A Solution of the P versus NP Problem

Frank Vega[1] [0000−0001−8210−4126]

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. A major complexity classes are L and $\oplus L$. A logarithmic Turing machine has a read-only input tape, a write-only output tape, and some read/write work tapes. The work tapes may contain at most $O(\log n)$ symbols. L is the complexity class containing those decision problems that can be decided by a deterministic logarithmic Turing machine. The complexity class $\oplus L$ has the same relation to L as $\oplus P$ does to P. We demonstrate there is a complete problem for $\oplus L$ that can be logarithmic space reduced to a problem in L. Consequently, we show $L = \oplus L$. To attack the P versus NP problem, the *NP–completeness* is a useful result. We demonstrate the result $L = \oplus L$ implies there is a well-known *NP–complete* in P. In this way, we guarantee the complexity class P is equal to NP.

Keywords: complexity classes · completeness · polynomial time · XOR-3SAT · MONOTONE-1-IN-3-3SAT.

1 Introduction

The P versus NP problem is a major unsolved problem in computer science [1]. This is considered by many to be the most important open problem in the field [1]. It is one of the seven Millennium Prize Problems selected by the Clay Mathematics Institute [1]. 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 [1].

In 1936, Turing developed his theoretical computational model [4]. The deterministic and nondeterministic Turing machines have become in two of the most important definitions related to this theoretical model for computation [4]. A deterministic Turing machine has only one next action for each step defined in its program or transition function [4]. 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 [4].

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 [5]. A complexity class is a set of problems, which are represented as a language, grouped by measures such as the running time, memory, etc [5].

In the computational complexity theory, the class P contains those languages that can be decided in polynomial time by a deterministic Turing machine [8]. The class NP consists in those languages that can be decided in polynomial time by a nondeterministic Turing machine [8]. The biggest open question in theoretical computer science concerns the relationship between these classes: Is P equal to NP? 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 [7].

It is fully expected that $P \neq NP$ [9]. For that reason, $P = NP$ is considered as a very unlikely event [9]. 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 for computer science, but for many other fields as well [1]. Whether $P = NP$ is still a controversial possible solution to this problem [1]. However, we prove the complexity class P is equal to NP. Hence, we solve one of the most important open problems in computer science with a solution which was certainly unexpected and with stunning practical consequences [1].

2 Definitions

Let Σ be a finite alphabet with at least two elements, and let Σ^* be the set of finite strings over Σ [4]. A Turing machine M has an associated input alphabet Σ [4]. For each string w in Σ^* there is a computation associated with M on input w [4]. We say that M accepts w if this computation terminates in the accepting state, that is $M(w) =$ "*yes*" [4]. 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 [4].

The language accepted by a Turing machine M, denoted $L(M)$, has an associated alphabet Σ and is defined by

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

We denote by $t_M(w)$ the number of steps in the computation of M on input w [4]. 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 [4]. We say that M runs in polynomial time if there is a constant k such that for all n, $T_M(n) \leq n^k + k$ [4].

In other words, this means the language $L(M)$ can be accepted by the Turing machine M in polynomial time. Therefore, P is the complexity class of languages that can be accepted in polynomial time by deterministic Turing machines [5]. 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 [4]. 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 [9].

NL is the class of languages that are decidable by a nondeterministic logarithmic Turing machine [4]. We can give a certificate-based definition for NL [4]. The certificate-based definition of NL assumes that a logarithmic Turing machine has another separated read-only tape [4]. On each step of the machine the machine's head on that tape can either stay in place or move to the right [4]. In particular, it cannot reread any bit to the left of where the head currently is [4]. For that reason this kind of special tape is called “read once” [4].

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

$$x \in L_1 \Leftrightarrow \exists u \in \{0, 1\}^{p([x])}\ such\ that\ M\ accepts\ \langle x, u\rangle$$

where by $M(x, u)$ we denote the computation of M where x is placed on its input tape and u is placed on its special read-once tape, and M uses at most $O(\log[x])$ space on its read/write work tapes for every input x where $[\ldots]$ is the bit-length function. We will call this Turing machine a logarithmic space verifier.

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

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

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

– $L_1 \in NP$, and
– $L_2 \leq_p L_1$ for every $L_2 \in NP$.

If L_1 is a language such that $L_2 \leq_p L_1$ for some $L_2 \in$ *NP–complete*, then L_1 is *NP–hard* [8]. Moreover, if $L_1 \in NP$, then $L_1 \in$ *NP–complete* [8].

A logarithmic space transducer is a Turing machine with a read-only input tape, a write-only output tape, and some read/write work tapes [4]. The work tapes must contain at most $O(\log n)$ symbols [4]. 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 [4]. We call f a logarithmic space computable function [4]. 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 for the complexity class P and the classes below.

A Boolean formula ϕ 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 formula with a satisfying truth assignment is a satisfiable formula. 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 [5]. 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 [5]. A Boolean formula is in 3-conjunctive normal form or $3CNF$, if each clause has exactly three distinct literals [5]. 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 $\neg x_2$.

3 Results

Definition 1. ***XOR–3SAT***

INSTANCE: A Boolean formula ψ that is the conjunctions of a set C of clauses $c_1, \ldots, c_m$, where each c_i consists of the EXCLUSIVE OR (denoted $\oplus$) of three literals.

QUESTION: Is it the case that ψ is satisfiable?

REMARKS: XOR–3SAT is complete for $\oplus L$ [2].

Definition 2. ***MONOTONE–XOR–4SAT***

INSTANCE: A Boolean formula F that is the conjunctions of a set C of clauses $c_1, \ldots, c_m$, where each c_i consists of the EXCLUSIVE OR (denoted $\oplus$) of four positive literals.

QUESTION: Is it the case that F is satisfiable?
REMARKS: MONOTONE–XOR–4SAT $\in \oplus L$ *because XOR–SAT* $\in \oplus L$ *[2].*

Theorem 1. *MONOTONE–XOR–4SAT is complete for* $\oplus L$.

Proof. We assume that each variable in the Boolean formula ψ of n variables in *XOR–3SAT* is a unique positive integer between 1 and n. We denote as X the set of variables in ψ from *XOR–3SAT*. The negative literals are represented as the negative value of each variable value, such that the negative literal of the variable a is $-a$. Since we need to create the positive literals in the language *MONOTONE–XOR–4SAT*, then we use the function h such that

$$h(x) = \textbf{if } (x > 0) \textbf{ return } (2 \times x) \textbf{ else return } (-2 \times x + 1).$$

The polynomial time reduction is described in the pseudo code Algorithm 1.

Algorithm 1 Logarithmic space reduction from *XOR–3SAT* to *MONOTONE–XOR–4SAT*

```
/*A set of clauses $C$ and the set of variables $X$ of an instance in XOR–3SAT*/
procedure REDUCTION($C, X$)
    output $(x \oplus y \oplus y \oplus y)$
    output $(z \oplus y \oplus y \oplus y)$
    /*Iterate for each variable in $X$*/
    for all $a \in X$ do
        output $(z \oplus x \oplus h(a) \oplus h(-a))$
    end for
    /*Iterate from the clauses of $C$*/
    for all $(a \oplus b \oplus c) \in C$ do
        output $(z \oplus h(a) \oplus h(b) \oplus h(c))$
    end for
end procedure
```

Note, in this reduction we guarantee that every literal is positive through the function h. In addition, we add new three positive literals x, y and z represented by the positive integers $(2 \times n + 2)$, $(2 \times n + 3)$ and $(2 \times n + 4)$ respectively, where n is the cardinality in the set of variables X. If every clause $(a \oplus b \oplus c) \in C$ can be converted to $(h(a) \oplus h(b) \oplus h(c))$, then we obtain a new Boolean formula where every literal is positive. However, this does not guarantee that for every variable a, the literals $h(a)$ and $h(-a)$ have oppositive values. This is guaranteed with the new variables x and y, such that we consider the clauses $(x \oplus y \oplus y \oplus y)$, $(y \oplus y \oplus y)$ and $(x \oplus h(a) \oplus h(-a))$. The reason is because the value $x = 0$ is obligated with the clauses $(x \oplus y \oplus y \oplus y)$ and $(y \oplus y \oplus y)$ and the evaluated clauses $(0 \oplus h(a) \oplus h(-a))$ guarantee the oppositive values of the literals $h(a)$ and $h(-a)$. Finally, we add the new variable z for every modified clause which has three positive literals. In this way, if we have a satisfying truth assignment of the new Boolean formula in *MONOTONE–XOR–4SAT* where $z = 0$, then

ψ will be satisfiable under the same truth assignment just excluding the literals x, y and z and replacing the value of the positive literal $h(a)$ by the value of a that can be a positive or negative literal. In addition, if the new Boolean formula in *MONOTONE–XOR–4SAT* has a satisfying truth assignment where $z = 1$, then we can create a new satisfying truth assignment where $z = 0$ just replacing each evaluation of $h(a) = 1$ by $h(a) = 0$ and $h(b) = 0$ by $h(b) = 1$ and including the same procedure for the variables x and y where a and b are literals of ψ. In general, the whole algorithm uses logarithmic space in the work tapes since the new Boolean formula is created in the output tape into a write-only way. Consequently, we obtain *XOR–3SAT* $\leq_l$ *MONOTONE–XOR–4SAT*. Since the language *XOR–3SAT* can be reduced to *MONOTONE–XOR–4SAT* in logarithmic space, then *MONOTONE–XOR–4SAT* is hard for $\oplus L$. Furthermore, *MONOTONE–XOR–4SAT* is in $\oplus L$. To sum up, we obtain the language *MONOTONE–XOR–4SAT* is complete for $\oplus L$.

Definition 3. ***HITTING–SET–2***

INSTANCE: A "universe" set U and a family of n sets $S_i \subseteq U$ with the property that $|S_i| \leq 2$ for $1 \leq i \leq n$ where $|\ldots|$ is the cardinality function.

QUESTION: Is it the case that there is a subset H of U such that $|S_i \cap H| = 1$?

REMARKS: HITTING–SET–2 $\in$ L [3], [10].

Theorem 2. *MONOTONE–XOR–4SAT $\in$ L.*

Proof. Given a collection of clauses C from an instance F of the language *MONOTONE–XOR–4SAT*, then we create for each clause $(a \oplus b \oplus c \oplus d) \in C$ the sets $\{(a, b), (c, d)\}$, $\{(a, c), (b, d)\}$ and $\{(a, d), (b, c)\}$ where every tuple (x, y) is a pair of unordered literals which these literals might be equals. The "universe" set U consists in all the tuples that we could create for each clause $c_i \in C$. The family of sets $S_i \subseteq U$ are those that we create for each clause $c_i \in C$ containing two pairs of unordered literals. Is it the case that there is a subset H of U such that $|S_i \cap H| = 1$ and F does not belong to *MONOTONE–XOR–4SAT*? The answer is no. Think in the properties of a subset H of U such that $|S_i \cap H| = 1$. The key observation is the sets $\{(a, b), (c, d)\}$, $\{(a, c), (b, d)\}$ and $\{(a, d), (b, c)\}$ of each clause $(a \oplus b \oplus c \oplus d) \in C$ satisfies that exactly one of the clauses

1. $(a \oplus b)$ or $(c \oplus d)$ in the set $\{(a, b), (c, d)\}$ should be true and the other false and,
2. $(a \oplus c)$ or $(b \oplus d)$ in the set $\{(a, c), (b, d)\}$ should be true and the other false and,
3. $(a \oplus d)$ or $(b \oplus c)$ in the set $\{(a, d), (b, c)\}$ should be true and the other false

if and only if the clause $(a \oplus b \oplus c \oplus d)$ is satisfiable for some truth assignment. Hence, if there is a subset H of U such that $|S_i \cap H| = 1$, then F is satisfiable, and this property is necessary to prove that the Boolean formula F can be satisfiable. Indeed, when there no is a subset H of U such that $|S_i \cap H| = 1$, then this would mean we cannot separate each pair of every set $\{tuple_1, tuple_2\}$ by exactly one evaluated as true and the other as false which is equivalent to assume that F is unsatisfiable. Is this a logarithmic space reduction? Yes, since we can iterate for each clause $c_1 \in C$ and write to the output tape the respective three sets $S_i \subseteq U$ that require each clause. In this reduction, there could be repeated sets $S_i \subseteq U$ in the output, but these duplications can be ignored from an instance of *HITTING–SET–2*. Therefore, this algorithm is a logarithmic space reduction from *MONOTONE–XOR–4SAT* to *HITTING–SET–2* such that

$$F \in MONOTONE\text{–}XOR\text{–}4SAT \textit{ if and only if } \{S_1, S_2, \ldots\} \in HITTING\text{–}SET\text{–}2$$

and thus, *MONOTONE–XOR–4SAT* $\leq_l$ *HITTING–SET–2*. In conclusion, we obtain *MONOTONE–XOR–4SAT* $\in L$, because every problem that could be logarithmic space reduced to a problem in L is in L as well [9].

Theorem 3. *$L = \oplus L$.*

Proof. The single existence of a complete problem in $\oplus L$ in L is sufficient to show $L = \oplus L$. Hence, this is a consequence of Theorems 1 and 2.

Definition 4. ***MONOTONE–1–IN–3–3SAT***

INSTANCE: A Boolean formula ϕ in $3CNF$ such that there is no clause which contains a negated literal.

QUESTION: Is there a truth assignment for ϕ such that each clause in ϕ has exactly one true literal?

REMARKS: MONOTONE–1–IN–3–3SAT is in NP–complete [6].

Definition 5. ***MONOTONE–2SAT***

INSTANCE: A Boolean formula φ that is the conjunctions of a set C of clauses $c_1, \ldots, c_m$, where each c_i consists of the OR (denoted $\vee$) of two negated literals.

QUESTION: Is it the case that φ is satisfiable?

Theorem 4. *MONOTONE–1–IN–3–3SAT $\in P$.*

Proof. Consider a Boolean formula ϕ in $3CNF$ with m clauses such that there is no clause which contains a negated literal. Suppose there is a positive literal x which appears k times. We replace the first occurrence of x by x_1, the second by x_2, and so on, where $x_1, x_2, \ldots, x_k$ are k new variables. Later, we add

$$(\neg\, x_1 \vee x_2) \wedge (\neg\, x_2 \vee x_3) \wedge \ldots \wedge (\neg\, x_k \vee x_1)$$

to a new Boolean formula ϕ' in CNF which contains the modified clauses of ϕ plus these above clauses of fewer than 3 literals. We create the Boolean formula ϕ' as result of making this procedure for each positive literal x in ϕ. Note, this is logically equivalent to

$$x_1 \Rightarrow x_2 \Rightarrow \ldots \Rightarrow x_k \Rightarrow x_1$$

such that the resulting expression in ϕ' satisfies the condition for a truth assignment on x. In this new formula ϕ' every variable appears at most 3 times. If

$k = 1$, then the literal x_1 appears once. If $k > 1$, for every integer i between 1 and k, we have that the literal x_i appears twice and $\neg x_i$ appears once. Suppose we have the following instance ϕ of *MONOTONE–1–IN–3–3SAT*

$$\ldots (x \vee w \vee g) \wedge \ldots \wedge (x \vee y \vee z) \ldots$$

then the transformed expression over the variable x would be

$$\ldots (x_1 \vee w \vee g) \wedge \ldots \wedge (x_2 \vee y \vee z) \ldots (\neg x_1 \vee x_2) \wedge (\neg x_2 \vee x_1)$$

where

– the variable x_1 appears thrice and,
– the literal x_1 appears twice and,
– the literal $\neg x_1$ appears once.

Now, from the expression ϕ' in CNF, we enumerate the variables which appear in the clauses of three literals from 1 to $3 \times m$ such that for the first clause we assign the enumeration from 1 to 3 to the variables, for the second clause we assign the enumeration from 4 to 6 and so on until we reach the last clause of three literals where we assign the enumeration from $3 \times m - 2$ to $3 \times m$. This can be simplified in this way: For the i^{th} clause $c_i = (a_p \vee b_q \vee c_r)$ of three literals we just transform it in $c_i = (a_{3\times i-2} \vee b_{3\times i-1} \vee c_{3\times i})$. We replace the same variables in the clauses of two literals according to the new enumeration. Suppose we have the following expression ϕ' in CNF

$$\ldots (x_1 \vee w_1 \vee g_1) \wedge (x_2 \vee y_1 \vee z_1) \wedge \ldots (\neg x_1 \vee x_2) \wedge (\neg x_2 \vee x_1) \ldots$$

then the transformed expression after the new enumeration would be

$$\ldots (x_1 \vee w_2 \vee g_3) \wedge (x_4 \vee y_5 \vee z_6) \wedge \ldots (\neg x_1 \vee x_4) \wedge (\neg x_4 \vee x_1) \ldots$$

where we replace the new enumerated variable in a clause of three literals over the two clauses of two literals such that one contains the positive literal and the other the negative literal. After that, we replace in the modified expression ϕ' in CNF the operator OR (denoted $\vee$) by the EXCLUSIVE OR (denoted $\oplus$). In this way, we create a new Boolean formula ψ that is an instance of *XOR–SAT*.

Next, from the clauses of three literals in the new Boolean formula ψ, we create a Boolean formula φ that would be an instance of *MONOTONE–2SAT*. This new formula will be constructed from the clauses of three literals in ψ which have already the enumeration in ascending order such that if we go through each integer i from 1 to m, then we take the i^{th} clause $c_i = (a_{3\times i-2} \oplus b_{3\times i-1} \oplus c_{3\times i})$ of three literals and we add at the end of φ the formula

$$P_i = (\neg a_{3\times i-2} \vee \neg b_{3\times i-1}) \wedge (\neg b_{3\times i-1} \vee \neg c_{3\times i}) \wedge (\neg a_{3\times i-2} \vee \neg c_{3\times i}).$$

Since c_i is evaluated as true if and only if exactly 1 or 3 members of the set $\{a_{3\times i-2}, b_{3\times i-1}, c_{3\times i}\}$ are true and P_i is evaluated as true if and only if exactly 1 or 0 members of the set $\{a_{3\times i-2}, b_{3\times i-1}, c_{3\times i}\}$ are true, then we obtain the original clause $d_i = (a \vee b \vee c)$ in *MONOTONE–1–IN–3–3SAT* has exactly one true literal if and only if both formulas c_i and P_i have the same satisfying truth assignment. Finally, we construct the Boolean formula φ as the conjunction of P_i for every clause c_i in ψ of three literals, that is, $\varphi = P_1 \wedge \ldots \wedge P_m$. In addition, the clauses of two literals in ψ guarantee the appropriated truth assignment of every positive variable in ϕ: There is a truth assignment for ϕ such that each clause in ϕ has exactly one true literal if and only if ψ has the same property. Moreover, there is a truth assignment for ψ such that each clause in ψ has exactly one true literal if and only if this is also a satisfying truth assignment for both formulas φ and ψ at the same time. Note the clauses in φ appear from left to right in the same ascending enumerated order that we provide in the enumeration step.

In this way, we can create a logarithmic space verifier M that receives as input the Boolean formula φ and as certificate a truth assignment such that the variables are sorted by the enumerated order that we provide in the above step. This would mean in a truth assignment the appearance from left to right in the order of the evaluation of the variables is firstly the one which obtained the assigned enumeration 1, secondly the one which obtained the assigned enumeration 2 and so forth... Since we know the first three variables in the certificate string corresponds to the first three clauses from left to right in φ, and the next forward three variables corresponds to the contiguous next three clauses in φ and so on, then we can store only three current variables each time in order to know whether the truth assignment satisfies the Boolean formula. M is a logarithmic space verifier since we do not have to look backward over the certificate for the evaluation of the clauses and we can store only three variables each time, so the work tapes have at most logarithmic space.

However, *XOR–SAT* is in NL because we prove that $L = \oplus L$. Consequently, there is a nondeterministic logarithmic Turing machine N which outputs every possible satisfying truth assignment of ψ in a nondeterministic way. We can assume the nondeterministic logarithmic Turing machine N outputs the satisfying truth assignment of ψ in the same enumerated variable order we provide in the enumeration step. In this way, if the logarithmic space verifier M computes $M(\varphi, N(\psi)) =$ "*yes*", then φ and ψ have the same satisfying truth assignment which also means ϕ is in *MONOTONE–1–IN–3–3SAT*. M would not be anymore a deterministic Turing machine, but a nondeterministic because in the certificate string there will be the output of $N(\psi)$. The special tape will still be a read-once input tape polynomial in M, since we simulate $N(\psi)$ at once: We compute $N(\psi)$ as much as we need to read a new symbol in the read-once input tape in M. In addition, the possible output of $N(\psi)$ will still be polynomial in relation to φ. Since we have the result $M(\varphi, N(\psi)) =$ "*yes*" for the input (φ, ψ) if and only if ϕ is indeed in *MONOTONE–1–IN–3–3SAT*, then we affirm that *MONOTONE–1–IN–3–3SAT* can be solved in polynomial time, because $NL \subseteq P$ [9]. The whole procedure can run in polynomial time:

– We can create the Boolean formula ϕ' in time $O(m^2)$ just replacing the modified clauses of three literals and adding the clauses of two literals. Certainly, we can go each time through every clause ϕ just adding the new clauses of

two literals and modifying the other clauses of three literals which are at most $m - 1$.
- After that, the enumeration in ϕ' can be done in time $O(m^2)$, since we need to replace each variable in a clause of three literals by a new variable and substitute it in the clause of two literals when the old variable still exists as a positive or negative literal.
- Next, we create the Boolean formula ψ and φ in time $O(m)$ since the replacement of the operator OR (denoted $\vee$) by the EXCLUSIVE OR (denoted $\oplus$) can be done in linear time. Besides, the addition of the clauses in φ can be done in linear time iterating each clause of three literals in ψ and positioning it in the right place according to the enumeration from left to right in φ. We can use an array to locate the position in this last step and thus, the final Boolean formula will be sorted after the creation of each formula P_i.

Certainly, we can create the Boolean formulas ψ and φ just running the whole computation in time $O(m^2)$. Furthermore, the creation of M and N can previously be done at once for all the instances of *MONOTONE–1–IN–3–3SAT* and thus, this is a polynomial time algorithm and the proof is completed.

Theorem 5. *$P = NP$.*

Proof. From the Theorem 4, we obtain *MONOTONE–1–IN–3–3SAT* $\in P$. If any *NP–complete* problem can be solved in polynomial time, then every language in NP has a polynomial time algorithm [5]. In conclusion, we finally prove that $P = NP$.

References

1. Aaronson, S.: P $\stackrel{?}{=}$ NP. Electronic Colloquium on Computational Complexity, Report No. 4 (2017)
2. Allender, E., Bauland, M., Immerman, N., Schnoor, H., Vollmer, H.: The Complexity of Satisfiability Problems: Refining Schaefer's theorem. Journal of Computer and System Sciences **75**(4), 245–254 (2009). https://doi.org/https://doi.org/10.1016/j.jcss.2008.11.001
3. Álvarez, C., Greenlaw, R.: A Compendium of Problems Complete for Symmetric Logarithmic Space. Computational Complexity **9**(2), 123–145 (2000). https://doi.org/10.1007/PL00001603
4. Arora, S., Barak, B.: Computational Complexity: A Modern Approach. Cambridge University Press (2009)
5. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms. The MIT Press, 3rd edn. (2009)
6. Garey, M.R., Johnson, D.S.: Computers and Intractability: A Guide to the Theory of NP-Completeness. San Francisco: W. H. Freeman and Company, 1 edn. (1979)
7. Gasarch, W.I.: Guest Column: The Second P $\stackrel{?}{=}$ NP Poll. SIGACT News **43**(2), 53–77 (Jun 2012). https://doi.org/10.1145/2261417.2261434
8. Goldreich, O.: P, NP, and NP-Completeness: The basics of computational complexity. Cambridge University Press (2010)
9. Papadimitriou, C.H.: Computational Complexity. Addison-Wesley (1994)
10. Reingold, O.: Undirected Connectivity in Log-space. J. ACM **55**(4), 1–24 (Sep 2008). https://doi.org/10.1145/1391289.1391291