5. EQUIVALENCE AND REDUCTION OF FINITE STATE MACHINES

§5.1. Equivalent Machines

If you and I independently wrote computer programs to perform the same function it's almost certain that our programs would be different. Except for the most trivial examples the chance of getting identical programs is so small that students submitting identical programs for a computing assignment are generally considered to have copied.

Finite-state machines are more constrained and so the chance of you and I independently producing identical machines is much greater. Nevertheless it's quite common to come across two quite different machines that behave identically.

Identical behaviour lies behind the concept of equivalence. Two finite-state machines are considered to be equivalent if they produce the same output for each input. If we think of them as black-box machines they can only be distinguished if we open them up.

Two finite-state machines are **equivalent** if they have the same input alphabet (and for machines that output strings, the same output alphabet), and if for every input string, the two machines produce the same output.

(For an FSA, output means whether or not the string is accepted, so two FSA's are equivalent if and only if they accept precisely the same strings.)

The concept of equivalence raises two questions. Firstly, how can we tell if two given finite-state machines are equivalent? Even though their alphabets are finite, there are infinitely many possible input strings because an input string can be arbitrarily long. We cannot possibly test them all.

The second question concerns efficiency. If you build a 100-state machine that is equivalent to my 1000-state machine then yours is clearly more efficient. If the logic of the machine had to be incorporated into mass-produced microprocessors then your design will save the company money. So the second question is whether we can take any finite-state machine and improve it by some routine process until we obtain the most efficient machine possible that does the same job.

Both questions can be asked of computer programs in general. But there the answers are less than satisfactory. Yes, it is possible in certain cases for a programmer to compare two programs and decide whether they are equivalent. Indeed those whose job it is to mark student programs are required to do just that – declare that the student's program is equivalent to the lecturer's. It's even possible, in limited cases, to have such a decision process done by a computer, and some computer science

departments have experimented with student programming assignments being marked by a computer.

But as a generalised dream it is just that – a dream. And what is more it will always remain so. It has been shown that it is logically impossible for a computer program to determine, in all possible cases, whether or not two given programs are equivalent.

The other dream, never to be fulfilled in its entirety on purely logical grounds, is to be able to write a program and have the compiler optimise it completely. Very clever compilers do carry out some optimisation but none ever will, none ever can, guarantee to always come up with the shortest, fastest or best equivalent program (under any reasonable criterion).

With finite-state machines the position is very much better. We can, by a mindless process, that can easily be automated, determine whether two finite-state machines are equivalent. Moreover we can take a finite-state machine and convert it to an equivalent one using the smallest possible number of states.

Shortly we'll be getting down to the details, but here's an overview. Don't worry if you don't understand these next couple of paragraphs on first reading. You will when you come back to it.

It can be shown that in any equivalence class of finite-state machines there's a unique machine that uses the fewest states – unique, that is, apart from differences in labelling. To determine whether two machines are

equivalent we simply convert each of them to their shortest ‘cousin’. If these are the same, when suitably relabelled, the original two machines were equivalent. If no amount of relabelling will change one of these shortest cousins into the other (especially if they have different numbers of states) then the original two machines were not equivalent.

Given any finite-state machine M we can carry out two processes. The first, called the **Reduction Process**, produces a reduced machine that is equivalent to M. The second process, called the **Standardisation Process**, relabels the states so as to put the machine into a standard form.

If you and I each construct a machine with the same specifications (that is they are supposed to perform equivalently) we’ll probably produce different machines on differing numbers of states. After we put our machines through the reduction process our machines will probably still be different, but the number of states will be the same for both. Finally, putting our machines through the standardisation process, they’ll become absolutely identical. If they’re not then they were not equivalent to begin with.

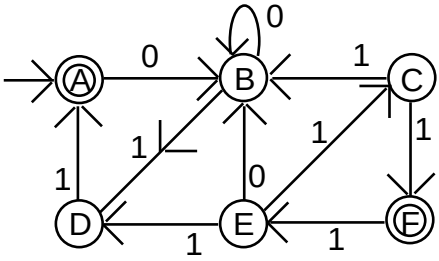
§5.2. Removing Inaccessible States

The first stage in the Reduction Process is to remove any states that can never be reached. While we would certainly never create such inaccessible states on purpose, in the process of designing a complicated

machine certain states may be created which are later by-passed by some improvement and we may not notice that they have become isolated. Indeed most computer applications contain significant sections of ‘dead code. Or at least the programmers suspect that these fossils from earlier versions will never be reached. But because it’s not possible to guarantee this, the code is left in, just in case. After all, for computer programs an extra megabyte or two of probably redundant code is no problem. For a small microprocessor every extra byte is significant.

A state in an FSA is **inaccessible** if there is no input string for which the machine will reach that state.

Example 1: Consider the following FSA.



Starting with the initial state, A, we can reach the state B (and also the black hole, Z, since the diagram has no arrow leading out of A with a ‘1’ label). From B we can get to B and D. From D we can reach Z and A. So starting from A the only states we can ever reach are A, B, D and Z. The others, C, E and F, are inaccessible and so can be deleted.

The systematic way of doing this is to prepare a list of accessible states. But first it's a good idea to express the FSA as a state table. Pictures are good, but it's surprisingly easy to overlook arrows in a diagram. And in any case, if the reduction process is to be done on a computer this is the format we'd have to use. Don't forget to include a black hole if there is one.

The State Table for the above FSA is:

	0	**1**	
→**A**	B	Z	*
B	B	D	
C	F	B	
D	Z	A	
E	B	D	
F	Z	E	*
Z	Z	Z	

We begin by writing down the initial state, in this case A. Even if it's not possible to ever return to that state, it's accessible simply because we start there.

Accessible States: A

Next we examine the transitions that lead out from that state. In this example there are two, one to B and one to Z. So we record that fact by adding these states to the

list and we mark A in some way, such as underlining it, to record the fact that we've finished examining it.

Accessible States: A B Z

Now it is B's turn to be examined. Not because it's the next letter after A in the alphabet, but because it's the next state on the list to be considered. From B we get to B and D. Since B is already on the list we only write down D. And, having finished with B, we underline it.

Accessible States: A B Z D

From Z we only get to Z, which we already have so we don't add anything to our list.

Accessible States: A B Z D

From D we get to Z and A. Again we have obtained nothing new.

Accessible States: A B Z D

But now there is nothing further to be considered. So our list of accessible states is complete. All other states are inaccessible. So C, E and F are inaccessible and may be removed. We can simply erase the corresponding rows from the table to give:

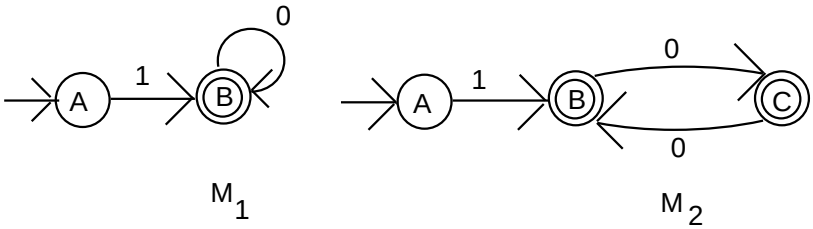
	0	1	
→**A**	B	Z	*
B	B	D	
D	Z	A	
Z	Z	Z	

We have thus reduced the 7-state FSA to one with only 4 states. (Of course if we were to convert this back to a state diagram we'd leave out the black hole, *Z*, but it would still have 4 states – it is just that only 3 of them would be drawn.)

§5.3. Equivalent States

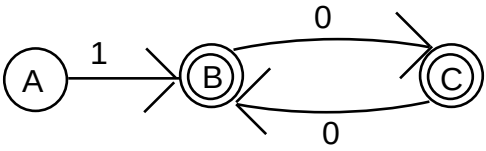
Example 2:

Consider the following two binary FSA's.



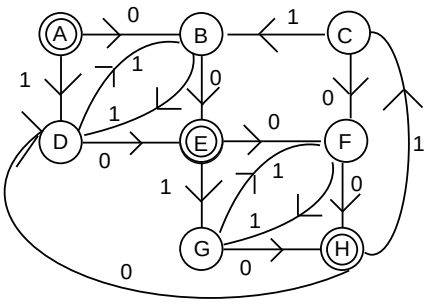
Clearly both accept 1, or any string that starts with a single 1, followed by 0's. Since they accept precisely the same strings these FSAs are equivalent.

Consider the second machine in the above example and let's allow the machine to be started in any of its states.



It's clear that states B and C are equivalent. To make this idea clearer, suppose we had two copies of this machine, with one currently in state B and the other in state C. If these machines were enclosed in 'black boxes' so that all we could do was to observe the output under differing input, there's no way we could determine which copy had started in state B and which started state C – to do that we'd have to look inside the 'black boxes' at the workings of the two copies of the machine. But, from a functional point of view, that is irrelevant. It is for this reason that we say that states B and C are equivalent.

Example 3: Suppose you were confronted with three copies of this FSA and were told that one copy started in state A, a second is in state E and the third in state H. Which is which? With the mechanisms hidden inside black boxes, all you could see would be the light bulb on the outside.



Since A, E and H are all accepting states all three machines would have their lights ON to begin with. So

we cannot distinguish between the three states A, E and H without applying any input because they are all accepting states. We describe this situation by saying that these three states are **0-equivalent.**

If we now input a ‘0’ to each machine, the one that was in state A will have moved to state B, the one that was in state E will now be in state F and the one that started in state H will now be in state D. In all three machines the light will have gone OFF. If instead we’d input a ‘1’ the machines would have gone into states D, G or C respectively. But the external behaviour would be identical in all cases – light OFF.

Now it may be that with further input we could distinguish between the three copies of the machine, and hence between states A, E and H, but with an input string of length 1 they are indistinguishable, or as we say, they are **1-equivalent** states.

Can we distinguish between them if we’re allowed an input string of length 2? The following table describes the successive states of the three copies of the machine under all four possibilities for an input string of length 2.

	start in A	start in E	start in H
00	A B E	E F H	H D E
01	A B D	E F G	H D B
10	A D E	E G H	H C F
11	A D B	E G F	H C B

With the covers off the differences are obvious, but representing each of the states by a '0' or '1' according to whether the light is OFF or ON, the *visible* histories of the two machines are given as follows:

	start in A	start in E	start in H
00	1 0 1	1 0 1	1 0 1
01	1 0 0	1 0 0	1 0 0
10	1 0 1	1 0 1	1 0 0
11	1 0 0	1 0 0	1 0 0

States A and E are clearly 2-equivalent. They can't be distinguished by input strings of length 2. But a difference now shows up with H. The state H is not 2-equivalent to the other two states because the behaviour with an input string of 10 is different.

If we went on to consider input strings of length 3 we'd find the following patterns of output:

	start in A	start in E	start in H
000	1 0 1 0	1 0 1 0	1 0 1 0
001	1 0 1 0	1 0 1 0	1 0 1 0
010	1 0 0 1	1 0 0 1	1 0 0 1
011	1 0 0 0	1 0 0 0	1 0 0 0
100	1 0 1 0	1 0 1 0	1 0 0 1
101	1 0 1 0	1 0 1 0	1 0 0 0
110	1 0 0 1	1 0 0 1	1 0 0 1
111	1 0 0 0	1 0 0 0	1 0 0 0

Not surprisingly H differs from the other two. Once two states are inequivalent at some level they are inequivalent at all higher levels. But A and E are still together. Perhaps we can distinguish between them with longer inputs.

But what if they are really equivalent at all levels. We could go on testing for longer and longer strings but we wouldn't know whether they would differ at the next level. Are we doomed, like the Flying Dutchman, to have to continue the process indefinitely, never getting a definite answer? Fortunately not.

§5.4. k-equivalence of States

We define ***k*-equivalence** of states, for all natural numbers *k*, inductively as follows.

(1) Suppose M is an FSA with transition function T. Two states are **0-equivalent** if they're both accepting or they are both non-accepting.

(2) Two states *s*, *t* are (*k*+1)-equivalent if:

- they're *k*-equivalent;
- for each input character c, the states T(s, c) and T(t, c) are *k*-equivalent.

We denote the relation of *k*-equivalence by $\equiv_k$. So if the set of accepting states is A we can express the above definition symbolically as follows. Here A is the set of accepting states:

(1) $s_1 \equiv_0 s_2$ means $s_1 \in A \leftrightarrow s_2 \in A$;
(2) $s_1 \equiv_{k+1} s_2$ means that $(s_1 \equiv_k s_2) \wedge \forall c[T(s_1, c) \equiv_k T(s_2, c)]$

It can be easily checked that for each *k*, *k*-equivalence is indeed, as the name suggests, an equivalence relation. So the set of states splits up as a disjoint union of equivalence classes under *k*-equivalence.

For any FSA having both accepting and non-accepting states there are exactly two 0-equivalence classes, one consisting of the accepting states and the

other consisting of the non-accepting ones. Since (k+1)-equivalence implies k-equivalence (we built this fact into the definition) the difference as we pass from k-equivalence to (k+1)-equivalence is that some of the k-equivalence classes might split up. To use technical jargon we say that the partition of (k+1)-equivalence classes is a **refinement** of the partition of the k-equivalence classes.

Example 4:

The following hypothetical example (with many of the details omitted) illustrates how this might occur in a specific case. We don't supply the details of the FSA and so we're not in a position to determine the partitions. We're merely assuming that the refinement process proceeds as stated.

Suppose we have states 1, 2, 3, 4, 5, 6, 7, 8 with 2, 3, 5 and 7 being the accepting states. This means that the partition into 0-equivalence classes will be

$$\{\{1, 4, 6, 8\}, \{2, 3, 5, 7\}\}.$$

Suppose the partition into 1-equivalence classes turns out to be

$$\{\{1, 4\}, \{6, 8\}, \{2, 5, 7\}, \{3\}\}.$$

Notice how this is a refinement of the earlier partition.

The partition at the 2-level might be

$$\{\{1, 4\}, \{6, 8\}, \{2, 5\}, \{7\}, \{3\}\}.$$

The classes {1,4} and {6, 8} didn't split up at this stage while the class {2, 5,7} did. The partition at the 3-level might be

$$\{\{1\}, \{4\}, \{6, 8\}, \{2, 5\}, \{7\}, \{3\}\},$$

a further refinement.

Suppose the partition at the 4-level is

$$\{\{1\}, \{4\}, \{6, 8\}, \{2, 5\}, \{7\}, \{3\}\}.$$

No further decomposition has occurred, that is, 4-equivalence is identical to 3-equivalence. Since equivalence at each level is defined in terms of the previous level this means that 5-equivalence, 6-equivalence and so on, must all be identical to 3-equivalence.

Once we reach a stage where *k*-equivalence is identical to (*k*+1)-equivalence then
***n*-equivalence will be identical for all $n \geq k$.**

Two states in a FSA are **equivalent** if they are *k*-equivalent for all *k*.

Since *k*-equivalence is an equivalence relation for all *k*, it follows that equivalence is indeed an equivalence relation. In our hypothetical example, states 6 and 8 will be equivalent to one another, as will states 2 and 5. All other states will be equivalent to themselves only.

NOTES:
(1) Starting with a finite set of states we must reach a situation where there is no change from one step to the next.
(2) If we ever reach a stage where every state is in an equivalence class all by itself then we can stop the process at that point because clearly no further refinement is possible.

(3) If two states are equivalent (in the final partition) we can never distinguish between them and so the machine will work just as well if we combine those states. To use technical jargon, we **identify** states that are equivalent to one another. This results in a smaller, but equivalent, machine.

In our hypothetical example above we can reduce the number of states from 8 to 6 by identifying states 2 and 5 and identifying states 6 and 8. This can be achieved by selecting one state from each equivalence class to be the representative and to delete any others. Any reference to a deleted state would be redirected to its representative.

§5.5. The Reduction Algorithm: Locating Equivalent States

Since there's a practical benefit in locating and identifying equivalent states, it's worth organising what we've been doing into a workable algorithm. We do this in a table. We illustrate this process with the FSA in example 3. In table form this is:

	0	**1**	
→**A**	B	D	*
B	E	D	
C	F	B	
D	E	B	
E	F	G	*
F	H	G	
G	H	F	
H	D	C	*

We rule up a table with as many rows as in the original table, but with many columns. We copy the states, and the transition table into this larger table and then copy the accepting state column into the next column, converting *'s to 1's and blanks to 0's.

	0	**1**	**≡$_0$**					
→**A**	B	D	1					
B	E	D	0					
C	F	B	0					
D	E	B	0					

E	F	G	1					
F	H	G	0					
G	H	F	0					
H	D	C	1					

The rows correspond to the states in the original machine, with the first column naming those states. The next few columns give the transition table of the machine. For a binary machine, where the input alphabet is {0, 1}, this amounts to two columns.

The next column gives the 0-equivalence partition, '1' for accepting states and '0' for non-accepting states.

Now we translate the transition table according to the $\equiv_0$ column. In this case, every occurrence of A, E or H in columns 2 and 3 gets converted to a '1' and put in the corresponding position in columns 5 and 6. Every occurrence of B, C, D, F or G is converted to a '0' and placed in their corresponding positions. Let's do it now.

	0	**1**	$\equiv_0$	**0**	**1**			
$\rightarrow$**A**	B	D	**1**	0	0			
B	E	D	**0**	1	0			
C	F	B	**0**	0	0			
D	E	B	**0**	1	0			
E	F	G	**1**	0	0			
F	H	G	**0**	1	0			
G	H	F	**0**	1	0			
H	D	C	**1**	0	0			

Those last 3 columns, taken together, give us 1-equivalence. The states that have the same entries in these columns are 1-equivalent. From now on we'll always put a '0' in the first row of that column indicating that this will be our name for that equivalence class. Any other row that has the same entries in these last three columns also gets the tag '0'.

	0	**1**	$\equiv_0$	**0**	**1**	$\equiv_1$		
→**A**	B	D	**1**	**0**	**0**	0		
B	E	D	**0**	**1**	**0**			
C	F	B	**0**	**0**	**0**			
D	E	B	**0**	**1**	**0**			
E	F	G	**1**	**0**	**0**	0		
F	H	G	**0**	**1**	**0**			
G	H	F	**0**	**1**	**0**			
H	D	C	**1**	**0**	**0**	0		

State B is not in this equivalence class so we name it with the next available number, that is, '1'. Any other row which is identical to the B row in the last three columns, is also a '1'.

	0	**1**	**≡$_0$**	**0**	**1**	**≡$_1$**		
→A	B	D	**1**	**0**	**0**	0		
B	E	D	**0**	**1**	**0**	1		
C	F	B	**0**	**0**	**0**			
D	E	B	**0**	**1**	**0**	1		
E	F	G	**1**	**0**	**0**	0		
F	H	G	**0**	**1**	**0**	1		
G	H	F	**0**	**1**	**0**	1		
H	D	C	**1**	**0**	**0**	0		

Clearly, in this example, there's just one more 1-equivalence class which contains all those states that have 0 0 0 in those three columns. We label this '2'.

	0	**1**	**≡$_0$**	**0**	**1**	**≡$_1$**		
→A	B	D	**1**	**0**	**0**	0		
B	E	D	**0**	**1**	**0**	1		
C	F	B	**0**	**0**	**0**	2		
D	E	B	**0**	**1**	**0**	1		
E	F	G	**1**	**0**	**0**	0		
F	H	G	**0**	**1**	**0**	1		
G	H	F	**0**	**1**	**0**	1		
H	D	C	**1**	**0**	**0**	0		

Again we translate the transition table (from columns 2 and 3 in our example), but this time we use the $\equiv_1$ column. Then we scan those columns, together with the $\equiv_1$ column itself, to get the $\equiv_2$ column. We continue in this way until we get two successive $\equiv_k$ columns being identical.

	0	**1**	$\equiv_0$	**0**	**1**	$\equiv_1$	**0**	**1**	$\equiv_2$	**0**	**1**	$\equiv_3$	**0**	**1**
→A	B	D	1	0	0	0	1	1	0	1	1	0	1	1
B	E	D	0	1	0	1	0	1	1	0	1	1	0	1
C	F	B	0	0	0	2	1	1	2	1	1	2	3	1
D	E	B	0	1	0	1	0	1	1	0	1	1	0	1
E	F	G	1	0	0	0	1	1	0	1	1	0	3	3
F	H	G	0	1	0	1	0	1	1	3	1	3	4	3
G	H	F	0	1	0	1	0	1	1	3	1	3	4	3
H	D	C	1	0	0	0	1	2	3	1	2	4	1	2

	0	**1**	$\equiv_3$	**0**	**1**	$\equiv_4$	**0**	**1**	$\equiv_5$
→A	B	D	0	1	1	0	1	1	0
B	E	D	1	0	1	1	3	1	1
C	F	B	2	3	1	2	4	1	2
D	E	B	1	0	1	1	3	1	1
E	F	G	0	3	3	3	4	4	3
F	H	G	3	4	3	4	5	4	4
G	H	F	3	4	3	4	5	4	4
H	D	C	4	1	2	5	1	2	5

Since 5-equivalence is identical to 4-equivalence we've obtained the final partition. States B and D are equivalent (at all levels) so having them both involves unnecessary redundancy. We can identify them (roll them up together) and so save a state. Also, states F and G can be identified.

§5.6. The Reduction Process: Identifying States

To identify two equivalent states s and t, we select one of them, say s, to represent them both and redirect all references to t to a reference to s.

Example 5:
Identifying states B and D and also F and G in the above example we get the reduced machine. We change all references to D to a B and change all G's in the table to F.

	0	**1**	
→A	B	B	*
B	E	B	
C	F	B	
B	E	B	
E	F	F	*
F	H	F	
F	H	F	
H	B	C	*

Now we omit all repeated rows:

	0	**1**	
→A	B	B	*
B	E	B	
C	F	B	
E	F	F	*
F	H	F	
H	B	C	*

Having a gap in the labelling of our states we may wish to relabel them from A to F.

§5. 7. Standard Form

A binary FSA with transition function T(x,y) is in **standard form** if:

(1) the states are labelled 0, 1, 2, 3, ...;

(2) state 0 is the initial state;

(3) if an entry in the table is greater than the maximum of those above and to the left of it, it must be exactly one more than this maximum.

In a nutshell, standard form means that states are labelled 0, 1, 2, ... in the order in which they are needed as you fill out the transition table row by row. Whenever a state needs a new label it is always given the next available number.

Example 6:

Put the following FSA in standard form:

	0	**1**	
→**F**	A	F	
S	Z	F	
A	S	A	*
Z	A	F	

Because F is the initial state we must label it 0. Since T(F, 0) = A, state A must be labelled ‘1’ (the next available number). But T(F, 1) = F which already has a code. So far in the relabelling process we have:

	0	**1**		
→0	1	0		F
1			*	A
2				
3				

The * that went with A is now associated with its new name, 1.

The next blank occurs in the shaded cell and that corresponds to T(A, 0) = S. Looking down the last column we note that S has not yet been assigned a code, so it gets the next available code of 2. Finally Z must be relabelled as 3. We have now put our machine in standard form:

	0	**1**		
→0	1	0		F
1	2	1	*	A
2	3	0		S
3	1	0		Z

Once we have the machine in Standard Form we no longer need the final column. So our final answer is simply:

	0	1	
→**0**	1	0	
1	2	1	*
2	3	0	
3	1	0	

It can be shown that if two FSA's are equivalent they will produce identical FSA's when minimised and put into Standard Form.

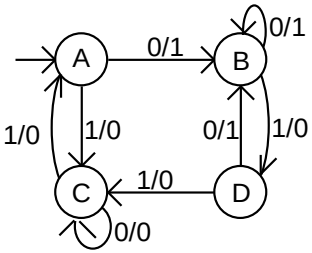
§5.8. Reduction of Mealy and Moore Machines

The only modification needed in order to apply the Reduction Process to Mealy and Moore machines is in the setting up of 0-equivalence.

For a Mealy machine, where the output depends on the combination of the current state and the input character, two states are 0-equivalent if, for each input character, the same output results from each state. In other words two states are 0-equivalent if the entries in the P-table are identical.

For a Moore machine, where the appropriate output is printed as the machine enters a state, two states are 0-equivalent if and only if they produce the same output.

Example 7:
Reduce the Mealy machine:



Solution: All states are accessible. The state table is:

	T		**P**	
	0	**1**	**0**	**1**
→A	B	C	1	0
B	B	D	1	0
C	C	A	0	0
D	B	C	1	0

Checking for equivalence:

	0	**1**	**0**	**1**	**≡$_0$**	**0**	**1**	**≡$_1$**	**0**	**1**	**≡$_2$**
→A	B	C	1	0	0	0	1	0	1	2	0
B	B	D	1	0	0	0	0	1	1	0	1
C	C	A	0	0	1	1	0	2	2	0	2
D	B	C	1	0	0	0	1	0	1	2	0

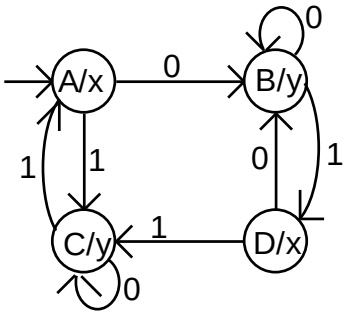
Identifying states A and D we get:

	T		P	
	0	**1**	**0**	**1**
→A	B	C	1	0
B	B	A	1	0
C	C	A	0	0

Finally, putting it into standard form:

	T		P	
	0	**1**	**0**	**1**
→0	1	2	1	0
1	1	0	1	0
2	2	0	0	0

Example 8:
Reduce the Moore machine:



Solution: All states are accessible.

	0	**1**	$\equiv_0$	**0**	**1**	$\equiv_1$	**0**	**1**	$\equiv_2$
→A	B	C	x	y	y	0	1	1	0
B	B	D	y	y	x	1	1	0	1
C	C	A	y	y	x	1	1	0	1
D	B	C	x	y	y	0	1	1	0

Identifying state C with B and D with A:

	0	**1**	**P**
→A	B	B	x
B	B	A	y

Putting it in standard form we get:

	0	**1**	**P**
→0	1	1	x
1	1	0	y

EXERCISES FOR CHAPTER 5

EXERCISES 5A (Inaccessible States)

Ex 5A1: Remove any inaccessible states from the following FSA.

	0	1	
→**A**	E	C	
B	A	D	*
C	A	E	*
D	E	B	
E	C	C	

Ex 5A2: Remove any inaccessible states from the following FSA.

	0	1	
→**A**	A	E	
B	D	B	
C	E	E	*
D	C	E	*
E	D	B	*

Ex 5A3: Remove any inaccessible states from:

	0	1	
A	A	E	
→**B**	D	B	
C	E	E	*
D	C	E	*
E	D	B	*

EXERCISES 5B (Standard Form)

Ex 5B1: Put the following FSA in Standard Form.

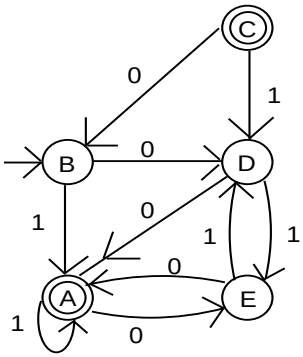
	0	1	
A	D	E	*
→B	E	B	
C	A	E	
D	A	D	
E	A	C	*

Ex 5B2: Put the following FSA in Standard Form.

	0	1	
A	D	E	*
B	E	B	
C	A	E	
D	A	D	
→E	A	C	*

EXERCISES 5C (Equivalent States)

Ex 5C1: Reduce the following FSA, expressing your answer as a State Table in Standard Form.



Ex 5C2: Reduce the following FSA and put it in Standard Form:

	0	1	
A	B	A	*
B	J	K	
C	C	H	
→D	F	K	
E	G	F	
F	K	I	*
G	E	I	
H	H	C	
I	K	F	*
J	B	J	*
K	I	A	

Ex 5C3: Reduce the following FSA and put in Standard Form:

	0	1	
→A	D	B	*
B	E	B	
C	D	A	*
D	C	D	
E	B	A	*

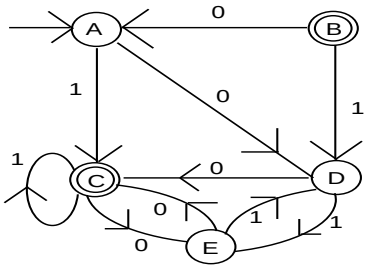
Ex 5C4: Reduce the following FSA and put in Standard Form:

	0	1	
→**A**	B	C	*
B	D	A	
C	D	B	
D	B	C	

Ex 5C5: Reduce the following FSA and put in Standard Form:

	0	1	
A	G	G	*
B	C	H	*
→**C**	B	I	
D	I	G	
E	G	G	*
F	B	C	
G	F	D	
H	B	B	*
I	B	C	

Ex 5C6: Reduce this FSA and put in Standard Form. Give a state diagram for your final machine.



Ex 5C7: Reduce the following Finite State Acceptor to an equivalent one on the smallest number of states. Give your answer as a state diagram.

	0	1	
→**A**	B	E	
B	E	C	
C	E	B	
D	A	B	*
E	C	E	*

Ex 5C8: Reduce the following FSA and put it into Standard Form:

	0	1	
→**A**	D	K	
B	K	H	
C	J	C	*
D	A	I	
E	J	E	*
F	F	G	
G	G	F	
H	I	E	
I	H	K	*
J	C	H	
K	H	I	*

Ex 5C9: Reduce the following Finite State Acceptor to an equivalent one on the smallest number of states. Give your answer as a state table in standard form.

	0	1	
→**A**	B	E	
B	E	C	
C	E	B	
D	D	F	
E	C	E	*
F	A	D	*

EXERCISES 5D (Reduction of Mealy Machines)

Ex 5D1: Reduce the following Mealy machine and put it into standard form:

	T		**P**	
	0	1	0	1
→**0**	1	3	0	0
1	7	2	1	1
2	7	1	1	1
3	8	3	1	1
4	8	4	1	1
5	6	1	1	0
6	5	2	1	0
7	2	3	0	0
8	4	7	0	0

Ex 5D2: Reduce the Mealy machine from Ex 4C11 (tennis scoring).

SOLUTIONS FOR CHAPTER 5

Ex 5A1: Starting at state A we can reach E and C. From E we reach C (which we already had) and from C we reach A and E (again nothing new). So only states A, E and C are accessible. Hence states B and D are inaccessible. Removing them we get:

	0	1	
→**A**	E	C	
C	A	E	*
E	C	C	

Ex 5A2: Listing the accessible states, in the order in which they occur, we get: A, E, D, B, C. So all 5 states are accessible.

Ex 5A3: This FSA is identical to the previous one, except that it has a different initial state. This time we must start with B. Listing the accessible states, in the order in which they occur, we get: B, D, C, E as accessible. So A is inaccessible.
Removing it we get.

	0	1	
→**B**	D	B	
C	E	E	*
D	C	E	*
E	D	B	*

Ex 5B1: B is coded as 0, because this is the initial state. From B we reach E (code as 1) and B (already coded). From 1 (i.e. E) we reach A (codes as 2) and C (code as 3). From 2 (ie A) we reach D (code as 4) and E (already coded).
All states are now coded:

0 = B, 1 = E, 2 = A, 3 = C, 4 = D.

Coding the State Table we get:

	0	**1**	
2	4	1	*
→0	1	0	
3	2	1	
4	2	4	
1	2	3	*

and rearranging the rows gives:

	0	**1**	
→0	1	0	
1	2	3	*
2	4	1	*
3	2	1	
4	2	4	

Ex 5B2: 0 = E, 1 = A, 2 = C, 3 = D. State B never arises, indicating that it is inaccessible. We therefore omit it, giving the Standard Form as:

	0	1	
→**0**	1	2	*
1	3	0	*
2	1	0	
3	1	3	

Ex 5C1:

STEP 1: Convert to a State Table

	0	1	
A	E	A	*
→**B**	D	A	
C	B	D	*
D	A	E	
E	A	D	

STEP 2: Delete any inaccessible states.

Starting at B, B→D and A, D→A and E. A leads to nothing new; nor does E.

So B, D, A, E are accessible, leaving C as inaccessible. Delete C.

	0	1	
A	E	A	*
→**B**	D	A	
D	A	E	
E	A	D	

STEP 3: Investigate equivalence of the remaining states. Begin by copying out the state table, coding accepting

states in the $\equiv_0$ column by 1's and the non-accepting states by 0's.

	0	1	$\equiv_0$
A	**E**	**A**	1
→B	**D**	**A**	0
D	**A**	**E**	0
E	**A**	**D**	0

STEP 4: Now continue the equivalence process until two successive ≡ columns are identical.

	0	1	$\equiv_0$	0	1	$\equiv_1$	0	1	$\equiv_2$
A	**E**	**A**	1	0	1	**0**	2	0	**0**
→B	**D**	**A**	0	0	1	**1**	2	0	**1**
D	**A**	**E**	0	1	0	**2**	0	2	**2**
E	**A**	**D**	0	1	0	**2**	0	2	**2**

Thus D ≡ E.

STEP 5: Select one state from each equivalence class and convert all references to the others to their chosen representative. We choose D from the equivalence class {D, E} and proceed to eliminate E by converting all references to E in the state table by a reference to D.

	0	1	
A	D	A	*
→B	D	A	
D	A	D	

STEP 6: Finally we put it in standard form:

B = 0 (since it is the initial state); D = 1 (the next to arise); A = 2.

	0	**1**	
→**0**	1	2	
1	2	1	
2	1	2	*

Ex 5C2:

The accessible states are: D, F, K, I, A, B, J, so C, E, G, H are inaccessible and may be deleted.

	0	**1**	
A	B	A	*
B	J	K	
→**D**	F	K	
F	K	I	*
I	K	F	*
J	B	J	*
K	I	A	

Now we carry out the Reduction Process.

	0	**1**	≡	**0**	**1**	≡	**0**	**1**	≡	**0**	**1**	≡	**0**	**1**	≡
			0			**1**			**2**			**3**			**4**
A	**B**	**A**	1	0	1	0	1	0	0	1	0	**0**	1	0	**0**
B	**J**	**K**	0	1	0	1	0	2	1	0	3	**1**	0	4	**1**
→ **D**	**F**	**K**	0	1	0	1	0	2	1	2	3	**2**	3	4	**2**
F	**K**	**I**	1	0	1	0	2	0	2	3	2	**3**	4	3	**3**
I	**K**	**F**	1	0	1	0	2	0	2	3	2	**3**	4	3	**3**
J	**B**	**J**	1	0	1	0	1	0	0	1	0	**0**	1	0	**0**

K	**I**	**A**	0	1	1	2	0	0	3	2	0	**4**	3	0	**4**

So A ≡ J and F ≡ I. Merging each pair we get:

	0	**1**	
A	B	A	*
B	A	K	
→D	F	K	
F	K	F	*
K	F	A	

Putting it into Standard Form we get:

	0	**1**	
→0	1	2	
1	2	1	*
2	1	3	
3	4	3	*
4	3	2	

(with 0 = D, 1 = F, 2 = K, 3 = A, 4 = B)

Ex 5C3: All states are accessible.

	0	**1**	**≡$_0$**	**0**	**1**	**≡$_1$**	**0**	**1**	**≡$_2$**
→A	**D**	**B**	1	0	0	**0**	1	1	**0**
B	**E**	**B**	0	1	0	**1**	2	1	**1**
C	**D**	**A**	1	0	1	**2**	1	0	**2**
D	**C**	**D**	0	1	0	**1**	2	1	**1**
E	**B**	**A**	1	0	1	**2**	1	0	**2**

So B ≡ D and C ≡ E. Identifying and putting into Standard Form we get:

	0	1	
→**0**	1	1	*
1	2	1	
2	1	0	*

Ex 5C4: All states are accessible and the equivalence classes are all of size 1, so the machine was already reduced. Putting it in Standard Form we get:

	0	1	
→**0**	1	2	*
1	3	0	
2	3	1	
3	1	2	

Ex 5C5: The accessible states are C, B, I, H. Deleting the others and carrying out the Equivalence Process:

	0	**1**	$\equiv_0$	**0**	**1**	$\equiv_1$	**0**	**1**	$\equiv_2$
B	**C**	**H**	1	0	1	**0**	1	2	**0**
→**C**	**B**	**I**	0	1	0	**1**	0	1	**1**
H	**B**	**B**	1	1	1	**2**	0	0	**2**
I	**B**	**C**	0	1	0	**1**	0	1	**1**

So C ≡ I. Merging these equivalent states and putting it into Standard Form we get:

	0	1	
→**0**	1	0	
1	0	2	*
2	1	1	*

Ex 5C6: We firstly convert it to a State Table.

	0	**1**	
→A	D	C	
B	A	D	*
C	E	C	*
D	C	E	
E	C	D	

The accessible states are A, D, C, E and so B is inaccessible. Deleting B and carrying out the Equivalence Process we get:

	0	**1**	$\equiv_0$	**0**	**1**	$\equiv_1$	**0**	**1**	$\equiv_2$
→A	**D**	**C**	0	0	1	**0**	2	1	**0**
C	**E**	**C**	1	0	1	**1**	2	1	**1**
D	**C**	**E**	0	1	0	**2**	1	2	**2**
E	**C**	**D**	0	1	0	**2**	1	2	**2**

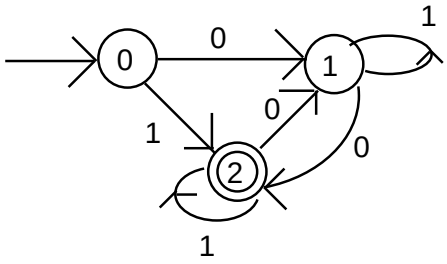
So D ≡ E. This gives:

	0	**1**	
→A	D	C	
C	D	C	*
D	C	D	

Putting this in Standard Form we get:

	0	**1**	
→0	1	2	
1	2	1	
2	1	2	*

The state diagram for this machine is:

Ex 5C7: D 

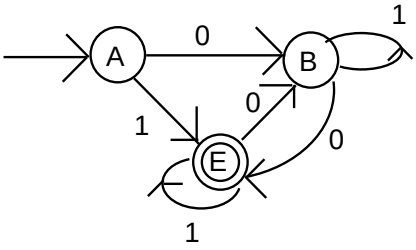
is inaccessible.

	0	**1**	$\equiv_0$	**0**	**1**	$\equiv_1$	**0**	**1**	$\equiv_2$
→A	**B**	**E**	0	0	1	**0**	1	2	**0**
B	**E**	**C**	0	1	0	**1**	2	1	**1**
C	**E**	**B**	0	1	0	**1**	2	1	**1**
E	**C**	**E**	1	0	1	**2**	1	2	**2**

So B ≡ C. Hence we may reduce the machine to 3 states:

	0	**1**	
→A	B	E	
B	E	B	
E	B	E	*

The state diagram for this machine is:



Ex 5C8: The accessible states are A, D, K, I, H, E, J, C. Thus B, F, G are inaccessible and may be omitted.

	0	**1**	$\equiv_0$	**0**	**1**	$\equiv_1$	**0**	**1**	$\equiv_2$	**0**	**1**	$\equiv_3$
→A	**D**	**K**	0	0	1	0	0	1	0	0	3	0
C	**J**	**C**	1	0	1	1	3	1	1	4	1	1
D	**A**	**I**	0	0	1	0	0	1	0	0	3	0
E	**J**	**E**	1	0	1	1	3	1	1	4	1	1
H	**I**	**E**	0	1	1	2	1	1	2	3	1	2
I	**H**	**K**	1	0	1	1	2	1	3	2	3	3
J	**C**	**H**	0	1	0	3	1	2	4	1	2	4
K	**H**	**I**	1	0	1	1	2	1	3	2	3	3

Thus A ≡ D, C ≡ E, I ≡ K. We may thus omit D, E and K and redirect references to them.

	0	**1**	
→A	A	I	
C	J	C	*
H	I	C	
I	H	I	*
J	C	H	

Putting this in Standard Form we get:

	0	1	
→**0**	0	1	
1	2	1	*
2	1	3	
3	4	3	*
4	3	2	

Ex 5C9: The accessible states are A, B, E, C. Hence we may remove D and F.

	0	**1**	**≡$_0$**	**0**	**1**	**≡$_1$**	**0**	**1**	**≡$_2$**
→**A**	B	E	0	0	1	0	1	2	0
B	E	C	0	1	0	1	2	1	1
C	E	B	0	1	0	1	2	1	1
E	C	E	1	0	1	2	1	2	2

Hence B ≡ C.

	0	1	
→**A**	B	E	
B	E	B	
E	B	E	*

In standard form it is:

	0	1	
→**0**	1	2	
1	2	1	
2	1	2	*

Ex 5D1: The accessible states are 0, 1, 3, 7, 2, 8, 4. Deleting the inaccessible states 5, 6 we get:

	T		P	
	0	**1**	**0**	**1**
→0	1	3	0	0
1	7	2	1	1
2	7	1	1	1
3	8	3	1	1
4	8	4	1	1
7	2	3	0	0
8	4	7	0	0

The first step in the Equivalence Process is a little different for Mealy Machines. We code the different combinations of output as 0, 1, etc.

	T		P		
	0	**1**	**0**	**1**	**≡0**
→0	1	3	**0**	**0**	0
1	7	2	**1**	**1**	1
2	7	1	**1**	**1**	1
3	8	3	**1**	**1**	1
4	8	4	**1**	**1**	1
7	2	3	**0**	**0**	0
8	4	7	**0**	**0**	0

The Equivalence Process continues just as for FSAs.

	T		P											
	0	**1**	**0**	**1**	**≡$_0$**	**0**	**1**	**≡$_1$**	**0**	**1**	**≡$_2$**	**0**	**1**	**≡$_3$**
→0	**1**	**3**	0	0	0	1	1	0	1	1	**0**	1	2	**0**
1	**7**	**2**	1	1	1	0	1	1	0	1	**1**	0	1	**1**
2	**7**	**1**	1	1	1	0	1	1	0	1	**1**	0	1	**1**
3	**8**	**3**	1	1	1	0	1	1	2	1	**2**	3	2	**2**
4	**8**	**4**	1	1	1	0	1	1	2	1	**2**	3	2	**2**
7	**2**	**3**	0	0	0	1	1	0	1	1	**0**	1	2	**0**
8	**4**	**7**	0	0	0	1	0	2	1	0	**3**	2	0	**3**

So $0 \equiv 7$, $1 \equiv 2$ and $3 \equiv 4$. Merging these equivalent states we get:

	T		P	
	0	**1**	**0**	**1**
→0	1	3	0	0
1	0	1	1	1
3	8	3	1	1
8	3	0	0	0

Putting it into Standard Form we get:

	T		P	
	0	**1**	**0**	**1**
→0	1	2	0	0
1	0	1	1	1
2	3	2	1	1
3	2	0	1	1

Ex 5D2:

	T		P											
	S	**R**	**S**	**R**	**≡1**	**0**	**1**	**≡2**			**≡3**			**≡4**
→A	E	B			0	0	0	0	0	0	**0**	4	1	**0**
B	F	C			0	0	0	0	0	1	**1**	5	2	**1**
C	G	D			0	0	1	1	1	2	**2**	6	3	**2**
D	H	A		R	1	1	0	2	2	0	**3**	7	0	**3**
E	I	F			0	0	0	0	3	0	**4**	8	5	**4**
F	J	G			0	0	0	0	3	1	**5**	9	6	**5**
G	K	H			0	0	1	1	4	2	**6**	10	7	**6**
H	L	A		R	1	1	0	2	5	0	**7**	11	0	**7**
I	M	J			0	2	0	3	6	3	**8**	12	9	**8**
J	N	K			0	2	0	3	6	4	**9**	13	10	**9**
K	O	L			0	2	1	4	7	5	**10**	14	11	**10**
L	P	A		R	1	0	0	5	4	0	**11**	10	0	**11**
M	A	N	S		2	0	2	6	0	6	**12**	0	13	**12**
N	A	O	S		2	0	2	6	0	7	**13**	0	14	**13**
O	A	P	S		2	0	0	7	0	4	**14**	0	10	**14**
P	Q	R			0	2	1	4	7	5	**10**	14	11	**10**
Q	A	P	S		2	0	0	7	0	4	**14**	0	10	**14**
R	P	A		R	1	0	0	5	4	0	**11**	10	0	**11**

Hence P ≡ K, Q ≡ O, R ≡ L.

If you play tennis you may have had a situation arise where you were unsure as to whether the score was 30 all or deuce. Then somebody may have said, “it doesn’t make any difference”. Indeed, in terms of the final outcome of the game “30 all” is equivalent to “deuce”. The reduced machine is thus:

	T		P	
	S	**R**	**S**	**R**
→**A**	E	B		
B	F	C		
C	G	D		
D	H	A		R
E	I	F		
F	J	G		
G	K	H		
H	L	A		R
I	M	J		
J	N	K		
K	O	L		
L	K	A		R
M	A	N	S	
N	A	O	S	
O	A	K	S	