P VERSUS NP

Frank Vega

P VERSUS NP

FRANK VEGA

Abstract. P versus NP is one of the most important and unsolved problems in computer science. This consists on knowing the answer of the following question: Is P equal to NP? This incognita was first mentioned in a letter written by Kurt Gödel to John von Neumann in 1956. However, the precise statement of the P versus NP problem was introduced in 1971 by Stephen Cook. Since that date, all efforts to find a proof for this huge problem have failed. It is currently accepted that a positive answer for P versus NP would have tremendous effect not only in computer science, but also in mathematics, biology, etc.

We consider two new complexity classes that are called equivalent-P and equivalent-NP which have a close relation to this problem. The class equivalent-P contains those languages that are ordered-pairs of instances of two specific problems in P, such that the elements of each ordered-pair have the same solution, which means, the same certificate. The class equivalent-NP has almost the same definition, but in each case we define the pair of languages explicitly in NP. We demonstrate that equivalent-P = equivalent-NP, but this brings as a consequence that P is equal to NP.

Introduction

P versus NP is a major unsolved problem in computer science. This problem was introduced in 1971 by Stephen Cook [2]. It is considered by many to be the most important open problem in the field [5]. 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 [5].

In 1936, Turing developed his theoretical computational model [10]. 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 [12]. 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 [12].

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 [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].

In the computational complexity theory, the class P contains those languages that can be decided in polynomial-time by a deterministic Turing machine [8]. The class NP consists on those languages that can be decided in polynomial-time by a nondeterministic Turing machine [8].

2000 *Mathematics Subject Classification.* 68-XX, 68Qxx, 68Q15.

Key words and phrases. P, NP, NP-complete, polynomial-time verifier.

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 not, 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 [7].

The complexity class *NP–complete* was defined by Cook in his seminal paper [2]. The class *NP-complete* is a set of problems of which any other NP problem can be reduced in polynomial-time, but whose solution may still be verified in polynomial-time [8]. If any single *NP–complete* problem can be solved in polynomial-time, then every NP problem has a polynomial-time algorithm [3]. All efforts to find polynomial-time algorithms for the *NP–complete* problems have failed [5].

We will define two new complexity classes that will be called *equivalent-P* and *equivalent-NP* and denoted as $\equiv P$ and $\equiv NP$ respectively. Moreover, we will prove that every problem in NP is also in $\equiv NP$ and each problem in *equivalent-NP* is in NP too. In this way, we conclude that both classes, NP and $\equiv NP$, are equal. Besides, we will show the complexity class $\equiv P$ is closed under reductions. We will also prove that there is an *NP-complete* problem in $\equiv P$. In this way, all languages in NP are reduced to $\equiv P$, but this implies $NP \subseteq \equiv P$ due to $\equiv P$ is closed under reductions. Since we also demonstrate that $\equiv P \subseteq NP$, then we can sustain that $\equiv P = NP$. Consequently, we show that $\equiv P$ is equal to $\equiv NP$. Then, this would also imply that $P = NP$.

1. Basic theoretical notions

1.1. **The class NP-complete.** We define $\{0,1\}^*$ as the infinite set of binary strings [3]. We say that a language L_1 is polynomial-time reducible to a language L_2, 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 $x \in \{0,1\}^*$,

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

An important complexity class is *NP-complete* [8]. A language $L \subseteq \{0,1\}^*$ is *NP-complete* if

(1) $L \in NP$, and
(2) $L' \leq_p L$ for every $L' \in NP$.

Furthermore, if L is a language such that $L' \leq_p L$ for some $L' \in$ *NP-complete*, then L is in *NP-hard* [3]. Moreover, if $L \in NP$, then $L \in$ *NP-complete* [3]. No polynomial-time algorithm has yet been discovered for any *NP-complete* problem [5].

A principal *NP-complete* problem is SAT [6]. An instance of SAT is a Boolean formula ϕ which is composed of

(1) Boolean variables: $x_1, x_2, \ldots, x_n$;
(2) Boolean connectives: Any Boolean function with one or two inputs and one output, such as $\wedge$(AND), $\vee$(OR), $\rightarrow$(NOT), $\Rightarrow$(implication), $\Leftrightarrow$(if and only if);
(3) and parentheses.

A truth assignment for a Boolean formula ϕ is a set of values for the variables in ϕ. A satisfying truth assignment is a truth assignment that causes ϕ to be evaluated

as true. A formula with a satisfying truth assignment is a satisfiable formula. The problem SAT asks whether a given Boolean formula is satisfiable [6].

Another *NP-complete* language is $3CNF$ satisfiability, or $3SAT$ [3]. We define $3CNF$ satisfiability using the following terms. A literal in a Boolean formula is an occurrence of a variable or its negation. A Boolean formula is in conjunctive normal form, or CNF, if it is expressed as an AND of clauses, each of which is the OR of one or more literals. A Boolean formula is in 3-conjunctive normal form or $3CNF$, if each clause has exactly three distinct literals.

For example, the Boolean formula

$$(x_1 \vee \neg\, x_1 \vee \neg\, x_2) \wedge (x_3 \vee x_2 \vee x_4) \wedge (\neg\, x_1 \vee \neg\, x_3 \vee \neg\, x_4)$$

is in $3CNF$. The first of its three clauses is $(x_1 \vee \neg\, x_1 \vee \neg\, x_2)$, which contains the three literals x_1, $\neg\, x_1$, and $\neg\, x_2$. In $3SAT$, it is asked whether a given Boolean formula ϕ in $3CNF$ is satisfiable.

It can be demonstrated that many problems belong to *NP-complete* using the polynomial-time reduction from $3SAT$ [6]. For example, the well known problem *ONE-IN-THREE 3SAT* which is defined as follows: Given a Boolean formula ϕ in $3CNF$, is there a truth assignment such that each clause in ϕ has exactly one true literal?

1.2. **Problems in P.** Another special case is the class of problems where each clause contains XOR (i.e. exclusive or) rather than (plain) OR operators. This is in P, since a *XOR SAT* formula can also be viewed as a system of linear equations mod 2, and can be solved in cubic time by Gaussian elimination [11]. We represent the XOR function inside a Boolean formula as $\oplus$. The problem *XOR 3SAT* will be equivalent to *XOR SAT*, but the clauses in the Boolean formula have exactly three distinct literals. Since $a \oplus b \oplus c$ is evaluated as true if and only if exactly 1 or 3 members of $\{a, b, c\}$ are true, then each solution of the problem *ONE-IN-THREE 3SAT* for a given $3CNF$ formula is also a solution of the problem *XOR 3SAT* and in turn each solution of *XOR 3SAT* is a solution of $3SAT$.

In addition, a Boolean formula is in 2-conjunctive normal form, or $2CNF$, if it is in CNF and each clause has exactly two distinct literals. There is a well known problem called $2SAT$. In $2SAT$, it is asked whether a given Boolean formula ϕ in $2CNF$ is satisfiable. This language is in P [1].

2. Definition of $\equiv P$

Let L be a language and M a Turing machine. We say that M is a verifier for L if L can be written as

$$L = \{x : (x, y) \in R \textit{ for some } y\}$$

where R is a polynomially balanced relation decided by M [12]. According to Cook's Theorem, a language L is in NP if and only if it has a polynomial-time verifier [12].

Definition 2.1. *We say that a language L belongs to $\equiv P$ if there are two languages $L_1 \in P$ and $L_2 \in P$ and two deterministic Turing machines M_1 and M_2, where M_1 and M_2 are the polynomial-time verifiers of L_1 and L_2 respectively, such that*

$$L = \{(x, y) : \exists z \textit{ such that } M_1(x, z) = \text{“}yes\text{”} \textit{ and } M_2(y, z) = \text{“}yes\text{”}\}.$$

We will call the complexity class $\equiv P$ as "equivalent-P". We will represent the language L in $\equiv P$, which consists on the pairs of instances (x, y) where $x \in L_1$ and $y \in L_2$, as (L_1, L_2). The order of the pairs of instances of a problem $\equiv P$ is really important, because we will not consider the languages (L_1, L_2) and (L_2, L_1) as equal when $L_1 \neq L_2$.

3. Reduction in $\equiv P$

We say that a language L_1 is logarithmic-space reducible to a language L_2, if there is a logarithmic-space computable function $f : \{0,1\}^* \to \{0,1\}^*$ such that for all $x \in \{0,1\}^*$,

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

The logarithmic-space reduction is frequently used for P and below [12]. There is a different kind of reduction for $\equiv P$: the *e-reduction*.

Definition 3.1. *Given two languages L_1 and L_2, where the instances of L_1 and L_2 are ordered-pairs of strings, we say that the language L_1 is e-reducible to the language L_2, written $L_1 \leq_{\equiv} L_2$, if there are two logarithmic-space computable functions $f : \{0,1\}^* \to \{0,1\}^*$ and $g : \{0,1\}^* \to \{0,1\}^*$ such that for all $x \in \{0,1\}^*$ and $y \in \{0,1\}^*$,*

$$(x, y) \in L_1 \text{ if and only if } (f(x), g(y)) \in L_2.$$

We say that a complexity class C is closed under reductions if, whenever L_1 is reducible to L_2 and $L_2 \in C$, then $L_1 \in C$ [12].

Theorem 3.2. *$\equiv P$ is closed under reductions.*

Proof. Let L and L' be two arbitrary languages, where their instances are ordered-pairs of strings. Suppose that $L \leq_{\equiv} L'$ where L' is in $\equiv P$. We will show that L is in $\equiv P$ too.

By definition of $\equiv P$, there are two languages $L'_1 \in P$ and $L'_2 \in P$, such that for each $(v, w) \in L'$ we have that $v \in L'_1$ and $w \in L'_2$. Moreover, there are two Turing machines M'_1 and M'_2 which are the polynomial-time verifiers of L'_1 and L'_2 respectively. For each $(v, w) \in L'$, there will be a succinct certificate z, such that $M'_1(v, z) =$ "*yes*" and $M'_2(w, z) =$ "*yes*". Besides, by definition of *e-reduction*, there are two logarithmic-space computable functions $f : \{0,1\}^* \to \{0,1\}^*$ and $g : \{0,1\}^* \to \{0,1\}^*$ such that for all $x \in \{0,1\}^*$ and $y \in \{0,1\}^*$,

$$(x, y) \in L \text{ if and only if } (f(x), g(y)) \in L'.$$

Based on this preliminary information, we can support that there are two languages $L_1 \in P$ and $L_2 \in P$, such that for each $(x, y) \in L$ we have that $x \in L_1$ and $y \in L_2$. Indeed, we can define L_1 and L_2 as the ordered-pairs of strings $(f^{-1}(v), g^{-1}(w))$, such that $f^{-1}(v) \in L_1$ and $g^{-1}(w) \in L_2$ if and only if $v \in L'_1$ and $w \in L'_2$. Certainly, for all $x \in \{0,1\}^*$ and $y \in \{0,1\}^*$, we can decide whether $x \in L_1$ or $y \in L_2$ in polynomial-time just verifying that $f(x) \in L'_1$ or $g(y) \in L'_2$ respectively, since $L'_1 \in P$, $L'_2 \in P$, and $SPACE(\log n) \in P$ [12].

Furthermore, there are two deterministic Turing machines M_1 and M_2 which are the polynomial-time verifiers of L_1 and L_2 respectively. For each $(x, y) \in L$, there will be a succinct certificate z such that $M_1(x, z) =$ "*yes*" and $M_2(y, z) =$ "*yes*". Indeed, we can know whether $M_1(x, z) =$ "*yes*" and $M_2(y, z) =$ "*yes*" just verifying whether $M'_1(f(x), z) =$ "*yes*" and $M'_2(g(y), z) =$ "*yes*". Certainly, for every triple

of strings (x, y, z), we can define the polynomial-time computation from the verifiers M_1 and M_2 as $M_1(x, z) = M_1'(f(x), z)$ and $M_2(y, z) = M_2'(g(y), z)$, since we can evaluate $f(x)$ and $g(y)$ in polynomial-time because of $SPACE(\log n) \in P$ [12]. In addition, $max(|f(x)|, |g(y)|)$ is polynomially bounded by $min(|x|, |y|)$ where $|\dots|$ is the string length function, due to the logarithmic-space computable functions f and g cannot produce an exponential amount of symbols in relation to the size of the input. Consequently, $|z|$ is polynomially bounded by $min(|x|, |y|)$, because $|z|$ would be polynomially bounded by $min(|f(x)|, |g(y)|)$. Hence, we have just proved the necessary properties to state that L would be in $\equiv P$. □

4. $\equiv P = NP$

Theorem 4.1. *$\equiv P \subseteq NP$.*

Proof. Every instance $(x, y) \in L$ of a language L in $\equiv P$ can always be checked by a polynomial-time verifier M. The existence of a polynomial-time verifier for a language is sufficient to show that this language is in NP [12]. Certainly, we will always obtain a succinct certificate z such that $M(x, y, z) =$ "*yes*" for every ordered-pair of instances $(x, y) \in L$. Indeed, we can define M as $M(x, y, z) =$ "*yes*" if and only if $M_1(x, z) =$ "*yes*" and $M_2(y, z) =$ "*yes*". Since M_1 and M_2 are the polynomial-time verifiers of Definition 2.1, then $L \in NP$. □

We will define $\equiv$ *ONE-IN-THREE 3SAT* as follows,

$$\equiv \textit{ONE-IN-THREE 3SAT} = \{(\phi, \phi) : \phi \in \textit{ONE-IN-THREE 3SAT}\}.$$

Theorem 4.2. *$\equiv$ ONE-IN-THREE 3SAT $\in$ NP-complete.*

Proof. Given a Boolean formula ϕ in $3CNF$, we can make a simple polynomial-time reduction from *ONE-IN-THREE 3SAT* as follows,

$\phi \in$ ONE-IN-THREE 3SAT if and only if $(\phi, \phi) \in \equiv$ ONE-IN-THREE 3SAT.

$\equiv$ ONE-IN-THREE 3SAT is in NP, because *ONE-IN-THREE 3SAT* is in NP too. Since we have already proved that *$\equiv$ ONE-IN-THREE 3SAT* is in *NP-hard* and *$\equiv$ ONE-IN-THREE 3SAT* must be in NP, then *$\equiv$ ONE-IN-THREE 3SAT $\in$ NP-complete.* □

Definition 4.3. *3XOR-2SAT is a problem in $\equiv P$, where every instance (ψ, φ) is an ordered-pair of Boolean formulas, such that if $(\psi, \varphi) \in$ 3XOR-2SAT, then $\psi \in$ XOR 3SAT and $\varphi \in 2SAT$. By the definition of $\equiv P$, this language will be the ordered-pairs of instances of XOR 3SAT and $2SAT$ such that they would have the same satisfying truth assignment using the same variables.*

Theorem 4.4. *$\equiv$ 3XOR-2SAT $\in$ NP-complete.*

Proof. Given an arbitrary Boolean formula ϕ in $3CNF$ of m clauses, we will iterate for $i = 1, 2, 3, \dots, m$ over each clause $c_i = (x \vee y \vee z)$ in ϕ, where x, y and z are literals, creating the following formulas,

$$Q_i = (x \oplus y \oplus z)$$

$$P_i = (\rightharpoonup x \vee \rightharpoonup y) \wedge (\rightharpoonup y \vee \rightharpoonup z) \wedge (\rightharpoonup x \vee \rightharpoonup z).$$

Since Q_i is evaluated as true if and only if exactly 1 or 3 members of $\{x, y, z\}$ are true and P_i is evaluated as true if and only if exactly 1 or 0 members of $\{x, y, z\}$

are true, then we obtain the clause c_i has exactly one true literal if and only if both formulas Q_i and P_i have the same satisfying truth assignment.

Hence, we can construct the Boolean formulas ψ and φ as the conjunction of Q_i or P_i for every clause c_i in ϕ, that is, $\psi = Q_1 \wedge \ldots \wedge Q_m$ and $\varphi = P_1 \wedge \ldots \wedge P_m$. Finally, we obtain that,

$$(\phi, \phi) \in \equiv \textit{ ONE-IN-THREE 3SAT if and only if } (\psi, \varphi) \in \textit{3XOR-2SAT}.$$

Moreover, there are two logarithmic-space computable functions $f : \{0,1\}^* \to \{0,1\}^*$ and $g : \{0,1\}^* \to \{0,1\}^*$ such that $f(\langle\phi\rangle) = \langle\psi\rangle$ and $g(\langle\phi\rangle) = \langle\varphi\rangle$. Indeed, we only need a logarithmic-space to analyze every time each clause c_i in the input ϕ and generate Q_i or P_i to the output, since the complexity class $SPACE(\log n)$ will not take the length of the input and the output into consideration [12]. Then, we have proved that $\equiv$ *ONE-IN-THREE 3SAT* $\leq_{\equiv}$ *3XOR-2SAT*.

The *e-reduction* is also a polynomial-time reduction, since $SPACE(\log n) \in P$. Consequently, $\equiv$ *3XOR-2SAT* is in *NP-hard* due to the previous *e-reduction* from $\equiv$ *ONE-IN-THREE 3SAT*. Moreover, as result of Theorem 4.1, we obtain that $\equiv$ *3XOR-2SAT* is in NP, and therefore, $\equiv$ *3XOR-2SAT* is in *NP-complete*. □

Theorem 4.5. $\equiv P = NP$.

Proof. Since $\equiv$ *3XOR-2SAT* is complete for NP, thus all language in NP reduce to $\equiv P$. Since $\equiv P$ is closed under reductions, it follows that $NP \subseteq \equiv P$. The inclusion $\equiv P \subseteq NP$ follows from Theorem 4.1. Hence, if $\equiv P \subseteq NP$ and $NP \subseteq \equiv P$, then $\equiv P = NP$ [3]. □

5. $P = NP$

Definition 5.1. *We say that a language L belongs to $\equiv NP$ if there are two languages $L_1 \in NP$ and $L_2 \in NP$ and two deterministic Turing machines M_1 and M_2, where M_1 and M_2 are the polynomial-time verifiers of L_1 and L_2 respectively, such that*

$$L = \{(x, y) : \exists z \textit{ such that } M_1(x, z) = \text{“}yes\text{”} \textit{ and } M_2(y, z) = \text{“}yes\text{”}\}.$$

We will call the complexity class $\equiv NP$ as “equivalent-NP”. Indeed, this definition fulfills the same parameters than Definition 2.1, except that in this one the pairs of languages are explicitly in NP.

Theorem 5.2. $\equiv NP = NP$.

Proof. A language L in $\equiv NP$ will always have a polynomial-time verifier M. Certainly, for every ordered-pair of instances $(x, y) \in L$ there is always a succinct certificate z such that $M(x, y, z) =$ “*yes*”, because we can define M as $M(x, y, z) =$ “*yes*” if and only if $M_1(x, z) =$ “*yes*” and $M_2(y, z) =$ “*yes*”. Since M_1 and M_2 are the polynomial-time verifiers of Definition 5.1, then $L \in NP$. Indeed this proof is the same as Theorem 4.1, except that in this one there is a change in the classes, that is, P is replaced by NP in this case. Hence, we obtain that $\equiv NP \subseteq NP$.

We can also transform every language $L \in NP$ as follows,

$$\equiv L = \{(x, y) : xy \in L \textit{ and } |x| = \lfloor \frac{|xy|}{2} \rfloor, |y| = \lceil \frac{|xy|}{2} \rceil\}.$$

where xy is the concatenation of the strings x and y and $|\ldots|$, $\lfloor\ldots\rfloor$ and $\lceil\ldots\rceil$ are the string length, floor and ceiling functions respectively. For the short strings xy in L which consist on only one symbol, the value of x can be the empty string.

We can also define the language $\equiv L$ as the ordered-pairs of instances of two languages L_1 and L_2, such that L_1 would contain the first half part of the elements in L while L_2 would have the second half part. L_1 and L_2 would be in NP, because the membership in L_1 of the first half part x_1 from an element $x \in L$ can be verified in polynomial-time using as certificate the strings x and z where z would also be the certificate of x. The same happens with the second half part x_2, because we can also check it in polynomial-time with x and z too. Certainly, they would use the certificate x to check when x_1 or x_2 is the half part of the string x, and they would use the certificate z to prove that x is in L. Furthermore, the strings x and z would be polynomially bounded by x_1 and x_2 due to $|x| \leq (2 \times |x_1| + 1) \leq (2 \times |x_2| + 1)$. Since z is polynomially bounded by x, then it would be polynomially bounded by x_1 and x_2 too. Consequently, $\equiv L$ will be in $\equiv NP$, because the ordered-pairs of instances (x_1, x_2) would have the same certificate, that is the strings x and z, when $x \in L$ and z is the certificate of x. However, every instance (x, y) of $\equiv L$ can be represented by the string xy, because we can easily recognize x and y from xy since

$$|x| = \lfloor \frac{|xy|}{2} \rfloor \ \textit{and}\ |y| = \lceil \frac{|xy|}{2} \rceil.$$

Nevertheless, if we represent it in this way, then $\equiv L = L$, and thus, L would be in $\equiv NP$. Since we took L as an arbitrary language in NP, then it follows that $NP \subseteq \equiv NP$. Hence, if $\equiv NP \subseteq NP$ and $NP \subseteq \equiv NP$, then $\equiv NP = NP$ [3]. □

Let A and B be subsets of $\{0,1\}^*$. We define the join, $A \oplus B$, as the union of $\{0x : x \in A\}$ and $\{1y : y \in B\}$ [4].

Theorem 5.3. *$P = NP$.*

Proof. As result of Theorems 4.5 and 5.2, we obtain that $\equiv P$ is equal to $\equiv NP$. This means that there is one bijective function $h :\equiv NP \to \equiv P$, because both classes are equal [9]. For every language L in NP, we can create one language (L, L) in $\equiv NP$, such that an instance (x, y) is in (L, L) if and only if $x = y$ and $x \in L$. Since the certificate of $x \in L$ will be the same one of $(x, x) \in (L, L)$, then (L, L) will never be an empty language when L is not an empty set. By the bijective definition of h, (L, L) is connected to a single language (L_1, L_2) in $\equiv P$, where (L, L) would be the unique language in $\equiv NP$ that is related to (L_1, L_2) through h. Hence, there will be an injective and total function $f : NP \to P$, because we can always make a connection between this unique and arbitrary language L in NP with the following language $L' = L_1 \oplus L_2$ in P, since $L_1 \in P$, $L_2 \in P$, and P is closed under joins [4]. Certainly, the bijective definition of h guarantees the injective property of the function f. Moreover, if there are two languages $L_3 \in P$ and $L_4 \in P$ such that $L' = L_3 \oplus L_4$, then $L_1 = L_3$ and $L_2 = L_4$, because the join between sets is an injective function too [4].

On the other hand, there is another injective and total function $g : P \to NP$, that will be the identity function over the languages in P. Certainly, we can define g as $g(L) = L$ for every language $L \in P$, because a language L in P will also be in NP and the identity function is always injective [12]. However, the existence of these two injective and total functions implies that P and NP will have the same cardinality when we apply the Schröder Bernstein Theorem [9]. Nevertheless, we already know that $P \subseteq NP$, and therefore, if P and NP have the same cardinality, then we can conclude that $P = NP$. □

Conclusions

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 [6]. Even though this proof might not be a practical solution, it shows in a formal way that many currently mathematical problems can be solved efficiently, including those in *NP–complete*.

At the same time, this demonstration would represent a very significant advance in computational complexity theory and provide guidance for future research. On the one hand, it proves that most of the existing cryptosystems such as the public-key cryptography are not safe [8]. On the other hand, we will be able to find a formal proof for every theorem which has a proof of a reasonable length by a feasible algorithm.

Acknowledgement

I thank Professor Magda La Serna for her English Lessons.

References

1. Bengt Aspvall, Michael F. Plass, and Robert E. Tarjan, *A Linear-Time Algorithm for Testing the Truth of Certain Quantified Boolean Formulas*, Information Processing Letters **8** (1979), no. 3, 121–123.
2. Stephen A. Cook, *The complexity of theorem-proving procedures*, Proceedings of the 3rd IEEE Symp. on the Foundations of Computer Science, 1971, pp. 151–158.
3. Thomas H. Cormen, Charles Eric Leiserson, Ronald L. Rivest, and Clifford Stein, *Introduction to Algorithms*, 2 ed., MIT Press, 2001.
4. Lance Fortnow, *The Union of Complexity Classes*, 2002, available at `http://blog.computationalcomplexity.org/2002/11/union-of-complexity-classes.html`.
5. ———, *The Status of the P versus NP Problem*, Communications of the ACM **52** (2009), no. 9, 78–86.
6. Michael R. Garey and David S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, 1 ed., San Francisco: W. H. Freeman and Company, 1979.
7. William I. Gasarch, *The P=?NP poll*, SIGACT News **33** (2002), no. 2, 34–47.
8. Oded Goldreich, *P, Np, and Np-Completeness*, Cambridge: Cambridge University Press, 2010.
9. Arie Hinkis, *Proofs of the Cantor-Bernstein theorem. A mathematical excursion*, vol. 45, Heidelberg: Birkhäuser/Springer, 2013.
10. David Leavitt, *The man who knew too much: Alan turing and the invention of the computer*, Nueva York: W. W. Norton, 2006.
11. Cristopher Moore and Stephan Mertens, *The Nature of Computation*, Oxford University Press, 2011.
12. Christos H. Papadimitriou, *Computational Complexity*, Addison-Wesley, 1994.