COMPUTATION AND UNSOLVABILITY

by 1264

STEVEN KENT THORNBRUGH

B. Sc., Kansas State University, 1968

A MASTER'S REPORT

submitted in partial fulfillment of the

requirements for the degree

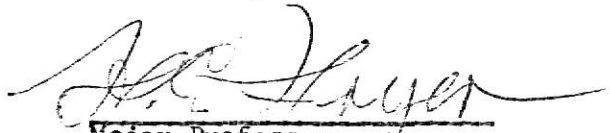
MASTER OF SCIENCE

Department of Statistics and Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1969

Approved by:



Major Professor

TABLE OF CONTENTS

Chapter I

Introduction

I.1 Purpose

Each day the role of the modern electronic digital computer-- herein referred to as the computer-- becomes larger and larger. The tasks presented to the computer are growing at unimaginable speed. As such, questions arise as to what kinds of problems the computer can solve? Or better yet, what programming restrictions the computer has, if any?

The purpose of this paper is to answer such questions. To define the limits of the computer and the computation, in general, a precise definition for an algorithm will be given along with some characterization for representing such algorithms. Only when this is done is it possible to discuss problems which are algorithmically unsolvable.

This paper discusses both computation and unsolvability in mathematical terms which are for the most part self-contained within the paper. Section 1.2 defines the mathematical representation necessary to define computability and unsolvability.

I.2 Definitions

Unless otherwise stated the following definitions will be used throughout the paper. The letters x or y or those letters in the vicinity of x and y will represent variables whose values assume the values of the positive integers. The symbol '(prime) when used in conjunction with a variable, such as x', will denote a constant. The letters i, j, k, n, as well as those in the vicinity, will be used as subscripts on variables. The variables f, g, h and those also in the vicinity will represent functions with arbitrary lengths of arguments. Arguments can be either variables, or constants, or

other functions which evaluate to constants. The symbols Φ, Ψ, and λ represent number-theoretic functions whose values and arguments are natural numbers. An n-tuple or k-tuple represents an expression, such as $(a_1, a_2, \ldots a_n)$ where the a's are taken to represent integer constants. The juxtaposition of functions such as fg will represent f(g(x)) where x may represent any n-tuple.

The operator $\longrightarrow$ means "goes to". The operator $\neg$ stands for "not" or "negation". The operator $\subset$ means "is a subset of" while $\subseteq$ means "is a subset of or equivalent to", where the equivalent to means that for two n-tuple's of the form $(a_1, a_2, \ldots, a_n)$, $(b_1, b_2, \ldots, b_n)$ then, $a_1 = b_1, \ldots, a_n = b_n$. The symbol $\in$ means "is an element of". The symbols { }, or ( ), when enclosing integers, generally indicates a closed set. The symbol $\vdash$ means replacement while $\models$ means terminal replacement, or replacement which ceases after this replacement is made. The operator $\overline{\ }$ (bar) when used with an integer, means the number of 1's that that integer represents plus one. If x has a value of 3 then $\overline{x}$ = 1 1 1 1. The last operator is a prime used in conjunction with ( ). The value of the expression is the successor to the value enclosed in parentheses. For example, if x has a value of 3 then $(x)' = 4$ or 3+1.

The capital letters A, B, and those in the vicinity represent alphabets which contain all legal character representations. The letter S will represent any character in A or B. The capital letter P or Q will represent words made up of characters $S_1 \ldots S_k$ where the S's may be null. PQ is the juxtaposition of words. The symbol Λ represents the null set and B on occasions will stand for the "blank" symbol. The symbol $\approx$ stands for "is similar to".

Chapter II

Recursive Functions and Computational Representation

2.1 The Notion of an Algorithm

The term algorithm in first light appears to be a fairly simple concept. The informal meaning generally associated with the concept algorithm is that of a sequence of instructions specifying some sequence of operations which will yield an output for a given class of problems. Rogers (1967) suggests a more complete and meaningful definition of an algorithm as follows:

1) An algorithm is a specified set of instructions of finite size.

2) There is some form of computing agent which reacts with these instructions to carry out the computations.

3) The computing agent has available to it facilities for making, storing, and retrieving various steps in a computation.

4) If P is a set of instructions as in 1 and L is the computing agent in 2, then L reacts to P in a manner such that, for any given input, the computation is carried out in some discrete stepwise fashion.

5) The reaction of L to P carries forward the computation deterministically.

These five steps seem inherent in the definition of any algorithm, whether those of pure mathematics or those of some applied mathematical areas such as can be evidenced by computer programming. But these five steps alone do not give the precise picture of an algorithm. One must decide whether there are any limits to be placed upon the definition of inputs, set of instructions, memory, capacity of the computing agent, and finally on the length of time for a computation.

All computing machinery is bounded by the above five restrictions. Is it not then justifiable to also put such restrictions on our definition

of an algorithm? For theoretical purposes the answer is a firm no! The size of inputs need not be bounded in any form. Similarily, there need be no restriction as to the size of the sets of instructions used by the algorithm. Memory size need not be limited for if, when computation is about to exceed available space more external memory can be added. It is necessary, however, to put a restriction on the computing agent L. Since the set of instructions and memory are unbounded it is not necessary to allow an unrestricted computing agent. In this case, the computing agent L is kept simple at the sacrifice of efficiency. This parallels doing such complex numerical problems such as multiplication of x_1 timex x_2 as x_1 successive additions of x_2. as for the final question of length of computation time, the only limit required is that computations end in a finite number of steps. The reason for bounding computation length this way is due to the technological advancement of recent years. Faster and faster computer hardware render the question of time meaningless.

2.2 Primitive Recursive Functions

In light of the preceding definition of algorithm the definition of recursive functions can now be given. Kleene (1962) defines primitive recursion definition by induction. It is merely an algorithmic way of expressing functions in terms of simpler or more "primitive" functions. By the most primitive function is meant the simplest function within a given class of functions. For example, the Fibonacci sequence 1, 1, 2, 3, 5, 8, ... can be generated as follows:

$$\begin{aligned} \text{let} \quad & g(0) = 1, \\ & g(1) = 1, \\ & g(x+2) = g(x+1) + g(x) \ ; \ x = 0, 1, \ldots \end{aligned}$$

The functions $g(0) = 1$ and $g(1) = 1$ are the primitive functions.

All primitive recursive functions can be generated by the application of the following algorithm:

1) $\Phi(x_1, x_2, \ldots, x_k) = x'$; $1 \leq k$, $0 \leq x'$ (Constant)

2) $\Phi(x_1, x_2, \ldots, x_k) = x' + 1$ (Successor)

3) $\Phi(x_1, x_2, \ldots, x_i, \ldots, x_k) = x_i$; $1 \leq i \leq k$ (Identity)

4) If Φ is a function of k variables and $\Psi_1, \Psi_2, \ldots \Psi_k$ are functions of m variables then the function $\Phi(\Psi_1(x_1, x_2, \ldots x_m), \Psi_2(x_1, x_2, \ldots x_m), \ldots, \Psi_k(x_1, x_2, \ldots, x_m))$ is defined.

5) If λ is a function of k + 1 variables and Ψ is a function of k - 1 variables and Φ is a function of k variables then

$$\Phi(0, x_2, \ldots, x_k) = \Psi(x_2, \ldots, x_k)$$

$$\Phi(y+1, x_2, \ldots, x_k) = \lambda(y, \Phi(y, x_2, \ldots, x_k), x_2, \ldots, x_k)$$

are both defined.

An example of multiplication by recursion will help illustrate the above definition. Let g(2x) be given the following recursive derivation:

$g_1(x) = x$ from (3) for one variable

$g_2(x) = x+1$ from (2) for one variable

$g_3(x_1, x_2, x_3) = x_2$ from (3) for three variables

$g_4 = g_2 g_3$ from (4) for three variables

g_5 must satisfy:

$g_5(0, x_2) = g_1(x_2)$ from (5) for two variables

$g_5(y+1, x_2) = g_4(y, g_5(y, x_2), x_2)$

$g = g_6 = g_5(g_1, g_1)$ from (3) for one variable

When g(3) is given, the algorithm computes in successive steps

$$g(3) = g_6(3)$$

$$= g_5(g_1(3), g_1(3))$$

$$= g_5(3, g_1(3))$$

$$= g_5(3,3)$$

$$= g_4(2,\ g_5(2,3),\ 3)$$
$$= g_4(2,\ g_4(1,\ g_5(1,\ 3),\ 3),\ 3)$$
$$\vdots$$
$$= g_2(5)$$
$$= 6$$

From the above example it is easily seen that primitive recursion functions appear to work algorithmically. Rogers (1967) makes the distinction between algorithm and algorithmic function, however. Algorithms always yield solutions while algorithmic functions may or may not give solutions for given inputs.

2.3 The Markov Characterization of Algorithms

Modern compiler and grammar theories are characterized in much the same way as Markov characterized algorithms. His algorithmic [Mendelson (1964)] theories are centered around the notion of "effective computability" which is to find effective procedures for solving large classes of problems. The classes of problems are formulated as expressions of some particular language. Expressions are considered as sequences of valid configurations of the language. The number of valid expressions may be countable. Blanks separate words in the expressions. An algorithm in alphabet A is defined if there is an effective computable function μ whose domain is a subset of the set of words in A. The "algorithm schema" [Mendelson (1964)] is defined as a set of "productions" from alphabet A where P and Q are words in A, and the productions are of the form $P \rightarrow Q$ or $P \rightarrow (\neg)Q$. The first production is termed "simple", the second, "terminal." The finite productions might include

$$P_1 \longrightarrow (\neg)Q_1$$
$$P_2 \longrightarrow (\neg)Q_2$$
$$\vdots$$
$$P_r \longrightarrow (\neg)Q_r$$

as its algorithmic schema.

The Markov algorithm or normal algorithm is stated as follows: The word T occurs in Q if there exist, words U and V (possibly null) such that Q = UTV. If P is a word in alphabet A then either: 1) U = P if there is no production P_i, i=1, ..., r in which P occurs, or 2) If R is a word which results from replacing the left most occurrence of P_m in P by Q_m — where m is the first production where P_m occurs in P — then let

a) $\mathcal{M}$ = P⊢ R if $P_m \rightarrow (\neg)Q_m$ is simple, or

b) $\mathcal{M}$ = P⊢ ¬R if $P_m \rightarrow (\neg)Q_m$ is terminal.

$\mathcal{M}$ is defined as a) P ⊨ R or the sequence $R_0, R_1, \ldots, R_k$ such that P = R_0 ; R = R_k or b) $R_j \vdash R_{j+1}$ for $0 \leq j \leq (k-2)$ and c) as either i) $R_{k-1} \vdash R_k$ or ii) $R_{k-1} \vdash \neg R_k$.

The above can be described as the action of $\mathcal{M}$ on a given word P where the productions are searched until a production $P_m \rightarrow (\neg)Q_m$ occurs in P. Q_m is then substituted into the left most occurrence of P_m in P. R_i is the new word obtained for each ith step. When the production $P_m \rightarrow (\neg)Q_m$ is terminal the process stops and the value of the algorithm is R_K. If the process never terminates in a finite number of steps the process $\mathcal{M}$ is not applicable to the word P.

Given algorithms $\mathcal{M}$ in β and a word P, $\mathcal{M}(P)$ is similar to $\beta(P)$ if and only if either $\mathcal{M}$ or β are both applicable to P and $\mathcal{M}(P) = \beta(P)$ or neither $\mathcal{M}$ or β are applicable to P. Algorithms $\mathcal{M}$ and β are fully equivalent relative to alphabet A if and only if $\mathcal{M}(P) \approx \beta(P)$ for every word P in A. $\mathcal{M}$ and β are equivalent in alphabet A if and only if $\mathcal{M}(P)$ or $\beta(P)$ exists in A for every P which implies $\mathcal{M}(P) \approx \beta(P)$.

If N is an alphabet consisting of $\{1, \Lambda\}$ then let the function Φ represent a partial effectively computable function of the natural

numbers with K arguments given for Φ. Denote β_{Φ} as the corresponding algorithm in N. It follows that $\beta_{\Phi}(\overline{x_1, x_2, \ldots, x_k}) = \Phi(\overline{n_1, n_2, \ldots, n_k})$ if either function is defined. The function Φ is partially Markov-Computable if and only if there exists a normal algorithm μ over alphabet N which is fully equivalent to β_{Φ}. If Φ is partially Markov-Computable and Φ is defined for all k-tuples of integers, the Φ is said to be Markov-Computable. From a series of proofs, beyond the scope of this paper, it can be shown that any composition of Markov-Computable algorithms yields a valid algorithm and that all primitive recursive functions are Markov-Computable or partially Markov-Ccomputable.

2.4 The Kleene Characterization of Algorithms

Again consider the set of instructions P over some alphabet A. Further, let P be any recursive function. If each successive P is derived from the preceding P either by direct numerical substitution or by "equals for equals" [Rogers (1967)] substitution than a "computation" occurs in a finite sequence.

In addition to the five recursive functions in 2.2, a set of auxiliary functions (ψ) must be defined. The auxiliary functions tend to make the computation more efficient by increasing the size of the computing agent L as defined in 2.1.

Computation of $\Phi(2x)'$ is evaluated from the derivation of Φ as

$$\Phi(0) = 0$$
$$\psi_1(x) = \Phi(x) + 1$$
$$\psi_2(x) = \psi_1(x) + 1$$
$$\Phi(x+1) = \psi_2(x) + 1$$

Auxiliary functions cause three inherent difficulties. First is the fact that the computation is not directly determined from the inputs. Secondly, several possible outcomes may be obtained from the same inputs

if several different computations are tried. Finally, it is possible that no outcomes may occur at all [Rogers (1967)].

The question arises, can one avoid the above difficulties? The first two can be avoided by using a technique similar to the "British Museum Algorithm" of Simon, Newell, and Shaw [Feigenbaum (1963)]. The process is that of generating all possible equations deducible from P and selecting the first or principal output for P unique to the input. Hence, the system is standardized since now there is only one unique output for each input. This procedure will yield what Kleene [Rogers (1967)] called a partial function. Terms are then defined as follows:

A variable is a term;

A number is a term;

If τ is a term then $\tau + 1$ is a term;

If ∇ is a function with terms as arguments

then ∇ is a term;

Equations are denoted as terms separated by an "=" sign. The set of instructions P is made up of all possible (finite) sequences of equations. All derivable equations can likewise be generated by a technique similar to the "British Museum Algorithm". It can be shown that this definition of computation will generate all recursive or primitive recursive functions.

2.5 The Herbrand-Gödel Characterization of Algorithms

Kleene (1962) defines an enumerable set as one which has a one-to-one correspondence with the positive integers. Thus 1, 4, 9, 16, ..., n^2, ... is an enumerable set with 1, 2, 3, 4, ..., n, ... Herbrand and Gödel [Mendelson (1964)] suggest the following definition of computability on recursively enumerable sets.

Define terms as:

a) All variables are terms

b) 0 is a term

c) If t is a term, then (t)' is a term.

d) If $t_1, \ldots, t_n$ are terms and f_j^n is a function, then $f_j^n(t_1, \ldots, t_n)$ is a term.

All natural numbers are defined. The number $\overline{0}$ is 0 and $\overline{n+1}$ is $(\overline{n})'$. Equations are made up of terms separated by an equal sign. E is a system of equations of the form $r_1 = s_1, r_2 = s_2, \ldots, r_k = s_k$ where r_k is of the form $f_j^n(t_1, \ldots, t_n)$. The function f_j^n is called the principal letter of E and all r_i, $1 \leq i \leq k-1$ are called initial letters of E. All functions on the right-hand side of an equal sign are called auxiliary letters of E.

Two rules of inference concerning a computation are

R 1: An equation e_2 results from e_1 by R_1 if and only if e_2 arises from e_1 by substitution of a number $\overline{n}$ for all occurrences of a variable.

R 2: An equation e is a result of R_2 for equations $f_n^m(\overline{n}_1, \overline{n}_2, \ldots, \overline{n}_m) = \overline{q}$ and r = s if and only if e arises from r = s by replacing one or more occurrences of $f_n^m(\overline{n}_1, \overline{n}_2, \ldots, \overline{n}_m)$ in s by $\overline{q}$, and r = s contains no variables.

Consider the example in the system E where there are three function letters f_1^1 and f_1^2 and f_1^3. Let,

$$f_1^1(x_1) = (x_1)'$$

$$f_1^2(x_1, x_2) = f_1^3(\overline{2}, x_2, f_1^1(x_1))$$

The principal letter of E is f_1^2 and f_1^1 is an auxiliary letter while f_1^3 is an initial letter. The following sequence shows that $f_1^2(\overline{2}, \overline{1}) = f_1^3(\overline{2}, \overline{1}, \overline{3})$ from E.

$$f_1^2(x_1, x_2) = f_1^3(\bar{2}, x_2, f_1(x_1))$$
$$f_1^2(\bar{2}, x_2) = f_1^3(\bar{2}, x_2, f_1^1(\bar{2}))$$
$$f_1^2(\bar{2}, \bar{1}) = f_1^3(\bar{2}, \bar{1}, f_1^1(\bar{2}))$$
$$f_1^1(x_1) = (x_1)'$$
$$f_1^1(\bar{2}) = (\bar{2})' \text{ or } \bar{3}$$
$$f_1^2(\bar{2}, \bar{1}) = f_1^3(\bar{2}, \bar{1}, \bar{3})$$

A partial number theoretic function $\Phi(x_1, \ldots, x_n)$ is said to be computable on E if and only if the principal letter on E is a letter f_j^n with n arguments and is defined for all $k_1, \ldots k_n$, and p. The function Φ is Herbrand-Gödel computable if and only if there is some system E where Φ is computable. As in 2.3 and 2.4 the Herbrand-Gödel characterization will compute all recursive or partial recursive functions.

Chapter III

Turing Machines

3.1 The Turing Characterization of Algorithms

Turing Proposed another type of configuration which has a more direct bearing upon electronic computers than the other proposed configurations in sections 2.3 through 2.5. Extensive mathematical research has shown that the previous configurations are equivalent to the Turing configuration. The reason that the Turing configuration for algorithms is the preferred one, results from the fact that Turing directly related the realization of algorithms to mechanical devices which he called machines or Turing machines [Trakhtenbrot (1966)].

Each Turing machine will consist of the following parts:

1) There is an external alphabet consisting of a finite set of symbols $s_1, s_2, \ldots, s_k$ which will be used as inputs to the Turing machine. The letter s_1 will denote the empty (Λ) letter. Memory of the Turing machine is an infinitely long tape which will store entries in consecutive cells. Each cell will contain at most one symbol. The tape can be thought of as finite with an additional empty cell added to either end of the tape if an end of tape condition is encountered. When this information is being processed, the response of the machine will be some predetermined action and this will carry out the computation automatically.

When the Turing machine is processing information it can read, at most, one cell in each cycle, i.e., it can "scan" at most one cell at a time. In order to scan one cell at a time the symbols R and L, standing for right and left, respectively, will be needed as commands for the machine to move the tape. Movement will be limited to one cell

at a time, either right or left. If the symbol s appears it means scan the same cell again.

2) For processing information in memory, a logical unit $\mathcal{L}$ is needed. The unit $\mathcal{L}$ can have at most a finite number of states or configurations denoted by $q_1, q_2, \ldots, q_m$. The unit has two input lines and three output lines. The input lines consist of the present state of the unit q_ℓ and the symbol s_i currently being scanned. The outputs consist of the triple s_j, P, q_i where s_j replaces the present symbol (s_i) being scanned, P is the positional symbol which designates the motion of the tape, and q_i denotes the next state of the unit.

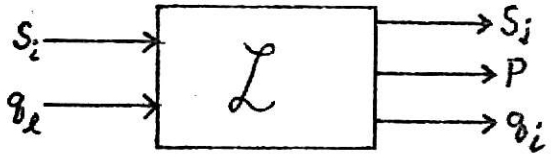


Fig. 1 Logic Unit $\mathcal{L}$ of a Turing Machine

The logical function of each unique input symbol and each initial state can be represented by a "functional matrix" [Trakhtenbrot (1966)] where the symbols represent the rows of the matrix, the states represent the columns, and some triple represents the contents of cell determined by the inputs to $\mathcal{L}$. Figure 2 illustrates the basic components of all Turing machines.

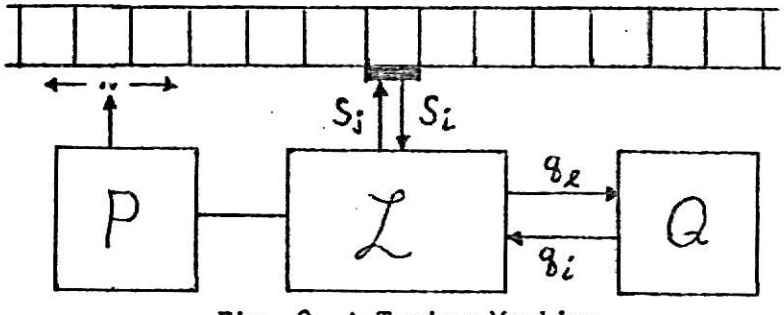


Fig. 2 A Turing Machine

Turing machines can also be thought of as consisting of "quadruples" [Davis (1958)] consisting of the following forms:

1) $q_i \ s_j \ s_k \ q_\ell$

2) $q_i \ s_j \ R \ q_\ell$

3) $q_i \ s_j \ L \ q_\ell$

4) $q_i \ s_j \ q_k \ q_\ell$

The first two symbols in each quadruple are called its internal configuration and alphabet, respectively. The Turing machine scans the tape, assuming that it has been given an initial symbol and some initial internal state, until a match is made with the first two symbols and then action is taken by the machine to change the conditions of the machine to those specified by the second pair of symbols in the quadruple. Note that Davis's representation is even more simplified than is Trakhtenbrot's.

A Turing machine can now be defined as a finite set of quadruples where no two quadruples contain the same first two symbols. The Turing machine, denoted as Z, is simple if Z contains no type 4 quadruples. Type 4 will be discussed later.

An instantaneous description of Turing machine Z is an expression such that there is exactly one q_i which is not a rightmost symbol and not an R or L symbol. If α and β are instantaneous expressions in Z then $\alpha \longrightarrow \beta$ (Z) if

1) There exist expressions P and Q, and Z contains $q_i \ s_j \ s_k \ q_\ell$ such that $\alpha = P \ q_i \ s_j \ Q$,
$\beta = P \ q_\ell \ s_k \ Q$

or 2) There exist expressions P and Q, and Z contains $q_i \ s_j \ R \ q_\ell$ such that $\alpha = P \ q_i \ s_j \ s_k \ Q$,
$\beta = P \ s_j \ q_\ell \ s_k \ Q$

or 3) There exists an expression P and Z contains $q_i \ s_j \ R \ q_\ell$ such that $\alpha = P \ q_i \ s_j$,
$\beta = P \ s_j \ q_\ell \ s_l$

or 4) There exists expressions P and Q and Z contains $q_i\ s_j\ L\ q_\ell$ such that

$$\alpha = P\ s_k\ q_i\ s_j\ Q,$$

$$\beta = P\ q_\ell\ s_k\ s_j\ Q$$

or 5) There exists an expression Q and Z contains $q_i\ s_j\ L\ q_\ell$ such that

$$\alpha = q_i\ s_j\ Q,$$

$$\beta = q_\ell\ s_1\ s_j\ Q$$

An instantaneous expression is terminal if has no such that $\alpha \longrightarrow \beta\ (Z)$.

Turing computation is now defined as a finite sequence $\alpha_1, \alpha_2, \ldots, \alpha_p$ of instantaneous descriptions such that $\alpha_i \longrightarrow \alpha_{i+1}\ (Z)$ for $1 \leq i < p$. The instantaneous description α_p is called terminal with respect to Z. Thus, α_p is the resultant of α_1 on Turing machine Z denoted as $\alpha_p = R\ E\ S_Z(\alpha_1)$.

Any number can be represented as a tape expression $\bar{x}$ where $\bar{x} = 1^{n+1}$ (n+1 successive 1's). Each k-tuple $(x_1, x_2, \ldots, x_k)$ of integers associated as a tape expression is represented as $\overline{(x_1, x_2, \ldots, x_k)} = \bar{x}_1 \Lambda \bar{x}_2 \Lambda \ldots \Lambda \bar{x}_k$. Thus (1, 2, 0) = 11Λ111Λ1.

Associated with each Z there is an n-ary function $\psi_Z^{(n)}(x_1, x_2, \ldots, x_n)$ such that for each n-tuple $(m_1, m_2, \ldots, m_n)$ the instantaneous description $\alpha_1 = q_1\ (m_1, m_2, \ldots, m_n)$ and either,

1) There exists a computation of Z, $\alpha_1, \alpha_2, \ldots, \alpha_p$. Thus $\psi_Z^{(n)}(m_1, m_2, \ldots, m_n)$ = the number of occurrences of 1's in α_p or the number of occurrences of 1's in $R\ ES_Z(\alpha_1)$.

or 2) There exists no computation $\alpha_1, \alpha_2, \ldots, \alpha_p$ and thus $\psi_Z^{(n)}(m_1, m_2, \ldots, m_n)$ is left undefined.

Partial computability of an n-ary function is $f(x_1, x_2, \ldots, x_n)$ on Z if $f(x_1, x_2, \ldots, x_n) = \psi_Z^{(n)}(x_1, x_2, \ldots, x_n)$. If f computes outputs for all valid arguments then it is said to be computable.

3.2 The Realization of Algorithms on Turing Machines

It can be shown that any function which is algorithmically defined can be solved by means of a Turing machine. A few examples will help illustrate this fact along with how Turing machines operate.

Perhaps one of the easiest algorithms to realize by means of Z is addition. Z will thus compute $f(x_1, x_2) = x_1 + x_2$ such that $\psi_Z^{(2)} = x_1 + x_2$. The quadruples consist of

$$q_1\ 1\ \Lambda\ q_1$$
$$q_1\ \Lambda\ R\ q_2$$
$$q_2\ 1\ R\ q_2$$
$$q_2\ \Lambda\ R\ q_3$$
$$q_3\ 1\ \Lambda\ q_3$$

If the instantaneous description $\alpha_1 = q_1(\overline{a_1, a_2}) = q_1\ \bar{a}_1\ \Lambda\ \bar{a}_2$ then computation proceeds as

$$\alpha_1 = q_1\ 1^{a_1+1}\ \Lambda\ 1^{a_2+1}$$
$$\rightarrow q_1\ \Lambda\ 1^{a_1}\ \Lambda\ 1^{a_2+1}$$
$$\rightarrow \Lambda\ q_2\ 1^{a_1}\ \Lambda\ 1^{a_2+1}$$
$$\rightarrow \cdots\cdots\cdots \text{ (it stays at } q_2 \text{ and scans all 1's)}$$
$$\rightarrow \Lambda\ 1^{a_1}\ q_2\ \Lambda\ 1^{a_2+1}$$
$$\rightarrow \Lambda\ 1^{a_1}\ \Lambda\ q_3\ 1^{a_2+1}$$
$$\rightarrow \Lambda\ 1^{a_1}\ \Lambda\ q_3\ \Lambda\ 1^{a_2}$$

which is terminal. Therefore

$\psi_Z^{(2)}(a_1, a_2)$ = the number of occurrences of $RES_Z(\alpha_1) = a_1 + a_2$

Subtraction is handled in a similar manner by erasing successive 1's in a_1 and a_2.

The successor function developed earlier is also computable. If Z is terminal with respect to $q_1\ \bar{m}$ and Z consists of only one quadruple $q_1\ \Lambda\ \Lambda q_1$ then $\psi_z^{(1)}(m)$ = the number of 1's of $q_1\ \bar{m} = m+1$.

The identity function established earlier is realizable as $I(x) = x$. If Z has a single quadruple $q_1\ 1\ \Lambda\ q_1$ then $q_1\ \overline{n} = q_1\ 1^{n+1} \longrightarrow q_1\ \Lambda\ 1^n$ which is now terminal. Therefore

$\psi_Z^{(1)}(n)$ = the number of occurrences of 1's in $q_1\ \Lambda\ 1^n = n$

This notion can be expanded to identities with n-tuples as function arguments. For example

$$U_i^n(x_1, x_2, \ldots, x_n) = x_i \text{ for } 1 \leq i \leq n$$

has a Turing machine Z such that

$$\psi_Z^{(n)}(x_1, x_2, \ldots, x_n) = x_i$$

The Turing machine Z has the following quadruples:

$q_j\ 1\ \Lambda\ q_{2n+j}$

$q_j\ \Lambda\ R\ q_{j+1}$

$q_{2n+j}\ \Lambda\ R\ q_j$

$q_i\ 1\ \Lambda\ q_i$

$q_i\ \Lambda\ R\ q_{2n+i}$

$q_{2n+i}\ 1\ R\ q_{2n+i}$

$q_{2n+i}\ \Lambda\ R\ q_{i+1}$

then a function $\psi_Z^{(n)}(\overline{a_1, a_2, \ldots, a_n}) = q_1\ 1^{a_1+1}\ \Lambda\ 1^{a_2+1}\ \Lambda \ldots \Lambda 1^{a_n+1}$

$\longrightarrow \ldots\ldots$

$\longrightarrow \Lambda^{a_1+1}\ \Lambda\ q_2\ 1^{a_2+1}\ \Lambda \ldots \Lambda\ 1^{a_i+1}\ \Lambda \ldots \Lambda_1^{a_n+1}$

$\longrightarrow \ldots\ldots$

$\longrightarrow \Lambda^{a_1+1}\ \Lambda\ \Lambda^{a_2+1}\ \Lambda \ldots \Lambda\ q_i\ 1^{a_i+1}\ \Lambda \ldots \Lambda_1^{a_n+1}$

$\longrightarrow \Lambda^{a_1+1}\ \Lambda\ \Lambda^{a_2+1}\ \Lambda \ldots \Lambda\ q_i\ 1^{a}\ \Lambda \ldots \Lambda_1^{a+1}$

$\longrightarrow \Lambda^{a_1+1}\ \Lambda\ \Lambda^{a_2+1}\ \Lambda \ldots \Lambda\ q_{2n+i}\ 1^{a}\ \Lambda \ldots \Lambda_1^{a+1}$

$\longrightarrow \ldots\ldots$

$\longrightarrow \ldots \Lambda\ 1^{a}\ \Lambda \ldots q_{n+1}\ \Lambda$ which is terminal

Thus $\psi_Z^{(n)}(a_1, a_2, \ldots, a_n) = a_i$

Multiplication is handled through successive additions using some auxiliary storage as counters.

Until now type 4 quadruple has not appeared. If the quadruple $q_i\ s_j\ q_k\ q_\ell$ appears, then effectively the Turing machine Z is thought of as asking if the number $n \in A$ where A is a fixed set of integers. If α and β are instantaneous descriptions then $\alpha \xrightarrow[A]{} \beta(Z)$ if there exists expressions P and Q where $\alpha \equiv Pq_i\ s_j\ Q$ and Z contains $q_i\ s_j\ q_k\ q_\ell$ such that either

1) The number of 1's of $\alpha \in A$ and β is $Pq_k\ s_j\ Q$ or

2) The number of 1's of $\alpha \notin A$ and β is $Pq_\ell\ s_j\ Q$

Thus, when Z is at an instantaneous description $Pq_i\ s_j\ Q$ and $q_i\ s_j\ q_k\ q_\ell$ is the quadruple of the machine, it is asking the "external world" whether n is the number of occurrences of 1's on the tape at the present time. If yes, change the state of Z to q_k, if no, change the state to q_ℓ. The instantaneous description α is final with respect to Z if there is no quadruple whose pair begins with $q_i\ s_j$. The search for the number of occurrences of 1's will be limited to a certain range between two special tape markers, thus making the search finite.

Computations of these sorts are virtually unlimited. Composition of quadruples will effectively compute all algorithms. The next section of Universal Turing machines illustrates this fact.

3.3 The Universal Turing Machine

Trakhtenbrot (1966) suggests an imitation algorithm where the Turing machine will mimic any other Turing machine, thus becoming Universal. Imitation is done through the functional matrix described earlier.

The algorithm proceeds as

1) Scan the cell being considered by the present state of $\mathcal{S}$.
2) Find the column in the functional matrix for this state.
3) Find the row in the functional matrix corresponding to the symbol in the cell.
4) Replace the symbol in the cell by the first symbol in the triple unique to this row and column.
5) If the second symbol in the triple is terminal then stop.
6) If the second in the triple is S replace the state of by the third symbol in the triple.
7) If the second symbol in the triple is L then change states to the third symbol in the triple and move the tape one cell left.
8) If the second symbol in the triple is R then change states to the third symbol in the triple and move the tape one cell right.
9) Proceed to step 1.

In this manner all algorithms are solvable on the Universal Turing machine.

Chapter IV

Algorithmically Unsolvable Problems

4.1 The Halting Problem

The question arises, is there any procedure such that, for any given integers x and y, one can determine if there is a function which when given x as inputs will yield y as outputs? More clearly, can a set of productions be defined which when applied to x will generate y as an output? Through a series of proofs beyond the scope of this paper it can be shown that there is no such procedure which can do this. Essentially there is no algorithm which when given another algorithm as input, will be able to recognize if that algorithm will terminate after a finite number of steps.

4.2 The General Word Problem

The above example has shown how certain recursive problems were unsolvable. There are many other types of unsolvable problems. One such type is the word problem of Trakhtenbrot (1966).

If word R can be transformed into S by any admissible substitution rule then R is said to be adjacent to S and vica versa. A sequence of successively adjacent words R_1, R_2, ..., R_n will form what is called a deductive chain. If there is a deductive chain from R to S then R and S are said to be equivalent. It can be proved that if P and Q are expressions which are equivalent then if P occurs in R an application of Q for P in R yields a word which is equivalent to R. However, is there an algorithm which will determine if any two given words are equivalent?

In order to solve the equivalence problem it will be necessary to define the concept of active cells in any Turing machine Z. The cells will be made up of the scanned cell, all non-empty cells and all cells between the first two.

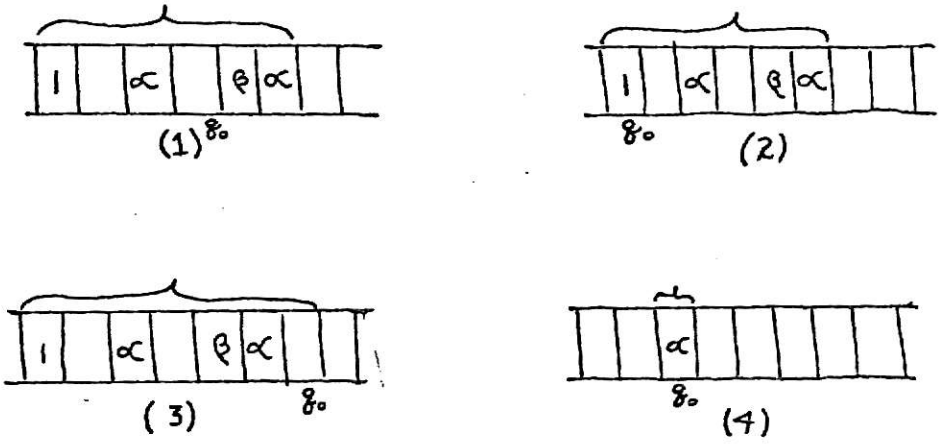


Fig. 3 Active Cells

The scanned cell in Figure 3 is denoted as one which has state q_0 under it. It is interior if the cell is not the leftmost or rightmost cell. Cell 1 is interior, cell 2 is left, cell 3 is right, and cell 4 is a unit configuration. Let h be a symbol outside of the alphabet of Z which bounds the tape of Z. Each configuration is of the form h R h. Call these bounded configurations C - words.

Define a calculus π_Z for Z as follows:

1) The alphabet of π_Z is h and the alphabet of Z. All C -words $\subseteq$ π_Z but not the converse.

2) Substitutions in π_Z are constructed so as to guarantee all legal C - words. The substitution of s' q' for sq ($sq \longrightarrow s'q'$) in π_Z where the tape of Z does not move implies that the active portion of cells do not change and thus there is no guarantee that there is a single substitution which can be used to establish equivalence. This implies that the general word problem is algorithmically unsolvable.

4.3 Other Areas of Mathematics Where Unsolvability Appears

Unsolvability appears in most branches of mathematics. Rogers (1967) suggests that there exists problems in such areas as groups, topology, polynomial solving, and manifold problems. Mendelson (1964) demonstrates Tarshi's unsolvable problem in formal number theory. Davis (1958) suggests such unsolvable decision problems as semi-computable predicates and recursively enumerable sets. The list continues to grow as rapidly as the uses for computers and algorithmic solutions grow.

Chapter V

Summary

It was shown that algorithms could be expressed as recursive or partial recursive functions. In order to define either effective computation or unsolvability, it is necessary to give a precise definition and characterization of an algorithm. Markov, Kleene, Herbrand and Gödel, Turing, and many others have produced such characterizations. However, all of these characterizations can be shown to be equivalent.

Turing's characterization is used frequently by computer scientists because it directly relates the process of computation to a mechanical process. All computations can be realized on a Turing machine by use of the imitation algorithm to yield a Universal Turing machine.

It is critical for computer scientists to be aware of the various characterizations of algorithms since it is an underlying foundation upon which modern day computers are predicated.

Acknowlegdments

The author wishes to thank Dr. P. S. Fisher for his time and assistance in making the writing of this paper possible.

Index of Figures

BIBLIOGRAPHY

Davis, Martin (1958). Computability and Unsolvability. New York, McGraw-Hill.

Feigenbaum, Edward and Julian Feldman (1963). Computers and Thought. New York, McGraw-Hill.

Kleene, S. C. (1962). Introduction to Metamathematics. Princeton, N. J. Van Nostrand.

Mendelson, Elliott (1964). Introduction to Mathematical Logic. Princeton, N. J., Van Nostrand.

Rogers, Hartley, Jr. (1967). Theory of Recursive Functions and Effective Computability. New York, McGraw-Hill.

Trakhtenbrot, B. A. (1966). Algorithms and Automatic Computing Machines. Boston, Mass., Heath.

COMPUTATION AND UNSOLVABILITY

by

Steven Kent Thornbrugh

B. Sc., Kansas State University, 1968

AN ABSTRACT OF A MASTER'S REPORT

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Statistics and Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1969

The purpose of this paper was to survey the theory of computation and some aspects of unsolvability. Algorithmic characterizations of several of the leading specialists in this area were presented. All terms which were described in mathematical notation presented at the start of the paper are self contained within the paper. The paper deals largely with recursive functions.

It was established later in the paper that all of the previous characterizations were equivalent to Turing's characterization of an algorithm. Consequently, Turing's characterization is discussed in considerable detail. These algorithmic characterizations lay the ground work for the discussion of algorithmic unsolvability. Several examples are given to illustrate what some of the programming limitations of modern computers are.