P versus NP

Frank Vega

Joysonic, Belgrade, 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. Given a positive integer x and a collection S of positive integers, MAXIMUM is the problem of deciding whether x is the maximum of S. We prove this problem is complete for P. Another major complexity classes are LOGSPACE, LOGTIME and coNP. Whether LOGSPACE = P is a fundamental question that it is as important as it is unresolved. We show the problem MAXIMUM can be decided in logarithmic space. Consequently, we demonstrate the complexity class LOGSPACE is equal to P. Moreover, we define a problem called SUCCINCT-MAXIMUM. SUCCINCT-MAXIMUM contains the instances of MAXIMUM that can be represented by an exponentially more succinct way. We show this succinct version of MAXIMUM is in coNP-complete under logarithmic reductions. Hence, under the assumption of P = NP, we obtain the padded version of SUCCINCT-MAXIMUM is in LOGTIME and P-hard. However, this is not possible according to LOGTIME is strictly contained in LOGSPACE, because that result would imply LOGTIME = LOGSPACE. In this way, we demonstrate the assumption of several computer scientists whom fully expect that P is not equal to NP.

Keywords: Complexity Classes · Polynomial Time · Logarithmic Space · Complete Problem · Succinct Representation.

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 to carry a US$1,000,000 prize for the first correct solution [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 [9]. The deterministic and nondeterministic Turing machines have become in two of the most important definitions related to this theoretical model for computation [9]. A deterministic Turing machine has only one next action for each step defined in its program or transition function [9]. 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 [9].

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 [3]. A complexity class is a set of problems, which are represented as a language, grouped by measures such as the running time, memory, etc [3].

In the computational complexity theory, the class P contains those languages that can be decided in polynomial time by a deterministic Turing machine [4]. The class NP consists in those languages that can be decided in polynomial time by a nondeterministic Turing machine [4]. The biggest open question in theoretical computer science concerns the relationship between these classes: Is P equal to NP?

A logarithmic Turing machine has a read-only input tape, a write-only output tape, and a read/write work tape [9]. The work tape may contain $O(\log n)$ symbols [9]. $LOGSPACE$ is the complexity class containing those decision problems that can be decided by a deterministic logarithmic Turing machine [9]. On the other hand, $LOGTIME$ is another language based on the languages that can be solved by a logarithmic time algorithm [8]. In this work, we prove the complexity class $LOGSPACE$ is equal to P. Moreover, we demonstrate the complexity class P is not equal to NP.

2 Basic Definitions

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 [2].

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

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

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 running 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]. The notations we use to describe the asymptotic running time of an algorithm are defined in terms of functions whose domains are the set of natural numbers [3]. Such notations are convenient for describing the worst and better case running time functions, which is usually defined only on integer input sizes [3]. For a given function $g(n)$, we denote by $O(g(n))$ the set of functions

$$O(g(n)) = \{f(n) : \textit{ There exist positive constants } c \textit{ and } n_0$$

$$\textit{such that } 0 \leq f(n) \leq c \times g(n) \textit{ for all } n \geq n_0\}.$$

O-notation provides an asymptotic upper bound [3]. We say that M runs in polynomial time if there is a constant k such that for all n, $T_M(n) \leq n^k + k$ [2]. In other words, this means the language $L(M)$ can be accepted by the Turing machine M in polynomial time or more specific in a running time $O(n^k)$ for some constant k [2]. Therefore, P is the complexity class of languages that can be accepted in polynomial time by deterministic Turing machines [3]. A verifier for a language L is a deterministic Turing machine M, where

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

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 [2]. A verifier uses additional information, represented by the symbol c, to verify that a string w is a member of L. This information is called certificate. NP is also the complexity class of languages defined by polynomial time verifiers [8]. If NP is the class of problems that have succinct certificates, then the complexity class $coNP$ must contain those problems that have succinct disqualifications [8]. That is, a “*no*” instance of a problem in $coNP$ possesses a short proof of its being a “*no*” instance [8].

A logarithmic space transducer is a Turing machine with a read-only input tape, a write-only output tape, and a read/write work tape [2]. The work tapes must contain at most $O(\log n)$ symbols [2]. 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 [2]. We call f a logarithmic space computable function [2]. 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 $x \in \{0,1\}^*$,

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

An important complexity class is *P–complete* [5]. A language $L \subseteq \{0,1\}^*$ is *P–complete* if

- $L \in P$, and
- $L' \leq_l L$ for every $L' \in P$.

If L is a language such that $L' \leq_l L$ for some $L' \in$ *P–complete*, then L is *P–hard* [8]. Moreover, if $L \in P$, then $L \in$ *P–complete* [8]. 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. 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 [3]. 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 [3]. A Boolean formula is in 3-conjunctive normal form or $3CNF$, if each clause has exactly three distinct literals [3].

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 $\neg x_2$.

For every $n, m \in \mathbb{N}$ a Boolean circuit C with n inputs and m outputs is a directed acyclic graph [2]. It contains n nodes with no incoming edges; called the input gates and m nodes with no outgoing edges, called the output gates [2]. All other nodes are labeled with one of $\vee$, $\wedge$ or $\neg$ (in other words, the logical operations OR, AND, and NOT) [2]. The $\vee$ and $\wedge$ nodes have fanin (i.e., number of incoming edges) of 2 and the $\neg$ nodes have fanin 1. The size of C is the number of nodes in it [2].

3 Results

Definition 1. ***MAXIMUM***

INSTANCE: A positive integer x and a collection S of positive integers. The collection S could not be a set, since by the definition of a collection, this can contain repeated elements.

QUESTION: Is x the maximum number in S?

Lemma 1. *$MAXIMUM \in P$.*

Proof. How many comparisons are necessary to determine whether a positive integer x is the maximum of a collection of n positive integers? We can easily obtain an upper bound of n comparisons: examine each element of the collection in turn and keep track of the largest element seen so far and finally, we compare the ultimate result with x. In the following procedure, we assume that the collection resides in an array A of length n.

Is this the best amount of comparisons we can do? Yes, since we can obtain a lower bound of $n-1$ comparisons for the problem of determining the maximum and one another comparison to check whether this is equal to x [3]. Think of any

Algorithm 1 MAXIMUM's Polynomial Time Algorithm

```
1: procedure $MAXIMUM(x, A)$
2:     /*Assign the first element*/
3:     $max \leftarrow A[0]$
4:     /*Iterate for the elements of the collection*/
5:     for $i \leftarrow 1$ to $n - 1$ do
6:         /*When the element $A[i]$ is greater than $max$*/
7:         if $max < A[i]$ then
8:             /*Update the new value of $max$*/
9:             $max \leftarrow A[i]$
10:        end if
11:    end for
12:    /*If the number $x$ is equal to the maximum of the collection*/
13:    if $max = x$ then
14:        /*Accept*/
15:        return "yes"
16:    else
17:        /*Otherwise reject*/
18:        return "no"
19:    end if
20: end procedure
```

algorithm that determines the maximum as a tournament among the elements [3]. Each comparison is a match in the tournament in which the larger of the two elements wins [3]. The key observation is that every element except the winner must lose at least one match [3]. Finally, we compare the winner with x [3]. Hence, n comparisons are necessary to determine whether x is the maximum of the collection of positive integers, and the algorithm $MAXIMUM$ is optimal with respect to the number of comparisons performed [3].

Definition 2. ***Unweighted, Not–All–Equal Clauses, 3SAT/FLIP***

INSTANCE: A Boolean formula ϕ in $3CNF$ and a truth assignment T. Each clause has a weight of 1. The clauses are not–all–equals clauses with positive literals. A truth assignment satisfies a clause c under the not–all–equals criterion if it is such that c has at least one true and one false literal.

QUESTION: Is the truth assignment T the maximum cost assignment of ϕ over all neighbors of T? The cost of the assignment is the sum of the weights of the clauses it satisfies. The neighbors of T are truth assignments that differ from T in one bit position.

REMARKS: We denote this language as $U3NSATFLIP$ [5].

Theorem 1. *$U3NSATFLIP \leq_l MAXIMUM$.*

Proof. Given a Boolean formula ϕ in $3CNF$ and a truth assignment T, we can calculate the cost assignment of T based on the *not–all–equals* criterion in a logarithmic space algorithm. In the following function $COST$, we assume the

Algorithm 2 COST's Logarithmic space algorithm

```
 1: function $COST(\phi, T)$
 2:     /*Initialize the cost assignment to 0*/
 3:     $num \leftarrow 0$
 4:     /*For each clause in $\phi$*/
 5:     for all $c \in \phi$ do
 6:         /*The clause $c$ is equal to $(p \vee q \vee r)$*/
 7:         if $0 < T[p] + T[q] + T[r] < 3$ then
 8:             /*Increment $num$ because $c$ complies with the *not–all–equals* criterion*/
 9:             $num \leftarrow num + 1$
10:         end if
11:     end for
12:     /*Return the cost assignment*/
13:     return $num$
14: end function
```

truth assignment T is a dictionary that maps every variable in ϕ to 1 or 0 (true or false).

This function uses logarithmic space in its work tapes and assumes the clauses contain only positive literals. Certainly, the calculation of $T[p] + T[q] + T[r]$ can be made storing a constant amount of space where p, q and r are the positive literals of each clause c in ϕ. In addition, if m is the number of clauses in ϕ, then the number num will not exceed the number m and thus, the work tapes will contain at most $O(\log m)$ space.

On the other hand, we can reduce an instance of $U3NSATFLIP$ into another of $MAXIMUM$ in logarithmic space. For this purpose, we are going to use the function $COST$ into a new algorithm. In the following function $REDUCE$, we represent the input instance as a given Boolean formula ϕ in $3CNF$ of $n - 1$ variables with a truth assignment T and the output instance as a positive integer x with an array A filled with n elements of a collection of positive integers. We will assume the truth assignment T given in the input is a dictionary that maps every variable in ϕ to 1 or 0 (true or false) as well.

Is this a logarithmic space reduction from $U3NSATFLIP$ to $MAXIMUM$? Given a Boolean formula ϕ in $3CNF$ and a truth assignment T, we will obtain the positive integer x as the cost assignment of ϕ in T and in the array A the cost assignment of ϕ from all the neighbors of T included the cost assignment of T. In this way, if x is the maximum in the collection of positive integers represented by A, then $\langle \phi, T \rangle$ belongs to $U3NSATFLIP$ where $\langle \ldots \rangle$ is the binary encoding. However, if x is the maximum in the collection of positive integers represented by A (remember that A contains x), then this will be an element of the language $MAXIMUM$ as well. Certainly, $\langle \phi, T \rangle$ is in $U3NSATFLIP$ if and only if x is the maximum in the collection of positive integers in A. The function $REDUCE$ uses logarithmic space since the bit-length of the index i is $O(\log n)$ because there are $n - 1$ variables and thus, there are at most n costs assignments that we need to calculate which is the cost of the original truth assignment T and the $n - 1$

Algorithm 3 REDUCE's Logarithmic space algorithm

```
1: function $REDUCE(\phi, T)$
2:     /*Create an empty array A*/
3:     $A \leftarrow [\ldots]$
4:     /*Initialize the index of $A$ in 0*/
5:     $i \leftarrow 0$
6:     /*For each variable $y$ in $\phi$*/
7:     for all $y \in \phi$ do
8:         /*Flip the value of $T[y]$ (0 to 1 or 1 to 0)*/
9:         $T[y] \leftarrow (T[y] - 1) \times (-1)$
10:        /*Calculate the cost of the flipped $T$ based on the *not–all–equals* criterion*/
11:        $num \leftarrow COST(\phi, T)$
12:        /*Assign the cost assignment of the neighbor of $T$ after flipping over $T$ the
           bit position in the variable $y$*/
13:        $A[i] \leftarrow num$
14:        /*Increment the index to store the new neighbor cost assignment of $T$*/
15:        $i \leftarrow i + 1$
16:        /*Return the value of $T[y]$ to the original bit number*/
17:        $T[y] \leftarrow (T[y] - 1) \times (-1)$
18:    end for
19:    /*Calculate the cost of $T$ based on the *not–all–equals* criterion*/
20:    $x \leftarrow COST(\phi, T)$
21:    /*Assign the cost assignment of the original $T$ without flipping any bit posi-
       tion*/
22:    $A[i] \leftarrow x$
23:    /*Return the reduction*/
24:    return $(x, A)$
25: end function
```

truth assignment after flipping one bit position in T. Moreover, the bit position that we flip in T will use at most two symbols encoded in binary over the work tapes: the new bit value and the variable. In addition, the algorithm $COST$ runs in logarithmic space in relation to ϕ and the truth assignment T with at most one bit flipped. The algorithm $COST$ will take into account the original truth assignment T which remains in the input tape and the changed bit position which is stored in the work tapes. After the computation of $COST$ over each iteration, we will erase from the work tapes the at most $O(\log m)$ space that could contain those tapes where m is the number of clauses in ϕ. Furthermore, we do not need to store the value of the elements of A in the work tapes since they can be written directly to the output tape. The array A can be written to the output tape as the pairs (i, v_i) where i is an index between 0 and $n-1$ and v_i is equal to the positive integer $A[i]$. We also write the binary string of the number x to the output tape where this string contains at most $O(\log m)$ space. Consequently, we demonstrate $U3NSATFLIP \leq_l MAXIMUM$.

Theorem 2. *$MAXIMUM \in$ P–complete.*

Proof. We prove $U3NSATFLIP$ can be logarithmic reduced to $MAXIMUM$ and $U3NSATFLIP \in$ *P–complete* [7], thus $MAXIMUM$ belongs to *P–hard*. Moreover, since $MAXIMUM \in P$, then $MAXIMUM$ is in *P–complete*.

Theorem 3. *$MAXIMUM \in LOGSPACE$.*

Proof. Given a positive integer x and a collection S of positive integers, we are going to demonstrate we can decide this problem in logarithmic space. In the following procedure, we assume that the collection resides in array A of length n. Besides, we assume the function *length* calculates the bit-length of a binary string and uses a logarithmic space for the calculation.

Is this a logarithmic space algorithm? Yes, since we compare the value of the functions $length(x)$ and $length(A[i])$ (the i^{th} element of A) using a logarithmic space although we could partially calculate the $length(A[i])$. In addition, the calculated bit-length of x only uses at most $O(\log x)$ space. Besides, in the comparison with the bit-length of $A[i]$ and x we halt and reject immediately when $length(A[i])$ exceeds $length(x)$ at least in one digit and thus, we do not need to calculate completely the $length(A[i])$ to reject. In this way, we just keep at most $O(\log x)$ space in the calculation of $length(A[i])$. Finally, when both bit-lengths are equal, then we compare the elements $A[i]$ and x bit by bit. For this purpose, we compare only two bits in the input tape over the same position j from x and $A[i]$ in a descending order for each step. Notice, that we start to compare from the last bit position in a descending order. For example, in the binary string 100 which represents the number 4, we start iterating from the last bit element, that is the bit 1. Moreover, we store the position j in the work tapes and this value has at most $O(\log x)$ space. If it would be the case that $A[i]$ and x have the same bit-length, but $A[i]$ is greater than x, then we reject. We continue the iteration with the next value i while the property that x is the maximum number in the array remains as true. However, we only accept when the value of the variable

Algorithm 4 MAXIMUM's Logarithmic space algorithm

```
procedure $MAXIMUM(x, A)$
    /*Initialize the variable $answer$*/
    $answer \leftarrow$ "$no$"
    /*Iterate for each element of the collection*/
    for $i \leftarrow 0$ to $n - 1$ do
        /*If the bit-length of $x$ is lesser than the bit-length of element $A[i]$*/
        if $length(x) < length(A[i])$ then
            /*Reject because $A[i]$ is greater than $x$*/
            return "$no$"
            /*If the bit-length of $x$ is greater than the bit-length of element $A[i]$*/
        else if $length(x) > length(A[i])$ then
            /*Continue to the next iteration on $i$*/
            continue
            /*If the bit-length of $x$ is equal to the bit-length of element $A[i]$*/
        else
            /*Assign the index to the last bit element*/
            $j \leftarrow length(x) - 1$
            /*While there are bits to compare*/
            while $j \geq 0$ do
                /*Compare the bit in the position $j$ of $x$ with the bit in the position $j$ of $A[i]$*/
                if $x[j] < A[i][j]$ then
                    /*Reject because $A[i]$ is greater than $x$*/
                    return "$no$"
                    /*Compare the bit in the position $j$ of $x$ with the bit in the position $j$ of $A[i]$*/
                else if $x[j] > A[i][j]$ then
                    /*Continue to the next iteration on $i$*/
                    break
                else
                    /*Decrement the bit position $j$ of $x$ and $A[i]$*/
                    $j \leftarrow j - 1$
                end if
            end while
            /*After iterating from all the bits of $x$ and $A[i]$*/
            if $j < 0$ then
                /*$x$ is equal to $A[i]$*/
                $answer \leftarrow$ "$yes$"
            end if
        end if
    end for
    /*Accept if $answer$ = "$yes$" and reject when $answer$ = "$no$"*/
    return $answer$
end procedure
```

answer is "*yes*" when initially has the value of "*no*" by default. The value will be "*yes*" in the variable *answer* after the whole iteration for each element in the array if and only if there is at least one element $A[i]$ that is equal to x. Furthermore, if the iteration is completed until the last item, then x is greater than or equal to every element in the array A. To sum up, we show we can decide whether x is the maximum of the collection represented by the array A in logarithmic space and thus, $MAXIMUM \in LOGSPACE$.

Theorem 4. $LOGSPACE = P$.

Proof. As result of Theorems 2 and 3 we obtain $LOGSPACE = P$, because the complexity class $LOGSPACE$ is closed under logarithmic space reductions [8].

Definition 3. ***SUCCINCT–MAXIMUM***

INSTANCE: A positive integer x and a collection S of positive integers such that the collection S is represented by a Boolean circuit C where some positive integer i belongs to S if and only if $C(i)$ accepts. The size of the Boolean circuit C is bounded by m^k where k is a feasible constant and m is the number of input gates in C.

QUESTION: Is x the maximum number in S?

Theorem 5. *SUCCINCT–MAXIMUM is a succinct representation of the language $MAXIMUM$.*

Proof. Considering that m is the number of input gates in C, then the Boolean circuit C could be a succinct representation of a collection of positive integers S. Indeed, this will happen in many cases since the collection S could contain more than or approximately to 2^m elements (remember that a collection could contain repeated elements) if we represent it by a Boolean circuit of m input gates. However, the size of the Boolean circuit C is polynomially bounded by m for a feasible constant as exponent. Certainly, *SUCCINCT–MAXIMUM* is nothing else but a language that contains the instances of the problem $MAXIMUM$ which could be represented by an exponentially more succinct input in relation to S [8].

Theorem 6. *SUCCINCT–MAXIMUM* $\in coNP$.

Proof. The language of *SUCCINCT–MAXIMUM* is in $coNP$. Certainly, we can check in polynomial time a disqualification from an instance $\langle x, C\rangle$ of this language that is a positive integer y where $x < y$ and y is in S or we can simply verify in polynomial time when x is not in S where $\langle \ldots \rangle$ is the binary encoding. Indeed, we can check whether the both evaluations of y and x in C accept and check later whether $x < y$ or we can just verify when $C(x)$ does not accept. Certainly, we can polynomially make the verification when $\langle x, C\rangle$ is a "*no*" instance of the problem *SUCCINCT–MAXIMUM*, because the evaluation in the Boolean circuit can be done in polynomial time as well. Indeed, by definition its size does not exceed the value m^k where k is a feasible constant and m is the number of input gates in C.

Given a Boolean circuit C, the problem *coCIRCUIT–SAT* consists in deciding whether there is not any input such that C accepts [8].

Theorem 7. *coCIRCUIT–SAT* $\leq_l$ *SUCCINCT–MAXIMUM.*

Proof. Given a Boolean circuit C we can check whether $C(0)$ does not accept. In that case, we create a succinct Boolean circuit C' which only accepts the input string 0 and has the same number of input gates of C. We combine C with C' through the input gates into a new Boolean circuit C'' which accepts only when C or C' accept. This is possible just adding a gate OR between the output gates of C and C'. The instance of the positive integer 0 and the final Boolean circuit C'' belongs to *SUCCINCT–MAXIMUM* if and only if C is in *coCIRCUIT–SAT*. Certainly, 0 is the maximum of the collection that represents C'' if there is not any other input which C'' accepts. In addition, C'' accepts the positive integer 0 because of the construction of C' on C. Since we can create the succinct Boolean circuit C' and evaluate C on the input 0 in logarithmic space due to Theorem 4, then *coCIRCUIT–SAT* $\leq_l$ *SUCCINCT–MAXIMUM.*

Theorem 8. *SUCCINCT–MAXIMUM* $\in$ *coNP–complete.*

Proof. *coCIRCUIT–SAT* is a known *coNP–complete* problem [8]. Hence, the language *SUCCINCT–MAXIMUM* is in *coNP–hard* because of the Theorem 7 and due to the logarithmic reduction is also a polynomial reduction [8]. As result of Theorem 6, we obtain *SUCCINCT–MAXIMUM* is in $coNP$ and thus, the proof is completed.

Theorem 9. $P \neq NP$.

Proof. Given a positive integer x and a collection of positive integers S represented by an array, we can convert them into another instance the same x with a Boolean circuit C where some positive integer i belongs to S if and only if $C(i)$ accepts. Certainly, we can construct C just iterating from each element i of the array and creating a single circuit C_i which only accepts i. We assume that every circuit C_i has the same number of input gates. Later, we create C just unifying all the circuits C_i coinciding their input gates and joining all their output gates with an OR gate for obtaining a single output gate. In this way, we obtain a Boolean circuit where some positive integer i belongs to S if and only if $C(i)$ accepts. Given a Boolean circuit C, *MINIMUM–CIRCUIT* is the problem of deciding whether there is no circuit with fewer gates that computes the same Boolean function of C [8]. This problem belongs to PH [8]. However, if $P = NP$ then the polynomially hierarchy collapses to the first level [8]. In this way, *MINIMUM–CIRCUIT* would be in P when $P = NP$. Consequently, the complement language of *MINIMUM–CIRCUIT* would be in P because P is closed under complement. Moreover, the function problem of this complement would be in FP and that would mean, with a circuit C as input we can find another circuit C' which has the fewest gates such that C and C' compute the same Boolean function. Certainly, we can try with that function problem the search of another circuit with fewer gates and we can repeat this process over and over again until we reach the circuit C' with the fewest amount of gates. This can be done in polynomial time, because the class FP finds the solutions in polynomial time and the search of C' depends mostly of the amount of gates in the original C.

In this way, given an instance of $MAXIMUM$ represented by some positive integer x and a Boolean circuit C, we could reduce it in polynomial time to another instance of *SUCCINCT–MAXIMUM* that would be the same x with a succinctly representation of S that would be the circuit C' with the fewest gates which computes the same of C. Since the positive integer x and the circuit C can be reduced into an exponentially more succinct way such as x and C', then we can pad x and C' by enough "quasiblanks" symbols ∇ such that $\langle x, C' \rangle \nabla\nabla\nabla\nabla \ldots$ has the same string length of $\langle x, C \rangle$ where $\langle \ldots \rangle$ is the binary encoding. Since this polynomial reduction under the assumption of $P = NP$ is possible, then this can be converted into a logarithmic reduction according to Theorem 4. Consequently, the padded version of the language *SUCCINCT–MAXIMUM* would be in *P–hard*. Nevertheless, every instance $\langle x, C' \rangle \nabla\nabla\nabla\nabla \ldots$ of the padded version of the language *SUCCINCT–MAXIMUM* can be solved in logarithmic time, because $\langle x, C' \rangle$ can be solved in polynomial time due to P would be equal to $coNP$ when $P = NP$ and $\langle x, C' \rangle$ is exponentially more succinct than its padded version. As result, the padded version of *SUCCINCT–MAXIMUM* would be in $LOGTIME$ and in *P–hard* under the assumption of $P = NP$, but that would mean $LOGTIME = LOGSPACE$ which is a trivial contradiction due to $LOGTIME$ is strictly contained in $LOGSPACE$ [6]. In conclusion, we demonstrate $P \neq NP$ based on the reduction ad absurdum logic rule after finding a contradiction assuming that $P = NP$.

References

1. Aaronson, S.: P $\stackrel{?}{=}$ NP. Electronic Colloquium on Computational Complexity, Report No. 4 (2017)
2. Arora, S., Barak, B.: Computational complexity: a modern approach. Cambridge University Press (2009)
3. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms. The MIT Press, 3rd edn. (2009)
4. Goldreich, O.: P, NP, and NP-Completeness: The basics of computational complexity. Cambridge University Press (2010)
5. Greenlaw, R., Hoover, H.J., Ruzzo, W.L.: A Compendium of Problems Complete for P. Oxford University Press (1993)
6. Hopcroft, J.E., Paul, W.J., Valiant, L.G.: On time versus space. J. ACM **24**, 332–337 (1977)
7. Papadimitriou, C.H., Schäffer, A.A., Yannakakis, M.: On the complexity of local search. In: Proceedings of the Twenty-second Annual ACM Symposium on Theory of Computing. pp. 438–445. STOC '90, ACM, New York, NY, USA (1990). https://doi.org/10.1145/100216.100274
8. Papadimitriou, C.H.: Computational complexity. Addison-Wesley (1994)
9. Sipser, M.: Introduction to the Theory of Computation, vol. 2. Thomson Course Technology Boston (2006)