Internally uncomputable functions, with application to the P versus NP problem.

Gerard Westendorp

September 21, 2019

Abstract

We clarify the concept of uncomputable functions, by considering a machine M_1 with bounded computation time. We show that important functions, for example one that decides the Halting problem for M_1, are *externally* computable (by a larger machine that can simulate M_1), but not *internally* computable (by M_1 itself). A similar line of reasoning leads us to understand when diagonal arguments for demonstrating P $\neq$ NP relativise. We attempt to prove that P $\neq$ NP, and are for the moment able to give a lower bound for deciding problems in NP.

1 Introduction

The halting problem for Turing machines with unlimited memory and time is undecidable. As explained by Turing [1] in his 1936 paper, this is related to undecidable propositions in logic. It seems at first somewhat perplexing. It gives us the feeling that perhaps there is something wrong with our understanding of logic itself.

Part of the difficulty for our intuition lies with the unlimited size and allowable computation time of the machine considered. The case of a machine with bounded computation time and bounded memory (We will call this a "bounded machine") is easier to understand. The Halting problem is in that case decidable by a function $H_{ex}(\text{“}P''\text{)}$ on a larger machine M_{ex}, that is able to simulate M_1, run **P** for the bounded time of M_1, and check if **P** halts. We call H_{ex} an *externally* computable function. However, as we will show, the Halting problem is not decidable by an *internally* computable function, one that can run on M_1 itself. In fact, we will show:

Theorem 1 (Bounded time Halting theorem) *There is no computable function on a bounded machine M_1 that can determine in less than T_1 steps if programs on M_1 halt within T_1 steps.*

The essential feature that causes undecidability (at least in the cases considered in this article) is the inability of an internal computation to do things

that only an external computation can. This observation helps us to better understand a number of important questions. In particular, we suggest we might use it to prove that P is not NP.

Theorem 2 ( ) *There is no computable function G on a bounded machine M_1 that can determine in less than $T_1 - d$ steps if functions (P) on M_1 with a bounded computation time of T_1 steps, have an input such that their output equals "1". T_1 is a polynomial function of the size of the input of P, and d a small positive integer.*

Our method is very similar to the "diagonal" arguments used by Turing to prove his theorem on the Halting problem, except we have to take special care with regards to the computation time, and the use of oracles.

Definition 1 ("P'' of **P)**

We will be considering programs **P**, that implement a 1-output bit function P(N_1) of N_0 input bits, where $N_1 < 2^{N_0}$. "P'' will denote the *description* of the program It does not matter for our argumentation if it is machine or source code, or whether we treat "P'' as a string or integer)

2 The bounded time halting problem (proof of theorem 1)

On a Machine M_1 a bounded allowed computation time T_1 is externally imposed. We could choose T_1 such that it is equal to the (very large) number of possible internal states of M_1. If M_1 has not halted by then, it never will. But our argument works for any T_1.
We will call programs whose computation time is limited to T_1 *internal* programs of M_1.

Lemma 1 (bounded time Halting theorem) *There is no computable function on a bounded machine M_1 that can determine in less than (T_1–d) steps if programs on M_1 halt within T_1 steps, where d is a small positive integer, independent of T_1.*

Let **H** be a program implementing the function $H($"P'') that takes as input the description "P" of an internal program **P** such that:

Definition 2 (Program **H** implementing $H($"P'')**)**

$$\begin{cases} H(\text{“}P''\text{”}) = 1 \text{ if } P(\text{“}P''\text{”}) \text{ halts in } T_1 \text{ steps} \\ H(\text{“}P''\text{”}) = 0 \text{ if } P(\text{“}P''\text{”}) \text{ does not halt in } T_1 \text{ steps} \end{cases}$$

Assume $\mathbf{H}$ halts in time T_2.

Next define program $\mathbf{H}_2$, implementing the function $H_2(\text{“}P''\text{)}$:

Definition 3 (Program $\mathbf{H}_2$ with input “P'')

$$\begin{cases} \\ \text{if } H(\text{“}P'') = 1 \text{ , then } \mathbf{H}_2 \text{ loops forever.} \\ \\ \text{if } H(\text{“}P'') = 0 \text{ , then } \mathbf{H}_2 \text{ halts.} \\ \\ \end{cases}$$

The output of H_2 is not used.
$\mathbf{H}_2$ is basically as copy of $\mathbf{H}$, with a conditional "Halt" instruction. For our argument to work, $\mathbf{H}$ must halt in $T_1 - d$ steps, where d is the time needed for H_2 to do a conditional halt Typically, $d = 1$ (The code for entering an endless loop need not be completed, since all it must accomplish is keep busy until T_1).

We can now show that $\mathbf{H}$ cannot exist if $T_2 <= T_1 + d$:

If $H(\text{“}H_2'') = 1$, then according to definition 2, $H_2(\text{“}H_2'')$ halts in T_1 steps. But according to definition 3, this would imply that $H(\text{“}H_2'') = 0$.

If $H(\text{“}H_2'') = 0$, then according to definition 2, $H_2(\text{“}H_2'')$ does not halts in T_1 steps. But according to definition 3, this would imply that $H(\text{“}H_2'') = 1$.

This concludes the proof of Lemma 1

Our arguments so far do not rule out the existence of an algorithm $\mathbf{H}_a$ with runtime T_{2a}, such that $T_1 - d \leq T_{2a} \leq T_1$. $\mathbf{H}_a$ would use up almost all allowed computation time, except for the time needed for 1 conditional instruction. Let $\mathbf{P}_1$ be a program for which $\mathbf{H}_a$ takes time T_{2a} to decide. Define $\mathbf{P}_2$ and $\mathbf{P}_3$, such that:

$\mathbf{P}_2$: execute $\mathbf{P}_1$; if(1+1=2) then Halt.
$\mathbf{P}_3$: if(1+1=2) then execute $\mathbf{P}_1$.

If $\mathbf{P}_1$ halts, then in analysing $\mathbf{P}_2$, $\mathbf{H}_a$ must also do an extra check that condition (1+1=2) halts. If $\mathbf{P}_1$ does not halt, then in analysing $\mathbf{P}_3$, $\mathbf{H}_a$ must also do an extra check (1+1=2) to see if it needs to execute $\mathbf{P}_1$.

So if $\mathbf{H}_a$ would exist, we could construct programs from it which take longer than T_1. QED theorem 1.

2.1 Discussion

As soon as any method for analysing programs is expressible as a program with a fixed description, then there will be programs sufficiently sophisticated to use

this description to "fool" it. These sophisticated programs however, cannot fool an *external* function. This distinction becomes confusing for machine with unlimited capacity.
For a bounded machine, the Halting problem is not so mysterious. Functions can be uncomputable, simply because there is not enough time to compute them.

2.2 The bounded time halting problem in the presence of an oracle

Suppose we equip M_1 with an oracle (A) that implements **H** on M_1 by some special hardware, using only 1 computational step. Call this modified machine M_{1A}. Such an oracle would be very useful for determining if programs halt on M_1. If the oracle is external, meaning that programs other than H on M_1 cannot use A, then the Halting problem on M_1 is be decidable. If however, the oracle is internal, meaning that all programs on M_{1A} are free to use the oracle, then the proof of theorem 1 relativises: it remains valid regardless of the presence of an oracle. This argumentation is already known for showing that oracles do not make the halting problem for unbounded time decidable.

3 The P versus NP problem (proof of theorem 2)

We again consider a bounded machine M_1. This time we consider programs **P** implementing functions $P(N_1)$ that have N_0 input bits, where $N_1 < 2^{N_0}$ is an integer which represents the input bits. We ask whether there exists an input N_1, such that $P(N_1) = 1$. This is an NP complete problem, because it is equivalent to the boolean satisfiability (nSAT) problem. This time we set T_1 to some polynomial function of N_0. We will cal T_1 the *verification* time.

Let **G** be a program that takes as input a description "P'' of an internal program P such that:

Definition 4 (Program **G** implementing the function $G(\text{“}P'')$)

$$\begin{cases} G(\text{“}P'') = 1 \text{ if } \exists N_1 \, (P(N_1) = 1) \\ G(\text{“}P'') = 0 \text{ if } \nexists N_1 \, (P(N_1) = 1) \end{cases}$$

Assume **G** halts in time T_2, also polynomial in N_0. We will cal T_2 the *analyses* time.

Define program $\mathbf{G}_2$, implementing the function $G_2(N_1)$:

Definition 5 (Program $\mathbf{G}_2$ implementing the function $G_2(N_1)$)

$$\begin{cases} \\ \text{if } G(\text{“}G2'') = 1 \text{ , then output } 0 \\ \text{if } G(\text{“}G2'') = 0 \text{ , then output } 1 \\ \\ \end{cases}$$

Note that G_2 ignores its input N_1, it just needs to have this input so that **G** can attempt to analyse it. Its computation time is slightly longer than **G**. But if **G** is polynomial, then so is **G2**. (It is a bit confusing to talk about the computation time of a program that can not exist.)
$\mathbf{G}_2$ Needs to supply its own description to the function G. This is possible, due to Rogers's fixed-point theorem [5]. In principle, Rogers's theorem does not exclude the possibility that $\mathbf{G}_2$ becomes some very large program. We therefore include an actual construction of $\mathbf{G}_2$ in Visual basic, given that the function G exists.

```
Function G_2(N1)
'(The self-generated code excludes all comments)
' First, put info needed for generating the code in a string array
a = Array("Function G_2(N1)", "a = Array(", "b = Chr(44)", _
"c = Chr(34)", "d = Chr(41)", "e = Chr(13)", _
"G_2_Code$ = a(0) & e & a(1)", "For i = 0 To 16", _
"G_2_Code$ = G_2_Code$ & c & a(i) & c & b", _
"Next i", "G_2_Code$ = G_2_Code$ & c & a(i) & c & b & d", _
"For i = 0 To 16", "G_2_Code$ = G_2_Code$ & c & a(i)& e", _
"Next i", "If G(G_2_Code$) = 1 Then G_2 = 0", _
"If G(G_2_Code$) = 0 Then G_2 = 1", "End Function")

'Generate the code of G_2 as a string
b = Chr(44) 'comma character
c = Chr(34) 'quote character
d = Chr(41) 'close brackets character
e = Chr(13) 'return character
G_2_Code$ = a(0) & e & a(1)
For i = 0 To 16
G_2_Code$ = G_2_Code$ & c & a(i) & c & b
Next i
G_2_Code$ = G_2_Code$ & c & a(12) & c & b & d
For i = 1 To 16
G_2_Code$ = G_2_Code$ & a(i) & e
Next i

'The actual call to G by G_2
If G(G_2_Code$) = 1 Then G_2 = 0
If G(G_2_Code$) = 0 Then G_2 = 1
End Function
```

The internal version **G** cannot exist, as we will show:

If $G(\text{“}G_2'') = 1$, then according to definition 4, there is a N_1 such that $G_2(N_1) = 1$. But according to definition 5, G_2 always outputs 0.

If $G(\text{“}G_2'') = 0$, then according to definition 4, there is no N_1 such that $G_2(N_1) = 1$. But according to definition 5, G_2 always outputs 1.

So $\mathbf{G}_2$ and **G** cannot exist.

An external version of **G**, $\mathbf{G}_{ex}$, exists, that works by simulating **P** for all possible values of N_1. This program will need a computation time exponential in N_0. This is not excluded by the above proof, because $\mathbf{G}_2$ does not have enough computation time to include a call $\mathbf{G}_{ex}$.

We also cannot exclude a version of **G2** that has a polynomial analyses time T_2, but with a greater polynomial than the maximum verification time T_1. On

the other hand, if **G2** claims to be able to decide our problem within some polynomial upper bound, we seem free to choose our verification upper bound to a greater polynomial than the one claimed by **G2**. That would contradict the existence of **G2**.

3.1 Discussion

The reason that a program like **G** cannot exist, is that it claims something very ambitious: to predict what other all other programs in its own class will do. Its Achilles heel is that it is internal. This means the other programs can call it. They have a much easier job. They only need to exist *if* **G** exists, and then outsmart it.

We feel our arguments come quite close to proving $P \neq NP$, and we are currently working to achieve further progress in this direction.

4 Diagonal arguments in the presence of an oracle

In their 1975 paper, Baker Gill Solovay [3] demonstrate some seemingly discouraging results on trying to settle the P versus NP question with diagonal arguments, such as we presented here. They argue that if the proof works also in the presence of an oracle for **G** (the proof "relativises"), then the conclusion that no algorithm exists would contradict the very definition of the oracle. Analogous to the bounded time halting problem, if we give both **G** and the programs **P** access to an oracle (A) turning **G** in to $\mathbf{G}^A$, and **P** into $\mathbf{P}^A$ we have an *internal* oracle, and the proof relativises: The oracle is only useful for analysing problems that do not themselves use the oracle. One can still construct the relavitised $\mathbf{G}_2^A$, that $\mathbf{G}^A$ cannot solve.

Suppose $\mathbf{G}^A$ is dealing with the Boolean satisfiability problem, instead of with programs **P**. The Boolean satisfiability problem is formulated in terms of Boolean variables, and Boolean expressions the variables must satisfy. If the problem is to have the same oracle as the solver, we must build the oracle in terms of Boolean expressions, using auxiliary Boolean variables, in the fashion described by Cook [2] in his 1971 paper. If we attempted to do this for $\mathbf{G}_{ex}$, we would need an amount of extra variables exponential in N_0. In that case the proof does not relativise.

So either

- the problem can also use the oracle (the oracle is internal) and the proof relativises. This is not a contradiction, because the oracle is no longer effective when problems themselves use the oracle.
- the problem can not use the oracle (the oracle is external) and the proof does not relativise. In that case $P^A = NP^A$. This fact however is of no use for practical computation; when we actually try to implement the oracle with the same tools that the problem is allowed to use, it will require exponential computation time.

References

[1] Turing, A.M. *"On Computable Numbers, with an Application to the Entscheidungsproblem"*. Proceedings of the London Mathematical Society, 1937.

[2] Cook, Stephen. *The complexity of theorem proving procedures* Proceedings of the Third Annual ACM Symposium on Theory of Computing, 1971.

[3] Baker, Theodore, John Gill, and Robert Solovay. *Relativizations of the P=?NP question* SIAM Journal on computing, 1975.

[4] Radó, Tibor. *On non-computable functions* Bell System Technical Journal, 1962.

[5] Rogers, H. *Theory of Recursive Functions and Effective Computability* MIT Press, 1967.