Testing a new idea to solve the P = NP problem with mathematical induction

Background. *P* and *NP* are two classes (sets) of languages in Computer Science. An open problem is whether $P = NP$. This paper tests a new idea to compare the two language sets and attempts to prove that these two language sets consist of same languages by elementary mathematical methods and basic knowledge of Turing machine. **Methods**. By introducing a filter function $C(M,w)$ that is the number of configurations which have more than one children (nondeterministic moves) in the shortest accept computation path of a nondeterministic Turing machine M for input w, for any language $L(M) \in NP$, we can define a series of its subsets, $L_i(M) = \{w \mid w \in L(M) \wedge C(M,w) \leq i\}$, and a series of the subsets of NP as $Li = \{Li(M) \mid \forall M \bullet L(M) \in NP\}$. The nondeterministic multi-tape Turing machine is used to bridge two language sets L_i and L_{i+1}, by simulating the $(i+1)$-th nondeterministic move deterministically in multiple work tapes, to reduce one (the last) nondeterministic move. **Results**. The main result is that, with the above methods, the language set L_{i+1}, which seems more powerful, can be proved to be a subset of L_i. This result collapses $L_i \subseteq P$ for all $i \in \mathrm{N}$. With $NP = \bigcup_{i \in \mathrm{N}} L_i$, it is clear that $NP \subseteq P$. Because by definition $P \subseteq NP$, we have $P = NP$. **Discussion**. There can be other ways to define the subsets L_i and prove the same result. The result can be extended to cover any sets of time functions C, if $\forall f \bullet f \in C \Rightarrow f^2 \in C$, then $DTIME(C) = NTIME(C)$. This paper does not show any ways to find a solution in P for the problem known in NP.

Testing New Idea to Solve P=NP Problem with Mathematical Induction

HUANG, YU BIN

yubinhuang@yahoo.com

PeerJ PrePrints | https://dx.doi.org/10.7287/peerj.preprints.1455v1 | CC-BY 4.0 Open Access | rec: 27 Oct 2015, publ: 27 Oct 2015

Abstract

Background. P and NP are two classes (sets) of languages in Computer Science. An open problem is whether $P = NP$. This paper tests a new idea to compare the two language sets and attempts to prove that these two language sets consist of same languages by elementary mathematical methods and basic knowledge of Turing machine.

Methods. By introducing a filter function $C(M,w)$ that is the number of configurations which have more than one children (nondeterministic moves) in the shortest accept computation path of a nondeterministic Turing machine M for input w, for any language $L(M) \in NP$, we can define a series of its subsets, $L_i(M) = \{w \mid w \in L(M) \wedge C(M,w) \le i\}$, and a series of the subsets of NP as $L_i = \{L_i(M) \mid \forall M \cdot L(M) \in NP\}$. The nondeterministic multi-tape Turing machine is used to bridge two language sets L_i and L_{i+1}, by simulating the $(i+1)$-th nondeterministic move deterministically in multiple work tapes, to reduce one (the last) nondeterministic move.

Results. The main result is that, with the above methods, the language set L_{i+1}, which seems more powerful, can be proved to be a subset of L_i. This result collapses $L_i \subseteq P$ for all $i \in \mathrm{N}$. With $NP = \bigcup_{i \in \mathrm{N}} L_i$, it is clear that $NP \subseteq P$. Because by definition $P \subseteq NP$, we have $P = NP$.

Discussion. There can be other ways to define the subsets L_i and prove the same result. The result can be extended to cover any sets of time functions C, if $\forall f \cdot f \in C \Rightarrow f^2 \in C$, then $DTIME(C) = NTIME(C)$. This paper does not show any ways to find a solution in P for the problem known in NP.

Keywords: P versus NP, Computational Complexity, nondeterministic multi-tape Turing machine

Testing New Idea to Solve P=NP Problem with Mathematical Induction

INTRODUCTION

P and NP are two classes (sets) of languages in Computer Science. An open problem is whether $P = NP$. *Stephen Cook* provided the formal description of this problem[1]. For more information please reference the textbook *Theory of Computational Complexity*[2].

To compare two sets, the mathematical way to prove $P = NP$ will be proving $\forall L \cdot L \in P \Leftrightarrow L \in NP$ (since by definition $P \subseteq NP$, it can also be $\forall L \cdot L \in NP \Rightarrow L \in P$); and $P \neq NP$ equals $\exists L \cdot L \notin P \wedge L \in NP$. However, a quick scan of *Milestones*[3] of researches reveals that:

1. Most of the approaches to prove $P = NP$ attempt to prove $\exists L \cdot L \in NP \Rightarrow L \in P$. The concern of this approach is, for a certain problem with a well-known solution in NP, giving a solution in P does not prove $\forall L \cdot L \in NP \Rightarrow L \in P$. Because $\forall L \cdot L \in P \Rightarrow L \in NP$, anyone can construct unlimited different $L \in P$ that satisfy $\exists L \cdot L \in NP \Rightarrow L \in P$. But this does not construct a proof of $\forall L \cdot L \in NP \Rightarrow L \in P$.
2. Most of the approaches to declare $\exists L \cdot L \notin P \wedge L \in NP$ only prove that they fail to find/prove a particular language $L \in NP$ that is actually also in P. Furthermore, even they prove that it is impossible to find $L \in P$, maybe the Turing machine searching such language never halts or it is undecidable, this still does not prove that $L \in P$ does not exist.
3. Some other papers are not written with well-established computational models and, hence, make difficult for the community to understand.

This paper tests a new idea[3] to solve $\forall L \cdot L \in NP \Rightarrow L \in P$ using elementary methods that compare two language sets, and prove P and NP are consists of same languages.

PeerJ PrePrints | https://dx.doi.org/10.7287/peerj.preprints.1455v1 | CC-BY 4.0 Open Access | rec: 27 Oct 2015, publ: 27 Oct 2015

METHOD

P and NP are two sets consist of infinite languages and each such language can have infinite accepted inputs. However, Turing machines are all about countable numbers. And the length of the accept computation paths of inputs are also countable integer numbers. These numbers of lengths can be arbitrary big, but by definition never infinite. These hint that elementary method such as mathematical induction can be helpful.

To apply mathematical induction, the languages or inputs should be associated with integer numbers. To achieve this goal, as the new idea, we can introduce a filter function $C(M,w)$ that is the number of configurations which have more than one children (nondeterministic moves) in the shortest accept computation path of a nondeterministic Turing machine M for input w. Without loss of generality, assume M is a one-tape Turing machine. For any language $L(M) \in NP$, we can then define:

1. A series of subsets of $L(M)$, $L_i(M) = \{w \mid w \in L(M) \wedge C(M,w) \le i\}$ for all $i \in \mathrm{N}$; and
2. A series of subsets of NP as $L_i = \{L_i(M) \mid \forall M \cdot L(M) \in NP\}$ for all $i \in \mathrm{N}$.

Apparently, $L_0(M) \in P$ and $L_0 \subseteq P$.

To apply mathematical induction, the nondeterministic multi-tape Turing machine is used to bridge two languages $L_i(M)$ and $L_{i+1}(M)$, and two language sets L_i and L_{i+1}, defined by $C(M,w) \le i$ and $C(M,w) \le i+1$. Given any $w \in L_{i+1}(M)$ whose values of $C(M,w)$ are at most $i+1$, we can construct a nondeterministic multi-tape Turing machine to simulate the M that accepts all $w \in L_{i+1}(M)$ with at most i nondeterministic moves and same time complexity, by simulating the $(i+1)$-th nondeterministic (multiple possible) move deterministically in multiple work tapes.

PeerJ PrePrints | https://dx.doi.org/10.7287/peerj.preprints.1455v1 | CC-BY 4.0 Open Access | rec: 27 Oct 2015, publ: 27 Oct 2015

Because the square of a polynomial run time function is still a polynomial run time function, we can construct a one tape nondeterministic Turing machine M' to simulate the above nondeterministic multi-tape Turing machine which still has polynomial run time. Means for any $L_{i+1}(M) \in NP$, there exists M', $L_i(M') = L_{i+1}(M)$. Hence, we can get $L_{i+1}(M) \in L_i$. That means $L_{i+1} \subseteq L_i \subseteq P$ for any $i \in \mathrm{N}$. Finally, we can prove $NP = \bigcup_{i \in \mathrm{N}} L_i$ and $NP \subseteq P$.

SYMBOL & DEFINITIONS

ϕ

Empty set.

N

The set of non-negative integer numbers.

$\{s_k \mid k \in \mathrm{N}\}$

A set whose items are sets s_k. $\{s_k \mid k \in \mathrm{N}\} = \{s_0, s_1, \ldots s_k \ldots\}$.

$\bigcup$ **operator**

The union operator of sets.

1. $\bigcup_{i \in \mathrm{N}} s_i = s_0 \cup s_1 \cup \ldots \cup s_i \cup \ldots$ for all $i \in \mathrm{N}$; or
2. $\bigcup_{j \le i} s_i = s_0 \cup s_1 \cup \ldots \cup s_i$.

Language[1]

Let Σ be a finite alphabet (that is, a finite nonempty set) with at least two elements, and let Σ^* be the set of finite strings over Σ. Then a *language* over Σ is a subset L of Σ^*.

Turing machine[1]

A (one-tape) *Turing machine* M consists of a finite state control (i.e., a finite program) attached to a read/write head moving on an infinite tape. The tape is divided into squares, each

capable of storing one symbol from a finite alphabet Γ that includes the blank symbol b. Each machine M has a specified input alphabet Σ, which is a subset of Γ, not including the blank symbol b. At each step in a computation, M is in some state q in a specified finite set Q of possible states. Initially, a finite input over Σ is written on adjacent squares of the tape, all other squares are blank (contains b), the head scans the left-most symbol of the input, and M is in the initial state q_0. At each step M is in some state q and the head is scanning a tape square containing some tape symbol s, and the action performed depends on the pair (q,s) and is specified by the machine's transition function (or program) δ. The action consists of printing a symbol on the scanned square, moving the head left or right one square, and assuming a new state.

Formally, a Turing machine M is a tuple $\langle \Sigma, \Gamma, Q, \delta \rangle$, where Σ, Γ, Q are finite nonempty sets with $\Sigma \subseteq \Gamma$ and $b \in \Gamma - \Sigma$. The state set Q contains three special states q_0, q_{accept}, and q_{reject}. The *transition function* δ satisfies

$$\delta : (Q - \{q_{accept}, q_{reject}\}) \times \Gamma \to Q \times \Gamma \times \{-1,1\}.$$

If $\delta(q,s) = (q',s',h)$, the interpretation is that, if M is in state q scanning the symbol s, then q' is the new state, s' is the symbol printed, and the tape head moves left or right one square depending on whether h is -1 or 1.

We assume that the sets Q and Γ are disjoint.

Multi-tape Turing machine[2][12]

A *multi-tape Turing machine* is similar to a one-tape Turing machine with the following exceptions:

1. It has a finite number of tapes that extends infinitely to the both ends. Each tape is equipped with its own head. All tape heads are controlled by a common finite control.
2. There are two special tapes: an *input tape* and an *output tape*.
    a. The *input tape* is used to hold the inputs only; it is a read-only tape that prohibits erasing and writing.
    b. The *output tape* is used to hold the output string when the computation of a function is concerned; it is a write-only tape.
3. The other tapes are called the *work tapes*. All *work tapes* are allowed to read, erase, and write.
4. For a *k*-tape Turing machine, the *transition function* δ satisfies

    $\delta:\left(Q-\{q_{accept},q_{reject}\}\right)\times\Gamma^k \rightarrow Q\times\Gamma^k\times\{-1,1\}^k$
5. The initial setting of the input tape of the multi-tape Turing machine is the same as that of the one-tape Turing machine, and all other tapes of the multi-tapes Turing machine initially contain only blanks.

Configuration[1]

A *configuration* of M is a string xqy with $x\in\Gamma^*$, $y\in\Gamma^+$ and $q\in Q$.

The interpretation of the configuration xqy is that M is in state q with xy on its tape, with its head scanning the left-most symbol of y.

If C and C' are configurations, then $C\xrightarrow{M}C'$ if $C=xqsy$ and $\delta(q,s)=(q',s',h)$ and one of the following holds:

$C'=xs'q'y$ and $h=1$ and y is nonempty.

$C'=xs'q'b$ and $h=1$ and y is empty.

$C' = xq'as'y$ and $h = -1$ and $x = x'a$ for some $a \in \Gamma$.

$C' = q'bs'y$ and $h = -1$ and x is empty.

A configuration xqy *halts* if $q \in \{q_{accept}, q_{reject}\}$. Note that for each non-halting configuration C there is a unique configuration C' such that $C \xrightarrow{M} C'$.

Discussion. The possible number of C' is no more than the number of possible transitions of $\delta(q,s)$, regardless of what x or y is.

Computation[1]

The *computation* of M on input $w \in \Sigma^*$ is the unique sequence C_0, C_1, ... of configurations such that $C_0 = q_0 w$ (or $C_0 = q_0 b$ if w is empty) and $C_i \xrightarrow{M} C_{i+1}$ for each i with C_{i+1} in the computation, and either the sequence is infinite or it ends in a halting configuration. If the computation is finite, then the number of steps is one less than the number of configurations; otherwise the number of steps is infinite.

Accept[1]

We say that M *accepts* w if and only if the computation is finite and the final configuration contains the state q_{accept}.

The *language accepted by* M, denoted $L(M)$, has associated alphabet Σ and is defined by $L(M) = \{w \in \Sigma^* \mid M \quad accepts \quad w\}$.

Deterministic Turing machine[2][14]

Each configuration of a machine there is at most one move to make, and hence there is at most one next configuration, this kind of the machines are defined as *deterministic Turing machine*.

PeerJ PrePrints | https://dx.doi.org/10.7287/peerj.preprints.1455v1 | CC-BY 4.0 Open Access | rec: 27 Oct 2015, publ: 27 Oct 2015

Since each configuration of M may at most have one next configuration, the computation of a deterministic Turing machine is a computation path. Such as:

$$C_{initial} \xrightarrow{M} \ldots \xrightarrow{M} C_i \xrightarrow{M} \ldots \xrightarrow{M} C_{final}$$

Nondeterministic Turing machine[2][14]

If allow more than one moves for some configurations, and hence those configurations have more than one next configurations, the machine is called a *nondeterministic Turing machine*.

Since each configuration of M may have more than one next configurations, the computation of a nondeterministic Turing machine M on an input w is, in general, a computation tree. In the computation tree, each node is a configuration and all its next configurations are its children. The root of the tree is the initial configuration.

We can still find the computation path accepts the given input by cutting all branches that do not lead to the (shortest) final accept configuration. After that, it looks like:

$$C_{initial} \xrightarrow{M} \ldots \xrightarrow{M} C_i \xrightarrow{M} C_{i+1} \xrightarrow{M} \ldots \xrightarrow{M} C_{final}$$

All configurations after the initial configuration can be one of the many possible configurations. For example, if C_i has more than one children, C_{i+1} is the one of the children on the path to C_{final}.

Polynomial run time[1]

We denote by $t_M(w)$ the number of steps in the computation of M on input w. If this computation never halts, then $t_M(w) = \infty$.

Note that, for deterministic Turing machine, the runtime is the length of the computation path. For the nondeterministic Turing machine, the runtime is the shortest computation path in the computation tree which accepts the given input.

For $n \in \mathrm{N}$ we denote by $T_M(n)$ the *worst case run time* of M ; that is,

$T_M(n) = \max\{t_M(w) | w \in \Sigma^n\}$, where Σ^n is the set of all strings over Σ of length n.

We say that M *runs in polynomial time* if there exists k such that for all n, $T_M(n) \leq n^k + k$.

DTIME[2][18]

We define $DTIME(t)$ to be the class of languages L that are accepted by deterministic Turing machines M with $t_M(n) \leq t(n)$ for almost all $n \geq 0$. We let

$DTIME(C) = \bigcup_{t \in C} DTIME(t)$.

NTIME[2][19]

We define $NTIME(t)$ to be the class of languages L that are accepted by nondeterministic Turing machines M with $t_M(n) \leq t(n)$ for almost all $n > 0$. We let

$NTIME(C) = \bigcup_{t \in C} NTIME(t)$.

P[2][21]

$$P = DTIME(poly) = \bigcup_{k \in \mathrm{N}} DTIME(n^k)$$

NP[2][21]

$$NP = NTIME(poly) = \bigcup_{k \in \mathrm{N}} NTIME(n^k)$$

$C(M,w)$

For any input $w \in L(M)$, in the computation tree of the nondeterministic Turing machine M which accepts $L(M)$, let $C(M,w)$ be the number of configurations in the accept computation path for w which have more than one children in the computation tree.

$L_i(M)$

For any language $L(M) \in NP$, we can define a series of subsets of $L(M)$ as:

$$L_i(M) = \{w \mid w \in L(M) \wedge C(M,w) \le i\},\ i \in \mathrm{N}$$

By definition we have: $L_i(M) \in NP$, for any $i \in \mathrm{N}$.

Apparently we have: $L_0(M) \subseteq L_1(M) \subseteq \ldots \subseteq L_i(M) \subseteq \ldots \subseteq L(M)$ and $L_0(M) \in P$.

L_i

Let L_i be the set of $L_i(M)$ of all $L(M) \in NP$.

Apparently we have: $L_0 \subseteq P$.

PROPOSITIONS

Proposition 1. $L(M) = \bigcup_{i \in \mathrm{N}} L_i(M)$

Proof. Assume $L(M) \neq \bigcup_{i \in \mathrm{N}} L_i(M)$. It means there must exist at least one $w \in L(M)$ but $w \notin L_i(M)$ for all $i \in \mathrm{N}$. According to the definition of $L_i(M)$, for this w, $C(M,w) \notin \mathrm{N}$. So for input w, the Turing machine M will never halt. Means the Turing machine does not accept w. $w \notin L(M)$. This establishes a contradiction. ■

Proposition 2. $L(M) \in \{\bigcup_{j \le k} L_j(M) \mid k \in \mathrm{N}\}$

Proof. It is clear that $\bigcup_{i\in \mathrm{N}} L_i(M) \in \{\bigcup_{j\le k} L_j(M) \mid k \in \mathrm{N}\}$. According to Proposition 1, we have $L(M) \in \{\bigcup_{j\le k} L_j(M) \mid k \in \mathrm{N}\}$. ■

Proposition 3. $NP = \bigcup_{i\in \mathrm{N}} L_i$

Proof. By definition, $\bigcup_{i\in \mathrm{N}} L_i \subseteq NP$ and $NP = \{L(M) \mid L(M) \in NP\}$.

According to Proposition 2, $L(M) \in \{\bigcup_{j\le i} L_j(M) \mid i \in \mathrm{N}\}$, we have

$NP \subseteq \{\bigcup_{j\le i} L_j(M) \mid L(M) \in NP \wedge i \in \mathrm{N}\}$.

By definition, $\bigcup_{j\le i} L_j(M) = L_i(M)$. So $NP \subseteq \{L_i(M) \mid L(M) \in NP \wedge i \in \mathrm{N}\}$.

Because $L_i = \{L_i(M) \mid L(M) \in NP\}$, we have $NP \subseteq \bigcup_{i\in \mathrm{N}} L_i$.

Therefore $NP = \bigcup_{i\in \mathrm{N}} L_i$. ■

LEMMA

Lemma. For any multi-tape Turing machine M, there exists a one-tape Turing machine M' computing the same function as M in time $t_{M'}(n) = O((t_M(n))^2)$.

Proof. See Reference [2], Page 12, Page 23 ■

THEOREMS

Theorem 1. $L_{i+1} \subseteq L_i$

Proof. For any $L_{i+1}(M) \in L_{i+1}$. Let k be the maximum number of values that a δ can assume on some $(q,s) \in Q\times\Gamma$ of the one tape nondeterministic Turing machine M. We can construct a k work tape nondeterministic Turing machine M' that accepts the same language $L_{i+1}(M)$ with at most i nondeterministic moves.

The idea is to track and count every nondeterministic moves in the accept computation tree of input w for M. Before the $(i+1)$-th nondeterministic move, the multi-tape Turing machine works like M. At the time of the $(i+1)$-th nondeterministic move, the multi-tape Turing machine will move deterministically and all, at most k, possible moves will be simulated at the k work tapes. Because for every $w \in L_{i+1}(M)$, $C(M,w) \le i+1$. After the $(i+1)$-th nondeterministic move, both M and M' work deterministically until accept or reject the input w.

Assume M is a tuple $\langle \Sigma, \Gamma, Q, \delta \rangle$ and M' is another tuple $\langle \Sigma', \Gamma', Q', \delta' \rangle$. We have:

1. $\Gamma = \Gamma'$. Two Turing machines have same tape symbols.
2. For any state $q \in Q$, creates state $q_j \in Q'$ for $0 \le j \le i$. q_j is used by M' for states after the j-th and before the $(j+1)$-th nondeterministic move.
3. For any $0 < r \le k$ and any r number of $q \in Q$, denoted as $q_t \in Q$ for all $0 \le t < r$, creates state $q'_0 \# \cdots \# q'_{r-1} \in Q'$. $q'_0 \# \cdots \# q'_{r-1}$ is used by M' for states at and after the $(i+1)$-th nondeterministic move.
4. For any deterministic move $\delta(q,s) \to (q',s',l)$, creates $(i+1)$ different deterministic transitions for M': $\delta'\left(q_j, \underbrace{s,\ldots,s}_{k}\right) \to \left(q'_j, \underbrace{s',\ldots,s'}_{k}, \underbrace{l,\ldots,l}_{k}\right)$ for $0 \le j \le i$. q_j is the j-th state created for $q \in Q$ for M'. q'_j is the j-th state created for $q' \in Q'$ for M'.
5. For each move of nondeterministic moves $\delta(q,s) \to (q',s',l)$, creates $(i+1)$ (nondeterministic) transitions for M': $\delta'\left(q_j, \underbrace{s,\ldots,s}_{k}\right) \to \left(q'_{j+1}, \underbrace{s',\ldots,s'}_{k}, \underbrace{l,\ldots,l}_{k}\right)$ for $0 \le j \le i$. q_j is the j-th state created for $q \in Q$ for M'. q'_{j+1} is the $(j+1)$-th state

created for $q' \in Q'$ for M'. Note the difference, a nondeterministic move will switch M' using the next set of the states.

6. For nondeterministic moves $\delta(q,s) \to \{(q_0,s_0,l_0),\ldots,(q_r,s_r,l_r)\}$ where $0 \le r < k$, creates one deterministic transition for M':

$\delta'\left(q_i,\underbrace{s,\ldots,s}_{k}\right) \to \left(q_0'\#\cdots\#q_r',\underbrace{s_0,\ldots,s_r,s_0,\ldots}_{k},\underbrace{l_0,\ldots,l_r,l_0,\ldots}_{k}\right)$. Here, the s_0 and l_0 are written repeatedly to show that, if $r < k-1$, reuse s_0 and l_0 for the rest of the work tapes and work heads. So all work tapes and work heads have definitions (things to do).

7. For any combination $\left(q_0'\#\cdots\#q_r',\underbrace{s_0,\ldots,s_r,s_0,\ldots}_{k}\right)$ where $0 \le r < k$ and $q_0'\#\cdots\#q_r' \in Q'$,

   a. Creates

   $\delta'\left(q_0'\#\cdots\#q_r',\underbrace{s_0,\ldots,s_r,s_0,\ldots}_{k}\right) \to \left(q_0''\#\cdots\#q_r'',\underbrace{s_0',\ldots,s_r',s_0',\ldots}_{k},\underbrace{l_0,\ldots,l_r,l_0,\ldots}_{k}\right)$

   if and only if there exists deterministic move $\delta(q_t,s_t) \to (q_t',s_t',l_t)$ for all $0 \le t \le r$.

   b. Creates $\delta'\left(q_0'\#\cdots\#q_r',\underbrace{s_0,\ldots,s_r,s_0,\ldots}_{k}\right) \to \left(q_{reject}',\underbrace{s_0,\ldots,s_r,s_0,\ldots}_{k},\underbrace{1,\ldots}_{k}\right)$ if there exists at least one $0 \le t \le r$, deterministic move $\delta(q_t,s_t) \to (q_t',s_t',l_t)$ does not exist (no definition or only nondeterministic moves available).

8. The initial state of M' is $q_{initial}'$ that is the one created for $q_{initial}$ of M.

9. For any $q_{accept} \in Q$, the $q'_{accept,j} \in Q'$ where $q'_{accept,j}$ is the j-th state created for M' for q_{accept}.
10. State $q'_0 \# \cdots \# q'_r$ is an accept state of M' if and only if there exists t that q_t is an accept state of M.
11. For any $q_{reject} \in Q$, the $q'_{reject,j} \in Q'$ where $q'_{reject,j}$ is the j-th state created for M' for q_{reject}.
12. State $q'_0 \# \cdots \# q'_r$ is a reject state of M' if and only if for all t that q_t is a reject state of M.
13. M' first copy the input tape to all the work tapes.

It is clear that M' accepts same language as M. For every $w \in L(M)$, the length of the shortest computation path of the accept computation tree are the same. Regardless the initial $O(n)$ work to copy input tape to all work tapes, M and M' have same polynomial run time complexity.

Because $L_{i+1}(M) \in NP$ for $i \geq 0$, there exists r such that for all n, $T_M(n) = T_{M'}(n) \leq n^r + r$. According to the Lemma, there exists an one tape nondeterministic Turing machine M'' accepts $L_i(M')$ with $T_{M''}(n) = O(n^{2r})$, which is also in polynomial run time.

Notice that M' only need at most i nondeterministic moves to accept the input. So for any $L_{i+1}(M)$, we have an one tape nondeterministic Turing machine M'' in polynomial run time accepts the same language with at most i nondeterministic moves. Means $L_{i+1}(M) = L_i(M'')$.

By definition of L_i, for any $L(M) \in NP$, $L_{i+1}(M) \in L_i$.

Therefore, $L_{i+1} \subseteq L_i$. ■

Theorem 2. $L_i \subseteq P$ for all $i \in \mathrm{N}$.

Proof. Because of $L_0 \subseteq P$, and Theorem 1, if $L_i \subseteq P$, $L_{i+1} \subseteq L_i \subseteq P$, according to mathematical induction, $L_i \subseteq P$ for all $i \in \mathrm{N}$. ■

Theorem 3. $NP \subseteq P$

Proof. According to Theorem 2, we have $L_i \subseteq P$ for all $i \in \mathrm{N}$. Hence we have $\bigcup_{i\in\mathrm{N}} L_i \subseteq \bigcup_{i\in\mathrm{N}} P$. According to Proposition 3, we have $NP = \bigcup_{i\in\mathrm{N}} L_i$, means $NP \subseteq \bigcup_{i\in\mathrm{N}} P$. Because $\bigcup_{i\in\mathrm{N}} P = P$, $NP \subseteq P$. ■

Theorem 4. $P = NP$

Proof. It is trivial that $P \subseteq NP$. According to Theorem 3, $NP \subseteq P$. Hence $P = NP$. ■

DISCUSSION

Notice that there are unlimited ways to define the subsets of NP which satisfy $NP = \bigcup_{i\in\mathrm{N}} L_i$. With the conclusion of $P = NP$, all of such $L_i \subseteq P$. This paper only give one way to define such L_i and prove all such $L_i \subseteq P$. There may be other ways to define L_i and $L_i \subseteq P$ is also provable.

The result can be extended to cover any sets of time functions C, if $\forall f \cdot f \in C \Rightarrow f^2 \in C$, then $DTIME(C) = NTIME(C)$.

This paper does not give a way to find a solution in P for a problem in NP.

References

[1] COOK, STEPHEN. "THE P VERSUS NP PROBLEM.". Web.
<http://www.claymath.org/sites/default/files/pvsnp.pdf>

[2] DU, DING-ZHU and KO, KER-I. *THEORY OF COMPUTATIONAL COMPLEXITY*. New York: JOHN WILEY & SONS, INC., 2000. Print.

[3] The P-versus-NP page. Web
http://www.win.tue.nl/~gwoegi/P-versus-NP.htm