

# 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? 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. 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, one-way

## 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 Babai 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]. It was essentially mentioned in 1955 from a letter written by John Nash to the United States National Security Agency [1]. However, 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 [11]. 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 [21]. The deterministic and nondeterministic Turing machines have become in two of the most important definitions related to this theoretical model for computation [21]. A deterministic Turing machine has only one next action for each step defined in its program or transition function [21]. 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 [21].

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

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

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  $\Sigma^n$  is the set of all strings over  $\Sigma$  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:

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

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]. If  $NP$  is the class of problems that have succinct certificates, then the complexity class  $coNP$  must contain those problems that have succinct disqualifications [19]. That is, a “no” instance of a problem in  $coNP$  possesses a short proof of its being a “no” instance [19].

► **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  $c$  such that  $M(w, c) = \text{"no"}$  and  $M$  is the polynomial time verifier of the complement of  $L_2$  in  $NP$ .

## 2.2 First 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 [21]. Let  $\{0, 1\}^*$  be the infinite set of binary strings, we say that a language  $L_1 \subseteq \{0, 1\}^*$  is polynomial time reducible to a language  $L_2 \subseteq \{0, 1\}^*$ , written  $L_1 \leq_p L_2$ , if there is a polynomial time computable function  $f : \{0, 1\}^* \rightarrow \{0, 1\}^*$  such that for all  $x \in \{0, 1\}^*$ :

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

An important complexity class is *NP-complete* [10]. 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  $L' \in NP\text{-complete}$ , then  $L_1$  is *NP-hard* [7]. Moreover, if  $L_1 \in NP$ , then  $L_1 \in NP\text{-complete}$  [7]. A principal *NP-complete* problem is *SAT* [10]. An instance of *SAT* is a Boolean formula  $\phi$  which is composed of:

1. Boolean variables:  $x_1, x_2, \dots, x_n$ ;
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  $\phi$  is a set of values for the variables in  $\phi$ . A satisfying truth assignment is a truth assignment that causes  $\phi$  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 [10]. 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  $\phi$  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 [21]. The work tapes may contain at most  $O(\log n)$  symbols [21]. 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 [21]. The work tapes must contain at most  $O(\log n)$  symbols [21]. A logarithmic space transducer  $M$  computes a function  $f : \Sigma^* \rightarrow \Sigma^*$ , where  $f(w)$  is the string remaining on the output tape after  $M$  halts when it is started with

$w$  on its input tape [21]. We call  $f$  a logarithmic space computable function [21]. 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  $f : \{0, 1\}^* \rightarrow \{0, 1\}^*$  such that for all  $x \in \{0, 1\}^*$ :

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

The logarithmic space reduction is used 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  $\phi$  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], [20].

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

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

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  $|\dots|$  is the bit-length function [3].  $M$  is called a logarithmic space verifier [3].

We state the following Hypothesis:

▷ **Hypothesis 3.** Given a nonempty language  $L_1 \in L$ , there is a language  $L_2$  in  $NP$ -complete with a deterministic Turing machine  $M$ , where:

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

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  $L$ .

From the early days of automata and complexity theory, two different models of Turing machines are considered, the offline and online machines [15]. Each model has a read-only input tape and some work tapes [15]. The offline machines may read their input two-way while the online machines are not allowed to move the input head to the left [15]. In the terminology of the (generalized) Turing machine models are called two-way and one-way Turing machines, respectively [15].

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]. 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 [12]. Furthermore, they have proven that  $1NL \subseteq L$  if and only if  $L = NL$  [12].

► **Theorem 4.** *If the Hypothesis 3 is true, therefore when  $L \neq NL$ , then  $P = NP$ .*

**Proof.** We can simulate the computation  $M(w, u) = y$  in the Hypothesis 3 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  $u$  [19]. 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 [19].

If we suppose that  $L \subset 1NL$ , then we can accept the elements of the language  $L_1 \in L$  by a nondeterministic one-way logarithmic space Turing machine  $M'$ . In this way, there is a nondeterministic logarithmic space Turing machine  $M''(w) = M'(N(w))$  which will accept when  $w \in L_2$ . Consequently,  $M''$  is a nondeterministic logarithmic space Turing machine which decides the language  $L_2$ . The reason is because we can simulate the output string of  $N(w)$  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 [19]. Certainly, we do not need to reset the computation of  $N(w)$  for the reading at once of a symbol in the output string of  $N(w)$  by the nondeterministic one-way logarithmic space Turing machine  $M'$ . Therefore,  $L_2$  is in  $NL$  and thus,  $L_2 \in P$  due to  $NL \subseteq P$  [19]. If any single  $NP$ -complete problem can be solved in polynomial time, then  $P = NP$  [7]. Since  $L_2 \in P$  and  $L_2 \in NP$ -complete, then we obtain the complexity class  $P$  is equal to  $NP$  under the assumption that  $L \subset 1NL$ .

Hartmanis and Mahaney have also shown with their result that if  $1NL \subseteq L$  or even  $1NL \subset L$ , then  $L = NL$ , because they proved there is a complete problem for both  $1NL$  and  $NL$  at the same time [12]. If this way, if  $L \neq NL$ , then  $L \subset 1NL$  by contraposition [19]. Since we already obtained that  $P = NP$  under the assumption that  $L \subset 1NL$ , therefore if  $L \neq NL$ , then  $P = NP$ . ◀

### 2.3 Second Hypothesis

An important complexity class is *coNP-complete* [10]. 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* [7]. Moreover, if  $L_1 \in coNP$ , then  $L_1 \in coNP$ -complete [7]. A principal *coNP-complete* problem is *UNSAT* [10]. A Boolean formula without any satisfying truth assignment is unsatisfiable. The problem *UNSAT* asks whether a given Boolean formula is unsatisfiable [10].

*coNL* is the complexity class containing the languages such that their complements belong to  $NL$  [19]. 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 [19]. For example, a truth assignment for a Boolean formula  $\phi$  in  $3CNF$  is appropriated for the instance when every possible variable in  $\phi$  could be evaluated in that truth assignment string, but we cannot affirm the same for every possible binary string.

► **Definition 5.** 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  $p : \mathbb{N} \rightarrow \mathbb{N}$  such that for every  $x \in \{0, 1\}^*$ :

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

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  $|\dots|$  is the bit-length function [3].  $M$  is called a logarithmic space disqualifier.

We state the following Hypothesis:

▷ **Hypothesis 6.** Given a nonempty language  $L_1 \in 1NL$ , there is a language  $L_2$  in  $coNP$ -complete with a deterministic Turing machine  $M$ , where:

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

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 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 7.** If the Hypothesis 6 is true, therefore when  $L = NL$ , then  $P = NP$ .

**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  $u$ . 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 [19]. 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  $M'$ . 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  $coNL \subseteq P$  [19]. If any single  $coNP$ -complete problem can be solved in polynomial time, then  $P = NP$  [19]. Since  $L_2 \in P$  and  $L_2 \in coNP$ -complete, then we obtain the complexity class  $P$  is equal to  $NP$  under the assumption that  $L = NL$ . ◀

## 3 Results

### 3.1 First Hypothesis

We show a previous known  $NP$ -complete problem:

► **Definition 8. NAE 3SAT**

*INSTANCE:* A Boolean formula  $\phi$  in 3CNF.

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

*REMARKS:* NAE 3SAT  $\in$  NP-complete [10].

We define a new problem:

► **Definition 9. MAXIMUM EXCLUSIVE-OR 2-SATISFIABILITY**

*INSTANCE:* A positive integer  $K$  and a Boolean formula  $\phi$  that is an instance of XOR 2SAT.

*QUESTION:* Is there a truth assignment in  $\phi$  such that at most  $K$  clauses are unsatisfied?

*REMARKS:* We denote this problem as  $MAX \oplus 2SAT$ .

► **Theorem 10.**  $MAX \oplus 2SAT \in NP$ -complete.

**Proof.** It is trivial to see  $MAX \oplus 2SAT \in NP$  [19]. Given a Boolean formula  $\phi$  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  $\phi$ , 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  $\psi$  as the conjunction of the  $P_i$  formulas for every clause  $c_i$  in  $\phi$ , such that  $\psi = P_1 \wedge \dots \wedge P_m$ . Finally, we obtain that:

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

Consequently, we prove  $\text{NAE 3SAT} \leq_p \text{MAX} \oplus 2SAT$  where we already know the language  $\text{NAE 3SAT} \in NP$ -complete [10]. 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 11.** There is a deterministic Turing machine  $M$ , where:

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

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  $(\psi, K)$  for  $MAX \oplus 2SAT$  when  $\psi$  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  $\psi$  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  $(\psi, K)$  for  $MAX \oplus 2SAT$  just creating another instance  $\phi$  for XOR 2SAT where the Boolean formula  $\phi$  does not contain the  $K$  different indexed clauses  $\psi$  represented by the numbers in  $A$ . Therefore, we obtain the array  $A$  would be valid according to the Theorem 11 when:

$$(\psi, K) \in \text{MAX} \oplus 2SAT \text{ if and only if } \phi \in \text{XOR 2SAT}.$$



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  $\psi$  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  $\psi$  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  $A[i] = \text{undefined}$ . 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  $\max - 1 < \min$ .

---

**Algorithm 1** Logarithmic space verifier

---

```

1: /*A valid instance for  $MAX \oplus 2SAT$  with its certificate*/
2: procedure VERIFIER(( $\psi, K$ ),  $A$ )
3:   /*Initialize minimum and maximum values*/
4:    $\min \leftarrow 1$ 
5:    $\max \leftarrow 0$ 
6:   /*Iterate for the elements of the certificate array  $A$ */
7:   for  $i \leftarrow 1$  to  $K + 1$  do
8:     if  $i = K + 1$  then
9:       /*There exists a  $K + 1$  element in the array*/
10:      if  $A[i] \neq \text{undefined}$  then
11:        /*Reject the certificate*/
12:        return "no"
13:      end if
14:      /* $m$  is the number of clauses in  $\psi$ */
15:       $\max \leftarrow m + 1$ 
16:    else if  $A[i] = \text{undefined} \vee A[i] \leq \max \vee A[i] < 1 \vee A[i] > m$  then
17:      /*Reject the certificate*/
18:      return "no"
19:    else
20:       $\max \leftarrow A[i]$ 
21:    end if
22:    /*Iterate for the clauses of the Boolean formula  $\psi$ */
23:    for  $j \leftarrow \min$  to  $\max - 1$  do
24:      /*Output the indexed  $j^{th}$  clause in  $\psi$ */
25:      output " $\wedge c_j$ "
26:    end for
27:     $\min \leftarrow \max + 1$ 
28:  end for
29: end procedure

```

---

► **Theorem 12.** *The Hypothesis 3 is true.*

**Proof.** This is a consequence of Theorems 10 and 11.



### 3.2 Second Hypothesis

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

► **Definition 13. 3UNSAT**

*INSTANCE:* A Boolean formula  $\phi$  in 3CNF.

*QUESTION:* Is  $\phi$  unsatisfiable?

*REMARKS:* 3UNSAT  $\in$  *coNP*-complete [10].

We define a new problem:

► **Definition 14. 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 15. 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 bit-length, 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 16. There is a deterministic Turing machine  $M$ , where:**

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

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  $\phi$  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  $\phi$  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  $\phi$  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  $\phi$ . Therefore, we obtain that all the appropriated array  $A$  would be valid according to the Theorem 16 when:

$$\phi \in 3UNSAT \text{ if and only if } C \in SUM-ZERO$$

since we assume the positive and negated literals of some variable in the input  $\phi$  correspond to a positive integer and its negative value. 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  $\phi$  when we write the final integers that represent these literals to the output. This logarithmic space disqualification will be the Algorithm 2. We assume whether a value does not exist in the array  $A$  into the cell of some position  $i$  when  $A[i] = \text{undefined}$ . 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  $\phi$  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  $|\dots|$  is the bit-length function. ◀

► **Theorem 17.** *The Hypothesis 6 is true.*

**Proof.** This is a consequence of Theorems 15 and 16. ◀

### 3.3 Consequences

► **Theorem 18.** *If  $L \neq NL$  then  $P = NP$ .*

**Proof.** This is a direct consequence of Theorems 4 and 12. ◀

► **Theorem 19.** *If  $L = NL$  then  $P = NP$ .*

**Proof.** This is a direct consequence of Theorems 7 and 17. ◀

► **Theorem 20.**  $P = NP$ .

**Proof.** Since we have either  $L \neq NL$  or  $L = NL$  is true, then the complexity class  $P$  is equal to  $NP$ . ◀

► **Theorem 21.**  $NL = NP$ .

**Proof.** Since the polynomial time reduction in Theorem 10 could be easily transformed in a logarithmic space reduction, then the  $NP$ -complete problem in Theorem 4, that would be  $MAX \oplus 2SAT$ , is necessarily in  $NL$  and thus all the problems in  $NP$  when  $L \neq NL$ , because of the Cook's Theorem can also be transformed in a logarithmic space reduction [10]. For the same reason, the  $coNP$ -complete problem in Theorem 7, that would be  $3UNSAT$ , is necessarily in  $coNL$  and thus all the problems in  $coNP$  when  $L = NL$ . Since we have either  $L \neq NL$  or  $L = NL$  is true, then the complexity class  $NL$  is equal to  $NP$ . ◀

**Algorithm 2** Logarithmic space disqualifier

---

```

1: /*A valid instance for 3UNSAT with its disqualification*/
2: procedure DISQUALIFIER( $\phi, A$ )
3:   /*Initialize an index*/
4:    $j \leftarrow 0$ 
5:   /* $m$  is the number of clauses in  $\phi^*$ */
6:   /*Iterate for the elements of the disqualification array  $A^*$ */
7:   for  $i \leftarrow 1$  to  $m + 1$  do
8:     if  $i = m + 1$  then
9:       /*There exists an  $m + 1$  element in the array*/
10:      if  $A[i] \neq \text{undefined}$  then
11:        /*Reject the disqualification*/
12:        return “no”
13:      end if
14:      /*Break the for loop*/
15:      break
16:    else if  $A[i] = \text{undefined} \vee A[i] < 1 \vee A[i] > 3$  then
17:      /*Reject the disqualification*/
18:      return “no”
19:    else
20:       $j \leftarrow A[i]$ 
21:    end if
22:    /*From the indexed  $i^{\text{th}}$  clause  $c_i = (x_j \vee y_k \vee z_r)$  in  $\phi^*$ */
23:    /*Where  $x, y$  and  $z$  are literals with local indexes  $\{j, k, r\} = \{1, 2, 3\}$  in  $c_i^*$ */
24:    /*Output the integer representation of the  $j^{\text{th}}$  literal, that is  $n(x_j)^*$ */
25:    /*Filled with zeroes to the left until a total of  $|3 \times m|$  bits including the literal*/
26:    /*But, the bit-length of the symbol minus is ignored in filling the negated literals*/
27:    output “ ,  $n(x_j)$ ”
28:  end for
29: 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 [10]. 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:

1. 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 [13], 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.
2. 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 [10]. 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 [4].
3. Since all the *NP-complete* optimization problems become easy, everything will be much more efficient [9]. Transportation of all forms will be scheduled optimally to move people and goods around quicker and cheaper [9]. Manufacturers can improve their production to increase speed and create less waste [9]. Learning becomes easy by using the principle of Occam's razor: We simply find the smallest program consistent with the data [9]. Near perfect vision recognition, language comprehension and translation and all other learning tasks become trivial [9]. We will also have much better predictions of weather and earthquakes and other natural phenomenon [9].
4. There would be disruption, including maybe displacing programmers [14]. The practice of programming itself would be more about gathering training data and less about writing code [14]. Google would have the resources to excel in such a world [14].
5. 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].

We also prove that  $NL = NP$ . There are several important consequences from this result, but we will only mention one of the principal outcomes, that is the Theorem Buhrman-Fortnow-van Melkebeek-Torenvliet [5]. A set  $A$  is auto-reducible if there exists an oracle polynomial time Turing machine  $M$  such that  $L(M^A) = A$  with the restriction that for all  $x$ ,  $M^A(x)$  does not query whether  $x$  is in  $A$  [5]. When  $NL = NP$ , Buhrman, Fortnow, van Melkebeek and Torenvliet created a series of constructions to get an  $A$  such that:  $A$  is in *EXSPACE*,  $A$  is Turing-hard for *EXSPACE* and  $A$  "diagonalizes" against all possible auto-reductions [5].

---

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 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.
- 5 Harry Buhrman, Lance Fortnow, Dieter van Melkebeek, and Leen Torenvliet. Separating Complexity Classes Using Autoreducibility. *SIAM J. Comput.*, 29(5):1497–1520, March 2000. doi:10.1137/S0097539798334736.
- 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 Lance Fortnow. The Status of the P Versus NP Problem. *Commun. ACM*, 52(9):78–86, September 2009. doi:10.1145/1562164.1562186.
- 10 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.
- 11 William I. Gasarch. Guest column: The second  $P \stackrel{?}{=} NP$  poll. *ACM SIGACT News*, 43(2):53–77, 2012. doi:10.1145/2261417.2261434.
- 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 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.
- 14 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.
- 15 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.
- 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 Omer Reingold. Undirected Connectivity in Log-space. *J. ACM*, 55(4):1–24, September 2008. doi:10.1145/1391289.1391291.
- 21 Michael Sipser. *Introduction to the Theory of Computation*, volume 2. Thomson Course Technology Boston, 2006.