

算法设计与分析

Lecture 11: NP Complete Theory

卢杨

厦门大学信息学院计算机科学系

luyang@xmu.edu.cn

A Story

- Suppose you work in industry, and your boss gives you the task of finding an efficient algorithm for some problem very important to the company.
- After laboring long hours on the problem for over a month, you have no idea at all toward an efficient algorithm.
- Giving up, you return to your boss and ashamedly announce that you simply can't find an efficient algorithm.
 - Your boss: "I'm going to fire you and hire some smarter guys to solve this problem!"
 - You: "It is not because I'm stupid, but it is not possible to find an efficient algorithm for this problem!"
- So, your boss gives you another month to prove that it is impossible.



A Story

- After a second month of hardworking, you fail again.
 - At this point you've failed to obtain an efficient algorithm and you've failed to prove that such an algorithm is not possible.
 - You are on the edge of being fired.
- Suddenly, you discover that the company's problem is similar to the 0/1 knapsack problem, and if the company's problem is solved, it will lead to a more efficient algorithm for the 0/1 knapsack problem.



A Story

- However, no one has ever found a more efficient algorithm for the 0/1 knapsack problem or proven that such an algorithm is not possible.
- You see one last hope. If you could prove that an efficient algorithm for the company's problem would automatically yield a more efficient algorithm for the 0/1 knapsack problem, it would mean that your boss is asking you to accomplish something that even the greatest computer scientists can't solve.
- You ask for a chance to prove this, and your boss agrees.



A Story

- After only a week of effort, you do indeed prove that an efficient algorithm for the company's problem would automatically yield a more efficient algorithm for the 0/1 knapsack problem.
- Rather than being fired, you're given a promotion because you have saved the company a lot of money.
- Your boss now realizes that it would not be worthy to continue to expend great effort looking for an exact algorithm for the company's problem and that other directions, such as looking for an approximate solution, should be explored.
- Happy ending~



Computational Complexity

- What we have just described is exactly what computer scientists have successfully done for the last 30 years.
- Given a problem, we can't solve it, and we can't prove it is impossible either, so we show that are equally hard as other similar problems.
 - If we had an efficient algorithm for any one of them, we would have efficient algorithms for all of them.
 - Such an algorithm has never been found, but it's never been proven that one is not possible.
- These interesting problems are called **NP-complete (NP完全)** and are the focus of this lecture.





DECISION PROBLEM

Polynomial-Time Algorithm

Definition

A **polynomial-time algorithm** (多项式时间算法) is one whose worst-case time complexity is bounded above by a polynomial function of its input size. That is, if n is the input size, there exists a polynomial function $p(n)$ such that

$$W(n) = O(p(n)).$$

where $W(n)$ is the worst-case time complexity.



Example

Algorithms with the following worst-case time complexities are all polynomial-time.

$$2n \quad 3n^3 + 4n \quad 5n + n^{10} \quad n \lg n$$

Algorithms with the following worst-case time complexities are not polynomial-time.

$$2^n \quad 2^{0.01n} \quad 2^{\sqrt{n}} \quad n!$$



Polynomial-Time Algorithm

- In computer science, we divide problems into “easy group” and “difficult group” based on the boundary: **can be solved in polynomial-time or not**.
- A problem is called **intractable (难解的)** if it is **impossible** to solve it with a polynomial-time algorithm.
- A problem can be solved in $\Theta(n^{100})$ is also in “easy group”.
 - Once we have a $\Theta(n^{100})$ algorithm, we may improve it to a $\Theta(n^{90})$ algorithm a few moments later.



The Three General Problem Categories

- There are three general categories of problems we concern:
 1. Problems for which polynomial-time algorithms have been found.
 2. Problems that have been proven to be intractable.
 3. Problems that have not been proven to be intractable, but for which polynomial-time algorithms have never been found.
- It is a surprising phenomenon that **most problems in computer science fall into either the first or third category.**
 - It's extremely hard to prove the intractability of a problem (can't find a polynomial-time algorithm for it).



The Three General Problem Categories

1. Problems for which polynomial-time algorithms have been found.
 - For example, we have found:
 - $\Theta(n \lg n)$ algorithms for sorting.
 - $\Theta(\lg n)$ algorithm for searching a sorted array.
 - $\Theta(n^{2.38})$ algorithm for matrix multiplication.
 - $\Theta(n^3)$ algorithm for chained matrix multiplication.
 - There are some other problems for which we have developed polynomial-time algorithms, but for which the obvious brute-force algorithms are nonpolynomial (usually exponential).
 - E.g. the shortest paths problem, the optimal BST problem, and the MST problem.



The Three General Problem Categories

3. Problems that have not been proven to be intractable, but for which polynomial-time algorithms have never been found.
- For example, the 0-1 knapsack problem, the traveling salesperson problem, the sum-of-subsets problem, the m -coloring problem for $m \geq 3$, and the Hamiltonian cycle problem all fall into this category.
 - We have found branch-and-bound algorithms, backtracking algorithms, and other algorithms for these problems that are efficient for many large instances. **However, they are still not polynomial-time.**



Decision Problem

- It is more convenient to develop the theory if we restrict ourselves to **decision problems (判定问题)**.
- The output of a decision problem is a simple “yes” or “no” answer.
- Yet when we introduced some of the problems mentioned previously, we presented them as optimization problems.
 - The output is an optimal solution.
- Each optimization problem has a corresponding decision problem.



Example

- The **0-1 knapsack optimization problem** is to determine the maximum total profit of the items that can be placed in a knapsack with capacity W .
- The corresponding **0-1 knapsack decision problem** is to determine, for a given profit P , whether it is possible to load the knapsack so as to keep the total weight no greater than W , while making the total profit at least equal to P .
- This problem has the same parameters as the 0-1 knapsack optimization problem plus the additional parameter P .



Example

Optimization problem

i	v_i	w_i
1	\$10	1kg
2	\$12	1kg
3	\$15	2kg
4	\$20	3kg

$W=5\text{kg}$

What is the maximum total profit?

Decision problem

i	v_i	w_i
1	\$10	1kg
2	\$12	1kg
3	\$15	2kg
4	\$20	3kg

$W=5\text{kg}$

Can we make maximum total profit not less than 40?



Example

- The **shortest path optimization problem** is to find the shortest path from vertex u to vertex v in an unweighted graph $G = (V, E)$.
- The corresponding **shortest path decision problem** is to determine whether it is possible to find a path from vertex u to vertex v in an unweighted graph $G = (V, E)$, though at most k edges.
 - This problem has the same parameters as the shortest path optimization problem plus the additional parameter k .



Decision Problem

- A polynomial-time algorithm for an optimization problem automatically solves the corresponding decision problem.
 - E.g., find the shortest path and compare it with k .
- For many decision problems, it's been shown that a polynomial-time algorithm for the decision problem would also yield a polynomial-time algorithm for the corresponding optimization problem.
 - Therefore, we can initially develop our theory considering only decision problems.
- They are equivalent!



Example

- Considering the minimum vertex coloring problem.
- If we have a polynomial-time algorithm for the corresponding decision problem: $\text{ColorIsTrue}(G, k)$. It returns true if and only if graph G can be k -colored.
- The algorithm for minimum vertex coloring problem can be constructed by:

```
MinColoring( $G$ )  
1  $k \leftarrow 0$   
2 while  $\text{ColorIsTrue}(G, k) \neq \text{True}$  do  
3    $k \leftarrow k + 1$   
4 return  $k$ 
```

- It can be easily verified that $\text{MinColoring}(G)$ is also polynomial-time.





P AND NP

Encoding

- If a computer program is to solve an **abstract problem (抽象问题)**, problem instances must be represented in a way that the program understands.
- An encoding is a mapping e from abstract objects to the set of binary strings.
- Thus, a computer algorithm that "solves" some abstract decision problem actually takes an **encoding of a problem instance as input**.



Encoding

- Abstract problem: 0/1 knapsack problem.
- Problem instance: 0/1 knapsack problem with $w_1 = 1, w_2 = 1, w_3 = 2, v_1 = 10, v_2 = 12, v_3 = 15, W = 3$.
- Encoding of this problem instance:

11001010101 ... 101010010101



Encoding

- We call a problem whose instance set is the set of binary strings a **concrete problem (具体问题)**.
- Given a problem instance i of length $n = |i|$, we say that an algorithm **solves** a concrete problem in time $O(T(n))$ if the algorithm can produce the solution in $O(T(n))$ time.
- A concrete problem is **polynomial-time solvable (多项式时间可解的)**, therefore, if there exists an algorithm to solve it in time $O(n^k)$ for some constant k .



A Formal-Language Framework

- The formal-language framework allows us to express the **relation between decision problems and algorithms** that solve them concisely.
- An **alphabet (字符集)** Σ is a finite set of symbols.
- A **language (语言)** L over Σ is a set of strings made up of symbols from Σ .
 - For example, if $\Sigma = \{0,1\}$, the language $L = \{10, 11, 101, 111, 1011, 1101, 10001, \dots\}$ is the language of binary representations of prime numbers.



A Formal-Language Framework

- Denote empty string by ε , and the empty language by \emptyset .
- The language of all strings over Σ is denoted Σ^* .
 - For example, if $\Sigma = \{0,1\}$, then $\Sigma^* = \{\varepsilon, 0, 1, 00, 01, 10, 11, 000, \dots\}$ is the set of all binary strings.
- Every language L over Σ is a subset of Σ^* .



A Formal-Language Framework

- As we encode every problem instance into binary representation, the set of instances for any decision problem Q is simply a subset of Σ^* , where $\Sigma = \{0,1\}$.
- Let x be an instance of the decision problem Q and $Q(x) = 1$ if x is true.
- Since Q is entirely characterized by those problem instances that produce a 1 (yes) answer, **we can use language L over $\Sigma = \{0,1\}$ to represent problem Q** , where

$$L = \begin{bmatrix} 10010010 \dots 1010010 \\ 10111010 \dots 1110100 \\ \vdots \\ 11001001 \dots 1100011 \end{bmatrix} = \{x \in \Sigma^* : Q(x) = 1\}.$$



A Formal-Language Framework

- Now, we can **use language to represent problem**.
- For example, the decision problem PATH has the corresponding language:

$\text{PATH} = \{\langle G, u, v, k \rangle : G = (V, E) \text{ is an undirected graph, } u, v \in V, k \geq 0 \text{ is an integer, and there exists a path from } u \text{ to } v \text{ in } G \text{ consisting of at most } k \text{ edges}\}.$



A Formal-Language Framework

Definition 11.1

We say that an algorithm A **accepts** (接受) a string $x \in \{0,1\}^*$ if, given input x , the algorithm's output $A(x)$ is 1.

The language accepted by an algorithm A is the set of strings:

$$L = \{x \in \{0,1\}^* : A(x) = 1\}$$

that is, the set of strings that the algorithm accepts.

An algorithm A **rejects** (拒绝) a string x if $A(x) = 0$.

- However, we can't say that algorithm A solves L , because we're not sure if A will reject all $x \notin L$.
 - A may iterate forever for such x .



A Formal-Language Framework

Definition 11.2

A language L is **decided** (判定) by an algorithm A if $\forall x \in L$ is accepted by A and $\forall x \notin L$ is rejected by A .

A language L is **accepted** in polynomial time by an algorithm A if there is a constant k such that for any length- n string $x \in L$, algorithm A **accepts** x in time $O(n^k)$.

A language L is **decided** in polynomial time by an algorithm A if there is a constant k such that for any length- n string $x \in \{0,1\}^*$, algorithm A correctly **decides** whether $x \in L$ in time $O(n^k)$.



A Formal-Language Framework

- We can informally define a **complexity class (复杂类)** as a set of languages, membership in which is determined by a complexity measure, such as running time, of an algorithm that determines whether a given string x belongs to language L .
- Using this language-theoretic framework, we can provide an alternative definition of the complexity class P:

Definition 11.3

$P = \{L \in \{0, 1\}^* : \text{there exists an algorithm } A \text{ that decides } L \text{ in polynomial time}\}.$

Theorem 11.1

$P = \{L : L \text{ is accepted by a polynomial-time algorithm}\}.$



P and NP

- All decision problems for which we have found polynomial-time algorithms are certainly in P.
 - Searching, sorting, matrix multiplication...
- However, could some decision problem like 0/1 knapsack decision problem for which we have not found a polynomial-time algorithm also be in P?
 - We don't know! It could possibly be in P.
 - No one finds a polynomial-time algorithm and no one proves that it is not in P.



P and NP

- It seems that the research on this kind of hard problem gets stuck.
- Instead, for this kind of decision problem like 0/1 knapsack decision problem, we'd like to if we can **verify an answer in polynomial-time**.



Hamiltonian Cycle Problem

- A **Hamiltonian cycle (哈密顿回路)** of an undirected graph $G = (V, E)$ is a simple cycle that contains each vertex in V .
- A graph that contains a Hamiltonian cycle is said to be Hamiltonian; otherwise, it is non-Hamiltonian.
- The Hamiltonian cycle problem "Does a graph G have a Hamiltonian cycle?" can be defined as a formal language:
$$\text{HamCycle} = \{G : G \text{ is a Hamiltonian graph}\}.$$
- Solving this problem can trace back to hundreds years ago. Now, we still can't find a polynomial-time algorithm that decides HamCycle.



Hamiltonian Cycle Problem

- Now, if god tells you a cycle in G and let you verify if this cycle is a Hamiltonian cycle.
- Is this problem easy to solve?
- Of course! Simply check if it is a simple cycle and if every edge in this cycle is in E .
- The verification can be done in $O(n^2)$. It is obviously polynomial time.



Verification Algorithms

- A **verification algorithm** (验证算法) can be defined as an algorithm A with two arguments:
 - An ordinary input string x , which is a problem instance;
 - A string y called a **certificate** (证书), which is a given answer of x .
- Verification algorithm A verifies an problem instance string x if there exists a certificate y such that $A(x, y) = 1$.
- The language can be verified by a verification algorithm A is
$$L = \{x \in \{0,1\}^* : \text{there exists } y \in \{0,1\}^* \text{ such that } A(x, y) = 1\}.$$
 - Intuitively, an algorithm A verifies a language L if for any string $x \in L$, there is a certificate y that A can use to prove that $x \in L$.
 - Moreover, for any string $x \notin L$, there must be no certificate proving that $x \in L$.



NP

Definition 11.4

The complexity class NP is the class of languages that can be **verified by a polynomial-time algorithm**.

- Notice that NP stands for “nondeterministic polynomial”, rather than “non-polynomial”.
 - What is nondeterministic???



Nondeterministic Algorithm

- To state the notion of polynomial-time verifiability more concretely, we introduce the concept of a **nondeterministic algorithm (非确定性算法)**.
- We can think of such an algorithm as being composed of the following two separate stages:
 1. **Guessing (Nondeterministic) Stage:** Guess a solution to the given instance. It is called nondeterministic because this stage is totally random.
 2. **Verification (Deterministic) Stage:** Verify the answer: (1) the answer for this instance is “yes”, (2) the answer for this instance is “no”, or (3) can't verify at all (that is, going into an infinite loop). It is called deterministic because only “yes” or “no” can be produced in this stage.



Nondeterministic Algorithm

- For the Hamiltonian cycle problem, we can design a nondeterministic algorithm:

```
NondeterministicHamCycle( $G$ )
1  $S \leftarrow V$ 
2  $p \leftarrow \emptyset$ 
3 for  $i \leftarrow 1$  to  $|V|$  do
4   Randomly select one vertex  $v$  from  $S$ 
5    $p \leftarrow p \cup \{v\}$ 
6    $S \leftarrow S - \{v\}$ 
7 if  $\text{verify}(G, p)=1$  then return True
8 else return False
```



Polynomial-Time Nondeterministic Algorithm

Definition

A **polynomial-time nondeterministic algorithm** is a nondeterministic algorithm whose verification stage is a polynomial-time algorithm.

Definition

NP is the set of all decision problems that can be solved by polynomial-time nondeterministic algorithms.

- We are not sure if one can be solved in polynomial-time, but we know that we can verify an answer of it in polynomial-time.
- This definition is equivalent to the previous definition.



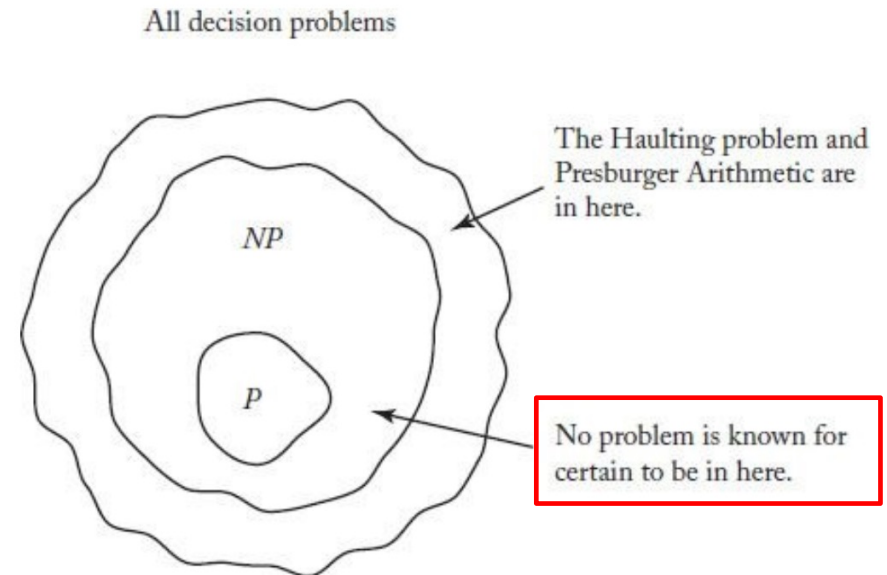
P and NP

- There are thousands of problems that no one has been able to solve with polynomial-time algorithms but that have been proven to be in NP because polynomial-time nondeterministic algorithms have been developed for them.
 - The answers to them can be verified in polynomial-time.
- Every problem in P is also in NP.
 - This is trivially true because any problem in P can be solved by a polynomial-time algorithm.
 - When it is solved, it is also verified.
- There are only few problems that have been proved not in NP.
 - The intractable problems, e.g. the Halting problem and Presburger Arithmetic.










P and NP

- Because P is in NP , it is easy to think that NP contains P as a proper subset.
- However, this may not be the case. That is, **no one has ever proven that there is a problem in NP that is not in P .**
- No one knows if $P=NP$ or not yet.
 - If $P=NP$, it means that once we can verify answer of a problem in polynomial-time, we can solve it in polynomial-time!



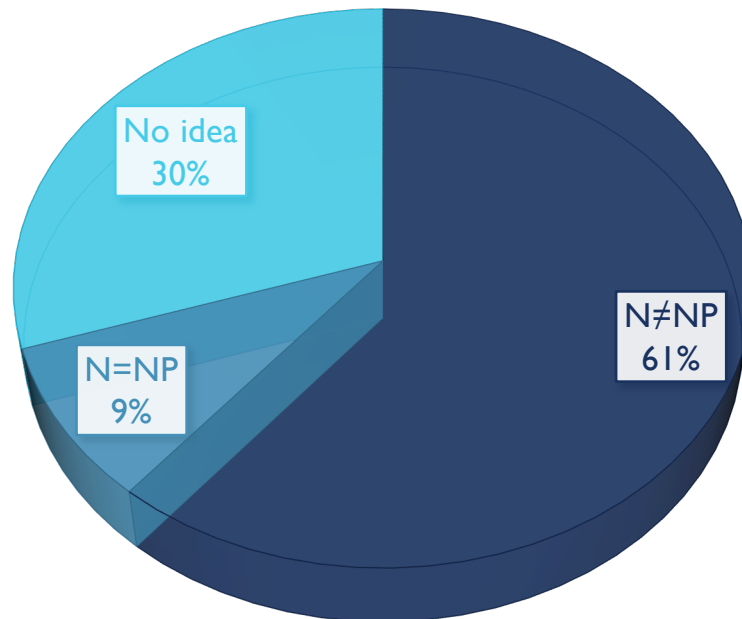
P=NP?

- To prove $P=NP$ or $P \neq NP$ is one of the **Millennium Prize Problems (千禧年大奖难题)**, which are seven problems in mathematics that were stated by the Clay Mathematics Institute on May 24, 2000.
 - A correct solution to any of the problems results in a US\$1 million prize being awarded by the institute to the discoverer(s).
- They are:
 -  P versus NP (P/NP问题)
 -  Birch and Swinnerton-Dyer conjecture (贝赫和斯维讷通-戴尔猜想)
 -  Hodge conjecture (霍奇猜想)
 -  Navier-Stokes existence and smoothness (纳维-斯托克斯存在性与光滑性)
 -  Poincaré conjecture (庞加莱猜想)
 -  Riemann hypothesis (黎曼猜想)
 -  Yang-Mills existence and mass gap (杨-米尔斯存在性与质量间隙)



P=NP?

A POLL OVER 100 COMPUTER SCIENTISTS IN 2002



Consequences of Solution

- Either $P=NP$ or $P \neq NP$ is proved would advance theory enormously, and perhaps have huge practical consequences as well.
- If $P=NP$ is proved:
 - Some cryptography encryption methods in NP will not be treated safe any more.
 - Researchers will be more confident to propose polynomial-time algorithms for problems in NP.
- If $P \neq NP$ is proved:
 - Some cryptography encryption methods in NP will be treated safe.
 - It would allow one to show in a formal way that many common problems cannot be solved efficiently, so that the attention of researchers can be focused on partial solutions or solutions to other problems.



Classroom Exercise

Design a polynomial-time nondeterministic algorithm for the 0/1 knapsack problem.





NP COMPLETE

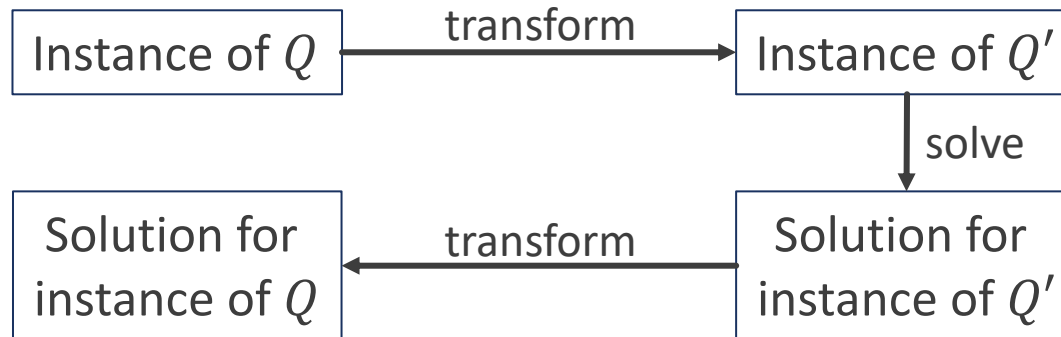
NP Complete

- There are a lot of problems:
 - No polynomial-time algorithm is found;
 - Can't prove it is impossible to find a polynomial-time algorithm.
- There is a special group among these problems. If a polynomial-time algorithm is developed for one of it, every problem in this group can be solved in polynomial time.
- This group is called **NP complete**.
 - There are more than 3000 known NP-complete problems. See [here](#) for some typical ones.



Reducibility

- Before formally define NP complete, we first introduce the concept of **reducibility** (约简).
- If any instance of problem Q can be easily transformed into an instance of problem Q' , we say that problem Q reduces to problem Q' , and the solution for the transformed instance of Q' is also the solution for the instance of Q .



- If we find such a transformation, Q will not be harder than Q' .



Example

- Problem Q : solve linear equation $ax + b = 0$.
- Problem Q' : solve quadratic equation $ax^2 + bx + c = 0$.
- Any instance of Q can be transformed into an instance of Q' :

$$ax + b = 0 \quad \longrightarrow \quad 0x^2 + ax + b = 0$$



Reducibility

Definition 11.6

A language L_1 is polynomial-time reducible to a language L_2 , written $L_1 \leq_P L_2$, if there exists a polynomial-time computable function $f: \{0, 1\}^* \rightarrow \{0, 1\}^*$ such that for all $x \in \{0, 1\}^*$,

$$x \in L_1 \text{ if and only if } f(x) \in L_2$$

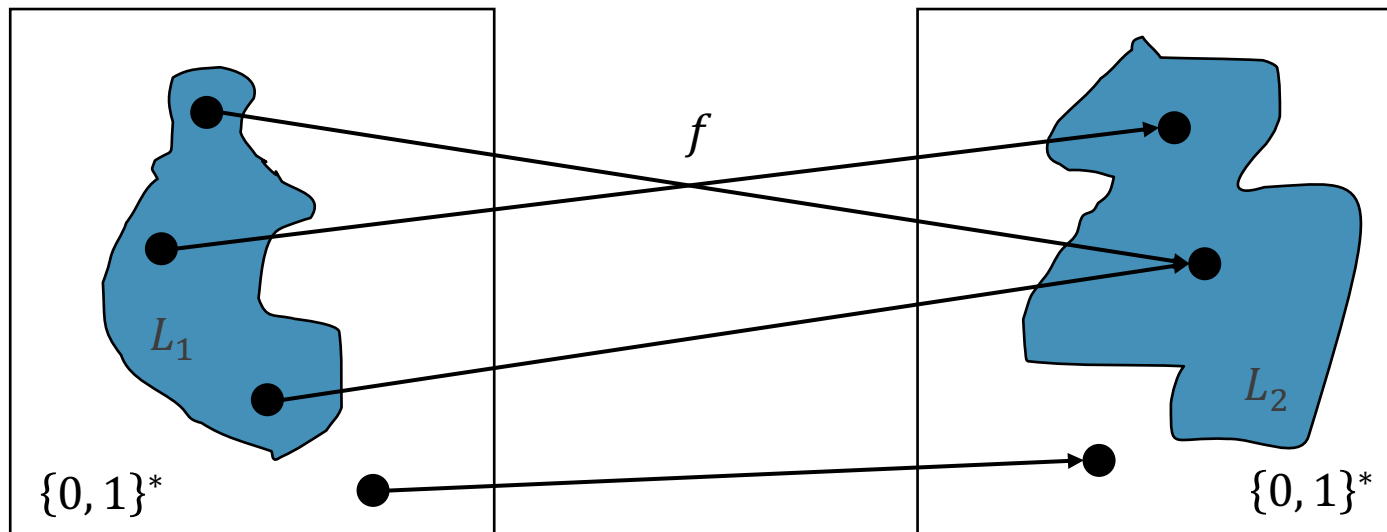
We call the function f the **reduction function (约简函数)**, and a polynomial-time algorithm A that computes f is called a **reduction algorithm (约简算法)**.

- The symbol \leq_P means polynomial-time reducible. $L_1 \leq_P L_2$ indicates the difficulty of L_1 will not be higher than L_2 .



Reducibility

- The reduction function f provides a polynomial-time mapping such that if $x \in L_1$, then $f(x) \in L_2$. Moreover, if $x \notin L_1$, then $f(x) \notin L_2$.



Reducibility

Lemma 11.1

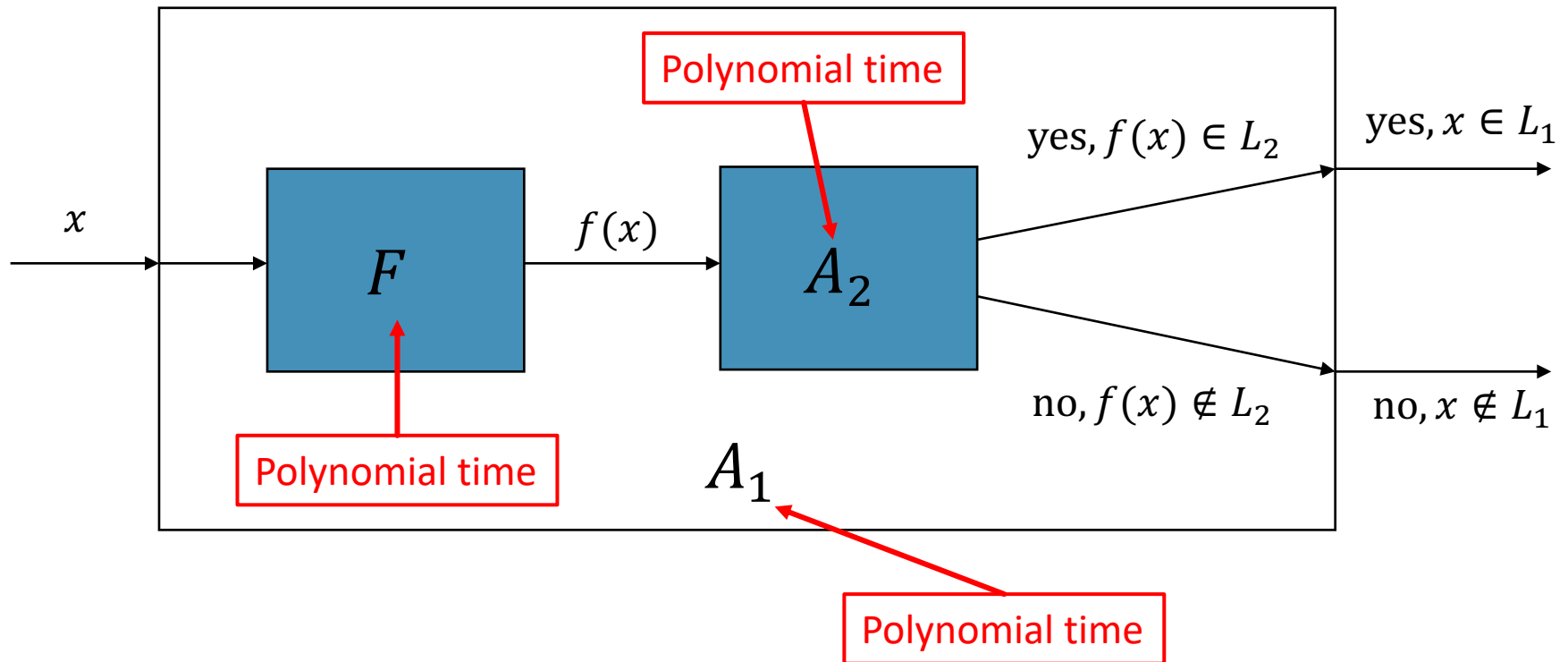
If $L_1, L_2 \in \{0,1\}^*$ are languages such that $L_1 \leq_P L_2$, then $L_2 \in P$ implies $L_1 \in P$.

Proof:

- Let A_2 be a polynomial-time algorithm that decides L_2 , and let F be a polynomial-time reduction algorithm that computes the reduction function f .
- We can construct a polynomial time algorithm A_1 that decides L_1 .



Reducibility



NP-Completeness

- Polynomial-time reductions provide a formal means for showing that one problem is at least as hard as another, to within a polynomial-time factor.

Definition 11.7

A language $L \in \{0,1\}^*$ is **NP-complete (NPC, NP完全)** if

1. $L \in \text{NP}$, and
2. $L' \leq_P L$ for every $L' \in \text{NP}$.

If a language L satisfies property 2, but not necessarily property 1, we say that L is **NP-hard (NP难)**.

- Usually, the optimization problem of an NPC decision problem is NP-hard, because verifying an optimization problem is not in NP.



NP-Completeness

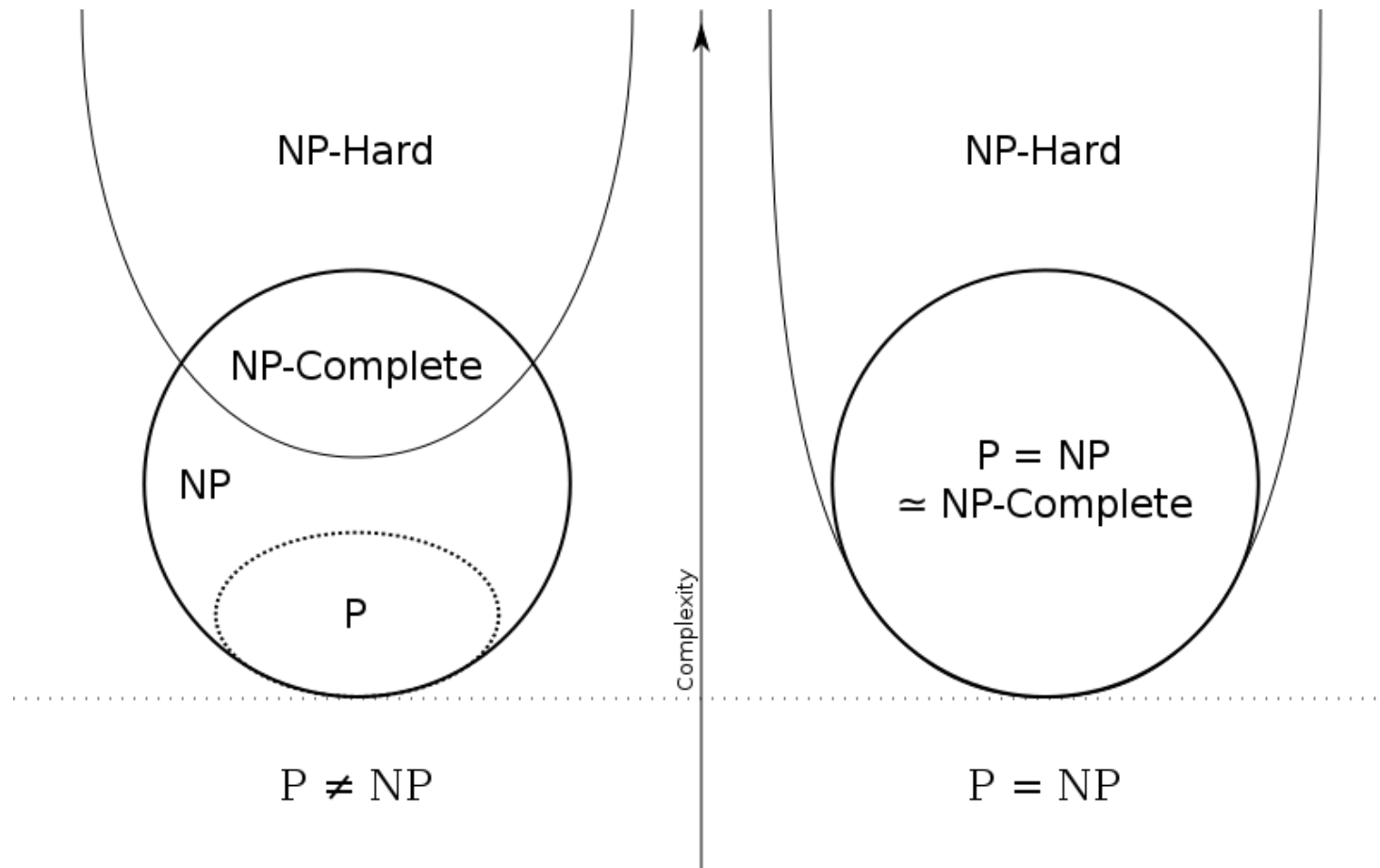
- A problem is NP-complete if it is both in NP and NP-hard.
- The NP-complete problems represent the **hardest problems in NP**.
- If any NP-complete problem has a polynomial time algorithm, all problems in NP do.

Theorem 11.2

If any NP-complete problem is polynomial-time solvable, then $P=NP$. Equivalently, if any problem in NP is not polynomial-time solvable, then no NP-complete problem is polynomial time solvable.



NP-Completeness



NP-Completeness Proofs

- By Definition 11.7, if we want to prove language L be NPC, we should prove:
 1. $L \in NP$, and
 2. $L' \leq_P L$ for every $L' \in NP$.
- Property 1 is easy to prove by constructing a polynomial-time verification algorithm.
- How to prove property 2? Find all $L' \in NP$ in the university and show $L' \leq_P L$?



NP-Completeness Proofs

Lemma 11.4

If L is a language such that $L' \leq_P L$ for some $L' \in \text{NPC}$, then L is NP-hard. Moreover, if $L \in \text{NP}$, then $L \in \text{NPC}$.

- This Lemma tells you: you don't need to prove $L' \leq_P L$ for all $L' \in \text{NP}$, just prove $L' \leq_P L$ for one $L' \in \text{NPC}$.

Proof:

- Since L' is NPC, for all $L'' \in \text{NP}$, we have $L'' \leq_P L'$.
- By supposition, $L' \leq_P L$, and thus by transitivity, we have $L'' \leq_P L' \leq_P L$ for all $L'' \in \text{NP}$, which shows that L is NP-hard.
- If $L \in \text{NP}$, we also have $L \in \text{NPC}$.



NP-Completeness Proofs

Method to prove that a language L is NP-complete:

1. Prove $L \in \text{NP}$.
2. Select a known NP-complete language L' .
3. Describe an algorithm that computes a function f mapping every instance $x \in \{0,1\}^*$ of L' to an instance $f(x)$ of L .
4. Prove that the function f satisfies $x \in L'$ **if and only if** $f(x) \in L$ for all $x \in \{0,1\}^*$.
5. Prove that the algorithm computing f runs in polynomial time.



NP-Completeness Proofs

- Things seem to be easy now, we can use an NPC problem to prove more NPC problems.
- However, where is the first NPC problem?



Satisfiability Problem

- A Boolean formula φ composed of:
 - n Boolean variables: x_1, x_2, \dots, x_n ;
 - m Boolean connectives (连接符): such as \wedge (AND), \vee (OR), \neg (NOT), \rightarrow (implication), \leftrightarrow (if and only if);
 - Parentheses.
- If we can find a set of values for the variables of φ that causes it to evaluate to 1, we call this formula φ is **satisfiable (可满足的)**.
- The **satisfiability problem (SAT, 可满足性问题)** is to determine if a formula is satisfiable.



Example

- Given a Boolean formula:

$$\varphi = \left((x_1 \rightarrow x_2) \vee \neg((\neg x_1 \leftrightarrow x_3) \vee x_4) \right) \wedge \neg x_2$$

is this formula satisfiable?

- Yes, assign $x_1 = 0, x_2 = 0, x_3 = 1, x_4 = 1$, we get:

$$\begin{aligned} & \left((0 \rightarrow 0) \vee \neg((\neg 0 \leftrightarrow 1) \vee 1) \right) \wedge \neg 0 \\ &= (1 \vee \neg(1 \vee 1)) \wedge 1 \\ &= 1 \end{aligned}$$



Satisfiability Problem

Theorem 11.4

SAT is NP-complete.

- In 1971, Stephen Cook proves the **first NPC problem**.
- This problem itself is not interesting at all.
- However, we can use it to prove NP-completeness of other problems.



Stephen Cook
Turing Award in 1982



Satisfiability Problem

Proof:

- Show that SAT is in NP.
 - The verifying algorithm simply replaces each variable in the formula with its corresponding value and then evaluates the expression.
 - This task is easily done in polynomial time.
 - If the expression evaluates to 1, the formula is satisfiable.
- Show that SAT is NP-hard.
 - Not that easy to prove.



Satisfiability Problem

- It can be shown that:
 - $SAT \leq_P$ Hamiltonian cycle decision problem.
 - Hamiltonian cycle decision problem \leq_P undirected traveling salesperson decision problem.
 - Undirected traveling Salesperson decision problem \leq_P traveling salesperson decision problem.
 - ...
- Now, we don't need to use the definition to prove the NP-completeness of a problem. **Instead, simply find another NP-complete problem and show the reducibility.**



3-CNF Satisfiability

- A literal in a Boolean formula is an occurrence of a variable x_i or its negation $\neg x_i$.
- A Boolean formula is in **conjunctive normal form (CNF, 合取范式)**, if it is expressed as an AND of clauses, each of which is the OR of one or more literals.
- A Boolean formula is in 3-CNF, if each clause has exactly three distinct literals. For example:
$$(x_1 \vee \neg x_1 \vee \neg x_2) \wedge (x_3 \vee x_2 \vee x_4) \wedge (\neg x_1 \vee \neg x_3 \vee \neg x_4)$$
- In 3-CNF-SAT problem, we are asked whether a given Boolean formula φ in 3-CNF is satisfiable.



NP-Completeness of 3-CNF Satisfiability

Theorem 11.5

3-CNF-SAT is NP-complete.

Proof:

- **Step 1:** 3-CNF-SAT in NP is obvious, just like SAT problem.
- **Step 2:** Prove $SAT \leq_P 3\text{-CNF-SAT}$.
- Therefore, we need to construct a reduction algorithm to transform any formula into 3-CNF.
 - We have learned how to transform any formula into CNF in discrete math.
- The reduction algorithm has three steps.



NP-Completeness of 3-CNF Satisfiability

Proof (cont'd):

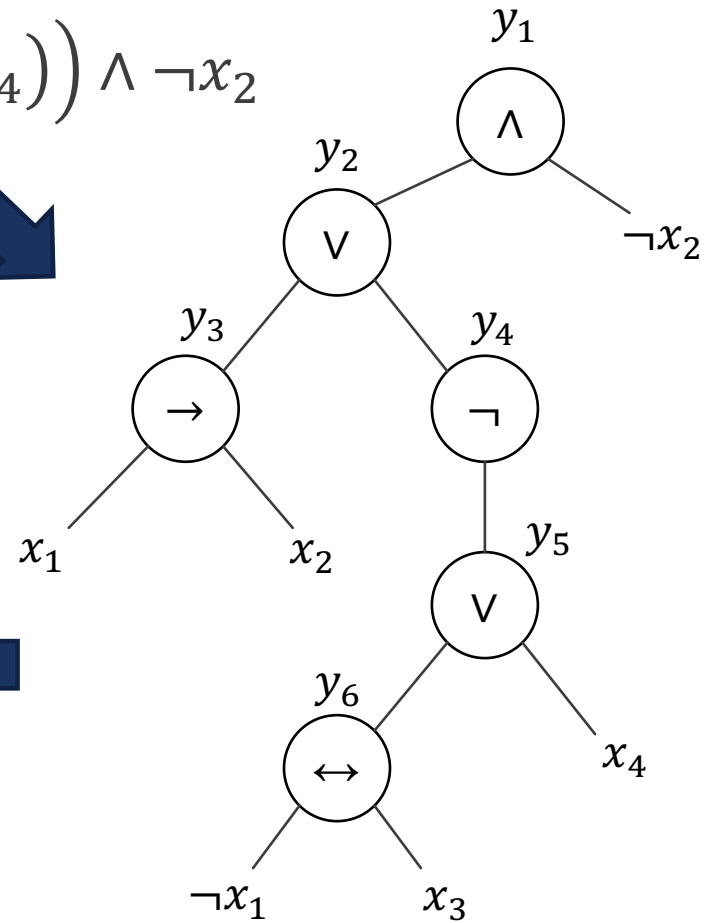
- The first step is to construct a binary "parse" tree for the input formula φ .
- The literals are leaves and connectives are internal nodes.
- Introduce a variable y_i for the output of each internal node.
- Then, we rewrite the original formula φ as the AND of the root variable and a conjunction of clauses describing the operation of each node.



NP-Completeness of 3-CNF Satisfiability

$$\varphi = \left((x_1 \rightarrow x_2) \vee \neg((\neg x_1 \leftrightarrow x_3) \vee x_4) \right) \wedge \neg x_2$$

$$\begin{aligned} \varphi' = & y_1 \wedge (y_1 \leftrightarrow (y_2 \wedge \neg x_2)) \\ & \wedge (y_2 \leftrightarrow (y_3 \vee y_4)) \\ & \wedge (y_3 \leftrightarrow (x_1 \rightarrow x_2)) \\ & \wedge (y_4 \leftrightarrow \neg y_5) \\ & \wedge (y_5 \leftrightarrow (y_6 \vee x_4)) \\ & \wedge (y_6 \leftrightarrow (\neg x_1 \leftrightarrow x_3)) \end{aligned}$$



φ' is conjunction of clause φ'_i . Each clause φ'_i has at most 3 literals.



NP-Completeness of 3-CNF Satisfiability

Proof (cont'd):

- The second step converts each clause φ_i into CNF.
- Construct a truth table for φ_i by evaluating all possible assignments to its variables.
- Build a formula in disjunctive normal form (DNF, 析取范式) that is equivalent to $\neg\varphi_i$.
- Convert $\neg\varphi_i$ into CNF φ_i by using DeMorgan's laws (德·摩根法则) to complement all literals and switch OR and AND.



NP-Completeness of 3-CNF Satisfiability

- For example:

$$\varphi_2' = y_1 \leftrightarrow (y_2 \wedge \neg x_2)$$

- The DNF formula equivalent to $\neg\varphi_2'$ is

$$(y_1 \wedge y_2 \wedge x_2) \vee (y_1 \wedge \neg y_2 \wedge x_2) \vee (y_1 \wedge \neg y_2 \wedge \neg x_2) \vee (\neg y_1 \wedge y_2 \wedge \neg x_2)$$

- Applying DeMorgan's laws, we get φ_2'' :

$$(\neg y_1 \vee \neg y_2 \vee \neg x_2) \wedge (\neg y_1 \vee y_2 \vee \neg x_2) \wedge (\neg y_1 \vee y_2 \vee x_2) \wedge (y_1 \vee \neg y_2 \vee x_2)$$

y_1	y_2	x_2	$(y_1 \leftrightarrow (y_2 \wedge \neg x_2))$
1	1	1	0
1	1	0	1
1	0	1	0
1	0	0	0
0	1	1	1
0	1	0	0
0	0	1	1
0	0	0	1



NP-Completeness of 3-CNF Satisfiability

Proof (cont'd):

- The final step further transforms the formula so that **each clause has exactly 3 distinct literals**.
 - If φ_i'' has 3 distinct literals, then simply include φ_i'' as clauses of φ''' .
 - If φ_i'' has 2 distinct literals l_1 and l_2 , that is, if $\varphi_i'' = (l_1 \vee l_2)$, then include $(l_1 \vee l_2 \vee p) \wedge (l_1 \vee l_2 \vee \neg p)$ as clauses of φ''' .
 - If φ_i'' has just 1 distinct literal l , then include $(l \vee p \vee q) \wedge (l \vee p \vee \neg q) \wedge (l \vee \neg p \vee q) \wedge (l \vee \neg p \vee \neg q)$ as clauses of φ''' .
- p and q can be arbitrary literal.



NP-Completeness of 3-CNF Satisfiability

Proof (cont'd):

- Thus, every formula can be transformed into 3-CNF φ''' , and φ''' is satisfiable if and only if φ is satisfiable.
- This reduction algorithm is polynomial time.
- Therefore, 3-CNF is NPC.



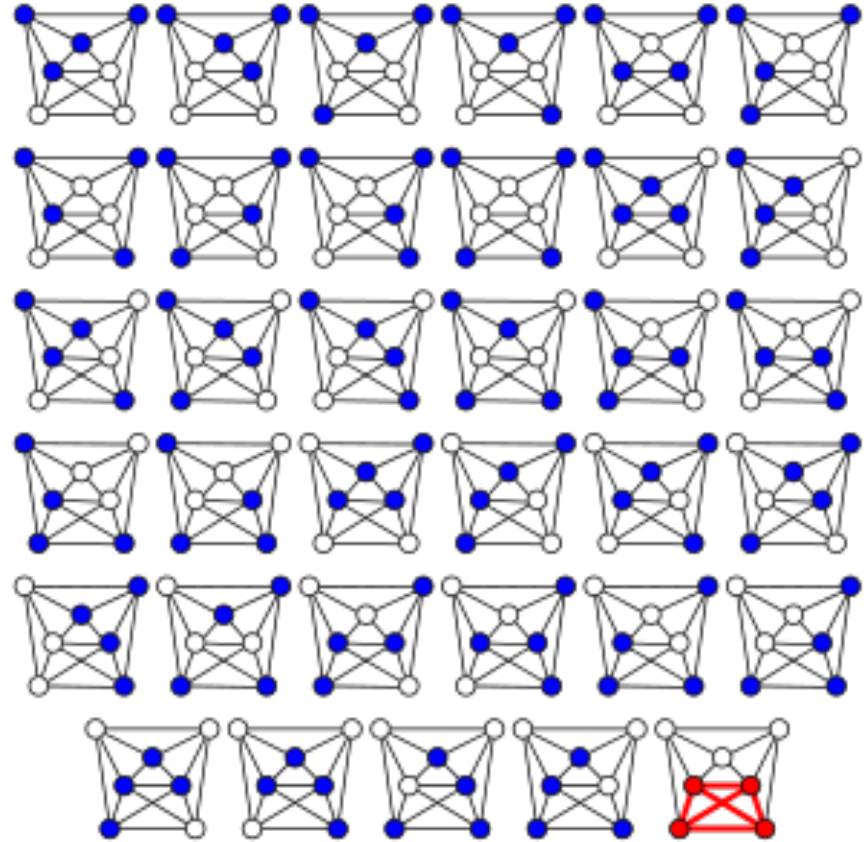
Clique Problem

- Now, we have known two NPC problem: SAT and 3-CNF-SAT. We can use them to prove more NPC problems.
- A **clique (团)** is a complete subgraph of an undirected graph $G = (V, E)$.
 - Each pair of vertex in $V' \subseteq V$ is connected by an edge in E .
 - The size of a clique is the number of vertices it contains.
- The clique problem is the optimization problem of finding a clique of maximum size in a graph.
- As a decision problem, we ask simply whether a clique of a given size k exists in the graph.



Clique Problem

- The brute force algorithm finds a 4-clique in this 7-vertex graph.
- Systematically check all $C(7,4) = 35$ 4-vertex subgraphs for completeness.



Clique Problem

Theorem 11.6

Clique is NP-complete.

Proof:

- **Step 1:** We first prove clique is in NP.
- For a given graph $G = (V, E)$, we use the set $V' \subseteq V$ of vertices in the clique as a certificate for G .
- Checking whether V' is a clique can be accomplished in polynomial time by checking whether $(u, v) \in E, \forall u, v \in V'$.
- Then, we use 3-CNF-SAT to prove clique problem.



Clique Problem

Proof (cont'd):

- **Step 2:** We next prove that $3\text{-CNF-SAT} \leq_P \text{Clique}$, which shows that the clique problem is NP-hard.
- That we should be able to prove this result is somewhat surprising, since logical formulas seem to have little to do with graphs.
- We need to construct a reduction algorithm that transforms any instance of 3-CNF-SAT into an instance of clique.



Clique Problem

Proof (cont'd):

- Let φ be a Boolean formula in 3-CNF with k clauses:

$$\varphi = C_1 \wedge C_2 \wedge \cdots \wedge C_k$$

- For $r = 1, 2, \dots, k$, each clause C_r has exactly three distinct literals l_1^r , l_2^r and l_3^r :

$$C_r = l_1^r \vee l_2^r \vee l_3^r$$

- We shall construct a graph G such that φ is satisfiable if and only if G has a clique of size k .

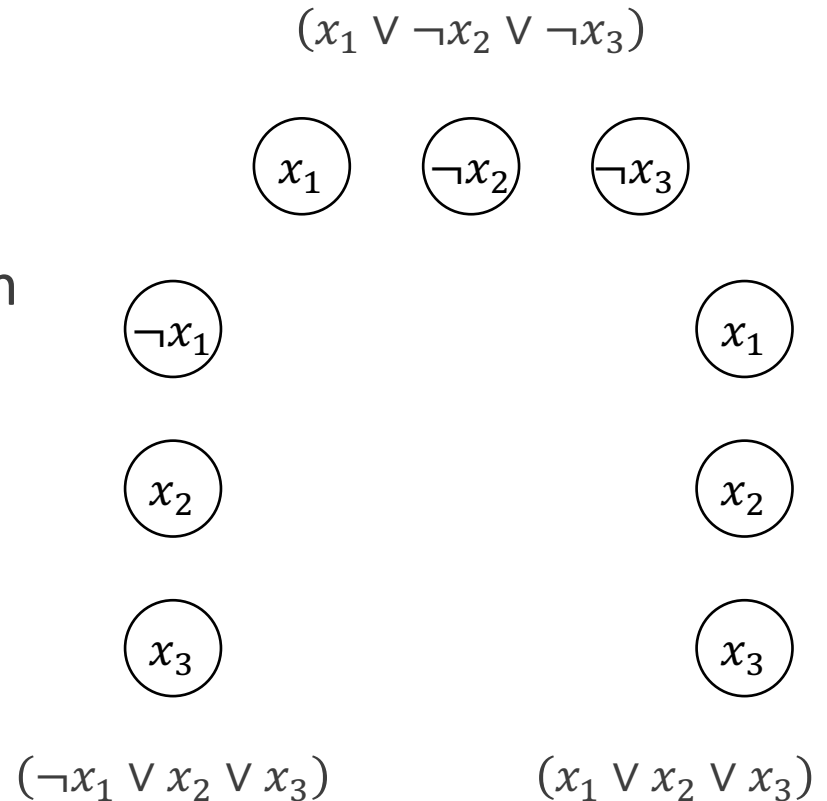


Clique Problem

Proof (cont'd):

- The graph $G = (V, E)$ is constructed as follows.
- For each clause $C_r = l_1^r \vee l_2^r \vee l_3^r$ in φ , we place a triple of vertices l_1^r , l_2^r and l_3^r into V .
- For example:

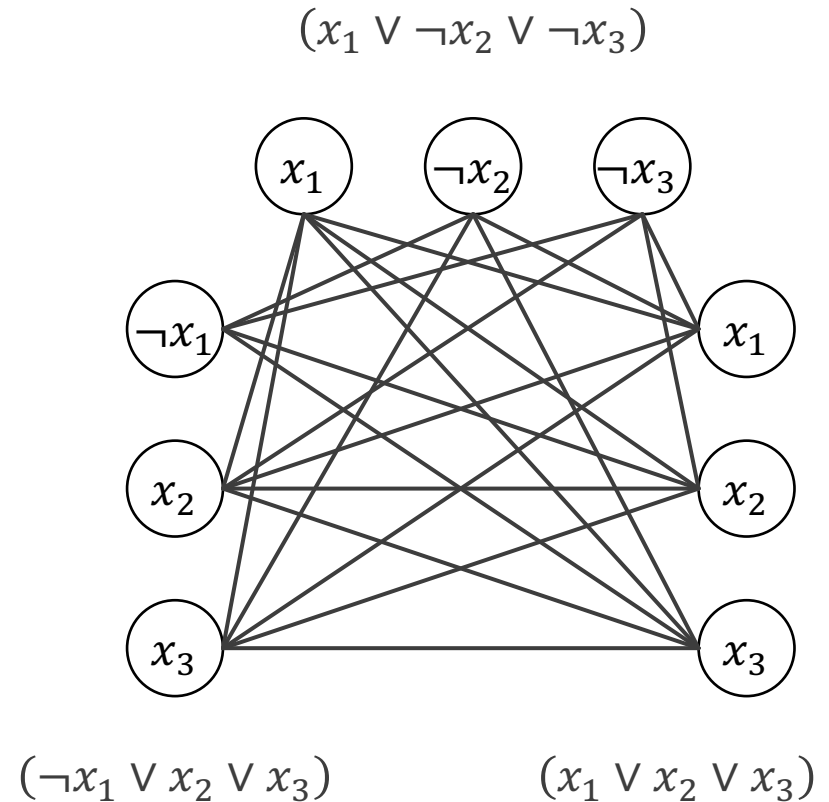
$$\begin{aligned} \varphi = & (x_1 \vee \neg x_2 \vee \neg x_3) \\ & \wedge (\neg x_1 \vee x_2 \vee x_3) \\ & \wedge (x_1 \vee x_2 \vee x_3) \end{aligned}$$



Clique Problem

Proof (cont'd):

- We put an edge between two vertices l_i^r and l_j^s if both of the following hold:
 - l_i^r and l_j^s are in different triples, that is, $r \neq s$.
 - l_i^r is not the negation of l_j^s .
- This graph can easily be computed from φ in polynomial time.



Clique Problem

Proof (cont'd):

- A satisfiable assignment is:

$$x_2 = 0, x_3 = 1$$

and x_1 can be either 0 or 1. The corresponding 3-clique is:

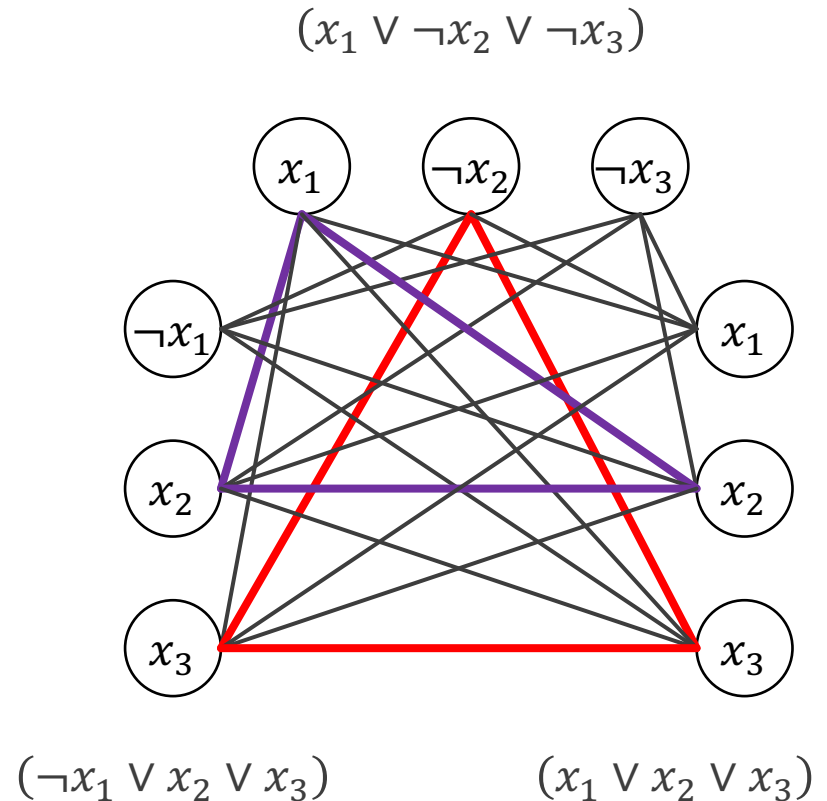
$$\{\neg x_2, x_3, x_3\}$$

- Another satisfiable assignment is:

$$x_1 = 1, x_2 = 1$$

and x_3 can be either 0 or 1. The corresponding 3-clique is:

$$\{x_1, x_2, x_2\}$$



Clique Problem

Proof (cont'd):

- We must show that this transformation of φ into G is a reduction, i.e. φ is satisfiable if and only if G has k -clique.
- We first prove \Rightarrow . Suppose that φ has a satisfying assignment.
- Then each clause C_r contains at least one literal that is assigned 1.

$$\varphi = \overbrace{(x_1 \vee \neg x_2 \vee \neg x_3)}^1 \wedge \overbrace{(\neg x_1 \vee x_2 \vee x_3)}^1 \wedge \overbrace{(x_1 \vee x_2 \vee x_3)}^1$$

- Picking one such "true" literal from each clause yields a set V' of k vertices.
- It is easy to show that V' is a clique by the construction of G .



Clique Problem

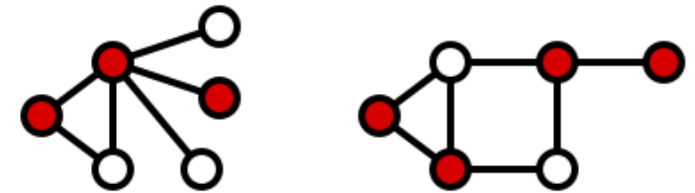
Proof (cont'd):

- Then we prove \Leftarrow . Suppose that G has a clique V' of size k .
- No edges in G connect vertices in the same triple, and so V' contains exactly one vertex per triple.
- Assigning 1 to each literal is safe, because G contains no edges between complementary literals.
- Each clause is satisfied, and so φ is satisfied.

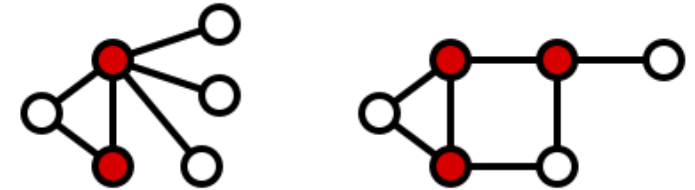


Vertex Cover Problem

- A **vertex cover** (顶点覆盖) of an undirected graph $G = (V, E)$ is a subset $V' \subseteq V$ such that if (u, v) is an edge of G , then either $u \in V'$ or $v \in V'$ (or both).
- The size of a vertex cover is the number of vertices in it.
- The vertex-cover problem is to find a vertex cover of minimum size in a given undirected graph.
- We can prove the completeness of vertex cover problem by showing $\text{Clique} \leq_P \text{VertexCover}$.



A vertex cover

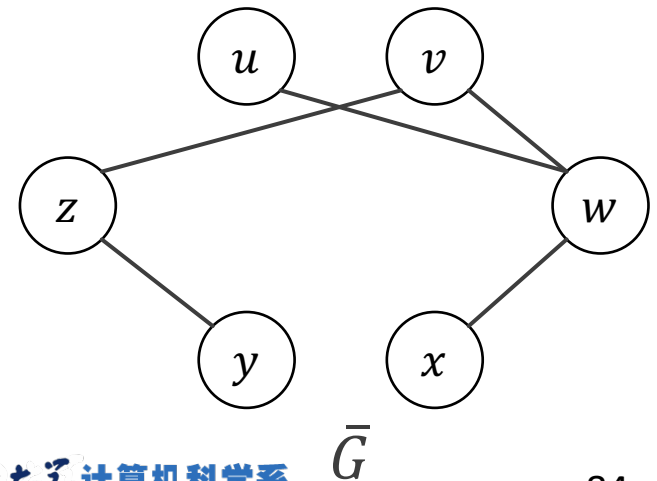
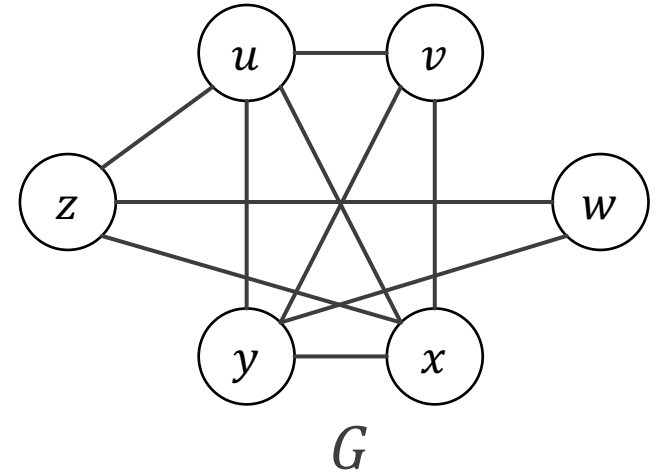


An minimum vertex cover



Vertex Cover Problem

- We first introduce the notion of **complement graph (补图)**.
- Given an undirected graph $G = (V, E)$, we define the complement of G as $\bar{G} = (V, \bar{E})$, where $\bar{E} = \{(u, v) : u, v \in V \text{ and } (u, v) \notin E\}$.



Vertex Cover Problem

Proof:

- We first show that VertexCover \in NP.
- Suppose we are given a graph $G = (V, E)$ and an integer k .
- The certificate we choose is the vertex cover $V' \subseteq V$ itself.
- The verification algorithm affirms that $|V'| = k$, and then it checks:

$$u \in V' \text{ or } v \in V', \forall (u, v) \in E$$

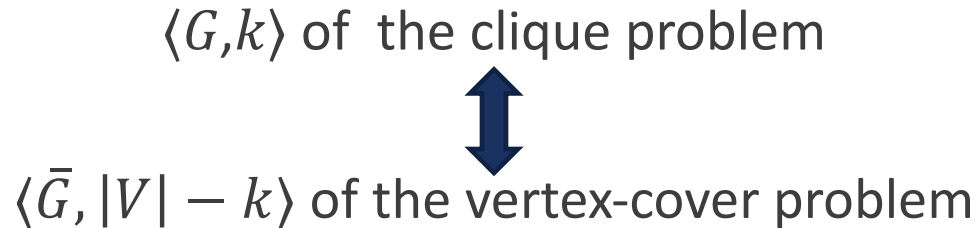
- This verification can be performed straightforwardly in polynomial time.



Vertex Cover Problem

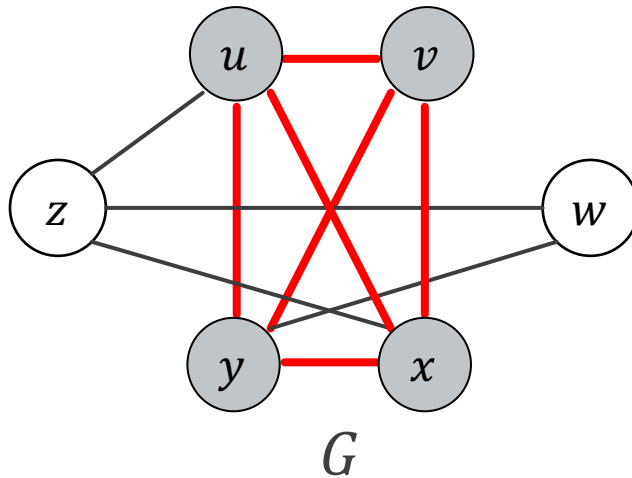
Proof (cont'd):

- The reduction algorithm takes as input an instance $\langle G, k \rangle$ of the clique problem.
- It computes the complement \bar{G} , which is easily done in polynomial time.
- The output of the reduction algorithm is the instance $\langle \bar{G}, |V| - k \rangle$ of the vertex-cover problem.
- To complete the proof, we show that this transformation is indeed a reduction:

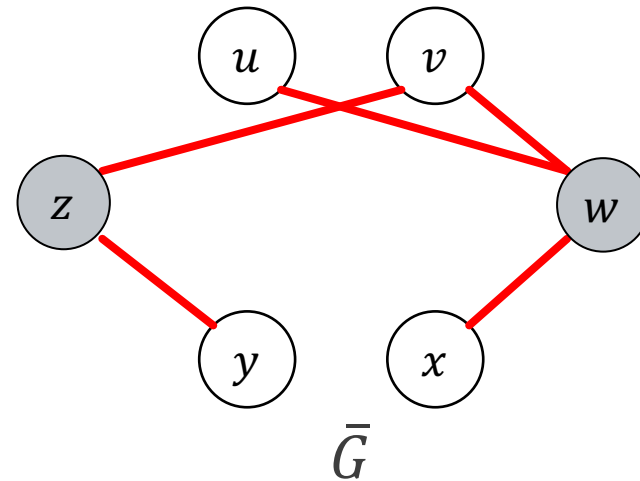


Vertex Cover Problem

Proof (cont'd):



Clique size = 4



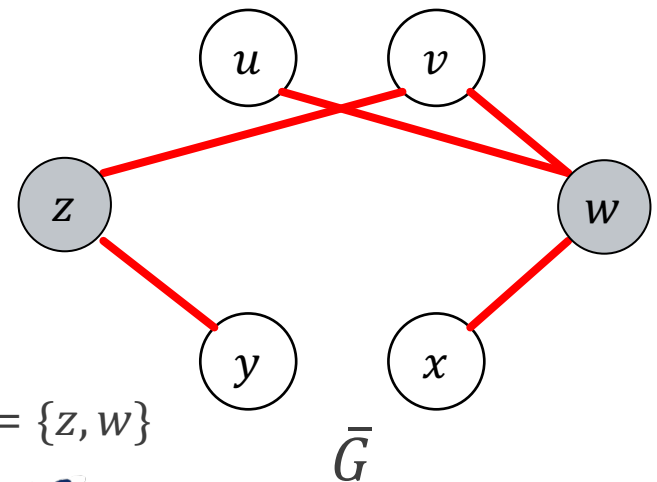
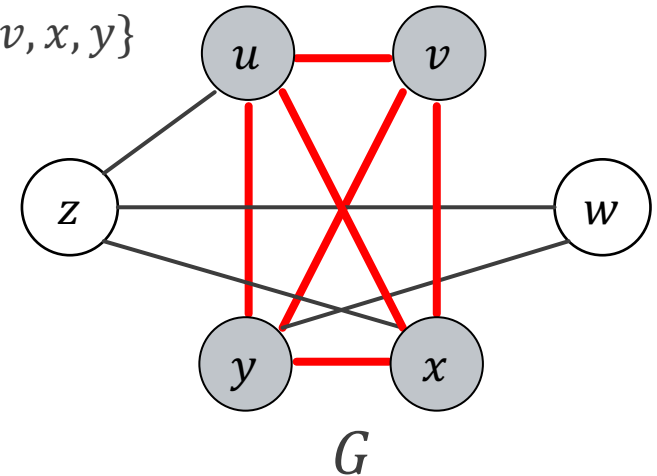
Vertex cover = $6 - 4 = 2$

Vertex Cover Problem

Proof (cont'd):

- We first prove \Rightarrow . Suppose that G has a clique $V' \subseteq V$ with $|V'| = k$.
- Since every pair of vertices in V' is connected by an edge of E , if $(u, v) \notin E$, which implies that at least one of u or v does not belong to V' .

$$V' = \{u, v, x, y\}$$



$$V - V' = \{z, w\}$$

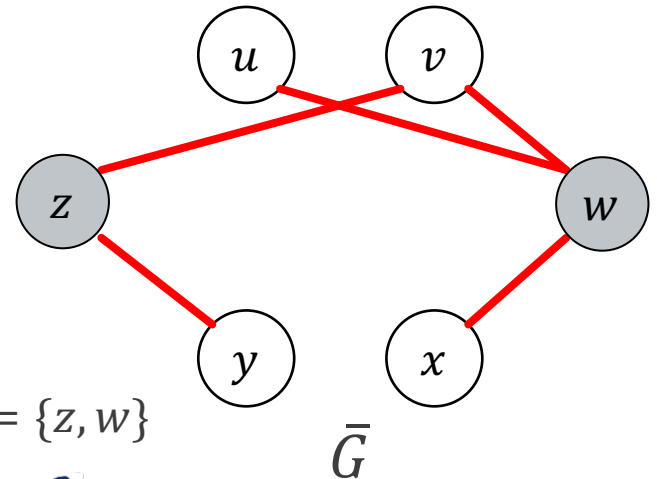
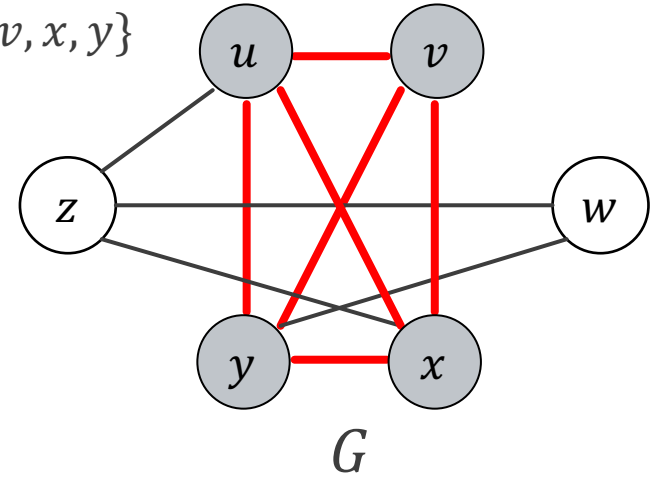


Vertex Cover Problem

Proof (cont'd):

- We get: $(u, v) \in \bar{E} \Rightarrow u \in V - V'$ or $v \in V - V'$
- It means that every edge $(u, v) \in \bar{E}$ is covered by $V - V'$.
- Hence, the set $V - V'$, which has size $|V| - k$, forms a vertex cover for \bar{G} .

$$V' = \{u, v, x, y\}$$



$$V - V' = \{z, w\}$$



Vertex Cover Problem

$$V - V' = \{u, v, x, y\}$$

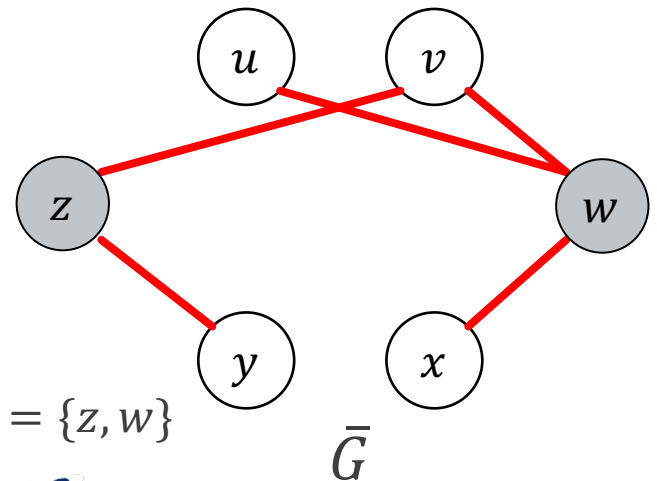
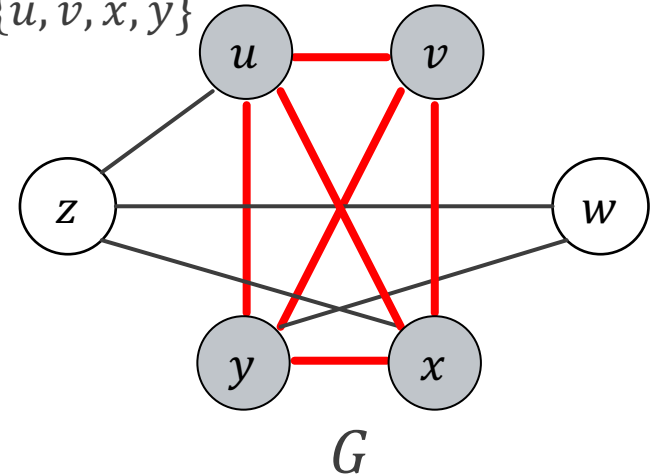
Proof (cont'd):

- Then we prove \Leftarrow . Suppose that \bar{G} has a vertex cover $V' \subseteq V$, where $|V'| = |V| - k$.
- Then, $\forall u, v \in V$:

$$(u, v) \in \bar{E} \Rightarrow u \in V' \text{ or } v \in V'$$

- The contrapositive of this implication is:

$$(u, v) \notin \bar{E} \Leftarrow u \notin V' \text{ and } v \notin V'$$



$$V' = \{z, w\}$$



Vertex Cover Problem

$$V - V' = \{u, v, x, y\}$$

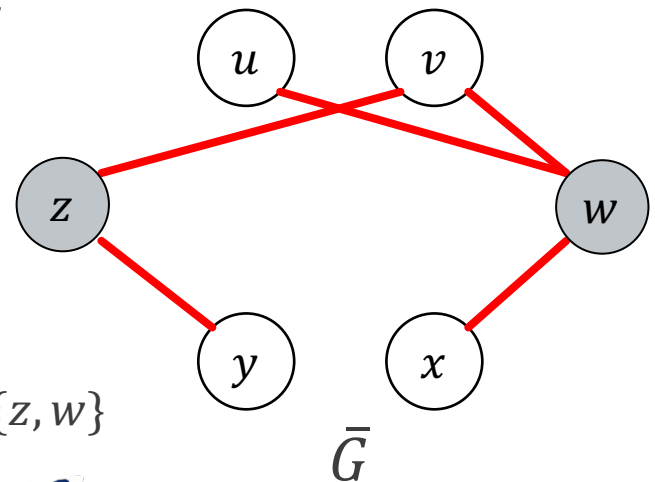
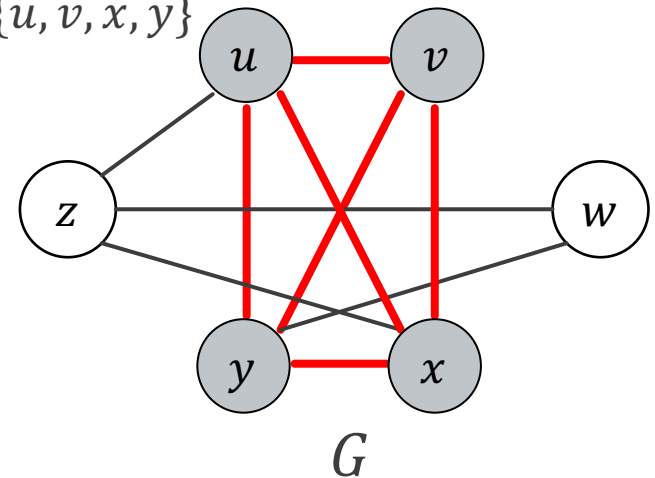
Proof (cont'd):

$$(u, v) \notin \bar{E} \iff u \notin V' \text{ and } v \notin V'$$



$$(u, v) \in E \iff u \in V - V' \text{ and } v \in V - V'$$

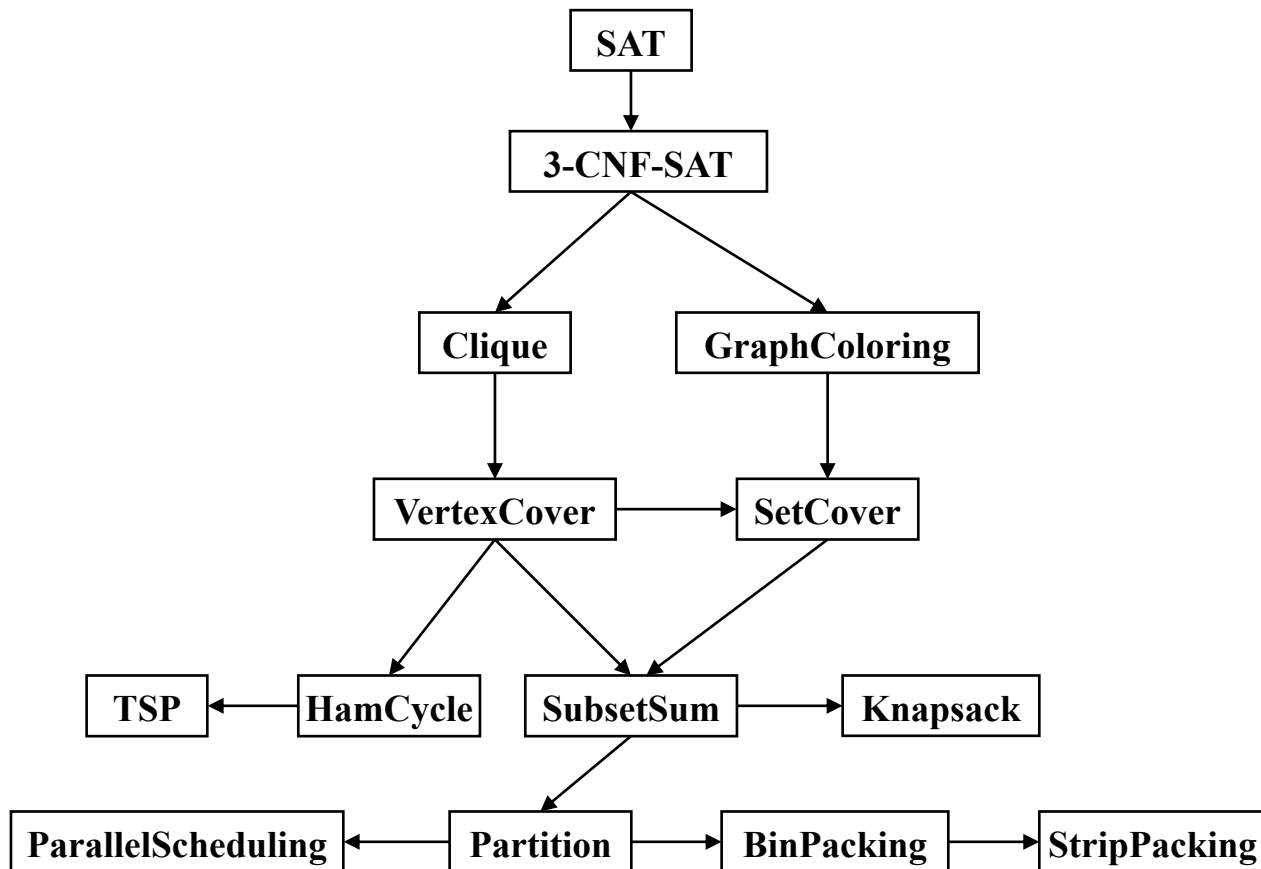
- In other words, $V - V'$ is a clique, and it has size $|V| - (|V| - k) = k$.



$$V' = \{z, w\}$$



NP-Completeness Proof Graph



Classroom Exercise

- The subset-sum problem (子集和问题) is defined as follows:
 - Given a sequence of integers $A = \{a_1, \dots, a_n, t\}$, determine whether there is a subset of the integers such that the sum is equal to t .
 - For example, given $\{a_1, a_2, a_3, a_4, a_5, t\} = \{1, 6, 4, 3, 2, 8\}$, we have $a_1 + a_3 + a_5 = 8$.
- The partition problem (划分问题) is defined as follows:
 - Given a sequence of integers $A = \{a_1, \dots, a_n\}$, determine whether there is a partition into two subsets such that their sums are equal.
 - For example, given $\{a_1, a_2, a_3, a_4, a_5\} = \{1, 6, 4, 3, 2\}$, we have $a_1 + a_3 + a_4 = a_2 + a_5 = 8$.
- If we know that subset-sum problem is NPC, prove that partition problem is NPC.



Classroom Exercise

Proof:

- **Step 1:** Show that partition decision problem is in NP by checking its verification stage in polynomial-time or not.
 - Input: A set of index S .
 - Output: Yes or No.
 - $sum_1 = \sum_{i \in S} a_i, sum_2 = \sum_{i=1}^n a_i - sum_1$.
 - If $sum_1 == sum_2$, return Yes; else return No.
- It is obviously polynomial-time.



Classroom Exercise

Proof (cont'd):

- **Step 2:** Prove that subset-sum \leq_P partition.
- Compare the two problems:
 - Subset-Sum: Given (a_1, \dots, a_n, t) , where t and all a_i are integers, whether there exists an answer $S \subseteq \{1, \dots, n\}$ such that $\sum_{i \in S} a_i = t$.
 - Partition: Given (a_1, \dots, a_n) , where all a_i are integers, whether there exists an answer $S \subseteq \{1, \dots, n\}$ such that $\sum_{i \in S} a_i = \sum_{j \notin S} a_j$.



Classroom Exercise

Proof (cont'd):

- Transform every instance of subset-sum to an instance of partition.
- Let $x = (a_1, \dots, a_n, t)$ be an instance of subset-sum and $a = \sum_{i=1}^n a_i$. We define the transformation algorithm as

$$f(x) = (a_1, a_2, \dots, a_n, a_{n+1})$$

where $a_{n+1} = 2t - a$.

- It is clear that this transform can be done in polynomial time.
- Then, we want to show that:

The answer S to subset-sum problem is “yes” for x
 \Leftrightarrow The answer S to partition problem is “yes” for $f(x)$.



Classroom Exercise

Proof (cont'd):

■ Prove \Rightarrow :

- Let the answer $S \subseteq \{1, 2, \dots, n\}$ to subset-sum problem is “yes” for x such that $\sum_{i \in S} a_i = t$ and $\sum_{i=1}^n a_i = a$.

- Let $T = \{1, 2, \dots, n+1\} - S$, we have

$$\sum_{j \in T} a_j = a + a_{n+1} - t = a + 2t - a - t = t = \sum_{i \in S} a_i.$$

- Because $\sum_{i \in S} a_i = \sum_{j \in T} a_j = \sum_{j \notin S} a_j$, the answer S to partition problem is also “yes” for $f(x)$.



Classroom Exercise

Proof (cont'd):

■ Prove \Leftarrow :

- If there exists the solution $S \subseteq \{1, 2, \dots, n\}$ to partition problem is also “yes” for $f(x)$, such that letting $T = \{1, 2, \dots, n + 1\} - S$ we have

$$\sum_{i \in S} a_i = \sum_{j \in T} a_j = \frac{a + (2t - a)}{2} = t.$$

- Without loss of generality, assume that $n + 1 \in T$, then we have $S \subseteq \{1, 2, \dots, n\}$ and $\sum_{i \in S} a_i = t$.
- Because $\sum_{i \in S} a_i = t$, the answer S to subset-sum problem is also “yes” for $f(x)$.



Conclusion

After this lecture, you should know:

- What is a polynomial-time algorithm.
- What is a decision problem.
- What are P and NP.
- How a problem can be reduced to another problem.
- What is an NP-complete problem.
- What is an NP-hard problem.
- How to prove NP-completeness of a problem.



Homework

Page 215-217

11.2

11.3

11.4

11.7

11.16



谢谢

有问题欢迎随时跟我讨论



厦门大学信息学院
SCHOOL OF INFORMATICS XIAMEN UNIVERSITY



厦门大学计算机科学系
Computer Science Department of Xiamen University