Polynomial Time Reductions

Lecture 21

April 14, 2015

21.1: Introduction to Reductions

21.2: Overview

Reductions

1. Reduction from Problem X to Problem Y (informally): having algorithm for Y, then have algorithm for Problem X.
2. We use reductions to find algorithms to solve problems.
3. We also use reductions to show that we can't find algorithms for some problems. (We say that these problems are hard.)
4. Also, the right reductions might win you a million dollars!

Reductions

1. Reduction from Problem X to Problem Y (informally): having algorithm for Y, then have algorithm for Problem X.
2. We use reductions to find algorithms to solve problems.
3. We also use reductions to show that we can't find algorithms for some problems. (We say that these problems are hard.)
4. Also, the right reductions might win you a million dollars!

Reductions

1. Reduction from Problem X to Problem Y (informally): having algorithm for Y, then have algorithm for Problem X.
2. We use reductions to find algorithms to solve problems.
3. We also use reductions to show that we can't find algorithms for some problems. (We say that these problems are hard.)
4. Also, the right reductions might win you a million dollars!

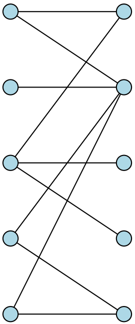
Reductions

1. Reduction from Problem X to Problem Y (informally): having algorithm for Y, then have algorithm for Problem X.
2. We use reductions to find algorithms to solve problems.
3. We also use reductions to show that we can't find algorithms for some problems. (We say that these problems are hard.)
4. 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 $\geq k$?



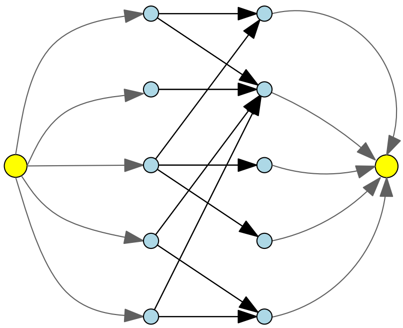
Solution

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



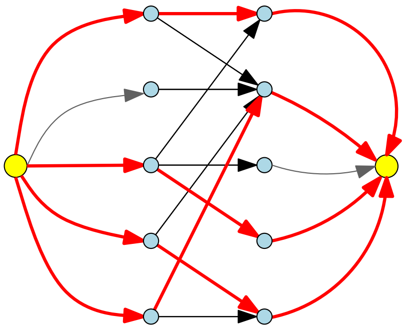
Solution

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



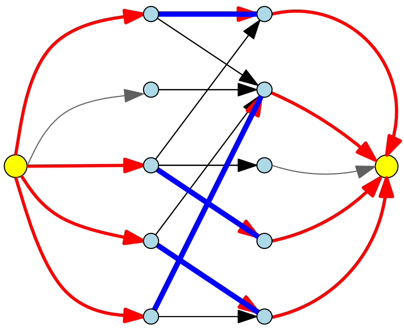
Solution

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 $G = (U \cup V, E)$ 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 $\geq k$.

21.3: Definitions

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.

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...

1. Max-flow as optimization problem:

Problem (Max-Flow optimization version)

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

2. Max-flow as decision problem:

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?

3. 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...

1. Max-flow as optimization problem:

Problem (Max-Flow optimization version)

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

2. Max-flow as decision problem:

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?

3. 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...

1. Max-flow as optimization problem:

Problem (Max-Flow optimization version)

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

2. Max-flow as decision problem:

Problem (Max-Flow decision version)

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

3. 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.

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.

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.

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

1. Instance **Bipartite Matching**: a bipartite graph, and integer k.
2. Solution is "YES" if graph has matching size $\geq k$, else "NO".

---

3. Instance **Max-Flow**: graph G with edge-capacities, two vertices s, t, and an integer k.
4. Solution to instance is "YES" if there is a flow from s to t of value $\geq k$, else "NO".

---

5. An algorithm for a decision Problem X?
6. **Decision algorithm**: Input an instance of X, and outputs either "YES" or "NO".

Examples

1. Instance **Bipartite Matching**: a bipartite graph, and integer k.
2. Solution is "YES" if graph has matching size $\geq k$, else "NO".

---

3. Instance **Max-Flow**: graph G with edge-capacities, two vertices s, t, and an integer k.
4. Solution to instance is "YES" if there is a flow from s to t of value $\geq k$, else "NO".

---

5. An algorithm for a decision Problem X?
6. **Decision algorithm**: Input an instance of X, and outputs either "YES" or "NO".

Examples

1. Instance **Bipartite Matching**: a bipartite graph, and integer k.
2. Solution is "YES" if graph has matching size $\geq k$, else "NO".
3. Instance **Max-Flow**: graph G with edge-capacities, two vertices s, t, and an integer k.
4. Solution to instance is "YES" if there is a flow from s to t of value $\geq k$, else "NO".
5. An algorithm for a decision Problem X?
6. **Decision algorithm**: Input an instance of X, and outputs either "YES" or "NO".

Examples

1. Instance **Bipartite Matching**: a bipartite graph, and integer k.
2. Solution is "YES" if graph has matching size $\geq k$, else "NO".

---

3. Instance **Max-Flow**: graph G with edge-capacities, two vertices s, t, and an integer k.
4. Solution to instance is "YES" if there is a flow from s to t of value $\geq k$, else "NO".

---

5. An algorithm for a decision Problem X?
6. **Decision algorithm**: Input an instance of X, and outputs either "YES" or "NO".

Examples

1. Instance **Bipartite Matching**: a bipartite graph, and integer k.
2. Solution is "YES" if graph has matching size $\geq k$, else "NO".

---

3. Instance **Max-Flow**: graph G with edge-capacities, two vertices s, t, and an integer k.
4. Solution to instance is "YES" if there is a flow from s to t of value $\geq k$, else "NO".

---

5. An algorithm for a decision Problem X?
6. **Decision algorithm**: Input an instance of X, and outputs either "YES" or "NO".

Examples

1. Instance **Bipartite Matching**: a bipartite graph, and integer k.
2. Solution is "YES" if graph has matching size $\geq k$, else "NO".

---

3. Instance **Max-Flow**: graph G with edge-capacities, two vertices s, t, and an integer k.
4. Solution to instance is "YES" if there is a flow from s to t of value $\geq k$, else "NO".

---

5. An algorithm for a decision Problem X?
6. **Decision algorithm**: Input an instance of X, and outputs either "YES" or "NO".

Examples

1. Instance **Bipartite Matching**: a bipartite graph, and integer k.
2. Solution is "YES" if graph has matching size $\geq k$, else "NO".

---

3. Instance **Max-Flow**: graph G with edge-capacities, two vertices s, t, and an integer k.
4. Solution to instance is "YES" if there is a flow from s to t of value $\geq k$, else "NO".

---

5. An algorithm for a decision Problem X?
6. **Decision algorithm**: Input an instance of X, and outputs either "YES" or "NO".

Examples

1. Instance **Bipartite Matching**: a bipartite graph, and integer k.
2. Solution is "YES" if graph has matching size $\geq k$, else "NO".

---

3. Instance **Max-Flow**: graph G with edge-capacities, two vertices s, t, and an integer k.
4. Solution to instance is "YES" if there is a flow from s to t of value $\geq k$, else "NO".

---

5. An algorithm for a decision Problem X?
6. **Decision algorithm**: Input an instance of X, and outputs either "YES" or "NO".

Examples

1. Instance **Bipartite Matching**: a bipartite graph, and integer k.
2. Solution is "YES" if graph has matching size $\geq k$, else "NO".

---

3. Instance **Max-Flow**: graph G with edge-capacities, two vertices s, t, and an integer k.
4. Solution to instance is "YES" if there is a flow from s to t of value $\geq k$, else "NO".

---

5. An algorithm for a decision Problem X?
6. **Decision algorithm**: Input an instance of 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).

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).

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).

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).

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.
3. Language $\equiv$ decision problem.
    1. For any language $L \implies$ there is a decision problem Π_L.
    2. For any decision problem $\Pi \implies$ an associated language L_Π.
4. Given L, Π_L is the 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.
5. Given Π the associated language is $L_\Pi = \left\{ I \;\middle|\; I \text{ is an instance of } \Pi \text{ for which answer is YES} \right\}$.
6. 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.
3. Language $\equiv$ decision problem.
   1. For any language $L \implies$ there is a decision problem Π_L.
   2. For any decision problem $\Pi \implies$ an associated language L_Π.
4. Given L, Π_L is the 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.
5. Given Π the associated language is $L_\Pi = \left\{ I \;\middle|\; I \text{ is an instance of } \Pi \text{ for which answer is YES} \right\}$.
6. 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.
3. Language $\equiv$ decision problem.
   1. For any language $L \implies$ there is a decision problem Π_L.
   2. For any decision problem $\Pi \implies$ an associated language L_Π.
4. Given L, Π_L is the 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.
5. Given Π the associated language is $L_\Pi = \left\{ I \;\middle|\; I \text{ is an instance of } \Pi \text{ for which answer is YES} \right\}$.
6. 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.
3. Language $\equiv$ decision problem.
   1. For any language $L \implies$ there is a decision problem Π_L.
   2. For any decision problem $\Pi \implies$ an associated language L_Π.
4. Given L, Π_L is the 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.
5. Given Π the associated language is $L_\Pi = \left\{ I \;\middle|\; I \text{ is an instance of } \Pi \text{ for which answer is YES} \right\}$.
6. 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.
3. Language $\equiv$ decision problem.
    1. For any language $L \implies$ there is a decision problem Π_L.
    2. For any decision problem $\Pi \implies$ an associated language L_Π.
4. Given L, Π_L is the 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.
5. Given Π the associated language is
   $L_\Pi = \left\{ I \,\middle|\, I \text{ is an instance of } \Pi \text{ for which answer is YES} \right\}$.
6. 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.
3. Language $\equiv$ decision problem.
   1. For any language $L \implies$ there is a decision problem Π_L.
   2. For any decision problem $\Pi \implies$ an associated language L_Π.
4. Given L, Π_L is the 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.
5. Given Π the associated language is $L_\Pi = \left\{ I \;\middle|\; I \text{ is an instance of } \Pi \text{ for which answer is YES} \right\}$.
6. 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.
3. Language $\equiv$ decision problem.
   1. For any language $L \implies$ there is a decision problem Π_L.
   2. For any decision problem $\Pi \implies$ an associated language L_Π.
4. Given L, Π_L is the 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.
5. Given Π the associated language is $L_\Pi = \left\{ I \;\middle|\; I \text{ is an instance of } \Pi \text{ for which answer is YES} \right\}$.
6. 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.
3. Language $\equiv$ decision problem.
    1. For any language $L \implies$ there is a decision problem Π_L.
    2. For any decision problem $\Pi \implies$ an associated language L_Π.
4. Given L, Π_L is the 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.
5. Given Π the associated language is
$$L_\Pi = \left\{ I \;\middle|\; I \text{ is an instance of } \Pi \text{ for which answer is YES} \right\}.$$
6. Thus, decision problems and languages are used interchangeably.

Example

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

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

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

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

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

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

Example

1. The decision problem **Primality**, and the language
$$L = \left\{ \#p \;\middle|\; p \text{ is a prime number} \right\}.$$
Here $\#p$ is the string in base $\mathbf{10}$ representing p.
2. **Bipartite** (is given graph is bipartite. The language is
$$L = \left\{ \mathcal{S}(\mathsf{G}) \;\middle|\; \mathsf{G} \text{ is a bipartite graph} \right\}.$$
Here $\mathcal{S}(\mathsf{G})$ is the string encoding the graph G.

Reductions, revised.

1. For decision problems X, Y, a **reduction from X to Y** is:
   1. An algorithm ...
   2. Input: I_X, an instance of X.
   3. Output: I_Y an instance of Y.
   4. Such that:

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

2. There are other kinds of reductions.

Reductions, revised.

1. For decision problems X, Y, a **reduction from X to Y** is:
   1. An algorithm ...
   2. Input: I_X, an instance of X.
   3. Output: I_Y an instance of Y.
   4. Such that:

      $\boxed{I_Y \text{ is YES instance of } Y} \iff \boxed{I_X \text{ is YES instance of } X}$
2. There are other kinds of reductions.

Reductions, revised.

1. For decision problems X, Y, a **reduction from X to Y** is:
   1. An algorithm ...
   2. Input: I_X, an instance of X.
   3. Output: I_Y an instance of Y.
   4. Such that:

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

2. There are other kinds of reductions.

Reductions, revised.

1. For decision problems X, Y, a **reduction from X to Y** is:
   1. An algorithm ...
   2. Input: I_X, an instance of X.
   3. Output: I_Y an instance of Y.
   4. Such that:

   $$\boxed{I_Y \text{ is YES instance of } Y} \iff \boxed{I_X \text{ is YES instance of } X}$$
2. There are other kinds of reductions.

Using reductions to solve problems

1. $\mathcal{R}$: Reduction $X \rightarrow Y$
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 $\mathcal{A}_Y(I_Y)$

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

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:

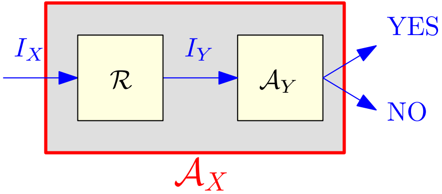
$\mathcal{A}_X(I_X)$:
    // I_X: instance of X.
    $I_Y \Leftarrow \mathcal{R}(I_X)$
    return $\mathcal{A}_Y(I_Y)$

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

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 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.

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.

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.

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.

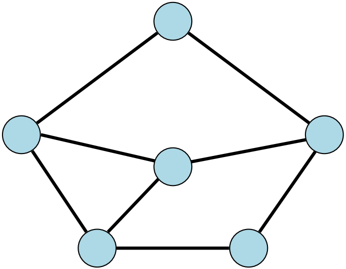
Comparing Problems

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.

21.4: Examples of Reductions

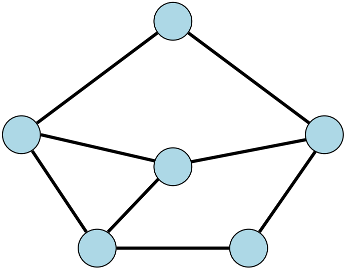
21.4.1: Independent Set and Clique

Independent Sets and Cliques



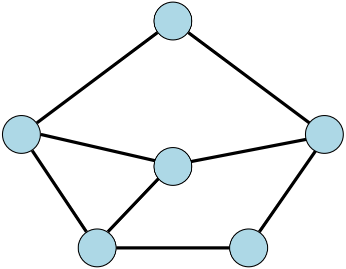
1. Given a graph G.
2. A set of vertices V' is an **independent set**: no two vertices of V' connected by an edge.
3. **clique**: *every* pair of vertices in V' is connected by an edge of G.

Independent Sets and Cliques

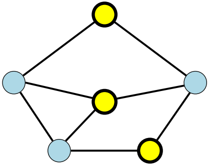


1. Given a graph G.
2. A set of vertices V' is an **independent set**: no two vertices of V' connected by an edge.
3. **clique**: *every* pair of vertices in V' is connected by an edge of G.

Independent Sets and Cliques



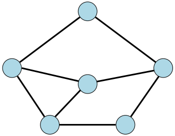
1. Given a graph G.

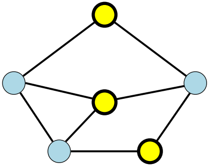


2. A set of vertices V' is an **independent set**: no two vertices of V' connected by an edge.

3. **clique**: *every* pair of vertices in V' is connected by an edge of G.

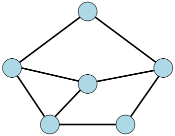
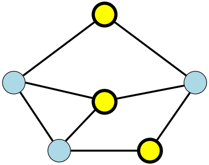
Independent Sets and Cliques

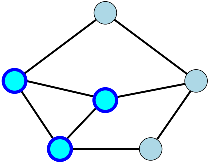
1. Given a graph G. 

2. A set of vertices V' is an **independent set**: no two vertices of V' connected by an edge. 

3. **clique**: *every* pair of vertices in V' is connected by an edge of G.

Independent Sets and Cliques

1. Given a graph G. 
2. A set of vertices V' is an **independent set**: no two vertices of V' connected by an edge. 
3. **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 $\geq k$?

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 $\geq k$?

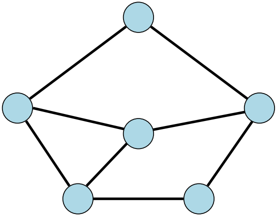
Recall

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

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

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

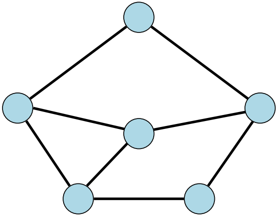
G:



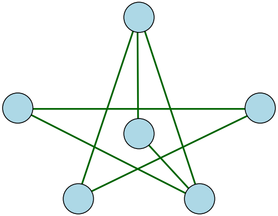
1. An instance of **Independent Set** is a graph G and an integer k.
2. Convert G to $\overline{G}$, in which (u, v) is an edge $\iff$ (u, v) is not an edge of G. ($\overline{G}$ is the *complement* of G.)
3. $([])\overline{G}, k$: instance of **Clique**.

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

G:



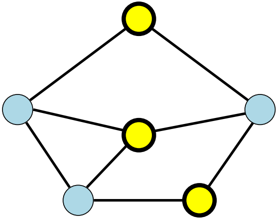
$\overline{G}$:



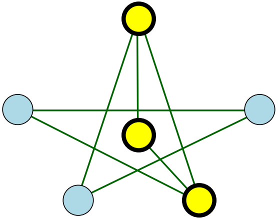
1. An instance of **Independent Set** is a graph G and an integer k.
2. Convert G to $\overline{G}$, in which (u, v) is an edge $\iff$ (u, v) is not an edge of G. ($\overline{G}$ is the *complement* of G.)
3. $([)]\overline{G}, k$: instance of **Clique**.

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

G:



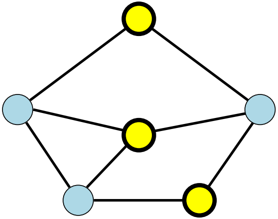
$\overline{G}$:



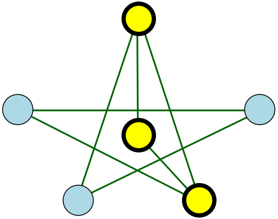
1. An instance of **Independent Set** is a graph G and an integer k.
2. Convert G to $\overline{G}$, in which (u, v) is an edge $\iff$ (u, v) is not an edge of G. ($\overline{G}$ is the *complement* of G.)
3. $([)]\overline{G}, k$: instance of **Clique**.

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

G:



$\overline{G}$:



1. An instance of **Independent Set** is a graph G and an integer k.
2. 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.)
3. $\mathbf{([)]}\overline{G}, k$: 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**.

21.4.2: NFAs/DFAs and Universality

DFAs and NFAs

1. DFAs (Remember 373?) are determinstic automata that accept regular languages.
2. NFAs are the same, except that non-deterministic.
3. Every NFA can be converted to a DFA that accepts the same language using the subset construction.
4. (How long does this take?)
5. The smallest DFA equivalent to an NFA with n states may have $\approx 2^n$ states.

DFAs and NFAs

1. DFAs (Remember 373?) are determinstic automata that accept regular languages.
2. NFAs are the same, except that non-deterministic.
3. Every NFA can be converted to a DFA that accepts the same language using the subset construction.
4. (How long does this take?)
5. The smallest DFA equivalent to an NFA with n states may have $\approx 2^n$ states.

DFAs and NFAs

1. DFAs (Remember 373?) are determinstic automata that accept regular languages.
2. NFAs are the same, except that non-deterministic.
3. Every NFA can be converted to a DFA that accepts the same language using the subset construction.
4. (How long does this take?)
5. The smallest DFA equivalent to an NFA with n states may have $\approx 2^n$ states.

DFAs and NFAs

1. DFAs (Remember 373?) are determinstic automata that accept regular languages.
2. NFAs are the same, except that non-deterministic.
3. Every NFA can be converted to a DFA that accepts the same language using the subset construction.
4. (How long does this take?)
5. The smallest DFA equivalent to an NFA with n states may have $\approx 2^n$ states.

DFAs and NFAs

1. DFAs (Remember 373?) are determinstic automata that accept regular languages.
2. NFAs are the same, except that non-deterministic.
3. Every NFA can be converted to a DFA that accepts the same language using the subset construction.
4. (How long does this take?)
5. The smallest DFA equivalent to an NFA with n states may have $\approx 2^n$ states.

DFA Universality

1. A DFA M is universal if it accepts every string.
2. That is, $L(M) = \Sigma^*$, the set of all strings.
3. DFA universality problem:

Problem (DFA universality)

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

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

DFA Universality

1. A DFA M is universal if it accepts every string.
2. That is, $L(M) = \Sigma^*$, the set of all strings.
3. DFA universality problem:

Problem (DFA universality)

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

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

DFA Universality

1. A DFA M is universal if it accepts every string.
2. That is, $L(M) = \Sigma^*$, the set of all strings.
3. DFA universality problem:

Problem (DFA universality)

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

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

DFA Universality

1. A DFA M is universal if it accepts every string.
2. That is, $L(M) = \Sigma^*$, the set of all strings.
3. DFA universality problem:

Problem (DFA universality)

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

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

DFA Universality

1. A DFA M is universal if it accepts every string.
2. That is, $L(M) = \Sigma^*$, the set of all strings.
3. DFA universality problem:

Problem (DFA universality)

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

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

DFA Universality

1. A DFA M is universal if it accepts every string.
2. That is, $L(M) = \Sigma^*$, the set of all strings.
3. DFA universality problem:

Problem (DFA universality)

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

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

NFA Universality

1. An NFA N is universal if it accepts every string. That is, $L(N) = \Sigma^*$, the set of all strings.
2. NFA universality problem:

Problem (NFA universality)

Input: *A* NFA M.
Goal: *Is* M *universal?*

3. How do we solve **NFA Universality**?
4. Reduce it to **DFA Universality**...
5. Given an NFA N, convert it to an equivalent DFA M, and use the **DFA Universality** Algorithm.
6. The reduction takes exponential time!

NFA Universality

1. An NFA N is universal if it accepts every string. That is, $L(N) = \Sigma^*$, the set of all strings.
2. NFA universality problem:

Problem (NFA universality)

Input: *A* NFA M.
Goal: *Is* M *universal?*

3. How do we solve **NFA Universality**?
4. Reduce it to **DFA Universality**...
5. Given an NFA N, convert it to an equivalent DFA M, and use the **DFA Universality** Algorithm.
6. The reduction takes exponential time!

NFA Universality

1. An NFA N is universal if it accepts every string. That is, $L(N) = \Sigma^*$, the set of all strings.
2. NFA universality problem:

Problem (NFA universality)

Input: *A* NFA M.
Goal: *Is* M *universal?*

3. How do we solve **NFA Universality**?
4. Reduce it to **DFA Universality**...
5. Given an NFA N, convert it to an equivalent DFA M, and use the **DFA Universality** Algorithm.
6. The reduction takes exponential time!

NFA Universality

1. An NFA N is universal if it accepts every string. That is, $L(N) = \Sigma^*$, the set of all strings.
2. NFA universality problem:

Problem (NFA universality)

Input: *A NFA M.*
Goal: *Is M universal?*

3. How do we solve **NFA Universality**?
4. Reduce it to **DFA Universality**...
5. Given an NFA N, convert it to an equivalent DFA M, and use the **DFA Universality** Algorithm.
6. The reduction takes exponential time!

NFA Universality

1. An NFA N is universal if it accepts every string. That is, $L(N) = \Sigma^*$, the set of all strings.
2. NFA universality problem:

Problem (NFA universality)

Input: *A* NFA M.
Goal: *Is* M *universal?*

3. How do we solve **NFA Universality**?
4. Reduce it to **DFA Universality**...
5. Given an NFA N, convert it to an equivalent DFA M, and use the **DFA Universality** Algorithm.
6. The reduction takes exponential time!

NFA Universality

1. An NFA N is universal if it accepts every string. That is, $L(N) = \Sigma^*$, the set of all strings.
2. NFA universality problem:

Problem (NFA universality)

Input: *A* NFA M.
Goal: *Is* M *universal?*

3. How do we solve **NFA Universality**?
4. Reduce it to **DFA Universality**...
5. Given an NFA N, convert it to an equivalent DFA M, and use the **DFA Universality** Algorithm.
6. The reduction takes exponential time!

Polynomial-time reductions

1. An algorithm is **efficient** if it runs in polynomial-time.
2. To find efficient algorithms for problems, we are only interested in polynomial-time reductions. Reductions that take longer are not useful.
3. If we have a polynomial-time reduction from problem X to problem Y (we write $X \leq_P Y$), and a poly-time algorithm $\mathcal{A}_Y$ for Y, we have a polynomial-time/efficient algorithm for X.

Polynomial-time reductions

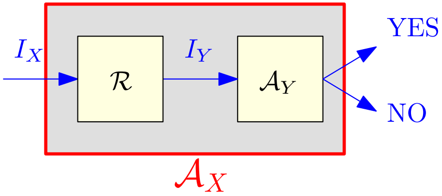
1. An algorithm is **efficient** if it runs in polynomial-time.
2. To find efficient algorithms for problems, we are only interested in polynomial-time reductions. Reductions that take longer are not useful.
3. If we have a polynomial-time reduction from problem X to problem Y (we write $X \leq_P Y$), and a poly-time algorithm $\mathcal{A}_Y$ for Y, we have a polynomial-time/efficient algorithm for X.

Polynomial-time reductions

1. An algorithm is **efficient** if it runs in polynomial-time.
2. To find efficient algorithms for problems, we are only interested in polynomial-time reductions. Reductions that take longer are not useful.
3. If we have a polynomial-time reduction from problem X to problem Y (we write $X \leq_P Y$), and a poly-time algorithm $\mathcal{A}_Y$ for Y, we have a polynomial-time/efficient algorithm for X.

Polynomial-time reductions

1. An algorithm is **efficient** if it runs in polynomial-time.
2. To find efficient algorithms for problems, we are only interested in polynomial-time reductions. Reductions that take longer are not useful.
3. If we have a polynomial-time reduction from problem X to problem Y (we write $X \leq_P Y$), and a poly-time algorithm $\mathcal{A}_Y$ for Y, we have a polynomial-time/efficient algorithm for X.



Polynomial-time Reduction

1. 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.
2. Polynomial transitivity:

Proposition

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

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

Polynomial-time Reduction

1. 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.
2. Polynomial transitivity:

Proposition

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

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

Polynomial-time Reduction

1. 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.
2. Polynomial transitivity:

Proposition

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

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

Polynomial-time reductions and hardness

1. For decision problems X and Y, if $X \leq_P Y$, and Y has an efficient algorithm, X has an efficient algorithm.
2. If you believe that **Independent Set** does not have an efficient algorithm, why should you believe the same of **Clique**?
3. Because we showed **Independent Set** $\leq_P$ **Clique**. If **Clique** had an efficient algorithm, so would **Independent Set**!
4. If $X \leq_P Y$ and X does not have an efficient algorithm, Y cannot have an efficient algorithm!

Polynomial-time reductions and hardness

1. For decision problems X and Y, if $X \le_P Y$, and Y has an efficient algorithm, X has an efficient algorithm.
2. If you believe that **Independent Set** does not have an efficient algorithm, why should you believe the same of **Clique**?
3. Because we showed **Independent Set** $\le_P$ **Clique**. If **Clique** had an efficient algorithm, so would **Independent Set**!
4. If $X \le_P Y$ and X does not have an efficient algorithm, Y cannot have an efficient algorithm!

Polynomial-time reductions and hardness

1. For decision problems X and Y, if $X \leq_P Y$, and Y has an efficient algorithm, X has an efficient algorithm.
2. If you believe that **Independent Set** does not have an efficient algorithm, why should you believe the same of **Clique**?
3. Because we showed **Independent Set** $\leq_P$ **Clique**. If **Clique** had an efficient algorithm, so would **Independent Set**!
4. If $X \leq_P Y$ and X does not have an efficient algorithm, Y cannot have an efficient algorithm!

Polynomial-time reductions and hardness

1. For decision problems X and Y, if $X \leq_P Y$, and Y has an efficient algorithm, X has an efficient algorithm.
2. If you believe that **Independent Set** does not have an efficient algorithm, why should you believe the same of **Clique**?
3. Because we showed **Independent Set** $\leq_P$ **Clique**. If **Clique** had an efficient algorithm, so would **Independent Set**!
4. If $X \leq_P Y$ and X does not have an efficient algorithm, Y cannot have an efficient algorithm!

Polynomial-time reductions and hardness

1. For decision problems X and Y, if $X \leq_P Y$, and Y has an efficient algorithm, X has an efficient algorithm.
2. If you believe that **Independent Set** does not have an efficient algorithm, why should you believe the same of **Clique**?
3. Because we showed **Independent Set** $\leq_P$ **Clique**. If **Clique** had an efficient algorithm, so would **Independent Set**!
4. 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 X to Y. Then for any instance I_X of X, the size of the instance I_Y of Y produced from I_X by $\mathcal{R}$ is polynomial in the size of I_X.

Proof.

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

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

Proof.

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

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

Proof.

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

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

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

Transitivity of Reductions

1. Reductions are transitive:

Proposition

$X \leq_P Y$ and $Y \leq_P Z$ implies that $X \leq_P Z$.

2. 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.
3. To prove $X \leq_P Y$ you need to show a reduction FROM X TO Y.
4. In other words show that an algorithm for Y implies an algorithm for X.

Transitivity of Reductions

1. Reductions are transitive:

Proposition

$X \leq_P Y$ *and* $Y \leq_P Z$ *implies that* $X \leq_P Z$.

2. 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.
3. To prove $X \leq_P Y$ you need to show a reduction FROM X TO Y.
4. In other words show that an algorithm for Y implies an algorithm for X.

Transitivity of Reductions

1. Reductions are transitive:

Proposition

$X \leq_P Y$ and $Y \leq_P Z$ implies that $X \leq_P Z$.

2. 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.
3. To prove $X \leq_P Y$ you need to show a reduction FROM X TO Y.
4. In other words show that an algorithm for Y implies an algorithm for X.

Transitivity of Reductions

1. Reductions are transitive:

Proposition

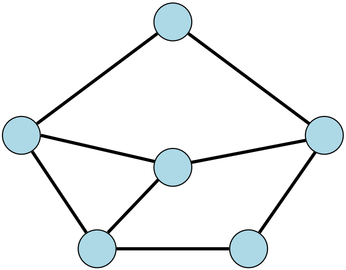
$X \leq_P Y$ and $Y \leq_P Z$ implies that $X \leq_P Z$.

2. 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.
3. To prove $X \leq_P Y$ you need to show a reduction FROM X TO Y.
4. In other words show that an algorithm for Y implies an algorithm for X.

21.5: Independent Set and Vertex Cover

Vertex Cover

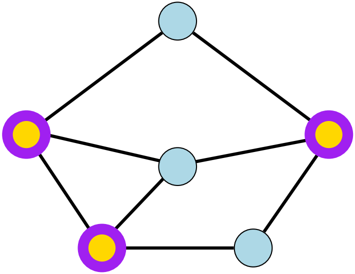
Given a graph $G = (V, E)$, a set of vertices S is:



Vertex Cover

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

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



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**?

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**?

Proposition

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

Proof.

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

1. Consider any edge $uv \in E$.
2. Since S is an independent set, either $u \notin S$ 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 $u, v \in S$
2. uv is not an edge of G, as otherwise $V \setminus S$ does not cover uv.
3. $\implies$ S is thus an independent set. □

Relationship between...

Vertex Cover and Independent Set

Proposition

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

Proof.

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

1. Consider any edge $uv \in E$.
2. Since S is an independent set, either $u \notin S$ 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 $u, v \in S$
2. uv is not an edge of G, as otherwise $V \setminus S$ does not cover uv.
3. $\implies$ S is thus an independent set. □

Relationship between...

Vertex Cover and Independent Set

Proposition

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

Proof.

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

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

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

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

Relationship between...

Vertex Cover and Independent Set

Proposition

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

Proof.

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

1. Consider any edge $uv \in E$.
2. Since S is an independent set, either $u \notin S$ 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 $u, v \in S$
2. uv is not an edge of G, as otherwise $V \setminus S$ does not cover uv.
3. $\implies$ S is thus an independent set. □

Relationship between...

Vertex Cover and Independent Set

Proposition

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

Proof.

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

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

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

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

Relationship between...

Vertex Cover and Independent Set

Proposition

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

Proof.

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

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

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

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

Proposition

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

Proof.

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

1. Consider any edge $uv \in E$.
2. Since S is an independent set, either $u \notin S$ 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 $u, v \in S$
2. uv is not an edge of G, as otherwise $V \setminus S$ does not cover uv.
3. $\implies$ S is thus an independent set. □

Relationship between...

Vertex Cover and Independent Set

Proposition

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

Proof.

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

1. Consider any edge $uv \in E$.
2. Since S is an independent set, either $u \notin S$ 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 $u, v \in S$
2. uv is not an edge of G, as otherwise $V \setminus S$ does not cover uv.
3. $\implies$ S is thus an independent set. $\square$

Relationship between...

Vertex Cover and Independent Set

Proposition

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

Proof.

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

1. Consider any edge $uv \in E$.
2. Since S is an independent set, either $u \notin S$ 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 $u, v \in S$
2. uv is not an edge of G, as otherwise $V \setminus S$ does not cover uv.
3. $\implies$ S is thus an independent set. $\square$

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. (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**.

21.6: Vertex Cover and Set Cover

A problem of Languages

1. 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.
2. 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?
3. More General problem: Find/Hire a small group of people who can accomplish a large number of tasks.

A problem of Languages

1. 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.
2. 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?
3. More General problem: Find/Hire a small group of people who can accomplish a large number of tasks.

A problem of Languages

1. 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.
2. 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?
3. More General problem: Find/Hire a small group of people who can accomplish a large number of tasks.

A problem of Languages

1. 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.
2. 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?
3. 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 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

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

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:

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 $G = (V, E)$ 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; $S_v = \{e \mid e \text{ is incident on } v\}$.

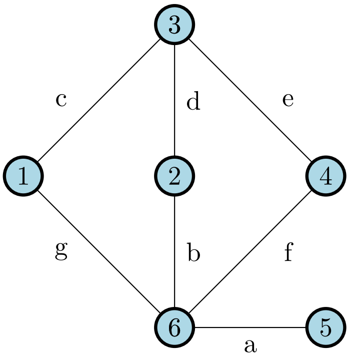
Vertex Cover $\leq_P$ Set Cover

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



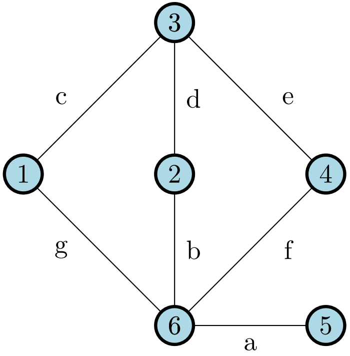
Let $U = \{a, b, c, d, e, f, g\}$, $k = 2$ with

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

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

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

Vertex Cover $\leq_P$ Set Cover: Example



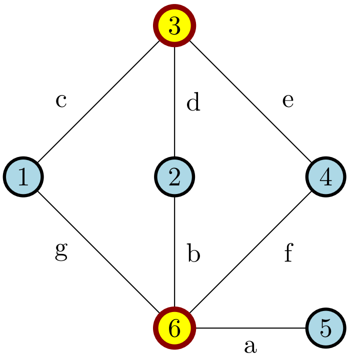
Let $U = \{a, b, c, d, e, f, g\}$, $k = 2$ with

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

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

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

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.
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_X is NO if answer to I_Y is NO).
3. Runs in **polynomial** time.

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_X is NO if answer to I_Y is NO).
3. Runs in **polynomial** time.

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_X is NO if answer to 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 $G = (V, E)$ 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.

Example

"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!

"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!

"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. $\square$

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!

"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!

"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. $\square$

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!

Summary

1. We looked at polynomial-time reductions.
2. Using polynomial-time reductions
   1. If $X \leq_P Y$, and we have an efficient algorithm for Y, we have an efficient algorithm for X.
   2. If $X \leq_P Y$, and there is no efficient algorithm for X, there is no efficient algorithm for Y.
3. We looked at some examples of reductions between **Independent Set**, **Clique**, **Vertex Cover**, and **Set Cover**.

Summary

1. We looked at polynomial-time reductions.
2. Using polynomial-time reductions
   1. If $X \leq_P Y$, and we have an efficient algorithm for Y, we have an efficient algorithm for X.
   2. If $X \leq_P Y$, and there is no efficient algorithm for X, there is no efficient algorithm for Y.
3. We looked at some examples of reductions between **Independent Set**, **Clique**, **Vertex Cover**, and **Set Cover**.

Summary

1. We looked at polynomial-time reductions.
2. Using polynomial-time reductions
    1. If $X \leq_P Y$, and we have an efficient algorithm for Y, we have an efficient algorithm for X.
    2. If $X \leq_P Y$, and there is no efficient algorithm for X, there is no efficient algorithm for Y.
3. We looked at some examples of reductions between **Independent Set**, **Clique**, **Vertex Cover**, and **Set Cover**.

Summary

1. We looked at polynomial-time reductions.
2. Using polynomial-time reductions
   1. If $X \leq_P Y$, and we have an efficient algorithm for Y, we have an efficient algorithm for X.
   2. If $X \leq_P Y$, and there is no efficient algorithm for X, there is no efficient algorithm for Y.
3. We looked at some examples of reductions between **Independent Set**, **Clique**, **Vertex Cover**, and **Set Cover**.

Summary

1. We looked at polynomial-time reductions.
2. Using polynomial-time reductions
   1. If $X \leq_P Y$, and we have an efficient algorithm for Y, we have an efficient algorithm for X.
   2. If $X \leq_P Y$, and there is no efficient algorithm for X, there is no efficient algorithm for Y.
3. We looked at some examples of reductions between **Independent Set**, **Clique**, **Vertex Cover**, and **Set Cover**.

Summary

1. We looked at polynomial-time reductions.
2. Using polynomial-time reductions
   1. If $X \leq_P Y$, and we have an efficient algorithm for Y, we have an efficient algorithm for X.
   2. If $X \leq_P Y$, and there is no efficient algorithm for X, there is no efficient algorithm for Y.
3. We looked at some examples of reductions between **Independent Set**, **Clique**, **Vertex Cover**, and **Set Cover**.

Notes

Notes

Notes

Notes