



CS201 DISCRETE MATHEMATICS FOR COMPUTER SCIENCE

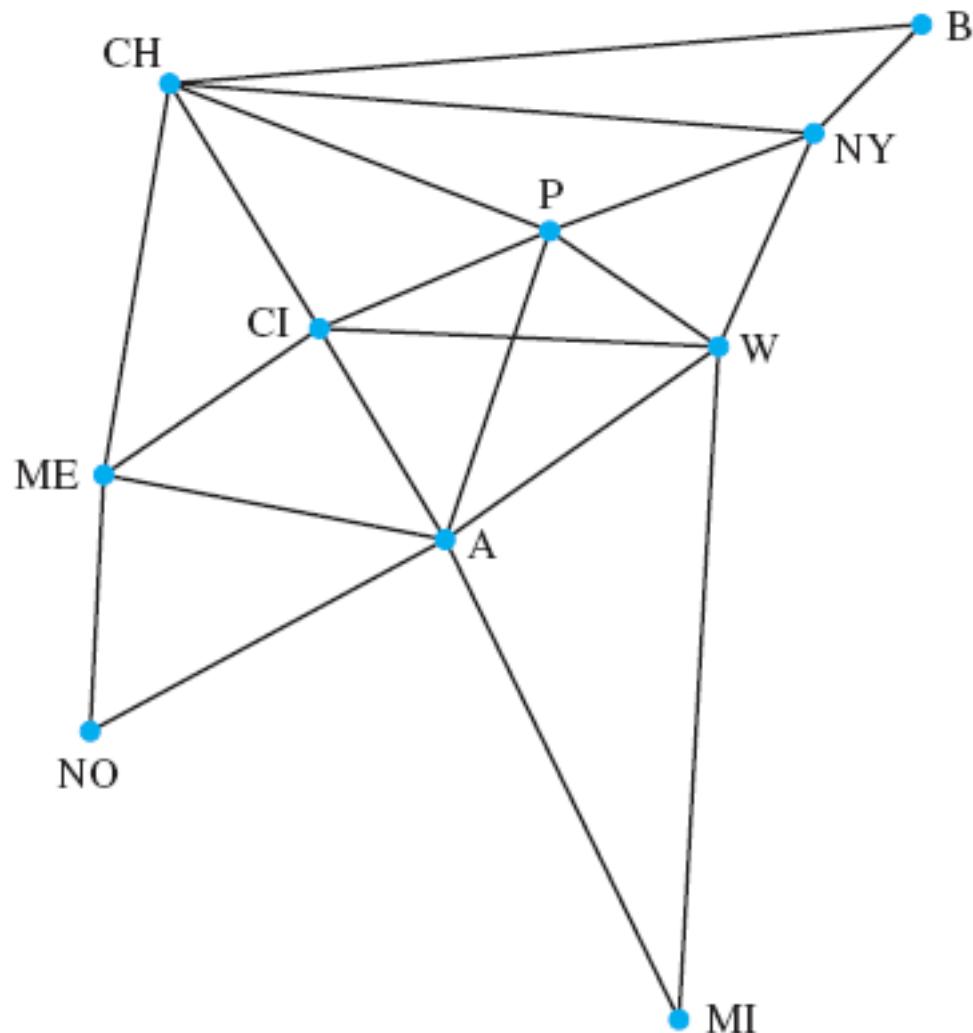
Dr. QI WANG

Department of Computer Science and Engineering

Office: Room413, CoE South Tower

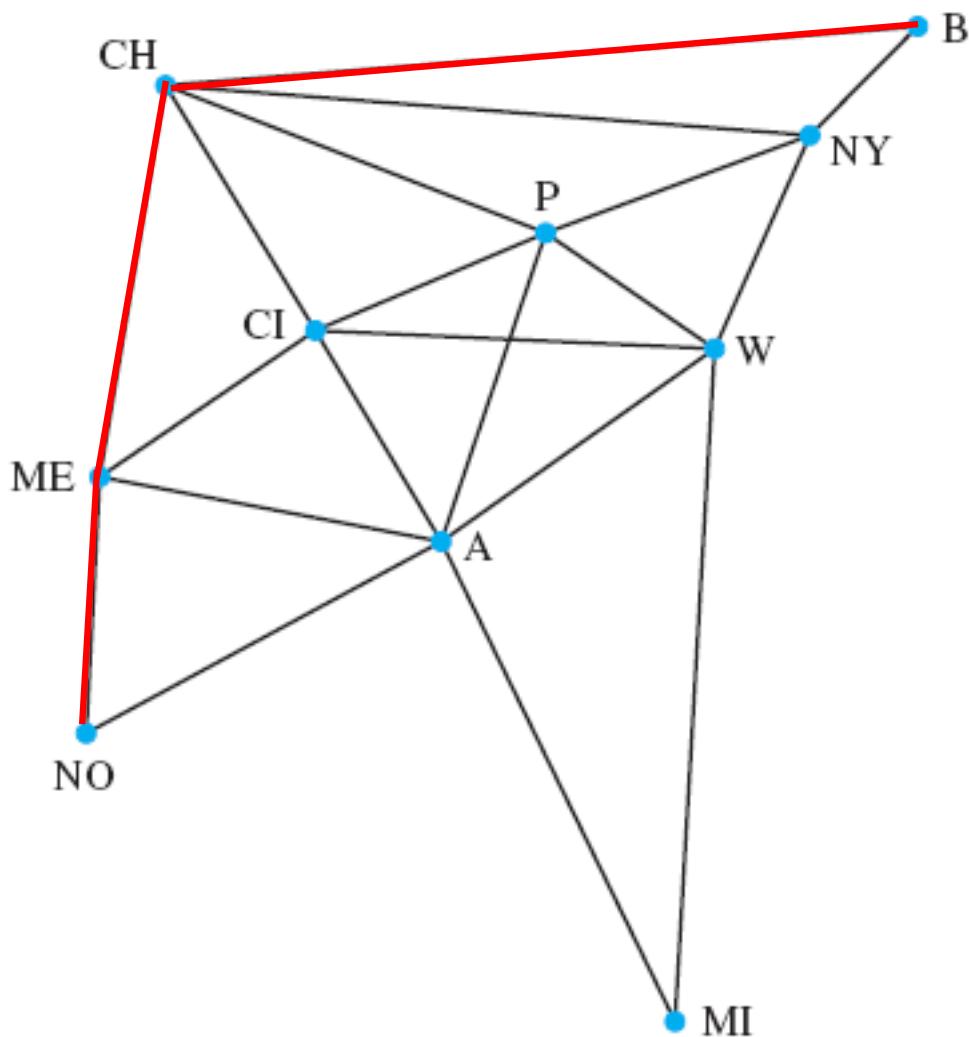
Email: wangqi@sustech.edu.cn

Example



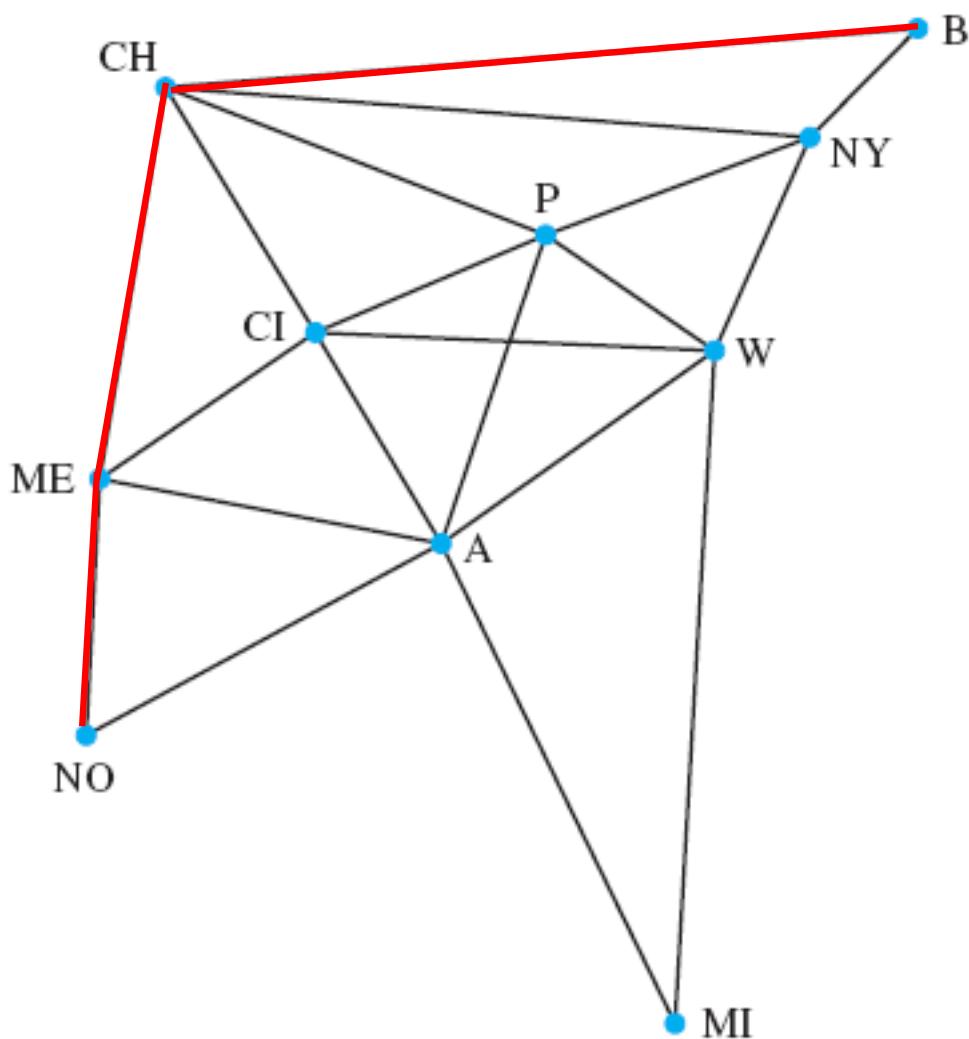
What is the **minimum number** of links to send a message from **B** to **NO**?

Example



What is the **minimum number** of links to send a message from **B** to **NO**?

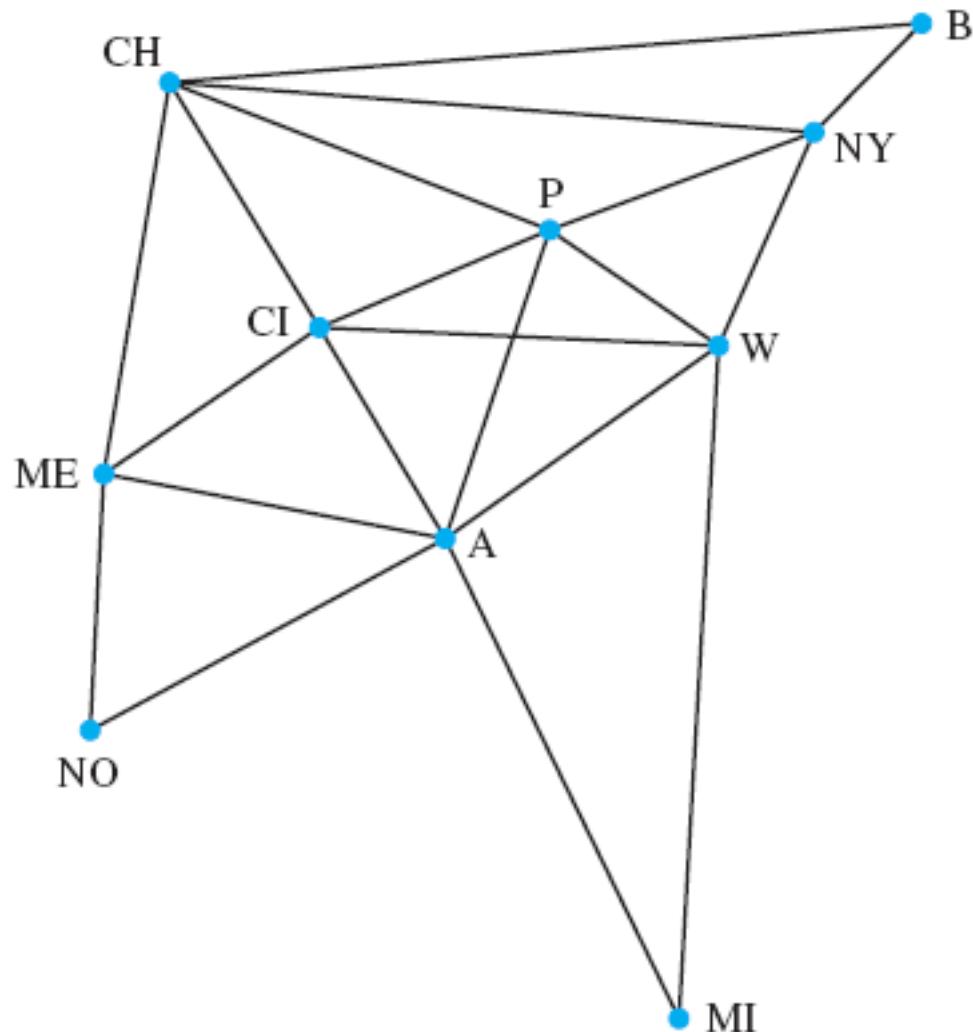
Example



What is the **minimum number** of links to send a message from **B** to **NO**?

3: **B - CH - ME - NO**

Example

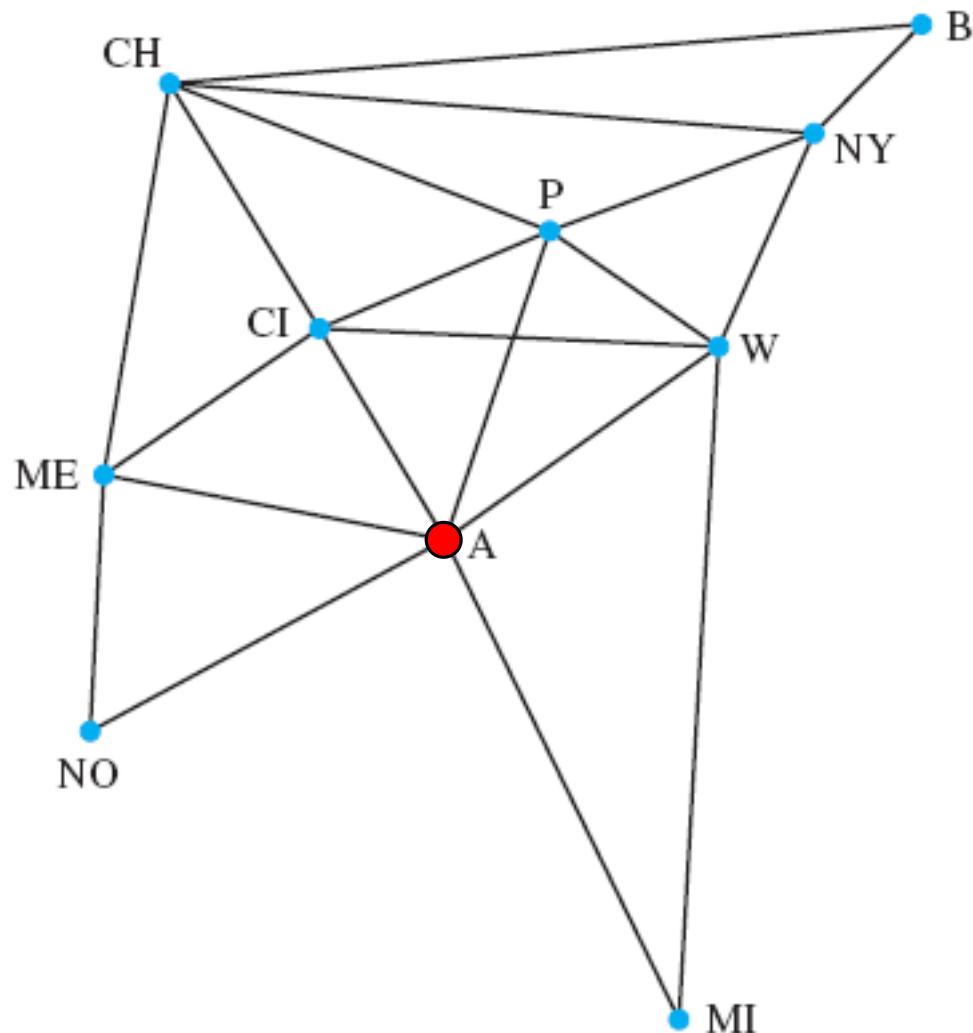


What is the **minimum number** of links to send a message from B to NO?

3: B - CH - ME - NO

Which city/cities has/have the **most** communication links emanating from it/them?

Example

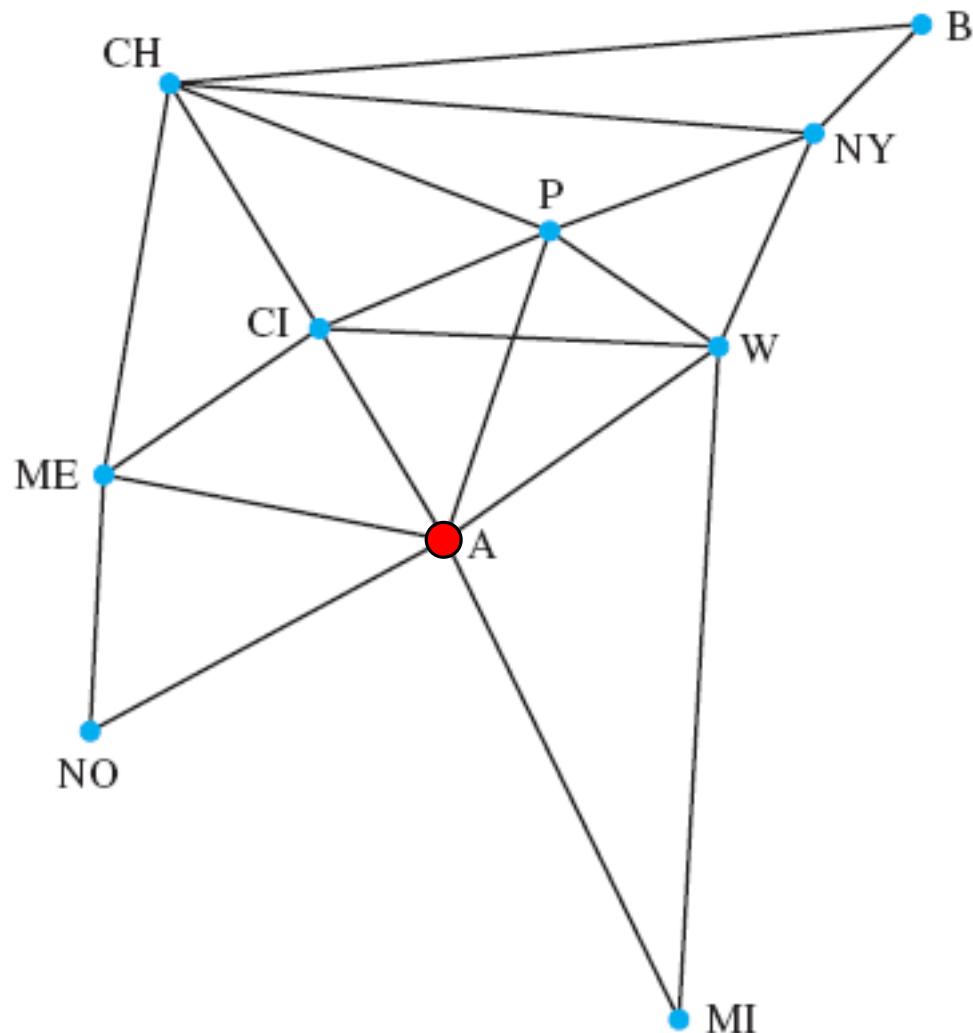


What is the **minimum number** of links to send a message from B to NO?

3: B - CH - ME - NO

Which city/cities has/have the **most** communication links emanating from it/them?

Example



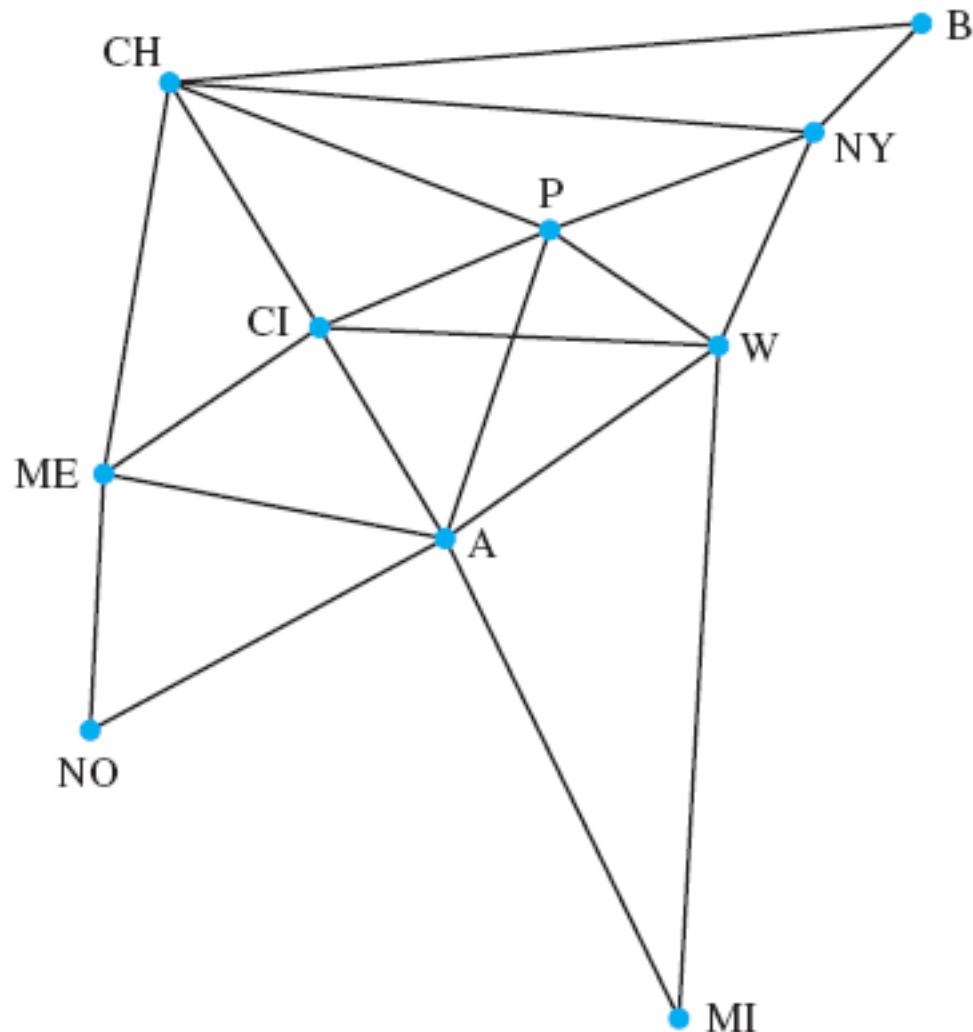
What is the **minimum number** of links to send a message from **B** to **NO**?

3: B - CH - ME - NO

Which city/cities has/have the **most** communication links emanating from it/them?

A: 6 links

Example



What is the **minimum number** of links to send a message from B to NO?

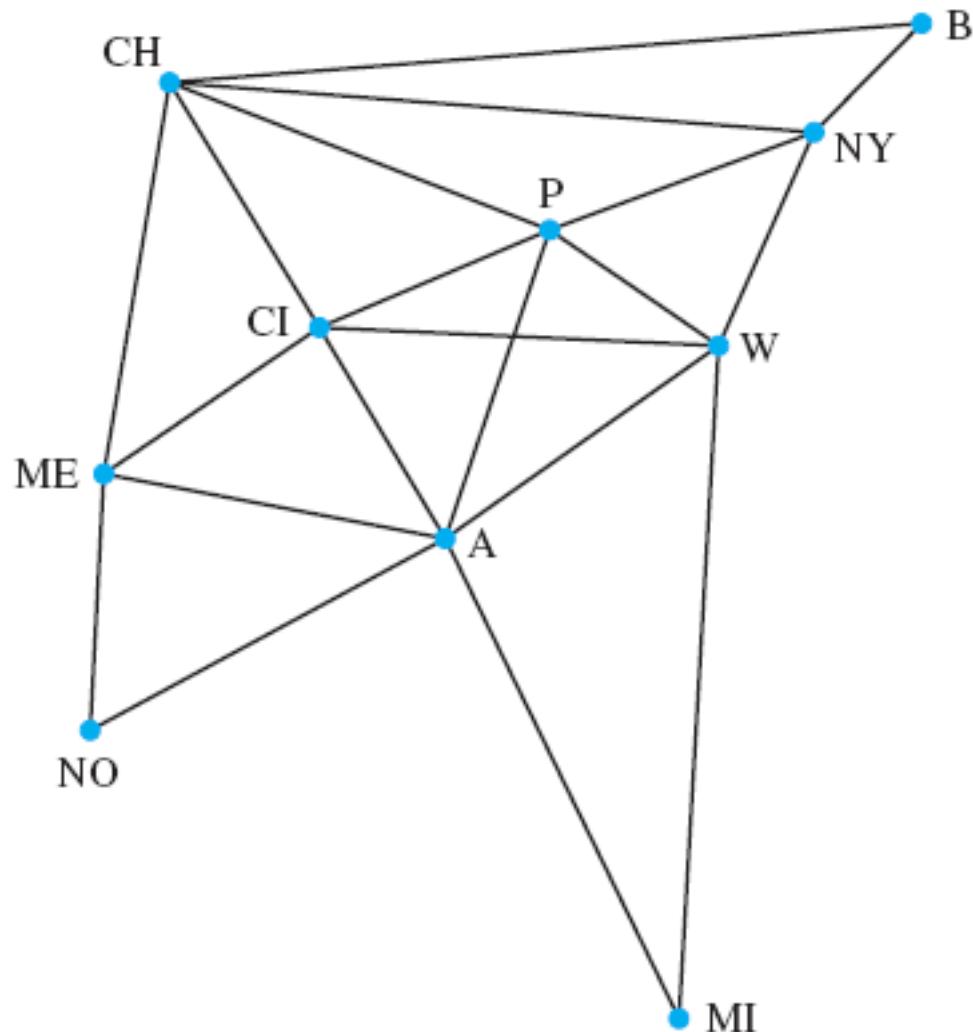
3: B - CH - ME - NO

Which city/cities has/have the **most** communication links emanating from it/them?

A: 6 links

What is the **total** number of communication links?

Example



What is the **minimum number** of links to send a message from B to NO?

3: B - CH - ME - NO

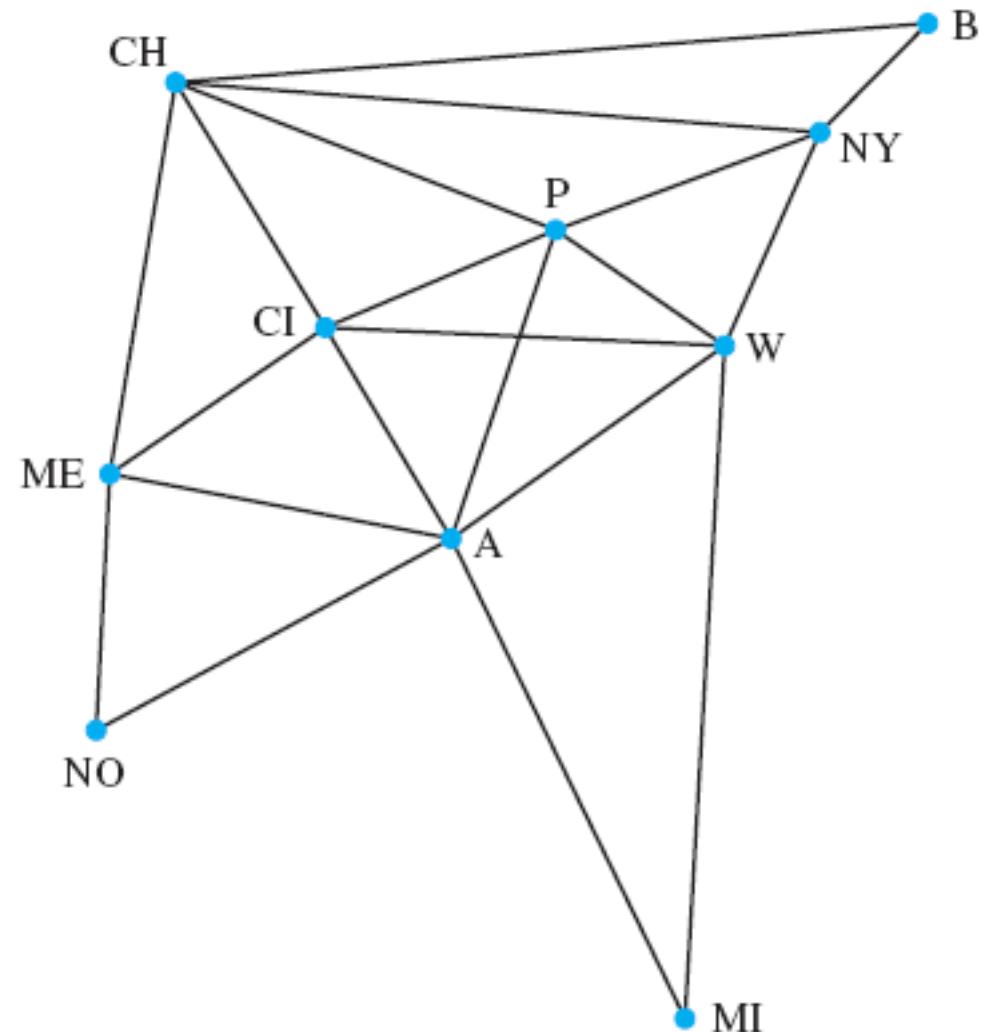
Which city/cities has/have the **most** communication links emanating from it/them?

A: 6 links

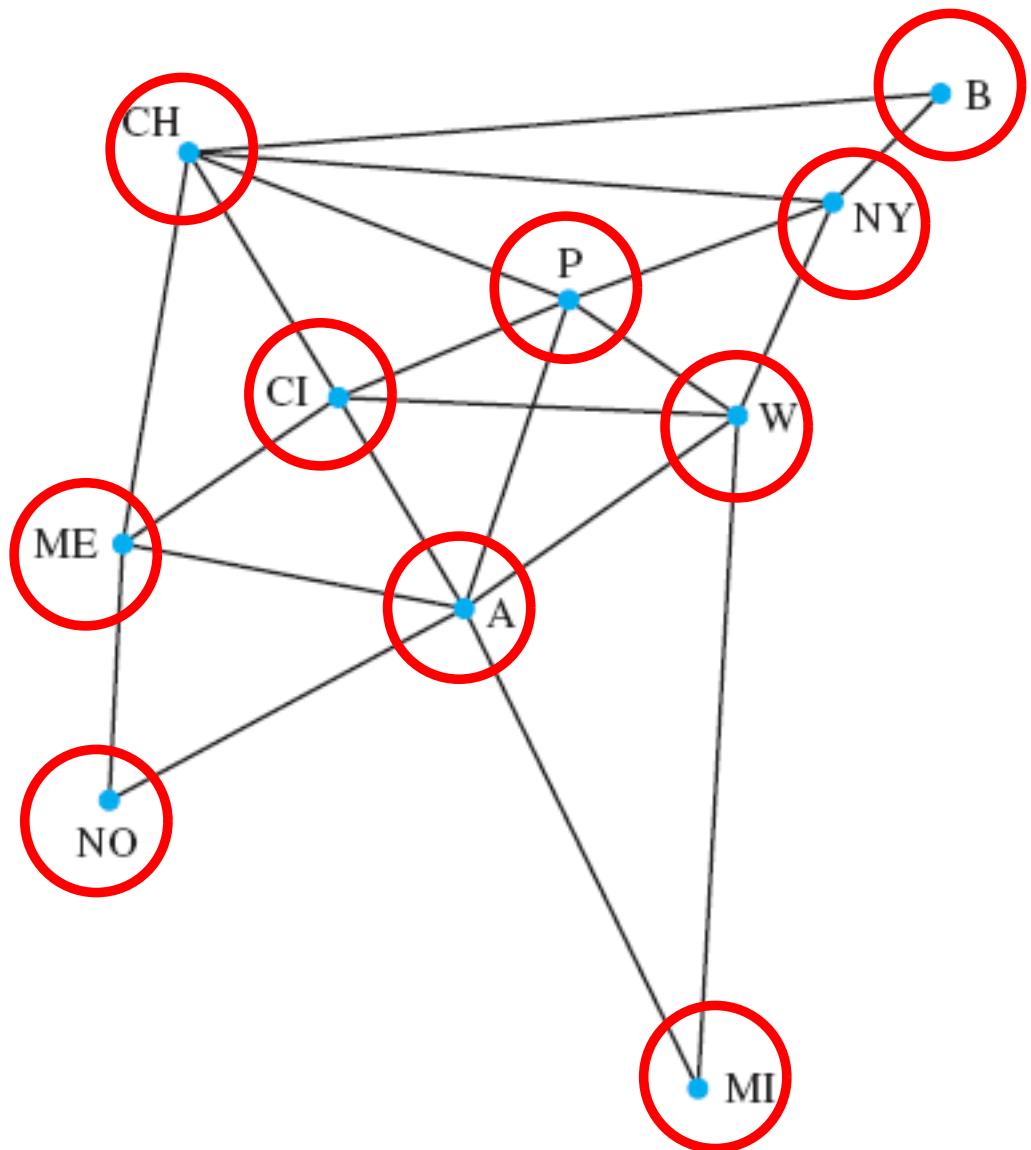
What is the **total** number of communication links?

20 links

Graph G

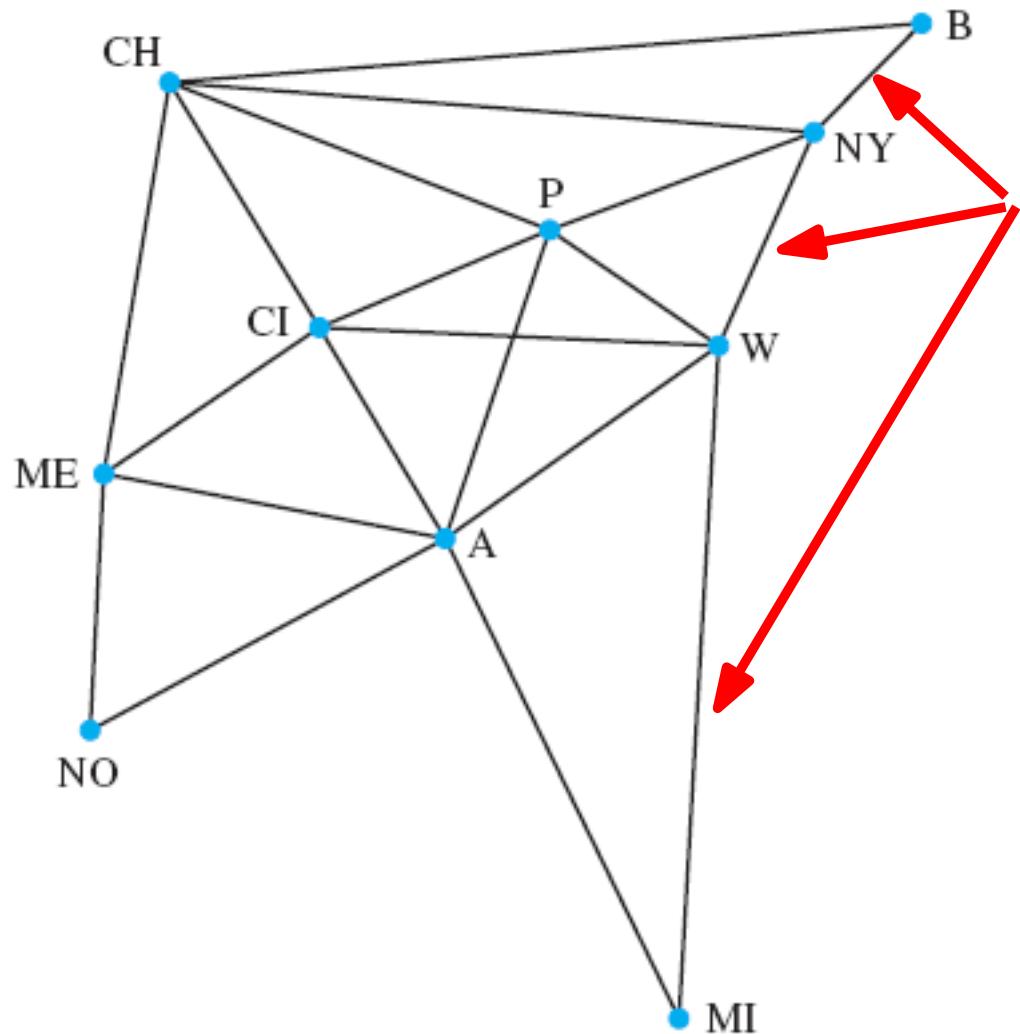


Graph G



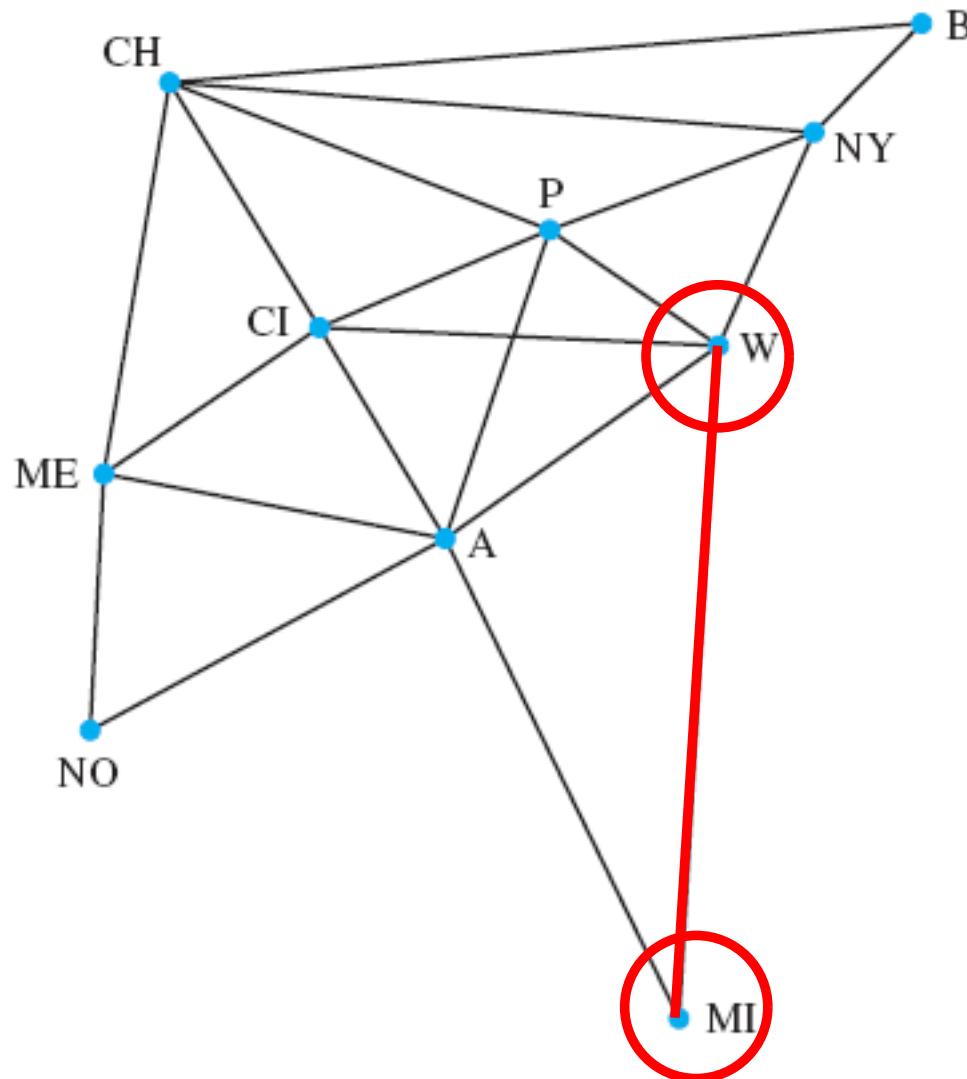
consists of a set of **vertices**
 V , $|V| = n$

Graph G



consists of a set of **vertices**
 V , $|V| = n$
and a set of **edges** E ,
 $|E| = m$

Graph G

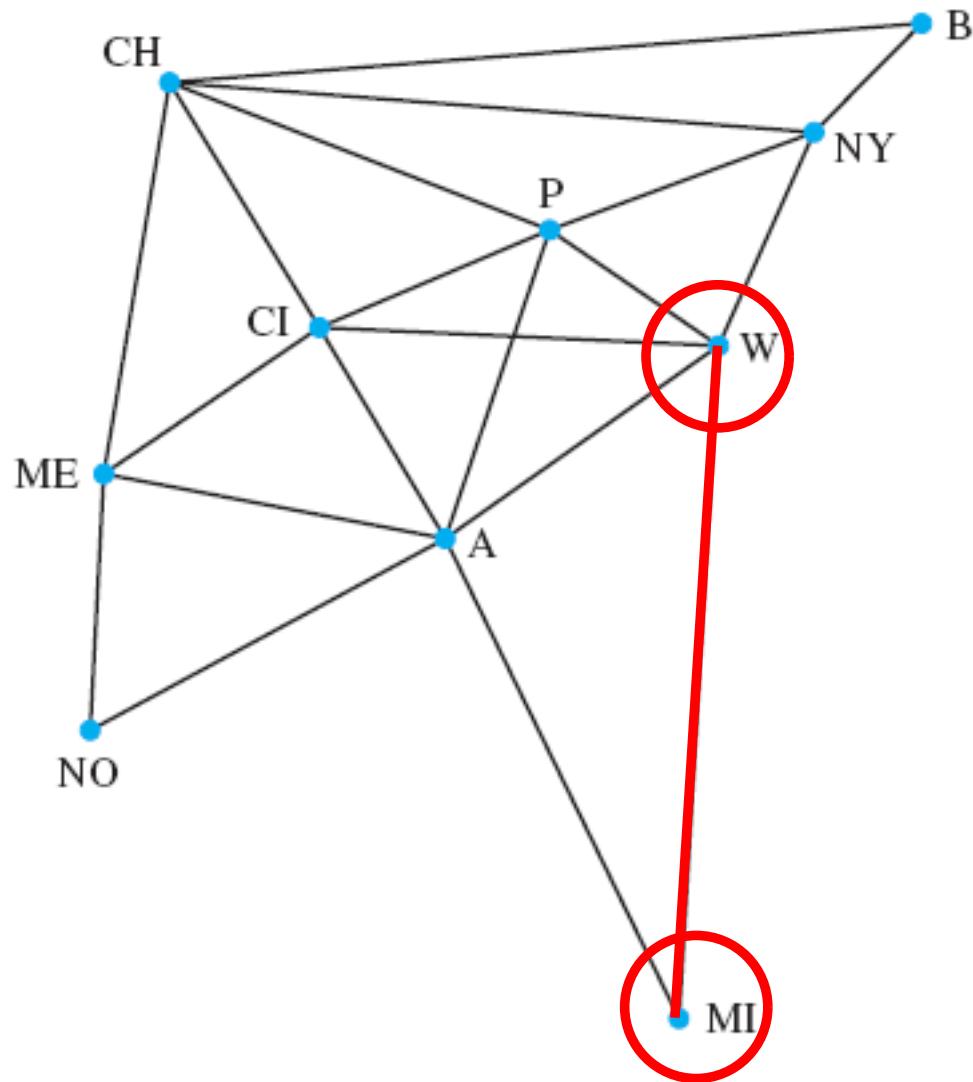


consists of a set of **vertices**
 V , $|V| = n$

and a set of **edges** E ,
 $|E| = m$

Each edge has **two endpoints**

Graph G



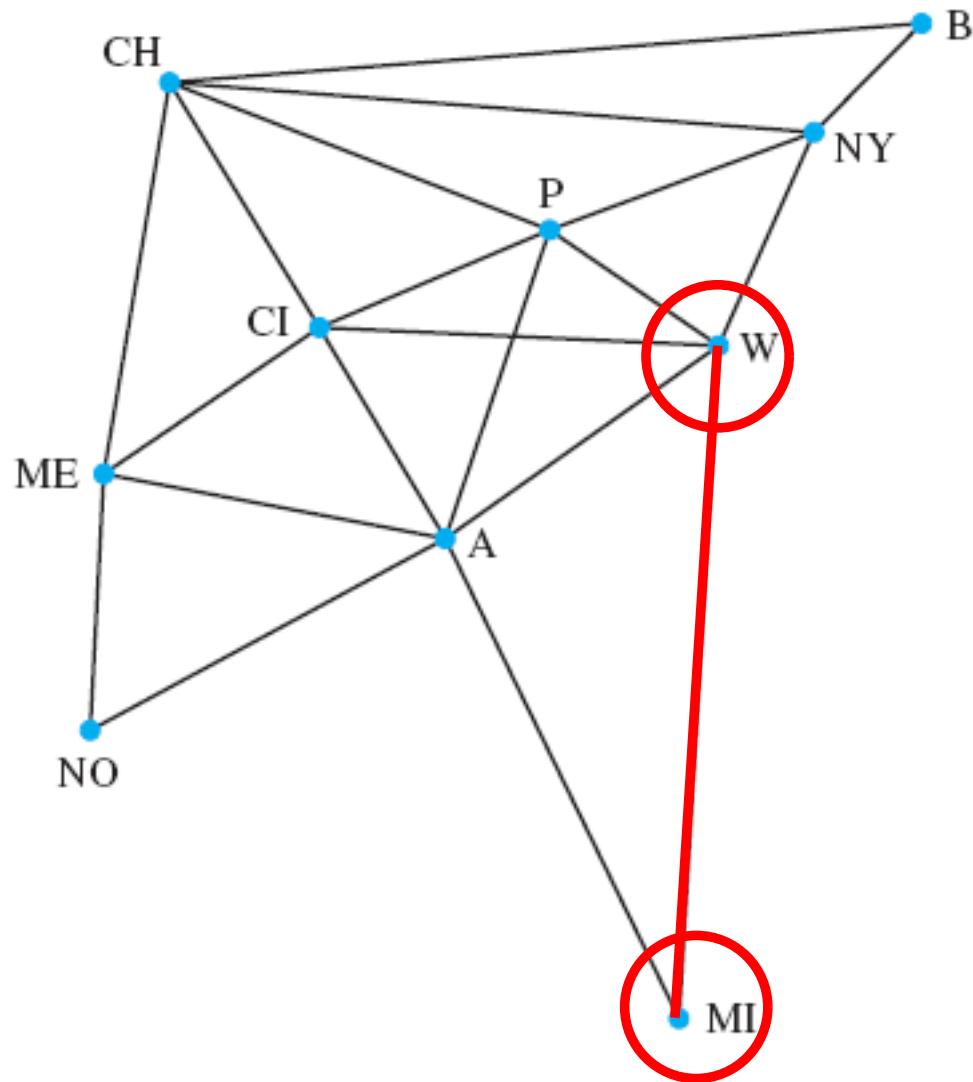
consists of a set of **vertices**
 V , $|V| = n$

and a set of **edges** E ,
 $|E| = m$

Each edge has **two endpoints**

An edge **joins** its endpoints,
two endpoints are **adjacent**
if they are joined by an edge

Graph G



consists of a set of **vertices**
 V , $|V| = n$

and a set of **edges** E ,
 $|E| = m$

Each edge has **two endpoints**

An edge **joins** its endpoints,
two endpoints are **adjacent**
if they are joined by an edge

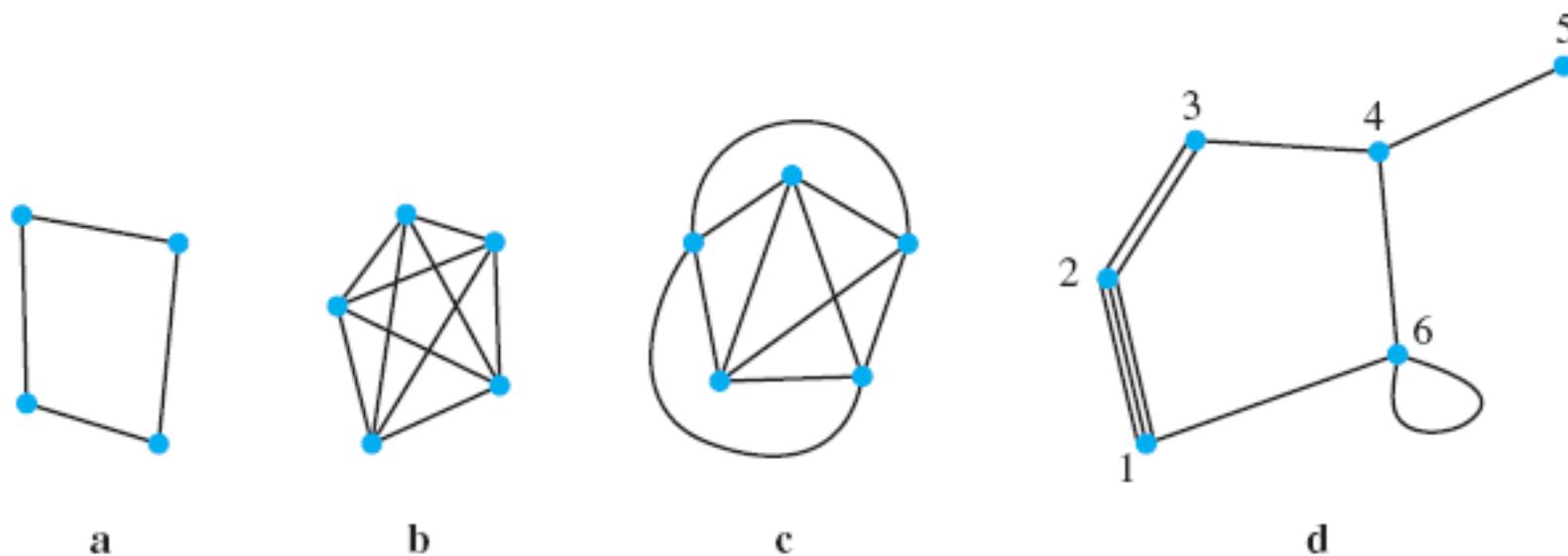
When a vertex is an
endpoint of an edge, we say
that the edge and the vertex
are **incident** to each other

Definition of a Graph

- **Definition.** A $graph G = (V, E)$ consists of a nonempty set V of *vertices* (or *nodes*) and a set E of *edges*. Each edge has either one or two vertices associated with it, called its *endpoints*. An edge is said to be *incident to* (or *connect*) its endpoints.

Definition of a Graph

- **Definition.** A *graph* $G = (V, E)$ consists of a nonempty set V of *vertices* (or *nodes*) and a set E of *edges*. Each edge has either one or two vertices associated with it, called its *endpoints*. An edge is said to be *incident to* (or *connect*) its endpoints.



More Definitions

- *Simple graph* vs. *multigraph pseudograph*

A graph in which **at most one edge** joins each pair of distinct vertices (vs. **multiple** edges) and **no edge** joins a vertex to itself (= **loop**)

More Definitions

- *Simple graph* vs. *multigraph pseudograph*

A graph in which **at most one edge** joins each pair of distinct vertices (vs. **multiple** edges) and **no edge** joins a vertex to itself (= **loop**)

- *Complete graph* K_n

A graph with n vertices that has an edge between **each pair** of vertices

Graphs

- Graphs and graph theory can be used to model:
 - ◊ Computer networks
 - ◊ Social networks
 - ◊ Communication networks
 - ◊ Information networks
 - ◊ Software design
 - ◊ Transportation networks
 - ◊ Biological networks

Graph Models

- Computer Networks

Vertices: computers

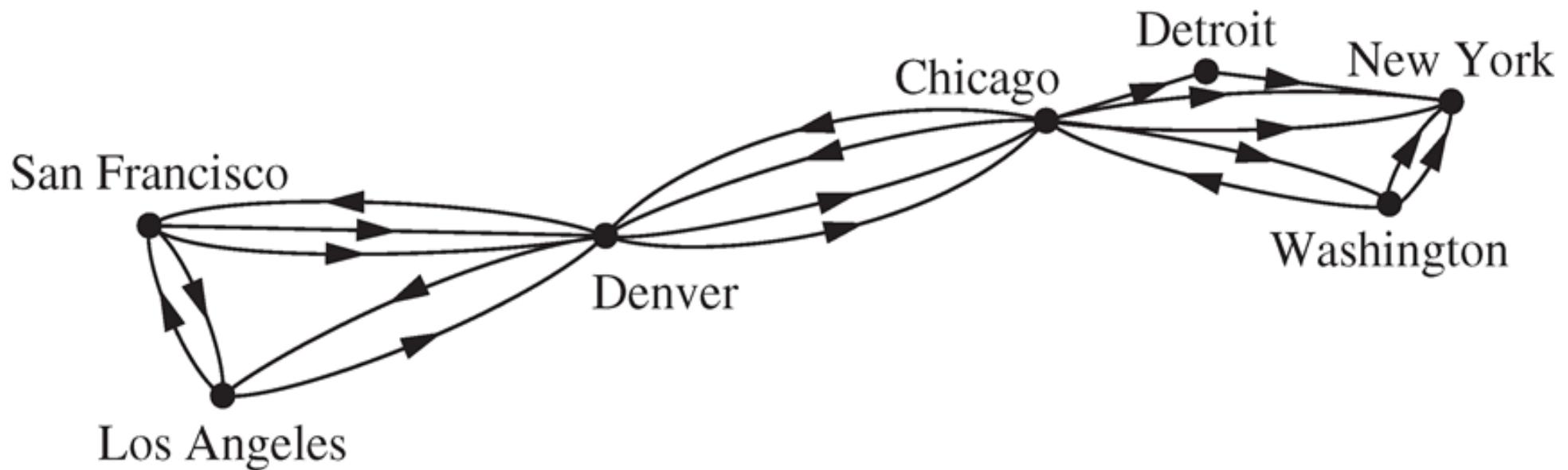
Edges: connections

Graph Models

- Computer Networks

Vertices: computers

Edges: connections



Graph Models

- Social Networks

Vertices: individuals

Edges: relationships

Graph Models

■ Social Networks

Vertices: individuals

Edges: relationships

Friendship graphs: undirected graphs where two people are connected if they are friends (in the real world, wechat, or Facebook, etc.)

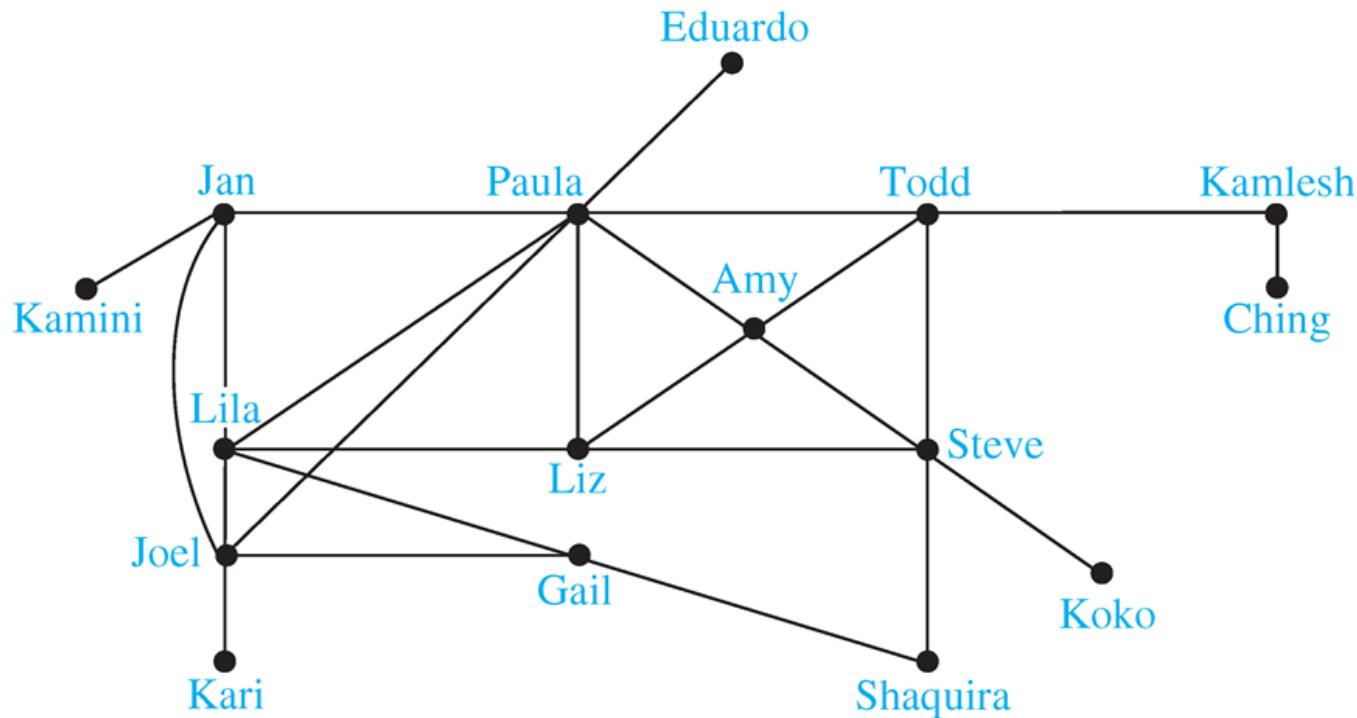
Graph Models

Social Networks

Vertices: individuals

Edges: relationships

Friendship graphs: undirected graphs where two people are connected if they are friends (in the real world, wechat, or Facebook, etc.)



Graph Models

- Influence graphs

directed graphs where there is an edge from one person to another if the first person can influence the second one

Graph Models

- Influence graphs

directed graphs where there is an edge from one person to another if the first person can influence the second one

- Collaboration graphs

undirected graphs where two people are connected if they collaborate in some way

Graph Models

- Influence graphs

directed graphs where there is an edge from one person to another if the first person can influence the second one

Collaboration graphs

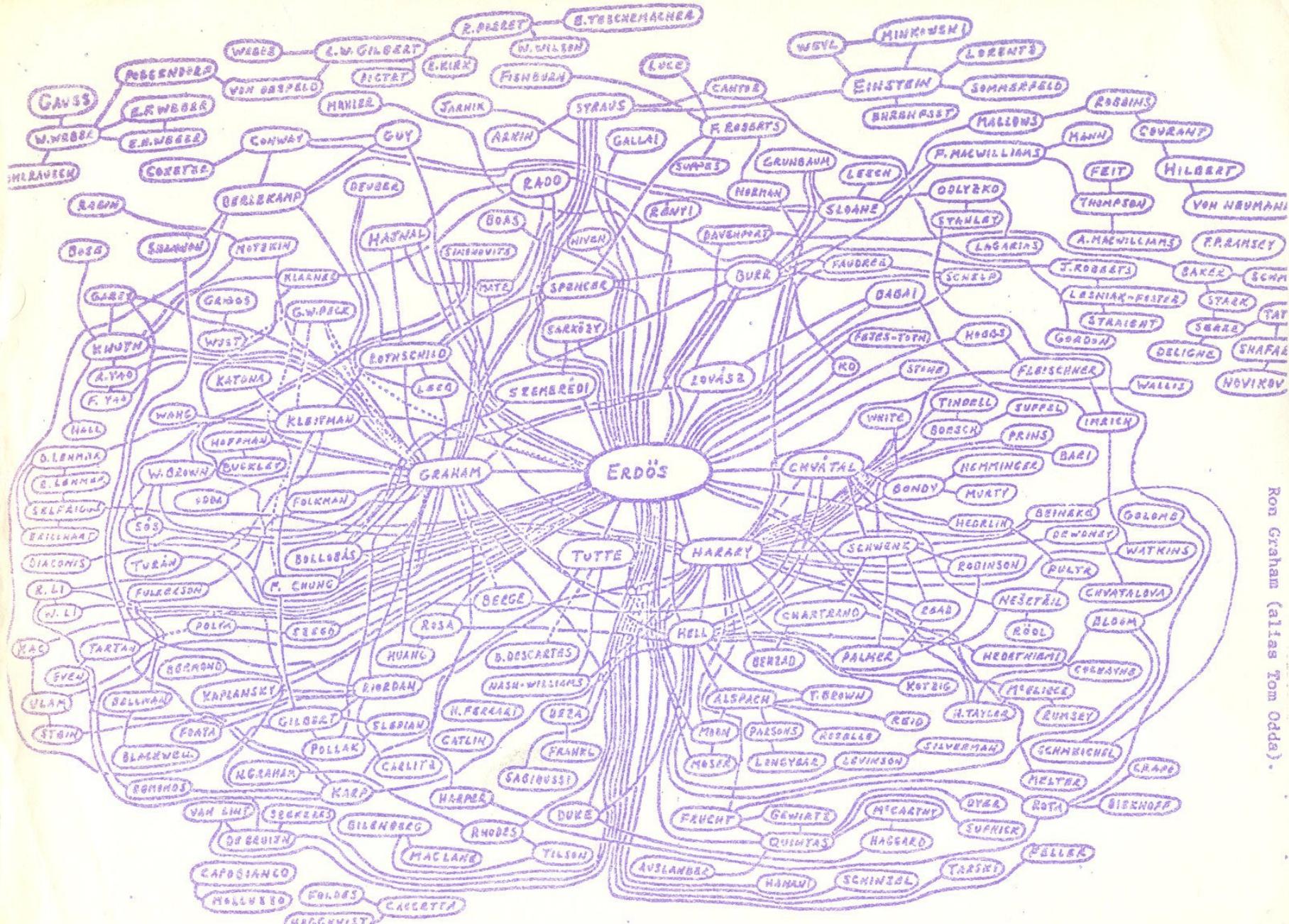
undirected graphs where two people are connected if they collaborate in some way

Example

the Hollywood graph

the Erdős number

The Erdős Number

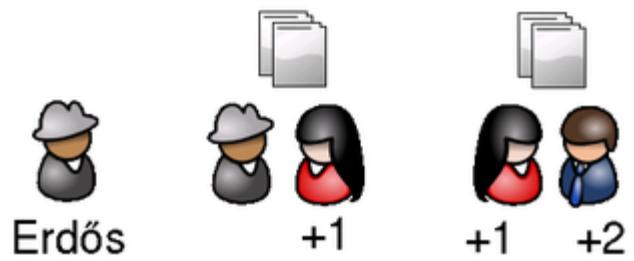


Ron Graham (alias Tom Odda).

Figure 1

To appear in Topics in Graph Theory (F. Harary, ed.), New York Academy of Sciences (1979).

The Erdős Number



The Erdős Number



Erdős number 0	---	1 person
Erdős number 1	---	504 people
Erdős number 2	---	6593 people
Erdős number 3	---	33605 people
Erdős number 4	---	83642 people
Erdős number 5	---	87760 people
Erdős number 6	---	40014 people
Erdős number 7	---	11591 people
Erdős number 8	---	3146 people
Erdős number 9	---	819 people
Erdős number 10	---	244 people
Erdős number 11	---	68 people
Erdős number 12	---	23 people
Erdős number 13	---	5 people

The Erdős Number



Erdős number	0	---	1 person
Erdős number	1	---	504 people
Erdős number	2	---	6593 people
Erdős number	3	---	33605 people
Erdős number	4	---	83642 people
Erdős number	5	---	87760 people
Erdős number	6	---	40014 people
Erdős number	7	---	11591 people
Erdős number	8	---	3146 people
Erdős number	9	---	819 people
Erdős number	10	---	244 people
Erdős number	11	---	68 people
Erdős number	12	---	23 people
Erdős number	13	---	5 people

Statistics on Mathematical Collaboration, 1903-2016

◆	#Laureates ◆	#Erdős ◆	%Erdős ◆	Min ◆	Max ◆	Average ◆	Median ◆
Fields Medal	56	56	100.0%	2	6	3.36	3
Nobel Economics	76	47	61.84%	2	8	4.11	4
Nobel Chemistry	172	42	24.42%	3	10	5.48	5
Nobel Medicine	210	58	27.62%	3	12	5.50	5
Nobel Physics	200	159	79.50%	2	12	5.63	5

Undirected Graphs

- **Definition** Two vertices u, v in an **undirected** graph G are called *adjacent* (or *neighbors*) in G if there is an edge e between u and v . Such an edge e is called *incident* with the vertices u and v and e is said to connect u and v .

Undirected Graphs

- **Definition** Two vertices u, v in an **undirected** graph G are called *adjacent* (or *neighbors*) in G if there is an edge e between u and v . Such an edge e is called *incident* with the vertices u and v and e is said to connect u and v .

Definition The set of all neighbors of a vertex v of $G = (V, E)$, denoted by $N(v)$, is called *the neighborhood of v* . If A is a subset of V , we denote by $N(A)$ the set of all vertices in G that are adjacent to at least one vertex in A .



Undirected Graphs

- **Definition** Two vertices u, v in an **undirected** graph G are called *adjacent* (or *neighbors*) in G if there is an edge e between u and v . Such an edge e is called *incident* with the vertices u and v and e is said to connect u and v .

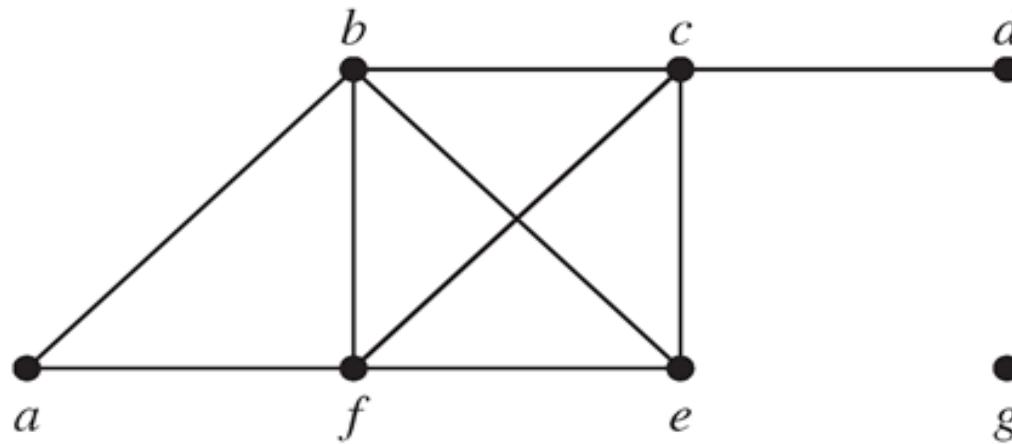
Definition The set of all neighbors of a vertex v of $G = (V, E)$, denoted by $N(v)$, is called *the neighborhood of v* . If A is a subset of V , we denote by $N(A)$ the set of all vertices in G that are adjacent to at least one vertex in A .

Definition The *degree of a vertex in an undirected graph* is the number of edges incident with it, except that a loop at a vertex contributes two to the degree of that vertex. The degree of the vertex v is denoted by $\deg(v)$.



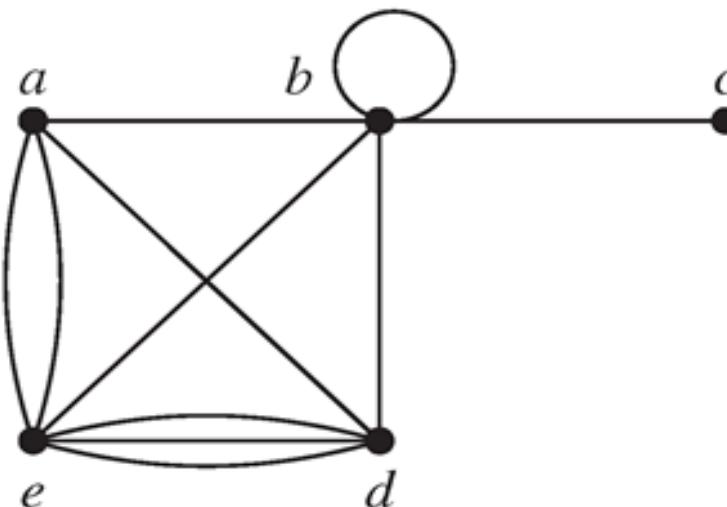
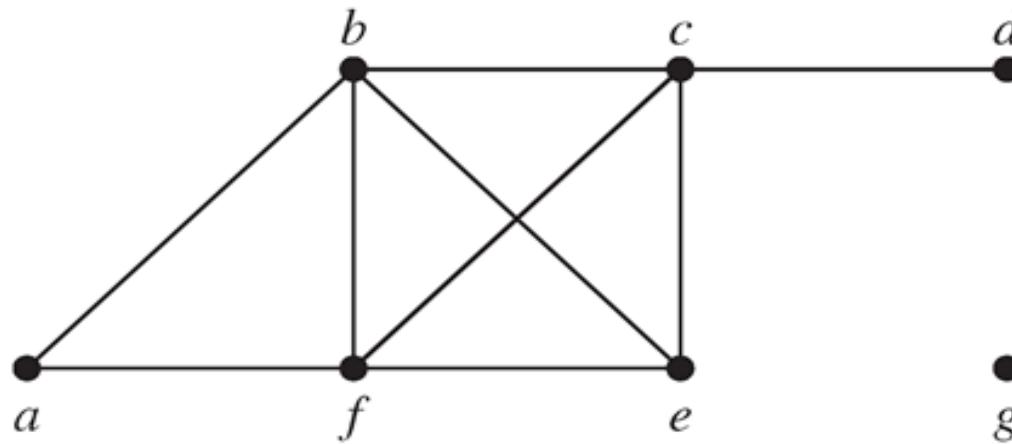
Undirected Graphs

- **Example:** What are the degrees and neighborhoods of the vertices in the graph G ?



Undirected Graphs

- **Example:** What are the degrees and neighborhoods of the vertices in the graph G ?



Undirected Graphs

- **Theorem 1 (Handshaking Theorem)** If $G = (V, E)$ is an undirected graph with m edges, then

$$2m = \sum_{v \in V} \deg(v)$$

Proof

Undirected Graphs

- **Theorem 2** An undirected graph has an even number of vertices of odd degree.

Undirected Graphs

- **Theorem 2** An undirected graph has an even number of vertices of odd degree.

Proof Let V_1 be the vertices of even degrees and V_2 be the vertices of odd degree.

Undirected Graphs

- **Theorem 2** An undirected graph has an even number of vertices of odd degree.

Proof Let V_1 be the vertices of even degrees and V_2 be the vertices of odd degree.

$$2m = \sum_{v \in V} \deg(v) = \sum_{v \in V_1} \deg(v) + \sum_{v \in V_2} \deg(v)$$

Undirected Graphs

- **Theorem 2** An undirected graph has an even number of vertices of odd degree.

Proof Let V_1 be the vertices of even degrees and V_2 be the vertices of odd degree.

$$2m = \sum_{v \in V} \deg(v) = \boxed{\sum_{v \in V_1} \deg(v)} + \boxed{\sum_{v \in V_2} \deg(v)}$$

Directed Graphs

- **Definition** An *directed graph* $G = (V, E)$ consists of V , a nonempty set of vertices, and E , a set of directed edges. Each edge is an **ordered** pair of vertices. The directed edge (u, v) is said to **start at u and end at v** .



Directed Graphs

- **Definition** An *directed graph* $G = (V, E)$ consists of V , a nonempty set of vertices, and E , a set of directed edges. Each edge is an **ordered** pair of vertices. The directed edge (u, v) is said to **start at u and end at v** .

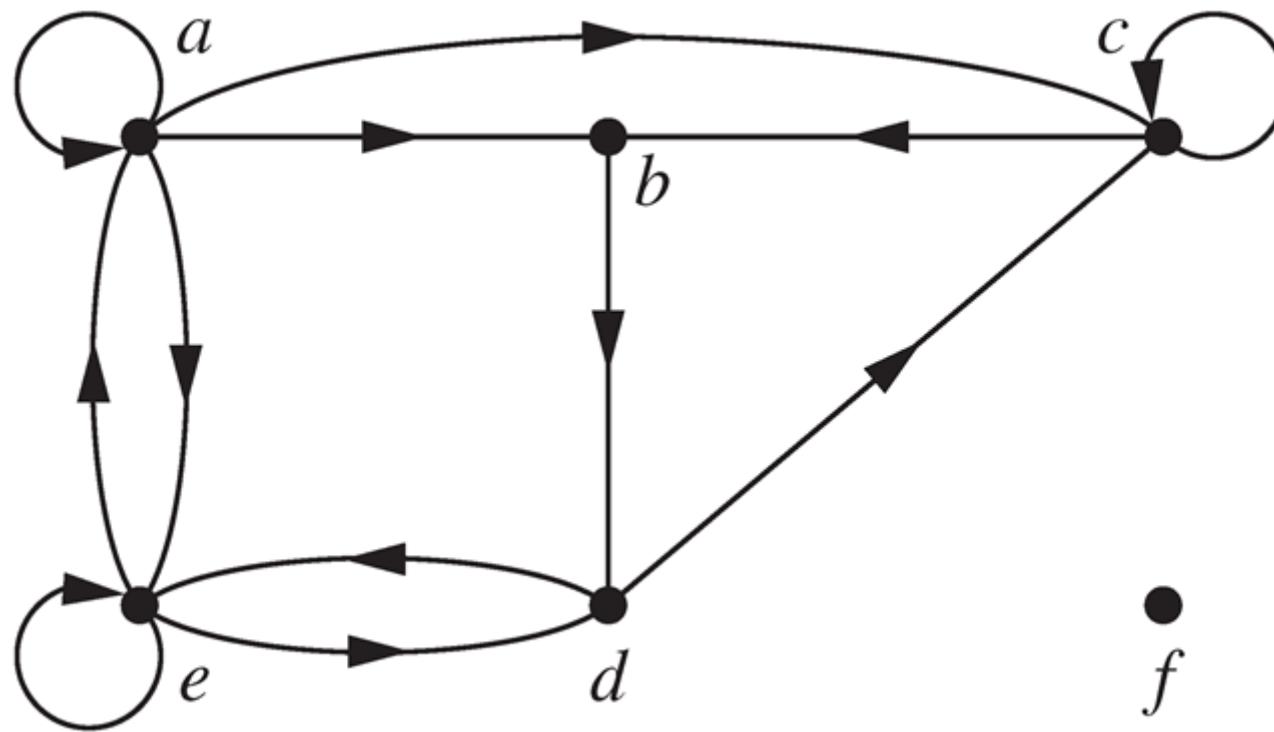
Definition Let (u, v) be an edge in G . Then u is the *initial vertex* of the edge and is *adjacent to v* and v is the *terminal vertex* of this edge and is *adjacent from u* . The initial and terminal vertices of a loop are the same.

Directed Graphs

- **Definition** The *in-degree* of a vertex v , denoted by $\deg^-(v)$, is the number of edges which terminate at v . The *out-degree* of v , denoted by $\deg^+(v)$, is the number of edges with v as their initial vertex. Note that a **loop** at a vertex contributes 1 to both the in-degree and the out-degree of the vertex.

Directed Graphs

- **Definition** The *in-degree* of a vertex v , denoted by $\deg^-(v)$, is the number of edges which terminate at v . The *out-degree* of v , denoted by $\deg^+(v)$, is the number of edges with v as their initial vertex. Note that a **loop** at a vertex contributes 1 to both the in-degree and the out-degree of the vertex.



Directed Graphs

- **Theorem 3** Let $G = (V, E)$ be a graph with directed edges. Then

$$|E| = \sum_{v \in V} \deg^-(v) = \sum_{v \in V} \deg^+(v)$$

Proof



Complete Graphs

- A *complete graph* on n vertices, denoted by K_n , is the simple graph that contains exactly one edge between **each pair** of distinct vertices.

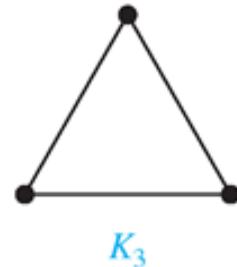
Complete Graphs

- A *complete graph* on n vertices, denoted by K_n , is the simple graph that contains exactly one edge between **each pair** of distinct vertices.

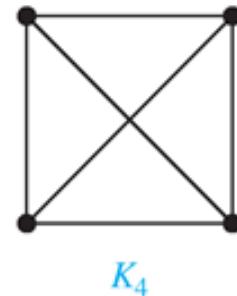
K_1



K_2



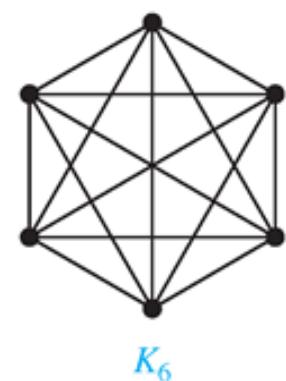
K_3



K_4



K_5



K_6

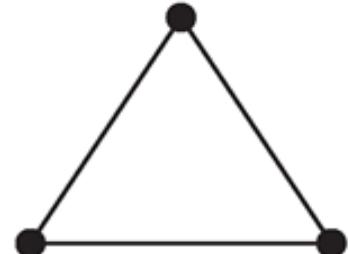


Cycles

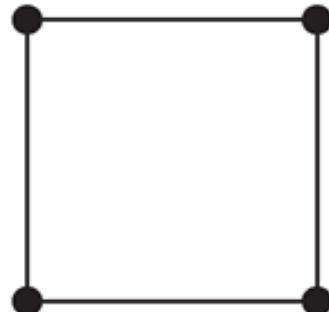
- A *cycle* C_n for $n \geq 3$ consists of n vertices v_1, v_2, \dots, v_n , and edges $\{v_1, v_2\}, \{v_2, v_3\}, \dots, \{v_{n-1}, v_n\}, \{v_n, v_1\}$.

Cycles

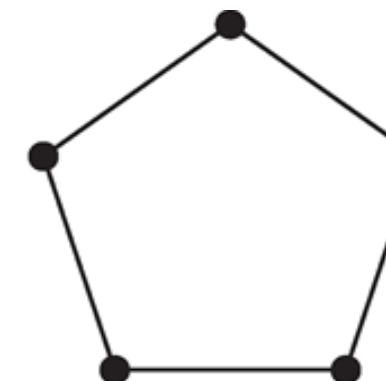
- A *cycle* C_n for $n \geq 3$ consists of n vertices v_1, v_2, \dots, v_n , and edges $\{v_1, v_2\}, \{v_2, v_3\}, \dots, \{v_{n-1}, v_n\}, \{v_n, v_1\}$.



C_3



C_4



C_5



C_6

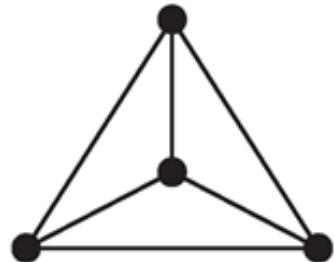
Wheels

- A *wheel* W_n is obtained by adding an additional vertex to a cycle C_n .

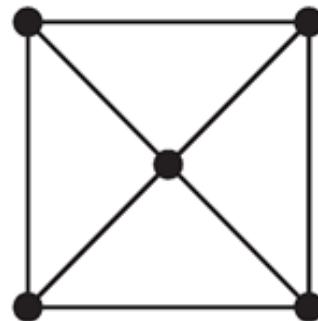


Wheels

- A *wheel* W_n is obtained by adding an additional vertex to a cycle C_n .



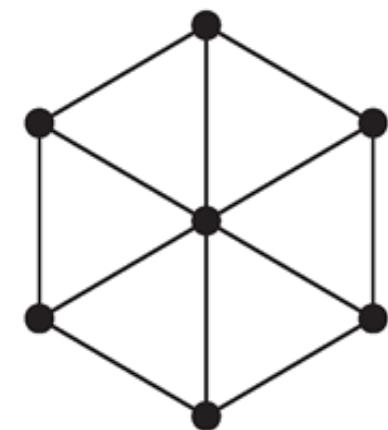
W_3



W_4



W_5



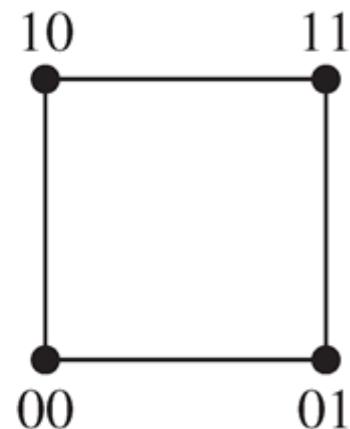
W_6

N -dimensional Hypercube

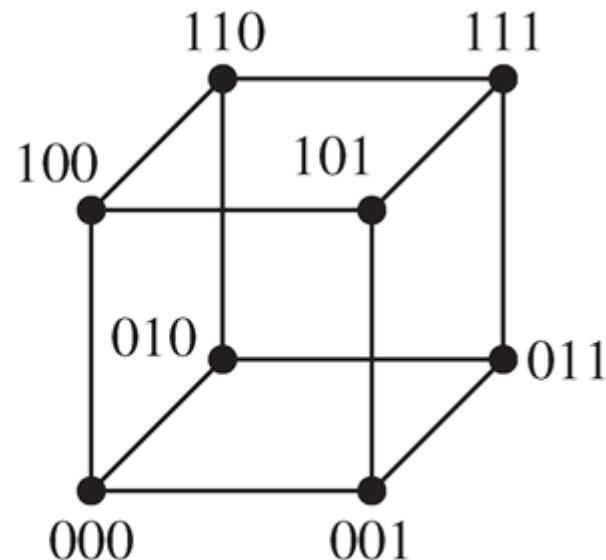
- An *n-dimensional hypercube*, or *n-cube*, Q_n is a graph with 2^n vertices representing all bit strings of length n , where there is an edge between two vertices that differ in exactly one bit position.

N -dimensional Hypercube

- An *n -dimensional hypercube*, or *n -cube*, Q_n is a graph with 2^n vertices representing all bit strings of length n , where there is an edge between two vertices that differ in exactly one bit position.



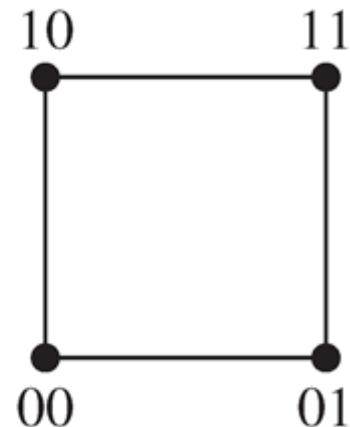
Q_1



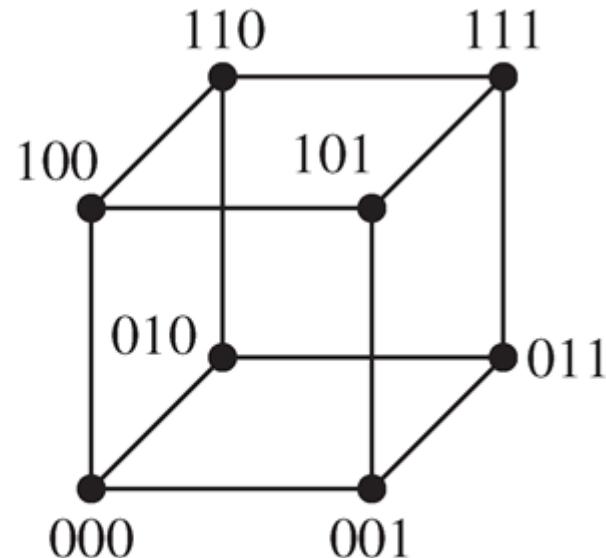
Q_3

N -dimensional Hypercube

- An *n -dimensional hypercube*, or *n -cube*, Q_n is a graph with 2^n vertices representing all bit strings of length n , where there is an edge between two vertices that differ in exactly one bit position.



Q_1



Q_3

How many vertices? How many edges?

Bipartite Graphs

- **Definition** A simple graph G is *bipartite* if V can be partitioned into two disjoint subsets V_1 and V_2 such that every edge connects a vertex in V_1 and a vertex in V_2 .

Bipartite Graphs

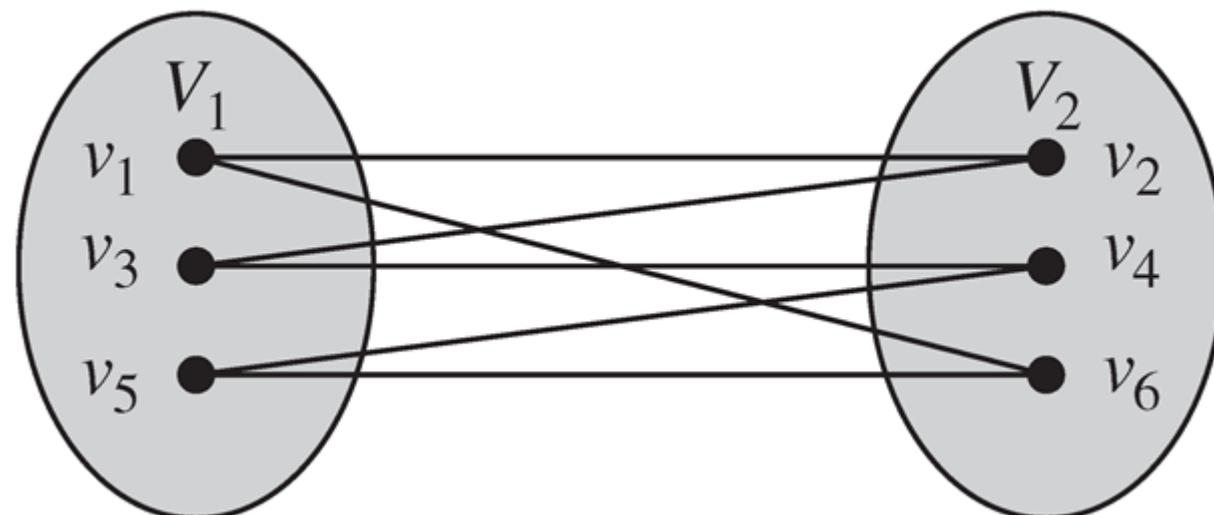
- **Definition** A simple graph G is *bipartite* if V can be partitioned into two disjoint subsets V_1 and V_2 such that every edge connects a vertex in V_1 and a vertex in V_2 .

An equivalent definition of a *bipartite graph* is a graph where it is possible to color the vertices **red** or **blue** so that no two adjacent vertices are of the same color.

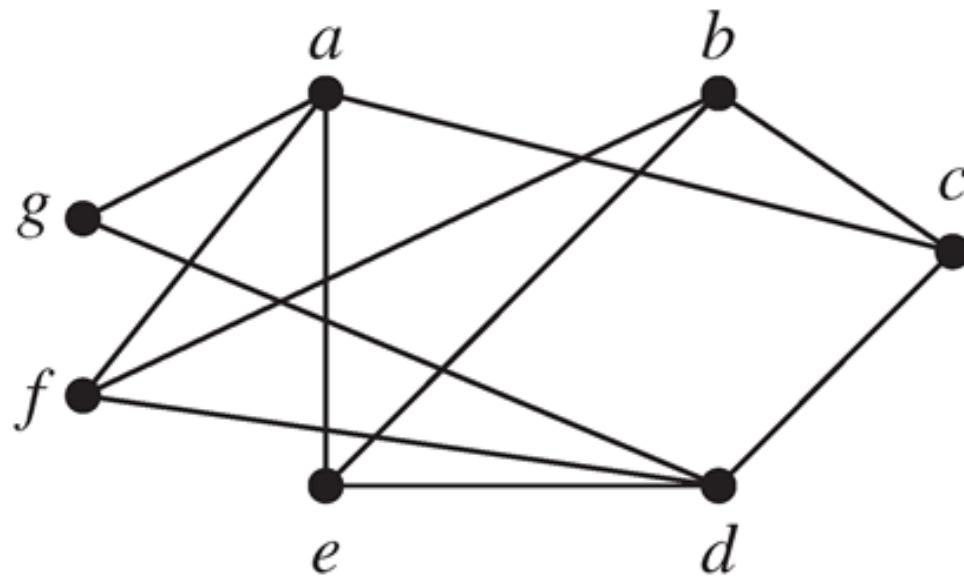
Bipartite Graphs

- **Definition** A simple graph G is *bipartite* if V can be partitioned into two disjoint subsets V_1 and V_2 such that every edge connects a vertex in V_1 and a vertex in V_2 .

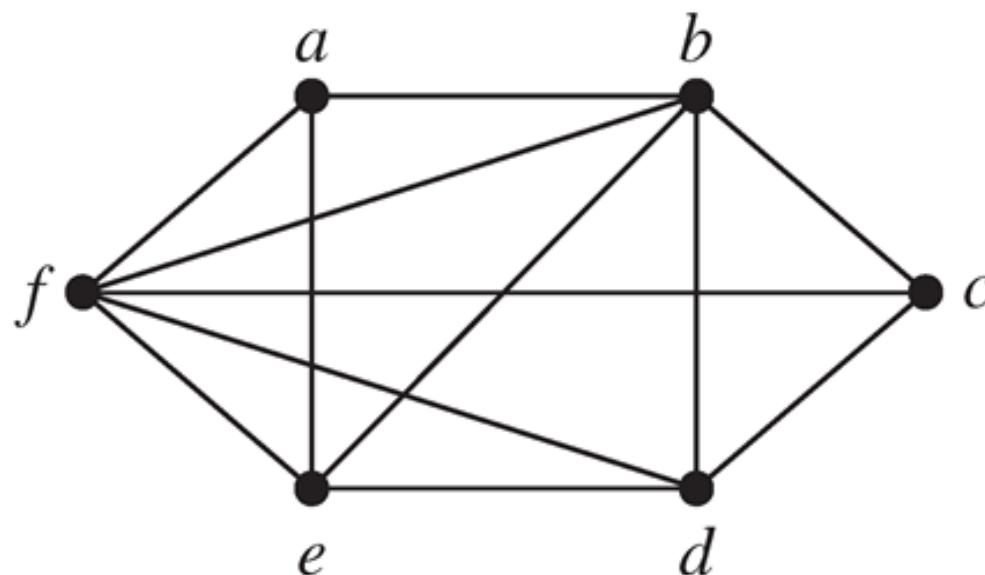
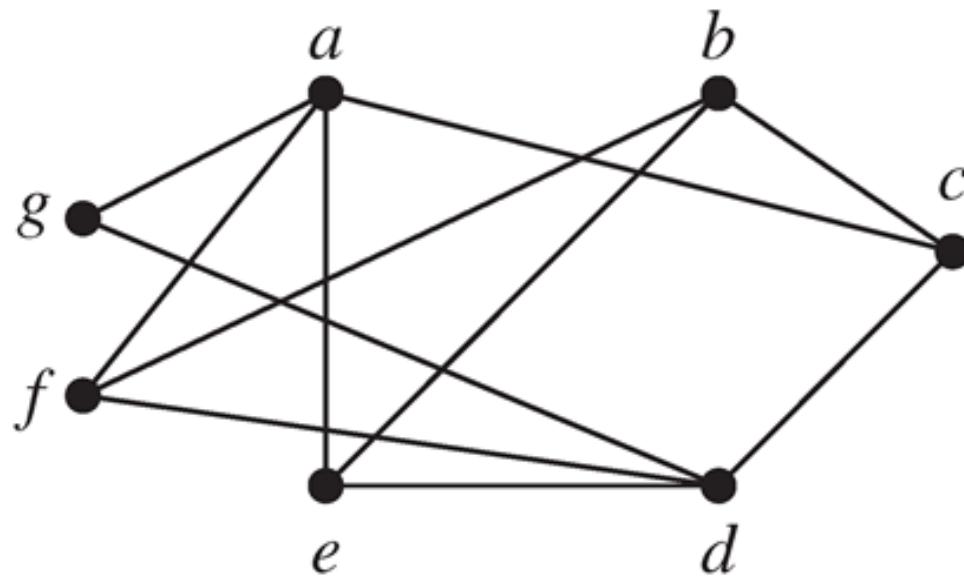
An equivalent definition of a *bipartite graph* is a graph where it is possible to color the vertices red or blue so that no two adjacent vertices are of the same color.



Bipartite Graphs

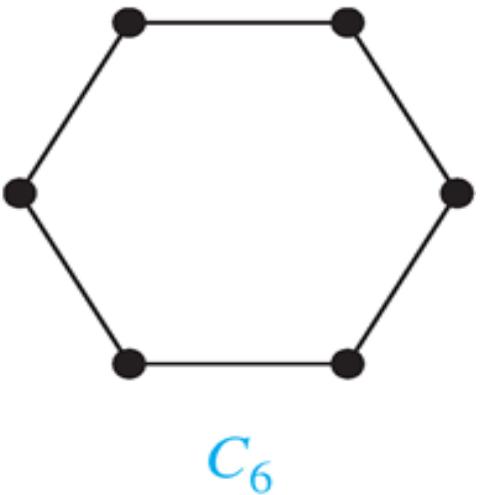


Bipartite Graphs



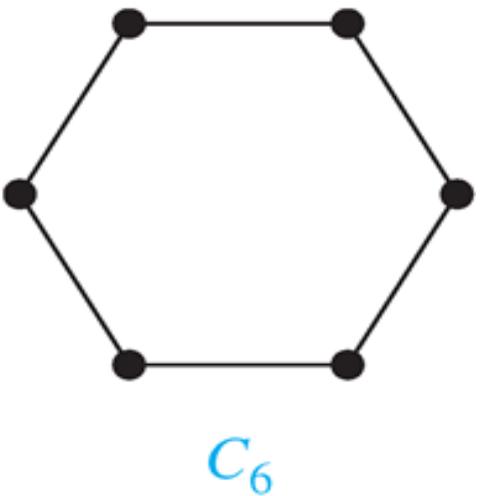
Bipartite Graphs

- **Example** Show that C_6 is bipartite.

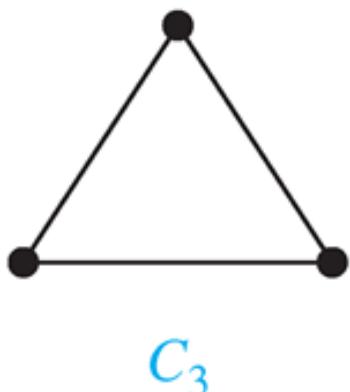


Bipartite Graphs

- **Example** Show that C_6 is bipartite.



- **Example** Show that C_3 is not bipartite.



Complete Bipartite Graphs

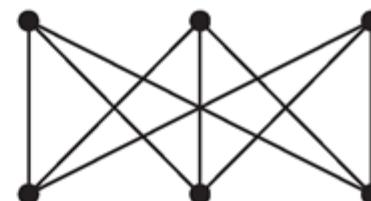
- **Definition** A *complete bipartite graph* $K_{m,n}$ is a graph that has its vertex set partitioned into two subsets V_1 of size m and V_2 of size n such that there is an edge from every vertex in V_1 to every vertex in V_2 .

Complete Bipartite Graphs

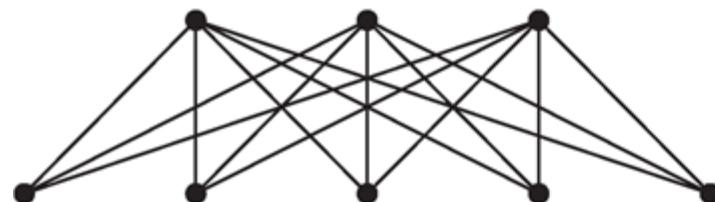
- **Definition** A *complete bipartite graph* $K_{m,n}$ is a graph that has its vertex set partitioned into two subsets V_1 of size m and V_2 of size n such that there is an edge from every vertex in V_1 to every vertex in V_2 .



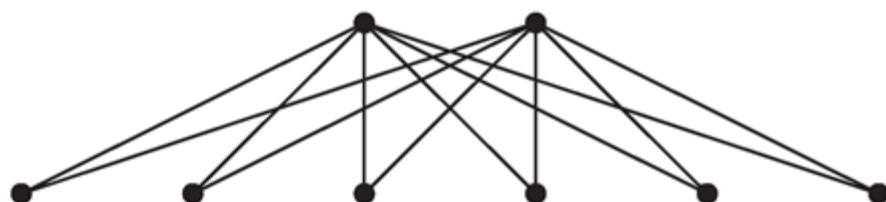
$K_{2,3}$



$K_{3,3}$



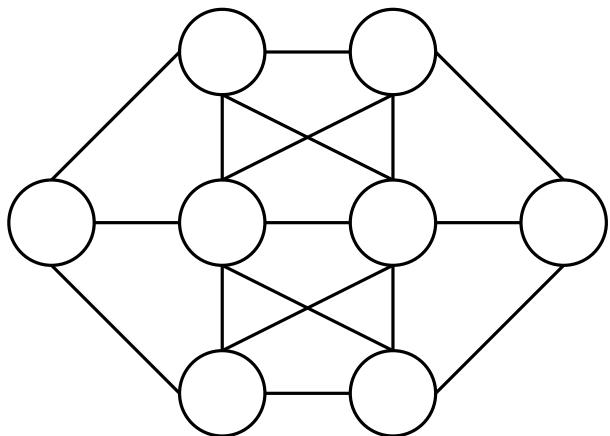
$K_{3,5}$



$K_{2,6}$

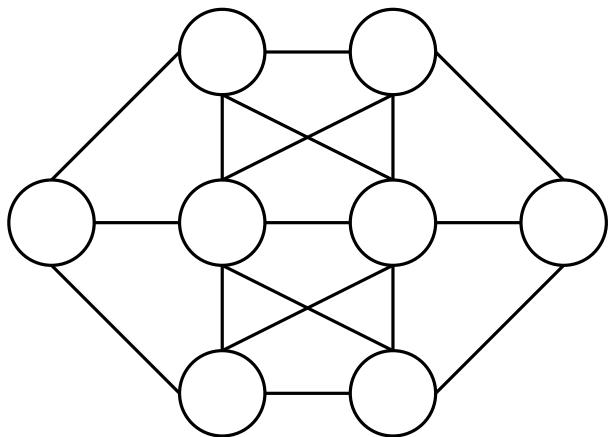
Puzzles using Graphs

- **The eight-circles problem** Place the letters A, B, C, D, E, F, G, H into the eight circles in the figure, in such a way that **no** letter is adjacent to a letter that is next to it in the alphabet.



Puzzles using Graphs

- **The eight-circles problem** Place the letters A, B, C, D, E, F, G, H into the eight circles in the figure, in such a way that **no** letter is adjacent to a letter that is next to it in the alphabet.



- **Six people at a party** Show that, in any gathering of six people, there are either three people who all know each other, or three people none of which knows either of the other two.

Bipartite Graphs and Matchings

- *Matching* the elements of one set to elements in another. A *matching* is a subset of E s.t. no two edges are incident with the same vertex.



Bipartite Graphs and Matchings

- *Matching* the elements of one set to elements in another. A *matching* is a subset of E s.t. no two edges are incident with the same vertex.

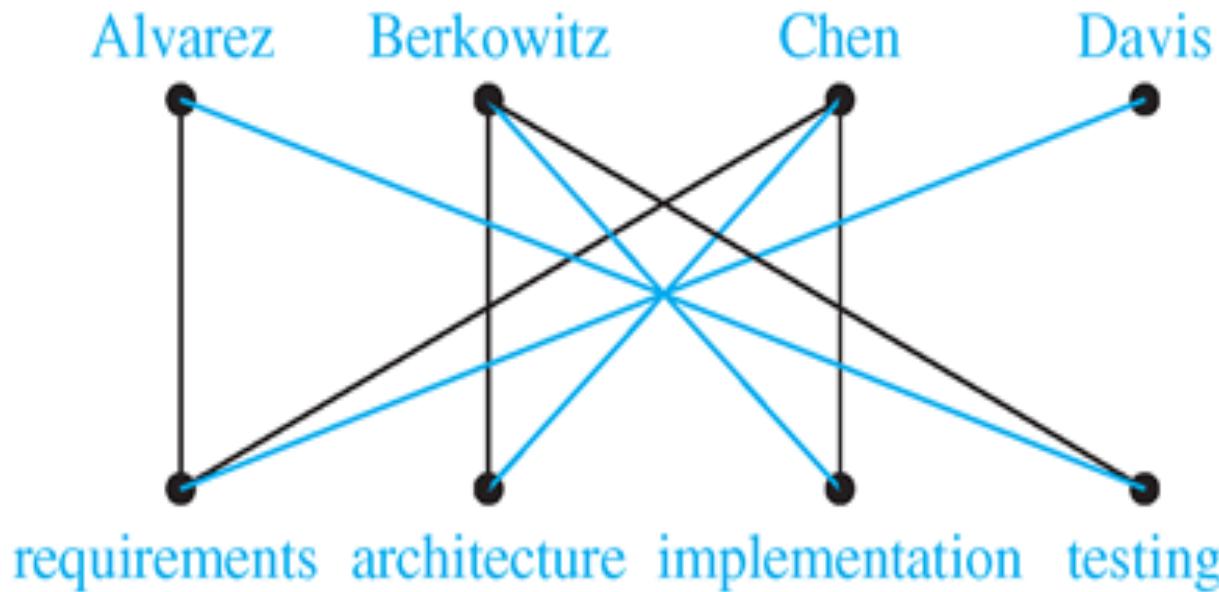
Job assignments: vertices represent the jobs and the employees, edges link employees with those jobs they have been trained to do. A **common goal** is to match jobs to employees so that the most jobs are done.



Bipartite Graphs and Matchings

- *Matching* the elements of one set to elements in another. A *matching* is a subset of E s.t. no two edges are incident with the same vertex.

Job assignments: vertices represent the jobs and the employees, edges link employees with those jobs they have been trained to do. A **common goal** is to match jobs to employees so that the most jobs are done.



Bipartite Graphs and Matchings

- **Definition** A simple graph G is *bipartite* if V can be partitioned into two disjoint subsets V_1 and V_2 such that every edge connects a vertex in V_1 and a vertex in V_2 .
An equivalent definition of a *bipartite graph* is a graph where it is possible to color the vertices **red** or **blue** so that no two adjacent vertices are of the same color.

Bipartite Graphs and Matchings

- **Definition** A simple graph G is *bipartite* if V can be partitioned into two disjoint subsets V_1 and V_2 such that every edge connects a vertex in V_1 and a vertex in V_2 .

An equivalent definition of a *bipartite graph* is a graph where it is possible to color the vertices **red** or **blue** so that no two adjacent vertices are of the same color.

Matching the elements of one set to elements in another. A *matching* is a subset of E s.t. no two edges are incident with the same vertex.

Bipartite Graphs and Matchings

- A *maximum matching* is a matching with the largest number of edges.

Bipartite Graphs and Matchings

- A *maximum matching* is a matching with the largest number of edges.
A matching M in a bipartite graph $G = (V, E)$ with bipartition (V_1, V_2) is a *complete matching from V_1 to V_2* if every vertex in V_1 is the endpoint of an edge in the matching, or equivalently, if $|M| = |V_1|$.

Bipartite Graphs and Matchings

- A *maximum matching* is a matching with the largest number of edges. A matching M in a bipartite graph $G = (V, E)$ with bipartition (V_1, V_2) is a *complete matching from V_1 to V_2* if every vertex in V_1 is the endpoint of an edge in the matching, or equivalently, if $|M| = |V_1|$.

Theorem (Hall's Marriage Theorem) The bipartite graph $G = (V, E)$ with bipartition (V_1, V_2) has a complete matching from V_1 to V_2 if and only if $|N(A)| \geq |A|$ for all subsets A of V_1 .

Proof of Hall's Theorem

- **Theorem** (Hall's Marriage Theorem) The bipartite graph $G = (V, E)$ with bipartition (V_1, V_2) has a complete matching from V_1 to V_2 if and only if $|N(A)| \geq |A|$ for all subsets A of V_1 .

Proof of Hall's Theorem

- **Theorem** (Hall's Marriage Theorem) The bipartite graph $G = (V, E)$ with bipartition (V_1, V_2) has a complete matching from V_1 to V_2 if and only if $|N(A)| \geq |A|$ for all subsets A of V_1 .

Proof. “only if” →

Suppose that there is a complete matching M from V_1 to V_2 . Consider an arbitrary subset $A \subseteq V_1$.

Proof of Hall's Theorem

- **Theorem** (Hall's Marriage Theorem) The bipartite graph $G = (V, E)$ with bipartition (V_1, V_2) has a complete matching from V_1 to V_2 if and only if $|N(A)| \geq |A|$ for all subsets A of V_1 .

Proof. “only if” →

Suppose that there is a complete matching M from V_1 to V_2 .

Consider an arbitrary subset $A \subseteq V_1$.

Then, for every vertex $v \in A$, there is an edge in M connecting v to a vertex in V_2 . Thus, there are at least as many vertices in V_2 that are neighbors of vertices in V_1 as there are vertices in V_1 .

Proof of Hall's Theorem

- **Theorem** (Hall's Marriage Theorem) The bipartite graph $G = (V, E)$ with bipartition (V_1, V_2) has a complete matching from V_1 to V_2 if and only if $|N(A)| \geq |A|$ for all subsets A of V_1 .

Proof. “only if” →

Suppose that there is a complete matching M from V_1 to V_2 .

Consider an arbitrary subset $A \subseteq V_1$.

Then, for every vertex $v \in A$, there is an edge in M connecting v to a vertex in V_2 . Thus, there are at least as many vertices in V_2 that are neighbors of vertices in V_1 as there are vertices in V_1 .

Hence, $|N(A)| \geq |A|$.

Proof of Hall's Theorem

- **Theorem** (Hall's Marriage Theorem) The bipartite graph $G = (V, E)$ with bipartition (V_1, V_2) has a complete matching from V_1 to V_2 if and only if $|N(A)| \geq |A|$ for all subsets A of V_1 .

Proof. “if” \leftarrow

Use **strong induction** to prove it.

Proof of Hall's Theorem

- **Theorem** (Hall's Marriage Theorem) The bipartite graph $G = (V, E)$ with bipartition (V_1, V_2) has a complete matching from V_1 to V_2 if and only if $|N(A)| \geq |A|$ for all subsets A of V_1 .

Proof. “if” \leftarrow

Use **strong induction** to prove it.

Basic step: $|V_1| = 1$

Proof of Hall's Theorem

- **Theorem** (Hall's Marriage Theorem) The bipartite graph $G = (V, E)$ with bipartition (V_1, V_2) has a complete matching from V_1 to V_2 if and only if $|N(A)| \geq |A|$ for all subsets A of V_1 .

Proof. “if” \leftarrow

Use **strong induction** to prove it.

Basic step: $|V_1| = 1$

Inductive hypothesis: Let k be a positive integer. If $G = (V, E)$ is a bipartite graph with bipartition (V_1, V_2) , and $|V_1| = j \leq k$, then there is a complete matching M from V_1 to V_2 whenever the condition that $|N(A)| \geq |A|$ for all $A \subseteq V_1$ is met.

Proof of Hall's Theorem

- **Theorem** (Hall's Marriage Theorem) The bipartite graph $G = (V, E)$ with bipartition (V_1, V_2) has a complete matching from V_1 to V_2 if and only if $|N(A)| \geq |A|$ for all subsets A of V_1 .

Proof. “if” \leftarrow

Use **strong induction** to prove it.

Basic step: $|V_1| = 1$

Inductive hypothesis: Let k be a positive integer. If $G = (V, E)$ is a bipartite graph with bipartition (V_1, V_2) , and $|V_1| = j \leq k$, then there is a complete matching M from V_1 to V_2 whenever the condition that $|N(A)| \geq |A|$ for all $A \subseteq V_1$ is met.

Inductive step: suppose that $H = (W, F)$ is a bipartite graph with bipartition (W_1, W_2) and $|W_1| = k + 1$.



Proof of Hall's Theorem

- **Theorem** (Hall's Marriage Theorem) The bipartite graph $G = (V, E)$ with bipartition (V_1, V_2) has a complete matching from V_1 to V_2 if and only if $|N(A)| \geq |A|$ for all subsets A of V_1 .

Proof. “if” \leftarrow

Inductive step: suppose that $H = (W, F)$ is a bipartite graph with bipartition (W_1, W_2) and $|W_1| = k + 1$.

Proof of Hall's Theorem

- **Theorem** (Hall's Marriage Theorem) The bipartite graph $G = (V, E)$ with bipartition (V_1, V_2) has a complete matching from V_1 to V_2 if and only if $|N(A)| \geq |A|$ for all subsets A of V_1 .

Proof. “if” \leftarrow

Inductive step: suppose that $H = (W, F)$ is a bipartite graph with bipartition (W_1, W_2) and $|W_1| = k + 1$.

Case (i): For all integers j with $1 \leq j \leq k$, the vertices in every set of j elements from W_1 are adjacent to at least $j + 1$ elements of W_2

Case (ii): For some integer j with $1 \leq j \leq k$, there is a subset W'_1 of j vertices such that there are exactly j neighbors of these vertices in W_2

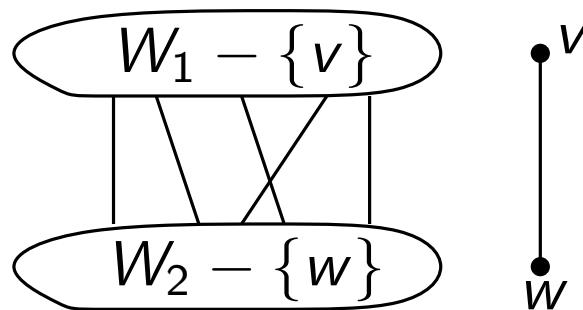
Proof of Hall's Theorem

- **Theorem** (Hall's Marriage Theorem) The bipartite graph $G = (V, E)$ with bipartition (V_1, V_2) has a complete matching from V_1 to V_2 if and only if $|N(A)| \geq |A|$ for all subsets A of V_1 .

Proof. “if” \leftarrow

Inductive step: suppose that $H = (W, F)$ is a bipartite graph with bipartition (W_1, W_2) and $|W_1| = k + 1$.

Case (i):



Proof of Hall's Theorem

- **Theorem** (Hall's Marriage Theorem) The bipartite graph $G = (V, E)$ with bipartition (V_1, V_2) has a complete matching from V_1 to V_2 if and only if $|N(A)| \geq |A|$ for all subsets A of V_1 .

Proof. “if” \leftarrow

Inductive step: suppose that $H = (W, F)$ is a bipartite graph with bipartition (W_1, W_2) and $|W_1| = k + 1$.

Case (ii): For some integer j with $1 \leq j \leq k$, there is a subset W'_1 of j vertices such that there are exactly j neighbors of these vertices in W_2

Proof of Hall's Theorem

- **Theorem** (Hall's Marriage Theorem) The bipartite graph $G = (V, E)$ with bipartition (V_1, V_2) has a complete matching from V_1 to V_2 if and only if $|N(A)| \geq |A|$ for all subsets A of V_1 .

Proof. “if” \leftarrow

Inductive step: suppose that $H = (W, F)$ is a bipartite graph with bipartition (W_1, W_2) and $|W_1| = k + 1$.

Case (ii): For some integer j with $1 \leq j \leq k$, there is a subset W'_1 of j vertices such that there are exactly j neighbors of these vertices in W_2 .

Let W'_2 be the set of these neighbors. Then by i.h., there is a complete matching from W'_1 to W'_2 . Now consider the graph $K = (W_1 - W'_1, W_2 - W'_2)$. We will show that the condition $|N(A)| \geq |A|$ is met for all subsets A of $W_1 - W'_1$.



Proof of Hall's Theorem

- **Theorem** (Hall's Marriage Theorem) The bipartite graph $G = (V, E)$ with bipartition (V_1, V_2) has a complete matching from V_1 to V_2 if and only if $|N(A)| \geq |A|$ for all subsets A of V_1 .

Proof. “if” \leftarrow

Inductive step: suppose that $H = (W, F)$ is a bipartite graph with bipartition (W_1, W_2) and $|W_1| = k + 1$.

Case (ii): For some integer j with $1 \leq j \leq k$, there is a subset W'_1 of j vertices such that there are exactly j neighbors of these vertices in W_2 .

Let W'_2 be the set of these neighbors. Then by i.h., there is a complete matching from W'_1 to W'_2 . Now consider the graph $K = (W_1 - W'_1, W_2 - W'_2)$. We will show that the condition $|N(A)| \geq |A|$ is met for all subsets A of $W_1 - W'_1$.

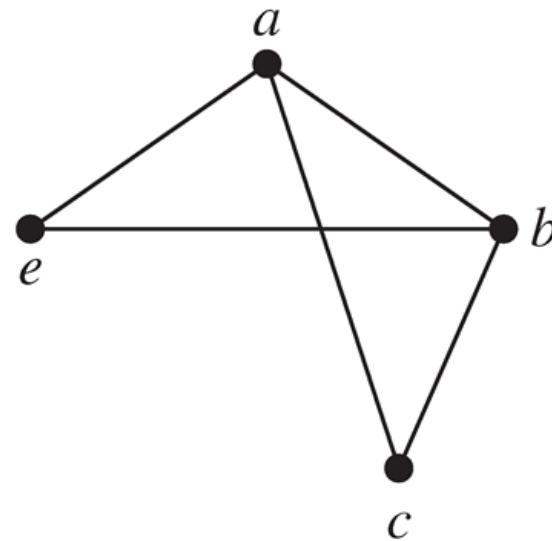
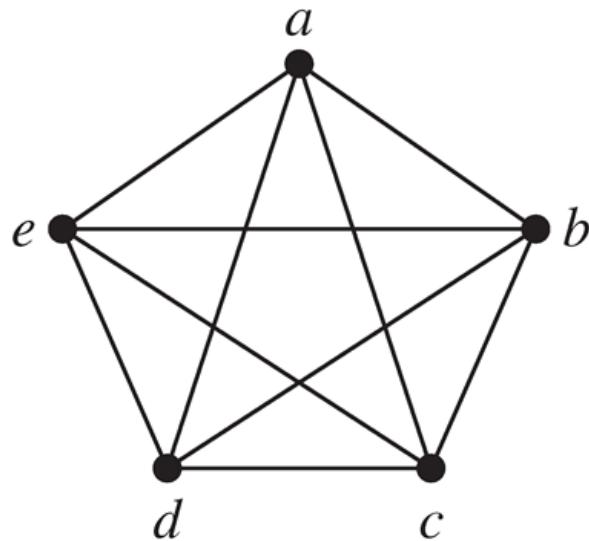
If not, there is a subset B of t vertices with $1 \leq t \leq k + 1 - j$ s.t. $|N(B)| < t$.

Subgraphs

- **Definition** A *subgraph of a graph* $G = (V, E)$ is a graph (W, F) , where $W \subseteq V$ and $F \subseteq E$. A subgraph H of G is a *proper subgraph* of G if $H \neq G$.

Subgraphs

- **Definition** A *subgraph of a graph* $G = (V, E)$ is a graph (W, F) , where $W \subseteq V$ and $F \subseteq E$. A subgraph H of G is a *proper subgraph* of G if $H \neq G$.

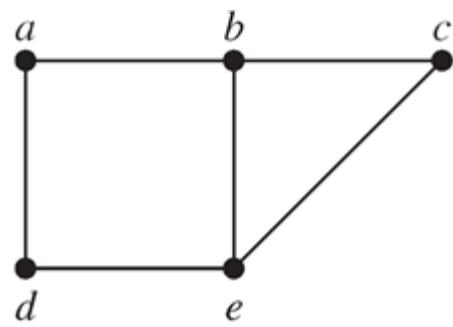


Union of Graphs

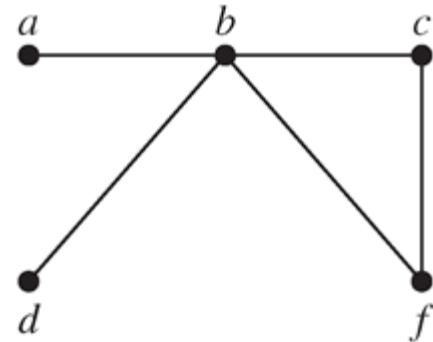
- **Definition** The *union of two simple graphs* $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ is the simple graph with vertex set $V_1 \cup V_2$ and edge set $E_1 \cup E_2$, denoted by $G_1 \cup G_2$.

Union of Graphs

- **Definition** The *union of two simple graphs* $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ is the simple graph with vertex set $V_1 \cup V_2$ and edge set $E_1 \cup E_2$, denoted by $G_1 \cup G_2$.



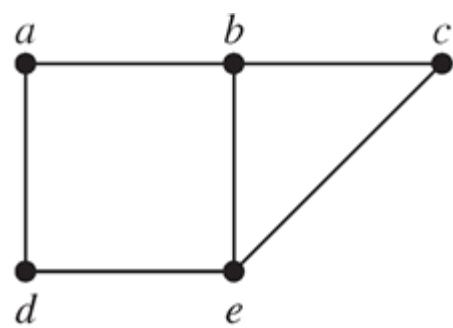
G_1



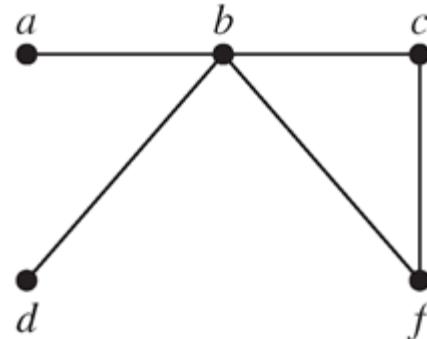
G_2

Union of Graphs

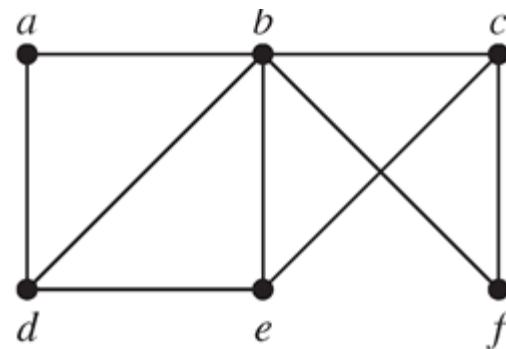
- **Definition** The *union of two simple graphs* $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ is the simple graph with vertex set $V_1 \cup V_2$ and edge set $E_1 \cup E_2$, denoted by $G_1 \cup G_2$.



G_1



G_2



$G_1 \cup G_2$

Representation of Graphs

- To represent a graph, we may use *adjacency lists*, *adjacency matrices*, and *incidence matrices*.

Representation of Graphs

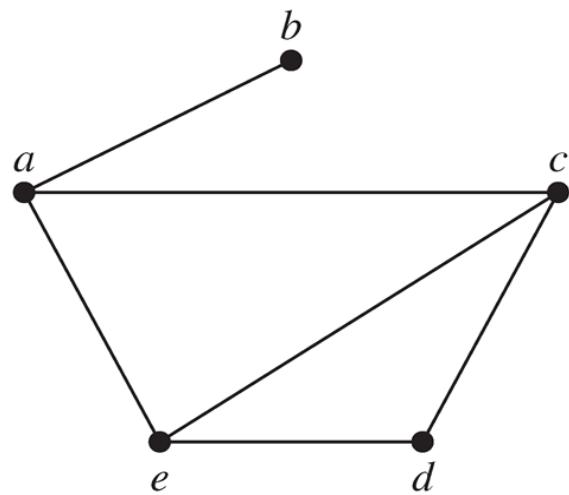
- To represent a graph, we may use *adjacency lists*, *adjacency matrices*, and *incidence matrices*.

Definition An *adjacency list* can be used to represent a graph with **no multiple edges** by specifying the vertices that are adjacent to each vertex of the graph.

Representation of Graphs

- To represent a graph, we may use *adjacency lists*, *adjacency matrices*, and *incidence matrices*.

Definition An *adjacency list* can be used to represent a graph with **no multiple edges** by specifying the vertices that are adjacent to each vertex of the graph.



Representation of Graphs

- To represent a graph, we may use *adjacency lists*, *adjacency matrices*, and *incidence matrices*.

Definition An *adjacency list* can be used to represent a graph with **no multiple edges** by specifying the vertices that are adjacent to each vertex of the graph.

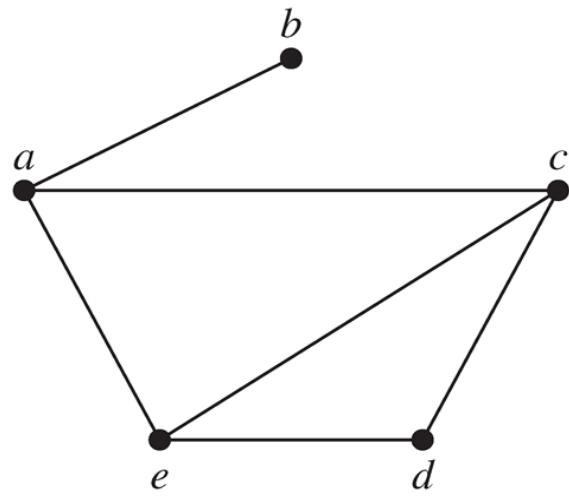
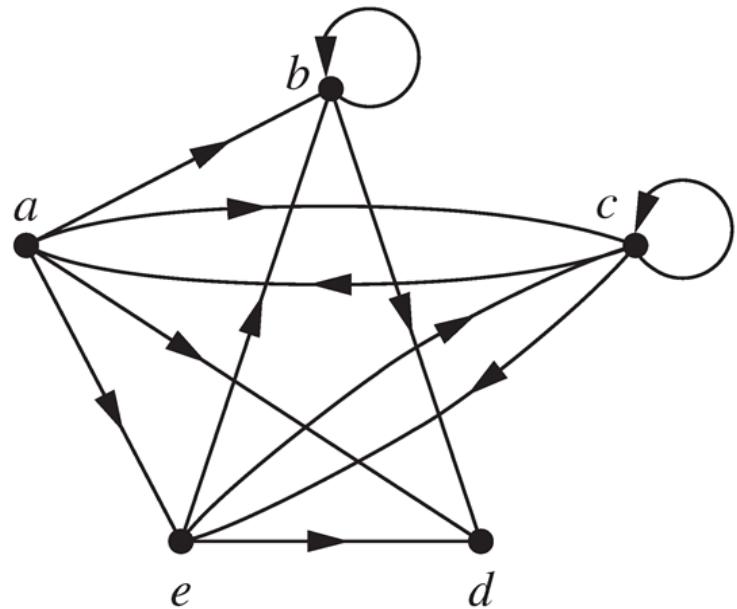


TABLE 1 An Adjacency List for a Simple Graph.

Vertex	Adjacent Vertices
a	b, c, e
b	a
c	a, d, e
d	c, e
e	a, c, d

Representation of Graphs

- **Definition** An *adjacency list* can be used to represent a graph with **no multiple edges** by specifying the vertices that are adjacent to each vertex of the graph.



Representation of Graphs

- **Definition** An *adjacency list* can be used to represent a graph with **no multiple edges** by specifying the vertices that are adjacent to each vertex of the graph.

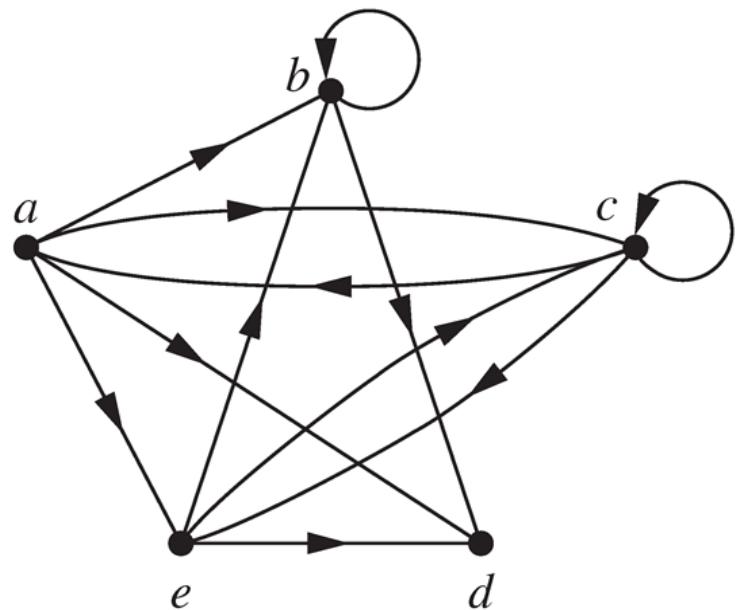


TABLE 2 An Adjacency List for a Directed Graph.

<i>Initial Vertex</i>	<i>Terminal Vertices</i>
<i>a</i>	<i>b, c, d, e</i>
<i>b</i>	<i>b, d</i>
<i>c</i>	<i>a, c, e</i>
<i>d</i>	
<i>e</i>	<i>b, c, d</i>

Adjacency Matrices

- **Definition** Suppose that $G = (V, E)$ is a simple graph with $|V| = n$. Arbitrarily list the vertices of G as v_1, v_2, \dots, v_n . The *adjacency matrix* \mathbf{A}_G of G , is the $n \times n$ zero-one matrix with 1 as its (i, j) -th entry when v_i and v_j are adjacent, and 0 as its (i, j) -th entry when they are not adjacent.

Adjacency Matrices

- **Definition** Suppose that $G = (V, E)$ is a simple graph with $|V| = n$. Arbitrarily list the vertices of G as v_1, v_2, \dots, v_n . The *adjacency matrix* \mathbf{A}_G of G , is the $n \times n$ zero-one matrix with 1 as its (i, j) -th entry when v_i and v_j are adjacent, and 0 as its (i, j) -th entry when they are not adjacent.

$\mathbf{A}_G = [a_{ij}]_{n \times n}$, where

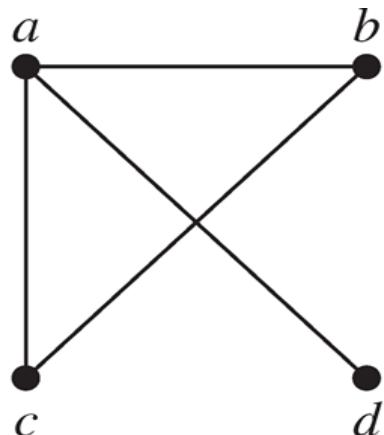
$$a_{ij} = \begin{cases} 1 & \text{if } \{v_i, v_j\} \text{ is an edge of } G, \\ 0 & \text{otherwise.} \end{cases}$$

Adjacency Matrices

- **Definition** Suppose that $G = (V, E)$ is a simple graph with $|V| = n$. Arbitrarily list the vertices of G as v_1, v_2, \dots, v_n . The *adjacency matrix* \mathbf{A}_G of G , is the $n \times n$ zero-one matrix with 1 as its (i, j) -th entry when v_i and v_j are adjacent, and 0 as its (i, j) -th entry when they are not adjacent.

$$\mathbf{A}_G = [a_{ij}]_{n \times n}, \text{ where}$$

$$a_{ij} = \begin{cases} 1 & \text{if } \{v_i, v_j\} \text{ is an edge of } G, \\ 0 & \text{otherwise.} \end{cases}$$

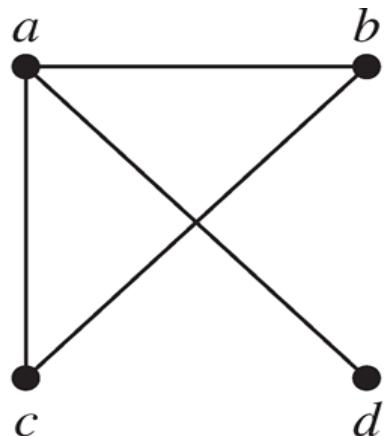


Adjacency Matrices

- **Definition** Suppose that $G = (V, E)$ is a simple graph with $|V| = n$. Arbitrarily list the vertices of G as v_1, v_2, \dots, v_n . The *adjacency matrix* \mathbf{A}_G of G , is the $n \times n$ zero-one matrix with 1 as its (i, j) -th entry when v_i and v_j are adjacent, and 0 as its (i, j) -th entry when they are not adjacent.

$$\mathbf{A}_G = [a_{ij}]_{n \times n}, \text{ where}$$

$$a_{ij} = \begin{cases} 1 & \text{if } \{v_i, v_j\} \text{ is an edge of } G, \\ 0 & \text{otherwise.} \end{cases}$$



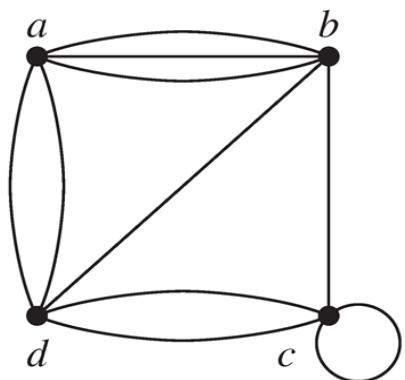
$$\begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}$$

Adjacency Matrices

- Adjacency matrices can also be used to represent graphs with loops and multiple edges. The matrix is no longer a zero-one matrix.

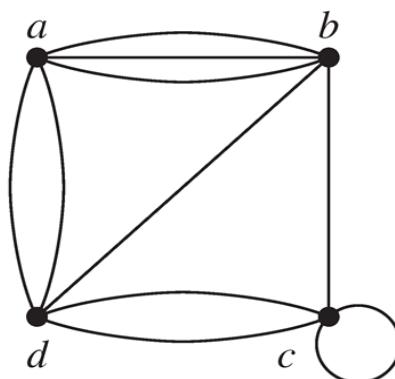
Adjacency Matrices

- Adjacency matrices can also be used to represent graphs with loops and multiple edges. The matrix is no longer a zero-one matrix.



Adjacency Matrices

- Adjacency matrices can also be used to represent graphs with loops and multiple edges. The matrix is no longer a zero-one matrix.



$$\begin{bmatrix} 0 & 3 & 0 & 2 \\ 3 & 0 & 1 & 1 \\ 0 & 1 & 1 & 2 \\ 2 & 1 & 2 & 0 \end{bmatrix}$$

Incidence Matrices

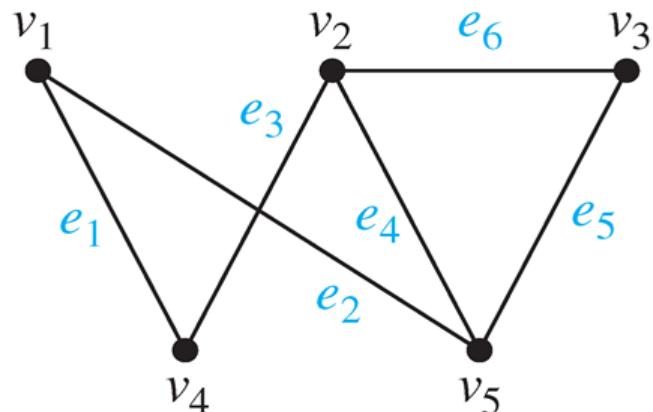
- **Definition** Let $G = (V, E)$ be an undirected graph with vertices v_1, v_2, \dots, v_n and edges e_1, e_2, \dots, e_m . The *incidence matrix* with respect to the ordering of V and E is the $n \times m$ matrix $\mathbf{M} = [m_{ij}]_{n \times m}$, where

$$m_{ij} = \begin{cases} 1 & \text{if edge } e_j \text{ is incident with } v_i, \\ 0 & \text{otherwise.} \end{cases}$$

Incidence Matrices

- **Definition** Let $G = (V, E)$ be an undirected graph with vertices v_1, v_2, \dots, v_n and edges e_1, e_2, \dots, e_m . The *incidence matrix* with respect to the ordering of V and E is the $n \times m$ matrix $\mathbf{M} = [m_{ij}]_{n \times m}$, where

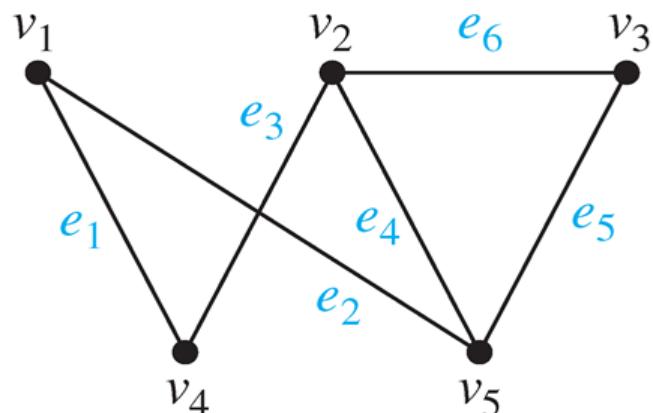
$$m_{ij} = \begin{cases} 1 & \text{if edge } e_j \text{ is incident with } v_i, \\ 0 & \text{otherwise.} \end{cases}$$



Incidence Matrices

- **Definition** Let $G = (V, E)$ be an undirected graph with vertices v_1, v_2, \dots, v_n and edges e_1, e_2, \dots, e_m . The *incidence matrix* with respect to the ordering of V and E is the $n \times m$ matrix $\mathbf{M} = [m_{ij}]_{n \times m}$, where

$$m_{ij} = \begin{cases} 1 & \text{if edge } e_j \text{ is incident with } v_i, \\ 0 & \text{otherwise.} \end{cases}$$

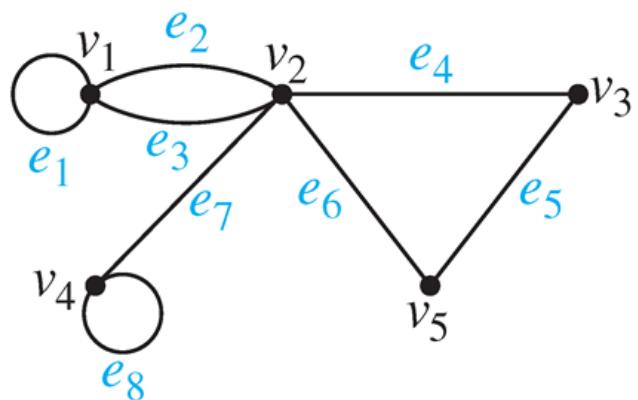


$$\begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 & 0 \end{bmatrix}$$

Incidence Matrices

- **Definition** Let $G = (V, E)$ be an undirected graph with vertices v_1, v_2, \dots, v_n and edges e_1, e_2, \dots, e_m . The *incidence matrix* with respect to the ordering of V and E is the $n \times m$ matrix $\mathbf{M} = [m_{ij}]_{n \times m}$, where

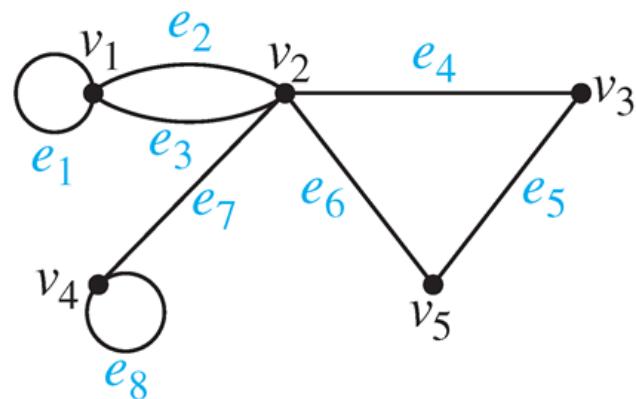
$$m_{ij} = \begin{cases} 1 & \text{if edge } e_j \text{ is incident with } v_i, \\ 0 & \text{otherwise.} \end{cases}$$



Incidence Matrices

- **Definition** Let $G = (V, E)$ be an undirected graph with vertices v_1, v_2, \dots, v_n and edges e_1, e_2, \dots, e_m . The *incidence matrix* with respect to the ordering of V and E is the $n \times m$ matrix $\mathbf{M} = [m_{ij}]_{n \times m}$, where

$$m_{ij} = \begin{cases} 1 & \text{if edge } e_j \text{ is incident with } v_i, \\ 0 & \text{otherwise.} \end{cases}$$



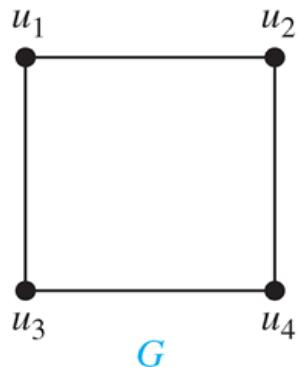
$$\begin{bmatrix} 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \end{bmatrix}$$

Isomorphism of Graphs

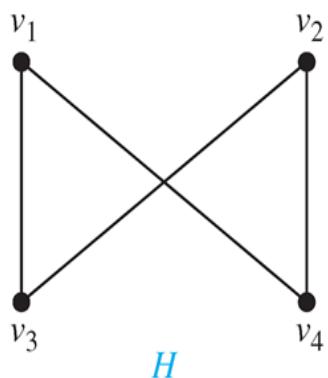
- **Definition** The simple graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ are *isomorphic* if there is a one-to-one and onto function from V_1 to V_2 with the property that a and b are adjacent in G_1 if and only if $f(a)$ and $f(b)$ are adjacent in G_2 , for all a and b in V_1 . Such a function is called an *isomorphism*.

Isomorphism of Graphs

- **Definition** The simple graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ are *isomorphic* if there is a one-to-one and onto function from V_1 to V_2 with the property that a and b are adjacent in G_1 if and only if $f(a)$ and $f(b)$ are adjacent in G_2 , for all a and b in V_1 . Such a function is called an *isomorphism*.

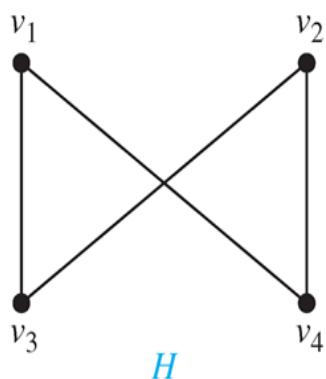
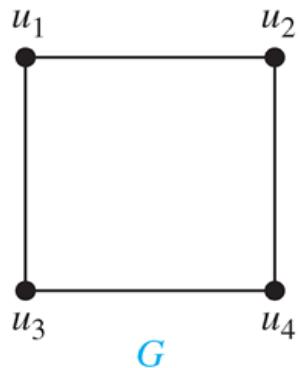


Are the two graphs *isomorphic*?



Isomorphism of Graphs

- **Definition** The simple graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ are *isomorphic* if there is a one-to-one and onto function from V_1 to V_2 with the property that a and b are adjacent in G_1 if and only if $f(a)$ and $f(b)$ are adjacent in G_2 , for all a and b in V_1 . Such a function is called an *isomorphism*.



Are the two graphs *isomorphic*?

Define a one-to-one correspondence:
 $f(u_1) = v_1$, $f(u_2) = v_4$, $f(u_3) = v_3$, and
 $f(u_4) = v_2$

Isomorphism of Graphs

- It is usually **difficult** to determine whether two simple graphs are isomorphic **using brute force** since there are $n!$ possible one-to-one correspondences.

Isomorphism of Graphs

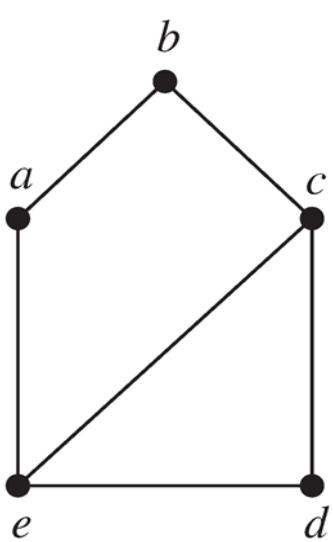
- It is usually **difficult** to determine whether two simple graphs are isomorphic **using brute force** since there are $n!$ possible one-to-one correspondences.
- Sometimes it is **not difficult** to show that two graphs are **not isomorphic**. We can achieve this by checking some **graph invariants**.

Isomorphism of Graphs

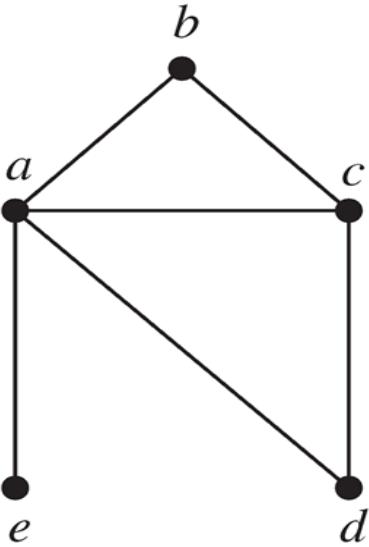
- It is usually **difficult** to determine whether two simple graphs are isomorphic **using brute force** since there are $n!$ possible one-to-one correspondences.
- Sometimes it is **not difficult** to show that two graphs are **not isomorphic**. We can achieve this by checking some *graph invariants*.
- Useful graph invariants include the number of vertices, number of edges, degree sequence, etc.

Isomorphism of Graphs

- **Example** Determine whether these two graphs are **isomorphic**.



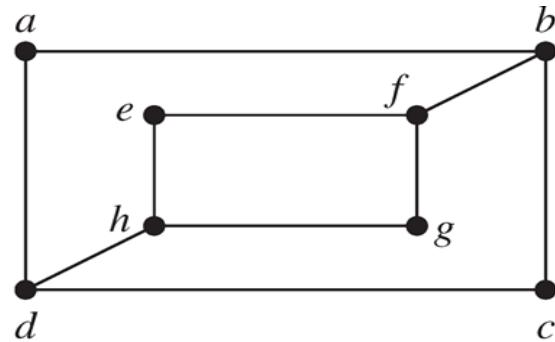
G



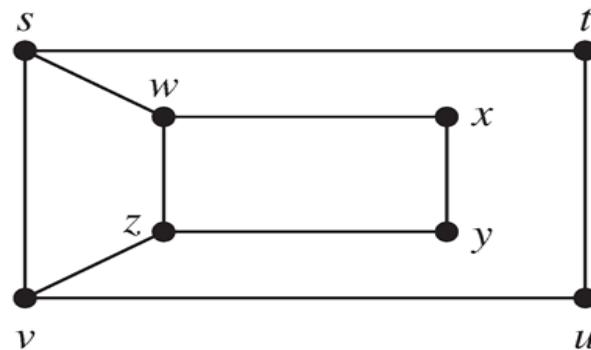
H

Isomorphism of Graphs

- **Example** Determine whether these two graphs are **isomorphic**.



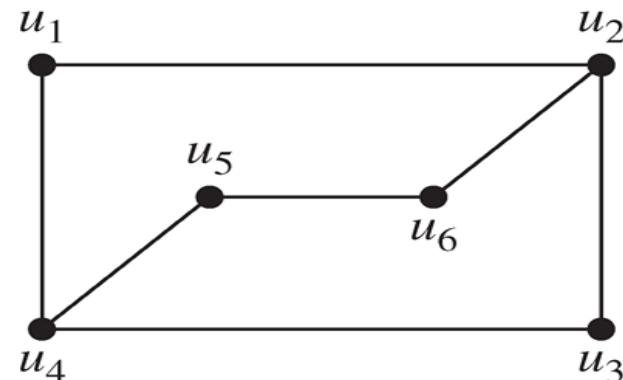
G



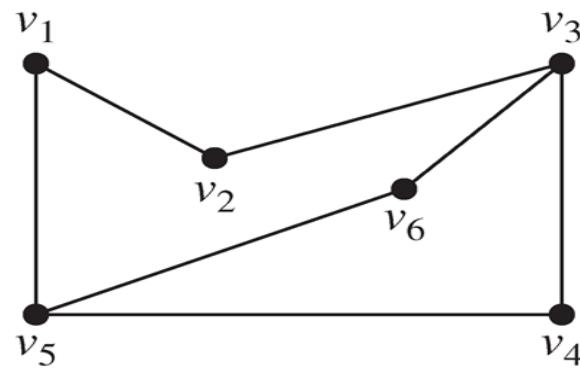
H

Isomorphism of Graphs

- **Example** Determine whether these two graphs are **isomorphic**.



G



H

Path

- **Definition** Let n be a nonnegative integer and G an undirected graph. A *path of length n* from u to v in G is a sequence of n edges e_1, e_2, \dots, e_n of G for which there exists a sequence $x_0 = u, x_1, \dots, x_{n-1}, x_n = v$ of vertices such that e_i has the endpoints x_{i-1} and x_i for $i = 1, \dots, n$. The path is a *circuit* if it begins and ends at the same vertex, i.e., if $u = v$ and has length greater than zero. A path or circuit is *simple* if it does not contain repeating edges.

Path

- **Definition** Let n be a nonnegative integer and G an undirected graph. A *path of length n* from u to v in G is a sequence of n edges e_1, e_2, \dots, e_n of G for which there exists a sequence $x_0 = u, x_1, \dots, x_{n-1}, x_n = v$ of vertices such that e_i has the endpoints x_{i-1} and x_i for $i = 1, \dots, n$. The path is a *circuit* if it begins and ends at the same vertex, i.e., if $u = v$ and has length greater than zero. A path or circuit is *simple* if it does not contain repeating edges.
 - ◊ it starts and ends with a vertex
 - ◊ each edge joins the vertex before it in the sequence to the vertex after it in the sequence
 - ◊ no edge appears more than once in the sequence

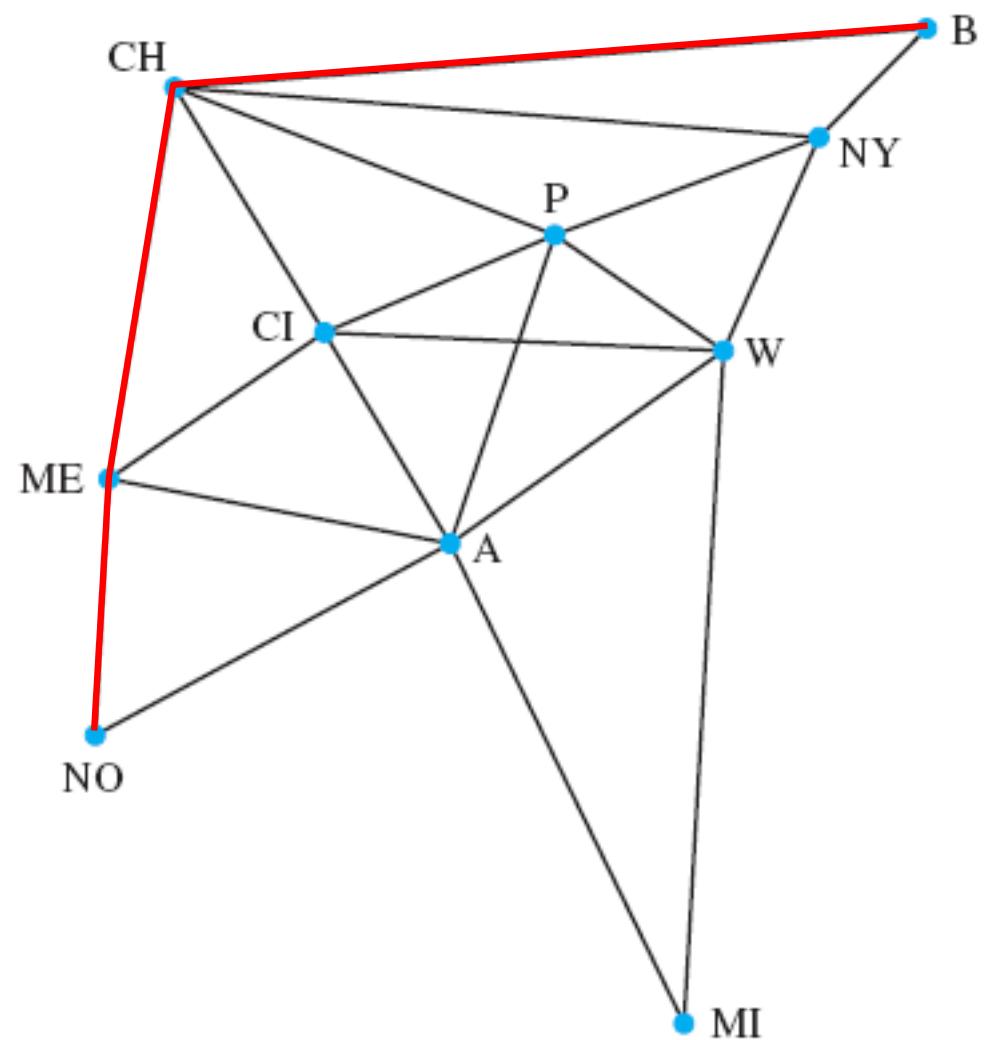
Path

- **Definition** Let n be a nonnegative integer and G an undirected graph. A *path of length n* from u to v in G is a sequence of n edges e_1, e_2, \dots, e_n of G for which there exists a sequence $x_0 = u, x_1, \dots, x_{n-1}, x_n = v$ of vertices such that e_i has the endpoints x_{i-1} and x_i for $i = 1, \dots, n$. The path is a *circuit* if it begins and ends at the same vertex, i.e., if $u = v$ and has length greater than zero. A path or circuit is *simple* if it does not contain repeating edges.
 - ◊ it starts and ends with a vertex
 - ◊ each edge joins the vertex before it in the sequence to the vertex after it in the sequence
 - ◊ no edge appears more than once in the sequence

Length of a path = # of edges on path

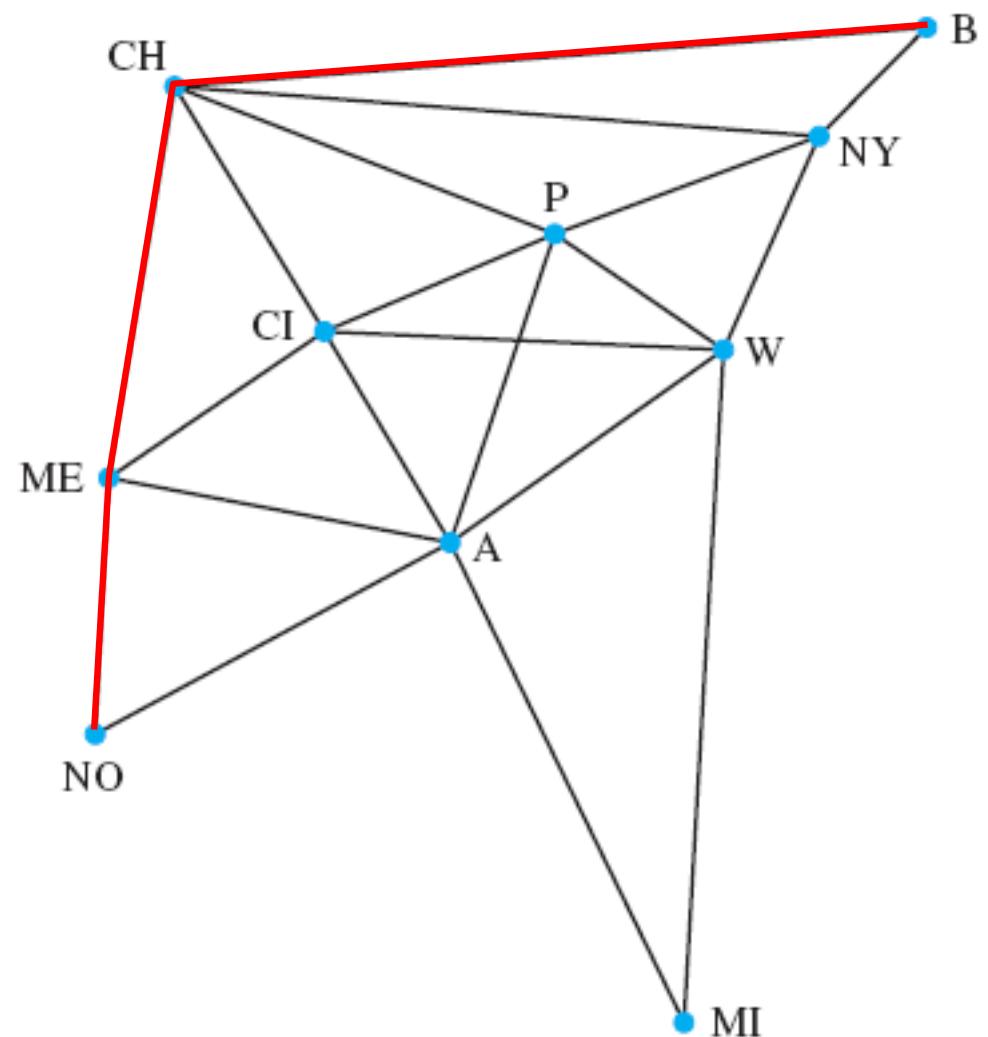


Path



Path

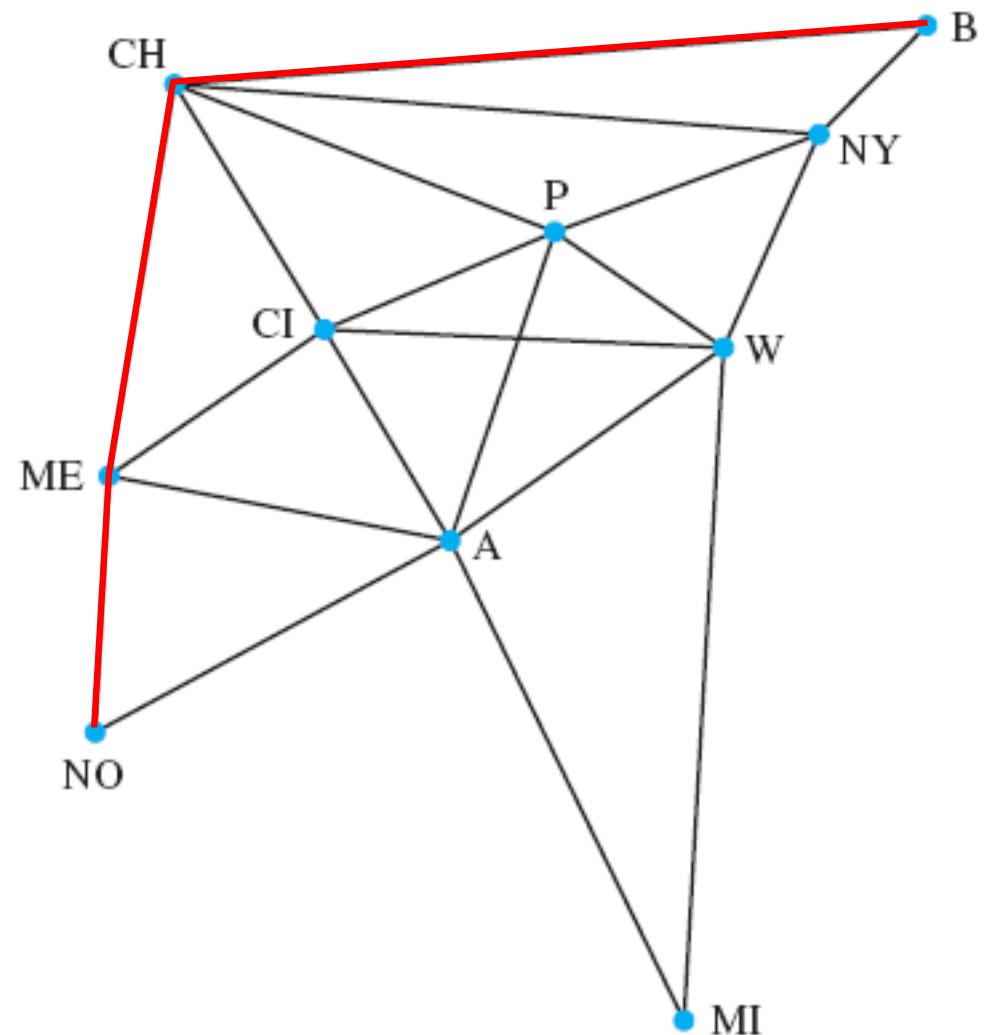
Path from Boston to New Orleans is B, CH, ME, NO



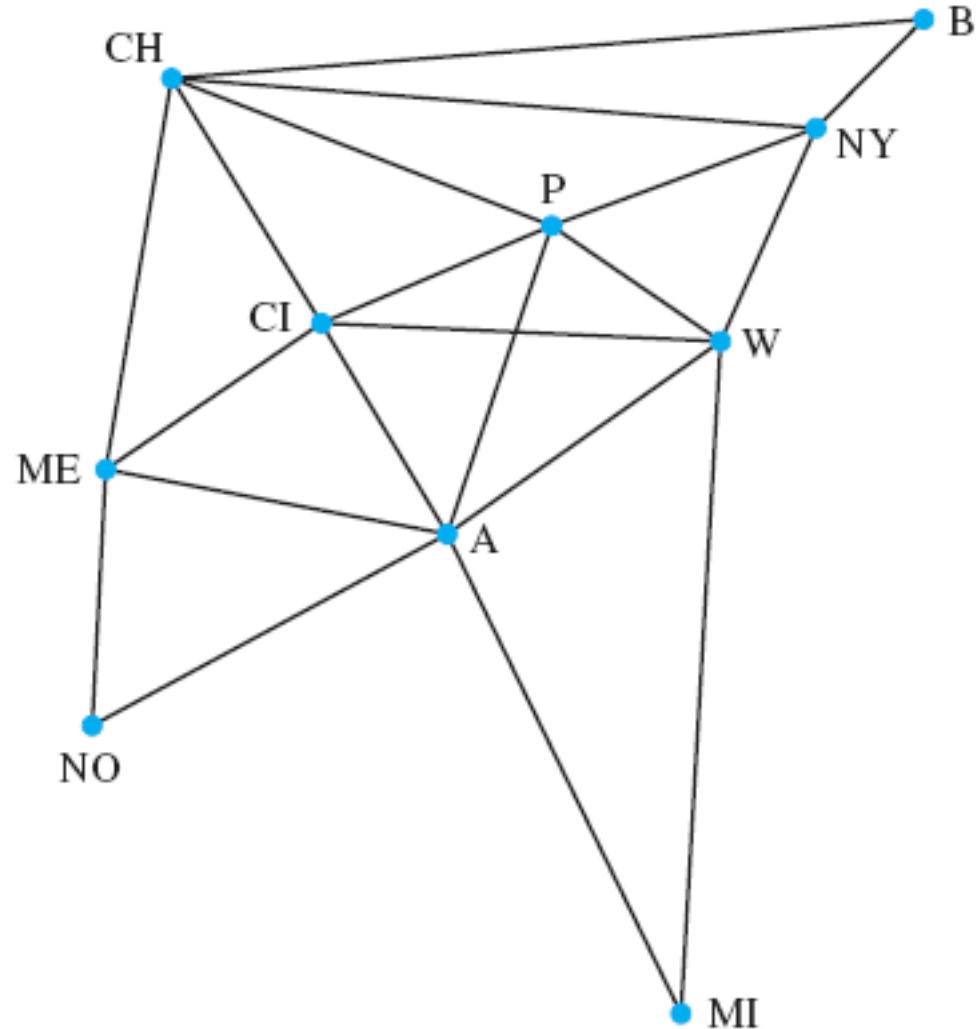
Path

Path from Boston to New Orleans is B, CH, ME, NO

This path has length 3.



Connectivity



Company decides to lease only **minimum number** of communication lines it needs to be able to send a message from any city to any other city by using any number of intermediate cities.

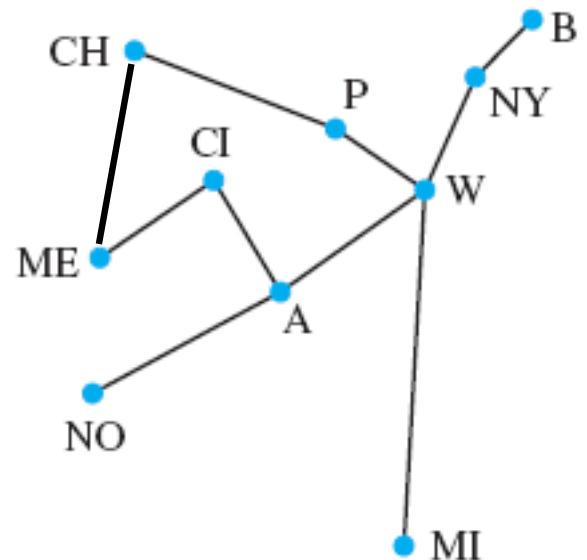
What is the **minimum** number of lines it needs to lease?

Connectivity

- Choosing 10 edges?

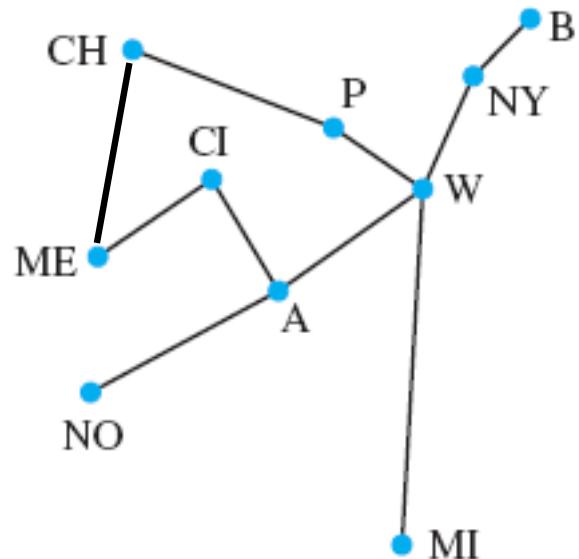
Connectivity

- Choosing 10 edges?



Connectivity

- Choosing 10 edges?

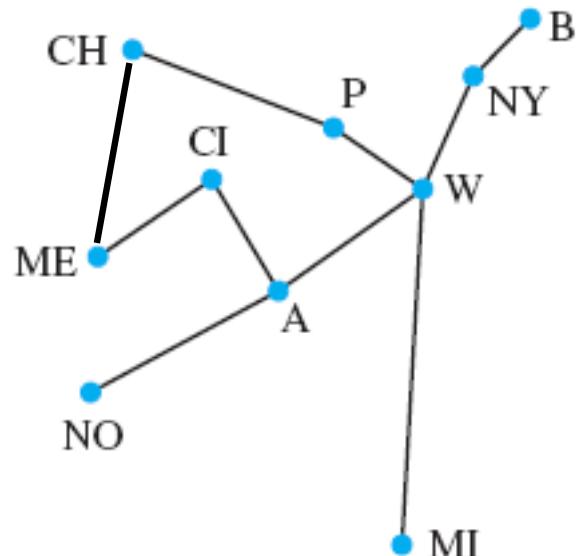


Too many.

Could throw away edge CL, A, and still have a solution.

Connectivity

- Choosing 10 edges?



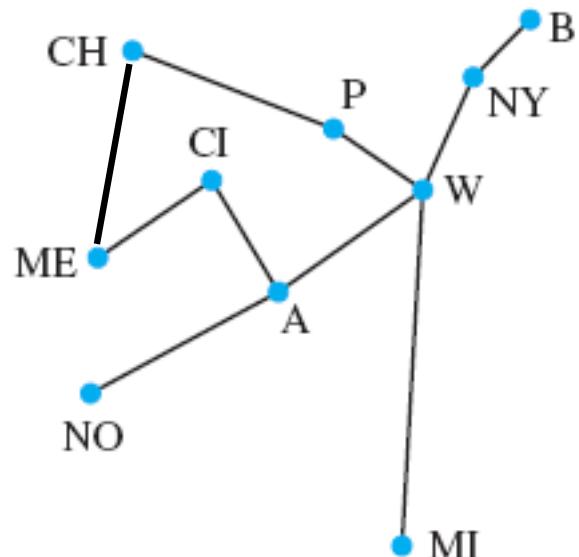
Choosing 8 edges?

Too many.

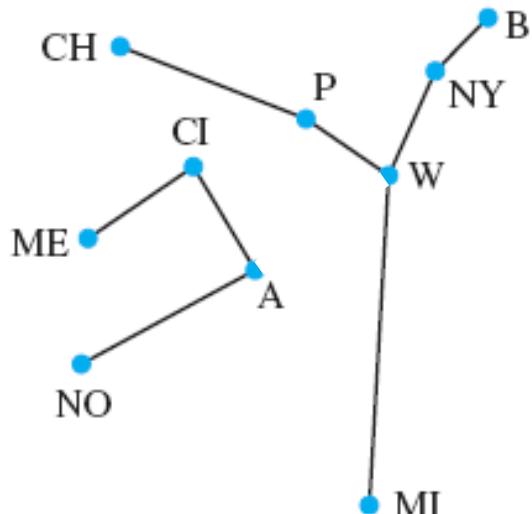
Could throw away edge CL, A, and still have a solution.

Connectivity

- Choosing 10 edges?



- Choosing 8 edges?

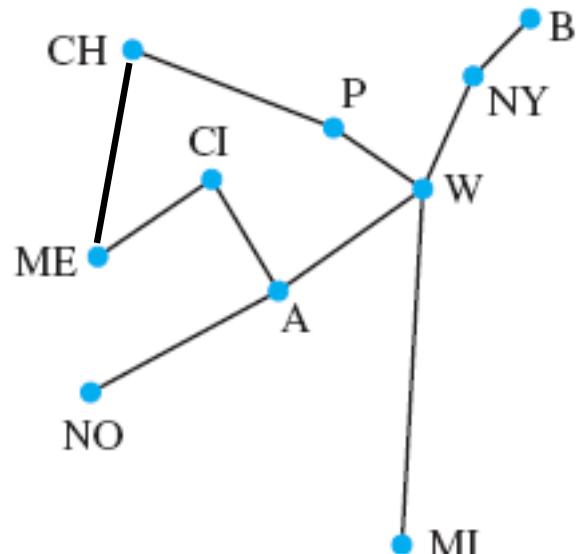


Too many.

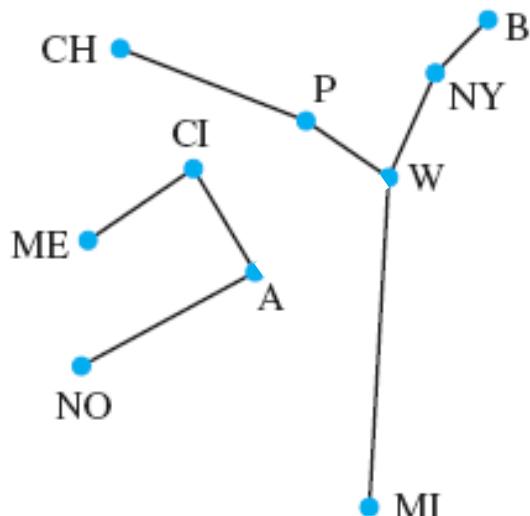
Could throw away edge **CL, A**, and still have a solution.

Connectivity

- Choosing 10 edges?



- Choosing 8 edges?



Too many.

Could throw away edge CL, A, and still have a solution.

Not enough.

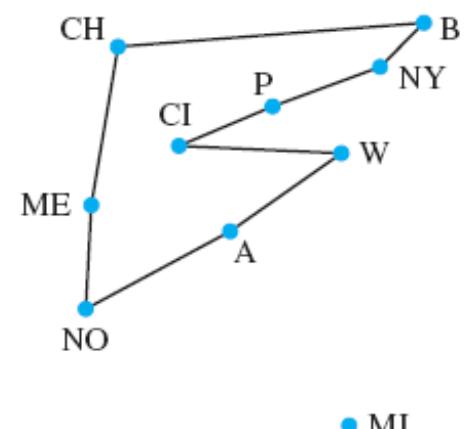
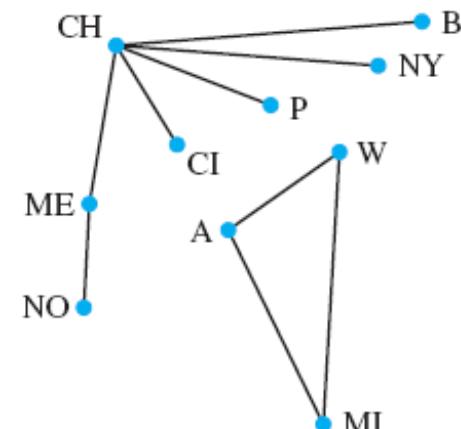
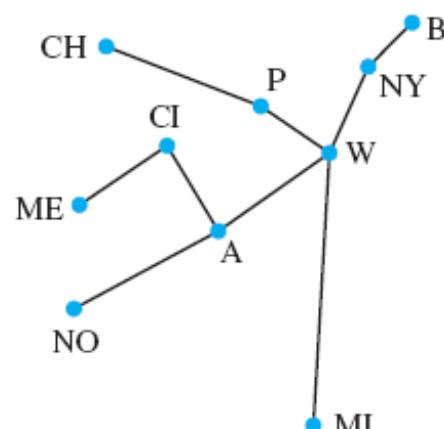
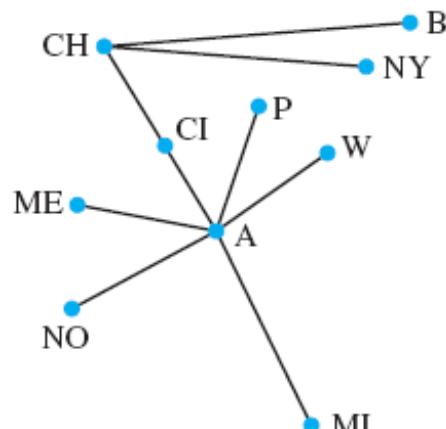
There is no path from, e.g., NO to B.

Connectivity

- Choosing 9 edges:

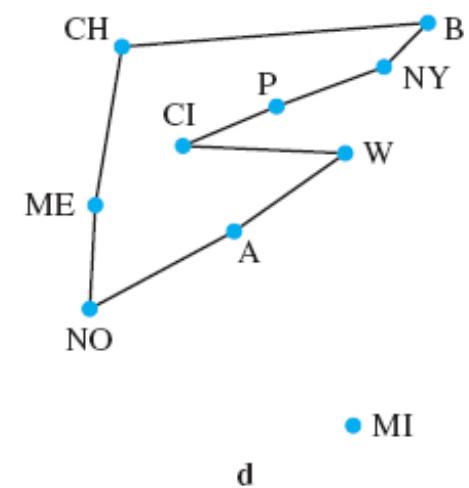
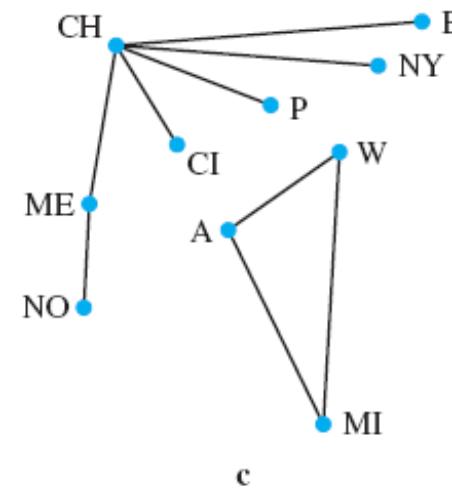
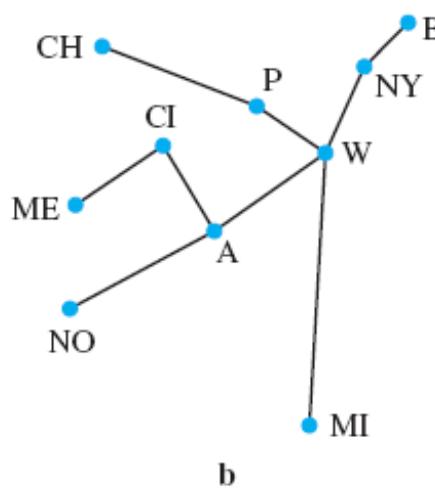
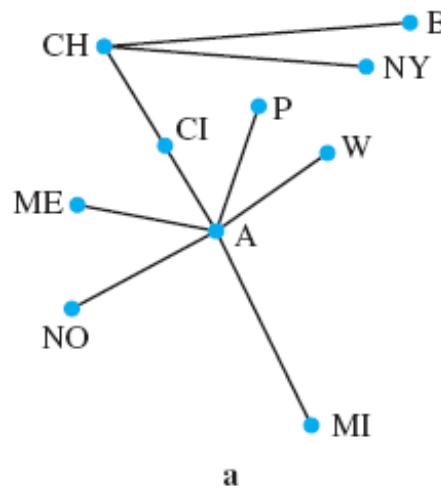
Connectivity

- Choosing 9 edges:



Connectivity

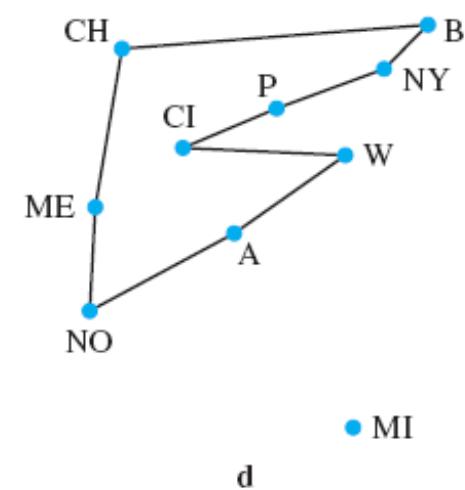
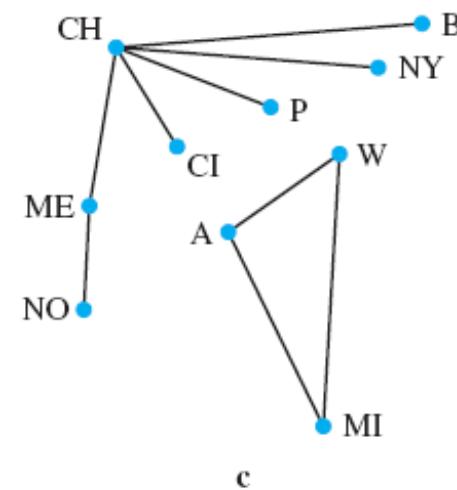
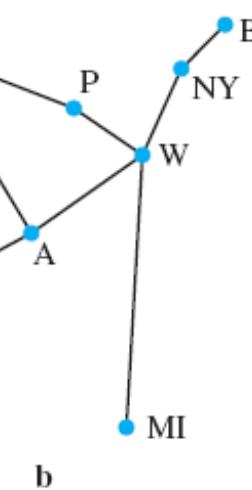
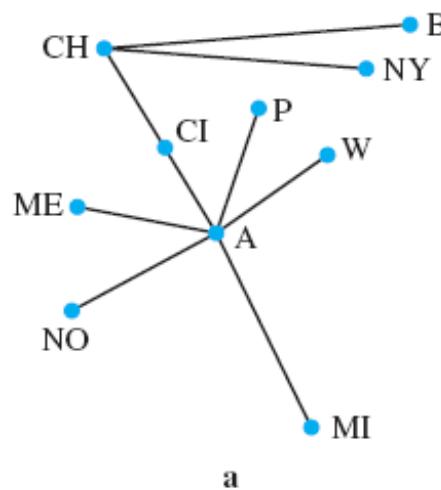
- Choosing 9 edges:



Two vertices are *connected* if there is a path between them.

Connectivity

- Choosing 9 edges:

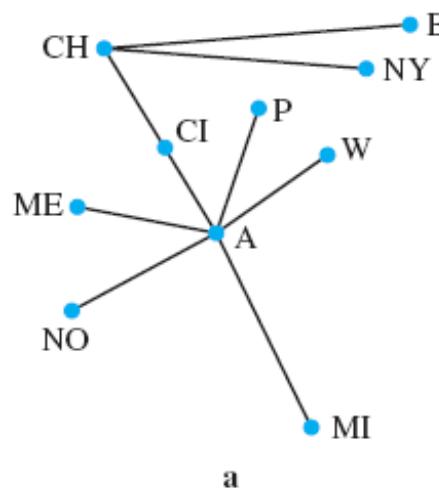


Two vertices are *connected* if there is a path between them.

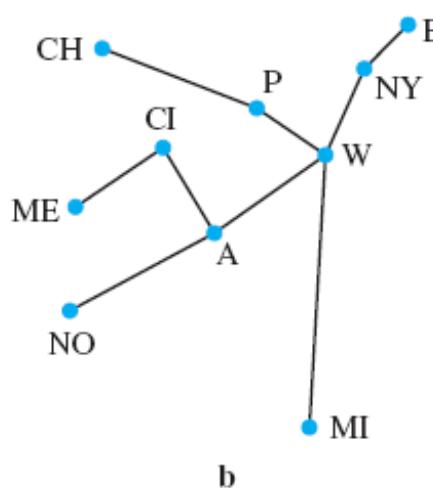
Example: W, B are *connected* in (b), but are *disconnected* in (c).

Connectivity

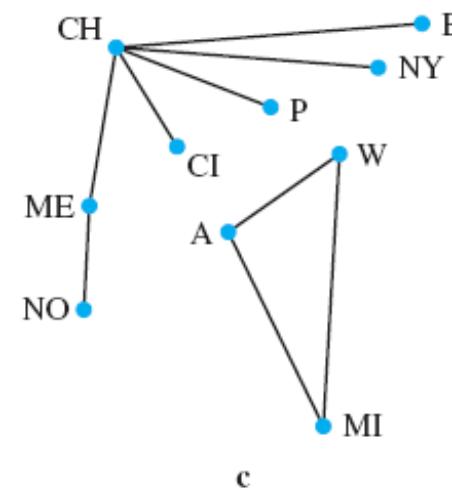
- Choosing 9 edges:



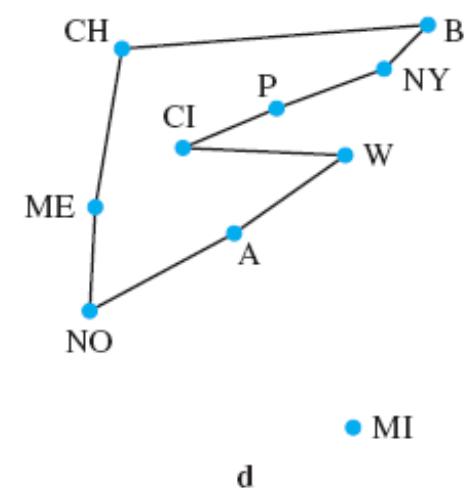
a



b



c



d

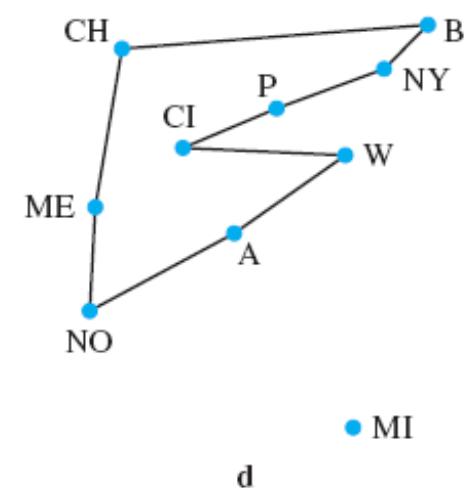
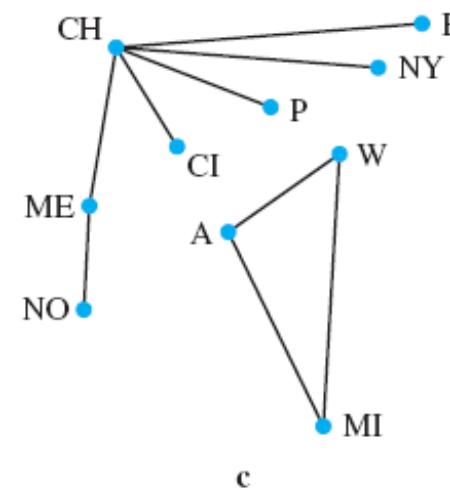
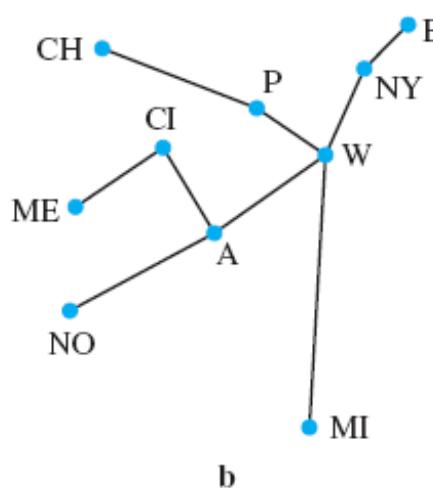
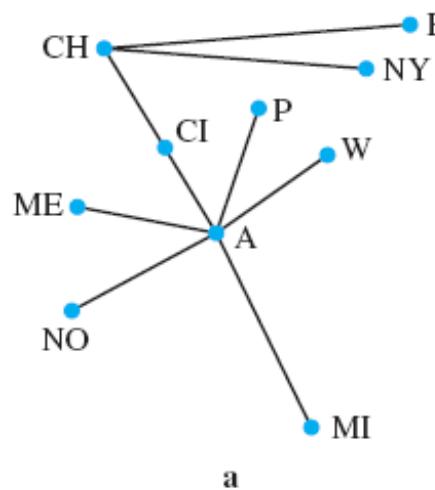
Two vertices are *connected* if there is a path between them.

Example: W, B are *connected* in (b), but are *disconnected* in (c).

Definition An undirected graph is called *connected* if there is a path between every pair of distinct vertices of the graph.

Connectivity

- Choosing 9 edges:



Two vertices are *connected* if there is a path between them.

Example: W, B are *connected* in (b), but are *disconnected* in (c).

Definition An undirected graph is called *connected* if there is a path between every pair of distinct vertices of the graph.

Example: (a) and (b) are *connected*, (c) and (d) are *disconnected*.

Path

- **Lemma** If there is a path between two distinct vertices x and y of a graph G , then there is a simple path between x and y in G .

Path

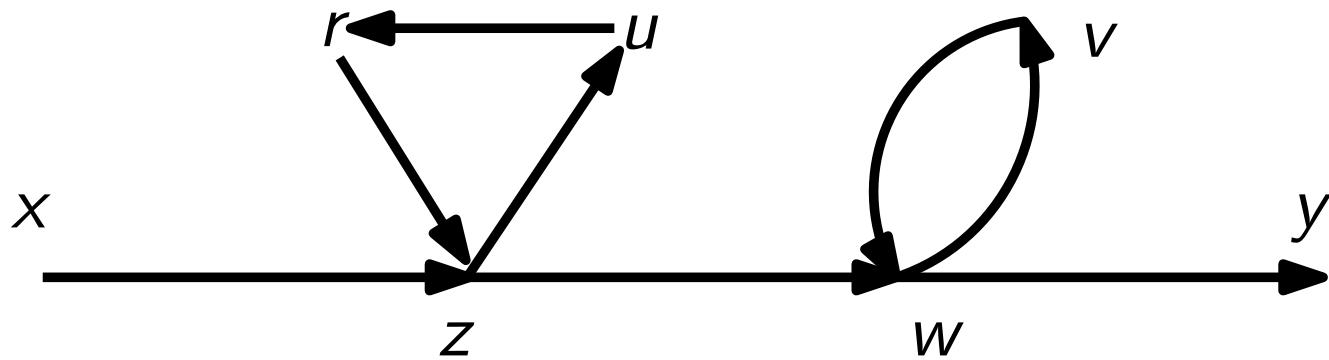
- **Lemma** If there is a path between two distinct vertices x and y of a graph G , then there is a simple path between x and y in G .

Proof Just delete cycles (loops).

Path

- **Lemma** If there is a path between two distinct vertices x and y of a graph G , then there is a simple path between x and y in G .

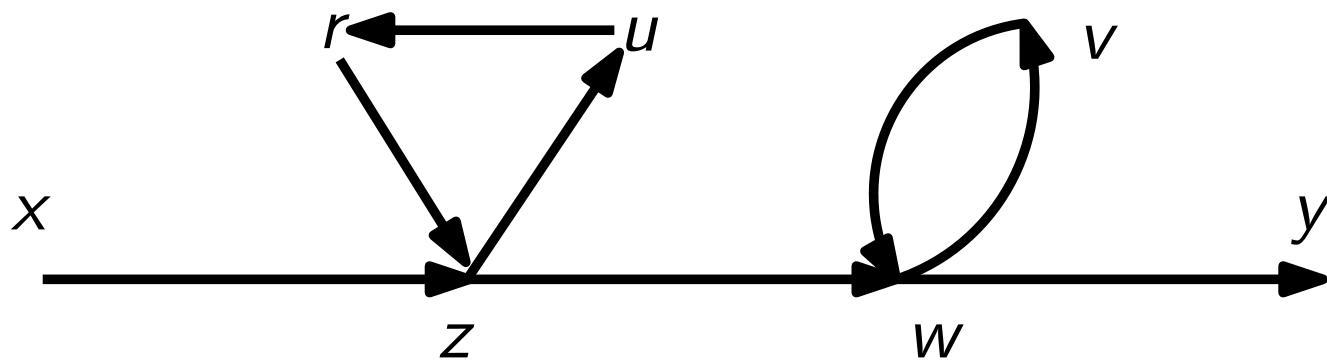
Proof Just delete cycles (loops).



Path

- **Lemma** If there is a path between two distinct vertices x and y of a graph G , then there is a simple path between x and y in G .

Proof Just delete cycles (loops).



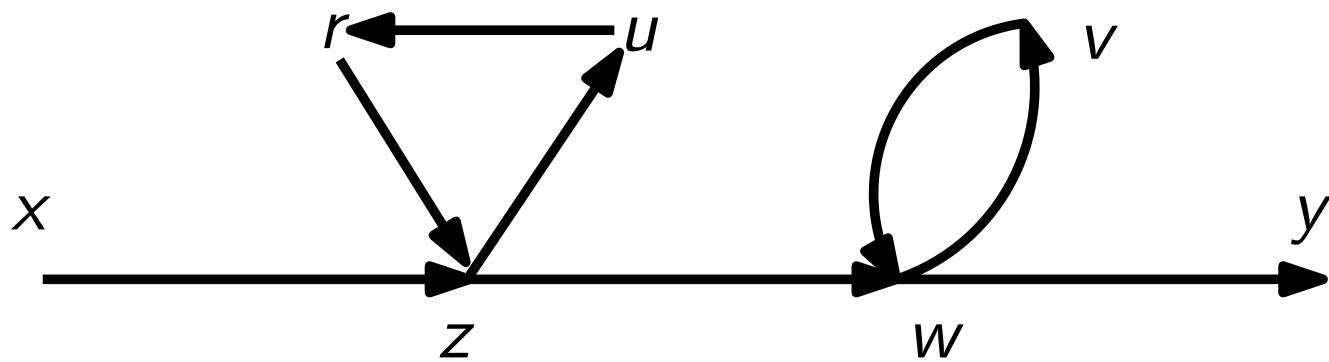
Path from x to y

$x, z, u, r, z, w, v, w, y$.

Path

- **Lemma** If there is a path between two distinct vertices x and y of a graph G , then there is a simple path between x and y in G .

Proof Just delete cycles (loops).



Path from x to y

$x, z, u, r, z, w, v, w, y$.

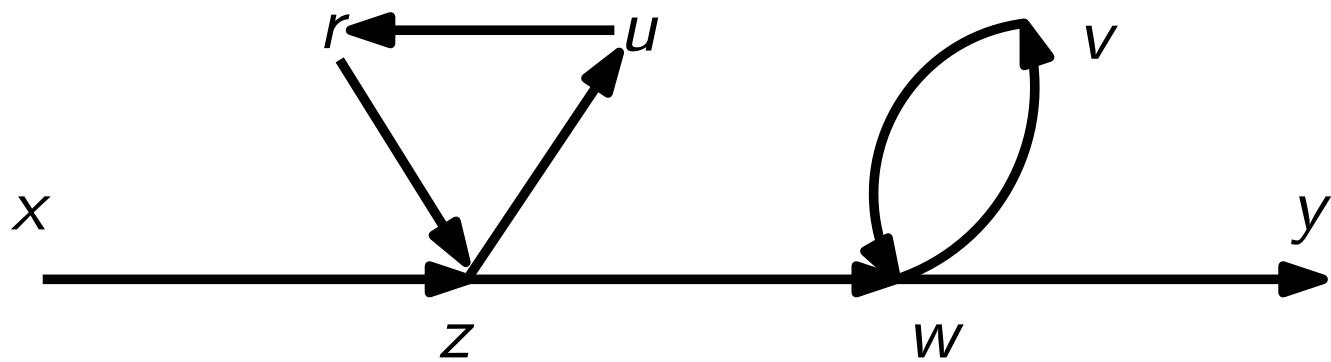
Path from x to y

x, z, w, y .

Path

- **Lemma** If there is a path between two distinct vertices x and y of a graph G , then there is a simple path between x and y in G .

Proof Just delete cycles (loops).



Path from x to y

$x, z, u, r, z, w, v, w, y$.

Path from x to y

x, z, w, y .

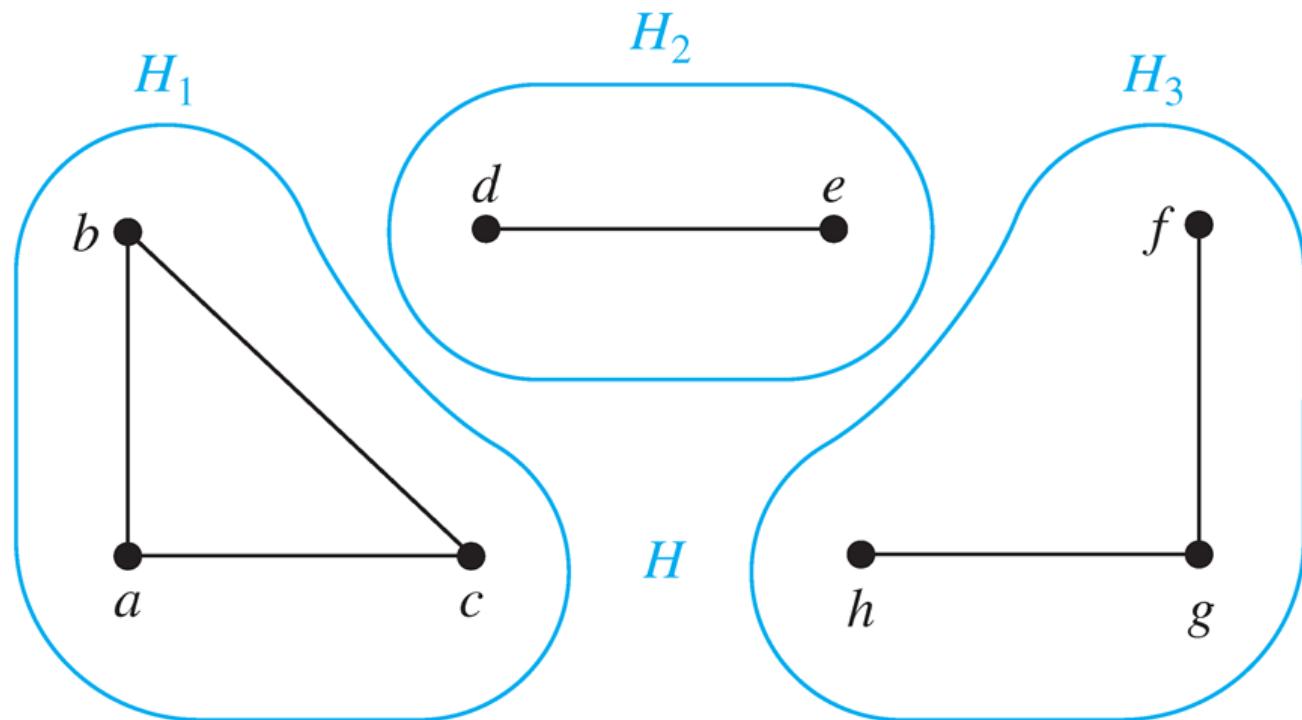
Theorem There is a simple path between every pair of distinct vertices of a connected undirected graph.

Connected Components

- **Definition** A *connected component* of a graph G is a connected subgraph of G that is not a proper subgraph of another connected subgraph of G .

Connected Components

- **Definition** A *connected component* of a graph G is a connected subgraph of G that is not a proper subgraph of another connected subgraph of G .



Connectedness in Directed Graphs

- **Definition** A directed graph is *strongly connected* if there is a path **from a to b** and a path **from b to a** whenever a and b are vertices in the graph.

Connectedness in Directed Graphs

- **Definition** A directed graph is *strongly connected* if there is a path **from a to b** and a path **from b to a** whenever a and b are vertices in the graph.

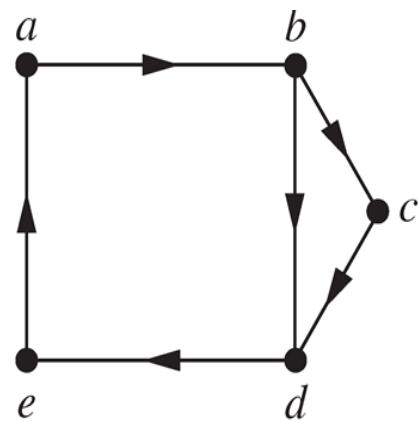
Definition A directed graph is *weakly connected* if there is a path between **every two vertices in the underlying undirected graph**, which is the undirected graph obtained by ignoring the directions of the edges in the directed graph.



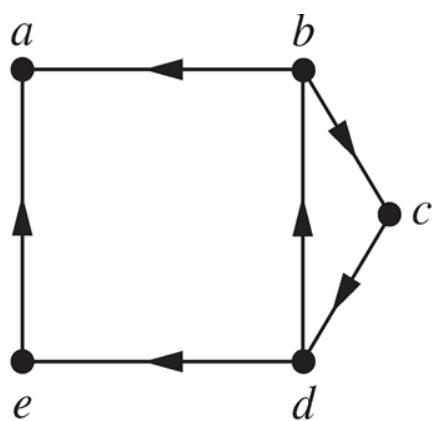
Connectedness in Directed Graphs

- **Definition** A directed graph is *strongly connected* if there is a path **from a to b** and a path **from b to a** whenever a and b are vertices in the graph.

Definition A directed graph is *weakly connected* if there is a path between **every two vertices in the underlying undirected graph**, which is the undirected graph obtained by ignoring the directions of the edges in the directed graph.



G



H

Cut Vertices and Cut Edges

- Sometimes the removal from a graph of a vertex and all incident edges **disconnect** the graph. Such vertices are called *cut vertices*. Similarly we may define *cut edges*.

Cut Vertices and Cut Edges

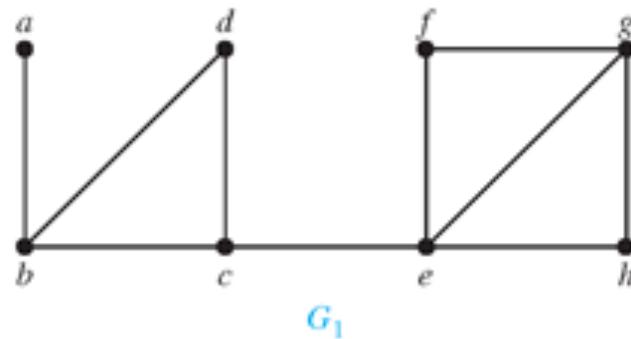
- Sometimes the removal from a graph of a vertex and all incident edges **disconnect** the graph. Such vertices are called *cut vertices*. Similarly we may define *cut edges*.

A set of edges E' is called an *edge cut* of G if the subgraph $G - E'$ is **disconnected**. The *edge connectivity* $\lambda(G)$ is the **minimum number** of edges in an edge cut of G .

Cut Vertices and Cut Edges

- Sometimes the removal from a graph of a vertex and all incident edges **disconnect** the graph. Such vertices are called *cut vertices*. Similarly we may define *cut edges*.

A set of edges E' is called an *edge cut* of G if the subgraph $G - E'$ is **disconnected**. The *edge connectivity* $\lambda(G)$ is the **minimum number** of edges in an edge cut of G .

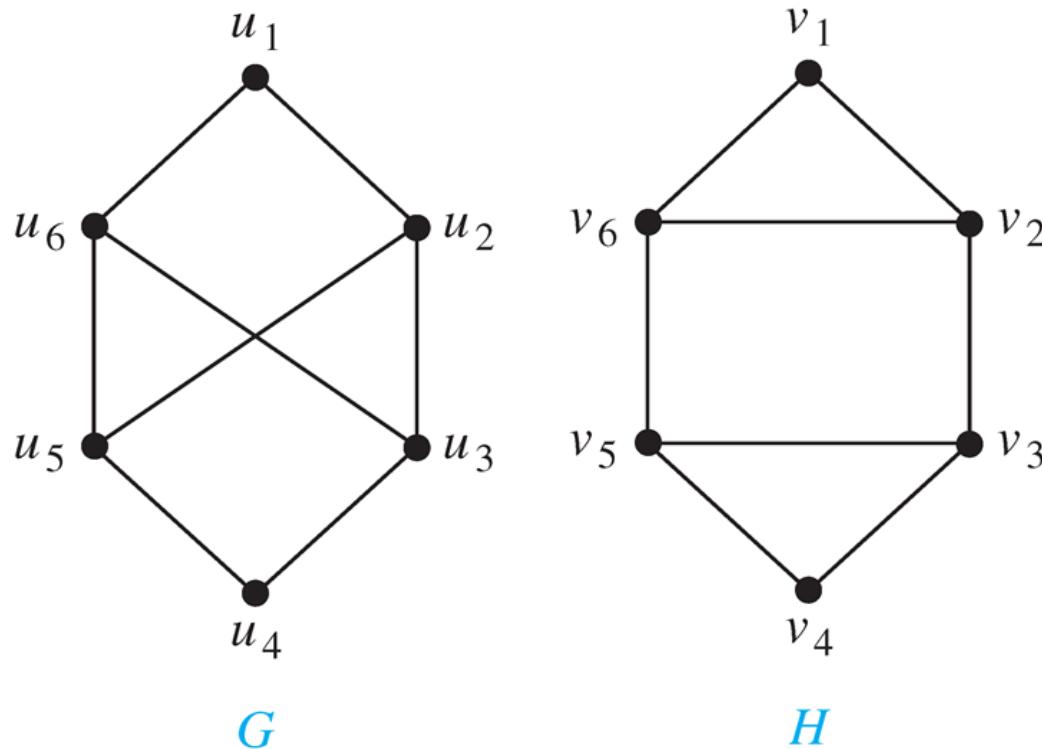


Paths and Isomorphism

- The existence of a simple circuit of length k is **isomorphic invariant**. In addition, **paths** can be used to construct mappings that may be **isomorphisms**.

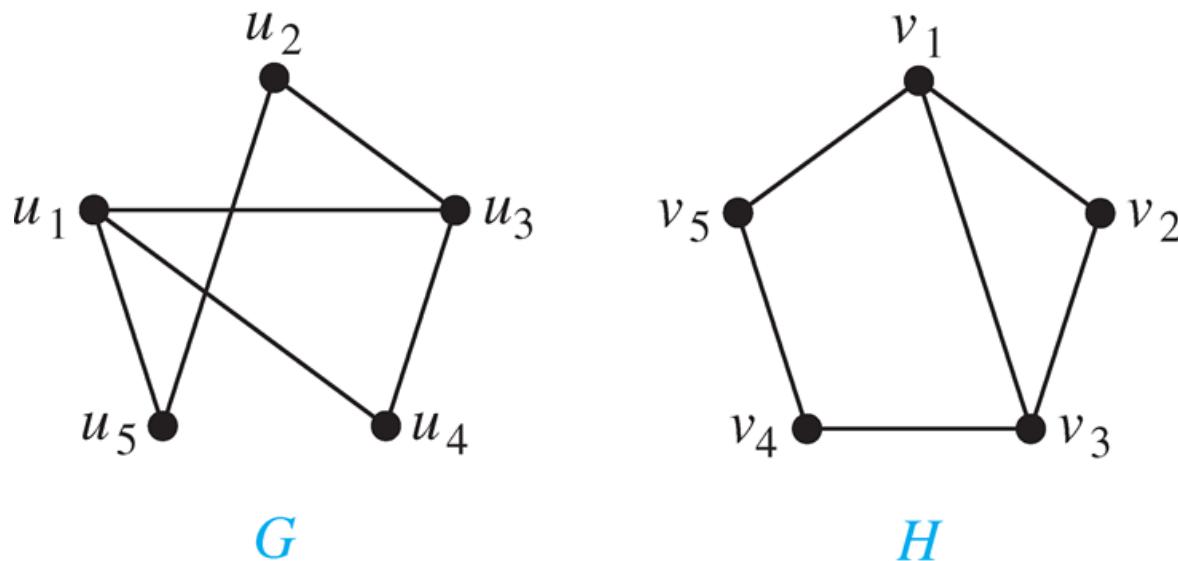
Paths and Isomorphism

- The existence of a simple circuit of length k is **isomorphic invariant**. In addition, **paths** can be used to construct mappings that may be **isomorphisms**.



Paths and Isomorphism

- The existence of a simple circuit of length k is **isomorphic invariant**. In addition, **paths** can be used to construct mappings that may be **isomorphisms**.



Counting Paths between Vertices

- **Theorem** Let G be a graph with adjacency matrix \mathbf{A} with respect to the ordering v_1, v_2, \dots, v_n of vertices. The number of different paths of length r from v_i to v_j , where $r > 0$ is positive, equals the (i, j) -th entry of \mathbf{A}^r .

Counting Paths between Vertices

- **Theorem** Let G be a graph with adjacency matrix \mathbf{A} with respect to the ordering v_1, v_2, \dots, v_n of vertices. The number of different paths of length r from v_i to v_j , where $r > 0$ is positive, equals the (i, j) -th entry of \mathbf{A}^r .

Proof (by **induction**)

Counting Paths between Vertices

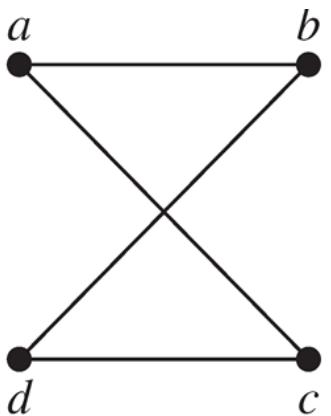
- **Theorem** Let G be a graph with adjacency matrix \mathbf{A} with respect to the ordering v_1, v_2, \dots, v_n of vertices. The number of different paths of length r from v_i to v_j , where $r > 0$ is positive, equals the (i, j) -th entry of \mathbf{A}^r .

Proof (by induction)

$\mathbf{A}^{r+1} = \mathbf{A}^r \mathbf{A}$, the (i, j) -th entry of \mathbf{A}^{r+1} equals $b_{i1}a_{1j} + b_{i2}a_{2j} + \dots + b_{in}a_{nj}$, where b_{ik} is the (i, k) -th entry of \mathbf{A}^r .

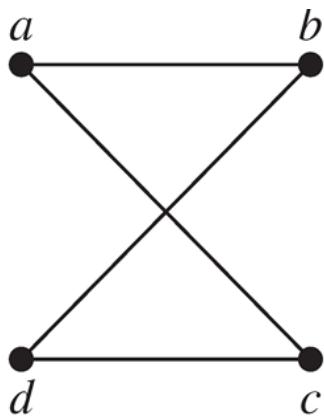
Counting Paths between Vertices

- **Example** How many paths of length 4 are there from a to d in the graph G ?



Counting Paths between Vertices

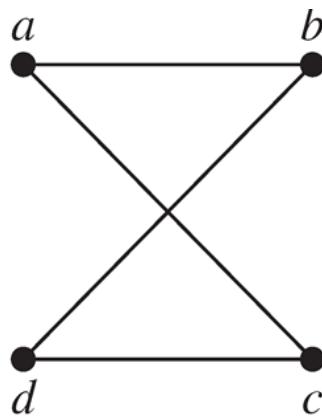
- **Example** How many paths of length 4 are there from a to d in the graph G ?



$$\begin{bmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{bmatrix}$$

Counting Paths between Vertices

- **Example** How many paths of length 4 are there from a to d in the graph G ?



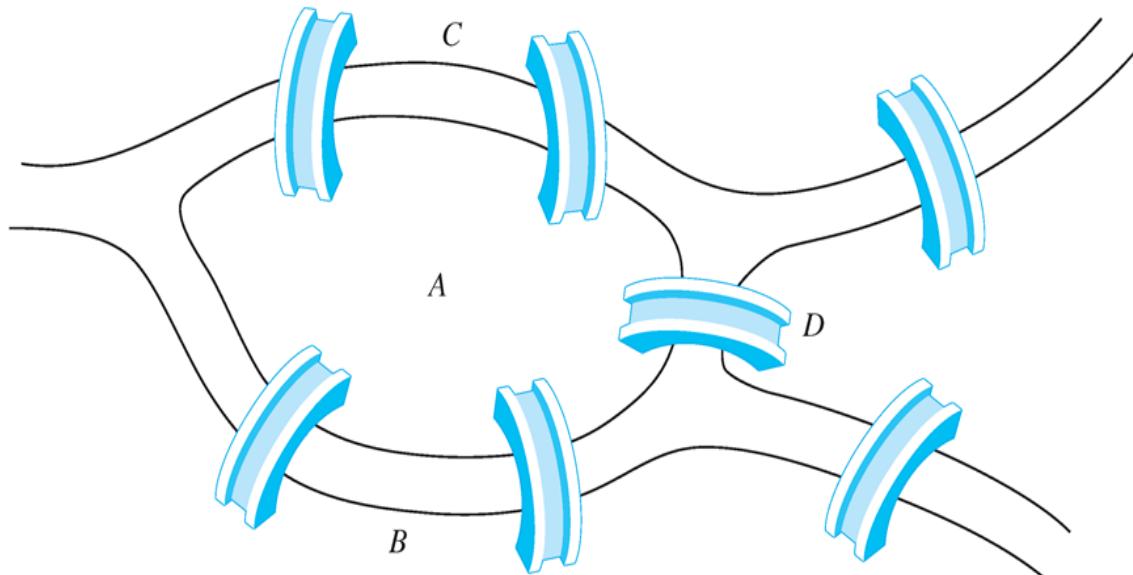
$$\begin{bmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{bmatrix}$$

$$\begin{bmatrix} 8 & 0 & 0 & 8 \\ 0 & 8 & 8 & 0 \\ 0 & 8 & 8 & 0 \\ 8 & 0 & 0 & 8 \end{bmatrix}$$

Euler Paths

■ Königsberg seven-bridge problem

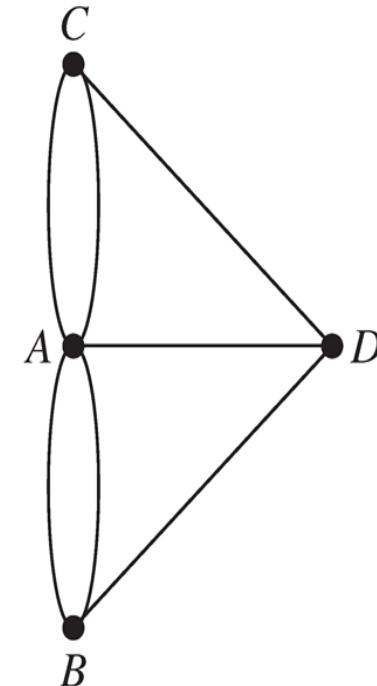
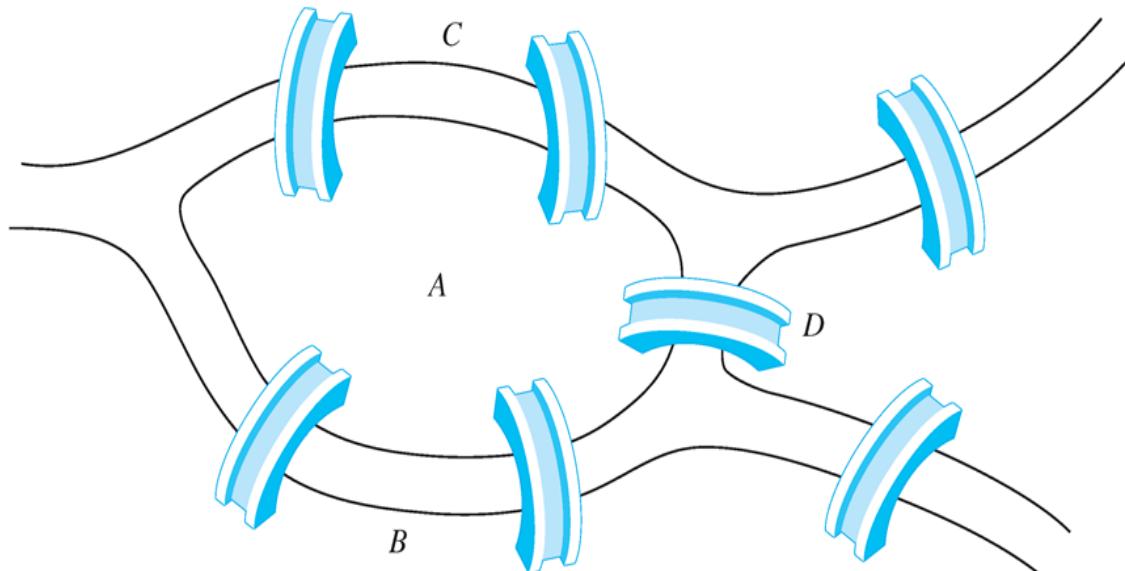
People wondered whether it was possible to start at some location in the town, travel across all the bridges once without crossing any bridge twice, and return to the starting point.



Euler Paths

■ Königsberg seven-bridge problem

People wondered whether it was possible to start at some location in the town, travel across all the bridges once without crossing any bridge twice, and return to the starting point.



Euler Paths and Circuits

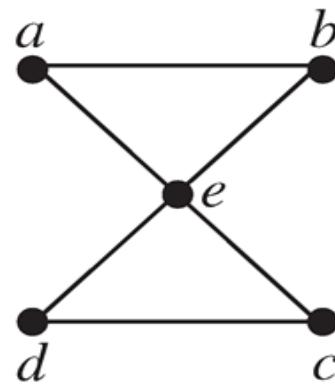
- **Definition** An *Euler circuit* in a graph G is a simple circuit containing every edge of G . An *Euler path* in G is a simple path containing every edge of G .



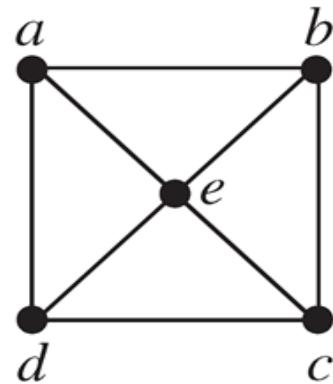
Euler Paths and Circuits

- **Definition** An *Euler circuit* in a graph G is a simple circuit containing every edge of G . An *Euler path* in G is a simple path containing every edge of G .

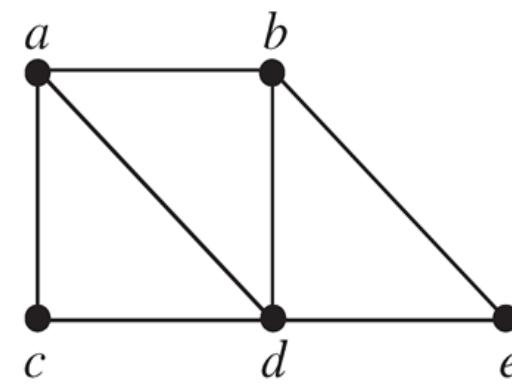
Example Which of the undirected graphs have an Euler circuit? Of those that do not, which have an Euler path?



G_1



G_2

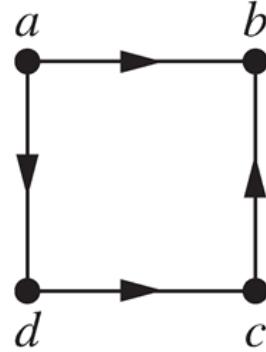


G_3

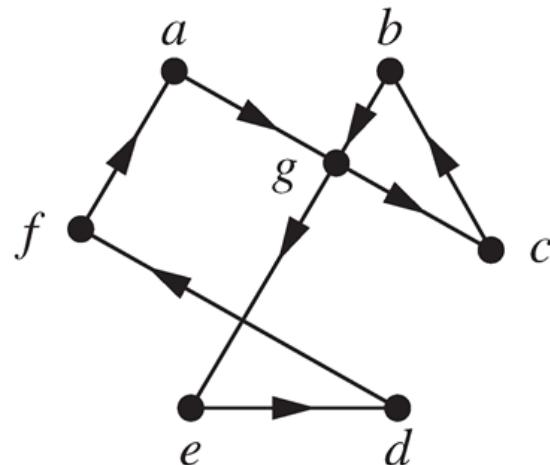
Euler Paths and Circuits

- **Definition** An *Euler circuit* in a graph G is a simple circuit containing every edge of G . An *Euler path* in G is a simple path containing every edge of G .

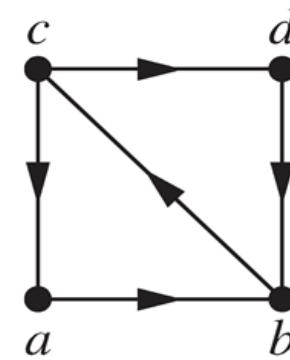
Example Which of the directed graphs have an Euler circuit? Of those that do not, which have an Euler path?



H_1



H_2



H_3

Necessary Conditions for Euler Circuits and Paths

- Euler Circuit \Rightarrow The degree of every vertex must be even



Necessary Conditions for Euler Circuits and Paths

- Euler Circuit \Rightarrow The degree of every vertex must be even
 - ◊ Each time the circuit passes through a vertex, it contributes two to the vertex's degree.

Necessary Conditions for Euler Circuits and Paths

- Euler Circuit \Rightarrow The degree of every vertex must be even
 - ◊ Each time the circuit passes through a vertex, it contributes two to the vertex's degree.
 - ◊ The circuit starts with a vertex a and ends at a , then contributes two to $\deg(a)$.

Necessary Conditions for Euler Circuits and Paths

- Euler Circuit \Rightarrow The degree of every vertex must be even
 - ◊ Each time the circuit passes through a vertex, it contributes two to the vertex's degree.
 - ◊ The circuit starts with a vertex a and ends at a , then contributes two to $\deg(a)$.

Euler Path \Rightarrow The graph has exactly two vertices of odd degree



Necessary Conditions for Euler Circuits and Paths

- Euler Circuit \Rightarrow The degree of every vertex must be even
 - ◊ Each time the circuit passes through a vertex, it contributes two to the vertex's degree.
 - ◊ The circuit starts with a vertex a and ends at a , then contributes two to $\deg(a)$.

Euler Path \Rightarrow The graph has exactly two vertices of odd degree

- ◊ The initial vertex and the final vertex of an Euler path have odd degree.



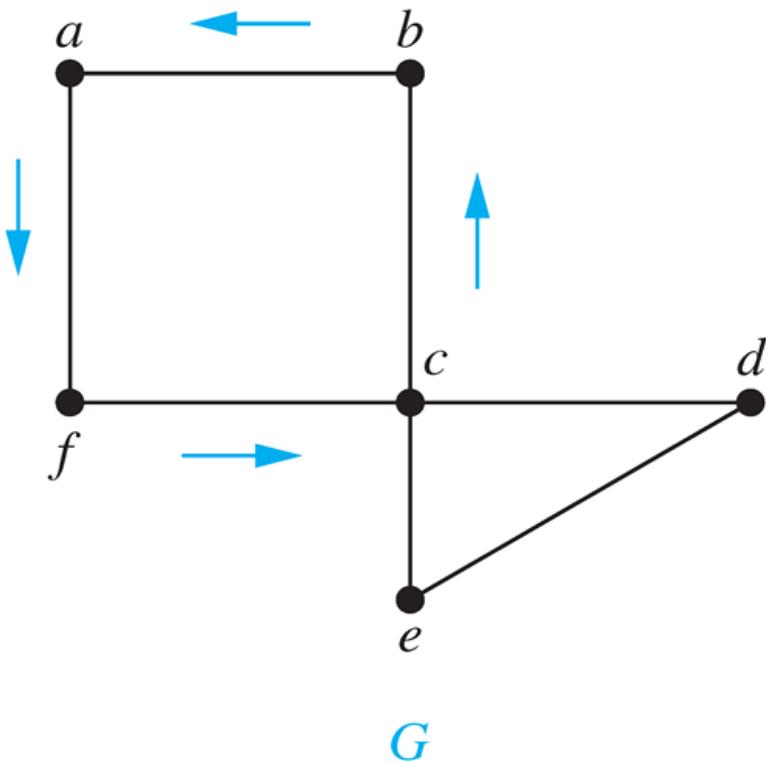
Sufficient Conditions for Euler Circuits and Paths

- Suppose that G is a **connected** multigraph with ≥ 2 vertices, all of **even degree**.



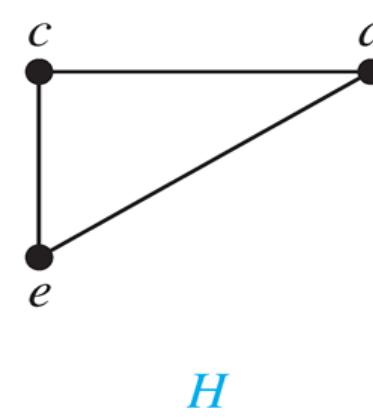
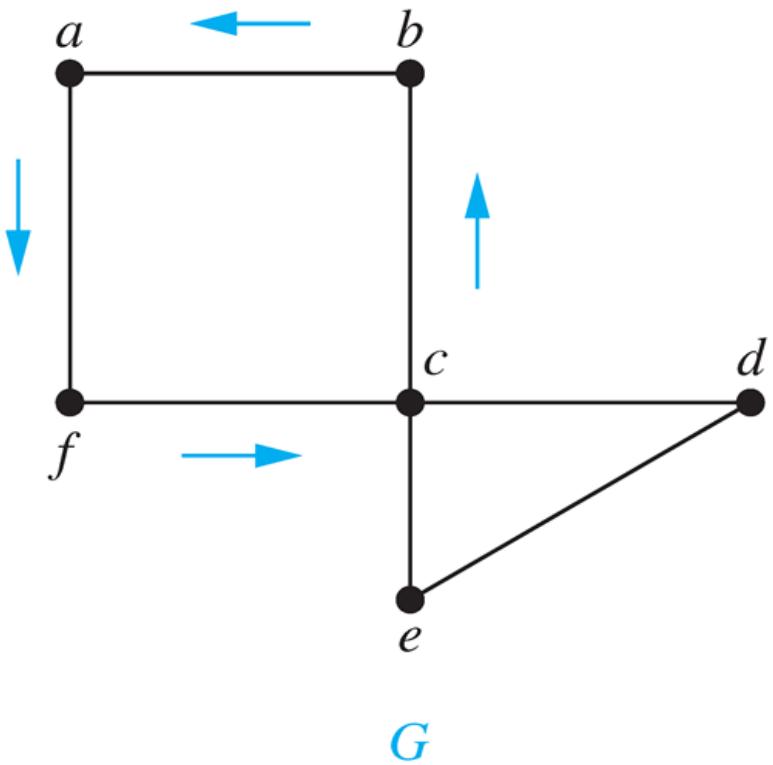
Sufficient Conditions for Euler Circuits and Paths

- Suppose that G is a **connected** multigraph with ≥ 2 vertices, all of **even degree**.



Sufficient Conditions for Euler Circuits and Paths

- Suppose that G is a **connected** multigraph with ≥ 2 vertices, all of **even degree**.



Algorithm for Constructing an Euler Circuit

```
procedure Euler(G: connected multigraph with all vertices of even degree)  
    circuit := a circuit in G beginning at an arbitrarily chosen vertex with edges  
        successively added to form a path that returns to this vertex.  
    H := G with the edges of this circuit removed  
    while H has edges  
        subcircuit := a circuit in H beginning at a vertex in H that also is  
            an endpoint of an edge in circuit.  
        H := H with edges of subcircuit and all isolated vertices removed  
        circuit := circuit with subcircuit inserted at the appropriate vertex.  
return circuit{circuit is an Euler circuit}
```

Algorithm for Constructing an Euler Circuit

```
procedure Euler(G: connected multigraph with all vertices of even degree)
  circuit := a circuit in G beginning at an arbitrarily chosen vertex with edges
    successively added to form a path that returns to this vertex.
  H := G with the edges of this circuit removed
  while H has edges
    subcircuit := a circuit in H beginning at a vertex in H that also is
      an endpoint of an edge in circuit.
    H := H with edges of subcircuit and all isolated vertices removed
    circuit := circuit with subcircuit inserted at the appropriate vertex.
  return circuit{circuit is an Euler circuit}
```

Algorithm for Constructing an Euler Circuit

```
procedure Euler( $G$ : connected multigraph with all vertices of even degree)
   $circuit :=$  a circuit in  $G$  beginning at an arbitrarily chosen vertex with edges
    successively added to form a path that returns to this vertex.
   $H := G$  with the edges of this circuit removed
  while  $H$  has edges
     $subcircuit :=$  a circuit in  $H$  beginning at a vertex in  $H$  that also is
      an endpoint of an edge in circuit.
     $H := H$  with edges of  $subcircuit$  and all isolated vertices removed
     $circuit := circuit$  with  $subcircuit$  inserted at the appropriate vertex.
  return  $circuit$  { $circuit$  is an Euler circuit}
```

Necessary and Sufficient Conditions

- **Theorem** A connected multigraph with at least two vertices has an *Euler circuit* if and only if each of its vertices has even degree.

Necessary and Sufficient Conditions

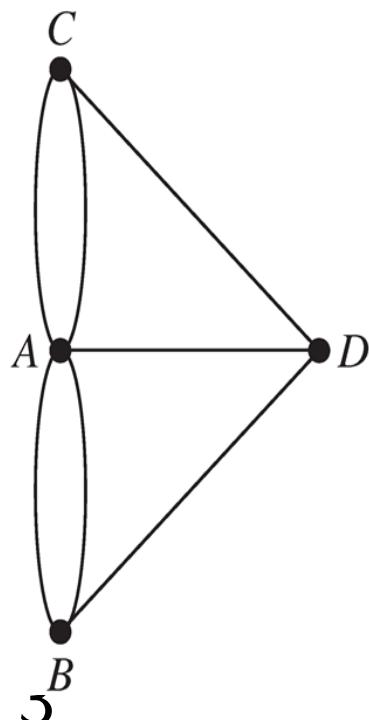
- **Theorem** A connected multigraph with at least two vertices has an *Euler circuit* if and only if each of its vertices has even degree.

Theorem A connected multigraph has an *Euler path* but not an *Euler circuit* if and only if it has exactly two vertices of odd degree.

Necessary and Sufficient Conditions

- **Theorem** A connected multigraph with at least two vertices has an *Euler circuit* if and only if each of its vertices has even degree.

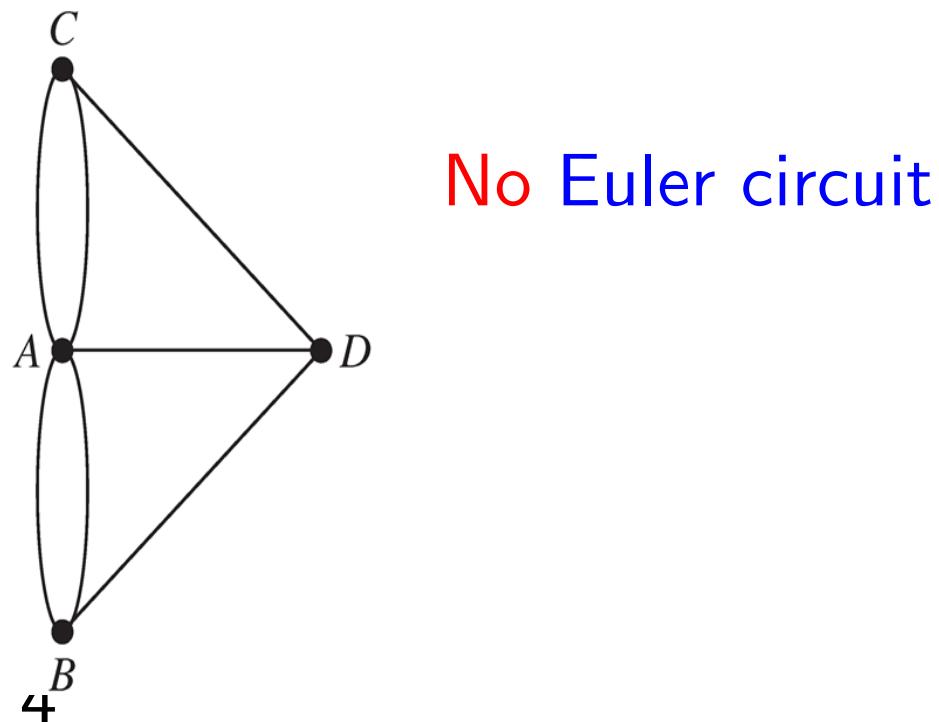
Theorem A connected multigraph has an *Euler path* but not an *Euler circuit* if and only if it has exactly two vertices of odd degree.



Necessary and Sufficient Conditions

- **Theorem** A connected multigraph with at least two vertices has an *Euler circuit* if and only if each of its vertices has even degree.

Theorem A connected multigraph has an *Euler path* but not an *Euler circuit* if and only if it has exactly two vertices of odd degree.



Euler Circuits and Paths

■ Example

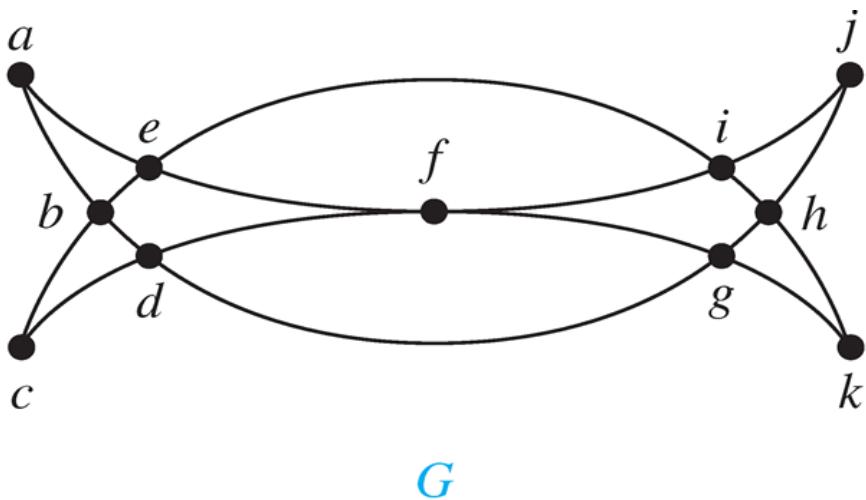
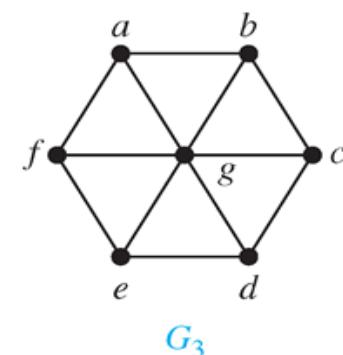
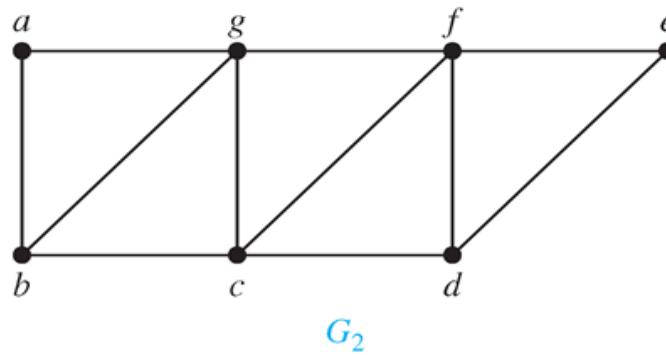
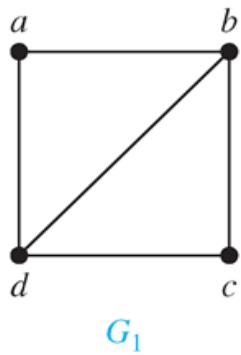


FIGURE 6 Mohammed's Scimitars.

Euler Circuits and Paths

■ Example



Applications of Euler Paths and Circuits

- Finding a path or circuit that traverses each
 - ◊ street in a neighborhood
 - ◊ road in a transportation network
 - ◊ link in a communication network
 - ◊ ...

Applications of Euler Paths and Circuits

- Finding a path or circuit that traverses each
 - ◊ street in a neighborhood
 - ◊ road in a transportation network
 - ◊ link in a communication network
 - ◊ ...

Chinese Postman Problem

Meigu Guan [60']

Applications of Euler Paths and Circuits

- Finding a path or circuit that traverses each
 - ◊ street in a neighborhood
 - ◊ road in a transportation network
 - ◊ link in a communication network
 - ◊ ...

Chinese Postman Problem

Meigu Guan [60']

Given a graph $G = (V, E)$, for every $e \in E$, there is a nonnegative weight $w(e)$. Find a **circuit** W such that

$$\sum_{e \in W} w(e) = \min$$

Applications of Euler Paths and Circuits

- Finding a path or circuit that traverses each
 - ◊ street in a neighborhood
 - ◊ road in a transportation network
 - ◊ link in a communication network
 - ◊ ...

Chinese Postman Problem Meigu Guan [60']

Given a graph $G = (V, E)$, for every $e \in E$, there is a nonnegative weight $w(e)$. Find a *circuit* W such that

$$\sum_{e \in W} w(e) = \min$$

k-Postman Chinese Postman Problem (k-PCPP)

Applications of Euler Paths and Circuits

- Finding a path or circuit that traverses each
 - ◊ street in a neighborhood
 - ◊ road in a transportation network
 - ◊ link in a communication network
 - ◊ ...

Chinese Postman Problem

Meigu Guan [60']

Given a graph $G = (V, E)$, for every $e \in E$, there is a nonnegative weight $w(e)$. Find a *circuit* W such that

$$\sum_{e \in W} w(e) = \min$$

k-Postman Chinese Postman Problem (k-PCPP)

16 - 5 \in NPC

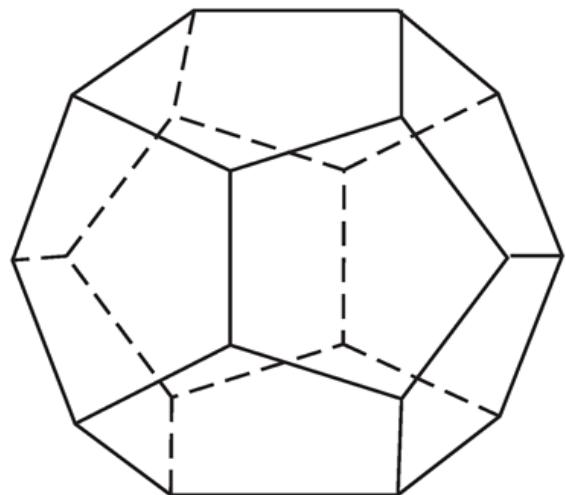


Hamilton Paths and Circuits

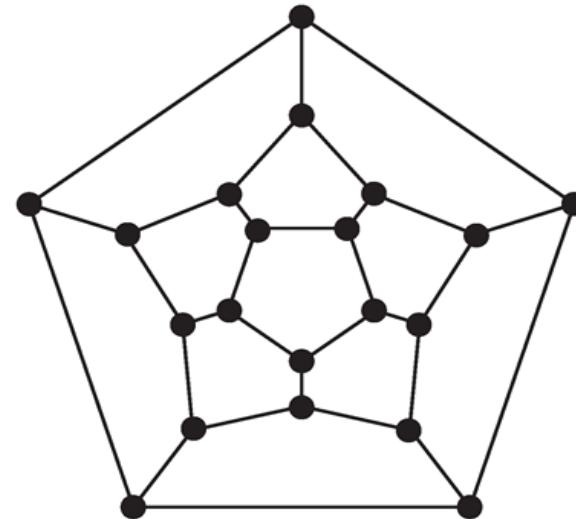
- Euler paths and circuits contained every edge only once.
What about containing every vertex exactly once?

Hamilton Paths and Circuits

- Euler paths and circuits contained every edge only once.
What about containing every vertex exactly once?



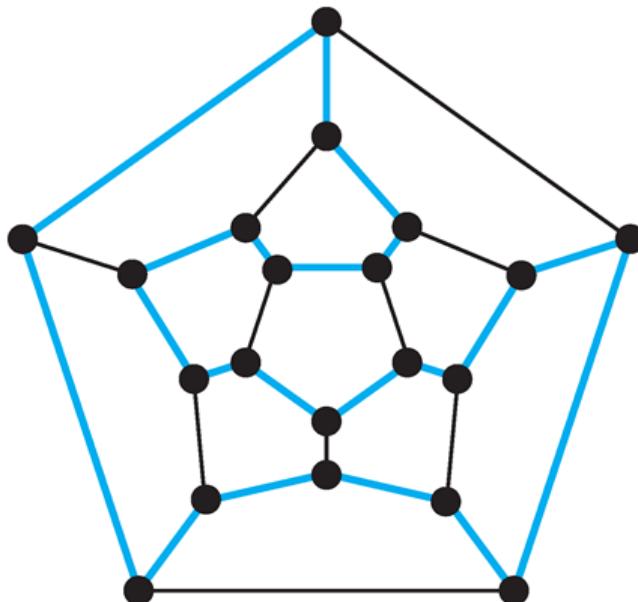
(a)



(b)

Hamilton Paths and Circuits

- Euler paths and circuits contained every edge only once.
What about containing every vertex exactly once?



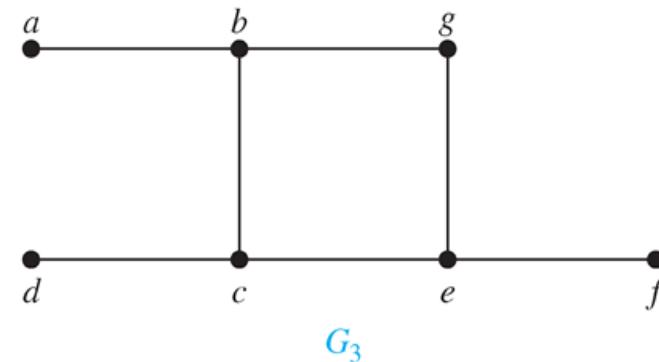
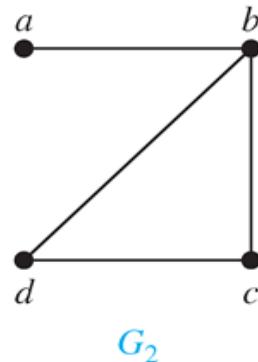
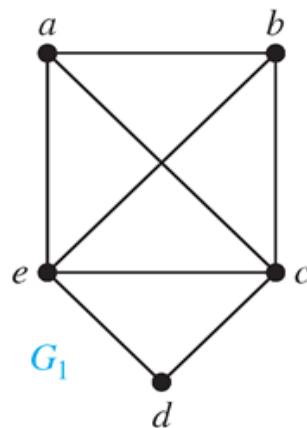
Hamilton Paths and Circuits

- **Definition:** A simple path in a graph G that passes through every vertex exactly once is called a *Hamilton path*, and a simple circuit in a graph G that passes through every vertex exactly once is called a *Hamilton circuit*.

Hamilton Paths and Circuits

- **Definition:** A simple path in a graph G that passes through every vertex exactly once is called a *Hamilton path*, and a simple circuit in a graph G that passes through every vertex exactly once is called a *Hamilton circuit*.

Example Which of these simple graphs has a Hamilton circuit or, if not, a Hamilton path?



Sufficient Conditions for Hamilton Circuits

- No simple necessary and sufficient conditions are known for the existence of a Hamilton circuit.



Sufficient Conditions for Hamilton Circuits

- No simple necessary and sufficient conditions are known for the existence of a Hamilton circuit.

But, there are some useful sufficient conditions.



Sufficient Conditions for Hamilton Circuits

- No simple necessary and sufficient conditions are known for the existence of a Hamilton circuit.

But, there are some useful sufficient conditions.

Dirac's Theorem If G is a simple graph with $n \geq 3$ vertices such that the degree of every vertex in G is $\geq n/2$, then G has a Hamilton circuit.

Sufficient Conditions for Hamilton Circuits

- No simple necessary and sufficient conditions are known for the existence of a Hamilton circuit.

But, there are some useful sufficient conditions.

Dirac's Theorem If G is a simple graph with $n \geq 3$ vertices such that the degree of every vertex in G is $\geq n/2$, then G has a Hamilton circuit.

Ore's Theorem If G is a simple graph with $n \geq 3$ vertices such that $\deg(u) + \deg(v) \geq n$ for every pair of nonadjacent vertices, then G has a Hamilton circuit.

Sufficient Conditions for Hamilton Circuits

- No simple necessary and sufficient conditions are known for the existence of a Hamilton circuit.

But, there are some useful sufficient conditions.

Dirac's Theorem If G is a simple graph with $n \geq 3$ vertices such that the degree of every vertex in G is $\geq n/2$, then G has a Hamilton circuit.

Ore's Theorem If G is a simple graph with $n \geq 3$ vertices such that $\deg(u) + \deg(v) \geq n$ for every pair of nonadjacent vertices, then G has a Hamilton circuit.

Hamilton path problem \in NPC

Applications of Hamilton Paths and Circuits

- A path or a circuit that visits each city, or each node in a communication network **exactly once**, can be solved by finding a **Hamilton path**.



Applications of Hamilton Paths and Circuits

- A path or a circuit that visits each city, or each node in a communication network **exactly once**, can be solved by finding a **Hamilton path**.

Traveling Salesperson Problem (TSP) asks for the **shortest route** a traveling salesperson should take to visit a set of cities.



Applications of Hamilton Paths and Circuits

- A path or a circuit that visits each city, or each node in a communication network **exactly once**, can be solved by finding a **Hamilton path**.

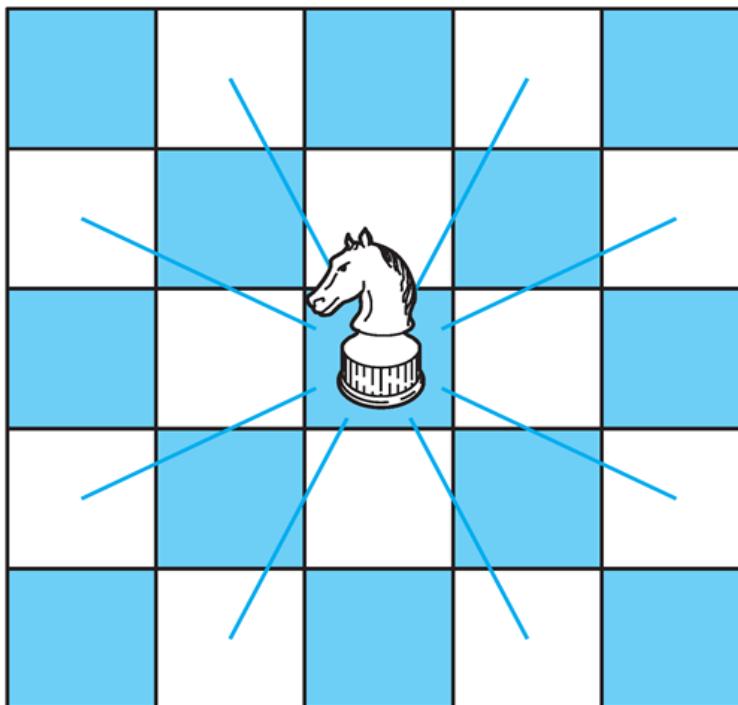
Traveling Salesperson Problem (TSP) asks for the **shortest route** a traveling salesperson should take to visit a set of cities.

the decision version of the TSP \in NPC



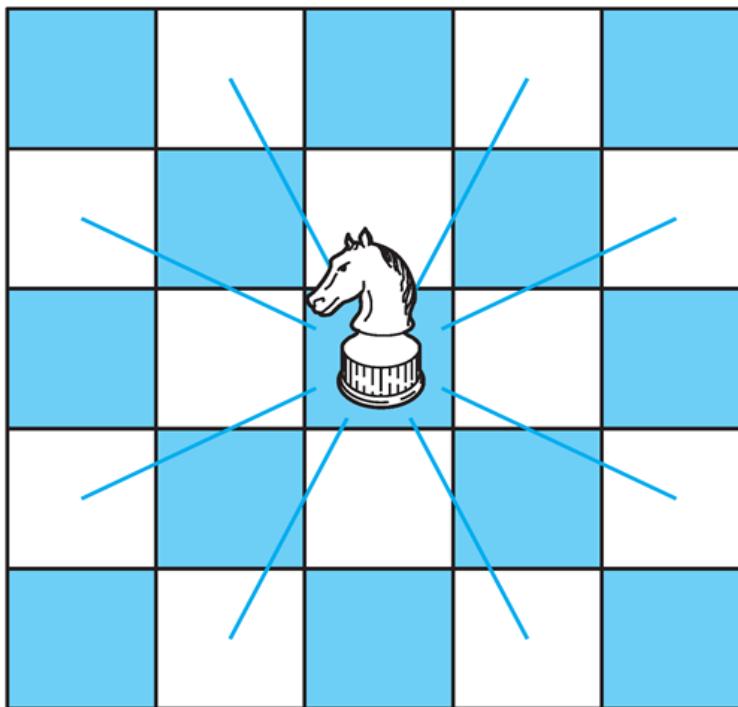
Applications of Hamilton Paths and Circuits

- Can we traverse every space (and come back) in the 5×5 chessboard?



Applications of Hamilton Paths and Circuits

- Can we traverse every space (and come back) in the 5×5 chessboard?



What about in 6×6 chessboard?

Shortest Path Problems

- Using graphs with **weights** assigned to their edges

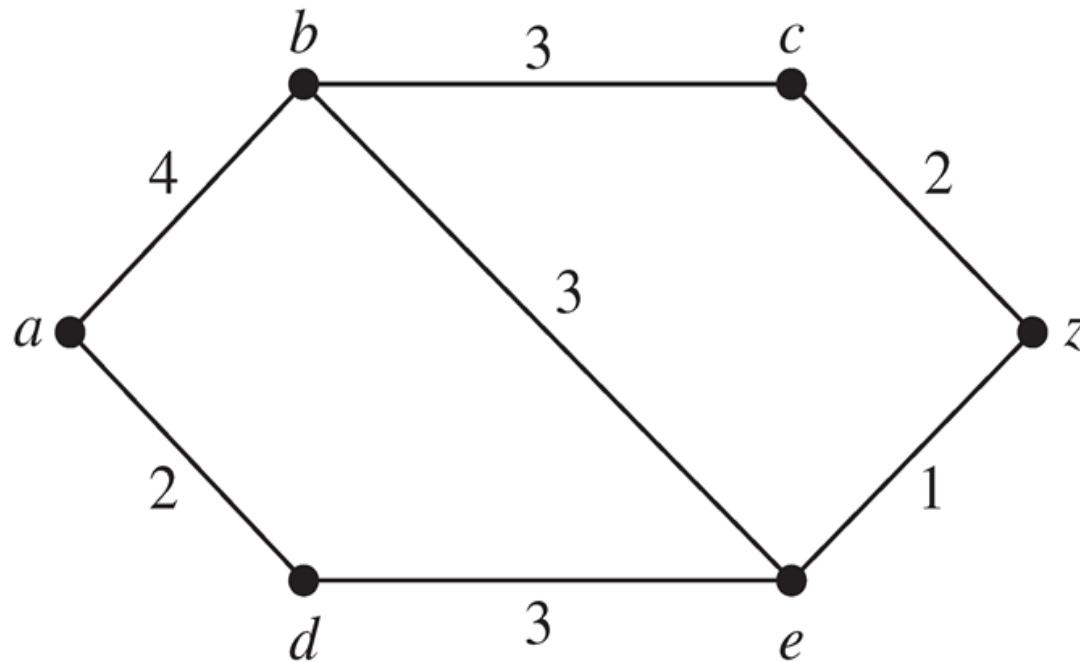
Shortest Path Problems

- Using graphs with **weights** assigned to their edges
Such graphs are called *weighted graphs* and can model lots of questions involving **distance**, time consuming, **fares**, etc.

Shortest Path Problems

- Using graphs with **weights** assigned to their edges

Such graphs are called ***weighted graphs*** and can model lots of questions involving **distance, time consuming, fares, etc.**



Shortest Path Problems

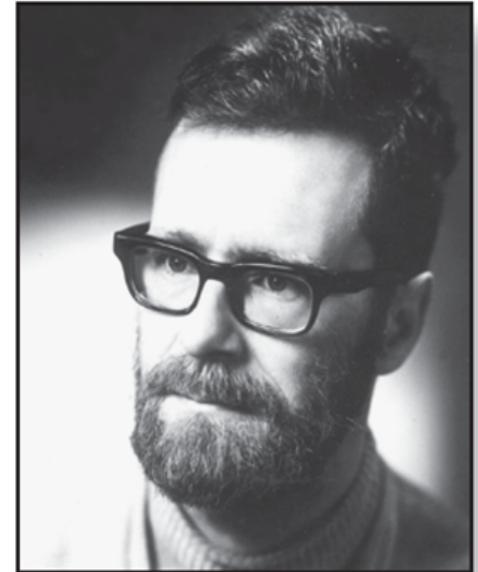
- **Definition** Let G^α be an weighted graph, with a weight function $\alpha : E \rightarrow \mathbb{R}^+$ on its edges. If $P = e_1 e_2 \cdots e_k$ is a path, then its weight is $\alpha(P) = \sum_{i=1}^k \alpha(e_i)$. The *minimum weighted distance* between two vertices is

$$d(u, v) = \min\{\alpha(P) | P : u \rightarrow v\}$$

Shortest Path Problems

- **Definition** Let G^α be an **weighted graph**, with a **weight function** $\alpha : E \rightarrow \mathbb{R}^+$ on its edges. If $P = e_1 e_2 \cdots e_k$ is a path, then its weight is $\alpha(P) = \sum_{i=1}^k \alpha(e_i)$. The **minimum weighted distance** between two vertices is

$$d(u, v) = \min\{\alpha(P) | P : u \rightarrow v\}$$



Edsger Wybe Dijkstra

Dijkstra's Algorithm

- (i) Set $d(v_0) = 0$ and $d(v) = \infty$ for all $v \neq v_0$, $S = \emptyset$

Dijkstra's Algorithm

- (i) Set $d(v_0) = 0$ and $d(v) = \infty$ for all $v \neq v_0$, $S = \emptyset$
- (ii) while $S \neq V$
 - let $v \notin S$ be the vertex with the least value $d(v)$,
 - $S = S \cup \{v\}$ for each $u \notin S$, replace $d(u)$ by $\min\{d(u), d(v) + \alpha(u, v)\}$

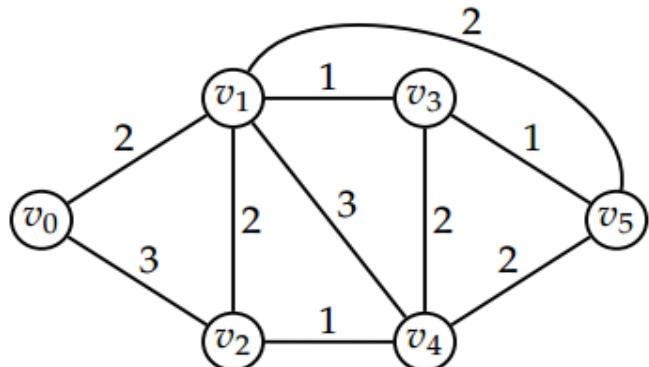
Dijkstra's Algorithm

- (i) Set $d(v_0) = 0$ and $d(v) = \infty$ for all $v \neq v_0$, $S = \emptyset$
- (ii) while $S \neq V$
 let $v \notin S$ be the vertex with the least value $d(v)$,
 $S = S \cup \{v\}$ for each $u \notin S$, replace $d(u)$ by
 $\min\{d(u), d(v) + \alpha(u, v)\}$
- (iii) return all $d(v)$'s

Dijkstra's Algorithm

- (i) Set $d(v_0) = 0$ and $d(v) = \infty$ for all $v \neq v_0$, $S = \emptyset$
- (ii) while $S \neq V$
 - let $v \notin S$ be the vertex with the least value $d(v)$,
 - $S = S \cup \{v\}$ for each $u \notin S$, replace $d(u)$ by $\min\{d(u), d(v) + \alpha(u, v)\}$
- (iii) return all $d(v)$'s

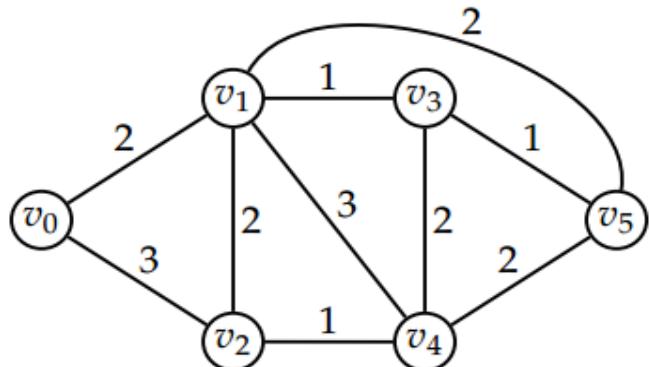
Example



Dijkstra's Algorithm

- (i) Set $d(v_0) = 0$ and $d(v) = \infty$ for all $v \neq v_0$, $S = \emptyset$
- (ii) while $S \neq V$
 - let $v \notin S$ be the vertex with the least value $d(v)$,
 - $S = S \cup \{v\}$ for each $u \notin S$, replace $d(u)$ by $\min\{d(u), d(v) + \alpha(u, v)\}$
- (iii) return all $d(v)$'s

Example

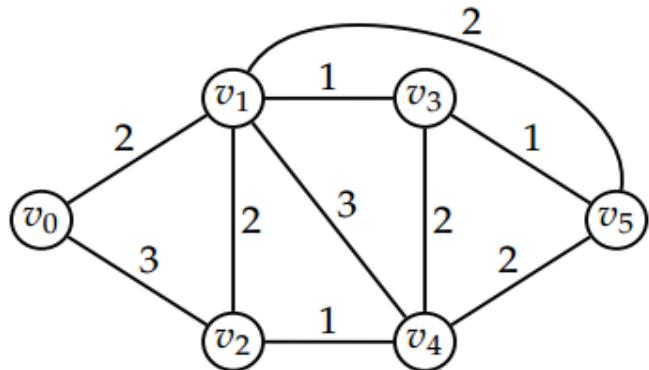


$$d(v_0) = 0, \text{ all other } d(v) = \infty$$

Dijkstra's Algorithm

- (i) Set $d(v_0) = 0$ and $d(v) = \infty$ for all $v \neq v_0$, $S = \emptyset$
- (ii) while $S \neq V$
 - let $v \notin S$ be the vertex with the least value $d(v)$,
 - $S = S \cup \{v\}$ for each $u \notin S$, replace $d(u)$ by $\min\{d(u), d(v) + \alpha(u, v)\}$
- (iii) return all $d(v)$'s

Example



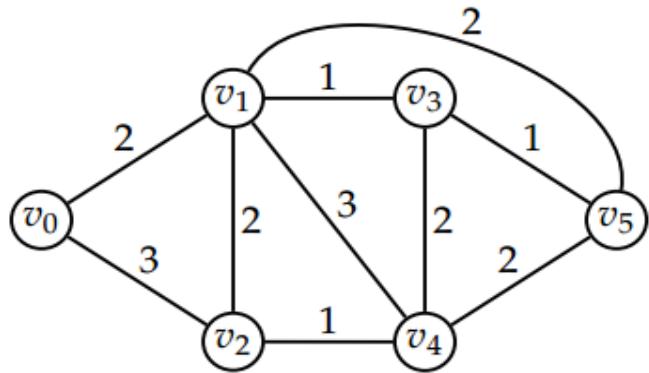
v_0	v_1	v_2	v_3	v_4	v_5
0	∞	∞	∞	∞	∞

$$d(v_0) = 0, \text{ all other } d(v) = \infty$$

Dijkstra's Algorithm

- (i) Set $d(v_0) = 0$ and $d(v) = \infty$ for all $v \neq v_0$, $S = \emptyset$
- (ii) while $S \neq V$
 - let $v \notin S$ be the vertex with the least value $d(v)$,
 - $S = S \cup \{v\}$ for each $u \notin S$, replace $d(u)$ by $\min\{d(u), d(v) + \alpha(u, v)\}$
- (iii) return all $d(v)$'s

Example



v_0	v_1	v_2	v_3	v_4	v_5
0	∞	∞	∞	∞	∞

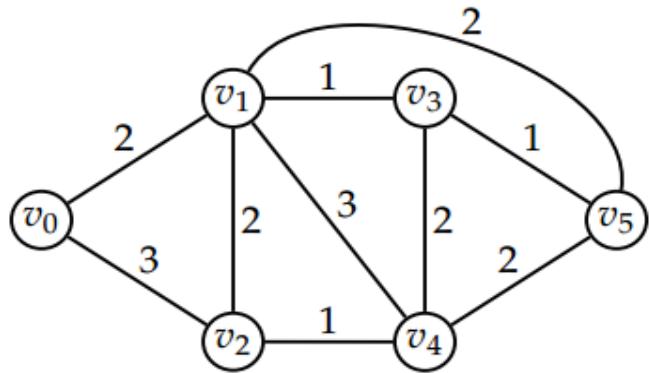
$$i = 0$$

$$d(v_1) = \min\{\infty, 2\} = 2, d(v_2) = \min\{\infty, 3\} = 3$$

Dijkstra's Algorithm

- (i) Set $d(v_0) = 0$ and $d(v) = \infty$ for all $v \neq v_0$, $S = \emptyset$
- (ii) while $S \neq V$
 - let $v \notin S$ be the vertex with the least value $d(v)$,
 - $S = S \cup \{v\}$ for each $u \notin S$, replace $d(u)$ by $\min\{d(u), d(v) + \alpha(u, v)\}$
- (iii) return all $d(v)$'s

Example



v_0	v_1	v_2	v_3	v_4	v_5
0	2	3	∞	∞	∞

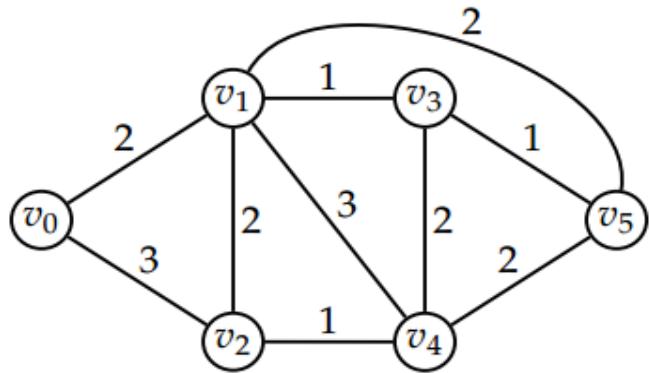
$$i = 0$$

$$d(v_1) = \min\{\infty, 2\} = 2, d(v_2) = \min\{\infty, 3\} = 3$$

Dijkstra's Algorithm

- (i) Set $d(v_0) = 0$ and $d(v) = \infty$ for all $v \neq v_0$, $S = \emptyset$
- (ii) while $S \neq V$
 - let $v \notin S$ be the vertex with the least value $d(v)$,
 - $S = S \cup \{v\}$ for each $u \notin S$, replace $d(u)$ by $\min\{d(u), d(v) + \alpha(u, v)\}$
- (iii) return all $d(v)$'s

Example



v_0	v_1	v_2	v_3	v_4	v_5
0	2	3	∞	∞	∞

$$i = 1$$

$$d(v_2) = \min\{3, d(v_1) + \alpha(v_1 v_2)\} = \min\{3, 4\} = 3,$$

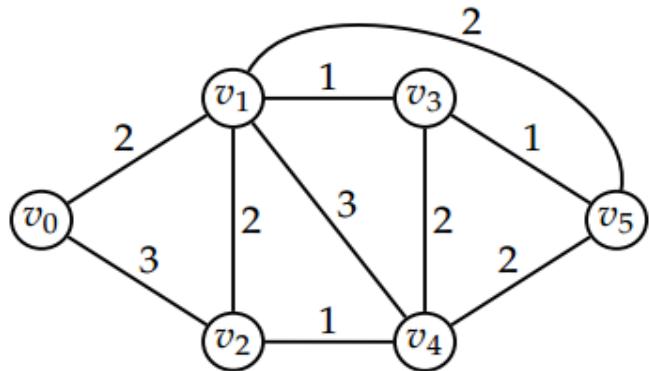
$$d(v_3) = 2 + 1 = 3, d(v_4) = 2 + 3 = 5,$$

$$d(v_5) = 2 + 2 = 4$$

Dijkstra's Algorithm

- (i) Set $d(v_0) = 0$ and $d(v) = \infty$ for all $v \neq v_0$, $S = \emptyset$
- (ii) while $S \neq V$
 - let $v \notin S$ be the vertex with the least value $d(v)$,
 - $S = S \cup \{v\}$ for each $u \notin S$, replace $d(u)$ by $\min\{d(u), d(v) + \alpha(u, v)\}$
- (iii) return all $d(v)$'s

Example



v_0	v_1	v_2	v_3	v_4	v_5
0	2	3	3	5	4

$$i = 1$$

$$d(v_2) = \min\{3, d(v_1) + \alpha(v_1 v_2)\} = \min\{3, 4\} = 3,$$

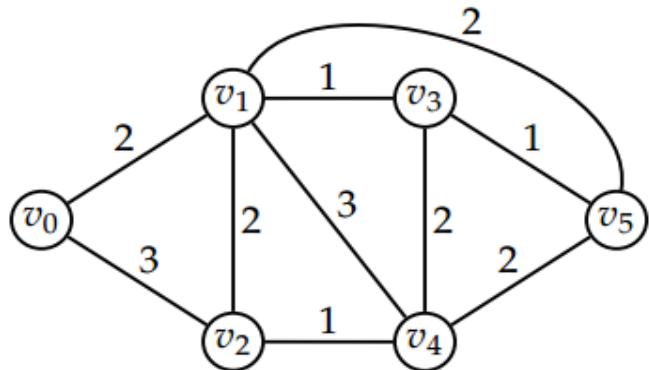
$$d(v_3) = 2 + 1 = 3, d(v_4) = 2 + 3 = 5,$$

$$d(v_5) = 2 + 2 = 4$$

Dijkstra's Algorithm

- (i) Set $d(v_0) = 0$ and $d(v) = \infty$ for all $v \neq v_0$, $S = \emptyset$
- (ii) while $S \neq V$
 - let $v \notin S$ be the vertex with the least value $d(v)$,
 - $S = S \cup \{v\}$ for each $u \notin S$, replace $d(u)$ by $\min\{d(u), d(v) + \alpha(u, v)\}$
- (iii) return all $d(v)$'s

Example



v_0	v_1	v_2	v_3	v_4	v_5
0	2	3	3	5	4

$$i = 2$$

$$d(v_3) = \min\{3, \infty\} = 3,$$

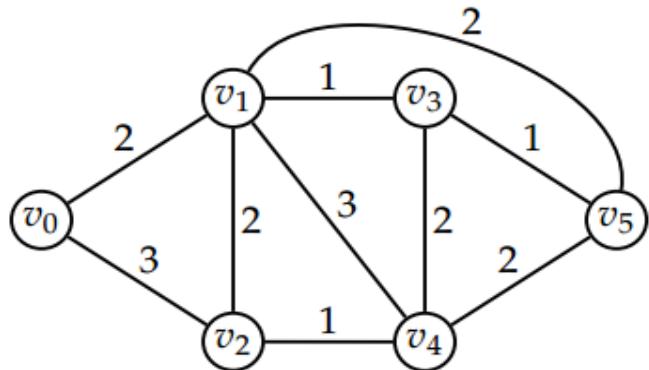
$$d(v_4) = \min\{5, 3 + 1\} = 4,$$

$$d(v_5) = \min\{4, \infty\} = 4$$

Dijkstra's Algorithm

- (i) Set $d(v_0) = 0$ and $d(v) = \infty$ for all $v \neq v_0$, $S = \emptyset$
- (ii) while $S \neq V$
 - let $v \notin S$ be the vertex with the least value $d(v)$,
 - $S = S \cup \{v\}$ for each $u \notin S$, replace $d(u)$ by $\min\{d(u), d(v) + \alpha(u, v)\}$
- (iii) return all $d(v)$'s

Example



v_0	v_1	v_2	v_3	v_4	v_5
0	2	3	3	4	4

$$i = 2$$

$$d(v_3) = \min\{3, \infty\} = 3,$$

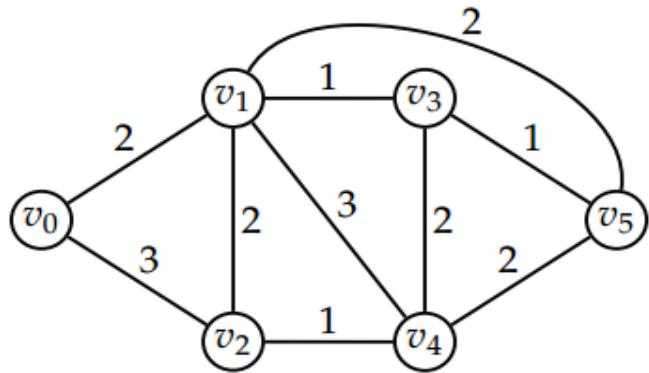
$$d(v_4) = \min\{5, 3 + 1\} = 4,$$

$$d(v_5) = \min\{4, \infty\} = 4$$

Dijkstra's Algorithm

- (i) Set $d(v_0) = 0$ and $d(v) = \infty$ for all $v \neq v_0$, $S = \emptyset$
- (ii) while $S \neq V$
 - let $v \notin S$ be the vertex with the least value $d(v)$,
 - $S = S \cup \{v\}$ for each $u \notin S$, replace $d(u)$ by $\min\{d(u), d(v) + \alpha(u, v)\}$
- (iii) return all $d(v)$'s

Example



v_0	v_1	v_2	v_3	v_4	v_5
0	2	3	3	4	4

$$i = 3$$

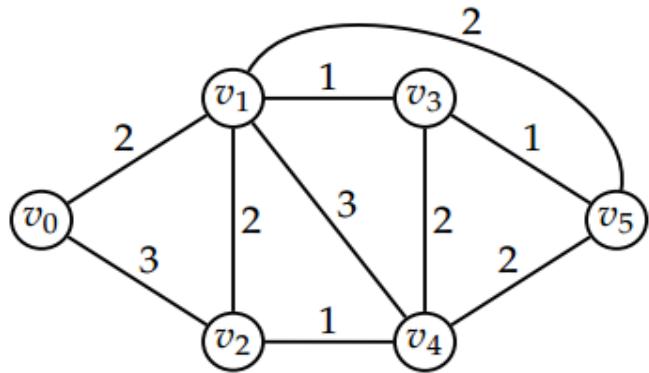
$$d(v_4) = \min\{4, 3 + 2\} = 4,$$

$$d(v_5) = \min\{4, 3 + 1\} = 4$$

Dijkstra's Algorithm

- (i) Set $d(v_0) = 0$ and $d(v) = \infty$ for all $v \neq v_0$, $S = \emptyset$
- (ii) while $S \neq V$
 - let $v \notin S$ be the vertex with the least value $d(v)$,
 - $S = S \cup \{v\}$ for each $u \notin S$, replace $d(u)$ by $\min\{d(u), d(v) + \alpha(u, v)\}$
- (iii) return all $d(v)$'s

Example



v_0	v_1	v_2	v_3	v_4	v_5
0	2	3	3	4	4

$$i = 4$$

$$d(v_5) = \min\{4, 4 + 2\} = 4$$

Dijkstra's Algorithm

- **Theorem** Dijkstra's algorithm finds the length of a shortest path between two vertices in a connected simple undirected weighted graph.

Dijkstra's Algorithm

- **Theorem** Dijkstra's algorithm finds the length of a shortest path between two vertices in a connected simple undirected weighted graph.

Correctness

Dijkstra's Algorithm

- **Theorem** Dijkstra's algorithm finds the length of a shortest path between two vertices in a connected simple undirected weighted graph.

Correctness

Theorem Dijkstra's algorithm uses $O(n^2)$ operations (additions and comparisons) in a connected simple undirected weighted graph with n vertices.

Dijkstra's Algorithm

- **Theorem** Dijkstra's algorithm finds the length of a shortest path between two vertices in a connected simple undirected weighted graph.

Correctness

Theorem Dijkstra's algorithm uses $O(n^2)$ operations (additions and comparisons) in a connected simple undirected weighted graph with n vertices.

Complexity

Dijkstra's Algorithm

- **Theorem** Dijkstra's algorithm finds the length of a shortest path between two vertices in a connected simple undirected weighted graph.

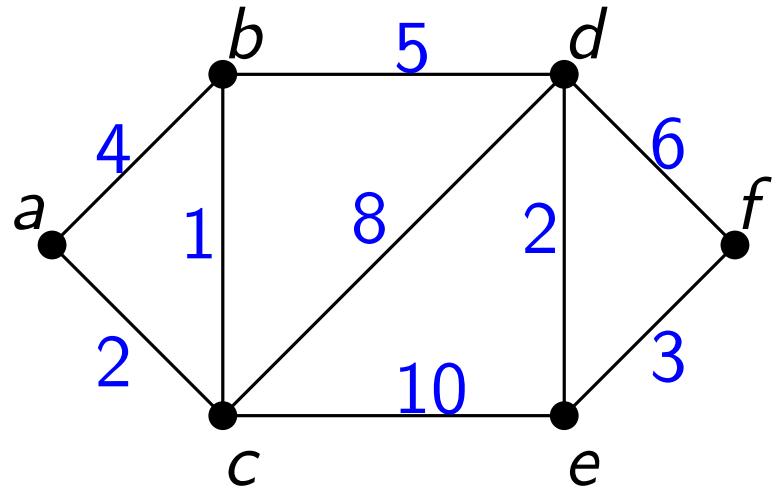
Correctness

Theorem Dijkstra's algorithm uses $O(n^2)$ operations (additions and comparisons) in a connected simple undirected weighted graph with n vertices.

Complexity

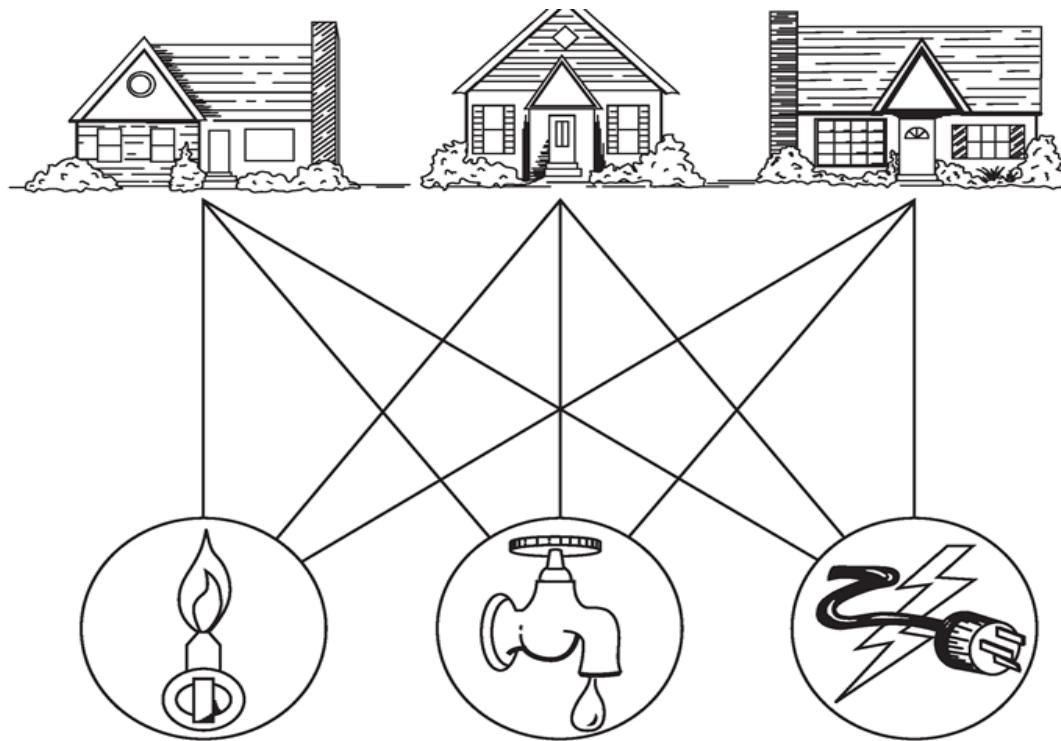
read the Textbook p.712 – p.714

Another Example



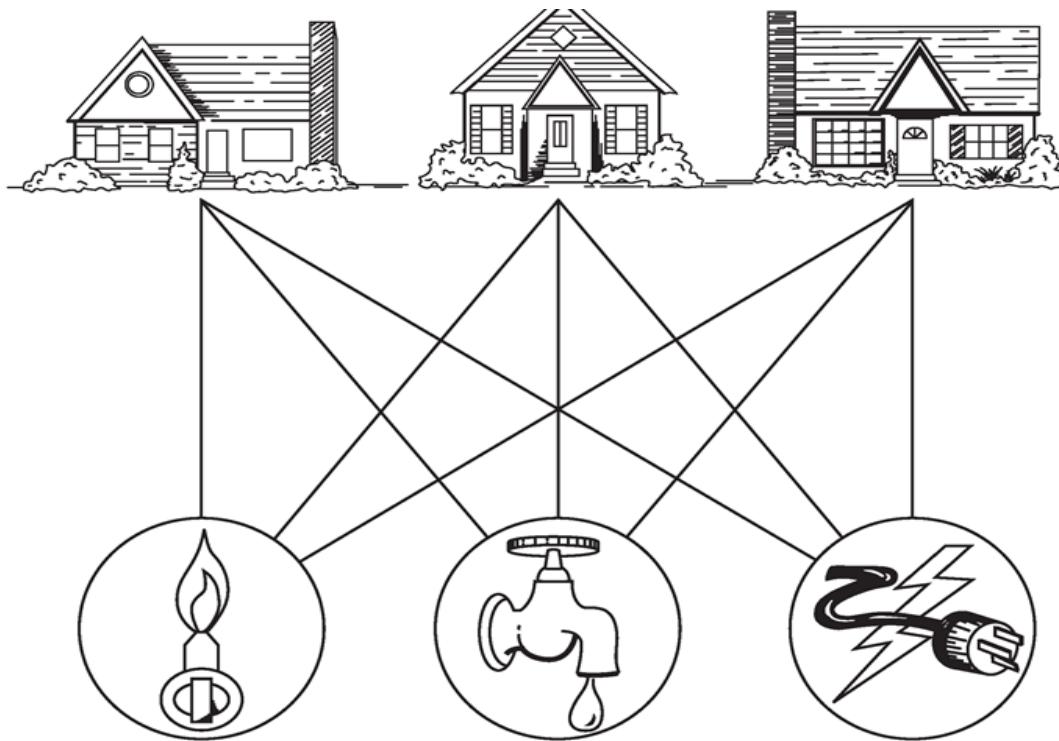
Planar Graphs

- Join three houses to each of three separate utilities.



Planar Graphs

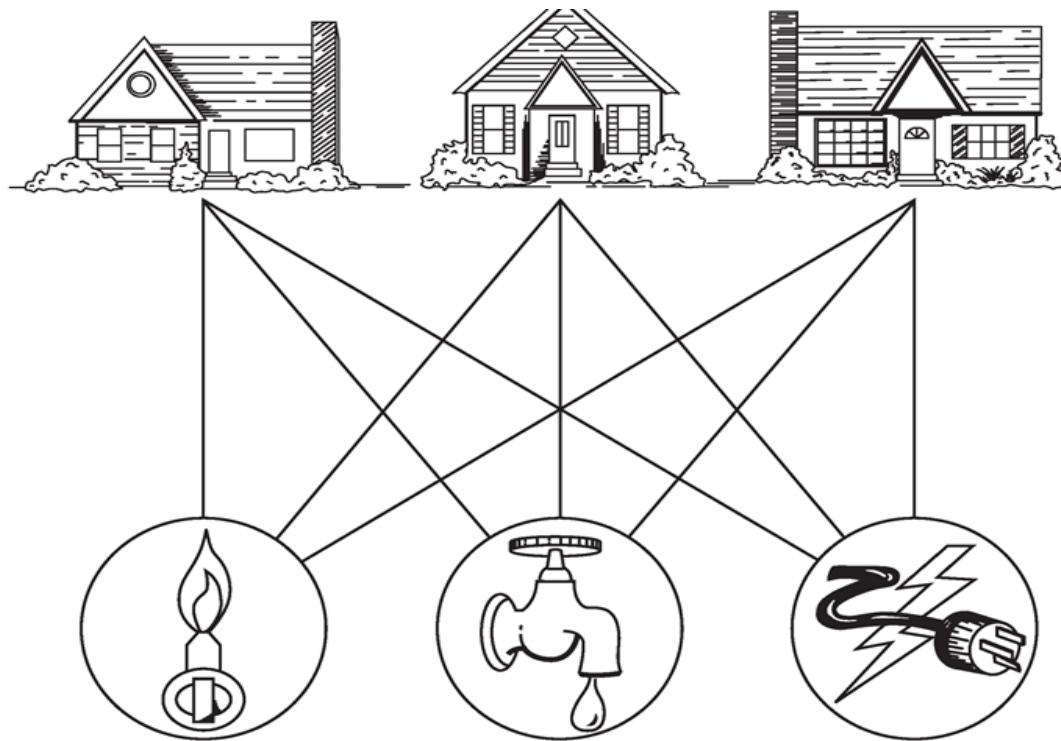
- Join three houses to each of three separate utilities.



Can this graph be drawn **in the plane** s.t. no two of its edges cross?

Planar Graphs

- Join three houses to each of three separate utilities.



Can this graph be drawn **in the plane** s.t. no two of its
edges cross? $K_{3,3}$

Planar Graphs

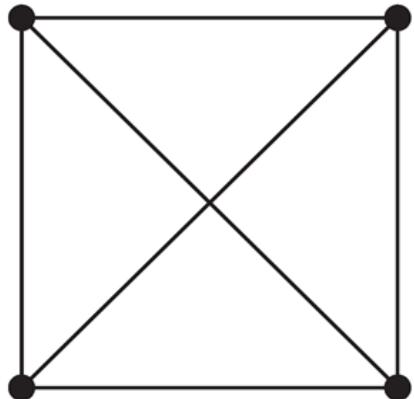
- **Definition** A graph is called *planar* if it can be drawn in the plane **without any edges crossing**. Such a drawing is called a *planar representation* of the graph.



Planar Graphs

- **Definition** A graph is called *planar* if it can be drawn in the plane **without any edges crossing**. Such a drawing is called a *planar representation* of the graph.

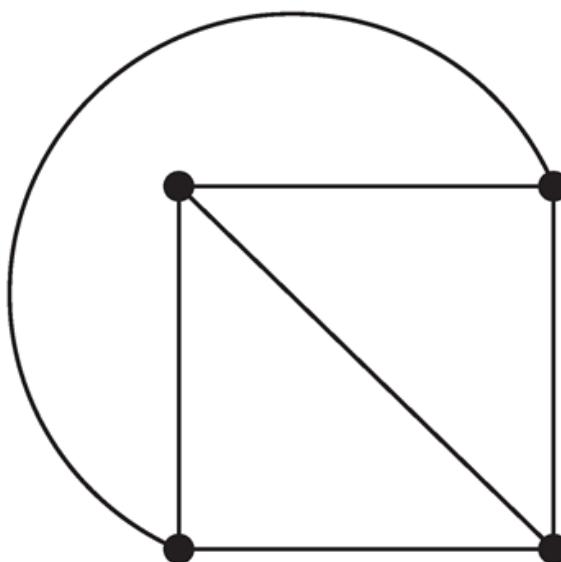
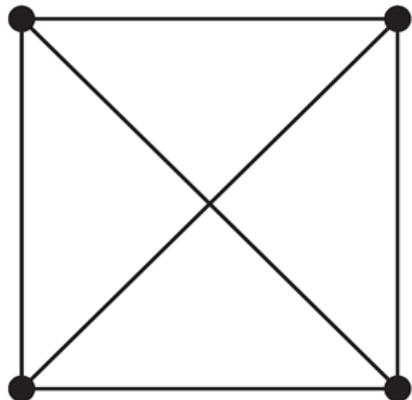
Example Is K_4 planar?



Planar Graphs

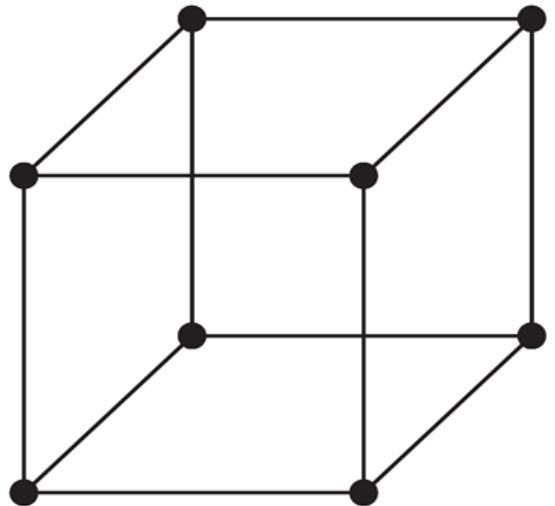
- **Definition** A graph is called *planar* if it can be drawn in the plane **without any edges crossing**. Such a drawing is called a *planar representation* of the graph.

Example Is K_4 planar?



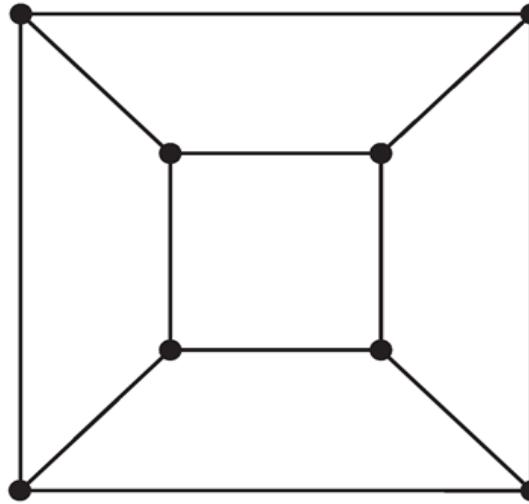
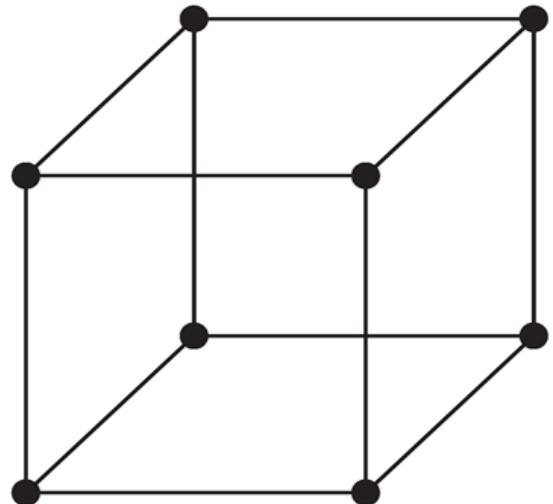
Planar Graphs

■ Example



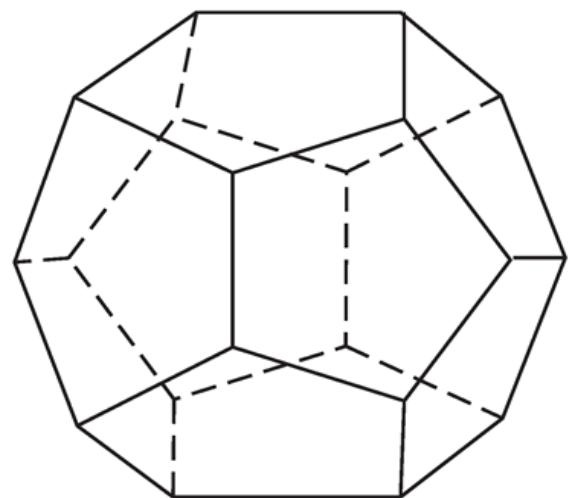
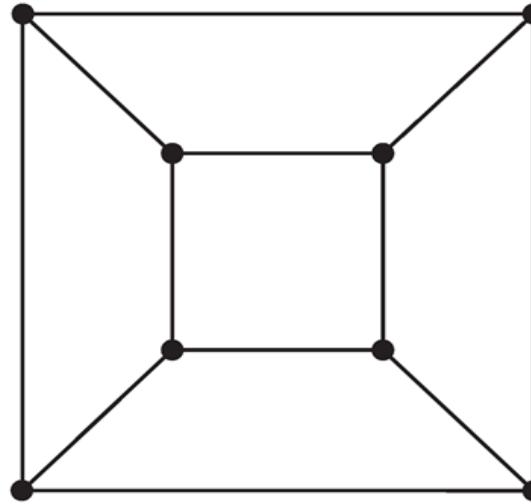
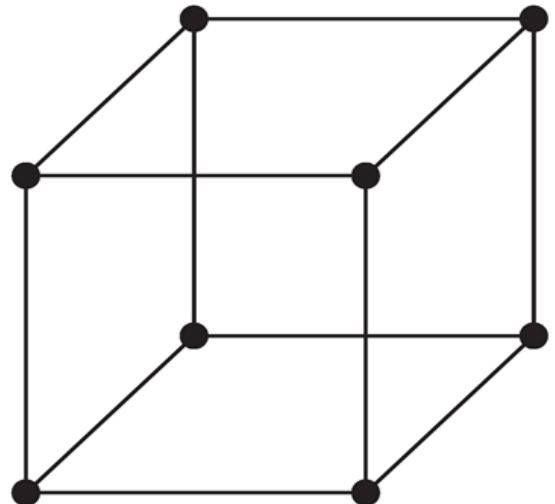
Planar Graphs

■ Example



Planar Graphs

■ Example

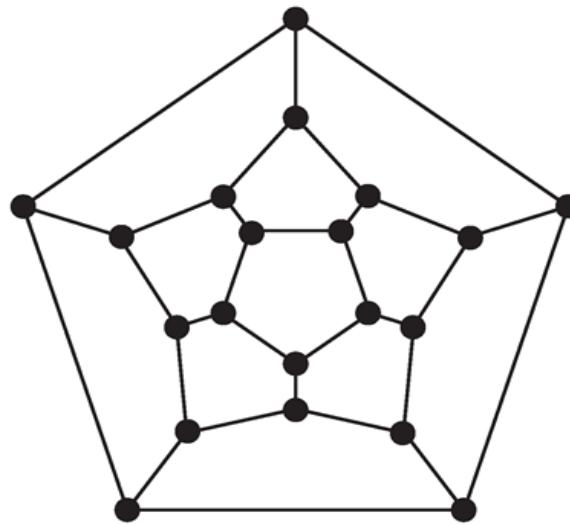
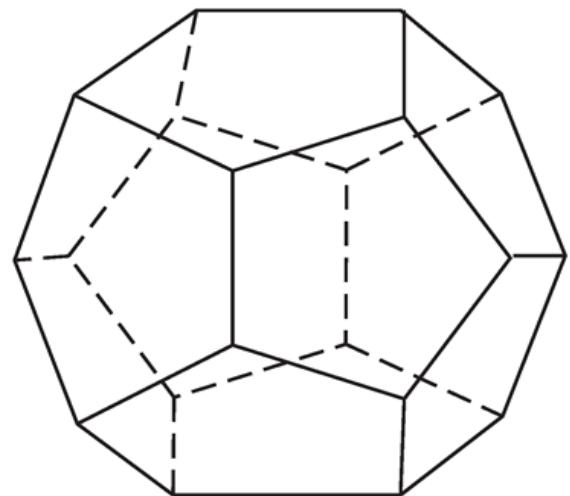
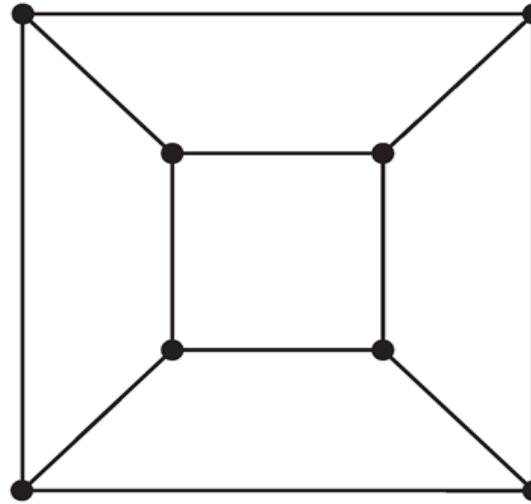
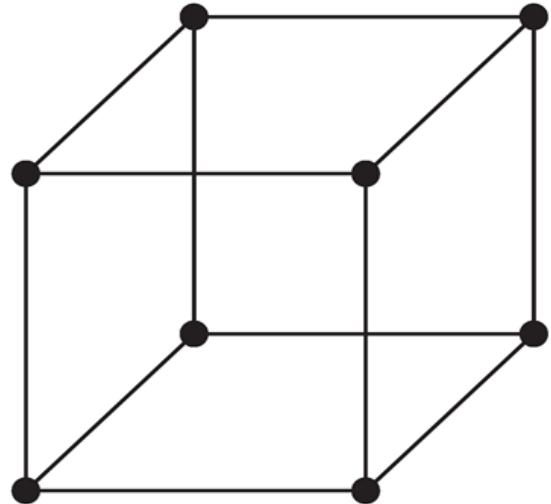


5 - 5

(a)

Planar Graphs

■ Example



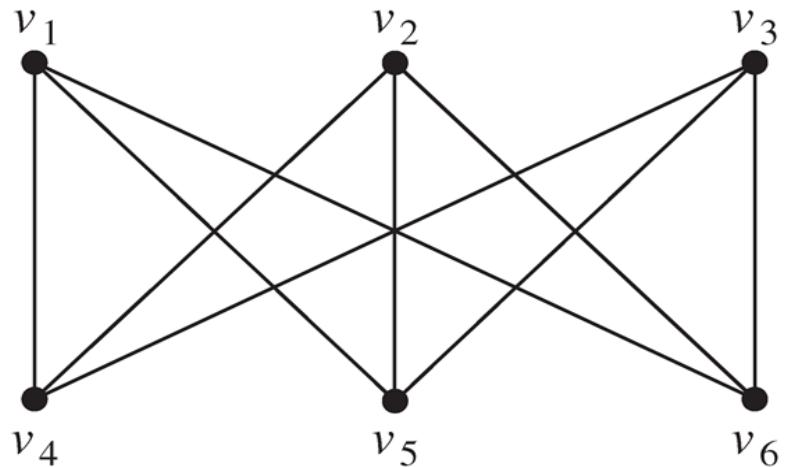
5 -

(a)

(b)

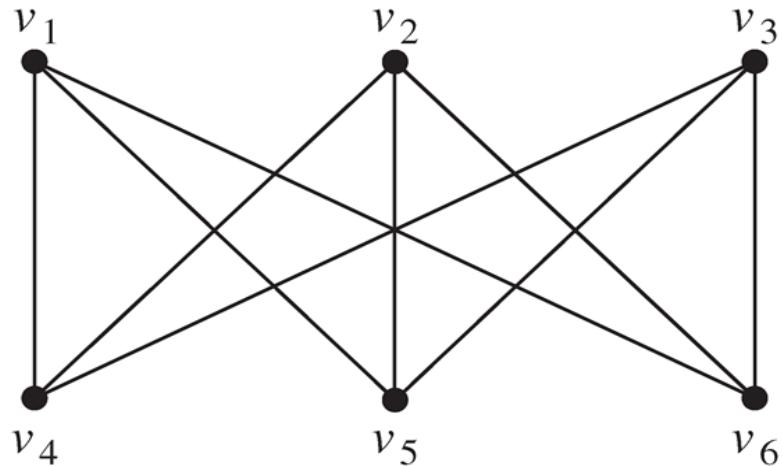
Planar Graphs

■ Example



Planar Graphs

■ Example



Applications

- ◊ IC design
- ◊ design of road networks

Euler's Formula

- **Theorem** (Euler's Formula) Let G be a connected planar simple graph with e edges and v vertices. Let r be the number of regions in a planar representation of G . Then $r = e - v + 2$.

Proof (by induction)

Euler's Formula

- **Theorem (Euler's Formula)** Let G be a connected planar simple graph with e edges and v vertices. Let r be the number of regions in a planar representation of G . Then $r = e - v + 2$.

Proof (by induction)

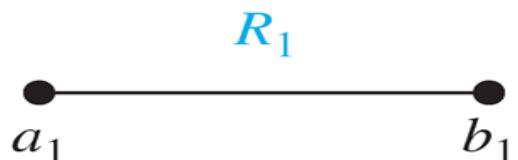
Basic step:

Euler's Formula

- **Theorem (Euler's Formula)** Let G be a connected planar simple graph with e edges and v vertices. Let r be the number of regions in a planar representation of G . Then $r = e - v + 2$.

Proof (by induction)

Basic step:



Euler's Formula

- **Theorem (Euler's Formula)** Let G be a connected planar simple graph with e edges and v vertices. Let r be the number of regions in a planar representation of G . Then $r = e - v + 2$.

Proof (by induction)

Basic step:

Inductive Hypothesis:

Euler's Formula

- **Theorem** (Euler's Formula) Let G be a connected planar simple graph with e edges and v vertices. Let r be the number of regions in a planar representation of G . Then $r = e - v + 2$.

Proof (by induction)

Basic step:

Inductive Hypothesis:

$$r_k = e_k - v_k + 2$$

Euler's Formula

- **Theorem** (Euler's Formula) Let G be a connected planar simple graph with e edges and v vertices. Let r be the number of regions in a planar representation of G . Then $r = e - v + 2$.

Proof (by induction)

Basic step:

Inductive Hypothesis:

Inductive step:

Euler's Formula

- **Theorem** (Euler's Formula) Let G be a connected planar simple graph with e edges and v vertices. Let r be the number of regions in a planar representation of G . Then $r = e - v + 2$.

Proof (by induction)

Basic step:

Inductive Hypothesis:

Inductive step:

Let $\{a_{k+1}, b_{k+1}\}$ be the edge that is added to G_k to obtain G_{k+1} .

Euler's Formula

- **Theorem (Euler's Formula)** Let G be a connected planar simple graph with e edges and v vertices. Let r be the number of regions in a planar representation of G . Then $r = e - v + 2$.

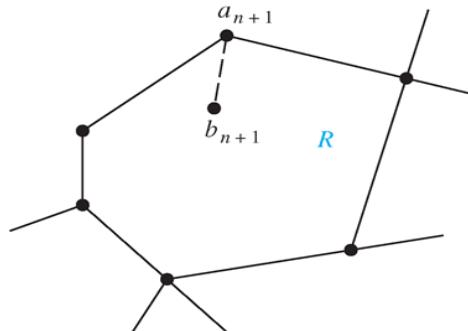
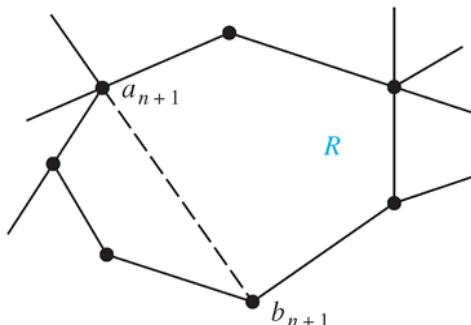
Proof (by induction)

Basic step:

Inductive Hypothesis:

Inductive step:

Let $\{a_{k+1}, b_{k+1}\}$ be the edge that is added to G_k to obtain G_{k+1} .

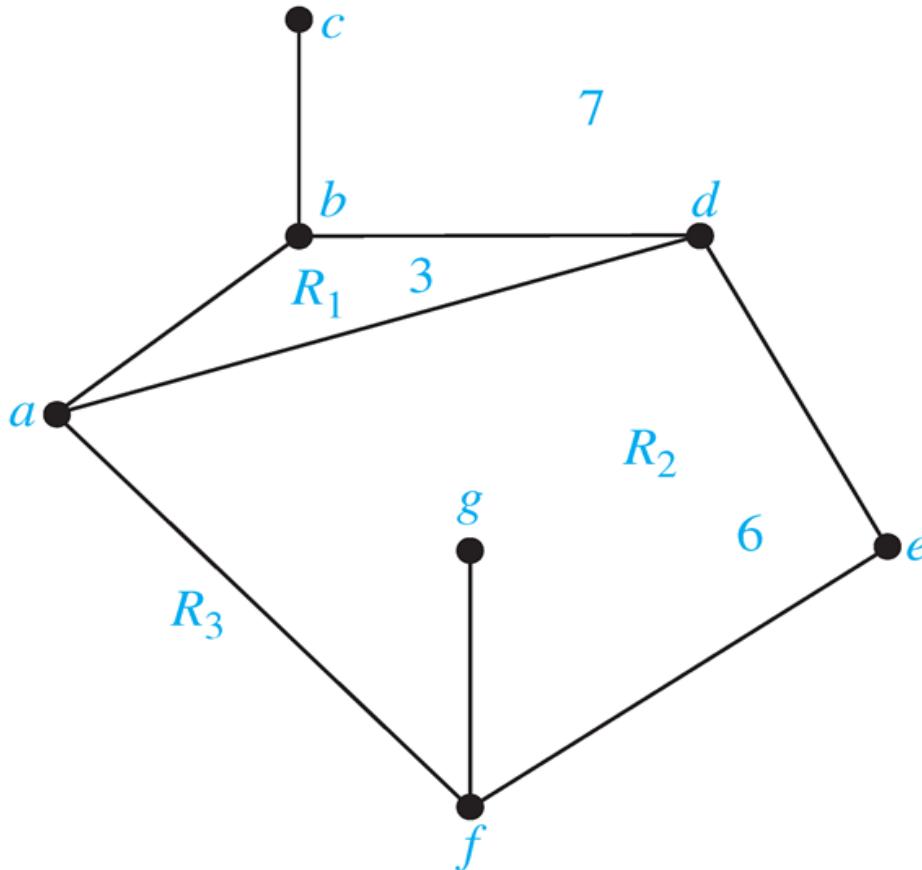


The Degree of Regions

- **Definition** The *degree* of a **region** is defined to be the number of edges on the **boundary of this region**. When an edge occurs **twice** on the boundary, it contributes **two** to the degree.

The Degree of Regions

- **Definition** The *degree* of a **region** is defined to be the number of edges on the **boundary** of this region. When an edge occurs **twice** on the boundary, it contributes **two** to the degree.



Corollaries

- **Corollary 1** If G is a connected planar simple graph with e edges and v vertices, where $v \geq 3$, then $e \leq 3v - 6$.



Corollaries

- **Corollary 1** If G is a connected planar simple graph with e edges and v vertices, where $v \geq 3$, then $e \leq 3v - 6$.

Proof The degree of every region is at least 3.



Corollaries

- **Corollary 1** If G is a connected planar simple graph with e edges and v vertices, where $v \geq 3$, then $e \leq 3v - 6$.

Proof The degree of every region is at least 3.

- ◊ G is simple
- ◊ $v \geq 3$

Corollaries

- **Corollary 1** If G is a connected planar simple graph with e edges and v vertices, where $v \geq 3$, then $e \leq 3v - 6$.

Proof The degree of every region is at least 3.

- ◊ G is simple
- ◊ $v \geq 3$

The sum of the degrees of the regions is exactly twice the number of edges in the graph.

Corollaries

- **Corollary 1** If G is a connected planar simple graph with e edges and v vertices, where $v \geq 3$, then $e \leq 3v - 6$.

Proof The degree of every region is at least 3.

- ◊ G is simple
- ◊ $v \geq 3$

The sum of the degrees of the regions is exactly twice the number of edges in the graph.

$$2e = \sum_{\text{all regions } R} \deg(R) \geq 3r$$

Corollaries

- **Corollary 1** If G is a connected planar simple graph with e edges and v vertices, where $v \geq 3$, then $e \leq 3v - 6$.

Proof The degree of every region is at least 3.

- ◊ G is simple
- ◊ $v \geq 3$

The sum of the degrees of the regions is exactly twice the number of edges in the graph.

$$2e = \sum_{\text{all regions } R} \deg(R) \geq 3r$$

By Euler's formula, the proof is completed.

Corollaries

- **Corollary 2** If G is a connected planar simple graph, then G has a vertex of degree not exceeding 5.



Corollaries

- **Corollary 2** If G is a connected planar simple graph, then G has a vertex of degree not exceeding 5.

Proof



Corollaries

- **Corollary 2** If G is a connected planar simple graph, then G has a vertex of degree not exceeding 5.

Proof

(By contradiction)

By Corollary 1 and the Handshaking Theorem.

Corollaries

- **Corollary 2** If G is a connected planar simple graph, then G has a vertex of degree not exceeding 5.

Proof

(By contradiction)

By Corollary 1 and the Handshaking Theorem.

Corollary 3 In a connected planar simple graph has e edges and v vertices with $v \geq 3$ and no circuits of length three, then $e \leq 2v - 4$.



Corollaries

- **Corollary 2** If G is a connected planar simple graph, then G has a vertex of degree not exceeding 5.

Proof

(By contradiction)

By Corollary 1 and the Handshaking Theorem.

Corollary 3 In a connected planar simple graph has e edges and v vertices with $v \geq 3$ and no circuits of length three, then $e \leq 2v - 4$.

Proof similar to that of Corollary 1.

Examples

- Show that K_5 is nonplanar.

Examples

- Show that K_5 is nonplanar.

Using Corollary 1

Examples

- Show that K_5 is nonplanar.

Using Corollary 1

Show that $K_{3,3}$ is nonplanar.

Examples

- Show that K_5 is nonplanar.

Using Corollary 1

Show that $K_{3,3}$ is nonplanar.

Using Corollary 3

Examples

- Show that K_5 is nonplanar.

Using Corollary 1

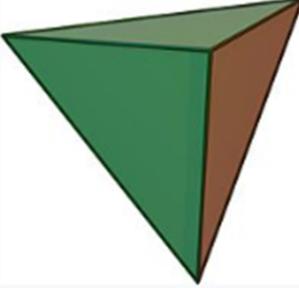
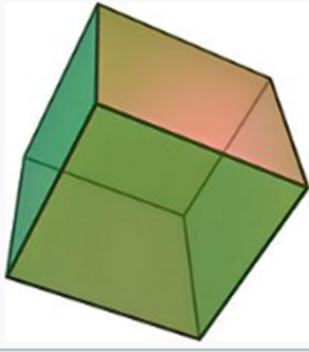
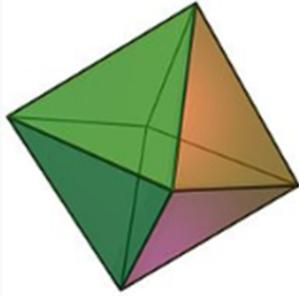
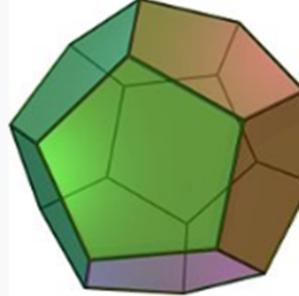
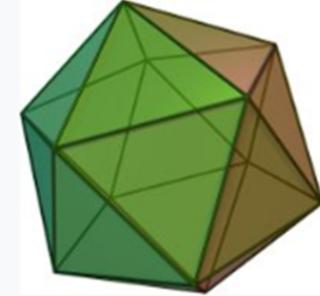
Show that $K_{3,3}$ is nonplanar.

Using Corollary 3

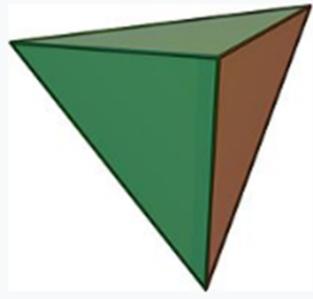
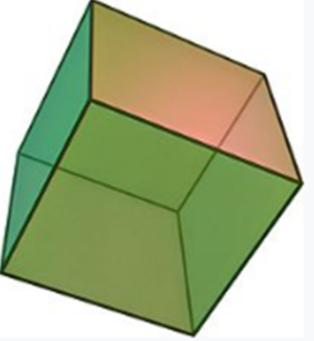
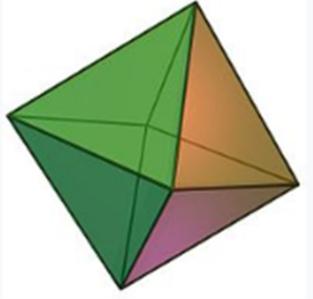
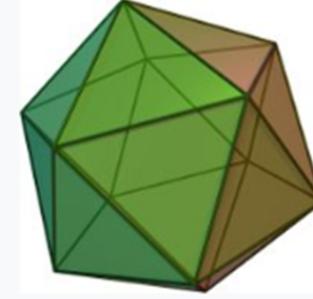
Corollary 2 is used in the proof of Five Color Theorem.

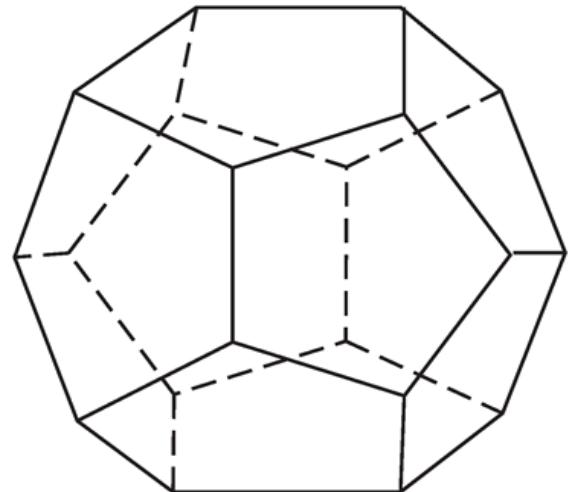


Only 5 Platonic Solids

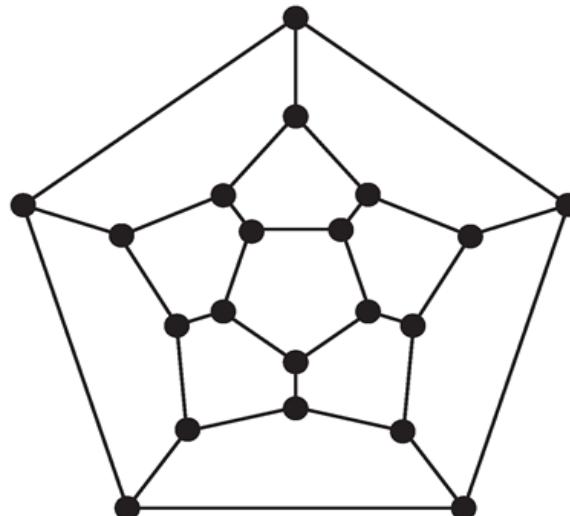
				
Tetrahedron $\{3, 3\}$	Cube $\{4, 3\}$	Octahedron $\{3, 4\}$	Dodecahedron $\{5, 3\}$	Icosahedron $\{3, 5\}$

Only 5 Platonic Solids

				
Tetrahedron $\{3, 3\}$	Cube $\{4, 3\}$	Octahedron $\{3, 4\}$	Dodecahedron $\{5, 3\}$	Icosahedron $\{3, 5\}$



(a)



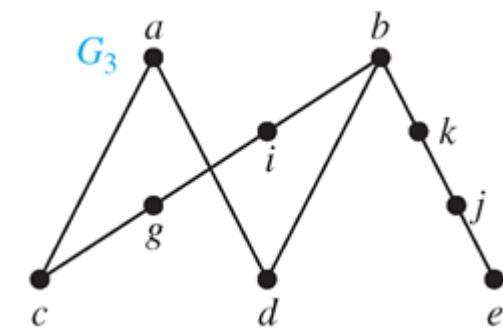
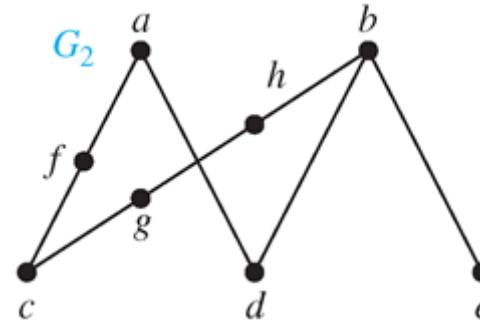
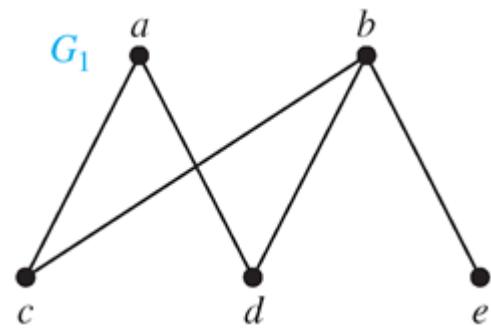
(b)

Kuratowski's Theorem

- **Definition** If a graph is planar, so will be **any graph** obtained by removing an edge $\{u, v\}$ and adding a new vertex w together with edges $\{u, w\}$ and $\{w, v\}$. Such an operation is called an ***elementary subdivision***. The graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ are called ***homeomorphic*** if they can be obtained from **the same graph** by a sequence of elementary subdivisions.

Kuratowski's Theorem

- **Definition** If a graph is planar, so will be **any graph** obtained by removing an edge $\{u, v\}$ and adding a new vertex w together with edges $\{u, w\}$ and $\{w, v\}$. Such an operation is called an *elementary subdivision*. The graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ are called *homeomorphic* if they can be obtained from **the same graph** by a sequence of elementary subdivisions.

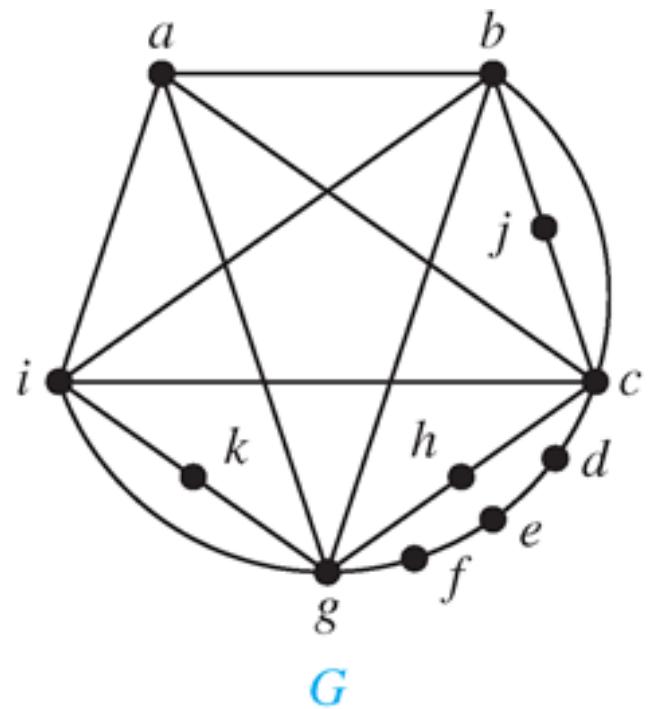


Kuratowski's Theorem

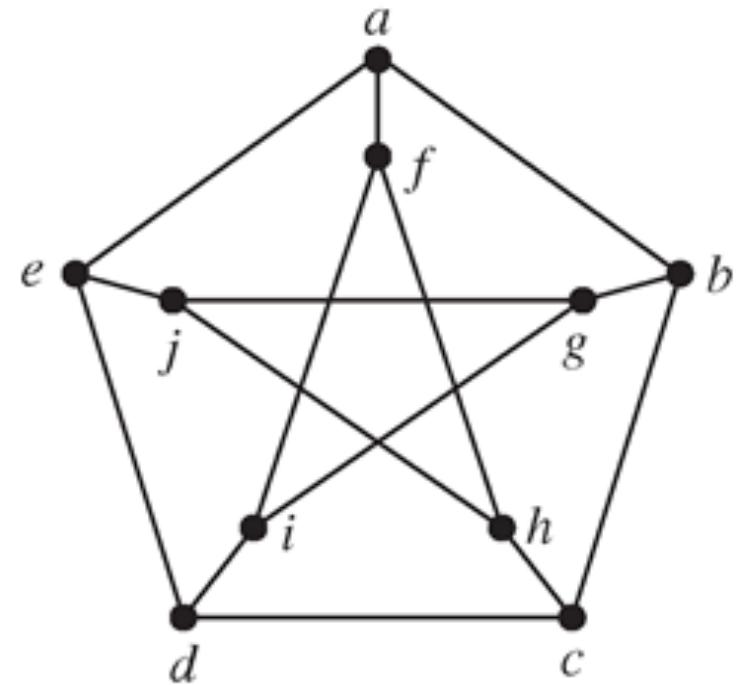
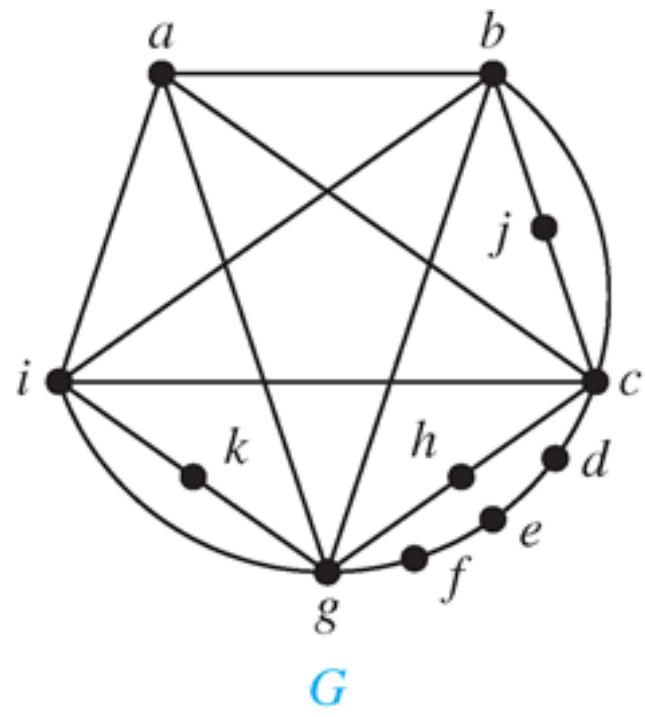
- **Definition** If a graph is planar, so will be **any graph** obtained by removing an edge $\{u, v\}$ and adding a new vertex w together with edges $\{u, w\}$ and $\{w, v\}$. Such an operation is called an *elementary subdivision*. The graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ are called *homeomorphic* if they can be obtained from **the same graph** by a sequence of elementary subdivisions.

Theorem A graph is **nonplanar** if and only if it contains a subgraph **homomorphic** to $K_{3,3}$ or K_5 .

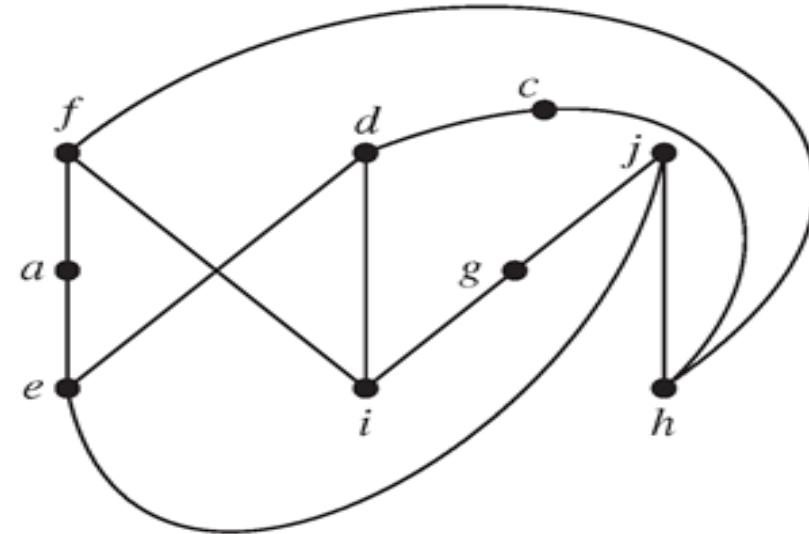
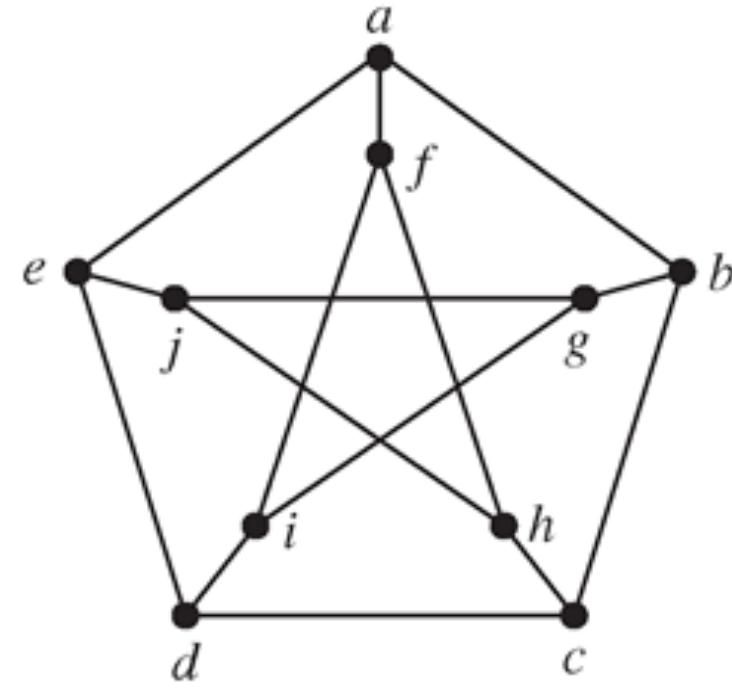
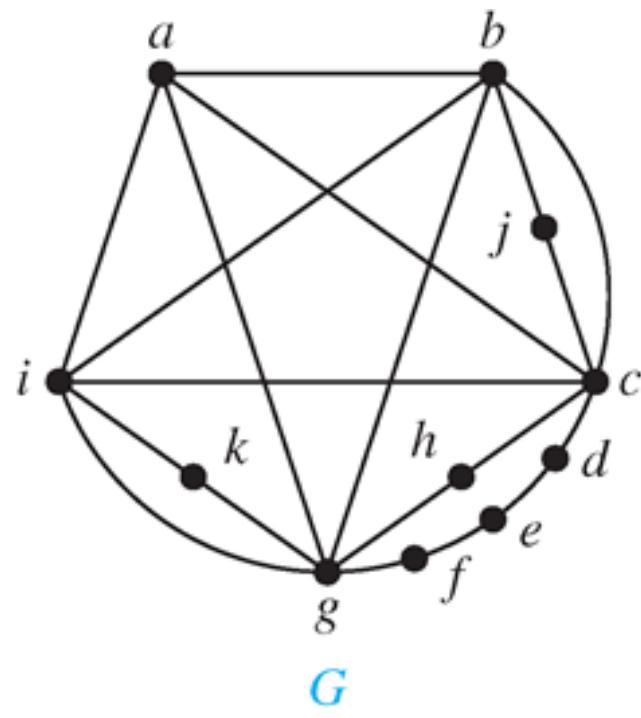
Examples



Examples

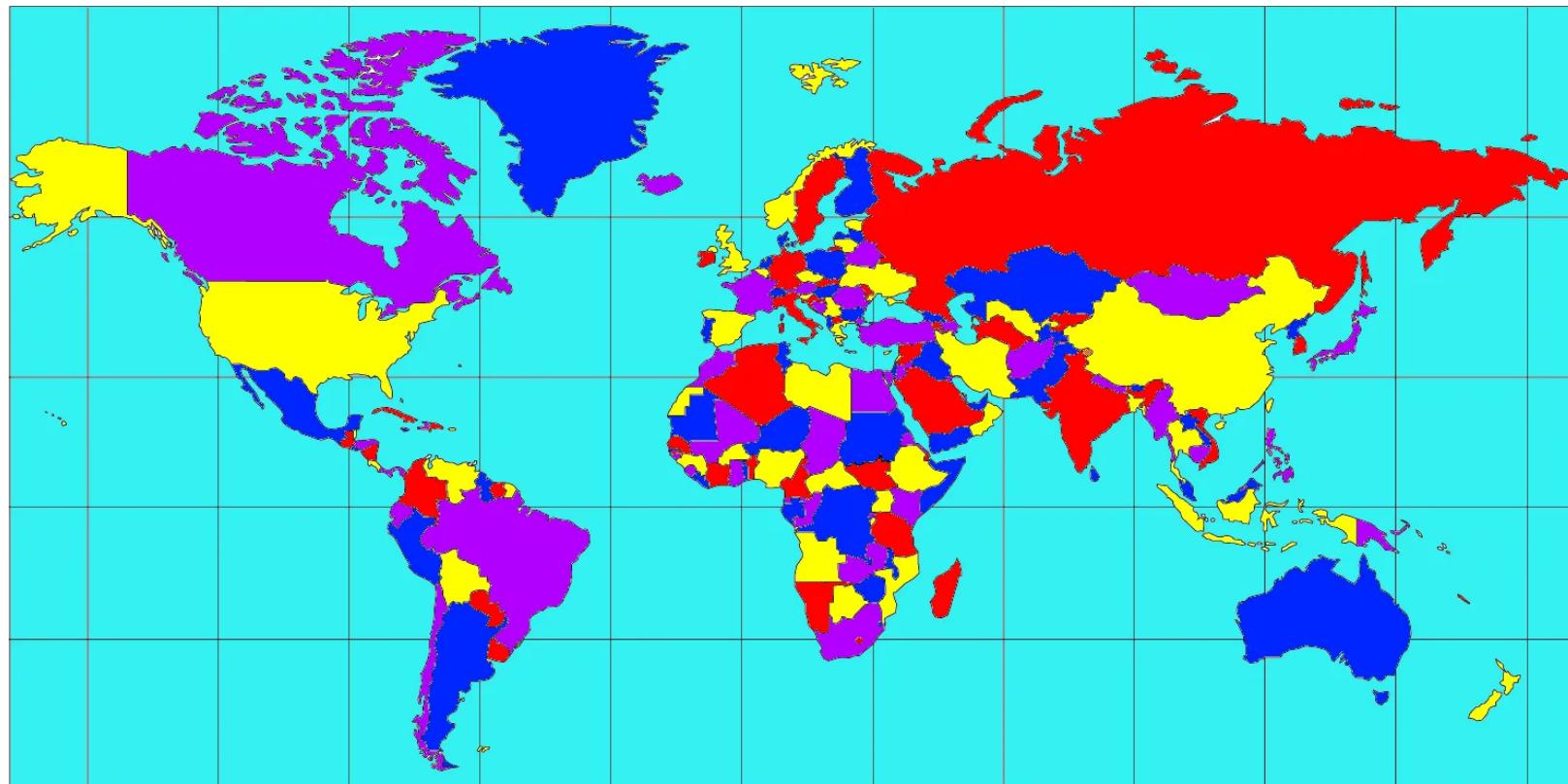


Examples



Graph Coloring

- **Four-color theorem** Given any separation of a plane into contiguous regions, producing a figure called a *map*, no more than four colors are required to color the regions of the map so that no two adjacent regions have the same color.



Graph Coloring

■ Four-color theorem

- ◊ first proposed by Francis Guthrie in 1852
- ◊ his brother Frederick Guthrie told Augustus De Morgan
- ◊ De Morgan wrote to William Hamilton
- ◊ Alfred Kempe proved it **incorrectly** in 1879
- ◊ Percy Heawood found an error in 1890 and proved the *five-color theorem*
- ◊ Finally, Kenneth Appel and Wolfgang Haken proved it with case by case analysis by computer in 1976 (*the first computer-aided proof*)
- ◊ Kempe's incorrect proof serves as a basis

Graph Coloring

- A *coloring* of a simple graph is the **assignment** of a color to each **vertex** of the graph so that **no two adjacent vertices** are assigned the same color.



Graph Coloring

- A *coloring* of a simple graph is the **assignment** of a color to each **vertex** of the graph so that **no two adjacent vertices** are assigned **the same color**.

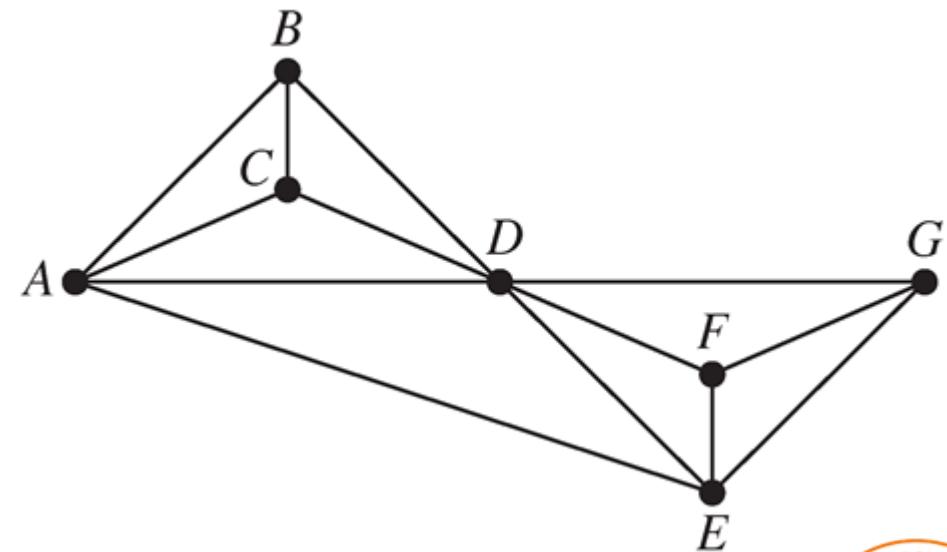
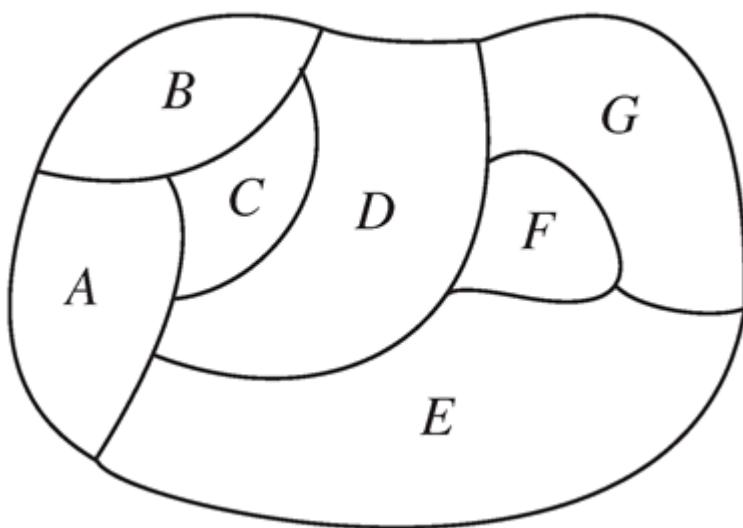
The *chromatic number* of a graph is the **least number** of colors needed for a coloring of this graph, denoted by $\chi(G)$.



Graph Coloring

- A *coloring* of a simple graph is the *assignment* of a color to each *vertex* of the graph so that *no two adjacent vertices* are assigned the same color.

The *chromatic number* of a graph is the *least number* of colors needed for a coloring of this graph, denoted by $\chi(G)$.

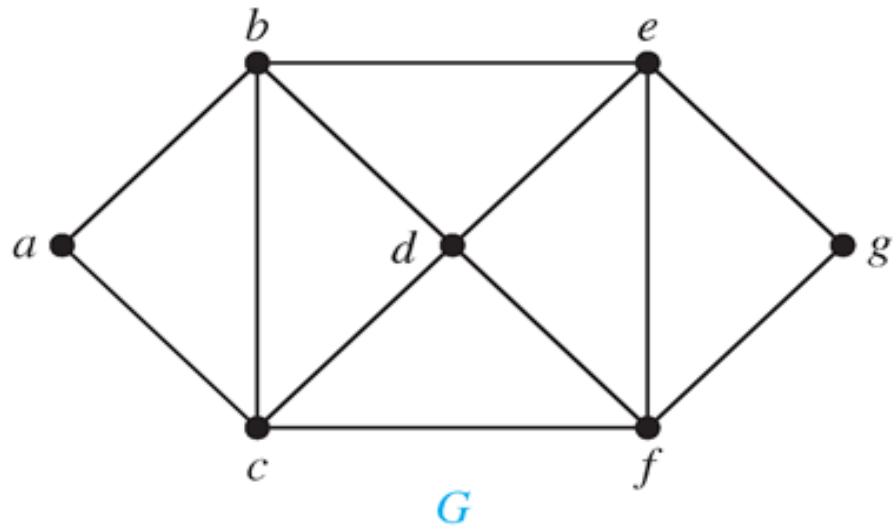


Graph Coloring

- **Theorem** (Four Color Theorem) The chromatic number of a planar graph is no greater than four.

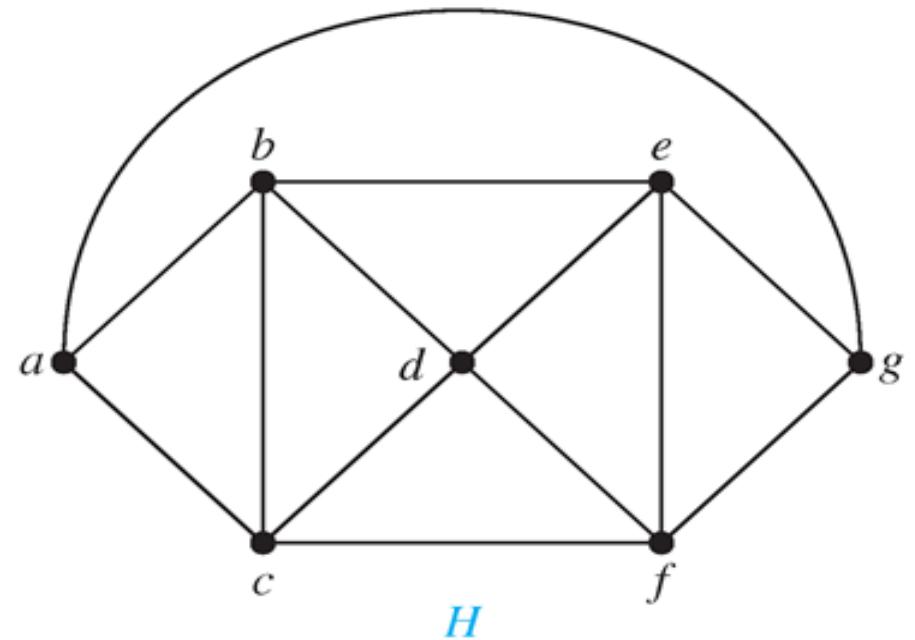
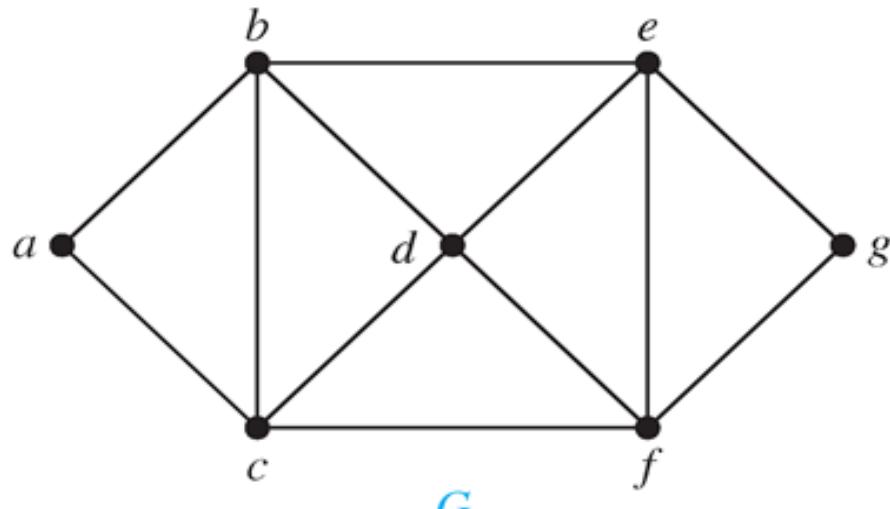
Graph Coloring

- **Theorem** (Four Color Theorem) The chromatic number of a planar graph is no greater than four.



Graph Coloring

- **Theorem** (Four Color Theorem) The chromatic number of a planar graph is no greater than four.



Graph Coloring

- **Theorem** (Six Color Theorem) The chromatic number of a planar graph is no greater than six.

Graph Coloring

- **Theorem** (Six Color Theorem) The chromatic number of a planar graph is no greater than six.

Proof (by induction on the number of vertices)
w.l.o.g., assume that the graph is connected.



Graph Coloring

- **Theorem** (Six Color Theorem) The chromatic number of a planar graph is no greater than six.

Proof (by induction on the number of vertices)
w.l.o.g., assume that the graph is connected.

Basic step: For one single vertex, pick an arbitrary color.

Graph Coloring

- **Theorem** (Six Color Theorem) The chromatic number of a planar graph is no greater than six.

Proof (by induction on the number of vertices)
w.l.o.g., assume that the graph is connected.

Basic step: For one single vertex, pick an arbitrary color.

Inductive hypothesis: Assume that every planar graph with $k \geq 1$ or fewer vertices can be 6-colored.

Graph Coloring

- **Theorem** (Six Color Theorem) The chromatic number of a planar graph is no greater than six.

Proof (by induction on the number of vertices)
w.l.o.g., assume that the graph is connected.

Basic step: For one single vertex, pick an arbitrary color.

Inductive hypothesis: Assume that every planar graph with $k \geq 1$ or fewer vertices can be 6-colored.

Inductive step: Consider a planar graph with $k + 1$ vertices.

Graph Coloring

- **Theorem** (Six Color Theorem) The chromatic number of a planar graph is no greater than six.

Proof (by induction on the number of vertices)
w.l.o.g., assume that the graph is connected.

Basic step: For one single vertex, pick an arbitrary color.

Inductive hypothesis: Assume that every planar graph with $k \geq 1$ or fewer vertices can be 6-colored.

Inductive step: Consider a planar graph with $k + 1$ vertices. Recall Corollary 2 (the graph has a vertex of degree 5 or fewer). Remove this vertex, by i.h., we can color the remaining graph with 6 colors. Put the vertex back in. Since there are at most 5 colors adjacent, so we have at least one color left.



Graph Coloring

- **Theorem** (Five Color Theorem) The chromatic number of a planar graph is no greater than five.

Graph Coloring

- **Theorem** (Five Color Theorem) The chromatic number of a planar graph is no greater than five.

Proof (by induction on the number of vertices)
w.l.o.g., assume that the graph is connected.

Graph Coloring

- **Theorem** (Five Color Theorem) The chromatic number of a planar graph is no greater than five.

Proof (by induction on the number of vertices)
w.l.o.g., assume that the graph is connected.

Basic step: For one single vertex, pick an arbitrary color.

Graph Coloring

- **Theorem** (Five Color Theorem) The chromatic number of a planar graph is no greater than five.

Proof (by induction on the number of vertices)
w.l.o.g., assume that the graph is connected.

Basic step: For one single vertex, pick an arbitrary color.

Inductive hypothesis: Assume that every planar graph with $k \geq 1$ or fewer vertices can be 5-colored.

Graph Coloring

- **Theorem** (Five Color Theorem) The chromatic number of a planar graph is no greater than five.

Proof (by induction on the number of vertices)
w.l.o.g., assume that the graph is connected.

Basic step: For one single vertex, pick an arbitrary color.

Inductive hypothesis: Assume that every planar graph with $k \geq 1$ or fewer vertices can be 5-colored.

Inductive step: Consider a planar graph with $k + 1$ vertices.

Graph Coloring

- **Theorem** (Five Color Theorem) The chromatic number of a planar graph is no greater than five.

Proof (by induction on the number of vertices)
w.l.o.g., assume that the graph is connected.

Basic step: For one single vertex, pick an arbitrary color.

Inductive hypothesis: Assume that every planar graph with $k \geq 1$ or fewer vertices can be 5-colored.

Inductive step: Consider a planar graph with $k + 1$ vertices. Recall Corollary 2 (the graph has a vertex of degree 5 or fewer). Remove this vertex, by i.h., we can color the remaining graph with 5 colors. Put the vertex back in.

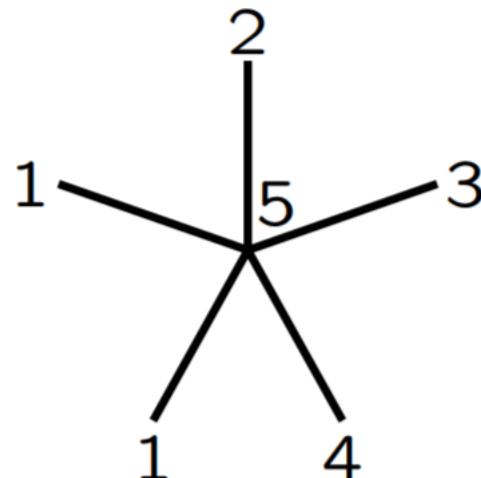
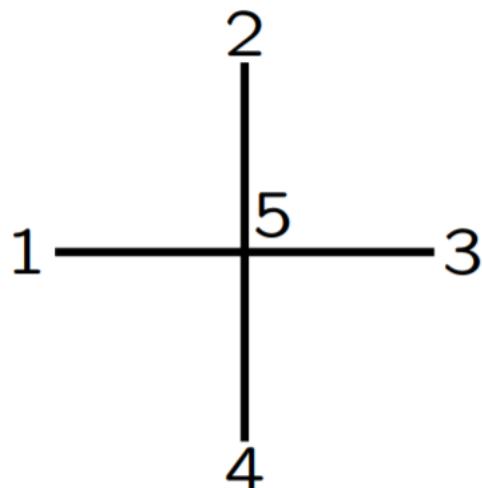


Graph Coloring

- **Theorem** (Five Color Theorem) The chromatic number of a planar graph is no greater than five.

Proof (by induction on the number of vertices)
w.l.o.g., assume that the graph is connected.

If the vertex has degree less than 5, or if it has degree 5 and only ≤ 4 colors are used for vertices connected to it, we can pick an available color for it.

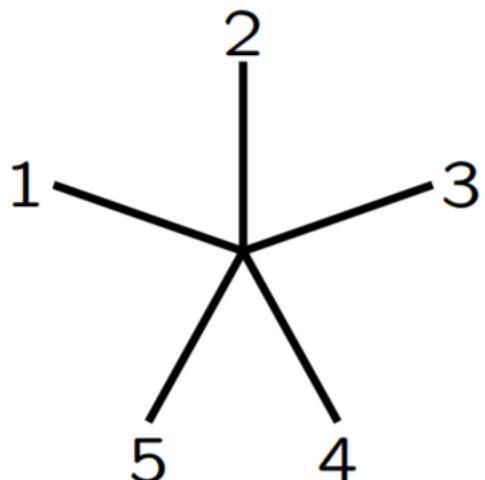


Graph Coloring

- **Theorem** (Five Color Theorem) The chromatic number of a planar graph is no greater than five.

Proof (by induction on the number of vertices)

If the vertex has degree 5, and all 5 colors are connected to it, we label the vertices adjacent to the “special” vertex (degree 5) 1 to 5 (in order).



Graph Coloring

- **Theorem** (Five Color Theorem) The chromatic number of a planar graph is no greater than five.

Proof (by induction on the number of vertices)

We make a subgraph out of all the vertices colored 1 or 3. If the adjacent vertex colored 1 and the adjacent vertex colored 3 are not connected by a path in the subgraph.

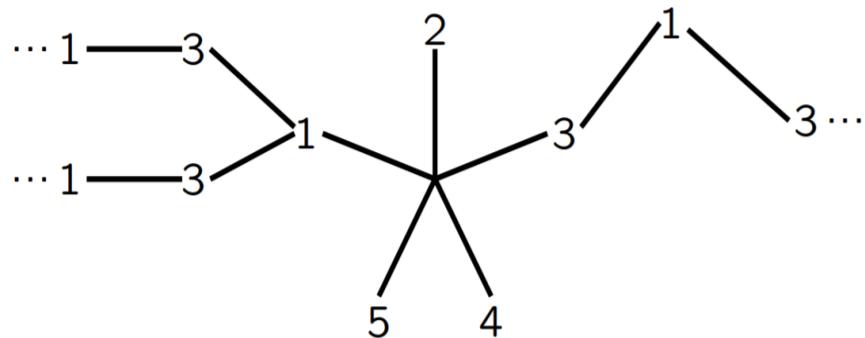


Graph Coloring

- **Theorem** (Five Color Theorem) The chromatic number of a planar graph is no greater than five.

Proof (by induction on the number of vertices)

We make a subgraph out of all the vertices colored 1 or 3. If the adjacent vertex colored 1 and the adjacent vertex colored 3 are not connected by a path in the subgraph.

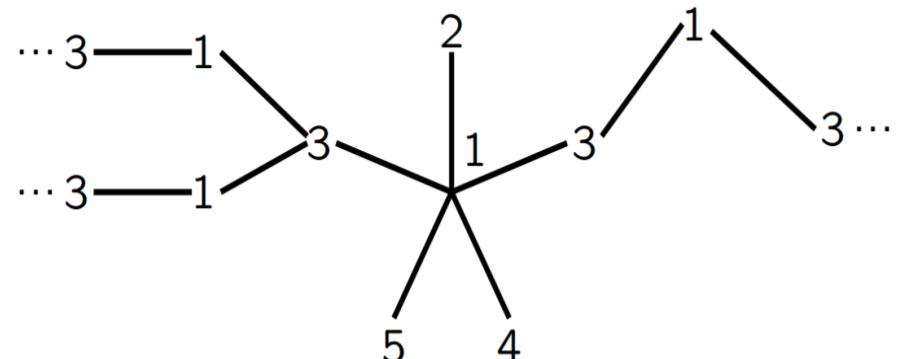
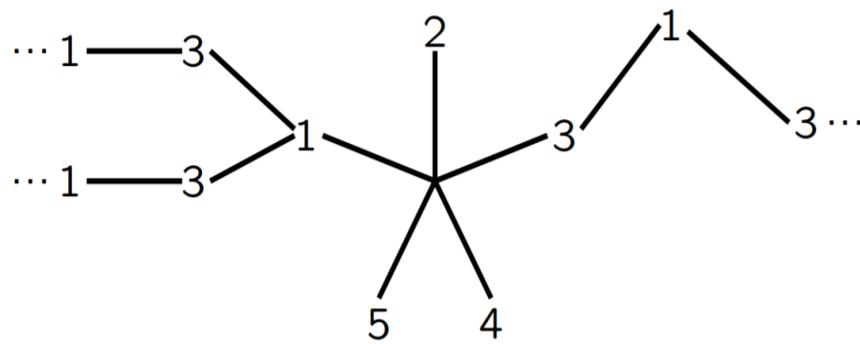


Graph Coloring

- **Theorem** (Five Color Theorem) The chromatic number of a planar graph is no greater than five.

Proof (by induction on the number of vertices)

We make a subgraph out of all the vertices colored 1 or 3. If the adjacent vertex colored 1 and the adjacent vertex colored 3 are not connected by a path in the subgraph.



Graph Coloring

- **Theorem** (Five Color Theorem) The chromatic number of a planar graph is no greater than five.

Proof (by induction on the number of vertices)

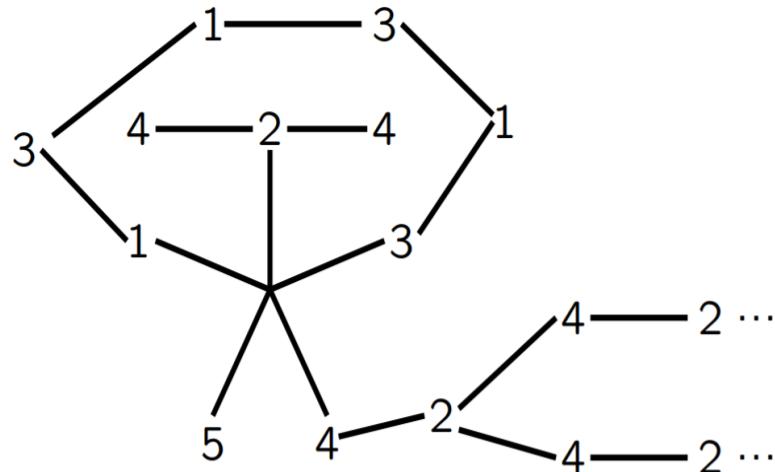
On the other hand, if the vertices colored 1 and 3 are connected via a path in the subgraph, we do the **the same** for the vertices colored 2 and 4. Note that this will be a disconnected pair of subgraphs, separated by a path connecting the vertices colored 1 and 3 (**Why?**)

Graph Coloring

- **Theorem** (Five Color Theorem) The chromatic number of a planar graph is no greater than five.

Proof (by induction on the number of vertices)

On the other hand, if the vertices colored 1 and 3 are connected via a path in the subgraph, we do the **the same** for the vertices colored 2 and 4. Note that this will be a disconnected pair of subgraphs, separated by a path connecting the vertices colored 1 and 3 (**Why?**)

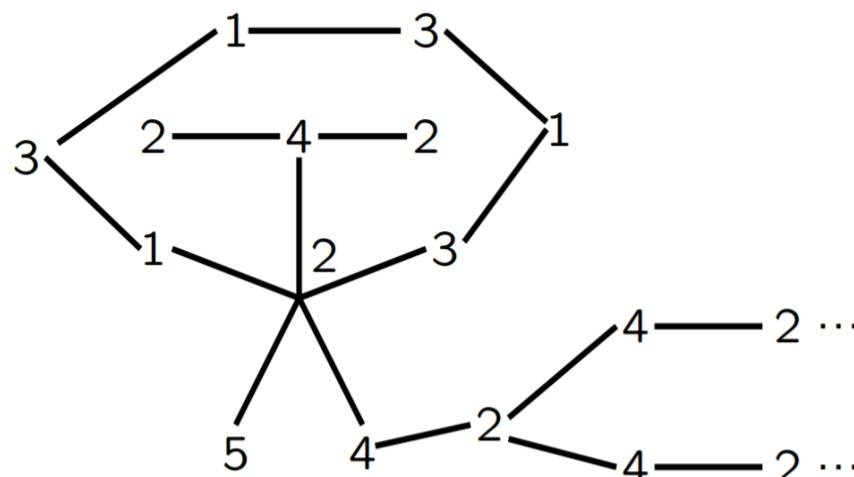
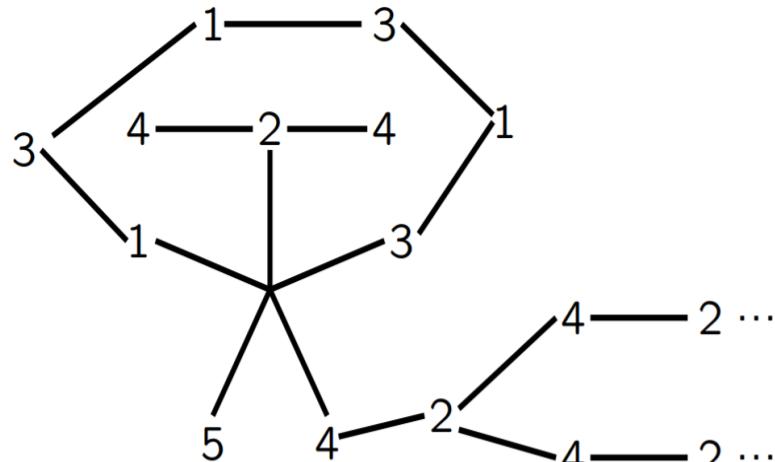


Graph Coloring

- **Theorem** (Five Color Theorem) The chromatic number of a planar graph is no greater than five.

Proof (by induction on the number of vertices)

On the other hand, if the vertices colored 1 and 3 are connected via a path in the subgraph, we do the **the same** for the vertices colored 2 and 4. Note that this will be a disconnected pair of subgraphs, separated by a path connecting the vertices colored 1 and 3 (**Why?**)

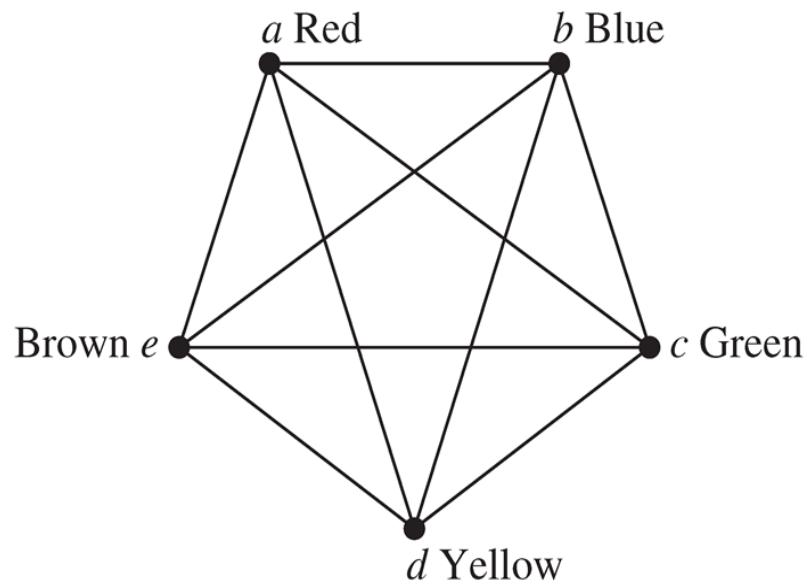


Examples

- What is the chromatic number of K_n , $K_{m,n}$, C_n ?

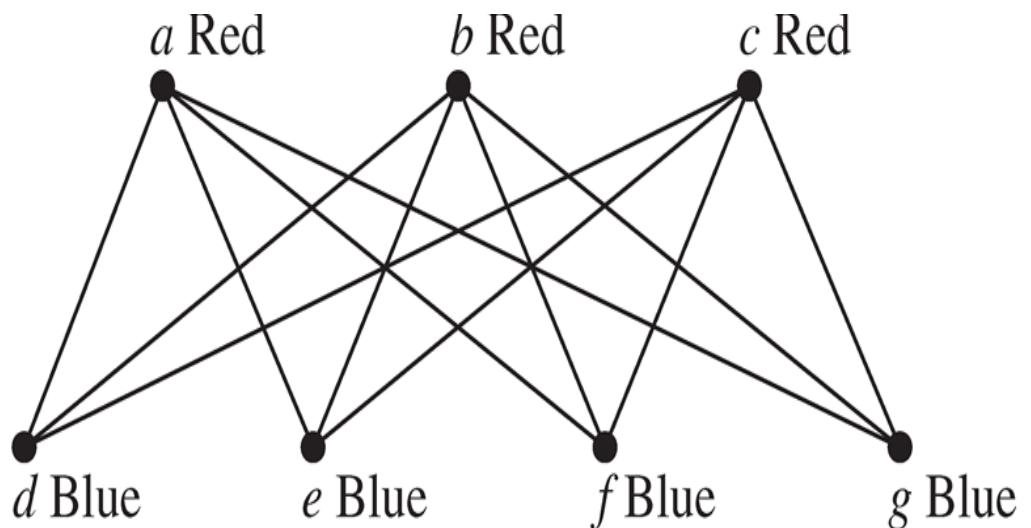
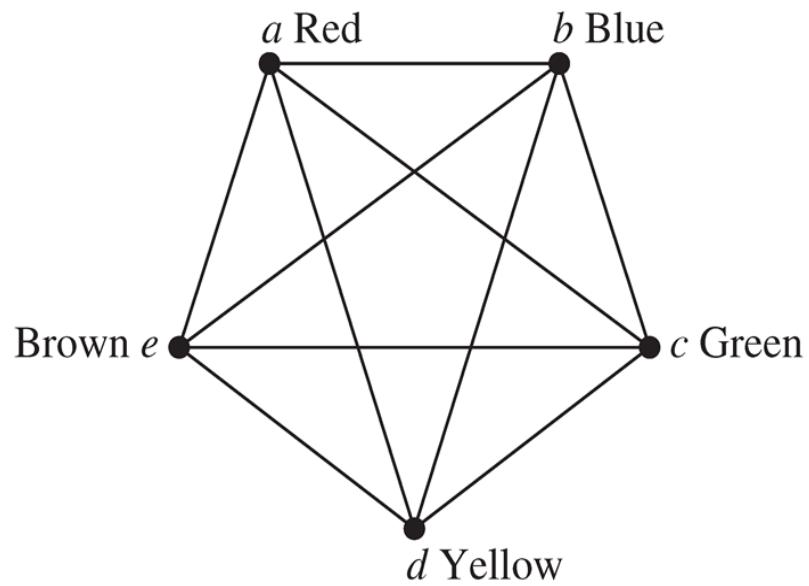
Examples

- What is the chromatic number of K_n , $K_{m,n}$, C_n ?



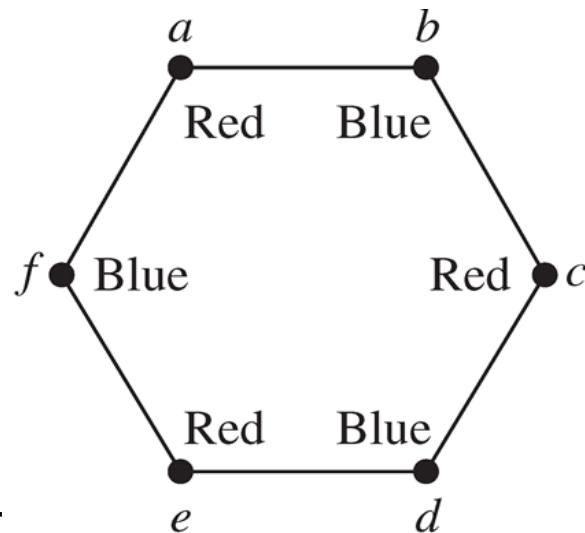
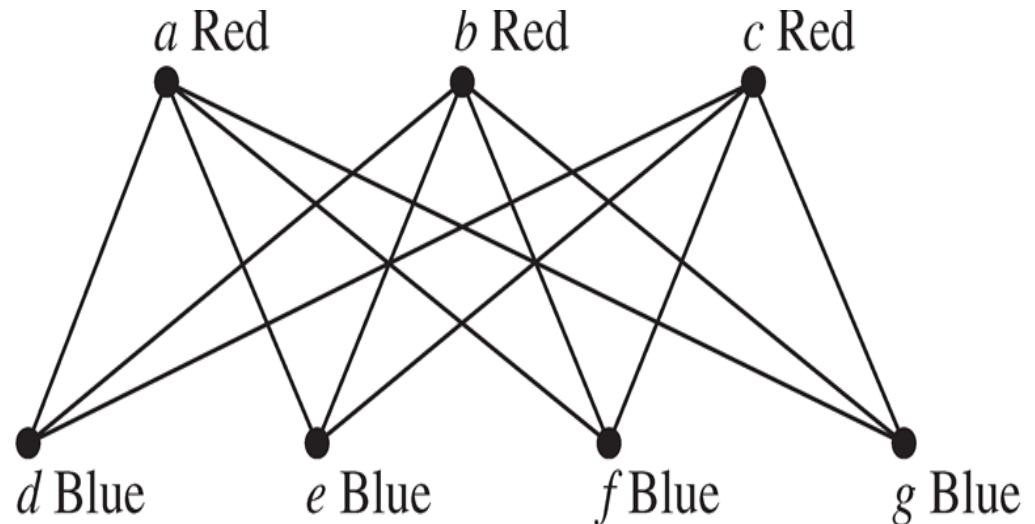
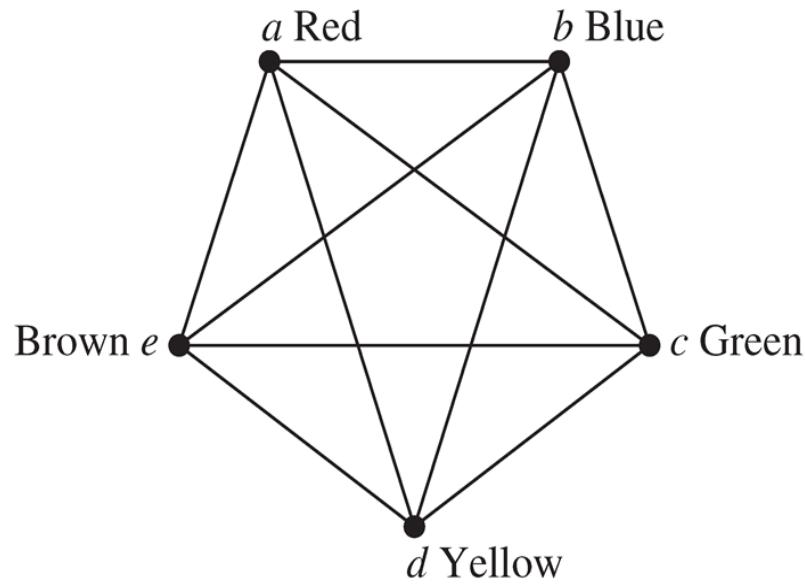
Examples

- What is the chromatic number of K_n , $K_{m,n}$, C_n ?



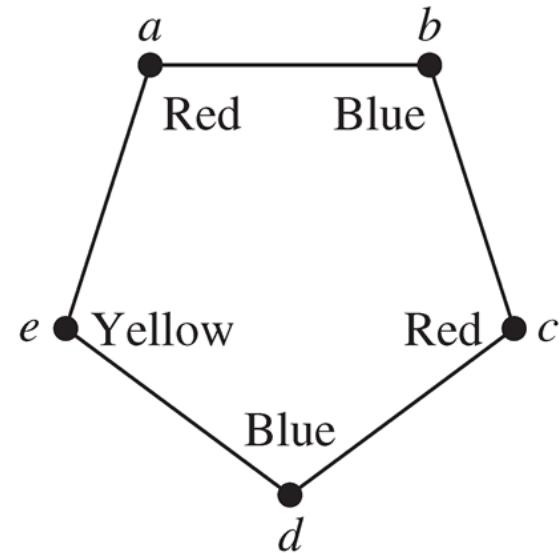
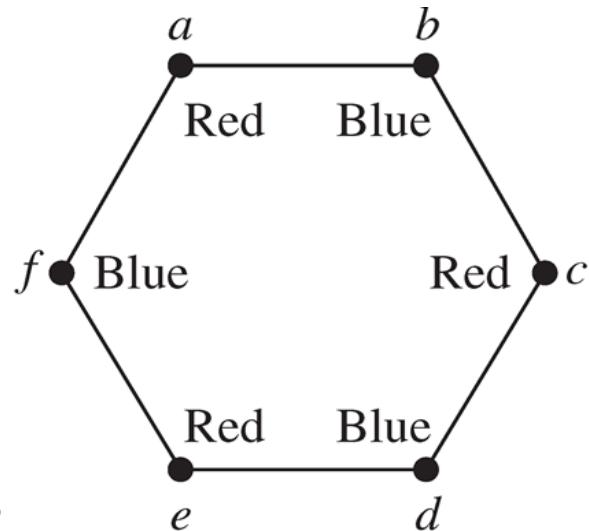
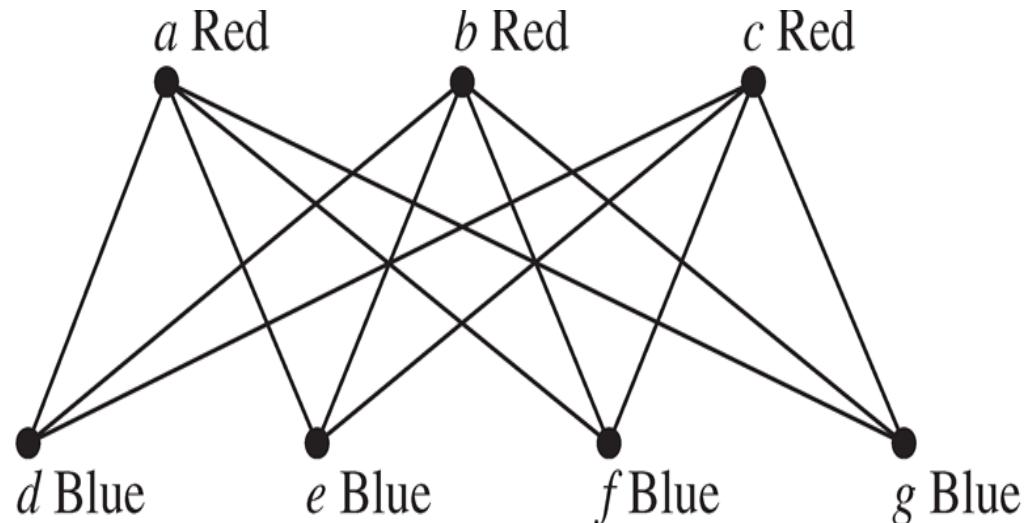
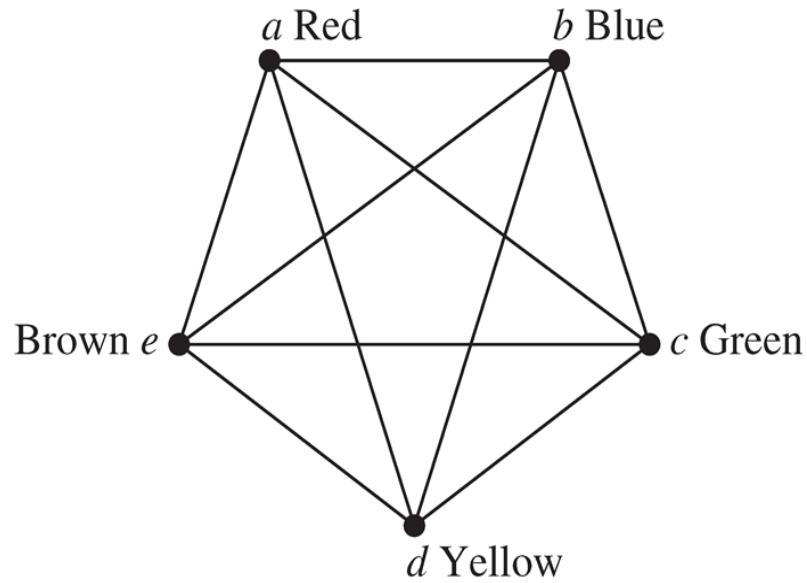
Examples

- What is the chromatic number of K_n , $K_{m,n}$, C_n ?



Examples

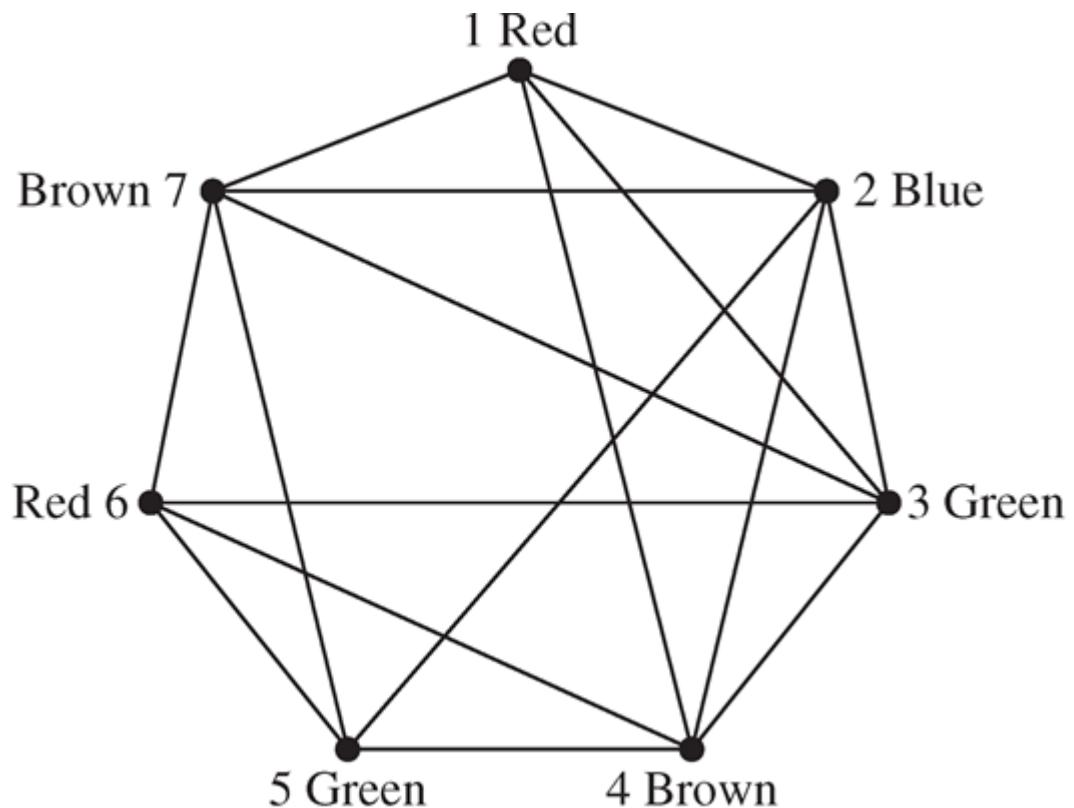
- What is the chromatic number of K_n , $K_{m,n}$, C_n ?



Applications of Graph Coloring

Scheduling Final Exams

Vertices represent courses, and there is an edge between two vertices if there is a common student in the courses.



Time Period	Courses
I	1, 6
II	2
III	3, 5
IV	4, 7

Applications of Graph Coloring

■ Channel Assignments

Television channels 2 through 13 are assigned to stations in North America so that no two stations within 150 miles can operate on the same channel . How can the assignment of channels be modeled by graph coloring?

Applications of Graph Coloring

■ Channel Assignments

Television channels 2 through 13 are assigned to stations in North America so that no two stations within 150 miles can operate on the same channel . How can the assignment of channels be modeled by graph coloring?

Graph Coloring \in NPC

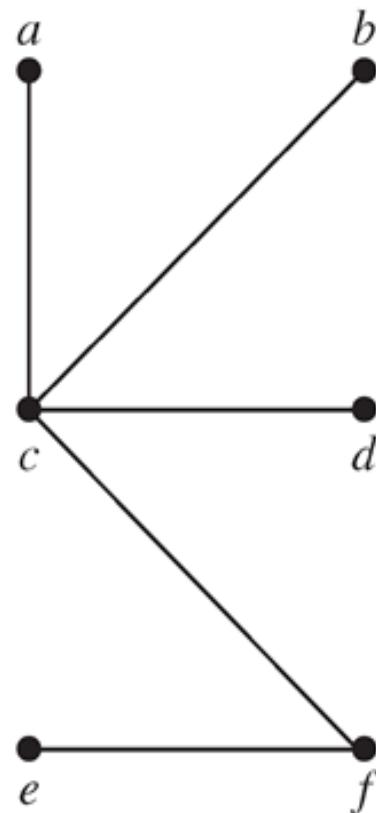
Trees

- **Definition** A *tree* is a connected undirected graph with no simple circuits.



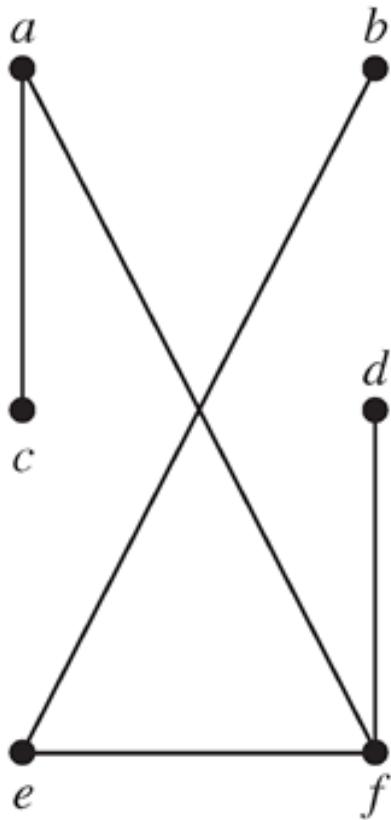
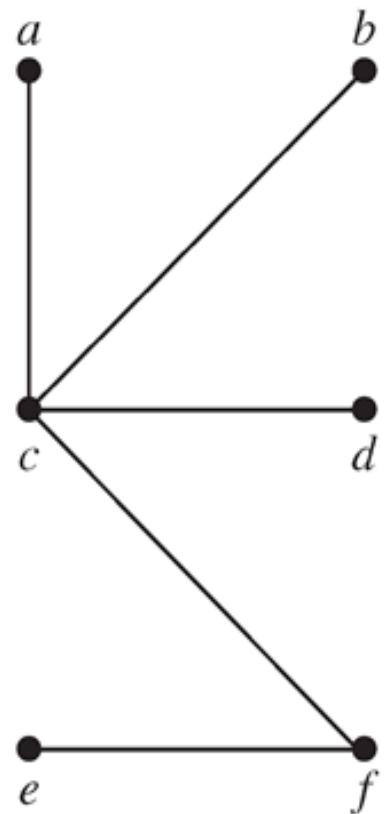
Trees

- **Definition** A *tree* is a connected undirected graph with no simple circuits.



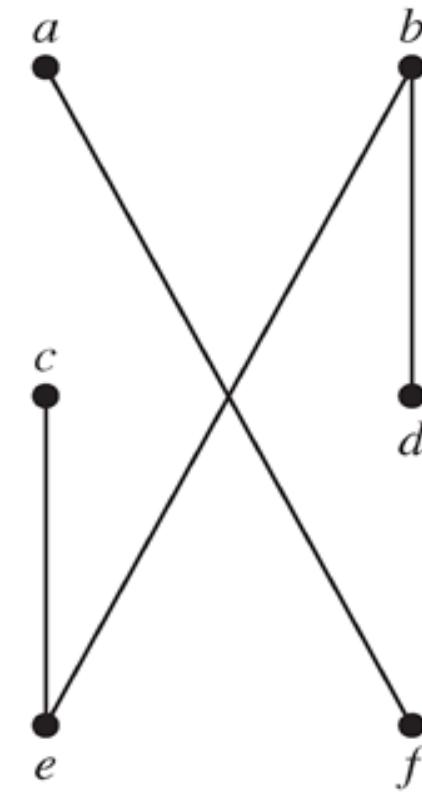
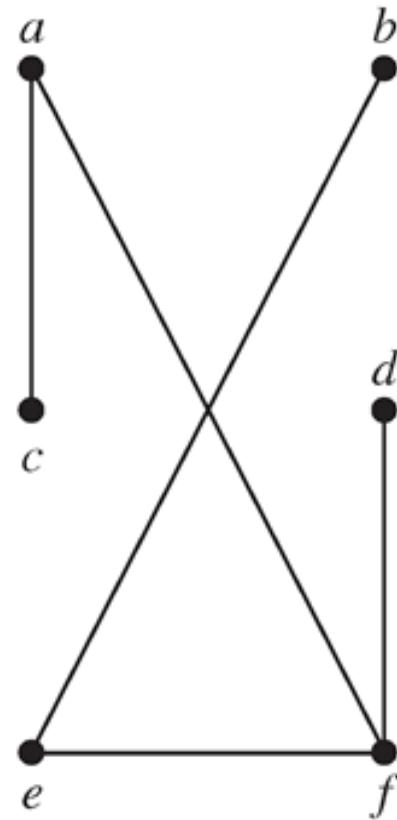
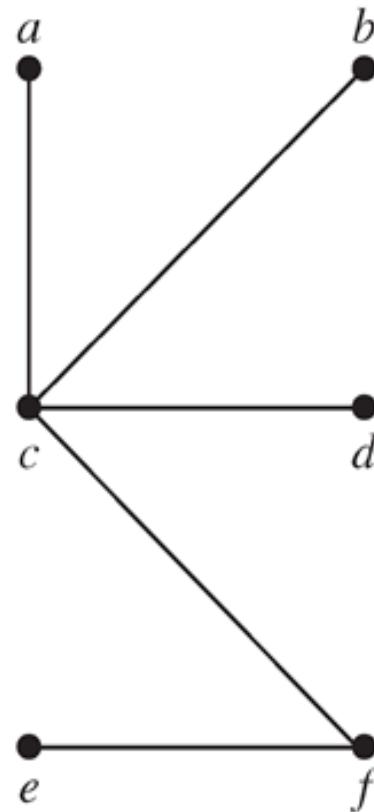
Trees

- **Definition** A *tree* is a connected undirected graph with no simple circuits.



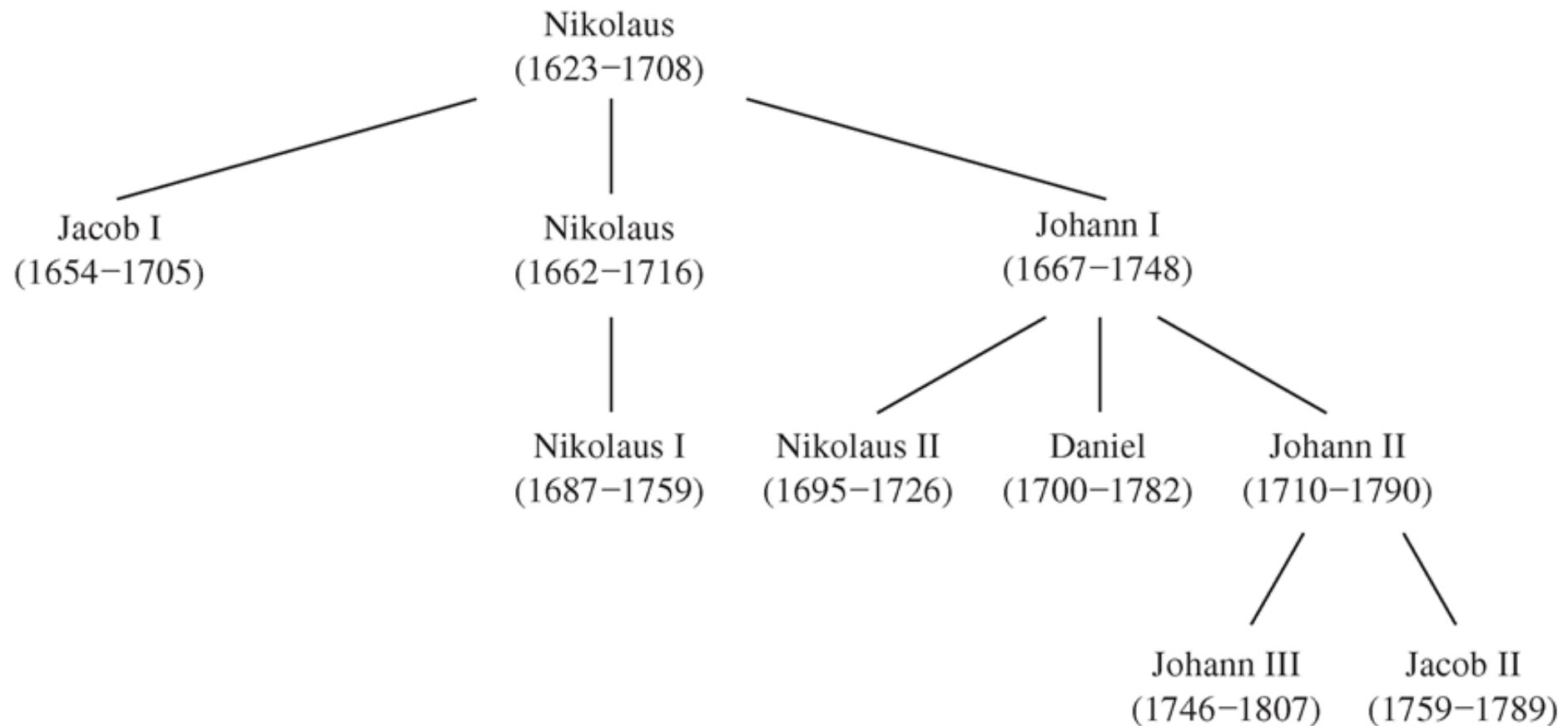
Trees

- **Definition** A *tree* is a connected undirected graph with no simple circuits.



Trees

- **Definition** A *tree* is a connected undirected graph with no simple circuits.



Trees

- **Theorem** An undirected graph is a tree if and only if there is a unique simple path between any two of its vertices.



Trees

- **Theorem** An undirected graph is a tree if and only if there is a unique simple path between any two of its vertices.

Proof



Trees

- **Theorem** An undirected graph is a tree if and only if there is a unique simple path between any two of its vertices.

Proof

Two properties of tree: connected, no circuit

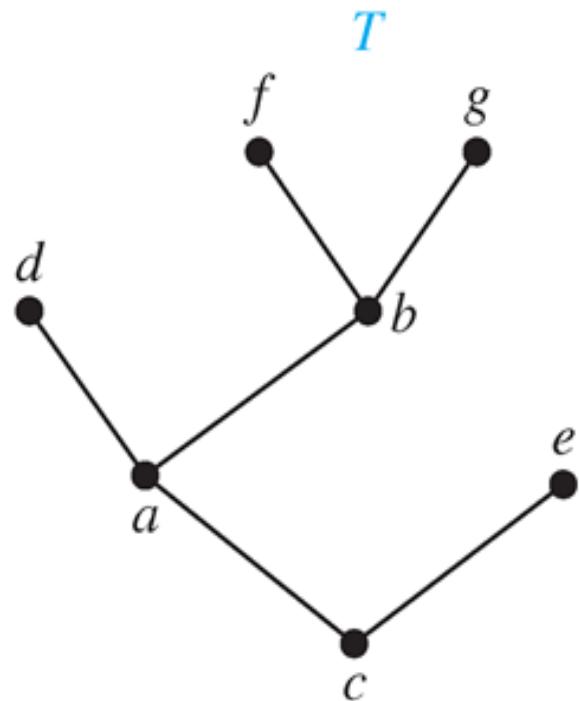
Rooted Trees

- **Definition** A *rooted tree* is a tree in which one vertex has been designated as the **root** and every edge is directed away from the root.



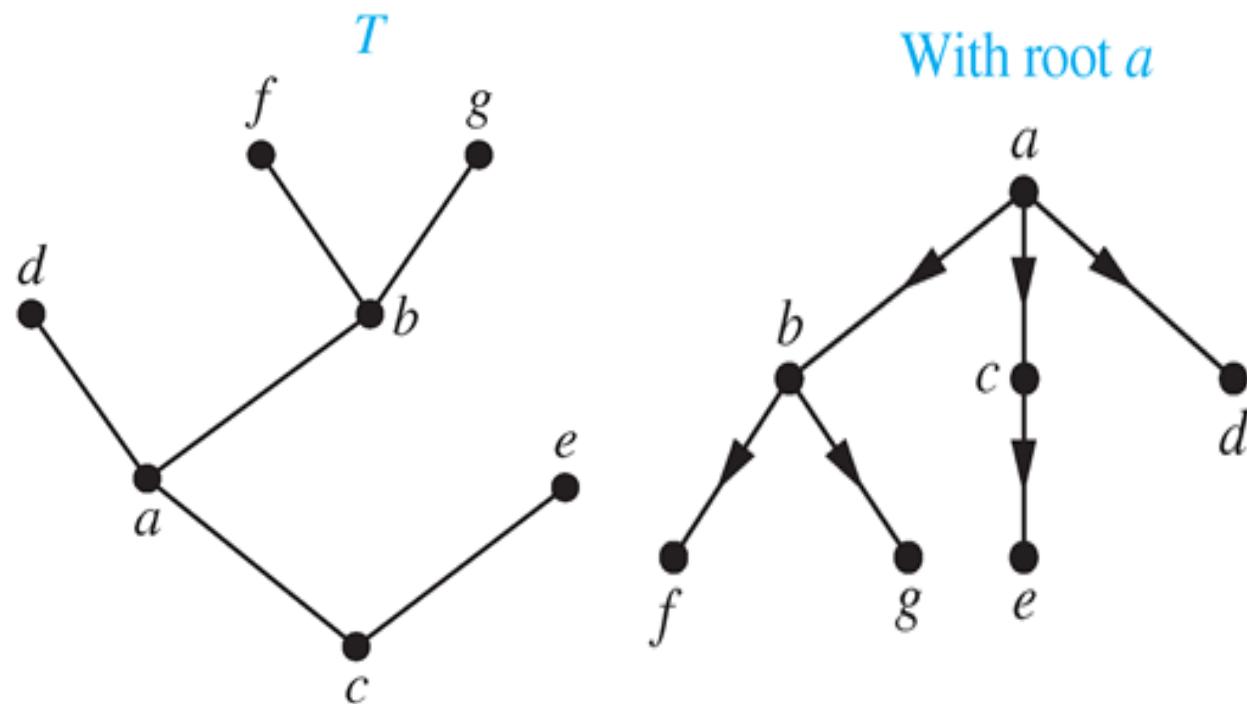
Rooted Trees

- **Definition** A *rooted tree* is a tree in which one vertex has been designated as the *root* and every edge is directed away from the root.



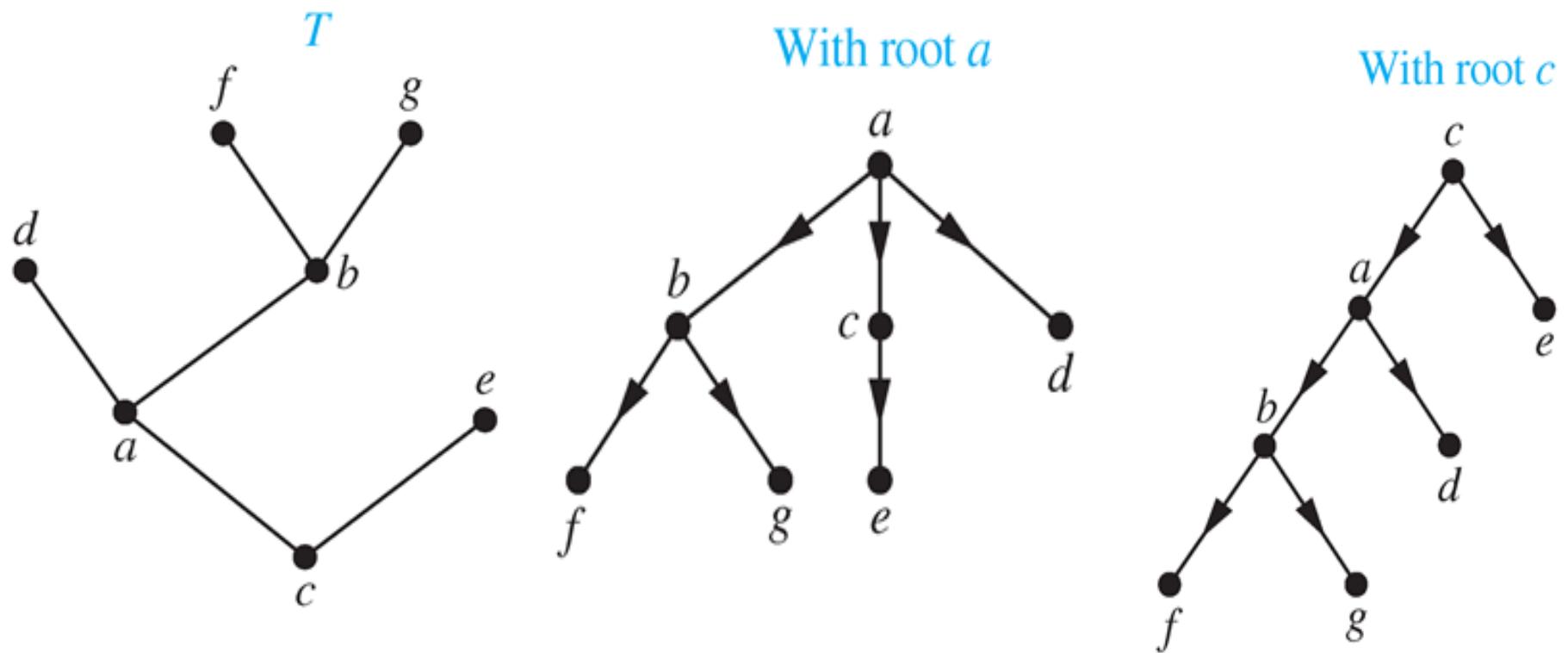
Rooted Trees

- **Definition** A *rooted tree* is a tree in which one vertex has been designated as the *root* and every edge is directed away from the root.



Rooted Trees

- **Definition** A *rooted tree* is a tree in which one vertex has been designated as the *root* and every edge is directed away from the root.



Rooted Trees

- *parent, child, sibling*

Rooted Trees

- *parent, child, sibling
ancestor, descendant*

Rooted Trees

- *parent, child, sibling*

- ancestor, descendant*

- leaf, internal vertex*

Rooted Trees

- *parent, child, sibling*

- ancestor, descendant*

- leaf, internal vertex*

subtree with a as its root: consists of a and its descendants and all edges incident to these descendants



m-Ary Trees

- **Definition** A rooted tree is called an *m-ary tree* if every internal vertex has **no more than** m children. The tree is called a *full m-ary tree* if every internal vertex has **exactly** m children. In particular, an *m*-ary tree with $m = 2$ is called a *binary tree*.

m-Ary Trees

- **Definition** A rooted tree is called an *m-ary tree* if every internal vertex has **no more than** m children. The tree is called a *full m-ary tree* if every internal vertex has **exactly** m children. In particular, an *m*-ary tree with $m = 2$ is called a *binary tree*.

Definition A binary tree is an *ordered rooted tree* where the children of each internal vertex are ordered. In a binary tree, the first child is called the *left child*, and the second child is called the *right child*.

m-Ary Trees

- **Definition** A rooted tree is called an *m-ary tree* if every internal vertex has **no more than** m children. The tree is called a *full m-ary tree* if every internal vertex has **exactly** m children. In particular, an *m*-ary tree with $m = 2$ is called a *binary tree*.

Definition A binary tree is an *ordered rooted tree* where the children of each internal vertex are ordered. In a binary tree, the first child is called the *left child*, and the second child is called the *right child*.

left subtree, right subtree

Counting Vertices in a Full m -Ary Trees

- **Theorem** A full m -ary tree with i internal vertices has $n = mi + 1$ vertices.

Counting Vertices in a Full m -Ary Trees

- **Theorem** A full m -ary tree with i internal vertices has $n = mi + 1$ vertices.

Theorem A full m -ary tree with

- (i) n vertices
- (ii) i internal vertices
- (iii) ℓ leaves

Counting Vertices in a Full m -Ary Trees

- **Theorem** A full m -ary tree with i internal vertices has $n = mi + 1$ vertices.

Theorem A full m -ary tree with

(i) n vertices

(ii) i internal vertices

(iii) ℓ leaves

(i) n vertices, $i = (n - 1)/m$, $\ell = [(m - 1)n + 1]/m$

(ii) i internal vertices, $n = mi + 1$, $\ell = (m - 1)i + 1$

(iii) ℓ leaves, $n = (m\ell - 1)/(m - 1)$, $i = (\ell - 1)/(m - 1)$

Counting Vertices in a Full m -Ary Trees

- **Theorem** A full m -ary tree with i internal vertices has $n = mi + 1$ vertices.

Theorem A full m -ary tree with

(i) n vertices

(ii) i internal vertices

(iii) ℓ leaves

(i) n vertices, $i = (n - 1)/m$, $\ell = [(m - 1)n + 1]/m$

(ii) i internal vertices, $n = mi + 1$, $\ell = (m - 1)i + 1$

(iii) ℓ leaves, $n = (m\ell - 1)/(m - 1)$, $i = (\ell - 1)/(m - 1)$

using $n = mi + 1$ and $n = i + \ell$



Level and Height

- The *level* of a vertex v in a rooted tree is the length of the unique path from the root to this vertex.



Level and Height

- The *level* of a vertex v in a rooted tree is the length of the unique path from the root to this vertex.

The *height* of a rooted tree is the maximum of the levels of the vertices.



Level and Height

- The *level* of a vertex v in a rooted tree is the length of the unique path from the root to this vertex.

The *height* of a rooted tree is the maximum of the levels of the vertices.

Definition A rooted m -ary tree of height h is *balanced* if all leaves are at levels h or $h - 1$. (differ no greater than 1)



The Number of Leaves

- **Theorem** There are at most m^h leaves in an m -ary tree of height h .

The Number of Leaves

- **Theorem** There are **at most** m^h leaves in an m -ary tree of height h .

Proof (by induction)

The Number of Leaves

- **Theorem** There are **at most** m^h leaves in an m -ary tree of height h .

Proof (by induction)

Corollary If an m -ary tree of height h has ℓ leaves, then $h \geq \lceil \log_m \ell \rceil$. If the m -ary tree is **full and balanced**, then $h = \lceil \log_m \ell \rceil$.

The Number of Leaves

- **Theorem** There are **at most** m^h leaves in an m -ary tree of height h .

Proof (by induction)

Corollary If an m -ary tree of height h has ℓ leaves, then $h \geq \lceil \log_m \ell \rceil$. If the m -ary tree is **full and balanced**, then $h = \lceil \log_m \ell \rceil$.

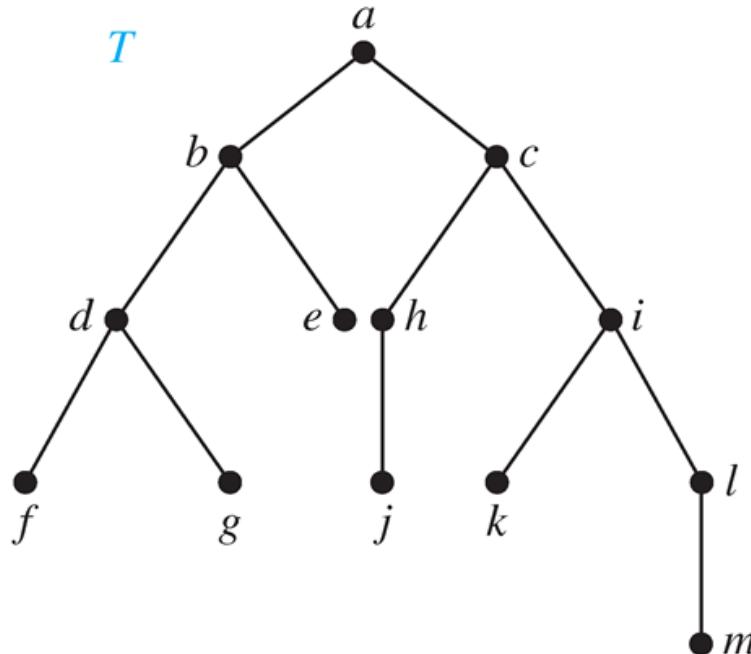
Proof

Binary Trees

- **Definition** A *binary tree* is an *ordered* rooted tree where each internal tree has *two children*, the first is called the *left child* and the second is the *right child*. The tree rooted at the left child of a vertex is called the *left subtree* of this vertex, and the tree rooted at the right child of a vertex is called the *right subtree* of this vertex.

Binary Trees

- **Definition** A *binary tree* is an *ordered* rooted tree where each internal tree has *two children*, the first is called the *left child* and the second is the *right child*. The tree rooted at the left child of a vertex is called the *left subtree* of this vertex, and the tree rooted at the right child of a vertex is called the *right subtree* of this vertex.



Tree Traversal

- The procedures for *systematically* visiting every vertex of an ordered tree are called *traversals*.



Tree Traversal

- The procedures for *systematically* visiting every vertex of an ordered tree are called *traversals*.

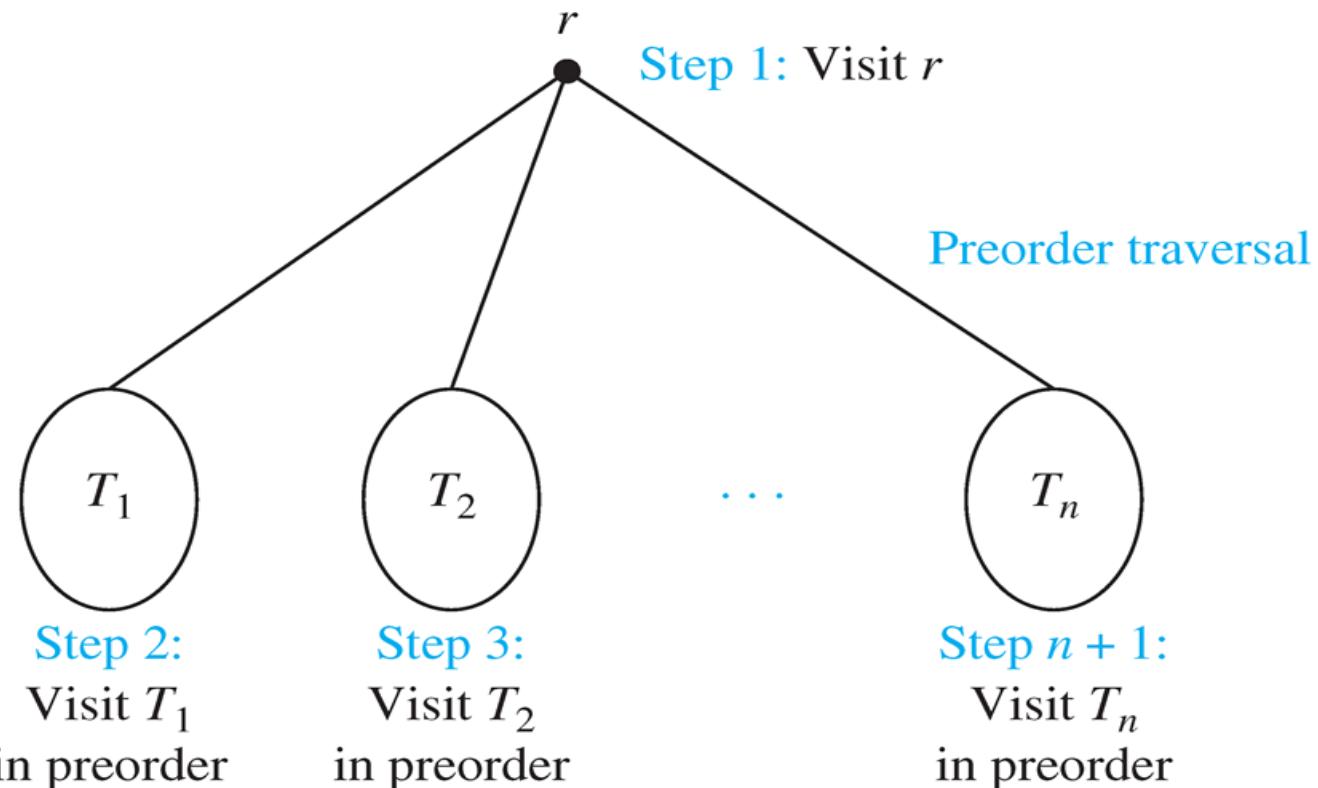
The three most commonly used traversals are *preorder traversal*, *inorder traversal*, *postorder traversal*.

Preorder Traversal

- **Definition** Let T be an ordered rooted tree with root r . If T consists only of r , then r is the *preorder traversal* of T . Otherwise, suppose that T_1, T_2, \dots, T_n are the subtrees of r from left to right in T . The *preorder traversal* begins by visiting r , and continues by traversing T_1 in preorder, then T_2 in preorder, and so on, until T_n is traversed in preorder.

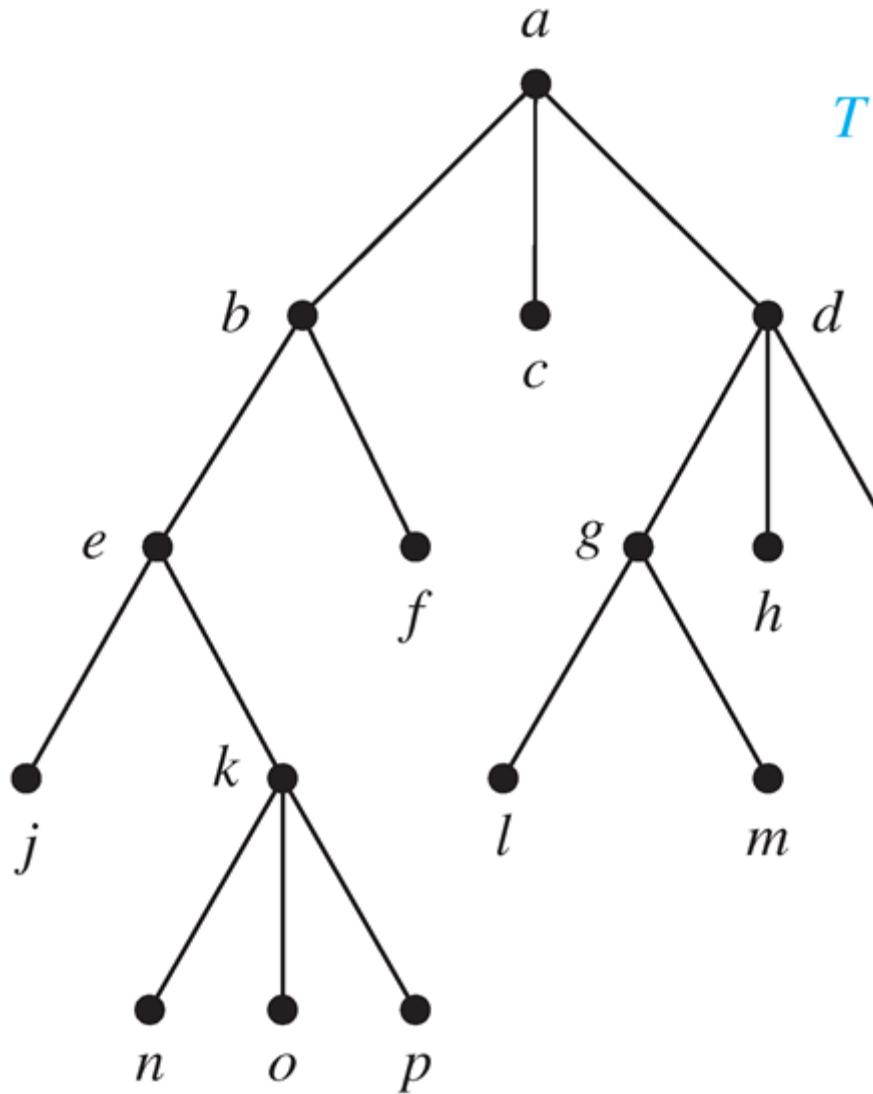
Preorder Traversal

- **Definition** Let T be an ordered rooted tree with root r . If T consists only of r , then r is the *preorder traversal* of T . Otherwise, suppose that T_1, T_2, \dots, T_n are the subtrees of r from left to right in T . The *preorder traversal* begins by visiting r , and continues by traversing T_1 in preorder, then T_2 in preorder, and so on, until T_n is traversed in preorder.



Preorder Traversal

■ Example



Preorder Traversal

```
procedure preorder ( $T$ : ordered rooted tree)  
 $r :=$  root of  $T$   
list  $r$   
for each child  $c$  of  $r$  from left to right  
     $T(c) :=$  subtree with  $c$  as root  
    preorder( $T(c)$ )
```

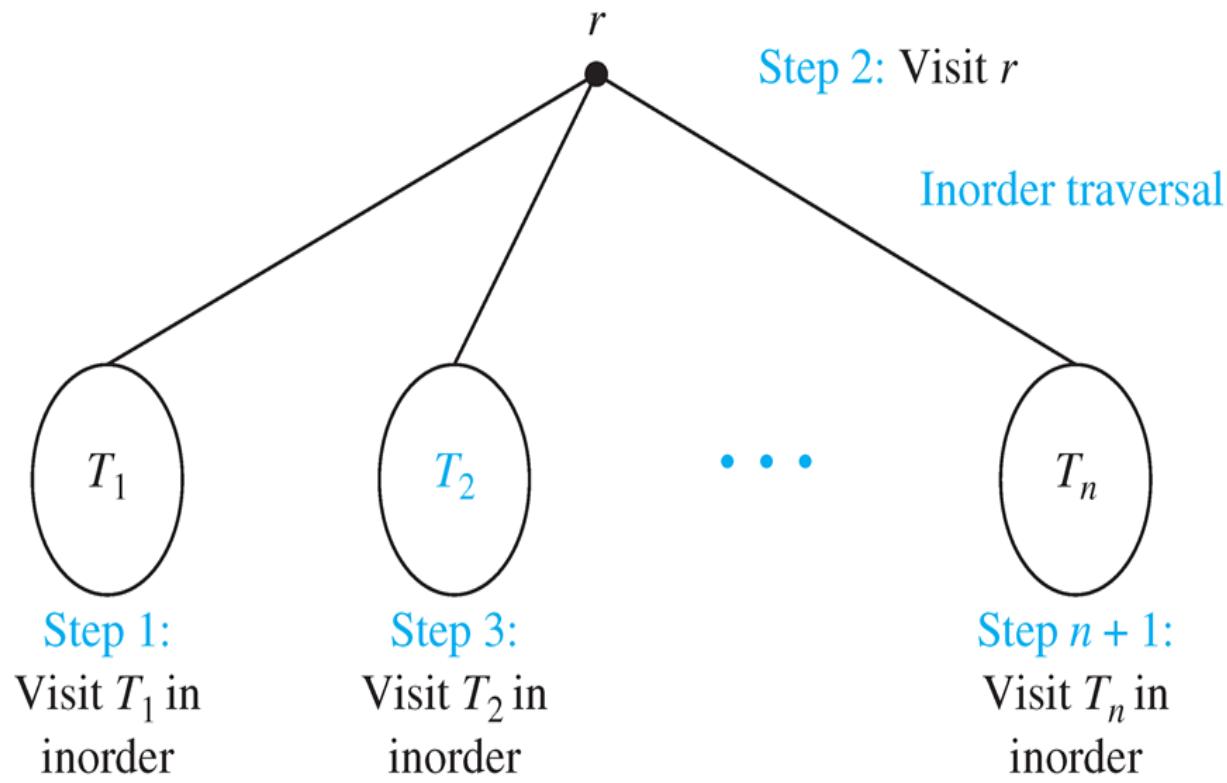


Inorder Traversal

- **Definition** Let T be an ordered rooted tree with root r . If T consists only of r , then r is the *inorder traversal* of T . Otherwise, suppose that T_1, T_2, \dots, T_n are the subtrees of r from left to right in T . The *inorder traversal* begins by traversing T_1 **in inorder**, then visiting r , and continues by traversing T_2 in inorder, and so on, until T_n is traversed in inorder.

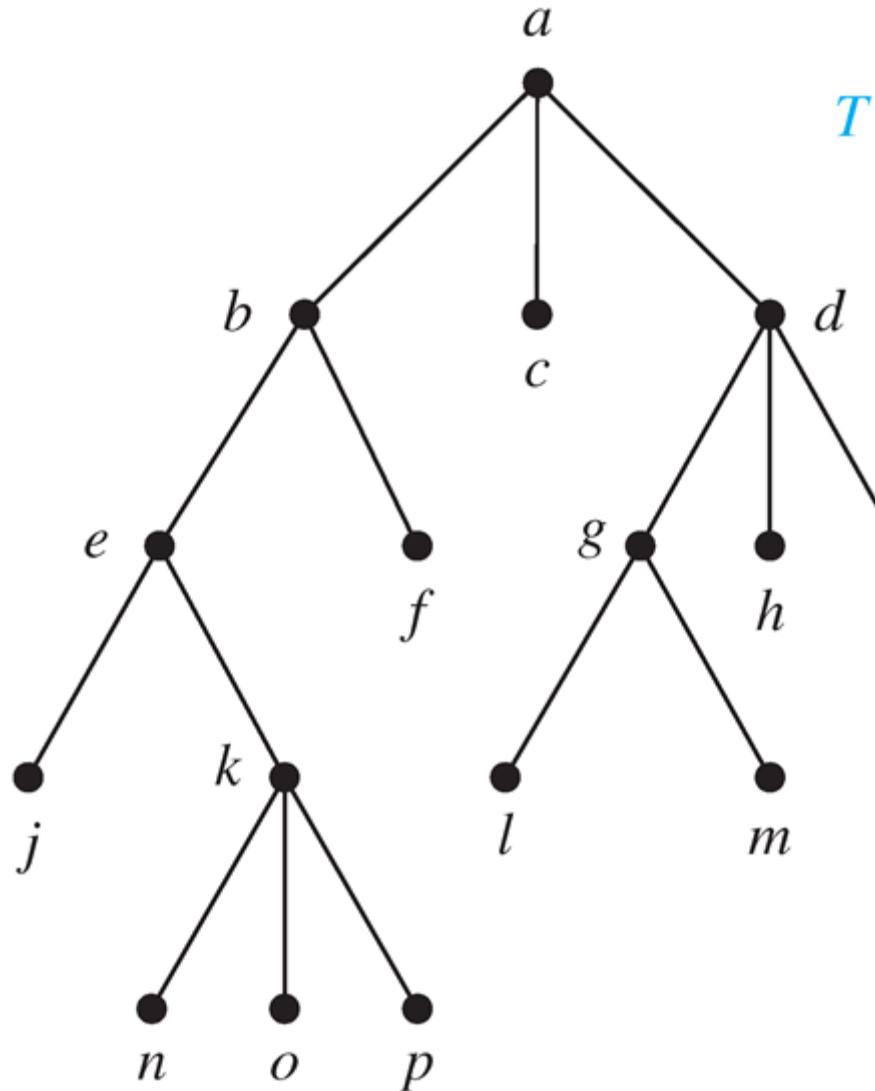
Inorder Traversal

- **Definition** Let T be an ordered rooted tree with root r . If T consists only of r , then r is the *inorder traversal* of T . Otherwise, suppose that T_1, T_2, \dots, T_n are the subtrees of r from left to right in T . The *inorder traversal* begins by traversing T_1 **in inorder**, then visiting r , and continues by traversing T_2 in inorder, and so on, until T_n is traversed in inorder.



Inorder Traversal

Example



T

Inorder Traversal

```
procedure inorder ( $T$ : ordered rooted tree)
```

```
     $r :=$  root of  $T$ 
```

```
    if  $r$  is a leaf then list  $r$ 
```

```
    else
```

```
         $l :=$  first child of  $r$  from left to right
```

```
         $T(l) :=$  subtree with  $l$  as its root
```

```
        inorder( $T(l)$ )
```

```
        list( $r$ )
```

```
        for each child  $c$  of  $r$  from left to right
```

```
             $T(c) :=$  subtree with  $c$  as root
```

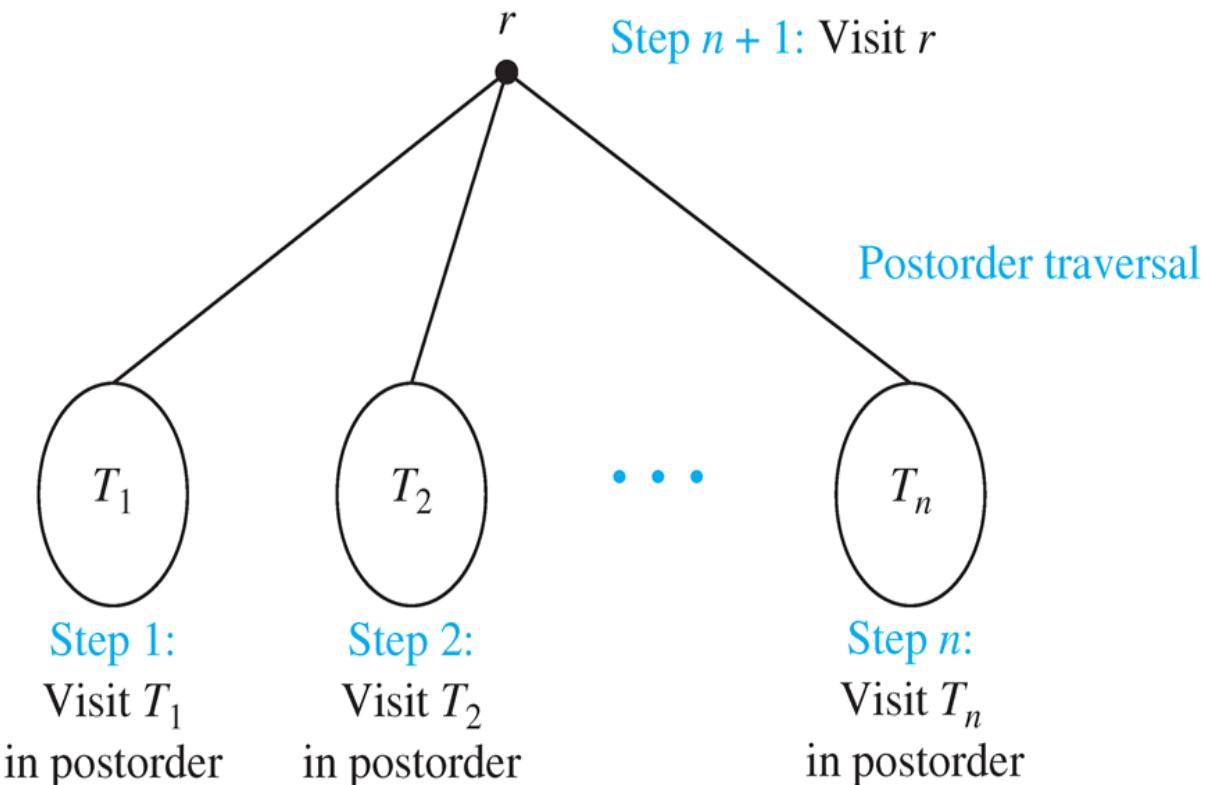
```
            inorder( $T(c)$ )
```

Postorder Traversal

- **Definition** Let T be an ordered rooted tree with root r . If T consists only of r , then r is the *postorder traversal* of T . Otherwise, suppose that T_1, T_2, \dots, T_n are the subtrees of r from left to right in T . The *postorder traversal* begins by traversing T_1 **in postorder**, then T_2 in postorder, and so on, **after** T_n is traversed in postorder, r is visited.

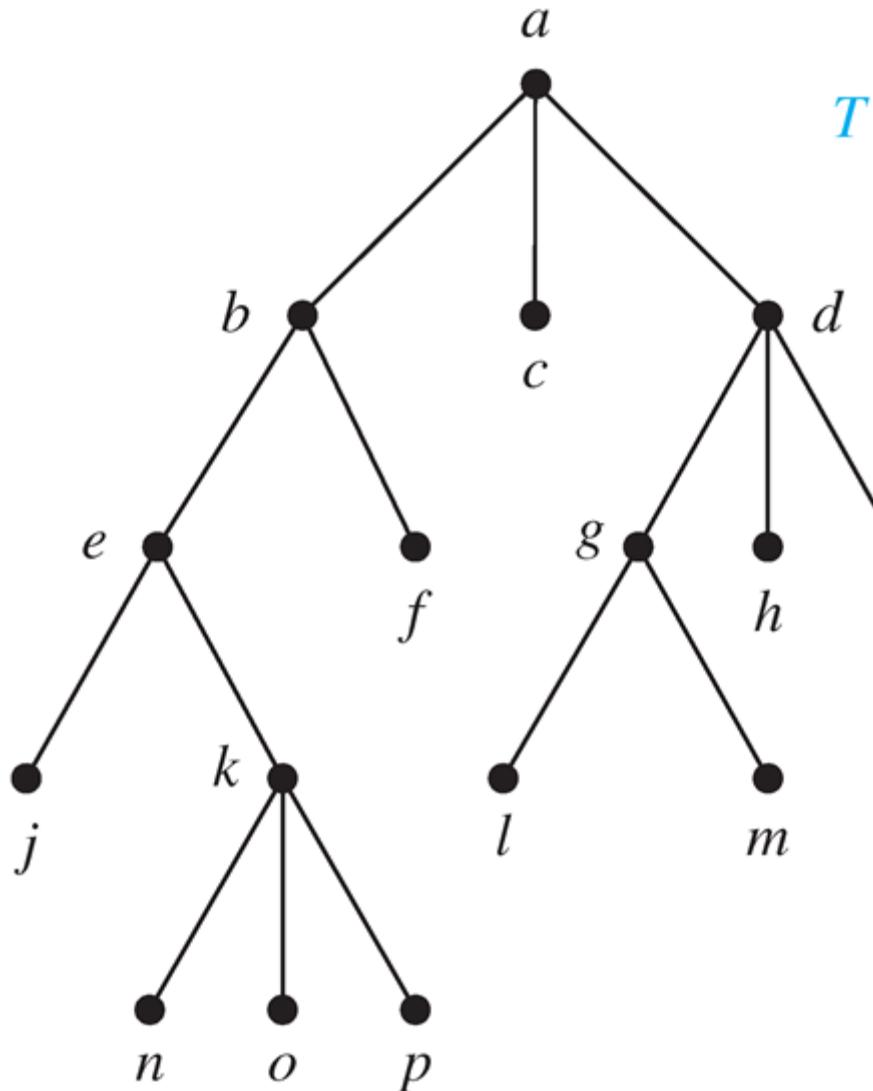
Postorder Traversal

- **Definition** Let T be an ordered rooted tree with root r . If T consists only of r , then r is the *postorder traversal* of T . Otherwise, suppose that T_1, T_2, \dots, T_n are the subtrees of r from left to right in T . The *postorder traversal* begins by traversing T_1 in postorder, then T_2 in postorder, and so on, after T_n is traversed in postorder, r is visited.



Postorder Traversal

■ Example

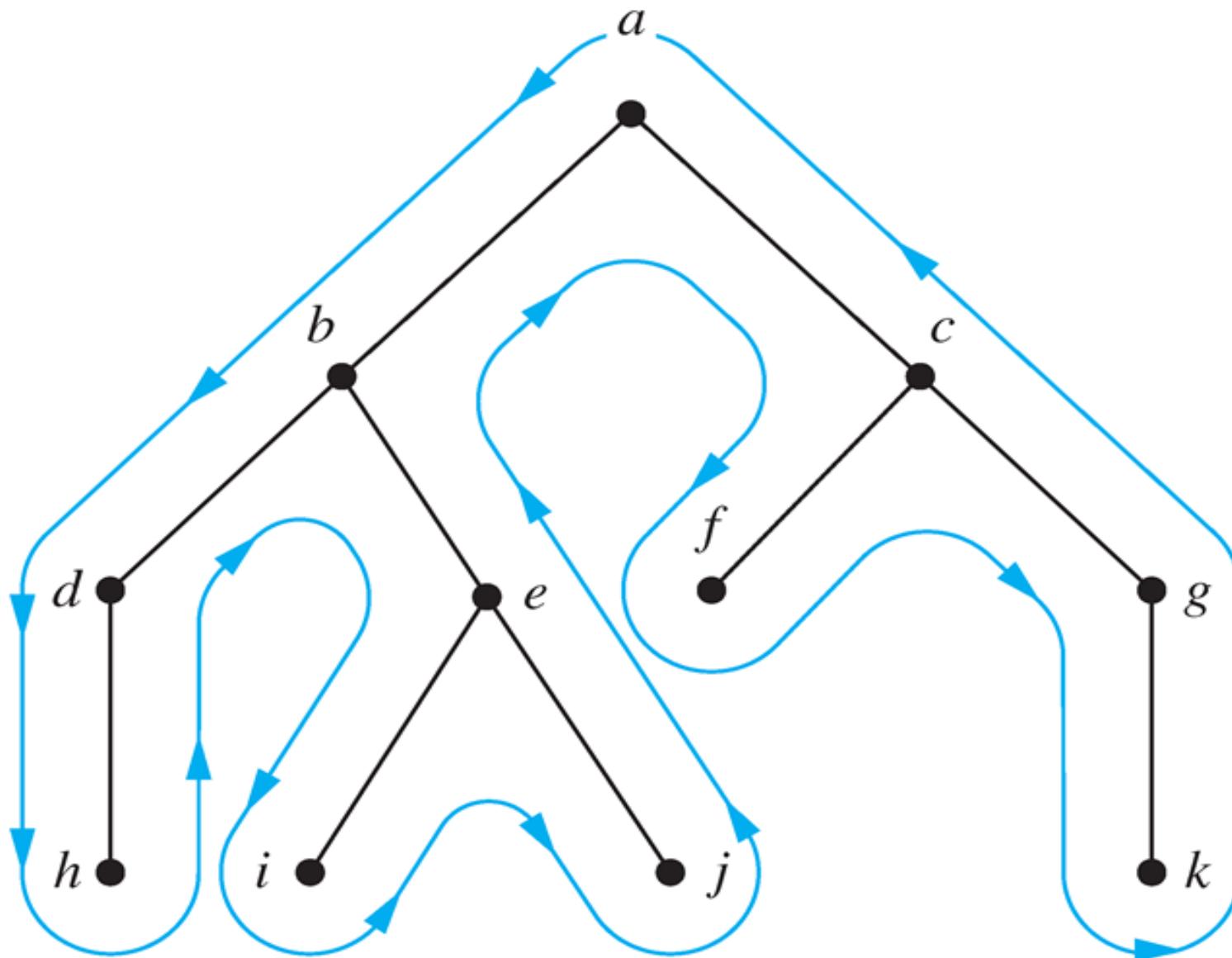


Postorder Traversal

```
procedure postordered ( $T$ : ordered rooted tree)
 $r :=$  root of  $T$ 
for each child  $c$  of  $r$  from left to right
     $T(c) :=$  subtree with  $c$  as root
    postorder( $T(c)$ )
list  $r$ 
```



Preorder, Inorder, Postorder Traversal



Expression Trees

- Complex expressions can be represented using ordered rooted trees



Expression Trees

- Complex expressions can be represented using ordered rooted trees

Example

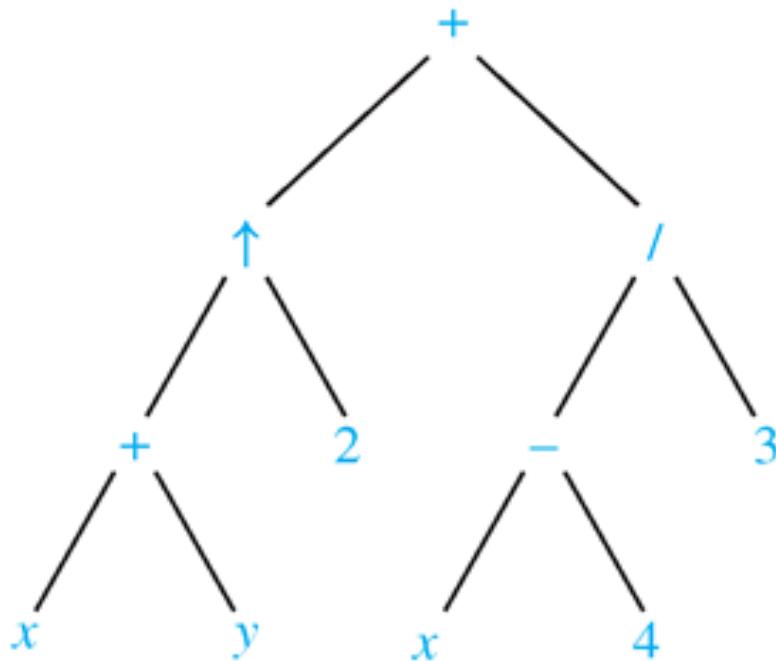
consider the expression $((x + y) \uparrow 2) + ((x - 4)/3)$

Expression Trees

- Complex expressions can be represented using ordered rooted trees

Example

consider the expression $((x + y) \uparrow 2) + ((x - 4)/3)$



Infix Notation

- An **inorder traversal** of the tree representing an expression produces the **original expression** when **parentheses are included** except for unary operation.



Infix Notation

- An **inorder traversal** of the tree representing an expression produces the **original expression** when **parentheses are included** except for unary operation.

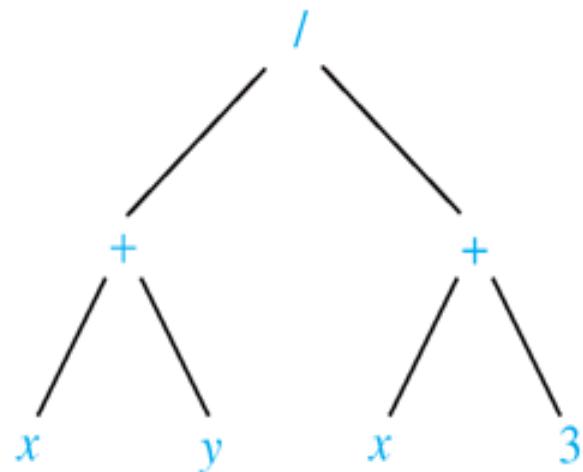
Why parentheses are needed?



Infix Notation

- An **inorder traversal** of the tree representing an expression produces the **original expression** when **parentheses are included** except for unary operation.

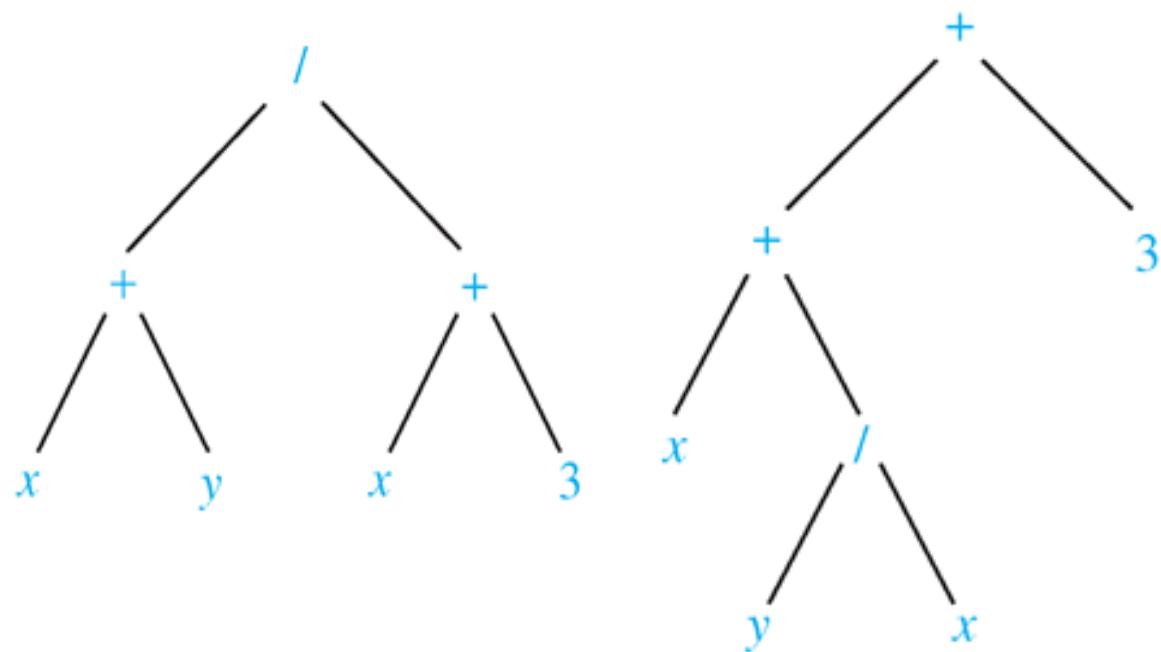
Why parentheses are needed?



Infix Notation

- An **inorder traversal** of the tree representing an expression produces the **original expression** when **parentheses are included** except for unary operation.

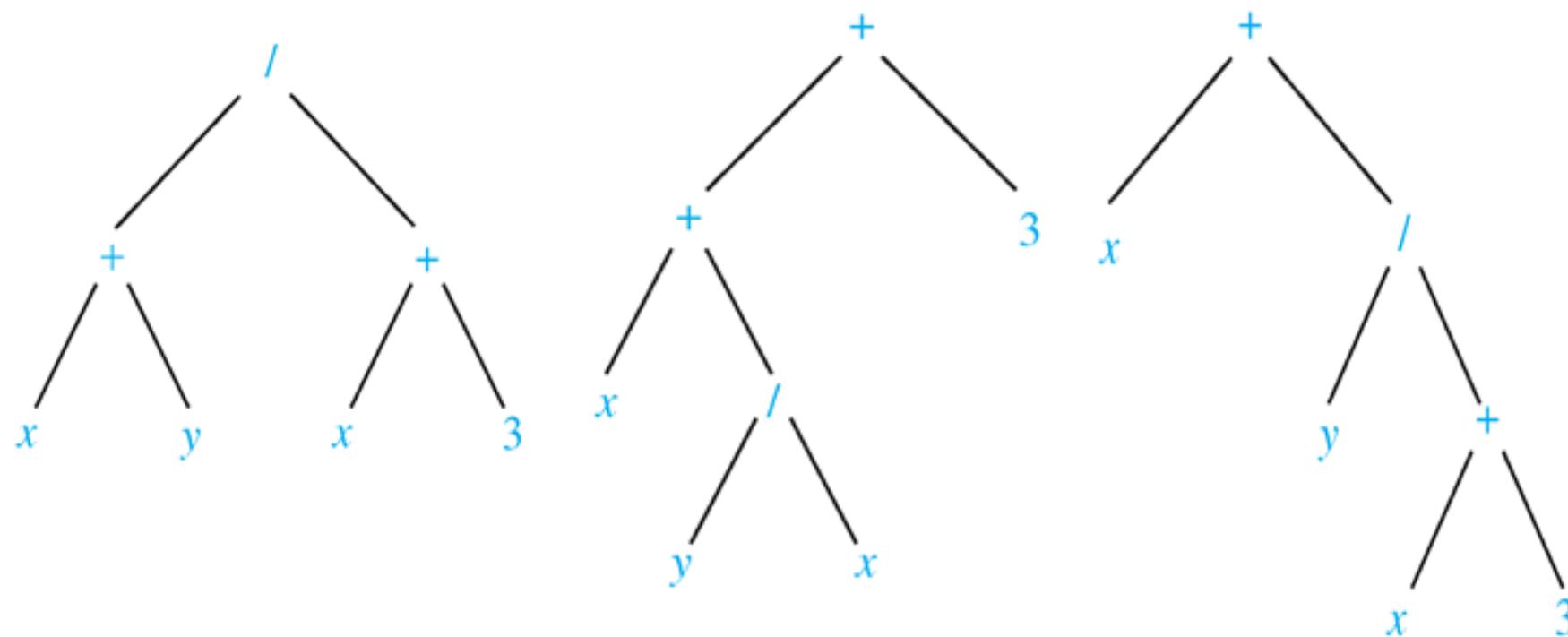
Why parentheses are needed?



Infix Notation

- An **inorder traversal** of the tree representing an expression produces the **original expression** when **parentheses are included** except for unary operation.

Why parentheses are needed?



Prefix Notation

- The **preorder traversal** of expression trees leads to the ***prefix form*** of the expression (*Polish notation*).

Prefix Notation

- The **preorder traversal** of expression trees leads to the **prefix form** of the expression (*Polish notation*).

Operators **precede** their operands in the **prefix notation**.

Parentheses are **not needed** as the representation is **unambiguous**.



Prefix Notation

- The **preorder traversal** of expression trees leads to the **prefix form** of the expression (*Polish notation*).

Operators **precede** their operands in the **prefix notation**.
Parentheses are **not needed** as the representation is **unambiguous**.

Prefix expressions are evaluated by working **from right to left**. When we encounter an operator, we perform the operation with **the two operands to the right**.



Prefix Notation

■ Example

+ - * 2 3 5 / ↑ 2 3 4

Prefix Notation

■ Example

+ - * 2 3 5 / ↑ 2 3 4

$$\begin{array}{ccccccccc} + & - & * & 2 & 3 & 5 & / & \uparrow & 2 \\ & & & & & & & & 3 \\ & & & & & & & \text{---} & 4 \\ & & & & & & & & \\ & & & & & & & 2 \uparrow 3 = 8 & \end{array}$$

$$\begin{array}{ccccccccc} + & - & * & 2 & 3 & 5 & / & 8 & 4 \\ & & & & & & \text{---} & & \\ & & & & & & & 8 & \\ & & & & & & & / & 4 \\ & & & & & & & & \\ & & & & & & & 8 / 4 = 2 & \end{array}$$

$$\begin{array}{ccccccccc} + & - & * & 2 & 3 & 5 & 2 \\ & & \text{---} & & & & \\ & & & 2 * 3 = 6 & & & \end{array}$$

$$\begin{array}{ccccccccc} + & - & 6 & 5 & 2 \\ & & \text{---} & & & \\ & & & 6 - 5 = 1 & & & \end{array}$$

$$\begin{array}{ccccccccc} + & 1 & 2 \\ & \text{---} & & \\ & & 1 + 2 = 3 & & & \end{array}$$

Postfix Notation

- The **postorder traversal** of expression trees leads to the *postfix form* of the expression (*reverse Polish notation*).



Postfix Notation

- The postorder traversal of expression trees leads to the *postfix form* of the expression (*reverse Polish notation*).

Operators **follow** their operands in the postfix notation.
Parentheses are **not needed** as the representation is unambiguous.



Postfix Notation

- The postorder traversal of expression trees leads to the *postfix form* of the expression (*reverse Polish notation*).

Operators **follow** their operands in the **postfix notation**.
Parentheses are **not needed** as the representation is **unambiguous**.

Postfix expressions are evaluated by working **from left to right**. When we encounter an operator, we perform the operation with **the two operands to the left**.

Postfix Notation

■ Example

7 2 3 * - 4 ↑ 9 3 / +

Postfix Notation

■ Example

7 2 3 * - 4 ↑ 9 3 / +

7 2 3 * - 4 ↑ 9 3 / +

$$2 * 3 = 6$$

7 6 - 4 ↑ 9 3 / +

$$7 - 6 = 1$$

1 4 ↑ 9 3 / +

$$1^4 = 1$$

1 9 3 / +

$$9 / 3 = 3$$

1 3 +

$$1 + 3 = 4$$



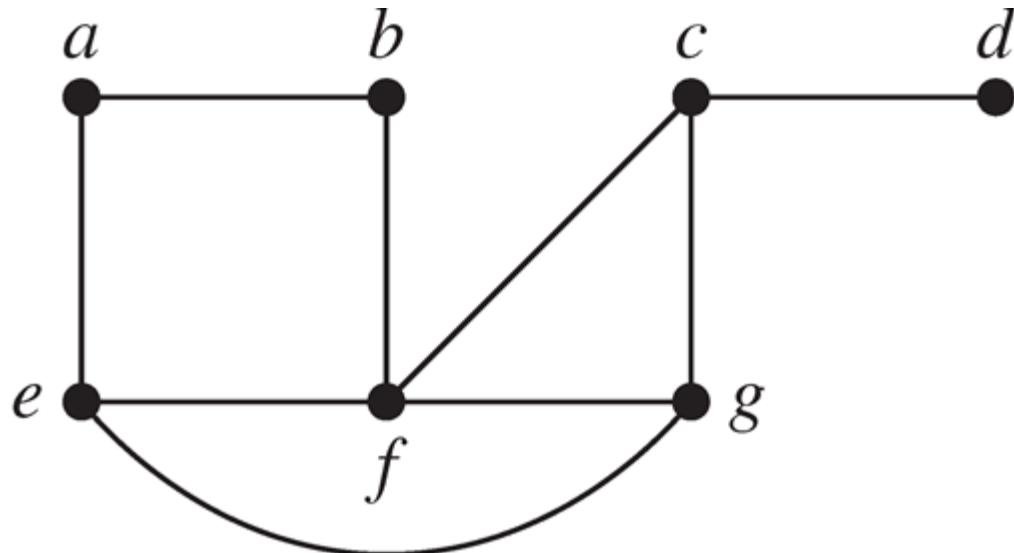
Spanning Trees

- **Definition** Let G be a simple graph. A *spanning tree* of G is a subgraph of G that **is** a tree containing **every vertex** of G .



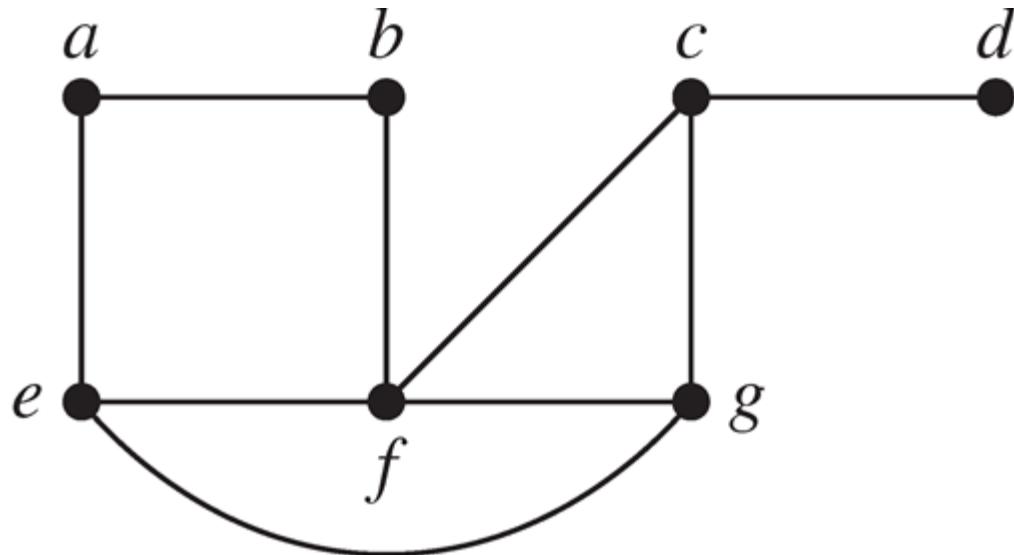
Spanning Trees

- **Definition** Let G be a simple graph. A *spanning tree* of G is a subgraph of G that **is a tree containing every vertex of G .**



Spanning Trees

- **Definition** Let G be a simple graph. A *spanning tree* of G is a subgraph of G that **is a tree containing every vertex of G .**



remove edges to avoid circuits

Spanning Trees

- **Theorem** A simple graph is **connected** if and only if it has a **spanning tree**.

Spanning Trees

- **Theorem** A simple graph is **connected** if and only if it has a spanning tree.

Proof

Spanning Trees

- **Theorem** A simple graph is **connected** if and only if it has a **spanning tree**.

Proof

“only if ” part

Spanning Trees

- **Theorem** A simple graph is **connected** if and only if it has a **spanning tree**.

Proof

“only if ” part

The **spanning tree** can be obtained by removing edges from **simple circuits**.



Spanning Trees

- **Theorem** A simple graph is **connected** if and only if it has a **spanning tree**.

Proof

“only if ” part

The **spanning tree** can be obtained by removing edges from **simple circuits**.

“if ” part

Spanning Trees

- **Theorem** A simple graph is **connected** if and only if it has a **spanning tree**.

Proof

“only if ” part

The **spanning tree** can be obtained by removing edges from **simple circuits**.

“if ” part

easy

Depth-First Search

- We can find spanning trees by removing edges from simple circuits.

Depth-First Search

- We can find spanning trees by removing edges from simple circuits.

But, this is **inefficient**, since simple circuits should be identified first.



Depth-First Search

- We can find spanning trees by removing edges from simple circuits.

But, this is inefficient, since simple circuits should be identified first.

Instead, we build up spanning trees by successively adding edges.



Depth-First Search

- We can find spanning trees by removing edges from simple circuits.

But, this is inefficient, since simple circuits should be identified first.

Instead, we build up spanning trees by successively adding edges.

- ◊ First arbitrarily choose a vertex of the graph as the root.



Depth-First Search

- We can find spanning trees by removing edges from simple circuits.

But, this is inefficient, since simple circuits should be identified first.

Instead, we build up spanning trees by successively adding edges.

- ◊ First arbitrarily choose a vertex of the graph as the root.
- ◊ Form a path by successively adding vertices and edges. Continue adding to this path as long as possible.



Depth-First Search

- We can find spanning trees by removing edges from simple circuits.

But, this is inefficient, since simple circuits should be identified first.

Instead, we build up spanning trees by successively adding edges.

- ◊ First arbitrarily choose a vertex of the graph as the root.
- ◊ Form a path by successively adding vertices and edges. Continue adding to this path as long as possible.
- ◊ If the path goes through all vertices of the graph, the tree is a spanning tree.

Depth-First Search

- We can find spanning trees by removing edges from simple circuits.

But, this is inefficient, since simple circuits should be identified first.

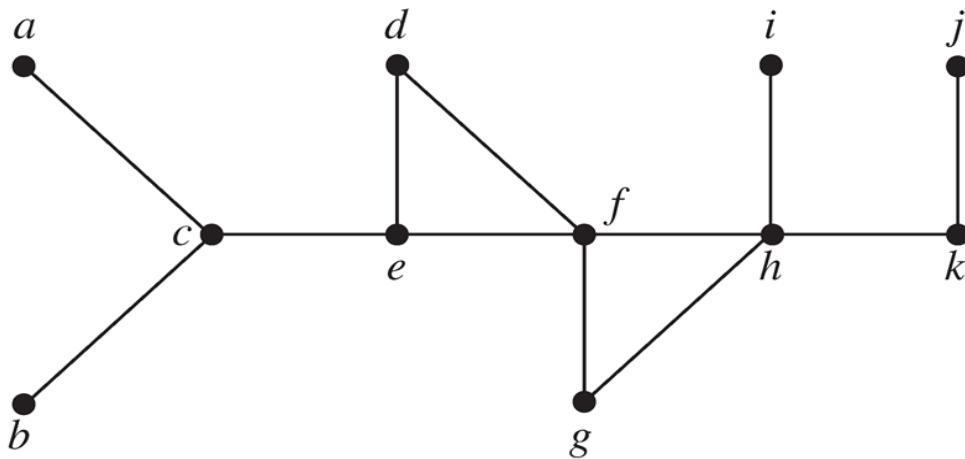
Instead, we build up spanning trees by successively adding edges.

- ◊ First arbitrarily choose a vertex of the graph as the root.
- ◊ Form a path by successively adding vertices and edges. Continue adding to this path as long as possible.
- ◊ If the path goes through all vertices of the graph, the tree is a spanning tree.
- ◊ Otherwise, move back to some vertex to repeat this procedure (*backtracking*)



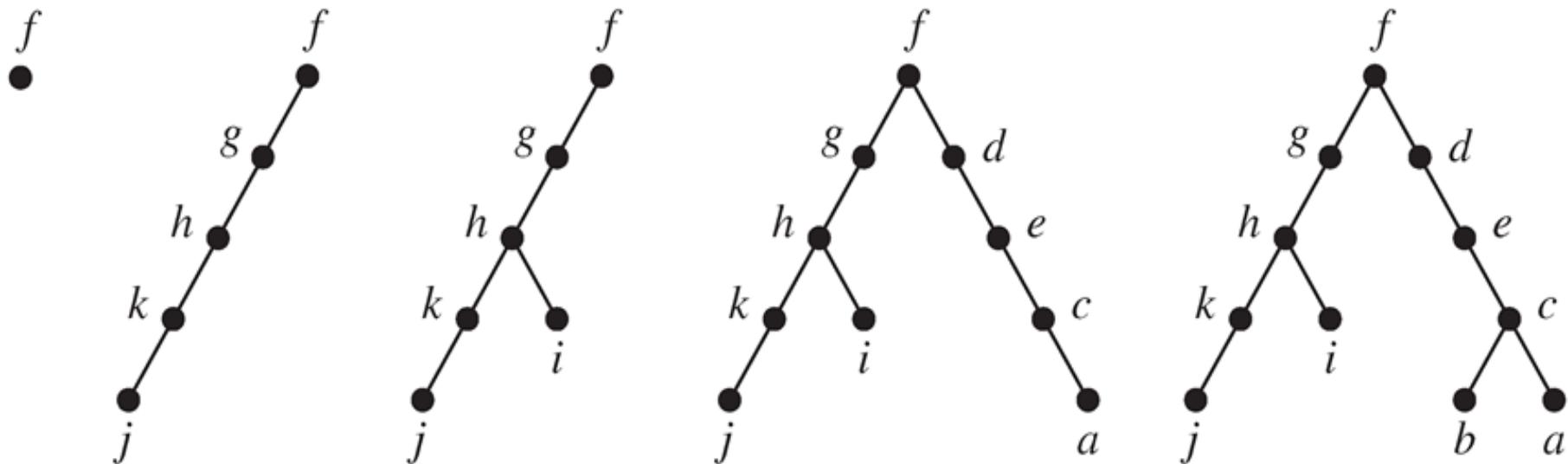
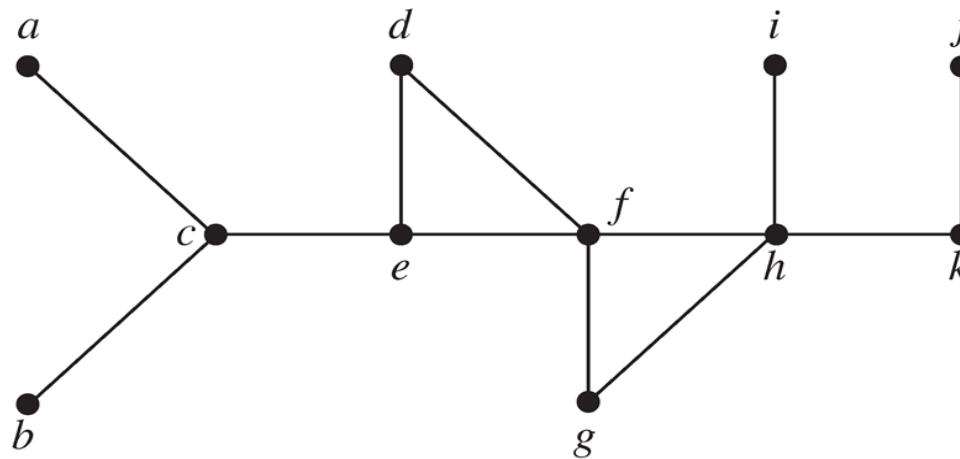
Depth-First Search

■ Example



Depth-First Search

Example



Depth-First Search Algorithm

```
procedure DFS( $G$ : connected graph with vertices  $v_1, v_2, \dots, v_n$ )
 $T :=$  tree consisting only of the vertex  $v_1$ 
visit( $v_1$ )
```

```
procedure visit( $v$ : vertex of  $G$ )
for each vertex  $w$  adjacent to  $v$  and not yet in  $T$ 
    add vertex  $w$  and edge  $\{v, w\}$  to  $T$ 
    visit( $w$ )
```



Depth-First Search Algorithm

```
procedure DFS( $G$ : connected graph with vertices  $v_1, v_2, \dots, v_n$ )
 $T :=$  tree consisting only of the vertex  $v_1$ 
visit( $v_1$ )
```

```
procedure visit( $v$ : vertex of  $G$ )
for each vertex  $w$  adjacent to  $v$  and not yet in  $T$ 
    add vertex  $w$  and edge  $\{v, w\}$  to  $T$ 
    visit( $w$ )
```

time complexity: $O(e)$

Breadth-First Search

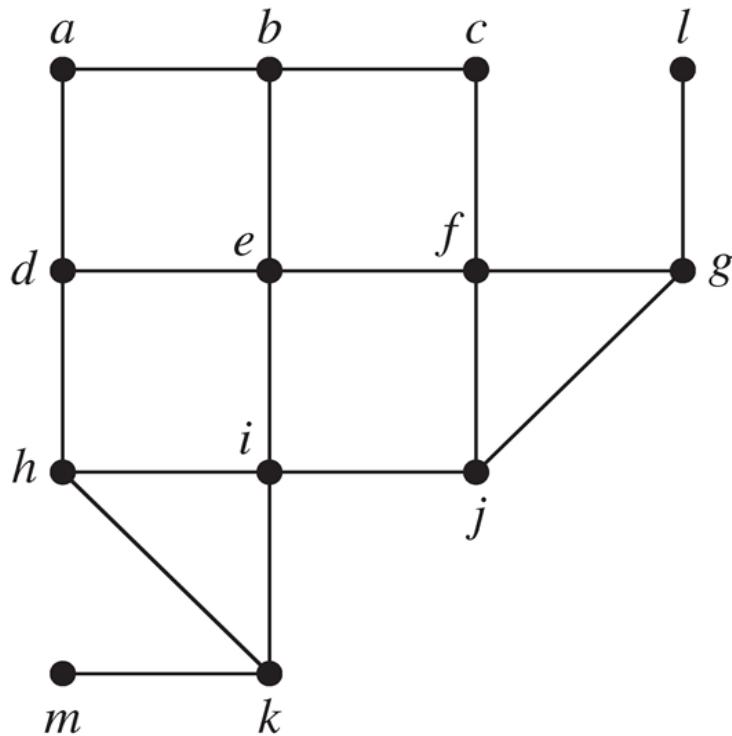
- This is the **second** algorithm that we build up **spanning trees** by **successively adding edges**.

Breadth-First Search

- This is the **second** algorithm that we build up **spanning trees** by **successively adding edges**.
 - ◊ First arbitrarily choose a vertex of the graph as the root.
 - ◊ Form a path by **adding all edges incident to this vertex and the other endpoint of each of these edges**
 - ◊ For each vertex added at the **previous level**, **add edge incident to this vertex**, as long as it does **not** produce a simple circuit.
 - ◊ Continue in this manner until **all vertices have been added**.

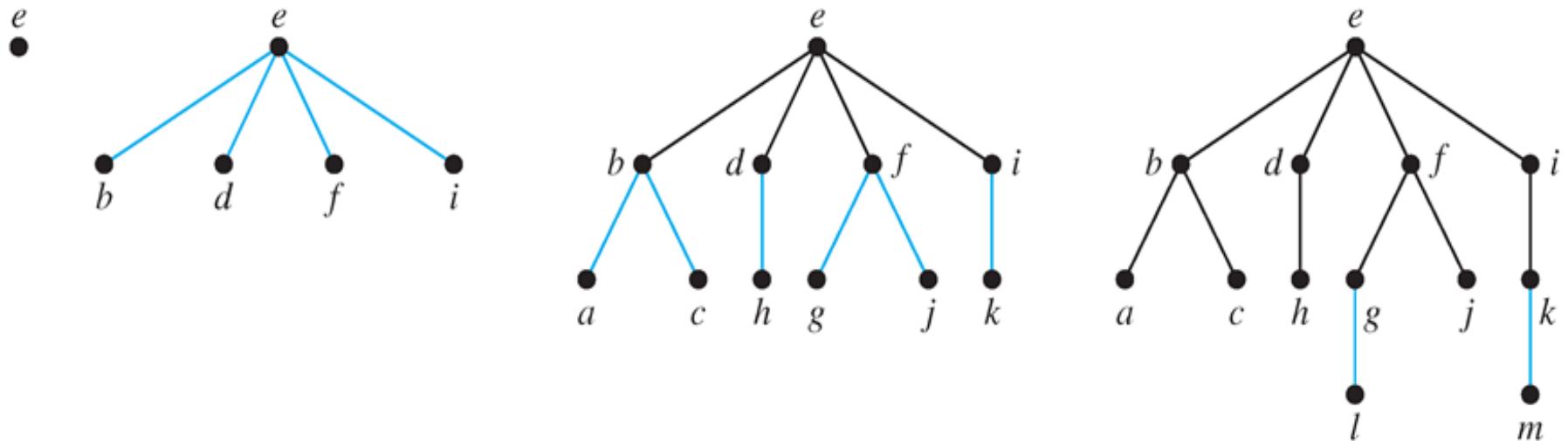
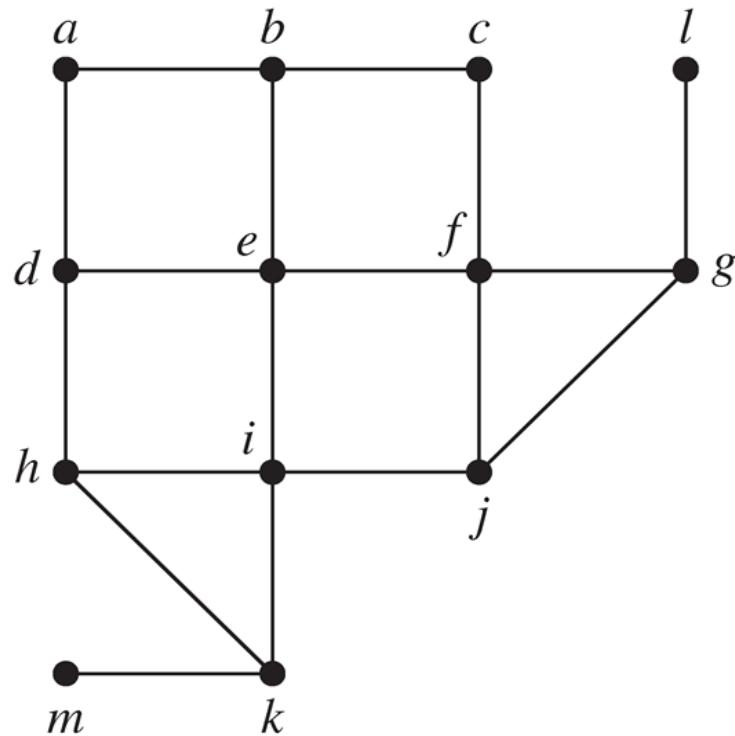
Breadth-First Search

- **Example**



Breadth-First Search

Example



Breadth-First Search

```
procedure BFS( $G$ : connected graph with vertices  $v_1, v_2, \dots, v_n$ )  
     $T :=$  tree consisting only of the vertex  $v_1$   
     $L :=$  empty list  $visit(v_1)$   
    put  $v_1$  in the list  $L$  of unprocessed vertices  
    while  $L$  is not empty  
        remove the first vertex,  $v$ , from  $L$   
        for each neighbor  $w$  of  $v$   
            if  $w$  is not in  $L$  and not in  $T$  then  
                add  $w$  to the end of the list  $L$   
                add  $w$  and edge  $\{v,w\}$  to  $T$ 
```

Breadth-First Search

```
procedure BFS( $G$ : connected graph with vertices  $v_1, v_2, \dots, v_n$ )  
     $T :=$  tree consisting only of the vertex  $v_1$   
     $L :=$  empty list  $visit(v_1)$   
    put  $v_1$  in the list  $L$  of unprocessed vertices  
    while  $L$  is not empty  
        remove the first vertex,  $v$ , from  $L$   
        for each neighbor  $w$  of  $v$   
            if  $w$  is not in  $L$  and not in  $T$  then  
                add  $w$  to the end of the list  $L$   
                add  $w$  and edge  $\{v,w\}$  to  $T$ 
```

time complexity: $O(e)$

Applications of DFS, BFS

- find paths, circuits, connected components, cut vertices, ...



Applications of DFS, BFS

- find paths, circuits, connected components, cut vertices, ...
find shortest paths, determine whether bipartite, ...



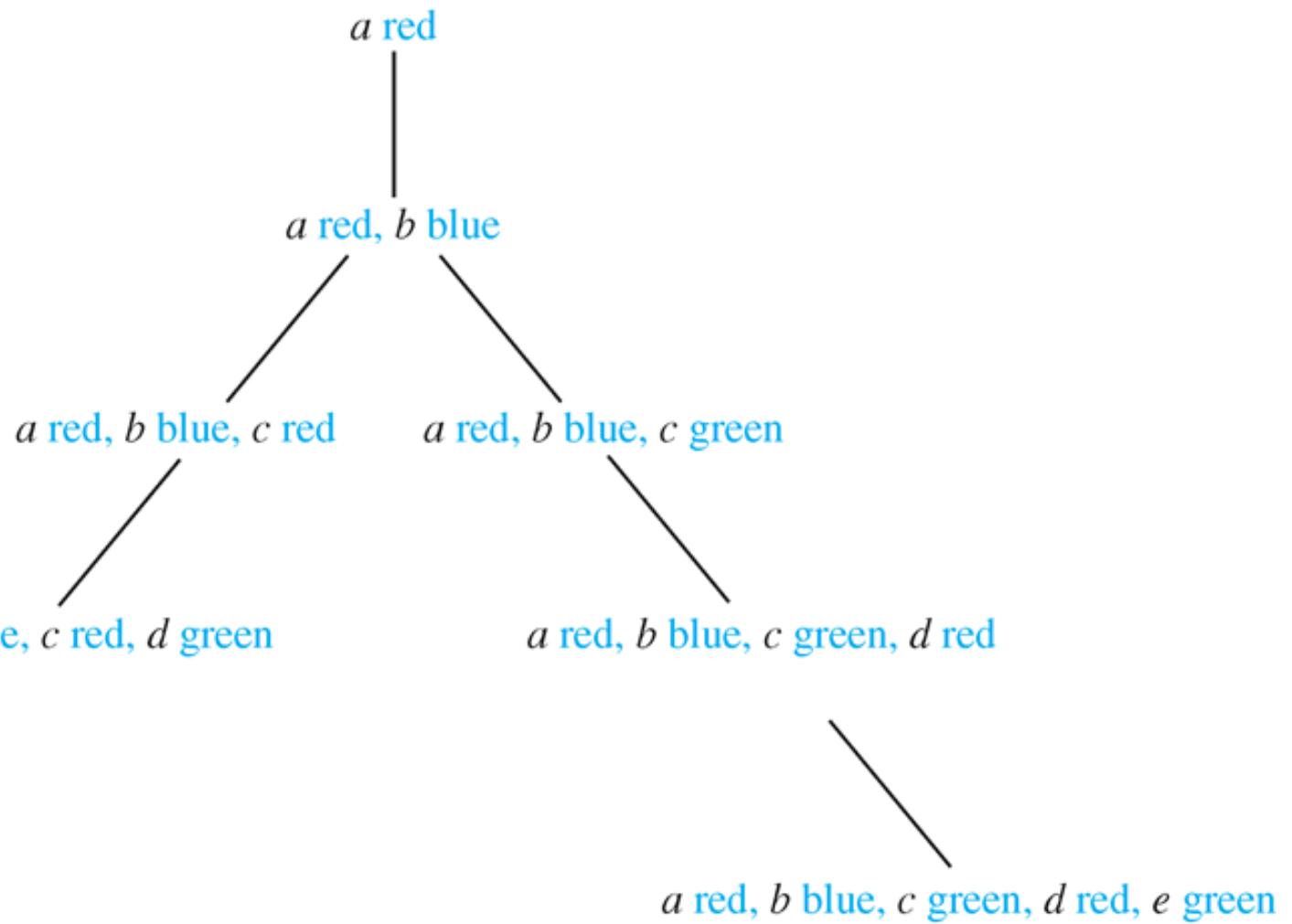
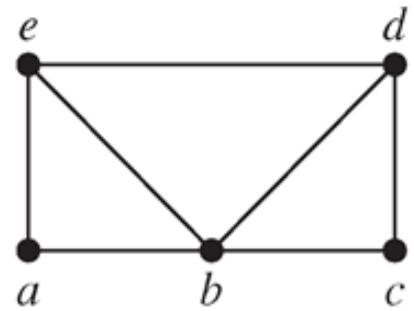
Applications of DFS, BFS

- find paths, circuits, connected components, cut vertices, ...
- find shortest paths, determine whether bipartite, ...
- graph coloring, sums of subsets, ...



Applications of DFS, BFS

■

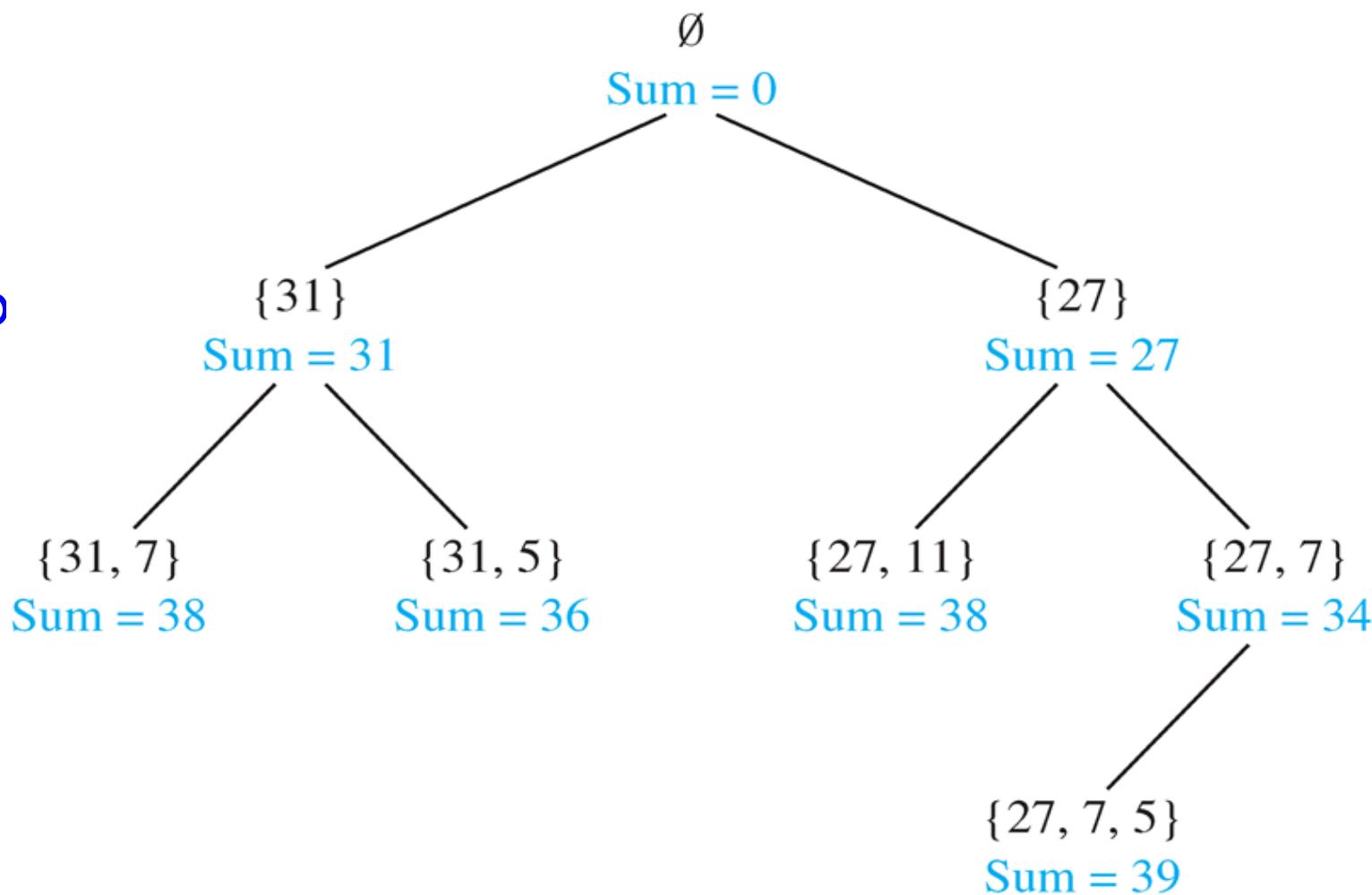


Applications of DFS, BFS

- find ~~other~~ ~~disjoint~~ connected components and subtrees, ...

find

graph



find a subset of $\{31, 27, 15, 11, 7, 5\}$ with the sum 39

Minimum Spanning Trees

- **Definition** A *minimum spanning tree* in a connected weighted graph is a spanning tree that has the **smallest possible sum of weights** of its edges.

Minimum Spanning Trees

- **Definition** A *minimum spanning tree* in a connected weighted graph is a spanning tree that has the **smallest possible sum of weights** of its edges.

two **greedy algorithms**: Prim's Algorithm, Kruscal's Algorithm

Prim's Algorithm

ALGORITHM 1 Prim's Algorithm.

```
procedure Prim( $G$ : weighted connected undirected graph with  $n$  vertices)  
     $T :=$  a minimum-weight edge  
    for  $i := 1$  to  $n - 2$   
         $e :=$  an edge of minimum weight incident to a vertex in  $T$  and not forming a  
            simple circuit in  $T$  if added to  $T$   
         $T := T$  with  $e$  added  
    return  $T$  { $T$  is a minimum spanning tree of  $G$ }
```



Prim's Algorithm

ALGORITHM 1 Prim's Algorithm.

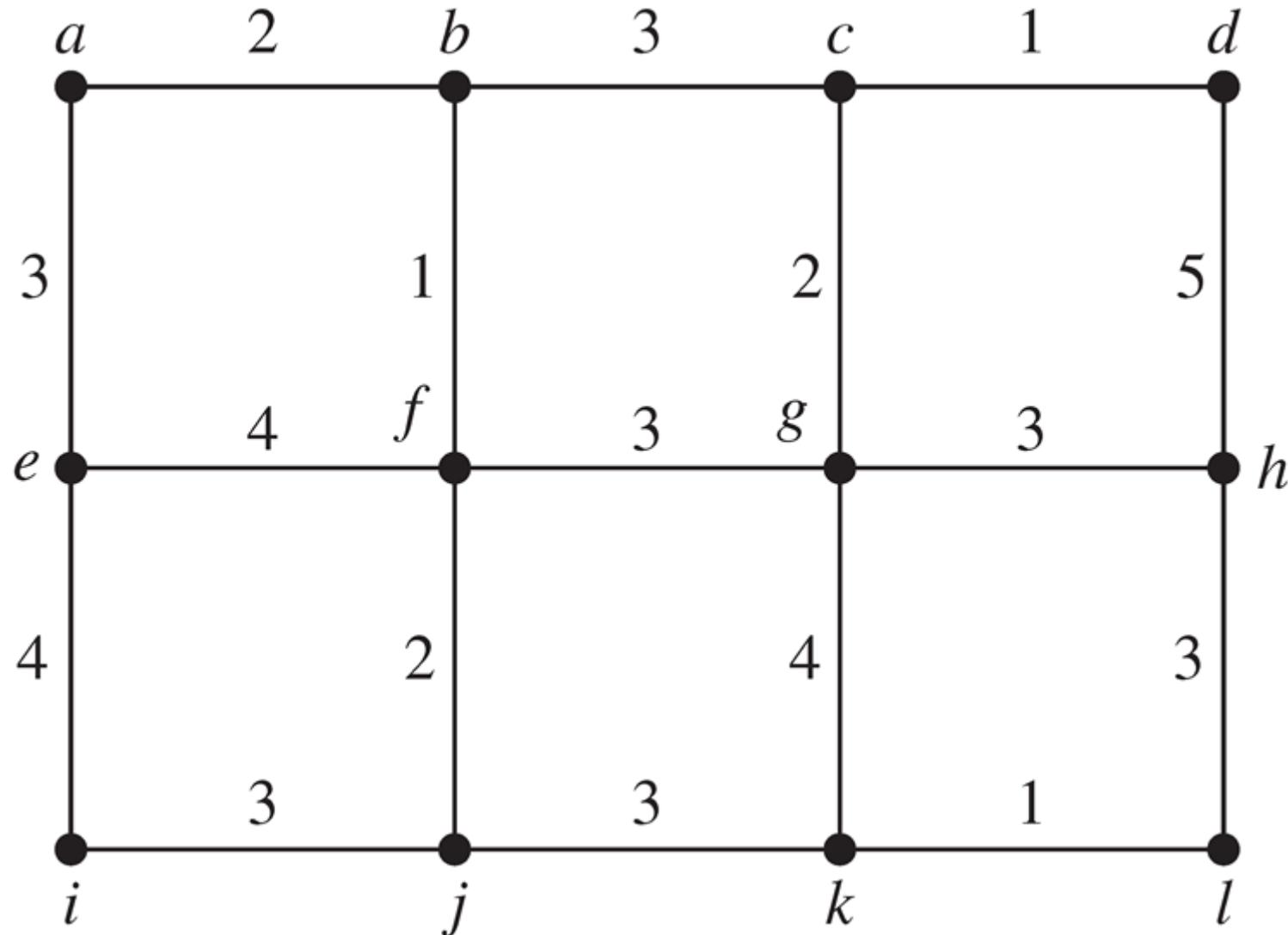
```
procedure Prim( $G$ : weighted connected undirected graph with  $n$  vertices)
   $T :=$  a minimum-weight edge
  for  $i := 1$  to  $n - 2$ 
     $e :=$  an edge of minimum weight incident to a vertex in  $T$  and not forming a
      simple circuit in  $T$  if added to  $T$ 
     $T := T$  with  $e$  added
  return  $T$  { $T$  is a minimum spanning tree of  $G$ }
```

time complexity: $e \log v$



Prim's Algorithm

Example



Kruscal's Algorithm

ALGORITHM 2 Kruskal's Algorithm.

```
procedure Kruskal( $G$ : weighted connected undirected graph with  $n$  vertices)
   $T :=$  empty graph
  for  $i := 1$  to  $n - 1$ 
     $e :=$  any edge in  $G$  with smallest weight that does not form a simple circuit
      when added to  $T$ 
     $T := T$  with  $e$  added
  return  $T$  { $T$  is a minimum spanning tree of  $G$ }
```

Kruscal's Algorithm

ALGORITHM 2 Kruskal's Algorithm.

```
procedure Kruskal( $G$ : weighted connected undirected graph with  $n$  vertices)
   $T :=$  empty graph
  for  $i := 1$  to  $n - 1$ 
     $e :=$  any edge in  $G$  with smallest weight that does not form a simple circuit
      when added to  $T$ 
     $T := T$  with  $e$  added
  return  $T$  { $T$  is a minimum spanning tree of  $G$ }
```

time complexity: $e \log e$

Kruscal's Algorithm

ALGORITHM 2 Kruskal's Algorithm.

```
procedure Kruskal( $G$ : weighted connected undirected graph with  $n$  vertices)
   $T :=$  empty graph
  for  $i := 1$  to  $n - 1$ 
     $e :=$  any edge in  $G$  with smallest weight that does not form a simple circuit
      when added to  $T$ 
     $T := T$  with  $e$  added
  return  $T$  { $T$  is a minimum spanning tree of  $G$ }
```

time complexity: $e \log e$

see *CLRS / Algorithm Design*, J. Kleinberg, E. Tardos



Kruscal's Algorithm

Example

