The P =? NP solution

Frank Vega

The P =? NP solution

Frank Vega

April 5, 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. Since then, many computer scientists have believed the most probable answer is that P is different to NP. A key reason for this belief is the possible solution of P=NP would imply many startling results that are believed to be false. We prove 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 [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.

Informally, the solution of this problem requires to find out whether every problem whose solution can be quickly verified by a computer can also be quickly solved by a computer. The informal term quickly used above means the existence of an algorithm for the task that runs in polynomial time. The general class of questions for which some algorithm can provide an answer in polynomial time is called P. For some questions, there is no known way to find an answer quickly, but if one is provided with information showing what the answer is, it may be possible to verify the answer quickly. The class of questions for which an answer can be verified in polynomial time is called NP.

ACM Classification: F.1.3

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

Key words and phrases: P, NP, Turing machine, POLYLOGTIME, NLOGTIME, Maximum

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 [5]. We prove the belief of almost all computer scientists was a truly supposition.

2 Theoretical framework

Let's explain a simple notation that we frequently use in the paper.

Definition 2.1. For any instance I, the notation $|I|$ is the bit-length of I.

The following subsections will help you to understand better this proof.

2.1 The Turing machine model

The argument made by Alan Turing in the twentieth century states that for any algorithm we can create an equivalent Turing machine [9].

Definition 2.2. A Turing machine is a quadruple $M = (K, \Sigma, \delta, s)$. K is a finite set of states; $s \in K$ is the initial state. Σ is a finite set of symbols (we say Σ is the alphabet of M). We assume K and Σ are disjoint sets. Σ always contains the special symbols $\sqcup$ and $\triangleright$: The blank and first symbol. Finally, δ is a transition function, which maps $K \times \Sigma$ to $(K \cup \{h, \text{"}yes\text{"}, \text{"}no\text{"}\}) \times \Sigma \times \{\leftarrow, \rightarrow, -\}$. We assume that h (the halting state), "*yes*" (the accepting state), "*no*" (the rejecting state), and the cursors directions $\leftarrow$ for "$left$", $\rightarrow$ for "*right*" and $-$ for "*stay*", are not in $K \cup \Sigma$.

Function δ is also called the "program" of the Turing machine [6]. It specifies for each current state $q \in K$ and current symbol $\sigma \in \Sigma$, a triple $\delta(q, \sigma) = (p, \rho, D)$ [6]. p is the next state, ρ is the symbol to be overwritten on σ, and $D \in \{\leftarrow, \rightarrow, -\}$ is the direction in which the cursor will move [6]. For $\triangleright$ we require that, if for states q and p we have $\delta(q, \triangleright) = (p, \rho, D)$, then $\rho = \triangleright$ and $D = \rightarrow$ [6]. That is, $\triangleright$ always directs the cursor to the right and it is never erased [6].

How is the program start? Initially the state is s [6]. The string is initialized to a $\triangleright$, followed by finitely long string $x \in (\Sigma - \{\sqcup\})^*$ [6]. We say that x is the input of the Turing machine [6]. The cursor is pointing to the first symbol, always a $\triangleright$ [6].

From this initial configuration the machine takes a step according to δ, changing its state, printing a symbol and moving the cursor; then it takes another step, and another [6]. In this process the Turing machine could not continue when it reaches a final state $\{h, \text{"}yes\text{"}, \text{"}no\text{"}\}$ [6]. If this happens, we say the Turing machine has halted [6]. If the state "*yes*" has been reached, we say the machine accepts its input; if "*no*" has been reached, then it rejects its input. If a Turing machine M accepts or rejects a string x, then we write $M(x) = \text{"}yes\text{"}$ or $M(x) = \text{"}no\text{"}$ respectively. If it reaches the halting state h, then 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 [6].

We can define the operation of a Turing machine formally using the notion of configuration. Intuitively, a configuration contains a complete description of the current state of the computation [6].

Definition 2.3. A configuration of M is a triple (q, w, u), where $q \in K$ is a state and w, u are strings in Σ^*. w is the string to the left of the cursor, where its last symbol would be the symbol scanned by the cursor in the state q, and u is the string to the right of the cursor, possibly empty (ε denotes the empty string).

2.2 Complexity classes

The Turing machine has been a useful concept in theory of computing since it was created by Alan Turing in the last century [9]. 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 [6]. 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].

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 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 [1]. Moreover, the set of languages decided by nondeterministic Turing machines within time f is denoted $NTIME(f(n))$ [1]. In case of languages would be decided by deterministic Turing machines within time f is denoted $DTIME(f(n))$ [1]. Then, the $P =? NP$ question would be equivalent to $DTIME(n^k) =? NTIME(n^k)$ where $DTIME(n^k) = \bigcup_{j>0} DTIME(n^j)$ and $NTIME(n^k) = \bigcup_{j>0} NTIME(n^j)$ [1]. There are another major complexity classes such as the $POLYLOGTIME = DTIME((log(n))^k)$ and $NLOGTIME = NTIME(log(n))$ [1].

2.3 The Maximum problem

Definition 2.4. Given an array A of n integer numbers and an integer x, *Maximum* is the problem of deciding whether x is the maximum number in A.

How many comparisons are necessary to determine when some integer is the maximum of an array of n elements? We can easily obtain a upper bound of n comparisons: examine each element of the array in turn and keep track of the largest element seen so far and finally we compare the final result with x [3]. 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 maximum in an array of integers and one final comparison to verify whether that maximum is equal to x or not [3]. Hence, n comparisons are necessary to determine whether an element x is the maximum in A and this naive algorithm for *Maximum* is optimal with respect to the number of comparisons performed [3].

3 Results

3.1 A bounded version of Maximum problem

Definition 3.1. Given an array A of n positive integer numbers and an integer x where $|A| \leq n^2$, *BoundedMaximum* is the problem of deciding whether x is the maximum number in A.

Theorem 3.2. *BoundedMaximum* $\notin$ *POLYLOGTIME*.

Proof. If the pair A and x belongs to *BoundedMaximum*, then the maximum bit-length of x should be less than or equal to $|A|$, because x will be in A. As we see in Definition 2.4, we should use n comparisons to know whether x is the maximum in array of n integers and this number of comparisons will be optimal [3]. This would mean we cannot always accept any instance (A,x) of *BoundedMaximum* in time $(log(|A|))^k$, because we must use at least n comparisons in many cases and it will not exist a constant number k such that $(log(|A|))^k > n$ for every value of n. The reason is $|A| \leq n^2$, and thus, $log(|A|) \leq log(n^2) = 2 \times log(n)$, but n will be exponentially greater than $2 \times log(n)$. □

3.2 NLOGTIME problems

Definition 3.3. Given an array A of n positive integer numbers and an integer x where $|A| \leq n^2$, *IsInArray* is the problem of deciding whether x is in A.

Theorem 3.4. *IsInArray* $\in$ *NLOGTIME*.

Proof. Given an array A of n positive integer numbers and an integer x where $|A| \leq n^2$, we are going to create an algorithm which decides whether x is in A by a nondeterministic Turing machine in time $O(log(|A|))$. For that purpose, we create a nondeterministic Turing machine N such that for the empty string as input, N will output a positive integer i in a nondeterministic way where $0 \leq i \leq (2^{(\lfloor log_2(n) \rfloor + 1)} - 1)$. The program δ of N is built as follows:

(i) we create $k = (\lfloor log_2(n) \rfloor + 1)$ different states p_j in N where $1 \leq j \leq k$ and;

(ii) for each p_j and p_{j+1} states, we create the following actions in N:

$$\delta(p_j, \sqcup) = (p_{j+1}, 0, \longrightarrow) \tag{3.1}$$
$$\delta(p_j, \sqcup) = (p_{j+1}, 1, \longrightarrow) \tag{3.2}$$

(iii) the halting state in N will be related to the *k-th* state p_k with the following actions:

$$\delta(p_k, \sqcup) = (h, 0, -) \tag{3.3}$$
$$\delta(p_k, \sqcup) = (h, 1, -) \tag{3.4}$$

(iv) and finally, the initial state in N will be into a single action:

$$\delta(s, \rhd) = (p_1, \rhd, \longrightarrow). \tag{3.5}$$

After that, an algorithm for *IsInArray* will be very simple:

(1) first, we take the positive integer i as output of the running of N with the empty string where $0 \leq i \leq (2^{(\lfloor log_2(n) \rfloor + 1)} - 1)$;

(2) next, if $i > n$ or $i = 0$, then we reject;

(3) else, we obtain the positive integer $y = A[i]$ using the array indexing;

(4) finally, we accept when x is equal to y else we reject.

The running time of the first until the third step would be $O(log_2(n))$, because the running of N with the empty string does not exceed the $(\lfloor log_2(n) \rfloor + 2)$ steps. The fourth step will use a single comparison with an element in A. In addition, if (A,x) belongs to *IsInArray*, then $|x| \leq |A|$. Hence, we could always accept any instance $(A,x) \in IsInArray$ in time $O(log(|A|))$ using this algorithm, because $n \leq |A|$. Moreover, we could also decide any instance (A,x) in *IsInArray* using $O(log(|A|))$, because we can always accept (A,x) when $(A,x) \in IsInArray$ in that time [3]. Furthermore, for this algorithm we could create an equivalent nondeterministic Turing machine, due to the use of the N nondeterministic Turing machine. Consequently, $IsInArray \in NLOGTIME$. □

Definition 3.5. Given an array A of n positive integer numbers and an integer x where $|A| \leq n^2$, *LessThan* is the problem of deciding whether x complies with $x < y$ for some y in A.

Theorem 3.6. *$LessThan \in NLOGTIME$.*

Proof. The *LessThan* and *IsInArray* share the same kind of instance: an array A of n positive integer numbers and an integer x where $|A| \leq n^2$. Indeed, we could use the same idea of algorithm in Theorem 3.4 for the proof of $LessThan \in NLOGTIME$ in the following way:

(1) first, we take the positive integer i as output of the running of N with the empty string where $0 \leq i \leq (2^{(\lfloor log_2(n) \rfloor + 1)} - 1)$;

(2) next, if $i > n$ or $i = 0$, then we reject;

(3) else, we obtain the positive integer $y = A[i]$ using the array indexing;

(4) finally, we accept when $x < y$ else we reject.

We only changed the fourth step in relation to the algorithm in Theorem 3.4. Indeed, we changed the comparison of equal to by less than. To sum up, $LessThan \in NLOGTIME$, because $IsInArray \in NLOGTIME$. □

3.3 POLYLOGTIME=?NLOGTIME

Definition 3.7. Given an array A of n positive integer numbers and an integer x where $|A| \leq n^2$, $GreaterOrEqual$ is the problem of deciding whether x complies with $x \geq y$ for every element y in A.

Lemma 3.8. *$GreaterOrEqual$ is the complement of $LessThan$ problem.*

Proof. Indeed, the acceptance of some instance (A,x) for $LessThan$ will imply the rejection of (A,x) for $GreaterOrEqual$ and viceversa. This will happen because of the contraposition of " $<$ " and " $\geq$ " operators. □

Theorem 3.9. *$POLYLOGTIME \neq NLOGTIME$.*

Proof. Let's state a hypothesis that has an absurd consequence if it is true.

Hypothesis 3.10. *$POLYLOGTIME = NLOGTIME$.*

Given an array A of n positive integer numbers and an integer x where $|A| \leq n^2$, if x is in A, then the instance (A,x) will be in $BoundedMaximum$ if and only if (A,x) is in $GreaterOrEqual$. In addition, we could verify whether x is in A just in polylogarithmic time by a deterministic Turing machine if $IsInArray \in POLYLOGTIME$. If the Hypothesis 3.10 is true, then $IsInArray \in POLYLOGTIME$. Therefore, it will exist a polylogarithmic time reduction from $BoundedMaximum$ to $GreaterOrEqual$ when the Hypothesis 3.10 is true. We could take any instance (A,x) and verify whether x is in A in polylogarithmic time. In case of x is in A and we would want to know whether (A,x) is in $BoundedMaximum$, then we would only need to check whether (A,x) is in $GreaterOrEqual$.

On the other hand, if the Hypothesis 3.10 is true, then $LessThan \in POLYLOGTIME$. But, if we solve $LessThan$ within a polylogarithmic time using a deterministic Turing machine, then we could solve $GreaterOrEqual$ in polylogarithmic time by the same Turing machine because of Lemma 3.8. Indeed, when a deterministic Turing machine accepts or rejects any instance of $LessThan$, then it would also be rejecting or accepting the same instance for $GreaterOrEqual$ respectively. As result, if the Hypothesis 3.10 is true, then $GreaterOrEqual \in POLYLOGTIME$.

Nevertheless, if $GreaterOrEqual \in POLYLOGTIME$, then $BoundedMaximum \in POLYLOGTIME$, because we could use a polylogarithmic time reduction from $BoundedMaximum$ to $GreaterOrEqual$ through $IsInArray$ over the supposition that the Hypothesis 3.10 is true. But, this is not possible, as we proved in Theorem 3.2, and therefore, the Hypothesis 3.10 is false. In conclusion, $POLYLOGTIME \neq NLOGTIME$ as a direct consequence of using the Reductio ad absurdum rule [8]. □

3.4 P=?NP

Theorem 3.11. *If $P = NP$, then $POLYLOGTIME = NLOGTIME$.*

Proof. Let $L \in NLOGTIME$; under the assumption that $P = NP$, we shall show that $L \in POLYLOGTIME$. By definition, L is decided by a 1-string nondeterministic Turing machine N in time at most $\lceil k \times log(n) \rceil$, for some constant $k > 0$, where $\lceil ... \rceil$ is the ceiling function. Intuitively, we can extend our proof to a multiple string machine if it would be necessary. Consider now the following version of L:

$$L' = \{x : xy \in L \; where \; |x| = \lceil k \times log(|xy|) \rceil \} \tag{3.6}$$

where L' would be all the strings x which are the prefixes of the strings $z \in L$ to bring the total length of $\lceil k \times log(|z|) \rceil$.

We claim that $L' \in NP$. This is easy to show: the polynomial time nondeterministic Turing machine that decides L' is precisely N. Indeed, for any instance $xy \in L$ where $x \in L'$, the program in N will accept the input xy without reaching the string y through the moving of the cursor during the computation, because L is decided by N in time at most $\lceil k \times log(n) \rceil$ and N will start its computation from the initial configuration $(s, \triangleright, xy)$. For that reason, for every instance $x \in L'$, N will accept x in time at most $|x|$.

Since $L' \in NP$, and we are assuming that $P = NP$, we know that $L' \in P$. In addition, $L' \in NTIME(n)$, and thus, we have that $L' \notin DTIME(n)$, because $DTIME(n) \neq NTIME(n)$ [7]. Therefore, there will be a 1-string deterministic Turing machine M' that decides L' in time $n^{k'}$, for some constant $k' > 1$. M' can be a 1-string machine, because every multiple string Turing machine that decides its inputs in polynomial time can be converted to a 1-string machine which decides the same inputs in polynomial time too [6].

We can create a 2-string deterministic Turing machine M such that for each instance $xy \in L$ where $x \in L'$, M will always start in the initial configuration with x in the first string and y in the second. The transition function of M will always simulate M' on its first string while in the second it will always be reading the $\triangleright$ symbol, overwriting $\triangleright$ with the same $\triangleright$ and staying the cursor in that symbol. Since M' could accept $x \in L'$ in time $|x|^{k'}$, then M could accept $xy \in L$ in time $(\lceil k \times log(|xy|) \rceil)^{k'}$.

Certainly, we can affirm that L is decided by M, because we only need to accept $x \in L'$ for the respective acceptance of $xy \in L$, where $|x| = \lceil k \times log(|xy|) \rceil$. Indeed, we can deduce this from the self behavior of N that always accepts $xy \in L$ without taking into consideration the string y during the actions of its computation. Hence, we obtain that $L \in POLYLOGTIME$. □

Theorem 3.12. *$P \neq NP$.*

Proof. As result of Theorem 3.11, we have if $POLYLOGTIME \neq NLOGTIME$, then $P \neq NP$ [8]. However, the Theorem 3.9 states that $POLYLOGTIME \neq NLOGTIME$. Consequently, we obtain that $P \neq NP$. □

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, 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.

Although this demonstration removes the practical computational benefits of a proof that $P = NP$, it would nevertheless 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. 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

I thank Marzio de Biasi for his help with the revision of my previous intents in solving this problem.

References

[1] SANJEEV ARORA AND BOAZ BARAK: *Computational Complexity: A Modern Approach*. Cambridge, 2009. 3

[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] THOMAS H. CORMEN, CHARLES ERIC LEISERSON, RONALD L. RIVEST, AND CLIFFORD STEIN: *Introduction to Algorithms*. MIT Press, 2 edition, 2001. 3, 4, 5

[4] LANCE FORTNOW: The Status of the P versus NP Problem. *Communications of the ACM*, 52(9):78–86, 2009. available at http://www.cs.uchicago.edu/~fortnow/papers/pnp-cacm.pdf. [doi:10.1145/1562164.1562186] 1

[5] WILLIAM I. GASARCH: The P=?NP poll. *SIGACT News*, 33(2):34–47, 2002. available at http://www.cs.umd.edu/~gasarch/papers/poll.pdf. 2

[6] CHRISTOS H. PAPADIMITRIOU: *Computational complexity*. Addison-Wesley, 1994. 2, 3, 7

[7] W. J. PAUL, N. PIPPENGER, E. SZEMERÉDI, AND W. T. TROTTER: On determinism versus nondeterminism and related problems. *Proc. 24th IEEE Symp. on the Foundations of Computer Science*, pp. 429–438, 1983. 7

[8] STEPHEN READ: *Thinking About Logic*. Oxford, 1995. 6, 7

[9] ALAN M. TURING: On Computable Numbers, with an Application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 42:230–265, 1936. 2, 3

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.