Satisfiability problems: SAT, 3SAT, Circuit-SAT
1 Complexity classes
   - The class $\mathcal{NP}$
   - The class of $\mathcal{NP}$-complete problems

2 Satisfiability: a fundamental $\mathcal{NP}$-complete problem

3 The art of proving $\mathcal{NP}$-completeness
   - Circuit-SAT $\leq_P$ SAT
   - $\text{IS(D)} \leq_P 3\text{SAT}$
1 Complexity classes
   - The class \( \mathcal{NP} \)
   - The class of \( \mathcal{NP} \)-complete problems

2 Satisfiability: a fundamental \( \mathcal{NP} \)-complete problem

3 The art of proving \( \mathcal{NP} \)-completeness
   - Circuit-SAT \( \leq_P \) SAT
   - IS(D) \( \leq_P \) 3SAT
$X, Y$ are computational problems; $R$ is a polynomial time transformation from input $x$ to $y$ so that $x, y$ are equivalent.

We used reductions
- as a means to design efficient algorithms
- for arguing about the relative hardness of problems
An optimization problem $X$ may be transformed into a roughly equivalent problem with a yes/no answer, called the decision version $X(D)$ of the optimization problem, by

1. supplying a target value for the quantity to be optimized;
2. asking whether this value can be attained.

Examples:

- $\text{IS}(D)$: given a graph $G$ and an integer $k$, does $G$ have an independent set of size $k$?
- $\text{VC}(D)$: given a graph $G$ and an integer $k$, does $G$ have a vertex cover of size $k$?
Definition 1.

\( \mathcal{P} \) is the set of problems that can be \textbf{solved} by polynomial-time algorithms.

Beyond \( \mathcal{P} \)?
Definition 1.

$\mathcal{P}$ is the set of problems that can be solved by polynomial-time algorithms.

**Beyond $\mathcal{P}$?**

Problems like $\text{IS}(\mathcal{D})$ and $\text{VC}(\mathcal{D})$:

- No polynomial time algorithm has been found despite significant effort, so we don’t believe they are in $\mathcal{P}$.
- *Is there anything positive we can say about such problems?*
The class $\mathcal{NP}$

**Definition 2.**
An efficient certifier (or *verification algorithm*) $B$ for a problem $X(D)$ is a *polynomial-time* algorithm that

1. takes two input arguments, the instance $x$ and the *short* certificate $t$ (both encoded as binary strings)
2. there is a polynomial $p(\cdot)$ so that for every string $x$, we have $x \in X(D)$ if and only if there is a string $t$ such that $|t| \leq p(|x|)$ and $B(x, t) = \text{yes}$.

Note that existence of the certifier $B$ *does not* provide us with any efficient way to solve $X(D)$! *(why?)*

**Definition 3.**
We define $\mathcal{NP}$ to be the set of decision problems that have an efficient certifier.
Fact 4.

\[ \mathcal{P} \subseteq \mathcal{NP} \]

Proof.

Let \( \textbf{x(D)} \) be a problem in \( \mathcal{P} \).

- There is an efficient algorithm \( A \) that solves \( \textbf{x(D)} \), that is, \( A(x) = \text{yes} \) if and only if \( x \in \textbf{x(D)} \).

- To show that \( \textbf{x(D)} \in \mathcal{NP} \), we need exhibit an efficient certifier \( B \) that takes two inputs \( x \) and \( t \) and answers \text{yes} if and only if \( x \in \textbf{x(D)} \).

- The algorithm \( B \) that on inputs \( x, t \), simply discards \( t \) and simulates \( A(x) \) is such an efficient certifier.
$P$ vs $NP$

$P \quad = \quad NP \quad ?$
Arguably the biggest question in theoretical CS

We do not think so: finding a solution should be harder than checking one, especially for hard problems...
Why would $\mathcal{NP}$ contain more problems than $\mathcal{P}$?

- Intuitively, the **hardest** problems in $\mathcal{NP}$ are the **least likely** to belong to $\mathcal{P}$.

- How do we identify the hardest problems?
Why would $NP$ contain more problems than $P$?

- Intuitively, the hardest problems in $NP$ are the least likely to belong to $P$.

- How do we identify the hardest problems?

The notion of reduction is useful again.

**Definition 5 ($NP$-complete problems:).**

A problem $X(D)$ is $NP$-complete if

1. $X(D) \in NP$, and
2. for all $Y \in NP$, $Y \leq_P X$. 
Why would $\mathcal{NP}$ contain more problems than $\mathcal{P}$?

- Intuitively, the hardest problems in $\mathcal{NP}$ are the least likely to belong to $\mathcal{P}$.

- How do we identify the hardest problems?

The notion of reduction will be useful again.

**Definition 5 ($\mathcal{NP}$-complete problems).**

A problem $X(D)$ is $\mathcal{NP}$-complete if

1. $X(D) \in \mathcal{NP}$ and

2. for all $Y \in \mathcal{NP}$, $Y \leq_P X$.

**Fact 6.**

Suppose $X$ is $\mathcal{NP}$-complete. Then $X$ is solvable in polynomial time (i.e., $X \in \mathcal{P}$) if and only if $\mathcal{P} = \mathcal{NP}$. 
Why we should care whether a problem is $\mathsf{NP}$-complete

- If a problem is $\mathsf{NP}$-complete it is among the least likely to be in $\mathsf{P}$: it is in $\mathsf{P}$ if and only if $\mathsf{P} = \mathsf{NP}$.
Why we should care whether a problem is $NP$-complete

- If a problem is $NP$-complete it is among the least likely to be in $P$: it is in $P$ if and only if $P = NP$.
- Therefore, from an algorithmic perspective, we need to stop looking for efficient algorithms for the problem.
Why we should care whether a problem is $\mathcal{NP}$-complete

- If a problem is $\mathcal{NP}$-complete it is among the least likely to be in $\mathcal{P}$: it is in $\mathcal{P}$ if and only if $\mathcal{P} = \mathcal{NP}$.
- Therefore, from an algorithmic perspective, we need to stop looking for efficient algorithms for the problem.

Instead we have a number of options

1. **approximation algorithms**, that is, algorithms that return a solution within a provable guarantee from the optimal
2. exponential algorithms practical for **small instances**
3. work on interesting **special cases**
4. study the average performance of the algorithm
5. examine **heuristics** (algorithms that work well in practice, yet provide no theoretical guarantees regarding how close the solution they find is to the optimal one)
How do we show that a problem is $\mathcal{NP}$-complete?

Suppose we had an $\mathcal{NP}$-complete problem $X$.

To show that another problem $Y$ is $\mathcal{NP}$-complete, we only need show that

1. $Y \in \mathcal{NP}$ and
2. $X \leq_P Y$
How do we show that a problem is $\mathcal{NP}$-complete?

Suppose we had an $\mathcal{NP}$-complete problem $X$.

To show that another problem $Y$ is $\mathcal{NP}$-complete, we only need show that

1. $Y \in \mathcal{NP}$ and
2. $X \leq_p Y$

Why?

Fact 7 (Transitivity of reductions).

If $X \leq_p Y$ and $Y \leq_p Z$, then $X \leq_p Z$.

We know that for all $A \in \mathcal{NP}$, $A \leq_p X$. By Fact 15, $A \leq_p Y$. Hence $Y$ is $\mathcal{NP}$-complete.
How do we show that a problem is $\mathcal{NP}$-complete?

Suppose we had an $\mathcal{NP}$-complete problem $X$.

To show that another problem $Y$ is $\mathcal{NP}$-complete, we only need show that

1. $Y \in \mathcal{NP}$ and
2. $X \leq_P Y$

So, if we had a first $\mathcal{NP}$-complete problem $X$, discovering a new problem $Y$ in this class would require an easier kind of reduction: just reduce $X$ to $Y$ (instead of reducing ever problem in $\mathcal{NP}$ to $Y$!).
Suppose we had an $\mathcal{NP}$-complete problem $X$.

To show that another problem $Y$ is $\mathcal{NP}$-complete, we only need to show that

1. $Y \in \mathcal{NP}$ and
2. $X \leq_p Y$

The first $\mathcal{NP}$-complete problem

**Theorem 7 (Cook-Levin).**

*Circuit SAT is $\mathcal{NP}$-complete.*
Today

1. Complexity classes
   - The class $\mathcal{NP}$
   - The class of $\mathcal{NP}$-complete problems

2. *Satisfiability*: a fundamental $\mathcal{NP}$-complete problem

3. The art of proving $\mathcal{NP}$-completeness
   - Circuit-SAT $\leq_P$ SAT
   - $\text{IS}(D) \leq_P 3\text{SAT}$
Boolean logic

Syntax of Boolean expressions

- **Boolean variable** $x$: a variable that takes values from \{0, 1\} (equivalently, \{F, T\}, standing for **False**, **True**).
- Suppose you are given a set of $n$ boolean variables \{${x_1, x_2, \ldots, x_n}$\}.
- **Boolean connectives**: logical AND $\land$, logical OR $\lor$ and logical NOT $\neg$.
- **Boolean expression or Boolean formula**: boolean variables connected by boolean connectives.
- **Notational convention**: $\phi$ is a boolean formula.
A **boolean expression** may be any of the following

1. A boolean variable, e.g., $x_i$.

2. The **negation** of a Boolean expression $\phi$, denoted by $\neg \phi_1$ or $\overline{\phi_1}$.

3. The **disjunction** (logical OR) of two Boolean expressions in parentheses $(\phi_1 \lor \phi_2)$.

4. The **conjunction** (logical AND) of two Boolean expressions in parentheses $(\phi_1 \land \phi_2)$. 
Basic properties of Boolean expressions (associativity, commutativity, distribution laws)

1. \( \neg \neg \phi \equiv \phi \)
2. \((\phi_1 \lor \phi_2) \equiv (\phi_2 \lor \phi_1)\)
3. \((\phi_1 \land \phi_2) \equiv (\phi_2 \land \phi_1)\)
4. \(((\phi_1 \lor \phi_2) \lor \phi_3) \equiv (\phi_1 \lor (\phi_2 \lor \phi_3))\)
5. \(((\phi_1 \land \phi_2) \land \phi_3) \equiv (\phi_1 \land (\phi_2 \land \phi_3))\)
6. \(((\phi_1 \lor \phi_2) \land \phi_3) \equiv ((\phi_1 \lor \phi_3) \land (\phi_2 \land \phi_3))\)
7. \(((\phi_1 \land \phi_2) \lor \phi_3) \equiv ((\phi_1 \lor \phi_3) \land (\phi_2 \lor \phi_3))\)
8. \(\neg (\phi_1 \lor \phi_2) \equiv (\neg \phi_1 \land \neg \phi_2)\)
9. \(\neg (\phi_1 \land \phi_2) \equiv (\neg \phi_1 \lor \neg \phi_2)\)
10. \(\phi_1 \lor \phi_1 \equiv \phi_1\)
11. \(\phi_1 \land \phi_1 \equiv \phi_1\)
A literal $\ell_i$ is a variable or its negation.

**Definition 8.**

A Boolean formula $\phi$ is in CNF if it consists of conjunctions of clauses each of which is a disjunction of literals.

- In symbols, a formula $\phi$ with $m$ clauses is in CNF if
  \[ \phi = C_1 \land C_2 \land \ldots \land C_m \]
  and each clause $C_i$ is the disjunction of a number of literals
  \[ \ell_1 \lor \ell_2 \lor \ldots \lor \ell_k \]

- **Example:** $n = 3$, $m = 2$, $\phi = (x_1 \lor \overline{x}_2) \land (\overline{x}_1 \lor x_2 \lor x_3)$

**Remark:** we will henceforth work with formulas in CNF.
Semantics of boolean formulas

- Let \( X = \{x_1, \ldots, x_n\} \).

- A **truth assignment** for \( X \) is an assignment of truth values from \( \{0, 1\} \) to each \( x_i \).
  - So a truth assignment is a function \( v : X \rightarrow \{0, 1\} \).
  - It is implied that \( \overline{x_i} \) obtains value opposite from \( x_i \).
  - **Example:** \( X = \{x_1, x_2, x_3\} \)
    - Truth assignment for \( X \): \( x_1 = 1, x_2 = x_3 = 0 \)

- A truth assignment causes a boolean formula to receive a value from \( \{0, 1\} \).
  - **Example:** \( \phi = (x_1 \lor \overline{x_2}) \land (\overline{x_1} \lor x_2 \lor x_3) \)
    - The above truth assignment causes \( \phi \) to evaluate to 0.
A truth assignment satisfies a clause if it causes the clause to evaluate to 1.

Example: \( \phi = (x_1 \lor \overline{x_2}) \land (\overline{x_1} \lor x_2 \lor x_3) \)  
Then \( x_1 = x_2 = 1, x_3 = 0 \) satisfies both clauses in \( \phi \).

A truth assignment satisfies a formula in CNF if it satisfies every clause in the formula.

Example: \( x_1 = x_2 = 1, x_3 = 0 \) satisfies the above \( \phi \).  
But \( x_1 = 1, x_2 = x_3 = 0 \) does not satisfy \( \phi \).

A formula \( \phi \) is satisfiable if it has a satisfying truth assignment.

Example: the above \( \phi \) is satisfiable; a certificate of its satisfiability is the truth assignment \( x_1 = x_2 = 1, x_3 = 0 \).
Definition 9 (SAT).

Given a formula $\phi$ in CNF with $n$ variables and $m$ clauses, is $\phi$ satisfiable?
Satisfiability (SAT) and 3SAT

Definition 9 (SAT).

Given a formula \( \phi \) in CNF with \( n \) variables and \( m \) clauses, is \( \phi \) satisfiable?

A convenient (and not easier) variant of SAT requires that every clause consists of exactly three literals.

Definition 10 (3SAT).

Given a formula \( \phi \) in CNF with \( n \) variables and \( m \) clauses such that each clause has exactly 3 literals, is \( \phi \) satisfiable?

Are these problems hard?
Definition 9 (SAT).

Given a formula $\phi$ in CNF with $n$ variables and $m$ clauses, is $\phi$ satisfiable?

A convenient (and not easier) variant of SAT requires that every clause consists of exactly three literals.

Definition 10 (3SAT).

Given a formula $\phi$ in CNF with $n$ variables and $m$ clauses such that each clause has exactly 3 literals, is $\phi$ satisfiable?

Are these problems hard?

Theorem 11.

SAT, 3SAT are $\mathcal{NP}$-complete.
Today

1. Complexity classes
   - The class $NP$
   - The class of $NP$-complete problems

2. Satisfiability: a fundamental $NP$-complete problem

3. The art of proving $NP$-completeness
   - Circuit-$SAT \leq_P SAT$
   - $IS(D) \leq_P 3SAT$
A physical circuit consists of gates that perform logical AND, OR and NOT.

We will model such a circuit by a boolean combinatorial circuit which is a labelled DAG with:

- **Source nodes**: these are the inputs of the circuit and may be hardwired to 0 or 1, or labelled with some variable.
- **Intermediate nodes**: these correspond to the gates of the circuit and are labelled with $\land$ (AND), $\lor$ (OR) or $\lnot$ (NOT).
  - $\land$, $\lor$ gates have two incoming and one outgoing edge
  - $\lnot$ gates have one incoming and one outgoing edge
- **Sink node**: corresponds to the output of the circuit and has no outgoing edges.
Example circuit

A circuit $C$ with 2 hardwired source nodes, 3 variable inputs $y_1, y_2, y_3$ and 5 logical gates.
Evaluating a circuit:
- edges are wires that carry the value of their tail node;
- intermediate nodes perform their label operation on their incoming edges, pass the result along their outgoing edge;
- the value of the circuit is the value of its output node.

**Definition 12 (Circuit-SAT).**

Given a circuit $C$, is there an assignment of truth values to its inputs that causes the output to evaluate to 1?

It is easy to see that Circuit-SAT is in $\mathcal{NP}$. Cook and Levin showed that it is $\mathcal{NP}$-complete.
Lemma 13.

Circuit-SAT $\leq_P$ SAT

Intuitively, this reduction should not be too difficult: formulas and circuits are just different ways of representing boolean functions and translating from one to the other should be easy.

The following two boolean connectives are very useful.

1. $(\phi_1 \Rightarrow \phi_2)$ is a shorthand for $(\overline{\phi_1} \lor \phi_2)$.
   
   *Intuition:* if $\phi_1 = 1$, then $\phi_2 = 1$ too (o.w., $(\phi_1 \Rightarrow \phi_2) = 0$).

2. $(\phi_1 \Leftrightarrow \phi_2)$ is a shorthand for $(\overline{(\phi_1 \Rightarrow \phi_2)} \land \overline{(\phi_2 \Rightarrow \phi_1)}))$, which may be expanded to $(\phi_1 \lor \phi_2) \land (\phi_1 \lor \phi_2)$.
   
   *This clause evaluates to 1 if and only if $\phi_1 = \phi_2$.*
Consider an arbitrary instance of Circuit-SAT, that is, a circuit $C$ with source nodes, intermediate nodes and an output node.

For every node $v$ in $C$, we introduce to $\phi$

- a variable $x_v$ that encodes the truth value computed by node $v$ in $C$;
- clauses that ensure that $x_v$ takes on the same value as the output of node $v$ given its inputs.

Then any satisfying truth assignment for the circuit $C$ will imply that $\phi$ is satisfiable, while, if $\phi$ is satisfiable, setting the variable inputs of $C$ to the truth values of their corresponding variables in $\phi$ will result in $C$ computing an output with value 1.
$\phi$ is the conjunction of the following clauses

1. If $v$ is a source node corresponding to a variable input of the circuit $C$, we do not add any clause.
2. If $v$ is a source node hardwired to 0, add $(\overline{x_v})$.
3. If $v$ is a source node hardwired to 1, add $(x_v)$.
4. If $v$ is the output node, add $(x_v)$.
5. If $v$ is a node labelled by NOT and its input edge is from node $u$, add $(x_v \leftrightarrow \overline{x_u})$.
6. If $v$ is a node labelled by OR and its input edges are from nodes $u$ and $w$, add $(x_v \leftrightarrow (x_u \lor x_w))$.
7. If $v$ is a node labelled by AND and its input edges are from nodes $u$ and $w$, add $(x_v \leftrightarrow (x_u \land x_w))$. 
This completes our construction of the clauses of $\phi$.

For example, for the circuit in slide 34, we construct the following formula.

$$\phi = (\neg x_1) \land (x_2) \land (x_6 \iff (x_1 \land x_2)) \land (x_7 \iff (x_3 \lor x_4)) \land (x_8 \iff \neg x_5) \land (x_9 \iff (x_6 \lor x_7)) \land (x_{10} \iff (x_9 \land x_8)) \land (x_{10})$$

The construction is polynomial in the size of the input circuit (why?).

Moreover, every clause consists of at most three literals, once $\phi$ is in CNF (exercise).
Proof of equivalence

⇒ Let $T_C$ be a truth assignment to the variable inputs of $C$ that causes $C$ to evaluate to 1. Propagate $T_C$ to assign a truth value to every node $v$ in $C$. Define a truth assignment $T_\phi$ for $\phi$ as follows: $x_v$ takes on the truth value of $v$, for every node $v$ in $C$. Then $T_\phi$ satisfies $\phi$.

⇐ Suppose $\phi$ has a satisfying truth assignment. Then the truth values of the variables of $\phi$ that correspond to inputs in $C$ satisfy $C$: the clauses in $\phi$ guarantee that, for every node in $C$, the value assigned to that node is exactly what that node computes in $C$. Since $\phi = 1$, $C$ evaluates to 1.
So far, we have stated (with or without proofs) that

- Circuit-SAT is $\mathcal{NP}$-complete
- Circuit-SAT $\leq_P$ SAT
- SAT $\leq_P$ 3SAT

$\Rightarrow$ SAT and 3SAT are $\mathcal{NP}$-complete.

Is IS(D) as “hard” as SAT?
So far, we have stated (with or without proofs) that

- Circuit-SAT is $\mathcal{NP}$-complete
- Circuit-SAT $\leq_P$ SAT
- SAT $\leq_P$ 3SAT

$\Rightarrow$ SAT and 3SAT are $\mathcal{NP}$-complete.

**Claim 1.**

**IS(D)** is $\mathcal{NP}$-complete.

**Proof.**

Reduction from 3SAT.
Structure of the proof

Given an arbitrary instance formula $\phi$ of 3SAT, we need to transform it into a graph $G$ and an integer $k$, so that

1. The transformation is completed in polynomial time.

2. The instance $(G, k)$ is a yes instance of $\text{IS}(D)$ if and only if $\phi$ is a yes instance of 3SAT.
Given an arbitrary instance formula $\phi$ of 3SAT, we need to transform it into a graph $G$ and an integer $k$, so that

1. The transformation is completed in polynomial time.

2. $G$ has an independent set of size at least $k$ if and only if $\phi$ is satisfiable

**Example:** given

$$\phi = (x_1 \lor x_2 \lor x_3) \land (\neg x_1 \lor x_2 \lor \neg x_3) \land (x_1 \lor \neg x_2 \lor \neg x_3)$$

construct

$$(G, k)$$
Given an arbitrary instance formula $\phi$ of 3SAT, we need to transform it into a graph $G$ and an integer $k$, so that

1. The transformation is completed in polynomial time.

2. $G$ has an independent set of size at least $k$ if and only if $\phi$ is satisfiable.

Remark 1.

- Heart of reduction $X \leq_P Y$: understand why some small instance of $Y$ makes it difficult.
- For $\text{IS}(D)$, such an instance is a triangle: it’s not clear which of its vertices to add to our independent set.
When reducing from \textbf{3SAT}, we often use \textit{gadgets}. Gadgets are constructions that ensure:

1. \textbf{Consistency of truth values in a truth assignment}: once $x_i$ is assigned a truth value, we must henceforth consistently use it under this truth value.

2. \textbf{Clause constraints}: since $\phi$ is in CNF, we must provide a way to satisfy \textit{every} clause. Equivalently, we must exhibit at least one literal that is set to 1 in every clause.

In effect, these gadgets will allow us to derive a \textit{valid} and \textit{satisfying} truth assignment for $\phi$ when the transformed instance is a \textbf{yes} instance of our problem, so we can prove equivalence of the two instances.
Clause constraint gadget: for every clause, introduce a triangle where a node is labelled by a literal in the clause.

Example: $\phi = (x_1 \lor x_2 \lor x_3) \land (\neg x_1 \lor x_2 \lor \neg x_3) \land (x_1 \lor \neg x_2 \lor \neg x_3)$

- Hence our graph $G$ consists of $m$ isolated triangles.
- The max independent set in this graph has size $m$: pick one vertex from every triangle. So we will set $k = m$.

Goal: derive a truth assignment from our independent set $S$.
Idea: when a node from a triangle is added to $S$, set the corresponding literal to 1.
Consistency gadgets

2. Is this truth assignment consistent?
   - Suppose $x_1$ was picked from the first triangle.
   - Can still pick $\overline{x_1}$ from the second triangle!
   - But then we are setting $x_1$ to both 1 and 0.
   $\Rightarrow$ This is obviously not a valid truth assignment!

Consistency of truth assignment: must ensure that we cannot add a node labelled $x_i$ and a node labelled $\overline{x_i}$ to our independent set.
2. Is this truth assignment consistent?
   ▶ Suppose $x_1$ was picked from the first triangle.
   ▶ Can still pick $\overline{x_1}$ from the second triangle!
   ▶ But then we are setting $x_1$ to both 1 and 0.
   ⇒ This is obviously not a valid truth assignment!

Consistency of truth assignment: must ensure that we cannot add a node labelled $x_i$ and a node labelled $\overline{x_i}$ to our independent set.

Consistency gadget: add edges between all occurrences of $x_i$ and $\overline{x_i}$, for every $i$, in $G$. 
Example: given the formula $\phi$ below ($n=m=3$)

$$\phi = (x_1 \lor x_2 \lor x_3) \land (\neg x_1 \lor x_2 \lor \neg x_3) \land (x_1 \lor \neg x_2 \lor \neg x_3),$$

the derived graph $G$ is as follows:

Set $k=m=3$; the input instance $R(\phi)$ to $\text{IS(D)}$ is $(G, 3)$.

**Remark:** the construction requires time polynomial in the size of $\phi$. 

"Constructed instance $(G, k)$ of $\text{IS(D)}$"
Proof of equivalence

We need to show that

\( \phi \) is satisfiable

if and only if

\( G \) has an independent set of size at least \( m \)
Proof of equivalence, reverse direction

- Suppose that $G$ has an independent set $S$ of size $m$.
- Then **every** triangle contributes one node to $S$.
- Define the following **truth assignment**
  - Set the literal corresponding to that node to 1.
  - Any variables left unset by this assignment may be set to 0 or 1 arbitrarily.

\[
\phi = (x_1 \lor x_2 \lor x_3) \land (\neg x_1 \lor x_2 \lor \neg x_3) \land (x_1 \lor \neg x_2 \lor \neg x_3)
\]

Independent set $S = \{x_1, x_2, x_3\}$

Derived truth assignment: $x_1=1$, $x_2=1$, $x_3=0$
Proof of equivalence, reverse direction

- Suppose that $G$ has an independent set $S$ of size $m$.
- Then every triangle contributes one node to $S$.
- Define the following truth assignment
  - Set the literal corresponding to that node to 1.
  - Any variables left unset by this assignment may be set to 0 or 1 arbitrarily.

We need to show that this truth assignment

1. is valid
2. satisfies $\phi$
Proof of equivalence, reverse direction

- Suppose that $G$ has an independent set $S$ of size $m$.
- Then every triangle contributes one node to $S$.
- Define the following truth assignment
  - Set the literal corresponding to that node to 1.
  - Any variables left unset by this assignment may be set to 0 or 1 arbitrarily.

We need to show that this truth assignment

1. is valid: by construction, $x_i, \overline{x_i}$ cannot both appear in $S$.
2. satisfies $\phi$: since every triangle contributes one node to $S$, every clause has a true literal, thus every clause is satisfied.
Proof of equivalence, forward direction

- Now suppose there is a satisfying truth assignment for $\phi$.
- Then there is (at least) one True literal in every clause.
- Construct an independent set $S$ as follows:
  From every triangle, add to $S$ a node labelled by such a literal; hence $S$ has size $m$.

We claim that $S$ thus constructed is indeed an independent set.
Proof of equivalence, forward direction

- Now suppose there is a **satisfying truth assignment** for $\phi$.
- Then there is (at least) one **True** literal in every clause.
- Construct an **independent set** $S$ as follows:
  From every triangle, add to $S$ a node labelled by such a literal; hence $S$ has size $m$.

We claim that $S$ thus constructed is indeed an **independent set**.

1. $S$ would not be an independent set *if* there was an edge between any two nodes in it.

2. Since all nodes in $S$ belong to *different* triangles, an edge implies that the two nodes are labelled by opposite literals.

3. Impossible: $S$ only contains **True** literals (so it cannot contain both a literal and its negation).