Finite State Machines 1

95-771 Data Structures and Algorithms for Information Processing

Some Results First

Computing Model	Finite Automata	Pushdown Automata	Linear Bounded Automata	Turing Machines
Language Class	Regular Languages	Context-Free Languages	Context-Sensitive Languages	Recursively Enumerable Languages
Non-determinism	Makes no difference	Makes a difference	No one knows	Makes no difference

Alphabets

- Σ (sigma): a finite (non-empty) set of symbols called the *alphabet.*

- Each symbol in Σ is a *letter.*

- Letters in the alphabet are usually denoted by lower case letters: a, b, c, …

Strings

- A *word* w is a string of letters from Σ in a linear sequence.
- We are interested only in finite words (bounded length).
- $|w|$ denotes the *length* of word w.
- The *empty string* contains no letters and is written as ε.

Languages

- A *language* L is a set (finite or infinite) of words from a given Σ.

- The set of all strings over some fixed alphabet Σ is denoted by Σ*.
- For example, if Σ = {a},

  then Σ* = {ε, a, aa, aaa, …}.

Languages

- The set of all strings of length i over some fixed alphabet Σ is denoted by Σ^i.
- For example, let $\Sigma = \{a, b\}$.
- Then $L = \Sigma^2 = \{aa, ab, ba, bb\}$ is the set of words w such that $|w| = 2$.

Operations on Words and Languages

- Concatenation: putting two strings together

  $x = aa; y=bb; x.y = xy = aabb$

- Power: concatenating multiple copies of a letter or word

  $a^n = a.a^{n-1}; a^1 = a; a^2 = a.a;$ etc.

  $x = ab; x^3 = ababab$

- Kleene Star: zero or more copies of a letter or word

  $a^* = \{\varepsilon, a, aa, aaa, \ldots\}$

  $x = ab; x^* = \{\varepsilon, ab, abab, ababab, \ldots\}$

Deterministic, Finite State Automata

- A *finite-state automaton* comprises the following elements:
  - A sequence of *input symbols* (the input "tape").
  - The current location in the input, which indicates the current input symbol (the read "head").
  - The current *state* of the machine (denoted $q_0, q_1, \ldots, q_n$).
  - A *transition function* which inputs the current state and the current input, and outputs a new (next) state.

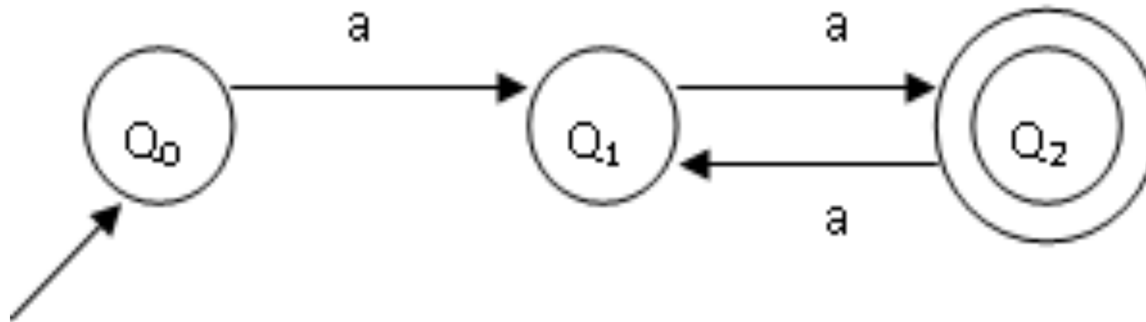
During computation,

- The FSA begins in the start state (usually, q0).
- At each step, the transition function is called on the current input symbol and the current state, the state is updated to the new state, and the read head moves one symbol to the right.
- The end of computation is reached when the FSA reaches the end of the input.

One or more states may be marked as final states, such that the computation is considered successful if and only if computation ends in a final state.

An Example

- An FSA can be represented graphically as a directed graph, where the nodes in the graph denote states and the edges in the graph denote transitions. Final states are denoted by a double circle.
- For example, here is a graphical representation of a DFSA that accepts the language $L = \{a^{2n} : n \geq 1\}$:



- Input: aaa

  States: q_0, q_1, q_2, q_1 (not accepted)

- Input: aaaa

  States: q_0, q_1, q_2, q_1, q_2 (accepted)

DFSA Definition

A DFSA can be formally defined as

$A = (Q, \Sigma, \partial, q_0, F)$:

- Q, a finite set of states
- Σ, a finite alphabet of input symbols
- $q_0 \in Q$, an initial start state
- $F \subseteq Q$, a set of final states
- ∂ (delta): $Q \times \Sigma \rightarrow Q$, a transition function

Transition function - ∂

- We can expand the notion of ∂ on letters to ∂ on words, ∂_w, by using a recursive definition:
- $\partial_w : Q \times \Sigma^* \rightarrow Q$ - (a function of (state, word) to a state)
- $\partial_w(q,\varepsilon) = q$ - (in state q, output state q if word is ε)
- $\partial_w(q,xa) = \partial(\partial_w(q,x),a)$ - (otherwise, use ∂ for one step and recurse)

Language Recognition

- For an automaton A, we can define the language of A:
  - $L(A) = \{w \in \Sigma^* : \partial_w(q_0,w) \in F\}$
  - L(A) is a subset of all words w of finite length over Σ, such that the transition function $\partial_w(q_0,w)$ produces a state in the set of final states (F).
  - Intuitively, if we think of the automaton as a graph structure, then the words in L(A) represent the "paths" which end in a final state. If we concatenate the labels from the edges in each such path, we derive a string $w \in L(A)$.

Carnegie Mellon

Implementing DFSA in Java (a first attempt)

- Implementing a DFSA in Java has some similarities to implementing a graph structure. As mentioned earlier, the states in a DFSA correspond to nodes in a graph, and the transitions correspond to edges in a graph.
- First, let's consider how to implement the transition function, ∂. Recall that ∂(<state>,<letter>) = <state> . So, for any given state q, we need to know if there is a transition to the other states, and if so, what letter of the alphabet must be read for the transition to occur.

- If each vertex in a directed graph has at most one edge leading to another vertex (possibly the vertex itself) then we can model the graph using a two-dimensional array.
- Suppose we want to model a DFSA in this manner. For each state in the DFSA, there is a row in the array; each column in that row corresponds to a (possible) transition to another state. Each cell is empty (null) if there is no such transition; otherwise, a cell contains the letter of the alphabet which must be read for the transition to be valid.
- Assuming we name the array *delta*, then *delta*(m,n) == null if there is no transition from Q_m to Q_n, and *delta*(m,n) == <letter> if $\partial(Q_m, \text{<letter>}) = Q_n$.

What is wrong with this approach?

The DFSA on the previous page has 3 states, so it can be modeled by a 3x3 array:

	Q0	**Q1**	**Q2**	**array**	**transition function**
Q0	null	a	null	*delta*(0,1) = a	$\partial(Q_0,a) = Q_1$
Q1	null	null	a	*delta*(1,2) = a	$\partial(Q_1,a) = Q_2$
Q2	null	a	null	*delta*(2,1) = a	$\partial(Q_2,a) = Q_1$

- Note that this works only when there is a single transition from one state to another. If Σ were expanded to contain two characters rather than one, and if there were two transitions leading from Q_0 to say, Q_1, then we could not use this implementation.

- Before we write a Java class for this DFSA, we have one more thing to consider.
  - How can we represent the set of final states?
- A straightforward solution is to use a one-dimensional boolean array, *final*, of length n (assuming we have n states). Then *final*(i) = true if q_i is a final state, and *final*(i) = false otherwise.

```
public class DFSA {

   private boolean[] finalStates;
   private char[][]   delta;
   private int          startState;
   private int          currentState;
   private int          totalStates;

   public DFSA (int n, int start) {
      totalStates = n;
      finalStates = new boolean[totalStates];
     delta = new char[totalStates][totalStates];
     startState = start;
   }

   private boolean isFinal () {
     return finalStates[currentState];
   }

   public void addTransition (int fromState, int toState, char letter) {
     delta[fromState][toState] = letter;
   }
   public void addFinalState (int q) {
      finalStates[q] = true;
   }
```

```
public boolean isAccepted (String s) {
    currentState = startState;
    readingSymbols:
      for (int i = 0; i < s.length(); i++) {
         System.out.println(" current state: "+currentState);
         System.out.println(" next symbol: "+s.charAt(i));
         for (int j = 0; j < totalStates; j++) {
            if (delta[currentState][j] == s.charAt(i)) {
               currentState = j;
               continue readingSymbols;
            }
         }
         System.out.println(" d("+currentState+","+s.charAt(i)+") is undefined");
         return false;
      }
    System.out.println(" final state: "+currentState);
    return isFinal();
  }
```

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

    // Define the three-state DFSA from the handout.

    DFSA a = new DFSA(3,0);
    a.addTransition(0,1,'a');
    a.addTransition(1,2,'a');
    a.addTransition(2,1,'a');
    a.addFinalState(2);

    // For each input string, check value of isAccepted()

    for (int i = 0; i < args.length; i++) {
       System.out.println("\nInput String: "+args[i]);
       if (a.isAccepted(args[i])) {
          System.out.println("Accepted");
       } else {
          System.out.println("Rejected");
       }
    }
  }

}
```

```
> java DFSA aaa aaaa aabaa

Input String: aaa
 current state: 0
 next symbol: a
 current state: 1
 next symbol: a
 current state: 2
 next symbol: a
 final state: 1
Rejected

Input String: aaaa
 current state: 0
 next symbol: a
 current state: 1
 next symbol: a
 current state: 2
 next symbol: a
 current state: 1
 next symbol: a
 final state: 2
Accepted

Input String: aabaa
 current state: 0
 next symbol: a
 current state: 1
 next symbol: a
 current state: 2
 next symbol: b
 d(2,b) is undefined
Rejected
```

Let's consider making certain modifications to the program above so that it allows for multiple transitions from one state to another based on a Σ with more than one character.

There are several ways this could be done.

- One might use a large two-dimensional array whose rows are indexed with state numbers and whose columns are indexed with all the characters found in Σ. This would give O(1) transition lookups but would be inefficient with respect to space.
- Another approach would be to use an adjaceny set representation. A single dimensional array, say states, could be indexed based on the state number. The element at states[i] would contain a set of <character><next state> pairs. Each set would allow fast <next state> lookups given the character symbol from Σ. The set could be implemented with a linked list or, perhaps, with a balanced tree.

- Consider modifying the program above so that it uses an adjaceny set representation to compute $\partial(Q_m$,<letter>). The set could be implemented as a digital search tree. The DST class would allow for the insertion and search of (<letter>,<next-state>) pairs. The <letter> part of the pair would be the search key. When inserting or searching, the decision to move left or right in the search tree would be based on the bits of the key. The implementation effort for this type of data structure is about the same as binary search trees but usually has better performance.

Another Example

Consider running such a program that would model the DFSA below. Such a program might read a series of strings from the command line (just like the program above) and tell the user whether the machine below accepts or rejects each string.

$Q = \{ q_0, q_1, q_2 \}$

$\Sigma = \{ R,0,1,2 \}$

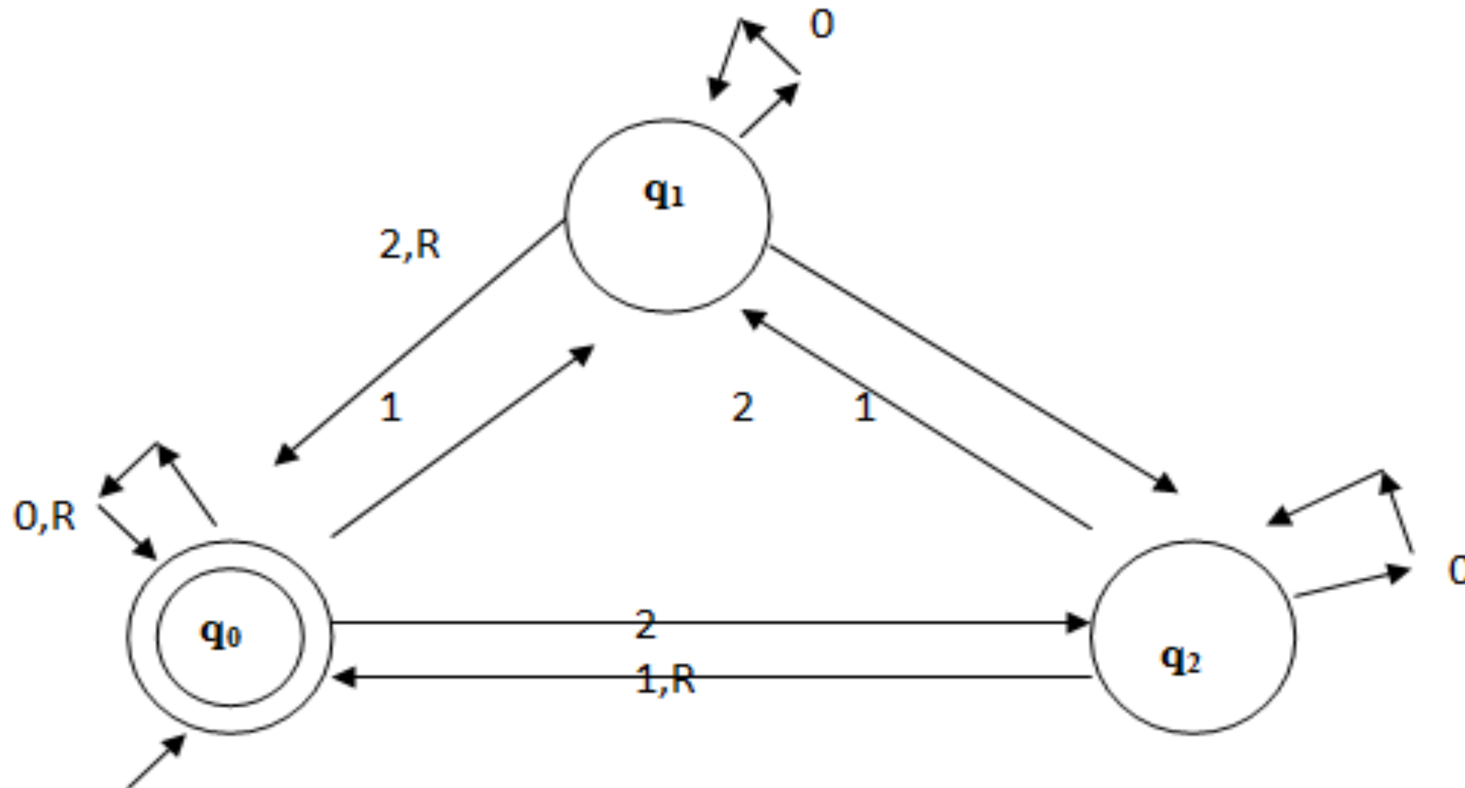
q_0: the start state

$F = \{ q_0 \}$

∂ (delta): $Q \times \Sigma \rightarrow Q$

This automaton keeps a running count of the sum of the numerical input symbols it reads, modulo 3. Every time it receives the R (reset) symbol it resets the count to 0. It accepts the if the sum is 0, modulo 3, or in other words, if the sum is a multiple of 3.

This automaton is from "Introduction to the Theory of Computation" by Michael Sipser



Homework Questions (not to be turned in but to prepare for exam)

Give state diagrams for DFA's recognizing the following languages. $\Sigma = \{ 0,1 \}$.

1. { w | w begins with a 1 and ends with a 0 }
2. { w | w contains at least three 1's }
3. { w | the length of w is at most 5 }
4. { w | w contains at least two 0's and at most one 1. }
5. { w | w contains an even number of 0's, or exactly two 1's }
6. { w | w is not ε }