Polynomial Time Reductions

Lecture 20

April 9, 2013

Part I

Introduction to 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.

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.)

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

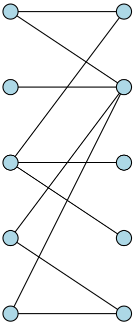
1. We use reductions to find algorithms to solve problems.
2. We also use reductions to show that we can't find algorithms for some problems. (We say that these problems are hard.)

Also, the right reductions might win you a million dollars!

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 $\geq k$?



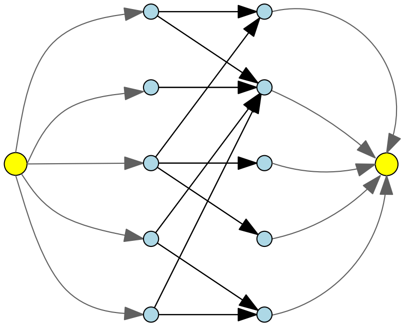
Solution

Reduce it to **Max-Flow**. G has a matching of size $\geq k$ iff there is a flow from s to t of value $\geq k$.

Example 1: Bipartite Matching and Flows

How do we solve the Bipartite Matching Problem?

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



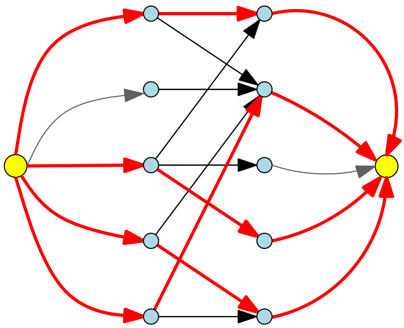
Solution

Reduce it to **Max-Flow**. G has a matching of size $\geq \mathbf{k}$ iff there is a flow from $\mathbf{s}$ to $\mathbf{t}$ of value $\geq \mathbf{k}$.

Example 1: Bipartite Matching and Flows

How do we solve the Bipartite Matching Problem?

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



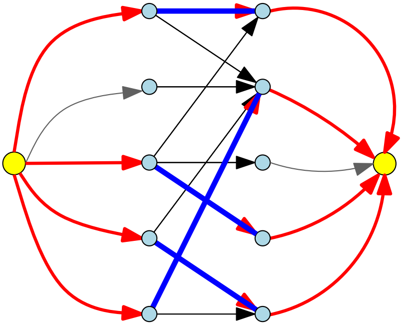
Solution

Reduce it to **Max-Flow**. G has a matching of size $\geq \mathbf{k}$ iff there is a flow from $\mathbf{s}$ to $\mathbf{t}$ of value $\geq \mathbf{k}$.

Example 1: Bipartite Matching and Flows

How do we solve the Bipartite Matching Problem?

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



Solution

Reduce it to **Max-Flow**. G has a matching of size $\geq \mathbf{k}$ iff there is a flow from $\mathbf{s}$ to $\mathbf{t}$ of value $\geq \mathbf{k}$.

Types of Problems

Decision, Search, and Optimization

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

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**.

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.

Optimization and Decision problems

For max flow...

Problem (Max-Flow optimization version)

Given an instance G of network flow, find the maximum flow between $\mathbf{s}$ *and* $\mathbf{t}$.

Problem (Max-Flow decision version)

Given an instance G of network flow and a parameter $\mathbf{K}$, *is there a flow in G, from* $\mathbf{s}$ *to* $\mathbf{t}$, *of value at least* $\mathbf{K}$?

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.

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.
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 $\mathbf{k}$. The solution to this instance is "YES" if the graph has a matching of size $\geq \mathbf{k}$, and "NO" otherwise.

Example

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

What is an algorithm for a decision Problem $\mathbf{X}$?

It takes as input an instance of $\mathbf{X}$, and outputs either "YES" or "NO".

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".

Examples

Example

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

Example

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

What is an algorithm for a decision Problem $\mathbf{X}$?

It takes as input an instance of $\mathbf{X}$, and outputs either "YES" or "NO".

Examples

Example

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

Example

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

What is an algorithm for a decision Problem $\mathbf{X}$?

It takes as input an instance of $\mathbf{X}$, and outputs either "YES" or "NO".

Encoding an instance into a string

1. **I**; Instance of some problem.
2. **I** can be fully and precisely described (say in a text file).
3. Resulting text file is a binary string.
4. $\implies$ Any input can be interpreted as a binary string **S**.
5. ... Running time of algorithm: Function of length of **S** (i.e., **n**).

Decision Problems and Languages

1. A finite alphabet Σ. Σ^* is set of all finite strings on Σ.
2. A language L is simply a subset of Σ^*; a set of strings.

For every language L there is an associated decision problem Π_L and conversely, for every decision problem Π there is an associated language L_Π.

1. Given L, Π_L is the following decision problem: Given $x \in \Sigma^*$, is $x \in L$? Each string in Σ^* is an instance of Π_L and L is the set of instances for which the answer is YES.
2. Given Π the associated language is

$$L_\Pi = \left\{ I \;\middle|\; I \text{ is an instance of } \Pi \text{ for which answer is YES} \right\}.$$

Thus, decision problems and languages are used interchangeably.

Decision Problems and Languages

1. A finite alphabet Σ. Σ^* is set of all finite strings on Σ.
2. A language L is simply a subset of Σ^*; a set of strings.

For every language L there is an associated decision problem Π_L and conversely, for every decision problem Π there is an associated language L_Π.

1. Given L, Π_L is the following decision problem: Given $x \in \Sigma^*$, is $x \in L$? Each string in Σ^* is an instance of Π_L and L is the set of instances for which the answer is YES.
2. Given Π the associated language is

$$L_\Pi = \left\{ I \;\middle|\; I \text{ is an instance of } \Pi \text{ for which answer is YES} \right\}.$$

Thus, decision problems and languages are used interchangeably.

Decision Problems and Languages

1. A finite alphabet Σ. Σ^* is set of all finite strings on Σ.
2. A language L is simply a subset of Σ^*; a set of strings.

For every language L there is an associated decision problem Π_L and conversely, for every decision problem Π there is an associated language L_Π.

1. Given L, Π_L is the following decision problem: Given $x \in \Sigma^*$, is $x \in L$? Each string in Σ^* is an instance of Π_L and L is the set of instances for which the answer is YES.
2. Given Π the associated language is

$$L_\Pi = \left\{ I \;\middle|\; I \text{ is an instance of } \Pi \text{ for which answer is YES} \right\}.$$

Thus, decision problems and languages are used interchangeably.

Example

1. The decision problem **Primality**, and the language

$$\mathbf{L} = \left\{ \#\mathbf{p} \;\middle|\; \mathbf{p} \text{ is a prime number} \right\}.$$

   Here $\#\mathbf{p}$ is the string in base $\mathbf{10}$ representing $\mathbf{p}$.

2. **Bipartite** (is given graph is bipartite. The language is

$$\mathbf{L} = \left\{ \boldsymbol{\mathcal{S}}(\mathsf{G}) \;\middle|\; \mathsf{G} \text{ is a bipartite graph} \right\}.$$

   Here $\boldsymbol{\mathcal{S}}(\mathsf{G})$ is the string encoding the graph G.

Reductions, revised.

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

1. An algorithm ...
2. Input: $\mathbf{I_X}$, an instance of **X**.
3. Output: $\mathbf{I_Y}$ an instance of **Y**.
4. Such that:

$$\boxed{\mathbf{I_Y} \text{ is YES instance of } \mathbf{Y}} \iff \boxed{\mathbf{I_X} \text{ is YES instance of } \mathbf{X}}$$

There are other kinds of reductions.

Reductions, revised.

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

1. An algorithm ...
2. Input: $\mathbf{I_X}$, an instance of **X**.
3. Output: $\mathbf{I_Y}$ an instance of **Y**.
4. Such that:

$$\boxed{\mathbf{I_Y} \text{ is YES instance of } \mathbf{Y}} \iff \boxed{\mathbf{I_X} \text{ is YES instance of } \mathbf{X}}$$

There are other kinds of reductions.

Using reductions to solve problems

1. $\mathcal{R}$: Reduction $\mathbf{X} \rightarrow \mathbf{Y}$
2. $\mathcal{A}_{\mathbf{Y}}$: algorithm for $\mathbf{Y}$:
3. $\implies$ New algorithm for $\mathbf{X}$:

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

If $\mathcal{R}$ and $\mathcal{A}_{\mathbf{Y}}$ polynomial-time $\implies$ $\mathcal{A}_{\mathbf{X}}$ polynomial-time.

Using reductions to solve problems

1. $\mathcal{R}$: Reduction $\mathbf{X} \rightarrow \mathbf{Y}$
2. $\mathcal{A}_{\mathbf{Y}}$: algorithm for $\mathbf{Y}$:
3. $\implies$ New algorithm for $\mathbf{X}$:

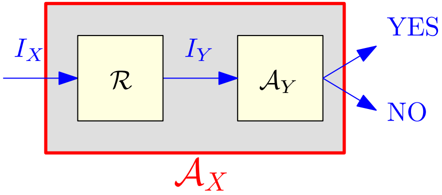
```
A_X(I_X):
        // I_X:  instance of X.
        I_Y <= R(I_X)
        return A_Y(I_Y)
```

If $\mathcal{R}$ and $\mathcal{A}_{\mathbf{Y}}$ polynomial-time $\implies$ $\mathcal{A}_{\mathbf{X}}$ polynomial-time.

Using reductions to solve problems

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

$\mathcal{A}_{\mathbf{X}}(\mathbf{I_X})$:
   // $\mathbf{I_X}$: instance of $\mathbf{X}$.
   $\mathbf{I_Y} \Leftarrow \mathcal{R}(\mathbf{I_X})$
   return $\mathcal{A}_{\mathbf{Y}}(\mathbf{I_Y})$



If $\mathcal{R}$ and $\mathcal{A}_{\mathbf{Y}}$ polynomial-time $\implies$ $\mathcal{A}_{\mathbf{X}}$ 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. **Bipartite Matching** $\leq$ **Max-Flow**.
   Bipartite Matching cannot be harder than **Max-Flow**.
4. Equivalently,
   Max-Flow is at least as hard as **Bipartite Matching**.
5. **X** $\leq$ **Y**:
   1. **X** is no harder than **Y**, or
   2. **Y** is at least as hard as **X**.

Part II

Examples of Reductions

Independent Sets and Cliques

Given a graph $\mathbf{G}$, a set of vertices $\mathbf{V}'$ is:

1. **independent set**: no two vertices of $\mathbf{V}'$ connected by an edge.

Independent Sets and Cliques

Given a graph $\mathbf{G}$, a set of vertices $\mathbf{V'}$ is:

1. **independent set**: no two vertices of $\mathbf{V'}$ connected by an edge.

Independent Sets and Cliques

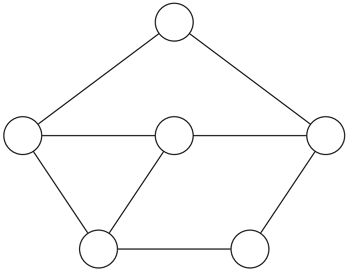
Given a graph $\mathbf{G}$, a set of vertices $\mathbf{V'}$ is:

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

Independent Sets and Cliques

Given a graph **G**, a set of vertices **V′** is:

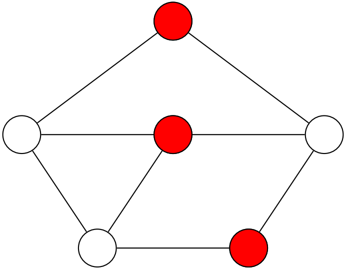
1. **independent set**: no two vertices of **V′** connected by an edge.
2. **clique**: *every* pair of vertices in **V′** is connected by an edge of **G**.



Independent Sets and Cliques

Given a graph **G**, a set of vertices **V′** is:

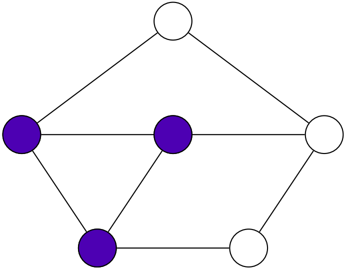
1. **independent set**: no two vertices of **V′** connected by an edge.
2. **clique**: *every* pair of vertices in **V′** is connected by an edge of **G**.



Independent Sets and Cliques

Given a graph $\mathbf{G}$, a set of vertices $\mathbf{V'}$ is:

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



The Independent Set and Clique Problems

Problem: Independent Set

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

Problem: Clique

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

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

Problem: Independent Set

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

Problem: Clique

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

Recall

For decision problems $\mathbf{X}, \mathbf{Y}$, a reduction from $\mathbf{X}$ to $\mathbf{Y}$ is:

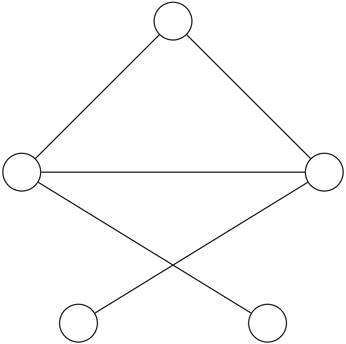
1. An algorithm ...
2. that takes $\mathbf{I_X}$, an instance of $\mathbf{X}$ as input ...
3. and returns $\mathbf{I_Y}$, an instance of $\mathbf{Y}$ as output ...
4. such that the solution (YES/NO) to $\mathbf{I_Y}$ is the same as the solution to $\mathbf{I_X}$.

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

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

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

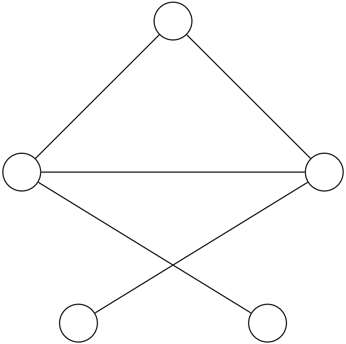
An instance of **Independent Set** is a graph **G** and an integer **k**.



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

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

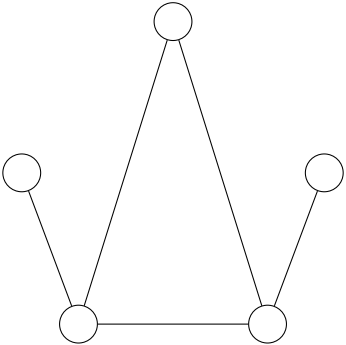
Convert **G** to $\overline{\mathbf{G}}$, in which $(\mathbf{u}, \mathbf{v})$ is an edge iff $(\mathbf{u}, \mathbf{v})$ is not an edge of **G**. ($\overline{\mathbf{G}}$ is the *complement* of **G**.)
We use $\overline{\mathbf{G}}$ and **k** as the instance of **Clique**.



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

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

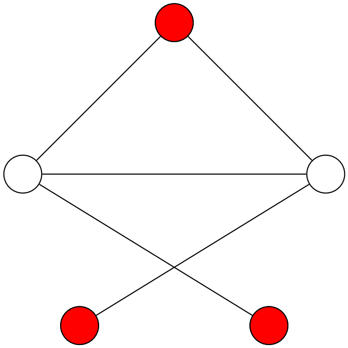
Convert $\mathbf{G}$ to $\overline{\mathbf{G}}$, in which $(\mathbf{u}, \mathbf{v})$ is an edge iff $(\mathbf{u}, \mathbf{v})$ is not an edge of $\mathbf{G}$. ($\overline{\mathbf{G}}$ is the *complement* of $\mathbf{G}$.)
We use $\overline{\mathbf{G}}$ and $\mathbf{k}$ as the instance of **Clique**.



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

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

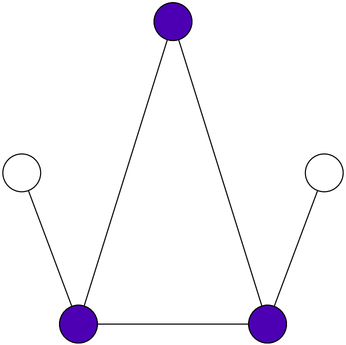
Convert $\mathbf{G}$ to $\overline{\mathbf{G}}$, in which $(\mathbf{u}, \mathbf{v})$ is an edge iff $(\mathbf{u}, \mathbf{v})$ is not an edge of $\mathbf{G}$. ($\overline{\mathbf{G}}$ is the *complement* of $\mathbf{G}$.)
We use $\overline{\mathbf{G}}$ and $\mathbf{k}$ as the instance of **Clique**.



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

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

Convert $\mathbf{G}$ to $\overline{\mathbf{G}}$, in which $(\mathbf{u}, \mathbf{v})$ is an edge iff $(\mathbf{u}, \mathbf{v})$ is not an edge of $\mathbf{G}$. ($\overline{\mathbf{G}}$ is the *complement* of $\mathbf{G}$.)
We use $\overline{\mathbf{G}}$ and $\mathbf{k}$ as the instance of **Clique**.



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**.

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**.

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**.

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**.

DFAs and NFAs

DFAs (Remember 373?) are automata that accept regular languages. NFAs are the same, except that they are non-deterministic, while DFAs are deterministic.

Every NFA can be converted to a DFA that accepts the same language using the subset construction.

(How long does this take?)
The smallest DFA equivalent to an NFA with $\mathbf{n}$ states may have $\approx \mathbf{2^n}$ states.

DFAs and NFAs

DFAs (Remember 373?) are automata that accept regular languages. NFAs are the same, except that they are non-deterministic, while DFAs are deterministic.

Every NFA can be converted to a DFA that accepts the same language using the subset construction.

(How long does this take?)
The smallest DFA equivalent to an NFA with n states may have $\approx 2^n$ states.

DFAs and NFAs

DFAs (Remember 373?) are automata that accept regular languages. NFAs are the same, except that they are non-deterministic, while DFAs are deterministic.

Every NFA can be converted to a DFA that accepts the same language using the subset construction.

(How long does this take?)
The smallest DFA equivalent to an NFA with $\mathbf{n}$ states may have $\approx \mathbf{2^n}$ states.

DFA Universality

A DFA $\mathbf{M}$ is universal if it accepts every string.
That is, $\mathbf{L(M)} = \mathbf{\Sigma}^*$, the set of all strings.

Problem (DFA universality)

Input: *A DFA* $\mathbf{M}$*.*
Goal: *Is* $\mathbf{M}$ *universal?*

How do we solve **DFA Universality**?
We check if $\mathbf{M}$ has *any* reachable non-final state.
Alternatively, minimize $\mathbf{M}$ to obtain $\mathbf{M'}$ and see if $\mathbf{M'}$ has a single state which is an accepting state.

DFA Universality

A DFA $\mathbf{M}$ is universal if it accepts every string.
That is, $\mathbf{L(M)} = \mathbf{\Sigma}^*$, the set of all strings.

Problem (DFA universality)

Input: *A* DFA $\mathbf{M}$.
Goal: *Is* $\mathbf{M}$ *universal?*

How do we solve **DFA Universality**?
We check if $\mathbf{M}$ has *any* reachable non-final state.
Alternatively, minimize $\mathbf{M}$ to obtain $\mathbf{M'}$ and see if $\mathbf{M'}$ has a single state which is an accepting state.

DFA Universality

A DFA **M** is universal if it accepts every string.
That is, $\mathbf{L(M) = \Sigma^*}$, 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.

DFA Universality

A DFA **M** is universal if it accepts every string.
That is, $\mathbf{L(M)} = \mathbf{\Sigma}^*$, 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.

NFA Universality

An NFA **N** is said to be universal if it accepts every string. That is, $\mathbf{L(N) = \Sigma^*}$, 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!

NFA Universality

An NFA **N** is said to be universal if it accepts every string. That is, $\mathbf{L(N) = \Sigma^*}$, 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!

NFA Universality

An NFA **N** is said to be universal if it accepts every string. That is, $\mathbf{L(N) = \Sigma^*}$, 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!

NFA Universality

An NFA **N** is said to be universal if it accepts every string. That is, $\mathbf{L(N) = \Sigma^*}$, 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!

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 $\mathbf{X}$ to problem $\mathbf{Y}$ (we write $\mathbf{X} \leq_P \mathbf{Y}$), and a poly-time algorithm $\mathcal{A}_\mathbf{Y}$ for $\mathbf{Y}$, we have a polynomial-time/efficient algorithm for $\mathbf{X}$.

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 $\mathbf{X}$ to problem $\mathbf{Y}$ (we write $\mathbf{X} \leq_P \mathbf{Y}$), and a poly-time algorithm $\mathcal{A}_Y$ for $\mathbf{Y}$, we have a polynomial-time/efficient algorithm for $\mathbf{X}$.

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.

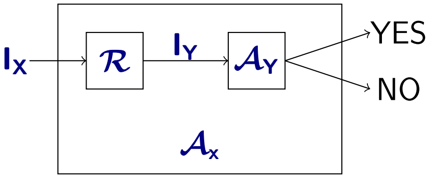
If we have a polynomial-time reduction from problem **X** to problem **Y** (we write $\mathbf{X} \leq_{\mathbf{P}} \mathbf{Y}$), and a poly-time algorithm $\mathcal{A}_{\mathbf{Y}}$ for **Y**, we have a polynomial-time/efficient algorithm for **X**.

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 $\mathbf{X}$ to problem $\mathbf{Y}$ (we write $\mathbf{X} \leq_P \mathbf{Y}$), and a poly-time algorithm $\mathcal{A}_{\mathbf{Y}}$ for $\mathbf{Y}$, we have a polynomial-time/efficient algorithm for $\mathbf{X}$.



Polynomial-time Reduction

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

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

Proposition

If $\mathbf{X} \leq_{\mathbf{P}} \mathbf{Y}$ *then a polynomial time algorithm for* $\mathbf{Y}$ *implies a polynomial time algorithm for* $\mathbf{X}$.

Such a reduction is called a **Karp reduction**. Most reductions we will need are Karp reductions.

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**?

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!

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**?

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!

Polynomial-time reductions and hardness

For decision problems **X** and **Y**, if $\mathbf{X} \leq_{\mathbf{P}} \mathbf{Y}$, 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** $\leq_{\mathbf{P}}$ **Clique**. If **Clique** had an efficient algorithm, so would **Independent Set**!

If $\mathbf{X} \leq_{\mathbf{P}} \mathbf{Y}$ and **X** does not have an efficient algorithm, **Y** cannot 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.

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!

Polynomial-time reductions and instance sizes

Proposition

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

Proof.

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

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

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

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 and instance sizes

Proposition

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

Proof.

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

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 and instance sizes

Proposition

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

Proof.

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

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

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

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

1. Given an instance $\mathbf{I_X}$ of $\mathbf{X}$, $\mathcal{A}$ produces an instance $\mathbf{I_Y}$ of $\mathbf{Y}$.
2. $\mathcal{A}$ runs in time polynomial in $|\mathbf{I_X}|$. This implies that $|\mathbf{I_Y}|$ (size of $\mathbf{I_Y}$) is polynomial in $|\mathbf{I_X}|$.
3. Answer to $\mathbf{I_X}$ YES *iff* answer to $\mathbf{I_Y}$ is YES.

Proposition

If $\mathbf{X} \leq_{\mathbf{P}} \mathbf{Y}$ then a polynomial time algorithm for $\mathbf{Y}$ implies a polynomial time algorithm for $\mathbf{X}$.

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

Transitivity of Reductions

Proposition

$\mathbf{X} \leq_{\mathbf{P}} \mathbf{Y}$ *and* $\mathbf{Y} \leq_{\mathbf{P}} \mathbf{Z}$ *implies that* $\mathbf{X} \leq_{\mathbf{P}} \mathbf{Z}$.

Note: $\mathbf{X} \leq_{\mathbf{P}} \mathbf{Y}$ does not imply that $\mathbf{Y} \leq_{\mathbf{P}} \mathbf{X}$ and hence it is very important to know the FROM and TO in a reduction.

To prove $\mathbf{X} \leq_{\mathbf{P}} \mathbf{Y}$ you need to show a reduction FROM $\mathbf{X}$ TO $\mathbf{Y}$
In other words show that an algorithm for $\mathbf{Y}$ implies an algorithm for $\mathbf{X}$.

Vertex Cover

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

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

Vertex Cover

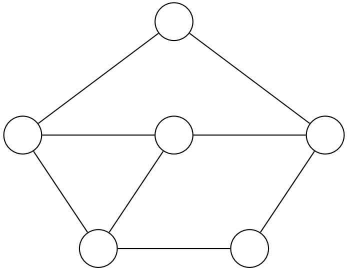
Given a graph $\mathbf{G} = (\mathbf{V}, \mathbf{E})$, a set of vertices $\mathbf{S}$ is:

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

Vertex Cover

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

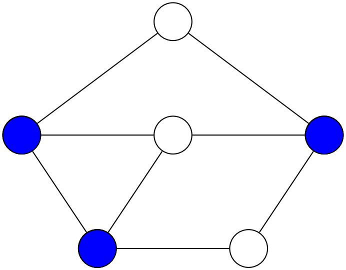
1. A **vertex cover** if every $\mathbf{e} \in \mathbf{E}$ has at least one endpoint in $\mathbf{S}$.



Vertex Cover

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

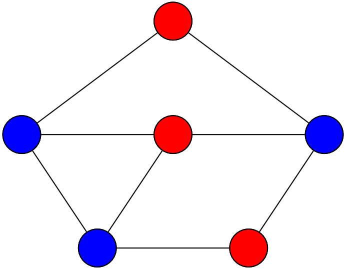
1. A **vertex cover** if every $\mathbf{e} \in \mathbf{E}$ has at least one endpoint in $\mathbf{S}$.



Vertex Cover

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

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



The **Vertex Cover** Problem

Problem (**Vertex Cover**)

Input: *A graph G and integer $\mathbf{k}$.*
Goal: *Is there a vertex cover of size $\leq \mathbf{k}$ in G?*

Can we relate **Independent Set** and **Vertex Cover**?

The **Vertex Cover** Problem

Problem (**Vertex Cover**)

Input: *A graph G and integer* $\mathbf{k}$.
Goal: *Is there a vertex cover of size* $\leq \mathbf{k}$ *in* G*?*

Can we relate **Independent Set** and **Vertex Cover**?

Relationship between...

Vertex Cover and Independent Set

Proposition

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

Proof.

($\Rightarrow$) Let $\mathbf{S}$ be an independent set

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

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

1. Consider $\mathbf{u}, \mathbf{v} \in \mathbf{S}$
2. $\mathbf{uv}$ is not an edge of G, as otherwise $\mathbf{V} \setminus \mathbf{S}$ does not cover $\mathbf{uv}$.
3. $\implies$ $\mathbf{S}$ 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. 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**.

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**.

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. $(\mathbf{G}, \mathbf{k})$ is an instance of **Independent Set** , and $(\mathbf{G}, \mathbf{n} - \mathbf{k})$ 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**.

Independent Set $\leq_P$ Vertex Cover

1. $\mathbf{G}$: graph with $\mathbf{n}$ vertices, and an integer $\mathbf{k}$ be an instance of the **Independent Set** problem.
2. $\mathbf{G}$ has an independent set of size $\geq \mathbf{k}$ iff $\mathbf{G}$ has a vertex cover of size $\leq \mathbf{n} - \mathbf{k}$
3. $(\mathbf{G}, \mathbf{k})$ is an instance of **Independent Set** , and $(\mathbf{G}, \mathbf{n} - \mathbf{k})$ 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**.

A problem of Languages

Suppose you work for the United Nations. Let **U** be the set of all languages spoken by people across the world. The United Nations also has a set of translators, all of whom speak English, and some other languages from **U**.

Due to budget cuts, you can only afford to keep **k** translators on your payroll. Can you do this, while still ensuring that there is someone who speaks every language in **U**?

More General problem: Find/Hire a small group of people who can accomplish a large number of tasks.

A problem of Languages

Suppose you work for the United Nations. Let **U** be the set of all languages spoken by people across the world. The United Nations also has a set of translators, all of whom speak English, and some other languages from **U**.

Due to budget cuts, you can only afford to keep **k** translators on your payroll. Can you do this, while still ensuring that there is someone who speaks every language in **U**?

More General problem: Find/Hire a small group of people who can accomplish a large number of tasks.

A problem of Languages

Suppose you work for the United Nations. Let $\mathbf{U}$ be the set of all languages spoken by people across the world. The United Nations also has a set of translators, all of whom speak English, and some other languages from $\mathbf{U}$.

Due to budget cuts, you can only afford to keep $\mathbf{k}$ translators on your payroll. Can you do this, while still ensuring that there is someone who speaks every language in $\mathbf{U}$?

More General problem: Find/Hire a small group of people who can accomplish a large number of tasks.

The Set Cover Problem

Problem (Set Cover)

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

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

Example

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

The Set Cover Problem

Problem (Set Cover)

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

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

Example

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

The Set Cover Problem

Problem (Set Cover)

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

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

Example

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

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

$\mathbf{\{S_2, S_6\}}$ 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:

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.

Vertex Cover $\leq_P$ Set Cover

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

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

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; $S_v = \{e \mid e \text{ is incident on } v\}$.

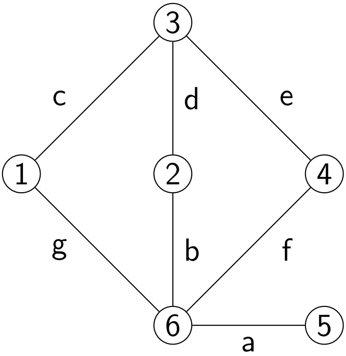
Vertex Cover $\leq_P$ Set Cover

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



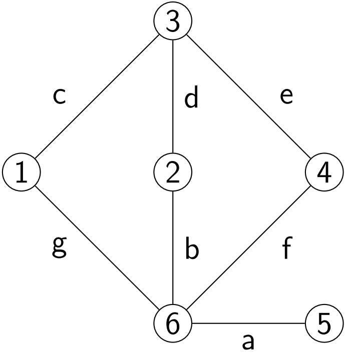
Let $\mathbf{U} = \{a, b, c, d, e, f, g\}$, $\mathbf{k} = \mathbf{2}$ with

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

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

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

Vertex Cover $\leq_P$ Set Cover: Example



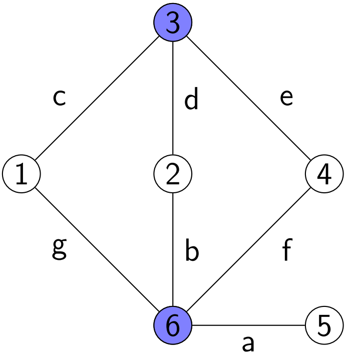
Let $\mathbf{U} = \{a, b, c, d, e, f, g\}$, $\mathbf{k} = 2$ with

$$
\begin{array}{ll}
S_1 = \{c, g\} & S_2 = \{b, d\} \\
S_3 = \{c, d, e\} & S_4 = \{e, f\} \\
S_5 = \{a\} & S_6 = \{a, b, f, g\}
\end{array}
$$

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

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

Vertex Cover $\leq_P$ Set Cover: Example



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

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

Proving Reductions

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

1. Transforms an instance $\mathbf{I_X}$ of $\mathbf{X}$ into an instance $\mathbf{I_Y}$ of $\mathbf{Y}$.
2. Satisfies the property that answer to $\mathbf{I_X}$ is YES iff $\mathbf{I_Y}$ is YES.
    1. typical easy direction to prove: answer to $\mathbf{I_Y}$ is YES if answer to $\mathbf{I_X}$ is YES
    2. typical difficult direction to prove: answer to $\mathbf{I_X}$ is YES if answer to $\mathbf{I_Y}$ is YES (equivalently answer to $\mathbf{I_X}$ is NO if answer to $\mathbf{I_Y}$ 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 $\mathbf{G} = (\mathbf{V}, \mathbf{E})$ obtain a bipartite graph $\mathbf{G}' = (\mathbf{V}', \mathbf{E}')$ as follows.
    1. Let $\mathbf{V_1} = \{\mathbf{u_1} \mid \mathbf{u} \in \mathbf{V}\}$ and $\mathbf{V_2} = \{\mathbf{u_2} \mid \mathbf{u} \in \mathbf{V}\}$. We set $\mathbf{V}' = \mathbf{V_1} \cup \mathbf{V_2}$ (that is, we make two copies of $\mathbf{V}$)
    2. $\mathbf{E}' = \left\{ \mathbf{u_1 v_2} \;\middle|\; \mathbf{u} \neq \mathbf{v} \text{ and } \mathbf{uv} \in \mathbf{E} \right\}$
2. Given $\mathbf{G}$ and integer $\mathbf{k}$ the reduction outputs $\mathbf{G}'$ and $\mathbf{k}$.

Example

"Proof"

Claim

Reduction is a poly-time algorithm. If $\mathbf{G}$ *has a matching of size* $\mathbf{k}$ *then* $\mathbf{G'}$ *has a matching of size* $\mathbf{k}$.

Proof.

Exercise. □

Claim

If $\mathbf{G'}$ *has a matching of size* $\mathbf{k}$ *then* $\mathbf{G}$ *has a matching of size* $\mathbf{k}$.

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

"Proof"

Claim

Reduction is a poly-time algorithm. If $\mathbf{G}$ has a matching of size $\mathbf{k}$ then $\mathbf{G'}$ has a matching of size $\mathbf{k}$.

Proof.

Exercise. □

Claim

If $\mathbf{G'}$ has a matching of size $\mathbf{k}$ then $\mathbf{G}$ has a matching of size $\mathbf{k}$.

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

"Proof"

Claim

Reduction is a poly-time algorithm. If $\mathbf{G}$ has a matching of size $\mathbf{k}$ then $\mathbf{G'}$ has a matching of size $\mathbf{k}$.

Proof.

Exercise. □

Claim

If $\mathbf{G'}$ has a matching of size $\mathbf{k}$ then $\mathbf{G}$ has a matching of size $\mathbf{k}$.

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

"Proof"

Claim

Reduction is a poly-time algorithm. If $\mathbf{G}$ *has a matching of size* $\mathbf{k}$ *then* $\mathbf{G'}$ *has a matching of size* $\mathbf{k}$*.*

Proof.

Exercise. □

Claim

If $\mathbf{G'}$ *has a matching of size* $\mathbf{k}$ *then* $\mathbf{G}$ *has a matching of size* $\mathbf{k}$*.*

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

"Proof"

Claim

Reduction is a poly-time algorithm. If $\mathbf{G}$ *has a matching of size* $\mathbf{k}$ *then* $\mathbf{G'}$ *has a matching of size* $\mathbf{k}$.

Proof.

Exercise. □

Claim

If $\mathbf{G'}$ *has a matching of size* $\mathbf{k}$ *then* $\mathbf{G}$ *has a matching of size* $\mathbf{k}$.

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

Summary

We looked at polynomial-time reductions.

Using polynomial-time reductions

1. If $\mathbf{X} \leq_{\mathbf{P}} \mathbf{Y}$, and there is no efficient algorithm for $\mathbf{X}$, there is no efficient algorithm for $\mathbf{Y}$.

Summary

We looked at polynomial-time reductions.

Using polynomial-time reductions

1. If $\mathbf{X} \leq_{\mathbf{P}} \mathbf{Y}$, and there is no efficient algorithm for $\mathbf{X}$, there is no efficient algorithm for $\mathbf{Y}$.

Summary

We looked at polynomial-time reductions.

Using polynomial-time reductions

1. If $\mathbf{X} \leq_P \mathbf{Y}$, and we have an efficient algorithm for $\mathbf{Y}$, we have an efficient algorithm for $\mathbf{X}$.
2. If $\mathbf{X} \leq_P \mathbf{Y}$, and there is no efficient algorithm for $\mathbf{X}$, there is no efficient algorithm for $\mathbf{Y}$.

Summary

We looked at polynomial-time reductions.

Using polynomial-time reductions

1. If $\mathbf{X} \leq_{\mathbf{P}} \mathbf{Y}$, and we have an efficient algorithm for $\mathbf{Y}$, we have an efficient algorithm for $\mathbf{X}$.
2. If $\mathbf{X} \leq_{\mathbf{P}} \mathbf{Y}$, and there is no efficient algorithm for $\mathbf{X}$, there is no efficient algorithm for $\mathbf{Y}$.

Summary

We looked at polynomial-time reductions.

Using polynomial-time reductions

1. If $\mathbf{X} \leq_{\mathbf{P}} \mathbf{Y}$, and we have an efficient algorithm for $\mathbf{Y}$, we have an efficient algorithm for $\mathbf{X}$.

We looked at some examples of reductions between **Independent Set**, **Clique**, **Vertex Cover**, and **Set Cover**.

Notes

Notes

Notes

Notes