5 Finite-State Machines and Regular Languages

This section looks at a simple model of computation for solving decision problems: a finite-state machine, or FSM.

5.1 Intuitive idea of a FSM

Figure 5-1 shows a diagram, called a transition diagram, of FSM $M_1$. Each circle or double-circle is called a state. One of the states, marked by an arrow, is called the start state. A state with a double circle is called an accepting state and a state with a single circle is called a rejecting state.

The arrows between states are called transitions, and each transition is labeled by a member of the FSM’s alphabet $\Sigma$ (set \{a, b\} for $M_1$). For each state $q$ and each member $c$ of $\Sigma$, there must be exactly one transition going out of $q$ labeled $c$.

A FSM is used to recognize a language (a decision problem). To “run” a FSM on string $s$, start in the start state. Read each character, and follow the transition labeled by that character to the next state. On input "aabab", $M_1$ starts in state 1, then hits states 2, 1, 1, 2, 2, ending in state 2.

The end state determines whether the FSM accepts or rejects the string. Since state 2 is a rejecting state, $M_1$ rejects "aabab". It should be easy to see

Figure 5-1. Transition diagram of FSM $M_1$ that recognizes language \{s $\in$ \{a, b\}*$ | s has an even number of as\}. There are two states. State 1 is the start state. State 1 is an accepting state and state 2 is a rejecting state.
that $M_1$ accepts strings with an even number of $a$s and rejects strings with an odd number of $a$s.

A FSM $M$ with alphabet $\Sigma$ \textit{recognizes} the set

$$L(M) = \{s \mid s \in \Sigma \text{ and } M \text{ accepts } s\}.$$ 

For example, $L(M_1) = \{s \mid s \in \{a,b\}^* \text{ and has an even number of } a\}$.

Figures 5-2 and 5-3 show two finite-state machines $M_2$ and $M_3$ with alphabet $\{a,b\}$ where

$$L(M_2) = \{s \mid |s| \text{ is divisible by 3}\}$$
$$L(M_3) = \{\}$$

## 5.2 Designing FSMs

There is a simple and versatile way to design a FSM machine to recognize a selected language $L$. Associate with each state $q$ the set of strings $\text{Set}(q)$ that end on state $q$. For example, in machine $M_2$,

$$\text{Set}(0) = \{s \mid |s| \equiv 0 \pmod{3}\}$$
$$\text{Set}(1) = \{s \mid |s| \equiv 1 \pmod{3}\}$$
$$\text{Set}(2) = \{s \mid |s| \equiv 2 \pmod{3}\}$$
Your goals in designing a FSM that recognizes language $L$ are:

(a) Start by deciding what the states will be and what $\text{Set}(q)$ will be for each state. Make sure that, for each state $q$, either $\text{Set}(q) \subseteq L$ or $\text{Set}(q) \subseteq \overline{L}$.

(b) Make $q$ be an accepting state if $\text{Set}(q) \subseteq L$ and make $q$ a rejecting state if $\text{Set}(q) \subseteq \overline{L}$.

(c) Draw transitions so that, if $x \in \text{Set}(q)$ and there is a transition from state $q$ to state $q'$ labeled $a$, then $x \cdot a \in \text{Set}(q')$.

5.2.1 Example: even binary numbers

Figure 5-4 shows a FSM with alphabet \{0,1\} that accepts all even binary numbers. For example, it accepts "10010" and rejects "1101". $\text{Set}(0) = \{s \in \{0,1\}^* | s \text{ is an even binary number}\}$ and $\text{Set}(1) = \{s \in \{0,1\}^* | s \text{ is an odd binary number}\}$. The transitions are obvious: adding a 0 to the end of any binary number makes the number even, and adding a 1 to the end makes the number odd.

5.2.2 A FSM recognizing binary numbers that are divisible by 3

Figure 5-5 shows a FSM that recognizes binary numbers that are divisible by 3. For example, it accepts "1001" and "1100", since "1001" is the binary representation of 9 and "1100" is the binary representation of 12. But it rejects "100", the binary representation of 4.
Thinking of binary strings as representing numbers,

\[
\begin{align*}
\text{Set}(0) & = \{ n \mid n \equiv 0 \pmod{3} \} \\
\text{Set}(1) & = \{ n \mid n \equiv 1 \pmod{3} \} \\
\text{Set}(2) & = \{ n \mid n \equiv 2 \pmod{3} \}
\end{align*}
\]

Suppose that \( m \) is a binary number that is divisible by 3. Adding a 0 to the end doubles the number, so \( m \cdot 0 \) is also divisible by 3. Adding a 1 to \( m \) doubles \( m \) and adds 1. But modular arithmetic tells us that

\[
\begin{align*}
m \equiv 0 \pmod{3} & \rightarrow 2m \equiv 0 \pmod{3} \\
& \rightarrow 2m + 1 \equiv 1 \pmod{3}
\end{align*}
\]

so there is a transition from state 0 to state 1 on symbol 1.

### 5.2.3 Strings containing at least two \( a \)s and at most one \( b \).

Figure 5-6 shows a FSM that recognizes language

\[
\{ w \in \{a,b\}^* \mid w \text{ contains at least two } a \text{ and at most one } b \}
\]

The idea is to keep track of the number of \( a \)s (up to a maximum of 2) and the number of \( b \)s (up to a maximum of 2). That suggests that we need nine states: (0, 0), (0, 1), (0, 2), (1, 0), (1, 1), (1, 2), (2, 0), (2, 1) and (2, 2), where the first number is the count of \( a \)s and the second the count of \( b \)s, and 2 means at least 2. The accepting states and transitions should be obvious.

![Figure 5-5. A FSM recognizing binary numbers that are divisible by 3. An empty string is treated as 0.](image)
5.3 Definition of a FSM and the class of regular languages

The introduction above only shows transition diagrams, and does not adequately say exactly what a FSM is and how to determine the language that it recognizes. This section corrects that with a careful definition of both. The first definition says what a FSM is without saying about what it means to run it on a string.

5.3.1 Definition of a FSM

**Definition 5.1.** A *finite-state machine* is a 5-tuple \((\Sigma, Q, q_0, F, \delta)\). That is, it is described by those five parts.

- \(\Sigma\) is the machine’s alphabet.
- \(Q\) is a finite nonempty set whose members are called *states*.
- \(q_0 \in Q\) is called the *start state*.
- \(F \subseteq Q\) is the set of *accepting states*. 
• \( \delta : Q \times \Sigma \rightarrow Q \) is called the *transition function*.

Most of that should be clear from the transition diagrams that we have looked at. From state \( q \), if you read symbol \( a \), you go to state \( \delta(q, a) \). Notice that, because \( \delta \) is a function, there must be exactly one state to go to from state \( q \) upon reading symbol \( a \).

### 5.3.2 When does FSM \( M \) accept string \( s \)?

Consider a FSM \( M = (\Sigma, Q, q_0, F, \delta) \).

**Definition 5.2.** If \( q \in Q \) and \( x \in \Sigma^* \), then \( q : x \) is defined inductively as follows.

1. \( q : \varepsilon = q \).
2. If \( x = cy \) where \( c \in \Sigma \) and \( y \in \Sigma^* \) then \( q : x = \delta(q, c) : y \).

The idea is that \( q : x \) is the state that \( M \) reaches if it starts in state \( q \) and reads string \( x \). To find that out for string \( x = cy \), first find the state \( q' = \delta(q, c) \), then finish by finding \( q' : y \).

Every FSM \( M \) has a language \( L(M) \) that it recognizes, and the following definition says what that is.

**Definition 5.3.** \( L(M) = \{ x \in \Sigma^* \mid q_0 : x \in F \} \).

That is, \( M \) accepts string \( x \) if \( M \) reaches an accepting state when it is run on \( x \) starting in the start state, \( q_0 \).

### 5.3.3 The class of regular languages

**Definition 5.4.** Language \( A \) is *regular* if there exists a FSM \( M \) such that \( L(M) = A \).

We have see a few regular languages above, including \( \{ \} \) and the set of binary numbers that are divisible by 3.
5.4 A theorem about \( q : x \)

Notation \( q : x \) satisfies a certain kind of associativity.

**Theorem 5.1.** \( (q : x) : y = q : (xy) \).

**Proof.** The proof is by induction of the length of \( x \). The introduction to proofs does not cover proof by induction because this is the only such proof that we need. It suffices to

(a) show that \( (q : x) : y = q : (xy) \) for all \( q \) and \( y \) when \(|x| = 0\), and

(b) show that \( (q : x) : y = q : (xy) \) for an arbitrary nonempty string \( x \), under the assumption (called the *induction hypothesis*) that \( (r : z) : y = r : (zy) \) for any state \( r \), string \( y \) and string \( z \) that is shorter than \( x \).

**Case 1** \(|x| = 0\). That is, \( x = \varepsilon \). By definition, \( q : \varepsilon = q \). So

\[
(q : x) : y = q : y \\
= q : (xy)
\]

because, when \( x = \varepsilon \), \( xy = y \).

**Case 2** \(|x| > 0\). A nonempty string \( x \) can be broken into \( x = cz \) where \( c \) is the first symbol of \( x \) and \( z \) is the rest.

\[
(q : x) : y = (q : (cz)) : y \\
= (\delta(q, c) : z) : y \quad \text{by the definition of } q : (cz) \\
= \delta(q, c) : (zy) \quad \text{by the induction hypothesis} \\
= q : (czy) \quad \text{by the definition of } q : (czy) \\
= q : (xy) \quad \text{since } x = cz
\]

5.5 Closure results

A *closure* result tells you that a certain operation does not take you out of a certain set. For example, \( \mathbb{Z} \) is *closed under addition* because the sum of two integers is an integer. \( \mathbb{Z} \) is also *closed under multiplication*. But \( \mathbb{Z} \) is not closed under division, since \( 1/2 \) is not an integer.
The class of regular languages possesses some useful closure results.

**Definition 5.5.** Suppose that \( A \subseteq \Sigma^* \) is a language. The complement \( A \) of \( A \) is \( \Sigma^* - A \).

**Theorem 5.2.** The class of regular languages is closed under complementation. That is, if \( A \) is a regular language then \( \overline{A} \) is also a regular language. Put another way, for every FSM \( M \), there is another FSM \( M' \) where \( L(M') = \overline{L(M)} \). Moreover, there is an algorithm that, given \( M \), finds \( M' \). That is, the proof is constructive.

**Proof.** Suppose that \( M = (\Sigma, Q, q_0, F, \delta) \). Then \( M' = (\Sigma, Q, q_0, Q - F, \delta) \). That is, simply convert each accepting state to a rejecting state and each rejecting state to an accepting state.

\[ \diamond \]

**Theorem 5.3.** The class of regular languages is closed under intersection. That is, if \( A \) and \( B \) are regular languages then \( A \cap B \) is also a regular language. Put another way, suppose \( M_1 \) and \( M_2 \) are FSMs with the same alphabet \( \Sigma \). There is a FSM \( M' \) so that \( L(M') = L(M_1) \cap L(M_2) \). That is, \( M' \) accepts \( x \) if and only if both \( M_1 \) and \( M_2 \) accept \( x \). Moreover, there is an algorithm that takes parameters \( M_1 \) and \( M_2 \) and produces \( M' \).

**Proof.** The idea is to make \( M' \) simulate \( M_1 \) and \( M_2 \) at the same time. For that, we want a state of \( M' \) to be an ordered pair holding a state of \( M_1 \) and a state \( M_2 \). Recall that the cross product \( A \times B \) of two sets \( A \) and \( B \) is \( \{(a, b) \mid a \in A \land b \in B\} \).

Suppose that \( M_1 = (\Sigma, Q_1, q_{0,1}, F_1, \delta_1) \) and \( M_2 = (\Sigma, Q_2, q_{0,2}, F_2, \delta_2) \). Then \( M' = (\Sigma, Q', q'_0, F', \delta') \) where

\[
Q' = Q_1 \times Q_2 \\
q'_0 = (q_{0,1}, q_{0,2}) \\
F' = F_1 \times F_2 \\
\delta'((r, s), a) = (\delta_1(r, a), \delta_2(s, a))
\]

State \( (r, s) \) of \( M' \) indicates that \( M_1 \) is in state \( r \) and \( M_2 \) is in state \( s \). Transition function \( \delta' \) runs \( M_1 \) and \( M_2 \) each one step separately. Notice that the set \( F' \) of accepting states of \( M' \) contains all states \( (r, s) \) where \( r \) is an accepting
state of $M_1$ and $s$ is an accepting state of $M_2$. So $M'$ accepts $x$ if and only if both $M_1$ and $M_2$ accept $x$. 

\[ \text{Theorem 5.4.} \text{ The class of regular languages is closed under union. That is, if } A \text{ and } B \text{ are regular languages then } A \cup B \text{ is also a regular language.} \]

**Proof.** By DeMorgan’s laws for sets, 

\[ A \cup B = \overline{A} \cap \overline{B}. \]

By we know that the class of regular languages is closed under complementation and intersection.