The equality P = NP on programs with infinite length as another barrier to the proof of P ≠ NP

Evgeny Kuznetsov
eak@mail.ru

Abstract

Things become different at infinity. A school example - if you count the number of even numbers up to 100, there are half as many of them as all numbers. And at infinity, every natural number corresponds to an even number. And it turns out that there is an equal quantity of each of them. Without limiting the quantity of numbers, it is impossible to mathematically prove that there are fewer even numbers than all numbers. A similar story is observed in complexity theory. In this paper, using a lazy Turing machine, it is proved that for programs with infinite length, the maximum complexity is **O(n)**. And one of the consequences of this fact is that **P = NP** at infinity. And because of this, as one of the consequences of the last statement, it is impossible to prove that **P ≠ NP** without limiting the program's length. In time hierarchy theorem, diagonalization implicitly limits the program's length. We need a similar trick to keep progress going.

1. Introduction

The alleged inequality **P ≠ NP** is deeply embedded in human culture. It takes a long exponential (**NP**) time to find a good answer to a question. But once a good solution is found, it can be very quickly (**P)** transferred and used. For example, numbers have been known since at least the 4th millennium BC. And the modern understanding of zero developed only in the 5th century AD. Schoolchildren learn in one lesson what humanity has spent several thousand years to understand. One of the reasons for our communication is to save time by getting successful answers found by others. But there is still no mathematical proof that it is not always possible to quickly find the answer by yourself. There is a great mathematical formulation in Stephen Cook's "The P vs. NP Problem" [1]. Additionally, it is worth reading Scott Aaronson's review "P ?= NP" [2].

No proof that **P ≠ NP** has been found yet. The progress lies in the fact that approximately every 15 years there pops up another mathematical reason why this formula is so difficult to prove. Such reasons are called barriers. We know three barriers, or rather three proof schemes insufficient to prove that **P ≠ NP**: relativizing proofs [3], natural proofs [4], algebraic proofs [5]. We will prove below that no proof scheme will work until it uses the length of a program.

The point is that there are programs with **O(n)** complexity that accept languages from **NP** class even though their size is infinite. The set of states and transition function are finite by formal definition of Turing machine, but they can be extended to infinity through lazy evaluations. Here is the list of theses introduced in this paper:

1. **DC** (Decision Tree) shows how to achieve **O(n)** for any input word
2. **LTM** (Lazy Turing Machine) shows how to get a program of infinite size
3. **PSB** (Program Size Barrier): the upper bound of an infinite program is **O(n)**
4. Infinite programs exist in any Turing complete computing models
5. Infinite programs on an **LTM** are a countable set
6. **P = NP** when program size is unlimited
7. We cannot prove **P ≠ NP** (aka **P** ⊂ **NP**) with no restriction on program size
8. We can prove that **P ≠ NP** is unprovable, but it may be not proven for a limited program size
9. Subpolynomial proofs are allowed (such as **AC0** ⊂ **NC1**)
10. Inclusions and equalities are allowed ("simple" **P** ⊆ **NP**, "impossible" **P** ⊂ **NP**)
11. Proofs involving size are allowed (as diagonalization in **P** ⊂ **EXPTIME**)

2. What is **P** and what is **NP**

Formally **P** and **NP** are classes of languages.

Definition 1. Language **L** is a subset of all possible finite strings **Σ**∗ containing at least two elements **Σ**.

Definition 2. A Turing machine **M** is a tuple ⟨**Σ, Γ, Q, δ**⟩, where **Σ** is the input word alphabet**, Γ** (**Σ** ⊆ **Γ**) is the tape alphabet**, Q** is the state set**, δ (δ : (Q − {qAccept, qReject}) × Γ → Q × Γ × {−1, 1})** is the transition function [1]. The term program can be used hereafter as a synonym for the set of states and values of the transition function.

Definition 3. The given language **L** is a member of class **P** if there exists a corresponding Turing machine **M** that accepts any string **x** from **L** in the number of steps limited by some polynomial function **f(k)**, where **k** is the length of the input string **x**. [1]

Definition 4. The given language **L** is an element of class **NP** a) if there exists a corresponding Turing machine **M** b) for any string **x** from **L** there exists a string "witness" **w** from some language **L1** c) **M** takes (**x, w)** in the number of steps limited by some polynomial function **f(k)**, where **k** is the length of string **x**. [2]

Lemma 1. Any language **L** included in **P** is also included in **NP**.

Proof. We take machine **M** from the definition of **P**, **w** zero-length string is used as a "witness" . Machine **M** accepts **(x, w)** in polynomial time. So **L** belongs to **NP** class.

3. **DT** (Decision Tree) with **O(n)** complexity in finite languages

It is possible to obtain a **DT** with **O(n)** complexity through a divide-and-conquer tactic: we take each input symbol and proceed further down to the corresponding subtree. Then only one more step will be required for each input symbol.

For example, let us define the input alphabet as **Σ = {a, б}** and language as **L = {a, бa, бб}**, **b** as a blank symbol. And the transition function of Turing machine will be defined as:

Input		**Output**	
State	**Tape symbol**	**Tape operation**	**Next**
q0	b		qReject
q0	a	R	q1
q0	б	R	q2
q1	b		qAccept
q1	a		q3
q1	б		q3
q2	b		qReject
q2	a	R	q4
q2	б	R	q4
q3	b		qReject
q3	a		qReject
q3	б		qReject
q4	b		qAccept
q4	a		qReject
q4	б		qReject

This **DT** is built in a regular manner for any language containing up to 2 symbols and 2-letter words, and can be similarly extended to any alphabet and any word length.

Turing machine views **qAccept** as a step, so the word **“a”** is taken as “**|{q0, q1, qAccept}| = length("a") + 2 = 3"** steps. And the word “**ба”** is taken as “**|{q0, q2, q4, qAccept}| = length(“ба”) + 2 = 4”** steps. By induction, any word from the finite language will be taken as its length plus two, that is, with **O(n)** complexity.

Lemma 2. It’s possible to create a **DT** that accepts any word from the finite language with **O(n)** complexity .

Proof. The construction method for such a **DT** has been described above.

4. **LTM** (Lazy Turing Machine)

The concept of lazy evaluation was introduced by Christopher Wadsworth in “The Semantics and Pragmatics of the Lambda Calculus” [7]. With lazy evaluation, the result is calculated on the fly, as needed. This allows to create "infinite" states and "infinite" values of transition functions.

Definition 5. LTM is a modified Turing machine, where the state set and transition function are lazily evaluated (constructed) by another conventional **Generator** Turing machine.

Definition 6. Generator is a conventional Turing machine, where the input is the tag and **LTM** tape symbol, and output is tape operations and the following **LTM** state.

Definition 7. Generator-DT generates **DT** for **L** language for **LTM.**

Lemma 3. Generator-DT can be created if language **L** is computable

Proof. For an infinite language, the tree’s nodes are where the alphabetic symbols are, and the leaves are where the blank symbol is. Due to the regularity, tree nodes can be created by the state tag: the only possible operations in nodes are shift to the right and transition to another tag. Again, due to regularity, one can determine the corresponding input word in the leaves of the tree. If language **L** is computable, there is a corresponding Turing machine **ML**. We feed this input word to **ML** input in order to get **qAccept** or **qReject** for the transition function.

Lemma 4. For a finite computation, the set of states and transition function values created by **Generator-DT** are finite.

Proof. In case of a finite computation, **Generator-DT** is called a finite number of times.

Lemma 5. The set of infinite programs generated on **LTM** is countable.

Proof. Each infinite program on **LTM** corresponds to a finite **Generator** program on the conventional Turing machine. And finite programs on Turing machine form a countable set.

Theorem 1. Computations in **LTM** with **Generator-DT** are indistinguishable from computations on a conventional Turing machine for which a fairly large number of states and values of transition functions were determined in advance on **Generator-DT** .

Proof. By induction. The same transition at the first step. Further on, a deviation to

1. transition to another state
2. number of steps
3. somewhere in output to the tape
4. final accept/reject status

suggests a different value somewhere in the transition function, but **Generator-DT** runs on a deterministic Turing machine, and generates the same value for the same input.

Theorem 2. Computable language **L** can be computed in **O(n)** on **LTM**.

Proof. Create an **LTM** with the appropriate **Generator-DT**. The program generated by **Generator-DT** has **O(n)** complexity.

Theorem 3. **LTM** concept can be carried over to any computing model.

Proof. LTM can be emulated on a Turing machine, so it can be emulated on any computing model.

Theorem 4. For computable languages, a proof for the conventional Turing machine will also be the proof for LTM if the size of the program is not included in the proof.

Proof. This is a corollary of Theorem 1. The chain of reasoning is the same for **LTM** and the conventional Turing machine, as long as the size of the program is not included in it.

LTM is an almost invisible extension of the conventional Turing machine. The only difference is that **LTM** allows to create an infinite set of states and transition function values through lazy evaluation. And it means that for any Turing complete, i.e. computationally universal models, programs of infinite size do exist. It turns out that from the mathematical point of view, the program's finite size is an artificial limitation, and not an inherent attribute of computations. The upper bound is **O(n)** for any computable language in case a program is infinite.

5. **PSB** (Program Size Barrier)

DT allows any number of special cases to be included in a program. This complicates the first part of “proof by induction”: “let us assume **P ≠ NP** for some **k**”. Because, as a matter of fact, the opposite is true. A program with linear complexity exists for any **k** if it is of a sufficient size. **Theorem 2**, i.e. **LTM** with **Generator-DT,** makes things even more complicated. So much so that it could be called the Program Size Barrier - **PSB**.

Corollary 1. For an unlimited program size, i.e. on **LTM**, **P = NP**.

Proof. According to Theorem 2, any language **L** from **NP** has an upper bound **O(n)**, i.e., is included in the extended (on **LTM**) understanding of class **P**. Any language from **P** is also included in **NP**. That is, **NP** and **P** classes are equal on **LTM**.

Corollary 2. It is impossible to prove **P ≠ NP** without using the program size.

Proof. It follows from Theorem 4, that the same proof will be true for **LTM**. But **P = NP** on **LTM**. That is, such proof cannot exist.

Corollary 3. Without using the program size, it is impossible to prove that **P = NP** is unprovable.

Proof. Similar to **Corollary 2**.

Conjecture 1. Without limiting the length of the program, it is possible to “prove” that **P ≠ NP** is unprovable.

Rationale. A counterexample exists on **LTM**. Most likely, it can be plausibly approached in other ways, "losing" the length limitation for programs in Turing machine along the way. And it turns out that we must always make sure that this kind of proof is also true for finite programs.

Corollary 4. PSB does not restrict subpolynomial proofs.

Proof. Subpolynomial algorithms have lower complexity than polynomial ones. If the complexity is already limited stronger than **O(n)**, **Theorem 2** does not change anything.

Corollary 5. Unlike strict inclusion (⊂), **PSB** does not restrict proofs with equality (**=**) and non-strict (⊆) inclusion.

Proof. Because of **Theorem 2**, it turns out that many classes are equal at infinity. Accordingly, it is problematic to prove the inequality of classes, but it does not interfere with equality or non-strict inclusions.

Corollary 6. PSB does not restrict proofs with a finite program length.

Proof. If the size of program is included in the proof, no case of program of infinite length occurs, i.e. Theorem 2 is not applicable.

Indeed, the easiest way to avoid **PSB** is to "diagonalize", i.e. feed the description of a Turing machine as an input to another Turing machine. As, for example, in the proof of time hierarchy theorem. Diagonalization also hides the existence of **PSB** because it limits the size of program in a rather implicit way - as an input to another machine. It is impossible to process the entire infinite program in finite time. Unfortunately, proof by diagonalization is in turn complicated by the relativizing barrier.

6. Conclusion

It is known that the size of program matters in **P vs NP** . However, it seems that this issue has not been thoroughly explored yet. While other barriers prohibit transferring the existing proofs to **P vs NP** problem, **PSB** requires that a certain detail be provided in the proof. We do not need to try all the proofs, but only those with restrictions on program length. It is necessary to limit the size of program no matter what is being proved. So hopefully this is the last major barrier before the final proof. Other good news is that now we know that **P = NP** if program length is not limited.

Acknowledgments

Thanks to my beloved family and friends Ksenia Kuznetsova, Alexander Kuznetsov, Tatiana Kuznetsova, Alena Drozhdina and Vitaly Moskovkin for their support and helpful discussions. Thanks to my recently deceased cat Semyon for waiting for the first version of this paper.

References

[1] S. Cook. The P versus NP Problem. Manuscript prepared for the Clay Mathematics Institute for the Millennium Prize Problems, http://www.claymath.org/millennium/ P vs NP/pvsnp.pdf, November 2000.

[2] Scott Aaronson, P ?= NP, https:/ /www.scottaaronson.com/papers/pnp.pdf

[3] TP Baker; J. Gill; R. Solovay. (1975). "Relativizations of the P =? NP Question". SIAM Journal on Computing. 4(4): 431–442. doi:10.1137/0204037.

[4] Razborov, Alexander A.; Steven Rudich (1997). "Natural proofs". Journal of Computer and System Sciences. 55(1): 24–35. doi:10.1006/jcss.1997.1494.

[5] S. Aaronson & A. Wigderson (2008). Algebrization: A New Barrier in Complexity Theory (PDF). Proceedings of ACM STOC'2008. pp. 731–740. doi:10.1145/1374376.1374481.

[6] Rosenberger, Jack (May 2012). "P vs. NP poll results". Communications of the ACM. 55(5):10.

[7] Hudak, Paul (September 1989). "Conception, Evolution, and Application of Functional Programming Languages". ACM Computing Surveys. 21(3): 383–385. doi:10.1145/72551.72554. S2CID 207637854.

Addresses

https://docs.google.com/document/d/1pCCWqBLSg4ucMYRdwgsg_LOg8PI7sW7X76WEBgU-AkY/

Medium:
https://medium.com/@evgenykuznetsov_93995/the-equality-p-np-on-programs-with-infinite-length-as-another-barrier-to-the-proof-of-p-np-f52255a1b56f

Mirror: https://cloud.mail.ru/public/5TfP/odYmnfqx4

Endorsement request: https://arxiv.org/auth/endorse?x=FY949X

Discussion

https://www.reddit.com/r/computerscience/comments/z22np0/another_barrier_to_the_proof_of_pnp/

FAQ

1. I can't understand some or all parts, so they are not correct.

Other people's text is difficult to understand. But in this case, if you try, it is possible.

2. Some or all parts are trivial, it's unclear why so much should be written.

Details are needed for those who cannot understand some parts immediately.

3. Is **LTM** something like an oracle?

No, this is an operation opposite to the addition of an oracle. Oracles are added to Turing machines. Here, on the contrary, we remove the axiom of finiteness from the Turing machine and obtain infinite programs and **LTM** as one of the possible implementations of infinite programs.

4. Endless program is suspicious and something bad might happen there.

The **LTM** with **Generator-DT** is specifically constructed to be finite at any given time with finite input data. Strictly speaking, on the contrary, it is more correct to make an infinite tape in TM through lazy execution.

5. Is it true that **P = EXP** on **LTM**?

Yes, that's true. For infinite programs it is generally true that all complexity classes from polynomial and higher have collapsed to linear.

6. If on infinite programs **P = EXP**, then infinite programs are not needed.

Infinite programs are needed at least to prove the **PSB**. In general, the aggregate of infinite+finite programs is a more universal and basic concept than just finite programs. Although infinite programs are indeed less diverse than finite ones, if we consider, for example, complexity classes.

7. It is incorrect that **LTM** does not count steps on the **Generator** machine.

LTM is one way of finite description of infinite programs. For example, the number 2 can and should be described mathematically as the result of adding two 1s. But in fact, the number 2 exists regardless of how we got it or described it. In the same way, infinite programs exist regardless of the way they are described. And since we don't count the steps of writing a finite program, we also don't need to count the steps of writing an infinite program.

8. Previous barriers are more important, so they are better.

It all depends on the criteria of importance. You can define the criterion of importance through the reduction of search: the previous barriers prohibit one method each, and the **PSB** prohibits all methods except one. **PSB** is the only known result that partially defines what the proof of **P != NP** should look like.

9. The previous barriers are more complex, so they are better.

The relativizing barrier is roughly at the same level of complexity as the **PSB**. But the complexity of PSB is rather conceptual. It turned out that it is possible to separate the finiteness of programs as an independent axiom in the manner of the parallel postulate. For many, this is a very difficult moment and it takes some time to stop looking for flaws simply because it is too unexpected and not generally accepted.

10. Some results already use program size and/or do not violate **PSB**.

Valid results and cannot violate **PSB**. The point is to save effort by not trying to prove something by violating the **PSB**.

11. No one will try to prove **P != NP** without using program size.

It is not clear from what this follows.

12. Anything else?

Ok, briefly. There are more **EXP** languages than possible traces of polynomial programs, so the diagonalization proof works there. And **P** and **NP** languages are the same number, since they can be derived from each other, which is why these languages are so difficult to distinguish mathematically. Hehe.