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. The answer P = NP is considered as a very unlikely event. However, we prove the complexity class P 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 [18]. Interactive proofs introduced by Goldwasser, Micali and Rackoff and Babi can be viewed as a model of the verification process [18]. 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 [18]. In addition, Blum and Kannan have studied another model where the goal is to check a computation based solely on the final answer [18]. More about probabilistic logarithmic space verifiers and the complexity class NP has been investigated on a technique of Lipton [18]. 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 [22]. 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 $P \neq NP$ [21]. Indeed, if $P = NP$ then there are stunning practical consequences [21]. 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

In 1936, Turing developed his theoretical computational model [24]. The deterministic and nondeterministic Turing machines have become in two of the most important definitions related to this theoretical model for computation [24]. A deterministic Turing machine has only one next action for each step defined in its program or transition function [24]. 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 [24].

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 $M(w) =$ "yes" [3]. Note that M fails to accept w either if this computation ends in the rejecting state, that is $M(w) =$ "no", or if the computation fails to terminate, or the computation ends in the halting state with some output, that is $M(w) = y$ (when M outputs the string y on the input w) [3].

Another relevant advance in the last century has been the definition of a complexity class. A language over an alphabet is any set of strings made up of symbols from that alphabet [7]. A complexity class is a set of problems, which are represented as a language, grouped by measures such as the running time, memory, etc [7]. The language accepted by a Turing machine M, denoted $L(M)$, has an associated alphabet Σ and is defined by:

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

Moreover, $L(M)$ is decided by M, when $w \notin L(M)$ if and only if $M(w) =$ "no" [7]. We denote by $t_M(w)$ the number of steps in the computation of M on input w [3]. For $n \in \mathbb{N}$ we denote by $T_M(n)$ the worst case run time of M; that is:

$$T_M(n) = max\{t_M(w) : w \in \Sigma^n\}$$

where Σ^n is the set of all strings over Σ of length n [3]. We say that M runs in polynomial time if there is a constant k such that for all n, $T_M(n) \leq n^k + k$ [3]. In other words, this means the language $L(M)$ can be decided by the Turing machine M in polynomial time. Therefore, P is the complexity class of languages that can be decided by deterministic Turing machines in polynomial time [7]. A verifier for a language L_1 is a deterministic Turing machine M, where:

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

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 [21]. If NP is the class of problems that have succinct certificates, then the complexity class $coNP$ must contain those problems that have succinct disqualifications [21]. That is, a "*no*" instance of a problem in $coNP$ possesses a short proof of its being a "*no*" instance [21].

▶ **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 [24]. 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* [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 [24]. The work tapes may contain at most $O(\log n)$ symbols [24]. In computational complexity theory, L is the complexity class containing those decision problems that can be decided by a deterministic logarithmic space Turing machine [21]. NL is the complexity class containing the decision problems that can be decided by a nondeterministic logarithmic space Turing machine [21].

A logarithmic space transducer is a Turing machine with a read-only input tape, a write-only output tape, and read/write work tapes [24]. The work tapes must contain at most $O(\log n)$ symbols [24]. 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 [24]. We call f a logarithmic space computable function [24]. 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 $x \in \{0,1\}^*$:

$x \in L_1$ *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 [21]. 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 [21]. 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 [20]. 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], [23].

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

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

where by $M(x,u)$ we denote the computation of M where x is placed on its input tape and the certificate 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* [11]. 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 $UNSAT$ [11]. A Boolean formula without any satisfying truth assignment is unsatisfiable. The problem $UNSAT$ asks whether a given Boolean formula is unsatisfiable [11].

$coNL$ is the complexity class containing the languages such that their complements belong to NL [21]. 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 [21].

▶ **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 $x \in \{0,1\}^*$:*

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

where by $M(x,u)$ we denote the computation of M where x is placed on its input tape and the 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.

3 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 [17]. 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 [14]. 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 [14]. Furthermore, they have proven that $1NL \subseteq L$ if and only if $L = NL$ [14].

We state the following Hypothesis:

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

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

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 5.** *A language L_1 is in coNL if and only if L_1 complies with there is a nondeterministic logarithmic space Turing machine M_1, such that for every appropriate instance x, then all the paths on $M_1(x)$ accept when $x \in L_1$ or at least one of the paths on $M_1(x)$ rejects when $x \notin L_1$.*

Proof. The proof is simple, since we can interchange the acceptance and rejection states of M_1 into a new Turing machine M_1', such that $M_1'(x) =$ "*yes*" when $M_1(x) =$ "*no*" and $M_1'(x) =$ "*no*" when $M_1(x) =$ "*yes*". Certainly, the language $L(M_1')$ is the complement of L_1 and it is in NL according to the definition of NL and because of the string x is an appropriate instance [21]. ◀

▶ **Theorem 6.** *When the Hypothesis 4 is true, then $P = NP$.*

Proof. We can simulate the computation $M(w, u) = y$ in the Hypothesis 4 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 [21]. 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 [21]. We remove each symbol generated in the work tapes, when we try to generate the next symbol contiguous to the right on the string u. In this way, the generation will always be in logarithmic space. In addition, we can guarantee that the generation of symbols in the work tapes always produces an appropriated string u, since we can check this by symbol per symbol from the generated string u.

Since we have that $L_1 \in 1NL$, then we can accept the elements of the language L_1 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 $w \in L_2$ for all the possible paths and will reject when $w \notin L_2$ for at least one of the paths.

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 [21]. 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'. Actually, the logarithmic space reduction is possible, because of M' is in one way. Indeed, it is not necessary to reset the computation of N in the composition $M'(N(w))$ on the input w, because M' never moves to the left the head on the input tape (that would be the output tape of N).

As a consequence of Theorem 5, then L_2 would be in $coNL$ and thus, $L_2 \in P$ due to $coNL \subseteq P$ [21]. If any single *coNP–complete* problem can be solved in polynomial time, then $P = NP$ [21]. 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 the Hypothesis 4 is true. ◀

4 Results

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

▶ **Definition 7.** ***3UNSAT***
INSTANCE: A Boolean formula ϕ in $3CNF$.
QUESTION: Is ϕ unsatisfiable?
REMARKS: $3UNSAT \in$ coNP–complete [11].

We define a new problem:

▶ **Definition 8.** ***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 9.** *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 10.** *There is a deterministic Turing machine M, where:*

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

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 10 when:

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

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. ◀

▶ **Theorem 11.** *The Hypothesis 4 is true.*

Proof. This is a direct consequence of Theorems 9 and 10. ◀

▶ **Theorem 12.** $P = NP$.

Proof. This is a direct consequence of Theorems 6 and 11. ◀

5 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 [15], symmetric ciphers [19] 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 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

Algorithm 1 Logarithmic space disqualifier

```
1: /*A valid instance for 3UNSAT with its disqualification*/
2: procedure DISQUALIFIER(φ, A)
3:     /*Initialize an index*/
4:     j ← 0
5:     /*m is the number of clauses in φ*/
6:     /*Iterate for the elements of the disqualification array A*/
7:     for i ← 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] ≠ undefined then
11:                /*Reject the disqualification*/
12:                return "no"
13:            end if
14:            /*Break the for loop*/
15:            break
16:        else if A[i] = undefined ∨ A[i] < 1 ∨ A[i] > 3 then
17:            /*Reject the disqualification*/
18:            return "no"
19:        else
20:            j ← A[i]
21:        end if
22:        /*From the indexed i^th clause c_i = (x_j ∨ y_k ∨ z_r) in φ*/
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^th literal, that is n(x_j)*/
25:        /*Filled with zeroes to the left until a total of |3 × 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
```

[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 [16]. The practice of programming itself would be more about gathering training data and less about writing code [16]. Google would have the resources to excel in such a world [16].

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. Clay Mathematics Institute. http://www.claymath.org/sites/default/files/pvsnp.pdf. Accessed 31 December 2019.

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. Woeginger Home Page. https://www.win.tue.nl/~gwoegi/P-versus-NP/Deolalikar.pdf. Accessed 31 December 2019.

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 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.

15 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.

16 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.

17 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.

18 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.

19 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.

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

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

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

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

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