E versus NE

Frank Vega

Abstract

The outstanding $P = NP$ incognita would be equivalent to $DTIME(n^k) = NTIME(n^k)$ where $DTIME(n^k) = \bigcup_{j>0} DTIME(n^j)$ and $NTIME(n^k) = \bigcup_{j>0} NTIME(n^j)$. We show the solution of this question in another level that would be the answer of $E = DTIME(2^{O(n)})$ versus $NE = NTIME(2^{O(n)})$ problem.

Keywords: P, NP, E, NE, DTIME, NTIME
2000 MSC: 68-XX, 68Qxx, 68Q15

1. Introduction

The $P$ versus $NP$ problem is a major unsolved problem in computer science. This problem was introduced in 1971 by Stephen Cook [1]. It is considered by many to be the most important open problem in the field [2].

The Turing machine has been an useful concept in theory of computing since it was created by Alan Turing in the last century [3]. Since then, it has appeared new definitions related with this concept such as the deterministic or nondeterministic Turing machine. A deterministic Turing machine has only one next action for each step defined in its program or transition function [4]. A nondeterministic Turing machine can contain more than one action defined for each step of the program where this program is not a function but a relation [5].

Another huge advance was the definition of a complexity class. A language $L$ over an alphabet is any set of strings made up of symbols from that alphabet [6]. A complexity class is a set of problems, which are represented as a language, grouped by measures such as the running time, memory, etc [6].

In computational complexity theory, the class $P$ consists of all those decision problems (defined as languages) that can be solved on a deterministic Turing machine in an amount of time that is polynomial in the size of the input; the class $NP$ consists of all those decision problems whose positive solutions can be verified in polynomial time given the right information, or equivalently, whose solution can be found in polynomial time on a nondeterministic Turing machine [7].

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

Is $P$ equal to $NP$?
In a 2002 poll of 100 researchers, 61 believed the answer to be no, 9 believed the answer is yes, and 22 were unsure; 8 believed the question may be independent of the currently accepted axioms and so impossible to prove or disprove.

The set of languages decided by nondeterministic Turing machines within time $f(n)$ is denoted $\text{NTIME}(f(n))$. In case of the languages would be decided by deterministic Turing machines is denoted $\text{DTIME}(f(n))$. Then, the $P = \text{NP}$ question would be equivalent to $\text{DTIME}(n^k) = \text{NTIME}(n^k)$, where $\text{DTIME}(n^k) = \bigcup_{j>0} \text{DTIME}(n^j)$ and $\text{NTIME}(n^k) = \bigcup_{j>0} \text{NTIME}(n^j)$. In 1983 was published an important theorem which states that $\text{DTIME}(n^k) \neq \text{NTIME}(n^k)$.

We pretend to show in the next few pages the solution of $\text{DTIME}(n^k) = \text{NTIME}(n^k)$ with the answer of $E = \text{DTIME}(2^{O(n)})$ versus $\text{NE} = \text{NTIME}(2^{O(n)})$ problem.

2. Theory

2.1. Turing machines

The argument made by Alan Turing in the twentieth century proves mathematically that for any computer program we can create an equivalent Turing machine. A Turing machine $M$ has a finite set of states and a finite set of symbols 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 $\#$ and the blank symbol $\$. The operations of a Turing machine are based on a transition function $\delta$ 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 $\leftarrow$, right $\rightarrow$ or remained in the same place $\rightarrow$ for each cell. Finally, this process is interrupted if it halts in a final state: the state of acceptance "$\text{yes}$", the rejection "$\text{no}$" or halting $h$.

A Turing machine halts if it reaches a final state. If a Turing machine $M$ accepts or rejects a string $x$, then $M(x) =$ "$\text{yes}$" or "$\text{no}$" 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.

A transition function $\delta$ 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 $\sigma$ of the alphabet, the Turing machine will move to the next state $p$, overwriting the symbol $\sigma$ by $\rho$ and moving the cursor in the direction $D \in \{\leftarrow, \rightarrow, \rightarrow\}$.

2.2. The MINIMUM Problem

Definition 2.1. MINIMUM is the problem of deciding in a given finite set $A$ of $n$ (distinct) integer numbers and an integer $x$ whether $x$ is the minimum number in the set.

How many comparisons are necessary to determine when some integer is the minimum of a set of $n$ elements? We can easily obtain an upper bound of $n$ comparisons: examine each element of the set in turn and keep track of the smallest element seen so far and finally we compare the final result with $x$. Is this the best we can do? Yes, since we can obtain a lower bound of $n - 1$ comparisons for the problem of determining the minimum when the set is represented with an unordered array and one final comparison to verify whether that minimum is equal to $x$ or not. Hence, $n$ comparisons are necessary to determine when an element $x$ is the minimum in $A$ and this algorithm for MINIMUM is optimal with respect to the number of comparisons performed.
3. Results

3.1. The MINIMUM\textsubscript{2} Problem

Let’s see a definition that is the key of this work.

**Definition 3.1.** Given a finite set \( A \) of \( n \) (distinct) deterministic Turing machines, we will call the Integer Turing Collection for the set \( A \), denoted as \( A_{ITC} \), to the collection of positive integers \( m \) where \( m \) is equal to the amount of steps in the halting running of some deterministic Turing machine \( M \in A \) with the empty string as input. The deterministic Turing machines in \( A \) are not taken into account for the definition of \( A_{ITC} \) when they do no halt after \( 2^n \) steps.

The following problem uses the definition above.

**Definition 3.2.** MINIMUM\textsubscript{2} is the problem of deciding in a given finite set \( A \) of \( n \) (distinct) deterministic Turing machines, where each Turing machine has a binary representation with length less than or equal to \( n^2 \), and the integer \( 2^n \) whether \( 2^n \) is the minimum number in \( A_{ITC} \).

**Theorem 3.3.** MINIMUM\textsubscript{2} \( \notin E \).

How many running on the Turing machines in \( A \) with the empty string are necessary to determine when \( 2^n \) is the minimum number in \( A_{ITC} \)? We can easily obtain an upper bound of \( n \) running: run with the empty string each deterministic Turing machine in turn, count the amount of steps until the Turing machines halt or ignore when the amount of steps is greater than the \( 2^n \) bound (we force to halt when the running exceeds the \( 2^n \) steps) and keep track of the smallest count seen so far and finally we compare the final result with \( 2^n \). Is this the best we can do? Yes, since at priori we cannot know the amount of steps that would be obtained in the running of any of these deterministic Turing machines with the empty string, and therefore, we should test all of them with at most \( 2^n \) steps. In addition, when \( 2^n \) is the minimum in \( A_{ITC} \), we should take at least \( 2^n \) steps in the running of every deterministic Turing machine in \( A \). Hence, \( n \times 2^n \) steps are necessary when the integer \( 2^n \) is the minimum in \( A_{ITC} \) and this algorithm in case of acceptance for MINIMUM\textsubscript{2} is optimal.

3.2. Problems in \( NE \)

Let’s define an interesting problem.

**Definition 3.4.** \( IS \pm \text{IN} \pm \text{SET}_2 \) is the problem of deciding in a given finite set \( A \) of \( n \) (distinct) deterministic Turing machines, where each Turing machine has a binary representation with length less than or equal to \( n^2 \), and the integer \( 2^n \) whether \( 2^n \) complies with \( 2^n \in A_{ITC} \).

**Theorem 3.5.** \( IS \pm \text{IN} \pm \text{SET}_2 \in NE \).

Given a finite set \( A \) of \( n \) (distinct) deterministic Turing machines, we create a nondeterministic Turing machine \( N_{IS} \) which simulates all these Turing machines with the following program \( \delta \).

- For each integer \( 1 \leq i \leq n \)
- we add the transition function of the \( i \)-th Turing machine \( M \) to \( \delta \)
• renaming the states with new names to avoid any conflict and,
• changing the renamed initial state \( s_s \) in \( M \)
• with an unique initial state \( s \) in \( \delta \) with this action:

\[
\delta(s, \triangleright) = (s_s, \triangleright, \leftarrow)
\]

The program \( \delta \) could simulate any of these deterministic Turing machines with the empty string as input. \( N_{IIS} \) is the nondeterministic Turing machine represented by the program \( \delta \). We rename the old states in \( M \in A \) with new ones in \( N_{IIS} \) and we only take an unique state \( s \) as the initial state in \( N_{IIS} \) as we show above. We abort the running of \( N_{IIS} \) with the empty string when \( N_{IIS} \) exceeds the amount of \( 2^n \) steps. Indeed, with the program \( \delta \) we could find some possible halting path in any of the deterministic Turing machines in \( A \) when they halt in less than or equal to \( 2^n \) steps with the empty string and verify if the amount of steps in the halting running with the empty string is equal to \( 2^n \). There is always a halting path in \( N_{IIS} \) in \( 2^n \) steps with the empty string if \( 2^n \in A_{ITC} \) otherwise all the possible paths in \( N_{IIS} \) have less or more than \( 2^n \) steps when \( 2^n \notin A \). For that reason, \( IS - IN - SET_2 \) could be decided by a nondeterministic Turing machine in \( O(2^n) \) order, because we only need this order to prove \( 2^n \in A_{ITC} \) by \( N_{IIS} \) using the construction of \( \delta \).

Definition 3.6. \( IS - GREATER_2 \) is the problem of deciding in a given finite set \( A \) of \( n \) (distinct) deterministic Turing machines, where each Turing machine has a binary representation with length less than or equal to \( n^2 \), and the integer \( 2^n \) whether \( 2^n \) complies with \( 2^n > x \) for some \( x \in A_{ITC} \).

Lemma 3.7. \( IS - GREATER_2 \in NE \).

It is easy to see \( IS - GREATER_2 \) is very similar to \( IS - IN - SET_2 \). Indeed, they have the same input. The algorithm will be almost equal to the way of solving \( IS - IN - SET_2 \); the only difference between \( IS - IN - SET_2 \) and \( IS - GREATER_2 \) is that we change the question of \( 2^n \) is equal to by \( 2^n \) is greater than the amount of steps in some possible halting running of \( N_{IIS} \) with the empty string. To sum up, the arguments which prove \( IS - IN - SET_2 \in NE \) would be almost the same that we need to prove \( IS - GREATER_2 \in NE \).

3.3. An Absurd Hypothesis

Definition 3.8. \( IS - LESS - OR - EQUAL_2 \) is the problem of deciding in a given finite set \( A \) of \( n \) (distinct) deterministic Turing machines, where each Turing machine has a binary representation with length less than or equal to \( n^2 \), and the integer \( 2^n \) whether \( 2^n \) complies with \( 2^n \leq x \) for all the elements \( x \in A_{ITC} \). \( IS - LESS - OR - EQUAL_2 \) is the complement of problem \( IS - GREATER_2 \).

This problem would be equivalent to \( MINIMUM_2 \) if \( 2^n \in A_{ITC} \). Let’s state the following hypothesis.

Hypothesis 3.9. \( E = NE \).

There are important consequences if this Hypothesis is true. For example, the following reduction.
Lemma 3.10. If the Hypothesis 3.9 is true, then there is a reduction from $\text{MINIMUM}_2$ to $\text{IS} - \text{LESS} - \text{OR} - \text{EQUAL}_2$ that is in $E$.

Given an unordered array $b$ that represents a finite set $A$ of $n$ (distinct) deterministic Turing machines and the integer $2^n$, we could reduce every instance $<2^n, b> \in \text{MINIMUM}_2$ to $<2^n, b> \in \text{IS} - \text{LESS} - \text{OR} - \text{EQUAL}_2$ if we first verify $2^n \in A_{ITC}$. The reduction from $\text{MINIMUM}_2$ with an instance $<2^n, b>$ would be the rejection of $<2^n, b>$ when $2^n, b \notin IS - IN - SET_2$ otherwise we decide whether $<2^n, b> \in IS - LESS - OR - EQUAL_2$.

Certainly, a nondeterministic Turing machine for $\text{IS} - \text{IN} - \text{SET}_2$ would accept to $<2^n, b>$ when $2^n \in A_{ITC}$ in $2^n$ steps as we prove in Theorem 3.5. Therefore, if the Hypothesis 3.9 is true, then $\text{IS} - \text{IN} - \text{SET}_2 \in E$. In conclusion, if we assume the Hypothesis 3.9 is true, then there is a reduction from $\text{MINIMUM}_2$ to $\text{IS} - \text{LESS} - \text{OR} - \text{EQUAL}_2$ that is in $E$ because we use $\text{IS} - \text{IN} - \text{SET}_2$ which first verifies whether $2^n \in A_{ITC}$.

Theorem 3.11. The Hypothesis 3.9 is false, and therefore, $E \neq NE$.

If the Hypothesis 3.9 is true, then $\text{IS} - \text{GREATER}_2 \in E$. But, if $\text{IS} - \text{GREATER}_2$ is in $E$, then $\text{IS} - \text{LESS} - \text{OR} - \text{EQUAL}_2$ is in $E$ too. When the answer for an instance of $\text{IS} - \text{GREATER}_2$ would be “no” or “yes”, then the answer for the same instance for $\text{IS} - \text{LESS} - \text{OR} - \text{EQUAL}_2$ would be “yes” or “no” respectively. However, if we assume $\text{IS} - \text{LESS} - \text{OR} - \text{EQUAL}_2$ is in $E$, then $\text{MINIMUM}_2$ would be in $E$, because we could use the reduction from $\text{MINIMUM}_2$ to $\text{IS} - \text{LESS} - \text{OR} - \text{EQUAL}_2$ through $\text{IS} - \text{IN} - \text{SET}_2$ that would be in $E$ when the Hypothesis 3.9 is true. But, this is not possible, because $\text{MINIMUM}_2 \notin E$ as we prove in Theorem 3.3 and therefore, the Hypothesis 3.9 is false and $E \neq NE$.

Theorem 3.12. $P \neq NP$.

There is a proved result if $E \neq NE$, then $P \neq NP$ [10]. Therefore, this a direct consequence of Theorem 3.11.

4. Discussion

This proof 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.
Acknowledgement

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

References