Section 8.2: Finite-State Machines

April 21, 2008

Abstract

We model a machine as a set of states, inputs which lead to a change in state, a clock to synchronize the machine world, and outputs, which result from a particular state. We use tables and graphs to describe how the inputs relate to changes in state and the outputs of each state, then practice creating simple finite-state machines.

Finite-state machines can be used to recognize input, and we will look at the kinds of input that can be recognized, as well as construct the machines that recognize given input. Furthermore, some machines are overly complicated, in that we can simplify them and get the same operation. We will examine some ways in which we can "minimize" a finite-state machine.

1 Finite-State Machines

Definition: A finite-state machine M is a structure $[S, I, O, f_s, f_o]$ where The machine is initialized to start in state s_0, and the machine

Table 1: Elements of a finite-state machine.

S	finite set of states of the machine
I	input alphabet (finite set of symbols)
O	output alphabet (finite set of symbols)
f_s	$f_s : S \text{ x } I \rightarrow S$, the next-state function
f_o	$f_o : S \rightarrow O$, the output function

operates *deterministically* (meaning there is no randomness associated with its operation given a sequence of inputs).

We assume discrete times, synchronized by a clock, so that

$$f_s(state(t_i), input(t_i)) = state(t_{t+1})$$

and that

$$f_o(state(t_i)) = output(t_i)$$

We represent f_s and f_o by

- state tables (e.g. Table 8.1, p. 545/619)
- state graphs (e.g. Figure 8.1, p. 546/620)

A summary of these elements for Example 16, p. 545/619:

Table 2: Elements of finite-state machine of Example 16, p. 545/619.

S	$\{s_0, s_1, s_2\}$
I	$\{0,1\}$
O	$\{0,1\}$
f_s	$f_s(s_0, 0) = s_1$, $f_s(s_0, 1) = s_0$, etc.
f_o	$f_o(s_0) = 0$, $f_o(s_1) = 1$, $f_o(s_2) = 1$

Example: Practice 34, p. 546/620 . (First, create the state table from the state graph: what are $[S, I, O, f_s, f_o]$ in this example?)

(Table 8.1)

Input sequence : 1 1 0 0 1

Time	t_0	t_1	t_2	t_3	t_4	t_5
Input	1	1	0	0	1	-
State	s_0	s_0	s_0	s_1	s_2	s_0
Output	0	0	0	1	1	0

Example: Practice 36, p. 546/620

Present	Next state 0	1	2	Output
s_0	s_0	s_1	s_1	0
s_1	s_1	s_0	s_0	1

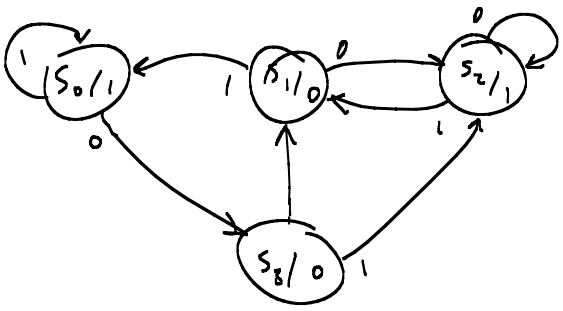
a)

b) Output given input 2 1 1 0 ?

Input	2	1	1	0	-
State	s_0	s_1	s_0	s_1	s_1
Output	0	1	0	1	1

Output

Example: Exercise 4, p. 563/637 1 1 0 1 1 0 0



Input	1	1	0	1	1	0	0	-
State	S_0	S_0	S_0	S_3	S_2	S_1	S_2	S_2
Output	1	1	1	0	1	0	1	1

output corresponding to our input string

the initial state

2 Construction of a machine: the Binary Adder

In section 7.2 we saw how one might create a logic network in hardware for the addition of binary numbers. We now consider how this can be incorporated into a finite-state machine which is analogous (p. 547/621).

We must specify the five elements of a finite-state machine: $[S, I, O, f_s, f_o]$. What is the set of states, what the set of inputs, what the set of outputs, and how are the functions f_s and f_o defined?

Example: Practice 37, p. 547/621

S_0 - out 0, carry 0
S_1 - out 0, carry 1
S_2 - out 1, carry 0
S_3 - out 1, carry 1

Present state	Next state 00	01	10	11	Output
S_0	S_0	S_2		S_1	0
S_1	S_2	S_1		S_3	0
S_2	S_0	S_1		S_1	1
S_3	S_2	S_1		S_3	1

Compute the sum

$$\begin{array}{r} 0\,1\,1\,1\,0\,1\,1\,0 \\ +\ 0\,1\,0\,1\,0\,1\,0\,1 \\ \hline 1\,1\,0\,0\,1\,0\,1\,1 \end{array}$$

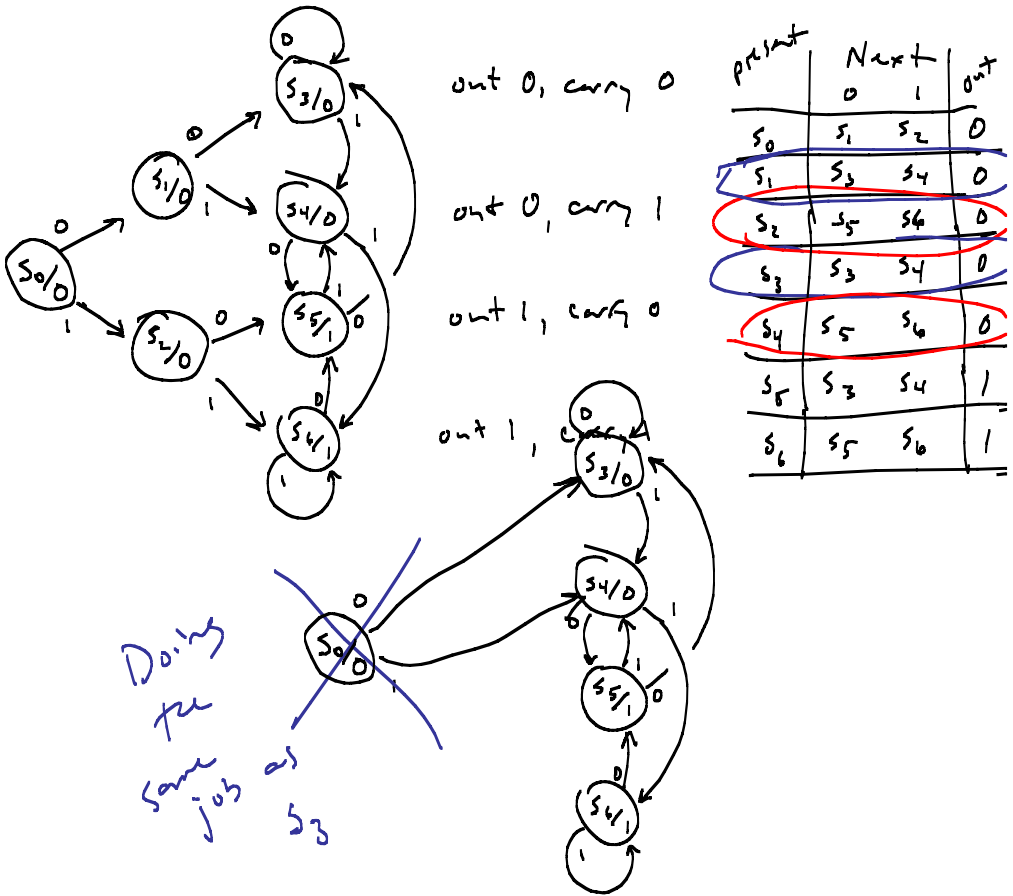
Input	01	10	11	00	11	10	11	00	-
State	S_0	S_2	S_2	S_1	S_2	S_1	S_1	S_3	S_2
Output	0	1	1	0	1	0	0	1	1

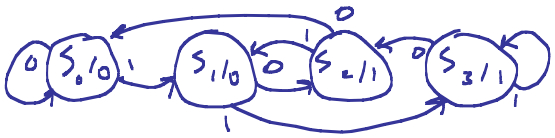
translates to 1 1 0 0 1 0 1 1
as a binary number

Example: Practice 38, p. 548/622

Now let's try something a little different:

Example: Exercise 15(a), p. 565/638 This is a modification, in some sense, of the binary adder. First of all, recognize that only one bit is being stored: the author intends in this problem that the first bit in the output sequence is the output of state s_0, in which the machine started. We need to "carry" the bit which we will write next time, and write the current bit. We'll solve this in two ways: in a sloppy way first, and then in a better way - illustrating the need to be able to minimize a finite-state machine.

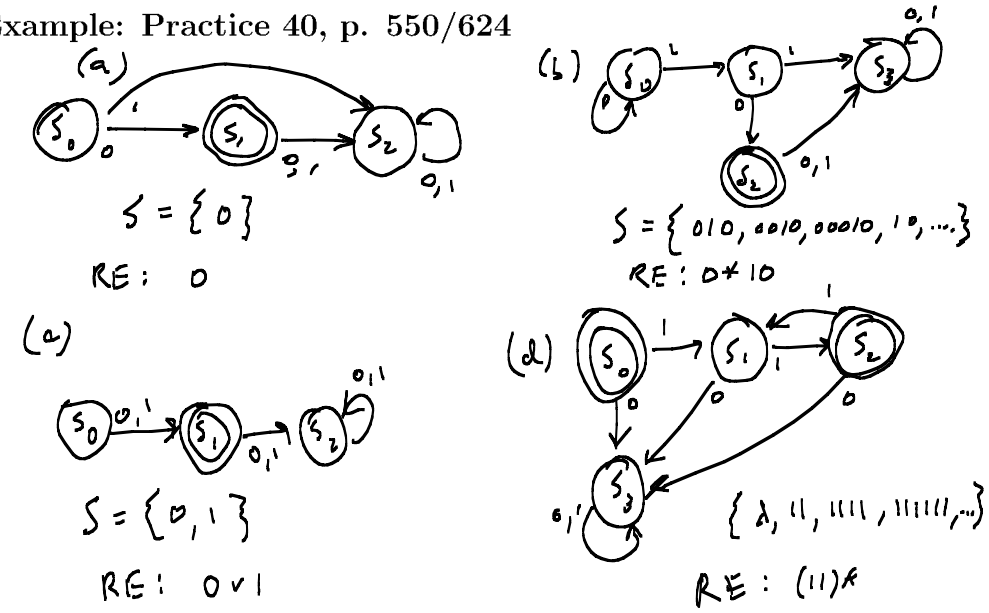




3 Recognition

Definition: Finite-State Machine Recognition A finite-state machine M with input alphabet I recognizes a subset S of I^* (the set of finite-length strings over the input alphabet I) if M, beginning in state s_0 and processing an input string α, **ends** in a final state (a state with output 1) if and only if $\alpha \in S$.

Example: Practice 40, p. 550/624



Notes:

- Note the emphasis on the word "ends": we assume that the input stops, and when the input stops the final output is a 1.
- Note also the "if and only if": this indicates that, if the output ends in a 1, then the string α is in S; and if string α is in S, then the output ends in a 1.

What kinds of input can a finite-state machine recognize? **Regular expressions**. **Regular expressions over** I are defined recursively by

1 the symbol $\emptyset$ and the symbol λ;

2 the symbol i for any $i \in I$; and

3 the expressions (AB), $(A \vee B)$, and $(A)^*$ if A and B are regular expressions.

Kleene's Theorem assures us that a finite-state machine can recognize a set S of input strings if and only if the set S is a regular set (that is, a set represented by a regular expression).

Since some very reasonable sets are not regular (e.g. $S = \{0^n 1^n\}$, where a^n stands for n copies of a), finite-state machines are obviously not sufficient to understand all of computation.

Examples of regular sets given by regular expressions:

- #20b. The set of all strings beginning with 000: $000(0 \vee 1)^*$
- #20e. The set of all strings ending in 110: $(0 \vee 1)^*110$
- #20f. The set of all strings containing 00: $(0 \vee 1)^*00(0 \vee 1)^*$
- #20d. The set of all strings consisting entirely of any number (including none) of 01 pairs or consisting entirely of two 1s followed by any number (including none) of 0s: $(01)^* \vee 110^*$
- #32b. The set of all strings of 0s and 1s having an odd number of 0s: $1^*01^*(01^*01^*)^*$.

Example: Exercise 19/23(~~c~~), p. 566/639 - recognition and minimization motivation d, e, f

f

#25/29

4 Machine Minimization

4.1 Unreachable States

One obvious way in which a machine can be minimized is if there is an **unreachable state**: if so, then that state can certainly be trimmed from the machine without any consequences (from the standpoint of output). For example: Table 8.3, p. 552/626; and Figure 8.7, p. 553/627.

Example: Practice 43, p. 553/627

S_3 & S_2 are unreachable — throw them out!

4.2 Equivalent States

It would be nice if we had some general way of minimizing a machine, however. It turns out that we can find a minimized machine by using the idea of equivalent states. The idea is that several redundant states might operate in such confusing fashion that it appears there's lots going on, when there's not!

In the first step, the unreachable states are removed. That's the easy part! Then we define

Equivalent States: two states s_i and s_j of M are **equivalent** if, for any $\alpha \in I^*$, $f_o(s_i, \alpha) = f_o(s_j, \alpha)$ where by the **awful notation** $f_o(s, \alpha)$ we mean the **sequence** of output which occurs given that we start in state s and receive input α.

(There is no way that our author should have used notation which seems to imply that f_o is somehow both a function from $S \rightarrow S$ and a function from S x $I^* \rightarrow S$, except that she's proving herself a computer scientist and an object-oriented one at that, and overloading the function f_o...).

In order to find equivalent states, we define the notion of **k-equivalency**: two states are k-equivalent if the machine matches output on an input of k symbols to the two states.

1 States having the same output symbol are 0-equivalent.

2 For 1-equivalency, we check two 0-equivalent states to see that the next-states under all input symbols (of length 1) are 0-equivalent.

3 For 2-equivalency, we check 1-equivalent states to see that the next-states under all input symbols (of length 1) are 1-equivalent - and hence equivalent for strings of length 2, total.

4 Etc.!

We iteratively step through equivalencies (from 0 on up): as soon as the states do not change, from k-equivalency to (k+1)-equivalency, then we have minimized our machine.

Best to look at an example!

Example: Exercise 41/53, p. 570/644

Present	Next 0	Next 1	Output
0	3	6	1
1	4	2	0
2	4	1	0
3	2	0	1
4	5	0	1
5	3	5	0
6	4	2	1

① Check for unreachable states (starting from 0); didn't find any.

② 0-equivalency - if they're not outputting the same thing, they can't be combined

$\{1,2,5\}$, $\{0,3,4,6\}$

③ 1-equivalency:

$\{1,2,5\}$, $\{0\}$, $\{3,4\}$, $\{6\}$

④ 2-equivalency $\underbrace{\{1,2,5\}}_{B}$, $\underbrace{\{0\}}_{A}$, $\underbrace{\{3,4\}}_{C}$, $\underbrace{\{6\}}_{D}$ Done.

The set of states is divided up into subsets of the initial set which have for their union the entire set S, and no common intersections. This is called a **partition** of the set S. As we progress from 0-equivalency on up, each subset can be divided, but none ever coalesce. There can be **partition refinement** (finer partition) only.