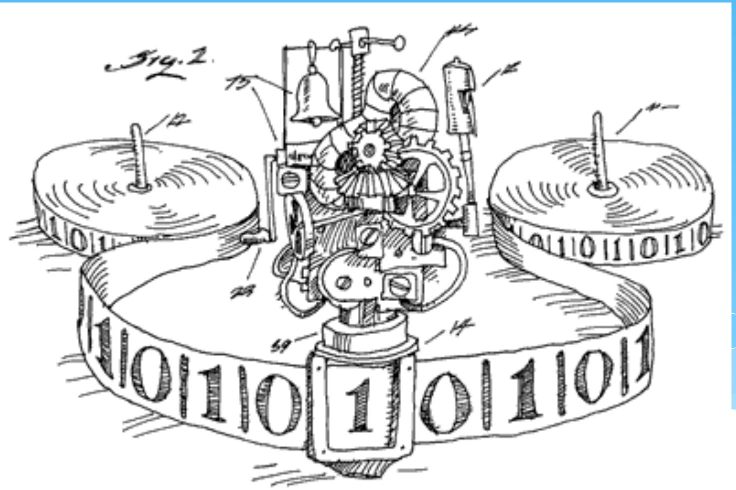
EECS 376: Foundations of Computer Science

Chris Peikert

13 March 2023



Today's Agenda

1) Recap: NP and Cook-Levin Theorem
2) NP-Completeness and mapping reductions
3) Some more NP-Complete languages

The Class NP

* **Definition:** A decision problem L is ***efficiently verifiable*** if there exists an algorithm $V(x, c)$, called a ***verifier***, satisfying:

1. $V(x, c)$ is *efficient* with respect to x, i.e., polynomial time in $|x|$.
2. For every $x \in L$, there *exists* some c such that $V(x, c)$ accepts.
3. For every $x \notin L$, $V(x, c)$ rejects for *all* c.

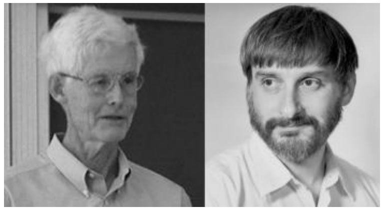
Given 1, conditions 2+3 are equivalent to:

$$x \in L \iff \exists\, c \text{ s.t. } V(x, c) \text{ accepts.}$$

Definition: the class **NP** = the set of all efficiently verifiable languages.
I.e.: $L \in \mathbf{NP}$ if L is efficiently verifiable.

Two Amazing Works (Given Turing Awards)

Cook-Levin (1971): SAT is "NP-hard." In particular:
If SAT is in P, then all of NP is in P, i.e., P=NP.
(Easy: if SAT is not in P, then P $\neq$ NP.)



So, to resolve P vs. NP, we "just" need to determine the status of SAT!

Karp (1972): TSP, Ham-Cycle, Subset Sum, …
all of these are "equivalent" to SAT.

Either all of them are in P (so P=NP), or none are (so P $\neq$ NP).

Boolean Formulas and SAT

- A Boolean ***formula*** is a formula involving Boolean literals and operators, e.g.,
  $\phi(x, y, z) = (\neg x \vee y) \wedge (\neg x \vee z) \wedge (y \vee z) \wedge (x \vee \neg z)$
- An ***assignment*** is a map from variables to truth values, e.g., $x = 0, y = 1, z = 0.$
- A ***satisfying assignment*** for ϕ is an assignment that makes ϕ evaluate to *true*.
- A formula ϕ is ***satisfiable*** if it has a satisfying assignment.
- $\mathrm{SAT} = \{\phi : \phi \text{ is a satisfiable Boolean formula}\}$

Cook-Levin Outline

* **Theorem [Cook-Levin]:** If SAT $\in$ **P**, then **NP** $\subseteq$ **P**.
* Let D_{SAT} be an efficient decider for SAT.
* Let L $\in$ **NP**, so L has an efficient verifier V.
* Goal: L $\in$ **P** via efficient decider D_L that uses D_{SAT} & V.
* $D_L(x)$:
  * Efficiently construct a poly-sized Boolean formula $\phi_{V,x}$ so that:
  * $x \in L \iff \phi_{V,x}$ is satisfiable.
  * Output $D_{\text{SAT}}(\phi_{V,x})$.

Reductions, Then and Now

* **Recall:**
  * We proved that L_{BARBER} is ***undecidable*** by an ingenious ad-hoc argument.
  * We proved that many other languages (L_{HALT}, L_{EQ}, …) are undecidable via ***Turing reductions***. E.g., $L_{\text{ACC}} \leq_T L_{\text{HALT}}$ shows that L_{HALT} is also undecidable.
* **Now:**
  * We proved that SAT is "**NP**-hard" by an ingenious ad-hoc argument.
  * We will prove that other languages are **NP**-hard by a special kind of reduction: ***polynomial-time mapping reduction***.

Poly-Time Mapping Reductions

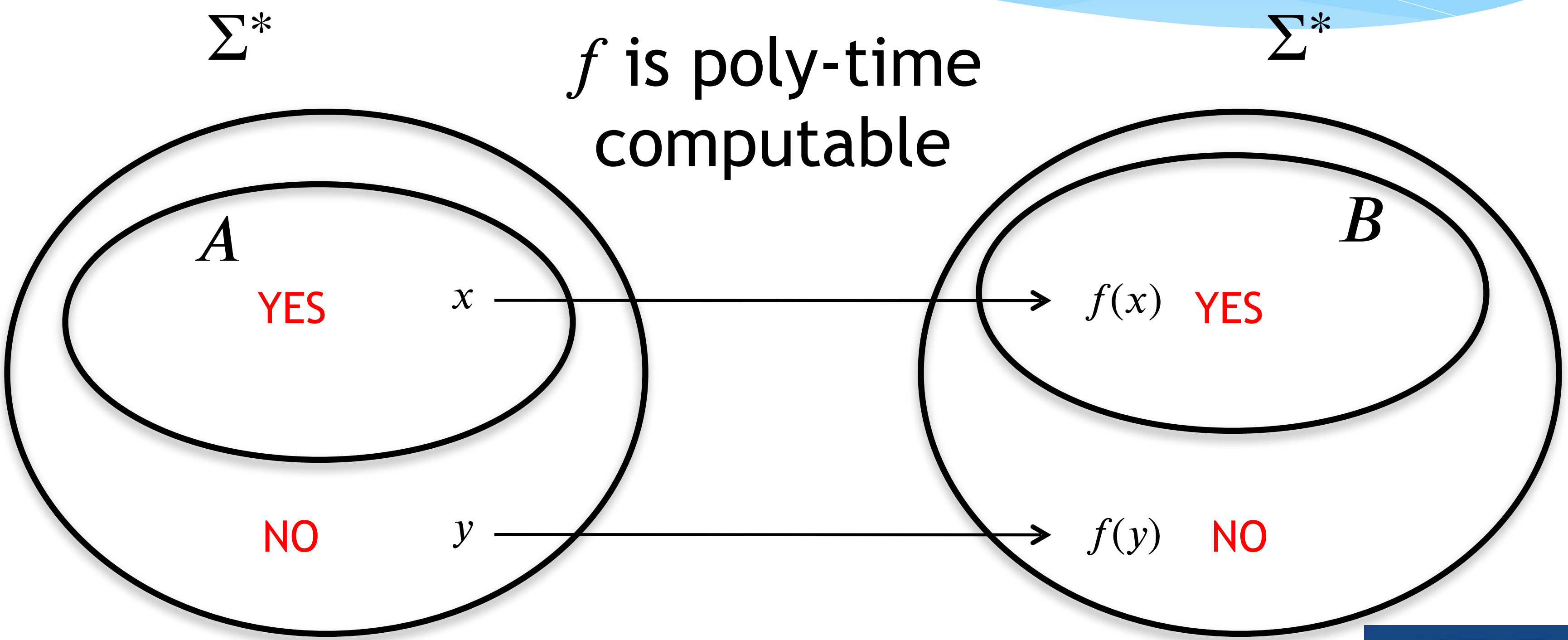
- **Theorem [Cook-Levin]:** For any $L \in \mathbf{NP}$, there is a poly-time algorithm f such that $x \in L \iff f(x) \in \text{SAT}$.
- **Definition:** Language A is ***polynomial-time mapping reducible*** to language B, written $A \leq_p B$, if there is a polynomial-time algorithm f such that:

No "flipping" the answer; only one "oracle call"

$$x \in A \iff f(x) \in B.$$

- **Recall:** If $A \leq_T B$ and B is decidable then so is A.
- **Theorem:** If $A \leq_p B$ and $B \in \mathbf{P}$ then $A \in \mathbf{P}$.
- **Proof:** given x, run B-decider on $f(x)$.

$A \leq_p B$



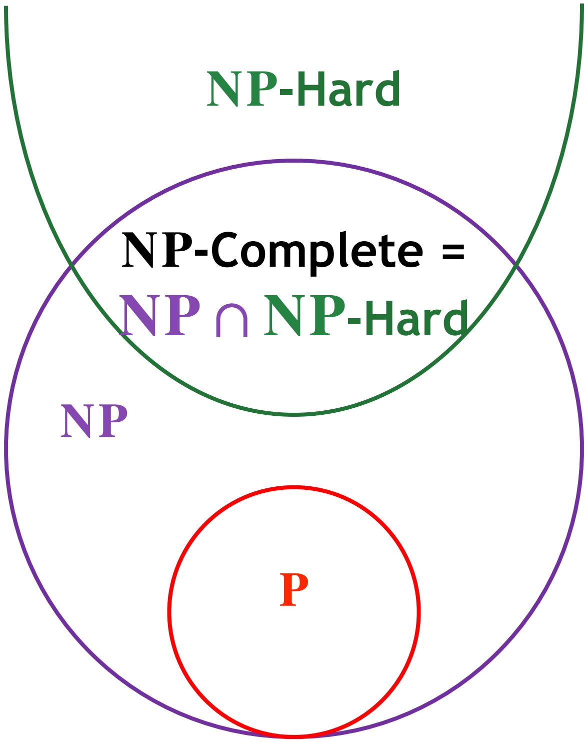
- **Remark:** f need not be injective nor surjective!

$\mathbf{NP}$-Completeness

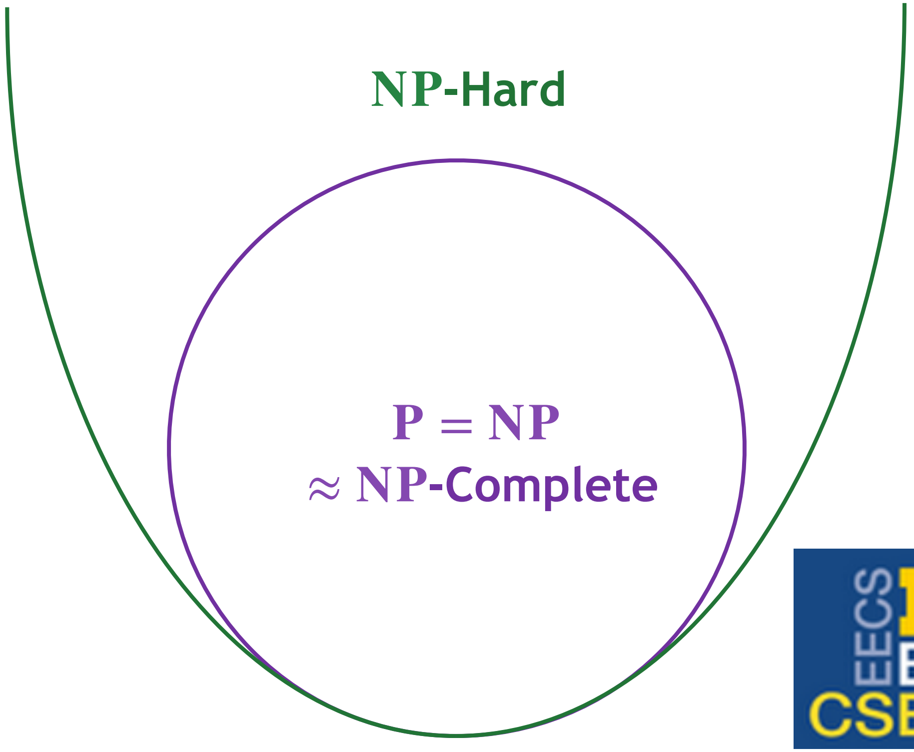
* **Theorem [Cook-Levin]:** For every $A \in \mathbf{NP}$, $A \leq_p$ SAT.
* **Definition:** Language B is $\mathbf{NP}$-*Hard* if $A \leq_p B$ for *all* $A \in \mathbf{NP}$.
* **Definition:** Language B is $\mathbf{NP}$-*Complete* if:
  1. $B \in \mathbf{NP}$
  2. B is $\mathbf{NP}$-Hard
* **We saw:**
  * SAT $\in \mathbf{NP}$
  * SAT is $\mathbf{NP}$-Hard
  * Thus, SAT is $\mathbf{NP}$-Complete.

NP-Hard and -Complete

P ⊂ NP



P = NP



More **NP**-Complete Languages

* **Question:** To prove that B is NP-Hard, must we redo Cook-Levin?
* **Answer:** No, because $\leq_p$ is a transitive relation! Just show SAT $\leq_p B$.
* **Claim:** If $A \leq_p B$ and $B \leq_p C$, then $A \leq_p C$.
* **Proof:** HW…
* **Example 1:** 3SAT
* **Definition:** A ***3CNF clause*** is an OR of 3 literals, e.g., $(x \vee \neg y \vee z)$.
* **Definition:** A ***3CNF formula*** is an AND of 3CNF clauses, e.g., $(x \vee \neg y \vee z) \wedge (\neg x \vee z \vee w) \wedge \ldots$
* **Definition:** $\text{3SAT} = \{\phi : \phi \text{ is a satisfiable 3CNF formula}\}$

More $\mathbf{NP}$-Complete Languages

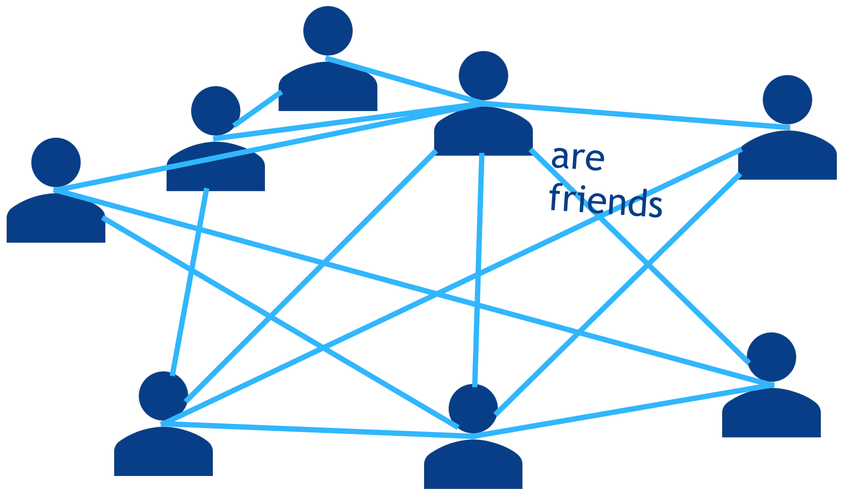
* **Definition:** $3\text{SAT} = \{\phi : \phi \text{ is a satisfiable 3CNF formula}\}$
* **Theorem:** $\text{SAT} \leq_p 3\text{SAT}$ (proof given in the notes)
* **Conclusion:** 3SAT is $\mathbf{NP}$-Hard.
* **Proof:** Let $A \in \mathbf{NP}$. We know from Cook-Levin that $A \leq_p$ SAT $\leq_p$ 3SAT. By transitivity: $A \leq_p$ 3SAT.
* We also can show that $3\text{SAT} \in \mathbf{NP}$: given (ϕ, c) check that ϕ is in 3CNF format, and that c satisfies ϕ.
* Thus, 3SAT is $\mathbf{NP}$-Complete.

More $\mathbf{NP}$-Complete Languages

* **In general:** To show that a language B is $\mathbf{NP}$-Complete:
  1. Show that $B \in \mathbf{NP}$.
     * Write a verifier V for B, show that it is correct and efficient.
  2. Show that $A \leq_p B$ for some *known* $\mathbf{NP}$-Complete A.
     * Write a procedure f mapping instances of A to instances of B, show that it is efficient and correct:
       * $x \in A \iff f(x) \in B$ (both directions!)
       * Does **NOT** require converting instances of B to instances of A! Typically, many valid instances of B will *not* be output by f. (I.e., f is not surjective.)

Example: Clique Problem
(Friendship problem)

* **Recall:** Given a group of people and their (non-)friendships, are there k people that are *mutual* friends?
  * $\text{CLIQUE} = \{(G, k) : G \text{ is a graph with a clique of size } k\}$
* Straightforward to see that $\text{CLIQUE} \in \mathbf{NP}$. We now show that it is $\mathbf{NP}$-Hard via reduction: $3\text{SAT} \leq_p \text{CLIQUE}$.



$3SAT \leq_p CLIQUE$

Goal: "*transform*" 3CNF formula ϕ into (G, k) such that:
- ϕ satisfiable $\Longleftrightarrow$ G has a k-clique

* Consider the following example formula:

$$\phi = (x \vee y \vee z) \wedge (\neg x \vee \neg y \vee \neg z) \wedge (x \vee \neg y \vee z)$$

* Think of each clause as a "house." Is there a group of three "friends," each one living in a different house?
  * Each literal is a "person."
  * Two people/literals are compatible ("friends") if they live in different houses, and can both assigned *true* simultaneously.
    * x in clause 1 is compatible with x in clause 3
    * x in clause 1 is compatible with $\neg y$ in clause 2
    * x in clause 1 is *not* compatible with $\neg x$ in clause 2

$3SAT \leq_p CLIQUE$

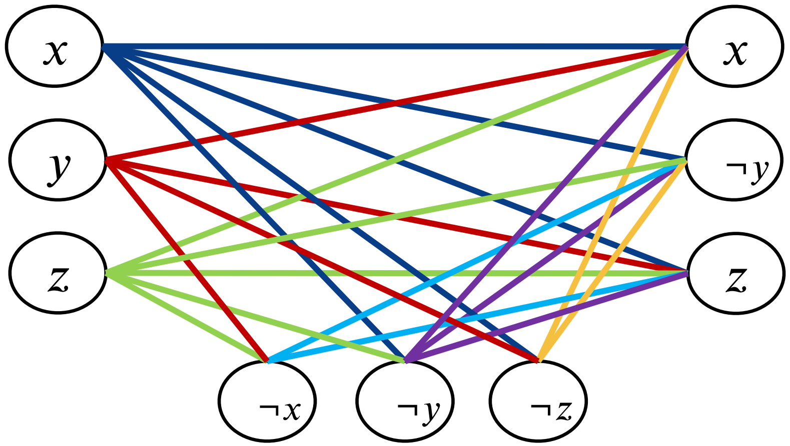
Goal: "*transform*" 3CNF formula ϕ into (G, k) such that:

- ϕ satisfiable $\iff$ G has a k-clique

* Consider the following example formula:

$$\phi = (x \vee y \vee z) \wedge (\neg x \vee \neg y \vee \neg z) \wedge (x \vee \neg y \vee z)$$

* **Result:**



3SAT $\leq_p$ CLIQUE

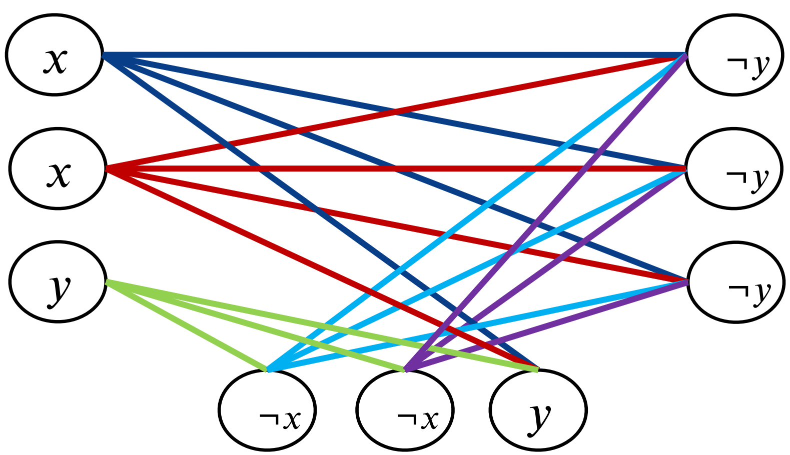
Goal: "*transform*" 3CNF formula ϕ into (G, k) such that:

- ϕ satisfiable $\Longleftrightarrow$ G has a k-clique

* Example that is not satisfiable:

$$\phi = (x \vee x \vee y) \wedge (\neg x \vee \neg x \vee y) \wedge (\neg y \vee \neg y \vee \neg y)$$

* **Result:**



$3SAT \le_p$ CLIQUE

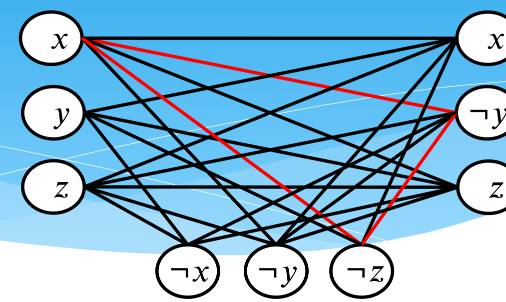
Goal: "*transform*" 3CNF formula ϕ into (G, k) such that:
- ϕ satisfiable $\Longleftrightarrow$ G has a k-clique

- To show $3SAT \le_p$ CLIQUE, we need to:
  - Define an f that converts a formula ϕ to a some (G, k).
  - Show that f is correct: $\phi \in$ 3SAT $\Longleftrightarrow$ $(G, k) \in$ CLIQUE.
  - Show that f is efficient.

(graph, int) **transformFormula**(formula ϕ):
1. $G \leftarrow$ empty graph
2. for each literal $l_i \in \phi$: add a vertex v_i to G
3. for each pair of literals l_i, l_j from *distinct* clauses of ϕ:
4. if $l_i \neq \neg l_j$: add the edge (v_i, v_j) to G
5. **return** (G, k) where k is the number of clauses in ϕ

Correctness Analysis (1/2)

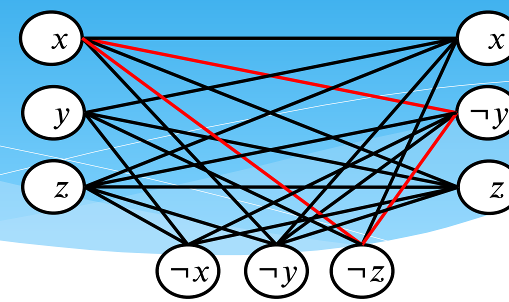
```
(graph, int) transformFormula(formula ϕ):
G ← empty graph
for each literal l_i ∈ ϕ: add a vertex v_i to G
for each pair of literals l_i, l_j from distinct clauses of ϕ:
    if l_i ≠ ¬l_j: add the edge (v_i, v_j) to G
return (G, k) where k is the number of clauses in ϕ
```



* **Direction 1:** $\phi \in \text{3SAT} \Longrightarrow (G, k) \in \text{CLIQUE}$
* Suppose that $\phi = C_1 \wedge C_2 \wedge \ldots \wedge C_k$ has k clauses C_i.
* Consider any satisfying assignment α of ϕ.
* Since ϕ is satisfied by α, for $1 \leq i \leq k$, each C_i (e.g., $x \vee y \vee z$) has some literal ℓ_i that is true under α.
* We claim that $\{\ell_1, \ell_2, \ldots, \ell_k\}$ is a k-clique in G.
  * Consider any two literals ℓ_i and ℓ_j from different clauses.
  * If $\ell_i = \ell_j$, then there's an edge between them in G.
  * Otherwise, ℓ_i and ℓ_j must refer to *different variables*! (Why?)
  * Hence, they also have an edge between them.

Correctness Analysis (2/2)

(graph, int) **transformFormula**(formula ϕ):
1. $G \leftarrow$ empty graph
2. for each literal $l_i \in \phi$: add a vertex v_i to G
3. for each pair of literals l_i, l_j from *distinct* clauses of ϕ:
4.   if $l_i \neq \neg l_j$: add the edge (v_i, v_j) to G
5. **return** (G, k) where k is the number of clauses in ϕ

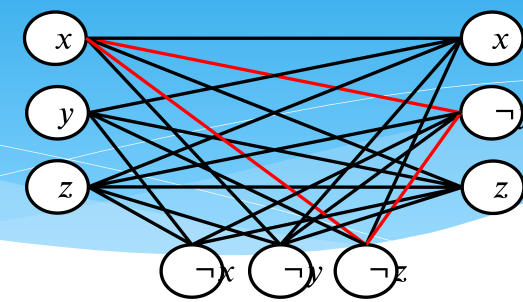


* **Direction 2:** $(G, k) \in \text{CLIQUE} \Longrightarrow \phi \in \text{3SAT}$
* Suppose that $\{\ell_1, \ell_2, \ldots \ell_k\}$ is a k-clique in G.
* Define an assignment α of ϕ by taking each literal ℓ_i and setting the underlying variable so that ℓ_i is true (and set any remaining variables arbitrarily).
* By construction of G and that $\{\ell_1, \ldots, \ell_k\}$ is a clique:
  * There are *no conflicts* in setting the variables this way.
    * For any edge (ℓ_i, ℓ_j), either $\ell_i = \ell_j$ or they refer to different variables.
  * The literals ℓ_i are from *distinct* clauses. (Why?)
* Since α satisfies each clause of ϕ, it satisfies ϕ!

Runtime Analysis

(graph, int) **transformFormula**(formula ϕ):

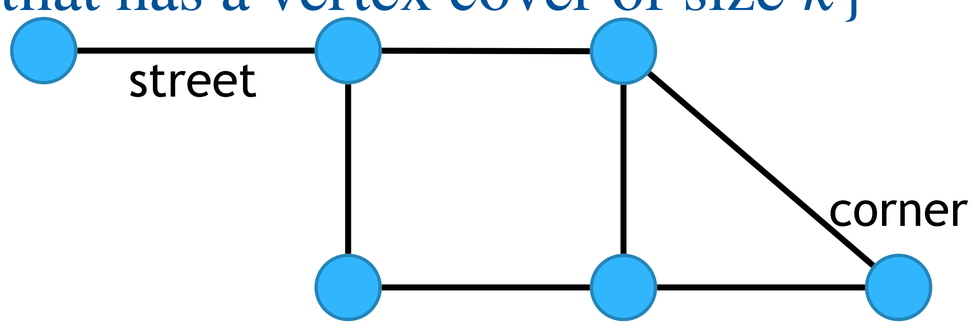
1. $G \leftarrow$ empty graph
2. for each literal $l_i \in \phi$: add a vertex v_i to G
3. for each pair of literals l_i, l_j from *distinct* clauses of ϕ:
4. if $l_i \neq \neg l_j$: add the edge (v_i, v_j) to G
5. **return** (G, k) where k is the number of clauses in ϕ



- **Claim: transformFormula** is efficient.
- Say that $\phi = C_1 \wedge C_2 \wedge \ldots \wedge C_k$ has k clauses.
- The input size is $\geq k$.
- Step 1 takes constant time.
- Step 2 takes $O(k)$ time.
- Steps 3-4 take $O(k^2)$ time.
- So, runtime is polynomial in the input size.

Vertex Cover
("Starbucks Problem")

- Given a city, is it possible to put stores on k street corners so that *every* street is "covered" by some store?
- **Formally:** A ***vertex cover*** of a graph $G = (V, E)$ is a set $C \subseteq V$ s.t. for all $(u, v) \in E$: $u \in C$ or $v \in C$ (or both).
  (all *edges* are "***covered***" by C)
  - $\text{VERTEXCOVER} = \{(G, k) : G \text{ is an undirected graph that has a vertex cover of size } k\}$

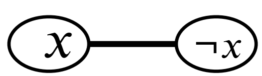


VERTEXCOVER is NP-C

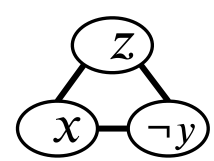
- **Claim:** VERTEXCOVER is $\mathbf{NP}$-Complete
- **Proof:** General Strategy:
  1. $\text{VERTEXCOVER} \in \mathbf{NP}$ (Exercise)
  2. $A \leq_p \text{VERTEXCOVER}$ for *some* $\mathbf{NP}$-C language A
- We will show that $3\text{SAT} \leq_p \text{VERTEXCOVER}$.
- **Detailed Goal:** Show an algorithm f
  1. $f : \{3CNF\ formula\} \rightarrow \{(graph,\ k)\}$ (CGraph f(formula phi);)
     $f(\phi) = (G, k)$
  2. f is efficient
  3. ϕ is satisfiable $\iff G$ has a vertex cover of size k

3SAT $\leq_p$ VERTEXCOVER

- **Proof idea:**
  - Given a 3CNF formula ϕ with n variables, m clauses:
  - Make subgraphs ("***gadgets***") that represent variables and clauses.
  - Connect the gadgets together in the right way.



Variable gadget for x



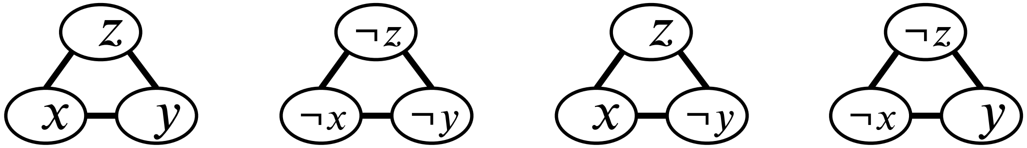
Clause gadget for $(x \vee \neg y \vee z)$

3SAT $\leq_p$ VERTEXCOVER

* Include the edge (u, v) if:
  * u is in a ***variable gadget*** and v is in a ***clause gadget*** *AND*
  * u and v have the ***same variable label*** (e.g., x, $\neg z$, etc.)
* **Example:**
  $(x \vee y \vee z) \wedge (\neg x \vee \neg y \vee \neg z) \wedge (x \vee \neg y \vee z) \wedge (\neg x \vee y \vee \neg z)$

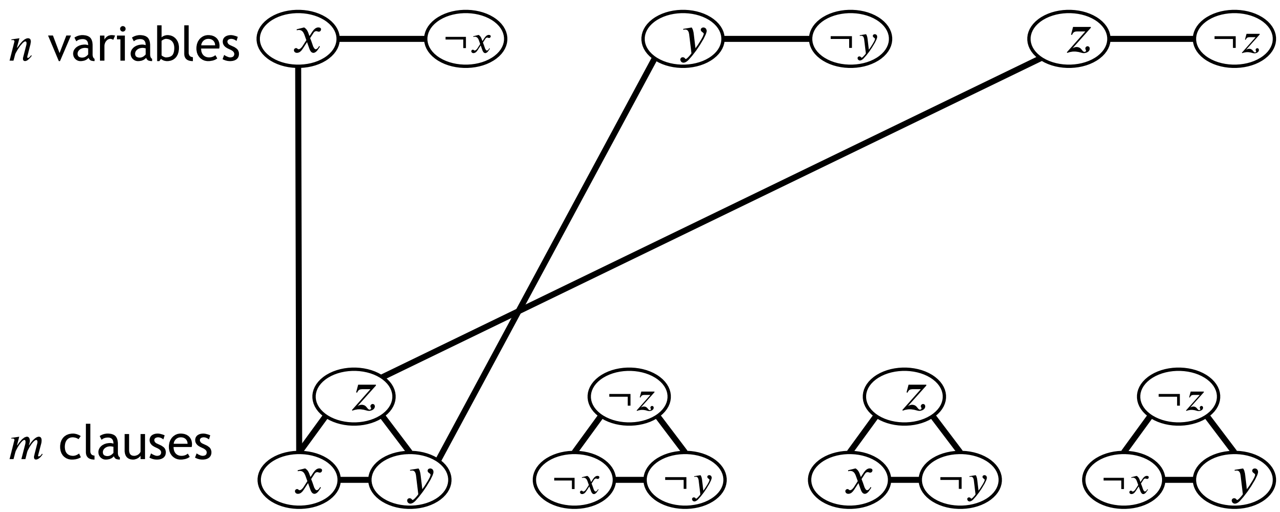
n variables

m clauses



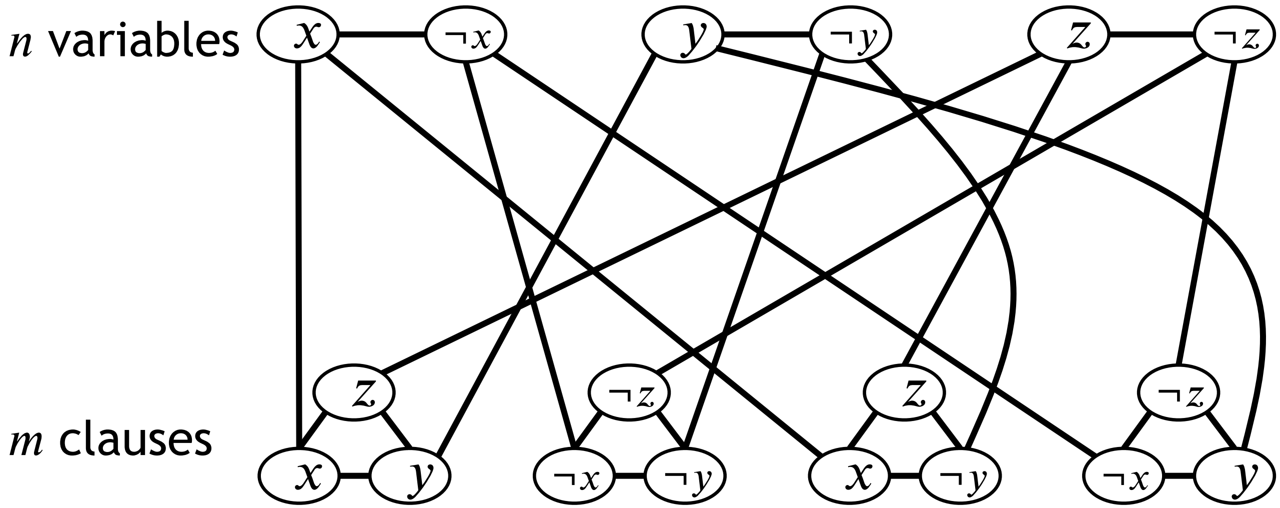
3SAT $\leq_p$ VERTEXCOVER

- Include the edge (u, v) if:
  - u is in a *variable gadget* and v is in a *clause gadget* *AND*
  - u and v have the *same variable label* (e.g., x, $\neg z$, etc.)
- **Example:**
  $(x \vee y \vee z) \wedge (\neg x \vee \neg y \vee \neg z) \wedge (x \vee \neg y \vee z) \wedge (\neg x \vee y \vee \neg z)$



3SAT $\leq_p$ VERTEXCOVER

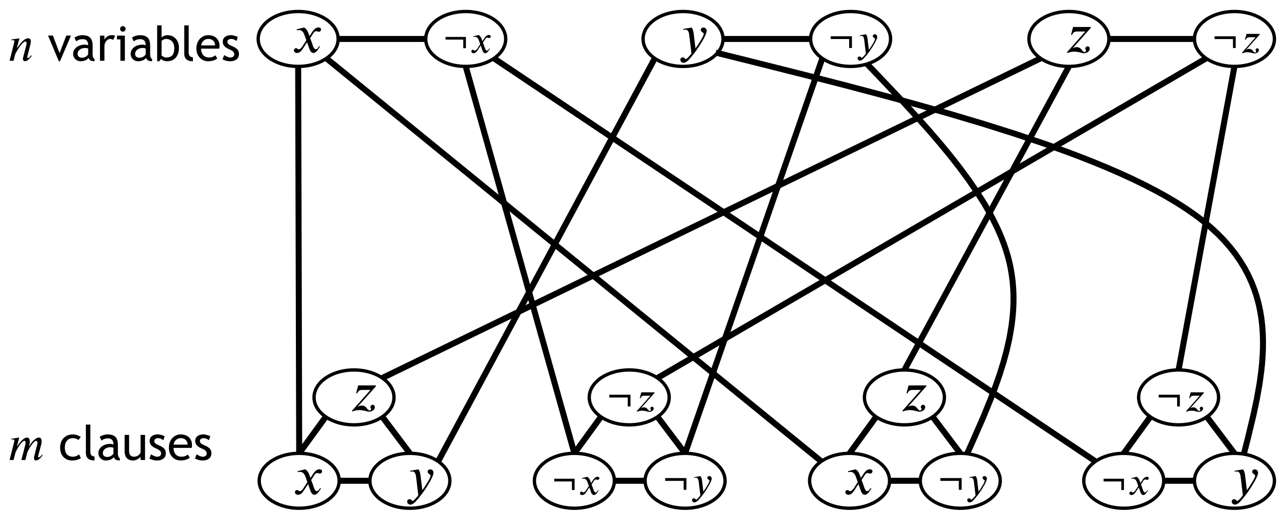
- Include the edge (u, v) if:
  - u is in a ***variable gadget*** and v is in a ***clause gadget*** *AND*
  - u and v have the ***same variable label*** (e.g., x, $\neg z$, etc.)
- **Example:**
  $(x \vee y \vee z) \wedge (\neg x \vee \neg y \vee \neg z) \wedge (x \vee \neg y \vee z) \wedge (\neg x \vee y \vee \neg z)$



3SAT $\leq_p$ VERTEXCOVER

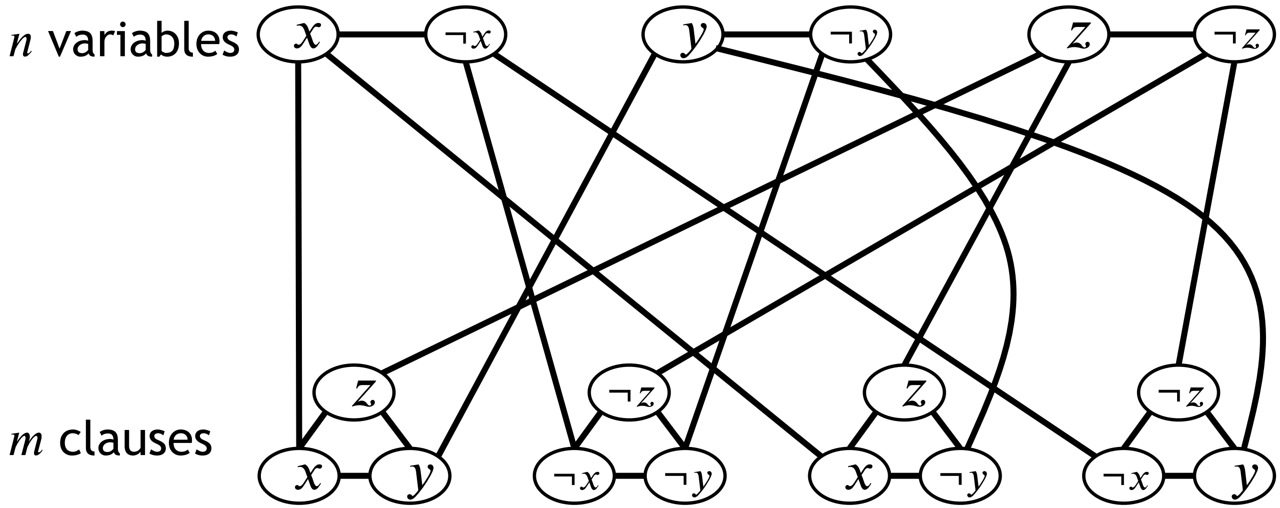
$f(\phi) = (G, n + 2m)$

* **Claim:** Let ϕ be a 3CNF with n variables and m clauses; then
  1. The graph G is constructible in time $O(mn)$.
  2. ϕ is satisfiable iff G has a V.C. of size $k = n + 2m$.



3SAT $\leq_p$ VERTEXCOVER

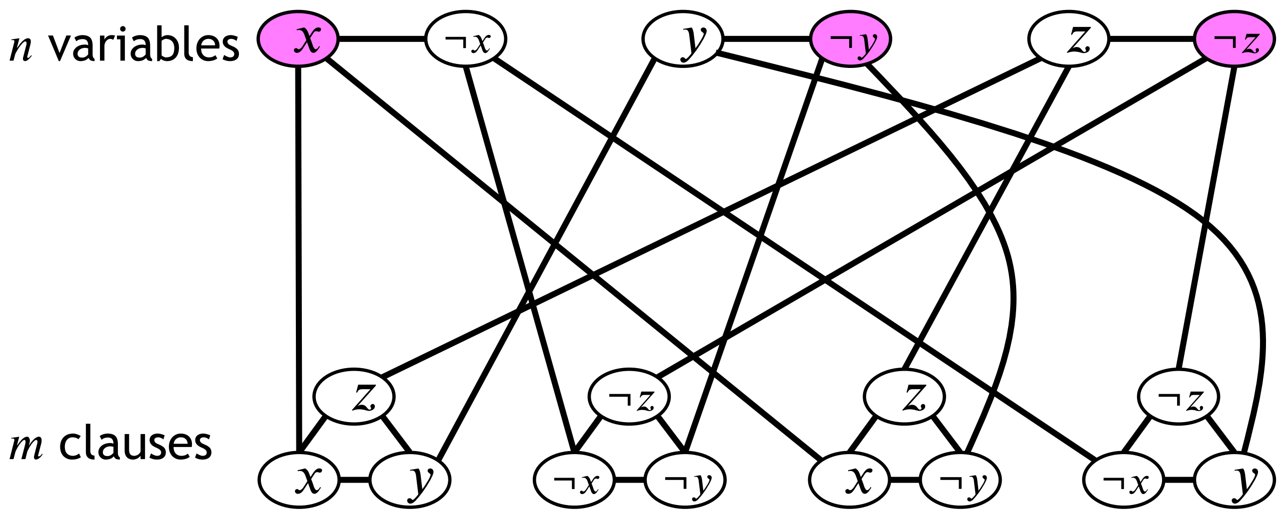
* **Observation:** Any vertex cover has size $\geq n + 2m$.
  * Needs ≥ 1 node per variable gadget and ≥ 2 nodes per clause
* **Example:**
  $(x \vee y \vee z) \wedge (\neg x \vee \neg y \vee \neg z) \wedge (x \vee \neg y \vee z) \wedge (\neg x \vee y \vee \neg z)$



$3SAT \leq_p$ VERTEXCOVER

- **If ϕ satisfiable**: Let α be a satisfying assignment (e.g., (1,0,0)).
  - For each variable gadget: take x if $\alpha_x = 1$ and $\neg x$ if $\alpha_x = 0$.
  - Each clause has ≥ 1 literal covered.
- **Example:**

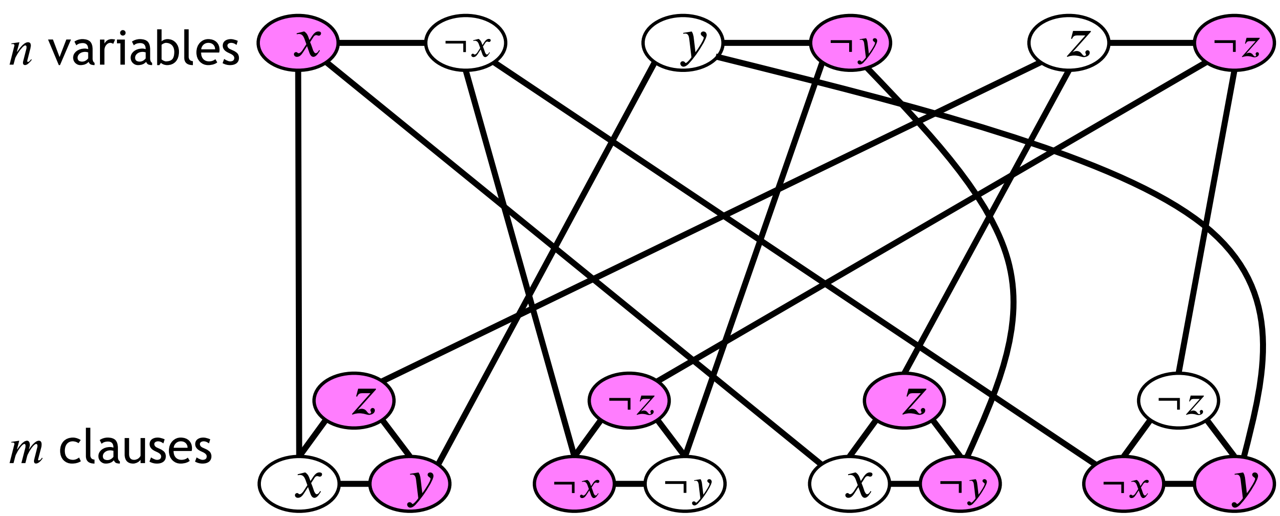
$$(x \vee y \vee z) \wedge (\neg x \vee \neg y \vee \neg z) \wedge (x \vee \neg y \vee z) \wedge (\neg x \vee y \vee \neg z)$$



3SAT $\leq_p$ VERTEXCOVER

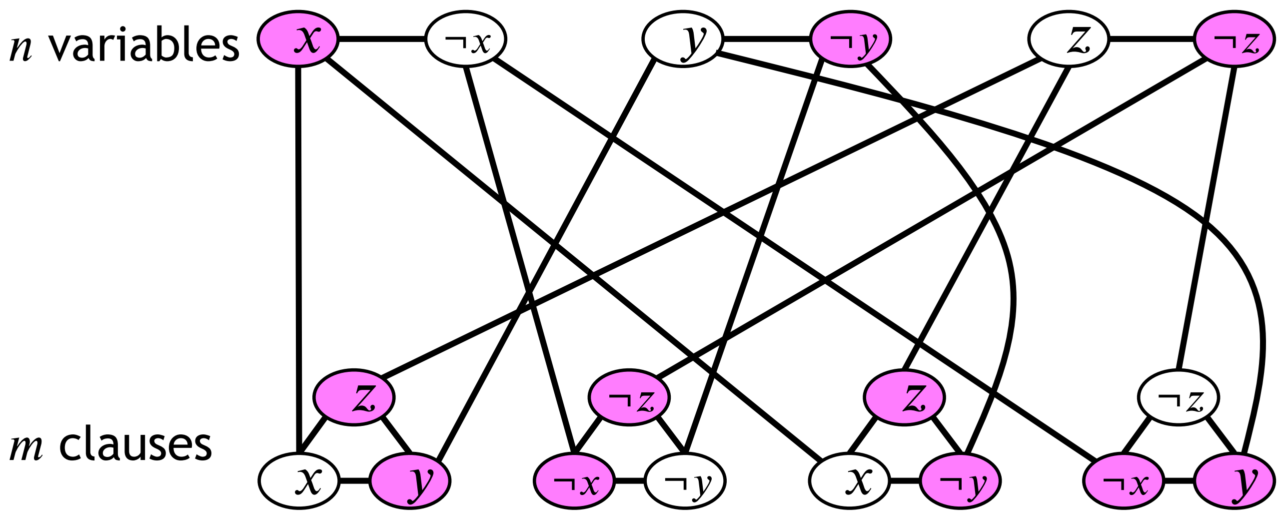
- **If ϕ satisfiable**: Let α be a satisfying assignment.
  - For each variable gadget: take x if $\alpha_x = 1$ and $\neg x$ if $\alpha_x = 0$.
  - Each clause has ≥ 1 literal covered, so take ≤ 2 more.
- **Example:**

$(x \vee y \vee z) \wedge (\neg x \vee \neg y \vee \neg z) \wedge (x \vee \neg y \vee z) \wedge (\neg x \vee y \vee \neg z)$



$3SAT \leq_p$ VERTEXCOVER

* **Conclusion/Claim 1:**
  $\phi \in 3SAT \Longrightarrow (G, n+2m) \in \text{VERTEXCOVER}$



3SAT $\leq_p$ VERTEXCOVER

- **Claim 2:** $\phi \in \text{3SAT} \Longleftarrow (G, n+2m) \in \text{VERTEXCOVER}$
  - A vertex cover of size $n + 2m$ *must* include the following:
    - Exactly 1 vertex from each variable gadget, to cover its edge.
    - Exactly 2 vertices from each clause gadget, to cover its 3 edges.
  - The 2 vertices from a clause gadget cover only 2 of the crossing edges.
  - The 3rd crossing edge must be covered by the variable gadget's vertex.
  - Setting the literals from the selected vertices of the *variable* gadgets to "true" satisfies every clause, so the whole formula is satisfied.