P VERSUS NP

FRANK VEGA*

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? This question was first mentioned in a letter written by John Nash to the National Security Agency in 1955. 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 problem have failed. Another major complexity class is Sharp-P. Whether P = Sharp-P is another fundamental question that it is as important as it is unresolved. If any single Sharp-P-complete problem can be solved in polynomial time, then every NP problem has a polynomial time algorithm. The problem Sharp-MONOTONE-2SAT is known to be Sharp-P-complete. We prove Sharp-MONOTONE-2SAT is in P. In this way, we demonstrate the P versus NP problem.

Key words. Complexity Classes, Completeness, Polynomial Time, Counting Solutions, Number Theory

AMS subject classifications. 68Q15, 68Q17, 68R01

1. Introduction.

The P versus NP problem is a major unsolved problem in computer science [5]. This 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]. 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 [5].

In 1936, Turing developed his theoretical computational model [18]. The deterministic and nondeterministic Turing machines have become in two of the most important definitions related to this theoretical model for computation [18]. A deterministic Turing machine has only one next action for each step defined in its program or transition function [18]. 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 [18].

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 [6]. A complexity class is a set of problems, which are represented as a language, grouped by measures such as the running time, memory, etc [6].

The set of languages decided by deterministic Turing machines within time f is an important complexity class denoted $TIME(f(n))$ [14]. In addition, the complexity class $NTIME(f(n))$ consists in those languages that can be decided within time f by nondeterministic Turing machines [14]. 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 [14]. At the same time, NP consists in all languages in $NTIME(n^k)$ for every possible positive constant k [14].

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

*Joysonic, Uzun Mirkova 5, Belgrade, 11000, Serbia (vega.frank@gmail.com).

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 [9]. It is fully expected that $P \neq NP$ [14]. Indeed, if $P = NP$ then there are stunning practical consequences [14]. For that reason, $P = NP$ is considered as a very unlikely event [14]. Certainly, P versus NP is one of the greatest open problems in science and a correct solution for this incognita will have a great impact not only for computer science, but for many other fields as well [1].

2. Theory. Let Σ be a finite alphabet with at least two elements, and let Σ^* be the set of finite strings over Σ [3]. A Turing machine M has an associated input alphabet Σ [3]. For each string w in Σ^* there is a computation associated with M on input w [3]. We say that M accepts w if this computation terminates in the accepting state, that is $M(w) = $ "*yes*" [3]. 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 [3].

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 [3]. 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 [3]. We say that M runs in polynomial time if there is a constant k such that for all n, $T_M(n) \leq n^k + k$ [3]. 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 [6]. 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 [3]. 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 [14].

There is a close relation between the polynomial time verifiers and another important class: The complexity class *Sharp-P* (denoted as $\#P$). Let $\{0,1\}^*$ be the infinite set of binary strings, a function $f : \{0,1\}^* \to \mathbb{N}$ is in $\#P$ if there exists a polynomial time verifier M such that for every $x \in \{0,1\}^*$,

$$f(x) = |\{y : M(x, y) = \text{"}yes\text{"}\}|$$

where $|\ldots|$ denotes the cardinality set function [3].

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 [18]. 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* [11]. 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 $L' \in NP\text{–}complete$, then L is *NP–hard* [6]. Moreover, if $L \in NP$, then $L \in NP\text{–}complete$ [6]. A principal *NP–complete* problem is SAT [8]. 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), $\rightharpoondown$(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 [8]. We define a CNF Boolean formula using the following terms. A literal in a Boolean formula is an occurrence of a variable or its negation [6]. 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 [6]. A Boolean formula is in 3-conjunctive normal form or $3CNF$, if each clause has exactly three distinct literals [6].

For example, the Boolean formula:

$$(x_1 \vee \rightharpoondown x_1 \vee \rightharpoondown x_2) \wedge (x_3 \vee x_2 \vee x_4) \wedge (\rightharpoondown x_1 \vee \rightharpoondown x_3 \vee \rightharpoondown x_4)$$

is in $3CNF$. The first of its three clauses is $(x_1 \vee \rightharpoondown x_1 \vee \rightharpoondown x_2)$, which contains the three literals x_1, $\rightharpoondown x_1$, and $\rightharpoondown x_2$. Another relevant *NP–complete* language is $3CNF$ satisfiability, or $3SAT$ [6]. In $3SAT$, it is asked whether a given Boolean formula ϕ in $3CNF$ is satisfiable.

In computational complexity theory, *#P–complete* is another complexity class. A problem is *#P–complete* if and only if it is in $\#P$, and every problem in $\#P$ can be reduced to it by a polynomial time counting reduction [3]. A Boolean formula ϕ is in $2CNF$ if each clause contains exactly two literals [14]. A Boolean formula ϕ in $2CNF$ is $MONOTONE$ if no clause in ϕ contains a negated variable [14]. Counting the number of satisfying truth assignments in a $MONOTONE$ $2CNF$ formula is a well-known *#P–complete* problem (denoted as *#MONOTONE 2SAT*) [19].

3. Results. In number theory, an integer q is called a quadratic residue modulo n if it is congruent to a perfect square modulo n [10]; i.e., if there exists an integer x such that:

$$x^2 \equiv q(mod\ n).$$

Otherwise, q is called a quadratic nonresidue modulo n [10]. When in the context is clear the terminology "quadratic residue" and "quadratic nonresidue", then it is dropped the adjective "quadratic" [10]. We use the shorthand notations $q\ R\ p$ and $q\ N\ p$, to indicated that q is a quadratic residue or nonresidue, respectively. [10].

THEOREM 3.1. *#MONOTONE 2SAT* $\in P$.

Proof. Let ϕ be a Boolean formula in $2CNF$ of n variables and m clauses. Let $p_1, \ldots, p_{2\times m}$ be the first $2 \times m$ odd primes such that they have 2 as a quadratic nonresidue. Then, we assign for each literal inside of every clause in the Boolean formula ϕ a unique of these prime numbers. We shall say a number z satisfies ϕ if the assignment $(z\ R\ (p_{1,a} \times p_{1,b} \times \ldots \times p_{1,s}), z\ R\ (p_{2,c} \times p_{2,d} \times \ldots \times p_{2,r}), \ldots, z\ R\ (p_{n,e} \times p_{n,f} \times \ldots \times p_{n,t}))$ satisfies ϕ such that each prime $p_{i,j}$ was assigned to the variable x_i which is contained in the clause c_j. This means in a satisfying truth assignment T the variable x_1 is true if $z\ R\ p_{1,j}$ for every prime $p_{1,j}$ assigned to the literal x_1 which is contained into a clause c_j or x_2 is false when $z\ N\ p_{2,j'}$ for some prime $p_{2,j'}$ assigned to the literal x_2 that is contained into the clause $c_{j'}$ and so forth. We can argument this condition by the following properties:

1. A number z is a nonresidue modulo y when z is a nonresidue modulo for at least one prime power dividing y [10].
2. A number z is a residue modulo y when z is a residue modulo for every prime power dividing y [10].

Now, for each clause c_k in ϕ we construct an expression of nonresidues that make the clause false for a possible candidate z. For example, in the clause $c_k = (x_r \vee x_t)$ for $1 \leq r, t \leq n$, then a solution of the simultaneous nonresidues $z\ N\ p_{r,k}$ and $z\ N\ p_{t,k}$ guarantee the clause will be false because x_r would be false and x_t would be false. However, we already know that when $z\ N\ p_{r,k}$ and $z\ N\ p_{t,k}$, then $(2 \times z)\ R\ p_{r,k}$ and $(2 \times z)\ R\ p_{t,k}$ because 2 is a nonresidue modulo every of these chosen primes and the multiplication of a nonresidue with a nonresidue is a residue [10]. In contraposition, the multiplication of a residue with a nonresidue is a nonresidue [10]. Since $p_{r,k}$ and $p_{t,k}$ are primes, then we can assure that $(2 \times z)\ R\ (p_{r,k} \times p_{t,k})$ due to the mentioned property (2). Therefore, when $(2 \times z)\ R\ (p_{r,k} \times p_{t,k})$, then we guarantee the clause c_k will be evaluated as false.

In this way, if we guarantee that for some number z we obtain $(2 \times z)\ N\ (p_{r,k} \times p_{t,k})$ for every clause $c_k = (x_r \vee x_t)$ in ϕ, then z will correspond to a satisfying truth assignment for ϕ. However, when $(2 \times z)\ N\ (p_{r,k} \times p_{t,k})$ for some clause $c_k = (x_r \vee x_t)$, then $(4 \times z)\ R\ (p_{r,k} \times p_{t,k})$ because 2 is a nonresidue modulo every of these chosen primes and the multiplication of a nonresidue with a nonresidue is a residue [10]. Consequently, if we guarantee that for some number z we obtain $(4 \times z)\ R\ (p_{r,k} \times p_{t,k})$ for every clause $c_k = (x_r \vee x_t)$ in ϕ, then z will correspond to a satisfying truth assignment for ϕ.

We can find all the values $q < (p_{r,k} \times p_{t,k})$ such that $z \equiv q(mod\ (p_{r,k} \times p_{t,k}))$ for every clause c_k where $q = (d^2\ mod\ (p_{r,k} \times p_{t,k}))$, $d < (p_{r,k} \times p_{t,k})$ and q is divisible by 4 [10]. The number n_k is equal to the amount of all these previous different values q for a clause c_k. If we combine all of these congruences into m simultaneous congruences such that we always pick exactly one arbitrary congruence in the group of q values for every clause c_k, then we can apply the Chinese Remainder Theorem to obtain a single and unique solution $z < p_1 \times p_2 \times \ldots \times p_{2 \times m}$ which will certainly correspond to a satisfying truth assignment in ϕ [16]. Therefore, the multiplication of $n_1 \times n_2 \times \ldots \times n_m$ (that is equal to the number of all possible combinations of m simultaneous congruences) will be equal to the amount of different satisfying truth assignments for ϕ.

Thus, $\#MONOTONE\ \ 2SAT \in P$. Certainly, we can find the first $2 \times m$ odd primes such that they have 2 as a quadratic nonresidue just checking for every odd prime p whether

$$p \equiv 3(mod\ 8)$$

or

$$p \equiv 5(mod\ 8)$$

as a consequence of the Euler's criterion [17]. Indeed, there are infinitely many primes of the form $8 \times k + 3$ or $8 \times k + 5$ [17]. Moreover, the n^{th} odd prime which has 2 as a quadratic nonresidue is polynomially bounded by $n \times \ln n$ [17]. In addition, we can make the primality test of a number in polynomial time [2]. ∎

THEOREM 3.2. *$P = NP$.*

Proof. *#MONOTONE 2SAT* is a well-known *#P–complete* problem [19]. If any single *#P–complete* problem can be solved in polynomial time, then $P = NP$ [14]. Therefore, as a consequence of Theorem 3.1, the answer of the P versus NP problem will be $P = NP$. ∎

4. Conclusion. No one has been able to find a polynomial time algorithm for any of more than 300 important known *NP–complete* problems [8]. Most complexity theorists already assume P is not equal to NP, but no one has found an accepted and valid proof yet [9]. There are several consequences if P is not equal to NP, such as many common problems cannot be solved efficiently [5]. However, a proof of $P = NP$ will have stunning practical consequences, because it leads to efficient methods for solving some of the important problems in NP [5]. The consequences, both positive and negative, arise since various *NP–complete* problems are fundamental in many fields [5]. This result explicitly concludes with the answer of the P versus NP problem: $P = NP$.

Cryptography, for example, relies on certain problems being difficult. A constructive and efficient solution to an *NP–complete* problem such as $3SAT$ will break most existing cryptosystems including: Public-key cryptography [12], symmetric ciphers [13] and one-way functions used in cryptographic hashing [7]. These would need to be modified or replaced by information-theoretically secure solutions not inherently based on *P–NP* equivalence.

There are enormous consequences that will follow from rendering tractable many currently mathematically intractable problems. For instance, many problems in operations research are *NP–complete*, such as some types of integer programming and the traveling salesman problem [11]. Efficient solutions to these problems have enormous implications for logistics [5]. Many other important problems, such as some problems in protein structure prediction, are also *NP–complete*, so this will spur considerable advances in biology [4].

But such changes may pale in significance compared to the revolution an efficient method for solving *NP–complete* problems will cause in mathematics itself. Stephen Cook says: "...it would transform mathematics by allowing a computer to find a formal proof of any theorem which has a proof of a reasonable length, since formal proofs can easily be recognized in polynomial time." [5].

Indeed, with this proof of $P = NP$ we could solve not merely one Millennium Problem but all seven of them [1]. This observation is based on once we fix a formal system such as the first-order logic plus the axioms of ZF set theory, then we can find a demonstration in time polynomial in n when a given statement has a proof with at most n symbols long in that system [1]. This is assuming that the other six Clay conjectures have ZF proofs that are not too large such as it was the Perelman's case [15].

Besides, a $P = NP$ proof reveals the existence of an interesting relationship between humans and machines [1]. For example, suppose we want to program a computer to create new Mozart-quality symphonies and Shakespeare-quality plays. When $P = NP$, this could be reduced to the easier problem of writing a computer program to recognize great works of art [1].

REFERENCES

[1] S. Aaronson, *$P \stackrel{?}{=} NP$*, Electronic Colloquium on Computational Complexity, Report No. 4, (2017).

[2] M. Agrawal, N. Kayal, and N. Saxena, *PRIMES is in P*, Annals of Mathematics, 160 (2004), pp. 781–793, https://doi.org/http://doi.org/10.4007/annals.2004.160.781.

[3] S. Arora and B. Barak, *Computational complexity: a modern approach*, Cambridge University Press, 2009.

[4] B. Berger and T. Leighton, *Protein folding in the hydrophobic-hydrophilic (HP) model is NP-complete*, Journal of Computational Biology, 5 (1998), pp. 27–40, https://doi.org/https://doi.org/10.1145/279069.279080.

[5] S. A. Cook, *The P versus NP Problem*, April 2000. At http://www.claymath.org/sites/default/files/pvsnp.pdf.

[6] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, The MIT Press, 3 ed., 2009.

[7] D. De, A. Kumarasubramanian, and R. Venkatesan, *Inversion Attacks on Secure Hash Functions Using sat Solvers*, in International Conference on Theory and Applications of Satisfiability Testing, Springer, 2007, pp. 377–382, https://doi.org/https://doi.org/10.1007/978-3-540-72788-0_36.

[8] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, San Francisco: W. H. Freeman and Company, 1 ed., 1979.

[9] W. I. Gasarch, *Guest column: The second $P \stackrel{?}{=} NP$ poll*, ACM SIGACT News, 43 (2012), pp. 53–77.

[10] C. F. Gauss and A. A. Clarke, *Disquisitiones Arithemeticae*, New York: Springer, 2 ed., 1986.

[11] O. Goldreich, *P, NP, and NP-Completeness: The basics of computational complexity*, Cambridge University Press, 2010.

[12] S. Horie and O. Watanabe, *Hard instance generation for SAT*, Algorithms and Computation, (1997), pp. 22–31, https://doi.org/https://doi.org/10.1007/3-540-63890-3_4.

[13] F. Massacci and L. Marraro, *Logical Cryptanalysis as a SAT Problem*, Journal of Automated Reasoning, 24 (2000), pp. 165–203, https://doi.org/https://doi.org/10.1023/A:1006326723002.

[14] C. H. Papadimitriou, *Computational complexity*, Addison-Wesley, 1994.

[15] G. Perelman, *The entropy formula for the Ricci flow and its geometric applications*, November 2002. At http://www.arxiv.org/abs/math.DG/0211159.

[16] K. H. Rosen, *Elementary Number Theory and its Applications*, Addison-Wesley, 3 ed., 1993.

[17] J. H. Silverman, *A Friendly Introduction to Number Theory*, Pearson Education, Inc., 4 ed., 2012.

[18] M. Sipser, *Introduction to the Theory of Computation*, vol. 2, Thomson Course Technology Boston, 2006.

[19] L. G. Valiant, *The complexity of enumeration and reliability problems*, SIAM Journal on Computing, (1979), pp. 410–421, https://doi.org/http://dx.doi.org/10.1137/0208032.