Research Inventy: International Journal of Engineering And Science
Vol.4, Issue 4 (April 2014), PP 53-60
Issn (e): 2278-4721, Issn (p):2319-6483, www.researchinventy.com

Turing Machine and the Conceptual Problems of Computational Theory

Edward E. Ogheneovo
Department of Computer Science, University of Port Harcourt, Nigeria.

ABSTRACT: *As computer hardware and software continue to develop at an ever increasing rate, one will be forced to believe that no problem is too hard for a computer to solve. Given enough memory, time, and ingenuity on the part of the programmer, one will think that there is no problem too difficult for the computer to solve. Yet there are many problems inherently unsolvable by a computer. There are problems for which, if a program were to exist, whether or not there was a machine big enough and fast enough to actually perform it, a logical contradiction would result. In this paper, we discuss Turing machines and the conceptual problems of computational Theory. The paper argues that there are some set of problems that cannot be computed by Turing machine and these set of problems are called uncomputable sets and functions. Examples of such sets and functions were provided. The paper also discuss how we can simulate one Turing machine to another Turing machine which of course can act as a universal Turing machine that can be used to solve all computable problems. A proof of the theorem was proposed.*

KEYWORDS: *Turing machine, computation, computational theory, computability, computer simulation*

I. INTRODUCTION

Computers differ from each other in terms of their hardware and software. As a result, there is need to construct a standard computation theory that will apply to all standard computers. There are several abstract models of computer devices: non-deterministic finite automata (NFA), deterministic finite automata (DFA), non-deterministic finite automata with ε–transition, pushdown automata (PDA), and deterministic pushdown automata (DPDA). However, none of these models of computing devices is as useful as real computer. Thus we need to consider the theoretical model for a computer that will be equivalent to all other standard computers. This standard theoretical model is referred to as the Turing machine. Turing machines are simple, abstract computational devices intended to help investigate the extent and limitations of what can be computed [4], [9]. As a computer hardware and software continue to develop at an ever increasing rate, one will be forced to believe that no problem is too hard for a computer to solve. Given enough memory, time, and ingenuity on the part of the programmer, one will think that there is no problem too difficult for the computer to solve. Yet there are many problems inherently unsolvable by a computer. There are problems for which, if a program were to exist, whether or not there was a machine big enough and fast enough to actually perform it, a logical contradiction would result.

However, computability is also relevant to the more logically fundamental parts of mathematics. As an example, consider the concept of real number in mathematics. Most of them are irrational and as a result cannot be defined by writing out their decimal expansion. However, for numbers like $\sqrt{2}$ and π, we could write a computer program which, if left to run, would run forever and would print out all their digits [8]. Unless we can write a program to do this and a computer to solve such a problem, otherwise they will forever remain unsolvable or uncomputable [5]. However, with uncountably many real numbers and only countably many potential computer programs, most real numbers are inaccessible to human thought. More so, there are some real numbers that can be precisely defined, yet no computer program can be devised to print out all their digits. As another example, we can think of a BASIC program in which the GOTO statement is used [15]. The effect of this command is that the program will run forever as it will cause an infinite loop. For this reason, the use of GOTO statement has been discouraged and is no longer used in programming languages.

This is exactly what Turing was intended in. He was interested in the question of what it means for a task or problem to be computable, which is one of the foundational questions in the philosophy of computer science. Therefore, a task is computable if it is possible to specify a sequence of instructions which will result in the completion of the task when they are carried out by some machine. The set of instructions written for solving such problems or task is referred to as effective procedure or algorithm. It must be noted that an effective procedure may depend on the capabilities of the machine used to carry out the instructions. Therefore, devices with different capabilities may be able to complete different instruction sets, and may thus result in different classes of computable tasks.

Thus the standard theoretical model used in computer science for all computers is the Turing machine. The Turing machine was first proposed by Alan Turing in 1936. A Turing machine [16] is an imaginary machine that interacts with the outside world by reading an infinite tape. The tape is divided into cells on which symbols or blanks are written, usually one symbol at a time, and writing its output on the same tape as the reading continues. The machine has a head that executes the reading and writing. The head is positioned on a specific cell of the tape such that it can move left (L) or right (R) one cell at a time. Thus in Turing machine, the computation is done through a transition function that tells the machine how to react to the symbols on the tape.

Turing proposed a class of machines that are referred to as Turing machines. These machines lead to a formal notion of computation called Turing Computable [17]. A task is Turing computable if it can be carried out by some Turing machine. Turing machines are not physical machines objects but mathematical ones. According to Turing, it is not necessary to talk about how the machine carries out its actions, but to believe that the machine can carry out the specified actions, and that those actions may be uniquely described.

II. DESCRIBING A TURING MACHINE

A definition of computation is needed to study computation mathematically. A Turing machine is 'a general-purpose' computer with an infinite tape. It consists of

- The control unit which help to read the current tape symbol
- Writes a symbol on the tape
- Moves one position to the left or right
- Switches to the next state

Thus each transition o the machine is a 4-tuple:

$$(State_{current}, Symbol, State_{next}, Action)$$

which is read as "if the machine is in current state ($State_{current}$) and the cell being scanned contains Symbol then move into next state ($State_{next}$) taking Action." In taking an action, a Turing machine may either write a symbol on the tape in the current cell or move the head cell to the left or right. However, if the machine reaches a situation where there is no unique transition rule to be carried out, then the machine is said to halt.

Thus a Turing machine is an infinite tape positioned in horizontal form stretching from left to right and divided into various cells, each cell holding an item or symbol "0" or "1". In this text, we denote a blank by "0" and a symbol (non-blank) by "1". The machine has a read/write head used for scanning the tape a single cell at a time either to the left or to the right. Thus the read/write head can move left and right along the tape to scan successive cells. In general, the action of a Turing machine is determined by:

- The current state of the machine
- The symbol in the cell currently being scanned by the head
- A table of transition rules, which serve as the program for the machine

Therefore, a Turing machine can be thought of as a finite state machine sitting on an infinitely long tape containing symbols from some finite alphabet, $\sum$. Therefore, we can say that the tape is infinite in length and that the memory of the machine is infinite. Also, from the definition of this machine, we can say that a function is said to be Turing-computable if there exists a set of instructions that will result in a Turing machine computing the function irrespective of the amount of time it takes. Thus it can be said that given an infinite time, the machine will complete the computation.

These two assumptions are intended to ensure that the definition of computation is not too narrow. This ensures that no computable function will fail to be Turing-computable solely because there is inefficient time or memory to complete the computation [11]. Thus there may be some Turing-computable functions which may not be carried out by any existing computer, perhaps because no existing machine has sufficient memory to carry out the task. However, some Turing-computable functions may never be computable in practice since they may require more memory than can be built using all of the (finite number of) atoms in the universe. On the other hand, a result that shows that a function is not Turing-computable is very strong, since it implies that no computer that we could ever build could carry out the computation [10].

2.1 Formal Definition of Turing Machines

Formally, we define a Turing machine as a 7-tuple $(Q, q_0, \Gamma, b, \sum, F, \sigma)$ where

- Q is a finite set of states with one of them $q_0 \in Q$ being a designated starting state (this is the state the machine starts its operation in).
- Γ is a finite set of symbols with one of them $b \in \Gamma$ being a designated starting state (this is the state the machine starts its operation in).
- $\sum \subset \Gamma$ is a subset of input symbols
- $P \subset Q$ is a subset of accepting states which finalizes the computation (when the machine reaches F, the computation final state).
- σ: $Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}$ is a partial transition function (if the machine reaches a state and input that are not defined for σ, the machine halts). In this transition function, $q_0 \in Q$ is the start state, $q_{accept} \in Q$ is the accept state, $q_{reject} \in Q$ is the reject state, where $q_{accept \neq q}$reject

From the foregoing, it can be seen that the heart of a Turing machine is its transition function σ since it tells us how the machine gets on from one configuration to another. A Turing machine configuration is often described by its current state, the current tape contents, and the current head location.

The machine at state $Q \in q$, reads the current symbol $\gamma \in \Gamma$ on the tape loading to $\sigma(q, \gamma) = (q^1, \gamma^1, d)$ where

$q^1 \in Q$ is the next state

$\gamma^1 \in \Gamma$ is the output symbol being written by the head on the tape.

$d \in \{L, R\}$ is the movement of the head (left or right) on the infinite tape

As an example, consider a Turing machine that is supposed to process a binary tape to the right of the starting position and halt execution when the number of one's reaches 10. In this case, we have

$Q = \{q_0, q_1, \ldots, q_{10}\}$, q_0 is the starting state,

$F = \{q_{10}\}$, $\Gamma = \{0, 1, b\}$, $\Sigma = \{0, 1\}$ and σ defined as follows:

$\sigma(q_i, 0) = (q_i, b, R)$ and

$\sigma(q_i, 1) = (q_{i+1}, b, R)$.

The machine reads the input left to right. It does not modify the tape since it writes blanks) and halts computation after reading 10 1-symbols. At this point it transits to the final state, q_{10}.

From the discussion above, it can be seen that the tape serves three main purposes: (1) input, (2) output, (3) memory, while the state of the machine $q \in Q$ is a different form of memory –the control. The control, or state, is hard-wired into the machine in a similar way to hardware. The tape, on the other hand, may change from one operation of the machine to another. In other words, the control unit defines what the machine does like the program code of a function. However, while the tape is infinite, Q is finite. This means that the Turing machine corresponds to a finite program –a program that has at its disposal an infinite ream of scratch paper.

The example above shows how difficult it is do design a Turing machine that will perform a specific task. This is why programming languages, which are a much more convenient way to design a computation, are used instead of the Turing machine. However, to analyze the theoretical properties of computers, the Turing machine is more convenient to use. It has the properties of being simple and does not change as computers do when changes are made to the software or hardware.

If a Turing machine can accomplish a task, the task is called computable. If a Turing machine cannot accomplish a task, the task is said to be non-computable or unsolvable. The study of these classes of problems is the topic of the theory of computability. The time it takes a Turing machine to finish its computation is called its time complexity. The amount of tape it uses is called the space complexity. The study space and time complexity of different problems is the topic of the theory of complexity.

III. COMPUTABLE AND UNCOMPUTABLE SETS AND FUNCTIONS

In computational theory, there are problems that the Turing machine and by extension the modern computer can solve and there are problems that are unsolvable. Those problems that can be solved are called computable problems while those that defy solution are called uncomputable problems. The study of these problems is collectively referred to as computational theory [1].

3.1 Computable Sets and Functions

These are functions that can be computed with Turing machines and other computational function has an algorithm (procedure) that is explicitly and have unambiguous instructions on how to compute it. Such a procedure must also be encoded in the finite alphabet used by the computational model.

3.2 Uncomputable Functions

In this section, we discuss functions that cannot be computed. Thus functions having real numbers are uncomputable. This is so because real numbers are uncountable. Also, the set of finitary functions on the natural numbers is uncountable and most of them are therefore uncomputable [12], [14]. We discuss the real numbers and the set of natural numbers such as the halting problem and the Entscheidungsproblem problem.

3.2.1 Real Numbers

A real number is a number that has continuous value. A real number is computable if it can be approximated by some computable function in such that given any integer n $\geq$ 1, the function produces an integer *k* such that:

$$\frac{k-1}{n} \leq a \leq \frac{k+1}{n}$$

The real numbers ($\mathbb{R}$) include all the rational numbers include all the rational numbers, such as integer -2 and the fraction $5/4$, and all the irrational numbers such as $\sqrt{2}$ and π. A real number can be determined by a possibly infinite decimal representation such as 1.41421356… (for $\sqrt{2}$) and 3.14159265… (for π). The real numbers are uncountable and so must real numbers are not computable. A computable number is a real number that can be computed to within a desired precision by a finite, terminating algorithm. Most real numbers have infinite values and as such they are said to be uncomputable. Examples include the busy beaver and the kolmogorvo complexity or any function that outputs the digits of a noncomputable number.

The busy beaver function, $\Sigma : N \rightarrow N$, is defined such that $\Sigma(n)$ is the maximum attainable score (the maximum of 1s on a tape) among all halting 2-symbol n-state Turing machines when started on blank tape. Thus since the busy beaver function (Σ) is well-defined, therefore, for every n, there are at most finitely many n-state Turing machine with at most finitely many running times.

The kolmogorvo complexity is a measure of the computability resources needed to specify the object. The complexity of a string is the length of the shortest possible description of the string in some fixed universal description language. However, the kolmogorvo complexity of any string cannot be more than a few bytes larger than the length of the string itself.

3.2.2 The Set of Natural Numbers

Most sets and subsets of natural numbers (ℕ) are not computable. Examples include the Halting problem and the Entscheidungsproblem. Church and Turing [17] independently showed in that this set of natural numbers is not computable. According to the Church-Turing thesis, there is no effective procedure (an algorithm) that can perform these computations.

The halting problem is a decision problem about properties of computer program on a fixed Turing-complete model of computation. The halting problem states that given a description of an arbitrary computer program, decide whether the program finishes running or continue to run forever. According to Turing [17], the halting problem is undecidable over Turing machine. Turing proved that a general algorithm to solve the halting problem for all possible input pairs does not exist [2].

The Entscheidungsproblem problem was posed by David Hilbert in 1928 as the 10^{th} problem that has no solution. The Entscheidungsproblem problem asks for an algorithm that will take as input a description of a formal language and a mathematical statement in the language (or Diophantine equations) and produce as either "Yes" or "No" ("TRUE" or "FALSE") according to whether the statement is true or false. Put another way, Entscheidungsproblem asks for an algorithm to decide whether a given statement is provable from the axioms using logical rules. The Entscheidungsproblem is solved when we know a procedure that allows for any given logical expression to decide by finitely many operations its validity or satisfiability. A quite definite generally applicable prescription is required to allow one to decide in a finite number of steps the truth or falsity of a given purely logical assertion. The Entscheidungsproblem is considered as one of the main problems in mathematical logic. Church and Turing [17] provide the answer to Hilbert's problem by stating that a general solution to the Entscheidungsproblem is impossible. According to Church-Turing thesis, whenever there is an effective method (algorithm) for obtaining the values of a mathematical function, the function can be computed by a Turing machine.

3.3 The Limit of Computation

Computer science was born knowing its limitations. The strength of the universal machine leads directly to a negative consequence of uncomputability [10]. By computability we mean certain natural computational problems cannot be solved by the Turing machines or, by extension computer programs. This leap from the notion of universality to impossibility is rooted in two basic issues: 1) the difficulty in determining the "ultimate" behavior of a program; and 2) the self-referential character of the universal Turing machine T_u [6], [7]. In the first case, recall that our universal machine T_u simply performed a step-by-step simulation of a Turing machine T_m on an input *n*. This means that if T_m computes forever, without halting, then T_m's simulation will run forever as well. This phenomenon is often referred to as "infinite loop." A situation where by a machine or program run forever without producing desired result—a program keeps running without producing any result or output [13], [3].

However, to solve the problem of infinite loop where a machine runs forever without halting, it is necessary to introduce a device that can help detect whether a machine can halt by producing an out or it will run forever; and to also determine what can be done in case where the program runs forever instead for us to just producing a blind simulation of T_u which cannot detect these signs. Thus the machine used for this process is referred to as "Universal Terminator Detector" — a Turing machine *D* that behaves as follows: Given a description of a Turing machine T_m and an input a to T_m, the machine *D* performs a finite number of steps, and then correctly reports whether or not T_m will ever halt with a result when it is run on *n* [12].

IV. INITIALIZING A TURING MACHINE FOR COMPUTATION

The pseudocode described in algorithm 1 uses variables q, *i*, σ_1 ,… σ_k, σ^1_1, …, σ^1_k, d_1, …, d_k to initialize the variables in our Turing machine. Each of these variables can assume only finitely many values. Let K, Σ denote the state set and alphabet of *M*. Also, let [k] = [1, …, k] and L denote the set of line numbers of the pseudocode in the algorithm. Then the k^a is defined as:

$$K^a = K \times [k] \times \Sigma^k \times \Sigma^k \times \{\leftarrow, \rightarrow, -\})^k \times L$$

Therefore, a state of M^a determines the values of the variables q, *i*, σ_1 ,… σ_k, σ^1_1, …, σ^1_k, d_1, …, d_k as well as the line number of the line of pseudocode that M^a is currently working on.

As the algorithm shows, the initialization process starts by copying the transition function of the Turing machine *M* from the input tape to the description tape. The tape header then moves right until after the second comma on the input tape. Then a **while** loop is introduced to ensure that the input symbols have not been exhausted, since the semicolon ";' is not reached. This is followed by a **for** loop is set to indicate the number of tapes. Finally, a **repeat … until** loop set to ensure that all the input alphabets are read until the input tape symbols are exhausted.

Algorithm 1: Initializing a Turing machine

```
// Copy the transition function of M from the input tape to the description tape.
Move right past the first and second commas on the input tape.
while not reading ';' on input tape do
    Read input symbol, copy to description tape, move right on both tapes
end while // end of while loop
// Write the identifiers of the halting states and the direction of which the input symbols move
Move to start of input
Using binary addition subroutine, write m + n in binary on working tape
for i = 0, 1, 2, 3, 4, 5 do
    On special tape, write binary representation of m + n + I, followed by ';'
    // The value of m + n is stored in the working tape and not memory
end for
// Copy the input string x unto the working tape, ℓ, and separate the with commas
Write m + n + 6 in binary on the state tape,  // to store the value of ℓ
Starting from the left edge of input tape, move right until ';' is reached
Move to left edge of working tape and state tape
Move one step right on state tape
repeat
    while state tape symbol is not ⊔
        Move right on input tape, working tape, and state tape
        Copy symbol from input tape to working tape
    end while // while loop ends
    Write ';' on working tape
    Rewind to left edge of state tape, then move one step right
until input tape symbol is ⊔
Copy m from input tape to state tape
// end initialization
```

4.1 Simulating One Turing Machine to Another Turing Machine

A multi-tape Turing machine k tapes is defined almost the same way as conventional single-tape Turing machine, except that a multi-tape Turing machine stores k strings simultaneously. It does this by maintaining a position in each of the k strings, updates these k positions independently (i.e., it can move left in one string while moving right in another string), and its state changes are based on the entire k-tuple of symbols that it reads at any point in time. More formally, a k-tape Turing machine has a finite alphabet Σ and state set K as before, but its transition function is:

$$\delta : K \times \Sigma^k \rightarrow (K \cup \{\text{halt, yes, no}\}) \times (\Sigma \times \{\leftarrow, \rightarrow, -\})^k ,$$

The difference being that the transition is determined by the entire k-tuple of symbols that it is reading, and that the instruction for the next action taken by the machine must include the symbol to be written, and the direction of motion, for each of the k-strings. The algorithm 2 is used to simulate the multi-tape Turing machine M using a single-tape Turing machine M^o.

Algorithm 2: Simulation a Turing machine M by Turing machine M_a

```
q ← s // Initialize state
repeat
    // Simulation round
    // First, find out what symbols M is seeing
    repeat
        Move right without changing contents of string
        for I = 1, …, k do
            if i^th flag at current location equals 1 then
                σ_I ← i^th symbol at current location
            end if
        end for
    until M_a reaches ⊔
```

13: **repeat**
14: Move left without changing contents of string
15: **until** M_a reaches ⊳
16: // Let $\sigma_1,\ldots, \sigma_k$ store the k symbols that M is seeing
17: // Evaluate state transition function for machine M
18: $q^1, (\sigma^1_1, d_1), (\sigma^1_2, d_2), \ldots, (\sigma^1_k, d_k) \leftarrow \delta(q, \sigma_1, \ldots,\sigma_k)$
19: **for** $i = 1, \ldots, k$ **do**
20: // Phase i
21: **repeat**
22: Move right without changing the contents of the string
23: **until** a location whose i^{th} flag is 1 is reached
24: Change the i^{th} symbol at this location to σ^1_1
25: **if** $d_i = \leftarrow$ **then**
26: Change i^{th} flag at to 0 at this location
27: Move one step left
28: Change i^{th} flag to 1
29: **else if** $d_i = \rightarrow$ **then**
30: Change i^{th} to 0 at this location
31: Move one step right
32: Change i^{th} flag to 1
33: **end if**
34: **repeat**
35: Move left without changing the contents of the string
36: **until** the symbol ⊳ is reached
37: **end for**
38: $q \leftarrow q^1$
39: **until** $q \in$ {halt, yes, no}
40: // If the simulation reaches this line, $q \in$ {halt, yes, no}
41: **if** q = yes **then**
42: Transition to "yes" state
43: **else if** q = no **then**
44: Transition to "no" state
45: **else**
46: Transition to "halt" state
47: **end if**

4.2 Theorem: Any multi-tape Turing machine M can be simulated with a single-tape Turing machine M_a.

Proof: Simulating a Turing machine *M* by Turing machine M_a

Using algorithm 2, we then show that any multi-tape Turing machine M can be simulated with a single-tape Turing machine M_a. this is done by showing the content of the *k*-strings of the two machines simultaneously, as well as the position of M within each of these strings. The single tape machine M^s will have alphabet $\Sigma^s = \Sigma^k$ x $\{0, 1\}^k$. the k-tuple of symbols $(\sigma^1, \ldots,\sigma_2) \in \Sigma^k$ at the i^{th} location on the tape is interpreted as the symbols at the i^{th} location on each of the k tapes used by machine M. the k-tuple of the binary values $(p_1, \ldots, p_k) \in \{0, 1\}^k$ at the i^{th} location on the tape represents whether the position of machine M on each M of its k tapes is currently at location i with binary values 1 = TRUE and 0 = FALSE. However, the alphabet must have two special symbols, ⊔. For alphabet Σ^s, the special symbol ω is identical to $(\omega)^k$ x $(0)^k$ and the special symbol ⊔ is identical to $(⊔)^k$ x $(0)^k$.

Machine M^s runs in a sequence of simulation rounds, each divided into k phases. The purpose of each round is to simulate one step of M, and the purpose of phase i in a round is to simulate what happens in the i^{th} string of M during that step. At the start of a phase, M^s is always at the left edge of its tape, on the symbol ω. It moves right or left as the string end is trying to locate the flag that indicates the position of M on the i^{th} tape and updating that symbol (and possibly the adjacent one) to reflect the way that M updates its i^{th} string during the corresponding step of its execution.

The pseudocode described in algorithm 1 uses variables q, i, $\sigma_1,\ldots\ \sigma_k, \sigma^1_1, \ldots, \sigma^1_k, d_1, \ldots, d_k$. Each of these can assume only finitely many values. Let K, Σ denote the state set and alphabet of M. Also, let [k] = [1, …, k] and L denote the set of line numbers of the pseudocode in the algorithm. Then the k^a is defined as:

$$K^a = K \times [k] \times \Sigma^k \times \Sigma^k \times \{\leftarrow, \rightarrow, -\})^k \times L$$

Therefore, a state of M^a determines the values of the variables q, *i*, σ_1 ,… σ_k, σ^1_1, …, σ^1_k, d_1, …, d_k as well as the line number of the line of pseudocode that M^a is currently working on.

V. CONCLUSION

Computers differ from each other in terms of their hardware and software. As a result, there is need to construct a standard computation theory that will apply to all standard computers. There are several abstract models of computer devices: non-deterministic finite automata (NFA), deterministic finite automata (DFA), non-deterministic finite automata with ε–transition, pushdown automata (PDA), and deterministic pushdown automata (DPDA). However, none of these models of computing devices is as useful as real computer. Thus we need to consider the theoretical model for a computer that will be equivalent to all other standard computers. This standard theoretical model is referred to as the Turing machine.

Turing machines are simple, abstract computational devices intended to help investigate the extent and limitations of what can be computed. As a computer hardware and software continue to develop at an ever increasing rate, one will be forced to believe that no problem is too hard for a computer to solve. Given enough memory, time, and ingenuity on the part of the programmer, one will think that there is no problem too difficult for the computer to solve. Yet there are many problems inherently unsolvable by a computer. There are problems for which, if a program were to exist, whether or not there was a machine big enough and fast enough to actually perform it, a logical contradiction would result.

This paper discusses Turing machines and the conceptual problems of computational Theory. The paper argues that there are some set of problems that cannot be computed by Turing machine and these set of problems are called uncomputable sets and functions. Examples of such sets and functions were provided. The paper also discuss how we can simulate one Turing machine to another Turing machine which of course can act as a universal Turing machine that can be used to solve all computable problems. A proof of the theorem was proposed.

REFERENCES

[1] Arora, S. and Barak B. (2009). Computational Complexity: A Modern Approach, Cambridge University Press.

[2] Bassey, P. C., Asoquo, D. E., and Akpan, I. O. (2010). Undecidability of the Halting Problem for Recursively Enumerable Sets, World Journal of Applied Science and Technology, Vol. 2, No. 1, ISSN: 2141 – 3290, pp. 41-48.

[3] Boolos, G. S. and Jeffrey, R. C. (1974). Computability and Logic, Cambridge: Cambridge University Press.

[4] Davis, M. (1982). Computability and Solvability, New York: McGraw-Hill Post, E. (1947). Recursive Unsovability of Problem of Thue, The Journal of Symbolic Logic, Vol. 12, pp. 1-11.

[5] Enderton, H. (1977). Elements of Recursion Theory. Handbook of Mathematical Logic, Edited by Barwise, North-Holland (1977), pp. 527-566.

[6] Herken, R., (ed.) (1988). The Universal Turing Machine: A half-Century Survey, New York, Oxford University Press.

[7] Jarvis, J. and Lucas, J. M. (2008). Understanding the Universal Turing Machine: An Implementation in JFLAp, Journal of ACM, Vol. 23, Issue 5, pp. 180-188.

[8] Kleene, S. C. (1936). General Recursive Function of Natural Numbers, Mathematics Annalen, Vol. 112, pp. 727-742.

[9] Kleinberg, J. (2004). Computability and Complexity. In Computer Science: Reflections fromi the Field, Academics Press.

[10] Lewis, H. R. and Papadimitrinu, C. H. (1981). Elements of the Theory of Computation, Englewood Cliffs, N. S. Prentice-Hall.

[11] Lin, S. and Radó, T. (1965). Computer Studies of Turing Machine Problems, Journal of ACM, Vol. 12, pp. 196-212.

[12] Minsky, M. (1967). Computation: Finite and Infinite Machines, Prentice-Hall, Inc., N. J., 1967.

[13] Petrzold, G. (2008). The Annotated Turing Machine: A Guided Tour through Alan Turing's Historic Paper on Computability and Turing Machines, Indianapolis, Indiana, Wiley Publisher

[14] Radó, T. (1962). On Non-Computable Numbers, with an Application to the Etscheidungsproblem. In Proceedings of London Mathematical Society, Ser. 2 , Vol. 42, pp. 230-265.

[15] Sumitha, C. H. and Geddam, K. O. (2011). Implementation of Recursive Enumerable Languages in Universal Turing Machine. Int'l journal of Computer Theory and Engineering, Vol. 3, No. 1, 1793-8201, pp. 153-157.

[16] Turing, A. M. (1936). On Computable Numbers with Application to the Etscheidungsproblem. In Proceedings of London Mathematical Society, 42, pp. 230-265; correction in 43 (1937), pp. 544-546; reprinted in (Davis, 1965, pp. 115-154).

[17] Turing, A. M. (1937). Computability and λ-Definability. The Journal of Symbolic Logic, Vol. 2, pp. 153-163.