Directed Graph for Finite-State Machine

Tito D. Kesumo Siregar (13511018)[1]
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
[1]*13511018@std.stei.itb.ac.id*

***Abstract*–Finite-state machine is a widely used logic abstraction for a system in which the output depends on the current state of the machine. It turns out that a subset of finite-state machine with certain state table is conveniently expressed as simple state diagram, which is a directed graph. The state diagram is able to be recognized as input by various algorithms. One commonly and quite important process is to simulate the finite-state machine, whose speed is in linear growth, but can be reduced to a constant time by pre-computation and pre-analyze. From the paper, it is deduced that it might be possible that all finite state machine is expressed in a simple state diagram, thus allowing a vast number of finite-state machine to be simulated, analyzed, and processed using various standard graph algorithms.**

***Index Terms*–finite-state machine, directed graph, state table, state diagram, simulation, circuit detection, cycle-finding.**

I. INTRODUCTION

Many kinds of machines, including computer components, can be modeled using a structure called a finite-state machine. Finite state machines are the basis for programs for spell checking, grammar checking, traffic lights, dictionary, and electronic devices.

A formal definition of a finite-state machine with output is given in [1]. It basically states that a finite-state machine is described by a finite set of states, an (optional) input, output, a transition function, an output function, and an initial state. Reference [2] states that a finite-state machine is informally known as sequential circuits, which is a class of circuits with the outputs depend on the past behavior of the circuit, as well on the present values of inputs.

Designing a finite-state machine takes several steps and some time. Reference [2] summarizes part of the steps involved in designing the logical part of finite-state machine – more generally, a synchronous sequential circuit:

1. Obtain the specification of the desired circuit.
2. Select a starting state and derive the states for the machine.
3. Create a state table from the state diagram.
4. Minimize the number of states.
5. Decide on the number of state variables needed to represent all states and perform the state assignment.

Given a description of finite-state machine, it is interesting if one is able to simulate it with a computer. Simulations are useful to check whether a finite-state machine works correctly or not.

Reference [2] uses various tables to describe a finite state machine. These tables are interesting in order to build a finite-state machine, but as we only interested to simulating finite-state machines, we would prefer a simpler and better method to look at the finite-state machine logic. A simpler method would ease programmers to create finite-state machine simulator programs.

As it turns out, a directed graph is able to describe a finite-state machine in an easier way for human – and possibly computers, too. Furthermore, as we shall see, there exists an algorithm to determine a loop in a finite-state machine if the input is limited to determined patterns. Determination of a circuit is important to reduce the time complexity of finite-state machine simulator with certain input down to $O(1)$; given a finite-state machine, one is able to compute the state (and possibly the output if the machine is of *Moore* type, described in [2]) in a short time.

II. BASIC THEORY

A. Basic Terminology of Graph

Mathematically, a graph is defined as a pair of set of non-empty vertices set *V* and edges set *E*, with each member of *E* connects a pair of vertices in *V* [3]. In this paper, we shall focus on a type of graph called directed graph, which edges connect a vertex v_1 with v2 but not backward (from v2 to v1) – formally, each member of edge set *E* in a directed graph G_D is an ordered pair [4].

We shall use a simple diagram to show a graph with circles as vertices and arrows as edges. More information will be added inside the circle or along the arrow. This is also the same representation of graph used in [1]. An example of this diagram is shown is Figure 1.

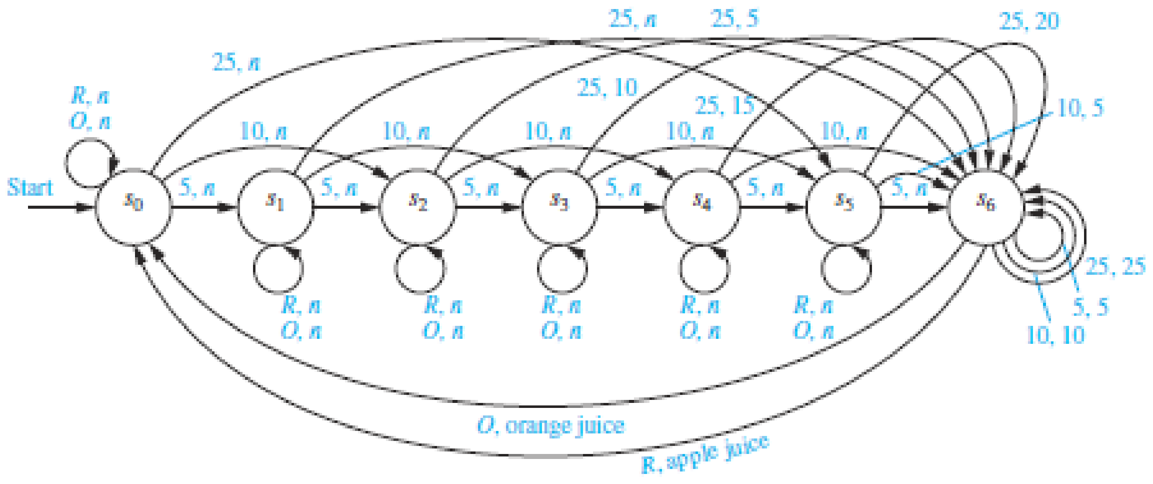


Figure 1. A representation of directed graph using circles and arrows.

Later in this paper, we shall show how a graph representation of a finite-state machine can be used to find a circuit. A circuit is a path in the graph that starts and ends in a same vertex. A path itself is an ordered set of vertex v_1, v_2, v_3, …, v_n in which v_i and v_{i+1} is connected for all $i \in [1, n-1]$.

B. Finite-State Machine

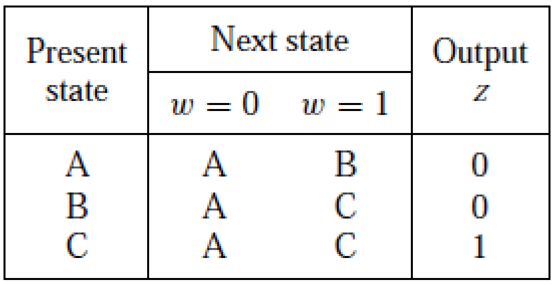
A sequential circuit is a class of circuits in which the outputs depend on the past behavior of the circuit, as well on the present values of inputs. Furthermore, there are two different kinds of sequential circuit, a Moore type and a Mealy type. A Moore type sequential circuit, which corresponds with the class of finite-state machines we shall focus on, is described as a sequential circuit whose outputs depend only on the state of the circuit.

According to [2], formal definition of a finite-state machine $M = (S, I, O, f, g, s_0)$ consists of a finite set *S* of states, a finite input alphabet *I*, a finite output alphabet *O*, a transition function *f* that assigns to each state and input pair a new state, an output function *g* that assigns to each state and input pair an output, and an initial state s_0. We shall, however, not strictly follow this definition, and let *I* and *O* be any type of input and output we can represent – for example, binary digit 0 and 1 is a widely used input in digital system, and "red", "yellow", and "green" is an appropriate representation of output in a traffic light system.

In this paper, we shall focus on finite-state machines correspond with finite-state machines with outputs in [1] and Moore type of sequential circuits in [2]. We focus on Moore type since the state of the finite-state machine directly determine the output, where Mealy type also depend on the input. Hence, the output in Mealy type is not unique with the state of the machine, making the current state information of finite-state machine less useful.

Sometimes, a state table is useful to determine the next state, given a current state. A state table is a table containing information of the next state of the finite-state machine, given the current state and the input. This table makes determination of the next state easy for human, and if represented using two-dimensional array in computer, makes determination of the next state for computer can be done in $O(1)$.

An example of state table is shown in Figure 2. To use the table, for example, consider that the current state of a finite-state machine with state table of Figure 2 is *A*. From the table, it is implied that the next state of *A* is *A* if *w* = 0 and B if *w* = 1. Furthermore, from the table, it is known that the output of the finite-state machine is 0, since the current state is *A*.



Present state	Next state		Output
	$w = 0$	$w = 1$	z
A	A	B	0
B	A	C	0
C	A	C	1

Figure 2. An example of state table.

To further show how the directed graph and the state table are tied – we shall peek at what will be described in this paper, consider Figure 3. Figure 3a shows a state table and 3b shows a state diagram can be constructed with the information in the state diagram. Please note that the output is not unique with the state and is represented in the graph as extra information along the edge. As it will be shown below, it feels more natural to represent an output directly by its state, and it is actually possible to represent the Mealy type finite-state machine with an equivalent Moore type finite-state machine.

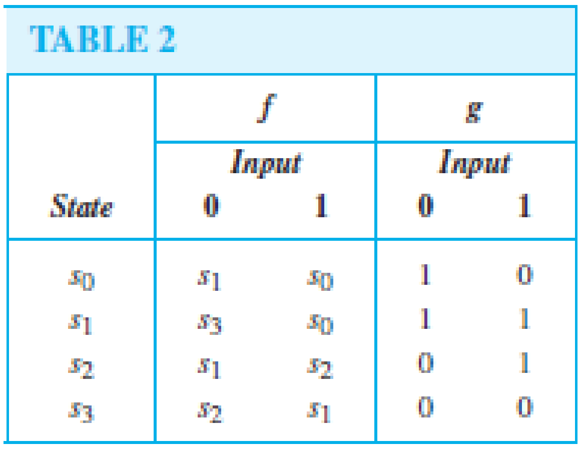
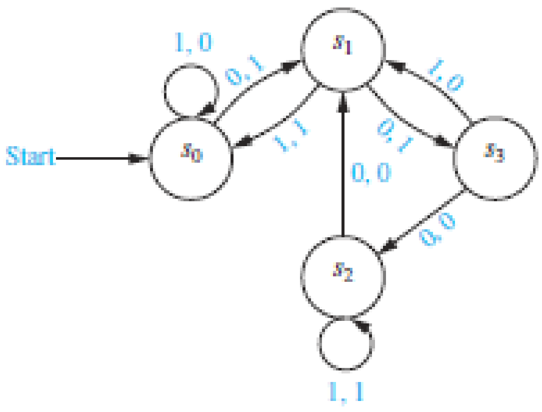


TABLE 2

	f		g	
	Input		*Input*	
State	**0**	**1**	**0**	**1**
s_0	s_1	s_0	1	0
s_1	s_3	s_0	1	1
s_2	s_1	s_2	0	1
s_3	s_2	s_1	0	0

(a)



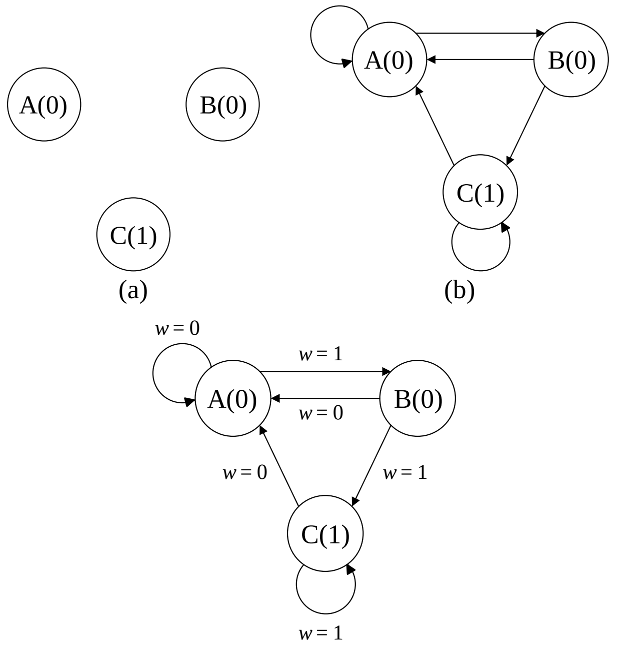
(b)

Figure 3. (a) A state table, (b) A state diagram corresponding with state table (a).

III. Building a State Diagram

A. Conversion of a State Table into a State Diagram

Conversion of a state table into a state diagram is actually a relatively easy problem. Consider the state table in Figure 2. From the table, it is obvious that the state set is {A, B, C}, so we draw three circles to represent this states (Figure 4a). Then, to represent the transition, we observe that the state *A* change into state *A* (a loop) and *C*. We then draw all the possible transitions as edges from the present state to all the next states (Figure 4b). Notice, however, that each edge must be differentiable, since state transition depends on certain values, so we add labels to the edge to create a proper state diagram for Figure 2 (Figure 4c).



(c)

Figure 4. (a) A set of vertex, (b) A set of vertex and edges, (c) The edges is labeled, making traversing the graph easier.

B. Properties of a State Diagram

Observing Figure 4c, there are some differences with the previous state diagrams in Figure 3b and Figure 1. There is a value inside the diagram and the edges are labeled with values 0 and 1 only (the label *w* is merely for cosmetic looks, since the label can be dropped for the reasons will be explained below). We take advantages on our previous limitations to produce a simpler graph.

First, we limit the output of the finite-state machine to be unique with the state. Thus, it is not possible for a state to have two different outputs. So we can easily include the output into the diagram. Second, we limit the transition to be limited by one variable only. This makes it possible to traverse the graph by keeping only one variable at mind. These limitations actually will be proved useful later, as it is possible to represent the state diagram in two familiar graph representations: adjacency list and adjacency matrix. We name the state diagram with these limitations with *simple state diagram* – a simple name.

However, many real-world finite-state machines are not limited to one variable and have multiple outputs for a state. We try work around one of the limitation we have imposed in order to represent a larger space of finite-state machines.

In order to formalize the process, we introduce the *state diagram creation process* below to create a simple state diagram from a single-output state table.

State diagram creation process. Given a single-output state table T_S, do the following steps to produce a state diagram G_S from T_S:

1. Create states $S_0, S_1, S_2, \ldots, S_n$ as vertices.
2. Create edge $e(S_p, S_q, w_i)$ from S_p to S_q labeled w_i if there is a transition from S_p to S_q when the value of variable w is w_i.
3. Repeat (2) until all possible transitions in the state table T_S is covered.

C. Converting a Multi-Output State Table into a State Diagram

Present state	Next state		Output	
	w=0	w=1	w=0	w=1
A	A	B	0	1
B	C	C	0	1
C	A	C	1	0

Figure 5. A state table with multiple outputs per state.

Consider Figure 5, which is a Figure 2 state table with an addition of output value depending on *w*, and the B output is changed. It is actually possible to convert this table into a similar state table whose finite state machine will just work like this table. Observe that it is possible to separate the state *A* into two state A_0 and A_1, and change the table into a state table with only one output per state. We can then apply the same procedure to convert the state table into a simple state diagram. Figure 6a shows the converted state table and 6b shows the corresponding state diagram.

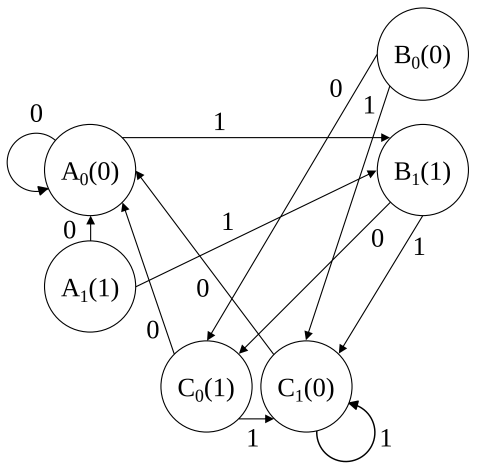
We are able to proof the following theorem with the steps just described:

Theorem 1. It is possible to create a simple state diagram from a multi-output state table whose outputs depend on the same variable with the transition variable.

Proof: Assume a multi-output state table T_S, whose outputs depend on the same variable with the transition variable. For each n possible value of $w = w_0, w_1, w_2, \ldots, w_n$, create additional states $S_{0w0}, S_{0w1}, S_{0w2}, \ldots, S_{0wn}, S_{1w0}, S_{1w1}, S_{1w2}, S_{1w3}, \ldots, S_{1wn}, \ldots S_{nwn}$. The resulting table T_S* is a single-output state table. To prove the lemma, convert T_S* into a state diagram D_S* using the state diagram creation process.

Present state	Next state		Output
	w=0	w=1	
A_0	A_0	B_1	0
A_1	A_0	B_1	1
B_0	C_0	C_1	0
B_1	C_0	C_1	1
C_0	A_0	C_1	1
C_1	A_0	C_1	0

(a)



(b)

Figure 6. (a) A single-output state table coverted from Figure 5, (b) A corresponding state diagram of (a).

Notice that we omitted the *w* label in the state diagram. Also, notice that state A_1 and B_0 does not have any incoming edge; the only time for the state to occur is at the beginning of the state. These 'lonely state' can be predicted from Figure 5; there are no present state whose next state when $w = 1$ is *A* and $w = 0$ is *B*, correspondingly.

IV. STATE DIAGRAM IN GRAPH REPRESENTATION

A. Graph Representation

In order to process the graph, computers need to have a computable graph representation. The most important process the graph representation must have is *Next(S, w)* – given the current state *S* and the transition input *w*, return the next state *S'*. This process is crucial in order to traverse the graph. We also need *Output(S)* – given the state *S*, output the value of the finite-state machine. However, *Output(S)* can be implemented with a simple lookup table, which, given the number of state *n*, the space complexity is in order of $O(n)$, and the time complexity up to order of $O(1)$ depending on the lookup table implementation. We can safely ignore the process of addition and deletion of vertices or edges, since it is highly unlikely for a finite state machine to be modified once it is specified (the process to do so is actually a fairly complex process).

The first approach is using an adjacency matrix. We can store the transition value *w* in the matrix element $A[S_1,S_2]$, with the column and row representing the states. Values of empty Figure 7 shows state table in Figure 2 as adjacency matrix, with -1 marking that there are no connection. Notice that the matrix is not a triangle matrix.

	A	B	C
A	0	1	-1
B	0	-1	1
C	0	-1	1

Figure 7. An adjacency matrix representation of Figure 2.

However, this approach is not efficient. For a state diagram with *n* states, we need to create a $n \times n$ matrix. To find *Next(S, w)*, we also need to traverse the column (or row, depending on implementation), which is in order of $O(n)$. In fact, if the number of possible transition value is small compared to the amount of state, the matrix will become a sparse matrix, which is space inefficient. For example, imagine a state table similar with Figure 2, which has only 2 possible value but suppose that the number of state is now 10. There are 100 matrix elements, in which only $2 \times 10 = 20$ (each row has only two elements filled) is used.

A	<0, A>	<1, B>
B	<0, A>	<1, C>
C	<0, A>	<1, C>

Figure 8. An adjacency list representation of Figure 2.

Compressing the matrix into an adjacency list is better, but not best. See Figure 8 for an adjacency list representation of Figure 2, compressing the elements into a tuple of next state and the transition value. Each element's list contains exactly the number of possible transition value, reducing the space complexity in order of $O(mn)$, given the number of possible transition value is *m*. However, we still need to traverse the row list in order to find *Next(S, w)*.

The best representation is an implicit representation using the state table. For each state, we store an array containing the next state with the key is the possible transition value. Using a two-dimensional array and Figure 2 as the example, the state table is reproduced in Figure 9. The space complexity is still in order of $O(mn)$, but *Next(S, w)* can be produced in only $O(1)$, since it merely look up the matrix $M[S, w]$.

As a side note, the state table representation can be derived from the adjacency list. Observe from the adjacency list there are exactly *m* elements in each row, and in each row all the possible state is ensured to be valued. We can then set the possible state as the key to produce Figure 9.

	0	1
A	A	B
B	A	C
C	A	C

Figure 9. A state table graph representation of Figure 2.

C. Circuit Detection

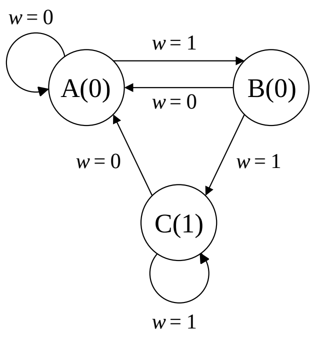


Figure 10. Redrawing of Figure 4c.

Given the computable graph representation, we can now use various graph algorithms. One case study is to check a circuit detection. Consider Figure 10, which is a Figure 4c redrawn for the sake of simplicity. Suppose that we simulate the usage of this finite-state machine with the *w* looping in {1, 1, 0}, starting from A for a time of *t*, which *t* denoting the number of state transition (including looping, such as *Next(A, 0) = A*). We wish to produce the sequence of outputs produced. In order to do this, we can simply calling in *Next* and *Output* on each loop (note that we introduce *NextW(w)* to return the next *w* in the loop):

```
state S = A {state initialized}
variable W = w {transition variable initialized}
repeat t times
   print(Output(S))
   S = Next(S, W)
   W = NextW(W)
```

The output produced is a continuous sequence of 001 (001001001001001001001001…). However, if the time *t* is a big number, for example, 10^9, the loop will be run 10^9 times. Assuming the code runs on a computer running 10^6 processes per second, the code needs 10^3 second (close to 15 minutes) to complete. The code itself is in order of $O(t)$, assuming the state is implemented using a state table representation described above.

We can reduce the running time by analyzing the partial output. Observe the output, which is a sequence of 001 starting from the first output. If we associates the output with an integer number *i* starting from 0, we obtain that the result of 0 is obtained at $i = 0, 3, 6, 9, \ldots$, 0 is obtained at $i = 1, 4, 7, \ldots$, and 0 is obtained at $i = 2, 5, 8, \ldots$. We can then construct the following function *f*:

$$f(t) = \begin{cases} 0, t \equiv 0(mod\ 3) \\ 0, t \equiv 1(mod\ 3) \\ 1, t \equiv 2(mod\ 3) \end{cases}$$

Once we have the function *f*, we can return the answer of the output of a finite-state machine in one time, given the initial state and the transition value loop in a constant $O(1)$ modulo calculation. Furthermore, if the output analysis is included in the process of returning the value, we can produce an algorithm to compute it in $O(g(S,W))$ with $g(S, W)$ is the time needed to analyze the partial output, instead of full simulation of the finite-state machine, which is useful when $n > g(S, W)$. Because the algorithm speed depends on $g(S, W)$, it is important to use a fast algorithm in analyzing the output.

Fortunately, there exist some algorithms to detect circuit in the graph [5].

Since the state diagram is a graph, one can use depth-first search algorithm with constraint in only traversing the vertices pointed by *Next(S, W)*. This reduces to a simple simulation of the finite-state machine until it hits a circle in its evolution. If depth-first search is used as circuit-finding algorithm, the time complexity is in order of $O(\mu + \lambda)$, where μ is the largest index before the circuit is recognized (in the previous example, $\mu = 0$ because the circuit is immediately started from the beginning of simulation) and λ is the loop length (in the previous example, $\lambda = 3$). The space complexity is kept to O(1) if theHowever, depth-first search is useful to pre-analyze the state diagram in $O(d)$ space when *d* is the depth of depth-first search calls (which is possibly the longest circuit in the state diagram), making all subsequent calls to the output is in order of $O(1)$.

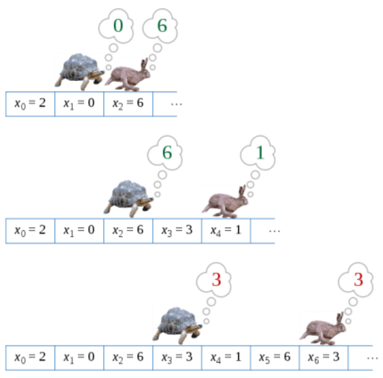


Figure 11. An illustration of Floyd's cycle finding algorithm, using tortoise and hare. [6]

Another useful algorithm is Floyd's cycle-finding algorithm [5]. We can use the algorithm, since the output is a sequence, thus can be used as input for the algorithm. Like depth-first search, it runs in order of $O(\mu + \lambda)$, but with $O(1)$ space. The hidden constant is also lower than depth-first search, since the algorithm does not need a stack call.

V. Conclusion

We have demonstrated that a finite-state machine with a certain type of state table can be expressed in a simple state diagram, which itself can be expressed in a graph representation suitable for computation. However, the number of finite-state machine currently proven to be expressed in simple state diagram is limited to Moore type, multi-output single-variable finite state machine, whereas a vast number of finite state machines is a Mealy type and many machines use many variables in the finite state machine. However, one hint is shown in [2], which states that a Mealy type machine can be turned into a Moore type machine. Thus, it might be possible to find workarounds in order to be able to convert all finite-state machines into a simple state table, thus converting it into state diagram and graph representation.

Furthermore, once a graph representation is obtained – the fastest implementation is actually a trivial state table – one can use various directed graph algorithms to achieve something. In the paper, depth-first search approach is used. Another algorithm, Floyd's cycle-finding, while itself not a graph algorithm, is usable to find circuit because it follows a straight path (no branching). This provides a clue that various graph algorithm might be able to be used in order to solve certain problems in finite-state machine design, simulation, or evaluation.

Reference

[1] K. H. Rosen, *Discrete Mathematics and Its Applications*, 7th ed., New York: McGraw-Hill, 2012, pp. 858-863

[2] S. Brown and Z. Vranesic, *Fundamentals of Digital Logic with VHDL Design*, 3rd ed., New York: McGraw-Hill, 2009, ch. 8.

[3] R. Munir, "Diktat Kuliah IF2091 Struktur Diskrit", 4th ed., periodical style, 2012, ch. 8.

[4] J. Bang-Jensen and G. Gutin, *Digraphs: Theory, Algorithms and Applications*, 1st ed., digital copy, http://www.cs.rhul.ac.uk/books/dbook/, 12/17/2012 22:49, pp. 2.

[5] S. Halim and F. Halim, *Competitive Programming: Increasing the Lower Bound of Programming Contest*, School of Computing, National University of Singapore, unpublished, 2010, ch. 4.

[6] Image courtesy of Wikipedia: http://en.wikipedia.org/wiki/File:Tortoise_and_hare_algorithm.svg, 12/17/2012 23:15.

STATEMENT

I hereby claimed that I have wrote this work by myself, and this work is not a copy, a translation from another people's work, or a plagiated work.

Bandung, 29 April 2010

[illegible]

Tito D. Kesumo Siregar
13511018