P versus NP

Frank Vega

P versus NP

Frank Vega[1]

1 **Joysonic, Uzun Mirkova 5/Belgrade, Serbia**
vega.frank@gmail.com

Abstract

P versus NP is considered one of the great open problems of science. This consists in knowing the answer of the following question: Is P equal to NP? This incognita was first mentioned in a letter written by John Nash to the National Security Agency in 1955. However, a precise statement of the P versus NP problem was introduced independently in 1971 by Stephen Cook and Leonid Levin. Since that date, all efforts to find a proof for this huge problem have failed. We prove $P = NP \Rightarrow \exists k \in \mathbb{N} : TIME(n^{\frac{k}{2}}) = NTIME(n^{\frac{k}{2}})$. However, we demonstrate that $\forall k \in \mathbb{N} : TIME(n^{\frac{k}{2}}) \neq NTIME(n^{\frac{k}{2}})$. In this way, we show that $P \neq NP$.

1998 ACM Subject Classification F.1.3.3: Relations among complexity classes

Keywords and phrases P, NP, TIME, NTIME, Minimum

1 Issue

P versus NP is a major unsolved problem in computer science [3]. It is considered by many to be the most important open problem in the field [3]. It is one of the seven Millennium Prize Problems selected by the Clay Mathematics Institute to carry a US$1,000,000 prize for the first correct solution [3].

In 1936, Turing developed his theoretical computational model [1]. The deterministic and nondeterministic Turing machines have become in two of the most important definitions related to this theoretical model for computation. A deterministic Turing machine has only one next action for each step defined in its program or transition function [6]. A nondeterministic Turing machine could contain more than one action defined for each step of its program, where this one is no longer a function, but a relation [6].

Another huge advance in the last century has been the definition of a complexity class. A language over an alphabet is any set of strings made up of symbols from that alphabet [2]. A complexity class is a set of problems, which are represented as a language, grouped by measures such as the running time, memory, etc [2].

The set of languages decided by deterministic Turing machines within time f is an important complexity class denoted $TIME(f(n))$ [6]. In addition, the complexity class $NTIME(f(n))$ consists in those languages that can be decided within time f by nondeterministic Turing machines [6]. The most important complexity classes are P and NP. The class P is the union of all languages in $TIME(n^k)$ for every possible positive constant k [6]. At the meantime, NP consists in all languages in $NTIME(n^k)$ for every possible positive constant k [6].

The biggest open question in theoretical computer science concerns the relationship between these classes: Is P equal to NP? In 2002, a poll of 100 researchers showed that 61 believed that the answer was no, 9 believed that the answer was yes, and 22 were unsure; 8 believed the question may be independent of the currently accepted axioms and so impossible to prove or disprove [4]. All efforts to solve the P versus NP problem have failed [6]. The goal of this work is to show that $P \neq NP$.

2 Motivation

Most complexity theorists already assume P is not equal to NP, but nobody has found a valid proof yet [4]. There are several consequences if P is not equal to NP, such as many common problems cannot be solved efficiently [3]. Indeed, that could help us to certainly deduce the separation of classes below P from classes above P [1]. Besides, if P is not equal to NP, then this could be a good indication that important mathematical problems, such as *GRAPH ISOMORPHISM* and *FACTORIZATION* might not be solved in polynomial time [3]. To sum up, P versus NP is one of the great open problems in science and a correct solution for this incognita will have a great impact not only for computer science, but for other fields too [3].

3 Summary

In this work, we proved that $\forall k \in \mathbb{N} : TIME(n^{\frac{k}{2}}) \neq NTIME(n^{\frac{k}{2}})$. This is a new approach for an open question in complexity theory, that is understanding the relation between $TIME(n^k)$ and $NTIME(n^k)$ for an arbitrarily selected positive constant k [7]. We also demonstrated that $P = NP \Rightarrow \exists k \in \mathbb{N} : TIME(n^{\frac{k}{2}}) = NTIME(n^{\frac{k}{2}})$. In this way, it is showed that $P \neq NP$. This result is based on the conclusion if we have to know the answer of a total number of different computations from a deterministic Turing machine over different inputs, then the only optimal and possible way for an arbitrary case is to do it just executing each computation and checking the answer for every input. This notion is very similar of what it has been studied as optimal for the number of comparisons in the problem of finding the minimum inside an arbitrary array of positive integers [2].

4 Highlights

For the total comprehension of this work, it is important to take into special consideration the parts in the paper where we used the word "*note*", because in those short fragments we tried to explain the real intention and purpose for those principal ideas that might be cause confusion to the reader. Here are some few examples in the paper where was used that word (there are more ...):

- ... Note that $L[M]$ definition is different of $L(M)$ ...
- ... Note that Ω_s definition is different of Ω ...
- ... Note that the intention is not exactly to define a linear order ...
- ... It is important to note the size of N and $c \times |x|^j$ would not affect the membership of L in $TIME(n^j)$ ...

But, we really think the most important detail in the paper was inside the Definition of the language H_{NP}, where we guaranteed this problem is actually in *NP–complete* due to the following restriction: "$\ldots 1^i$ *is strictly non-superpolynomial in relation to the size of* $x \ldots$", because without that restriction, then the problem would trivially be *EXP–complete* since the mentioned integer i in the Definition should be given in binary encoding.

Finally, we did not waste so much effort in explaining some details that are trivial to understand even by a non-specialist reader in complexity theory and we cited a reference which explains those kinds of topics in each case, such as the following statement: "*... we can decide a finite subset of a language in linear time* ...", because we can trivially simulate a (probably large) deterministic Turing machine which identifies each string of the

finite subset just reading each symbol into a sequential way until it halts in the blank symbol and deciding whether that element belongs to that finite set or not (this can be always done in linear time no matter how large could be the finite set).

5 Theory

▶ **Definition 1.** Given an array a_i of m positive integers, *SEARCH–MINIMUM* is the problem of finding the minimum of a_i.

How many comparisons are necessary to determine the minimum of an array of m positive integers? We can easily obtain an upper bound of $m-1$ comparisons: examine each integer of the array in turn and keep track of the smallest element seen so far [2]. Is this the best we can do? Yes, since we can obtain a lower bound of $m-1$ comparisons for the problem of determining the minimum [2].

▶ **Definition 2.** Given a number x and an array a_i of m positive integers, $MINIMUM$ is the problem of deciding whether x is the minimum of a_i.

How many comparisons are necessary to determine whether some x is the minimum of an array of m positive integers? We can easily obtain an upper bound of m comparisons: find the minimum in the array and check whether the result is equal to x. Is this the best we can do? Yes, since we can obtain a lower bound of $m-1$ comparisons for the problem of determining the minimum and another obligatory comparison for checking whether that minimum is equal to x.

Let Σ be a finite alphabet with at least two elements, and let Σ^* be the set of finite strings over Σ [1]. A Turing machine M has an associated input alphabet Σ [1]. For each string w in Σ^* there is a computation associated with M on input w [1]. We say that M accepts w if this computation terminates in the accepting state, that is $M(w) =$ "*yes*" [1]. Note that M fails to accept w either if this computation ends in the rejecting state, that is $M(w) =$ "*no*", or if the computation fails to terminate [1].

The language accepted by a Turing machine M, denoted $L(M)$, has an associated alphabet Σ and is defined by

$$L(M) = \{w \in \Sigma^* : M(w) = \text{“}yes\text{”}\}.$$

We denote by $t_M(w)$ the number of steps in the computation of M on input w [1]. For $n \in \mathbb{N}$ we denote by $T_M(n)$ the worst case run time of M; that is

$$T_M(n) = max\{t_M(w) : w \in \Sigma^n\}$$

where Σ^n is the set of all strings over Σ of length n [1]. We say that M runs in polynomial time if there exists k such that for all n, $T_M(n) \leq n^k + k$ [1].

A verifier for a language L is a deterministic Turing machine M, where

$$L = \{w : M(w, c) = \text{“}yes\text{”} \textit{ for some string } c\}.$$

We measure the time of a verifier only in terms of the length of w, so a polynomial time verifier runs in polynomial time in the length of w [1]. A verifier uses additional information, represented by the symbol c, to verify that a string w is a member of L. NP is also known as the complexity class of languages defined by polynomial time verifiers [6].

A function $f : \Sigma^* \to \Sigma^*$ is a polynomial time computable function if some deterministic Turing machine M, on every input w, halts in polynomial time with just $f(w)$ on its tape

[6]. Let $\{0,1\}^*$ be the infinite set of binary strings, we say that a language $L_1 \subseteq \{0,1\}^*$ is polynomial time reducible to a language $L_2 \subseteq \{0,1\}^*$, written $L_1 \leq_p L_2$, if there exists a polynomial time computable function $f : \{0,1\}^* \to \{0,1\}^*$ such that for all $x \in \{0,1\}^*$,

$x \in L_1$ *if and only if* $f(x) \in L_2$.

An important complexity class is *NP–complete* [5]. A language $L \subseteq \{0,1\}^*$ is *NP–complete* if

- $L \in NP$, and
- $L' \leq_p L$ for every $L' \in NP$.

If any single *NP–complete* problem can be solved in polynomial time, then every NP problem has a polynomial time algorithm [2]. No polynomial time algorithm has yet been discovered for any *NP–complete* problem [3].

6 Results

▶ **Definition 3.** Given a deterministic Turing machine M, we define a language $L[M] \subseteq \{0,1\}^*$ when $L[M]$ has as a polynomial time verifier the Turing machine M. Note that $L[M]$ definition is different of $L(M)$.

Just as *O–notation* provides an asymptotic upper bound, Ω*–notation* provides an asymptotic lower bound [2]. In computer science, *O–notation* and Ω*–notation* are used to classify algorithms according to their running time or space requirements [2].

▶ **Definition 4.** Given a deterministic Turing machine M, we say that $L[M] \in \Omega_s(n^k)$ for some $k \in \mathbb{N}$ if the verifier M has the best asymptotic lower bound in comparison with all possible verifiers of $L[M]$ and decides $M(w,c)$ in $\Omega(|w|^k)$ where $|\ldots|$ denotes the bit-length function. Note that Ω_s definition is different of Ω.

▶ **Definition 5.** Given a deterministic Turing machine M and a binary string x, we say that x is equal to another binary string y on M when $M(x,y) =$ "*yes*" and $M(y,x) =$ "*yes*", x is smaller than or equal to y on M when $M(x,y) =$ "*yes*" or y is smaller than or equal to x on M when $M(y,x) =$ "*yes*". Note that the intention is not exactly to define a linear order.

▶ **Definition 6.** If there is an array a_i of m binary strings, then we would say x is the minimum of a_i on M if and only if for every element y in the array a_i we have that x is smaller than or equal to y on M. Note, in this definition the minimum on M of an array a_i of m binary strings could not be unique.

▶ **Definition 7.** Given a string x and an array a_i of m binary strings such that $|x| = m$ and $|a_i| \leq m^2$, *GENERALIZED–MINIMUM*$[M]$ is a problem defined only into a specific deterministic Turing machine M and consists of deciding whether x is the minimum of a_i on M where $|\ldots|$ denotes the bit-length function. We abbreviate this problem as $GM[M]$.

▶ **Theorem 8.** *Given a problem* $GM[M]$ *based on a given deterministic Turing machine* M *that is a polynomial time verifier of a language* $L[M]$*, if* $L[M] \in \Omega_s(n^k)$ *for some* $k \in \mathbb{N}$*, then* $GM[M]$ *must be solved in* $\Omega(n^{\frac{k+1}{2}})$.

Proof. How many computations are necessary to determine whether some x is the minimum on M of an array of m binary strings? We can easily obtain an upper bound of m computations: examine each string y of the array in turn and check whether x is smaller than or equal to y on M. Is this the best we can do?

Think of any algorithm that determines the minimum on M as a tournament among the elements [2]. Each computation of M is a match in the tournament in which the smaller or equal on M of the two elements wins [2]. The key observation is that every element must lose at least one match [2]. Hence, m computations are necessary to determine whether x is the minimum on M of an array a_i of m binary strings, and this algorithm for $GM[M]$ is optimal with respect to the number of computations performed.

Finally, since $L[M] \in \Omega_s(n^k)$, then we must verify m computations on M in $\Omega(m^k)$ because $|x| = m$. For that reason, we must decide $(x, a_i) \in GM[M]$ in $\Omega(m^{k+1})$. Due to $|a_i| \leq m^2$, then the bit-length n of the instance (x, a_i) will comply with $m = \Omega(n^{\frac{1}{2}})$. Note that " = " in $m = \Omega(n^{\frac{1}{2}})$ is not meant to express "*is equal to*", but rather a more colloquial "*is*" [2]. Consequently, we must decide the elements of $GM[M]$ in $\Omega(n^{\frac{k+1}{2}})$ under the assumption of $L[M] \in \Omega_s(n^k)$, because the verifier M has the best asymptotic lower bound for the language $L[M]$. ◀

Note that the relation between $TIME(n^k)$ and $NTIME(n^k)$ is widely open for general k. This is only answered for $k = 1$ until now, where both classes are different [7].

▶ **Theorem 9.** $\forall k \in \mathbb{N} : TIME(n^{\frac{k}{2}}) \neq NTIME(n^{\frac{k}{2}})$.

Proof. Suppose there exists a $k \in \mathbb{N}$ such that $TIME(n^{\frac{k}{2}}) = NTIME(n^{\frac{k}{2}})$. We define a language $GM[M]$ based on a given deterministic Turing machine M such that there exists a language $L[M] \in \Omega_s(n^k)$ where M is the polynomial time verifier of the language $L[M]$. Let's define the complement of $GM[M]$ as $coGM[M]$ such that for an appropriate input (x, a_i) we have $(x, a_i) \in coGM[M]$ if and only if $(x, a_i) \notin GM[M]$. Now, let's define an algorithm for $coGM[M]$ as follows:

- On appropriate input (x, a_i)
- Choose nondeterministically an integer j such that $0 \leq j \leq length(a_i) - 1$
- if $(M(x, a_i[j]) =$ "no")
- return "yes"
- else
- return "no".

Certainly, if there is some $(x, a_i) \in coGM[M]$, then there will be an acceptance path on this algorithm for that input. This algorithm is nondeterministic since we choose j in a nondeterministic way. Indeed, we could create a random positive integer j in $O(length(a_i))$ just initializing it to 0 and adding 1 until at most $length(a_i) - 1$ times. In addition, once we generate j, then we can compute $M(x, a_i[j])$ in $\Omega(|x|^k)$ because of $L[M] \in \Omega_s(n^k)$ where $a_i[j]$ is the element of a_i in the j^{th} index position. For that reason, we conclude $coGM[M]$ can be solved in $\Omega(m^k)$ where $m = length(a_i)$, $|x| = m$ and $|a_i| \leq m^2$ because (x, a_i) is an appropriate input. Thus $coGM[M] \in NTIME(n^{\frac{k}{2}})$, because the bit-length n of the instance (x, a_i) will comply with $m = \Omega(n^{\frac{1}{2}})$.

If $TIME(n^{\frac{k}{2}}) = NTIME(n^{\frac{k}{2}})$, then $GM[M] \in TIME(n^{\frac{k}{2}})$ because the language $coGM[M]$ would be in $TIME(n^{\frac{k}{2}})$ and we have that $TIME(n^{\frac{k}{2}})$ is closed under complement [6]. But this is a contradiction with Theorem 8 because $GM[M]$ must be solved in $\Omega(n^{\frac{k+1}{2}})$ under the assumption of $L[M] \in \Omega_s(n^k)$. Therefore, for the sake of contradiction $TIME(n^{\frac{k}{2}}) \neq NTIME(n^{\frac{k}{2}})$. Since we take an arbitrary $k \in \mathbb{N}$, then $\forall k \in \mathbb{N} : TIME(n^{\frac{k}{2}}) \neq NTIME(n^{\frac{k}{2}})$. ◀

▶ **Definition 10.** Let define H_{NP} to be the restricted nondeterministic polynomial time bounded version of $HALTING$ language [6]:

$H_{NP} = \{(i, x, N) : N$ *accepts input* x *after at most* i *steps*

and 1^i *is strictly non-superpolynomial in relation to the size of* $x\}$

when N is a nondeterministic Turing machine and $i \in \mathbb{N}$ where 1^i is a string of 1's of length i. Certainly, $H_{NP} \in NP$*–complete* [6].

▶ **Theorem 11.** $P = NP \Rightarrow \exists k \in \mathbb{N} : TIME(n^{\frac{k}{2}}) = NTIME(n^{\frac{k}{2}})$.

Proof. Suppose that $P = NP$. Thus, there will be a polynomial time algorithm for H_{NP}. Therefore, there would exist a $j \in \mathbb{N}$ such that $H_{NP} \in TIME(n^j)$. We could reduce every instance $x \in L$ of an arbitrary language $L \in NTIME(n^j)$ to $(c \times |x|^j, x, N) \in H_{NP}$ where N is the nondeterministic Turing machine that decides L and $c \times |x|^j$ is a polynomially upper bound of the time in which N accepts x and $c \in \mathbb{N}$. Since the string $1^{c \times |x|^j}$ is polynomially bounded by x and N is previously fixed, then we can reduce every instance $x \in L$ of a language $L \in NTIME(n^j)$ to H_{NP} in $O(n^j)$.

As conclusion, we obtain that $L \in TIME(n^j)$ because $H_{NP} \in TIME(n^j)$. It is important to note the size of N and $c \times |x|^j$ would not affect the membership of L in $TIME(n^j)$. It could be the case the string $1^{c \times |x|^j}$ would be superpolynomially larger in relation to the size of certain instances $x \in L$, because the constant $j \in \mathbb{N}$ could be a big number. In this way, it might be affected the reduction of L to H_{NP} for those instances. However, that occurs for a finite amount of instances $x \in L$, because when the size of the string x is bigger, then the size of the string $1^{c \times |x|^j}$ stops of being superpolynomial in relation to x. But, we already know we can decide a finite subset of a language in linear time and thus this would not affect the membership of L in $TIME(n^j)$ too [6].

Consequently, since we take L as an arbitrary language of $NTIME(n^j)$, then we can conclude $TIME(n^j) = NTIME(n^j)$. Finally, if we take the natural number $k = 2 \times j$, then $TIME(n^{\frac{k}{2}}) = NTIME(n^{\frac{k}{2}})$ and in this way the Theorem is proved. ◀

▶ **Theorem 12.** $P \neq NP$

Proof. This is a direct consequence of Theorems 9 and 11. ◀

7 Significance

This proof explains why after decades of studying the NP problems no one has been able to find a polynomial time algorithm for any of more than 3000 important known *NP–complete* problems [5]. Indeed, it shows in a formal way that many currently mathematically problems cannot be solved efficiently, so that the attention of researchers can be focused on partial solutions or solutions to other problems [3].

Although this demonstration removes the practical computational benefits of a proof that $P = NP$, it would represent a very significant advance in computational complexity theory and provide guidance for future research. On the one hand, it proves that could be safe most of the existing cryptosystems such as the public key cryptography [5]. On the other hand, we will not be able to find a formal proof for every theorem which has a proof of a reasonable length by a feasible algorithm [3].

References

1 Sanjeev Arora and Boaz Barak. *Computational complexity: A modern approach*. Cambridge University Press, 2009.

2 Thomas H. Cormen, Charles Eric Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, 2 edition, 2001.

3 Lance Fortnow. *The Golden Ticket: P, NP, and the Search for the Impossible*. Princeton University Press. Princeton, NJ, 2013.

4 William I. Gasarch. The P=?NP poll. *SIGACT News*, 33(2):34–47, 2002. doi:10.1145/1052796.1052804.

5 Oded Goldreich. *P, Np, and Np-Completeness*. Cambridge: Cambridge University Press, 2010.

6 Christos H. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.

7 W. J. Paul, N. Pippenger, E. Szemeredi, and W. T. Trotter. On Determinism Versus Non-Determinism and Related Problems. *IEEE Symp. on Foundations of Comp. Sci.*, 24:429–438, 1983. doi:10.1109/SFCS.1983.39.