The P versus NP Problem

Frank Vega[1 [0000−0001−8210−4126]]

Joysonic,
Uzun Mirkova 5,
Belgrade, 11000, 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? 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 problem have failed. To attack the P versus NP problem, the NP-completeness is a useful tool. We prove the known NP-complete problem MONOTONE 1-IN-3 3SAT can be polynomially reduced to the polynomial language 2SET PACKING. In this way, MONOTONE 1-IN-3 3SAT must be in P. If any NP-complete problem can be solved in polynomial time, then every language in NP has a polynomial time algorithm. Hence, we demonstrate the complexity class P is equal to NP.

Keywords: Complexity Classes · Completeness · Polynomial Time · MONOTONE 1-IN-3 3SAT · 2SET PACKING.

1 Introduction

The P versus NP problem is a major unsolved problem in computer science [4]. This is considered by many to be the most important open problem in the field [4]. It is one of the seven Millennium Prize Problems selected by the Clay Mathematics Institute [4]. 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 [4].

In 1936, Turing developed his theoretical computational model [15]. The deterministic and nondeterministic Turing machines have become in two of the most important definitions related to this theoretical model for computation [15]. A deterministic Turing machine has only one next action for each step defined in its program or transition function [15]. 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 [15].

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 [5]. A complexity class is a set of problems, which

are represented as a language, grouped by measures such as the running time, memory, etc [5].

In the computational complexity theory, the class P contains those languages that can be decided in polynomial time by a deterministic Turing machine [11]. The class NP consists in those languages that can be decided in polynomial time by a nondeterministic Turing machine [11]. 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 [10].

It is fully expected that $P \neq NP$ [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]. Whether $P = NP$ is still a controversial possible solution to this problem [1]. However, we prove the complexity class P is equal to NP. Hence, we solve one of the most important open problems in computer science with a solution which was certainly unexpected and with stunning practical consequences [1].

2 Definitions

Let Σ be a finite alphabet with at least two elements, and let Σ^* be the set of finite strings over Σ [2]. A Turing machine M has an associated input alphabet Σ [2]. For each string w in Σ^* there is a computation associated with M on input w [2]. We say that M accepts w if this computation terminates in the accepting state, that is $M(w) =$ "*yes*" [2]. Note that M fails to accept w either if this computation ends in the rejecting state, that is $M(w) =$ "*no*", or if the computation fails to terminate [2].

The language accepted by a Turing machine M, denoted $L(M)$, has an associated alphabet Σ and is defined by

$$L(M) = \{w \in \Sigma^* : M(w) = \text{“}yes\text{”}\}.$$

We denote by $t_M(w)$ the number of steps in the computation of M on input w [2]. For $n \in \mathbb{N}$ we denote by $T_M(n)$ the worst case run time of M; that is

$$T_M(n) = max\{t_M(w) : w \in \Sigma^n\}$$

where Σ^n is the set of all strings over Σ of length n [2]. We say that M runs in polynomial time if there is a constant k such that for all n, $T_M(n) \leq n^k + k$ [2]. In other words, this means the language $L(M)$ can be accepted by the Turing machine M in polynomial time. Therefore, P is the complexity class of languages that can be accepted in polynomial time by deterministic Turing machines [5]. A verifier for a language L is a deterministic Turing machine M, where

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

We measure the time of a verifier only in terms of the length of w, so a polynomial time verifier runs in polynomial time in the length of w [2]. A verifier uses additional information, represented by the symbol c, to verify that a string w is a member of L. This information is called certificate. NP is also the complexity class of languages defined by polynomial time verifiers [14].

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 [15]. 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–complete*, then L is *NP–hard* [11]. Moreover, if $L \in NP$, then $L \in$ *NP–complete* [11]. A Boolean formula ϕ 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. 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 [5]. 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 [5]. A Boolean formula is in 3-conjunctive normal form or $3CNF$, if each clause has exactly three distinct literals [5]. 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 $\neg\, x_2$.

3 Results

Definition 1. ***MONOTONE 1–IN–3 3SAT***

INSTANCE: A Boolean formula ϕ in $3CNF$ such that there is no clause which contains a negated literal.

QUESTION: Is there a truth assignment for ϕ such that each clause in ϕ has exactly one true literal?

REMARKS: MONOTONE 1–IN–3 3SAT is in NP–complete [9].

Definition 2. ***2SET PACKING***

INSTANCE: A collection C of finite sets and a positive integer $K \leq |C|$ such that for all $c \in C$ we have $|c| \leq 2$ where $|\ldots|$ is the cardinality function.

QUESTION: Does C contain at least K mutually disjoint sets?

REMARKS: 2SET PACKING is solvable in polynomial time by matching techniques [9].

Theorem 1. *MONOTONE 1–IN–3 3SAT $\leq_p$ 2SET PACKING.*

Proof. Consider a Boolean formula ϕ in $3CNF$ with m clauses such that there is no clause which contains a negated literal in which the positive literal x appears k times. We replace the first occurrence of x by x_1, the second by x_2, and so on, where $x_1, x_2, \ldots, x_k$ are k new variables. Later, we add

$$(\neg\, x_1 \vee x_2) \wedge (\neg\, x_2 \vee x_3) \wedge \ldots \wedge (\neg\, x_k \vee x_1)$$

to a new Boolean formula ϕ' in CNF which contains the modified clauses of ϕ plus these above clauses of fewer than 3 literals where we do this procedure for each positive literal x in ϕ. Note, this is logically equivalent to

$$x_1 \Rightarrow x_2 \Rightarrow \ldots \Rightarrow x_k \Rightarrow x_1$$

such that the resulting expression in ϕ' satisfies the condition for a truth assignment on x. In this new formula ϕ' every variable appears at most 3 times. If $k = 1$, then the literal x_1 appears once. If $k > 1$, for every integer i between 1 and k, we have that the literal x_i appears twice and $\neg\, x_i$ appears once. Suppose we have the following instance ϕ of *MONOTONE 1–IN–3 3SAT*

$$\ldots (x \vee w \vee g) \wedge \ldots \wedge (x \vee y \vee z) \ldots$$

then the transformed expression ϕ' in CNF over the variable x is

$$\ldots (x_1 \vee w \vee g) \wedge \ldots \wedge (x_2 \vee y \vee z) \ldots (\neg\, x_1 \vee x_2) \wedge (\neg\, x_2 \vee x_1)$$

where

- the variable x_1 appears thrice and,
- the literal x_1 appears twice and,
- the literal $\neg\, x_1$ appears once.

Now from the expression ϕ' in CNF, we create an instance of *2SET PACKING* in the following two steps:

1. In the first step, we create a set for each literal that appears in every clause. For every variable x_i, we assign the set $\{x_{1,i}, x_{2,i}\}$ to the negated literal $\neg x_i$, for the positive literal x_i in the clause of three literals the set $\{x_{1,i}\}$ and for the positive literal x_i in the clause of two literals the set $\{x_{2,i}\}$.
2. In the second step, for each clause c_i of three literals we add to the set assigned to its literals a single element d_i which is unique for every clause c_i of three literals obtaining three sets of two elements. Moreover, for each clause c_j of two literals, there is a negated literal $\neg x_i$ with a set $\{x_{1,i}, x_{2,i}\}$ and another positive literal $x_j$with a set of a single element $\{x_{2,j}\}$, therefore we add to the set of cardinality equal to one the another element $x_{1,i}$ obtaining the set $\{x_{2,j}, x_{1,i}\}$ of two elements.

If we consider an instance of *2SET PACKING* created from the sets of each literal from every clause of ϕ', then the first step guarantee that we should analyze separately the sets from the positive literals in relation to the set of the single negated literal. In this way, in the first step there is no a possibility where we could choose the three sets for a single variable in ϕ' at the same time because they are not mutually disjoint sets. For that reason, we can assure if we choose some K mutually disjoint sets from this instance, then we can assign as true the literals which are represented by these sets, since there is no a violation when for some variable x_i the literals x_i and $\neg x_i$ could be both true or false at the same time.

In the second step, we guarantee this truth assignment created from a K mutually disjoint sets of this instance complies that every clause of three literals in ϕ' has exactly one true literal since the sets from the literals of a clause of three literals are not mutually disjoint since they contain the single element d_i which is unique for every clause c_i of three literals. Moreover, this second step makes possible that exactly one literal for a clause of two literals in ϕ' can be true just making possible the condition of assignment for every variable x in the original ϕ. Note, that we add to the set of the positive literal of clauses of two literals in ϕ', the element $x_{1,i}$ and not $x_{2,i}$ from the negated literal $\neg x_i$, because that guarantee there should not be a violation against the same truth assignment between the variables x_j and x_i that represents the variable x over two distinct clauses of three literals that contain x_j and x_i in ϕ' respectively.

Look at the example of the Figure 1 and see how is applied the first step and finally the second step which is represented in the Figure 2 over this reduction. We have the clause $c_i = (x_1 \vee y_2 \vee z_3)$ of three literals and the clauses $c_j = (\neg x_1 \vee x_2)$ and $c_r = (x_1 \vee \neg x_k)$ of two literals. Note, the variable x_1 only appears in these three clauses. According to the first step in the Figure 1, we add the set $\{x_{1,1}, x_{2,1}\}$ to the negated variable $\neg x_1$ and the another sets $\{x_{1,1}\}$ and $\{x_{2,1}\}$ to the positive literals in the clause c_i of three literals and the clause c_r of two literals respectively. We also see how is applied the second step in the Figure 2 where it is added the element d_i to all the sets of the literals in the

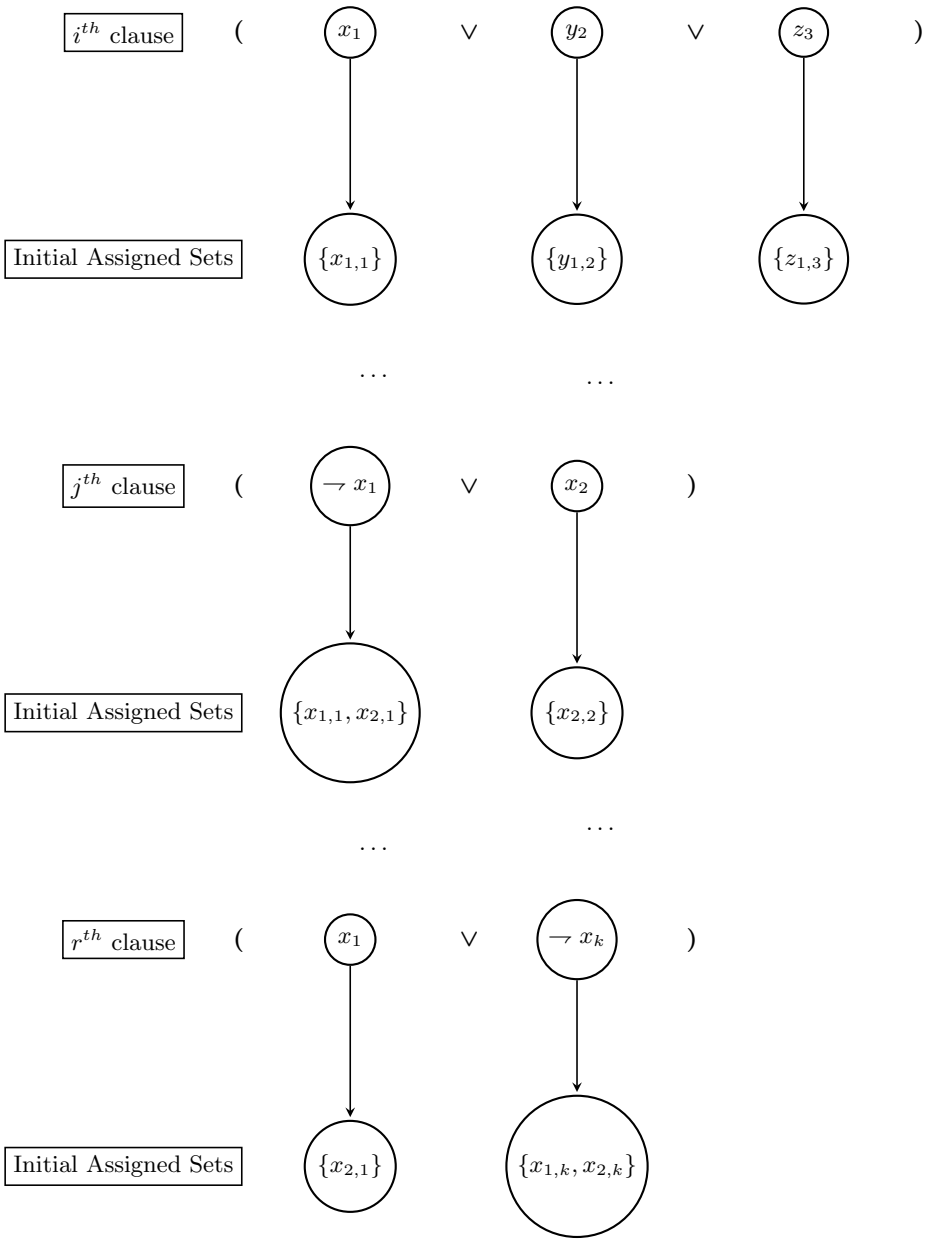

Fig. 1. First step

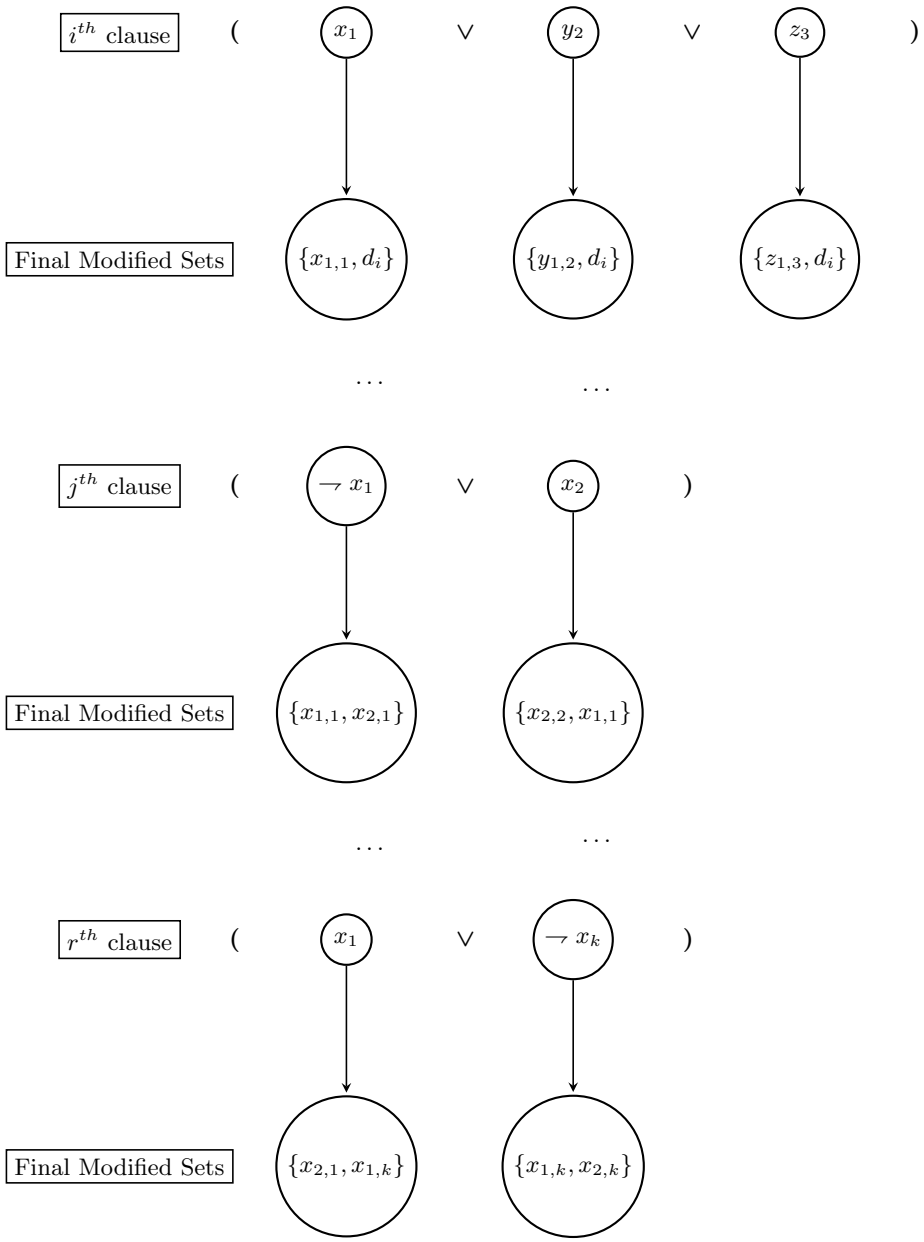

Fig. 2. Second step

clause c_i of three literals. The transformation in the clauses of two literals in the second step is also visible in the Figure 2 where it is added the element $x_{1,1}$ to the set $\{x_{2,2}\}$ assigned to the positive literal x_2 in the clause c_j of two literals. With these two steps, we obtain the sets of each literal from every clause have a cardinality equal to 2.

Now, if the Boolean formula ϕ' has m' clauses, then the collection C of sets for each literal from every clause complies that

$$\phi \in \textit{MONOTONE 1–IN–3 3SAT if and only if } (C, m') \in \textit{2SET PACKING}.$$

Is it the case that C contains m' mutually disjoint sets and ϕ is not a Boolean formula in *MONOTONE 1–IN–3 3SAT*? The answer is no. Think for example in some m' mutually disjoint sets, then we can transform them in a truth assignment T evaluating every literal assigned in these sets as true. In this truth assignment T from the clauses of two literals exactly one literal would be true and from the clauses of three literals exactly one literal would be true in the Boolean formula ϕ'. Certainly, we guarantee that since the literals of every clause have not mutually disjoint sets in ϕ' after the reduction. In this way, the truth assignment T will be a satisfying assignment for the Boolean formula ϕ' where every clause has exactly one true literal which also implies that the original Boolean formula ϕ will be in *MONOTONE 1–IN–3 3SAT*. Take into account that we assume that $K = m'$ since at least the m' clauses which contains ϕ' should be satisfiable with this truth assignment T in order to satisfy the original Boolean formula ϕ as well.

This is a polynomial time reduction:

- We can create the Boolean formula ϕ' in time $O(m^2)$ just replacing the modified clauses of three literals and adding the clauses of two literals.
- After that, we create the instance for *2SET PACKING* in time $O(m)$ just creating the new sets in the first step and modifying them in the second step.

Certainly, we can create the instance (C, m') just running the whole reduction in time $O(m^2)$ and thus, the proof is completed.

Theorem 2. $P = NP$.

Proof. The known *NP–complete* problem *MONOTONE 1–IN–3 3SAT* can be reduced in polynomial time to *2SET PACKING* where *2SET PACKING* $\in P$ [9]. Consequently, *MONOTONE 1–IN–3 3SAT* $\in P$. If any *NP–complete* problem can be solved in polynomial time, then every language in NP has a polynomial time algorithm [5]. In conclusion, we finally prove that $P = NP$.

4 Conclusions

No one has been able to find a polynomial time algorithm for any of more than 300 important known *NP–complete* problems [9]. 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 [4]. The consequences, both positive and negative, arise since various *NP–complete* problems are fundamental in many fields [4]. This result explicitly concludes supporting the existence of a practical solution for the *NP–complete* problems.

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 [6]. These would need to be modified or replaced by information-theoretically secure solutions not inherently based on *P–NP* equivalence.

Learning becomes easy by using the principle of Occam's razor-we simply find the smallest program consistent with the data [7]. Near perfect vision recognition, language comprehension and translation and all other learning tasks become trivial [7]. We will also have much better predictions of weather and earthquakes and other natural phenomenon [7].

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 [9]. Efficient solutions to these problems have enormous implications for logistics [4]. Many other important problems, such as some problems in protein structure prediction, are also *NP–complete*, so this will spur considerable advances in biology [3]. Furthermore, this could be a huge advance in medicine, since the cancer might be cured by this solution [8].

But such changes may pale in significance compared to the revolution an efficient method for solving *NP–complete* problems will cause in mathematics itself. Research mathematicians spend their careers trying to prove theorems, and some proofs have taken decades or even centuries to find after problems have been stated. For instance, Fermat's Last Theorem took over three centuries to prove. A method that is guaranteed to find proofs to theorems, should one exist of a "reasonable" size, would essentially end this struggle.

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 [1].

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. Arora, S., Barak, B.: Computational complexity: a modern approach. Cambridge University Press (2009)
3. Berger, B., Leighton, T.: Protein folding in the hydrophobic-hydrophilic (HP) model is NP-complete. Journal of Computational Biology **5**(1), 27–40 (1998). https://doi.org/10.1145/279069.279080
4. Cook, S.A.: The P versus NP Problem. Clay Mathematics Institute (April 2000), at http://www.claymath.org/sites/default/files/pvsnp.pdf
5. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms. The MIT Press, 3rd edn. (2009)
6. 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)
7. Fortnow, L.: The Status of the P Versus NP Problem. Commun. ACM **52**(9), 78–86 (Sep 2009). https://doi.org/10.1145/1562164.1562186
8. Fortnow, L.: P = NP and Cancer. Computational Complexity Blog (September 2018), at https://blog.computationalcomplexity.org/2018/09/p-np-and-cancer.html
9. 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)
10. Gasarch, W.I.: Guest column: The second P $\stackrel{?}{=}$ NP poll. ACM SIGACT News **43**(2), 53–77 (2012)
11. Goldreich, O.: P, NP, and NP-Completeness: The basics of computational complexity. Cambridge University Press (2010)
12. Horie, S., Watanabe, O.: Hard instance generation for SAT. Algorithms and Computation pp. 22–31 (1997). https://doi.org/10.1007/3-540-63890-3_4
13. Massacci, F., Marraro, L.: Logical cryptanalysis as a SAT problem. Journal of Automated Reasoning **24**(1), 165–203 (2000). https://doi.org/10.1023/A:1006326723002
14. Papadimitriou, C.H.: Computational complexity. Addison-Wesley (1994)
15. Sipser, M.: Introduction to the Theory of Computation, vol. 2. Thomson Course Technology Boston (2006)