Turing Machines

Eugen Iordache

January 2018

“ I am convinced more and more of the utility and reality of this general science, and I see that very few people have understood its extent. . . . This characteristic consists of a certain script or language . . . that perfectly represents the relationships between our thoughts. The characters would be quite different from what has been imagined up to now. Because one has forgotten the principle that the characters of this script should serve invention and judgment as in algebra and arithmetic. This script will have great advantages; among others, there is one that seems particularly important to me. This is that it will be impossible to write, using these characters, chimerical notions (chim‘eres) such as suggest themselves to us. An ignoramus will not be able to use it, or, in striving to do so, he himself will become erudite.”

G.W. v. Leibniz

Abstract

As it happens to everyone interested in Computer Science, it also happened to me to hear before about Turing Machines. And as the question “What is a Turing Machine?” appeared, the answer was always, inevitably, “The first model of a computer, with a tape and a head that moves along the tape to write numbers on it.” Practically, this is a Turing Machine, but explained this way it doesn’t sound as something impressive at all.

In this paper I will try to go beyond the practical implementation of a Turing Machine and describe its implications in the theory of computation.

The two main books I used as documentation were “The road from Leibniz to Turing” - Martin Davis and “Elements of the Theory of Computation” - Harry R. Lewis

I always put high importance on where exactly some concepts came to life from and what necessity of the human being lead to their creation. So,“The road...” provided great support in this sense since it describes the entire journey of some important ideas along the centuries.

The other book ”Elements ..” provided all the needed information, the general structure of this paper, definitions and examples.

1 From Leibniz to Turing

In this preliminary section I will present the historical evolution that leaded to Turing machines. I will briefly introduce a few personalities which brought contributions to the creation of Turing machines, together with the notations they introduced.

Leibniz is primarily known for the notation he developed for the differential and integral calculus. The importance of this in computation is universally recognized. It transformed a very meticulous computation into a very simple one. This discovery is what inspired Leibniz to think about the possibility of an entire language consisting of such notations which would reflect the entire human knowledge.

He also created a machine called "Stepped Reckoner" which was capable of addition, subtraction, multiplication and division. His dream was to create a machine which given basic axioms would be able to give answer to any question by applying logical reasoning.

Unfortunately, from financial reasons, he had to perform duties unrelated to his interests(He was engaged to write the history of the house of Brunswick to which George I, the king of England, belonged).

He introduced the following notations:

$$B \oplus N = N \oplus B$$
$$A \oplus A = A$$

George Boole was Irish and mainly self taught in the field of mathematics. Despite his achievements in this field he wasn't very recognized academically. His most important contribution was to demonstrate that logical deduction could be seen as branch of mathematics.

He introduced 0 and 1 for the $false$ and the *truth*, and formalised Aristotle's logic under the following form:

All X are Y.
No X are Y.
Some X are Y.
Some X are no Y.

Gottlob Frege believed mathematic is nothing but logic and expanded Boole's logic by introducing the following notations:

$\neg$	*not...*
$\vee$	*...or...*
$\wedge$	*...and...*
$\supset$	$if..., then...$
$\forall$	*every*
$\exists$	*some*

Thanks to his notations now the following sentence could be expressed:

"All failing students are either stupid or lazy"

Georg Cantor created the mathematical theory of the infinite and invented set theory.

2 The beginings

In 1936 Alan Turing published the paper "On Computable Numbers, with an Application to the Entscheidungsproblem" and defined for the first time Turing Machines. The "Entscheidungsproblem" ("Decision problem"), proposed by David Hilbert, asked whether there is an algorithm that takes as input a first-order logic statement and decides if it's provable from the axioms using the rules of logic. Turing interpreted "provable" as "computable" and gave his own definition to the term.

" *The "computable" numbers may be described briefly as the real numbers whose expressions as a decimal are calculable by finite means. According to my definition, a number is computable if its decimal can be written down by a machine.*"

3 Constructing a Turing Machine

Before offering a formal definition I will informally describe Turin Machines the way they were first described by Alan Turing (and gradually I will introduce the specific notations).
He compared a machine with a man in the process of computing a number. The machine will be constructed with included instructions which will simulate the reasoning applied by a person to compute a number. These instructions will be represented by a series of conditions which will be applied step by step. Let's notate them $q_1, q_2, q_3...q_K$. As the human memory is limited, the number of condition will be considered finite. In order to achieve it's purpose, the machine will be provided with a tape and a control unit which will apply the instructions.
The tape will have a left end, will be infinitely long to the right and will be divided into sections called squares.

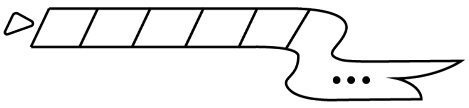


Figure 1: The tape

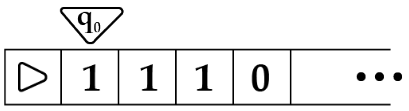
The machine will use the tape for its computation, but at each step it will only be aware of one square on the tape (as a person couldn't be aware of two things at the same time).

Figure 2: A person aware of two things

Let's illustrate what we described with a practical example.

So, we have:

- a tape
- tape squares (which may or may not contain a symbol)
- a control unit
- a set of conditions



Current state	Current state	Next state	The action taken
q_0	1	q_1	0
q_0	0	h	
q_1	0	q_0	$\longrightarrow$

Figure 3: First machine

The machine will be provided with a set of symbols to work with. In this case it takes as input the numbers $1, 1, 1, 0$. For simplicity the symbols were chosen from the set $\{0, 1\}$.
As you will notice, the first square of the tape is occupied by the special symbol$\triangleright$. We use it to mark the left end of the tape. The control unit will start from the second square and will be in the state q_0.
The table with the conditions has to be analyzed at each step as follows:

- First, identify the current state (1^{st} column) and the scanned symbol (2^{nd} column).
- Than, depending on the configuration of the first two columns, we proceed to the next ones. The 3^{rd} column specifies the state the machine will enter at the next step. The 4^{th} represents the main action of the machine. It can either write a symbol or move to the left ($\leftarrow$) or to the right ($\rightarrow$).

Now, let's analyze the behavior of our promising little machine and see what it does.

First Step:
The machine starts in the state q_0 and scans a "1". By identifying these two variables, now it "knows" that it has to follow the instruction from the first line. So, according to the 4^{th} column the machine will write a "0", and, as 3^{rd} column says, it will enter the state q_1.

Second Step:
Note that the machine didn't received the instruction to move its head. That's why

the scanned symbol it's the one written by the machine itself. Now the scanned symbol is "0" and the machine it's in the state q_1. According to the table, the machine will move to the right and will go back to state q_0.

For the 3rd step we notice that the machine would repeat the first step, and in the same way at the 4th step it would repeat the 2nd. In fact, it will repeat these two steps as long as there's a consecutive sequence of "1" on the tape. The change comes when a "0" appears. The machine will go to the right, and unlike so far it will scan a "0". So, it will execute the instructions from the second line of the table and will enter state "h", which stands for "halt". This is the moment the machine will finish the computation. The machine erased the entire sequence of "1", and so will do for any sequence of "1" given as input, no matter how long.

4 The formal definition of a Turing Machine

As we saw, a few components of the machine are essential every time we construct one.
Proceeding to the formal definition, we say that a Turing Machine is a quintuple $(K, \Sigma, \delta, s, H)$, where:

- K is the finite set of states $q_0, q_1, q_2, q_3 ... q_K$
- Σ will be the alphabet, the set of symbols given as input, the blank symbol $\sqcup$ and the left end symbol $\triangleright$ (but it will not contain the specific symbols for moving left $\leftarrow$ or right $\rightarrow$)
- s is the initial state
- $H \subset K$ is the set of halting states (the ones which announce that the computation is finished)
- δ is the transition function, a function from $(K \setminus H) \times \Sigma$ to $K \times (\Sigma \cup \{\leftarrow, \rightarrow\})$ (practically a function from $(1^{st} column \times 2^{nd} column)$ to $(3^{rd} column \times 4^{th} column)$ of the table we saw before)
  $eg : \delta(q_0, 1) = (q_1, 0)$

A few more observations about Turing Machines:

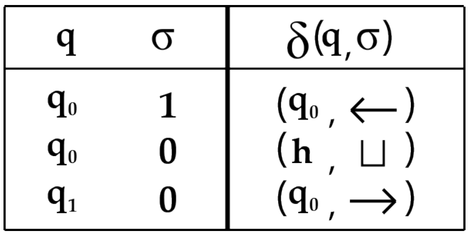
- At each step the control unit changes its state and can either write a symbol or move one square to the left or tho the right
- The tape contains three main components
  - the special symbol $\triangleright$ from the first square. It will always act as a "protective barrier" and will tell the machine to move one square to the right
  - the input with symbols from the alphabet Σ
  - the blank symbol $\sqcup$ which will fill the rest of the tape
- the machine needs to have at least one state h which will tell that the computation is finished

5 Looping Turing Machines

Now, let's observe something interesting regarding Turing Machines. Let's consider the Turing Machine $M = (K, \Sigma, \delta, s, H)$ where

$$K = q_0, h,$$
$$\Sigma = \{a, \sqcup, \triangleright\},$$
$$s = q_0,$$
$$H = \{h\}$$

and the δ given bt the following table:



q	σ	δ(q,σ)
q_0	1	$(q_0, \leftarrow)$
q_0	0	$(h, \sqcup)$
q_1	0	$(q_0, \rightarrow)$

Figure 4: Looping machine

The machine scans consecutive sequences of a to the left until it finds an $\sqcup$. When the $\sqcup$ is found the machine halts. But what would happen if no $\sqcup$ is found and it reaches the left end? The scanned symbol would be $\triangleright$ and machine would move to the right. Than, it would scan again an a and will go to the left. This is the moment when the loop begins, and we can tell that the machine will never reach its halting state. So, there are Turing Machines which may never stop.

6 Let's improve our notations

As you probably noticed so far, our current notations asks us to interpret quite complicated tables of extremely simple machines. So, a more graphic and easy to interpret notation is needed.

We will use the fact that simple machines can be combined to create more complex ones. So we will give notations to these simple machines and combine them in a convenient way. For example a machines that move the head or one that simply writes a symbol. To combine them we will use arrows.

Let's start by giving notations to three of them. These machines will finish their computation in one single step. The set of states K will contain only a state s and the halting state h.

For each $a \in \Sigma \cup \{\rightarrow, \leftarrow\} \setminus \{\triangleright\}$, we define a Turing machine $M_a = (\{s, h\}, \Sigma, \delta, s, \{h\})$, where for each $b \in \Sigma \setminus \{\triangleright\}$, $\delta(s, b) = (h, a)$. Now, there are two case:

CASE I: $a \in \Sigma \setminus \{\triangleright\}$

The machine will write a symbol a on the current square of the tape. We will simply notate this machine as a.

CASE II: $a \in \{\leftarrow, \rightarrow\}$

In this case the machine will move to the left or to the right. And we will notate two more machines: L, for the machines that moves to the left and R for the machine that moves to the right. So,

$$M_a = a$$

, for a being a symbol;

$$M_a = L$$

, for a being $\leftarrow$;

$$M_a = R$$

, for a being $\rightarrow$.

7 Combining Turing machines

Now that we have these three machines, we can look at how we can combine time. For example, let's combine two of them and create a machine that moves to the right and writes a symbol.

It would look as follows:

$$\begin{tikzcd}
\mathsf{R} \arrow[r] & \mathsf{a}
\end{tikzcd}$$

Figure 5: More complex machine

The arrow says that **only after the machine R halted** we can pursue to the machine a. Let's say for example that R receives a blank tape with the head positioned on the second square. It would move one square to the right and would write the symbol.

To equip our machines with even more possibilities we will also write symbols on the arrow. This will say that only if the scanned symbol is the one written on the arrow, we will proceed to the next machine. By adding this notation, we can give to the machine the possibility to choose the next one depending on the currently scanned symbol. For example:

$$\begin{tikzcd}
\mathsf{R} \arrow[r, "1"] \arrow[dr, "0"] & 0 \\
& 1
\end{tikzcd}$$

Figure 6:

This machine moves to the right and if the scanned symbol is 0 it will write an 1 and if the scanned symbol is 1 it will write a 0.

$$\begin{tikzcd} R \arrow[loop above, "1"] \arrow[r, "0"] & 1 \end{tikzcd}$$

Figure 7:

In addition, with this notation we can tell the machine to point back to itself, without creating an infinite loop. For example:

This machine will go to the right as long there is a consecutive sequence of 1 and when it finds a 0 it writes another 1 at the end;

4 MORE USEFUL MACHINES TO NOTATE

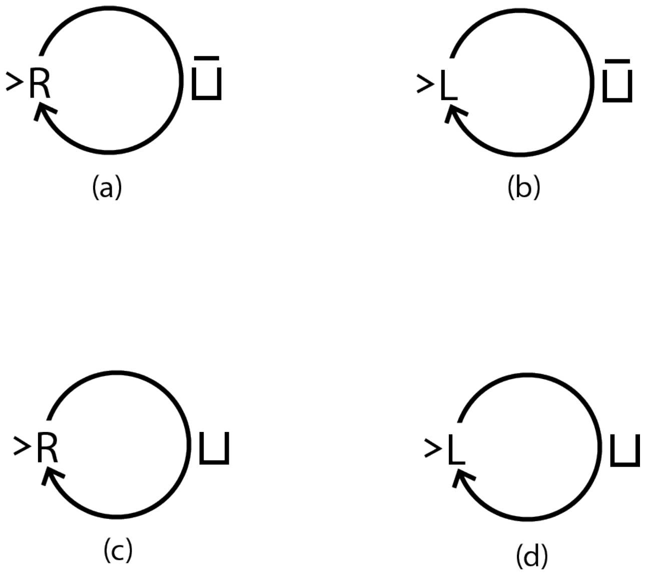


Figure 8:

- (a) which will be notated as $R_{\sqcup}$ is the machine that scans to the right until it finds the first blank space
- (b) notated as $L_{\sqcup}$, finds the first blank square to the left
- (c) notated as $R_{\overline{\sqcup}}$, finds the first nonblank square to the right
- (d) notated as $L_{\overline{\sqcup}}$, finds the first nonblank square to the left

The copying machine C

$$\begin{tikzcd}
>L_{\sqcup} \arrow[r] & R \arrow[r, "a \neq \sqcup"] \arrow[d, "\sqcup"] & \sqcup R^{2}_{\sqcup} a L^{2}_{\sqcup} a \arrow[l, bend right] \\
& R_{\sqcup} &
\end{tikzcd}$$

Figure 9: Copying machine

The tape will contain $\sqcup w \sqcup$, where $w \in \Sigma^*$ is the input.
How does it work?
(I) It starts by finding the blank symbol at the left of the input
(II) Than, it moves to the right and scans the first symbol, replacing it by a $\sqcup$
(III) Now, it moves to the right, finding the second blank after the input and writes the previously scanned symbol
(IV) After the first symbol was copied, it goes back to the second blank symbol to the left, writes back the erased symbol and proceed to the next one
At this moment we can anticipate that the machine will loop through the steps $(II), (III), and (IV)$ until all the symbols will be copied to the right of the input.

The right-shifting machine $S_{\rightarrow}$

$$\begin{tikzcd}
>L \arrow[r, "a \neq \sqcup"] \arrow[d, "\sqcup"] & \sqcup R_{\sqcup} a L_{\sqcup} a \arrow[l, bend right] \\
R_{\sqcup} &
\end{tikzcd}$$

Figure 10: Right-shifting machine

This machine transforms $\sqcup w \sqcup$ into $\sqcup \sqcup w \sqcup$.

8 Language acceptors and functions

8.1 Pre-notations

Before starting this section we need to fix the notation for the reflexive transitive closure and explain what it means.

The reflexive closure of a binary relation R over some finite set A is defined as $R \cup \Delta \cup tr(R)$, where tr(R) = {(a, b): there is a non-trivial path in R from a to b} is the transitive closure of R.

The following is a graph representation of the reflexive transitive closure:

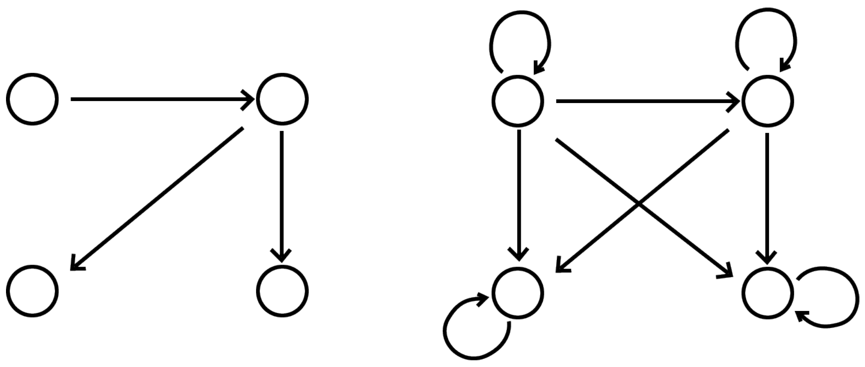


Figure 11: Graph representation

An alphabet Σ is a **finite set** of symbols. For example the Roman alphabet {a, b, ... z} or the alphabet 0,1 which is particularly important for computation. A string over an alphabet is a finite sequence of symbols from the alphabet. For example the string *"ttourrrrringggtouriing"*. We will notate the set of all strings as Σ^*.

To notate a configuration of a Turing machine we will have to specify the current state, the content of the tape anf the position of the head. This configuration will be a member of $K \times \triangleright\Sigma^* \times (\Sigma^*(\Sigma \setminus \{\sqcup\} \cup \{e\}))$ where e is the right end of the input.

For example the configuration of a machine with the current state $q3$, the string $aaba$ as content of the tape, and the head positioned on the second a will be written as $(q_3, \triangleright a\underline{a}ba)$. If at the next step the configuration is $(q_4, \triangleright a\underline{b}ba)$ we write $(q_3, \triangleright a\underline{a}ba) \vdash_M (q_4, \triangleright a\underline{b}ba)$.

This way, a computation of the machine M can be written as $C_0 \vdash_M C_1 \vdash_M ... \vdash_M C_K$.

We will notate $\vdash_M^*$ the reflexive transitive closure of $\vdash_M$. We say that a configuration C_1 yields a configuration C_2 if $C_1 \vdash_M^* C_2$.

8.2 Conventions

In this section we will ask our machines to perform some quite serious computational tasks. It's time, therefore, to fix some conventions for the use of Turing Machines:

- The input string will not contain the blank symbol $\sqcup$ and will be written to the right of the left end symbol with a blank to its left and blanks to its right.
- The head will be positioned on the blank symbol between $\triangleright$ and the input. So, with an input $w \in (\Sigma \setminus \{\sqcup, \triangleright\})^*$, the initial configuration is $(s, \triangleright\underline{\sqcup}w)$.

8.3 Turing machines as language recognizers

The task a machine will have to perform in this case will be to recognize a language. So, given a string w as input, the machine $M = (K, \Sigma, \delta, s, H)$ will have to be able to provide two different answers: y, if the language is accepted and n if it rejects it. In this case H will contain two distinguish halting states: $H = \{y, n\}$. These machines will also operate with an input alphabet $\Sigma_0 \subseteq \Sigma \setminus \{\sqcup, \triangleright\}$. Fixing it as a subset we allow our machines to use extra symbols beside those given as input.

We say that M **decides** a language $L \subseteq \Sigma_0^*$ if for any string $w \in \Sigma_0^*$, M halts and accepts w if $w \in L$ or rejects w if $w \notin L$. In addition, if a language L is decided by a Turing machine we call it **recursive**.

Let's consider the language $L = \{a^n b^n c^n : n \geq 0\}$. The following machine will decide L:

$$\begin{tikzcd}
R \arrow[loop above, "d"] \arrow[r, "a"] \arrow[d, "\sqcup"'] \arrow[dr, "{b,c}"] & dR \arrow[loop above, "{a,d}"] \arrow[r, "b"] \arrow[d, "{c,\sqcup}"] & dR \arrow[loop above, "{b,d}"] \arrow[r, "c"] \arrow[dl, "{a,\sqcup}"] & dL_{\sqcup} \arrow[lll, bend right=40] \\
y & n & &
\end{tikzcd}$$

Figure 12:

The machine will use the additional symbol d to sequentially replace an a, a b and a c.

It starts from the left end and moves to the right looking for an a, ignoring all letters d which may appear. When an a is found, it's replaced by a d and the head continues to move to the right, looking for a b. (this time ignoring all letters d and a which may appear. When b is found, the same will happen with the c: the machine will look for it, ignoring b and d. At any moment, if another symbol excepting the ignored ones or the searched one is found, the machine will immediately halt on state n. The situation may occur when: the input contains the symbols $\{a, b, c\}$ in any other order, or if their number of apparitions differ.

8.4 Recursive functions

As we saw in the previous example, an important feature of the Turing machines is that they are able to write on their tapes. Let's expand this advantage and let our machines compute tape's content in the same way functions do.

We have a Turing machine $M = (K, \Sigma, \delta, s, \{h\})$, an input alphabet $\Sigma_0 \subseteq \Sigma \backslash \{\sqcup, \triangleright\}$ and the string $w \in \Sigma_0^*$. This machine will halt and $(s, \triangleright\underline{\sqcup}w) \vdash_M^* (h, \triangleright\underline{\sqcup}y)$ with $y \in \Sigma_0^*$. y is called the output of M on input w, notated as $M(w)$. Now our Turing machine has all the necessary features to compute a function.

We will say that M computes function f, if for all $w \in \Sigma_0^*$, $M(w) = f(w)$ for any $w \in \Sigma_0^*$.

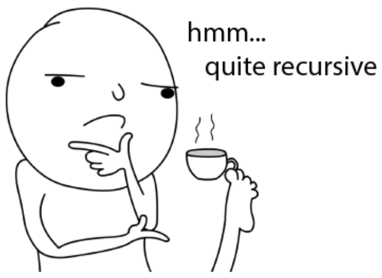
A function f is called **recursive**, if there is a Turing machine M that computes f.

For example, the function $k : \Sigma^* \to \Sigma^*$, defined as $k(w) = ww$ can be computed by the machine $CS_{\leftarrow}$ (the copying machine followed by the left-shifting machine.

A more practical example would by using strings $w \in \{0, 1\}*$. This would allow our machine to perform computations using natural numbers.

See for example the successor function $succ(n) = n + 1$ together with a possible implementation in C.

$$\begin{tikzcd}
 & 0 \arrow[dl, bend right] \\
R_{\sqcup}L \arrow[ur, "1"] \arrow[r, "0"] \arrow[dr, "\sqcup"] & 1 \\
 & 1\,S_L
\end{tikzcd}$$



```
void    shift_right()
{
    int i;
    for (i = 1; i <= n; i++)
        v[i] = v[i - 1];
    n++;
}

void    succesor(int i)
{
    if (i < 0)
    {
        shift_right();
        v[0] = 1;
    }
    else if (v[i] == 0)
    {
        v[i] = 1;
    }
    else
    {
        v[i] = 0;
        succesor(i - 1);
    }
}
```

Figure 13: Hmm... quite recursive

8.5 What's the matter with Turing machines and recursion?

We just called the functions and languages decided by Turing machines recursive. Why is this? The term of recursive functions came before there were any other model of computation like Turing machines. But, when Turing machines came in, it was proved that they are equivalent. This lies in the fact that Turing machines were, themselves, constructed in a recursive manner. We started with simple machines and gradually created complex machines by repeatedly combining and calling the simple ones.

8.6 Recursively Enumerable Languages

Given the Turing machine $M = (K, \Sigma, \delta, s, H)$, the input alphabet $\Sigma_0 \subseteq \Sigma \setminus \{\sqcup, \triangleright\}$ and the language $L \subseteq \Sigma_0^*$, we say that M **semidecides** L if for any string $w \in \Sigma_0^*$, $w \in L$ if and only if M halts on input w. In this case L will be called **recursively enumerable**.

After we get some headaches from the definition, we finally understand that what it wants is M not to halt in order to consider L semideciable. If $w \in \Sigma_0^* \setminus L$, then M must never enter the halting state. In a more simple way, we cloud say that M never halts on input w if and only if $w \notin L$.

Taking a practical example, let $L = \{w \in \{a, b\}^* :$ w contains at least one $a\}$. L is "semidecided" by the following Turing machine:

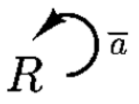


Figure 14: Looking for an a

The machine simply scans to the right until it finds an a. We can anticipate that if there is no a in the string provided as input, this machine will never halt. But, it will always halt on y if there is an a. Considering that we can't know if we waited enough for the machine to halt, we can see that these machines are not reliable at all.

What's the connection between recursive and recursively enumerable languages? Are recursive languages also recursively enumerable? The answer is yes. If a language L is decided by a machine M, we can create the machine M' which semdecides L by replacing the n halting state with an infinite loop.

The challenging part is if we can transform any machine that semidecides L into one that decide L. Sometimes yes, but there are recursively enumerable languages that are not recursive.

9 Extensions of the Turing machines

From the results we obtained so far, some people may consider the model of Turing Machines as being weak and slow. Why to only have one tape and not two? Or why not adding multiple head to move along the tape at the same time? These are the questions we will address in this section. We will show that these models may increase the speed, but regarding the class of functions they will compute or the class of languages they will decide, these will remain the same. In the end, we will show that all these “new and improved” machines can be simulated by a standard Turing machine.

9.1 Multiple tapes

What about a Turing machine with multiple tapes? We will notate the number of tapes with the finite integer $k \geq 1$. Each tape will have a corresponding head that will start right beside the left end and will move along it.

By convention, the input will be provided on the first tape and the others will contain only blank symbols. So, having the Turing machine $M = (K, \Sigma, \delta, s, H)$, its configuration will be a number of $K \times (\triangleright\Sigma^* \times (\Sigma^*(\Sigma \setminus \{\sqcup\}) \cup \{e\}))^k$. We will notate with a_i $(1 \leq i \leq k)$ the currently scanned symbol on tape i.

The configuration $(q, (w_1 a_1 u_1, ...w_k a_k u_k))$ will yield in one step the configuration $(p, (w_1^{'} a_1^{'} u_1^{'}, ...w_k^{'} a_k^{'} u_k^{'}))$.

Taking a practical example, let’s see how a 2-tape Turing machine would perform the task of the copying machine we presented earlier:

(I) First, it would move along both tapes at the same time, copying each element from the first to the second

(II) Than, only on the second tape, it will move the head to the left until it finds the blank at the beginning

(III) The lase step will be to move again along the both tapes at the same time, but this time it will copy each symbol from the second tape to the first

These machines can be represented in the same way we represented earlier complex ones, by adding an index in the right upper corner corresponding to the machine they act on. For example, we will write $R^{1,2}$ to say that the head will move to the right on both tapes. Or a^2 to say that we write an a on the second tape.

So, the previous machine would look like this:

$$\begin{tikzcd}
R^{1,2} \arrow[r, "a^1 \neq \sqcup"] \arrow[rr, bend right, "\sqcup^1"'] & a^2 \arrow[l, bend right] & L_{\sqcup}^{2} R^{1,2} \arrow[r, "a^2 \neq \sqcup"] & a^1 \arrow[l, bend right]
\end{tikzcd}$$

Figure 15: 2-tapes Copying machine

Or we could create the following machine for adding two numbers using their binary representation.

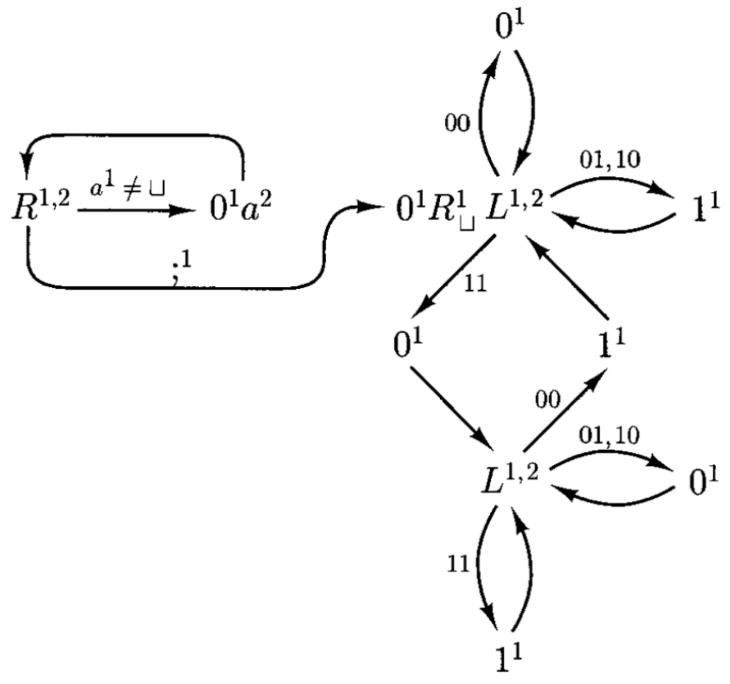


Figure 16: Adding two numbers

Theorem: Let $M = (K, \Sigma, \delta, s, H)$ be a k-tape Turing machine for some $k \geq 1$. Than, there is a standard Turing machine $M' = (K', \Sigma', \delta', s', H)$, where $\Sigma \subseteq \Sigma'$, and such that the following holds: For any input string $x \in \Sigma^*$, M on input x halts with output y on the first tape if and only if M' on input x halts at the same halting state and with the same output y on its tape.

Proof:

In order to prove this theorem we will build a standard Turing machine M' which will simulate the computation of M. First, M' must somehow contain all the information from the k tapes of M on its single tape. So, we will divide its tape in several sections, each containing a specific tape of M. More exactly, the single tape will be divided into $2k$ sections: the odd sections will hold the content of the tapes and the even ones will be used for keeping track of the position of the head for each odd tape. This will be achieved by letting the even sections contain a string $(0*10*)$. Having an 1 on the $i-th$ square of the section, will mean that the head will be positioned on the $i-th$ square of the previous odd section.

For example, if we have a machine with two tapes containing the string $aa\underline{a}b$ on the first tape, and the string $\underline{b}aa$ on the second tape, our one-tape machine which simulates this will contain $\triangleright\, aaab$ 0010 $\triangleright\, baa$ 100 . $\triangleright$ will be the left end of each section.

So, the alphabet of M' will be $\Sigma' = \Sigma \cup (\Sigma \times \{0,1\})^k$.

Our machine M' will receive the same input as M.

(I) The first step is to divide its only tape into k sections. It will start by shifting its input one square to the right, than it will write $(\triangleright, 0, \triangleright, 0, ..., \triangleright, 0)$, the first symbol of each tape, in front of the provided input. Than, at the end of each section which currently has length 1, it will add a blank symbol, respectively an 1, meaning that the first blank is the currently scanned symbol on each tape. Now that the "template" of each tape is set, it will start adding the input in the first section and blanks on the others. It does this by scanning each element a of the input and adding $(a, 0, \sqcup, 0, ..., \sqcup, 0)$ until it finds a $\sqcup$(meaning that it reached the end of the input).

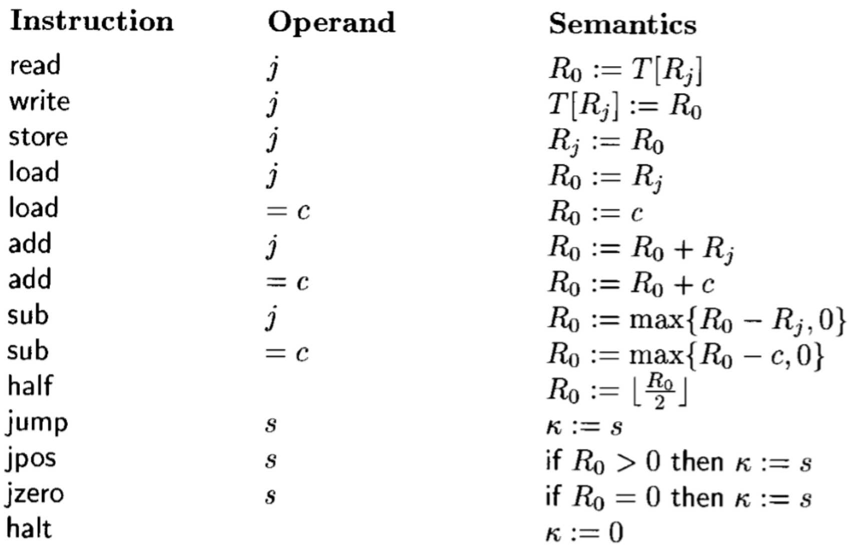
(II) Now, M' has to simulate each step of the computation of M. To simulate one step it will go to through the whole tape until it reaches the left end, gathering information about the currently scanned symbol of each section("tape"). Than it goes again through the tape simulating the steps of M. More precisely, it will either move the head position marker or write another symbol for each section(if any of the two happens on the corresponding tape of M).

(III) When M halts, M' halts in the same state, ignoring anything but the first section. (the same way M would provide its output on the first tape).

10 Random Access Turing machines

One obvious drawback of the Turing machines we have seen so far is that they sequentially access the content of the tape, instead of directly accessing any needed square. In order to achieve this more natural way of computing we will define Random Access Turing machines. They will be equipped with tape squares called registers which will have a special role in the computation. Instead of the table and the transition function δ we previously used, this time we will have the **program** $\Pi = \{\pi_1, \pi_2, ...\pi_p\}$, where $\pi_i (i \leq p)$ is an instruction.

The available instructions are given by the following table:



Instruction	**Operand**	**Semantics**
read	j	$R_0 := T[R_j]$
write	j	$T[R_j] := R_0$
store	j	$R_j := R_0$
load	j	$R_0 := R_j$
load	$= c$	$R_0 := c$
add	j	$R_0 := R_0 + R_j$
add	$= c$	$R_0 := R_0 + c$
sub	j	$R_0 := \max\{R_0 - R_j, 0\}$
sub	$= c$	$R_0 := \max\{R_0 - c, 0\}$
half		$R_0 := \lfloor \frac{R_0}{2} \rfloor$
jump	s	$\kappa := s$
jpos	s	if $R_0 > 0$ then $\kappa := s$
jzero	s	if $R_0 = 0$ then $\kappa := s$
halt		$\kappa := 0$

Figure 17: The set of instruction

- $T[i]$ - the content of the $i - th$ tape square

- R_j - the content of register j
- K - the index of the executed instruction

We will notate out Random access Turing machine as a pair $M = (p, \Pi)$, where p is the number of registers, and Π the program.

A configuration of such a machine is a $k + 2 - tuple$ $(k, R_0, R_1, ...R_p, T)$

At each step a configuration $(k, R_0, R_1, ...R_p, T)$ would yield in one step the configuration $(k', R'_0, R'_1, ...R'_p, T')$, denoted as $(k, R_0, R_1, ...R_p, T) \vdash_M (k', R'_0, R'_1, ...R'_p, T')$.

Using RAM Turing machines, we could write the following program which would compute Fibonacci infinitely:

1. load =1
2. store 2
3. load =1
4. store 1
5. add 1
6. add 2
7. store 3
8. load 1
9. store 2
10. load 3
11. store 1
12. jump 5

As we previously proved that a Multiple tapes Turing machine can be simulated by the basic machine we'll now do the same with Random access memory Turing machines.

Let $M = (k, \Pi)$ be our Random access Turing machine. We will build M', a $(k + 3) - tape$ Turing machine, where k is the number of registers of M. (As we previously proved that a k-tape Turing machine can be simulated by the one tape, it's enough if we prove that the RAM Turing machine can be simulated by a k-tape one).

We will make use of the tapes as follows:

- 1^{st} tape will keep track of the input and eventually the output.
- 2^{nd} tape will simulate the tape of M. It will hold sequences of the form (x, y), where x and y are binary integers, representing the tape square position on M and the element it contains.
- the next k tapes will contain each register of M, also represented as binary integers
- the $(k + 3)^{th}$ tape will hold, when necessary, constants (for example when the instruction $load = c$ is executed)

After it receives the input, M' starts simulating each instruction of M.

For example:

- The instruction $add4$. It will add the binaries from the tapes representing Registers 4 and 0 and will leave the result in Register 0.
- The instruction $add = 33$. It starts by writing the binary representation of 33 on the $(k + 3) - th$ tape, than adds it to Register 0 and in the end erases its content.

11 Grammars

In a previous section we analyzed Turing machines as **language acceptors** and two important families of languages have resulted: **recursive** and **recursively enumerable languages**. There is an another important family of devices called **language generators** which are used to define different classes of languages. We will now introduce the a language generator called **grammar**.

A grammar is the set of structural rules governing the composition of clauses, phrases, and words in any given natural language. The purpose of this section is to show that the class of languages generated by grammar is equivalent to the class of recursively enumerable languages.

12 Personal considerations

12.1 A general proof

Turing machines serve as a great theoretical abstract model of computing devices on which one can easily analyze aspects of computation. Since anything that can be computed by a real machine can also be computed by a Turing machine, limitations of Turing machines also apply to our real-world machines. These models gave rise to a lot of theoretical analyses that resulted in a profound understanding of aspects of computability.

It became quite intuitively that any normal features we would add to our machines, those will be in the end simulated by the basic machine. The proofs we have seen so far in this sense are not unique. One may think of another machine M' which would simulate the k-tape machine or the Random access Turing machine. In my opinion, a broad proof is needed for all of those machines which would encapsulate the properties of those added features like data accessing and data storing. Such a proof would lead to a better understanding of the universality of Turing machines or maybe the contrary. Because it was not yet proved there there is no computable function that can't be computed by a Turing machine.

12.2 The Turing Test

In 1950 Turing published a paper called "Computing Machinery and Intelligence" and proposed the question "Can a computer talk like a human?" together with a test called nowadays the Turing test. The test consists a person, let's call him "the tester", talking to other persons only through text messages without any other kind of interactions. During the test, a computer would have to replace a person's responses. If it manages pass unobserved from the tester, it passed the test and it's considered intelligent. This kind of tests are held nowadays annually with prizes for whom designs the most intelligent machine. But there is also a prize for "The most human human". It goes to the person which succeeded the best at convincing the testers that he is a human and not be considered a robot. This is because testers may consider the answer of a actual person as being received from a robot.

I thought about this and I realized that humans often choose their answers based on some kind of algorithm. Especially if we speak about introverts like myself. I often find me analyzing the situations I'm in and giving answers based on the data I get from outside. Like an input. For example I have very often thoughts like "The boy I met yesterday said he's into reggae music and he also seemed to be an easy going person. So probably I should reply to his message with 'thanks' instead of 'Thank you!'"; Maybe we could design some Turing machine to help introverts dealing faster with this kind of thoughts?

The Turing test, as he was proposed by the hero of this paper, is paired with discussion. But what if we would consider other interactions between human beings? There are some expensive machines which make a great coffee. I certainly fall in love with some of them. Or at least.. I have some kind of appreciation. This may be considered as a passed Turing Test.

I just told my girlfriend about the incredible philosophy I just wrote in the previous lines and she immediately became jealous: "How could you? A machine does that because it was programmed to do so, not with love like the people close to you do." (I don't really appreciate her coffee). I think this event just makes our machine pass another Turing test, considering that someone became jealous on it and jealousy it.

Should the machines deserve rights? Mostly, the key aspect of living that gives people rights is suffering. A robot doesn't perceive that unless it's programmed to. Human rights are a product of a instruction in ourselves: "survive". They were created in order to keep the human beings away from danger and help them prosper.

What if a super artificial intelligence machine, let's call it "Rici", would be programmed with such kind of instruction like: "evolve". And what if ...

$$evolve \vdash^*_{Rici} get\ rid\ of\ humans\ ?$$

Or maybe...

$$evolve \vdash^*_{Rici} make\ some\ coffee\ ?$$

References

[1] HARRY R. LEWIS *Elements of the Theory of Computation*, cap IV

[2] MARTIN DAVIS *The road from Leibniz to Turing*