ILP, Reductions, and SAT

Lecture 20

April 20, 2021

Most slides are courtesy Prof. Chekuri

Outline

Integer Linear Programming (ILP)

Reductions

- Reductions and consequences: Algorithmic and hardness
- Poly-time reduction (Karp)
- Examples
- Turing reduction

SAT: Satisfiability problem, 3SAT, and equivalence.

Part I

Integer Linear Programming (ILP)

Integer Linear Programming

Problem

Find a vector $x \in Z^d$ **(integer values)** that

$$\begin{aligned} &\text{maximize} && \sum_{j=1}^{d} c_j x_j \\ &\text{subject to} && \sum_{j=1}^{d} a_{ij} x_j \leq b_i \quad \text{for } i = 1 \ldots n \end{aligned}$$

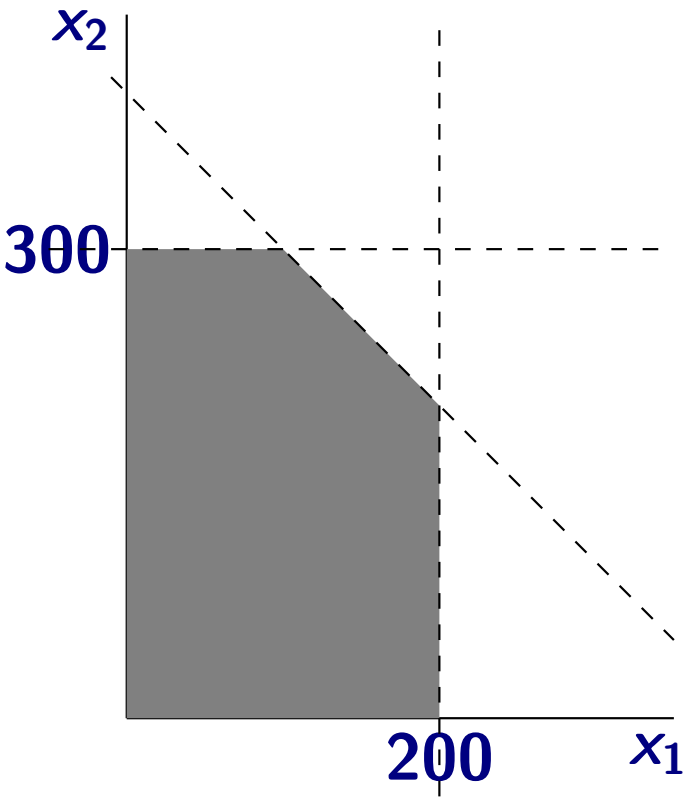
Input is matrix $A = (a_{ij}) \in \mathbb{R}^{n \times d}$, column vector $b = (b_i) \in \mathbb{R}^n$, and row vector $c = (c_j) \in \mathbb{R}^d$

Factory Example

$$\begin{array}{lll} \text{maximize} & & x_1 + 6x_2 \\ \text{subject to} & x_1 \leq 200 \quad x_2 \leq 300 & x_1 + x_2 \leq 400 \\ & x_1, x_2 \geq 0 & \end{array}$$

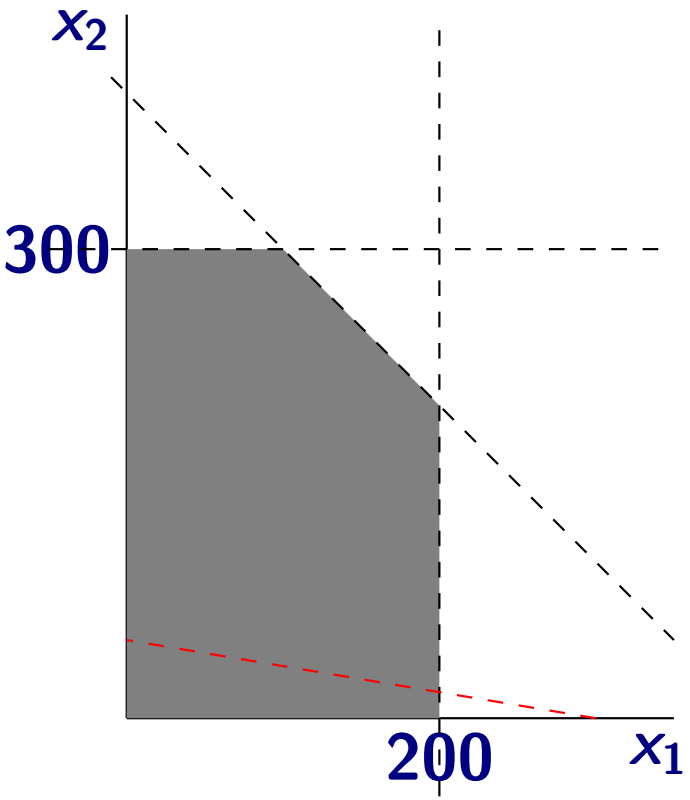
Suppose we want x_1, x_2 to be integer valued.

Factory Example Figure



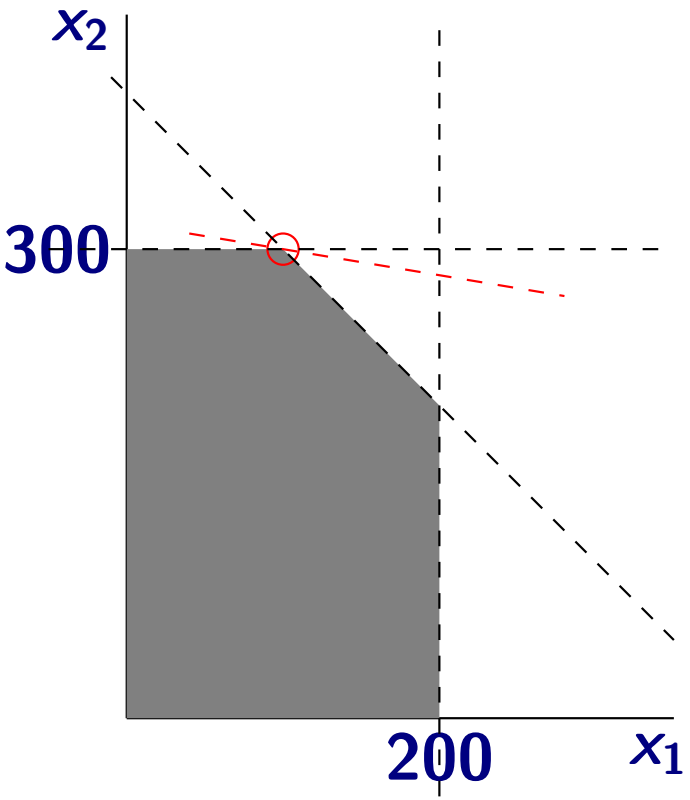
1. Feasible values of x_1 and x_2 are **integer points in shaded region**
2. Optimization function is a line; moving the line until it just leaves the final integer point in feasible region, gives optimal values

Factory Example Figure



1. Feasible values of x_1 and x_2 are **integer points in shaded region**
2. Optimization function is a line; moving the line until it just leaves the final integer point in feasible region, gives optimal values

Factory Example Figure



1. Feasible values of x_1 and x_2 are **integer points in shaded region**
2. Optimization function is a line; moving the line until it just leaves the final integer point in feasible region, gives optimal values

Integer Programming

Can model many difficult discrete optimization problems as integer programs!

Therefore integer programming is a hard problem. NP-hard.

Integer Programming

Can model many difficult discrete optimization problems as integer programs!

Therefore integer programming is a hard problem. NP-hard.

Can relax integer program to linear program and *approximate*.

Integer Programming

Can model many difficult discrete optimization problems as integer programs!

Therefore integer programming is a hard problem. NP-hard.

Can relax integer program to linear program and *approximate*.

Practice: integer programs are solved by a variety of methods

1. branch and bound
2. branch and cut
3. adding cutting planes
4. linear programming plays a fundamental role

Example: Maximum Independent Set

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 $(u, v) \notin E$.

Input Graph $G = (V, E)$

Goal Find maximum sized independent set in G

Example: Dominating Set

Definition

Given undirected graph $G = (V, E)$ a subset of nodes $S \subseteq V$ is a dominating set if for all $v \in V$, either $v \in S$ or a neighbor of v is in S.

Input Graph $G = (V, E)$, weights $w(v) \geq 0$ for $v \in V$

Goal Find minimum weight dominating set in G

Linear Programs with Integer Vertices

Suppose we know that for a linear program *all* vertices have integer coordinates.

Linear Programs with Integer Vertices

Suppose we know that for a linear program *all* vertices have integer coordinates.
Then solving linear program $\equiv$ solving integer program. Linear programs can be solved efficiently (polynomial time) and hence we get an integer solution for free!

Linear Programs with Integer Vertices

Suppose we know that for a linear program *all* vertices have integer coordinates.
Then solving linear program $\equiv$ solving integer program. Linear programs can be solved efficiently (polynomial time) and hence we get an integer solution for free!

Luck or *Structure*:

1. Linear program for flows with integer capacities have integer vertices
2. Linear program for matchings in bipartite graphs have integer vertices
3. A complicated linear program for matchings in general graphs have integer vertices.

All of above problems can hence be solved efficiently.

Linear Programs with Integer Vertices

Meta Theorem: A combinatorial optimization problem can be solved efficiently if and only if there is a linear program for problem with integer vertices.

Consequence of the Ellipsoid method for solving linear programming.

In a sense linear programming and other geometric generalizations such as convex programming are the most general problems that we can solve efficiently.

Summary

1. Linear Programming is a useful and powerful (modeling) problem.

Summary

1. Linear Programming is a useful and powerful (modeling) problem.
2. Can be solved in polynomial time. Practical solvers available commercially as well as in open source. Whether there is a strongly polynomial time algorithm is a major open problem.

Summary

1. Linear Programming is a useful and powerful (modeling) problem.
2. Can be solved in polynomial time. Practical solvers available commercially as well as in open source. Whether there is a strongly polynomial time algorithm is a major open problem.
3. Geometry and linear algebra are important to understand the structure of LP and in algorithm design. Vertex solutions imply that LPs have poly-sized optimum solutions. This implies that LP is in **NP**.

Summary

1. Linear Programming is a useful and powerful (modeling) problem.
2. Can be solved in polynomial time. Practical solvers available commercially as well as in open source. Whether there is a strongly polynomial time algorithm is a major open problem.
3. Geometry and linear algebra are important to understand the structure of LP and in algorithm design. Vertex solutions imply that LPs have poly-sized optimum solutions. This implies that LP is in **NP**.
4. Integer Programming in **NP-Complete**. LP-based techniques critical in heuristically solving integer programs.

Part II

Reductions

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.

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.)

Example 1: Bipartite Matching and Flows

How do we solve the **Bipartite Matching** Problem?

Given a bipartite graph $G = (U \cup V, E)$ and number k, does G have a matching of size $\geq k$?

Solution

Reduce it to **Max-Flow**. G has a matching of size $\geq k$ iff there is a flow from s to t of value $\geq k$ in the auxiliary graph G'.

Types of Problems

Decision, Search, and Optimization

1. **Decision problem**. Example: given n, is n prime?.
2. **Search problem**. Example: given n, find a factor of n if it exists.
3. **Optimization problem**. Example: find the smallest prime factor of n.

Optimization and Decision problems

For max flow...

Problem (Max-Flow optimization version)

Given an instance G of network flow, find the maximum flow between s and t.

Problem (Max-Flow decision version)

Given an instance G of network flow and a parameter K, is there a flow in G, from s to t, of value at least K?

While using reductions and comparing problems, we typically work with the decision versions. Decision problems have Yes/No answers. This makes them easy to work with.

Problems vs Instances

1. A problem Π consists of an **infinite** collection of inputs $\{I_1, I_2, \ldots, \}$. Each input is referred to as an instance.

Problems vs Instances

1. A problem Π consists of an **infinite** collection of inputs $\{I_1, I_2, \ldots, \}$. Each input is referred to as an instance.

Example

Max-Flow is a problem. While a graph G with edge-capacities, two vertices s, t, and an integer k constitutes an instance.

Problems vs Instances

1. A problem Π consists of an **infinite** collection of inputs $\{I_1, I_2, \ldots, \}$. Each input is referred to as an instance.

Example

Max-Flow is a problem. While a graph G with edge-capacities, two vertices s, t, and an integer k constitutes an instance.

2. The size of an instance I is the number of bits in its representation.

Problems vs Instances

1. A problem Π consists of an **infinite** collection of inputs $\{I_1, I_2, \ldots, \}$. Each input is referred to as an instance.

Example

Max-Flow is a problem. While a graph G with edge-capacities, two vertices s, t, and an integer k constitutes an instance.

2. The size of an instance I is the number of bits in its representation.
3. For an instance I, $sol(I)$ is a set of feasible solutions to I.
4. For optimization problems each solution $s \in sol(I)$ has an associated value.

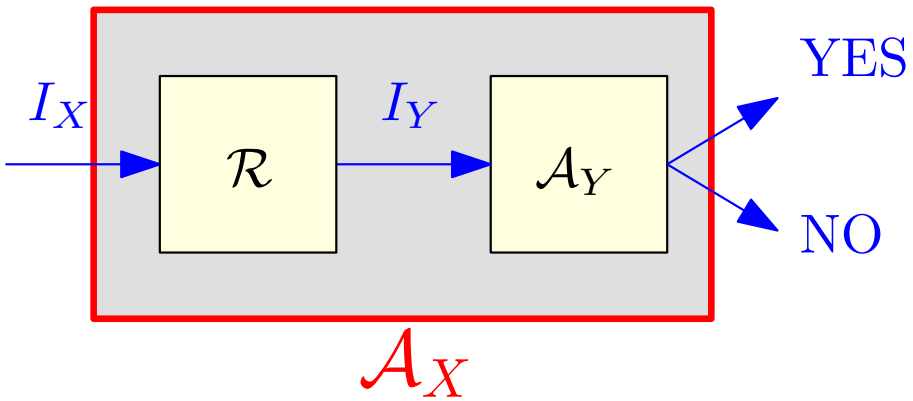
Using reductions to solve problems

1. $\mathcal{R}$: Reduction $X \rightarrow Y$
2. $\mathcal{A}_Y$: algorithm for Y

Using reductions to solve problems

1. $\mathcal{R}$: Reduction $X \rightarrow Y$
2. $\mathcal{A}_Y$: algorithm for Y
3. $\Longrightarrow$ New algorithm for X:

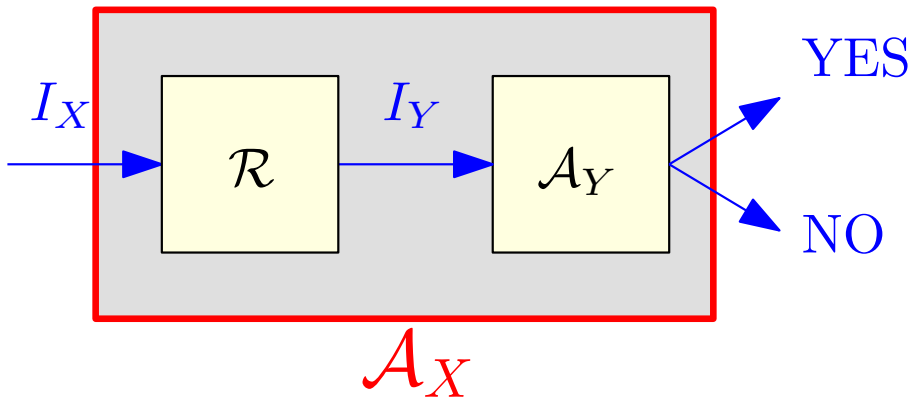
```
A_X(I_X):
        // I_X: instance of X.
        I_Y ← R(I_X)
        return A_Y(I_Y)
```



Using reductions to solve problems

1. $\mathcal{R}$: Reduction $X \rightarrow Y$
2. $\mathcal{A}_Y$: algorithm for Y
3. $\Longrightarrow$ New algorithm for X:

```
A_X(I_X):
        // I_X: instance of X.
        I_Y ← R(I_X)
        return A_Y(I_Y)
```



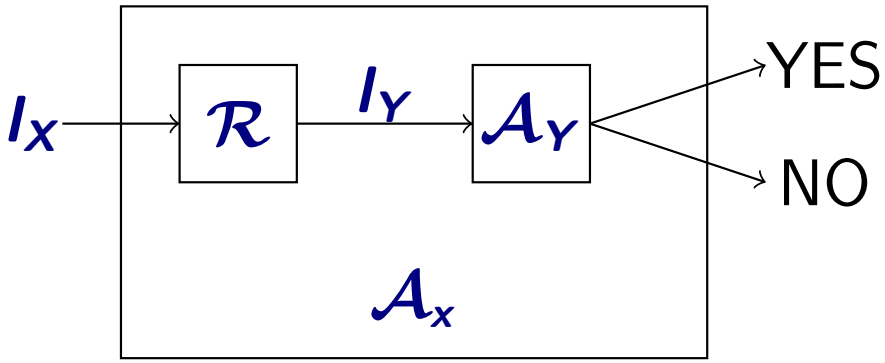
If $\mathcal{R}$ and $\mathcal{A}_Y$ polynomial-time $\Longrightarrow$ $\mathcal{A}_X$ polynomial-time.

Polynomial-time Reductions

Efficient Algorithm: runs in polynomial-time.

To find efficient algorithms for problems, only polynomial-time reductions are useful.

If reduction $\mathcal{R}$: $X \rightarrow Y$ is poly-time computable then denote $X \leq_P Y$

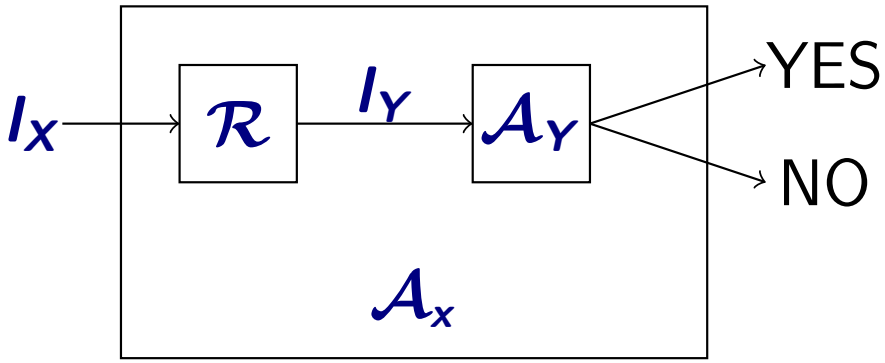


Polynomial-time Reductions

Efficient Algorithm: runs in polynomial-time.

To find efficient algorithms for problems, only polynomial-time reductions are useful.

If reduction $\mathcal{R}$: $X \rightarrow Y$ is poly-time computable then denote $X \leq_P Y$



Claim.

- If $\mathcal{A}_Y$ poly-time, then $\mathcal{A}_X$ is poly-time.

Polynomial-time reductions and instance sizes

Proposition

Let $\mathcal{R}$ be a polynomial-time algorithm reducing X to Y. Then for any instance I_X of X, if $I_Y = \mathcal{R}(I_X)$ then $|I_Y|$ 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 $p()$.

Polynomial-time reductions and instance sizes

Proposition

Let $\mathcal{R}$ be a polynomial-time algorithm reducing X to Y. Then for any instance I_X of X, if $I_Y = \mathcal{R}(I_X)$ then $|I_Y|$ 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.

Polynomial-time reductions and instance sizes

Proposition

Let $\mathcal{R}$ be a polynomial-time algorithm reducing X to Y. Then for any instance I_X of X, if $I_Y = \mathcal{R}(I_X)$ then $|I_Y|$ 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 $|I_Y| \leq p(|I_X|)$. □

Polynomial-time reductions and instance sizes

Proposition

Let $\mathcal{R}$ be a polynomial-time algorithm reducing X to Y. Then for any instance I_X of X, if $I_Y = \mathcal{R}(I_X)$ then $|I_Y|$ 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 $|I_Y| \leq p(|I_X|)$. □

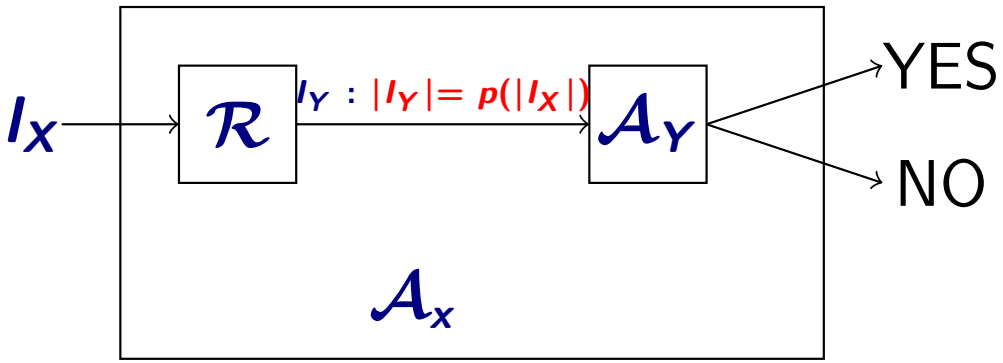
Note: Converse is not true. A reduction need not be polynomial-time even if output of reduction is of size polynomial in its input.

Polynomial-time Reductions

Efficient Algorithm: runs in polynomial-time.

To find efficient algorithms for problems, only polynomial-time reductions are useful.

If reduction $\mathcal{R}$: $X \rightarrow Y$ is poly-time computable then denote $X \leq_P Y$



Claim.

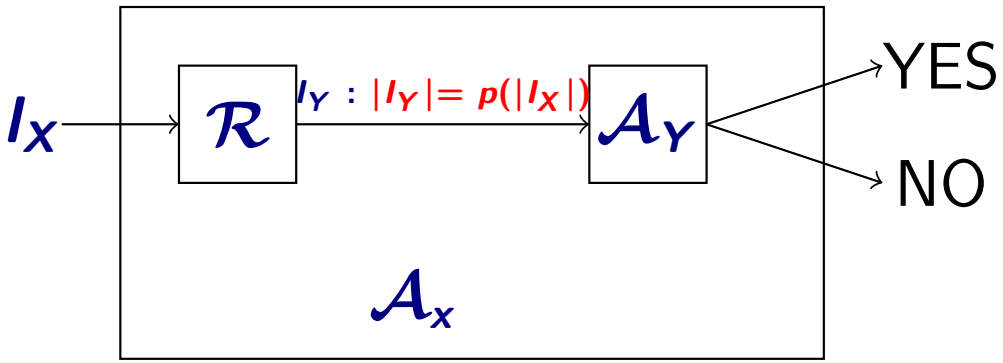
- If $\mathcal{A}_Y$ poly-time, then $\mathcal{A}_X$ is poly-time.

Polynomial-time Reductions

Efficient Algorithm: runs in polynomial-time.

To find efficient algorithms for problems, only polynomial-time reductions are useful.

If reduction $\mathcal{R}$: $X \rightarrow Y$ is poly-time computable then denote $X \leq_P Y$



Claim.

- If $\mathcal{A}_Y$ poly-time, then $\mathcal{A}_X$ is poly-time.
- If X is hard (no poly-time alogirthm), then so is Y ($\mathcal{A}_Y$ can not be poly-time)

Reductions again...

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.

Transitivity of Reductions

Proposition

$X \leq_P Y$ and $Y \leq_P Z$ 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 $X \leq_P Z$ you need to show a reduction FROM X TO Z
In other words show that an algorithm for Z implies an algorithm for X.

Using Reductions to show Hardness

We say that a problem is "hard" if there is no polynomial-time algorithm known for it (and it is believed that such an algorithm does not exist).

To show that Y is a hard problem:

- Start with an existing "hard" problem X
- Prove that $X \leq_P Y$
- Then we have shown that Y is a "hard" problem

Examples of hard problems

Problems

1. **SAT**
2. **3SAT**
3. **Independent Set** and **Clique**
4. **Vertex Cover**
5. **Set Cover**
6. **Hamilton Cycle**
7. **Knapsack** and **Subset Sum** and **Partition**
8. **Integer Programming**
9. ...

Part III

Examples of Reductions

Independent Sets and Cliques

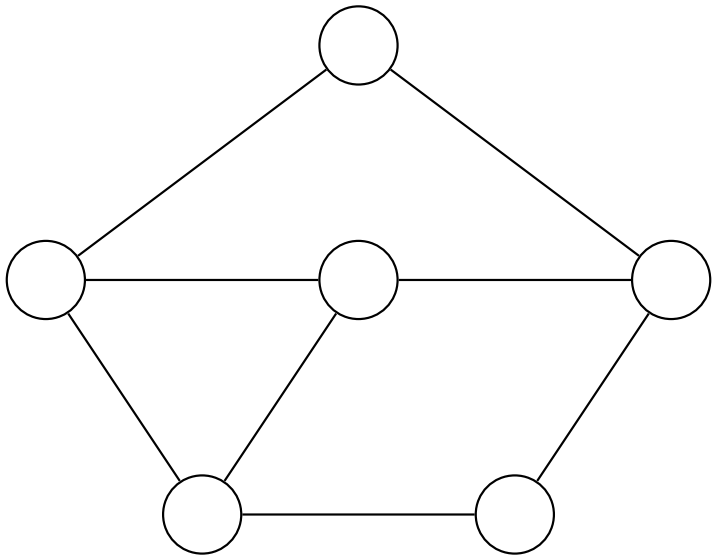
Given a graph $G = (V, E)$, a set of vertices $V' \subseteq V$ is:

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 $G = (V, E)$, a set of vertices $V' \subseteq V$ is:

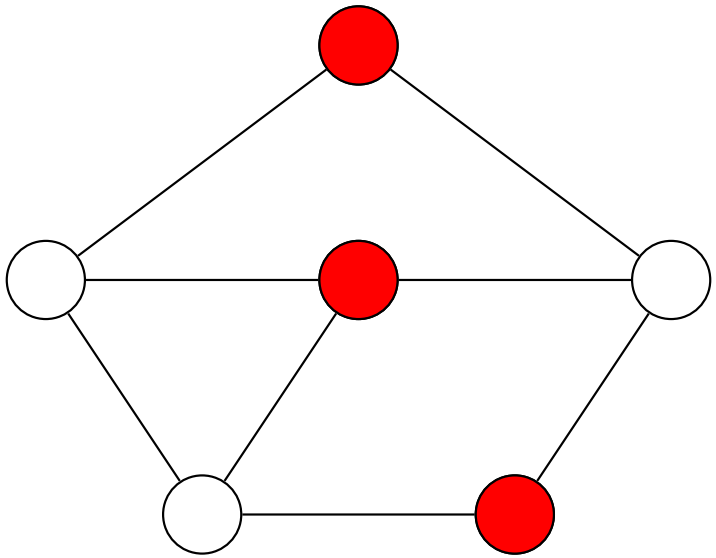
1. **independent set**: no two vertices of V' connected by an edge.
2. **clique**: *every* pair of vertices in V' is connected by an edge of G.



Independent Sets and Cliques

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

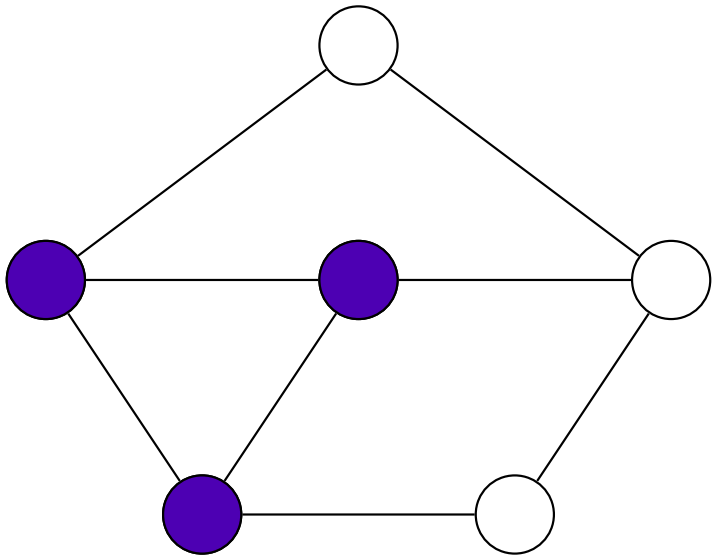
1. **independent set**: no two vertices of V' connected by an edge.
2. **clique**: *every* pair of vertices in V' is connected by an edge of G.



Independent Sets and Cliques

Given a graph $G = (V, E)$, a set of vertices $V' \subseteq V$ 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.



The **Independent Set** and **Clique** Problems

Problem: Independent Set

Instance: A graph G and an integer k.
Question: Does G has an independent set of size $\geq k$?

Problem: Clique

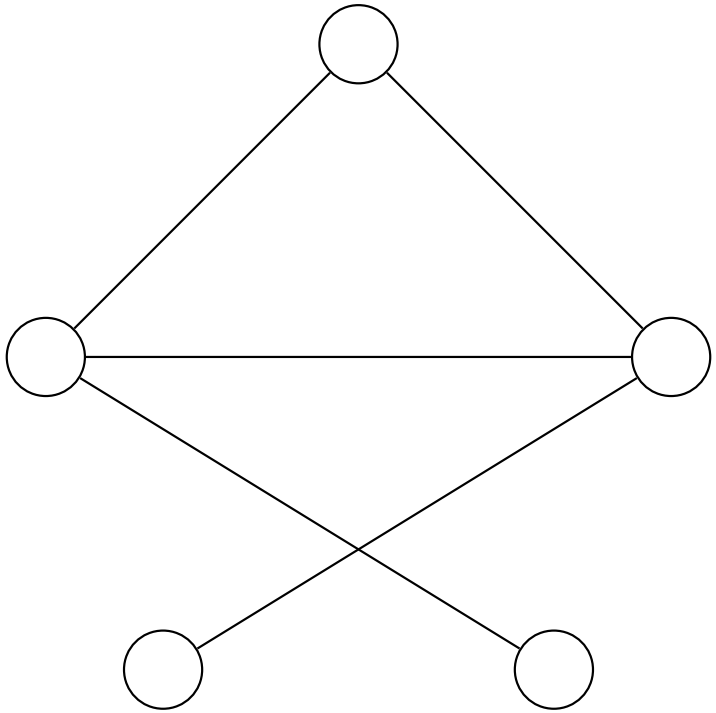
Instance: A graph G and an integer k.
Question: Does G has a clique of size $\geq k$?

Reducing **Independent Set** to **Clique**

Instance of **Independent Set**: graph G and an integer k.

Reducing **Independent Set** to **Clique**

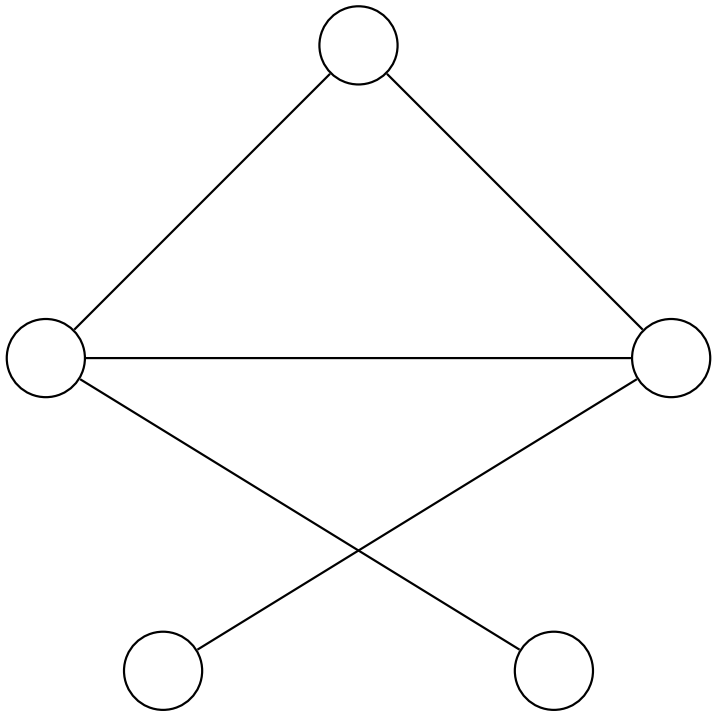
Instance of **Independent Set**: graph G and an integer k.



Reducing **Independent Set** to **Clique**

Instance of **Independent Set**: graph G and an integer k.

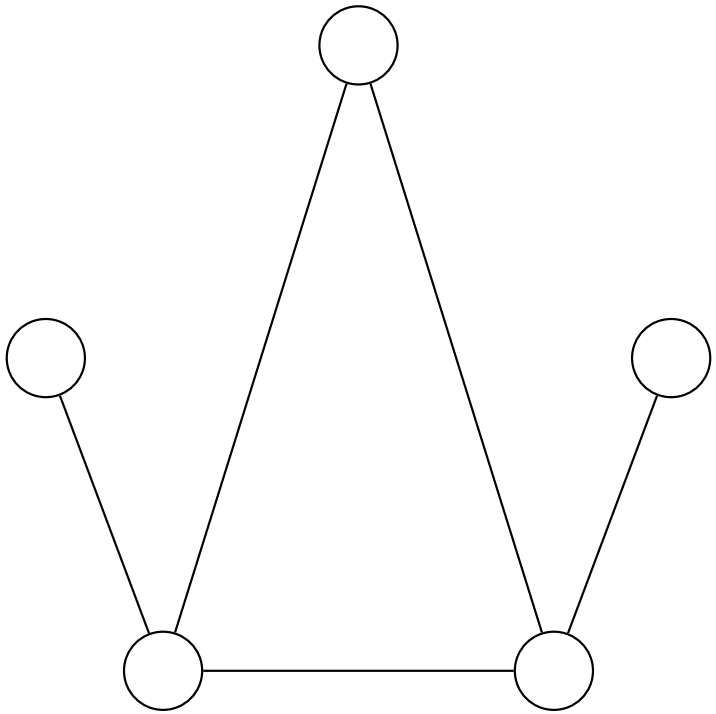
$\overline{G}$ (complement of G): where (u, v) is an edge iff (u, v) is not an edge of G.
Instance of **Clique**: graph $\overline{G}$ and integer k.



Reducing **Independent Set** to **Clique**

Instance of **Independent Set**: graph G and an integer k.

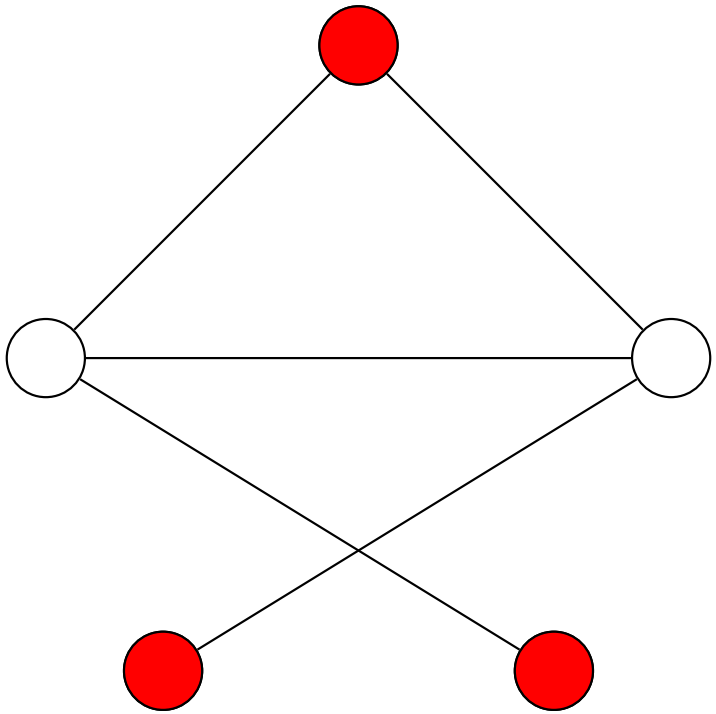
$\overline{G}$ (complement of G): where (u, v) is an edge iff (u, v) is not an edge of G.
Instance of **Clique**: graph $\overline{G}$ and integer k.



Reducing **Independent Set** to **Clique**

Instance of **Independent Set**: graph G and an integer k.

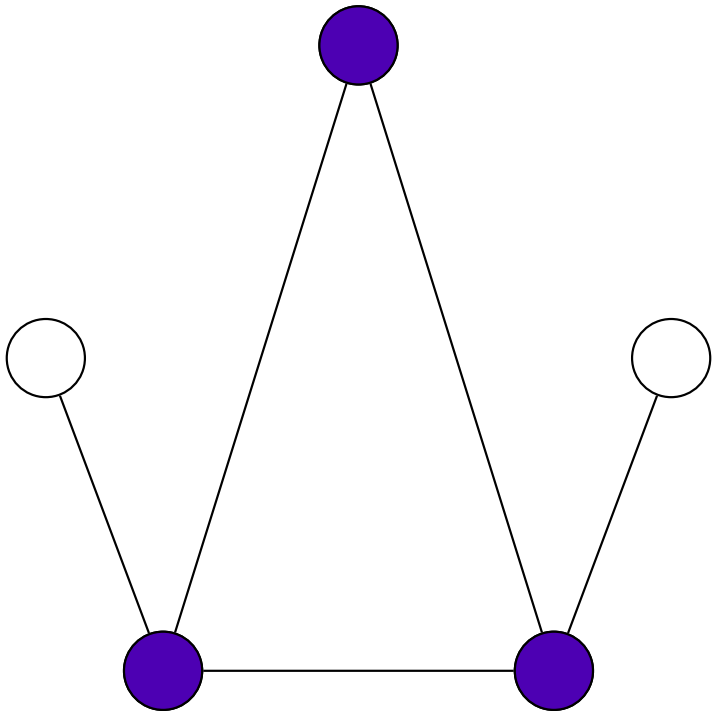
$\overline{G}$ (complement of G): where (u, v) is an edge iff (u, v) is not an edge of G.
Instance of **Clique**: graph $\overline{G}$ and integer k.



Reducing **Independent Set** to **Clique**

Instance of **Independent Set**: graph G and an integer k.

$\overline{G}$ (complement of G): where (u, v) is an edge iff (u, v) is not an edge of G.
Instance of **Clique**: graph $\overline{G}$ and integer k.



Independent Set and Clique

1. **Independent Set** $\leq_P$ **Clique**.
   What does this mean?
2. If we 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** $\leq_P$ **Clique**.
   What does this mean?
2. If we have an algorithm for **Clique**, then we have an algorithm for **Independent Set**.
3. **Clique** is *at least as hard as* **Independent Set**.
4. Does **Clique** $\leq_P$ **Independent Set**?

Independent Set and Clique

1. **Independent Set** $\leq_P$ **Clique**.
   What does this mean?
2. If we have an algorithm for **Clique**, then we have an algorithm for **Independent Set**.
3. **Clique** is *at least as hard as* **Independent Set**.
4. Does **Clique** $\leq_P$ **Independent Set**?

   YES!

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

Vertex Cover

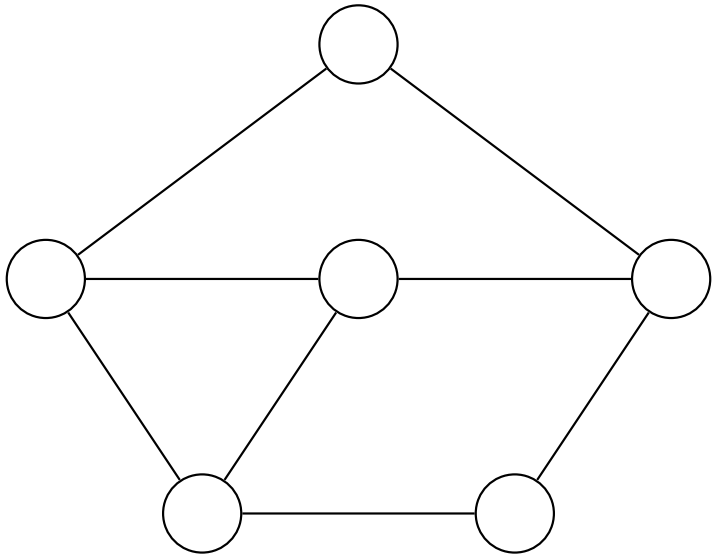
Given a graph $G = (V, E)$, a set of vertices S is:

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:

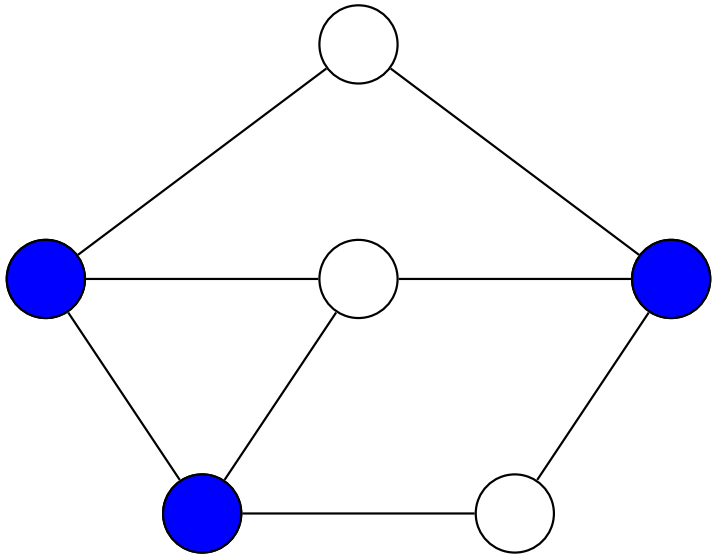
1. A **vertex cover** if every $e \in E$ has at least one endpoint in S.



Vertex Cover

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

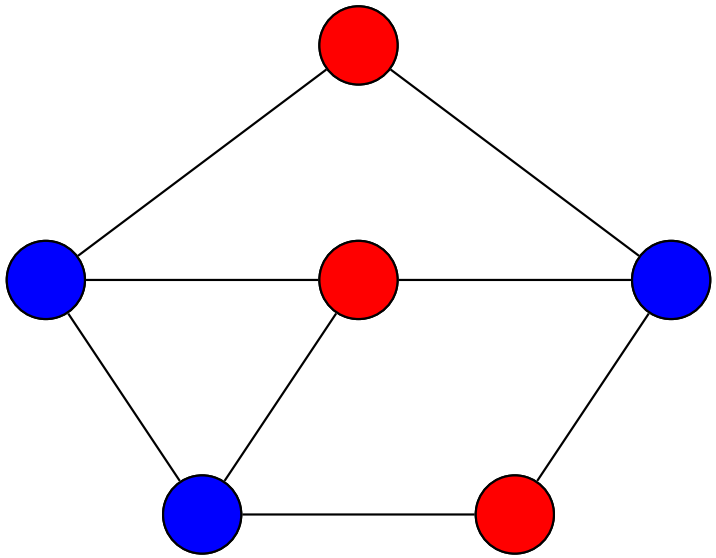
1. A **vertex cover** if every $e \in E$ has at least one endpoint in S.



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.



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.

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.

Exercise. □

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

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. **Claim.** 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. **Claim.** 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**.

Proving Reductions

To prove that $X \leq_P Y$ you need to give an algorithm $\mathcal{A}$ that:

1. Transforms an instance I_X of X into an instance I_Y of Y.

Proving Reductions

To prove that $X \leq_P Y$ you need to give an algorithm $\mathcal{A}$ 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.

Proving Reductions

To prove that $X \leq_P Y$ you need to give an algorithm $\mathcal{A}$ 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.
    1. typical easy direction to prove: answer to I_Y is YES if answer to I_X is YES

Proving Reductions

To prove that $X \leq_P Y$ you need to give an algorithm $\mathcal{A}$ 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.
    1. typical easy direction to prove: answer to I_Y is YES if answer to I_X is YES
    2. typical difficult direction to prove: answer to I_X is YES if answer to I_Y is YES (equivalently answer to I_Y is NO if answer to I_X is NO).

Proving Reductions

To prove that $X \leq_P Y$ you need to give an algorithm $\mathcal{A}$ 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.
   1. typical easy direction to prove: answer to I_Y is YES if answer to I_X is YES
   2. typical difficult direction to prove: answer to I_X is YES if answer to I_Y is YES (equivalently answer to I_Y is NO if answer to I_X is NO).
3. Runs in **polynomial** time.

Example of incorrect reduction proof

Try proving **Matching $\leq_P$ Bipartite Matching** via following reduction:

1. Given graph $G = (V, E)$ obtain a bipartite graph $G' = (V', E')$ as follows.
   1. Let $V_1 = \{u_1 \mid u \in V\}$ and $V_2 = \{u_2 \mid u \in V\}$. We set $V' = V_1 \cup V_2$ (that is, we make two copies of V)

Example of incorrect reduction proof

Try proving **Matching $\leq_P$ Bipartite Matching** via following reduction:

1. Given graph $G = (V, E)$ obtain a bipartite graph $G' = (V', E')$ as follows.
   1. Let $V_1 = \{u_1 \mid u \in V\}$ and $V_2 = \{u_2 \mid u \in V\}$. We set $V' = V_1 \cup V_2$ (that is, we make two copies of V)
   2. $E' = \left\{ u_1 v_2 \;\middle|\; u \neq v \text{ and } uv \in E \right\}$

Example of incorrect reduction proof

Try proving **Matching $\leq_P$ Bipartite Matching** via following reduction:

1. Given graph $G = (V, E)$ obtain a bipartite graph $G' = (V', E')$ as follows.
   1. Let $V_1 = \{u_1 \mid u \in V\}$ and $V_2 = \{u_2 \mid u \in V\}$. We set $V' = V_1 \cup V_2$ (that is, we make two copies of V)
   2. $E' = \left\{ u_1 v_2 \;\middle|\; u \neq v \text{ and } uv \in E \right\}$
2. Given G and integer k the reduction outputs G' and k.

"Proof"

Claim

Reduction is a poly-time algorithm. If G has a matching of size k then G' has a matching of size k.

Proof.

Exercise. □

Claim

If G' has a matching of size k then G has a matching of size k.

"Proof"

Claim

Reduction is a poly-time algorithm. If G has a matching of size k then G' has a matching of size k.

Proof.

Exercise. □

Claim

If G' has a matching of size k then G has a matching of size k.

Incorrect! Why?

"Proof"

Claim

Reduction is a poly-time algorithm. If G has a matching of size k then G' has a matching of size k.

Proof.

Exercise. □

Claim

If G' has a matching of size k then G has a matching of size k.

Incorrect! Why? Vertex $u \in V$ has two copies u_1 and u_2 in G'. A matching in G' may use both copies!

Part IV

More General Reductions

Turing Reduction

A More General Reduction

Definition (Turing reduction.)

Problem X polynomial time reduces to Y if there is an algorithm $\mathcal{A}$ for X that has the following properties:

1. on an instance I_X of X, $\mathcal{A}$ uses polynomial in $|I_X|$ "steps"
2. a step is either a standard computation step, or
3. a sub-routine call to an algorithm that solves Y.

This is a **Turing reduction**.

Turing Reduction

A More General Reduction

Definition (Turing reduction.)

Problem X polynomial time reduces to Y if there is an algorithm $\mathcal{A}$ for X that has the following properties:

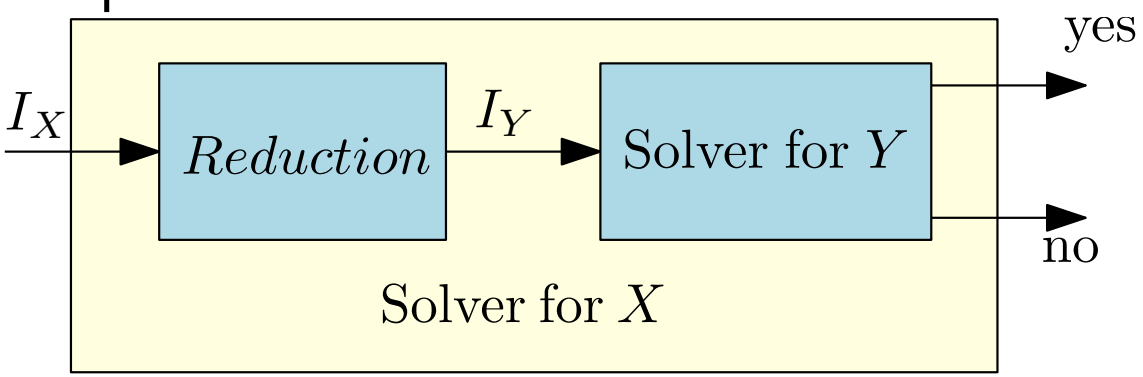
1. on an instance I_X of X, $\mathcal{A}$ uses polynomial in $|I_X|$ "steps"
2. a step is either a standard computation step, or
3. a sub-routine call to an algorithm that solves Y.

This is a **Turing reduction**.

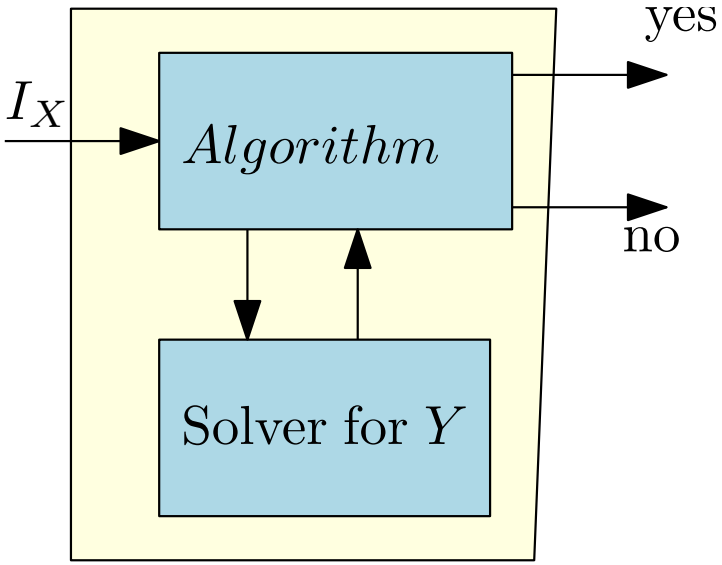
Note: In making sub-routine call to algorithm to solve Y, $\mathcal{A}$ can only ask questions of size polynomial in $|I_X|$. Why?

Comparing reductions

1. Karp reduction:



2. Turing reduction:



Turing reduction

1. Algorithm to solve X can call solver for Y $poly(|I_x|)$ many times.
2. Input to every call is of size $poly(|I_x|)$.

Example of Turing Reduction

Problem (Independent set in circular arcs graph.)

Input: *Collection of arcs on a circle.*
Goal: *Compute the maximum number of non-overlapping arcs.*

Reduced to the following problem:?

Problem (Independent set of intervals.)

Input: *Collection of intervals on the line.*
Goal: *Compute the maximum number of non-overlapping intervals.*

How?

Example of Turing Reduction

Problem (Independent set in circular arcs graph.)

Input: *Collection of arcs on a circle.*
Goal: *Compute the maximum number of non-overlapping arcs.*

Reduced to the following problem:?

Problem (Independent set of intervals.)

Input: *Collection of intervals on the line.*
Goal: *Compute the maximum number of non-overlapping intervals.*

How? Uses algorithm for interval problem multiple times.

Turing vs Karp Reductions

1. Turing reductions more general than Karp reductions.
2. Turing reduction useful in obtaining algorithms via reductions.
3. Karp reduction is simpler and easier to use to prove hardness of problems.
4. Perhaps surprisingly, Karp reductions, although limited, suffice for most known **NP-Completeness** proofs.
5. Karp reductions allow us to distinguish between **NP** and **co-NP** (more on this later).

Part V

The Satisfiability Problem (SAT)

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 $\neg x_i$.

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, $x_1 \vee x_2 \vee \neg x_4$ is a clause.

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, $x_1 \vee x_2 \vee \neg x_4$ is a clause.
3. A **formula in conjunctive normal form** (CNF) is propositional formula which is a conjunction of clauses
   1. $(x_1 \vee x_2 \vee \neg x_4) \wedge (x_2 \vee \neg x_3) \wedge x_5$ 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, $x_1 \vee x_2 \vee \neg x_4$ is a clause.
3. A **formula in conjunctive normal form** (CNF) is propositional formula which is a conjunction of clauses
   1. $(x_1 \vee x_2 \vee \neg x_4) \wedge (x_2 \vee \neg x_3) \wedge x_5$ is a CNF formula.
4. A formula φ is a 3CNF:
   A CNF formula such that every clause has **exactly** 3 literals.
   1. $(x_1 \vee x_2 \vee \neg x_4) \wedge (x_2 \vee \neg x_3 \vee x_1)$ is a 3CNF formula, but $(x_1 \vee x_2 \vee \neg x_4) \wedge (x_2 \vee \neg x_3) \wedge x_5$ is not.

Satisfiability

Problem: SAT

Instance: A CNF formula φ.
Question: Is there a truth assignment to the variable of φ such that φ evaluates to true?

Problem: 3SAT

Instance: A $3\mathrm{CNF}$ formula φ.
Question: Is there a truth assignment to the variable of φ such that φ evaluates to true?

Satisfiability

SAT

Given a CNF formula φ, is there a truth assignment to variables such that φ evaluates to true?

Example

1. $(x_1 \vee x_2 \vee \neg x_4) \wedge (x_2 \vee \neg x_3) \wedge x_5$ is satisfiable; take $x_1, x_2, \ldots x_5$ to be all true
2. $(x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2) \wedge (x_1 \vee x_2)$ is not satisfiable.

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**.

3SAT $\leq_P$ SAT

1. **3SAT** $\leq_P$ **SAT**.
2. Because...
   A **3SAT** instance is also an instance of **SAT**.

SAT $\leq_P$ 3SAT

Claim

SAT $\leq_P$ 3SAT.

SAT $\leq_P$ 3SAT

Claim

SAT $\leq_P$ 3SAT.

Given φ a **SAT** formula we create a **3SAT** formula φ' such that

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

SAT $\leq_P$ 3SAT

How SAT is different from 3SAT?

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

$$\Big(x \vee y \vee z \vee w\Big) \wedge \Big(\neg x \vee \neg y \vee \neg z \vee w \vee u\Big) \wedge \Big(\neg x\Big)$$

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:

$$\Big(x \vee y \vee z \vee w\Big) \wedge \Big(\neg x \vee \neg y \vee \neg z \vee w \vee u\Big) \wedge \Big(\neg x\Big)$$

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

Consider $(x \vee y \vee z \vee w)$

Replace it with $\Big(x \vee y \vee \alpha\Big) \wedge \Big(\neg\alpha \vee w \vee u\Big)$

SAT $\leq_P$ 3SAT

How SAT is different from 3SAT?

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

$$\Big(x \vee y \vee z \vee w\Big) \wedge \Big(\neg x \vee \neg y \vee \neg z \vee w \vee u\Big) \wedge \Big(\neg x\Big)$$

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

Consider $(x \vee y \vee z \vee w)$

Replace it with $\Big(x \vee y \vee \alpha\Big) \wedge \Big(\neg\alpha \vee w \vee u\Big)$

1. Pad short clauses so they have 3 literals.
2. Break long clauses into shorter clauses. (Need to add new variables)
3. Repeat the above till we have a 3CNF.

What about **2SAT**?

What about **2SAT**?

2SAT can be solved in polynomial time! (specifically, linear time!)

What about **2SAT**?

2SAT can be solved in polynomial time! (specifically, linear time!)

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

What about 2SAT?

2SAT can be solved in polynomial time! (specifically, linear time!)

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

Why the reduction from 3SAT to 2SAT fails?

Consider a clause $(x \vee y \vee z)$. We need to reduce it to a collection of **2**CNF clauses. Introduce a face variable α, and rewrite this as

$$(x \vee y \vee \alpha) \wedge (\neg\alpha \vee z) \quad \text{(bad! clause with 3 vars)}$$

$$\text{or} \quad (x \vee \alpha) \wedge (\neg\alpha \vee y \vee z) \quad \text{(bad! clause with 3 vars).}$$

(In animal farm language: **2SAT** good, **3SAT** bad.)

What about **2SAT**?

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

Look in books etc.