Lec20: NP-Completeness

Algorithm I
COMP319-003
Spring 2023

Instructor: Jiyeon Lee
School of Computer Science and Engineering
Kyungpook National University **(KNU)**

Last time

- We talked about s-t cuts and s-t flows.
- The **Min-Cut Max-Flow Theorem** says that minimizing the cost of cuts is the same as maximizing the value of flows.
- The **Ford-Fulkerson algorithm** does this!
  - Find an augmenting path
  - Increase the flow along that path
  - Repeat until you can't find any more paths and then you're done!

NP-Completeness

- So far we've seen a lot of good news.
  - Such a problem can be solved quickly.
    - i.e., in close to linear time, or at least a time that is some small polynomial function of the input size.

- **NP-completeness** is a form of bad news.
  - Evidence that many important problems can not be solved quickly.
  - NP-complete problems really come up all the time.

Why should we care?

- Knowing that they are hard lets you stop beating your head against a wall trying to solve them.
  - **Use a heuristic:** come up with a method for solving a reasonable fraction of the common cases.
  - **Solve approximately:** come up with a solution that you can prove that is close to right.
  - **Use an exponential time solution:** if you really have to solve the problem exactly and stop worrying about finding a better solution.

Algorithmic vs Problem Complexity

- The **algorithmic complexity** of a computation is some measure of how difficult is to perform the computation (i.e., specific to an algorithm).
- The **complexity of a computational problem** or task is the complexity of the algorithm with the lowest order of growth of complexity for solving that problem or performing that task.
  - e.g., the problem of searching an ordered list has at most *logn* time complexity.

- **Computational Complexity:** deals with classifying problems by how hard they are.

Decision & Optimization Problems

- **Decision problems**
  - Given an input and a question regarding a problem, determine if the answer is **yes or no.**
- **Optimization problems**
  - Find a solution with the **"best" value.**
  - Optimization problems can be cast as decision problems.
    - e.g., Find a path between u and v in graph G that uses the fewest edges --> Does a path exist from u to v consisting of at most k edges?

- Decision problems are easier than optimization problems.
  - If we prove a decision problem is NP-complete (i.e., does not have an efficient solution), then optimization problem will be at least as hard.
  - In NP-complete problem, we deal with the decision version.

Class of "P" Problems

- **Class P** consists of (decision) problems that are solvable in polynomial time.
- Polynomial-time algorithms
  - Worst-case running time is $O(n^k)$ for some constant k
  - Examples of polynomial time:
    - $O(n^2)$, $O(n^3)$, $O(1)$, $O(n\lg n)$
  - Examples of non-polynomial time:
    - $O(2^n)$, $O(n^n)$, $O(n!)$

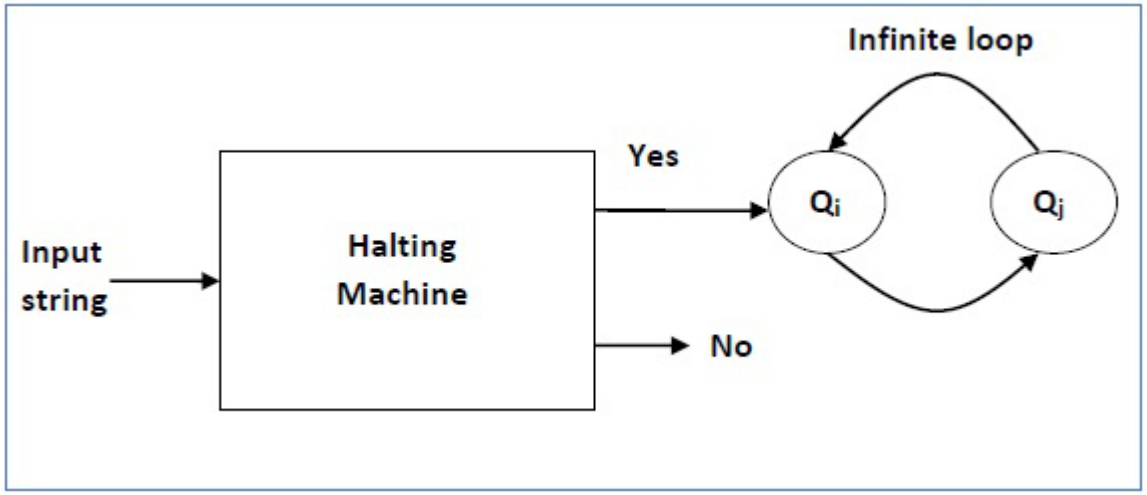
Tractable/Intractable Problems

- Problems in P are also called **tractable.**
- Problems not in P are **intractable or unsolvable.**
  - Can be solved in reasonable time only for small inputs.
  - Or, can not be solved at all.

  - Note that non-polynomial algorithms are not always worse than polynomial algorithms.
    - $n^{1,000,000}$ is technically tractable, but really impossible.
    - $n^{loglogloglogn}$ is technically intractable, but easy.

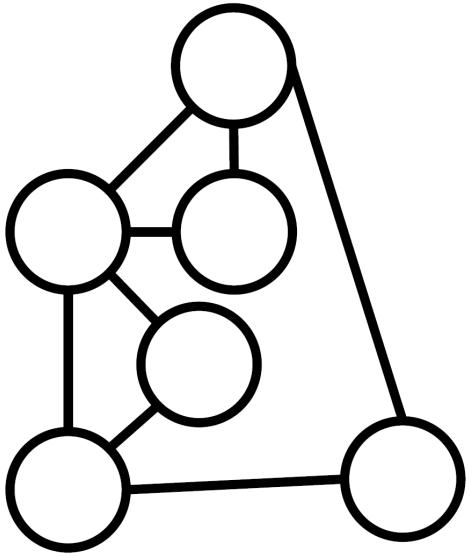
Example of Unsolvable Problem

- Turing discovered in the 1930's that there are problems unsolvable by any algorithm.
- The most famous of them is the **halting problem.**
  - Given an arbitrary algorithm and its input, will that algorithm eventually halt, or will it continue forever in an "infinite loop?"



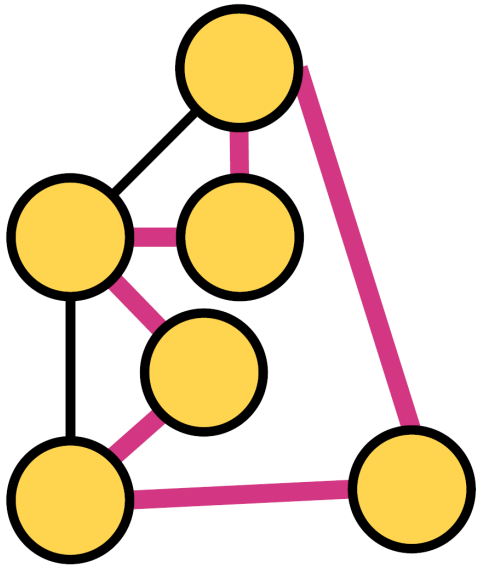
Examples of Intractable Problems

- **Hamiltonian Path:** Given a graph, find a path that passes through every vertex exactly once.
  - *Decision version: Does a given graph have a Hamiltonian Path?*



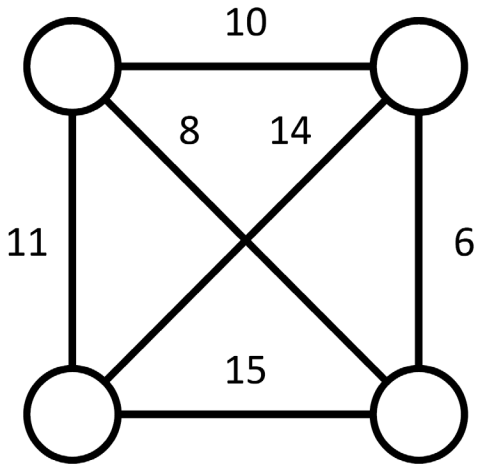
Examples of Intractable Problems

- **Hamiltonian Path:** Given a graph, find a path that passes through every vertex exactly once.
  - *Decision version: Does a given graph have a Hamiltonian Path?*
- This problem is known to be NP-complete.



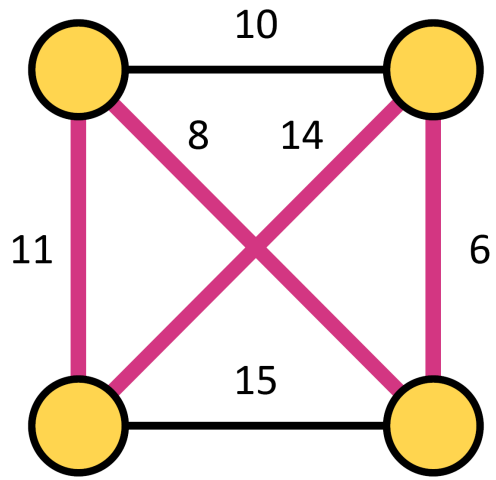
Examples of Intractable Problems

- **Traveling Salesman:** Given a complete, undirected weighted graph G, find a minimum weight Hamiltonian Path.
  - *Decision version: Given a complete, undirected weighted graph G, and an integer k, is there a Hamiltonian Path with a total weight at most k?*
  - Note that since G is complete, there must be at least one Hamiltonian cycle.



Examples of Intractable Problems

- **Traveling Salesman:** Given a complete, undirected weighted graph G, find a minimum weight Hamiltonian Path.
  - *Decision version: Given a complete, undirected weighted graph G, and an integer k, is there a Hamiltonian Path with a total weight at most k?*
  - Note that since G is complete, there must be at least one Hamiltonian cycle.
- This problem is known to be NP-hard.



Intractable Problems

- Can be classified in various categories based on their degree of difficulty, e.g.,
  - NP
  - NP-complete
  - NP-hard

- Let's define NP algorithms and NP problems.

Nondeterministic and NP Algorithms

- **Solving problem** = two stage procedure:
  1) "Guessing" stage:
     - Generate an arbitrary string that can be thought of as a candidate solution (a.k.a, certificate).
  2) "Verification" stage:
     - Take the certificate and the instance to the problem and returns YES if the certificate represents a solution.

- **NP algorithms (Nondeterministic Polynomial)**
  - Guessing stage is non-polynomial.
  - Verification stage is polynomial.

Class of “NP” Problems

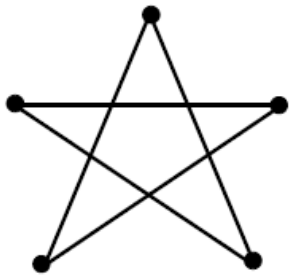
- **Class NP** consists of problems that could be solved by NP algorithms.
  - i.e., verifiable in polynomial time.
- If we were given a “certificate” of a solution, we could verify that the certificate is correct in time polynomial to the size of the input.

- Warning: NP does **not** mean “non-polynomial”

P and NP

- **P** = problems that can be solved in polynomial time.
- **NP** = problems for which a solution can be verified in polynomial time.

- e.g.,
  - Is sorting in NP? -> Not a decision problem.
  - Is sortedness in NP? -> Yes. Easy to verify.
  - Is this sequence $\langle v_1, v_2, v_3, \ldots, v_{|V|} \rangle$ a Hamiltonian cycle?
    -> Also easy to verify.

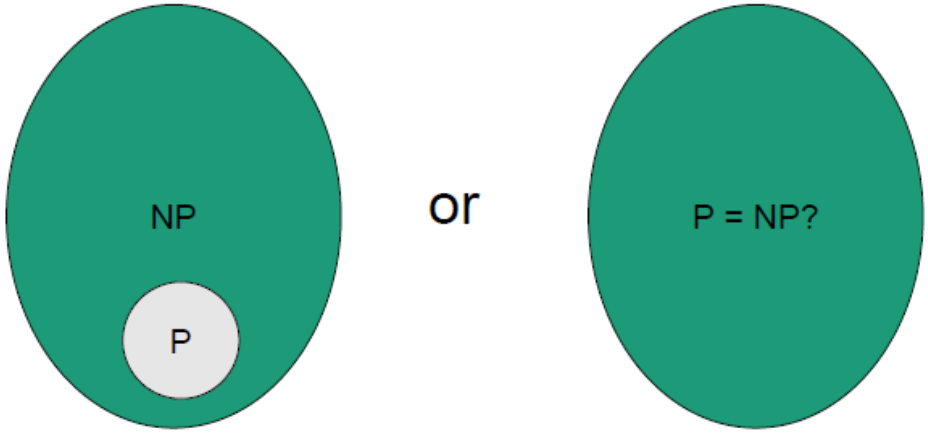


hamiltonian

Is P = NP?

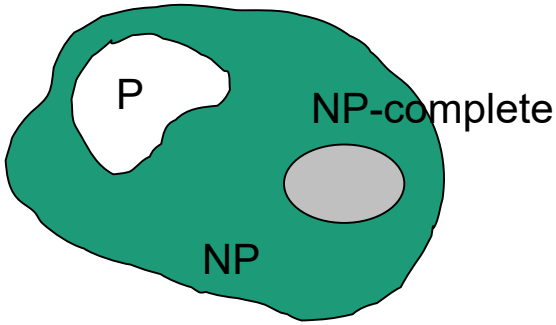
- Any problem in P is also in NP: **P ⊆ NP.**

- The big and open question is whether **NP ⊆ P or P = NP.**
  - i.e., If it is always easy to check a solution, should it also be easy to find a solution?
- Most computer scientists believe that this is false but we do not have a proof yet.



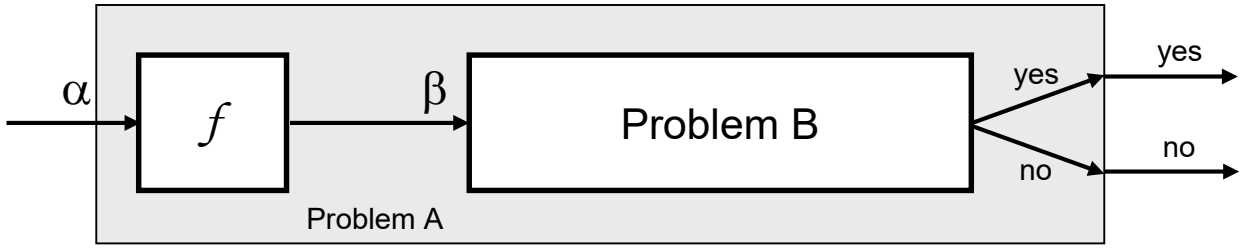
NP-Completeness (informally)

- **The NP-Complete** is a set of problems with special relationships among problems belonging to the NP.
  - None of the problems belonging to NP-Complete have been shown solvable in polynomial time.
  - However, if one problem belonging to NP-Complete is solvable in polynomial time, the answer to this problem can also tell the answer to another problem, so that all problems in this group are solved in polynomial time.



Reductions

- **Reduction** is a way of saying that one problem is "easier" than another.
  - We say that problem A is easier than problem B, (i.e., we write "A ≤ B") if we can solve A using the algorithm that solves B.
- Idea: transform the inputs of A to inputs of B.



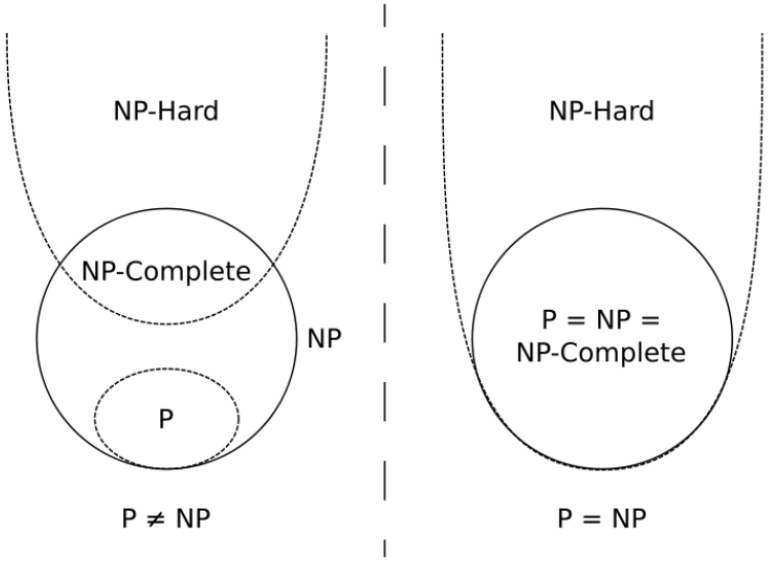
Polynomial Reductions

- Given two problems A and B, we say that A is **polynomially reducible** to B ($A \leq_p B$) if:
  - There exists a function *f* that converts the input of A to inputs of B in polynomial time.
    - A(i) = YES ⇔ B(f(i)) = YES
  - An example:
    - A: Given a set of Booleans, is at least one TRUE?
    - B: Given a set of integers, is their sum positive?
    - Transformation: $(x_1, x_2, \ldots, x_n) = (y_1, y_2, \ldots, y_n)$ where $y_i = 1$ if x_i = TRUE, $y_i = 0$ if x_i = FALSE.

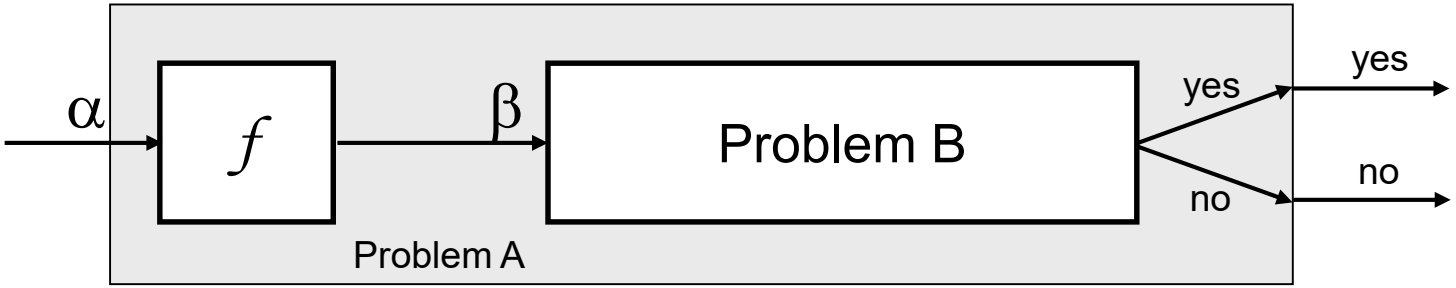
NP-Completeness (formally)

- A problem B is **NP-complete** if:
  1) $B \in \mathbf{NP}$
  2) $A \leq_p B$ for all $A \in \mathbf{NP}$

- If B satisfies only property (2), we say that B is **NP-hard.**
  - NP-complete is also NP-hard.
    - NP-hard includes a wider range of problems than NP-complete.

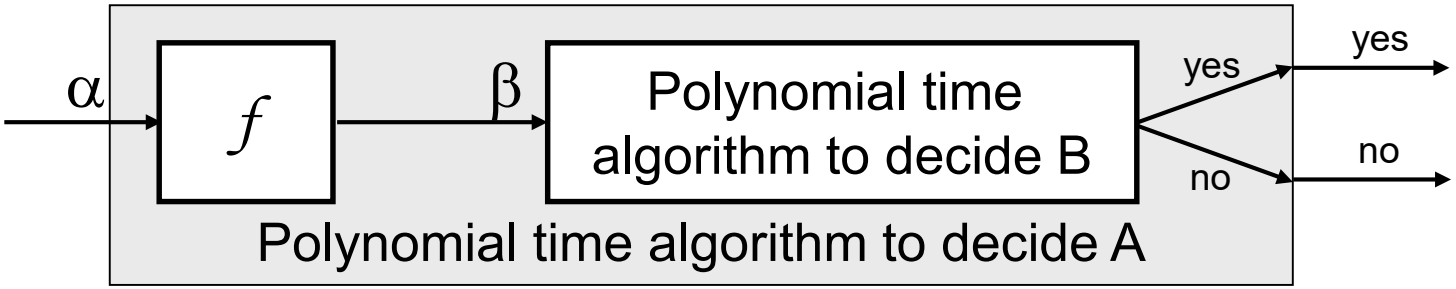


Implications of Reduction



- If $A \leq_p B$ and $B \in P$, then $A \in P$.
- If $A \leq_p B$ and $A \notin P$, then $B \notin P$.

Proving Polynomial Time



1. Use a **polynomial time** reduction algorithm to transform A into B.
2. Run a known **polynomial time** algorithm for B.
3. Use the answer for B as the answer for A.

Proving NP-Completeness In Practice

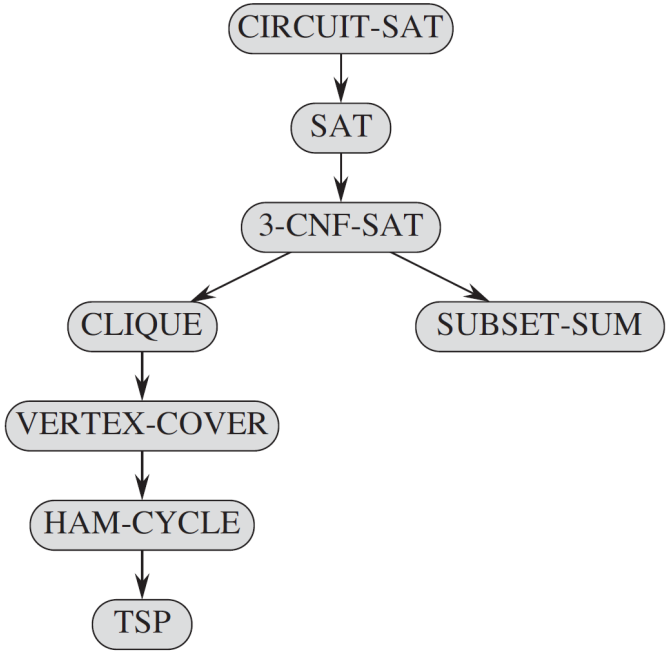
- **Actually, it is very difficult to show that a problem A is NP-hard.**
  - The condition that "all NP problems can be converted in polynomial time with A" is a very strong condition.
  - It includes all known NP problems as well as numerous unknown NP problems.

- To overcome, the proof of NP-Hard uses the following theorem.
  - **Theorem:** A problem A is NP-hard if $C \leq_p A$ for any $C \in$ NP-Hard.
  - **Proof:**
    - Problem C is NP-hard, so $L \leq_p C$ for all NP problems L by definition.
    - Since $C \leq_p A$, $L \leq_p C \leq_p A$, that is, $L \leq_p A$.

Proving NP-Completeness In Practice

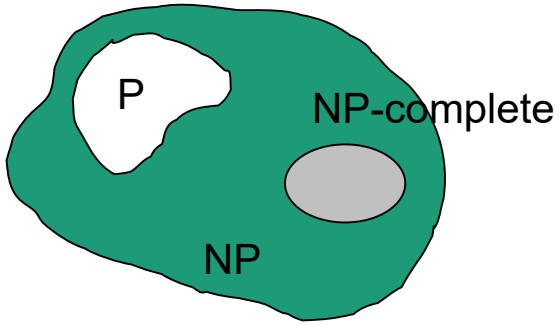
- **Theorem** makes it easier to work because you only need to convert one known NP-hard problem without having to convert all NP problems.
- However, applying this theorem is possible only when there is a known NP-hard problem.

- In 1971, Cook proved that the SAT problem is NP-Hard.
  - After Cook, many researchers convert directly or indirectly from SAT, proving that numerous problems are NP-complete.
  - A series of NP-complete problems starting with SAT is proven.



Revisit “Is P = NP?”

- **Theorem:** If any NP-Complete problem can be solved in polynomial time ⇒ then P = NP.



  - No polynomial time algorithm has been discovered for an NP-Complete problem.
    - This is the biggest open problem in CS.
    - The Clay Mathematics Institute has offered a $1 million prize for the first proof.

P & NP-Complete Problems

- **Shortest path**
  - Given a graph G = (V, E) find a shortest path from a source to all other vertices.
  - Polynomial solution: O(VE)

- **Longest simple path**
  - Given a graph G = (V, E) find a longest path from a source to all other vertices.
  - NP-complete

P & NP-Complete Problems

- **Euler tour**
  - G = (V, E) a connected, directed graph find a cycle that traverses each edge of G exactly once (may visit a vertex multiple times).
  - Polynomial solution: O(E)

- **Hamiltonian path**
  - G = (V, E) a connected, directed graph find a cycle that visits each vertex of G exactly once.
  - NP-complete

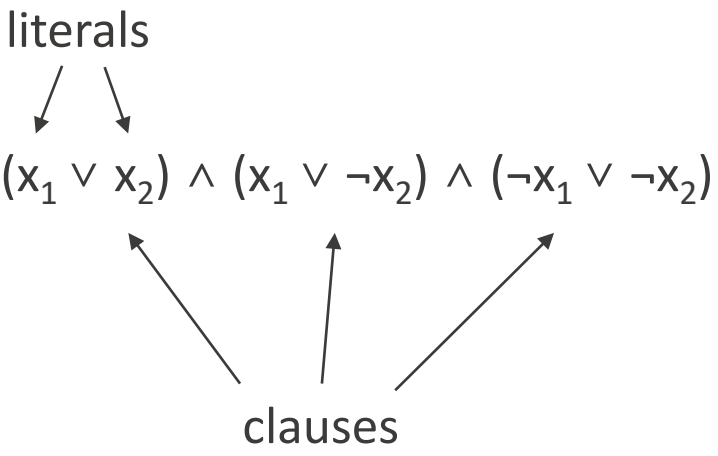
Satisfiability Problem (SAT)

- **Satisfiability problem:** given a logical expression Φ, find an assignment of values (F, T) to variables x_i that causes Φ to evaluate to T.
  - $\Phi = x_1 \vee \neg x_2 \wedge x_3 \vee \neg x_4$

- SAT was the first problem shown to be NP-complete.

CNF Satisfiability

- **CNF** is a special case of SAT.
- Φ is in "Conjunctive Normal Form" (CNF)
  - "AND" of expressions (i.e., clauses)
  - Each clause contains only "OR"s of the variables and their complements



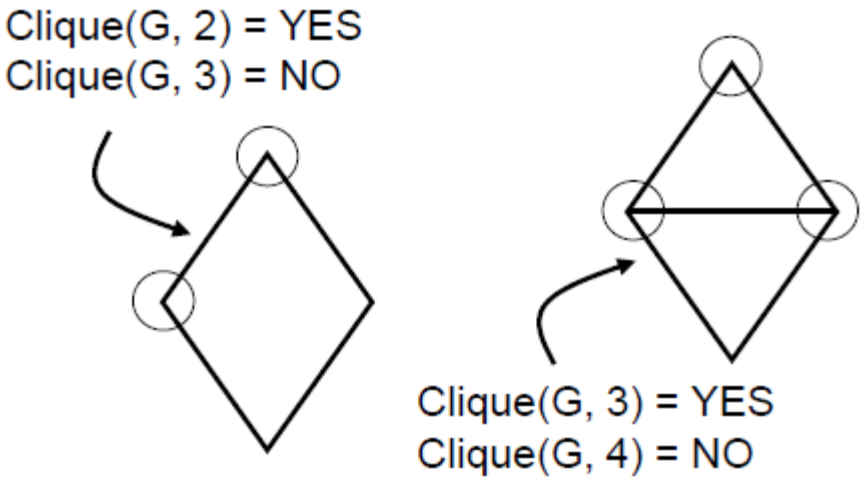
3-CNF Satisfiability

- A subcase of CNF problem:
  - Each clause has exactly 3 distinct literals.
  - E.g.: $\Phi = (x_1 \vee \neg x_1 \vee \neg x_2) \wedge (x_3 \vee x_2 \vee x_4) \wedge (\neg x_1 \vee \neg x_3 \vee \neg x_4)$

- 3-CNF is NP-Complete.
  - Interestingly enough, 2-CNF is in P.

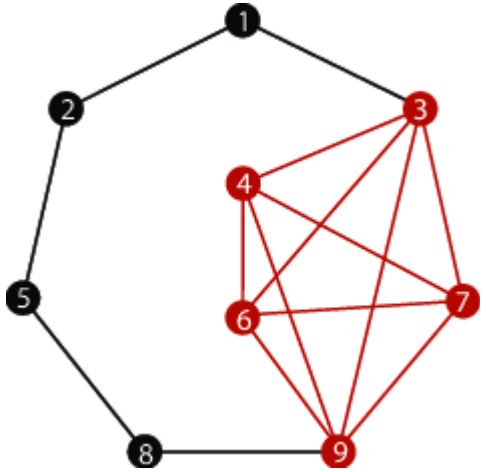
Proof Exercise

- **Clique:**
  - Given an undirected graph G = (V, E), a subset of vertices in V all connected to each other by edges in E (i.e., forming a complete graph).
- **Size of a clique:**
  - Number of vertices it contains.



Proof Exercise

- **Clique Problem:**
  - Given an undirected graph G = (V, E), does G have a clique of size k?



- **Idea of transformation:**
  - Construct a graph G such that Φ is satisfiable only if G has a clique of size k.

3-CNF-SAT $\leq_p$ Clique

- We transform a 3-CNF formula ϕ into (G, k) such that

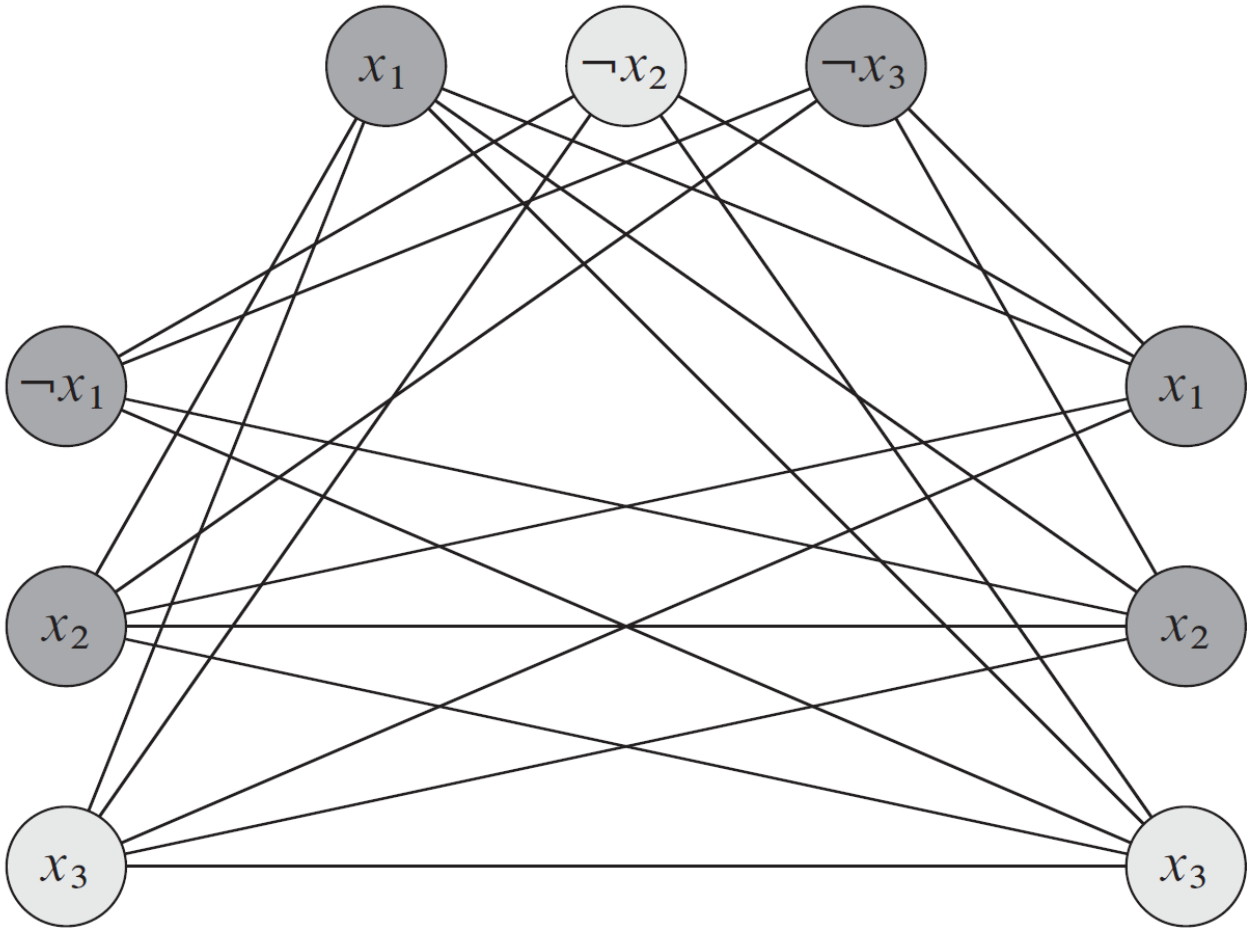
$$\phi \in \text{3-CNF} \Leftrightarrow (G, k) \in \text{CLIQUE}$$

  - If ϕ has *m* clauses, we create a graph with *m* clusters of 3 nodes each, and set $k = m$.
  - Each cluster corresponds to a clause.
  - Each node in a cluster is labeled with a literal from the clause.

  - Rule:
    - We do not connect any nodes in the same cluster.
    - We connect nodes in different clusters whenever they are not contradictory.

3-CNF-SAT $\leq_p$ Clique

$(\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2 \vee \neg x_3) \wedge (x_1 \vee x_2 \vee x_3)$



nodes = 3 (# literals in each clause)

k = # clauses

Recap

- **The theory of NP-completeness**
  - Can be used cast doubt on the existence of any polynomial-time algorithm for a given problem.
    - So far, we have concentrated mostly on designing and analyzing efficient algorithms for various problems. (Algorithms are evidence of how easy those problems are.)
    - By showing that a problem is NP-complete, we are giving evidence of how hard a problem is.

- Practically, we can think of an NP-completeness proof as a 'license' to stop looking for an efficient algorithm, and settle for approximation or to consider only special cases.


Any Question?