Euler Circle ToC Expository Paper: Computable Functions

Agastya Goel

December 15, 2022

1 Introduction

When we think of a function not being computable, we think of some obtuse, contrived function designed specifically to not be computable. However, the Busy Beaver function shatters this expectation. It was first conceived by Tibor Radó in 1962 and is defined as follows.

Definition 1.1. Consider an n-state Turing machine T, operating over the language $\{0, 1\}$. Let the *halting time* of T be the number of steps T makes before halting, on an input tape filled with 0s. If T doesn't halt on such an input, its halting time is defined as 0. Then, $BB(n)$ is defined as the maximum halting time over all Turing machines with n states. This is well-defined since there are only finitely many Turing machines of a given size.

Theorem 1.2. *Computing the Busy Beaver function is, computationally, at least as difficult as solving the halting problem.*

This function is, perhaps, the most famous uncomputable function. We call this function uncomputable because there exists no Turing machine which recursively checks if $BB(n) = k$. In fact, the value of $BB(5)$ still isn't known. At first glance, it may seem strange that such a simple, well-defined function might be uncomputable. But Turing machines can be used to solve many problems: there exists a Turing machine which halts iff the Goldbach conjecture is false, and a similar one for the Riemann hypothesis. From an intuitive perspective, we shouldn't expect to be able to bound the running time of solving these problems. This paper gives a proof that the Busy Beaver function is uncomputable and provides intuition for why this should be true. We also explore the implications of the Busy Beaver function in regards to Zermelo-Fraenkel set theory with the axiom of choice (ZFC).

In addition, we consider the general topic of computable functions. We present a variety of results that are surprising on the surface level, but are truly intuitive and share a deep connection with Turing machine decidability. We start with early attempts to classify computable functions, such as primitive recursive functions, but we then show that such attempts were unsuccessful. We also extend this result to any recursive definition of computable functions, showing that no recursive framework can encompass all computable functions. Finally, we circle back to the halting problem, and discuss its relationship with computability.

2 The Computational Difficulty of the Busy Beaver Function

In this section, we will provide a finite-time reduction from the halting problem to computing the Busy Beaver function.

Proof of Theorem 1.2. As aforementioned, we will provide a reduction from the halting problem to computing the Busy Beaver function. This means that the halting problem can be no more difficult than the Busy Beaver function's computation, since we can use the Busy Beaver function to solve the halting problem.

Consider some Turing machine B which accepts the input (n, m) iff $BB(n) = m$. Then, we can construct the following machine H, which accepts the input (M, w) iff M halts on the input w. To do this, we begin by converting M into some machine M' over the language $\{0, 1\}$. To do this, we can replace each letter in our language into some string of 0s and 1s. Specifically, if we have k letters, we require strings of length

$\lceil \log_2(k) \rceil$. Then, whenever we want to make a transition, we can read our sequence and store it in extra states, then return to our original position, remembering the information. We can also perform transitions in a similar way, by overwriting one bit at a time. All of these operations can be performed by expanding our state space, since $\lceil \log_2(k) \rceil$ is constant. Finally, we create a new machine D_w, which does the following. First, it writes the input sequence w onto the tape. Then, it returns to the beginning, and runs M'.

Now, let n' be the number of states of D_w. The output of D_w on a tape consisting of infinite 0s must be the same as M run on w. Then, using B, find $BB(n')$. Then, we can run D_w for $BB(n')$ steps, and if it doesn't halt at that point, we can reject. Hence, we have constructed a solution to the halting problem using the Busy Beaver function. □

Corollary 2.1. *The Busy Beaver function is not computable.*

3 The Equivalence of the computation of the Busy Beaver function and the Halting Problem

In Theorem 1.2, we showed that computing the Busy Beaver function is at least as hard as the halting problem. In this section, we will show the converse, indicating that the two problems are equivalent.

Theorem 3.1. *The halting problem is, computationally, at least as difficult as computing the Busy Beaver function.*

Proof. As done previously, we will show this with a reduction argument, reducing the Busy Beaver function to the halting problem. Given a Turing machine H which recursively solves the halting problem, we can construct the Turing machine B, which accepts on the input (M) iff the Turing machine M runs on a tape of 0s for time $BB(|M|)$, where $|M|$ denotes the number of states in M.

To do this, we first compute $BB(|M|)$. This can be done by enumerating all Turing machines with size $|M|$, of which there can only be a finite amount. Then, for each machine M', we run H on the input M', and if it indicates that M' does halt, we run M' until it halts. We can then take the maximum over all of these halting times and check if M halts at the same time. □

This gives the following corollary:

Corollary 3.2. *The Busy Beaver function is computationally equivalent to the halting problem. More formally, any Turing machine with an oracle for the halting problem can compute the Busy Beaver function, and vice-versa.*

We can also say:

Lemma 3.3. *The Busy Beaver function grows faster than any other computable function.*

Proof. If there was some computable function that grew faster than the Busy Beaver function, we could use it to bound the Busy Beaver function, and thus, use it to solve the halting problem. Hence, no computable function grows faster than the Busy Beaver function. □

This fact should also make intuitive sense. Computing a function, from a Turing machine sense, means that there is a constant size Turing machine which produces different output depending on the size of the input. It seems incredibly natural that the Busy Beaver function should grow faster than anything computed by a Turing machine of a fixed size because, in a sense, the Busy Beaver function encodes the largest value of a function represented by n states.

4 Uncomputable Numbers

Previously, we have shown that the busy beaver function isn't computable. This means that there exists no Turing machine which can check whether $BB(n) = k$, for all given n and k. This statement is equivalent to saying that we cannot prove whether $BB(n) = k$ for all n, k under ZFC, since the computation history of a Turing machine is a valid proof. Now, consider the following theorem from Gödel:

Theorem 4.1 (Gödel's Second Incompleteness Theorem)**.** *If ZFC is consistent (i.e. both a statement and its negation cannot be proved simultaneously), it cannot prove its own consistency.*

Nearly all of modern mathematics is built upon the assumption that ZFC is consistent, so we will assume its consistency. Recall that we can encode a proof in a Turing machine. Specifically, we can construct a Turing machine that halts iff ZFC is inconsistent. However, though we presume that the machine wouldn't halt, we cannot *prove* it. Thus, the key to finding a Turing machine whose halting is uncertain is to build a Turing machine that confirms the consistency of ZFC. This would, in turn, show that a given value of BB is uncomputable in ZFC.

Constructing a specific Turing machine to check whether ZFC is consistent isn't hard. Since any false statement is equivalent to $0 = 1$, we use the axioms of ZFC to recursively enumerate all true statements, halting if we derive the statement $0 = 1$. Other methods can be used to reduce the size of the resulting Turing machine, such as compiling a program from a higher-level language into a Turing machine, which can then be read and interpreted. These methods allow a Turing machine with 748 states which verifies the consistency of ZFC. Hence, we cannot compute $BB(748)$ under ZFC, leaving only a two order of magnitude gap between the largest value of the Busy Beaver function we can calculate (4) and the smallest value which we definitely cannot calculate (748).

5 Other Fast Growing Functions

One of the largest well-known computable functions is the Ackermann function. The Ackermann function extends well-known operations such as addition, multiplication, exponentiation, and tetration. Formally, it can be defined recursively as follows:

$$A(0, n) = n + 1$$
$$A(m + 1, 0) = A(m, 1)$$
$$A(m + 1, n + 1) = A(m, A(m + 1, n)).$$

For an idea of how fast this function grows, $A(1, 1) = 3$, $A(2, 2) = 7$, $A(3, 3) = 61$, and $A(4, 4) = 2^{2^{2^{65536}}} - 3$.

While bigger functions can be described, the Ackermann function is one of few practical large functions. It's most common application is in the union find (disjoint set union) algorithm, which has an amortized complexity of $O(nA^{-1}(n))$. From its definition, it is clear that the Ackermann function is computable. However, the Ackermann function lies outside of a class of recursive functions known as primitive recursive functions.

Definition 5.1. A function $f(x_1, \ldots, x_n)$ is *primitive recursive* if either:

1. f is always 0, i.e.
$$f(x_1, \ldots, x_n) = 0.$$
This can be denoted by simply Z if the number of arguments is implicit, and we call this function the Zero function.

2. f is the successor function, i.e.
$$f(x_1, \ldots, x_n) = x_i + 1.$$
This rule for deriving a primitive recursive function is called the Successor rule.

3.
$$f(x_1, \ldots, x_n) = x_i.$$
In this case, we call f the projection function, and similarly, deriving functions this way is known as the Projection rule.

4. Composing primitive recursive functions also results in a primitive recursive function. Specifically,

$$f(x_1, \ldots, x_n) = h(g_1(x_1, \ldots, x_n), \ldots, g_n(x_1, \ldots, x_n))$$

is primitive recursive if $h, g_1, g_2, \ldots, g_n$ are all primitive recursive. This rule is called the Composition rule.

5. The final production rule is called the Recursion rule, and it is what gives primitive recursive functions their name. It is defined recursively, with

$$f(x_1, \ldots, x_{n-1}, 0) = g(x_1, \ldots, x_{n-1})$$

and

$$f(x_1, \ldots, x_{n-1}, m+1) = h(x_1, \ldots, x_{n-1}, m, f(x_1, \ldots, x_{n-1}, m))$$

where both g and h are primitive recursive. This rule allows for most of the complexity of primitive recursive functions.

Most natural functions can be expressed in this form. For example, addition consists of repeated applications of the primitive recursive successor operation. Once we've established this, multiplication can be expressed as repeated addition. Further operations, such as primality, evaluating polynomials, and more all turn out to be primitive recursive. In fact some intepretations of the Ackermann function are also primitive recursive. If we look at simply the rows or columns of the Ackermann function, i.e. setting either m or n to a constant, this new, single-variable function is primitive recursive. However, the function we are truly interested in, $f(k) = A(k, k)$ isn't primitive recursive, for reasons beyond the scope of this paper. This shouldn't be surprising. Since the Ackermann function requires nested recursion ($A(m+1, n+1) = A(m, A(m+1, n))$), it seems reasonable that we can't express it using a very basic set of production rules.

6 Computable Functions

This brings us back to the topic of computable functions. Primitive recursive functions were one of the first attempts to classify computable functions, but the Ackermann function showed that they didn't work. However, any similar attempt would be doomed to a similar failure.

Theorem 6.1. *Consider the following procedure for producing functions. Let us call the set of functions we aim to produce R. We begin with some set of base functions, which can be determined by a Turing machine and add them to R. Then, we will be given both recursive and nonrecursive production rules. A nonrecursive production rules gives a way to combine functions in R to produce a new function f which also must be in R. This means that the definition of f cannot reference f itself. In addition, we can consider some number of recursive production rules, in which the definition of f references itself. Then if no recursive production rule results in an infinite loop of function calls, the set of functions R cannot contain all computable functions.*

Proof. First note that the set of computable functions is countable, and that R can only generate computable functions. With this in mind, let us order the functions in R. This we can do by implementing the production rules on a FIFO queue, which will guarantee that every function in R is eventually generated. Then, let ϕ_x be the x^{th} function generated. We will then define $g(x) = \phi_x(x) + 1$. Since we can generate ϕ_x and compute it recursively, g is computable, but it differs from every function in R, and hence R doesn't contain all computable functions. □

The above proof uses a diagonalization argument. However, we have to be very careful when using such an argument. The specific requirements we imposed on the construction of R ensured that every recursive function call would eventually halt. Without this property, the diagonalization argument wouldn't work since we couldn't guarantee that $g(x)$ would actually halt.

This idea of halting is central to such arguments. One such thought experiment is to consider ordering all Turing machines. We know this can be done, and by definition, every computable function is represented in this ordering of Turing machines. However, we cannot apply such a diagonalization to this ordering since we can't know whether a given Turing machine halts.

7 An Alternate Proof to the Halting Problem

The insights in the previous section give us the following alternate proof of the halting problem.

Theorem 7.1. *There exists no halting Turing machine T which on all pairs S, ω of Turing machine and input can correctly determine if the machine S halts on ω.*

Proof. Consider ordering Turing machines, which we know can be done. Then, consider a machine T which solves the halting problem. We will then derive a contradiction by constructing a machine N which isn't equivalent to any machine in our ordering, which is impossible.

On an input x, N begins by computing the x^{th} Turing machine ϕ_x. It then uses T to determine if ϕ_x halts on x. If it does halt, N outputs the result of running ϕ_x on the input x plus one. Otherwise, N outputs 0. Since this new machine always halts and differs from every Turing machine in our ordering, it must not be in our ordering. Thus we have a contradiction and T cannot always halt. □

8 References

On Non-Computable Functions.
Yedidia and Aaronson, 2016.
A Turing machine that verifies the consistency of ZFC with 748 states.
Classes of Recursive Functions.