Design and Analysis of Algorithms
Part V: Dealing with Hard Problems
Lecture 32: Problem Classes: P, NP, NP-Completeness


Ke Xu and Yongxin Tong
（许可 与 童咏昕）
School of CSE, Beihang University

Outline

- Introduction to Part V
- Problem Classes: P and NP
  - Input size of a problem
  - Optimization problems vs Decision problems
  - The class P and class NP
- Introduction to NP-Completeness (NPC)
  - Polynomial-time reductions
  - The class NPC
  - NP-Complete problems
  - Optimization vs. Decision problems about NPC
- Summary

Introduction

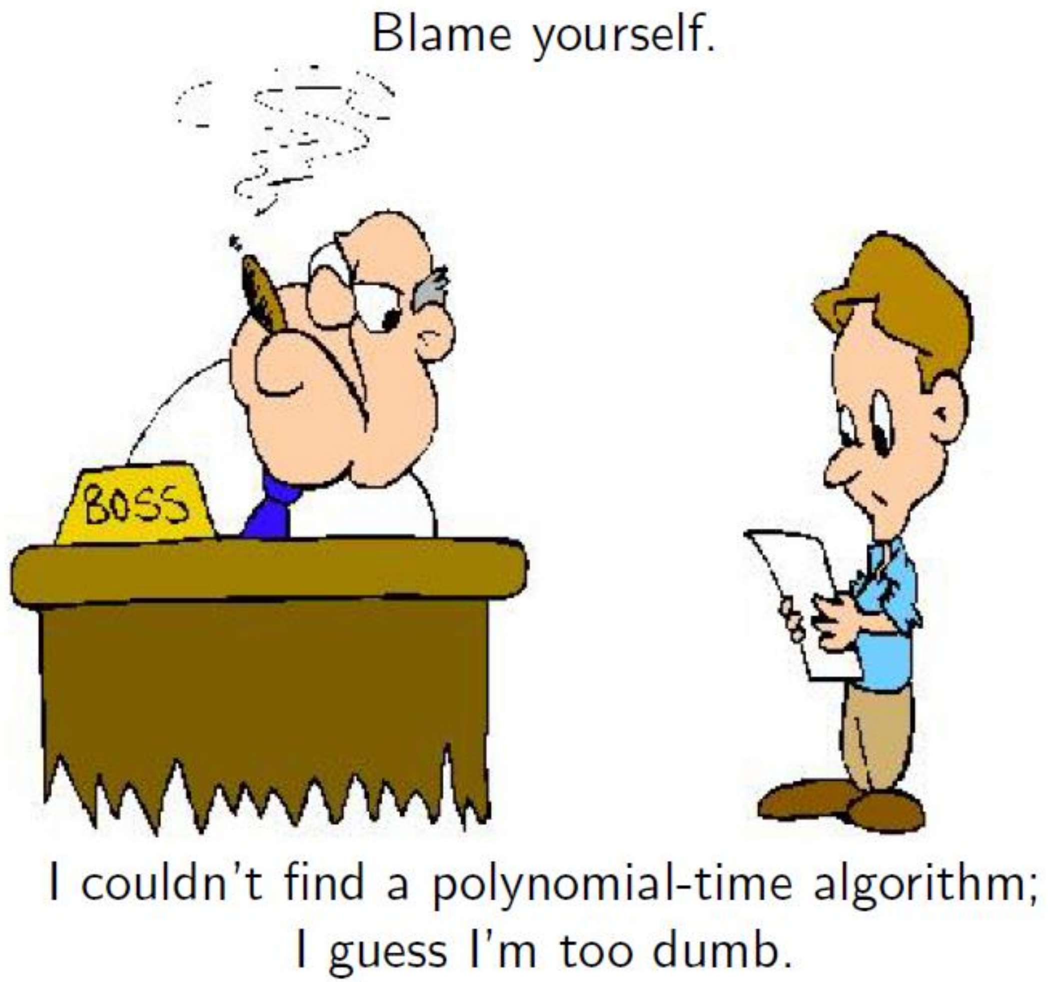
- So far: techniques for designing efficient algorithms: divide-and-conquer, dynamic-programming, greedy-algorithms.

- What happens if you can't find an efficient algorithm for a given problem?

Introduction

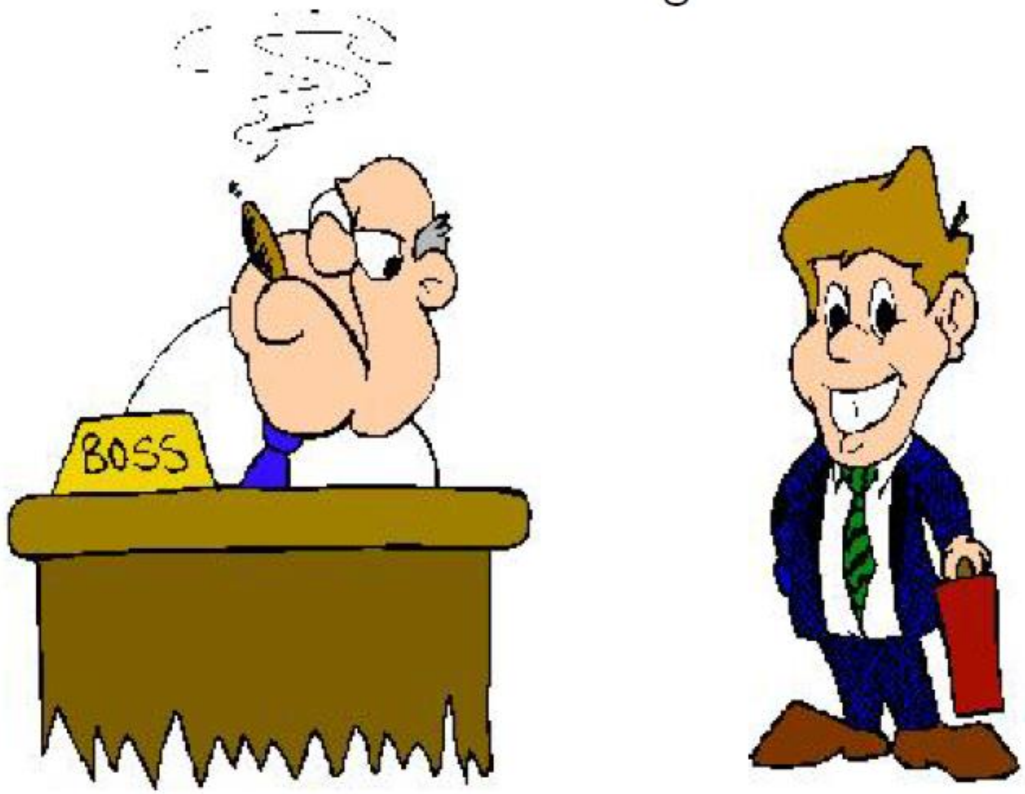
- Showing that a problem has an efficient algorithm is, relatively, easy.
  - “All’ that is needed is to demonstrate an algorithm.
- Proving that no efficient algorithm exists for a particular problem is difficult.

Introduction



Introduction

Show that no-efficient algorithm exists.



I couldn't find a polynomial-time algorithm,
because no such algorithm exists!

Introduction

- Showing that a problem has an efficient algorithm is, relatively, easy
  - “All that is needed is to demonstrate an algorithm.
- Proving that no efficient algorithm exists for a particular problem is difficult.

Question

How can we prove the non-existence of something?

- We will now learn about NP-Complete Problems, which provide us with a way to approach this question.

Introduction

- A very large class of thousands of practical problems for which it is not known if the problems have “efficient” solutions.
- It is known that if any one of the NP-complete problems has an efficient solution then all of the NP-complete problems have efficient solutions.
- Researchers have spent innumerable man-years trying to find efficient solutions to these problems and failed.
- So, NP-Complete problems are very likely to be hard.
- What do you do: prove that your problem is NP-complete.

Introduction

What do you actually do:

I couldn't find a polynomial-time algorithm,
but neither could all these other smart people!

Outline

- Introduction to Part V
- Problem Classes: P and NP
  - Input size of a problem
  - Optimization problems vs Decision problems
  - The class P and class NP
- Introduction to NP-Completeness (NPC)
  - Polynomial-time reductions
  - The class NPC
  - NP-Complete problems
  - Optimization vs. Decision problems about NPC
- Summary

Encoding the Inputs of Problems

- Complexity of a problem is measured w.r.t the size of input.
- In order to formally discuss how hard a problem is, we need to be much more formal than before about the input size of a problem.
- We will therefore spend some time now discussing how to encode the inputs of problems.

Example

Question

How do we encode a graph?

- A graph G may be represented by its adjacency matrix $A = [a_{ij}]$.
- G can then be encoded as a binary string of length n^2:

$$a_{11}\ldots a_{1n}a_{21}\ldots a_{2n}\ldots a_{n1}\ldots a_{nn}$$

- Given the binary string, the computer can count the number of bits and then determine n, the vertices, and the edges.

Remark: In general, the inputs of any problem can be encoded as binary strings.

The Input Sizes of Problems

- The input size of a problem may be defined in a number of ways.

Definition (Standard definition)

The input size of a problem is the minimum number of bits ($\{0, 1\}$) needed to encode the input of the problem.

- Remark: The exact input size s, (minimal number of bits) determined by an optimal encoding method, is hard to compute in most cases.
  - However, for the complexity problems we will study, we do not need to determine s exactly.
  - For most problems, it is sufficient to choose some natural, and (usually) simple, encoding and use the size s of this encoding.

Input Size Example: Composite

Example (Composite)

Given a positive integer n, are there integers $j, k > 1$ such that $n = jk$? (i.e., is n a composite number?)

Question

What is the input size of this problem?

- Any integer n > 0 can be represented in the binary number system as a string $a_0 a_1 \ldots a_k$ of length $\lceil \log_2(n + 1) \rceil$, because

$$n = \sum_{i=0}^{k} a_i 2^i \quad \text{where } k = \lceil \log_2(n+1) \rceil - 1$$

- Therefore, a natural measure of input size is $\lceil \log_2(n+1) \rceil$ (or just $\log_2 n$).

Input Size Example: Sorting

Example (Sorting)

Sort n integers $a_1, \ldots, a_n$

Question

What is the input size of this problem?

- Using fixed length encoding, we write a_i as a binary string of length

$$m = \lceil \log_2 \max(|a_i| + 1) \rceil.$$

- This coding gives an input size nm.

Outline

- Introduction to Part V
- Problem Classes: P and NP
  - Input size of a problem
  - Optimization problems vs Decision problems
  - The class P and class NP
- Introduction to NP-Completeness (NPC)
  - Polynomial-time reductions
  - The class NPC
  - NP-Complete problems
  - Optimization vs. Decision problems about NPC
- Summary

Decision Problems

Definition

A decision problem is a question that has two possible answers: yes and no.

- If L is the problem and x is the input, we will often write $x \in L$ to denote a yes answer and $x \notin L$ to denote a no answer.
- This notation comes from thinking of L as a language and asking whether x is in the language L (yes) or not (no).
- See CLRS, pp. 975-977 for more details.

Optimization Problems

Definition

An optimization problem requires an answer that is an optimal configuration.

- An optimization problem usually has a corresponding decision problem.
- Examples that we will see:
  - MST vs. Decision Spanning Tree (DST),
  - Knapsack vs. Decision Knapsack (DKnapsack),
  - SubSet Sum vs. Decision Subset Sum (DSubset Sum)

Decision Problems: MST

Optimization problem: Minimum Spanning Tree

Given a weighted graph G, find a minimum spanning tree (MST) of G.

Decision problem: Decision Spanning Tree (DST)

Given a weighted graph G and an integer k, does G have a spanning tree of weight at most k?

- The inputs are of the form (G, k).
- So we will write (G, k) $\in$ DST or (G, k) $\notin$ DST to denote, respectively, yes and no answers.

Decision Problems: Knapsack

- We have a knapsack of capacity W (a positive integer) and n objects with weights $w_1, \ldots, w_n$ and values $v_1, \ldots, v_n$, where v_i and w_i are positive integers.

Optimization problem: Knapsack

Find the largest value $\sum_{i \in T} v_i$ of any subset T that fits in the knapsack, that is, $\sum_{i \in T} w_i \leq W$.

Decision problem: Decision Knapsack (DKnapsack)

Given k, is there a subset of the objects that fits in the knapsack and has total value at least k?

Decision Problems: Subset Sum

- The input is a positive integer C and n objects whose values are positive integers $s_1, \ldots, s_n$.
  - For a more formal definition see CLRS, Section 34.5.5

Optimization problem: Subset Sum

Among subsets of the objects with sum at most C, what is the largest subset sum?

Decision problem: Decision Subset Sum (DSubset Sum)

Is there a subset of objects whose values add up to exactly C?

Optimization and Decision Problems

- For almost all optimization problems there exists a corresponding simpler decision problem.
- Given a subroutine for solving the optimization problem, solving the corresponding decision problem is usually trivial.

Example

If we know how to solve MST, we can solve DST which asks if there is an Spanning Tree with weight at most k.
How? First solve the MST problem and then check if the MST has cost $\leq k$. If it does, answer Yes. If it doesn't, answer No.

- Thus if we prove that a given decision problem is hard to solve efficiently, then it is obvious that the optimization problem must be (at least as) hard.

  Note: It will be more convenient to compare the 'hardness' of decision problems than of optimization problems (since all decision problems share the same form of output, either yes or no.)

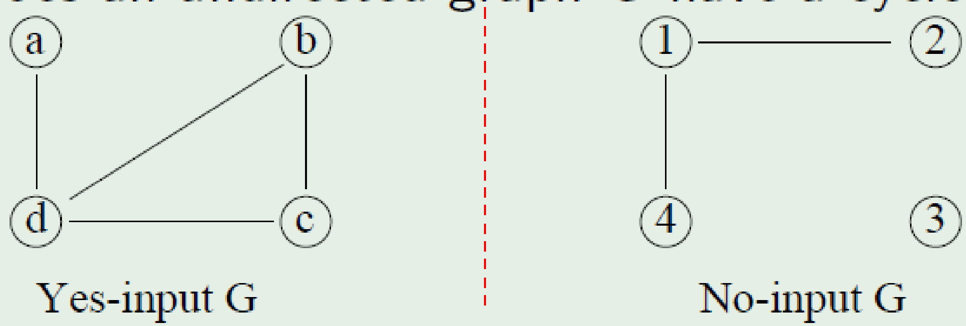
Decision Problems: Yes-Inputs ant No-Inputs

Definition

An instance of a decision problem is called a yes-input (respectively no-input) if the answer to the instance is yes (respectively no).

Example (CYC Problem)

Does an undirected graph G have a cycle?



Example (Decision Problem (TRIPLE))

Does a triple (n, e, t) of nonnegative integers satisfy $n - e = t$?

Yes-Inputs: $(9, 7, 2)$, $(20, 2, 18)$.

No-Inputs: $(10, 1, 2)$, $(20, 5, 18)$.

Outline

- Introduction to Part V
- Problem Classes: P and NP
  - Input size of a problem
  - Optimization problems vs Decision problems
  - The class P and class NP
- Introduction to NP-Completeness (NPC)
  - Polynomial-time reductions
  - The class NPC
  - NP-Complete problems
  - Optimization vs. Decision problems about NPC
- Summary

Complexity Classes

- The Theory of Complexity deals with
  - the classification of certain "decision problems" into several classes:
    - the class of "easy" problems,
    - the class of "hard" problems,
    - the class of "hardest" problems;
  - relations among the three classes;
  - properties of problems in the three classes.

Question

How to classify decision problems?

Answer: Use "polynomial-time algorithms."

Polynomial-Time Algorithms

Definition

An algorithm is polynomial-time if its running time is $O(n^k)$, where k is a constant independent of n, and n is the input size of the problem that the algorithm solves.

- Remark: Whether you use n or n^α (for fixed a > 0) as the input size, it will not affect the conclusion of whether an algorithm is polynomial time.
  - This explains why we introduced the concept of two functions being of the same type earlier on.
  - Using the definition of polynomial-time it is not necessary to fixate on the input size as being the exact minimum number of bits needed to encode the input!

Polynomial-Time Algorithms

Example

- The standard multiplication algorithm learned in school has time $O(m_1 m_2)$ where m_1 and m_2 are, respectively, the number of digits in the two integers.
- DFS has time $O(n + e)$.
- Kruskal's MST algorithm runs in time $O((e + n) \log n)$.

Nonpolynomial-Time Algorithms

Definition

An algorithm is non-polynomial-time if the running time is not $O(n^k)$ for any fixed $k \geq 0$.

- Let's return to the brute force algorithm for determining whether a positive integer N is a prime:
  - it checks, in time $\Theta((\log N)^2)$, whether K divides N for each K with $2 \leq K \leq N - 1$.
  - The complete algorithm therefore uses $\Theta(N(\log N)^2)$ time.
- Conclusion: The algorithm is nonpolynomial!

Question

Why?

The input size is $n = \log_2 N$, and so

$$\Theta(N(\log N)^2) = \Theta(2^n n^2).$$

Is Knapsack Problem Polynomial?

- Recall the problem. We have a knapsack of capacity W (a positive integer) and n objects with weights $w_1, \ldots, w_n$ and values $v_1, \ldots, v_n$, where vi and wi are positive integers.

Optimization problem

Find the largest value $\sum_{i \in T} v_i$ of any subset T that fits in the knapsack, that is, $\sum_{i \in T} w_i \leq W$.

Decision problem

Given k, is there a subset of the objects that fits in the knapsack and has total value at least k?

Question

In class we saw a $\Theta(nW)$ dynamic programming algorithm for solving the optimization version of Knapsack. Is this a polynomial algorithm?

Is Knapsack Problem Polynomial?

- Answer: No!
  - The size of the input is

$$size(I) = \log_2 W + \sum_i \log_2 w_i + \sum_i \log_2 v_i.$$

  - nW is not polynomial in size(I). Depending upon the values of the w_i and v_i , nW could even be exponential in size(I).

- It is unknown as to whether there exists a polynomial time algorithm for Knapsack.
  - In fact, Knapsack is a NP-Complete problem.

Polynomial- vs. Exponential-Time

- Exponential-time algorithms are impractical.

Example

To run an algorithm of time complexity 2^n for $n = 100$ on a computer which does 1 Terraoperation (10^{12} operations) per second: It takes $2^{100}/10^{12} \approx 10^{18.1}$ seconds $\approx 4 \cdot 10^{10}$ years.

- For the sake of our discussion of complexity classes Polynomial-time algorithms are ”practical”.
  - Note: in reality an $O(n^{20})$ algorithm is not really practical.

Polynomial-Time Solvable Problems

- Exponential-time algorithms are impractical.

Definition

A problem is solvable in polynomial time (or more simply, the problem is in polynomial time) if there exists an algorithm which solves the problem in polynomial time.

Example

The integer multiplication problem, and the cycle detection problem for undirected graphs.

- Remark: Polynomial-time solvable problems are also called tractable problems.

The Class P

Definition

The class $\mathcal{P}$ consists of all decision problems that are solvable in polynomial time. That is, there exists an algorithm that will decide in polynomial time if any given input is a yes-input or a no-input.

Question

How to prove that a decision problem is in $\mathcal{P}$?

Ans: You need to find a polynomial-time algorithm for this problem.

Question

How to prove that a decision problem is not in $\mathcal{P}$?

Ans: You need to prove there is no polynomial-time algorithm for this problem (much harder).

The Class P: An Example

Example problem

Is a given connected graph G a tree?

Claim

This problem is in $\mathcal{P}$.

Proof.

We need to show that this problem is solvable in polynomial time.

- We run DFS on G for cycle detection.
- If a back edge is seen, then output NO, and stop.
- Otherwise output YES.

Recall that the input size is $n + e$, and DFS has running time $O(n + e)$. So this algorithm is linear, and the problem is in $\mathcal{P}$. $\square$

The Class P: An Example

Example problem: DST

Given a weighted graph G and a parameter $k > 0$, does G have a spanning tree with total weight $\leq k$?

Claim

This problem is in $\mathcal{P}$.

Proof.

- Run Kruskal's algorithm and find a minimal spanning tree, T, of G.
- Calculate $w(T)$ the weight of T.
- If $k \geq w(T)$, answer Yes; otherwise, answer No.

Recall that Kruskal's algorithm runs in $O((e + n) \log n)$ time, so this is polynomial in the size of the input. $\square$

Certificates and Verifying Certificates

- We have already seen the class P. We are now almost ready to introduce the class NP.

- Observation: A decision problem is usually formulated as:

  Is there an object satisfying some conditions?

Certificates and Verifying Certificates

Definition

A Certificate is a specific object corresponding to a yes-input, such that it can be used to show that the input is indeed a yes-input.

- By definition, only yes-input needs a certificate (a no-input does not need to have a ’certificate’ to show it is a no-input).

- Verifying a certificate: Given a presumed yes-input and its corresponding certificate, by making use of the given certificate, we verify that the input is actually a yes-input.

The Class NP

Definition

The class $\mathcal{NP}$ consists of all decision problems such that, for each yes-input, there exists a certificate which allows one to verify in polynomial time that the input is indeed a yes-input.

- Remark: NP stands for "nondeterministic polynomial time". The class NP was originally studied in the context of nondeterminism, here we use an equivalent notion of verification.

COMPOSITE ∈ NP

COMPOSITE

Is a given positive integer n composite?

- For COMPOSITE, an yes-input is just the integer n that is composite.

Question (Certificate)

What is needed to show n (a presumed yes-input) is actually a yes-input? The 'object' needed is the certificate for COMPOSITE.

Ans: The certificate is an integer a (1 < a < n) with the property that it divides n.

Proof (Verifying a certificate).

- Given a certificate a, check whether a divides n.
- This can be done in time $O((\log_2 n)^2)$ (recall that input size is $\log_2 n$ so this is polynomial in input size).
- Hence, COMPOSITE $\in \mathcal{NP}$.

$\square$

DSubsetSum $\in$ NP

DSubsetSum

Input is a positive integer C and n positive integers $s_1, \ldots, s_n$. Is there a subset of these integers that add up to exactly C?

Example

$\{1, 2, 7, 14, 49, 98, 343, 686, 2409, 2793, 16808, 17206, 117705, 117993\}$
and $C = 138457$
Subset: $\{1, 2, 7, 98, 343, 686, 2409, 17206, 117705\}$

- A DSubsetSum yes-input consists of n numbers, and an integer C, such that there is a subset of those integers that add up to C.

DSubsetSum ∈ NP

Question (Certificate)

What is needed to show that the given input is actually a yes-input?

- Ans: A subset T of subscripts with the corresponding integers add up to C.

Proof (Verifying a certificate).

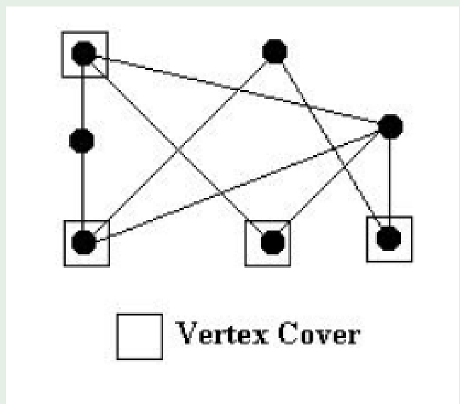
- Given a subset T of subscripts, check whether $\sum_{i \in T} s_i = C$.
- Input-size is $m = (\log_2 C + \sum_{i=1}^{n} \log_2 s_i)$ and verification can be done in time $O(\log_2 C + \sum_{i \in T} \log_2 s_i) = O(m)$, so this is polynomial time.
- Hence we have DSubsetSum $\in \mathcal{NP}$. $\square$

DVC ∈ NP

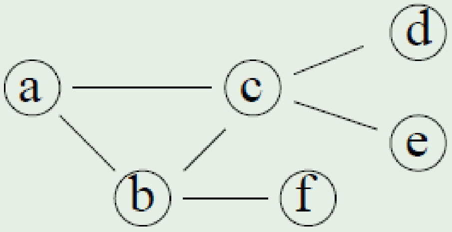
Definition (Vertex Cover)

A vertex cover of a graph G is a set of vertices such that every edge in G is incident to at least one of these vertices.

Example



Example



Find a vertex cover of G of size two

DVC ∈ NP

Decision Vertex Cover (DVC) Problem

Given an undirected graph G and an integer k, does G have a vertex cover with k vertices?

Claim

$DVC \in \mathcal{NP}$.

Proof.

- A certificate will be a set C of k vertices.
- The brute force method to check whether C is a vertex cover takes time $O(ke)$. As $ke < (n+e)^2$, the time to verify is $O((n+e)^2)$. So a certificate can be verified in polynomial time.

$\square$

Satisfiability I

- We will now introduce Satisfiability (SAT), which, we will see later, is one of the most important NP problems.

Definition

A Boolean formula is a logical formula consisting of

1. boolean variables (0=false, 1=true),
2. logical operations
   - $\bar{x}$: NOT,
   - $x \vee y$: OR,
   - $x \wedge y$: AND.

These are defined by:

x	y	$\bar{x}$	$x \vee y$	$x \wedge y$
0	0	1	0	0
0	1		1	0
1	0	0	1	0
1	1		1	1

Satisfiability II

- A given Boolean formula is satisfiable if there is a way to assign truth values (0 or 1) to the variables such that the final result is 1.

Example

$f(x, y, z) = (x \wedge (y \vee \bar{z})) \vee (\bar{y} \wedge z \wedge \bar{x})$.

x	y	z	$(x \wedge (y \vee \bar{z}))$	$(\bar{y} \wedge z \wedge \bar{x})$	$f(x, y, z)$
0	0	0	0	0	0
0	0	1	0	1	1
0	1	0	0	0	0
0	1	1	0	0	0
1	0	0	1	0	1
1	0	1	0	0	0
1	1	0	1	0	1
1	1	1	1	0	1

For example, the assignment $x = 1$, $y = 1$, $z = 0$ makes $f(x, y, z)$ true, and hence it is satisfiable.

Satisfiability III

Example

$$f(x, y) = (x \vee y) \wedge (\bar{x} \vee y) \wedge (x \vee \bar{y}) \wedge (\bar{x} \vee \bar{y}).$$

x	y	$x \vee y$	$\bar{x} \vee y$	$x \vee \bar{y}$	$\bar{x} \vee \bar{y}$	$f(x, y)$
0	0	0	1	1	1	0
0	1	1	1	0	1	0
1	0	1	0	1	1	0
1	1	1	1	1	0	0

There is no assignment that makes $f(x, y)$ true, and hence it is NOT satisfiable.

SAT ∈ NP

SAT problem

Determine whether an input Boolean formula is satisfiable. If a Boolean formula is satisfiable, it is a yes-input; otherwise, it is a no-input.

Claim

$SAT \in \mathcal{NP}$.

Proof.

- The certificate consists of a particular 0 or 1 assignment to the variables.
- Given this assignment, we can evaluate the formula of length n (counting variables, operations, and parentheses), it requires at most n evaluations, each taking constant time.
- Hence, to check a certificate takes time $O(n)$.
- So we have SAT $\in \mathcal{NP}$.

□

k-SAT ∈ NP

- For a fixed k, consider Boolean formulas in k-conjunctive normal form (k-CNF):

$$f_1 \wedge f_2 \wedge \cdots \wedge f_n$$

where each f_i is of the form

$$f_i = y_{i,1} \vee y_{i,2} \vee \cdots \vee y_{i,k}$$

where each $y_{i,j}$ is a variable or the negation of a variable.

Example (3-CNF formula)

$$(x_1 \vee \bar{x}_2 \vee x_3) \wedge (\bar{x}_1 \vee \bar{x}_3 \vee x_4).$$

***k*-SAT problem**

Determine whether an input Boolean k-CNF formula is satisfiable.

Claim

3-SAT $\in \mathcal{NP}$. *2-SAT* $\in \mathcal{P}$

Some Decision Problems in NP

- We have given proofs for:
  - Decision subset sum problem (DSubsetSum),
  - Satisfiability (SAT),
  - Decision vertex cover problem (DVC).

- Some others (without proofs given; try to find proofs):
  - Decision minimum spanning tree problem (DMST),
  - Decision 0-1 knapsack problem (DKnapsack).

P = NP?

- One of the most important problems in computer science is whether P = NP or P ≠ NP? Observe that P ⊆ NP.
  - Given a problem π ∈ P, and a certificate, to verify the validity of a yes-input (an instance of π), we can simply solve π in polynomial time (since π ∈ P). It implies π ∈ NP.

- Intuitively, NP ⊆ P is doubtful.
  - After all, just being able to verify a certificate (corresponding to a yes-input) in polynomial time does not necessary mean we can tell whether an input is a yes-input in polynomial time.

  - However, 30 years after the P = NP? problem was first proposed, we are still no closer to solving it and do not know the answer. The search for a solution, though, has provided us with deep insights into what distinguishes an “easy” problem from a “hard” one.

Outline

- Introduction to Part V
- Problem Classes: P and NP
  - Input size of a problem
  - Optimization problems vs Decision problems
  - The class P and class NP
- Introduction to NP-Completeness (NPC)
  - Polynomial-time reductions
  - The class NPC
  - NP-Complete problems
  - Optimization vs. Decision problems about NPC
- Summary

What is a Reduction?

- Reduction is a relationship between problems.
- Problem Q can be reduced to Q’ if every instance of Q can be “rephrased” as an instance of Q’
- Example 1:
  - Q: multiplying two positive numbers.
  - Q’: adding two numbers.
  - Q can be reduced to Q’ via a logarithmic transformation

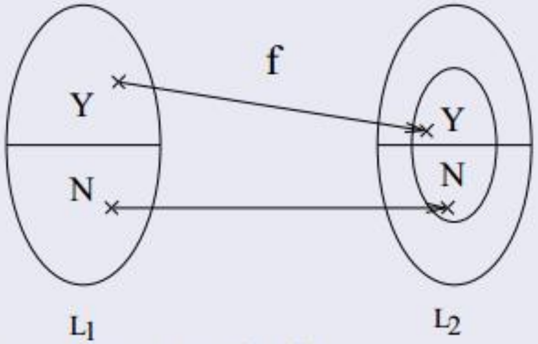
$$xy = exp[logx + logy]$$

- If Q can be reduced to Q’, Q is “no harder to solve” than Q’.

Polynomial-Time Reductions

Definition

- Let L_1 and L_2 be two decision problems.
- A Polynomial-Time Reduction from L_1 to L_2 is a transformation f with the following two properties:
  1. f transforms an input x for L_1 into an input $f(x)$ for L_2 such that
     - a yes-input of L_1 maps to a yes-input of L_2, and a no-input of L_1 maps to a no-input of L_2.

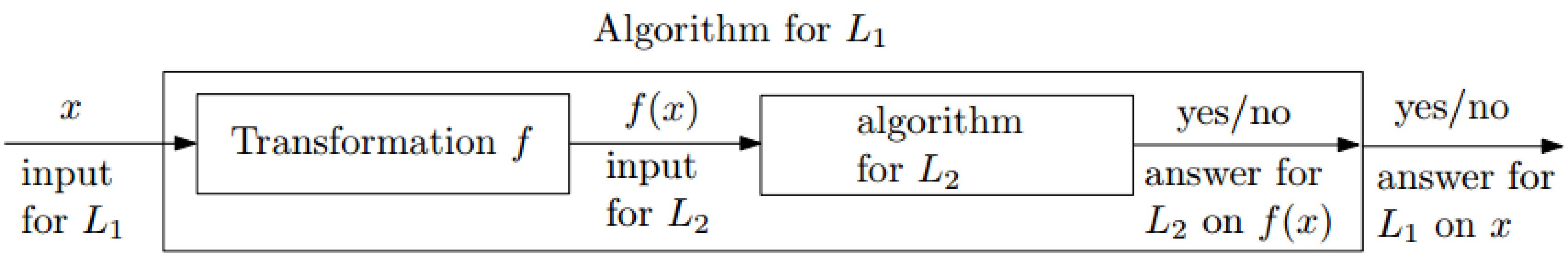


  2. $f(x)$ is computable in polynomial time (in $size(x)$).

If such an f exists, we say that L_1 is polynomial-time reducible to L_2, and write $L_1 \leq_P L_2$.

Polynomial-Time Reductions

- Intuitively, $L_1 \leq_P L_2$ means L_1 is no harder than L_2.
- Given an algorithm A_2 for the decision problem L_2, we can develop an algorithm A_1 to solve L_1:



- If A_2 is polynomial-time algorithm, so is A_1.

Polynomial-Time Reductions $f: L_1 \to L_2$

Theorem

If $L_1 \leq_P L_2$ and $L_2 \in \mathcal{P}$, then $L_1 \in \mathcal{P}$.

Proof.

- $L_2 \in \mathcal{P}$ means we have a polynomial-time algorithm A_2 for L_2.
- Since $L_1 \leq_P L_2$, we have a polynomial-time transformation f mapping input x for L_1 to an input for L_2.

Combining these, we get the following polynomial-time algorithm for solving L_1:

(1) take input x for L_1 and compute $f(x)$;

(2) run algorithm A_2 on input $f(x)$, and return the answer found (for L_2 on $f(x)$) as the answer for L_1 on x.

Each of Steps (1) and (2) takes polynomial time. So the combined algorithm takes polynomial time. Hence $L_1 \in \mathcal{P}$. $\square$

Warning: Note that this does not imply that if $L_1 \leq_P L_2$ and $L_1 \in \mathcal{P}$, then $L_2 \in \mathcal{P}$. This statement is not true.

Reduction between Decision Problems

Lemma (Transitivity of the relation $\leq_P$)

If $L_1 \leq_P L_2$ and $L_2 \leq_P L_3$, then $L_1 \leq_P L_3$.

Proof.

- Since $L_1 \leq_P L_2$, there is a polynomial-time reduction f_1 from L_1 to L_2.
- Similarly, since $L_2 \leq_P L_3$, there is a polynomial-time reduction f_2 from L_2 to L_3.
- Note that $f_1(x)$ can be calculated in time polynomial in $size(x)$. In particular this implies that $size(f_1(x))$ is polynomial in $size(x)$. $f(x) = f_2(f_1(x))$ can therefore be calculated in time polynomial in $size(x)$.
- Furthermore x is a yes-input for L_1 if and only if $f(x)$ is a yes-input for L_3 (why). Thus the combined transformation defined by $f(x) = f_2(f_1(x))$ is a polynomial-time reduction from L_1 to L_3. Hence $L_1 \leq_P L_3$.

Outline

- Introduction to Part V
- Problem Classes: P and NP
  - Input size of a problem
  - Optimization problems vs Decision problems
  - The class P and class NP
- Introduction to NP-Completeness (NPC)
  - Polynomial-time reductions
  - The class NPC
  - NP-Complete problems and proofs
  - Optimization vs. Decision problems about NPC
- Summary

The Class NP-Complete (NPC)

- We have finally reached our goal of introducing class NPC.

Definition

The class $\mathcal{NPC}$ of $\mathcal{NP}$-complete problems consists of all decision problems L such that

1. $L \in \mathcal{NP}$;
2. for every $L' \in \mathcal{NP}$, $L' \leq_P L$.

- Intuitively, NPC consists of all the hardest problems in NP.

NP-Completeness and Its Properties

- Let L be any problem in NPC.

Theorem

1. *If there is a polynomial-time algorithm for L, then there is a polynomial-time algorithm for every $L' \in \mathcal{NP}$.*
2. *If there is no polynomial-time algorithm for L, then there is no polynomial-time algorithm for any $L' \in \mathcal{NPC}$.*

Proof.

1. By definition of $\mathcal{NPC}$, for every $L' \in \mathcal{NP}$, $L' \leq_P L$. Since $L \in \mathcal{P}$, by the theorem on Slide 6, $L' \in \mathcal{P}$.
2. By the previous conclusion.

$\square$

NP-Completeness and Its Properties

- According to the above theorem,
  - either all NP-Complete problems are polynomial time solvable, or
  - all NP-Complete problems are not polynomial time solvable.

- This is the major reason we are interested in NP-Completeness.

The Classes P, NP, and NPC

Recall

$\mathcal{P} \subseteq \mathcal{NP}$.

Question 1

Is $\mathcal{NPC} \subseteq \mathcal{NP}$?

Yes, by definition!

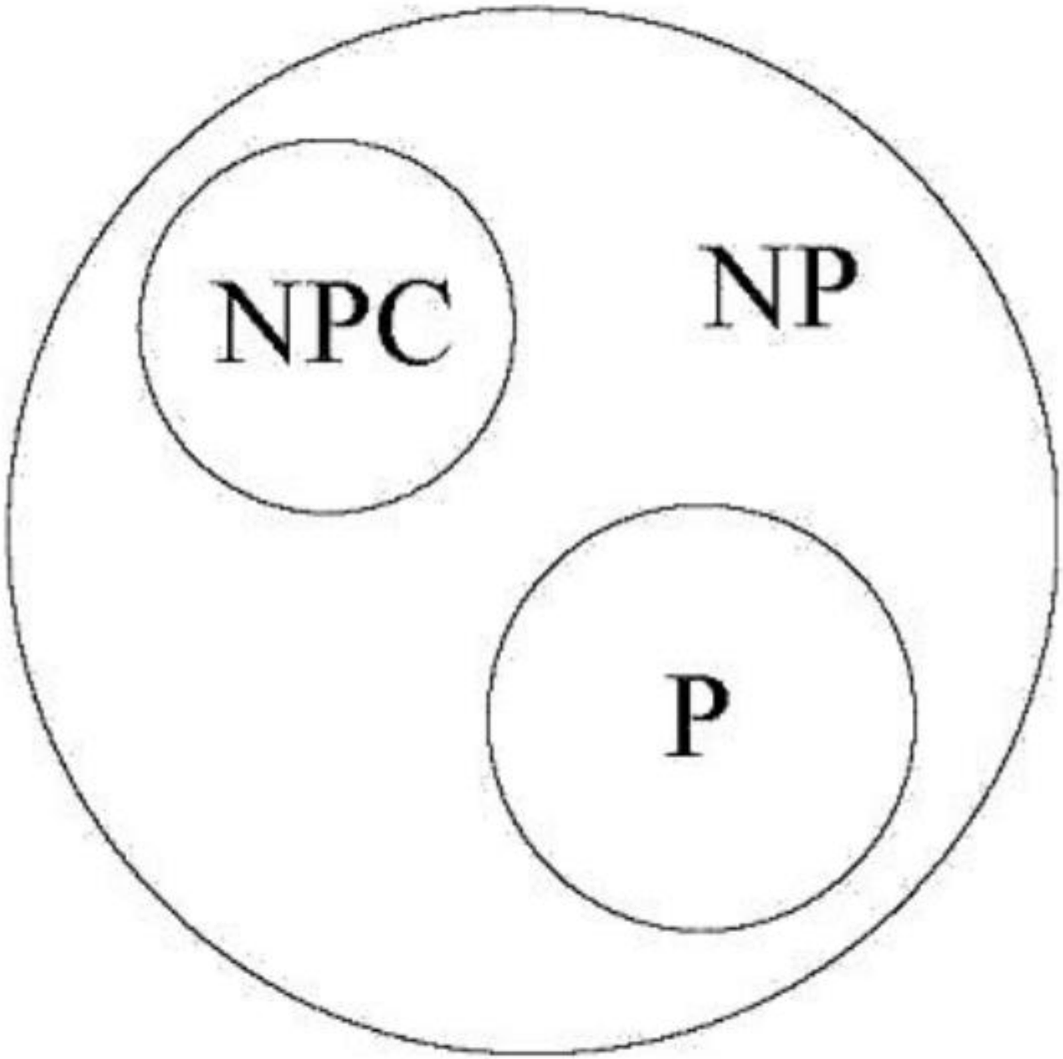
Question 2

Is $\mathcal{P} = \mathcal{NP}$?

Open problem! Probably very hard

It is generally believed that $\mathcal{P} \neq \mathcal{NP}$.

The Classes P, NP, and NPC



Outline

- Introduction to Part V
- Problem Classes: P and NP
  - Input size of a problem
  - Optimization problems vs Decision problems
  - The class P and class NP
- Introduction to NP-Completeness (NPC)
  - Polynomial-time reductions
  - The class NPC
  - NP-Complete problems
  - Optimization vs. Decision problems about NPC
- Summary

The Class NP-Complete (NPC)

- From the definition of NP-complete, it appears impossible to prove one problem L $\in$ NPC!
  - By definition, it requires us to show every L' $\in$ NP, L' $\leq_P$ L.
  - But there are infinitely many problems in NP, so how can we argue there exists a reduction from every L' to L?
- Fortunately, due to the transitivity property of the relation $\leq_p$, we have an alternative way to show that a decision problem L $\in$ NPC:
  - (a) L $\in$ NP;
  - (b) for some L' $\in$ NPC, L' $\leq_P$ L.

Proof.

Let L'' be any problem in $\mathcal{NP}$. Since L' is $\mathcal{NP}$-complete, $L'' \leq_p L'$. Since $L' \leq_p L$, by transitivity, $L'' \leq_p L$. $\square$

Cook's Theorem (SAT ∈ NPC)

Question

How do we prove one problem in $\mathcal{NPC}$ to start with?

Theorem (Cook's Theorem (1971))

$\mathrm{SAT} \in \mathcal{NPC}$.

- Remark: Since Cook showed that SAT ∈ NPC, thousands of problems have been shown to be in NPC using the reduction approach described earlier.
- Remark: With a little more work we can also show that 3-SAT ∈ NPC as well. pp. 998-1002.
- Note: For the purposes of this course you only need to know the validity of Cook's Theorem, and 3-SAT ∈ NPC but do not need to know how to prove them.

Proving that problems are NPC

- In the rest of this lecture, we will discuss the following specific NP-Complete problems.
  - SAT and 3-SAT.
    - We will assume that they are NP-complete (from textbook).
  - DCLIQUE:
    - by showing 3-SAT $\leq_P$ DCLIQUE
    - The reduction used is very unexpected!
  - Decision Vertex Cover (DVC):
    - by showing DCLIQUE $\leq_P$ DVC
    - The reduction used is very natural.
  - Decision Independent Set (DIS):
    - by showing DCLIQUE $\leq_P$ DIS
    - The reduction used is very natural.

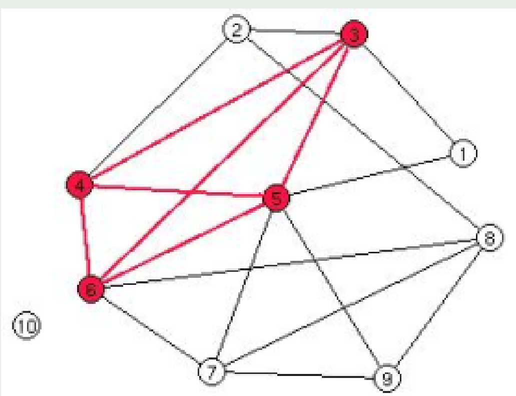
Problem: CLIQUE

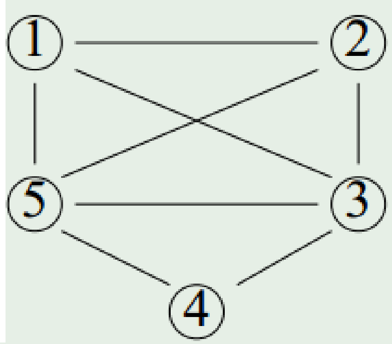
Definition (Clique)

A clique in an undirected graph $G = (V, E)$ is a subset $V' \subseteq V$ of vertices such that each pair $u, v \in V'$ is connected by an edge $(u, v) \in E$. In other words, a clique is a complete subgraph of G

Example

- a vertex is a clique of size 1, an edge a clique of size 2.





Find a clique with 4 vertices

CLIQUE

Find a clique of maximum size in a graph.

NPC Problem: CLIQUE

The Decision Clique Problem DCLIQUE

Given an undirected graph G and an integer k, determine whether G has a clique with k vertices.

Theorem

$\mathrm{DCLIQUE} \in \mathcal{NPC}$.

Proof

We need to show two things.
(a) That $\mathrm{DCLIQUE} \in \mathcal{NP}$ and
(b) That there is some $L \in \mathcal{NPC}$ such that

$$L \leq_P \mathrm{DCLIQUE}.$$

Proof that DCLIQUE ∈ NPC

Claim (a)

$\text{DCLIQUE} \in \mathcal{NP}$

Proof.

Proving (a) is easy.

- A certificate will be a set of vertices $V' \subseteq V$, $|V'| = k$ that is a possible clique.
- To check that V' is a clique all that is needed is to check that all edges (u, v) with $u \neq v$, $u, v \in V'$, are in E.
- This can be done in time $O(|V|^2)$ if the edges are kept in an adjacency matrix (and even if they are kept in an adjacency list – how?).

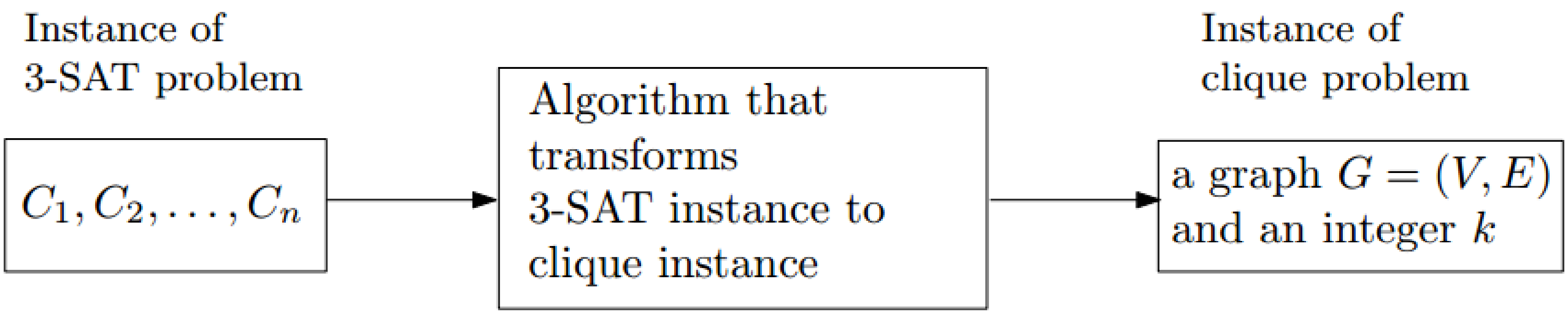
□

Proof that DCLIQUE ∈ NPC (cont)

Claim (b)

There is some $L \in \mathcal{NPC}$ such that $L \leq_P \text{DCLIQUE}$.

- To prove (b) we will show that 3-SAT $\leq_P$ DCLIQUE.



  - This will be the hard part.
  - We will do this by building a 'gadget' that allows a reduction from the 3-SAT problem (on logical formulas) to the DCLIQUE problem (on graphs, and integers).

Proof that DCLIQUE ∈ NPC (cont)

- Recall that the input to 3-SAT is a logical formula Φ of the form

$$\phi = C_1 \wedge C_2 \wedge \cdots \wedge C_n$$

where each C_i is a triple of the form

$$C_i = y_{i,1} \vee y_{i,2} \vee y_{i,3}$$

where each $y_{i,j}$ is a variable or the negation of a variable.

Example

$C_1 = (x_1 \vee \neg x_2 \vee \neg x_3)\,,\; C_2 = (\neg x_1 \vee x_2 \vee x_3)\,,\; C_3 = (x_1 \vee x_2 \vee x_3)$

- We will define a polynomial transformation f from 3-SAT to DCLIQUE

$$f : \phi \mapsto (G, k)$$

that builds a graph G and integer k such that ϕ is a Yes-input to 3-SAT if and only if (G, k) is a Yes-input to DCLIQUE.

Proof that DCLIQUE ∈ NPC（cont）

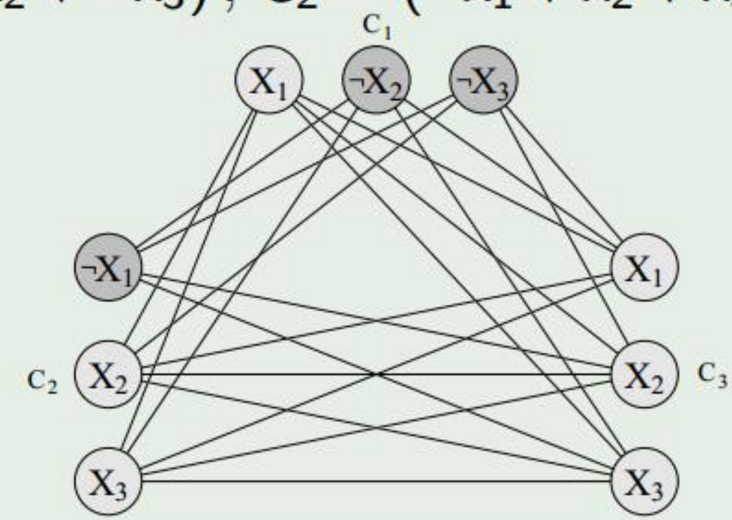
- Suppose that Φ is a 3-SAT formula with n clauses, i.e., $\Phi = C_1 \land C_2 \land \cdot \cdot \cdot \land C_n$.
- We start by setting k = n.
- We now construct the graph G = (V , E).
  - For each clause $C_i = x_{i,1} \lor x_{i,2} \lor x_{i,3}$ we create 3 vertices,v_1^i, v_2^i, v_3^i, in V so G has 3n vertices. We will label these vertices with the corresponding variable or variable negation that they represent. (Note that many vertices might share the same label)
  - We create an edge between vertices v_j^i and $v_{j'}^{i'}$ if and only if the following two conditions hold:
    - (a) v_j^i and $v_{j'}^{i'}$ are in different triples, i.e., i ≠ i', and
    - (b) v_j^i is not the negation of $v_{j'}^{i'}$.
- Note that the transformation maps all 3-SAT inputs to some DCLIQUE inputs, i.e., it does not require that all DCLIQUE inputs have pre-images from 3-SAT inputs.

Proof that DCLIQUE ∈ NPC（cout）

Example

$$\phi = C_1 \wedge C_2 \wedge C_3$$

$$C_1 = (x_1 \vee \neg x_2 \vee \neg x_3)\,,\; C_2 = (\neg x_1 \vee x_2 \vee x_3)\,,\; C_3 = (x_1 \vee x_2 \vee x_3)$$



- Observe that the assignment $X_1 =$false, $X_2 =$false, $X_3 =$true satisfies ϕ (a yes-input for 3-SAT).
- This corresponds to the clique of size 3 comprising the $\neg x_2$ node in C_1, the x_3 node in C_2, and the x_3 node in C_3 (a yes-input for DCLIQUE).

Proof that DCLIQUE ∈ NPC(cout)

Correctness

We claim that a 3-CNF formula ϕ with k clauses is satisfiable if and only if $f(\phi) = (G, k)$ has a clique of size k.

- ⇒: Suppose Φ is satisfiable. Consider the satisfying truth assignment.
  - Each of the k clauses has at least one true literal.
  - Select one such true literal from each clause.
  - Observe that these true literals must be logically consistent with each other (i.e., for any i, x_i and $\neg x_i$ will not both appear).
  - Recall that in our construction of G we connect a pair of vertices if they are in different clauses and are logically consistent.
  - Thus, for every pair of these literals, there must be an edge in G connecting the corresponding vertices.
  - Thus these k vertices must form a clique.

Proof that DCLIQUE ∈ NPC (cout)

- ⇐: Suppose G has a clique of size k.
  - Observe that there is no edge between vertices in the same clause.
  - Hence, each clause ’contributes’ exactly one vertex to the clique.
  - Moreover, since the construction of G connects only logically consistent vertices by an edge, every vertex in the clique must be logically consistent.
  - Hence we can assign all the vertices in the clique to be true, and this truth assignment makes Φ satisfiable.

Proof that DCLIQUE ∈ NPC（cout）

- Note that the graph G has 3k vertices and at most 3k(3k − 1)/2 edges and can be built in $O(k^2)$ time
- So f is a polynomial-time reduction.

- We have therefore just proven that 3-SAT $\leq_P$ DCLIQUE.
- Since we already know that 3-SAT ∈ NPC and have seen that DCLIQUE ∈ NP, we have just proven that DCLIQUE ∈ NPC .

Outline

- Introduction to Part V
- Problem Classes: P and NP
  - Input size of a problem
  - Optimization problems vs Decision problems
  - The class P and class NP
- Introduction to NP-Completeness (NPC)
  - Polynomial-time reductions
  - The class NPC
  - NP-Complete problems
  - Optimization vs. Decision problems about NPC
- Summary

Decision versus Optimization Problems

In General,

- If a decision problem can be solved in polynomial time, then the corresponding optimization problems can also be solved in polynomial time.

- If a decision problem cannot be solved in polynomial time, then the corresponding optimization problems cannot be solved in polynomial time either.

Decision versus Optimization Problems

- One decision problem and two optimization problems:

DVC

Given an undirected graph G and k, is there a vertex cover of size k?

VC

Given an undirected graph G, find a minimum size vertex cover.

MVC

Given an undirected graph G, find the size of a minimum vertex cover.

Note that $DVC(G, k)$ returns Yes if G has a vertex cover of size k and No, otherwise.

Decision versus Optimization Problems

- Consider the following algorithm for solving MVC:

$k = 0;$
while *not* $DVC(G, k)$ **do**
  $k = k + 1$
end
return $k;$

- Note that MVC calls DVC at most |V| times, so if there is a polynomial time algorithm for DVC, then our algorithm for MVC is also polynomial.

Decision versus Optimization Problems

- Here is an algorithm for calculating VC(G) that uses the algorithm for MVC on the previous page. First set t = MVC(G) and then run the following algorithm VC(G, t):

VC(G, t) // find a VC of size t

```
begin
    // Let G_u be a graph such that the vertex u, and its
       corresponding edges are removed from G.
    // We find the vertex u such that
    for Each vertex u in G do
        if MVC(G_u)=t-1 then
            output u, break;
        end
    end
    // such u must exist, why?
    if t > 0 then
        VC(G_u, t-1)
    end
end
```

Decision versus Optimization Problems

- Note that this algorithm calls MVC at most $|V|^2$ times. So, if MVC is polynomial in size(G), then so is the algorithm VC.
- We already saw that if DVC is polynomial in size(G), then so is MVC, so we've just shown that if we can solve DVC in polynomial time, we can solve VC in polynomial time.

NP-Hard Problems

Definition

A problem L is $\mathcal{NP}$-hard if problem in $\mathcal{NPC}$ can be polynomially reduced to it (but L does not need to be in $\mathcal{NP}$).

- In general, the optimization versions of NP-Complete problems are NP-Hard.

Example

VC: Given an undirected graph G, find a minimum-size vertex cover.
DVC: Given an undirected graph G and k, is there a vertex cover of size k?
If we can solve the optimization problem VC, we can easily solve the decision problem DVC.

- Simply run VC on graph G and find a minimum vertex cover S.
- Now, given (G, k), solve $DVC(G, k)$ by checking whether $k \geq |S|$. If $k \geq |S|$, answer Yes, if not, answer No.

Outline

- Introduction to Part V
- Problem Classes: P and NP
  - Input size of a problem
  - Optimization problems vs Decision problems
  - The class P and class NP
- Introduction to NP-Completeness (NPC)
  - Polynomial-time reductions
  - The class NPC
  - NP-Complete problems and proofs
  - Optimization vs. Decision problems about NPC
- Summary

A Review on NP-Completeness

- Input size of problems.
- Polynomial-time and nonpolynomial-time algorithms.
- Polynomial-time solvable problems.
- Decision problems.
- Optimization problems and their decision problems.
- The classes P, NP, and NPC.
- Polynomial-time reduction.
- How to prove $L \in P$, or NP, or NPC?
- Examples of problems in these classes.
    - Satisfiability of Boolean formulas (SAT)
    - Decision clique (DCLIQUE)
    - Decision vertex cover (DVC)
    - Decision independent set (DIS)

谢谢

BDA

Beihang University