ILP and Reductions

Lecture 21

April 10, 2018

Most slides are courtesy Prof. Chekuri

Part I

Integer Linear Programming (ILP)

Integer Linear Programming

Problem

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

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

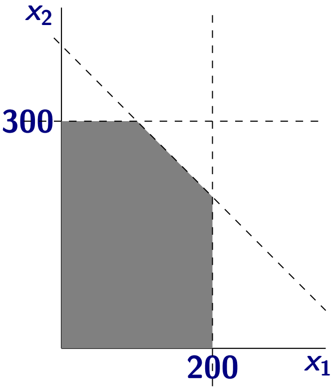
Input is matrix $A = (a_{ij}) \in \mathbb{R}^{n \times d}$, column vector $b = (b_i) \in \mathbb{R}^n$, and row vector $c = (c_j) \in \mathbb{R}^d$

Factory Example

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

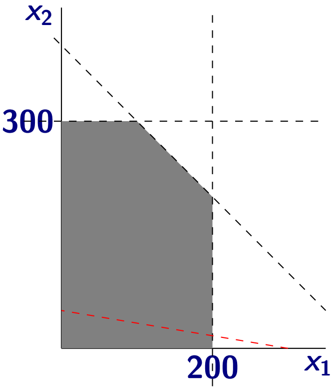
Suppose we want x_1, x_2 to be integer valued.

Factory Example Figure



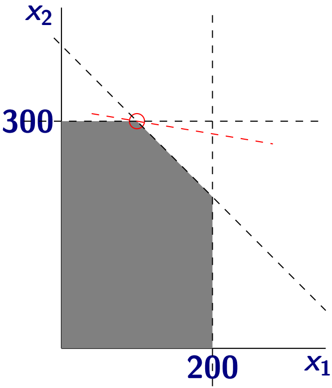
1. Feasible values of x_1 and x_2 are **integer points in shaded region**
2. Optimization function is a line; moving the line until it just leaves the final integer point in feasible region, gives optimal values

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

Example: s-t minimum cut and implicit constraints

Input Graph $G = (V, E)$, edge capacities $c(e), e \in E$. $s, t \in V$

Goal Find minimum capacity s-t cut 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 is same as solving integer program. We know how to solve linear programs 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 is same as solving integer program. We know how to solve linear programs 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. Duality is a critical tool in the theory of linear programming. Duality implies the Linear Programming is in **co-NP**. Do you see why?

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. Duality is a critical tool in the theory of linear programming. Duality implies the Linear Programming is in **co-NP**. Do you see why?
5. 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.

Examples

Example

An instance of **Bipartite Matching** is a bipartite graph, and an integer k. The solution to this instance is “YES” if the graph has a matching of size $\geq k$, and “NO” otherwise.

Examples

Example

An instance of **Bipartite Matching** is a bipartite graph, and an integer k. The solution to this instance is "YES" if the graph has a matching of size $\geq k$, and "NO" otherwise.

Example

An instance of **Max-Flow** is a graph G with edge-capacities, two vertices s, t, and an integer k. The solution to this instance is "YES" if there is a flow from s to t of value $\geq k$, else 'NO".

What is an algorithm for a decision Problem X?

It takes as input an instance of X, and outputs either "YES" or "NO".

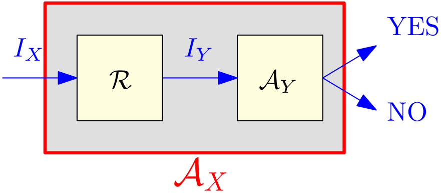
Using reductions to solve problems

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

Using reductions to solve problems

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

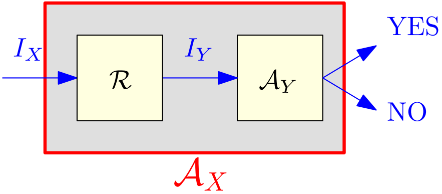
$\mathcal{A}_X(I_X)$:
    // I_X: instance of X.
    $I_Y \leftarrow \mathcal{R}(I_X)$
    return $\mathcal{A}_Y(I_Y)$



Using reductions to solve problems

1. $\mathcal{R}$: Reduction $X \to Y$
2. $\mathcal{A}_Y$: algorithm for Y
3. $\implies$ 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 $\implies$ $\mathcal{A}_X$ polynomial-time.

Comparing hardness of problems

1. If Problem X reduces to Problem Y, **written as $X \leq Y$,** then X cannot be harder to solve than Y.
2. **Bipartite Matching** $\leq$ **Max-Flow**.
   Bipartite Matching cannot be harder than **Max-Flow**.

Comparing hardness of problems

1. If Problem X reduces to Problem Y, **written as $X \leq Y$,** then X cannot be harder to solve than Y.
2. **Bipartite Matching** $\leq$ **Max-Flow**.
   Bipartite Matching cannot be harder than **Max-Flow**.
3. Equivalently,
   Max-Flow is at least as hard as **Bipartite Matching**.

Comparing hardness of problems

1. If Problem X reduces to Problem Y, **written as $X \leq Y$,** then X cannot be harder to solve than Y.
2. **Bipartite Matching** $\leq$ **Max-Flow**.
   Bipartite Matching cannot be harder than **Max-Flow**.
3. Equivalently,
   Max-Flow is at least as hard as **Bipartite Matching**.
4. $X \leq Y$:
   1. X is no harder than Y, or
   2. Y is at least as hard as X.

Polynomial-time Reductions

Efficient Algorithm: runs in polynomial-time.

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

$X \leq_P Y$ **:** poly-time reduction from problem X to problem Y.

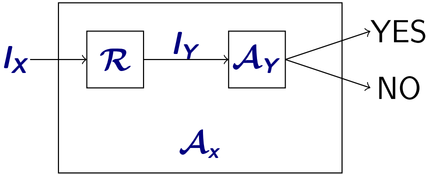
Polynomial-time Reductions

Efficient Algorithm: runs in polynomial-time.

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

$X \leq_P Y$ **:** poly-time reduction from problem X to problem Y.

Then, polynomial-time algorithm $\mathcal{A}_Y$ for Y, gives an efficient algorithm for X.



Polynomial-time Reduction

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

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

Polynomial-time Reduction

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

1. given an instance I_X of X, $\mathcal{A}$ produces an instance I_Y of Y
2. $\mathcal{A}$ 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.

Polynomial-time reductions and instance sizes

Proposition

Let $\mathcal{A}$ be a polynomial-time algorithm reducing 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 $\mathcal{A}$ is polynomial in the size of I_X.

Proof.

$\mathcal{A}$ 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 $\mathcal{A}$ be a polynomial-time algorithm reducing 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 $\mathcal{A}$ is polynomial in the size of I_X.

Proof.

$\mathcal{A}$ 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 $\mathcal{A}$ on input I_X.

Polynomial-time reductions and instance sizes

Proposition

Let $\mathcal{A}$ be a polynomial-time algorithm reducing 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 $\mathcal{A}$ is polynomial in the size of I_X.

Proof.

$\mathcal{A}$ 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 $\mathcal{A}$ on input I_X.

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

Polynomial-time reductions and instance sizes

Proposition

Let $\mathcal{A}$ be a polynomial-time algorithm reducing 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 $\mathcal{A}$ is polynomial in the size of I_X.

Proof.

$\mathcal{A}$ 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 $\mathcal{A}$ on input I_X.
$\mathcal{A}$ 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 Reduction

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

1. Given an instance I_X of X, $\mathcal{A}$ produces an instance I_Y of Y.
2. $\mathcal{A}$ 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

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.

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.

Suppose **Independent Set** $\leq_P$ **Clique**.

If you believe that **Independent Set** does not have an efficient algorithm, then can **Clique** have 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.

Suppose **Independent Set** $\leq_P$ **Clique**.

If you believe that **Independent Set** does not have an efficient algorithm, then can **Clique** have an efficient algorithm?

If **Clique** had an efficient algorithm, so would **Independent Set**!

So, NO!

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

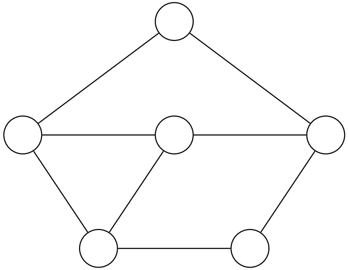
Given a graph G, a set of vertices V' is:

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:

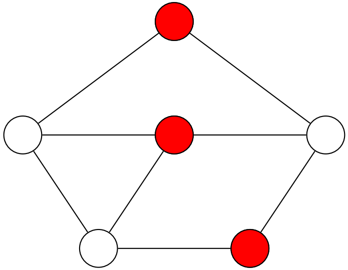
1. **independent set**: no two vertices of V' connected by an edge.
2. **clique**: *every* pair of vertices in V' is connected by an edge of G.



Independent Sets and Cliques

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

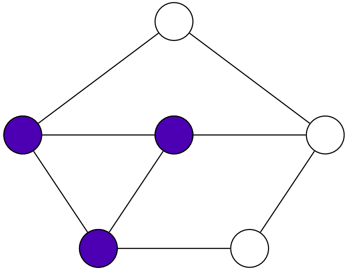
1. **independent set**: no two vertices of V' connected by an edge.
2. **clique**: *every* pair of vertices in V' is connected by an edge of G.



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.



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

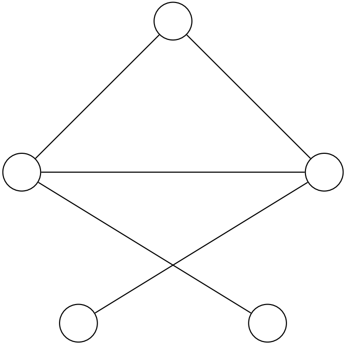
Instance: A graph G and an integer k.
Question: Does G has a clique of size $\geq k$?

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

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

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

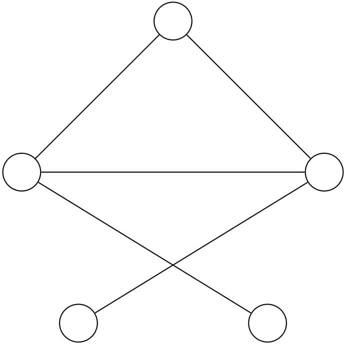
Instance of **Independent Set**: graph G and an integer k.



Reducing Independent Set to Clique

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

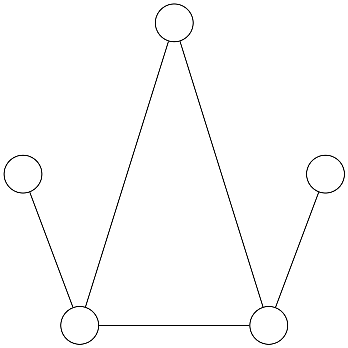
Convert G to $\overline{G}$, in which (u, v) is an edge iff (u, v) is not an edge of G. ($\overline{G}$ is the *complement* of G.)
Instance of **Clique**: graph $\overline{G}$ and integer k.



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

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

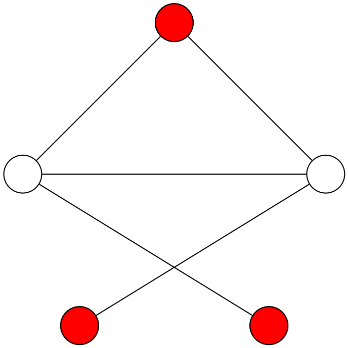
Convert G to $\overline{G}$, in which (u, v) is an edge iff (u, v) is not an edge of G. ($\overline{G}$ is the *complement* of G.)
Instance of **Clique**: graph $\overline{G}$ and integer k.



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

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

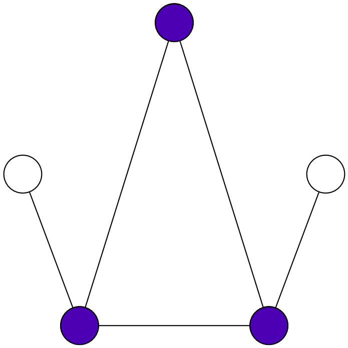
Convert G to $\overline{G}$, in which (u, v) is an edge iff (u, v) is not an edge of G. ($\overline{G}$ is the *complement* of G.)
Instance of **Clique**: graph $\overline{G}$ and integer k.



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

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

Convert G to $\overline{G}$, in which (u, v) is an edge iff (u, v) is not an edge of G. ($\overline{G}$ is the *complement* 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

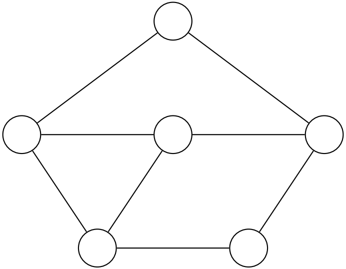
Given a graph $G = (V, E)$, a set of vertices S is:

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:

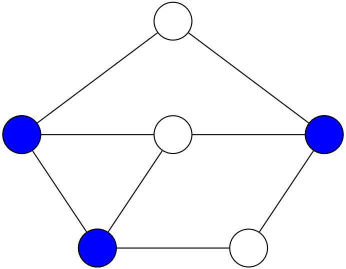
1. A **vertex cover** if every $e \in E$ has at least one endpoint in S.



Vertex Cover

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

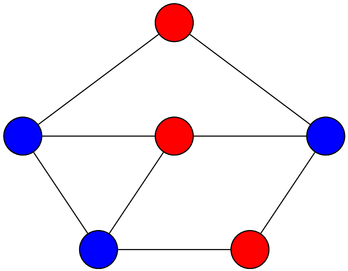
1. A **vertex cover** if every $e \in E$ has at least one endpoint in S.



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.

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

1. Consider any edge $(u, v) \in E$. Is u or v in $V \setminus S$?

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 $(u, v) \in E$. Is u or v in $V \setminus S$?
2. Since S is an independent set, either $u \not\in S$ or $v \not\in S$.

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 $(u, v) \in E$. Is u or v in $V \setminus S$?
2. Since S is an independent set, either $u \notin S$ or $v \notin S$.
3. Thus, either $u \in V \setminus S$ or $v \in V \setminus S$.
4. $V \setminus S$ 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.

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

1. Consider any edge $(u, v) \in E$. Is u or v in $V \setminus S$?
2. Since S is an independent set, either $u \notin S$ or $v \notin S$.
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 $u, v \in S$. Is $(u, v) \in E$?

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 $(u, v) \in E$. Is u or v in $V \setminus S$?
2. Since S is an independent set, either $u \notin S$ or $v \notin S$.
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 $u, v \in S$. Is $(u, v) \in E$?
2. $(u, v) \notin E$, as otherwise $V \setminus S$ does not cover (u, v).
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.
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**.

The **Set Cover** Problem

Problem (**Set Cover**)

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

Goal: *Is there a collection of at most* k *of these sets* S_i *whose union is equal to* U*?*

The **Set Cover** Problem

Problem (**Set Cover**)

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

Goal: *Is there a collection of at most k of these sets S_i whose union is equal to U?*

Example

Let $U = \{1, 2, 3, 4, 5, 6, 7\}$, $k = 2$ with

$$\begin{array}{ll} S_1 = \{3, 7\} & S_2 = \{3, 4, 5\} \\ S_3 = \{1\} & S_4 = \{2, 4\} \\ S_5 = \{5\} & S_6 = \{1, 2, 6, 7\} \end{array}$$

The Set Cover Problem

Problem (Set Cover)

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

Goal: *Is there a collection of at most* k *of these sets* S_i *whose union is equal to* U*?*

Example

Let $U = \{1, 2, 3, 4, 5, 6, 7\}$, $k = 2$ with

$$\begin{array}{ll} S_1 = \{3,7\} & S_2 = \{3,4,5\} \\ S_3 = \{1\} & S_4 = \{2,4\} \\ S_5 = \{5\} & S_6 = \{1,2,6,7\} \end{array}$$

$\{S_2, S_6\}$ is a set cover

Vertex Cover $\leq_P$ Set Cover

Given graph $G = (V, E)$ and integer k as instance of **Vertex Cover**, construct an instance of **Set Cover** as follows:

Vertex Cover $\leq_P$ Set Cover

Given graph $G = (V, E)$ and integer k as instance of **Vertex Cover**, construct an instance of **Set Cover** as follows:

1. Number k for the **Set Cover** instance is the same as the number k given for the **Vertex Cover** instance.

Vertex Cover $\leq_P$ Set Cover

Given graph $G = (V, E)$ and integer k as instance of **Vertex Cover**, construct an instance of **Set Cover** as follows:

1. Number k for the **Set Cover** instance is the same as the number k given for the **Vertex Cover** instance.
2. $U = E$.

Vertex Cover $\leq_P$ Set Cover

Given graph $G = (V, E)$ and integer k as instance of **Vertex Cover**, construct an instance of **Set Cover** as follows:

1. Number k for the **Set Cover** instance is the same as the number k given for the **Vertex Cover** instance.
2. $U = E$.
3. We will have one set corresponding to each vertex; $S_v = \{e \mid e \text{ is incident on } v\}$.

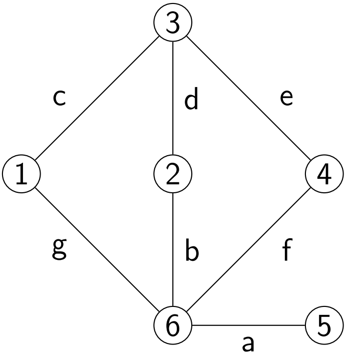
Vertex Cover $\leq_P$ Set Cover

Given graph $G = (V, E)$ and integer k as instance of **Vertex Cover**, construct an instance of **Set Cover** as follows:

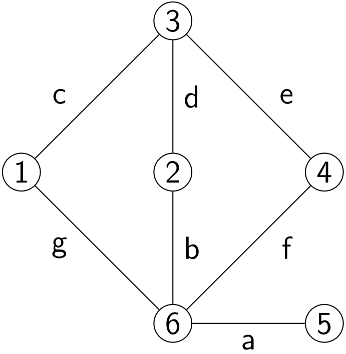
1. Number k for the **Set Cover** instance is the same as the number k given for the **Vertex Cover** instance.
2. $U = E$.
3. We will have one set corresponding to each vertex; $S_v = \{e \mid e \text{ is incident on } v\}$.

Observe that G has vertex cover of size k if and only if $U, \{S_v\}_{v \in V}$ has a set cover of size k. (Exercise: Prove this.)

Vertex Cover $\leq_P$ Set Cover: Example



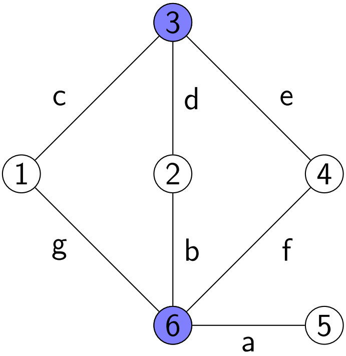
Vertex Cover $\leq_P$ Set Cover: Example



Let $U = \{a, b, c, d, e, f, g\}$, $k = 2$ with

$$S_1 = \{c, g\} \quad S_2 = \{b, d\}$$
$$S_3 = \{c, d, e\} \quad S_4 = \{e, f\}$$
$$S_5 = \{a\} \quad S_6 = \{a, b, f, g\}$$

Vertex Cover $\leq_P$ Set Cover: Example



Let $U = \{a, b, c, d, e, f, g\}$, $k = 2$ with

$S_1 = \{c, g\}$ $\quad S_2 = \{b, d\}$
$S_3 = \{c, d, e\}$ $\quad S_4 = \{e, f\}$
$S_5 = \{a\}$ $\quad S_6 = \{a, b, f, g\}$

$\{S_3, S_6\}$ is a set cover

$\{3, 6\}$ is a vertex cover

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!

Subset sum and Partition?

Problem: Subset Sum

Instance: S - set of positive integers, t: - an integer number (target).
Question: Is there a subset $X \subseteq S$ such that $\sum_{x \in X} x = t$?

Problem: Partition

Instance: A set S of n numbers.
Question: Is there a subset $T \subseteq S$ s.t. $\sum_{t \in T} t = \sum_{s \in S \setminus T} s$?

Subset sum and Partition?

Problem: Subset Sum

Instance: S - set of positive integers, t: - an integer number (target).
Question: Is there a subset $X \subseteq S$ such that $\sum_{x \in X} x = t$?

Problem: Partition

Instance: A set S of n numbers.
Question: Is there a subset $T \subseteq S$ s.t. $\sum_{t \in T} t = \sum_{s \in S \setminus T} s$?

Assume that we can solve **Subset Sum** in polynomial time, then we can solve **Partition** in polynomial time. This statement is

(A) True.

(B) Mostly true.

(C) False.

(D) Mostly false.

II: Partition and subset sum?

Problem: Partition

Instance: A set S of n numbers.
Question: Is there a subset $T \subseteq S$ s.t. $\sum_{t \in T} t = \sum_{s \in S \setminus T} s$?

Problem: Subset Sum

Instance: S - set of positive integers, t: - an integer number (target).
Question: Is there a subset $X \subseteq S$ such that $\sum_{x \in X} x = t$?

Assume that we can solve **Partition** in polynomial time, then we can solve **Subset Sum** in polynomial time. This statement is

- **(A)** True.
- **(B)** Mostly true.
- **(C)** False.
- **(D)** Mostly false.