Solution of P versus NP Problem
Frank Vega

To cite this version:
Frank Vega. Solution of P versus NP Problem. 2015. <hal-01161668>

HAL Id: hal-01161668
https://hal.archives-ouvertes.fr/hal-01161668
Submitted on 8 Jun 2015

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
Solution of P versus NP Problem

Frank Vega

June 7, 2015

Abstract: The P versus NP problem is one of the most important and unsolved problems in computer science. This consists in 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 in a seminal paper. We consider a new complexity class, called equivalent-P, which has a close relation with this problem. The class equivalent-P has those languages that contain ordered pairs of instances, where each one belongs to a specific problem in P, such that the two instances share a same solution, that is, the same certificate. We demonstrate that equivalent-P = NP and equivalent-P = P. In this way, we find the solution of P versus NP problem, that is, P = NP.

1 Introduction

The P versus NP problem 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 [4]. 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.

The argument made by Alan Turing in the twentieth century states that for any algorithm we can create an equivalent Turing machine [10]. There are some definitions related with this model such as the deterministic or nondeterministic Turing machine. A deterministic Turing machine has only one next action for each step defined in its program or transition function [9]. A nondeterministic Turing

ACM Classification: F.1.3
AMS Classification: 68-XX, 68Qxx, 68Q15
Key words and phrases: P, NP, NP-complete, P-complete, logarithmic-space, verifier
A machine can contain more than one action defined for each step of the program, where this program is not a function, but a relation [9].

Another huge advance in the last century was the definition of a complexity class. A language \( L \) 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 computational complexity theory, the class \( P \) consists in all those decision problems (defined as languages) that can be decided on a deterministic Turing machine in an amount of time that is polynomial in the size of the input; the class \( NP \) consists in all those decision problems whose positive solutions can be verified in polynomial-time given the right information, or equivalently, that can be decided on a nondeterministic Turing machine in polynomial-time [7].

The biggest open question in theoretical computer science concerns the relationship between those two classes:

Is \( P \) equal to \( NP \)?

In a 2002 poll of 100 researchers, 61 believed the answer to be no, 9 believed the answer is yes, and 22 were unsure; 8 believed the question may be independent of the currently accepted axioms and so impossible to prove or disprove [6].

There is an important complexity class called \( NP-complete \) [7]. The \( NP-complete \) problems are a set of problems to which any other \( NP \) problem can be reduced in polynomial-time, but whose solution may still be verified in polynomial-time [7]. In addition, there is another important complexity class called \( P-complete \) [9]. The \( P-complete \) problems are a set of problems to which any other \( P \) problem can be reduced in logarithmic-space, but they still remain in \( P \) [9]. We shall define a new complexity class that we called equivalent-\( P \) (see the Abstract) and denoted as \( \sim P \). We shall show that there is an \( NP-complete \) problem in \( \sim P \) and a \( P-complete \) problem in \( \sim P \). Moreover, we shall prove the complexity class \( \sim P \) is closed under reductions. Since \( P \) and \( NP \) are also closed under reductions, then we can conclude that \( P = NP \).

2 Theoretical framework

2.1 NP-complete class

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 exists a polynomial-time computable function \( f : \{0, 1\}^* \rightarrow \{0, 1\}^* \) such that for all \( x \in \{0, 1\}^* \),

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

(2.1)

There is an important complexity class called \( NP-complete \) [7]. A language \( L \subseteq \{0, 1\}^* \) is \( NP-complete \) if

- \( L \in NP \), and
- \( L' \leq_p L \) for every \( L' \in NP \).
P = NP

Furthermore, if $L$ is a language such that $L' \leq_p L$ for some $L' \in \text{NP-complete}$, then $L$ is \textit{NP-hard} [3]. Moreover, if $L \in \text{NP}$, then $L \in \text{NP-complete}$ [3].

One of the first discovered \textit{NP-complete} problems was SAT [5]. An instance of SAT is a Boolean formula $\phi$ which is composed of

- Boolean variables: $x_1, x_2, \ldots$;
- Boolean connectives: Any Boolean function with one or two inputs and one output, such as $\land$(AND), $\lor$(OR), $\neg$(NOT), $\rightarrow$(implication), $\leftrightarrow$(if and only if); and
- parentheses.

A truth assignment for a Boolean formula $\phi$ is a set of values for the variables of $\phi$ and a satisfying truth assignment is a truth assignment that causes it to evaluate to true. A formula with a satisfying truth assignment is a satisfiable formula. The SAT asks whether a given Boolean formula is satisfiable.

One convenient 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

$$((x_1 \lor \neg x_1 \lor \neg x_2) \land (x_3 \lor x_2 \lor x_4) \land (\neg x_1 \lor \neg x_3 \lor \neg x_4))$$

(2.2)

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

Many problems can be proved that belong to NP-complete by a polynomial-time reduction from 3SAT [5]. For example, the problem ONE-IN-THREE 3SAT defined as follows: Given a Boolean formula $\phi$ in 3CNF, is there a truth assignment such that each clause in $\phi$ has exactly one true literal?

\section*{2.2 P-complete class}

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

$$x \in L_1 \text{ if and only if } f(x) \in L_2.$$  \hfill (2.3)

The logarithmic space reduction is frequently used for $P$ and below [9].

There is an important complexity class called \textit{P-complete} [9]. A language $L \subseteq \{0, 1\}^*$ is \textit{P-complete} if

- $L \in P$, and
- $L'$ is logarithmic-space reducible to $L$ for every $L' \in P$. 

One of the $P$-complete problems is $HORNSAT$ [9]. We say that a clause is a Horn clause if it has at most one positive literal [9]. That is, all its literals, except possibly for one, are negations of variables. An instance of $HORNSAT$ is a Boolean formula $\phi$ in $CNF$ which is composed only of Horn clauses [9].

For example, the Boolean formula
\[(\neg x_2 \lor x_3) \land (\neg x_1 \lor x_2 \lor \neg x_3 \lor \neg x_4) \land (x_1)\]  
(2.4)
is a conjunction of Horn clauses. The $HORNSAT$ asks whether an instance of this problem is satisfiable [9].

2.3 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 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 [8]. We denote the $XOR$ function as $\oplus$. The $XOR$-$3SAT$ problem will be equivalent to $XOR$-$SAT$, but the clauses in the formula have exactly three distinct literals. Since $a \oplus b \oplus c$ evaluates to true if and only if exactly 1 or 3 members of $\{a, b, c\}$ are true, each solution of the $ONE-IN-THREE$-$3SAT$ problem for a given $3CNF$ formula is also a solution of the $XOR$-$3SAT$ problem 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 problem called $2SAT$, where we asked whether a given Boolean formula $\phi$ in $2CNF$ is satisfiable. This problem is in $P$ [1].

3 Definition of $\sim P$

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

$L = \{x : (x,y) \in R \text{ for some } y\}$  
(3.1)

where $R$ is a polynomially balanced relation decided by $M$ [9]. According to Cook’s Theorem, a language $L$ is in $NP$ if and only if it has a polynomial-time verifier [9].

**Definition 3.1.** Given two languages, $L_1$ and $L_2$, and two Turing machines, $M_1$ and $M_2$, such that $L_1 \in P$ and $L_2 \in P$ where $M_1$ and $M_2$ are the verifiers of $L_1$ and $L_2$ respectively, we say that a language $L$ belongs to $\sim P$ if,

$L = \{(x,y) : \exists z \text{ such that } M_1(x,z) = \text{"yes" and } M_2(y,z) = \text{"yes" where } x \in L_1 \text{ and } y \in L_2\}$.  
(3.2)

We will call the complexity class $\sim P$ as “equivalent-$P$”.

4 Reduction in $\sim P$

There is a different kind of reduction for $\sim P$: The $e$-reduction.
We define

\[ L \]

Definition 4.1. Given two languages \( L_1 \) and \( L_2 \), such that the instances of \( L_1 \) and \( L_2 \) are ordered pairs of strings, we say that a language \( L_1 \) is \( e \)-reducible to a language \( L_2 \), written \( L_1 \leq_{e} L_2 \), if there exist 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 \( y \in \{0,1\}^* \),

\[
(x,y) \in L_1 \text{ if and only if } (f(x),g(y)) \in L_2.
\] (4.1)

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 also \( L_1 \in C \) [9].

Theorem 4.2. \( \sim P \) is closed under reductions.

Proof. Let \( L \) and \( L' \) be two arbitrary languages, where their instances are ordered pairs of strings, \( L \leq_{e} L' \) and \( L' \in \sim P \). We shall show that \( L \) is in \( \sim P \) too. By definition of \( \sim P \), there are two languages \( L'_1 \) and \( L'_2 \), such that for each \( (v,w) \in L' \) we have that \( v \in L'_1 \) and \( w \in L'_2 \) where \( L'_1 \in P \) and \( L'_2 \in P \). Moreover, there are two Turing machines \( M'_1 \) and \( M'_2 \) which are the verifiers of \( L'_1 \) and \( L'_2 \) respectively, and for each \( (v,w) \in L' \) exists a polynomially bounded certificate \( z \) such that \( M'_1(v,z) = "yes" \) and \( M'_2(w,z) = "yes" \). Besides, by definition of \( e \)-reduction, there exist 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 \( y \in \{0,1\}^* \),

\[
(x,y) \in L \text{ if and only if } (f(x),g(y)) \in L'.
\] (4.2)

From this preliminary information, we can conclude there exist two languages \( L_1 \) and \( L_2 \), such that for each \( (x,y) \in L \) we have that \( x \in L_1 \) and \( y \in L_2 \) where \( L_1 \in P \) and \( L_2 \in P \). Indeed, we could define \( L_1 \) and \( L_2 \) as the instances \( f^{-1}(v) \) and \( g^{-1}(w) \) respectively, 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 \( x \in L_1 \) or \( y \in L_2 \) in polynomial-time just verifying that \( f(x) \in L'_1 \) or \( g(y) \in L'_2 \) respectively, because \( L'_1 \in P, L'_2 \in P \) and \( SPACE(\log n) \in P \) [9]. Furthermore, there exist two Turing machines \( M_1 \) and \( M_2 \) which are the verifiers of \( L_1 \) and \( L_2 \) respectively, and for each \( (x,y) \in L \) exists a polynomially bounded certificate \( z \) such that \( M_1(x,z) = "yes" \) and \( M_2(y,z) = "yes" \). Indeed, we could know whether \( M_1(x,z) = "yes" \) and \( M_2(y,z) = "yes" \) for some polynomially bounded string \( z \) just verifying whether \( M'_1(f(x),z) = "yes" \) and \( M'_2(g(y),z) = "yes" \). That is, we may have that \( M_1(x,z) = M'_1(f(x),z) \) and \( M_2(y,z) = M'_2(g(y),z) \), because we can evaluate \( f(x) \) and \( g(y) \) in polynomial-time since \( SPACE(\log n) \in P \) [9]. In this way, we have proved that \( L \in \sim P \).

\[ \square \]

5 \( \sim P = NP \)

We define \( \sim ONE-IN-THREE \text{ 3SAT} \) as follows,

\[
\sim ONE-IN-THREE \text{ 3SAT} = \{ (\phi, \phi) : \phi \in ONE-IN-THREE \text{ 3SAT} \}.
\] (5.1)

It is trivial to see the \( \sim ONE-IN-THREE \text{ 3SAT} \) problem remains in \( NP \)-complete (see Section 2).

Definition 5.1. \( 3XOR-2SAT \) is a problem in \( \sim P \), such that if \( (\psi, \phi) \in 3XOR-2SAT \), then \( \psi \in XOR \text{ 3SAT} \) and \( \phi \in 2SAT \). That is, the instances of \( XOR \text{ 3SAT} \) and \( 2SAT \) (see Section 2) that can have the same satisfying truth assignment (with the same variables).
Theorem 5.2. $\sim$ONE-IN-THREE 3SAT $\leq$ 3XOR-2SAT.

Proof. Given an arbitrary Boolean formula $\phi$ in 3CNF of $m$ clauses, we will iterate for each clause $c_i = (x \lor y \lor z)$ in $\phi$, where $x$, $y$ and $z$ are literals, and create the following formulas,

$$Q_i = (x \oplus y \oplus z)$$  \hspace{1cm} (5.2)

$$P_i = (\neg x \lor \neg y) \land (\neg y \lor \neg z) \land (\neg x \lor \neg z).$$  \hspace{1cm} (5.3)

Since $Q_i$ evaluates to true if and only if exactly 1 or 3 members of $\{x, y, z\}$ are true and $P_i$ evaluates to true if and only if exactly 1 or 0 members of $\{x, y, z\}$ are true, we obtain the clause $c_i$ has exactly one true literal if and only if both formulas $Q_i$ and $P_i$ are satisfiable with the same truth assignment. Hence, we can create the $\psi$ and $\varphi$ formulas as the conjunction of the $Q_i$ and $P_i$ formulas for every clause $c_i$ in $\phi$, that is, $\psi = Q_1 \land \ldots \land Q_m$ and $\varphi = P_1 \land \ldots \land P_m$. Finally, we obtain that,

$$(\phi, \varphi) \in \sim\text{ONE-IN-THREE 3SAT} \text{ if and only if } (\psi, \varphi) \in 3\text{XOR-2SAT}.$$  \hspace{1cm} (5.4)

In addition, there exist two logarithmic-space computable functions $f : \{0, 1\}^* \rightarrow \{0, 1\}^*$ and $g : \{0, 1\}^* \rightarrow \{0, 1\}^*$ 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 at once each clause $c_i$ in the input $\phi$ and generate $Q_i$ or $P_i$ to the output, since the complexity class $\text{SPACE}(\log n)$ does not take the length of the input and the output into consideration [9].

Theorem 5.3. $\sim P = NP$.

Proof. If there is an NP-complete problem reducible to a problem in $\sim P$, then this NP-complete problem will be in $\sim P$, and thus, $\sim P = NP$, because $\sim P$ is closed under reductions (see Theorem 4.2) and NP too [9]. Therefore, this is a direct consequence of Theorem 5.2.

6 P = NP

We define $\sim\text{HORNSAT}$ as follows,

$$\sim\text{HORNSAT} = \{ (\phi, \varphi) : \phi \in \text{HORNSAT} \}.$$  \hspace{1cm} (6.1)

It is trivial to see the $\sim\text{HORNSAT}$ problem remains in $P$-complete (see Section 2).

Theorem 6.1. $\sim\text{HORNSAT} \in \sim P$.

Proof. The $\sim\text{HORNSAT}$ problem complies with all the properties of a language in $\sim P$. That is, for each $(\phi, \varphi) \in \sim\text{HORNSAT}$, the Boolean formula $\phi$ belongs to a language in $P$, that is, the same HORNSAT. In addition, the verifier $M$ of HORNSAT complies that always exists a polynomially bounded certificate $z$ when $\phi$ is satisfiable, that is the satisfying truth assignment of $\phi$, such that $M(\phi, z) = \text{"yes"}$. Certainly, we can prove this result, because any ordered pair of Boolean formulas in $\sim\text{HORNSAT}$ can share the same certificate due to they are equals.
Theorem 6.2. $\sim P = P$.

Proof. If a $P$-complete problem is in $\sim P$, then $\sim P = P$, because $\sim P$ is closed under reductions (see Theorem 4.2) and $P$ too [9]. Therefore, this is a direct consequence of Theorem 6.1. ☐

Theorem 6.3. $P = NP$.

Proof. Since $\sim P = NP$ and $\sim P = P$ as result of Theorems 5.3 and 6.2, then we can conclude that $P = NP$. ☐

References


AUTHOR

Frank Vega
La Portada, Cotorro
Havana, Cuba
vega.frank@gmail.com

ABOUT THE AUTHOR

FRANK VEGA is graduated as Bachelor of Computer Science from The University of Havana since 2007. He has worked as specialist in Datys, Playa, Havana, Cuba. His principal area of interest is in computational complexity.