BASIC COMPUTABILITY

JAIME SEVILLA

CONTENTS

Date: December 2015.

1. Introduction

In this article we present some basic notions of computability. The study of what is doable by theoretical machines in idealized conditions gives us a better insight of the limits of actual machines.

A function is effectively computable if there is a procedure of finitely many steps that gives us the value of the function for every input. This is not a formal notion, but it is tightly related to the concept of Turing Machines, as we proceed to see.

2. Turing Computability

A Turing Machine is comprised by a tuple (Γ, Q, δ). Γ is a finite alphabet, which through this article will be $\{0,1\}$. Q is a finite set of states. with two special states, START and HALT. δ a transition function $Q \times \Gamma \to Q \times A$, also called its program. A is the set of actions available in each step of the computation. In this article they will be A = 0,1,L,R, which correspond to writing a 0, writing a 1, moving left and moving right, respectively.

Example: Program of a TM computing the successor function

```
START, 0 -> HALT , 1
START, 1 -> START, L
```

The machine is over a cell in an infinite 1D tape, which is full with 0s but for a finite number of cells. In every step of the computation, the machine reads its input and applies its transition function to do an action and change its internal state. The computation ends when the machine reaches the HALT state.

Our standard interpretation of computation of a given TM M with an input $< a_0, ..., a_n >$ is as follows. We start on a blank tape (all 0s) except for the input, given in unary. The machine is placed on the leftmost 1 of the tape, and in the state START. We will call that an standard configuration.

Example: Tape encoding the input $< 3, 2 >$: ...00111011000...

Then the transition function is applied sequentially, until the HALT state is reach, if ever. If the machine started in the standard configuration $< a_0, ..., a_n >$ reaches HALT in the standard configuration $< b_0, ..., b_m >$, we will say that such a TM computes a partial function f such that $f(a_0, ..., a_n) = (b_0, ..., b_m)$. If it halts in a non-standard configuration, or never halts, then the function computed by the TM is not defined on that input.

We will say that a partial function is Turing Computable if there is a TM that agrees with the function on every input.

Note that TM are enumerable, since they are finite words over a finite alphabet. We will denote the ith TM for a given enumeration as M_i.

Thesis 1 (Church-Turing Thesis). Every effectively computable function is Turing computable.

2.1. **Exercises.**

(1) Program whatever you fancy! Check out this link[1] to start writing your own TMs. Some ideas: the maximum of two numbers, the sum of two numbers.

[1]https://github.com/Jsevillamol/PythonUTM

3. Beyond The Realm of Turing Computability

3.1. **The Diagonal Function.** Since the cardinal of possible TMs is enumerable but the cardinal of $\mathbb{N}^{\mathbb{N}}$ is non-enumerable there must be a non-enumerable quantity of functions from N to N that are not computable by a TM.

Furthermore, given an encoding of the TM, and interpreting their behaviour as the computation of a function from N to N, we can construct such an uncomputable function:

$$f(n) = \begin{cases} 1 \; if \; M_n(n) = 0 \\ 0 \; if \; M_n(n) \text{ is undefined or defined and greater than } 0 \end{cases}$$

We have defined f in such a way that it disagrees with every function computed by every TM in at least the entry corresponding to its own code number. Thus f is not computable.

3.2. **The Halting Problem.** The diagonal function is not of particular interest because of its artificiality. We can however think what would we need to compute the diagonal function. Clearly, if the machine M_n halts, we can determine $f(n)$ by simply running it and outputting the opposite answer of what we are given. So the problem is with machines that we are not sure whether they will halt or continue forever. Define:

$$HALT(m,x) = \begin{cases} 1 \; if \; M_n(x) \text{ halts} \\ 0 \; if \; M_n(x) \text{ does not halt} \end{cases}$$

If we could compute $HALT$, then we could run $HALT(n,n)$, and simulate M_n if $HALT(n,n) = 1$. Otherwise, we know that the partial function computed by M_n is undefined on n, without a need to simulate it at all. Thus we have shown that the diagonal function is reducible to the Halting Problem; that is, diagonal is computable if $HALT$ is computable. As we know that diagonal is not computable, we have shown the following:

Proposition 2 (The Halting Theorem)**.** *HALT is not computable.*

3.3. **BB-like functions.** The Busy Beaver-like functions are some functions describing undecidable properties of k-states TM. The first one, productivity, was introduced by Tibor Radó.

Definition 3 (Scoring Function). The scoring function s is defined for every n as the greater output produced by a n-state TM for input n.

Proposition 4. *The scoring function is not computable.*

Proof. Suppose it was computable. Then we could append to the TM computing s another state that computes the successor function. Such a machine would have k states for some k, and on input k would produce s(k)+1, which is absurd. ■

Definition 5 (Productivity Function). The productivity function p is defined for every n as the greater output produced by a n-state TM when started on a blank tape.

Let us examine some properties of p.

Proposition 6. *(1)* $p(1) = 1$
(2) $p(n) > p(m) \iff n > m$
(3) $\exists\, i \; st \; p(n+i) > 2p(n)$ *for all n.*

Proposition 7. *There is no computable function that grows faster or as fast as the productivity function. That is, for every computable function f, $f(n) \geq p(n)$ for at most a finite number of n.*

Proof. Suppose there was a computable function f st there exist n_0 st for all $n > n_0$ we have that $f(n) \geq p(n)$. If M is a j-state TM computing f, we can build a machine that writes n in a tape, and then runs M twice, computing $f(f(n))$. Such a machine would have $n+2j$ states. Choosing $n > n_0$, as p is strictly increasing, we have that $p(n+2j) \geq f(f(n)) \geq p(f(n)) \geq p(p(n))$.

That implies that $n+2j \geq p(n)$. Letting $n = m+i$, $m+i+2j \geq p(m+i) \geq 2p(m) \geq 2m$. And that must hold for every m except for maybe a finite quantity, which is absurd. ■

Corollary 8. The productivity function is not computable.

3.4. Rice's Theorem.

Definition 9. A set of computable partial functions is called trivial if it is either the empty set or it contains every partial computable function.

3.5. Exercises.

(1) Is it there a TM such that it halts iff started on a tape which is not totally blank? And one that halts if the tape is totally blank?

Theorem 10 (Rice's Theorem)**.** *Let S be a non trivial set of computable partial functions. Then the problem of deciding whether an arbitrary TM M computes a function in S is uncomputable.*

Proof. Suppose there was a nontrivial set S for which we could decide if a given machine computes a function in S. We will show we can use such a procedure to compute the Halting Problem, which we know is uncomputable.

Since S is non-trivial, we can suppose wlog that the empty function that is undefined for every function does not belong to S, while there is a computable function defined in at least one input n that belongs to S.

Now, suppose we are given a machine M_α and an input x, and we want to decide whether $M_\alpha(x)$ halts. We can build a machine that on input n will first simulate $M_\alpha(x)$ and then output $f(n)$. Such a machine will compute f iff $M_\alpha(x)$ halts, and the empty function otherwise. So we can test whether the machine computes a function in S to decide if $M_\alpha(x)$ halts. ■

References

[1] George S. Boolos, John P. Burgess, Richard C. Jeffrey, *Computability and Logic*

[2] S. Arora, B. Barack *Computational Complexity: A Modern Approach*