CHAPTER 4

Deterministic-Finite-Automata Applications

We have seen how DFAs can be used to define formal languages. In addition to this formal use, DFAs have practical applications. DFA-based pieces of code lie at the heart of many commonly used computer programs.

4.1 An Overview of DFA Applications

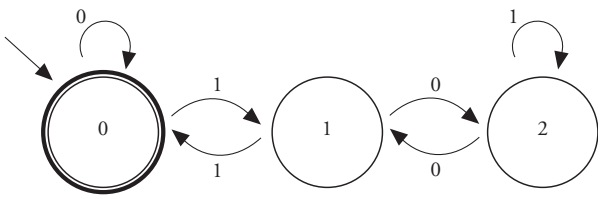
Deterministic finite automata have many practical applications:

- Almost all compilers and other language-processing systems use DFA-like code to divide an input program into tokens like identifiers, constants, and keywords and to remove comments and white space.
- For many applications that accept typed commands, the command language is quite complex, almost like a little programming language. Such applications use DFA-like code to process the input command.
- Text processors often use DFA-like code to search a text file for strings that match a given pattern. This includes most versions of Unix tools like awk, egrep, and Procmail, along with a number of platform-independent systems such as MySQL.
- Speech-processing and other signal-processing systems often use a DFA-like technique to transform an incoming signal.
- Controllers use DFA-like techniques to track the current state of a wide variety of finite-state systems, from industrial processes to video games. They can be implemented in hardware or in software.

Sometimes, the DFA-like code in such applications is generated automatically by special tools such as lex, which we discuss more later. In this chapter we concentrate on examples of DFA-like code constructed by hand.

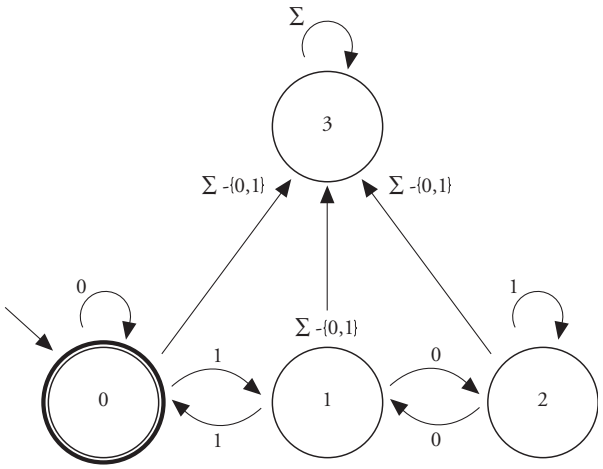
4.2 A DFA-based Text Filter in Java

The last chapter introduced a DFA that accepts strings that are binary representations of numbers that are divisible by three. Here is that DFA again:



Now let's see how this DFA can be implemented in the Java programming language.

The first thing to deal with is the input alphabet. The DFA above uses the alphabet {0, 1}, which is the alphabet of interest for this problem. But the program will work with a typed input string, so we do not have the luxury of restricting the alphabet in this way. The program should accept `"011"` (a representation for the number 3) and reject `"101"` (a representation for the number 5), but it must also properly reject strings like `"01i"` and `"fred"`. The alphabet for the Java implementation must be the whole set of characters that can occur in a Java string—that is, the whole set of values making up the Java `char` type. The DFA we actually implement will have four states, like this:



An object of the `Mod3` class represents such a DFA. A `Mod3` object has a current state, which is encoded using the integers 0 through 3. The class definition begins like this:

```
/**
 * A deterministic finite-state automaton that
 * recognizes strings that are binary representations
 * of natural numbers that are divisible
 * by 3. Leading zeros are permitted, and the
 * empty string is taken as a representation for 0
 * (along with "0", "00", and so on).
 */
public class Mod3 {

  /*
   * Constants q0 through q3 represent states, and
   * a private int holds the current state code.
   */
  private static final int q0 = 0;
  private static final int q1 = 1;
  private static final int q2 = 2;
  private static final int q3 = 3;

  private int state;
```

The `int` variables `q0`, `q1`, `q2`, and `q3` are `private` (visible only in this class), `static` (shared by all objects of this class), and `final` (not permitted to change after initialization).

In effect, they are named constants. The `int` field named `state` will hold the current state of each `Mod3` object.

The next part of this class definition is the transition function, a method named `delta`.

```
/**
 * The transition function.
 * @param s state code (an int)
 * @param c char to make a transition on
 * @return the next state code
 */
static private int delta(int s, char c) {
  switch (s) {
    case q0: switch (c) {
          case '0': return q0;
          case '1': return q1;
          default: return q3;
        }
    case q1: switch (c) {
          case '0': return q2;
          case '1': return q0;
          default: return q3;
        }
    case q2: switch (c) {
          case '0': return q1;
          case '1': return q2;
          default: return q3;
        }
    default: return q3;
  }
}
```

The `delta` method is declared `private` (cannot be called from outside the class) and `static` (is not given an object of the class to work on). It is thus a true function, having no side effects and returning a value that depends only on its parameters. It computes the transition function shown in the previous DFA diagram.

Next, the class defines methods that modify the `state` field of a `Mod3` object. The first resets it to the start state; the second applies the transitions required for a given input string.

```
  /**
   * Reset the current state to the start state.
   */
  public void reset() {
    state = q0;
  }

  /**
   * Make one transition on each char in the given
   * string.
   * @param in the String to use
   */
  public void process(String in) {
    for (int i = 0; i < in.length(); i++) {
      char c = in.charAt(i);
      state = delta(state, c);
    }
  }
```

The `process` method makes one transition on each character in the input string. Note that `process` handles the empty string correctly—by doing nothing.

The only other thing required is a method to test whether, after processing an input string, the DFA has ended in an accepting state:

```
  /**
   * Test whether the DFA accepted the string.
   * @return true if the final state was accepting
   */
  public boolean accepted() {
    return state==q0;
  }
}
```

That is the end of the class definition. To test whether a string `s` is in the language defined by this DFA, we would write something like

```
Mod3 m = new Mod3();
m.reset();
m.process(s);
if (m.accepted()) ...
```

To demonstrate the `Mod3` class we will use it in a Java application. This is a simple filter program. It reads lines of text from the standard input, filters out those that are not binary representations of numbers that are divisible by three, and echoes the others to the standard output.

```
import java.io.*;

/**
 * A Java application to demonstrate the Mod3 class by
 * using it to filter the standard input stream. Those
 * lines that are accepted by Mod3 are echoed to the
 * standard output.
 */
public class Mod3Filter {
  public static void main(String[] args)
        throws IOException {

    Mod3 m = new Mod3(); // The DFA
    BufferedReader in =  // Standard input
      new BufferedReader(new InputStreamReader(System.in));

    // Read and echo lines until EOF

    String s = in.readLine();
    while (s!=null) {
      m.reset();
      m.process(s);
      if (m.accepted()) System.out.println(s);
      s = in.readLine();
    }
  }
}
```

To test this program, we can create a file named `numbers` containing the numbers zero through ten in binary:

```
0
1
10
11
100
101
110
111
1000
1001
1010
```

After compiling `Mod3Filter` (and `Mod3`), we can use it on the file `numbers` to filter out all the numbers not divisible by 3. On a Unix system, the command `java Mod3Filter < numbers` produces this output:

```
0
11
110
1001
```

4.3 Table-driven Alternatives

The `Mod3` class implements the transition function in the obvious way, as a Java method `delta`. The `delta` method branches on the current state, and then within each alternative branches again on the current character.

An alternative implementation is to encode the transition function as a table, a two-dimensional array `delta`, indexed using the current state and symbol. Instead of calling the function `delta(s,c)`, passing it the state `s` and character `c` as parameters, we perform an array reference `delta[s,c]`. Array references are generally faster than function calls, so this should make the DFA run faster. For example, the `process` method could look something like this:

```
static void process(String in) {
  for (int i = 0; i < in.length(); i++) {
    char c = in.charAt(i);
    state = delta[state, c];
  }
}
```

Of course, the array `delta` must first be initialized with the appropriate transitions, so that `delta[q0,'0']` is `q0`, `delta[q0,'1']` is `q1`, and so on. To avoid the possibility of the reference `delta[state,c]` being out of bounds, `delta` will have to be initialized with a very large array. The program uses only 4 values for `state`, but there are 65,536 possible values for `c`! That is because Java uses Unicode, a 16-bit character encoding, for the `char` type. Depending on the source of the input string, we may be able to restrict this considerably—we may know, for example, that the characters are 7-bit ASCII (with values 0 through 127). Even so, we will have to initialize the array so that `delta[state,c]` is `q3` for every value of `c` other than `'0'` and `'1'`.

Instead of using a very large array, we could use a small array but handle the exception that occurs when the array reference is out of bounds. In Java this is the `ArrayIndexOutOfBoundsException`; the `process` method could catch this exception and use the state `q3` as the next state whenever it occurs. The definition of the `delta` array and the `process` method would then be the following:

```
/*
 * The transition function represented as an array.
 * The next state from current state s and character c
 * is at delta[s,c-'0'].
 */
static private int[][] delta =
   {{q0,q1},{q2,q0},{q1,q2},{q3,q3}};

/**
 * Make one transition on each char in the given
 * string.
 * @param in the String to use
 */
public void process(String in) {
  for (int i = 0; i < in.length(); i++) {
    char c = in.charAt(i);
    try {
      state = delta[state][c-'0'];
    }
    catch (ArrayIndexOutOfBoundsException ex) {
      state = q3;
    }
  }
}
```

This is a reasonable way to solve the problem by hand. Automatically generated systems usually use the full table with an element for every possible input character. One reason for this is that when the full array is used, `process` need contain no reference to individual states or characters. This way, any DFA can be implemented using the same `process` code, just by substituting a different transition table.

Incidentally, the transition table is usually stored in a more compressed form than we have shown. Our implementation used a full 32-bit `int` for each entry in the table, which is quite wasteful. It would be relatively easy to implement this using one byte per entry, and it would be possible to use even less, since we really need only two bits to represent our four possible states. The degree of compression chosen is, as always, a trade-off: heavily compressed representations take less space, but using them slows down each table access and thus slows down each DFA transition.

Exercises

EXERCISE 1

Reimplement `Mod3` using the full transition table. Assume that the characters in the input string are in the range 0 through 127.

EXERCISE 2

Using the DFA-based approach, write a Java class `Mod4Filter` that reads lines of text from the standard input, filters out those that are not binary representations of numbers that are divisible by four, and echoes the others to the standard output.

EXERCISE 3

Using the DFA-based approach, write a Java class `ManWolf` that takes a string from the command line and reports whether or not it represents a solution to the man-wolf-goat-cabbage problem of Chapter 2. For example, it should have this behavior:

```
> java ManWolf gncgwng
That is a solution.
> java ManWolf ggggggggg
That is not a solution.
```