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 if L is equal to NL, then L = NP. In this way, we demonstrate that the L versus NL problem is as hard as P versus NP.

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 and logarithmic space disqualifiers applied to the class $coNP$ which solve one of the most important open problems in computer science, that is P versus NP.

The P versus NP problem is a major unsolved problem in computer science [5]. This is considered by many to be the most important open problem in the field [5]. 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 [5]. The precise statement of the $P = NP$ problem was introduced in 1971 by Stephen Cook in a seminal paper [5]. 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 [9].

The $P = NP$ question is also singular in the number of approaches that researchers have brought to bear upon it over the years [7]. From the initial question in logic, the focus moved to complexity theory where early work used diagonalization and relativization techniques [7]. 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 [7]. Once again, a negative result showed that a class of techniques known as "Natural Proofs" that subsumed the above 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 [7]. More precisely, the question might be independent of standard axioms of set theory [7]. Some results have showed that some relativized versions of the $P = NP$ question are independent of reasonable formalizations of set theory [10].

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

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 [6]. A complexity class is a set of problems, which are represented as a language, grouped by measures such as the running time, memory, etc [6]. 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 $M(w) = \text{“}no\text{”}$ [6]. 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 [6]. A verifier for a language L_1 is a deterministic Turing machine M, where:

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

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]. If NP is the class of problems that have succinct certificates, then the complexity class $coNP$ must contain those problems that have succinct disqualifications [15]. That is, a "*no*" instance of a problem in $coNP$ possesses a short proof of its being a "*no*" instance [15].

▶ **Definition 1.** *We will extend the definition of succinct disqualification for an element $w \in L_2$ when $L_2 \in coNP$ as the polynomially bounded string c by w such that $M(w, c) =$ "no" and M is the polynomial time verifier of the complement of L_2 in NP.*

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 $x \in \{0, 1\}^*$:

$x \in L_1$ *if and only if* $f(x) \in L_2$.

An important complexity class is *NP–complete* [8]. 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* [6]. Moreover, if $L_1 \in NP$, then $L_1 \in$ *NP–complete* [6]. A principal *NP–complete* problem is SAT [8]. 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 [8]. 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 [6]. 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 [6]. A Boolean formula is in 3-conjunctive normal form or $3CNF$, if each clause has exactly three distinct literals [6].

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 $3SAT$ [6]. 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 $O(\log n)$ 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].

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]. Besides, in the certificate-based definition of NL, we assume the certificate string is appropriated for the instance [15]. For example, a truth assignment for a Boolean formula ϕ is appropriated for the instance when every possible variable in ϕ could be evaluated in that truth assignment string, but we cannot affirm the same for every possible binary string.

▶ **Definition 2.** *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 $x \in \{0,1\}^*$:*

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

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].

An important complexity class is *coNP–complete* [8]. A language $L_1 \subseteq \{0,1\}^*$ is *coNP–complete* if:

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

If L_1 is a language such that $L' \leq_p L_1$ for some $L' \in$ *coNP–complete*, then L_1 is *coNP–hard* [6]. Moreover, if $L_1 \in coNP$, then $L_1 \in$ *coNP–complete* [6]. A principal *coNP–complete* problem is $UNSAT$ [8]. A Boolean formula without any satisfying truth assignment is unsatisfiable. The problem $UNSAT$ asks whether a given Boolean formula is unsatisfiable [8].

$coNL$ is the complexity class containing the languages such that their complements belong to NL [15]. We can give a disqualification-based definition for $coNL$ [3]. The disqualification-based definition of $coNL$ assumes that a logarithmic space Turing machine has another separated read-only tape, that is the same kind of special tape called "read-once" that we use in the certificate-based definition for NL [3]. Besides, in the disqualification-based definition of $coNL$, we assume the disqualification string is appropriated for the instance [15].

▶ **Definition 3.** *A language L_1 is in $coNL$ 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 $x \in \{0,1\}^*$:*

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

where by $M(x,u)$ we denote the computation of M where x is placed on its input tape and the disqualification 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. M is called a logarithmic space disqualifier.

For example, there is a well-known $coNL$ problem that states: Given a directed graph $G = (V,E)$ and two nodes $s,t \in V$, is there no possible path from s to t? In that problem, an appropriated disqualification u is a sequence of nodes contained in V when s is the first node and t is the last one such that this sequence of nodes is not a path: There is at least a consecutive pair of nodes in the sequence where they are not connected by an edge.

2.1 Hypothesis

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 [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 [11]. 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 [11]. Furthermore, they have proven that $1NL \subseteq L$ if and only if $L = NL$ [11].

We state the following Hypothesis:

▷ Hypothesis 4. Given a nonempty language $L_1 \in 1NL$, there is a language L_2 in *coNP–complete* under logarithmic space reductions with a deterministic Turing machine M, where:

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

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. In this way, there is a *coNP–complete* language defined by a logarithmic space disqualifier M such that when the input is an element of the language with any of its appropriated disqualification, then M always outputs a string which belongs to a single language in $1NL$.

▶ **Theorem 5.** *When the Hypothesis 4 is true, therefore if $L = NL$, then $L = NP$.*

Proof. We can accept the elements of the language $L_1 \in 1NL$ by a nondeterministic one-way logarithmic space Turing machine M'. In this way, there is a nondeterministic logarithmic space Turing machine $M''(w, u) = M'(M(w, u))$ which will accept when $w \in L_2$ for all the appropriated disqualification u, where u is placed on the special read-once tape of M''.

The reason is because we can simulate the output string of $M(w, u)$ within a read-once tape and thus, we can compute in a nondeterministic logarithmic space the logarithmic space composition using the same techniques of the logarithmic space composition reduction, but without any reset of the computation [15]. Certainly, we do not need to reset the computation of $M(w, u)$ for the reading at once of a symbol in the output string of $M(w, u)$ by the nondeterministic one-way logarithmic space Turing machine M'. Actually, the logarithmic space reduction is possible, because of M' is in one way. Indeed, it is not necessary to reset the computation of M in the composition $M'(M(w, u))$ on the input w and disqualification u, because M' never moves to the left the head on the input tape (that would be the output tape of M).

Consequently, M'' can be converted into a logarithmic space disqualifier for the language L_2 just assuming that $L = NL$, because of the nondeterministic logarithmic space Turing machine M'' could be simulated by a deterministic logarithmic space Turing machine. Therefore, L_2 is in $coNL$ and thus, $L_2 \in P$ due to $coNL \subseteq P$ [15]. If any single *coNP–complete* problem can be solved in polynomial time, then $P = NP$ [15]. Since $L_2 \in P$ and $L_2 \in$ *coNP–complete*, then we obtain the complexity class P is equal to NP. Since $coNL = NL$ and the language L_2 is in *coNP–complete* under logarithmic space reductions, then we obtain $L = NP$ under the assumption that $L = NL$ when the Hypothesis 4 is true. ◀

3 Results

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

▶ **Definition 6.** ***3UNSAT***
INSTANCE: A Boolean formula ϕ in $3CNF$.
QUESTION: Is ϕ unsatisfiable?
REMARKS: $3UNSAT \in$ coNP–complete [8].

We define a new problem:

▶ **Definition 7.** ***SUM ZERO***
INSTANCE: A collection of integers C such that $0 \notin C$ and every integer in C has the same bit-length of the number that represents the cardinality of C multiplied by 3 (we do not take into account the symbol minus in counting the bit-length of the negative integers).
QUESTION: Are there two elements $a, b \in C$, such that $a + b = 0$?
REMARKS: We denote this problem as SUM–ZERO.

▶ **Theorem 8.** *SUM–ZERO $\in 1NL$.*

Proof. Given a collection of integers C, we can read its elements from left to right, verify that every element is not equal to 0, check that every element in C has the same bit-length and count the amount of elements in C to finally multiply it by 3 and compare its bit-length with the single bit-length from the elements in C. In addition, we can nondeterministically pick two elements a and b from C and accept in case of $a + b = 0$ otherwise we reject. We can make all this computation in a nondeterministic one-way using logarithmic space. Certainly, the calculation and store of the bit-length of the elements in C could be done in logarithmic space since this is a unique value. On the one hand, we can count and store the number of elements that we read from the input and multiply it by 3 to finally compare its bit-length with the stored unique bit-length from the elements of the collection, since the cardinality of C multiplied by 3 could be stored in a binary number of bit-length that is logarithmic in relation to the encoded length of C. On the other hand, the two elements a and b that we pick from C have a logarithmic space in relation to the encoded length of C, because of every integer in C has the same bit-length of the number that represents the cardinality of C multiplied by 3. Indeed, we never need to read to the left on the input for the acceptance of the elements in *SUM–ZERO* in a nondeterministic logarithmic space. ◀

▶ **Theorem 9.** *There is a deterministic Turing machine M, where:*

$$3UNSAT = \{w : M(w, u) = y, \forall \text{ appropriated } u \text{ such that } y \in \text{SUM–ZERO}\}$$

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. Given a Boolean formula ϕ in $3CNF$ with n variables and m clauses, we can create a disqualification array A which contains m positive integers between 1 and 3 which represents the literals of the clauses in ϕ which appear from left to right. We read at once the elements of the array A and we reject whether this is not an appropriated disqualification: That is when the array A does not contain exactly m elements, or the array A contains a number that is not between 1 and 3. While we read the elements of the array A, we select from the clauses ϕ the literals such that these ones occupy the position that represents the number between 1 and 3, that is the first, second or third place within the clause from left to right. In this way, we output the selected literals that are represented by a positive or negative (in case of a negated variable) integer just creating another instance C for *SUM–ZERO* where the collection C contains those integers which are the selected literals for each clause in ϕ. Therefore, we obtain that all the appropriated array A would be valid according to the Theorem 9 when:

$$\phi \in 3UNSAT \Leftrightarrow (\forall \text{ appropriated array } A \text{ such that } C \in \text{SUM–ZERO})$$

since we assume the positive and negated literals of some variable in the input ϕ correspond to a positive integer and its negative value, respectively. Furthermore, we can make this disqualification 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. Hence, we only need to iterate from the elements of the array A to verify whether the array is an appropriated disqualification and pick the m literals from the Boolean formula ϕ when we write the final integers that represent these literals to the output. This logarithmic space disqualification 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 $A[i] = \textit{undefined}$. In addition, we reject immediately when the following comparisons:

$$A[i] < 1 \vee A[i] > 3$$

hold at least into one single binary digit. Note, that every possible literal in ϕ could have a representation by an integer between $-3 \times m$ and $3 \times m$ with the exception of 0, where m is the cardinality of the collection C. In this way, we guarantee the output collection C is an appropriated instance of *SUM–ZERO* just filling with zeroes to the left the elements with bit-length lesser than $|3 \times m|$ where $|\ldots|$ is the bit-length function. ◀

Algorithm 1 Logarithmic space disqualifier

```
/*A valid instance for 3UNSAT with its disqualification*/
procedure DISQUALIFIER(φ, A)
    /*Initialize an index*/
    j ← 0
    /*m is the number of clauses in φ*/
    /*Iterate for the elements of the disqualification array A*/
    for i ← 1 to m + 1 do
        if i = m + 1 then
            /*There exists an m + 1 element in the array*/
            if A[i] ≠ undefined then
                /*Reject the disqualification*/
                return "no"
            end if
            /*Break the for loop*/
            break
        else if A[i] = undefined ∨ A[i] < 1 ∨ A[i] > 3 then
            /*Reject the disqualification*/
            return "no"
        else
            j ← A[i]
        end if
        /*From the indexed i^th clause c_i = (x_j ∨ y_k ∨ z_r) in φ*/
        /*Where x, y and z are literals with local indexes {j, k, r} = {1, 2, 3} in c_i*/
        /*Output the integer representation of the j^th literal, that is n(x_j)*/
        /*Filled with zeroes to the left until a total of |3 × m| bits including the literal*/
        /*But, the bit-length of the symbol minus is ignored in filling the negated literals*/
        output " , n(x_j)"
    end for
end procedure
```

▶ **Theorem 10.** *The Hypothesis 4 is true.*

Proof. Every *coNP–complete* is logarithmic space reduced to $3UNSAT$. Certainly, every $coNP$ problem could be logarithmic space reduced to $3UNSAT$ by the Cook's Theorem algorithm [8]. Hence, this is a direct consequence of Theorems 8 and 9. ◀

▶ **Theorem 11.** *If $L = NL$, then $L = NP$.*

Proof. This is a direct consequence of Theorems 5 and 10. ◀

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

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

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

8 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.

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

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

11 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`.

12 Martin Kutrib, Julien Provillard, György Vaszil, and Matthias Wendlandt. Deterministic One-Way Turing Machines with Sublinear Space. *Fundamenta Informaticae*, 136(1-2):139–155, 2015. `doi:10.3233/FI-2015-1147`.

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.