NP-completeness

204213 Theory of Computation

Jittat Fakcharoenphol

Kasetsart University

February 17, 2009

Outline

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.

NP problems

- To show that a problem is in NP, we have to show that it has a polynomial verifier.

SAT

- A boolean formula is satisfiable if there is an assignment to its variables that makes it true.
- $SAT = \{\langle\phi\rangle \mid \phi$ is a boolean formula and ϕ is satisfiable$\}$.
- $SAT \in \mathrm{N}P$.

SAT

- A boolean formula is satisfiable if there is an assignment to its variables that makes it true.
- $SAT = \{\langle\phi\rangle \mid \phi \text{ is a boolean formula and } \phi \text{ is satisfiable}\}$.
- $SAT \in \mathrm{N}P$.
  - Can you provide a verifier for SAT?

Are problems in NP hard to solve?

Are problems in NP hard to solve?

No, not all of them.

Is there any?

Is there any?

Not sure; depending on what you mean by “hard”.

Reduction

- How can we compare problem hardness?
- If we can show that problem A reduces to problem B, we establish that problem A can be not harder than problem B.
- Please be careful on how you perform the **reduction**.

Polynomial time reduction

- The reduction step can't be too "complicated".

Polynomial time reduction

- The reduction step can't be too "complicated".
- If we want to show undecidability, the reduction step can't be undecidable.

Polynomial time reduction

- The reduction step can't be too "complicated".
- If we want to show undecidability, the reduction step can't be undecidable.
- In this case, we want to relate two problems in terms of polynomial time solvability; thus, the reduction step should run in polynomial time.

Polynomial time reduction: formal definition

- We say that problem A is **polynomial-time reducible** to problem B if there's a polynomial-time algorithm f such that
  - for all $w \in A$, $f(w) \in B$, and
  - for all $w \notin A$, $f(w) \notin B$.

Polynomial time reduction: formal definition

- We say that problem A is **polynomial-time reducible** to problem B if there's a polynomial-time algorithm f such that
  - for all $w \in A$, $f(w) \in B$, and
  - for all $w \notin A$, $f(w) \notin B$.
- If that's the case, we write $A \leq_P B$.

What does $A \leq_P B$ imply...

...in terms of solvability of A and B?

What does $A \leq_P B$ imply...

...in terms of solvability of A and B?

If B is solvable in polynomial time, A is as well.

What does $A \leq_P B$ imply...

...in terms of solvability of A and B?
If B is solvable in polynomial time, A is as well.
Can you prove that? (Hint: use f.)

Hardest problem in NP

- We can compare the hardness of two problems in NP:
  - if $A \leq_P B$, the A can't be harder than B.
- Can you define "the hardest problems in NP"?

Hardest problem in NP

- We can compare the hardness of two problems in NP:
  - if $A \leq_P B$, the A can't be harder than B.
- Can you define "the hardest problems in NP"?
- Two properties:
  - must be in NP.
  - no problems in NP are harder.

Hardest problem in NP

- We can compare the hardness of two problems in NP:
  - if $A \leq_P B$, the A can't be harder than B.
- Can you define "the hardest problems in NP"?
- Two properties:
  - must be in NP.
  - no problems in NP are harder.
- We even have a name for them: **NP-complete problems**.

Hardest problem in NP: definition

NP-complete

A problem A is NP-complete iff

- $A \in \mathrm{NP}$, and
- for every problem $B \in \mathrm{NP}$, $B \leq_P A$.

Cook-Levin theorem

Theorem 1

SAT is NP-complete

For me, it is amazing that there even exists one NP-complete problem.

Proving NP-completeness

- To show that problem A is NP-complete, we have to show that it meets both requirements.
- Extremely hard to show the 2nd requirement directly.
- We are fortunate that we have Cook-Levin's theorem, can just show that $SAT \leq_P A$.

Proving NP-completeness

- To show that problem A is NP-complete, we have to show that it meets both requirements.
- Extremely hard to show the 2nd requirement directly.
- We are fortunate that we have Cook-Levin's theorem, can just show that $SAT \leq_P A$. why?

Theorem 2

For any NP problem A and an NP-complete problem B, if

$$B \leq_P A,$$

then A is NP-complete

Other NP-complte problems

- 3SAT
- Independent Set
- Clique
- Vertex Cover