Open Peer Review on Qeios

NP on Logarithmic Space

Frank Vega

Funding: No specific funding was received for this work.
Potential competing interests: No potential competing interests to declare.

Abstract

P versus NP is considered as one of the most important open problems in computer science. This consists in knowing the answer of the following question: Is P equal to NP? It was essentially mentioned in 1955 from a letter written by John Nash to the United States National Security Agency. However, a precise statement of the P versus NP problem was introduced independently by Stephen Cook and Leonid Levin. Since that date, all efforts to find a proof for this problem have failed. Another major complexity classes are L and NL. Whether L = NL is another fundamental question that it is as important as it is unresolved. We prove that NL = NP just using 1L-reductions and thus, we show that P = NP.

Frank Vega
NataSquad, 10 rue de la Paix 75002 Paris, France
vega.frank@gmail.com
https://orcid.org/0000-0001-8210-4126

2012 ACM Subject Classification: Theory of computation → Complexity classes; Theory of computation → Problems, reductions and completeness; Theory of computation → Abstract machines.

Keywords: Complexity Classes, Completeness, Polynomial Time, Reduction, Logarithmic Space.

1. Introduction

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 [1]. A deterministic Turing machine has only one next action for each step defined in its program or transition function [1]. 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 [1].

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, that is $M(w) =$ "yes" [2]. Note that, M fails to accept w either if this computation ends in the rejecting state, that is $M(w) =$ "no", or if the computation fails to terminate, or the computation ends in the halting state with some output, that is $M(w) = y$ (when M outputs the string y on the input w) [2].

Another relevant 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 [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]. The language accepted by a Turing machine M, denoted $L(M)$, has an associated alphabet Σ and is defined by:

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

Moreover, $L(M)$ is decided by M, when $w \notin L(M)$ if and only if $M(w) =$ "no" [3]. 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 is a constant k such that for all n, $T_M(n) \leq n^k + k$ [2]. In other words, this means the language $L(M)$ can be decided by the Turing machine M in polynomial time. Therefore, P is the complexity class of languages that can be decided by deterministic Turing machines in polynomial time [3]. A verifier for a language L_1 is a deterministic Turing machine M, where:

$$L_1 = \{w : M(w, u) = \text{"yes" for some string } u\}.$$

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 [2]. A verifier uses additional information, represented by the string u, to verify that a string w is a member of L_1. This information is called certificate. NP is the complexity class of languages defined by polynomial time verifiers[4].

It is fully expected that $P \neq NP$ [4]. Indeed, if $P = NP$ then there are stunning practical consequences [4]. For that reason, $P = NP$ is considered as a very unlikely event [4]. Certainly, P versus NP is one of the greatest open problems in science and a correct solution for this incognita will have a great impact not only in computer science, but for many other fields as well [5]. Whether $P = NP$ or not is still a controversial and unsolved problem [6]. We provide a final answer for this outstanding problem using the logarithmic space complexity.

1.1. The Hypothesis

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

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

An important complexity class is NP—complete [7]. If L_1 is a language such that $L' \leq_p L_1$ for some $L' \in$ NP—complete, then L_1 is NP—hard [3]. Moreover, if $L_1 \in NP$, then $L_1 \in$ NP—complete [3]. A principal NP—complete problem is SAT [7]. An instance of SAT is a Boolean formula ϕ which is composed of:

1. Boolean variables: $x_1, x_2, \ldots, x_n$;
2. Boolean connectives: Any Boolean function with one or two inputs and one output, such as $\wedge$ (AND), $\vee$ (OR), $\neg$ (NOT), $\Rightarrow$ (implication), $\Leftrightarrow$ (if and only if);
3. and parentheses.

A truth assignment for a Boolean formula ϕ is a set of values for the variables in ϕ. A satisfying truth assignment is a truth assignment that causes ϕ to be evaluated as true. A Boolean formula with a satisfying truth assignment is satisfiable. The problem SAT asks whether a given Boolean formula is satisfiable[7]. We define a *CNF* Boolean formula using the following terms:

A literal in a Boolean formula is an occurrence of a variable or its negation[3]. 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[3]. A Boolean formula is in 3-conjunctive normal form or 3*CNF*, if each clause has exactly three distinct literals[3]. For example, the Boolean formula:

$$(x_1 \vee \neg x_1 \vee \neg x_2) \wedge (x_3 \vee x_2 \vee x_4) \wedge (\neg x_1 \vee \neg x_3 \vee \neg x_4)$$

is in 3*CNF*. The first of its three clauses is $(x_1 \vee \neg x_1 \vee \neg x_2)$, which contains the three literals x_1, $\neg x_1$, and $\neg x_2$.

A logarithmic space Turing machine has a read-only input tape, a write-only output tape, and read/write work tapes[1]. The work tapes may contain at most $O(\log n)$ symbols [1]. In computational complexity theory, L is the complexity class containing those decision problems that can be decided by a deterministic logarithmic space Turing machine [4]. NL is the complexity class containing the decision problems that can be decided by a nondeterministic logarithmic space Turing machine [4].

A function $f: \Sigma^* \rightarrow \Sigma^*$ is a logarithmic space computable function if some deterministic Turing machine M, on every input w, halts using logarithmic space in its work tapes with just $f(w)$ on its output tape [1]. Let $\{0,1\}^*$ be the infinite set of binary strings, we say that a language $L_1 \subseteq \{0,1\}^*$ is logarithmic space reducible to a language $L_2 \subseteq \{0,1\}^*$, written $L_1 \leq_l L_2$, if there is a logarithmic space computable function $f: \{0,1\}^* \rightarrow \{0,1\}^*$ such that for all $x \in \{0,1\}^*$:

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

The logarithmic space reduction is used for the completeness of the complexity classes L, NL and P among others.

The two-way Turing machines may move their head on the input tape into two-way (left and right directions) while the one-way Turing machines are not allowed to move the input head on the input tape to the left [8]. Hartmanis and Mahaney have investigated the classes $1L$ and $1NL$ of languages recognizable by deterministic one-way logarithmic space Turing machine and nondeterministic one-way logarithmic space Turing machine, respectively [8]. They have shown that $1L \neq 1NL$ (by looking at a uniform variant of the string non-equality problem from communication complexity theory) and have defined a natural complete problem for $1NL$ under deterministic one-way logarithmic space reductions[8]. Furthermore, they have proven that $1NL \subseteq L$ if and only if $L = NL$ [8].

We can give a certificate-based definition for NL [2]. The certificate-based definition of NL assumes that a logarithmic space Turing machine has another separated read-only tape, that is called ''read-once '' , where the head never moves to the left on that special tape [2].

Definition1. *A language L_1 is in NL if there exists a deterministic logarithmic space Turing machine M with an additional special read-once input tape polynomial $p : \mathbb{N} \to \mathbb{N}$ such that for every $x \in \{0, 1\}^*$:*

$$x \in L_1 \Leftrightarrow \exists u \in \{0, 1\}^{p(|x|)} \text{ then } M(x, u) = \text{"yes"}$$

where by $M(x, u)$ we denote the computation of M, x is placed on its input tape, the certificate string 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 where $|\ldots|$ is the bit-length function. The Turing machine M is called a logarithmic space verifier.

An oracle Turing Machine M has an additional tape, the oracle tape, and three states $q_?$, q_{yes} and q_{no} [9]. When M enters $q_?$ (M is said to query the oracle), then M goes to the state q_{yes} or the state q_{no} according to whether the string written in the oracle tape belongs or does not belong to a set called the oracle [9]. A language accepted by an oracle Turing Machine M with oracle A is denoted by $L^A(M)$ [9]. The class of languages accepted by deterministic and nondeterministic oracle Turing Machine M working in space $S(n)$, with oracle A, is denoted by $DSPACE^A(S(n))$ and $NSPACE^A(S(n))$, respectively [9]. In this definition, we bound the space of the oracle tape by a space $2^{O(S(n))}$ [9]. A nondeterministic oracle Turing machine can query $2^{2^{O(S(n))}}$ strings in the tree of all possible computations[9]. There is another definition such that the oracle tape is not space-bounded and the machine works deterministically from the time it begins to write on the oracle tape [9]. The complexity classes $DSPACE^{\langle A \rangle}(S(n))$ and $NSPACE^{\langle A \rangle}(S(n))$ are the respective complexity classes based on this definition on an oracle A [9]. It is trivial to see that $DSPACE^{\langle A \rangle}(S(n)) = DSPACE^A(S(n))$ [9].

We state the following Hypothesis:

Hypothesis 1. *There is a nonempty language $L_2 \in 1L$, such that there is another language L_3 which is closed under logarithm space reductions in* NP—complete *with a deterministic logarithmic space Turing machine M using an additional special read-once input tape polynomial $p : \mathbb{N} \to \mathbb{N}$, where:*

$$L_3 = \{w : M(w, u) = y, \exists u \in \{0, 1\}^{p(|w|)} \text{ such that } y \in L_2\}$$

when by $M(w, u)$ we denote the computation of M, w is placed on its input tape, and the certificate string u is placed on the special read-once tape of M. In this way, there is a NP—complete *language defined by a logarithmic space verifier M such that when the input is an element of the language, then there exists a certificate u such that M outputs a string which belongs to a single language in $1L$.*

We show the principal consequences of this Hypothesis:

Theorem 2. *If the Hypothesis 1 is true, then $NL = NP$.*

Proof. We can simulate the computation $M(w, u) = y$ in the Hypothesis 1 by a nondeterministic logarithmic space oracle Turing machine N such that the string y is written in the oracle tape in the computation of $N(w)$, since we can read the certificate string u within the read-once tape by a work tape in a nondeterministic logarithmic space generation of symbols contained in u [4]. Certainly, we can simulate the reading of one symbol from the string u into the read-once tape just nondeterministically generating the same symbol in the work tapes using a logarithmic space [4]. We could remove each symbol or a logarithmic amount of symbols generated in the work tapes, when we try to generate the next symbol contiguous to the right on the string u. In this way, the generation will always be in logarithmic space. This proves that L_3 is in NL^{1L} since the string y written in the oracle tape is queried whether $y \in L_2$ or not. That is equivalent to say that L_3 is in NL when the Hypothesis 1 is true, since $NL^{1L} = NL$ as a consequence of that NL is closed under 1L—reductions [9]. Due to L_3 is closed under logarithm space reductions in NP—complete, then every NP problem is logarithmic space reduced to L_3. This implies that $NL = NP$ since NL is closed under logarithm space reductions as well. □

1.2. The Problems

We define the problems that we used.

Definition 3. ONE-IN-THREE 3-TIMES 3SAT (3T-3SAT)
INSTANCE: A Boolean formula ϕ in CNF such that each variable appears exactly thrice with two positive and one negated literal occurrences.
QUESTION: Is there a truth assignment such that each clause has at least one true literal?

Definition 4. 1SUM
INSTANCE: A positive integer K and a collection B of positive real numbers such that K has a logarithmic size in relation to the bit-length of B and every element in B has the same bit-length $|K|$ where $|\ldots|$ means the bit-length function.
QUESTION: Is the sum of all elements of the collection B equal to K?

2. Results

Theorem 5. 3T-3SAT $\in$ NP—complete.

Proof. Consider the problem ONE-IN-THREE 3SAT with no negated literals. The question is the same as 3T-3SAT but the instances in ONE-IN-THREE 3SAT are Boolean formulas in $3CNF$. Besides, we know that ONE-IN-THREE 3SAT $\in$ NP—complete [7]. Consider an instance ψ of ONE-IN-THREE 3SAT in which the variable x appears k times. So, we replace the first occurrence of x by x_1, the second by x_2 and so on, where $x_1, x_2, \ldots, x_k$ are k new variables. Next, we add the expression

$$(\neg x_1 \vee x_2) \wedge (\neg x_2 \vee x_3) \wedge \ldots \wedge (\neg x_k \vee x_1)$$

which is equivalent to

$$x_1 \Rightarrow x_2 \Rightarrow \ldots \Rightarrow x_1.$$

Note that, each clause above has fewer than 3 literals. The final result satisfies the condition on the selected variable x. Suppose we are given in ψ the expression

$$\ldots(u \vee v \vee x)\ldots \wedge \ldots(x \vee y \vee z)\ldots$$

and therefore, the transformed expression into another Boolean formula ϕ would be

$$\ldots(u \vee v \vee x_1)\ldots \wedge \ldots(x_2 \vee y \vee z)\ldots \wedge \ldots(\neg x_1 \vee x_2) \wedge (\neg x_2 \vee x_3) \wedge \ldots$$

where the variable x_2 appears exactly thrice since x_2 appears twice and finally $\neg x_2$ appears once. Hence, we proceed with this subroutine for each variable in ψ in ONE-IN-THREE 3SAT to finally obtain the equivalent instance ϕ in 3T-3SAT. $\square$

Theorem 6. 1SUM $\in$ 1L.

Proof. Given an collection of positive real numbers B, we can read its elements from left to right, check that every element in B has the same bit-length $|K|$, sum them one by one into a single value, measure the bit-length of B to compare this number by K and compare the calculated value whether it is equal to K. We can make all this computation in a deterministic one-way using logarithmic space. Certainly, the comparison between K and the bit-length of B could be done in logarithmic space since K is a unique value. On the one hand, we can count and store the number of bits of each element of the collection that we read from the input and check whether they are all equal to the unique bit-length $|K|$. Indeed, we never need to read to the left on the input for the acceptance of the elements in $1SUM$ in a deterministic logarithmic space. $\square$

Algorithm 1 Logarithmic space verifier with output

```
/*A valid instance for $3T$-$3SAT$ with its certificate*/
procedure $VERIFIER(\phi, A)$
    /*Initialize the number of variables*/
    $n \leftarrow number\text{–}of\text{–}variables(\phi)$
    /*Output the value $K$*/
    output $\frac{n\cdot(n+1)}{2}$
    /*Initialize the number of clauses*/
    $m \leftarrow number\text{–}of\text{–}clauses(\phi)$
    /*Output the open square bracket of collection $B$*/
    output , [
    /*Iterate for the elements of the certificate array $A$*/
    for $i \leftarrow 1$ to $m$ do
        /*Assign the current index*/
        $j \leftarrow A[i]$
        if $j = null$ then
            return "$no$"
        else if $j > number\text{–}of\text{–}literals(c_i) \vee j < 1$ then
            return "$no$"
        else if $i = m \wedge A[i+1] \neq null$ then
            return "$no$"
        else if $c_i[j] < 0$ then
            /*We fill it by zeroes until the bit-length of $K$*/
            output $fill\text{–}by\text{–}zeroes(-c_i[j])$
        else
            /*We fill it by zeroes until the bit-length of $K$*/
            output $fill\text{–}by\text{–}zeroes(\frac{c_i[j]}{2})$
        end if
        if $i < m$ then
            output ,
        else
            /*Output the close square bracket of collection $B$*/
            output ]
        end if
    end for
end procedure
```

Theorem 7. *There is a deterministic logarithmic space Turing machine M, where:*

$$\text{3T-3SAT} = \{w : M(w,u) = y, \exists u \text{ such that } y \in \text{1SUM}\}$$

when by $M(w,u)$ we denote the computation of M, w is placed on its input tape, u is placed on the special read-once tape *of M, and u is polynomially bounded by w.*

Proof. The input could be a Boolean formula ϕ in *CNF* such that each variable appears exactly thrice with two positive and one negated literal occurrences. The Boolean formula ϕ contain n variables and m clauses. We can create a certificate array A which contains indexes values that represents the position of exactly one literal per clause. We read at once the indexes values of the array A and we reject when this index is out of range in relation to the clause in the i^{th} position. Besides, we check that the array contains exactly m element: one index per clause. While we read the indexes values of the array A using every position i, we check those constraints in A and output half of the number that represent the positive literals and the absolute value for negated literals just assuming the literals are defined as integer according to the DIMACS files representation as input (http://www.satcompetition.org/2009/format-benchmarks2009.html).

By Theorems 5 and 6, we obtain that for all:

$$\phi \in \text{3T-3SAT} \Leftrightarrow \exists A \text{ such that } (K, B) \in \text{1SUM}$$

with the output of real numbers B that sums $K = \frac{n \cdot (n+1)}{2}$ when we guarantee do not output the positive and negated literal of a single variable and we indeed do output one single value for each clause. To sum up, we can create this verifier that only uses a logarithmic space in the work tapes such that the array A is placed on the special read-once tape, because we read at once the indexes values in the array A. Hence, we only need to iterate from the cells of the array A to verify whether the array is an appropriated certificate according to the described constraints and check that every index j is correct.

This logarithmic space verifier with output will be the Algorithm 1. We introduce some constraints in the Algorithm 1 in order to guarantee the algorithmic procedure. For example, we assume that a value does not exist in the array A into a cell of some position i when $A[i] = \text{null}$. In addition, we immediately reject when the mentioned comparisons between the indexes values j and the size of the clause do not hold at least into one single binary digit. That means the machine enters into the rejecting state when the certificate is not valid. Note that, we assume the variables are between 1 and n due to the DIMACS files representation as input (http://www.satcompetition.org/2009/format-benchmarks2009.html). □

Theorem 8. *NL* = *NP*.

Proof. This is a directed consequence of Theorems 2 and 7 because of the Hypothesis 1 is true. Certainly, 3T-3SAT is closed under logarithm space reductions in NP—complete. Indeed, we can reduced SAT to 3T-3SAT in logarithmic space and every *NP* problem could be logarithmic space reduced to SAT by the Cook's Theorem Algorithm [7]. □

3. Conclusions

No one has been able to find a polynomial time algorithm for any of more than 300 important known NP—complete problems [7]. A proof of $P = NP$ will have stunning practical consequences, because it possibly leads to efficient methods for solving some of the important problems in NP [3]. The consequences, both positive and negative, arise since various

NP—complete problems are fundamental in many fields[10]. This work theoretically proves that $P = NP$ since $NL \subseteq P$ and so, we should seriously take into account these positive and negative consequences[4].

Cryptography, for example, relies on certain problems being difficult. A constructive and efficient solution to an NP—complete problem such as SAT will break most existing cryptosystems including: Public-key cryptography, symmetric ciphers and one-way functions used in cryptographic hashing. These would need to be modified or replaced by information-theoretically secure solutions not inherently based on P—NP equivalence.

There are positive consequences that will follow from rendering tractable many currently mathematically intractable problems. For instance, many problems in operations research are NP—complete, such as some types of integer programming and the traveling salesman problem [10]. Efficient solutions to these problems have enormous implications for logistics [10]. Many other important problems, such as some problems in protein structure prediction, are also NP—complete, so this will spur considerable advances in biology[11].

Since all the NP—complete optimization problems become easy, everything will be much more efficient[10]. Transportation of all forms will be scheduled optimally to move people and goods around quicker and cheaper [10]. Manufacturers can improve their production to increase speed and create less waste [10]. Learning becomes easy by using the principle of Occam's razor: We simply find the smallest program consistent with the data [10]. Near perfect vision recognition, language comprehension and translation and all other learning tasks become trivial [10]. We will also have much better predictions of weather and earthquakes and other natural phenomenon [10].

But such changes may pale in significance compared to the revolution an efficient method for solving NP—complete problems will cause in mathematics itself[6]. Research mathematicians spend their careers trying to prove theorems, and some proofs have taken decades or even centuries to find after problems have been stated [6]. For instance, Fermat's Last Theorem took over three centuries to be proved [6]. A method that guarantees to find proofs for theorems, should one exist of a “reasonable” size, would essentially end this struggle [6].

References

1. a, b, c, d, e, f, g, h *Michael Sipser. Introduction to the Theory of Computation, volume 2. Thomson Course Technology Boston, USA, 2006.*
2. a, b, c, d, e, f, g, h, i, j, k *Sanjeev Arora and Boaz Barak. Computational complexity: a modern approach. Cambridge University Press, USA, 2009.*
3. a, b, c, d, e, f, g, h, i, j *Thomas Cormen, Charles Eric Leiserson, Ronald Linn Rivest, and Clifford Stein. Introduction to Algorithms. The MIT Press, USA, 3rd edition, 2009.*
4. a, b, c, d, e, f, g, h, i *Christos Harilaos Papadimitriou. Computational complexity. Addison-Wesley, USA, 1994.*
5. ^ *Stephen Arthur Cook. The P versus NP Problem. http://www.claymath.org/sites/default/files/pvsnp.pdf, April 2000. Clay Mathematics Institute. Accessed 9 January 2023.*
6. a, b, c, d, e *Scott Aaronson. P ? NP. Electronic Colloquium on Computational Complexity, Report No. 4, 2017.*

7. [a, b, c, d, e, f] *Michael Randolph Gare and David Stifler Johnson. Computers and Intractability: A Guide to the Theory of NP-Completeness. San Francisco: W. H. Freeman and Company, USA, 1 edition, 1979.*
8. [a, b, c, d] *Juris Hartmanis and Stephen Ross Mahaney. Languages Simultaneously Complete for One-Way and Two-Way Log-Tape automata. SIAM Journal on Computing, 10(2):383–390, 1981. doi:10.1137/0210027.*
9. [a, b, c, d, e, f, g, h, i, j] *Pascal Michel. A survey of space complexity. Theoretical computer science, 101(1):99–132, 1992. doi:10.1016/0304-3975(92)90151-5.*
10. [a, b, c, d, e, f, g, h, i] *Lance Fortnow. The status of the P versus NP problem. Communications of the ACM, 52(9):78–86, 2009.*
11. [^] *Bonnie Berger and Tom Leighton. Protein folding in the hydrophobic-hydrophilic (hp) model is np-complete. Journal of computational biology: a journal of computational molecular cell biology, 5(1):27–40, 1998.*