算法设计与分析(2024年春季学期)

NP-Completeness

授课老师: 栗师
南京大学计算机科学与技术系
The topics we discussed so far are positive results: how to design efficient algorithms for solving a given problem.

NP-Completeness provides negative results: some problems cannot be solved efficiently.

Q: Why do we study negative results?
The topics we discussed so far are positive results: how to design efficient algorithms for solving a given problem.

NP-Completeness provides negative results: some problems cannot be solved efficiently.

Q: Why do we study negative results?

A given problem $X$ cannot be solved in polynomial time.

Without knowing it, you will have to keep trying to find polynomial time algorithm for solving $X$. All our efforts are doomed!
Efficient = Polynomial Time

- Polynomial time: $O(n^k)$ for any constant $k > 0$
- Example: $O(n), O(n^2), O(n^{2.5} \log n), O(n^{100})$
- Not polynomial time: $O(2^n), O(n^{\log n})$
Efficient = Polynomial Time

- Polynomial time: $O(n^k)$ for any constant $k > 0$
- Example: $O(n), O(n^2), O(n^{2.5} \log n), O(n^{100})$
- Not polynomial time: $O(2^n), O(n^{\log n})$
- Almost all algorithms we learnt so far run in polynomial time
Efficient = Polynomial Time

- Polynomial time: $O(n^k)$ for any constant $k > 0$
- Example: $O(n), O(n^2), O(n^{2.5} \log n), O(n^{100})$
- Not polynomial time: $O(2^n), O(n^{\log n})$
- Almost all algorithms we learnt so far run in polynomial time

Reason for Efficient = Polynomial Time

- For natural problems, if there is an $O(n^k)$-time algorithm, then $k$ is small, say 4
- A good cut separating problems: for most natural problems, either we have a polynomial time algorithm, or the best algorithm runs in time $\Omega(2^{nc})$ for some $c$
- Do not need to worry about the computational model
Outline

1 Some Hard Problems

2 P, NP and Co-NP

3 Polynomial Time Reductions and NP-Completeness

4 NP-Complete Problems

5 Dealing with NP-Hard Problems

6 Summary
**Example: Hamiltonian Cycle Problem**

**Def.** Let $G$ be an undirected graph. A Hamiltonian Cycle (HC) of $G$ is a cycle $C$ in $G$ that passes each vertex of $G$ exactly once.

**Hamiltonian Cycle (HC) Problem**

**Input:** graph $G = (V, E)$

**Output:** whether $G$ contains a Hamiltonian cycle
**Def.** Let $G$ be an undirected graph. A Hamiltonian Cycle (HC) of $G$ is a cycle $C$ in $G$ that passes each vertex of $G$ exactly once.

**Hamiltonian Cycle (HC) Problem**

**Input:** graph $G = (V, E)$

**Output:** whether $G$ contains a Hamiltonian cycle
Example: Hamiltonian Cycle Problem

- The graph is called the **Petersen Graph**. It has no HC.
### Hamiltonian Cycle (HC) Problem

**Input:** graph $G = (V, E)$

**Output:** whether $G$ contains a Hamiltonian cycle
Example: Hamiltonian Cycle Problem

Hamiltonian Cycle (HC) Problem

**Input:** graph $G = (V, E)$

**Output:** whether $G$ contains a Hamiltonian cycle

Algorithm for Hamiltonian Cycle Problem:

- Enumerate all possible permutations, and check if it corresponds to a Hamiltonian Cycle
Hamiltonian Cycle (HC) Problem

**Input:** graph $G = (V, E)$

**Output:** whether $G$ contains a Hamiltonian cycle

Algorithm for Hamiltonian Cycle Problem:

- Enumerate all possible permutations, and check if it corresponds to a Hamiltonian Cycle
- Running time: $O(n!m) = 2^{O(n \lg n)}$
- Better algorithm: $2^{O(n)}$
- Far away from polynomial time
Example: Hamiltonian Cycle Problem

Hamiltonian Cycle (HC) Problem

**Input:** graph $G = (V, E)$

**Output:** whether $G$ contains a Hamiltonian cycle

Algorithm for Hamiltonian Cycle Problem:

- Enumerate all possible permutations, and check if it corresponds to a Hamiltonian Cycle
- Running time: $O(n!m) = 2^{O(n \lg n)}$
- Better algorithm: $2^{O(n)}$
- Far away from polynomial time
- HC is **NP-hard**: it is **unlikely** that it can be solved in polynomial time.
Maximum Independent Set Problem

**Def.** An independent set of $G = (V, E)$ is a subset $I \subseteq V$ such that no two vertices in $I$ are adjacent in $G$. 

![Graph representation of an independent set](image-url)
Maximum Independent Set Problem

**Def.** An independent set of $G = (V, E)$ is a subset $I \subseteq V$ such that no two vertices in $I$ are adjacent in $G$. 
Maximum Independent Set Problem

**Def.** An independent set of $G = (V, E)$ is a subset $I \subseteq V$ such that no two vertices in $I$ are adjacent in $G$.

Maximum Independent Set Problem

**Input:** graph $G = (V, E)$

**Output:** the size of the maximum independent set of $G$
**Def.** An independent set of $G = (V, E)$ is a subset $I \subseteq V$ such that no two vertices in $I$ are adjacent in $G$.

**Maximum Independent Set Problem**

**Input:** graph $G = (V, E)$

**Output:** the size of the maximum independent set of $G$

- Maximum Independent Set is NP-hard
Formula Satisfiability

**Input:** boolean formula with \(n\) variables, with \(\lor, \land, \neg\) operators.

**Output:** whether the boolean formula is satisfiable

- **Example:** \(\neg((\neg x_1 \land x_2) \lor (\neg x_1 \land \neg x_3) \lor x_1 \lor (\neg x_2 \land x_3))\) is not satisfiable

- **Trivial algorithm:** enumerate all possible assignments, and check if each assignment satisfies the formula. The algorithm runs in exponential time.
**Formula Satisfiability**

**Input:** boolean formula with \( n \) variables, with \( \lor, \land, \lnot \) operators.

**Output:** whether the boolean formula is satisfiable

- Example: \( \lnot((\lnot x_1 \land x_2) \lor (\lnot x_1 \land \lnot x_3) \lor x_1 \lor (\lnot x_2 \land x_3)) \) is not satisfiable

- Trivial algorithm: enumerate all possible assignments, and check if each assignment satisfies the formula. The algorithm runs in exponential time.

- Formula Satisfiability is NP-hard
Outline

1. Some Hard Problems
2. P, NP and Co-NP
3. Polynomial Time Reductions and NP-Completeness
4. NP-Complete Problems
5. Dealing with NP-Hard Problems
6. Summary
Def. A problem $X$ is called a **decision problem** if the output is either 0 or 1 (yes/no).
Def. A problem $X$ is called a decision problem if the output is either 0 or 1 (yes/no).

- When we define the P and NP, we only consider decision problems.
A problem $X$ is called a **decision problem** if the output is either 0 or 1 (yes/no).

When we define the P and NP, we only consider decision problems.

For each optimization problem $X$, there is a decision version $X'$ of the problem. If we have a polynomial time algorithm for the decision version $X'$, we can solve the original problem $X$ in polynomial time.
Optimization to Decision

Shortest Path

**Input:** graph $G = (V, E)$, weight $w$, $s$, $t$ and a bound $L$

**Output:** whether there is a path from $s$ to $t$ of length at most $L$
Optimization to Decision

**Shortest Path**

**Input:** graph $G = (V, E)$, weight $w$, $s$, $t$ and a bound $L$

**Output:** whether there is a path from $s$ to $t$ of length at most $L$

**Maximum Independent Set**

**Input:** a graph $G$ and a bound $k$

**Output:** whether there is an independent set of size at least $k$
The input of a problem will be encoded as a binary string.
The input of a problem will be encoded as a binary string.

Example: Sorting problem
Encoding

The input of a problem will be **encoded** as a binary string.

**Example: Sorting problem**

- Input: \((3, 6, 100, 9, 60)\)
The input of a problem will be encoded as a binary string.

Example: Sorting problem
- Input: (3, 6, 100, 9, 60)
- Binary: (11, 110, 1100100, 1001, 111100)
The input of a problem will be encoded as a binary string.

Example: Sorting problem

- Input: (3, 6, 100, 9, 60)
- Binary: (11, 110, 1100100, 1001, 111100)
- String:
The input of a problem will be encoded as a binary string.

Example: Sorting problem

- Input: (3, 6, 100, 9, 60)
- Binary: (11, 110, 1100100, 1001, 111100)
- String: 111101
The input of a problem will be **encoded** as a binary string.

**Example: Sorting problem**

- **Input:** (3, 6, 100, 9, 60)
- **Binary:** (11, 110, 1100100, 1001, 111100)
- **String:** 111101**11110001**
The input of a problem will be *encoded* as a binary string.

**Example: Sorting problem**

- **Input:** (3, 6, 100, 9, 60)
- **Binary:** (11, 110, 1100100, 1001, 111100)
- **String:** 11110111110001111000011000001
The input of a problem will be **encoded** as a binary string.

**Example: Sorting problem**

- **Input:** (3, 6, 100, 9, 60)
- **Binary:** (11, 110, 1100100, 1001, 111100)
- **String:** 111101111100001111110000110000001100001101
The input of a problem will be encoded as a binary string.

Example: Sorting problem

- **Input:** (3, 6, 100, 9, 60)
- **Binary:** (11, 110, 1100100, 1001, 111100)
- **String:** 1111011111000111110000011000001100001101111111100001
The input of an problem will be encoded as a binary string.
The input of an problem will be **encoded** as a binary string.

**Example: Interval Scheduling Problem**

The figure illustrates the intervals for the problem. Each interval is represented by a horizontal bar, and the order of intervals may be decoded into a binary string for further processing.
Encoding

The input of a problem will be encoded as a binary string.

Example: Interval Scheduling Problem

Encode the sequence into a binary string as before

\[(0, 3, 0, 4, 2, 4, 3, 5, 4, 6, 4, 7, 5, 8, 7, 9, 8, 9)\]
Encoding

The input of a problem will be encoded as a binary string.

Example: Interval Scheduling Problem

- (0, 3, 0, 4, 2, 4, 3, 5, 4, 6, 4, 7, 5, 8, 7, 9, 8, 9)
- Encode the sequence into a binary string as before
Def. The size of an input is the length of the encoded string $s$ for the input, denoted as $|s|$.

Q: Does it matter how we encode the input instances?
**Def.** The size of an input is the length of the encoded string $s$ for the input, denoted as $|s|$. 

**Q:** Does it matter how we encode the input instances? 

**A:** No! As long as we are using a “natural” encoding. We only care whether the running time is polynomial or not.
Define Problem as a Function

\[ X : \{0, 1\}^* \rightarrow \{0, 1\} \]

**Def.** A decision problem \( X \) is a function mapping \( \{0, 1\}^* \) to \( \{0, 1\} \) such that for any \( s \in \{0, 1\}^* \), \( X(s) \) is the correct output for input \( s \).

- \( \{0, 1\}^* \): the set of all binary strings of any length.
Def. A decision problem $X$ is a function mapping $\{0, 1\}^*$ to $\{0, 1\}$ such that for any $s \in \{0, 1\}^*$, $X(s)$ is the correct output for input $s$.

- $\{0, 1\}^*$: the set of all binary strings of any length.

Def. An algorithm $A$ solves a problem $X$ if, $A(s) = X(s)$ for any binary string $s$. 
Def. A decision problem $X$ is a function mapping $\{0,1\}^*$ to $\{0,1\}$ such that for any $s \in \{0,1\}^*$, $X(s)$ is the correct output for input $s$.

- $\{0,1\}^*$: the set of all binary strings of any length.

Def. An algorithm $A$ solves a problem $X$ if, $A(s) = X(s)$ for any binary string $s$.

Def. $A$ has a polynomial running time if there is a polynomial function $p(\cdot)$ so that for every string $s$, the algorithm $A$ terminates on $s$ in at most $p(|s|)$ steps.
**Def.** The *complexity class P* is the set of decision problems $X$ that can be solved in polynomial time.
Def. The *complexity class P* is the set of decision problems $X$ that can be solved in polynomial time.

- The decision versions of interval scheduling, shortest path and minimum spanning tree all in P.
Certifier for Hamiltonian Cycle (HC)

- Alice has a supercomputer, fast enough to run the $2^{O(n)}$ time algorithm for HC

Q: Given a graph $G = (V, E)$ with a HC, how can Alice convince Bob that $G$ contains a Hamiltonian cycle?

A: Alice gives a Hamiltonian cycle to Bob, and Bob checks if it is really a Hamiltonian cycle of $G$.

Def. The message Alice sends to Bob is called a certificate, and the algorithm Bob runs is called a certifier.
Certifier for Hamiltonian Cycle (HC)

- Alice has a supercomputer, fast enough to run the $2^{O(n)}$ time algorithm for HC
- Bob has a slow computer, which can only run an $O(n^3)$-time algorithm

Q: Given a graph $G = (V, E)$ with a HC, how can Alice convince Bob that $G$ contains a Hamiltonian cycle?

A: Alice gives a Hamiltonian cycle to Bob, and Bob checks if it is really a Hamiltonian cycle of $G$.

Def. The message Alice sends to Bob is called a certificate, and the algorithm Bob runs is called a certifier.
Certifier for Hamiltonian Cycle (HC)

- Alice has a supercomputer, fast enough to run the $2^O(n)$ time algorithm for HC
- Bob has a slow computer, which can only run an $O(n^3)$-time algorithm

Q: Given a graph $G = (V, E)$ with a HC, how can Alice convince Bob that $G$ contains a Hamiltonian cycle?
Alice has a supercomputer, fast enough to run the $2^{O(n)}$ time algorithm for HC

Bob has a slow computer, which can only run an $O(n^3)$-time algorithm

Q: Given a graph $G = (V, E)$ with a HC, how can Alice convince Bob that $G$ contains a Hamiltonian cycle?

A: Alice gives a Hamiltonian cycle to Bob, and Bob checks if it is really a Hamiltonian cycle of $G$
Certifier for Hamiltonian Cycle (HC)

- Alice has a supercomputer, fast enough to run the $2^O(n)$ time algorithm for HC
- Bob has a slow computer, which can only run an $O(n^3)$-time algorithm

Q: Given a graph $G = (V, E)$ with a HC, how can Alice convince Bob that $G$ contains a Hamiltonian cycle?

A: Alice gives a Hamiltonian cycle to Bob, and Bob checks if it is really a Hamiltonian cycle of $G$

Def. The message Alice sends to Bob is called a certificate, and the algorithm Bob runs is called a certifier.
Certifier for Independent Set (Ind-Set)

- Alice has a supercomputer, fast enough to run the $2^{O(n)}$ time algorithm for Ind-Set
- Bob has a slow computer, which can only run an $O(n^3)$-time algorithm

Given graph $G = (V, E)$ and integer $k$, such that there is an independent set of size $k$ in $G$, how can Alice convince Bob that there is such a set?

A: Alice gives a set of size $k$ to Bob and Bob checks if it is really an independent set in $G$.

Certificate: a set of size $k$

Certifier: check if the given set is really an independent set
Certifier for Independent Set (Ind-Set)

- Alice has a supercomputer, fast enough to run the $2^{O(n)}$ time algorithm for Ind-Set
- Bob has a slow computer, which can only run an $O(n^3)$-time algorithm

Q: Given graph $G = (V, E)$ and integer $k$, such that there is an independent set of size $k$ in $G$, how can Alice convince Bob that there is such a set?
**Certifier for Independent Set (Ind-Set)**

- Alice has a supercomputer, fast enough to run the $2^{O(n)}$ time algorithm for Ind-Set
- Bob has a slow computer, which can only run an $O(n^3)$-time algorithm

**Q:** Given graph $G = (V, E)$ and integer $k$, such that there is an independent set of size $k$ in $G$, how can Alice convince Bob that there is such a set?

**A:** Alice gives a set of size $k$ to Bob and Bob checks if it is really a independent set in $G$. 
Alice has a supercomputer, fast enough to run the $2^{O(n)}$ time algorithm for Ind-Set

Bob has a slow computer, which can only run an $O(n^3)$-time algorithm

**Q:** Given graph $G = (V, E)$ and integer $k$, such that there is an independent set of size $k$ in $G$, how can Alice convince Bob that there is such a set?

**A:** Alice gives a set of size $k$ to Bob and Bob checks if it is really a independent set in $G$.

- Certificate: a set of size $k$
**Certifier for Independent Set (Ind-Set)**

- Alice has a supercomputer, fast enough to run the $2^{O(n)}$ time algorithm for Ind-Set
- Bob has a slow computer, which can only run an $O(n^3)$-time algorithm

Q: Given graph $G = (V, E)$ and integer $k$, such that there is an independent set of size $k$ in $G$, how can Alice convince Bob that there is such a set?

A: Alice gives a set of size $k$ to Bob and Bob checks if it is really a independent set in $G$.

- Certificate: a set of size $k$
- Certifier: check if the given set is really an independent set
The Complexity Class NP

Def. \( B \) is an efficient certifier for a problem \( X \) if

- \( B \) is a polynomial-time algorithm that takes two input strings \( s \) and \( t \), and outputs 0 or 1.
- there is a polynomial function \( p \) such that, \( X(s) = 1 \) if and only if there is string \( t \) such that \( |t| \leq p(|s|) \) and \( B(s, t) = 1 \).

The string \( t \) such that \( B(s, t) = 1 \) is called a certificate.
The Complexity Class NP

Def. $B$ is an efficient certifier for a problem $X$ if
- $B$ is a polynomial-time algorithm that takes two input strings $s$ and $t$, and outputs $0$ or $1$.
- there is a polynomial function $p$ such that, $X(s) = 1$ if and only if there is string $t$ such that $|t| \leq p(|s|)$ and $B(s, t) = 1$.

The string $t$ such that $B(s, t) = 1$ is called a certificate.

Def. The complexity class NP is the set of all problems for which there exists an efficient certifier.
HC (Hamiltonian Cycle) ∈ NP

- Input: Graph $G$

Clearly, $B$ runs in polynomial time.
HC (Hamiltonian Cycle) $\in$ NP

- Input: Graph $G$
- Certificate: a permutation $S$ of $V$ that forms a Hamiltonian Cycle
- $|\text{encoding}(S)| \leq p(|\text{encoding}(G)|)$ for some polynomial function $p$
HC (Hamiltonian Cycle) $\in$ NP

- Input: Graph $G$
- Certificate: a permutation $S$ of $V$ that forms a Hamiltonian Cycle
- $|\text{encoding}(S)| \leq p(|\text{encoding}(G)|)$ for some polynomial function $p$
- Certifier $B$: $B(G, S) = 1$ if and only if $S$ gives an HC in $G$
- Clearly, $B$ runs in polynomial time
HC (Hamiltonian Cycle) ∈ NP

- **Input:** Graph $G$
- **Certificate:** a permutation $S$ of $V$ that forms a Hamiltonian Cycle
- $|\text{encoding}(S)| \leq p(|\text{encoding}(G)|)$ for some polynomial function $p$
- **Certifier $B$:** $B(G, S) = 1$ if and only if $S$ gives an HC in $G$
- Clearly, $B$ runs in polynomial time
- $\text{HC}(G) = 1 \iff \exists S, B(G, S) = 1$
MIS (Maximum Independent Set) $\in$ NP

- Input: graph $G = (V, E)$ and integer $k$
MIS (Maximum Independent Set) $\in \text{NP}$

- **Input:** graph $G = (V, E)$ and integer $k$
- **Certificate:** a set $S \subseteq V$ of size $k$
- $|\text{encoding}(S)| \leq p(|\text{encoding}(G, k)|)$ for some polynomial function $p$

**Certifier** $B$:

$B((G, k), S) = 1$ if and only if $S$ is an independent set in $G$

Clearly, $B$ runs in polynomial time

$\text{MIS}(G, k) = 1 \iff \exists S, B((G, k), S) = 1$
MIS (Maximum Independent Set) $\in$ NP

- **Input:** graph $G = (V, E)$ and integer $k$
- **Certificate:** a set $S \subseteq V$ of size $k$
- $|\text{encoding}(S)| \leq p(|\text{encoding}(G, k)|)$ for some polynomial function $p$
- **Certifier $B$:** $B((G, k), S) = 1$ if and only if $S$ is an independent set in $G$
- Clearly, $B$ runs in polynomial time
MIS (Maximum Independent Set) $\in$ NP

- **Input:** graph $G = (V, E)$ and integer $k$
- **Certificate:** a set $S \subseteq V$ of size $k$
- $|\text{encoding}(S)| \leq p(|\text{encoding}(G, k)|)$ for some polynomial function $p$
- **Certifier** $B$: $B((G, k), S) = 1$ if and only if $S$ is an independent set in $G$
- Clearly, $B$ runs in polynomial time
- **MIS**$(G, k) = 1$ $\iff$ $\exists S, B((G, k), S) = 1$
Circuit Satisfiability (Circuit-Sat) Problem

**Input:** a circuit with and/or/not gates

**Output:** whether there is an assignment such that the output is 1?
Circuit Satisfiability (Circuit-Sat) Problem

**Input:** a circuit with and/or/not gates

**Output:** whether there is an assignment such that the output is 1?

Is Circuit-Sat \(\in\) NP?
\( \text{Input: graph } G = (V, E) \)

\( \text{Output: whether } G \text{ does not contain a Hamiltonian cycle} \)
\textbf{HC}

**Input:** graph $G = (V, E)$

**Output:** whether $G$ does not contain a Hamiltonian cycle

- Is $\overline{HC} \in \text{NP}$?
Input: graph $G = (V, E)$
Output: whether $G$ does not contain a Hamiltonian cycle

- Is $\overline{HC} \in \text{NP}$?
- Can Alice convince Bob that $G$ is a yes-instance (i.e., $G$ does not contain a HC), if this is true.
**Input:** graph \( G = (V, E) \)

**Output:** whether \( G \) does not contain a Hamiltonian cycle

- Is \( \overline{HC} \in \text{NP} \)?
- Can Alice convince Bob that \( G \) is a yes-instance (i.e., \( G \) does not contain a HC), if this is true.
- Unlikely
**HC**

**Input:** graph $G = (V, E)$

**Output:** whether $G$ does not contain a Hamiltonian cycle

- Is $\overline{HC} \in \text{NP}$?
- Can Alice convince Bob that $G$ is a yes-instance (i.e., $G$ does not contain a HC), if this is true?
- Unlikely

- Alice can only convince Bob that $G$ is a no-instance
Input: graph $G = (V, E)$

Output: whether $G$ does not contain a Hamiltonian cycle

- Is $\overline{HC} \in \text{NP}$?
- Can Alice convince Bob that $G$ is a yes-instance (i.e, $G$ does not contain a HC), if this is true.
- Unlikely
- Alice can only convince Bob that $G$ is a no-instance
- $\overline{HC} \in \text{Co-NP}$
The Complexity Class Co-NP

**Def.** For a problem $X$, the problem $\overline{X}$ is the problem such that $\overline{X}(s) = 1$ if and only if $X(s) = 0$.

**Def.** Co-NP is the set of decision problems $X$ such that $\overline{X} \in \text{NP}$. 
Def. A tautology is a boolean formula that always evaluates to 1.

Tautology Problem

Input: a boolean formula
Output: whether the formula is a tautology

- e.g. \((\neg x_1 \land x_2) \lor (\neg x_1 \land \neg x_3) \lor x_1 \lor (\neg x_2 \land x_3)\) is a tautology
Def. A tautology is a boolean formula that always evaluates to 1.

**Tautology Problem**

**Input:** a boolean formula  
**Output:** whether the formula is a tautology

- e.g. $(\neg x_1 \land x_2) \lor (\neg x_1 \land \neg x_3) \lor x_1 \lor (\neg x_2 \land x_3)$ is a tautology
- Bob can certify that a formula is not a tautology
**Def.** A **tautology** is a boolean formula that always evaluates to 1.

**Tautology Problem**

*Input:* a boolean formula  
*Output:* whether the formula is a tautology

- e.g. \((\neg x_1 \land x_2) \lor (\neg x_1 \land \neg x_3) \lor x_1 \lor (\neg x_2 \land x_3)\) is a tautology
- Bob can certify that a formula is not a tautology
- Thus Tautology ∈ Co-NP
Let $X \in P$ and $X(s) = 1$.

Q: How can Alice convince Bob that $s$ is a yes instance?

A: Since $X \in P$, Bob can check whether $X(s) = 1$ by himself, without Alice's help. Thus, $X \in NP$ and $P \subseteq NP$. Similarly, $P \subseteq \text{Co-NP}$, thus $P \subseteq \text{NP} \cap \text{Co-NP}$.
Let \( X \in \text{P} \) and \( X(s) = 1 \)

**Q:** How can Alice convince Bob that \( s \) is a yes instance?
Let $X \in P$ and $X(s) = 1$

**Q:** How can Alice convince Bob that $s$ is a yes instance?

**A:** Since $X \in P$, Bob can check whether $X(s) = 1$ by himself, without Alice’s help.
Let $X \in P$ and $X(s) = 1$

**Q:** How can Alice convince Bob that $s$ is a yes instance?

**A:** Since $X \in P$, Bob can check whether $X(s) = 1$ by himself, without Alice’s help.

- The certificate is an empty string
Let $X \in P$ and $X(s) = 1$

**Q:** How can Alice convince Bob that $s$ is a yes instance?

**A:** Since $X \in P$, Bob can check whether $X(s) = 1$ by himself, without Alice’s help.

- The certificate is an empty string
- Thus, $X \in NP$ and $P \subseteq NP$
Let $X \in P$ and $X(s) = 1$

Q: How can Alice convince Bob that $s$ is a yes instance?

A: Since $X \in P$, Bob can check whether $X(s) = 1$ by himself, without Alice’s help.

- The certificate is an empty string
- Thus, $X \in NP$ and $P \subseteq NP$
- Similarly, $P \subseteq Co-NP$, thus $P \subseteq NP \cap Co-NP$
Is \( P = NP \)?

A famous, big, and fundamental open problem in computer science. Little progress has been made. Most researchers believe \( P \neq NP \). It would be too amazing if \( P = NP \): if one can check a solution efficiently, then one can find a solution efficiently.

We assume \( P \neq NP \) and prove that problems do not have polynomial time algorithms.

We said it is unlikely that Hamiltonian Cycle can be solved in polynomial time: \( \text{if } P \neq NP, \text{ then } HC \notin P \), unless \( P = NP \).
Is $P = NP$?

- A famous, big, and fundamental open problem in computer science

- It would be too amazing if $P = NP$: if one can check a solution efficiently, then one can find a solution efficiently

- We assume $P \neq NP$ and prove that problems do not have polynomial time algorithms.

- We said it is unlikely that Hamiltonian Cycle can be solved in polynomial time: if $P \neq NP$, then $HC \in P$, unless $P = NP$
Is $P = NP$?

- A famous, big, and fundamental open problem in computer science
- Little progress has been made
- Most researchers believe $P \neq NP$
- It would be too amazing if $P = NP$: if one can check a solution efficiently, then one can find a solution efficiently
Is $P = NP$?

- A famous, big, and fundamental open problem in computer science
- Little progress has been made
- Most researchers believe $P \neq NP$
- It would be too amazing if $P = NP$: if one can check a solution efficiently, then one can find a solution efficiently
- We assume $P \neq NP$ and prove that problems do not have polynomial time algorithms.
Is $P = NP$?

- A famous, big, and fundamental open problem in computer science
- Little progress has been made
- Most researchers believe $P \neq NP$
- It would be too amazing if $P = NP$: if one can check a solution efficiently, then one can find a solution efficiently
- We assume $P \neq NP$ and prove that problems do not have polynomial time algorithms.
- We said it is unlikely that Hamiltonian Cycle can be solved in polynomial time:
  - if $P \neq NP$, then $HC \notin P$
  - $HC \notin P$, unless $P = NP$
Is $NP = Co-NP$?

- Again, a big open problem
Is $\text{NP} = \text{Co-NP}$?

- Again, a big open problem
- Most researchers believe $\text{NP} \neq \text{Co-NP}$. 
Notice that $X \in \text{NP} \iff \overline{X} \in \text{Co-NP}$ and $P \subseteq \text{NP} \cap \text{Co-NP}$

- People commonly believe we are in the 4th scenario
Outline

1. Some Hard Problems
2. P, NP and Co-NP
3. Polynomial Time Reductions and NP-Completeness
4. NP-Complete Problems
5. Dealing with NP-Hard Problems
6. Summary
Polynomial-Time Reductions

**Def.** Given a black box algorithm $A$ that solves a problem $X$, if any instance of a problem $Y$ can be solved using a polynomial number of standard computational steps, plus a polynomial number of calls to $A$, then we say $Y$ is polynomial-time reducible to $X$, denoted as $Y \leq_P X$. 
Def. Given a black box algorithm $A$ that solves a problem $X$, if any instance of a problem $Y$ can be solved using a polynomial number of standard computational steps, plus a polynomial number of calls to $A$, then we say $Y$ is polynomial-time reducible to $X$, denoted as $Y \leq_P X$.

To prove positive results:

Suppose $Y \leq_P X$. If $X$ can be solved in polynomial time, then $Y$ can be solved in polynomial time.

Polynomial-Time Reducations

**Def.** Given a black box algorithm $A$ that solves a problem $X$, if any instance of a problem $Y$ can be solved using a polynomial number of standard computational steps, plus a polynomial number of calls to $A$, then we say $Y$ is polynomial-time reducible to $X$, denoted as $Y \leq_P X$.

To prove positive results:

Suppose $Y \leq_P X$. If $X$ can be solved in polynomial time, then $Y$ can be solved in polynomial time.

To prove negative results:

Suppose $Y \leq_P X$. If $Y$ cannot be solved in polynomial time, then $X$ cannot be solved in polynomial time.
## Hamiltonian-Path (HP) problem

**Input:** \( G = (V, E) \) and \( s, t \in V \)

**Output:** whether there is a Hamiltonian path from \( s \) to \( t \) in \( G \)
Polynomial-Time Reduction: Example

Hamiltonian-Path (HP) problem

- **Input:** $G = (V, E)$ and $s, t \in V$
- **Output:** whether there is a Hamiltonian path from $s$ to $t$ in $G$

**Lemma** $\text{HP} \leq_p \text{HC}$.
Polynomial-Time Reduction: Example

Hamiltonian-Path (HP) problem

**Input:** $G = (V, E)$ and $s, t \in V$

**Output:** whether there is a Hamiltonian path from $s$ to $t$ in $G$

Lemma $\text{HP} \leq_P \text{HC}$.
Polynomial-Time Reduction: Example

Hamiltonian-Path (HP) problem

**Input:** $G = (V, E)$ and $s, t \in V$

**Output:** whether there is a Hamiltonian path from $s$ to $t$ in $G$

**Lemma** $\text{HP} \leq_p \text{HC.}$
Polynomial-Time Reduction: Example

Hamiltonian-Path (HP) problem

**Input:** $G = (V, E)$ and $s, t \in V$

**Output:** whether there is a Hamiltonian path from $s$ to $t$ in $G$

**Lemma** $\text{HP} \leq_P \text{HC}$.

**Obs.** $G$ has a HP from $s$ to $t$ if and only if graph on right side has a HC.
Def. A problem $X$ is called \textbf{NP-complete} if

1. $X \in \text{NP}$, and
2. $Y \leq^p X$ for every $Y \in \text{NP}$.
Def. A problem $X$ is called NP-hard if

2. $Y \leq_{P} X$ for every $Y \in NP$. 

NP-Completeness
Def. A problem $X$ is called **NP-complete** if

1. $X \in \text{NP}$, and
2. $Y \leq_P X$ for every $Y \in \text{NP}$.
NP-Completeness

**Def.** A problem $X$ is called **NP-complete** if

1. $X \in \text{NP}$, and
2. $Y \leq_{P} X$ for every $Y \in \text{NP}$.

**Theorem** If $X$ is NP-complete and $X \in \text{P}$, then $\text{P} = \text{NP}$. 
**NP-Completeness**

**Def.** A problem $X$ is called **NP-complete** if

1. $X \in \text{NP}$, and
2. $Y \leq_P X$ for every $Y \in \text{NP}$.

**Theorem** If $X$ is NP-complete and $X \in \text{P}$, then $\text{P} = \text{NP}$.

- NP-complete problems are the hardest problems in NP
NP-Completeness

**Def.** A problem $X$ is called **NP-complete** if

1. $X \in \text{NP}$, and
2. $Y \leq_P X$ for every $Y \in \text{NP}$.

**Theorem** If $X$ is NP-complete and $X \in \text{P}$, then $\text{P} = \text{NP}$.

- NP-complete problems are the hardest problems in NP
- NP-hard problems are at least as hard as NP-complete problems (a NP-hard problem is not required to be in NP)
Outline

1. Some Hard Problems
2. P, NP and Co-NP
3. Polynomial Time Reductions and NP-Completeness
4. NP-Complete Problems
5. Dealing with NP-Hard Problems
6. Summary
Def. A problem $X$ is called NP-complete if

1. $X \in \text{NP}$, and
2. $Y \leq_P X$ for every $Y \in \text{NP}$.
Def. A problem $X$ is called NP-complete if

1. $X \in \text{NP}$, and
2. $Y \leq_p X$ for every $Y \in \text{NP}$.

How can we find a problem $X \in \text{NP}$ such that every problem $Y \in \text{NP}$ is polynomial time reducible to $X$? Are we asking for too much?
Def. A problem $X$ is called NP-complete if

1. $X \in \text{NP}$, and
2. $Y \leq_p X$ for every $Y \in \text{NP}$.

- How can we find a problem $X \in \text{NP}$ such that every problem $Y \in \text{NP}$ is polynomial time reducible to $X$? Are we asking for too much?
- No! There is indeed a large family of natural NP-complete problems.
The First NP-Complete Problem: Circuit-Sat

Circuit Satisfiability (Circuit-Sat)

Input: a circuit
Output: whether the circuit is satisfiable
key fact: algorithms can be converted to circuits

**Fact** Any algorithm that takes $n$ bits as input and outputs 0/1 with running time $T(n)$ can be converted into a circuit of size $p(T(n))$ for some polynomial function $p(\cdot)$. 
Circuit-Sat is NP-Complete

- key fact: algorithms can be converted to circuits

**Fact** Any algorithm that takes \( n \) bits as input and outputs 0/1 with running time \( T(n) \) can be converted into a circuit of size \( p(T(n)) \) for some polynomial function \( p(\cdot) \).

- Then, we can show that any problem \( Y \in \text{NP} \) can be reduced to Circuit-Sat.
- We prove \( \text{HC} \leq_P \text{Circuit-Sat} \) as an example.
Let \( \text{check-HC}(G, S) \) be the certifier for the Hamiltonian cycle problem: \( \text{check-HC}(G, S) \) returns 1 if \( S \) is a Hamiltonian cycle in \( G \) and 0 otherwise.
HC \leq_p \text{Circuit-Sat}

\text{check-HC}(G, S)

- Let \text{check-HC}(G, S) be the certifier for the Hamiltonian cycle problem: \text{check-HC}(G, S) returns 1 if \(S\) is a Hamiltonian cycle in \(G\) and 0 otherwise.
- \(G\) is a yes-instance if and only if there is an \(S\) such that \text{check-HC}(G, S) returns 1.
HC \leq_p \text{Circuit-Sat}

Let check-HC(G, S) be the certifier for the Hamiltonian cycle problem: check-HC(G, S) returns 1 if S is a Hamiltonian cycle is G and 0 otherwise.

- G is a yes-instance if and only if there is an S such that check-HC(G, S) returns 1
- Construct a circuit $C'$ for the algorithm check-HC
HC \leq_P \text{Circuit-Sat}

Let check-HC(G, S) be the certifier for the Hamiltonian cycle problem: check-HC(G, S) returns 1 if S is a Hamiltonian cycle is G and 0 otherwise.

\( G \) is a yes-instance if and only if there is an \( S \) such that check-HC(G, S) returns 1

Construct a circuit \( C' \) for the algorithm check-HC

hard-wire the instance \( G \) to the circuit \( C' \) to obtain the circuit \( C \)
Let check-HC\((G, S)\) be the certifier for the Hamiltonian cycle problem: check-HC\((G, S)\) returns 1 if \(S\) is a Hamiltonian cycle is \(G\) and 0 otherwise.

\(G\) is a yes-instance if and only if there is an \(S\) such that check-HC\((G, S)\) returns 1

Construct a circuit \(C'\) for the algorithm check-HC

hard-wire the instance \(G\) to the circuit \(C'\) to obtain the circuit \(C\)

\(G\) is a yes-instance if and only if \(C\) is satisfiable
Let check-$Y(s, t)$ be the certifier for problem $Y$: check-$Y(s, t)$ returns 1 if $t$ is a valid certificate for $s$.

$s$ is a yes-instance if and only if there is a $t$ such that check-$Y(s, t)$ returns 1

Construct a circuit $C'$ for the algorithm check-$Y$

hard-wire the instance $s$ to the circuit $C'$ to obtain the circuit $C$

$s$ is a yes-instance if and only if $C$ is satisfiable
Let check-$Y(s, t)$ be the certifier for problem $Y$: check-$Y(s, t)$ returns 1 if $t$ is a valid certificate for $s$.

$s$ is a yes-instance if and only if there is a $t$ such that check-$Y(s, t)$ returns 1.

Construct a circuit $C'$ for the algorithm check-$Y$.

Hard-wire the instance $s$ to the circuit $C'$ to obtain the circuit $C$.

$s$ is a yes-instance if and only if $C$ is satisfiable.

**Theorem** Circuit-Sat is NP-complete.
Reductions of NP-Complete Problems
3-CNФ (conjunctive normal form) is a special case of formula:
3-CNF (conjunctive normal form) is a special case of formula:

- Boolean variables: $x_1, x_2, \ldots, x_n$
3-Sat

3-CNF (conjunctive normal form) is a special case of formula:

- **Boolean variables**: $x_1, x_2, \cdots, x_n$
- **Literals**: $x_i$ or $\neg x_i$
3-Sat

3-CNF (conjunctive normal form) is a special case of formula:

- Boolean variables: $x_1, x_2, \cdots, x_n$
- Literals: $x_i$ or $\neg x_i$
- Clause: disjunction ("or") of at most 3 literals: $x_3 \lor \neg x_4$, $x_1 \lor x_8 \lor \neg x_9$, $\neg x_2 \lor \neg x_5 \lor x_7$
3-Sat

3-CNF (conjunctive normal form) is a special case of formula:

- **Boolean variables:** $x_1, x_2, \ldots, x_n$
- **Literals:** $x_i$ or $\neg x_i$
- **Clause:** disjunction ("or") of at most 3 literals: $x_3 \lor \neg x_4$, $x_1 \lor x_8 \lor \neg x_9$, $\neg x_2 \lor \neg x_5 \lor x_7$
- **3-CNF formula:** conjunction ("and") of clauses:
  $$(x_1 \lor \neg x_2 \lor \neg x_3) \land (x_2 \lor x_3 \lor x_4) \land (\neg x_1 \lor \neg x_3 \lor \neg x_4)$$
3-Sat

Input: a 3-CNF formula
Output: whether the 3-CNF is satisfiable

To satisfy a 3-CNF, we need to satisfy all clauses.
To satisfy a clause, we need to satisfy at least 1 literal.

Assignment:

\[
\begin{align*}
x_1 &= 1, & x_2 &= 1, & x_3 &= 0, & x_4 &= 0
\end{align*}
\]

satisfies

\[
\left( x_1 \lor \neg x_2 \lor \neg x_3 \right) \land 
\left( x_2 \lor x_3 \lor x_4 \right) \land 
\left( \neg x_1 \lor \neg x_3 \lor \neg x_4 \right)
\]
## 3-Sat

<table>
<thead>
<tr>
<th><strong>Input:</strong></th>
<th>a 3-CNF formula</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Output:</strong></td>
<td>whether the 3-CNF is satisfiable</td>
</tr>
</tbody>
</table>

- To satisfy a 3-CNF, we need to satisfy all clauses.
3-Sat

**Input:** a 3-CNF formula

**Output:** whether the 3-CNF is satisfiable

- To satisfy a 3-CNF, we need to satisfy all clauses
- To satisfy a clause, we need to satisfy at least 1 literal
3-Sat

Input: a 3-CNF formula

Output: whether the 3-CNF is satisfiable

- To satisfy a 3-CNF, we need to satisfy all clauses
- To satisfy a clause, we need to satisfy at least 1 literal
- Assignment $x_1 = 1, x_2 = 1, x_3 = 0, x_4 = 0$ satisfies

$$(x_1 \lor \neg x_2 \lor \neg x_3) \land (x_2 \lor x_3 \lor x_4) \land (\neg x_1 \lor \neg x_3 \lor \neg x_4)$$
Circuit-Sat $\leq_p$ 3-Sat

Associate every wire with a new variable

The circuit is equivalent to the following formula:

$$(x_4 = \neg x_3) \land (x_5 = x_1 \lor x_2) \land (x_6 = \neg x_4) \land (x_7 = x_1 \land x_2 \land x_4) \land (x_8 = x_5 \lor x_6) \land (x_9 = x_6 \lor x_7) \land (x_{10} = x_8 \land x_9 \land x_7) \land x_{10}$$
Circuit-Sat $\leq_P$ 3-Sat

- Associate every wire with a new variable

The circuit is equivalent to the following formula:

$$(x_4 \equiv \neg x_3) \land (x_5 \equiv x_1 \lor x_2) \land (x_6 \equiv \neg x_4) \land (x_7 \equiv x_6 \land x_2 \land x_4) \land (x_8 \equiv x_5 \lor x_6) \land (x_9 \equiv x_7 \lor x_8) \land (x_10 \equiv x_8 \land x_9 \land x_7) \land x_10$$
**Circuit-Sat \( \leq_P 3\text{-Sat} \)**

- Associate every wire with a new variable
- The circuit is equivalent to the following formula:

\[
(x_4 = \neg x_3) \land (x_5 = x_1 \lor x_2) \land (x_6 = \neg x_4) \\
\land (x_7 = x_1 \land x_2 \land x_4) \land (x_8 = x_5 \lor x_6) \\
\land (x_9 = x_6 \lor x_7) \land (x_{10} = x_8 \land x_9 \land x_7) \land x_{10}
\]
Circuit-Sat $\leq_P$ 3-Sat

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

Convert each clause to a 3-CNF
Circuit-Sat $\leq_P$ 3-Sat

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

Convert each clause to a 3-CNF

$x_5 = x_1 \lor x_2 \iff$

<table>
<thead>
<tr>
<th>$x_1$</th>
<th>$x_2$</th>
<th>$x_5$</th>
<th>$x_5 \iff x_1 \lor x_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
**Circuit-Sat** $\leq_P$ **3-Sat**

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

**Convert each clause to a 3-CNF**

\[x_5 = x_1 \lor x_2\] $\iff$

<table>
<thead>
<tr>
<th>$x_1$</th>
<th>$x_2$</th>
<th>$x_5$</th>
<th>$x_5 \leftrightarrow x_1 \lor x_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Circuit-Sat $\leq_P$ 3-Sat

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

Convert each clause to a 3-CNF

$x_5 = x_1 \lor x_2 \iff$

$(x_1 \lor x_2 \lor \neg x_5) \land$

<table>
<thead>
<tr>
<th>$x_1$</th>
<th>$x_2$</th>
<th>$x_5$</th>
<th>$x_5 \leftrightarrow x_1 \lor x_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
**Circuit-Sat \( \leq_P 3\text{-Sat} \)**

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

Convert each clause to a 3-CNF

\[
x_5 = x_1 \lor x_2 \iff (x_1 \lor x_2 \lor \neg x_5) \land
\]

<table>
<thead>
<tr>
<th>( x_1 )</th>
<th>( x_2 )</th>
<th>( x_5 )</th>
<th>( x_5 \leftrightarrow x_1 \lor x_2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Circuit-Sat \( \leq_P \) 3-Sat

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

Convert each clause to a 3-CNF

\(x_5 = x_1 \lor x_2 \iff\)
\[(x_1 \lor x_2 \lor \neg x_5) \land (x_1 \lor \neg x_2 \lor x_5) \land\]

<table>
<thead>
<tr>
<th>(x_1)</th>
<th>(x_2)</th>
<th>(x_5)</th>
<th>(x_5 \leftrightarrow x_1 \lor x_2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Circuit-Sat $\leq_P$ 3-Sat

$$(x_4 = \neg x_3) \land (x_5 = x_1 \lor x_2) \land (x_6 = \neg x_4)$$

$$\land (x_7 = x_1 \land x_2 \land x_4) \land (x_8 = x_5 \lor x_6)$$

$$\land (x_9 = x_6 \lor x_7) \land (x_{10} = x_8 \land x_9 \land x_7) \land x_{10}$$

Convert each clause to a 3-CNF

$$x_5 = x_1 \lor x_2 \iff$$

$$(x_1 \lor x_2 \lor \neg x_5) \land$$

$$(x_1 \lor \neg x_2 \lor x_5) \land$$

<table>
<thead>
<tr>
<th>$x_1$</th>
<th>$x_2$</th>
<th>$x_5$</th>
<th>$x_5 \iff x_1 \lor x_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Circuit-Sat $\leq_P$ 3-Sat

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

Convert each clause to a 3-CNF

$x_5 = x_1 \lor x_2 \iff$

$$(x_1 \lor x_2 \lor \neg x_5) \land (x_1 \lor \neg x_2 \lor x_5) \land (\neg x_1 \lor x_2 \lor x_5) \land$$

<table>
<thead>
<tr>
<th>$x_1$</th>
<th>$x_2$</th>
<th>$x_5$</th>
<th>$x_5 \leftrightarrow x_1 \lor x_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Circuit-Sat $\leq_P$ 3-Sat

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

Convert each clause to a 3-CNF

\[
x_5 = x_1 \lor x_2 \iff
\]

\[
(x_1 \lor x_2 \lor \neg x_5) \land
\]

\[
(x_1 \lor \neg x_2 \lor x_5) \land
\]

\[
(\neg x_1 \lor x_2 \lor x_5) \land
\]
Circuit-Sat $\leq_P$ 3-Sat

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

Convert each clause to a 3-CNF

$x_5 = x_1 \lor x_2 \iff$

$$(x_1 \lor x_2 \lor \neg x_5) \land$$
$$(x_1 \lor \neg x_2 \lor x_5) \land$$
$$(\neg x_1 \lor x_2 \lor x_5) \land$$
$$(\neg x_1 \lor \neg x_2 \lor x_5)$$

<table>
<thead>
<tr>
<th>$x_1$</th>
<th>$x_2$</th>
<th>$x_5$</th>
<th>$x_5 \leftrightarrow x_1 \lor x_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Circuit-Sat $\leq_P$ 3-Sat

- Circuit $\iff$ Formula $\iff$ 3-CNF
Circuit-Sat $\leq_P$ 3-Sat

- Circuit $\iff$ Formula $\iff$ 3-CNF
- The circuit is satisfiable if and only if the 3-CNF is satisfiable
Circuit-Sat $\leq_p$ 3-Sat

- Circuit $\iff$ Formula $\iff$ 3-CNF
- The circuit is satisfiable if and only if the 3-CNF is satisfiable
- The size of the 3-CNF formula is polynomial (indeed, linear) in the size of the circuit
Circuit-Sat $\leq_P$ 3-Sat

- Circuit $\iff$ Formula $\iff$ 3-CNF
- The circuit is satisfiable if and only if the 3-CNF is satisfiable
- The size of the 3-CNF formula is polynomial (indeed, linear) in the size of the circuit
- Thus, Circuit-Sat $\leq_P$ 3-Sat
Reductions of NP-Complete Problems

Circuit-Sat

3-Sat

Vertex-Cover

HC

3D-Matching

Subset-Sum

TSP

Knapsack

3-Coloring

Clique

Ind-Set

Set-Cover
Recall: Independent Set Problem

**Def.** An independent set of $G = (V, E)$ is a subset $I \subseteq V$ such that no two vertices in $I$ are adjacent in $G$.

**Independent Set (Ind-Set) Problem**

**Input:** $G = (V, E), k$

**Output:** whether there is an independent set of size $k$ in $G$
3-Sat $\leq_P$ Ind-Set

$(x_1 \lor \neg x_2 \lor \neg x_3) \land (x_2 \lor x_3 \lor x_4) \land (\neg x_1 \lor \neg x_3 \lor x_4)$
3-Sat $\leq_p$ Ind-Set

- $(x_1 \lor \neg x_2 \lor \neg x_3) \land (x_2 \lor x_3 \lor x_4) \land (\neg x_1 \lor \neg x_3 \lor x_4)$

- A clause $\Rightarrow$ a group of 3 vertices, one for each literal

- An edge between every pair of vertices in same group
3-Sat $\leq^P$ Ind-Set

- $(x_1 \lor \neg x_2 \lor \neg x_3) \land (x_2 \lor x_3 \lor x_4) \land (\neg x_1 \lor \neg x_3 \lor x_4)$

- A clause $\Rightarrow$ a group of 3 vertices, one for each literal
- An edge between every pair of vertices in same group
- An edge between every pair of contradicting literals

Problem: whether there is an IS of size $k = \#\text{clauses}$
3-Sat $\leq_P$ Ind-Set

- $(x_1 \lor \neg x_2 \lor \neg x_3) \land (x_2 \lor x_3 \lor x_4) \land (\neg x_1 \lor \neg x_3 \lor x_4)$

- A clause $\Rightarrow$ a group of 3 vertices, one for each literal
- An edge between every pair of vertices in same group
- An edge between every pair of contradicting literals
- Problem: whether there is an IS of size $k = \#\text{clauses}$
3-Sat $\leq_P$ Ind-Set

- $(x_1 \lor \neg x_2 \lor \neg x_3) \land (x_2 \lor x_3 \lor x_4) \land (\neg x_1 \lor \neg x_3 \lor x_4)$

- A clause $\Rightarrow$ a group of 3 vertices, one for each literal
- An edge between every pair of vertices in same group
- An edge between every pair of contradicting literals
- Problem: whether there is an IS of size $k = \#\text{clauses}$

3-Sat instance is yes-instance $\iff$ Ind-Set instance is yes-instance:
3-Sat $\leq_P$ Ind-Set

- $(x_1 \lor \neg x_2 \lor \neg x_3) \land (x_2 \lor x_3 \lor x_4) \land (\neg x_1 \lor \neg x_3 \lor x_4)$

- A clause $\Rightarrow$ a group of 3 vertices, one for each literal
- An edge between every pair of vertices in same group
- An edge between every pair of contradicting literals
- Problem: whether there is an IS of size $k = \#\text{clauses}$

3-Sat instance is yes-instance $\iff$ Ind-Set instance is yes-instance:
- satisfying assignment $\Rightarrow$ independent set of size $k$
- independent set of size $k$ $\Rightarrow$ satisfying assignment
Satisfying Assignment $\Rightarrow$ IS of Size $k$

$$(x_1 \lor \neg x_2 \lor \neg x_3) \land (x_2 \lor x_3 \lor x_4) \land (\neg x_1 \lor \neg x_3 \lor x_4)$$
Satisfying Assignment $\Rightarrow$ IS of Size $k$

- $(x_1 \lor \neg x_2 \lor \neg x_3) \land (x_2 \lor x_3 \lor x_4) \land (\neg x_1 \lor \neg x_3 \lor x_4)$

- For every clause, at least 1 literal is satisfied
Satisfying Assignment $\Rightarrow$ IS of Size $k$

- $(x_1 \lor \neg x_2 \lor \neg x_3) \land (x_2 \lor x_3 \lor x_4) \land (\neg x_1 \lor \neg x_3 \lor x_4)$

- For every clause, at least 1 literal is satisfied
- Pick the vertex correspondent to the literal
Satisfying Assignment $\Rightarrow$ IS of Size $k$

- $(x_1 \lor \neg x_2 \lor \neg x_3) \land (x_2 \lor x_3 \lor x_4) \land (\neg x_1 \lor \neg x_3 \lor x_4)$

- For every clause, at least 1 literal is satisfied
- Pick the vertex correspondent the literal
- So, 1 literal from each group
Satisfying Assignment $\implies$ IS of Size $k$

- $(x_1 \lor \neg x_2 \lor \neg x_3) \land (x_2 \lor x_3 \lor x_4) \land (\neg x_1 \lor \neg x_3 \lor x_4)$

- For every clause, at least 1 literal is satisfied
- Pick the vertex correspondent the literal
- So, 1 literal from each group
- No contradictions among the selected literals
Satisfying Assignment $\Rightarrow$ IS of Size $k$

- $$(x_1 \lor \neg x_2 \lor \neg x_3) \land (x_2 \lor x_3 \lor x_4) \land (\neg x_1 \lor \neg x_3 \lor x_4)$$

- For every clause, at least 1 literal is satisfied
- Pick the vertex correspondent the literal
- So, 1 literal from each group
- No contradictions among the selected literals
- An IS of size $k$
IS of Size $k \Rightarrow$ Satisfying Assignment

$$(x_1 \lor \neg x_2 \lor \neg x_3) \land (x_2 \lor x_3 \lor x_4) \land (\neg x_1 \lor \neg x_3 \lor x_4)$$
IS of Size $k \Rightarrow$ Satisfying Assignment

$(x_1 \lor \neg x_2 \lor \neg x_3) \land (x_2 \lor x_3 \lor x_4) \land (\neg x_1 \lor \neg x_3 \lor x_4)$

For every group, exactly one literal is selected in IS
IS of Size $k \implies$ Satisfying Assignment

- $(x_1 \lor \neg x_2 \lor \neg x_3) \land (x_2 \lor x_3 \lor x_4) \land (\neg x_1 \lor \neg x_3 \lor x_4)$

- For every group, exactly one literal is selected in IS

- No contradictions among the selected literals
IS of Size $k \Rightarrow$ Satisfying Assignment

- $(x_1 \lor \neg x_2 \lor \neg x_3) \land (x_2 \lor x_3 \lor x_4) \land (\neg x_1 \lor \neg x_3 \lor x_4)$

- For every group, exactly one literal is selected in IS
- No contradictions among the selected literals
- If $x_i$ is selected in IS, set $x_i = 1$
IS of Size $k \Rightarrow$ Satisfying Assignment

- $(x_1 \lor \neg x_2 \lor \neg x_3) \land (x_2 \lor x_3 \lor x_4) \land (\neg x_1 \lor \neg x_3 \lor x_4)$

- For every group, exactly one literal is selected in IS
- No contradictions among the selected literals
- If $x_i$ is selected in IS, set $x_i = 1$
- If $\neg x_i$ is selected in IS, set $x_i = 0$
IS of Size $k \Rightarrow$ Satisfying Assignment

- $(x_1 \lor \neg x_2 \lor \neg x_3) \land (x_2 \lor x_3 \lor x_4) \land (\neg x_1 \lor \neg x_3 \lor x_4)$

- For every group, exactly one literal is selected in IS
- No contradictions among the selected literals
- If $x_i$ is selected in IS, set $x_i = 1$
- If $\neg x_i$ is selected in IS, set $x_i = 0$
- Otherwise, set $x_i$ arbitrarily
Reductions of NP-Complete Problems

Circuit-Sat

3-Sat

Clique

Ind-Set

Vertex-Cover

Set-Cover

HC

TSP

3D-Matching

Subset-Sum

Knapsack

3-Coloring
Def. A **clique** in an undirected graph $G = (V, E)$ is a subset $S \subseteq V$ such that $\forall u, v \in S$ we have $(u, v) \in E$.
Def. A clique in an undirected graph $G = (V, E)$ is a subset $S \subseteq V$ such that $\forall u, v \in S$ we have $(u, v) \in E$. 

Clique Problem

Input: $G = (V, E)$ and integer $k > 0$

Output: whether there exists a clique of size $k$ in $G$
Def. A **clique** in an undirected graph $G = (V, E)$ is a subset $S \subseteq V$ such that $\forall u, v \in S$ we have $(u, v) \in E$

**Clique Problem**

**Input:** $G = (V, E)$ and integer $k > 0$,

**Output:** whether there exists a clique of size $k$ in $G$
**Def.** A clique in an undirected graph $G = (V, E)$ is a subset $S \subseteq V$ such that $\forall u, v \in S$ we have $(u, v) \in E$.

**Clique Problem**

**Input:** $G = (V, E)$ and integer $k > 0$,  
**Output:** whether there exists a clique of size $k$ in $G$.

- What is the relationship between Clique and Ind-Set?
Clique $=\ _P$ Ind-Set

Def. Given a graph $G = (V, E)$, define $\overline{G} = (V, \overline{E})$ be the graph such that $(u, v) \in \overline{E}$ if and only if $(u, v) \notin E$.

Obs. $S$ is an independent set in $G$ if and only if $S$ is a clique in $\overline{G}$.
Reductions of NP-Complete Problems

- Circuit-Sat
- 3-Sat
- Ind-Set
- Vertex-Cover
- HC
- 3D-Matching
- Subset-Sum
- TSP
- Knapsack
- 3-Coloring
- Set-Cover
- Clique

Reduction arrows point from the top problem to the problems it reduces to. The diagram shows the relationships between these NP-Complete problems.
Vertex-Cover

**Def.** Given a graph $G = (V, E)$, a vertex cover of $G$ is a subset $S \subseteq V$ such that for every $(u, v) \in E$ then $u \in S$ or $v \in S$. 

![Graph Diagram](image-url)
Vertex-Cover

**Def.** Given a graph $G = (V, E)$, a vertex cover of $G$ is a subset $S \subseteq V$ such that for every $(u, v) \in E$ then $u \in S$ or $v \in S$.
**Def.** Given a graph $G = (V, E)$, a **vertex cover** of $G$ is a subset $S \subseteq V$ such that for every $(u, v) \in E$ then $u \in S$ or $v \in S$.

**Vertex-Cover Problem**

**Input:** $G = (V, E)$ and integer $k$

**Output:** whether there is a vertex cover of $G$ of size at most $k$
Vertex-Cover $= P$ Ind-Set

Q: What is the relationship between Vertex-Cover and Ind-Set?

A: $S$ is a vertex-cover of $G = (V, E)$ if and only if $V \setminus S$ is an independent set of $G$. 
Vertex-Cover $=^p$ Ind-Set

**Q:** What is the relationship between Vertex-Cover and Ind-Set?
Q: What is the relationship between Vertex-Cover and Ind-Set?

A: $S$ is a vertex-cover of $G = (V, E)$ if and only if $V \setminus S$ is an independent set of $G$. 
Reductions of NP-Complete Problems
Set Cover

**Input:** $S_1, S_2, \cdots, S_M \subseteq [N]$ with $\bigcup_{i \in [m]} S_i = [N]$

**Output:** The smallest set $I \subseteq [M]$ satisfying $\bigcup_{i \in I} S_i = [N]$
Set Cover

**Input:** \( S_1, S_2, \cdots, S_M \subseteq [N] \) with \( \bigcup_{i \in [m]} S_i = [N] \)

**Output:** The smallest set \( I \subseteq [M] \) satisfying \( \bigcup_{i \in I} S_i = [N] \)

- decision version: given \( t \), does there exist a solution \( I \) with \( |I| \leq t \)?
Set Cover

**Input:** $S_1, S_2, \cdots, S_M \subseteq [N]$ with $\bigcup_{i \in [m]} S_i = [N]$

**Output:** The smallest set $I \subseteq [M]$ satisfying $\bigcup_{i \in I} S_i = [N]$

- decision version: given $t$, does there exist a solution $I$ with $|I| \leq t$?

Vertex Cover $\leq_P$ Set Cover

- $m$ edges $\iff N$ elements
- $n$ vertices $\iff M$ sets
- vertex is incident to edge $e$ $\iff$ set contains element

- Vertex cover is the special case of set cover where each element appears in exactly two sets.
Reductions of NP-Complete Problems

- Circuit-Sat
- 3-Sat
- Ind-Set
- Vertex-Cover
- HC
- 3D-Matching
- Subset-Sum
- TSP
- Knapsack
- 3-Coloring
- Clique
- Ind-Set
- Vertex-Cover
- Set-Cover
- HC
- TSP
- Subset-Sum
- Knapsack
- 3-Coloring
**Def.** A $k$-coloring of $G = (V, E)$ is a function $f : V \rightarrow \{1, 2, 3, \ldots, k\}$ so that for every edge $(u, v) \in E$, we have $f(u) \neq f(v)$. $G$ is $k$-colorable if there is a $k$-coloring of $G$. 
Def. A $k$-coloring of $G = (V, E)$ is a function $f : V \rightarrow \{1, 2, 3, \ldots, k\}$ so that for every edge $(u, v) \in E$, we have $f(u) \neq f(v)$. $G$ is $k$-colorable if there is a $k$-coloring of $G$. 
Def. A $k$-coloring of $G = (V, E)$ is a function $f : V \rightarrow \{1, 2, 3, \ldots, k\}$ so that for every edge $(u, v) \in E$, we have $f(u) \neq f(v)$. $G$ is $k$-colorable if there is a $k$-coloring of $G$. 

$k$-coloring problem

Input: a graph $G = (V, E)$

Output: whether $G$ is $k$-colorable or not
2-Coloring Problem

**Obs.** A graph $G$ is 2-colorable if and only if it is bipartite.

**Q:** How do we check if a graph $G$ is 2-colorable?
2-Coloring Problem

**Obs.** A graph $G$ is 2-colorable if and only if it is bipartite.

**Q:** How do we check if a graph $G$ is 2-colorable?

**A:** We check if $G$ is bipartite.
3-SAT $\leq_P$ 3-Coloring

- Construct the base graph

Base Graph

```
- True
- False
- Base
```
3-SAT $\leq_P$ 3-Coloring

- Construct the base graph
3-SAT $\leq_P$ 3-Coloring

- Construct the base graph
- Construct a gadget from each clause: gadget is 3-colorable if and only if the clause is satisfied.

Base Graph

$$x_1 \lor \neg x_2 \lor x_3$$
3-SAT $\leq_P$ 3-Coloring

- Construct the base graph
- Construct a gadget from each clause: gadget is 3-colorable if and only if the clause is satisfied.
3-SAT $\leq_P$ 3-Coloring

- Construct the base graph
- Construct a gadget from each clause: gadget is 3-colorable if and only if the clause is satisfied.
3-SAT $\leq_P$ 3-Coloring

- Construct the base graph
- Construct a gadget from each clause: gadget is 3-colorable if and only if the clause is satisfied.

Base Graph

$$x_1 \lor \neg x_2 \lor x_3$$
3-SAT \( \leq_p \) 3-Coloring

- Construct the base graph
- Construct a gadget from each clause: gadget is 3-colorable if and only if the clause is satisfied.
3-SAT $\leq_P$ 3-Coloring

- Construct the base graph
- Construct a gadget from each clause: gadget is 3-colorable if and only if the clause is satisfied.
3-SAT $\leq_P$ 3-Coloring

- Construct the base graph
- Construct a gadget from each clause: gadget is 3-colorable if and only if the clause is satisfied.
Construct the base graph

Construct a gadget from each clause: gadget is 3-colorable if and only if the clause is satisfied.
3-SAT $\leq_P$ 3-Coloring

- Construct the base graph
- Construct a gadget from each clause: gadget is 3-colorable if and only if the clause is satisfied.

Base Graph

$x_1 \lor \lnot x_2 \lor x_3$

True
False

Base Graph

$x_1 \lor \lnot x_2 \lor x_3$

True
False

Base
$x_1 \lor \lnot x_2 \lor x_3$

True
False
3-SAT \leq_P 3-Coloring

- Construct the base graph
- Construct a gadget from each clause: gadget is 3-colorable if and only if the clause is satisfied.
3-SAT $\leq_P$ 3-Coloring

- Construct the base graph
- Construct a gadget from each clause: gadget is 3-colorable if and only if the clause is satisfied.

**Base Graph**

$x_1 \lor \neg x_2 \lor x_3$
3-SAT $\leq_p$ 3-Coloring

- Construct the base graph
- Construct a gadget from each clause: gadget is 3-colorable if and only if the clause is satisfied.

Base Graph

$x_1 \lor \neg x_2 \lor x_3$
3-SAT $\leq_p$ 3-Coloring

- Construct the base graph
- Construct a gadget from each clause: gadget is 3-colorable if and only if the clause is satisfied.
3-SAT $\leq_P$ 3-Coloring

- Construct the base graph
- Construct a gadget from each clause: gadget is 3-colorable if and only if the clause is satisfied.

Base Graph

$\begin{align*}
x_1 \lor \neg x_2 \lor x_3
\end{align*}$
3-SAT $\leq_p$ 3-Coloring

- Construct the base graph
- Construct a gadget from each clause: gadget is 3-colorable if and only if the clause is satisfied.
3-SAT $\leq_P$ 3-Coloring

- Construct the base graph
- Construct a gadget from each clause: gadget is 3-colorable if and only if the clause is satisfied.
3-SAT $\leq_P$ 3-Coloring

- Construct the base graph
- Construct a gadget from each clause: gadget is 3-colorable if and only if the clause is satisfied.

Base Graph

```
x_1 \lor \neg x_2 \lor x_3
```

True, False

Base Graph

```
x_1 \lor \neg x_2 \lor x_3
```

True, False
3-SAT \leq_P 3\text{-Coloring}

- Construct the base graph
- Construct a gadget from each clause: gadget is 3-colorable if and only if the clause is satisfied.

Base Graph

\[ x_1 \lor \neg x_2 \lor x_3 \]
3-SAT \leq_P 3\text{-}Coloring

- Construct the base graph
- Construct a gadget from each clause: gadget is 3-colorable if and only if the clause is satisfied.

**Base Graph**

\[
x_1 \lor \neg x_2 \lor x_3
\]
Construct the base graph

Construct a gadget from each clause: gadget is 3-colorable if and only if the clause is satisfied.
3-SAT $\leq_P$ 3-Coloring

- Construct the base graph
- Construct a gadget from each clause: gadget is 3-colorable if and only if the clause is satisfied.
3-SAT $\leq_P$ 3-Coloring

- Construct the base graph
- Construct a gadget from each clause: gadget is 3-colorable if and only if the clause is satisfied.

**Base Graph**

$$x_1 \lor \neg x_2 \lor x_3$$
3-SAT $\leq_P$ 3-Coloring

- Construct the base graph
- Construct a gadget from each clause: gadget is 3-colorable if and only if the clause is satisfied.

Base Graph

$$x_1 \lor \overline{x}_2 \lor x_3$$
3-SAT $\leq_P$ 3-Coloring

- Construct the base graph
- Construct a gadget from each clause: gadget is 3-colorable if and only if the clause is satisfied.

Base Graph

$x_1 \lor \neg x_2 \lor x_3$
3-SAT $\leq_P$ 3-Coloring

- Construct the base graph
- Construct a gadget from each clause: gadget is 3-colorable if and only if the clause is satisfied.

Base Graph
Reductions of NP-Complete Problems

- Circuit-Sat
- 3-Sat
- Ind-Set
- Vertex-Cover
- HC
- 3D-Matching
- Subset-Sum
- TSP
- Knapsack
- 3-Coloring
- Clique
- Ind-Set
- Vertex-Cover
- Set-Cover
- HC
- TSP
- Subset-Sum
- Knapsack
Recall: Hamiltonian Cycle (HC) Problem

**Input:** graph $G = (V, E)$

**Output:** whether $G$ contains a Hamiltonian cycle
Recall: Hamiltonian Cycle (HC) Problem

**Input:** graph $G = (V, E)$

**Output:** whether $G$ contains a Hamiltonian cycle

- We consider Hamiltonian Cycle Problem in directed graphs
Recall: Hamiltonian Cycle (HC) Problem

**Input:** graph $G = (V, E)$

**Output:** whether $G$ contains a Hamiltonian cycle

- We consider Hamiltonian Cycle Problem in **directed** graphs
- Exercise: $HC$-directed $\leq_P HC$
3-Sat $\leq_P$ Directed-HC

- Vertices $s, t$
- A long enough double-path $P_i$ for each variable $x_i$

![Diagram of vertices and paths for 3-Sat to Directed-HC reduction]
3-Sat $\leq_P$ Directed-HC

- Vertices $s, t$
- A long enough double-path $P_i$ for each variable $x_i$
- Edges from $s$ to $P_1$
- Edges from $P_n$ to $t$
- Edges from $P_i$ to $P_{i+1}$

Vertices $s, t$

A long enough double-path $P_i$ for each variable $x_i$

Edges from $s$ to $P_1$

Edges from $P_n$ to $t$

Edges from $P_i$ to $P_{i+1}$
3-Sat $\leq_P$ Directed-HC

- Vertices $s, t$
- A long enough double-path $P_i$ for each variable $x_i$
- Edges from $s$ to $P_1$
- Edges from $P_n$ to $t$
- Edges from $P_i$ to $P_{i+1}$
- $x_i = 1 \iff$ traverse $P_i$ from left to right
3-Sat $\leq_P$ Directed-HC

- Vertices $s, t$
- A long enough double-path $P_i$ for each variable $x_i$
- Edges from $s$ to $P_1$
- Edges from $P_n$ to $t$
- Edges from $P_i$ to $P_{i+1}$
- $x_i = 1 \iff$ traverse $P_i$ from left to right
- e.g., $x_1 = 1, x_2 = 1, x_3 = 0, x_4 = 0$
3-Sat $\leq_P$ Directed-HC

- Vertices $s, t$
- A long enough double-path $P_i$ for each variable $x_i$
- Edges from $s$ to $P_1$
- Edges from $P_n$ to $t$
- Edges from $P_i$ to $P_{i+1}$
- $x_i = 1 \iff$ traverse $P_i$ from left to right
- e.g.,
  $x_1 = 1, x_2 = 1, x_3 = 0, x_4 = 0$
3-Sat $\leq_P$ Directed-HC

- There are exactly $2^n$ different Hamiltonian cycles, each corresponding to one assignment of variables.

Add a vertex for each clause, so that the vertex can be visited only if one of the literals is satisfied.
There are exactly $2^n$ different Hamiltonian cycles, each correspondent to one assignment of variables.

Add a vertex for each clause, so that the vertex can be visited only if one of the literals is satisfied.
A Path Should Be Long Enough

\[ \leq 3k + 1 \text{ vertices} \]

- \( k \): number of clauses
Yes-Instance for 3-Sat $\Rightarrow$ Yes-Instance for Di-HC

\[ c_1 = x_1 \lor \overline{x}_2 \lor x_3 \]

- In base graph, construct an HC according to the satisfying assignment

For every clause, one literal is satisfied
Visit the vertex for the clause by taking a "detour" from the path for the literal
Yes-Instance for 3-Sat $\Rightarrow$ Yes-Instance for Di-HC

$c_1 = x_1 \lor \overline{x}_2 \lor x_3$

- In base graph, construct an HC according to the satisfying assignment
- For every clause, one literal is satisfied
Yes-Instance for 3-Sat $\Rightarrow$ Yes-Instance for Di-HC

In base graph, construct an HC according to the satisfying assignment.

- For every clause, one literal is satisfied.
- Visit the vertex for the clause by taking a “detour” from the path for the literal.

$$c_1 = x_1 \lor \overline{x}_2 \lor x_3$$
Yes-Instance for Di-HC \Rightarrow Yes-Instance for 3-Sat

Idea: for each path $P_i$, must follow the left-to-right or right-to-right pattern.
Yes-Instance for Di-HC $\Rightarrow$ Yes-Instance for 3-Sat

Idea: for each path $P_i$, must follow the left-to-right or right-to-right pattern.

To visit vertex $b$, can either go $a-b-c$ or $b-c-a$
Yes-Instance for Di-HC $\Rightarrow$ Yes-Instance for 3-Sat

- Idea: for each path $P_i$, must follow the left-to-right or right-to-right pattern.
- To visit vertex $b$, can either go $a-b-c$ or $b-c-a$
Yes-Instance for Di-HC $\Rightarrow$ Yes-Instance for 3-Sat

- Idea: for each path $P_i$, must follow the left-to-right or right-to-right pattern.
- To visit vertex $b$, can either go $a-b-c$ or $b-c-a$
- Created “chunks” of 3 vertices.
Yes-Instance for Di-HC $\Rightarrow$ Yes-Instance for 3-Sat

- Idea: for each path $P_i$, must follow the left-to-right or right-to-right pattern.
- To visit vertex $b$, can either go $a-b-c$ or $b-c-a$
- Created “chunks” of 3 vertices.
- Directions of the chunks must be the same
Yes-Instance for Di-HC $\Rightarrow$ Yes-Instance for 3-Sat

- Idea: for each path $P_i$, must follow the left-to-right or right-to-right pattern.
- To visit vertex $b$, can either go $a-b-c$ or $b-c-a$
- Created “chunks” of 3 vertices.
- Directions of the chunks must be the same
- Can not take a detour to some other path
Reductions of NP-Complete Problems
Traveling Salesman Problem

- A salesman needs to visit $n$ cities $1, 2, 3, \ldots, n$
- He needs to start from and return to city 1
- Goal: find a tour with the minimum cost
Traveling Salesman Problem

- A salesman needs to visit \( n \) cities \( 1, 2, 3, \ldots, n \)
- He needs to start from and return to city 1
- Goal: find a tour with the minimum cost
Traveling Salesman Problem

- A salesman needs to visit \( n \) cities \( 1, 2, 3, \ldots, n \).
- He needs to start from and return to city 1.
- Goal: find a tour with the minimum cost.
Traveling Salesman Problem

- A salesman needs to visit \( n \) cities 1, 2, 3, \( \cdots \), \( n \)
- He needs to start from and return to city 1
- Goal: find a tour with the minimum cost
Traveling Salesman Problem

- A salesman needs to visit \( n \) cities 1, 2, 3, \( \cdots \), \( n \)
- He needs to start from and return to city 1
- Goal: find a tour with the minimum cost
Traveling Salesman Problem

- A salesman needs to visit $n$ cities $1, 2, 3, \ldots, n$
- He needs to start from and return to city 1
- Goal: find a tour with the minimum cost
Traveling Salesman Problem

- A salesman needs to visit \( n \) cities \( 1, 2, 3, \ldots, n \)
- He needs to start from and return to city 1
- Goal: find a tour with the minimum cost
Traveling Salesman Problem

- A salesman needs to visit \( n \) cities 1, 2, 3, \( \cdots \), \( n \)
- He needs to start from and return to city 1
- Goal: find a tour with the minimum cost

**Travelling Salesman Problem (TSP)**

**Input:** a graph \( G = (V, E) \), weights \( w : E \to \mathbb{R}_{\geq 0} \), and \( L > 0 \)

**Output:** whether there is a tour of length at most \( D \)
There is a Hamilton cycle in $G$ if and only if there is a tour for the salesman of length $n = |V|$. 
Reductions of NP-Complete Problems

Clique

Ind-Set

Vertex-Cover

Set-Cover

HC

TSP

Subset-Sum

Knapsack

3D-Matching

3-Sat

Circuit-Sat

3-Coloring
3D-Matching

**Input:** \(|X| = |Y| = |Z| = n, (x_1, y_1, z_1), (x_2, y_2, z_2), \cdots, (x_m, y_m, z_m) \in X \times Y \times Z\)

**Output:** whether there exists \(S \subseteq [m], |S| = n\) such that

- \(\{x_i : i \in S\} = X, \{y_i : i \in S\} = Y, \{z_i : i \in S\} = Z\)
3D-Matching

**Input:** \(|X| = |Y| = |Z| = n,\)
\((x_1, y_1, z_1), (x_2, y_2, z_2), \cdots, (x_m, y_m, z_m) \in X \times Y \times Z\)

**Output:** whether there exists \(S \subseteq [m], |S| = n\) such that
\(\{x_i : i \in S\} = X, \{y_i : i \in S\} = Y, \{z_i : i \in S\} = Z\)
3D-Matching

**Input:** $|X| = |Y| = |Z| = n,$

$(x_1, y_1, z_1), (x_2, y_2, z_2), \cdots, (x_m, y_m, z_m) \in X \times Y \times Z$

**Output:** whether there exists $S \subseteq [m], |S| = n$ such that

- $\{x_i : i \in S\} = X, \{y_i : i \in S\} = Y, \{z_i : i \in S\} = Z$
3D-Matching

**Input:** \(|X| = |Y| = |Z| = n,\)
\[(x_1, y_1, z_1), (x_2, y_2, z_2), \cdots, (x_m, y_m, z_m) \in X \times Y \times Z\]

**Output:** whether there exists \(S \subseteq [m], |S| = n\) such that
\[
\{x_i : i \in S\} = X, \{y_i : i \in S\} = Y, \{z_i : i \in S\} = Z
\]
3D-Matching

**Input:** $|X| = |Y| = |Z| = n,$

$$(x_1, y_1, z_1), (x_2, y_2, z_2), \ldots, (x_m, y_m, z_m) \in X \times Y \times Z$$

**Output:** whether there exists $S \subseteq [m], |S| = n$ such that

- $\{x_i : i \in S\} = X$, $\{y_i : i \in S\} = Y$, $\{z_i : i \in S\} = Z$
3D-Matching

**Input:** \( |X| = |Y| = |Z| = n, \)

\((x_1, y_1, z_1), (x_2, y_2, z_2), \cdots, (x_m, y_m, z_m) \in X \times Y \times Z\)

**Output:** whether there exists \( S \subseteq [m], |S| = n \) such that

- \( \{x_i : i \in S\} = X, \{y_i : i \in S\} = Y, \{z_i : i \in S\} = Z \)
3D-Matching

**Input:** \(|X| = |Y| = |Z| = n,\)
\[(x_1, y_1, z_1), (x_2, y_2, z_2), \ldots, (x_m, y_m, z_m) \in X \times Y \times Z\]

**Output:** whether there exists \(S \subseteq [m], |S| = n\) such that
- \(\{x_i : i \in S\} = X, \{y_i : i \in S\} = Y, \{z_i : i \in S\} = Z\)
A Strategy of Polynomial Reduction

Recall the definition of polynomial time reductions:

**Def.** Given a black box algorithm $A$ that solves a problem $X$, if any instance of a problem $Y$ can be solved using a polynomial number of standard computational steps, plus a polynomial number of calls to $A$, then we say $Y$ is polynomial-time reducible to $X$, denoted as $Y \leq_P X$. 
Recall the definition of polynomial time reductions:

**Def.** Given a black box algorithm $A$ that solves a problem $X$, if any instance of a problem $Y$ can be solved using a polynomial number of standard computational steps, plus a polynomial number of calls to $A$, then we say $Y$ is polynomial-time reducible to $X$, denoted as $Y \leq_P X$.

- In general, algorithm for $Y$ can call the algorithm for $X$ many times.
A Strategy of Polynomial Reduction

Recall the definition of polynomial time reductions:

**Def.** Given a black box algorithm $A$ that solves a problem $X$, if any instance of a problem $Y$ can be solved using a polynomial number of standard computational steps, plus a polynomial number of calls to $A$, then we say $Y$ is polynomial-time reducible to $X$, denoted as $Y \leq_P X$.

- In general, algorithm for $Y$ can call the algorithm for $X$ many times.
- However, for most reductions, we call algorithm for $X$ only once.
A Strategy of Polynomial Reduction

Recall the definition of polynomial time reductions:

**Def.** Given a black box algorithm $A$ that solves a problem $X$, if any instance of a problem $Y$ can be solved using a polynomial number of standard computational steps, plus a polynomial number of calls to $A$, then we say $Y$ is polynomial-time reducible to $X$, denoted as $Y \leq_P X$.

- In general, algorithm for $Y$ can call the algorithm for $X$ many times.
- However, for most reductions, we call algorithm for $X$ only once.
- That is, for a given instance $s_Y$ for $Y$, we only construct one instance $s_X$ for $X$. 
A Strategy of Polynomial Reduction

- Given an instance $s_Y$ of problem $Y$, show how to construct in polynomial time an instance $s_X$ of problem such that:
  - $s_Y$ is a yes-instance of $Y \Rightarrow s_X$ is a yes-instance of $X$
  - $s_X$ is a yes-instance of $X \Rightarrow s_Y$ is a yes-instance of $Y$
Outline

1. Some Hard Problems
2. P, NP and Co-NP
3. Polynomial Time Reductions and NP-Completeness
4. NP-Complete Problems
5. Dealing with NP-Hard Problems
6. Summary
Q: How far away are we from proving or disproving P = NP?
Q: How far away are we from proving or disproving P = NP?

- Try to prove an “unconditional” lower bound on running time of algorithm solving a NP-complete problem.
Q: How far away are we from proving or disproving $P = \text{NP}$?

- Try to prove an “unconditional” lower bound on running time of algorithm solving a NP-complete problem.
- For 3-Sat problem:
Q: How far away are we from proving or disproving $P = NP$?

- Try to prove an “unconditional” lower bound on running time of algorithm solving a NP-complete problem.
- For 3-Sat problem:
  - Assume the number of clauses is $\Theta(n)$, $n = \text{number variables}$
Q: How far away are we from proving or disproving $P = NP$?

- Try to prove an “unconditional” lower bound on running time of algorithm solving a NP-complete problem.
- For 3-Sat problem:
  - Assume the number of clauses is $\Theta(n)$, $n = \text{number variables}$
  - Best algorithm runs in time $O(c^n)$ for some constant $c > 1$
Q: How far away are we from proving or disproving P = NP?

- Try to prove an “unconditional” lower bound on running time of algorithm solving a NP-complete problem.

- For 3-Sat problem:
  - Assume the number of clauses is \( \Theta(n) \), \( n = \) number variables
  - Best algorithm runs in time \( O(c^n) \) for some constant \( c > 1 \)
  - Best lower bound is \( \Omega(n) \)
Q: How far away are we from proving or disproving P = NP?

- Try to prove an “unconditional” lower bound on running time of algorithm solving a NP-complete problem.
- For 3-Sat problem:
  - Assume the number of clauses is $\Theta(n)$, $n =$ number variables
  - Best algorithm runs in time $O(c^n)$ for some constant $c > 1$
  - Best lower bound is $\Omega(n)$
- Essentially we have no techniques for proving lower bound for running time
Dealing with NP-Hard Problems

- Faster exponential time algorithms
- Solving the problem for special cases
- Fixed parameter tractability
- Approximation algorithms
Faster Exponential Time Algorithms

3-SAT:

Brute-force: $O(2^n \cdot \text{poly}(n))$

Practical SAT Solver: solves real-world sat instances with more than 10,000 variables

Travelling Salesman Problem:

Brute-force: $O(n! \cdot \text{poly}(n))$

Better algorithm: $O(2^n \cdot \text{poly}(n))$

In practice: TSP Solver can solve Euclidean TSP instances with more than 100,000 vertices
Faster Exponential Time Algorithms

3-SAT:
- Brute-force: $O(2^n \cdot \text{poly}(n))$
Faster Exponential Time Algorithms

3-SAT:
- Brute-force: $O(2^n \cdot \text{poly}(n))$
- $2^n \rightarrow 1.844^n \rightarrow 1.3334^n$
Faster Exponential Time Algorithms

3-SAT:
- Brute-force: $O(2^n \cdot \text{poly}(n))$
- $2^n \rightarrow 1.844^n \rightarrow 1.3334^n$
- Practical SAT Solver: solves real-world sat instances with more than 10,000 variables
Faster Exponential Time Algorithms

3-SAT:
- Brute-force: $O(2^n \cdot \text{poly}(n))$
- $2^n \rightarrow 1.844^n \rightarrow 1.3334^n$
- Practical SAT Solver: solves real-world sat instances with more than 10,000 variables

Travelling Salesman Problem:
Faster Exponential Time Algorithms

3-SAT:
- Brute-force: $O(2^n \cdot \text{poly}(n))$
- $2^n \rightarrow 1.844^n \rightarrow 1.3334^n$
- Practical SAT Solver: solves real-world sat instances with more than 10,000 variables

Travelling Salesman Problem:
- Brute-force: $O(n! \cdot \text{poly}(n))$
Faster Exponential Time Algorithms

3-SAT:
- Brute-force: $O(2^n \cdot \text{poly}(n))$
- $2^n \rightarrow 1.844^n \rightarrow 1.3334^n$
- Practical SAT Solver: solves real-world sat instances with more than 10,000 variables

Travelling Salesman Problem:
- Brute-force: $O(n! \cdot \text{poly}(n))$
- Better algorithm: $O(2^n \cdot \text{poly}(n))$
Faster Exponential Time Algorithms

3-SAT:
- Brute-force: $O(2^n \cdot \text{poly}(n))$
- $2^n \rightarrow 1.844^n \rightarrow 1.3334^n$
- Practical SAT Solver: solves real-world sat instances with more than 10,000 variables

Travelling Salesman Problem:
- Brute-force: $O(n! \cdot \text{poly}(n))$
- Better algorithm: $O(2^n \cdot \text{poly}(n))$
- In practice: TSP Solver can solve Euclidean TSP instances with more than 100,000 vertices
Solving the problem for special cases

Maximum independent set problem is NP-hard on general graphs, but easy on
Solving the problem for special cases

Maximum independent set problem is NP-hard on general graphs, but easy on
trees
Solving the problem for special cases

Maximum independent set problem is NP-hard on general graphs, but easy on
- trees
- bounded tree-width graphs
Maximum independent set problem is NP-hard on general graphs, but easy on
- trees
- bounded tree-width graphs
- interval graphs
Solving the problem for special cases

Maximum independent set problem is NP-hard on general graphs, but easy on
- trees
- bounded tree-width graphs
- interval graphs
- ...
Problem: whether there is a vertex cover of size $k$, for a **small** $k$ (number of nodes is $n$, number of edges is $\Theta(n)$.)
Fixed Parameter Tractability

- Problem: whether there is a vertex cover of size $k$, for a small $k$ (number of nodes is $n$, number of edges is $\Theta(n)$.)
- Brute-force algorithm: $O(kn^{k+1})$
Problem: whether there is a vertex cover of size $k$, for a small $k$ (number of nodes is $n$, number of edges is $\Theta(n)$.)

- Brute-force algorithm: $O(kn^{k+1})$
- Better running time: $O(2^k \cdot kn)$
Fixed Parameter Tractability

- Problem: whether there is a vertex cover of size $k$, for a small $k$ (number of nodes is $n$, number of edges is $\Theta(n)$.)
- Brute-force algorithm: $O(kn^{k+1})$
- Better running time: $O(2^k \cdot kn)$
- Running time is $f(k)n^c$ for some $c$ independent of $k$
Fixed Parameter Tractability

- Problem: whether there is a vertex cover of size $k$, for a small $k$ (number of nodes is $n$, number of edges is $\Theta(n)$.)
- Brute-force algorithm: $O(kn^{k+1})$
- Better running time: $O(2^k \cdot kn)$
- Running time is $f(k)n^c$ for some $c$ independent of $k$
- Vertex-Cover is fixed-parameter tractable.
Approximation Algorithms

For optimization problems, approximation algorithms will find sub-optimal solutions in \textit{polynomial time}.
Approximation Algorithms

- For optimization problems, approximation algorithms will find sub-optimal solutions in *polynomial time*
- **Approximation ratio** is the ratio between the quality of the solution output by the algorithm and the quality of the optimal solution
Approximation Algorithms

- For optimization problems, approximation algorithms will find sub-optimal solutions in polynomial time.
- **Approximation ratio** is the ratio between the quality of the solution output by the algorithm and the quality of the optimal solution.
- We want to make the approximation ratio as small as possible, while maintaining the property that the algorithm runs in polynomial time.

There is a 2-approximation for the vertex cover problem: we can efficiently find a vertex cover whose size is at most 2 times that of the optimal vertex cover.
Approximation Algorithms

- For optimization problems, approximation algorithms will find sub-optimal solutions in polynomial time.
- **Approximation ratio** is the ratio between the quality of the solution output by the algorithm and the quality of the optimal solution.
- We want to make the approximation ratio as small as possible, while maintaining the property that the algorithm runs in polynomial time.
- There is an 2-approximation for the vertex cover problem: we can efficiently find a vertex cover whose size is at most 2 times that of the optimal vertex cover.
Outline

1. Some Hard Problems
2. P, NP and Co-NP
3. Polynomial Time Reductions and NP-Completeness
4. NP-Complete Problems
5. Dealing with NP-Hard Problems
6. Summary
Summary

- We consider decision problems
- Inputs are encoded as \( \{0, 1\} \)-strings

**Def.** The complexity class \( P \) is the set of decision problems \( X \) that can be solved in polynomial time.

- Alice has a supercomputer, fast enough to run an exponential time algorithm
- Bob has a slow computer, which can only run a polynomial-time algorithm

**Def.** (Informal) The complexity class \( NP \) is the set of problems for which Alice can convince Bob a yes instance is a yes instance.
Summary

**Def.** $B$ is an efficient certifier for a problem $X$ if

- $B$ is a polynomial-time algorithm that takes two input strings $s$ and $t$
- there is a polynomial function $p$ such that, $X(s) = 1$ if and only if there is string $t$ such that $|t| \leq p(|s|)$ and $B(s, t) = 1$.

The string $t$ such that $B(s, t) = 1$ is called a certificate.

**Def.** The complexity class **NP** is the set of all problems for which there exists an efficient certifier.
**Def.** Given a black box algorithm $A$ that solves a problem $X$, if any instance of a problem $Y$ can be solved using a polynomial number of standard computational steps, plus a polynomial number of calls to $A$, then we say $Y$ is polynomial-time reducible to $X$, denoted as $Y \leq_P X$.

**Def.** A problem $X$ is called NP-complete if

1. $X \in \text{NP}$, and
2. $Y \leq_P X$ for every $Y \in \text{NP}$.

- If any NP-complete problem can be solved in polynomial time, then $P = NP$.
- Unless $P = NP$, a NP-complete problem can not be solved in polynomial time.
Summary

- 3D-Matching
- Circuit-Sat
- 3-Sat
- Ind-Set
- Vertex-Cover
- HC
- Set-Cover
- Subset-Sum
- TSP
- Knapsack
- 3-Coloring
- Clique
- Ind-Set
- HC
- 3D-Matching
- 3-Coloring
- Vertex-Cover
- Set-Cover
- TSP
- Subset-Sum
- Knapsack
Summary

Proof of NP-Completeness for Circuit-Sat

- Fact 1: a polynomial-time algorithm can be converted to a polynomial-size circuit
- Fact 2: for a problem in NP, there is an efficient certifier.

Given a problem $X \in \text{NP}$, let $B(s, t)$ be the certifier
- Convert $B(s, t)$ to a circuit and hard-wire $s$ to the input gates
- $s$ is a yes-instance if and only if the resulting circuit is satisfiable

- Proof of NP-Completeness for other problems by reductions