar ../lib/weka.jar
```

Once you have loaded (and possibly browsed) the data as seen in Figure 8.1 you can then select the classifier tab, and using the "Choose" Classifier option, find J48 under the trees submenu, and click the "Start" button. The results can be seen in Figure 8.2.

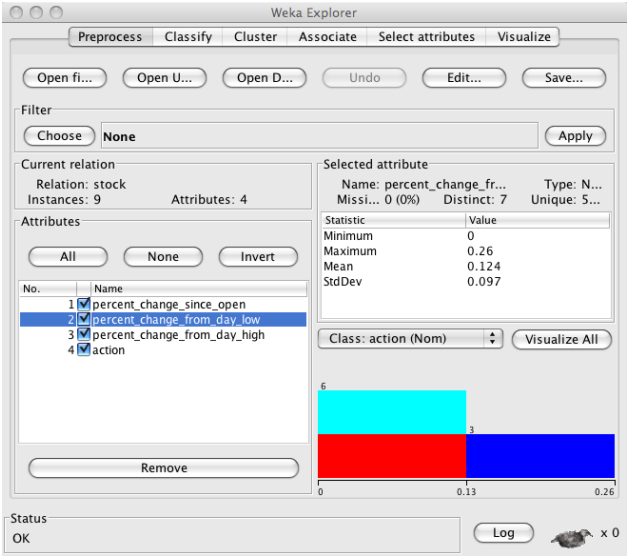

Figure 8.1: Running the Weka Data Explorer

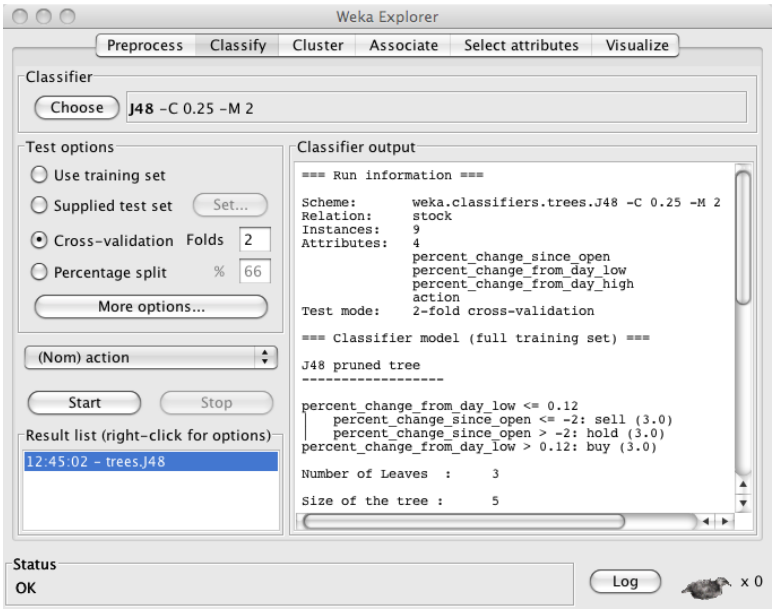

Figure 8.2: Running the Weka Data Explorer

The decision tree is displayed in the "Classifier output" window pane. We will run this same problem from the command line in the next section and then discuss the generated decision tree seen in the lower right panel of the GUI display seen in Figure 8.2.

8.2 Interactive Command Line Use of Weka

We will run the same problem as in the previous section and discuss the sections of the output report:

```
java -cp ../lib/weka.jar \\
     weka.classifiers.trees.J48 -t \\
     stock\_training_data.arff -x 2

J48 pruned tree
------------------

percent_change_from_day_low <= 0.12
|  percent_change_since_open <= -2: sell (3.0)
|  percent_change_since_open > -2: hold (3.0)
percent_change_from_day_low > 0.12: buy (3.0)

Number of Leaves  :  3

Size of the tree :  5
```

The generated decision tree can be described in English as "If the percent change of a stock from the day low is less than or equal to 0.12 then if the percent change since the open is less than -2 then sell the stock, otherwise keep it. If the percent change from the day low is greater than 0.12 then purchase more shares."

```
Time taken to build model: 0.01 seconds
Time taken to test model on training data: 0 seconds

=== Error on training data ===

Correctly Classified Instances    9     100  %
Incorrectly Classified Instances  0       0  %
Kappa statistic                   1
Mean absolute error               0
Root mean squared error           0
```

```
Relative absolute error           0  %
Root relative squared error       0  %
Total Number of Instances         9
```

This output shows results for testing on the original training data so the classification is perfect. In practice, you will test on separate data sets.

```
=== Confusion Matrix ===

 a b c   <-- classified as
 3 0 0 | a = buy
 0 3 0 | b = sell
 0 0 3 | c = hold
```

The confusion matrix shows the prediction (columns) for each data sample (rows). Here we see the original data (three buy, three sell, and three hold samples). The following output shows random sampling testing:

```
=== Stratified cross-validation ===

Correctly Classified Instances     4       44.4444 %
Incorrectly Classified Instances   5       55.5556 %
Kappa statistic                    0.1667
Mean absolute error                0.3457
Root mean squared error            0.4513
Relative absolute error           75.5299 %
Root relative squared error       92.2222 %
Total Number of Instances          9
```

With random sampling, we see in the confusion matrix that the three buy recommendations are still perfect, but that both of the sell recommendations are wrong (with one buy and two holds) and that two of what should have been hold recommendations are buy recommendations.

```
=== Confusion Matrix ===

 a b c   <-- classified as
 3 0 0 | a = buy
 1 0 2 | b = sell
 2 0 1 | c = hold
```

8.3 Embedding Weka in a Java Application

The example in this section is partially derived from documentation at the web site http://weka.sourceforge.net/wiki. This example loads the training ARFF data file seen at the beginning of this chapter and loads a similar ARFF file for testing that is equivalent to the original training file except that small random changes have been made to the numeric attribute values in all samples. A decision tree model is trained and tested on the new test ARFF data.

```
import weka.classifiers.meta.FilteredClassifier;
import weka.classifiers.trees.J48;
import weka.core.Instances;
import weka.filters.unsupervised.attribute.Remove;

import java.io.BufferedReader;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;

public class WekaStocks {

  public static void main(String[] args) throws Exception {
```

We start by creating a new training instance by supplying a reader for the stock training ARFF file and setting the number of attributes to use:

```
    Instances training_data = new Instances(
      new BufferedReader(
        new FileReader(
          "test_data/stock_training_data.arff")));
    training_data.setClassIndex(
      training_data.numAttributes() - 1);
```

We want to test with separate data so we open a separate examples ARFF file to test against:

```
    Instances testing_data = new Instances(
      new BufferedReader(
        new FileReader(
          "test_data/stock_testing_data.arff")));
    testing_data.setClassIndex(
      training_data.numAttributes() - 1);
```

The method *toSummaryString* prints a summary of a set of training or testing instances.

```
String summary = training_data.toSummaryString();
int number_samples = training_data.numInstances();
int number_attributes_per_sample =
  training_data.numAttributes();
System.out.println(
  "Number of attributes in model = " +
  number_attributes_per_sample);
System.out.println(
  "Number of samples = " + number_samples);
System.out.println("Summary: " + summary);
System.out.println();
```

Now we create a new classifier (a J48 classifier in this case) and we see how to optionally filter (remove) samples. We build a classifier using the training data and then test it using the separate test data set:

```
// a classifier for decision trees:
J48 j48 = new J48();

// filter for removing samples:
Remove rm = new Remove();
// remove first attribute
rm.setAttributeIndices("1");

// filtered classifier
FilteredClassifier fc = new FilteredClassifier();
fc.setFilter(rm);
fc.setClassifier(j48);
// train using stock_training_data.arff:
fc.buildClassifier(training_data);
// test using stock_testing_data.arff:
for (int i = 0;
     i < testing_data.numInstances(); i++) {
  double pred =
         fc.classifyInstance(testing_data.
                             instance(i));
  System.out.print("given value: " +
    testing_data.classAttribute().
      value((int)testing_data.instance(i).
                 classValue()));
  System.out.println(". predicted value: " +
```

```
        testing_data.classAttribute().value((int)pred));
    }
  }
}
```

This example program produces the following output (some output not shown due to page width limits):

```
Number of attributes in model = 4
Number of samples = 9
Summary: Relation Name:  stock
Num Instances:  9
Num Attributes: 4

     Name                      Type  Nom  Int Real  ...
   1 percent_change_since_open  Num   0%  11%  89%  ...
   2 percent_change_from_day_l  Num   0%  22%  78%  ...
   3 percent_change_from_day_h  Num   0%   0% 100%  ...
   4 action                     Nom 100%   0%   0%  ...

given value: hold. predicted value: hold
given value: sell. predicted value: sell
given value: buy. predicted value: buy
given value: hold. predicted value: buy
given value: sell. predicted value: sell
given value: buy. predicted value: buy
given value: hold. predicted value: hold
given value: sell. predicted value: buy
given value: buy. predicted value: buy
```

8.4 Suggestions for Further Study

Weka is well documented in the book *Data Mining: Practical Machine Learning Tools and Techniques, Second Edition* [Ian H. Witten (Author), Eibe Frank. 2005]. Additional documentation can be found at weka.sourceforge.net/wiki/index.php.

9 Statistical Natural Language Processing

We will cover a wide variety of techniques for processing text in this chapter. The part of speech tagger, text categorization, clustering, spelling, and entity extraction examples are all derived from either my open source projects or my commercial projects. I wrote the Markov model example code for an earlier edition of this book.

I am not offering you a very formal view of Statistical Natural Language Processing in this chapter; rather, I collected Java code that I have been using for years on various projects and simplified it to (hopefully) make it easier for you to understand and modify for your own use. The web site http://nlp.stanford.edu/links/statnlp.html is an excellent resource for both papers when you need more theory and additional software for Statistical Natural Language Processing. For Python programmers I can recommend the statistical NLP toolkit NLTK (nltk.sourceforge.net) that includes an online book and is licensed using the GPL.

9.1 Tokenizing, Stemming, and Part of Speech Tagging Text

Tokenizing text is the process of splitting a string containing text into individual tokens. Stemming is the reduction of words to abbreviated word roots that allow for easy comparison for equality of similar words. Tagging is identifying what part of speech each word is in input text. Tagging is complicated by many words having different parts of speech depending on context (examples: "*bank* the airplane," "the river *bank*," etc.) You can find the code in this section in the code ZIP file for this book in the files src/com/knowledgebooks/nlp/fasttag/FastTag.java and src/com/knowledgebooks/nlp/util/Tokenizer.java. The required data files are in the directory test_data in the files lexicon.txt (for processing English text) and lexicon_medpost.txt (for processing medical text). The FastTag project can also be found on my open source web page:

http://www.markwatson.com/opensource

We will also look at a public domain word stemmer that I frequently use in this

section.

Before we can process any text we need to break text into individual tokens. Tokens can be words, numbers and punctuation symbols. The class $Tokenizer$ has two static methods, both take an input string to tokenize and return a list of token strings. The second method has an extra argument to specify the maximum number of tokens that you want returned:

```
static public List<String> wordsToList(String s)
static public List<String> wordsToList(String s,
                                       int maxR)
```

The following listing shows a fragment of example code using this class with the output:

```
String text =
  "The ball, rolling quickly, went down the hill.";
List<String> tokens = Tokenizer.wordsToList(text);
System.out.println(text);
for (String token : tokens)
   System.out.print("\""+token+"\" ");
System.out.println();
```

This code fragment produces the following output:

```
The ball, rolling quickly, went down the hill.
"The" "ball" "," "rolling" "quickly" "," "went"
"down" "the" "hill" "."
```

For many applications, it is better to "stem" word tokens to simplify comparison of similar words. For example "run," "runs," and "running" all stem to "run." The stemmer that we will use, which I believe to be in the public domain, is in the file src/public_domain/Stemmer.java. There are two convenient APIs defined at the end of the class, one to stem a string of multiple words and one to stem a single word token:

```
public List<String> stemString(String str)
public String stemOneWord(String word)
```

We will use both the $FastTag$ and $Stemmer$ classes often in the remainder of this chapter.

The FastTag project resulted from my using the excellent tagger written by Eric Brill while he was at the University of Pennsylvania. He used machine learning techniques to learn transition rules for tagging text using manually tagged text as training examples. In reading through his doctoral thesis I noticed that there were a few transition rules that covered most of the cases and I implemented a simple "fast tagger" in Common Lisp, Ruby, Scheme and Java. The Java version is in the file src/com/knowledgebooks/nlp/fasttag/FastTag.java.

The file src/com/knowledgebooks/nlp/fasttag/README.txt contains information on where to obtain Eric Brill's original tagging system and also defines the tags for both his English language lexicon and the Medpost lexicon. Table 9.1 shows the most commonly used tags (see the README.txt file for a complete description).

Tag	**Description**	**Examples**
NN	singular noun	dog
NNS	plural noun	dogs
NNP	singular proper noun	California
NNPS	plural proper noun	Watsons
CC	conjunction	and, but, or
CD	cardinal number	one, two
DT	determiner	the, some
IN	preposition	of, in, by
JJ	adjective	large, small, green
JJR	comparative adjective	bigger
JJS	superlative adjective	biggest
PP	proper pronoun	I, he, you
RB	adverb	slowly
RBR	comparative adverb	slowest
RP	particle	up, off
VB	verb	eat
VBN	past participle verb	eaten
VBG	gerund verb	eating
VBZ	present verb	eats
WP	wh* pronoun	who, what
WDT	wh* determiner	which, that

Table 9.1: Most commonly used part of speech tags

Brill's system worked by processing manually tagged text and then creating a list of words followed by the tags found for each word. Here are a few random lines selected from the test_data/lexicon.txt file:

```
Arco NNP
Arctic NNP JJ
fair JJ NN RB
```

Here "Arco" is a proper noun because it is the name of a corporation. The word "Arctic" can be either a proper noun or an adjective; it is used most frequently as a proper noun so the tag "NNP" is listed before "JJ." The word "fair" can be an adjective, singular noun, or an adverb.

The class $Tagger$ reads the file lexicon either as a resource stream (if, for example, you put lexicon.txt in the same JAR file as the compiled $Tagger$ and $Tokenizer$ class files) or as a local file. Each line in the lexicon.txt file is passed through the utility method $parseLine$ that processes an input string using the first token in the line as a hash key and places the remaining tokens in an array that is the hash value. So, we would process the line "fair JJ NN RB" as a hash key of "fair" and the hash value would be the array of strings (only the first value is currently used but I keep the other values for future use):

```
["JJ", "NN", "RB"]
```

When the tagger is processing a list of word tokens, it looks each token up in the hash table and stores the first possible tag type for the word. In our example, the word "fair" would be assigned (possibly temporarily) the tag "JJ." We now have a list of word tokens and an associated list of possible tag types. We now loop through all of the word tokens applying eight transition rules that Eric Brill's system learned. We will look at the first rule in some detail; i is the loop variable in the range [0, number of word tokens - 1] and $word$ is the current word at index i:

```
//  rule 1: DT, {VBD | VBP} --> DT, NN
if (i > 0 && ret.get(i - 1).equals("DT")) {
    if (word.equals("VBD") ||
        word.equals("VBP") ||
        word.equals("VB")) {
                    ret.set(i, "NN");
    }
}
```

In English, this rule states that if a determiner (DT) at word token index $i - 1$ is followed by either a past tense verb (VBD) or a present tense verb (VBP) then replace the tag type at index i with "NN."

I list the remaining seven rules in a short syntax here and you can look at the Java source code to see how they are implemented:

```
rule 2: convert a noun to a number (CD) if "."
        appears in the word
rule 3: convert a noun to a past participle if
```

```
        words.get(i) ends with "ed"
rule 4: convert any type to adverb if it ends in "ly"
rule 5: convert a common noun (NN or NNS) to an
        adjective if it ends with "al"
rule 6: convert a noun to a verb if the preceding
        work is "would"
rule 7: if a word has been categorized as a common
        anoun nd it ends with "s", then set its type
        to plural common noun (NNS)
rule 8: convert a common noun to a present participle
        verb (i.e., a gerund)
```

My FastTag tagger is not quite as accurate as Brill's original tagger so you might want to use his system written in C but which can be executed from Java as an external process or with a JNI interface.

In the next section we will use the tokenizer, stemmer, and tagger from this section to develop a system for identifying named entities in text.

9.2 Named Entity Extraction From Text

In this section we will look at identifying names of people and places in text. This can be useful for automatically tagging news articles with the people and place names that occur in the articles. The "secret sauce" for identifying names and places in text is the data in the file test_data/propername.ser – a serialized Java data file containing hash tables for human and place names. This data is read in the constructor for the class $Names$; it is worthwhile looking at the code if you have not used the Java serialization APIs before:

```
ObjectInputStream p = new ObjectInputStream(ins);
Hashtable lastNameHash = (Hashtable) p.readObject();
Hashtable firstNameHash = (Hashtable) p.readObject();
Hashtable placeNameHash = (Hashtable) p.readObject();
Hashtable prefixHash = (Hashtable) p.readObject();
```

If you want to see these data values, use code like

```
while (keysE.hasMoreElements()) {
  Object key = keysE.nextElement();
  System.out.println(key + " : " +
                     placeNameHash.get(key));
}
```

to see data values like the following:

```
Mauritius : country
Port-Vila : country_capital
Hutchinson : us_city
Mississippi : us_state
Lithuania : country
```

Before we look at the entity extraction code and how it works, we will first look at an example of using the main APIs for the $Names$ class. The following example uses the methods $isPlaceName$, $isHumanName$, and $getProperNames$:

```
System.out.println("Los Angeles: " +
            names.isPlaceName("Los Angeles"));
System.out.println("President Bush: " +
            names.isHumanName("President Bush"));
System.out.println("President George Bush: " +
     names.isHumanName("President George Bush"));
System.out.println("President George W. Bush: " +
     names.isHumanName("President George W. Bush"));
ScoredList[] ret = names.getProperNames(
      "George Bush played golf. President     \
       George W. Bush went to London England, \
       and Mexico to see Mary    \
       Smith in Moscow. President Bush will   \
       return home Monday.");
System.out.println("Human names: " +
                   ret[0].getValuesAsString());
System.out.println("Place names: " +
                   ret[1].getValuesAsString());
```

The output from running this example is:

```
Los Angeles: true
President Bush: true
President George Bush: true
President George W. Bush: true
* place name: London,
       placeNameHash.get(name): country_capital
* place name: Mexico,
       placeNameHash.get(name): country_capital
* place name: Moscow,
```

```
        placeNameHash.get(name): country_capital
Human names: George Bush:1,
             President George W . Bush:1,
             Mary Smith:1,
             President Bush:1
Place names: London:1, Mexico:1, Moscow:1
```

The complete implementation that you can read through in the source file ExtractNames.java is reasonably simple. The methods $isHumanName$ and $isPlaceName$ simply look up a string in either of the human or place name hash tables. For testing a single word this is very easy; for example:

```
public boolean isPlaceName(String name) {
  return placeNameHash.get(name) != null;
}
```

The versions of these APIs that handle names containing multiple words are just a little more complicated; we need to construct a string from the words between the starting and ending indices and test to see if this new string value is a valid key in the human names or place names hash tables. Here is the code for finding multi-word place names:

```
public boolean isPlaceName(List<String> words,
                           int startIndex,
                           int numWords) {
  if ((startIndex + numWords) > words.size()) {
    return false;
  }
  if (numWords == 1) {
    return isPlaceName(words.get(startIndex));
  }
  String s = "";
  for (int i=startIndex;
       i<(startIndex + numWords); i++) {
    if (i < (startIndex + numWords - 1)) {
      s = s + words.get(startIndex) + " ";
    } else {
      s = s + words.get(startIndex);
    }
  }
  return isPlaceName(s);
}
```

This same scheme is used to test for multi-word human names. The top-level utility method $getProperNames$ is used to find human and place names in text. The code in $getProperNames$ is intentionally easy to understand but not very efficient because of all of the temporary test strings that need to be constructed.

9.3 Using the WordNet Linguistic Database

The home page for the WordNet project is http://wordnet.princeton.edu and you will need to download version 3.0 and install it on your computer to use the example programs in this section and in Chapter 10. As you can see on the WordNet web site, there are several Java libraries for accessing the WordNet data files; we will use the JAWS library written by Brett Spell as a student project at the Southern Methodist University. I include Brett's library and the example programs for this section in the directory src-jaws-wordnet in the ZIP file for this book.

9.3.1 Tutorial on WordNet

The WordNet lexical database is an ongoing research project that includes many man years of effort by professional linguists. My own use of WordNet over the last ten years has been simple, mainly using the database to determine synonyms (called synsets in WordNet) and looking at the possible parts of speech of words. For reference (as taken from the Wikipedia article on WordNet), here is a small subset of the type of relationships contained in WordNet for verbs shown by examples (taken from the Wikipedia article):

hypernym travel (less general) is an hypernym of movement (more general)

entailment to sleep is entailed by to snore because you must be asleep to snore

Here are a few of the relations supported for nouns:

hypernyms canine is a hypernym of dog since every dog is of type canine

hyponyms dog (less general) is a hyponym of canine (more general)

holonym building is a holonym of window because a window is part of a building

meronym window is a meronym of building because a window is part of a building

Some of the related information maintained for adjectives is:

related nouns

similar to

I find the WordNet book (*WordNet: An Electronic Lexical Database (Language, Speech, and Communication)* by Christiane Fellbaum, 1998) to be a detailed reference for WordNet but there have been several new releases of WordNet since the book was published. The WordNet site and the Wikipedia article on WordNet are also good sources of information if you decide to make WordNet part of your toolkit:

```
http://wordnet.princeton.edu/
http://en.wikipedia.org/wiki/WordNet
```

We will Brett's open source Java WordNet utility library in the next section to experiment with WordNet. There are also good open source client applications for browsing the WordNet lexical database that are linked on the WordNet web site.

9.3.2 Example Use of the JAWS WordNet Library

Assuming that you have downloaded and installed WordNet on your computer, if you look at the data files themselves you will notice that the data is divided into index and data files for different data types. The JAWS library (and other WordNet client libraries for many programming languages) provides a useful view and convenient access to the WordNet data. You will need to define a Java property for the location of the raw WordNet data files in order to use JAWS; on my system I set:

```
wordnet.database.dir=/Users/markw/temp/wordnet3/dict
```

The example class $WordNetTest$ finds the different word senses for a given word and prints this data to standard output. We will tweak this code slightly in the next section where we will be combining WordNet with a part of speech tagger in another example program.

Accessing WordNet data using Brett's library is easy, so we will spend more time actually looking at the WordNet data itself. Here is a sample program that shows how to use the APIs. The class constructor makes a connection to the WordNet data files for reuse:

```
public class WordNetTest {
  public WordNetTest() {
    database =
       WordNetDatabase.getFileInstance();
  }
```

Here I wrap a JAWS utility method to return lists of synsets instead of raw Java arrays:

```
  public List<Synset> getSynsets(String word) {
    return Arrays.asList(database.getSynsets(word));
  }
  public static void main(String[] args) {
```

The constant $PropertyNames.DATABASE\_DIRECTORY$ is equal to "wordnet.database.dir." It is a good idea to make sure that you have this Java property set; if the value prints as null, then either fix the way you set Java properties, or just set it explicitly:

```
  System.setProperty(PropertyNames.DATABASE_DIRECTORY,
                  "/Users/markw/temp/wordnet3/dict");
  WordNetTest tester = new WordNetTest();
  String word = "bank";
  List<Synset> synset_list = tester.getSynsets(word);
  System.out.println("\n\n** Process word: " + word);
  for (Synset synset : synset_list) {
    System.out.println("\nsynset type:       " +
           SYNSET_TYPES[synset.getType().getCode()]);
    System.out.println("       definition: " +
                       synset.getDefinition());
    // word forms are synonyms:
    for (String wordForm : synset.getWordForms()) {
      if (!wordForm.equals(word)) {
        System.out.println("       synonym:    " +
                            wordForm);
```

Antonyms are the opposites to synonyms. Notice that antonyms are specific to individual senses for a word. This is why I have the following code to display antonyms inside the loop over word forms for each word sense for "bank":

```
          // antonyms mean the opposite:
          for (WordSense antonym :
             synset.getAntonyms(wordForm)) {
            for (String opposite :
                 antonym.getSynset().getWordForms()) {
              System.out.println(
                        "             antonym (of " +
                        wordForm+"): " + opposite);
            }
          }
        }
      }
```

```
      System.out.println("\n");
    }
  }
  private WordNetDatabase database;
  private final static String[] SYNSET_TYPES =
                          {"", "noun", "verb"};
}
```

Using this example program, we can see the word "bank" has 18 different "senses," 10 noun, and 8 verb senses:

```
** Process word: bank

synset type:       noun
       definition: sloping land (especially the slope
                   beside a body of water)
synset type:       noun
       definition: a financial institution that accepts
                   deposits and channels the money into
                   lending activities
       synonym:    depository financial institution
       synonym:    banking concern
       synonym:    banking company
synset type:       noun
       definition: a long ridge or pile
synset type:       noun
       definition: an arrangement of similar objects
                   in a row or in tiers
synset type:       noun
       definition: a supply or stock held in reserve
                   for future use (especially in
                   emergencies)
synset type:       noun
       definition: the funds held by a gambling house
                   or the dealer in some gambling games
synset type:       noun
       definition: a slope in the turn of a road or
                   track; the outside is higher than
                   the inside in order to reduce the
                   effects of centrifugal force
       synonym:    cant
       synonym:    camber
synset type:       noun
       definition: a container (usually with a slot
```

```
                    in the top) for keeping money
                    at home
       synonym:     savings bank
       synonym:     coin bank
       synonym:     money box
synset type:        noun
       definition:  a building in which the business
                    of banking transacted
       synonym:     bank building
synset type:        noun
       definition:  a flight maneuver; aircraft
                    tips laterally about its
                    longitudinal axis
                    (especially in turning)
synset type:        verb
       definition:  tip laterally
synset type:        verb
       definition:  enclose with a bank
synset type:        verb
       definition:  do business with a bank or
                    keep an account at a bank
synset type:        verb
       definition:  act as the banker in a game
                    or in gambling
synset type:        verb
       definition:  be in the banking business
synset type:        verb
       definition:  put into a bank account
       synonym:     deposit
             antonym (of deposit): withdraw
             antonym (of deposit): draw
             antonym (of deposit): take out
             antonym (of deposit): draw off
synset type:        verb
       definition:  cover with ashes so to control
                    the rate of burning
synset type:        verb
       definition:  have confidence or faith in
       synonym:     trust
             antonym (of trust): distrust
             antonym (of trust): mistrust
             antonym (of trust): suspect
             antonym (of trust): distrust
             antonym (of trust): mistrust
             antonym (of trust): suspect
```

```
synonym:    swear
synonym:    rely
```

WordNet provides a rich linguistic database for human linguists but although I have been using WordNet since 1999, I do not often use it in automated systems. I tend to use it for manual reference and sometimes for simple tasks like augmenting a list of terms with synonyms. In the next two sub-sections I suggest two possible projects both involving use of synsets (synonyms). I have used both of these suggested ideas in my own projects with some success.

9.3.3 Suggested Project: Using a Part of Speech Tagger to Use the Correct WordNet Synonyms

We saw in Section 9.3 that WordNet will give us both synonyms and antonyms (opposite meaning) of words. The problem is that we can only get words with similar and opposite meanings for specific "senses" of a word. Using the example in Section 9.3, synonyms of the word "bank" in the sense of a verb meaning "have confidence or faith in" are:

- trust
- swear
- rely

while synonyms for "bank" in the sense of a noun meaning "a financial institution that accepts deposits and channels the money into lending activities" are:

- depository financial institution
- banking concern
- banking company

So, it does not make too much sense to try to maintain a data map of synonyms for a given word. It does make some sense to try to use some information about the context of a word. We can do this with some degree of accuracy by using the part of speech tagger from Section 9.1 to at least determine that a word in a sentence is a noun or a verb, and thus limit the mapping of possible synonyms for the word in its current context.

9.3.4 Suggested Project: Using WordNet Synonyms to Improve Document Clustering

Another suggestion for a WordNet-based project is to use the Tagger to identify the probable part of speech for each word in all text documents that you want to cluster, and augment the documents with sysnset (synonym) data. You can then cluster the documents similarly to how we will calculate document similarity in Section 9.5.

9.4 Automatically Assigning Tags to Text

By tagging I mean assigning zero or more categories like "politics", "economy", etc. to text based on the words contained in the text. While the code for doing this is simple there is usually much work to do to build a word count database for different classifications.

I have been working on commercial products for automatic tagging and semantic extraction for about ten years (see www.knowledgebooks.com if you are interested). In this section I will show you some simple techniques for automatically assigning tags or categories to text using some code snippets from my own commercial product. We will use a set of tags for which I have collected word frequency statistics. For example, a tag of "Java" might be associated with the use of the words "Java," "JVM," "Sun," etc. You can find my pre-trained tag data in the file:

```
test_data/classification_tags.xml
```

The Java source code for the class $AutoTagger$ is in the file:

```
src-statistical-nlp/
      com/knowledgebooks/nlp/AutoTagger.java
```

The $AutoTagger$ class uses a few data structures to keep track of both the names of tags and the word count statistics for words associated with each tag name. I use a temporary hash table for processing the XML input data:

```
  private static
    Hashtable<String, Hashtable<String, Float>>
      tagClasses;
```

The names of tags used are defined in the XML tag data file: change this file, and you alter both the tags and behavior of this utility class. Here is a snippet of data

defined in the XML tag data file describing some words (and their scores) associated with the tag "religion_buddhism":

```
<tags>
  <topic name="religion_buddhism">
    <term name="buddhism" score="52" />
    <term name="buddhist" score="50" />
    <term name="mind" score="50" />
    <term name="medit" score="41" />
    <term name="buddha" score="37" />
    <term name="practic" score="31" />
    <term name="teach" score="15" />
    <term name="path" score="14" />
    <term name="mantra" score="14" />
    <term name="thought" score="14" />
    <term name="school" score="13" />
    <term name="zen" score="13" />
    <term name="mahayana" score="13" />
    <term name="suffer" score="12" />
    <term name="dharma" score="12" />
    <term name="tibetan" score="11" />
    . . .
  </topic>
  . . .
</tags>
```

Notice that the term names are stemmed words and all lower case. There are 28 tags defined in the input XML file included in the ZIP file for this book.

For data access, I also maintain an array of tag names and an associated list of the word frequency hash tables for each tag name:

```
  private static String[] tagClassNames;
  private static
    List<Hashtable<String, Float>> hashes =
       new ArrayList<Hashtable<String, Float>>();
```

The XML data is read and these data structures are filled during static class load time so creating multiple instances of the class $AutoTagger$ has no performance penalty in either memory use or processing time. Except for an empty default class constructor, there is only one public API for this class, the method $getTags$:

```
  public List<NameValue<String, Float>>
                 getTags(String text) {
```

The utility class $NameValue$ is defined in the file:

```
src-statistical-nlp/
      com/knowledgebooks/nlp/util/NameValue.java
```

To determine the tags for input text, we keep a running score for each defined tag type. I use the internal class $SFtriple$ to hold triple values of word, score, and tag index. I choose the tags with the highest scores as the automatically assigned tags for the input text. Scores for each tag are calculated by taking each word in the input text, stemming it, and if the stem is in the word frequency hash table for the tag then add the score value in the hash table to the running sum for the tag. You can refer to the AutoTagger.java source code for details. Here is an example use of class $AutoTagger$:

```
AutoTagger test = new AutoTagger();
String s = "The President went to Congress to argue
            for his tax bill before leaving on a
            vacation to Las Vegas to see some shows
            and gamble.";
List<NameValue<String, Float>> results =
                                  test.getTags(s);
for (NameValue<String, Float> result : results) {
  System.out.println(result);
}
```

The output looks like:

```
[NameValue: news_economy : 1.0]
[NameValue: news_politics : 0.84]
```

9.5 Text Clustering

The text clustering system that I have written for my own projects, in simplified form, will be used in the section. It is inherently inefficient when clustering a large number of text documents because I perform significant semantic processing on each text document and then compare all combinations of documents. The runtime performance is $O(N^2)$ where N is the number of text documents. If you need to cluster or compare a very large number of documents you will probably want to use a K-Mean clustering algorithm (search for "K-Mean clustering Java" for some open source projects).

I use a few different algorithms to rate the similarity of any two text documents and I will combine these depending on the requirements of the project that I am working on:

1. Calculate the intersection of common words in the two documents.
2. Calculate the intersection of common word stems in the two documents.
3. Calculate the intersection of tags assigned to the two documents.
4. Calculate the intersection of human and place names in the two documents.

In this section we will implement the second option: calculate the intersection of word stems in two documents. Without showing the package and import statements, it takes just a few lines of code to implement this algorithm when we use the *Stemmer* class.

The following listing shows the implementation of class *ComparableDocument* with comments. We start by defining constructors for documents defined by a *File* object and a *String* object:

```
public class ComparableDocument {
    // disable default constructor calls:
    private ComparableDocument() { }
    public ComparableDocument(File document)
                 throws FileNotFoundException {
        this(new Scanner(document).
                   useDelimiter("\\Z").next());
    }
    public ComparableDocument(String text) {
        List<String> stems =
               new Stemmer().stemString(text);
        for (String stem : stems) {
            stem_count++;
            if (stemCountMap.containsKey(stem)) {
                Integer count = stemCountMap.get(stem);
                stemCountMap.put(stem, 1 + count);
            } else {
                stemCountMap.put(stem, 1);
            }
        }
    }
```

In the last constructor, I simply create a count of how many times each stem occurs in the document.

The public API allows us to get the stem count hash table, the number of stems in the original document, and a numeric comparison value for comparing this document with another (this is the first version – we will add an improvement later):

```
  public Map<String, Integer> getStemMap() {
    return stemCountMap;
  }
  public int getStemCount() {
    return stem_count;
  }
  public float
         compareTo(ComparableDocument otherDocument) {
    long count = 0;
    Map<String,Integer> map2 = otherDocument.getStemMap();
    Iterator iter = stemCountMap.keySet().iterator();
    while (iter.hasNext()) {
      Object key = iter.next();
      Integer count1 = stemCountMap.get(key);
      Integer count2 = map2.get(key);
      if (count1!=null && count2!=null) {
        count += count1 * count2;
      }
    }
    return (float) Math.sqrt(
             ((float)(count*count) /
              (double)(stem_count *
                       otherDocument.getStemCount())))
           / 2f;
  }
  private Map<String, Integer> stemCountMap =
                   new HashMap<String, Integer>();
  private int stem_count = 0;
}
```

I normalize the return value for the method $compareTo$ to return a value of 1.0 if compared documents are identical (after stemming) and 0.0 if they contain no common stems. There are four test text documents in the test_data directory and the following test code compares various combinations. Note that I am careful to test the case of comparing identical documents:

```
  ComparableDocument news1 =
     new ComparableDocument("testdata/news_1.txt");
  ComparableDocument news2 =
```

```
    new ComparableDocument("testdata/news_2.txt");
ComparableDocument econ1 =
    new ComparableDocument("testdata/economy_1.txt");
ComparableDocument econ2 =
    new ComparableDocument("testdata/economy_2.txt");
System.out.println("news 1 - news1: " +
                   news1.compareTo(news1));
System.out.println("news 1 - news2: " +
                   news1.compareTo(news2));
System.out.println("news 2 - news2: " +
                   news2.compareTo(news2));
System.out.println("news 1 - econ1: " +
                   news1.compareTo(econ1));
System.out.println("econ 1 - econ1: " +
                   econ1.compareTo(econ1));
System.out.println("news 1 - econ2: " +
                   news1.compareTo(econ2));
System.out.println("econ 1 - econ2: " +
                   econ1.compareTo(econ2));
System.out.println("econ 2 - econ2: " +
                   econ2.compareTo(econ2));
```

The following listing shows output that indicates mediocre results; we will soon make an improvement that makes the results better. The output for this test code is:

```
news 1 - news1: 1.0
news 1 - news2: 0.4457711
news 2 - news2: 1.0
news 1 - econ1: 0.3649214
econ 1 - econ1: 1.0
news 1 - econ2: 0.32748842
econ 1 - econ2: 0.42922822
econ 2 - econ2: 1.0
```

There is not as much differentiation in comparison scores between political news stories and economic news stories. What is up here? The problem is that I did not remove common words (and therefore common word stems) when creating stem counts for each document. I wrote a utility class $NoiseWords$ for identifying both common words and their stems; you can see the implementation in the file NoiseWords.java. Removing noise words improves the comparison results (I added a few tests since the last printout):

```
news 1 - news1: 1.0
```

```
news 1 - news2: 0.1681978
news 1 - econ1: 0.04279895
news 1 - econ2: 0.034234844
econ 1 - econ2: 0.26178515
news 2 - econ2: 0.106673114
econ 1 - econ2: 0.26178515
```

Much better results! The API for com.knowledgebooks.nlp.util.NoiseWords is:

```
public static boolean checkFor(String stem)
```

You can add additional noise words to the data section in the file NoiseWords.java, depending on your application.

9.6 Spelling Correction

Automating spelling correction is a task that you may use for many types of projects. This includes both programs that involve users entering text that will be automatically processed with no further interaction with the user and for programs that keep the user "in the loop" by offering them possible spelling choices that they can select. I have used five different approaches in my own work for automating spelling correction and getting spelling suggestions:

- An old project of mine (overly complex, but with good accuracy)
- Embedding the GNU ASpell utility
- Use the LGPL licensed Jazzy spelling checker (a port of the GNU ASpell spelling system to Java)
- Using Peter Norvig's statistical spelling correction algorithm
- Using Norvig's algorithm, adding word pair statistics

We will use the last three options in the next Sections 9.6.1, 9.6.2 and in Section 9.6.3 where we will extend Norvig's algorithm by also using word pair statistics. This last approach is computationally expensive and is best used in applications with a highly specialized domain of discourse (e.g., systems dealing just with boats, sports, etc.).

Section 9.6.3 also provides a good lead in to Section 9.7 dealing with a similar but more general technique covered later in this chapter: Markov Models.

9.6.1 GNU ASpell Library and Jazzy

The GNU ASpell system is a hybrid system combining letter substitution and addition (which we will implement as a short example program in Section 9.6.2), the Soundex algorithm, and dynamic programming. I consider ASpell to be a best of breed spelling utility and I use it fairly frequently with scripting languages like Ruby where it is simple to "shell out" and run external programs.

You can also "shell out" external commands to new processes in Java but there is no need to do this if we use the LGPLed Jazzy library that is similar to ASpell and written in pure Java. For the sake of completeness, here is a simple example of how you would use ASpell as an external program; first, we will run ASpell on in a command shell (not all output is shown):

```
markw$ echo "ths doog" | /usr/local/bin/aspell  -a list
@(#) International Ispell (but really Aspell 0.60.5)
& ths 22 0: Th's, this, thus, Th, \ldots
& doog 6 4: dog, Doug, dong, door, \ldots
```

This output is easy enough to parse; here is an example in Ruby (Python, Perl, or Java would be similar):

```
def ASpell text
  s = `echo "#{text}" | /usr/local/bin/aspell -a list`
  s = s.split("\n")
  s.shift
  results = []
  s.each {|line|
    tokens = line.split(",")
    header = tokens[0].gsub(':','').split(' ')
    tokens[0] = header[4]
    results <<
       [header[1], header[3],
        tokens.collect {|tt| tt.strip}] if header[1]
  }
  results
end
```

I include the source code to the LGPLed Jazzy library and a test class in the directory src-spelling-Jazzy. The Jazzy library source code is in the sub-directory com/swabunga. We will spend no time looking at the implementation of the Jazzy library: this short section is simply meant to get you started quickly using Jazzy.

Here is the test code from the file SpellingJazzyTester.java:

```
File dict =
   new File("test_data/dictionary/english.0");
SpellChecker checker =
   new SpellChecker(new SpellDictionaryHashMap(dict));
int THRESHOLD = 10; // computational cost threshold
System.out.println(checker.getSuggestions("runnng",
                                          THRESHOLD));
System.out.println(checker.getSuggestions("season",
                                          THRESHOLD));
System.out.println(checker.getSuggestions(
                           "advantagius", THRESHOLD));
```

The method $getSuggestions$ returns an $ArrayList$ of spelling suggestions. This example code produces the following output:

```
[running]
[season, seasons, reason]
[advantageous, advantages]
```

The file test_data/dictionary/english.0 contains an alphabetically ordered list of words, one per line. You may want to add words appropriate for the type of text that your applications use. For example, if you were adding spelling correction to a web site for selling sailboats then you would want to insert manufacturer and product names to this word list in the correct alphabetical order.

The title of this book contains the word "Practical," so I feel fine about showing you how to use a useful Open Source package like Jazzy without digging into its implementation or APsell's implementation. The next section contains the implementation of a simple algorithm and we will study its implementation some detail.

9.6.2 Peter Norvig's Spelling Algorithm

Peter Norvig designed and implemented a spelling corrector in about 20 lines of Python code. I will implement his algorithm in Java in this section and in Section 9.6.3 I will extend my implementation to also use word pair statistics.

The class $SpellingSuggestions$ uses static data to create an in-memory spelling dictionary. This initialization will be done at class load time so creating instances of this class will be inexpensive. Here is the static initialization code with error handling removed for brevity:

```
private static Map<String, Integer> wordCounts =
```

```
        new HashMap<String, Integer>();
static {
  // Use Peter Norvig's training file big.txt:
  // http://www.norvig.com/spell-correct.html
  FileInputStream fstream =
             new FileInputStream("/tmp/big.txt");
  DataInputStream in = new DataInputStream(fstream);
  BufferedReader br =
      new BufferedReader(new InputStreamReader(in));
  String line;
  while ((line = br.readLine()) != null) {
    List<String> words = Tokenizer.wordsToList(line);
    for (String word : words) {
      if (wordCounts.containsKey(word)) {
         Integer count = wordCounts.get(word);
         wordCounts.put(word, count + 1);
      } else {
         wordCounts.put(word, 1);
      }
    }
  }
  in.close();
}
```

The class has two static methods that implement the algorithm. The first method *edits* seen in the following listing is private and returns a list of permutations for a string containing a word. Permutations are created by removing characters, by reversing the order of two adjacent characters, by replacing single characters with all other characters, and by adding all possible letters to each space between characters in the word:

```
private static List<String> edits(String word) {
  int wordL = word.length(), wordLm1 = wordL - 1;
  List<String> possible = new ArrayList<String>();
  // drop a character:
  for (int i=0; i < wordL; ++i) {
    possible.add(word.substring(0, i) +
                 word.substring(i+1));
  }
  // reverse order of 2 characters:
  for (int i=0; i < wordLm1; ++i) {
    possible.add(word.substring(0, i) +
                 word.substring(i+1, i+2) +
                 word.substring(i, i+1) +
```

```
                    word.substring(i+2));
    }
    // replace a character in each location in the word:
    for (int i=0; i < wordL; ++i) {
      for (char ch='a'; ch <= 'z'; ++ch) {
          possible.add(word.substring(0, i) + ch +
                       word.substring(i+1));
      }
    }
    // add in a character in each location in the word:
    for (int i=0; i <= wordL; ++i) {
      for (char ch='a'; ch <= 'z'; ++ch) {
        possible.add(word.substring(0, i) + ch +
                     word.substring(i));
      }
    }
    return possible;
  }
```

Here is a sample test case for the method *edits* where we call it with the word "cat" and get a list of 187 permutations:

```
[at, ct, ca, act, cta, aat, bat, cat, .., fat, ..,
 cct, cdt, cet, .., caty, catz]
```

The public static method *correct* has four possible return values:

- If the word is in the spelling hash table, simply return the word.
- Generate a permutation list of the input word using the method *edits*. Build a hash table *candidates* from the permutation list with keys being the word count in the main hashtable *wordCounts* with values of the words in the permutation list. If the hash table *candidates* is not empty then return the permutation with the best key (word count) value.
- For each new word in the permutation list, call the method *edits* with the word, creating a new *candidates* hash table with permutations of permutations. If candidates is not empty then return the word with the highest score.
- Return the value of the original word (no suggestions).

```
  public static String correct(String word) {
    if(wordCounts.containsKey(word)) return word;
    List<String> list = edits(word);
```

```
    /**
      * Candidate hash has word counts as keys,
      * word as value:
      */
    HashMap<Integer, String> candidates =
                  new HashMap<Integer, String>();
    for (String testWord : list) {
      if(wordCounts.containsKey(testWord)) {
        candidates.put(wordCounts.get(testWord),
                       testWord);
      }
    }
    /**
     *  If candidates is not empty, then return
     *  the word with the largest key (word
     *  count) value:
     */
    if(candidates.size() > 0) {
      return candidates.get(
             Collections.max(candidates.keySet()));
    }
    /**
     * If the edits method does not provide a
     * candidate word that matches then we will
     * call edits again with each previous
     * permutation words.
     *
     * Note: this case occurs only about 20%
     *       of the time and obviously increases
     *       the runtime of method correct.
     */
    candidates.clear();
    for (String editWords : list) {
      for (String wrd : edits(editWords)) {
        if(wordCounts.containsKey(wrd)) {
          candidates.put(wordCounts.get(wrd),wrd);
        }
      }
    }
    if (candidates.size() > 0) {
      return candidates.get(
             Collections.max(candidates.keySet()));
    }
    return word;
  }
```

Although Peter Norvig's spelling algorithm is much simpler than the algorithm used in ASpell it works well. I have used Norvig's spelling algorithm for one customer project that had a small specific vocabulary instead of using ASpell. We will extend Norvig's spelling algorithm in the next section to also take advantage of word pair statistics.

9.6.3 Extending the Norvig Algorithm by Using Word Pair Statistics

It is possible to use statistics for which words commonly appear together to improve spelling suggestions. In my experience this is only worthwhile when applications have two traits:

1. The vocabulary for the application is specialized. For example, a social networking site for people interested in boating might want a more accurate spelling system than one that has to handle more general English text. In this example, common word pairs might be multi-word boat and manufacturer names, boating locations, etc.

2. There is a very large amount of text in this limited subject area to use for training. This is because there will be many more combinations of word pairs than words and a very large training set helps to determine which pairs are most common, rather than just coincidental.

We will proceed in a similar fashion to the implementation in the last section but we will also keep an additional hash table containing counts for word pairs. Since there will be many more word pair combinations than single words, you should expect both the memory requirements and CPU time for training to be much larger. For one project, there was so much training data that I ended up having to use disk-based hash tables to store word pair counts.

To make this training process take less training time and less memory to hold the large word combination hash table, we will edit the input file big.txt from the last section deleting the 1200 lines that contain random words added to the end of the Project Gutenberg texts. Furthermore, we will experiment with an even smaller version of this file (renamed small.txt) that is about ten percent of the size of the original training file. Because we are using a smaller training set we should expect marginal results. For your own projects you should use as much data as possible.

In principle, when we collect a word pair hash table where the hash values are the number of times a word pair occurs in the training test, we would want to be sure that we do not collect word pairs across sentence boundaries and separate phrases occurring inside of parenthesis, etc. For example consider the following text fragment:

```
He went to Paris. The weather was warm.
```

Optimally, we would not want to collect statistics on word (or token) pairs like “Paris .” or “Paris The” that include the final period in a sentence or span a sentence. In a practical sense, since we will be discarding seldom occurring word pairs, it does not matter too much so in our example we will collect all tokenized word pairs at the same time that we collect single word frequency statistics:

```
Pattern p = Pattern.compile("[,.()'\";:\\s]+");
Scanner scanner =
        new Scanner(new File("/tmp/small.txt"));
scanner.useDelimiter(p);
String last = "ahjhjhdsgh";
while (scanner.hasNext()) {
  String word = scanner.next();
  if (wordCounts.containsKey(word)) {
    Integer count = wordCounts.get(word);
    wordCounts.put(word, count + 1);
  } else {
    wordCounts.put(word, 1);
  }
  String pair = last + " " + word;
  if (wordPairCounts.containsKey(pair)) {
    Integer count = wordPairCounts.get(pair);
    wordPairCounts.put(pair, count + 1);
  } else {
    wordPairCounts.put(pair, 1);
  }
  last = word;
}
scanner.close();
```

For the first page of text in the test file, if we print out word pairs that occur at least two times using this code:

```
for (String pair : wordPairCounts.keySet()) {
  if (wordPairCounts.get(pair) > 1) {
    System.out.println(pair + ": " +
                       wordPairCounts.get(pair));
  }
}
```

then we get this output:

```
Arthur Conan: 3
by Sir: 2
of Sherlock: 2
Project Gutenberg: 5
how to: 2
The Adventures: 2
Sherlock Holmes: 2
Sir Arthur: 3
Adventures of: 2
information about: 2
Conan Doyle: 3
```

The words "Conan" and "Doyle" tend to appear together frequently. If we want to suggest spelling corrections for "the author Conan *Doyyle* wrote" it seems intuitive that we can prefer the correction "Doyle" since if we take the possible list of corrections for "*Doyyle*" and combine each with the preceding word "Conan" in the text, then we notice that the hash table $wordPairCounts$ has a relatively high count for the key "Conan Doyle" that is a single string containing a word pair.

In theory this may look like a good approach, but there are a few things that keep this technique from being generally practical:

- It is computationally expensive to train the system for large training text.
- It is more expensive computationally to perform spelling suggestions.
- The results are not likely to be much better than the single word approach unless the text is in one narrow domain and you have a lot of training text.

In the example of misspelling *Doyyle*, calling the method $edits$:

```
edits("Doyyle")
```

returns a list with 349 elements.

The method $edits$ is identical to the one word spelling corrector in the last section. I changed the method $correct$ by adding an argument for the previous word, factoring in statistics from the word pair count hash table, and for this example by not calculating "edits of edits" as we did in the last section. Here is the modified code:

```
public  String correct(String word,
                       String previous_word) {
  if(wordCounts.containsKey(word)) return word;
  List<String> list = edits(word);
```

```
    // candidate hash has as word counts
    // as keys, word as value:
    HashMap<Integer, String> candidates =
                 new HashMap<Integer, String>();
    for (String testWord : list) {
      // look for word pairs with testWord in the
      // second position:
      String word_pair = previous_word + " " + testWord;
      int count_from_1_word = 0;
      int count_from_word_pairs = 0;
      if(wordCounts.containsKey(testWord)) {
        count_from_1_word += wordCounts.get(testWord);
        candidates.put(wordCounts.get(testWord),
                                      testWord);
      }
      if (wordPairCounts.containsKey(word_pair)) {
        count_from_word_pairs +=
                   wordPairCounts.get(word_pair);
      }
      // look for word pairs with testWord in the
      // first position:
      word_pair = testWord + " " + previous_word;
      if (wordPairCounts.containsKey(word_pair)) {
        count_from_word_pairs +=
                    wordPairCounts.get(word_pair);
      }
      int sum = count_from_1_word +
                count_from_word_pairs;
      if (sum > 0)  {
        candidates.put(sum, testWord);
      }
    }
    /**
     * If candidates is not empty, then return the
     * word with the largest key (word count) value:
     */
    if(candidates.size() > 0) {
      return candidates.get(
              Collections.max(candidates.keySet()));
    }
    return word;
  }
```

Using word pair statistics can be a good technique if you need to build an automated spelling corrector that only needs to work on text in one subject area. You will need a lot of training text in your subject area and be prepared for extra work performing the training: as I mentioned before, for one customer project I could not fit the word pair hash table in memory (on the server that I had to use) so I had to use a disk-based hash table – the training run took a long while. Another good alternative for building systems for handling text in one subject area is to augment a standard spelling library like ASpell or Jazzy with custom word dictionaries.

9.7 Hidden Markov Models

We used a set of rules in Section 9.1 to assign parts of speech tags to words in English text. The rules that we used were a subset of the automatically generated rules that Eric Brill's machine learning thesis project produced. His thesis work used Markov modeling to calculate the most likely tag of words, given precceding words. He then generated rules for taging – some of which we saw in Section 9.1 where we saw Brill's published results of the most useful learned rules made writing a fast tagger relatively easy.

In this section we will use word-use statistics to assign word type tags to each word in input text. We will look in some detail at one of the most popular approaches to tagging text: building Hidden Markov Models (HMM) and then evaluating these models against input text to assign word use (or part of speech) tags to words.

A complete coverage of the commonly used techniques for training and using HMM is beyond the scope of this section. A full reference for these training techniques is *Foundations of Statistical Natural Language Processing* [Manning, Schutze, 1999]. We will discuss the training algorithms and sample Java code that implements HMM. The example in this chapter is purposely pedantic: the example code is intended to be easy to understand and experiment with.

In Hidden Markov Models (HMM), we speak of an observable sequence of events that moves a system through a series of states. We attempt to assign transition probabilities based on the recent history of states of the system (or, the last few events).

In this example, we want to develop an HMM that attempts to assign part of speech tags to English text. To train an HMM, we will assume that we have a large set of training data that is a sequence of words and a parallel sequence of manually assigned part of speech tags. We will see an example of this marked up training text that looks like "John/NNP chased/VB the/DT dog/NN" later in this section.

For developing a sample Java program to learn how to train a HMM, we assume that we have two Java lists words and tags that are of the same length. So, we will have one list of words like ["John", "chased", "the", "dog"] and an associated list of part

of speech tags like [“NNP”, “VB”, “DT”, “NN”].

Once the HMM is trained, we will write another method $test\_model$ that takes as input a Java vector of words and returns a Java vector of calculated part of speech tags.

We now describe the assumptions made for Markov Models and Hidden Markov Models using this part of speech tagging problem. First, assume that the desired part of speech tags are an observable sequence like:

$t[1], t[2], t[3], ..., t[N]$

and the original word sequence is:

$w[1], w[2], w[3], ..., w[N]$

We will also assume that the probability of tag t[M] having a specific value is only a function of:

$t[M-1]$

and:

$w[M] and w[M-1]$

Here we are only using the last state: in some applications, instead of using the last observed state, we might use the last two states, greatly increasing the resources (CPU time and memory) required for training.

For our example, we will assume that we have a finite lexicon of words. We will use a hash table that uses the words in the lexicon as keys and the values are the possible parts of speech.

For example, assuming the lexicon hash table is named lexicon, we use the notation:

lexicon[“a-word”] -> list of possible tags

Table 9.2 shows some of the possible tags used in our example system.

Tag Name	Part of Speech
VB	verb
NN	noun
ADJ	adjective
ADV	adverb
IN	preposition
NNP	noun

Table 9.2: Sample part of speech tags

As an example, we might have a lexicon entry:

lexicon[“bank”] -> NN, VB

where the work “bank” could be a noun (“I went to the bank”) or a verb (“To turn, bank the airplane”). In the example program, I use a hash table to hold the lexicon in the file Markov.java:

```
Map<String,List<String>> lexicon =
        new Hashtable<String,List<String>>();
```

Another hash table keeps a count of how frequently each tag is used:

```
Map<String, Integer> tags =
        new Hashtable<String, Integer>();
Map<String, Integer> words =
        new Hashtable<String, Integer>();
```

As you will see in Tables 9.3, 9.4, and 9.5 we will be operating on 2D arrays where in the first two tables the rows and columns represent unique tag names and in the last table the columns represent unique words and the columns represent unique tag names. We use the following data structures to keep a list of unique tags and words (a hash table will not work since we need an ordered sequence):

```
List<String> uniqueTags = new ArrayList<String>();
List<String> uniqueWords = new ArrayList<String>();
```

We will look at the training algorithm and implementation in the next section.

9.7.1 Training Hidden Markov Models

We will be using code in the file Markov.java and I will show snippets of this file with comments in this section. You can refer to the source code for the complete implementation. There are four main methods in the class $Markov$:

- build_words_and_tags()
- print_statistics()
- train_model
- test_model

	JJ	*IN*	*VB*	*VBN*	*TO*	*NNP*	*PRP*	*NN*	*RB*	*VBG*	*DT*
JJ	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	1.0	0.0	0.0
IN	0.0	0.0	0.0	0.0	0.0	1.0	0.0	2.0	0.0	0.0	4.0
VB	0.0	3.0	0.0	0.0	3.0	3.0	0.0	1.0	1.0	0.0	14.0
VBN	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0
TO	0.0	0.0	2.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	2.0
NNP	0.0	1.0	16.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
PRP	0.0	0.0	2.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
NN	0.0	3.0	5.0	1.0	2.0	0.0	0.0	1.0	0.0	0.0	0.0
RB	0.0	0.0	0.0	0.0	0.0	1.0	1.0	0.0	0.0	0.0	0.0
VBG	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0
DT	1.0	0.0	1.0	0.0	0.0	0.0	0.0	25.0	0.0	0.0	0.0

Table 9.3: Transition counts from the first tag (shown in row) to the second tag (shown in column). We see that the transition from NNP to VB is common.

In order to train a Markov model to tag parts of speech, we start by building a two-dimensional array using the method $build\_words\_and\_tags$ that uses the following 2D array to count transitions; part of this array was seen in Figure 9.3:

$tagToTagTransitionCount[uniqueTagCount][uniqueTagCount]$

where the first index is the index of tag_n and the second index is the index of tag_{n+1}. We will see later how to calculate the values in this array and then how the values in this two-dimensional array will be used to calculate the probabilities of transitioning from one tag to another. First however, we simply use this array for counting transitions between pairs of tags. The purpose of the training process is to fill this array with values based on the hand-tagged training file:

training_data/markov/tagged_text.txt

That looks like this:

```
John/NNP chased/VB the/DT dog/NN down/RP the/DT
street/NN ./.  I/PRP saw/VB John/NNP dog/VB
Mary/NNP and/CC later/RB Mary/NNP throw/VB
the/DT ball/NN to/TO John/NNP on/IN the/DT
street/NN ./.
```

The method $build\_words\_and\_tags$ parses this text file and fills the $uniqueTags$ and $uniqueWords$ collections.

The method $train\_model$ starts by filling the tag to tag transition count array(see Table 9.3):

$tagToTagTransitionCount[][]$

The element $tagToTagTransitionCount[indexTag0][indexTag1]$ is incremented whenever we find a transition of tag_n to tag_{n+1} in the input training text. The example program writes a spreadsheet style CSV file for this and other two-dimensional arrays that are useful for viewing intermediate training results in any spreadsheet program. We normalized the data seen in Table 9.3 by dividing each element by the count of the total number of tags. This normalized data can be seen in Table 9.4. The code for this first step is:

```
// start by filling in the tag to tag transition
// count matrix:
tagToTagTransitionCount =
       new float[uniqueTagCount][uniqueTagCount];
p("tagCount="+tagCount);
p("uniqueTagCount="+uniqueTagCount);
for (int i = 0; i < uniqueTagCount; i++) {
  for (int j = 0; j < uniqueTagCount; j++) {
    tagToTagTransitionCount[i][j] = 0;
  }
}
String tag1 = (String) tagList.get(0);
int index1 = uniqueTags.indexOf(tag1); // inefficient
int index0;
for (int i = 0, size1 = wordList.size() - 1;
     i < size1; i++) {
  index0 = index1;
  tag1 = (String) tagList.get(i + 1);
  index1 = uniqueTags.indexOf(tag1);   // inefficient
  tagToTagTransitionCount[index0][index1]++;
}
WriteCSVfile(uniqueTags, uniqueTags,
             tagToTagTransitionCount, "tag_to_tag");
```

Note that all calls to the utility method $WriteCSVfile$ are for debug only: if you use this example on a large training set (i.e., a large text corpus like Treebank of hand-tagged text) then these 2D arrays containing transition and probability values will be very large so viewing them with a spreadsheet is convenient.

Then the method $train\_model$ calculates the probabilities of transitioning from tag[N] to tag[M] (see Table 9.4).

Here is the code for calculating these transition probabilities:

```
// now calculate the probabilities of transitioning
```

	JJ	*IN*	*VB*	*VBN*	*TO*	*NNP*	*PRP*	*NN*	*RB*	*VBG*	*DT*
JJ	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.50	0.50	0.00	0.00
IN	0.00	0.00	0.00	0.00	0.00	0.14	0.00	0.29	0.00	0.00	0.57
VB	0.00	0.11	0.00	0.00	0.11	0.11	0.00	0.04	0.04	0.00	0.52
VBN	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00
TO	0.00	0.00	0.40	0.00	0.00	0.20	0.00	0.00	0.00	0.00	0.40
NNP	0.00	0.05	0.76	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
PRP	0.00	0.00	0.67	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
NN	0.00	0.10	0.16	0.03	0.06	0.00	0.00	0.03	0.00	0.00	0.00
RB	0.00	0.00	0.00	0.00	0.00	0.33	0.33	0.00	0.00	0.00	0.00
VBG	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00
DT	0.04	0.00	0.04	0.00	0.00	0.00	0.00	0.93	0.00	0.00	0.00

Table 9.4: Normalize data in Table 9.3 to get probability of one tag (seen in row) transitioning to another tag (seen in column)

```
// from tag[N] to tag[M]:
probabilityTag1ToTag2 =
     new float[uniqueTagCount][uniqueTagCount];
for (int i = 0; i < uniqueTagCount; i++) {
  int count =
   ((Integer)tags.get(
             (String)uniqueTags.get(i))).intValue();
  p("tag: " + uniqueTags.get(i) + ", count="+count);
  for (int j = 0; j < uniqueTagCount; j++) {
    probabilityTag1ToTag2[i][j] =
      0.0001f + tagToTagTransitionCount[i][j]
                / (float)count;
  }
}
WriteCSVfile(uniqueTags, uniqueTags,
             probabilityTag1ToTag2,
             "test_data/markov/prob_tag_to_tag");
```

Finally, in the method $train\_model$ we complete the training by defining the array

$probabilityWordGivenTag[uniqueWordCount][uniqueTagCount]$

which shows the probability of a tag at index N producing a word at index N in the input training text.

Here is the code for this last training step:

```
// now calculate the probability of a word, given
```

	JJ	*IN*	*VB*	*VBN*	*TO*	*NNP*	*PRP*	*NN*	*RB*
went	0.00	0.00	0.07	0.00	0.00	0.00	0.00	0.00	0.00
mary	0.00	0.00	0.00	0.00	0.00	0.52	0.00	0.00	0.00
played	0.00	0.00	0.07	0.00	0.00	0.00	0.00	0.00	0.00
river	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.03	0.00
leave	0.00	0.00	0.07	0.00	0.00	0.00	0.00	0.03	0.00
dog	0.00	0.00	0.04	0.00	0.00	0.00	0.00	0.23	0.00
away	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.33
chased	0.00	0.00	0.11	1.00	0.00	0.00	0.00	0.00	0.00
at	0.00	0.14	0.00	0.00	0.00	0.00	0.00	0.00	0.00
tired	0.50	0.00	0.04	0.00	0.00	0.00	0.00	0.00	0.00
good	0.50	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
had	0.00	0.00	0.04	0.00	0.00	0.00	0.00	0.00	0.00
throw	0.00	0.00	0.07	0.00	0.00	0.00	0.00	0.03	0.00
from	0.00	0.14	0.00	0.00	0.00	0.00	0.00	0.00	0.00
so	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.33
stayed	0.00	0.00	0.04	0.00	0.00	0.00	0.00	0.00	0.00
absense	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.03	0.00
street	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.06	0.00
john	0.00	0.00	0.00	0.00	0.00	0.48	0.00	0.00	0.00
ball	0.00	0.00	0.04	0.00	0.00	0.00	0.00	0.06	0.00
on	0.00	0.29	0.00	0.00	0.00	0.00	0.00	0.00	0.00
cat	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.32	0.00
later	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.33
she	0.00	0.00	0.00	0.00	0.00	0.00	0.33	0.00	0.00
of	0.00	0.14	0.00	0.00	0.00	0.00	0.00	0.00	0.00
with	0.00	0.29	0.00	0.00	0.00	0.00	0.00	0.00	0.00
saw	0.00	0.00	0.19	0.00	0.00	0.00	0.00	0.03	0.00

Table 9.5: Probabilities of words having specific tags. Only a few tags are shown in this table.

```
// a proceeding tag:
probabilityWordGivenTag =
      new float[uniqueWordCount][uniqueTagCount];
for (int i = 0; i < uniqueWordCount; i++) {
  String tag = uniqueTags.get(j);
  for (int j = 0; j < uniqueTagCount; j++) {
    String tag = uniqueTags.get(j);
    // note: index of tag is one less than index
    // of emitted word we are testing:
    int countTagOccurence = tags.get(tag);
    float wordWithTagOccurence = 0;
    for (int n=0, sizem1=wordList.size()-1;
         n<sizem1; n++) {
      String testWord = wordList.get(n);
      String testTag  = tagList.get(n);
      if (testWord.equals(word) &&
          testTag.equals(tag)) {
        wordWithTagOccurence++;
      }
    }
    probabilityWordGivenTag[i][j] =
      wordWithTagOccurence / (float)countTagOccurence;
  }
}
WriteCSVfile(uniqueWords, uniqueTags,
             probabilityWordGivenTag,
             "test_data/markov/prob_word_given_tag");
```

9.7.2 Using the Trained Markov Model to Tag Text

From Section 9.7.1 we have the probabilities of a given tag being assigned to words in the lexicon and we have the probability of a given tag, given the preceding tag. We will use this information in a "brute force" way in the method $test\_model$: we will iterate through all possible tagging possibilities and rate them using the formula from *Foundations of Statistical Natural Language Processing* [Manning/Schutze, 1999] page 347:

$Rating = \prod_{i=1} P(word_i|tag_i) * P(tag_i|tag_{i-1})$

$P(word_i|tag_i)$ is the probability of $word$ having a tag value tag and $P(tag_i|tag_{i-1})$ is the probability of tag_i following tag_{i-1}. We can simply implement two nested loops over all possible tags for each input word and use the tag for each word with the highest rating (score).

The arrays for these probabilities in Markov.java are $probabilityWordGivenTag$ and $probabilityTag1ToTag2$. The logic for scoring a specific tagging possibility for a sequence of words in the method score.

The method $exponential\_tagging\_algorithm$ is the top level API for tagging words. Please note that the word sequence that you pass to $exponential\_tagging\_algorithm$ must not contain any words that were not in the original training data (i.e., in the file tagged_text.txt).

```
public List<String>
   exponential_tagging_algorithm(List<String> words) {
  possibleTags = new ArrayList<ArrayList<String>>();
  int num = words.size();
  indices = new int[num];
  counts = new int[num];
  int [] best_indices = new int[num];
  for (int i=0; i<num; i++) {
    indices[i] = 0; counts[i] = 0;
  }
  for (int i=0; i<num; i++) {
    String word = "" + words.get(i);
    List<String> v = lexicon.get(word);
     // possible tags at index i:
    ArrayList<String> v2 = new ArrayList<String>();
    for (int j=0; j<v.size(); j++) {
      String tag = "" + v.get(j);
      if (v2.contains(tag) == false) {
        v2.add(tag);  counts[i]++;
      }
    }
    // possible tags at index i:
    possibleTags.add(v2);
    System.out.print("^^ word: " + word + ",
                     tag count: " + counts[i] +
                     ", tags: ");
    for (int j=0; j<v2.size(); j++) {
      System.out.print(" " + v2.get(j));
    }
    System.out.println();
  }
  float best_score = -9999;
  do {
    System.out.print("Current indices:");
    for (int k=0; k<num; k++) {
      System.out.print(" " + indices[k]);
```

```
      }
      System.out.println();
      float score = score(words);
      if (score > best_score) {
        best_score = score;
        System.out.println(" * new best score: " +
                           best_score);
        for (int m=0; m<num; m++) {
          best_indices[m] = indices[m];
        }
      }
    } while (incrementIndices(num)); // see text below

    List<String> tags = new ArrayList<String>(num);
    for (int i=0; i<num; i++) {
      List<String> v = possibleTags.get(i);
      tags.add(v.get(best_indices[i]));
    }
    return tags;
  }
```

The method $incrementIndices$ is responsible for generating the next possible tagging for a sequence of words. Each word in a sequence can have one or more possible tags. The method $incrementIndices$ counts with a variable base per digit position. For example, if we had four words in an input sequence with the first and last words only having one possible tag value and the second having two possible tag values and the third word having three possible tag values, then $incrementIndices$ would count like this:

```
0 0 0 0
0 1 0 0
0 0 1 0
0 1 1 0
0 0 2 0
0 1 1 0
```

The generated indices (i.e., each row in this listing) are stored in the class instance variable $indices$ which is used in method $score$:

```
  /**
   *  Increment the class variable indices[] to point
   *  to the next possible set of tags to check.
   */
```

```
private boolean incrementIndices(int num) {
 for (int i=0; i<num; i++) {
   if (indices[i] < (counts[i] - 1)) {
      indices[i] += 1;
      for (int j=0; j<i; j++) {
        indices[j] = 0;
      }
      return true;
    }
  }
  return false;
}
```

We are not using an efficient algorithm if the word sequence is long. In practice this is not a real problem because you can break up long texts into smaller pieces for tagging; for example, you might want to tag just one sentence at a time.

10 Information Gathering

We saw techniques for extracting semantic information in Chapter 9 and we will augment that material with the use of Reuters Open Calais web services for information extraction from text. We will then look at information discovery in relational database, indexing and search tools and techniques.

10.1 Open Calais

The Open Calais system was developed by Clear Forest (later acquired by Reuters). Reuters allows free use (with registration) of their named entity extraction web service; you can make 20,000 web service calls a day. You need to sign up and get an access key at: www.opencalais.com. Starting in 1999, I have developed a similar named entity extraction system (see www.knowledgebooks.com) and I sometimes use both Open Calais and my own system together.

The example program in this section (OpenCalaisClient.java) expects the key to be set in your environment; on my MacBook I set (here I show a fake key – get your own):

```
OPEN_CALAIS_KEY=al4345lkea48586dgfta3129aq
```

You will need to make sure that this value can be obtained from a $System.getenv()$ call.

The Open Calais web services support JSON, REST, and SOAP calls. I will use the REST architectural style in this example. The Open Calais server returns an XML RDF payload that can be directly loaded into RDF data stores like Sesame (see Chapter 4). The example class $OpenCalaisClient$ depends on a trick that may break in future versions of the Open Calais web service: an XML comment block at the top of the returned RDF payload lists the types of entities and their values. For example, here is a sample of the header comments with most of the RDF payload removed for brevity:

```
<?xml version="1.0" encoding="utf-8"?>
```

```
<string xmlns="http://clearforest.com/">
<!--Use of the Calais Web Service is governed by the Terms
    of Service located at http://www.opencalais.com. By
    using this service or the results of the service you
    agree to these terms of service.
-->
<!--Relations:
 Country: France, United States, Spain
 Person: Hillary Clinton, Doug Hattaway, Al Gore
 City: San Francisco
 ProvinceOrState: Texas
-->
<rdf:RDF xmlns:rdf="http://www.w3.org/1 ..."
         xmlns:c="http://s.opencalais.com/1/pred/">
 ...
<rdf:type ...>
  ... .
</rdf:RDF>
</string>
```

Here we will simply parse out the relations from the comment block. If you want to use Sesame to parse the RDF payload and load it into a local RDF repository then you can alternatively load the returned Open Calais response by modifying the example code from Chapter 4 using:

```
StringReader sr = new StringReader(result);
RepositoryConnection connection =
                repository.getConnection();
connection.add(sr, "", RDFFormat.RDFXML);
```

Here are a few code snippets (incomplete code: please see the Java source file for more details) from the file OpenCalaisClient.java:

```
public Hashtable<String, List<String>>
       getPropertyNamesAndValues(String text)
         throws MalformedURLException, IOException {
  Hashtable<String, List<String>> ret =
         new Hashtable<String, List<String>>();
```

You need an Open Calais license key. The following code sets up the data for a REST style web service call and opens a connection to the server, makes the request, and retrieves the response in the string variable *payload*. The Java libraries for handling

HTTP connections make it simple to make a architecture style web service call and get the response as a text string:

```
String licenseID = System.getenv("OPEN_CALAIS_KEY");
String content = text;
String paramsXML = "<c:params  ... </c:params>";
StringBuilder sb =
        new StringBuilder(content.length() + 512);
sb.append("licenseID=").append(licenseID);
sb.append("&content=").append(content);
sb.append("&paramsXML=").append(paramsXML);
String payload = sb.toString();
URLConnection connection =
  new URL("http://api.opencalais.com ...").
      openConnection();
connection.addRequestProperty("Content-Type",
            "application/x-www-form-urlencoded");
connection.addRequestProperty("Content-Length",
            String.valueOf(payload.length()));
connection.setDoOutput(true);
OutputStream out = connection.getOutputStream();
OutputStreamWriter writer =
                  new OutputStreamWriter(out);
writer.write(payload);
writer.flush();
// get response from Open Calais server:
String result = new Scanner(
           connection.getInputStream()).
           useDelimiter("\\Z").next();
result = result.replaceAll("&lt;", "<").
                replaceAll("&gt;", ">");
```

The text that we are parsing looks like:

```
Country: France, United States, Spain
Person: Hillary Clinton, Doug Hattaway, Al Gore
```

so the text response is parsed to extract a list of values for each property name contained in the string variable $result$:

```
int index1 =
    result.indexOf("terms of service.-->");
```

```
    index1 = result.indexOf("<!--", index1);
    int index2 = result.indexOf("-->", index1);
    result =
        result.substring(index1 + 4, index2 - 1 + 1);
    String[] lines = result.split("\\n");
    for (String line : lines) {
      int index = line.indexOf(":");
      if (index > -1) {
        String relation = line.substring(0, index).trim();
        String[] entities =
              line.substring(index + 1).trim().split(",");
        for (int i = 0, size = entities.length;
             i < size; i++) {
          entities[i] = entities[i].trim();
        }
        ret.put(relation, Arrays.asList(entities));
      }
    }
    return ret;
  }
```

Again, I want to point out that the above code depends on the format of XML comments in the returned XML payload so this code may break in the future and require modification. Here is an example use of this API:

```
  String content =
    "Hillary Clinton likes to remind Texans that ...";
    Map<String, List<String>> results =
         new OpenCalaisClient().
             getPropertyNamesAndValues(content);
    for (String key : results.keySet()) {
      System.out.println("  " + key + ": " +
                         results.get(key));
    }
```

In this example the string value assigned to the variable *content* was about 500 words of text from a news article; the full text can be seen in the example data files. The output of this example code is:

```
  Person: [Hillary Clinton, Doug Hattaway, Al Gore]
  Relations: []
  City: [San Francisco]
  Country: [France, United States, Spain]
  ProvinceOrState: [Texas]
```

There are several ways that you might want to use named entity identification. One idea is to create a search engine that identifies people, places, and products in search results and offers users a linked set of documents or web pages that discuss the same people, places, and/or products. Another idea is to load the RDF payload returned by the Open Calais web service calls to an RDF repository and support SPARQL queries. You may also want to modify any content management systems (CMS) that you use to add tags for documents maintained in a CMS; using Open Calais you are limited to the types of entities that they extract. This limitation is one reason why I maintain and support my own system for named entity and classification (knowledgebooks.com) – I like some flexibility in the type of semantic information that I extract from text data. I covered some of the techniques that I use in my own work in Section 9.2 if you decide to implement your own system to replace or augment Open Calais.

10.2 Information Discovery in Relational Databases

We will look at some techniques for using the JDBC meta-data APIs to explore relational database resources where you at least have read access rights. In order to make installation of the example programs easier we will use the Derby pure Java database that is bundled with JDK 1.6. If you are still using JDK 1.5, please download the derby.jar file and copy it to the "lib" directory for the Java book examples:

http://db.apache.org/derby/

There are small differences in setting up a JDBC connection to an embedded Derby instance rather than accessing a remote server: these differences are not important to the material in this section, it is mostly a matter of changing a connection call.

I will use two XML data sources (data on US states and the CIA World FactBook) for these examples, and start with the program to insert these XML data files into the relational database:

```
src-info-disc-rdbs/CreateSampleDatabase.java
```

and continue with a program to print out all metadata that is implemented in the files:

```
src-info-disc-rdbs/DumpMetaData.java
src-info-disc-rdbs/DatabaseDiscovery.java
```

We will not implement any specific "database spidering" applications but I will provide some basic access techniques and give you some ideas for using database meta data in your own projects.

10.2.1 Creating a Test Derby Database Using the CIA World FactBook and Data on US States

The file test_data/XML/FactBook.xml contains data that I obtained from the FactBook web site and converted to XML. This XML file contains data for individual countries and a few general regions:

```
<FactBook year="2001">
  <country name="Aruba"
           location="Caribbean, island in the ..."
           background="Discovered and claimed ..."
           climate="tropical marine; little seasonal ..."
           terrain="flat; scant vegetation"
           resources="NEGL; white sandy beaches"
           hazards="lies outside the Caribbean
                    hurricane belt"
           population="70,007 (July 2001 est.)"
           government="parliamentary democracy"
           economy="Tourism is the mainstay
                    of the Aruban ..."
           inflation="4.2% (2000 est.)"
           languages="Dutch (official), Papiamento ..."
           religions="Roman Catholic 82%,
                      Protestant 8%, ..."
           capital="Oranjestad"
           unemployment="0.6% (1999 est.)"
           industries="tourism, transshipment
                       facilities, ..."
           agriculture="aloes; livestock; fish"
           exports="$2.2 billion (including oil
                    reexports) ..."
           imports="$2.5 billion (2000 est.)"
           debt="$285 million (1996)"
           aid="$26 million (1995); note -
                the Netherlands ..."
           internet_code=".aw"
  />
  ...
</FactBook>
```

The other sample XML file USstates.xml contains information on individual states:

```
<USstates year="2003">
  <state name="Alabama"
         abbrev="AL"
         capital="Montgomery"
         industry="Paper, lumber and wood products ..."
         agriculture="Poultry and eggs, cattle, ..."
         population="4447100">
  ...
</USstates>
```

The example class $CreateSampleDatabases$ reads both the files FactBook.xml and USsattes.xml and creates two tables "factbook" and "states" in a test database. The implementation of this utility class is simple: just parsing XML data and making JDBC calls to create and populate the two tables. You can look at the Java source file for details.

10.2.2 Using the JDBC Meta Data APIs

This chapter is about processing and using data from multiple sources. With the wealth of data stored in relational database systems, it is important to know how to "spider" databases much as you might need to spider data stored on specific web sites. The example class $DumpMetaData$ shows you how to discover tables, information about table columns, and query all tables in a specific database.

The constructor of class $DumpMetaData$ is called with a database URI and prints meta data and data to standard output. This code should be portable to database systems other than Derby by changing the driver name.

```
class DumpMetaData {
  public DumpMetaData(String connectionUrl)
         throws SQLException, ClassNotFoundException {
    Class.forName("org.apache.derby.jdbc.EmbeddedDriver");
    Connection conn =
               DriverManager.getConnection(connectionUrl);
    System.out.println("conn: " + conn);
    Statement s = conn.createStatement();
    DatabaseMetaData md = conn.getMetaData();

    // Discovery all table names in this database:
    List<String> tableNames = new ArrayList<String>(5);
```

We will use the method $getTables()$ to fetch a list of all tables in the database. The four arguments are:

- String catalog: can be used when database systems support catalogs. We will use null to act as a wildcard match.
- String schemaPattern: can be used when database systems support schemas. We will use null to act as a wildcard match.
- String tableNamePattern: a pattern to match table names; we will use "%" as a wildcard match.
- String types[]: the types of table names to return. Possible values include TABLE, VIEW, ALIAS, SYNONYM, and SYSTEM TABLE.

The method $getTables()$ returns a $ResultSet$ so we iterate through returned values just as you would in a regular SQL query using the JDBC APIs:

```
ResultSet table_rs =
   md.getTables(null, null, "%",
                new String[]{"TABLE"});
while (table_rs.next()) {
  System.out.println("Table: " +
                     table_rs.getString(3));
  tableNames.add(table_rs.getString(3));
}

// Loop over all tables printing column meta data and
// the first row:
for (String tableName : tableNames) {
  System.out.println("\n\n** Processing table " +
                     tableName + "\n");
  String query = "SELECT * from " + tableName;
  System.out.println(query);
  ResultSet rs = s.executeQuery(query);
  ResultSetMetaData table_meta = rs.getMetaData();
  int columnCount = table_meta.getColumnCount();
  System.out.println("\nColumn meta data for table:");
  List<String> columnNames = new ArrayList<String>(10);
  columnNames.add("");
  for (int col=1; col<=columnCount; col++) {
    System.out.println("Column " + col +  " name: " +
                       table_meta.getColumnLabel(col));
    System.out.println("  column data type: " +
                  table_meta.getColumnTypeName(col));
    columnNames.add(table_meta.getColumnLabel(col));
```

```
      }
      System.out.println("\nFirst row in table:");
      if (rs.next()) {
        for (int col=1; col<=columnCount; col++) {
          System.out.println("   " + columnNames.get(col) +
                              ": " + rs.getString(col));
        }
      }
    }
  }
}
```

Output looks like this:

```
Table: FACTBOOK
Table: USSTATES

** Processing table FACTBOOK

SELECT * from FACTBOOK

Column meta data for table:
Column 1 name: NAME
  column data type: VARCHAR
Column 2 name: LOCATION
  column data type: VARCHAR
Column 3 name: EXPORT
  column data type: BIGINT
Column 4 name: IMPORT
  column data type: BIGINT
Column 5 name: DEBT
  column data type: BIGINT
Column 6 name: AID
  column data type: BIGINT
Column 7 name: UNEMPLOYMENT_PERCENT
  column data type: INTEGER
Column 8 name: INFLATION_PERCENT
  column data type: INTEGER

First row in table:
   NAME: Aruba
   LOCATION: Caribbean, island in the Caribbean Sea,
             north of Venezuela
   EXPORT: 2200000000
```

```
   IMPORT: 2500000000
   DEBT: 285000000
   AID: 26000000
   UNEMPLOYMENT_PERCENT: 0
   INFLATION_PERCENT: 4

** Processing table USSTATES

SELECT * from USSTATES

Column meta data for table:
Column 1 name: NAME
  column data type: VARCHAR
Column 2 name: ABBREVIATION
  column data type: CHAR
Column 3 name: INDUSTRY
  column data type: VARCHAR
Column 4 name: AGRICULTURE
  column data type: VARCHAR
Column 5 name: POPULATION
  column data type: BIGINT

First row in table:
   NAME: Alabama
   ABBREVIATION: AL
   INDUSTRY: Paper, lumber and wood products, mining,
             rubber and plastic products, transportation
             equipment, apparel
   AGRICULTURE: Poultry and eggs, cattle, nursery stock,
                peanuts, cotton, vegetables, milk,
                soybeans
   POPULATION: 4447100
```

Using the JDBC meta data APIs is a simple technique but can be very useful for both searching many tables for specific column names and for pulling meta data and row data into local search engines. While most relational databases provide support for free text search of text fields in a database it is often better to export specific text columns in a table to an external search engine.

We will spend the rest of this chapter on index and search techniques. While we usually index web pages and local document repositories, keep in mind that data in relational databases can also easily be indexed either with hand written export utilities or automated techniques using the JDBC meta-data APIs that we used in this section.

10.2.3 Using the Meta Data APIs to Discern Entity Relationships

When database schemas are defined it is usually a top down approach: entities and their relationships are modeled and then represented as relational database tables. When automatically searching remote databases for information we might need to discern which entities and their relationships exist depending on table and column names.

This is likely to be a domain specific development effort. While it is feasible and probably useful to build a "database spider" for databases in a limited domain (for example car parts or travel destinations) to discern entity models and their relations, it is probably not possible without requiring huge resources to build a system that handles multiple data domains.

The expression "dark web" refers to information on the web that is usually not "spidered" – information that lives mostly in relational databases and often behind query forms. While there are current efforts by search engine companies to determine the data domains of databases hidden behind user entry forms using surrounding text, for most organizations this is simply too large a problem to solve. On the other hand, using the meta data of databases that you or your organization have read access to for "database spidering" is a more tractable problem.

10.3 Down to the Bare Metal: In-Memory Index and Search

Indexing and search technology is used in a wide range of applications. In order to get a good understanding of index and search we will design and implement an in-memory library in this section. In Section 10.4 we will take a quick look at the Lucene library and in Section 10.5 we will look at client programs using the Nutch indexing and search system that is based on Lucene.

We need a way to represent data to be indexed. We will use a simple package-visible class (no getters/setters, assumed to be in the same package as the indexing and search class):

```
class TestDocument {
  int id;
  String text;
  static int count = 0;
  TestDocument(String text) {
    this.text = text;
```

```
    id = count++;
  }
  public String toString() {
    int len = text.length();
    if (len > 25) len = 25;
    return "[Document id: " + id + ": " +
           text.substring(0,len) + "...]";
  }
}
```

We will write a class $InMemorySearch$ that indexes instances of the $TestDocument$ class and supplies an API for search. The first decision to make is how to store the index that maps search terms to documents that contain the search terms. One simple idea would be to use a map to maintain a set of document IDs for each search term; something like:

```
  Map<String, Set<Integer>> index;
```

This would be easy to implement but leaves much to be desired so we will take a different approach. We would like to rank documents by relevance but a relevance measure just based on containing all (or most) of the search terms is weak. We will improve the index by also storing a score of how many times a search term occurs in a document, scaled by the number of words in a document. Since our document model does not contain links to other documents we will not use a Google-like page ranking algorithm that increases the relevance of search results based on the number of incoming links to matched documents. We will use a utility class (again, assuming same package data visibility) to hold a document ID and a search term count. I used generics for the first version of this class to allow alternative types for counting word use in a document and later changed the code to hardwiring the types for ID and word count to native integer values for runtime efficiency and to use less memory. Here is the second version of the code:

```
class IdCount implements Comparable<IdCount> {
  int id = 0;
  int count = 0;
  public IdCount(int k, int v) {
    this.id = k;
    this.count = v;
  }
  public String toString() {
    return "[IdCount: " + id + " : " + count + "]";
  }
  @Override
```

```
  public int compareTo(IdCount o) {
    // don't use o.count - count: avoid overflows
    if (o.count == count) return 0;
    if (o.count > count) return 1;
    return -1;
  }
}
```

We can now define the data structure for our index:

```
  Map<String,TreeSet<IdCount>> index =
          new Hashtable<String, TreeSet<IdCount>>();
```

The following code is used to add documents to the index. I score word counts by dividing by the maximum word size I expect for documents; in principle it would be better to use a $Float$ value but I prefer working with and debugging code using integers – debug output is more readable. The reason why the number of times a word appears in a document needs to be scaled by the the size of the document is fairly obvious: if a given word appears once in a document with 10 words and once in another document with 1000 words, then the word is much more relevant to finding the first document.

```
  public void add(TestDocument document) {
    Map<String,Integer> wcount =
              new Hashtable<String,Integer>();
    StringTokenizer st =
        new StringTokenizer(document.text.toLowerCase(),
                            " .,;:!");
    int num_words = st.countTokens();
    if (num_words == 0)  return;
    while (st.hasMoreTokens()) {
      String word = st.nextToken();
      System.out.println(word);
      if (wcount.containsKey(word)) {
        wcount.put(word, wcount.get(word) +
                   (MAX_WORDS_PER_DOCUMENT / num_words));
      } else {
        wcount.put(word, MAX_WORDS_PER_DOCUMENT
                         / num_words);
      }
    }
    for (String word : wcount.keySet()) {
      TreeSet<IdCount> ts;
```

```
      if (index.containsKey(word)) {
        ts = index.get(word);
      } else {
        ts = new TreeSet<IdCount>();
        index.put(word, ts);
      }
      ts.add(new IdCount(document.id, wcount.get(word) *
                    MAX_WORDS_PER_DOCUMENT / num_words));
    }
  }
```

If a word is in the index hash table then the hash value will be a sorted $TreeSet$ of $IdCount$ objects. Sort order is in decreasing size of the scaled word count. Notice that I converted all tokenized words in document text to lower case but I did not stem the words. For some applications you may want to use a word stemmer as we did in Section 9.1. I used the temporary hash table $wcount$ to hold word counts for the document being indexed and once $wcount$ was created and filled, then looked up the $TreeSet$ for each word (creating it if it did not yet exist) and added in new $IdCount$ objects to represent the currently indexed document and the scaled number of occurrences for the word that is the index hash table key.

For development it is good to have a method that prints out the entire index; the following method serves this purpose:

```
  public void debug() {
    System.out.println(
          "*** Debug: dump of search index:\n");
    for (String word : index.keySet()) {
      System.out.println("\n* " + word);
      TreeSet<IdCount> ts = index.get(word);
      Iterator<IdCount> iter = ts.iterator();
      while (iter.hasNext()) {
        System.out.println("   " + iter.next());
      }
    }
  }
```

Here are a few lines of example code to create an index and add three test documents:

```
  InMemorySearch ims = new InMemorySearch();
  TestDocument doc1 =
    new TestDocument("This is a test for index and
                      a test for search.");
```

```
  ims.add(doc1);
  TestDocument doc2 =
    new TestDocument("Please test the index code.");
  ims.add(doc2);
  TestDocument doc3 =
     new TestDocument("Please test the index code
                       before tomorrow.");
  ims.add(doc3);
  ims.debug();
```

The method $debug$ produces the following output (most is not shown for brevity). Remember that the variable $IdCount$ contains a data pair: the document integer ID and a scaled integer word count in the document. Also notice that the $TreeSet$ is sorted in descending order of scaled word count.

```
*** Debug: dump of search index:

* code
   [IdCount: 1 : 40000]
   [IdCount: 2 : 20285]

* please
   [IdCount: 1 : 40000]
   [IdCount: 2 : 20285]

* index
   [IdCount: 1 : 40000]
   [IdCount: 2 : 20285]
   [IdCount: 0 : 8181]

 ...
```

Given the hash table $index$ it is simple to take a list of search words and return a sorted list of matching documents. We will use a temporary hash table $ordered\_results$ that maps document IDs to the current search result score for that document. We tokenize the string containing search terms, and for each search word we look up (if it exists) a score count in the temporary map $ordered\_results$ (creating a new $IdCount$ object otherwise) and increment the score count. Note that the map $ordered\_results$ is ordered later by sorting the keys by the hash table value:

```
  public List<Integer> search(String search_terms,
                              int max_terms) {
    List<Integer> ret = new ArrayList<Integer>(max_terms);
```

```
    // temporary tree set to keep ordered search results:
    final Map<Integer,Integer> ordered_results =
                        new Hashtable<Integer,Integer>(0);
    StringTokenizer st =
         new StringTokenizer(search_terms.toLowerCase(),
                                 " .,;:!");
    while (st.hasMoreTokens()) {
      String word = st.nextToken();
      Iterator<IdCount> word_counts =
                           index.get(word).iterator();
      while (word_counts.hasNext()) {
        IdCount ts = word_counts.next();
        Integer id = ts.id;
        if (ordered_results.containsKey(id)) {
          ordered_results.put(id,
                      ordered_results.get(id) + ts.count);
        } else {
          ordered_results.put(id, ts.count);
        }
      }
    }
    List<Integer> keys =
        new ArrayList<Integer>(ordered_results.keySet());
    Collections.sort(keys, new Comparator<Integer>() {
        public int compare(Integer a, Integer b) {
            return -ordered_results.get(a).
                      compareTo(ordered_results.get(b)) ;
        }
    });
    int count = 0;
  result_loop:
    for (Integer id : keys) {
      if (count++ >= max_terms) break result_loop;
      ret.add(id);
    }
    return ret;
  }
```

For the previous example using the three short test documents, we can search the index, in this case for a maximum of 2 results, using:

```
    List<Integer> search_results =
                        ims.search("test index", 2);
    System.out.println("result doc IDs: "+search_results);
```

getting the results:

```
result doc IDs: [1, 2]
```

If you want to use this "bare metal" indexing and search library, there are a few details that still need to be implemented. You will probably want to persist the $TestDocument$ objects and this can be done simply by tagging the class with the $Serializable$ interface and writing serialized files using the document ID as the file name. You might also want to serialize the $InMemorySearch$ class.

While I sometimes implement custom indexing and search libraries for projects that require a lightweight and flexible approach to indexing and search as we did in this section, I usually use either the Lucene search library or a combination of the Hibernate Object Relational Mapping (ORM) library with Lucene (Hibernate Search). We will look at Lucene in Section 10.4.

10.4 Indexing and Search Using Embedded Lucene

Books have been written on the Lucene indexing and search library and in this short section we will look at a brief application example that you can use for a quick reference for starting Lucene based projects. I consider Lucene to be an important tool for building intelligent text processing systems.

Lucene supports the concept of a document with one or more fields. Fields can either be indexed or not, and optionally stored in a disk-based index. Searchable fields can be automatically tokenized using either one of Lucene's built in text tokenizers or you can supply your customized tokenizer.

When I am starting a new project using Lucene I begin by using a template class $LuceneManager$ that you can find in the file src-index-search/LuceneManager.java. I usually clone this file and make any quick changes for adding fields to documents, etc. We will look at a few important code snippets in the class $LuceneManager$ and you can refer to the source code for more details. We will start by looking at how indices are stored and managed on disk. The class constructor stores the file path to the Lucene disk index. You can optionally use method $createAndClearLuceneIndex$ to delete an existing Lucene index (if it exists) and creates an empty index.

```
public LuceneManager(String data_store_file_root) {
    this.data_store_file_root = data_store_file_root;
}
```

```
public void createAndClearLuceneIndex()
              throws CorruptIndexException,
                     LockObtainFailedException,
                     IOException {
  deleteFilePath(new File(data_store_file_root +
                          "/lucene_index"));
  File index_dir = new File(data_store_file_root +
                            "/lucene_index");
  new IndexWriter(index_dir,
                  new StandardAnalyzer(), true).close();
}
```

If you are using an existing disk-based index that you want to reuse, then do **not** call method $createAndClearLuceneIndex$. The last argument to the class $IndexWriter$ constructor is a flag to create a new index, overwriting any existing indices. I use the utility method $deleteFilePath$ to make sure that all files from any previous indices using the same top level file path are deleted. The method $addDocumentToIndex$ is used to add new documents to the index. Here we call the constructor for the class $IndexWriter$ with a value of false for the last argument to avoid overwriting the index each time method $addDocumentToIndex$ is called.

```
public void addDocumentToIndex(
              String document_original_uri,
              String document_plain_text)
        throws CorruptIndexException, IOException {
  File index_dir =
    new File(data_store_file_root + "/lucene_index");
  writer =  new IndexWriter(index_dir,
                    new StandardAnalyzer(), false);
  Document doc = new Document();
  // store URI in index; do not index
  doc.add(new Field("uri",
                  document_original_uri,
                  Field.Store.YES,
                  Field.Index.NO));
  // store text in index; index
  doc.add(new Field("text",
                  document_plain_text,
                  Field.Store.YES,
                  Field.Index.TOKENIZED));
  writer.addDocument(doc);
  writer.optimize(); // optional
  writer.close();
}
```

You can add fields as needed when you create individual Lucene *Document* objects but you will want to add the same fields in your application: it is not good to have different documents in an index with different fields. There are a few things that you may want to change if you use this class as an implementation example in your own projects. If you are adding many documents to the index in a short time period, then it is inefficient to open the index, add one document, and then optimize and close the index. You might want to add a method that passes in collections of URIs and document text strings for batch inserts. You also may not want to store the document text in the index if you are already storing document text somewhere else, perhaps in a database.

There are two search methods in my *LuceneManager* class: one just returns the document URIs for search matches and the other returns both URIs and the original document text. Both of these methods open an instance of *IndexReader* for each query. For high search volume operations in a multi-threaded environment, you may want to create a pool of *IndexReader* instances and reuse them. There are several text analyzer classes in Lucene and you should use the same analyzer class when adding indexed text fields to the index as when you perform queries. In the two search methods I use the same *StandardAnalyzer* class that I used when adding documents to the index. The following method returns a list of string URIs for matched documents:

```
public List<String>
       searchIndexForURIs(String search_query)
            throws ParseException, IOException {
  reader = IndexReader.open(data_store_file_root +
                            "/lucene_index");
  List<String> ret = new ArrayList<String>();
  Searcher searcher = new IndexSearcher(reader);
  Analyzer analyzer = new StandardAnalyzer();
  QueryParser parser =
              new QueryParser("text", analyzer);
  Query query = parser.parse(search_query);
  Hits hits = searcher.search(query);
  for (int i = 0; i < hits.length(); i++) {
    System.out.println(
      " * * searchIndexForURIs: hit: " + hits.doc(i));
    Document doc = hits.doc(i);
    String uri = doc.get("uri");
    ret.add(uri);
  }
  reader.close();
  return ret;
}
```

The Lucene class *Hits* is used for returned search matches and here we use APIs to get the number of hits and for each hit get back an instance of the Lucene class *Document*. Note that the field values are retrieved by name, in this case "uri." The other search method in my utility class *searchIndexForURIsAndDocText* is almost the same as *searchIndexForURIs* so I will only show the differences:

```
public List<String[]>
       searchIndexForURIsAndDocText(
          String search_query) throws Exception {
  List<String[]> ret = new ArrayList<String[]>();
  ...
  for (int i = 0; i < hits.length(); i += 1) {
    Document doc = hits.doc(i);
    System.out.println("     * *  hit: " +
                       hits.doc(i));
    String [] pair =
      new String[]{doc.get("uri"), doc.get("text")};
    ret.add(pair);
  }
  ...
  return ret;
}
```

Here we also return the original text from matched documents that we get by fetching the named field "text." The following code snippet is an example for using the *LuceneManager* class:

```
LuceneManager lm = new LuceneManager("/tmp");
// start fresh: create a new index:
lm.createAndClearLuceneIndex();
lm.addDocumentToIndex("file://tmp/test1.txt",
   "This is a test for index and a test for search.");
lm.addDocumentToIndex("file://tmp/test2.txt",
   Please test the index code.");
lm.addDocumentToIndex("file://tmp/test3.txt",
   "Please test the index code before tomorrow.");
// get URIs of matching documents:
List<String> doc_uris =
  lm.searchIndexForURIs("test, index");
System.out.println("Matched document URIs: "+doc_uris);
// get URIs and document text for matching documents:
List<String[]> doc_uris_with_text =
  lm.searchIndexForURIsAndDocText("test, index");
```

```
  for (String[] uri_and_text : doc_uris_with_text) {
    System.out.println("Matched document URI:  " +
                       uri_and_text[0]);
    System.out.println("        document text: " +
                       uri_and_text[1]);
  }
```

and here is the sample output (with debug printout from deleting the old test disk-based index removed):

```
Matched document URIs: [file://tmp/test1.txt,
                        file://tmp/test2.txt,
                        file://tmp/test3.txt]
Matched document URI:  file://tmp/test1.txt
        document text: This is a test for index
                       and a test for search.
Matched document URI:  file://tmp/test2.txt
        document text: Please test the index code.
Matched document URI:  file://tmp/test3.txt
        document text: Please test the index code
                       before tomorrow.
```

I use the Lucene library frequently on customer projects and although tailoring Lucene to specific applications is not simple, the wealth of options for analyzing text and maintaining disk-based indices makes Lucene a very good tool. Lucene is also very efficient and scales well to very large indices.

In Section 10.5 we will look at the Nutch system that is built on top of Lucene and provides a complete turnkey (but also highly customizable) solution to implementing search in large scale projects where it does not make sense to use Lucene in an embedded mode as we did in this Section.

10.5 Indexing and Search with Nutch Clients

This is the last section in this book, and we have a great topic for finishing the book: the Nutch system that is a very useful tool for information storage and retrieval. Out of the box, it only takes about 15 minutes to set up a "vanilla" Nutch server with the default web interface for searching documents. Nutch can be configured to index documents on a local file system and contains utilities for processing a wide range of document types (Microsoft Office, OpenOffice.org, PDF, TML, etc.). You can also configure Nutch to spider remote and local private (usually on a company LAN) web sites.

The Nutch web site http://lucene.apache.org/nutch contains binary distributions and tutorials for quickly setting up a Nutch system and I will not repeat all of these directions here. What I do want to show you is how I usually use the Nutch system on customer projects: after I configure Nutch to periodically "spider" customer specific data sources I then use a web services client library to integrate Nutch with other systems that need both document repository and search functionality.

Although you can tightly couple your Java applications with Nutch using the Nutch API, I prefer to use the OpenSearch API that is an extension of RSS 2.0 for performing search using web service calls. OpenSearch was originally developed for Amazon's A9.com search engine and may become widely adopted since it is a reasonable standard. More information on the OpenSearch standard can be found at http://www.opensearch.org but I will cover the basics here.

10.5.1 Nutch Server Fast Start Setup

For completeness, I will quickly go over the steps I use to set up Tomcat version 6 with Nutch. For this discussion, I assume that you have unpacked Tomcat and changed the directory name to Tomcat6_Nutch, that you have removed all files from the directory Tomcat6_Nutch/webapps/, and that you have then moved the nutch-0.9.war file (I am using Nutch version 0.9) to the Tomcat $webapps$ directory changing its name to $ROOT.war$:

```
Tomcat6_Nutch/webapps/ROOT.war
```

I then move the directory nutch-0.9 to:

```
Tomcat6_Nutch/nutch
```

The file Tomcat6_Nutch/nutch/conf/crawl-urlfilter.txt needs to be edited to specify a combination of local and remote data sources; here I have configured it to spider just my http://knowledgebooks.com web site (the only changes I had to make are the two lines, one being a comment line containing the string "knowledgebooks.com"):

```
# skip file:, ftp:, & mailto: urls
-^(file|ftp|mailto):
# skip image and other suffixes we can't yet parse
-\.(gif|GIF|jpg|JPG| ...)$
# skip URLs containing certain characters as probable
# queries, etc.
-[?*!@=]
```

```
# skip URLs with slash-delimited segment that repeats
# 3+ times, to break loops
-.*(/.+?)/.*?\1/.*?\1/
# accept hosts in knowledgebooks.com
+^http://([a-z0-9]*\.)*knowledgebooks.com/
# skip everything else
-.
```

Additional regular expression patterns can be added for more root web sites. Nutch will not spider any site that does not match any regular expression pattern in the configuration file. It is important that web search spiders properly identify themselves so it is important that you also edit the file Tomcat6_Nutch/nutch/conf/nutch-site.xml, following the directions in the comments to identify yourself or your company to web sites that you spider.

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>

<configuration>
    <property>
      <name>http.agent.name</name>
      <value>YOUR NAME Nutch spider</value>
      <description>Test spider</description>
    </property>

    <property>
      <name>http.agent.url</name>
      <value>http://YOURDOMAIN.com</value>
      <description>URL of spider server</description>
    </property>

    <property>
      <name>http.agent.email</name>
      <value>YOUR EMAIL ADDRESS</value>
      <description>markw at markwatson dot com</description>
    </property>
</configuration>
```

Then create an empty directory:

```
Tomcat6_Nutch/nutch/urls
```

and create a text file (any file name is fine) with a list of starting URLs to spider; in this case, I will just add:

```
http://knowledgebooks.com
```

Then make a small test spider run to create local indices in the subdirectory ./crawl and start the Tomcat server interactively:

```
cd nutch/
bin/nutch crawl urls -dir crawl -depth 3 -topN 80
../bin/catalina.sh run
```

You can run Tomcat as a background service using "start" instead of "run" in production mode. If you rerun the spidering process, you will need to first delete the subdirectory ./crawl or put the new index in a different location and copy it to ./crawl when the new index is complete. The Nutch web app running in Tomcat will expect a subdirectory named ./crawl in the directory where you start Tomcat.

Just to test that you have Nutch up and running with a valid index, access the following URL (specifying localhost, assuming that you are running Tomcat on your local computer to try this):

```
http://localhost:8080
```

You can then try the OpenSearch web service interface by accessing the URL:

```
http://localhost:8080/opensearch?query=Java%20RDF
```

Since I indexed my own web site that I often change, the RSS 2.0 XML that you get back may look different than what we see in this example:

```
<?xml version="1.0" encoding="UTF-8"?>

<rss xmlns:nutch="http://www.nutch.org/ ...>
  <channel>
    <title>Nutch: Java RDF</title>
    <description>Nutch search results for
                 query: Java RDF</description>
    <link>http://localhost:8080/search ...</link>
    <opensearch:totalResults>1</opensearch:totalResults>
    <opensearch:startIndex>0</opensearch:startIndex>
    <opensearch:itemsPerPage>10</opensearch:itemsPerPage>

    <nutch:query>Java RDF</nutch:query>
```

```
    <item>
      <title>Knowledgebooks.com: AI  ...</title>
      <description> ...  HTML snippet ... </description>
      <link>http://knowledgebooks.com/</link>
      <nutch:site>knowledgebooks.com</nutch:site>
      <nutch:cache> ... </nutch:cache>
      <nutch:explain> ... </nutch:explain>
      <nutch:segment>20080930151220</nutch:segment>
      <nutch:digest>923fb80f9f8fd66f47d70</nutch:digest>
      <nutch:tstamp>20080930221225918</nutch:tstamp>
      <nutch:boost>1.0</nutch:boost>
    </item>
  </channel>
</rss>
```

For multiple search results, there are multiple <item> elements in the returned XML data. We will write web service clients to submit remote queries and process the returned RSS 2.0 XML data in Section 10.5.2.

10.5.2 Using the Nutch OpenSearch Web APIs

A Java OpenSearch web services client is fairly easy to write: build a REST query URL with the search phrase URL encoded, open a $HttpURLConnection$ for this query URL, read the response, and then use an XML parser to process the returned RSS 2.0 XML payload. We will first look at an implementation of a Nutch client and then look at some interesting things you can do, given your own Nutch server installation and a client. The client class $NutchClient$ has three public static APIs:

- $search$ – Returns a list of Maps, each map having values for keys "title," "description," "cache_uri," and "link." The title is the web page title, the description is an HTM snippet showing search terms in original web page text, the cache URI is a Nutch cache of the original web page, and the link is the URL to the matched web page.
- $searchGetCache$ – Like $search$ but each Map also contains a key "cache_content" with a value equal to the cached HTML for the original web page.
- $getCacheContent$ – Use this API if you first used $search$ and later want the cached web page.

The implementation is in the file src-index-search/NutchClient.java. Here are a few code snippets showing the public APIs:

```
  static public List<Hashtable<String,String>>
```

```
    searchGetCache(String opensearch_url, String query)
                   throws IOException,
                          ParserConfigurationException,
                         SAXException {
  return search_helper(opensearch_url, query, true);
}
static public List<Hashtable<String,String>>
   search(String opensearch_url, String query)
                   throws IOException,
                          ParserConfigurationException,
                          SAXException {
  return search_helper(opensearch_url, query, false);
}
static public String
     getCacheContent(String cache_uri)
                     throws IOException {
  URL url = new URL(cache_uri);
  URLConnection uc = url.openConnection();
  return new Scanner(uc.getInputStream()).
             useDelimiter("\\Z").next();
}
```

The implementation of the private helper method is (reformatted to fit the page width and with comments on the code):

```
static private List<Hashtable<String,String>>
     search_helper(String opensearch_url,
                   String query,
                   boolean return_cache) throws ... {
  List<Hashtable<String,String>> ret =
            new ArrayList<Hashtable<String,String>>();
```

We are using a REST style call so we need to URL encode the search terms. This involves replacing space characters with "+," etc. A search for "Java AI" using a Nutch server on my local laptop on port 8080 would look like:

http://localhost:8080/opensearch?query=Java+AI

```
  String url_str = opensearch_url + "?query=" +
                   URLEncoder.encode(query, "UTF-8");
  URL url = new URL(url_str);
  URLConnection uc = url.openConnection();
```

```
BufferedInputStream bis =
     new BufferedInputStream(uc.getInputStream());
```

While I usually prefer SAX XML parsers for less memory use and efficiency, it is easier for small XML payloads just to use the DOM-based APIs:

```
DocumentBuilder docBuilder =
     DocumentBuilderFactory.newInstance().
                              newDocumentBuilder();
Document doc  = docBuilder.parse(bis);
doc.getDocumentElement().normalize();
```

Here we use the DOM XML APIs to get all "item" tags and for each "item" tag get the text for the child nodes:

```
NodeList listItems = doc.getElementsByTagName("item");
int numItems = listItems.getLength();
for (int i=0; i<numItems; i++) {
  Node item = listItems.item(i);
  Hashtable<String,String> new_item =
           new Hashtable<String,String>();
  ret.add(new_item);
  NodeList item_data = item.getChildNodes();
  int num = item_data.getLength();
  for (int n=0; n<num; n++) {
    Node data = item_data.item(n);
    String name = data.getNodeName();
```

Nutch returns many extra parameters encoded as items that we do not need. Here we just keep what we need:

```
    if (name.equals("title") ||
        name.equals("description") ||
        name.equals("link")) {
      new_item.put(name, data.getTextContent());
    }
    if (name.equals("nutch:cache")) {
      new_item.put("cache_uri", data.getTextContent());
    }
  }
```

We may want to optionally make another web service call to get the cached web page for this search result. Doing this approximately doubles the time required for a search query:

```
      if (return_cache &&
         new_item.get("cache_uri")!=null) {
        new_item.put("cache_content",
             getCacheContent(new_item.get("cache_uri")));
      }
    }
    return ret;
  }
```

Here is a sample use of the client class:

```
  List<Hashtable<String,String>> results =
    NutchClient.search(
       "http://localhost:8080/opensearch", "Java");
  System.out.println("results: " + results);
```

and the output (edited for brevity):

```
results:
[{cache_uri=http://localhost:8080/cached.jsp?idx=0&id=0,
  link=http://knowledgebooks.com/,
  description= ... Java AI ...,
  title=Knowledgebooks.com: AI Technology for ...},
 {cache_uri=http://localhost:8080/cached.jsp?idx=0&id=1,
  link=http://knowledgebooks.com/license.txt,
  description= .. using <span class="highlight">Java ..,
  title=http://knowledgebooks.com/license.txt}]
```

The average time for a Nutch client web service call on my MacBook is 130 milliseconds when I ran both Tomcat and the Nutch web services client are on the same laptop. Average response times will only increase slightly when the client and the server are on the same local area network. Average response times will be longer and less predictable when using any of the public OpenSearch servers on the Internet.

What can you use a search client for? Here are a few ideas based on my own work projects:

- Roughly determine if two words or phrases are associated with each other by concatenating the words or phrases and counting the number of search results for the combined search query.
- Determine if a product name or ID code is spelled correctly or if a company carries a product by setting up a custom Nutch instance that only spiders the

company's web site(s). Always follow the terms and conditions of a web site when setting up a spider.

- Improve search results by adding a list of project-specific synonyms to the search client. Expand search terms using the synonym list.
- If you need access to public information, spider the information infrequently and then perform local search queries and use the local page caches.

For very little effort you can set up Nutch server instances that spider specific information servers. You can often add significant value to application programs by adding search functionality and by using Nutch you can locally control the information.

11 Conclusions

The material in this book was informed by my own work writing software for information processing. If you enjoyed reading it and you make practical use of at least some of the material I covered, then I consider my effort to be worthwhile.

Writing software is a combination of a business activity, promoting good for society, and an exploration to try out new ideas for self improvement. I believe that there is sometimes a fine line between spending too many resources tracking many new technologies versus getting stuck using old technologies at the expense of lost opportunities. My hope is that reading this book was an efficient and pleasurable use of your time in learning some new techniques and technologies that you had not considered before.

When we can expend resources to try new things it is almost always best to perform many small experiments and then dig deeper into areas that have a good chance for providing high value and capturing your interest. "Fail fast" is a common meme but failure that we do not learn from is a waste.

I have been using the Java platform from the very beginning and although I also use many other programming languages in my work and studies, both the Java language and platform provide high efficiency, scalability, many well-trained developers, and a wealth of existing infrastructure software and libraries. Investment in Java development also pays when using alternative JVM languages like JRuby, Scala, and Clojure.

If we never get to meet in person or talk on the telephone, then I would like to thank you now for taking the time to read this book.

Index