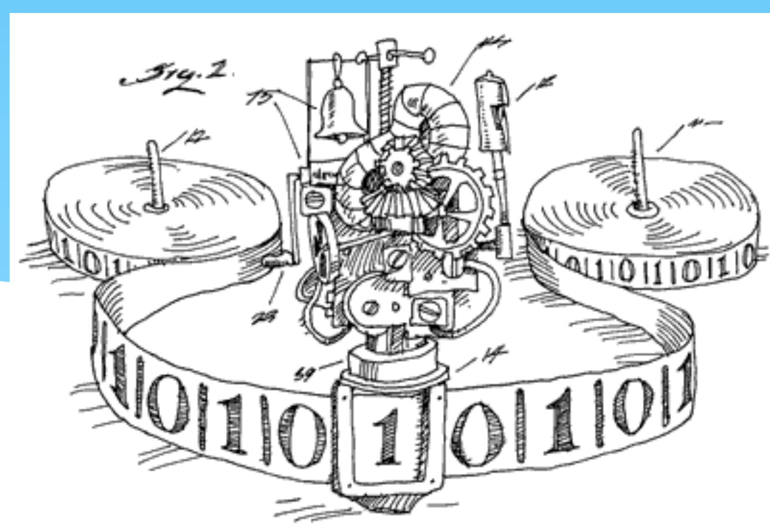
EECS 376: Foundations of Computer Science

Seth Pettie

Lecture 16



Today's Agenda

* Recap: Cook-Levin Theorem and Satisfiability
* 3-CNF Satisfiability
* Clique
* Vertex-Cover

Recap

* **P** is the class of efficiently decidable languages
* **NP** is the class of efficiently verifiable languages

* $L \in$ **NP** if there exists a polynomial time in $|x|$ algorithm $V(x, c)$ such that:
  * $x \in L \Longrightarrow V(x, c)$ accepts for at least one c
  * $x \notin L \Longrightarrow V(x, c)$ rejects for every c

Intuitively: If I somehow "knew" a valid c, I could convince anyone (who can do poly-time computations) that x ∈ L.

Recap

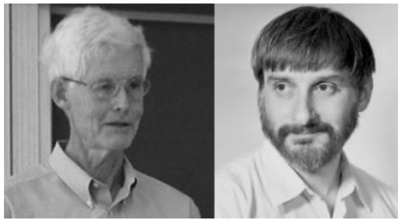
- ***Satisfiability problem*** (SAT): Given a Boolean formula, is there some satisfying assignment (that makes it true)
  - $L_{\mathrm{SAT}} = \{ \Phi \mid \Phi \text{ is satisfiable} \}$ (also just denoted SAT)

- **Example 1:** $\phi(x,y) = \neg x \wedge y$
- **Example 2:** $\phi(x,y,z) = (\neg x \vee y) \wedge (\neg x \vee z) \wedge (y \vee z) \wedge (x \vee \neg z)$

- **Informal Definition:** L is called **NP-Hard** if L ∈ **P** implies that **NP** = **P**.
- **In other words:** A poly-time algorithm for L can be converted to yield poly-time algorithms for all efficiently verifiable languages! That is, for every language in **NP**.

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.
(Also: if SAT is not in P, then P≠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 ≠ NP).

Cook-Levin Outline

Theorem [Cook-Levin]: If SAT ∈ **P**, then **NP** ⊆ **P**.

Proof outline:

* Let D_{SAT} be an efficient decider for SAT.
* Let L ∈ **NP**, so L has an efficient verifier V.
* **Goal:** Build an efficient decider D_L that uses D_{SAT} and V.
* $D_L(x)$:
  * Efficiently construct a poly-size Boolean formula $\phi_{V,x}$ so that:
    * $x \in L \iff$ $\phi_{V,x}$ is satisfiable.
  * Output $D_{\text{SAT}}(\phi_{V,x})$.

3-CNF Satisfiability Problem

* SAT formulas can be complicated.
* **Question:** Can we make them all simpler?

* **Definition:** A ***clause*** is a disjunction (OR) of 3 *literals* $(x \vee \neg y \vee z)$
  Definition: A ***3-CNF formula*** (CNF = conjunctive normal form) is a conjunction (AND) of clauses E.g., $(x \vee \neg y \vee z) \wedge (\neg x \vee z \vee w) \wedge \ldots$

* **Can show:** $\phi_{V,x}$ can be computed using a ***3-CNF formula*** (proof in the notes)
* **Conclusion:** 3SAT = $\{\phi \mid \phi$ is a satisfiable 3-CNF formula$\}$ is **NP-Hard.**

Q: To prove some L is NP-Hard, must we redo Cook-Levin?

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 (and 3SAT) 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***.

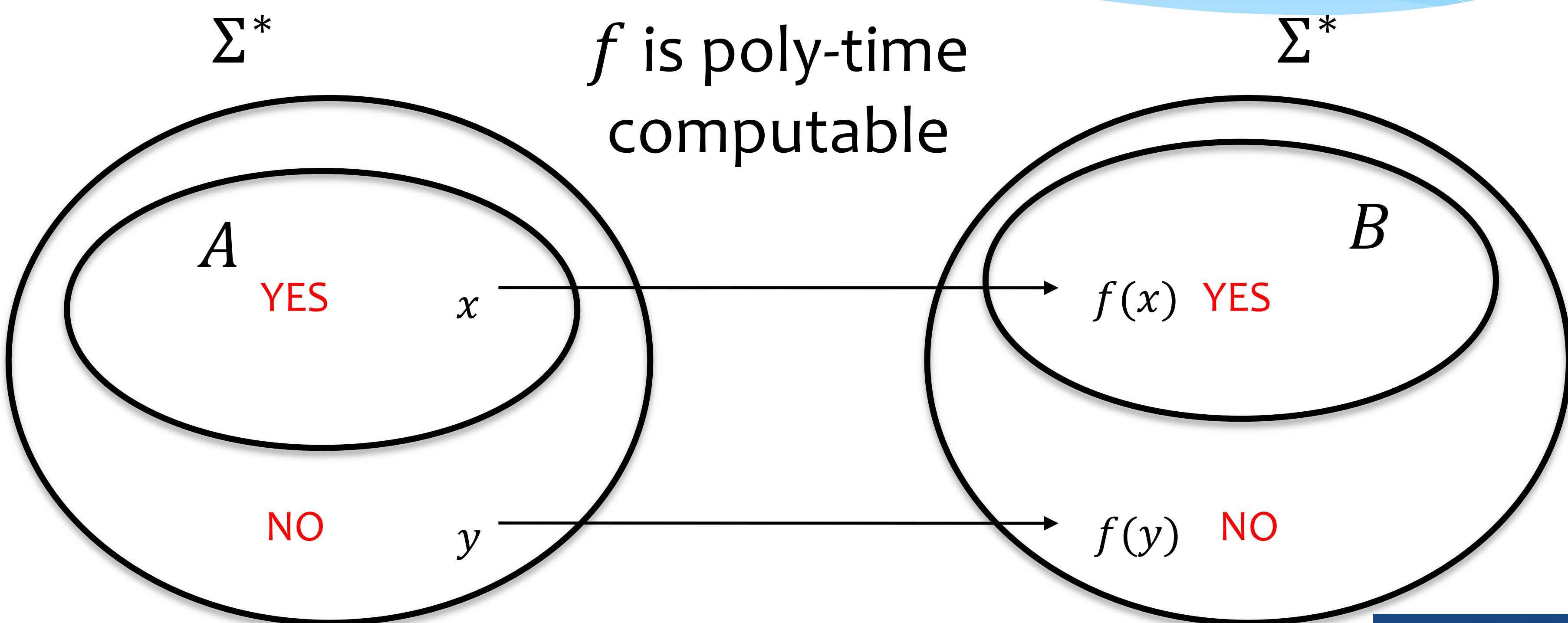
Poly-Time Mapping Reductions

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

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

* **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:** Construct efficient decider for A: given x,
  1. Compute $f(x)$
  2. Run B-decider on $f(x)$.

$A \leq_p B$



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

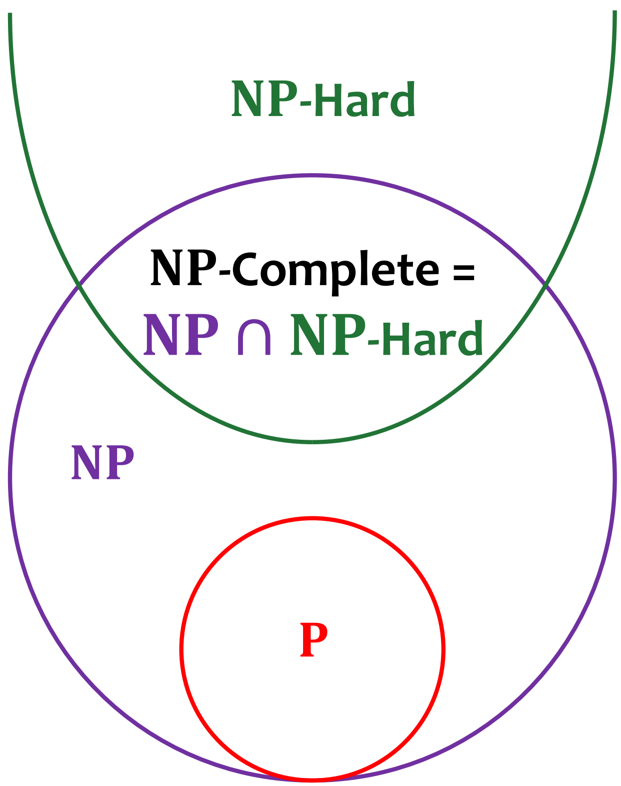
NP-Completeness

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

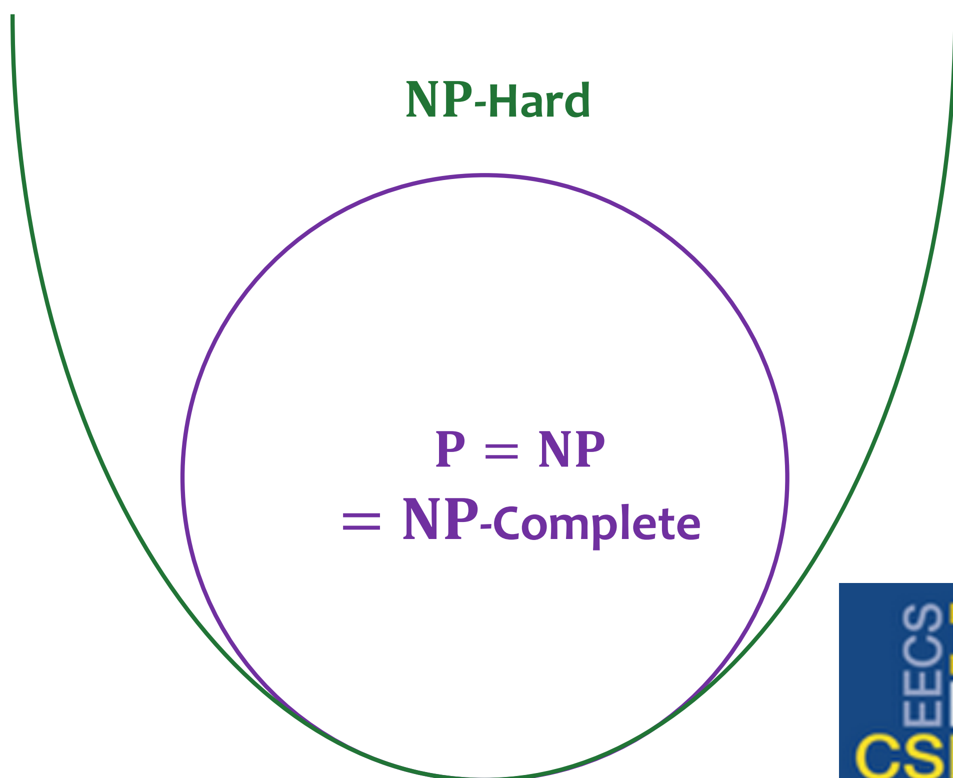
- **We saw:**
  - SAT $\in$ **NP**
  - SAT is **NP**-Hard
  - Thus, SAT is **NP**-Complete.

NP-Hard and -Complete

$\mathbf{P} \subset \mathbf{NP}$



$\mathbf{P} = \mathbf{NP}$



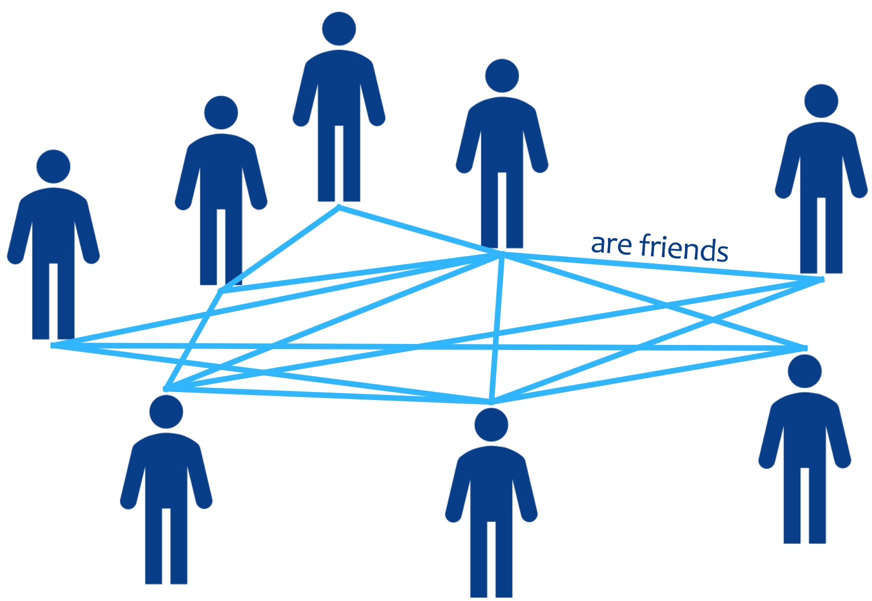
Showing **NP**-Completeness

- To show that a language B is **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* **NP**-Complete A.
     - Write a procedure f mapping instances of A to instances of B, show that it is efficient and correct:
       - $x \in A \Leftrightarrow f(x) \in B$ (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

Given a group of people, are there k people that are <u>*mutual*</u> friends?

* **Formally:** Given a graph G and integer $k \geq 0$, is there a ***clique*** of size k in G?
  * $\mathrm{CLIQUE} = \{\langle G, k\rangle \mid G$ is a graph with a clique of size $k\}$



Clique seems hard

* **Verifying is easy:** We know that CLIQUE $\in$ **NP** (see last lecture)
* **Deciding seems hard:** The naive algorithm for the clique problem (try all subsets of size k) runs in $\Omega\binom{n}{k}$ time, where $n = \#$ vertices
  * If $k = \Theta(n)$, then runtime is *exponential* $2^{\Theta(n)}$ in size of input

But maybe there is a smarter algorithm out there?

* **Idea:** We'll show that given a (hypothetical) *efficient* program **clique** for CLIQUE, we could build an *efficient* program for 3SAT

* **Formally:** We will show that 3SAT $\leq_p$ CLIQUE
* **Conclusion:** Since 3SAT is NP-Complete and CLIQUE $\in$ **NP**, then CLIQUE must also be NP-Complete

3SAT $\leq_p$ CLIQUE

* Need to "*translate*" a 3SAT formula φ into (G_φ, k_φ) such that:
  * φ is satisfiable $\Rightarrow G_\varphi$ has k_φ-clique (**clique**: "yes")
  * φ is *not* satisfiable $\Rightarrow G_\varphi$ *doesn't* have k_φ-clique (**clique**: "no")

* **Proof outline** for showing $\mathrm{3SAT} \leq_p \mathrm{CLIQUE}$:
  * Define an f that converts a formula ϕ to some (G_φ, k_φ)
  * Show that f is correct: $\phi \in \mathrm{3SAT} \Leftrightarrow (G_\varphi, k_\varphi) \in \mathrm{CLIQUE}$.
  * Show that f is efficient.

Example

Goal: "*translate*" 3CNF formula φ into (G_φ, k_φ) such that:

- φ satisfiable $\Rightarrow G_\varphi$ has k_φ-clique (**clique**: "yes")
- φ *not* satisfiable $\Rightarrow G_\varphi$ *doesn't* have k_φ-clique (**clique**: "no")

* $\varphi = (x \vee y \vee z) \wedge (\neg x \vee y \vee \neg z) \wedge (x \vee \neg y \vee z)$
* **Q:** How can we satisfy clause $(x \vee y \vee z)$?
* **Q:** What about clause $(\neg x \vee y \vee \neg z)$?
* **Observation:** We can satisfy a clause by picking any literal (e.g., x) and making it true.
  * We *can't* use the underlying variable (e.g., x) to satisfy any clause in which it appears differently (e.g., $\neg x$)
* Idea: ...Maybe k_φ-clique $\equiv$ a satisfying assignment?

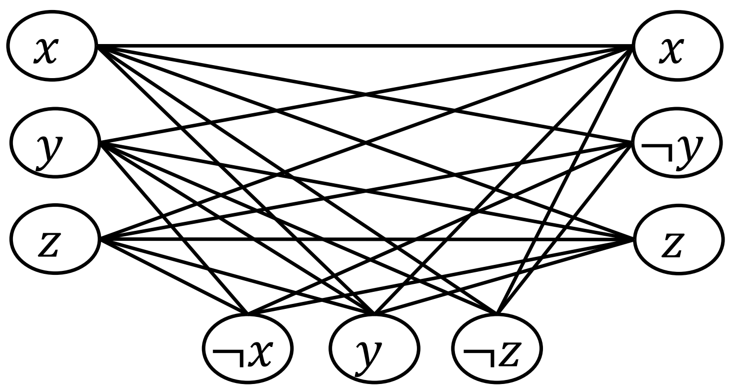
Example

Goal: "*translate*" 3CNF formula φ into (G_φ, k_φ) such that:

- φ satisfiable $\Rightarrow$ G_φ has k_φ-clique (**clique**: "yes")
- φ *not* satisfiable $\Rightarrow$ G_φ *doesn't* have k_φ-clique (**clique**: "no")

* $\varphi = (x \vee y \vee z) \wedge (\neg x \vee y \vee \neg z) \wedge (x \vee \neg y \vee z)$ with 3 clauses
* **Idea for G_φ:** make each literal a vertex; add an edge between two literals in *different clauses* only if they're "compatible"

(*refer to different variables* **or** *they're the same*)



Observations:

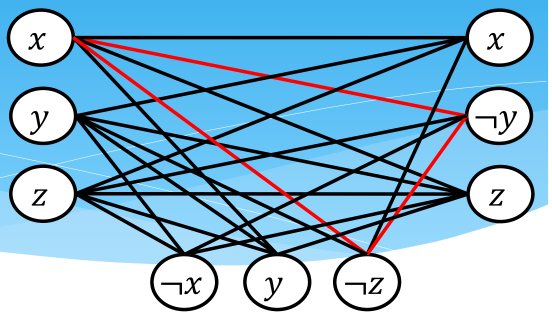
1) Any clique can have at most one vertex from a clause $(k_\varphi \leq 3)$

2) A clique can have several x or $\neg x$, but not both

Will show: This graph has a clique of size 3 if and only if φ is satisfiable

Correctness Analysis (1/2)

Build graph G_φ as follows:
- Make a vertex for each literal of each clause
- Add an edge between two literals in <u>*different*</u> clauses *only if* they are "compatible" (*refer to different variables* **or** *they're the same*)

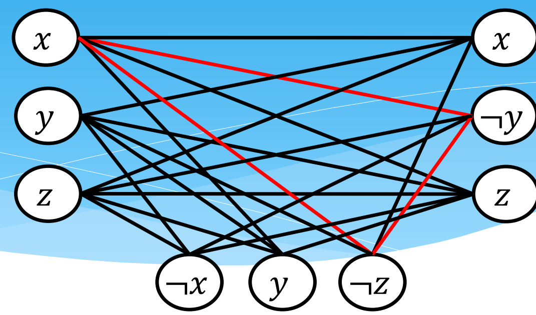


* Suppose that $\varphi = C_1 \wedge C_2 \wedge \cdots \wedge C_m$ has m clauses
* **Claim:** φ is satisfiable $\Rightarrow G_\varphi$ has an m-clique (i.e. use $k_\varphi = m$).
* Consider any satisfying assignment A of φ
* Since φ is satisfied by A, for $1 \le i \le m$, each C_i (e.g., $x \vee y \vee z$) has a literal ℓ_i (pick any if several choices) that evaluates to true under A
* We claim that $\{\ell_1, \ell_2, \dots, \ell_m\}$ is an m-clique in G_φ
  * Consider any two literals ℓ_i and ℓ_j, $i \ne j$
  * If $\ell_i = \ell_j$, then there's an edge between them in G_φ.
  * Otherwise, ℓ_i and ℓ_j must refer to <u>*different variables*</u>! (why?)
  * Hence, they also have an edge between them.

Correctness Analysis (2/2)

Build graph G_φ as follows:

- Make a vertex for each literal of each clause
- Add an edge between two literals in <u>*different*</u> clauses *only if* they are "compatible" (*refer to different variables* **or** *they're the same*)



- Suppose that $\varphi = C_1 \wedge C_2 \wedge \cdots \wedge C_m$ has m clauses

- **Claim:** φ is *not* satisfiable $\Rightarrow$ G_φ *doesn't* have an m-clique.
- *Equivalently*: G_φ has an m-clique $\Rightarrow$ φ is satisfiable

- Suppose that $\{\ell_1, \ell_2, \ldots, \ell_m\}$ is an m-clique in G_φ (one literal per clause)
- Define an *assignment* A of φ by taking each literal ℓ_i and setting the underlying variable so that ℓ_i is true (if any variables are unset at the end, set them arbitrarily)

- By construction of G_φ, since $\{\ell_1, \ell_2, \ldots, \ell_m\}$ is a clique in G_φ, there are <u>*no conflicts*</u> in setting the variables this way
- Since A satisfies each clause of φ, it satisfies φ!

Runtime Analysis

Build graph G_φ as follows:

- Make a vertex for each literal of each clause
- Add an edge between two literals in *different* clauses *only if* they are "compatible" (*refer to different variables* **or** *they're the same*)

* **Claim:** We can build graph G_φ efficiently (poly-time in size of φ)
* Suppose φ has m clauses. $|\langle\varphi\rangle| = O(m)$
* There are $3m$ literals in φ
* The graph G_φ has $3m$ vertices and $O\left((3m)^2\right) = O(m^2)$ edges
  * Takes $O(m^2)$ time to build and $\left|\langle G_\varphi, k_\varphi\rangle\right| = O(m^2)$

* **Conclusion:** Given a hypothetical efficient program **clique**($\langle G_\varphi, k_\varphi\rangle$), we can build an efficient program for 3SAT,

  i.e. $\text{3SAT} \leq_p \text{CLIQUE}$, so CLIQUE is NP-Complete

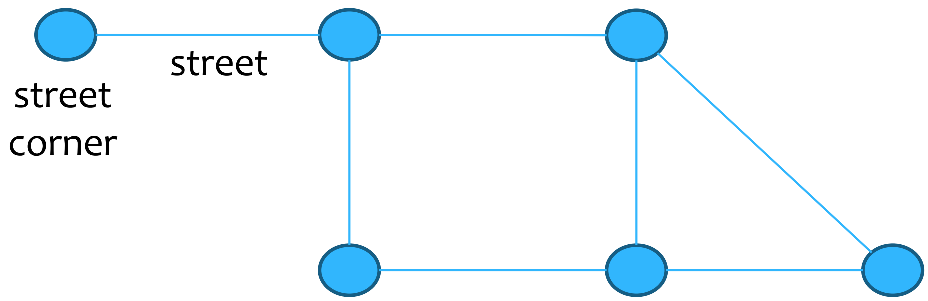
Vertex-Cover Problem

(Starbucks Problem)

* Given a city, is it possible to build stores on k street corners so that *every* street is "covered" by some store?
* (pick vertices to cover at least one end point of each edge)

* **Problem:** Given a graph G and integer $k \geq 0$, is there a ***vertex-cover*** of G of size k?

VERTEX–COVER $= \{ \langle G, k \rangle \mid G$ is a graph w/ a vertex–cover of size $k \}$

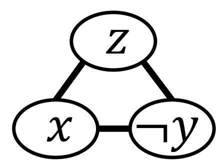


3SAT $\leq_p$ VERTEX–COVER

Goal: "*translate*" φ to (G_φ, k_φ) st:
- φ sat $\Rightarrow$ G_φ has some k_φ-VC
- φ unsat $\Rightarrow$ G_φ has no k_φ-VC

- **Claim:** 3SAT $\leq_p$ VERTEX–COVER
- **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

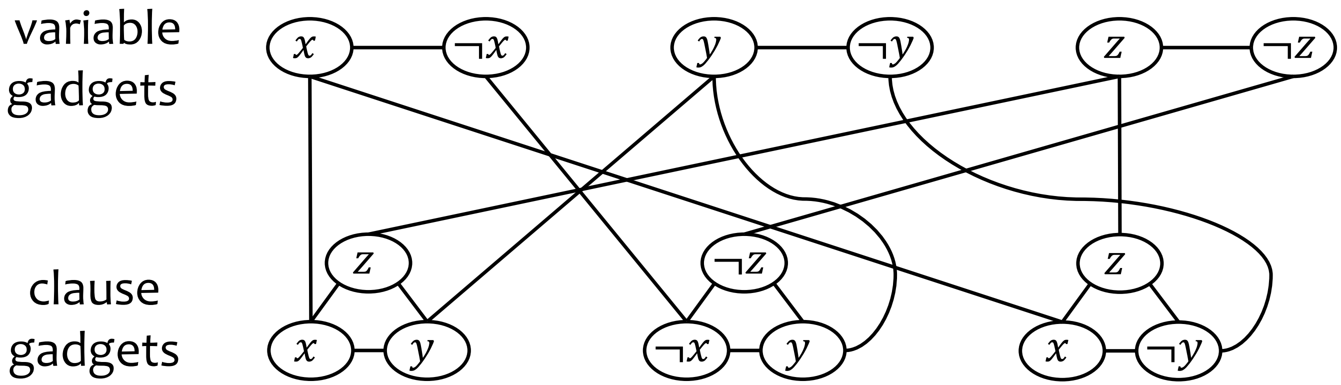


Clause gadget for $(x \vee \neg y \vee z)$

3SAT $\leq_p$ VERTEX–COVER

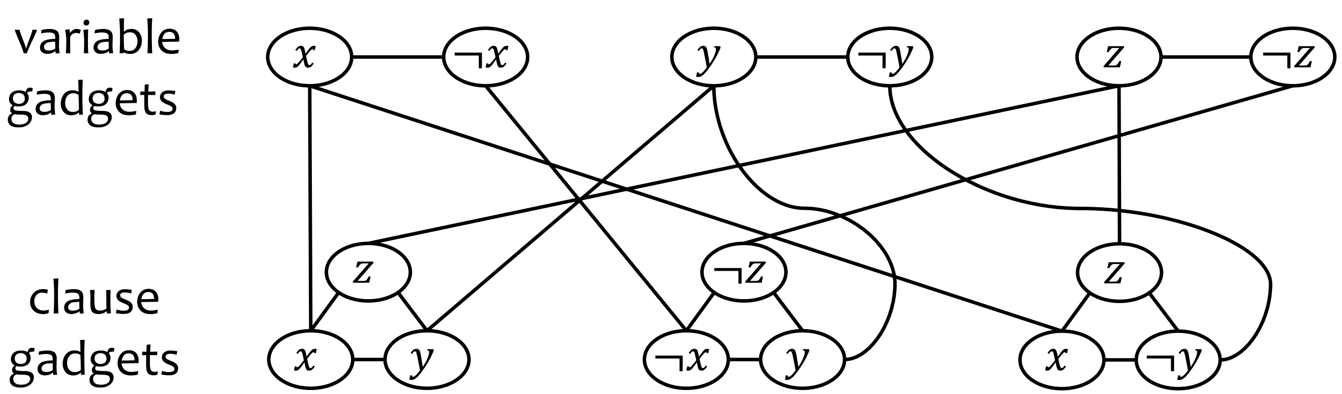
Goal: "*translate*" φ to (G_φ, k_φ) st:
- φ sat $\Rightarrow G_\varphi$ has some k_φ-VC
- φ unsat $\Rightarrow G_\varphi$ has no k_φ-VC

* $\varphi = (x \vee y \vee z) \wedge (\neg x \vee y \vee \neg z) \wedge (x \vee \neg y \vee z)$
* **Construction of G_φ: build** *variable gadgets* and *clause gadgets;* **add** edge $\{u, v\}$ if u is in a *variable gadget* and v is in a *clause gadget and* u and v are *labeled the same*
* **Set k_φ** to $n + 2m$ (n – number of variables, m – number of clauses)



ϕ satisfiable $\Rightarrow (n + 2m)$-VC

* $\varphi = (x \vee y \vee z) \wedge (\neg x \vee y \vee \neg z) \wedge (x \vee \neg y \vee z)$
* $A = (1,0,0)$ is a satisfying assignment
* For every variable gadget: put x in vertex-cover if $A_x = 1$ and $\neg x$ otherwise
* Pick 2 vertices per clause gadget to cover other edges



$(n + 2m)$-VC $\Rightarrow \phi$ satisfiable

* $\varphi = (x \vee y \vee z) \wedge (\neg x \vee y \vee \neg z) \wedge (x \vee \neg y \vee z)$
* **Claim:** In a $(n + 2m)$-VC of G_φ, each variable/clause gadget has exactly one/two vertices in cover.
* For each variable x, set $A_x = 1$ if x is in cover; 0 o/w
* A is a satisfying assignment!