The P versus NP Problem

Frank Vega

The P versus NP Problem

Frank Vega

February 16, 2015

Abstract: The *P* versus *NP* problem has become in one of the most interesting and crucial questions for many fields such as computer science, mathematics, biology and others. This outstanding problem consists in knowing the answer of the following incognita: Is *P* equal to *NP*? Many computer scientists have believed the most probable answer is $P \neq NP$. A key reason for this belief is the possible solution of $P = NP$ would imply many startling results that are currently believed to be false. Besides *P* and *NP*, another major complexity class is *coNP*. It is a known result if $P = NP$, then $NP = coNP$. Therefore, if we prove $P \neq coNP$, then this would be sufficient to show $P \neq NP$. Indeed, in this work we have demonstrated the existence of a *coNP* problem that is not in *P*, and thus, $P \neq coNP$. In this way, we show the belief of almost all computer scientists was a truly supposition.

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 [5].

The Turing machine has been an useful concept in theory of computing since it was created by Alan Turing in the last century [12]. Since then, it has appeared new definitions related with this concept 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 machine can contain more than one action defined for each step of the program where this program is not a function but a relation [6].

ACM Classification: F.1.3

AMS Classification: 68-XX, 68Qxx, 68Q15

Key words and phrases: P, NP, coNP, Turing machine

Frank Vega
Google+

Another huge advance 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 [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].

In computational complexity theory, the class P consists of all those decision problems (defined as languages) that can be solved on a deterministic Turing machine in an amount of time that is polynomial in the size of the input; the class NP consists of all those decision problems whose positive solutions can be verified in polynomial time given the right information, or equivalently, whose solution can be found in polynomial time on a nondeterministic Turing machine [11].

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 [7].

Besides P and NP, another major complexity class is $coNP$: the "complement" of NP [1]. A problem is in $coNP$ if a "*no*" answer can be checked in polynomial time [1]. The string which is used to verify an instance of a problem in NP is known as certificate [10]. If NP is the class of problems that have succinct certificates, then the complexity class $coNP$ must contain those problems that have succinct disqualifications [10].

There is a known result which states if $P = NP$, then $NP = coNP$ [1]. We prove the existence of a $coNP$ problem that is not in P, and thus, $P \neq coNP$. Hence, we show the P versus NP problem has the following solution: $P \neq NP$.

2 Theory

The argument made by Alan Turing in the twentieth century proves mathematically that for any computer program we can create an equivalent Turing machine [12]. A Turing machine M has a finite set of states and a finite set of symbols called the alphabet of M. The set of states has a special state which is known as the initial state.

The operations of a Turing machine are based on a transition function which takes the initial state with a string of symbols of the alphabet that is known as the input. Then, it proceeds to reading the symbols on the cells contained in a tape through a head or cursor. At the same time, the symbols on each step are erased and written by the transition function and later moved to the left, right or remained in the same place for each cell. Finally, this process is interrupted if it halts in a final state: the state of acceptance "*yes*", the rejection "*no*" or halting h [10].

A Turing machine halts if it reaches a final state. If a Turing machine M accepts or rejects a string x, then $M(x) =$ "*yes*" or "*no*" is respectively written. If it reaches the halting state h, we write $M(x) = y$, where the string y is considered as the output string, i.e., the string remaining in M when this halts [10].

Let $\sum$ be a finite alphabet (that is, a finite nonempty set) with at least two elements, and let $\sum^*$ be the set of finite strings over $\sum$ [3]. Then a language over $\sum$ is a subset L of $\sum^*$ [3]. The language accepted by M Turing machine, denoted $L(M)$, has associated alphabet $\sum$ and is defined by

$$\{< w >: M\ accepts\ w\} \tag{2.1}$$

where we denote $< w > \in \sum^{*}$ as the encoding of the instance w in $\sum$ [3], [4]. In conclusion, these are some basic notions that could help you to understand this paper.

3 Results

We are going to explain two simple notations and define our principal problem.

Definition 3.1. For a binary string as argument, the notations $|..|$ and $n(..)$ are the length and the decimal integer representation respectively.

Definition 3.2.

$$\{< M, n(x) >: \nexists y \in \{0,1\}^{*}\ such\ that\ M(y) = \text{“}yes\text{”} \tag{3.1}$$

$$in\ less\ than\ or\ equal\ to\ (|M| + |y|)^{3}\ steps\ and\ n(y) \leq n(x)\} \tag{3.2}$$

An instance of this problem would be M and $n(x)$ where M is a deterministic Turing machine and $n(x)$ is the decimal integer representation of the x binary string. This language consists in all the M deterministic Turing machines with $n(x)$ positive integers where there is not any y binary string such that M accepts y in less than or equal to $(|M| + |y|)^{3}$ steps (the amount of steps is the number of actions on the transition function with the running of $M(y)$) and the decimal integer representation of y would be a positive integer that is less than or equal to $n(x)$. We will denote this language as $NEVER - BOUNDED - HALT$.

Next, we show two important results which are the keys in this proof.

Lemma 3.3. *$NEVER - BOUNDED - HALT \in coNP$.*

Proof. If some deterministic Turing machine M and an $n(x)$ positive integer are not an instance of $NEVER - BOUNDED - HALT$, then this is because of the existence of some y binary string which is accepted by M in less than or equal to $(|M| + |y|)^{3}$ steps where $|y|^{3} \leq |x|^{3}$ due to $n(y) \leq n(x)$. Indeed, this y string will be a succinct disqualification in which we could prove in at most $(|M| + |x|)^{3}$ steps (this is in polynomial time) that $< M, n(x) >$ is a "*no*" instance of $NEVER - BOUNDED - HALT$. □

Theorem 3.4. *$NEVER - BOUNDED - HALT \notin P$.*

Proof. How many running of M with the y binary strings are necessary to determine when some deterministic Turing machine M and an $n(x)$ positive integer belongs to $NEVER - BOUNDED - HALT$ where $n(y) \leq n(x)$? We can easily obtain an upper bound of $(n(x) + 1)$ running: examine each y string from 0 to $n(x)$ as binary encoding and verify whether $M(y)$ does not accept in less than or equal to $(|M| + |y|)^{3}$ steps. However, this is not a polynomial time algorithm, because the amount $(n(x) + 1)$ could be exponential in relation with the size of $< M, n(x) >$. The aim question would be: Is there a polynomial time algorithm for $NEVER - BOUNDED - HALT$?

Definition 3.5.

$$\{< M, n(y) >: M(y) \text{ does not accept in less than or equal to } (|M| + |y|)^3 \text{ steps}\} \tag{3.3}$$

An instance of this problem would be M and $n(y)$ where M is a deterministic Turing machine and $n(y)$ is the decimal integer representation of the y binary string. We are going to denote this languages as $ONE - NEVER - BOUNDED - HALT$.

Suppose we have a deterministic Turing machine M_{MAGIC} which decides $ONE - NEVER - BOUNDED - HALT$ for any input $< M, n(y) >$ in less than or equal to $((|M| + |y|)^3 - 10)$ steps. Then, we can have a Turing machine D which receives a deterministic Turing machine M and its $n(M)$ decimal integer representation as input and we just translate it from the Turing machine model into an understandable pseudo-code in the following lines.

```
(1) if $M_{MAGIC}(M, n(M)) =$ "yes"
(2)     then accept
(3)     else never halt
```

Does $D(D, n(D))$ accept in less than or equal to $(|D| + |D|)^3$ steps? Certainly, we could not know the answer of this question, because we assumed an absurd statement, i.e., the existence of M_{MAGIC}. Hence, we can affirm about the $ONE - NEVER - BOUNDED - HALT$ problem the following conclusion: we cannot always accept any instance $< M, n(y) > \in ONE - NEVER - BOUNDED - HALT$ by a Turing machine in less than or equal to $((|M| + |y|)^3 - 10)$ steps.

We can see any instance $< M, n(x) >$ of $NEVER - BOUNDED - HALT$ is an exponentially more succinct input of $(n(x) + 1)$ different instances $< M, n(y) >$ of $ONE - NEVER - BOUNDED - HALT$ where $n(y) \leq n(x)$. Indeed, the definition of $NEVER - BOUNDED - HALT$ for any input $< M, n(x) >$ could be rephrased as the acceptance of $(n(x) + 1)$ different inputs $< M, n(y) >$ of $ONE - NEVER - BOUNDED - HALT$ when each $n(y)$ is between 0 and $n(x)$.

In addition, we could reduce any $(n(x) + 1)$ different instances $< M, n(y) > \in ONE - NEVER - BOUNDED - HALT$ into a single instance $< M, n(x) > \in NEVER - BOUNDED - HALT$ by a polynomial time algorithm in $O(n(x))$ where $n(y) \leq n(x)$. The reduction algorithm will be very simple if we use as a key the $n(y)$ integer for each instance $< M, n(y) >$.

Let's see the steps of this algorithm.

- First, we count the amount of instances $< M, n(y) >$ (the result would be a positive integer m).
- Next, we create an array of length m (denoted as b).
- Then, we sort in linear time the instances $< M, n(y) >$ with the array indexing (using the $n(y)$ keys in this way $b[n(y)] =< M, n(y) >$ and verifying before that each $n(y)$ integer complies with $n(y) \leq m - 1$ otherwise we halt the reduction in rejection) as a tool for determining the relative order just like we do in the counting sort algorithm [4].

- After that, we verify whether each positive integer between 0 and $m-1$ corresponds to an $n(y)$ index of a cell with some $< M, n(y) >$ information inside the b array of length m just in the following way: we are going to check in a sequential and ascending way whether there is not any empty cell inside the b array.

- Finally, if the verification is successful, then we take the $n(x)$ value as the positive integer $m-1$ else we reject.

Now, suppose we have a polynomial time algorithm for $NEVER-BOUNDED-HALT$. This would mean, when we utilize the reduction above, that we could accept any $(n(x)+1)$ different instances $< M, n(y) >\in ONE-NEVER-BOUNDED-HALT$ in less than $c \times (n(x)+| < M, n(x) > |^{k})$ steps for a feasible and fixed constants c and k where $n(y) \leq n(x)$. However, this is only possible if we can always accept, at the same time, every element of a nonempty subset of these $(n(x)+1)$ different and arbitrary instances $< M, n(y) >\in ONE-NEVER-BOUNDED-HALT$ by a Turing machine in less than $((|M|+|y|)^{3}-10)$ steps when $n(x)$ is a sufficiently large integer. But, this is an absurd, because there is no such deterministic Turing machine (this means there is no possible algorithm) that could achieve this implication as we proved before. For that reason, we obtain $NEVER-BOUNDED-HALT \notin P$. □

Finally, we show our aim result.

Theorem 3.6. *$P \neq NP$.*

Proof. If $P = NP$, then $P = NP = coNP$ [1]. Then, a single problem in $coNP$ and not in P is sufficient to prove $P \neq NP$, because $P \neq coNP$ will imply the $P = NP = coNP$ statement is false. Therefore, this is a direct consequence of Lemma 3.3 and Theorem 3.4. □

4 Conclusions

This proof explains why after decades of studying these problems no one has been able to find a polynomial time algorithm for any of more than 3000 important known $NP-complete$ problems. Indeed, this demonstration removes the practical computational benefits of a proof that $P = NP$, but would nevertheless represent a very significant advance in computational complexity theory and provide guidance for future research.

It shows in a formal way that many currently mathematically problems cannot be solved efficiently, so that the attention of researchers can be focused on partial solutions or solutions to other problems. In addition, it proves that could be safe most of the existing cryptosystems such as the public-key cryptography and the symmetric ciphers [8]. 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.

Acknowledgement

The author would like to thank Marzio de Biasi for his comments about this paper provided by email.

References

[1] SCOTT AARONSON: PHYS771 Lecture 6: P, NP, and Friends, 2007. 2, 5

[2] STEPHEN A. COOK: The complexity of theorem proving procedures. In *Proceedings of the 3rd Annual ACM Symposium on the Theory of Computing (STOC'71)*, pp. 151–158. ACM Press, 1971. 1

[3] STEPHEN A. COOK: The P versus NP Problem. *Clay Mathematics Institute*, 2000. 2, 3

[4] THOMAS H. CORMEN, CHARLES ERIC LEISERSON, RONALD L. RIVEST, AND CLIFFORD STEIN: *Introduction to Algorithms*. MIT Press, Second edition, 2001. 2, 3, 4

[5] LANCE FORTNOW: The Status of the P versus NP Problem. *Communications of the ACM*, 52(9):78–86, September 2009. 1

[6] MICHAEL R. GAREY AND DAVID S. JOHNSON: *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, First edition, 1979. 1

[7] WILLIAM I. GASARCH: The P=?NP poll. *SIGACT News*, 33(2):34–47, 2002. 2

[8] ODED GOLDREICH: *Foundations of Cryptography, Basic Tools*. Cambridge University Press, 2001. 5

[9] HARRY R. LEWIS AND CHRISTOS H. PAPADIMITRIOU: *Elements of the Theory of Computation*. Prentice Hall, Second edition, 1998. 1

[10] CHRISTOS H. PAPADIMITRIOU: *Computational Complexity*. Addison-Wesley, 1994. 2

[11] MICHAEL SIPSER: *Introduction to the Theory of Computation*. Thomson Course Technology, Second, International edition, 2006. 2

[12] ALAN M. TURING: On Computable Numbers, with an Application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 42:230–265, 1936. 1, 2

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.