Notes on Finite State Machines

Scribe: Peter Orselli   Lecturer/Editor: Chris Eppolito

27 April 2020

We want to mathematically analyze algorithms; we begin with a simple model of an algorithm, which allows both input and output.

Definition. A *finite state machine* is a sextuple $M = (S, I, O, t, w, s_0)$ with

1. a finite set S of *states* with a distinguished *initial state* $s_0 \in S$,
2. finite sets I and O of *input symbols* and *output symbols* respectively,
3. a *transition function* $t \colon S \times I \to S$,
4. an *output function* $w \colon S \times I \to O$, and

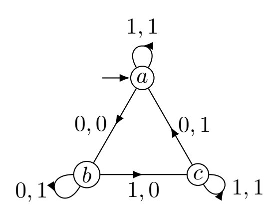
We sometimes abbreviate the name "finite state machine" to "state machine" or simply " machine".

Remark. This way of describing a machine is verbose! We can more compactly express them as directed graphs with some decorations. We represent states by vertices; the initial state is marked by an unmarked arrow pointing in. The transition and output functions are represented with directed edges with labels.

Example 1. We describe a finite state machine $M = (S, I, O, t, w, s_0)$. Let $S = \{a, b, c\}$, $s_0 = a$, and $I = \{0, 1\} = O$. We define t and w in the table below.

	t		w	
	0	1	0	1
a	b	a	0	1
b	b	c	1	0
c	a	c	1	1

We may represent this finite state machine as a digraph, given below.



Before continuing, we give some additional terminology to facilitate our discussion.

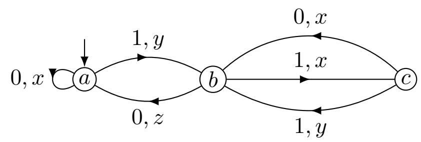
Definition. An *alphabet* is a finite set of symbols. A *word* or *string* in an alphabet A is a finite sequence of symbols of A. The set of all words in A is denoted A^*.

The *empty word*, denoted ϵ, is the unique word of length 0. A *bit string* is a word in $\{0, 1\}$.

For the purposes of the following discussion, we write a state machine M with initial state s_0 as a pair (M, s_0).[1] A machine (M, s_0) defines a function $f_{(M,s_0)} \colon I^* \to O^*$ (recursively) as follows. We define $f_{(M,s_0)}(\epsilon) = \epsilon$, and $f_{(M,s_0)}(a_1 a_2 \dots a_k) = w_M(s_0, a_1) f_{(M, t_M(w_1, s_0))}(a_2 \dots a_k)$ where $a_1, a_2, \cdots, a_k \in I$. Using the digraph representation, computing $f_M(a)$ amounts to following arrows labeled by the input characters and recording the corresponding output characters.

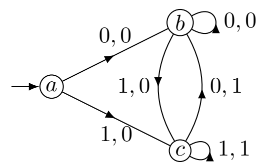
[1] We do so because we will need to change the state current state as we read; the easiest way to do so for this discussion is to consider the machine with a different initial state. We encode the overwrite as the second entry of our ordered pair.

Example 2. Consider the finite state machine M below.



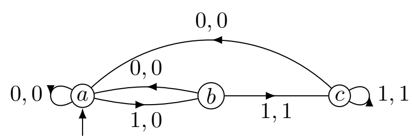
The function f_M maps (for example) $01010 \mapsto xyzyx$, $0010110 \mapsto xxyzyxx$, and $11110 \mapsto yxyxx$.

Example 3. We write a finite state machine for a unit delay of a bit string; the corresponding function adds a 0 prefix to a bit string and replicates the rest of the string, except for the last character.



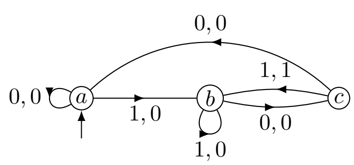
Intuitively if we are at state b, the machine has just read a 0; similarly, if we are at state c, then the machine has just read a 1. The transition function appropriately moves between states based on this idea. The write function then behaves appropriately, always writing a 0 when transitioning from state b, and always writing a 1 when transitioning from state c; the first step (transitioning from state a) always writes a 0 as a pad.

Example 4. We give a machine to recognize each 11 substring of a bit string, writing 1 upon recognition.



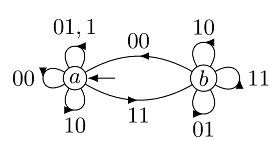
The states a, b, and c intuitively correspond to having read zero, one, and two 1's in sequence respectively. Thus we only write a 1 when transitioning into state c. We transition to state a whenever we read a 0.

Example 5. We give a machine to recognize each 101 substring of a bit string, writing 1 upon recognition.



The states a, b, and c intuitively correspond to having read zero, one, and two correct characters in sequence.

Example 6. We give a machine for performing binary addition; for this machine $I = \{0,1\}^2$ and $O = \{0,1\}$.



First we make a few notes on how to apply our machine. A bit string $x_0x_1 \dots x_n$ of length $n+1$ encodes the integer $m = \sum_{k=0}^{n} x_k 2^k$; this is the reverse of the binary representation of m. To add two integers m and n, encode them as above, add an end zero to both strings and pad the end of the shorter by enough zeroes that the two have the same length; say this procedure results in $m \rightsquigarrow x_0x_1 \dots x_k$ and $n \rightsquigarrow y_0y_1 \dots y_k$. Now apply the machine to the input string $z = (x_0y_0)(x_1y_1)\dots(x_ky_k)$. Intuitively state a means all addition carries are resolved, and state b means there is an unresolved carry; our padding ensures we resolve all carries.