P versus 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? To attack the P = NP question the concept of NP-completeness is very useful. If any single NP-complete problem is in P, then P = NP. We prove there is a problem in NP-complete and P. Therefore, we demonstrate P = NP.

Keywords: Complexity classes · Completeness · Polynomial time · Boolean formula.

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 [5].

In 1936, Turing developed his theoretical computational model [13]. The deterministic and nondeterministic Turing machines have become in two of the most important definitions related to this theoretical model for computation [13]. A deterministic Turing machine has only one next action for each step defined in its program or transition function [13]. 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 [13]. 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 $TIME(f(n))$ [13]. In addition, the complexity class $NTIME(f(n))$ consists in those languages that can be decided within time f by nondeterministic Turing machines [13]. 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 [13]. At the same time,

NP consists in all languages in $NTIME(n^k)$ for every possible positive fixed constant k [13]. NP is also the complexity class of languages whose solutions may be verified in polynomial time [13]. 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 [9].

To attack the $P = NP$ question the concept of *NP–completeness* is very useful [1]. *NP–complete* problems are a set of problems to each of which any other NP problem can be reduced in polynomial time, and whose solution may still be verified in polynomial time [13]. That is, any NP problem can be transformed into any of the *NP–complete* problems [13]. If any single *NP–complete* problem can be solved in polynomial time, then every NP problem has a polynomial time algorithm [6]. In this work, we prove there is a problem in *NP–complete* and P. Thus, we demonstrate $P = NP$ [13]. There are stunning practical consequences when $P = NP$ [13]. 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:

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

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 [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 [13].

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 [16]. 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* [8]. 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* [6]. Moreover, if $L \in NP$, then $L \in$ *NP–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), $\neg$(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:

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

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 $\neg x_2$. 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. Many problems have been proved that belong to *NP-complete* by a polynomial time reduction from $3SAT$ [8]. For example, the problem *NAE 3SAT* defined as follows: Given a Boolean formula ϕ in $3CNF$, is there a truth assignment such that each clause in ϕ has at least one true literal and at least one false literal?

A logarithmic space Turing machine has a read-only input tape, a write-only output tape, and a read/write work tape [16]. The work tape may contain $O(\log n)$ symbols [16]. In computational complexity theory, *LOGSPACE* is the complexity class containing those decision problems that can be decided by a logarithmic space Turing machine which is deterministic [13]. $NLOGSPACE$ is the complexity class containing the decision problems that can be decided by a logarithmic space Turing machine which is nondeterministic [13]. 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 problem called $2SAT$, where we asked whether a given Boolean formula ϕ in $2CNF$ is satisfiable. $2SAT$ is complete for $NLOGSPACE$ [13]. 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 an *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 [12]. We denote the XOR function as $\oplus$. The *XOR 2SAT* problem will be equivalent to *XOR SAT*, but the clauses in the formula have exactly two distinct literals. *XOR 2SAT* is in *LOGSPACE* [2], [15].

3 Result

Definition 1. ***MAXIMUM EXCLUSIVE-OR 2-UNSATISFIABILITY***

INSTANCE: A positive integer K and a formula ϕ that is an instance of XOR 2SAT.

QUESTION: Is there a truth assignment in ϕ such that at most K clauses are unsatisfiable?

We denote this problem as $MAX \oplus 2UNSAT$.

Theorem 1. $MAX \oplus 2UNSAT \in$ *NP–complete.*

Proof. It is trivial to see $MAX \oplus 2UNSAT \in NP$ [13]. Given a Boolean formula ϕ in $3CNF$ with n variables and m clauses, we create the following formulas for each clause $c_i = (x \vee y \vee z)$ in ϕ, where x, y and z are literals,

$$P_i = (x \oplus y) \wedge (y \oplus z) \wedge (x \oplus z).$$

We can see P_i has at most one unsatisfiable clause if and only if at least one member of $\{x, y, z\}$ is true and at least one member of $\{x, y, z\}$ is false. Hence, we can create the Boolean formula ψ as the conjunction of the P_i formulas for every clause c_i in ϕ, such that $\psi = P_1 \wedge \ldots \wedge P_m$. Finally, we obtain that

$$\phi \in \textit{NAE 3SAT if and only if } (\psi, m) \in MAX \oplus 2UNSAT.$$

Consequently, we prove $NAE\ 3SAT \leq_p MAX \oplus 2UNSAT$ where $NAE\ 3SAT \in$ *NP–complete*. To sum up, we show $MAX \oplus 2UNSAT \in$ *NP–hard* and $MAX \oplus 2UNSAT \in NP$ and thus, $MAX \oplus 2UNSAT \in$ *NP–complete*.

Theorem 2. $MAX \oplus 2UNSAT \in P$.

This problem is solved by the algorithm $ALGO$ which receives as input an instance of $MAX \oplus 2UNSAT$. In this algorithm, we represent the Boolean formula ϕ as a set of clauses such that a clause $(x \oplus y)$ is equal to $(y \oplus x)$ where x and y are literals. The problem is solved by an inner procedure called $SOLUTION$. The algorithm $SOLUTION$ receives the Boolean formula ϕ and a set S of integers. The procedure $SOLUTION$ accepts if and only if there is a truth assignment where there are at most K' clauses which are unsatisfiable in ϕ and $K' \in S$. We reject in $SOLUTION$ when S is equal to the empty set $\emptyset$, because in that case there could be at most K' clauses which are unsatisfiable in ϕ but $K' \notin S$. On the other hand, we accept when the Boolean formula ϕ is empty, that is when $\phi = \emptyset$, because for every integer $K' \in S$ there is always at most K' clauses which are unsatisfiable in the empty formula. In case the number 0 is in S, then that will mean there could be at most 0 clauses which are unsatisfiable in ϕ. This case will be true if and only if $\phi \in$ *XOR 2SAT*. For that reason, we accept when $\phi \in$ *XOR 2SAT* else we remove this false case from S. This three main conditional statements can be done in polynomial time since *XOR 2SAT* $\in LOGSPACE$ and $LOGSPACE \subseteq P$ [13].

Next, we iterate from each pair of clauses $c_i, c_j \in \phi$ just checking whether $c_i = (x \oplus y)$ and $c_j = (x \oplus \neg y)$. In case of these clauses exists in ϕ, then for every truth assignment one of these clauses will be satisfiable and the other will be unsatisfiable in ϕ. In this way, we can remove them from ϕ and increment a variable num which indicates the number of obligatory unsatisfiable clauses for every truth assignment in the original ϕ (that is the formula which exists before removing the pair of clauses). After that, we subtract the number num from every integer $K' \in S$, because for every number $K' \in S$ there must be at most $K' - num$ clauses which are unsatisfiable in ϕ since there are num clauses that are obligatory unsatisfiable in the original ϕ. We add the new elements in a new set S'. In case of $K' \in S$ and $K' - num < 0$, then we will not consider this number $K' - num$ in S' since it cannot exist at a negative upper bound $K' - num$ of at most $K' - num$ clauses which are unsatisfiable in ϕ. This iteration can be done in polynomial time since we iterate quadratically from the clauses of ϕ and linear from the elements in S.

Finally, we iterate from each pair of clauses $c_i, c_j \in \phi$ just checking whether $(x \oplus y)$ and $c_j = (x \oplus z)$. In case of these clauses exists in ϕ, then for every truth assignment

- when the two clauses are unsatisfiable in ϕ then $(z \oplus \neg y)$ is satisfiable in ϕ,
- and when the two clauses are satisfiable in ϕ then $(z \oplus \neg y)$ is satisfiable in ϕ,
- and when one clause is unsatisfiable and the other satisfiable in ϕ then $(z \oplus \neg y)$ is unsatisfiable in ϕ.

Algorithm 1 ALGO's Polynomial Algorithm

Proof. **procedure** $ALGO(\phi, K)$ ▷ Appropriate input (ϕ, K) for $MAX \oplus 2UNSAT$
    return $SOLUTION(\phi, \{K\})$ ▷ Convert the second parameter to a set
end procedure
procedure $SOLUTION(\phi, S)$ ▷ A set ϕ of clauses and a set S of integers
    if $S = \emptyset$ **then** ▷ If the set is empty
        return "*no*" ▷ Reject
    else if $\phi = \emptyset$ **then** ▷ If ϕ is equal to the empty set
        return "*yes*" ▷ Accept
    else if $0 \in S$ **then** ▷ If S contains the number 0
        if $\phi \in XOR\ 2SAT$ **then** ▷ If ϕ is satisfiable
            return "*yes*" ▷ Accept
        else
            $S \leftarrow S - \{0\}$ ▷ Remove the number 0 from S
        end if
    end if
    $num \leftarrow 0$ ▷ Initialize num on 0
    for $c_i \in \phi$ **do** ▷ Iterate for each clause c_i in ϕ
        for $c_j \in \phi$ **do** ▷ Iterate for each clause c_j in ϕ
            if $c_i = (x \oplus y) \wedge c_j = (x \oplus \neg y)$ **then**
                $num \leftarrow num + 1$ ▷ Increment num by 1
                $\phi \leftarrow \phi - \{(x \oplus y), (x \oplus \neg y)\}$ ▷ Remove the clauses from ϕ
            end if
        end for
    end for
    $S' \leftarrow \emptyset$ ▷ Initialize S' to the empty set
    for $i \in S$ **do** ▷ Iterate for each integer i in S
        if $(i - num) \geq 0$ **then**
            $S' \leftarrow S' \cup \{(i - num)\}$ ▷ Add the number $(i - num)$ to S'
        end if
    end for
    for $i \in S'$ **do** ▷ Iterate for each integer i in S'
        if $(i - 2) \geq 0$ **then**
            $S' \leftarrow S' \cup \{(i - 2)\}$ ▷ Add the number $(i - 2)$ to S'
        end if
    end for
    for $c_i \in \phi$ **do** ▷ Iterate for each clause c_i in ϕ
        for $c_j \in \phi$ **do** ▷ Iterate for each clause c_j in ϕ
            if $c_i = (x \oplus y) \wedge c_j = (x \oplus z)$ **then**
                $\phi \leftarrow \phi - \{(x \oplus y), (x \oplus z)\}$ ▷ Remove the clauses from ϕ
                $\phi \leftarrow \phi \cup \{(z \oplus \neg y)\}$ ▷ Add a new clause into ϕ
                return $SOLUTION(\phi, S')$ ▷ Recursively
            end if
        end for
    end for
    if $S' = \emptyset$ **then** ▷ If the set S' is empty
        return "*no*" ▷ Reject
    else
        return "*yes*" ▷ Otherwise accept
    end if
end procedure

In the new formula ϕ after removing the two clauses and adding the new one, we can consider for each integer $K' \in S'$ only the two cases $K-2$ (which is when the two clauses are unsatisfiable in ϕ) and K (for the other cases). Since the number K' is already in the set, then we will only need to add $K'-2$ to S'. In case of $K'-2$ is negative, then we ignore it since it cannot exist at a negative upper bound $K'-2$ of at most $K'-2$ clauses which are unsatisfiable in ϕ. Hence, we call recursively to the procedure $SOLUTION$ with the new Boolean formula ϕ and the set S'. In the final step, when there is no a pair of clauses $c_i, c_j \in \phi$ which contain the same literal, then we can accept if $S' \neq \emptyset$ because all the clauses in ϕ could be arbitrarily unsatisfiable or satisfiable and therefore, we can guarantee there is a truth assignment where there are at most K' clauses which are unsatisfiable in ϕ and $K' \in S'$. We also reject in $SOLUTION$ when S' is equal to the empty set $\emptyset$, because in that case there could be at most K' clauses which are unsatisfiable in ϕ but $K' \notin S'$. This last iteration can be done in polynomial time since we iterate quadratically from the clauses of ϕ and linear from the elements in S'. At the end, we solve $MAX \oplus 2UNSAT$ in polynomial time and thus, $MAX \oplus 2UNSAT \in P$.

Lemma 1. *$P = NP$.*

Proof. If any single *NP–complete* problem can be solved in polynomial time, then every NP problem has a polynomial time algorithm [6]. Hence, this is a direct consequence of Theorems 1 and 2.

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]. 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 [10], symmetric ciphers [11] 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 positive 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 [8]. 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, this proof of $P = NP$ 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 [14].

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. Aaronson, S.: P $\stackrel{?}{=}$ NP. Electronic Colloquium on Computational Complexity, Report No. 4 (2017)
2. Alvarez, C., Greenlaw, R.: A compendium of problems complete for symmetric logarithmic space. Computational Complexity **9**(2), 123–145 (2000)
3. Arora, S., Barak, B.: Computational complexity: a modern approach. Cambridge University Press (2009)
4. Berger, B., Leighton, T.: Protein folding in the hydrophobic-hydrophilic (HP) model is NP-complete. Journal of Computational Biology **5**(1), 27–40 (1998)
5. Cook, S.A.: The P versus NP Problem (April 2000), available at `http://www.claymath.org/sites/default/files/pvsnp.pdf`
6. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms. The MIT Press, 3rd edn. (2009)
7. De, D., Kumarasubramanian, A., Venkatesan, R.: Inversion attacks on secure hash functions using SAT solvers. In: International Conference on Theory and Applications of Satisfiability Testing. pp. 377–382. Springer (2007)
8. 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)
9. Gasarch, W.I.: Guest column: The second P $\stackrel{?}{=}$ NP poll. ACM SIGACT News **43**(2), 53–77 (2012)
10. Horie, S., Watanabe, O.: Hard instance generation for SAT. Algorithms and Computation pp. 22–31 (1997)
11. Massacci, F., Marraro, L.: Logical cryptanalysis as a SAT problem. Journal of Automated Reasoning **24**(1), 165–203 (2000)
12. Moore, C., Mertens, S.: The Nature of Computation. Oxford University Press (2011)
13. Papadimitriou, C.H.: Computational complexity. Addison-Wesley (1994)
14. Perelman, G.: The entropy formula for the Ricci flow and its geometric applications (November 2002), available at `http://www.arxiv.org/abs/math.DG/0211159`

15. Reingold, O.: Undirected connectivity in log-space. Journal of the ACM **55**(4), 1–24 (2008)
16. Sipser, M.: Introduction to the Theory of Computation, vol. 2. Thomson Course Technology Boston (2006)