P vs NP

Frank Vega

Joysonic, Belgrade, Serbia
vega.frank@gmail.com

Abstract. P versus NP is considered as one of the most important open problems in computer science. This consists in knowing the answer of the following question: Is P equal to NP? We obtain a contradiction taking as assumption that P = NP. Therefore, we prove P is not equal to NP by reduction ad absurdum.

Keywords: Complexity classes · Completeness · Polynomial time · Graph · Circuit.

1 Introduction

The P versus NP problem is a major unsolved problem in computer science [1]. This is considered by many to be the most important open problem in the field [1]. 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 [1]. It was essentially mentioned in 1955 from a letter written by John Nash to the United States National Security Agency [1]. However, the precise statement of the $P = NP$ problem was introduced in 1971 by Stephen Cook in a seminal paper [1].

In 1936, Turing developed his theoretical computational model [6]. The deterministic and nondeterministic Turing machines have become in two of the most important definitions related to this theoretical model for computation [6]. 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 relevant 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 [3]. A complexity class is a set of problems, which are represented as a language, grouped by measures such as the running time, memory, etc [3].

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 fixed constant k [6]. At the same time, NP consists in all languages in $NTIME(n^k)$ for every possible positive fixed

constant k [6]. NP is also the complexity class of languages whose solutions may be verified in polynomial time [6]. The biggest open question in theoretical computer science concerns the relationship between these classes: Is P equal to NP? In 2012, a poll of 151 researchers showed that 126 (83%) believed the answer to be no, 12 (9%) believed the answer is yes, 5 (3%) believed the question may be independent of the currently accepted axioms and therefore impossible to prove or disprove, 8 (5%) said either do not know or do not care or don't want the answer to be yes nor the problem to be resolved [5]. We obtain a contradiction assuming that $P = NP$. In this way, we prove $P \neq NP$ by reduction ad absurdum.

2 Theory

Let Σ be a finite alphabet with at least two elements, and let Σ^* be the set of finite strings over Σ [2]. A Turing machine M has an associated input alphabet Σ [2]. For each string w in Σ^* there is a computation associated with M on input w [2]. We say that M accepts w if this computation terminates in the accepting state, that is $M(w) =$ "*yes*" [2]. 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 [2].

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 [2]. 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 [2]. We say that M runs in polynomial time if there is a constant k such that for all n, $T_M(n) \leq n^k + k$ [2]. In other words, this means the language $L(M)$ can be accepted by the Turing machine M in polynomial time. Therefore, P is the complexity class of languages that can be accepted in polynomial time by deterministic Turing machines [3]. 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 [2]. 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. NP is also 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 [2]. 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 is a polynomial time computable function $f : \{0,1\}^* \to \{0,1\}^*$ such that for all $x \in \{0,1\}^*$:

$$x \in L_1 \textit{ if and only if } f(x) \in L_2.$$

An important complexity class is *NP–complete* [4]. A language $L \subseteq \{0,1\}^*$ is *NP–complete* if

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

If L is a language such that $L' \leq_p L$ for some $L' \in$ *NP–complete*, then L is *NP–hard* [3]. Moreover, if $L \in NP$, then $L \in$ *NP–complete* [3].

HAMILTON–PATH is an important *NP–complete* problem [4]. An instance of the language *HAMILTON–PATH* is a graph $G = (V, E)$ where V is the set of vertices and E is the set of edges, each edge being an ordered pair of vertices [4]. We say $(u, v) \in E$ is an edge in a graph $G = (V, E)$ where u and v are vertices. For a graph $G = (V, E)$ a simple path in G is a sequence of distinct vertices $< v_0, v_1, v_2, ..., v_k >$ such that $(v_{i-1}, v_i) \in E$ for $i = 1, 2, ..., k$ [3]. A Hamilton path is a simple path of the graph which contains all the vertices of the graph. The problem *HAMILTON–PATH* asks whether a graph has a Hamilton path [4].

Another *NP–complete* problem is *CIRCUIT–SAT* [4]. A Boolean circuit is an acyclic graph $C = (V, E)$, where the nodes $V = \{1, ..., n\}$ are called the gates of C. We can assume that all edges are of the form (i, j) where $i < j$. All nodes in the graph have in-degree (number of incoming edges) equal to 0, 1 and 2. Also, each gate $i \in V$ has a sort $c(i)$ associated with it, where $c(i) \in \{true, false, \wedge, \vee, \rightarrow\} \cup \{x_1, x_2, ...\}$. If $c(i) \in \{true, false\} \cup \{x_1, x_2, ...\}$, then the in-degree of i is 0, that is, i must have no incoming edges. Gates with no incoming edges are called the inputs of C. If $c(i) = \rightarrow$, then i has in-degree one. If $c(i) \in \{\wedge, \vee\}$, then the in-degree of i must be two. Finally, node n (the largest numbered gate in the circuit, which necessarily has no outgoing edges), is called the output gate of the circuit. Let $X(C)$ be the set of all Boolean variables that appear in the circuit C (that is, $X(C) = \{x \in X : c(i) = x \textit{ for some gate } i \textit{ in } C\}$). We say that a truth assignment T is appropriate for C if it is defined for all the variables in $X(C)$. The problem *CIRCUIT–SAT* asks whether a given circuit C has a truth assignment T, appropriate to C, such that $C(T) = true$. Consider, however, the same problem for circuits with no variable gates. This problem, known as *CIRCUIT–VALUE*, obviously has a polynomial time algorithm [6].

On the other hand, EXP is the complexity class of languages that can be accepted in exponential time by deterministic Turing machines [3]. $NEXP$ is the complexity class of languages defined by exponential time verifiers [6]. *NEXP–complete* is also defined under polynomial time reductions but each problem is in $NEXP$. One of the most important problems related to circuits and graph is *SUCCINCT–HAMILTON–PATH*. A succinct representation of a graph with $2 \times n - 1$ nodes is a Boolean circuit C with $2 \times b$ input gates where $n = 2^b$

is a power of two [6]. The graph represented by C, denoted G_C, is defined as follows: The nodes of G_C are $\{0, 1, 2, \ldots, 2 \times n - 1\}$. And (i, j) is an edge of G_C if and only if C accepts the binary representations of the *b-bits* integers i, j as inputs [6]. The problem *SUCCINCT–HAMILTON–PATH* is now this: Given the succinct representation C of a graph G_C with $2 \times n - 1$ nodes, does G_C have a Hamilton path? The problem *SUCCINCT–HAMILTON–PATH* is in *NEXP–complete* [6].

3 Results

Definition 1. ϵ–***HAMILTON–PATH***

INSTANCE: A string $C\epsilon \ldots \epsilon$ where C is a succinct representation of a graph G_C with $2 \times n - 1$ nodes concatenated with an amount of empty strings ϵ [6].

QUESTION: Does G_C have a Hamilton path?

Theorem 1. *SUCCINCT–HAMILTON–PATH $\leq_p$ ϵ–HAMILTON–PATH.*

Proof. A succinct representation C of a graph G_C with $2 \times n - 1$ nodes belongs to *SUCCINCT–HAMILTON–PATH* if and only if $C\epsilon \in \epsilon$–*HAMILTON–PATH.*

Theorem 2. *Every language L complies with ϵ–HAMILTON–PATH $\leq_p L$ when L contains every element of SUCCINCT–HAMILTON–PATH concatenated with some amount of empty strings ϵ.*

Proof. Every string

$$C\underbrace{\epsilon \ldots \epsilon}_{s}$$

belongs to ϵ–*HAMILTON–PATH* if and only if

$$C\underbrace{\epsilon \ldots \epsilon}_{t}$$

belongs to L where s and t are two positive integers not necessarily equals. This reduction can be done in polynomial time since each string

$$C\underbrace{\epsilon \ldots \epsilon}_{s}$$

is equivalent to

$$C\underbrace{\epsilon \ldots \epsilon}_{t}$$

and viceversa, because of $\epsilon\epsilon = \epsilon$.

Definition 2. *We define a coding κ to be a mapping from Σ to Σ (not necessarily one-to-one) [6]. If $x = \sigma_1 \ldots \sigma_n$, we define $\kappa(x) = \kappa(\sigma_1) \ldots \kappa(\sigma_n)$ [6]. Finally, if $L \subseteq \Sigma^*$ is a language, we define $\kappa(L) = \{\kappa(x) : x \in L\}$ [6].*

Definition 3. ***CIRCUIT–HAMILTON–PATH***

INSTANCE: A graph $G = (V, E)$ and a string $\kappa(C)$ where C is a Boolean circuit and κ is a one-to-one mapping in the alphabet $\Sigma = \{+, -, 0, 1\}$ such that κ is defined by $\kappa(0) = +$ and $\kappa(1) = -$.

QUESTION: Does G have a Hamilton path where C is a succinct representation of G?

Theorem 3. *CIRCUIT–HAMILTON–PATH $\in NP$.*

Proof. We can convert in polynomial time the string $\kappa(C)$ to the Boolean circuit C encoded in binary since κ is a one-to-one mapping and we can easily revert it. We can check whether a simple path is a Hamilton path in polynomial time since *HAMILTON–PATH* $\in NP$. Moreover, we can check in polynomial time whether G has $2 \times n - 1$ nodes where $n = 2^b$ is a power of two. Furthermore, we can verify in polynomial time whether every ordered pair of vertices (u, v) complies with $(u, v) \in E$ if and only if C accepts the binary representations of the *b-bits* integers u, v as inputs. Finally, we can measure whether the size of C is upper bounded by b^k for a **"feasible"** positive integer k.

Definition 4. *We define a coding κ' to be a mapping over the alphabet $\Sigma = \{0, 1, +, -, \epsilon\}$ where ϵ is the empty string which can be considered as a symbol too [6]. κ' is defined by $\kappa'(+) = 0$, $\kappa'(-) = 1$, $\kappa'(0) = \epsilon$ and $\kappa'(1) = \epsilon$.*

Theorem 4. $P \neq NP$.

Proof. If $P = NP$, then *CIRCUIT–HAMILTON–PATH* $\in P$. As a consequence of Theorem 2, we obtain the language κ'(*CIRCUIT–HAMILTON–PATH*) is in *NEXP–hard* when we represent each instance of *CIRCUIT–HAMILTON–PATH* as $\kappa(C)G$ that would be $\kappa'(\kappa(C)G) = C\epsilon \ldots \epsilon$. P is closed under codings if and only if $P = NP$ [6]. Consequently, we would have this *NEXP–hard* language is in P if we assume that $P = NP$ [6]. However, this is not possible since there is not any *NEXP–hard* problem in P due to the Hierarchy Theorem [6]. Hence, we prove $P \neq NP$ by reduction ad absurdum.

4 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 [4]. 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 [6]. 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 [1].

References

1. Aaronson, S.: P $\stackrel{?}{=}$ NP. Electronic Colloquium on Computational Complexity, Report No. 4 (2017)
2. Arora, S., Barak, B.: Computational complexity: a modern approach. Cambridge University Press (2009)
3. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms. The MIT Press, 3rd edn. (2009)
4. Garey, M.R., Johnson, D.S.: Computers and Intractability: A Guide to the Theory of NP-Completeness. San Francisco: W. H. Freeman and Company, 1 edn. (1979)
5. Gasarch, W.I.: Guest column: The second P $\stackrel{?}{=}$ NP poll. ACM SIGACT News **43**(2), 53–77 (2012)
6. Papadimitriou, C.H.: Computational complexity. Addison-Wesley (1994)