NL versus P

Frank Vega

NL versus P

Frank Vega

Abstract

We consider two new complexity classes that are called equivalent-L and equivalent-NL which have a close relation to the P versus NP problem. The class equivalent-L contains those languages that are ordered-pairs of instances of two specific problems in LSPACE, such that the elements of each ordered-pair have the same solution, which means, the same certificate. The class equivalent-NL has almost the same definition, but in each case we define the pair of languages explicitly in NL. In addition, we define the class double-P as the set of languages that contain each instance of another language in P, but in a double way, that is, in form of a pair with two identical instances. We demonstrate that double-P is a subset of equivalent-L. Moreover, we prove that equivalent-NL is a subset of NL. In this way, we show not only that every problem in double-P can be reformulated as the pairs of elements of two languages in LSPACE which have the same certificate, but we also show NL = P.

Keywords: P, NL, NP, logarithmic space, certificate
2000 MSC: 68-XX, 68Qxx, 68Q15

Introduction

Let Σ be a finite alphabet with at least two elements, and let Σ^* be the set of finite strings over Σ [1]. A Turing machine M has an associated input alphabet Σ [1]. For each string w in Σ^* there is a computation associated with M on input w [1]. We say that M accepts w if this computation terminates in the accepting state [1]. Note that M fails to accept w either if this computation ends in the rejecting state, or if the computation fails to terminate [1].

In 1936, Turing developed his theoretical computational model [1]. The deterministic and nondeterministic Turing machines have become in two of the most important definitions related to this theoretical model for computation. A deterministic Turing machine has only one next action for each step defined in its program or transition function [2]. 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 [2].

P versus NP is a major unsolved problem in computer science [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:

Email address: vega.frank@gmail.com (Frank Vega)

Is P equal to NP?

We can also state the complexity class NP using the following definition.

Definition 0.1. *A verifier for a language L is a deterministic Turing machine M, where*

$$L = \{w : M \text{ accepts } \langle w, c\rangle \text{ for some string } c\}.$$

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 [5]. 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.

Observe that, for polynomial time verifiers, the certificate is polynomially bounded by the length of w, because that is all the verifier can access in its time bound [5].

Definition 0.2. *NP is the class of languages that have polynomial time verifiers [5].*

A logspace machine is a Turing machine with a read-only input tape, a write-only output tape, and a read/write work tapes [5]. The work tapes may contain at most $O(\log n)$ symbols [5]. Another major complexity classes are $LSPACE$ and NL. The class $LSPACE$ has all the languages that are decidable on a deterministic logspace machine [6]. NL is the class of languages that are decidable on a nondeterministic logspace machine [6].

We can give a certificate-based definition for NL [6]. The certificate-based definition of NL assumes that a logspace machine has another separated read-only tape [6]. On each step of the machine the machine's head on that tape can either stay in place or move to the right [6]. In particular, it cannot reread any bit to the left of where the head currently is [6]. For that reason this kind of special tape is called "read once" [6].

Definition 0.3. *A language L is in NL if there exists a deterministic logspace machine and with an additional special read-once input tape polynomial $p : \mathbb{N} \to \mathbb{N}$ such that for every $x \in \{0, 1\}^*$,*

$$x \in L \Leftrightarrow \exists u \in \{0, 1\}^{p(|x|)} \text{ such that } M \text{ accepts } \langle x, u\rangle$$

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 tapes for every input x. We will call this Turing machine M a logspace verifier.

It is known that $LSPACE \subseteq NL \subseteq P$ [2]. Whether $LSPACE = NL$ is another fundamental question that it is as important as it is unresolved [2]. All efforts to solve the $LSPACE$ versus NL problem have failed [2]. Nevertheless, we prove $NL = P$.

1. Theoretical notions

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 [5]. 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 exists a polynomial time 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 *NP–complete* [4]. A language $L \subseteq \{0, 1\}^*$ is *NP–complete* if

1. $L \in NP$, and
2. $L' \leq_p L$ for every $L' \in NP$.

Furthermore, if L is a language such that $L' \leq_p L$ for some $L' \in$ *NP–complete*, then L is in *NP–hard* [7]. Moreover, if $L \in NP$, then $L \in$ *NP–complete* [7]. If any single *NP–complete* problem can be solved in polynomial time, then every NP problem has a polynomial time algorithm [7]. No polynomial time algorithm has yet been discovered for any *NP–complete* problem [3].

A principal *NP–complete* problem is SAT [8]. An instance of SAT is a Boolean formula ϕ which is composed of

1. Boolean variables: $x_1, x_2, \ldots, x_n$;
2. Boolean connectives: Any Boolean function with one or two inputs and one output, such as $\wedge$(AND), $\vee$(OR), $\neg$(NOT), $\Rightarrow$(implication), $\Leftrightarrow$(if and only if);
3. and parentheses.

A truth assignment for a Boolean formula ϕ is a set of values for the variables in ϕ. A satisfying truth assignment is a truth assignment that causes ϕ to be evaluated as true. A formula with a satisfying truth assignment is a satisfiable formula. The problem SAT asks whether a given Boolean formula is satisfiable [8].

Another *NP–complete* language is $3CNF$ satisfiability, or $3SAT$ [7]. We define $3CNF$ satisfiability using the following terms. A literal in a Boolean formula is an occurrence of a variable or its negation [7]. A Boolean formula is in conjunctive normal form, or CNF, if it is expressed as an AND of clauses, each of which is the OR of one or more literals [7]. A Boolean formula is in 3-conjunctive normal form or $3CNF$, if each clause has exactly three distinct literals [7].

For example, the Boolean formula

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

is in $3CNF$. The first of its three clauses is $(x_1 \vee \neg x_1 \vee \neg x_2)$, which contains the three literals x_1, $\neg x_1$, and $\neg x_2$. In $3SAT$, it is asked whether a given Boolean formula ϕ in $3CNF$ is satisfiable.

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 [5]. The work tapes must contain at most $O(\log n)$ symbols [5]. 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 [5]. We call f a logarithmic space computable function [5]. 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* [2]. A language $L \subseteq \{0,1\}^*$ is *P-complete* if

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

A principal *P–complete* problem is $HORNSAT$ [2]. We say that a clause is a Horn clause if it has at most one positive literal [2]. That is, all its literals, except possibly for one, are negations of variables. An instance of $HORNSAT$ is a Boolean formula ϕ in CNF which is composed only of Horn clauses [2].

For example, the Boolean formula

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

is a conjunction of Horn clauses. The *HORNSAT* asks whether an instance of this problem is satisfiable [2].

Another special case is the language where the instances are Boolean formulas in *CNF* and each clause contains only un-negated variables. This language, that we call *UN–NEGATED*, asks whether an instance of this problem is satisfiable. This is in *LSPACE*, since every Boolean formula in *CNF* is always satisfiable when every literal inside of the clauses is a un-negated variable. In a similar way, we define the language *NEGATED* where the Boolean formulas are in *CNF* and each clause contains only negated variables. This language is in *LSPACE* too.

2. Class equivalent-L

Definition 2.1. *We say that a language L belongs to $\equiv L$ if there exist two languages $L_1 \in LSPACE$ and $L_2 \in LSPACE$ and two deterministic logspace machines M_1 and M_2, where M_1 and M_2 are the logspace verifiers of L_1 and L_2 respectively, such that*

$$L = \{(x, y) : \exists z \text{ such that } M_1(x, z) = \text{“}yes\text{”} \text{ and } M_2(y, z) = \text{“}yes\text{”}\}.$$

We call the complexity class $\equiv L$ as “equivalent–L”. We represent this language L in $\equiv L$ as (L_1, L_2).

The logarithmic space reduction is frequently used for P and below [2]. There is a different kind of reduction for $\equiv P$: The *e–reduction*.

Definition 2.2. *Given two languages L_1 and L_2, where the instances of L_1 and L_2 are ordered-pairs of binary strings, we say that the language L_1 is e–reducible to the language L_2, written $L_1 \leq_{\equiv} L_2$, if there exist two logarithmic space computable functions $f : \{0,1\}^* \to \{0,1\}^*$ and $g : \{0,1\}^* \to \{0,1\}^*$ such that for all $x \in \{0,1\}^*$ and $y \in \{0,1\}^*$,*

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

Lemma 2.3. *The e–reduction is a logarithmic space reduction.*

Proof. We can construct a logarithmic space transducer M that computes an arbitrary *e–reduction* and receives as input an ordered-pair of string $\langle x, y\rangle$ and outputs $\langle f(x), g(y)\rangle$ where f and g are the two logarithmic space computable functions of this *e–reduction*. Suppose we use a delimiter symbol for the strings x and y. For example, let's take the blank symbol $\sqcup$ as delimiter [2]. Since f is a logarithmic space computable function, then we can simulate f on M using a logarithmic amount of space in its read/write work tape. In the meantime, M is printing the string $f(x)$ to the output without wander off after the symbol $\sqcup$ that separates x from y on the input tape. When the simulation of f halts, then M starts to simulate g from the other string y. At the same time, M outputs the result of $g(y)$ using only a logarithmic space in its work tape, since g is also a logarithmic space computable function. Finally, we obtain the output $\langle f(x), g(y)\rangle$ from the input $\langle x, y\rangle$ using the logarithmic space transducer M. Since we take an arbitrary *e–reduction*, then we prove the *e–reduction* is also a logarithmic space reduction. ☐

Theorem 2.4. *If $A \leq_{\equiv} B$ and $B \leq_{\equiv} C$, then $A \leq_{\equiv} C$.*

Proof. This is because of the logarithmic space reduction is transitive [2]. □

We say that a complexity class C is closed under reductions if, whenever L_1 is reducible to L_2 and $L_2 \in C$, then $L_1 \in C$ [2].

Theorem 2.5. *$\equiv L$ is closed under reductions.*

Proof. Let L and L' be two arbitrary languages, where their instances are ordered-pairs of binary strings. Suppose that $L \leq_{\equiv} L'$ where L' is in $\equiv L$. We will show that L is in $\equiv L$ too. By definition of $\equiv L$, there are two languages $L'_1 \in LSPACE$ and $L'_2 \in LSPACE$, such that for each $(v, w) \in L'$ we have that $v \in L'_1$ and $w \in L'_2$. Moreover, there are two deterministic logspace machines M'_1 and M'_2 which are the logspace verifiers of L'_1 and L'_2 respectively. For each $(v, w) \in L'$, there will be a succinct certificate z, such that $M'_1(v, z) =$ "*yes*" and $M'_2(w, z) =$ "*yes*". Besides, by definition of *e–reduction*, there are two logarithmic space computable functions $f : \{0, 1\}^* \to \{0, 1\}^*$ and $g : \{0, 1\}^* \to \{0, 1\}^*$ such that for all $x \in \{0, 1\}^*$ and $y \in \{0, 1\}^*$,

$$(x, y) \in L \textit{ if and only if } (f(x), g(y)) \in L'.$$

Based on this preliminary information, we can support that there exist two languages $L_1 \in LSPACE$ and $L_2 \in LSPACE$, such that for each $(x, y) \in L$ we have that $x \in L_1$ and $y \in L_2$. Indeed, we can define L_1 and L_2 as the strings $f^{-1}(v)$ and $g^{-1}(w)$, such that $f^{-1}(v) \in L_1$ and $g^{-1}(w) \in L_2$ if and only if $v \in L'_1$ and $w \in L'_2$. Certainly, for all $x \in \{0, 1\}^*$ and $y \in \{0, 1\}^*$, we can decide in logarithmic space whether $x \in L_1$ or $y \in L_2$ just verifying that $f(x) \in L'_1$ or $g(y) \in L'_2$ respectively, since $L'_1 \in LSPACE$, $L'_2 \in LSPACE$ and f and g are logarithmic space computable functions.

Furthermore, there are two deterministic logspace machines M_1 and M_2 that are the logspace verifiers of L_1 and L_2 respectively. For each $(x, y) \in L$, there will be a succinct certificate z such that $M_1(x, z) =$ "*yes*" and $M_2(y, z) =$ "*yes*". Indeed, we can know whether $M_1(x, z) =$ "*yes*" and $M_2(y, z) =$ "*yes*" just verifying whether $M'_1(f(x), z) =$ "*yes*" and $M'_2(g(y), z) =$ "*yes*". Certainly, for every triple of strings (x, y, z), we can define the logarithmic space computation of the verifiers M_1 and M_2 as $M_1(x, z) = M'_1(f(x), z)$ and $M_2(y, z) = M'_2(g(y), z)$, since we can evaluate $f(x)$ and $g(y)$ in logarithmic space, M'_1 and M'_2 are logspace machines and the logarithmic space computable transformations are closed under composition [2]. These compositions can be made in the same way that is composed the logarithmic space reductions [2]. Indeed, these compositions do not change the certificate, and therefore, they do not affect the read-once behavior of the special tape on these verifiers. In addition, $min(|f(x)|, |g(y)|)$ is polynomially bounded by $min(|x|, |y|)$ where $| \ldots |$ is the string length function, due to the logarithmic space transducers of f and g cannot output an exponential amount of symbols in relation to the size of the input. Consequently, $|z|$ is polynomially bounded by $min(|x|, |y|)$, because $|z|$ will be polynomially bounded by $min(|f(x)|, |g(y)|)$. Hence, we have just proved the necessary properties to state that L is in $\equiv L$. □

3. Class double-P

All the transformations that have been used in proving *P–completeness* are logarithmic space transformations [6]. We define a complexity class which has a close relation to this property.

Definition 3.1. *We say that a language L belongs to $2P$ if there exists a language $L' \in P$, such that*

$$L = \{(x, x) : x \in L'\}.$$

We call the complexity class $2P$ as "double–P". We represent this language L in $2P$ as (L', L').

We define the completeness of $2P$ using the *e–reduction*.

Definition 3.2. *A language $L \subseteq \{0, 1\}^*$ is 2P–complete if*

1. *$L \in 2P$, and*
2. *$L' \leq_{\equiv} L$ for every $L' \in 2P$.*

Furthermore, if L is a language such that $L' \leq_{\equiv} L$ for some $L' \in$ 2P–complete, then L is in 2P–hard. Moreover, if $L \in 2P$, then $L \in$ 2P–complete. This definitions are based on the result of Theorem 2.4.

We define *2–HORNSAT* as follows,

$$\text{2–HORNSAT} = \{(\phi, \phi) : \phi \in HORNSAT\}.$$

Theorem 3.3. *2–HORNSAT $\in$ 2P–complete.*

Proof. *2–HORNSAT* is in $2P$, because $HORNSAT$ is in P. Since $HORNSAT$ is in *P–complete*, then we obtain *2–HORNSAT* $\in$ *2P–complete*, because *2–HORNSAT* contains the elements of $HORNSAT$ but in a duplicated way. □

Definition 3.4. *MONOTONE–CNF is a problem in $\equiv L$, where every instance (ψ, φ) is an ordered-pair of Boolean formulas, such that if $(\psi, \varphi) \in$ MONOTONE–CNF, then we have $\psi \in$ UN–NEGATED and $\varphi \in$ NEGATED. According to the definition of $\equiv L$, this language is the ordered-pairs of elements of UN–NEGATED and NEGATED such that they will have the same satisfying truth assignment using the same variables.*

Theorem 3.5. *MONOTONE–CNF $\in$ 2P–hard.*

Proof. Given an arbitrary Boolean formula ϕ in CNF of n variables and m clauses which are Horn clauses, we iterate for $i = 1, 2, \ldots, m$ over each clause c_i in ϕ and analyze the following cases,

1. If the clause has all the variables negated, that is $c_i = (\neg x_1 \vee \neg x_2 \vee \ldots \vee \neg x_j)$, then we create a new variable x_{c_i} for this clause and create the formulas

$$Q_i = (x_{c_i})$$

$$P_i = (\neg x_{c_i} \vee \neg x_1 \vee \neg x_2 \vee \ldots \vee \neg x_j).$$

2. If the clause has all the variables negated except a single one, that is $c_i = (y \vee \neg x_1 \vee \neg x_2 \vee \ldots \vee \neg x_j)$, then we create a new variable x_{c_i} for this clause and create the formulas

$$Q_i = (y \vee x_{c_i})$$

$$P_i = (\neg x_{c_i} \vee \neg x_1 \vee \neg x_2 \vee \ldots \vee \neg x_j).$$

3. If the clause has a single un-negated variable, that is $c_i = (y)$, then we create a new variable x_{c_i} for this clause and create the formulas

$$Q_i = (y)$$

$$P_i = (\neg x_{c_i} \vee \neg y).$$

Finally, we create a new variable x_{final} and the formulas

$$Q_{final} = (x_{final} \vee x_1 \vee x_2 \vee \ldots \vee x_n \vee x_{c_1} \vee x_{c_2} \vee \ldots \vee x_{c_m})$$

$$P_{final} = (\neg x_{final} \vee \neg x_1 \vee \neg x_2 \vee \ldots \vee \neg x_n \vee \neg x_{c_1} \vee \neg x_{c_2} \vee \ldots \vee \neg x_{c_m})$$

where these final formulas contain all the variables x_j of ϕ and the created variables x_{c_i} for each clause c_i.

Every clause c_i is satisfiable if and only if both formulas Q_i and P_i in each case have a satisfying truth assignment such that the evaluation in the common variables coincide. Hence, we can construct the Boolean formulas ψ and φ as the conjunction of Q_i or P_i for every clause c_i in ϕ, that is, $\psi = Q_1 \wedge \ldots \wedge Q_m \wedge Q_{final}$ and $\varphi = P_1 \wedge \ldots \wedge P_m \wedge P_{final}$. The last constructed formulas Q_{final} and P_{final} guarantee the Boolean formulas ψ and φ contain the same variables. Finally, we obtain that,

$$(\phi, \phi) \in \text{2–HORNSAT if and only if } (\psi, \varphi) \in \text{MONOTONE–CNF}.$$

Moreover, there are two logarithmic space computable functions $f : \{0,1\}^* \to \{0,1\}^*$ and $g : \{0,1\}^* \to \{0,1\}^*$ such that $f(\langle\phi\rangle) = \langle\psi\rangle$ and $g(\langle\phi\rangle) = \langle\varphi\rangle$. Indeed, we only need a logarithmic space to analyze every time each clause c_i in the instance ϕ and generate Q_i or P_i to the output. Finally, we can also create the formulas Q_{final} and P_{final} in logarithmic space. Then, we have proved that *2–HORNSAT* $\leq_{\equiv}$ *MONOTONE–CNF*. Consequently, we obtain that *MONOTONE–CNF* $\in$ *2P–hard*. □

Theorem 3.6. $2P \subseteq \equiv L$.

Proof. Since *MONOTONE–CNF* is hard for $2P$, thus all language in $2P$ reduce to $\equiv L$. Since $\equiv L$ is closed under reductions, it follows that $2P \subseteq \equiv L$. □

4. Class equivalent-NL

Definition 4.1. *We say that a language L belongs to $\equiv NL$ if there exist two languages $L_1 \in NL$ and $L_2 \in NL$ and two deterministic logspace machines M_1 and M_2, where M_1 and M_2 are the logspace verifiers of L_1 and L_2 respectively, such that*

$$L = \{(x, y) : \exists z \text{ such that } M_1(x, z) = \text{“yes” and } M_2(y, z) = \text{“yes”}\}.$$

We call the complexity class $\equiv NL$ as “equivalent–NL”. We represent this language L in $\equiv NL$ as (L_1, L_2).

Theorem 4.2. $\equiv NL \subseteq NL$.

Proof. Let's take an arbitrary language L in $\equiv NL$ which is defined from the two languages $L_1 \in NL$ and $L_2 \in NL$ and two deterministic logspace machines M_1 and M_2, such that $L = (L_1, L_2)$ where M_1 and M_2 are the logspace verifiers of L_1 and L_2 respectively. We are going to prove there is a deterministic logspace machine M, such that M will be the logspace verifiers of L. M contains the read/write work tapes of M_1 and the ones from M_2. Besides, it will contain three additional read/write work tapes. Now, we could define M in the following way:

For some input (x, y) and a certificate u on the special read-once tape,

1. We simulate the logspace machines M_1 over the string x on the input tape using the read/write work tapes of M_1 which contains M. For each step, it is stored the current position of the head over the input tape just using one of the three additional read/write work tapes. This position can be calculated just adding 1 if the head moves to the right, or subtracting 1 if the head moves to the left or adding 0 if it remains in the same place on the input tape. This simulation ends when the program of M_1 makes a single forward step on the special read-once tape that contains the certificate u. At this point, it is saved the position of the head in the input tape in one of the three additional read/write work tapes without change it. Besides, it will store in another of the three additional read/write work tapes the previous symbol that is on the special read-once tape to the left of where the head currently is, that is the previous read symbol. This is done before the program of M_1 makes this single forward step on the special read-once tape, and thus, it will not violate the definition of the special read-once tape.
2. Then, M starts to simulate the logspace machines M_2 over the string y on the input tape using the read/write work tapes of M_2 which contains M. For each step, it is stored the current position of the head over the input tape just using the last of the three additional read/write work tapes. This position can be calculated just adding 1 if the head moves to the right, or subtracting 1 if the head moves to the left or adding 0 if it remains in the same place on the input tape. But, in this simulation instead of using the special read-once tape to read the certificate u, it will use the previous symbol that was stored in one of the three additional read/write work tapes. This simulation ends when the program of M_2 tries to make a single forward step on the special read-once tape that contains the certificate u. But in this case, instead of moving forward on the special read-once tape, it will replace the previous symbol that is stored in one of the three additional read/write work tapes by the blank symbol. At this point, it is saved the position of the head in the input tape in the last of the three additional read/write work tapes without change it.
3. Next, it will take the last position that was stored over the string x on the input tape that was used in the previous simulation of M_1 that will contain one of the three additional read/write work tapes. And thus, the head of the input tape will be placed in that position and it will continue the simulation of M_1 over the read/write work tapes of M_1 which contains M. Hence, M will make again the same actions of the step (1) that we already explained. After that, it will continue with the actions of the step (2) be placing the head of the input tape in the last position over the string y in the input tape that was already stored, and thus, it will repeat the same routine over and over again until one of the logspace machine M_1 or M_2 halts.
4. When one of the logspace machine M_1 or M_2 halts and the final state is the acceptance state "yes", then M will simulate in a normal way the other machine that have not halted yet. That means it will not alternate at the same time the simulations of M_1 and M_2, but only one of them. In this way, if M_1 is the first machine that halts, then M simulates M_2

over the string y from the last position on the input tape in a normal way, that is using only the special read-once tape and not the read/write work tape that we were using in the step (2). The same happens if M_2 is the first machine that halts. However, if the simulation that firstly halts over the possible logspace machine M_1 or M_2 has reached the reject state, then machine M will reject too.

5. Finally, M accepts whether the final and normal simulation, that continues after the first halted simulation, just halts in the acceptance state otherwise M will reject.

Certainly, the Turing machine M can be deterministic, because the logspace machines M_1 and M_2 are deterministic. Besides, the Turing machine M is a logspace machine, since the read/write work tapes of M_1 and M_2 which contains M only use a logarithmic space. Furthermore, the twice storing of the last position in the input tape over the string x and y in the additional read/write work tapes can only use a logarithmic space if we represent that position as a binary number. Moreover, the additional read/write work tape that saves the previous symbol, that is on the special read-once tape to the left of where the head currently is, can store that information into at most one single symbol on that tape. In addition, M will accept the input (x, y) with a certificate u on the special read-once tape if and only if $(x, y) \in L$. In this way, M will be a logspace verifiers of L, and therefore, we obtain that $L \in NL$. Since we took arbitrarily a language L in $\equiv NL$, then we can deduce $\equiv NL \subseteq NL$. □

Lemma 4.3. $\equiv L \subseteq \equiv NL$.

Proof. Since $LSPACE \subseteq NL$, then we can support that $\equiv L \subseteq \equiv NL$ as a direct consequence of the Definitions 2.1 and 4.1 [2]. □

Theorem 4.4. $NL = P$.

Proof. As result of applying the Theorems 3.6 and 4.2 with the Lemma 4.3, then we can deduce $2P \subseteq NL$. Alternatively, we can reduce in logarithmic space every language in $L \in P$ to another language $(L, L) \in 2P$ just using a logarithmic space transducer that copies the content on its input tape to the output twice. Since NL is closed under logarithmic space reductions and every language in $2P$ is in NL, then every language in P will be in NL too [2]. Consequently, we obtain that $P \subseteq NL$. However, we already know that $NL \subseteq P$ [2]. Since we have that $P \subseteq NL$ and $NL \subseteq P$, then $NL = P$ [7]. □

References

[1] S. A. Cook, The P versus NP Problem, available at http://www.claymath.org/sites/default/files/pvsnp.pdf (April 2000).

[2] C. H. Papadimitriou, Computational Complexity, Addison-Wesley, 1994.

[3] L. Fortnow, The Golden Ticket: P, NP, and the Search for the Impossible, Princeton University Press. Princeton, NJ, 2013.

[4] O. Goldreich, P, Np, and Np-Completeness, Cambridge: Cambridge University Press, 2010.

[5] M. Sipser, Introduction to the Theory of Computation, 2nd Edition, Thomson Course Technology, 2006.

[6] S. Arora, B. Barak, Computational complexity: A modern approach, Cambridge University Press, 2009.

[7] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, Introduction to Algorithms, 2nd Edition, MIT Press, 2001.

[8] M. R. Garey, D. S. Johnson, Computers and Intractability: A Guide to the Theory of NP-Completeness, 1st Edition, San Francisco: W. H. Freeman and Company, 1979.