IOSR Journal of Computer Engineering (IOSR-JCE)
e-ISSN: 2278-0661,p-ISSN: 2278-8727, Volume 22, Issue 5, Ser. I (Sep. – Oct. 2020), PP 01-07
www.iosrjournals.org

Computing Theories

Hieu D. Vu
Fort Hays State University 600 Park Street Hays, KS. 67601

Abstract
Science and technology have been improving rapidly, and it changed everything. Today, very seldom we can find a super computer as in the late 1970s. The computing, processing of data rely on a server or multiple servers depending on the need of a firm or a company. Everything became smaller, a server is just a little bigger than a personal computer (PC), and we have many different programming languages for developing software applications.
The subject, concept of Automata and Computing Theories was introduced in 1979, and in those early years, Automata, Computing Theory, and Language Theory were still listed as an area of active research, based heavily in Mathematics and largely a graduate course [1].
Keywords: *Automata, Computing theories, Formal language, Science, Technology, Computing machines, Turing machines, Mathematics.*

Date of Submission: 29-08-2020 Date of Acceptance: 14-09-2020

I. Introduction

I. 1. AUTOMATA THEORY

In theory, Automata is the study of abstract machines or more appropriately "Mathematical Machines or Mathematical Systems" and the computation problems that these computing machines can solve. These abstract machines are called automata, a Greek word means a machine that can do something (work) by itself. Automata theory is related to formal language theory, and often classified by a class of formal languages. An automaton can be represented as a finite set of formal language that may be an infinite set.

- Finite Automata: are used in text processing, compilers, and hardware design.
- Context-Free Grammars: used to define programming languages, and in AI.
- Turing machines: are simple abstract models of "real computers".

I. 2. COMPUTABILITY THEORY

Primarily, computability theory deals with all problems in which they are solvable on a computer. As an example, one of the most popularmachinein theory is the Turing machine, the problem of "halting problem" cannot be solved by this Turing machine. Much of the computing theory is built on the results of halting problems. Computability theory is closely related to recursion theory, a branch of Mathematical Logic

Halting problem is the problem of determining of an input string, or a computer program, whether the program will finish running or keep running forever (cannot exit the program). The halting problem is undecidable over Turing machine [2].

This research paper will focus in computability theory.

II. Mathematical Logic

II. 1. DEFINITION OF LOGIC

It is the analysis of methods of reasoning. In examining these methods, it is more important in the form than the content of the argument. For example the statement:

- All men are mortal. Socrates is a man. Therefore, Socrates is mortal.

The statement above can be interpreted as of the form: All A are B. S is an A, therefore S is B. In a logistic point of view, true or false of the premises and conclusions is not matter, the important point is whether the premises imply the conclusions. If the work is primarily devoted to the study of mathematical reasoning then it might be called mathematical logic [3].

II. 2. BASIC CONCEPTS OF LOGIC

A proposition is a declarative statement (sentence) that is the building blocks of logic. The value of a declarative statement is true or false but not both. Examples:

- Toronto is the capital of Canada. (The value is false)

- 5 + 5 = 10 (The value is true)

II. 2. 1. Propositional Logic

Propositional logic is another piece of logic where new statements can be built from given statements, using connectives such as:

- "not" (¬), (the negation of p is "not" p (¬p))
- "or" (∨), (p "or" q (p∨ q))
- "and" (∧) (p "and" q (p∧ q))

Example: the proposition: p∧ q → ¬r ∨ q is interpreted as "if p and q then not r or q". It sounds ambiguity, to clarify, parenthesis are required: (p∧ q) → ((¬r) ∨ q).

There are two more frequently used mathematical symbols: "true" (⊤) and "false" (⊥). In computer science context, the value of a propositional statement is true (1) or false (0) only [3].

Negation: The negation of proposition p is 'not' p, denoted by (¬p) is the statement "It is not the case of p". The truth value of 'not' p is the opposite value of p.

(p = 1, ¬p = 0) and (p = 0, ¬p = 1)

Disjunction: The disjunction of propositions p and q is the proposition "p or q", denoted by (p∨ q). The disjunction of (p∨ q) is false when both p and q are false, and true otherwise.

Conjunction: The conjunction of propositions p and q is the proposition "p and q", denoted by (p∧ q). The conjunction of (p∧ q) is true when both p and q are true, and false otherwise.

Truth table for the disjunction 'or' of two propositions

p	q	p ∨ q
T	T	T
T	F	T
F	T	T
F	F	F

Truth table for the conjunction 'and' of two propositions

p	q	p ∧ q
T	T	T
T	F	F
F	T	F
F	F	F

Exclusive or: The exclusive or of proposition p and q is the proposition "p exclusive or q", denoted by (p ⊕ q). The "exclusive or" of p and q is true when exactly one of them is true and false otherwise.

Truth table for the 'Exclusive or' of two propositions

p	q	p ⊕ q
T	T	F
T	F	T
F	T	T
F	F	F

Truth table for the conditional statement p → q

p	q	p → q
T	T	T
T	F	F
F	T	T
F	F	T

Conditional: The conditional statement p → q is the proposition "if p then q". It is false when p is true and q is false, and is true otherwise. In the condition statement p → q, p is called the hypothesis (or antecedent or premise), and q is called the conclusion (or consequence).

Example: Let p is the statement "Tom learns discrete math"

and q is the statement "Tom will find a good job"

The following statements are equivalent:

"If Tom learns discrete math, then he will find a good job"

"Tom will find a good job when he learns discrete math"

"For Tom to get a good job, it is sufficient for him to learn discrete math"

Biconditionals: The biconditional statement p ↔ q is the proposition "p if and only if q". It is true when p and q have the same truth values (true or false), and is false otherwise. Biconditional statements are also called bi-implications. There are other common ways to express p ↔ q:

"p is necessary and sufficient for q"

"if p then q, and conversely"

"piff q"

The truth table for the biconditional statement $p \leftrightarrow q$.

p	q	$p \leftrightarrow q$
T	T	T
T	F	F
F	T	F
F	F	T

Example: You can go to the movie if and only if you buy a ticket. This biconditional-statement is true if both propositions p (go to the movie) and q (buy a ticket) are true or false.

II. 2. 2. Truth Tables for Compound Propositions

We have introduced the four important logical connectives: disjunction, conjunction, conditional and biconditional statements, and negation. With these connectives, we can build complicated compound propositions that involve several propositional variables such as: p, q, r, s,… And we can set up truth table to determine the truth values of these compound propositions.

Example: Set up the truth table for compound proposition: $(p \vee \neg q) \rightarrow (p \wedge q)$

p	q	$\neg q$	$p \vee \neg q$	$p \wedge q$	$(p \vee \neg q) \rightarrow (p \wedge q)$
T	T	F	T	T	T
T	F	T	T	F	F
F	T	F	F	F	T
F	F	T	T	F	F

[4]

III. Computing Theory

III. 1. COMPLEXITY THEORY

The problem in this area is "Why are some problem hard to solve (computability) and the others are easy or simple?". A problem is considered "easy" if it is solvable efficiently. Examples for easy problems are:

- Sorting a sequence of 1,000 numbers
- Searching for a name in the telephone directory
- Computing the fastest routes from Washington D. C. to Miami, Florida.

Otherwise, a problem is considered "hard" if it cannot be solved efficiently, or we don't know how to solve it efficiently. Examples for hard problems are:

- Set up time table for all courses offered at Fort Hays State University.
- Find all prime factors of a 100-digit integer number.
- Design a layout for chips in a VLSI.

III. 2. COMPUTABILITY THEORY

In the years 1930s, three computer scientists: Turing, Church and Godel found out that some simple, fundamental mathematical problems cannot be solved by a computing machine (computing instrument, or computers that were invented in the years 1940s). One example for that problem is logic problem "Is a mathematical statement true or false?"

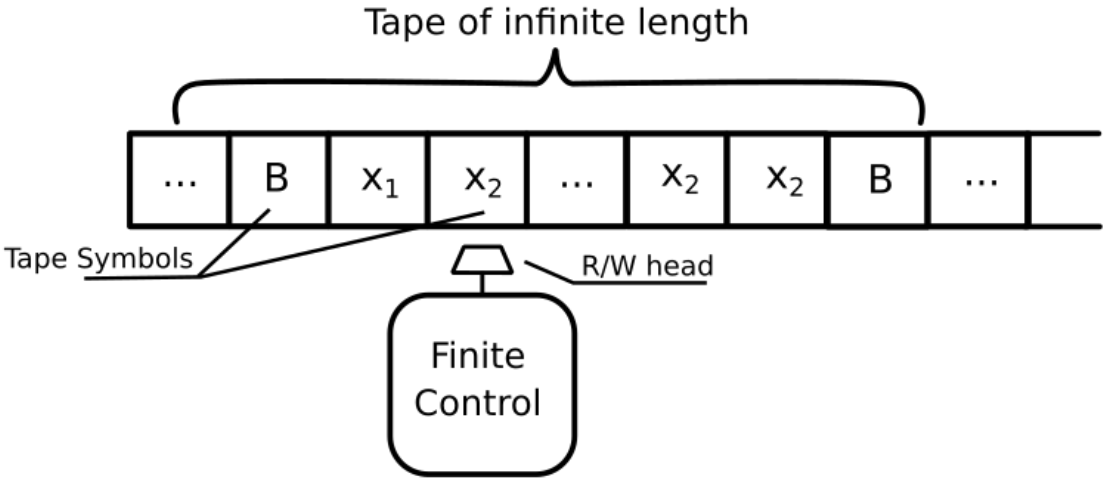
The theoretical computing models were proposed for the understanding of solvable or unsolvable problems, providing the concept for the development of real computers [5].

III. 3. TURING MACHINES

One of computing model proposed by Alan Turing in1936, therefore it is called the Turing machine. This abstract computing machine is similar to a finite automaton and with an unlimited and unrestricted memory. Comparing with other abstract computing models, the Turing machine is more accurate and a general purpose computer that can do anything, a real computer can do. As the topic of this research paper, a Turing machine cannot solve a certain number of problems (halting problems) which are beyond the theoretical limits of computation.

The Turing machine uses a tape as its unlimited memory. It has a read/write head that can move back and forth on the tape, so Turing machine can read or write symbols on the tape. Initially, the tape contains only a

string as input, and blank characters. To store information, the machine writes data on the tape. To read the information stored on the tape, the machine moves its head over the characters. The machine continues to move its head (processing, computing) over the tape until it decides to decide to (stop) produce the output (result). The outputs have only one of the two outcomes (accept or reject), when the Turing machine enters designated accepting and rejecting states. If it does not enter accepting/rejecting state, it will continue forever. (never halting).



The Turing machine is different from finite automata in the following:

- Turing machines can read and write from a tape
- The read-write head can move back and forth (left / right) on the tape
- The tape is infinite (unlimited memory)
- The states of accepting, rejecting take effect immediately

Example: A Turing machine TM is used to test whether an input string is a member of the language L = {c#c | c ∈ {0, 1}*}. The Turing machine TM will accept the input if it is a member of L or reject it otherwise.

Solution: The goal for determination an input string is of the form "two identical strings separated by symbol '#' (pound sign)" or not. The most simple strategy to solve this problem is to move back and forth (zig-zag) over the input string, and to determine the corresponding characters on the two side of symbol '#' whether they are match.

The Turing machine TM will make multiple passes over the input string with read / write head. Each pass, the TM matches one character one each side of the symbol '#'. To keep track of the characters have been checked, it will cross out each character as it examined. Whenever the Turing machine TM discovers a mismatch, it will enter the reject state, otherwise if it crosses all characters (all characters matched successfully), TM will enter the accept state. The algorithm to solve this problem for the Turing machine TM is following:

1. Move the read / write head across the tape to the corresponding positions on both sides of the symbol '#' and determine if the characters matched. If they do not, or if the symbols '#' is not found, reject the input string. Cross off the characters as they are checked, then move on to the next characters.
2. When all characters on the left side of symbol '#' have been crossed off, the TM checks for any remaining characters to the right side of symbol '#'. If there are remaining characters, reject the input string, otherwise accept it.

Figure of some snapshots of the tape illustration:

Suppose the input string entered: 011000#011000

0 1 1 0 0 0 # 0 1 1 0 0 0 b bb . . .
x1 1 0 0 0 # x 1 1 0 0 0 b bb . . .
xx 1 0 0 0 # x 1 1 0 0 0 b bb . . .
. . .
x xxxxx # x xxxxxbbb . . .

(accept) [6]

IV. Computability

A formal system is used for inferring theorems from axioms following a set of rules. Therefore, a formal system is also an "axiomatic" system. These systems can be regarded as the foundation knowledge in mathematics. A formal system represents a well-defined system of abstract thought. A satisfactory formal system should have the following properties:

- Completeness: It should be able to prove or disprove any proposition that can be expressed (input) in the system.
- Consistency: It should be able to prove or disprove a proposition in the system, but not both. The system should produce the same result consistently for each proposition.

In sets theory, a set can have other sets as members. In 1901 Bertrand Russell discovered the Russell's paradox (or Russell's antimony) that proved that some attempted formalizations of the naïve set theory created by George Cantor led to contradiction.

"Consider the set of all sets that do not have themselves as a member. Is this set a member of itself?" This theory of types though not fully accepted has paved the way for new mathematical philosophy.

Look at a computer program that contains a very large binary number, Kurt Godel, a mathematic logician was able to prove "If it is possible to prove, within a formal system that the system is consistent, then the formal system itself is not consistent." or equivalently "If a formal system is consistent, then it is impossible to prove that it is consistent."

IV. 1. RECURSIVE FUNCTION THEORY

According to Godel, a sufficient powerful formal systemcannot be both complete and consistent. A system that can do simple arithmetic on integers could be considered "sufficient powerful", therefore, it is preferable to give up completeness rather than consistency, because a consistency system can prove any proposition within.

We want to have an algorithmic theorem to prove the procedure to distinguish between provable/not-provable propositions. Alan Turing invented the abstract Turing machine for solving this problem, but the Turing machine encountered halting problem, that proved it is not possible to distinguish between solvable /unsolvable.

IV. 2. PRIMITIVE RECURSIVE FUNCTIONS

The recursive functions are defined over the natural numbers I = {0, 1, 2, 3, …}. The recursive functions are considered as "pure symbol systems", numbers are not used in the system alone. We use the system to construct both numbers and arithmetical functions that operates on these numbers. It is a different number system that is similar to Roman numerals

(next(number) = current(number) + 1)

z(x) = 0, s(z(x)) = 1, s(s(z(x))) = 2, …

To find decimal value, just count the number of s's surrounding the central z(x).

- Zero Function: z(x) = z(y) for all x, y ∈ I. This is the number "zero" (0) is written as a function.
- Successor Function: s(x) = x + 1. This function does not return a value, the result of s(x) is s(x)
- Projector Function: $P_1(x) = x$

$P_1(x, y) = x$

$P_2(x, y) = y$

. . .

$P_i^n(x_1, x_2, \ldots, x_n) = x_i$

The projector functions are used to extract one of the parameters. P_1, P_2 are the only defined functions and with no more than two arguments.

Definition: Atotal function f over N (Set N of natural numbers) is primitively recursive if

1. It is one of the three initial functions: Zero, Successor, and Projector functions or
2. It can be gotten by applying composition and recursion finite number of times to the set of initial functions.

IV. 3. COMPOSITION AND RECURSION

Let g_1, g_2, g_3 and h be predefined functions. They can be combined to form new functions, and the combination also must be in defined ways.

a. Composition: if $f(x, y) = h(g_1(x, y), g_2(x, y))$. This composition allows us to use functions (g_1, g_2) as arguments to functions (h).

b. Primitive Recursion: This is a "recursive routine" structured in the form:

$f(x, 0) = g_1(x)$

$f(x, s(y)) = h(g_2(x, y), g_3(f(x, y)))$ (1)

A primitive recursive function is a function formed from the functions: z (zero function), s (successor function), and p_1, p_2 (projector functions), using composition and primitive recursion.
To ensure the recursion terminates, the function should have an extra parameter for decrement every time the function is called. In the following, s(x) is replaced by x and the recursion terminated when it reaches zero (z(x)).

f(……., z(x)) = ……..
f(……., s(x)) = ……..f(……., x), ……..

Examples:

a. Adding two numbers: Following the form (1) above, we have:

add(x, z(x)) = g_1(x)
add(x, s(y)) = h(g_2(x, y), g_3(add(x, y)))
Choosing: $g_1 = p_1$, $g_2 = p_1$, g_3 = s and h = p_2, we have:
add(x, z(x)) = p_1(x)
add(x, s(y)) = p_2(p_1(x, y), s(add(x, y)))
Which is simplified to:
add(x, z(x)) = x
add(x, s(y)) = s(add(x, y))
add(2, 1) = add(s(s(z(x))), s(z(x)))
= s(add(s(s(z(x))), z(x)))
= s(s(s(z(x))))

b. Multiplication two numbers: Using previously defined function add, we go to the simplified form:

multiply(x, s(z(x))) = x
multiply(x, s(y)) = add(x, multiply(x, y))

c. Predecessor of a number: The importantfact here is, the predecessor of a number cannot be negative, so 0 – 1 = 0.

pred(z(x)) = z(x)
pred(s(x)) = x

d. Subtraction two numbers:

subtract(x, z(x)) = x
subtract(x, s(y)) = pre(subtract(x, y))

Examples 1: Given:

g_1(x, y) = x + y
g_2(x, y) = 3xy
g_3(x, y) = 12x

and h(x, y, z) = x + y + z are functions over set N (natural numbers). Find the composition of h with g_1, g_2, g_3.

Solution 1:

h(g_1(x, y), g_2(x, y), g_3(x, y)) = h(x + y, 3xy, 12x) = x + y + 3xy + 12x
Therefore the composition function h with g_1, g_2, g_3 is the function:
f(x, y) = x + y + 3xy + 12x

IV. 4. ACKERMANN'S FUNCTION

Ackermann's function is called mu-recursive function (not primitive recursive), that has the power of a Turing machine. It is defined as:

A(0, y) = y + 1
A(x, 0) = A(x – 1, 1)
A(x, y) = A(x – 1, A(x, y – 1))

It is otherwise defined as:

A(0, y) = y + 1 (i)
A(x + 1, 0) = A(x, 1) (ii)
A(x + 1, y + 1) = A(x, A(x + 1, y)) (iii)

A(x, y) can be computed for any (x, y), hence A(x, y) is a total function and recursive but not primitive recursive.

Example: Calculate A(1, 1), A(2, 1) where A(x, y) is a Ackermann's function.

Solution: Using Ackermann's function defined as (i), (ii), (iii) above

A(1, 1) = A(0 + 1, 0 + 1)
= A(0, A(1, 0)) (iii)
= A(0, A(0, 1)) (ii)

= A(0, 2) (i)
A(1, 1) = 2 + 1 = 3 (i)
A(2, 2) = A(1 + 1, 1 +1)
= A(1, A(2, 1)) (iii)
A(2, 1) = A(1 + 1, 0 + 1)
= A(1, A(2, 0)) (iii)
= A(1, A(1, 1)) (ii)
= A(1, A(0, 2)) (From A(1, 1) = A(0, 2) above)
= A(1, 3) (i)
= A(0 + 1, 2 + 1)
= A(0, A(1, 2)) (iii)
= A(0, 4)
A(2, 1) = 4 + 1 = 5 [7]

V. Conclusion

Mathematics is always regarded as the key to open gates for other areas, especially in scientific disciplines such as engineering and computer science.Computing theories provide computation and logic concepts that became the foundation for building of today computers.

Computability theory provides the computing necessary for answering the question: what type of problems can we solve with a computer and other types that we cannot? The today computer is so powerful that some of us might thing it can do everything (problem solving). The reality is not true or at least "not always true". As an example, consider the question "Does a computer program meet a given specification?"

How can we build a computer for solving that particular question? The answer is impossible, the question or problem above is not solvable for a computer (uncomputable). The question is for computer programmers who write the application program that satisfy program's requirements, specifications.

Computability theory introduced several different formal models of computation. The simplest one is "Finite Automata" which is the foundation for computing, data processing by today computer.Computability theory had its hayday during the early days of computer in the 1930s to 1950s. Today, it is very much finished as a field of research [8].

References

[1]. Hopcroft, Motwani, Ullman. "Introduction to Automata Theory, Languages and Computation".2e, Addition-Wesley 2001. Pages: iii.
[2]. https://en.wikipedia.org/wiki/Theory_of_computation downloaded 07/23/2020 at 12:35pm
[3]. Lou van den Dries. "Mathematical Logic (Math 570) – Lecture Note". Illinois University, Fall semester 2016. Pages: 13. https://faculty.math.illinois.edu/~vddries/main.pdf downloaded 07/26/2020 at9:42AM
[4]. Kenneth H. Rosen. "Discrete Mathematics and its Applications".7e. McGraw Hill 2012.Pages: 2 – 13.
[5]. Anil Maheshwari, MichielSmid. "Introduction to Theory of Computation".School of Computer Science, Carleton University, Ottawa, Canada. April 17, 2019. Pages: 1-3.
[6]. Michael Sipser. "Introduction to the Theory of Computation". 2e. Thomson Course Technology. 2006. Pages: 137-159
[7]. Xavier, Eugene S.P., "Theory of Automata, Formal Languages and Computation". New Age International (P) Limited, Publisher. 2005. Pages: 218-230
[8]. Michael Sipser lectures delivered, Holden Lee noted. "Theory of Computation". MIT Fall 2012. Pages: 5, 6

Hieu D. Vu. "Computing Theories." *IOSR Journal of Computer Engineering (IOSR-JCE)*, 22(5), 2020, pp. 01-07.