P versus NP

Frank Vega
CopSonic, 1471 Route de Saint-Nauphary 82000 Montauban, France
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. 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. Another major complexity class is *P-Sel*. *P-Sel* is the class of decision problems for which there is a polynomial time algorithm (called a selector) with the following property: Whenever it's given two instances, a "yes" and a "no" instance, the algorithm can always decide which is the "yes" instance. It is known that if NP is contained in *P-Sel*, then $P = NP$.

In this paper we consider the problem of computing the sum of the weighted densities of states of a Boolean formula in $3CNF$. Given a Boolean formula ϕ, the density of states $n(E)$ counts the number of truth assignments that leave exactly E clauses unsatisfied in ϕ. The weighted density of states $m(E)$ is equal to $E \times n(E)$. The sum of the weighted densities of states of a Boolean formula in $3CNF$ with m clauses is equal to $\sum_{E=0}^{m} m(E)$. We prove that we can calculate the sum of the weighted densities of states in polynomial time. The lowest value of E with a non-zero density (i.e. $min_E\{E|n(E) > 0\}$) is the solution of the corresponding *MAX-SAT* problem.

The minimum lowest value with a non-zero density from the two formulas ϕ_1 and ϕ_2 is equal to the minimum value between E_1 and E_2, where E_i is the lowest value with a non-zero density of ϕ_i for $i \in \{1, 2\}$. Given two Boolean formulas ϕ_1 and ϕ_2 in $3CNF$ with n variables and m clauses, the combinatorial optimization problem *SELECTOR-3SAT* consists in selecting the formula which has the minimum lowest value with a non-zero density, where every clause from ϕ_1 and ϕ_2 can be unsatisfied for some truth assignment. We assume that the formula with the minimum lowest value with a non-zero density has the minimum sum of the weighted densities of states. In this way, we solve *SELECTOR-3SAT* with an exact polynomial time algorithm. We claim that this could be used for a possible selector of $3SAT$ and thus, $P = NP$.

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

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

1 Introduction

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

The $P = NP$ question is also singular in the number of approaches that researchers have brought to bear upon it over the years [7]. From the initial question in logic, the focus moved to complexity theory where early work used diagonalization and relativization techniques [7]. It was showed that these methods were perhaps inadequate to resolve P versus NP by demonstrating relativized worlds in which $P = NP$ and others in which $P \neq NP$ [3]. This shifted the focus to methods using circuit complexity and for a while this approach was deemed the one most likely to resolve the question [7]. Once again, a negative result showed that a class of techniques known as "Natural Proofs" that subsumed the above could not separate the classes NP and P, provided one-way functions exist [15]. There has been speculation that resolving the $P = NP$ question might be outside the domain of mathematical techniques [7]. More precisely, the question might be independent of standard axioms of set theory [7]. Some results have showed that some relativized versions of the $P = NP$ question are independent of reasonable formalizations of set theory [11].

In 1936, Turing developed his theoretical computational model [17]. The deterministic and nondeterministic Turing machines have become in two of the most important definitions related to this theoretical model for computation [17]. A deterministic Turing machine has only one next action for each step defined in its program or transition function [17]. 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 [17]. Another relevant advance in the last century has been the definition of a complexity class. A language over an alphabet is any set of strings made up of symbols from that alphabet [6]. A complexity class is a set of problems, which are represented as a language, grouped by measures such as the running time, memory, etc [6]. NP is the complexity class which contains those languages that can be decided in polynomial time by nondeterministic Turing machines.

A major complexity class is *Sharp-P* (denoted as $\#P$) [18]. This can be defined by the class of function problems of the form "*compute* $f(x)$", where f is the number of accepting paths of a nondeterministic Turing machines, where this machine always accepts in polynomial time [18]. In previous years there has been great interest in the verification or checking of computations [13]. Interactive proofs introduced by Goldwasser, Micali and Rackoff and Babi can be viewed as a model of the verification process [13]. Dwork and Stockmeyer and Condon have studied interactive proofs where the verifier is a space bounded computation instead of the original model where the verifier is a time bounded computation [13]. In addition, Blum and Kannan have studied another model where the goal is to check a computation based solely on the final answer [13]. More about probabilistic logarithmic space verifiers and the complexity class NP has been investigated on a technique of Lipton [13]. We show some results about the logarithmic space verifiers applied to a problem in the class $\#P$.

A set $L_1 \subseteq \{0,1\}^*$ is defined to be p-selective if there is a function $f : \{0,1\}^* \times \{0,1\}^* \to \{0,1\}^*$ so that

- f is computable in polynomial time,
- $f(x, y) = x$ or $f(x, y) = y$,
- $x \in L_1$ or $y \in L_1$ implies that $f(x, y) \in L_1$.

The function f is a selector for L_1. *P-Sel* is the class of decision problems defined on languages which are p-selective [16]. We show a possible selector for some *NP-complete* problem and thus, $P = NP$ [16]. Indeed, we prove there is a combinatorial optimization problem *SELECTOR-3SAT* that could be used for a selector of $3SAT$. We claim that given two Boolean formulas ϕ_1 and ϕ_2 in $3CNF$ with n variables and m clauses such that this tuple of ϕ_1 and ϕ_2 consists in a pair of a satisfiable and an unsatisfiable formula, then the problem *SELECTOR-3SAT* can always select the satisfiable formula. Certainly, we could extend this to use it for every pair of Boolean formulas ϕ_1 and ϕ_2 in $3CNF$ with not necessarily the same amount of variables and clauses.

2 Materials & Methods

2.1 Polynomial time verifiers

Let Σ be a finite alphabet with at least two elements, and let Σ^* be the set of finite strings over Σ [2]. A Turing machine M has an associated input alphabet Σ [2]. For each string w in Σ^* there is a computation associated with M on input w [2]. We say that M accepts w if this computation terminates in the accepting state, that is $M(w) = $ "*yes*" [2]. 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) [2].

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*" [6]. We denote by $t_M(w)$ the number of steps in the computation of M on input w [2]. 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 [2]. We say that M runs in polynomial time if there is a constant k such that for all n, $T_M(n) \leq n^k + k$ [2]. In other words, this means the language $L(M)$ can be decided by the Turing machine M in polynomial time. Therefore, P is the complexity class of languages that can be decided by deterministic Turing machines in polynomial time [6]. A verifier for a language L_1 is a deterministic Turing machine M, where:

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

We measure the time of a verifier only in terms of the length of w, so a polynomial time verifier runs in polynomial time in the length of w [2]. 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 also the complexity class of languages defined by polynomial time verifiers [14].

A decision problem in NP can be restated in this way: There is a string c with $M(w, c) = $ "*yes*" if and only if $w \in L_1$, where L_1 is defined by the polynomial time verifier M [14]. The function problem associated with L_1, denoted FL_1, is the following computational problem: Given w, find a string c such that $M(w, c) = $ "*yes*" if such string exists; if no such string exists, then reject, that is, return "*no*" [14]. The complexity class of all function problems associated with languages in NP is called FNP [14]. FP is the complexity class that contains those problems in FNP which can be solved in polynomial time [14].

To attack the P versus NP question the concept of *NP-completeness* has been very useful [9]. The equivalent definition for the function problems is the *FNP-completeness* [14]. A principal *NP-complete* problem is SAT [9]. An instance of SAT is a Boolean formula ϕ which is composed of:

1. Boolean variables: $x_1, x_2, \ldots, x_n$;

2. Boolean connectives: Any Boolean function with one or two inputs and one output, such as $\wedge$(AND), $\vee$(OR), $\neg$(NOT), $\Rightarrow$(implication), $\Leftrightarrow$(if and only if);
3. and parentheses.

A truth assignment for a Boolean formula ϕ is a set of values for the variables in ϕ. On the one hand, a satisfying truth assignment is a truth assignment that causes ϕ to be evaluated as true. On the other hand, a truth assignment that causes ϕ to be evaluated as false is a unsatisfying truth assignment. A Boolean formula with a satisfying truth assignment is satisfiable and without any satisfying truth assignment is unsatisfiable. The problem SAT asks whether a given Boolean formula is satisfiable [9].

A literal in a Boolean formula is an occurrence of a variable or its negation [6]. A Boolean formula is in conjunctive normal form, or CNF, if it is expressed as an AND of clauses, each of which is the OR of one or more literals [6]. Given a CNF formula, *MAX-SAT* consists in finding a truth assignment that satisfies the largest number of clauses. *MAX-SAT* belongs to *FNP-complete* [14]. A Boolean formula is in 3-conjunctive normal form or $3CNF$, if each clause has exactly three distinct literals [6].

For example, the Boolean formula:

$$(x_1 \vee \neg x_1 \vee \neg x_2) \wedge (x_3 \vee x_2 \vee x_4) \wedge (\neg x_1 \vee \neg x_3 \vee \neg x_4)$$

is in $3CNF$. The first of its three clauses is $(x_1 \vee \neg x_1 \vee \neg x_2)$, which contains the three literals x_1, $\neg x_1$, and $\neg x_2$. Another relevant *NP-complete* language is $3CNF$ satisfiability, or $3SAT$ [6]. In $3SAT$, it is asked whether a given Boolean formula ϕ in $3CNF$ is satisfiable.

An important complexity is *Sharp-P* (denoted as $\#P$) [18]. We can also define the class $\#P$ using polynomial time verifiers. Let $\{0,1\}^*$ be the infinite set of binary strings, a function $f : \{0,1\}^* \to \mathbb{N}$ is in $\#P$ if there exists a polynomial time verifier M such that for every $x \in \{0,1\}^*$,

$$f(x) = |\{y : M(x,y) = \text{“}yes\text{”}\}|$$

where $|\cdots|$ denotes the cardinality set function [2]. *#P-complete* is another complexity class. A problem is *#P-complete* if and only if it is in $\#P$, and every problem in $\#P$ can be reduced to it by a polynomial time counting reduction [14].

2.2 Logarithmic space verifiers

A logarithmic space Turing machine has a read-only input tape, a write-only output tape, and read/write work tapes [17]. The work tapes may contain at most $O(\log n)$ symbols [17]. In computational complexity theory, L is the complexity class containing those decision problems that can be decided by a deterministic logarithmic space Turing machine [14]. NL is the complexity class containing the decision problems that can be decided by a nondeterministic logarithmic space Turing machine [14].

A logarithmic space transducer is a Turing machine with a read-only input tape, a write-only output tape, and read/write work tapes [17]. The work tapes must contain at most $O(\log n)$ symbols [17]. 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 [17]. We call f a logarithmic space computable function [17]. 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 [14].

We can give a certificate-based definition for NL [2]. The certificate-based definition of NL assumes that a logarithmic space Turing machine has another separated read-only tape [2]. On each step of the machine, the machine's head on that tape can either stay in place or move to the right [2]. In particular, it cannot reread any bit to the left of where the head currently is [2]. For that reason, this kind of special tape is called "read-once" [2].

▶ **Definition 1.** *A language L_1 is in NL if there exists a deterministic logarithmic space Turing machine M with an additional special read-once input tape polynomial $p : \mathbb{N} \to \mathbb{N}$ such that for every $x \in \{0,1\}^*$:*

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

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 $[\ldots]$ is the bit-length function [2]. M is called a logarithmic space verifier [2].

An interesting complexity class is *Sharp-L* (denoted as $\#L$). $\#L$ has the same relation to L as $\#P$ does to P [1]. We can define the class $\#L$ using logarithmic space verifiers as well.

▶ **Definition 2.** *Let $\{0,1\}^*$ be the infinite set of binary strings, a function $f : \{0,1\}^* \to \mathbb{N}$ is in $\#L$ if there exists a logarithmic space verifier M such that for every $x \in \{0,1\}^*$,*

$$f(x) = |\{u : M(x,u) = \text{“yes”}\}|$$

where $|\cdots|$ denotes the cardinality set function [1].

The two-way Turing machines may move their head on the input tape into two-way (left and right directions) while the one-way Turing machines are not allowed to move the head on the input tape to the left [12]. Hartmanis and Mahaney have investigated the classes $1L$ and $1NL$ of languages recognizable by deterministic one-way logarithmic space Turing machine and nondeterministic one-way logarithmic space Turing machine, respectively [12].

▶ **Lemma 3.** *NL is closed under nondeterministic logarithmic space reductions to every language in $1NL$.*

Proof. Suppose, we have two languages L_1 and $L_2 \in 1NL$, such that there is a nondeterministic logarithmic space Turing machine M which makes a reduction from $x \in L_1$ into $M(x) \in L_2$. Besides, we assume there is a nondeterministic one-way logarithmic space Turing machine M' which decides L_2. Hence, we only need to prove that $M'(M(x))$ is a nondeterministic logarithmic space Turing machine. The solution to this problem is simple: We do not explicitly store the output result of M in the work tapes of M'. Instead, whenever M' needs to move the head on the input tape (this tape will be the output tape of M), then we continue the computation of M on input x long enough for it to produce the new output symbol; this is the symbol that will be the next scanned symbol on the input tape of M'.

If M' only needs to read currently from the work tapes, then we just pause the computation of M on the input x and continue the computation of M' until this needs to move to the right on the input tape. We can always continue the simulation, because M' never moves the head on the input tape to the left. We only accept when the machine M enters in the halting state and M' enters in the accepting state otherwise we reject. It is clear that this simulation indeed computes $M'(M(x))$ in a nondeterministic logarithmic space. In this way, we obtain $x \in L_1$ if and only if $M'(M(x)) =$ "*yes*" which is a clear evidence that L_1 is in NL. ◀

We can give an equivalent definition for NL, but this time the output is a string which belongs to a language in $1NL$.

▶ **Definition 4.** *A language L_1 is in NL if there exists another nonempty language $L_2 \in 1NL$ and a deterministic logarithmic space Turing machine M with an additional special read-once input tape polynomial $p : \mathbb{N} \to \mathbb{N}$ such that for every $x \in \{0,1\}^*$:*

$$x \in L_1 \Leftrightarrow \exists\ u \in \{0,1\}^{p([x])} \text{ such that } M(x,u) = y, \text{ where } y \in L_2$$

and by $M(x,u) = y$ we denote the computation of M where x is placed on its input tape, and y is the remaining string in the output tape on M after the halting state, 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 [2]. We call M a one-way logarithmic space verifier. This definition is still valid, because of Lemma 3.

According to the previous definition, we can redefine $\#L$ as follows:

▶ **Definition 5.** *Let $\{0,1\}^*$ be the infinite set of binary strings, a function $f : \{0,1\}^* \to \mathbb{N}$ is in $\#L$ if there exists another nonempty language $L_2 \in 1NL$, and a nondeterministic one-way logarithmic space Turing machine M' which decides L_2, and a one-way logarithmic space verifier M such that for every $x \in \{0,1\}^*$,*

$$f(x) = |\{(u,p) : M(x,u) = y, \text{ where } y \in L_2 \text{ and } p \text{ is an accepting path of } M'(y)\}|$$

and $|\cdots|$ denotes the cardinality set function. This definition is still valid under the result of Lemma 3.

3 Results

▶ **Definition 6.** *Given a Boolean formula ϕ, the density of states $n(E)$ counts the number of truth assignments that leave exactly E clauses unsatisfied in ϕ [8]. The weighted density of states $m(E)$ is equal to $E \times n(E)$. The sum of the weighted densities of states of a Boolean formula in $3CNF$ with m clauses is equal to $\sum_{E=0}^{m} m(E)$. The lowest value of E with a non-zero density (i.e. $min_E\{E|n(E) > 0\}$) is the solution of the corresponding MAX-SAT problem [8]. The minimum lowest value with a non-zero density from the two formulas ϕ_1 and ϕ_2 is equal to the minimum value between E_1 and E_2, where E_i is the lowest value with a non-zero density of ϕ_i for $i \in \{1,2\}$.*

We define a new problem:

▶ **Definition 7.** ***EXACTLY-THRICE***

INSTANCE: A unary string 0^q and a collection of binary strings, such that each element in the collection represents a power number in base 2 with a bit-length lesser than or equal to q. The collection of numbers is represented by an array N.

QUESTION: Is there an element repeated exactly thrice in N?

▶ **Theorem 8.** *EXACTLY-THRICE $\in 1NL$.*

ALGORITHM 1: *ONE-WAY-ALGO*

Data: $(0^q, N)$ where $(0^q, N)$ is an instance of *EXACTLY-THRICE*
Result: A nondeterministic acceptance or rejection in one-way logarithmic space

```
// Get the length of the unary string 0^q as a binary string
q ⟵ length(0^q);
// Generate nondeterministically an arbitrary integer between 1 and q
d ⟵ random(1, q);
t ⟵ 0;
// Initial position in N
i ⟵ 1;
while N[i] ≠ undefined do
    s ⟵ 0;
    // N[i][j] represents the j^th digit of the string in N[i]
    for j ⟵ 1 to q + 1 do
        if j = q + 1 then
            if N[i][j] ≠ undefined then
                // There exists an element with bit-length greater than q
                return "no";
            end
        end
        else if (j = 1 ∧ N[i][j] ≠ 1) ∨ (j > 1 ∧ N[i][j] = 1) ∨ N[i][j] ∉ {0, 1, undefined} then
            // The element N[i] is not a binary string
            return "no";
        end
        else if N[i][j] = undefined then
            // Break the current for loop statement
            break;
        end
        else
            // Store the current position of digit N[i][j] in N[i]
            s ⟵ s + 1;
        end
    end
    if s = d ∧ t < 4 then
        // The element N[i] is equal to 2^(d−1)
        t ⟵ t + 1;
    end
    i ⟵ i + 1;
end
if t = 3 then
    // The element 2^(d−1) appears exactly thrice in N
    return "yes";
end
else
    return "no";
end
```

Proof. Given an instance $(0^q, N)$ of *EXACTLY-THRICE*, then we can read its elements from left to right on the input tape, verify that every element in the collection is a binary string, and finally check whether every element in N has a bit-length lesser than or equal to q. In addition, we can nondeterministically pick a binary integer d between 1 and q and accept in case of there exists the number 2^{d-1} exactly thrice in N.

We can make all this computation in a nondeterministic one-way using logarithmic space. Certainly, the verification of the membership of 2^{d-1} in N could be done in logarithmic space, since it is trivial to check whether a binary string represents the power 2^{d-1}. Besides, we can store a logarithmic amount of symbols, because of d has an exponential more succinct representation in relation to the unary string 0^q [14]. Moreover, the variables that we could use for the iteration of the elements in N have a logarithmic space in relation to the length of the instance $(0^q, N)$.

We never need to move to the left on the input tape for the acceptance or rejection of the elements in *EXACTLY-THRICE* in a nondeterministic logarithmic space. We describe this nondeterministic one-way logarithmic space computation in the Algorithm 1. In this algorithm, we assume a value does not exist in the array N into the cell of some position i when $N[i] = undefined$. To sum up, we actually prove that *EXACTLY-THRICE* is in $1NL$. ◀

Let's consider a counting problem:

▶ **Definition 9.** ***#CLAUSES-3UNSAT***

INSTANCE: Two natural numbers n, m, and a Boolean formula ϕ in $3CNF$ of n variables and m clauses. The clauses are represented by an array C, such that C represents a set of m set elements, where $C[i] = S_i$ if and only if S_i is exactly the set of literals into a clause c_i in ϕ for $1 \leq i \leq m$. Besides, each variable in ϕ is represented by a unique integer between 1 and n. In addition, a negative or positive integer represents a negated or non-negated literal, respectively.

ANSWER: The sum of the weighted densities of states of the Boolean formula ϕ.

▶ **Theorem 10.** *#CLAUSES-3UNSAT* $\in FP$.

Proof. We are going to show there is a deterministic Turing machine M, where:

$$\#CLAUSES\text{-}3UNSAT = \{w : M(w, u) = y, \exists\ u \text{ such that } y \in EXACTLY\text{-}THRICE\}$$

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. Given an instance (n, m, C) of *#CLAUSES-3UNSAT*, we firstly check whether this instance has an appropriate representation according to the constraints introduced in the Definition 9. The constraints for the Definition 9 are the following ones:

1. The array C must contain exactly m sets and,
2. each variable must be represented by a unique integer between 1 and n and,
3. there are no two equals sets inside of C and finally,
4. every set element must contain exactly three distinct literals.

All these requirements are verified in the logarithmic space Algorithm 2, where this subroutine decides whether the instance has an appropriate representation according to the Definition 9. After that verification, we use a certificate as an array A, such that this consists in an array of n different integer numbers in ascending absolute value order. We read at once

ALGORITHM 2: *CHECK-ALGO*

Data: (n, m, C) where (n, m, C) is an instance of *#CLAUSES-3UNSAT*
Result: A logarithmic space subroutine

```
if n < 1 ∨ m < 1 then
    // n or m is not a natural number
    return "no";
end
for i ← 1 to m + 1 do
    if (i < m + 1 ∧ C[i] = undefined) ∨ (i = m + 1 ∧ C[i] ≠ undefined) then
        // C does not contain exactly m sets
        return "no";
    end
end
for i ← 1 to n do
    t ⟵ 0;
    foreach j ← 1 to m; C[j] = {x, y, z} do
        // abs(...) denotes the absolute value operation
        if abs(x) < 1 ∨ abs(y) < 1 ∨ abs(z) < 1 ∨ abs(x) > n ∨ abs(y) > n ∨ abs(z) > n then
            // C does not contain exactly n variables from 1 to n
            return "no";
        end
        if t = 0 ∧ (i = abs(x) ∨ i = abs(y) ∨ i = abs(z)) then
            // Store the existence of the variable i in C
            t ⟵ 1;
        end
    end
    if t = 0 then
        // C does not contain the variable i
        return "no";
    end
end
for i ← 1 to m − 1 do
    for j ← i + 1 to m do
        // ∩ denotes the intersection set operation
        if C[i] ∩ C[j] = C[i] then
            // The array C is not exactly a "set" of set elements
            return "no";
        end
        // |···| denotes the cardinality set function
        if |C[i]| + |C[j]| ≠ 6 then
            // C[i] or C[j] does not contain exactly three elements
            return "no";
        end
    end
end
// The instance (n, m, C) is appropriate for #CLAUSES-3UNSAT
return "yes";
```

ALGORITHM 3: *VERIFIER-ALGO*

```
Data: (n, m, C, A) where (n, m, C) is an instance of #CLAUSES-3UNSAT and A is a
      certificate
Result: A one-way logarithmic space verifier
if CHECK-ALGO(n, m, C) = "no" then
  | // (n, m, C) is not an appropriate instance of #CLAUSES-3UNSAT
  | return "no";
end
else
  | output 0^m;
end
// Minimum current variable during the iteration of the array A
x ⟵ 0;
for i ← 1 to n + 1 do
  | if i = n + 1 then
  |   | if A[i] ≠ undefined then
  |   |   | // The array A contains more than n elements
  |   |   | return "no";
  |   | end
  | end
  | else if A[i] = undefined ∨ abs(A[i]) < 1 ∨ abs(A[i]) > n ∨ abs(A[i]) ≤ x then
  |   | // The certificate A is not appropriate
  |   | return "no";
  | end
  | else
  |   | // abs(...) denotes the absolute value operation
  |   | x ⟵ abs(A[i]);
  |   | y ⟵ A[i];
  |   | for j ← 1 to m do
  |   |   | if y ∈ C[j] then
  |   |   |   | // Output the number 2^{j-1} when the set C[j] contains the literal y
  |   |   |   | output ,1;
  |   |   |   | if j − 1 > 0 then
  |   |   |   |   | output 0^{j-1};
  |   |   |   | end
  |   |   | end
  |   | end
  | end
end
```

the elements of the array A and we reject whether this is not an appropriate certificate: That is, when the absolute value of the numbers are not sorted in ascending order, or the array A does not contain exactly n elements, or the array A contains a number that its absolute value is not between 1 and n, since every variable is represented by an integer between 1 and n in C.

While we read each element x of the array A, then we copy the binary numbers 2^{j-1} that represent the sets $C[j]$ which contain the literal x just creating another instance $(0^q, N)$ of *EXACTLY-THRICE*, where the value of q is equal to m. Since the array A does not contain repeated elements, then we could correspond each certificate A to a truth assignment for ϕ with a representation of all the variables in ϕ, such that the literals in A are false. We know a set $C[j]$ that represents a clause is false if and only if the three literals in $C[i]$ are false. Therefore, the evaluation as false into the literals of the array A corresponds to a unsatisfying truth assignment in ϕ if and only if we write some number 2^{j-1} thrice to the output tape, where 2^{j-1} represents a set $C[j]$ for some $1 \leq j \leq m$.

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. Indeed, the variables that we could use for the iteration of the elements in A and C have a logarithmic space in relation to the length of the instance (n, m, C). Hence, we only need to iterate from the elements of the array A to verify whether the array is an appropriate certificate and write to the output tape the representation as a power of two of the sets in C that contain the literals in A. This logarithmic space verification will be the Algorithm 3. We assume whether a value does not exist in the arrays A or C into the cell of some position i when $A[i] = \textit{undefined}$ or $C[i] = \textit{undefined}$, respectively.

The Algorithm 3 is a one-way logarithmic space verifier, since this never moves the head on the special read-once tape to the left, where it is placed the certificate A. Moreover, for every unsatisfying truth assignment represented by the array A, then the output of this one-way logarithmic space verifier will always belong to the language *EXACTLY-THRICE*, where we know that *EXACTLY-THRICE* $\in 1NL$ as a consequence of Theorem 8. In addition, every appropriate certificate A is always polynomially bounded by the instance (n, m, C).

Consequently, we demonstrate that *#CLAUSES-3UNSAT* belongs to the complexity class $\#L$ under the Definition 5. Certainly, every truth assignment in ϕ corresponds to a single certificate in our one-way logarithmic space verifier. In addition, the number of accepting paths in the Algorithm 1 for the generated instance $(0^q, N)$ of *EXACTLY-THRICE* is exactly the number of unsatisfied clauses for a single truth assignment.

The number of accepting paths in the Algorithm 1 for a single instance is equal to the number of different powers of two that exist exactly thrice in the array N. Actually, this corresponds to the number of unsatisfied clauses for the truth assignment that represents the certificate A. We know that $\#L$ is contained in the class FP [1], [4], [2]. As result, $\#L$ remains in the class FP under the Definition 5 as a consequence of Lemma 3. In conclusion, we show that *#CLAUSES-3UNSAT* is indeed in FP. ◀

Let's consider an interesting problem:

▶ **Definition 11.** ***SELECTOR-3SAT***

INSTANCE: Two Boolean formulas ϕ_1 and ϕ_2 in $3CNF$ with n variables and m clauses, where every clause from ϕ_1 and ϕ_2 can be unsatisfied for some truth assignment. The clauses in the Boolean formula ϕ_j is represented by a set S_j, such that S_j represents a set of m set elements, where $S_{i,j} \in S_j$ if and only if $S_{i,j}$ is exactly the set of literals into a clause c_i in ϕ_j for $1 \leq i \leq m$ and $j \in \{1, 2\}$. Besides, each variable from the formulas ϕ_1 and ϕ_2 is *represented by a unique integer between* 1 *and* n *within the sets* S_1 *and* S_2*, respectively. In addition, a negative or positive integer represents a negated or non-negated literal, respectively.*

ANSWER: The formula that has the minimum lowest value with a non-zero density.

ALGORITHM 4: *SELECTOR-ALGO*

Data: (S_1, S_2) where (S_1, S_2) represents two Boolean formulas in $3CNF$ with n variables and m clauses

Result: A polynomial time algorithm

```
if (S1, S2) is not an appropriate instance of SELECTOR-3SAT then
  | return "no";
end
else if POLY-ALGO(S1) ≤ POLY-ALGO(S2) then
  | return S1;
end
else
  | return S2;
end
```

▶ **Theorem 12.** *SELECTOR-3SAT* $\in FP$.

Proof. Consider the Algorithm 4, where *POLY-ALGO* is a polynomial time algorithm for *#CLAUSES-3UNSAT*. Indeed, *POLY-ALGO* converts a set of clauses S in an appropriate instance of *#CLAUSES-3UNSAT* and solve it. We state that the Algorithm 4 solves *SELECTOR-3SAT*. Certainly, given two Boolean formulas ϕ_1 and ϕ_2 in $3CNF$ with n variables and m clauses, then they comply that the one which has the minimum lowest value with a non-zero density contains the minimum sum of the weighted densities of states, when every clause from the formulas ϕ_1 and ϕ_2 can be unsatisfied for some truth assignment.

Consequently, the formula which has the minimum lowest value with a non-zero density contains the truth assignment that satisfies the largest number of clauses from the truth assignments of the Boolean formulas ϕ_1 and ϕ_2. Indeed, the truth assignment that satisfies the largest number of clauses of every Boolean formula ϕ_j for $j \in \{1, 2\}$ complies that contains the smallest amount of unsatisfied clauses from all the truth assignments in ϕ_j. In addition, the truth assignment that satisfies the largest number of clauses in a Boolean formula ϕ_j for $j \in \{1, 2\}$ directly affects to the number of unsatisfied clauses within the other truth assignments in ϕ_j. Actually, if the truth assignment that satisfies the largest number of clauses in a Boolean formula ϕ_i contains less unsatisfied clauses than the truth assignment that satisfies the largest number of clauses in another Boolean formula ϕ_k, then the other truth assignments in ϕ_i contain less unsatisfied clauses than the other truth assignments in ϕ_k, where ϕ_i and ϕ_k contain the same amount of clauses and variables and every clause in ϕ_i and ϕ_k can be unsatisfied for some truth assignment.

However, the Boolean formula that contains less unsatisfied clauses from all the truth assignments complies that this has the minimum sum of the weighted densities of states. Certainly, if $n(E) > 0$ for some Boolean formula ϕ_i and $n(E) = 0$ for another Boolean formula ϕ_k, then there are more chances that the other values $n(E')$ in ϕ_i are greater than the values of $n(E')$ in ϕ_k as much as E' is close to E and the values of $n(E')$ in ϕ_i are lesser than the values of $n(E')$ in ϕ_k when E' is less close to E, where the tuple of ϕ_i and ϕ_k is an instance of *SELECTOR-3SAT*. The Algorithm 4 is computable in polynomial time due to Theorem 10. In this way, we show that *SELECTOR-3SAT* is in FP. ◀

4 Conclusions

No one has been able to find a polynomial time algorithm for any of more than 300 important known *NP-complete* problems [9]. 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 [5]. The consequences, both positive and negative, arise since various *NP-complete* problems are fundamental in many fields [5]. The combinatorial optimization problem *SELECTOR-3SAT* could be used for a possible selector of $3SAT$ [16]. In this way, we would prove that $P = NP$.

References

1 Carme Álvarez and Birgit Jenner. A Very Hard Log-Space Counting Class. *Theor. Comput. Sci.*, 107(1):3–30, January 1993. doi:10.1016/0304-3975(93)90252-O.

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

3 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.

4 Allan Borodin, Stephen A. Cook, and Nick Pippenger. Parallel Computation for Well-Endowed Rings and Space-Bounded Probabilistic Machines. *Inf. Control*, 58(1–3):113–136, July 1984. doi:10.1016/S0019-9958(83)80060-6.

5 Stephen A. Cook. The P versus NP Problem, April 2000. In Clay Mathematics Institute at http://www.claymath.org/sites/default/files/pvsnp.pdf. Retrieved June 26, 2020.

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

7 Vinay Deolalikar. P $\neq$ NP, 2010. In Woeginger Home Page at https://www.win.tue.nl/~gwoegi/P-versus-NP/Deolalikar.pdf. Retrieved June 26, 2020.

8 Stefano Ermon, Carla P. Gomes, and Bart Selman. Computing the Density of States of Boolean Formulas. In *Proceedings of the 16th International Conference on Principles and Practice of Constraint Programming*, pages 38–52, Berlin, Heidelberg, 2010. Springer-Verlag. doi:10.5555/1886008.1886016.

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

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

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

12 Juris Hartmanis and Stephen R. Mahaney. Languages Simultaneously Complete for One-Way and Two-Way Log-Tape automata. *SIAM Journal on Computing*, 10(2):383–390, 1981. doi:10.1137/0210027.

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

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

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

16 Alan L. Selman. P-Selective Sets, Tally Languages, and the Behavior of Polynomial Time Reducibilities on NP. In *Proceedings of the 6th Colloquium, on Automata, Languages and Programming*, pages 546–555, Berlin, Heidelberg, 1979. Springer-Verlag.

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

18 Leslie G. Valiant. The complexity of computing the permanent. *Theoretical Computer Science*, (2):189–201, 1979. doi:10.1016/0304-3975(79)90044-6.