Complexity Theory

Tian-Ming Bu

East China Normal University

Outline

Time Complexity

Outline

Time Complexity

Time Complexity of Deterministic Turing Machine

Let M be a deterministic Turing machine that halts on all inputs. The ***running time*** or ***time complexity*** of M is the function $f: \mathcal{N} \longrightarrow \mathcal{N}$, where $f(n)$ is the maximum number of steps that M uses on any input of length n. If $f(n)$ is the running time of M, we say that M runs in time $f(n)$ and that M is an $f(n)$ time Turing machine. Customarily we use n to represent the length of the input.

Time Complexity of Deterministic Turing Machine

Let M be a deterministic Turing machine that halts on all inputs. The ***running time*** or ***time complexity*** of M is the function $f: \mathcal{N} \longrightarrow \mathcal{N}$, where $f(n)$ is the maximum number of steps that M uses on any input of length n. If $f(n)$ is the running time of M, we say that M runs in time $f(n)$ and that M is an $f(n)$ time Turing machine. Customarily we use n to represent the length of the input.

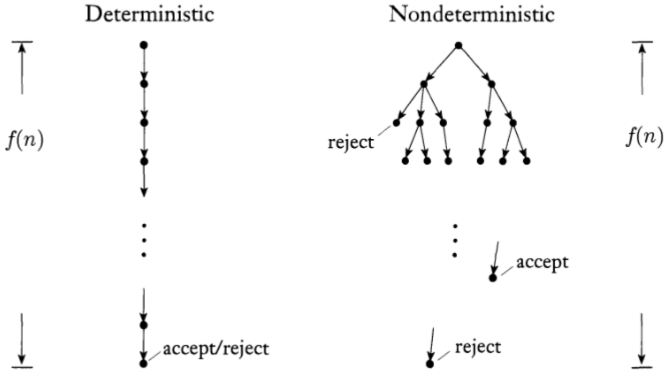
Let $t: \mathcal{N} \longrightarrow \mathcal{R}^+$ be a function. Define the ***time complexity class***, $\mathbf{TIME}(\boldsymbol{t(n)})$, to be the collection of all languages that are decidable by an $O(t(n))$ time Turing machine.

Time Complexity of Nondeterministic Turing Machine

Let N be a nondeterministic Turing machine that is a decider. The ***running time*** of N is the function $f: \mathcal{N} \longrightarrow \mathcal{N}$, where $f(n)$ is the maximum number of steps that N uses on any branch of its computation on any input of length n, as shown in the following figure.

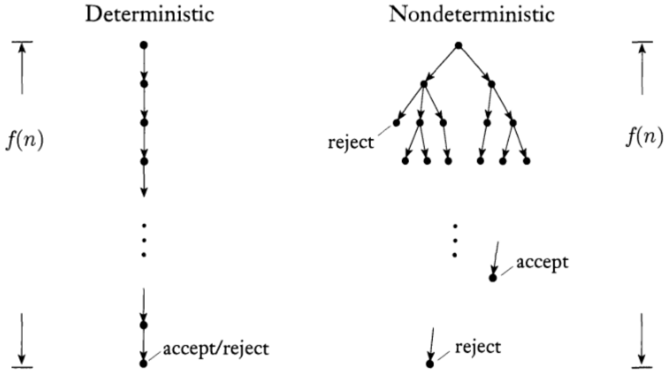
Time Complexity of Nondeterministic Turing Machine

Let N be a nondeterministic Turing machine that is a decider. The ***running time*** of N is the function $f: \mathcal{N} \longrightarrow \mathcal{N}$, where $f(n)$ is the maximum number of steps that N uses on any branch of its computation on any input of length n, as shown in the following figure.



Time Complexity of Nondeterministic Turing Machine

Let N be a nondeterministic Turing machine that is a decider. The ***running time*** of N is the function $f: \mathcal{N} \longrightarrow \mathcal{N}$, where $f(n)$ is the maximum number of steps that N uses on any branch of its computation on any input of length n, as shown in the following figure.



$$\textbf{NTIME}(\boldsymbol{t(n)}) = \{L \mid L \text{ is a language decided by a } O(t(n)) \text{ time nondeterministic Turing machine}\}.$$

Complexity Relationships among Models

Theorem

Let $t(n)$ be a function, where $t(n) \geq n$. Then every $t(n)$ time mulitape Turing machine has an equivalent $\mathcal{O}(t^2(n))$ time single-tape Turing machine.

Complexity Relationships among Models

Theorem

Let $t(n)$ be a function, where $t(n) \geq n$. Then every $t(n)$ time mulitape Turing machine has an equivalent $\mathcal{O}(t^2(n))$ time single-tape Turing machine.

Theorem

Let $t(n)$ be a function, where $t(n) \geq n$. Then every $t(n)$ time nondeterministic single-tape Turing machine has an equivalent $2^{\mathcal{O}(t(n))}$ time deterministic single-tape Turing machine.

The Class P and NP

P is the class of languages that are decidable in polynomial time on a deterministic single-tape Turing machine. In other words,

$$\mathrm{P} = \bigcup_k \mathrm{TIME}(n^k).$$

The Class P and NP

P is the class of languages that are decidable in polynomial time on a deterministic single-tape Turing machine. In other words,

$$\mathrm{P} = \bigcup_k \mathrm{TIME}(n^k).$$

A ***verifier*** for a language A is an algorithm V, where

$$A = \{w \mid V \text{ accepts } \langle w, c \rangle \text{ for some string } c\}.$$

We measure the time of a verifier only in terms of the length of w, so a ***polynomial time verifier*** runs in polynomial time in the length of w. A language A is ***polynomially verifiable*** if it has a polynomial time verifier.

The Class P and NP

P is the class of languages that are decidable in polynomial time on a deterministic single-tape Turing machine. In other words,

$$\mathrm{P} = \bigcup_k \mathrm{TIME}(n^k).$$

A ***verifier*** for a language A is an algorithm V, where

$$A = \{w \mid V \text{ accepts } \langle w, c\rangle \text{ for some string } c\}.$$

We measure the time of a verifier only in terms of the length of w, so a ***polynomial time verifier*** runs in polynomial time in the length of w. A language A is ***polynomially verifiable*** if it has a polynomial time verifier.

NP is the class of languages that have polynomial time verifiers.

Examples

- HAMPATH=$\{\langle G, s, t\rangle | G$ is a directed graph with a Hamiltonian path from s to $t\}$ is in NP;

Examples

- HAMPATH=$\{\langle G, s, t\rangle | G$ is a directed graph with a Hamiltonian path from s to $t\}$ is in NP;
- $COMPOSITES = \{x | x = pq, \text{ for integers } p, q > 1\}$ is in NP;

Examples

- HAMPATH=$\{\langle G, s, t\rangle | G$ is a directed graph with a Hamiltonian path from s to $t\}$ is in NP;
- $COMPOSITES = \{x | x = pq,$ for integers $p, q > 1\}$ is in NP;
- $CLIQUE = \{\langle G, k\rangle | G$ is an undirected graph with a k-clique$\}$ is in NP;

Examples

- HAMPATH=$\{\langle G, s, t\rangle | G$ is a directed graph with a Hamiltonian path from s to $t\}$ is in NP;
- $COMPOSITES = \{x | x = pq, \text{ for integers } p, q > 1\}$ is in NP;
- $CLIQUE = \{\langle G, k\rangle | G \text{ is an undirected graph with a } k\text{-clique}\}$ is in NP;
- $SUBSET\text{-}SUM = \{\langle S, t\rangle | S = \{x_1, \ldots, x_k\} \text{ and for some } \{y_1, \ldots, y_l\} \subseteq \{x_1, \ldots, x_k\}, \text{ we have } \sum_{y_i} = t\}$ is in NP.

An Equivalent Definition of **NP**

Theorem

$NP = \bigcup_k NTIME(n^k)$.

An Equivalent Definition of **NP**

Theorem

$NP = \bigcup_k NTIME(n^k)$.

Proof.

$N =$ "On input w of length n:

1. Nondeterministically select string c of length at most n^k.
2. Run V on input $\langle w, c \rangle$.
3. If V accepts, *accept*; otherwise, *reject*."

An Equivalent Definition of **NP**

Theorem

$NP = \bigcup_k NTIME(n^k)$.

Proof.

$N =$ "On input w of length n:

1. Nondeterministically select string c of length at most n^k.
2. Run V on input $\langle w, c\rangle$.
3. If V accepts, *accept*; otherwise, *reject*."

$V =$ "On input $\langle w, c\rangle$, where w and c are strings:

1. Simulate N on input w, treating each symbol of c as a description of the nondeterministic choice to make at each step (as in the proof of Theorem 3.16).
2. If this branch of N's computation accepts, *accept*; otherwise, *reject*."

$\square$

$P = NP$?

- $P \subseteq NP$;

$P = NP$?

- $P \subseteq NP$;
- To prove $P \neq NP$, we need to show that there exists a problem in NP which can't be solved in polynomial time;

$P = NP$?

- $P \subseteq NP$;
- To prove $P \neq NP$, we need to show that there exists a problem in NP which can't be solved in polynomial time;
- To prove $P = NP$, we need to show that all the problems in NP can be solved in polynomial time.

Polynomial Time Reducibility

A function $f : \Sigma^* \longrightarrow \Sigma^*$ is a ***polynomial time computable function*** if some polynomial time Turing machine M exists that halts with just $f(w)$ on its tape, when started on any input w.

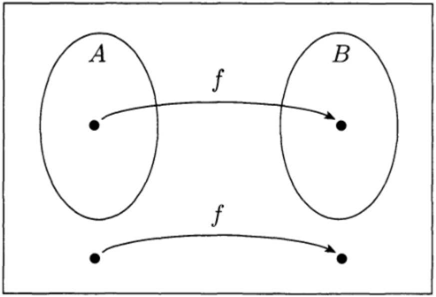
Polynomial Time Reducibility

A function $f: \Sigma^* \longrightarrow \Sigma^*$ is a ***polynomial time computable function*** if some polynomial time Turing machine M exists that halts with just $f(w)$ on its tape, when started on any input w.

Language A is ***polynomial time mapping reducible***,[1] or simply ***polynomial time reducible***, to language B, written $A \leq_P B$, if a polynomial time computable function $f: \Sigma^* \longrightarrow \Sigma^*$ exists, where for every w,

$$w \in A \iff f(w) \in B.$$

The function f is called the ***polynomial time reduction*** of A to B.



NP-Completeness

Theorem

If $A \leq_P B$ and $B \in P$, then $A \in P$.

NP-Completeness

Theorem

If $A \leq_P B$ *and* $B \in P$*, then* $A \in P$*.*

Corollary

If $A \leq_P B$ *and* $A \notin P$*, then* $B \notin P$*.*

NP-Completeness

Theorem

If $A \leq_P B$ and $B \in P$, then $A \in P$.

Corollary

If $A \leq_P B$ and $A \notin P$, then $B \notin P$.

A language A is NP-completeness if

1. $A \in NP$;

NP-Completeness

Theorem

If $A \leq_P B$ and $B \in P$, then $A \in P$.

Corollary

If $A \leq_P B$ and $A \notin P$, then $B \notin P$.

A language A is NP-completeness if

1. $A \in NP$;
2. $\forall A' \in NP$, $A' \leq_{\mathrm{P}} A$.

NP-Completeness

Theorem
If $A \leq_P B$ and $B \in P$, then $A \in P$.

Corollary
If $A \leq_P B$ and $A \notin P$, then $B \notin P$.

A language A is NP-completeness if

1. $A \in NP$;
2. $\forall A' \in NP$, $A' \leq_{\mathrm{P}} A$.

Theorem
If B is NP-complete and $B \leq_P C$ for $C \in NP$, then C is NP-complete.

The History of NP-completeness

- The first NP-complete problem was given by Stephen Cook and Leonid Levin independently in 1971. Particularly, Cook proved that the Boolean satisfiability problem is NP-complete;

The History of *NP*-completeness

- The first *NP*-complete problem was given by Stephen Cook and Leonid Levin independently in 1971. Particularly, Cook proved that the Boolean satisfiability problem is *NP*-complete;
- In 1972, Richard Karp showed 21 diverse combinatorial and graph theoretical problems, each infamous for its computational intractability, are *NP*-complete;

The History of *NP*-completeness

- The first *NP*-complete problem was given by Stephen Cook and Leonid Levin independently in 1971. Particularly, Cook proved that the Boolean satisfiability problem is *NP*-complete;
- In 1972, Richard Karp showed 21 diverse combinatorial and graph theoretical problems, each infamous for its computational intractability, are *NP*-complete;
- Garey and Johnson's 1979 book, "Computers and Intractability: A Guide to NP-Completeness" collected much more *NP*-complete problems.