P versus NP

Frank Vega

Noname manuscript No.
(will be inserted by the editor)

P versus NP

Frank Vega

the date of receipt and acceptance should be inserted later

Abstract There are some function problems in $FEXP-complete$, which has a corresponding function problem in FNP, such that each of these function problems in $FEXP-complete$ could be solved by some solution that has the corresponding function problem in FNP for the same input, when the solution for this input in the $FEXP-complete$ function problem exists.

This event is not necessary true when the solution for the input in the $FEXP-complete$ function problem does not exist, that is, the corresponding function problem in FNP could not answer "*no*" when the input does not have a solution in the respective function problem in $FEXP-complete$.

In this way, if $FP = FNP$, then we might find the solutions by a polynomial time algorithm when those solutions exist for the inputs in some $FEXP-complete$ function problem, but this is not possible by the time hierarchy theorem. Therefore, $P \neq NP$.

Keywords Complexity classes · Turing Machines · P · NP · EXP

Mathematics Subject Classification (2000) MSC 68-XX · MSC 68Qxx · MSC 68Q15

1 Introduction

The P versus NP problem is the major unsolved problem in computer science. It was introduced in 1971 by Stephen Cook [1]. Today is considered by many scientists as the most important open problem in this field [3]. A solution to this problem will have a great impact in other fields such as mathematics and biology.

Frank Vega
Datys, 1ra y 4, Playa, Havana, Cuba
Tel.: +53-53457956
E-mail: vega.frank@gmail.com

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 $P \neq NP$, then it would ensure that there are hundreds of problems that have no a feasible solution. This is largely derived from this result that there will be a huge amount of problems that can be checked easily but without some feasible solution [7]. It will remain the best option to use brute force or a heuristic algorithm in many cases.

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 [8]. 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 remain 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 [6].

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 [6].

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 $D \in \{\longleftarrow, \longrightarrow, -\}$ [6]. 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, during any instant in which there is a transition on δ [6]. 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 [5], [4]. 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 [5], [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 [2]. There are three complexity classes that have a close relationship with the previous concepts and are represented as P, EXP and NP. In computational complexity theory, the class P contains the languages that are decided by a deterministic Turing Machine in polynomial time [5]. The complexity class EXP is the set of all decision problems solvable by a deterministic Turing machine in $O(2^{p(n)})$ time, where $p(n)$ is a polynomial function of n. The class NP contains the languages that are decided by a non-deterministic Turing Machines in polynomial time [4]. Problems that are $EXP - complete$ might be thought of as the hardest problems in EXP. We do know that $EXP - complete$ problems are not in P: it has been proven that these problems cannot be solved in polynomial time, by the time hierarchy theorem [6].

On the other hand, a language $L \in NP$ if there is a polynomial-time decidable, 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 $x \in L$ [6]. The function problem associated with L is the following computational problem: given x, find a string y such that $R_L(x, y)$ if such a string exists; if no such string exists, return "*no*" [6]. The class of all function problems associated as above with languages in NP is called FNP [6].

The resulting class from FNP is the class FP which represents all function problems that can be solved in polynomial time [6]. We also could define $FEXP$ and the completeness of this class in a similar way, but the relation would not be polynomial-time decidable. The P versus NP problem is to know whether P is equal to NP or not. This would be equivalent to prove whether FP is equal to FNP or not

3 Results

Lemma 1 *Every language $L_{exp} \in EXP - complete$ has a deterministic Turing Machine M_{exp} that has only one tape and always accepts in the configuration $(\text{“}yes\text{”}, \triangleright, x)$ when $x \in L_{exp}$.*

Every 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 [6]. Therefore, the deterministic Turing Machine that decides L_{exp} could be of one tape. 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 there, it will simulate the original Turing Machine of one tape. When the simulation of the original Turing Machine accepts, it will delete the content in the second tape. Finally, it will set the cursors in the start symbols 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 one-tape Turing Machine M_{exp} complying with the Lemma 1.

Therefore, it will exist a language $L_{exp} \in EXP - complete$ that it is decided by a one-tape deterministic Turing Machine M_{exp}, such that for every element $x \in L_{exp}$ the Turing Machine M_{exp} will accept in the configuration $(\text{“}yes\text{”}, \triangleright, x)$. If M_{exp} will accept $x \in L_{exp}$ in k steps where k would be different for each element, then it will exist a function $f_{L_{exp}}$ which receives the configuration in the $k - \lfloor log_2 \mid x \mid \rfloor$ steps on M_{exp}, that is, the configuration in the $\lfloor log_2 \mid x \mid \rfloor$ steps before M_{exp} accepts x and $f_{L_{exp}}$ returns x for that configuration.

$f_{L_{exp}}$ is from strings to strings because the configurations could be represented as strings. The input x is at most polynomially longer or shorter than the corresponding configuration in the $k - \lfloor log_2 \mid x \mid \rfloor$ steps because from that configuration we cannot add or delete more than $\lfloor log_2 \mid x \mid \rfloor$ symbols until the state of acceptance in $\lfloor log_2 \mid x \mid \rfloor$ steps due to M_{exp} will accept in the configuration $(\text{“}yes\text{”}, \triangleright, x)$. Moreover, $f_{L_{exp}}$ can be computed in polynomial time if we simulate the execution of $M_{exp}(x)$ in the configuration of $k - \lfloor log_2 \mid x \mid \rfloor$ steps until the state of acceptance with the string x using only $\lfloor log_2 \mid x \mid \rfloor$ steps. We will call the configurations that receives $f_{L_{exp}}$ on M_{exp} as $config - 1(x)$ when $f_{L_{exp}}$ produces the string x in the state of acceptance with that configuration. Notice that $f_{L_{exp}}$ could receive a configuration $config - 1(x)$ where x does not belong to L_{exp}.

Definition 1 $ASIA_{L_{exp}}$ will be the function problem defined by the inverse function $f^{-1}_{L_{exp}}$ for some language $L_{exp} \in EXP - complete$ on the deterministic Turing Machine M_{exp} of Lemma 1.

Theorem 1 $ASIA_{L_{exp}} \in FNP$

We could invert the deterministic Turing Machine M_{exp} changing the state of acceptance with the initial state and reversing the transition function of

M_{exp} and in this way, we would create a new non-deterministic Turing Machine N_{exp}. We are going to define the rejection state in N_{exp} in the following way: for every q state in the set of states of N_{exp} and every σ symbol of its alphabet, then $\delta(q, \sigma) = (\text{“}no\text{”}, \sigma, -)$, where δ will be the program of N_{exp}. Indeed, the non-deterministic Turing Machine N_{exp} will simulate the behavior of M_{exp} moving backwards.

In this simulation, we are going to halt just in the first $\lfloor log_2 \mid x \mid \rfloor$ steps, and thus, N_{exp} will execute from the initial configuration $(s, \triangleright, x)$ until some candidate configuration $config - 1(x)$, where we are going to interrupt the possible exponential execution of $N_{exp}(x)$ in only $\lfloor log_2 \mid x \mid \rfloor$ steps. Then, $ASIA_{L_{exp}} \in FNP$ for any language $L_{exp} \in EXP - complete$.

Theorem 2 *For each language $L_{exp} \in EXP - complete$ the function problem of finding the configuration $config - 1(x)$ which belongs to the accepting computation of some input $x \in L_{exp}$ is in $FEXP - complete$. We will denote this problem as $FURONES_{L_{exp}}$.*

$FURONES_{L_{exp}} \in FEXP$ because if we could find the configuration $config - 1(x)$ which belongs to the accepting computation of some input $x \in L_{exp}$ in polynomial time, then we could compute $M_{exp}(x)$ in polynomial time due to we could do it by reaching $config - 1(x)$ in polynomial time, accepting in the following $\lfloor log_2 \mid x \mid \rfloor$ steps and checking if the final configuration is $(\text{“}yes\text{”}, \triangleright, x)$. Hence, as we showed in the Theory section this execution is impossible in polynomial time. Moreover, $FURONES_{L_{exp}} \in FEXP - complete$ because we could decide L_{exp} which is in $EXP - complete$ using $FURONES_{L_{exp}}$ as we showed above.

Theorem 3 *For each language $L_{exp} \in EXP - complete$ the function problem $FURONES_{L_{exp}}$ is solved by some solution that contains $ASIA_{L_{exp}}$ for the same input x when $x \in L_{exp}$.*

The polynomially balanced relation of $ASIA_{L_{exp}}$ has a least one certificate $config - 1(x)$ which belongs to the accepting computation of the input $x \in L_{exp}$. This event is not necessarily true when x does not belong to L_{exp}.

Theorem 4 $P \neq NP$

There are some function problems in $FEXP - complete$, which has a corresponding function problem in FNP, such that each of these function problems in $FEXP - complete$ could be solved by some solution that has the corresponding function problem in FNP for the same input, when the solution for this input in the $FEXP - complete$ function problem exists, which is a direct consequence of the Theorem above and Theorem 1.

This event is not necessary true when the solution for the input in the $FEXP - complete$ function problem does not exist, that is, the corresponding function problem in FNP could not answer “*no*” when the input does not have a solution in the respective function problem in $FEXP - complete$.

In this way, if $FP = FNP$, then we might find the solutions by a polynomial time algorithm when those solutions exist for the inputs in some $FEXP - complete$ function problem, but this is not possible by the time hierarchy theorem. Therefore, $P \neq NP$.

4 Discussion

This result removed the practical computational benefits of a proof that $P = NP$, but would nevertheless represent a very significant advance in computational complexity theory and provide guidance for future research. It shows in a formal way that many currently mathematically problems cannot be solved efficiently, so that the attention of researchers can be focused on partial solutions or solutions to other problems. In addition, it proves that could be safe many of the encryption and authentication methods such as the public-key cryptography. On the other hand, we will not be able to find a formal proof for every theorem which has a proof of a reasonable length in polynomial time by a feasible algorithm.

5 Conclusions

Many computer scientists have believed that $P \neq NP$. A key reason for this belief is that 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. Furthermore, the result $P = NP$ would imply many other startling results that are currently believed to be false. This work shows the belief of almost all computer scientists was a truly supposition.

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

References

1. Cook, S.A.: The complexity of theorem proving procedures. In: Proceedings of the 3rd Annual ACM Symposium on the Theory of Computing (STOC'71), pp. 151–158. ACM Press (1971)
2. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, Second Edition. MIT Press (2001)
3. Fortnow, L.: The status of the P versus NP problem. Communications of the ACM **52**(9), 78–86 (2009)
4. Garey, M.R., Johnson, D.S.: Computers and Intractability: A Guide to the Theory of NP-Completeness (Series of Books in the Mathematical Sciences), first edition edn. W. H. Freeman (1979)
5. Lewis, H.R., Papadimitriou, C.H.: Elements of the theory of computation (2. ed.). Prentice Hall (1998)
6. Papadimitriou, C.H.: Computational complexity. Addison-Wesley (1994)
7. Sipser, M.: Introduction to the Theory of Computation. International Thomson Publishing (1996)
8. Turing, A.M.: On computable numbers, with an application to the entscheidungsproblem. Proceedings of the London Mathematical Society **42**, 230–265 (1936)