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

Elementary set theory is of crucial importance in the understanding of the formal notion of computibility. Before diving into the concept of computability, one must understand the relevant prerequisite concepts in elementary set theory which enable the various definitions in theoretical computer science definitions to be given.

Definition 2.1. A **set** is a well-defined collection of objects, known as its **members**.

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

Definition 2.2. If each element $y \in Y$ is such that $y \in X$, then y is a **subset** of x, which can be written as $Y \subseteq X$. If Y is a subset of X, but there exsists at least one $x \in X$ such that $x \notin Y$ then we say that Y is a **proper subset** of X, which can be written as $Y \subset X$.

- Any set X has 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 more than two sets may be 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 set X and some $n \in \mathbb{Z}_{\geqslant 0}$, 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 essentially the cartesian product of X by itself n times.

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

- Note that the sets X and Y above are arbitrary, opening the possibility for a function to be potentially defined from the cartesian product of multiple arbitrary sets to the cartesian product of multiple arbitrary sets.

3 Turing Machines

An effective model for the general-purpose computer is the Turing Machine, developed by Alan Turing in 1936 [**2**]. The Turing Machine is a conceptual model of a general-purpose computing machine which involves the following conceptual components:

- A "control box" which stores a finitely-large program

- A tape with infinite spaces in which symbols can be stored, read, and written

- A read-write mechanism for the tape [**3**]

Definition 3.1. 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** $\textvisiblespace \in \Gamma$ and $\Sigma \subseteq \Gamma$

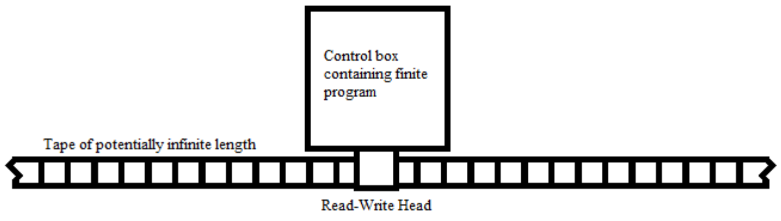
Control box containing finite program
Tape of potentially infinite length
Read-Write Head

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.

- 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 $q_{reject} \neq q_{accept}$

A **configuration** represents the state of a Turing machine's read-write head along its tape, as well as the characters on the tape. For feasibly-sized tape inputs, a configuration is given 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. To exemplify this, we can introduce a new instance of a Turing Machine with a specific input, and an algorithm running on the input.

Example 3.1. Let's imagine a Turing machine running an algorithm which verifies whether or not a binary string of length n has an even weight (number of 1s in the binary string). It might achieve this by running the following algorithm:

1. Read each bit from left to right on the input string and cross off every other '1' character.
2. If in step 1 the tape contained no '1' characters, accept.
3. If in step 1 the tape contained a single '1' character, reject.
4. Return to the left-most character on the tape.
5. Return to step 1.

This particular Turing machine can be given formally as $M \coloneqq (Q, \Sigma, \Gamma, \delta, q_1, q_{accept}, q_{reject})$ such that:

- $Q \coloneqq \{q_1, q_2, q_3, q_4, q_5, q_{accept}, q_{reject}\}$
- $\Sigma \coloneqq \{0, 1\}$

- $\Gamma \coloneqq \{0, 1, \sqcup\}$
- The transition function δ is given as an instruction set (see figure 2)

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

Figure 2: The transition function δ given as an instruction set

Using the transition function δ as given by figure 2, 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):

$$\begin{gathered}
q_1 1101 \\
1 q_2 101 \\
11 q_3 01 \\
110 q_3 1 \\
1101 q_2 \sqcup \\
1101 \sqcup q_{reject}
\end{gathered}$$

We can clearly see that, since our input string has an odd number of '1' characters, our turing machine rejects it, signifying that it does not have an even weight. Next, let us walk through each configuration of the even-weighted input string $\vec{x} = 1001$. We have the following configurations:

$$\begin{gathered}
q_1 1001 \\
1 q_2 001 \\
10 q_2 01 \\
100 q_2 1 \\
1001 q_3 \sqcup \\
1001 \sqcup q_{accept}
\end{gathered}$$

Now, since our input string has an even number of '1' characters, our turing machine accepts it. [**2**]

4 Proof of Computability by Construction

The outcomes observed in **Example 3.1** are rather self-evident in the definition of a Turing machine, as two of its parameters are the accept state q_{accept} and the reject state q_{reject}. If a Turing machine ever reaches either of these states, then its algorithm will terminate. However, a third potential outcome when running an algorithm using a Turing machine is 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. In this section, we will build the definitions which found the theory of computability, and explore how we may prove that a function is computable.

Definition 4.1. An **alphabet** is a finite set of arbitrary 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 4.2. A **word** $\vec{x}$ is a string of characters, each of which belonging to some 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.

Definition 4.3. A **language** L is the set of all possible words $\vec{x} \in L$ of varying length, over some alphabet A.

Any combination of english characters imaginable will certainly be a member of the language $L_{english}$ which is defined on the alphabet $A_{english}$ mentioned previously.

Definition 4.4. A language L is **Turing-decidable** if there exists some Turing machine M such that, for each input word $\vec{x} \in L$, M either accepts or rejects it. [**2**]

The above definitions are those which found much of the theory of computability. It does so by use of the Church-Turing thesis, and the wealth of empirical evidence backing it, albeit there is no single, rigorous mathematical proof for this 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 [**4**]. 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 arbitrary length $n \geqslant 1$. Thus, our input language can be enumerated using $\mathbb{N}^2$. We must 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 will be handled differently by our Turing machine, even though their decimal values are identical. Therefore, we allow one degree of freedom to represent the length of the arbitrary input word $\vec{x}$, and the other to represent the decimal value of the arbitrary input word $\vec{x}$. 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 $\sqcup$. The Turing machine will terminate in state q_{accept} since $\delta(q_3, \sqcup) = (q_{accept}, \sqcup, R)$ 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 $\sqcup$. The Turing machine will terminate in state q_{accept} since $\delta(q_3, \sqcup) = (q_{accept}, \sqcup, R)$ 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 $\sqcup$. The Turing machine will terminate in state q_{reject} since $\delta(q_2, \sqcup) = (q_{reject}, \sqcup, R)$ 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. $\square$

5 The Halting Problem

Alan Turing, during the 20th century, used similar concepts from computability theory 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 program 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. [**5**] $\square$

6 Conclusions

Here we finally discuss the conclusions of the paper

References

[1] National Research Council. (1999). Funding a Revolution: Government Support for Computing Research. Washington, DC: The National Academies Press. https://doi.org/10.17226/6323.

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

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

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

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

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

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