NL 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 not equal to NL, then P = NP. In addition, we show if L is equal to NL, then P = NP. In this way, we prove the complexity class P is equal to NP. Furthermore, we demonstrate the complexity class NL is equal to 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

1 Introduction

In previous years there has been great interest in the verification or checking of computations [16]. Interactive proofs introduced by Goldwasser, Micali and Rackoff and Babi can be viewed as a model of the verification process [16]. 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 [16]. In addition, Blum and Kannan has studied another model where the goal is to check a computation based solely on the final answer [16]. More about probabilistic logarithmic space verifiers and the complexity class NP has been investigated on a technique of Lipton [16]. 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 [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 [12].

The $P = NP$ question is also singular in the number of approaches that researchers have brought to bear upon it over the years [9]. From the initial question in logic, the focus moved to complexity theory where early work used diagonalization and relativization techniques [9]. 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 [9]. 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 [20]. There has been speculation that resolving the $P = NP$ question might be outside the domain of mathematical techniques [9]. More precisely, the question might be independent of standard axioms of set theory [9]. Some results have showed that some relativized versions of the $P = NP$ question are independent of reasonable formalizations of set theory [13].

It is fully expected that $P \neq NP$ [19]. Indeed, if $P = NP$ then there are stunning practical consequences [19]. For that reason, $P = NP$ is considered as a very unlikely event [19]. 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 prove this outstanding problem with the unexpected solution of $P = NP$.

2 Theory and Methods

2.1 Preliminaries

In 1936, Turing developed his theoretical computational model [22]. The deterministic and nondeterministic Turing machines have become in two of the most important definitions related to this theoretical model for computation [22]. A deterministic Turing machine has only one next action for each step defined in its program or transition function [22]. 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 [22].

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 $M(w) = 1$ [3]. Note that M fails to accept w either if this computation ends in the rejecting state, that is $M(w) = 0$, 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:

$$L(M) = \{w \in \Sigma^* : M(w) = 1\}.$$

Moreover, $L(M)$ is decided by M, when $w \notin L(M)$ if and only if $M(w) = 0$ [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:

$$L_1 = \{w : M(w, c) = 1 \textit{ for some string } c\}.$$

We measure the time of a verifier only in terms of the length of w, so a polynomial time verifier runs in polynomial time in the length of w [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 [19].

2.2 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 [22]. 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* [11]. 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 [11]. 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 [11]. 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 [22]. The work tapes may contain at most $O(\log n)$ symbols [22]. In computational complexity theory, L is the complexity class containing those decision problems that can be decided by a deterministic logarithmic space Turing machine [19]. NL is the complexity class containing the decision problems that can be decided by a nondeterministic logarithmic space Turing machine [19].

A logarithmic space transducer is a Turing machine with a read-only input tape, a write-only output tape, and read/write work tapes [22]. The work tapes must contain at most $O(\log n)$ symbols [22]. 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 [22]. We call f a logarithmic space computable function [22]. 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 [19]. 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 [19]. 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 [18]. 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], [21].

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

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

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].

We state the following Hypothesis:

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

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

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 an *NP–complete* language defined by a 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.

▶ **Theorem 3.** *When the Hypothesis 2 is true, therefore if L_2 is NP–complete under logarithmic space reduction, then $NL = NP$.*

Proof. Since every problem in L is *L–complete* under logarithmic space reduction, then we can interpret the quantification of the above statement as "For any nonempty language L_1 in L, there is an *NP–complete* language L_2 ... (rest is the same)". However, if we can choose an arbitrary language L_1 in L, then choosing L_1 the trivial language $\{1\}$ will result in the right hand side of the expression defining L_2 contained in NL. Moreover, if that problem in NL is *NP–complete* under logarithmic space reduction, then we obtain that certainly $NL = NP$. ◀

3 Results

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

▶ **Definition 4.** ***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?

REMARKS: NAE 3SAT $\in$ NP–complete [11].

We define a new problem:

▶ **Definition 5.** ***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 6.** *$MAX \oplus 2SAT \in$ NP–complete.*

Proof. It is trivial to see $MAX \oplus 2SAT \in NP$ [19]. 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* [11]. 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 7.** *There is a deterministic Turing machine M, where:*

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

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 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 7 when:

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

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 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 $A[i] = \textit{undefined}$. In addition, we reject immediately when the following comparisons:

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

hold at least into one single binary digit. Note, in the loop j from $\textit{min}$ to $\textit{max} - 1$, we do not output any clause when $\textit{max} - 1 < \textit{min}$.

◀

▶ **Theorem 8.** *The Hypothesis 2 is true.*

Proof. This is a consequence of Theorems 6 and 7. ◀

▶ **Theorem 9.** *$NL = NP$ and thus, $P = NP$.*

Proof. The Hypothesis 2 is true according to Theorem 8. Since the polynomial time reduction in Theorem 6 could be easily transformed in a logarithmic space reduction, then the *NP–complete* problem in Hypothesis 2, that would be $MAX \oplus 2SAT$, is necessarily in NL and thus all the problems in NP, because of the Cook's Theorem can also be transformed in a logarithmic space reduction [11]. Certainly, every NP problem could be logarithmic space reduced to SAT by the Cook's Theorem algorithm and SAT can be indeed logarithmic space reduced to *NAE 3SAT* [11]. In addition, as a consequence of Theorem 6, the problem *NAE 3SAT* could be logarithmic space reduced to $MAX \oplus 2SAT$. In this way, we obtain that $NL = NP$ as result of Theorem 3. Since $NL \subseteq P$, then $P = NP$ [19]. ◀

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 0
            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 0
        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
```

4 Conclusions

No one has been able to find a polynomial time algorithm for any of more than 300 important known *NP–complete* problems [11]. A proof of $P = NP$ will have stunning practical consequences, because it leads to efficient methods for solving some of the important problems in NP [6]. The consequences, both positive and negative, arise since various *NP–complete* problems are fundamental in many fields [6]. The following consequences are assuming that we have a practical solution for the *NP–complete* problems where such existence was proven with our nonconstructive result:

Cryptography, for example, relies on certain problems being difficult. A constructive and efficient solution to an *NP–complete* problem such as $3SAT$ will break most existing cryptosystems including: Public-key cryptography [14], symmetric ciphers [17] and one-way functions used in cryptographic hashing [8]. These would need to be modified or replaced by information-theoretically secure solutions not inherently based on *P–NP* equivalence.

There are enormous 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 [11]. Efficient solutions to these problems have enormous implications for logistics [6]. Many other important problems, such as some problems in protein structure prediction, are also *NP–complete*, so this will spur considerable advances in biology [5].

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].

There would be disruption, including maybe displacing programmers [15]. The practice of programming itself would be more about gathering training data and less about writing code [15]. Google would have the resources to excel in such a world [15].

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 [1]. For instance, Fermat's Last Theorem took over three centuries to prove [1]. A method that is guaranteed to find proofs to theorems, should one exist of a "reasonable" size, would essentially end this struggle [6].

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 Bonnie Berger and Tom Leighton. Protein Folding in the Hydrophobic-Hydrophilic (HP) Model is NP-complete. *Journal of Computational Biology*, 5(1):27–40, 1998. doi:10.1145/279069.279080.

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 Debapratim De, Abishek Kumarasubramanian, and Ramarathnam Venkatesan. Inversion Attacks on Secure Hash Functions Using SAT Solvers. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 377–382. Springer, 2007. doi:10.1007/978-3-540-72788-0_36.

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

10 Lance Fortnow. The Status of the P Versus NP Problem. *Commun. ACM*, 52(9):78–86, September 2009. doi:10.1145/1562164.1562186.

11 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.

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

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

14 Satoshi Horie and Osamu Watanabe. Hard instance generation for SAT. *Algorithms and Computation*, pages 22–31, 1997. doi:10.1007/3-540-63890-3_4.

15 Russell Impagliazzo. A personal view of average-case complexity. In *Proceedings of Structure in Complexity Theory. Tenth Annual IEEE Conference*, pages 134–147. IEEE, 1995. doi:10.1109/SCT.1995.514853.

16 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.

17 Fabio Massacci and Laura Marraro. Logical Cryptanalysis as a SAT Problem. *Journal of Automated Reasoning*, 24(1):165–203, 2000. doi:10.1023/A:1006326723002.

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

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

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

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

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