Turing Machine Simulator and its Underlying Mechanism

A Thesis Presented to

The Faculty of the Computer Science Department

In (Partial) Fulfillment

of the Requirements for the Degree

Masters of Science in Computer Science

by

Student Name:
Hang ZHANG

Advisor:
Dr.Soltys

November 2019

APPROVED FOR MS IN COMPUTER SCIENCE

_____ Dec 10, 2019

Advisor: Dr. Michael Solty   Date

_____ 12-10-19

Dr. Brian Thoms   Date
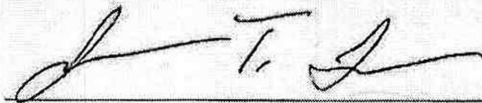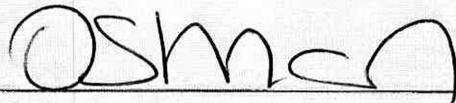
_____ Dec 10, 2019

Dr. Jason Isaacs   Date

APPROVED FOR THE UNIVERITY

_____ 12/13/19

Dr. Ozturgut, Osman   Date

Non-Exclusive Distribution License

In order for California State University Channel Islands (CSUCI) to reproduce, translate and distribute your submission worldwide through the CSUCI Institutional Repository, your agreement to the following terms is necessary. The author(s) retain any copyright currently on the item as well as the ability to submit the item to publishers or other repositories.

By signing and submitting this license, you (the author(s) or copyright owner) grants to CSUCI the nonexclusive right to reproduce, translate (as defined below), and/or distribute your submission (including the abstract) worldwide in print and electronic format and in any medium, including but not limited to audio or video.

You agree that CSUCI may, without changing the content, translate the submission to any medium or format for the purpose of preservation.

You also agree that CSUCI may keep more than one copy of this submission for purposes of security, backup and preservation.

You represent that the submission is your original work, and that you have the right to grant the rights contained in this license. You also represent that your submission does not, to the best of your knowledge, infringe upon anyone's copyright. You also represent and warrant that the submission contains no libelous or other unlawful matter and makes no improper invasion of the privacy of any other person.

If the submission contains material for which you do not hold copyright, you represent that you have obtained the unrestricted permission of the copyright owner to grant CSUCI the rights required by this license, and that such third party owned material is clearly identified and acknowledged within the text or content of the submission. You take full responsibility to obtain permission to use any material that is not your own. This permission must be granted to you before you sign this form.

IF THE SUBMISSION IS BASED UPON WORK THAT HAS BEEN SPONSORED OR SUPPORTED BY AN AGENCY OR ORGANIZATION OTHER THAN CSUCI, YOU REPRESENT THAT YOU HAVE FULFILLED ANY RIGHT OF REVIEW OR OTHER OBLIGATIONS REQUIRED BY SUCH CONTRACT OR AGREEMENT.

The CSUCI Institutional Repository will clearly identify your name(s) as the author(s) or owner(s) of the submission, and will not make any alteration, other than as allowed by this license, to your submission.

Turing Machine Simulator and its Underlying Mechanism

Title of Item

Turing Machine Simulator, Universal Turing Machine, General Recursive Function, Lambda Calculus

3 to 5 keywords or phrases to describe the item

Hang Zhang

Author(s) Name (Print)

Hang Zhang   Dec 12, 2014

Author(s) Signature   Date

Turing Machine Simulator and its Underlying Mechanism

Hang Zhang

December 10, 2019

Abstract

Turing machines(TMs) are mathematical models of computation that define abstract machines. Because of their simplicity and consistency, they are amenable to mathematical analysis. These hypothetical machines are intended to help explore the concept of what it meant to be computable. These models form the foundation of theoretical computer science. A lot of theoretical computer science has been beard on TMs, and so a lot of the primary results are in the language of Turing machines. It's essential for computer science students to build up a strong foundation of computer science by understanding theoretical models behind this field. Nowadays, Universities provide computer science education usually offers one or two classes to introduce Turing Machine. However, studying Turing Machine without a useful visual tool can be a little bit less intuitive. In this thesis, a Turing Machine simulator was created. Also, a universal Turing machine was proposed to show how a simulator works at a lower level.

Contents

List of Figures

1 Introduction

1.1 Motivation

Turing machines are hypothetical machines which were created by Alan Turing in 1936 to explore the concept of what it meant to be computable[4]. A typical Turing Machine has an infinitely long stripe of tape composed of a number of squares with symbols written on them. These squares are used to store inputs and outputs computations for this machine. It also has a head that can read symbols from the squares, write symbols to the squares, and move left and right. It also has a set of transition functions which is essentially a program to tell Turing Machine what tasks to perform based on the machine's state and the read symbol. Every Turing Machine has an initial state and a halt state. A Turing Machine starts its computations from the initial state and finishes its task when it enters a halt state. Without a halt state, a turing machine will never terminate.

Turing Machine is a foundation of modern computer programming. It is very important for computer science students to understand this computing model to build up a strong foundation of their knowledge for computer science. Those popular programming languages and technical frameworks for practical use today will eventually be replaced by new technologies. Instead,the theory behind computer programming has a long-lasting value, making people think about what problems can be solved and how can they be addressed effectively. As for its importance, universities take "Computability" Theory (this course may be called as "Format Language and Automata")as a required class for computer science majors. Exposure to Turing Machine, students can get a general idea of what a computing model can do. However, statically presenting Turing Machine might be less intuitive. Some difficulties may occur in the learning process. First, how should an instructor explain the process from a Turing Machine starts until it terminates? It may need to draw a giant table to show all transition functions and an extremely long turing machine tape and walk students through this process by writing and erasing a massive amount of symbols. If the instructor would like to show students the different outputs of a Turing Machine given by various inputs, that's an enormous burden on the instructor's shoulder. Second, what if a student designs a Turing Machine and then wants to test it with some inputs? He/She needs to check his/her transition table back and forth, while

simulating the turing machine computing process, and it's very easy to make a mistake. Also, this procedure is a little bit longer, and the student might lose his focus on where his work is on, and then starts from the very beginning again. An Turing Machine Simulator can be a beneficial visual aid for this topic. Instructors no longer have to work on drawing Turing Machine on a whiteboard, instead focus on explaining what is happening on the machine. Also, if they want to switch Turing Machines, it can be done by just a few clicks.

A Universal Turing machine (UTM) is a Turing Machine(TM) that can simulate another Turing Machine based on its encoded description and its input. Some ideas in computer science such as compiler, interpreter are inspired by UTM. Designing a Universal Turing Machine is essentially creating a language. Like Micheal Soltys states in his book, "every computer science students should do this exercise "[13]. This is very helpful to students to gain knowledge of what a language is and what requirements are for a language. However, textbooks used in computability theory class either don't mention Universal Turing Machine[15], or lacking explantion for designing a UTM[11]. A Turing Machine simulator is essentially doing the same work as a UTM, but it hides the detail of how UTM works. In this thesis, a Universal Turing Machine is created to reveal how a Turing Machine can simulate another Turing Machine and how a Universal Turing Machine can simulate all Turing Machines.

2 Background

In 1920, David Hilbert proposed a project now known as Hilbert's program[2]. He asked for a complete logical foundation to formulate mathematics. In 1928, he introduced Entscheidungsproblem for this purpose[1]. This problem asks for an algorithm that takes an axiomatic formal system and a logical statement in that system as input, and the algorithm outputs true or false depending on the value of that statement. He believed that every problem is solvable.

However, a lot of mathematicians gave their negative answers to entscheidungsproblem. In the 1930s, GÖdel presented a complete axiomatic system based on first-order logic in his Ph.D. paper[7]. In this paper, GÖdel proved his Completeness Theorem, and it states that for any self-consistent axiomatic system, it must contain theorem that cannot be proven. To prove this statement, Gödel developed a technique now known as Gödel numbering, which can code any formal expression as a single natural number. In 1933, Kurt GÖdel, with Jacques Herbrand, gave a formal definition of his axiomatic system called general recursive functions[8]. General recursive functions are a class of functions built from basic functions with composition, recursion, and minimization operators.

In 1936, Church also proved that entscheidungsproblem is unsolvable by his model of computability of $\lambda - calclus$. Within and by $\lambda - calculus$, there is no effective way to determine whether a λ term has a norm form. Alan Turing introduced Turing Machines[16] in the same year. He proved that entscheidungsproblem is unsolvable by presenting the Halting Problem. Halting Problem is asking whether a program **A** that takes a description of an arbitrary program **B** and **B**'s input as its own input can determine program **B** is able to finish its computations or run forever.

There is an extensive research about computability for mathematical problems. The basic research objects in this field are computable functions. In the notation of an algorithm, if there is an algorithm that can get the same job done for a function, then this function is computable. Formally, a function $f : \mathbb{N}^k \to \mathbb{N}$ is computable $\Leftrightarrow \exists$ Algorithm A for $x \in \mathbb{N}^k$ yields y and $f(x) = y$ [6].

A lot of mathematicians from around the world gave their own opinions about what is a computable function. Kurt GÖdel described Recursive

functions[8], Alonzo Church defined the Lambda calculus[5], Stephen Kleene introduced Formal systems[10], Alan Turing described imaginary computing device as Turing Machines, Emil Post defined Post machines[14], And Markov defined what became known as Markov algorithms [12]. Computation functions are used to discuss computability without referencing any models above. When it comes to any definition, however, it must refer to some specific model of computation. As a function in Turing Machine, it is computable if only if it is computable in $\lambda - Calculus$, and similarly for any other pairs of the above formal languages. Church expressed the belief that the intuitive notion of "computable" is identical to the above models' concept. This belief is called "Church-Turing Thesis" which is universally accepted by mathematicians.

2.1 General Recursive Function

In mathematical logic and computer science, the general recursive functions are a class of partial functions that take a series of natural numbers as input and yield a natural number as output.They are functions that are build from basic functions by composition, primitive recursion, minimisation operators. It is shown that the general recursive functions compute the the same class of functions as by Turing Machines[3].

Basic functions:

1. **Constant function:** A function takes a tuple of number as input, and awalys yields the same value, which is given by

$$C_n(x_1, ...x_k) = n.$$

where n and k are natural numbers

2. **Successor function S:** A function take a natural number x as input, and yields $x + 1$, which is given by

$$S(x) = x + 1$$

3. **Projection function** P_i^k**:** A function takes a tpule of number as input, and gives the value on certain position as output, which is given

4

by $1 \leq i \leq k$:

$$P_i^k(x_1, ..., x_k) = x_i$$

where i and k are natural numbers and $i \geq k$

Operators:

1. **Composition operator** $\circ$: Given a m-place function $h(x_1, ..., x_m)$ and m-place functions $g_1(x_1, ..., x_k), ..., g_m(x_1, ..., x_k)$:

$$h \circ (g_1, ... g_m) = f \quad \text{where} \quad f(x_1, ..., x_k) = h(g_1(x_1, ..., x_k), ..., g_m(x_1, ..., x_k))$$

2. **Primitive recursion operator** ρ: Given a (k + 1)-place function $g(x_1, ..., x_k)$ and (k + 2)-place function $h(y, z, x_1, ..., x_k)$:

$$\rho(g, h) = f \text{ where the (k + 1)-place function } f \text{ is defined by}$$

$$f(0, x_1, ... x_k) = g(x_1, ... x_k)$$
$$(y + 1, x_1, ..., x_k) = h(y, f(y, x_1, ..., x_k), x_1, ..., x_k)$$

3. **Minimization operator** μ: Given a (k + 1)-place total function $f(y, x_1, ..., x_k)$, the k-place **function** $\mu(f)$ is defined by:

$$\mu(f)(x_1, ..., x_k) = z \iff f(z, x_1, ..., x_k) = 0$$

and $f(i, x_i, ..., x_k) > 0$ for $i = 0, ..., z - 1$
minimization seeks that smallet z that meets $f(z, x_1, ..., x_k) = 0$, if this z doesn't exist, search will never terminate.

Example

- The predecessor function Pred:$\mathbb{N} \to \mathbb{N}$, is a partial recursive function, defined by the recusion $pred(0) = 0, Pred(S(n)) = P_1^2(n, Pred(n))$

- Truncated subtraction, where $x \dot{-} y = x - y$ if $x \geq y$, else 0. It is defined by the recursion $x \dot{-} 0 = x, x \dot{-} (y + 1) = Pred(x \dot{-} y)$

2.2 Lambda Calculus

The lambda calculus is older than Turing's machine model, apparently dating from the period 1928-1929[9], and was invented to encapsulate the notion of a schematic function that Church needed for a foundational logic he devised.

Definition Lambda calculus consists of constructing lambda terms and performing reduction operations on them. Lambda terms are defined recursively, t is a λ term when it meets one of the following requirements[9],

1. $t = x$, x is a variable

2. $t = \lambda x.M$, x is variable and M is lambda term

3. $t = (MN)$, M and N are both lambda terms

Hence, there are 3 kinds of λ terms.

- Variable(referencing lambda expressions)

- lambda abstraction(defining functions)

- applications(invoking functions)

Reduction Basic lambda calculus has no "built-in" functions. The meaning of lambda expressions is defined by how expressions can be reduced. There are three kinds of reductions:

- $\alpha - conversion$: changing bound variables, for example

$$\lambda xy.xy = \lambda pq.pq$$

- $\beta - reduction$: applying functions to their arguments, for example:

$$(\lambda x.x)y = y$$

- $\eta - conversion$: if two functions yields the same result for all their arguments, they are the same. for exmple:

$$\lambda x.(fx) = f$$

if f doesn't contain x.

Free and Bound Variables In an expression, each appearance of a variable is either "free" (to λ) or "bound" (to a λ). For example, x is free in the expressions $\lambda qp.x$ but not in expression $\lambda xy.xyxy$.

Normal Form A **normal form** term is a λ term that does not contain any reducible expression, which means no $\beta - Reduction$ can be applied to this term. If a λ term that can become a norm form term, after a finite step of reductions, is called normalizable. For example:

$$(\lambda a.a \quad b) \rightarrow b$$

where the last term is a λ norm form.
Some λ terms are not normalizable. For example:

$$(\lambda z.zz)(\lambda y.yy)$$

. Applying a $\beta - reduction$ to this term always yileds $(\lambda y.yy)(\lambda y.yy)$, $\beta - reduction$ can always be applied to this term.

Weak Norm Form: A λ term has no reducible expression left, but its sub-λ term can still be reduced. For example:

$$\lambda abc.((\lambda x.a(\lambda xy))bc)$$

Evaluation Strategies Evaluation Stategies are used to evaluate a λ expression. Evaluation on an expression can terminate or not depends on the choice of evaluation strategies. **Leftmost strategy** Reduce left most reduciable expression first. This is called `call-by-name`, i.e. functions are called without evaluating their inter λ terms.

$$(\lambda x.z)((\lambda x.xx)(\lambda x.xx)) \rightarrow z$$

If a λ expression, through a finite step of reduction, can become a λ norm form term, the left strategy reaches this normal form.
Rightmost-innermost strategy Evaluate the innermost λ terms among the rightmost reduciable expression. This is called `call-by-value`, i.e. sub-λ terms are evaluated before being reduced in a higher level λ term:

$$(\lambda x.z)((\lambda x.xx)(\lambda x.xx)) \rightarrow$$

$$(\lambda x.z)((\lambda x.xx)(\lambda x.xx)) \rightarrow \cdots$$

Evaluation of inter-most reduciable expression $((\lambda x.xx)(\lambda x.xx))$ yields itself and evaluation does not terminate.

7

Logic and predicates Church booleans are used for boolean values true(T) and false(F) as follows:

$$T = \lambda x \lambda y.x$$

$$F = \lambda x \lambda y.y$$

Some logic operators can be defined by Church booleans.

$$AND = \lambda x \lambda y.xyx$$

$$T = \lambda x \lambda y.xxy$$

$$T = \lambda x.xFT$$

A predicate function can be defined with Church booleans. For example, Z is a predicate, which return T if its argument is Church numberal 0 and return F if it is not.

$$Z = \lambda x.x(\lambda x.F)T$$

$$PRED = (\lambda n.\lambda f.\lambda x.n)(\lambda g.\lambda h.h(gf))(\lambda u.x)(\lambda u.u)$$

Recursion To define a recursion function in $\lambda - Calculus$, it needs a auxillary function Y to reference the recursion function itself becuause of $\lambda - Calculus$'s anonymity.

$$Y = (\lambda y.(\lambda x.y(xx))(\lambda x.y(xx))$$

Applying Y to a function R yields:

$$YR = \lambda x.R(xx)(\lambda x.R(xx)) \rightarrow R((\lambda x.R(xx))(\lambda x.R(xx))) \rightarrow R(YR) \rightarrow R(R(YR))$$

$$\rightarrow R(R(R(...(YR))$$

This means the function R is evaluated using the recursive call YR as the first argument.

2.3 Turing Machine

Turing machines are hypothetical machines that were created by Alan Turing in 1936 to explore the concept of what it meant computable[16]. It consists of an infinitely long tape divided into cells with symbols written on them, a head which can read symbols from cells, write symbols to the cells, move left
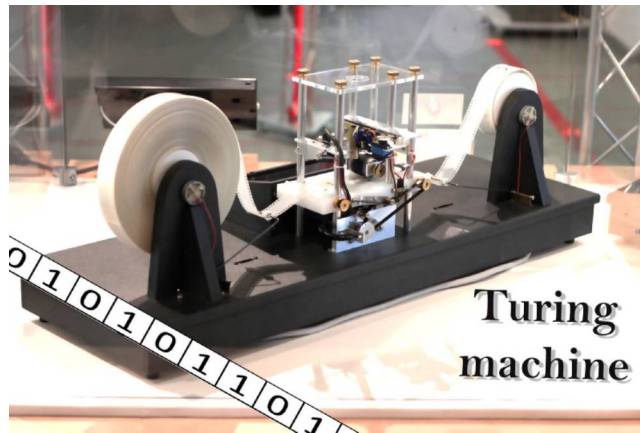
Figure 1: Turing Machine Entity

and right, a state register stores the state of the Turing machine. A stage of computation is determined by the content of the read cell and the state in the state register, head write symbol to the read cell, move left or right and then the state register enters a new state. Every Turing Machine has a halt state, if Turing Machine enters this halt state, it completes its computation task.

Definition Following Soltys' Notation[13], a (one-tape) Turing machine can be formally defined as a 4-tuple $M = \langle Q, \Sigma, \Gamma, \delta \rangle$ where

- Q is a finite, non-empty set of states which includes $q_{init}, q_{accpet}, q_{reject}$

- Σ is the set of input symbols, that is, the set of symbols allowed to appear in the initial tape contents

- Γ is the set of tape symbols, and it is always the case that $\Sigma \subseteq \Gamma$

- δ is a transition function; $\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{Left, Right\}$

An example of Turing Machine M which decides whether an input string has even 0s.

Table 1: Transition table for even 0s

	State	
Letter	q0	q1
B	T	F
0	0, q1,R	0, q0, R
1	1, q0, R	1, q0, R

In this case we have:
$$Q = \{q0, q1, T, F\}$$
$$q_{init} = q0 \qquad q_{accpet} = T \qquad q_{reject} = F$$
$$\Sigma = \{0, 1, B\}$$
$$\Gamma = \{0, 1, B\}$$

2.4 Computational Equivalance

Computability is the ability to solve a problem efficiently. It is also a crucial topic in computer science. Computable functions are the basic objects of this field. As to what is computable, different mathematicians gave their different Answers. Intuitively, a function f from natural numbers to natural numbers is called computable if it can determine the value of $f(x_1, x_2, \cdots, x_n)$ for any argument $x_1, x_2, \cdots, x_n$ given a list of instructions. The most widely studied computation models are the Turing Machines and general recursive functions, and the lambda calculus, all of which have the same power about computability.

Proof Outline

- A turing machine program $\langle M \rangle$ is partial recursive.

- General recursive function can be reduced to λ function.

- λ function can be simulated by Turing machines.

1. Every turing machine program $\langle M \rangle$ is recursive.
   First of all, some recursive functions are listed in an intuitive manner for better understanding the proof.

10

(a) add(0, x) = x
add(n + 1, x) = add(n, x) + 1

(b) mul(0, x) = 0
mul(n + 1, x) = x + mul(n, x)

(c) quo(x, y) = 0 if $x\dot{-}y = 0$ else quo(x, y) = 1 + $quo(x\dot{-}y, y)$

(d) mod(x, y) = 0 if $x = 0$
mod(x + 1, y) = mod(x, y) + 1 if mod(x, y) + 1 $\neq$ y else 0

(e) exp(x, 0) = 1
exp(x, n+1) = exp(x, n) * x

(f) log(0, x) = 0
log(n, x) = 1 + log(quo(n, x), x)

(g) flog(c, n) = 0 if $mod(c, n) \neq 0$ else flog(c, n) = 1 + flog(quo(c, n), n)

GÖdel numbering is used to encode the Turing Machines. The Turing Machines are restricted to to the alphabet $\{0, 1\}$, and 0 is used to take place of black cell. Then, a turing machine tape can be shown by binary numerals.

Given a configuration of a Turing Machine, e.g. A configuration is

0	0	1	0	1	0	1	1	0	0

↑
q7
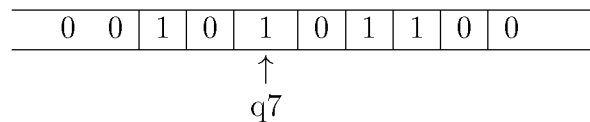
Turing Machine Configuration

coded as the triple of integers $\langle l, q, r \rangle$ where q is the state number (e.g. 7), l is the tape contents to the left of the tape arrow interpreted as a binary integer(e.g. 10 which is 2), and r is the tape contents under and to the right of the arrow interpreted as a binary number reversely (e.g. 1011 is interpreted as 1101 which is 13). The code for the above configuration is

$$c = \langle 2, 7, 13 \rangle = 2^2 \cdot 3^7 \cdot 5^{13}$$

l, q, r can be recoverd from c using recursive function flog:

(a) l = left(c) = flog(c, 2)

(b) q = state(c) = flog(c, 3)

(c) r = right(c) = flog(c, 5)

A computation task is coded by the GÖdel number of a sequence:

$$\langle\langle l_0, q_0, r_0\rangle, \langle l_1, q_1, r_1\rangle, \cdots, \langle l_n, q_n, r_n\rangle\rangle$$

The code shows every thing about the each configuration, but says nothing about how a configuration transfers to another. Consider a Turing Machine $< M >$ that computes function $f(x)$, consecutive x + 1 1s as input for the Turing Machine, f(x) + 1 consecutive 1s for its output. When the Turing Machine starts its computations, the left side of the tape is completely blank, the right side is $11\cdots 1$, a block of $x + 1$ digits 1 with the head with at leftmost 1.

Thus the left number at the start of $< M >' s$ computation of $f(x)$ is 0, and the right number will be

$$exsub(x) = 2^{x+1}\dot{-}1$$

Note that exsub is a composition of Exponentiation function and Truncated Subtraction in formular 2.1 which is a partial recursive function.

When A Turing Machine performs one step, the left and right numbers change based on what symbol is being read, as well as on what action is performed. The read number either 0 or 1, and the right number on the tape is interpreted reversely, it can be calculated by its remainder of 2. If the right number is r, then the symbol read will be

$$read(r) = mod(r, 2)$$

Here,**mod** is a function to compute remainder which is a partial recursive function.

Let l and r be the old left number and old right number, $\bar{l}$ and $\bar{r}$ be the new left number and new right number. Suppose the action is to write

12

a 0 to the read cell. If there was a 0 in the read cell, the content of the cell does not change. So, $\bar{l} = l$ and $\bar{r} = r$. If the symbol in the read cell was 1, the right number decreases 1, $\bar{l} = l$ and $\bar{r} = r \dot{-} 1$. Then, $\bar{r} = r \dot{-} read(r)$ for this action. Similaryly, for action to write a 1 to the read cell. If there was a 1 in read cell, $\bar{l} = l$ and $\bar{r} = r$. If there was a 0 in read cell, $\bar{l} = l$, and right number increase 1 $\bar{r} = r + 1$. Then $\bar{r} = r + 1 \dot{-} read(r)$

Next, consider functions when $\langle M \rangle$ moves left or right. For action to move left, r appends the last digit of l, that is $\bar{r} = 2r + mod(l, 2)$. For left number, it drops its last digit, that is $\bar{l} = quo(l, 2)$. The same logic can be used for action to move right, then $barl = 2l + mod(l, 2)$ and $barr = quo(r, 2)$

Let writing 0, writing 1, moving left, and moving right actions be numbers 0, 1, 2, 3, and action number is a then the new left number will be given by

$$newleft(l, r, a) = \begin{cases} l & \text{if } a = 0 \text{ or } a = 1 \\ quo(l, 2) & \text{if } a = 2 \\ 2l + mod(r, 2) & \text{if } a = 3 \end{cases} \tag{1}$$

Similarly, the new right number will be given by

$$newright(p, r, a) = \begin{cases} r \dot{-} read(r) & \text{if } a = 0 \\ r + 1 \dot{-} read(r) & \text{if } a = 1 \\ 2r + mod(l, 2) & \text{if } a = 2 \\ quo(r, 2) & \text{if } a = 3 \end{cases} \tag{2}$$

When $\langle M \rangle$ halts, it ends with a **standard configuration** where left numberal is 0 and right numberal is $f(x) + 1$ consecutive 1s. The right number is kept tracked through computations. Then right number $r = 2^{f(x)+1} - 1$ when $\langle M \rangle$ finishes its computations, and $f(x)$ will be given by

$$value(r) = log(r, 2)$$

Here log is the partial recursive function, so value is also partial recursive.

So much, for the moment, for the topic of coding the contents of a Turing Machine Tape. The coding for Turing Machines transition rules as follows:

$$m = a_{10}, q_{10}, a_{11}, q_{11}, a_{20}, q_{20}, a_{21}, q_{21} \cdots a_{k0}, q_{k0}, a_{k1}, q_{k1}$$

and **0** will be used as halt state not coded in m. Where k is the number of state of Turing Machines and $a_{ij} <= 3$ **i** is Turing Machines' current state and **j** is the read symbol. Let m be the single code number for this sequence:

$$m = 2^{a_{10}} 3^{q_{10}} 5^{a_{11}} 7^{q_{11}} \cdots$$

Then what action to perform and what state to enter when in state q and read symbol $read(r)$ will be given by:

$$\text{action}(m, q, r) = \text{flog}(m, C(4(q \dot{-} 1) + 2 \times read(r)))$$

$$\text{newstate}(m, q, r) = \text{flog}(m, C(4(q \dot{-} 1) + 2 \times read(r) + 1))$$

Now, turing machine configuration and coding functions of each components have been defined. Next, consider a partial recursive function $code(m, x, t)$ that give the code c for the the configuration after t stages of computation with machine code m and function $f(x)$'s input x. Clearly, at Turing Machines start their computations, the initial configuration code c is given by

$$c_0 = code(m, x, 0) = \langle 0, 1, exsub(x) \rangle = 2^0 3^1 5^{exsub(x)}$$

that is left number is 0, state is 1, and right number is $exsub(x)$.

A procudure to get code c_{t+1} from c_t as follows:

1) $l_t = left(c_t) \quad q_t = state(c_t) \quad r_t = right(c_t)$
2) $a_t = action(m, q_t, r_t)$
3) $q_{t+1} = newstate(m, q_t, r_t)$
4) $l_{t+1} = newleft(a_t, l_t) \quad r_{t+1} = newright(a_t, r_{t+1})$
5) $c_{t+1} = \langle l_{t+1}, q_{t+1}, r_{t+1} \rangle$

Thus, a function from c_t to c_{c+1} is defined as follows:

$$c_{t+1} = \text{getcode}(m, c_t)$$

where getcode is a composition of the functions left, state, right, action, newstate, newleft, newright, and tuple, and is therefore is a partial recusive function.

The function $code(m, x, t)$, giving the code for the configuration after t stages of computation, can then be defined by partial recursion as follows:

$$\text{code}(m, x, 0) = \langle 0, 1, exsub(x) \rangle$$
$$\text{code}(m, x, t + 1) = \text{getcode}(m, \text{code}(m, x, t))$$

It follows that **code** is itself a partial recursive function. The machine will halt when $\text{state}(\text{code}(m, x, t)) = 0$, and will then halt in standard position where right number is $f(x) + 1$ consecutive 1s. If the machine halts in standard configuration at time t, then the output of the machine will be given by

$$\text{output}(m, x, t) = \text{value}(\text{right}(\text{code}(m, x, t)))$$

Then t will be given by

$$\text{halt}(m, x) = \begin{cases} \text{the least } t \text{ such that } state(code(m, x, t)) = 0 & \text{if such a t exists} \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$(3)$$

This function is obtained from compoisition and minimisation, thus it is partial recursive or total recursive.

Let $f'(m, x) = output(m, x, halt(m, x))$, then $f'(m, x)$ is recursive. If $\langle M \rangle$ computes $f(x)$, then $f(m, x)$ computes $f(x)$, so $\langle M \rangle$ is recursive.

2. partial recursive functions can be reduced to lambda terms.
   Here, we show how basic functions and operators from partial recursive function can be represented in lambda terms.

15

Integer, Constant, Projection, Successor

- **Church numerals** are used to represent integers in $\lambda - Caculus$. Church defined natural numbers in $\lambda - Caculus$ as follows:

$$0_c = \lambda f.\lambda x.x$$
$$1_c = \lambda f.\lambda x.fx$$
$$2_c = \lambda f.\lambda x.f(fx)$$
$$3_c = \lambda f.\lambda x.f(f(fx))$$
$$\vdots$$
$$n_c = \lambda f.\lambda x.\underbrace{f(f(f\cdots))}_{n}x)$$

- The constant functions return a constant no matter what their parameters that is $C^n(x_1, x_2, \cdots, x_k) = n_c$. This can be represented by the lambda expression $\lambda x_1, x_2, \cdots, x_k.n_c$,

- The projection functions takes a tuple of numbers as input and return an integer in this tuple, $P_i^k = (x_1, x_2, \cdots, x_k) = x_i$. This can be represented by the λ expression $\lambda x_1 x_2 \cdots x_k.x_i$

- The successor function take an integer **n** as input and returns $n + 1$, that is, $S(n) = n + 1$, this can be represented by the lambda expression $\lambda n_c.\lambda f.\lambda x.f(n_c fx)$,

Composition, Primitive Recusion, Minimization

- **Composition**
  The composition of function h with the functions $g_1, g_2, \cdots, g_m$, where h is a (m-place) function, and each of g function is a k-place function. Then the composition is

$$f(x_1, x_2, \cdots, x_k) = h(g_1(x_1, x_2, \cdots, x_k), g_2(x_1, x_2, \cdots, x_k), \cdots, g_m(x_1, x_2, \cdots, x_k)).$$

This can be represented by lambda expression

$$\lambda h g_1 g_2 \cdots g_m x_1 x_2 \cdots x_k.h(g_1 x_1 x_2 \cdots x_k)(g_2 x_1 x_2 \cdots x_k) \cdots (g_m x_1 x_2 \cdots x_k)$$

- **Primitive Recursion** $\rho^n(g,h)$ is the function f such that:

$$f(0, x_1, x_2 \cdots, x_k) = g(x_1, x_2, \cdots, x_k)$$

$$f(y+1, x_1, x_2 \cdots, x_k) = h(y, f(y, x_1, x_2, \cdots, x_k), x_1, x_2, \cdots, x_k)$$

  This can be represented by the following lambda expression:

$$\lambda ghx_1x_2 \cdots x_n.Y(\lambda fx.Z(gx_1x_2 \cdots x_n)(hx_1x_2 \cdots x_n)(Px(f(Px))))$$

  where Y is fixed-point, P is the predecessor function and Z is used to test whether its argument is 0.

- **Minimization** $\mu(f)(x_1x_2 \cdots x_n)$ returns the smallest z such that

$$f(x_1x_2, \cdots, x_n, z) = 0$$

  This can be represented by the following lambda expression:

$$\lambda fx_1x_2 \cdots x_n.(Y(\lambda hz.Z(fx_1x_2 \cdots x_nz)z(h(Sz)))Z)$$

  where S is Successor function

  Now, we have shown that general recursive function can be reduced to lambda calculus.

3. lambda calculus can be reduced to Turing Machine.
   Before thinking about the Turing machine itself, it is important to represent a λ term in a way that makes their evaluation easy. In particular, we wish to avoid α-conversion. De Bruijn indices allows us to do that.

De Bruijn indices   The unnamed λ-calculus is defined as follows:

$$M ::= n|\lambda M|(MM)$$

where n – natural numbers greater or equal to 0 are variables. The n indicates the position of the λ binding this variable, starting from inside. For example:

$$\lambda x.\lambda y.x = \lambda\lambda 1 \qquad \lambda x.(x \quad x) = \lambda(0 \quad 0)$$

17

Encoding we use alphabet $\Sigma = \{\lambda, B, (,), 0, 1, \odot\}$, and translate a lambda-term M into a tape $\bar{M}$ in the following way.

$$n = \begin{cases} \odot & \text{if n is a free variable} \\ \odot E & \text{if n is a bound variable} \end{cases}$$

$$\lambda M := \lambda \bar{M}$$

$$(M \quad N) := (\bar{M}) \quad (\bar{N})$$

where E is the binary representation of the De Bruijn index of that occurrence. Consider $M = (\lambda q.q)(\lambda q.\lambda p.q)$, then its

De Bruijn form is $(\lambda 0)(\lambda\lambda 1)$, and our tape for this term is $(\lambda \odot 0)(\lambda\lambda \odot 1)$.

The Turing Machine $\langle M_{TM} \rangle$ operates by iteratively performing the following step. Take the expression $(\lambda x.\lambda y.xyy)(\lambda y.y)(\lambda x.x)$ as an example, this expression can be transformed into $(\lambda\lambda \odot 1 \odot 0 \odot 0)(\lambda \odot 0)(\lambda \odot 0)$ as a input string for a TM $\langle M_{TM} \rangle$.

Step 1: The $\langle M_{TM} \rangle$ looks for the first pair of parenthesis. Here, first pair is in the sense of the first (we find, combined with its close counterpart. If we cannot find a pair of parenthesis, the machine halts.

Step 2: Locate the second pair of parenthesis with the similar definition of step 1. If the second pair can not be found, then erase the first pair of parenthesis. Otherwise, erase the first occurrence of λ and replace every occurrence of the bounded variable by the content inside the second pair (including the parenthesis), then erase the second pair of parenthesis and whatever inside it.

Step 3: Erase first parenthesis if no λ can be found inside it (without looking nested parenthesis).

Step 4: Concatenate the left part and the right part. In the example, the tape becomes $(\lambda(\lambda \odot 0) \odot 0 \odot 0)(\lambda \odot 0)$

Every time the sequence of steps is essentially a normalization step. So, $\langle M_{TM} \rangle$ halts on M if and only if M is normalizing and the output will

be the normal form of M. In this example, the tape evolves as follows:

$$(\lambda\lambda\;\odot\;1\;\odot\;0\;\odot\;0)(\lambda\;\odot\;0)(\lambda\;\odot\;0)$$

$$(\lambda(\lambda\;\odot\;0)\;\odot\;0\;\odot\;0)(\lambda\;\odot\;0)$$

$$(\lambda\;\odot\;0)(\lambda\;\odot\;0)(\lambda\;\odot\;0)$$

$$(\lambda\;\odot\;0)(\lambda\;\odot\;0)$$

$$\lambda\;\odot\;0$$

Turing Machines can intrepte $\lambda - Calculus$, and General recursive functions can be reduced to $\lambda - Calculus$. Hence, a turing machine simular that can animate the computing process of Tuinr Machines, is not only beneficial for learning the concept of Turing Mchines, but also $\lambda - Calculus$ and General recursive function, and from a higher level, computable funcitons.

3 Turing Machine Simulator

Turing Machines are imaginary devices for mathematical computation, which perform their computations by manipulating symbols on a finitely length of tape according to a set of transition functions. Turing machines are models that can simulate any computer algorithm no matter how complex it is.

Nowadays, Turing Machines are used to help people gain a better understanding of algorithms and how a computer works, which is why this simulator was created.

3.1 Code

Before going through the simulator's functionality, it is imperative to provide the data structure to be used.

Data Structure

- Node

```
class Node:
  def __init__(self, letter):
   self.letter = letter
   self.R = None
   self.L = None
# simulate ''cell'' in the turing machine's tape
```

- Action

```
class Action:
  def __init__(self, letter, go_to, state):
    self.write_letter = letter \\ write letter
    self.go_to = go_to \\ movie direction
    self.next_state = state \\ next state
    self.length = len(letter + go_to + state)
# this will be used to calculate cells' width
# of a transition table
```

20

```
                    # which simulates instruction part of each transition rule
```

- Tape

```
# A turing machine takes a string as input
# and we use double linkedlist to simulate the tape
class Tape:
  def __init__(self, str1):
    self.arrow_node = None
    self.length = 0
    self.constructor(str1)

  def constructor(self, str1):
    if not str1:
      raise RuntimeError('input cannot be empty')
    pre_node = None
    for char in str1:
      self.length += 1
      cur_node = Node(char)
      if self.arrow_node is None:
        self.arrow_node = cur_node
      if pre_node is not None:
        cur_node.L = pre_node
        pre_node.R = cur_node
      pre_node = cur_node
```

Utils

- parser

```
# simulator takes a file as input
#to construct transition rules.
def parse_tm(file):
  res = []
  with open(file) as f:
    strs = f.read()
```

```
        rules = strs.split(";")
        for i in rules:
          rule = i.split()
          if len(rule) == 5:
            res.append(rule)
      return res
```

- tape printer

```
def print_tape(node):
  while node.L is not None:
    node = node.L
    res = ""
    while node is not None:
      if node.letter != "_":
        res += node.letter
        node = node.R
  print(res)
  # this is used to print out the result given by
  # a string running on a turing machine
```

Main Function

- Construct transition table

```
def construct_transition_table(rules):
  init_state = None
  table = dict()
```

Since δ is a transition function; $\delta : Q \times \Gamma \to Q \times \Gamma \times \{Left, Right\}$

```
  for i in rules:
    key = i[0] + "-" + i[1]
  # i[0] for current state
  # i[1] for reading letter
    if init_state is None:
```

22

```
                    init_state = i[0]
                  if key not in table:
                    table[key] = Action(i[2], i[3], i[4])
            # i[2] for writing letter
            # i[3] for next state
            # i[4] for moving direction
                  else:
                    print("Transition table exists key conflict ")
              return table


tm_file = sys.argv[1]
init_state = sys.argv[2]
tm_rules = parse_tm(tm_file)
# parse file to transition rules

def run_tm(rules, input_str, init_state=None):
  transitions = construct_transition_table(rules)
  stop = ["T", "F", "H"]
# terminate states
# T: accept, F: reject H: halt
  if init_state is None:
    init_state = rules[0][0]
# if init_state is not provided,
# use current state in the first transition rule
# as initial state
  cur_state = init_state
# to record runing steps for turing machine
  step = 0
  tm_tape = Tape(input_str)
  arrow_node = tm_tape.arrow_node
  cur_ins = cur_state + "-" + arrow_node.letter
# current instruction

  while cur_state not in stop:
    step += 1
    if cur_ins not in transitions:
      break
```

```
      arrow_node.letter = transitions[cur_ins].write_letter
      if transitions[cur_ins].go_to == "L":
        if arrow_node.L is None:
          arrow_node.L = Node("_")
          arrow_node.L.R = arrow_node
          cur_ins = transitions[cur_ins].next_state + "-" \
+ arrow_node.L.letter
          arrow_node = arrow_node.L
      else:
        if arrow_node.R is None:
          arrow_node.R = Node("_")
          arrow_node.R.L = arrow_node
          cur_ins = transitions[cur_ins].next_state + "-" \
+ arrow_node.R.letter
          arrow_node = arrow_node.R
      cur_state = cur_ins[:cur_ins.index("-")]
    return step, cur_state
# return steps and final state
```

The code above consists of core logic of building a turing simulator without GUI part.

3.2   Configure a transition table

According to the transition funciton:

$$\delta : Q \times \Gamma \to Q \times \Gamma \times \{Left, Right\}$$

A turing machine file is composed as below:

```
q0 * 1 R q0; q0 1 1 R q0;
q0 _ _ R H; q0 0 * L q1;
q1 1 1 L q1; q1 _ _ R q2;
q1 0 0 R q2; q2 1 0 R q0;
q2 0 0 R q2; q2 * 0 R q0;
q2 _ _ R H; q1 * * R H;
```

This is a turing machine which can be used to do simple sort, given a string in the alphabet $\Sigma = \{0, 1\}$, yields a string all 0s are before 1s. A transition is defined by a tuple with 5 symbols(C, R, W, D, N) where

- C: current state

- R: reading letter

- W: writing letter

- D: moving direction

- N: next state

";" is used as delimiter to separate each transition.

3.3 Usage

Run **python view.py** <**Your Turing Machine Configuration file**> through terminal. Once you entered this program, you will see an overview of this simulator as Figure 2:
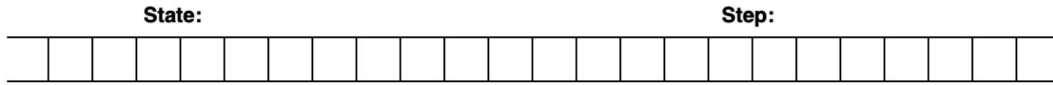
	q1	q3	q6	q2	q5	q0	q4
2	2-R-H	2-R-q3	2-L-q6	2-R-H	2-L-q5	*-R-q3	#-L-q4
_	_-R-q2	_-L-q4	_-R-H	_-R-H	_-R-q0	_-R-H	_-L-H
1	1-L-q1	1-R-q3	1-L-q6	0-R-q0	1-L-q5	1-R-q0	#-L-q6
0	0-R-q2	0-R-q3	0-L-q6	0-R-q2	0-L-q5	*-L-q1	#-L-q5
*	*-R-H	*-R-H	1-R-q0	0-R-q0	*-L-q1	1-R-q0	2-R-q0
#	#-R-H	#-L-q4	#-R-H	#-R-H	#-R-H	2-R-q0	#-R-H

< < < <arrow_speed>>>>>>>

70

play   stop   next

input string: ok

<<<<<<arrow_position>>>>>

1

Figure 2: Turing Machine Simulator Overview

Now we introduce each part of this simulation. Going from top to bottom:

State: Step:
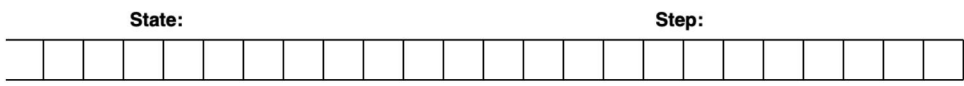
Figure 3: Turing Machine Tape

first, as it shows in Figure 3 we see a tape with empty cells that are meant to be filled with letters from its alphabet Σ, in this case for turing machine doing color sort, which will a string all 1s in the middle, all 0s before 1s, all 2s after 1s, given an input string composed by $\{0, 1, 2\}$.

There is an fixed arrow which enables you to see the tape moving animation when the machine is executing your code.

	q1	q3	q6	q2	q5	q0	q4
2	2-R-H	2-R-q3	2-L-q6	2-R-H	2-L-q5	*-R-q3	#-L-q4
_	_-R-q2	_-L-q4	_-R-H	_-R-H	_-R-q0	_-R-H	_-L-H
1	1-L-q1	1-R-q3	1-L-q6	0-R-q0	1-L-q5	1-R-q0	#-L-q6
0	0-R-q2	0-R-q3	0-L-q6	0-R-q2	0-L-q5	*-L-q1	#-L-q5
*	*-R-H	*-R-H	1-R-q0	0-R-q0	*-L-q1	1-R-q0	2-R-q0
#	#-R-H	#-L-q4	#-R-H	#-R-H	#-R-H	2-R-q0	#-R-H

Figure 4: Turing Machine Transition Table

A transition table in Figure 4 is used to show your transition rules of your turing machine.

	q1	q3	q6	q2	q5	q0	q4
2	2-R-H	2-R-q3	2-L-q6	2-R-H	2-L-q5	*-R-q3	#-L-q4
_	_-R-q2	_-L-q4	_-R-H	_-R-H	_-R-q0	_-R-H	_-L-H
1	1-L-q1	1-R-q3	1-L-q6	0-R-q0	1-L-q5	1-R-q0	#-L-q6
0	0-R-q2	0-R-q3	0-L-q6	0-R-q2	0-L-q5	*-L-q1	#-L-q5
*	*-R-H	*-R-H	1-R-q0	0-R-q0	*-L-q1	1-R-q0	2-R-q0
#	#-R-H	#-L-q4	#-R-H	#-R-H	#-R-H	2-R-q0	#-R-H

Figure 5: Turing Machine Transition Table Highlighted

A transition rule is highlighted(Figure 5) while the machine is executing it.

Figure 6: Turing Machine Operation Box

Now we introduce the functionality of each part: input box,play, stop, next, changing speed, set arrow position(6). Let's go over them one by one:

1. **Input Box:**
   This allows you to feed different inputs into this program at each run. After you give an input to this program, clicking OK will load your input to the tape.

2. **Play**
   This is used to run the code and continue the turing machine after you enter **Step Mode**

3. **Transition Table**
   It shows the actual code of the turing machine. With hightlighted cell while the machine is ruining, it enables us to know which code snippet is running.

28

4. **Next**

This is used for taking one step forward. Once we load the code in the machine it will get clear. As an instructor, it's quite useful to explain what is happening on the machine.

5. **Arrow Position**

It servers three main functionality.

   (a) set up initial arrow position

   It's convenient to set up initial arrow position with just intuitive operations, instead of writing turing machine transition rules.

   (b) moving position arrow after turing machine terminated can help us see the whole final output.

6. **Speed**

This allows you to change the moving speed of the arrow

7. **Stop**

This is used to terminate your program.

4 Universal Turing Machine

4.1 Background

The Turing Machine Similator is a software that takes Turing Machine program and the program's input decription as its own input, and it animates the TM program's computing process. Here, the Machine Simulator actually serves as a Universal Turing Machine. Before computers, there were no softwares avaiable. It brought a problem for Turing Machines. For each new computation task, a different Turing Machine must be created to serve the purpose. It is neither flexible nor cost-effective for hardwares. This is why Alan Turing created the idea of **Universal Turing Machine** . A **Universal Turing Machine(UTM)** is a Turing machine(TM) that can simulate an arbitrary TM on arbitrary input by taking description of the TM being simulated and its input as input and performing action based a set of transition functions.

4.2 Encoder

First, to create a description for a TM being simulated, it requires an encoder to UTM be able to recoginize this description. Here, we introduce how we encode $\langle M \rangle$(Turing Machine) to a $\langle M \rangle$ description and its input which are concatenated as a string that we can load it on $\langle UTM \rangle$'s tape.

1. Schema(structure)
   We still use the same symbolism for $\langle M \rangle$ we introduced previously. We add Caret $\wedge$ to these symbols denote encoded symbols and other symbols such as ., : are used literally. A description should look like below:
   $$\hat{Q}\hat{\Gamma}^{\,\cdot}\hat{\delta}_1, \hat{\delta}_2, \cdots\cdots \hat{\delta}_n :> \hat{\Gamma}_1.\hat{\Gamma}_2.\hat{\Gamma}_3.\cdots\cdots \hat{\Gamma}_m.$$

2. Encode TM alphabet
   Binary encoding is used in general where
   $$\|\hat{\Gamma}_i\| = \lceil log_2\|\Gamma\|\rceil$$

   For example:
   $$\Gamma = \{B, 0, 1\}$$

   then
   $$\hat{B} = 00 \qquad \hat{0} = 01 \qquad \hat{1} = 10$$

3. Encode TM state set
   The same encoding strategy is used for state set as encoding TM aphalbet.

   $$\|\hat{Q}\| = \lceil log_2\|Q\|\rceil$$

   For example:

   $$Q = \{q0, q1, q2, q3\}$$

   then

   $$\hat{q0} = 00 \qquad \hat{q1} = 01 \qquad \hat{q2} = 10 \qquad \hat{q3} = 11$$

4. Encode moving direction

   - 0:left
   - 1:right

After encoding these parts, we place them in the schema structure, this makes a $\langle UTM \rangle$ input. We can still set up initial arrow position, by adjusting the position of symbol $\wedge$. The $\langle M \rangle_{color\_sort}$ as in Figure 7 can be a good a example to explain the encoder.

```
q0 * 1 R q0; q0 1 1 R q0; q0 _ _ R H; q0 0 * L q1; q0 2 * R q3; q0 # 2 R q0;
q1 1 1 L q1; q1 _ _ R q2; q1 0 0 R q2; q1 * * R H; q1 # # R H; q1 2 2 R H;
q2 0 0 R q2; q2 * 0 R q0; q2 _ _ R H; q2 1 0 R q0; q2 # # R H; q2 2 2 R H;
q3 0 0 R q3; q3 1 1 R q3; q3 2 2 R q3; q3 # # L q4; q3 _ _ L q4; q3 * * R H;
q4 0 # L q5; q4 1 # L q6; q4 * 2 R q0; q4 # # R H; q4 2 # L q4; q4 _ _ L H;
q5 0 0 L q5; q5 1 1 L q5; q5 2 2 L q5; q5 * * L q1; q5 # # R H; q5 _ _ R q0;
q6 0 0 L q6; q6 1 1 L q6; q6 2 2 L q6; q6 * 1 R q0; q6 # # R H; q6 _ _ R H;
```

Figure 7: Color sort transition rules

1. state set:

   $$* : 000 \quad 1 : 001 \quad \_ : 010 \quad 0 : 001 \quad 2 : 110 \quad \# : 101$$

2. alphabet set:

   $$H : 001 \quad q0 : 000 \quad q1 : 010 \quad q2 : 100 \quad q3 : 011 \quad q4 : 101 \quad q5 : 110 \quad q6 : 111$$

31

3. description in Figure 8:

```
# 000000`0000000011000,0000010011000,0000100101001,0000110001010,0001000001011,
0001011001000,0100010011010,0100100101100,0100110111100,0100000001001,0101011011001,
0101001001001,1000110111100,1000000111000,1000100101001,1000010111000,1001011011001,
1001001001001,0110110111011,0110010011011,0111001001011,0111011011101,0110100101101,
0110000001001,1010111011110,1010011011111,1010001001000,1011011011001,1011001011101,
1010100101001,1100110111110,1100010011110,1101001001110,1100000001010,1101011011001,
1100100101000,1110110111111,1110010011111,1111001001111,1110000011000,1111011011001,
1110100101001
```

Figure 8: Color sort transition description

4. input:

  • original

$$> 20112011$$

  • encoded input

$$> 100.011.001.001.100.011.001.001.$$

5. tape in Figure 9

```
# 000000`0000000011000,0000010011000,0000100101001,0000110001010,
0001000001011, 0001011001000,0100010011010,0100100101100,0100110111100,
0100000001001,0101011011001, 0101001001001,1000110111100,1000000111000,
1000100101001,1000010111000,1001011011001, 1001001001001,0110110111011,
0110010011011,0111001001011,0111011011101,0110100101101, 0110000001001,
1010111011110,1010011011111,1010001001000,1011011011001,1011001011101,
1010100101001,1100110111110,1100010011110,1101001001110,1100000001010,
1101011011001, 1100100101000,1110110111111,1110010011111,1111001001111,
1110000011000,1111011011001, 1110100101001:.010>100.011.001.001.100.011.
001.001.010.010.010.010.010.010.
```

Figure 9: initial UTM tape

4.3 Design

1. Fetch Letter Turing Machine

Procedure   Start from left end

Step 1: Go to the right find symbol >

Step 2: Go to the right to find the next symbol .

Step 3: Go back to left find first **0** or **1** replace them with **x**, **y** repectively.
If find > before any **0**, **1** do step7

Step 4: Go to the left find symbol `

Step 5: Go to the left find first **0** or **1** and replace them with **x** if we get
x from step3, otherwise **y**.

Step 6: From this position do step1

Step 7: Find next .

Step 8: Go to the left reset all **x,y** to **0**, **1**

Transition Rules   (Figure 10)

```
q0 0 0 R q0; q0 x x R q0; q0 y y R q0; q0 1 1 R q0; q0 , , R q0; q0 . . R q0; q0 ` ` R q0; q0 > > R q1; q0 : : R q0; q0 _ _ R q0;
q1 > > R q1; q1 x x R q1; q1 y y R q1; q1 1 1 R q1; q1 0 0 R q1; q1 . . L q2; q1 ` ` R q1; q1 , , R q1; q1 : : R q1; q1 _ _ R q1;
q2 0 x L q3; q2 1 y L q4; q2 x x L q2; q2 y y L q2; q2 > > R q7; q2 ` ` L q2; q2 , , R q2; q2 . . R q2; q2 : : L q2; q2 _ _ R q2;
q3 0 0 L q3; q3 1 1 L q3; q3 , , L q3; q3 ` ` L q5; q3 > > L q3; q3 . . L q3; q3 x x L q3; q3 y y L q3; q3 : : L q3; q3 _ _ R q3;
q4 0 0 L q4; q4 1 1 L q4; q4 , , L q4; q4 ` ` L q6; q4 > > L q4; q4 . . L q4; q4 x x L q4; q4 y y L q4; q4 : : L q4; q4 _ _ R q4;
q5 0 x R q0; q5 1 x R q0; q5 x x L q5; q5 y y L q5; q5 > > R q5; q5 . . R q5; q5 , , R q5; q5 ` ` R q5; q5 : : R q5; q5 _ _ R q5;
q6 0 y R q0; q6 1 y R q0; q6 x x L q6; q6 y y L q6; q6 > > R q6; q6 . . R q6; q6 , , R q6; q6 ` ` R q6; q6 : : R q6; q6 _ _ R q6;
q7 x 0 R q7; q7 y 1 R q7; q7 0 0 R q7; q7 1 1 R q7; q7 . . L q8; q7 , , R q7; q7 ` ` R q7; q7 > > R q7; q7 : : R q7; q7 _ _ R q7;
q8 x 0 L q8; q8 y 1 L q8; q8 0 0 L q8; q8 1 1 L q8; q8 ` ` L q8; q8 , , L q8; q8 . . L q8; q8 > > L q8; q8 : : L q8; q8 _ _ R H;
```

Figure 10: Fetch letter TM

Example

- Tape configruation when the machine starts(Figure 11)

```
# 000000`0000000011000,0000010011000,0000100101001,0000110001010,
0001000001011, 0001011001000,0100010011010,0100110111100,
0100000001001,0101011011001, 0101001001001,1000110111100,1000000111000,
1000100101001,1000010111000,1001011011001, 1001001001001,0110110111011,
0110010011011,0111001001011,0111011011101,0110100101101, 0110000001001,
1010111011110,1010011011111,1010001001000,1011011011001,1011001011101,
1010100101001,1100110111110,1100010011110,1101001001110,1100000001010,
1101011011001, 1100100101000,1110110111111,1110010011111,1111001001111,
1110000011000,1111011011001, 1110100101001:.010>100.011.001.001.100.011.
001.001.010.010.010.010.010.010.
```

Figure 11: Color Sort Fetch letter TM Starts

- Tape configruation when the machine halts(Figure 12)

```
# 000100`0000000011000,0000010011000,0000100101001,0000110001010,
0001000001011, 0001011001000,0100010011010,0100100101100,0100110111100,
0100000001001,0101011011001, 0101001001001,1000110111100,1000000111000,
1000100101001,1000010111000,1001011011001, 1001001001001,0110110111011,
0110010011011,0111001001011,0111011011101,0110100101101, 0110000001001,
1010111011110,1010011011111,1010001001000,1011011011001,1011001011101,
1010100101001,1100110111110,1100010011110,1101001001110,1100000001010,
1101011011001, 1100100101000,1110110111111,1110010011111,1111001001111,
1110000011000,1111011011001, 1110100101001:.010>100.011.001.001.100.011.
001.001.010.010.010.010.010.010.
```

Figure 12: Color Sort Fetch letter TM Ends

2. Match Instruction Turing Machine
   instruction is a tuple (Q, Γ) which decides the action of TM which is $(Q, \{left, right\}, \Gamma)$

   Procedure Start from the left end

   Step 1: Go to the right find first **0** or **1** and replace them with **x**, **y** repectively

   Step 2: Go to the right find symbol `, if cannot find any
   0 or **1**, stop.

   Step 3: Go to the right find first **0** or **1**, and replace them with **x**, **y** repectively, if the symbol matches symbol from step 1, go back to the left end then, do step1. Otherwise, do step4.

   Step 4: Mark all **0** and **1** with **x**, **y** until meet {,} or {:}. If {,} then go to the left find symbol {`}, otherwise stop(this means cannot find instruction in transition table in the sense of simulating a turing machine).

   Step 5: Reset **x**, **y** to **0**, **1** to the left end

34

Transition Rules (Figure 13)

```
q0 0 x R q1; q0 1 y R q2; q0 x x R q0; q0 y y R q0; q0 ` ` R H; q0 , , R q0; q0 . . R q0; q0 : : R q0; q0 > > R q0; q0 _ _ R q0;
q1 x x R q1; q1 0 0 R q1; q1 1 1 R q1; q1 y y R q1; q1 ` ` R q3; q1 , , R q1; q1 . . R q1; q1 : : R q1; q1 > > R q1; q1 _ _ R q1;
q2 0 0 R q2; q2 1 1 R q2; q2 x x R q2; q2 y y R q2; q2 ` ` R q4; q2 , , R q2; q2 . . R q2; q2 : : R q2; q2 > > R q2; q2 _ _ R q2;
q3 x x R q3; q3 y y R q3; q3 , , R q3; q3 0 x L q5; q3 1 y R q6; q3 ` ` R q3; q3 . . R q3; q3 : : R q3; q3 > > R q3; q3 _ _ R q3;
q4 x x R q4; q4 y y R q4; q4 , , R q4; q4 1 y L q5; q4 0 x R q6; q4 ` ` R q4; q4 . . R q4; q4 : : R q4; q4 > > R q4; q4 _ _ R q4;
q5 ` ` L q5; q5 x x L q5; q5 y y L q5; q5 , , L q5; q5 _ _ R q0; q5 0 0 L q5; q5 1 1 L q5; q5 : : L q5; q5 > > R q5; q5 . . L q5;
q6 0 x R q6; q6 1 y R q6; q6 , , L q7; q6 : : R H; q6 x x R q6; q6 y y R q6; q6 . . R q6; q6 > > R q6; q6 ` ` R q6; q6 _ _ R q6;
q7 x x L q7; q7 y y L q7; q7 0 0 L q7; q7 1 1 L q7; q7 , , L q7; q7 . . L q7; q7 : : L q7; q7 > > L q7; q7 ` ` L q8; q7 _ _ L q7;
q8 x 0 L q8; q8 y 1 L q8; q8 _ _ R q0; q8 0 0 L q8; q8 1 1 L q8; q8 . . L q8; q8 : : L q8; q8 > > L q8; q8 ` `   L q8; q8 , , L q8;
```

Figure 13: Match Instruction TM

Example

- Tape configruation when the machine starts(Figure 14)

```
# 000100`0000000011000,0000010011000,0000100101001,0000110001010,
0001000001011, 0001011001000,0100010011010,0100100101100,0100110111100,
0100000001001,0101011011001, 0101001001001,1000110111100,1000000111000,
1000100101001,1000010111000,1001011011001, 1001001001001,0110110111011,
0110010011011,0111001001011,0111011011101,0110100101101, 0110000001001,
1010111011110,1010011011111,1010001001000,1011011011001,1011001011101,
1010100101001,1100110111110,1100010011110,1101001001110,1100000001010,
1101011011001, 1100100101000,1110110111111,1110010011111,1111001001111,
1110000011000,1111011011001, 1110100101001:.010>100.011.001.001.100.011.
001.001.010.010.010.010.010.010.
```

Figure 14: Color Sort Match Instruction TM Starts

- Tape configruation when the machine halts(Figure 15)

```
# xxxyxx`xxxxxxxxyyxxx,xxxxxyxxyyxxx,xxxxyxxyxyxxy,xxxxyyxxxyxyx,
xxxyxx0001011, 0001011001000,0100010011010,0100100101100,0100110111100,
0100000001001,0101011011001, 0101001001001,1000110111100,1000000111000,
1000100101001,1000010111000,1001011011001, 1001001001001,0110110111011,
0110010011011,0111001001011,0111011011101,0110100101101, 0110000001001,
1010111011110,1010011011111,1010001001000,1011011011001,1011001011101,
1010100101001,1100110111110,1100010011110,1101001001110,1100000001010,
1101011011001, 1100100101000,1110110111111,1110010011111,1111001001111,
1110000011000,1111011011001, 1110100101001:.010>100.011.001.001.100.011.
001.001.010.010.010.010.010.010.
```

Figure 15: Color Sort Match Instruction TM Ends

3. Write And Move Turing Machine

Procedure   Start from right of the symbol `` ` ``

Step 1:   Go to right find first **0** or **1** and replace them with **x**, **y** repectively

Step 2:   Go to right find >

Step 3:   Go to right find first **0** or **1** before {.} and replace them with **x** if fetched symbol from step 1 is **0**, otherwise **y**. If cannot get **0** or **1** before {.}, do step 5

Step 4:   Go to left find the first **x** or **y** do step 1

Step 5:   If fetched symbol from step 1 is **0**, go to left to find >, replace it with {.}, then go to left again, find {.}, replace it with {>} and stop.
Else replace current symbol {.} with >, then go to left to find > and replace it with {.} and stop.

Transition Rules   (Figure 16)

```
q0 0 x R q1; q0 1 y R q2; q0 x x R q0; q0 y y R q0; q0 , , R q0; q0 . . R q0; q0 : : R q0; q0 > > R q0; q0 _ _ R q0; q0 ` ` R q0;
q1 0 0 R q1; q1 1 1 R q1; q1 , , R q1; q1 . . R q1; q1 : : R q1; q1 > > R q3; q1 x x R q1; q1 y y R q1; q1 _ _ R q1; q1 ` ` R q1;
q2 0 0 R q2; q2 1 1 R q2; q2 , , R q2; q2 . . R q2; q2 : : R q2; q2 > > R q4; q2 x x R q2; q2 y y R q2; q2 _ _ R q2; q2 ` ` R q2;
q3 x x R q3; q3 y y R q3; q3 0 x L q5; q3 1 x L q5; q3 . . L q6; q3 > > L q3; q3 : : L q3; q3 ` ` L q3; q3 _ _ R q3; q3 , , L q3;
q6 x 0 L q6; q6 y 1 L q6; q6 0 0 L q6; q6 1 1 L q6; q6 > . L q6; q6 . > L H; q6 : : L q6; q6 ` ` L q6; q6 _ _ R q6; q6 , , L q6;
q4 x x R q4; q4 y y R q4; q4 0 y L q5; q4 1 y L q5; q4 . > L q8; q4 > > L q4; q4 : : L q4; q4 ` ` L q4; q4 _ _ R q4; q4 , , L q4;
q8 x 0 L q8; q8 y 1 L q8; q8 . . L q8; q8 0 0 L q8; q8 1 1 L q8; q8 > . L H; q8 : : L q8; q8 ` ` L q8; q8 _ _ L q8; q8 , , L q8;
q5 0 0 L q5; q5 1 1 L q5; q5 x x L q5; q5 y y L q5; q5 > > L q5; q5 . . L q5; q5 : : L q7; q5 , , L q5; q5 _ _ L q5; q5 ` ` L q5;
q7 0 0 L q7; q7 1 1 L q7; q7 x x R q0; q7 y y R q0; q7 , , L q7; q7 ` ` R q0; q7 : : L q7; q7 . . L q7; q7 _ _ L q7; q7 > > L q7;
```

Figure 16: Write And Move Letter TM

Example

- Tape configruation when the machine starts(Figure 17)

```
# xxxyxx`xxxxxxxxyyxxx,xxxxxyxxyyxxx,xxxxyxxyxyxxy,xxxxyyxxxyxyx,
xxxyxx0001011, 0001011001000,0100010011010,0100100101100,0100110111100,
0100000001001,0101011011001, 0101001001001,1000110111100,1000000111000,
1000100101001,1000010111000,1001011011001, 1001001001001,0110110111011,
0110010011011,0111001001011,0111011011101,0110100101101, 0110000001001,
1010111011110,1010011011111,1010001001000,1011011011001,1011001011101,
1010100101001,1100110111110,1100010011110,1101001001110,1100000001010,
1101011011001, 1100100101000,1110110111111,1110010011111,1111001001111,
1110000011000,1111011011001, 1110100101001:.010>100.011.001.001.100.011.
001.001.010.010.010.010.010.010.
```

Figure 17: Color Sort Write And Move TM Starts

- Tape configruation when the machine halts(Figure 18)

```
# xxxyxx`xxxxxxxxyyxxx,xxxxxyxxyyxxx,xxxxyxxyxyxxy,xxxxyyxxxyxyx,
xxxyxx0001011, 0001011001000,0100010011010,0100100101100,0100110111100,
0100000001001,0101011011001, 0101001001001,1000110111100,1000000111000,
1000100101001,1000010111000,1001011011001, 1001001001001,0110110111011,
0110010011011,0111001001011,0111011011101,0110100101101, 0110000001001,
1010111011110,1010011011111,1010001001000,1011011011001,1011001011101,
1010100101001,1100110111110,1100010011110,1101001001110,1100000001010,
1101011011001, 1100100101000,1110110111111,1110010011111,1111001001111,
1110000011000,1111011011001, 1110100101001:.010>100.011.001.001.100.011.
001.001.010.010.010.010.010.010.
```

Figure 18: Color Sort Write And Move TM Ends

4. Update State Turing Machine

Procedure   Start at > or from the left first{.} of >

Step 1: Go to the left find symbol {:}

Step 2: Go to the left find the first **x** or **y**

Step 3: Go to the right find first **0** or **1** before symbol {:}, if meet {:} stop, otherwise go to right.

Step 4: go to the left if the left symbol **x** or **y**, go to the right replace **0**, **1** with **0**, **1** repectively.

Step 5: Go to left end (for simplicity)

37

Step 6: Go to right to find first **0** or **1** before and replace them with **x** if fetched symbol from step 1 is **0**, otherwise **y**.

Step 7: Go to right to find symbol `` ` ``, then do step3

Transition Rules (Figure 19)

```
q0 0 0 L q0; q0 1 1 L q0; q0 . . L q0; q0 : : L q0; q0 > > L q0; q0 x x R q1; q0 y y R q1; q0 ` ` L q0; q0 _ _ L q0; q0 , , L q0;
q1 0 x L q2; q1 1 y L q3; q1 , , R H; q1 : :  R H; q1 x x R q4; q1 y y R q1; q1 > > R q1; q1 . . R q1; q1 _ _ R q1; q1 ` ` R q1;
q2 x x L q2; q2 y y L q2; q2 , , L q2; q2 ` ` L q2; q2 0 0 L q2; q2 1 1 L q2; q2 _ _ R q4; q2 . . R q2; q2 > > R q2; q2 : : R q2;
q3 x x L q3; q3 y y L q3; q3 , , L q3; q3 ` ` L q3; q3 0 0 L q3; q3 1 1 L q3; q3 _ _ R q5; q3 . . R q3; q3 > > R q3; q3 : : R q3;
q4 0 0 R q4; q4 1 1 R q4; q4 x 0 R q6; q4 y 0 R q6; q4 ` ` R q4; q4 , , R q4; q4 . . R q4; q4 > > R q4; q4 _ _ R q4; q4 : : R q4;
q5 0 0 R q5; q5 1 1 R q5; q5 x 1 R q6; q5 y 1 R q6; q5 ` ` R q5; q5 , , R q5; q5 . . R q5; q5 > > R q5; q5 _ _ R q5; q5 : : R q5;
q6 x x R q6; q6 y y R q6; q6 ` ` R q6; q6 , , R q6; q6 0 0 L q0; q6 1 1 L q0; q6 : : L H; q6 _ _ R q6; q6 > > R q6; q6 . . R q6;
```

Figure 19: Update State TM

Example

- Tape configruation when the machine starts(Figure 20)

```
# xxxyxx`xxxxxxxxyyxxx,xxxxxyxxyyxxx,xxxxyxxyxyxxy,xxxxyyxxxyxyx,
xxxyxx0001011, 0001011001000,0100010011010,0100100101100,0100110111100,
0100000001001,0101011011001, 0101001001001,1000110111100,1000000111000,
1000100101001,1000010111000,1001011011001, 1001001001001,0110110111011,
0110010011011,0111001001011,0111011011101,0110100101101, 0110000001001,
1010111011110,1010011011111,1010001001000,1011011011001,1011001011101,
1010100101001,1100110111110,1100010011110,1101001001110,1100000001010,
1101011011001, 1100100101000,1110110111111,1110010011111,1111001001111,
1110000011000,1111011011001, 1110100101001:.010>100.011.001.001.100.011.
001.001.010.010.010.010.010.010.
```

Figure 20: Color Sort Update TM Starts

- Tape configruation when the machine halts(Figure 21)

```
# 011yxx`xxxxxxxxyyxxx,xxxxxyxxyyxxx,xxxxyxxyxyxxy,xxxxyyxxxyxyx,
xxxyxxxxxyxyy, 0001011001000,0100010011010,0100100101100,0100110111100,
0100000001001,0101011011001, 0101001001001,1000110111100,1000000111000,
1000100101001,1000010111000,1001011011001, 1001001001001,0110110111011,
0110010011011,0111001001011,0111011011101,0110100101101, 0110000001001,
1010111011110,1010011011111,1010001001000,1011011011001,1011001011101,
1010100101001,1100110111110,1100010011110,1101001001110,1100000001010,
1101011011001, 1100100101000,1110110111111,1110010011111,1111001001111,
1110000011000,1111011011001, 1110100101001:.010.100>011.001.001.100.011.
001.001.010.010.010.010.010.010.
```

Figure 21: Color Sort Update TM Ends

5. Clean markers (Figure 22)

```
# 011100`0000000011000,0000010011000,0000100101001,0000110001010,
0001000001011, 0001011001000,0100010011010,0100100101100,0100110111100,
0100000001001,0101011011001, 0101001001001,1000110111100,1000000111000,
1000100101001,1000010111000,1001011011001, 1001001001001,0110110111011,
0110010011011,0111001001011,0111011011101,0110100101101, 0110000001001,
1010111011110,1010011011111,1010001001000,1011011011001,1011001011101,
1010100101001,1100110111110,1100010011110,1101001001110,1100000001010,
1101011011001, 1100100101000,1110110111111,1110010011111,1111001001111,
1110000011000,1111011011001, 1110100101001:.010.100>011.001.001.100.011.
001.001.010.010.010.010.010.010.
```

Figure 22: Final Tape

These four examples show how a Universal Turing Machine simulates for another Turing Machine for its one step Computation. At the beginning, we have initial state $q0$, reading letter 2, write letter $*$, moving

39

right, enter state $q3$. Before the Universal Turing Machine runs, on the left of its tape, we have state $q0$ represented by 000. When it begins running, the arrow $>$ points 100 which represents 2 according to its encoded code {100 : 2}. After simulating one step of that Turing Machine, . The arrow $>$ moved to the next {.} and points 011 which represents 0 here(Note the input is 20112011). Now, look at the left 3 letters of the symbol $>$, we have 000 for $*$ in the original Turing Machine. Now, left tape ends up with 011100, the first 3 letters represent the original Turing Machine state, also according to its encoded code, it represents $q3$({011 : $q3$}). This is exactly what we want for the transition rule $q0, 2, *, R, q3$

6. Creating UTM with 4 TMs above

Procedure With the four Turing Machines we showed above. Arrow starts at the left most position.

Step 1: Run Fetch Letter Turing Machine with current Turing Machine configuration.

Step 2: Run Match Instruction Turing machine with the end configuration by running Fetch Letter Turing Machine. If this machine stops at its own step 4, then the Universal Turing Machine stops. Otherwise, do next step.

Step 3: Run Write And Move Turing Machine with the end configuration by running Match Instruction Turing machine.

Step 4: Run State Update Turing machine with the end configuration by running Write And Move Turing Machine.

Step 5: Reset every x, y to 0, 1, then go to the left most symbol, pass end configuration to Fetch Letter Turing Machine, do step 1.

40

Transition Rules (Figure 23 and Figure 24)

```
q1 > > R q1; q1 x x R q1; q1 y y R q1; q1 1 1 R q1; q1 0 0 R q1; q1 . . L q2; q1 ` ` R q1; q1 , , R q1; q1 : : R q1; q1 _ _ R q1;
q2 0 x L q3; q2 1 y L q4; q2 x x L q2; q2 y y L q2; q2 > > R q7; q2 ` ` L q2; q2 , , R q2; q2 . . R q2; q2 : : L q2; q2 _ _ R q2;
q3 0 0 L q3; q3 1 1 L q3; q3 , , L q3; q3 ` ` L q5; q3 > > L q3; q3 . . L q3; q3 x x L q3; q3 y y L q3; q3 : : L q3; q3 _ _ R q3;
q4 0 0 L q4; q4 1 1 L q4; q4 , , L q4; q4 ` ` L q6; q4 > > L q4; q4 . . L q4; q4 x x L q4; q4 y y L q4; q4 : : L q4; q4 _ _ R q4;
q5 0 x R q0; q5 1 x R q0; q5 x x L q5; q5 y y L q5; q5 > > R q5; q5 . . R q5; q5 , , R q5; q5 ` ` R q5; q5 : : R q5; q5 _ _ R q5;
q6 0 y R q0; q6 1 y R q0; q6 x x L q6; q6 y y L q6; q6 > > R q6; q6 . . R q6; q6 , , R q6; q6 ` ` R q6; q6 : : R q6; q6 _ _ R q6;
q7 x 0 R q7; q7 y 1 R q7; q7 0 0 R q7; q7 1 1 R q7; q7 . . L q8; q7 , , R q7; q7 ` ` R q7; q7 > > R q7; q7 : : R q7; q7 _ _ R q7;
q8 x 0 L q8; q8 y 1 L q8; q8 0 0 L q8; q8 1 1 L q8; q8 ` ` L q8; q8 , , L q8; q8 . . L q8; q8 > > L q8; q8 : : L q8; q8 _ _ R q9;
```

Figure 23: UTM Part 1

```
qi 0 x R qj; qi 1 y R qk; qi x x R qi; qi y y R qi; qi , , R qi; qi . . R qi; qi : : R qi; qi > > R qi; qi _ _ R qi; qi ` ` R qi;
qj 0 0 R qj; qj 1 1 R qj; qj , , R qj; qj . . R qj; qj : : R qj; qj > > R ql; qj x x R qj; qj y y R qj; qj _ _ R qj; qj ` ` R qj;
qk 0 0 R qk; qk 1 1 R qk; qk , , R qk; qk . . R qk; qk : : R qk; qk > > R qm; qk x x R qk; qk y y R qk; qk _ _ R qk; qk ` ` R qk;
ql x x R ql; ql y y R ql; ql 0 x L qn; ql 1 x L qn; ql . . L qo; ql > > L ql; ql : : L ql; ql ` ` L ql; ql _ _ R ql; ql , , L ql;
qo x 0 L qo; qo y 1 L qo; qo 0 0 L qo; qo 1 1 L qo; qo > . L qo; qo . > L qr; qo : : L qo; qo ` ` L qo; qo _ _ R qo; qo , , L qo;
qm x x R qm; qm y y R qm; qm 0 y L qn; qm 1 y L qn; qm . > L qr; qm : : L qm; qm ` ` L qm; qm _ _ R qm; qm , , L qm;
qq x 0 L qq; qq y 1 L qq; qq . . L qq; qq 0 0 L qq; qq 1 1 L qq; qq > . L qr; qq : : L qq; qq ` ` L qq; qq _ _ L qq; qq , , L qq;
qn 0 0 L qn; qn 1 1 L qn; qn x x L qn; qn y y L qn; qn > > L qn; qn . . L qn; qn : : L qp; qn , , L qn; qn _ _ L qn; qn ` ` L qn;
qp 0 0 L qp; qp 1 1 L qp; qp x x R qi; qp y y R qi; qp , , L qp; qp ` ` R qi; qp : : L qp; qp . . L qp; qp > > L qp;

qr 0 0 L qr; qr 1 1 L qr; qr . . L qr; qr : : L qr; qr > > L qr; qr x x R qs; qr y y R qs; qr ` ` L qr; qr _ _ L qr; qr , , L qr;
qs 0 x L qt; qs 1 y L qu; qs , , R qy; qs : : R qy; qs x x R qv; qs y y R qs; qs > > R qs; qs . . R qs; qs _ _ R qs; qs ` ` R qs;
qt x x L qt; qt y y L qt; qt , , L qt; qt ` ` L qt; qt 0 0 L qt; qt 1 1 L qt; qt _ _ R qv; qt . . R qt; qt > > R qt; qt : : R qt;
qu x x L qu; qu y y L qu; qu , , L qu; qu ` ` L qu; qu 0 0 L qu; qu 1 1 L qu; qu _ _ R qw; qu . . R qu; qu > > R qu; qu : : R qu;
qv 0 0 R qv; qv 1 1 R qv; qv x 0 R qx; qv y 0 R qx; qv ` ` R qv; qv , , R qv; qv . . R qv; qv > > R qv; qv _ _ R qv; qv : : R qv;
qw 0 0 R qw; qw 1 1 R qw; qw x 1 R qx; qw y 1 R qx; qw ` ` R qw; qw , , R qw; qw . . R qw; qw > > R qw; qw _ _ R qw; qw : : R qw;
qx x x R qx; qx y y R qx; qx ` ` R qx; qx , , R qx; qx 0 0 L qr; qx 1 1 L qr; qx : : L qr; qx _ _ R qx; qx > > R qx; qx . . R qx;

qy 0 0 L qy; qy 1 1 L qy; qy x 0 L qy; qy y 1 L qy; qy , , L qy; qy . . L qy; qy : : L qy; qy ` ` L qy; qy > > L qy; qy _ _ R q0;
```

Figure 24: UTM Part 2

4.4 Usage and Example

Mathematicians and computer scientists usually describe a Universal Turing Machine in the following format.

$$\langle M_{utm}\rangle(\langle \hat{M}_{tm}\rangle, \hat{x}) = \langle M_{tm}\rangle(x)$$

In this case, users need to encode $\langle M_{tm}\rangle$ and its input x, then run these two inputs on $\langle M_{utm}\rangle$. The program has implemented a subroutine for users to encode inputs. Users only need to provide a Turing Machine transition rules and its input for this UTM.

Through terminal, run **python utm.py <Your transition file> <Your input for your Turing Machine>**. For example,

python utm.py color_sort.tm '>20112011'

The quote for input for the input is required, since ">" is a keyword in most shells, such as bash, and zsh. '>' is used to specify the arrow's location on a Turing Machine. If users would like to have some space on both sides of the input string add "_" to them, like this "_ _ _ _ > 20112011 _ _ _ _" This

UTM program by default does not provide an animation for simulating a Turing Machine program(If a user really wants to see an animation process, he can use the encoder provided in this project, then get the encoded string and run command **view.py** utm.tm through terminal, next copy this string into GUI input box), since it will take too long to show this whole process. After the UTM program terminated, it will provide the information as below.

UTM Result Information   (Figure 25)

```
-----------------tm-info-----------------
tm-input tape:
20112011
tm-output tape:
00111122
tm-output steps:
45
-----------------utm-info-----------------
encoding map: {'*': '000', '1': '001', '_': '010', '0': '011', '2': '100', '#': '101', 'q0': '000', 'H': '001', 'q1': '010', 'q3': '011', 'q2': '100', 'q4'
: '101', 'q5': '110', 'q6': '111'}
reverse state map: {'000': 'q0', '001': 'H', '010': 'q1', '011': 'q3', '100': 'q2', '101': 'q4', '110': 'q5', '111': 'q6'}
reverse letter map: {'000': '*', '001': '1', '010': '_', '011': '0', '100': '2', '101': '#'}
utm-input tape:
000000`0000000011000,0000010011000,0000100101001,0000110000010,0001000001011,0001011001000,0100010010010,0100100101100,0100110111100,0100000001001,01010110
11001,0101001001001,1000110111100,1000000111000,1000100101001,1000010111000,1001011011001,1001001001001,0110110111011,0110010011011,0111001001011,011101101
0101,0110100100101,0110000001001,1010111010110,1010011010111,1010001001000,1011011011001,1011001010101,1010100100001,1100110110110,1100010010110,1101001000
110,1100000000010,1101011011001,1100100101000,1110110110111,1110010010111,1111001000111,1110000011000,1111011011001,1110100101001:.010>100.011.001.001.100.
011.001.001.010.010.010.010.010.010.010.010.010.010.010.
utm-output tape:
x01010`xxxxxxxxyyxxx,xxxxxxxxyyxxx,xxxxyyxxxxyxx,xxxxyyxxxxxyx,xxxxyxxxxxxyy,xxxxyyyxxyxxx,xyxxxxyxxyxxx,xyxxyxxyyxxx,xyxxyyyyxxx,xxxxxxxxyxxy,xyxyxxyyx
yyxxy,xyxyxxyxxyxxy,yxxxyyxyyyyxx,yxxxxxxyyxxx,yxxxxyxxyxxy,yxxxxyyyyxxx,yxxyxyyxyyxxy,yxxyxxxxyxxxy,xyyxyxxyyyxy,xyyxxyyxyxxy,xyyyxyyyyxyy,xyyyxxxyyxxy,xyyyxyyyx
xyxy,xyyxxyxxxyxy,xyyxxxxxxyxxy,yxyxyyyxyxyxx,yxyxyyxyyxyyy,yxyxxxxyxxxx,yyxyxyyxyyxxy,yyxyyxyyyyxy,yxyxyxxyxyxy,xyxyxxyxxxxxy,yyxxyyyyyyxyx,yyxxxxyxxyyx,yyxyxxyxxx
yyx,yyxxxxxxxxyx,yyxyxyyyxyxxy,yyxxxyyxyyyy,yyyyxyxyyyy,yyyyxxyxxxyyy,yyyyxxxxyyxxx,yyyyxyyyyxxxy,yyyxxxyxyxxxy,yyyyxxyxxyxxy:.010.011.011.001.001.001.
001.100.100.010>010.010.010.010.010.010.010.010.010.
utm-output steps:
1200143
```

Figure 25: UTM Info

The steps of running the original Turing Machine and the Universal Turing Machine are provided to show how much efforts it takes for UTM to simulate a Turing machine given by specific inputs. Also, letter map and state map are provided in reverse direction of our encoding system, to help to check if the UTM output can interpret the original Turing Machine's terminated configuration. In UTM output tape, everything after {:} represents what left on the original Turing Machine when it halts. Moreover, the most left three letters on UTM tape represent the final state. According to the letter map and state map provided above, after translating it back, the result is the same as the one being simulated. For tape, we have 011.011.001.001.001.001.100.100.010 > which can map to 00111122 and arrow ended up at the right of the blank cell. For the state, we have $x01$ interpreted as 001, which presents H in the original Turing Machine.

5   Observation

A Turing machine is an abstract computer.It use a tape which works like memory to store inputs and outputs. Also, it contains a set of transition functions which is the same concept as CPU instruction set. A Turing machine perform its action based on its transition functions, and a computer does its tasks based on its instructions from a operating system and data in memory.

In our example, it takes **1200143** steps for our

Universal Turing Machine to get the same job done by the original Turing Machine with 45 steps. Formally, consider a TM with $|\Gamma| = n$ and $|Q| = m$ given arbitary input with the length of k. In our UTM, the tape length for this machine will be $logn + logm + 1 + lognlogm(2logmn + 2) + 2 + (k+1)logn$, since a Turing Machine Program P description is fixed, we can take k as the length for this UTM. Also, stages for Match Instruction and Update State never go through input description part on UTM tape, we take them out of our consideration. Then, stage for Restore, Fetch, Write take $O(k)$, $O(log(n)k)$, $O(log(n)k)$. Overall, it takes $O(k)$ for this UTM to simulate one step computation of a Turing Machine since $logn$ is fixed. If it takes time t for a TM being simulated to finish its computation task, then it takes $O(tk)$ for UTM to complete the simulation. And This different computing cost can be a good explanation of why Interpreting languages(code runs on a virtual machine) such as **python**, **javascript**, **php** are cross-platform and slower, while executable files from compling languages(complied code runs on a operating system) such as **c**, **c++**, **go** do not run on different operating systems, but faster.

Note *Java* is also a Interpreting language from this point of view. Next, we compare $C++$ and *Java* to illustate the difference. First, consider the process of a $c++$ program compilation.

- **Processing**:
  Copy the content from header files to the source file, replace constants with customized symbols defined by key word $\#define$ in the source file

- **Compiling**: ComplieD code in the source file into assembler code.

- **Assembling**: Assemble the code into the object code for a specific

43

platform

- **Linking**: Object code file is used to produce an executable file.

Notice all OS are different, so the compiled code is different. Moreover, each OS has different CPU instruction set architecture to interpret the final binaries, and the same binaries may have different meanings on different instruction set architecture, as it for different Turing Machines–faster but not portable.

With java, things are different. Its virtual machine–JVM plays the role of such OS and hardware abstraction. JVM is not a library. Instead, it is a whole platform working on top of native OS, and its implementation is different for different OS. A Java compiler does not compile a Java program into CPU instructions. It compiles a program to the instructions of the virtual machine, byte-code. Also, the virtual machine interprets byte-code instructions as native code and executes it, in the sense that a Univeral Turing machine interprets a Turing Machine's description and its input(byte-code) as the Turing Machine runs for its input(native code)–portable yet slower.

6   Conclusion and future work

In this thesis, a Turing Machine Simulator was built to serve a visual educational aid for computability theory teaching and learning. Also a Universal Turing Machine was created to present the core idea about how a Turing Machine can simulate another Turing Machine. It shows how a computer and algorithm work in the aspect of the computation model. Also, it uses the difference in computing performance between $Java$ and $C++$ as an example, explains why a compiling programming language is generally faster than an interpreting language. Computer science origins from these computation models we discussed. Learning computation models is an essential step for building up an architecture of knowledge for computer science. Looking at computer science with "computing" in mind, not only helps students find the solution when they encounter some problem but also builds their muscle to seek the 'solution.' for creating solutions –"Meta-Solution". This thesis presents a simulator for the original Alan Turing's machine to show how a single tape Turing Machine works. Multi-tapes Turing Machine simulator can be built to show how multi-tapes Turing The machine completes a computing task. How it is different from a single tape Turing Machine. Also, a Non-deterministic Turing Machine is beneficial for the understanding of time complexity for an algorithm. Think about one of the most critical questions in computer science–$P = NP$?. NP is a set of questions that are defined by a non-deterministic Turing Machine. Presenting how different Turing Machine variants yield the same result for the problem in a visual animation manner can be very helpful for students to gain an understanding of why these models are equivalent. Turing Machine is a milestone of computability theory, but now all of it. A visual tool for deterministic finite automata(DFA), a Regular Expression – DFA converter, a Lambda interpreter, and Lambda – Turing Machine converter, all have a significant meaning for a computability theory class.

References

[1] Wilhelm Ackermann and David Hilbert. Grundzüge der theoretischen logik. *Berlin, Springer*, 1037(23):4, 1928.

[2] Otto Blumenthal. David hilbert. *Naturwissenschaften*, 10(4):67–72, 1922.

[3] George S Boolos, John P Burgess, and Richard C Jeffrey. *Computability and logic*. Cambridge university press, 2002.

[4] Jonathan P Bowen. The impact of alan turing: Formal methods and beyond. In *International Summer School on Engineering Trustworthy Software Systems*, pages 202–235. Springer, 2018.

[5] Alonzo Church. An unsolvable problem of elementary number theory. *American journal of mathematics*, 58(2):345–363, 1936.

[6] Herbert B Enderton. A mathematical introduction to logic. san diego (calif.)[etc.], 2002.

[7] Kurt Gödel. Die vollständigkeit der axiome des logischen funktionenkalküls. *Monatshefte für Mathematik*, 37(1):349–360, 1930.

[8] Kurt Gödel. Zur intuitionistischen arithmetik und zahlentheorie. *Ergebnisse eines mathematischen Kolloquiums*, 4(1933):34–38, 1933.

[9] J Roger Hindley and Jonathan P Seldin. *Lambda-calculus and Combinators, an Introduction*, volume 13. Cambridge University Press Cambridge, 2008.

[10] Stephen Cole Kleene. A theory of positive integers in formal logic. part i. *American journal of mathematics*, 57(1):153–173, 1935.

[11] Dexter C Kozen. *Automata and computability*. Springer Science ∧ Business Media, 2012.

[12] Andrei Markov. Impossibility of certain algorithms in the theory of associative systems. *Journal of Symbolic Logic*, 13:52–53, 1948.

[13] Soltys-kulinicz Michael. *Introduction To The Analysis Of Algorithms, An*. World Scientific, 2018.

[14] Emil L Post. Finite combinatory processes—formulation 1. *The Journal of Symbolic Logic*, 1(3):103–105, 1936.

[15] Michael Sipser et al. *Introduction to the Theory of Computation*, volume 2. Thomson Course Technology Boston, 2006.

[16] Alan Mathison Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London mathematical society*, 2(1):230–265, 1937.