Lecture 21

NP-completeness II

Chinmay Nirkhe | CSE 421 Winter 2026

Calendar

- We are in the final stretch!
- I will be gone next week for two lectures — Prof. Anup Rao will be substituting for me
- I will host office hours virtually (details will be on Ed)
- Lecture 27 will be final thoughts + final review
- Section 10 will also focus on review

Mon	Tues	Wed	Th	Fri
2/23 Lec 19	2/24	2/25 Lec 20	2/26 Sec 8	2/27 Lec 21
3/2 Lec 22	3/3	3/4 Lec 23 (Anup)	3/5 Sec 9	3/6 Lec 24 (Anup)
3/9 Lec 25	3/10	3/11 Lec 26	3/12 Sec 10	3/12 Lec 27
3/16 Final				

How do we decide if a problem is in NP?

- What might a certificate/witness look like for the problem?
- Most proofs of containment in NP are ridiculously simple
  - See the examples from section for how short our proofs are
  - Usually the hard problem is proving that problems are NP-complete — i.e., the hardest NP problems

What is the “hardest” problem in NP?

- First of all, what does it mean for a problem to be hard?
- **Intuition:** Let us say that a problem B is **as hard as** than a problem A if a fast algorithm for a problem B implies a fast algorithm for a problem B.
  - Example: Max flow is as hard as bipartite matching.
  - Example: Breadth-first search is as hard as 2-coloring graphs.
  - Example: Constructing generators and power grids is as hard as MST.
  - Example: Max flow is exactly as hard as min cut.
- If an instance of problem B can be efficiently converted into an instance of a problem A this is called a **reduction**.

If problem B reduces to problem A, then

(1) a (fast) algorithm for A implies a (fast) algorithm for B .

(2) a impossibility result for B implies an impossibility result for A.

Reductions throughout this class

- We've seen reductions many times before in this class
- Anytime you used an algorithm as a subroutine — you were *morally* performing a reduction
- Examples:
  - Bipartite matching as a flow problem
  - Ship port assignment as a stable matching
  - Little Johnny walking to his mother's house as a shortest path problem

Example of a reduction

Subset Sum $\leq_p$ Decision-Knapsack

- Subset Sum: Give input $a_1, \ldots, a_n, T$, decide if there exists a subset $S \subseteq [n]$ such that $\sum_{i \in S} a_i = T$.
- Decision-Knapsack: Given input $w_1, \ldots, w_n, v_1, \ldots, v_n, W, V$, decide if there exists a subset $S \subseteq [n]$ such that $\sum_{i \in S} w_i \leq W$ and $\sum_{i \in S} v_i \geq V$.
- **Reduction:** We want to come up with an algorithm $\mathscr{A}'$ for solving Subset Sum from an algorithm $\mathscr{A}$ for solving Knapsack.
  - Given input $a_1, \ldots, a_n, T$, define $w_i = v_i \leftarrow a_i$ and $W \leftarrow T, \quad V \leftarrow T$.
  - Then run $\mathscr{A}$ on $(w_1, \ldots, w_n, v_1, \ldots, v_n, W, V)$.

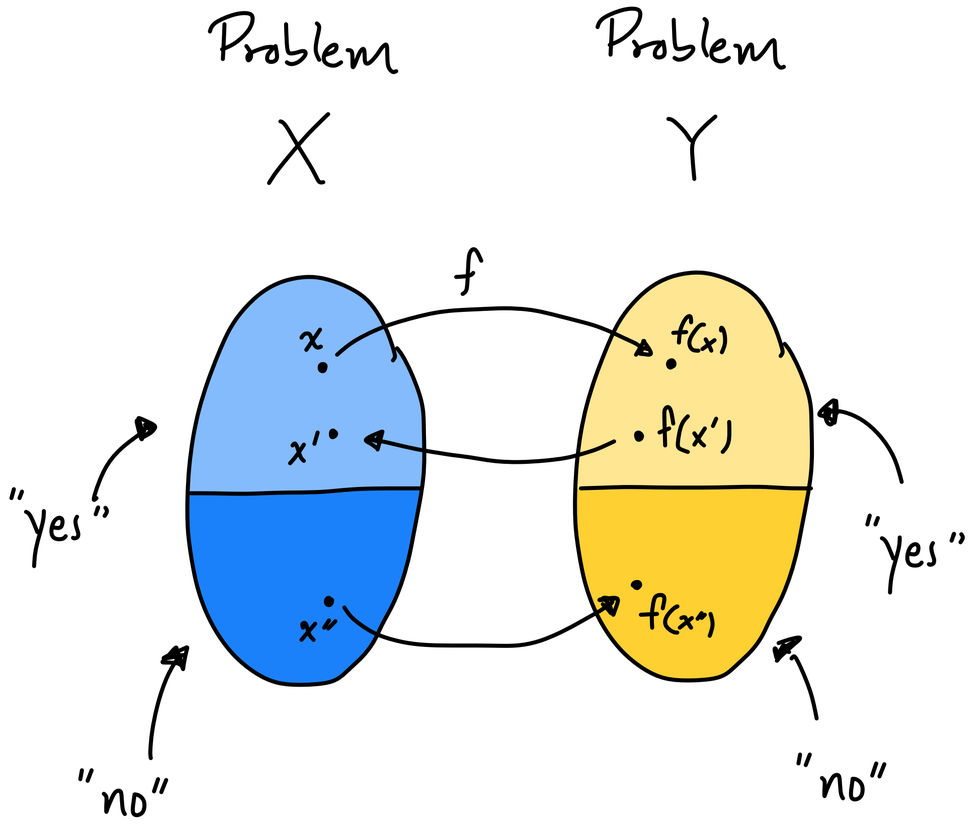
Example of a reduction

Subset Sum $\leq_p$ Decision-Knapsack

- $x = (\vec{a}, T) \in$ Subset Sum, $f(x) = (\vec{w} = \vec{v} \leftarrow \vec{a}, W = V \leftarrow T)$
- If x is a "yes" instance, then there exists $S \subseteq [n]$ s.t. $\sum_{i \in S} a_i = T$
  - Therefore, $\sum_{i \in S} w_i = \sum_{i \in S} a_i = T \leq W,$ $\sum_{i \in S} v_i = \sum_{i \in S} a_i = T \geq V.$ So $f(x)$ is a "yes" instance.
- If $f(x)$ is a "yes" instance, then there exists $S \subseteq [n]$ s.t. $\sum_{i \in S} w_i \geq W$ and $\sum_{i \in S} v_i \leq V$.
  - So $T = V \leq \sum_{i \in S} v_i = \sum_{i \in S} a_i \leq \sum_{i \in S} w_i \leq W = T$, proving that x is a "yes" instance.

Proving a reduction is correct

- The previous example is a reduction between Subset Sum and D-Knapsack
- To generate a reduction $X \leq_p Y$ between two decision problems X and Y
  - We need to find a **poly-time computable** function $f : X \to Y$ that converts instances x of X into instances $f(x)$ of Y
    - Morally, f is the precompute we would apply before the algorithm for Y
  - If for every $x \in X$ that is a "yes", then $f(x) \in Y$ is also a "yes" instance
  - If for every $f(x') \in Y$ that is a "yes", then $x' \in X$ is also a "yes" instance
    - Equiv. to: If $x'' \in X$ is a "no", then $f(x'') \in Y$ is a "no"
- For Subset-Sum $\leq_p$ D-Knapsack:
  $f(\vec{a}, V) = (\overrightarrow{w} = \vec{a}, \vec{v} = \vec{a}, W = T, V = T)$



NP-completeness

- **Simple definition:** A problem is NP-complete problem if (a) it is in NP and (b) it is the "hardest" problem in NP
- **Necessary consequence (we will show soon):** A problem X is NP-complete iff
  - If X has a poly-time algorithm, then every problem in NP has a poly-time algorithm.
  - If some problem $Y \in$ NP does not have a poly-time algorithm, then neither does X.
- **Punchline:** Once we show Knapsack is NP-complete, then if you find a way to solve Knapsack in poly-time, then you will have solved every problem in NP in poly-time.

NP-completeness

- Proving that a problem Y is the hardest problem in NP requires showing
  - that if there exists a poly-time algorithm $\mathscr{A}$ for solving Y, then for any problem $X \in$ NP, there exists a poly-time algorithm $\mathscr{A}'$ for solving X
  - This is the **reduction** of X to Y. We denote this by $X \leq_p Y$.
- Formally, we say X reduces to Y (denoted $X \leq_p Y$) if <u>any</u> instance x of X can be solved by the following algorithm:
  - In $\text{poly}(|x|)$ time, compute $y = f(x)$, an instance of the problem Y
  - Run a subroutine to decide if y is a "yes" instance of Y — returning the answer exactly
  - This is known as a Karp or many-to-one reduction.

NP-completeness

- **Formal definition:** A problem Y is NP-complete if $Y \in$ NP and for every problem $X \in$ NP, $X \leq_p Y$.
- **Theorem:** Let Y be a NP-complete problem. Then Y is solvable in poly-time iff P $=$ NP.
- **Proof:**
  - ($\Longleftarrow$) If P $=$ NP, then Y has a poly-time algorithm since $Y \in$ NP.
  - ($\Longrightarrow$) Let X be any problem in NP. Since $X \leq_p Y$, we can solve X in poly-time using the poly-time algorithm for Y as a subroutine. So $X \in$ P. So P $=$ NP.
- **Fundamental question:** Do there exist "natural" NP-complete problems?

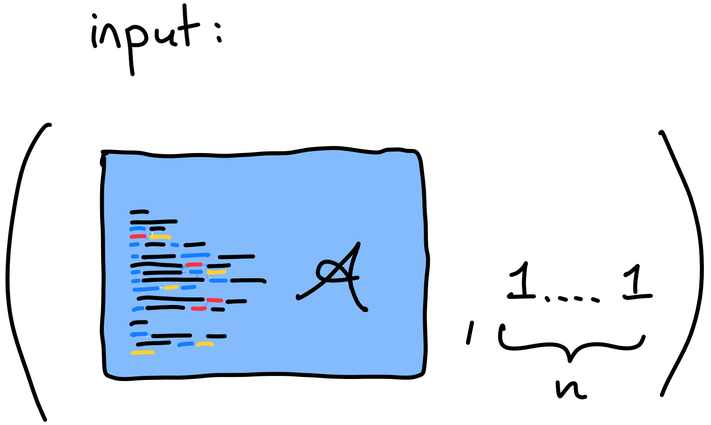
A list of NP-complete problems

- Boolean function satisfiability
- 0-1 Integer programming
- Graph problems: Vertex cover, 3-color, independent set, set cover, max cut
- Path and cycle problems: Hamiltonian path, traveling salesman
- Combinatorial optimization problems: Knapsack, Subset sum
- **Corollary:** If any of these problems admit a polynomial-time algorithm, **IFF** they all have a polynomial time algorithm!!

The “first” NP-complete problem

Satisfiability

- **Satisfiability**:
  Input: $(\langle \mathscr{A} \rangle, n)$, the description of a decision algorithm $\mathscr{A}$ and integer n in unary.
  Output: Whether there exists a π such that $\mathscr{A}(\pi) = 1$ and $|\pi| = n$.
- **Theorem**: Satisfiability is NP-complete.
- **Proof**:
  - Satisfiability is in NP as π is a proof of the satisfiability.
  - For any other problem $X \in$ NP, there exists a certifier $\mathscr{V}(x, \pi)$ such that x is a “yes” instance iff there exists a π such that $\mathscr{V}(x, \pi)$ accepts.
    - Let $n = |\pi|$ taken as input by $\mathscr{V}$.
    - Define $\mathscr{A}(\pi) :=$ as the poly-sized program computing $\mathscr{V}(x, \pi)$.
    - Then x is a “yes” instance iff exists a π such that $\mathscr{A}(\pi) = 1$ and $|\pi| = n$.
    - So $X \leq_p Y$, proving NP-completeness.



Proving more NP-complete problems

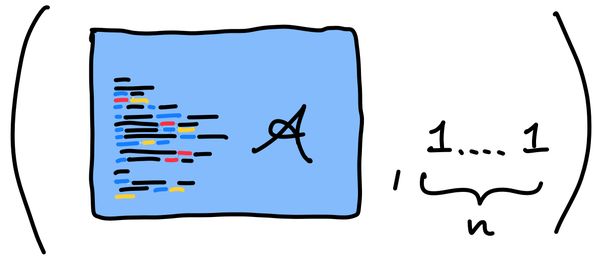
- **Recipe** for showing that problem Y is NP-complete
  - Step 1: Show that $Y \in$ NP.
  - Step 2: Choose a known NP-complete problem X.
  - Step 3: Prove that $X \leq_p Y$.
- **Correctness** of recipe: We claim that $\leq_p$ is a transitive operation.
  - If $W \leq_p X$ and $X \leq_p Y$ then $W \leq_p Y$.
  - For any problem $W \in$ NP, then $W \leq_p Y$, proving that Y is NP-complete.

3-SAT problem

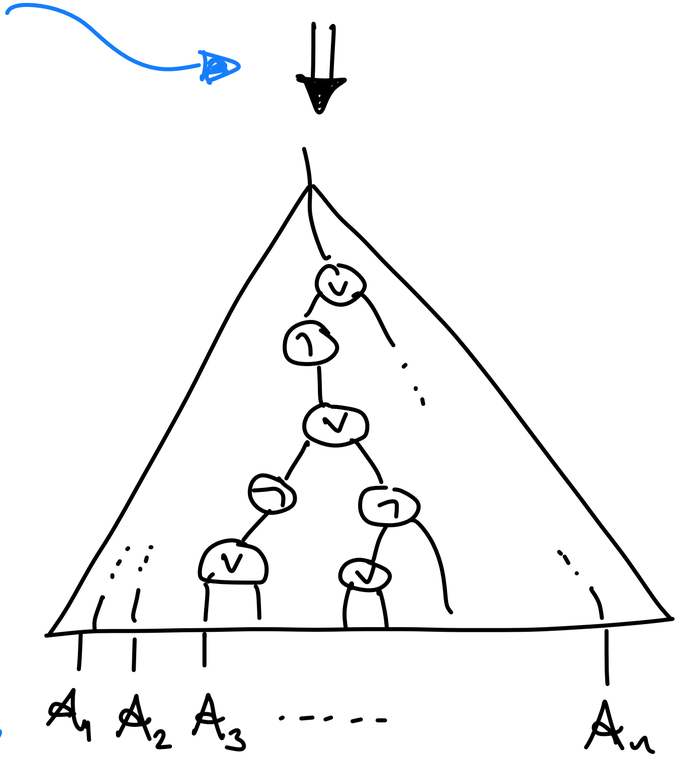
- The 3-SAT problem is the most well known of all NP-complete problems
- A boolean formula φ is a 3-SAT formula over variables $x_1, \ldots, x_n \in \{0,1\}$ if
  - $\varphi = \varphi_1 \wedge \varphi_2 \wedge \ldots \wedge \varphi_k$, the "AND" of k-subformulas
  - Each φ_j is the "OR" of ≤ 3 variables or their negations from $x_1, \ldots, x_n$.
- Examples: $\varphi(z_1, \ldots, z_4) = (z_1 \vee \neg z_2 \vee z_3) \wedge (\neg z_1 \vee \neg z_2 \vee z_4)$
- **Theorem:** 3-SAT is NP-complete.
- Proof next. For all the details, see KT 8.4.

Proof that 3-SAT is NP-complete

This transform is hard to describe exactly but is rigorous by Church–Turing thesis that both forms of computation are equivalent under polynomials.



- **Key idea:** Show that Satisfiability $\leq_p$ 3-SAT.
- **Proof:** We saw that 3-SAT is in NP (the proof is just the satisfying assignment).
  - To show that Satisfiability reduces to 3-SAT, we follow the following outline to convert an instance of Satisfiability into an instance of 3-SAT:
  - Step 1: Convert every input $(\langle \mathscr{A} \rangle, n)$ to Satisfiability into a boolean circuit G.
  - Step 2: Adjust G so that it is nicely structured: Use De Morgan's laws to ensure G consists of only OR and NOT gates, has no double negations.
  - Step 3: Label every input wire and output wire of an OR gate, with a variable z_i.
  - Step 4: Convert each gate of G into a set of clauses in the 3-SAT formula φ.
  - Step 5: Prove that φ has a solution **iff** $(\mathscr{A}, 1^n)$ is a yes instance of Satisfiability.



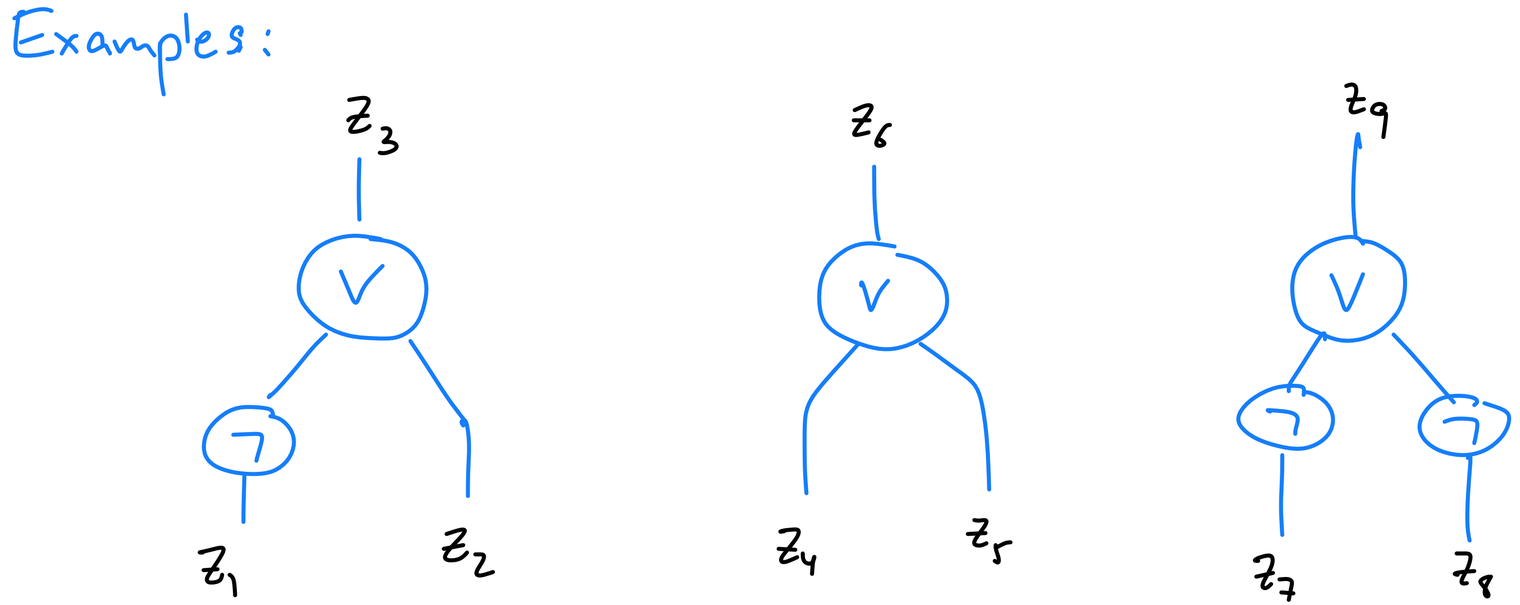
boolean circuit on the description of A

Proof that 3-SAT is NP-complete

- Step 2 elaborated:
  - De Morgan's laws:
    - Switching ANDs to ORs: $(y_1 \wedge y_2) = \neg(\neg y_1 \vee \neg y_2)$
    - Double negations: $\neg\neg y_1 = y_1$
    - Decomposing Big ORs: $y_1 \vee y_2 \vee y_3 \vee y_4 = (y_1 \vee y_2) \vee (y_3 \vee y_4)$
  - Using these boolean formula transforms, any boolean circuit can be converted into one with only OR and NOT gates

Proof that 3-SAT is NP-complete

- Step 3: Label every input wire and output wire of an OR gate, with a variable z_i.

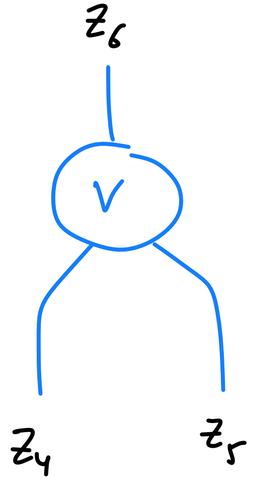


Proof that 3-SAT is NP-complete

Remember that $a \Rightarrow b$ is equiv. to $\neg a \vee b$.

- Step 4: Convert each gate of G into clauses to include in the 3-SAT formula.

Consider the following OR gate.



We want $z_6 \iff z_4 \vee z_5$.

$$= \big((z_4 \vee z_5) \Rightarrow z_6\big) \wedge \big(z_6 \Rightarrow (z_4 \vee z_5)\big)$$

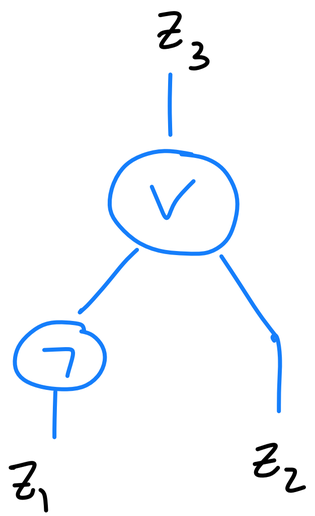
$$= \big(z_4 \Rightarrow z_6\big) \wedge \big(z_5 \Rightarrow z_6\big) \wedge \big(z_6 \Rightarrow (z_4 \vee z_5)\big)$$

$$= \big(\neg z_4 \vee z_6\big) \wedge \big(\neg z_5 \vee z_6\big) \wedge \big(\neg z_6 \vee z_4 \vee z_5\big)$$

Proof that 3-SAT is NP-complete

- Step 4: Convert each gate of G clauses to include in the 3-SAT formula.

Consider the following OR gate.



Construct the following clauses:

$$(\neg(\neg z_1) \vee z_3) \wedge (\neg z_2 \vee z_3) \wedge (\neg z_3 \vee (\neg z_1) \vee z_2)$$

$$= (z_1 \vee z_3) \wedge (\neg z_2 \vee z_3) \wedge (\neg z_3 \vee \neg z_1 \vee z_2)$$

Proof that 3-SAT is NP-complete

- Step 4: Convert each gate of G into clauses to include in the 3-SAT formula.
- **Key lemma:** If a 3-CNF φ includes $(\neg a \vee c)$, $(\neg b \vee c)$, $(\neg c \vee a \vee b)$ as clauses, then any satisfying assignment for the φ must set c to equal $a \vee b$.
- **Proof**:
  - The clauses imply the following statements: $a \Rightarrow c$, $b \Rightarrow c$, $c \Rightarrow (a \vee b)$.
  - The first two combine to $(a \vee b) \Rightarrow c$.
  - Therefore, $c \Leftrightarrow (a \vee b)$.

Proof that 3-SAT is NP-complete

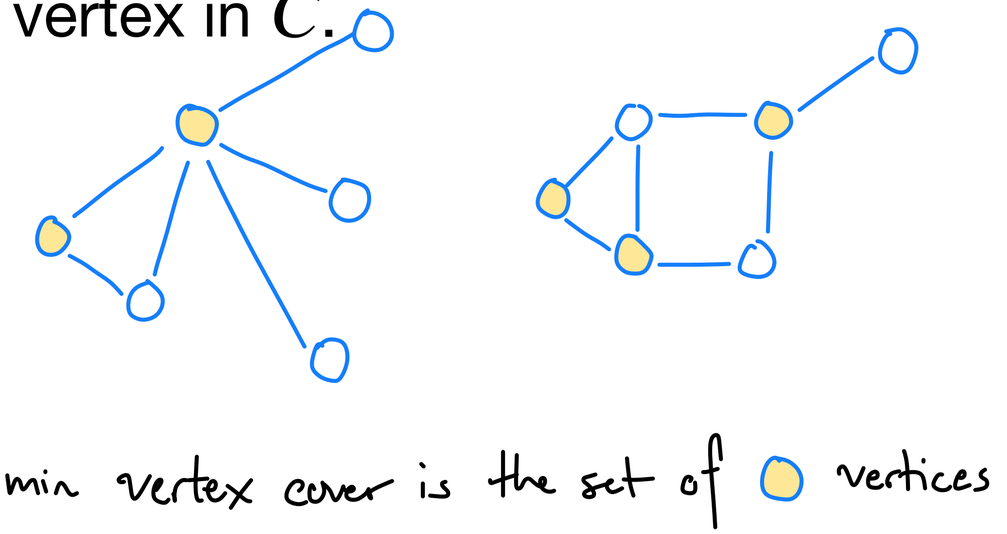
- Proving that the reduction is correct:
  - The reduction is polynomial time as it takes a constant number of passes over the input to generate (we don't need any answer more specific than this).
  - "Yes" $\rightarrow$ "Yes": If $(\langle \mathscr{A} \rangle, n)$ is a "Yes" instance, then there is an input x of length n such that $\mathscr{A}(x) = 1$. Let z be the value of the wires of the corresponding circuit G. Then z satisfies the 3-SAT φ by construction.
  - "Yes" $\leftarrow$ "Yes": If z satisfies the 3-SAT φ, then let x be the values assigned by z to the inputs. Then $G(x) = 1$ as each intermediate gate will evaluate to match z due to the previous lemma. And $\mathscr{A}(x) = 1$ iff $G(x) = 1$ so $\mathscr{A}$ is satisfiable.

General suggestions about proving NP-completeness

- There is not a clear cut set of techniques you can always apply
- Proving NP-completeness is a bit of an art —
  - To prove problem X is NP-complete, the most difficult step is finding a problem Y which is known to be is NP-complete such that $Y \leq_p X$
  - You are converting instances of Y into instances of X
  - I.e., every instance of Y is a special case of an instance of X

Minimum vertex cover problem

- **Definition:** A subset of vertices $C \subseteq V$ is a *vertex cover* of an undirected graph $G = (V, E)$ iff every edge is touched by some vertex in C.
  - V is a trivial vertex cover for G.
- **Input:** An undirected graph $G = (V, E)$
- **Output:** A minimal vertex cover C for G.

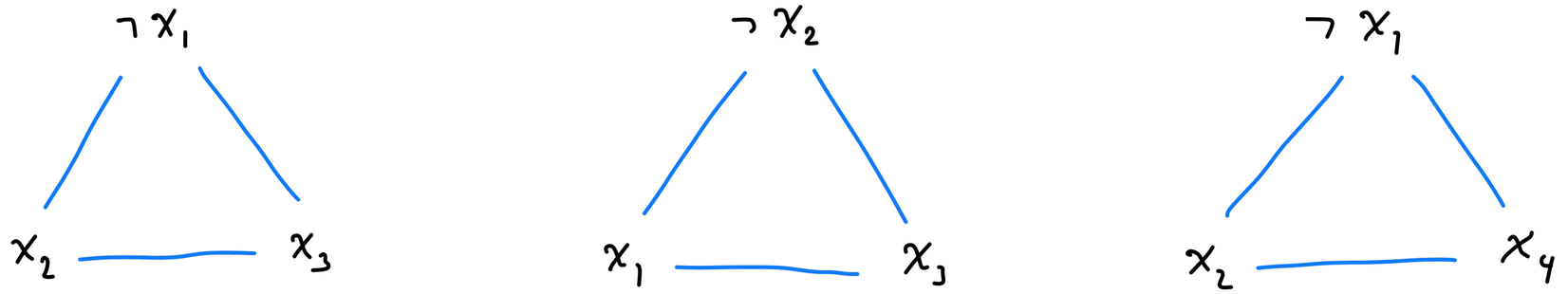


- **Next let's prove** Min Vertex Cover is a NP-complete problem

Vertex Cover is NP-complete

- We've seen that Vertex Cover is in NP. Let's show that 3-SAT $\leq_p$ Vertex Cover.
- We need to create a graph G and integer k which captures the structure of a 3-SAT formula φ.
- **The graph will have a k vertex cover iff φ was satisfiable.**
- **Construction:** G contains 3 vertices per clause, one per literal. $k = 2m$ where m is the number of clauses in φ.

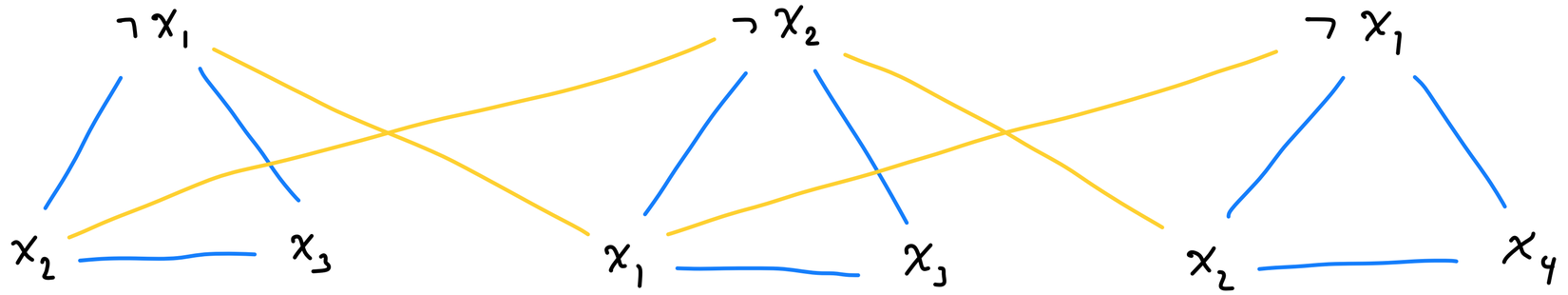
$$\varphi = (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee x_4)$$



Vertex Cover is NP-complete

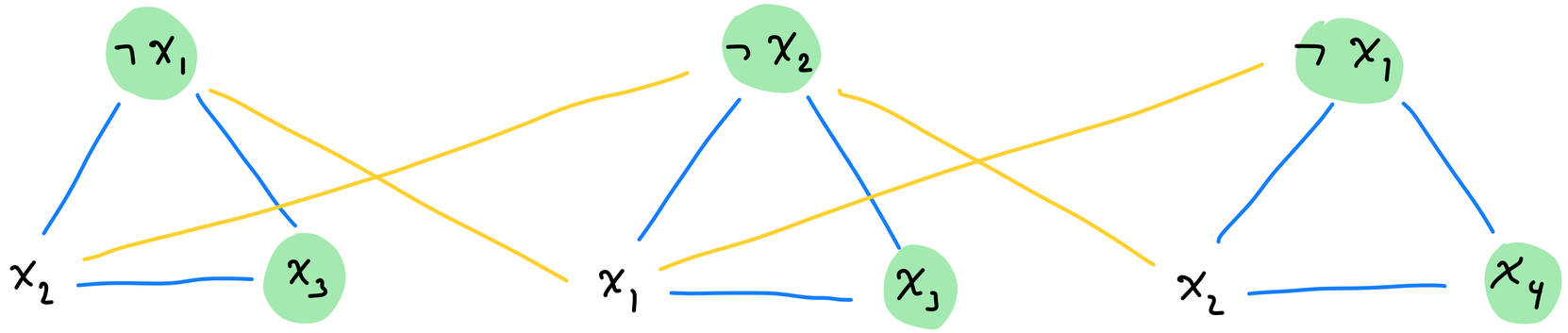
- We've seen that Vertex Cover is in NP. Let's show that 3-SAT $\leq_p$ Vertex Cover.
- We need to create a graph G and integer k which captures the structure of a 3-SAT formula φ.
- **The graph will have a k vertex cover iff φ was satisfiable.**
- **Construction:**
  - G contains 3 vertices per clause, one per literal. $k = 2m$ where m is the number of clauses in φ.
  - Add an edge between each pair of literals in a clause. Add edges connecting each variable to its negation.

$$\varphi = (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee x_4)$$



Vertex Cover is NP-complete

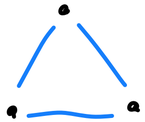
$$\varphi = (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee x_4)$$

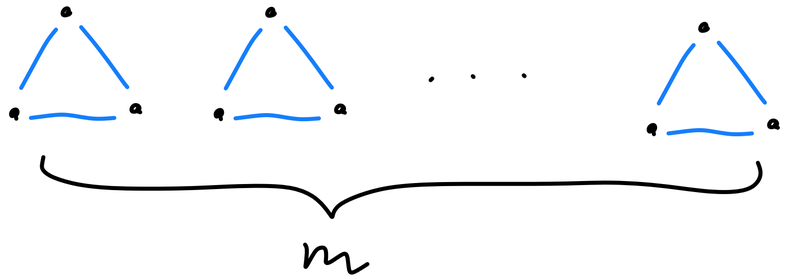


Observe: $x_1 = x_2 = 1$, $x_3 = x_4 = 0$ is a satisfying instance.

And the identified vertices form a vertex cover.

Vertex Cover is NP-complete

Observation: If a  exists in the graph, then at least 2 of the vertices must be included in a vertex cover.

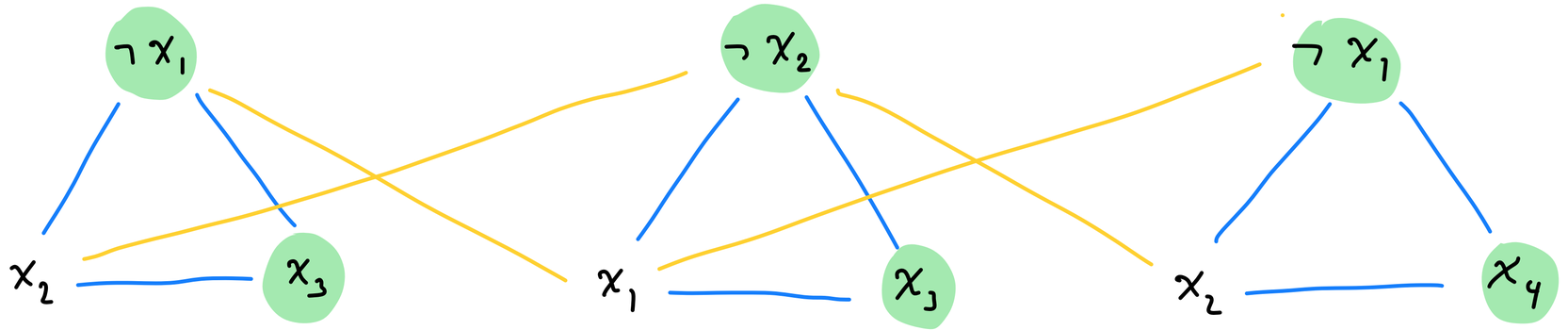
Corollary: If a graph has m disjoint triangles 

then the vertex cover must be size $\geq 2m$.

Vertex Cover is NP-complete

- **Theorem:** φ is satisfiable iff G has a vertex cover of size $\leq 2m$.
- **Proof**:
  - "Yes" $\rightarrow$ "Yes": If φ is satisfiable, let x be a satisfying assignment.
    - For each clause, pick one of the literals that must be set to be true and include the other two in the vertex cover. This is a vertex cover of size $2m$.
    - Each "triangle edge" is covered as 2 vertices are selected per triangle.
    - Each "negation edge" is covered as not selecting both endpoints would have both x_i and $\neg x_i$ to be true in the satisfying assignment.

$$\varphi = (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee x_4)$$



Vertex Cover is NP-complete

- **Theorem:** φ is satisfiable iff G has a vertex cover of size $\leq 2m$.
- **Proof**:
  - "Yes" $\leftarrow$ "Yes": If G has a vertex cover of size $\leq 2m$, then by previous lemma, the vertex cover is exactly size $2m$ and selects two vertices per triangle.
    - Set the excluded literal in each triangle to be *true*.
    - Since each "negation edge" is covered, the assignment will set at most one of x_i and $\neg x_i$ to be true.
    - Each clause is satisfied as one literate must be excluded in each clause.

$$\varphi = (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee x_4)$$