P vs NP

Frank Vega

P VS NP

FRANK VEGA

Abstract. $UNIQUE\ SAT$ is the problem of deciding whether a given Boolean formula has exactly one satisfying truth assignment. The $UNIQUE\ SAT$ is $coNP - hard$. We prove the $UNIQUE\ SAT$ is in NP, and therefore, $NP = coNP$. Furthermore, we prove if $NP = coNP$, then some problem in $coNPC$ is in P, and thus, $P = NP$. In this way, the P versus NP problem is solved with a positive answer.

1. Introduction

The P versus NP problem is the major unsolved problem in computer science. It was introduced in 1971 by Stephen Cook [2]. Today is considered by many scientists as the most important open problem in this field [4].

During the first half of the twentieth century many investigations were focused on formalizes the knowledge about the algorithms using the theoretical model described by Turing Machines. On this time appeared the first computers and the mathematicians were able to model the capabilities and limitations of such devices appearing precisely what is now known as the science of computational complexity theory.

Since the beginning of computation, many tasks that man could not do, were done by computers, but sometimes some difficult and slow to resolve were not feasible for even the fastest computers. The only way to avoid the delay was to find a possible method that cannot do the exhaustive search that was accompanied by "brute force". Even today, there are problems which have not a known method to solve easily yet.

If $P = NP$, then it would ensure that there are hundreds of problems that have a feasible solution. This is largely derived from this result that there will be a huge amount of problems that can be checked easily and have some practical solution at the same time [8].

The studies of this incognita brought along new unsolved questions such as the NP versus $coNP$ problem. We show in this work the $UNIQUE\ SAT$ problem belongs to NP, and in this way, we prove the complexity classes NP and $coNP$ are equals, where $coNP$ represents the complements of languages in NP. It is a proved result if $P = NP$, then $NP = coNP$ [7]. Moreover, we prove if $NP = coNP$, then $P = NP$ and for that reason $P = NP$.

2. Theory

The argument made by Alan Turing in the twentieth century proves mathematically that for any computer program we can create an equivalent Turing Machine

2000 *Mathematics Subject Classification.* Primary 68-XX, 68Qxx, 68Q15.

Key words and phrases. Complexity class, P, NP, coNP, Turing Machine, Language.

[9]. A Turing Machine M has a finite set of states K and a finite set of symbols A called the alphabet of M. The set of states has a special state s which is known as the initial state. The alphabet contains special symbols such as the start symbol $\triangleright$ and the blank symbol $\$$.

The operations of a Turing Machine are based on a transition function δ, which takes the initial state with a string of symbols of the alphabet that is known as the input. Then, it proceeds to reading the symbols on the cells contained in a tape, through a head or cursor. At the same time, the symbols on each step are erased and written by the transition function, and later moved to the left $\longleftarrow$, right $\longrightarrow$ or remained in the same place $-$ for each cell. Finally, this process is interrupted if it halts in a final state: the state of acceptance "*yes*", the rejection "*no*" or halting h [7].

A Turing Machine halts if it reaches a final state. If a Turing Machine M accepts or rejects a string x, then $M(x) =$ "*yes*" or "*no*" is respectively written. If it reaches the halting state h , we write $M(x) = y$, where the string y is considered as the output string, i.e., the string remaining in M when this halts [7].

A transition function δ is also called the "program" of the Turing Machine and is represented as the triple $\delta(q, \sigma) = (p, \rho, D)$. For each current state q and current symbol σ of the alphabet, the Turing Machine will move to the next state p, overwriting the symbol σ by ρ, and moving the cursor in the direction $D \in \{\longleftarrow, \longrightarrow, -\}$ [7]. When there is more than one tape, δ remains deciding the next state, but it can overwrite different symbols and move in different directions over each tape.

Operations by a Turing Machine are defined using a configuration that contains a complete description of the current state of the Machine. A configuration is a triple (q, w, u) where q is the current state and w, u are strings over the alphabet showing the string to the left of the cursor including the scanned symbol and the string to the right of the cursor respectively and this is during any instant in which there is a transition on δ [7]. The configuration definition can be extended to multiple tapes using the corresponding cursors.

A deterministic Turing Machine is a Turing Machine that has only one next action for each step defined in the transition function [6], [5]. However, a non-deterministic Turing Machine can contain more than one action defined for each step of the program, where this program was no longer a function but a relation [6], [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 [3]. There are two complexity classes that have a close relationship with the previous concepts and are represented as P and NP. In computational complexity theory, the class P contains the languages that are decided by a deterministic Turing Machine in polynomial time [6]. The class NP contains the languages that are decided by a non-deterministic Turing Machines in polynomial time [5].

Moreover, a language $L \in NP$ if there is a polynomial-time decidable and polynomially balanced relation R_L such that for all strings x: there is a string y with $R_L(x, y)$ if and only if $x \in L$ [7]. This string y is known as certificate. If NP is the class of problems that have succinct certificates, then the complexity class $coNP$ must contain those problems that have succinct disqualifications [7]. That is, a "*no*" instance of a problem in $coNP$ possesses a short proof of its being a "*no*" instance; and only "*no*" instances have such proofs [7].

There are another derived complexity classes from NP and $coNP$ that are $NP-complete$ and $coNP - complete$ denoted as NPC and $coNPC$. We show a NPC problem and other that is in $coNPC$ in the following paragraph:

The problem $ONE - IN - THREE$ $3SAT$ is the following: Given a Boolean formula ϕ in $3CNF$, is there a truth assignment that satisfies ϕ such that each clause in ϕ has exactly one true literal? The problem $UNSATISFIABLE$ $3SAT$ is the following: Given a Boolean formula ϕ in $3CNF$, does ϕ have not any satisfying truth assignment?

On the other hand, the problem $UNIQUE$ SAT is the following: Given a Boolean formula ϕ, is it true that it has a unique satisfying truth assignment? If $UNIQUE$ SAT is in NP, then $NP = coNP$ [1].

3. Results

We are going to assume the reader has basic elements of computational complexity. However, if you have not any knowledge about this science, it will be very useful to take a look before to the Theory section even though it has a very basic content.

3.1. NP = coNP.

We start clarifying that all the elements that we mentioned in this section, such as a Boolean formula ϕ and so forth, are assumed as they were in a binary encoding when they are used as input for a Turing Machine to make more clear our arguments.

Definition 3.1. *Let M_{SAT} be a Turing Machine which decides for a Boolean formula ϕ and a truth assignment T if $T \models \phi$, that is when the truth assignment T satisfies ϕ* [7].

When $M_{SAT}(\phi, T) =$ "*yes*", then the existence of the mapping of the Boolean variables to the values of $true$ or $false$, that is T, implies that the formula ϕ is satisfiable. When $M_{SAT}(\phi, T) =$ "*no*", we could make the conclusion that ϕ is not a tautology or T is not an appropriate truth assignment for ϕ [7].

Lemma 3.2. *The Turing Machine M_{SAT} could be deterministic and make their decisions in polynomial time.*

This is a direct consequence of $SAT \in NP$, because M_{SAT} defines the polynomial-time decidable and polynomially balanced relation R_{SAT} which relates a Boolean formula ϕ with a truth assignment T if and only if $M_{SAT}(\phi, T) =$ "*yes*".

Lemma 3.3. *The deterministic Turing Machine M_{SAT} could have only one tape and always accept in the configuration ("yes", $\triangleright, \phi$) in polynomial time with the input $\phi\$T$ when $T \models \phi$ and where \$ is the blank symbol. Besides, the transition function of M_{SAT} will visit the initial state only in the first action where it will read the start symbol in the computation of any input.*

The language $SAT \in NP$ has a deterministic Turing Machine which decides in polynomial time the polynomially balanced relation R_{SAT}. This Turing Machine could be transformed into another Turing Machine of one tape which has a polynomial time in relation with the running time of the original [7]. Therefore, the deterministic Turing Machine that decides R_{SAT} could be of one tape.

If the initial state is visited more than once in the transition function of this one-tape deterministic Turing Machine, then we will create a new initial state replacing the old one that it will only read the start symbol in the first action and continue the normal execution.

This one-tape deterministic Turing Machine can be transformed into two-tapes deterministic Turing Machine that receives the input in the first tape. This new Turing Machine will copy the input in the second tape and simulate the original one-tape Turing Machine on this second tape. When the simulation of the original Turing Machine accepts, it will delete the content in the second tape and remove the certificate T from the first tape. Finally, it will set the cursors in the start symbol of each tape and halt in the state of acceptance. In case of rejection, the two-tapes deterministic Turing Machine will reject too. This new Turing Machine can be transformed into a one-tape Turing Machine M_{SAT} complying with the Lemma 3.3.

Theorem 3.4. *The deterministic Turing machine M_{SAT}, which complies with the Lemma 3.3, can be inverted into a non-deterministic Turing machine N_{SAT}.*

We are going to change the transition function δ of the deterministic Turing machine M_{SAT} of Lemma 3.3 in the following way:

$$\forall p, q, r \in K \wedge \forall \sigma_1, \sigma_2, \rho_1, \rho_2 \in A : \tag{1}$$
$$[\delta(q, \sigma_1) = (p, \rho_1, D_p) \wedge \delta(r, \sigma_2) = (q, \rho_2, D_q)] \Longrightarrow \tag{2}$$
$$([\delta^{'}(q, \rho_1) = (r, \sigma_1, D^{'}_q)] \tag{3}$$
$$\wedge\ [(D_q = \longleftarrow) \Longrightarrow (D^{'}_q = \longrightarrow)] \tag{4}$$
$$\wedge\ [(D_q = \longrightarrow) \Longrightarrow (D^{'}_q = \longleftarrow)] \tag{5}$$
$$\wedge\ [(D_q = -) \Longrightarrow (D^{'}_q = -)]) \tag{6}$$

This new program $\delta^{'}$ will represent a new Turing machine N_{SAT} where K and A are the set of states and alphabet of M_{SAT} respectively. The initial state of M_{SAT} will be replaced by the state of acceptance in N_{SAT}. The state of acceptance in M_{SAT} will be replaced by the initial state in N_{SAT} with the following actions:

$$\forall q \in K \wedge \forall \sigma_1, \rho_1 \in A : \tag{7}$$
$$[\delta(q, \sigma_1) = (\text{``}yes\text{''}, \rho_1, D_{\text{``}yes\text{''}})] \Longrightarrow \tag{8}$$
$$([\delta^{'}(s, \rhd) = (q, \rhd, D^{'}_{\text{``}yes\text{''}})] \tag{9}$$
$$\wedge\ [(D_{\text{``}yes\text{''}} = \longleftarrow) \Longrightarrow (D^{'}_{\text{``}yes\text{''}} = \longrightarrow)] \tag{10}$$
$$\wedge\ [(D_{\text{``}yes\text{''}} = -) \Longrightarrow (D^{'}_{\text{``}yes\text{''}} = -)]) \tag{11}$$

We define the rejection state in N_{SAT} in the following way: for every q state in the set of states of N_{SAT} except for the state of acceptance and every σ symbol of its alphabet, if there is no action in $\delta^{'}$ such that from the state q we could read the σ symbol, then $\delta^{'}(q, \sigma) = (\text{``}no\text{''}, \sigma, -)$.

In N_{SAT} over the M_{SAT} construction it is achieved that in almost all states, those symbols that are read in the tapes and the symbols that overwrite them are exchanged among them during the steps of M_{SAT} transition function. It is also possible to verify that N_{SAT} is nearly a "mirror" of M_{SAT} and transits backwards along M_{SAT} states. The N_{SAT} Turing machine in every state is directed towards predecessors appearing in transaction function δ, thus changing the movement direction of M_{SAT} simulating "backwards". This new Turing machine N_{SAT} will be a non-deterministic Turing machine.

Lemma 3.5. *The non-deterministic Turing machines N_{SAT} accepts in polynomial time.*

The configuration in the state of acceptance in N_{SAT} will be ("*yes*", $\rhd, \phi\$T$), where $\phi\$T$ is the original input of M_{SAT} when $\phi \in SAT$ with the certificate T. The non-deterministic Turing machines N_{SAT} could accept in polynomial time, because the amount of steps in the execution of $N_{SAT}(\phi)$ when $N_{SAT}(\phi) =$ "*yes*" could be at most equal to the number of actions in the longer running time of $M_{SAT}(\phi, T)$ for all T if $M_{SAT}(\phi, T) =$ "*yes*". Moreover, the binary relation R_{SAT} which defines the language SAT in NP is polynomially balanced, and therefore, if the longer running time of $M_{SAT}(\phi, T)$ for all T if $M_{SAT}(\phi, T) =$ "*yes*" is of order $O(\mid \phi\$T \mid^d)$, where $\mid \phi\$T \mid$ is the size of $\phi\$T$, then the execution of $N_{SAT}(\phi)$ will be of order $O(\mid \phi \mid^k)$ where k could be always a fixed and small constant.

Definition 3.6. *Let $M^{'}_{SAT}$ be a Turing Machine which has all the properties and the same behavior of the deterministic Turing machine M_{SAT} of Lemma 3.3, except that for a specific and single input $\phi^{'}\$T^{'}$ we have that $M^{'}_{SAT}(\phi^{'}, T^{'}) =$ "no" when $M_{SAT}(\phi^{'}, T^{'}) =$ "yes".*

Indeed, $M^{'}_{SAT}$ continues deciding each Boolean formula ϕ with a truth assignment T if $T \models \phi$, except that only for the formula $\phi^{'}$ and truth assignment $T^{'}$ the Turing Machine $M^{'}_{SAT}$ makes a rejection even though $T^{'} \models \phi^{'}$.

Theorem 3.7. *We could build $M^{'}_{SAT}$ adding a polynomial amount of states and actions inside of M_{SAT} in relation with the size of the input $\phi^{'}\$T^{'}$ and rejecting $\phi^{'}\$T^{'}$ in polynomial time.*

We could add the following actions to the transition function of M_{SAT}:

$$Iterating\ in\ reverse\ order\ through\ the\ symbols\ of\ \phi^{'}\$T^{'} \tag{12}$$

$$We\ take\ each\ symbol\ \sigma_i\ from\ the\ i-th\ position \tag{13}$$

$$Adding\ the\ states\ p_i\ and\ p_{i+1}\ if\ they\ do\ not\ exist\ in\ K \tag{14}$$

$$With\ a\ new\ action: \tag{15}$$

$$\delta(p_i, \sigma_i) = (p_{i+1}, \sigma_i, \longrightarrow) \tag{16}$$

We rename the initial state of M_{SAT} as ss and we add a new action to the state p_1 that represents the state which reads the first symbol of $\phi^{'}\$T^{'}$.

$$\delta(s, \rhd) = (p_1, \rhd, \longrightarrow) \tag{17}$$

With the previous action we recreate the initial state in M_{SAT}. Then, we add another action to the state $p_{|\phi^{'}\$T^{'}|+1}$ that is the other state in the first action created in this transformation related with $p_{|\phi^{'}\$T^{'}|}$ which represents the state that reads the last symbol of $\phi^{'}\$T^{'}$.

$$\delta(p_{|\phi^{'}\$T^{'}|+1}, \$) = (\text{"}no\text{"}, \$, -) \tag{18}$$

In this way, we check the input is equal to $\phi^{'}\$T^{'}$ and reject. In case the input is not equal to $\phi^{'}\$T^{'}$, then we need to continue the normal execution of M_{SAT}. For this purpose we add a new step r and the following actions:

$$\text{For each state } p_i \text{ from the } i-th \text{ symbol in } \phi^{'}\$T^{'} \tag{19}$$

$$\text{We add a new action for each symbol } \sigma_j \in A \text{ where } \sigma_j \neq \sigma_i : \tag{20}$$

$$\delta(p_i, \sigma_j) = (r, \sigma_j, -) \tag{21}$$

We related r with the special state $p_{|\phi'\$T'|+1}$ in the following actions too:

$$\delta(p_{|\phi'\$T'|+1}, 0) = (r, 0, -) \tag{22}$$

$$\delta(p_{|\phi'\$T'|+1}, 1) = (r, 1, -) \tag{23}$$

$$\delta(p_{|\phi'\$T'|+1}, \rhd) = (r, \rhd, -) \tag{24}$$

After that, we transit backward through the symbols of the input from the state r to the created state ss with the following actions:

$$\delta(r, 0) = (r, 0, \longleftarrow) \tag{25}$$

$$\delta(r, 1) = (r, 1, \longleftarrow) \tag{26}$$

$$\delta(r, \$) = (r, \$, \longleftarrow) \tag{27}$$

$$\delta(r, \rhd) = (ss, \rhd, -) \tag{28}$$

Finally, we start to simulate the usual computation of the Turing Machine M_{SAT} of Lemma 3.3 from the state ss where ss was the old initial state in M_{SAT} that was renamed.

The final result of this transformation is the Turing Machine $M^{'}_{SAT}$ which was created with a polynomial amount of states and actions inside of M_{SAT} in relation with the size of the input $\phi^{'}\$T^{'}$ and the rejection of $\phi^{'}\$T^{'}$ would be in polynomial time.

Lemma 3.8. *The deterministic Turing machine $M^{'}_{SAT}$ can be inverted into a non-deterministic Turing machine $N^{'}_{SAT}$ and $N^{'}_{SAT}$ accepts in polynomial time.*

This is possible because we can use the same construction that we did in Theorem 3.4 for M_{SAT}. For that reason, the non-deterministic Turing machine $N^{'}_{SAT}$ has the same behavior of N_{SAT} due to the similarity between M_{SAT} and $M^{'}_{SAT}$.

Theorem 3.9. *$UNIQUE\ SAT \in NP$.*

We could compute $UNIQUE\ SAT$ for any Boolean formula ϕ in a non-deterministic in polynomial time in the following way:

- First, we build the non-deterministic Turing machine N_{SAT} from M_{SAT} in constant time.
- Next, we check that $N_{SAT}(\phi) =$ "*yes*" and obtain a satisfying truth assignment T. If $N_{SAT}(\phi) =$ "*no*", then we finish rejecting ϕ for $UNIQUE\ SAT$.
- After that, we change M_{SAT} into another Turing Machine $M^{'}_{SAT}$ which has the same behavior of M_{SAT}, except that $M^{'}_{SAT}(\phi, T) =$ "*no*" where T is the satisfying truth assignment in the previous step.
- Then, we build the non-deterministic Turing machine $N^{'}_{SAT}$ from $M^{'}_{SAT}$.

- Finally, we check that $N'_{SAT}(\phi) =$ "*no*" and finish accepting ϕ for $UNIQUE\ SAT$. If $N'_{SAT}(\phi) =$ "*yes*", then we finish rejecting ϕ for $UNIQUE\ SAT$.

When $N_{SAT}(\phi) =$ "*yes*" and $N'_{SAT}(\phi) =$ "*no*", we could assure there is a unique truth assignment T such that $T \models \phi$, and therefore, $\phi \in UNIQUE\ SAT$. The previous pseudo-algorithm proves that $UNIQUE\ SAT \in NP$, because N_{SAT} and N'_{SAT} are non-deterministic Turing Machines that could compute ϕ in polynomial time. Moreover, the transformation of M_{SAT} into M'_{SAT} could be made in polynomial time in relation with the size of ϕ, because the binary relation R_{SAT} which defines the language SAT in NP is polynomially balanced. Furthermore, the difference between M_{SAT} and M'_{SAT} is only a polynomial amount of states and actions in relation with the size of the input $\phi\$T$, and therefore, the construction of N'_{SAT} is possible in a polynomial time considering the size of ϕ plus the constant time of the first step.

Lemma 3.10. *$NP = coNP$.*

This is a consequence of the Theorem above [1].

3.2. **P = NP.**

Definition 3.11. *Let L_{Nash} be a language that contains Boolean formulas ϕ in $3CNF$ such that $\phi \in L_{Nash} \Leftrightarrow (\phi \in UNSATISFIABLE\ 3SAT \vee \phi \in ONE-IN-THREE\ 3SAT)$ where $\vee$ is the OR Boolean function. We will call this new language as Nash's language.*

The Boolean formulas ϕ that belongs to L_{Nash} are all the Boolean expressions ϕ in $3CNF$ which are in $UNSATISFIABLE\ 3SAT$ or $ONE-IN-THREE\ 3SAT$ languages.

Lemma 3.12. *If $NP = coNP$, then $L_{Nash} \in NPC$.*

If $NP = coNP$, then $UNSATISFIABLE\ 3SAT \in NPC$, this is possible because $UNSATISFIABLE\ 3SAT \in coNPC$ and every language in $coNPC$ would be in NPC when $NP = coNP$ [5]. Indeed, we could easily deduce when ϕ is not in $UNSATISFIABLE\ 3SAT$, then $\phi \in 3SAT$. Moreover, $ONE-IN-THREE\ 3SAT \in NPC$ [5]. Hence, $L_{Nash} \in NPC$ because $L_{Nash} = UNSATISFIABLE\ 3SAT \cup ONE-IN-THREE\ 3SAT$ and the complexity class NPC is close under the union set operation [5].

Definition 3.13. *Let coL_{Nash} be a language that is the complement of L_{Nash}. We could define coL_{Nash} as the Boolean formulas ϕ in $3CNF$ that has a satisfying truth assignment such that each clause in ϕ has at least two true literals.*

This new language will be the key in our proof.

Lemma 3.14. *If $NP = coNP$, then $coL_{Nash} \in coNPC$.*

It is a known result that the complement of a language in NPC is in $coNPC$ [7]. Therefore, this is a direct consequence of Lemma 3.12.

Theorem 3.15. *$coL_{Nash} \in P$.*

We could compute coL_{Nash} for any Boolean formula ϕ in $3CNF$ of m clauses with a deterministic Turing Machine in polynomial time in the following way:

- First, we build for every $i-th$ clause $c_i = (x \vee y \vee z)$ in ϕ, where x, y and z are literals, the following formula $d_i = (x \vee y) \wedge (y \vee z)$.
- Next, we create a Boolean formula ϕ_2 that is equal to $d_1 \wedge d_2 \wedge ... \wedge d_m$ that is the conjunction of all the formulas d_i of the previous step.
- Finally, we check that $\phi_2 \in 2SAT$ and finish accepting ϕ for coL_{Nash}, otherwise we reject ϕ.

We can check if the clause $(x \vee y \vee z)$ has at least two true literals for some truth assignment if and only if the formula $(x \vee y) \wedge (y \vee z)$ has a satisfying truth assignment contained in this truth assignment. In general, if we want to guarantee this property through all the clauses of ϕ, then each formula d_i must have a satisfying truth assignment contained into a single truth assignment for ϕ at the same time. Then, the union of simultaneous truth assignment in each formula d_i could be achieved by joining the d_i formulas with the AND function and creating a new Boolean formula in $2CNF$ that would be ϕ_2. Therefore, a satisfying truth assignment to ϕ_2 is possible if and only if with this truth assignment each clause c_i has at least two true literals that is when $\phi \in coL_{Nash}$.

The creation of ϕ_2 is possible in polynomial time, because we only need to iterate with a polynomial steps through the m clauses of ϕ. The last step could be computed in polynomial time because $2SAT \in P$ [7]. In conclusion, the three steps of this pseudo-algorithm could be computed in polynomial time by a deterministic Turing Machine.

Lemma 3.16. *$P = NP \Leftrightarrow NP = coNP$.*

This is a consequence of the Theorem above, because if $NP = coNP$ and some problem in $coNPC$ is in P, then $P = NP$ and it is a known result if $P = NP$, then $NP = coNP$ [7].

Theorem 3.17. *$P = NP$.*

This is the result of applying the Lemmas 3.10 and 3.16.

4. Conclusions

This proof will have stunning practical consequences, because this leads to efficient methods for solving some of the important problems in NP. After decades of studying these problems no one has been able to find a polynomial time algorithm for any of more than 3000 important known NP-complete problems and this work shows that a feasible solution for all NP-complete problems is possible. There are enormous positive consequences because many problems in operations research are NP-complete, such as some types of integer programming, and the travelling salesman problem. Besides, many other important problems, such as some problems in protein structure prediction, are also NP-complete, and so, this work implies a considerable advance in biology too. In addition, this proof will transform mathematics by allowing a computer to find a formal proof of any theorem which has a proof of a reasonable length, since formal proofs can easily be recognized in polynomial time.

Acknowledgement

I thank my mother Iris Delgado for her support and confidence.

References

1. Andreas Blass and Yuri Gurevich, *On the unique satisfiability problem*, Information and Control **55** (1982), 80–88.
2. Stephen A. Cook, *The complexity of theorem proving procedures*, Proceedings of the 3rd Annual ACM Symposium on the Theory of Computing (STOC'71), ACM Press, 1971, pp. 151–158.
3. Thomas H. Cormen, Charles Eric Leiserson, Ronald L. Rivest, and Clifford Stein, *Introduction to algorithms, second edition*, MIT Press, 2001.
4. Lance Fortnow, *The status of the P versus NP problem*, Communications of the ACM **52** (2009), no. 9, 78–86.
5. M. R. Garey and D. S. Johnson, *Computers and intractability: A guide to the theory of np-completeness (series of books in the mathematical sciences)*, first edition ed., W. H. Freeman, 1979.
6. Harry R. Lewis and Christos H. Papadimitriou, *Elements of the theory of computation (2. ed.)*, Prentice Hall, 1998.
7. Christos H. Papadimitriou, *Computational complexity*, Addison-Wesley, 1994.
8. M. Sipser, *Introduction to the theory of computation*, International Thomson Publishing, 1996.
9. Alan M. Turing, *On computable numbers, with an application to the entscheidungsproblem*, Proceedings of the London Mathematical Society **42** (1936), 230–265.

Datys, Playa, Havana, Cuba
E-mail address: vega.frank@gmail.com