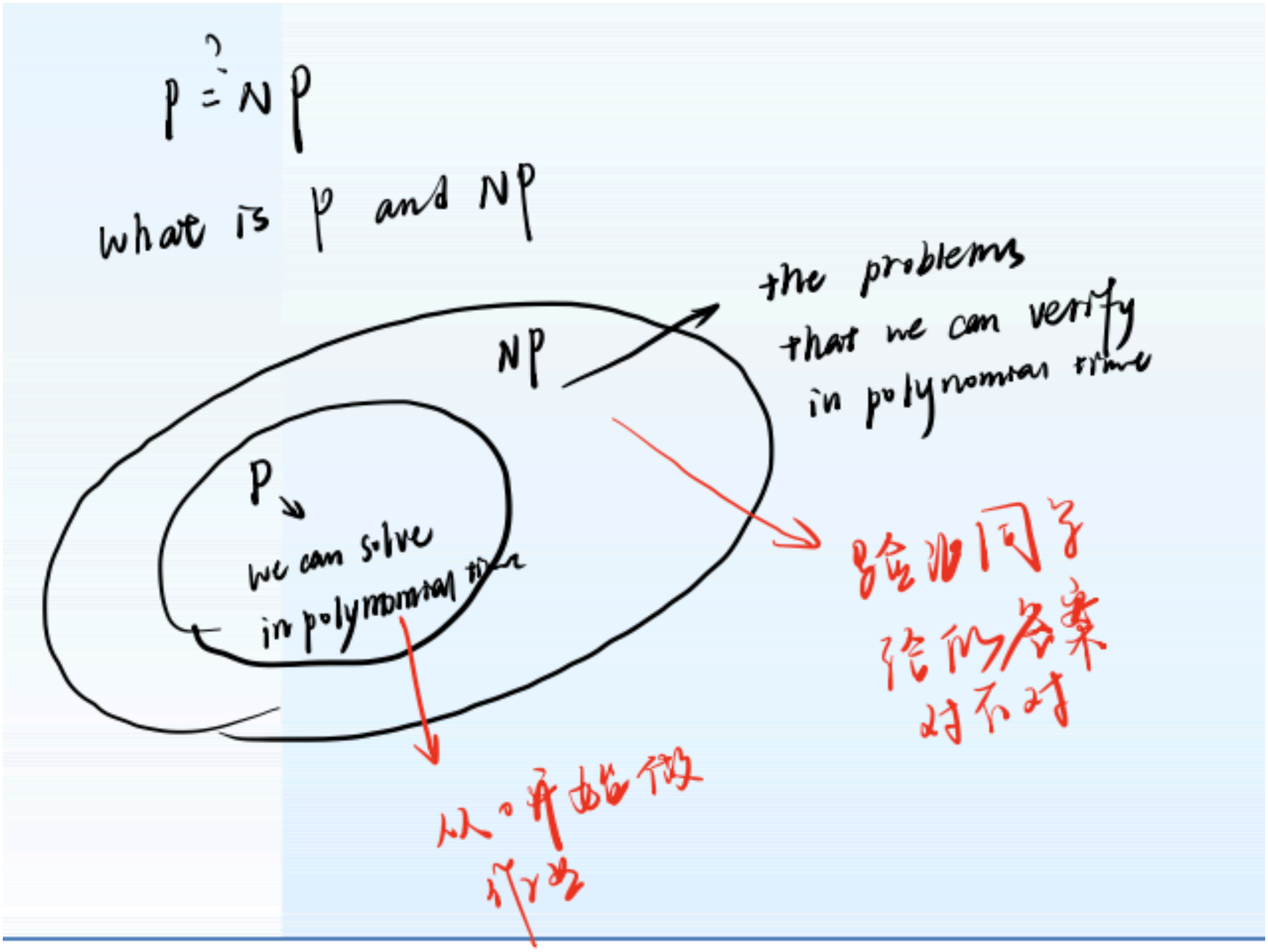
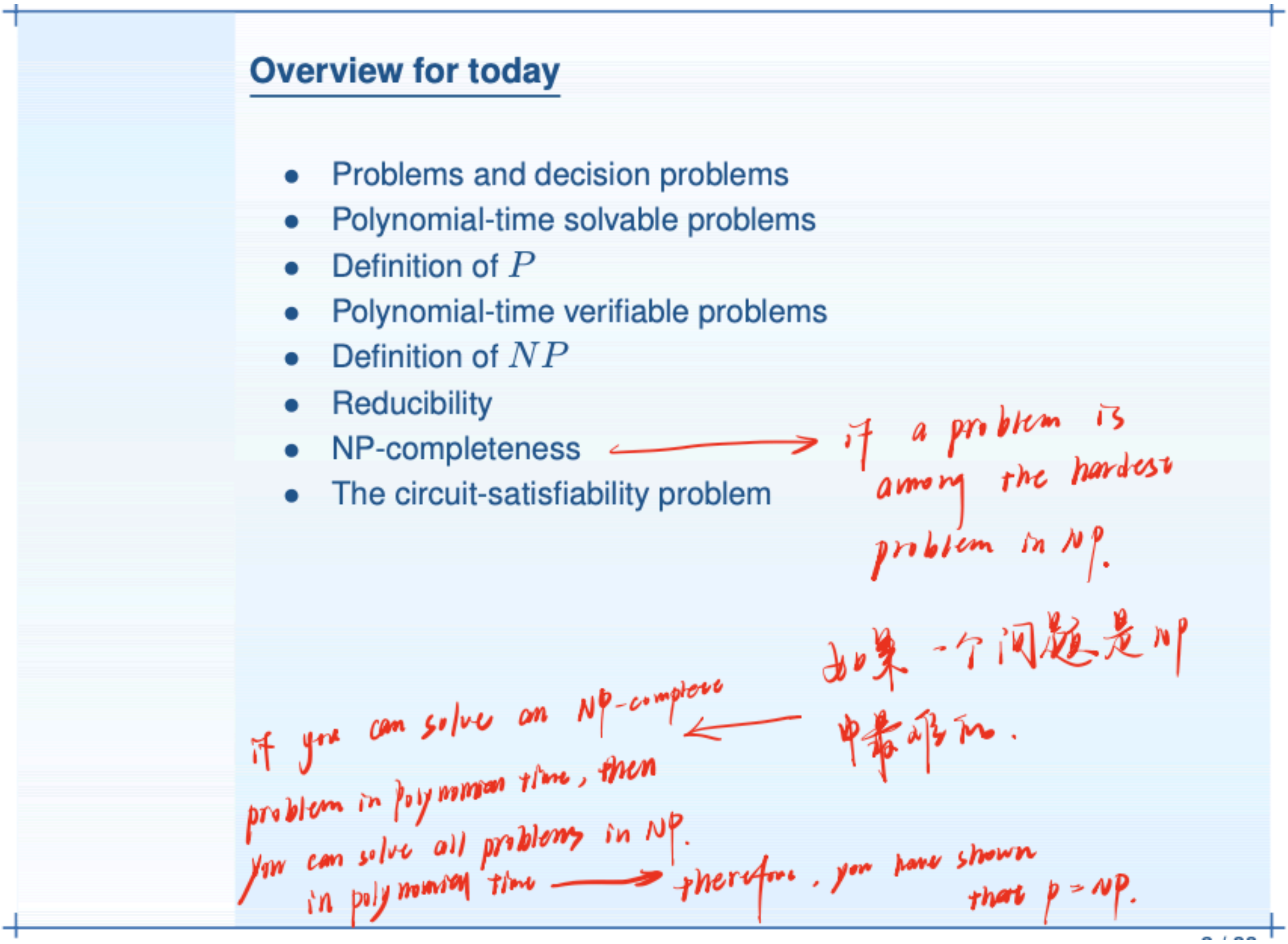
NPC

- ***Overview for today***



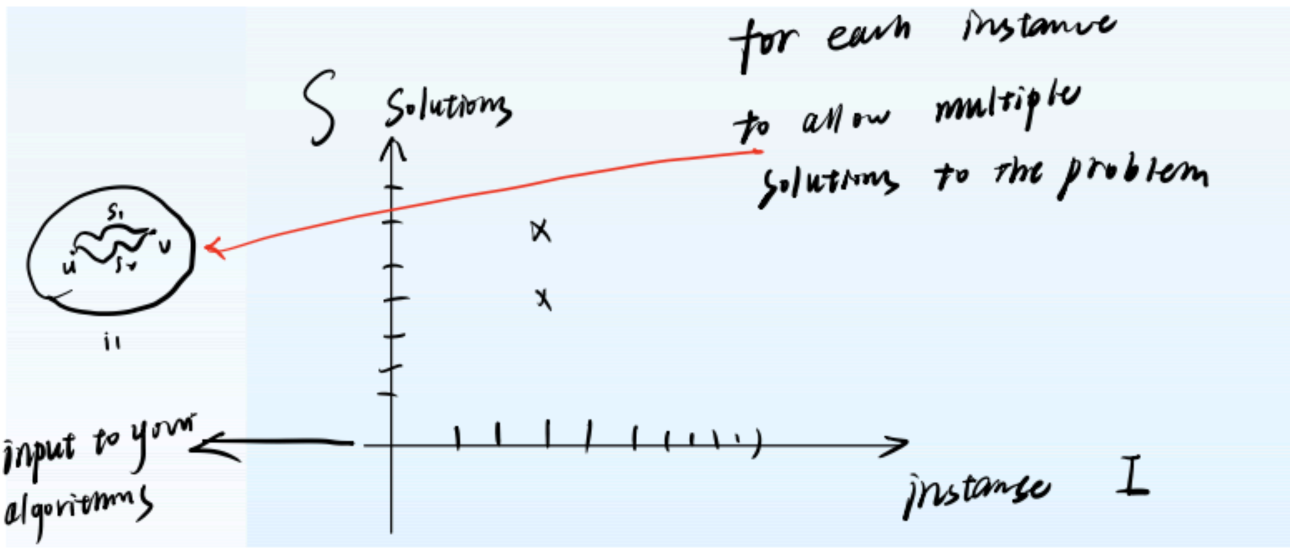
Reducibility => Show that one problem is essentially at least as hard as another problem.



- ***Definition of a problem***

1. Consider a set I of *instances* and a set S of *solutions.*
2. An abstract *problem* is *a binary relation between* I *and* S*,* i.e., a subset of $I \times S$.

   For $SHORTEST-PATH$, an instance is a triple $\langle G, s, t\rangle$.



A solution is a sequence of vertices forming a shortest s to t path.

- ***Decision problems***

Problems with $1/0$ (yes/no) answers. Hence, $S = \{0, 1\}$.

the algorithm should
output either 0/1
yes/no,
instead of saying it is
a shortest path

Example of a decision problem: $PATH$.

$PATH(\langle G, u, v, k\rangle) = 1$ if there is a $u - to - v$ path in G with at most k edges. Otherwise, $PATH(< G, u, v, k >) = 0$.

We can regard a decision problem as a mapping from instances to $S = \{0, 1\}$.

$<G, u, v, k> \rightarrow \{0, 1\}$

Instances with solution 1 are called *yes*-instances. Instances with solution 0 are called *no*-instances.

Optimization problems (like SHORTEST-PATH) can usually be turned into decision problems (like PATH).

> Optimization → Decision problem
> 之间的关联？如果已经有一个在poly时间内运行的path算法，如何得到一个poly内找shortest path的算法？
> decrement k until no instance
> 4 / 33

如果有了一个算法(PATH)，我们可以
从 $k=|V|-1$ 开始，如果 $s=1$，则 $k-1$
直到 $s=0$ → 最短PATH 算法　　持续run PATH 算法
↓
最小的 $s=1$ 的 k ↗
但只能找到 distance 而非 PATH

- ***Polynomial-time solvable problems => Class of P***

能在多项式时间内解决的。

1. We assume that ***instances*** of a problem are ***encoded as binary strings***.
2. An algorithm ***solves*** a problem in time $O(T(n))$ if for any instance of length n, the algorithm returns a solution (0 or 1) in time $O(T(n))$.

在 $O(T(n))$ 内解决任何输入长度为 n 的问题，输出为 $\{0,1\}$ 的算法。
在 $O(T(n))$ 内

3. If $T(n) = O(n^k)$ for some constant k, the problem is *polynomial-time solvable*.

Suppose we define P as the class of polynomial-time solvable problems.

- What is missing in this definition? Which encoding of the input is assumed?

⇓

We haven't specified how we encode the input?
很重要，因为我们的运行时间取决于 input size
而 input size 取决于我们如何表示

In the lecture, we pick binary encoding, giving input size $n = \lfloor \lg k \rfloor + 1$.

- In this case, running time is $\Theta(k) = \Theta(2^n)$ which is exponential in the input size.

!!! Remember that we need to specify how do we represent our input.

- These two ways of encoding k correspond to two different problems.

☆ 当说一个问题时也要说如何 encode 一个问题的输入们.

In particular, numbers are represented in ***binary***, not unary.

We use the notation $< x >$ to refer to a chosen encoding of instance x of a problem. => Already converted to a binary string.

Encodings are always ***binary*** strings in our setting.

- ***Languages***

Alphabet: finite set $\sum$ of symbols.

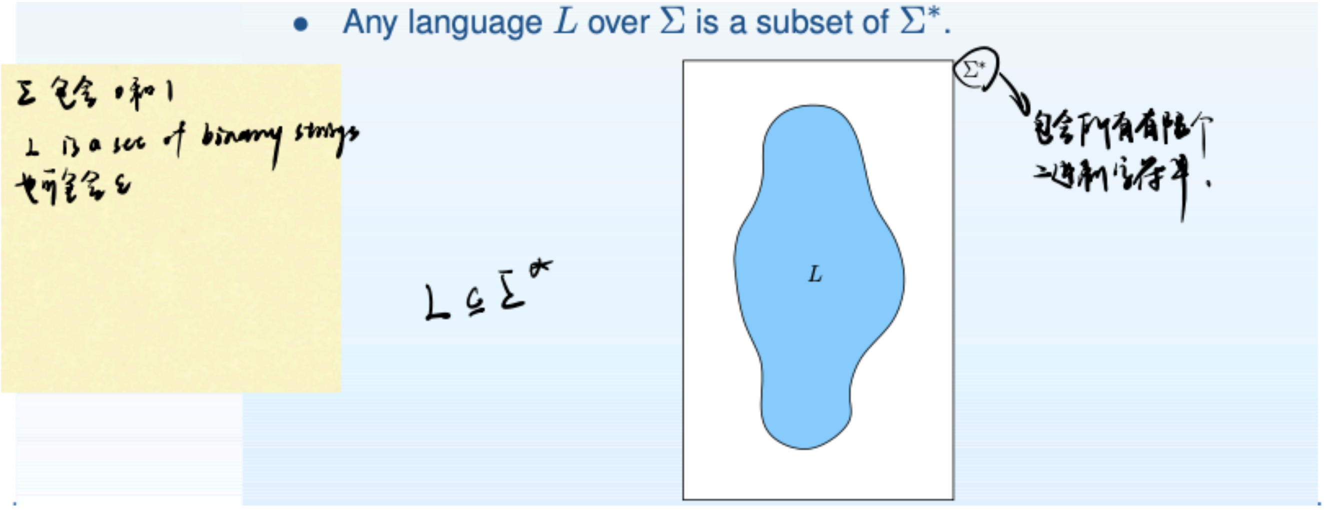
Language L over $\sum$: a set of strings of symbols from $\sum$.

Example: $\sum = \{a, b, c\}$ and $L = \{a, ba, cab, bbac, \ldots\}$.

We also allow an empty string and denote it by ϵ. => $\epsilon \in L$

The empty language is denoted $\emptyset$ (It does not contain ϵ).

$\sum^*$ denotes the language of all strings (including ϵ).

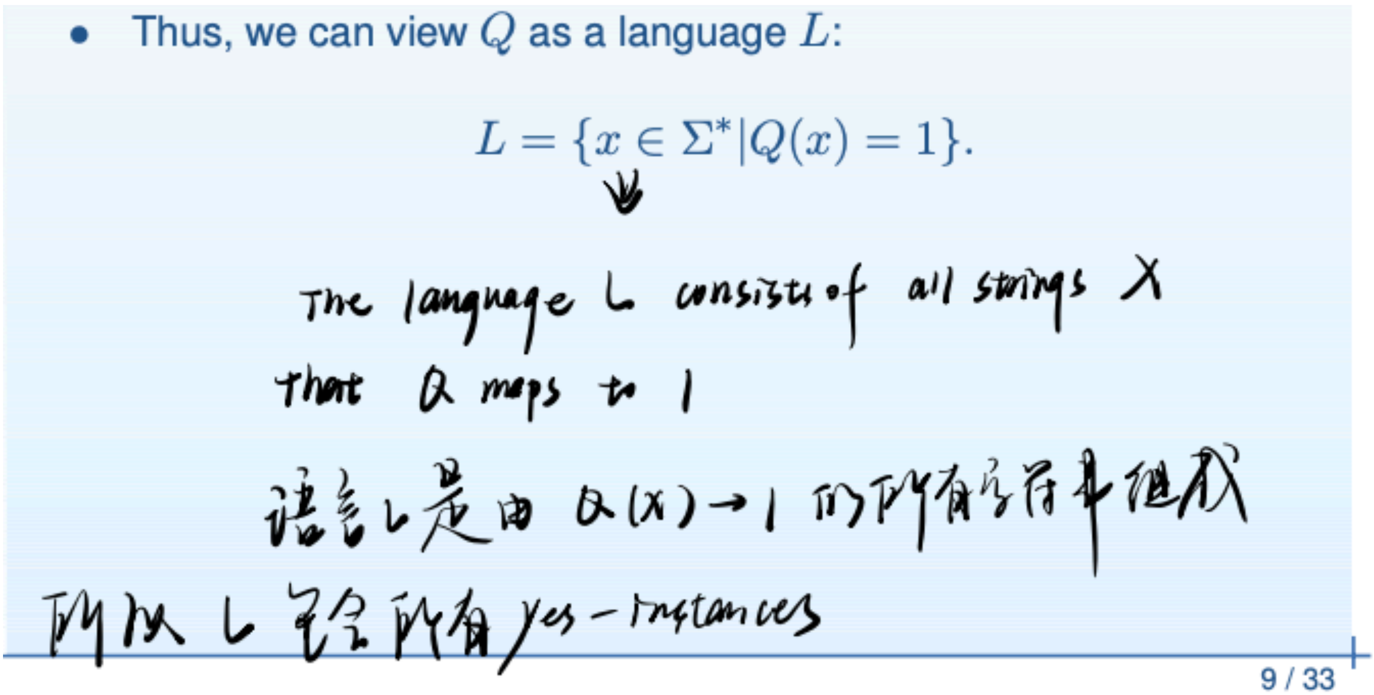


- ***Languages and decision problems***

Recall that we *encode instances of a decision problem as binary strings.*

Also recall that we may *view a decision problem as a mapping* $Q(x)$ *from instances* x *to* $\sum = \{0, 1\}$.

Q can be specified by the binary strings that encode yes-instances of the problem. => If you just specify which strings are yes-instances, then you have also specified Q.



My Understanding: Q can encode x and maps it into 1. L contains all the yes-instances.

- For instance, PATH is the language of binary strings $\langle G, u, v, k \rangle$ where G is a graph, u and v are vertices of G, and there is a u-to-v path in G with at most k edges.

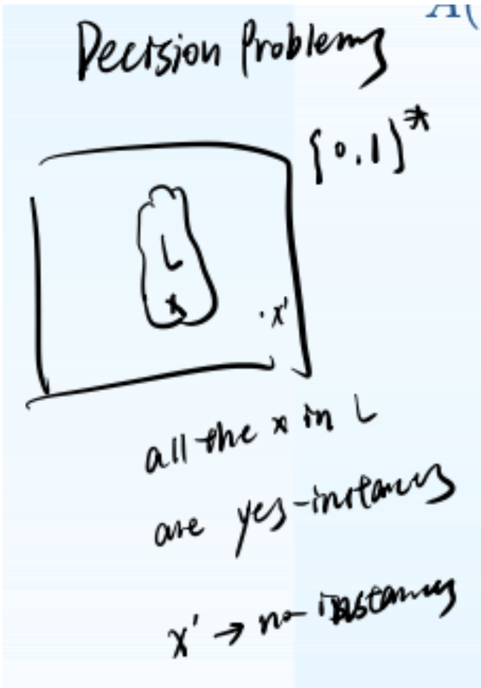
PATH is a language consists of yes-instances.

all these

9 / 33

- ***Language accepted/decided by an algorithm***

Let A be an algorithm for a decision problem and denote by $A(x) \in \{0, 1\}$ its output (if any) on input x.



A *accepts* a string x if $A(x) = 1$, A *rejects* a string x if $A(x) = 0$.

There may be strings that A neither accepts nor rejects. => A loop forever.

The language *accepted* by A is,

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

Suppose in addition that all strings not in L are rejected by A, i.e., $A(x) = 0$ for all $x \in \{0, 1\}^* \setminus L$.

Then we say that L is decided by A.

- Then we say that L is *decided* by A.

terminate on all strings

Accept → 1, 0, loop

Decide → {1, 0}

stronger★

Deciding a language is stronger than accepting it.

Accepting/deciding in polynomial time

允许 loop

- Language L is *accepted by an algorithm A in polynomial time* if A accepts L and runs in polynomial time on strings from L.
- L is *decided by A in polynomial time* if A decides L and runs in polynomial time on all strings.

require that it terminates in poly time.

Example: $PATH$ can both be accepted and decided in polynomial time.

Define the complexity class P:

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

All the languages

0, 1

if you have a language L and you can find an algorithm A that decides L in polynomial time → then L ∈ P

- ***P in terms of acceptance***

Lemma

$$
\begin{aligned}
P \overset{\text{def}}{=} \{L \subseteq \{0,1\}^* \mid & \text{there exists an algorithm that} \\
& \text{decides } L \text{ in polynomial time}\} \\
= \{L \subseteq \{0,1\}^* \mid & \text{there exists an algorithm that} \\
& \text{accepts } L \text{ in polynomial time}\}.
\end{aligned}
$$

- ⊆: straightforward. 上面被下面包含了。

1

Stronger: if A decides L also accepts L ⇒ ⊆

But for the other direction, we need to show that if L is accepted by a polynomial-time algorithm A, it is decided by a polynomial-time algorithm $A\prime$.

P in terms of acceptance

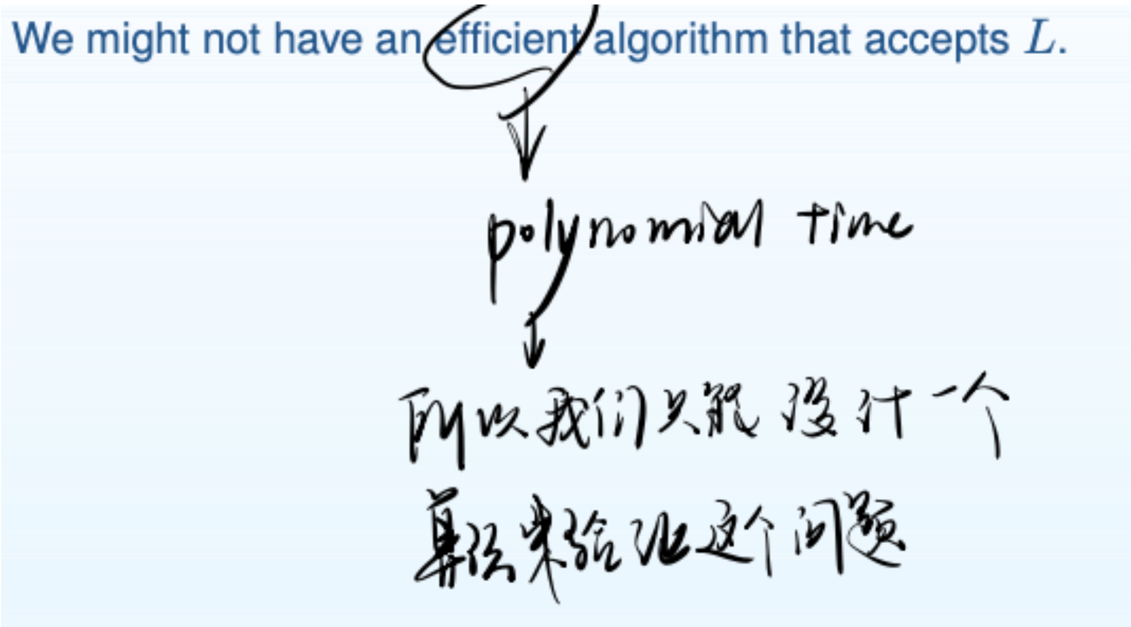
- Need to show: if L is accepted by a polynomial-time algorithm A, it is decided by a polynomial-time algorithm A'.
- Since A accepts L, it runs in at most cn^k steps before halting on any n-length string from L, where c and k are constants.
- Now let s be any string in Σ^*.
- A' simulates A with input s for at most $c|s|^k$ steps.
- If the simulation has not halted after this many steps, A' halts and outputs 0.
- Otherwise, A' outputs whatever A outputs.
- A' decides L and runs in polynomial time.

P是一个可以在多项式里被解决的问题的集合.

- ***Verfication***

Let L be a language.

We might not have an efficient algorithm that accepts L.



Consider an algorithm A taking two parameters, $x, c \in \sum^*$. => If you need to solve an assignment, it might be simpler to verify the solution of your fellow students and having to solve the assignment from scratch. You can think of c is a candidate solution that you are given with the problem instance.

Instead of trying to ***find*** a solution to x (which may take long time), A instead ***verifies*** that c is a solution to x.

- ***HAM-CYCLE problem***

An undirected graph G is hamiltonian if it contains a simple cycle containing every vertex of G. => A cycle visits every vertex exactly once in a graph. Cannot visit the same vertex more than once.

We define:

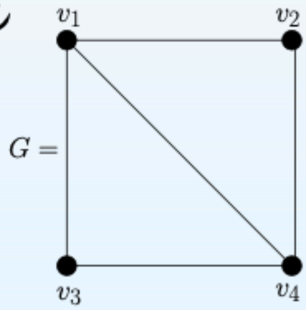
$$HAM - CYCLE = \{< G > | G \; is \; Hamiltonian\}.$$

It is very hard can be decided in polynomial time, however, it is easy to show that $HAM - CYCLE$ can be verified in polynomial time.

Verifying HAM-CYCLE

- Consider instead an algorithm A_{ham} taking two parameters, $\langle G \rangle$ and $\langle C \rangle$.
- A_{ham} checks that $\langle G \rangle$ defines an undirected graph G and that $\langle C \rangle$ encodes a cycle C containing every vertex of G exactly once.
- If so, A_{ham} outputs 1, otherwise 0.

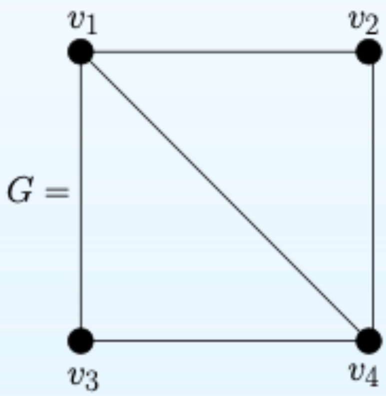
Does not need to find HAMILTonian cycle just to verify c represents a Hamtonian cycle.



$C = [v_1, v_2, v_3, v_4]$

- What is $A_{ham}(\langle G \rangle, \langle C \rangle)$?

$A_{ham}(< G >, < C >) = 0.$



$C = [v_1, v_2, v_4, v_3]$

$A_{ham}(< G >, < C >) = 1.$

Designing A_{ham} to run in polynomial time is easy. Hence, we can verify $HAM - CYCLE$ in polynomial time.

- ***Verifying a language***

A *verification algorithm* is an algorithm A taking two arguments, $x, y \in \{0, 1\}^*$, where y is the *certificate*. $x \rightarrow instances, y \rightarrow certificate.$

A *verifies* a string x if there is a certificate y such that $A(x, y) = 1$.

The language ***verified*** by A is,

$$L = \{x \in \{0,1\}^* | \textit{there is a } y \in \{0,1\}^* \textit{ such that } A(x, y) = 1\}.$$

- Example:

$$\text{HAM-CYCLE} = \{x \in \{0,1\}^* | \text{there is a } y \in \{0,1\}^* \text{ such that } A_{ham}(x, y) = 1\}.$$

如果x无环则输出永远是0.

17 / 33

- ***The complexity class NP***

NP is the class of languages that can be ***verified*** in polynomial time.

More precisely, $L \in NP$ if and only if there is a polynomial-time ***verification*** algorithm A and a constant c such that

$$L = \{x \in \{0,1\}^* | \text{there is a } y \in \{0,1\}^* \text{ with } |y| = O(|x|^c) \text{ such that } A(x, y) = 1\}.$$

have to be short.

- We have seen that HAM-CYCLE $\in$ NP.
- If $L \in$ P then $L \in$ NP. Why? → why p is contained in NP?

$\Rightarrow P \subseteq NP \Rightarrow$ 既然正解都出来了，给定c 只须比较下即可.

之所以定义NP是因为通常只有NP才能找到多项式解法。

18 / 33

如果有一个算法A以决定L(多项式内)
那么验证这个必然在多项式内。
直接从A去算，不用certificate也能
在多项式内。

- ***NP-complete problems***

There are problems in NP that are "***the most difficult***" in that class.

If any one of them can be ***solved*** in polynomial time then *every* ***problem in NP can be solved in polynomial time***.

These difficult problems are called *NP-complete*.

- HAM−CYCLE is NP-complete.
- Hence, if we could show HAM−CYCLE $\in$ P then P $=$ NP.

- ***Polynomial-time Reducibility***

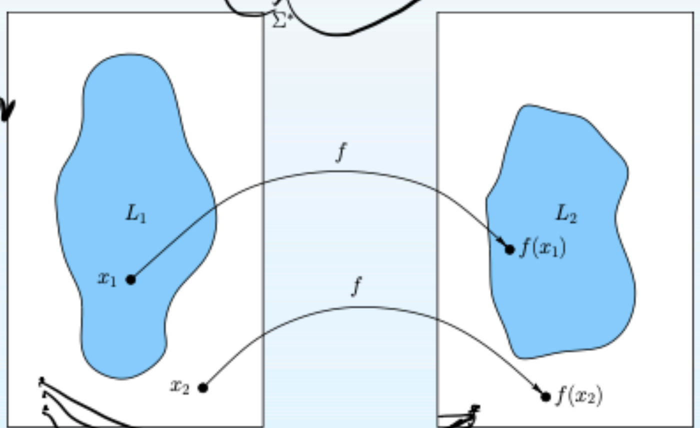
- Language L_1 is polynomial-time *reducible* to language L_2 if there is a polynomial-time computible function $f : \{0,1\}^* \to \{0,1\}^*$ such that for all $x \in \{0,1\}^*$,

$$x \in L_1 \Leftrightarrow f(x) \in L_2.$$

- In this case, we write $L_1 \leq_P L_2$.

difficult-level

L_1 is no harder than L_2



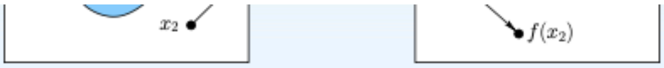
不一定要一对一

If $L_1 \leq_P L_2$ then L_1 is in a sense no harder to solve than L_2.

- More precisely,

$$L_1 \leq_P L_2 \wedge L_2 \in P \Rightarrow L_1 \in P.$$

if L_2 is easy, then L_1 is no more difficult than L_2, then L_1 must be easy as well.



如果你有一个 fast 算法 for L_2 使你可以迅速判定 x 是否属于 L_2.
那你如何 get a fast 算法 能确定一个 x 是否属于 L_1 呢?
⇓
reduce L_1 to L_2 and then solve L_2.

- More precisely,

$$L_1 \leq_P L_2 \wedge L_2 \in P \Rightarrow L_1 \in P.$$

- This follows since any instance x of L_1 can be solved by transforming it in polynomial time to an instance $y = f(x)$ of L_2 and then solving y with a polynomial-time algorithm for L_2.

→ 可以用B的解法解决A

- ***NP-complete languages***

- Language L is *NP-complete* if
  1. $L \in$ NP and
  2. $L' \leq_P L$ for every $L' \in$ NP.

至少跟 NP 问题中所有问题一样难

L is *NP-hard* if property 2 holds (and possibly not property 1).

- The class of NP-complete languages is denoted NPC.
- If some language of NPC belongs to P then P $=$ NP. Why?

$\Downarrow$

"$L_1 \leq_p L_2 \land L_2 \in P \Rightarrow L_1 \in P$."

$\Downarrow$

Suppose L_2 is NPC

$\therefore$ any other problems in NP is L_1

$\therefore$ $L_1 \in P$

So you take any problem in NP, you reduce that to your NPC, and then you run your NPC's poly algorithm, therefore, you are basically shown that arbitary problem and NP could be solved in polytime.

$\Rightarrow$ P = NP.

- ***Circuit satisfiability***

- A *boolean combinational circuit* consists of a collection of logic gates connected together with wires.
- The logic gates allowed are AND, OR, and NOT.
- Each wire has a value which is either 0 or 1.
- Some wires are specified by input values and the rest by the logic gates.
- Other wires specify output values.
- We can represent a circuit as an acyclic graph.

- Given a boolean combinational circuit C with one output wire.
- A *satisfying assignment* for C is an assignment of values to input wires of C causing an output of 1.
- The *circuit satisfiability problem* $\texttt{CIRCUIT-SAT}$ is to decide if a given circuit has a satisfying assignment:

$$\texttt{CIRCUIT-SAT} = \{\langle C \rangle | C \text{ is a satisfiable boolean combinational circuit}\}.$$

- We will show that $\texttt{CIRCUIT-SAT}$ is NP-complete.

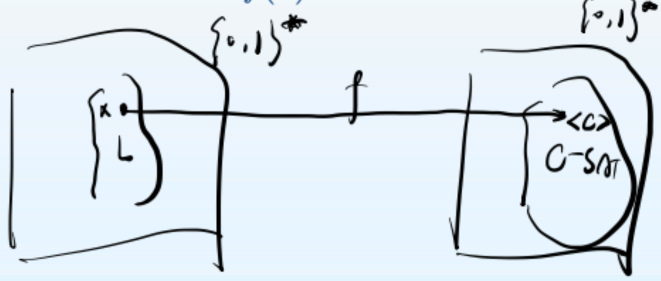
1. Showing $CIRCUIT - SAT \in NP$

   - We construct algorithm A with inputs x and y.
   - A checks that x represents a boolean combinational circuit C with one output wire and that y represents an assignment of truth values to the wires of C.
   - If so, A checks that y represents a valid truth assignment.
   - If so, A checks that the single output is 1.
   - If this is the case, A returns 1; otherwise it returns 0.
   - A is a verification algorithm for $\texttt{CIRCUIT-SAT}$ and can easily be made to run in polynomial time.
   - Thus, $\texttt{CIRCUIT-SAT} \in \text{NP}$.

2. Showing that $CIRCUIT - SAT$ is NP-hard

- Consider any language $L \in \mathrm{NP}$.
- We need to give a polynomial-time reduction from L to CIRCUIT-SAT.
- In other words, we need to find a polynomial-time algorithm A computing a function $f : \{0,1\}^* \to \{0,1\}^*$ such that

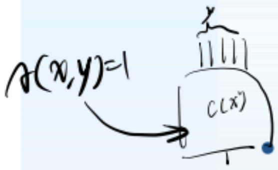
$$x \in L \Leftrightarrow f(x) \in \texttt{CIRCUIT-SAT}.$$



Showing that CIRCUIT-SAT is NP-hard

- Since $L \in \mathrm{NP}$, there is a polynomial-time algorithm A such that

$$L = \{x \in \{0,1\}^* | \text{there is a } y \in \{0,1\}^* \text{ with } |y| = O(|x|^c) \text{ such that } A(x,y) = 1\}.$$



- Given string x, f outputs a circuit $C(x)$ with $O(|x|^c)$ input wires.
- We ensure that $C(x)$ has a satisfying assignment of its input wires if and only if $A(x,y) = 1$ for some y with $|y| = O(|x|^c)$.
- This way,

问题转化 $$x \in L \Leftrightarrow f(x) = \langle C(x) \rangle \in \texttt{CIRCUIT-SAT}.$$

- Each y with $|y| = O(|x|^c)$ defines an input to $C(x)$.
- Intuition: Circuit $C(x)$ implements algorithm A on input (x,y) with x fixed.
- We ensure that $A(x,y) = 1$ if and only if y is a satisfying assignment.

- There is a constant k such that the worst-case running time $T(n)$ of A on an input (x,y) is $O(n^k)$ where $n = |x|$.
- The machine executing A has a certain *configuration* at each time step.
- The configuration gives a complete specification of the current memory, CPU state, and so on.
- When executing A on (x,y), the machine goes through a series of configurations $c_0, c_1, \ldots, c_{T(n)}$ (assume for simplicity that A runs for exactly $T(n)$ steps on (x,y)).
- Configuration c_0 specifies inputs x and y and the program code for A.
- One bit of the last configuration $c_{T(n)}$ specifies the $0/1$-output of A.

- Let M be the circuit implementing the hardware of the machine.
- We feed the initial configuration c_0 as input wires to M.
- M performs a single step of A and the new configuration c_1 is stored on output wires.
- These output wires feed into M which makes another step, forming c_2 as output, and so on.
- In total, we glue $T(n)$ copies of M together.
- This gives a BIG circuit representing the entire execution of A on input (x, y).
- The size of the circuit is still polynomial in n, however.

- We modify the circuit by hard-wiring part of the input to that specified by binary string x and so that the only output wire is that corresponding to the output of A.
- The circuit now only takes inputs y.
- The resulting circuit $C(x)$ has a satisfying assignment y if and only if $A(x, y) = 1$.
- $C(x)$ can be computed from x in time polynomial in $|x|$.
- This shows that $L \leq_P$ `CIRCUIT-SAT`.
- Thus, `CIRCUIT-SAT` is NP-hard.
- Since also `CIRCUIT-SAT` $\in$ NP, it follows that `CIRCUIT-SAT` is NP-complete.

$$\begin{aligned} \texttt{CIRCUIT-SAT} &\leq_P \texttt{SAT} \leq_P \texttt{3-CNF-SAT} \\ &\leq_P \texttt{SUBSET-SUM}, \\ \texttt{3-CNF-SAT} &\leq_P \texttt{CLIQUE} \leq_P \texttt{VERTEX-COVER} \\ &\leq_P \texttt{HAM-CYCLE} \leq_P \texttt{TSP} \end{aligned}$$

Overview for today

- NP-completeness and reductions
- NP-completeness of:
  - `SAT`
  - `3-CNF-SAT`
  - `CLIQUE`
  - `VERTEX-COVER`
  - `(HAM-CYCLE)` → 不学它
  - `TSP`
  - `SUBSET-SUM`

为什么有用 证明NPC ⇒ 有些问题适合用 Approximation来解决 而不是精确值，所以 如果表明 那个问题是NP，那么你不可能找到一个 多项式时间内运行的 算法，那么你就会去找 别的工具而不是找 精确的算法.

- ***Decision problems and languages***

- A *decision problem* Q consists of yes-instances and no-instances.
- Example, $Q = \texttt{HAM-CYCLE}$: $\langle G \rangle$ is a yes-instance if G contains a simple cycle containing all vertices of G; otherwise $\langle G \rangle$ is a no-instance.
- We can view a problem Q as a mapping of yes-instances to 1 and no-instances to 0.
- We can also view Q as a language L:

$$L = \{x \in \{0,1\}^* | Q(x) = 1\}.$$

- A *verification algorithm* is an algorithm A taking two arguments, $x, y \in \{0,1\}^*$, where y is the *certificate*.
- A *verifies* a string x if there is a certificate y such that $A(x, y) = 1$.
- The language verified by A is

$$L = \{x \in \{0,1\}^* | \text{there is a } y \in \{0,1\}^* \text{ such that } A(x,y) = 1\}.$$

The complexity class NP

- NP is the class of languages that can be verified in polynomial time.
- In other words, $L \in \text{NP}$ if and only if there is a polynomial-time verification algorithm A and a constant c such that

$$L = \{x \in \{0,1\}^* | \text{there is a } y \in \{0,1\}^* \text{ with } |y| = O(|x|^c) \text{ such that } A(x,y) = 1\}.$$

- We saw that $\text{P} \subseteq \text{NP}$.
- Big open problem: is $\text{P} = \text{NP}$?

Reducibility

- Language L_1 is polynomial-time *reducible* to language L_2 if there is a polynomial-time computible function $f : \{0,1\}^* \to \{0,1\}^*$ such that for all $x \in \{0,1\}^*$,

There should be an algorithm that implements this function and it should run in poly time.

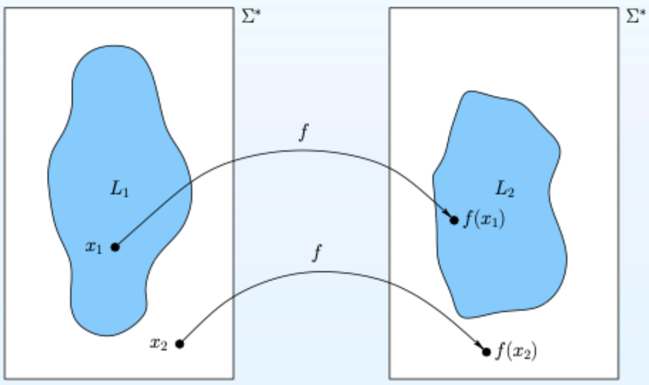
$$x \in L_1 \Leftrightarrow f(x) \in L_2.$$

reductions

要证明2个方向 ← iff

如果 x在 L1里, 那么 f(x) 就在 L2里, 如果 f(x) 不在 L2里, 则 x不在 L1里。

$$x \in L_1 \Leftrightarrow f(x) \in L_2.$$



- We use the notation $L_1 \leq_P L_2$ for this.
- We saw that

$$L_1 \leq_P L_2 \wedge L_2 \in P \Rightarrow L_1 \in P.$$

- Language L is *NP-complete* if
  1. $L \in$ NP and
  2. $L' \leq_P L$ for every $L' \in$ NP.
- L is *NP-hard* if L satisfies property 2 (and possibly not property 1).
- We saw that if any language of NPC belongs to P then P $=$ NP.
- We also showed that `CIRCUIT-SAT` is NP-complete.

- ***NP-completeness of other problems via reduction***

- Let L and L' be two languages with $L' \in$ NPC.
- If $L' \leq_P L$ then L is NP-hard.

NPc

L 至少比 L' 难

NPc

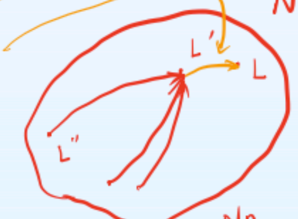
为啥

transitivity

$L'' \leq_P L$

$L' \leq_P L$

NP



L'' ∈ any language in NP

NP-completeness of other problems via reduction

之所以在证明 …是npc时，我们不得不从NP里挑一个一般情况，然后证明所有问题可以Reduce to 电路问题的，所有都小于…

- Let L and L' be two languages with $L' \in$ NPC.
- If $L' \leq_P L$ then L is NP-hard.
- If in addition $L \in$ NP then L is NP-complete.
- General technique to show NP-completeness of a language L:
  - Show that $L \in$ NP.

Just to reduce from one language if that language is NP hard.

只要约到比Npc一样难

Recipe

- General technique to show NP-completeness of a language L:
  - ① Show that $L \in$ NP.
  - ② Pick another language L' known to be NP-complete (for instance, CIRCUIT-SAT).
  - ③ Show that $L' \leq_P L$, i.e., show that there is a polynomial-time computible function $f : \{0,1\}^* \to \{0,1\}^*$ such that for all $x \in \{0,1\}^*$,

$$x \in L' \Leftrightarrow f(x) \in L.$$

- ***SAT problem***

The SAT problem

- A *boolean formula* ϕ consists of boolean variables $x_1, \ldots, x_n$, boolean connectives $\wedge$, $\vee$, $\neg$, $\rightarrow$, $\leftrightarrow$, and parentheses (and).
- Example: $\phi = (x_1 \vee x_2) \wedge (x_2 \vee x_3 \vee \neg x_4)$.
- A *satisfying assignment* for a boolean formula ϕ is an assignment of $0/1$-values to variables that makes ϕ evaluate to 1.
- ϕ is *satisfiable* if there exists a satisfying assignment for ϕ.
- We can now define the problem $\mathtt{SAT}$:

$$\mathtt{SAT} = \{\langle \phi \rangle | \phi \text{ is a satisfiable boolean formula}\}.$$

- We will show that $\mathtt{SAT}$ is NP-complete.

存在正解的ϕ集合

SAT

A boolean formula ϕ belongs to SAT exactly when it is satisfiable

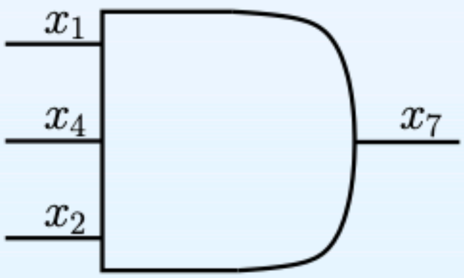
- To show $\mathtt{SAT} \in$ NPC, we follow our recipe:
  - Show that $\mathtt{SAT} \in$ NP.
  - Show that $\mathtt{CIRCUIT\text{-}SAT} \leq_P \mathtt{SAT}$.
- To show that $\mathtt{SAT} \in$ NP, we construct a verification algorithm A taking inputs x and y.
- It regards x as a boolean formula ϕ and y as an assignment of values to variables of ϕ.
- A returns 1 if y defines a satisfying assignment for ϕ; otherwise, A returns 0.
- We can easily make A run in polynomial time.
- Thus, $\mathtt{SAT} \in$ NP.

Showing $\texttt{CIRCUIT-SAT} \leq_P \texttt{SAT}$

- Given a circuit C, we transform it into a boolean function ϕ as follows.
- Associate a variable x_i with each wire of C; let x_m be the output wire variable.

  xi -> i^th wire
- We can view each gate of C as a function mapping the values on its input wires to the value on its output wire.

  subformula for each gate
- Construct a sub-formula for each such function.
- Example:



Sub-formula for gate: $x_7 \leftrightarrow (x_1 \wedge x_2 \wedge x_4)$

- If $\phi_1, \ldots, \phi_k$ are the sub-formulas, we define ϕ to be $x_m \wedge \phi_1 \wedge \phi_2 \wedge \ldots \wedge \phi_k$.
- Example:

$$
\begin{aligned}
\phi = x_{10} &\wedge (x_4 \leftrightarrow \neg x_3) \wedge (x_5 \leftrightarrow (x_1 \vee x_2)) \\
&\wedge (x_6 \leftrightarrow \neg x_4) \wedge (x_7 \leftrightarrow (x_1 \wedge x_2 \wedge x_4)) \\
&\wedge (x_8 \leftrightarrow (x_5 \vee x_6)) \wedge (x_9 \leftrightarrow (x_6 \vee x_7)) \\
&\wedge (x_{10} \leftrightarrow (x_7 \wedge x_8 \wedge x_9)).
\end{aligned}
$$

- ϕ can be constructed in polynomial time.
- In words, ϕ is stating that the output wire is 1 and that each gate behaves as it is supposed to.
- Thus, C is satisfiable if and only if ϕ is satisfiable:

$$\langle C \rangle \in \texttt{CIRCUIT-SAT} \Leftrightarrow \langle \phi \rangle \in \texttt{SAT}.$$

The formula we constructed is equivalent to the circuit we started with

- We have now shown that SAT is NP-complete.

- $3-CNF-SAT$

$SAT \leq_p 3-CNF-SAT.$

- $SUBSET-SUM$

$3-CNF-SAT \leq_P SUBSET-SUM.$

- $CLIQUE$

$3-CNF-SAT \leq_P CLIQUE.$ => 构造 vertex triple 图。正过来是选1，连起来发现是个 CLIQUE。反过来是找一个 CLIQUE 然后令其分别为 1，未知的点随便选，发现 $\phi = 1$。

- $VERTEX-COVER$

$CLIQUE \leq_P VERTEX-COVER.$ => 找补集。

- $HAM-CYCLE \leq_P TSP$

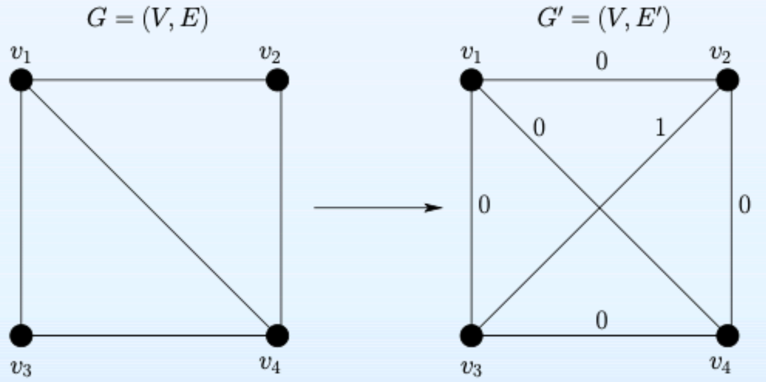
1. 构造 Ham cycle
2. 取两点相连
3. Ham cyle cost=0
4、新加的是 1
5. 得到一个TSP

反

1. 有一个TSP

- Let $G = (V, E)$ be an instance of the Hamilton-cycle problem.
- We construct a complete graph $G' = (V, E')$ on vertex set V.
- Define a cost function c on E' by

$$c(u, v) = \begin{cases} 0 & \text{if } (u, v) \in E, \\ 1 & \text{if } (u, v) \notin E. \end{cases}$$



- Show that: $\langle G \rangle \in \texttt{HAM-CYCLE} \Leftrightarrow \langle G', c, 0 \rangle \in \texttt{TSP}$.

完全图。

All the proof refers to the *Slides - NPC2*.