NP

Completeness
CS Kickball!
4PM at FloMo Field
Announcements

- Problem Set 8 due right now.
- Problem Set 9 out, due next Wednesday, June 6 at 2:15 PM.
  - Play around with $P$, $NP$, and $NP$-completeness.
  - Get a sense for the big picture.
Final Exam Logistics

- Final Exam next Friday, June 8 from 12:15PM – 3:15PM
- Location distributed by last name:
  - A – R: Go to Nvidia Auditorium (right here!)
  - S – Z: Go to Huang 018 (next door!)
- Covers material up through and including next Monday's lecture.
  - Heavily weighted toward computability and complexity theory; however, there may be a midterm-style question on the exam for completeness.
- Open-book, open-note, open-computer, closed-network.
Alternate Exams

• Alternate Exam Times:
  • **Thursday, June 7** from 4:00PM – 7:00PM (location TBA)
  • **Friday, June 8** from 8:30AM – 11:30AM (location TBA)
  • Please let us know about alternate exam preferences by **Monday** so we can plan accordingly.
Practice Final Exams

• There is an extra credit practice final available.
• If you make a good-faith effort to complete it, it's worth +5 extra credit points.
  • Your answers don't have to be right; they just have to show that you made a good-faith effort to solve everything.
• Due when you take the final!
• Other practice finals available, but they aren't worth any extra credit.
Please evaluate this course on Axess!

Your feedback really does make a difference.
Previously on CS103...
NP-Completeness

- A language $L$ is called **NP-hard** if there is a polynomial-time reduction from every problem in **NP** to $L$.
- A language in $L$ is called **NP-complete** if $L \in \text{NP}$ and $L$ is NP-hard.
- The class **NPC** is the set of NP-complete problems.
A language $L$ is called **NP-hard** if there is a polynomial-time reduction from every problem in **NP** to $L$.

A language in $L$ is called **NP-complete** if $L \in \text{NP}$ and $L$ is **NP-hard**.

The class **NPC** is the set of **NP-complete** problems.
NP-Completeness

- A language $L$ is called **NP-hard** if there is a polynomial-time reduction from *every* problem in **NP** to $L$.
- A language in $L$ is called **NP-complete** if $L \in \text{NP}$ and $L$ is **NP-hard**.
- The class **NPC** is the set of **NP-complete** problems.
The Tantalizing Truth

- **Theorem**: If *any* NP-complete language is in P, then \( P = NP \).
- **Proof**: If \( L \in NPC \) and \( L \in P \), we know that for any \( L' \in NP \), that \( L' \leq_P L \) because \( L \) is NP-complete. Since \( L' \leq_P L \) and \( L \in P \), this means that \( L' \in P \). Since our choice of \( L' \) was arbitrary, any language \( L' \in NP \) satisfies \( L' \in P \), so \( NP \subseteq P \). Since \( P \subseteq NP \), this means \( P = NP \). ■
The Tantalizing Truth

- **Theorem**: If *any* NP-complete language is in \( P \), then \( P = NP \).

- **Proof**: If \( L \in NPC \) and \( L \in P \), we know that for any \( L' \in NP \), that \( L' \leq_p L \) because \( L \) is NP-complete. Since \( L' \leq_p L \) and \( L \in P \), this means that \( L' \in P \). Since our choice of \( L' \) was arbitrary, any language \( L' \in NP \) satisfies \( L' \in P \), so \( NP \subseteq P \). Since \( P \subseteq NP \), this means \( P = NP \). ■
The Tantalizing Truth

- **Theorem:** If *any* NP-complete language is in P, then P = NP.

- **Proof:** If $L \in \text{NPC}$ and $L \in \text{P}$, we know that for any $L' \in \text{NP}$, that $L' \leq_p L$ because $L$ is NP-complete. Since $L' \leq_p L$ and $L \in \text{P}$, this means that $L' \in \text{P}$. Since our choice of $L'$ was arbitrary, any language $L' \in \text{NP}$ satisfies $L' \in \text{P}$, so NP $\subseteq$ P. Since P $\subseteq$ NP, this means P = NP. ■
The Tantalizing Truth

- **Theorem**: If *any* NP-complete language is in P, then P = NP.

- **Proof**: If \( L \in \text{NPC} \) and \( L \in P \), we know that for any \( L' \in \text{NP} \), that \( L' \leq_p L \) because \( L \) is NP-complete. Since \( L' \leq_p L \) and \( L \in P \), this means that \( L' \in P \). Since our choice of \( L' \) was arbitrary, any language \( L' \in \text{NP} \) satisfies \( L' \in P \), so \( \text{NP} \subseteq P \). Since \( P \subseteq \text{NP} \), this means \( P = \text{NP} \). ■
The Tantalizing Truth

- **Theorem:** If *any* NP-complete language is in \( P \), then \( P = \text{NP} \).

- **Proof:** If \( L \in \text{NPC} \) and \( L \in P \), we know that for any \( L' \in \text{NP} \), that \( L' \leq_p L \) because \( L \) is NP-complete. Since \( L' \leq_p L \) and \( L \in P \), this means that \( L' \in P \). Since our choice of \( L' \) was arbitrary, any language \( L' \in \text{NP} \) satisfies \( L' \in P \), so \( \text{NP} \subseteq P \). Since \( P \subseteq \text{NP} \), this means \( P = \text{NP} \).

\[ P = \text{NP} \]
The Tantalizing Truth

- **Theorem:** If *any* \textit{NP}-complete language is not in \textit{P}, then \textit{P} ≠ \textit{NP}.

- **Proof:** If \textit{L} is \textit{NP}-complete, it is in \textit{NP}. If it is in \textit{NP} but not \textit{P}, then \textit{P} ≠ \textit{NP}. ■
What Problems are NP-Complete?

- **NP-complete** problems give a promising approach for resolving $P \equiv NP$:
  - If any **NPC** problem is in $P$, then $P = NP$.
  - If any **NPC** problem is not in $P$, then $P \neq NP$.
- However, we haven't shown that any problems are **NP**-complete in the first place!
- How do we even know they exist?
Satisfiability

- A propositional logic formula $\phi$ is called **satisfiable** if there is some assignment to its variables that makes it evaluate to true.

- An assignment of true and false to the variables of $\phi$ that makes it evaluate to true is called a **satisfying assignment**.

- Similar terms:
  - $\phi$ is **tautological** if it is always true.
  - $\phi$ is **satisfiable** if it *can* be made true.
  - $\phi$ is **unsatisfiable** if it is always false.
SAT

• The **boolean satisfiability problem** (**SAT**) is the following:

  *Given a propositional logic formula \( \phi \), is \( \phi \) satisfiable?*

• Formally:

  \[
  \text{SAT} = \{ \langle \phi \rangle \mid \phi \text{ is a satisfiable PL formula} \} 
  \]
Theorem (Cook-Levin): SAT is $\textbf{NP}$-complete.
Sketch of the Proof

- We need to show that every single language in $\textbf{NP}$ has a polynomial-time reduction to SAT.
- To do so, we will use the fact that every language in $\textbf{NP}$ has a polynomial-time NTM.
- We can build a SAT formula that encodes the rules for how that NTM operates.
- If there is some set of choices where the NTM accepts, our formula will be satisfiable.
- If there are no choices we can make where the NTM accepts, our formula will be unsatisfiable.
Polynomial-Time NTMs

- Recall: The time complexity of an NTM is defined as the height of its execution tree.
- If an NTM runs in polynomial time, there is some polynomial $p(n)$ such that no execution of the NTM on a string of size $n$ takes time more than $p(n)$.
- This means that the NTM never uses more than $p(n)$ tape when running on a string of length $n$. 
State

\begin{align*}
q_0 & : 0101 \\
q_3 & : 1101 \\
q_4 & : 1001 \\
q_{\text{acc}} & : 1101
\end{align*}
Proving the Cook-Levin Theorem

• Build a PL formula encoding the following:
  • The machine starts off with the tape head at far left, in state $q_0$, and tape initialized to the input string.
  • Each snapshot of the execution legally follows from the previous snapshot.
  • The computation accepts.
• This formula can be enormously complex, but its size is still a polynomial!
• See Sipser pgs. 276-283 for details.
Literals and Clauses

- A **literal** in propositional logic is a variable or its negation:
  - $x$
  - $\neg y$
  - But not $x \land y$.

- A **clause** is a many-way OR (*disjunction*) of literals.
  - $\neg x \lor y \lor \neg z$
  - $x$
  - But not $x \lor \neg(y \lor z)$
Conjunctive Normal Form

- A propositional logic formula $\phi$ is in **conjunctive normal form (CNF)** if it is the many-way AND (**conjunction**) of clauses.
  - $(x \lor y \lor z) \land (\neg x \lor \neg y) \land (x \lor y \lor z \lor \neg w)$
  - $x \lor z$
  - But not $(x \lor (y \land z)) \lor (x \lor y)$
- Only legal operators are $\neg$, $\lor$, $\land$.
- No nesting allowed.
3CNF and 3SAT

• A propositional logic formula $\phi$ is in 3CNF iff it is in CNF, and each clause has exactly three literals.

• The language $3SAT$ is

\[
3SAT = \{ \langle \phi \rangle \mid \phi \text{ is a satisfiable 3CNF formula} \}
\]

• You saw how to convert arbitrary CNF formulas into 3CNF formulas in Problem Set 4.
Theorem (Cook-Levin): 3SAT is \textbf{NP}-complete.
Using the Cook-Levin Theorem

• When discussing decidability, we used the undecidability of $A_{TM}$ to prove that many other languages are undecidable.
  
  • **Idea:** Reduce $A_{TM}$ to some other language.

• Using the **NP**-completeness of 3SAT, we can show the **NP**-completeness of many other problems.
  
  • **Idea:** Reduce 3SAT to some other language.
NP-Completeness

- **Theorem**: If $L_1$ is $\text{NP}$-complete and $L_1 \leq_p L_2$ for some $L_2 \in \text{NP}$, then $L_2$ is $\text{NP}$-complete as well.

- **Proof Sketch**: For any language $L' \in \text{NP}$, we know that $L' \leq_p L_1$. We also know $L_1 \leq_p L_2$. Thus $L' \leq_p L_2$. Since our choice of $L'$ was arbitrary, we have that any $\text{NP}$ language is reducible to $L_2$ is in polynomial time, so $L_2$ is $\text{NP}$-hard. Since $L_2 \in \text{NP}$, this means that $L_2$ is $\text{NP}$-complete. ■
NP-Completeness

- **Theorem**: If $L_1$ is NP-complete and $L_1 \leq_p L_2$ for some $L_2 \in \text{NP}$, then $L_2$ is NP-complete as well.

- **Proof Sketch**: For any language $L' \in \text{NP}$, we know that $L' \leq_p L_1$. We also know $L_1 \leq_p L_2$. Thus $L' \leq_p L_2$. Since our choice of $L'$ was arbitrary, we have that any NP language is reducible to $L_2$ is in polynomial time, so $L_2$ is NP-hard. Since $L_2 \in \text{NP}$, this means that $L_2$ is NP-complete. ■
**NP-Completeness**

- **Theorem:** If $L_1$ is NP-complete and $L_1 \leq_p L_2$ for some $L_2 \in \text{NP}$, then $L_2$ is NP-complete as well.

- **Proof Sketch:** For any language $L' \in \text{NP}$, we know that $L' \leq_p L_1$. We also know $L_1 \leq_p L_2$. Thus $L' \leq_p L_2$. Since our choice of $L'$ was arbitrary, we have that any NP language is reducible to $L_2$ is in polynomial time, so $L_2$ is NP-hard. Since $L_2 \in \text{NP}$, this means that $L_2$ is NP-complete. ■
NP-Completeness

- **Theorem**: If \( L_1 \) is \( \text{NP} \)-complete and \( L_1 \leq_p L_2 \) for some \( L_2 \in \text{NP} \), then \( L_2 \) is \( \text{NP} \)-complete as well.

- **Proof Sketch**: For any language \( L' \in \text{NP} \), we know that \( L' \leq_p L_1 \). We also know \( L_1 \leq_p L_2 \). Thus \( L' \leq_p L_2 \). Since our choice of \( L' \) was arbitrary, we have that any \( \text{NP} \) language is reducible to \( L_2 \) is in polynomial time, so \( L_2 \) is \( \text{NP} \)-hard. Since \( L_2 \in \text{NP} \), this means that \( L_2 \) is \( \text{NP} \)-complete. ■
NP-Completeness

- **Theorem**: If $L_1$ is NP-complete and $L_1 \leq_p L_2$ for some $L_2 \in \text{NP}$, then $L_2$ is NP-complete as well.

- **Proof Sketch**: For any language $L' \in \text{NP}$, we know that $L' \leq_p L_1$. We also know $L_1 \leq_p L_2$. Thus $L' \leq_p L_2$. Since our choice of $L'$ was arbitrary, we have that any NP language is reducible to $L_2$ is in polynomial time, so $L_2$ is NP-hard. Since $L_2 \in \text{NP}$, this means that $L_2$ is NP-complete. ■
Be Careful!

- To show that a language is \textbf{NP}-complete, prove that it is in \textbf{NP}, then reduce a known \textbf{NP}-complete problem to it.
- \textbf{Do not} reduce the language to a known \textbf{NP}-complete problem!
  - We already knew that you could do this; every \textbf{NP} problem is reducible to any \textbf{NP}-complete problem!
Be Careful!

- To show that a language is \textbf{NP}-complete, prove that it is in \textbf{NP}, then reduce a known \textbf{NP}-complete problem to it.

- \textbf{Do not} reduce the language to a known \textbf{NP}-complete problem!
  - We already knew that you could do this; every \textbf{NP} problem is reducible to any \textbf{NP}-complete problem!
Be Careful!

- To show that a language is \textbf{NP}-complete, prove that it is in \textbf{NP}, then reduce a known \textbf{NP}-complete problem to it.
- \textbf{Do not} reduce the language to a known \textbf{NP}-complete problem!
  - We already knew that you could do this; every \textbf{NP} problem is reducible to any \textbf{NP}-complete problem!
Be Careful!

- To show that a language is \textbf{NP}-complete, prove that it is in \textbf{NP}, then reduce a known \textbf{NP}-complete problem to it.

- \textbf{Do not} reduce the language to a known \textbf{NP}-complete problem!
  - We already knew that you could do this; every \textbf{NP} problem is reducible to any \textbf{NP}-complete problem!
So what other problems are \textbf{NP-complete}?
An **independent set** in an undirected graph is a set of vertices that have no edges between them.
An **independent set** in an undirected graph is a set of vertices that have no edges between them.
An **independent set** in an undirected graph is a set of vertices that have no edges between them.
An **independent set** in an undirected graph is a set of vertices that have no edges between them.
An independent set in an undirected graph is a set of vertices that have no edges between them.
The Independent Set Problem

• Given an undirected graph $G$ and a natural number $n$, the independent set problem is:

  **Does $G$ contain an independent set of size at least $n$?**

• As a formal language:

  $$INDSET = \{ \langle G, n \rangle \mid G \text{ is an undirected graph with an independent set of size at least } n \}$$
The independent set problem is in \textbf{NP}.
Here is a polynomial-time verifier that checks whether $S$ is an $n$-element independent set:

$$V = \text{"On input } \langle G, n, S \rangle:\text{ If } |S| < n, \text{ reject.}\text{ For each edge in } G, \text{ if both endpoints are in } S, \text{ reject.}\text{ Otherwise, accept."}$$
INDSET is \textbf{NP}-Complete

- The \textit{INDSET} problem is \textbf{NP}-complete.
- Use a polynomial-time reduction from 3SAT:
  - Given a 3-CNF formula $\phi$ with $n$ clauses, construct a graph $G$ such that $\phi$ is satisfiable iff $G$ has an independent set of size $n$.
- How can we accomplish this?
The Structure of 3SAT

\[(x \lor y \lor \neg z) \land (\neg x \lor \neg y \lor z) \land (\neg x \lor y \lor \neg z)\]
The Structure of 3SAT

\[ (x \lor y \lor \neg z) \land (\neg x \lor \neg y \lor z) \land (\neg x \lor y \lor \neg z) \]
The Structure of 3SAT

\[(x \lor y \lor \neg z) \land (\neg x \lor \neg y \lor z) \land (\neg x \lor y \lor \neg z)\]

Each clause must have at least one true literal in it.
The Structure of 3SAT

\[(x \lor y \lor z) \land (\neg x \lor \neg y \lor z) \land (\neg x \lor y \lor \neg z)\]
The Structure of 3SAT

One way to think about solving 3SAT is to pick which literals in each clause need to be made true.
The Structure of 3SAT

\((x \lor y \lor \neg z) \land (\neg x \lor \neg y \lor z) \land (\neg x \lor y \lor \neg z)\)
The Structure of 3SAT

\[(x \lor y \lor \neg z) \land (\neg x \lor \neg y \lor z) \land (\neg x \lor y \lor \neg z)\]
The Structure of 3SAT

\[(x \lor y \lor \neg z) \land (\neg x \lor \neg y \lor z) \land (\neg x \lor y \lor \neg z)\]
The Structure of 3SAT

\[(x \lor y \lor \neg z) \land (\neg x \lor \neg y \lor z) \land (\neg x \lor y \lor \neg z)\]
The Structure of 3SAT

\[(x \lor y \lor \neg z) \land (\neg x \lor \neg y \lor z) \land (\neg x \lor y \lor \neg z)\]

... subject to the constraint that we never choose a literal and its negation
From 3SAT to INDSET

- To convert a 3SAT instance to $INDSET$, we need a graph such that an independent set in that graph
  - gives us a way to choose which literal in each clause should be true,
  - doesn't simultaneously choose a literal and its negation, and
  - has size polynomially large in the length of the original formula.
From 3SAT to INDSET

$$(x \lor y \lor z) \land (\neg x \lor \neg y \lor z) \land (\neg x \lor y \lor \neg z)$$
From 3SAT to INDSET

\[ (x \lor y \lor z) \land (\neg x \lor \neg y \lor z) \land (\neg x \lor y \lor \neg z) \]
From 3SAT to INDSET

\[(x \lor y \lor \neg z) \land (\neg x \lor \neg y \lor z) \land (\neg x \lor y \lor \neg z)\]
From 3SAT to INDSET

\[(x \lor y \lor \neg z) \land (\neg x \lor \neg y \lor z) \land (\neg x \lor y \lor \neg z) \land (\neg x \lor y \lor \neg z)\]
From 3SAT to INDSET

\[(\neg x \lor y \lor \neg z) \land (\neg y \lor \neg z \lor x) \land (\neg x \lor y \lor \neg z)\]

Any independent set in the graph chooses exactly one literal from each clause to be true.
From 3SAT to INDSET

\[
(x \lor y \lor \neg z) \land (\neg x \lor \neg y \lor z) \land (\neg x \lor y \lor \neg z)
\]

Any independent set in the graph chooses exactly one literal from each clause to be true.
From 3SAT to INDSET

\[(x \lor y \lor \neg z) \land (\neg x \lor \neg y \lor z) \land (\neg x \lor y \lor \neg z)\]

Any independent set in the graph chooses exactly one literal from each clause to be true.
From 3SAT to INDSET

\[(x \lor y \lor \neg z) \land (\neg x \lor \neg y \lor z) \land (\neg x \lor y \lor \neg z)\]

Any independent set in the graph chooses **exactly one** literal from each clause to be true.
Any independent set in the graph chooses exactly one literal from each clause to be true.
From 3SAT to INDSET

\[(x \lor y \lor \neg z) \land (\neg x \lor \neg y \lor z) \land (\neg x \lor y \lor \neg z)\]

Any independent set in the graph chooses exactly one literal from each clause to be true.
From 3SAT to INDSET

$(x \lor y \lor \neg z) \land (\neg x \lor \neg y \lor z) \land (\neg x \lor y \lor \neg z)$

Any independent set in the graph chooses exactly one literal from each clause to be true.
From 3SAT to INDSET

\[(\neg x \lor y \lor z) \land (\neg x \lor \neg y \lor \neg z) \land (x \lor \neg y \lor \neg z)\]

Any independent set in the graph chooses exactly one literal from each clause to be true.
From 3SAT to INDSET

We need a way to ensure that we don't pick a literal and its negation!
From 3SAT to INDSET

\[(x \lor y \lor \neg z) \land (\neg x \lor \neg y \lor z) \land (\neg x \lor y \lor \neg z) \land (\neg y \lor \neg z \lor x) \land (\neg z \lor y \lor x)\]
From 3SAT to INDSET

\[(x \lor y \lor \neg z) \land (\neg x \lor \neg y \lor z) \land (\neg x \lor y \lor \neg z)\]
From 3SAT to INDSET

\[ (x \lor y \lor z) \land (\neg x \lor \neg y \lor z) \land (\neg x \lor y \lor \neg z) \]
From 3SAT to INDSET

\[(x \lor y \lor z) \land (\neg x \lor \neg y \lor z) \land (\neg x \lor y \lor \neg z) \land (\neg y \lor x \lor z) \land (\neg z \lor y \lor x)\]
From 3SAT to INDSET

\[(x \lor y \lor \neg z) \land (\neg x \lor \neg y \lor z) \land (\neg x \lor y \lor \neg z)\]

No independent set in this graph can choose any \(x\) and \(\neg x\) node at the same time!
From 3SAT to INDSET

\[ (x \lor y \lor z) \land (\neg x \lor \neg y \lor z) \land (\neg x \lor y \lor \neg z) \land (\neg z \lor x \lor y) \land (\neg z \lor x \lor y) \]
From 3SAT to INDSET

\((x \lor y \lor z) \land (\neg x \lor \neg y \lor z) \land (\neg x \lor y \lor \neg z)\)
From 3SAT to INDSET

\[(x \lor y \lor z) \land (\neg x \lor \neg y \lor z) \land (\neg x \lor y \lor \neg z)\]
From 3SAT to INDSET

\[(x \lor y \lor z) \land (\neg x \lor \neg y \lor z) \land (\neg x \lor y \lor \neg z)\]
From 3SAT to INDSET

\[(x \lor y \lor z) \land (\neg y \lor \neg z \lor x) \land (\neg x \lor y \lor \neg z)\]
From 3SAT to INDSET

\[(x \lor y \lor z) \land (\neg y \lor \neg z) \land (\neg x \lor y \lor \neg z) \land (\neg x \lor \neg y \lor z)\]
From 3SAT to INDSET

\[
(x \lor y \lor \neg z) \land (\neg x \lor \neg y \lor z) \land (\neg x \lor y \lor \neg z)
\]

If this graph has an independent set of size three, the original formula is satisfiable.
From 3SAT to INDSET

\[(x \lor y \lor \neg z) \land (\neg x \lor \neg y \lor z) \land (\neg x \lor y \lor \neg z)\]

If this graph has an independent set of size three, the original formula is satisfiable.
If this graph has an independent set of size three, the original formula is satisfiable.
From 3SAT to INDSET

\[(x \lor y \lor \neg z) \land (\neg x \lor \neg y \lor z) \land (\neg x \lor y \lor \neg z)\]

\[x = \text{false}, \ y = \text{false}, \ z = \text{false}.

If this graph has an independent set of size three, the original formula is satisfiable.
From 3SAT to INDSET

\[(x \lor y \lor \neg z) \land (\neg x \lor \neg y \lor z) \land (\neg x \lor y \lor \neg z)\]

If this graph has an independent set of size three, the original formula is satisfiable.
From 3SAT to INDSET

If this graph has an independent set of size three, the original formula is satisfiable.
From 3SAT to INDSET

\[(x \lor y \lor \neg z) \land (\neg x \lor \neg y \lor z) \land (\neg x \lor y \lor \neg z)\]

If this graph has an independent set of size three, the original formula is satisfiable.

\[x = true, \ y = true, \ z = true.\]
From 3SAT to INDSET

If this graph has an independent set of size three, the original formula is satisfiable.
From 3SAT to INDSET

If this graph has an independent set of size three, the original formula is satisfiable.
From 3SAT to INDSET

\[(x \lor y \lor \neg z) \land (\neg x \lor \neg y \lor z) \land (\neg x \lor y \lor \neg z)\]

If this graph has an independent set of size three, the original formula is satisfiable.

x = false, y = ??, z = false.
From 3SAT to INDSET

\[(x \lor y \lor \overline{z}) \land (\overline{x} \lor \overline{y} \lor z) \land (\overline{x} \lor y \lor \overline{z})\]

\[x = false, \ y = true, \ z = false.\]

If this graph has an independent set of size three, the original formula is satisfiable.
From 3SAT to INDSET

\[(x \lor y \lor \neg z) \land (\neg x \lor \neg y \lor z) \land (\neg x \lor y \lor \neg z)\]

\[
x = \text{false}, \quad y = \text{false}, \quad z = \text{false}.
\]

If this graph has an independent set of size three, the original formula is satisfiable.
From 3SAT to INDSET

\[(x \lor y \lor \neg z) \land (\neg x \lor \neg y \lor z) \land (\neg x \lor y \lor \neg z) \land (\neg y \lor \neg z \lor x)\]
From 3SAT to INDSET

\[(x \lor y \lor \neg z) \land (\neg x \lor \neg y \lor z) \land (\neg x \lor y \lor \neg z)\]

If the original formula is satisfiable, this graph has an independent set of size three.
From 3SAT to INDSET

If the original formula is satisfiable, this graph has an independent set of size three.

\[
(x \lor y \lor z) \land (\neg y \lor \neg z \lor x) \land (\neg x \lor y \lor \neg z) \land (\neg x \lor \neg y \lor z)
\]

$x = \text{false}, y = \text{true}, z = \text{false}$.
From 3SAT to INDSET

\[(x \lor y \lor \neg z) \land (\neg x \lor \neg y \lor z) \land (\neg x \lor y \lor \neg z)\]

\[
\begin{align*}
\text{If the original formula is satisfiable,} \\
\text{this graph has an independent set of size three.}
\end{align*}
\]

\[x = \text{false}, \ y = \text{true}, \ z = \text{false}.\]
From 3SAT to INDSET

\[
(x \lor y \lor \neg z) \land \neg x \lor \neg y \lor z \land \neg x \lor y \lor \neg z \land \neg x \lor \neg y \lor z
\]

\[x = \text{false}, \; y = \text{true}, \; z = \text{false}.\]

If the original formula is satisfiable, this graph has an independent set of size three.
From 3SAT to INDSET

If the original formula is satisfiable, this graph has an independent set of size three.

\[(x \lor y \lor \neg z) \land (\neg x \lor \neg y \lor z) \land (\neg x \lor y \lor \neg z)\]

\[x = \text{false}, \ y = \text{true}, \ z = \text{false}.\]
From 3SAT to INDSET

If the original formula is satisfiable, this graph has an independent set of size three.

\[ (x \lor y \lor \neg z) \land (\neg x \lor \neg y \lor z) \land (\neg x \lor y \lor \neg z) \]

\[
\begin{align*}
\text{If } x = \text{false, } y = \text{true, } z = \text{false.}
\end{align*}
\]
From 3SAT to INDSET

- Let \( \varphi = C_1 \land C_2 \land \ldots \land C_n \) be a 3-CNF formula.

- Construct the graph \( G \) as follows:
  - For each clause \( C_i = x_1 \lor x_2 \lor x_3 \), where \( x_1, x_2, \) and \( x_3 \) are literals, add three new nodes into \( G \) with edges connecting them.
  - For each pair of nodes \( v_i \) and \( \neg v_i \), where \( v_i \) is some variable, add an edge connecting \( v_i \) and \( \neg v_i \). (Note that there are multiple copies of these nodes)

- **Claim One:** This reduction can be computed in polynomial time.

- **Claim:** \( G \) has an independent set of size \( n \) iff \( \varphi \) is satisfiable.
A Polynomial-Time Reduction

**Lemma**: This reduction can be computed in polynomial time.
A Polynomial-Time Reduction

Lemma: This reduction can be computed in polynomial time.

Proof: Suppose that the original 3-CNF formula $\phi$ has $n$ clauses, each of which has three literals.
A Polynomial-Time Reduction

Lemma: This reduction can be computed in polynomial time.

Proof: Suppose that the original 3-CNF formula $\phi$ has $n$ clauses, each of which has three literals. Then we construct $3n$ nodes in our graph.
A Polynomial-Time Reduction

*Lemma:* This reduction can be computed in polynomial time.

*Proof:* Suppose that the original 3-CNF formula $\phi$ has $n$ clauses, each of which has three literals. Then we construct $3n$ nodes in our graph. Each clause contributes 3 edges, so there are $O(n)$ edges added from clauses.
A Polynomial-Time Reduction

**Lemma:** This reduction can be computed in polynomial time.

**Proof:** Suppose that the original 3-CNF formula \( \varphi \) has \( n \) clauses, each of which has three literals. Then we construct \( 3n \) nodes in our graph. Each clause contributes 3 edges, so there are \( O(n) \) edges added from clauses. For each pair of nodes representing opposite literals, we introduce one edge.
A Polynomial-Time Reduction

Lemma: This reduction can be computed in polynomial time.

Proof: Suppose that the original 3-CNF formula $\phi$ has $n$ clauses, each of which has three literals. Then we construct $3n$ nodes in our graph. Each clause contributes 3 edges, so there are $O(n)$ edges added from clauses. For each pair of nodes representing opposite literals, we introduce one edge. Since there are $O(n^2)$ pairs of literals, this introduces at most $O(n^2)$ new edges.
Lemma: This reduction can be computed in polynomial time.

Proof: Suppose that the original 3-CNF formula φ has n clauses, each of which has three literals. Then we construct 3n nodes in our graph. Each clause contributes 3 edges, so there are $O(n)$ edges added from clauses. For each pair of nodes representing opposite literals, we introduce one edge. Since there are $O(n^2)$ pairs of literals, this introduces at most $O(n^2)$ new edges. This gives a graph with $O(n)$ nodes and $O(n^2)$ edges.
**A Polynomial-Time Reduction**

*Lemma:* This reduction can be computed in polynomial time.

*Proof:* Suppose that the original 3-CNF formula $\phi$ has $n$ clauses, each of which has three literals. Then we construct $3n$ nodes in our graph. Each clause contributes 3 edges, so there are $O(n)$ edges added from clauses. For each pair of nodes representing opposite literals, we introduce one edge. Since there are $O(n^2)$ pairs of literals, this introduces at most $O(n^2)$ new edges. This gives a graph with $O(n)$ nodes and $O(n^2)$ edges. Each node and edge can be constructed in polynomial time, so overall this reduction can be computed in polynomial time.
A Polynomial-Time Reduction

Lemma: This reduction can be computed in polynomial time.

Proof: Suppose that the original 3-CNF formula $\varphi$ has $n$ clauses, each of which has three literals. Then we construct $3n$ nodes in our graph. Each clause contributes 3 edges, so there are $O(n)$ edges added from clauses. For each pair of nodes representing opposite literals, we introduce one edge. Since there are $O(n^2)$ pairs of literals, this introduces at most $O(n^2)$ new edges. This gives a graph with $O(n)$ nodes and $O(n^2)$ edges. Each node and edge can be constructed in polynomial time, so overall this reduction can be computed in polynomial time. ■
One Direction of Implication

Lemma: If the graph $G$ has an independent set of size $n$ (where $n$ is the number of clauses) in $\phi$, then $\phi$ is satisfiable.

Proof:
Suppose $G$ has an independent set of size $n$, call it $S$. No two nodes in $S$ can correspond to $v$ and $\neg v$ for any variable $v$, because there is an edge between all nodes with this property. Thus for each variable $v$, either there is a node in $S$ with label $v$, or there is a node in $S$ with label $\neg v$, or no node in $S$ has either label.

In the first case, set $v$ to true; in the second case, set $v$ to false; in the third case, choose a value for $v$ arbitrarily. We claim that this gives a satisfying assignment for $\phi$. To see this, we show that each clause $C_i$ in $\phi$ is satisfied. By construction, no two nodes in $S$ can come from nodes added by $C_i$, because each has an edge to the other. Since there are $n$ nodes and $n$ clauses, there must be some node in $S$ corresponding to some literal in each clause. If that node has the form $x$, then $C_i$ contains $x$, and since we set $x$ to true, $C_i$ is satisfied. If that node has the form $\neg x$, then $C_i$ contains $\neg x$, and since we set $x$ to false, $C_i$ is satisfied. Thus all clauses in $\phi$ are satisfied, so $\phi$ is satisfiable. ■
One Direction of Implication

Lemma: If the graph $G$ has an independent set of size $n$ (where $n$ is the number of clauses) in $\varphi$, then $\varphi$ is satisfiable.

Proof: Suppose $G$ has an independent set of size $n$, call it $S$. 

No two nodes in $S$ can correspond to $v$ and $\neg v$ for any variable $v$, because there is an edge between all nodes with this property. Thus for each variable $v$, either there is a node in $S$ with label $v$, or there is a node in $S$ with label $\neg v$, or no node in $S$ has either label.

In the first case, set $v$ to true; in the second case, set $v$ to false; in the third case, choose a value for $v$ arbitrarily. We claim that this gives a satisfying assignment for $\varphi$.

To see this, we show that each clause $C_i$ in $\varphi$ is satisfied. By construction, no two nodes in $S$ can come from nodes added by $C_i$, because each has an edge to the other. Since there are $n$ nodes and $n$ clauses, there must be some node in $S$ corresponding to some literal in each clause. If that node has the form $x$, then $C_i$ contains $x$, and since we set $x$ to true, $C_i$ is satisfied. If that node has the form $\neg x$, then $C_i$ contains $\neg x$, and since we set $x$ to false, $C_i$ is satisfied. Thus all clauses in $\varphi$ are satisfied, so $\varphi$ is satisfied by this assignment. ■
One Direction of Implication

Lemma: If the graph G has an independent set of size n (where n is the number of clauses) in φ, then φ is satisfiable.

Proof: Suppose G has an independent set of size n, call it S. No two nodes in S can correspond to v and ¬v for any variable v, because there is an edge between all nodes with this property.
One Direction of Implication

*Lemma:* If the graph G has an independent set of size n (where n is the number of clauses) in φ, then φ is satisfiable.

*Proof:* Suppose G has an independent set of size n, call if S. No two nodes in S can correspond to v and ¬v for any variable v, because there is an edge between all nodes with this property. Thus for each variable v, either there is a node in S with label v, or there is a node in S with label ¬v, or no node in S has either label.
One Direction of Implication

Lemma: If the graph $G$ has an independent set of size $n$ (where $n$ is the number of clauses) in $\varphi$, then $\varphi$ is satisfiable.

Proof: Suppose $G$ has an independent set of size $n$, call if $S$. No two nodes in $S$ can correspond to $v$ and $\neg v$ for any variable $v$, because there is an edge between all nodes with this property. Thus for each variable $v$, either there is a node in $S$ with label $v$, or there is a node in $S$ with label $\neg v$, or no node in $S$ has either label. In the first case, set $v$ to true; in the second case, set $v$ to false; in the third case, choose a value for $v$ arbitrarily.
One Direction of Implication

Lemma: If the graph $G$ has an independent set of size $n$ (where $n$ is the number of clauses) in $\phi$, then $\phi$ is satisfiable.

Proof: Suppose $G$ has an independent set of size $n$, call it $S$. No two nodes in $S$ can correspond to $v$ and $\neg v$ for any variable $v$, because there is an edge between all nodes with this property. Thus for each variable $v$, either there is a node in $S$ with label $v$, or there is a node in $S$ with label $\neg v$, or no node in $S$ has either label. In the first case, set $v$ to true; in the second case, set $v$ to false; in the third case, choose a value for $v$ arbitrarily. We claim that this gives a satisfying assignment for $\phi$. 
One Direction of Implication

Lemma: If the graph G has an independent set of size n (where n is the number of clauses) in φ, then φ is satisfiable.

Proof: Suppose G has an independent set of size n, call it S. No two nodes in S can correspond to v and ¬v for any variable v, because there is an edge between all nodes with this property. Thus for each variable v, either there is a node in S with label v, or there is a node in S with label ¬v, or no node in S has either label. In the first case, set v to true; in the second case, set v to false; in the third case, choose a value for v arbitrarily. We claim that this gives a satisfying assignment for φ.

To see this, we show that each clause C in φ is satisfied.
One Direction of Implication

Lemma: If the graph $G$ has an independent set of size $n$ (where $n$ is the number of clauses) in $\phi$, then $\phi$ is satisfiable.

Proof: Suppose $G$ has an independent set of size $n$, call it $S$. No two nodes in $S$ can correspond to $v$ and $\neg v$ for any variable $v$, because there is an edge between all nodes with this property. Thus for each variable $v$, either there is a node in $S$ with label $v$, or there is a node in $S$ with label $\neg v$, or no node in $S$ has either label. In the first case, set $v$ to true; in the second case, set $v$ to false; in the third case, choose a value for $v$ arbitrarily. We claim that this gives a satisfying assignment for $\phi$.

To see this, we show that each clause $C$ in $\phi$ is satisfied. By construction, no two nodes in $S$ can come from nodes added by $C$, because each has an edge to the other.
**One Direction of Implication**

*Lemma:* If the graph $G$ has an independent set of size $n$ (where $n$ is the number of clauses) in $\varphi$, then $\varphi$ is satisfiable.

*Proof:* Suppose $G$ has an independent set of size $n$, call it $S$. No two nodes in $S$ can correspond to $v$ and $\neg v$ for any variable $v$, because there is an edge between all nodes with this property. Thus for each variable $v$, either there is a node in $S$ with label $v$, or there is a node in $S$ with label $\neg v$, or no node in $S$ has either label. In the first case, set $v$ to true; in the second case, set $v$ to false; in the third case, choose a value for $v$ arbitrarily. We claim that this gives a satisfying assignment for $\varphi$.

To see this, we show that each clause $C$ in $\varphi$ is satisfied. By construction, no two nodes in $S$ can come from nodes added by $C$, because each has an edge to the other. Since there are $n$ nodes in $S$ and $n$ clauses in $\varphi$, for any clause in $\varphi$ some node corresponding to a literal from that clause is in $S$. 
One Direction of Implication

**Lemma:** If the graph $G$ has an independent set of size $n$ (where $n$ is the number of clauses) in $\phi$, then $\phi$ is satisfiable.

**Proof:** Suppose $G$ has an independent set of size $n$, call if $S$. No two nodes in $S$ can correspond to $v$ and $\neg v$ for any variable $v$, because there is an edge between all nodes with this property. Thus for each variable $v$, either there is a node in $S$ with label $v$, or there is a node in $S$ with label $\neg v$, or no node in $S$ has either label. In the first case, set $v$ to true; in the second case, set $v$ to false; in the third case, choose a value for $v$ arbitrarily. We claim that this gives a satisfying assignment for $\phi$.

To see this, we show that each clause $C$ in $\phi$ is satisfied. By construction, no two nodes in $S$ can come from nodes added by $C$, because each has an edge to the other. Since there are $n$ nodes in $S$ and $n$ clauses in $\phi$, for any clause in $\phi$ some node corresponding to a literal from that clause is in $S$. If that node has the form $x$, then $C$ contains $x$, and since we set $x$ to true, $C$ is satisfied.
One Direction of Implication

*Lemma:* If the graph $G$ has an independent set of size $n$ (where $n$ is the number of clauses) in $\phi$, then $\phi$ is satisfiable.

*Proof:* Suppose $G$ has an independent set of size $n$, call it $S$. No two nodes in $S$ can correspond to $v$ and $\neg v$ for any variable $v$, because there is an edge between all nodes with this property. Thus for each variable $v$, either there is a node in $S$ with label $v$, or there is a node in $S$ with label $\neg v$, or no node in $S$ has either label. In the first case, set $v$ to true; in the second case, set $v$ to false; in the third case, choose a value for $v$ arbitrarily. We claim that this gives a satisfying assignment for $\phi$.

To see this, we show that each clause $C$ in $\phi$ is satisfied. By construction, no two nodes in $S$ can come from nodes added by $C$, because each has an edge to the other. Since there are $n$ nodes in $S$ and $n$ clauses in $\phi$, for any clause in $\phi$ some node corresponding to a literal from that clause is in $S$. If that node has the form $x$, then $C$ contains $x$, and since we set $x$ to true, $C$ is satisfied. If that node has the form $\neg x$, then $C$ contains $\neg x$, and since we set $x$ to false, $C$ is satisfied.
One Direction of Implication

**Lemma:** If the graph $G$ has an independent set of size $n$ (where $n$ is the number of clauses) in $\phi$, then $\phi$ is satisfiable.

**Proof:** Suppose $G$ has an independent set of size $n$, call if $S$. No two nodes in $S$ can correspond to $v$ and $\neg v$ for any variable $v$, because there is an edge between all nodes with this property. Thus for each variable $v$, either there is a node in $S$ with label $v$, or there is a node in $S$ with label $\neg v$, or no node in $S$ has either label. In the first case, set $v$ to true; in the second case, set $v$ to false; in the third case, choose a value for $v$ arbitrarily. We claim that this gives a satisfying assignment for $\phi$.

To see this, we show that each clause $C$ in $\phi$ is satisfied. By construction, no two nodes in $S$ can come from nodes added by $C$, because each has an edge to the other. Since there are $n$ nodes in $S$ and $n$ clauses in $\phi$, for any clause in $\phi$ some node corresponding to a literal from that clause is in $S$. If that node has the form $x$, then $C$ contains $x$, and since we set $x$ to true, $C$ is satisfied. If that node has the form $\neg x$, then $C$ contains $\neg x$, and since we set $x$ to false, $C$ is satisfied. Thus all clauses in $\phi$ are satisfied, so $\phi$ is satisfied by this assignment.
One Direction of Implication

*Lemma:* If the graph $G$ has an independent set of size $n$ (where $n$ is the number of clauses) in $\phi$, then $\phi$ is satisfiable.

*Proof:* Suppose $G$ has an independent set of size $n$, call it $S$. No two nodes in $S$ can correspond to $v$ and $\neg v$ for any variable $v$, because there is an edge between all nodes with this property. Thus for each variable $v$, either there is a node in $S$ with label $v$, or there is a node in $S$ with label $\neg v$, or no node in $S$ has either label. In the first case, set $v$ to true; in the second case, set $v$ to false; in the third case, choose a value for $v$ arbitrarily. We claim that this gives a satisfying assignment for $\phi$.

To see this, we show that each clause $C$ in $\phi$ is satisfied. By construction, no two nodes in $S$ can come from nodes added by $C$, because each has an edge to the other. Since there are $n$ nodes in $S$ and $n$ clauses in $\phi$, for any clause in $\phi$ some node corresponding to a literal from that clause is in $S$. If that node has the form $x$, then $C$ contains $x$, and since we set $x$ to true, $C$ is satisfied. If that node has the form $\neg x$, then $C$ contains $\neg x$, and since we set $x$ to false, $C$ is satisfied. Thus all clauses in $\phi$ are satisfied, so $\phi$ is satisfied by this assignment. ■
The Other Direction

*Lemma:* If $\phi$ is satisfiable and has $n$ clauses, then $G$ has an independent set of size $n$. 
The Other Direction

**Lemma:** If $\varphi$ is satisfiable and has $n$ clauses, then $G$ has an independent set of size $n$.

**Proof:** Suppose that $\varphi$ is satisfiable and consider any satisfying assignment for it.
The Other Direction

Lemma: If $\varphi$ is satisfiable and has $n$ clauses, then $G$ has an independent set of size $n$.

Proof: Suppose that $\varphi$ is satisfiable and consider any satisfying assignment for it. Thus under that assignment, for each clause $C$, there is some literal that evaluates to true.
The Other Direction

Lemma: If $\phi$ is satisfiable and has $n$ clauses, then $G$ has an independent set of size $n$.

Proof: Suppose that $\phi$ is satisfiable and consider any satisfying assignment for it. Thus under that assignment, for each clause $C$, there is some literal that evaluates to true. For each clause $C$, choose some literal that evaluates to true and add the corresponding node in $G$ to a set $S$. Then $S$ has size $n$, since it contains one node per clause. We claim moreover that $S$ is an independent set in $G$. To see this, note that there are two types of edges in $G$: edges between nodes representing literals in the same clause, and edges between variables and their negations. No two nodes joined by edges within a clause are in $S$, because we explicitly picked one node per clause. Moreover, no two nodes joined by edges between opposite literals are in $S$, because in a satisfying assignment both of the two could not be true. Thus no nodes in $S$ are joined by edges, so $S$ is an independent set. $\blacksquare$
The Other Direction

Lemma: If $\varphi$ is satisfiable and has $n$ clauses, then $G$ has an independent set of size $n$.

Proof: Suppose that $\varphi$ is satisfiable and consider any satisfying assignment for it. Thus under that assignment, for each clause $C$, there is some literal that evaluates to true. For each clause $C$, choose some literal that evaluates to true and add the corresponding node in $G$ to a set $S$. Then $S$ has size $n$, since it contains one node per clause.
The Other Direction

Lemma: If \( \phi \) is satisfiable and has \( n \) clauses, then \( G \) has an independent set of size \( n \).

Proof: Suppose that \( \phi \) is satisfiable and consider any satisfying assignment for it. Thus under that assignment, for each clause \( C \), there is some literal that evaluates to true. For each clause \( C \), choose some literal that evaluates to true and add the corresponding node in \( G \) to a set \( S \). Then \( S \) has size \( n \), since it contains one node per clause. We claim moreover that \( S \) is an independent set in \( G \).
The Other Direction

**Lemma:** If $\varphi$ is satisfiable and has $n$ clauses, then $G$ has an independent set of size $n$.

**Proof:** Suppose that $\varphi$ is satisfiable and consider any satisfying assignment for it. Thus under that assignment, for each clause $C$, there is some literal that evaluates to true. For each clause $C$, choose some literal that evaluates to true and add the corresponding node in $G$ to a set $S$. Then $S$ has size $n$, since it contains one node per clause. We claim moreover that $S$ is an independent set in $G$. To see this, note that there are two types of edges in $G$: edges between nodes representing literals in the same clause, and edges between variables and their negations.
The Other Direction

Lemma: If $\phi$ is satisfiable and has $n$ clauses, then $G$ has an independent set of size $n$.

Proof: Suppose that $\phi$ is satisfiable and consider any satisfying assignment for it. Thus under that assignment, for each clause $C$, there is some literal that evaluates to true. For each clause $C$, choose some literal that evaluates to true and add the corresponding node in $G$ to a set $S$. Then $S$ has size $n$, since it contains one node per clause. We claim moreover that $S$ is an independent set in $G$. To see this, note that there are two types of edges in $G$: edges between nodes representing literals in the same clause, and edges between variables and their negations. No two nodes joined by edges within a clause are in $S$, because we explicitly picked one node per clause.
The Other Direction

Lemma: If $\phi$ is satisfiable and has $n$ clauses, then $G$ has an independent set of size $n$.

Proof: Suppose that $\phi$ is satisfiable and consider any satisfying assignment for it. Thus under that assignment, for each clause $C$, there is some literal that evaluates to true. For each clause $C$, choose some literal that evaluates to true and add the corresponding node in $G$ to a set $S$. Then $S$ has size $n$, since it contains one node per clause. We claim moreover that $S$ is an independent set in $G$. To see this, note that there are two types of edges in $G$: edges between nodes representing literals in the same clause, and edges between variables and their negations. No two nodes joined by edges within a clause are in $S$, because we explicitly picked one node per clause. Moreover, no two nodes joined by edges between opposite literals are in $S$, because in a satisfying assignment both of the two could not be true.
The Other Direction

**Lemma**: If \( \varphi \) is satisfiable and has \( n \) clauses, then \( G \) has an independent set of size \( n \).

**Proof**: Suppose that \( \varphi \) is satisfiable and consider any satisfying assignment for it. Thus under that assignment, for each clause \( C \), there is some literal that evaluates to true. For each clause \( C \), choose some literal that evaluates to true and add the corresponding node in \( G \) to a set \( S \). Then \( S \) has size \( n \), since it contains one node per clause. We claim moreover that \( S \) is an independent set in \( G \). To see this, note that there are two types of edges in \( G \): edges between nodes representing literals in the same clause, and edges between variables and their negations. No two nodes joined by edges within a clause are in \( S \), because we explicitly picked one node per clause. Moreover, no two nodes joined by edges between opposite literals are in \( S \), because in a satisfying assignment both of the two could not be true. Thus no nodes in \( S \) are joined by edges, so \( S \) is an independent set.
Lemma: If \( \phi \) is satisfiable and has \( n \) clauses, then \( G \) has an independent set of size \( n \).

Proof: Suppose that \( \phi \) is satisfiable and consider any satisfying assignment for it. Thus under that assignment, for each clause \( C \), there is some literal that evaluates to true. For each clause \( C \), choose some literal that evaluates to true and add the corresponding node in \( G \) to a set \( S \). Then \( S \) has size \( n \), since it contains one node per clause. We claim moreover that \( S \) is an independent set in \( G \). To see this, note that there are two types of edges in \( G \): edges between nodes representing literals in the same clause, and edges between variables and their negations. No two nodes joined by edges within a clause are in \( S \), because we explicitly picked one node per clause. Moreover, no two nodes joined by edges between opposite literals are in \( S \), because in a satisfying assignment both of the two could not be true. Thus no nodes in \( S \) are joined by edges, so \( S \) is an independent set. ■
Putting it All Together

*Theorem:* INDSET is \textbf{NP}-complete.
Putting it All Together

*Theorem:* INDSET is $\textbf{NP}$-complete.

*Proof:* We know that INDSET $\in \textbf{NP}$, because we constructed a polynomial-time verifier for it.
Putting it All Together

Theorem: INDSET is NP-complete.

Proof: We know that INDSET ∈ NP, because we constructed a polynomial-time verifier for it. So all we need to show is that every problem in NP is polynomial-time reducible to INDSET.
Putting it All Together

Theorem: INDSET is $\text{NP}$-complete.
Proof: We know that INDSET $\in \text{NP}$, because we constructed a polynomial-time verifier for it. So all we need to show is that every problem in $\text{NP}$ is polynomial-time reducible to INDSET. To do this, we use the polynomial-time reduction from 3SAT to INDSET that we just gave.
Putting it All Together

Theorem: INDSET is $\text{NP}$-complete.

Proof: We know that INDSET $\in \text{NP}$, because we constructed a polynomial-time verifier for it. So all we need to show is that every problem in $\text{NP}$ is polynomial-time reducible to INDSET. To do this, we use the polynomial-time reduction from 3SAT to INDSET that we just gave. As we proved, $\varphi \in 3\text{SAT}$ iff $\langle G, n \rangle \in \text{INDSET}$, and this reduction can be computed in polynomial time.
Putting it All Together

Theorem: INDSET is $\textbf{NP}$-complete.

Proof: We know that INDSET $\in \textbf{NP}$, because we constructed a polynomial-time verifier for it. So all we need to show is that every problem in $\textbf{NP}$ is polynomial-time reducible to INDSET. To do this, we use the polynomial-time reduction from 3SAT to INDSET that we just gave. As we proved, $\varphi \in 3\text{SAT}$ iff $\langle G, n \rangle \in \text{INDSET}$, and this reduction can be computed in polynomial time. Thus 3SAT is polynomial-time reducible to INDSET, so INDSET is $\textbf{NP}$-complete.
Putting it All Together

**Theorem:** INDSET is **NP**-complete.

**Proof:** We know that INDSET $\in$ **NP**, because we constructed a polynomial-time verifier for it. So all we need to show is that every problem in **NP** is polynomial-time reducible to INDSET. To do this, we use the polynomial-time reduction from 3SAT to INDSET that we just gave. As we proved, $\varphi \in$ 3SAT iff $\langle G, n \rangle \in$ INDSET, and this reduction can be computed in polynomial time. Thus 3SAT is polynomial-time reducible to INDSET, so INDSET is **NP**-complete. ■
A **k-clique** in an undirected graph is a set of nodes where each pair of nodes in the set has an edge between them.
A **k-clique** in an undirected graph is a set of nodes where each pair of nodes in the set has an edge between them.
A **k-clique** in an undirected graph is a set of nodes where each pair of nodes in the set has an edge between them.
A **k-clique** in an undirected graph is a set of nodes where each pair of nodes in the set has an edge between them.
A **k-clique** in an undirected graph is a set of nodes where each pair of nodes in the set has an edge between them.
The Clique Problem

- The **clique problem** is
  
  Given an undirected graph $G$ and a number $k$, does $G$ contain a $k$-clique?

- As a formal language:
  
  \[ \text{CLIQUE} = \{ \langle G, k \rangle \mid G \text{ is an undirected graph that contains a } k\text{-clique} \} \]

- We will show that CLIQUE is **NP-complete**.
To show that CLIQUE is $\text{NP}$-complete, we first need to show that CLIQUE $\in \text{NP}$.

Here is a polynomial-time NTM for CLIQUE:

$M =$ “On input $\langle G, k \rangle$: 

Nondeterministically guess $k$ nodes.

Deterministically check whether all $k$ nodes have edges to one another.

If so, accept; otherwise, reject.”
CLIQUE is **NP**-Complete

- To prove that CLIQUE is **NP**-complete, we need to show that any problem in **NP** is polynomial-time reducible to it.
- Rather than reducing from 3SAT, we'll reduce INDSET to CLIQUE.
- Since INDSET is **NP**-complete, this proves that CLIQUE is **NP**-complete as well.
This graph is the complement graph of the left-hand graph. It has the same nodes, but contains all edges missing from the original graph. For simplicity, we omit self-loops.
Any independent set in the original graph is a clique in the complement graph!
CLIQUE is **NP-Complete**

- **Proof sketch:**
  - Reduce INDSET to CLIQUE.
  - Given as input \( \langle G, k \rangle \), construct \( \langle G', k \rangle \), where \( G' \) is the complement graph of \( G \).
  - There is an independent set of size \( k \) in \( G \) iff there is an clique of size \( k \) in \( G' \).
  - Since we can construct the complement graph in polynomial time, this is a polynomial-time reduction from INDSET to CLIQUE.
A Genealogy of NP-Completeness

- The Cook-Levin theorem establishes that SAT (and 3SAT) are $\text{NP}$-complete.
- From SAT and 3SAT, we can prove that many other problems are $\text{NP}$-complete as well.
- We can visualize the chain of reductions to any problem as a tree.
A Genealogy of NP-Completeness

SAT

CNF-SAT

0/1 IP  3SAT

INDSET  3COLOR

CLIQUE  SETPACK  EXCOVER
A Genealogy of NP-Completeness

• As of now, there are *thousands* of problems known to be NP-complete.

• For a fun list, check the Wikipedia article “List of NP-complete problems.”

• A polynomial-time solution to *any* of these problems proves that $P = NP$.

• Proving that *any* of these problems admits no polynomial-time solution proves that $P \neq NP$.

• *Yet no one has done either of these yet!*
A More Complex Reduction
The Shape of a Reduction

- Polynomial-time reductions work by solving one problem with a solver for a different problem.
- Most problems in \textbf{NP} have different pieces that must be solved simultaneously.
- For example, in 3SAT:
  - Each clause must be made true,
  - but no literal and its complement may be picked.
Reductions and Gadgets

- Many reductions used to show \textbf{NP}-completeness work by using \textit{gadgets}.
- Each piece of the original problem is translated into a “gadget” that handles some particular detail of the problem.
- These gadgets are then connected together to solve the overall problem.
Gadgets in INDSET

\[(x \lor y \lor z) \land (y \lor x \lor z) \land (z \lor x \lor y)\]
Gadgets in INDSET

\[
(x \lor y \lor z) \land (y' \lor x' \lor z) \land (x' \lor y \lor z')
\]
Gadgets in INDSET

\[ (x \lor y \lor \neg z) \land (\neg x \lor \neg y \lor z) \land (\neg x \lor y \lor \neg z) \]
Gadgets in INDSET

\((x \lor y \lor \neg z) \land (\neg x \lor \neg y \lor z) \land (\neg x \lor y \lor \neg z)\)
Gadgets in INDSET

Each of these gadgets is designed to solve one part of the problem: ensuring each clause is satisfied.
Gadgets in INDSET

\((x \lor y \lor \neg z) \land (\neg y \lor \neg z \lor y) \land (\neg y \lor x \lor z)\)
Gadgets in INDSET

( x \lor y \lor \neg z ) \land ( \neg y \lor z \lor \neg x ) \land ( \neg x \lor y \lor \neg z )
Gadgets in INDSET

These connections ensure that the solutions to each gadget are linked to one another.
Gadgets in INDSET
Gadgets in INDSET

This schematic shows (roughly) how the gadgets are assembled. Many proofs or descriptions of reductions will use drawings like these.
A 3-coloring of a graph is a way of coloring its nodes one of three colors such that no two connected nodes have the same color.
A **3-coloring** of a graph is a way of coloring its nodes one of three colors such that no two connected nodes have the same color.
A 3-coloring of a graph is a way of coloring its nodes one of three colors such that no two connected nodes have the same color.
The 3-Coloring Problem

• The 3-coloring problem is

  Given an undirected graph G, is there a legal 3-coloring of its nodes?

• As a formal language:

  \[ 3COLOR = \{ \langle G \rangle \mid G \text{ is an undirected graph with a legal 3-coloring.} \}\]

• This problem is known to be NP-complete by a reduction from 3SAT.
3COLOR ∈ NP

• We can prove that 3COLOR ∈ NP by designing a polynomial-time nondeterministic TM for 3COLOR.

• M = “On input ⟨G⟩:
  • **Nondeterministically** guess an assignment of colors to the nodes.
  • **Deterministically** check whether it is a legal 3-coloring.
  • If so, accept; otherwise reject.”
A Note on Terminology.

• Although 3COLOR and 3SAT both have 3 in their names, the two are very different problems.
  • 3SAT means “there are three literals in every clause.” However, each literal can take on one of two different values.
  • 3COLOR means “each node can take on one of three colors.”

• Key difference:
  • In 3SAT variables have two options.
  • In 3COLOR nodes have three options.
Why Not Two Colors?

• It would seem that 2COLOR (whether a graph has a 2-coloring) would be a better fit.
  • Every variable has one of two values.
  • Every node has one of two colors.

• Interestingly, 2COLOR is known to be in P and is conjectured not to be \( \text{NP} \)-complete.
  • Though, if you can prove that it is, you've just won $1,000,000!
From 3SAT to 3COLOR

- In order to reduce 3SAT to 3COLOR, we need to somehow make a graph that is 3-colorable iff some 3-CNF formula $\varphi$ is satisfiable.

- **Idea**: Use a collection of gadgets to solve the problem.
  - Build a gadget to assign two of the colors the labels “true” and “false.”
  - Build a gadget to force each variable to be either true or false.
  - Build a series of gadgets to force those variable assignments to satisfy each clause.
Gadget One: Assigning Meanings
Gadget One: Assigning Meanings

These nodes must all have different colors.
Gadget One: Assigning Meanings

These nodes must all have different colors.

The color assigned to T will be interpreted as "true."
The color assigned to F will be interpreted as "false."
We do not associate any special meaning with O.
Gadget One: Assigning Meanings

These nodes must all have different colors.

The color assigned to T will be interpreted as “true.”
The color assigned to F will be interpreted as “false.”
We do not associate any special meaning with 0.
Gadget One: Assigning Meanings

These nodes must all have different colors.

The color assigned to T will be interpreted as “true.”
The color assigned to F will be interpreted as “false.”
We do not associate any special meaning with O.
Gadget Two: Forcing a Choice

\((x \lor y \lor z) \land (y \lor z) \land (x \lor y \lor z)\)
Gadget Two: Forcing a Choice

\[(x \lor y \lor z) \land (y \lor x \lor z) \land (x \lor y \lor z)\]
Gadget Two: Forcing a Choice

( x ∨ y ∨ z ) ∧ ( y ∨ x ∧ z ) ∧ ( x ∨ y ∨ z )
Gadget Two: Forcing a Choice

( x ∨ y ∨ z ) ∧ ( ¬x ∨ ¬y ∨ z ) ∧ ( ¬x ∨ y ∨ ¬z )

We need to ensure that no literal and its complement become true at the same time.
Gadget Two: Forcing a Choice

\[(x \lor y \lor z) \land (y \lor x \lor z) \land (x \lor y \lor z)\]
Gadget Two: Forcing a Choice

\(( x \lor y \lor z ) \land ( y \lor y \lor z ) \land ( x \lor y \lor z )\)
Gadget Two: Forcing a Choice

\[
( x \lor y \lor \neg z ) \land ( \neg x \lor \neg y \lor z ) \land ( \neg x \lor y \lor \neg z )
\]

These gadgets ensure that any variable and its negation don't both get assigned the "true" color.
Gadget Two: Forcing a Choice

\[(x \lor y \lor \neg z) \land (\neg x \lor \neg y \lor z) \land (\neg x \lor y \lor \neg z)\]
Gadget Two: Forcing a Choice

\[( x \lor y \lor z ) \land ( \neg x \lor y \lor z ) \land ( \neg x \lor y \lor \neg z ) \]

```
T  F
O
```

```
x  \neg x
y  \neg y
z  \neg z
```
Gadget Two: Forcing a Choice

\[(x \lor y \lor \neg z) \land (\neg x \lor \neg y \lor z) \land (\neg x \lor y \lor \neg z)\]
Gadget Two: Forcing a Choice

\[(x \lor y \lor \neg z) \land (\neg x \lor y \lor z) \land (\neg x \lor y \lor \neg z)\]
Gadget Two: Forcing a Choice

\((x \lor y \lor \neg z) \land (\neg x \lor y \lor z) \land (\neg x \lor y \lor \neg z)\)
Gadget Two: Forcing a Choice

\[(x \lor y \lor z) \land (\neg x \lor \neg y \lor z) \land (\neg x \lor y \lor \neg z)\]

We need to ensure that none of these nodes get colored with the “other” color!
Gadget Two: Forcing a Choice

\[(x \lor y \lor z) \land (x' \lor y' \lor z') \land (x' \lor y' \lor z')\]
Gadget Two: Forcing a Choice

\[(x \lor y \lor z) \land (\neg x \lor y \lor z) \land (\neg x \lor y \lor z)\]

\[\text{T} \quad \text{F} \quad \text{O}\]

\[\text{x} \quad \text{y} \quad \text{z}\]
Gadget Two: Forcing a Choice

\[( x \lor y \lor z ) \land ( \neg x \lor y \lor z ) \land ( \neg x \lor y \lor z ) \]

\[
\begin{align*}
\text{T} & \rightarrow O \\
\text{F} & \rightarrow O
\end{align*}
\]
Gadget Two: Forcing a Choice

\[(x \lor y \lor \neg z) \land (\neg x \lor \neg y \lor z) \land (\neg x \lor y \lor \neg z)\]
Gadget Two: Forcing a Choice

\(( x \lor y \lor z ) \land ( \neg x \lor y \lor z ) \land ( \neg x \lor y \lor \neg z )\)
Gadget Two: Forcing a Choice

\[(x \lor y \lor \neg z) \land (\neg x \lor \neg y \lor z) \land (\neg x \lor y \lor \neg z)\]

The new edges force the literal nodes to be either true or false.
Gadget Three: Clause Satisfiability

\[(x \lor y \lor \neg z)\]
Gadget Three: Clause Satisfiability

\((x \lor y \lor \neg z)\)
Gadget Three: Clause Satisfiability

\((x \lor y \lor \neg z)\)

This node is colorable iff one of the inputs is the same color as T.
Gadget Three: Clause Satisfiability

\[(x \lor y \lor \neg z)\]
Gadget Three: Clause Satisfiability

\((x \lor y \lor \neg z)\)
Gadget Three: Clause Satisfiability

\((x \lor y \lor \neg z)\)
Gadget Three: Clause Satisfiability

$(x \lor y \lor \neg z)$
Gadget Three: Clause Satisfiability

\[(x \lor y \lor \neg z)\]

This node cannot be colored.
Gadget Three: Clause Satisfiability

\((x \lor y \lor \neg z)\)

\[\begin{array}{c}
x \\
T \\
y \\
F \\
\neg z \\
\end{array}\]
Gadget Three: Clause Satisfiability

\[(x \lor y \lor \neg z)\]
Gadget Three: Clause Satisfiability

\(( x \lor y \lor \lnot z )\)
Gadget Three: Clause Satisfiability

\((x \lor y \lor \neg z)\)
Gadget Three: Clause Satisfiability

\(( x \lor y \lor \neg z )\)
Gadget Three: Clause Satisfiability

\[(x \lor y \lor \neg z)\]
Gadget Three: Clause Satisfiability

\[(x \lor y \lor \neg z)\]
Gadget Three: Clause Satisfiability

\[(x \lor y \lor \neg z)\]

Diagram:
```
      (x ∨ y ∨ ¬z)
     /     \      |
    /       \     y
   /         \    /
  T         F    ¬z
  /  \      /  \  
 x   T   y   F   ¬z
```
Gadget Three: Clause Satisfiability

\[(x \lor y \lor \neg z)\]

Every other combination of inputs can give this a color.
Putting It All Together

• Construct the first gadget so we have a consistent definition of true and false.

• For each variable \( v \):
  • Construct nodes \( v \) and \( \neg v \).
  • Add an edge between \( v \) and \( \neg v \).
  • Add an edge between \( v \) and \( O \) and between \( \neg v \) and \( O \).

• For each clause \( C \):
  • Construct the earlier gadget from \( C \) by adding in the extra nodes and edges.
Putting It All Together

\[ C_1 \quad C_2 \quad \ldots \quad C_n \]

\[ x_1 \quad \neg x_1 \quad \ldots \quad \ldots \quad x_k \quad \neg x_k \]
Analyzing the Reduction

- How large is the resulting graph?
- We have $O(1)$ nodes to give meaning to “true” and “false.”
- Each variable gives $O(1)$ nodes for its true and false values.
- Each clause gives $O(1)$ nodes for its colorability gadget.
- Collectively, if there are $n$ clauses, there are $O(n)$ variables.
- Total size of the graph is $O(n)$. 
Correctness: A Sketch

To prove that the reduction is correct, we need to show that:

- If the original formula is satisfiable, there is some 3-coloring of the graph.
- If the original formula is unsatisfiable, there is no 3-coloring of the graph.

To do this, we reason about how the gadgets connect to one another.
3SAT $\rightarrow$ 3COLOR

- **Lemma**: If the 3-CNF formula $\phi$ is satisfiable, the graph $G$ has a 3-coloring.

- **Proof Sketch**: Color the nodes as follows:
  - Assign the T, F, and O nodes each their own color.
  - For each variable $v$ in $\phi$, color node $v$ with the color for T if $v$ is true and color the node $\neg v$ with the color for T otherwise.
  - Color the negation of $v$ the opposite color.
  - Because $\phi$ is satisfiable, each clause has at least one true literal in it.
  - Thus each clause gadget can be legally 3-colored.
\[ \neg3\text{SAT} \rightarrow \neg3\text{COLOR} \]

- **Lemma**: If the 3-CNF formula \( \phi \) is unsatisfiable, the graph \( G \) has no 3-coloring.

- **Proof Sketch**:
  - Since \( \phi \) is unsatisfiable, any assignment must leave some clause unsatisfied.
  - Thus any assignment must have some clause contain only false literals.
  - In any legal coloring, \( T, F, \) and \( O \) must have different colors.
  - In any legal coloring, nodes \( v \) and \( \neg v \) must either be colored \( T \) or \( F \), and neither get the same color.
  - For any coloring of the variable nodes, some clause gadget must have all three inputs colored the \( F \) color (because that clause is unsatisfied in \( \phi \))
  - Thus (as we saw before) that clause gadget cannot be colored.
Summary: **NP-Completeness Proofs**

• To prove that $L$ is **NP-complete**:
  
  • Prove that $L \in \textbf{NP}$.  
  • Find a polynomial-time reduction from some **NP-complete** problem $L'$ to $L$.
    
    – Possibly use gadgets to build an instance of $L$ from an instance of $L'$
  
  • Prove that the reduction works in polynomial time.
  • Prove that $w \in L$ iff $f(w) \in L'$.  
  • Conclude that $L$ is **NP-complete**.
Next Time

• **More NP-Complete Problems**
  • What sorts of problems turn out to be NP-complete?
  • What do these proofs look like?

• **(ITA) Beyond P and NP**
  • The class co-NP.
  • Polynomial-time refuters.
  • Even harder problems.