# Section 9.3: Finite-State Machines

April 23, 2021

**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 functionality. We will examine some ways in which we can "minimize" (streamline) 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

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) |
| $f_s$ | $f_s : S \times I \to S$, the next-state function |
| $f_o$ | $f_o : S \to O$, the output function |

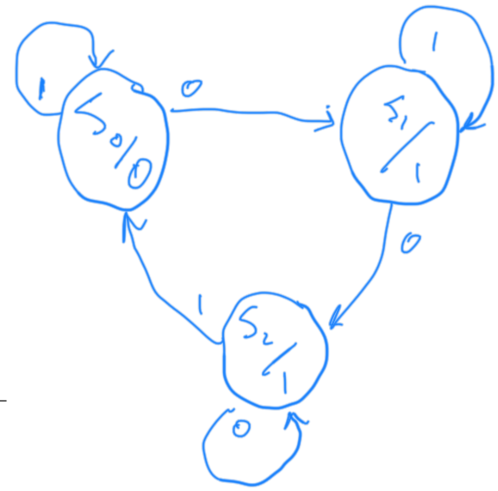
The machine is initialized to start in state $s_0$, and the machine operates *deterministically* (meaning that there is no randomness associated with its operation, given a fixed sequence of inputs).

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

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

and that

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




We represent $f_s$ and $f_o$ by

- state tables (e.g. Table 9.1, p. 730):

| Present state | next state, given input 0 | 1 | Output |
|---|---|---|---|
| $s_0$ | $s_1$ | $s_0$ | 0 |
| $s_1$ | $s_2$ | $s_1$ | 1 |
| $s_2$ | $s_2$ | $s_0$ | 1 |

S   $f_o$   $f_s$

- state graphs (e.g. Figure 9.3, p. 730)

A summary of these elements for Example 29, p. 730:

Table 2: Elements of finite-state machine of Example 29, p. 730.

| | |
|---|---|
| $S$ | $\{s_0, s_1, s_2\}$ |
| $I$ | {0,1} |
| $O$ | {0,1} |
| $f_s$ | $f_s(s_0, 0) = s_1,\ f_s(s_0, 1) = s_0$<br>$f_s(s_1, 0) = s_2,\ f_s(s_1, 1) = s_1$<br>$f_s(s_2, 0) = s_2,\ f_s(s_2, 1) = s_0$ |
| $f_o$ | $f_o(s_0) = 0,\ f_o(s_1) = 1,\ f_o(s_2) = 1$ |

**Example: Practice 43, p. 731** . (Note: the state table corresponding to the state graph is in figure 9.3 - we could use either the table or the graph to generate the output sequence.)

For the machine above), what output sequence is produced by the input sequence 1 1 0 0 1 ?

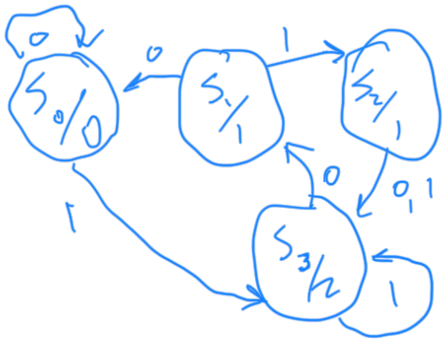
| time | $t_0$ | $t_1$ | $t_2$ | $t_3$ | $t_4$ | $t_5$ |
|---|---|---|---|---|---|---|
| Input | 1 | 1 | 0 | 0 | 1 | — |
| State | $s_0$ | $s_0$ | $s_0$ | $s_1$ | $s_2$ | $s_0$ |
| Output | 0 | 0 | 0 | 1 | 1 | 0 |

↑ spurious output

Output: 0 0 1 1 0

A machine M is given by the state graph below. Give the state table:

**Example: Practice 44, p. 731**

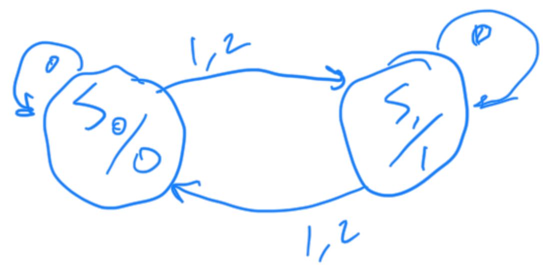



| Present | Next State 0 | Next State 1 | Output |
|---|---|---|---|
| $s_0$ | $s_0$ | $s_3$ | 0 |
| $s_1$ | $s_0$ | $s_2$ | 1 |
| $s_2$ | $s_3$ | $s_3$ | 1 |
| $s_3$ | $s_0$ | $s_3$ | 2 |

**Example: Practice 45, p. 731**

| Present | Next State 0 | Next State 1 | Next State 2 | Output |
|---|---|---|---|---|
| $s_0$ | $s_0$ | $s_1$ | $s_1$ | 0 |
| $s_1$ | $s_1$ | $s_0$ | $s_0$ | 1 |

a.

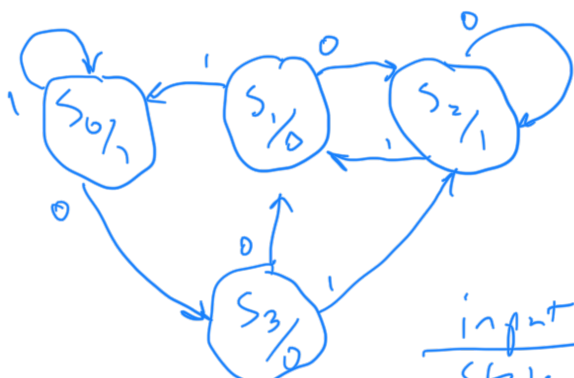



b. 2 1 1 0

| Input | 2 | 1 | 1 | 0 | – |
|---|---|---|---|---|---|
| State | $s_0$ | $s_1$ | $s_0$ | $s_1$ | $s_1$ |
| Output | 0 | 1 | 0 | 1 | 1 |

1 0 1 1

**Example: Exercise 4, p. 751**




| input | 1 | 1 | 0 | 1 | 1 | 0 | 0 | – |
|---|---|---|---|---|---|---|---|---|
| State | $s_0$ | $s_0$ | $s_0$ | $s_3$ | $s_2$ | [illegible] | $s_2$ | $s_2$ |
| Output | 1 | 1 | 1 | 0 | 1 | 0 | 1 | 1 |

spurious.

## 2 Construction of a machine: the Binary Adder

In section 8.2 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 (pp. 731-732).

$S = \{s_0, s_1, s_2, s_3\}$

$I = \{00, 01, 10, 11\}$

$O = \{0, 1\}$

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?

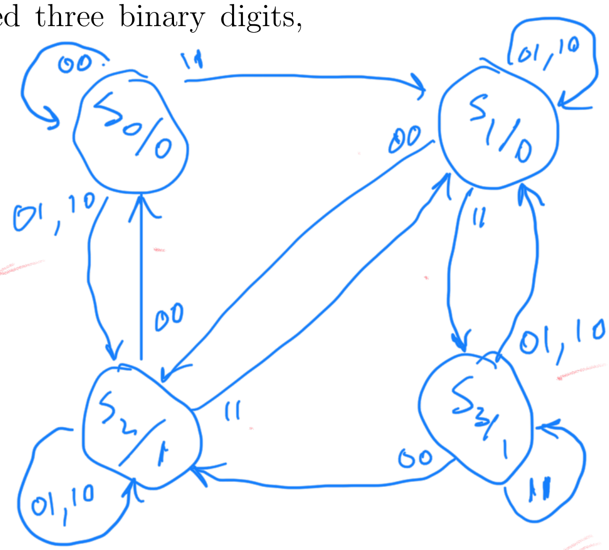
If you recall the full binary adder, which added three binary digits, there are four possible outcomes:

a. 00: carry 0, write 0 $s_0$

b. 01: carry 0, write 1 $s_2$

c. 10: carry 1, write 0 $s_1$

d. 11: carry 1, write 1 $s_3$




Now we just have to figure out

a. What the output should be, and

b. What the "next state" function looks like.

**Example: Practice 47, p. 733**

Compute the sum of

0 1 1 1 0 1 1 0 +
0 1 0 1 0 1 0 1

using the binary adder.

1 1 0 0 1 0 1 1

1 1 0 0 1 0 1 1 ✓

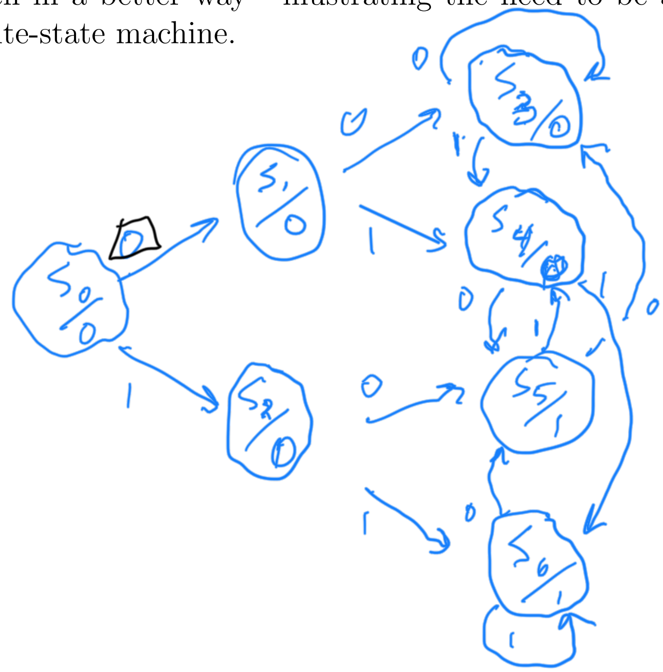
| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Input | 01 | 10 | 11 | 00 | 11 | 10 | 11 | 00 | |
| State | $s_0$ | $s_2$ | $s_2$ | $s_1$ | $s_2$ | $s_2$ | $s_1$ | $s_3$ | $s_2$ |
| Output | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 1 | 1 |

Now let's try something a little different:

Input: 101101

Output: 00101101

**Example: Exercise 15(a), p. 752** 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.



$s_3$ - write a delayed 0 & store a 0

$s_4$ - write a delayed 0 & store a 1

$s_5$ - write a delayed 1 & store a 0

$s_6$ - write a delayed 1 & store a 1

| Input | 1 | 0 | 1 | 1 | 0 | 1 | – | |
|---|---|---|---|---|---|---|---|---|
| State | $s_0$ | $s_2$ | $s_5$ | $s_4$ | $s_6$ | $s_5$ | $s_4$ | ? |
| Output | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 1 |

no matter what input we get an output of 1.

either $s_5$ or $s_6$

$s_3$ does the job of $s_1$

$s_4$ does the job of $s_2$

$s_6$ is replaceable by $s_3$!

| I | 1 | 0 | 1 | 1 | 0 | 1 | – |
|---|---|---|---|---|---|---|---|
| S | $s_3$ | $s_4$ | $s_5$ | | | | |
| O | 0 | 0 | 1 | | | | |

all the same now (deterministic)

## 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 $\alpha$, **ends** in a final state (a state with output 1) if and only if $\alpha \in S$.

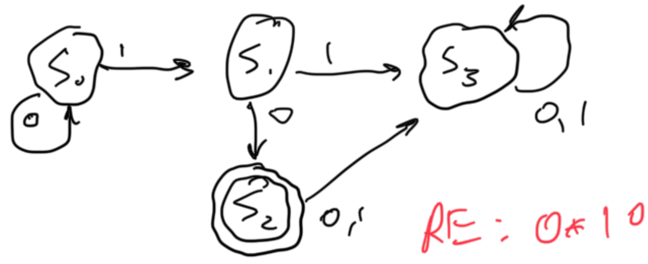
$S = \{10, 010, 0010, 00010, \ldots\}$

**Example: Practice 49, p. 735**


(a)

regular set $S = \{0\}$

regular expression: 0



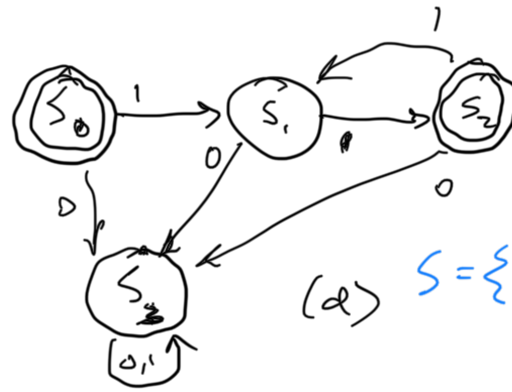

RE: 0*10

(b)


(c)

$S = \{0, 1\}$

RE: $0 \vee 1$




(d)

$S = \{\lambda, 11, 1111, 111111, \ldots\}$

RE: (11)* — parentheses are essential.

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 $\alpha$ is in $S$; and if string $\alpha$ is in $S$, then the output ends in a 1.

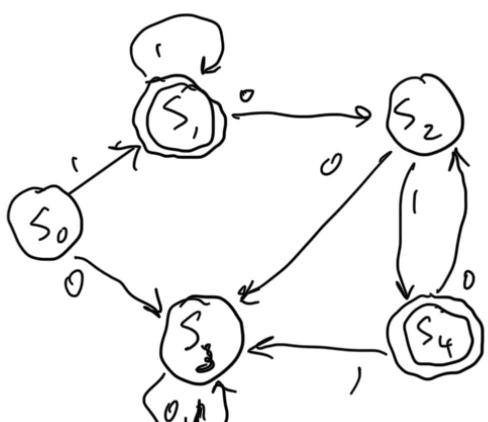
What kinds of input can a finite-state machine recognize? **Regular expressions**. **Regular expressions over** $I$ are defined recursively by

a. the symbol $\emptyset$ and the symbol $\lambda$;

b. the symbol $i$ for any $i \in I$; and

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

**Example: Exercise #36, p. 755**




$S = \{1, 11, 111, 1111, \ldots$ one of these, followed by 0, $1, 101, 10101, 1010101, \ldots\}$

RE: 11*(01)*

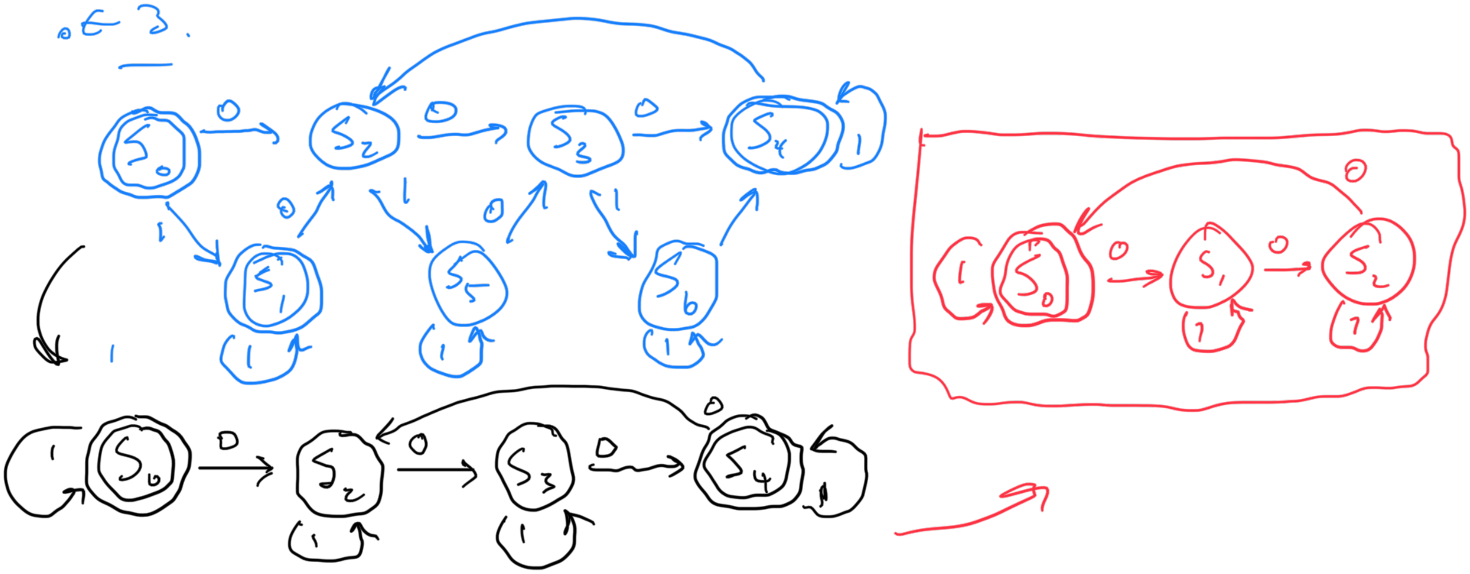
**Kleene's Theorem** assures us that a finite-state machine can recognize a set $S$ of input strings if and only if the set $S$ is a regular set (that is, a set represented by a regular expression).

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.

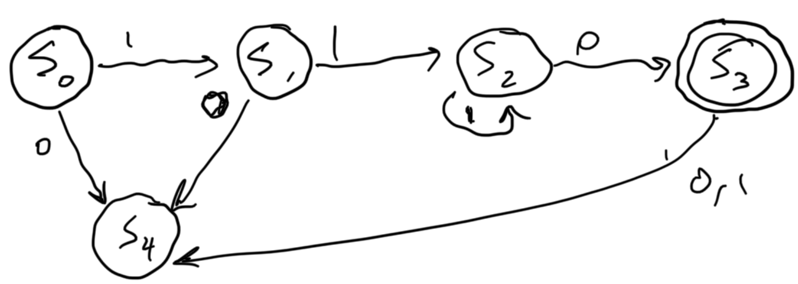
Examples of regular sets given by regular expressions:

- #27b. The set of all strings beginning with 000: $000(0 \vee 1)^*$
- #28a. 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: $((01)^*) \vee (110^*)$
- #28b. The set of all strings ending in 110: $(0 \vee 1)^* 110$
- #28c. The set of all strings containing 00: $(0 \vee 1)^* 00 (0 \vee 1)^*$
- #43b. The set of all strings of 0s and 1s having an odd number of 0s: $1^* 0 (1^* 0 1^* 0 1^*)^* \cong 1^* 0 1^* (0 1^* 0 1^*)^*$

**Example: Exercise 26(b), p. 754** - recognition and minimization motivation Set of all strings where the # of 0s is a multiple of 3.



**Example: Exercise 25(b), p. 754** : a machine to recognize all strings consisting of two or more 1s followed by a single 0.

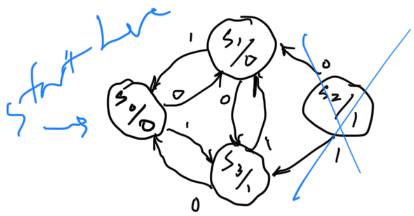


# 4 Machine Minimization

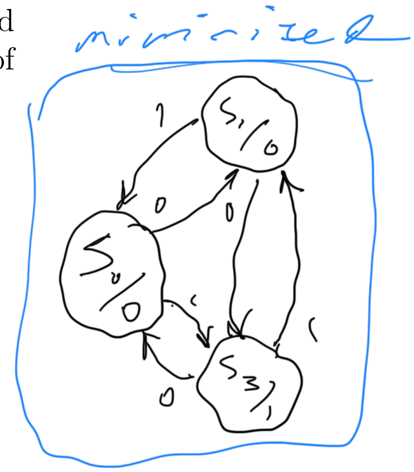
## 4.1 Unreachable States

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 9.3, p. 738; and Figure 9.9, p. 738.

**Example: Practice 52, p. 738**



| Present | Next 0 | Next 1 | Out |
|---|---|---|---|
| $s_0$ | $s_1$ | $s_3$ | 0 |
| $s_1$ | $s_3$ | $s_0$ | 0 |
| $s_2$ | $s_1$ | $s_3$ | 1 |
| $s_3$ | $s_0$ | $s_1$ | 1 |



0-equivalent: $\{s_0, s_1\}$, $\{s_3\}$

1-equivalent: $\{s_0\}$, $\{s_1\}$, $\{s_3\}$

The machine is minimized, already. No savings ... :)

## 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 $\alpha$.

(Our author seeds confusion by re-using notation: we are redefining $f_o$ as a function from $S \times I^* \to O^*$, where $O^*$ is finite strings of output.)

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.

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

b. 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.

c. 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.

d. 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 a few examples!

**Example: The sloppy delay machine of Exercise 15a**

| *state* | 0 | 1 | *out* |
|---|---|---|---|
| $s_0$ | $s_1$ | $s_2$ | 0 |
| $s_1$ | $s_3$ | $s_4$ | 0 |
| $s_2$ | $s_5$ | $s_6$ | 0 |
| $s_3$ | $s_3$ | $s_4$ | 0 |
| $s_4$ | $s_5$ | $s_6$ | 0 |
| $s_5$ | $s_3$ | $s_4$ | 1 |
| $s_6$ | $s_5$ | $s_6$ | 1 |

0-equivalent:
$\{s_0, s_1, s_2, s_3, s_4\}$, $\{s_5, s_6\}$
1-equivalent:
$\{s_0, s_1, s_3\}$, $\{s_2, s_4\}$, $\{s_5\}$, $\{s_6\}$
2-equivalents:
$\{s_0, s_1, s_3\}$, $\{s_2, s_4\}$, $\{s_5\}$, $\{s_6\}$
A B C D
No change! We're done! This machine is minimized!

| | Next | | |
|---|---|---|---|
| A | A | B | 0 |
| B | C | D | 0 |
| C | A | B | 1 |
| D | C | D | 1 |

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.

**Example: Exercise #65, p. 758**

| *state* | 0 | 1 | *out* |
|---|---|---|---|
| $s_0$ | $s_3$ | $s_6$ | 1 |
| $s_1$ | $s_4$ | $s_2$ | 0 |
| $s_2$ | $s_4$ | $s_1$ | 0 |
| $s_3$ | $s_2$ | $s_0$ | 1 |
| $s_4$ | $s_5$ | $s_0$ | 1 |
| $s_5$ | $s_3$ | $s_5$ | 0 |
| $s_6$ | $s_4$ | $s_2$ | 1 |

0-equivalent:
$\{s_1, s_2, s_5\}$, $\{s_0, s_3, s_4, s_6\}$
$\{1, 2, 5\}$, $\{0, 3, 4, 6\}$
1-equivalent:
$\{1, 2, 5\}$, $\{0\}$, $\{3, 4\}$, $\{6\}$
2-equivalent:
$\{1, 2, 5\}$, $\{0\}$, $\{3, 4\}$, $\{6\}$
B A C D
no change; we're done!

| | | | |
|---|---|---|---|
| A | C | D | 1 |
| B | C | B | 0 |
| C | B | A | 1 |
| D | C | B | 1 |

**Example: Exercise #67, p. 758**

| *state* | 0 | 1 | *out* |
|---|---|---|---|
| $s_0$ | $s_1$ | $s_2$ | 0 |
| $s_1$ | $s_2$ | $s_3$ | 1 |
| $s_2$ | $s_3$ | $s_4$ | 0 |
| $s_3$ | $s_2$ | $s_1$ | 1 |
| $s_4$ | $s_5$ | $s_4$ | 1 |
| $s_5$ | $s_6$ | $s_7$ | 0 |
| $s_6$ | $s_5$ | $s_6$ | 1 |
| $s_7$ | $s_8$ | $s_1$ | 0 |
| $s_8$ | $s_7$ | $s_3$ | 0 |

① no unreachable states!

0-equivalent:
$\{0, 2, 5, 7, 8\}, \{1, 3, 4, 6\}$

1-equivalent:
$\{0, 5\}, \{2\}, \{7, 8\}, \{1, 3, 4, 6\}$

2-equivalent:
$\{0\}, \{5\}, \{2\}, \{7, 8\}, \{1, 3\}, \{4, 6\}$

no change

3-equivalent
$\{0\}, \{5\}, \{2\}, \{7, 8\}, \{1, 3\}, \{4, 6\}$
A B C D E F

| | | | |
|---|---|---|---|
| A | E | C | 0 |
| B | F | D | 0 |
| C | E | F | 0 |
| D | D | E | 0 |
| E | C | E | 1 |
| F | B | F | 1 |