2025 Fall CSE5025

Combinatorial Optimization

组合优化

Instructor: 刘晟材

Lecture 2-1: Complexity, P, NP, Reductions, NP-Complete

The Quest for Efficient Algorithms

So far, we have seen some algorithms:

- Dynamic Programming, Greedy Algorithms, Branch-and-bound, etc.

Sometimes we can't find an efficient algorithm?

- Is the problem itself inherently “hard”?
- Or have we just not been clever enough?

Proving a problem has an efficient algorithm is ***easy***: just show the algorithm

Proving one doesn‘t exist is **much, much harder**

How can we prove the non-existence of something? We will now learn about **NP Complete (NPC)** Problems, which provide us with a way to approach this question

Facts about NP-Complete Problems

This is a very large class of thousands of practical problems for which

- it is not known if the problems have "efficient" algorithms
- it is known that if any one of the NP-Complete Problems has an efficient algorithm then all the NP-Complete Problems have efficient algorithms
- Researchers have spent innumerable man-years trying to find efficient algorithms to these problems and failing
- there is a large body of tools that often permit us to prove when a new problem is NP-Complete
- The problem of finding an efficient algorithm to an NP-Complete problem is known, in shorthand as $\boldsymbol{P = NP}$?

Agenda for Today's Lecture

In this lecture we will introduce the concepts that will permit us to discuss whether a problem is “hard” or “easy”

- Input size of problems
- Decision problems (判定问题)
- Polynomial time algorithms
- The Class P
- The Class NP
- Reductions (规约) between decision problems
- The Class NPC

Formalizing Hardness

To discuss if a problem is "easy" or "hard", we need a formal framework. This involves understanding three concepts

Input Size (输入规模): How do we measure the size of the problem?

Running Time (运行时间): How does the algorithm's time scale with input size?

Complexity Classes (复杂度类): How can we group problems by their difficulty?

Measuring Input size (1)

Standard Definition: The input size is the **minimum number** of **bits** (0 or 1) needed to **encode** the problem's input.

An algorithm's running time should be measured as a function of this bit-size.

Example: How do we encode graphs?

A graph $G = (V, E)$ may be represented by its adjacency matrix $A = [a_{ij}]$ $(a_{ij} \in \{0,1\})$.
Then G can be encoded as the binary string

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

of length n^2

Remark: The inputs of any problem can be encoded as binary strings.

Measuring Input size (2)

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

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 (合数)

Problem: 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?

Answer: Any integer $n > 0$ can be represented in the binary number system as:

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

and so be represented by the string $a_0 a_1 \dots a_k$ of length $\lceil \log_2(n+1) \rceil$.

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

Input Size Example: Sorting

Sorting Problem: Sort n integers $a_1, a_2, \dots, a_n$

Question: What is the input size of this problem?

Answer: Using fixed length encoding writes $\boldsymbol{a_i}$ as binary string of length

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

This coding gives inputs size nm

Running Time

Running times of algorithms, unless otherwise specified, should be **expressed in terms of input size (**minimal bit-size**)**.

For example, the naive algorithm for determining whether n is composite compares n against the first $n - 1$ numbers to see if any of them divides n. This makes $O(n)$ comparisons so it might seem linear and very efficient.

But note that the size of the problem is $size(n) = \log_2 n$ so the number of comparisons performed is actually $O(n) = O(2^{size(n)})$ which is exponential and not very good.

Why measuring in Binary bits?

Computers process and store all data as sequences of binary bits.

A number n is represented by approximately $\log_2 n$ bits.

Crucially, basic operations (addition, comparison, move, etc.) on numbers take time proportional to their bit-length in the worst case:

- Compare two numbers requires examine all bits in the worst case
- Adding two small numbers can be considered constant time
- Consider adding two 1000-bit numbers. This isn't a single CPU instruction; it involves multiple operations on smaller “chunks” of bits. Thus, the actual “work” scales with the number of bits

Decision Problems

Definition of Decision Problem: A *decision problem* is a question that has two possible answers, yes and no.

Definition of Optimization Problem: An *optimization* requires an answer that is an optimal configuration.

Remark: An optimization problem usually has a corresponding decision problem. Examples that we will see:

- Knapsack Problem vs. Decision Knapsack (DKP)
- Traveling Salesman Problem vs. Decision Traveling Salesman Problem (DTSP)

Decision Problems: DKP

We have a knapsack of capacity W and n items with weights w_1 $\dots, w_n$ and values v_1 $\dots, v_n$

Knapsack Problem (KP):

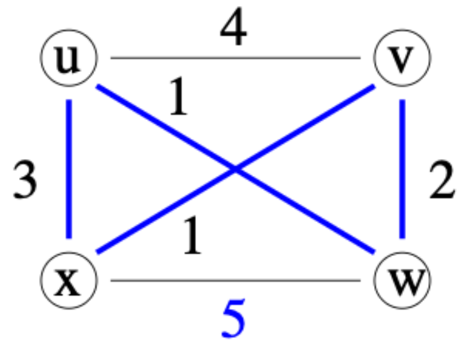
Goal: Select a subset of the items to put into the knapsack such that the sum of their weights does not exceed the capacity W, and the sum of their values is maximized

Decision Problem: Decision Knapsack (DKP)

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

Decision Problems: DTSP

Optimization Problem TSP: Given a complete weighted undirected graph with n vertices, find a Hamiltonian cycle of least weight.



Tour (Hamiltonian cycle) with minimum cost 7

Decision Problem: Decision TSP (DTSP)

Given a complete weighted undirected graph with n vertices, and a bound B, is there a Hamiltonian cycle of weight $\leq B$?

Optimization Problems 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.

Examples: If we know how to solve KP and obtain its optimal solution and then we check if the solution has value larger than k. If it does, answer Yes. Otherwise, answer No.

The reasons for studying decision problems here:

- 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.
- It will be more convenient to compare the "hardness" of decision problems than of optimization problems (because decision problems share the same output, yes or no.)

Problem and Problem Instances

	Problem	Problem Instance
Definition	A general description of a computational goal, specifying the relationship between a set of inputs and their desired outputs.	A specific realization of a problem, where all inputs are provided with concrete values.
Nature	Abstract, general, a template.	Concrete, specific, a filled-in template.
Example	**The Composite Number Problem: Input:** A positive integer, n - **Output:** "Yes" if n is composite, "No" otherwise.	**An Instance of the Composite Number Problem: - Input:** The integer n = 15. - **Output:** The answer "Yes".

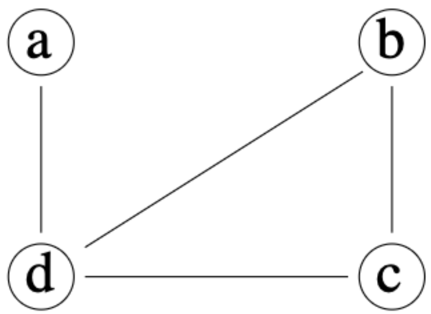
Decision Problems: Yes-Instances and No-Instances

Yes-Instance and No-Instance: An instance of a decision problem is called a yes-instance (resp. no-instance) if the answer to the instance is yes (resp. no).

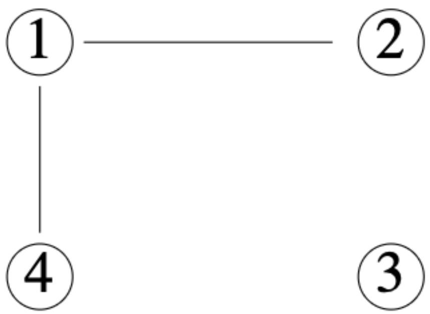
Note: Yes/no outputs are also noted as Yes/No-Inputs in the literature

CYC Problem: Does an undirected graph G have a cycle?

Example of Yes-Instances and No-Instances:



Yes-instance G



No-instance G

Decision Problems: Yes-Instances and No-Instances

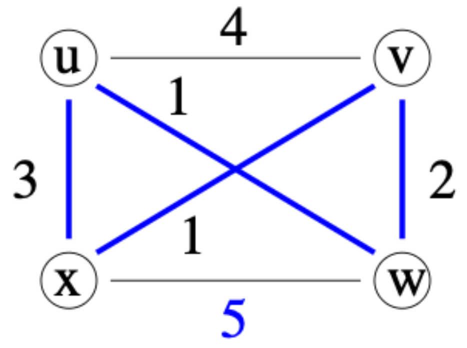
Decision Problem TRIPLE: Does a triple (n, e, t) of nonnegative integers satisfy $e \neq n - t$?

Example of Yes-Instances: (9, 8, 2), (20, 2, 17).

Example of No-Instances: (10, 8, 2), (20, 2, 18).

Decision Problems: Yes-Instances and No-Instances

DTSP: Given a complete weighted undirected graph with n vertices, and a bound B, is there a Hamiltonian cycle of weight $\leq B$?



Graph G

Tour (Hamiltonian cycle) with minimum cost 7

Example of Yes-Instances: (G, B= 10), (G, B= 9), (G, B= 7)

Example of No-Instances: (G, B= 1), (G, B= 2)

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 these classes;
- properties of problems in these classes.

Question: How to classify decision problems?

Answer: Use **polynomial-time algorithms**

Polynomial-Time Algorithms

Definition: An algorithm is *polynomial-time* if

- it solves every possible instances of the problem
- its running time is $O(n^k)$, where k is a constant independent of n, and n is the input size of the problem instance that the algorithm solves.

Remark: Whether you use n or n^a (for fixed $a > 0$) as the input size, it will not affect the conclusion of whether an algorithm is polynomial time.

Examples of Polynomial-Time Algorithms

Decision Sorting problem: Given n integers $a_1, a_2, \ldots, a_n$, is there a permutation of these numbers such that number of the "inversions" (pairs of elements that are in descending order) is 0.

An algorithm to solve this problem: Merge sort

- Input size $s = nm$
- it takes $O(nlogn)$ comparisons, every comparison takes $O(m)$ time
- Total running time: $O(mnlogn) = O(slogmn) \leq O(slogs) \leq O(s^2)$

Examples of Polynomial-Time Algorithms

Comparison of Computational Models for Algorithm Analysis

Aspect	Bit Complexity Model	Word RAM Model
Fundamental Unit	The **bit** is the basic unit of data and operation.	The **machine word** (e.g., 64 bits) is the basic unit.
Assumption on Number Size	Numbers can be arbitrarily large, represented by *m* bits.	Numbers have a fixed size that fits within one machine word.
Cost of a Comparison	**O(m)** (linear in the number of bits)	**O(1)** (a single, constant-time hardware operation)
Total Complexity (e.g., for Merge Sort)	**O(mn log n)**	**O(n log n)**
Primary Use Case	Theoretical Computer Science, cryptography, arbitrary-precision arithmetic (bignums).	Standard algorithm courses, competitive programming, and most practical software development.

当runtime取决于**输入的“数量”规模时，**通常两种模型分析出来的复杂度的量级是一致的.

Nonpolynomial-Time Algorithms (1)

Definition: An algorithm is *non-polynomial-time* if

- it solves every possible instances of the problem
- the running time is not $O(n^k)$ for any fixed $k \geq 0$.

Example: Let's examine the brute force algorithm for determining whether a positive integer n is a prime: it checks, in time $O((\log n)^2)$, whether k divides n for each k with $2 \leq k \leq n-1$. The complete algorithm therefore uses $O(n(\log n)^2)$ time.

Conclusion: The algorithm is nonpolynomial! Why? The input size is $s = \log_2 n$, and so $O(n(\log n)^2) = O(2^s s^2)$.

Nonpolynomial-Time Algorithms (2)

Recall the dynamic programming (DP) approach for solving KP (optimization version). Is this a polynomial algorithm?

No! The input size of an instance DP (denoted by *I*) is

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

Runtime of DP is $O(nW)$, which is not polynomial in $size(I)$. Depending upon the values of w_i and v_i, nW could be exponential in $size(I)$.

It is unknown as to whether there exists a polynomial time algorithm for KP.

In fact, DKP is a NP-Complete problem.

Polynomial vs. Nonpolynomial

Polynomial-time algorithms are usually considered "practical"

Problem Size (n)	Polynomial (n^3)	Exponential (2^n)	**Factorial ((n-1)!)**
10	1,000 ops (1 µs)	1,024 ops (1 µs)	**362,880 ops (0.36 ms)**
20	8,000 ops (8 µs)	~1 million ops (1 ms)	**~1.2 x 10^{17} ops (~3,857 years)**
30	27,000 ops (27 µs)	~1 billion ops (1 sec)	**~2.4 x 10^{30} ops (~7.7 x 10^{13} years)** *(约77万亿年)*
50	125,000 ops (0.1ms)	**~35.7 years**	**~6.1 x 10^{62} ops (远超宇宙年龄)**
60	216,000 ops (0.2ms)	**~36,559 years**	**~1.4 x 10^{80} ops (远超宇宙年龄)**

Note: in practice an $O(n^{20})$ algorithm is not really practical

Polynomial-Time Solvable Problems

Definition: A problem is solvable in polynomial time (or more simply, the problem is in polynomial time) if there exists an algorithm which solves every possible instances belong to the problem in polynomial time.

Examples: The **decision sorting problem**

Remark: Polynomial-time solvable problems are also called **tractable** (易解决的) problems.

The Class P

Definition: The class 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 instance is a yes-instance or a no-instance

How to prove that a decision problem is in class P?

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

How to prove that a decision problem is not in P?

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

The Class P: Some Examples

PATH (Reachability) Given a directed graph G and two vertices s and t, it asks if there is a path from s to t.

Linear Programming Feasibility Given a set of linear inequalities, this problem asks whether there exists a solution that simultaneously satisfies all of them.

PRIMES (Primality Testing) Given a positive integer n, this problem asks whether n is a prime number. This was famously proven to be in P by the AKS primality test in 2002.

Certificates and Verifying Certificates

We are now almost ready to introduce the class NP.

Before doing this we must first introduce the concept of **Certificates (证据)**.

Observation: A decision problem is usually formulated as:

Is there an object satisfying some conditions?

A **Certificate** is a specific object satisfying the conditions (exists only for yes-instances by definition).

Verifying a **certificate**: Check that the given object (certificate) satisfies the conditions (that is, verifying that the instance is yes-instance).

Certificates and Verifying Certificates: COMPOSITE

COMPOSITE: Is given positive integer n composite?

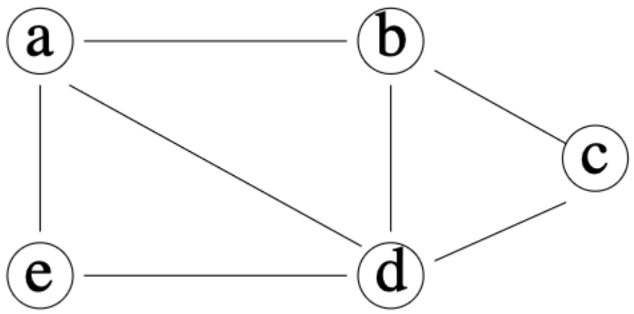
Certificate: an integer a dividing n such that $1 < a < n$.

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

Certificates and Verifying Certificates: DHamCyc

Hamiltonian Cycle: Given a graph $G = (V, E)$, a cycle of graph G is called Hamiltonian if it contains every vertex exactly once.

Example:



Find a Hamiltonian cycle for this graph

Decision problem DHamCyc: Does G have a Hamiltonian cycle?

DHamCyc: Verifying a Certificate

Certificate: an ordering of the n vertices in G (corresponding to their order along the Hamiltonian Cycle), i.e., $v_1, v_2, \ldots, v_n$.

Verification: Given a certificate the verification algorithm checks whether it is a Hamiltonian cycle of G by simply checking whether all the edges $(v_1, v_2), (v_2, v_3), \ldots, (v_{n-1}, v_n), (v_n, v_1)$ appear in the graph.

This can be done in $O(n)$ time so this is polynomial.

The Class NP

Definition: The class NP consists of all **decision problems** such that, for **each yes-instance,** there exists a **certificate** that can be **verified** in **polynomial time**.

Example: DKP ∈ NP.

Example: DHamCyc ∈ NP. (As shown earlier, there is a polynomial time algorithm to verify a certificate.)

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

$P = NP$ or $P \neq NP$

One of the most important problems in computer science is P = NP?

Put another way, is every problem that can be **verified** in polynomial time also **decidable** in polynomial time?

At first glance it seems "obvious" that P ≠ NP; after all, deciding a problem is much more restrictive than verifying a certificate.

However, 50 years after the P = NP? problem was first proposed, still no answer. The search for a solution has provided us with deep insights into what distinguishes an "easy" problem from a "hard" one.

SAT (1)

We will now introduce **Satisfiability (SAT),** which is one of the most important NP problems

Definition: A **Boolean formula** is a logical formula which consists of

boolean variables (0=false, 1=true),

logical operations

$\bar{x}$, NOT,

$x \vee y$, OR,

$x \wedge y$, AND.

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

SAT (2)

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 true (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.

SAT (3)

A given Boolean formula is ***not satisfiable*** if there is **no assignment** of truth values (0 or 1) to the variables such that the final result is true (1).

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.

Claim: SAT ∈ N P.

Proof: The evaluation of a formula of length n (counting variables, operations, and parentheses) requires at most n evaluations, each taking constant time. Hence, to check a certificate takes time $O(n)$.

k-SAT ∈ NP

For a fixed k, consider Boolean formulas in k-conjunctive normal form (k-CNF, 合取范式): $f_1 \wedge f_2 \wedge \cdots \wedge f_n$, 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 $y_{i,1}$ is a variable or the negation of a variable.

An example of a 3-CNF formula is:

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

***k*-SAT problem**: Determine whether a Boolean k-CNF formula is satisfiable.

Claim: 3-SAT ∈ NP

Claim: 2-SAT ∈ P

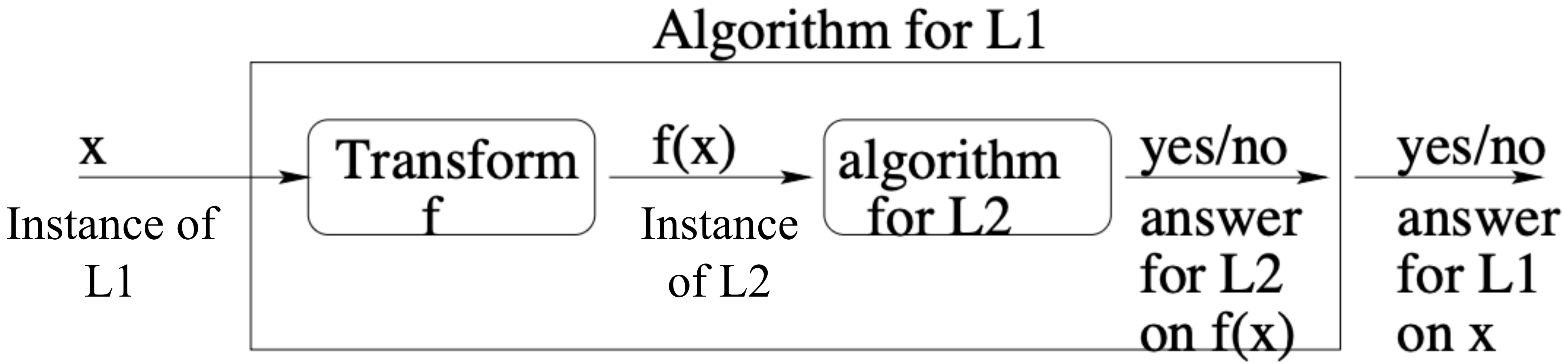
Reductions between Decision Problems

We have seem some decision problems, since we want to group problems better, can we formalize some kind of relationship between them?

What is Reduction? Let L_1 and L_2 be two decision problems. Suppose algorithm A_2 solves L_2. That is, if x is an instance of L2 then algorithm A_2 will answer Yes or No to x correctly.

The idea is to find a transformation f from L_1 to L_2 so that the algorithm A_2 can be part of an algorithm A_1 to solve L_1

Reductions between Decision Problems



Polynomial-Time Reductions (1)

Definition: A Polynomial-Time Reduction from L_1 to L_2 is a transformation f with the following properties:

- f transforms an instance x of L_1 into an instance $f(x)$ for L_2 s.t.

  $f(x)$ is a yes-instance for L_2 if and only if x is a yes-instance for L_1.
- $f(x)$ is computable in polynomial time (in size(x)).

If such an f exists, we say that $\boldsymbol{L_1}$ **is polynomial-time reducible (多项式时间内可规约) to** $\boldsymbol{L_2}$, and write:

$$\boldsymbol{L_1 \leq_P L_2}.$$

Implications: If $L_1 \leq_P L_2$, then L_1 is no harder than L_2

Polynomial-Time Reductions (2)

Question: What can we do with a polynomial time reduction f: $L_1 \leq_P L_2$?

Answer: Given an algorithm A_2 for the decision problem L_2, we can develop an algorithm A_1 to solve L_1.

In particular (proof on next slide)

- if A_2 is a **polynomial time algorithm for L_2** and $\boldsymbol{L_1 \leq_P L_2}$
- then we can construct **a polynomial time algorithm for L_1**

Polynomial-Time Reductions (3)

Theorem:

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

Proof: $L_2 \in$ P means that 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 instance x of L_1 to an instance of L_2. Combining these, we get the following polynomial-time algorithm for solving L_1 :

(1) take instance x of L_1 and compute $f(x)$;

(2) run A_2 on instance $f(x)$ 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 $L_1 \in$ P .

Polynomial-Time Reductions (3)

WARNING

We have just seen

Theorem:

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

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.

Transitivity (传递性) of Polynomial Reduction

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-instance for L_1 if and only if $f(x)$ is a yes-instance 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 $L_1 \leq_P L_3$.

Example of Polynomial-Time Reduction (1)

Let's look at two famous problems on a graph G = (V, E) with *n* vertices.

- **VERTEX COVER (VC):**
  - **Question:** Does G have a "vertex cover" of size at most *k*?
  - **Definition:** A *vertex cover* is a subset of vertices S ⊆ V such that every edge in E is touched by at least one vertex in S.
- **INDEPENDENT SET (IS):**
  - **Question:** Does G have an "independent set" of size at least *k*?
  - **Definition:** An *independent set* is a subset of vertices S' ⊆ V such that no two vertices in S' are connected by an edge.

Example of Polynomial-Time Reduction (2)

The Reduction: Vertex Cover $\leq_p$ Independent Set

We will show how to solve any Vertex Cover problem if we have a "magic box" (oracle) that can solve Independent Set.

The Core Claim

For any graph G with *n* vertices:

A set of vertices **S** is a **Vertex Cover** if and only if its complement **V \ S** (all vertices not in S) is an **Independent Set**.

This means: G has a Vertex Cover of size *k* **if and only if** G has an Independent Set of size *n* - *k*.

Example of Polynomial-Time Reduction (3)

The Reduction Algorithm

1. **INPUT:** An instance of Vertex Cover: a graph **G** and an integer **k**.
2. **TRANSFORMATION:** Create an instance of Independent Set:
   - The graph is the same: **G' = G**.
   - The new target size is **k' = n - k**.

   This transformation takes polynomial time
3. **SOLVE:** Ask the Independent Set oracle: "Does **G'** have an Independent Set of size at least **k'**?"
4. **OUTPUT:** If the oracle says YES, we answer YES to the original Vertex Cover problem. If it says NO, we answer NO.

Example of Polynomial-Time Reduction (4)

Why the Reduction Works & Its Implications

Proof of Correctness

Why is a set **S** a Vertex Cover iff its complement **V \ S** is an Independent Set?

- **(⇒) If S is a Vertex Cover, then V \ S is an Independent Set.**
  - Assume S is a vertex cover. Take any edge (u, v). By definition of VC, at least one of *u* or *v* must be in S.
  - This means it's impossible for *both u and v* to be in the complement set V \ S.
  - Since no two vertices in V \ S are connected by an edge, V \ S is an independent set.
- **(⇐) If V \ S is an Independent Set, then S is a Vertex Cover.**
  - Assume V \ S is an independent set. Take any edge (u, v).
  - By definition of IS, it's impossible for *both u and v* to be in V \ S.
  - Therefore, at least one of *u* or *v* must be in S.
  - Since this holds for every edge, S is a vertex cover.

Finally, the Class NP-Complete (NPC)

We have finally reached our goal of introducing the class NPC

Definition: The class NPC of **NP-complete problems** consists of all decision problems L such that

$$(a)\ L \in NP;$$

$$(b)\text{for every } L' \in NP, \qquad L' \leq_P L$$

Intuitively, NPC consists of all the **hardest** problems in NP

Note: it is not obvious that there exist any such problems at all. There are an infinite number of problems in NP. How can we prove that some problem is at least as hard as all of them? We will that there are actually many such problems in the next lecture.

NP-Completeness and Its Properties

The major reason we are interested in NP-Completeness is the following theorem which states that either all NP-complete problems are polynomial time solvable or all NP-Complete problems are not polynomial time solvable.

Theorem: Suppose that L is $\mathcal{NPC}$.

- If there is a polynomial-time algorithm for L, then there is a polynomial-time algorithm for every $L' \in \mathcal{NP}$.

  Proof: **By the previous theorem**

- If there is no polynomial-time algorithm for L, then there is no polynomial-time algorithm for any $L' \in \mathcal{NPC}$.

Why?

Proving that Problem L is NPC

Two steps:

(a) Show L ∈ N P.

(b) Show that $L' \leq_P L$ for a suitable $L' \in \mathrm{NPC}$

Question 1: How do we get one problem in NPC to start with?

Answer: We need to prove, **from scratch** that one problem is in NPC.

Question 2: Which problem is "suitable"?

Answer: There is no general procedure to determine this. You have to be knowledgeable, clever and (some-times) lucky.

Brief History of NP-Complete Problems (1)

The Cornerstone: The First NP-Complete Problem

Problem: Boolean Satisfiability Problem (SAT)
Proven By: Stephen Cook & Leonid Levin (1971)

Contribution: They independently proved the famous **Cook-Levin Theorem**.

Significance: The theorem states that **any problem in NP can be reduced in polynomial time to SAT**. This established SAT as the original “hardest” problem in NP and the foundational cornerstone of NP-Completeness theory.

Brief History of NP-Complete Problems (2)

The Expansion: Karp's 21 NP-Complete Problems (1972)

Core Contribution: Building on Cook's work, Karp published a landmark paper, *"Reducibility Among Combinatorial Problems."* Using **polynomial-time reductions** from SAT, he proved that 21 other fundamental combinatorial problems were also NP-Complete.

Problem Name	Description
3-SAT	A special case of SAT where each clause has exactly 3 literals.
Clique	Does a graph contain a fully connected subgraph of at least *k* vertices?
Vertex Cover	Does a graph have a set of *k* vertices that touches every edge?
Hamiltonian Cycle	Does a graph contain a simple cycle that visits every vertex exactly once?
0/1 Knapsack	(Decision Version) Can a subset of items be chosen that meets a value goal without exceeding a weight limit?
Set Cover	Can a collection of *k* sets be chosen whose union covers a "universe" of elements?

The Classes P, NP, and NPC

Proposition: P ⊆ NP

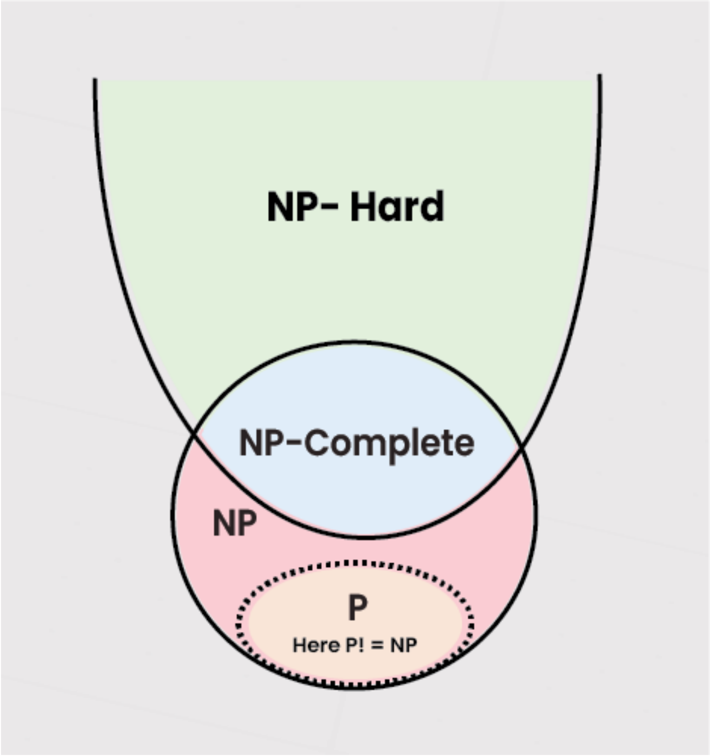
Simple proof omitted

Question 1: Is NPC ⊆ NP?

Yes, by definition!

Question 2: Is P = NP?

Open problem! Probably very hard



Question 3: are there problems that are even harder than NP-complete problems?

Yes! NP-hard problems, and we will touch it in the next lecture.

Conclusions

In this lecture, we have introduced the following key concentps:

- Input size of problems
- Decision problems (判定问题)
- Polynomial time algorithms.
- The Class P, NP, NPC, reductions between decision problems

In the next lecture, we will show

- How to prove a problem is NPC
- The Class NP-Hard
- Optimization problems vs. Decision Problems
- How to prove an optimization problem is NP-Hard