EasyChair Preprint

№ 11063

On Feasibly Solving NP-Complete Problems

Frank Vega

October 23, 2023

On Feasibly Solving NP-complete Problems

Frank Vega
GROUPS PLUS TOURS INC., 9611 Fontainebleau Blvd, Miami, FL, 33172, US

Abstract

ONE–IN–THREE 3SAT consists in knowing whether a Boolean formula ϕ in $3CNF$ has a truth assignment such that each clause contains exactly one true literal or exactly two true literals. *ONE–IN–THREE 3SAT* remains *NP–complete* when all clauses are monotone. We create a polynomial time reduction which converts the monotone version into a bounded number of linear constraints on real numbers. Since the linear optimization on real numbers can be solved in polynomial time, then we can decide this *NP–complete* problem in polynomial time. Certainly, the problem of solving linear constraints on real numbers is equivalent to solve the particular case when there is a linear optimization without any objective to maximize or minimize. If any *NP–complete* can be solved in polynomial time, then we obtain that $P = NP$. Moreover, our polynomial reduction is feasible since it can be done in linear time.

2012 ACM Subject Classification Theory of computation → Complexity classes; Theory of computation → Problems, reductions and completeness

Keywords and phrases complexity classes, boolean formula, completeness, polynomial time

1 Introduction

In 1936, Turing developed his theoretical computational model [9]. The deterministic and nondeterministic Turing machines have become in two of the most important definitions related to this theoretical model for computation [9]. A deterministic Turing machine has only one next action for each step defined in its program or transition function [9]. 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 [9].

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 $M(w) =$ "yes" [1]. 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, or the computation ends in the halting state with some output, that is $M(w) = y$ (when M outputs the string y on the input w) [1].

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 [4]. A complexity class is a set of problems, which are represented as a language, grouped by measures such as the running time, memory, etc [4]. The language accepted by a Turing machine M, denoted $L(M)$, has an associated alphabet Σ and is defined by:

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

Moreover, $L(M)$ is decided by M, when $w \notin L(M)$ if and only if $M(w) =$ "no" [4]. 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 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 [1]. We say that M runs in polynomial time if there is a constant k such that for all n, $T_M(n) \leq n^k + k$ [1]. In other words, this

means the language $L(M)$ can be decided by the Turing machine M in polynomial time. Therefore, P is the complexity class of languages that can be decided by deterministic Turing machines in polynomial time [4]. A verifier for a language L_1 is a deterministic Turing machine M, where:

$$L_1 = \{w : M(w, u) = \text{“}yes\text{”} \textit{ for some string } u\}.$$

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 [1]. A verifier uses additional information, represented by the string u, to verify that a string w is a member of L_1. This information is called certificate. NP is the complexity class of languages defined by polynomial time verifiers [8].

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* [7]. If L_1 is a language such that $L' \leq_p L_1$ for some $L' \in$ *NP–complete*, then L_1 is *NP–hard* [4]. Moreover, if $L_1 \in NP$, then $L_1 \in$ *NP–complete* [4]. A principal *NP–complete* problem is SAT [7]. 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 Boolean formula with a satisfying truth assignment is satisfiable. The problem SAT asks whether a given Boolean formula is satisfiable [7]. 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 [4]. 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 [4]. A Boolean formula is in 3-conjunctive normal form or $3CNF$, if each clause has exactly three distinct literals [4]. 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 $\neg x_2$. Using these initial definitions as background, then we may be able to proceed with our main results.

2 Issues and Motivation

We show there is an *NP–complete* problem that can be solved in polynomial time. We can feasibly solve SAT using our algorithm. The whole reduction algorithm runs in polynomial time since we can reduce SAT to *ONE–IN–THREE 3SAT* in a feasible way: This is a trivial and well-known polynomial time reduction. We could transform the output of this reduction

into a linear optimization problem which has only constraints without any objective to maximize or minimize. The whole algorithm is based on the problem of linear optimization which is feasible when we do not restrict the variables to be solely integers [2].

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? It was essentially mentioned in 1955 from a letter written by John Nash to the United States National Security Agency. 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 problem have failed. A polynomial time algorithm for some *NP–complete* problem would imply that $P = NP$: This is the goal of this manuscript.

3 Summary of the Main Results

In computational complexity, one-in-three 3-satisfiability (*ONE–IN–THREE 3SAT*) is an *NP–complete* variant of SAT over $3CNF$ Boolean formulas. *ONE–IN–THREE 3SAT* consists in knowing whether a Boolean formula ϕ in $3CNF$ has a truth assignment such that each clause contains exactly one true literal or exactly two true literals [7]. *ONE–IN–THREE 3SAT* remains *NP–complete* when all clauses are monotone (meaning that variables are never negated) [7]. We define another another problem that we know it can be solved in polynomial time.

▶ **Definition 1.** ***Linear Constraints on Real Numbers (LCRN)***

INSTANCE: A set of linear equations and inequalities.

QUESTION: Is there a simultaneously satisfiability to all these constraints on real solutions?

REMARKS: $LCRN \in P$ [2].

We state our principal result.

▶ **Theorem 2.** *$LCRN \in NP$–complete.*

After this theorem we can assure the following result:

▶ **Theorem 3.** $P = NP$.

Proof. This is a direct proof of Theorem 2. ◀

4 Main Results

4.1 Proof of Theorem 2

Proof. Let's take a Boolean formula ϕ in $3CNF$ with n variables and m clauses when all clauses are monotone. For each variable b in the original formula we introduce the inequality

$$x_b \geq 0.0$$
$$x_b \neq \frac{1.0}{3.0}$$

where the real variable x_b is one-to-one linked to the Boolean variable b. Now, we iterate for each clause $c_i = (a \vee b \vee c)$ and create the linear equation and inequalities

$$x_a + x_b + x_c = 1.0$$
$$x_a + x_b \geq \frac{2.0}{3.0}$$
$$x_a + x_c \geq \frac{2.0}{3.0}$$

according to the real variables x_a, x_b, x_c one-to-one linked to the Boolean variables from the clause c_i in ϕ. Note that, the clause c_i contains exactly one true variable or exactly two true literals if and only if the equation and inequalities

$$x_a + x_b + x_c = 1.0$$
$$x_a + x_b \geq \frac{2.0}{3.0}$$
$$x_a + x_c \geq \frac{2.0}{3.0}$$

can be evaluated into real solutions that contains exactly one variable assignment lesser than $\frac{1}{3}$ or exactly one variable assignment greater than $\frac{1}{3}$. Note that, a solution for the inequalities

$$x_a + x_b \geq \frac{2.0}{3.0}$$
$$x_a + x_c \geq \frac{2.0}{3.0}$$

guarantee that there exist at least one or two variables greater than $\frac{1}{3}$ while a solution

$$x_a + x_b + x_c = 1.0$$

guarantee that not all the variables are greater than $\frac{1}{3}$ and not all the variables are lesser than $\frac{1}{3}$. Certainly, we define that a variable b is true for an assignment in the Boolean formula ϕ if and only if the solution for x_b is lesser than $\frac{1}{3}$. Finally, we create a polynomial time bounded number of equations and inequalities from the variables and clauses in the formula ϕ. In this way, we make a polynomial time reduction from ϕ in *ONE–IN–THREE 3SAT* on monotone clauses to a polynomially bounded instance of $LCRN$. Moreover, this already explained reduction can be done iterating over the variables and clauses of ϕ in linear time. Finally, we can see that $LCRN$ is trivially in NP, since we could check a whole solution for every single constraint in polynomial time. Therefore, the proof is done. ◀

5 Algorithm Implementation

We implement the polynomial time reduction using the programming language Python [10]. Moreover, we use the Microsoft Library Z3 for solving the linear optimization in polynomial time [5]. Z3 is a theorem prover from Microsoft Research with support for bitvectors, booleans, arrays, floating point numbers, strings, and other data types [5]. The whole project was developed by the author and it is available in GitHub on MIT License [11].

6 Explanation of their Significance

No one has been able to find a polynomial time algorithm for any of more than 300 important known *NP–complete* problems [7]. A proof of $P = NP$ will have stunning

practical consequences, because it possibly leads to efficient methods for solving some of the important problems in computer science [3]. The consequences, both positive and negative, arise since various *NP–complete* problems are fundamental in many fields [6].

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

But such changes may pale in significance compared to the revolution an efficient method for solving *NP–complete* problems will cause in mathematics itself [3]. Research mathematicians spend their careers trying to prove theorems, and some proofs have taken decades or even centuries to be discovered after problems have been stated [3]. For instance, Fermat's Last Theorem took over three centuries to be proved [3]. A method that guarantees to find proofs for theorems, should one exist of a "reasonable" size, would essentially end this struggle [3].

Acknowledgments

The author wishes to thank the mathematician Arthur Rubin for his constructive comments.

References

1 Sanjeev Arora and Boaz Barak. *Computational complexity: a modern approach*. Cambridge University Press, USA, 2009.

2 Dimitris Bertsimas and John N Tsitsiklis. *Introduction to linear optimization*, volume 6. Athena scientific Belmont, MA, 1997.

3 Stephen Arthur Cook. The P versus NP Problem. `https://www.claymath.org/wp-content/uploads/2022/06/pvsnp.pdf`, June 2022. Clay Mathematics Institute. Accessed 9 September 2023.

4 Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, 3rd edition, 2009.

5 Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

6 Lance Fortnow. The status of the P versus NP problem. *Communications of the ACM*, 52(9):78–86, 2009. `doi:10.1145/1562164.1562186`.

7 Michael R Garey and David S Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. San Francisco: W. H. Freeman and Company, 1 edition, 1979.

8 Christos Harilaos Papadimitriou. *Computational complexity*. Addison-Wesley, USA, 1994.

9 Michael Sipser. *Introduction to the Theory of Computation*, volume 2. Thomson Course Technology Boston, USA, 2006.

10 Guido VanRossum and Fred L Drake. *The Python Language Reference*, volume 561. Python Software Foundation Amsterdam, Netherlands, 2010.

11 Frank Vega. DEVOO PY (Github repository). `https://github.com/frankvegadelgado/devoo_py`, October 2023. Commit (SHA-1): 8f1262624d98e185e517afa917664d6eca695197.