Time complexity: Classes P and NP

204213 Theory of Computation

Jittat Fakcharoenphol

Kasetsart University

February 17, 2009

Outline

Terminology

- worst-case analysis, average-case analysis
- running time, time complexity
- asymptotic notations: big-O, little-O

Time complexity class

Definition

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

Multitape TM's and single-tape TM's

Theorem 1

Let $t(n)$ be a function, where $t(n) \geq n$. Then every $t(n)$ time multitape TM has a equivalent $O(t^2(n))$-time single-tape TM.

Proof (sketch)

Proof (sketch)

- Let M be a k-tape TM. We'll simulate M on a single-tape TM S.
- The contents of the other tape (not input tape) have total length of $O(t(n))$. (why?)
- Thus, time to simulate each step in M is $O(n) + O(t(n))$.
- There are $O(t(n))$ steps; thus the running time is $O(t(n) \times (n + t(n)) = O(t^2(n))$, since $t(n) > n$.

Nondeterministic TM's

Definition

The **running time of a nondeterministic TM N** is a function $f : \mathcal{N} \to \mathcal{N}$, where $f(n)$ is the maximum number of steops that N uses on any branch of its computation on any input of length n.

Nondeterministic TM's

Theorem 2

Let $t(n)$ be a function, where $t(n) > n$. Then, every $t(n)$-time nondeterministic TM has an equivalent $2^{O(t(n))}$-time deterministic TM.

Differences in models

- At most polynomial difference between multi-tape TM's and single-tape TM's.
- At most exponential difference between nondeterministic TM's and deterministic TM's.

Class P

- We see that different models of computation have different "power" in terms of efficiency (or, running time).

Class P

- We see that different models of computation have different "power" in terms of efficiency (or, running time).
- They also vary a lot.

Class P

- We see that different models of computation have different "power" in terms of efficiency (or, running time).
- They also vary a lot. So, it doesn't really make sense to say just "they are equivalence".

Class P

- We see that different models of computation have different "power" in terms of efficiency (or, running time).
- They also vary a lot. So, it doesn't really make sense to say just "they are equivalence".
- Since there are many models of TM's, can we group them? And how?

- Single-tape deterministic TM's
- Multi-tape deterministic TM's
- Nondeterministic TM's

- Single-tape deterministic TM's
- Multi-tape deterministic TM's
- Nondeterministic TM's
- Real computers

Grouping by relationship of power

- Single-tape deterministic TM's, Multi-tape deterministic TM's and Real computers
- Nondeterministic TM's

Grouping by relationship of power

- Single-tape deterministic TM's, Multi-tape deterministic TM's and Real computers
- Nondeterministic TM's
  - Consider a brute-force search algorithm.

Grouping by relationship of power

- Single-tape deterministic TM's, Multi-tape deterministic TM's and Real computers
- Nondeterministic TM's
  - Consider a brute-force search algorithm. How fast is it?

Definition of class **P**

Definition

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

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

Practicality

Are brute-force algorithms practical?

Practicality

Are brute-force algorithms practical?
Can we avoid them?

Problems in **P**: Path

$$PATH = \{\langle G, s, t\rangle \mid G \text{ is a directed graph that has a directed path from } s \text{ to } t\}.$$

Theorem 3

$PATH \in \mathbf{P}$

Problems in **P**: Relprime

$$RELPRIME = \{\langle x, y\rangle \mid x \text{ and } y \text{ are relatively prime}\}.$$

Theorem 4

$RELPRIME \in \mathbf{P}$

Search problems

In many cases, we **do not** know how to avoid brute-force search algorithms.

Hamiltonian path

- A Hamiltonian path is a path that goes through each node exacly once.
- Let

$$HAMPATH = \{\langle G, s, t\rangle \mid G \text{ is a directed graph}$$

$$\text{that has a directed Hamiltonian path from } s \text{ to } t\}.$$

- Can you decide *HAMPATH* quickly?

Similarity between brute-force algorithms

- Brute-force algorithms for *PATH* and *HAMPATH* are quite similar.

Similarity between brute-force algorithms

- Brute-force algorithms for *PATH* and *HAMPATH* are quite similar.
- They generate possible solutions, and check.

Similarity between brute-force algorithms

- Brute-force algorithms for *PATH* and *HAMPATH* are quite similar.
- They generate possible solutions, and check.
  - The checking procedures usually takes the input and the "candidate" output, and determines if that candidate is indeed the solution.
  - How fast are those checking procedures?

Similarity between brute-force algorithms

- Brute-force algorithms for *PATH* and *HAMPATH* are quite similar.
- They generate possible solutions, and check.
    - The checking procedures usually takes the input and the "candidate" output, and determines if that candidate is indeed the solution.
    - How fast are those checking procedures?
    - How can they be fast?

Similarity between brute-force algorithms

- Brute-force algorithms for *PATH* and *HAMPATH* are quite similar.
- They generate possible solutions, and check.
    - The checking procedures usually takes the input and the "candidate" output, and determines if that candidate is indeed the solution.
    - How fast are those checking procedures?
    - How can they be fast? They just **check!**.

Similarity between brute-force algorithms

- Brute-force algorithms for *PATH* and *HAMPATH* are quite similar.
- They generate possible solutions, and check.
  - The checking procedures usually takes the input and the "candidate" output, and determines if that candidate is indeed the solution.
  - How fast are those checking procedures?
  - How can they be fast? They just **check!**. They also have hints.

Let's abstract those checking procedures out.

Verifiers

Definition

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

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

Verifiers

Definition

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

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

- The running time of A is measured in terms of only w. We call A a **polynomial time verifier** if A runs in time polynomial in the length of w.

Verifiers

Definition

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

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

- The running time of A is measured in terms of only w. We call A a **polynomial time verifier** if A runs in time polynomial in the length of w.
- A language A is **polynomially verifiable** if it has a polynomial time verifier.

Verifiers

Definition

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

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

- The running time of A is measured in terms of only w. We call A a **polynomial time verifier** if A runs in time polynomial in the length of w.
- A language A is **polynomially verifiable** if it has a polynomial time verifier.

Notes

- We call c a **certificate**.
- For polynomial time verifiers, the certificates can only have polynomial-length (in terms of the length of w).

Class **NP**

Definition

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