NP and Polynomial Time

Reductions

Lecture 22
April 16, 2015
Part I

NP, Showing Problems to be in NP
NP

- **P**: class of all languages that have a polynomial-time decision algorithm
- **NP**: class of all languages that have a *non-deterministic* polynomial-time algorithm

It makes sense to care about **P** since this is the class of problems for which we have efficient algorithms. Why should we care about **NP**? Is it a natural class? We will see that many useful, interesting, and important problems are in **NP** but we do not know whether there in **P** or not.
P: class of all languages that have a polynomial-time decision algorithm

NP: class of all languages that have a non-deterministic polynomial-time algorithm

It makes sense to care about P since this is the class of problems for which we have efficient algorithms. Why should we care about NP? Is it a natural class?
**P**: class of all languages that have a polynomial-time decision algorithm

**NP**: class of all languages that have a *non-deterministic* polynomial-time algorithm

It makes sense to care about **P** since this is the class of problems for which we have efficient algorithms. Why should we care about **NP**? Is it a natural class?

We will see that many useful, interesting, and important problems are in **NP** but we do not know whether there in **P** or not.
Some Classical Optimization Problems

- Maximum Independent Set
- Maximum Clique
- Minimum Vertex Cover
- Traveling Salesman Problem
- Knapsack Problems
- Integer Linear Programming
- ...

All of these optimization problems have a decision version which is an NP problem. And there are many, many other problems too.
Definition

Given undirected graph $G = (V, E)$ a subset of nodes $S \subseteq V$ is an independent set (also called a stable set) if for there are no edges between nodes in $S$. That is, if $u, v \in S$ then $(u, v) \not\in E$.

Some independent sets in graph above: $\{D\}, \{A, C\}, \{B, E, F\}$
Maximum Independent Set Problem

Input Graph $G = (V, E)$

Goal Find maximum sized independent set in $G$

MIS is an optimization problem. How do we cast it as a decision problem?
Decision version of Maximum Independent Set

Input  Graph \( G = (V, E) \) and integer \( k \) written as \( G, k \)

Question  Is there an independent set in \( G \) of size at least \( k \)?

The answer to \( \langle G, k \rangle \) is YES if \( G \) has an independent set of size at least \( k \). Otherwise the answer is NO. Sometimes we say \( \langle G, k \rangle \) is a YES instance or a NO instance.

The language associated with this decision problem is

\[
L_{\text{MIS}} = \{ \langle G, k \rangle \mid G \text{ has an independent set of size } \geq k \}
\]
MIS is in NP

$L_{\text{MIS}} = \{< G, k > | G \text{ has an independent set of size } \geq k\}$

A non-deterministic polynomial-time algorithm for $L_{\text{MIS}}$.

Input: a string $< G, k >$ encoding graph $G = (V, E)$ and integer $k$

1. Non-deterministically guess a subset $S \subseteq V$ of vertices
2. Verify (deterministically) that
   1. $S$ forms an independent set in $G$ by checking that there is no edge in $E$ between any pair of vertices in $S$
   2. $|S| \geq k$.
3. If $S$ passes the above two tests output YES Else NO
MIS is in NP

\[ L_{MIS} = \{ <G, k> | G \text{ has an independent set of size } \geq k \} \]

A non-deterministic polynomial-time algorithm for \( L_{MIS} \).

Input: a string \( <G, k> \) encoding graph \( G = (V, E) \) and integer \( k \)

1. Non-deterministically guess a subset \( S \subseteq V \) of vertices
2. Verify (deterministically) that
   1. \( S \) forms an independent set in \( G \) by checking that there is no edge in \( E \) between any pair of vertices in \( S \)
   2. \( |S| \geq k \).
3. If \( S \) passes the above two tests output YES Else NO

Key points:
- string encoding \( S, <S> \) has length polynomial in length of input \( <G, k> \)
- verification of guess is easily seen to be polynomial in length of \( <S> \) and \( <G, k> \).
Minimum Vertex Cover

**Definition**

Given undirected graph $G = (V, E)$ a subset of nodes $S \subseteq V$ is an **vertex cover** if every edge $(u, v)$ has at least one of its end points in $S$. That is, every edge is covered by $S$.

Examples of vertex covers in graph above:

![Graph diagram]

Chandra & Lenny (UIUC)  CS374  9 / 62  Spring 2015
Minimum Vertex Cover

**Input** Graph $G = (V, E)$

**Goal** Find minimum sized vertex cover in $G$

Decision version: given $G$ and $k$, does $G$ have a vertex cover of size at most $k$?

$$L_{VC} = \{ < G, k > | \ G \text{ has a vertex cover size } \leq k \}$$
Minimum Vertex Cover is in NP

$$L_{VC} = \{ < G, k > | G \text{ has a vertex cover size } \leq k \}$$

A non-deterministic polynomial-time algorithm for $$L_{VC}$$.

Input: a string $$< G, k >$$ encoding graph $$G = (V, E)$$ and integer $$k$$

1. Non-deterministically guess a subset $$S \subseteq V$$ of vertices
2. Verify (deterministically) that
   1. $$S$$ forms a vertex cover in $$G$$ by checking that for each edge $$(u, v) \in E$$ at least one of $$u, v$$ is in $$S$$
   2. $$|S| \leq k$$.
3. If $$S$$ passes the above two tests output YES Else NO
Minimum Vertex Cover is in NP

$$L_{VC} = \{<G, k> | G \text{ has a vertex cover size } \leq k\}$$

A non-deterministic polynomial-time algorithm for $$L_{VC}$$.

Input: a string $$<G, k>$$ encoding graph $$G = (V, E)$$ and integer $$k$$

1. Non-deterministically guess a subset $$S \subseteq V$$ of vertices
2. Verify (deterministically) that
   1. $$S$$ forms a vertex cover in $$G$$ by checking that for each edge $$(u, v) \in E$$ at least one of $$u, v$$ is in $$S$$
   2. $$|S| \leq k$$.
3. If $$S$$ passes the above two tests output YES Else NO

Key points:
- string encoding $$S$$, $$<S>$$ has length polynomial in length of input $$<G, k>$$
- verification of guess is easily seen to be polynomial in length of $$<S>$$ and $$<G, k>$$.
Given $n \times n$ sudoku puzzle, does it have a solution?
Certifier/Proof Interpretation of NP

\( L \in \text{NP} \) implies that there is a non-deterministic poly-time algorithm/TM \( M \) that accepts \( L \).

**Claim:** If \( L \in \text{NP} \) if and only if there is a poly-time TM \( M \) that accepts \( L \) with the following properties:

- On input \( w \), \( M \) first *non-deterministically* guesses a string \( y \) where \( |y| \leq p(|x|) \) for some fixed polynomial \( p() \).
- \( M \) then runs a *deterministic* poly-time TM \( M' \) on the string \( w\#y \) and accepts \( w \) if \( M' \) accepts \( w\#y \) and rejects \( w \) if \( M' \) rejects \( w\#y \).

Non-determinism is used to guess a “proof/solution” \( y \) for \( w \). Verifier/Certifier \( M' \) is a deterministic algorithm that checks if \( y \) is a valid solution for \( w \).
Certifier/Proof Interpretation of NP

\( L \in \text{NP} \) implies that there is a non-deterministic poly-time algorithm/TM \( M \) that accepts \( L \).

Alternate definition of \( \text{NP} \). \( L \in \text{NP} \) if and only if there is a deterministic TM/algorithm \( M \) and two polynomials \( p() \) and \( q() \) such that

- \( M \) runs in time \( p(|x|) \) where \( x \) is its input (hence efficient)
- if \( w \in L \) then there is a string \( y \) with \( |y| \leq q(|w|) \) such that \( M \) accepts \( w\#y \)
- if \( w \notin L \) then for every \( y \) with \( |y| \leq q(|w|) \), \( M \) on input \( w\#y \) rejects

\( \text{NP} \) is “natural” because there are plenty of problems where “verification” of solutions is easy. Hundreds of well-studied problems are in \( \text{NP} \).
Many natural problems we would like to solve are in \textbf{NP}.

Every problem in \textbf{NP} has an exponential time algorithm.

\textbf{P} \subseteq \textbf{NP}

Some problems in \textbf{NP} are in \textbf{P} (example, shortest path problem).

\textbf{Big Question:} Does every problem in \textbf{NP} have an efficient algorithm? Same as asking whether \textbf{P} = \textbf{NP}.

We don’t know the answer and many people believe that \textbf{P} \neq \textbf{NP}.

Given some new problem \textbf{L} that we want to solve we can

1. Prove that \textbf{L} \in \textbf{P}, that is develop an efficient algorithm for it or
2. Prove that \textbf{L} \in \textbf{NP} and proving that \textbf{L} \in \textbf{P} would imply that \textbf{P} = \textbf{NP} (that is, show that solving \textbf{L} would solve major open problems) or
3. Prove that \textbf{L} is even harder ...
Part II

Introduction to Reductions
A reduction from Problem $X$ to Problem $Y$ means (informally) that if we have an algorithm for Problem $Y$, we can use it to find an algorithm for Problem $X$. 
A reduction from Problem $X$ to Problem $Y$ means (informally) that if we have an algorithm for Problem $Y$, we can use it to find an algorithm for Problem $X$.

**Using Reductions**

1. We use reductions to find algorithms to solve problems.
Reductions

A reduction from Problem $X$ to Problem $Y$ means (informally) that if we have an algorithm for Problem $Y$, we can use it to find an algorithm for Problem $X$.

Using Reductions

1. We use reductions to find algorithms to solve problems.
2. We also use reductions to show that we can’t find algorithms for some problems. (We say that these problems are hard.)
For languages $L_X, L_Y$, a **reduction from $L_X$ to $L_Y$** is:

1. An algorithm . . .
2. Input: $w \in \Sigma^*$
3. Output: $w' \in \Sigma^*$
4. Such that:

$$w \in L_Y \iff w' \in L_X$$

(Actually, this is only one type of reduction, but this is the one we'll use most often.) There are other kinds of reductions.
Reductions for decision problems/languages

For languages $L_X, L_Y$, a reduction from $L_X$ to $L_Y$ is:

1. An algorithm . . .
2. Input: $w \in \Sigma^*$
3. Output: $w' \in \Sigma^*$
4. Such that:

$$w \in L_Y \iff w' \in L_X$$

(Actually, this is only one type of reduction, but this is the one we’ll use most often.) There are other kinds of reductions.
For decision problems $X, Y$, a reduction from $X$ to $Y$ is:

1. An algorithm . . .
2. Input: $I_X$, an instance of $X$.
4. Such that:
   
   $I_Y$ is YES instance of $Y \iff I_X$ is YES instance of $X$
Using reductions to solve problems

1. **R**: Reduction \( X \rightarrow Y \)

2. **A_Y**: algorithm for \( Y \):

   New algorithm for \( X \):
   
   \[
   A_X(I_X) = \begin{cases} 
   I_Y \Leftarrow R(I_X) \\
   A_Y(I_Y) 
   \end{cases}
   \]

   If \( R \) and \( A_Y \) polynomial-time \( \Rightarrow A_X \) polynomial-time.
Using reductions to solve problems

1. $R$: Reduction $X \rightarrow Y$
2. $A_Y$: algorithm for $Y$:
3. $\implies$ New algorithm for $X$:

$$A_X(I_X):$$

// $I_X$: instance of $X$.

$I_Y \leftarrow R(I_X)$

return $A_Y(I_Y)$
Using reductions to solve problems

1. \( \mathcal{R} \): Reduction \( X \rightarrow Y \)
2. \( A_Y \): algorithm for \( Y \):
3. \( \implies \) New algorithm for \( X \):

\[ A_X(I_X) : \]

// \( I_X \): instance of \( X \).
\[ I_Y \leftarrow \mathcal{R}(I_X) \]
\[ \text{return } A_Y(I_Y) \]

If \( \mathcal{R} \) and \( A_Y \) polynomial-time \( \implies \) \( A_X \) polynomial-time.
Comparing Problems

1. “Problem X is no harder to solve than Problem Y”.
2. If Problem X reduces to Problem Y (we write $X \leq Y$), then X cannot be harder to solve than Y.
3. $X \leq Y$:
   1. X is no harder than Y, or
   2. Y is at least as hard as X.
Part III

Examples of Reductions
Independent Sets and Cliques

Given a graph $G$, a set of vertices $V'$ is:
Independent Sets and Cliques

Given a graph $G$, a set of vertices $V'$ is:

1. **Independent set:** no two vertices of $V'$ connected by an edge.
Independent Sets and Cliques

Given a graph $G$, a set of vertices $V'$ is:

1. **independent set**: no two vertices of $V'$ connected by an edge.
2. **clique**: every pair of vertices in $V'$ is connected by an edge of $G$. 
Independent Sets and Cliques

Given a graph $G$, a set of vertices $V'$ is:

1. **independent set**: no two vertices of $V'$ connected by an edge.
2. **clique**: every pair of vertices in $V'$ is connected by an edge of $G$. 
Independent Sets and Cliques

Given a graph $G$, a set of vertices $V'$ is:

1. **independent set**: no two vertices of $V'$ connected by an edge.
2. **clique**: every pair of vertices in $V'$ is connected by an edge of $G$. 

![Graph with independent set and clique highlighted]
Independent Sets and Cliques

Given a graph $G$, a set of vertices $V'$ is:

1. **independent set**: no two vertices of $V'$ connected by an edge.
2. **clique**: every pair of vertices in $V'$ is connected by an edge of $G$. 

![Graph Diagram]
Problem: Independent Set

**Instance:** A graph $G$ and an integer $k$.

**Question:** Does $G$ has an independent set of size $\geq k$?
Problem: **Independent Set**

**Instance:** A graph \( G \) and an integer \( k \).

**Question:** Does \( G \) has an independent set of size \( \geq k \)?

Problem: **Clique**

**Instance:** A graph \( G \) and an integer \( k \).

**Question:** Does \( G \) has a clique of size \( \geq k \)?
Recall

For decision problems $X, Y$, a reduction from $X$ to $Y$ is:

1. An algorithm . . .
2. that takes $I_X$, an instance of $X$ as input . . .
3. and returns $I_Y$, an instance of $Y$ as output . . .
4. such that the solution (YES/NO) to $I_Y$ is the same as the solution to $I_X$. 
Reducing **Independent Set** to **Clique**

An instance of **Independent Set** is a graph $G$ and an integer $k$. 
Reducing **Independent Set** to **Clique**

An instance of **Independent Set** is a graph $G$ and an integer $k$. 

![Diagram of a graph with four vertices connected in a cycle](image)
Reducing **Independent Set** to **Clique**

An instance of **Independent Set** is a graph $G$ and an integer $k$.

Reduction given $< G, k >$ outputs $< \overline{G}, k >$ where $\overline{G}$ is the *complement* of $G$. $\overline{G}$ has an edge $(u, v)$ if and only if $(u, v)$ is not an edge of $G$. 

![Diagram of a graph with edges](image-url)
Reducing Independent Set to Clique

An instance of Independent Set is a graph $G$ and an integer $k$.

Reduction given $< G, k >$ outputs $< \overline{G}, k >$ where $\overline{G}$ is the complement of $G$. $\overline{G}$ has an edge $(u, v)$ if and only if $(u, v)$ is not an edge of $G$. 
Reducing **Independent Set** to **Clique**

An instance of **Independent Set** is a graph $G$ and an integer $k$.

Reduction given $< G, k >$ outputs $< \overline{G}, k >$ where $\overline{G}$ is the *complement* of $G$. $\overline{G}$ has an edge $(u, v)$ if and only if $(u, v)$ is not an edge of $G$.
Reducing **Independent Set** to **Clique**

An instance of **Independent Set** is a graph $G$ and an integer $k$.

Reduction given $< G, k >$ outputs $< \overline{G}, k >$ where $\overline{G}$ is the *complement* of $G$. $\overline{G}$ has an edge $(u, v)$ if and only if $(u, v)$ is not an edge of $G$. 

![Graph Diagram](attachment:graph.png)
Correctness of reduction

Lemma

G has an independent set of size k if and only if G has a clique of size k.

Proof.

Need to prove two facts:
G has independent set of size at least k implies that G has a clique of size at least k.
G has a clique of size at least k implies that G has an independent set of size at least k.
Easy to see both from the fact that S ⊆ V is an independent set in G if and only if S is a clique in G.
Independent Set and Clique

Independent Set $\leq$ Clique.
Independent Set and Clique

1. **Independent Set** ≤ **Clique**. What does this mean?

2. If have an algorithm for **Clique**, then we have an algorithm for **Independent Set**.
**Independent Set** and **Clique**

1. **Independent Set ≤ Clique.**
   What does this mean?

2. If have an algorithm for **Clique**, then we have an algorithm for **Independent Set**.

3. **Clique** is *at least as hard as Independent Set*. 
Independent Set and Clique

1. Independent Set $\leq$ Clique.
   What does this mean?

2. If have an algorithm for Clique, then we have an algorithm for Independent Set.

3. Clique is at least as hard as Independent Set.

4. Also... Independent Set is at least as hard as Clique.
Assume you can solve the **Clique** problem in $T(n)$ time. Then you can solve the **Independent Set** problem in

(A) $O(T(n))$ time.

(B) $O(n \log n + T(n))$ time.

(C) $O(n^2 T(n^2))$ time.

(D) $O(n^4 T(n^4))$ time.

(E) $O(n^2 + T(n^2))$ time.

(F) Does not matter - all these are polynomial if $T(n)$ is polynomial, which is good enough for our purposes.
A DFA $M$ is universal if it accepts every string. That is, $L(M) = \Sigma^*$, the set of all strings.
A DFA $M$ is universal if it accepts every string. That is, $L(M) = \Sigma^*$, the set of all strings.

**Problem (DFA universality)**

**Input:** A DFA $M$.
**Goal:** Is $M$ universal?
A DFA $M$ is universal if it accepts every string. That is, $L(M) = \Sigma^*$, the set of all strings.

**Problem (DFA universality)**

**Input:** A DFA $M$.

**Goal:** Is $M$ universal?

How do we solve DFA Universality?
A DFA $M$ is universal if it accepts every string. That is, $L(M) = \Sigma^*$, the set of all strings.

**Problem (DFA universality)**

**Input:** A DFA $M$.
**Goal:** Is $M$ universal?

How do we solve DFA Universality? We check if $M$ has any reachable non-final state. Alternatively, minimize $M$ to obtain $M'$ and see if $M'$ has a single state which is an accepting state.
An NFA $N$ is said to be universal if it accepts every string. That is, $L(N) = \Sigma^*$, the set of all strings.

**Problem (NFA universality)**

**Input:** A NFA $M$.

**Goal:** Is $M$ universal?

How do we solve NFA Universality?
An NFA $N$ is said to be universal if it accepts every string. That is, $L(N) = \Sigma^*$, the set of all strings.

Problem (NFA universality)

**Input:** A NFA $M$.

**Goal:** Is $M$ universal?

How do we solve NFA Universality?
Reduce it to DFA Universality?
An **NFA** \( N \) is said to be **universal** if it accepts every string. That is, \( L(N) = \Sigma^* \), the set of all strings.

**Problem (NFA universality)**

**Input:** A **NFA** \( M \).

**Goal:** Is \( M \) **universal**?

How do we solve **NFA Universality**?
Reduce it to **DFA Universality**?
Given an **NFA** \( N \), convert it to an equivalent **DFA** \( M \), and use the **DFA Universality** Algorithm.
An NFA $N$ is said to be universal if it accepts every string. That is, $L(N) = \Sigma^*$, the set of all strings.

**Problem (NFA universality)**

**Input:** A NFA $M$.

**Goal:** Is $M$ universal?

How do we solve NFA Universality? Reduce it to DFA Universality?

Given an NFA $N$, convert it to an equivalent DFA $M$, and use the DFA Universality Algorithm.

The reduction takes exponential time! Problem is known to be PSPACE-Complete and we do not expect a polynomial-time algorithm.
We say that an algorithm is **efficient** if it runs in polynomial-time.
We say that an algorithm is **efficient** if it runs in polynomial-time. To find efficient algorithms for problems, we are only interested in polynomial-time reductions. Reductions that take longer are not useful.
We say that an algorithm is **efficient** if it runs in polynomial-time.

To find efficient algorithms for problems, we are only interested in **polynomial-time** reductions. Reductions that take longer are not useful.

If we have a polynomial-time reduction from problem $X$ to problem $Y$ (we write $X \leq_P Y$), and a poly-time algorithm $A_Y$ for $Y$, we have a polynomial-time/efficient algorithm for $X$. 
Polynomial-time reductions

We say that an algorithm is **efficient** if it runs in polynomial-time.

To find efficient algorithms for problems, we are only interested in **polynomial-time** reductions. Reductions that take longer are not useful.

If we have a polynomial-time reduction from problem $X$ to problem $Y$ (we write $X \leq_P Y$), and a poly-time algorithm $A_Y$ for $Y$, we have a polynomial-time/efficient algorithm for $X$.

![Diagram](image-url)
Polynomial-time Reduction

A polynomial time reduction from a decision problem $X$ to a decision problem $Y$ is an algorithm $A$ that has the following properties:

1. given an instance $I_X$ of $X$, $A$ produces an instance $I_Y$ of $Y$
2. $A$ runs in time polynomial in $|I_X|$.
3. Answer to $I_X$ YES iff answer to $I_Y$ is YES.

Proposition

If $X \leq_P Y$ then a polynomial time algorithm for $Y$ implies a polynomial time algorithm for $X$.

Such a reduction is called a Karp reduction. Most reductions we will need are Karp reductions.
Let $X$ and $Y$ be two decision problems, such that $X$ can be solved in polynomial time, and $X \leq_P Y$. Then

(A) $Y$ can be solved in polynomial time.
(B) $Y$ can NOT be solved in polynomial time.
(C) If $Y$ is hard then $X$ is also hard.
(D) None of the above.
(E) All of the above.
Polynomial-time reductions and hardness

For decision problems $X$ and $Y$, if $X \leq_p Y$, and $Y$ has an efficient algorithm, $X$ has an efficient algorithm.
Polynomial-time reductions and hardness

For decision problems $X$ and $Y$, if $X \leq_p Y$, and $Y$ has an efficient algorithm, $X$ has an efficient algorithm.

If you believe that Independent Set does not have an efficient algorithm, why should you believe the same of Clique?
For decision problems $X$ and $Y$, if $X \leq_p Y$, and $Y$ has an efficient algorithm, $X$ has an efficient algorithm.

If you believe that Independent Set does not have an efficient algorithm, why should you believe the same of Clique?

Because we showed Independent Set $\leq_p$ Clique. If Clique had an efficient algorithm, so would Independent Set!
For decision problems $X$ and $Y$, if $X \leq_P Y$, and $Y$ has an efficient algorithm, $X$ has an efficient algorithm.

If you believe that Independent Set does not have an efficient algorithm, why should you believe the same of Clique?

Because we showed Independent Set $\leq_P$ Clique. If Clique had an efficient algorithm, so would Independent Set!

If $X \leq_P Y$ and $X$ does not have an efficient algorithm, $Y$ cannot have an efficient algorithm!
Proposition

Let $\mathcal{R}$ be a polynomial-time reduction from $X$ to $Y$. Then for any instance $I_X$ of $X$, the size of the instance $I_Y$ of $Y$ produced from $I_X$ by $\mathcal{R}$ is polynomial in the size of $I_X$. 

Proof.
$\mathcal{R}$ is a polynomial-time algorithm and hence on input $I_X$ of size $|I_X|$ it runs in time $p(|I_X|)$ for some polynomial $p()$.

$I_Y$ is the output of $\mathcal{R}$ on input $I_X$.

$\mathcal{R}$ can write at most $p(|I_X|)$ bits and hence $|I_Y| \leq p(|I_X|)$. 

Note: Converse is not true. A reduction need not be polynomial-time even if output of reduction is of size polynomial in its input.
Proposition

Let $R$ be a polynomial-time reduction from $X$ to $Y$. Then for any instance $I_X$ of $X$, the size of the instance $I_Y$ of $Y$ produced from $I_X$ by $R$ is polynomial in the size of $I_X$.

Proof.

$R$ is a polynomial-time algorithm and hence on input $I_X$ of size $|I_X|$ it runs in time $p(|I_X|)$ for some polynomial $p()$. $I_Y$ is the output of $R$ on input $I_X$. $R$ can write at most $p(|I_X|)$ bits and hence $|I_Y| \leq p(|I_X|)$. □
Proposition

Let $\mathcal{R}$ be a polynomial-time reduction from $X$ to $Y$. Then for any instance $I_X$ of $X$, the size of the instance $I_Y$ of $Y$ produced from $I_X$ by $\mathcal{R}$ is polynomial in the size of $I_X$.

Proof.

$\mathcal{R}$ is a polynomial-time algorithm and hence on input $I_X$ of size $|I_X|$ it runs in time $p(|I_X|)$ for some polynomial $p()$. $I_Y$ is the output of $\mathcal{R}$ on input $I_X$. $\mathcal{R}$ can write at most $p(|I_X|)$ bits and hence $|I_Y| \leq p(|I_X|)$.

Note: Converse is not true. A reduction need not be polynomial-time even if output of reduction is of size polynomial in its input.
A polynomial time reduction from a decision problem $X$ to a decision problem $Y$ is an algorithm $A$ that has the following properties:

2. $A$ runs in time polynomial in $|I_X|$. This implies that $|I_Y|$ (size of $I_Y$) is polynomial in $|I_X|$.
3. Answer to $I_X$ YES iff answer to $I_Y$ is YES.

**Proposition**

If $X \leq_P Y$ then a polynomial time algorithm for $Y$ implies a polynomial time algorithm for $X$.

Such a reduction is called a Karp reduction. Most reductions we will need are Karp reductions.
Transitivity of Reductions

**Proposition**

\[ X \leq_P Y \text{ and } Y \leq_P Z \text{ implies that } X \leq_P Z. \]

**Note:** \( X \leq_P Y \) does not imply that \( Y \leq_P X \) and hence it is very important to know the FROM and TO in a reduction.

To prove \( X \leq_P Y \) you need to show a reduction FROM \( X \) TO \( Y \). That is, show that an algorithm for \( Y \) implies an algorithm for \( X \).
Vertex Cover

Given a graph $G = (V, E)$, a set of vertices $S$ is:
Vertex Cover

Given a graph $G = (V, E)$, a set of vertices $S$ is:

1. A vertex cover if every $e \in E$ has at least one endpoint in $S$. 
Vertex Cover

Given a graph $G = (V, E)$, a set of vertices $S$ is:

1 A **vertex cover** if every $e \in E$ has at least one endpoint in $S$. 

![Graph example](image-url)
Given a graph $G = (V, E)$, a set of vertices $S$ is:

1. A vertex cover if every $e \in E$ has at least one endpoint in $S$. 

![Graph Diagram]
Given a graph $G = (V, E)$, a set of vertices $S$ is:

- A vertex cover if every $e \in E$ has at least one endpoint in $S$. 

![Graph Diagram]
## The Vertex Cover Problem

### Problem (Vertex Cover)

**Input:** A graph $G$ and integer $k$.

**Goal:** Is there a vertex cover of size $\leq k$ in $G$?
The **Vertex Cover** Problem

**Problem (Vertex Cover)**

**Input:** A graph $G$ and integer $k$.

**Goal:** Is there a vertex cover of size $\leq k$ in $G$?

Can we relate **Independent Set** and **Vertex Cover**?
Relationship between...
Vertex Cover and Independent Set

Proposition

Let $G = (V, E)$ be a graph. $S$ is an independent set if and only if $V \setminus S$ is a vertex cover.

Proof.

$(\Rightarrow)$ Let $S$ be an independent set

1. Consider any edge $uv \in E$.
2. Since $S$ is an independent set, either $u \not\in S$ or $v \not\in S$.
3. Thus, either $u \in V \setminus S$ or $v \in V \setminus S$.
4. $V \setminus S$ is a vertex cover.

$(\Leftarrow)$ Let $V \setminus S$ be some vertex cover:

1. Consider $u, v \in S$
2. $uv$ is not an edge of $G$, as otherwise $V \setminus S$ does not cover $uv$.
3. $\implies S$ is thus an independent set.
Independent Set $\leq_P$ Vertex Cover

1. $G$: graph with $n$ vertices, and an integer $k$ be an instance of the Independent Set problem.
Independent Set $\leq_P$ Vertex Cover

1. **G**: graph with $n$ vertices, and an integer $k$ be an instance of the Independent Set problem.

2. **G** has an independent set of size $\geq k$ iff **G** has a vertex cover of size $\leq n - k$.
**Independent Set \( \leq_p \text{Vertex Cover} \)**

1. \( G \): graph with \( n \) vertices, and an integer \( k \) be an instance of the **Independent Set** problem.

2. \( G \) has an independent set of size \( \geq k \) iff \( G \) has a vertex cover of size \( \leq n - k \)

3. \((G, k)\) is an instance of **Independent Set**, and \((G, n - k)\) is an instance of **Vertex Cover** with the same answer.
Independent Set $\leq_P$ Vertex Cover

1. $G$: graph with $n$ vertices, and an integer $k$ be an instance of the Independent Set problem.

2. $G$ has an independent set of size $\geq k$ iff $G$ has a vertex cover of size $\leq n - k$

3. $(G, k)$ is an instance of Independent Set, and $(G, n - k)$ is an instance of Vertex Cover with the same answer.

4. Therefore, Independent Set $\leq_P$ Vertex Cover. Also Vertex Cover $\leq_P$ Independent Set.
To prove that $X \leq_P Y$ you need to give an algorithm $A$ that:

1. Transforms an instance $I_X$ of $X$ into an instance $I_Y$ of $Y$.
2. Satisfies the property that answer to $I_X$ is YES iff $I_Y$ is YES.
   - typical easy direction to prove: answer to $I_Y$ is YES if answer to $I_X$ is YES
   - typical difficult direction to prove: answer to $I_X$ is YES if answer to $I_Y$ is YES (equivalently answer to $I_X$ is NO if answer to $I_Y$ is NO).
3. Runs in polynomial time.
Part IV

The Satisfiability Problem (SAT)
Definition

Consider a set of boolean variables $x_1, x_2, \ldots, x_n$.

1. A **literal** is either a boolean variable $x_i$ or its negation $\neg x_i$.

2. A **clause** is a disjunction of literals.
   For example, $x_1 \lor x_2 \lor \neg x_4$ is a clause.

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

Definition

Consider a set of boolean variables $x_1, x_2, \ldots, x_n$.

1. A **literal** is either a boolean variable $x_i$ or its negation $\neg x_i$.

2. A **clause** is a disjunction of literals. For example, $x_1 \lor x_2 \lor \neg x_4$ is a clause.

3. A **formula in conjunctive normal form (CNF)** is a propositional formula which is a conjunction of clauses.

   - $(x_1 \lor x_2 \lor \neg x_4) \land (x_2 \lor \neg x_3) \land x_5$ is a CNF formula.

4. A formula $\varphi$ is a **3CNF**: A CNF formula such that every clause has exactly 3 literals.

   - $(x_1 \lor x_2 \lor \neg x_4) \land (x_2 \lor \neg x_3 \lor x_1)$ is a 3CNF formula, but
   - $(x_1 \lor x_2 \lor \neg x_4) \land (x_2 \lor \neg x_3) \land x_5$ is not.
Satisfiability

Problem: **SAT**

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

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

Problem: **3SAT**

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

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

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

**Example**

1. $(x_1 \lor x_2 \lor \neg x_4) \land (x_2 \lor \neg x_3) \land x_5$ is satisfiable; take $x_1, x_2, \ldots, x_5$ to be all true.

2. $(x_1 \lor \neg x_2) \land (\neg x_1 \lor x_2) \land (\neg x_1 \lor \neg x_2) \land (x_1 \lor x_2)$ is not satisfiable.

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

(More on 2SAT in a bit...)
Importance of SAT and 3SAT

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

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

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

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

(D) $z \oplus x$.

(E) $(z \lor x) \land (\overline{z} \lor \overline{x}) \land (z \lor \overline{x}) \land (\overline{z} \lor x)$.
Given three bits $x, y, z$ which of the following SAT formulas is equivalent to the formula $z = x \land y$:

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

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

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

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

(E) $(z \lor x \lor y) \land (z \lor x \lor \bar{y}) \land (z \lor \bar{x} \lor y) \land (z \lor \bar{x} \lor \bar{y}) \land (\bar{z} \lor x \lor y) \land (\bar{z} \lor x \lor \bar{y}) \land (\bar{z} \lor \bar{x} \lor y) \land (\bar{z} \lor \bar{x} \lor \bar{y})$. 
Given three bits $x, y, z$ which of the following SAT formulas is equivalent to the formula $z = x \lor y$:

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

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

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

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

(E) $(\overline{z} \lor x \lor y) \land (z \lor \overline{x} \lor y) \land (z \lor x \lor \overline{y}) \land (z \lor \overline{x} \lor \overline{y})$. 
SAT $\leq_P$ 3SAT

How **SAT** is different from **3SAT**?

In **SAT** clauses might have arbitrary length: $1, 2, 3, \ldots$ variables:

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

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

How SAT is different from 3SAT?

In SAT clauses might have arbitrary length: 1, 2, 3, \ldots variables:

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

In 3SAT every clause must have \textit{exactly} 3 different literals.

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

Basic idea

1. Pad short clauses so they have 3 literals.
2. Break long clauses into shorter clauses.
3. Repeat the above till we have a 3CNF.
3SAT \leq_p SAT

1. 3SAT \leq_p SAT.
2. Because...
   A 3SAT instance is also an instance of SAT.
Claim

\[ \text{SAT} \leq_p \text{3SAT} \]
SAT \leq_P 3SAT

Claim

SAT \leq_P 3SAT.

Given \( \varphi \) a SAT formula we create a 3SAT formula \( \varphi' \) such that

1. \( \varphi \) is satisfiable iff \( \varphi' \) is satisfiable.
2. \( \varphi' \) can be constructed from \( \varphi \) in time polynomial in \( |\varphi| \).

Idea: if a clause of \( \varphi \) is not of length 3, replace it with several clauses of length exactly 3.
Claim

SAT \leq_p 3SAT.

Given \( \varphi \) a SAT formula we create a 3SAT formula \( \varphi' \) such that

1. \( \varphi \) is satisfiable iff \( \varphi' \) is satisfiable.
2. \( \varphi' \) can be constructed from \( \varphi \) in time polynomial in \( |\varphi| \).

Idea: if a clause of \( \varphi \) is not of length 3, replace it with several clauses of length exactly 3.
Challenge: Some of the clauses in $\varphi$ may have less or more than 3 literals. For each clause with $< 3$ or $> 3$ literals, we will construct a set of logically equivalent clauses.

Case clause with one literal: Let $c$ be a clause with a single literal (i.e., $c = \ell$). Let $u, v$ be new variables. Consider

$$c' = (\ell \lor u \lor v) \land (\ell \lor u \lor \neg v) \land (\ell \lor \neg u \lor v) \land (\ell \lor \neg u \lor \neg v).$$

Observe that $c'$ is satisfiable iff $c$ is satisfiable.
Case clause with 2 literals: Let $c = \ell_1 \lor \ell_2$. Let $u$ be a new variable. Consider

$$c' = (\ell_1 \lor \ell_2 \lor u) \land (\ell_1 \lor \ell_2 \lor \neg u).$$

Again $c$ is satisfiable iff $c'$ is satisfiable
Lemma

For any boolean formulas $X$ and $Y$ and $z$ a new boolean variable. Then

$$X \lor Y \text{ is satisfiable}$$

if and only if, $z$ can be assigned a value such that

$$\left( X \lor z \right) \land \left( Y \lor \neg z \right) \text{ is satisfiable}$$

(with the same assignment to the variables appearing in $X$ and $Y$).
Let $c = \ell_1 \lor \cdots \lor \ell_k$. Let $u_1, \ldots, u_{k-3}$ be new variables. Consider

$$c' = \left( \ell_1 \lor \ell_2 \lor u_1 \right) \land \left( \ell_3 \lor \neg u_1 \lor u_2 \right) \land \left( \ell_4 \lor \neg u_2 \lor u_3 \right) \land \cdots \land \left( \ell_{k-2} \lor \neg u_{k-4} \lor u_{k-3} \right) \land \left( \ell_{k-1} \lor \ell_k \lor \neg u_{k-3} \right).$$

**Claim**

$c$ is satisfiable iff $c'$ is satisfiable.

Another way to see it — reduce size of clause by one:

$$c' = \left( \ell_1 \lor \ell_2 \cdots \lor \ell_{k-2} \lor u_{k-3} \right) \land \left( \ell_{k-1} \lor \ell_k \lor \neg u_{k-3} \right).$$
An Example

Example

\[ \varphi = \left( \neg x_1 \lor \neg x_4 \right) \land \left( x_1 \lor \neg x_2 \lor \neg x_3 \right) \land \left( \neg x_2 \lor \neg x_3 \lor x_4 \lor x_1 \right) \land \left( x_1 \right) . \]

Equivalent form:

\[ \psi = \left( \neg x_1 \lor \neg x_4 \lor z \right) \land \left( \neg x_1 \lor \neg x_4 \lor \neg z \right) \]
An Example

Example

\[ \varphi = (\neg x_1 \lor \neg x_4) \land (x_1 \lor \neg x_2 \lor \neg x_3) \]
\[ \land (\neg x_2 \lor \neg x_3 \lor x_4 \lor x_1) \land (x_1) . \]

Equivalent form:

\[ \psi = (\neg x_1 \lor \neg x_4 \lor z) \land (\neg x_1 \lor \neg x_4 \lor \neg z) \]
\[ \land (x_1 \lor \neg x_2 \lor \neg x_3) \]
An Example

Example

\[ \varphi = (\neg x_1 \lor \neg x_4) \land (x_1 \lor \neg x_2 \lor \neg x_3) \land (\neg x_2 \lor \neg x_3 \lor x_4 \lor x_1) \land (x_1) . \]

Equivalent form:

\[ \psi = (\neg x_1 \lor \neg x_4 \lor z) \land (\neg x_1 \lor \neg x_4 \lor \neg z) \land (x_1 \lor \neg x_2 \lor \neg x_3) \land (\neg x_2 \lor \neg x_3 \lor y_1) \land (x_4 \lor x_1 \lor \neg y_1) \]
An Example

Example

\[ \varphi = (\neg x_1 \lor \neg x_4) \land (x_1 \lor \neg x_2 \lor \neg x_3) \land (\neg x_2 \lor \neg x_3 \lor x_4 \lor x_1) \land (x_1) . \]

Equivalent form:

\[ \psi = (\neg x_1 \lor \neg x_4 \lor z) \land (\neg x_1 \lor \neg x_4 \lor \neg z) \land (x_1 \lor \neg x_2 \lor \neg x_3) \land (\neg x_2 \lor \neg x_3 \lor y_1) \land (x_4 \lor x_1 \lor \neg y_1) \land (x_1 \lor u \lor v) \land (x_1 \lor u \lor \neg v) \land (x_1 \lor \neg u \lor v) \land (x_1 \lor \neg u \lor \neg v) . \]
Overall Reduction Algorithm
Reduction from \textbf{SAT} to \textbf{3SAT}

\begin{algorithm}
\textbf{ReduceSATTo3SAT}(\varphi):
\begin{algorithmic}
\State // \varphi: CNF formula.
  \For {each clause \( c \) of \varphi}
    \If {\( c \) does not have exactly 3 literals}
      \State construct \( c' \) as before
    \Else
      \State \( c' = c \)
    \EndIf
  \EndFor
  \State \( \psi \) is conjunction of all \( c' \) constructed in loop
\State \textbf{return} \textbf{Solver3SAT}(\psi)
\end{algorithmic}
\end{algorithm}

Correctness (informal)
\( \varphi \) is satisfiable iff \( \psi \) is satisfiable because for each clause \( c \), the new \textbf{3CNF} formula \( c' \) is logically equivalent to \( c \).
What about $2\text{SAT}$?

$2\text{SAT}$ can be solved in polynomial time! (specifically, linear time!)

No known polynomial time reduction from $\text{SAT}$ (or $3\text{SAT}$) to $2\text{SAT}$. If there was, then $\text{SAT}$ and $3\text{SAT}$ would be solvable in polynomial time.

Why the reduction from $3\text{SAT}$ to $2\text{SAT}$ fails?

Consider a clause $(x \lor y \lor z)$. We need to reduce it to a collection of $2\text{CNF}$ clauses. Introduce a face variable $\alpha$, and rewrite this as

$$\begin{align*}
(x \lor y \lor \alpha) \land (\neg \alpha \lor z) & \quad \text{(bad! clause with 3 vars)} \\
\text{or} \quad (x \lor \alpha) \land (\neg \alpha \lor y \lor z) & \quad \text{(bad! clause with 3 vars)}.
\end{align*}$$

(In animal farm language: $2\text{SAT}$ good, $3\text{SAT}$ bad.)
What about 2SAT?

A challenging exercise: Given a 2SAT formula show to compute its satisfying assignment...

(Hint: Create a graph with two vertices for each variable (for a variable $x$ there would be two vertices with labels $x = 0$ and $x = 1$). For ever 2CNF clause add two directed edges in the graph. The edges are implication edges: They state that if you decide to assign a certain value to a variable, then you must assign a certain value to some other variable.

Now compute the strong connected components in this graph, and continue from there...)