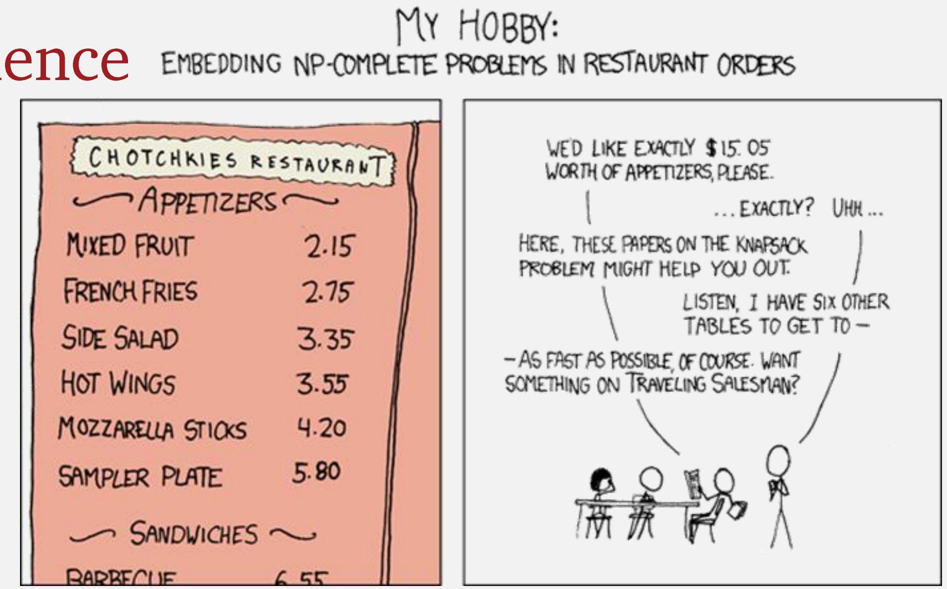
CS 420 / CS 620

NP-Completeness

Monday, December 8, 2025

UMass Boston Computer Science

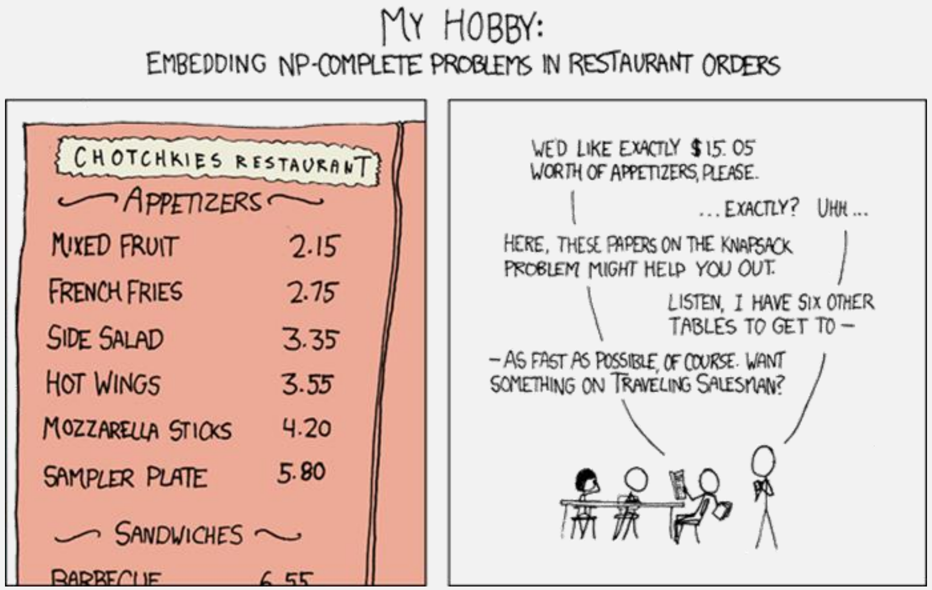


Announcements

- HW 12
  - ~~Out: **Mon 11/24 12pm (noon)**~~
  - ~~Thanksgiving: 11/26-11/30~~
  - ~~Due: **Fri 12/5 12pm (noon)**~~

Last HW

- HW 13
  - Out: **Fri 12/5 12pm (noon)**
  - Due: **Fri 12/12 12pm (noon)** (classes end)
  - Late due: **Mon 12/15 12pm (noon)** (exams start)
    - **Nothing accepted after this** (please don't ask)



In-class question (in Gradescope)

Q1 NP-Completeness
1 Point

Which of following is required for a language L to be NP-complete

(select all that apply)

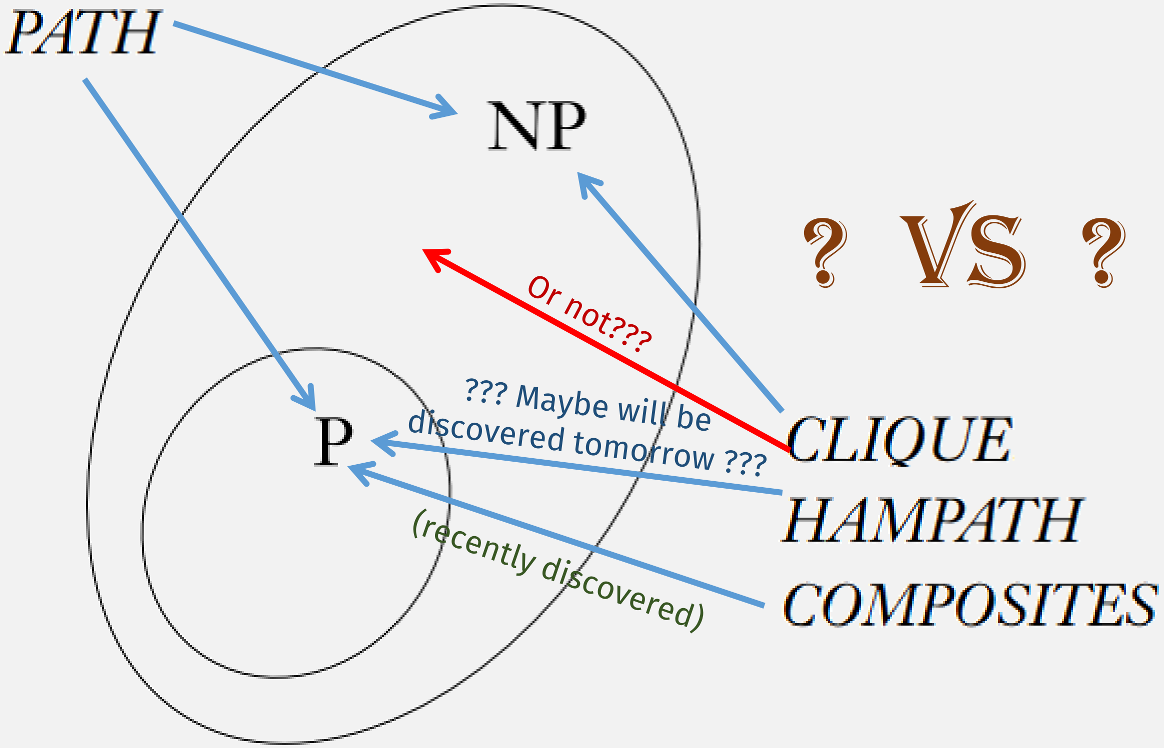
- [ ] $L \in \mathbf{NP}$
- [ ] $L \in \mathbf{P}$
- [ ] for all $A \in \mathbf{NP}, A \leq_{\mathrm{P}} L$
- [ ] for all $A \in \mathbf{NP}, L \leq_{\mathrm{P}} A$

One of the Greatest unsolved

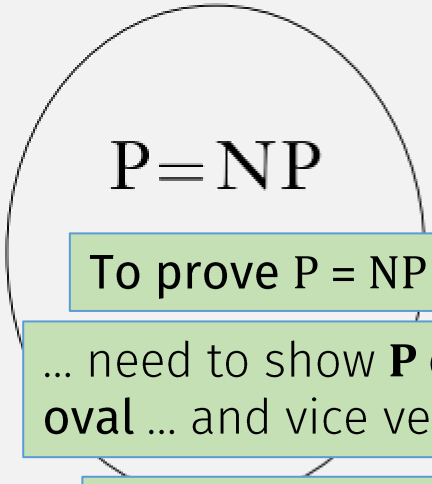
~~Bonus~~ ~~HW~~ Question: Does P = NP?

To prove P ≠ NP … | … need to find a **language in NP** but **not in P!**

(you know how to do it!)



? VS ?



To prove P = NP …

(you also know how to do it!)

… need to show **P oval overlaps with NP oval** … and vice versa**!**

… need need to show every **language in NP** is also **in P**, and vice versa**!**

BUT … **How to prove an algorithm doesn't have poly time algorithm?**
(in general it's hard to prove that something doesn't exist)

Not this course, see Sipser Ch8-9

<u>Last Time</u>: **P** vs **NP**

- **P** = **class of languages** that can be **decided "quickly"**
  - i.e., **"solvable"** with a <u>**deterministic**</u> TM

- **NP** = **class of languages** that can be **verified "quickly"**
  - or, **"solvable"** with a <u>**nondeterministic**</u> TM

- Does **P** = **NP** ?
  - **Problem first posed by John Nash**

- It's a **difficult problem** because **how do you prove:**
  "we'll never find a poly time algorithm for X"?

Progress on whether **P = NP** ?

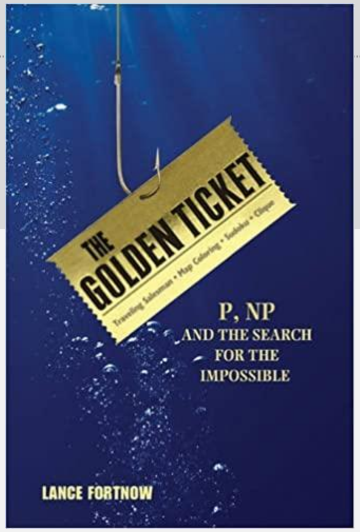
- Some, but still not close

$$P \stackrel{?}{=} NP$$

Scott Aaronson*

The Status of the P Versus NP Problem

By Lance Fortnow
Communications of the ACM, September 2009, Vol. 52 No. 9, Pages 78-86
10.1145/1562164.1562186



- One important concept discovered:
  - **NP**-Completeness

NP-Completeness

Must prove for all langs, not just a single lang

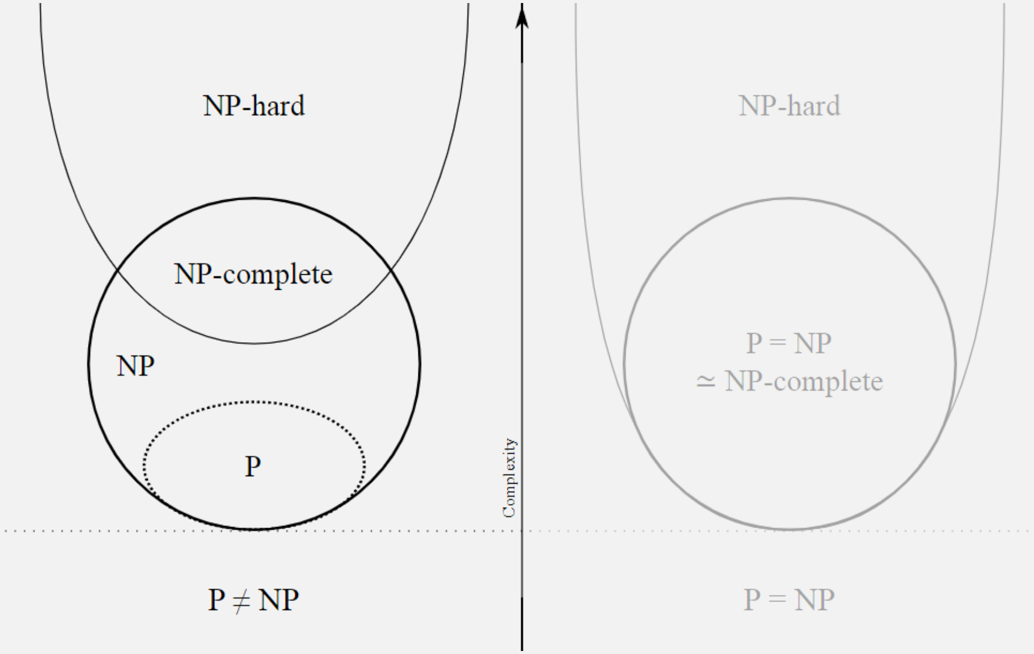
DEFINITION

A language B is ***NP-complete*** if it satisfies two conditions:

1. B is in NP, and easy
2. every A in NP is polynomial time reducible to B. "**NP**-hard"

hard????

What's this?



Flashback: Mapping Reducibility

Language A is ***mapping reducible*** to language B, written $\boxed{A \leq_{\mathrm{m}} B}$, if there is a $\boxed{\text{computable function } f\colon \Sigma^* \longrightarrow \Sigma^*}$, where for every w,

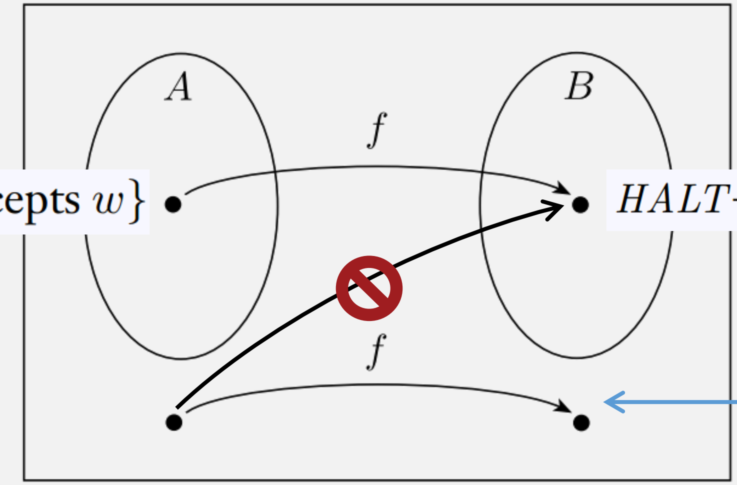
$$\boxed{w \in A \iff f(w) \in B.}$$

IMPORTANT: "if and only if" ...

The function f is called the ***reduction*** from A to B.

To show mapping reducibility:
1. create **computable fn**
2. and then show **forward direction**
3. and **reverse direction**
   (or **contrapositive** of **reverse direction**)

$A_{\mathsf{TM}} = \{\langle M, w\rangle \mid M \text{ is a TM and } M \text{ accepts } w\}$



$HALT_{\mathsf{TM}} = \{\langle M, w\rangle \mid M \text{ is a TM and } M \text{ halts on input } w\}$

... means $\boxed{\overline{A} \leq_{\mathrm{m}} \overline{B}}$

A function $f\colon \Sigma^* \longrightarrow \Sigma^*$ is a ***computable function*** if some Turing machine M, on every input w, halts with just $f(w)$ on its tape.

Polynomial Time Mapping Reducibility

To show poly time mapping reducibility:
1. create **computable fn**
2. **show computable fn runs in poly time**
3. then show **forward direction**
4. and show **reverse direction** (or **contrapositive** of **reverse direction**)

Language A is ***mapping reducible*** to language
if there is a computable function $f\colon \Sigma^* \longrightarrow \Sigma^*$,

$$w \in A \Longleftrightarrow f(w) \in B.$$

The function f is called the ***reduction*** from A

Language A is ***polynomial time mapping reducible***, or simply ***polynomial time reducible***, to language B, written $\boxed{A \leq_{\mathrm{P}} B}$, if a $\boxed{\text{polynomial time computable function}}$ $f\colon \Sigma^* \longrightarrow \Sigma^*$ exists, where for every w,

$$\boxed{w \in A \Longleftrightarrow f(w) \in B.}$$

Don't forget: "if and only if" ...

The function f is called the ***polynomial time reduction*** of A to B.

poly time poly time

A function $f\colon \Sigma^* \longrightarrow \Sigma^*$ is a ***computable function*** if some Turing machine M, on every input w, halts with just $f(w)$ on its tape.

Flashback: If $A \leq_m B$ and B is decidable, then A is decidable.

Has a decider

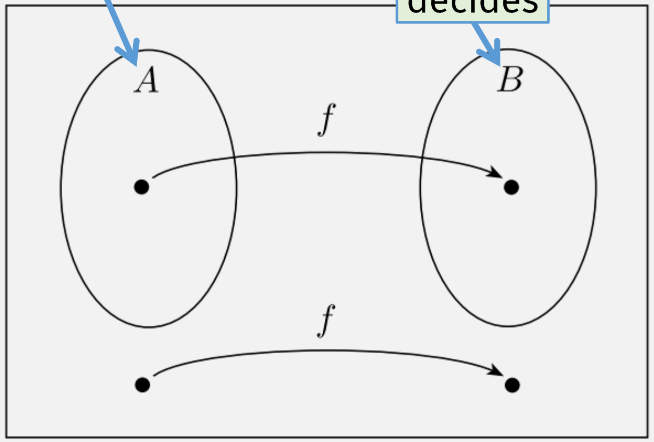
PROOF We let M be the decider for B and f be the reduction from A to B. We describe a decider N for A as follows.

$N =$ "On input w:

1. Compute $f(w)$.
2. Run M on input $f(w)$ and output whatever M outputs."

decides

decides



This proof only works because of the if-and-only-if requirement

Language A is ***mapping reducible*** to language B, written $A \leq_m B$, if there is a computable function $f\colon \Sigma^* \longrightarrow \Sigma^*$, where for every w,

$$\boxed{w \in A \iff f(w) \in B.}$$

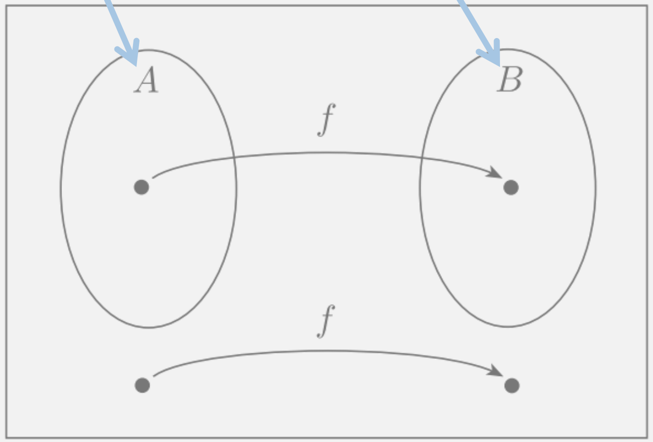
The function f is called the ***reduction*** from A to B.

Thm: If $A \leq_{\cancel{\mathrm{m}}\,\mathbf{P}} B$ and B ~~is decidable,~~ $\in \mathrm{P}$ then A ~~is decidable.~~ $\in \mathrm{P}$

PROOF We let M be the decider for B and f be the reduction from A to B. We describe a decider N for A as follows.

$N =$ "On input w:

1. Compute $f(w)$.
2. Run M on input $f(w)$ and output whatever M outputs."



Language A is ***mapping reducible*** to language B, written $A \leq_{\mathrm{m}} B$, if there is a computable function $f\colon \Sigma^* \longrightarrow \Sigma^*$, where for every w,

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

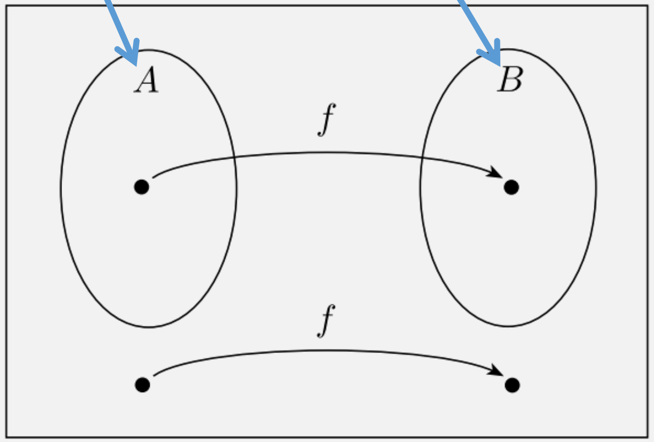
The function f is called the ***reduction*** from A to B.

Thm: If $A \le_{\cancel{\mathrm{m}}\,\mathrm{P}} B$ and B $\in \mathrm{P}$ ~~is decidable,~~ then A $\in \mathrm{P}$ ~~is decidable.~~

PROOF We let M be the poly time decider for B and f be the poly time reduction from A to B. We describe a poly time decider N for A as follows.

$N =$ "On input w:

1. Compute $f(w)$.
2. Run M on input $f(w)$ and output whatever M outputs."



Language A is poly time ***mapping reducible*** to language B, written $A \le_{\mathrm{m}} B$, if there is a computable function $f\colon \Sigma^* \longrightarrow \Sigma^*$, where for every w,

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

The function f is called the ***reduction*** from A to B.

NP-Completeness

DEFINITION

A language B is ***NP-complete*** if it satisfies two conditions:

1. B is in NP, and
2. every A in NP is polynomial time reducible to B.

- How does this help the **P** = **NP** problem?

THEOREM

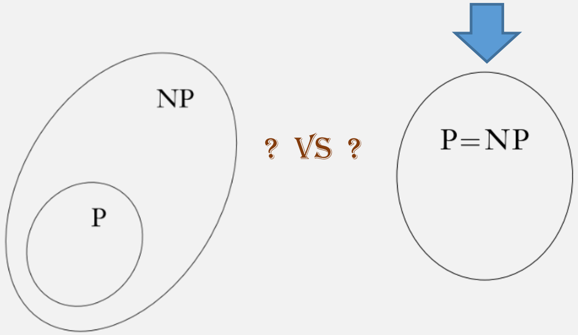
If B is NP-complete and $B \in \mathrm{P}$, then $\mathrm{P} = \mathrm{NP}$.

assume

THEOREM

Proof:

If B is NP-complete and $B \in \mathrm{P}$, then $\mathrm{P} = \mathrm{NP}$.



DEFINITION

A language B is ***NP-complete*** if it satisfies two conditions:

1. B is in NP, and
2. every A in NP is polynomial time reducible to B.

$A \leq_{\mathrm{P}} B$

P

r A → verifier for A that ignores its certificate

2. If **a language** A ∈ **NP**, then A ∈ **P**
   - Given **a language** A ∈ **NP** …
   - … **can poly time mapping reduce** A **to** B --- why?
     - because B **is NP-Complete** (assumption)
   - Then A **also** ∈ **P** …
     - Because if $A \leq_{\mathrm{P}} B$ and $B \in \mathrm{P}$, then $A \in \mathrm{P}$

(prev slide)

So to prove **P** = **NP**, we only need to **find a poly-time algorithm for** <u>one</u> **NP-Complete problem!**

Thus, **if a language** B **is NP-complete** and **in P**, then **P** = **NP**

An **NP**-Complete Language?

$SAT = \{\langle\phi\rangle \mid \phi \text{ is a satisfiable Boolean formula}\}$

DEFINITION

A language B is ***NP-complete*** if it satisfies two conditions:

1. B is in NP, and
2. every A in NP is polynomial time reducible to B.

So to prove **P** = **NP**, we only need to **find a poly-time algorithm for** <u>one</u> **NP-Complete problem!**

Thus, **if a language** B **is NP-complete** and **in P**, **then P = NP**

The Boolean Satisfiability Problem

$SAT = \{\langle\phi\rangle \mid \phi \text{ is a satisfiable Boolean formula}\}$

??

Theorem: SAT **NP**-complete

Boolean Formulas

A Boolean _____	Is ...	Example:
Value	TRUE or FALSE (or 1 or 0)	TRUE, FALSE

Boolean Formulas

A Boolean _____	Is …	Example:
Value	TRUE or FALSE (or 1 or 0)	TRUE, FALSE
Variable	Represents a Boolean **value**	x, y, z

Boolean Formulas

A Boolean _____	Is …	Example:
Value	TRUE or FALSE (or 1 or 0)	TRUE, FALSE
Variable	Represents a Boolean **value**	x, y, z
Operation	Combines Boolean **variables**	AND, OR, NOT ($\wedge$, $\vee$, and $\neg$)

Boolean Formulas

A Boolean _____	Is …	Example:
Value	TRUE or FALSE (or 1 or 0)	TRUE, FALSE
Variable	Represents a Boolean **value**	x, y, z
Operation	Combines Boolean **variables**	AND, OR, NOT ($\wedge$, $\vee$, and $\neg$)
Formula ϕ	Combines **vars** and **operations**	$(\overline{x} \wedge y) \vee (x \wedge \overline{z})$

Boolean Satisfiability

- A **Boolean formula** is **satisfiable** if …

- … there is **some assignment** of TRUE or FALSE $(1$ or $0)$ to its **variables** that **makes the entire formula** TRUE

- Is $(\overline{x} \wedge y) \vee (x \wedge \overline{z})$ satisfiable?
  - Yes
  - $x = \mathrm{FALSE}$,
    $y = \mathrm{TRUE}$,
    $z = \mathrm{FALSE}$

The Boolean Satisfiability Problem

$SAT = \{\langle\phi\rangle \mid \phi \text{ is a satisfiable Boolean formula}\}$

Theorem: SAT is **NP**-complete

DEFINITION

A language B is ***NP-complete*** if it satisfies two conditions:

1. B is in NP, and
2. every A in NP is polynomial time reducible to B.

The Boolean Satisfiability Problem

$SAT = \{\langle\phi\rangle \mid \phi \text{ is a satisfiable Boolean formula}\}$

Theorem: SAT is in **NP**:

- Let $\boldsymbol{n}$ = the number of variables in the formula

Verifier:

On input <ϕ, c>, where c is a possible assignment of variables in ϕ to values:

- Plug values from c into ϕ, **Accept** if result is TRUE

Running Time: $O(\boldsymbol{n})$

Non-deterministic Decider:

On input <ϕ>, where ϕ is a boolean formula:

- Non-deterministically try all possible assignments in parallel
- **Accept** if any satisfy ϕ

Running Time: Checking each assignment takes time $O(\boldsymbol{n})$

The Boolean Satisfiability Problem

$SAT = \{\langle\phi\rangle \mid \phi \text{ is a satisfiable Boolean formula}\}$

Theorem: *SAT* **NP**-complete

DEFINITION

A language B is ***NP-complete*** if it satisfies two conditions:

☑ **1.** B is in NP, and

?? → **2.** every A in NP is polynomial time reducible to B.

Proving the first! **NP**-Completeness problem is hard!

(Just like figuring out the **first undecidable problem** was hard!)

But **after we find one,** then **we can use that problem to prove other problems NP-Complete!**

THEOREM

If B is NP-complete and $B \leq_P C$ for C in NP, then C is NP-complete.

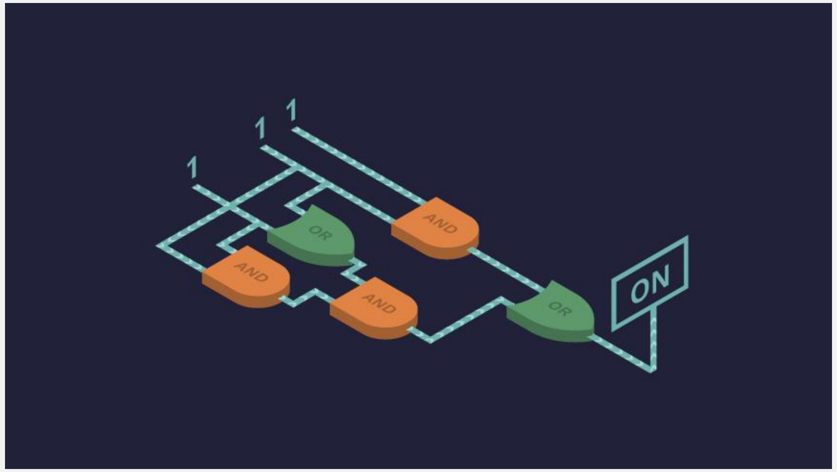
The Boolean Satisfiability Problem

$SAT = \{\langle\phi\rangle \mid \phi \text{ is a satisfiable Boolean formula}\}$

Theorem: *SAT* **NP**-complete

The first **NP**-Complete problem

It sort of makes sense that every problem can be reduced to it …



PROOF: The Cook-Levin Theorem

Will prove on Wed! (today: assume it's true)

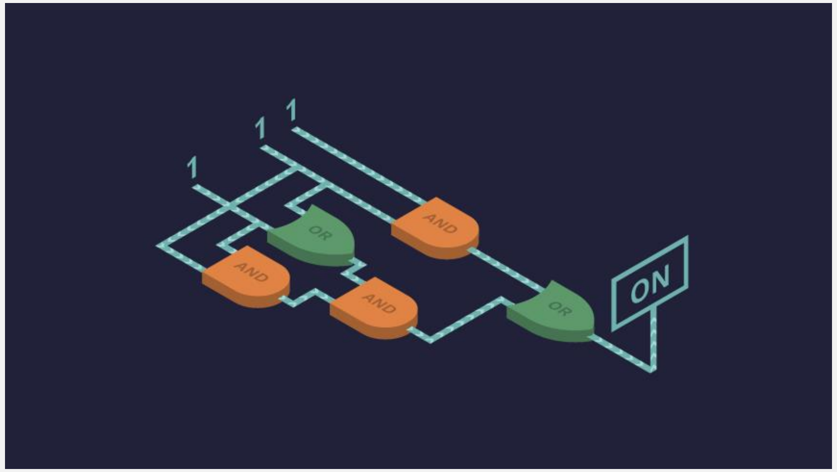
The Boolean Satisfiability Problem

$SAT = \{\langle\phi\rangle \mid \phi \text{ is a satisfiable Boolean formula}\}$

Theorem: *SAT* **NP**-complete

PROOF: The Cook-Levin Theorem

Will prove on Wed! (today: assume it's true)



Then we can use *SAT* to prove other problems **NP**-Complete!

THEOREM

If B is NP-complete and $B \leq_{\mathrm{P}} C$ for C in NP, then C is NP-complete.

The *3SAT* Problem

$$3SAT = \{\langle\phi\rangle \mid \phi \text{ is a satisfiable 3cnf-formula}\}$$

??

<u>Theorem:</u> 3SAT is **NP**-complete

More Boolean Formulas

A Boolean _____	Is …	Example:
Value	TRUE or FALSE (or 1 or 0)	TRUE, FALSE
Variable	Represents a Boolean value	x, y, z
Operation	Combines Boolean variables	AND, OR, NOT ($\wedge$, $\vee$, and $\neg$)
Formula ϕ	Combines vars and operations	$(\overline{x} \wedge y) \vee (x \wedge \overline{z})$

More Boolean Formulas

A Boolean _____	Is …	Example:
Value	TRUE or FALSE (or 1 or 0)	TRUE, FALSE
Variable	Represents a Boolean value	x, y, z
Operation	Combines Boolean variables	AND, OR, NOT ($\wedge$, $\vee$, and $\neg$)
Formula ϕ	Combines vars and operations	$(\overline{x} \wedge y) \vee (x \wedge \overline{z})$
Literal	A var or a negated var	x or $\overline{x}$.

More Boolean Formulas

A Boolean _____	Is …	Example:
Value	TRUE or FALSE (or 1 or 0)	TRUE, FALSE
Variable	Represents a Boolean value	x, y, z
Operation	Combines Boolean variables	AND, OR, NOT ($\wedge$, $\vee$, and $\neg$)
Formula ϕ	Combines vars and operations	$(\overline{x} \wedge y) \vee (x \wedge \overline{z})$
Literal	A var or a negated var	x or $\overline{x}$.
Clause	**Literals** ORed together	$(x_1 \vee \overline{x_2} \vee \overline{x_3} \vee x_4)$

More Boolean Formulas

A Boolean _____	Is …	Example:
Value	TRUE or FALSE (or 1 or 0)	TRUE, FALSE
Variable	Represents a Boolean value	x, y, z
Operation	Combines Boolean variables	AND, OR, NOT ($\wedge$, $\vee$, and $\neg$)
Formula ϕ	Combines vars and operations	$(\overline{x} \wedge y) \vee (x \wedge \overline{z})$
Literal	A var or a negated var	x or $\overline{x}$.
Clause	**Literals** ORed together	$(x_1 \vee \overline{x_2} \vee \overline{x_3} \vee x_4)$
Conjunctive Normal Form (**CNF**)	**Clauses** ANDed together	$(x_1 \vee \overline{x_2} \vee \overline{x_3} \vee x_4) \wedge (x_3 \vee \overline{x_5} \vee x_6)$

$\wedge$ = AND = "Conjunction"
$\vee$ = OR = "Disjunction"
$\neg$ = NOT = "Negation"

More Boolean Formulas

A Boolean _____	Is …	Example:
Value	TRUE or FALSE (or 1 or 0)	TRUE, FALSE
Variable	Represents a Boolean value	x, y, z
Operation	Combines Boolean variables	AND, OR, NOT ($\land$, $\lor$, and $\neg$)
Formula ϕ	Combines vars and operations	$(\overline{x} \land y) \lor (x \land \overline{z})$
Literal	A var or a negated var	x or $\overline{x}$.
Clause	**Literals** ORed together	$(x_1 \lor \overline{x_2} \lor \overline{x_3} \lor x_4)$
Conjunctive Normal Form (**CNF**)	**Clauses** ANDed together	$(x_1 \lor \overline{x_2} \lor \overline{x_3} \lor x_4) \land (x_3 \lor \overline{x_5} \lor x_6)$
3CNF Formula	Three **literals** in each **clause**	$(x_1 \lor \overline{x_2} \lor \overline{x_3}) \land (x_3 \lor \overline{x_5} \lor x_6) \land (x_3 \lor \overline{x_6} \lor x_4)$

$\land$ = AND = “Conjunction”
$\lor$ = OR = “Disjunction”
$\neg$ = NOT = “Negation”

Key thm:
Let's prove it so
we can use it

THEOREM

known | unknown

If B is NP-complete and $B \leq_P C$ for C in NP, then C is NP-complete.

To use this theorem, C must be in **NP**

DEFINITION

A language B is ***NP-complete*** if it satisfies two conditions:

☑ **1.** B is in NP, and
☑ **2.** every A in NP is polynomial time reducible to B.

Proof:

- Need to show: C is **NP**-complete:
  - it's in **NP** (given), and
  - every lang A in **NP** reduces to C in poly time (must show)
- For every language A in **NP**, reduce $A \rightarrow C$ by:
  - First reduce $A \rightarrow B$ in poly time
    - Can do this because: B is **NP-Complete** (given)
  - Then reduce $B \rightarrow C$ in poly time
    - This is also given
- Total run time: Poly time + poly time = poly time

THEOREM

<u>Using</u>: If $\boxed{B \text{ is NP-complete}}$ and $\boxed{B \leq_P C}$ for $\boxed{C \text{ in NP},}$ then $\boxed{C \text{ is NP-complete.}}$

<u>3 steps</u> to prove a language C is **NP**-complete:

1. Show C is in **NP**
2. Choose B, the **NP**-complete problem to reduce from
3. Show a poly time mapping reduction from B to C

To show poly time <u>mapping reducibility</u>:
1. create **computable fn**,
2. show that it **runs in poly time**,
3. then show **forward direction** of mapping red.,
4. and **reverse direction**
   (or **contrapositive** of **reverse direction**)

THEOREM

Using: If B is NP-complete and $B \leq_P C$ for C in NP, then C is NP-complete.

3 steps to prove a language C is **NP**-complete:

1. Show C is in **NP**
2. Choose B, the **NP**-complete problem to reduce from
3. Show a poly time mapping reduction from B to C

Example:

Let C = *3SAT*, to prove *3SAT* is **NP**-Complete:

1. Show *3SAT* is in **NP**

Flashback: 3*SAT* is in **NP**

$3SAT = \{\langle\phi\rangle \mid \phi \text{ is a satisfiable Boolean formula}\}$

Let $\boldsymbol{n}$ = the number of variables in the formula

Verifier:

On input <ϕ, c>, where c is a possible assignment of variables in ϕ to values:

- Accept if c satisfies ϕ

Running Time: $O(\boldsymbol{n})$

Non-deterministic Decider:

On input <ϕ>, where ϕ is a boolean formula:

- Non-deterministically try all possible assignments in parallel
- Accept if any satisfy ϕ

Running Time: Checking each assignment takes time $O(\boldsymbol{n})$

THEOREM

Using: If $\boxed{B \text{ is NP-complete}}$ and $\boxed{B \leq_P C}$ for C in NP, then C is NP-complete.

3 steps to prove a language is **NP**-complete:

1. Show C is in **NP**
2. Choose B, the **NP**-complete problem to reduce from
3. Show a poly time mapping reduction from B to C

Example:

Let C = *3SAT*, to prove *3SAT* is **NP**-Complete:

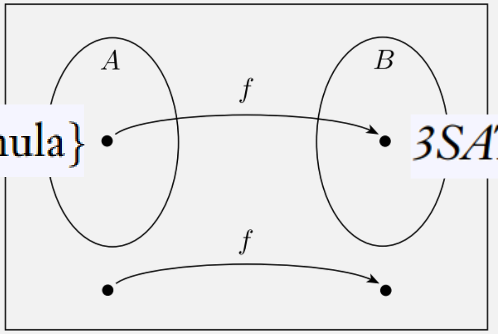
☑ 1. Show *3SAT* is in **NP**

☑ 2. Choose B, the **NP**-complete problem to reduce from: *SAT* (the only possibility, so far)

3. Show a poly time mapping reduction from *SAT* to *3SAT*

<u>Theorem</u>: *SAT* is Poly Time Reducible to *3SAT*

$SAT = \{\langle \phi \rangle \mid \phi \text{ is a satisfiable Boolean formula}\}$



$3SAT = \{\langle \phi \rangle \mid \phi \text{ is a satisfiable 3cnf-formula}\}$

To show poly time <u>mapping reducibility</u>:

1. create **computable fn** f,
2. show that it **runs in poly time**,
3. then show **forward direction** of mapping red.,

   ⇒ if $\phi \in SAT$, then $f(\phi) \in 3SAT$
4. and **reverse direction**

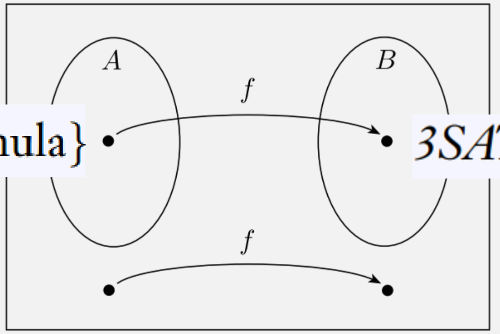
   ⇐ if $f(\phi) \in 3SAT$, then $\phi \in SAT$

   (or **contrapositive** of **reverse direction**)

   ⇐ (alternative) if $\phi \notin SAT$, then $f(\phi) \notin 3SAT$

Theorem: *SAT* is Poly Time Reducible to *3SAT*

$SAT = \{\langle\phi\rangle \mid \phi \text{ is a satisfiable Boolean formula}\}$



$3SAT = \{\langle\phi\rangle \mid \phi \text{ is a satisfiable 3cnf-formula}\}$

Want: poly time computable fn converting a Boolean formula ϕ to 3CNF:

1. Convert ϕ to CNF (an AND of OR clauses)
   a) Use DeMorgan's Law to push negations onto literals

      $\neg(P \vee Q) \boxed{\iff} (\neg P) \wedge (\neg Q)$ $\quad$ $\neg(P \wedge Q) \boxed{\iff} (\neg P) \vee (\neg Q)$ $\quad$ $\boxed{O(n)}$

   b) Distribute ORs to get ANDs outside of parens

      $(P \vee (Q \wedge R)) \boxed{\Leftrightarrow} ((P \vee Q) \wedge (P \vee R))$ $\quad$ $\boxed{O(n)}$

2. Convert to 3CNF by adding new variables

   $(a_1 \vee a_2 \vee a_3 \vee a_4) \boxed{\Leftrightarrow} (a_1 \vee a_2 \vee z) \wedge (\overline{z} \vee a_3 \vee a_4)$ $\quad$ $\boxed{O(n)}$

Remaining step: show iff relation holds …

… this thm is a special case, don't need to separate forward/reverse dir bc each step is already a known "law"

THEOREM

Using: If B is NP-complete and $B \leq_P C$ for C in NP, then C is NP-complete.

3 steps to prove a language is **NP**-complete:

1. Show C is in **NP**
2. Choose B, the **NP**-complete problem to reduce from
3. Show a poly time mapping reduction from B to C

Example:

Let $C = 3SAT$, to prove $3SAT$ is **NP**-Complete:

☑ 1. Show $3SAT$ is in **NP**

☑ 2. Choose B, the **NP**-complete problem to reduce from: SAT

☑ 3. Show a poly time mapping reduction from SAT to $3SAT$

Each **NP**-complete problem we prove makes it easier to prove the next one!

NP-Complete problems, so far

- $SAT = \{\langle\phi\rangle |\ \phi \text{ is a satisfiable Boolean formula}\}$ (assumed true, but havent proven yet)

- $3SAT = \{\langle\phi\rangle |\ \phi \text{ is a satisfiable 3cnf-formula}\}$ (reduced *SAT* to *3SAT*)

- $CLIQUE = \{\langle G, k\rangle |\ G \text{ is an undirected graph with a } k\text{-clique}\}$ (reduce *???* to *CLIQUE*)?

Each **NP**-complete problem we prove makes it easier to prove the next one!

THEOREM

Using: If B is NP-complete and $B \leq_P C$ for C in NP, then C is NP-complete.

3 steps to prove a language is **NP**-complete:

1. Show C is in **NP**
2. Choose B, the **NP**-complete problem to reduce from
3. Show a poly time mapping reduction from B to C

Example:

Let C = *3SAT*, to prove *3SAT* is **NP**-Complete:

☑ 1. Show *3SAT* is in **NP**

☑ 2. Choose B, the **NP**-complete problem to reduce from: *SAT*

☑ 3. Show a poly time mapping reduction from *SAT* to *3SAT*

Each **NP**-complete problem we prove makes it easier to prove the next one!

THEOREM

Using: If B is NP-complete and $B \leq_P C$ for C in NP, then C is NP-complete.

3 steps to prove a language is **NP**-complete:

1. Show *C* is in **NP**
2. Choose *B*, the **NP**-complete problem to reduce from
3. Show a poly time mapping reduction from *B* to *C*

Example:

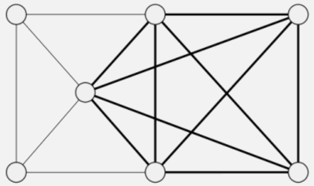
Let *C* = ~~*3SAT*~~ ***CLIQUE***, to prove ~~*3SAT*~~ ***CLIQUE*** is **NP**-Complete:

?1. Show ~~*3SAT*~~ ***CLIQUE*** is in **NP**

?2. Choose *B*, the **NP**-complete problem to reduce from: ~~*SAT*~~ ***3SAT***

?3. Show a poly time mapping reduction from ***3SAT*** to ~~*3SAT*~~ ***CLIQUE***

Flashback: CLIQUE is in NP



$CLIQUE = \{\langle G, k\rangle \mid G \text{ is an undirected graph with a } k\text{-clique}\}$

PROOF IDEA The clique is the certificate.

Let n = # nodes in G

c is at most n

PROOF The following is a $\boxed{\text{verifier } V}$ for $CLIQUE$.

V = "On input $\langle\langle G, k\rangle, c\rangle$:

1. Test whether c is a subgraph with k nodes in G.
2. Test whether G contains all edges connecting nodes in c.
3. If both pass, *accept*; otherwise, *reject*."

For each node in c, check whether it's in G: $O(\boldsymbol{n})$

For each pair of nodes in c, check whether there's an edge in G: $O(\boldsymbol{n}^2)$

THEOREM

<u>Using</u>: If B is NP-complete and $B \leq_P C$ for C in NP, then C is NP-complete.

<u>3 steps</u> to prove a language is **NP**-complete:

1. Show *C* is in **NP**
2. Choose *B*, the **NP**-complete problem to reduce from
3. Show a poly time mapping reduction from *B* to *C*

<u>Example</u>:

Let *C* = ~~*3SAT*~~ ***CLIQUE***, to prove ~~*3SAT*~~ ***CLIQUE*** is **NP**-Complete:

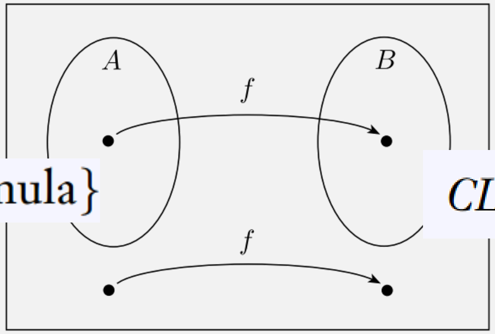
☑1. Show ~~*3SAT*~~ ***CLIQUE*** is in **NP**

☑2. Choose *B*, the **NP**-complete problem to reduce from: ~~*SAT*~~ ***3SAT***

?3. Show a poly time mapping reduction from ***3SAT*** to ~~*3SAT*~~ ***CLIQUE***

Theorem: *3SAT* is polynomial time reducible to *CLIQUE*.

$3SAT = \{\langle\phi\rangle \mid \phi \text{ is a satisfiable 3cnf-formula}\}$

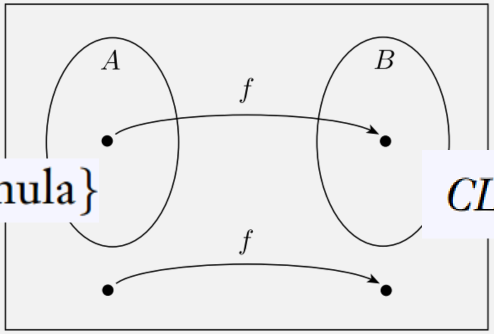


$CLIQUE = \{\langle G, k\rangle \mid G \text{ is an undirected graph with a } k\text{-clique}\}$

To show poly time mapping reducibility:

1. create **computable fn**,
2. show that it **runs in poly time**,
3. then show **forward direction** of mapping red.,
4. and **reverse direction**
   (or **contrapositive** of **reverse direction**)

<u>Theorem</u>: *3SAT* is polynomial time reducible to *CLIQUE*.



$3SAT = \{\langle\phi\rangle |\ \phi \text{ is a satisfiable 3cnf-formula}\}$

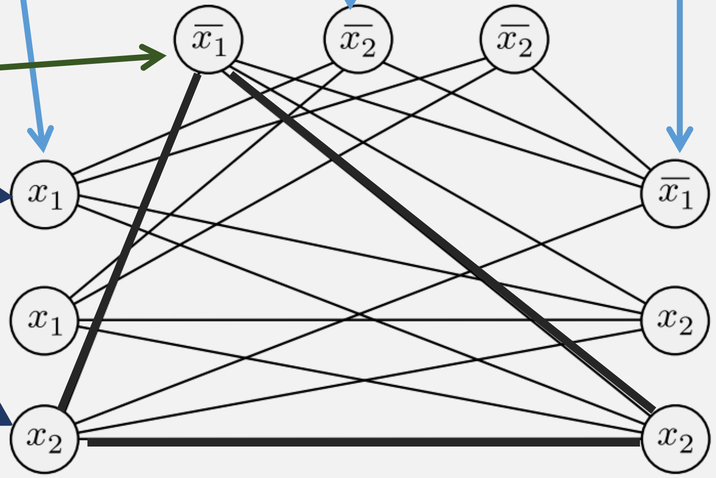
$CLIQUE = \{\langle G, k\rangle |\ G \text{ is an undirected graph with a } k\text{-clique}\}$

<u>Need</u>: poly time <u>computable fn</u> converting a 3cnf-formula …

Example:

$$\phi = (x_1 \vee x_1 \vee \boxed{x_2}) \wedge (\boxed{\overline{x_1}} \vee \overline{x_2} \vee \overline{x_2}) \wedge (\overline{x_1} \vee x_2 \vee \boxed{x_2})$$

- … to a graph containing a clique:
  - Each clause maps to a group of 3 nodes
  - Connect all nodes <u>except</u>:
    - Contradictory nodes
    - Nodes in the same group



Don't forget iff

Runs in **poly time:**

- \# literals = # nodes $O(\boldsymbol{n})$
- \# edges poly in # nodes $O(\boldsymbol{n^2})$

$\Rightarrow$ If $\phi \in 3SAT$

- Then each clause has a TRUE literal
  - Those are <u>nodes in the 3-clique</u>!
  - E.g., $x_1 = 0, x_2 = 1$

$\Leftarrow$ If $\phi \notin 3SAT$

- Then for any assignment, some clause must have a contradiction with another clause
- Then in the graph, some clause's group of nodes won't be connected to another group, <u>preventing the clique</u>

THEOREM

Using: If B is NP-complete and $B \leq_P C$ for C in NP, then C is NP-complete.

3 steps to prove a language is **NP**-complete:

1. Show C is in **NP**
2. Choose B, the **NP**-complete problem to reduce from
3. Show a poly time mapping reduction from B to C

Example:

Let C = ~~*3SAT*~~ ***CLIQUE***, to prove ~~*3SAT*~~ ***CLIQUE*** is **NP**-Complete:

☑ 1. Show ~~*3SAT*~~ ***CLIQUE*** is in **NP**

☑ 2. Choose B, the **NP**-complete problem to reduce from: ~~*SAT*~~ ***3SAT***

☑ 3. Show a poly time mapping reduction from ***3SAT*** to ~~*3SAT*~~ ***CLIQUE***

NP-Complete problems, so far

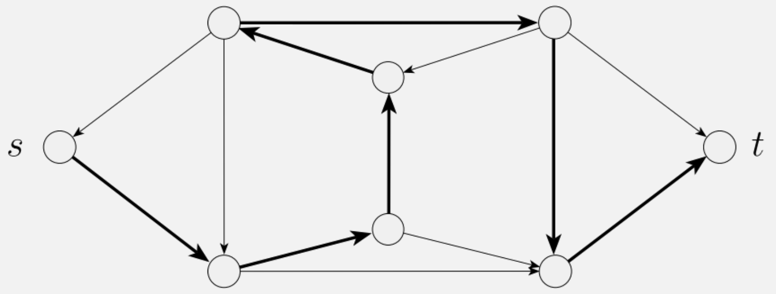
- $SAT = \{\langle \phi \rangle \mid \phi \text{ is a satisfiable Boolean formula}\}$ (havent proven yet)
- $3SAT = \{\langle \phi \rangle \mid \phi \text{ is a satisfiable 3cnf-formula}\}$ (reduced *SAT* to *3SAT*)
- $CLIQUE = \{\langle G, k \rangle \mid G \text{ is an undirected graph with a } k\text{-clique}\}$ (reduced *3SAT* to *CLIQUE*)

Each **NP**-complete problem we prove makes it easier to prove the next one!

Flashback: The *HAMPATH* Problem

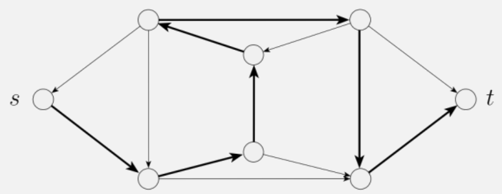
$$HAMPATH = \{\langle G, s, t\rangle \mid G \text{ is a directed graph with a Hamiltonian path from } s \text{ to } t\}$$

- A Hamiltonian path goes through <u>every</u> node in the graph



- The **Search** problem:
  - Exponential time (brute force) algorithm:
    - Check all possible paths of length $\boldsymbol{n}$
    - See if any connects s and t: $O(\boldsymbol{n!}) = O(2^n)$
  - Polynomial time algorithm:
    - <u>Unknown</u>!!!
- The **Verification** problem:
  - Still $O(\boldsymbol{n}^2)$, just like *PATH*!
- So *HAMPATH* is in **NP** but <u>not known</u> to be in **P**

Theorem: *HAMPATH* is NP-complete



$$HAMPATH = \{\langle G, s, t\rangle \mid G \text{ is a directed graph with a Hamiltonian path from } s \text{ to } t\}$$

THEOREM

Using: If B is NP-complete and $B \leq_{\mathrm{P}} C$ for C in NP, then C is NP-complete.

3 steps to prove a language is **NP**-complete:

1. Show C is in **NP**
2. Choose B, the known **NP**-complete problem to reduce from
3. Show a poly time mapping reduction from B to C

Theorem: *HAMPATH* is NP-complete

$$\mathit{HAMPATH} = \{\langle G, s, t\rangle \mid G \text{ is a directed graph with a Hamiltonian path from } s \text{ to } t\}$$

To prove *HAMPATH* is **NP**-complete:

☑ 1. Show *HAMPATH* is in **NP** (same verifier as *PATH*)

☑ 2. Choose *B*, the **NP**-complete problem to reduce from *3SAT*

3. Show a poly time mapping reduction from *B* to *HAMPATH*

Theorem: *HAMPATH* is NP-complete

$HAMPATH = \{\langle G, s, t\rangle \mid G$ is a directed graph with a Hamiltonian path from s to $t\}$

To prove *HAMPATH* is **NP**-complete:

☑ 1. Show *HAMPATH* is in **NP** (left as exercise)

☑ 2. Choose *B*, the **NP**-complete problem to reduce from *3SAT*

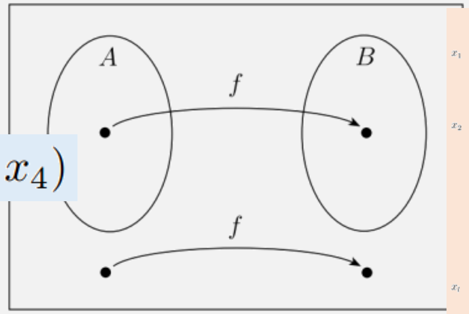
? 3. Show a poly time mapping reduction from *3SAT* to *HAMPATH*

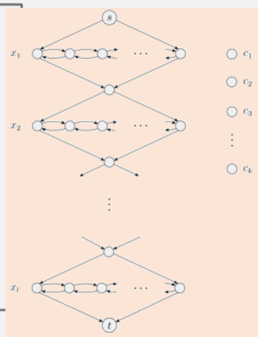
To show poly time mapping reducibility:

1. create **computable fn**,
2. show that it **runs in poly time**,
3. then show **forward direction** of mapping red.,
4. and **reverse direction**
   (or **contrapositive** of **reverse direction**)

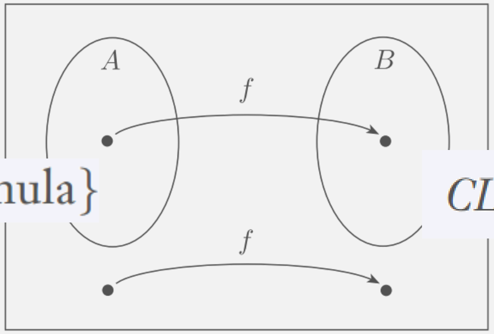
???

$(x_1 \vee \overline{x_2} \vee \overline{x_3}) \wedge (x_3 \vee \overline{x_5} \vee x_6) \wedge (x_3 \vee \overline{x_6} \vee x_4)$





Flashback: *3SAT* is polynomial time reducible to *CLIQUE*.



$3SAT = \{\langle\phi\rangle \mid \phi \text{ is a satisfiable 3cnf-formula}\}$

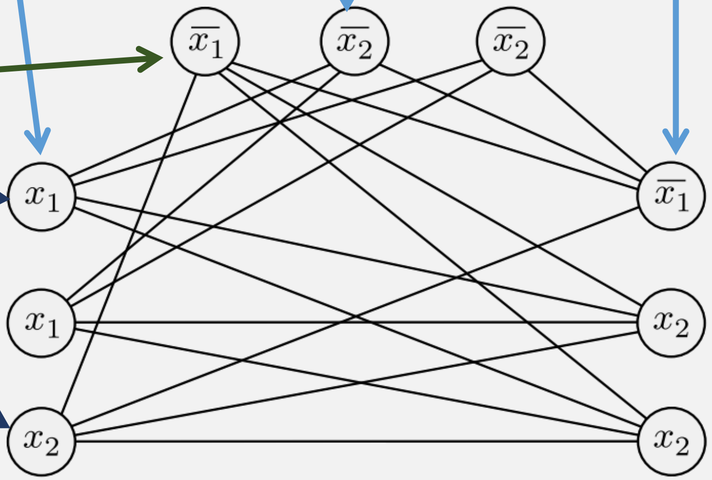
$CLIQUE = \{\langle G, k\rangle \mid G \text{ is an undirected graph with a } k\text{-clique}\}$

Need: poly time computable fn converting a 3cnf-formula …

Example:

$$\phi = (x_1 \vee x_1 \vee x_2) \wedge (\overline{x_1} \vee \overline{x_2} \vee \overline{x_2}) \wedge (\overline{x_1} \vee x_2 \vee x_2)$$

- … to a graph containing a clique:
  - Each **clause** maps to a group of 3 **nodes**
  - **Connect** all **nodes** except:
    - Contradictory nodes
    - Nodes in the same group



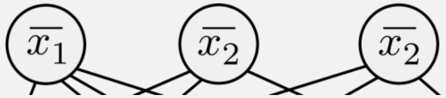
Do conversion piece by piece …

General Strategy: Reducing from *3SAT*

Create a **computable function** mapping formula to “**gadgets**”:

- Variable → “gadget”, e.g.,

- Clause → “gadget”, e.g.,



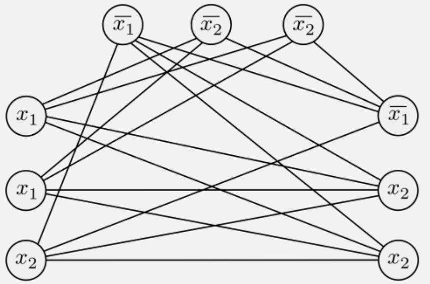
Gadget is typically “**used**” in **two** “**opposite**” **ways**:

1. “**something**” when **var is assigned** TRUE, or
2. “**something else**” when **var is assigned** FALSE

NOTE: “gadgets” are not always graphs; depends on the problem

Then **connect variable** and **clause** “**gadgets**” **together**:

- Literal x_i in clause c_j → gadget x_i “connects to” gadget c_j
- Literal $\overline{x_i}$ in clause c_j → gadget x_i “connects to” gadget c_j
- E.g., **connect each node to node not in clause**



Theorem: *HAMPATH* is NP-complete

$$HAMPATH = \{\langle G, s, t\rangle \mid G \text{ is a directed graph with a Hamiltonian path from } s \text{ to } t\}$$

To prove *HAMPATH* is **NP**-complete:

☑ 1. Show *HAMPATH* is in **NP** (in HW9)

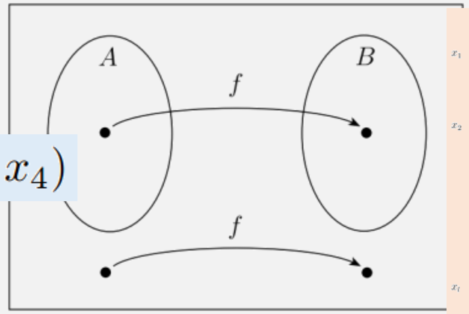
☑ 2. Choose *B*, the **NP**-complete problem to reduce from *3SAT*

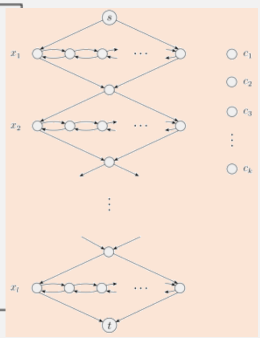
? 3. Show a poly time mapping reduction from *3SAT* to *HAMPATH*

To show poly time mapping reducibility:
1. create **computable fn**,
2. show that it **runs in poly time**,
3. then show **forward direction** of mapping red.,
4. and **reverse direction**
   (or **contrapositive** of **reverse direction**)

???

$$(x_1 \vee \overline{x_2} \vee \overline{x_3}) \wedge (x_3 \vee \overline{x_5} \vee x_6) \wedge (x_3 \vee \overline{x_6} \vee x_4)$$





Computable Fn: Formula (blue) → Graph (orange)

variable clause

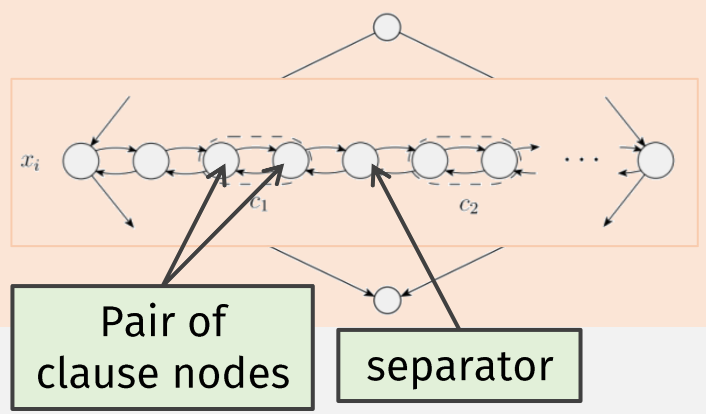
Example input: $\phi = (a_1 \vee b_1 \vee c_1) \wedge (a_2 \vee b_2 \vee c_2) \wedge \cdots \wedge (a_k \vee b_k \vee c_k)$

k = # clauses

- Clause → (extra) single nodes, Total = k

(extra)

- Variable → diamond-shaped graph "gadget"
  - Clause → 2 "connector" nodes + separator
  - Total = $3k+1$ "connector" nodes per "gadget"



Computable Fn: Formula (blue) → Graph (orange)

Example input: $\phi = (a_1 \vee b_1 \vee c_1) \wedge (a_2 \vee b_2 \vee c_2) \wedge \cdots \wedge (a_k \vee b_k \vee c_k)$

k = # clauses

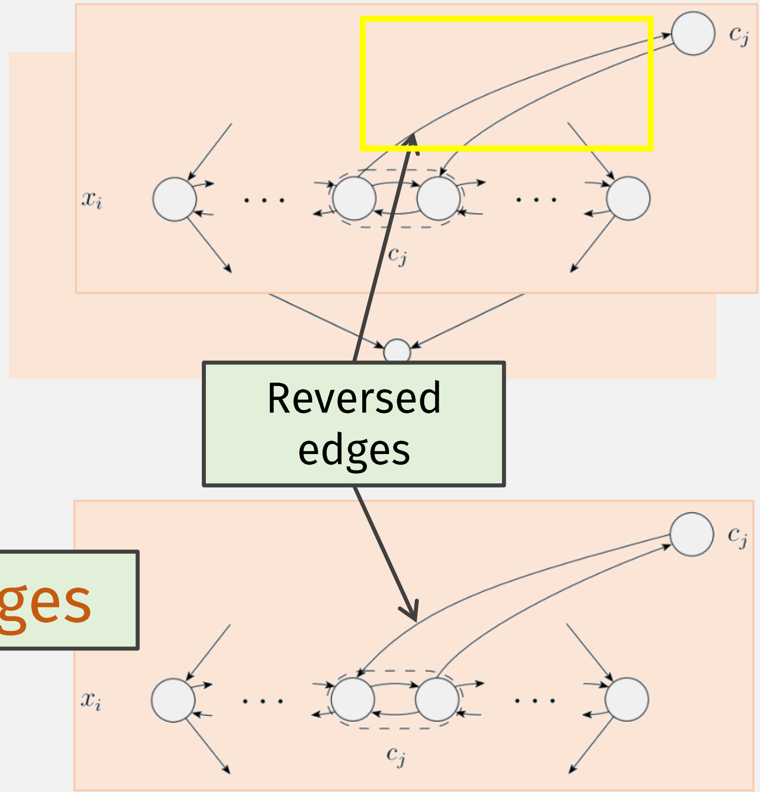
- Clause → (extra) single nodes, Total = k
- Variable → diamond-shaped graph "gadget"
  - Clause → 2 "connector" nodes + separator
  - Total = $3k+1$ "connector" nodes per "gadget"

Literal = variable or negated variable

- Lit x_i in clause c_j → c_j node edges in gadget x_i

Each extra c_j node has 6 edges

- Lit $\overline{x_i}$ in clause c_j → c_j edges in gadget x_i (rev)



<u>Theorem</u>: *HAMPATH* is NP-complete

$HAMPATH = \{\langle G, s, t\rangle \mid G$ is a directed graph with a Hamiltonian path from s to $t\}$

To prove *HAMPATH* is **NP**-complete:

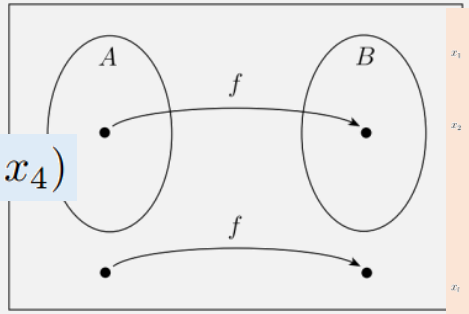
☑ 1. Show *HAMPATH* is in **NP**

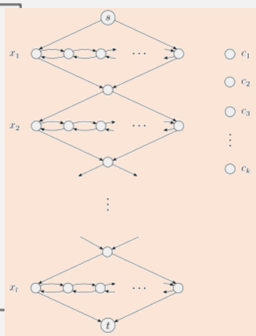
☑ 2. Choose *B*, the **NP**-complete problem to reduce from *3SAT*

? 3. Show a poly time mapping reduction from *3SAT* to *HAMPATH*

To show poly time <u>mapping reducibility</u>:
☑ 1. create **computable fn**,
2. show that it **runs in poly time**,
3. then show **forward direction** of mapping red.,
4. and **reverse direction**
(or **contrapositive** of **reverse direction**)

$$(x_1 \vee \overline{x_2} \vee \overline{x_3}) \wedge (x_3 \vee \overline{x_5} \vee x_6) \wedge (x_3 \vee \overline{x_6} \vee x_4)$$





Polynomial Time?

TOTAL: $O(k^2)$

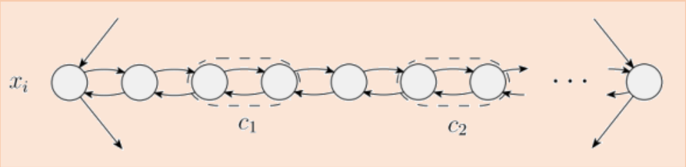
Example input: $\phi = (a_1 \vee b_1 \vee c_1) \wedge (a_2 \vee b_2 \vee c_2) \wedge \cdots \wedge (a_k \vee b_k \vee c_k)$

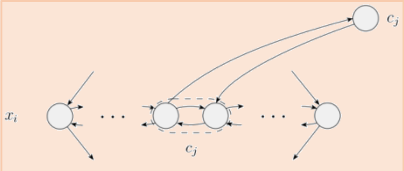
k = # clauses = at most $\mathbf{3}\boldsymbol{k}$ variables

- Clause → (extra) single nodes 

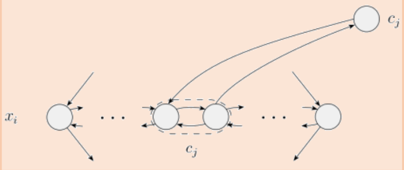
$O(\boldsymbol{k})$

- Variable → diamond-shaped graph "gadget" $O(\boldsymbol{k}^2)$
  - Clause → 2 "connector" nodes + separator
  - Total = $3k$+1 "connector" nodes per "gadget"



- Lit x_i in clause c_j → c_j node edges in gadget x_i 

$O(\boldsymbol{k})$

- Lit $\overline{x_i}$ in clause c_j → c_j edges in gadget x_i (rev) 

$O(\boldsymbol{k})$

Theorem: *HAMPATH* is NP-complete

$HAMPATH = \{\langle G, s, t\rangle \mid G \text{ is a directed graph with a Hamiltonian path from } s \text{ to } t\}$

To prove *HAMPATH* is **NP**-complete:

☑ 1. Show *HAMPATH* is in **NP**

☑ 2. Choose *B*, the **NP**-complete problem to reduce from *3SAT*

? 3. Show a poly time mapping reduction from *3SAT* to *HAMPATH*

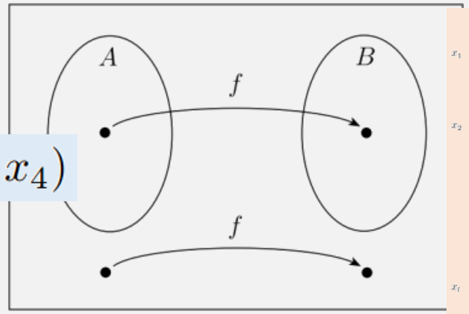
To show poly time mapping reducibility:

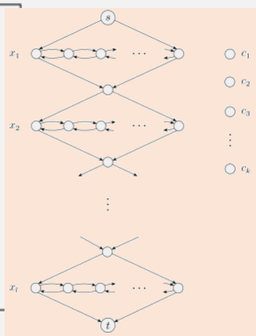
☑ 1. create **computable fn**,

☑ 2. show that it **runs in poly time**,

3. then show **forward direction** of mapping red.,
4. and **reverse direction** (or **contrapositive** of **reverse direction**)

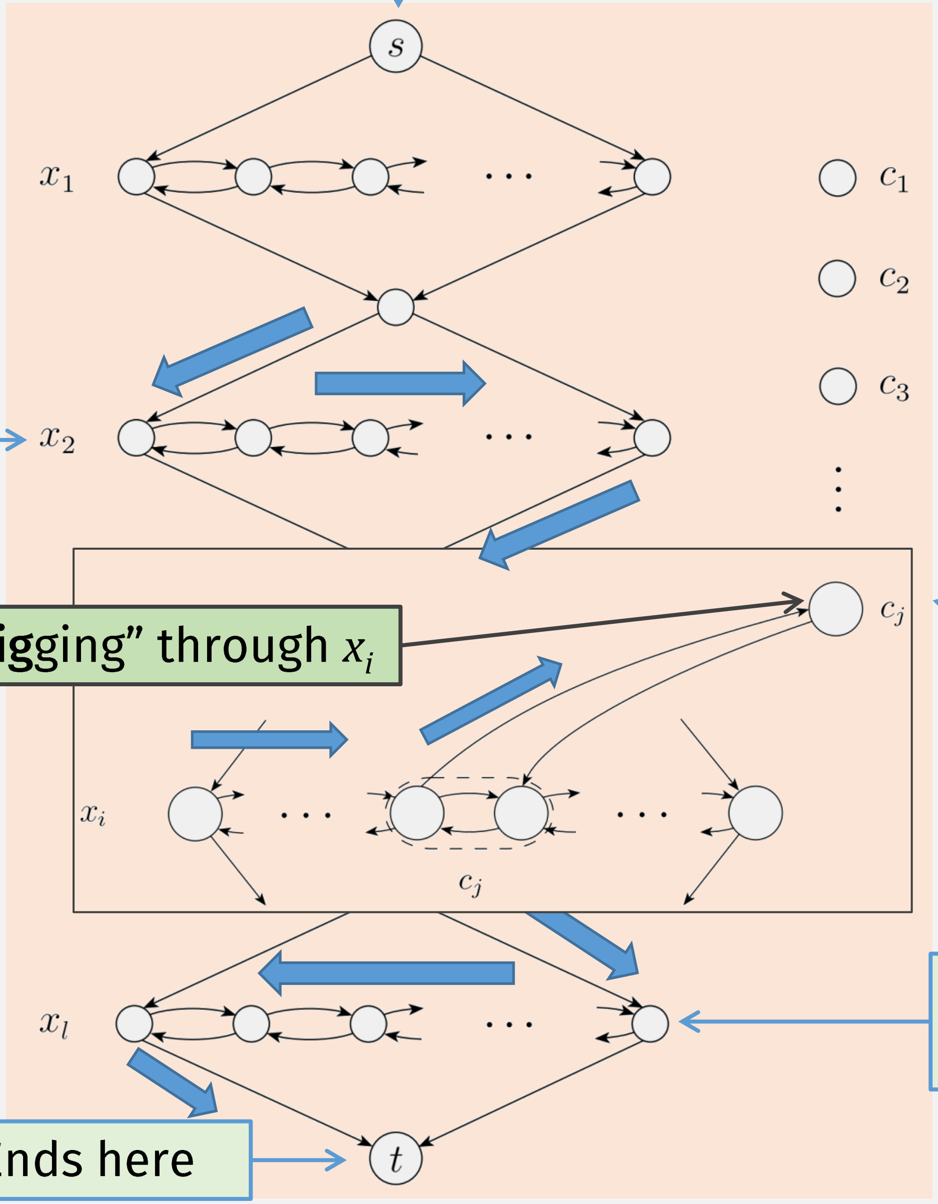
$(x_1 \vee \overline{x_2} \vee \overline{x_3}) \wedge (x_3 \vee \overline{x_5} \vee x_6) \wedge (x_3 \vee \overline{x_6} \vee x_4)$





A **Ham. Path** (must touch all nodes) **through this graph:**

1. Starts here



3a. and either **"zig-zags"** ...

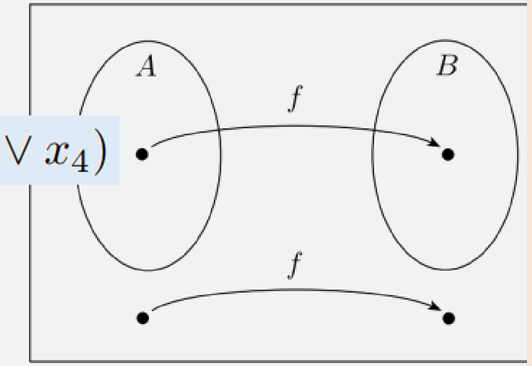
Can only go to this c_j if "**zig**ging" through x_i

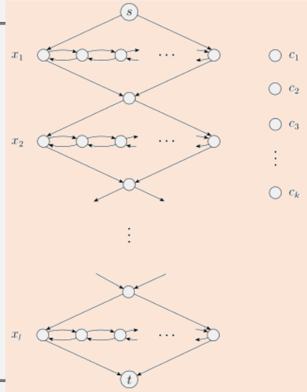
4. and **must "detour" to these clause gadgets** (at least once, has 3 chances)

3b. or **"zag-zigs"** through **each variable gadget**

2. Ends here

$(x_1 \vee \overline{x_2} \vee \overline{x_3}) \wedge (x_3 \vee \overline{x_5} \vee x_6) \wedge (x_3 \vee \overline{x_6} \vee x_4)$



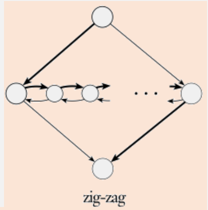


Want: **Satisfiable 3cnf formula** ⇔ **graph with Hamiltonian path**

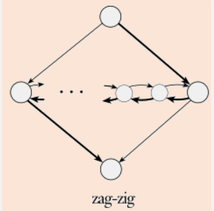
⇒ If **there is satisfying assignment,** then **Hamiltonian path exists**

These hit all nodes except extra c_js

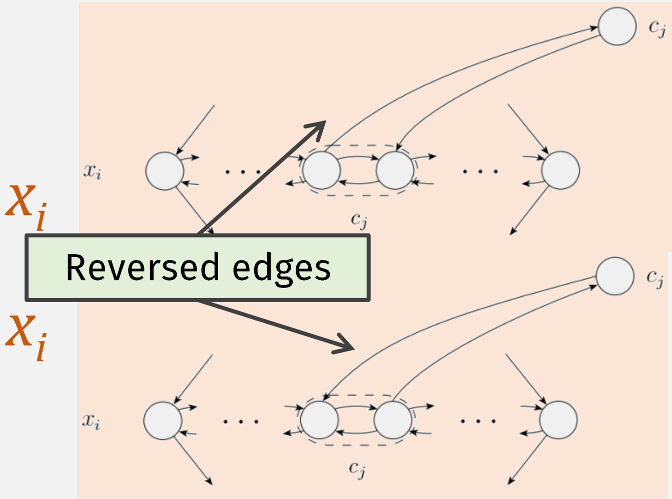
- x_i = TRUE → Hampath “zig-zags” gadget x_i



- x_i = FALSE → Hampath “zag-zigs” gadget x_i



- Lit x_i makes clause c_j TRUE → “detour” to c_j in gadget x_i
- Lit $\overline{x_i}$ makes clause c_j TRUE → “detour” to c_j in gadget x_i

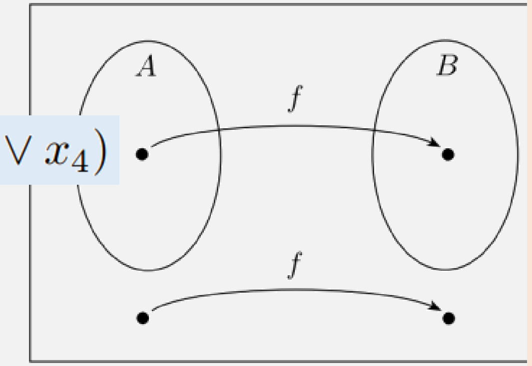


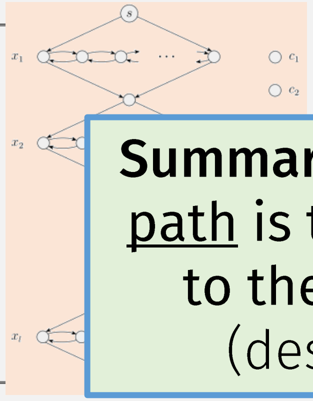
Now path goes through every node

Every clause must be TRUE so path hits all c_j nodes

- And edge directions align with TRUE/FALSE assignments

$(x_1 \vee \overline{x_2} \vee \overline{x_3}) \wedge (x_3 \vee \overline{x_5} \vee x_6) \wedge (x_3 \vee \overline{x_6} \vee x_4)$

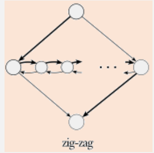




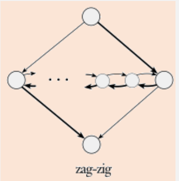
Summary: the only possible Ham. path is the one that corresponds to the satisfying assignment (described on prev slide)

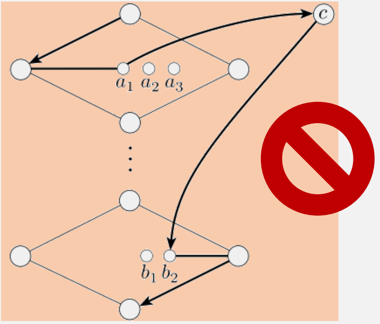
Want: Satisfiable 3cnf formula ⇔ graph with Hamiltonian path

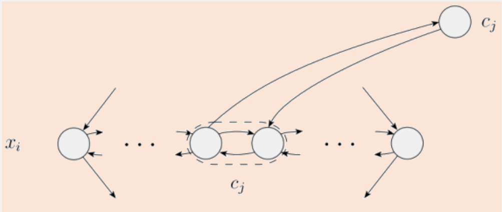
⇐ if output has Ham. path, then input had Satisfying assignment

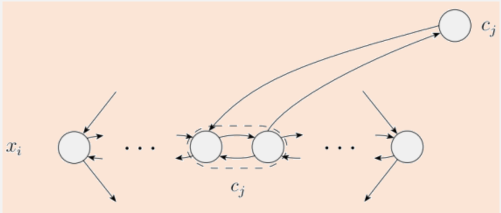


- A Hamiltonian path must choose to either zig-zag or zag-zig gadgets
- Ham path can only hit “detour” c_j nodes by coming right back
- Otherwise, it will miss some nodes





- 

gadget x_i “detours” from left to right → x_i = TRUE
- 

gadget x_i “detours” from right to left → x_i = FALSE

Theorem: *HAMPATH* is NP-complete

$$HAMPATH = \{\langle G, s, t\rangle \mid G \text{ is a directed graph with a Hamiltonian path from } s \text{ to } t\}$$

To prove *HAMPATH* is **NP**-complete:

☑ 1. Show *HAMPATH* is in **NP**

☑ 2. Choose *B*, the **NP**-complete problem to reduce from *3SAT*

☑ 3. Show a poly time mapping reduction from *3SAT* to *HAMPATH*

To show poly time <u>mapping reducibility</u>:

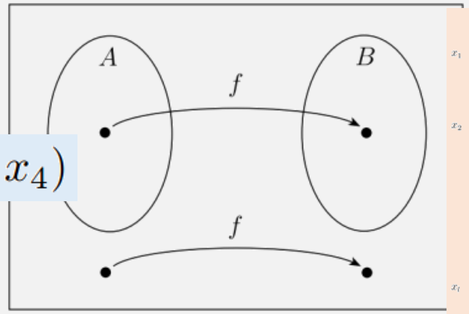
☑ 1. create **computable fn**,

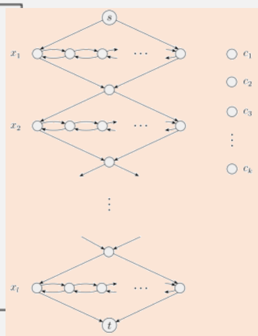
☑ 2. show that it **runs in poly time**,

☑ 3. then show **forward direction** of mapping red.,

☑ 4. and **reverse direction**
(or **contrapositive** of **reverse direction**)

$$(x_1 \vee \overline{x_2} \vee \overline{x_3}) \wedge (x_3 \vee \overline{x_5} \vee x_6) \wedge (x_3 \vee \overline{x_6} \vee x_4)$$





■

<u>Theorem</u>: *UHAMPATH* is NP-complete

un

$$UHAMPATH = \{\langle G, s, t\rangle \mid G \text{ is a directed graph with a Hamiltonian path from } s \text{ to } t\}$$

To prove *UHAMPATH* is **NP**-complete:

☑ 1. Show *UHAMPATH* is in **NP**

2. Choose the **NP**-complete problem to reduce from *HAMPATH*

3. Show a poly time mapping reduction from *???* to *UHAMPATH*

Theorem: *UHAMPATH* is NP-complete

$UHAMPATH = \{\langle G, s, t\rangle \mid G$ is a undirected graph with a Hamiltonian path from s to $t\}$

To prove *UHAMPATH* is **NP**-complete:

☑ 1. Show *UHAMPATH* is in **NP**

☑ 2. Choose the **NP**-complete problem to reduce from *HAMPATH*

3. Show a poly time mapping reduction from *HAMPATH* to *UHAMPATH*

Theorem: *UHAMPATH* is NP-complete

$$UHAMPATH = \{\langle G, s, t\rangle \mid G \text{ is a } \overset{\text{un}}{\text{directed}} \text{ graph with a Hamiltonian path from } s \text{ to } t\}$$

Need: **Computable function from *HAMPATH* to *UHAMPATH***

Naïve Idea: **Make all directed edges undirected?**

- But **we would create some paths that didn't exist before**

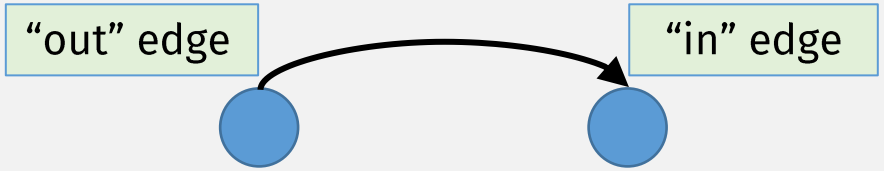
- **Doesn't work!**

Theorem: *UHAMPATH* is NP-complete

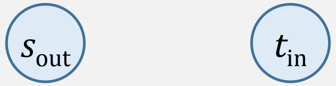
$$UHAMPATH = \{\langle G, s, t \rangle \mid G \text{ is a undirected graph with a Hamiltonian path from } s \text{ to } t\}$$

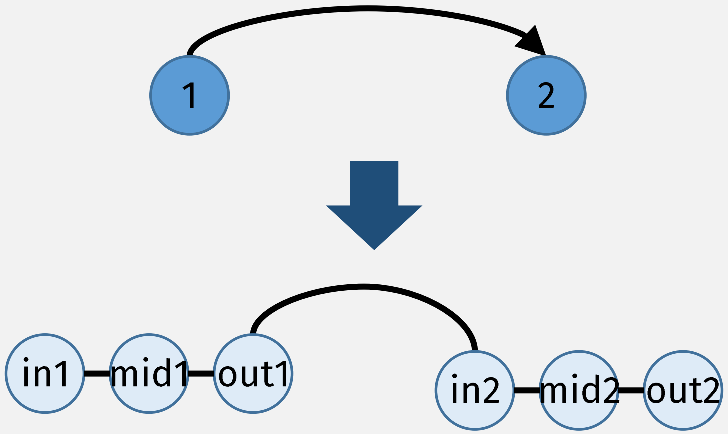
Need: **Computable function from *HAMPATH* to *UHAMPATH***

Better Idea:



- **Distinguish “in”** vs **“out” edges**
- **Nodes** (directed) → **3 Nodes** (undirected)**: in/mid/out**
  - Connect in/mid/out with edges
  - Directed edge (u, v) → $(u_{\text{out}}, v_{\text{in}})$
- Except: s → s_{out}, t → t_{in} **only!**





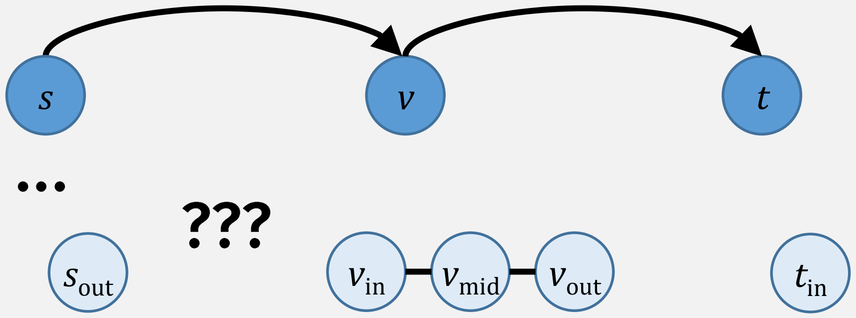
Theorem: *UHAMPATH* is NP-complete

$UHAMPATH = \{\langle G, s, t\rangle \mid G$ is a undirected graph with a Hamiltonian path from s to $t\}$

Need: **Computable function from *HAMPATH* to *UHAMPATH***

⇒ If **there is a directed path from** s **to** t **…**

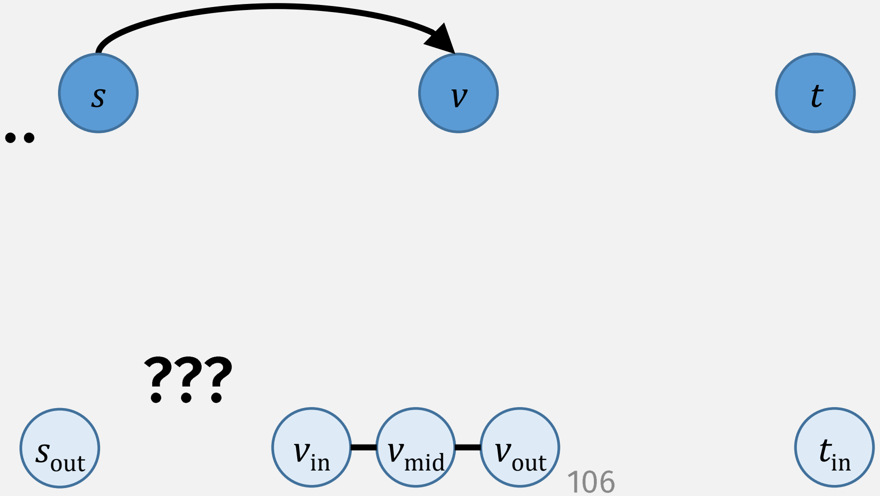
- … then **there must be an undirected path …**
- Because …



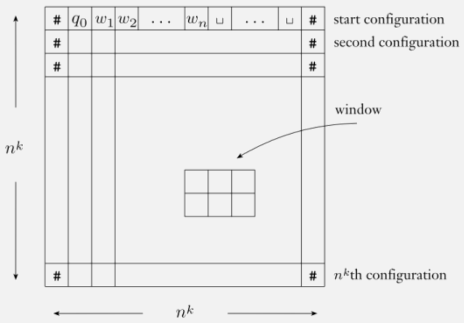
⇐ If **there is no directed path from** s **to** t **…**

- … then **there is no undirected path …**
- Because …

Left as exercise

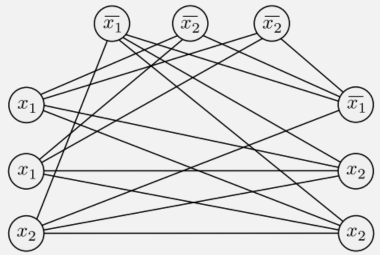


NP-Complete problems, so far

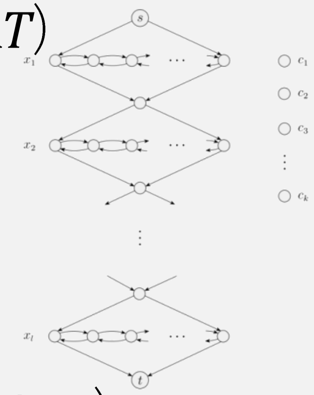


- $SAT = \{\langle \phi \rangle \mid \phi \text{ is a satisfiable Boolean formula}\}$ (Cook-Levin Theorem)

- $3SAT = \{\langle \phi \rangle \mid \phi \text{ is a satisfiable 3cnf-formula}\}$ (reduce from *SAT*)



- $CLIQUE = \{\langle G, k \rangle \mid G \text{ is an undirected graph with a } k\text{-clique}\}$ (reduce from 3*SAT*)



- $HAMPATH = \{\langle G, s, t \rangle \mid G \text{ is a directed graph with a Hamiltonian path from } s \text{ to } t\}$ (reduce from 3*SAT*)

- $UHAMPATH = \{\langle G, s, t \rangle \mid G \text{ is a undirected graph with a Hamiltonian path from } s \text{ to } t\}$ (reduce from *HAMPATH*)

More **NP**-Complete problems

- $$\begin{aligned} \textit{SUBSET-SUM} \; = \{\langle S, t\rangle |\ & S = \{x_1, \ldots, x_k\}, \text{ and for some} \\ & \{y_1, \ldots, y_l\} \subseteq \{x_1, \ldots, x_k\}, \text{ we have } \Sigma y_i = t\} \end{aligned}$$
  - (reduce from *3SAT*)
- $$\begin{aligned} \textit{VERTEX-COVER} = \{\langle G, k\rangle |\ & G \text{ is an undirected graph that} \\ & \text{has a } k\text{-node vertex cover}\} \end{aligned}$$
  - (reduce from *3SAT*)

<u>Theorem</u>: *SUBSET-SUM* is NP-complete

$$\mathit{SUBSET\text{-}SUM} = \{\langle S, t\rangle \mid S = \{x_1, \ldots, x_k\}, \text{ and for some } \{y_1, \ldots, y_l\} \subseteq \{x_1, \ldots, x_k\}, \text{ we have } \Sigma y_i = t\}$$

50 kg??

THEOREM

Using: If B is NP-complete and $B \leq_{\mathrm{P}} C$ for C in NP, then C is NP-complete.

3 steps to prove a language is **NP**-complete:

1. Show C is in **NP**
2. Choose B, the known **NP**-complete problem to reduce from
3. Show a poly time mapping reduction from B to C

Theorem: *SUBSET-SUM* is NP-complete

$SUBSET\text{-}SUM = \{\langle S, t\rangle \mid S = \{x_1, \ldots, x_k\}, \text{ and for some } \{y_1, \ldots, y_l\} \subseteq \{x_1, \ldots, x_k\}, \text{ we have } \Sigma y_i = t\}$

3 steps to prove *SUBSET-SUM* is **NP**-complete:

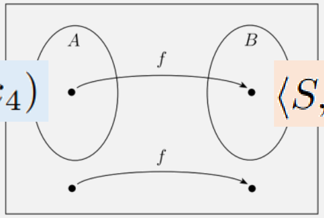
☑ 1. Show *SUBSET-SUM* is in **NP**

☑ 2. Choose the **NP**-complete problem to reduce from: *3SAT*

3. Show a poly time mapping reduction from *3SAT* to *SUBSET-SUM*

To show poly time mapping reducibility:
1. create **computable fn**,
2. show that it **runs in poly time**,
3. then show **forward direction** of mapping red.,
4. and **reverse direction**
   (or **contrapositive** of **reverse direction**)

$(x_1 \vee \overline{x_2} \vee \overline{x_3}) \wedge (x_3 \vee \overline{x_5} \vee x_6) \wedge (x_3 \vee \overline{x_6} \vee x_4)$

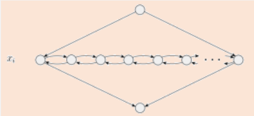


$\langle S, t\rangle \mid S = \{x_1, \ldots, x_k\}$

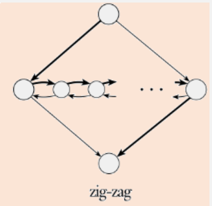
Review: Reducing from *3SAT*

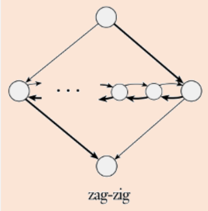
Create a **computable function** mapping formula to "gadgets":

- Clause → some "gadget", e.g., 

- Variable → another "gadget", e.g., 

  Gadget is typically used in two "opposite" ways, e.g.:
  - ZIG when var is assigned TRUE, or
  - ZAG when var is assigned FALSE

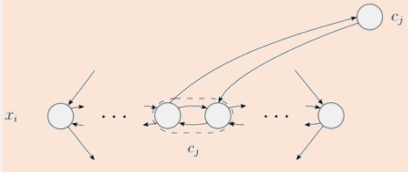
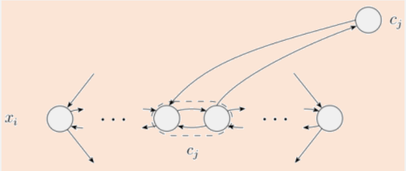




NOTE: "gadgets" are not always graphs

Then connect "gadgets" according to clause literals:

- Literal x_i in clause c_j → gadget x_i "detours" to c_j
- Literal $\overline{x_i}$ in clause c_j → gadget x_i "reverse detours" to c_j

Computable Fn: 3cnf → $\langle S, t\rangle$

E.g., $(\boxed{x_1} \vee \boxed{\overline{x_2}} \vee x_3) \wedge (x_2 \vee x_3 \vee \cdots) \wedge \cdots \wedge (\overline{x_3} \vee \cdots \vee \cdots)$ ➡

- Assume formula has:
  - ***l*** variables $x_1, \ldots, x_l$
  - ***k*** clauses $c_1, \ldots, c_k$
- Computable function *f* maps:
  - Variable x_i → two numbers y_i and z_i
  - Clause c_j → two numbers g_j and h_j
  - Digits arranged as rows in a table …
- Each number has max ***l+k*** digits:
  - Literal x_i in clause c_j → y_i: ***l***+j^{th} digit = 1
  - Literal $\overline{x_i}$ in clause c_j → z_i: ***l***+j^{th} digit = 1
- Sum is ***l*** 1s followed by ***k*** 3s

y_i and z_i: i^{th} digit = 1

y_i: ***l***+j^{th} digit = 1 if c_j has x_i

z_i: ***l***+j^{th} digit = 1 if c_j has $\overline{x_i}$

	1	2	3	4	$\cdots$	l	c_1	c_2	$\cdots$	c_k
y_1	1	0	0	0	$\cdots$	0	1	0	$\cdots$	0
z_1	1	0	0	0	$\cdots$	0	0	0	$\cdots$	0
y_2		1	0	0	$\cdots$	0	0	1	$\cdots$	0
z_2		1	0	0	$\cdots$	0	1	0	$\cdots$	0
y_3			1	0	$\cdots$	0	1	1	$\cdots$	0
z_3			1	0	$\cdots$	0	0	0	$\cdots$	1
$\vdots$					$\ddots$	$\vdots$	$\vdots$		$\vdots$	$\vdots$
y_l						1	0	0	$\cdots$	0
z_l						1	0	0	$\cdots$	0
g_1							1	0	$\cdots$	0
h_1							1	0	$\cdots$	0
g_2								1	$\cdots$	0
h_2								1	$\cdots$	0
$\vdots$									$\ddots$	$\vdots$
g_k										1
h_k										1
t	1	1	1	1	$\cdots$	1	3	3	$\cdots$	3

g_j and h_j: ***l***+j^{th} digit = 1, To help get correct sum

The sum

Theorem: *SUBSET-SUM* is NP-complete

$$\textit{SUBSET-SUM} = \{\langle S,t\rangle |\ S=\{x_1,\ldots,x_k\}, \text{ and for some } \{y_1,\ldots,y_l\} \subseteq \{x_1,\ldots,x_k\}, \text{ we have } \Sigma y_i = t\}$$

3 steps to prove *SUBSET-SUM* is **NP**-complete:

☑ 1. Show *SUBSET-SUM* is in **NP**

☑ 2. Choose the **NP**-complete problem to reduce from: *3SAT*

3. Show a poly time mapping reduction from *3SAT* to *SUBSET-SUM*

To show poly time mapping reducibility:

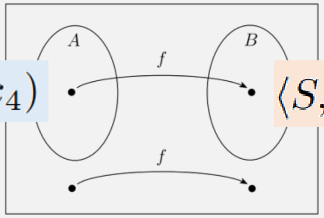
☑ 1. create **computable fn**,

2. show that it **runs in poly time**,

3. then show **forward direction** of mapping red.,

4. and **reverse direction**
(or **contrapositive** of **reverse direction**)

$(x_1 \vee \overline{x_2} \vee \overline{x_3}) \wedge (x_3 \vee \overline{x_5} \vee x_6) \wedge (x_3 \vee \overline{x_6} \vee x_4)$



$\langle S,t\rangle |\ S=\{x_1,\ldots,x_k\}$

Polynomial Time?

E.g., $(x_1 \vee \overline{x_2} \vee x_3) \wedge (x_2 \vee x_3 \vee \cdots) \wedge \cdots \wedge (\overline{x_3} \vee \cdots \vee \cdots)$ ➡

- Assume formula has:
  - ***l*** variables $x_1, \ldots, x_l$
  - ***k*** clauses $c_1, \ldots, c_k$
- Table size: $\boxed{(\boldsymbol{l} + \boldsymbol{k})} * \boxed{(2\boldsymbol{l} + 2\boldsymbol{k})}$
  - Creating it requires constant number of passes over the table
  - Num variables ***l*** = at most 3***k***
- Total: $\boxed{O(\boldsymbol{k}^2)}$

	1	2	3	4	$\cdots$	l	c_1	c_2	$\cdots$	c_k
y_1	1	0	0	0	$\cdots$	0	1	0	$\cdots$	0
z_1	1	0	0	0	$\cdots$	0	0	0	$\cdots$	0
y_2		1	0	0	$\cdots$	0	0	1	$\cdots$	0
z_2		1	0	0	$\cdots$	0	1	0	$\cdots$	0
y_3			1	0	$\cdots$	0	1	1	$\cdots$	0
z_3			1	0	$\cdots$	0	0	0	$\cdots$	1
$\vdots$					$\ddots$	$\vdots$	$\vdots$		$\vdots$	$\vdots$
y_l						1	0	0	$\cdots$	0
z_l						1	0	0	$\cdots$	0
g_1							1	0	$\cdots$	0
h_1							1	0	$\cdots$	0
g_2								1	$\cdots$	0
h_2								1	$\cdots$	0
$\vdots$									$\ddots$	$\vdots$
g_k										1
h_k										1
t	1	1	1	1	$\cdots$	1	3	3	$\cdots$	3

Theorem: *SUBSET-SUM* is NP-complete

$$\textit{SUBSET-SUM} = \{\langle S, t\rangle \mid S = \{x_1, \ldots, x_k\}, \text{ and for some } \{y_1, \ldots, y_l\} \subseteq \{x_1, \ldots, x_k\}, \text{ we have } \Sigma y_i = t\}$$

<u>3 steps</u> to prove *SUBSET-SUM* is **NP**-complete:

☑ 1. Show *SUBSET-SUM* is in **NP**

☑ 2. Choose the **NP**-complete problem to reduce from: *3SAT*

3. Show a poly time mapping reduction from *3SAT* to *SUBSET-SUM*

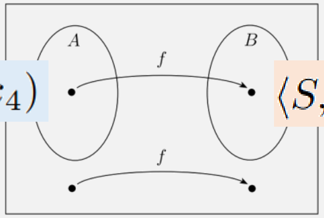
To show poly time <u>mapping reducibility</u>:

☑ 1. create **computable fn**,

☑ 2. show that it **runs in poly time**,

3. then show **forward direction** of mapping red.,
4. and **reverse direction**
   (or **contrapositive** of **reverse direction**)

$(x_1 \vee \overline{x_2} \vee \overline{x_3}) \wedge (x_3 \vee \overline{x_5} \vee x_6) \wedge (x_3 \vee \overline{x_6} \vee x_4)$



$\langle S, t\rangle \mid S = \{x_1, \ldots, x_k\}$

ϕ is a satisfiable 3cnf-formula $\iff$ $f(\langle\phi\rangle) = \langle S, t\rangle$ where some subset of S sums to t

Each column:
- At least one 1
- At most 3 1s

⇒ If **formula is satisfiable …**

- **Sum t = $\boldsymbol{l}$ 1s followed by $\boldsymbol{k}$ 3s**
- **Choose for the subset …**
  - y_i if x_i = TRUE
  - z_i if x_i = FALSE
  - and some of g_i and h_i to make the sum t
- … Then **this subset of S must sum to t bc:**
  - <u>Left digits</u>:
    - only one of y_i or z_i is in S
  - <u>Right digits</u>:
    - <u>Top right</u>: Each column sums to 1, 2, or 3
      - Because each clause has 3 literals
    - <u>Bottom right</u>:
      - Can always use g_i and/or h_i to make column sum to 3

S only includes one of these

	1	2	3	4	⋯	l	c_1	c_2	⋯	c_k
y_1	1	0	0	0	⋯	0	1	0	⋯	0
z_1	1	0	0	0	⋯	0	0	0	⋯	0
y_2		1	0	0	⋯	0	0	1	⋯	0
z_2		1	0	0	⋯	0	1	0	⋯	0
y_3			1	0	⋯	0	1	1	⋯	0
z_3			1	0	⋯	0	0	0	⋯	1
⋮					⋱	⋮	⋮		⋮	⋮
y_l						1	0	0	⋯	0
z_l						1	0	0	⋯	0
g_1							1	0	⋯	0
h_1							1	0	⋯	0
g_2								1	⋯	0
h_2								1	⋯	0
⋮									⋱	⋮
g_k										1
h_k										1
t	1	1	1	1	⋯	1	3	3	⋯	3

g_j and h_j: help get the correct sum

So each column sum (for left digits) is 1

ϕ is a satisfiable 3cnf-formula $\iff$ $f(\langle\phi\rangle) = \langle S, t\rangle$ where some sub … o t

Subset must have some number with 1 in each right column

⇐ If a subset of S sums to t ...

The <u>only way</u> to do it is as prev described:

- It can only include either y_i or z_i
  - Because each <u>left digit column</u> must sum to 1
  - And no carrying is possible
- Also, since each <u>right digit column</u> must sum to 3:
  - And only 2 can come from g_i and h_i
  - Then for every right column, some y_i or z_i in the subset has a 1 in that column
- ... Then table must have been created from a sat. ϕ:
  - x_i = TRUE if y_i in the subset
  - x_i = FALSE if z_i in the subset
- This is satisfying because:
  - Table was constructed so 1 in column c_j for y_i or z_i means that variable x_i satisfies clause c_j
  - We already determined, for every right column, some number in the subset has a 1 in the column
  - So all clauses are satisfied

S only includes y_i <u>or</u> z_i

	1	2	3	4	$\cdots$	l	c_1	c_2	$\cdots$	c_k
y_1	1	0	0	0	$\cdots$	0	1	0	$\cdots$	0
z_1	1	0	0	0	$\cdots$	0	0	0	$\cdots$	0
y_2		1	0	0	$\cdots$	0	0	1	$\cdots$	0
z_2		1	0	0	$\cdots$	0	1	0	$\cdots$	0
y_3			1	0	$\cdots$	0	1	1	$\cdots$	0
z_3			1	0	$\cdots$	0	0	0	$\cdots$	1
$\vdots$					$\ddots$	$\vdots$	$\vdots$		$\vdots$	$\vdots$
						1	0	0	$\cdots$	0
						1	0	0	$\cdots$	0
							1	0	$\cdots$	0
							1	0	$\cdots$	0
								1	$\cdots$	0
h_2								1	$\cdots$	0
$\vdots$									$\ddots$	$\vdots$
g_k										1
h_k										1
t	1	1	1	1	$\cdots$	1	3	3	$\cdots$	3

In each right column, g_i and h_i can account for at most 2

Because each column sum (for left digits) is 1

More **NP**-Complete problems

☑ • $$\mathit{SUBSET\text{-}SUM} = \{\langle S, t\rangle \mid S = \{x_1, \ldots, x_k\}, \text{ and for some } \{y_1, \ldots, y_l\} \subseteq \{x_1, \ldots, x_k\}, \text{ we have } \Sigma y_i = t\}$$

  - (reduce from *3SAT*)

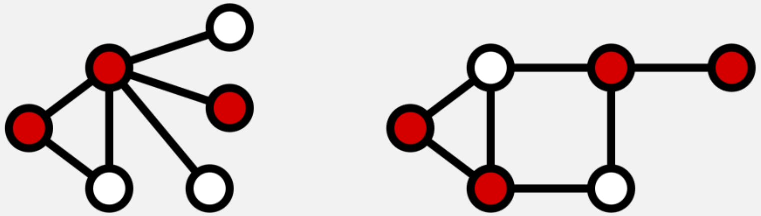
- $$\mathit{VERTEX\text{-}COVER} = \{\langle G, k\rangle \mid G \text{ is an undirected graph that has a } k\text{-node vertex cover}\}$$

  - (reduce from *3SAT*)

Theorem: *VERTEX-COVER* is NP-complete

$$VERTEX\text{-}COVER = \{\langle G, k\rangle \mid G \text{ is an undirected graph that has a } k\text{-node vertex cover}\}$$

- A **vertex cover** of a graph is …
  - … a subset of its nodes where every edge touches one of those nodes



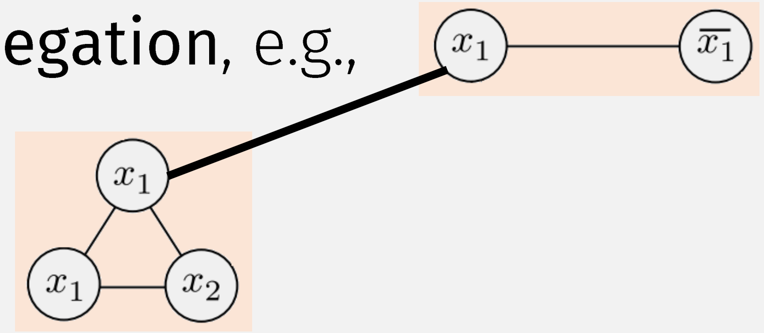
Theorem: *VERTEX-COVER* is NP-complete

$VERTEX\text{-}COVER = \{\langle G, k\rangle \mid G$ is an undirected graph that has a k-node vertex cover$\}$

- A **vertex cover** of a graph is …
  - … a subset of its nodes where every edge touches one of those nodes

Proof Sketch: Reduce *3SAT* to *VERTEX-COVER*

- The reduction maps:
- Variable x_i → 2 connected nodes
  - corresponding to the var and its negation, e.g.,
- Clause → 3 connected nodes
  - corresponding to its literals, e.g.,

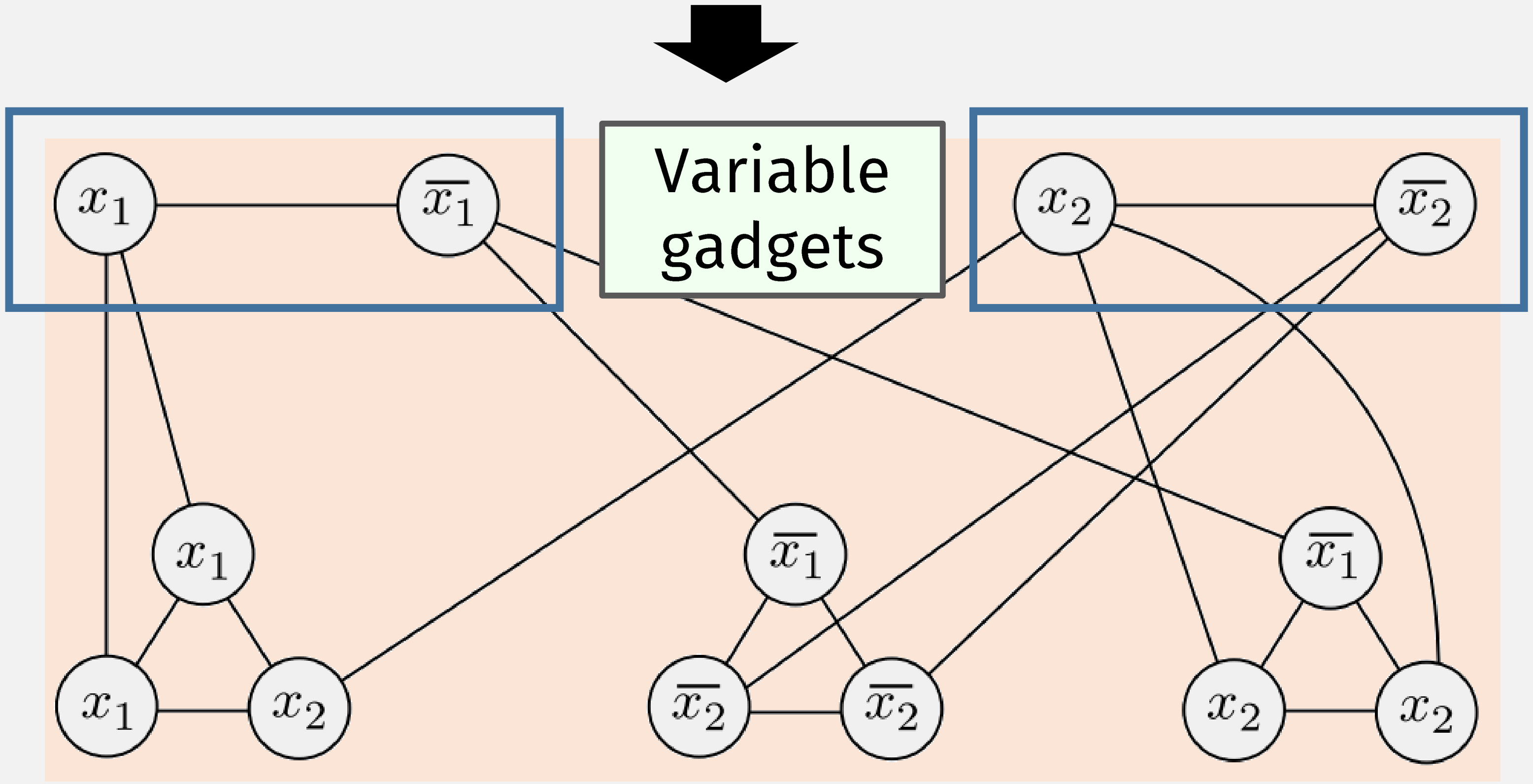


- Additionally,
  - connect var and clause gadgets by …
  - … connecting nodes that correspond to the same literal

VERTEX-COVER example

VERTEX-COVER $= \{\langle G, k\rangle |$ G is an undirected graph that has a k-node vertex cover$\}$

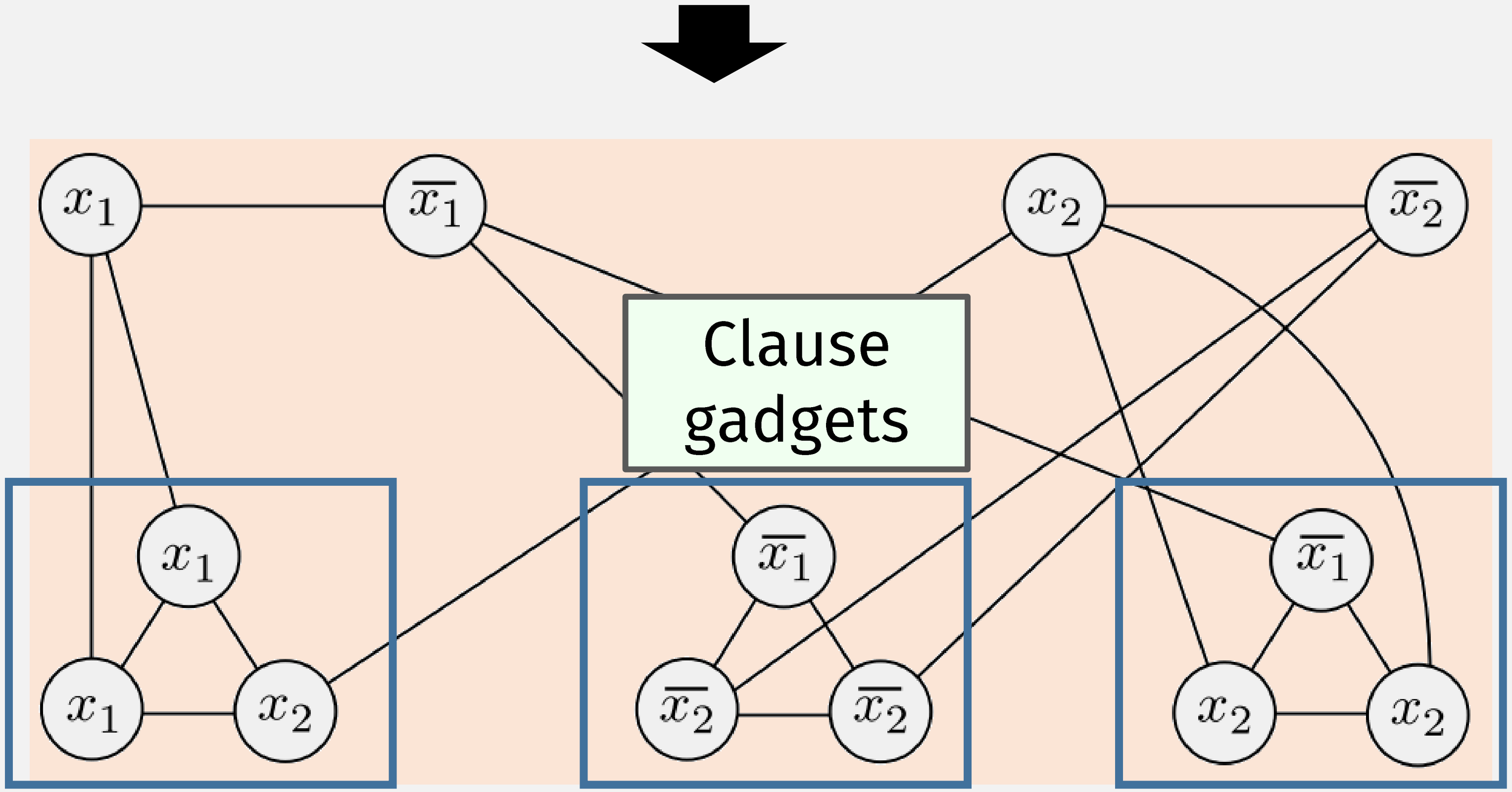
$$\phi = (x_1 \vee x_1 \vee x_2) \wedge (\overline{x_1} \vee \overline{x_2} \vee \overline{x_2}) \wedge (\overline{x_1} \vee x_2 \vee x_2)$$



VERTEX-COVER example

VERTEX-COVER $= \{\langle G, k\rangle \mid G$ is an undirected graph that has a k-node vertex cover$\}$

$$\phi = (x_1 \vee x_1 \vee x_2) \wedge (\overline{x_1} \vee \overline{x_2} \vee \overline{x_2}) \wedge (\overline{x_1} \vee x_2 \vee x_2)$$

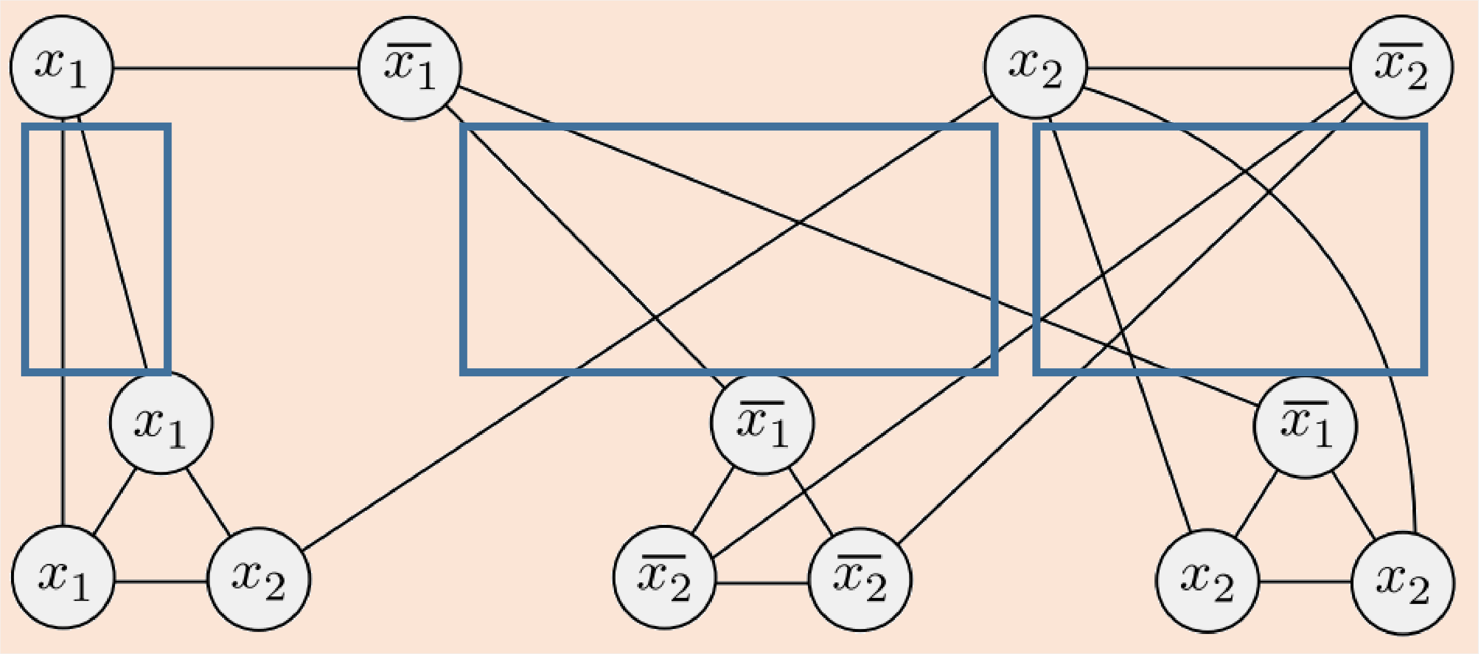


VERTEX-COVER example

VERTEX-COVER = $\{\langle G, k\rangle \mid G$ is an undirected graph that has a k-node vertex cover$\}$

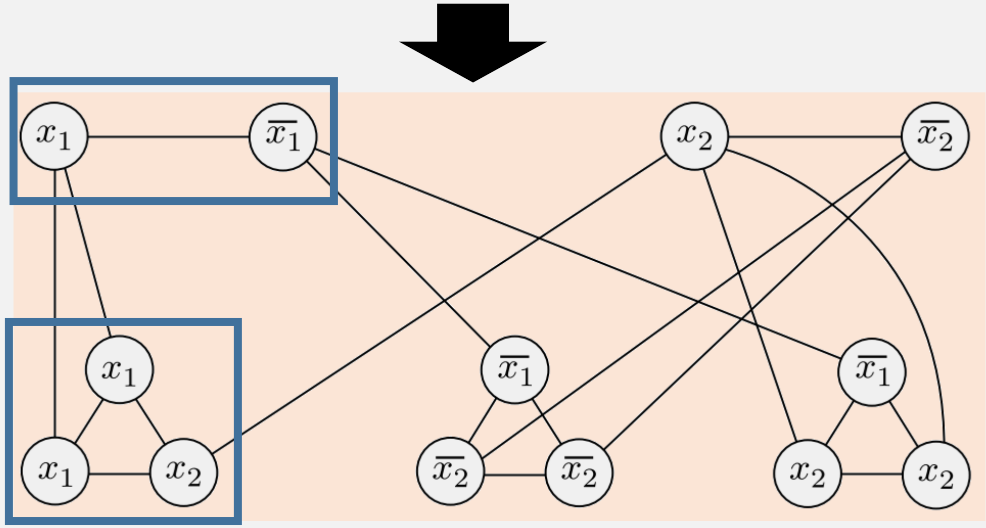
$$\phi = (x_1 \vee x_1 \vee x_2) \wedge (\overline{x_1} \vee \overline{x_2} \vee \overline{x_2}) \wedge (\overline{x_1} \vee x_2 \vee x_2)$$

Extra edges connecting variable and clause gadgets together



$$\phi = (x_1 \vee x_1 \vee x_2) \wedge (\overline{x_1} \vee \overline{x_2} \vee \overline{x_2}) \wedge (\overline{x_1} \vee x_2 \vee x_2)$$

VERTEX-COVER example



- If **formula has ...**
  - ***m*** = # variables
  - ***l*** = # clauses
- Then **graph has ...**
  - <u># nodes</u> = 2 × #vars + 3 × #clauses = <u>2***m*** + 3***l***</u>

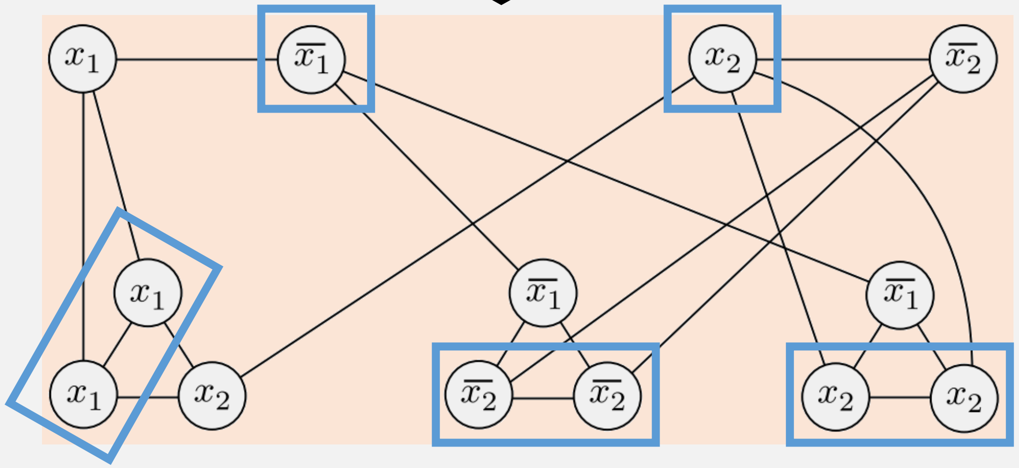
⇒ <u>If</u> satisfying assignment, <u>then</u> there is a ***k***-cover, where ***k*** = ***m*** + **2*l***

- Nodes in the cover are:
  - In each of ***m*** var gadgets, <u>choose 1</u> node corresponding to TRUE literal
  - For each of ***l*** clause gadgets, ignore 1 TRUE literal and <u>choose other 2</u>
  - Since there is satisfying assignment, each clause has a TRUE literal
  - <u>Total nodes in cover</u> = ***m*** + **2*l***

$VERTEX\text{-}COVER = \{\langle G, k\rangle \mid G$ is an undirected graph that has a k-node vertex cover$\}$

$\phi = (x_1 \vee x_1 \vee x_2) \wedge (\overline{x_1} \vee \overline{x_2} \vee \overline{x_2}) \wedge (\overline{x_1} \vee x_2 \vee x_2)$

VERTEX-COVER example



- If formula has …
  - $\boldsymbol{m}$ = # variables
  - $\boldsymbol{l}$ = # clauses
- Then graph has …
  - # nodes = 2$\boldsymbol{m}$ + 3$\boldsymbol{l}$

Example:
x_1 = FALSE
x_2 = TRUE

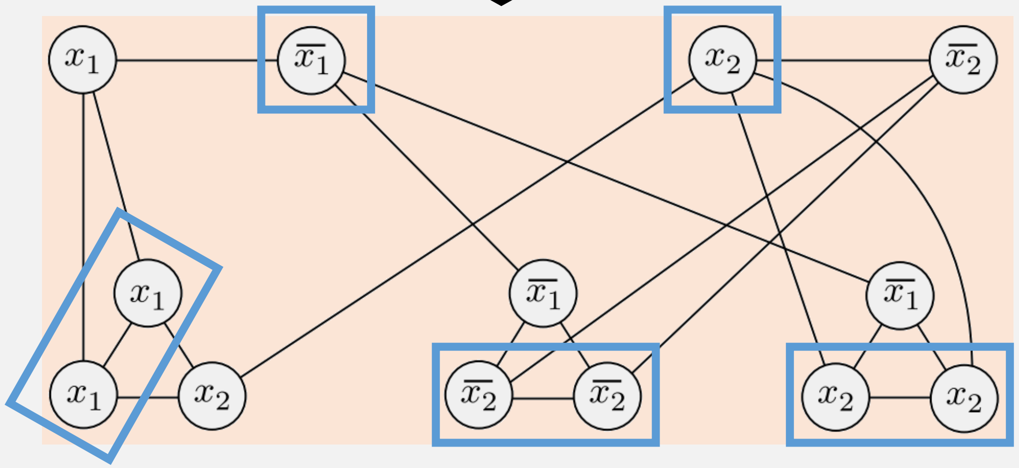
⇒ If satisfying assignment, then there is a $\boldsymbol{k}$-cover, where $\boldsymbol{k} = \boldsymbol{m} + 2\boldsymbol{l}$

- Nodes in the cover are:
  - In each of $\boldsymbol{m}$ var gadgets, choose 1 node corresponding to TRUE literal
  - For each of $\boldsymbol{l}$ clause gadgets, ignore 1 TRUE literal and choose other 2
  - Since there is satisfying assignment, each clause has a TRUE literal
  - Total nodes in cover = $\boldsymbol{m} + 2\boldsymbol{l}$

$VERTEX\text{-}COVER = \{\langle G, k\rangle \mid G$ is an undirected graph that has a k-node vertex cover$\}$

$$\phi = (x_1 \vee x_1 \vee x_2) \wedge (\overline{x_1} \vee \overline{x_2} \vee \overline{x_2}) \wedge (\overline{x_1} \vee x_2 \vee x_2)$$

VERTEX-COVER example



- If formula has …
  - $\boldsymbol{m}$ = # variables
  - $\boldsymbol{l}$ = # clauses
- Then graph has …
  - # nodes = $2\boldsymbol{m} + 3\boldsymbol{l}$

Example:
x_1 = FALSE
x_2 = TRUE

⇐ If there is a $\boldsymbol{k} = \boldsymbol{m} + 2\boldsymbol{l}$ cover,

- Then it can only be a $\boldsymbol{k}$-cover as described on the last slide …
  - 1 node (and only 1) from each of "var" gadgets
  - 2 nodes (and only 2) from each "clause" gadget
  - Any other set of $\boldsymbol{k}$ nodes is not a cover
- Which means that input has satisfying assignment:
  - x_i = TRUE if node x_i is in cover, else x_i = FALSE

VERTEX-COVER $= \{\langle G, k\rangle \mid G$ is an undirected graph that has a k-node vertex cover$\}$

More **NP**-Complete problems

- ☑ $$SUBSET\text{-}SUM = \{\langle S, t\rangle \mid S = \{x_1, \ldots, x_k\}, \text{ and for some } \{y_1, \ldots, y_l\} \subseteq \{x_1, \ldots, x_k\}, \text{ we have } \Sigma y_i = t\}$$
  - (reduce from *3SAT*)
- ☑ $$VERTEX\text{-}COVER = \{\langle G, k\rangle \mid G \text{ is an undirected graph that has a } k\text{-node vertex cover}\}$$
  - (reduce from *3SAT*)

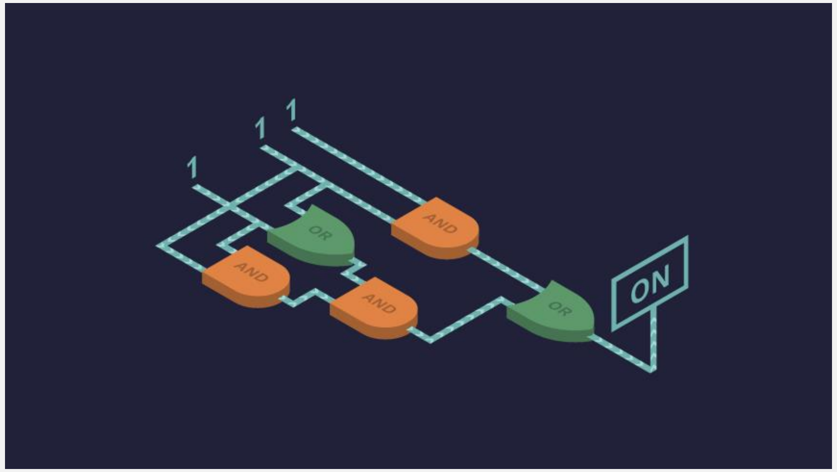
Next Time: The Cook-Levin Theorem

The first **NP**-Complete problem

THEOREM

SAT is NP-complete.

It sort of makes sense that every problem can be reduced to it …



After this, it'll be much easier to find other **NP**-Complete problems!

THEOREM

If B is NP-complete and $B \leq_{\mathrm{P}} C$ for C in NP, then C is NP-complete.