Computability, its Proof by Construction, and the Halting Problem

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

April 2021

1 Introduction

In the age of digital communications, it is difficult to imagine how, in a world without digital computers, one might work toward the development of one. As a result, often overlooked are the initial developments which led to today's digital age. Nevertheless, the emergence of digital computers would not be possible without initial, groundbreaking developments in mathematical logic and information theory from significant engineers and mathematicians like Alan Turing and Claude Shannon. Throughout this paper, we discuss computability in a rigorous, self-contained manner - both its proof by construction and its disproof by contradiction, with examples of each. Furthermore, we aim to provide our readers with some historical insight into the development of these methods in order to build both an appreciation of their significance, and of the current challenges researchers face as attempts are made to further progress in theoretical computer science [**1**].

2 Background

Elementary set theory and Mathematical logic are of crucial importance to the understanding of the formal notion of computibility. Before diving into the Turing Machine that essentially tackles the problems of solvability and insolvability in Mathematics, one needs to be introdued to these prerequisites beforehand.

Definition 2.1. A **set** is a collection of objects, known as its **elements/members**.

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

Definition 2.2. If every element of Y is also an element of X, then we say that y is a **subset** of x, and write $Y \subseteq X$. If y is a subset of X but not equal to X, then we say that y is a **proper subset** of X, and write $Y \subset X$.

- Every set X contains at least the subsets X and $\varnothing$; and these are distinct unless X = $\varnothing$.

Definition 2.3. The **Cartesian product** of two sets X and Y is defined to be the set of all ordered pairs whose first entry comes from X and whose second entry comes from Y.

- $X \times Y \coloneqq \{(x, y) : x \in X \text{ and } y \in Y\}$
- The cartesian product of a set X with itself is often denoted as X^2.
- Cartesian product of more than two sets can be defined in a similar way $X_1 \times X_2 \times X_3 \times ... \times X_n \coloneqq \{(x_1, x_2, x_3, ..., x_n) : x_i \in X_i \text{ for each } i = 1, 2, 3, ..., n\}$
- For a set X and positive integer n, the **Cartesian power** of the power n is defined by $X^n \coloneqq X \times ... \times X = \{(x_1, ..., x_n); x_1, x_2, ..., x_n \in X\}$

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**]

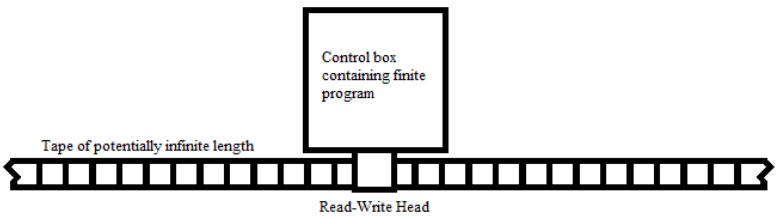

Figure 1: A simple visualization of a Turing machine. Note that spaces along the tape would be filled with symbols which can be read and written using the read-write head on the machine.

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 $\Sigma \subseteq \Gamma$
- 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:

- $Q \coloneqq \{q_1, q_2, q_3, q_4, q_5, q_{accept}, q_{reject}\}$
- $\Sigma \coloneqq \{0, 1\}$
- $\Gamma \coloneqq \{0, 1, x, \textvisiblespace\}$
- The transition function δ is given as an instruction set (see figure 2)

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):

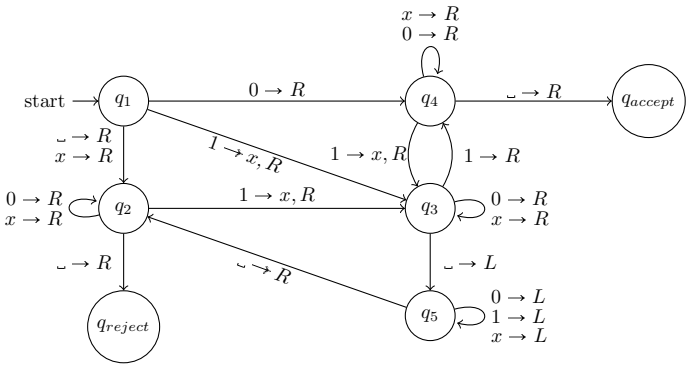

Figure 2: The transition function δ given as a finite state machine

$q_1 1101$	$x1q_5 0x$	$xxq_3 0x$	$q_5 xx0x$	$xx0xq_2\sqcup$
$xq_3 101$	$xq_5 10x$	$xx0q_3 x$	$q_5\sqcup xx0x$	$xx0x\sqcup q_{reject}$
$x1q_4 01$	$q_5 x10x$	$xx0xq_3\sqcup$	$q_2 xx0x$	
$x10q_4 1$	$q_5\sqcup x10x$	$xx0q_5 x$	$xq_2 x0x$	
$x10xq_3\sqcup$	$q_2 x10x$	$xxq_5 0x$	$xxq_2 0x$	
$x10q_5 x$	$xq_2 10x$	$xq_5 x0x$	$xx0q_2 x$	

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:

$$q_1 1001$$
$$xq_3 001$$
$$x0q_3 01$$
$$x00q_3 1$$
$$x00xq_4\sqcup$$
$$x00x\sqcup q_{accept}$$

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: Let $\vec{x} \in L$ be an arbitrary-length input string with even weight, and starting with the '0' character. Our Turing machine will begin in state q_4. Since we know that there are an even number of '1' characters in $\vec{x}$, we can expect our Turing machine to 'bounce' back and forth between states q_4 and q_3 an even number of times, crossing off every other '1' character with an 'x' character, until ultimately producing some configuration $\ldots q_4\sqcup$. This will finally yield the configuration $\ldots\sqcup q_{accept}$, and the Turing machine will terminate in state q_{accept} as expected.

Case 2: Let $\vec{x} \in L$ be an arbitrary-length input string with even weight, and starting with the '1' character. Our Turing machine will begin in state q_3 with the first '1' character crossed off. Since we know that there are an even number of '1' characters, and that we have already crossed off one of these characters, we can expect our Turing machine to 'bounce' back and forth between states q_3 and q_4 an odd number of times, crossing off every other '1' character with an 'x' character. Ultimately, this will produce the configuration $\ldots q_4\sqcup$, which will finally yield the configuration $\ldots\sqcup q_{accept}$, and the Turing machine will terminate in state q_{accept} as expected.

Case 3: Let $\vec{x} \in L$ be an arbitrary-length input string with an odd number of '1' characters, and starting with the '1' character. Our Turing machine will begin in state q_3 with the first '1' character crossed off. Since we know that there are an odd number of '1' characters, and that we have already crossed off one of these characters, we can expect our Turing machine to 'bounce' back and forth between states q_3 and q_4 an even number of times, crossing off every other '1' character with an 'x' character. Ultimately, this will produce the configuration $\ldots q_3\sqcup$, which will trigger the q_5 state, eventually returning the system to the q_2 state with the read-write head reading the first character of the string. Note that, since the input string had an odd weight, but had the first '1' character crossed off before the initial loop between states q_3 and q_4, it now has an odd number of '1' characters crossed off, and thus, an even number of '1' characters remaining. From here, the machine will proceed with this process recursively until it crosses off all '1' characters, and is left with the configuration $q_2\ldots$, where q_2 will inevitably sweep through the entire string, producing the configuration $\ldots q_2\sqcup$. As a result, the Turing machine will transition to the configuration $\ldots\sqcup q_{reject}$ as expected for our odd-weighted input string.

Case 4: Let $\vec{x} \in L$ be an arbitrary-length input string with odd weight, and starting with the '0' character. Our Turing machine will begin in state q_4. Since we know that there are an odd number of '1' characters, we can expect our Turing machine to 'bounce' back and forth between states q_4 and q_3, crossing off every other '1' character with an 'x' character. Ultimately, this will produce the configuration $\ldots q_3\sqcup$, which will trigger the q_5 state, eventually returning the system to the q_2 state with the read-write head reading

the first character of the string. Note that, since the input string had an odd weight, and since the transition between q_4 and q_3 took place an odd number of times starting with q_4, then in the current state with q_2 back at the beginning of the input string, we have the original input string, but with an odd number of '1' characters crossed off, and thus, an even number of '1' characters remaining. From here, our Turing machine will engage in an identical recursive process as given in **case 3**, meaning that it will terminate in state q_{reject} as expected.

Terefore, the binary 'even weight verification' function is computable, as it terminates for all input words $\vec{x} \in L$. $\square$

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**] □

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