Chapter Four: DFA 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.

Outline

- 4.1 DFA Applications
- 4.2 A DFA-Based Text Filter in Java
- 4.3 Table-Driven Alternatives

DFA Applications

- Programming language processing
  - Scanning phase: dividing source file into "tokens" (keywords, identifiers, constants, etc.), skipping whitespace and comments
- Command language processing
  - Typed command languages often require the same kind of treatment
- Text pattern matching
  - Unix tools like awk, egrep, and sed, mail systems like ProcMail, database systems like MySQL, and many others

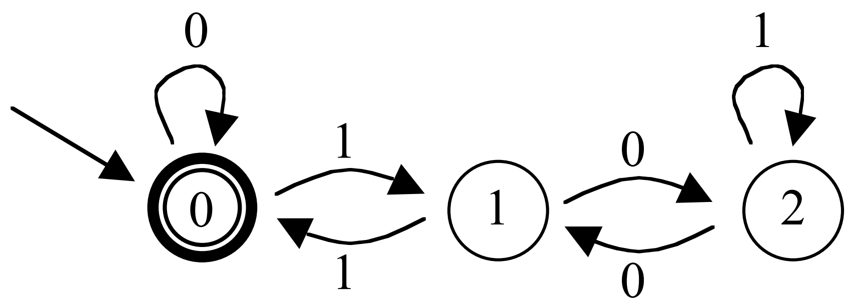
More DFA Applications

- Signal processing
  - Speech processing and other signal processing systems use finite state models to transform the incoming signal
- Controllers for finite-state systems
  - Hardware and software
  - A wide range of applications, from industrial processes to video games

Outline

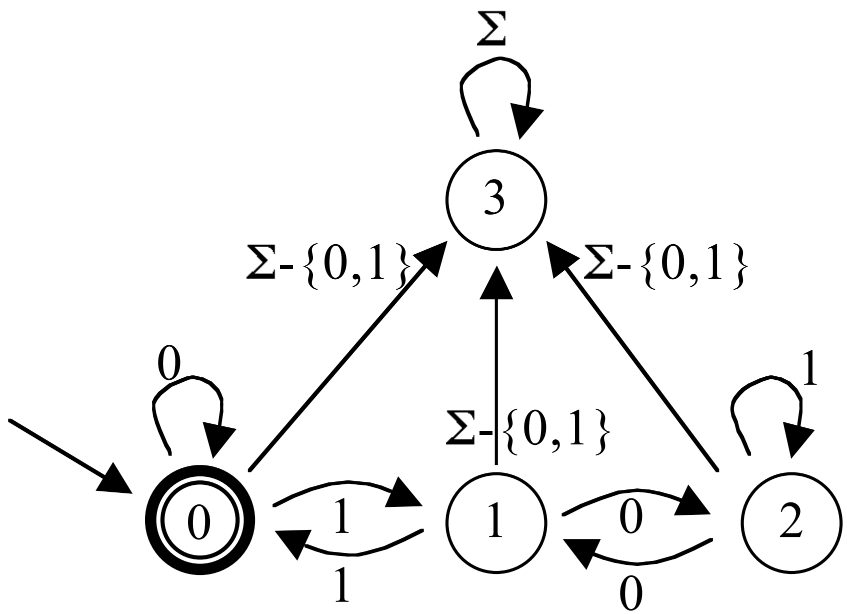
- 4.1 DFA Applications
- 4.2 A DFA-Based Text Filter in Java
- 4.3 Table-Driven Alternatives

The Mod3 DFA, Revisited



- We saw that this DFA accepts a language of binary strings that encode numbers divisible by 3
- We will implement it in Java
- We will need one more state, since our natural alphabet is Unicode, not {0,1}

The Mod3 DFA, Modified



- Here, Σ is the Unicode character set
- The DFA enters the non-accepting trap state on any symbol other than 0 or 1

```java
/**
 * A deterministic finite-state automaton that
 * recognizes strings that are binary
 * representations of integers 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;
```

```
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;
    }
}
```

```
/**
  * 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);
   }
 }
```

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

Usage example:

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

```
 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();
    }
  }
}
```

```
C:\>type numbers
000
001
010
011
100
101
110
111
1000
1001
1010

C:\>java Mod3Filter < numbers
000
011
110
1001

C:\>
```

Outline

- 4.1 DFA Applications
- 4.2 A DFA-Based Text Filter in Java
- 4.3 Table-Driven Alternatives

Making Delta A Table

- We might want to encode delta as a two-dimensional array
- Avoids method invocation overhead
- Then **`process`** could look like this:

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

Keeping The Array Small

- If `delta[state,c]` is indexed by state and symbol, it will be big: 4 by 65536!
- And almost all entries will be 3
- Instead, we could index it by state and integer, 0 or 1
- Then we could use exception handling when the array index is out of bounds

```java
/*
 * 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;
    }
  }
}
```

Tradeoffs

- Function or table?
- Truncated table or full table?
  - By hand, a truncated table is easier
  - Automatically generated systems generally produce the full table, so the same **`process`** can be used for different DFAs
- Table representation
  - We used an int for every entry: wasteful!
  - Could have used a byte, or even just two bits
  - Time/space tradeoff: table compression saves space but slows down access