A Proof “P≠NP” for P vs. NP Problem by Multiple-Tape Turing-Machine

Yaozhi Jiang

Correspondence: Yaozhi Jiang, Shijiazhuang High-Tech District, Hebei Province, China. E-mail: jiangyaozhi@126.com

Received: April 28, 2020   Accepted: June 19, 2020   Online Published: July 7, 2020
doi:10.5539/jmr.v12n4p1 URL: https://doi.org/10.5539/jmr.v12n4p1

Abstract

P vs. NP problem is very important research direction in computation complexity theory. In this paper, by an engineer’s viewpoint, establishes universal multiple-tape Turing-machine and $k$-homogeneous multiple-tape Turing-machine, and by them we can obtain an unified mathematical model for algorithm-tree, from the unified model for algorithm-tree, we can conclude that computation complexity for serial processing NP problem if under parallel processing sometimes we can obtain $P=NP$ in time-complexity, but that will imply another NP, non-deterministic space-complexity NP, i.e., under serial processing $P≠NP$ in space-complexity, and the result is excluded the case of NP problem that there exists a faster algorithm to replace the brute-force algorithm, and hence we can proof that under parallel processing time-complexity is depended on space-complexity, and vice verse, within P vs. NP problem, this point is just the natural property of P vs. NP problem so that “ P≠NP ”.

Keywords: P vs. NP problem, universal Turing-machine, parallel processing, comprehensive equivalent complexity, complexity-class $NP_1$ and $NP_2$

1. Introduction

P vs. NP problem is an important problem in computation complexity theory. It is from both time-complexity and space-complexity of deterministic/non-deterministic Turing-machine. The complexity is main property of an algorithm, hence the complexity becomes to an important standard in algorithm analysis. The P problem is the problem that can be solved in deterministic/non-deterministic Turing-machine by an algorithm which time-complexity/space-complexity is polynomial, and the NP problem is the problem that can be solved in deterministic/non-deterministic Turing-machine which time-complexity or space-complexity is non-deterministic polynomial, only exponential time-complexity or exponential space-complexity, possibly both exponential time-complexity and exponential space-complexity. P problem is easy, and NP problem is hard. Many NP problems have to waste so long time, or so many processors, that the problem is unsolvable in actual fact. We have many reasons to believe that some of current NP problems will belong to P problem, only we have not yet found exact faster algorithm to them. Author, stands in engineer’s viewpoint not mathematician’s, try to solve P vs. NP problem.

For a given problem, we can construct mathematical model, possibly not only one, and then algorithms, also possibly not only one, and at last we can program it by some computer-languages. To these/this algorithms/algorithm we must do the algorithm analysis to optimize it. Complexity, includes time-complexity and space-complexity, is just main method to estimate algorithm cost. Thus P vs. NP problem is yielded from complexity analysis of algorithm. The hard point of NP is that whether it is natural difficulty, intrinsic difficulty, or artificial imposed difficulty, i.e., whether exponential time-complexity or exponential space-complexity can be transformed into polynomial time-complexity or polynomial space-complexity by algorithm optimization, so that NP can be become into P.

2. Some General Concepts

A given problem $\mathcal{Q}$ is that we want to know something/object, in practice, that never have we known about $\mathcal{Q}$.

Turing-machine is mathematical model of computation, and algorithm is an instruction set of operation logical relationship between input and output.

2.1 Some Definitions

Turing-machine is a system that consists of infinite length string-tape for input and output, and read-write head with ability to read from tape and write on tape, and three-state of “accept”/“reject”/“halt”, and instruction mapping named as transition function.

A string $w$ is a sequence consisted of alphabet of Boolean binary $\{0,1\}$. The bit-number of string $w$, named as $n(w)$, is the string-length. A set $L$ of all string $w$, $L=\{w\}$, can be named as language. Any language $L$ which can be
accepted by a Turing-machine $TM$ will yield an algorithm $A$, with another word $A = \{LTM\}$.

**Definition 2.1.**

Universal multiple-tape Turing-machine $UTM$ with ability to generating any algorithm can be defined as below:

$UTM = \{n\}$ tapes and $n$ read-write-heads;

Input $n$-bit string $w_i \# w_{i+1} \# \ldots \# w_n$ on Tape 1, and scanned by Head 1;

Input $n$-bit string $w_{i+1} \# w_{i+2} \# \ldots \# w_{i+n}$ on Tape 2, and scanned by Head 2;

...;

Input $n$-bit string $w_{i+n} \# w_{i+2n} \# \ldots \# w_{i+3n}$ on Tape $n$, and scanned by Head $n$;

Any one from Head 2 to Head $n$ is dependent on Head 1, if and only if there exists an assignment $H_{i,j} \mapsto H_{i,j}$, when Head 1 scans on $w_{i}, w_{i+1}, \ldots, w_{i+n}$, in which $i$ is Tape $i$ and $j$ is $w_{i,j}$, i.e., $w_{i,j}$ are children of $w_{i,n}$ in tree-graph meaning;

1) Head 1 scans across $w_1$ on Tape 1, and reject if “0” is found in $w_1$, and accept if “1” is found in $w_1$;

2) Head 1 scans across $w_{i+1}, w_{i+2}, \ldots, w_{i+n}$ on Tape 1, and reject if “0” is found in anyone of $w_{i+1}, w_{i+2}, \ldots, w_{i+n}$ on Tape 1, and accept if “1” is found in anyone of $w_{i+1}, w_{i+2}, \ldots, w_{i+n}$ on Tape 1, and these 1s have assignments to certain $H_{i,j}$;

3) Head 2 scans across string $w_{i+1}, w_{i+2}, \ldots, w_{i+n}$ on Tape 2, if Head 1 accept, and if Head 1 accept is dependent on Head 2 under assignment, then Head 2 accept if “1” is found in $w_{i+1}, w_{i+2}, \ldots, w_{i+n}$, and Head 2 reject if “0” is found in $w_{i+1}, w_{i+2}, \ldots, w_{i+n}$;

4) Head 3 scans across string $w_{i+1}, w_{i+2}, \ldots, w_{i+n}$ on Tape 3, if Head 2 accept, and if Head 1 accept is dependent on Head 3 under assignment, then Head 3 accept if “1” is found in $w_{i+1}, w_{i+2}, \ldots, w_{i+n}$, and Head 2 reject if “0” is found in $w_{i+1}, w_{i+2}, \ldots, w_{i+n}$;

$n$) Head $n$ scans across string $w_{i+1}, w_{i+2}, \ldots, w_{i+n}$ on Tape $n$, if Head $(n-1)$ accept, and if Head 1 accept is dependent on Head $n$ under assignment, then Head $n$ accept if “1” is found in $w_{i+1}, w_{i+2}, \ldots, w_{i+n}$, and Head $n$ reject if “0” is found in $w_{i+1}, w_{i+2}, \ldots, w_{i+n}$.

Figure 1. An algorithm-tree example generated from $UTM$

All of algorithms, each of them can be seen as an algorithm-tree, can be represented by the $UTM$. With another word, the $UTM$ is the generalized model for algorithm such that to analyse any algorithm.
Definition 2.2.
If given problem $Q$ is solvable by algorithm $A$, denoted by $A(Q)$. An algorithm $A(Q)$ is valid in syntax, if and only if exists a Turing-machine $UTM$ to produce $A(Q)$. And an algorithm $A(Q)$ is valid in semantics, if and only if exists at least one real interpretation, i.e. one given real problem $Q$ can be solvable by the algorithm $A(Q)$. Obviously an algorithm $A(Q)$ is valid, if and only if it is valid both in syntax and in semantics.

Definition 2.3.
If exists $A_1(Q), A_2(Q) \supseteq A(Q)$, then named that algorithm $A_1(Q)$ is equivalent to algorithm $A_2(Q)$ at problem $Q$ and denoted by $A_1(Q) \equiv A_2(Q)$.

Any $UTM$ for problem $Q$, for certain input and output, is corresponding to an algorithm $A(Q)$, and any algorithm is corresponding to a directed tree-graph named as algorithm-tree and denoted as $T_e(A(Q))$. i.e., $UTM \Rightarrow A(Q) \Rightarrow T_e(A(Q))$.

An example for algorithm-tree as below:

Remarks for algorithm-tree:
1) Algorithm-tree $T_e(A(Q))$ is a directed tree graph without directed-cycle. In which the vertex $V_{r,1}$ is root-vertex, such vertex possibly not only one and root-vertexes have no incoming edge. The vertexes $V_{l,i}$ are leaf-vertex, the leaf-vertexes have no outgoing edge. The first label of suffix of vertex is its order-number of vertex-hierarchy, and the second label of suffix of vertex is its order-number of the vertex-branch.

2) A branch $B_{r,i \rightarrow l,k}$ is a path from root-vertex $V_{r,i}$ to leaf-vertex $V_{l,k}$ without reverse-directed edge. The path length $L(B_{r,i \rightarrow l,k})$ is the number of edge contained in the $B_{r,i \rightarrow l,k}$. The maximal path length $\max L(B_{r,i \rightarrow l,k})$ over algorithm-tree $T_e(A(Q))$ is depth of the algorithm-tree $T_e(A(Q))$. The number of branch, number of all path from root-vertex to leaf-vertex, $N(B)$ of algorithm-tree $T_e(A(Q))$ is breadth of algorithm-tree $T_e(A(Q))$. The degree of a vertex $D(V)$ is its number of outgoing edge.

3) Path $B_i$ and path $B_j$, both from root-vertex to leaf-vertex, are different each other, if and only if there exists at least one different edge.

4) There exists multiple-root-and-multiple-leaf algorithm-tree, named as generalized algorithm-tree.

Algorithm-tree can be divided into four types:
1) single-root-and-single-leaf,
2) single-root-and-multiple-leaf,
3) multiple-root-and-single-leaf, and
4) multiple-root-and-multiple-leaf.

Some examples for the four types algorithm-tree above are below.

![Figure 2. An example for algorithm-tree](image-url)
Hierarchy-branch structure, briefly \( HBS \), is produced by problem \( Q \) itself or by algorithm \( A(Q) \). Many problems acted on graphs have natural \( HBS \) in intuition. And all problems which contain recursion structure also have natural \( HBS \).

Figure 3. Algorithm-tree for single-root-and-single/multiple-leaf

Figure 4. Algorithm-tree for multiple-root-and-single-leaf

Figure 5. Algorithm-tree for multiple-root-and-multiple-leaf
Deterministic Turing-machine, briefly $DTM$ and can be represented as single-root-and-single-leaf algorithm-tree, is a special case of non-deterministic Turing-machine, briefly $NTM$ , which can be represented as single-root-and multiple-leaf algorithm-tree, i.e., $DTM \subseteq NTM$.

Lemma 2.4.
1) For single-root-and-single-leaf algorithm-tree, its breadth $N(B) = 1$;
2) For single-root-and-multiple-leaf/multiple-root-and-single-leaf algorithm-tree, breadth $N(B)$ of algorithm-tree $T_s(A(Q))$ is equal to the product of $n(R)$ times $n(L)$, i.e., $N(B) = n(R) \times n(L)$, in which $n(R)$ is root-vertex number and $n(L)$ is leaf-vertex number of the algorithm-tree $T_s(A(Q))$;
3) For multiple-root-and-multiple-leaf algorithm-tree, if root-vertex number is $s$ and leaf-vertex number is $m$, then its breadth $$N(B) = \sum_{i=1}^{s} \sum_{j=1}^{m} N\left(B_{i,j-ls}\right)$$

Proof.
The clause 1) and clause 2) is obvious, the proofs are unnecessary. Only proof for clause 3) as below.
In induction. For root-vertex $V_{e1}$, branch number from root-vertex $V_{e1}$ to all leaf-vertex is
$$N\left(B_{i,j-ls}\right) = \sum_{i=1}^{s} N\left(B_{i,j-ls}\right)$$
For root-vertex $V_{e2}$, branch number from root-vertex $V_{e2}$ to all leaf-vertex is
$$N\left(B_{i,j-ls}\right) = \sum_{i=1}^{s} N\left(B_{i,j-ls}\right)$$
Thus the total branch number in algorithm-tree $T_s(A(Q))$ is
$$N(B) = \sum_{i=1}^{s} \sum_{j=1}^{m} N\left(B_{i,j-ls}\right)$$

Property 2.5.
Depth $N(D)$ of algorithm-tree $T_s(A(Q))$ is equal to the maximal hierarchy number of algorithm-tree $T_s(A(Q))$ minus one, i.e., $N(D) = L(B) = H(B) - 1$

Definition 2.6.
An algorithm $A(Q)$ is named as $k$-homogeneous algorithm, if and only if its degree of vertex in any hierarchy and any branch is an integer constant $k$. An algorithm $A(Q)$ is named as $H_i$-homogeneous algorithm, if and only if its degree of vertex in same hierarchy is an integer constant $i$, $0 \leq k \leq n$, $0 \leq i \leq n$.

Definition 2.7.
For time-complexity $T(A(Q)) = f(n)$, i.e., running time cost of $A(Q)$, if $A(Q) = \{A_1(Q), A_2(Q), ..., A_m(Q)\}$, $T(A(Q)) = f_1(n)$, $T(A_i(Q)) = f_i(n)$, and $f_i(n) \leq f_j(n)$, $i, j \leq l$; then named as $A_i(Q) \leq A_j(Q)$.

Definition 2.8.
For space-complexity, only defined by that it is number of processor, $S(A(Q)) = g(n)$, i.e., running processor cost of $A(Q)$, if $A(Q) = \{A_1(Q), A_2(Q), ..., A_m(Q)\}$, $S(A(Q)) = g_1(n)$, $S(A_i(Q)) = g_i(n)$, and $g_i(n) \leq g_j(n)$, $i, j \leq l$; then named as $A_i(Q) \leq A_j(Q)$.

2.2 Optimization for Algorithm $A(Q)$

Definition 2.9.
If given problem $Q$ is solvable and its valid algorithm-set $A(Q) = \{A_1(Q), A_2(Q), ..., A_m(Q), A_1(Q)\}$, and $A_1(Q), A_2(Q), ..., A_m(Q)$ be $A_i(Q)$ in which terms in descending-sort by time-complexity or space-complexity, and $A_1(Q)$ is the optimal algorithm from optimization. If algorithm-set $A(Q)$ can be divided into two proper subsets $A'(Q)$, subset in polynomial time/space solvable, and $A''(Q)$, subset in non-deterministic polynomial time/space solvable, such that $A(Q) = A'(Q) \cup A''(Q)$ and $A'(Q) \subseteq P$, $A''(Q) \subseteq NP$. And if $A'(Q) \neq \emptyset$, then algorithm-set $A(Q)$ can be named as $P-A(Q)$, and if $A'(Q) = \emptyset$, then algorithm-set $A(Q)$ can be named as $NP-A(Q)$, especially if $A'(Q) \cup A''(Q) = \emptyset$, then the problem $Q$ can be named as unsolvable.

$P-A(Q)$ is easy-to-solvable, and $NP-A(Q)$ is hard-to-solvable.
Lemma 2.10.

Under \( NP = A(Q) \), \( (P \neq NP) \Leftrightarrow (NP = A(Q) \neq \emptyset) \).

Proof.

There exists three cases in \( A(Q) \):

1) \( A(Q) \subset P \Rightarrow P = A(Q) \),

2) \( A(Q) = A'(Q) \cup A''(Q) \) , and \( A'(Q) \subset P \), \( A''(Q) \subset NP \) , these will imply \( A(Q) \subset P \) , hence \( A(Q) = A'(Q) \cup A''(Q) \Rightarrow P - A(Q) \),

3) \( A(Q) \subset NP \Rightarrow NP = A(Q) \).

Because \( NP = A(Q) \Leftrightarrow P - A(Q) \neq \emptyset \), therefore it is impossible to optimize \( A(Q) \) from \( NP = A(Q) \) to \( P - A(Q) \), hence implies the class \( NP \) exists.

2.3 Definitions for Non-Deterministic Multiple-Tape Turing-Machine

Depth-first read means that we read the algorithm-tree each branch by each branch, i.e., each path by each path, and the reading must go upon every branch in algorithm-tree one time and only one time.

Breadth-first read means that we read the algorithm-tree each hierarchy by each hierarchy, and the reading must go upon every hierarchy in algorithm-tree one time and only one time.

Definition 2.11.

Breadth-first read \( NTM_k \) which generating \( k \) - homogeneous algorithm can be defined as below:

\[ NTM_k = "n\ Tapes and \ n\ Heads; \]

Input \( n \)-bit string \( w_1 \) on Tape 1, and scanned by Head 1;

Input \( n \)-bit string \( w_2 \) on Tape 2, and scanned by Head 2

... 

Input \( n \)-bit string \( w_n \) on Tape \( n \), and scanned by Head \( n \); 

Denote number of “1” in string \( w_i \) by \( m(w_i) \), then \( m(w_1) = m(w_2) = \ldots = m(w_n) = k \)

1) Head 1 scans across \( w_1 \) on Tape 1, and reject if “0” is found in \( w_1 \), and accept if “1” is found in \( w_1 \);

2) If Head 1 accept , Head 2 scans across \( w_2 \) on Tape 2, and reject if “0” is found in \( w_2 \), and accept if “1” is found in \( w_2 \);

3) If Head 2 accept , Head 3 scans across \( w_3 \) on Tape 3, and reject if “0” is found in \( w_3 \), and accept if “1” is found in \( w_3 \);

... 

\( n \) ) If Head \( (n-1) \) accept , Head \( n \) scans across \( w_n \) on Tape \( n \), and reject if “0” is found in \( w_n \), and accept if “1” is found in \( w_n \);"
Figure 6. An example of 2-homogeneous algorithm-tree generated from $NTM_i$

2.4 Some Properties of Brute-Force Algorithm

The time-complexity calculation is corresponding to $k$-homogeneous/$H_i$-homogeneous algorithm-tree $T_s(A(Q))$, there exists continued product of binomial below

$$T(n, i_j)=T\left(\prod_{j=1}^{k}(n-i_j)\right)-O(n^s), \quad n-j\geq 0,$$

where $n, i_j$ are integers.

If $(n-i_1)=1$, i.e., $i_1=n-1$ and $i_2=\cdots=i_s=n$ then $T(1)-O(1)=O(1)$, or if $i_1=\cdots=i_s=n-1$, then $T(1^s)-O(n)$;

If $(n-i_1)=2$, i.e. $i_1=\cdots=i_s=n-2$, then $T(2^s)-O(2^n)$;

...  

If $(n-i_1)=n$, i.e., $i_1=\cdots=i_s=0$, then $T(n^s)-O(n^s)$;

And if $(n-i_1)=(n-i_2)=(n-1), \cdots, (n-i_s)=1$, then $T\left(\prod_{j=1}^{k}(n-i_j)\right)-O(n!)$.

For the binary search, $T\left(\frac{n}{2}\right)-O(\log n), \quad T\left(\frac{n^s}{2}\right)-O(n\log n)$, in which $s$ is step number of binary search.

For the $T(n)-O(n^s)$, the function $n^s$ is power-exponent function, and its base number indicates branch-number of algorithm $A(Q)$, the exponent indicates hierarchy-number of algorithm $A(Q)$.

3. P vs. NP Problem

If we name polynomial time-complexity as polynomial bound, and exponential time-complexity as exponential bound, then the P vs. NP problem is that whether there exists an algorithm bridge from exponential bound to polynomial bound such that $P=NP$. For some given problems, as if under brute-force search it is exponential time-complexity, there indeed exists such algorithm that the problem is solvable in polynomial time, such as binary search for the certain problem that selecting someone within random integers. But for other exponential time problems, we have no ability to solve them in polynomial time, perhaps only we do not yet found the exact faster algorithm. With another word, all of P problems are solvable in polynomial time, and some of NP problems are also solvable in polynomial time, but other of NP problems seems that they are solvable only in exponential time.
Because of the incompleteness of algorithm-set, i.e., we do not know whether human has found all valid algorithms, so we have the difficult point hard to proof in computation theory called as P vs. NP problem.

**Definition 3.1.**

Comprehensive equivalent complexity \( C_{eq} = (T(A(Q)), S(A(Q))) \), if both \( T(A(Q)) \subseteq \text{POLY} \), i.e., \( T(A(Q)) \) is polynomial time solvable (the same below), and \( S(A(Q)) \subseteq \text{POLY} \), i.e., \( S(A(Q)) \) is polynomial space solvable (the same below), then \( C_{eq} \subseteq \text{POLY} \), named as \( C_{eq} - \text{POLY} \); if at least \( T(A(Q)) \subseteq \text{EXP} \), i.e., \( T(A(Q)) \) is exponential solvable (the same below), or \( S(A(Q)) \subseteq \text{EXP} \), i.e., \( S(A(Q)) \) is exponential solvable (the same below), then \( C_{eq} \subseteq \text{EXP} \), named as \( C_{eq} - \text{EXP} \). With another word, \( C_{eq} - \text{EXP} \) means that at least one of \( T(A(Q)) \subseteq \text{EXP} \) and \( S(A(Q)) \subseteq \text{EXP} \) appears.

Obviously, under parallel processing, if denote processor number by \( s \), then the case \( s = 1 \) is serial processing, and the case \( s > 1 \) is parallel processing.

**Definition 3.2.**

For any \( s \geq 1 \), i.e., under both serial processing and parallel processing, if \( C_{eq} \subseteq \text{POLY} \), named as the first-type NP problem, denoted by \( \mathbb{NP}_1 \); if \( C_{eq} \subseteq \text{EXP} \), named as the second-type NP problem, denoted by \( \mathbb{NP}_2 \).

Obviously \( \mathbb{NP} = \mathbb{NP}_1 \cup \mathbb{NP}_2 \).

So, we obtain that under \( s > 1 \) and \( A(Q) \subseteq \mathbb{NP} \), there exists a mapping

\[
g_{s,n}: (T(A(Q)) \subseteq \text{EXP}, S(A(Q)) \subseteq \text{POLY}) \rightarrow (T(A(Q)) \subseteq \text{POLY}, S(A(Q)) \subseteq \text{POLY})
\]

That is

\[
g_{s,n}: \mathbb{NP}_1 - A(Q) \rightarrow \mathbb{P} - A(Q)
\]

**Conjecture.**

\( \mathbb{NP} \) is yielded from hierarchy-branch structure, natural \( \text{HBS} \), inside problem \( Q \). And \( \mathbb{NP}_2 \) is yielded from the problem \( Q \) which \( \text{HBS} \) is not coupled with string-length \( n(w) \), \( \mathbb{NP}_2 \) is yielded from the problem \( Q \) which \( \text{HBS} \) is coupled with string-length \( n(w) \).

With another word, for any \( s \geq 1 \), \( \mathbb{NP}_1 \) is easy-to-solvable both in polynomial time and polynomial space; \( \mathbb{NP}_2 \) is hard-to-solvable if one of time-complexity and space-complexity is exponential, or both exponential in time-complexity and space-complexity.

![Figure 7. Venn-diagram for \( P \subseteq \mathbb{NP}_1 \subseteq \mathbb{NP}_2 \)](image)
1) If \( s=1 \), the sum needs \( n'-1 \) steps, and \( T\left(A(Q)\right) - O(n') \), and only brute-force search can be used.

2) If we properly divide the continued-sum into \( m \) subsets and each subset contains \( i \) elements, then the sum in each subset needs \( (i-1) \) steps, and the sum in all subsets needs \( (m-1) \) steps, then the total steps is \( m(i-1)+(m-1)=n'-1-T\left(A'\right)=O(n') \);

3) If \( s=m>1 \), then time-complexity \( T\left(A(Q)\right) - \Omega(i) \), \( S\left(A(Q)\right) - O\left(s=m\right) \);

4) The continued-sum exists only one solution;

5) This Example 1. belongs to \( NP \subset P \).

Example 2. An example for \( NP \)

Consider a simple secret-code, key for password coding system, its code-element number is \( n \), and code-length is \( i \times n \), and we can construct \( i \times n \)-bit key via at first permutation of \( n^n \) code-element, so we can obtain \( n^n \) essential-code, and then each time select \( i \) essential-code to bind by second permutation of \( i \) essential-code, thus the total number of \( i \times n \)-bit key is \( P^r_i=n^r\left(n^n-1\right)\left(n^n-2\right)\ldots\left(n^n-i+1\right) \) and \( P'_e=n^n! \).

The detail is below.

\[
\begin{align*}
a_{11}, & a_{12}, \ldots, a_{1n} \\
a_{21}, & a_{22}, \ldots, a_{2n} \\
\vdots & \\
a_{s1}, & a_{s2}, \ldots, a_{sn}
\end{align*}
\]

We select one element from each row to produce \( n^n \) essential-code in column in which every essential-code is different with others, for \( n^n \)-row code-element is not same as others. Then select \( i \) essential-code from \( n^n \) essential-code to bind, i.e., chain these \( i \) essential-code into \( i \times n \)-bit key by second permutation of \( i \) essential-code as below,

Suppose essential-code \( A=\{A_1, A_2, \ldots, A_i\} \), chain these \( i \) essential-code by the method of second permutation below

\[
P'_i(A_1, A_2, \ldots, A_i)=1!
\]

Thus the total number of \( i \times n \)-bit key is \( P^r_i=n^r\left(n^n-1\right)\left(n^n-2\right)\ldots\left(n^n-i+1\right) \) and \( P'_e=n^n! \).

1) To break the key, if \( s=1 \) only brute-force search can be used, and needs computation of \( \left(i!\right)P^r_i=(i)!\left(n^n\right) \) steps,

2) if \( s=n^{-1} \), then \( T\left(A\left(n^n\right)\right) - \Omega\left(n^s\right) \), \( S\left(n^n\right) - O\left(s=n^{-1}\right) \);

3) This Example 2. belongs to \( NP \).

That existence of the two examples implies that \( NP-A\left(Q\right) \neq \emptyset \), under \( s=1 \). And existence of the Example 2. implies that \( C_{NP} \subset EXP \Rightarrow NP-A\left(Q\right) \neq \emptyset \), under \( s>1 \).

The Lemma 3.3. shows that \( NP-A\left(Q\right) \) is valid both in syntax and in semantics, i.e., it is both be generated by \( NTM \) and exists at least some real interpretations.

Theorem 3.4. (Seesaw Theorem)

1) For \( A\left(n^n\right) \subset NP \), \( s>1 \), and \( s=n^{-1} \), then \( T\left(A\left(n^n\right)\right) - O\left(n^n\right) \), \( S\left(n^n\right) - O\left(n^n\right) \).

In fact, we have stronger result, the clause 1) is a corollary from clause 2) below.

2) For \( A\left(n^n\right) \subset NP \), \( s>1 \), and \( s=n^n, 0 \leq i \leq n^n \), i.e., \( n=i+1 \), then \( T\left(A\left(n^n\right)\right) - O\left(i+n^n\right) \), \( S\left(n^n\right) - O\left(n^n\right) \).

Proof.

In induction. In hierarchy-first-read to algorithm-tree \( T_s\left(A\left(Q\right)\right) \).

For the first hierarchy vertex must need \( 1 \) processor, at this time \( i=0 \), then \( T\left(A\left(n^n\right)\right) - O\left(i+n^n=i+n^n\right) \), \( S\left(n^n\right) - O\left(n^n=1\right) \);

For the second hierarchy vertex must need \( n-1 \) processors, because of the \( 1 \) processor used in the first hierarchy have finished its work in first hierarchy and now can be used in the second hierarchy vertex, at this time \( i=1 \), then \( T\left(A\left(n^n\right)\right) - O\left(i+n^n=1+n^n\right) \), \( S\left(n^n\right) - O\left(n^n\right) \);

For the third hierarchy vertex must need \( n^n-(n-1) \) processors, because of the \( n-1 \) processors used in second hierarchy have finished their work in the second hierarchy and now can be used in the third hierarchy vertex, at this
Thus, for the $n^s$ hierarchy vertex, at this time $i = n-1$, must need

$$p = n^{i-1} - \left(n^{i-2} - \left(n^{i-3} - \cdots - \left(n - (n-1)\right)\right)\right)$$

$$= n^{i-1} - \left(n^{i-2} - \cdots - \left(n - (n-1)\right)\right) - O(n^{i-1})$$

processors, therefore $S(A(n^s)) = p - O(n^{i-1})$, depth of algorithm-tree $T_n(A(n^s))$ is $D(N)$, $T(A(n^s)) = (D(N)-1+n) - O(i+n^s = 2n-1) = O(n)$. The Seesaw Theorem above means that in algorithm for $s > 1$, if we hope to save time, then will cost more space, and if we hope to save space, then will cost more time.

If we see parallel processing as a special algorithm, named as $NP - A_{s_i}(Q)$, that contains $s = n^{i-1}$ parallel processors, this Theorem 3.4., shows us that if for $NP_2$ there exists a mapping $f_{n_1}(T(A(n^s))) \subseteq EXP, S(A(n^s)) \subseteq POLY \rightarrow T(A(n^s)) \subseteq POLY, S(A(n^s)) \subseteq EXP$

That is $f_{n_1}: NP_2 - A(Q) \rightarrow NP_2 - A(Q)$

This means that parallel processing is no ability to become $NP_2 - A(Q)$ into $P - A(Q)$.

**Theorem 3.5.**

For $A_{s_i}(Q)\subseteq NP$, algorithm $NP_2 - A_{s_i}(Q)$ is the fastest algorithm in time-complexity, but the worst cost algorithm in space-complexity. **Proof.**

Theorem 3.4. has show that there exists a mapping $f_{n_1}$ such that $f_{n_1}(T(A(n^s))) \subseteq EXP, S(A(n^s)) \subseteq POLY \rightarrow T(A(n^s)) \subseteq POLY, S(A(n^s)) \subseteq EXP$

Now we will proof only that algorithm $NP_2 - A_{s_i}(Q)$ is the fastest algorithm in time-complexity. Recall the proof process in Theorem 3.4., each processor works immediately at first time therefore no waste time can be found, and for algorithm $A(n^s)$ only brute-force search can be used, $A(n^s) \subseteq NP$, therefore algorithm $NP_2 - A_{s_i}(Q)$ is the fastest algorithm for time-complexity $T(A(n^s))$. **Corollary 3.6.**

For $s > 1$, $s = 2^r$, and $A(2^r) \subseteq NP$, then $T(2^r) = O(D(N) - 1 + 2^{r-1})$, $S(2^r) = O(2^r)$. **Proof.**

In induction. In hierarchy-first-read to algorithm-tree $T_n(A(2^r))$.

For the first hierarchy vertex must need 1 processor, at this time $i = 0$, then $T(A(2^r)) = O(i + 2^{r-1} = 2^r)$, $S(A(2^r)) = O(2^r = 1)$;

For the second hierarchy vertex must need 2 processors, because of the 1 processor used in the first hierarchy have finished its work in first hierarchy and now can be used in the second hierarchy vertex, at this time $i = 1$, then $T(A(2^r)) = O(i + 2^{r-1} = 1 + 2^r)$, $S(A(2^r)) = O(2^r = 2)$;

For the third hierarchy vertex must need 2$^2$ processors, because of the 2 processors used in second hierarchy have finished their work in the second hierarchy and now can be used in the third hierarchy vertex, at this time $i = 2$, then $T(A(2^r)) = O(i + 2^{r-1} = 1 + 2^r)$, $S(A(2^r)) = O(2^r = 4)$;

For the fourth hierarchy vertex must need 2$^3$ processors, because of the 2$^2$ processors used in second hierarchy have finished their work in third hierarchy and now can be used in the fourth hierarchy vertex, at this time $i = 3$, then $T(A(2^r)) = O(i + 2^{r-1} = 1 + 2^r)$, $S(A(2^r)) = O(2^r = 8)$;

... Thus, for the $n^s$ hierarchy vertex, at this time $i = n-1$, must need
\[ p = 2^{n+1} - \left( 2^{n} - \left( 2^{n-1} - \left( 2^{n-2} - \ldots - (2^2 - (2 - 1)) \right) \right) \right) \]

\[ = 2^n - \left( 2^{n-1} - \left( 2^{n-2} - \ldots - (2^2 - (2 - 1)) \right) \right) - O(2^{n-1}) \]

processors, therefore \( S(A(2^n)) = p - O(2^{n-1} = 2^{n+1}) \), depth of algorithm-tree \( T_s(A(2^n)) \) is \( D(N) \),
\[ T(A(2^n)) = (D(N) - 1 + 2^n) - O(n - 1 + 2 = n + 1) = D(n) . \]

Corollary 3.7.

For \( s > 1 \), and \( A_q(n) \in NP \), obvious \( A_q(n) \) is \( H_s \)-homogeneous algorithm, then \( T(n) - O \left( i + (n - i - 1) \right) \),
\[ S(n) - O \left( (n-1)! \right) . \]

Proof.

In induction. In hierarchy-first read to algorithm-tree \( T_s(A(n)) \).

For the first hierarchy vertex must need \( 1 \) processor, at this time \( i = 0 \), then \( T(A(n)) - O(n) \), \( S(A(n)) - O(0! = 1) \);

For the second hierarchy vertex must need \( n \) processors, because of the \( 1 \) processor used in the first hierarchy have
finished its work in first hierarchy and now can be used in the second hierarchy vertex, at this time \( i = 1 \), then \( T(A(n)) - O(i + (n - i - 1)) \), \( S(A(n)) - O((n - 1)! = 1) \);

For the third hierarchy vertex must need \( n(n-1) \) processors, because of the \( n \) processors used in second hierarchy
have finished their work in second hierarchy and now can be used in the third hierarchy vertex, at this time \( i = 2 \), then \( T(A(n)) - O(i + n(n-1) = 2 + n(n-1)) \), \( S(A(n)) - O(n(n-1)) \);

For the fourth hierarchy vertex must need \( n(n-1)(n-2) \) processors, because of the \( n(n-1) \) processors used in the third hierarchy
have finished their work in third hierarchy and now can be used in the fourth hierarchy vertex, at this time \( i = 3 \), then
\[ T(A(n)) - O(i + n(n-1)(n-2) = 3 + n(n-1)(n-2)) \), \( S(A(n)) - O(n(n-1) - n - 1 = n(n-1)(n-3)) ; \]

Thus, for the \( n^s \) hierarchy vertex, at this time \( i = n-1 \), must need
\[ p = n(n-1)(n-2) \ldots (n-i-n(n-1)(n-2) \ldots (n-i-1) \]
\[ = (n-i)! \]

processors, therefore \( S(A(n)) = p - O((i - i - 2)! \) \), depth of algorithm-tree \( T_s(A(n)) \) is \( D(N) \),
\[ T(A(n)) = (D(N) - 1 + 1) - O(n) . \]

Theorem 3.8.

If \( f(n) \) is a function and \( n = g(m) \), for \( s > 1 \) and \( A_q(f(n)) \in NP \), then \( A_q(f(n)) \in NP \),
\[ O(n) = O \left( f(g(m)) \right) . \]

Proof.

Reduction to absurdity.

For function \( n = g(m) \), just only bit-number transformation function, do not change any properties of algorithm \( A(Q) \)
but bit-number function, of cause there exists a reverse-function \( m = f^{-1}(n) \) such that \( m \) satisfies definition of bit-number.

If \( m \) does not satisfy definition of bit-number, this is contradiction with that bit-number of string \( w \), i.e., string-length \( n(w) \) , is only one restriction, i.e., it is finite.

Therefore Theorem 3.8. has been proofed.

Corollary 3.9.

For \( s > 1 \) and \( A_q(Q) \in NP \), If \( \left( n_i \right) = n_i^{s_i} \), and \( \left( n_i \right) = n_i \), in which both \( i, j > 0 \) and both \( i, j \) integer constant, then
\[ j = \frac{n_i \log n_i}{\log i + \log n_i} , \]
and \( \left( \frac{n_i \log n_i}{\log i + \log n_i} \right) = n_i^{s_i} \), \( i = n_i^{s_i} \).

Proof.

For \( A_q(Q) \in NP \), there does not exist natural hierarchy-branch structure, i.e., natural \( HBS \) within problem \( Q \), therefore
there does not exist sequential logic structure within problem \( Q \), so we can treat input \( n_i \) freely. If we make \( s = i \), then
From \((m_i)^i = n_i\); we obtain below

\[
j(\log i + \log n_i) = n_i \log n_i, \quad \text{then} \quad j = \frac{n_i \log n_i}{\log i + \log n_i},
\]

And from \(m_i = n_i\), and \((m_i)^i = n_i\); then \(i = n_i^{\frac{1}{n_i^i}}\).

**4. Conclusion**

Because of P vs. NP problem is a hard problem in algorithm analysis, therefore it is an interesting problem.

Author divides the NP problem into two types, \(NP_1\) and \(NP_2\). \(NP_1\) is the class of both time-complexity and space-complexity are polynomial, \(NP_2\) is the class of at least one of time-complexity and space-complexity is exponential. And if we see parallel processing and serial processing as an unified processing, i.e., processor number is \(s\), and obvious that under if \(s = 1\) then serial processing, and if \(s > 1\) then parallel processing. And proofs that there exists mapping \(g_{s,1}\) such that \(g_{s,1}: NP_1 \rightarrow P\), and mapping \(f_{s,1}\) such that \(f_{s,1}: NP_2 \rightarrow NP\), therefore under \(s > 1\), \(NP_2 \neq \emptyset \Rightarrow P \neq NP\), obviously under \(s = 1\), \(NP \cup NP_2 = NP \Rightarrow P = NP\).

**References**


**Copyrights**

Copyright for this article is retained by the author(s), with first publication rights granted to the journal.

This is an open-access article distributed under the terms and conditions of the Creative Commons Attribution license (http://creativecommons.org/licenses/by/4.0/).