# NP and Computational Intractability

T. M. Murali

November 30, 2009

# Algorithm Design

- Patterns
  - Greed. $O(n \log n)$ interval scheduling.
  - Divide-and-conquer. $O(n \log n)$ closest pair of points.
  - Dynamic programming. $O(n^2)$ edit distance.
  - Duality. $O(n^3)$ maximum flow and minimum cuts.

# Algorithm Design

- Patterns
  - Greed. $O(n \log n)$ interval scheduling.
  - Divide-and-conquer. $O(n \log n)$ closest pair of points.
  - Dynamic programming. $O(n^2)$ edit distance.
  - Duality. $O(n^3)$ maximum flow and minimum cuts.
  - Reductions.
  - Local search.
  - Randomization.

# Algorithm Design

- Patterns
  - Greed. $O(n \log n)$ interval scheduling.
  - Divide-and-conquer. $O(n \log n)$ closest pair of points.
  - Dynamic programming. $O(n^2)$ edit distance.
  - Duality. $O(n^3)$ maximum flow and minimum cuts.
  - Reductions.
  - Local search.
  - Randomization.
- "Anti-patterns"
  - NP-completeness. $O(n^k)$ algorithm unlikely.
  - PSPACE-completeness. $O(n^k)$ certification algorithm unlikely.
  - Undecidability. No algorithm possible.

# Computational Tractability

- When is an algorithm an efficient solution to a problem?

# Computational Tractability

- When is an algorithm an efficient solution to a problem? When its running time is polynomial in the size of the input.

# Computational Tractability

- When is an algorithm an efficient solution to a problem? When its running time is polynomial in the size of the input.
- A problem is *computationally tractable* if it has a polynomial-time algorithm.

# Computational Tractability

- When is an algorithm an efficient solution to a problem? When its running time is polynomial in the size of the input.
- A problem is *computationally tractable* if it has a polynomial-time algorithm.

| **Polynomial time** | **Probably not** |
|---|---|
| Shortest path | Longest path |
| Matching | 3-D matching |
| Minimum cut | Maximum cut |
| 2-SAT | 3-SAT |
| Planar four-colour | Planar three-colour |
| Bipartite vertex cover | Vertex cover |
| Primality testing | Factoring |

# Problem Classification

- Classify problems based on whether they admit efficient solutions or not.
- Some extremely hard problems cannot be solved efficiently (e.g., chess on an $n$-by-$n$ board).

# Problem Classification

- Classify problems based on whether they admit efficient solutions or not.
- Some extremely hard problems cannot be solved efficiently (e.g., chess on an $n$-by-$n$ board).
- However, classification is unclear for a very large number of discrete computational problems.

# Problem Classification

- Classify problems based on whether they admit efficient solutions or not.
- Some extremely hard problems cannot be solved efficiently (e.g., chess on an $n$-by-$n$ board).
- However, classification is unclear for a very large number of discrete computational problems.
- We can prove that these problems are fundamentally equivalent and are manifestations of the same problem!

# Polynomial-Time Reduction

- Goal is to express statements of the type "Problem $X$ is at least as hard as problem $Y$."
- Use the notion of *reductions*.
- *$Y$ is polynomial-time reducible to $X$ ($Y \leq_P X$)*

# Polynomial-Time Reduction

- Goal is to express statements of the type "Problem $X$ is at least as hard as problem $Y$."
- Use the notion of *reductions*.
- *$Y$ is polynomial-time reducible to $X$ ($Y \leq_P X$)* if an arbitrary instance of $Y$ can be solved using a polynomial number of standard operations, plus a polynomial number of calls to a black box that solves problem $X$.
- $Y \leq_P X$ implies that "$X$ is at least as hard as $Y$."
- Such reductions are *Cook reductions*. *Karp reductions* allow only one call to the black box that solves $X$.

# Usefulness of Reductions

- Claim: If $Y \leq_P X$ and $X$ can be solved in polynomial time, then $Y$ can be solved in polynomial time.

# Usefulness of Reductions

- Claim: If $Y \leq_P X$ and $X$ can be solved in polynomial time, then $Y$ can be solved in polynomial time.
- Contrapositive: If $Y \leq_P X$ and $Y$ cannot be solved in polynomial time, then $X$ cannot be solved in polynomial time.
- Informally: If $Y$ is hard, and we can show that $Y$ reduces to $X$, then the hardness "spreads" to $X$.

# Reduction Strategies

- Simple equivalence.
- Special case to general case.
- Encoding with gadgets.

# Optimisation versus Decision Problems

- So far, we have developed algorithms that solve optimisation problems.
  - Compute the largest flow.
  - Find the closest pair of points.
  - Find the schedule with the least completion time.

# Optimisation versus Decision Problems

- So far, we have developed algorithms that solve optimisation problems.
  - Compute the largest flow.
  - Find the closest pair of points.
  - Find the schedule with the least completion time.
- Now, we will focus on *decision versions* of problems, e.g., is there a flow with value at least $k$, for a given value of $k$?

# Independent Set and Vertex Cover

- Given an undirected graph $G(V, E)$, a subset $S \subseteq V$ is an *independent set* if no two vertices in $S$ are connected by an edge.
- Given an undirected graph $G(V, E)$, a subset $S \subseteq V$ is a *vertex cover* if every edge in $E$ is incident on at least one vertex in $S$.

# Independent Set and Vertex Cover

- Given an undirected graph $G(V, E)$, a subset $S \subseteq V$ is an *independent set* if no two vertices in $S$ are connected by an edge.
- Given an undirected graph $G(V, E)$, a subset $S \subseteq V$ is a *vertex cover* if every edge in $E$ is incident on at least one vertex in $S$.

INDEPENDENT SET

**INSTANCE:** Undirected graph $G$ and an integer $k$

**QUESTION:** Does $G$ contain an independent set of size

VERTEX COVER

**INSTANCE:** Undirected graph $G$ and an integer $l$

**QUESTION:** Does $G$ contain a vertex cover of size

# Independent Set and Vertex Cover

- Given an undirected graph $G(V, E)$, a subset $S \subseteq V$ is an *independent set* if no two vertices in $S$ are connected by an edge.
- Given an undirected graph $G(V, E)$, a subset $S \subseteq V$ is a *vertex cover* if every edge in $E$ is incident on at least one vertex in $S$.

INDEPENDENT SET

**INSTANCE:** Undirected graph $G$ and an integer $k$

**QUESTION:** Does $G$ contain an independent set of size at least $k$?

VERTEX COVER

**INSTANCE:** Undirected graph $G$ and an integer $l$

**QUESTION:** Does $G$ contain a vertex cover of size at most $l$?

# Independent Set and Vertex Cover

- Given an undirected graph $G(V, E)$, a subset $S \subseteq V$ is an *independent set* if no two vertices in $S$ are connected by an edge.
- Given an undirected graph $G(V, E)$, a subset $S \subseteq V$ is a *vertex cover* if every edge in $E$ is incident on at least one vertex in $S$.

INDEPENDENT SET

**INSTANCE:** Undirected graph $G$ and an integer $k$

**QUESTION:** Does $G$ contain an independent set of size at least $k$?

VERTEX COVER

**INSTANCE:** Undirected graph $G$ and an integer $l$

**QUESTION:** Does $G$ contain a vertex cover of size at most $l$?

- Demonstrate simple equivalence between these two problems.

# Independent Set and Vertex Cover

- Given an undirected graph $G(V, E)$, a subset $S \subseteq V$ is an *independent set* if no two vertices in $S$ are connected by an edge.
- Given an undirected graph $G(V, E)$, a subset $S \subseteq V$ is a *vertex cover* if every edge in $E$ is incident on at least one vertex in $S$.

INDEPENDENT SET

**INSTANCE:** Undirected graph $G$ and an integer $k$

**QUESTION:** Does $G$ contain an independent set of size at least $k$?

VERTEX COVER

**INSTANCE:** Undirected graph $G$ and an integer $l$

**QUESTION:** Does $G$ contain a vertex cover of size at most $l$?

- Demonstrate simple equivalence between these two problems.
- Claim: INDEPENDENT SET $\leq_P$ VERTEX COVER and VERTEX COVER $\leq_P$ INDEPENDENT SET.

# Strategy for Proving Indep. Set $\leq_P$ Vertex Cover

1. Start with an arbitrary instance of INDEPENDENT SET: an undirected graph $G(V, E)$ and an integer $k$.
2. From $G(V, E)$ and $k$, create an instance of VERTEX COVER: an undirected graph $G'(V', E')$ and an integer $l$.
3. Prove that $G(V, E)$ has an independent set of size $\geq k$ iff $G'(V', E')$ has a vertex cover of size $\leq l$.

# Strategy for Proving Indep. Set $\leq_P$ Vertex Cover

1. Start with an arbitrary instance of INDEPENDENT SET: an undirected graph $G(V, E)$ and an integer $k$.
2. From $G(V, E)$ and $k$, create an instance of VERTEX COVER: an undirected graph $G'(V', E')$ and an integer $l$.
3. Prove that $G(V, E)$ has an independent set of size $\geq k$ iff $G'(V', E')$ has a vertex cover of size $\leq l$.

- Transformation and proof must be correct for all possible graphs $G(V, E)$ and all possible values of $k$.
- Why is the proof an iff statement?

# Strategy for Proving Indep. Set $\leq_P$ Vertex Cover

1. Start with an arbitrary instance of INDEPENDENT SET: an undirected graph $G(V, E)$ and an integer $k$.
2. From $G(V, E)$ and $k$, create an instance of VERTEX COVER: an undirected graph $G'(V', E')$ and an integer $l$.
3. Prove that $G(V, E)$ has an independent set of size $\geq k$ iff $G'(V', E')$ has a vertex cover of size $\leq l$.

- Transformation and proof must be correct for all possible graphs $G(V, E)$ and all possible values of $k$.
- Why is the proof an iff statement? In the reduction, we are using black box for VERTEX COVER to solve INDEPENDENT SET.
  - (i) If there is an independent set size $\geq k$, we must be sure that there is a vertex cover of size $\leq l$, so that we know that the black box will find this vertex cover.
  - (ii) If the black box finds a vertex cover of size $\leq l$, we must be sure we can construct an independent set of size $\geq k$ from this vertex cover.

# Proof that Independent Set $\leq_P$ Vertex Cover

1. Arbitrary instance of INDEPENDENT SET: an undirected graph $G(V, E)$ and an integer $k$.
2. Let $|V| = n$.
3. Create an instance of VERTEX COVER: same undirected graph $G(V, E)$ and integer $n - k$.

# Proof that Independent Set $\leq_P$ Vertex Cover

1. Arbitrary instance of INDEPENDENT SET: an undirected graph $G(V, E)$ and an integer $k$.
2. Let $|V| = n$.
3. Create an instance of VERTEX COVER: same undirected graph $G(V, E)$ and integer $n - k$.
4. Claim: $G(V, E)$ has an independent set of size $\geq k$ iff $G(V, E)$ has a vertex cover of size $\leq n - k$.

   Proof: $S$ is an independent set in $G$ iff $V - S$ is a vertex cover in $G$.

# Proof that Independent Set $\leq_P$ Vertex Cover

1. Arbitrary instance of INDEPENDENT SET: an undirected graph $G(V, E)$ and an integer $k$.
2. Let $|V| = n$.
3. Create an instance of VERTEX COVER: same undirected graph $G(V, E)$ and integer $n - k$.
4. Claim: $G(V, E)$ has an independent set of size $\geq k$ iff $G(V, E)$ has a vertex cover of size $\leq n - k$.

   Proof: $S$ is an independent set in $G$ iff $V - S$ is a vertex cover in $G$.

- Same idea proves that VERTEX COVER $\leq_P$ INDEPENDENT SET

# Vertex Cover and Set Cover

- Independent Set is a "packing" problem: pack as many vertices as possible, subject to constraints (the edges).
- Vertex Cover is a "covering" problem: cover all edges in the graph with as few vertices as possible.
- There are more general covering problems.

Set Cover

**INSTANCE:** A set $U$ of $n$ elements, a collection $S_1, S_2, \ldots, S_m$ of subsets of $U$, and an integer $k$.

**QUESTION:** Is there a collection of $\leq k$ sets in the collection whose union is $U$?

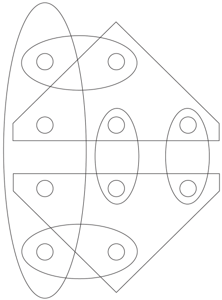

**Figure 8.2** An instance of the Set Cover Problem.

# Vertex Cover $\leq_P$ Set Cover

- Input to VERTEX COVER: an undirected graph $G(V, E)$ and an integer $k$.
- Let $|V| = n$.
- Create an instance $\{U, \{S_1, S_2, \ldots S_n\}\}$ of SET COVER where

# Vertex Cover $\leq_P$ Set Cover

- Input to VERTEX COVER: an undirected graph $G(V, E)$ and an integer $k$.
- Let $|V| = n$.
- Create an instance $\left\{U, \{S_1, S_2, \ldots S_n\}\right\}$ of SET COVER where
  - $U = E$,
  - for each vertex $i \in V$, create a set $S_i \subseteq U$ of the edges incident on $i$.

# Vertex Cover $\leq_P$ Set Cover

- Input to Vertex Cover: an undirected graph $G(V, E)$ and an integer $k$.
- Let $|V| = n$.
- Create an instance $\{U, \{S_1, S_2, \ldots S_n\}\}$ of Set Cover where
  - $U = E$,
  - for each vertex $i \in V$, create a set $S_i \subseteq U$ of the edges incident on $i$.
- Claim: $U$ can be covered with fewer than $k$ subsets iff $G$ has a vertex cover with at most $k$ nodes.
- Proof strategy:
  1. If $G(V, E)$ has a vertex cover of size at most $k$, then $U$ can be covered with at most $k$ subsets.
  2. If $U$ can be covered with at most $k$ subsets, then $G(V, E)$ has a vertex cover of size at most $k$.

# Boolean Satisfiability

- Abstract problems formulated in Boolean notation.
- Often used to specify problems, e.g., in AI.

# Boolean Satisfiability

- Abstract problems formulated in Boolean notation.
- Often used to specify problems, e.g., in AI.
- We are given a set $X = \{x_1, x_2, \ldots, x_n\}$ of $n$ Boolean variables.
- Each variable can take the value 0 or 1.
- A *term* is a variable $x_i$ or its negation $\overline{x_i}$.
- A *clause* of *length* $l$ is a disjunction of $l$ distinct terms $t_1 \vee t_2 \vee \cdots t_l$.
- A *truth assignment* for $X$ is a function $\nu : X \rightarrow \{0, 1\}$.
- An assignment *satisfies* a clause $C$ if it causes $C$ to evaluate to 1 under the rules of Boolean logic.
- An assignment *satisfies* a collection of clauses $C_1, C_2, \ldots C_k$ if it causes $C_1 \wedge C_2 \wedge \cdots C_k$ to evaluate to 1.
  - $\nu$ is a *satisfying assignment* with respect to $C_1, C_2, \ldots C_k$.
  - set of clauses $C_1, C_2, \ldots C_k$ is *satisfiable*.

# SAT and 3-SAT

Satisfiability Problem (SAT)

**INSTANCE:** A set of clauses $C_1, C_2, \ldots C_k$ over a set $X = \{x_1, x_2, \ldots x_n\}$ of $n$ variables.

**QUESTION:** Is there a satisfying truth assignment for $X$ with respect to $C$?

# SAT and 3-SAT

3-SATISFIABILITY PROBLEM (SAT)

**INSTANCE:** A set of clauses $C_1, C_2, \dots C_k$, each of length three, over a set $X = \{x_1, x_2, \dots x_n\}$ of $n$ variables.

**QUESTION:** Is there a satisfying truth assignment for $X$ with respect to $C$?

# SAT and 3-SAT

3-Satisfiability Problem (SAT)

**INSTANCE:** A set of clauses $C_1, C_2, \ldots C_k$, each of length three, over a set $X = \{x_1, x_2, \ldots x_n\}$ of $n$ variables.

**QUESTION:** Is there a satisfying truth assignment for $X$ with respect to $C$?

- SAT and 3-SAT are fundamental combinatorial search problems.
- We have to make $n$ independent decisions (the assignments for each variable) while satisfying a set of constraints.
- Satisfying each constraint in isolation is easy, but we have to make our decisions so that all constraints are satisfied simultaneously.

# Examples of 3-SAT

Example:

- $C_1 = x_1 \vee 0 \vee 0$
- $C_2 = x_2 \vee 0 \vee 0$
- $C_3 = \overline{x_1} \vee \overline{x_2}$

# Examples of 3-SAT

Example:

- $C_1 = x_1 \vee 0 \vee 0$
- $C_2 = x_2 \vee 0 \vee 0$
- $C_3 = \overline{x_1} \vee \overline{x_2}$

1. Is $C_1 \wedge C_2$ satisfiable?

# Examples of 3-SAT

Example:

- $C_1 = x_1 \vee 0 \vee 0$
- $C_2 = x_2 \vee 0 \vee 0$
- $C_3 = \overline{x_1} \vee \overline{x_2}$

1. Is $C_1 \wedge C_2$ satisfiable? Yes, by $x_1 = 1, x_2 = 1$.

# Examples of 3-SAT

Example:

- $C_1 = x_1 \vee 0 \vee 0$
- $C_2 = x_2 \vee 0 \vee 0$
- $C_3 = \overline{x_1} \vee \overline{x_2}$

1. Is $C_1 \wedge C_2$ satisfiable? Yes, by $x_1 = 1, x_2 = 1$.
2. Is $C_1 \wedge C_3$ satisfiable?

# Examples of 3-SAT

Example:

- $C_1 = x_1 \vee 0 \vee 0$
- $C_2 = x_2 \vee 0 \vee 0$
- $C_3 = \overline{x_1} \vee \overline{x_2}$

1. Is $C_1 \wedge C_2$ satisfiable? Yes, by $x_1 = 1, x_2 = 1$.
2. Is $C_1 \wedge C_3$ satisfiable? Yes, by $x_1 = 1, x_2 = 0$.

# Examples of 3-SAT

Example:

- $C_1 = x_1 \vee 0 \vee 0$
- $C_2 = x_2 \vee 0 \vee 0$
- $C_3 = \overline{x_1} \vee \overline{x_2}$

1. Is $C_1 \wedge C_2$ satisfiable? Yes, by $x_1 = 1, x_2 = 1$.
2. Is $C_1 \wedge C_3$ satisfiable? Yes, by $x_1 = 1, x_2 = 0$.
3. Is $C_2 \wedge C_3$ satisfiable?

# Examples of 3-SAT

Example:

- $C_1 = x_1 \vee 0 \vee 0$
- $C_2 = x_2 \vee 0 \vee 0$
- $C_3 = \overline{x_1} \vee \overline{x_2}$

1. Is $C_1 \wedge C_2$ satisfiable? Yes, by $x_1 = 1, x_2 = 1$.
2. Is $C_1 \wedge C_3$ satisfiable? Yes, by $x_1 = 1, x_2 = 0$.
3. Is $C_2 \wedge C_3$ satisfiable? Yes, by $x_1 = 0, x_2 = 1$.

# Examples of 3-SAT

Example:

- $C_1 = x_1 \vee 0 \vee 0$
- $C_2 = x_2 \vee 0 \vee 0$
- $C_3 = \overline{x_1} \vee \overline{x_2}$

1. Is $C_1 \wedge C_2$ satisfiable? Yes, by $x_1 = 1, x_2 = 1$.
2. Is $C_1 \wedge C_3$ satisfiable? Yes, by $x_1 = 1, x_2 = 0$.
3. Is $C_2 \wedge C_3$ satisfiable? Yes, by $x_1 = 0, x_2 = 1$.
4. Is $C_1 \wedge C_2 \wedge C_3$ satisfiable?

# Examples of 3-SAT

Example:

- $C_1 = x_1 \vee 0 \vee 0$
- $C_2 = x_2 \vee 0 \vee 0$
- $C_3 = \overline{x_1} \vee \overline{x_2}$

1. Is $C_1 \wedge C_2$ satisfiable? Yes, by $x_1 = 1, x_2 = 1$.
2. Is $C_1 \wedge C_3$ satisfiable? Yes, by $x_1 = 1, x_2 = 0$.
3. Is $C_2 \wedge C_3$ satisfiable? Yes, by $x_1 = 0, x_2 = 1$.
4. Is $C_1 \wedge C_2 \wedge C_3$ satisfiable? No.

# 3-SAT and Independent Set

- We want to prove 3-SAT $\leq_P$ Independent Set. Skip proof

# 3-SAT and Independent Set

- We want to prove 3-SAT $\leq_P$ Independent Set. Skip proof
- Two ways to think about 3-SAT:
  1. Make an independent 0/1 decision on each variable and succeed if we achieve one of three ways in which to satisfy each clause.
  2. Choose (at least) one term from each clause. Find a truth assignment that causes each chosen term to evaluate to 1. Ensure that no two terms selected *conflict*, i.e., select $x_i$ and $\overline{x_i}$.

# Proving 3-SAT $\leq_P$ Independent Set

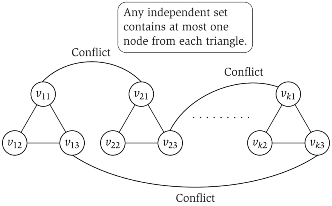


**Figure 8.3** The reduction from 3-SAT to Independent Set.

- We are given an instance of 3-SAT with $k$ clauses of length three over $n$ variables.
- Construct a graph $G(V, E)$ with $3k$ nodes.
  - For each clause $C_i, 1 \leq i \leq k$, add a triangle of three nodes $v_{i1}, v_{i2}, v_{i3}$ and three edges to $G$.
  - Label each node $v_{ij}, 1 \leq j \leq 3$ with the $j$th term in $C_i$.

# Proving 3-SAT $\leq_P$ Independent Set

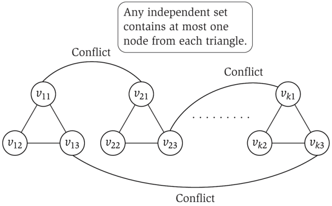


**Figure 8.3** The reduction from 3-SAT to Independent Set.

- We are given an instance of 3-SAT with $k$ clauses of length three over $n$ variables.
- Construct a graph $G(V, E)$ with $3k$ nodes.
  - For each clause $C_i, 1 \leq i \leq k$, add a triangle of three nodes $v_{i1}, v_{i2}, v_{i3}$ and three edges to $G$.
  - Label each node $v_{ij}, 1 \leq j \leq 3$ with the $j$th term in $C_i$.
  - Add an edge between each pair of nodes whose labels correspond to terms that conflict.

# Proving 3-SAT $\leq_P$ Independent Set

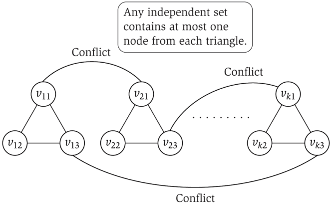


**Figure 8.3** The reduction from 3-SAT to Independent Set.

- Claim: $3\text{-SAT}$ instance is satisfiable iff $G$ has an independent set of size at least $k$.

# Proving 3-SAT $\leq_P$ Independent Set

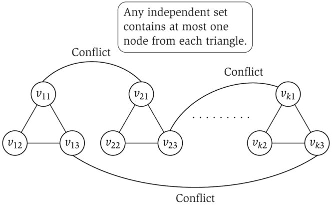


**Figure 8.3** The reduction from 3-SAT to Independent Set.

- Claim: $3\text{-SAT}$ instance is satisfiable iff $G$ has an independent set of size at least $k$.
- Satisfiable assignment $\rightarrow$ independent set of size $\geq k$:

# Proving 3-SAT $\leq_P$ Independent Set

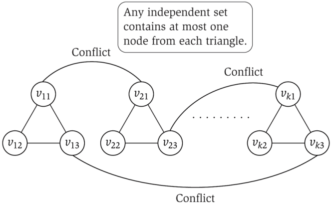


**Figure 8.3** The reduction from 3-SAT to Independent Set.

- Claim: $3\text{-SAT}$ instance is satisfiable iff $G$ has an independent set of size at least $k$.
- Satisfiable assignment $\rightarrow$ independent set of size $\geq k$: Each triangle in $G$ has at least one node whose label evaluates to 1. These nodes form an independent set of size $\geq k$. Why?

# Proving 3-SAT $\leq_P$ Independent Set

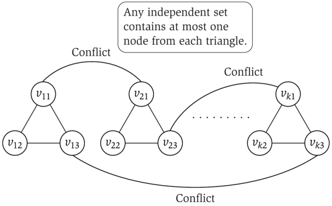


**Figure 8.3** The reduction from 3-SAT to Independent Set.

- Claim: $3\text{-SAT}$ instance is satisfiable iff $G$ has an independent set of size at least $k$.
- Satisfiable assignment $\rightarrow$ independent set of size $\geq k$: Each triangle in $G$ has at least one node whose label evaluates to 1. These nodes form an independent set of size $\geq k$. Why?
- Independent set of size $\geq k$ $\rightarrow$ satisfiable assignment:

# Proving 3-SAT $\leq_P$ Independent Set

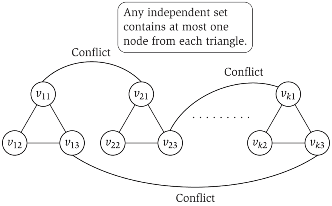


**Figure 8.3** The reduction from 3-SAT to Independent Set.

- Claim: $3\text{-SAT}$ instance is satisfiable iff $G$ has an independent set of size at least $k$.
- Satisfiable assignment $\rightarrow$ independent set of size $\geq k$: Each triangle in $G$ has at least one node whose label evaluates to 1. These nodes form an independent set of size $\geq k$. Why?
- Independent set of size $\geq k$ $\rightarrow$ satisfiable assignment: the size of this set is $k$. How do we construct a satisfying truth assignment from the nodes in the independent set?

# Transitivity of Reductions

- Claim: If $\text{Z} \leq_P \text{Y}$ and $\text{Y} \leq_P \text{X}$, then $\text{Z} \leq_P \text{X}$.

# Transitivity of Reductions

- Claim: If $\mathrm{Z} \leq_P \mathrm{Y}$ and $\mathrm{Y} \leq_P \mathrm{X}$, then $\mathrm{Z} \leq_P \mathrm{X}$.
- We have shown

$$\text{3-SAT} \leq_P \textsc{Independent Set} \leq_P \textsc{Vertex Cover} \leq_P \textsc{Set Cover}$$

# Finding vs. Certifying

- Is it easy to check if a given set of vertices in an undirected graph forms an independent set of size at least $k$?
- Is it easy to check if a particular truth assignment satisfies a set of clauses?

# Finding vs. Certifying

- Is it easy to check if a given set of vertices in an undirected graph forms an independent set of size at least $k$?
- Is it easy to check if a particular truth assignment satisfies a set of clauses?
- We draw a contrast between *finding* a solution and *checking* a solution (in polynomial time).
- Since we have not been able to develop efficient algorithms to solve many decision problems, let us turn our attention to whether we can check if a proposed solution is correct.

# Problems, Algorithms, and Strings

- Encode input to a computational problem as a finite binary string $s$ of length $|s|$.
- Identify a decision problem $X$ with the set of strings for which the answer is "yes",

# Problems, Algorithms, and Strings

- Encode input to a computational problem as a finite binary string $s$ of length $|s|$.
- Identify a decision problem $X$ with the set of strings for which the answer is "yes", e.g., $\text{PRIMES} = \{2, 3, 5, 7, 11, \ldots\}$.

# Problems, Algorithms, and Strings

- Encode input to a computational problem as a finite binary string $s$ of length $|s|$.
- Identify a decision problem $X$ with the set of strings for which the answer is "yes", e.g., $\text{PRIMES} = \{2, 3, 5, 7, 11, \ldots\}$.
- An algorithm $A$ for a decision problem receives an input string $s$ and returns $A(s) \in \{\texttt{yes}, \texttt{no}\}$.
- $A$ *solves* the problem $X$ if for every string $s$, $A(s) = \texttt{yes}$ iff $s \in X$.

# Problems, Algorithms, and Strings

- Encode input to a computational problem as a finite binary string $s$ of length $|s|$.
- Identify a decision problem $X$ with the set of strings for which the answer is "yes", e.g., $\text{PRIMES} = \{2, 3, 5, 7, 11, \ldots\}$.
- An algorithm $A$ for a decision problem receives an input string $s$ and returns $A(s) \in \{\texttt{yes}, \texttt{no}\}$.
- $A$ *solves* the problem $X$ if for every string $s$, $A(s) = \texttt{yes}$ iff $s \in X$.
- $A$ has a *polynomial running time* if there is a polynomial function $p(\cdot)$ such that for every input string $s$, $A$ terminates on $s$ in at most $O(p(|s|))$ steps,

# Problems, Algorithms, and Strings

- Encode input to a computational problem as a finite binary string $s$ of length $|s|$.
- Identify a decision problem $X$ with the set of strings for which the answer is "yes", e.g., $\text{PRIMES} = \{2, 3, 5, 7, 11, \ldots\}$.
- An algorithm $A$ for a decision problem receives an input string $s$ and returns $A(s) \in \{\texttt{yes}, \texttt{no}\}$.
- $A$ *solves* the problem $X$ if for every string $s$, $A(s) = \texttt{yes}$ iff $s \in X$.
- $A$ has a *polynomial running time* if there is a polynomial function $p(\cdot)$ such that for every input string $s$, $A$ terminates on $s$ in at most $O(p(|s|))$ steps, e.g., there is an algorithm such that $p(|s|) = |s|^8$ for $\text{PRIMES}$ (Agarwal, Kayal, Saxena, 2002).

# Problems, Algorithms, and Strings

- Encode input to a computational problem as a finite binary string $s$ of length $|s|$.
- Identify a decision problem $X$ with the set of strings for which the answer is "yes", e.g., $\mathrm{PRIMES} = \{2, 3, 5, 7, 11, \ldots\}$.
- An algorithm $A$ for a decision problem receives an input string $s$ and returns $A(s) \in \{\mathtt{yes}, \mathtt{no}\}$.
- $A$ *solves* the problem $X$ if for every string $s$, $A(s) = \mathtt{yes}$ iff $s \in X$.
- $A$ has a *polynomial running time* if there is a polynomial function $p(\cdot)$ such that for every input string $s$, $A$ terminates on $s$ in at most $O(p(|s|))$ steps, e.g., there is an algorithm such that $p(|s|) = |s|^8$ for $\mathrm{PRIMES}$ (Agarwal, Kayal, Saxena, 2002).
- $\mathcal{P}$: set of problems $X$ for which there is a polynomial time algorithm.

# Efficient Certification

- A "checking" algorithm for a decision problem $X$ has a different structure from an algorithm that solves $X$.
- Checking algorithm needs input string $s$ as well as a separate "certificate" string $t$ that contains evidence that $s \in X$.

# Efficient Certification

- A "checking" algorithm for a decision problem $X$ has a different structure from an algorithm that solves $X$.
- Checking algorithm needs input string $s$ as well as a separate "certificate" string $t$ that contains evidence that $s \in X$.
- An algorithm $B$ is an *efficient certifier* for a problem $X$ if
  1. $B$ is a polynomial time algorithm that takes two inputs $s$ and $t$ and
  2. there is a polynomial function $p$ so that for every string $s$, we have $s \in X$ iff there exists a string $t$ such that $|t| \leq p(|s|)$ and $B(s, t) = \text{yes}$.

# Efficient Certification

- A "checking" algorithm for a decision problem $X$ has a different structure from an algorithm that solves $X$.
- Checking algorithm needs input string $s$ as well as a separate "certificate" string $t$ that contains evidence that $s \in X$.
- An algorithm $B$ is an *efficient certifier* for a problem $X$ if
  1. $B$ is a polynomial time algorithm that takes two inputs $s$ and $t$ and
  2. there is a polynomial function $p$ so that for every string $s$, we have $s \in X$ iff there exists a string $t$ such that $|t| \leq p(|s|)$ and $B(s,t) = \texttt{yes}$.
- Certifier's job is to take a candidate short proof ($t$) that $s \in X$ and check in polynomial time whether $t$ is a correct proof.
- Certifier does not care about how to find these proofs.

# $\mathcal{NP}$

- $\mathcal{NP}$ is the set of all problems for which there exists an efficient certifier.
- 3-SAT $\in \mathcal{NP}$:

# $\mathcal{NP}$

- $\mathcal{NP}$ is the set of all problems for which there exists an efficient certifier.
- $3\text{-SAT} \in \mathcal{NP}$: $t$ is a truth assignment; $B$ evaluates the clauses with respect to the assignment.

# $\mathcal{NP}$

- $\mathcal{NP}$ is the set of all problems for which there exists an efficient certifier.
- $\text{3-SAT} \in \mathcal{NP}$: $t$ is a truth assignment; $B$ evaluates the clauses with respect to the assignment.
- $\text{Independent Set} \in \mathcal{NP}$:

# $\mathcal{NP}$

- $\mathcal{NP}$ is the set of all problems for which there exists an efficient certifier.
- $\text{3-SAT} \in \mathcal{NP}$: $t$ is a truth assignment; $B$ evaluates the clauses with respect to the assignment.
- $\textsc{Independent Set} \in \mathcal{NP}$: $t$ is a set of at least $k$ vertices; $B$ checks that no pair of these vertices are connected by an edge.

# $\mathcal{NP}$

- $\mathcal{NP}$ is the set of all problems for which there exists an efficient certifier.
- $\text{3-SAT} \in \mathcal{NP}$: $t$ is a truth assignment; $B$ evaluates the clauses with respect to the assignment.
- $\textsc{Independent Set} \in \mathcal{NP}$: $t$ is a set of at least $k$ vertices; $B$ checks that no pair of these vertices are connected by an edge.
- $\textsc{Set Cover} \in \mathcal{NP}$:

# $\mathcal{NP}$

- $\mathcal{NP}$ is the set of all problems for which there exists an efficient certifier.
- $\text{3-SAT} \in \mathcal{NP}$: $t$ is a truth assignment; $B$ evaluates the clauses with respect to the assignment.
- $\textsc{Independent Set} \in \mathcal{NP}$: $t$ is a set of at least $k$ vertices; $B$ checks that no pair of these vertices are connected by an edge.
- $\textsc{Set Cover} \in \mathcal{NP}$: $t$ is a list of $k$ sets from the collection; $B$ checks if their union is $U$.

# $\mathcal{P}$ vs. $\mathcal{NP}$

- Claim: $\mathcal{P} \subseteq \mathcal{NP}$.

# $\mathcal{P}$ vs. $\mathcal{NP}$

- Claim: $\mathcal{P} \subseteq \mathcal{NP}$. If $X \in P$, then there is a polynomial time algorithm $A$ that solves $X$. $B$ ignores $t$ and returns $A(s)$. Why is $B$ an efficient certifier?

# $\mathcal{P}$ vs. $\mathcal{NP}$

- Claim: $\mathcal{P} \subseteq \mathcal{NP}$. If $X \in P$, then there is a polynomial time algorithm $A$ that solves $X$. $B$ ignores $t$ and returns $A(s)$. Why is $B$ an efficient certifier?
- Is $\mathcal{P} = \mathcal{NP}$ or is $\mathcal{NP} - \mathcal{P} \neq \emptyset$?

# $\mathcal{P}$ vs. $\mathcal{NP}$

- Claim: $\mathcal{P} \subseteq \mathcal{NP}$. If $X \in P$, then there is a polynomial time algorithm $A$ that solves $X$. $B$ ignores $t$ and returns $A(s)$. Why is $B$ an efficient certifier?
- Is $\mathcal{P} = \mathcal{NP}$ or is $\mathcal{NP} - \mathcal{P} \neq \emptyset$? One of the major unsolved problems in computer science.

# $\mathcal{P}$ vs. $\mathcal{NP}$

- Claim: $\mathcal{P} \subseteq \mathcal{NP}$. If $X \in P$, then there is a polynomial time algorithm $A$ that solves $X$. $B$ ignores $t$ and returns $A(s)$. Why is $B$ an efficient certifier?
- Is $\mathcal{P} = \mathcal{NP}$ or is $\mathcal{NP} - \mathcal{P} \neq \emptyset$? One of the major unsolved problems in computer science.

# $\mathcal{NP}$-Complete Problems

- What are the hardest problems in $\mathcal{NP}$?

# $\mathcal{NP}$-Complete Problems

- What are the hardest problems in $\mathcal{NP}$?
- A problem $X$ is *$\mathcal{NP}$-Complete* if
  1. $X \in \mathcal{NP}$ and
  2. for every problem $Y \in \mathcal{NP}$, $Y \leq_P X$.

# $\mathcal{NP}$-Complete Problems

- What are the hardest problems in $\mathcal{NP}$?
- A problem $X$ is *$\mathcal{NP}$-Complete* if
  1. $X \in \mathcal{NP}$ and
  2. for every problem $Y \in \mathcal{NP}$, $Y \leq_P X$.
- Claim: Suppose $X$ is $\mathcal{NP}$-Complete. Then $X$ can be solved in polynomial time iff $\mathcal{P} = \mathcal{NP}$.

# $\mathcal{NP}$-Complete Problems

- What are the hardest problems in $\mathcal{NP}$?
- A problem $X$ is *$\mathcal{NP}$-Complete* if
  1. $X \in \mathcal{NP}$ and
  2. for every problem $Y \in \mathcal{NP}$, $Y \leq_P X$.
- Claim: Suppose $X$ is $\mathcal{NP}$-Complete. Then $X$ can be solved in polynomial time iff $\mathcal{P} = \mathcal{NP}$.
- Corollary: If there is any problem in $\mathcal{NP}$ that cannot be solved in polynomial time, then no $\mathcal{NP}$-Complete problem can be solved in polynomial time.

# $\mathcal{NP}$-Complete Problems

- What are the hardest problems in $\mathcal{NP}$?
- A problem $X$ is *$\mathcal{NP}$-Complete* if
  1. $X \in \mathcal{NP}$ and
  2. for every problem $Y \in \mathcal{NP}$, $Y \leq_P X$.
- Claim: Suppose $X$ is $\mathcal{NP}$-Complete. Then $X$ can be solved in polynomial time iff $\mathcal{P} = \mathcal{NP}$.
- Corollary: If there is any problem in $\mathcal{NP}$ that cannot be solved in polynomial time, then no $\mathcal{NP}$-Complete problem can be solved in polynomial time.
- Are there any $\mathcal{NP}$-Complete problems?
  1. Perhaps there are two problems $X_1$ and $X_2$ in $\mathcal{NP}$ such that there is no problem $X \in \mathcal{NP}$ where $X_1 \leq_P X$ and $X_2 \leq_P X$.
  2. Perhaps there is a sequence of problems $X_1, X_2, X_3, \ldots$ in $\mathcal{NP}$, each strictly harder than the previous one.

# Circuit Satisfiability

- Cook-Levin Theorem: CIRCUIT SATISFIABILITY is $\mathcal{NP}$-Complete.

# Circuit Satisfiability

- Cook-Levin Theorem: CIRCUIT SATISFIABILITY is $\mathcal{NP}$-Complete.
- A *circuit* $K$ is a labelled, directed acyclic graph such that
  1. the *sources* in $K$ are labelled with constants (0 or 1) or the name of a distinct variable (the *inputs* to the circuit).
  2. every other node is labelled with one Boolean operator $\wedge$, $\vee$, or $\neg$.
  3. a single node with no outgoing edges represents the *output* of $K$.

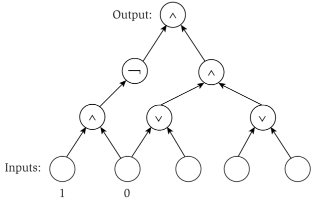


**Figure 8.4** A circuit with three inputs, two additional sources that have assigned truth values, and one output.

# Circuit Satisfiability

- Cook-Levin Theorem: CIRCUIT SATISFIABILITY is $\mathcal{NP}$-Complete.
- A *circuit* $K$ is a labelled, directed acyclic graph such that
  1. the *sources* in $K$ are labelled with constants (0 or 1) or the name of a distinct variable (the *inputs* to the circuit).
  2. every other node is labelled with one Boolean operator $\wedge$, $\vee$, or $\neg$.
  3. a single node with no outgoing edges represents the *output* of $K$.

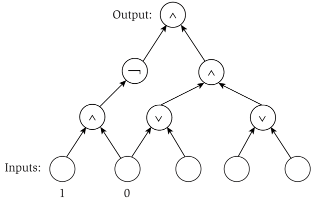


**Figure 8.4** A circuit with three inputs, two additional sources that have assigned truth values, and one output.

CIRCUIT SATISFIABILITY

**INSTANCE:** A circuit $K$.

**QUESTION:** Is there a truth assignment to the inputs that causes the output to have value 1?

Skip proof

# Proving Circuit Satisfiability is $\mathcal{NP}$-Complete

# Proving Circuit Satisfiability is $\mathcal{NP}$-Complete

- Take an arbitrary problem $X \in \mathcal{NP}$ and show that $X \leq_P$ Circuit Satisfiability.

# Proving Circuit Satisfiability is $\mathcal{NP}$-Complete

- Take an arbitrary problem $X \in \mathcal{NP}$ and show that $X \leq_P$ Circuit Satisfiability.
- Claim we will not prove: any algorithm that takes a fixed number $n$ of bits as input and produces a yes/no answer
  1. can be represented by an equivalent circuit and
  2. if the running time of the algorithm is polynomial in $n$, the size of the circuit is a polynomial in $n$.

# Proving Circuit Satisfiability is $\mathcal{NP}$-Complete

- Take an arbitrary problem $X \in \mathcal{NP}$ and show that $X \leq_P$ CIRCUIT SATISFIABILITY.
- Claim we will not prove: any algorithm that takes a fixed number $n$ of bits as input and produces a yes/no answer
  1. can be represented by an equivalent circuit and
  2. if the running time of the algorithm is polynomial in $n$, the size of the circuit is a polynomial in $n$.
- To show $X \leq_P$ CIRCUIT SATISFIABILITY, given an input $s$ of length $n$, we want to determine whether $s \in X$ using a black box that solves CIRCUIT SATISFIABILITY.

# Proving Circuit Satisfiability is $\mathcal{NP}$-Complete

- Take an arbitrary problem $X \in \mathcal{NP}$ and show that $X \leq_P$ Circuit Satisfiability.
- Claim we will not prove: any algorithm that takes a fixed number $n$ of bits as input and produces a yes/no answer
  1. can be represented by an equivalent circuit and
  2. if the running time of the algorithm is polynomial in $n$, the size of the circuit is a polynomial in $n$.
- To show $X \leq_P$ Circuit Satisfiability, given an input $s$ of length $n$, we want to determine whether $s \in X$ using a black box that solves Circuit Satisfiability.
- What do we know about $X$?

# Proving Circuit Satisfiability is $\mathcal{NP}$-Complete

- Take an arbitrary problem $X \in \mathcal{NP}$ and show that $X \leq_P$ Circuit Satisfiability.
- Claim we will not prove: any algorithm that takes a fixed number $n$ of bits as input and produces a yes/no answer
  1. can be represented by an equivalent circuit and
  2. if the running time of the algorithm is polynomial in $n$, the size of the circuit is a polynomial in $n$.
- To show $X \leq_P$ Circuit Satisfiability, given an input $s$ of length $n$, we want to determine whether $s \in X$ using a black box that solves Circuit Satisfiability.
- What do we know about $X$? It has an efficient certifier $B(\cdot, \cdot)$.

# Proving Circuit Satisfiability is $\mathcal{NP}$-Complete

- Take an arbitrary problem $X \in \mathcal{NP}$ and show that $X \leq_P$ Circuit Satisfiability.
- Claim we will not prove: any algorithm that takes a fixed number $n$ of bits as input and produces a yes/no answer
  1. can be represented by an equivalent circuit and
  2. if the running time of the algorithm is polynomial in $n$, the size of the circuit is a polynomial in $n$.
- To show $X \leq_P$ Circuit Satisfiability, given an input $s$ of length $n$, we want to determine whether $s \in X$ using a black box that solves Circuit Satisfiability.
- What do we know about $X$? It has an efficient certifier $B(\cdot, \cdot)$.
- To determine whether $s \in X$, we ask "Is there a string $t$ of length $p(n)$ such that $B(s, t) = \text{yes}$?"

# Proving Circuit Satisfiability is $\mathcal{NP}$-Complete

- To determine whether $s \in X$, we ask "Is there a string $t$ of length $p(|s|)$ such that $B(s, t) = \texttt{yes}$?"

# Proving Circuit Satisfiability is $\mathcal{NP}$-Complete

- To determine whether $s \in X$, we ask "Is there a string $t$ of length $p(|s|)$ such that $B(s,t) = \texttt{yes}$?"
- View $B(\cdot,\cdot)$ as an algorithm on $n + p(n)$ bits.
- Convert $B$ to a polynomial-sized circuit $K$ with $n + p(n)$ sources.
  1. First $n$ sources are hard-coded with the bits of $s$.
  2. The remaining $p(n)$ sources labelled with variables representing the bits of $t$.

# Proving Circuit Satisfiability is $\mathcal{NP}$-Complete

- To determine whether $s \in X$, we ask "Is there a string $t$ of length $p(|s|)$ such that $B(s, t) = \texttt{yes}$?"
- View $B(\cdot, \cdot)$ as an algorithm on $n + p(n)$ bits.
- Convert $B$ to a polynomial-sized circuit $K$ with $n + p(n)$ sources.
  1. First $n$ sources are hard-coded with the bits of $s$.
  2. The remaining $p(n)$ sources labelled with variables representing the bits of $t$.
- $s \in X$ iff there is an assignment of the input bits of $K$ that makes $K$ satisfiable.

# Example of Transformation to Circuit Satisfiability

- Does a graph $G$ on $n$ nodes have a two-node independent set?

# Example of Transformation to Circuit Satisfiability

- Does a graph $G$ on $n$ nodes have a two-node independent set?
- $s$ encodes the graph $G$ with $\binom{n}{2}$ bits.
- $t$ encodes the independent set with $n$ bits.
- Certifier needs to check if
  1. at least two bits in $t$ are set to 1 and
  2. no two bits in $t$ are set to 1 if they form the ends of an edge (the corresponding bit in $s$ is set to 1).

# Example of Transformation to Circuit Satisfiability

- Suppose $G$ contains three nodes $u, v$, and $w$ with $v$ connected to $u$ and $w$.

# Example of Transformation to Circuit Satisfiability

- Suppose $G$ contains three nodes $u$, $v$, and $w$ with $v$ connected to $u$ and $w$.

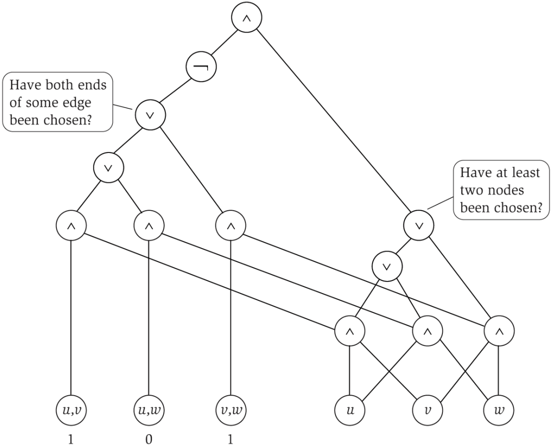


**Figure 8.5** A circuit to verify whether a 3-node graph contains a 2-node independent set.

# Proving Other Problems $\mathcal{NP}$-Complete

- A problem $X$ is $\mathcal{NP}$-Complete if
  1. $X \in \mathcal{NP}$ and
  2. for every problem $Y \in \mathcal{NP}$, $Y \leq_P X$.

# Proving Other Problems $\mathcal{NP}$-Complete

- A problem $X$ is $\mathcal{NP}$-Complete if
  1. $X \in \mathcal{NP}$ and
  2. for every problem $Y \in \mathcal{NP}$, $Y \leq_P X$.
- Claim: If $Z$ is $\mathcal{NP}$-Complete and $X \in \mathcal{NP}$ such that $Z \leq_P X$, then $X$ is $\mathcal{NP}$-Complete.

# Proving Other Problems $\mathcal{NP}$-Complete

- A problem $X$ is $\mathcal{NP}$-Complete if
  1. $X \in \mathcal{NP}$ and
  2. for every problem $Y \in \mathcal{NP}$, $Y \leq_P X$.
- Claim: If $Z$ is $\mathcal{NP}$-Complete and $X \in \mathcal{NP}$ such that $Z \leq_P X$, then $X$ is $\mathcal{NP}$-Complete.
- Given a new problem $X$, a general strategy for proving it $\mathcal{NP}$-Complete is

# Proving Other Problems $\mathcal{NP}$-Complete

- A problem $X$ is $\mathcal{NP}$-Complete if
    1. $X \in \mathcal{NP}$ and
    2. for every problem $Y \in \mathcal{NP}$, $Y \leq_P X$.
- Claim: If $Z$ is $\mathcal{NP}$-Complete and $X \in \mathcal{NP}$ such that $Z \leq_P X$, then $X$ is $\mathcal{NP}$-Complete.
- Given a new problem $X$, a general strategy for proving it $\mathcal{NP}$-Complete is
    1. Prove that $X \in \mathcal{NP}$.
    2. Select a problem $Z$ known to be $\mathcal{NP}$-Complete.
    3. Prove that $Z \leq_P X$.

# Proving Other Problems $\mathcal{NP}$-Complete

- A problem $X$ is $\mathcal{NP}$-Complete if
  1. $X \in \mathcal{NP}$ and
  2. for every problem $Y \in \mathcal{NP}$, $Y \leq_P X$.
- Claim: If $Z$ is $\mathcal{NP}$-Complete and $X \in \mathcal{NP}$ such that $Z \leq_P X$, then $X$ is $\mathcal{NP}$-Complete.
- Given a new problem $X$, a general strategy for proving it $\mathcal{NP}$-Complete is
  1. Prove that $X \in \mathcal{NP}$.
  2. Select a problem $Z$ known to be $\mathcal{NP}$-Complete.
  3. Prove that $Z \leq_P X$.
- If we use Karp reductions, we can refine the strategy:
  1. Prove that $X \in \mathcal{NP}$.
  2. Select a problem $Z$ known to be $\mathcal{NP}$-Complete.
  3. Consider an arbitrary instance $s_Z$ of problem $Z$. Show how to construct, in polynomial time, an instance $s_X$ of problem $X$ such that
     (a) If $s_Z \in Z$, then $s_X \in X$ and
     (b) If $s_X \in X$, then $s_z \in z$.