09 Graphs and Trees

CS201 Discrete Mathematics

Instructor: Shan Chen

Graphs

- Graphs are discrete structures consisting of vertices and edges that connect these vertices.
  - There are different kinds of graphs, depending on whether edges have directions, whether multiple edges can connect the same pair of vertices, and whether loops are allowed.
- Problems in almost every conceivable discipline can be solved using graph models.

Graphs and Graph Models

Example

- A computer network consists of computers (at different locations) and communication links between them.

- Some questions:
  - What is the minimum number of links between B and NO?

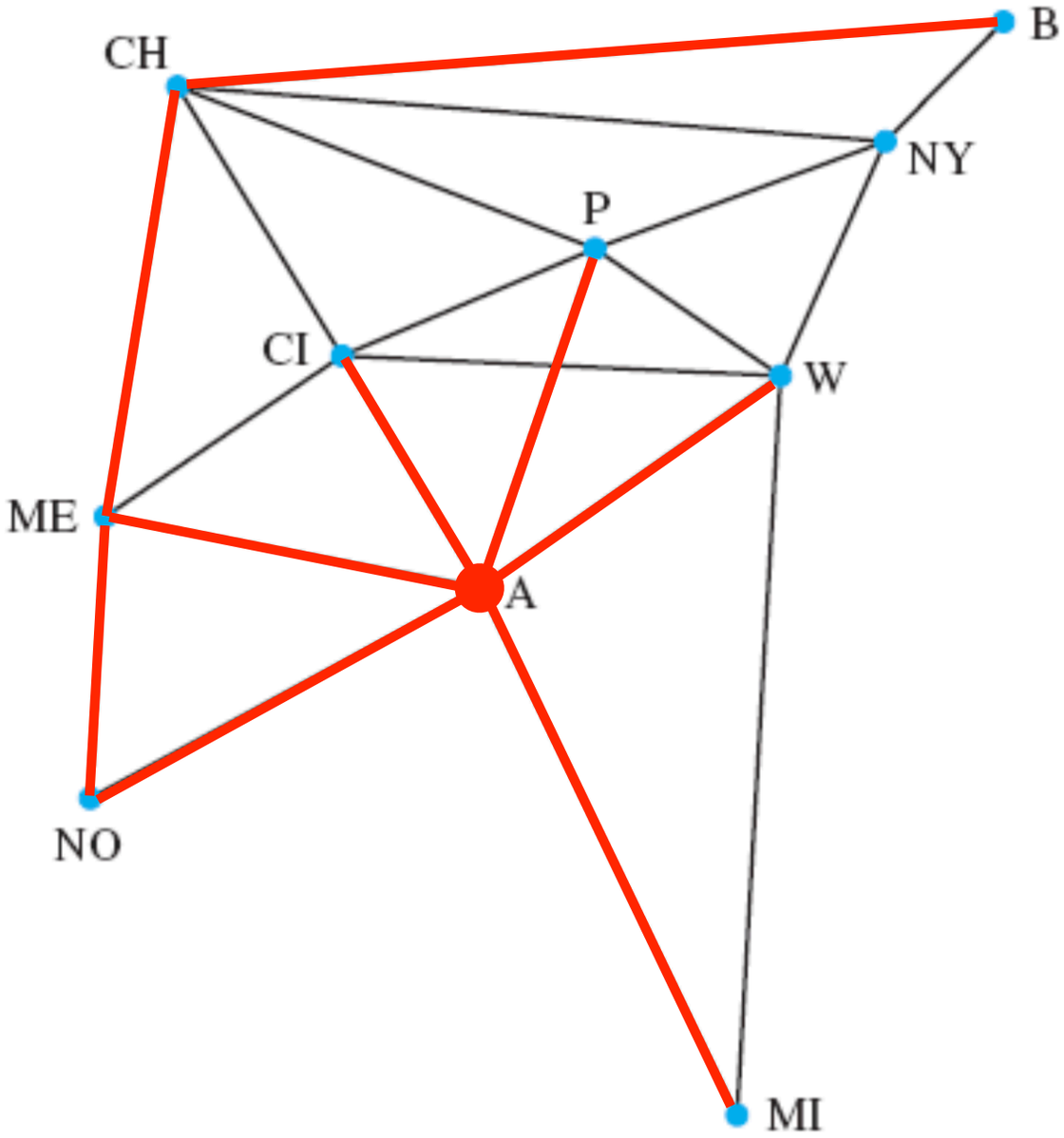
    3 (B-CH-ME-NO)

  - Which computer has the most links emanating from it?

    A (6 links)

  - What is the total number of communication links?

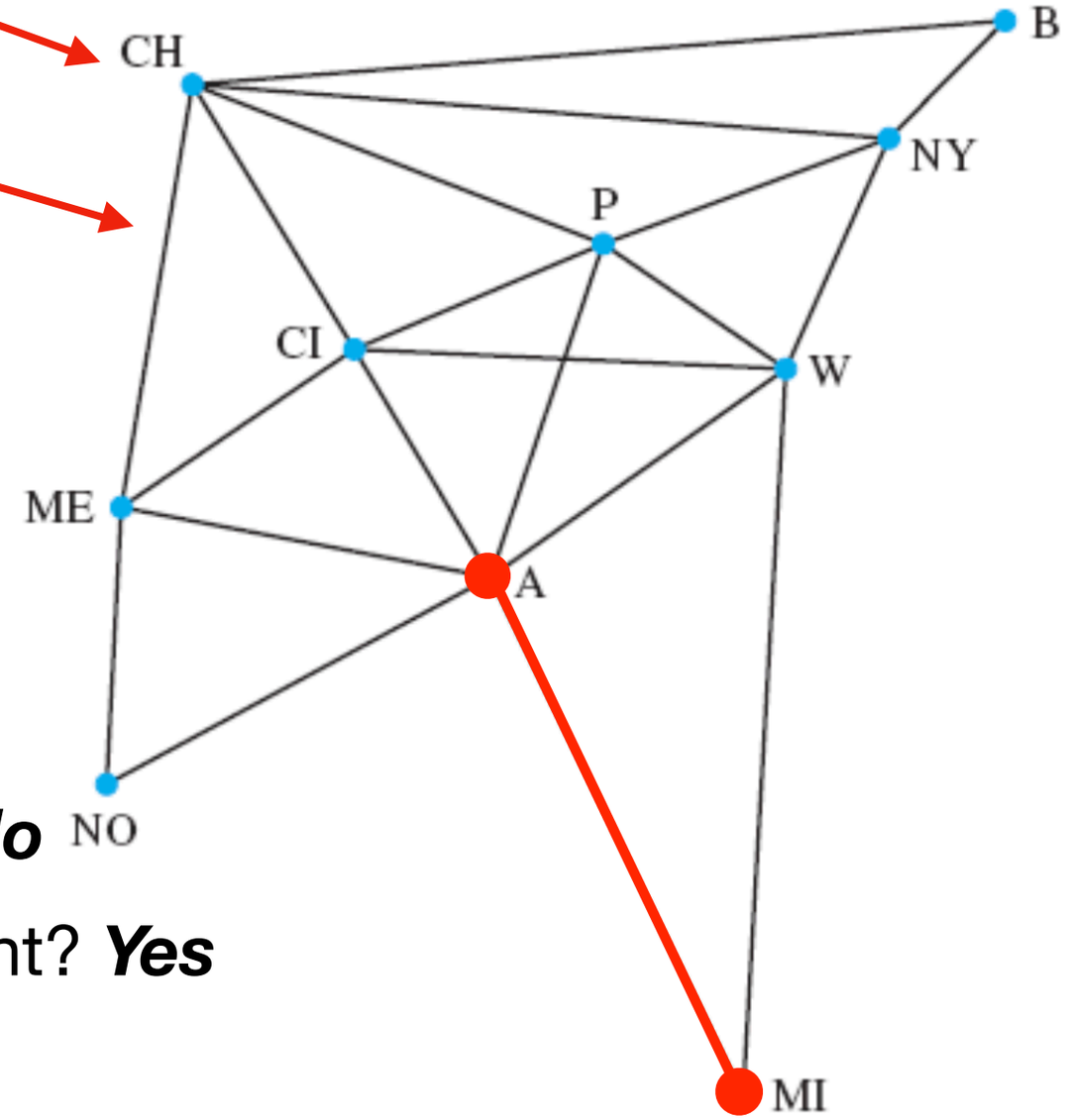
    20



SUSTech

Definition of a Graph

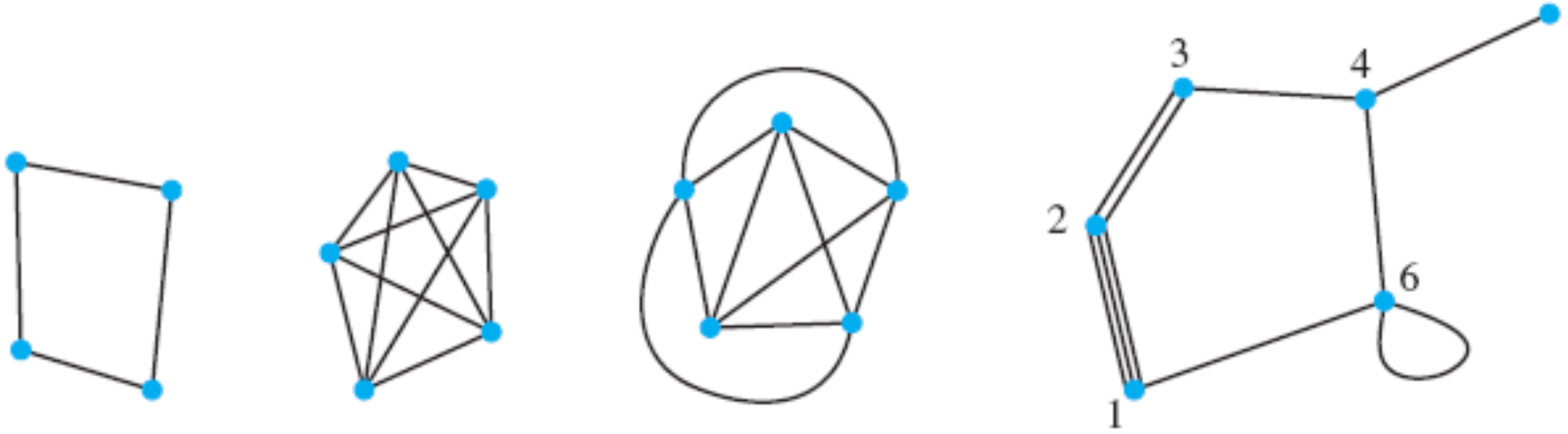
- **Definition:** A **graph** $G = (V, E)$ consists of
  - V, a nonempty set of **vertices** (or **nodes**), and
  - E, a set of **edges**.
- Each edge has either one or two vertices associated with it, called its **endpoints**.
  - An edge $\{u, v\}$ is said to **connect** its endpoints u and v.
- Questions:
  - Can an edge have no endpoints? ***No***
  - Can an edge have only one endpoint? ***Yes***



SUSTech

Definition of a Graph

- **Definition:** A **graph** $G = (V, E)$ consists of
  - V, a nonempty set of **vertices** (or **nodes**), and
  - E, a set of **edges**.
- More examples:



More Definitions

- **Simple graph:** a graph in which each edge connects two different vertices and no two edges connect the same pair of vertices.

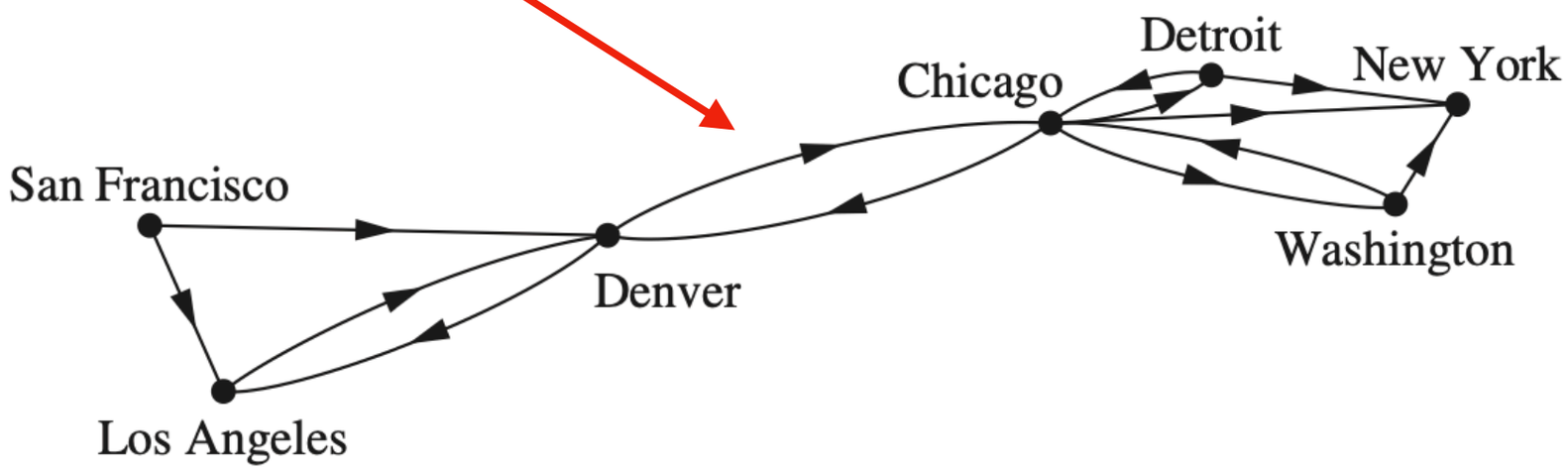
- **Multigraph:** a graph that may have multiple edges connecting the same pair of vertices.

- **Pseudograph:** a graph that may include loops (edges that connect a vertex to itself), and possibly multiple edges connecting the same pair of vertices or a vertex to itself.

** all of the above graphs are undirected graphs*

Directed Graphs

- **Definition:** A **directed graph** (or **digraph**) *G = (V, E)* consists of
  - *V*, a nonempty set of **vertices**, and
  - *E*, a set of **directed edges** (or **arcs**).



- Each edge is associated with an ordered pair of vertices.
  - The directed edge associated with the ordered pair *(u, v)* is said to **start at** *u* and **end at** *v*.
  - A **directed loop** associated with a vertex *u* is denoted by *(u, u)*.

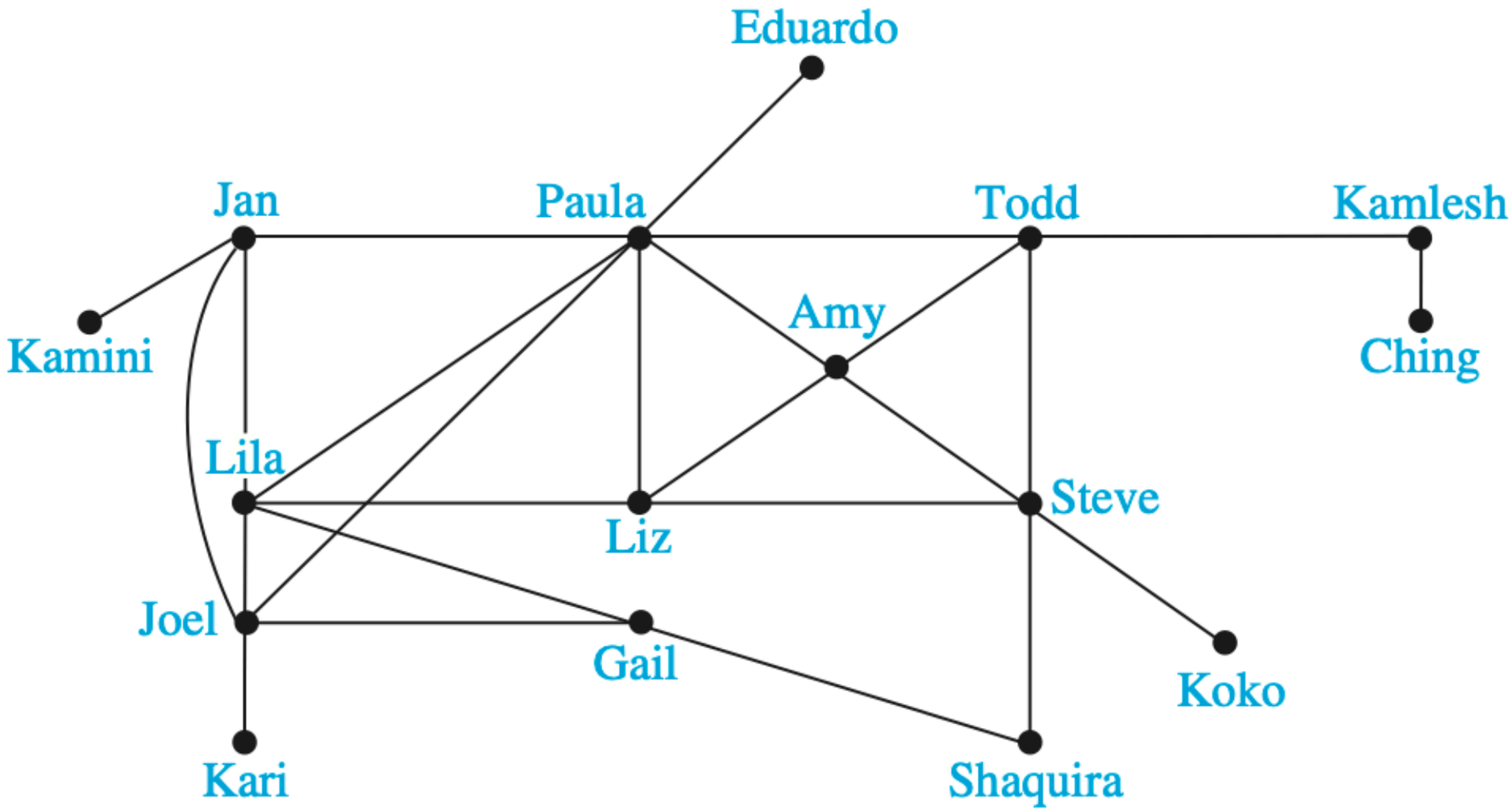
Graph Models

- **Graphs** can be used to model:
  - Social networks
  - Communication networks
  - Information networks
  - Software design applications
  - Transportation networks
  - Biological networks
  - Tournaments
  - …

** Can you find a subject where graphs have not been applied?*

Examples of Graph Models

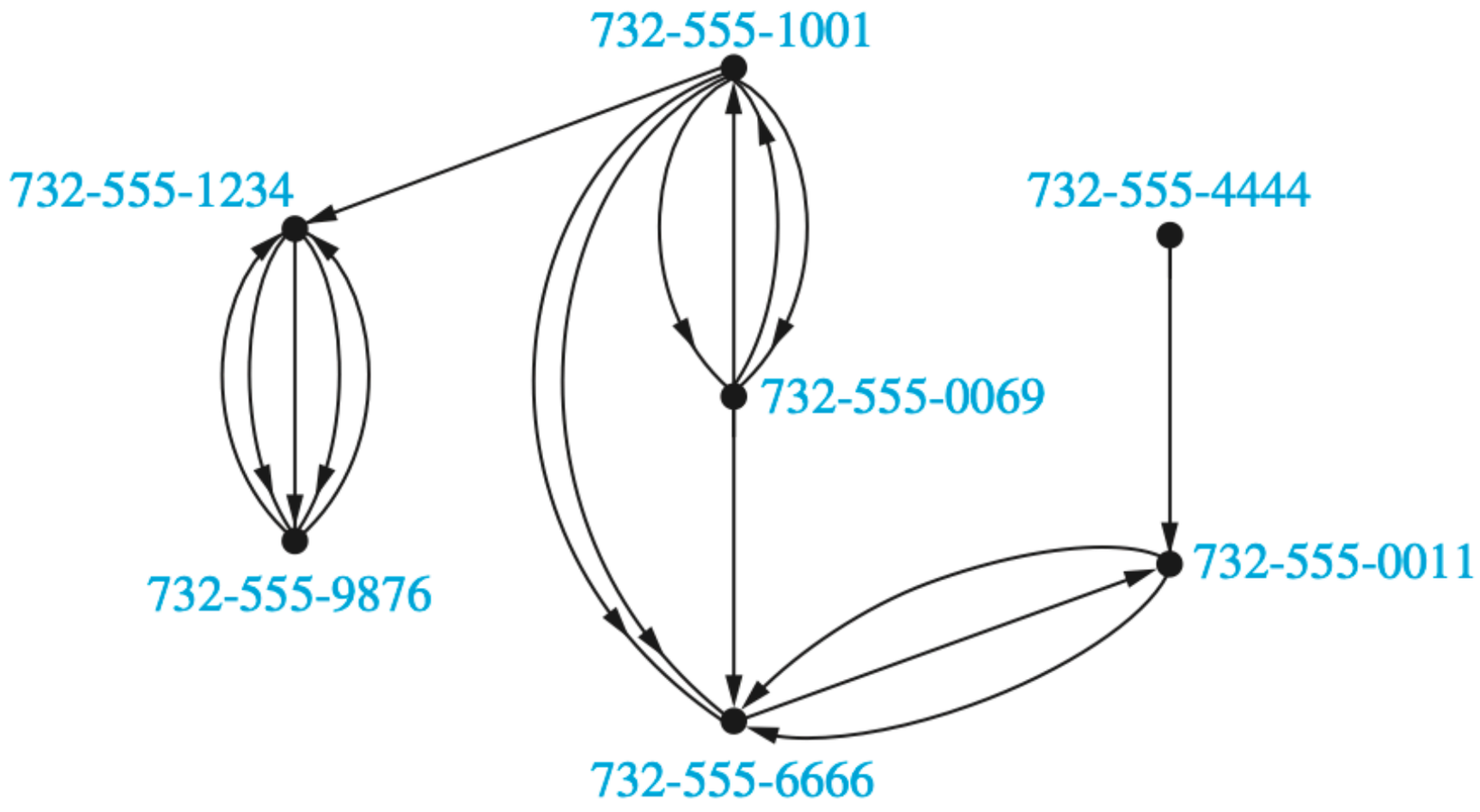
- A friendship graph:



SUSTech

Examples of Graph Models

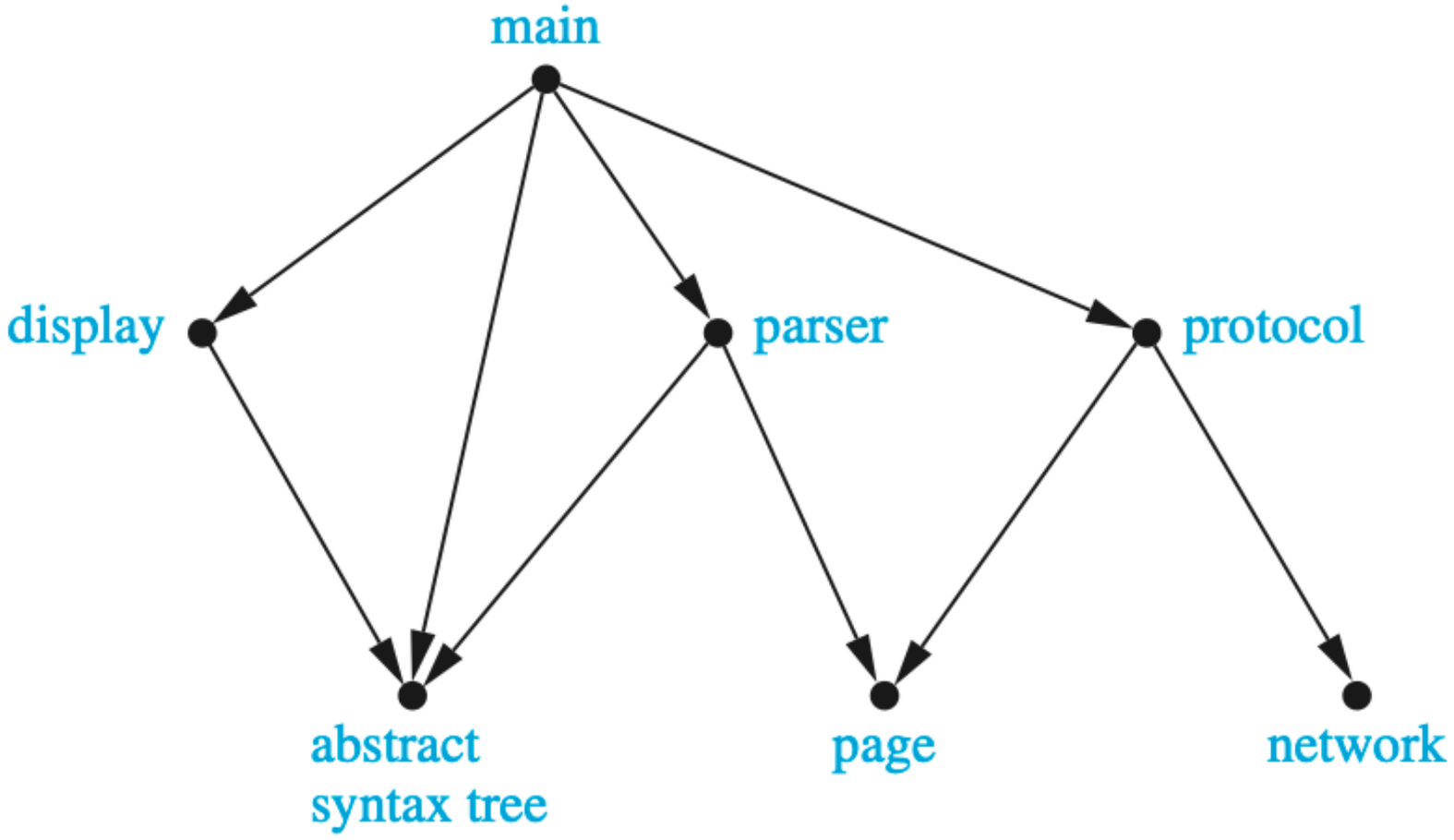
- A call graph:



SUSTech

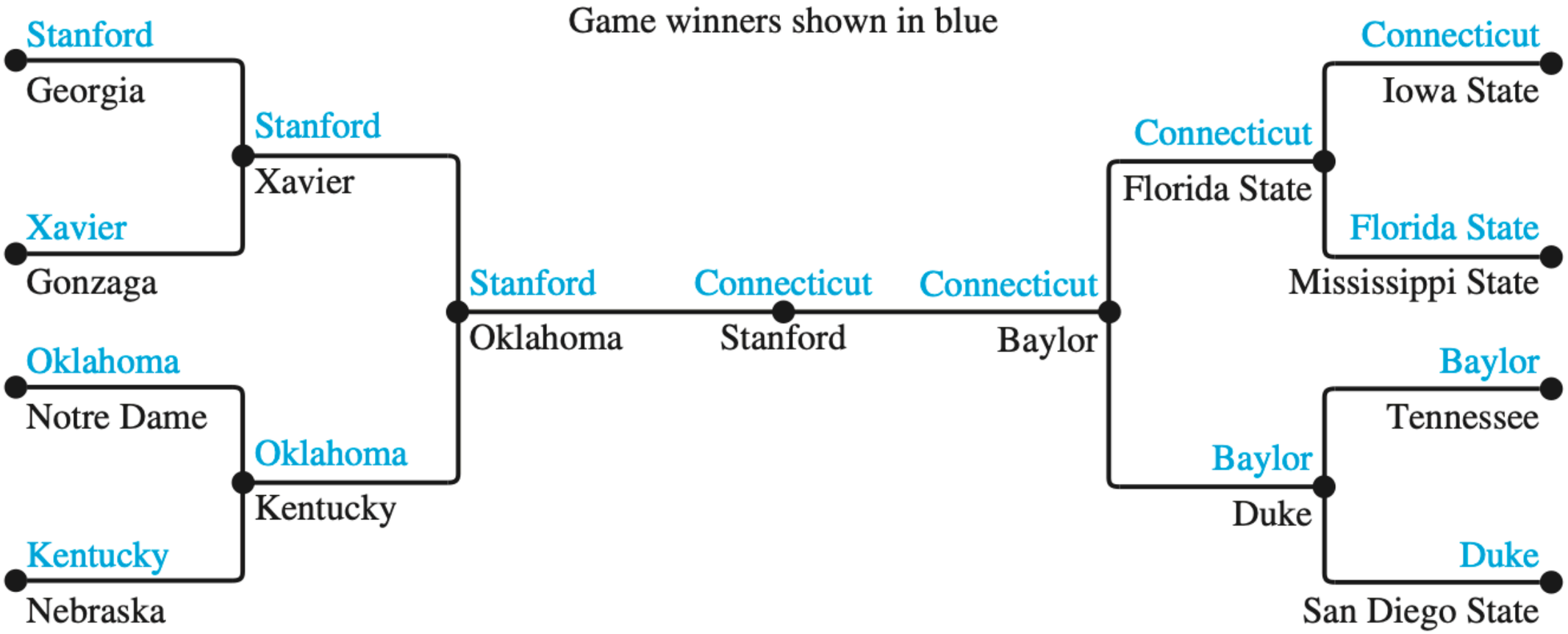
Examples of Graph Models

- A module dependency graph:



Examples of Graph Models

- A single-elimination tournament:



SUSTech

Graph Terminology and Operations

Undirected Graph Terminology

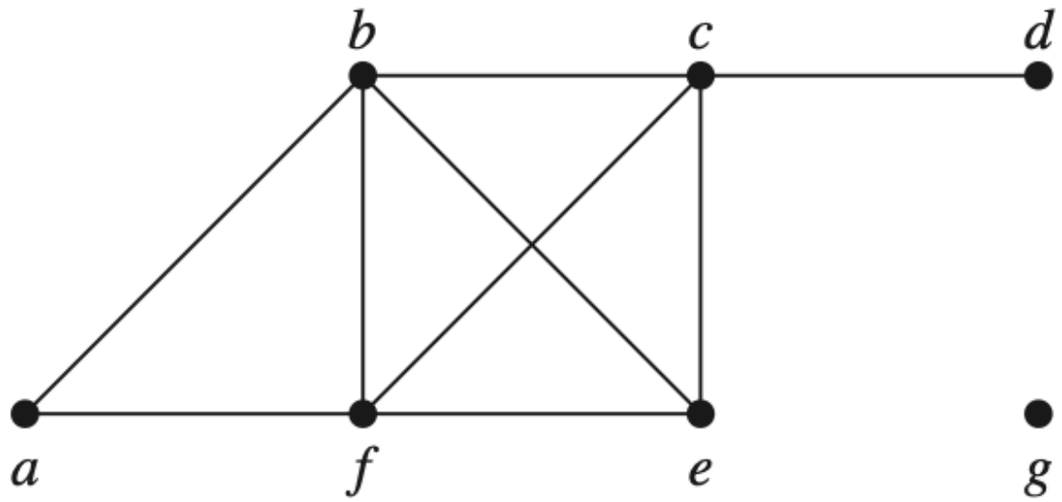
- **Definition:** Two vertices u and v in an undirected graph G are called **adjacent** (or **neighbors**) in G if u and v are endpoints of an edge e of G. Such an edge e is called **incident with/to** the vertices u and v and e is said to **connect** u and v.

- **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. So, $N(A) = \bigcup_{v \in A} N(v)$.

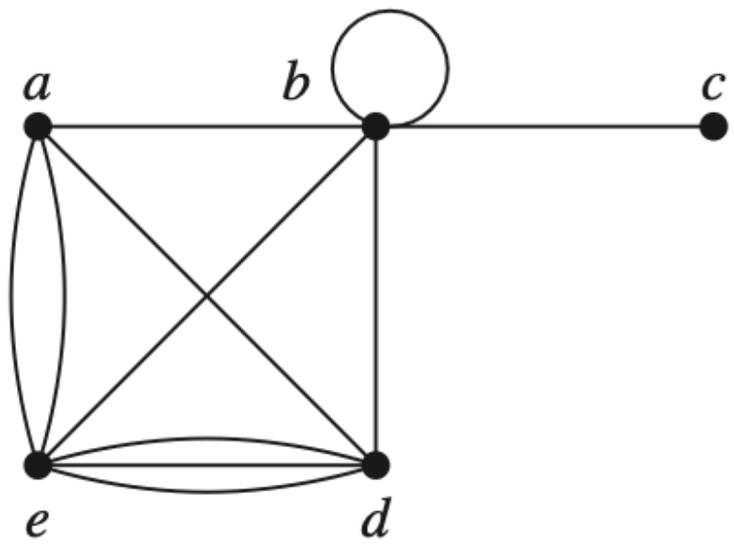
- **Definition:** The **degree** of a vertex in an undirected graph is the number of edges incident with it, except that a loop at a vertex contributes twice to the degree of that vertex. The degree of the vertex v is denoted by $\deg(v)$.
  - A vertex of degree 0 is called **isolated**.
  - A vertex of degree 1 is called **pendant**.

Exercise *(3 mins)*

- What are the degrees and neighborhoods of the vertices/sets in the following graphs?
  - deg(*b*) = ?
  - deg(*g*) = ?
  - *N*(*c*) = ?
  - *N*(*g*) = ?
  - *N*({*b*, *c*}) = ?



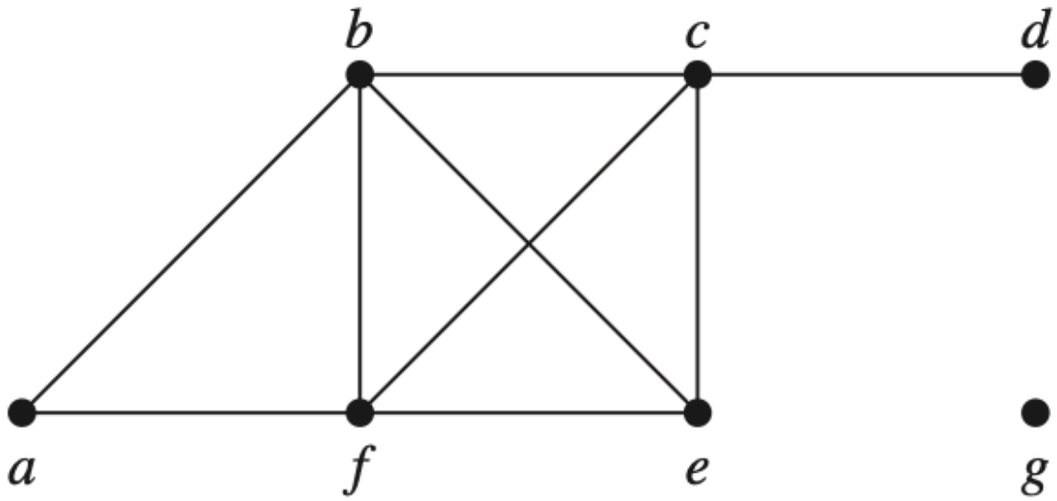
  - deg(*b*) = ?
  - deg(*d*) = ?
  - *N*(*b*) = ?



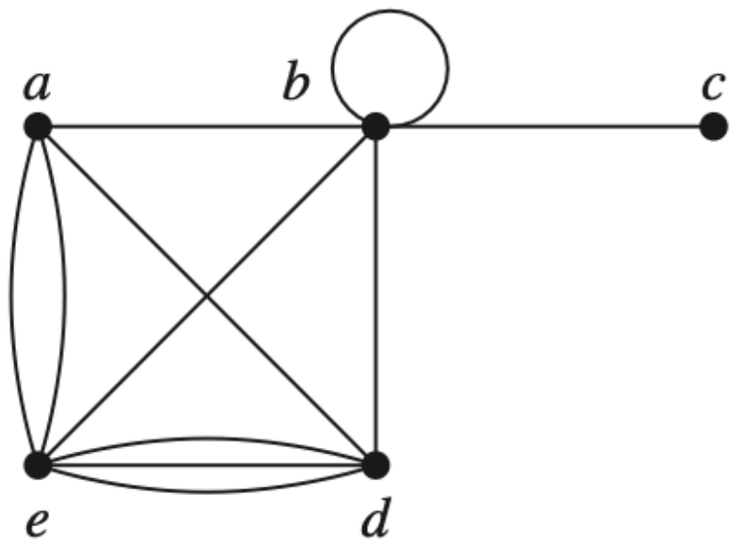
SUSTech

Exercise *(3 mins)*

- What are the degrees and neighborhoods of the vertices/sets in the following graphs?
  - deg(*b*) = *4*
  - deg(*g*) = *0*
  - *N*(*c*) = {*b*, *d*, *e*, *f*}
  - *N*(*g*) = ∅
  - *N*({*b*, *c*}) = {*a*, *b*, *c*, *d*, *e*, *f*}



  - deg(*b*) = *6*
  - deg(*d*) = *5*
  - *N*(*b*) = {*a*, *b*, *c*, *d*, *e*}



The Handshaking Theorem

- **The Handshaking Theorem:** If *G = (V, E)* is an undirected graph with *e* edges, then

$$2e = \Sigma_{v \in V} \deg(v)$$

  - This theorem applies even if multiple edges and loops are present.

- Proof: Each edge contributes two to the sum of the degrees of the vertices because an edge is incident with exactly two (possibly equal) vertices. This means that the sum of the degrees of the vertices is twice the number of edges.

- **Corollary:** An undirected graph has an even number of vertices of odd degree.

- Proof: Partition *V* into vertices of even degree and of odd degree. The above theorem shows the degree sum is an even number.

Directed Graph Terminology

- **Definition:** When (u, v) is an edge of the graph G with directed edges, u is said to be **adjacent to** v and v is said to be **adjacent from** u. The vertex u is called the **initial vertex** of (u, v), and v is called the **terminal/end vertex** of (u, v).
  - The initial vertex and terminal vertex of a loop are the same.
- **Definition:** In a graph with directed edges the **in-degree** of a vertex v, denoted by $\deg^-(v)$, is the number of edges with v as their terminal vertex. 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 one to both the in-degree and out-degree of this vertex.
- The undirected graph resulting from ignoring directions of edges of a directed graph is called the **underlying undirected graph**.

Theorem

- **Theorem:** If *G = (V, E)* is a directed graph with *e* edges, then

$$e = \Sigma_{v \in V} \deg^-(v) = \Sigma_{v \in V} \deg^+(v)$$

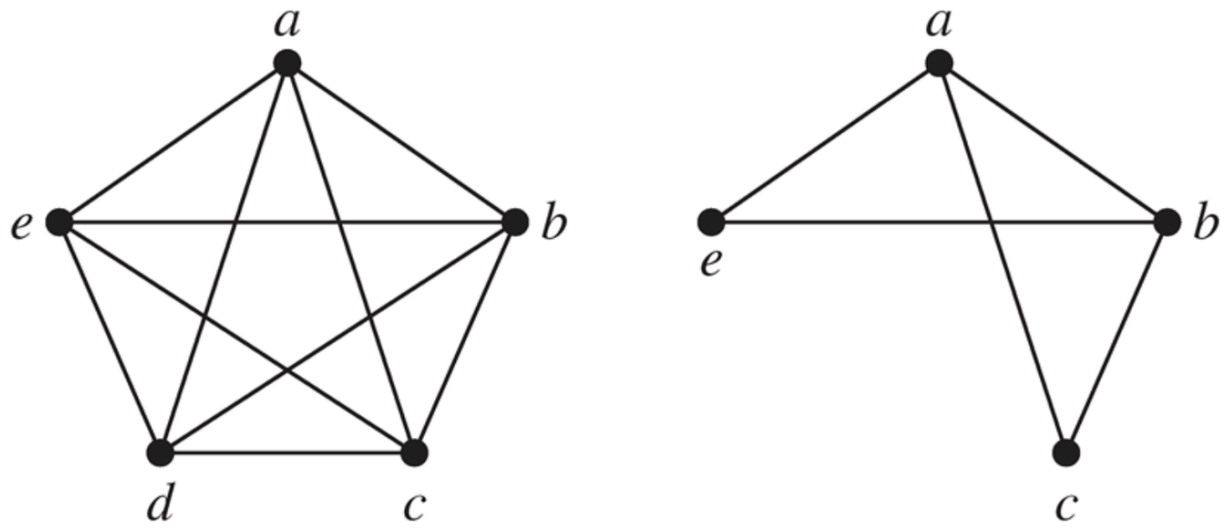
  - This theorem applies even if multiple edges and loops are present.

- Proof: Because each directed edge has an initial vertex and a terminal vertex, the sum of the in-degrees and the sum of the out-degrees of all vertices in a graph with directed edges are the same. Both of these sums are the number of edges in the graph.

Subgraphs

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

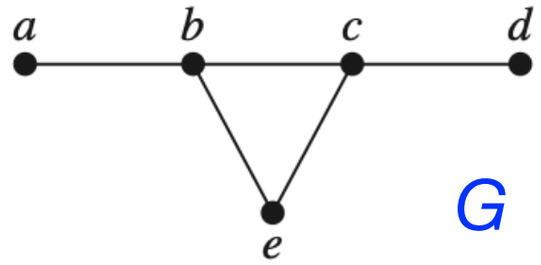
- Example:



- **Definition:** Let $G = (V, E)$ be a simple graph. The **subgraph induced by a subset** W of the vertex set V is the graph (W, F), where the edge set F contains an edge in E if and only if both endpoints of this edge are in W.

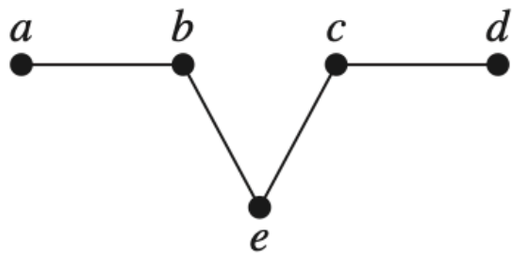
Graph Operations

- Consider an arbitrary graph $G = (V, E)$.

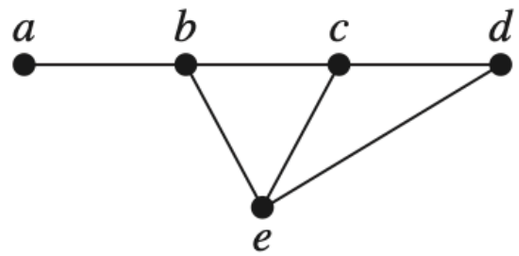


G

- **Removing/Adding edges:**
  - $G - e = (V, E \setminus \{e\})$
  - $G + e = (V, E \cup \{e\})$



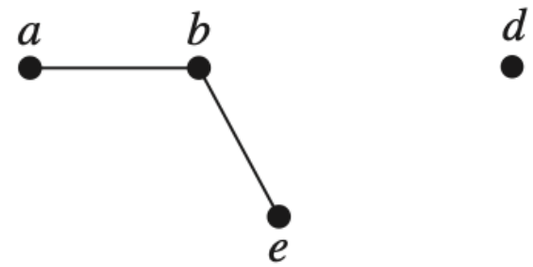
$G - \{b, c\}$



$G + \{e, d\}$

- **Removing vertices:**
  - $G - v = (V \setminus \{v\}, E')$

    where E' is the set of edges in E not incident to v



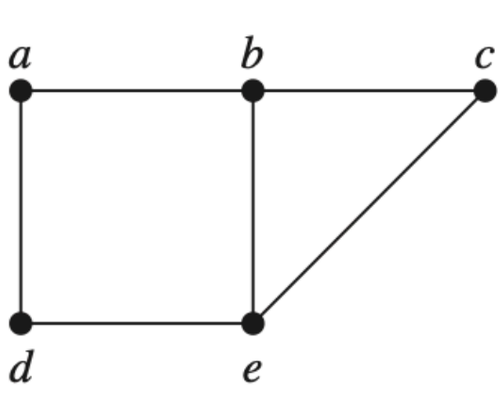
$G - c$

SUSTech

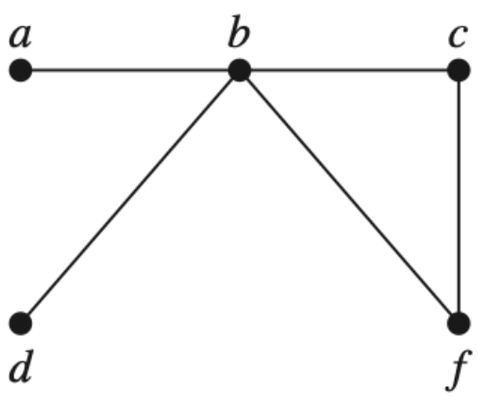
Graph Operations

- **Graph Unions:** 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 $E_1 \cup E_2$. The union of G_1 and G_2 is denoted by $G_1 \cup G_2$.

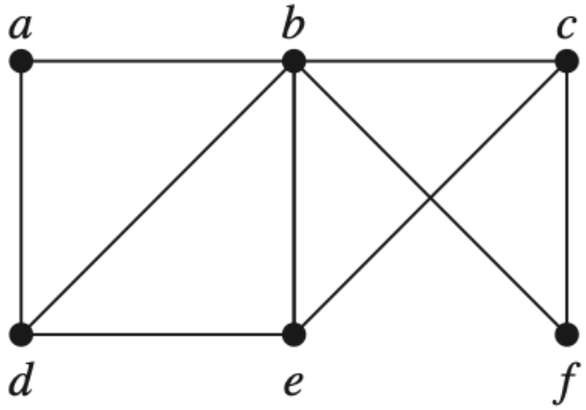
- Example:



G_1



G_2

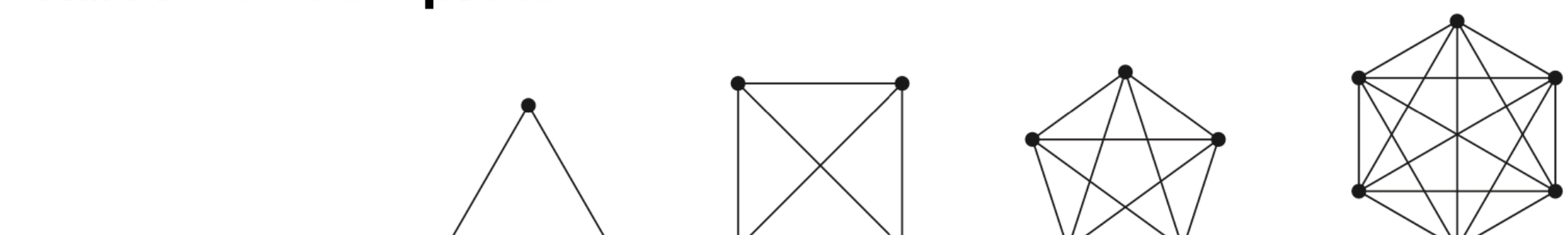


$G_1 \cup G_2$

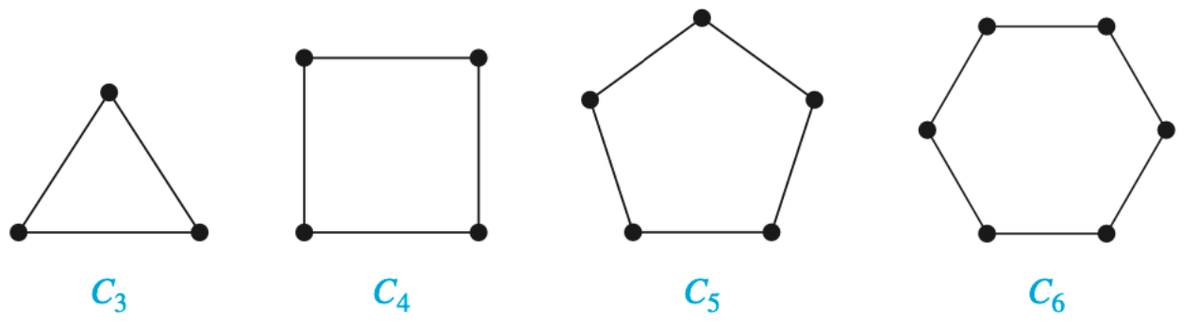
Special Types of Graphs

Complete Graphs and Cycles

- **Definition:** A **complete graph** on n vertices, denoted by K_n, is a simple graph that contains exactly one edge between each pair of distinct vertices. A simple graph that is not a complete graph is called **noncomplete**.

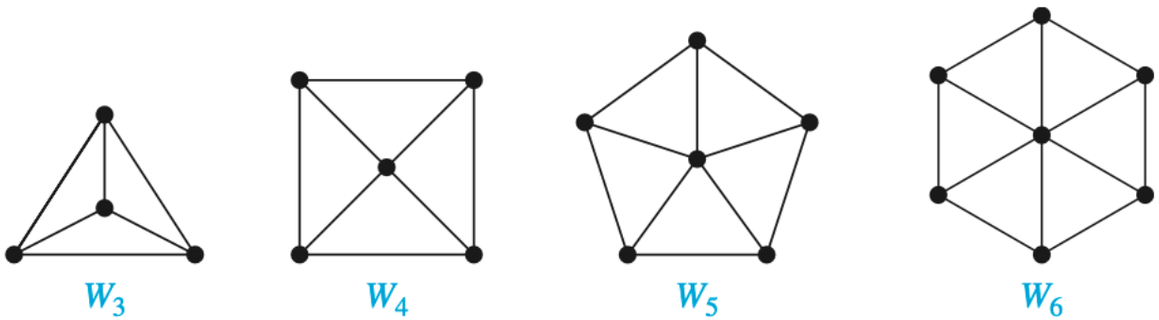


- **Definition:** A **cycle** C_n, $n \geq 3$, consists of n vertices $v_1, v_2, \ldots, v_n$, and edges $\{v_1, v_2\}, \{v_2, v_3\}, \ldots, \{v_{n-1}, v_n\}$ and $\{v_n, v_1\}$.

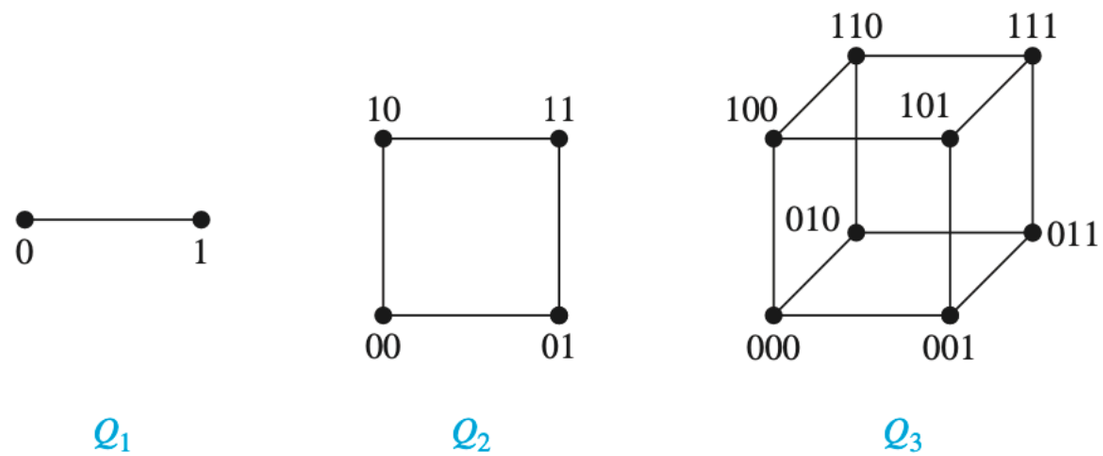


Wheels and *n*-Cubes

- **Definition:** A **wheel** W_n is obtained by adding an additional vertex to a cycle C_n, for $n \geq 3$, and connect this new vertex to each of the n vertices in C_n, by new edges.



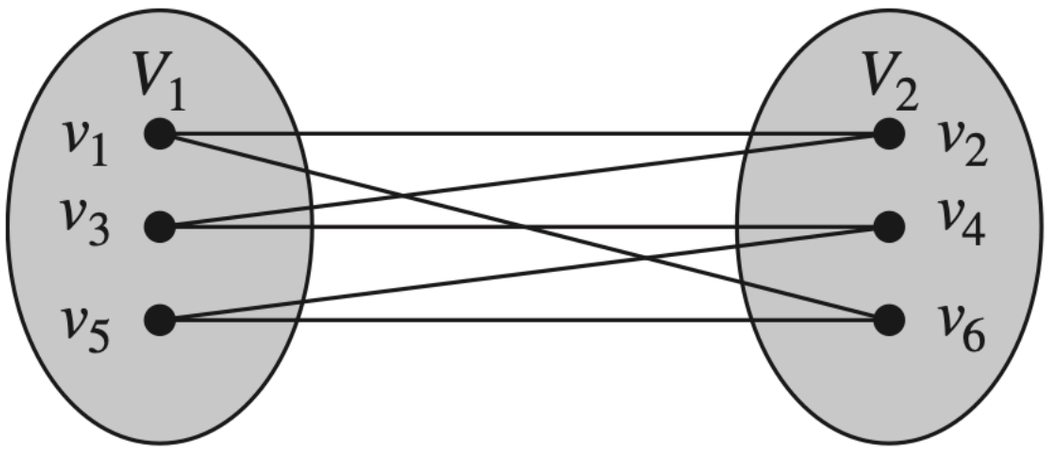
- **Definition:** An ***n*-dimensional hypercube**, or ***n*-cube**, denoted by Q_n, is a graph that has vertices representing the 2^n bit strings of length n. Two vertices are adjacent if and only if the bit strings that they represent differ in exactly one bit position.

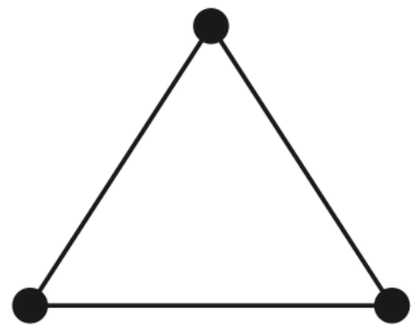


SUSTech

Bipartite Graphs

- **Definition:** A simple graph G is called **bipartite** if its vertex set V can be partitioned into two disjoint sets V_1 and V_2 so that every edge in the graph connects a vertex in V_1 and a vertex in V_2.
  - No edge connects two vertices in V_1 or two vertices in V_2.
  - The pair (V_1, V_2) is called a **bipartition** of the vertex set V *of* G.
- Example 1: C_6 is bipartite and K_3 (or C_3) is not bipartite





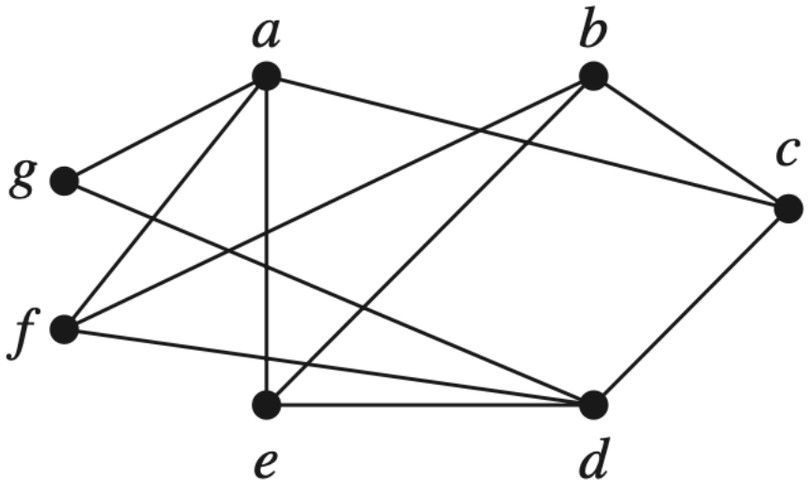
SUSTech

Bipartite Graphs

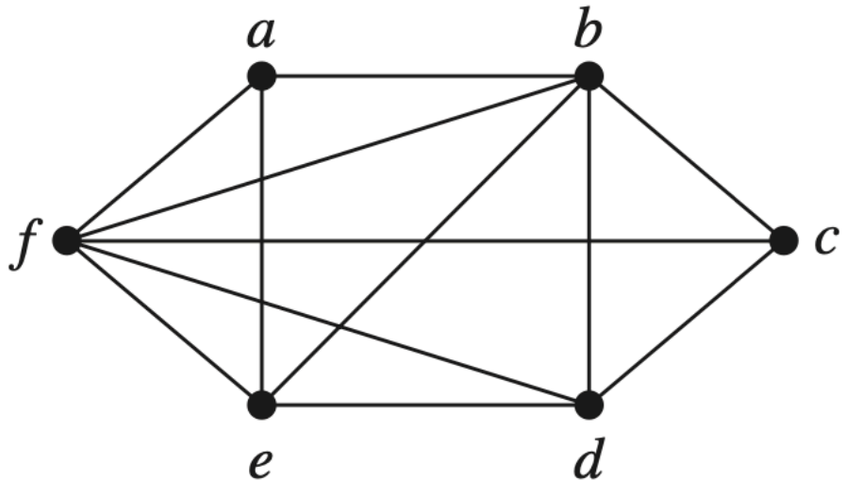
- **Definition:** A simple graph G is called **bipartite** if its vertex set V can be partitioned into two disjoint sets V_1 and V_2 so that every edge in the graph connects a vertex in V_1 and a vertex in V_2.
  - No edge connects two vertices in V_1 or two vertices in V_2.
  - The pair (V_1, V_2) is called a **bipartition** of the vertex set V *of* G.

- Example 2: Are the following graphs bipartite?

Yes



No



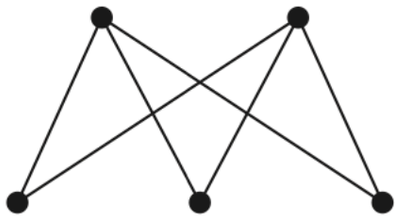
Bipartite Graphs

- **Theorem:** A simple graph G is bipartite if and only if it is possible to assign red or blue to each vertex of the graph so that no two adjacent vertices are assigned the same color.

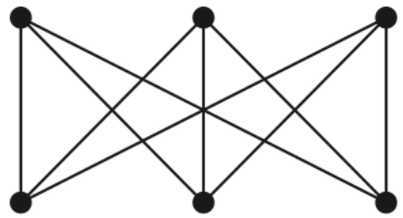
- Proof:
  - **"only if" part**: Assign red to V_1 vertices and blue to V_2 vertices.
  - **"if" part:** Let V_1 consists of all red vertices and V_2 consists of all blue vertices, then (V_1, V_2) is a bipartition of the vertex set V of G.

Complete Bipartite Graphs

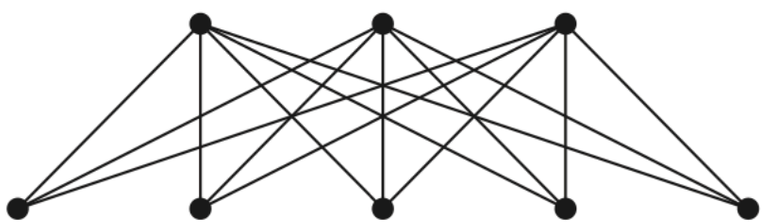
- **Definition:** A **complete bipartite graph** $K_{m,n}$ is a graph that has its vertex set partitioned into two subsets of m and n vertices, with an edge between two vertices if and only if one vertex is in the first subset and the other vertex is in the second subset.



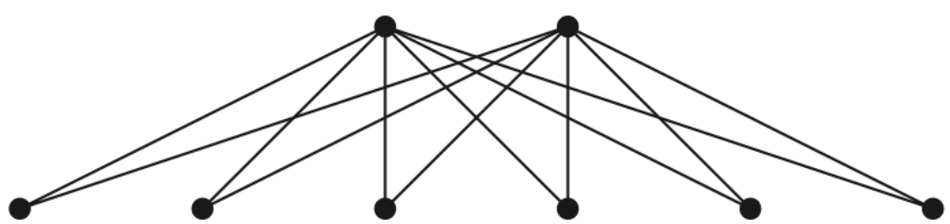
$K_{2,3}$



$K_{3,3}$



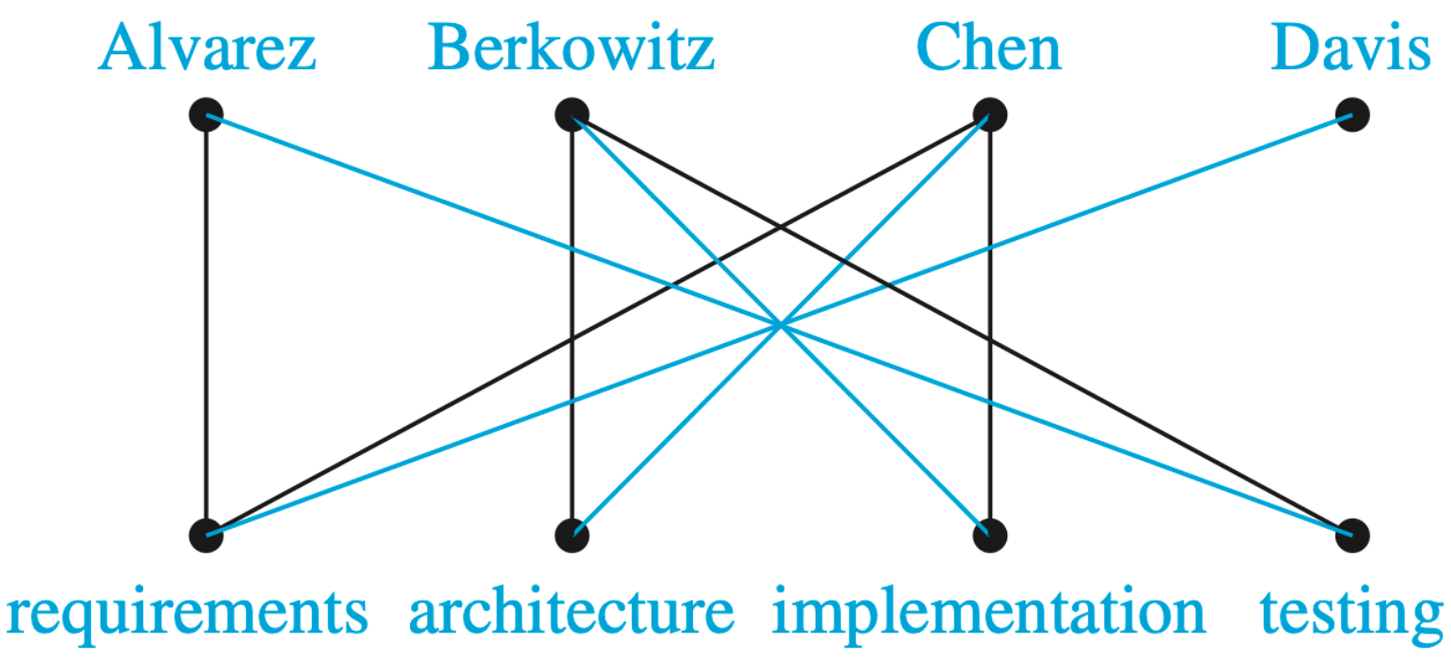
$K_{3,5}$



$K_{2,6}$

Bipartite Graphs and Matchings

- **Job assignments:** vertices represent the jobs and 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 **matching** M in a simple graph $G = (V, E)$ is a subset of the edge set E such that no two edges are incident with the same vertex, i.e., all edges in M are "disjoint".
  - A vertex that is the endpoint of an edge of a matching M is said to be **matched** in M; otherwise it is said to be **unmatched**.
- **Definition:** A **maximum matching** is a matching with the largest number of edges.
- **Definition:** 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 $|M| = |V_1|$.
  - A complete matching M from V_1 to V_2 can be viewed as the graph representation of an injective function from V_1 to V_2.

Hall's Marriage 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" part:**

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

    Then, if $A \subseteq V_1$, for every vertex $v \in A$, there is an edge in M connecting v to a different vertex in V_2. Consequently, 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. It follows that $|N(A)| \geq |A|$.

  - **"if" part:** proof by strong induction

Hall's Marriage 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 the **"if" part** by strong induction:
  - **Basis step:** obviously true for $|V_1| = 1$ since $|N(V_1)| \geq |V_1| = 1$
  - **Inductive step:** Let k be a positive integer and let $G = (V, E)$ is a bipartite graph with bipartition (V_1, V_2) such that $|V_1| \leq k$.

    Inductive hypothesis: If $|V_1| = k$ and $|N(A)| \geq |A|$ holds for all subset $A \subseteq V_1$, then there is a complete matching M from V_1 to V_2.

    Now, we prove the inductive step by cases. That is, we prove that if $|V_1| = k + 1$ and $|N(A)| \geq |A|$ holds for all subset $A \subseteq V_1$, then there is a complete matching M from V_1 to V_2.

Hall's Marriage 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 the **"if" part** by strong induction:
  - **Inductive step:** Suppose $|V_1| = k + 1$. We prove the inductive step by considering two cases:

Case 1: For every nonempty subset A of at most k vertices from V_1, the vertices in A are adjacent to at least $|A| + 1$ vertices of V_2, i.e., for every A such that $A \subseteq V_1$ and $1 \leq |A| \leq k$, $|N(A)| \geq |A| + 1$.

Case 2: For some integer j with $1 \leq j \leq k$, there exists a subset V_1' of j vertices from V_1 such that $|N(V_1')| = |V_1'| = j \leq k$.

** note that "$|N(A)| \geq |A|$ for all subsets A of V_1" = Case 1 + Case 2*

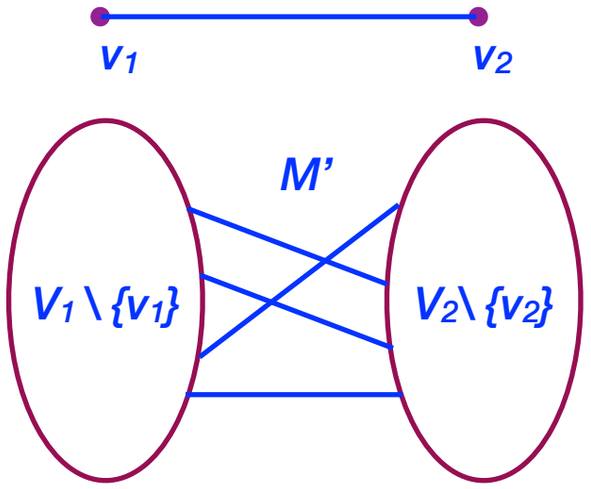
Hall's Marriage Theorem

- **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 the **"if" part** by strong induction:
  - **Case 1:** For all A such that $A \subseteq V_1$ and $1 \leq |A| \leq k$, $|N(A)| \geq |A| + 1$.

Pick any vertex $v_1 \in V_1$, we have $N(v_1) \geq |\{v_1\}| + 1 = 2$.
Remove this v_1 and a vertex $v_2 \in N(v_1)$ from G. This produces a new bipartite graph G' with bipartition $(V_1 - \{v_1\}, V_2 - \{v_2\})$.
By inductive hypothesis on G', since for all subset $A \subseteq V_1 - \{v_1\}$, $|N(A)| \geq |A| + 1 - 1 = |A|$, there exists a complete matching M' from $V_1 - \{v_1\}$ to $V_2 - \{v_2\}$.

Adding the edge $\{v_1, v_2\}$ to M' produces a complete matching from V_1 to V_2.



Hall's Marriage Theorem

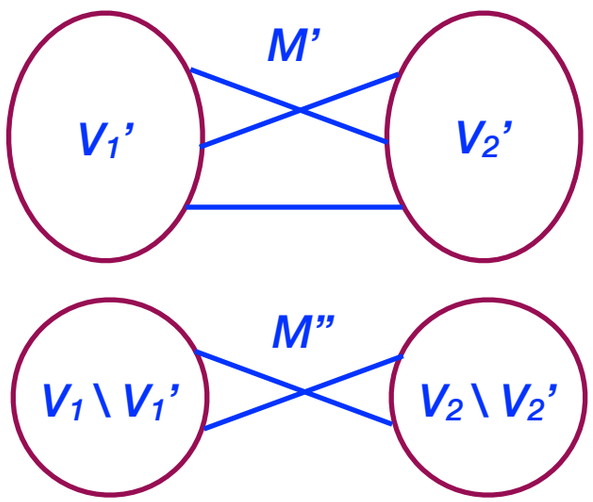
- **Hall's Marriage Theorem:** The bipartite graph $G = (V, E)$ with bipartition (V_1, V_2) has a complete matching from V_1 to V_2 if and only if $|N(A)| \geq |A|$ for all subsets A of V_1.

- Proof of the **"if" part** by strong induction:
  - **Case 2:** For some integer j with $1 \leq j \leq k$, there exists a subset $V_1' \subseteq V_1$ such that $|N(V_1')| = |V_1'| = j \leq k$. (Let $V_2' = N(V_1')$.)

    By inductive hypothesis on the subgraph G' induced by $V_1' \cup V_2'$, there exists a complete matching M' from V_1' to V_2'.

    Remove these $2j$ vertices (of $V_1' \cup V_2'$) from G. This produces a bipartite graph H with bipartition $(V_1 - V_1', V_2 - V_2')$.

    If we can find a complete matching M'' in H, then combining M' and M'' yields a complete matching from V_1 to V_2.



Hall's Marriage 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 the **"if" part** by strong induction:

  - **Case 2:** For some integer j with $1 \leq j \leq k$, there exists a subset $V_1' \subseteq V_1$ such that $|N(V_1')| = |V_1'| = j \leq k$. (Let $V_2' = N(V_1')$.)

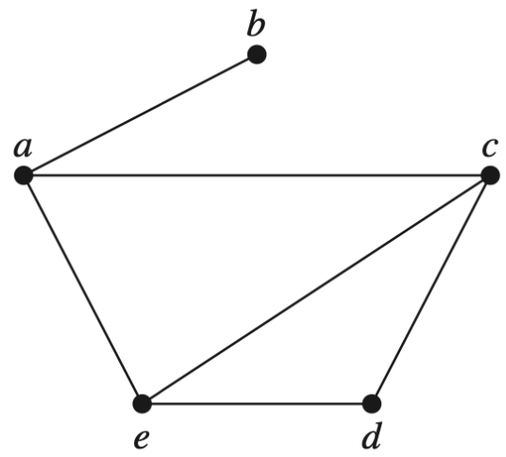
    By inductive hypothesis, we are only left to show that graph H satisfies $|N(A)| \geq |A|$ for all subsets $A \subseteq V_1 - V_1'$.

    Proof by **contradiction**: If not, there exists a subset $V_1'' \subseteq V_1 - V_1'$ such that $|V_1''| = t \geq 1$ and the vertices in V_1'' have $< t$ vertices in $V_2 - V_2'$ as neighbors. Therefore, $|N(V_1' \cup V_1'')| < t + j = |V_1' \cup V_1''|$, which contradicts with the "if" premise: $|N(A)| \geq |A|$ holds for all subsets $A \subseteq V_1$.

Graph Representation and Isomorphism

Adjacency Lists

- **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.



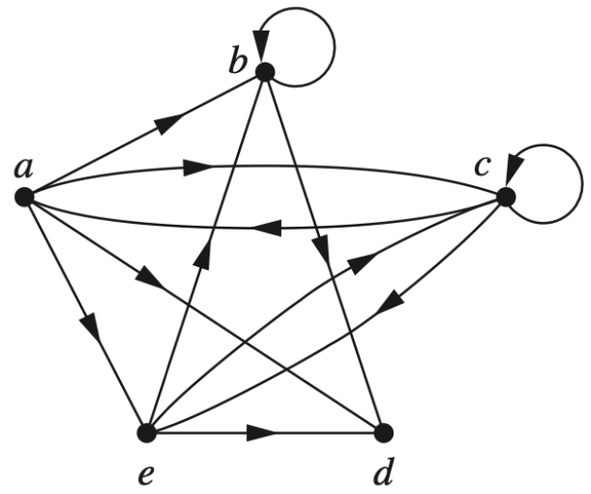


TABLE 1 An Adjacency List for a Simple Graph.

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

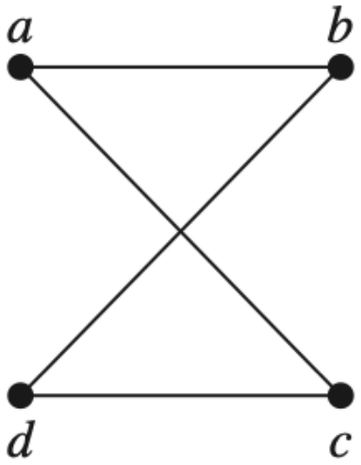
TABLE 2 An Adjacency List for a Directed Graph.

Initial Vertex	*Terminal Vertices*
a	b, c, d, e
b	b, d
c	a, c, e
d	
e	b, c, d

Adjacency Matrices

- **Definition:** Suppose $G = (V, E)$ is a simple graph where $|V| = n$. Suppose the vertices of G are listed arbitrarily as $v_1, v_2, \ldots, v_n$. The **adjacency matrix** $\boldsymbol{A}$ (or $\boldsymbol{A}_G$) of G, with respect to this listing of the vertices, is the n x 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. For adjacency matrix $\boldsymbol{A} = [a_{ij}]$, we have

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

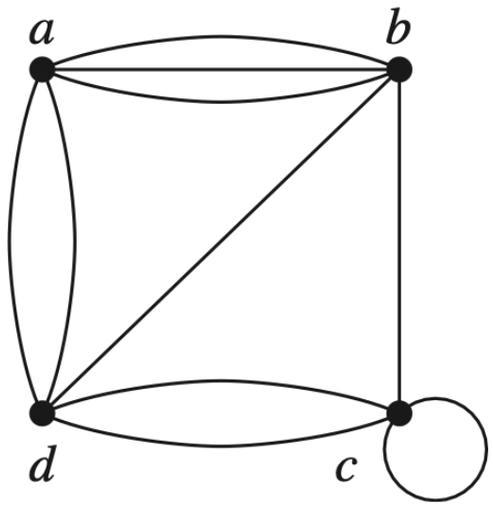


$$\begin{bmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{bmatrix}$$

Adjacency Matrices (Generalized)

- A generalized **adjacency matrix *A*** can also be used to represent undirected graphs with loops and with multiple edges.
  - The (i, i)-th entry of this matrix equals the number of loops at the vertex v_i.
  - The (i, j)-th entry of this matrix equals the number of edges that are associated to $\{v_i, v_j\}$.

- The above adjacency matrix ***A*** is no longer a zero–one matrix.

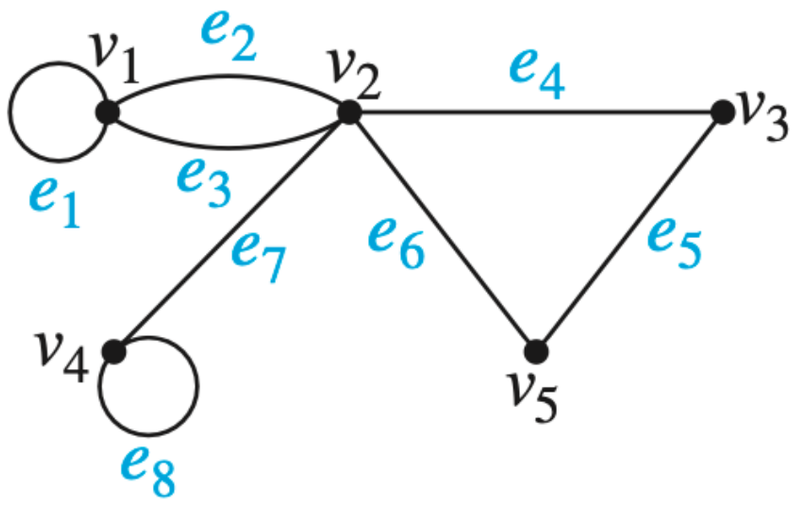


$$\begin{bmatrix} 0 & 3 & 0 & 2 \\ 3 & 0 & 1 & 1 \\ 0 & 1 & 1 & 2 \\ 2 & 1 & 2 & 0 \end{bmatrix}$$

Incidence Matrices

- **Definition:** Let $G = (V, E)$ be an undirected graph. Suppose that $v_1, v_2, \ldots, v_n$ are the vertices and $e_1, e_2, \ldots, e_n$ the edges of G. The **incidence matrix** with respect to this ordering of V and E is the $n \times m$ matrix $\boldsymbol{M} = [m_{ij}]$, where

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

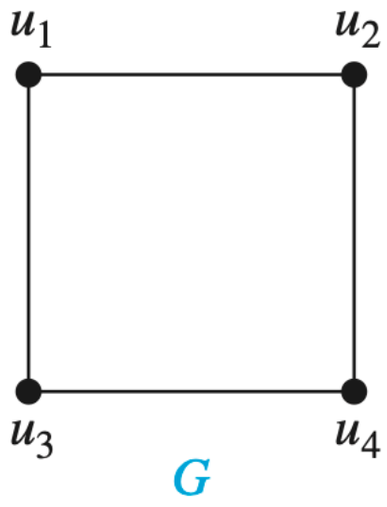


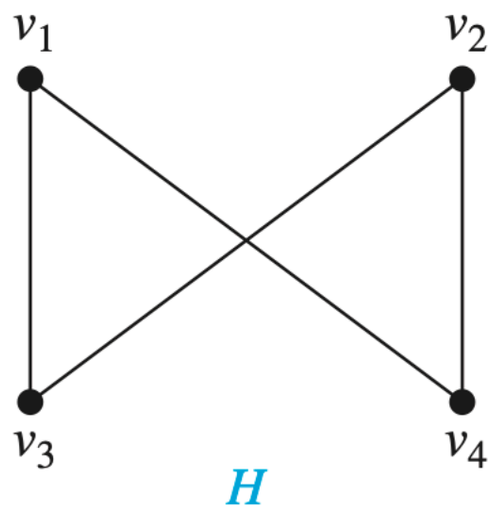
$$\begin{array}{c} \\ v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \end{array} \begin{array}{c} \begin{array}{cccccccc} e_1 & e_2 & e_3 & e_4 & e_5 & e_6 & e_7 & e_8 \end{array} \\ \left[\begin{array}{cccccccc} 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \end{array}\right] \end{array}$$

Isomorphism of Graphs

- **Definition:** The simple graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ are **isomorphic** if there exists a bijective function f from V_1 to V_2 with the property that, for all a and b in V_1, a and b are adjacent in G_1 if and only if $f(a)$ and $f(b)$ are adjacent in G_2.
  - Such a function f is called an **isomorphism**.
  - Simple graphs that are not isomorphic are called **nonisomorphic**.

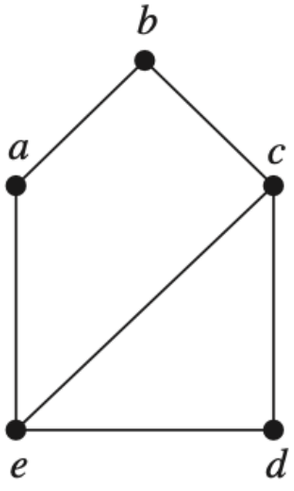
- Example: Are the following two graphs isomorphic?

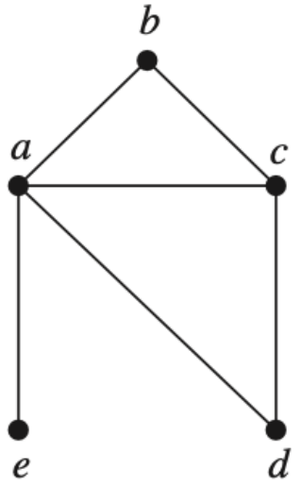


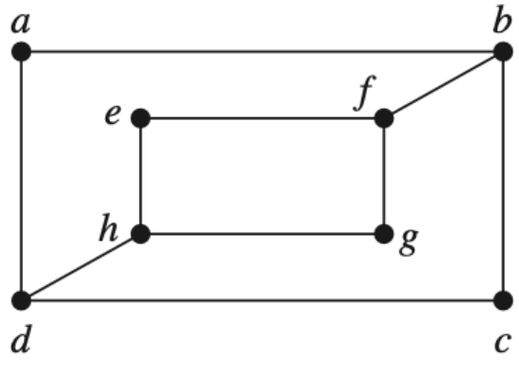


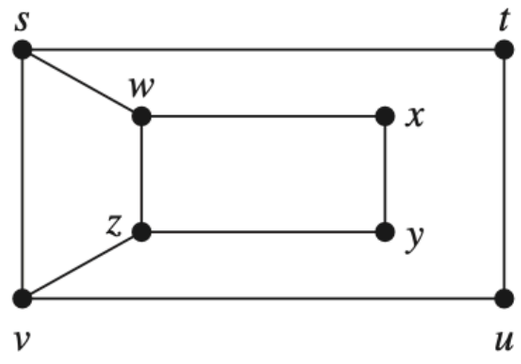
Isomorphism of Graphs

- It is usually difficult to determine whether two simple graphs are isomorphic using brute force since there are $n! = \Omega(2^n)$ possible one-to-one correspondences.
  - The graph isomorphism problem is in ***NP***, but is not known to be either in ***P*** or in ***NP***-complete.
- Sometimes, it is not difficult to show that two graphs are nonisomorphic, e.g., by checking the so-called graph invariants.
  - **Graph invariants** are properties preserved by a graph, e.g., the number of vertices, number of edges, degree sequence, etc.









same degree sequence but nonisomorphic

SUSTech

Connectivity

Paths in Undirected Graphs

- **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 $e_1, e_2, \ldots, e_n$ of G for which there exists a vertex sequence

  $$u = x_0, x_1, \ldots, x_{n-1}, x_n = v$$

  such that e_i has endpoints x_{i-1} and x_i, for $i = 1, \ldots, n$.

  - When the graph is simple, we denote this path by its vertex sequence $x_0, x_1, \ldots, x_n$ (because listing these vertices in order uniquely determines the path).

- The path is a **circuit** if it begins and ends at the same vertex, that is, $u = v$, and has length greater than zero.
- The path or circuit is said to **pass through** the vertices $x_1, x_2, \ldots, x_{n-1}$ or **traverse** the edges $e_1, e_2, \ldots, e_n$.
- A path or circuit is **simple** if it does not contain the same edge more than once.

Paths in Directed Graphs

- **Definition:** Let n be a nonnegative integer and G a directed graph. A **path** of length n from u to v in G is a sequence of n edges $e_1, e_2, \ldots, e_n$ of G for which there exists a vertex sequence

$$u = x_0, x_1, \ldots, x_{n-1}, x_n = v$$

such that e_i starts at x_{i-1} and end at x_i, for $i = 1, \ldots, n$.

  - When there are no multiple edges in the directed graph, we denote this path by its vertex sequence $x_0, x_1, \ldots, x_n$ (because listing these vertices in order uniquely determines the path).

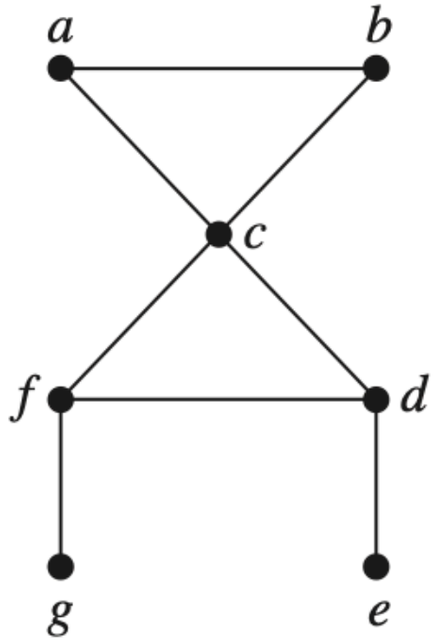
- The path is a **circuit** or **cycle** if it begins and ends at the same vertex, that is, $u = v$, and has length greater than zero.
- The path or circuit is said to **pass through** the vertices $x_1, x_2, \ldots, x_{n-1}$ or **traverse** the edges $e_1, e_2, \ldots, e_n$.
- A path or circuit is **simple** if it does not contain the same edge more than once.

Connected Undirected Graphs

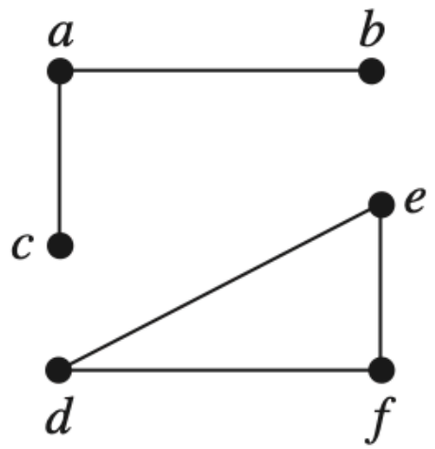
- **Definition:** An undirected graph is called **connected** if there is a path between every pair of distinct vertices of the graph.
  - An undirected graph that is not connected is called **disconnected**.
  - We say that we **disconnect** a graph when we remove vertices or edges, or both, to produce a disconnected subgraph.

- Example: Are the following graphs connected?

Yes



No



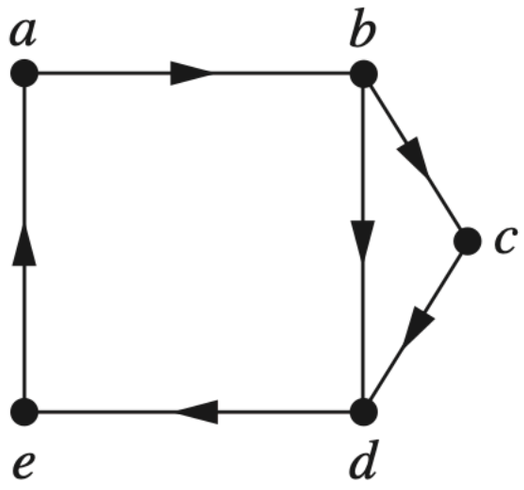
Connected Directed Graphs

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

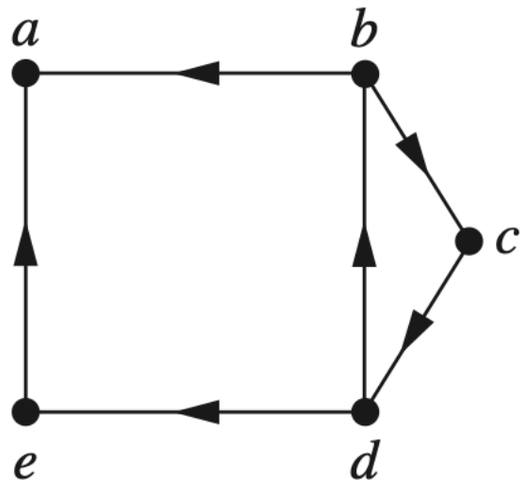
- **Definition:** A directed graph is **weakly connected** if the underlying undirected graph is connected.

- Example: Are the following graphs strongly connected?

Yes

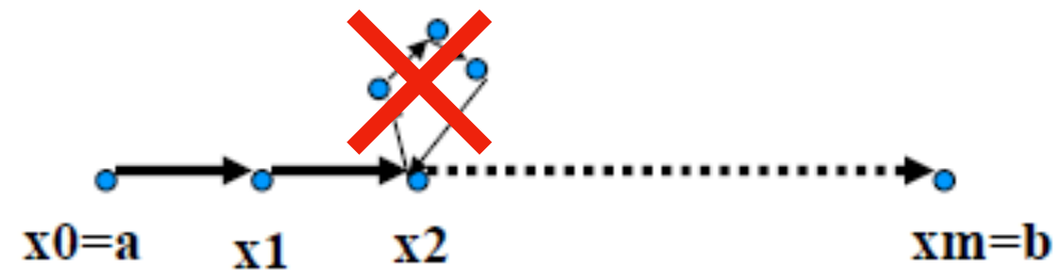


No



Theorem

- **Theorem**: There is a simple path between every pair of distinct vertices of a strongly connected directed graph.
  - The proof is very similar to the proof of a lemma in 08 Relations (just delete loops/cycles).

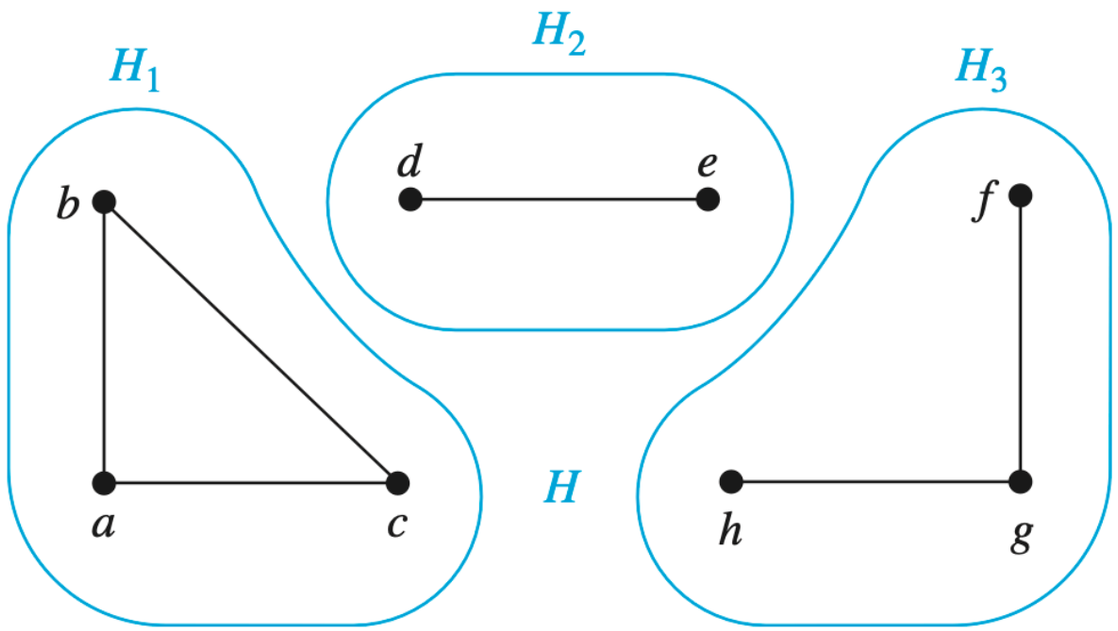


- **Note:** A similar theorem holds for undirected connected graphs.

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*. That is, a connected component of a graph *G* is a maximal connected subgraph of *G*.
  - A graph *G* that is not connected has two or more connected components that are disjoint and have *G* as their union.

- Example: The following graph *H* has *3* connected components.



SUSTech

Cut Vertices and Cut Edges

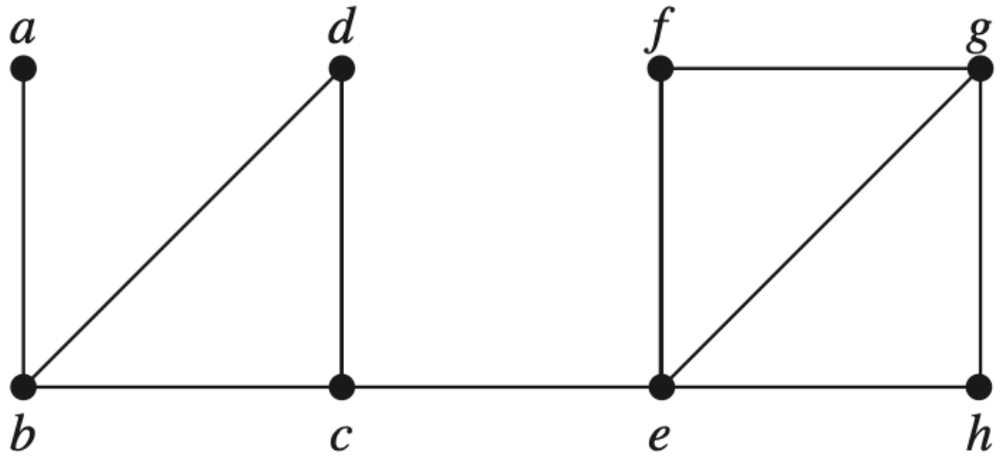
- **Definition:** If the removal from a graph of a vertex and all incident edges produces a subgraph with more connected components, then such vertices are called **cut vertices**.
  - In particular, the removal of a **cut vertex** from a connected graph produces a **disconnected** subgraph.

- **Definition:** An edge whose removal produces a subgraph with more connected components is called a **cut edge** or **bridge**.

- Example: Which vertices/edges are cut vertices/edges?

b, c, e

{a, b}, {c, e}

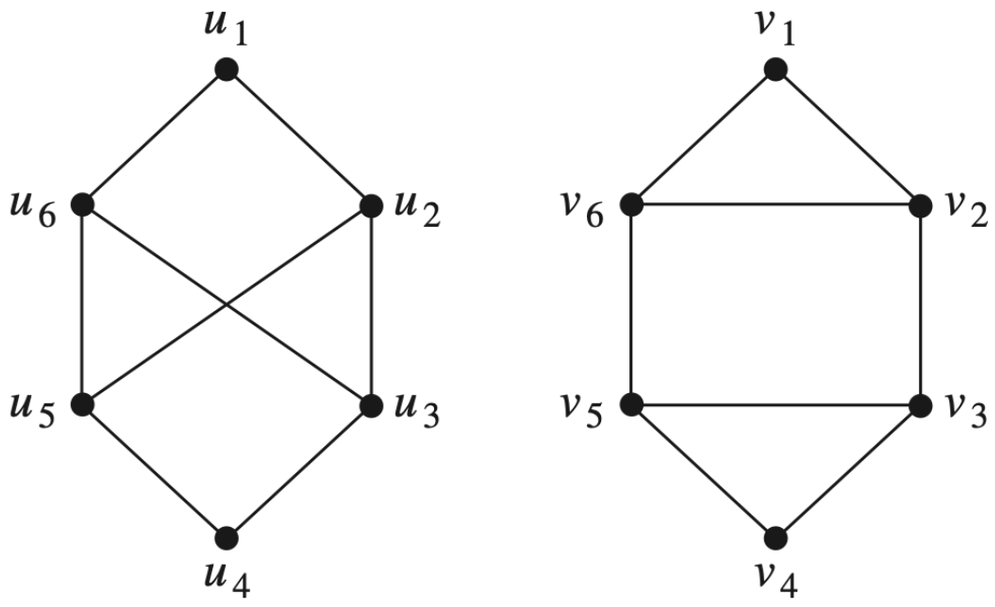


SUSTech

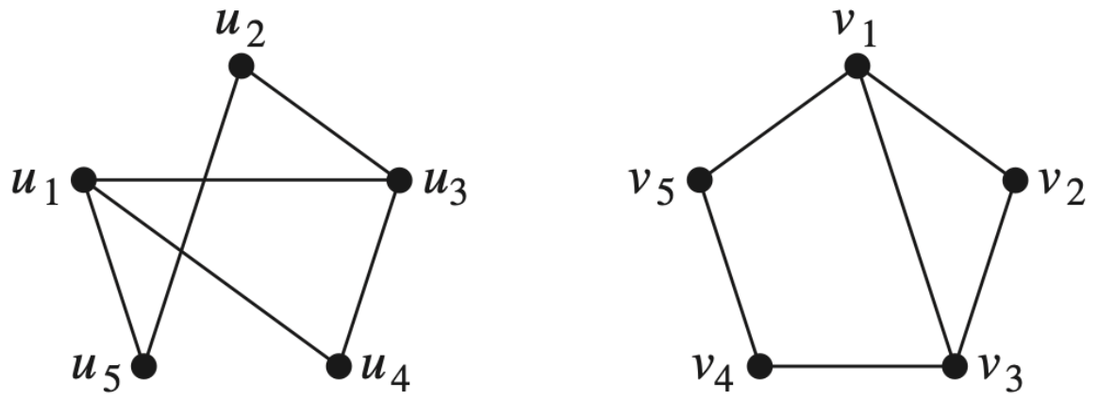
Paths and Isomorphism

- There are several ways that paths and circuits can help determine whether two graphs are isomorphic.
  - The existence of a simple circuit of length k is a useful invariant.
  - Paths can help construct mappings that may be isomorphisms.
- Example: Are these graphs isomorphic?

No



Yes



Counting Paths between Vertices

- **Theorem:** Let G be a graph with adjacency matrix $\boldsymbol{A}$ with respect to the ordering $v_1, v_2, \ldots, v_n$ of the vertices of the graph (with directed or undirected edges, with multiple edges and loops allowed). The number of different paths of length r from v_i to v_j, where r is a positive integer, equals the (i, j)-th entry of $\boldsymbol{A}^r$.

- Proof by induction:
  - **Basis step:** obviously true for $r = 1$ by definition
  - **Inductive step:** Since $\boldsymbol{A}^{r+1} = \boldsymbol{A}^r \boldsymbol{A}$, the (i, j)-th entry of $\boldsymbol{A}^{r+1}$ equals

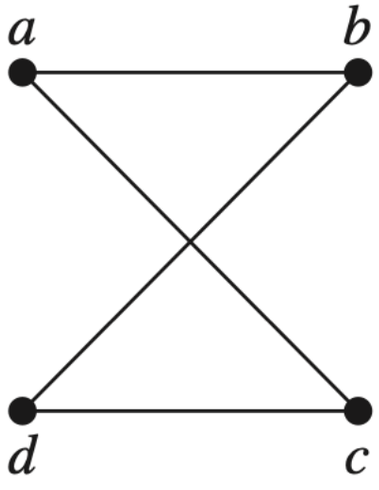
$$b_{i1}\, a_{1j} + b_{i2}\, a_{2j} + \cdots + b_{in}\, a_{nj} ,$$

    where b_{ik} is the (i, k)-th entry of $\boldsymbol{A}^r$ and a_{kj} is the (k, j)-th entry of $\boldsymbol{A}$.

    By the inductive hypothesis, the (i, k)-th entry of $\boldsymbol{A}^r$ is the number of different paths of length r from v_i to v_k. Therefore, the above sum is the number of different paths of length $r + 1$ from v_i to v_j.

Counting Paths between Vertices

- **Theorem:** Let G be a graph with adjacency matrix $\boldsymbol{A}$ with respect to the ordering $v_1, v_2, \ldots, v_n$ of the vertices of the graph (with directed or undirected edges, with multiple edges and loops allowed). The number of different paths of length r from v_i to v_j, where r is a positive integer, equals the (i, j)-th entry of $\boldsymbol{A}^r$.
- Example: How many paths of length 4 are there from a to d?

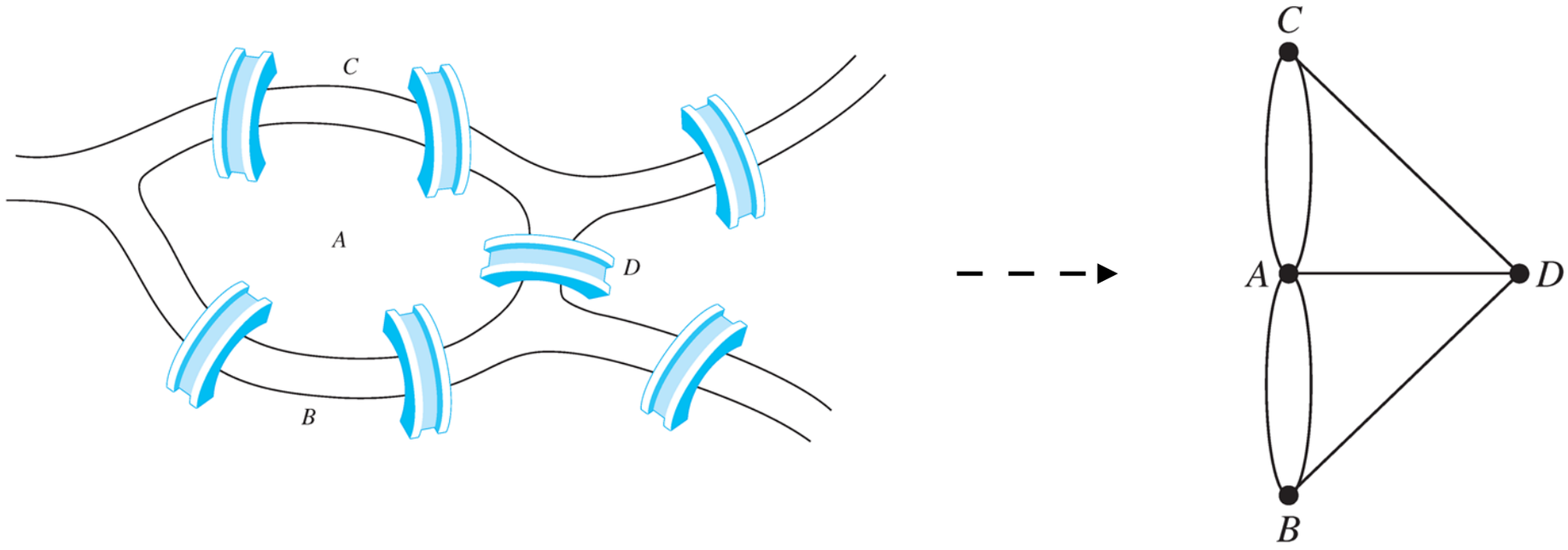


$$\mathbf{A} = \begin{bmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{bmatrix} \qquad \mathbf{A}^4 = \begin{bmatrix} 8 & 0 & 0 & 8 \\ 0 & 8 & 8 & 0 \\ 0 & 8 & 8 & 0 \\ 8 & 0 & 0 & 8 \end{bmatrix}$$

Euler and Hamilton Paths

Seven Bridges of Königsberg

- **Seven Bridges of Königsberg Problem:** People wondered whether it was possible to start at some location in the town, travel across all the bridges exactly once.



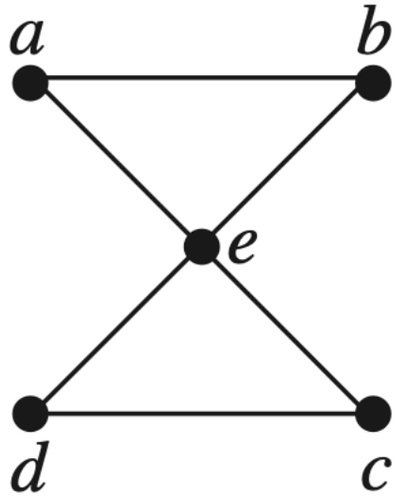
In 1736, **Leonhard Euler** proved this problem has no solution:

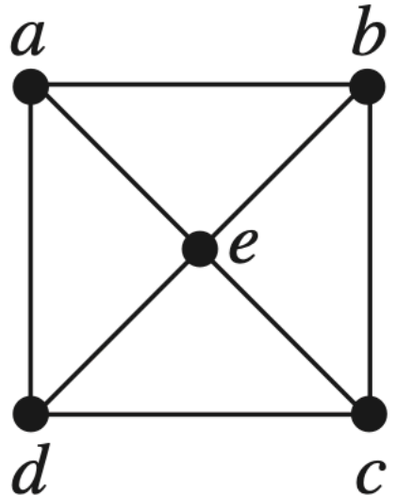
- An Euler walk (traversing each edge once) exists if and only if the graph is connected and it has exactly 0 or 2 nodes of odd degree.

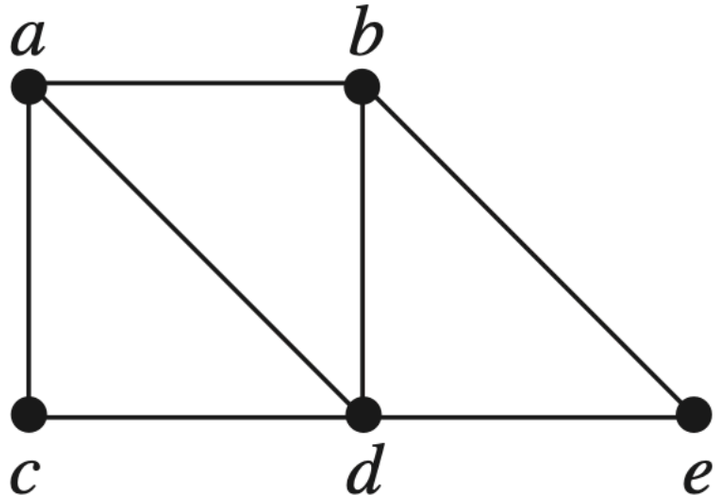
Euler Circuits and Euler Paths

- **Definition:** An **Euler circuit** in a (directed or undirected) 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 1: Which of the following undirected graphs have an Euler circuit? Of those that do not, which have an Euler path?



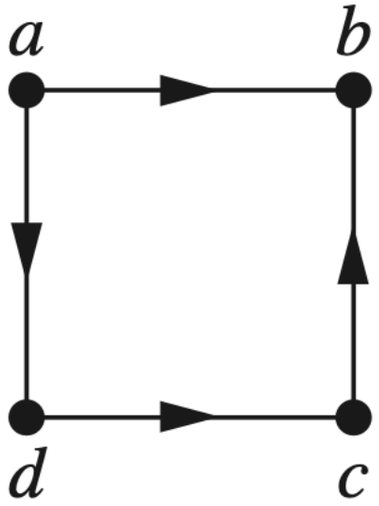


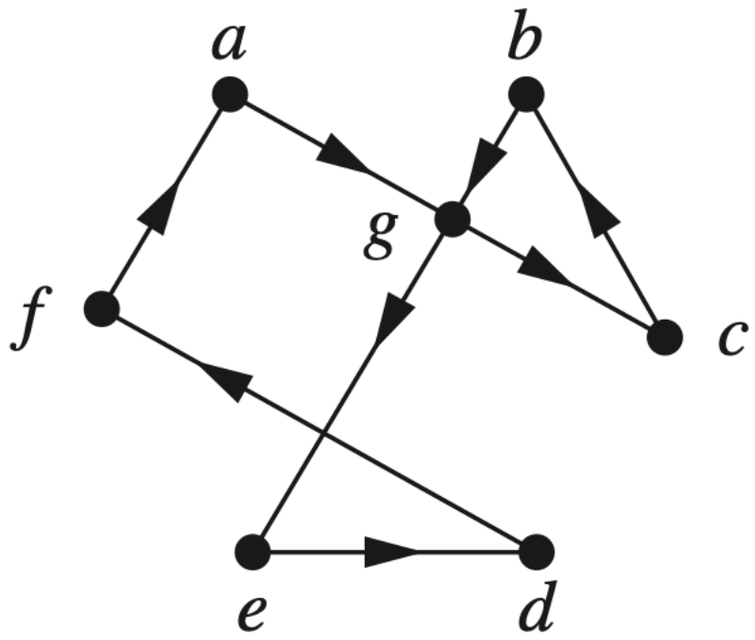


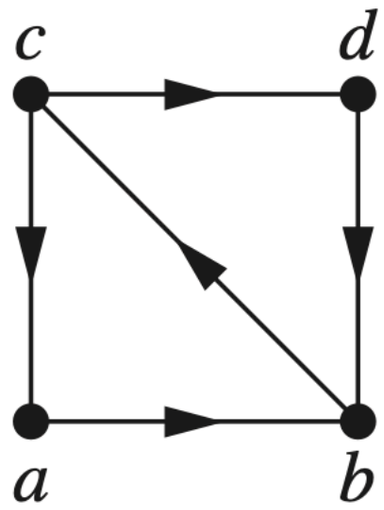
Euler Circuits and Euler Paths

- **Definition:** An **Euler circuit** in a (directed or undirected) 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 2: Which of the following directed graphs have an Euler circuit? Of those that do not, which have an Euler path?







Theorems and the “only if” Proofs

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

- Proof of the **“only if” part**:
  - Each time the circuit passes through a vertex, it contributes two to the vertex’s degree.
  - Since the circuit starts with a vertex *a* and ends at *a*, it also contributes two to the degree of *a*.

- **Theorem:** A connected multigraph has an Euler path but not an Euler circuit if and only if it has exactly *2* vertices of odd degree.

- Proof of the **“only if” part**:
  - The initial and terminal vertices of an Euler path have odd degrees.

“if” Proofs: Finding Euler Circuits

- If all vertices of a connected multigraph G have even degrees, the following algorithm can construct 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 of *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}

find an arbitrary circuit C
** why always possible?*

$H = G - C$

find a circuit C’ in H such that C and C’ share some vertex
** why always possible?*

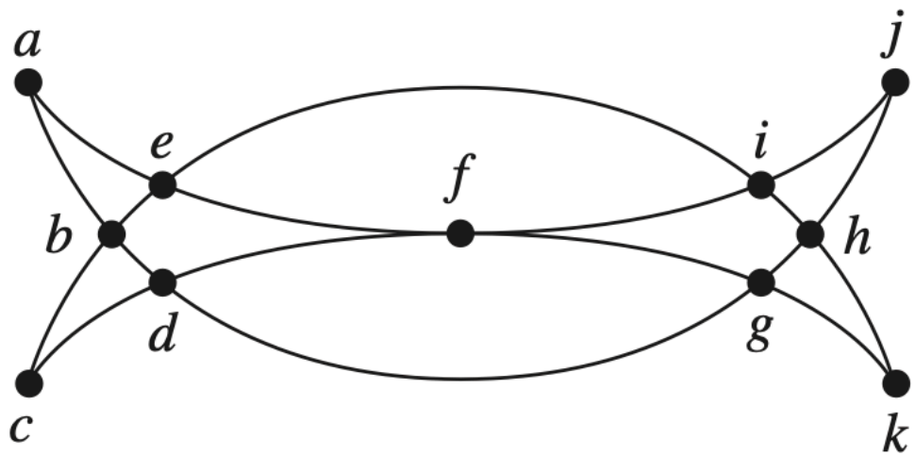
$H = H - C'$

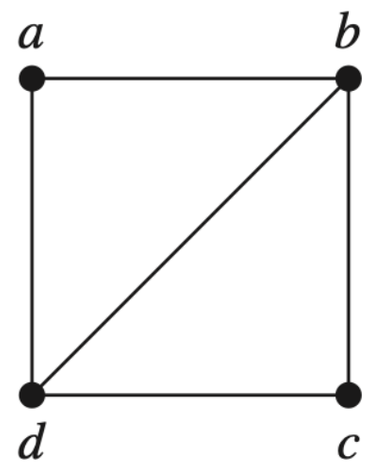
$C = C + C'$

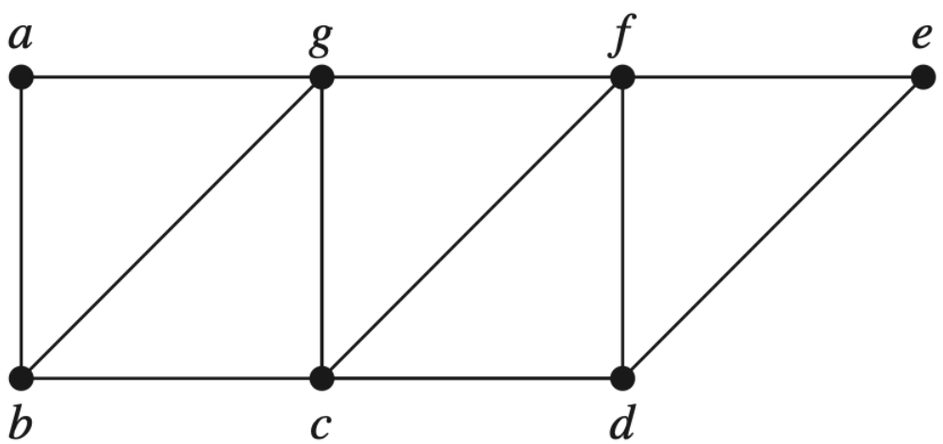
- How to construct Euler paths (that are not Euler circuits)?
  - add an edge between odd-degree vertices and run the algorithm

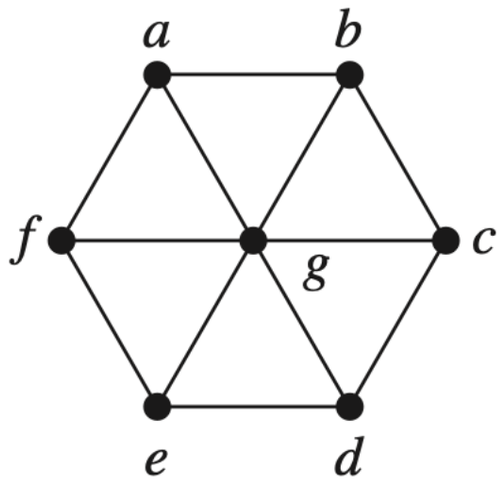
Examples

- Can you find Euler circuits or Euler paths in the following graphs?





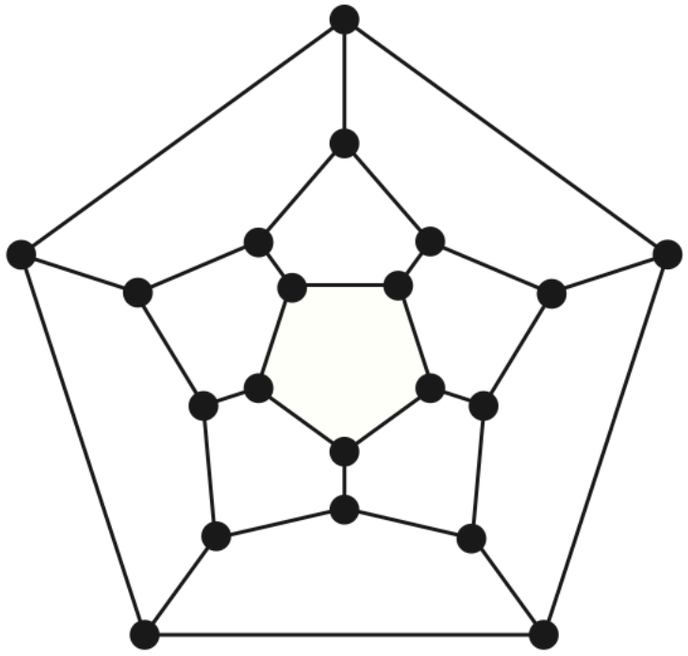


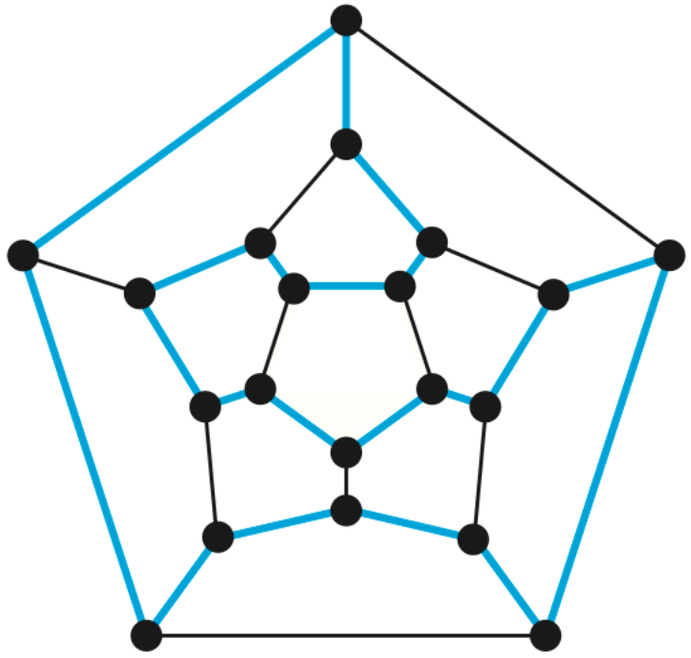


SUSTech

Hamilton Paths and Circuits

- **Definition:** A simple path in a (directed or undirected) 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**.

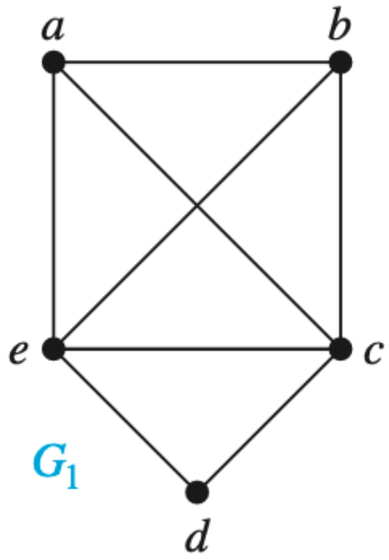




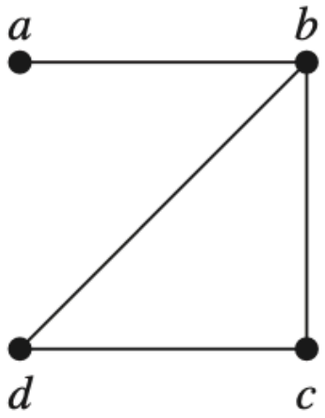
Hamilton Paths and Circuits

- **Definition:** A simple path in a (directed or undirected) 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**.

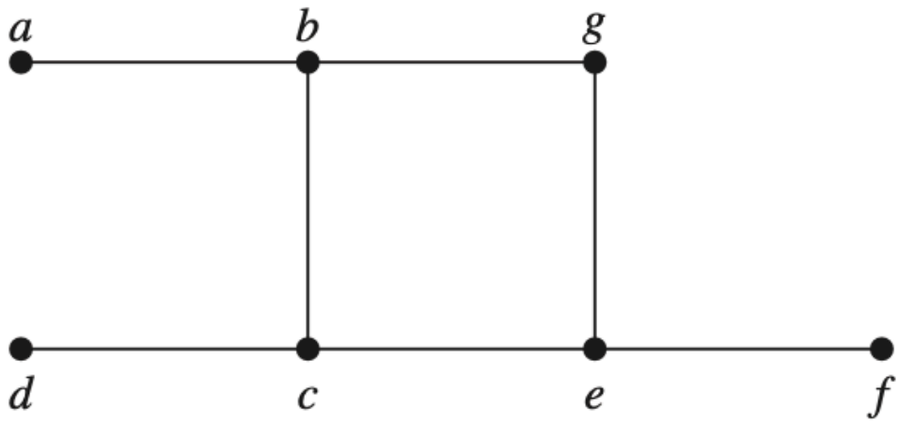
- Example: Which of the following simple graphs have a Hamilton circuit? Or, if not, a Hamilton path?



G_1



G_2



G_3

Sufficient Conditions

- So far, no simple necessary and sufficient conditions are known for the existence of a Hamilton circuit. But, there are some useful sufficient conditions.
  - Actually, the Hamilton path problem is ***NP***-complete.

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

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

  ** the proofs of the above theorems are advanced and out of the scope of our course*

Applications

- Applications of Euler paths and circuits:
  - Find a path or circuit that traverses each street in a neighborhood, each road in a transportation network, each connection in a utility grid, or each link in a communications network exactly once, etc.
  - The **Chinese Postman Problem** [**Meigu Guan** 60] asks for a circuit with the fewest edges that traverses every edge at least once.

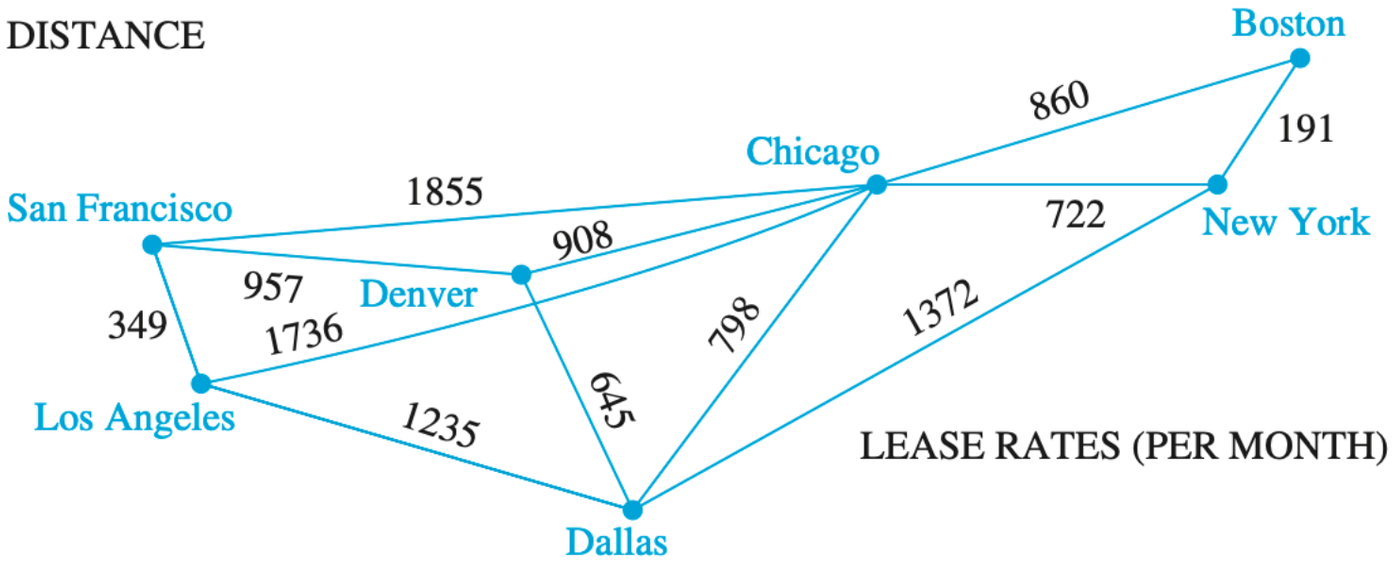
- Applications of Hamilton paths and circuits:
  - Find a path or circuit that visits each road intersection in a city, each place pipelines intersect in a utility grid, or each node in a communications network exactly once, etc.
  - The **Traveling Salesman Problem (TSP)** asks for the shortest route a traveling salesman should take to visit a set of cities. This problem reduces to finding a Hamilton circuit in a complete graph such that the total weight of its edges is as small as possible.

Shortest-Path Problems

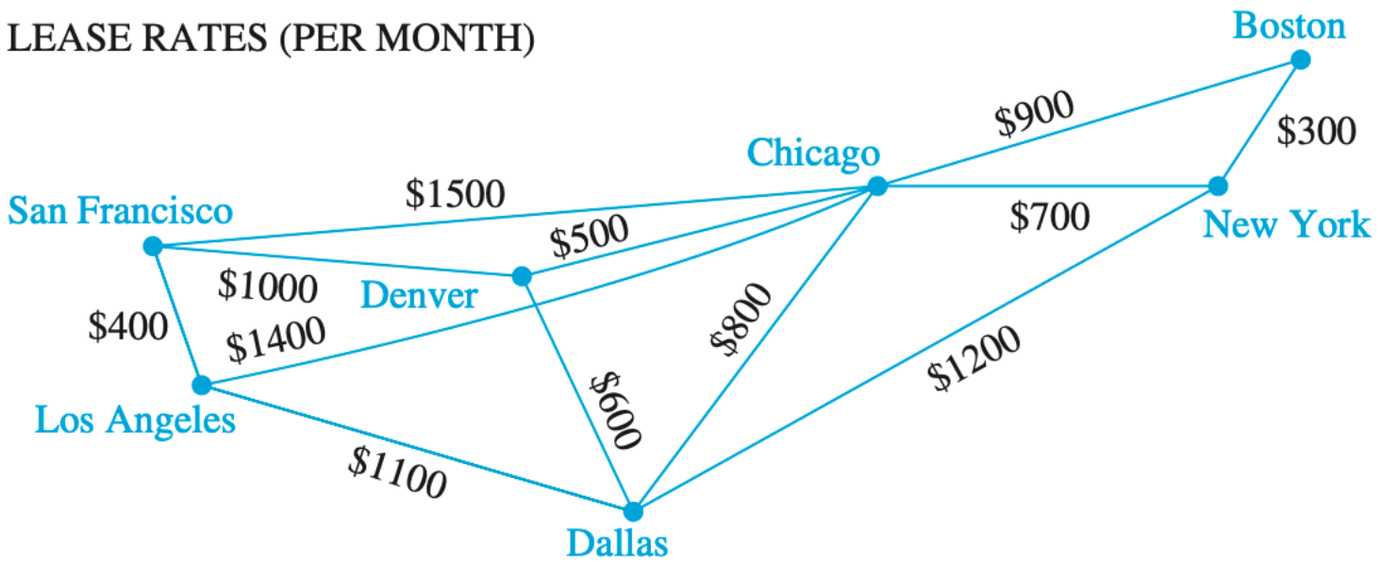
Weighted Graphs

- Graphs that have a number (or weight) assigned to each edge are called **weighted graphs**. The length of a path in a weighted graph is the sum of the weights of the edges of this path.

- Many problems can be modeled using weighted graphs.



*what is the **shortest path** (i.e., the path of least length) between two vertices?*



SUSTech

A Shortest-Path Algorithm

- A shortest path from any vertex *a* to any vertex *v* can be found by a brute force approach by examining the length of every path from *a* to *v*. However, this approach is impractical for humans and even for computers for graphs with a large number of edges.

- **Dijkstra's algorithm** finds the length of a shortest path between any two vertices *a* and *v* in a connected simple undirected weighted graph efficiently, when all weights are non-negative.
  - It can be easily adapted to find the shortest path.
  - It actually finds the shortest paths from *a* to all other vertices.

- **Edsger W. Dijkstra** **1972 Turing Award**
  - He made fundamental contributions to operating systems, programming languages, algorithms, etc.

Dijkstra's Algorithm

- Notation: *L(v)* = length of the shortest path from *a* to *v*

procedure *Dijkstra*(G: weighted connected simple graph, with all weights positive)
{G has vertices $a = v_0, v_1, \ldots, v_n = z$ and lengths $w(v_i, v_j)$ where $w(v_i, v_j) = \infty$ if $\{v_i, v_j\}$ is not an edge in G}

Dijkstra's algorithm actually finds L(v) for every vertex v

for $i := 1$ **to** n
  $L(v_i) := \infty$
$L(a) := 0$
$S := \emptyset$

initialize values in list L to infinity except that L(a) = 0
S consists of all vertices whose L(v) have been settled

{the labels are now initialized so that the label of a is 0 and all other labels are ∞, and S is the empty set}
while $z \notin S$
  $u :=$ a vertex not in S with $L(u)$ minimal
  $S := S \cup \{u\}$

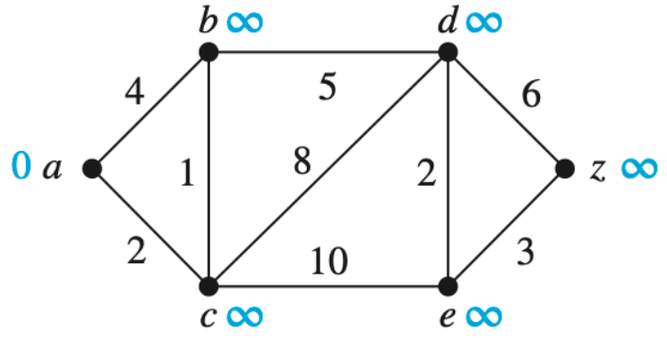
*add to S the vertex u ∉ S with **min** L(u)*

  for all vertices v not in S
    if $L(u) + w(u, v) < L(v)$ **then** $L(v) := L(u) + w(u, v)$
    {this adds a vertex to S with minimal label and updates the labels of vertices not in S}

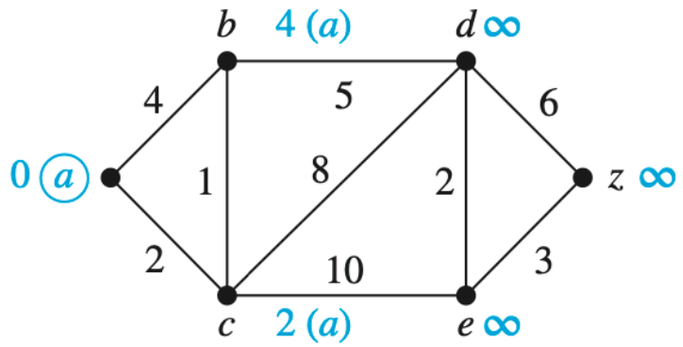
critical update step

return $L(z)$ {$L(z)$ = length of a shortest path from a to z}

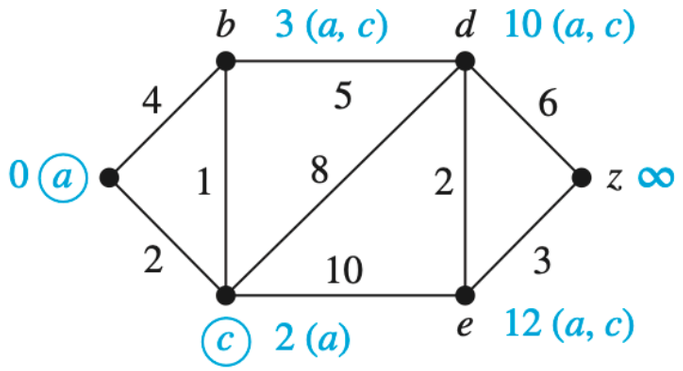
Dijkstra's Algorithm: Example



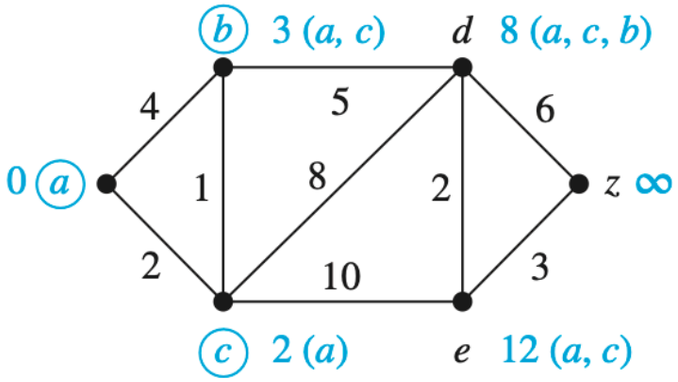
(a)



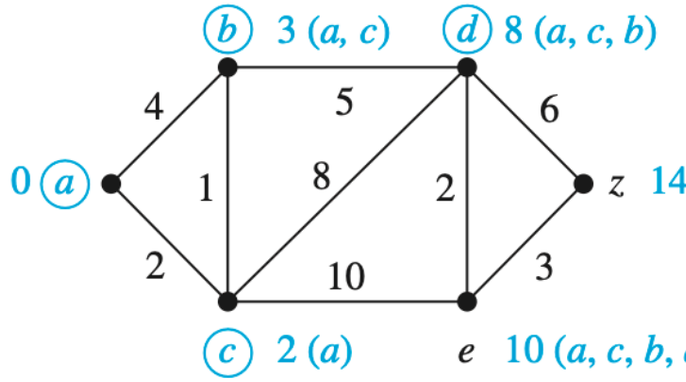
(b)



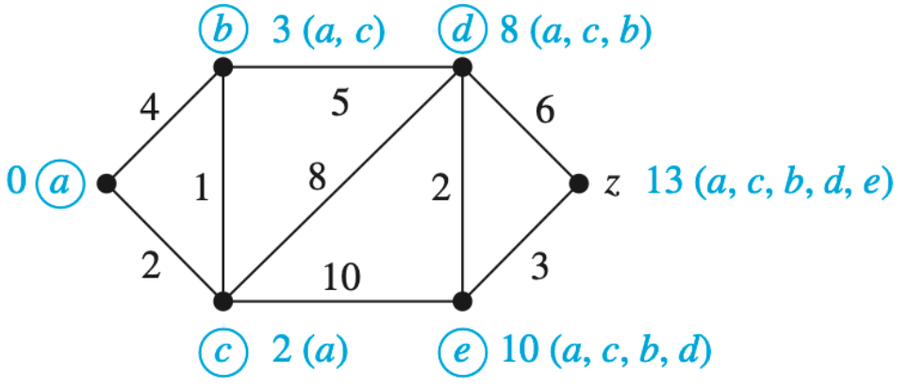
(c)



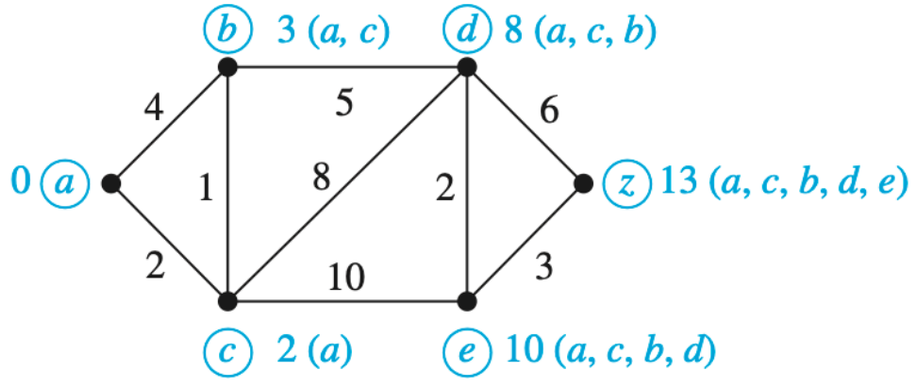
(d)



(e)



(f)



(g)

Correctness and Complexity

- **Theorem (correctness)**: Dijkstra's algorithm indeed finds the length of a shortest path between two vertices in a connected simple undirected (non-negative) weighted graph.

- Prove "$L(u)$ = min path length from a to u" by induction on $|S|$.
  - *see the textbook on pages 748-749 for details*

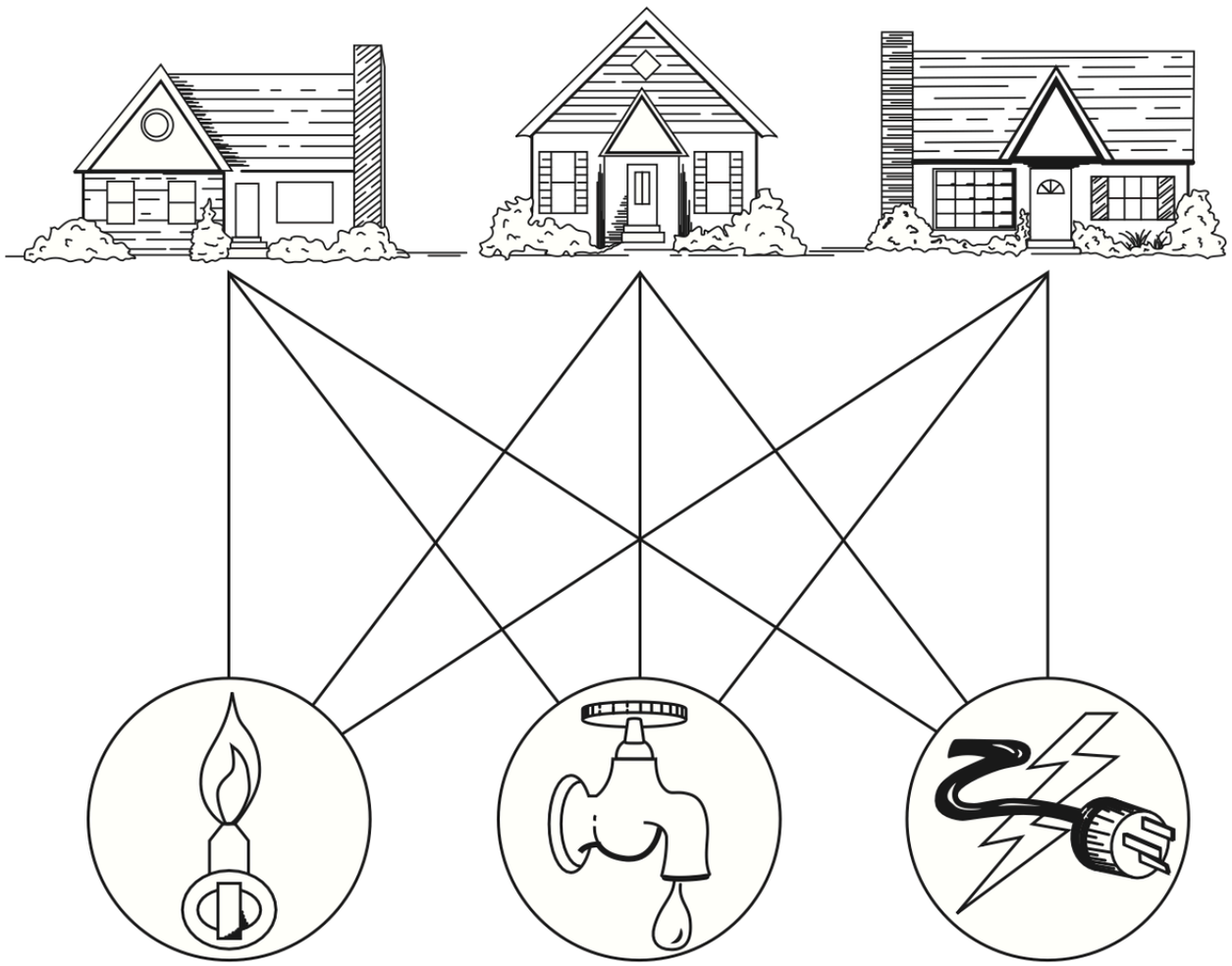
- **Theorem (complexity)**: Dijkstra's algorithm uses $O(n^2)$ operations (additions/comparisons) to find the length of a shortest path between two vertices in a connected simple undirected (non-negative) weighted graph with n vertices.

- Proof:
  - The algorithm runs in $\leq n$ iterations.
  - For each iteration there are $\leq n$ additions and $\leq n$ comparisons.

Planar Graphs

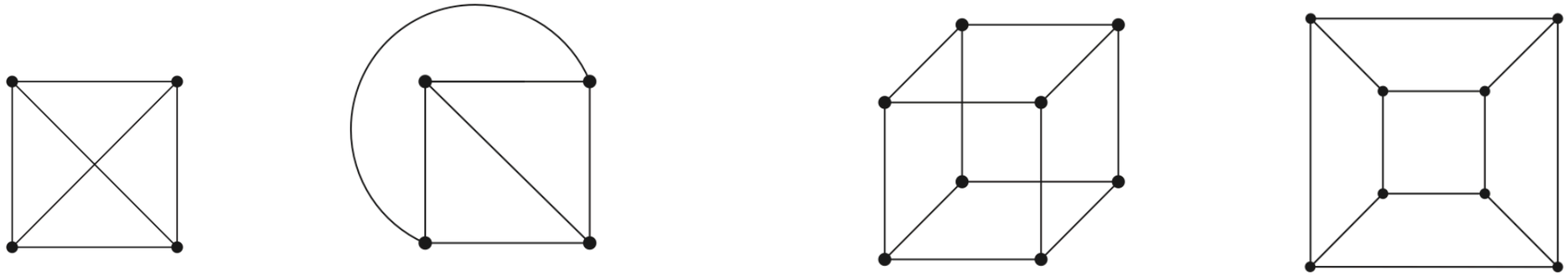
Example

- Consider the problem of joining three houses to each of three separate utilities. Can this graph $K_{3,3}$ be drawn in the plane so that no two of its edges cross?



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.
  - A graph is called **nonplanar** if it is not planar.
- Example: Are the following graphs (K_4 and Q_3) planar?



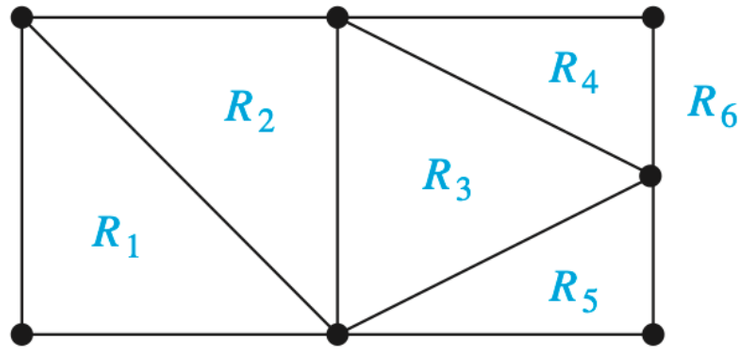
- Planarity of graphs is important in the following design:
  - electronic circuits: crossings require more layers
  - road networks: crossings require underpasses or overpasses

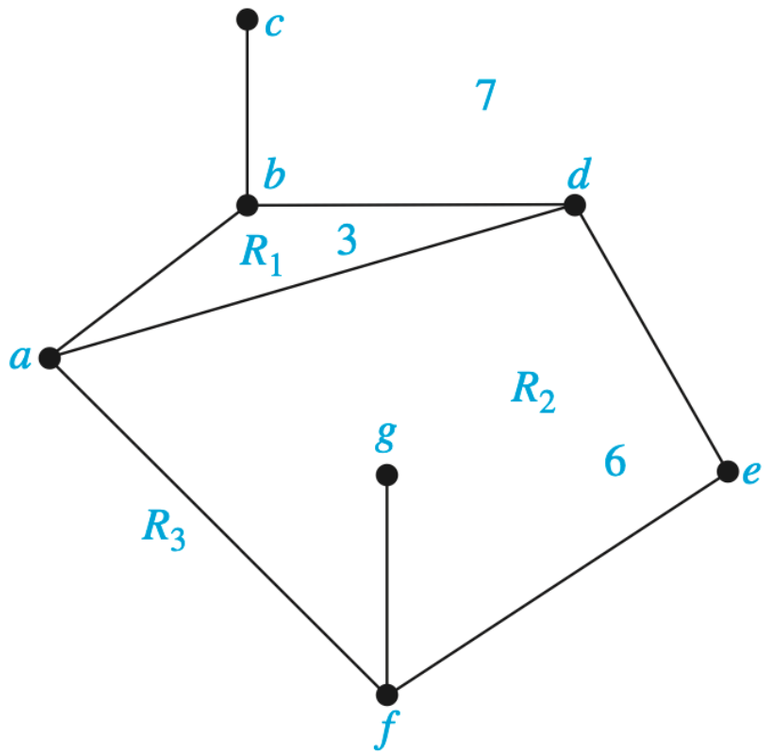
Regions and their Degrees

- **Definition:** A planar representation of a graph splits the plane into **regions**, including an unbounded region.

- **Definition:** The **degree** of a region R, denoted by $\deg(R)$, is the number of edges on the boundary of this region.
  - When an edge occurs twice on the boundary, it contributes 2 to the degree.

- **Lemma:** If a planar graph has e edges, then $2e = \sum_{\text{all regions } R} \deg(R)$.
  - *the proof is left as an exercise*

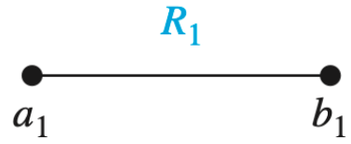




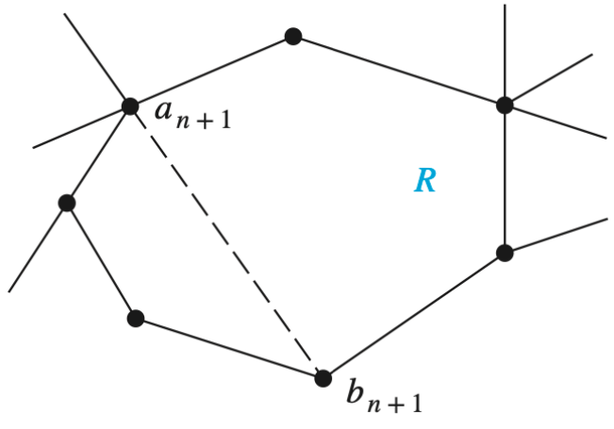
Euler's Formula

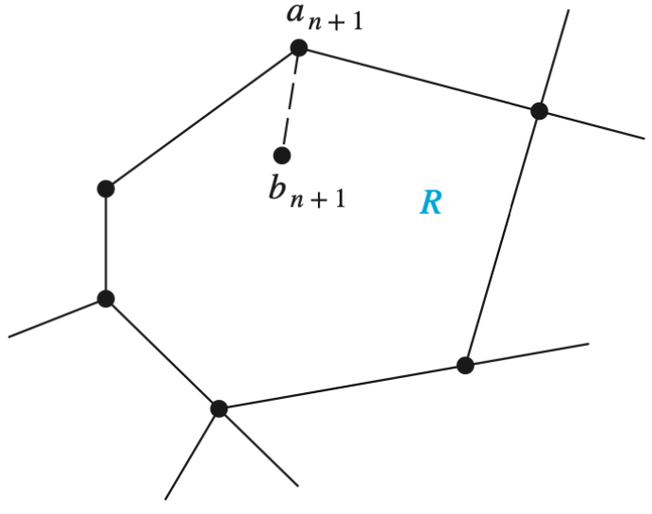
- **Euler's Formula:** Let G be a connected planar simple graph with e edges and v vertices. Let r be the number of regions in a planar representation of G. Then $r = e - v + 2$.

- Proof by induction (let G_n denote any such graph with n edges)
  - **Basis step:** obviously true for G_1 as $e = 1, v = 2, r = 1$



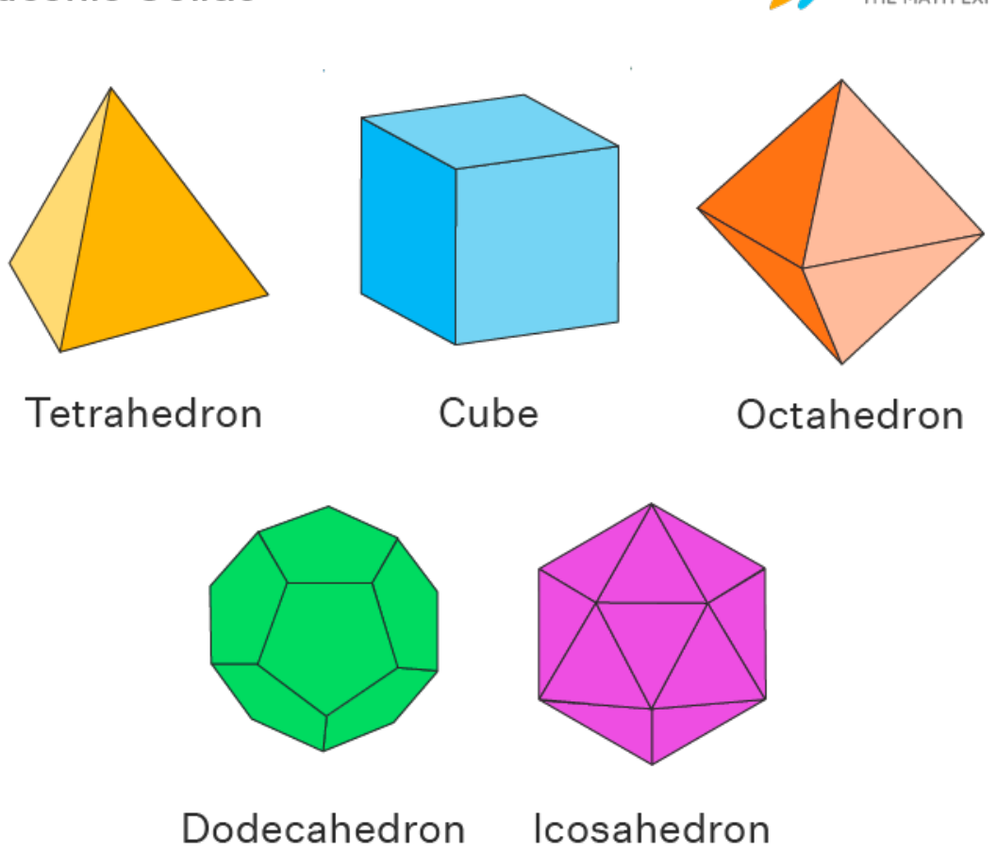
  - **Inductive step:** Let (a_{n+1}, b_{n+1}) denote the edge added to G_n to obtain G_{n+1}. This step is proved by considering two cases:





Example

- A **platonic solid** is a convex, regular polyhedron in *3D* Euclidean space, i.e., all the vertices have the same degree and all faces are the same regular polygon. Use **Euler's formula** $r = e - v + 2$ to show that there are only *5* such platonic solids. 正多面体
  Hint: the sum of all angles around each vertex is < 360°



Example

- A **platonic solid** is a convex, regular polyhedron in *3D* Euclidean space, i.e., all the vertices have the same degree and all faces are the same regular polygon. Use **Euler's formula** $r = e - v + 2$ to show that there are only *5* such platonic solids. 正多面体
  Hint: the sum of all angles around each vertex is $< 360°$

- Proof:

  - A regular polygon of $n \geq 3$ edges has every interior angle equal to $(n - 2)180°/n$. Since each vertex has $k \geq 3$ faces around it and the sum of angles around it is $< 360°$, we have $k(n - 2)180°/n < 360°$ and hence $n = 3, 4, 5$.
  - All region degrees are n and vertex degrees are k, so $nr = 2e = kv$.
  - If $n = 3$, the interior angle is $60°$, so $k \cdot 60° < 360°$ and $k = 3, 4, 5$. Then, with $nr = 2e = kv$, by **Euler's formula**, we solve $r = 4, 8, 20$.
  - Similarly, if $n = 4$, then $k = 3, r = 6$; if $n = 5$, then $k = 3, r = 12$.

Corollaries

- **Corollary 1:** If G is a connected planar simple graph with e edges and $v \geq 3$ vertices, then $e \leq 3v - 6$.
  - Every region has degree ≥ 3, so by the previous **Lemma**:

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

  - Plug the above $(2/3)\ e \geq r$ in **Euler's formula** $r = e - v + 2$.

- **Corollary 2:** If G is a connected planar simple graph, then it has a vertex of degree not exceeding 5.
  - implied by **Corollary 1** and the **handshaking theorem**.

- **Corollary 3:** If G is a connected planar simple graph with e edges and $v \geq 3$ vertices, and it has no circuits of length three, then $e \leq 2v - 4$.
  - similar to the proof of **Corollary 1** ** the proof is left as an exercise*

Examples

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

- Example: K_5 is nonplanar
  - K_5 has $v = 5$ vertices $e = 10$ edges, so $e = 10 > 9 = 3v - 6$.

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

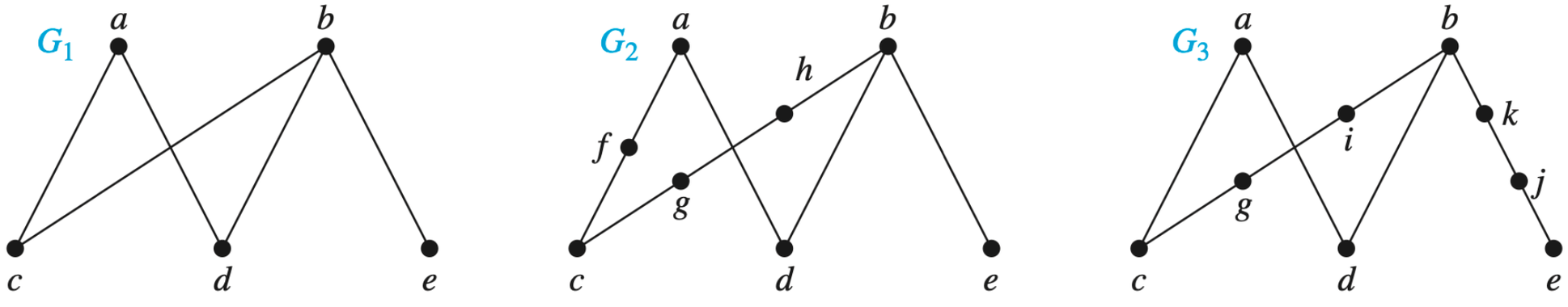
- Example: $K_{3,3}$ is nonplanar
  - As a bipartite graph, $K_{3,3}$ has no circuits of length three.
  - $K_{3,3}$ has $v = 6$ vertices $e = 9$ edges, so $e = 9 > 8 = 2v - 4$.

Homeomorphic Graphs

- **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**.

- **Definition:** 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.

- "homeomorphic" means "of similar shape" 同胚

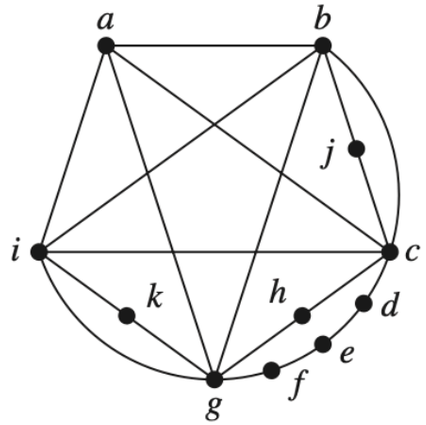


SUSTech

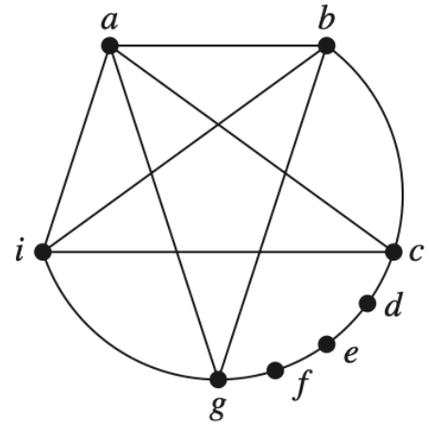
Kuratowski's Theorem

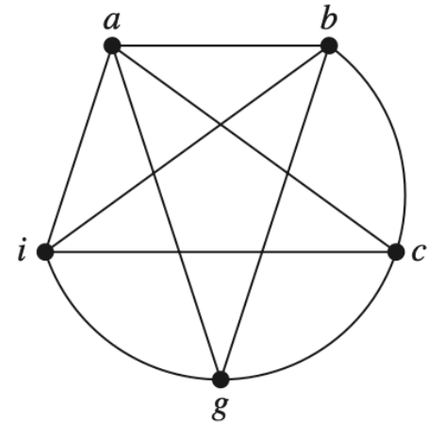
- **Kuratowski's Theorem:** A graph is nonplanar if and only if it contains a subgraph homeomorphic to $K_{3,3}$ or K_5.

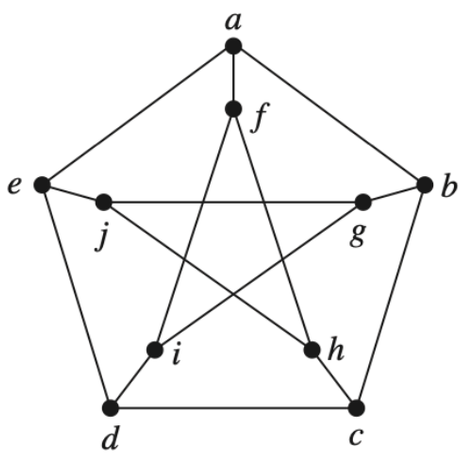
- Example: Are the following graphs planar?



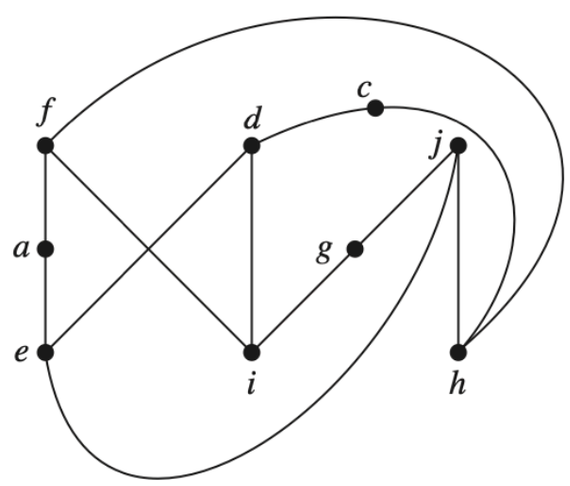
remove edges
b-j-c, i-k-g, g-h-c

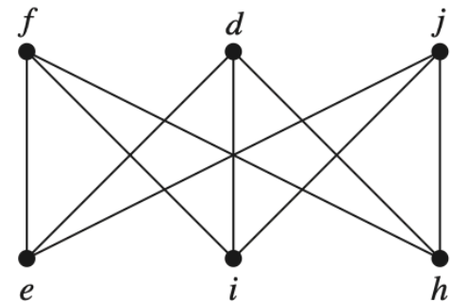






remove vertex b



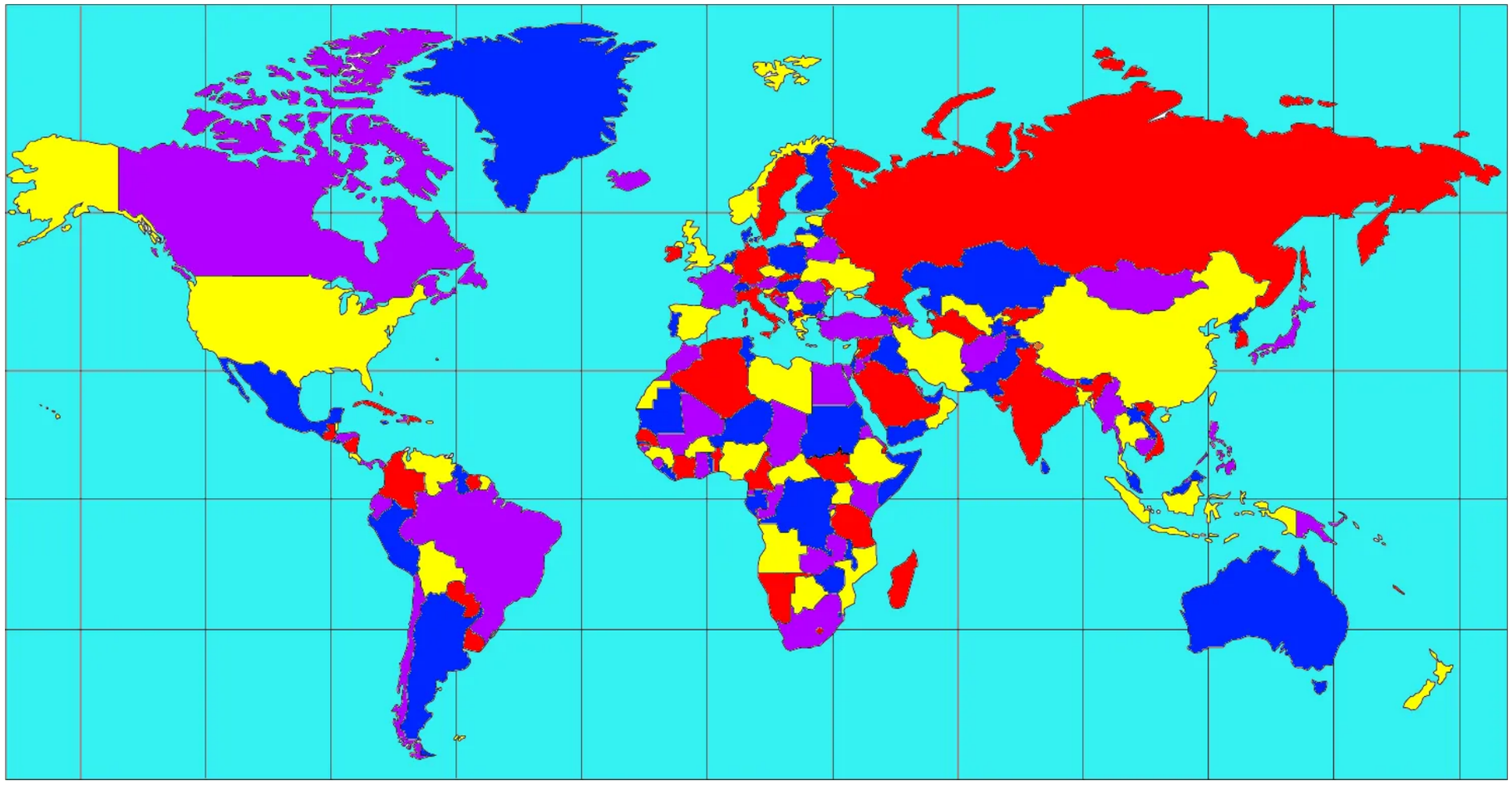


SUSTech

Graph Coloring

The Four-Color Theorem

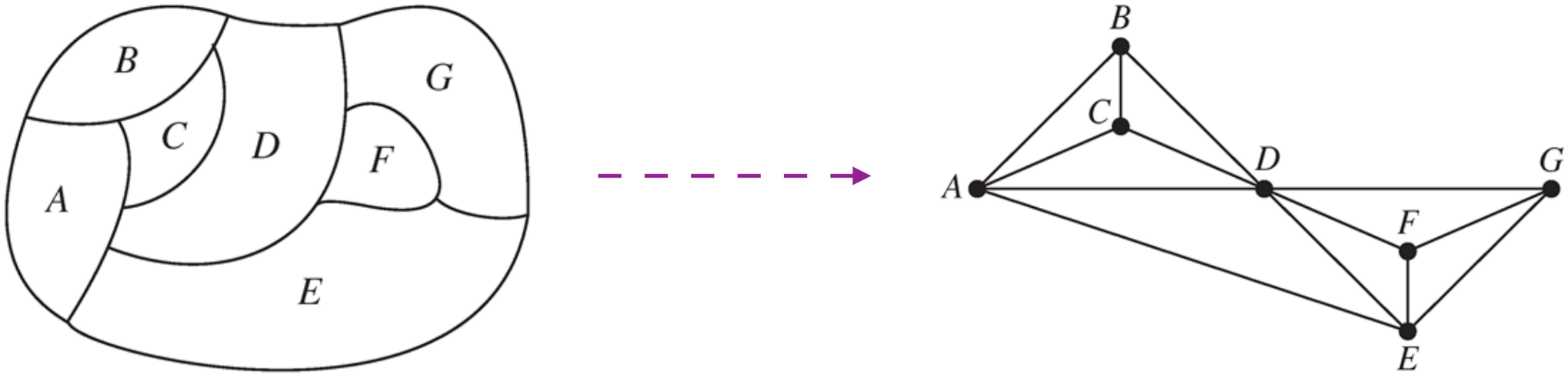
- **The Four-Color Theorem:** Given any separation of a plane into contiguous regions, producing a figure called a map, no more than four colors are required to color the regions of the map so that no two adjacent regions have the same color.



SUSTech

The Four-Color Theorem

- **The 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.



In 1976, **Kenneth Appel** and **Wolfgang Haken** proved it by case-by-case analysis using a computer — the first computer-aided proof.

Graph Coloring

- **Definition:** 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.

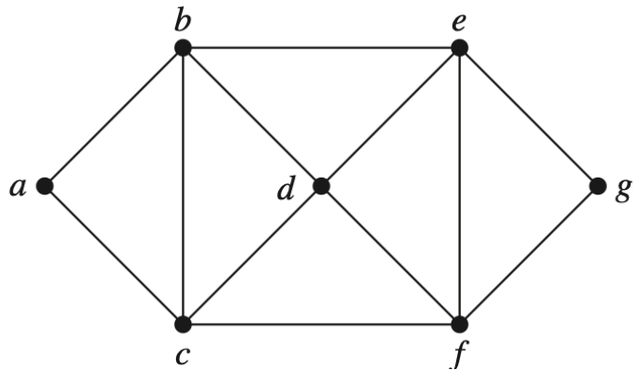
- **Definition:** The **chromatic number** of a graph G, denoted by $\chi(G)$ (here χ is the Greek letter chi), is the least number of colors needed for a coloring of this graph.

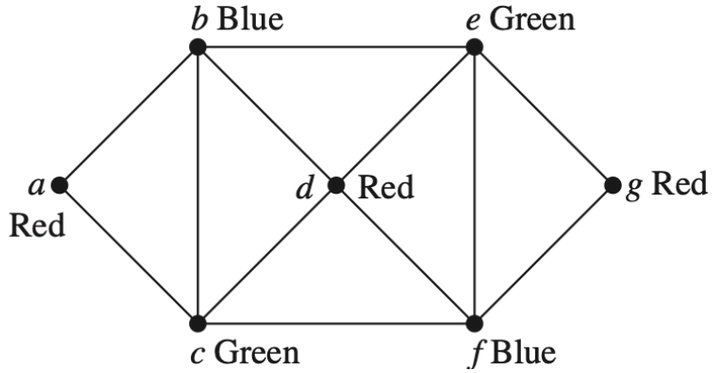
- **The Four-Color Theorem:** (rephrased with chromatic numbers) For any simple planar graph G, its chromatic number $\chi(G) \leq 4$.

- **Note:** Finding $\chi(G)$ of an arbitrary graph G is ***NP***-hard.

Examples

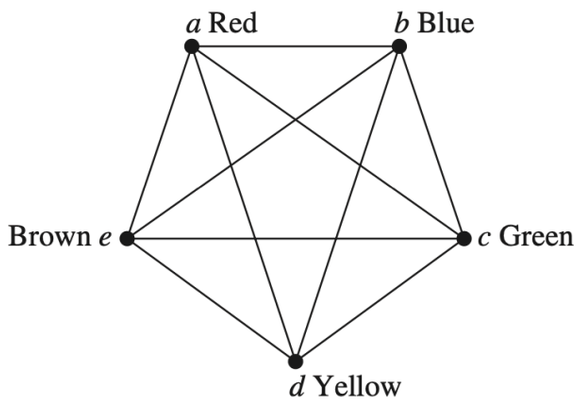
- What is the chromatic number of the following graph?



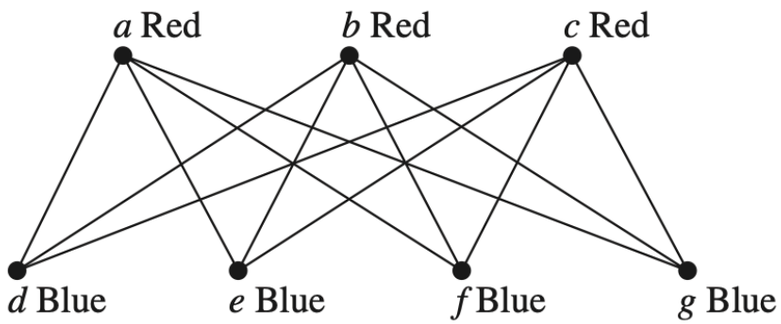


- What are the chromatic numbers of K_n, $K_{m,n}$, C_n?

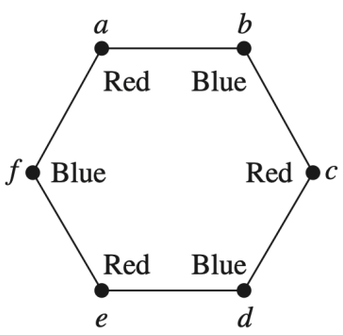
$\chi(K_n) = n$

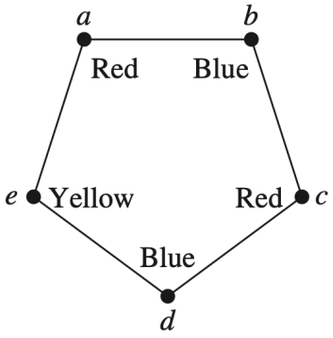


$\chi(K_{m,n}) = 2$



$\chi(C_n) = 2$ or 3





SUSTech

The Six-Color Theorem

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

- Proof by induction on the number of vertices:
  - Without loss of generality, assume the graph is connected.
  - **Basis step:** For a single-vertex graph, pick an arbitrary color.
  - **Inductive step:** Consider a planar graph G with $k + 1$ vertices.

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

    Recall **Corollary 2** implies that G has a vertex v of degree ≤ 5.

    By the inductive hypothesis, the graph $G - v$ can be 6-colored.

    Since v is adjacent to ≤ 5 vertices and hence ≤ 5 distinct colors, we have at least one remaining color to color v.

The Five-Color Theorem

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

- Proof by induction on the number of vertices:
  - Without loss of generality, assume the graph is connected.
  - **Basis step:** For a single-vertex graph, pick an arbitrary color.
  - **Inductive step:** Consider a planar graph G with $k + 1$ vertices.

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

    Recall **Corollary 2** implies that G has a vertex v of degree ≤ 5.

    By the inductive hypothesis, the graph $G - v$ can be 5-colored.

    If v is adjacent to ≤ 4 distinct colors, we have at least one remaining color to color v. Therefore, we are only left to prove the case where v is adjacent to 5 distinct colors.

The Five-Color Theorem

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

- Proof by induction on the number of vertices:
  - **Inductive step:** Consider a planar graph G with $k + 1$ vertices.

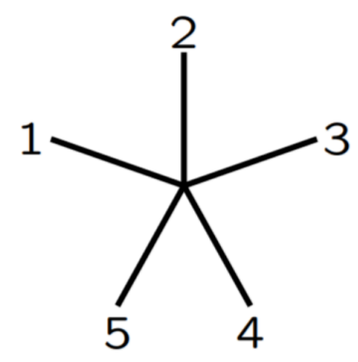
    By the inductive hypothesis, the graph $G - v$ can be 5-colored.

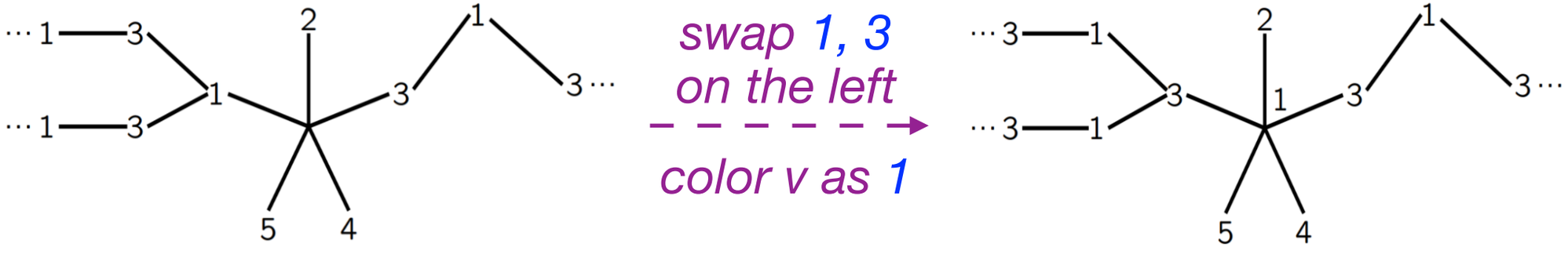
    Now, consider vertex v adjacent to 5 distinct colors.

    Let $v_1, \ldots, v_5$ denote these vertices with v_i colored i.

    Let G' be subgraph of G with vertices colored 1 or 3.

    Case 1: v_1 and v_3 are not connected by a path in G'





SUSTech

The Five-Color Theorem

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

- Proof by induction on the number of vertices:
  - **Inductive step:** Consider a planar graph G with $k + 1$ vertices.

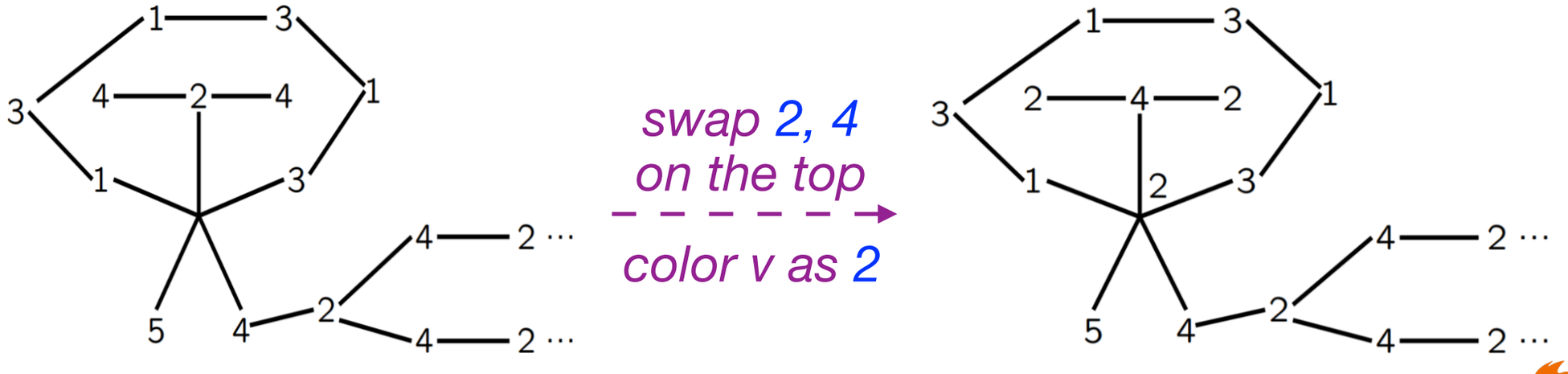
    By the inductive hypothesis, the graph $G - v$ can be 5-colored.

    Let G' be subgraph of G with vertices colored 1 or 3.

    Case 2: v_1 and v_3 are connected by a path in G'

    Let G'' be subgraph of G with vertices colored 2 or 4.

    Since G is planar, v_2 and v_4 are not connected by a path in G''.

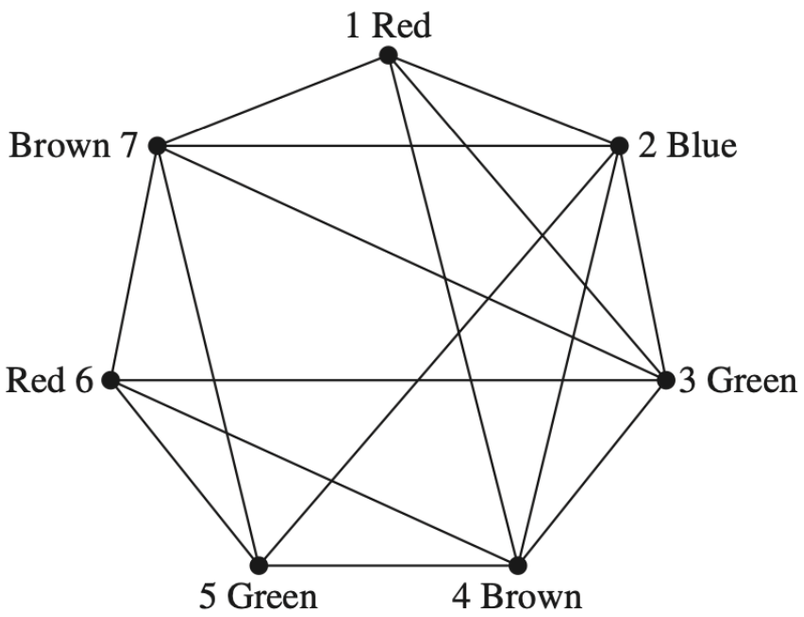


Applications of Graph Coloring

- Graph coloring has a variety of applications to problems involving scheduling and assignments.
  - Note that this does not lead to efficient algorithms because the graph coloring problem is ***NP***-hard.

- Example: How to schedule the final exams at a university such that no student has two exams at the same time?

  - Vertices represent courses, and there is an edge between two vertices if these courses have a common student.



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

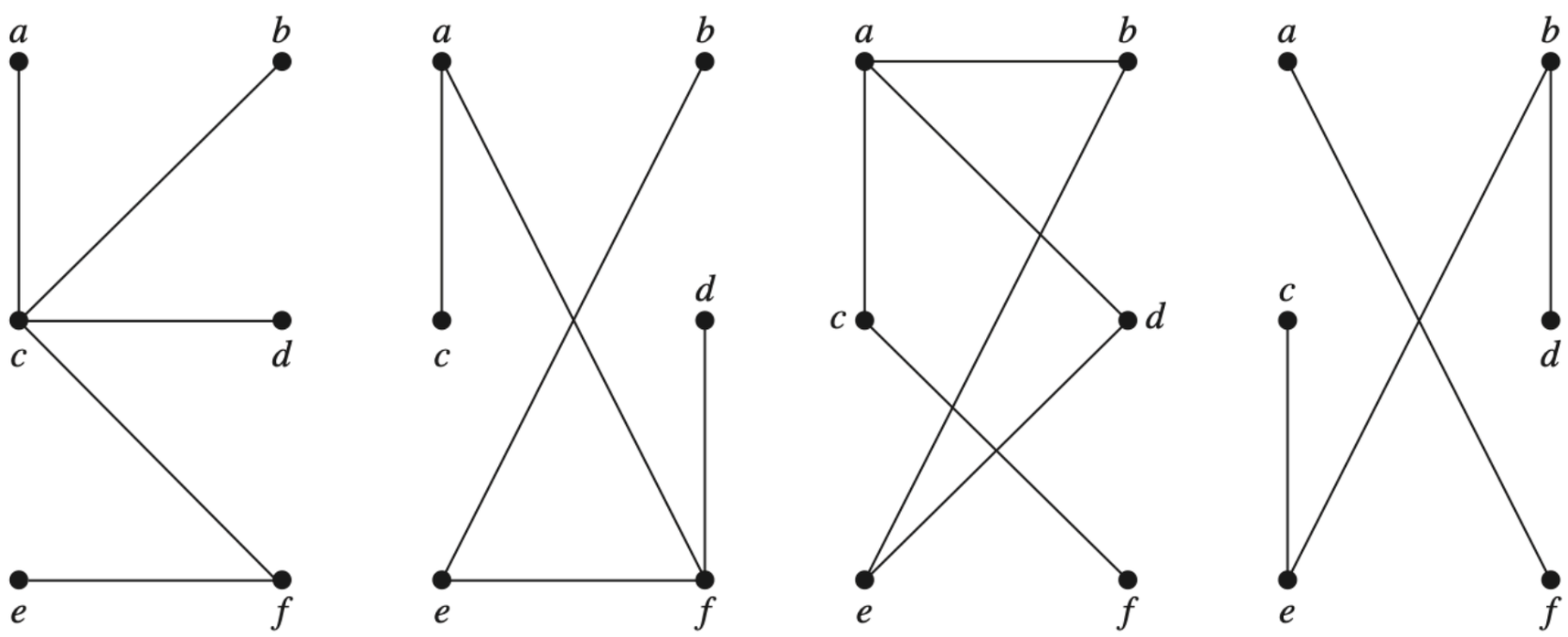
Trees

Trees

- **Definition:** A **tree** is a connected undirected graph without simple circuits (including loops).
  - That is, a tree is a connected simple graph without ~~simple~~ circuits.

- Example: Are the following graphs trees?

Yes ***Yes*** ***No*** ***No***



SUSTech

Properties of Trees

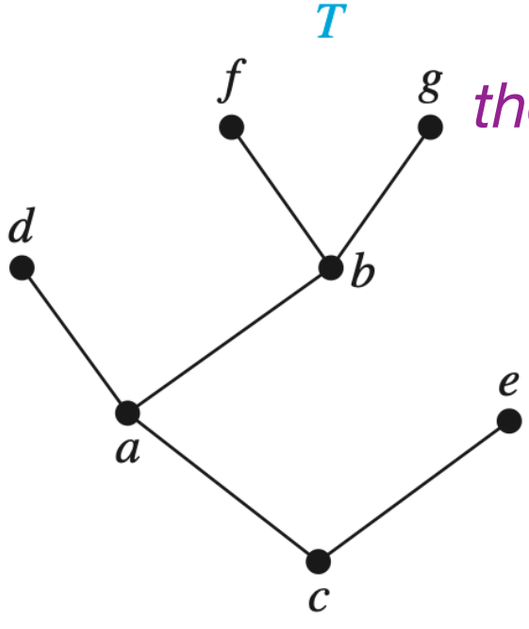
- **Theorem:** An undirected graph of at least two vertices is a tree if and only if there is a unique simple path between any pair of different vertices.

- **Theorem:** A tree with n vertices has $n - 1$ edges.

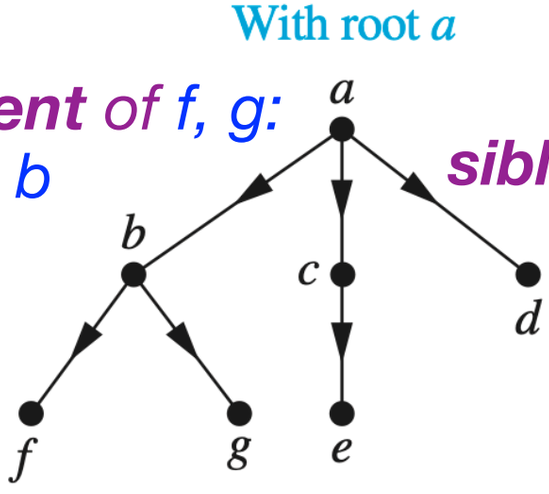
  ** the proofs of the above theorems are left as exercises*

Rooted Trees

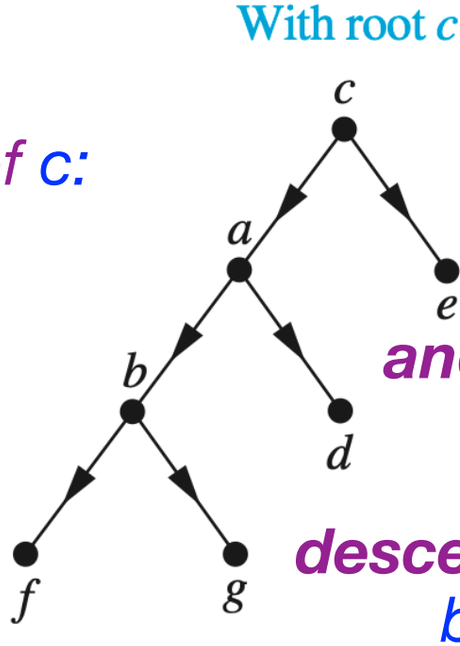
- **Definition:** A **rooted tree** is a tree in which one vertex has been designated as the **root** and every edge is directed away from the root.
  - Note that a rooted tree is still an undirected graph. The directed edges shown below are just for better understanding.



*the **parent** of f, g:*
b



***siblings** of c:*
b, d



***ancestors** of g:*
a, b, c

***children** of b:*
f, g

***descendants** of a:*
b, d, f, g

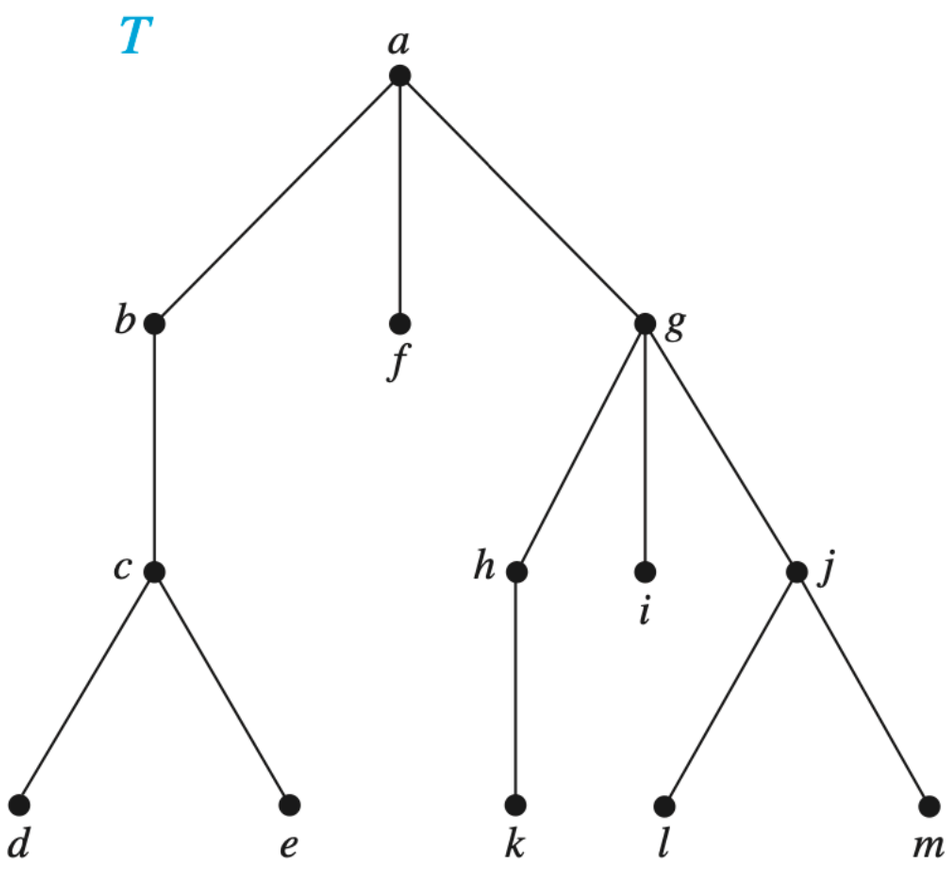
***internal vertices:** a, b, c*

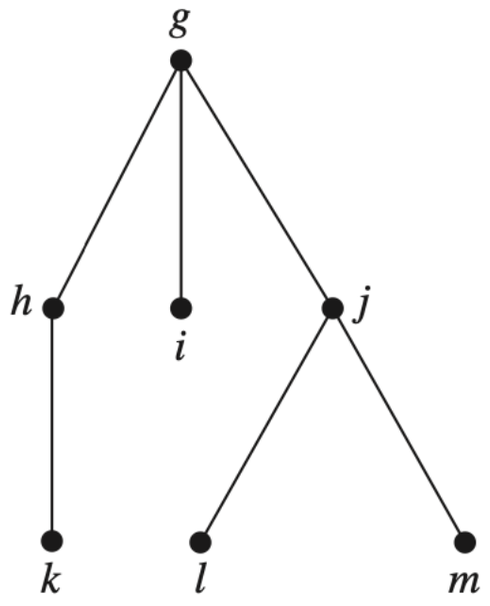
***leaves:** d, e, f, g*

Rooted Subtrees

- **Definition:** If *a* is a vertex in a rooted tree, the **subtree** with *a* as its root is the subgraph of the tree consisting of *a* and its descendants and all edges incident to these descendants.

- Example: What is the subtree of tree *T* with *g* as its root?



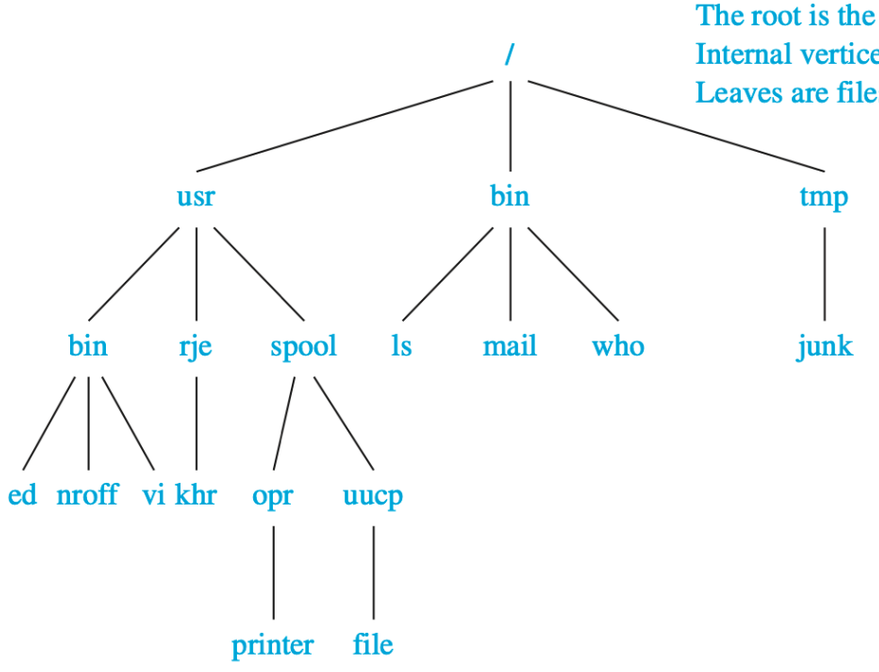


Trees as Models

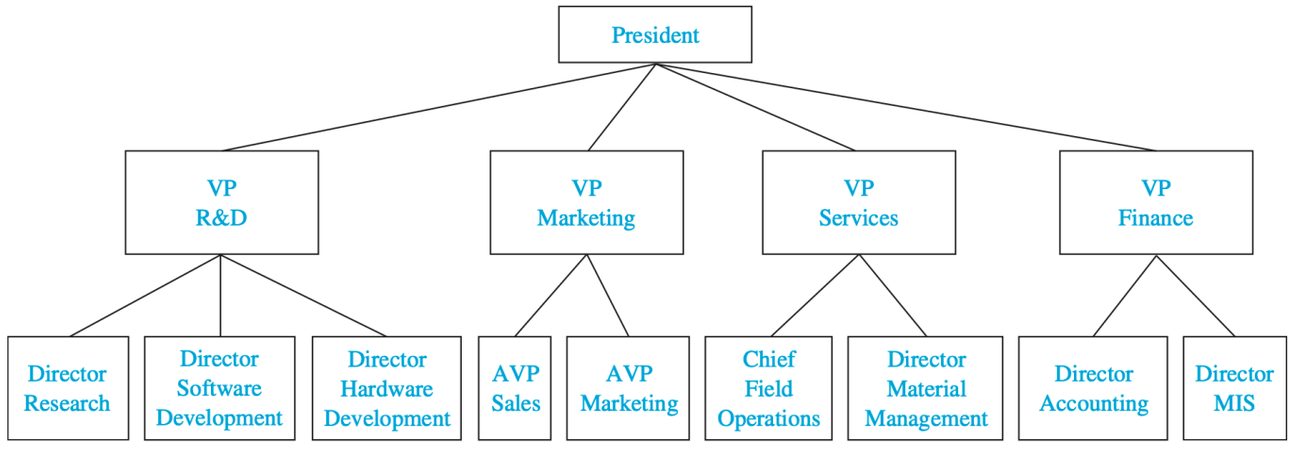
- Trees are used as models in computer science, chemistry, geology, botany, psychology, etc.

- Examples:

computer file systems



organizations



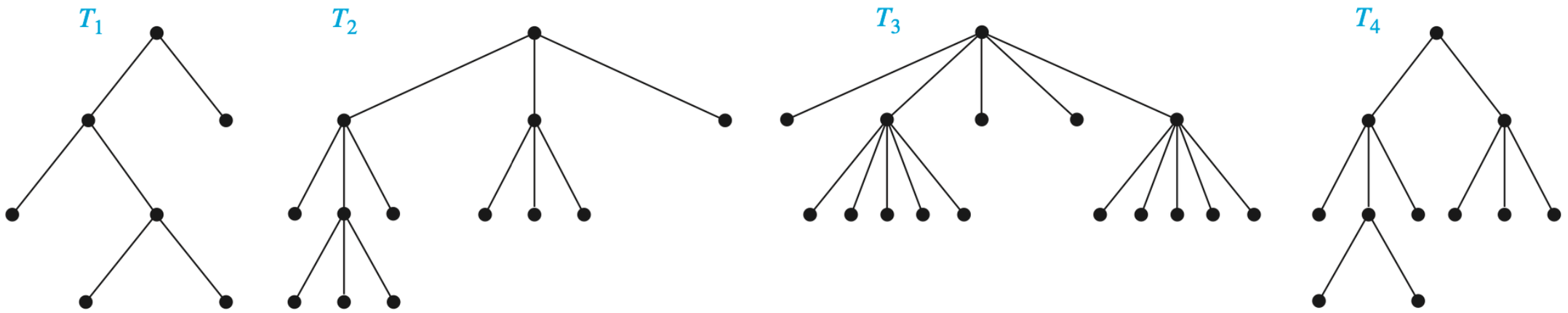
SUSTech

m-Ary Trees

- **Definition:** A rooted tree is called an *m*-**ary tree** if every internal vertex has no more than *m* children. It is called a **full** *m*-**ary tree** if every internal vertex has exactly *m* children. An *m*-**ary tree** with *m = 2* is called a **binary tree**.

- Example: Are the following rooted trees full *m*-ary trees for some positive integer *m*?

full binary ***full 3-ary*** ***full 5-ary*** ***No***



Counting Vertices in Full *m*-Ary Trees

- **Theorem:** A full m-ary tree with

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

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

  (iii) ℓ leaves
  has $n = (m\ell - 1)/(m - 1)$ vertices and $i = (\ell - 1)/(m - 1)$ internal vertices.

- Proof:
  - (ii) Every vertex, except the root, is the child of an internal vertex. By definition of a full m-ary tree, there are mi vertices in the tree excluding the root. Therefore, the tree contains $n = mi + 1$ vertices.
  - *the rest of the proof is left as an exercise (hint: $n = i + \ell$)*

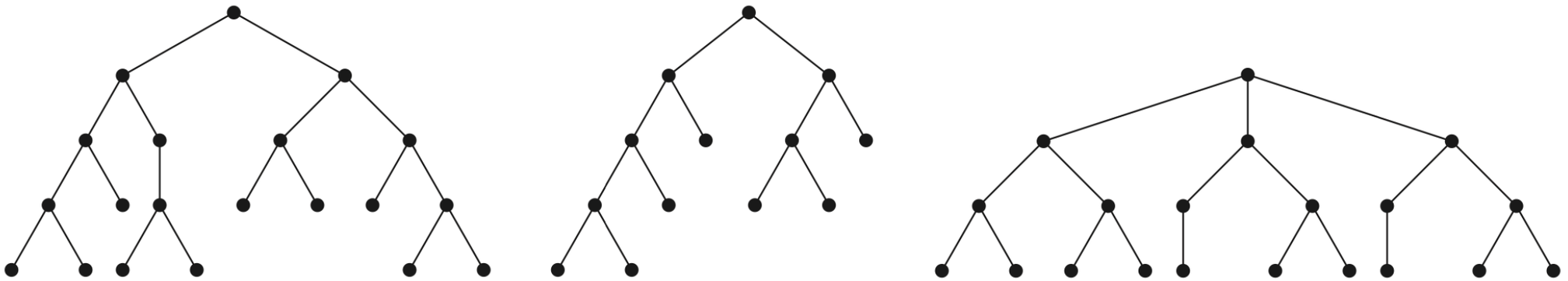
Balanced *m*-Ary Trees

- **Definition:** The **level** of a vertex *v* in a rooted tree is the length of the unique path from the root to this vertex. The level of the root is defined to be zero. The **height** of a rooted tree is the maximum of the levels of its vertices, i.e., the length of the longest path from the root to the leaves.

- **Definition:** A rooted *m*-ary tree of height *h* is **balanced** if all leaves are at levels *h* or *h* – *1*. Intuitively, this means subtrees at each vertex contain paths of approximately the same length.

- Examples: Which of the following rooted trees are balanced?

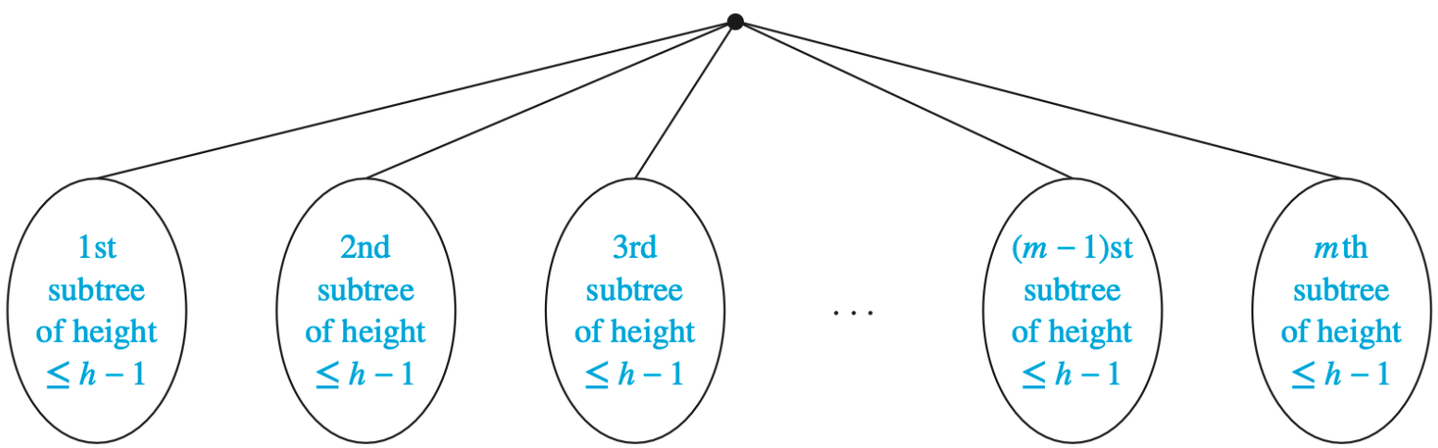
Yes ***No*** ***Yes***



T_1 T_2 T_3

Number of Leaves in *m*-Ary Trees

- **Theorem:** There are $\leq m^h$ leaves in an m-ary tree of height h.
- Proof by strong induction:
  - **Basis step:** obviously true for $h = 1$
  - **Inductive step:** Consider an m-ary tree of height h. Its leaves are the leaves of the subtrees obtained by deleting the root. By the inductive hypothesis, each of these subtrees has $\leq m^{h-1}$ leaves. Since there are $\leq m$ such subtrees, there are $\leq m \cdot m^{h-1} = m^h$ leaves.



Number of Leaves in *m*-Ary Trees

- **Corollary:** Consider any $m \geq 2$. 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:

  - From **theorem** we have $\ell \leq m^h$. Taking logarithms to the base m shows that $\log_m \ell \leq h$. Since h is an integer, we have $h \geq \lceil \log_m \ell \rceil$.

  - If the tree is full and balanced, then each leaf is at level h or $h - 1$, and there is at least one leaf at level h.

    A full and balanced m-ary tree has more than m^{h-1} leaves.

    ** the proof is left as an exercise (similar to that of the theorem)*

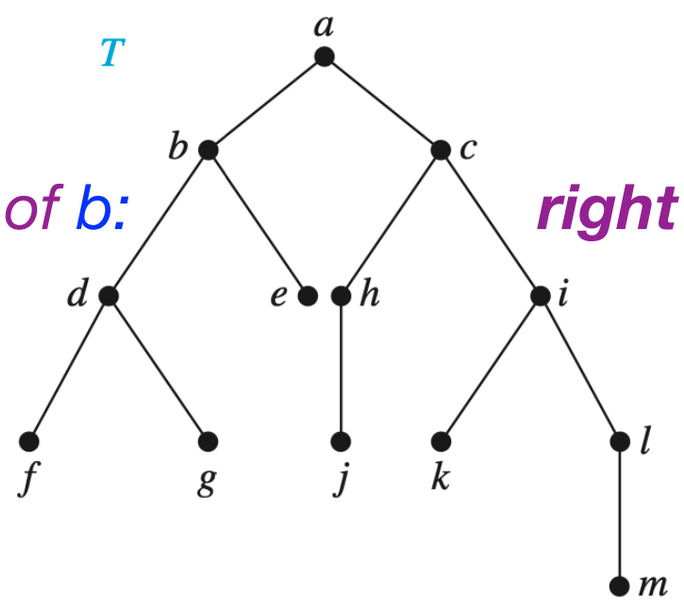
    Because $\ell \leq m^h$, we have $m^{h-1} < \ell \leq m^h$. Taking logarithms to the base m yields $h - 1 < \log_m \ell \leq h$. Hence, $h = \lceil \log_m \ell \rceil$.

Tree Traversal

Ordered Rooted Trees

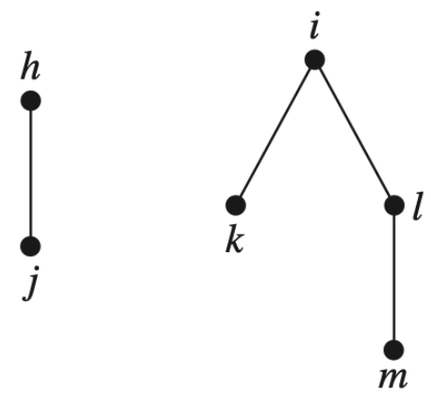
- **Definition:** An **ordered rooted tree** is a rooted tree where the children of each internal vertex are ordered.
  - In an **ordered binary tree** (usually called just a **binary tree**), if an internal vertex has two children, the first child is called the **left child** and the second child is called 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 the vertex.

T



***left child** of b:*
d

***right child** of c:*
i



***left** and **right subtrees** of c*

Traversal Algorithms

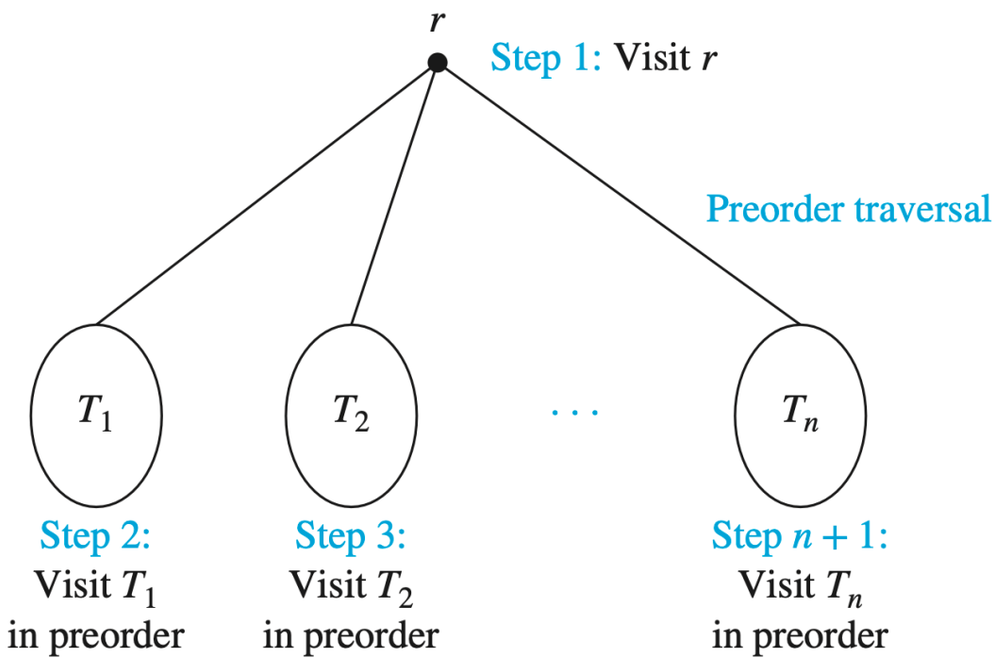
- Procedures for systematically visiting every vertex of an ordered rooted tree are called **traversal algorithms**.
- We will describe 3 of the most commonly used such algorithms, **preorder traversal**, **inorder traversal**, and **postorder traversal**.
  - Each of these algorithms can be defined recursively.
- A general principle for deciding which traversal to use is to explore the vertices of interest as soon as possible. For example:
  - A preorder traversal of the vertices of the relevant family tree produces the order of succession to the throne.
  - A postorder traversal is the best choice for deleting a tree, e.g., topological sorting.

Preorder Traversal

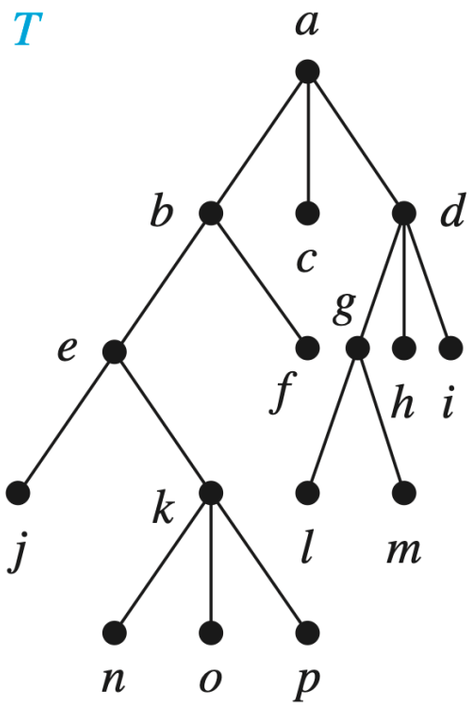
- **Definition:** Let T be an ordered rooted tree with root r. If T consists only of r, then r is the **preorder traversal** of T. Otherwise, suppose that $T_1, T_2, \ldots, T_n$ are the subtrees at r from left to right in T. The **preorder traversal** begins by visiting r. It continues by traversing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

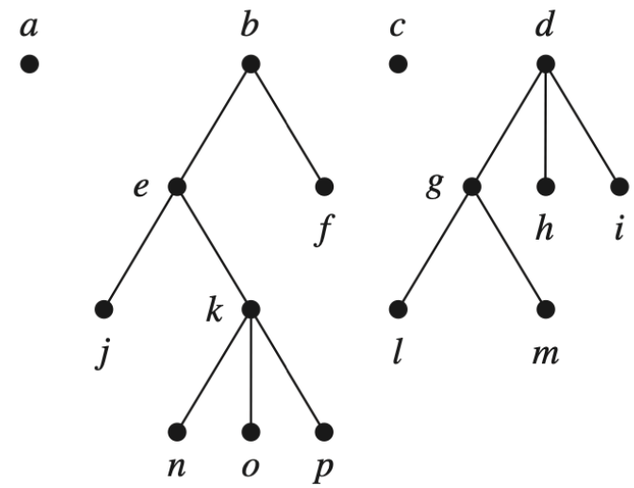
ALGORITHM 1 Preorder Traversal.

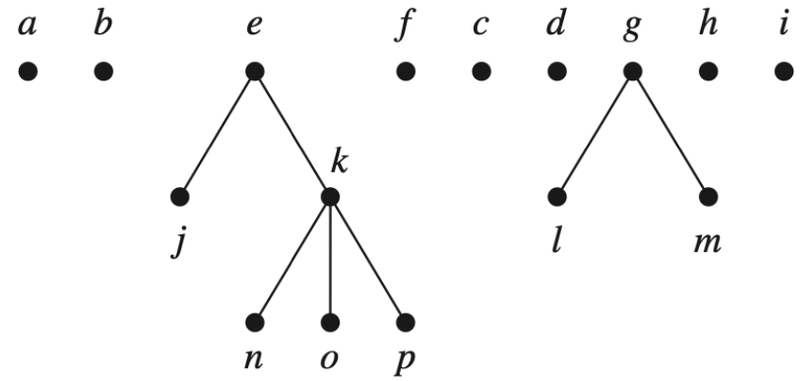
procedure *preorder*(T: ordered rooted tree)
r := root of T
list r
for each child c of r from left to right
  $T(c)$:= subtree with c as its root
  preorder($T(c)$)

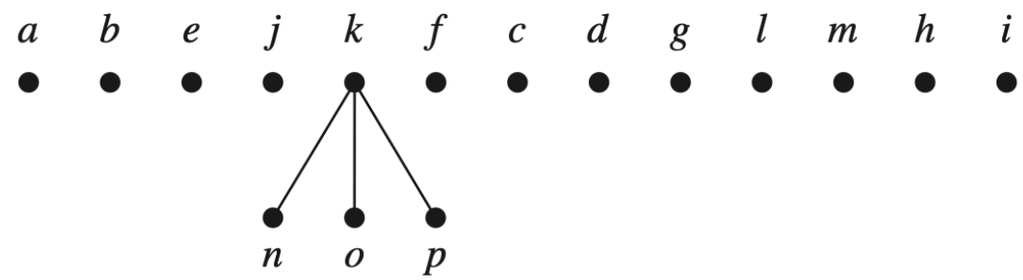


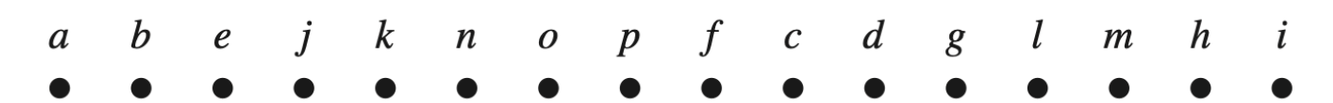
Preorder Traversal Example











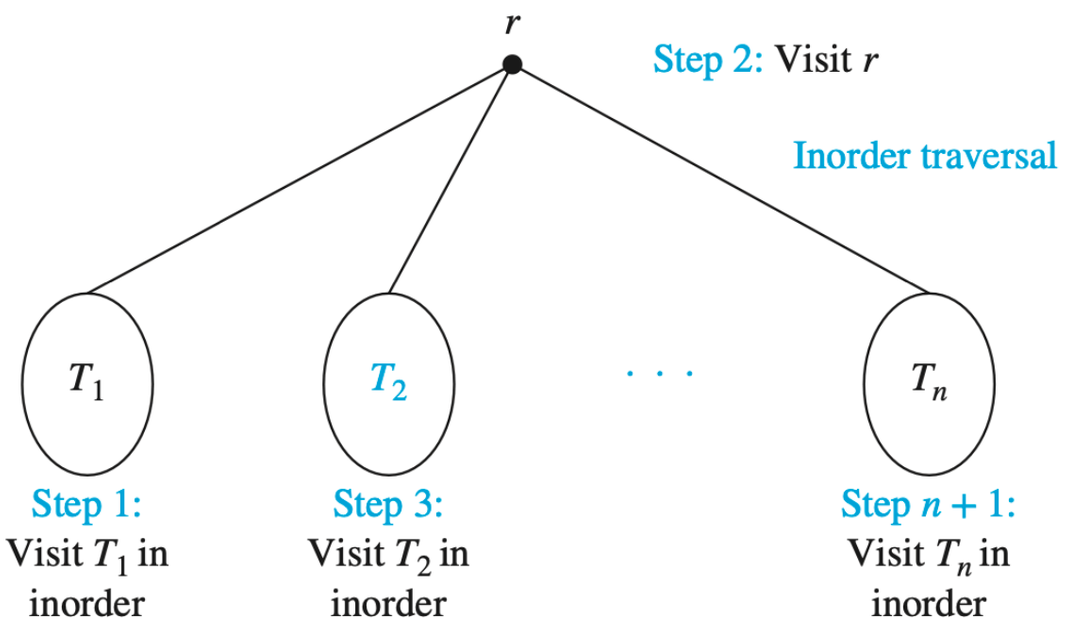
SUSTech

Inorder Traversal

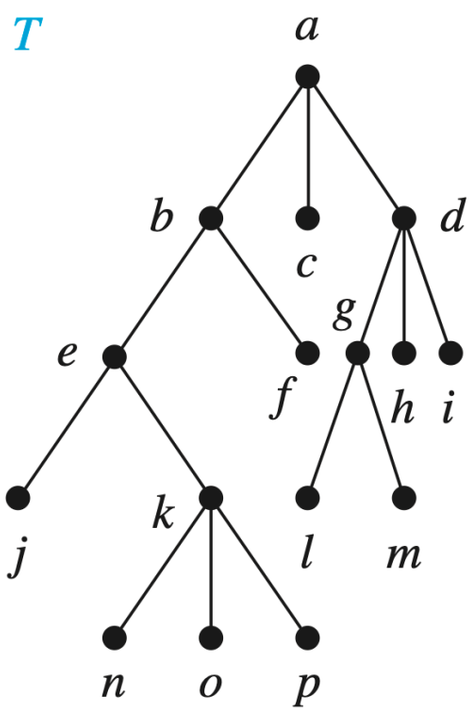
- **Definition:** Let T be an ordered rooted tree with root r. If T consists only of r, then r is the **inorder traversal** of T. Otherwise, suppose that $T_1, T_2, \ldots, T_n$ are the subtrees at r from left to right. The **inorder traversal** begins by traversing T_1 in inorder, then visiting r. It continues by traversing T_2 in inorder, then T_3 in inorder, …, and finally T_n in preorder.

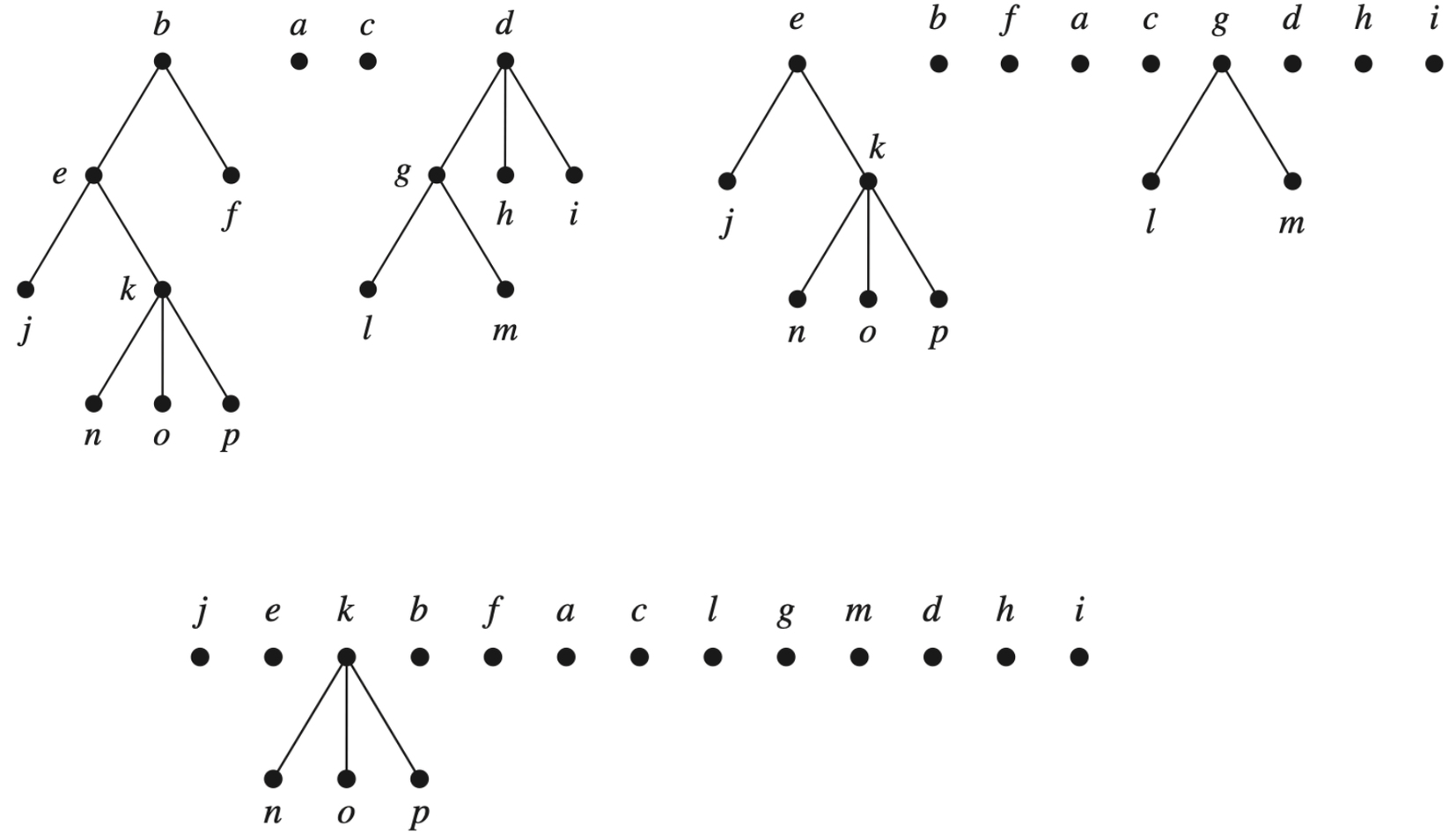
ALGORITHM 2 Inorder Traversal.

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 except for l from left to right
    $T(c)$:= subtree with c as its root
    inorder($T(c)$)



Inorder Traversal Example





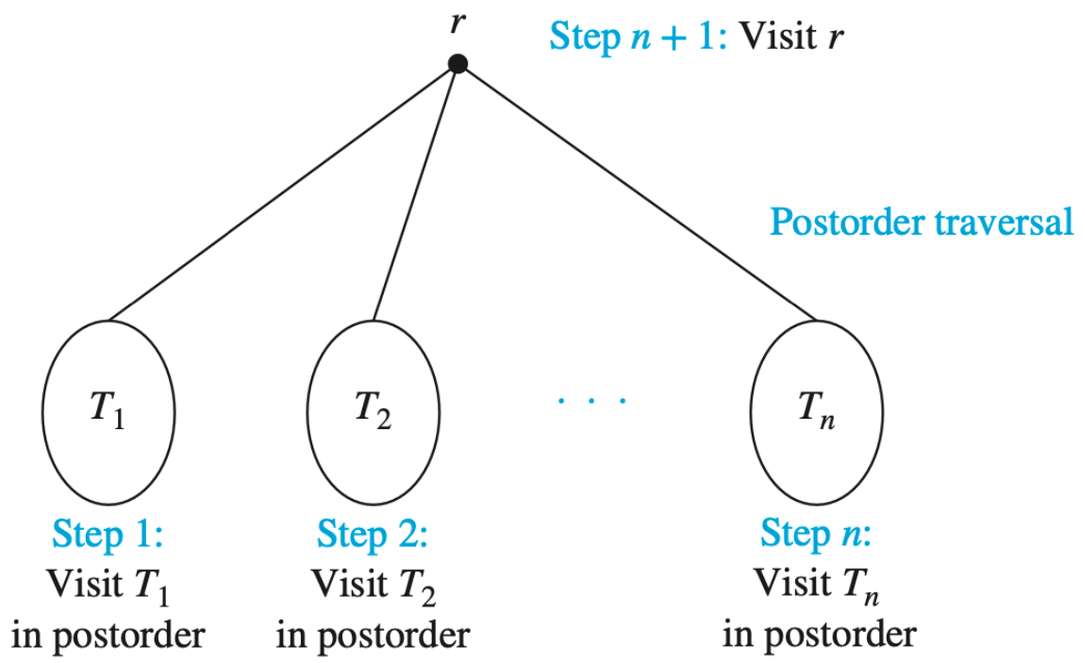
SUSTech

Postorder Traversal

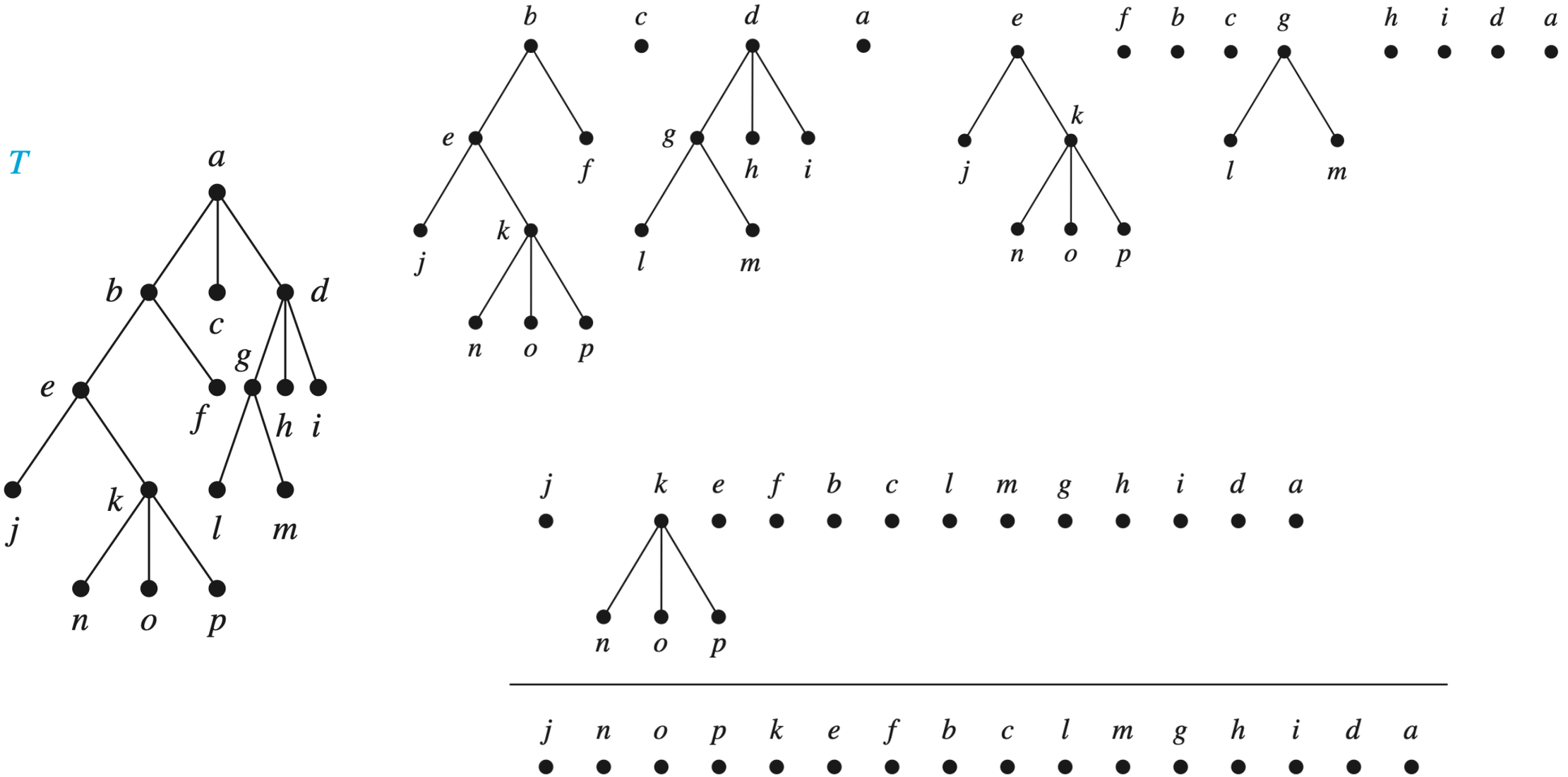
- **Definition:** Let T be an ordered rooted tree with root r. If T consists only of r, then r is the **postorder traversal** of T. Otherwise, suppose that $T_1, T_2, \ldots, T_n$ are the subtrees at r from left to right. The **postorder traversal** begins by traversing T_1 in postorder, then T_2 in postorder, …, then T_n in postorder, and ends by visiting r.

ALGORITHM 3 Postorder Traversal.

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



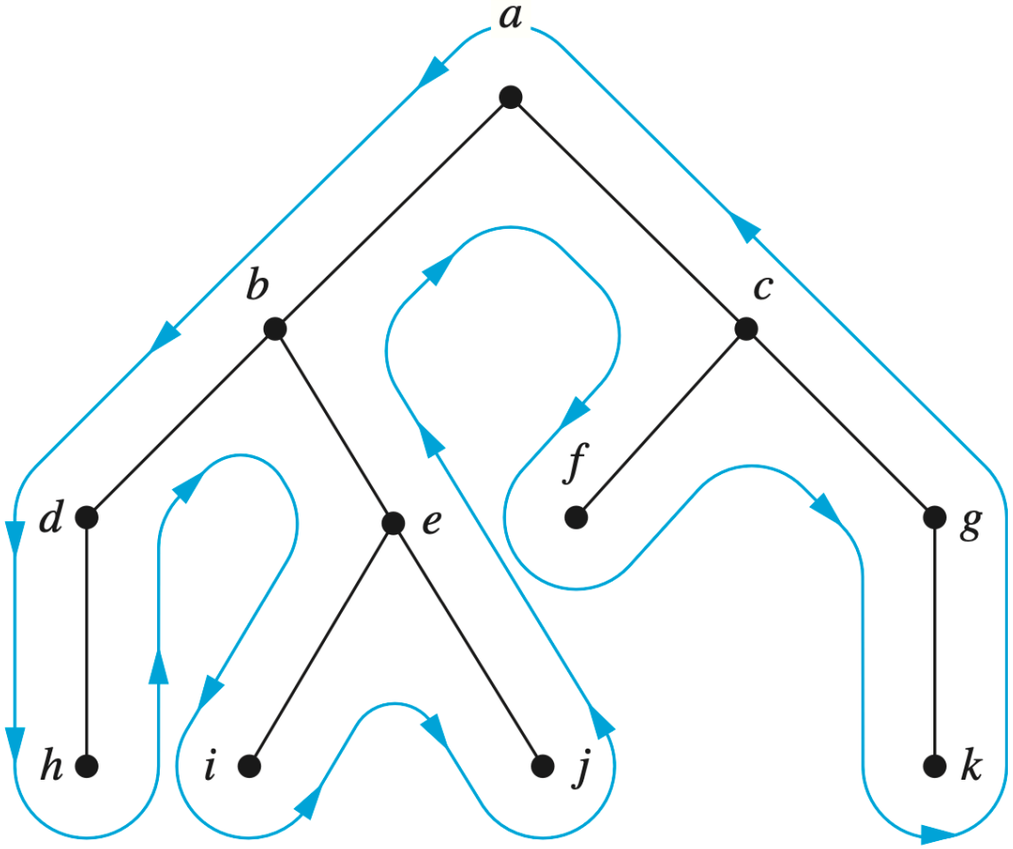
Postorder Traversal Example



SUSTech

Tree Traversal Summary

- We can list the vertices in **preorder** by listing each vertex the first time this curve passes it. We can list the vertices in **inorder** by listing a leaf the first time the curve passes it and listing each internal vertex the second time the curve passes it. We can list the vertices in **postorder** by listing a vertex the last time it is passed on the way back up to its parent.



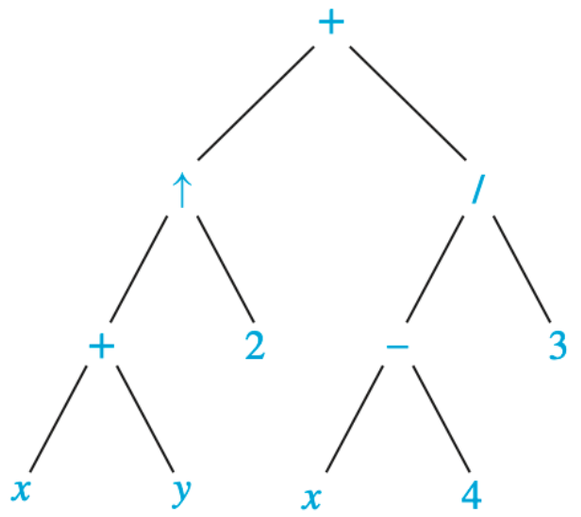
SUSTech

Binary Expression Trees

- We can represent complicated expressions, such as compound propositions, combinations of sets, and arithmetic expressions using ordered rooted trees.
  - E.g., consider the representation of an arithmetic expression involving the operators $+$, $-$, $*$, $/$, and $\uparrow$ (exponentiation).

- Such expressions are represented by a **binary expression tree**, where the internal vertices represent operations, and the leaves represent the variables or numbers. Each operation operates on its left and right subtrees (in that order), i.e., inorder traversal.

- Example: What is the binary expression tree that represents $((x + y) \uparrow 2) + ((x - 4)/3)$?



SUSTech

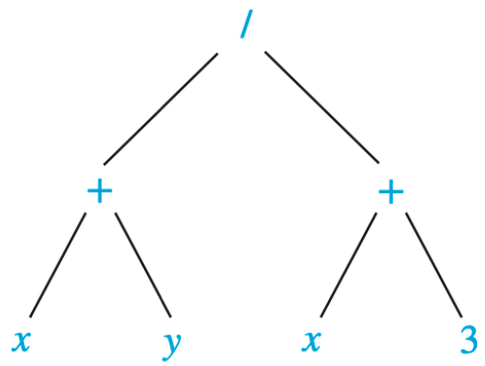
Infix Expression and Parentheses

- An inorder traversal of the binary expression tree produces the **infix expression**, which is the original expression with the elements and operations in the same order as they originally occurred when parentheses are included, except for unary operations, which instead immediately follow their operands.

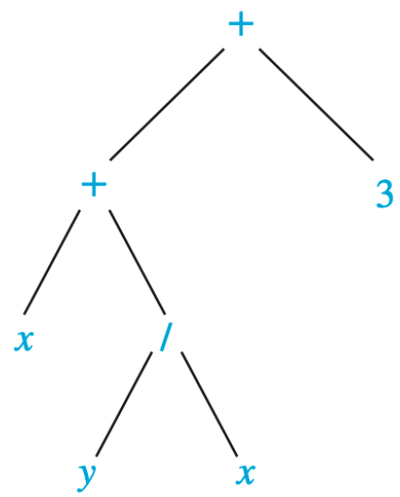
- **Q:** Why are parentheses necessary for infix expression?

- Without parentheses, the infix expression of the following binary expression trees are all the same: $x + y / x + 3$

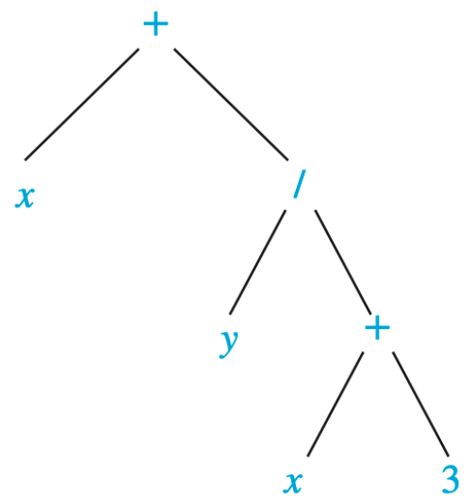
$(x + y) / (x + 3)$



$(x + (y / x)) + 3$



$x + (y / (x + 3))$

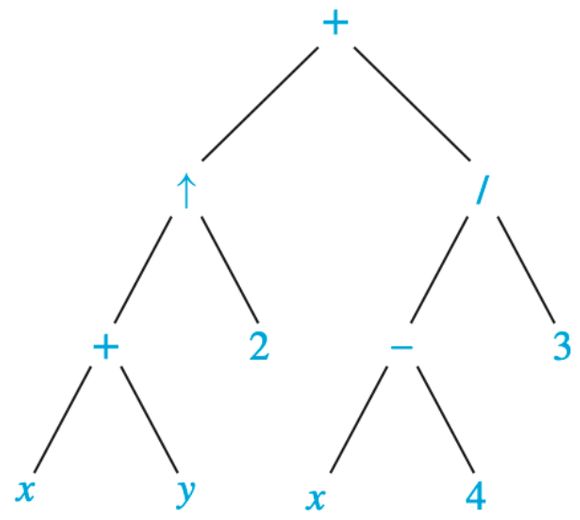


SUSTech

Prefix and Postfix Expression

- We obtain the **prefix form** of an expression when we traverse its binary expression tree in preorder.
  - Prefix expressions are said to be in **Polish notation**.
- We obtain the **postfix form** of an expression when we traverse its binary expression tree in postorder.
  - Postfix expressions are said to be in **reverse Polish notation**.
- Both prefix and postfix expressions are unambiguous, so no parentheses are needed in such expressions.
  - *the proof is left as an exercise*
- Example: What are the prefix and postfix forms of the expression *((x + y) ↑ 2) + ((x – 4) / 3)*?

  + ↑ + x y 2 / – x 4 3 *x y + 2 ↑ x 4 – 3 / +*



Prefix/Postfix Expression Evaluation

- How to evaluate a **prefix expression**?
  - Work from right to left. When an operator is encountered, perform the corresponding operation with the two operands immediately to the right of this operand.
  - After an operation is performed, consider the result a new operand.

- How to evaluate a **postfix expression**?
  - Work from left to right, carrying out operations whenever an operator follows two operands.
  - After an operation is carried out, consider the result a new operand.

Evaluation Examples

Evaluating a prefix expression:

+ − * 2 3 5 / ↑ 2 3 4

2 ↑ 3 = 8

+ − * 2 3 5 / 8 4

8 / 4 = 2

+ − * 2 3 5 2

2 * 3 = 6

+ − 6 5 2

6 − 5 = 1

+ 1 2

1 + 2 = 3

Value of expression: 3

Evaluating a postfix expression:

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

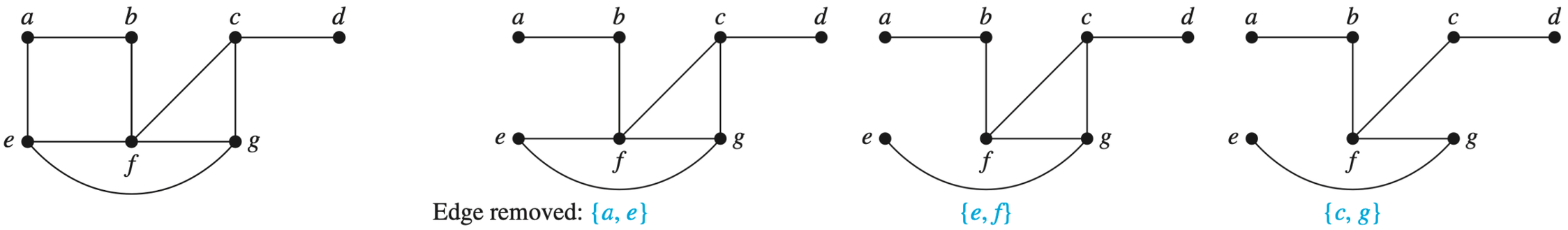
Value of expression: 4

Spanning Trees

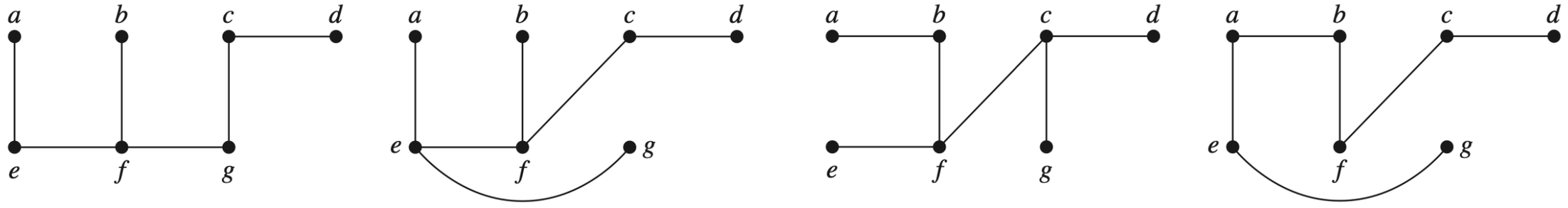
Spanning Trees

- **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.

- Example: How to find a spanning tree of a graph?



- Example: Is there only one spanning tree for the above graph?



Finding Spanning Trees

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

- Proof:
  - **"only if" part:** If a simple graph *G* is connected but not a tree, then by definition it consists of a simple circuit. The resulting subgraph has one fewer edge but still contains all the vertices of *G* and is connected. Repeat this removing edge process until no simple circuits remain. This is possible because there are only a finite number of edges in the graph.
  - **"if" part:** This is obviously true by definition of a tree.

- The above proof gives an algorithm for finding spanning trees by removing edges from simple circuits, but it is inefficient because it requires that simple circuits be identified.
  - Can we build up spanning trees by successively adding edges?

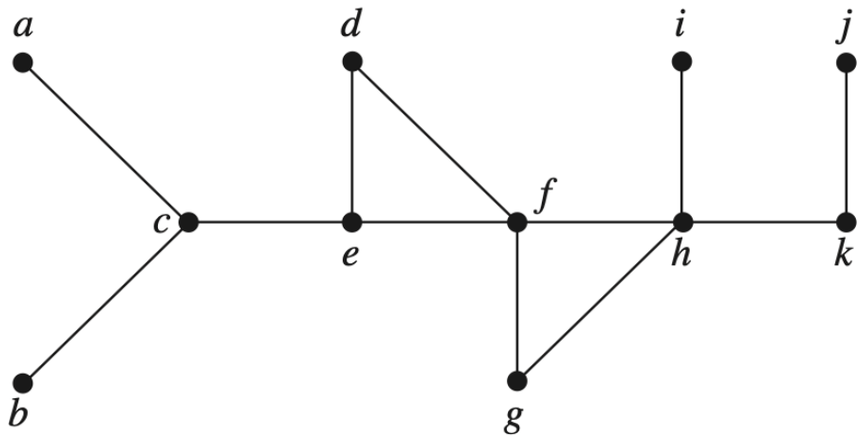
Depth-First Search

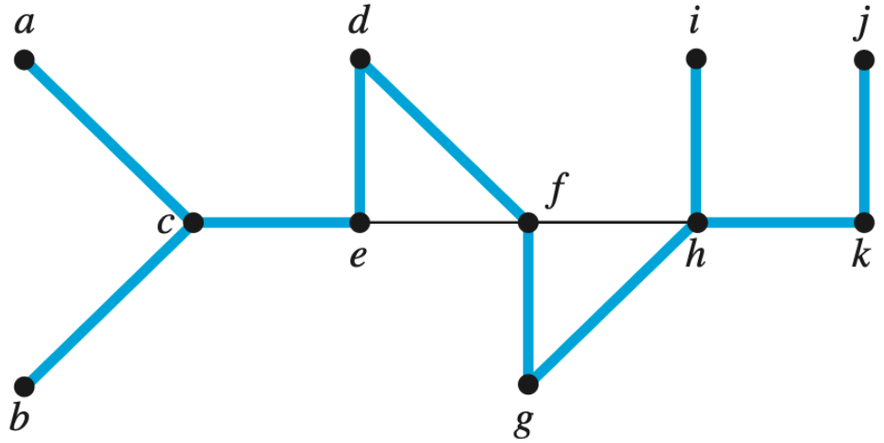
- Finding spanning trees by **depth-first search**:
  - Arbitrarily choose a vertex of the graph as the root.
  - Form a path by successively adding vertices and edges.
  - Continue adding to this path as long as possible.
  - If the path goes through all vertices of the graph, the tree consisting of this path is a spanning tree.
  - Otherwise, beginning at the last vertex visited, moving back up the path one vertex at a time, forming new paths that are as long as possible until no more edges can be added.
    ** this step reflects the **recursive** nature*

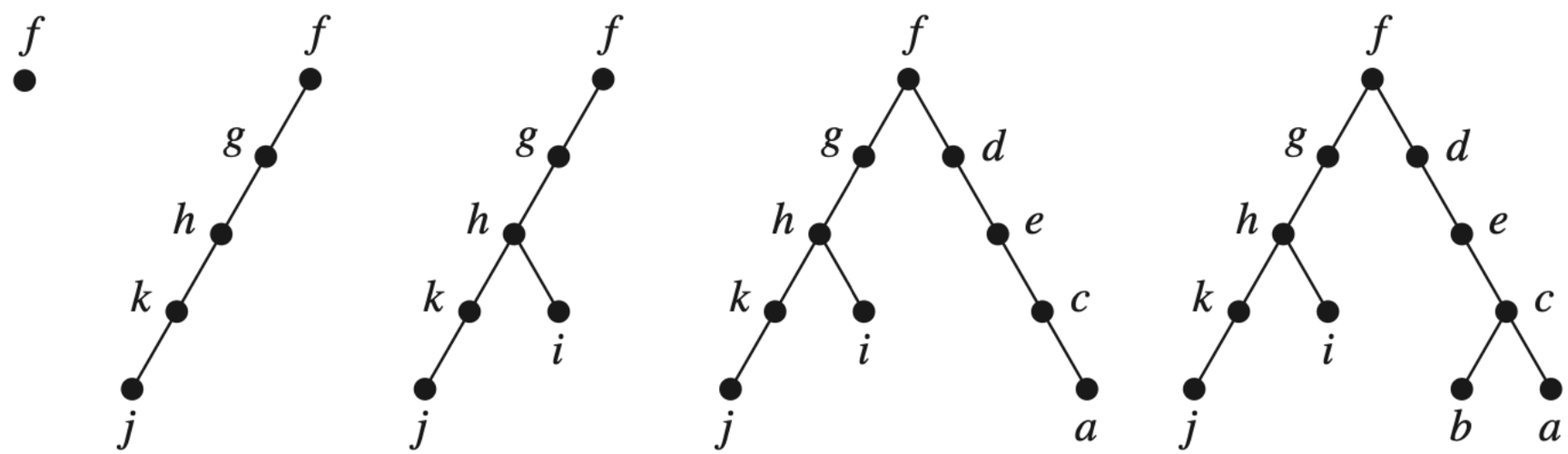
- Depth-first search is also called **backtracking**, because the algorithm returns to vertices previously visited to add paths.

Depth-First Search: Example

- Use depth-first search to find a spanning tree:







SUSTech

Depth-First Search: Algorithm

- The recursive algorithm for depth-first search (DFS):

ALGORITHM 1 Depth-First Search.

procedure *DFS*(G: connected graph with vertices $v_1, v_2, \ldots, v_n$)
$T :=$ 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)

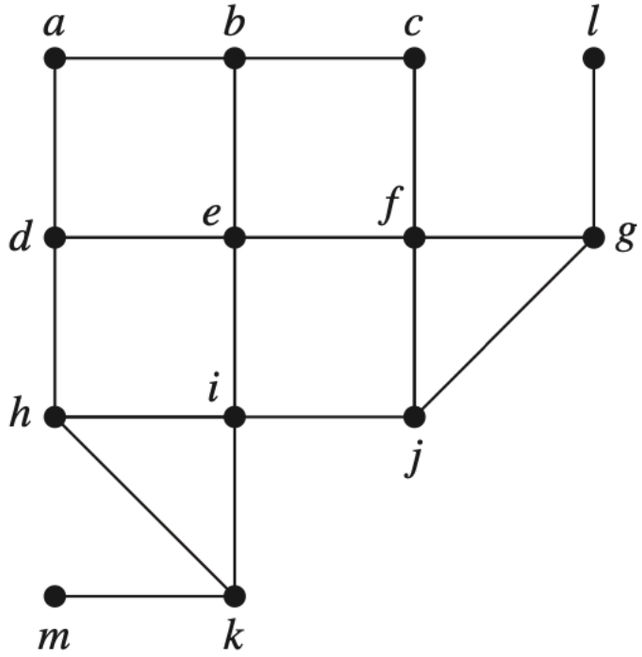
- Time complexity: $O(e)$

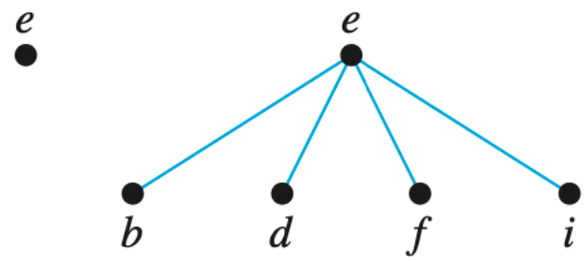
Breath-First Search

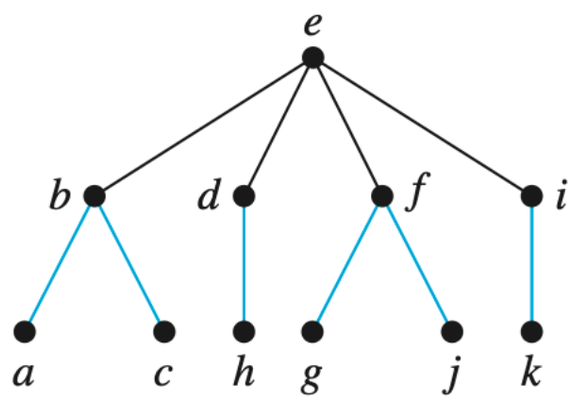
- Finding spanning trees by **breath-first search**:
  - Arbitrarily choose a vertex of the graph as the root.
  - Form a path by adding all edges incident to this vertex. The new vertices added at this stage become the vertices at this level in the spanning tree.
  - For each vertex at the previous level, visited in arbitrary order, add each edge incident to this vertex to the tree as long as it does not produce a simple circuit. The endpoints of those newly added edges, if not yet added to the tree, are also added.
  - Follow the same procedure until all the vertices have been added.

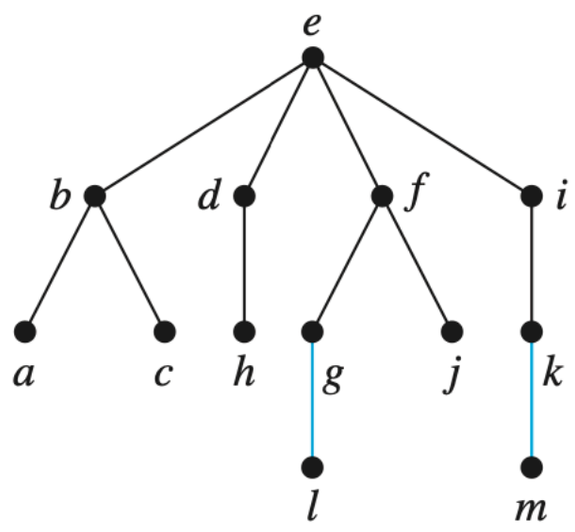
Breath-First Search: Example

- Use breath-first search to find a spanning tree:









SUSTech

Breath-First Search: Algorithm

- The algorithm for breath-first search (BFS):

ALGORITHM 2 Breadth-First Search.

procedure *BFS*(G: connected graph with vertices $v_1, v_2, \ldots, v_n$)
T := tree consisting only of vertex v_1
L := empty list
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 BFS and DFS

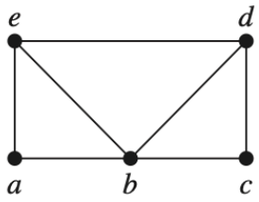
- **BFS** partitions the vertices of a graph into levels, which tell us how far away vertices are from the root, e.g., to find shortest paths, determine if graphs are bipartite, etc.

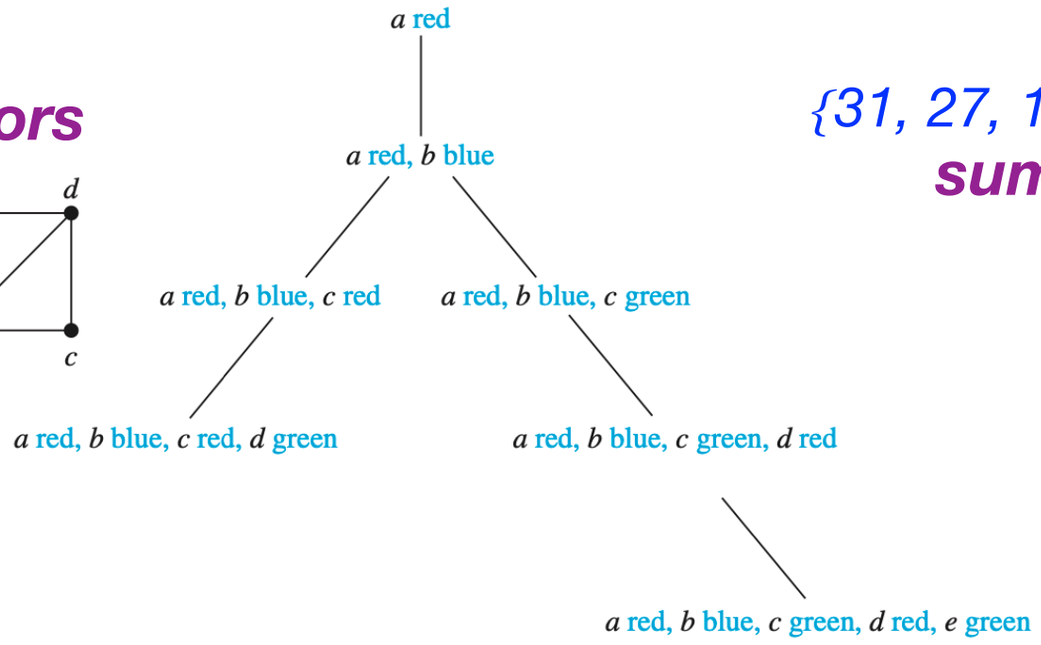
- **DFS** is a better choice when we need to explore a graph by exploring deeper rather than systematically level by level, e.g., to find paths, circuits, connected components, cut edges, etc.

- **backtracking** for an exhaustive search of all possible solutions

graph coloring

3 colors

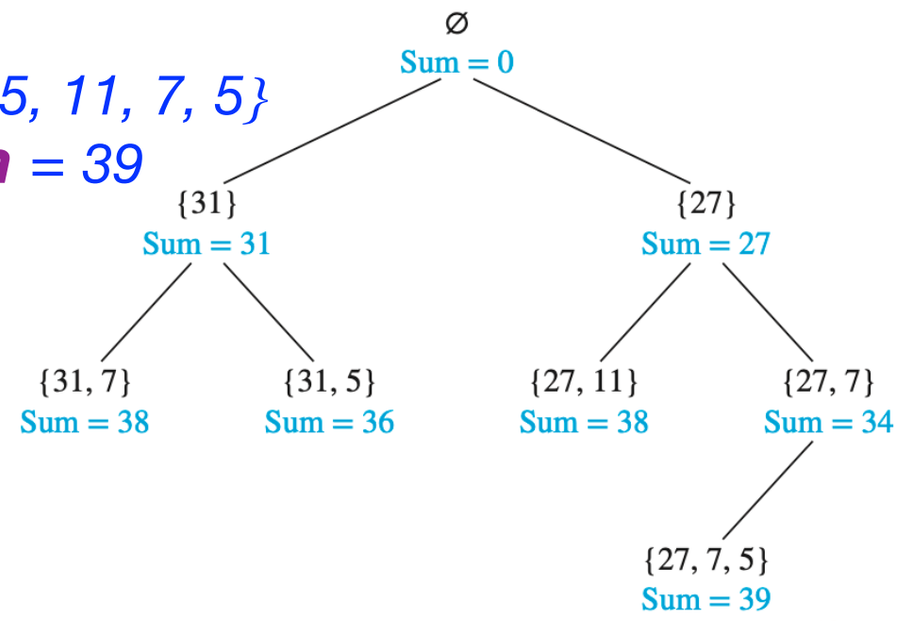




sums of subsets

{31, 27, 15, 11, 7, 5}

***sum** = 39*

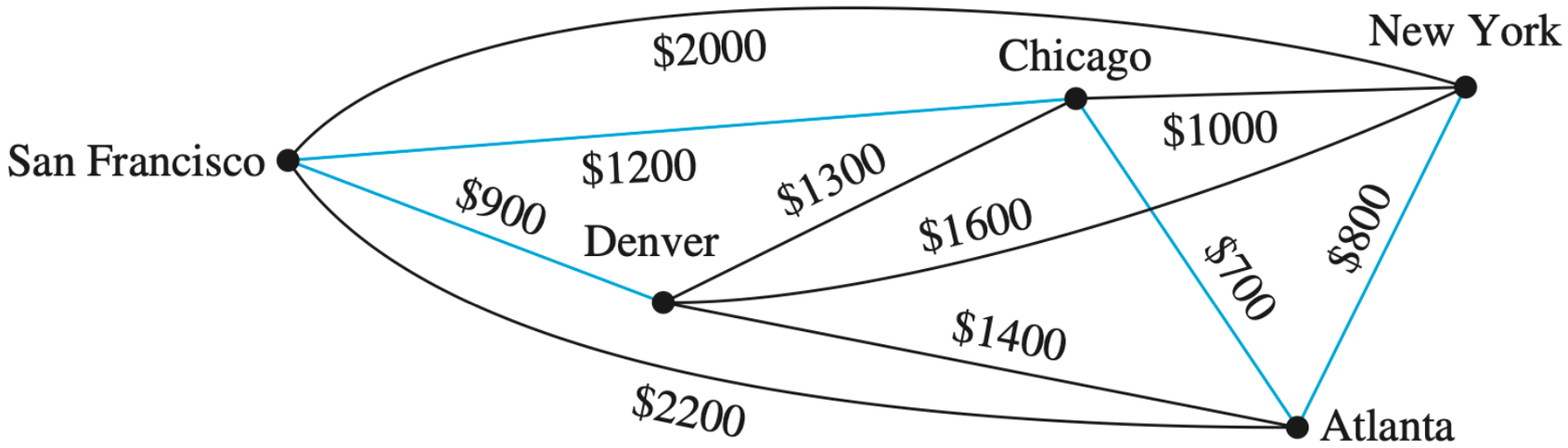


SUSTech

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.

- Example:



Choice	Edge	Cost
1	{Chicago, Atlanta}	$ 700
2	{Atlanta, New York}	$ 800
3	{Chicago, San Francisco}	$1200
4	{San Francisco, Denver}	$ 900
	Total:	$3600

Prim's Algorithm

- **Prim's algorithm** for finding a minimum spanning tree:

ALGORITHM 1 Prim's Algorithm.

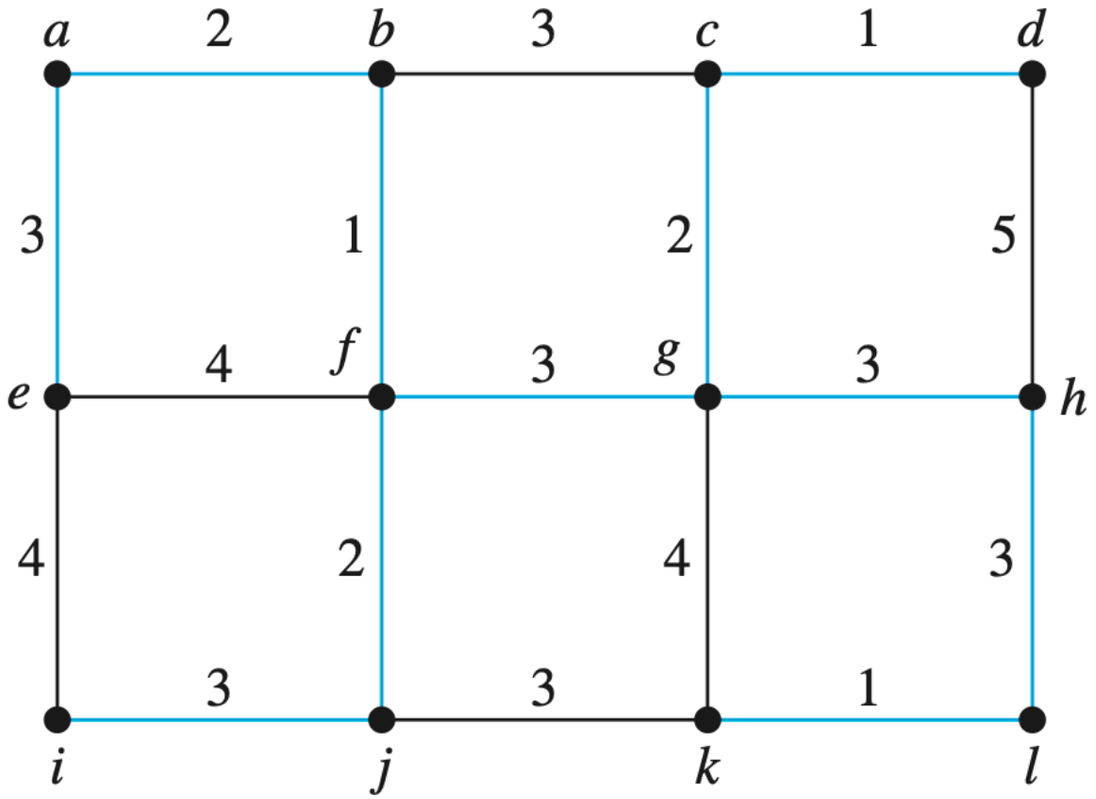
procedure *Prim*(G: weighted connected undirected graph with n vertices)
T := a minimum-weight edge
for $i := 1$ **to** $n - 2$
  e := an edge of minimum weight incident to a vertex in T and not forming a simple circuit in T if added to T
  $T := T$ with e added
return T {T is a minimum spanning tree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

start from a min-weight edge

add min-weight edge incident to T (not forming a circuit)

- Time complexity: $O(e \log v)$
  - keeping vertices not yet in T in a priority queue
  - better for dense graphs

Prim's Algorithm: Example



Choice	Edge	Weight
1	$\{b, f\}$	1
2	$\{a, b\}$	2
3	$\{f, j\}$	2
4	$\{a, e\}$	3
5	$\{i, j\}$	3
6	$\{f, g\}$	3
7	$\{c, g\}$	2
8	$\{c, d\}$	1
9	$\{g, h\}$	3
10	$\{h, l\}$	3
11	$\{k, l\}$	1
	Total:	24

Kruskal's Algorithm

- **Kruskal's algorithm** for finding a minimum spanning tree:

ALGORITHM 2 Kruskal's Algorithm.

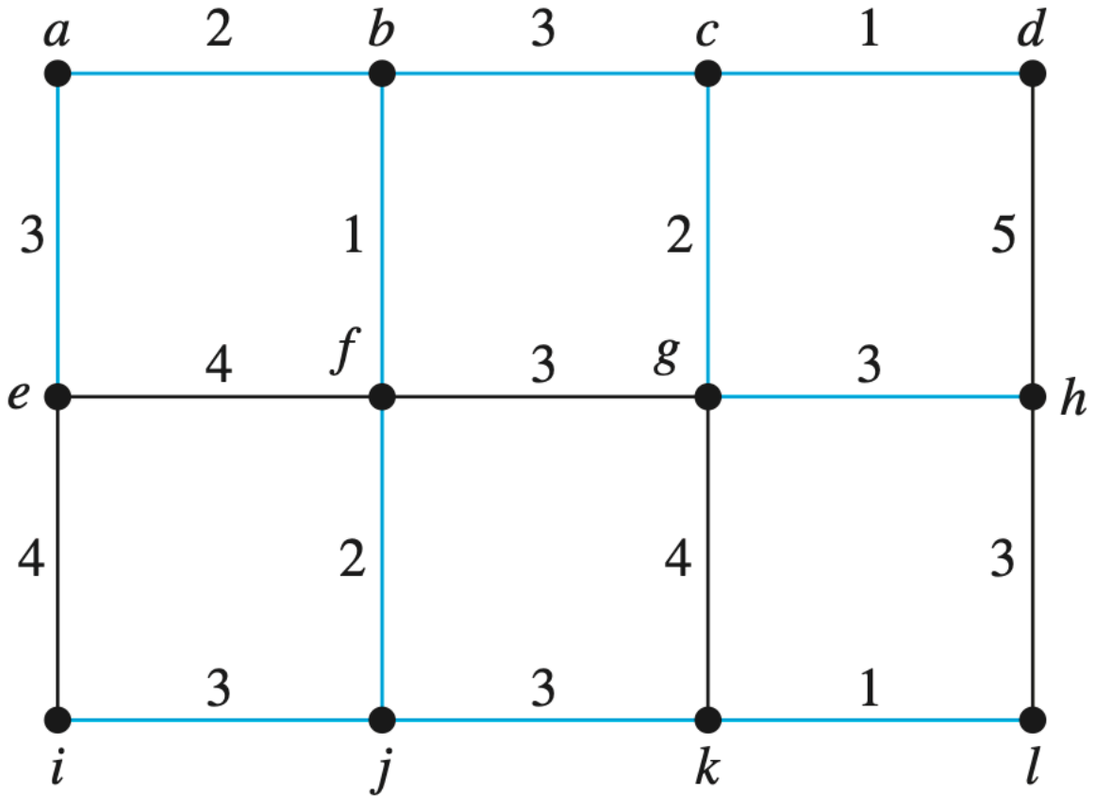
procedure *Kruskal*(G: weighted connected undirected graph with n vertices)
T := empty graph
for $i := 1$ **to** $n - 1$
  e := any edge in G with smallest weight that does not form a simple circuit when added to T
  $T := T$ with e added
return T {T is a minimum spanning tree of G}

start from an empty tree

add min-weight edge (not forming a circuit)

- Time complexity: $O(e \log e)$
  - sorting all edges + union-find data structure
  - better for sparse graphs

Kruskal's Algorithm: Example



Choice	Edge	Weight
1	$\{c, d\}$	1
2	$\{k, l\}$	1
3	$\{b, f\}$	1
4	$\{c, g\}$	2
5	$\{a, b\}$	2
6	$\{f, j\}$	2
7	$\{b, c\}$	3
8	$\{j, k\}$	3
9	$\{g, h\}$	3
10	$\{i, j\}$	3
11	$\{a, e\}$	3
	Total:	24

Announcements

- Deadline for Assignment 6: Dec 27
- Deadline for Project (optional): Dec 27
- Final exam will take place on Dec 30

- Please take some time to complete the teaching evaluation:
  - Your feedback is important to me and future students!
  - Scan this QR code for guidance: