NP on Logarithmic Space

Frank Vega

January 8, 2023

NP on Logarithmic Space

Frank Vega
CopSonic, 1471 Route de Saint-Nauphary 82000 Montauban, France

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 the breakthrough result that L = NL. Besides, we show that every NP problem is in L with oracle access to L.

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

Keywords and phrases Complexity Classes, Completeness, Polynomial Time, Reduction, Logarithmic Space

1 Introduction

In 1936, Turing developed his theoretical computational model [11]. The deterministic and nondeterministic Turing machines have become in two of the most important definitions related to this theoretical model for computation [11]. A deterministic Turing machine has only one next action for each step defined in its program or transition function [11]. 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 [11].

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

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

Moreover, $L(M)$ is decided by M, when $w \notin L(M)$ if and only if $M(w) =$ "*no*" [4]. 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 [4]. A verifier for a language L_1 is a deterministic Turing machine M, where:

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

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 [9].

It is fully expected that $P \neq NP$ [9]. Indeed, if $P = NP$ then there are stunning practical consequences [9]. For that reason, $P = NP$ is considered as a very unlikely event [9]. 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 [3]. Whether $P = NP$ or not is still a controversial and unsolved problem [1]. We provide some results in order to understand better this outstanding problem in computer science.

1.1 The Hypothesis

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 [11]. 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 $x \in \{0,1\}^*$:

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

An important complexity class is *NP–complete* [5]. 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* [4]. Moreover, if $L_1 \in NP$, then $L_1 \in$ *NP–complete* [4]. A principal *NP–complete* problem is SAT [5]. 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 [5]. 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 [4]. 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 [4]. A Boolean formula is in 2-conjunctive normal form or $2CNF$, if each clause has exactly two distinct literals [4]. For example, the Boolean formula:

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

is in $2CNF$. The first of its three clauses is $(x_1 \vee \rightharpoondown x_2)$, which contains the two literals x_1, and $\rightharpoondown x_2$.

A logarithmic space Turing machine has a read-only input tape, a write-only output tape, and read/write work tapes [11]. The work tapes may contain at most $O(\log n)$ symbols [11]. In computational complexity theory, L is the complexity class containing those decision problems that can be decided by a deterministic logarithmic space Turing machine [9]. NL is the complexity class containing the decision problems that can be decided by a nondeterministic logarithmic space Turing machine [9].

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 [11]. 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 $x \in \{0,1\}^*$:

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

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 [6]. 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 [6]. 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 [6]. Furthermore, they have proven that $1NL \subseteq L$ if and only if $L = NL$ [6]. The complexity class $co1NL$ can be defined as the set of languages such that every element inside of the language will be accepted for every possible path by a nondeterministic one-way logarithmic space Turing machine [9].

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].

▶ **Definition 1.** *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 $p : \mathbb{N} \rightarrow \mathbb{N}$ such that for every $x \in \{0,1\}^*$:*

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

where by $M(x,u)$ we denote the computation of M where x is placed on its input tape and 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} [8]. 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 [8]. A language accepted by an oracle Turing Machine M with oracle A is denoted by $L^A(M)$ [8]. 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 [8]. In this definition, we bound the space of the oracle tape by a space $2^{O(S(n))}$ [8]. A nondeterministic oracle Turing machine can query $2^{2^{O(S(n))}}$ strings in the tree of all possible computations [8].

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 [8]. 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 [8]. It is trivial to see that $DSPACE^{\langle A \rangle}(S(n)) = DSPACE^A(S(n))$ [8]. Moreover, $L = NL$ if and only if

- $\forall S(n)\ \forall A\ DSPACE^A(S(n)) = NSPACE^A(S(n))$
- and $\forall S(n)\ \forall A\ DSPACE^{\langle A \rangle}(S(n)) = NSPACE^{\langle A \rangle}(S(n))$

for space constructible $S(n) \geq \log n$ [8].

We state the following Hypothesis:

▶ **Hypothesis 1.** *There is a language $L_1 \in$ 1NL–complete that is in L. Moreover, there is a nonempty language $L_2 \in co1NL$, 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 M runs in logarithmic space in the bit-length of w, the certificate string u is placed on the special read-once tape of M, and u is polynomially bounded by w. 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 $co1NL$.

We show the principal consequences of this Hypothesis:

▶ **Theorem 2.** *If the Hypothesis 1 is true, then $L = NL$ and $NP \subseteq L^{\langle L \rangle}$.*

Proof. If there is a language $L_1 \in$ *1NL–complete* in L, then $L = NL$ [6]. 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 [9]. 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 [9]. 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^{co1NL} 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 $L^{\langle L \rangle}$ when the Hypothesis 1 is true, since $NL^{co1NL} = NL^L = L^L = L^{\langle L \rangle}$ as a consequence of $L = NL$ [8]. Due to L_3 is closed under logarithm space reductions in *NP–complete*, then every NP is logarithmic space reduced to L_3. This implies that $NP \subseteq L^{\langle L \rangle}$ since L is closed under logarithm space reductions as well. ◀

1.2 The Problems

We describe the problems that we use and their complexity properties. We will say that the representation of a directed, acyclic graph, G is topological sorted if for any pair of edges (a, b) and (b, c) in G, (a, b) is listed before (b, c) [6].

▶ **Definition 3.** ***TAGAP***
INSTANCE: Two vertices s and t and a directed and acyclic graph G that is a topological sorted representation.
QUESTION: Is there a directed path from s to t in G?
REMARKS: TAGAP ∈ 1NL–complete [6].

A subpath is a path making up part of a larger path

▶ **Definition 4.** ***SUBPATH TAGAP (SPG)***
INSTANCE: Two vertices s and t and a directed and acyclic graph G that is a topological sorted representation.
QUESTION: Is every path starting from s a subpath of some directed path from s to t in G?
REMARKS: We know that SPG ∈ co1NL: we can decide whether there is a path starting from s that does not reach to t in a $1NL$ machine.

The logic operator $\oplus$(XOR) is used in some Boolean formulas instead of using $\vee$(OR).

▶ **Definition 5.** ***⊕2UNSAT***
INSTANCE: A Boolean formula ϕ that is the conjunction of a set of clauses $c_1, c_2, \ldots, c_m$ where each c_i consists of either a literal or is the XOR (EXCLUSIVE OR) of two literals.
QUESTION: Is it the case that ϕ is not satisfiable?
REMARKS: ⊕2UNSAT ∈ L [7], [10].

An independent set of a directed graph G is a set of vertices of G such that no two vertices in the independent set are joined by an edge in G.

▶ **Definition 6.** ***INDEPENDENT SET (ISET)***
INSTANCE: A positive integer K and a directed graph G.
QUESTION: Does G contain an independent set with K vertices or more?
REMARKS: ISET ∈ NP–complete [5].

2 Results

▶ **Theorem 7.** *TAGAP ∈ L.*

Proof. Consider a general directed and acyclic graph G that is a topological sorted representation and two vertices s and t. We reduce it to a CNF expression ϕ such that for each edge (a, b) in G, we create the clause $(\neg x_a \oplus x_b)$. Finally, we add the two clauses with a single literal (x_s) and $(\neg x_t)$. Since the graph G is topological sorted, then a directed path $s, v, w, \ldots, t$ is logically equivalent to

$$x_s \Rightarrow x_v \Rightarrow x_w \Rightarrow \ldots \Rightarrow x_t$$

in the CNF expression ϕ. However, that is false when the clauses (x_s) and $(\neg x_t)$ are satisfied in the Boolean formula ϕ at the same time. If there is no directed path between the vertices s and t, then ϕ can be satisfiable since the vertices reachable from s can be assigned in their variable representations as true and the vertices that reach to t can be assigned in their variable representation as false. For that reason, there is a directed path from s and t if and only if ϕ is not satisfiable. This reduction can be made in logarithmic space and thus, *TAGAP* ∈ L because of *⊕2UNSAT* ∈ L. ◀

▶ **Theorem 8.** *There is a deterministic Turing machine M, where:*

$$ISET = \{w : M(w, u) = y, \exists u \text{ such that } y \in SPG\}$$

when M runs in logarithmic space in the length of w, u is placed on the special read-once tape of M, and u is polynomially bounded by w.

Proof. The input could be a positive integer K and a directed graph with n vertices such that each vertex is represented by a unique integer between 1 and n. We can create a certificate array A which contains $\frac{(K+1)\cdot(K+2)}{2}$ edges that represents a directed and acyclic graph G' that is a topological sorted representation, every vertex is represented by an integer between 0 and $n+1$ and for any pair of edges (a, b) and (a, c) in G' such that $b < c$, (a, b) is listed before (a, c) and for any pair of edges (a, b) and (c, d) in G' such that $a < c$ or $a < d$, (a, b) is listed before (c, d). We read at once the edges of the array A and we reject when this is not the described graph G'. Besides, we check that the first vertex contains $K+1$ edges (that vertex is represented by 0 in G'), the second vertex contains K edges (that is the vertex that represents the minimum integer greater than 0 in G') and so on until we reach the penultimate vertex (that is the vertex that represents the maximum integer lesser than $n+1$ in G') that contains 1 edge (that's why the number of edges is $size = \frac{(K+1)\cdot(K+2)}{2}$ in G'). While we read the edges of the array A using the index i, we check those constraints in G' and verify that every edge in G' is not in G. In this way, we output two vertices and the same certificate (i.e. the edges of the array A), where the edges in G' do not exist in the current input G.

We obtain that all:

$$(K, G) \in ISET \Leftrightarrow \exists A \text{ such that } (0, n+1, A) \in SPG$$

because of when $(0, n+1, A) \in SPG$, then this would mean that G' is a complete graph after a conversion of the directed edges to undirected and we guarantee that those vertices are exactly an independent set of size K in G during the computation of the logarithmic space verifier M (we exclude the vertices represented by 0 and $n+1$ in G' inside of the independent set in G). Indeed, we can make this verifier in logarithmic space such that the array A is placed on the special read-once tape, because we read at once the edges in the array A. Hence, we only need to iterate from the elements of the array A to verify whether the array is an appropriated certificate according to the constraints of G' and check that every edge in G' is missing in G.

This logarithmic space verifier with output will be the Algorithm 1. We introduce some constraints in the Algorithm 1 in order to guarantee the theoretical procedure. For example, we assume that a value does not exist in the array A into the cell of a position i when $A[i] = null$. In addition, we immediately reject when the mentioned comparisons between the vertices in G' does 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, the vertex 0 would be the source vertex and $n+1$ is the sink vertex in the instance $(0, n+1, A) \in SPG$. ◀

▶ **Theorem 9.** *$L = NL$ and $NP \subseteq L^{\langle L \rangle}$.*

Proof. This is a directed consequence of Theorems 2, 7 and 8. Certainly, *ISET* is closed under logarithm space reductions in *NP–complete*. Indeed, we can reduced *SAT* to *ISET* in logarithmic space and every NP problem could be logarithmic space reduced to *SAT* by the Cook's Theorem Algorithm [5]. ◀

Algorithm 1 Logarithmic space verifier with output

```
/*A valid instance for ISET with its certificate*/
procedure VERIFIER((K, G), A)
    /*Initialize the previous left vertex*/
    left ← 0
    /*Initialize the previous right vertex*/
    right ← 0
    /*Initialize the index, total and size variables*/
    j ← 0
    total ← (K + 1)
    size ← ((K+1)·(K+2))/2
    /*Iterate for the edges of the certificate array A*/
    for i ← 1 to size do
        /*Assign the current edge*/
        (v, w) ← A[i]
        if A[i] = null ∨ v > n + 1 ∨ w < left ∨ w > n + 1 then
            return "no"
        else if (v, w) is an edge in G then
            return "no"
        else if i = 1 ∧ v ≠ 0 then
            return "no"
        else if i = size ∧ (A[i + 1] ≠ null ∨ w ≠ n + 1) then
            return "no"
        else if v ≠ left then
            if v ≤ left ∨ j ≠ total then
                return "no"
            end if
            left ← v
            right ← w
            j ← 0
            total ← total − 1
        else if w ≤ right ∨ j > total then
            return "no"
        else
            right ← w
            j ← j + 1
        end if
    end for
    output (0, n + 1, A)
end procedure
```

References

1 Scott Aaronson. P $\stackrel{?}{=}$ NP. *Electronic Colloquium on Computational Complexity, Report No. 4*, 2017.

2 Sanjeev Arora and Boaz Barak. *Computational complexity: a modern approach*. Cambridge University Press, USA, 2009.

3 Stephen Arthur Cook. The P versus NP Problem. `http://www.claymath.org/sites/default/files/pvsnp.pdf`, April 2000. Clay Mathematics Institute. Accessed 30 December 2022.

4 Thomas Cormen, Charles Eric Leiserson, Ronald Linn Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, USA, 3rd edition, 2009.

5 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.

6 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`.

7 Neil Jones, Edmund Lien, and William Laaser. New problems complete for nondeterministic log space. *Mathematical systems theory*, 10(1):1–17, 1976. `doi:10.1007/BF01683259`.

8 Pascal Michel. A survey of space complexity. *Theoretical computer science*, 101(1):99–132, 1992. `doi:10.1016/0304-3975(92)90151-5`.

9 Christos Harilaos Papadimitriou. *Computational complexity*. Addison-Wesley, USA, 1994.

10 Omer Reingold. Undirected connectivity in log-space. *Journal of the ACM (JACM)*, 55(4):1–24, 2008. `doi:10.1145/1391289.1391291`.

11 Michael Sipser. *Introduction to the Theory of Computation*, volume 2. Thomson Course Technology Boston, USA, 2006.