CARPENTER, CECIL SEIGLER, JR. Theory of Computation and Computing Machines. (1974) Directed by: Dr. Michael Willett. Pp. 37

It was the purpose of this paper to present an introductory discussion of two areas of automata theory, computation and computing machines. The notions of machine design, equivalent machines, function tables, state graphs, and state minimization were discussed in Chapter II. The concept of algorithm was introduced in Chapter III and an algorithm to minimize the number of states of a computing machine was constructed. A more general form of state equivalence was defined in Chapter IV, resulting in an open question as to the existence of an algorithm for determining state equivalence for this more general form.

THEORY OF COMPUTATION AND COMPUTING MACHINES

by

Cecil Seigler Carpenter, Jr.

A Thesis Submitted to
the Faculty of the graduate School at
The University of North Carolina at Greensboro
in Partial Fulfillment
of the Requirements for the Degree
Master of Arts

Greensboro
August, 1974

Approved by

[illegible]
Thesis Adviser

APPROVAL SHEET

This thesis has been approved by the following committee of the Faculty of the Graduate School at The University of North Carolina at Greensboro.

Thesis Adviser Michael Willett

Oral Examination Committee Members K. A. Byrd

William A. Powers

E E Posey

19 July 74
Date of Examination

ACKNOWLEDGMENT

I would like to thank Dr. Michael Willett for his assistance in the preparation of this paper and for teaching me that the ability to write effectively is as important as the knowledge of one's subject.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

At the turn of the century, David Hilbert, a famous mathematician and leader of the formalist school, was convinced of the existence of an algorithm for establishing the consistency or inconsistency of any mathematical system. Kurt Gödel [2] showed in 1931 that the consistency of any system which included the natural numbers could not be established. This result was a corollary to his more startling "incompleteness theorem" which states that if any formal system which contains the natural numbers is consistent, then that system is necessarily incomplete. More directly, there is a statement P in the system such that neither P nor not-P is a theorem of the system. Since either P or not-P must be true, then there is a true statement in the theory which is not provable. Thus the algorithm which Hilbert believed existed, in fact did not exist.

The formal notion of algorithm - or "effective" procedure as it is often called - had concerned mathematicians before the result of Gödel. How was an algorithm to be defined? When an algorithm was constructed, could it be determined whether or not it was meaningful? These and other questions now appeared more ominous than ever. Logicians turned their efforts toward establishing some type of approach which would enable them to categorize those procedures which were meaningful as opposed to those which were not. Most notably, the work of Alonzo Church and Alan M. Turing led to the creation of a new area of mathematics now known as automata theory.

There are many areas of automata theory of which only two will be discussed here. These two areas are computation and machine theory. Computation deals with the concept of performing a procedure or process dictated by a given set of instructions. Admittedly the very statement as to what constitutes a computation is quite nebulous. Therefore formalizing the notion of computation is a major concern of automata theorists. We shall consider as adequate the following: a computation consists of the process of following a given set of instructions. In order to enable us to perform a computation, we require some type of mechanism. This mechanism will be called a computing machine. (We will use the terms machine and computing machine interchangeably.) It should be understood that the term machine as used here refers to a mathematical notion rather than an actual physical object. A computing machine will be a mathematically defined object, capable of mechanical realization, whose task is to perform a given algorithm in an unthinking, non-creative way.

When speaking of machines and computation, many questions arise quite naturally. What computations can be realized by a machine? Are there computations which no machine can perform? Do there exist computations for which the question of whether or not any machine can perform them is undecidable? The proposition that "any process for which an algorithm can be constructed can be realized by a computing machine" is known as Church's thesis [1]. It would be erroneous to assume that we could construct a proof of Church's thesis since it obviously deals not only with very simple, concrete procedures, but also with the thought processes of the human mind. It is remarkable, however, that several independent and fundamentally different approaches to defining an "effective" algorithm have all yielded the same class of algorithms [4]. These include the work of Turing [7], Kleene [3], Post [5], and Smullyan [6].

In this paper, it is our objective to present a formal definition of a computing machine and computation. Chapter II deals with these concepts as well as the process known as machine design. We will also present several examples which illustrate the operational aspects of a machine. The idea of designing a minimal state machine is considered in Chapter III. The concept of minimization will be discussed in terms of a special type of equivalence relation. This relation contains a restriction which will be lifted in Chapter IV. We will then investigate the difficulties which arise as a result of lifting this restriction.

It is not the purpose nor intent of this paper to furnish the reader with a thorough background in the theory of machines and computation but rather to serve as an introduction to these notions. For this reason, we will motivate and discuss the various topics in an intuitive yet sound manner as opposed to the more classical mathematical methods so often found in treatises on automata theory.

CHAPTER II

COMPUTING MACHINES

In this section we will formalize mathematically the notions of computing machines and computation discussed in the introduction.

Intuitively a computing machine is a functional relationship between input and output mediums which is further dependent upon the changing internal condition of the machine. For our purposes, a computing machine $M = (A, S, B, f)$ shall consist of

(1) a finite set A called the alphabet which contains a special symbol b called a blank,

(2) a finite set S of elements called states with a special state H called the halting state,

(3) a non-empty finite subset B of integers,

(4) a function $f: A \times S \to A \times S \times B$ called the computing function.

If all the integers in B are of the same sign, then M is called a finite state machine. The elements of A will be used as the input/output symbols. The set B is part of the output as will be explained below. The elements of S will be used to indicate the "internal" condition of M at any instant. The function f determines the relationship between input, internal condition, and output.

To relate this definition of a computing machine to some meaningful notion of computation, we need a mechanism for recording the input/output stream. To this end, we define

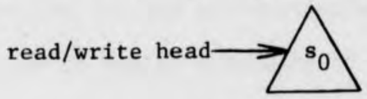
$$T \equiv \{\tau = (\cdots, a_{-1}, a_0, a_1, \cdots) \mid a_i \in A, \text{ only a finite number of the } a_i \neq b\}.$$

The elements of T are called tapes. For each $\tau \in T$, the machine M alters τ in the following sequential way:

(1) An internal state $s_0 \in S$ is selected and our attention is initially focused on the element a_0. We indicate this by using a pointer, called a read/write head, containing the symbol s_0 (see Figure 2.1). This initial configuration will also be denoted by $M/(s_0, \tau)$.

Figure 2.1: Initial Configuration of a Machine

$$\tau = (\ . \ . \ ., a_{-2}, a_{-1}, a_0, a_1, a_2, \ . \ . \ .)$$



(2) Assume that $f(a_0, s_0) = (a, s, n)$. Then f determines the following sequence of events:

   (a) the symbol a_0 is changed to a

   (b) our attention (the read/write head) is shifted $|n|$ positions to the right if $n > 0$, to the left if $n < 0$

   (c) the internal state is changed to s.

This sequence will constitute one unit of time.

(3) Step (2) is then repeated from this new position. These alterations and shifts continue recursively until the machine enters the halting state H, at which time it

terminates operation. If H is never encountered as an internal state, then the machine never halts. We denote by τ_i the altered tape created from τ after i time units. Therefore $\tau_0 = \tau$. If M halts after producing τ_N, then we set $\tau_k = \tau_N$ for all $k \geq N$. This sequential alteration of τ will be called a computation with initial input τ.

If we are given the function table for a particular machine, then computation on input tapes is well defined even though the sequence of alterations may not have any intuitive and/or algebraic interpretation. Our study of machines however will be from the point of view that machines are mechanisms which can be used to implement a number of verbally defined and meaningful algorithms. We are thus generally required to design a machine to perform computations described to us in non-mechanical terms. This design process consists of selecting an appropriate alphabet, determining what aspects of the algorithm correspond to different internal states when the algorithm is viewed mechanically, and finally, identifying the computing function. The following examples should clarify this process.

Example 2.1: In this example, we will not be concerned with machine design nor the computation to be performed, but rather with the description and operation of a computing machine.
Let $A = \{b, 0, 1\}$, $S = \{H, s_1, s_2, s_3\}$, and $B = \{-2, -1, 0, 1, 2, 3\}$. Then $M = (A, S, B, f)$ is the computing machine described by Table 2.1, called the computing function table for M. Notice we have selected s_1 as our starting state.

Table 2.1: Computing Function Table for Example 2.1.

	State	Input	Output	Next State	Head Movement
Start	s_1	0	1	s_3	3
Start	s_1	1	0	s_2	2
Start	s_1	b	b	H	0
	s_2	0	0	s_1	1
	s_2	1	0	s_3	1
	s_2	b	b	s_2	1
	s_3	0	0	s_1	-1
	s_3	1	1	s_2	-2
	s_3	b	b	H	0

Now let $\tau = (. . ., b, 0, 0, 0, 0, 0, 0, b, . . .)$ be an input tape. The "computation" governed by M proceeds in the following manner.

(1) The leftmost 0 is read with M in state s_1. The 0 is replaced with a 1, the read/write head is moved 3 positions to the right, and M transitions to state s_3.

(2) A 0 is read with M in state s_3. It is left unaltered, the read/write head is moved 1 position to the left, and M transitions to state s_1.

This sequence of events is continued until M enters the halting state H, which occurs after the sixth unit of time.

The sequentially altered tape appears as follows:

$$\tau_0 = (. . ., b, 0, 0, 0, 0, 0, 0, b, . . .)$$

s_1

$$\tau_1 = (. . ., b, 1, 0, 0, 0, 0, 0, b, . . .)$$

s_3

$$\tau_2 = (. . ., b, 1, 0, 0, 0, 0, 0, b, . . .)$$

s_1

$$\tau_3 = (. . ., b, 1, 0, 1, 0, 0, 0, b, . . .)$$

s_3

$$\tau_4 = (. . ., b, 1, 0, 1, 0, 0, 0, b, . . .)$$

s_1

$$\tau_5 = (. . ., b, 1, 0, 1, 0, 1, 0, b, b, . . .)$$

s_3

$$\tau_6 = (. . ., b, 1, 0, 1, 0, 1, 0, b, b, . . .)$$

H

Example 2.2: Let us now consider the more common situation of designing a machine to perform a meaningful computation. In this example we want a machine which will operate on any tape $\tau \in T$ of the form

$$\tau = (. . ., b, a_0, a_1, . . ., a_n, x, b, . . .),$$

where each a_i is either 0 or 1, and M will halt with altered tape

$$\tau_k = (. . ., b, a_0, a_1, . . ., a_n, x, a_0, a_1, . . ., a_n, b, . . .)$$

for some k. That is, we require a machine which will copy a non-blank sequence of zeros and/or ones immediately to the right of the symbol x. One such algorithm is as follows:

(1) "remember" whether a_0 is 0 or 1

(2) shift to the right to the first blank square and copy a_0

(3) shift to the left to find a_1

(4) shift to the right and record a_1 in the first blank square

(5) continue this process until we encounter the x which tells us the sequence has been copied.

Table 2.2 defines a machine which will perform the required computation.

To see how the machine design was accomplished, consider the operation of M. If $a_0 = 0$, then a_0 is replaced by A and M enters state s_2. The head shifts to the right, staying in s_2 without altering the tape, until it encounters a blank. A 0 is then printed and M enters state s_4. The operation of M is the same if $a_0 = 1$, except a_0 is replaced by B, M enters s_3 and prints a 1 in the first blank square. The fact that $a_0 = 0$ is "remembered" by use of s_2. Similarly if $a_0 = 1$, M "remembers" by entering s_3. After entering state s_4, the head moves to the left until an A or B is encountered, leaving the intermediate squares unaltered. Thus

A and/or B "tells" M the last symbol copied. Now the A or B is changed to a 0 or 1 respectively and M enters state s_1 and reads the next symbol in the sequence. This continues until the x is read, at which time M halts. Thus we have $A = \{b, 0, 1, x, A, B\}$, $S = \{H, s_1, s_2, s_3, s_4\}$, and $B = \{-1, 0, 1\}$.

Table 2.2: A "Copy" Machine

	State	Input	Output	Next State	Head Movement
Start	s_1	0	A	s_2	1
Start	s_1	1	B	s_3	1
Start	s_1	x	x	H	0
	s_2	0	0	s_2	1
	s_2	1	1	s_2	1
	s_2	x	x	s_2	1
	s_2	b	0	s_4	-1
	s_3	0	0	s_3	1
	s_3	1	1	s_3	1
	s_3	x	x	s_3	1
	s_3	b	1	s_4	-1
	s_4	0	0	s_4	-1
	s_4	1	1	s_4	-1
	s_4	x	x	s_4	-1
	s_4	A	0	s_1	1
	s_4	B	1	s_1	1

To illustrate the machine acting on some tape $\tau \in T$, suppose $\tau = (. . ., b, 1, 0, 1, 1, x, b, . . .)$ is our input tape. Then we have the following initial configuration:

$$\tau_0 = (. . ., b, \underset{\triangle\, s_1}{1}, 0, 1, 1, x, b, . . .).$$

After 47 time units we have entered the halting state H and our final tape appears as

$$\tau_{47} = (. . ., b, 1, 0, 1, 1, \underset{\triangle\, H}{x}, 1, 0, 1, 1, b, . . .).$$

In the above examples, we have displayed the computing function in tabular form. As long as the machine is fairly simple, a table will suffice as an adequate representation of the operations. However, as a machine becomes more complex in design, the table description necessarily loses much of its simplicity, and the nature of the computation becomes harder to visualize. For this reason, we shall adopt a graphic notation.

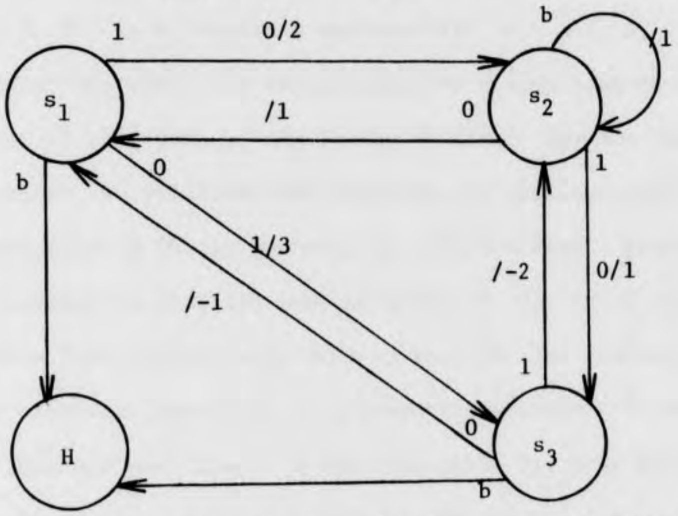
Suppose we are given a machine M in tabular form. We could represent one line of the computing function table in the following way:

$$\begin{tikzcd} \textcircled{s_0} \arrow[r, "a_0 \qquad a/n"] & \textcircled{s} \end{tikzcd}$$

The above symbol will stand for $f(a_0, s_0) = (a, s, n)$. We convert the table to graphical form by drawing a circle (node) for each state in S and then connecting various nodes in the above fashion, one arrow for each line of the computing function table.

We will adopt several conventions in constructing these state graphs. No input symbol at the base of an arrow will mean that that arrow is for <u>every</u> input symbol. No output symbol will mean that the input symbol is left unaltered. The following state graph represents the machine of Example 2.1.

<u>Figure 2.2</u>: A State Graph



As we have stated in the introduction, automata theorists are concerned with formalizing the concept of computation. Machine-like structures similar to Figure 2.2 have become the standard approach to this study. There are several standard machine models widely discussed in the literature. Our particular computing machine encompasses attributes of a number of these models. A natural question arises in such a case: are there algorithms which can be performed by a machine in one class and not by any machine of another class? Does one class of machines, by virtue of some design feature, possess more computing power than another class? Recall that a finite state machine is a computing machine with the set B restricted to integers of the same sign. Then it is clear that a finite state machine is not allowed to return to a previously scanned input position. It can be shown that no machine in this class can perform the computation of Example 2.2. Now consider a similar restriction on the set of computing machines. Call a machine M_T a Turing machine if $M_T = (A, S, B, f)$ is a computing machine with $B = \{-1, 0, 1\}$. Are there computations which can be performed by a computing machine but which cannot be performed by any Turing machine? For the remainder of this chapter, we will turn our attention to this question.

Physically, a Turing machine M_T differs from a general computing machine in that the head movement of M_T is of length at most 1. This restriction would seem to restrict the computational capability otherwise available to a computing machine. To examine this question, consider one line of a function table for some machine $M = (A, S, B, f)$, say $f(a, s) = (a', s', n)$ where $| n | > 1$. Then by adding states between s and s' which leave an input tape τ unaltered, we could accomplish the head movement by moving one square per unit time. Obviously the set S must be altered, which in turn will force us to define a new computing function f', but the alphabet A would remain the same. The implication is that for any computation which can be performed by a computing machine, there is a Turing machine which can perform the same computation. More formally we have

<u>Theorem 2.1</u>: Given a computing machine $M = (A, S, B, f)$, there exists a Turing machine $M' = (A, S', B', f')$ such that for

each $\tau \in T$: if M produces τ_i in i time units and M' produces τ'_i in i time units, then there exists a sequence of positive integers $\{n(i)\}_{i=1}^{\infty}$ (depending on τ) such that

(1) $n(i) < n(i+1)$

(2) $\tau_i = \tau'_{n(i)}$

(3) $\tau'_j = \tau'_{n(i)}$ for all $n(i) \leq j < n(i+1)$.

Proof: Consider any node s in the graph of the given machine M. Since M contains only a finite number of states and a finite alphabet A, there are only a finite number m of arrows into node s. Assume that the arrows are labeled so that $n_1, n_2, \ldots, n_k$ are all positive and $n_{k+1}, \ldots, n_m$ are all negative, where the n_i are the respective head movements. The no-head-movement transitions will remain unaltered so that they need not be considered. Let $n = \max\{n_1, n_2, \ldots, n_k\}$. Now modify M in the following way. First add $n - 1$ new states labeled $t_1, \ldots, t_{n-1}$ such that

$$\begin{tikzcd} t_{n-1} \arrow[r, "/+1"] & t_{n-2} \arrow[r, "/+1"] & \cdots \arrow[r, "/+1"] & t_1 \arrow[r, "/+1"] & s. \end{tikzcd}$$

If $n_1 > 1$, then remove the arrow

$$\begin{tikzcd} s_1 \arrow[r, dash, "a_1 \quad b_1/n_1"] & s \end{tikzcd}$$

and replace it with the arrow

$$\begin{tikzcd} s_1 \arrow[r, "a_1 \quad b_1/+1"] & t_{n-n_1} \end{tikzcd}$$

The one time unit operation of printing b_1 and then jumping n_1 squares to the right is replaced by the printing of b_1 and a move n_1 squares to the right <u>one</u> <u>square</u> <u>per</u> <u>time</u> <u>unit</u> without altering the intermediate squares. The tape alterations are the same except that this new machine takes longer to move its read/write head to a new position. This alteration and its affect on the original computation performed by M on a particular input tape will lead quite naturally to the sequence of positive integers required in the statement of the theorem. Now make this type of substitution for each arrow into node s. That is, replace

$$\begin{tikzcd} s_i \arrow[r, "a_i \quad b_i/n_i"] & s \end{tikzcd} \qquad 1 \le i \le k$$

with

$$\begin{tikzcd} s_i \arrow[r, "a_i \quad b_i/+1"] & t_{n-n_i} \end{tikzcd} \quad .$$

Similarly, let $N = \max\{|n_i| : k+1 \le i \le n\}$ and add $N - 1$ new states to M labeled $T_1, \ldots, T_{N-1}$ so that we have

$$\begin{tikzcd} T_{N-1} \arrow[r, "/-1"] & T_{N-2} \arrow[r, "/-1"] & \cdots \arrow[r, "/-1"] & T_1 \arrow[r, "/-1"] & s \end{tikzcd}$$

We now use the same construction as above to handle these left moving states. Finally, make these alterations at every node s in the configuration of M. Let $M' = (A, S', B', f')$ be the new machine created from M by the above process. Then $B' = \{-1, 0, 1\}$ and M' is a Turing machine.

Consider now the machines M and M' above with input tape τ. Let M produce the sequence of output tapes $\tau_1, \tau_2, \tau_3, \ldots$ and M' produce the sequence $s_\tau = (\tau'_1, \tau'_2, \ldots)$. In the construction of M', one notices that τ_k appears in the sequence s_τ for each k, i.e. $\tau_k = \tau'_{n(k)} \in s_\tau$ for some $n(k)$.

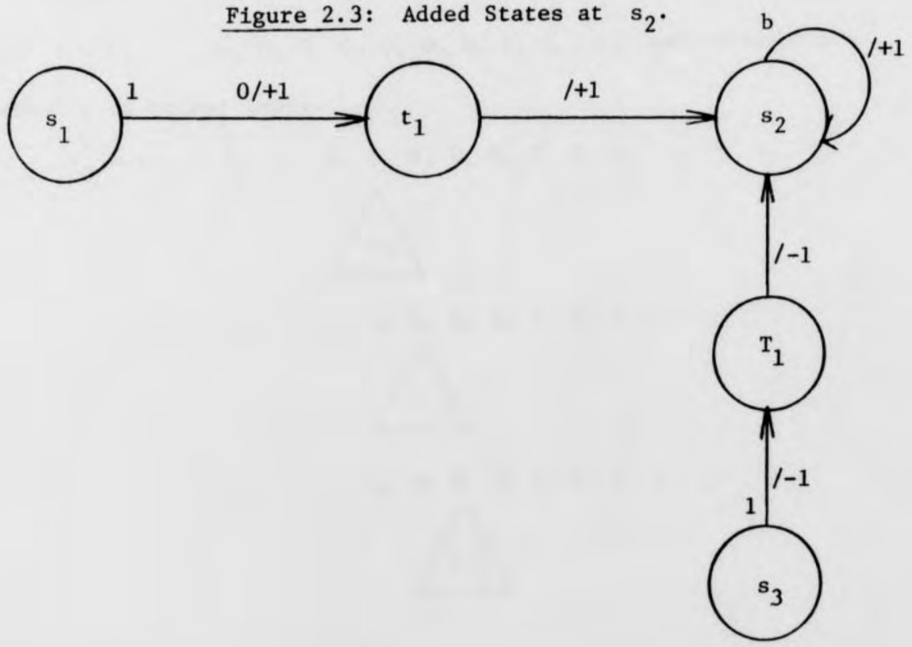
We now have the following:

$$\begin{tikzcd}
\tau_1 \arrow[drr] & \tau_2 \arrow[drrr] & \tau_3 \arrow[drrrr] & \cdots\cdots & & & \\
\tau'_1 & \cdots & \tau'_{n(1)} & \cdots & \tau'_{n(2)} & \cdots & \cdots
\end{tikzcd}$$

It is also seen that the tapes intermediate to $\tau'_{n(k)}$ and $\tau'_{n(k+1)}$ are identical to $\tau'_{n(k)}$ since M' is only concerned with head movement in these added states. Thus we have displayed the required sequence. Q.E.D.

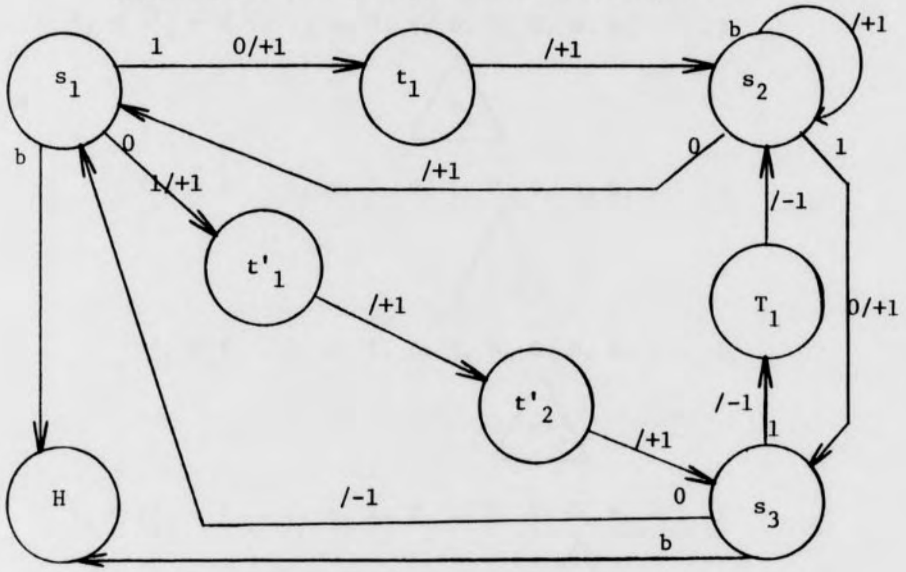
We now apply the technique of Theorem 2.1 to Figure 2.2. The only nodes that require modification are s_2 and s_3 since s_1 has no arrows entering it which have a move such that $| n | > 1$. At s_2 we find $n = 2$ and $N = 2$. Thus for the right moving states we add one state and for the left moving states we also add one state. This is shown in Figure 2.3.

Figure 2.3: Added States at s_2.



Apply the same technique to s_3. We have the new machine as shown by Figure 2.4.

Figure 2.4: The Turing Machine for Example 2.1.



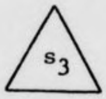
Let $\tau = (. . ., b, 0, 0, 0, 0, 0, 0, b, . . .)$ and consider the sequence of output tapes.

$\tau_0 = \tau'_0 = (. . ., b, 0, 0, 0, 0, 0, 0, b, . . .)$

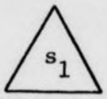
$\tau'_1 = (. . ., b, 1, 0, 0, 0, 0, 0, b, . . .)$

$\tau'_2 = (. . ., b, 1, 0, 0, 0, 0, 0, b, . . .)$

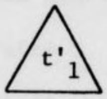
$$\tau_1 = \tau'_3 = (. . ., b, 1, 0, 0, 0, 0, 0, b, . . .)$$



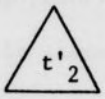
$$\tau_2 = \tau'_4 = (. . ., b, 1, 0, 0, 0, 0, 0, b, . . .)$$



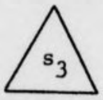
$$\tau'_5 = (. . ., b, 1, 0, 1, 0, 0, 0, b, . . .)$$



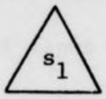
$$\tau'_6 = (. . ., b, 1, 0, 1, 0, 0, 0, b, . . .)$$



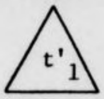
$$\tau_3 = \tau'_7 = (. . ., b, 1, 0, 1, 0, 0, 0, b, . . .)$$



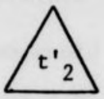
$$\tau_4 = \tau'_8 = (. . ., b, 1, 0, 1, 0, 0, 0, b, . . .)$$



$$\tau'_9 = (. . ., b, 1, 0, 1, 0, 1, 0, b, . . .)$$



$$\tau'10 = (. . ., b, 1, 0, 1, 0, 1, 0, b, . . .)$$



$$\tau_5 = \tau'_{11} = (. . ., b, 1, 0, 1, 0, 1, 0, b, \underset{\triangle\, s_3}{b}, . . .)$$

$$\tau_6 = \tau'_{12} = (. . ., b, 1, 0, 1, 0, 1, 0, b, \underset{\triangle\, H}{b}, . . .)$$

Thus from Example 2.1 and the above, we see that each τ_k of Example 2.1 is in the above sequence and the tape remains unaltered while cycling through the added states.

There are several other ways in which our computing machine may be modified without altering computing power. All of the following classes of machines can perform exactly those computations which can be performed by our particular machine.

(1) We may restrict the alphabet A to contain only two symbols.

(2) The set S may be restricted to contain only two states (the alphabet is necessarily enlarged).

(3) We need only have a set of tapes which are infinite in one direction.

(4) We can require that any input symbol on a tape be altered at most once.

The equivalences of the above classes is shown in the literature and we mention them only as a sampling of how machines can be modified. These are by no means the only alterations we could make. They serve to indicate that our computing machine is certainly not one of a kind.

CHAPTER III
STATE MINIMIZATION

It is quite probable that if several persons were asked to design a computing machine to perform a given computation, that each machine would be different in design. If we are only concerned with the "answers" that machines produce from an input tape, then any of the machines would serve our purpose as long as they had a logical structure which conformed with the algorithm. However, this is seldom the sole criteria in determining whether one machine is somehow preferable to any others which perform the same computation. The features to be considered in the selection of one machine from several should be emphasized in the machine design process. One such feature is utility of design, i.e. minimizing the number of states introduced. We ask the question, "If a computing machine M is given, is there any way to determine whether M has the least number of states necessary for the computation and if not, can we somehow reduce the number of states without changing the action of M?" In this chapter, we will answer the above question by presenting an algorithm which merges redundant states. This algorithm for minimizing the number of states is motivated by the following observation. If $M/(q, \tau)$ and $M/(r, \tau)$ produce the same output tape sequence for each initial tape τ, then it is somehow redundant to list q and r as separate states.

Suppose for $\tau \in T$ that τ_i is the tape produced by $M/(s, \tau)$ in i time units with the sequence of head movements given by $m_1, m_2, m_3, \ldots$ and τ'_i is the tape produced by $M/(s', \tau)$ in i time units with the sequence of head movements $m'_1, m'_2, m'_3, \ldots$. Consider the following

<u>Definition 3.1</u>: State s is E_n - equivalent to state s', denoted sE_ns', if $\tau_i = \tau'_i$ and $m_i = m'_i$ for $1 \leq i \leq n$, and for all $\tau \in T$.

It should be noted that Definition 3.1 treats the head movements as an integral part of the output. This treatment is crucial to the development of the algorithm for state minimization. As we shall show in Chapter IV, disregarding the sequence of head movements introduces a significant obstacle to further generalization of the notion of state equivalence.

We observe that the relation E_n on the set S is reflexive, symmetric, and transitive. Thus we have the following lemma.

<u>Lemma 3.1</u>: E_n is an equivalence relation on the set S.

Since E_n is an equivalence relation, we have a partitioning of the set of states S into equivalence classes. States s and s' belonging to the same equivalence class for E_n cannot be distinguished by observing the output for n time units or less. Thus we could consider s and s' as <u>approximately</u> equal if sE_ns' for large n. To be indistinguishable regardless of the machine run time, we define

<u>Definition 3.2</u>: s is said to be equivalent to s', denoted sEs', if sE_ns' for $n = 1, 2, 3, \ldots$.

Note that E is an equivalence relation on S. Also, if sEs', then s and s' could be merged into a single state. Let P_k be the set of equivalence classes for E_k and let P denote the equivalence classes for E. We then have

Lemma 3.2: States which belong to different sets in P_k belong to different sets in P_{k+1}.

Proof: Pick s, s' in S such that $s \not{E}_k s'$. Then there is some $\tau \in T$ such that either $\tau_i \neq \tau'_i$ or $m_i \neq m'_i$ for some i, $1 \leq i \leq k$. Since $1 \leq i \leq k < k + 1$, by Definition 3.1 we have $s \not{E}_{k+1} s'$. Q.E.D.

By the contrapositive of Lemma 3.2, we know that if $sE_{k+1}s'$, then sE_ks'. We then make the following observations.

(1) Two states that belong to different equivalence classes in P_k will belong to different equivalence classes in P_ℓ for $\ell > k$.

(2) To check two states for (n + 1) - equivalence, we need only consider those states which are already n-equivalent.

Let qE_nr. Assume that for a particular initial input symbol, that q and r transition to q' and r' respectively. If q and r are to be (n + 1) - equivalent, then q' and r' must be n-equivalent. This is true because one iteration with initial states q and r moves the read/write head to the same new position. This new position was previously considered as a starting position in checking the n-equivalence of q' and r'. If the transition states after one iteration (for each input symbol) are n-equivalent, then $qE_{n+1}r$.

To determine the equivalence classes for E, we must find the equivalence classes for E_n, $n = 1, 2, 3, \ldots$. The following lemma shows that this process must eventually terminate.

Lemma 3.3: There exists $N > 0$ such that $P_k = P_N$ for all $k \geq N$.

Proof: Assume the statement is not true, i.e. assume for all $n > 0$ there exists $s(n) > n$ such that $P_n \neq P_{s(n)}$. Then we have $P_1 \neq P_{s(1)} \neq P_{s(s(1))} \neq \ldots$ and, by assumption, $1 < s(1) < s(s(1)) < \ldots$. But since P_k is a refinement of P_n for any $k > n$, this implies that we have an infinite number of states. This is a contradiction to $|S| < \infty$. Q.E.D.

Perhaps we should examine just what has been established. Suppose we are given a machine M containing the states q and r, and we wish to determine whether qEr or $q\not{E}r$. We proceed as follows (see Figure 3.1):

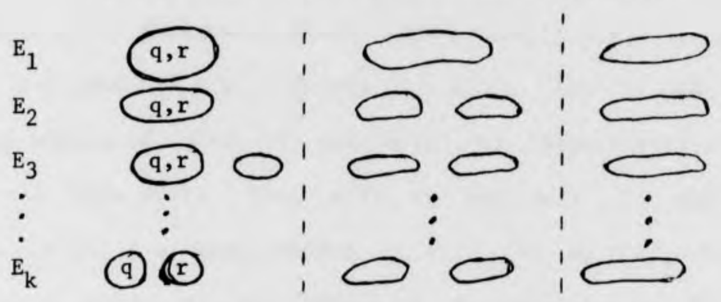
(1) check q and r for E_1 - equivalence

(2) if $q\not{E}_1r$, then $q\not{E}r$; otherwise check q and r for E_2 - equivalence

(3) if $q\not{E}_2r$, then $q\not{E}r$; otherwise check q and r for E_3 - equivalence

(4) continue this process until $q\not{E}_kr$ for some k.

Figure 3.1: Equivalence Class Refinements



By Lemma 3.2, we know that $q\not{E}r$ if q and r ever fail to belong to the same equivalence class in P_k for some k. We also know by Lemma 3.3 that if qE_Nr, then qEr. However, we have no way of knowing the value of N. Let up suppose $N > 1000$. Further, let us assume that $qE_{1000}r$. We might then conclude that qEr. But what is to prevent q and r from belonging to different equivalence classes at some time $k > 1000$? Fortunately, we are not only assured of being able to determine whether q and r are equivalent, but we can also determine the maximum number of iterations required to determine their equivalence or non-equivalence. The following development provides the necessary stepping stones toward our ultimate aim; establishing a bound for the number of iterations required to determine the equivalence or non-equivalence of states in any machine M.

Lemma 3.4: If a machine M contains two states s and s' such that $s\not{E}s'$ but sE_ks' for some k, then M contains two states t and t' such that tE_kt' but $t\not{E}_{k+1}t'$.

Proof: Let ℓ be the first time unit for which $M/(s, \tau)$ and $M/(s', \tau)$ disagree. Since $sE_k s'$, then $\ell > k$. If $\ell = k + 1$, let $t \equiv s$ and $t' \equiv s'$. Assume $\ell - k > 1$. Let t and t' be the present states of $M/(s, \tau)$ and $M/(s', \tau)$ respectively after $\ell - k - 1$ time units. Then $M/(t, \tau)$ and $M/(t', \tau)$ must disagree for $\ell - k \leq i \leq \ell$ since either $\tau_\ell \neq \tau'_\ell$ or $m_\ell \neq m'_\ell$ by assumption ($M/(s, \tau)$ and $M/(s', \tau)$ disagree at time ℓ). But from time $\ell - k$ to time ℓ there are $k + 1$ time units. Hence $t\not{E}_{k+1}t'$.

We now claim $tE_k t'$. If not, then there exists $\tau \in T$ such that $\tau_i \neq \tau'_i$ or $m_i \neq m'_i$ for some i, $1 \leq i \leq k$. But this implies that $M/(s, \tau)$ and $M/(s', \tau)$ disagree at least by time $\ell - 1$, a contradiction to our assumption. Q.E.D.

Theorem 3.1: $P_k = P_{k+1}$ if and only if $P_k = P$.

Proof: The if part is clear since $P_k = P$ obviously implies $P_k = P_{k+1}$. For the only if, we take the contrapositive of Lemma 3.4 which states: if for any states t and t' for which $tE_k t'$ we have $tE_{k+1}t'$, then there are no states s and s' for which $sE_k s'$ and $s\not{E}s'$. Q.E.D.

Lemma 3.5: If $P_{k-1} \neq P_k$, then $| P_k | \geq k + 1$.

Proof: Assume $P_{k-1} \neq P_k$. Then $P_{r-1} \neq P_r$ for $r = 1, 2, \ldots, k$. Now by the contrapositive of Lemma 3.2, the classes of P_r are further refinements of the classes of P_{r-1}. Then $| P_r | > | P_{r-1} |$. Suppose $| P_1 | = 1$. Then $P_1 = P$ which contradicts our assumption that $P_{k-1} \neq P_k$. So $| P_1 | \geq 2$. By iteration we have $| P_k | \geq k + 1$. Q.E.D.

We now have the capability whereby we can make a definitive statement as to the maximum number of iterations required to determine the equivalent states for any machine M. The crucial question as to when our minimization procedure will terminate is answered by the following theorem.

Theorem 3.2: In an N-state machine, $P_{N-1} = P_N = P$.

Proof: $P_{N-1} \neq P_N$ implies by Lemma 3.5 that $| P_N | \geq N + 1$ which is impossible since there can be at most N subsets in a partition of the N states. Q.E.D.

The algorithm we have constructed is meaningful for any computing machine M. We will first give the procedure for determining the equivalence classes for E and then apply this process to a specific machine.

(1) Determine P_1; i.e., s and s' belong to the same equivalence class if the output and head movement rows of the computing function table for $M/(s, \tau)$ and $M/(s', \tau)$ are identical.

(2) Determine P_{k+1} from P_k. Two states, s and s', <u>in the same class</u> in P_k stay together in P_{k+1} if and only if for each input symbol, their respective next states are together in P_k, after identical head movements.

(3) Stop when for some k, $P_k = P_{k+1}$. We then have $P = P_k$. This must occur for some $k \leq | S | - 1$.

Consider the following computing function table for a machine M.

Table 3.1: A Computing Function Table for a Non-minimized Computing Machine

State	Input	Output	Next State	Head Movement
s_1	0	0	s_1	+1
s_1	1	0	s_6	+1
s_2	0	0	s_1	+1
s_2	1	0	s_4	+1
s_3	0	1	s_2	+1
s_3	1	0	s_5	+1
s_4	0	1	s_5	+1
s_4	1	1	s_8	+1
s_5	0	1	s_1	+1
s_5	1	0	s_3	+1
s_6	0	1	s_8	+1
s_6	1	1	s_5	+1
s_7	0	1	s_6	-1
s_7	1	1	s_3	-1
s_8	0	1	s_2	+1
s_8	1	0	s_5	+1

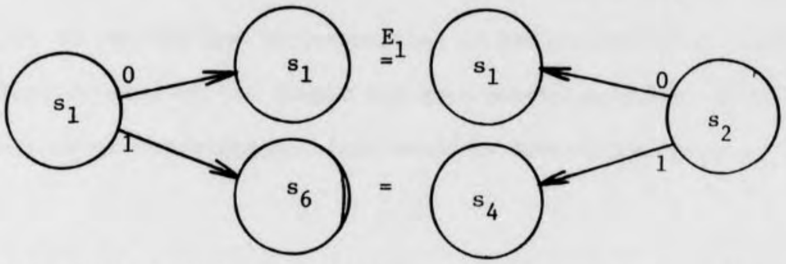
From the table we observe that

$$P_1 = \{(s_1, s_2), (s_3, s_5, s_8), (s_4, s_6), (s_7)\}.$$

To determine P_2, we observe that $s_1 E_1 s_2$. Thus we test s_1 and s_2 for E_2 - equivalence. The next states for s_1 are s_1 and s_6

respectively; for s_2 they are s_1 and s_4. Clearly $s_1 E_1 s_1$. Since $s_6 E_1 s_4$, we determine that $s_1 E_2 s_2$.

Figure 3.2: Sample Calculation of P_2.



Applying this technique to all E_1 - equivalent states, we have $P_2 = \{(s_1, s_2), (s_3, s_5, s_8), (s_4, s_6), (s_7)\}$. Now $P_1 = P_2$, so $P_1 = P$. Hence we merge states s_1 and s_2, s_3 and s_5 and s_8, and finally s_4 and s_6. Since s_7 is not E_1-equivalent to any other state, it is retained as a separate state. The minimized machine is denoted by Table 3.2.

Table 3.2: The Minimized Computing Machine

State	Input	Output	Next State	Head Movement
s'_1	0	0	s'_1	+1
s'_1	1	0	s'_3	+1
s'_2	0	1	s'_1	+1
s'_2	1	0	s'_2	+1
s'_3	0	1	s'_2	+1
s'_3	1	1	s'_2	+1
s'_4	0	1	s'_3	-1
s'_4	1	1	s'_2	-1

Notice that the new state s'_1 now represents the merged state created from s_1 and s_2. The merging of redundant states under the equivalence discussed above has physical significance for the person required to build such machines. But beyond that, the esthetic beauty contained in the derivation and implementation of the minimization procedure is typical of many of the deeper and more subtle questions about the structure of computing machines asked by modern theorists.

CHAPTER IV

A GENERALIZATION OF STATE EQUIVALENCE

The connotation probably most often associated with the term equivalence is one of "sameness". We classify objects as equivalent if they have the same worth, possess the same features, or perform the same function. Even though two procedures were entirely different in structure yet both produced the same results, we would consider them equivalent if results were our only concern. Recall that the function of a machine is to mechanize a computation, the results of which appear as an output tape. Thus it is natural to consider two states, s and s', of a machine M equivalent if for every $\tau \in T$, $M/(s, \tau)$ and $M/(s', \tau)$ produce the same output tape in any amount of time. In Chapter III, we defined an equivalence relation on the set of states S in which the sequence of head movements was considered to be an essential part of the output. But in actual practice, our only concern would be the output tape and not the particular motions of the read/write head. In the following discussion, we will explore the ramifications of altering the definition of state equivalence as given in Chapter III to the more natural concept we have introduced here.

For $\tau \in T$, let τ_i be the output tape produced by $M/(s, \tau)$ in i time units and let τ'_i be the tape produced by $M/(s', \tau)$ in i time units.

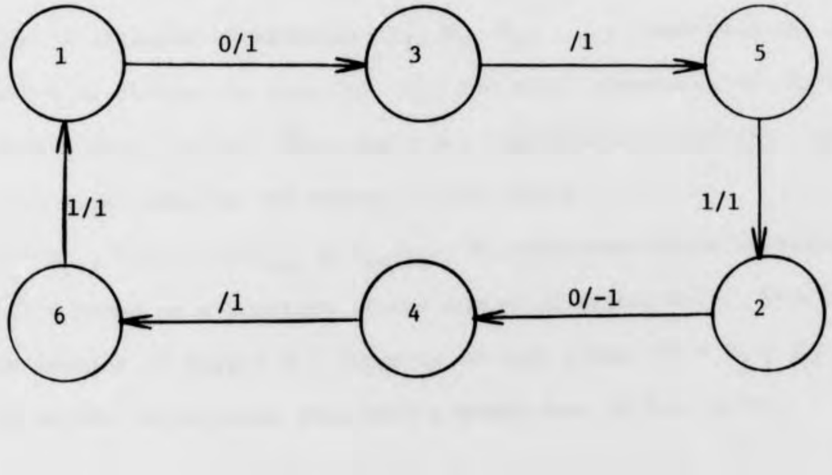
<u>Definition 4.1</u>: State s is R_n - equivalent to state s', denoted sR_ns', if $\tau_i = \tau'_i$ for $1 \le i \le n$, and for all $\tau \in T$. State s is equivalent to s', denoted sRs', if sR_ns' for $n = 1, 2, 3, \ldots$.

In considering Definition 4.1, we think of head movement as invisible to the machine observer. All he can see is the sequence of output tapes for each input tape.

We observe that R_n and R are equivalence relations on S. Let P_k and P be the equivalence classes for R_k and R respectively. The reader can easily verify that Lemma 3.2 and Lemma 3.3 are still true; that is, states which belong to different sets in P_k belong to different sets in P_{k+1}, and there exists $N > 0$ such that $P_k = P_N = P$ for all $k \ge N$.

Consider the machine of Figure 4.1.

<u>Figure 4.1</u>: State Graph of a Computing Machine



We determine the equivalence classes for R_k to be

$$P_1 = \{(1, 2), (3, 4), (5, 6)\}$$

$$P_2 = \{(1, 2), (3, 4), (5, 6)\}$$

$$P_3 = \{(1), (2), (3, 4), (5, 6)\}$$

$$P_4 = \{(1), (2), (3, 4), (5), (6)\}$$

$$P_5 = \{(1), (2), (3), (4), (5), (6)\}$$

and we observe that $P_1 = P_2 \neq P_3 \neq P_4 \neq P_5 = P$. Theorem 3.1 fails since we have $P_1 = P_2$ but $P_1 \neq P$. This means the algorithm for determining P, as constructed in Chapter III, is meaningless. A replacement for this algorithm would necessarily calculate $P_1, P_2, \ldots$ until P was encountered. A bound on the first occurrence of P in the sequence $P_1, P_2, \ldots$ would be needed to indicate the point at which the algorithm could be terminated. We pose the following question: is the number of iterations N required to determine P still bounded by some function of the number of states only?

If the answer to the above question were no, then there would exist a sequence of machines $M_1, M_2, M_3, \ldots$, each with the same number of states, so that for M_k, the first occurrence of $P_N = P$ is such that $N \geq k$. This could not happen if we could show that there is a bound on the number r for which $P_k = P_{k+1} = \ldots = P_{k+r} \neq P_{k+r+1}$. We have been unable to display such a bound as a function of the number of states only. Even though the machine of Figure 4.1 suggests no such bound ($P_1 = P_2 \neq P_3$), the author conjectures that such a bound does in fact exist.

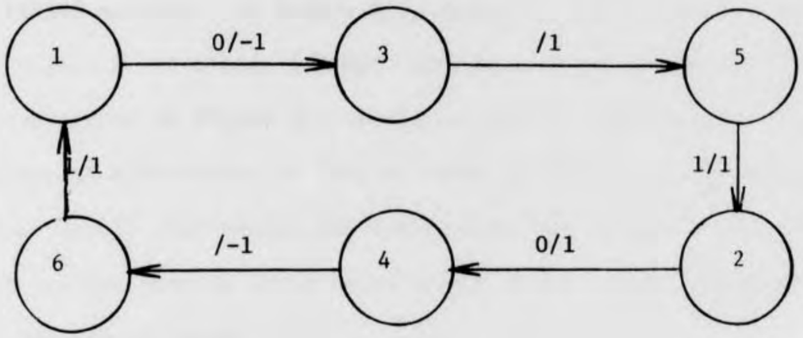
This conjecture is based upon experimentation and the following intuitive argument. Assume C is the set of L-state computing machines. If there are pairs of non-equivalent states of machines in C which are output indistinguishable unless observed for arbitrarily long periods of time, then the "memory" of the initial states is somehow contained in the state designations. But this would attribute an unlimited capacity for differentiation to the finite number of states. This is contrary to intuition.

An affirmative answer to the question of bounding N would enable us to more naturally define a notion of state equivalence and devise an effective algorithm for calculating P. In either case, it should be noted that P_n is no real help in finding P_{n+1}. This is because we are no longer concerned with head movement. It is possible for the relative positions of the read/write head to be different after one unit of time for $M/(s, \tau)$ and $M/(s', \tau)$. At this point, the two copies of machine M would be considering two different tapes. This is no obstacle however, since we can calculate P_{n+1} directly (by use of brute force).

We will now consider the minimization problem with our new type of equivalence. Though we have no effective algorithm for calculating P for every computing machine, we can find P for the machine of Figure 4.2. The equivalence classes for R are

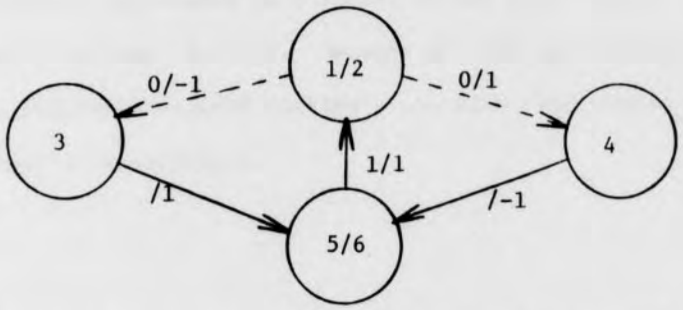
$$P_2 = \{(1, 2), (3), (4), (5, 6)\} = P$$

Figure 4.2: A Non-minimized Machine



Since $| P | = 4$, we construct a machine with only four states which produces identically the same output as the machine of Figure 4.2. (See Figure 4.3.)

Figure 4.3: Minimized Machine



In this minimized machine, states 3 and 4 transition to state 5/6 and state 5/6 transitions to state 1/2. But how do we transition out of state 1/2? More generally, suppose we merge states $s_1, \ldots, s_k$ of some machine M. Then for each input symbol, we have an arrow in the original machine, with the input symbol at the base, emanating from each state s_i. With our old definition of equivalence, equivalent states always transitioned to equivalent states, so we had no choice for the associated transition in the minimized machine. In Figure 4.2, state 1 transitions to state 3 and state 2 to state 4. Now 1R2, but $3\not R4$; hence which of the dotted arrows of Figure 4.3 should we choose? Fortunately, the essence of equivalence is that it makes no difference; we can choose either arrow. The reader can verify that the machine of Figure 4.3 will perform exactly those calculations which can be performed by the machine of Figure 4.2.

We also make the following observation. If we minimize the machine of Figure 4.2 using the equivalence of Chapter III, we find that it is already minimal, i.e., six states are required. However, only four states are required with the equivalence of Definition 4.1. This "coarser" refinement is a result of the fact that if sEs', then sRs' for any $s,s' \in S$. Because of this implication, we can obtain a smaller minimized machine - one with fewer states - with this new concept of equivalence.

CHAPTER V

SUMMARY

In this paper, we have discussed the theory of computation and computing machines. It was intended that this discussion remain as intuitive as possible, for the author subscribes to the opinion of Minsky [4] that classical mathematical analysis cannot play a significant role in this area.

The concepts of computation and computing machine were formalized in Chapter II. We also investigated the question of what restrictions could be placed on computing machines without altering their computing power. In Chapter III, the notion of equivalent states was discussed, and we constructed an algorithm to accomplish a minimization of any machine. In Chapter IV, the read/write head movement was ignored as part of the machine output, and the problem of recognizing equivalent states was discussed.

BIBLIOGRAPHY

1. Church, Alonzo (1936), "An unsolvable problem of elementary number theory," Amer. J. Math. 58, 345-363.

2. Gödel, Kurt (1931), "Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I," Monatshefte fur Mathematik und Physik 38, 173-198.

3. Kleene, Stephen C. (1952), Introduction to Metamathematics, Van Nostrand, Princeton.

4. Minsky, Marvin L. (1967), Computation: Finite and Infinite Machines, Prentice-Hall.

5. Post, Emil L. (1943), "Formal reductions of the general combinatorial decision problem," Amer. J. Math. 65, 197-268.

6. Smullyan, Raymond (1962), Theory of Formal Systems, Princeton.

7. Turing, Alan M. (1936), "On computable numbers, with an application to the Entscheidungsproblem," Proc. London Math. Soc., Ser 2-42, 230-265.