UP versus NP

Frank Vega

UP versus NP

Frank Vega

Abstract

The class equivalent-P contains those languages that are ordered-pairs of instances of two specific problems in P, such that the elements of each ordered-pair have the same solution, which means, the same certificate. The class equivalent-UP has almost the same definition, but in each case we define the pair of languages explicitly in UP. In addition, we define the class double-NP as the set of languages that contain each instance of another language in NP, but in a double way, that is, in form of a pair with two identical instances. We show that UP = NP using these classes.

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

Introduction

P versus *NP* is a major unsolved problem in computer science [1]. This problem was introduced in 1971 by Stephen Cook [2]. It 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].

In 1936, Turing developed his theoretical computational model [2]. 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 [3]. 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 [3].

Another huge 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 [4]. A complexity class is a set of problems, which are represented as a language, grouped by measures such as the running time, memory, etc [4].

In the computational complexity theory, the class *P* contains those languages that can be decided in polynomial time by a deterministic Turing machine [5]. The class *NP* consists on those languages that can be decided in polynomial time by a nondeterministic Turing machine [5].

The biggest open question in theoretical computer science concerns the relationship between these classes:

Is *P* equal to *NP*?

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

Another major complexity class is UP. The class UP has all the languages that are decided in polynomial time by a nondeterministic Turing machines with at most one accepting computation for each input [6]. The nondeterministic Turing machines which decide the languages in the class UP are called unambiguous Turing machines [3]. It is obvious that $P \subseteq UP \subseteq NP$ [3]. Whether $P = UP$ is another fundamental question that it is as important as it is unresolved [3]. All efforts to solve the P versus UP problem have failed [3]. Nevertheless, we prove that $UP = NP$.

1. Theoretical notions

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 [2]. Note that M fails to accept w either if this computation ends in the rejecting state, or if the computation fails to terminate [2].

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

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

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 exists k such that for all n, $T_M(n) \leq n^k + k$ [2].

Definition 1.1. *A language L is in class P if $L = L(M)$ for some deterministic Turing machine M which runs in polynomial time [2].*

We state the complexity class NP using the following definition.

Definition 1.2. *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 [7]. 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.

Definition 1.3. *NP is the class of languages that have polynomial time verifiers [7].*

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 [7]. Let $\{0, 1\}^*$ be the infinite set of binary strings, we say that a language L_1 is polynomial time reducible to a language L_2, 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 $x \in \{0, 1\}^*$,

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

An important complexity class is *NP–complete* [5]. 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* [4]. Moreover, if $L \in NP$, then $L \in$ *NP–complete* [4]. If any single *NP–complete* problem can be solved in polynomial time, then every NP problem has a polynomial time algorithm [4]. No polynomial time algorithm has yet been discovered for an *NP–complete* problem [1].

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 $3SAT$ [4]. We define $3CNF$ satisfiability using the following terms. A literal in a Boolean formula is an occurrence of a variable or its negation [4]. 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 [4]. A Boolean formula is in 3-conjunctive normal form or $3CNF$, if each clause has exactly three distinct literals [4].

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.

It can be demonstrated that many problems belong to *NP–complete* using the polynomial time reduction from $3SAT$ [8]. For example, the well-known problem *1–IN–3 3SAT* which is defined as follows: Given a Boolean formula ϕ in $3CNF$, is there a truth assignment such that each clause in ϕ has exactly one true literal?

Another special case is the class of problems where each clause contains *XOR* (i.e. exclusive or) rather than (plain) *OR* operators. This is in *P*, since a *XOR SAT* formula can also be viewed as a system of linear equations mod 2, and can be solved in cubic time by Gaussian elimination [9]. We represent the *XOR* function inside a Boolean formula as $\oplus$. The problem *XOR 3SAT* is similar to *XOR SAT*, but the clauses in the Boolean formula have exactly three distinct literals. Since $a \oplus b \oplus c$ is evaluated as true if and only if exactly 1 or 3 members of $\{a, b, c\}$ are true, then each solution of the problem *1–IN–3 3SAT* for a given $3CNF$ formula is also a solution of the problem *XOR 3SAT* and in turn each solution of *XOR 3SAT* is a solution of $3SAT$.

In addition, a Boolean formula is in 2-conjunctive normal form, or $2CNF$, if it is in CNF and each clause has exactly two distinct literals. There is a well-known problem called $2SAT$. In $2SAT$, it is asked whether a given Boolean formula ϕ in $2CNF$ is satisfiable. This language is in P [10].

2. Class equivalent-P

Definition 2.1. *We say that a language L belongs to $\equiv P$ if there exist two languages $L_1 \in P$ and $L_2 \in P$ and two deterministic Turing machines M_1 and M_2, where M_1 and M_2 are the polynomial time 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 $\equiv P$ as “equivalent–P”. We represent this language L in $\equiv P$ as (L_1, L_2). The order in the pairs of strings of a problem in $\equiv P$ is really important.

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 [7]. The work tape may contain $O(\log n)$ symbols [7]. 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 [7]. We call f a logarithmic space computable function [7]. We say that a language L_1 is logarithmic space reducible to a language L_2, 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 frequently used for P and below [3]. 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 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 $y \in \{0, 1\}^*$,*

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

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* which 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 [3]. 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 a consequence of Lemma 2.3, because the logarithmic space reduction is transitive [3]. □

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 $L_1 \in C$ [3].

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

Proof. Let L and L' be two arbitrary languages, where their instances are ordered-pairs of strings. Suppose that $L \leq_{\equiv} L'$ where L' is in $\equiv P$. We will show that L is in $\equiv P$ too.

By definition of $\equiv P$, there are two languages $L'_1 \in P$ and $L'_2 \in P$, 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 Turing machines M'_1 and M'_2 which are the polynomial time 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 $f : \{0, 1\}^* \rightarrow \{0, 1\}^*$ and $g : \{0, 1\}^* \rightarrow \{0, 1\}^*$ 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 $L_1 \in P$ and $L_2 \in P$, 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 ordered-pairs of strings $(f^{-1}(v), g^{-1}(w))$, 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 polynomial time 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 $L'_1 \in P$, $L'_2 \in P$, and $SPACE(\log n) \subseteq P$ [3].

Furthermore, there are two deterministic Turing machines M_1 and M_2 which are the polynomial time 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 polynomial time 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 polynomial time because of $SPACE(\log n) \subseteq P$ [3]. In addition, $max(|f(x)|, |g(y)|)$ 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 $\equiv P$. ☐

3. Class double-NP

It has been observed that most of the transformations used in proving *NP–completeness* are also logarithmic space transformations [8]. Thus the class of languages that are "logarithmic space complete for *NP*" is at least a large subclass of the *NP–complete* problems [8]. We define a complexity class which has a close relation to this property.

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

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

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

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

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

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

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

We define *double–1–IN–3 3SAT* as follows,

$$\textit{double–1–IN–3 3SAT} = \{(\phi, \phi) : \phi \in \textit{1–IN–3 3SAT}\}.$$

Theorem 3.3. *double–1–IN–3 3SAT ∈ 2NP–complete.*

Proof. *double–1–IN–3 3SAT* is in $2NP$, because *1–IN–3 3SAT* is in NP. We already know that the language *1–IN–3 3SAT* has been defined in *NP–complete* using logarithmic space reductions [8]. Certainly, we can reduce any instance of a language in NP to SAT, and this other instance of SAT in $3SAT$, and finally, the last instance in $3SAT$ to *1–IN–3 3SAT* just using in each case a reduction in logarithmic space [8]. Since the logarithmic space reduction is transitive, then *double–1–IN–3 3SAT ∈ 2NP–complete*. □

Definition 3.4. *3XOR–2SAT is a problem in ≡ P, where every instance (ψ, φ) is an ordered-pair of Boolean formulas, such that if $(\psi, \varphi) \in$ 3XOR–2SAT, then $\psi \in$ XOR 3SAT and $\varphi \in 2SAT$. By the definition of ≡ P, this language is the ordered-pairs of instances of XOR 3SAT and $2SAT$ such that they will have the same satisfying truth assignment using the same variables.*

Theorem 3.5. *3XOR–2SAT ∈ 2NP–hard.*

Proof. Given an arbitrary Boolean formula ϕ in $3CNF$ of m clauses, we iterate for $i = 1, 2, \ldots, m$ over each clause $c_i = (x \vee y \vee z)$ in ϕ, where x, y and z are literals, just creating the following formulas,

$$Q_i = (x \oplus y \oplus z)$$

$$P_i = (\neg x \vee \neg y) \wedge (\neg y \vee \neg z) \wedge (\neg x \vee \neg z).$$

Since Q_i is evaluated as true if and only if exactly 1 or 3 members of $\{x, y, z\}$ are true and P_i is evaluated as true if and only if exactly 1 or 0 members of $\{x, y, z\}$ are true, then we obtain the clause c_i has exactly one true literal if and only if both formulas Q_i and P_i have the same satisfying truth assignment.

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, $\psi = Q_1 \wedge \ldots \wedge Q_m$ and $\varphi = P_1 \wedge \ldots \wedge P_m$. Finally, we obtain that,

$$(\phi, \phi) \in \textit{double–1–IN–3 3SAT if and only if } (\psi, \varphi) \in \textit{3XOR–2SAT}.$$

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, since the complexity class $SPACE(\log n)$ does not take the length of the content on the input and output tapes into consideration [3]. Then, we have proved that *double–1–IN–3 3SAT* $\leq_{\equiv}$ *3XOR–2SAT*. Consequently, we obtain that *3XOR–2SAT ∈ 2NP–hard*. □

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

Proof. Since *3XOR–2SAT* is hard for $2NP$, thus all language in $2NP$ reduce to $\equiv P$. Since $\equiv P$ is closed under reductions, it follows that $2NP \subseteq \equiv P$. □

4. Class equivalent-UP

Definition 4.1. *We say that a language L belongs to $\equiv UP$ if there exist two languages $L_1 \in UP$ and $L_2 \in UP$ and two deterministic Turing machines M_1 and M_2, where M_1 and M_2 are the polynomial time 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 $\equiv UP$ as “equivalent–UP”. We represent this language L in $\equiv UP$ as (L_1, L_2). The order in the pairs of strings of a problem in $\equiv UP$ is really important.

Theorem 4.2. $\equiv UP \subseteq UP$.

Proof. Let's take an arbitrary language $L \in \equiv UP$ defined from the two languages $L_1 \in UP$ and $L_2 \in UP$ and the two deterministic Turing machines M_1 and M_2 such that $L = (L_1, L_2)$ where M_1 and M_2 are the polynomial time verifiers of L_1 and L_2 respectively. We can construct a nondeterministic Turing machine M such that M can decide every instance (x, y) of L in polynomial time with at most one accepting computation for each input. Certainly, since $UP \subseteq NP$, then every certificate z of the instances $x \in L_1$ or $y \in L_2$ can be polynomially bounded using a single constant c such that $|z| < |x|^c$ or $|z| < |y|^c$ where $|\ldots|$ is the string length function.

We define M in the following way: on input (x, y),

1. we nondeterministically generate a single string z with at most $max(|x|, |y|)^c$ symbols from the alphabets of M_1 and M_2 in unambiguous way,
2. then, we accept the instance (x, y) if $M_1(x, z) =$ “yes” and $M_2(y, z) =$ “yes” otherwise we reject.

For every instance x of L_1 if there exists a string z for x such that this proves the membership in L_1, then z is the only certificate that has x because $L_1 \in UP$. The same happens with every instance y in L_2: if there exists a certificate z for y that proves $y \in L_2$, then this is the only one for y. If $x \notin L_1$ or $y \notin L_2$, then there will not be another succinct certificate z for x or y. Indeed, UP should not be confused with the class US of problems that ask whether a given instance has a unique solution [3]. Hence, for every instance (x, y), there will be at most one polynomially bounded string z such that $M_1(x, z) =$ “yes” and $M_2(y, z) =$ “yes” if and only if $(x, y) \in L$.

Since the Turing machines M_1 and M_2 are deterministic and the generation of z can be done in unambiguous way, then we can support that M has at most one accepting computation for every instance (x, y) of L. In addition, the Turing machine M is nondeterministic due to the nondeterministic steps in the selection of the string z. Consequently, we obtain that $L \in UP$. Since we took $L \in \equiv UP$ in an arbitrary way, then $\equiv UP \subseteq UP$. □

Lemma 4.3. $\equiv P \subseteq \equiv UP$.

Proof. Since $P \subseteq UP$, then we can support that $\equiv P \subseteq \equiv UP$ as a direct consequence of the Definitions 2.1 and 4.1 [3]. □

Theorem 4.4. $UP = NP$.

Proof. As result of the Theorems 3.6 and 4.2 with the Lemma 4.3, we can also support that $2NP \subseteq UP$. In addition, we can reduce in logarithmic space every language in $L \in NP$ to another language $(L, L) \in 2NP$ just using a logarithmic space transducer that copies the content on its input tape to the output twice. Since every language in $2NP$ is in UP and UP is closed under logarithmic space reductions, then every language in NP is in UP too [3]. Consequently, we obtain that $NP \subseteq UP$. However, we already know that $UP \subseteq NP$ [3]. Since we have that $NP \subseteq UP$ and $UP \subseteq NP$, then $UP = NP$ [4]. □

Conclusions

There is a previous known result which states that $P = UP$ if and only if there are no one-way functions [3]. Indeed, for many years it has been accepted the P versus UP question as the correct complexity context for the discussion of the cryptography and one-way functions [3]. For that reason, the proof of Theorem 4.4 negates this accepted idea and also the belief that $UP = NP$ is a very unlikely event. In addition, this demonstration might be a shortcut to prove $P = NP$, because if somebody proves that $P = UP$, then he will be proving the outstanding and difficult P versus NP problem too [1]. Furthermore, if we have a possible proof of $P \neq NP$, then this work would also contribute to prove that $P \neq UP$.

References

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

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

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

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

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

[6] L. G. Valiant, Relative Complexity of Checking and Evaluating, Inf. Process. Lett. 5 (1976) 20–23.

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

[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.

[9] C. Moore, S. Mertens, The Nature of Computation, Oxford University Press, 2011.

[10] B. Aspvall, M. F. Plass, R. E. Tarjan, A Linear-Time Algorithm for Testing the Truth of Certain Quantified Boolean Formulas, Information Processing Letters 8 (3) (1979) 121–123. doi:10.1016/0020-0190(79)90002-4.