Unit 11
Lower Bound and NP-Completeness

T.H. Cormen et al., "Introduction to Algorithms", 3rd ed., Chapter 34

Anany Levitin, "Introduction to The Design & Analysis of Algorithms", Chapter10, 2003.

Computational Complexity

- In previous units, we mainly analyze the complexities (especially, time complexity) of algorithms.
- Here, we will study the ***complexities of problems***.
- We call $f(n)$ a (or ***asymptotically***) ***lower bound*** for a problem if ***for any*** algorithms that solving the problem, its worst case execution time is $f(n)$ (or $\Omega(f(n))$).
- On the other hand, every algorithm provides an ***upper bound*** for the problem it solves.

Computational Complexity (續)

- ***Goal*** **:** For a given problem, determine a lower bound of $\Omega(f(n))$ and develop a $\Theta(f(n))$ algorithm for the problem.
- Once we have done this, then except for improving the constant, we ***can not improve*** on the algorithm ***any further***.
- Such an algorithm is called an (***asymptotically***) ***optimal algorithm*** for the problem.

Problems with Trivial Lower Bounds

Problems	Lower bound	Tightness
Generating all permutations	$\Omega(n!)$	**yes**
Evaluating a polynomial	$\Omega(n)$	**yes** **(Horner's rule p.41)**
Multiplication of 2 $n \times n$ matrices	$\Omega(n^2)$	unknown
Multiplication of 2 n-digit integers	$\Omega(n)$	unknown
Finding max among n unsorted numbers	$n-1$	**yes**
TSP	$\Omega(n)$	unlikely

Problems with Tight Lower Bounds

Problems	Lower bound
Sorting	$\Omega(n\lg n)$
Searching in a sorted array	$\Omega(\lg n)$
Element uniqueness	$\Omega(n\lg n)$
Merging two sorted arrays of size n	$2n-1$ (p.208 Problem 8-6)
Finding max & min among n unsorted numbers	$\lceil 3n/2 \rceil - 2$ (p.215 Ex. 9.1-2 or p.7 Unit 4)

Based on the assumption that the only basic operation used is comparison.

Using Problem Reduction to Establish Lower Bound

- If we can show that problem X can be reduced to problem Y s.t. the transformation time is no larger than a lower bound of problem X. Then the bound is also a lower bound of problem Y.
- Examples
  - $x \cdot y = [(x + y)^2 - (x - y)^2]/4$
  - *Euclidean MST problem* & sorting.

演算法中常見之函數

Big-Oh form	Name
$\Theta(1)$	Constant
$\Theta(\lg n)$	Logarithmic
$\Theta(n)$	Linear
$\Theta(n \lg n)$	$n \log n$
$\Theta(n^2)$	Quadratic, Square
$\Theta(n^3)$	Cubic
$\Theta(n^k)$, k : constant	Polynomial
$\Theta(c^n)$, c : constant > 1	Exponential
$\Theta(n!)$	Factorial

奇怪的界線

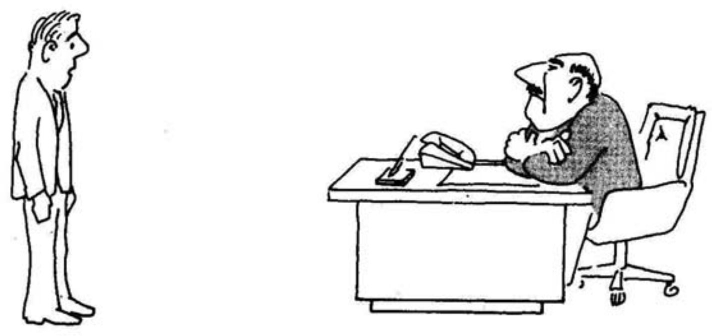
Tractable and Intractable Problems

- A problem is called ***tractable*** (easy) if it can be solved by a polynomial-time algorithm.
- A problem is called ***intractable*** (difficult) if it is impossible to solve it with a polynomial-time algorithm or a lower bound of the problem is super-polynomial.

Intractability and NP-Completeness

- There are three general categories of problems :
  1. Tractable problems (denoted as **P**).
  2. Intractable problems.
  3. Problems that ***have not been proven*** to be intractable or tractable.
- ***Most problems*** in computer science seem to ***fall into either the first or third*** category.
- An interesting class of problems, called the ***NP-complete*** problems, is in the third category.

A Cartoon about NP-Completeness (part 1)



"I can't find an efficient algorithm, I guess I'm just too dumb."

From : Michael R. Garey and David S. Johnson "Computer and Intractability : A Guide to the theory of NP-Completeness. 1979

A Cartoon about NP-Completeness (part 2)

"I can't find an efficient algorithm, because no such algorithm is possible!"

A Cartoon about NP-Completeness (part 3)

"I can't find an efficient algorithm, but neither can all these famous people."

A Description of NP-Completeness

- No one has ever found algorithms for any of NP-complete problems whose worst-case time complexities are better than exponential, but no one has ever proved that such algorithms are not possible.
- However, if we can solve any single NP-complete problem in polynomial time, then it implies that ***every*** NP-complete problem can be solved in polynomial time.

Abstract & Decision Problems

- An ***abstract problem*** is defined as a binary relation (a subset of $\boldsymbol{I} \times \boldsymbol{O}$).
- For simplicity, the theory of NP restricts attention to ***decision problems***, those having only a yes/no solution.
- Many problems have their ***related decision problems*** including optimization problems, e.g. TSP, SOS, 0/1 knapsack, HAM-CYCLE etc.
- If a problem is easy, its related decision problem is easy as well; in general, the converse is also true.

Encoding and Concrete Problems

- Encoding: ***e* : *I* → Σ*** (|Σ|≥2; here we use **Σ = {0,1}**) where ***I*** is the set of instances of a problem, **Σ* = {0,1}* = {ε, 0, 1, 00, 01, 10, 11, 000,… }**
- A problem whose instance set is the set of binary strings is called a ***concrete problem***.
- A concrete problem is **in the class P** if it can be solved by an algorithm in $O(n^k)$ time for any instance i of length $n = |i|$.
- Instances are assumed to be **encoded in a reasonable, concise fashion**.

A Formal-Language Framework

- We call any $L \subseteq \Sigma^*$ is a ***language*** over Σ.
- Any decision problem $Q \leftrightarrow$ a language L over Σ.

  L = the set of yes-instances of Q.
- For example:

  HAM-CYCLE = {⟨*G*⟩ : *G* is a graph that contains a hamiltonian cycle (a simple cycle that contains each vertex in *G*)}

  Here, ⟨*G*⟩ denotes an encoded binary string of *G*.

Operations on Languages

- Ordinary set operations: ***union***, ***intersection***.
- We define the ***complement*** of L by $L^c = \Sigma^* - L$
- The ***concatenation*** of two languages L_1 and L_2 is
  $L = L_1L_2 = \{x_1 x_2 \mid x_1 \in L_1, x_2 \in L_2\}$.
- The ***closure*** or ***Kleene star*** of L is
  $L^* = \{\varepsilon\} \cup L \cup L^2 \cup L^3 \cup \ldots$;
  where $L^2 = LL$, $L^3 = LLL$, …

Exercise : The class P is closed under union, intersection, complement, concatenation and Kleene star.

The Class NP

- The name "**NP**" stands for "the class of problems which can be ***accepted*** by a ***nondeterministic Turing machine*** in ***polynomial*** time."
- There are several (at least 3) different but equivalent definitions in the literature.
- We use the version of ***polynomial-time verification***.

NP & Polynomial-Time Verification

- A problem is said to be in the class **NP** if there exist a ***polynomial-time verification algorithm A*** and constant ***c*** such that for any ***yes-instance x***, we can find a ***certificate y*** with $|y|=O(|x|^c)$ and $A(x, y) = 1$.
- Examples : TSP, SOS, 0/1 knapsack, HAM-CYCLE, …
- Properties : $\mathbf{P} \subseteq \mathbf{NP}$ (How about $\mathbf{P} = \mathbf{NP}$ **?**)

Standard NP Problems

- Many decision problems has the form:
  Given ***x***, is there a ***y*** such that $p(y) = 1$?
- Many optimization problems have related decision problems of the form:
  Given $\langle x, k\rangle$, $\exists\, y$ s.t. $p(y) = 1$ and $c(y) \le$ (or $\ge$) k?
- If we can show that $|y|=O(|x|^c)$, and $p(y)$ & $c(y)$ are polynomial time computable, then the problem is an NP problem.

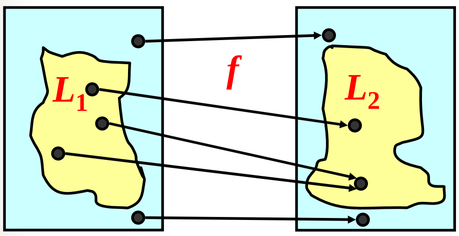
P = NP ?

- It is also called the P versus NP problem.
- In a detective TV series it is said : “It asks whether every problem whose solution can be quickly verified by a computer can also be quickly solved by a computer.”
- Whoever could solve this problem would receive $1 million in prize money (founded by Clay Mathematics Institute.)
- Many researchers believe that P ≠ NP.

Reducibility

- A problem L_1 is ***polynomial-time reducible*** to a problem L_2, written $L_1 \leq_P L_2$, if there exists a ***polynomial-time computable function f*** such that for any instance x :

x is a yes-instance of L_1 if and only if $f(x)$ is a yes-instance of L_2

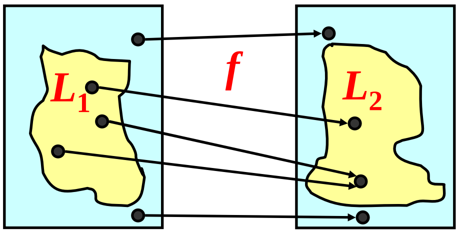


Reducibility and Hardness of Problems

If $L_1 \leq_P L_2$, then $L_2 \in \mathbf{P} \Rightarrow L_1 \in \mathbf{P}$

Proof :Assume that A_2 is a polynomial time algorithm for L_2, we can find a polynomial time algorithm A_1 for L_1 as :

```
A1(x)
{
   y = f(x);
   return A2(y);
}
```



NP-Completeness

- A problem L is ***NP-complete*** if
  1. $L \in \mathbf{NP}$, and
  2. $L' \leq_P L$ for every $L' \in \mathbf{NP}$ (***NP-hard***)
- For any NP-complete problem L, if we can show that $L \in \mathbf{P}$, then $\mathbf{P} = \mathbf{NP}$ (but most computer scientists believe that $\mathbf{P} \neq \mathbf{NP}$).
- $L_1 \leq_P L_2$ and $L_2 \leq_P L_3 \Rightarrow L_1 \leq_P L_3$
- If $L_2 \in \mathbf{NP}$, L_1 is **NP-complete** and $L_1 \leq_P L_2$, then L_2 is also **NP-complete**.

Formula Satisfiability

- The ***Formula satisfiability*** (**SAT**) problem asks whether there exists a variable assignment that causes the given ***boolean formula*** to evaluate to 1.
- For example consider the formula :

  $\phi = ((x_1 \rightarrow x_2) \vee \neg((\neg x_1 \leftrightarrow x_3) \vee x_4)) \wedge \neg x_2$

  and the assignment : $x_1 = 0, x_2 = 0, x_3 = 1, x_4 = 1$

  $\phi = ((0 \rightarrow 0) \vee \neg((\neg 0 \leftrightarrow 1) \vee 1)) \wedge \neg 0$

  $= (1 \vee \neg(1 \vee 1)) \wedge 1 = (1 \vee 0) \wedge 1 = 1$

  $\therefore$ ϕ is a yes-instance of **SAT**

Cook's Theorem

- In 1971, Cook showed that the problem **SAT** is **NP-complete**.
- With SAT we can show **thousands of NP-complete problems** including TSP, SOS, 0/1 knapsack, and HAM-CYCLE (all in their decision versions).

CNF-SAT

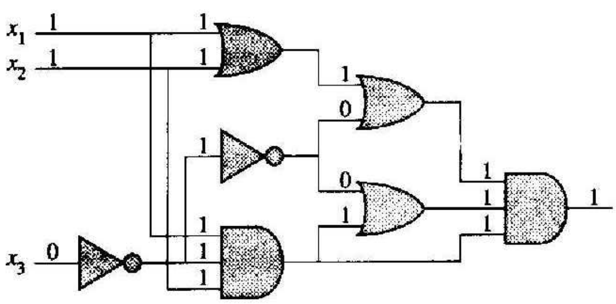
- A ***literal*** in a boolean formula is an occurrence of a variable or its negation. A boolean formula is in ***conjunctive normal form*** (**CNF**) if it is expressed as an AND of ***clauses***, each of which is the OR of one or more literals, e.g.

  $(x_1 \lor \neg x_1 \lor \neg x_2) \land (x_2 \lor x_3) \land (x_2 \lor \neg x_3 \lor \neg x_4)$

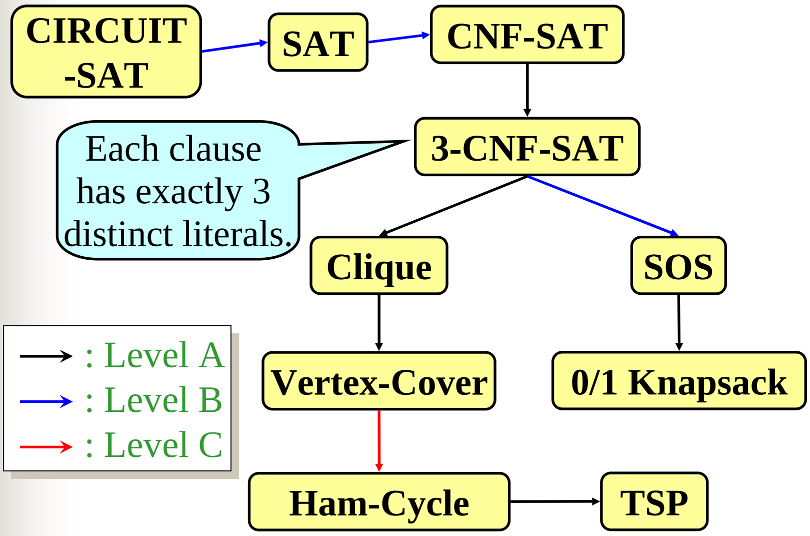
- The ***CNF satisfiability*** problem (**CNF-SAT**) is : Given a boolean formula in CNF, whether there exists a variable assignment that causes the formula to evaluate to 1.

Circuit Satisfiability

- The ***Circuit satisfiability*** problem (**CIRCIT-SAT**) is : Given a boolean combinational circuit composed of AND, OR, and NOT gates, is it satisfiable?

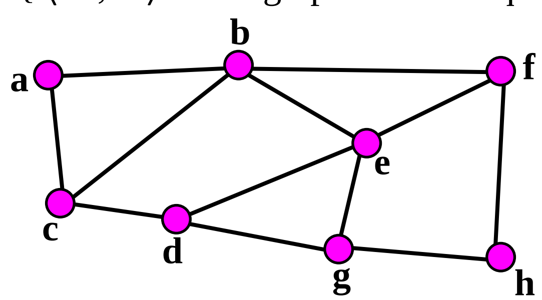


NP-Complete Problems



The Clique Problem

- A ***clique*** in a graph ***G=(V, E)***, is a complete subgraph of ***G***. The ***size*** of a clique is the number of vertices it contains.
- **CLIQUE** = { 〈 ***G***, ***k*** 〉 : ***G*** is graph with a clique of size ***k***}

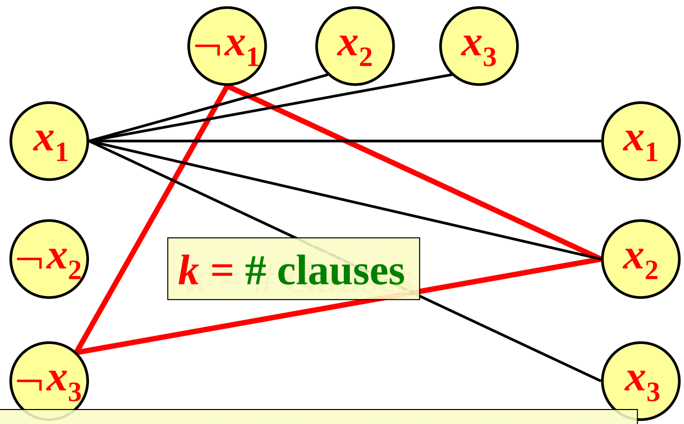


3-CNF-SAT $\leq_P$ CLIQUE

$\phi = (x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee x_3)$

⇩

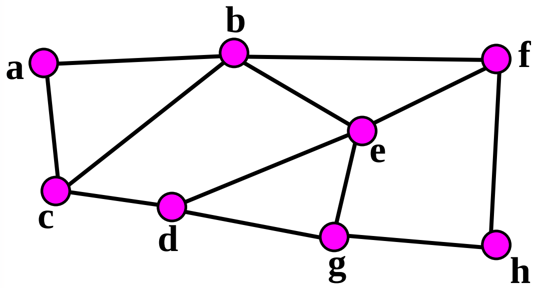
$\langle G, k \rangle$



$\langle \phi \rangle \in$ 3-CNF-SAT **iff.** $\langle G, k \rangle \in$ CLIQUE

The Vertex-Cover Problem

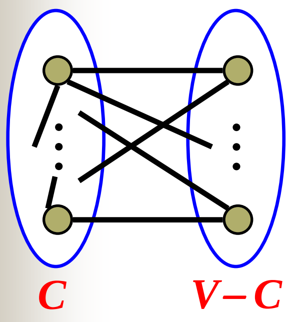
- A ***vertex cover*** of a graph $G=(V, E)$, is a subset C of V such that if $uv \in E$, then $u \in C$ or $v \in C$ (or both). The size of C is the number of vertices in it.
- **V-COVER** $= \{\langle G, k \rangle : G$ has a vertex cover of size $k\}$



CLIQUE $\leq_P$ V-COVER

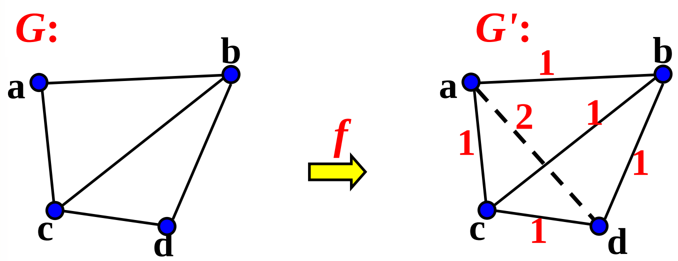
$<G, k> \;\Rightarrow\; <G^c, |V| - k>$

$<G, k> \in$ CLIQUE **iff.**
$<G^c, |V| - k> \in$ V-COVER



$V - C$ is an ***independent set***

HAM-CYCLE $\leq_P$ TSP



G has a Hamiltonian cycle **iff.**
G' has a tour of length $\leq n$.

SOS $\le_P$ 0/1 Knapsack

$x = \langle s_1, s_2, \ldots, s_n, K \rangle \xrightarrow{f} f(x) = \langle w_1, w_2, \ldots, w_n, W, p_1, p_2, \ldots, p_n, k \rangle$

Let $w_i = p_i = s_i$ for $1 \le i \le n$
and $W = k = K$

x is a yes-instance iff.
$f(x)$ is a yes-instance

How to Handle Intractable Problems

(~)E: (non-)Exact algorithms

- Branch-and-bound (E)
- Heuristic algorithms (~E)
- Randomized algorithms : simulated annealing (~E), genetic algorithms (~E), probabilistic algorithms (E or ~E)
- Approximation algorithms (~E)
- Fixed-parameter algorithms (E)
- Other models of computation : quantum, DNA, parallel, neural nets…
- …