HAL
archives-ouvertes.fr

P versus NP

Frank Vega

▶ To cite this version:

Frank Vega. P versus NP. 2016. hal-01316353v2

HAL Id: hal-01316353
https://hal.archives-ouvertes.fr/hal-01316353v2

Submitted on 24 May 2016

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.

P versus NP

Frank Vega
vega.frank@gmail.com

Abstract: P versus NP is one of the most important and unsolved problems in computer science. This consists on 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. Since that date, all efforts to find a proof for this huge problem have failed. It is currently accepted that a positive answer for P versus NP would have tremendous effects not only in computer science, but also in mathematics, biology, and so forth. This work is about an interesting class of problems whose status is unknown: the complexity class NP-complete. If any single NP-complete problem is in P, then P is equal to NP. Indeed, we show that a known NP-complete problem belongs to P, and therefore, P = NP.

Key Words: P, NP, NL, verifier
Category: F.1.3

Introduction

P versus NP is a major unsolved problem in computer science [1]. 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 [1]. 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 [1].

In 1936, Turing developed his theoretical computational model [2]. 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 [3]. 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 [3].

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 [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 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 on 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 [6].

The *NP–complete* is an interesting complexity class defined by Cook in his seminal paper [2]. The class *NP-complete* is a set of problems of which any other NP problem can be reduced in polynomial time, but whose solution may still be verified in polynomial time [5]. If any single *NP–complete* problem can be solved in polynomial time, then every NP problem has a polynomial time algorithm [4].

Another major complexity class is NL. NL is the class of languages that are decidable on a nondeterministic logspace machine [7]. A logspace machine is a Turing machine with a read-only input tape, a write-only output tape, and a read/write work tapes [7]. The work tapes may contain at most $O(\log n)$ symbols [7]. It is known that $NL \subseteq P \subseteq NP$ [3]. Whether $NL = P$ is another fundamental question that it is as important as it is unresolved [3]. All efforts to find polynomial time algorithms for the *NP–complete* problems have failed [1]. Nevertheless, we prove there exists an *NP–complete* in NL, and therefore, $P = NP$.

1 Theoretical notions

Let Σ be a finite alphabet with at least two elements, and let Σ^* be the set of finite strings over Σ [2]. A Turing machine M has an associated input alphabet Σ [2]. For each string w in Σ^* there is a computation associated with M on input w [2]. We say that M accepts w if this computation terminates in the accepting state [2]. Note that M fails to accept w either if this computation ends in the rejecting state, or if the computation fails to terminate [2].

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

$$L(M) = \{w \in \Sigma^* : M \ \textit{accepts}\ w\}.$$

We denote by $t_M(w)$ the number of steps in the computation of M on input w [2]. 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 [2]. We say that M runs in polynomial time if there exists k such that for all n, $T_M(n) \leq n^k + k$ [2].

Definition 1. A language L is in class P if $L = L(M)$ for some deterministic Turing machine M which runs in polynomial time [2].

We state the complexity class NP using the following definition.

Definition 2. A verifier for a language L is a deterministic Turing machine M, where

$$L = \{w : M \textit{ accepts } \langle w, c\rangle \textit{ for some string } c\}.$$

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.

Definition 3. NP is the class of languages that have polynomial time verifiers [7].

An important complexity class is *NP–complete* [5]. If any single *NP–complete* problem can be solved in polynomial time, then every NP problem has a polynomial time algorithm [4]. No polynomial time algorithm has yet been discovered for an *NP–complete* problem [1].

A principal *NP–complete* problem is $CLIQUE$ [8]. An instance of $CLIQUE$ is an undirected graph $G = (V, E)$, where V is a finite set and E is a binary relation on V [4]. The set V is called the vertex set of G, and its elements are called vertices or nodes [4]. The set E is called the edge set of G, and its elements are called edges [4].

A clique in an undirected graph $G = (V, E)$ is a subset $V' \subseteq V$ of vertices, each pair of which is connected by an edge in E [4]. A clique is a complete subgraph of G [4]. The size of a clique is the number of vertices it contains. The formal definition of this problem is

$$CLIQUE = \{(G, k) : G \textit{ is a graph with a clique of size } k\}.$$

Checking whether V' is a clique in a graph $G = (V, E)$ can be accomplished in polynomial time by checking whether, for every pair $u, v \in V'$, the edge (u, v) belongs to E.

2 P =? NP

We can give a certificate-based definition for NL [9]. The certificate-based definition of NL assumes that a logspace machine has another separated read-only tape [9]. On each step of the machine the machine's head on that tape can either stay in place or move to the right [9]. In particular, it cannot reread any bit to the left of where the head currently is [9]. For that reason this kind of special tape is called "read once" [9].

Definition 4. A language L is in NL if there exists a deterministic logspace machine and a with an additional special read-once input tape polynomial $p : \mathbb{N} \to \mathbb{N}$ such that for every $x \in \{0,1\}^*$,

$$x \in L \Leftrightarrow \exists u \in \{0,1\}^{p(|x|)} \textit{ such that } M \textit{ accepts } \langle x, u \rangle$$

where by $M(x,u)$ we denote the computation of M where x is placed on its input tape and u is placed on its special read-once tape, and M uses at most $O(\log |x|)$ space on its read/write tapes for every input x.

Theorem 5. $P = NP$.

Proof. We will show that a certificate u which represents a clique V' of size k on a graph G can be verified by a deterministic logspace machine M using an additional special read-once input tape, and thereby $CLIQUE$ will be in NL as a direct consequence of the Definition 4. The logarithmic space algorithm M that we present for the verification in $CLIQUE$ must accept whenever the input graph G contains a clique of size k using the certificate u.

Let c_k be an array of natural numbers that contains the nodes in the clique V' of size k on a graph G. We assume the c_k is provided in ascending order where in G every vertex $v \in V$ is mapped to a single and unique integer between 1 and $|V|$ where $|\ldots|$ denotes the cardinality function and V is the vertex set of G.

Given G, k and c_k, the machine M operates as follows. With the first node in c_k, M checks one by one whether this node has an edge with the others elements in the array. Whenever M checks whether those edges exist, M attempts to verify whether the array c_k is sorted in ascending way by storing the element that was verified in last step for the comparison to the next one in the following step. If these verifications fail, M rejects. In addition, M counts the number of nodes that contains c_k and if the number of elements in c_k is different to k, it rejects too. In the meantime, M calculates $c_k[1] \oplus c_k[2] \oplus c_k[3] \oplus \ldots \oplus c_k[k]$ where $\oplus$ represents the XOR function [10]. We will denote the value of this calculation as the binary integer b_k. We can make this evaluation in a forward way, that is, calculating first $c_k[1] \oplus c_k[2] = B$ and later $B \oplus c_k[3] = C$, and so on ... until we reach the last *k-th* element. Besides, we check whether each node is between 1 and $|V|$.

Certainly, we can make this whole computation reading at once into the array c_k. In other words, M can verify that the smallest node in V' has an edge with each one of the others vertices in V' using the array c_k as the first part of the certificate u on the additional read once tape.

After we store the values of $c_k[1]$ and b_k on the read/write tapes, then we continue with the second part of the certificate u, that is another array $c_{(k-1)}$ of natural numbers that contains the nodes in the clique V' but without the

node represented by $c_k[1]$. But now, $c_{(k-1)}$ is sorted in descending way. Then, we take the first element in $c_{(k-1)}$ and check whether each element starting from the second in $c_{(k-1)}$ has an edge with this one. If this verification fails, M rejects. In the same way, M rejects if the number of elements in $c_{(k-1)}$ is not $(k-1)$ and if they are not in a descending order. At the same time, M calculates $c_{(k-1)}[1] \oplus c_{(k-1)}[2] \oplus \ldots \oplus c_{(k-1)}[(k-1)]$. We also denote this new value as $b_{(k-1)}$. The purpose of this evaluation is to compare the result of $b_k \oplus b_{(k-1)}$ with $c_k[1]$ (the first element of c_k). In case these values were not equal, then M rejects, because this would mean $c_{(k-1)}$ will contain at least one different node which is not in c_k. This is supported by the properties of the function XOR which are quite used for cipher [10]. Likewise, we check whether each node is between 1 and $|V|$.

Certainly, these properties in the array $c_{(k-1)}$ can be verified through a reading at once on the special tape that contains the certificate u. In other words, M can verify that the largest node in V' has an edge with each one of the others vertices in V' (ignoring the vertex $c_k[1]$ since the edge $(c_k[1], c_{(k-1)}[1])$ was already checked) using the array $c_{(k-1)}$ as the second part of the certificate u on the additional read once tape.

Similarly, the array $c_{(k-2)}$ has the nodes in V' but without the previous $c_k[1]$ and $c_{(k-1)}[1]$. However, it is sorted in ascending way again. Consequently, M will make the same verifications that has been done on the previous arrays of u, but M uses on this time the last values of $c_{(k-1)}[1]$ and $b_{(k-1)}$ on its read/write tapes to check whether $b_{(k-1)} \oplus b_{(k-2)}$ is equal to $c_{(k-1)}[1]$.

To sum up, the certificate u will be a two dimensional array of integers between 1 and $|V|$ such that $u[1] = c_k$, $u[2] = c_{(k-1)}$, $u[3] = c_{(k-2)}$ and so on ... until the last array $c[k] = c_1$. For each i between 1 and k, the array $u[i]$ will be sorted in ascending way if i is odd and descending otherwise. Furthermore, every array $u[i]$ in u must have exactly $(k+1) - i$ elements.

Here is the algorithm for the verification of $(G, k) \in CLIQUE$ through the described certificate u. Let m be the number of nodes of G:

$$M = \textit{ On input } (G, k, u)$$

1. Let *previous* := 0
2. Let *previous-xor-value* := 0
3. For $i := 1$ to k
4. {
5. Let *current* := $u[i][1]$
6. Let *prior* := *current*

7. Let *current-xor-value* := *current*
8. Let *parity* := (*i* *mod* 2)
9. if (*current* is not a binary integer between 1 and m)
10. {
11. *reject*
12. }
13. *previous-xor-value* := *previous-xor-value* $\oplus$ *current*
14. For $j := 2$ to $(k + 1) - i$
15. {
16. Let *next* := $u[i][j]$
17. if ((*next* is a binary integer between 1 and m)
18. and ((*parity* = 0 and *next* < *prior*)
19. or (*parity* = 1 and *next* > *prior*))
20. and ((*current*, *next*) $\in E$))
21. {
22. *prior* := *next*
23. *current-xor-value* := *current-xor-value* $\oplus$ *next*
24. *previous-xor-value* := *previous-xor-value* $\oplus$ *next*
25. }
26. else
27. {
28. *reject*
29. }
30. }
31. if (($u[i][(k + 2) - i]$ is undefined in u)
32. and ((*previous* = 0)

33. or (*previous* = *previous-xor-value*)))
34. {
35. *previous* := *current*
36. *previous-xor-value* := *current-xor-value*
37. }
38. else
39. {
40. *reject*
41. }
42. }
43. *accept*

This algorithm only needs to store i, j, *previous*, *current*, *previous-xor-value*, *prior*, *current-xor-value*, *next* and *parity* at any given time. But, each one of these variables only uses a $\log m$ amount of space, since we can represent them as binary strings. Hence, M verifies (G, k) using the certificate u in logarithmic space. In addition, in this algorithm we never go backward on the contiguous arrays of integers in u. Furthermore, it is quite obvious that u is polynomially bounded by (G, k), since the size u is lesser than $k \times (2 \times \log m) \times m^2$. For all these reasons, we can support that $CLIQUE \in NL$. Since $NL \subseteq P$, then $CLIQUE \in P$ [3]. If any single *NP–complete* problem can be solved in polynomial time, then every NP problem has a polynomial time algorithm [4]. Consequently, we can conclude that $P = NP$.

Conclusions

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 [8]. Even though this proof might not be a practical solution, it shows in a formal way that many currently mathematical problems can be solved efficiently, including those in *NP–complete*.

At the same time, this demonstration would represent a very significant advance in computational complexity theory and provide guidance for future research. On the one hand, it proves that most of the existing cryptosystems such as the public-key cryptography are not safe [7]. On the other hand, we will be able to find a formal proof for every theorem which has a proof of a reasonable length by a feasible algorithm.

References

1. L. Fortnow, The Golden Ticket: P, NP, and the Search for the Impossible, Princeton University Press. Princeton, NJ, 2013.
2. S. A. Cook, The P versus NP Problem, available at `http://www.claymath.org/sites/default/files/pvsnp.pdf` (April 2000).
3. C. H. Papadimitriou, Computational Complexity, Addison-Wesley, 1994.
4. T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, Introduction to Algorithms, 2nd Edition, MIT Press, 2001.
5. O. Goldreich, P, Np, and Np-Completeness, Cambridge: Cambridge University Press, 2010.
6. W. I. Gasarch, The P=?NP poll, SIGACT News 33 (2) (2002) 34–47.
7. M. Sipser, Introduction to the Theory of Computation, 2nd Edition, Thomson Course Technology, 2006.
8. M. R. Garey, D. S. Johnson, Computers and Intractability: A Guide to the Theory of NP-Completeness, 1st Edition, San Francisco: W. H. Freeman and Company, 1979.
9. S. Arora, B. Barak, Computational complexity: A modern approach, Cambridge University Press, 2009.
10. R. Churchhouse, Codes and Ciphers: Julius Caesar, the Enigma and the Internet, Cambridge University Press, 2002.