Journal of Mathematics Research; Vol. 12, No. 4; August 2020
ISSN 1916-9795 E-ISSN 1916-9809
Published by Canadian Center of Science and Education

A Proof "$P \neq NP$" 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 author, 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 $P \neq NP$ 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 "$P \neq NP$".

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 Q is that we want to know something/object, in practice, that never have we known about 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 $A=\{L,TM\}$.

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 $w_1\#w_2\#\cdots\#w_n$ on Tape 1, and scanned by Head 1;

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

…

Input $n-$bit string $w_{n,1}\#w_{n,2}\#\cdots\#w_{n,n}$ 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 $H_{1,m}\mapsto H_{i,j}$ when Head 1 scans on $w_2,w_3,\cdots,w_n$, in which i is Tape i and j is $w_{i,j}$, i.e., $w_{i,j}$ are children of $w_{1,m}$ 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 $w_2,w_3,\cdots,w_n$ on Tape 1, and reject if "0" is found in anyone of $w_2,w_3,\cdots,w_n$ on Tape 1, and accept if "1" is found in anyone of $w_2,w_3,\cdots,w_n$ on Tape 1, and these 1s have assignments to certain $H_{i,j}$;

3) Head 2 scans across string $w_{2,1},w_{2,2},\cdots,w_{2,n}$ 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 $w_{2,1},w_{2,2},\cdots,w_{2,n}$, and Head 2 reject if "0" is found in $w_{2,1},w_{2,2},\cdots,w_{2,n}$;

4) Head 3 scans across string $w_{3,1},w_{3,2},\cdots,w_{3,n}$ 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 $w_{3,1},w_{3,2},\cdots,w_{3,n}$, and Head 2 reject if "0" is found in $w_{3,1},w_{3,2},\cdots,w_{3,n}$;

n) Head n scans across string $w_{n,1},w_{n,2},\cdots,w_{n,n}$ 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 $w_{n,1},w_{n,2},\cdots,w_{n,n}$, and Head n reject if "0" is found in $w_{n,1},w_{n,2},\cdots,w_{n,n}$; "

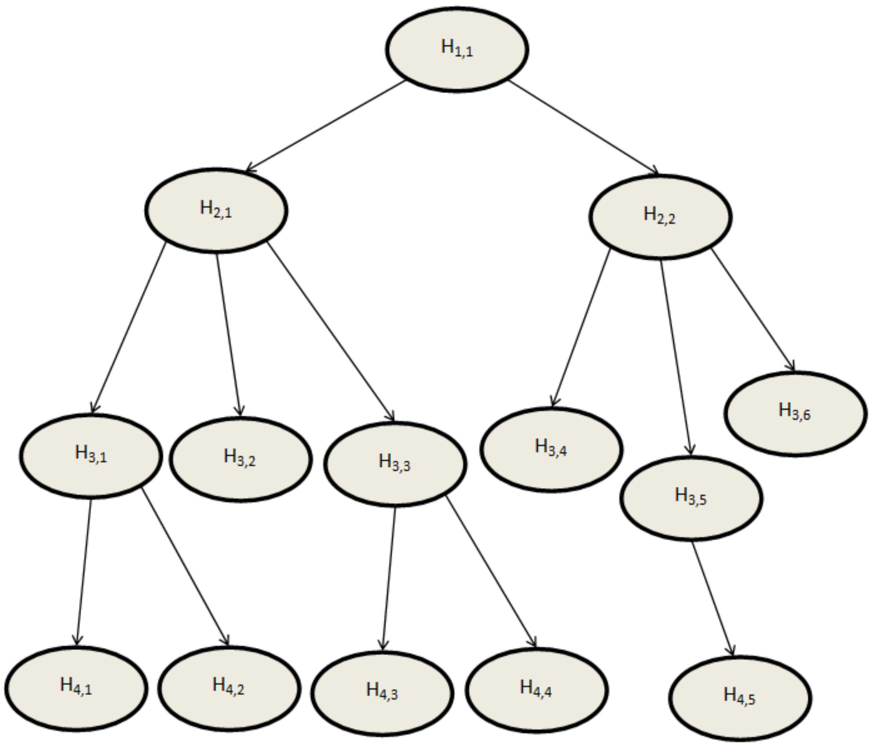

Figure 1. An algorithm-tree example generated from UTM

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 $A_1(Q), A_2(Q) \subset A(Q)$, then named that algorithm $A_1(Q)$ is equivalent to algorithm $A_2(Q)$ at problem Q and denoted by $A_1(Q) \overset{Q}{\Leftrightarrow} A_2(Q)$.

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 $T_R(A(Q))$. i.e., $UTM_Q \Rightarrow A(Q) \Rightarrow T_R(A(Q))$.

An example for algorithm-tree as below:

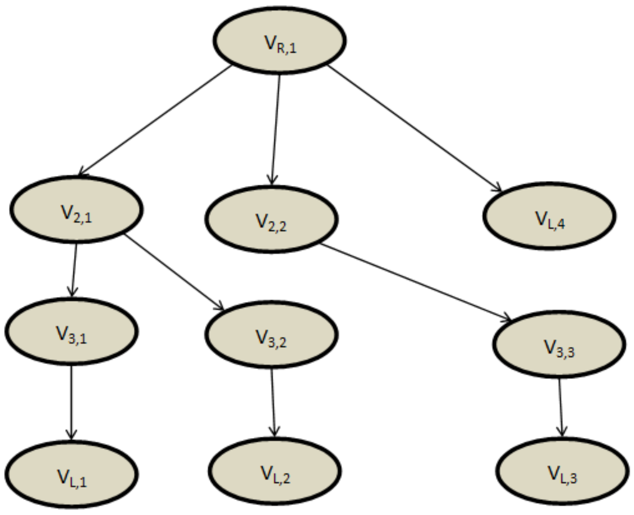

Figure 2. An example for algorithm-tree

Remarks for algorithm-tree:

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

3) Path B_1 and path B_2, 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.

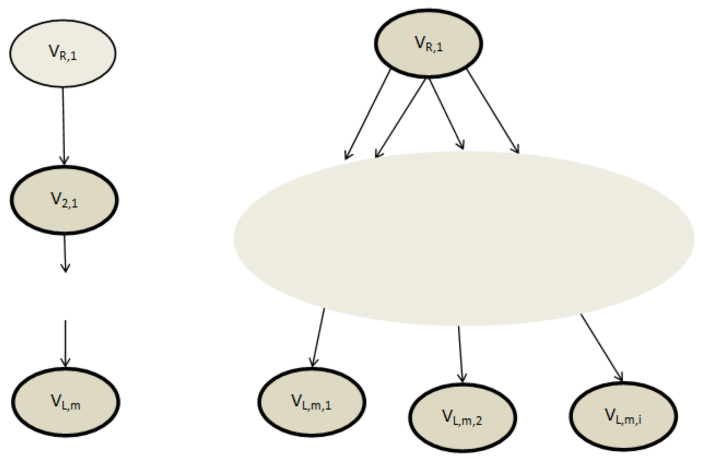

Figure 3. Algorithm-tree for single-root-and-single/multiple-leaf

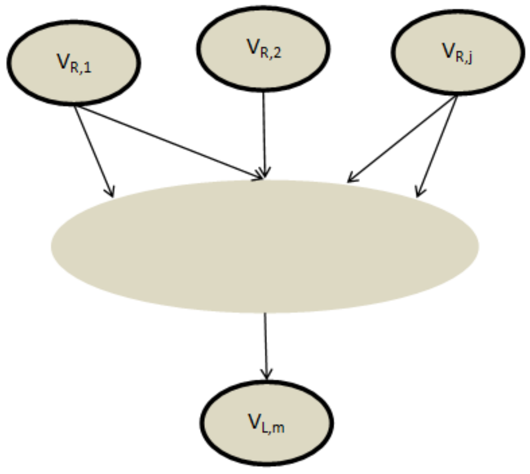

Figure 4. Algorithm-tree for multiple-root-and-single-leaf

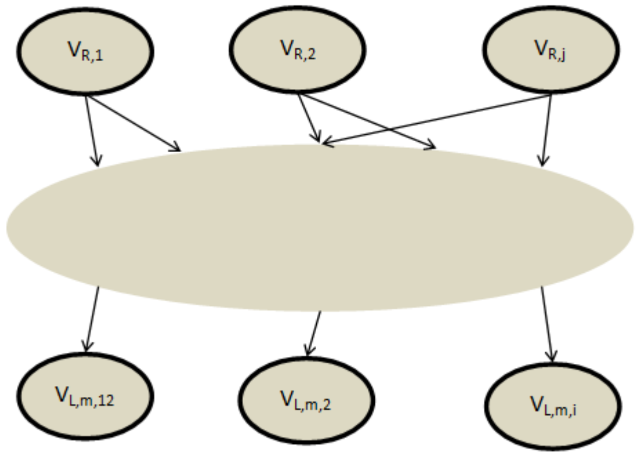

Figure 5. Algorithm-tree for multiple-root-and-multiple-leaf

Hierarchy-branch structure, briefly HBS, is produced by problem Q itself or by algorithm $A(Q)$. Many problems are acted on graphs have natural HBS in intuition. And all problems which contain recursion structure also have natural HBS.

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., $DTM \subset NTM$.

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 $T_R(A(Q))$ 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 $T_R(A(Q))$;

3) For multiple-root-and-multiple-leaf algorithm-tree, if root-vertex number is s and leaf-vertex number is m, then its breadth

$$N(B)=\sum_{i=1}^{s}\sum_{k=1}^{m}N\left(B_{R,i\to L,k}\right)$$

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 $V_{R,1}$, branch number from root-vertex $V_{R,1}$ to all leaf-vertex is

$N\left(B_{R,1\to L,k}\right)=\sum_{k=1}^{m}N\left(B_{R,1\to L,k}\right)$

For root-vertex $V_{R,2}$, branch number from root-vertex $V_{R,2}$ to all leaf-vertex is $N\left(B_{R,2\to L,k}\right)=\sum_{k=1}^{m}N\left(B_{R,2\to L,k}\right)$

Thus the total branch number in algorithm-tree $T_R(A(Q))$ is $N(B)=\sum_{i=1}^{s}\sum_{k=1}^{m}N\left(B_{R,i\to L,k}\right)$. ▮

Property 2.5.

Depth $N(D)$ of algorithm-tree $T_R(A(Q))$ is equal to the maximal hierarchy number of algorithm-tree $T_R(A(Q))$ 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. $0\le k\le n$, $0\le i\le n$.

Definition 2.7.

For time-complexity $T(A(Q))=f(n)$, i.e., running time cost of $A(Q)$, if

algorithm-set $A(Q)=\{A_1(Q),A_2(Q),\cdots,A_l(Q)\}$, $T(A_i(Q))=f_i(n)$, $T(A_j(Q))=f_j(n)$, and

$f_i(n)\le f_j(n)$, $i,j\le l$; then named as $A_i(Q)\overset{T}{\le}A_j(Q)$.

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 $A(Q)=\{A_1(Q),A_2(Q),\cdots,A_l(Q)\}$, $S(A_i(Q))=g_i(n)$, $S(A_j(Q))=g_j(n)$, and $g_i(n)\le g_j(n)$, $i,j\le l$; then named as $A_i(Q)\overset{S}{\le}A_j(Q)$.

2.2 Optimization for Algorithm $A(Q)$

Definition 2.9.

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

Proof.

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

1) $A(Q) \subset P \Rightarrow P-A(Q)$,

2) $A(Q) = A^{P}(Q) \cup A^{NP}(Q)$, and $A^{P}(Q) \subset P$, $A^{NP}(Q) \subset NP$, these will imply $A(Q) \subset P$, hence $A(Q) = A^{P}(Q) \cup A^{NP}(Q) \Rightarrow P-A(Q)$,

3) $A(Q) \subset NP \Rightarrow NP-A(Q)$,

Because $NP-A(Q) \Leftrightarrow P-A(Q) = \varnothing$, 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 $m(w_1) = m(w_2) = \cdots = m(w_n) = k$

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;"

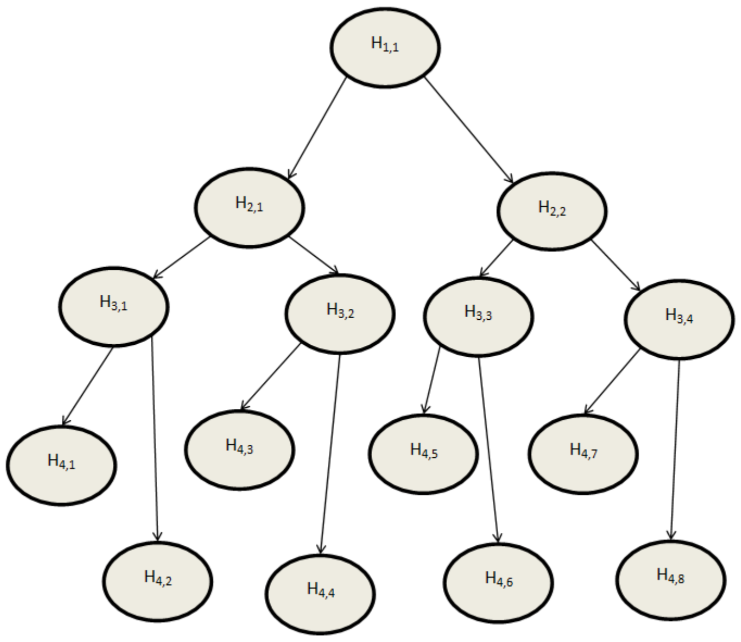

Figure 6. An example of $2-$ homogeneous algorithm-tree generated from NTM_k

2.4 Some Properties of Brute-Force Algorithm

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

$T(n,\ i_j)=T\left(\prod_{j=1}^{n}(n-i_j)\right)\sim O(n^n)$, $n-j\geq 0$, where n,i_j are integers .

If $(n-i_j)=1$, i.e., $i_1=n-1$ and $i_2=i_3=\cdots=i_n=n$ then $T(1)\sim O(1^1)=O(1)$, or if $i_1=i_2=\cdots=i_n=n-1$, then $T(1^n)\sim O(n)$;

If $(n-i_j)=2$, i.e. $i_1=i_2=\cdots=i_n=n-2$, then $T(2^n)\sim O(2^n)$;

…

If $(n-i_j)=n$, i.e., $i_i=i_2=\cdots=i_n=0$, then $T(n^n)\sim O(n^n)$;

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

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

For the $T(n)\sim O(n^n)$, the function n^n 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 $C_{CE}=\left(T\left(A(Q)\right),S\left(A(Q)\right)\right)$, if both $T\left(A(Q)\right)\subset POLY$, i.e., $T\left(A(Q)\right)$ is polynomial time solvable (the same below), and $S\left(A(Q)\right)\subset POLY$, i.e., $S\left(A(Q)\right)$ is polynomial space solvable(the same below), then $C_{CE}\subset POLY$, named as $C_{CE}-POLY$; if at least $T\left(A(Q)\right)\subset EXP$, i.e., $T\left(A(Q)\right)$ is exponential solvable (the same below), or $S\left(A(Q)\right)\subset EXP$, i.e., $S\left(A(Q)\right)$ is exponential solvable(the same below), then $C_{CE}\subset EXP$, named as $C_{CE}-EXP$. With another word, $C_{CE}-EXP$ means that at least one of $T\left(A(Q)\right)\subset EXP$ and $S\left(A(Q)\right)\subset EXP$ 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 $C_{CE}\subset POLY$, named as the first-type NP problem, denoted by NP_1; if $C_{CE}\subset EXP$, named as the second-type NP problem, denoted by NP_2.

Obviously $NP=NP_1\cup NP_2$.

So, we obtain that under $s>1$ and $A(Q)\subset NP_1$, there exists a mapping

$$g_{s>1}:\left(T\left(A(Q)\right)\subset EXP,S\left(A(Q)\right)\subset POLY\right)\rightarrow\left(T\left(A(Q)\right)\subset POLY,S\left(A(Q)\subset POLY\right)\right)$$

That is

$$g_{s>1}:NP_1-A(Q)\rightarrow P-A(Q)$$

Conjecture.

NP is yielded from hierarchy-branch structure, natural HBS, inside problem Q. And NP_1 is yielded from the problem Q which HBS is not coupled with string-length $n(w)$, NP_2 is yielded from the problem Q which HBS is coupled with string-length $n(w)$. ▌

With another word, for any $s\geq 1$, NP_1 is easy-to-solvable both in polynomial time and polynomial space; 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.

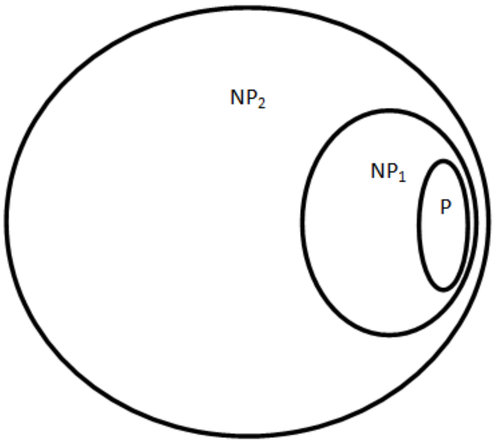

Figure 7. Venn-diagram for $P\subset NP_1\subset NP_2$

Lemma 3.3.

$NP-A(Q)\neq\varnothing$

Proof.

Because of the **Lemma 3.3.** is an existence lemma. We can proof this lemma by existence of examples.

Example 1. An example for NP_1

$$\overbrace{\sum_{i_1=1}^{n}\sum_{i_2=2}^{n}\cdots\sum_{i_n=1}^{n}}^{n}\left(a_{i_1,i_2,\cdots,i_n}\right)$$

1) If $s=1$, the sum needs n^n-1 steps, and $T(A(Q))\sim O(n^n)$, 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 $m(i-1)+(m-1)=(mi-1)=n^n-1=T(A(n^n))\sim O(n^n)$;

3) If $s=m>1$, then time-complexity $T(A(Q))\overset{S(A(Q))\sim O(s)}{\sim} O(i)$, $S(A(Q))\sim O(s=m)$;

4) The continued-sum exists only one solution;

5) This **Example 1.** belongs to $NP_1\subset P$.

Example 2. An example for NP_2

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 $n-$ code-element, so we can obtain n^n essential-code, and then each time select i essential-code to bind into $i\times n-$ bit key by second permutation of i essential-code, thus the total number of $i\times n-$ bit key is $P_i^{n^n}=n^n(n^n-1)(n^n-2)\cdots(n^n-i+1)$ and $P_{n^n}^{n^n}=n^n!$.

The detail is below.

$$\begin{cases} a_{1,1},a_{1,2},\cdots,a_{1,n} \\ a_{2,1},a_{2,2},\cdots,a_{2,n} \\ \cdots \\ a_{n,1},a_{n,2},\cdots,a_{n,n} \end{cases}$$

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 $n-$ 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 $A=\{A_1,A_{2,}\cdots,A_i\}$, chain these i essential-code by the method of second permutation below

$P_i^i\{A_1,A_2,\cdots,A_i\}=i!$

Thus the total number of $i\times n-$ bit key is $P_i^{n^n}=n^n(n^n-1)(n^n-2)\cdots(n^n-i+1)$ and $P_{n^n}^{n^n}=n^n!$.

1) To break the key, if $s=1$ only brute-force search can be used, and needs computation of $(i!)P_i^{n^n}=(i!)(n^n!)$ steps.

2) if $s=n^{n-1}$, then $T(A(n^n))\overset{S(A(Q))\sim O(n^n)}{\sim} O(n(i!))$, $S(n^n)\sim O(s=n^{n-1})$;

3) This **Example 2.** belongs to NP_2.

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

The **Lemma 3.3.** shows that $NP-A(Q)$ 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 $A(n^n)\subset NP_2$, $s>1$, and $s=n^{n-1}$, then $T(A(n^n))\sim O(n)$, $S(A(n^n))\sim O(n^{n-1})$.

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

2) For $A(n^n)\subset NP_2$, $s>1$, and $s=n^i$, $0\le i\le(n-1)$, i.e., $n=i+1$, then $T(A(n^n))\sim O(i+n^{n-i})$, $S(A(n^n))\sim O(n^i)$

Proof.

In induction. In hierarchy-first-read to algorithm-tree $T_R(A(Q))$.

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

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 $T(A(n^n))\sim O(i+n^{n-i}=1+n^{n-1})$, $S(A(n^n))\sim O(n^1)$;

For the third hierarchy vertex must need $n^2-(n-1)$ 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 time $i=2$, then $T\left(A\left(n^n\right)\right)\sim O\left(i+n^{n-i}=2+n^{n-2}\right)$, $S\left(A\left(n^n\right)\right)\sim O\left(n^2\right)$;

For the fourth hierarchy vertex must need $n^3-\left(n^2-(n-1)\right)$ processors, because of the $n^2-(n-1)$ 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 $T\left(A\left(n^n\right)\right)\sim O\left(i+n^{n-i}=3+n^{n-3}\right)$, $S\left(A\left(n^n\right)\right)\sim O\left(n^3\right)$;

…

Thus, for the n^{th} hierarchy vertex, at this time $i=n-1$, must need

$$p=n^{n-1}-\left(n^{n-2}-\left(n^{n-3}-\left(n^{n-4}-\cdots\left(n^2-(n-1)\right)\right)\right)\right)$$
$$=n^i-\left(n^{i-1}-\left(n^{i-2}-\left(n^{i-3}-\cdots\left(n^2-(n-1)\right)\right)\right)\right)\sim O\left(n^{n-1}\right)$$

processors, therefore $S\left(A\left(n^n\right)\right)=p\sim O\left(n^{n-1}\right)$, depth of algorithm-tree $T_R\left(A\left(n^n\right)\right)$ is $D(N)$, $T\left(A\left(n^n\right)\right)=\left(D(N)-1+n\right)\sim O\left(i+n^{n-i}=2n-1\right)=O(n)$. ▌

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 $NP-A_{s>1}(Q)$, that contains $s=n^{n-1}$ parallel processors, this **Theorem 3.4.** shows us that if for NP_2 there exists a mapping

$$f_{s>1}:\left(T\left(A\left(n^n\right)\right)\subset EXP,S\left(A\left(n^n\right)\right)\subset POLY\right)\to\left(T\left(A\left(n^n\right)\right)\subset POLY,S\left(A\left(n^n\right)\right)\subset EXP\right)$$

That is

$$f_{s>1}:NP_2-A(Q)\to NP_2-A(Q)$$

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

Theorem 3.5.

For $A_{s>1}(Q)\subset NP_2$, algorithm $NP_2-A_{s>1}(Q)$ 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 $f_{s>1}$ such that

$$f_{s>1}:\left(T\left(A\left(n^n\right)\right)\subset EXP,S\left(A\left(n^n\right)\right)\subset POLY\right)\to\left(T\left(A\left(n^n\right)\right)\subset POLY,S\left(A\left(n^n\right)\right)\subset EXP\right)$$

Now we will proof only that algorithm $NP_2-A_{s>1}(Q)$ 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 $A\left(n^n\right)$ only brute-force search can be used, $A\left(n^n\right)\subset NP_2$, therefore algorithm $NP_2-A_{s>1}(Q)$ is the fastest algorithm for time-complexity $T\left(A\left(n^n\right)\right)$. ▌

Corollary 3.6.

For $s>1$, $s=2^i$, and $A\left(2^n\right)\subset NP_2$, then $T\left(2^n\right)\sim O\left(D(N)-1+2^{n-i}\right)$, $S\left(2^n\right)\sim O\left(2^i\right)$.

Proof.

In induction. In hierarchy-first-read to algorithm-tree $T_R\left(A\left(2^n\right)\right)$.

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

For the second hierarchy vertex must need $2-1$ 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 $T\left(A\left(2^n\right)\right)\sim O\left(i+2^{n-i}=1+2^{n-1}\right)$, $S\left(A\left(2^n\right)\right)\sim O\left(2^{i=1}=2\right)$;

For the third hierarchy vertex must need $2^2-(2-1)$ processors, because of the $2-1$ 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 time $i=2$, then $T\left(A\left(2^n\right)\right)\sim O\left(i+2^{n-i}=2+2^{n-2}\right)$, $S\left(A\left(2^n\right)\right)\sim O\left(2^{i=2}=4\right)$;

For the fourth hierarchy vertex must need $2^3-\left(2^2-(2-1)\right)$ processors, because of the $2^2-(2-1)$ 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 $T\left(A\left(2^n\right)\right)\sim O\left(i+2^{n-i}=3+2^{n-3}\right)$, $S\left(A\left(2^n\right)\right)\sim O\left(2^{i=3}=8\right)$;

…

Thus, for the n^{th} hierarchy vertex, at this time $i=n-1$, must need

$$\begin{aligned} p &= 2^{n-1} - \left(2^{n-2} - \left(2^{n-3} - \left(2^{n-4} - \cdots \left(2^2 - (2-1)\right)\right)\right)\right) \\ &= 2^{i} - \left(2^{i-1} - \left(2^{i-2} - \left(2^{i-3} - \cdots \left(2^2 - (2-1)\right)\right)\right)\right) \sim O\left(2^{i=n-1}\right) \end{aligned}$$

processors, therefore $S\left(A\left(2^n\right)\right) = p \sim O\left(2^{i=n-1} = 2^{n-1}\right)$, depth of algorithm-tree $T_R\left(A\left(2^n\right)\right)$ is $D(N)$, $T\left(A\left(2^n\right)\right) = \left(D(N) - 1 + 2^{n-i}\right) \sim O(n - 1 + 2 = n + 1) = O(n)$. ∎

Corollary 3.7.

For $s > 1$, and $A_{s>1}(n!) \subset NP_2$, obvious $A_{s>1}(n!)$ is H_i- homogeneous algorithm, then $T(n!) \sim O\left(\left(i + (n-i-1)!\right)\right)$, $S(n!) \sim O\left((n-i-1)!\right)$.

Proof.

In induction. In hierarchy-first-read to algorithm-tree $T_R\left(A(n!)\right)$.

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

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 $T\left(A(n!)\right) \sim O\left(1 + (n-1)\right)$, $S\left(A(n!)\right) \sim O\left((n-1) - 1\right)$;

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 the second hierarchy and now can be used in the third hierarchy vertex, at this time $i = 2$, then $T\left(A(n!)\right) \sim O\left(i + n(n-1) = 2 + n(n-1)\right)$, $S\left(A(n!)\right) \sim O\left(n(n-1) - n = n(n-2)\right)$;

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

$T\left(A(n!)\right) \sim O\left(i + n(n-1)(n-2) = 3 + n(n-1)(n-2)\right)$, $S\left(A(n!)\right) \sim O\left(n(n-1)(n-2) - n(n-1) = n(n-1)(n-3)\right)$;

…

Thus, for the n^{th} hierarchy vertex, at this time $i = n-1$, must need

$$\begin{aligned} p &= n(n-1)(n-2)\cdots(n-i) - n(n-1)(n-2)\cdots(n-i-1) \\ &= (n-i-2)! \end{aligned}$$

processors, therefore $S\left(A(n!)\right) = p \sim O\left((n-i-2)!\right)$, depth of algorithm-tree $T_R\left(A(n!)\right)$ is $D(N)$, $T\left(A(n!)\right) = \left(D(N) - 1 + 1\right) \sim O(n)$. ∎

Theorem 3.8.

If $f(n)$ is a function and $n = g(m)$, for $s > 1$ and $A_{s>1}\left(Q = f(n)\right) \subset NP$, then $A_{s>1}\left(f(n)\right) \overset{n=g(m)}{=} A_{s>1}\left(f\left(g(m)\right)\right)$, $O(n) = O\left(f\left(g(m)\right)\right)$.

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 $A_{s>1}(Q) \subset NP_1$, If $(in_2)^j = n_2^{n_2}$, and $in_2 = n_1$, in which both $i, j > 0$ and both i, j integer constant, then

$$j = \frac{n_2 \log n_2}{\log i + \log n_2}, \quad \text{and} \quad n_1^{\frac{n_2 \log n_2}{\log i + \log n_2}} = n_2^{n_2}, \quad i = n_2^{\frac{n_2}{j} - 1}.$$

Proof.

For $A_{s>1}(Q) \subset NP_1$, 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_1 freely. If we make $s = i$, then from $(in_2)^j = n_2^{n_2}$ we obtain below

$j(\log i + \log n_2) = n_2 \log n_2$, then $j = \frac{n_2 \log n_2}{\log i + \log n_2}$,

And from $in_2 = n_1$, and , $(in_2)^j = n_2^{n_2}$, then $i = n_2^{\frac{n_2}{j}-1}$. ∎

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 $g_{s>1}: NP_1 \to P$, and mapping $f_{s>1}$ such that $f_{s>1}: NP_2 \to NP_2$, therefore under $s>1$, $NP_2 \neq \varnothing \Rightarrow P \neq NP$, obviously under $s=1$, $NP_1 \cup NP_2 = NP \Rightarrow P \neq NP$.

References

Bell, J. L., & Machover, M. (1977). *A course in mathematical logic*. Elsevier.

Jon, K., & Eva, T. (2006). *Algorithm Design*. Pearson Education Inc, Reprint by POSTS & TELECOM PRESS, Copyright 2019, ISBN: 978-7-115-49592-1.

Machael, S. (2018). *Introduction to the Theory of Computation* (3rd ed.). Cengage Learning Asia Pte Ltd.

Huth, M., & Ryan, M. (2004). *Logic in Computer Science: Modelling and reasoning about systems*. Cambridge university press. https://doi.org/10.1017/CBO9780511810275

Thomas, J. (2002). *Set theory*(The third millennium edition, revised and expanded) Berlin, Springer, 2002.

Jiang, Y. (2019). Mathematical foundation for dialectical logic. Lambert Academic Publishing, Singapore, 2019.

Manin, Y. I. (2009). *A course in mathematical logic for mathematicians* (Vol. 53). Springer Science & Business Media. https://doi.org/10.1007/978-1-4419-0615-1

Copyrights