Turing Machines and the Halting Problem

Nicholas Silva Tee

July 22, 2022

Abstract

A Turing machine is a mathematical model of computation and is represented as an abstract machine. As a part of Automata theory, a popular subject taught in undergraduate computer science departments. Turing machines are the most powerful of the different automata and have different versions, such as multi-tape and nondeterministic Turing machines. Coined by computer scientist Alan Turing, the halting problem is a popular decision problem considered to be unsolvable using Turing machines.

1 Introduction

This paper will review some fundamentals of automata theory before delving into Turing machines and their different variations. These variations include multi-tape and nondeterministic Turing machines. Other theoretical aspects of Turing machines will also be discussed, such as the Universal Turing machine. We will also cover the halting problem, it's proof of undecidability and applications of both the problem itself and Turing machines.

1.1 History

The initial intention of Turing was to recreate some of the "mechanical" processes we do as human beings. A paper was published titled "On Computable Numbers with an application to the Entscheidungsproblem" by Alan Turing himself in 1936. Entscheidungsproblem is the German word for "decision problem".[2] Turing suggested a universal machine that would be able to solve any problem and could be operated by using simple instructions on a tape, this would later be known as the "Universal Turing Machine" which we will discuss later on in this paper. The machine he described was completely a theoretical concept instead of an actual physical machine. The term "Turing Machine" was only actually coined by another mathematician Alonzo Church during a review of the paper.

2 Automata Theory

In order to properly understand Turing machines, we first need to understand some of the more basic concepts within automata theory. Computer scientists use automata theory in order to better describe and understand the behaviour of discrete systems through "machines". Some of these major machines are known as **finite-state machines**, **pushdown automata**, and Turing machines.

In automata theory, all the different types of machines have three characteristics in common:
Inputs: An arbitrary set of symbols $\{x_1, x_2, x_3, ..., x_n\}$ such that n is the number of symbols. We will see that this set is commonly referred to as the **alphabet** of the finite-state machine.
Outputs: A set of output symbols $\{y_1, y_2, y_3, ..., y_m\}$ such that m is the number of symbols.
States: A finite set, normally defined as Q, the definition of this set, depends on the type of machine we are using. These can be considered as the "steps" in the automata.

2.1 Finite-State Machines

Finite-state machines may be classified into three different types:
Acceptors: these machines give us a binary output. Essentially, they take in an arbitrary input, and either accept or reject it.
Classifiers: A more generalized version of accepters, where the output is not binary and there are an n types of outputs.
Transducers: Given some arbitrary input, it will generate an output(not binary)
In this paper, we will solely focus on acceptor types of state machines.

There are many different ways to define a finite-state machine, but we shall use the general mathematical definition.

Definition. *A deterministic finite state machine is a quintuple/5-tuple denoted as* $(\Sigma, S, s_0, \delta, F)$ *such that:*

- Σ*: Is the non-empty set of possible inputs, also known as the alphabet*
- S*: A non-empty, finite set of states.*
- s_0*: The starting state and an element of* S
- δ*: Known as the "state-transition function", it is a function that can be defined as* $\delta : S \times \Sigma \to S$*. F: The set of final states, or also known as accept states. The set may be empty, and is a subset of* S*.*

[5]

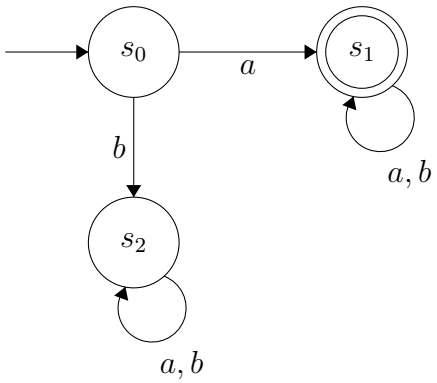
Note that the definition above is mean for a **deterministic** finite-state machine. There are Deterministic finite automata(DFA) and non-deterministic finite automata(NFA). The

only difference between the two is that for each state in a DFA, there is only one transition, whereas each state in a NFA, may have more than one.

Example 2.1. As a simple example, let us construct a DFA such that it only accepts strings that begin with the letter 'a'. With this high level description we can define the 5 elements that make a DFA such that, $\Sigma = \{a, b\}$, $S = \{s_0, s_1, s_2\}$, and $F = \{s_1\}$. We can then represent δ with the following table.

	s_0	s_1	s_2
a	s_1	s_1	s_2
b	s_2	s_1	s_2

The top row represents the three states we have, and the leftmost column are the possible inputs for each state, we can see that if our starting state is s_0 and the input it receives is 'a', then the machine will move onto s_1 and will stay there. DFA's are also commonly drawn out in a graphical notation.



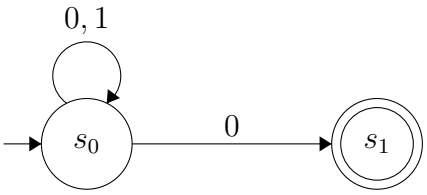
The state which has a blank arrow pointing to it, in this case s_0 is how we draw the starting state. The state or states that have another circle drawn around it is how we represent a final state.

Recall that NFA's are non-deterministic, so they have more than one possibility and will always lead into branching paths when processing an input.

Example 2.2. Take the alphabet $\Sigma = \{0, 1\}$ and the language $L = \{\text{all strings that end with } 0\}$. Constructing a DFA that accepts the language might be a little tricky, however, it is much easier when we use an NFA. We can simply use only 2 states to construct a machine that accepts the language, the transition table may be represented as: Notice how when s_0 reads

	s_0	s_1
1	s_0	–
0	s_0, s_1	–

a '0', it has two options, itself and s_1, this is where two branches are created and we must keep track of both. If any of the branches ends in a final state, then the NFA accepts the string. If no branch ends in a final state, it is rejected. A graphical drawing of this NFA may look like this:



For instance, let use '10' as an input. The first step is straightforward as it loops back to s_0. However, once '0' is read, we have two branches, one branch moves onto s_1 whilst the other stays in s_0. Since, one of the branches ends in a final state, the NFA accepts the string.

Similarly, if we take '01' as an input, we have our initial branch. However, since the next character in the input is '1', the branch that stayed in s_0 still stays there, and thus fails. The branch that moves onto s_1 does not have a transition, and thus automatically fails as well. Thus, the NFA does not accept the string.

Theorem 2.3. *Every NFA can be reconstructed into an equivalent DFA*

NFA's also allow for **empty string** transitions. Represented by the ϵ symbol, it essentially acts as a free move within the transition function. Although they can technically be used in a DFA, they are more commonly found when designing NFA's.

2.2 Pushdown Automata

Similar to a DFA, pushdown automata (PDA) have the same characteristics, but also add two more:
Γ: The set of pushdown symbols, essentially a stack.
Z: The initial symbol we push into the stack.

Remark. In computer science, a stack is a way of storing information, where you can only look at the most recent thing stored (the top of the stack). An analogy for understanding this concept is by thinking of a stack of plates, where you can only "access" the top plate, where you can remove it, or add another plate on top. When **push** onto the stack when we add a symbol, and **pop** when we remove symbols. If the PDA has gone through the entire input, and Z has been popped off the stack, then we accept the input.

Example 2.4. Given that $\Sigma = \{a, b\}$ and the language $L = \{a^n b^n | n \geq 0, n \in \mathbb{N}\}$ (Strings such as 'ab', 'aabb' and 'aaabbb'), lets design our PDA. We have $S = \{s_0, s_1, s_2, s_3\}$ and $F = \{s_0, s_3\}$. Since PDA's are more complicated than DFA's, lets start with a high level description of the process:

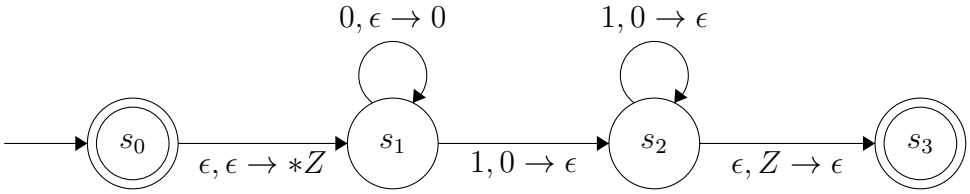
1. push Z onto the stack
2. if we read an 'a' in the input, push 'a' onto the stack. If we see 'b' while pushing, we halt. else, we move to (step 3)
3. if we read 'b', we pop 'a'. If we still read 'b' and there is no 'a' left in the stack, we halt.

4. Pop Z and accept the strin

We can draw out the PDA in graphical notation, similar to what we did with the DFA, but instead the transitions are formatted as such

$$x, y \to z$$

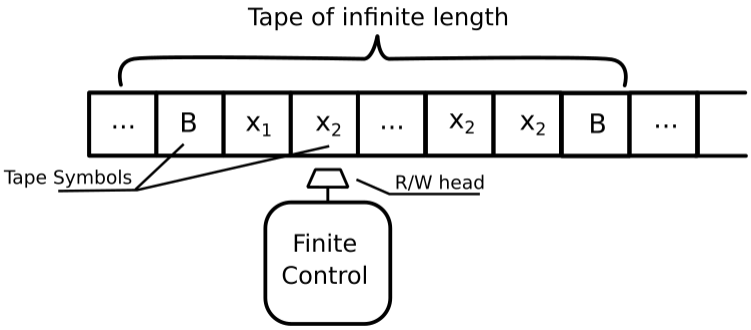
Where x is the input symbol being read, y is the symbol being popped from the stack, and z is the symbol being pushed. If we do not want to push or pop onto the stack, we can use ϵ.



3 Turing Machines

3.1 Basics

Turing machines can be thought of as more powerful versions of PDA's. First, instead of a stack, we now have an infinite tape to store our information, and we can only read and write to one digit on the tape at a time. A visualisation of this tape might look like this:



Turing machines have 4 main actions that it can do:

1. move right on the tape
2. move left on the tape
3. overwrite the current symbol on the tape with a new one

4. Halt and either accept the tape of reject.

The output of the Turing machine will be the tape at the end of the process. Note that there is also the possibility that the Turing machine can be stuck in a loop and never halt, which will be discussed later on.

Remark. When scanning a Turing machine tape, the current section of the tape we are looking at is called the **tape head**

Definition. *We can define a Turing machine as a 7-tuple of* $(Q, \Sigma, \Gamma, \delta, s_0, B, F)$ *where:*

- Q*: non-empty set of states*
- Σ*: alphabet / non-empty set of symbols used*
- Γ*: non-empty set of tape symbols*
- δ*: The transition function defined as* $Q \times \Sigma \to \Gamma \times (RL) \times Q$*. Where* R *and* L *represent the right and left movement on the tape.*
- s_0*: the initial state,* $s_0 \in Q$
- B*: the blank symbol* $\sqcup$*, a special symbol to fill out the empty spaces on the infinite tape*
- F*: Set of final states. However, in this case* $F \not\subset Q$ *as there are only two, 'ACCEPT' and 'REJECT'*

Furthermore, when creating the graph for a Turing machine, our transitions are formatted as such:

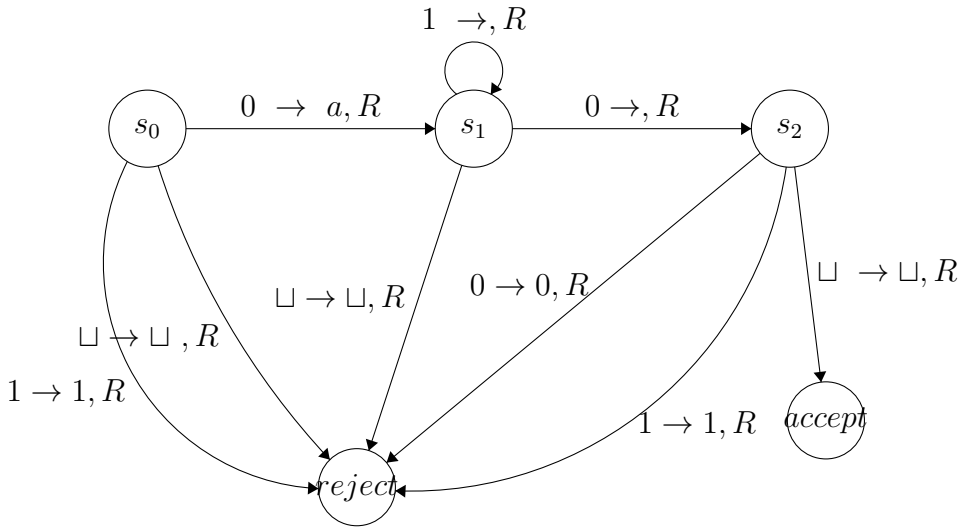
$$x \to y, z$$

Where x is the symbol currently being read on the tape, y is the symbol we want to write onto the tape, and z is the movement, whether we wish to move right or left. Similar to the PDA, if we do not wish to do any of these actions on a certain transition, we can always fill the spot with $\sqcup$.

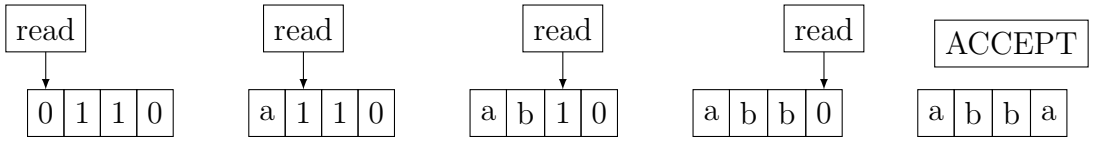
Example 3.1. Lets take the alphabet $\Sigma = \{1, 0, a, b\}$ and the language $L = \{01^n0 | n \geq 0\}$. Ideally, it is quite simple to create a Turing machine that can accept this language, from a high-level perspective, the following steps can be taken:

1. Read the first symbol, if it is not a 0, then halt and reject, else move to step 2 and move tape right.
2. Keep reading 1's and moving right on the tape, if there is a blank, halt and reject. If a 0 is read, move to step 3
3. If a 1 or 0 is read, halt and reject, else, if a blank is read, halt and accept.

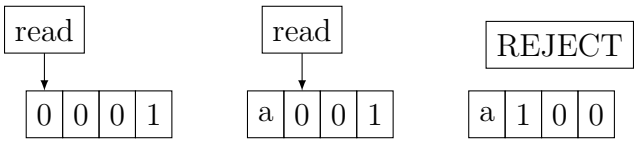
The following Turing machine can then be drawing out as such:



If we were to feed this Turing machine an input in L such as the string '0110'. The tape may look something like this.



Note that the language can be accepted without having to overwrite the original symbols, but was done so in this example merely as a demonstration. Similarly, if we were to give the Turing machine an input of '0010', it would be rejected as such:



Recall that Turing machines do not necessarily have to halt and accept or reject, there is the possibility of a Turing machine never halting. With the alphabet $\Sigma = \{1, 0\}$, let us construct a Turing machine with the following rules.

- If a 1 is read on the tape, move right
- If a 0 is read on the tape, move left
- If a $\sqcup$ is read on the tape, halt and accept
- If you are on the leftmost section of the tape, and have to move left, halt and reject

If we look at several different inputs, we can see that with '111', the Turing machine halts and accepts, and with '000' it halts and rejects. However, with inputs such as '10' or '101', the Turing machine gets stuck between 1 and 0 continuously moving right and left. In this case, the machine never halts.

3.2 Universal Turing Machine

The universal Turing machine was another theoretical concept that Alan Turing had come up with the idea of a Turing machine that could mimic other Turing machines by taking in a description of the Turing machine along with its input. Although the concept was a purely theoretical one, it is believed to be responsible for the creation of stored-program computers by John von Neumann.[12]

This Turing machine can be represented as U and takes in an input $< T, i >$ such that T is a Turing machine and i is an input. Once fed into U, it will perfectly simulate T. If T accepts or rejects, then U does the same. Similarly, if T never halts, then U also never halts.

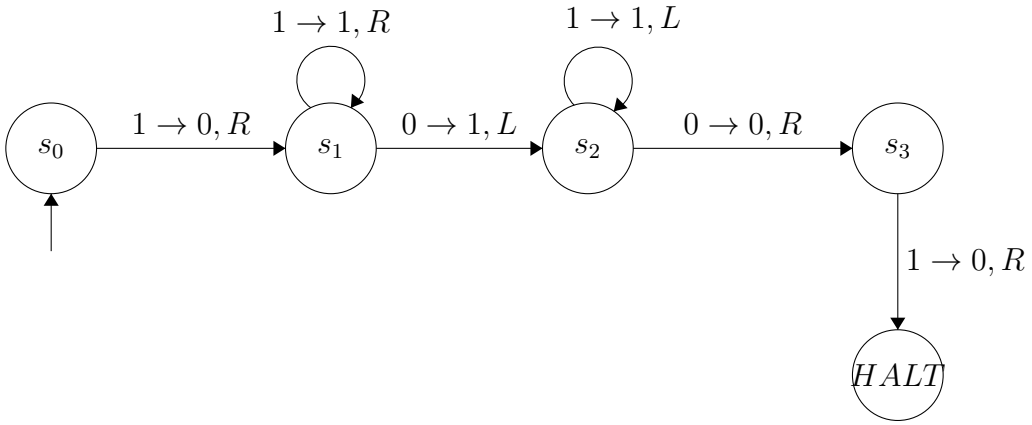
3.3 Church-Turing Thesis

The Church-Turing, named after mathematician Alonzo Church and Alan Turing himself. The two computer scientists concerned themselves with the idea of computability and how to define it. Despite not having a formal sentence that acts as the thesis, the general idea is that any problem or computation that can be done in the real world(with finite mean), can also be done with a Turing machine.

Example 3.2. As an example, let us construct a Turing machine that does simple addition of two numbers. This can be done using unary notation (2 is represented by 3 '1' in a row) and the two numbers added will be separated by a single 0. A tape that represents $2 + 3$ can be seen as such.

1	1	1	0	1	1	1	1

We can always assume that the input tape will have the correct formatting and that the Turing machine will always start at the leftmost '1' of the tape. All we need to do is move the left most '1' in between the two numbers and then remove an extra '1' at the beginning. We can represent this process with the following graph.



3.4 Multitape Turing Machine

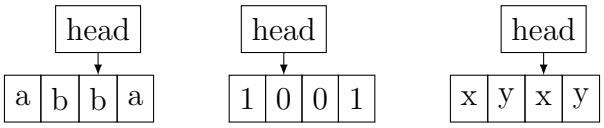
Turing machines do not necessarily only have to have a single tape. Turing machines can be constructed such that they can have any number of tapes at the same time. Each tape in a multitape Turing machine act independently from each other and their contents do not mix, they also have independent tape heads.

Definition. *The definition of a multitape Turing machine is the same as a regular one except for the transition function as it is now defined as*

- δ: $Q \times \Gamma^k \to Q \times (\Gamma \times (RL))^k$

such that k is the number of tapes the Turing machine has.

[17]
An example of a three tape Turing machine can be shown below.



From left to right, let us label each tape 1,2 and 3. If we are to draw out a multitape Turing machine in a graph as we have done before, we need to change our formatting slightly. Assume we want to create a transition that does the following for each tape:
tape 1) if 'b' is read, write 'a' and move right
tape 2) if '0' is read, write '0' and move right
tape 3) if 'x' is read, write 'y' and move left
We can then represent our transitions as such

$$b0x \to a0y, RRL$$

where for each section of the transitions, the leftmost character represents the first tape.

Theorem 3.3. *Every multitape Turing machine can be reconstructed as a single-tape Turing machine.*

Proof. We have some input string represented as $w_1 w_2 \cdots w_n$ where n is the length of the string. We then place all the separate tapes onto a single tape and add a dot on top of each symbol where the tape head resides for each tape, divide the tapes by using a '#'.

$$\#\dot{w}_1 w_2 \cdots w_n \# \dot{\sqcup} \# \dot{\sqcup} \# \cdots \#$$

Once we have created the single tape, we scan through the entire tape to find all the heads, then re-run through the entire tape to following the instructions of the original multitape Turing machine. If during the process one of the 'tape heads' reaches a #, then shift the entire tape and add in a blank symbol. □

From this process, let us take the three tapes we had and convert it into a single tape

$\dot{a}$	b	b	a	#	1	$\dot{0}$	0	1	#	x	y	$\dot{x}$	y

Note that although all multitape Turing machines can be converted into a single tape Turing machine, it increases the runtime making the process much slower.

3.5 Non-deterministic Turing Machines

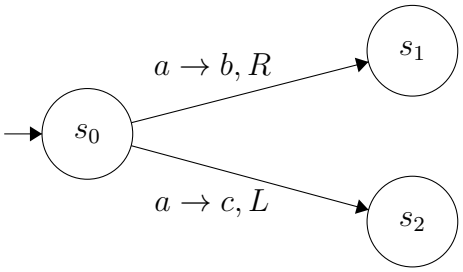
Similar to DFA's and NFA's, Turing machines can also be constructed to be non deterministic.

Definition. *The definition of a non-deterministic Turing machine is the same except for the transition function which we now define as*

- $\delta : Q \times \Sigma \to P\{\Gamma \times (RL) \times Q\}$

Where P represents the power set, to account for all the different branches possible in a non-deterministic machine

[7]
As an example, let us look at this graph of a non-deterministic Turing machine.



Let us have some input tape as such, where to dotted symbol indicates the location of the tape head.

b	$\dot{a}$	b	a

Similar to NFA's, since the state s_0 has two possibilities, we have to consider both possible branches. Now, we have two different tapes to account for

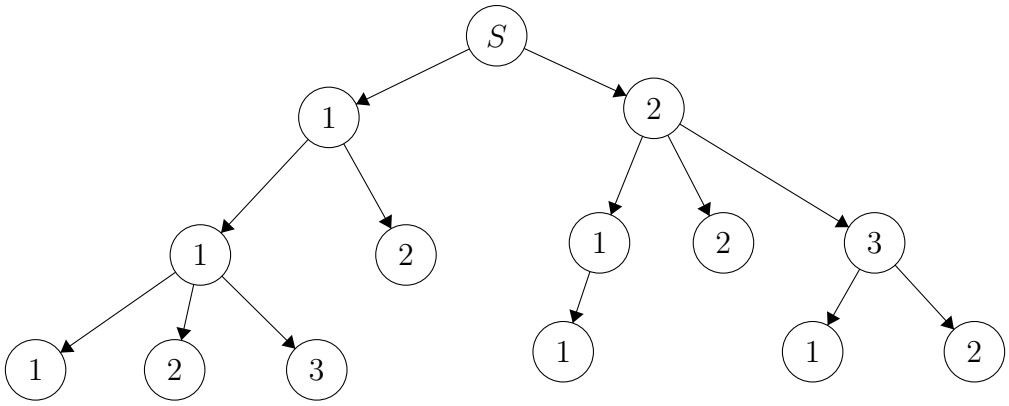
b	b	$\dot{b}$	a

$\dot{b}$	c	b	a

When a branch of the Turing machine calculations halt and accept the string, then the Turing machine accepts the string. However, in order for the Turing machine to reject, all branches have to either reject or die out(tape ends in a non final state). If a branch never halts, then we count that has the branch dying out.

Theorem 3.4. *Every non-deterministic Turing machine has an equivalent deterministic Turing machine that accepts the same language*

We understand that when running a non-deterministic Turing machine, we have to account for all branches. However, since it is impossible to tell which branch will make the Turing machine halt, we need to simulate all of them. Since non-deterministic automata in general have to consistently branch out, we can create a tree of all the possible 'decisions' that the Turing machine makes. Lets take an arbitrary tree to use in our example.



Where S is that start and the numbers simply represent the possible choices when we reach a non-deterministic situation. For instance, a path that we simulate may be represented as "231". With all the possible paths in mind, we can construct out deterministic Turing machine, specifically a multitape Turing machine. This machine will have three separate tapes which do the following:

- **Tape 1:** This tape is the input tape, this tape is never modified and simply read from.
- **Tape 2:** The 'main' tape, where the simulation of a regular single tape Turing machine occurs. This tape is empty during the start of the process.
- **Tape 3:** Contains all the possible paths within the tree.

With these three tapes, our process looks like this:

1. Copy the contents from tape 1 to tape 2
2. Consult tape 3 as to which 'path' on the tree to take
3. Use tape 2 to simulate the path stored in tape 3
4. Increment onto the next path in tape 3
5. reset tape 2 and repeat the whole process

Throughout the process, if one of the branches leads to the Turing machine accepting the branch, then we can halt early and accept. Otherwise, we keep simulating all the branches and eventually reject if nothing is accepted.

3.6 Turing Machines and Languages

Here are some terminology of Turing machines and languages [13]

Definition. *Let there be a language L and a Turing machine T. If the machine T accepts all strings in L and rejects all strings that do not exist within L. i.e. the Turing machine T will always halt, no matter the input. Then tha language L is known as a* ***recursive language***

Definition. *Let there be a language L and a Turing machine T. If T accepts and halts on all strings in L, but may not necessarily halt for strings not in L. Then the language L is known as* ***recursively enumerable language***

If a language L is either recursive or recursively enumerable on a Turing machine T. Then we can say that T recognizes the language L.

Definition. *If a language L is a recursive language, then L is known as a* ***decidable language****. Note that the reverse of this statement is also true.*

Definition. *A language L is a recursively enumerable language, then it is also known as a* ***partially decidable language****. The reverse is also true.*

Definition. *A language L is* ***undecidable*** *if it is not a deciable language. This means that it is possible for an undeciable language to be partially decidable. If the language is not partially decidable, then there exists no Turing machine that exists to accept L.*

4 The Halting Problem

We understand that Turing machines can either halt, accept or reject, or never halt from being stuck in an infinite loop within its program. The halting problem asks a very simple questions. For some Turing machine T and an input i, can we build a Turing machine that will always tell us if T will halt or never halt, given the input i?[15]

Formally, we can define the halting problem as a language H. Such that H is the set of all inputs $< T, i >$ such that T is a Turing machine and halts on the input i. If there exists some other Turing machine M such that for all $(T, i) \in H$, M will halt and accept the input, then we have solved the halting problem. However, the halting problem has been proven the be **undecidable** and thus, unsolvable.

4.1 Properties of the Halting Problem

We can show that the halting problem is undecidable through a proof by contradiction.

Proof. Assume that we have some Turing machine M that perfectly solves the halting problem. So given two inputs, some Turing machine T and and input i. If we were to give both inputs of M as itself, then M should halt and accept.

Now let us create a new Turing machine M'. M' will depend on M, as if M determines that the given inputs will halt, then M' will reject and not halt. If M rejects the input and doesn't halt, then M' will halt. Now, if the inputs of M' are M' itself, we have two cases:

- if M accepts the input and deems that M' will halt. However, by definition of M', it will never halt.
- if M deems that M' will never halt. Then by definition of M' it will halt

In either case, we have a contradiction. Since we had made the assumption that M will always give us the correct output given its inputs, M' shows us that a Turing machine M can not exist, and thus the halting problem is undecidable. □

However, we can show that the halting problem is **recognizable**.

Proof. Recall that a language L is recognizable if there exists a machine that accepts all inputs in L and either rejects, or never halts in strings not in L. In the case of inputs in H, we can simply use a universal Turing machine U. If the machine U takes inputs $< T, i > \in H$, then by definition, U will accept if T accepts i, and reject or never halt if T does so. Then this shows that H is a recognizable language. □

4.2 Applications of the Halting Problem

Despite the halting problem being unsolvable by computational means, it does not mean we have not learned anything from it. The main benefit mathematicians and computer scientist learned from the halting problem is the fact it is proven to be undecidable. From this, the halting problem has been used to prove other undecidable problems.

If we are able to reduce a problem to the halting problem, then it is immediately proven to be undecidable. This could be done by simply showing that the problem is the same, or formulating a proof similar in style to the halting problem.[18]
Hypothetically, if the halting problem were solvable, it would be possible to solve some unsolved problems, these include the following:

- **Goldbach Conjecture**: ever even counting number greater than 2 is equal to the sum of two primes. [9]
- **Kolomgorov Complexity**: The smallest amount of computing power needed to produce an object
- **The Collatz Conjecture**: More famously known as the $3n + 1$ problem[14]

If the halting problem was solvable, and there did exist a Turing machine that could solve it, then we could also construct one that could solve all of the above problems. However, since it is not possible, these problems have to remain unsolved for the time being.

4.3 Rice's Theorem

From the proof of the halting problem, Rice's theorem was developed to generalize all undecidable problems that were reducable to the halting problem.

Definition. *Let P be some non-trivial property of a language, and the language that has said property, we label as L_p. If the language is recognized by some Turing machine T then we say*

$$L_p = \{< T > | L(M) \in P\}$$

is undecidable

Note that P is merely a set of languages. If $L \in P$ then we say that L satisfies the property P. P is also called trivial if it is satisfied by all recursively enumerable languages, or it is no satisfied by any at all. [3]

4.4 Orcale machines

From the study of decision problems such as the halting problem, in complexity and computability theory, there cam about another abstract machine called an oracle machines. These machines can be thought of as a Turing machine with an **oracle** connected to them. These oracles are essentially black boxes that can solve decision problems. These machines are purely theoretical.

The definition of an oracle machine is similar to that of a Turing machine. However, in addition an oracle machine adds the following components:

- **Oracle tape**: a separate tape from the Turing machine, and has a different alphabet.
- **Oracle head**: similar to the tape head of a Turing machine, but purely meant for the oracle tape
- **States**: Two special states are added, the ASK and RESPONSE state.

Now, an oracle machine can solve the original halting problem of just regular Turing machines, since it can decide if a Turing machine will halt or not. However, we essentially have another halting problem with oracle machines itself as it is unable to tell if machines such as itself will halt or not. This can be proven in the same way we did with the original problem.

References

[1] Arora, Neeraj. (2018). Introduction to Turing Machine.

[2] Brodkorb, Laurel. The Entscheidungsproblem and Alan Turing - Georgia College & State https://www.gcsu.edu/sites/files/page-assets/node-808/attachments/brodkorb.pdf.

[3] Bulma. "CSC 341 (Fall 2021)." Rice's Theorem, https://osera.cs.grinnell.edu/csc341/readings/rices-theorem.html.

[4] Brown, Christopher W. SI340: Functions & a Formal Definition of a DFA, https://www.usna.edu/Users/cs/wcbrown/courses/F17SI340/lec/l05/lec.html.

[5] Chor, Benny. A Deterministic FInite Automaton (DFA) Is a 5-Tuple. https://www.cs.tau.ac.il/ bchor/CM/Compute2-Printer.pdf.

[6] De Mol, Liesbeth. "Turing Machines." Stanford Encyclopedia of Philosophy, Stanford University, 24 Sept. 2018, https://plato.stanford.edu/entries/turing-machine/text=Turing%20machines%2C%20first%20described%20by,the%20computing%20of%20re

[7] Erickson, Jeff. “Lecture 38: Nondeterministic Turing Machines.”

[8] Gao, Alice. Alice Gao’s CS 245 Resources Page, https://cs.uwaterloo.ca/ a23gao/cs245-s18/index.shtml.

[9] “Goldbach Conjecture.” Encyclopædia Britannica, Encyclopædia Britannica, Inc., https://www.britannica.com/science/Goldbach-conjecture.

[10] Hehner, Eric. Problems with the Halting Problem - Department of Computer Science ... https://www.cs.toronto.edu/ hehner/PHP.pdf.

[11] John E. Hopcroft and Rajeev Motwani and Jeffrey D. Ullman (2003). Introduction to Automata Theory, Languages, and Computation

[12] Kamvysselis, Manolis. Universal Turing Machine, https://web.mit.edu/manoli/turing/www/turing.html.

[13] Kozen, D.C. (1997), Automata and Computability

[14] Letherman, Simon, et al. The 3n+1 Problem and Holomorphic Dynamics. A K Peters, Ltd.

[15] Salvador Lucas,The origins of the halting problem,Journal of Logical and Algebraic Methods in Programming,Volume 121,2021,100687,ISSN 2352-2208,https://doi.org/10.1016/j.jlamp.2021.100687.(https://www.sciencedirect.com/science/article/

[16] Turing, Alan. On Computable Numbers, with an Application to Computer Science. https://www.cs.virginia.edu/robins/TuringPaper1936.pdf.

[17] Siu, Christopher. “Multitape Turing Machines.” Christopher Siu, http://users.csc.calpoly.edu/ cesiu/.

[18] Wood, Ben. “Computability and the Halting Problem.” CS 251: Computability and the Halting Problem, https://cs.wellesley.edu/ cs251/s21/notes/halt.html.

[19] Robertson, Benjamin W. CSCE-433-500/CSCE-627-600 (Spring 2022), https://people.engr.tamu.edu/j-chen3/courses/627/2022/courseweb.html.