



## On P versus NP

Frank Vega

### ► To cite this version:

| Frank Vega. On P versus NP. 2017. hal-01509423

**HAL Id: hal-01509423**

**<https://hal.science/hal-01509423>**

Preprint submitted on 17 Apr 2017

**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.

# On P versus NP

Frank Vega

*April 17, 2017*

**Abstract:** P versus NP is considered one of the great open problems of 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 John Nash to the National Security Agency in 1955. However, 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 huge problem have failed. Another major complexity class is coNP. Whether  $NP = coNP$  is another fundamental question that it is as important as it is unresolved. We prove there exists a problem in coNP that is not in P. In this way, we show that P is not equal to coNP. Since  $P = NP$  implies  $P = coNP$ , then we also demonstrate that P is not equal to NP.

## Introduction

P versus NP is a major unsolved problem in computer science [3]. It is considered by many to be the most important open problem in the field [3]. 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 [3].

In 1936, Turing developed his theoretical computational model [1]. The deterministic and nondeterministic Turing machines have become in two of the most important definitions related to this theoretical model for computation. A deterministic Turing machine has only one next action for each step defined in its program or transition function [6]. 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 [6].

Another huge 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 [2]. A complexity class is a

**ACM Classification:** F.1.3.3

**AMS Classification:** 68Q15, 68Q17

**Key words and phrases:** P, NP, coNP, coNP-complete, Minimum, Boolean circuit

set of problems, which are represented as a language, grouped by measures such as the running time, memory, etc [2].

In the computational complexity theory, the class  $P$  contains those languages that can be decided in polynomial time by a deterministic Turing machine [5]. The class  $NP$  consists in those languages that can be decided in polynomial time by a nondeterministic Turing machine [5].

The biggest open question in theoretical computer science concerns the relationship between these classes: Is  $P$  equal to  $NP$ ? In 2002, a poll of 100 researchers showed that 61 believed that the answer was not, 9 believed that the answer was yes, and 22 were unsure; 8 believed the question may be independent of the currently accepted axioms and so impossible to prove or disprove [4]. All efforts to solve the  $P$  versus  $NP$  problem have failed [6].

Another major complexity class is  $coNP$  [6]. We show a new kind of reduction that we called the Conjunction reduction. Using this definition as an argument, we prove there is a problem in  $coNP$  that is not in  $P$ . Since  $P = NP$  implies that every  $coNP$  problem is in  $P$ , then we can deduce that  $P \neq NP$  [6].

## 1 Theoretical notions

Let  $\Sigma$  be a finite alphabet with at least two elements, and let  $\Sigma^*$  be the set of finite strings over  $\Sigma$  [1]. A Turing machine  $M$  has an associated input alphabet  $\Sigma$  [1]. For each string  $w$  in  $\Sigma^*$  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,  $M(w) = \text{"yes"}$  [1]. Note that  $M$  fails to accept  $w$  either if this computation ends in the rejecting state, or if the computation fails to terminate [1].

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

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

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  $\Sigma^n$  is the set of all strings over  $\Sigma$  of length  $n$  [1]. We say that  $M$  runs in polynomial time if there exists  $k$  such that for all  $n$ ,  $T_M(n) \leq n^k + k$  [1].

A language  $L$  is in class  $P$  if  $L = L(M)$  for some deterministic Turing machine  $M$  which runs in polynomial time [1]. We state the complexity class  $NP$  using the following definition:

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  $w$  [7]. 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.

For polynomial time verifiers, the certificate is polynomially bounded by the length of  $w$ , because that is all the verifier can access in its time bound [7].  $NP$  is the class of languages that have polynomial time verifiers [7].

If  $NP$  is the class of problems that have succinct certificates, then the complexity class  $coNP$  must contain those problems that have succinct disqualifications [6]. That is, a “no” instance of a problem in  $coNP$  possesses a short proof of its being a “no” instance [6].

A function  $f : \Sigma^* \rightarrow \Sigma^*$  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 [7]. 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 exists a polynomial time computable function  $f : \{0, 1\}^* \rightarrow \{0, 1\}^*$  such that for all  $x \in \{0, 1\}^*$ ,

$$x \in L_1 \text{ iff } f(x) \in L_2$$

where *iff* means “if and only if”. An important complexity class is  $coNP$ -complete [5]. A language  $L \subseteq \{0, 1\}^*$  is  $coNP$ -complete if

1.  $L \in coNP$ , and
2.  $L' \leq_p L$  for every  $L' \in coNP$ .

Furthermore, if  $L$  is a language such that  $L' \leq_p L$  for some  $L' \in coNP$ -complete, then  $L$  is in  $coNP$ -hard [2]. Moreover, if  $L \in coNP$ , then  $L \in coNP$ -complete [2].

A principal  $coNP$ -complete problem is  $CIRCUIT$ -UNSAT [5]. An instance of  $CIRCUIT$ -UNSAT is a Boolean circuit  $C$  which is a directed acyclic graph  $C = (V, E)$ , where the nodes  $V = \{1, \dots, n\}$  are called the gates of  $C$ . We can assume that all edges are of the form  $(i, j)$  where  $i < j$ . All nodes in the graph have in-degree (number of incoming edges) equal to 0, 1 and 2. Also, each gate  $i \in V$  has a sort  $c(i)$  associated with it, where  $c(i) \in \{true, false, \wedge, \vee, \neg\} \cup \{x_1, x_2, \dots\}$ . If  $c(i) \in \{true, false\} \cup \{x_1, x_2, \dots\}$ , then the in-degree of  $i$  is 0, that is,  $i$  must have no incoming edges. Gates with no incoming edges are called the inputs of  $C$ . If  $c(i) = \neg$ , then  $i$  has in-degree one. If  $c(i) \in \{\wedge, \vee\}$ , then the in-degree of  $i$  must be two. Finally, node  $n$  (the largest numbered gate in the circuit, which necessarily has no outgoing edges), is called the output gate of the circuit.

Let  $X(C)$  be the set of all Boolean variables that appear in the circuit  $C$  (that is,  $X(C) = \{x \in X : c(i) = x \text{ for some gate } i \text{ in } C\}$ ). We say that a truth assignment  $T$  is appropriate for  $C$  if it is defined for all the variables in  $X(C)$ . Given such a  $T$ , the truth value of gate  $i \in V$ ,  $T(i)$ , is defined, by induction on  $i$ , as follows: If  $c(i) = true$  then  $T(i) = true$ , and similarly if  $c(i) = false$ . If  $c(i) \in X$ , then  $T(i) = T(c(i))$ . If now  $c(i) = \neg$ , then there is a unique gate  $j < i$  such that  $(j, i) \in E$ . By induction, we know  $T(j)$ , and then  $T(i)$  is true if  $T(j) = false$ , and vice versa. If  $c(i) = \vee$ , then there are two edges  $(j, i)$  and  $(j', i)$  entering  $i$ .  $T(i)$  is then true if and only if at least one of  $T(j)$ ,  $T(j')$  is true. If  $c(i) = \wedge$ , then  $T(i)$  is true if and only if both  $T(j)$  and  $T(j')$  are true, where  $(j, i)$  and  $(j', i)$  are the incoming edges. Finally, the value of the circuit,  $T(C)$ , is  $T(n)$ , where  $n$  is the output gate.

The  $CIRCUIT$ -UNSAT can be formulated as follows: Given a Boolean circuit  $C$ , is not there any truth assignment  $T$ , appropriate to  $C$ , such that  $T(C) = true$ ?

## 2 Results

### 2.1 Conjunction reduction

**Definition 2.1.** We say that two languages  $L_1 \subseteq \{0, 1\}^*$  and  $L_2 \subseteq \{0, 1\}^*$  are conjunctive reducible to a language  $L_3 \subseteq \{0, 1\}^*$ , written  $L_1 \wedge L_2 \leq_c L_3$ , if for all  $x \in \{0, 1\}^*$ ,  $y \in \{0, 1\}^*$  and  $z \in \{0, 1\}^*$ ,

$$(x, y) \in L_1 \wedge (y, z) \in L_2 \text{ iff } (x, z) \in L_3$$

where *iff* means “if and only if”.

Just as  $O$ -notation provides an asymptotic upper bound,  $\Omega$ -notation provides an asymptotic lower bound [2]. In computer science,  $O$ -notation and  $\Omega$ -notation are used to classify algorithms according to their running time or space requirements [2].

**Theorem 2.2.** If  $L_1 \wedge L_2 \leq_c L_3$  with  $L_1 \notin O(g(n))$  and  $L_2 \notin O(g(n))$ , then  $L_3 \notin O(g(n))$ .

*Proof.* Suppose that  $L_1 \wedge L_2 \leq_c L_3$  with  $L_1 \notin O(g(n))$  and  $L_2 \notin O(g(n))$ , but  $L_3 \in O(g(n))$ . If  $L_1 \notin O(g(n))$  and  $L_2 \notin O(g(n))$ , then for all  $x \in \{0, 1\}^*$ ,  $y \in \{0, 1\}^*$  and  $z \in \{0, 1\}^*$  the problem of deciding whether  $(x, y) \in L_1 \wedge (y, z) \in L_2$  will not be in  $O(g(n))$  due to the properties of the  $O$ -notation, but this is a contradiction since we assumed  $L_3 \in O(g(n))$ . Certainly, if we would have  $L_3 \in O(g(n))$ , then we could solve  $(x, y) \in L_1 \wedge (y, z) \in L_2$  in  $O(g(n))$  too. But this is not possible, because  $O(g)$  is a convex cone and thus  $f_1 \notin O(g) \wedge f_2 \notin O(g) \Rightarrow f_1 + f_2 \notin O(g)$ . Therefore, for the sake of contradiction we have  $L_3 \notin O(g(n))$ .  $\square$

### 2.2 The Problem MINIMUM

**Definition 2.3.** Given a set  $S$  of  $n$  positive integers, *SEARCH-MINIMUM* is the problem of finding the minimum of  $S$ .

How many comparisons are necessary to determine the minimum of a set of  $n$  positive integers? We can easily obtain an upper bound of  $n - 1$  comparisons: examine each integer of the set in turn and keep track of the smallest element seen so far [2]. Is this the best we can do? Yes, since we can obtain a lower bound of  $n - 1$  comparisons for the problem of determining the minimum [2]. Think of any algorithm that determines the minimum as a tournament among the elements [2]. Each comparison is a match in the tournament in which the smaller of the two elements wins [2]. The key observation is that every element except the winner must lose at least one match [2]. Hence,  $n - 1$  comparisons are necessary to determine the minimum, and the algorithm *SEARCH-MINIMUM* is optimal with respect to the number of comparisons performed [2].

**Definition 2.4.** Given a number  $x$  and a set  $S$  of  $n$  positive integers, *MINIMUM* is the problem of deciding whether  $x$  is the minimum of  $S$ .

How many comparisons are necessary to determine whether some  $x$  is the minimum of a set of  $n$  positive integers? We can easily obtain an upper bound of  $n$  comparisons: find the minimum in the set and check whether the result is equal to  $x$ . Is this the best we can do? Yes, since we can obtain a lower bound of  $n - 1$  comparisons for the problem of determining the minimum and another obligatory comparison for checking whether that minimum is equal to  $x$ .

**Theorem 2.5.**  $MINIMUM \notin O(\sqrt{|S|})$ .

*Proof.* As we mentioned above, the problem  $MINIMUM$  complies with  $MINIMUM \in \Omega(|S|)$  and therefore  $MINIMUM \notin O(\sqrt{|S|})$ , where  $|S| = n$  is the cardinality of the set  $S$  with  $n$  positive integers.  $\square$

### 2.3 The Problem REPRESENTATION

**Definition 2.6.** A representation of a set  $S$  with  $n$  positive integers is a Boolean circuit  $C$ , such that  $C$  accepts the binary representation of a bit integer  $i$  (translated the bit 1 to true and 0 to false over the input variable gates) iff  $i \in S$ .

**Definition 2.7.** Given a set  $S$  of  $n$  positive integers and a Boolean circuit  $C$ ,  $REPRESENTATION$  is the problem of deciding whether  $C$  is a representation of the set  $S$ .

**Theorem 2.8.**  $REPRESENTATION \notin O(\sqrt{|S|})$ .

*Proof.* Since the empty set cannot be represented by a Boolean circuit  $C$  such that there is some truth assignment  $T$ , appropriate to  $C$ , where  $T(C) = true$ , then we could make a polynomial time reduction as follows:

$$C \in CIRCUIT-UNSAT \text{ iff } (\emptyset, C) \in REPRESENTATION.$$

However, this reduction can be made in constant time. That means we cannot decide every instance  $(\emptyset, C) \in REPRESENTATION$  in constant time, because that would mean we can solve  $CIRCUIT-UNSAT$  in constant time. Indeed, even though we would have the assumption of  $P = NP$ , we could not solve a  $coNP$ -complete problem in constant time [6]. Nevertheless, in an instance  $(\emptyset, C) \in REPRESENTATION$ , we would have  $S = \emptyset$  and  $|S| = 0$ . Thus, we can assure if  $REPRESENTATION \in O(\sqrt{|S|})$ , then we could solve  $CIRCUIT-UNSAT$  in constant time. For that reason, we can confirm  $REPRESENTATION \notin O(\sqrt{|S|})$ .  $\square$

### 2.4 The Problem SUCCINCT-MINIMUM

**Definition 2.9.** Given a positive integer  $x$  and a Boolean circuit  $C$ , we define  $SUCCINCT-MINIMUM$  as the problem of deciding whether  $x$  is the minimum bit integer which accepts  $C$  as input.

**Theorem 2.10.**  $MINIMUM \wedge REPRESENTATION \leq_c SUCCINCT-MINIMUM$ .

*Proof.* Certainly, for every instance  $(x, C)$  of  $SUCCINCT-MINIMUM$  we have the following property,

$$(x, S) \in MINIMUM \wedge (S, C) \in REPRESENTATION$$

$$\text{iff } (x, C) \in SUCCINCT-MINIMUM.$$

$\square$

**Theorem 2.11.**  $SUCCINCT-MINIMUM \notin O(\sqrt{|S|})$  where  $S$  is the set that represents the Boolean circuit  $C$ .

*Proof.* As result of Theorems 2.2, 2.5, 2.8 and 2.10, then we have  $SUCCINCT-MINIMUM \notin O(\sqrt{|S|})$ .  $\square$

**Theorem 2.12.**  $SUCCINCT-MINIMUM \notin P$ .

*Proof.* For certain kind of instances, the input  $(x, C)$  is exponentially more succinct than the cardinality of the set  $S$  that represents  $C$ . Since we have that  $SUCCINCT-MINIMUM \notin O(\sqrt{|S|})$ , then we could not decide every instance of  $SUCCINCT-MINIMUM$  in polynomial time.  $\square$

**Theorem 2.13.**  $SUCCINCT-MINIMUM \in coNP$ .

*Proof.* If  $(x, C) \notin SUCCINCT-MINIMUM$ , then it would exist a positive integer  $y$  such that  $y < x$  and  $C$  accepts the bit integer  $y$ . Since we can evaluate whether  $C$  accepts the bit integer  $y$  in polynomial time and we have that  $y$  is polynomially bounded by  $x$ , then we can confirm  $SUCCINCT-MINIMUM \in coNP$ .  $\square$

**Theorem 2.14.**  $P \neq NP$ .

*Proof.* If any single  $coNP$  problem cannot be solved in polynomial time, then  $P \neq coNP$  [6]. Certainly,  $P = NP$  implies  $P = coNP$  because  $P$  is closed under complement, and therefore, we can conclude  $P \neq NP$  [6].  $\square$

## Conclusions

This proof explains why after decades of studying the  $NP$  problems no one has been able to find a polynomial time algorithm for any of more than 300 important known  $NP$ -complete problems [5]. Indeed, it shows in a formal way that many currently mathematical problems cannot be solved efficiently, so that the attention of researchers can be focused on partial solutions or solutions to other problems.

Although this demonstration removes the practical computational benefits of a proof that  $P = NP$ , it would represent a very significant advance in computational complexity theory and provide guidance for future research. In addition, it proves that could be safe most of the existing cryptosystems such as the public key cryptography [5]. On the other hand, we will not be able to find a formal proof for every theorem which has a proof of a reasonable length by a feasible algorithm.

## References

- [1] SANJEEV ARORA AND BOAZ BARAK: *Computational complexity: A modern approach*. Cambridge University Press, 2009. 1, 2
- [2] THOMAS H. CORMEN, CHARLES ERIC LEISERSON, RONALD L. RIVEST, AND CLIFFORD STEIN: *Introduction to Algorithms*. MIT Press, 2 edition, 2001. 1, 2, 3, 4
- [3] LANCE FORTNOW: *The Golden Ticket: P, NP, and the Search for the Impossible*. Princeton University Press. Princeton, NJ, 2013. 1
- [4] WILLIAM I. GASARCH: The  $P=?NP$  poll. *SIGACT News*, 33(2):34–47, 2002. 2

- [5] ODED GOLDREICH: *P, Np, and Np-Completeness*. Cambridge: Cambridge University Press, 2010. [2](#), [3](#), [6](#)
- [6] CHRISTOS H. PAPADIMITRIOU: *Computational Complexity*. Addison-Wesley, 1994. [1](#), [2](#), [3](#), [5](#), [6](#)
- [7] MICHAEL SIPSER: *Introduction to the Theory of Computation*. Thomson Course Technology, 2 edition, 2006. [2](#), [3](#)

#### AUTHOR

Frank Vega  
Joysonic  
Uzun Mirkova 5  
Belgrade, Serbia  
[vega.frank@gmail.com](mailto:vega.frank@gmail.com)

#### ABOUT THE AUTHOR

Frank Vega is graduated as Bachelor of Computer Science from The University of Havana since 2007. He lives and works as computer programmer in Belgrade, Serbia. He loves to write poems and stories in his free time and he has published a literary book called [Protesta](#).