Computability, its Proof by Construction, and the Halting Problem

Oluwafunke Alliyu, Ethan Balcik, Paul John Balderston, Sen Zhu

April 2021

1 Introduction

Our world's infastructure is entirely reliant upon digital computers. Despite their ubiquity, the high levels of abstraction at which such devices are typically used leads the majority of users to have little to knowledge of their underlying functionalitiy. It is in the spirit of elucidating the working of computational devices that we discuss the most essential topic in the theory of computation, the computatability of functions. We begin by providing a list of requisite definitions nessacry to define and describe mechanisms by which one may determine if a function is computable. We introduce the reader to the Turing Machine, a conceptual model of an all purpose computer. Once familiarzing the reader with the aforementioned concepts we invoke the use of a Turing Machine to rigorously prove of the computability of an even weight verification function for a binary input string. We conclude with a discussion of the Halting Problem in order to further extend our discussion to noncomputable functions. [**1**]

2 Background

It is essential to understanding the following definitions of elementary set theory prior to the discussion of Turing machines and their functionality.

Definition 2.1. A **set** is a well-defined collection of objects. The objects which a set is composed of are referred to as its **elements** or **members**.

- $a \in S$ represents that a is 'in' S, or that a is a elements of S
- $a \notin S$ represents that a is not an element of S
- $\varnothing$ represents an empty set

Definition 2.2. Consider the sets X and Y with $y \in Y$ and $x \in X$ where y and x are arbitary elements. If every element $y \in Y$ is such that $y \in X$, then Y is a **subset** of X, which may be written as $Y \subseteq X$. If Y is a subset of X, but there exsists at least one $x \in X$ for which $x \notin Y$ then Y is a **proper subset** of X, which we write as $Y \subset X$.

- Any set X contains the subsets $X \subseteq X$ and $\varnothing \subseteq X$, which are distinct unless X is empty.

Definition 2.3. The **Cartesian product** $X \times Y \coloneqq \{(x, y) : x \in X \text{ and } y \in Y\}$, is defined to be the set of all ordered pairs such that $x \in X$ and $y \in Y$.

- The Cartesian product of n sets, where $n \in \mathbb{N}$, is given as $X_1 \times X_2 \times ... \times X_n \coloneqq \{(x_1, x_2, ..., x_n) : x_i \in X_i \text{ for each } i = 1, 2, ..., n\}$
- For a given set X and some aribitrary $n \in \mathbb{Z}_{\geqslant 0}$, the nth **Cartesian power** of X is defined by $X^n \coloneqq X \times ... \times X = \{(x_1, ..., x_n); x_1, x_2, ..., x_n \in X\}$, and it is the cartesian product of X with itself taken n times.

Definition 2.4. A **function** $f \subseteq X \times Y$, such that each $x \in X$ is mapped to exactly one $y \in Y$, forming the 2-tuple (or ordered pair) $(x, y) \in f$

[**2**]

3 Turing Machines

The Turing Machine is a general-purpose computing device capable of performing any given algorithmic computation [**1**]. While the model itself is an abstraction, any instantion of a Turing Machine, physical or platonic, is comprised of the following:

- A "control box" which stores a program of finite size, formally known as the **transition function**.
- A tape with a potentially infinite number of memory spaces in which symbols can be stored, read, and written into each indivdua spacel.
- A read-write mechanism which the tape is fed through. [**3**]

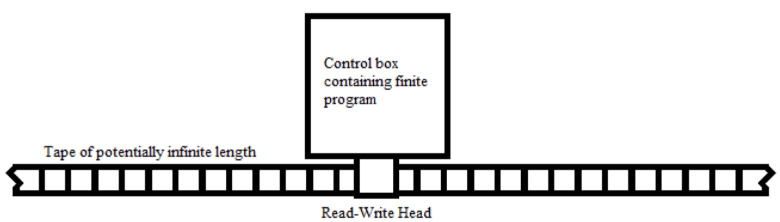

Figure 1: A simple visualization of a Turing machine. Note that spaces along the tape would be filled with symbols which can be read and written using the read-write head on the machine.

Prior to presenting the formal definiton of a Turing Machine we must introduce the following items.

Definition 3.1. An **alphabet** is a finite set of arbitrary, distinct characters which can be used in some code or language.

For example, the english alphabet (ignoring all punctuation and special characters) may define its alphabet as $A_{english} \coloneqq \{a, b, ..., z\}$

Definition 3.2. A **word** $\vec{x}$ is a string of n characters $a_1a_2 \dots a_n$, such that for every $a_i \in \vec{x}$, a_i is a member of an alphabet A.

Following from the previous example, we may construct words of varying lengths using the english alphabet, such as "the", "dog", "was", and "running". Each character composing each of these words belong to the alphabet $A_{english}$ defined above. [**4**]

Definition 3.3. A **language** L is a subset of the set of all possible words $\vec{x} \in L$ of any length over an alphabet A.

Any combination of English characters belonging to the English dictionary will certainly be a member of the language $L_{english}$ which is defined on the alphabet $A_{english}$. [**5**]

Having introduced the items above we may now develope a formal definiton of the Turing Machine.

Definition 3.4. A **Turing Machine** is a 7-Tuple, $(Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$, where:

- Q is a finite set containing the states of the machine
- Σ is a finite set containing the machine's **input alphabet**
- Γ is the finite set containing th emachine's **tape alphabet** such that the **blank symbol** $\sqcup \in \Gamma$ and $\Sigma \subseteq \Gamma$
- δ is the **transition function** $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$
- q_0 is the **starting state** $q_0 \in Q$
- q_{accept} is the **accept state** $q_{accept} \in Q$
- q_{reject} is the **reject state** $q_{reject} \in Q$ such that $q_{reject} \neq q_{accept}$ [**2**]

Definition 3.5. A **configuration** is a string of characters representing the position of a Turing machine's read-write head along its tape, as well as the characters on the tape. It is written as a string of the tape's characters listed starting from the left-most character and working right, with the state $q_n \in Q$ inserted to the left of the character currently being read by the read-write head of the machine.

Definition 3.6. A **computation** is a sequence of configurations which represents the sequence of transition function operations run by a Turing machine given some input word $\vec{x}$. [**6**]

To exemplify these definitions, we provide an instantiation of a Turing Machine with specific input words, and an algorthim running on the input words.

Example 3.1. Let's imagine a Turing machine running an algorithm which decides whether a binary word of length $n \in \mathbb{N}$ has an even weight. In other words, it verifies whether the number of 1s in the binary word is even. Our Turing Machine will execute the following algorithm:

1. Read the first bit of the string
2. If the first bit of the string is a '0', transition to state q_3.
3. If the first bit of the string is a '1', transition to state q_2.
4. While in states q_2 and q_3, transition to the other state if a '1' is read, remain in current state if '0' is read.
5. If in state q_2 and reading '␣' character, terminate in state q_{reject}
6. If in state q_3 and reading '␣' charactre, terminate in state q_{accept}.

This particular Turing machine may be given formally as

$$M \coloneqq (Q, \Sigma, \Gamma, \delta, q_1, q_{accept}, q_{reject})$$

such that:

- $Q \coloneqq \{q_1, q_2, q_3, q_{accept}, q_{reject}\}$
- $\Sigma \coloneqq \{0, 1\}$
- $\Gamma \coloneqq \{0, 1, \text{␣}\}$
- The transition function δ is given as the following instruction set

$$\begin{gathered}
\delta(q_1, 0) = (q_3, 0, R) \\
\delta(q_1, 1) = (q_2, 1, R) \\
\delta(q_1, \text{␣}) = (q_3, \text{␣}, R) \\
\delta(q_2, 0) = (q_2, 0, R) \\
\delta(q_2, 1) = (q_3, 1, R) \\
\delta(q_2, \text{␣}) = (q_{reject}, \text{␣}, R) \\
\delta(q_3, 0) = (q_3, 0, R) \\
\delta(q_3, 1) = (q_2, 1, R) \\
\delta(q_3, \text{␣}) = (q_{accept}, \text{␣}, R)
\end{gathered}$$

Using the transition function δ as given above, we can show every configuration for a given input string. For the sake of the example, let's consider the string $\vec{x} = 1101$. We have the following configurations (read down each column, and then from left to right):

$$q_1 1101$$
$$1 q_2 101$$
$$11 q_3 01$$
$$110 q_3 1$$
$$1101 q_2 \llcorner$$
$$1101 \llcorner q_{reject}$$

Since our input string has an odd number of '1' characters our Turing machine terminartes in state q_{reject}. Thus, $\vec{x}$ is not of even weight.

Let us now walk through each configuration of the even-weighted input string $\vec{x} = 1001$. We have the following configurations:

$$q_1 1001$$
$$1 q_2 001$$
$$10 q_2 01$$
$$100 q_2 1$$
$$1001 q_3 \llcorner$$
$$1001 \llcorner q_{accept}$$

Now, since our input string has an even number of '1' characters, our Turing machine terminates in the state q_{accept}. [**1**]

4 Proof of Computability by Construction

As observed in **Example 3.1**, it is apparent that when a Turing machine reaches states q_{accept} or q_{reject}, its algorithm will terminate. However, it is possible when running an algorithm using a Turing machine that the algorithm may never terminate. It is this possibility, the possibility that a Turing machine may run indefinitely for some input, from which much of the theory of computability emerges. Here, we define the formal notion of Turing decidability, we discuss its relationship with computability, and we develop a proof of computability for our previous instantiation of a Turing machine.

Definition 4.1. A language L is **Turing-decidable** if there exists some Turing machine M such that, for each input word $\vec{x} \in L$, M terminates in either state q_{accept} or q_{reject} [**1**]

Turing decidability relates to computability by considering the Church-Turing thesis. Essentially, one interpretation of the thesis states that if one wishes to prove that a certain operation is computable, one can do so by constructing a Turing machine which terminates for all possible inputs into that operation [**6**]. To display this, we will prove that the binary 'even weight verification' function introduced in **Example 3.1** is a computable function.

Proof 4.1. First, let us note that the input alphabet accepted by our Turing machine (its definition given in **Example 3.1**) $\Sigma \coloneqq \{0, 1\}$. This would imply that the **language** L emergent from this alphabet is simply the set of all binary strings (or **words**) of any length $n \in \mathbb{N}$. We enumerate this language using $\mathbb{N}^2$. We use $\mathbb{N}^2$ as opposed to simply $\mathbb{N}$ because we note that, for example, the strings $\vec{y} = 000101$ and $\vec{z} = 101$ are different inputs entirely, and may be handled differently by our Turing machine, even though their decimal values are identical. Therefore, we allow one degree of freedom to represent the weight of the arbitrary input word $\vec{x}$, and the other to represent the decimal value of the arbitrary input word $\vec{x}$ written in reverse order (i.e. $\vec{y} = 000101$ written in reverse order is 101000, and its decimal value is 40). Using this fact, we can partition our input language into four unique subsets, and engage in a 'proof by cases' on an arbitrary member of each such subset.

Case 1: Assume $\vec{x}$ is a binary string of even weight and leading character '0'. Observe that $\delta(q_1, 0) = (q_3, 0, R)$, therefore the Turing machine will enter state q_3 after observing the input string's leading '0' character. Note that since the Turing machine has yet to observe a '1' character, there still exist $2n : n \in \mathbb{Z}_{\geqslant 1}$ '1' characters remaining in the input string. Also note that $\delta(q_3, 0) = (q_3, 0, R)$, meaning that the Turing machine will remain in state q_3 until it inevitably observes a '1' character. Since $\delta(q_3, 1) = (q_2, 1, R)$, any time the Turing machine observes a '1' character from state q_3, it will transition to state q_2 as a result. The above is true with regard to state q_2 as well, and the Turing machine will transition from state q_2 to state q_3 upon observation of a '1' character. Since we start in state q_3, and there are $2n$ '1' characters remaining in the input string, we can expect this transition between states q_3 and q_2 to take place a total of $2n$ times, leaving the Turing machine in state q_3 reading the character $\llcorner\!\lrcorner$. The Turing machine will terminate in state q_{accept} since $\delta(q_3, \llcorner\!\lrcorner) = (q_{accept}, \llcorner\!\lrcorner, R)$ as expected.

Case 2: Assume $\vec{x}$ is a binary string of even weight and leading character '1'. Observe that $\delta(q_1, 1) = (q_2, 1, R)$, therefore the Turing machine will enter state q_2 after observing the input string's leading '1' character. Note that since the Turing machine has observed one '1' character, there still exist $2n - 1 : n \in \mathbb{Z}_{\geqslant 1}$ '1' characters remaining in the input string. As noted in **case 1**, this means that the Turing machine will inevitably make $2n - 1$ transitions between state q_2 and q_3, leaving the Turing machine in state q_3 reading the character $\llcorner\!\lrcorner$. The Turing machine will terminate in state q_{accept} since $\delta(q_3, \llcorner\!\lrcorner) = (q_{accept}, \llcorner\!\lrcorner, R)$ as expected.

Case 3: Assume $\vec{x}$ is a binary string of odd weight and leading character '0'. Observe that $\delta(q_1, 0) = (q_3, 0, R)$, therefore the Turing machine will enter state q_3 after observing the input string's leading '0' character. Note that since the Turing machine has yet to observe a '1' character, there still exist $2n + 1 : n \in \mathbb{Z}_{\geqslant 0}$ '1' characters remaining in the input string. As noted in **case 1**, this means that the Turing machine will inevitably make $2n + 1$ transitions between state q_3 and q_2, leaving the Turing machine in state q_2 reading the character $\llcorner\!\lrcorner$. The Turing machine will terminate in state q_{reject}

since $\delta(q_2, \llcorner\!\lrcorner) = (q_{reject}, \llcorner\!\lrcorner, R)$ as expected.

> **Case 4:** Assume $\vec{x}$ is a binary string of odd weight and leading character '1'. Observe that $\delta(q_1, 1) = (q_2, 1, R)$, therefore the Turing machine will enter state q_2 after observing the input string's leading '1' character. Note that since the Turing machine has observed one '1' character, there still exist $2n + 1 - 1 = 2n : n \in \mathbb{Z}_{\geqslant 0}$ '1' characters remaining in the input string. As noted in **case 1**, this means that the Turing machine will inevitably make $2n$ transitions between state q_2 and q_3, leaving the Turing machine in state q_2 reading the character $\llcorner\!\lrcorner$. The Turing machine will terminate in state q_{reject} since $\delta(q_2, \llcorner\!\lrcorner) = (q_{reject}, \llcorner\!\lrcorner, R)$ as expected.

Therefore, the Turing machine will terminate as expected for any given input. □

5 The Halting Problem

In the 20th century, Alan Turing used similar concepts from the theory of computation to prove that first-order logic is, in fact, non-decidable. This was a major question in the field of mathematical logic for some time, and Turing was among one of the first to prove this fact. In order to prove this fact, Turing showed that there exists no finite program which terminates for every possible input belonging to the language of first-order logic. The proof that Turing constructed is known commonly as **the halting problem**, and it goes as follows. [**7**]

Proof 5.1. We begin by instantiating a Turing machine $M \coloneqq (Q, \Sigma, \Gamma, \delta, q_1, q_{accept}, q_{reject})$ which accepts as its input parameters a program P such that any character $p \in P$ is also a member of the alphabet Σ, specifying some arbitrary Turing machine's transition function, and a particular input string for some arbitrary Turing machine running the program P which we'll call I. Assume that M determines whether the transition function given by P terminates for the input I, itself terminating in state q_{accept} if P terminates for the input I, and terminating in state q_{reject} if not. Finally, and most importantly as we work toward a contradiction, assume that M itself terminates for all possible inputs P and I. To begin the proof, we generate a new Turing machine by transforming M in such a way that simple logic is appended to the end of M's program. We say that, if the new Turing machine, which we'll call M', enters state q_{accept}, it should loop forever, and if M' enters state q_{reject}, then it should terminate in state q_{accept}. We now feed M' into itself with M' and some arbitrary string I' as its input.

> **Case 1:** If the program M' terminates on input M' and I', then M' will loop forever, leading to a contradiction.

Case 2: If we assume the program M' loops forever on input M' and I', then M' will terminate, also leading to a contradiction.

Therefore, if we assume that there exists some Turing machine M which determines whether an input program given its input terminates, then we are led to a contradiction. Thus, there exists no Turing machine which can decide whether some arbitrary program will terminate for some arbitrary input. [**8**] $\square$

6 Conclusions

Throughout this paper we have provided a breif introduction to the essentials of the theory of computation. We give a formal definition of the Turing machine, and we emphasize its signafcance as the mechanism by which one may prove if a function is indeed computable. We rigorously prove that a Turing machine can decide whether the weight of an arbitrary binary string is even by showing that there exists at least one Turing machine which terminates as expected for any given binary string input. Following our proof of computablity, we present the Halting Problem and its proof in order to exemplify how one might prove that a language is not Turing decidable. The above discussions results are but a small window into the theory of computation. These abstract mathematical ideas underly all computations and are the fundamental underpinings of our computational world. Computer science continues to evolve both as an engineering descipline and a theoretical pursuit. Nevertheless, it's modern instantation began with the conception of the Turing Machine and a proof showing that there are some functions which are and are not computable.

References

[1] Sipser, Michael. (2013). Introduction to the Theory of Computation. Cengage Learning. Third Edition. Print.

[2] Humphreys, J.F. (2012). Numbers, Groups and Codes. Cambridge University Press. Web. https://doi-org.ezaccess.libraries.psu.edu/10.1017/CB09780511812187

[3] Mainzer, Klaus. (2018). Proof of Computation: Digitization in Mathematics, Computer Science, and Philosophy. World Scientific. https://doi.org/10.1142/11005

[4] Hill, Raymond. (1986). A First Course in Coding Theory. Oxford University Press. Web. http://www.math.unipa.it/ difranco/documents/Hill.AFirstCourseIn CodingTheory.pdf

[5] Cornell University. (2012). Introduction to Algorithms: Notes on Turing Machines. Web. http://www.cs.cornell.edu/courses/cs4820/2012sp/handouts/turingm.pdf

[6] Evans, David. (2010). Church-Turing Thesis. University of Virginia. Web. http://www.cs.virginia.edu/ evans/cs3102-s10/classes/class15/class15.pdf

[7] Jago, Mark. (2014). Turing and The Halting Problem - Computerphile. YouTube. https://www.youtube.com/watch?v=macM_MtS_w4

[8] Pruhs, Kirk. (1997). Halting Problem - Simple Proof. Web. https://www.comp.nus.edu.sg/ cs5234/FAQ/halt.html