

离散数学及其应用 第10、11章 图论

10. Graph Theory

10.1 Graphs and Graph Models

10.2 Graph Terminology and Special Types of Graphs

10.3 Representing Graphs and Graph Isomorphism

10.3.1 Adjacency Lists

10.3.2 Adjacency Matrices

10.3.3 Incidence Matrices

10.3.4 Isomorphism of Graphs

10.4 Connectivity

10.4.1 Paths

10.4.2 Connectivities in Undirected Graphs

10.4.3 Connectivities in Digraphs

10.4.4 Applications of Paths

10.5 Euler and Hamilton Paths

10.5.1 Euler Path

10.5.2 Hamilton's Paths

10.6 Shortest Path Problems

10.6.1 Basic Concepts

10.6.2 Dijkstra's Algorithm

10.6.3 The Traveling Salesperson Problem (TSP)

10.7 Planar Graphs

10.7.1 Basic Concepts

10.7.2 Euler's Formula

10.7.3 Kuratowski's Theorem

10.8 Graph Coloring

11. Trees

11.1 Introduction to Trees

11.2 Applications of Trees

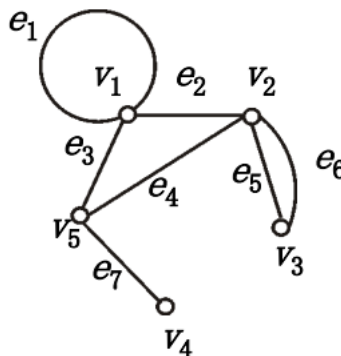
10. Graph Theory

Graphs are mathematical abstracts of relations existing in some concrete things in the objective world.

10.1 Graphs and Graph Models

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 is said to connect its endpoints.

🟡 e.g.



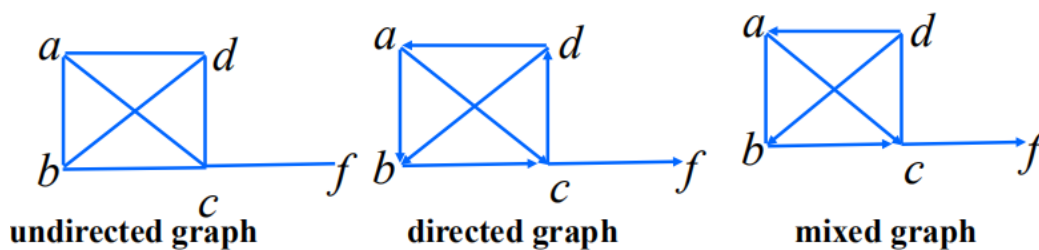
$$G = (V, E), \text{ where } V = \{v_1, v_2, v_3, v_4, v_5\},$$

$$E = \{(v_1, v_1), (v_1, v_2), (v_2, v_3), (v_2, v_3), (v_2, v_5), (v_1, v_5), (v_4, v_5)\}$$

Infinite graph : A graph with an infinite vertex set or an infinite number of edges.

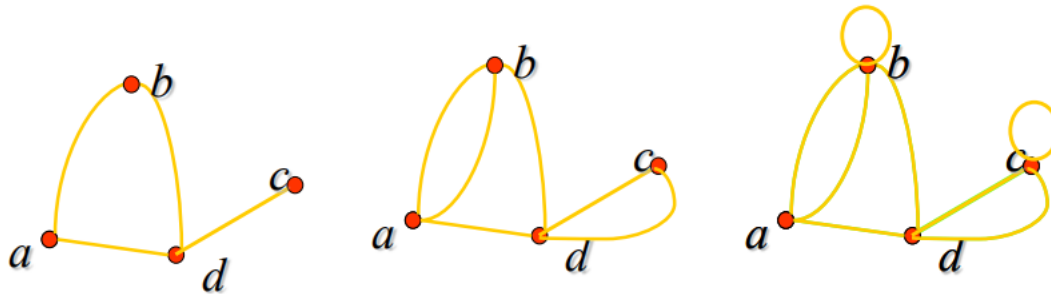
Finite graph : A graph with a finite vertex set and a finite number of edges.

💡 We shall consider only finite graph in this course.



Undirected graph : A graph with undirected edges. Undirected graph can be classified into:

- **Simple graph** : A graph in which each edge connects two different vertices and where no two edges connect the same pair of vertices.
- **Multigraph** : Graphs that may have multiple edges connecting the same vertices.
- **Pseudograph** : Graphs that may include loops, and possibly multiple edges connecting the same pair of vertices.



Directed graph : A graph with directed edges. A directed graph (or digraph) (V, E) consists of a nonempty set of vertices V and a set of directed edges (or arcs) E . Each directed 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 . Digraph can be classified into:

- **Simple directed graph** : A directed graph has no loops and has no multiple directed edges.
- **Directed multigraph** : A directed graphs that may have multiple directed edges from a vertex to a second (possibly the same) vertex.

Mixed graph : A graph with both directed and directed edges.

TABLE 1 Graph Terminology.

| Type | Edges | Multiple Edges Allowed? | Loops Allowed? |
|-----------------------|-------------------------|-------------------------|----------------|
| Simple graph | Undirected | No | No |
| Multigraph | Undirected | Yes | No |
| Pseudograph | Undirected | Yes | Yes |
| Simple directed graph | Directed | No | No |
| Directed multigraph | Directed | Yes | Yes |
| Mixed graph | Directed and undirected | Yes | Yes |

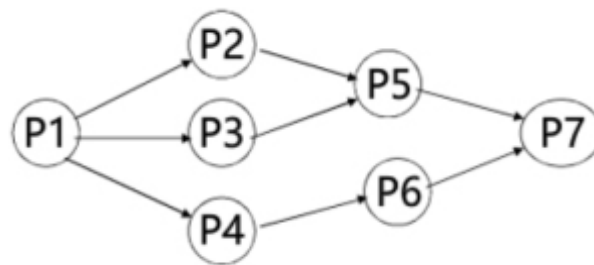
⚙ Problems in almost every conceivable discipline can be solved using **graph models** .

For example:

- Ramsey Problem: Each pair of individuals consists of two friends or two enemies, show that there are either three mutual friends or three mutual enemies in the group of

six people.

- Precedence Graph (前趋图)



Precedence graphs represent the order of completing a series of tasks using directed graphs instead of Hasse graphs.

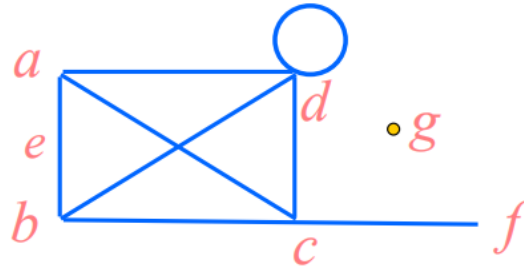
10.2 Graph Terminology and Special Types of Graphs

In Undirected Graphs $G = (V, E)$:

- Two vertices u and v in an undirected graph G are called **adjacent** (or **neighbors**) in G , if $\{u, v\}$ is an edge of G .
- An edge e connecting u and v is called **incident** with vertices u and v , or is said to **connect** u and v .
- The vertices u and v are called **endpoints** of edge $\{u, v\}$.
- An edge connects a vertex to itself is called a **loop**.
- The **neighborhood** of v , denoted as $N(v)$, is the set of all neighbors of a vertex v .
- 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.
Notation: $\deg(v)$.
- If $\deg(v) = 0$, v is called **isolated**.
- If $\deg(v) = 1$, v is called **pendant**.

🍌 e.g.

Find the degree of all the vertices.



$\deg(a) = 3$, $\deg(b) = 3$, $\deg(c) = 4$, $\deg(d) = 5$ because d is connected by a loop, $\deg(f) = 1$, $\deg(g) = 0$.

Total of degrees = $3 + 3 + 4 + 5 + 1 + 0 = 16$.

The Handshaking Theorem

Let $G = (V, E)$ be an undirected graph with e edges. Then $\sum_{v \in V} \deg(v) = 2e$.

e.g.

If a graph has 5 vertices, can each vertex have degree 3? 4?

Solution:

If each vertex have degree 3, then the sum is $3 \times 5 = 15$, which is an odd number, so it is not possible. If each vertex have degree 4, then it may be possible.

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

In Directed Graphs $G = (V, E)$:

- Let (u, v) be an edge in G . Then u is an **initial vertex** and is **adjacent to** v , and v is a **terminal vertex** and is **adjacent from** u .
- The **in degree** of a vertex v , denoted by $\deg^-(v)$, is the number of edges which terminate at v . Similarly, the **out degree** of v , denoted by $\deg^+(v)$, is the number of edges which initiate at v .
- The method of ignoring the directions in a digraph to find a path is called **underlying undirected graph** .

Let $G = (V, E)$ be a graph with direct edges (not necessarily a digraph), then

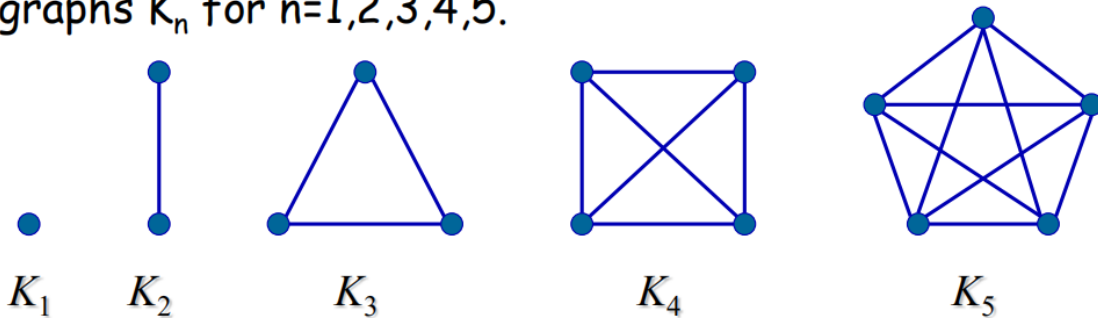
$$\sum_{v \in V} \deg^-(v) = \sum_{v \in V} \deg^+(v) . \text{ If it is a directed graph, then}$$

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

Some Special Simple Graphs

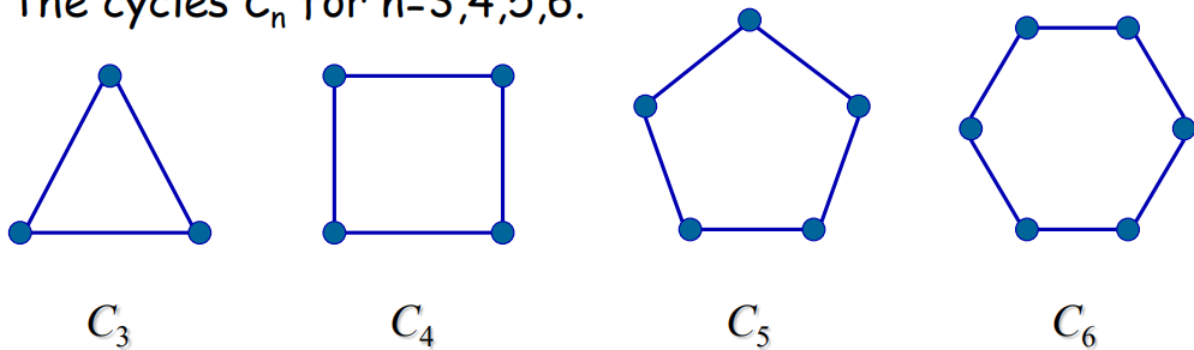
- Complete Graphs -- K_n : the simple graph with n vertices and exactly one edge between every pair of distinct vertices.

The graphs K_n for $n=1,2,3,4,5$.



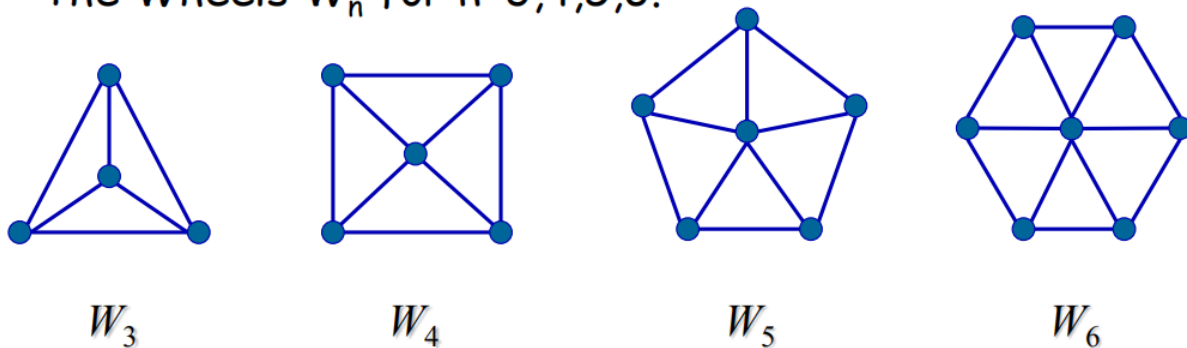
- Cycles -- $C_n (n > 2)$: $C_n = (V, E)$, where $V = \{v_1, v_2, \dots, v_n\}$,
 $E = \{(v_1, v_2), (v_2, v_3), \dots, (v_{n-1}, v_n), (v_n, v_1)\}$, $n \geq 3$.

The cycles C_n for $n=3,4,5,6$.



- Wheels $W_n (n > 2)$: Add one additional vertex to the cycle C_n and add an edge from each vertex to the new vertex to produce W_n .

• The Wheels W_n for $n=3,4,5,6$.



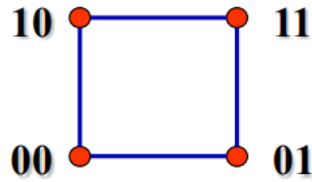
⚠ Note that W_n has $n + 1$ vertices.

- n -Cubes $Q_n (n > 0)$: $Q_n = (V, E)$ is the graph with 2^n vertices representing bit strings of length n , where $V = \{v | v = a_1 a_2 \dots a_n, a_i = 0, 1, i = 1, 2, \dots, n\}$,
 $E = \{(u, v) | u, v \in V \wedge u \text{ and } v \text{ differ by one bit position}\}$.

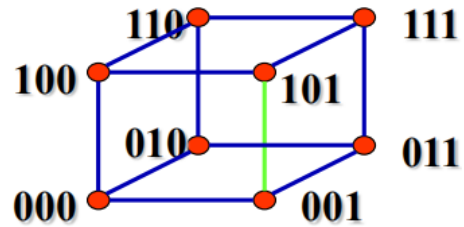
• The n-Cubes Q_n for $n=1,2,3$



Q_1



Q_2

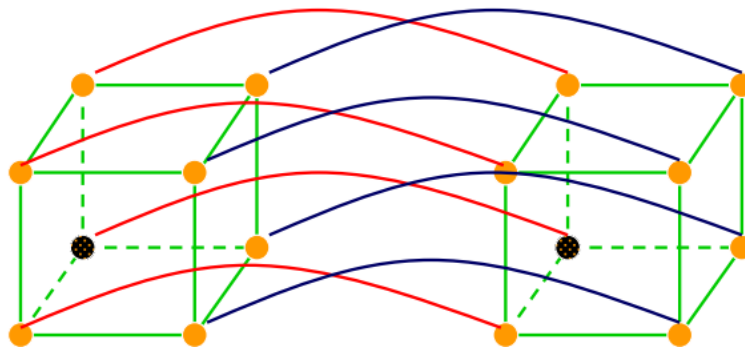


Q_3

$Q_4?$

Construct Q_{n+1} from Q_n

1. Making two copies of Q_n , prefacing the labels on the vertices with a 0 in one copy and with a 1 in the other copy.
2. Adding edges connecting two vertices that have labels differing only in the first bit.



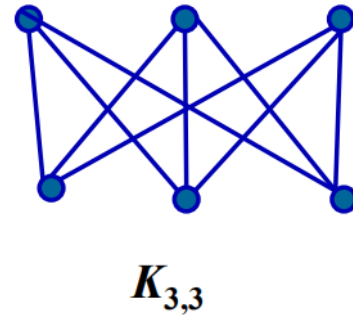
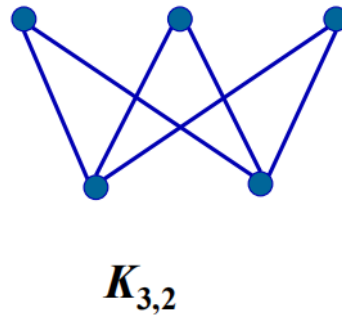
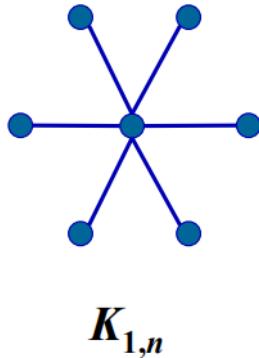
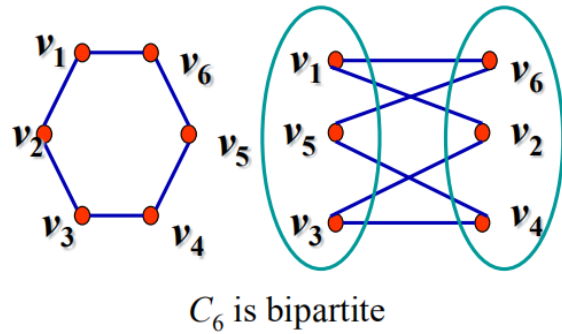
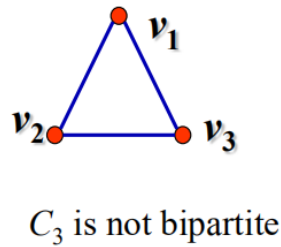
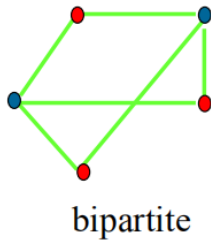
Construct Q_5 using 2 copies of Q_4

A simple graph G is **bipartite** (偶图) if V can be partitioned into two disjoint subsets V_1 and V_2 such that every edge connects a vertex in V_1 and a vertex in V_2 . The pair $\{V_1, V_2\}$ is called a **bipartition** of the vertex V of G . The **complete bipartite graph** is the simple graph that has its vertex set partitioned into two subsets V_1 and V_2 with m and n vertices, respectively, and every vertex in V_1 is connected to every vertex in V_2 , denoted by $K_{m,n}$, where $m = |V_1|$ and $n = |V_2|$.

A simple graph is bipartite if and only if it is possible to assign one of two different colors to each vertex of the graph so that no two adjacent vertices are assigned the same color.

💡 如果简单图中存在一个由奇数个结点构成的回路，那么这个简单图不是偶图，因为必然会有两个相邻的结点被涂成相同的颜色。→ 奇数个齿轮啮合成一圈形成的结构不能转动，因为必然会有两个相邻的齿轮需要同向转动。

For example,

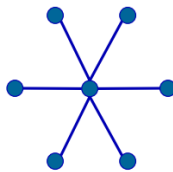


A simply graph is called **regular** if every vertex of this graph has the same degree. A regular graph is called **n-regular** if every vertex in this graph has degree n .

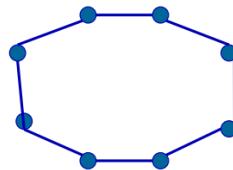
e.g.

K_n is a $(n-1)$ -regular.

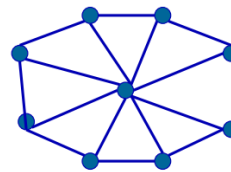
Special types of graphs can be used to represent local area networks.



Star topology



Ring topology



Hybrid topology

$G = (V, E)$, $H = (W, F)$. H is a **subgraph** of G if $W \subseteq V$ and $F \subseteq E$. Subgraph H is a **proper subgraph** of G if $H \neq G$. H is a **spanning subgraph** of G if $W = V$, $F \subseteq E$.

e.g.

How many subgraphs with at least one vertex does W_3 have?

Solution:

Choose 1 to 4 vertices from the wheel each time, and decide whether there are vertices between each pair of the vertices.

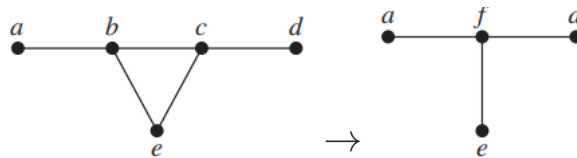
$$C(4, 1) + C(4, 2) \times 2 + C(4, 3) \times 2^3 + C(4, 4) \times 2^6 .$$

⚠ Note that here each vertex is considered to be different from others.

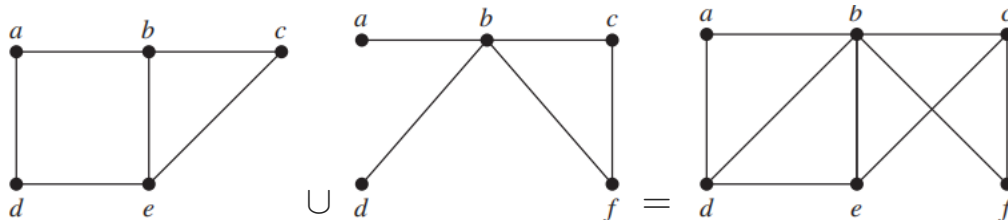
Let $G = (V, E)$ be a simple graph. The subgraph **induced** by a subset W of the vertex set V is the graph (W, F) , where the edge set F contains an edge in E iff both endpoints of this edge are in W .

📁 Graph Operation

- Removing edges of a graph: $G - e = (V, E - \{e\})$
- Adding edges to a graph: $G + e = (V, E \cup \{e\})$
- **Edge contraction** (边收缩) : Remove an edge e with endpoints u and v , merge u and v into a new single vertex w , and for each edge with u or v as an endpoint, replace the edge with one with w as endpoint in place of u and v and with the same second endpoint.



- Removing vertices from a graph: $G - v = (V - v, E')$, where E' is the set of edges of G not incident to v .
- 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 $V = V_1 \cup V_2$ and edge set $E = E_1 \cup E_2$, denoted as $G_1 \cup G_2 = (V_1 \cup V_2, E_1 \cup E_2)$.

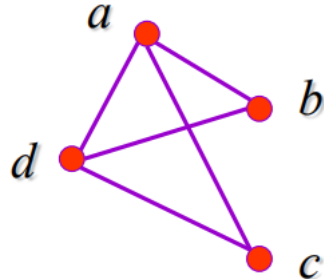


10.3 Representing Graphs and Graph Isomorphism

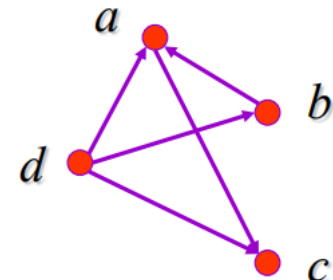
10.3.1 Adjacency Lists

Adjacency lists (邻接表) are lists that specify the vertices that are adjacent to each vertex.

For simple graphs:

|  | Vertices | Adjacent Vertices |
|---|----------|-------------------|
| | a | b, c, d |
| | b | a, d |
| | c | a, d |
| | d | a, b, c |

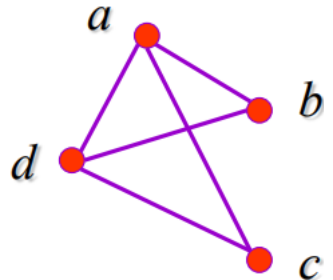
For digraphs:

|  | Initial Vertex | Terminal Vertices |
|--|----------------|-------------------|
| | a | c |
| | b | a |
| | d | a, b, c |

10.3.2 Adjacency Matrices

A simple graph $G = (V, E)$ with n vertices (v_1, v_2, \dots, v_n) can be represented by its **adjacency matrix** (邻接矩阵), A , where $a_{ij} = \begin{cases} 1, & \text{if } (v_i, v_j) \text{ is an edge of } G \\ 0, & \text{otherwise} \end{cases}$.

⚠ Note that an adjacency matrix of a graph is based on the ordering chosen for the vertices.

|  | Adjacency Matrix | | | | |
|---|------------------|---|---|---|---|
| | | a | b | c | d |
| | a | | | | |
| | b | | | | |

| | |
|---|--|
| c | $\begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix}$ |
| d | |

To represent a multigraph or pseudograph using adjacency matrixes, just let the (i, j) -th entry of such a matrix equals the number of edges that are associated to $\{v_i, v_j\}$.

⚠ Note that a loop only contribute 1 to the corresponding entry.

| Adjacency Matrix | | | | |
|------------------|--|---|---|---|
| | a | b | c | d |
| a | $\begin{bmatrix} 0 & 1 & 1 & 2 \\ 1 & 1 & 0 & 1 \\ 1 & 0 & 0 & 3 \\ 2 & 1 & 3 & 0 \end{bmatrix}$ | | | |
| b | | | | |
| c | | | | |
| d | | | | |

📦 Adjacency matrices of undirected graphs are always symmetric.

For directed graph $G = (V, E)$ with $|V| = n$, suppose that the vertices of G are listed in arbitrary order as v_1, v_2, \dots, v_n , the adjacency matrix $A = [a_{ij}]$, where

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

| Adjacency Matrix | | | | |
|------------------|--|---|---|---|
| | a | b | c | d |
| a | $\begin{bmatrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 \end{bmatrix}$ | | | |
| b | | | | |
| c | | | | |
| d | | | | |

📖 The sum of the entries in a row / column of the adjacency matrix for an undirected graph is the number of edges incident to the vertex i , which is the same as $\deg(i)$ minus the number of loops at i .

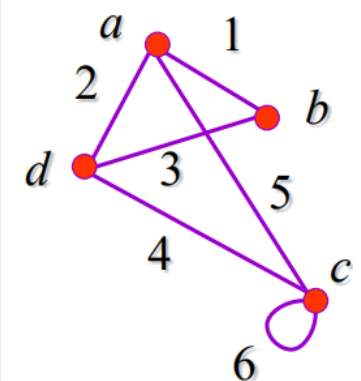
📖 The sum of the entries in a row of the adjacency matrix for an digraph is $\deg^+(v_i)$. In a column it is $\deg^-(v_i)$.

💡 计算机中，考虑到占用空间大小，邻接表比较适合表示稀疏矩阵，而邻接矩阵比较适合表示稠密矩阵。

10.3.3 Incidence Matrices

$G = (V, E)$, $V = \{v_1, v_2, \dots, v_n\}$, $E = \{e_1, e_2, \dots, e_m\}$. Then the **incidence matrix** with respect to this ordering of V and E is $n \times m$ matrix $M = [m_{ij}]_{n \times m}$,

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

| | | Incidence Matrix | | | | | |
|--|---|------------------|---|---|---|---|---|
| | | 1 | 2 | 3 | 4 | 5 | 6 |
|  | a | 1 | 1 | 0 | 0 | 1 | 0 |
| | b | 1 | 0 | 1 | 0 | 0 | 0 |
| | c | 0 | 0 | 0 | 1 | 1 | 1 |
| | d | 0 | 1 | 1 | 1 | 0 | 0 |

📖 Incidence matrices of undirected graphs contain two 1s per column for edges connecting two vertices, and one 1 per column for loops.

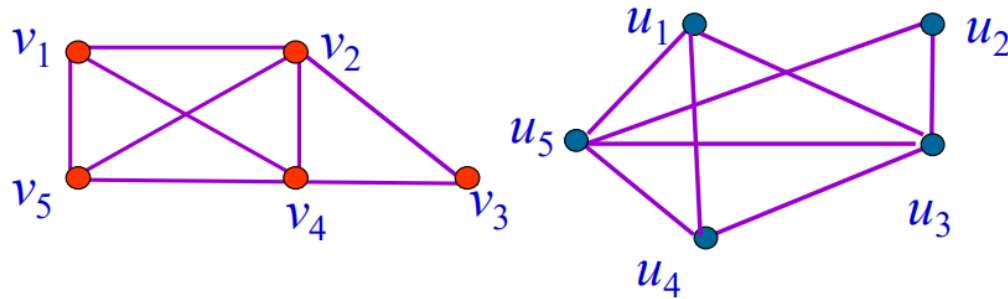
10.3.4 Isomorphism of Graphs

Two simple graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ are **isomorphic** (同构) if there is a one-to-one correspondence function f (f is called an **isomorphism**) from V_1 to V_2 such that for all a and b in V_1 , a and b are adjacent in G_1 iff $f(a)$ and $f(b)$ are adjacent in G_2 .

In other words, when two simple graphs are isomorphic, there is a one-to-one correspondence between vertices of the two graphs that preserves the adjacency relationship.

● e.g.

Show that the following two graphs are isomorphic.



Proof:

1. Try to find an isomorphism f .
2. Show that f preserves adjacency relation– The adjacency matrix of a graph G is the same as the adjacency matrix of another graph H , when rows and columns are labeled to correspond to the images under f of the vertices in G .

$$\text{Here } A_1 = \begin{matrix} & v_1 & v_2 & v_3 & v_4 & v_5 \\ \begin{matrix} v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \end{matrix} & \begin{bmatrix} 0 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 & 0 \end{bmatrix} \end{matrix} \text{ and } A_2 = \begin{matrix} & u_4 & u_5 & u_2 & u_3 & u_1 \\ \begin{matrix} u_4 \\ u_5 \\ u_2 \\ u_3 \\ u_1 \end{matrix} & \begin{bmatrix} 0 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 & 0 \end{bmatrix} \end{matrix} \text{ are the same.}$$

So f where $f(v_1) = u_4$, $f(v_2) = u_5$, $f(v_3) = u_2$, $f(v_4) = u_3$, $f(v_5) = u_1$ is a proper isomorphism.

It is usually difficult to find an isomorphism f since there are $n!$ possible 1-1 correspondence between the two vertex sets with n vertices.

However, there are some properties called **invariants** (不变量) in the graphs that may be used to show that they are not isomorphic. Graph invariants are properties preserved by isomorphism of graphs.

📁 Important Invariants in Isomorphic Graphs:

- the number of vertices
- the number of edges
- the degrees of corresponding vertices

- if one is bipartite, the other must be.
- if one is complete, the other must be.
- if one is a wheel, the other must be.

etc.

🍎 e.g.

Draw all nonisomorphic undirected simple graph with four vertices and three edges.

Solution:

By the handshaking theorem, the sum of the degrees over four vertices is 6. The maximal degree is 3, and the number of vertices with odd degrees must even. So there are 3 possible sequence of degrees:

(1) 1, 1, 2, 2; (2) 1, 1, 1, 3; (3) 0, 2, 2, 2.



10.4 Connectivity

10.4.1 Paths

A **path** of **length** n from u to v in an undirected graph is a sequence of n edges e_1, e_2, \dots, e_n for which there exists a sequence $x_0 = u, x_1, \dots, x_{n-1}, x_n = v$ such that e_i has endpoints x_{i-1}, x_i . When the graph is simple, we denote this path by its vertex sequence $x_0, x_1, \dots, x_{n-1}, x_n$. If the path begins and ends with the same vertex, we call it a **circuit**. The path or circuit is said to **pass through** the vertices x_1, \dots, x_{n-1} or **traverse** the edges e_1, \dots, e_n . If a path or a circuit does not contain the same edge more than once, we call it a **simple** path / circuit.

A **path** of length n from u to v in a directed graph is a sequence of edges e_1, e_2, \dots, e_n such that e_1 is associated with (x_0, x_1) , \dots , e_i is associated with (x_{i-1}, x_i) , \dots , e_n is associated with (x_{n-1}, x_n) . When there are no multiple edges in the directed graph, this path is denote by its vertex sequence $x_0, x_1, \dots, x_{n-1}, x_n$. The definitions of circuit and simple path / circuit still holds in directed graphs.

An undirected graph is **connected** if there is a path between every pair of distinct vertices, and is **disconnected** if the graph is not connected. To **disconnect** a graph means to remove vertices or edges, or both, to produce a disconnected subgraph.

⚠ K_1 is considered to be connected.

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

Proof:

Since the graph is connected, there is a path between u and v . Throw out all redundant circuits to make the path simple.

The maximally connected subgraphs of G are called the **connected components** or simply "components".

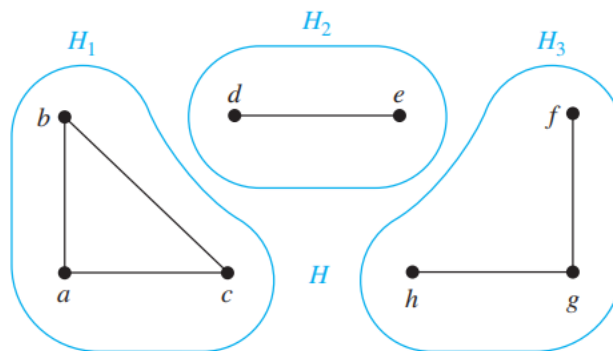


FIGURE 3 The graph H and its connected components H_1 , H_2 , and H_3 .

If removing a vertex and all edges incident with it results in more connected components than in the original graph, then it is called a **cut vertex** or an **articulation point**.

If removing an edge creates more components, then it is called a **cut edge** or a **bridge**. Connected graphs without cut vertices are called **nonseparable graphs**.

Nonseparable graphs can be thought of as more connected than those with a cut vertex. We can measure graph connectivity based on the minimum number of vertices removed to disconnect a graph.

10.4.2 Connectivities in Undirected Graphs

Here G is an undirected graph.

A **vertex cut**, or **separating set**, is a subset V' of the vertex set V of $G = (V, E)$ such that $G - V'$ is disconnected or K_1 .

📖 Every connected graph except a complete graph has a vertex cut.

The **vertex connectivity** (点连通度) of G , denote as $\kappa(G)$, is the minimum number of vertices in a vertex cut, or equivalently, the minimum number of vertices that can be removed from G to either disconnect G or produce a graph with a single vertex. The larger $\kappa(G)$ is, the more connected we consider G to be. A graph is said to be **k-connected** (or k-vertex-connected), if $\kappa(G) \geq K$.

κ is the 10th letter in Greek alphabet, called "kappa".

📖 Properties of Vertex Connectivities

- If G has n vertices, then $0 \leq \kappa(G) \leq n - 1$.
 - $\kappa(G) = 0$ iff G is disconnected or $G = K_1$.
 - $\kappa(G) = 1$ if G is connected with cut vertices or $G = K_2$.
 - $\kappa(G) = n - 1$ iff $G = K_n$.

We can also measure graph connectivity based on the minimum number of edges removed to disconnect a graph.

A set of edges E' is called an **edge cut** of G if the subgraph $G - E'$ is disconnected.

The **edge connectivity** (边连通度) of G , denoted as $\lambda(G)$, is the minimum number of edges in an edge cut of G , or equivalently, the minimum number of edges removed from G to disconnect G .

📖 Properties of Edge Connectivities

- If G has n vertices, then $0 \leq \lambda(G) \leq n - 1$.
 - $\lambda(G) = 0$ if G is disconnected or $G = K_1$.
 - $\lambda(G) = n - 1$ iff $G = K_n$.

📖 Whitney's Inequality (惠特尼不等式)

When $G = (V, E)$ is a noncomplete connected graph with at least three vertices,
$$\kappa(G) \leq \lambda(G) \leq \min_{v \in V} \deg(v).$$

💡 去掉结点的时候会把连接到这个点的边一起去掉。去点的同时也在去边，所以去点对连通度的影响比单纯去边的影响大，所以点连通度比边连通度小。不需要会证，记得有这么个结论就行。

⚙️ Vertex and edge connectivity can be used to analysis the reliability of network. The vertex connectivity of the graph representing network equals the minimum number of

routers that disconnect the network when they are out of service. The edge connectivity represents the minimum number of fiber optic links that can be down to disconnect the network.

10.4.3 Connectivities in Digraphs

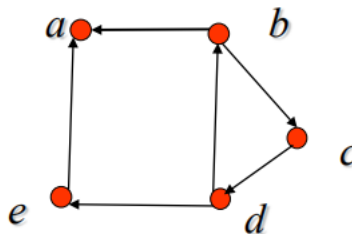
Here G is a digraph.

G is said to be **strongly connected** if there is a path from a to b and from b to a for all vertices a and b . If the underlying undirected graph is connected, then we say it is **weakly connected**.

⚠ By the definition, any strongly connected directed graph is also weakly connected.

For directed graph, the maximal strongly connected subgraphs are called the **strongly connected components** or just the **strong components**.

🔴 e.g.



There are 3 strong components in this digraph:

1. a .
2. e .
3. The subgraph consisting of vertices b , c , and d and edges (b, c) , (c, d) , (d, b) .

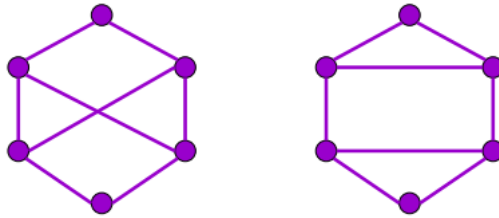
💡 定义可达性矩阵: $P = [p_{ij}]_{n \times n}$, 其中 p_{ij} 为 1 表示存在一条从结点 i 到 j 的路径, 为 0 表示不存在。那么 P 就是原有向图的邻接矩阵的传递闭包。

10.4.4 Applications of Paths

📖 Two graphs are isomorphic only if they have simple circuits of the same length.

📖 Two graphs are isomorphic only if they contain paths that go through vertices so that the corresponding vertices in the two graphs have the same degree.

🔴 e.g.



Are these two graphs isomorphic?

Solution:

They are not, because the right graph contains circuits of length 3, while the left graph does not.

▣ The number of different paths of length r from v_i to v_j is equal to the (i, j) th entry of A^r (the standard power of A , not the Boolean product), where A is the adjacency matrix representing the graph consisting of vertices v_1, v_2, \dots, v_n .

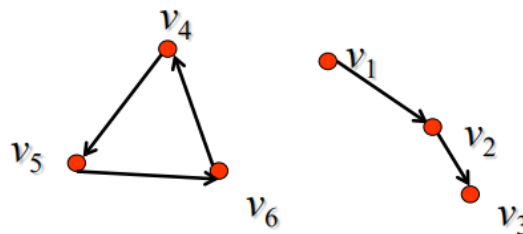
Proof:

The proof is based on MI. Let $A = [a_{ij}]_{n \times n}$.

Basic Step: $r = 1$.

Inductive Case: Assuming that the (i, j) th entry of A^r is the number of different paths of length r from v_i to v_j . Then for $r + 1$, let $A^{r+1} = A \cdot A^r = [d_{ij}]_{n \times n}$, then $d_{ij} = \sum_{k=1}^n c_{ik} a_{kj}$, where $c_{ik} a_{kj}$ is the number of paths of length $r + 1$ passing the vertex k .

● e.g.



1. How many paths of length 2 are there from v_5 to v_4 ?

Solution: a_{54} in A^2 is 1.

2. The number of paths not exceeding 6 are there from v_4 to v_5 ?

Solution: a_{54} in $A + A^2 + A^3 + A^4 + A^5 + A^6$ is 2.

3. The number of circuits starting at vertex v_5 whose length is not exceeding 6?

Solution: a_{55} in $A + A^2 + A^3 + A^4 + A^5 + A^6$ is 2.

10.5 Euler and Hamilton Paths

10.5.1 Euler Path

Euler Path : A simple path containing every edge of G . 这里 simple 的意思是同一条边不能走多次, 也就是说 Euler path 解决的是一笔画问题。

Euler Circuit : A simple circuit containing every edge of G .

Euler Graph : A graph contains an Euler circuit.

📄 Necessary and Sufficient Condition for Euler Circuits and Paths

A connected multigraph has an Euler circuit \leftrightarrow each of its vertices has even degree.

Proof:

(1) G has an Euler circuit \Rightarrow Every vertex in V has even degree

Consider the Euler circuit starting and ending at vertex a :

- First edge of the Euler circuit contributes one to the degree of a .
- Each time the circuit passes through a vertex it contributes two to the vertex's degree
- The circuit terminates where it started, contributing one to $\deg(a)$.

(2) Every vertex in V has even degree \Rightarrow We can form a Euler circuit that begins at an arbitrary vertex a of G

- Build a simple circuit $x_0 = a, x_1, x_2, \dots, x_n = a$.
- An Euler circuit has been constructed if all the edges have been used. Otherwise:
- Construct a simple path in the subgraph H obtained from G . Let w be a vertex which is the common vertex of the circuit and H . Beginning at w , construct a simple path in H .
- Form a circuit in G by splicing the circuit in H with the original circuit in G .
- Continue this process until all edges have been used.
- (Explained in detail later)

📄 A connected multigraph has an Euler path but not an Euler circuit \leftrightarrow It has exactly two vertices of odd degree.

Proof:

(1) Suppose the multigraph has an Euler path from a to b .

- The first edge of the Euler path contributes one to the degree of a .
- The last edge in the Euler path contributes one to the degree of b .
- The path contributes two to the degree of a vertex whenever it passes through it.

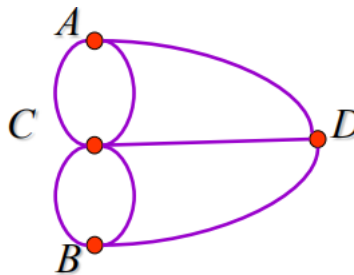
(2) Suppose that a graph has exactly two vertices of odd degree, say a and b .

- Add an edge $\{a, b\}$, then every vertex has even degree, so there is an Euler circuit.
- The removal of the new edge produces an Euler path.

🟡 e.g.

Königsberg Seven Bridge Problem:

Is it possible to pass each of the 7 bridges(represented as edges in the graph) exactly once?



Solution:

The graph has four vertices of odd degree. Therefore, it does not have an Euler circuit. It does not have an Euler path either.

📄 An algorithm to find an Euler path or Euler circuit in a given graph

```

1  PROCEDURE Euler (G: connected and all vertices of even degree)
2  circuit := a circuit in G beginning at an arbitrarily chosen vertex with
3  edges successively added to form a path that return to this ver
   tex
4  H := G with the edges of this circuit removed
5  WHILE H has edges
6  BEGIN
7  subcircuit := a circuit in H beginning at a vertex in H that is also an
8  endpoint of an edge of circuit
9  H := H with edges of subcircuit and all isolated vertices removed
10 circuit := circuit with subcircuit inserted at the appropriate vertex
11 END {circuit is an Euler circuit}

```

To form an Euler path in a graph without an Euler circuit, we follow the procedure above as well, but start and end at the two vertices of odd degrees.

e.g.

Determine whether the following graph has an Euler path. Construct such a path if it exists.

| 1 | 2 | 3 |
|------------|---|---|
| | | |
| ACEABCDEGJ | ACEABCDEGJ+ <u>EFGIJE</u> =ACE <u>EFGIJE</u> ABCDEGJ | ACEFGIJEABCDE <u>G</u> HIGJ+ <u>GHI</u> <u>G</u> =ACEFGIJEABCDE <u>GHI</u> GJ |

Solution:

The graph has exactly 2 vertices of odd degree, and all of the other vertices have even degree. Therefore, this graph has an Euler path.

The concept of Euler cuicuit can be extended to digraphs.

A directed multigraph having no isolated vertices has an Euler circuit if and only if:

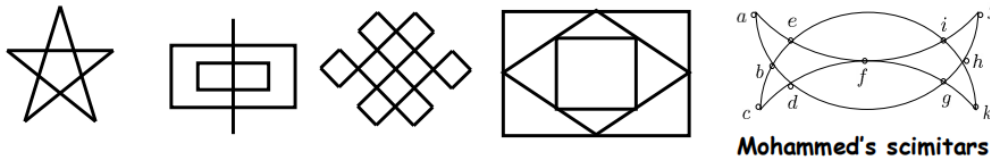
- The graph is weakly connected.
- The in-degree and out-degree of each vertex are equal.

■ A directed multigraph having no isolated vertices has an Euler path but not an Euler circuit if and only if:

- The graph is weakly connected.
- The in-degree and out-degree of each vertex are equal for all but two vertices, one that has in-degree 1 larger than its out-degree and the other that has out-degree 1 larger than its in-degree.

⚙ Application of Euler Paths

- One-Stroke Problem



Draw a picture in a continuous motion without lifting the pencil so that no part of the picture is retraced. The problem is equivalent to deciding whether there is a Euler path and constructing the path if it exists.

Some route planning problems, where we need to avoid retracing a road, are equivalent to a one-stroke problem.

- The Chinese Postman Problem (CPP)

中国邮递员问题是邮递员在某一地区的信件投递路程的问题。邮递员每天从邮局出发，走遍该地区所有街道再返回邮局，那么他应如何安排送信的路线可以使所走的总路程最短？这个问题由中国学者管梅谷在1960年首先提出，并给出了解法——“奇偶点图上作业法”。用图论的语言描述，就是给定一个连通图 G ，每边 e 有非负权，要求一条回路经过每条边至少一次，且满足总权最小。

10.5.2 Hamilton's Paths

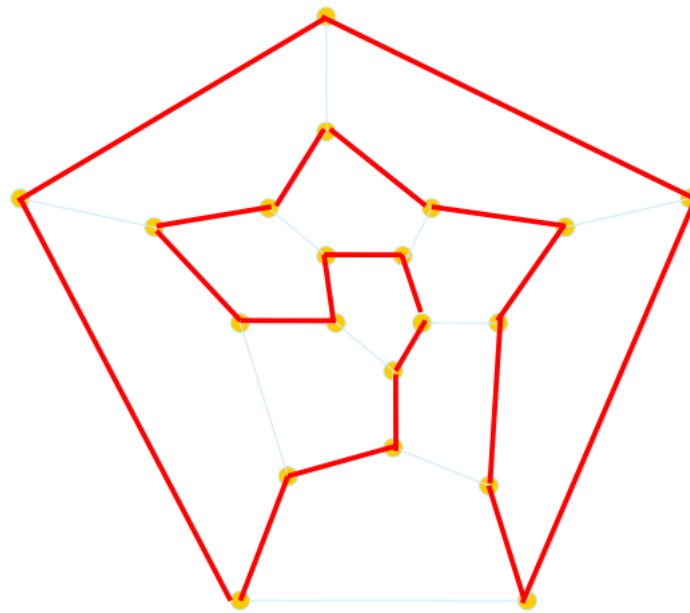
Hamilton Path : A path which visits every vertex in G exactly once.

Hamilton Circuit : A cycle which visits every vertex exactly once, except for the first vertex, which is also visited at the end of the cycle.

Hamilton Graph : A connected graph G with a Hamilton circuit

The definitions apply both to undirected as well as directed graphs of all types.

The idea of Hamilton path rose from Hamilton's puzzle, where we try passing every vertex of a graph without passing any vertex twice.



Hamilton's Puzzle (1856)

Currently, there are no useful necessary and sufficient conditions for the existence of Hamilton circuit. A few sufficient conditions have been found.

📖 Dirac's Theorem (迪拉克定理)

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 $\frac{n}{2}$, then G has a Hamilton path.

📖 Ore's Theorem (奥尔定理)

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.

💡 这两个定理的核心理念就是：有足够多的边的图中存在哈密顿道路/回路。

📖 Certain properties can be used to show that a graph has no Hamilton circuit:


- A graph with a vertex of degree one cannot have a Hamilton circuit.
- A graph with a cut vertex cannot have a Hamilton circuit.
- If a vertex in the graph has degree two, then both edges that are incident with this vertex must be part of any Hamilton circuit.
- When a Hamilton circuit is being constructed and this circuit has passed through a vertex, then all remaining edges incident with this vertex, other than the two used in the circuit, can be removed from consideration.

Another Important Necessary Condition

If there is a Hamilton circuit in G , for any nonempty subset S of set V , the number of connected components in $G - S$ is less than or equal to $|S|$. 可以这么记：有哈密顿回路说明这个图是连通的，每去掉一个结点就能产生一个新的连通分支已经是最极端的情况了。

 Note:

Suppose that C is a H circuit of G . For any nonempty subset S of set V , the number of connected components in $C - S$ is less than or equal to $|S|$, and the number of connected components in $G - S$ is less than or equal to the number of connected components in $C - S$. 可以这么记： C 中可能去掉了一些 G 中有的边，所以连通性只能比 G 差，连通分支数量一定不少于 G 。

 Hamilton path or circuit can be used to solve many practical problems, including the famous TSP (Traveling Salesperson Problem), where the salesperson wants to reach every city exactly once.

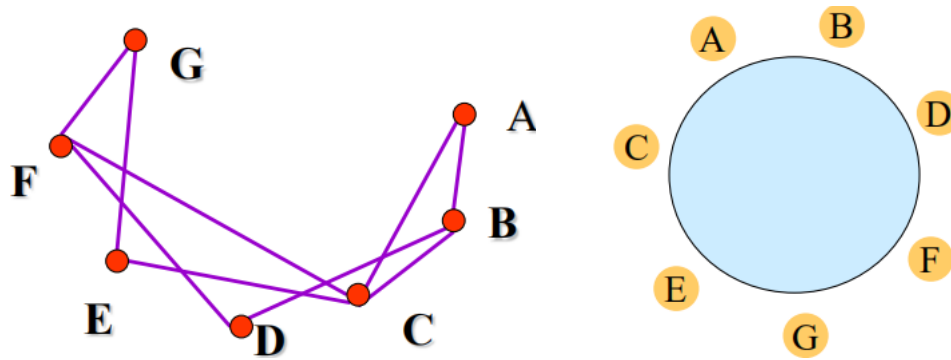
 e.g.

Seating Problem: There are seven people denoted by A, B, C, D, E, F, G. Suppose that the following facts are known.

- A -- English (A can speak English.)
- B -- English, Chinese
- C -- English, Italian, Russian
- D -- Japanese, Chinese
- E -- German, Italia
- F -- French, Japanese, Russian
- G -- French, German

How to arrange seat for the round desk such that the seven people can talk each other?

Solution:



1. Construct a graph G , where $V = \{A, B, C, D, E, F, G\}$,
 $E = \{(u, v) | u, v \text{ can speak at least one common language}\}$
2. If there is a H circuit, then we can arrange seat for the round desk such that the seven people can talk each other. The H circuit is A, B, D, F, G, E, C, A .

● e.g.

Seven examination must be arranged in a week. Each day has one examination. The examinations are in charged by the same teacher cannot be arranged in the adjacent two days. One teacher is in charge at most four examinations. Show that the arrangement is possible.

Proof:

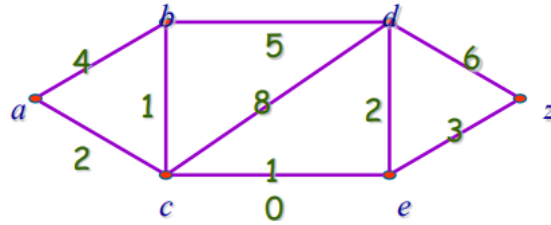
1. Construct graph, where V : seven examination,
 $E = \{(u, v) | u, v \text{ are examinations not in charge of the same teacher}\}$
2. If there is a Hamilton path, then the arrangement is possible. Since one teacher is in charge at most four examinations, then every vertex in the graph has at least three adjacent vertices. It follows that the sum of the degrees of a pair vertices is at least 6.
3. According to Ore's theorem, there is a H circuit connecting 6 of the vertices. By removing an edge and connect an endpoint to the 7th vertex, we get a H path.

10.6 Shortest Path Problems

In real life, we usually need to find a path to our destination with the lowest cost, but the graphs we have learned actually regard every edge as the same. So we need to introduce weight into graphs to represent the different cost of passing each edge.

10.6.1 Basic Concepts

Weighted graph : $G = (V, E, W)$, where W is the set of the **weights** (权重) of all edges. The length of a path in a weighted graph is defined as the sum of the weights of the edges of this path.



Shortest Path Problem : $G = (V, E, W)$ is a weighted graph, where $w(x, y)$ is the weight of edge associated vertices x and y (if $(x, y) \notin E$, then $w(x, y) = \infty$), and $a, z \in V$, find the path with the shortest length between a and z .

10.6.2 Dijkstra's Algorithm

It is a greedy algorithm discovered by the Dutch mathematician E. Dijkstra, to solve the problem in undirected weighted graphs where all the weights are positive. Its main idea is to find the length of the shortest path from a to a first vertex, then the length of the shortest path from a to a second vertex, and so on, until the length of the shortest path from a to z .

Dijkstra's Algorithm

Let S_k denote the set of vertices after k iterations of labeling procedure.

1. Initialization. Label a with 0 and other with ∞ , i.e. $L_0(a) = 0$, and $L_0(v) = \infty$ and $S_0 = \varphi$.
2. Form S_k . The set S_k is formed from S_{k-1} by adding a vertex u not in S_{k-1} with the smallest label.
3. Update the labels of all vertices not in S_k , so that $L_k(v)$, the label of the vertex v at the k th stage, is the length of the shortest path from a to v that containing vertices only in S_k . This shortest path is either the shortest path from a to v containing only elements of S_{k-1} or it is the shortest path from a to u at the $(k - 1)$ st stage with the edge (u, v) added. That is

$$L_k(v) = \min\{L_{k-1}(v), L_{k-1}(u) + w(u, v)\} .$$
4. Step 2 and 3 is iterated by successively adding vertices to the distinguished set the until z is added.

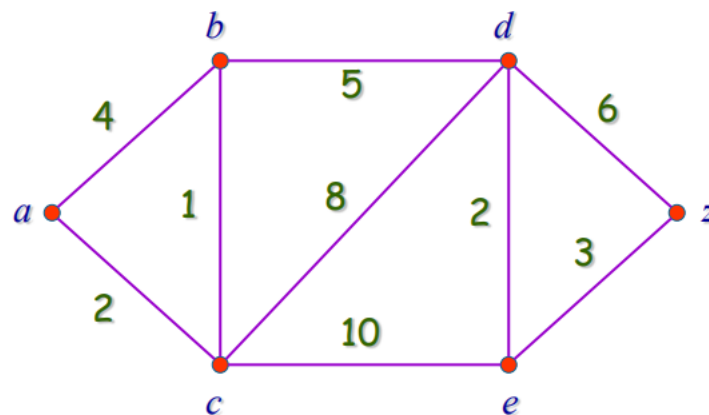
```

1  Procedure Dijkstra(G: weighted connected simple graph, with all weights po
   sitive)
2  {G has vertices  $a = v_0, v_1, \dots, v_n = z$  and weights  $w(v_i, v_j)$ ,
3  where  $w(v_i, v_j) = \infty$  if  $\{v_i, v_j\}$  is not an edge in G}
4  For  $i := 1$  to  $n$ 
5     $L(v_i) := \infty$ 
6   $L(a) := 0$ 
7   $S := \emptyset$ 
8  {the labels are now initialized so that the label of  $a$  is zero and all oth
   er labels
9  are  $\infty$ , and  $S$  is the empty set}
10 While  $z \notin S$ 
11 Begin
12    $u :=$  a vertex not in  $S$  with  $L(u)$  minimal
13    $S := S \cup \{u\}$ 
14   for all vertices  $v$  not in  $S$ 
15     if  $L(u) + w(u, v) < L(v)$ 
16        $L(v) := L(u) + w(u, v)$ 
17 {this adds a vertex to  $S$  with minimal label and updates the labels of vert
   ices not in  $S$ }
18 End { $L(z)$  = length of shortest path from  $a$  to  $z$  }

```

🟡 e.g.

Find the length of the shortest path between a and z in the given weighted graph.



Solution:

In performing Dijkstra's algorithm, it is sometimes more convenient to keep track of labels of vertices using a table. Mark the shortest path found in each iteration with red color.

| Vertex | Link | L_0 | L_1 | L_2 | L_3 | L_4 | L_5 |
|--------|------|-------|-------|-------|-------|-------|-------|
|--------|------|-------|-------|-------|-------|-------|-------|

| | | | | | | | |
|---|-----------|---|---|----|----|----|----|
| a | | 0 | | | | | |
| b | a→c | ∞ | 4 | 3 | | | |
| c | a | ∞ | 2 | | | | |
| d | a→c→b | ∞ | ∞ | 10 | 8 | | |
| e | a→c→b→d | ∞ | ∞ | 12 | 12 | 10 | |
| z | a→c→b→d→e | ∞ | ∞ | ∞ | ∞ | 14 | 13 |

💡 Such greedy algorithm can also be applied to digraphs.

📖 The Correctness of Dijkstra's Algorithm

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

Proof:

We use an inductive argument. Take as the induction hypothesis the following assertion: At the k th iteration, ① the label of every vertex v in S is the length of the shortest path from a to this vertex, and ② the label of every vertex not in S is the length of the shortest path from a to this vertex that contains only vertices in S .

1. When $k = 0$, $L_0(a) = 0$, $L_0(v) = \infty$, $S = \emptyset$.
2. Assume that the inductive hypothesis holds for the k th iteration. Let v be the vertex added to S at the $(k + 1)$ st iteration so that v is a vertex not in S at the end of the k th iteration with the smallest label.
 - a. ① still holds at the end of the $(k + 1)$ st iteration. The vertices in S before the $(k + 1)$ st iteration are labeled with the length of the shortest path from a , and v must be labeled with the length of the shortest path to it from a .
 - b. Let u be a vertex not in S after $(k + 1)$ iteration. A shortest path from a to u containing only elements of S either contains v or it does not. If it does not contain v , then by the inductive hypothesis its length is $L_k(u)$. If it does contain v , then it must be made up of a path from a to v of the shortest possible length containing elements of S other than v , followed by the edge from v to u . In this case its length would be $L_k(v) + w(v, u)$. This shows that ② is true, because $L_{k+1}(v) = \min\{L_k(v), L_k(u) + w(u, v)\}$.

📖 The Computational Complexity of Dijkstra's Algorithm

Dijkstra's algorithm uses $O(n^2)$ operations (additions and comparisons) to find the length of the shortest path between two vertices in a connected simple undirected weighted graph.

Proof:

The Dijkstra's algorithm uses no more than $(n - 1)$ iterations. In each iteration, it uses no more than $(n - 1)$ comparisons to determine the vertex not in S_k with the smallest label, and no more than $2(n - 1)$ operations are used to update no more than $(n - 1)$ labels.

10.6.3 The Traveling Salesperson Problem (TSP)

A traveling salesperson wants to visit each of n cities exactly once and return to his starting point with minimum total cost. The equivalent problem for TSP is to find a Hamilton circuit with minimum total weight in the weighted complete undirected graph.

The most straightforward method to solve TSP is to examine all possible Hamilton circuits and select one of minimum total length. There are $\frac{(n - 1)!}{2}$ H circuits in a complete graph with n vertices, so the complexity of this algorithm grows extremely rapidly. So we usually use approximation algorithms that do not necessarily produce the exact solution, to produce a solution that is close to an exact solution of TSP.

10.7 Planar Graphs

10.7.1 Basic Concepts

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. We can prove that a graph is planar by displaying a planar representation.

⚠️ A graph may be planar even if it is usually drawn with crossings. For example, K_4 and Q_3 are actually planar.

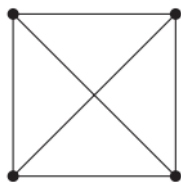


FIGURE 2 The graph K_4 .

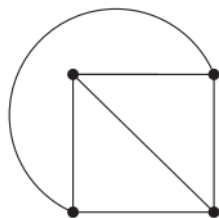


FIGURE 3 K_4 drawn with no crossings.

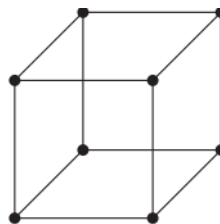


FIGURE 4 The graph Q_3 .

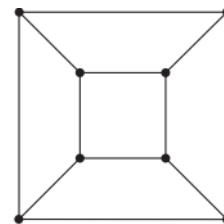


FIGURE 5 A planar representation of Q_3 .

e.g.

Complete bipartite graphs $K_{2,n} (n \geq 1)$ and $K_{1,n}$ are planar.

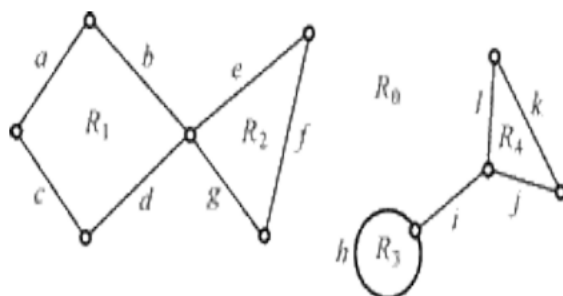
Planarity of graphs plays an important role in the design of electronic circuits and the design of road networks.

A part of the plane completely disconnected off from other parts of the plane by the edges of the graph is called a **region**. The edges that disconnect a region from other parts is called the **boundary** of the region. If the region has a finite area among several edges, then it is called a **bounded** region, and otherwise it is called an **unbounded** region.

Suppose R is a region of a connected planar simple graph, the number of the edges which surround R , is called the **degree** of region R , denoted as $\deg(R)$. Two regions with a common border are called **adjacent regions**.

e.g.

The following is a planar representation of a graph.



There are 5 regions. The boundaries of regions R_1 , R_2 , R_3 and R_4 are $abdc$, efg , h , kjl . So $\deg(R_1) = 4$, $\deg(R_2) = 3$, $\deg(R_3) = 1$, $\deg(R_4) = 3$. The boundary of unbounded region R_0 is constructed by $abefgdc$ and $kjihil$, so $\deg(R_0) = 13$.

⚠ Note that ij is counted twice.

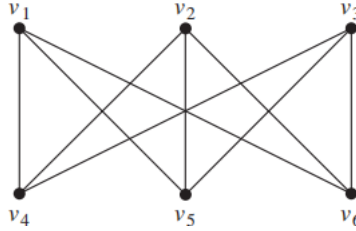
📁 If e is not a cut edge, then it must be the common border of two regions.

分析：如果 e 不是割边，那么一定存在一条包含了 e 的回路，这条回路里面包着至少一个区域，而 e 的两边分别是回路里外，必定分属不同的区域。感谢@lsshiki修的指导！

📖 The sum of the degrees of all regions is exactly twice the number of edges in the planar graph. That is $2e = \sum_{\text{all region } R} \deg(R)$.

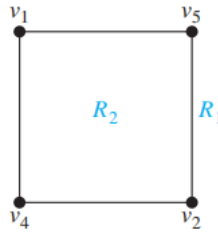
🔴 e.g.

Show that $K_{3,3}$ is not planar.

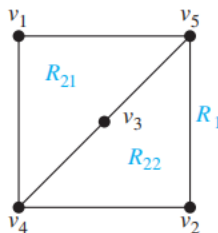


Proof:

In any planar representation of $K_{3,3}$, the vertices v_1 and v_2 must be connected to both v_4 and v_5 . These four edges form a closed curve that splits the plane into two regions, R_1 and R_2 . The vertex v_3 is in either R_1 or R_2 .



When v_3 is in R_2 , the inside of the closed curve, the edges between v_3 and v_4 and between v_3 and v_5 separate R_2 into two subregions, R_{21} and R_{22} . There is no way to place the final vertex v_6 without forcing a crossing.



For if v_6 is in R_1 , then the edge between v_6 and v_3 cannot be drawn without a crossing. If v_6 is in R_{21} , then the edge between v_2 and v_6 cannot be drawn without a crossing. If v_6 is in R_{22} , then the edge between v_1 and v_6 cannot be drawn without a crossing. And a similar argument can be used when v_3 is in R_1 .

10.7.2 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:

First, we specify a planar representation of G . We will prove the theorem by constructing a sequence of subgraphs $G_1, G_2, \dots, G_e = G$, successively adding an edge at each stage.

The constructing method: Arbitrarily pick one edge of G to obtain G_1 . Obtain G_n from G_{n-1} by arbitrarily adding an edge that is incident with a vertex already in G_{n-1} . Let r_n , e_n , and v_n represent the number of regions, edges, and vertices of the planar representation of G_n induced by the planar representation of G , respectively.

The relationship $r_1 = e_1 - v_1 + 2$ is true, since $e_1 = 1$, $v_1 = 2$, and $r_1 = 1$. Now assume that $r_n = e_n - v_n + 2$. Let $\{a_{n+1}, b_{n+1}\}$ be the edge that is added to G_n to obtain G_{n+1} .

If both a_{n+1} and b_{n+1} are already in G_n , then these two vertices must be on the boundary of a common region R , or else it would be impossible to add the edge $\{a_{n+1}, b_{n+1}\}$ to G_n without two edges crossing (and G_{n+1} is planar). The addition of this new edge splits R into two regions. Consequently, $r_{n+1} = r_n + 1$, $e_{n+1} = e_n + 1$, and $v_{n+1} = v_n$. Thus, $r_{n+1} = e_{n+1} - v_{n+1} + 2$.

Otherwise, if one of the two vertices of the new edge is not already in G_n , suppose that a_{n+1} is in G_n but that b_{n+1} is not. Adding this new edge does not produce any new regions, since b_{n+1} must be in a region that has a_{n+1} on its boundary. Consequently, $r_{n+1} = r_n$. Moreover, $e_{n+1} = e_n + 1$ and $v_{n+1} = v_n + 1$. Hence, $r_{n+1} = (e_{n+1} + 1) - (v_{n+1} + 1) + 2$.

Euler's formula Extended to Unconnected Simple Planar Graph

Suppose that a planar graph G has k connected components, e edges, and v vertices. Let r be the number of regions in a planar representation of G . Then $r = e - v + k + 1$.

Corollary 1

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

Proof:

Suppose that a connected planar simple graph divides the plane into r regions, the degree of each region is at least 3 (Note that the degree of a region in a simple graph is at least 3, because 三个不共线的点确定一个平面). Since $2e = \sum \deg(R_i) \geq 3r$, it implies $r \leq \frac{2}{3}e$. Using Euler's formula $e - v + 2 = r$, we obtain $e - v + 2 \leq \frac{2}{3}e$, which shows that $e \leq 3v - 6$.

The corollary also holds for unconnected planar simple graphs, because $e \leq 3v - 6$ holds for each of the connected components.

Corollary 2

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

Proof:

Similar to that of Corollary 1, except that in this case the fact that there are no circuits of length 3 implies that the degree of a region must be at least 4.

Corollary 3

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

Proof:

If G has 1 or 2 vertices, the result is true. If G has at least three vertices, by Corollary 1 we know that $e \leq 3v - 6$, so $2e \leq 6v - 12$. If the degree of every vertex were at least 6, then because $2e = \sum_{v \in V} \deg(v)$ by the handshaking theorem, we would have $2e \geq 6v$. But this contradicts the inequality $2e \leq 6v - 12$.

 e.g.

Show that K_5 , $K_{3,3}$ are nonplanar.

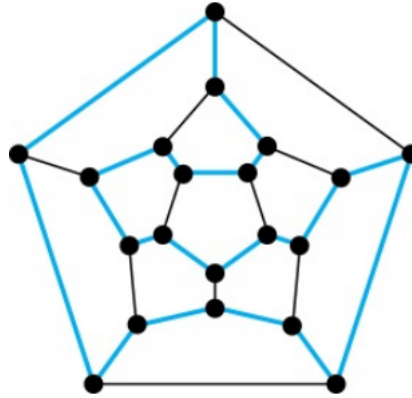
Proof:

The graph K_5 has 5 vertices and 10 edges. The inequality $e \leq 3v - 6$ is not satisfied for this graph. Therefore, K_5 is not planar.

$K_{3,3}$ has 6 vertices and 9 edges. Since $K_{3,3}$ has no circuits of length 3 (this is easy to see since it is bipartite), Corollary 2 can be used. The inequality $e \leq 2v - 4$ is not satisfied for this graph. Therefore, $K_{3,3}$ is nonplanar.

● e.g.

The construction of Dodecahedron.



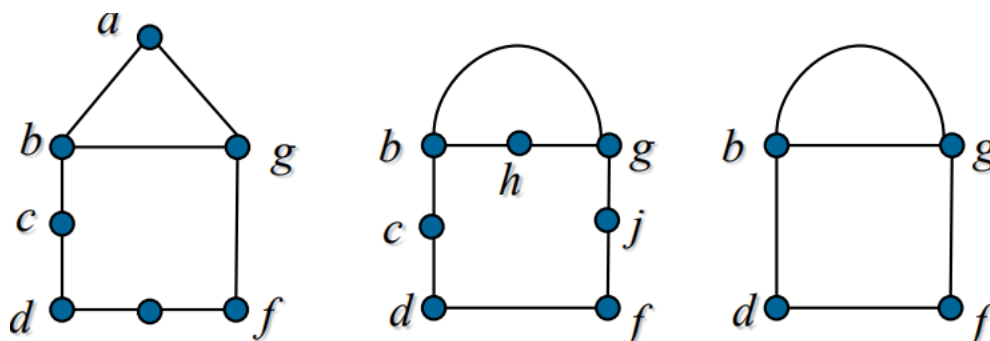
Solution:

Since the degree of every vertex is 3 and the degree of every region is 5. Then:

$$\begin{cases} 2e = \sum \deg(v_i) = 3v \\ 2e = \sum \deg(R_i) = 5r \\ r = e - v + 2 \end{cases} \Rightarrow \begin{cases} v = 20 \\ e = 30 \\ r = 12 \end{cases}$$

10.7.3 Kuratowski's Theorem

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 $\{w, v\}$, and this operation is called an **elementary subdivision** (初等细分). The graph $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ are called **homeomorphic** (同胚) if they can be obtained from the same graph by a sequence of elementary subdivision.



These three graphs are homeomorphic.

Kuratowski's Theorem (库拉托夫斯基定理)

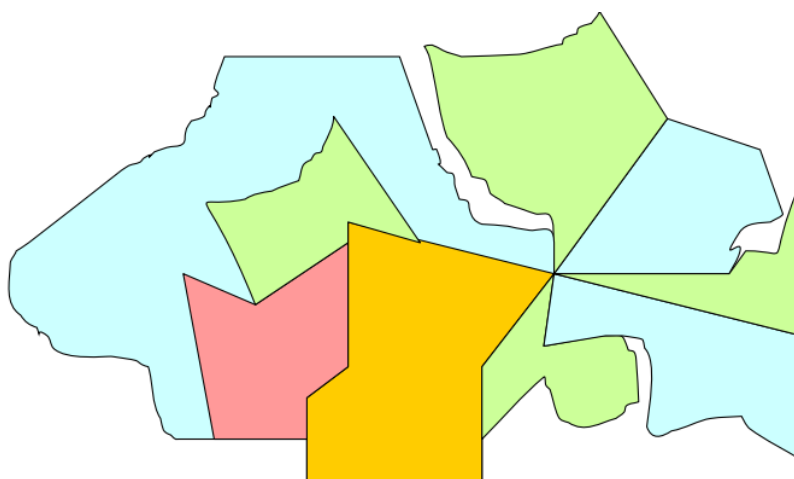
A graph is nonplanar \leftrightarrow it contains a subgraph homeomorphic to $K_{3,3}$ or K_5 .

Proof:

Obviously a graph containing a subgraph homeomorphic to $K_{3,3}$ or K_5 is nonplanar. Then we need to show that every nonplanar graph contains a subgraph homeomorphic to $K_{3,3}$ or K_5 . This part is so complicated that we don't need to learn it in this course. QwQ

10.8 Graph Coloring

The map coloring problem is about determining the least number of colors that can be used to color a map so that adjacent regions never have the same color. This problem can be reduced to a graph-theoretic problem.

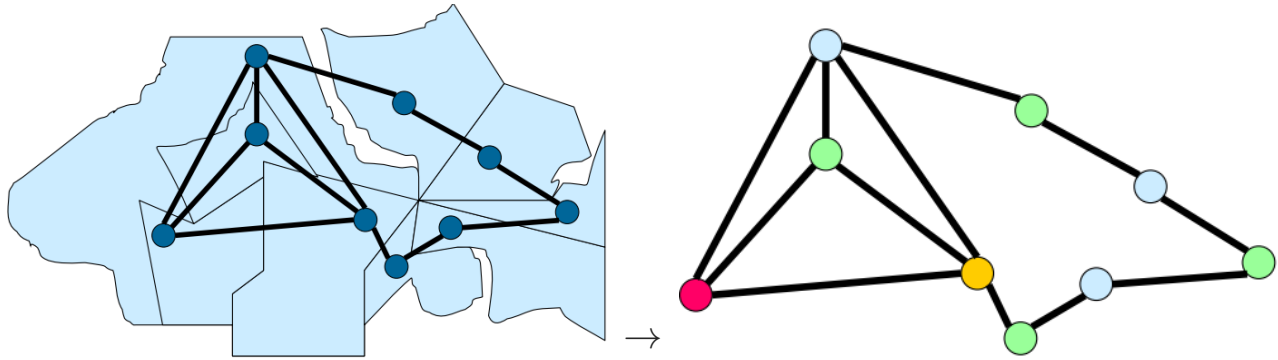


色图!

The **dual graph** of the map:

- Each region of the map is represented by a vertex.
- Edge connect two vertices iff the regions represented by these vertices have a common border.
- Two regions that touch at only one point are not considered adjacent.

As a terminology, **coloring** means the assignment of a color to each vertex of the graph so that no two adjacent vertices are assigned the same color. Coloring a map is equivalent to coloring the vertices of the dual graph so that no two adjacent vertices in this graph have the same color.



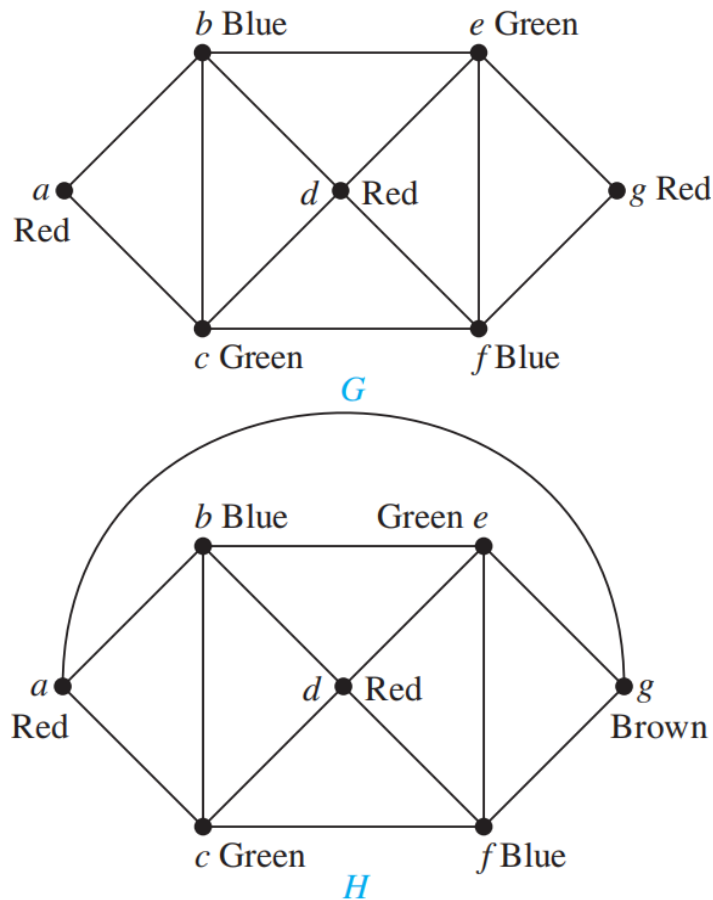
The **chromatic number**, denoted as $\chi(G)$, is the least number of colors needed for a coloring of a graph.

- If $G = K_1$, then $\chi(G) = 1$.
- If G is a path containing no circuits, then $\chi(G) = 2$.
- $\chi(C_n) = \begin{cases} 2, & \text{if } n \text{ is even} \\ 3, & \text{if } n \text{ is odd} \end{cases}$.
- $\chi(K_n) = n$, $\chi(K_n - e) = n - 1$.
- $\chi(K_{m,n}) = 2$. A simple graph with a chromatic number of 2 is bipartite. A connected bipartite graph has a chromatic number of 2.

Two things are required to show that the chromatic number of a graph is k . First, we must show that the graph can be colored with k colors. This can be done by constructing such a coloring. Second, we must show that the graph cannot be colored using fewer than k colors.

• e.g.

What are the chromatic numbers of the graphs G and H ?



Solution:

The chromatic number of G is at least 3, because the vertices a , b , and c must be assigned different colors. To see if G can be colored with 3 colors, assign red to a , blue

to b , and green to c . Then, d can (and must) be colored red because it is adjacent to b and c , and we have assumed that G can be colored using only the 3 colors.

Furthermore, e can (and must) be colored green because it is adjacent only to vertices colored red and blue, and f can (and must) be colored blue because it is adjacent only to vertices colored red and green. Finally, g can (and must) be colored red because it is adjacent only to vertices colored blue and green. This produces a coloring of G using exactly 3 colors.

The graph H is made up of the graph G with an edge connecting a and g . Any attempt to color H using three colors must follow the same reasoning as that used to color G , except at the last stage, when all vertices other than g have been colored. Then, because g is adjacent (in H) to vertices colored red, blue, and green, a fourth color, say brown, needs to be used. Hence, H has a chromatic number equal to 4.

💡 This solution shows a direct algorithm to find a coloring of a graph by trying to use color that has already been used at each step. Actually, the best algorithms known today for finding the chromatic number of a graph have exponential worst–case time complexity (in the number of vertices of the graph).

📖 The Four Color Theorem

The chromatic number of a planar graph is no greater than four. Or equivalently, any planar map of regions can be depicted using 4 colors so that no two regions that share a positive–length border have the same color.

The four color theorem was originally proposed as a conjecture in 1852. Many fallacious proof with hard–to–find errors were published. It was proved by Haken and Appel used exhaustive computer search in 1976. No proof not relying on computers has yet been found yet until today.

⚠️ The four color theorem applies only to planar graphs. Nonplanar graphs can have arbitrarily large chromatic numbers.

⚙️ Applications of Graph Coloring

1. Scheduling Exams: How can the exams at a university be scheduled so that no student has two exams at the same time?

Solution:

This scheduling problem can be solved using a graph model, with vertices representing courses and with an edge between two vertices if there is a common student in the courses they represent. Each time slot for a final exam is represented by a different color. A scheduling of the exams corresponds to a coloring of the associated graph.

2. Set up natural habitats of animal in a zoo.

Solution:

Let the vertices of a graph be the animals. Draw an edge between two vertices if the animals they represent cannot be in the same habitat because of their eating habits. A coloring of this graph gives an assignment of habitats.

11. Trees

11.1 Introduction to Trees

A **tree** is a connected simple graph with no simple circuits. A **forest** is a graph that has no simple circuit, but is not connected. Each of the connected components in a forest is a tree.

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

Proof:

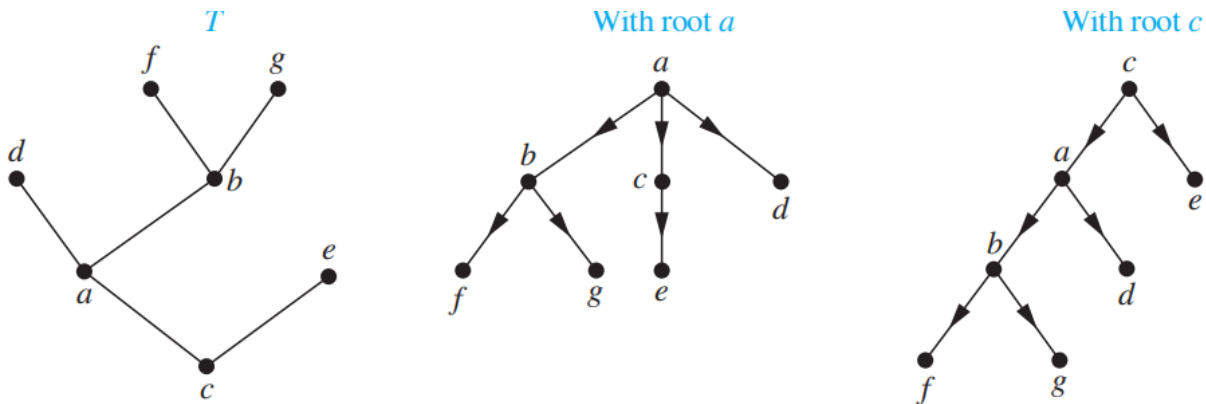
1. \Rightarrow

- There is a simple path between any two of its vertices because a tree is connected.
- The path is unique because there is no circuit.

2. \Leftarrow

- The graph is connected because there are paths between each pair of the vertices.
- The path is unique, so there is no circuit.

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. An unrooted tree is converted into different rooted trees when different vertices are chosen as the root.



The **parent** of a non-root vertex v is the unique vertex u with a directed edge from u to v . When u is the parent of v , v is called a **child** of u . Vertices with the same parent are called **siblings**.

The **ancestors** of a non-root vertex are all the vertices in the path from root to this vertex. The **descendants** of vertex v are all the vertices that have v as an ancestor. The **subtree** at vertex v is the subgraph of the tree consisting of vertex v and its descendants and all edges incident to those descendants. An **ordered rooted tree** is a rooted tree where the children of each internal vertex are ordered. In an ordered binary

tree, the two possible children of a vertex are called the **left child** and the **right child**, if they exist. The tree rooted at the left child is called the **left subtree**, and that rooted at the right child is called the **right subtree**.

A vertex is called a **leaf** if it has no children. A vertex that has children is called an **internal vertex**.

The **level** of vertex v in a rooted tree is the length of the unique path from the root to v , and the level of the root is defined to be 0. The **height** of a rooted tree is the maximum of the levels of its vertices.

A rooted tree is called a **m-ary tree** if every internal vertex has no more than m children. It is a **binary tree** if $m = 2$. The tree is called a **full m-ary tree** if every internal vertex has exactly m children. A rooted m-ary tree of height h is called **balanced** if all its leaves are at levels h or $h - 1$.

□ A tree with n vertices has $n - 1$ edges.

Proof:

$T = (V, E)$, $|V| = n$, $|E| = e$. Any tree must be planar and connected, so $r = e - n + 2$. Since any tree has no circuits, so $r = 1$, so $e = n - 1$.

● e.g.

A tree has two vertices of degree 2, one vertex of degree 3, three vertices of degree 4. How many leaves does this tree have?

Solution:

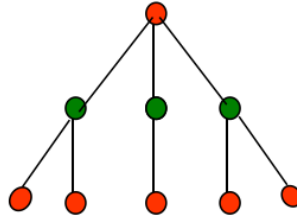
Suppose that there are x leaves.

$$\begin{cases} v = 2 + 1 + 3 + x \\ 2e = x \times 1 + 2 \times 2 + 1 \times 3 + 3 \times 4 \Rightarrow x = 9 \\ e = v - 1 \end{cases}$$

□ Every tree is a bipartite.

Proof:

Every tree can be colored using two colors. We can choose a root and color it red. Then we color all the vertices at odd levels blue, and all the vertices at even levels red.



▣ A full m -ary tree with i internal vertices contains $n = mi + 1$ vertices and $l = (m - 1)i + 1$ leaves.

Proof:

Every vertex, except the root, is the child of an internal vertex. Since each of the i internal vertices has m children, there are mi vertices in the tree other than the root. Therefore, the tree contains $n = mi + 1$ vertices.

▣ A full m -ary tree with n vertices has $i = \frac{n - 1}{m}$ internal vertices and $l = \frac{(m - 1)n + 1}{m}$ leaves.

▣ A full m -ary tree with l leaves has $n = \frac{ml - 1}{m - 1}$ vertices and $i = \frac{l - 1}{m - 1}$ internal vertices

▣ For a full binary tree, $l = i + 1$.

▣ There are at most m^h leaves in an m -ary tree of height h . This is proved using MI, from $h = 1$ to any height with respect to any m .

▣ Corollary

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

Proof:

1. $l \leq m^h$
2. Since the tree is balanced. Then each leaf is at level h or $h - 1$, and since the height is h , there is at least one leaf at level h . It follows that

$$\begin{cases} m^{h-1} < l \\ l \leq m^h \end{cases} \Rightarrow h - 1 < \log_m l \leq h$$

11.2 Applications of Trees

A **binary search tree (BST)** is an ordered rooted binary tree, each vertex of whom contains a distinct **key value**. The key values in the tree can be compared using "greater than" and "less than". The key value of each vertex is less than every key value in its right subtree, and greater than every key value in its left subtree.

Construction of BSTs

1. The first value to be inserted is put into the root.
2. Thereafter, each value to be inserted begins by comparing itself to the value in the root, moving left if it is less, or moving right if it is greater. This continues at each level, until it can be inserted as a new leaf.

Construct BST

Plain Text

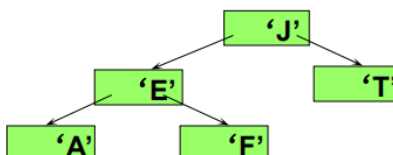
复制代码

```
1 Procedure insertion (T: binary search tree, x: item)
2   v:=root of T
3   While v≠null and label(v)≠x
4   Begin
5     if x<label(v) then
6       if left child of v ≠ null then
7         v:=left child of v
8       else
9         add new vertex as a left child of v and set v:=null
10    else
11      if right child of v ≠ null then
12        v:=right child of v
13      else
14        add new vertex as a right child of v and set v:=null
15  end
16  if root of T = null then
17    add a vertex r to the tree and label it with x
18  else if v is null or label(v)≠x then
19    label new vertex with x and let v be this new vertex
20  {v = location of x}
```

e.g.

Insert the elements 'J', 'E', 'F', 'T', 'A' in that order alphabetically.

Solution:



1. 'J' is the first value, so it should be put at the root.
2. 'E' is less than 'J', so it should be the left child of the root.
3. 'F' is less than 'J', so move left; then compare 'F' to 'E' to find 'F' is greater than 'E', so 'F' should be the right child of 'E'.
4. 'T' is greater than 'J', so it should be the right child of 'J'.
5. 'A' is less than 'J' and 'E', so it moves left twice and should be the left child of 'E'.

A rooted tree in which each internal vertex corresponds to a decision, with a subtree at these vertices for each possible outcome of the decision, is called a **decision tree**.

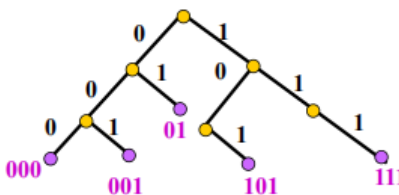
When using bit strings to encode the letters of the English alphabet, using bit strings of different lengths to encode letters can improve coding efficiency. But some method must be used to determine where the bits for each character start and end.

🟡 e.g.

e -- 0, a -- 1, t -- 01. Then 0101 can be "eat", "tea", "eaea" or "tt".

To ensure that no bit string corresponds to more than one sequence of letters, the bit string for a letter must never occur as the first part of the bit string for another letter. Codes with this property are called **prefix codes**.

A system of prefix codes can be constructed using a binary tree. The left edge at each internal vertex is labeled by 0, and the right edge at each internal vertex is labeled by 1. The leaves are labeled by characters which are encoded with the bit string constructed using the labels of the edges in the unique path from the root to the leaves



How to produce efficient codes based on the frequencies of occurrences of characters?

General problem:

Tree T has t leaves, w_1, w_2, \dots, w_t are weights representing the frequency of each leaf being used, l_i is the length of the code. Let the weight of tree T be

$$w(T) = \sum_{i=1}^t l_i w_i, \text{ then } \text{obj. min } w(T).$$

For example,

| | | | | | | |
|-----------|----|----|----|----|-----|-----|
| Letter | a | b | c | d | e | ... |
| Frequency | 82 | 14 | 28 | 38 | 131 | ... |


Let l_i be the length of prefix codes for letter i , then $obj. \min \sum_{i=1}^{26} l_i w_i$

Solution:

Huffman Coding

▼ Huffman Coding

Plain Text

 复制代码

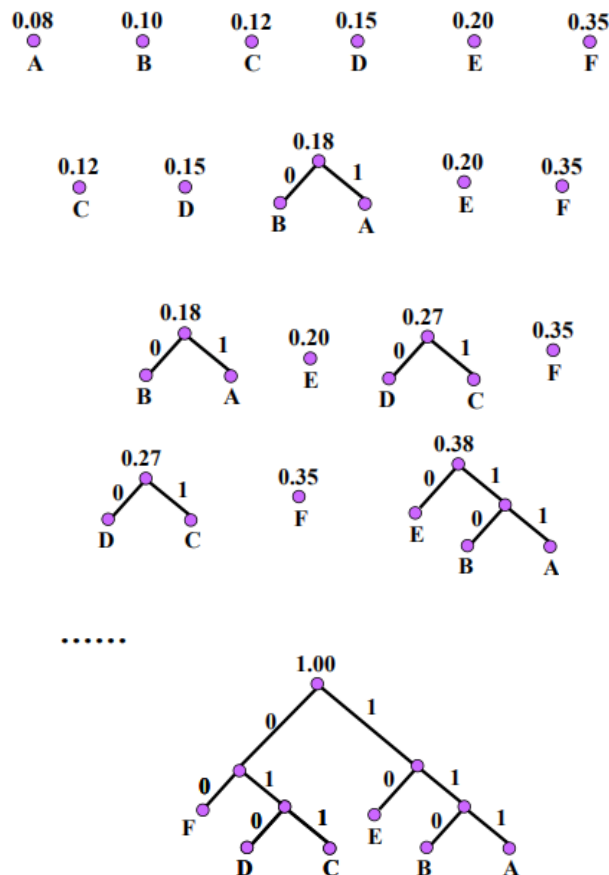
```

1  Procedure Huffman (C: symbols  $a_i$  with frequencies  $w_i$ ,  $i=1, \dots, n$ )
2  F:=forest of  $n$  rooted trees,
3    each consisting of the single vertex  $a_i$  and assigned weight  $w_i$ 
4  While F is not a tree
5  begin
6    Replace the rooted trees  $T$  and  $T'$  of least weights from F with  $w(T) \geq w(T')$ 
7    with a tree having a new root that has  $T$  as its left subtree and  $T'$  as its
8    right subtree. Label the new edge to  $T$  with 0 and the new edge to  $T'$  with 1.
9    Assign  $w(T) + w(T')$  as the weight of the new tree.
10 end
11 {The Huffman coding for the symbol  $a_i$  is the concatenation of the labels of
12 the edges in the unique path from the root to the vertex  $a_i$ }
```

 e.g.

Use Huffman coding to encode the following symbols with the frequencies listed: A -- 0.08, B -- 0.10, C -- 0.12, D -- 0.15, E -- 0.20, F -- 0.35. What is the average number of bits used to encode a character?

Solution:



At first, we have a forest of 6 single vertices, and list them in the order of increasing weight. The two vertices with the least weights are A and B, so construct a tree of weight 0.18 using them. List the forest in the order of increasing weight. Then the two vertices with the least weights are C and D, so construct a tree using them. List the forest in the order of increasing weight. Repeat this procedure until the forest become a tree.

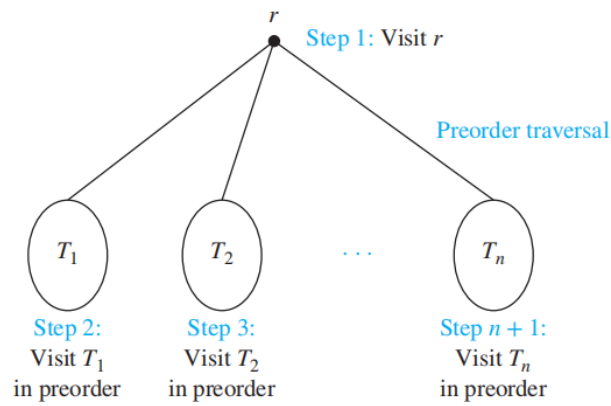
11.3 Tree Traversal

A **traversal algorithm** is a procedure for systematically visiting every vertex of an ordered rooted tree. Tree traversals are defined recursively.

Three traversals are named: **preorder** , **inorder** , **postorder** .

Preorder Traversal

Let T be an ordered tree with root r . If T has only r , then r is the preorder traversal of T . Otherwise, suppose T_1 , T_2 , ... , T_n are the left to right subtrees at r . The preorder traversal begins by visiting r . Then traverses T_1 in preorder, then traverses T_2 in preorder, and so on, until T_n is traversed in preorder.



▼ Preorder Traversal

Plain Text

📄 复制代码

```

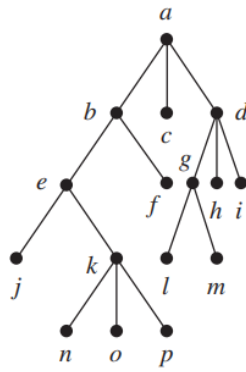
1  procedure preorder (T: ordered rooted tree)
2    r := root of T
3    list r
4    for each child c of r from left to right
5      T(c) := subtree with c as root
6      preorder(T(c))

```

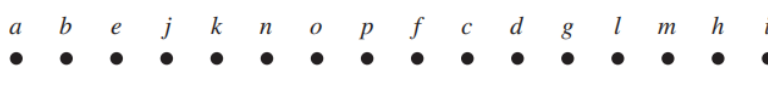
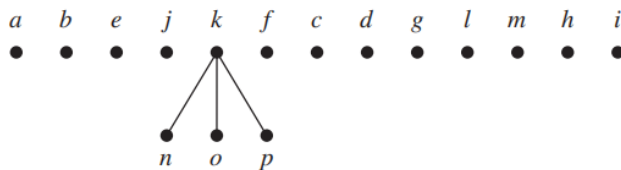
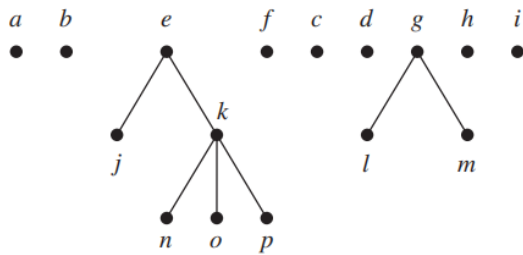
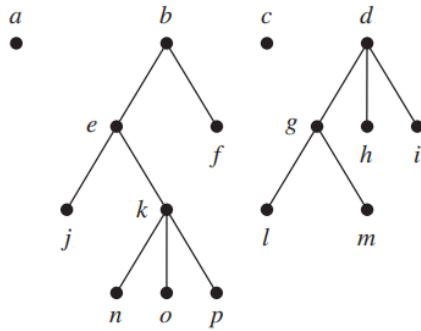
Preorder traversal of an binary ordered tree:

- Visit the root.
- Visit the left subtree, using preorder.
- Visit the right subtree, using preorder.

🍌 e.g.

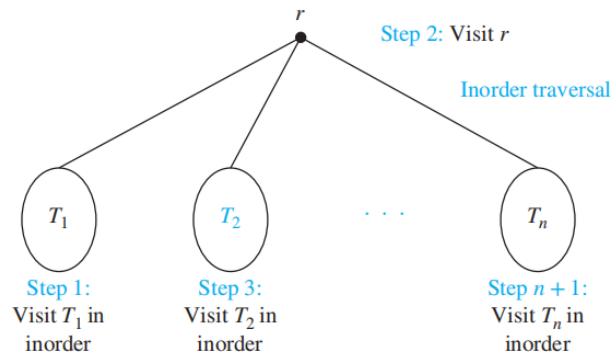


Preorder traversal: Visit root,
visit subtrees left to right



Inorder Traversal

Let T be an ordered tree with root r . If T has only r , then r is the inorder traversal of T . Otherwise, suppose T_1, T_2, \dots, T_n are the left to right subtrees at r . The inorder traversal begins by traversing T_1 in inorder. Then visits r , then traverses T_2 in inorder, and so on, until T_n is traversed in inorder.



▼ Inorder Traversal

Plain Text

📄 复制代码

```

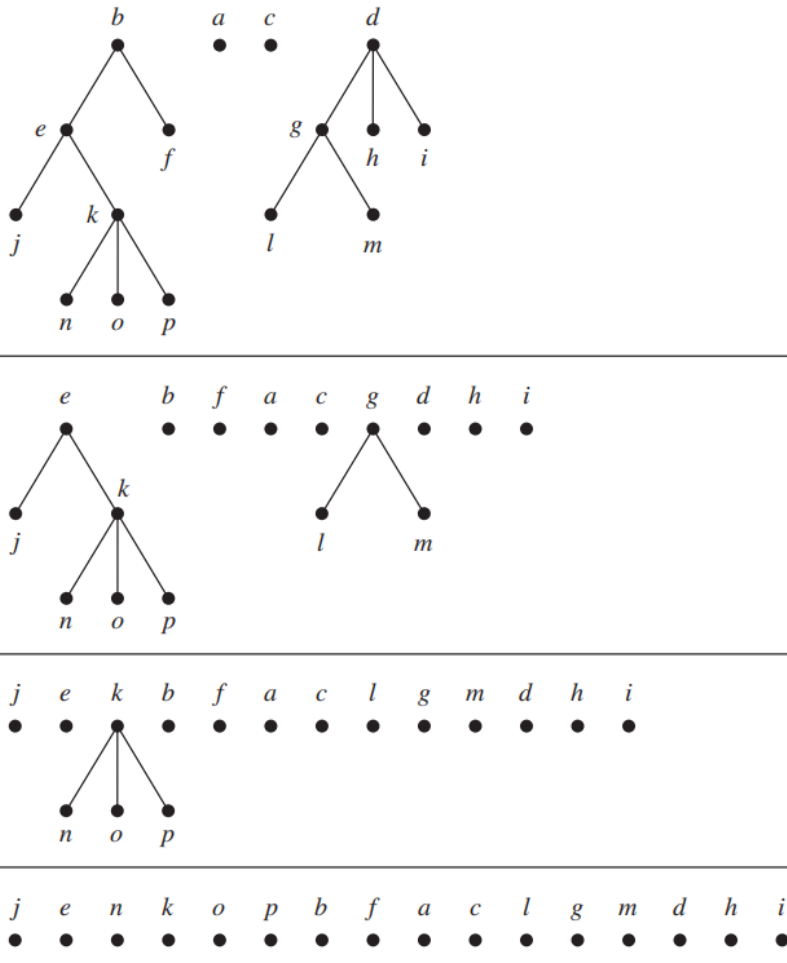
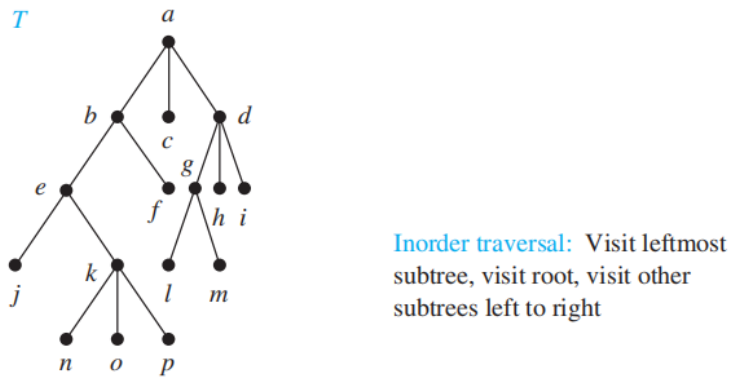
1  procedure inorder (T: ordered rooted tree)
2    r := root of T
3    if r is a leaf then
4      list r
5    else
6      l := first child of r from left to right
7      T(l) := subtree with l as its root
8      inorder(T(l))
9      list(r)
10   for each child c of r from left to right
11     T(c) := subtree with c as root
12     inorder(T(c))

```

Inorder traversal of an binary ordered tree:

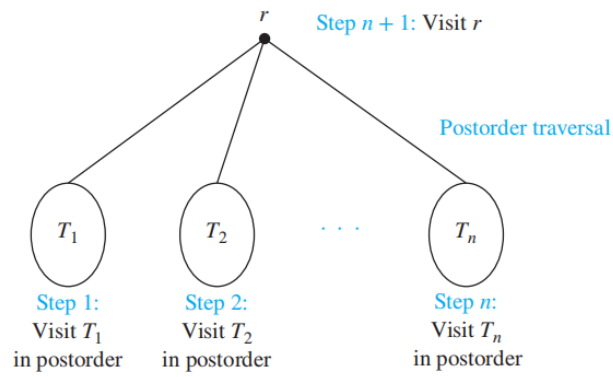
- Visit the left subtree, using inorder.
- Visit the root.
- Visit the right subtree, using inorder.

🍌 e.g.



Postorder Traversal

Let T be an ordered tree with root r . If T has only r , then r is the postorder traversal of T . Otherwise, suppose T_1, T_2, \dots, T_n are the left to right subtrees at r . The postorder traversal begins by traversing T_1 in postorder. Then traverses T_2 in postorder, until T_n is traversed in postorder, finally ends by visiting r .



▼ Postorder Traversal

Plain Text

📄 复制代码

```

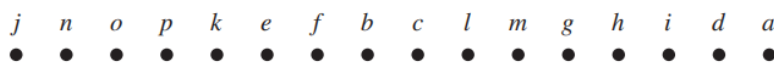
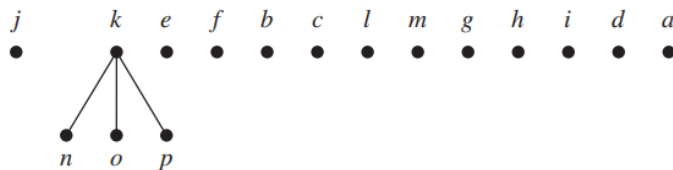
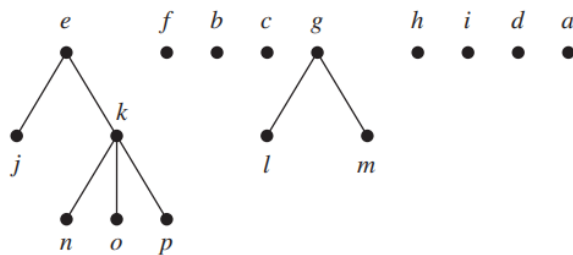
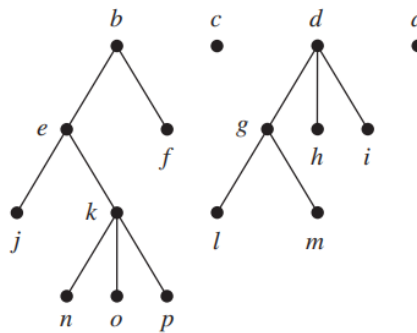
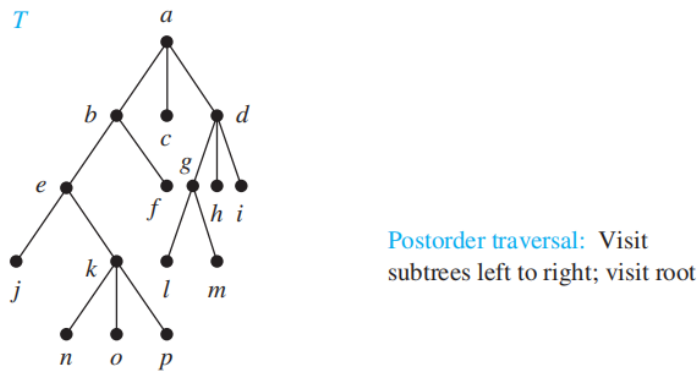
1  procedure postordered (T: ordered rooted tree)
2    r := root of T
3    for each child c of r from left to right
4      T(c) := subtree with c as root
5      postorder(T(c))
6    list r

```

Postorder traversal of an binary ordered tree:

- Visit the left subtree, using postorder.
- Visit the right subtree, using postorder.
- Visit the root.

🍌 e.g.



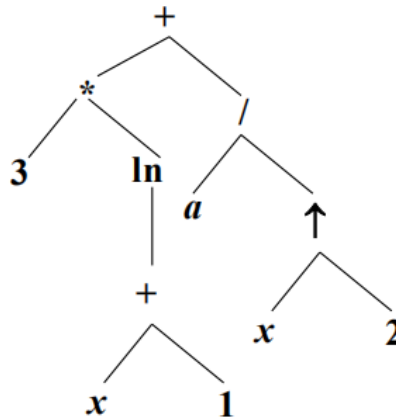
Complicated expressions, such as compound propositions, combinations of sets and arithmetic expressions, can be represented using ordered rooted trees called **expression trees**. A **binary expression tree** is a special kind of binary tree in which each leaf node contains a single operand, each nonleaf node contains a single operator, and the left and right subtrees of an operator node represent subexpressions that must be evaluated before applying the operator at the root of the subtree.

🍎 e.g.

What is the ordered tree that represents the expression $3 \ln(x + 1) + \frac{a}{x^2}$?

Solution:

A binary tree for the expression can be built from the bottom up, as is illustrated here. The higher the priority is, the nearer it should be to the leaves in the expression tree.



An inorder traversal of the expression tree produces the original expression, called the **infix form** (中綴形式), when parentheses are included except for unary operations, which now immediately follow their operands. For example, the infix form we get from the expression tree above is $3 * \ln(x + 1) + a / (x \uparrow 2)$.

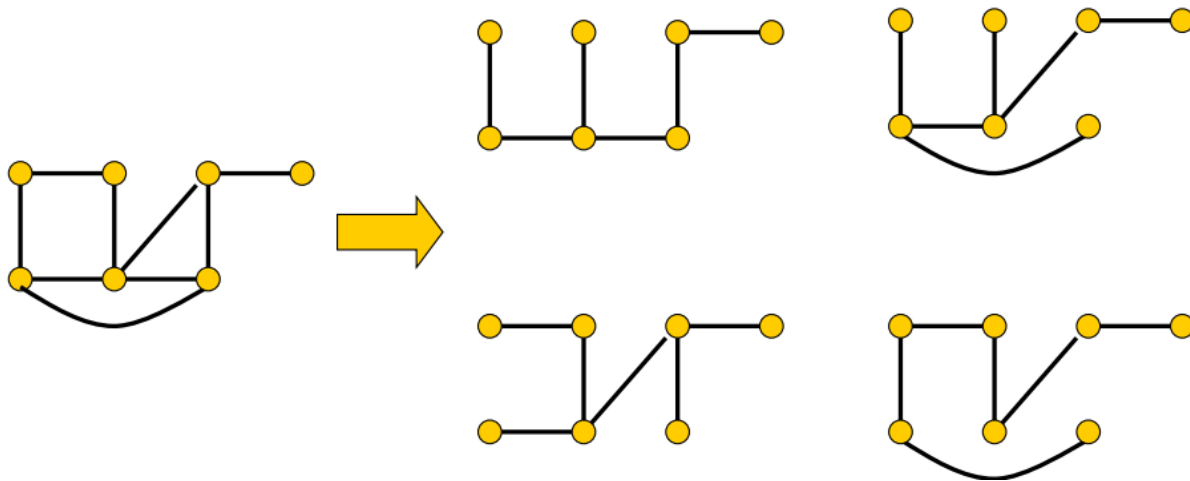
The expression obtained by an preorder traversal of the binary tree is said to be in **prefix form** (前綴形式) (Polish notation). For example, the prefix form of the expression tree above is $+ * 3 \ln + x 1 / a \uparrow x 2$. In a prefix form, operators precede their operands, parentheses are not needed, as the representation is unambiguous. Prefix expressions are evaluated by working from right to left.

The expression obtained by an postorder traversal of the binary tree is said to be in **postfix form** (后綴形式) (reverse Polish notation 逆波兰记号). For example, the postfix form of the expression tree above is $3x1 + \ln * ax2 \uparrow / +$. Parentheses are not needed as the postfix form is unambiguous, and it evaluate an expression by working from left to right.

11.4–11.5 Spanning Trees & Minimum Spanning Trees

Let G be a simple graph. A **spanning tree** (生成树) of G is a subgraph of G that is a tree containing every vertex of G . We can find spanning trees of a graph by removing edges from simple circuits (破圈法).

🍎 e.g.



■ A simple graph is connected if and only if it has a spanning tree.

Proof:

First, suppose that a simple graph G has a spanning tree T . T contains every vertex of G . There is a path in T between any two of its vertices. Since T is a subgraph of G , there is a path in G between any two of its vertices. Hence G is connected.

Second, suppose that G is connected. We can find a spanning tree by removing edges from simple circuits of G .

Apart from constructing spanning trees by removing edges, spanning trees can also be built up by successively adding edges. There are two algorithms: depth-first search and breadth-first search.

■ Depth-First Search (深度优先搜寻) / Backtracking (回溯)

This procedure forms a rooted tree, and the underlying undirected graph is a spanning tree.

1. Arbitrarily choose a vertex of the graph as root.
2. From a path starting at this vertex by successively adding edges, where each new edge is incident with the last vertex in the path and a vertex not already in the path.
3. Continue adding edges to this path as long as possible.
4. If the path goes through all vertices of the graph, the tree consisting of this path is a spanning tree.
5. If the path does not go through all vertices, more edges must be added. Move back to the next to last vertex in the path, if possible, form a new path starting at this vertex passing through vertices that were not already visited. If this cannot be done, move back another vertex in the path.

6. Repeat this process.

- ▼ Depth-First Search

Plain Text

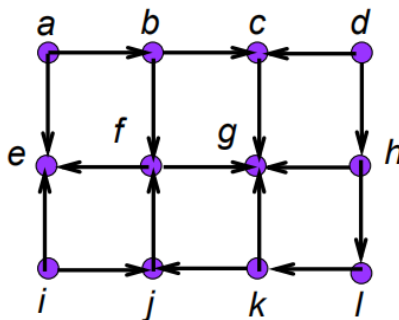
 复制代码

```

1  procedure DFS(G: connected graph with vertices v1, v2, ..., vn)
2    T := tree consisting only of the vertex v1
3    visit(v1)
4
5  procedure visit(v: vertex of G)
6    for each vertex w adjacent to v and not yet in T
7      add vertex w and edge {v,w} to T
8      visit(w)

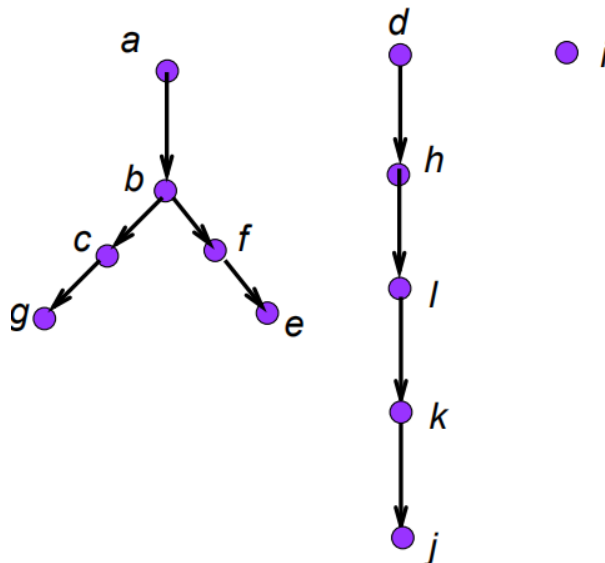
```

 e.g.



What is the output of depth-first search given the graph above as input?

Solution:



Breadth-First Search (广度优先搜寻)

1. Arbitrarily choose a vertex of the graph as a root, and add all edges incident to this vertex.
2. The new vertices added at this stage become the vertices at level 1 in the spanning

tree. Arbitrarily order them.

- For each vertex at level 1, visited in order, add each edge incident to this vertex to the tree as long as it does not produce a simple circuit. Arbitrarily order the children of each vertex at level 1. This produces the vertices at level 2 in the tree.
- Follow the same procedure until all the vertices in the tree have been added.

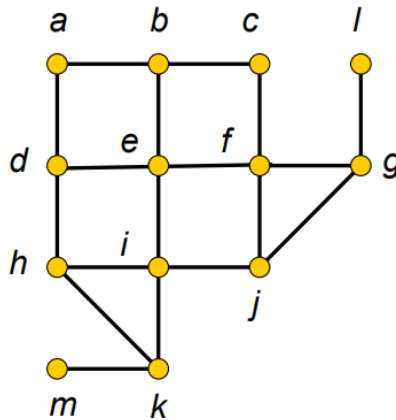
▼ Breadth-First Search

Plain Text

📄 复制代码

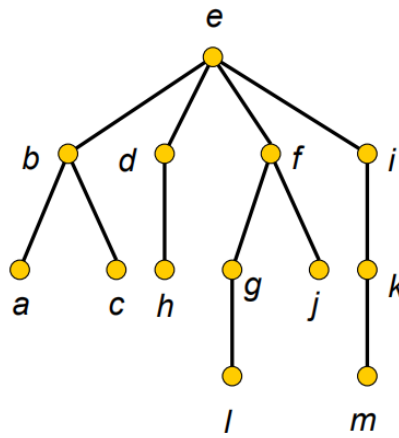
```
1  procedure BFS(G: connected graph with vertices v1, v2, ..., vn)
2  T := tree consisting only of the vertex v1
3  L := empty list visit(v1)
4  put v1 in the list L of unprocessed vertices
5  while L is not empty
6    remove the first vertex, v, from L
7    for each neighbor w of v
8      if w is not in L and not in T then
9        add w to the end of the list L
10     add w and edge {v,w} to T
```

🍎 e.g.



Use a breadth-first search to find a spanning tree for the graph above.

Solution:



There are problems that can be solved only by performing an exhaustive search of all possible solutions. One way to search systematically for a solution is to build a decision tree using **backtracking scheme** (回溯法), where each internal vertex represents a decision and each leaf a possible solution.


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

Two algorithms for constructing minimum spanning trees: Prim's algorithm and Kruskal's algorithm. These two algorithms are examples of greedy algorithms, both proceed by successively adding edges of smallest weight from those edges with a specified property that have not already been used.

Prim's Algorithm

▼ Prim's Algorithm

Plain Text

 复制代码

```

1  Procedure Prim (G: weighted connected undirected graph with n vertices)
2  T:= a minimum-weight edge
3  for i:= 1 to n-2
4  begin
5      e:= an edge of minimum weight incident to a vertex in
6          T and not forming a simple circuit in T if added to T.
7      T:= T with e added
8  end {T is a minimum spanning tree of G}
  
```

Kruskal's Algorithm (最简单)

▼ Kruskal's Algorithm

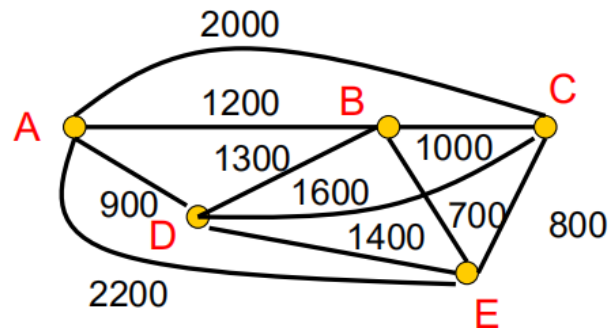
Plain Text | 复制代码

```

1  Procedure Kruskal (G: weighted connected undirected graph with n vertices)
2  T:= empty graph
3  for i:= 1 to n-1
4  begin
5      e:= any edge in G with smallest weight that does not
6          form a simple circuit when added to T
7      T:= T with e added
8  end {T is a minimum spanning tree of G}

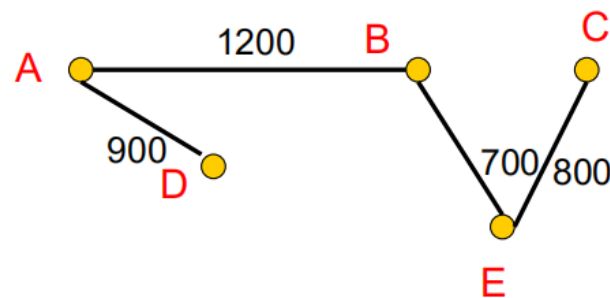
```

e.g.



Find a minimum spanning tree in the weighted graph.

Solution:



The minimum spanning tree

Prim's algorithm

| Choice | Edge | Cost |
|--------|------|------|
| 1 | BE | 700 |
| 2 | EC | 800 |
| 3 | BA | 1200 |
| 4 | AD | 900 |
| Total | | 3600 |

| Kruskal's algorithm | | |
|---------------------|------|------|
| Choice | Edge | Cost |
| 1 | BE | 700 |
| 2 | EC | 800 |
| 3 | AD | 900 |
| 4 | AB | 1200 |
| Total | | 3600 |