On P versus NP

Frank Vega

March 17, 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 huge 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. To attack the P versus NP problem, the concept of coNP-completeness is very useful. We prove there is a problem in coNP-complete 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].

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

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].

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 show a new kind of reduction that we called the Conjunction reduction. Using this definition as an argument, 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]. In contraposition, we can define the best case running time of M. For $n \in \mathbb{N}$ we denote by $T'_M(n)$ the best case running time of M; that is

$$T'_M(n) = min\{t_M(w) : w \in \Sigma^n\}$$

where Σ^n is the set of all strings over Σ of length n [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 and better case running time functions, 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

$$O(g(n)) = \{f(n) : \textit{There exist positive constants } c \textit{ and } n_0$$

$$such\ that\ 0 \leq f(n) \leq c \times g(n)\ for\ all\ n \geq n_0\}$$

where O-notation provides an asymptotic upper bound [2]. For a given function $g(n)$, we denote by $\Omega(g(n))$ the set of functions

$$\Omega(g(n)) = \{f(n) : \textit{There exist positive constants } c \textit{ and } n_0$$

$$such\ that\ 0 \leq c \times g(n) \leq f(n)\ for\ all\ n \geq n_0\}$$

where Ω-notation provides an asymptotic lower 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

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

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 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 proper complexity function is a function g mapping a natural number to a natural number such that:

- g is nondecreasing;
- there exists a k-string Turing machine M such that on any input of length n, M halts after $O(n + g(n))$ steps, uses $O(g(n))$ space, and outputs $g(n)$ consecutive blanks [6].

A function $f : \Sigma^* \rightarrow \Sigma^*$ 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 $f : \{0,1\}^* \rightarrow \{0,1\}^*$ such that for all $x \in \{0,1\}^*$,

$$x \in L_1\ iff\ f(x) \in L_2$$

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–complete*, then L is in *coNP–hard* [2]. Moreover, if $L \in coNP$, then $L \in$ *coNP–complete* [2].

A principal *coNP–complete* problem is *CIRCUIT–UNSAT* [5]. An instance of *CIRCUIT–UNSAT* is a Boolean circuit C which is a directed acyclic graph $C = (V, E)$, where the nodes $V = \{1, \ldots, n\}$ are called the gates of C [6]. We can assume that all edges are of the form (i, j) where $i < j$ [6]. All nodes in the graph have in-degree (number of incoming edges) equal to 0, 1 and 2 [6]. Also, each gate $i \in V$ has a sort $c(i)$ associated with it, where $c(i) \in \{true, false, \wedge, \vee, \neg\} \cup \{x_1, x_2, \ldots\}$ [6]. If $c(i) \in \{true, false\} \cup \{x_1, x_2, \ldots\}$, then the in-degree of i is 0, that is, i must have no incoming edges [6]. Gates with no incoming edges are called the inputs of C [6]. If $c(i) = \neg$, then i has in-degree one [6]. If $c(i) \in \{\wedge, \vee\}$, then the in-degree of i must be two [6]. Finally, node n (the largest numbered gate in the circuit, which necessarily has no outgoing edges), is called the output gate of the circuit [6].

Let $X(C)$ be the set of all Boolean variables that appear in the circuit C (that is, $X(C) = \{x \in X : c(i) = x \textit{ for some gate } i \textit{ in } C\}$) [6]. We say that a truth assignment T is appropriate for C if it is defined for all the variables in $X(C)$ [6]. Given such a T, the truth value of gate $i \in V$, $T(i)$, is defined, by induction on i, as follows: If $c(i) = true$ then $T(i) = true$, and similarly if $c(i) = false$ [6]. If $c(i) \in X$, then $T(i) = T(c(i))$ [6]. If now $c(i) = \neg$, then there is a unique gate $j < i$ such that $(j, i) \in E$ [6]. By induction, we know $T(j)$, and then $T(i)$ is true if $T(j) = false$, and vice versa [6]. If $c(i) = \vee$, then there are two edges (j, i) and (j', i) entering i. $T(i)$ is then true if and only if at least one of $T(j)$, $T(j')$ is true [6]. If $c(i) = \wedge$, then $T(i)$ is true if and only if both $T(j)$ and $T(j')$ are true, where (j, i) and (j', i) are the incoming edges [6]. Finally, the value of the circuit, $T(C)$, is $T(n)$, where n is the output gate [6].

The *CIRCUIT–UNSAT* can be formulated as follows: Given a Boolean circuit C, is not there any truth assignment T, appropriate to C, such that $T(C) = true$?

2 Results

2.1 Conjunction reduction

Definition 2.1. In computer science, *O–notation* and *Ω–notation* are used to classify algorithms according to their running time or space requirements [2]. In this work, we are going to use the definition of the *O–notation* and *Ω–notation* based on the running time requirement. We say that a language L is in $O(g(n))$ when the worst case running time is in $O(g(n))$ for some algorithm deciding L where $g(n)$ is a proper complexity function [2]. We say that a language L is in $\Omega(g(n))$ when the best case running time is in $\Omega(g(n))$ for all the algorithms deciding L where $g(n)$ is a proper complexity function [2]. We say that a language L is not in $O(g(n))$ when there is not any possible algorithm which decides L in the worst case running time $O(g(n))$ where $g(n)$ is a proper complexity function [2].

Definition 2.2. For two languages L_1 and L_2, the concatenation language $L_1 L_2$ consists of all strings of the form vw where v is a string from L_1 and w is a string from L_2, or formally $L_1 L_2 = \{vw : v \in L_1, w \in L_2\}$.

Theorem 2.3. *For two languages L_1 and L_2, if $L_1 \notin O(g(n))$ and $L_2 \notin O(g(n))$, then $L_1 L_2 \notin O(g(n))$ where $g(n)$ is a proper complexity function.*

Proof. Suppose that $L_1 \notin O(g(n))$ and $L_2 \notin O(g(n))$, but $L_1 L_2 \in O(g(n))$. If $L_1 \notin O(g(n))$ and $L_2 \notin O(g(n))$, then for all the strings $x \in L_1$ and $y \in L_2$ the problem of deciding whether $xy \in L_1 L_2$ cannot not be decided in a running time $O(g(n))$ due to the properties of the *O–notation*. However, this is a contradiction since when we assume $L_1L_2 \in O(g(n))$, then the instances xy can be decided in a running time $O(g(n))$. For instance, let $y' \in L_2$ be the shortest string that belongs to L_2. Now, we can consider each string xy' for every $x \in L_1$. If $L_1L_2 \in O(g(n))$, then we can decide the instances $xy' \in L_1L_2$ in a running time $O(g(n))$. Since the string length of y' has a constant size, then this produces if we can decide $xy' \in L_1L_2$ in a running time $O(g(n))$, then we can decided every $x \in L_1$ in the same running time $O(g(n))$. This is due to the properties of the *O–notation* remain equivalents when the function $g(n)$ is multiplied by a constant [2]. But this result is not possible, because we assumed that $L_1 \notin O(g(n))$ as the initial premise. The same happens when we assume that $L_2 \notin O(g(n))$ and take $y' \in L_1$ as the shortest string that belongs to L_1. However, we must assume that both languages comply with $L_1 \notin O(g(n))$ and $L_2 \notin O(g(n))$ to guarantee that $L_1L_2 \notin O(g(n))$. Certainly, for some cases if $L_1 \in O(g(n))$ and $L_2 \notin O(g(n))$ or $L_1 \notin O(g(n))$ and $L_2 \in O(g(n))$ then $L_1L_2 \in O(g(n))$. Therefore, we obtain $L_1L_2 \notin O(g(n))$ when $L_1 \notin O(g(n))$ and $L_2 \notin O(g(n))$ as a consequence of applying the reduction ad absurdum. □

Definition 2.4. We say that two languages $L_1 \subseteq \{0,1\}^*$ and $L_2 \subseteq \{0,1\}^*$ are conjunctive reducible to a language $L_3 \subseteq \{0,1\}^*$, written $L_1 \wedge L_2 \leq_c L_3$, if for all $x \in \{0,1\}^*$, $y \in \{0,1\}^*$ and $z \in \{0,1\}^*$,

$$(x,y) \in L_1 \wedge (y,z) \in L_2 \textit{ iff } (x,y,z) \in L_3$$

where *iff* means "if and only if".

Theorem 2.5. *If $L_1 \wedge L_2 \leq_c L_3$ with $L_1 \notin O(g(n))$ and $L_2 \notin O(g(n))$, then $L_3 \notin O(g(n))$ where $g(n)$ is a proper complexity function.*

Proof. Suppose that $L_1 \wedge L_2 \leq_c L_3$ with $L_1 \notin O(g(n))$ and $L_2 \notin O(g(n))$, but $L_3 \in O(g(n))$. The concatenation language L_1L_2 cannot be decided in a running time $O(g(n))$ due to Theorem 2.3. However, if $L_3 \in O(g(n))$ then we can accept the instance (x,y,z) in a running time $O(g(n))$ when $(x,y) \in L_1 \wedge (y,z) \in L_2$. Consequently, we can accept the instance $(x,y)(y,z)$ in a running time $O(g(n))$ since we can do the same with (x,y,z) when $(x,y) \in L_1 \wedge (y,z) \in L_2$. In this way, we obtain the concatenation language L_1L_2 can be decided in a running time $O(g(n))$ under the assumption of $L_3 \in O(g(n))$. Therefore, for the sake of contradiction we have $L_3 \notin O(g(n))$. □

2.2 The Problem MINIMUM

Definition 2.6. Given a set S of n positive integers, *SEARCH–MINIMUM* is the problem of finding the minimum of S.

Lemma 2.7. *How many comparisons are necessary to determine the minimum of a set of n positive integers?*

Proof. We can easily obtain an upper bound of $n-1$ comparisons: examine each integer of the set 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 $n-1$ comparisons for the problem of determining the minimum [2]. Think of any algorithm that determines the minimum as a tournament among the elements [2]. Each comparison is a match in the tournament in which the smaller of the two elements wins [2]. The key observation is that every element except the winner must lose at least one match [2]. Hence, $n-1$ comparisons are necessary to determine the minimum, and the algorithm *SEARCH–MINIMUM* is optimal with respect to the number of comparisons performed [2]. □

Definition 2.8. Given a number x and a set S of n positive integers, *MINIMUM* is the problem of deciding whether x is the minimum of S.

Lemma 2.9. *MINIMUM* $\in \Omega(|S|)$ *where* $|\ldots|$ *represents the cardinality of the set.*

Proof. How many comparisons are necessary to determine whether some x is the minimum of a set of n positive integers? We can easily obtain an upper bound of n comparisons: find the minimum in the set 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 $n-1$ comparisons for the problem of determining the minimum and another obligatory comparison for checking whether that minimum is equal to x. □

Theorem 2.10. *MINIMUM* $\notin O(\lfloor\sqrt{|S|}\rfloor)$ *where* $\lfloor\ldots\rfloor$ *represents the floor function and* $|\ldots|$ *the cardinality of the set.*

Proof. As we mentioned above, the problem *MINIMUM* complies with *MINIMUM* $\in \Omega(|S|)$ and therefore *MINIMUM* $\notin O(\lfloor\sqrt{|S|}\rfloor)$, where $|S| = n$ is the cardinality of the set S with n positive integers. □

2.3 The Problem REPRESENTATION

Definition 2.11. A representation of a set S with n positive integers is a Boolean circuit C, such that C accepts the binary representation of a bit integer i (translated the bit 1 to true and 0 to false over the input variable gates) iff $i \in S$ where *iff* means "if and only if".

Definition 2.12. Given a set S of n positive integers and a Boolean circuit C, *REPRESENTATION* is the problem of deciding whether C is a representation of the set S.

Theorem 2.13. *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]. 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. □

Theorem 2.14. *REPRESENTATION* $\notin O(\lfloor\sqrt{|S|}\rfloor)$ *where* $\lfloor\ldots\rfloor$ *represents the floor function and* $|\ldots|$ *the cardinality of the set.*

Proof. Since the empty set cannot be represented by a Boolean circuit C with some truth assignment T appropriate to C such that $T(C) = true$, then we could make a polynomial time reduction as follows:

$$C \in \textit{CIRCUIT–UNSAT} \textit{ iff } (\emptyset, C) \in \textit{REPRESENTATION}.$$

However, this reduction can be made in constant time. That means we cannot decide every instance $(\emptyset, C) \in REPRESENTATION$ in constant time, because that would mean we can decide *CIRCUIT–UNSAT* in constant time. In addition, we cannot decide the language *CIRCUIT–UNSAT* in constant time according to the Theorem 2.13. Nevertheless, from an instance $(\emptyset, C) \in REPRESENTATION$, we would have $S = \emptyset$ and $|S| = 0$. Thus, we can assure if $REPRESENTATION \in O(\lfloor\sqrt{|S|}\rfloor)$, then we could decide *CIRCUIT–UNSAT* in constant time. For that reason, we can confirm $REPRESENTATION \notin O(\lfloor\sqrt{|S|}\rfloor)$. □

2.4 The Problem SUCCINCT-MINIMUM

Definition 2.15. Given a positive integer x and a Boolean circuit C, we define *SUCCINCT–MINIMUM* as the problem of deciding whether x is the smallest bit integer which accepts C as input.

Definition 2.16. *REPRESENTATION–MINIMUM* is equal to

$$MINIMUM \wedge REPRESENTATION \leq_c$$

$$REPRESENTATION\text{–}MINIMUM$$

such that for every instance (x, S, C) of the language *REPRESENTATION–MINIMUM* we have the following property,

$$(x, S) \in MINIMUM \wedge (S, C) \in REPRESENTATION$$

$$\textit{iff } (x, S, C) \in REPRESENTATION\text{–}MINIMUM.$$

Theorem 2.17. *$REPRESENTATION\text{–}MINIMUM \notin O(\lfloor\sqrt{|S|}\rfloor)$ where $\lfloor \ldots \rfloor$ represents the floor function and $|\ldots|$ the cardinality of the set.*

Proof. As result of Theorems 2.5, 2.10, 2.14 and Definition 2.16, then we have the following statement $REPRESENTATION\text{–}MINIMUM \notin O(\lfloor\sqrt{|S|}\rfloor)$. □

Theorem 2.18. *$SUCCINCT\text{–}MINIMUM \notin O(\lfloor\sqrt{|S|}\rfloor)$ where S is the set that represents the Boolean circuit C, $\lfloor \ldots \rfloor$ represents the floor function and $|\ldots|$ the cardinality of the set.*

Proof. If we have $REPRESENTATION\text{–}MINIMUM \notin O(\lfloor\sqrt{|S|}\rfloor)$ then *SUCCINCT–MINIMUM* cannot be decided in a running time $O(\lfloor\sqrt{|S|}\rfloor)$. Indeed, if *SUCCINCT–MINIMUM* can be decided in a running time $O(\lfloor\sqrt{|S|}\rfloor)$ by a deterministic Turing machine, then $REPRESENTATION\text{–}MINIMUM \in O(\lfloor\sqrt{|S|}\rfloor)$. Since this is contradiction according to Theorem 2.17, then $SUCCINCT\text{–}MINIMUM \notin O(\lfloor\sqrt{|S|}\rfloor)$. □

Theorem 2.19. *$SUCCINCT\text{–}MINIMUM \notin P$.*

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 $SUCCINCT\text{–}MINIMUM \notin O(\lfloor\sqrt{|S|}\rfloor)$, then we could not decide every instance of *SUCCINCT–MINIMUM* in polynomial time. □

Theorem 2.20. *$SUCCINCT\text{–}MINIMUM \in coNP$.*

Proof. If $(x,C) \notin$ *SUCCINCT–MINIMUM*, then it would exist a positive integer y such that $y < x$ and C accepts the bit integer y or simply it would be the case when C does not accept the input bit integer x. Since we can evaluate whether C accepts or not the bit integers x and y in polynomial time and we have that y is quadratic polynomially bounded by x, then we can confirm *SUCCINCT–MINIMUM* $\in$ *coNP* due to the verification of $y < x$ and the evaluation on the Boolean circuit can be done in polynomial time. □

Theorem 2.21. *SUCCINCT–MINIMUM* $\in$ *coNP–complete.*

Proof. Given a Boolean circuit C, we can check whether C does not accept the positive integer $2^b - 1$ where b is the number of input gates in C. By input gates, we actually mean the input gates which are associated to some variable. In that case, we create a succinct Boolean circuit C' which only accepts the bit integer $2^b - 1$ and has the same number of input gates of C. We combine C with C' coinciding their input gates into a new Boolean circuit C'' which only accepts when C or C' accept. This is possible just adding a gate *OR* between the output gates of C and C'. The instance of the positive integer $2^b - 1$ and the final Boolean circuit C'' belongs to *SUCCINCT–MINIMUM* if and only if C is in *CIRCUIT–UNSAT*. Certainly, $2^b - 1$ is the minimum of the set S that represents C'' if there is not any other input which C'' accepts. In addition, C'' accepts the positive integer $2^b - 1$ because of the construction of C' under C. Since we can create the succinct Boolean circuit C' and evaluate C on the input $2^b - 1$ in polynomial time, then we can reduce *CIRCUIT–UNSAT* to *SUCCINCT–MINIMUM* in polynomial time. *CIRCUIT–UNSAT* is a known *coNP–complete* problem [6]. Hence, the language *SUCCINCT–MINIMUM* is in *coNP–hard* [6]. As result of Theorem 2.20, we obtain *SUCCINCT–MINIMUM* is also in *coNP* and thus, the proof is completed. □

Theorem 2.22. $P \neq NP$.

Proof. If any single *coNP–complete* 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. 2, 3, 4, 5, 6

[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, 4, 8

[6] CHRISTOS H. PAPADIMITRIOU: *Computational Complexity*. Addison-Wesley, 1994. 1, 2, 3, 4, 6, 7, 8

[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.