Turing Machines and the Halting Problem

Alexander Lue, UCSC

13 March 2019

Abstract

In this paper, we will discuss the basics of Turing machines, and some examples that use this computational model. Along the way, we will discuss an important problem these machines help address: the Halting Problem. In addition, we will talk about how they show other implications about decision problems and undecidability. We will then discuss how this affects problems in mathematics and our day to day life.

1 Introduction

The way we perceive computers is vastly different from what we had perceived almost 70 years ago. From the US made Electronic Numerical Integrator and Calculator (ENIAC), to the modern-day laptop, the fundamental logic that computers run on is largely preserved [2]. But how exactly could we compare these two, and why would these machines be related? Is there a way to interpret the computability of these technologies in a definitive way? Surprisingly, there is, and it was developed by mathematician and computer scientist Alan Turing in 1936 in his paper *On Computable Numbers, with an Application to the Entscheidungsproblem* [8]. In this paper, Turing devises an abstract "computing machine" that we now refer to as a Turing Machine.

While simplistic and straightforward, the Turing machine has proved to be one of the most important machines devised in early computer science. These abstract machines give means to describe computability of various problems posed before and after its inception that relate to mathematics and computer science in our everyday lives.

2 Computational Models and Automata Theory

Before we dive head first into Turing machines, we need to explain what computational models are. Computational Models are, at its core, models that describe how a set of outputs can be produced from given inputs with defined rules and function.

There are multiple classes of computational models, and all of which are encompassed in automata theory, the study of such machines. Specifically, automata theory delves into the computational logic that defines abstract machines. There are three main classes of models: finite state machines, pushdown automata, and Turing machines, where each is a subclass of the succeeding class [4].

But what does it mean when one automata is a subclass of the other? Let's consider finite state machines and pushdown automata. Any finite state machine can also be defined with a pushdown automata. This also means that any grammar, and by extension the language it produces, that can be constructed from a finite state machine, can also be defined with a pushdown automata. For example, finite state machines can only represent regular grammars, whereas pushdown automata can represent any context-free grammar, a superset of regular grammar.

Now with the general understanding of computational models and automata theory, we can now move onto Turing machines.

3 Turing Machines

As stated previously, Turing machines were developed by Alan Turing in 1936. They represent the largest class of automata in automata theory, and all computational models are encapsulated in the class of Turing machine.

3.1 Functionality and Design

A Turing machine is comprised of two components, a head pointer and an "infinite" tape reel with cells containing symbols. The machine works as such:

1. The head scans the symbol from the tape reel.
2. From the symbol read, the head will change its current state.
3. Head will then move one cell over to the Left or Right on the tape reel.
4. Head may overwrite the previous cell's symbol to a new symbol.

5. Repeat 1 - 4 until an end state has been reached.

With this functionality in mind, we can formally define a Turing machine [4], M, as a $7-$tuple $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$, where

- Q is a finite set of states
- Γ is a finite set of tape symbols, the "alphabet"
- $B \in \Gamma$ is the blank space symbol
- $\Sigma \subseteq \Gamma$, where B is not included, represents the "input symbols"
- $\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}$, is a partial function also known as the transition function
- $q_0 \in Q$ is the start state
- $F \subseteq Q$ is set of final states

We can relate the movement of the Turing Machine described in the procedure above as our δ function. δ is a partial function, and may therefore have inputs such that there will not be output. When this occurs, the machine will treat that as a termination and stop the machine from running. However, given an input into a program, this will not guarantee a termination and output to the program.

3.2 Examples of Turing Machines

Let's now go over a couple ways we can use Turing Machines to compute or determine outputs for a program. These simple examples[1] should be able to describe the general use and functionality of these machines.

Language Acceptance

Say we want to design a Turing Machine that can accept words in the language $L = \{0^n 1^n \mid n \geq 1\}$. We can create a Turing Machine that takes in a tape with the target word followed by blank spaces. The machine will first change the 0 into a X. The machine will then move to the right and change the first 1 it scans to a Y. The machine then moves left until it reaches the right of the X, and repeats

[1] All Turing Machine problems have been derived from Reference [4].

the movement until the machine reads a blank space at the end and accepts the word. If there are more 0s than 1s, then when the machine scans for another 1, it will reach a blank space first and not accept. Conversely, if there are more 1s than 0s, the last 0 will be read and will find another 1 before finding a blank, and such, will not accept the word.

We can define the Turing Machine as $M = (Q, \Sigma, \Gamma, \delta, q_0, B, \{q_4\})$ where $Q = \{q_0, q_1, q_2, q_3, q_4\}$, $\Sigma = \{0, 1\}$, and $\Gamma = \{0, 1, X, Y, B\}$. We will also represent δ as transition table:

		Symbol				
	δ	0	1	X	Y	B
State	q_0	(q_1, X, R)	-	-	(q_3, Y, R)	-
	q_1	$(q_1, 0, R)$	(q_2, Y, L)	-	(q_1, Y, R)	-
	q_2	$(q_2, 0, L)$	-	(q_0, X, R)	(q_2, Y, L)	-
	q_3	-	-	-	(q_3, Y, R)	(q_4, B, R)
	q_4	-	-	-	-	-

And so, we can see that when the machine is run on the input 0011 we reach the final state q_4:

$$
\begin{array}{llllllll}
q_00011 & \rightarrow & Xq_1011 & \rightarrow & X0q_111 & \rightarrow & Xq_20Y1 & \rightarrow \\
q_2X0Y1 & \rightarrow & Xq_00Y1 & \rightarrow & XXq_1Y1 & \rightarrow & XXYq_11 & \rightarrow \\
XXq_2YY & \rightarrow & Xq_2XYY & \rightarrow & XXq_0YY & \rightarrow & XXYq_3Y & \rightarrow \\
XXYYq_3B & \rightarrow & XXYYBq_4 & & & & & \\
\end{array}
$$

Notice that no matter what other input we put into the machine that is not in the language (i.e. 1001, 11111, or 00), δ is not defined to transition to a state. Therefore, when these inputs are put into the machine, it will terminate execution without accepting the word into the language.

Computational Machine

We can also use Turing Machines to conduct computations. Let's consider a simple type of computation, *proper subtraction* [2]. Proper subtraction between two non-negative integers m and n is the function:

[2]Turing Machines can theoretically work with normal subtraction as well, but for simplicity's sake, we want to use highlight this computation because it requires less state management.

$$m \dot{-} n = \begin{cases} m - n & \text{if } m \geq n \\ 0 & \text{if } m < n \end{cases}$$

So we can create a Turing Machine as follows. The input tape will contain the string $0^m 1 0^n$, so there will be m number of zeroes before the 1 and n number of zeroes after. The Turing Machine will start at the beginning of the tape and change the zero to a blank space, and then search for the first 0 after the 1 and change $0 \to 1$. The machine will then traverse back to the beginning and repeat the procedure until either occur:

1. **There are no more 0s right of the 1.** If this is the case, then the machine will backtrack and change all the 1s to blanks and then leaves the remaining 0s as the return difference.

2. **There are no more 0s left of the 1.** If this is the case and there are still 0s to the right, then the machine will recognize this and change all values to blank spaces.

We can model this with the Turing Machine as $M = (Q, \Sigma, \Gamma, \delta, q_0, B, \varnothing)$[3], where $Q = \{q_0, q_1, q_2, q_3, q_4, q_5, q_6\}$, $\Sigma = \{0, 1\}$, and $\Gamma = \{0, 1, B\}$. We will also define δ like so:

		Symbol		
	δ	0	1	B
State	q_0	(q_1, B, R)	(q_5, B, R)	-
	q_1	$(q_1, 0, R)$	$(q_2, 1, R)$	-
	q_2	$(q_3, 1, L)$	$(q_2, 1, R)$	(q_4, B, L)
	q_3	$(q_3, 0, L)$	$(q_3, 1, L)$	(q_0, B, R)
	q_4	$(q_4, 0, L)$	(q_4, B, L)	$(q_6, 0, R)$
	q_5	(q_5, B, R)	(q_5, B, R)	(q_6, B, R)
	q_6	-	-	-

Let's now work with two examples, 0010 and 0100. For 0010, the M will function in this order:

[3]Notice that because we are doing a computation and not a regex like the last example, there are no accept states because when the machine halts, we know that whatever is left on the tape afterwards is our solution.

$$
\begin{array}{llllllll}
q_0 0010 & \rightarrow & Bq_1 010 & \rightarrow & B0q_1 10 & \rightarrow & B01q_2 0 & \rightarrow \\
B0q_3 11 & \rightarrow & Bq_3 011 & \rightarrow & q_3 B011 & \rightarrow & Bq_0 011 & \rightarrow \\
BBq_1 11 & \rightarrow & BB1q_2 1 & \rightarrow & BB11q_2 & \rightarrow & BB1q_4 1 & \rightarrow \\
BBq_4 1 & \rightarrow & Bq_4 B & \rightarrow & B0q_6 & & & \\
\end{array}
$$

After the process is finished, only one 0 remains, which means the *proper difference* of 2 and 1 is 1. What if we now run this on 0100?

$$
\begin{array}{llllllll}
q_0 0100 & \rightarrow & Bq_1 00 & \rightarrow & B1q_2 00 & \rightarrow & Bq_3 110 & \rightarrow \\
q_3 B110 & \rightarrow & Bq_0 110 & \rightarrow & BBq_5 10 & \rightarrow & BBBq_5 0 & \rightarrow \\
BBBBq_5 B & \rightarrow & BBBBBq_6 & & & & & \\
\end{array}
$$

We now see that when the Turing Machine halts, it produces a tape with only blank spaces left. This falls in line with what we expect. Because we fed into the machine an n value larger than m, we should get zero back, or no 0s on the tape leftover.

3.3 Importance

Despite their simplicity, Turing machines are able to model the capability of a computer. Which means given any computer algorithm, theoretically there exists a Turing Machine that can be constructed to model it. For a machine that only has two components, this is a very powerful tool. We know this as the *Church-Turing Thesis*, such that any real-world computation can be modeled or translated into an equivalent computation using a Turing machine [6].

It is not just computers that can be modeled with a Turing Machine. Turing Machines gave computer scientists a way to categorically associate programming languages based off their capabilities. A programming language is considered *Turing-complete* if a language can replicate all properties to simulate a Turing machine, sans the unlimited memory (the tape reel). For example, languages like Python or Java can be considered Turing-complete, however languages like SQL and HTML are not.

While the Turing machine gives mathematicians and computer scientists a way to model computers and programs, Turing originally designed this machine to assist him to prove, or better yet disprove, a famous problem in complexity theory.

4 Undecidability

4.1 *Entscheidungsproblem*

Decision problems are integral in computability theory and complexity theory and rely on the ability to pose questions as yes-no problem. One of the most popular decision problems is *Entscheidungsproblem*. *Entscheidungsproblem*, German for "decision problem", is a problem posed by renowned mathematician David Hilbert in 1928. The problem states if one is given a set of axioms and mathematical propositions, is there an effective algorithm to prove whether any statement is or is not provable from the axioms [7]? That is, as referenced from one of Hilbert's problems posed in his 1900 paper, *Mathematische Probleme*, are all problems in mathematics decidable?

Hilbert's *Entscheidungsproblem* was long unanswered until mathematicians like Alan Turing and Alonzo Church solved and proved it to be impossible. In Turing's 1936 paper, he addresses the proof of this problem with the use of his newly defined computing machines. This problem using Turing machines became famously known as *The Halting Problem*.

4.2 The Halting Problem

In reference to Hilbert's question, we can rephrase the problem in a similar way: Given an arbitrary program and input for the program, can we determine if there is an output? As we discussed earlier, we can model any computing algorithm as a Turing machine. By the nature of these machines, if a Turing machine runs on an input, there are two different outcomes that may occur: either the machine halts and produces an output, or the machine continues to run forever. With the given clarifications, we can now reformulate the question as the Halting Problem.

The Halting Problem. *Is there a Turing machine that can determine if any Turing machine will eventually halt based off an input fed into the machine?*

We know that there is no machine that can make this statement possible, but how did Turing prove this[4]?

Proof. Let's assume there is a Turing machine, M, that gives solutions to the Halting Problem. For program p and input i, we can say that M works as such:

[4]Proof of the Halting Problem is adapted from Reference [5].

$$M(p,i) = \begin{cases} 1 & \text{if } p \textit{ halts} \text{ on } i \\ 0 & \text{if } p \textit{ runs forever} \text{ on } i \end{cases}$$

Because a program is just a sequence of instructions, we can also use that as an input into a Turing Machine as well. So now let's define a new machine, M' that takes in a program, and uses M. we can define M' like so:

$$M'(p) = \begin{cases} \textit{halts} & \text{if } M(p,p) = 0 \\ \textit{runs forever} & \text{if } M(p,p) = 1 \end{cases}$$

What happens now if we use M' as an input? And more specifically, what happens if we use M' as an input into itself? Let's consider the two scenarios that may occur.

1. $\boldsymbol{M'(M')}$ ***halts.*** If this is the case, then $M(M', M') = 0$. This means, by definition of M, M' *runs forever* on M'.

2. $\boldsymbol{M'(M')}$ ***runs forever.*** If this is the case, then $M(M', M') = 1$. This means, by definition of M, M' *halts* on M'.

In both scenarios, no matter what state we assume $M'(M')$ is in, the opposite also occurs, creating a paradox. Something can not be both terminated and running at the same time. Therefore, this is a contradiction to the assumption and there can not exist such a machine M that can solve the halting problem for all programs and inputs. □

4.3 *Entscheidungsproblem* and Other Undecidable Problems

The Halting Problem shows us the essence of undecidable problems, for which it is impossible to determine yes or no answers to the problem. Using the Halting problem as a basis, Turing also proved that *Entscheidungsproblem* is impossible and cannot have a solution either.

Over time, we discovered more problems that can be *reduced* down to the Halting Problem, meaning that solving problem A is just as hard as solving the Halting Problem. Take for instance Goldbach's Conjecture, which states all even integers greater than 2 can be represented as a sum of two prime numbers. The most straightforward way to prove this is to devise an algorithm that runs every even number and finds two pairs of primes that sum to the value, and it will only

stop once it finds a suitable even number that will not work. But due to the undecidability of the problem, we will not know when the algorithm will stop or if it ever will. Similarly, the same argument can be framed for a method to solving the Collatz Conjecture.

We also see the Halting Problem in many forms of our day to day lives. Decrypting RSAs is one of them. If there is a computational way to crack RSA encryptions, services like phones, bank accounts and all internet accounts would be in danger. Because of RSA complexity, cracking these encryptions would be pose to be a Halting Problem. In addition, the Halting Problem can be related to fields like cyber security. Let's say there was an anti-virus solution out on the market, will there ever come a time when the software actually executes malicious programs?

5 Moving Forward

Turing Machines and the Halting Problem open up more problems in the realm of complexity theory. One of the Millennium Prize Problems is the *P* vs. *NP* problem, a major unsolved computer science complexity problem. The problem states if it is easy to check if a problem is correct (NP), can you also easily find a solution to the problem (P) [1]. This idea can be written as $P \stackrel{?}{=} NP$ [3].

How does this relate to the Halting Problem? When we take a closer look, P and NP problems are all decidable, when the Halting Problem is not. The Halting Problem is *NP-hard*, which means solving this problem is as hard as an NP problem. This means that all the problems that we encountered previously have also been classified as NP-hard [9]. The difference in these categorizations help us understand the scope of a certain solution. While the Halting Problem is impossible, for different problem, if it is reducible down to the Halting problem, we can spend time searching for other practices and methodologies must be used to lead us closer to a solution.

Furthermore, In Stephen Cooks paper on *P* vs. *NP*, Turing's discovery of his machines laid the foundations to computable functions, which are used for the computability of problems [1, 10]. Turing Machines was the abstract basis to modern day computers and computer science, and hence the *P* vs. *NP* spawned from learning about the complexity and computability of these machines.

6 Conclusion

While relatively simple with easy to understand mechanics, the Turing Machine is an extremely powerful model. It can model any modern day computer and any computable algorithm that can be created. Alan Turing was truly a spearhead in modern computer science as he laid the foundation for many fields in computer science. And while he did prove the Halting Problem to be impossible, this led to more discoveries and undecidable questions in computer science.

References

1. Cook, Stephen. "The P vs. NP Problem." From
2. Encylcopedia Britannica. "ENIAC." Encyclopedia Entry.
3. "The Halting Problem | P vs. NP." From Kalamazoo University, CS 105.
4. Hopcroft, John E. "Introduction to Automata Theory, Languages, and Computation." Print. Cornell University.
5. Kaplan, Craig S. "The Halting Problem." From The University of Waterloo.
6. Rowland, Todd. "Church-Turing Thesis." From MathWorld–A Wolfram Web Resource, created by Eric W. Weisstein.
7. Stovicek, Jan. "Axioms, algorithms, and Hilbert's Entscheidungsproblem." From NTNU, Norwegian University of Science and Technology.
8. Turing, Alan. "On Computable Numbers with an Application to the Entscheidnungsproblem", Proc. London Math.
9. Weisstein, Eric W. "NP-Hard Problem." From MathWorld–A Wolfram Web Resource.
10. Weisstein, Eric W. "Computable Function." From MathWorld–A Wolfram Web Resource.