On P versus NP

Frank Vega

March 19, 2019

Abstract: P versus NP is considered as one of the great open problems of science. This consists in knowing the answer of the following question: Is P equal to NP? This problem 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 by Stephen Cook and Leonid Levin. Since that date, all efforts to find a proof for this problem have failed. Another major complexity class is coNP. Whether NP = coNP is another fundamental question that it is as important as it is unresolved. We prove there is a problem in coNP that is not in P. In this way, we show that P is not equal to coNP. Since P = NP implies P = coNP, then we also demonstrate that P is not equal to NP.

Introduction

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

ACM Classification: F.1.3.3, F.1.3.2

AMS Classification: 68Q15, 68Q17

Key words and phrases: P, NP, coNP, coNP-complete, Minimum, Boolean circuit

set of problems, which are represented as a language, grouped by measures such as the running time, memory, etc [2].

In the computational complexity theory, the class P contains those languages that can be decided in polynomial time by a deterministic Turing machine [5]. The class NP consists in those languages that can be decided in polynomial time by a nondeterministic Turing machine [5].

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].

Another major complexity class is $coNP$ [6]. We prove there is a problem in $coNP$ that is not in P. Since $P = NP$ implies that every $coNP$ problem is in P, then we can deduce that $P \neq NP$ [6].

1 Theoretical notions

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, $M(w) = \text{“}yes\text{”}$ [1]. Note that M fails to accept w either if this computation ends in the rejecting state, 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 running 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]. The notations we use to describe the asymptotic running time of an algorithm are defined in terms of functions whose domains are the set of natural numbers [2]. Such notations are convenient for describing the worst case running time function, which is usually defined only on integer input sizes [2]. For a given function $g(n)$, we denote by $O(g(n))$ the set of functions

$$O(g(n)) = \{f(n) : \textit{There exist positive constants } c \textit{ and } n_0$$
$$\textit{such that } 0 \leq f(n) \leq c \times g(n) \textit{ for all } n \geq n_0\}$$

where O-notation provides an asymptotic upper bound [2].

A language L is in class P if $L = L(M)$ for some deterministic Turing machine M which runs in polynomial time [1]. We state the complexity class NP using the following definition: 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 [7]. A verifier uses additional information, represented by the symbol c, to verify that a string w is a member of L. This information is called certificate. For polynomial time verifiers, the certificate is polynomially bounded by the length of w, because that is all the verifier can access in its time bound [7]. *NP* is also the class of languages that have polynomial time verifiers [7]. If *NP* is the class of problems that have succinct certificates, then the complexity class *coNP* must contain those problems that have succinct disqualifications [6]. That is, a "*no*" instance of a problem in *coNP* possesses a short proof of its being a "*no*" instance [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 [7]. 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 $x \in \{0,1\}^*$,

$$x \in L_1 \textit{ iff } f(x) \in L_2$$

where *iff* means "if and only if". An important complexity class is *coNP–complete* [5]. A language $L \subseteq \{0,1\}^*$ is *coNP–complete* if

1. $L \in coNP$, and
2. $L' \leq_p L$ for every $L' \in coNP$.

Furthermore, if L is a language such that $L' \leq_p L$ for some $L' \in coNP\text{–}complete$, then L is in *coNP–hard* [2]. Moreover, if $L \in coNP$, then $L \in coNP\text{–}complete$ [2].

For every $n, m \in \mathbb{N}$ a Boolean circuit C with n inputs and m outputs is a directed acyclic graph [1]. It contains n nodes with no incoming edges; called the input gates and m nodes with no outgoing edges, called the output gates [1]. All other nodes are labeled with one of $\vee$, $\wedge$ or $\neg$ (in other words, the logical operations OR, AND, and NOT) [1]. The $\vee$ and $\wedge$ nodes have fanin (i.e., number of incoming edges) of 2 and the $\neg$ nodes have fanin 1. The size of C is the number of nodes in it [1].

A principal *coNP–complete* problem is *CIRCUIT–UNSAT* [5]. The instances of *CIRCUIT–UNSAT* are the Boolean circuits with a single output gate [1]. We say that a Boolean circuit C accepts some bit positive integer y when the evaluation of C on y over the input gates finally returns 1 in the output gate [1]. The *CIRCUIT–UNSAT* can be formulated as follows: Given a Boolean circuit C, is not there any bit positive integer y such that C accepts y?

2 Results

Definition 2.1. In computational complexity theory, $DTIME$ (or $TIME$) is the computational resource of computation time for a deterministic Turing machine [6]. The resource $DTIME$ is used to define complexity classes, sets of all of the decision problems which can be solved using a certain amount of computation time [6]. If a problem of input size n can be solved in $O(f(n))$, we have a complexity class $DTIME(f(n))$ (or $TIME(f(n))$) [6].

Definition 2.2. Given an integer x and a Boolean circuit C, we define *SUCCINCT–INFERIOR* as the problem of deciding whether there is not any bit positive integer y which C accepts as input such that $x < y$.

Theorem 2.3. *CIRCUIT–UNSAT cannot be decided in constant time.*

Proof. Suppose that the language *CIRCUIT–UNSAT* can be decided in constant time. This would imply that *CIRCUIT–UNSAT* is a regular language [6]. Certainly, if *CIRCUIT–UNSAT* can be decided in constant time, then this can be decided in constant space [6]. If some language L is infinite and regular, then there are x, y and z in Σ^* such that y is not an empty string and $xy^iz \in L$ for all $i \geq 0$ where y^i is the i^{th} concatenation of the same repeated string y [6]. However, *CIRCUIT–UNSAT* is infinite and there are no instances in *CIRCUIT–UNSAT* for which the previous statement is true. Hence, *CIRCUIT–UNSAT* is not a regular language and therefore, this cannot be decided in constant time. □

Definition 2.4. A representation of a set S with n positive integers is a Boolean circuit C, such that C accepts the bit positive integer i if and only if $i \in S$.

Theorem 2.5. *SUCCINCT–INFERIOR* $\notin DTIME(|S|)$ *where* S *is the set that represents the Boolean circuit* C *and* $|\ldots|$ *is the cardinality set function.*

Proof. Since a Boolean circuit C can only accept the bit positive integers as appropriate inputs, then we could make a polynomial time reduction as follows:

$$C \in \textit{CIRCUIT–UNSAT} \textit{ iff } (-1, C) \in \textit{SUCCINCT–INFERIOR}$$

where *iff* means "if and only if". However, this reduction can be done in constant time. That means we cannot decide every instance $(-1, C) \in$ *SUCCINCT–INFERIOR* in constant time, because that would mean we can decide *CIRCUIT–UNSAT* in constant time. Certainly, we cannot decide the language *CIRCUIT–UNSAT* in constant time according to the Theorem 2.3. Nevertheless, from an instance $(-1, C) \in$ *SUCCINCT–INFERIOR*, we would have $S = \emptyset$ and so $|S| = 0$ where S is the set that represents the Boolean circuit C. Thus, we can assure if *SUCCINCT–INFERIOR* $\in DTIME(|S|)$, then we could decide *CIRCUIT–UNSAT* in constant time. For that reason, for the sake of contradiction we can confirm *SUCCINCT–INFERIOR* $\notin DTIME(|S|)$. □

Theorem 2.6. *SUCCINCT–INFERIOR* $\notin P$.

Proof. For certain kind of instances, the input (x, C) is exponentially more succinct than the cardinality of the set S that represents C [6]. Since we have that *SUCCINCT–INFERIOR* $\notin DTIME(|S|)$, then we could not decide every instance of *SUCCINCT–INFERIOR* in polynomial time. □

Theorem 2.7. *SUCCINCT–INFERIOR* $\in coNP$.

Proof. If $(x, C) \notin$ *SUCCINCT–INFERIOR*, then it would exist a bit positive integer y such that $x < y$ and C accepts y. Since we can evaluate whether C accepts the bit positive integer y and we have that y should be polynomially bounded by C, then we can confirm *SUCCINCT–INFERIOR* $\in coNP$ due to the verification of $x < y$ and the evaluation on the Boolean circuit can be done in polynomial time. □

Theorem 2.8. $P \neq NP$.

Proof. If any single *coNP* problem cannot be decided in polynomial time, then $P \neq coNP$ [6]. Certainly, the result $P = NP$ implies that $P = NP = coNP$ because P is closed under complement and therefore, we can conclude $P \neq NP$ due to $P = NP = coNP$ is false under the basis of $P \neq coNP$ [6]. □

Conclusions

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 300 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.

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. In addition, 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.

References

[1] SANJEEV ARORA AND BOAZ BARAK: *Computational complexity: A modern approach*. Cambridge University Press, 2009. 1, 2, 3

[2] THOMAS H. CORMEN, CHARLES ERIC LEISERSON, RONALD L. RIVEST, AND CLIFFORD STEIN: *Introduction to Algorithms*. MIT Press, 2 edition, 2001. 1, 2, 3

[3] LANCE FORTNOW: *The Golden Ticket: P, NP, and the Search for the Impossible*. Princeton University Press. Princeton, NJ, 2013. 1

[4] WILLIAM I. GASARCH: The P=?NP poll. *SIGACT News*, 33(2):34–47, 2002. 2

[5] ODED GOLDREICH: *P, Np, and Np-Completeness*. Cambridge: Cambridge University Press, 2010. 2, 3, 5

[6] CHRISTOS H. PAPADIMITRIOU: *Computational Complexity*. Addison-Wesley, 1994. 1, 2, 3, 4, 5

[7] MICHAEL SIPSER: *Introduction to the Theory of Computation*. Thomson Course Technology, 2 edition, 2006. 3

AUTHOR

Frank Vega
Computational Researcher
Joysonic
Belgrade, Serbia
vega.frank@gmail.com
https://uh-cu.academia.edu/FrankVega

ABOUT THE AUTHOR

FRANK VEGA is essentially a back-end programmer graduated in Computer Science since 2007. In August 2017, he was invited as a guest reviewer for a peer-review of a manuscript about Theory of Computation in the flagship journal of IEEE Computer Society. In October 2017, he contributed as co-author with a presentation in the 7^{th} International Scientific Conference on economic development and standard of living ("EDASOL 2017 - Economic development and Standard of living"). In February 2017, his book "Protesta" (a book of poetry and short stories in Spanish) was published by the Alexandria Library Publishing House. He was also Director of two IT Companies (Joysonic and Chavanasoft) created in Serbia.