 doi:10.20944/preprints201908.0037.v3

P versus NP

Frank Vega
Joysonic, Uzun Mirkova 5, Belgrade, 11000, Serbia
vega.frank@gmail.com

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 demonstrate that every problem in NP could be NL-reduced to another problem in L. In this way, we prove that every problem in NP is in NL with L Oracle. Moreover, we show the complexity class NP is equal to NL, since it is well-known that the logarithmic space oracle hierarchy collapses into NL.

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

Keywords and phrases complexity classes, completeness, polynomial time, reduction, logarithmic space, one-way

1 Introduction

In previous years there has been great interest in the verification or checking of computations [13]. Interactive proofs introduced by Goldwasser, Micali and Rackoff and Babi can be viewed as a model of the verification process [13]. Dwork and Stockmeyer and Condon have studied interactive proofs where the verifier is a space bounded computation instead of the original model where the verifier is a time bounded computation [13]. In addition, Blum and Kannan have studied another model where the goal is to check a computation based solely on the final answer [13]. More about probabilistic logarithmic space verifiers and the complexity class NP has been investigated on a technique of Lipton [13]. In this work, we show some results about the logarithmic space verifiers applied to the class NP.

The P versus NP problem is a major unsolved problem in computer science [6]. This is considered by many to be the most important open problem in the field [6]. 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 [6]. The precise statement of the $P = NP$ problem was introduced in 1971 by Stephen Cook in a seminal paper [6]. In 2012, a poll of 151 researchers showed that 126 (83%) believed the answer to be no, 12 (9%) believed the answer is yes, 5 (3%) believed the question may be independent of the currently accepted axioms and therefore impossible to prove or disprove, 8 (5%) said either do not know or do not care or don't want the answer to be yes nor the problem to be resolved [10].

The $P = NP$ question is also singular in the number of approaches that researchers have brought to bear upon it over the years [8]. From the initial question in logic, the focus moved to complexity theory where early work used diagonalization and relativization techniques [8]. It was showed that these methods were perhaps inadequate to resolve P versus NP by demonstrating relativized worlds in which $P = NP$ and others in which $P \neq NP$ [4]. This shifted the focus to methods using circuit complexity and for a while this approach was deemed the one most likely to resolve the question [8]. Once again, a negative result showed that a class of techniques known as "Natural Proofs" that subsumed the above

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 6 April 2020 doi:10.20944/preprints201908.0037.v3

could not separate the classes NP and P, provided one-way functions exist [16]. There has been speculation that resolving the $P = NP$ question might be outside the domain of mathematical techniques [8]. More precisely, the question might be independent of standard axioms of set theory [8]. Some results have showed that some relativized versions of the $P = NP$ question are independent of reasonable formalizations of set theory [11].

It is fully expected that $P \neq NP$ [15]. Indeed, if $P = NP$ then there are stunning practical consequences [15]. For that reason, $P = NP$ is considered as a very unlikely event [15]. 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 [1]. Whether $P = NP$ or not is still a controversial and unsolved problem [1]. We show some results that could help us to prove this outstanding problem.

2 Theory and Methods

2.1 Preliminaries

In 1936, Turing developed his theoretical computational model [18]. The deterministic and nondeterministic Turing machines have become in two of the most important definitions related to this theoretical model for computation [18]. A deterministic Turing machine has only one next action for each step defined in its program or transition function [18]. 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 [18].

Let Σ be a finite alphabet with at least two elements, and let Σ^* be the set of finite strings over Σ [3]. A Turing machine M has an associated input alphabet Σ [3]. For each string w in Σ^* there is a computation associated with M on input w [3]. We say that M accepts w if this computation terminates in the accepting state, that is $M(w) = \text{“}yes\text{”}$ [3]. Note that M fails to accept w either if this computation ends in the rejecting state, that is $M(w) = \text{“}no\text{”}$, 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) [3].

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 [7]. A complexity class is a set of problems, which are represented as a language, grouped by measures such as the running time, memory, etc [7]. 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 $M(w) = \text{“}no\text{”}$ [7]. We denote by $t_M(w)$ the number of steps in the computation of M on input w [3]. 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 [3]. We say that M runs in polynomial time if there is a constant k such that for all n, $T_M(n) \leq n^k + k$ [3]. 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 [7]. A verifier for a language L_1 is a deterministic Turing machine M, where:

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

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 6 April 2020 doi:10.20944/preprints201908.0037.v3

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 [3]. A verifier uses additional information, represented by the symbol c, 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 [15].

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 [18]. 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 $f : \{0,1\}^* \to \{0,1\}^*$ 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* [9]. A language $L_1 \subseteq \{0,1\}^*$ is *NP–complete* if:

- $L_1 \in NP$, and
- $L' \leq_p L_1$ for every $L' \in NP$.

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* [7]. Moreover, if $L_1 \in NP$, then $L_1 \in$ *NP–complete* [7]. A principal *NP–complete* problem is SAT [9]. 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 [9]. 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 [7]. 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 [7]. A Boolean formula is in 3-conjunctive normal form or $3CNF$, if each clause has exactly three distinct literals [7].

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 $3CNF$. 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 $\neg x_2$. Another relevant *NP–complete* language is $3CNF$ satisfiability, or $3SAT$ [7]. In $3SAT$, it is asked whether a given Boolean formula ϕ in $3CNF$ is satisfiable.

A logarithmic space Turing machine has a read-only input tape, a write-only output tape, and read/write work tapes [18]. The work tapes may contain at most $O(\log n)$ symbols [18]. In computational complexity theory, L is the complexity class containing those decision problems that can be decided by a deterministic logarithmic space Turing machine [15]. NL is the complexity class containing the decision problems that can be decided by a nondeterministic logarithmic space Turing machine [15].

A logarithmic space transducer is a Turing machine with a read-only input tape, a write-only output tape, and read/write work tapes [18]. The work tapes must contain at most

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 6 April 2020 doi:10.20944/preprints201908.0037.v3

$O(\log n)$ symbols [18]. A logarithmic space transducer M computes a function $f : \Sigma^* \to \Sigma^*$, where $f(w)$ is the string remaining on the output tape after M halts when it is started with w on its input tape [18]. We call f a logarithmic space computable function [18]. 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 exists a logarithmic space computable function $f : \{0,1\}^* \to \{0,1\}^*$ 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 in the definition of the complete languages for the classes L and NL [15]. A Boolean formula is in 2-conjunctive normal form, or $2CNF$, if it is in CNF and each clause has exactly two distinct literals. There is a problem called $2SAT$, where we asked whether a given Boolean formula ϕ in $2CNF$ is satisfiable. $2SAT$ is complete for NL [15]. Another special case is the class of problems where each clause contains XOR (i.e. exclusive or) rather than (plain) OR operators. This is in P, since an $XOR\ SAT$ formula can also be viewed as a system of linear equations mod 2, and can be solved in cubic time by Gaussian elimination [14]. We denote the XOR function as $\oplus$. The *XOR 2SAT* problem will be equivalent to $XOR\ SAT$, but the clauses in the formula have exactly two distinct literals. *XOR 2SAT* is in L [2], [17].

2.2 Hypothesis

We can give a certificate-based definition for NL [3]. The certificate-based definition of NL assumes that a logarithmic space Turing machine has another separated read-only tape [3]. On each step of the machine, the machine's head on that tape can either stay in place or move to the right [3]. In particular, it cannot reread any bit to the left of where the head currently is [3]. For that reason this kind of special tape is called "read-once" [3].

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

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

where by $M(x,u)$ we denote the computation of M where x is placed on its input tape and the certificate 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 [3]. M is called a logarithmic space verifier [3].

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 head on the input tape to the left [12]. 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 [12]. We state the following Hypothesis:

▷ Hypothesis 2. Given a nonempty language $L_1 \in L$, there is a language L_2 in *NP–complete* under logarithmic space reductions with a deterministic Turing machine M, where:

$$L_2 = \{w : M(w,u) = y,\ \exists\ u \textit{ such that } y \in L_1\}$$

when M runs in one-way 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. In this way, there is an *NP–complete* language defined by a one-way logarithmic space verifier M such that when the input is an element of the language with its certificate, then M outputs a string which belongs to a single language in L.

▶ **Definition 3.** $L^S = \{A|A$ *is accepted by a logarithmic space deterministic oracle Turing machine using a language in the class* S *as its oracle set*$\}$ *[5].*

$NL^S = \{A|A$ *is accepted by a logarithmic space nondeterministic oracle Turing machine using a language in the class* S *as its oracle set*$\}$ *[5].*

In this model the oracle tape is not subject to the space bound, but is write-only and is erased after each query, and in addition the machine is in deterministic mode while writing each query, and hence any query is of polynomial size [5].

▶ **Theorem 4.** *If the Hypothesis 2 is true, then every problem in* NP *is in* NL^L*.*

Proof. We can simulate the computation $M(w, u) = y$ in the Hypothesis 2 by a nondeterministic logarithmic space Turing machine N such that $N(w) = y$, 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 [15]. 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 [15]. We remove each symbol 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. Moreover, we know that N is a nondeterministic one-way logarithmic space Turing machine.

For every language $L_3 \in NP$, then we can reduce the elements of the language L_3 to the elements of the language L_1 by a nondeterministic logarithmic space Turing machine M''. In this way, there is a nondeterministic logarithmic space Turing machine $M''(x) = N(M'(x))$ which will nondeterministically output an element $y \in L_1$ when $x \in L_3$. The deterministic logarithmic space Turing machine M' will be the logarithmic space reduction of L_3 to L_2, because of L_2 is in *NP–complete* under logarithmic space reductions. Actually, the nondeterministic logarithmic space reduction is possible, because of N is in one way. Indeed, it is not necessary to reset the computation of M' in the composition $N(M'(x))$ on the input x, because N never moves to the left the head on the input tape (that would be the output tape of M'). Since $L_1 \in L$, then we obtain that every problem in NP could be NL-reduced to another problem in L when the Hypothesis 2 is true.

Since every problem in NP could be NL-reduced to another problem in L when the Hypothesis 2 is true, then we can simulate the writing in the oracle tape in a deterministic mode just copying nondeterministically the symbols in the work tapes and copying deterministically to the oracle tape from the work tapes (instead of copying in the output tape). After of each one of these copying process, then we erase the symbols that were written in the work tapes to keep a logarithmic space computation. Since in the oracle tape will remain an instance of a problem in L (we interchange the output tape with the oracle tape), then we have that every problem in NP is in NL^L when the Hypothesis 2 is true. ◀

3 Results

We show a previous known *NP–complete* problem:

▶ **Definition 5.** ***NAE 3SAT***

INSTANCE: A Boolean formula ϕ *in* $3CNF$*.*

QUESTION: Is there a truth assignment for ϕ *such that each clause has at least one true literal and at least one false literal?*

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 6 April 2020 doi:10.20944/preprints201908.0037.v3

REMARKS: NAE 3SAT $\in$ *NP–complete [9].*

We define a new problem:

▶ **Definition 6.** ***MAXIMUM EXCLUSIVE-OR 2SAT***
INSTANCE: A positive integer K *and a Boolean formula* ϕ *that is an instance of XOR 2SAT.*
QUESTION: Is there a truth assignment in ϕ *such that at most* K *clauses are unsatisfied?*
REMARKS: We denote this problem as $MAX \oplus 2SAT$*.*

▶ **Theorem 7.** $MAX \oplus 2SAT \in$ *NP–complete.*

Proof. It is trivial to see $MAX \oplus 2SAT \in NP$ [15]. Given a Boolean formula ϕ in $3CNF$ with n variables and m clauses, we create three new variables a_{c_i}, b_{c_i} and d_{c_i} for each clause $c_i = (x \vee y \vee z)$ in ϕ, where x, y and z are literals, in the following formula:

$$P_i = (a_{c_i} \oplus b_{c_i}) \wedge (b_{c_i} \oplus d_{c_i}) \wedge (a_{c_i} \oplus d_{c_i}) \wedge (x \oplus a_{c_i}) \wedge (y \oplus b_{c_i}) \wedge (z \oplus d_{c_i}).$$

We can see P_i has at most one unsatisfied clause for some truth assignment if and only if at least one member of $\{x, y, z\}$ is true and at least one member of $\{x, y, z\}$ is false for the same truth assignment. Hence, we can create the Boolean formula ψ as the conjunction of the P_i formulas for every clause c_i in ϕ, such that $\psi = P_1 \wedge \ldots \wedge P_m$. Finally, we obtain that:

$$\phi \in \textit{NAE 3SAT if and only if } (\psi, m) \in MAX \oplus 2SAT.$$

Consequently, we prove *NAE 3SAT* $\leq_p MAX \oplus 2SAT$ where we already know the language *NAE 3SAT* $\in$ *NP–complete* [9]. Moreover, this reduction is also a logarithmic space reduction [15]. To sum up, we show $MAX \oplus 2SAT \in$ *NP–hard* and $MAX \oplus 2SAT \in NP$ and thus, $MAX \oplus 2SAT \in$ *NP–complete.* ◀

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

$$MAX \oplus 2SAT = \{w : M(w, u) = y, \exists\ u \textit{ such that } y \in \textit{XOR 2SAT}\}$$

when M *runs in one-way 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. Given a valid instance (ψ, K) for $MAX \oplus 2SAT$ when ψ has m clauses, we can create a certificate array A which contains K different natural numbers in ascending order which represents the indexes of the clauses in ψ that we are going to remove from the instance. We read at once the elements of the array A and we reject whether this is not an appropriated certificate: That is when the numbers are not sorted in ascending order, or the array A does not contain exactly K elements, or the array A contains a number that is not between 1 and m. While we read the elements of the array A, we remove the clauses from the instance (ψ, K) for $MAX \oplus 2SAT$ just creating another instance ϕ for *XOR 2SAT* where the Boolean formula ϕ does not contain the K different indexed clauses ψ represented by the numbers in A. Therefore, we obtain the array A would be valid according to the Theorem 8 when:

$$(\psi, K) \in MAX \oplus 2SAT \Leftrightarrow (\exists \textit{ array } A \textit{ such that } \phi \in \textit{XOR 2SAT}).$$

Furthermore, we can make this verification in logarithmic space such that the array A is placed on the special read-once tape, because we read at once the elements in the array A and we assume the clauses in the input ψ are indexed from left to right. Hence, we only need

 doi:10.20944/preprints201908.0037.v3

Algorithm 1 Logarithmic space verifier

```
/*A valid instance for MAX ⊕ 2SAT with its certificate*/
procedure VERIFIER((ψ, K), A)
    /*Initialize minimum and maximum values*/
    min ← 1
    max ← 0
    /*Iterate for the elements of the certificate array A*/
    for i ← 1 to K + 1 do
        if i = K + 1 then
            /*There exists a K + 1 element in the array*/
            if A[i] ≠ undefined then
                /*Reject the certificate*/
                return "no"
            end if
            /*m is the number of clauses in ψ*/
            max ← m + 1
        else if A[i] = undefined ∨ A[i] ≤ max ∨ A[i] < 1 ∨ A[i] > m then
            /*Reject the certificate*/
            return "no"
        else
            max ← A[i]
        end if
        /*Iterate for the clauses of the Boolean formula ψ*/
        for j ← min to max − 1 do
            /*Output the indexed j^th clause in ψ*/
            output " ∧ c_j"
        end for
        min ← max + 1
    end for
end procedure
```

to iterate from the elements of the array A to verify whether the array is an appropriated certificate and also remove the K different clauses from the Boolean formula ψ when we write the final clauses to the output. This logarithmic space verification will be the Algorithm 1. We assume whether a value does not exist in the array A into the cell of some position i when $A[i] = \mathit{undefined}$. In addition, we reject immediately when the following comparisons:

$$A[i] \leq max \vee A[i] < 1 \vee A[i] > m$$

hold at least into one single binary digit. Note, in the loop j from min to $max - 1$, we do not output any clause when $max - 1 < min$. Certainly, the Algorithm 1 is in one-way, since this never moves the head on the input tape to the left. ◀

▶ **Theorem 9.** $NL = NP$.

Proof. Every *NP–complete* is logarithmic space reduced to $MAX \oplus 2SAT$. Certainly, every NP problem could be logarithmic space reduced to SAT by the Cook's Theorem algorithm [9]. In addition, the problem SAT could be logarithmic space reduced to *NAE 3SAT* [9]. Moreover, the problem *NAE 3SAT* could be logarithmic space reduced to $MAX \oplus 2SAT$ according to Theorem 7. Therefore, we obtain that every problem in NP is in NL^L, since this is a direct consequence of Theorems 4, 7 and 8. Every problem in NL^L is in NL^{NL} as well [5]. Since we know that $NL^{NL} = L^{NL}$ [5], then every problem in NP is in L^{NL}. Neil Immerman proved that $NL = coNL$ [15]. His result provides a direct proof and improvement of the main result in [5], by collapsing the logarithmic space oracle hierarchy into NL. In this way, we have that the complexity class NP is equal to NL. ◀

References

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

2 Carme Álvarez and Raymond Greenlaw. A Compendium of Problems Complete for Symmetric Logarithmic Space. *Computational Complexity*, 9(2):123–145, 2000. doi:10.1007/PL00001603.

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

4 Theodore Baker, John Gill, and Robert Solovay. Relativizations of the $\mathcal{P} =? \mathcal{NP}$ Question. *SIAM Journal on computing*, 4(4):431–442, 1975. doi:10.1137/0204037.

5 Samuel R Buss, Stephen A Cook, Patrick W Dymond, and Louise Hay. The Log Space Oracle Hierarchy Collapses. 1987.

6 Stephen A. Cook. The P versus NP Problem, April 2000. In Clay Mathematics Institute at http://www.claymath.org/sites/default/files/pvsnp.pdf.

7 Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, 3rd edition, 2009.

8 Vinay Deolalikar. P $\neq$ NP, 2010. In Woeginger Home Page at https://www.win.tue.nl/~gwoegi/P-versus-NP/Deolalikar.pdf.

9 Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. San Francisco: W. H. Freeman and Company, 1 edition, 1979.

10 William I. Gasarch. Guest column: The second P $\stackrel{?}{=}$ NP poll. *ACM SIGACT News*, 43(2):53–77, 2012. doi:10.1145/2261417.2261434.

11 Juris Hartmanis and John E. Hopcroft. Independence Results in Computer Science. *SIGACT News*, 8(4):13–24, October 1976. doi:10.1145/1008335.1008336.

12 Juris Hartmanis and Stephen R. 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.

13 Richard J. Lipton. Efficient checking of computations. In *STACS 90*, pages 207–215. Springer Berlin Heidelberg, 1990. doi:10.1007/3-540-52282-4_44.

14 Cristopher Moore and Stephan Mertens. *The Nature of Computation*. Oxford University Press, 2011.

15 Christos H. Papadimitriou. *Computational complexity*. Addison-Wesley, 1994.

16 Alexander A. Razborov and Steven Rudich. Natural Proofs. *J. Comput. Syst. Sci.*, 55(1):24–35, August 1997. doi:10.1006/jcss.1997.1494.

17 Omer Reingold. Undirected Connectivity in Log-space. *J. ACM*, 55(4):1–24, September 2008. doi:10.1145/1391289.1391291.

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