Consider the following algorithm which takes in an undirected graph \((G)\) and a vertex \(s\).

```plaintext
FindClique \((G, s)\)
    \(C = s\)
    for each vertex \(v \in V\)
        flag = 1
        for each vertex \(u \in C\)
            if \((u, v) \notin E\)
                flag = 0
        if flag == 1
            \(C = C \cup \{v\}\)
    return \(C\)
```

The algorithm represents a greedy algorithm which finds a clique depending on a start vertex \(s\).

- How fast is this algorithm?
ECE-374-B: Lecture 20 - P/NP and NP-completeness

Instructor: Abhishek Kumar Umrawal
April 09, 2024

University of Illinois at Urbana-Champaign
Consider the following algorithm which takes in an undirected graph \((G)\) and a vertex \(s\)

\[
\text{FindClique} \ (G, s) \\
C = s \\
\text{for each vertex } v \in V \\
\quad \text{flag} = 1 \\
\quad \text{for each vertex } u \in C \\
\quad \quad \text{if } (u, v) \notin E \\
\quad \quad \quad \text{flag} = 0 \\
\quad \text{if } \text{flag} == 1 \\
\quad \quad C = C \cup \{v\} \\
\text{return } C
\]

The algorithm is a greedy algorithm which finds a clique depending on a start vertex \(s\).

- How fast is this algorithm?
Consider the following algorithm which takes in a undirected graph \((G)\) and a vertex \(s\)

\[
\textbf{FindClique} \ (G, s) \\
C = s \\
\text{for each vertex } v \in V \\
\quad \text{flag} = 1 \\
\quad \text{for each vertex } u \in C \\
\quad \quad \text{if } (u, v) \notin E \\
\quad \quad \quad \text{flag} = 0 \\
\quad \text{if flag == 1} \\
\quad \quad C = C \cup \{v\} \\
\text{return } C
\]

The Clique-problem is NP-complete. But this algorithm provides us with the maximal clique containing \(s\). If we run it \(|V|\) times, does that solve the clique-problem.
Consider the following algorithm which takes in a undirected graph \((G)\) and a vertex \(s\)

\[\text{FindClique} \ (G, s)\]

\[
\begin{align*}
C &= s \\
\text{for each vertex } v \in V & \\
\quad \text{flag} &= 1 \\
\text{for each vertex } u \in C & \\
\quad \text{if } (u, v) \notin E & \\
\quad \quad \text{flag} &= 0 \\
\quad \text{if } \text{flag} == 1 & \\
\quad C &= C \cup \{v\} \\
\text{return } C
\end{align*}
\]
The Satisfiability Problem (SAT)
Propositional Formulas

Definition
Consider a set of boolean variables \( x_1, x_2, \ldots, x_n \).

- A **literal** is either a boolean variable \( x_i \) or its negation \( \neg x_i \).
- A **clause** is a disjunction of literals.
  For example, \( x_1 \lor x_2 \lor \neg x_4 \) is a clause.
- A **formula in conjunctive normal form (CNF)** is propositional formula which is a conjunction of clauses.
  - \(( x_1 \lor x_2 \lor \neg x_4 ) \land ( x_2 \lor \neg x_3 ) \land x_5 \) is a CNF formula.
Definition
Consider a set of boolean variables $x_1, x_2, \ldots, x_n$.

- A **literal** is either a boolean variable $x_i$ or its negation $\neg x_i$.
- A **clause** is a disjunction of literals.
  
  For example, $x_1 \lor x_2 \lor \neg x_4$ is a clause.

- A formula in conjunctive normal form (CNF) is propositional formula which is a conjunction of clauses.
  
  - $(x_1 \lor x_2 \lor \neg x_4) \land (x_2 \lor \neg x_3) \land x_5$ is a CNF formula.

- A formula $\varphi$ is a **3CNF**: 
  
  A CNF formula such that every clause has *exactly* 3 literals.
  
  - $(x_1 \lor x_2 \lor \neg x_4) \land (x_2 \lor \neg x_3 \lor x_1)$ is a 3CNF formula, but $(x_1 \lor x_2 \lor \neg x_4) \land (x_2 \lor \neg x_3) \land x_5$ is not.
Every boolean formula $f : \{0, 1\}^n \rightarrow \{0, 1\}$ can be written as a CNF formula.

<table>
<thead>
<tr>
<th>$x_1$</th>
<th>$x_2$</th>
<th>$x_3$</th>
<th>$x_4$</th>
<th>$x_5$</th>
<th>$x_6$</th>
<th>$f(x_1, x_2, \ldots, x_6)$</th>
<th>$\bar{x}_1 \lor x_2 \bar{x}_3 \lor x_4 \lor \bar{x}_5 \lor x_6$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>$f(0, \ldots, 0, 0)$</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>$f(0, \ldots, 0, 1)$</td>
<td>1</td>
</tr>
<tr>
<td>\vdots</td>
<td>\vdots</td>
<td>\vdots</td>
<td>\vdots</td>
<td>\vdots</td>
<td>\vdots</td>
<td>\vdots</td>
<td>\vdots</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>$f(0, \ldots, 1, 0)$</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>\vdots</td>
<td>\vdots</td>
<td>\vdots</td>
<td>\vdots</td>
<td>\vdots</td>
<td>\vdots</td>
<td>\vdots</td>
<td>\vdots</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>$f(1, \ldots, 1)$</td>
<td>1</td>
</tr>
</tbody>
</table>

How? For every row such that $f$ is zero, compute corresponding CNF clause. Then take the AND ($\land$) of all the CNF clauses computed. The resulting CNF formula is equivalent to $f$. 
Problem: **SAT**

**Instance:** A CNF formula $\varphi$.

**Question:** Is there a truth assignment to the variable of $\varphi$ such that $\varphi$ evaluates to true?

Problem: **3SAT**

**Instance:** A 3CNF formula $\varphi$.

**Question:** Is there a truth assignment to the variable of $\varphi$ such that $\varphi$ evaluates to true?
**Satisfiability**

**SAT**
Given a CNF formula $\varphi$, is there a truth assignment to variables such that $\varphi$ evaluates to true?

**Example**
- $(x_1 \lor x_2 \lor \neg x_4) \land (x_2 \lor \neg x_3) \land x_5$ is satisfiable; take $x_1, x_2, \ldots, x_5$ to be all true
- $(x_1 \lor \neg x_2) \land (\neg x_1 \lor x_2) \land (\neg x_1 \lor \neg x_2) \land (x_1 \lor x_2)$ is not satisfiable.

**3SAT**
Given a 3CNF formula $\varphi$, is there a truth assignment to variables such that $\varphi$ evaluates to true?
Importance of **SAT** and **3SAT**

- **SAT** and **3SAT** are basic constraint satisfaction problems.
- Many different problems can reduced to them because of the simple yet powerful expressively of logical constraints.
- Arise naturally in many applications involving hardware and software verification and correctness.
- As we will see, it is a fundamental problem in theory of NP-Completeness.
Given two bits $x, z$ which of the following SAT formulas is equivalent to the formula $z = \overline{x}$:

(A) $(\overline{z} \lor x) \land (z \lor \overline{x})$.
(B) $(z \lor x) \land (\overline{z} \lor \overline{x})$.
(C) $(\overline{z} \lor x) \land (\overline{z} \lor \overline{x}) \land (\overline{z} \lor \overline{x})$.
(D) $z \oplus x$.
(E) $(z \lor x) \land (\overline{z} \lor \overline{x}) \land (z \lor \overline{x}) \land (\overline{z} \lor x)$.

Answer: B
Given two bits $x, z$ which of the following SAT formulas is equivalent to the formula $z = \overline{x}$:

(A) $(\overline{z} \lor x) \land (z \lor \overline{x})$.
(B) $(z \lor x) \land (\overline{z} \lor \overline{x})$.
(C) $(\overline{z} \lor x) \land (\overline{z} \lor \overline{x}) \land (z \lor \overline{x})$.
(D) $z \oplus x$.
(E) $(z \lor x) \land (\overline{z} \lor \overline{x}) \land (z \lor \overline{x}) \land (\overline{z} \lor x)$.

<table>
<thead>
<tr>
<th>$x$</th>
<th>$y$</th>
<th>$z = \overline{x}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Given three bits $x, y, z$ which of the following SAT formulas is equivalent to the formula $z = x \land y$:

(A) $(\overline{z} \lor x \lor y) \land (z \lor \overline{x} \lor \overline{y})$.
(B) $(\overline{z} \lor x \lor y) \land (\overline{z} \lor \overline{x} \lor y) \land (z \lor \overline{x} \lor \overline{y})$.
(C) $(\overline{z} \lor x \lor y) \land (\overline{z} \lor \overline{x} \lor y) \land (z \lor \overline{x} \lor y) \land (z \lor \overline{x} \lor \overline{y})$.
(D) $(z \lor x \lor y) \land (\overline{z} \lor \overline{x} \lor y) \land (z \lor \overline{x} \lor y) \land (z \lor \overline{x} \lor \overline{y})$.
(E) $(z \lor x \lor y) \land (z \lor x \lor \overline{y}) \land (z \lor \overline{x} \lor y) \land (z \lor \overline{x} \lor \overline{y}) \land (\overline{z} \lor x \lor y) \land (\overline{z} \lor x \lor \overline{y}) \land (\overline{z} \lor \overline{x} \lor y) \land (\overline{z} \lor \overline{x} \lor \overline{y})$.

Answer: C
Given three bits $x, y, z$ which of the following SAT formulas is equivalent to the formula $z = x \land y$:

(A) $(\overline{z} \lor x \lor y) \land (z \lor \overline{x} \lor \overline{y})$.

(B) $(\overline{z} \lor x \lor y) \land (\overline{z} \lor \overline{x} \lor y) \land (z \lor \overline{x} \lor \overline{y})$.

(C) $(\overline{z} \lor x \lor y) \land (\overline{z} \lor \overline{x} \lor y) \land (z \lor \overline{x} \lor y) \land (z \lor \overline{x} \lor \overline{y})$.

(D) $(z \lor x \lor y) \land (\overline{z} \lor \overline{x} \lor y) \land (z \lor \overline{x} \lor y) \land (z \lor \overline{x} \lor \overline{y})$.

(E) $(z \lor x \lor y) \land (z \lor x \lor \overline{y}) \land (z \lor \overline{x} \lor y) \land (z \lor \overline{x} \lor \overline{y}) \land (\overline{z} \lor x \lor y) \land (\overline{z} \lor x \lor \overline{y}) \land (\overline{z} \lor \overline{x} \lor y) \land (\overline{z} \lor \overline{x} \lor \overline{y})$.
Reducing SAT to 3SAT
How SAT is different from 3SAT?
In SAT clauses might have arbitrary length: 1, 2, 3, \ldots variables:

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

In 3SAT every clause must have exactly 3 different literals.
How **SAT** is different from **3SAT**?

In **SAT** clauses might have arbitrary length: 1, 2, 3, ... variables:

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

In **3SAT** every clause must have exactly 3 different literals.

To reduce from an instance of **SAT** to an instance of **3SAT**, we must make all clauses to have exactly 3 variables...

**Basic idea**

- Pad short clauses so they have 3 literals.
- Break long clauses into shorter clauses.
- Repeat the above till we have a 3CNF.

Proof of this in Prof. Har-Peled’s async lectures!
Overview of Complexity Classes
In the beginning...
In the beginning...

Undecidable
In the beginning...

Undecidable

EXP

$EXP$
In the beginning...

Undecidable

EXP

PSPACE

PSPACE

EXP
In the beginning...

- Undecidable
- $\text{EXP}$
- $\text{PSPACE}$
- $\text{P}$
In the beginning...
In the beginning...

Undecidable

NP – Hard

PSPACE

P

NP

NP – Hard

co-NP

EXP
In the beginning...
In the beginning...

Undecidable

NP − Hard

NPC

NP

co-NP

PSPACE

EXP
Non-deterministic polynomial time - NP
P, NP and Turing Machines

- $P$: set of decision problems that have polynomial time (deterministic) algorithms, i.e. efficiently solvable using a (deterministic) Turing machine (DTM).
- $NP$: set of decision problems that have polynomial time non-deterministic algorithms, i.e. efficiently solvable using a non-deterministic Turing machine (NTM).
- Many natural problems we would like to solve are in $NP$.
- Every problem in $NP$ has an exponential time (deterministic) algorithm.
- $P \subseteq NP$.
- Some problems in $NP$ are in $P$ (e.g., shortest path problem).

**Big Question:** Does every problem in $NP$ have an efficient algorithm? Same as asking whether $P = NP$. 
Problems with no known deterministic polynomial time algorithms

Problems

• Independent Set
• Vertex Cover
• Set Cover
• SAT

There are of course undecidable problems (no algorithm at all!) but many problems that we want to solve are of similar flavor to the above.

**Question:** What is common to above problems?
Problems with no known deterministic polynomial time algorithms

Problems

- Independent Set
- Vertex Cover
- Set Cover
- SAT

There are of course undecidable problems (no algorithm at all!) but many problems that we want to solve are of similar flavor to the above.

**Question:** What is common to above problems?

They can all be solved via a non-deterministic computer in polynomial time!
Non-determinism in computing

Non-determinism is a special property of algorithms.

An algorithm that is capable of taking multiple states concurrently. Whenever it reaches a choice, it takes both paths.

If there is a path for the string to be accepted by the machine, then the string is part of the language.
Problems with no known deterministic polynomial time algorithms

Problems

- **Independent Set & Vertex Cover** - Can build algorithm to check all possible collection of vertices
- **Set Cover** - Can check all possible collection of sets
- **SAT** - Can build a non-deterministic algorithm that checks every possible boolean assignment.

But we don’t have access to a non-deterministic computer. So how can a deterministic computer verify that an algorithm is in NP?
Above problems share the following feature.

**Checkability**

For any YES instance $I_X$ of $X$ there is a proof/certificate/solution that is of length $\text{poly}(|I_X|)$ such that given a proof one can efficiently check that $I_X$ is indeed a YES instance.

Examples:

- SAT formula $\phi$: proof is a satisfying assignment.
- Independent Set in graph $G$ and $k$: a subset $S$ of vertices.
- Homework.
Above problems share the following feature.

**Checkability**

For any YES instance $I_X$ of $X$ there is a proof/certificate/solution that is of length $\text{poly}(|I_X|)$ such that given a proof one can efficiently check that $I_X$ is indeed a YES instance.

Examples:

- **SAT** formula $\varphi$: proof is a satisfying assignment.
- **Independent Set** in graph $G$ and $k$: a subset $S$ of vertices.
- **Homework**.
Definition
An algorithm $C(\cdot, \cdot)$ is a certifier for problem $X$ if the following two conditions hold.

- For every $s \in X$ there is some string $t$ such that $C(s, t) = \text{"yes"}$
- If $s \notin X$, $C(s, t) = \text{"no"}$ for every $t$.

The string $s$ is the problem instance. (Example: particular graph in independent set problem.) The string $t$ is called a certificate or proof for $s$. 
Definition (Efficient Certifier.)
A certifier $C$ is an efficient certifier for problem $X$ if there is a polynomial $p(\cdot)$ such that the following conditions hold.

- For every $s \in X$ there is some string $t$ such that $C(s, t) = \text{“yes” and } |t| \leq p(|s|)$.
- If $s \not\in X$, $C(s, t) = \text{“no” for every } t$.
- $C(\cdot, \cdot)$ runs in polynomial time.
Example: Independent Set

- **Problem:** Does $G = (V, E)$ have an independent set of size $\geq k$?
  - **Certificate:** Set $S \subseteq V$.
  - **Certifier:** Check $|S| \geq k$ and no pair of vertices in $S$ is connected by an edge.
Example: SAT

- **Problem:** Does formula $\varphi$ have a satisfying truth assignment?
  - **Certificate:** Assignment $a$ of 0/1 values to each variable.
  - **Certifier:** Check each clause under $a$ and say “yes” if all clauses are true.
Why is it called Non-deterministic Polynomial Time

A certifier is an algorithm $C(I, c)$ with the following two inputs.

- $I$: instance.
- $c$: proof/certificate that the instance is indeed a YES instance of the given problem.

One can think about $C$ as an algorithm for the original problem if the following hold.

- Given $I$, the algorithm guesses (non-deterministically, and who knows how) a certificate $c$.
- The algorithm now verifies the certificate $c$ for the instance $I$.

NP can be equivalently described using Turing machines.
Cook-Levin Theorem
Question
What is the hardest problem in NP? How do we define it?

Towards a definition

• Hardest problem must be in NP.
• Hardest problem must be at least as “difficult” as every other problem in NP.
NP-Complete Problems

**Definition**
A problem $X$ is said to be **NP-Complete** if

- $X \in NP$, and
- **(Hardness)** For any $Y \in NP$, $Y \leq_P X$.  


Lemma
Suppose $X$ is NP-Complete. Then $X$ can be solved in polynomial time if and only if $P = NP$.

Proof.

$\Rightarrow$ Suppose $X$ can be solved in polynomial time

- Let $Y \in NP$. We know $Y \leq_P X$.
- We showed that if $Y \leq_P X$ and $X$ can be solved in polynomial time, then $Y$ can be solved in polynomial time.
- Thus, every problem $Y \in NP$ is such that $Y \in P$; $NP \subseteq P$.
- Since $P \subseteq NP$, we have $P = NP$.

$\Leftarrow$ Since $P = NP$, and $X \in NP$, we have a polynomial time algorithm for $X$. \qed
**Definition**
A problem $Y$ is said to be **NP-Hard** if

- **(Hardness)** For any $X \in NP$, we have that $X \leq_{p} Y$.

An NP-Hard problem need not be in NP!

**Example:** Halting problem is NP-Hard (why?) but not NP-Complete.
Consequences of proving NP-Completeness

If X is NP-Complete

- Since we believe $P \neq NP$,
- and solving X implies $P = NP$.

X is unlikely to be efficiently solvable.

At the very least, many smart people before you have failed to find an efficient algorithm for X.
Consequences of proving NP-Completeness

If $X$ is NP-Complete

- Since we believe $P \neq NP$,
- and solving $X$ implies $P = NP$.

$X$ is unlikely to be efficiently solvable.

At the very least, many smart people before you have failed to find an efficient algorithm for $X$.

(This is proof by mob opinion — take with a grain of salt.)
NP-Complete Problems

**Question**
Are there any problems that are NP-Complete?

**Answer**
Yes! Many, many problems are NP-Complete.
Cook-Levin Theorem

Theorem (Cook-Levin)
SAT is NP-Complete.
Theorem (Cook-Levin)  
\(\text{SAT} \ is \ NP-Complete.\)

Need to show the following.

- \(\text{SAT}\) is in NP.
- Every NP problem \(X\) reduces to \(\text{SAT}\).

Steve Cook won the Turing award for his theorem.
To prove $X$ is NP-Complete, show the following.

- Show that $X$ is in NP.
- Give a polynomial-time reduction from a known NP-Complete problem such as SAT to $X$. 

Transitivity of reductions:

If $Y \leq_P SAT$ and $SAT \leq_P X$, then $Y \leq_P X$. Why?
To prove $X$ is NP-Complete, show the following.

- Show that $X$ is in NP.
- Give a polynomial-time reduction from a known NP-Complete problem such as SAT to $X$.

$\text{SAT} \leq_P X$ implies that every NP problem $Y \leq_P X$. Why?
Proving that a problem \( X \) is NP-Complete

To prove \( X \) is NP-Complete, show the following.

- Show that \( X \) is in NP.
- Give a polynomial-time reduction from a known NP-Complete problem such as \( SAT \) to \( X \).

\( SAT \leq_p X \) implies that every NP problem \( Y \leq_p X \). Why?

Transitivity of reductions:

\( Y \leq_p SAT \) and \( SAT \leq_p X \) and hence \( Y \leq_p X \).
3-SAT is NP-Complete

- 3-SAT is in NP.
- SAT $\leq^p$ 3-SAT as we saw.
NP-Completeness via Reductions

- **SAT** is NP-Complete due to Cook-Levin theorem.
- **SAT \( \leq_P 3\text{-SAT} \)**
- **3-SAT \( \leq_P \) Independent Set**
- **Independent Set \( \leq_P \) Vertex Cover**
- **Independent Set \( \leq_P \) Clique**
- **3-SAT \( \leq_P \) 3-Color**
- **3-SAT \( \leq_P \) Hamiltonian Cycle**
NP-Completeness via Reductions

- **SAT** is NP-Complete due to Cook-Levin theorem.
- **SAT** \(\leq_P 3\text{-SAT}\)
- **3-SAT** \(\leq_P\) **Independent Set**
- **Independent Set** \(\leq_P\) **Vertex Cover**
- **Independent Set** \(\leq_P\) **Clique**
- **3-SAT** \(\leq_P\) **3-Color**
- **3-SAT** \(\leq_P\) **Hamiltonian Cycle**

Hundreds and thousands of different problems from many areas of science and engineering have been shown to be NP-Complete.

A surprisingly frequent phenomenon!
Reducing 3-SAT to Independent Set
Problem: Independent Set

Instance: A graph $G$, integer $k$.

Question: Is there an independent set in $G$ of size $k$?
Problem: Independent Set

**Instance:** A graph $G$, integer $k$.

**Question:** Is there an independent set in $G$ of size $k$?
Problem: Independent Set

Instance: A graph $G$, integer $k$.

Question: Is there an independent set in $G$ of size $k$?
There are two ways to think about 3SAT.

1. Find a way to assign 0/1 (false/true) to the variables such that the formula evaluates to true, that is each clause evaluates to true.
2. Pick a literal from each clause and find a truth assignment to make all of them true. You will fail if two of the literals you pick are in conflict, i.e., you pick $x_i$ and $\neg x_i$.

We will take the second view of 3SAT to construct the reduction.
1. $G_\varphi$ will have one vertex for each literal in a clause.
2. Connect the literals in a clause to form a triangle; the independent set will pick at most one vertex from each clause, which will correspond to the literal to be set to true.
3. Connect 2 vertices if they label complementary literals; this ensures that the literals corresponding to the independent set do not have a conflict.
4. Take $k$ to be the number of clauses.

**Figure 1:** Graph for

$$\varphi = (\neg x_1 \lor x_2 \lor x_3) \land (x_1 \lor \neg x_2 \lor x_3) \land (\neg x_1 \lor x_2 \lor x_4).$$
1. $G_\varphi$ will have one vertex for each literal in a clause.
2. Connect the literals in a clause to form a triangle; the independent set will pick at most one vertex from each clause, which will correspond to the literal to be set to true.
3. Connect 2 vertices if they label complementary literals; this ensures that the literals corresponding to the independent set do not have a conflict.
4. Take $k$ to be the number of clauses.

Figure 1: Graph for
$\varphi = (\neg x_1 \lor x_2 \lor x_3) \land (x_1 \lor \neg x_2 \lor x_3) \land (\neg x_1 \lor x_2 \lor x_4)$. 
The Reduction

1. $G_{\varphi}$ will have one vertex for each literal in a clause.
2. Connect the literals in a clause to form a triangle; the independent set will pick at most one vertex from each clause, which will correspond to the literal to be set to true.
3. Connect 2 vertices if they label complementary literals; this ensures that the literals corresponding to the independent set do not have a conflict.
4. Take $k$ to be the number of clauses.

![Graph for $\varphi = (\neg x_1 \lor x_2 \lor x_3) \land (x_1 \lor \neg x_2 \lor x_3) \land (\neg x_1 \lor x_2 \lor x_4)$.]

**Figure 1:** Graph for $\varphi = (\neg x_1 \lor x_2 \lor x_3) \land (x_1 \lor \neg x_2 \lor x_3) \land (\neg x_1 \lor x_2 \lor x_4)$. 36
The Reduction

1. $G_\varphi$ will have one vertex for each literal in a clause.
2. Connect the literals in a clause to form a triangle; the independent set will pick at most one vertex from each clause, which will correspond to the literal to be set to true.
3. Connect 2 vertices if they label complementary literals; this ensures that the literals corresponding to the independent set do not have a conflict.
4. Take $k$ to be the number of clauses.

Figure 1: Graph for

$$\varphi = (\neg x_1 \lor x_2 \lor x_3) \land (x_1 \lor \neg x_2 \lor x_3) \land (\neg x_1 \lor x_2 \lor x_4).$$
The Reduction

1. $G_\varphi$ will have one vertex for each literal in a clause.
2. Connect the literals in a clause to form a triangle; the independent set will pick at most one vertex from each clause, which will correspond to the literal to be set to true.
3. Connect 2 vertices if they label complementary literals; this ensures that the literals corresponding to the independent set do not have a conflict.
4. Take $k$ to be the number of clauses.

Figure 1: Graph for $\varphi = (\neg x_1 \lor x_2 \lor x_3) \land (x_1 \lor \neg x_2 \lor x_3) \land (\neg x_1 \lor x_2 \lor x_4)$. 
Correctness

**Lemma**

\( \varphi \) is satisfiable iff \( G_\varphi \) has an independent set of size \( k \) (\( = \) number of clauses in \( \varphi \)).

**Proof.**

\[ \Rightarrow \text{ Let } a \text{ be the truth assignment satisfying } \varphi. \]

- Pick one of the vertices, corresponding to true literals under \( a \), from each triangle. This is an independent set of the appropriate size. Why? \( \square \)
Lemma

\( \varphi \) is satisfiable iff \( G_\varphi \) has an independent set of size \( k \) (= number of clauses in \( \varphi \)).

Proof.

\( \Leftarrow \) Let \( S \) be an independent set of size \( k \).

- \( S \) must contain exactly one vertex from each clause triangle.
- \( S \) cannot contain vertices labeled by conflicting literals.
- Thus, it is possible to obtain a truth assignment that makes in the literals in \( S \) true; such an assignment satisfies one literal in every clause.
Other NP-Complete problems
Graph Coloring
Problem: **Graph Coloring**

**Instance:** \( G = (V, E) \): Undirected graph, integer \( k \).

**Question:** Can the vertices of the graph be colored using \( k \) colors so that vertices connected by an edge do not get the same color?
Problem: 3 Coloring

**Instance:** $G = (V, E)$: Undirected graph.

**Question:** Can the vertices of the graph be colored using 3 colors so that vertices connected by an edge do not get the same color?
Problem: 3 Coloring

**Instance:** $G = (V, E)$: Undirected graph.

**Question:** Can the vertices of the graph be colored using 3 colors so that vertices connected by an edge do not get the same color?
**Observation:** If $G$ is colored with $k$ colors then each color class (nodes of same color) form an independent set in $G$. Thus, $G$ can be partitioned into $k$ independent sets iff $G$ is $k$-colorable.

Graph 2-Coloring can be decided in polynomial time.

$G$ is 2-colorable iff $G$ is bipartite! There is a linear time algorithm to check if $G$ is bipartite using breadth first search.
Hamiltonian Cycle
**Input**  Given a directed graph $G = (V, E)$ with $n$ vertices

**Goal**  Does $G$ have a Hamiltonian cycle?

- A Hamiltonian cycle is a cycle in the graph that visits every vertex in $G$ exactly once.
Input  Given a directed graph $G = (V, E)$ with $n$ vertices

Goal  Does $G$ have a Hamiltonian cycle?

- A Hamiltonian cycle is a cycle in the graph that visits every vertex in $G$ exactly once.