Design and Analysis of Algorithms

NP-Completeness

- Polynomial time
- Polynomial-time verification
- NP-completeness and reducibility
- NP-completeness proofs
- NP-complete problems

Zahed Rahmati
Dept of Math and CS
Tehran Polytechnic

Introduction

- You are given a problem. Suppose you give an algorithm to **solve the problem efficiently**.
  - What is the **running time of your algorithm**?
- The notion of what you mean by **efficient** is quite vague!
  - If n is small, a running time of 2^n may be just fine, but when n is huge, even n^2 may be unacceptably slow.
- **Two very general classes of combinatorial problems:**
  1. Those that can be solved by an intelligent search process and
  2. Those that involve simple brute-force search.
  - Most problems involve choosing from an exponential set of possibilities.
  - **The key distinguishing feature** in most cases was **whether there existed a polynomial time algorithm for solving the problem**.

Polynomial Time

- An ***algorithm*** is said to ***run in polynomial time*** if its ***worst-case running time is $\boldsymbol{O(n^c)}$***, where c is a nonnegative constant.
  - The running times like $O(n \log n)$ are also polynomial time, since $n \log n = O(n^2)$.
- **A problem is said to be solved *efficiently* if it is *solvable in polynomial time*.**
  - Higher worst-case running times, such as 2^n, $n!$, and n^n are **not polynomial time**.
- If you are ***interested*** only in ***small values of $\boldsymbol{n}$***,
  - An algorithm with running time of $O(2^n)$ with a small constant factor **may be vastly superior to**
  - An algorithm that runs in $O(n^{20})$ where the asymptotic notation hides big constant factors
- Note that there are ***many problems*** with ***good average-time-solutions***, but the ***worst case time may be very bad***, which may only arise in very rare instances.

Hard Problems

- By the end of the 60's, there was great success in finding **efficient solutions** to **many combinatorial problems**:
  - Minimum Spanning Trees, Shortest Paths, Chain Matrix Multiplication, LCS, Stable Marriage, Maximum Matching, Network Flows, Minimum Cut, …
- And, there was also a growing list of problems, called "**hard problems**", since ***no known efficient algorithmic solutions existed for these problems***.
  - Vertex Cover, Hamiltonian Cycle, Boolean Satisfiability, Set Cover, Clique Cover, Clique, Independent Set, Graph Coloring, Hitting Set, Feedback Vertex Set, …
- A remarkable discovery was made about the class of hard problems:
  - ***Many of these hard problems*** **turned out to be** ***equivalent***, which means:
  - If you could solve any one of them in polynomial time, then you could solve all of them in polynomial time.

Hard Problems

- Richard Karp and Stephen Cook developed the **mathematical theory**:
  - The notions of **P, NP, and NP-completeness** (which will be defined in next pages).
- Since then, thousands of problems were identified as being in this equivalence class.
- It is ***widely believed that none of them can be solved in polynomial time***, but ***there is no proof of this fact***.
- This has given rise to one of the ***biggest open problems in computer science***:
  - **Is P = NP?**
- Next, **we do NOT want to prove that a problem CAN be solved efficiently** by presenting an algorithm for it.
- We will be **trying to show that a problem CANNOT be solved efficiently.**

Encoding Inputs

- Here, ***we need to encode the input of our problems*** as ***a string over some alphabet*** that has a constant number (≥ 2) characters.
  - **Every data structure that we have seen can be serialized into a string**, ***without increasing its size significantly*. How?**
- **We should care the inputs to be encoded efficiently. Why?**
  - **If the input size grows exponentially**, then **an algorithm** (that ran in exponential time for the short input size) **may now run in linear time for the long input size**.
  - For example, encoding an integer in a inefficient manner (so that 8 is represented as 11111111), the length of the string can increase by exponentially.
- **To determine whether some new representation (of the encoding) is good:**
  - **It should be as concise as possible** (in the worst case).
  - **It should be possible to convert to the new form in polynomial time.**

Decision Problems

- So far we have discussed optimization problems.
  - Like finding the shortest path, finding the MST, and finding the maximum flow.
- **Here**, for considering the hardness, we need to **consider a decision version of the optimization problems**.

❖ A problem is called a **decision problem** if its output is a simple "yes" or "no".

- **MST decision problem:** Given a weighted graph G and an integer k, does G have a spanning tree whose weight is at most k?

- Note that *our job is to show that certain problems cannot be solved efficiently*.
  - ❖ **If we show that the *simple decision problem* cannot be solved efficiently, then certainly the more *general optimization problem* cannot be solved efficiently.**
  - If you can solve a decision problem efficiently, it is almost always possible to construct an efficient solution to the optimization problem, but this is a technicality…

Languages

- A decision problem can be thought of as a ***language recognition problem***.
- For example, we could define a language MST *encoding the MST problem* as:
  - **MST = {(G, k) | G has a spanning tree of weight at most k},**
  - when we say (G, k), we mean a reasonable, good encoding of the pair G and k as a string.
  - Let x = serialize(G, k).
- **What does it mean to solve the decision problem?**
  - **It means that *our proposed algorithm* would answer:**
    - **"yes"** if x ∈ MST (*i.e.*, if G has a spanning tree of weight at most k), and we say that the **algorithm accepts the input**.
    - **"no"** otherwise, and we say that the **algorithm rejects the input**.
- **For MST, given an input x, how would we determine whether $x \in$ MST?**
  - Run Kruskal's algorithm on x. If the final cost of the spanning tree is at most k, we accept x and otherwise we reject x.
- **Decision problems are equivalent to language membership problems.**

The Class P

❖ **P is the set of all languages for which membership can be determined in (worst case) polynomial time.**

- **P** corresponds to the set of ***all decisions problems that can be solved in (worst case) polynomial time***.
- Since Kruskal's algorithm runs in polynomial time, we have MST $\in$ P.

- Let **HC = {G | G has a simple cycle that visits every vertex of G}**.
- Is HC in P?
  - No one knows the answer for sure, but it is conjectured that it is not.
  - We will show that later that HC is NP-complete.

Certificates & Verification Process

- For many language recognition problems **there exists the property** that
  - **Would be *easy to verify* that a string *x* is in a language L.**
- For example, consider HC problem:
  - ***Suppose that a graph* did have *a Hamiltonian cycle*** and ***someone wanted to convince us of its existence***.
  - If this person could tell us the vertices along the cycle. ⟸ This cycle is called a **certificate**.
  - ➢ It is very easy for us to check if it is a legal cycle visiting all the vertices.

❖ Thus, even though we know of no efficient way to solve the HC problem, **there is a very efficient way to verify that a given graph has one**.

- **What if the graph did not have a Hamiltonian cycle?**
  - A *verification process is not required to do anything* ***if the input is not in the language.***

Certificates & Verification Process

❖ **Certificate:** A ***piece of information which allows us to verify*** that a given string is in a language ***in polynomial time***.

- Given a language L, given $x \in$ L, and given a string y (as the certificate).

❖ **A verification algorithm**: An algorithm which can **verify *x* is in the language L, using the certificate y** as help.

  – **If x is not in L then there is nothing to verify.**

- **If the verification algorithm runs in polynomial time**, we say that ***L can be verified in polynomial time***.
  – Examples: Are the following languages can be verified in polynomial time?
    – UHC = {G | G has a unique Hamiltonian cycle}.
    – CompHC = {G | G has no Hamiltonian cycle}.
  – What information would someone give us that allow us to verify G is in the language?

The Class NP

❖ **NP is the set of all languages that can be verified in polynomial time.**

- Formally, NP stands for *Nondeterministic Polynomial time*.

- **P $\subset$ NP. Why?**
- If we can solve a problem efficiently without a certificate, we can certainly solve given the additional help of a certificate.

- **However, it is not known whether P $=$ NP.**
- Most experts believe that P $\neq$ NP, but no one has a proof of this.

Reductions

- **Suppose that there are two problems, H and U.**
  - We know (or you strongly believe at least) that **H is hard**, that is, it cannot be solved in polynomial time.
  - The complexity of **U is unknown**.

- **We want to prove that U is also hard**. **How would we do this?**
  - Actually we want to show this: $(\boldsymbol{H} \notin \boldsymbol{P}) \Longrightarrow (\boldsymbol{U} \notin \boldsymbol{P})$.
- To do this, we could prove the contrapositive:
  - We suppose that there exists a polynomial time algorithm for U, and then we use this algorithm to solve H in polynomial time, thus yielding a contradiction.
  - *i.e.*, $(\boldsymbol{U} \in \boldsymbol{P}) \Longrightarrow (\boldsymbol{H} \in \boldsymbol{P})$.

Polynomial Time Reduction

- Suppose **there is a subroutine that can solve any instance of U in polynomial time**.

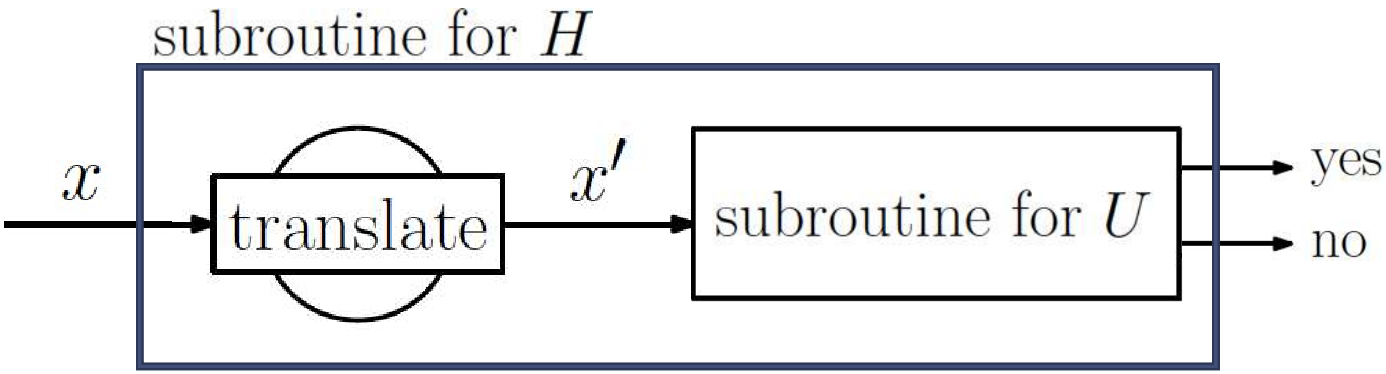
➢ Given an input x for H, we could translate it into an ***equivalent*** input x′ for U.

   – By "***equivalent***" we mean that $\boldsymbol{x \in H}$ **if and only if** $\boldsymbol{x' \in U}$.

   ➢ If U is solvable in polynomial time, then so is H.

   ❑ We assume that the ***translation module runs in polynomial time***.

   ❑ If so, we say we have a **polynomial reduction of H to U**, which is denoted $\boldsymbol{H \leq_P U}$.



3-Coloring $\leq_P$ Clique Cover

Consider the following as known hard problem H:

- **3-Coloring**: Given a graph G, can each of its vertices be labeled with one of three different colors, such that no two adjacent vertices have the same label.
  - It is NP-complete (will get to it).
  - Planar graphs can be colored with four colors, and there is a polynomial time algorithm.

Consider the following as unknown problem U:

- **Clique Cover**: Given a graph G and an integer k, can we partition the vertex set into k subsets of vertices $V_1, \ldots, V_k$ such that each V_i is a clique of G.
  - A subset of vertices $V' \subseteq V$ forms a clique if for every pair of distinct vertices $u, v \in V'$, $(u, v) \in E$. That is, the subgraph induced by V' is a complete graph.

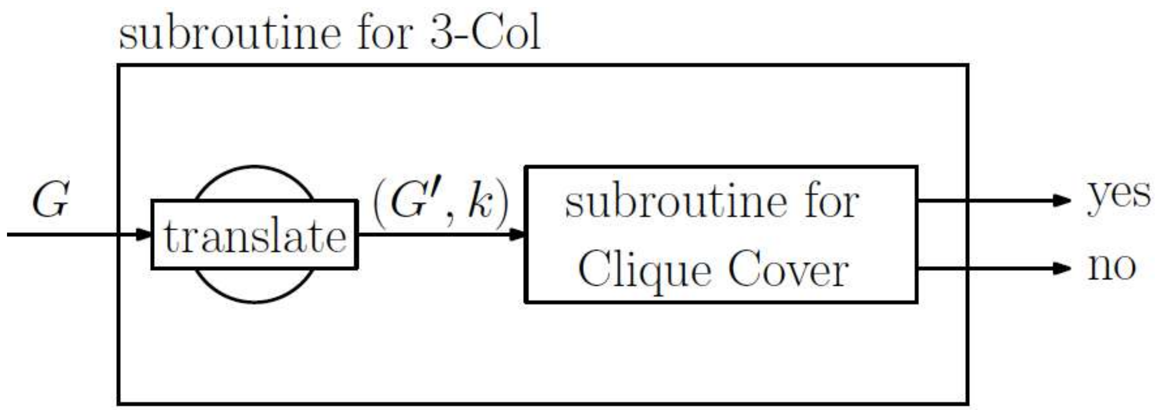
❖ **Assuming 3-Colering (3Col) is hard, we want to prove that Clique Cover (CCov) is hard.**

3-Coloring $\leq_P$ Clique Cover

- Need to prove this: $(3\text{Col} \notin \text{P}) \implies (\text{CCov} \notin \text{P})$

➢ We'll prove the contrapositive: $(\text{CCov} \in \text{P}) \implies (3\text{Col} \in \text{P})$

- To prove this, we **will find a translation**, that **maps an instance G** for 3-Col **into an instance (G', k)** for Ccov. **How to compute G'? k?**

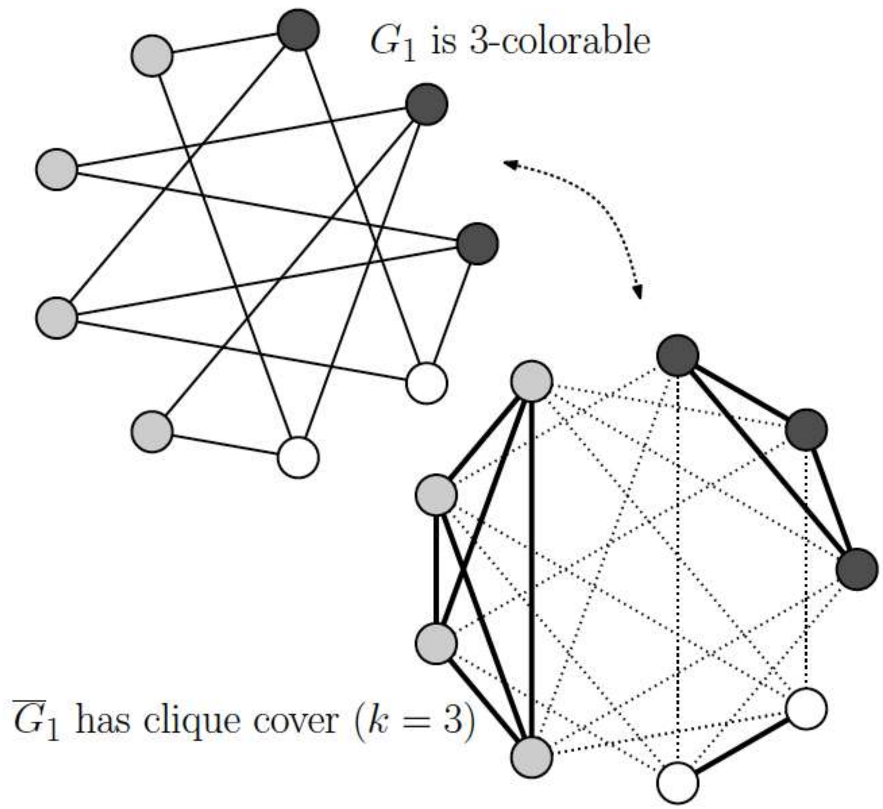


3-Coloring $\leq_P$ Clique Cover

- **Both** problems involve partitioning the vertices up into groups.
- In **3Col**, two vertices are in the same group if not having an edge between them. In **CCov**, if two vertices are in the same group, must have an edge between them.
- **The translator sets $(G', k) = (\bar{G}, 3)$.**

❑ **Claim:** $G \in \text{3Col} \iff (\overline{G}, 3) \in \text{CCov}$

- **Proof** $\Longrightarrow$: let V_1, V_2, V_3 be the three color classes:
  - This is a clique cover of size 3 for $\bar{G}$.
- **Proof** $\Longleftarrow$: If $\bar{G}$ has a clique cover of size 3, denoted by V_1, V_2, V_3:
  - Give the vertices of V_i color i. This is a legal coloring for G.21



Polynomial Time Reduction: Lemmas

- We say that a language (*i.e.* decision problem) $\boldsymbol{L_1}$ **is polynomial-time reducible to** $\boldsymbol{L_2}$ (written $\boldsymbol{L_1 \leq_P L_2}$) ***if there is a polynomial time computable function f***, *such that for all x*, $\boldsymbol{x \in L_1}$ **iff** $\boldsymbol{f(x) \in L_2}$.
  - In our example, 3Col $\leq_P$ CCov, because $f(G) = (\bar{G}, 3)$ can be computed in time $O(n^2)$.

- Intuitively, saying that $\boldsymbol{L_1 \leq_P L_2}$ means that ***if*** $\boldsymbol{L_2}$ ***is solvable in polynomial time, then so is*** $\boldsymbol{L_1}$.

- **Lemma**: If $L_1 \leq_P L_2$ and $L_2 \in P$ then $L_1 \in P$.

- **Lemma**: If $L_1 \leq_P L_2$ and $L_1 \notin P$ then $L_2 \notin P$.

- **Lemma**: If $L_1 \leq_P L_2$ and $L_2 \leq_P L_3$ then $L_1 \leq_P L_3$.

NP-Completeness

- The **set of NP-complete problems** are ***all problems in the complexity class NP***, for which it is known that:
  - **if any one is solvable in polynomial time, then they all are**, and conversely,
  - **if any one is not solvable in polynomial time, then none are**.

❖ A language **L is NP-hard** if $L' \leq_P L$, for all $L' \in NP$.

– Note that L does not need to be in NP.

❖ A language **L is NP-complete** if:

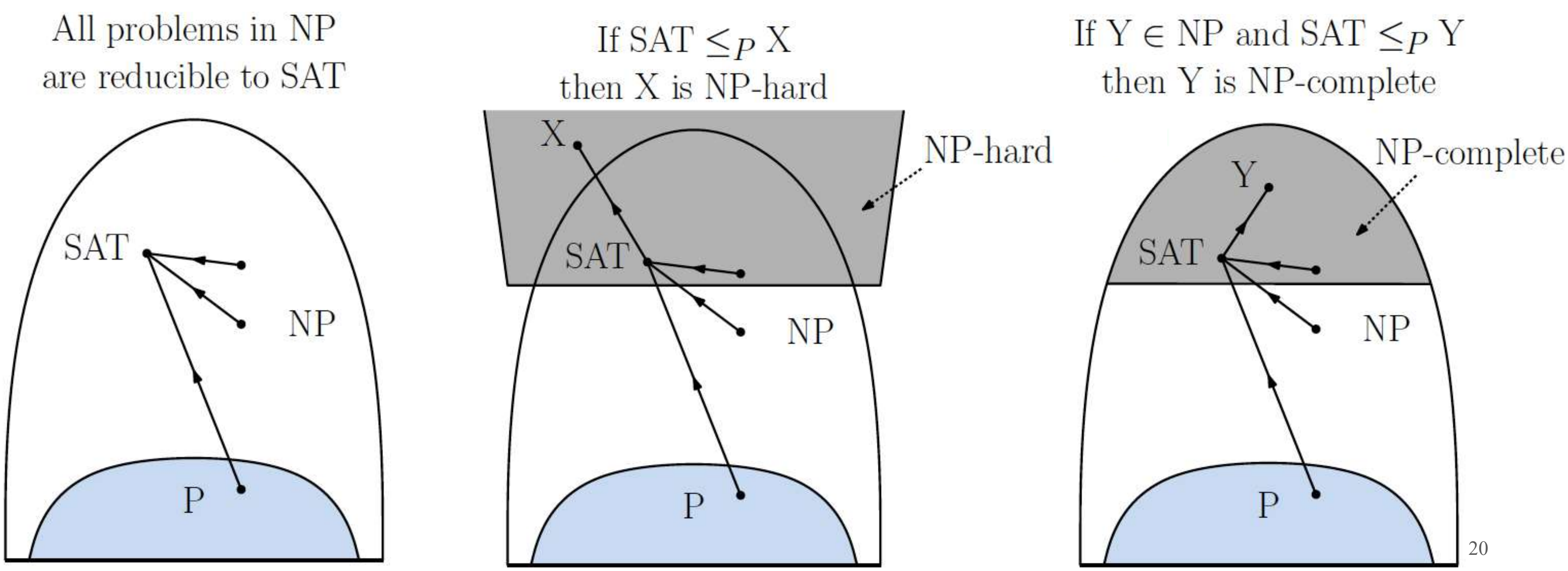
1. **L ∈ NP** (that is, it can be verified in polymomial time), and
2. **L is NP-hard** (that is, every problem in NP is polynomially reducible to it).
   - Unfortunately, it almost impossible to prove that one problem is NP-complete, because we have to be able to reduce every problem in NP to this problem.

➢ We can replace (2) by this: $\boldsymbol{L' \leq_P L}$**, for some known NP-complete language** $\boldsymbol{L'}$. Why?

- The reason is that all the languages in NP are reducible to $\boldsymbol{L'}$.

NP-Completeness

- Cook showed that ***there is one problem*** called SAT (boolean satisfiability) **that is NP-complete**.
- To **prove that *our problem*** is NP-complete, all we need to do is to show that (1) our problem is in NP and (2) reduce SAT to our problem.



Cook's Theorem

- **Boolean formula**: A formula that consists of ***variables*** (say x, y, z) and ***logical operations*** NOT ($\bar{x}$), AND ($x \wedge y$), and OR ($x \vee z$).

- Given a boolean formula, it is ***satisfiable*** if there is a way to assign values 0 or 1 to the variables such that it evaluates to 1.
  - Consider this formula: $F_1(x, y, z) = (x \wedge (y \vee \overline{z})) \wedge ((\overline{y} \wedge \overline{z}) \vee \overline{x})$
    - F_1 is satisfiable, by the assignment x = 1 and y = z = 0.
  - Consider this formula: $F_2(x, y) = (\overline{z} \vee x) \wedge (z \vee y) \wedge (\overline{x} \wedge (\overline{y})$
    - F_2 is not satisfiable.

- **Boolean Satisfiability Problem (SAT)**: Given a formula F, is it possible to assign values 0/1 to the variables, so that F evaluates to 1?

literal

$$\boldsymbol{F} = (\mathbf{a} \vee \boldsymbol{b} \vee \boldsymbol{c}) \wedge \overbrace{(\boldsymbol{d} \vee \boldsymbol{e} \vee \boldsymbol{f})}^{\text{clause}} \wedge \ldots$$

❖ **Cook's Theorem: 3SAT is NP-complete.**
  - 3SAT is in NP and is NP-Hard. Proofs?
    - ➢ Given a certificate consists of the assignment of values to the variables, we plug the values into the formula and evaluate it.

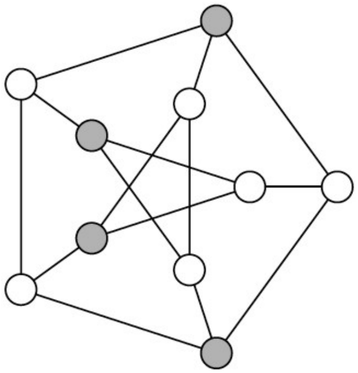
3SAT: NP-Hardness proof

To show that the 3SAT is NP-hard, Cook reasoned as follows:

- **Every NP-problem can be encoded as a program** that *runs in polynomial time on a given input*, subject to a number of nondeterministic guesses.
- Since the program runs in polynomial time, **we can express its execution** on a specific input **as straight-line program** that **contains a polynomial number of lines of code** in your favorite programming language.
- **Compile each line of code** into **machine code**.
- **Convert each machine code instruction** into **an equivalent boolean circuit**.
- ➢ Finally, we can **express each of these circuits** equivalently **as a boolean formula**.
- Note that, **nondeterministic choices** (of the certificate) **can be implemented as boolean variables in this formula**, whose values take on the possible values of 0 and 1.
- If you could determine the satisfiability of this formula in polynomial time, you could determine whether the original nondeterministic program output "yes" in polynomial time.

Independent Set (IS)

❖ **Given a graph G = (V, E) and an integer k, does G contain a subset V' of k vertices such that no two vertices in V' are adjacent to one another.**

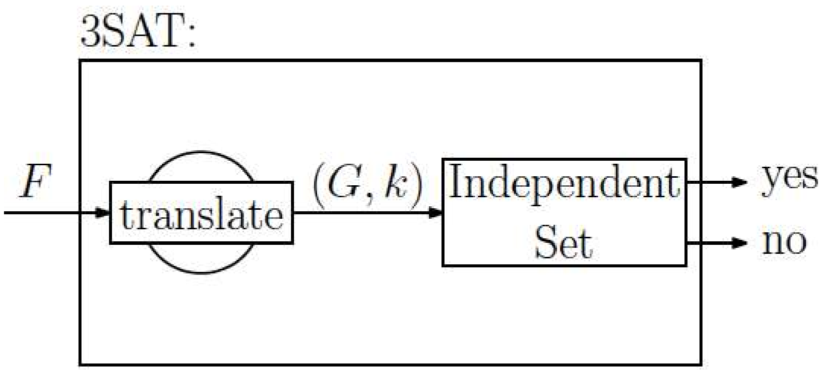


❖ **Claim**: **IS is NP-complete**.

1. **IS ∈ NP.** Why?
2. **IS is NP-hard.**

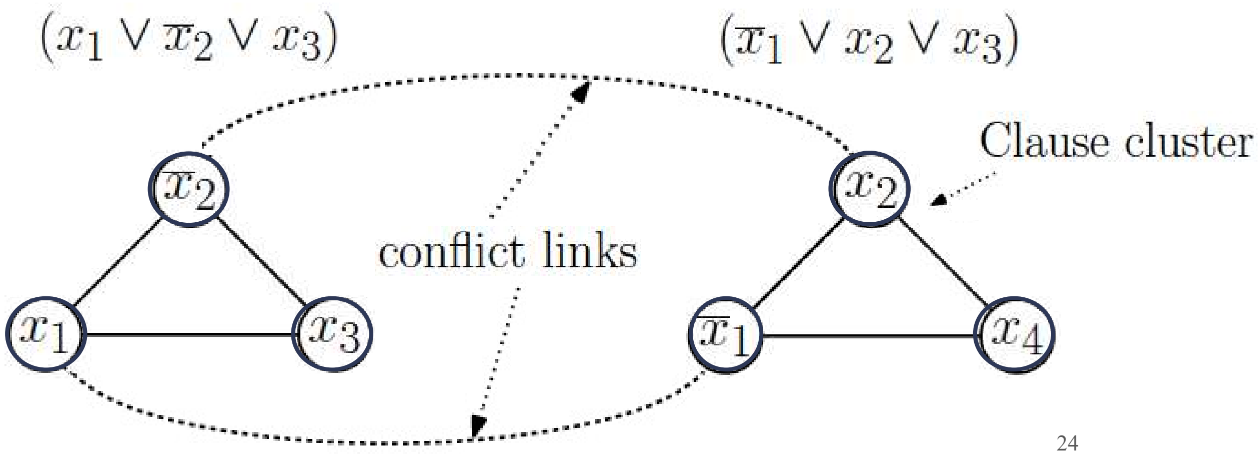
Proof (2): We show that 3SAT is polynomially reducible to IS, *i.e.*, 3SAT $\leq_P$ IS.

- We need **find a polynomial time computable function f** that $f(F) = (G, k)$, and **prove** that the formula **F is satisfiable iff G has an independent set of size k**.



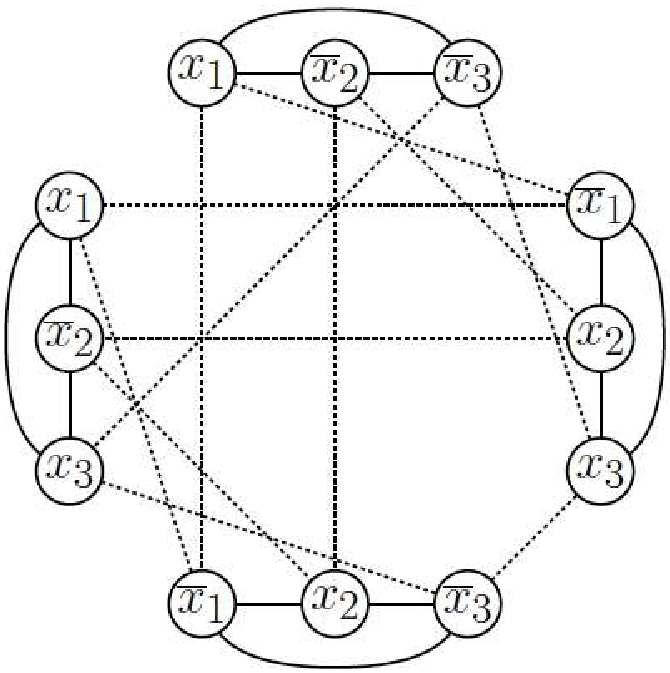
3SAT $\leq_P$ IS: Finding translator

- **For each literal in each clause, we create a vertex in G.**
  - ➢ Each clause must contain at least one literal whose value it true.
    - – At least one vertex of each group must be in any independent set.
- V' must contain at least k vertices.
  - ➢ Group vertices of each clause, and **create edges between all pairs of vertices in clause**.
    - – Thus exactly one vertex of each group must be in any independent set.
    - – Thus k = #clauses.
- If x_i is assigned true, then $\overline{x_i}$ must be false, and vice versa.
  - ➢ Thus we need to **add conflict links in the graph**.



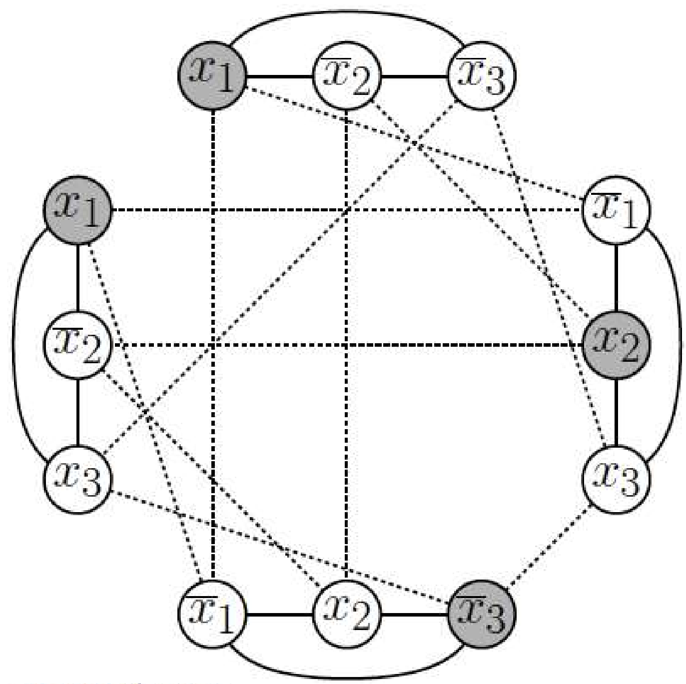
3SAT $\leq_P$ IS: Example

$$F = (x_1 \vee \overline{x}_2 \vee \overline{x}_3) \wedge (\overline{x}_1 \vee x_2 \vee x_3) \wedge (\overline{x}_1 \vee x_2 \vee \overline{x}_3) \wedge (x_1 \vee \overline{x}_2 \vee x_3)$$



The reduction

$k = 4$



correctness

$x_1 = x_2 = 1,\ x_3 = 0$

3SAT $\leq_P$ IS: Proof

- **F is satisfiable iff G has an independent set of size k.**

Proof $\Longrightarrow$ If F is satisfiable,

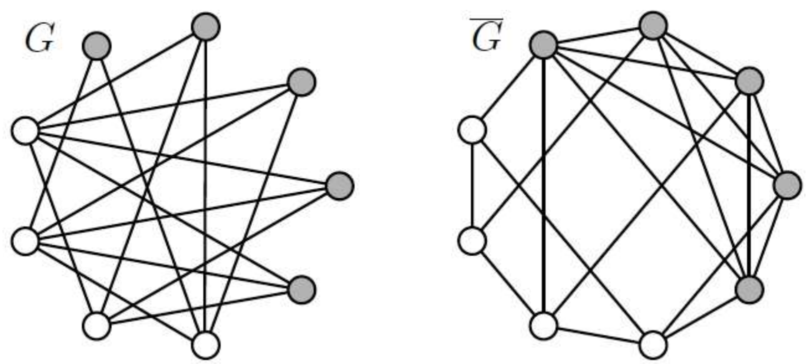
- Each of the k clauses of F must have at least one true literal. Let V' denote the corresponding vertices.
- Since there are k clauses, and **we cannot take two conflicting literals to be in V'** (both of its endpoints cannot be in V'), **V' is an independent set of size k**.

Proof $\Longleftarrow$ If G has an independent set of size k,

- We cannot select two vertices from a clause cluster.
- Since there are k clusters, **V' has exactly one vertex from each clause cluster**.
- Note that if a vertex x is in V', then the adjacent vertex $\bar{x}$ cannot also be in V'.
- ➢ There is an assignment in which the corresponding literal of each vertex in V' is set to 1.
- Such an assignment satisfies one literal in each clause, and therefore **F is satisfiable**.

Clique problem

❖ **Clique problem: Given an undirected graph $G = (V, E)$ and an integer k, does G have a subset V' of k vertices such that for each distinct vertices u,v $\in V'$, $(u, v) \in E$.**

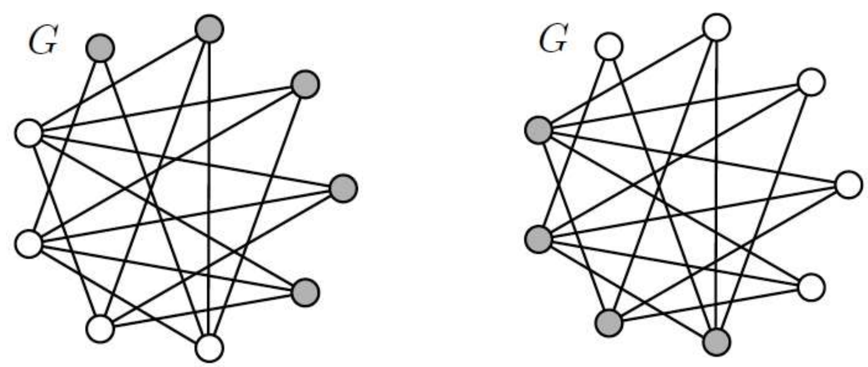


- **Clique (CLQ) is NP-Complete.**
  - **CLQ $\in$ NP.** Why?
  - **CLQ is NP-Hard.**
- **IS $\leq_P$ CLQ:**
  - **Given an instance (G, k) of the IS problem, we can produce an equivalent instance (G', k') of the Clique problem in polynomial time. How to translate?**
    - $f(G, k) = (\overline{G}, k)$.
  - **V' is an independent set of size k for G IFF V' is a clique of size k for $\overline{G}$.**
    - Proof?

Vertex Cover problem

- A **vertex cover** in an undirected graph $G = (V, E)$ is **a subset of vertices V' $\subseteq V$ such that *every edge in G has at least one endpoint in V'***.

❖ **Vertex cover problem (VC): Given an undirected graph $G = (V, E)$ and an integer k, does G have a vertex cover of size k?**



- **VC is NP-Complete.**
  - **VC ∈ NP.** Why?
  - **VC is NP-Hard.**
- **IS $\leq_P$ VC:**
  - $f(G, k) = (G, n - k)$.
  - **V' is an independent set of size k for G IFF $V \backslash V'$ is a vertex cover of size $n - k$ for G.**
    - Proof?

Dominating Set problem

- Dominating set is an example of a ***graph covering problem***.
- **Each vertex is adjacent to at least one member of the dominating set.**
  - Note that **each edge** is ***incident*** to at least one member of the ***vertex cover***.

❖ **Dominating Set (DS) problem: Given an undirected graph G and an integer k, does G have a dominating set of size k?**
  - If G is connected and has a vertex cover of size k, then it has a dominating set.

- **DS is NP-Complete.**
  - **DS ∈ NP.** Why?
  - **DS is NP-Hard.**
    - Proof?

VC $\leq_P$ DS: Finding translator

- Find a polynomial time translator $f(G, k) = (G', k')$, such that **G has a vertex cover of size k IFF G' has a dominating set of size k'**.
- How to translate between these problems?
  - In VC, **every edge is incident** to a vertex in V'.
  - In DS, **every vertex is either in V' or is adjacent** to a vertex in V'.
- The ***reduction function*** maps ***edges of G into vertices in G'***, such that **an incident edge in G is mapped to an adjacent vertex in G'**.
  - **Insert a vertex w_{uv} into the middle of each edge (u, v) of the graph.**
  - The fact that **u was incident to edge (u, v)** has now been replaced with the fact that **u is adjacent to the corresponding vertex w_{uv}**.
- Note that we still need to dominate the neighbor v.
  - To do this, we **will leave the edge (u, v) in the graph as well**.

VC $\leq_P$ DS: Finding translator

- In summary, **we create the graph G' as follows**:
  - **Initially $G' = G$.**
  - **For each edge (u, v) in G we create a new vertex w_{uv} in G′**, and
  - **Add two edges (u, w_{uv}) and (w_{uv}, v) in G'**.

- Let V_I denote the ***isolated vertices*** in G, and let $n_I = |V_I|$.
- The number of vertices to request for the dominating set will be $k' = k + n_I$.

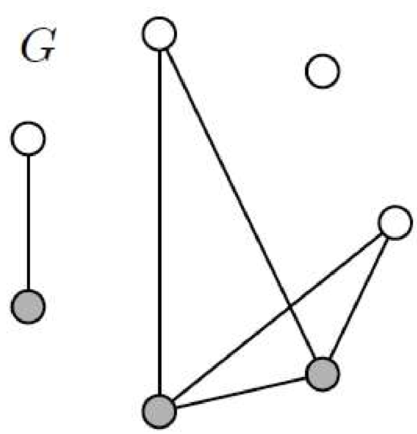
$$\boxed{f(G, k) = (G', k')}$$

❑ **G has a vertex cover of size k IFF G' has a dominating set of size k'.**

VC $\leq_P$ DS: Correctness

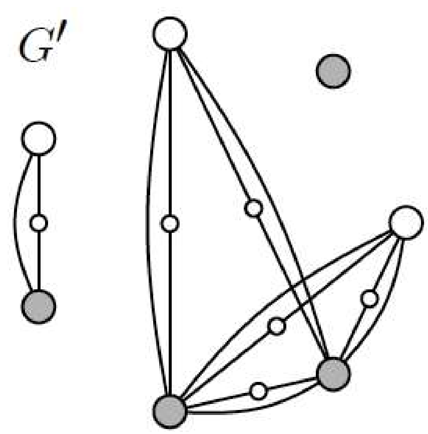
Proof $\Longrightarrow$ If V' is a vertex cover for G, then $V'' = V' \cup V_I$ is a dominating set for G'.

- **All the isolated vertices are in V''** and so **they are dominated**.
- **For each special vertex w_{uv} in G′, either u or v is in the vertex cover V'. Thus w_{uv} is dominated by the same vertex in V''.**
- **For each of the nonisolated vertex v, either it is in V' or all of its neighbors are in V'. Thus, in either case, v is either in V'' or adjacent to a vertex in V''.**



G has a vertex cover of size 3

$\Rightarrow$



G' has a dominating set of size 4

VC $\leq_P$ DS: Correctness

Proof $\Leftarrow$ If G' has a dominating set V'' of size $k' = k + n_I$, then G has a vertex cover V' of size k.

- **Ignore isolated vertices; let $V''' = V'' \setminus V_I$ be the remaining k vertices.**
- **If there is some special vertex w_{uv} in V''', then modify V''' as follows:**
  - **Replace w_{uv} with u.**
  - **Let V' denote the resulting set after this modification.**
  - **Claim: V' is a vertex cover for G. Why?**