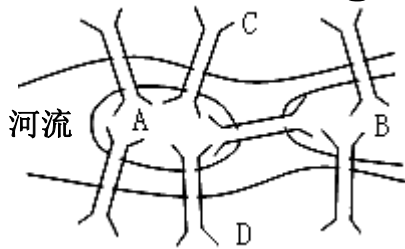


Ch7 图

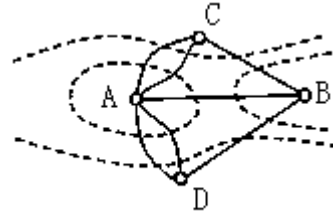
■ 几点说明

- **Mathematical background---Discrete mathematics (图论)**

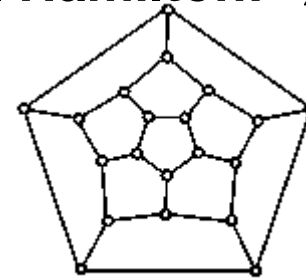
L. Euler: “Kölnsberg(哥尼斯堡)七桥问题”;



4块陆地7座桥:
“是否存在一条
路线能不重复
地走遍7座桥后
回到原地?”



W. Hamilton: “周游世界”

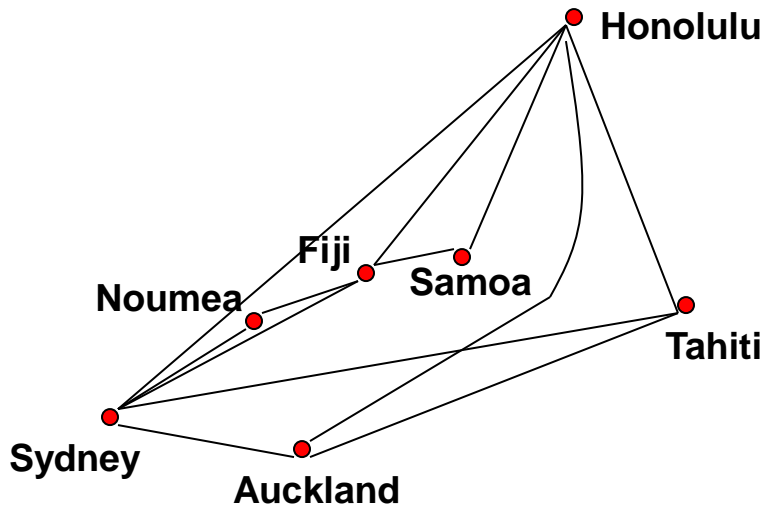


12正多面体中顶点
表示城市: “是否
有旅行路线沿着
交通线经过每个
城市恰好一次?”

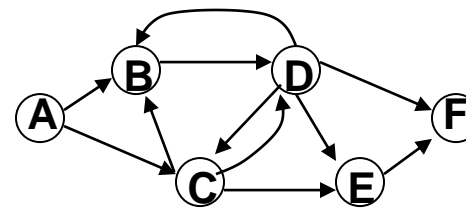
- 图(Graph)严格说应是数学结构而非数据结构, 元素之间有许多不规则的关系:

A classic story(N. Wirth)

- **Have applications in subjects as diverse as follows:**



Selected South Pacific air routes



Message
transmission in a
network



How to design the DS for a
very large SN like FB or 微信?
How to get the shortest
path between two people?

- **How to represent graphs in the data structures**


simple graph parallel edge selfloop

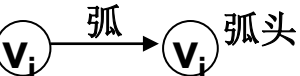
简单图: 不含有平行边和自回路的图

▪ Definitions and Examples

图G由 $V(G)$ 和 $E(G)$ 这两个集合组成, 记为 $G=(V, E)$. 其中

- $V(G)$ 是顶点(vertices)的非空有限集, 它一般小圆圈(circle)或圆点(point)表示
- $E(G)$ 是边(edges)的有限集合, 边是顶点的无序对(unordered pairs)或有序对(ordered pairs). **无序边**用线段(segment)表示, 记为 (v_i, v_j) , 满足 $(v_i, v_j) = (v_j, v_i)$; **有序边**用弧(arc)(带箭头的线段)表示, 记为 $\langle v_i, v_j \rangle$, v_i 是弧尾, v_j 是弧头, 且 $\langle v_i, v_j \rangle \neq \langle v_j, v_i \rangle$. 这里 v_i 和 v_j 互为**邻接(adjacent)**.

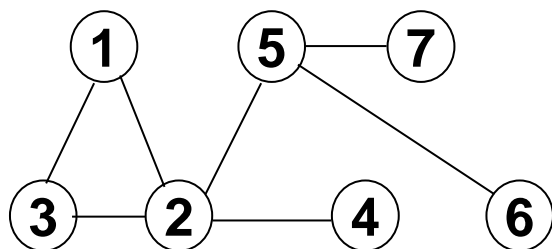
无序(无向)边 (v_i, v_j) : 

有序(有向)边 $\langle v_i, v_j \rangle$: 弧尾弧头

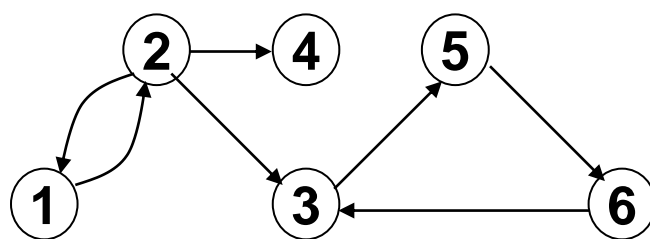
If the pairs are unordered, then G is called an **undirected graph**(无向图)

If the pairs are ordered, then G is called a **directed graph**(**di-graph**/有向图)

例



无向图G1



有向图G2

$V(G1) = \{ 1, 2, 3, 4, 5, 6, 7 \}$

$E(G1) = \{ (1,2), (1,3), (2,3), (2,4), (2,5), (5,6), (5,7) \}$

$V(G2) = \{ 1, 2, 3, 4, 5, 6 \}$

$E(G2) = \{ \langle 1,2 \rangle, \langle 2,1 \rangle, \langle 2,3 \rangle, \langle 2,4 \rangle, \langle 3,5 \rangle, \langle 5,6 \rangle, \langle 6,3 \rangle \}$

■ Definitions and Examples

无向完备图/完全图 n 个顶点的无向图最大边数是 $n(n-1)/2$

有向完备图 n 个顶点的有向图最大边数是 $n(n-1)$

权 与图中的边(线段或弧)相关的数[网 带权的图]

子图 如果图 $G(V, E)$ 和图 $G'(V', E')$, 且满足 $V' \subseteq V$ 和 $E' \subseteq E$, 则称 G' 为 G 的子图

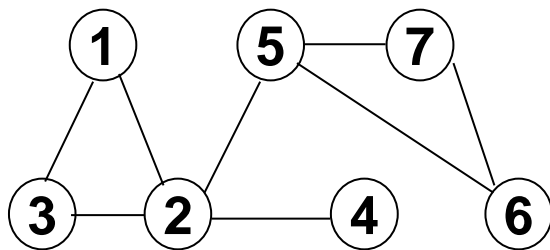
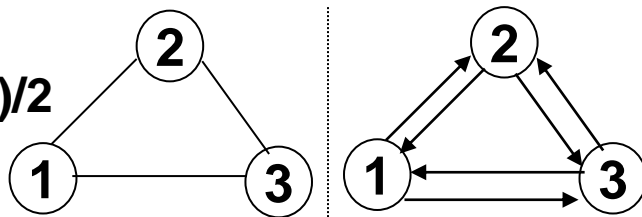
顶点的度 无向图中, 顶点的度为与每个顶点相关联的边数

有向图中, 顶点的度是其入度与出度之和

入度 以该顶点为头的弧的数目(incoming edges)

出度 以该顶点为尾的弧的数目(outgoing edges)

路径(path) 一个图中允许可以有多条路径, 一条路径是若干顶点的序列 $\text{Path} = (V_{i1}, V_{i2}, \dots, V_{in})$, 满足 $(V_{ij-1}, V_{ij}) \in E$ 或 $\langle V_{ij-1}, V_{ij} \rangle \in E$ (其中 i 是指某条路径, $1 < j \leq n$)



无向图 $G1'$

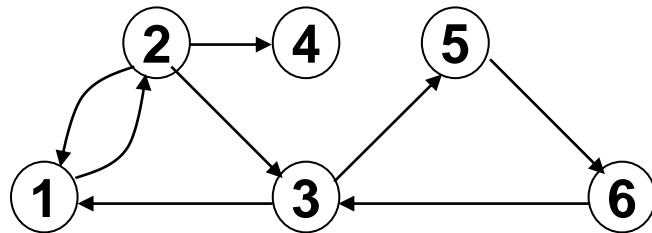
路径: 1, 2, 5, 7, 6, 5, 2, 3

路径长度: 7

简单路径: 1, 2, 5, 7, 6

回路: 1, 2, 5, 7, 6, 5, 2, 1

简单回路: 1, 2, 3, 1



有向图 $G2'$

路径: 1, 2, 3, 5, 6, 3

路径长度: 5

简单路径: 1, 2, 3, 5

回路: 1, 2, 3, 5, 6, 3, 1

简单回路: 3, 5, 6, 3

▪ Definitions and Examples

路径长度 沿路径边的数目或沿路径各边权值的和

回路/环(cycle) 第一个顶点和最后一个顶点相同的路径

简单路径 序列中顶点不重复出现的路径

简单回路 除了第一个顶点和最后一个顶点外, 其余顶点不重复出现的回路

连通(connected) 从顶点 V_i 到顶点 V_j 有一条路径, 则说 V_i 和 V_j 是连通的

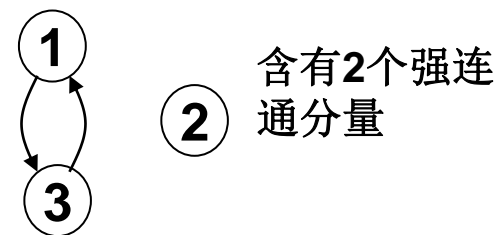
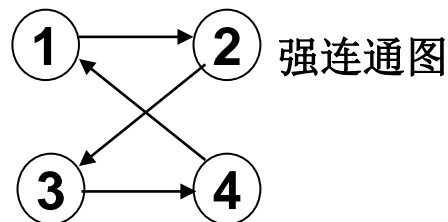
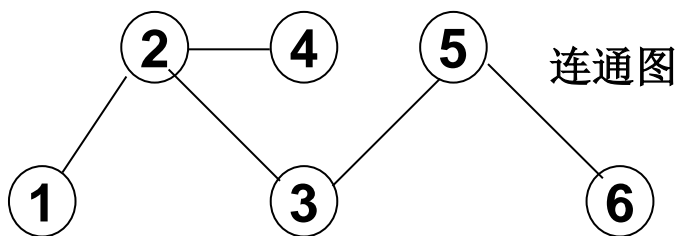
连通图(connected graph) 无向图中任意两个顶点都是连通的叫~, 否则称为**非连通图**

连通分量(connected component) 非连通图的每一个连通部分叫~

注: 这里的连通图、非连通图、连通分量均是针对无向图而言. 因此, 任何连通图的连通分量只有1个, 即是其自身, 非连通图有多个连通分量

强连通(strongly-connected) 有向图中, 从 V_i 到 V_j 和从 V_j 到 V_i 都存在路径, 则说 V_i 和 V_j 是强连通的

强连通图(strongly-connect graph) 有向图中, 如果对每一对 $V_i, V_j \in V, V_i \neq V_j$, 从 V_i 到 V_j 和从 V_j 到 V_i 都存在路径, 则称 G 是~. 否则称为**非强连通图**. 有向图的极大强连通子图称为**强连通分量**. 显然, 强连通图只有一个强连通分量, 即是其自身, 非强连通图有多个强连通分量

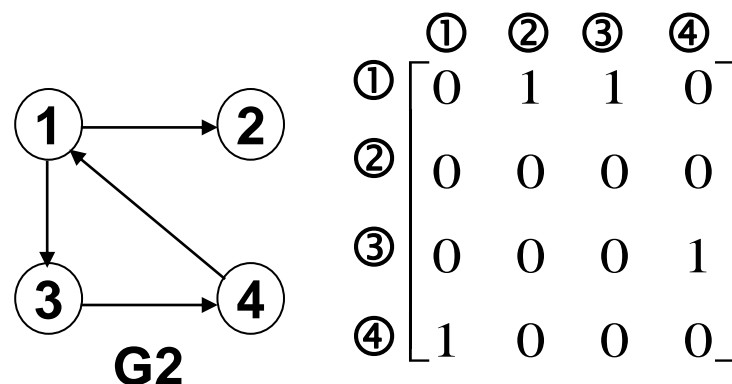
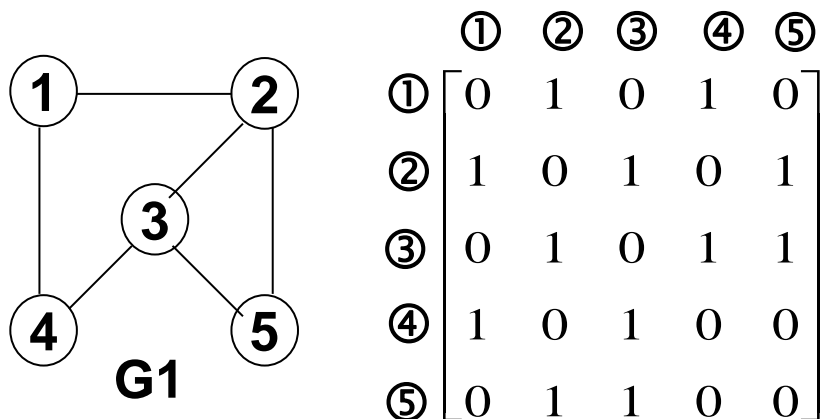


Computer Representation --- 图有2种常用的表示方法即存储结构

1. 邻接矩阵(adjacency matrix) -- a 2D array of size $n \times n$

定义 设 $G = (V, E)$ 是有 $n \geq 1$ 个顶点的图, G 的邻接矩阵 A 是具有以下性质的 n 阶方阵:

$$A[i, j] = \begin{cases} 1, & \text{若 } (v_i, v_j) \text{ 或 } \langle v_i, v_j \rangle \in E(G) \\ 0, & \text{其它} \end{cases}$$



网的邻接矩阵可定义为: $A[i, j] = \begin{cases} \omega_{ij}, & \text{若 } (v_i, v_j) \text{ 或 } \langle v_i, v_j \rangle \in E(G) \\ 0, & \text{其它} \end{cases}$

例: P162/Fig.7.9

其中: ω_{ij} 是边 (v_i, v_j) 或边 $\langle v_i, v_j \rangle$ 上的权值; 0 也可用 ∞ 表示

特点 • n 个顶点需存储空间为 n^2

- 无向图中顶点 V_i 的度 $D(V_i)$ 是邻接矩阵 A 中第 i 行元素之和
- 有向图中顶点 V_i 的出度是 A 中第 i 行元素之和;
顶点 V_i 的入度是 A 中第 i 列元素之和.

■ Computer Representation --- 图有2种常用的表示方法即存储结构

1. 邻接矩阵(adjacency matrix) -- a 2D array of size $n*n$

数据
结构

```
#define MaxVertex ?
typedef char VertexType; //顶点类型
typedef int EdgeType; //边:1/0 或边上的权值 $w_{ij}$ 
typedef struct graph {
    int n, e; //图中的实际顶点数和边数
    EdgeType edge[MaxVertex][MaxVertex]; //邻接矩阵, 可看作边表
    VertexType vertex[MaxVertex]; //顶点表
} Graph;
```

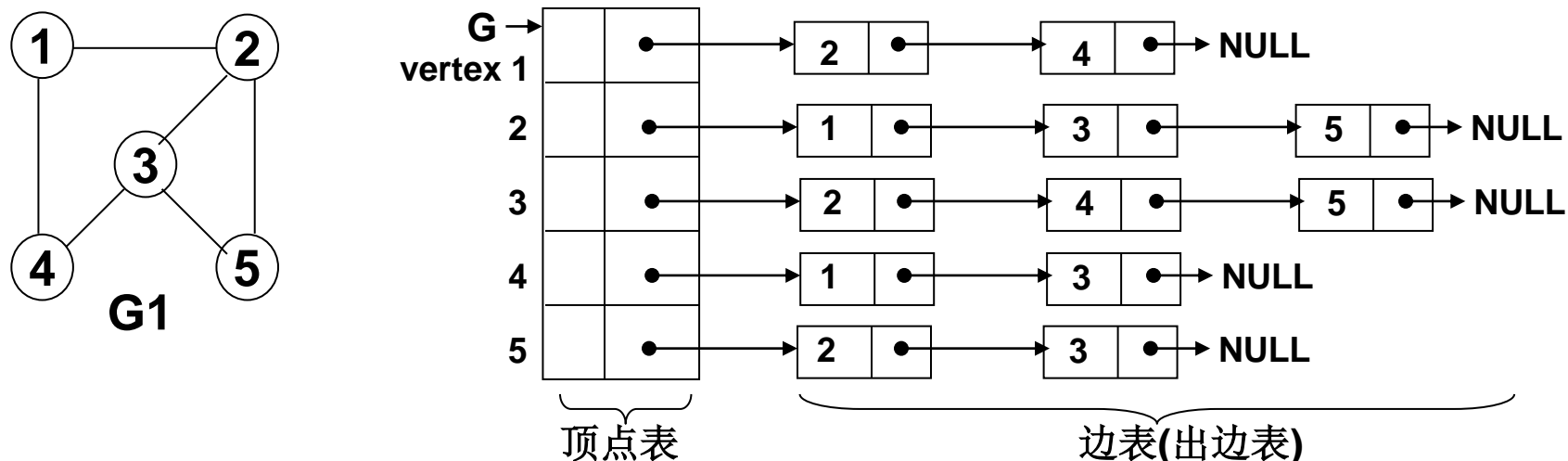
例: 写出利用邻接矩阵表示法建立一个无向网络的算法 P162/A.7.2

```
void CreateGraph(Graph *G)
{ scanf("%d&d", &G->n, &G->e); //读入顶点数和边数
  for ( i=0; i<G->n; i++ ) G->vertex[i] = getchar(); //读入顶点信息, 建立顶点表
  for ( i=0; i<G->n; i++ )
    for ( j=0; j<G->n; j++ ) G->edge[i][j] = 0; //邻接矩阵初始化
  for ( k=0; k<G->e; k++ ) { //读入e条边的权值, 建立邻接矩阵
    scanf("%d%d%d", &i, &j, &w);
    G->edge[i][j] = w; G->edge[j][i] = w; }
} //该算法时间复杂性:  $O(n+n^2+e) = O(n^2+e)$ 
```

Computer Representation --- 图有2种常用的表示方法即存储结构

2. 邻接表 (adjacency lists) -- an array of linked lists

定义 设 $G = (V, E)$ 是有 $n \geq 1$ 个顶点的图, G 的邻接表由**顶点表**和**边表**两部分组成. 其中**顶点表**是用顺序存储结构存储图 G 中的每个顶点; **边表**是为图中每个顶点建立的一个单链表(list), 第 i 个单链表中的结点表示依附于顶点 V_i 的边(有向图中指以 V_i 为尾的弧, 这时的边表又称为出边表); **顶点表**中的一个元素可看作是一个**边表**的表头结点.



在邻接表表示中, 每个边表对应于邻接矩阵的一行, 边表中结点的个数等于邻接矩阵的一行中非零元素的个数.

设图 $G=(V, E)$, 其中 G 有 n 个顶点, e 条边, 则:

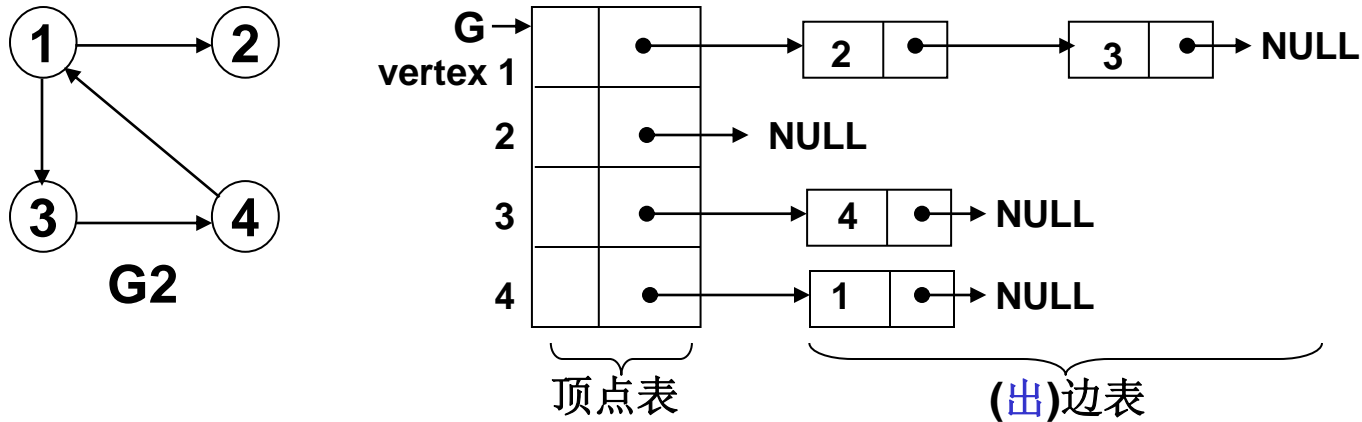
若 G 是无向图, 那么它的邻接表中则有 n 个顶点表结点和 $2e$ 个边表结点;

若 G 是有向图, 那么它的邻接表中则有 n 个顶点表结点和 e 个边表结点.

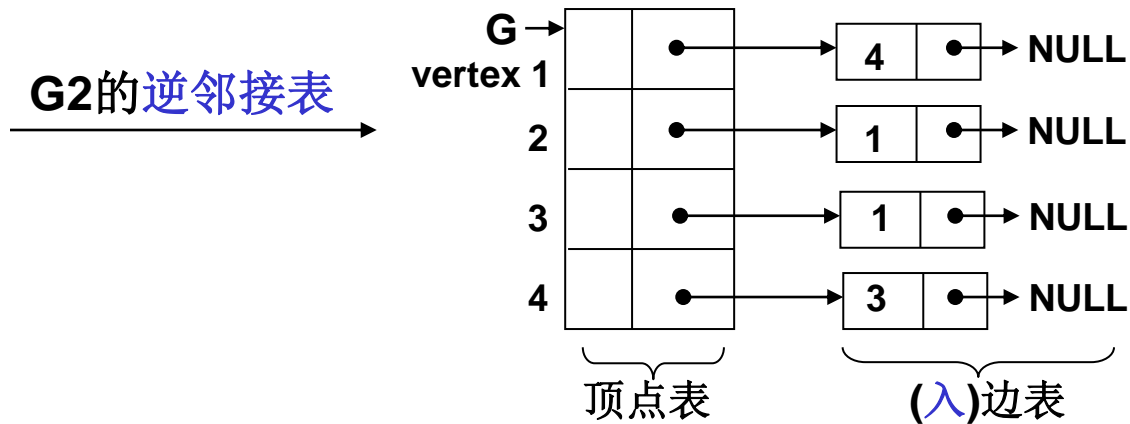
因此, 邻接表表示的空间复杂性为 $O(n+e)$.

Computer Representation --- 图有2种常用的表示方法即存储结构

2. 邻接表 (adjacency lists) -- an array of linked lists



对有向图还一种称为逆邻接表的表示法, 该方法为图中的每个顶点 V_i 建立一个入边表(即每个表结点是以 V_i 为头的弧).



▪ Computer Representation --- 图有2种常用的表示方法即存储结构

2. 邻接表 (adjacency lists) -- an array of linked lists

数据
结构

```
typedef struct edgenode { //边表结点
    VextexType adjvex; //邻接点域
    struct edgenode *nextedge; //指向下一个依附的顶点
    //若要表示边上的权, 则应增加一个数据域
} EdgeNode;
```

```
typedef struct vertexnode { //顶点表结点
    VertexType vertex; //顶点域
    EdgeNode *firstedge; //边表头指针
} VertexNode;
```

```
typedef VertexNode AdjList[MaxVertex]; //邻接表类型
```

```
typedef struct graph {
    int n, e; //图中的实际顶点数和边数
    AdjList adjlist; //邻接表
} Graph;
```

■ Computer Representation --- 图有2种常用的表示方法即存储结构

2. 邻接表 (adjacency lists) -- an array of linked lists

例: 写出利用邻接表表示法建立一个无向图的算法

```
void CreateGraph(Graph *G)
{ scanf("%d&d", &G->n, &G->e); //读入顶点数和边数
  for ( i=0; i<G->n; i++ ) { //建立顶点表
    G->adjlist[i].vertex = getchar();
    G->adjlist[i].firstedge = NULL; }
  for ( k=0; k<G->e; k++ ) { //建立边表
    scanf("%d%d", &i, &j); //读入边( $v_i, v_j$ )的顶点对序号
    s = (EdgeNode *)malloc(sizeof(EdgeNode)); //生成一个边表结点( $v_i, v_j$ )
    s->adjvex = j; //邻接点序号为j
    s->nextedge = G->adjlist[i].firstedge;
    G->adjlist[i].firstedge = s; //将新结点*s插入顶点 $v_i$ 的边表头部
    s = (EdgeNode *)malloc(sizeof(EdgeNode)); //生成一个边表结点( $v_j, v_i$ )
    s->adjvex = i; //邻接点序号为i
    s->nextedge = G->adjlist[j].firstedge;
    G->adjlist[j].firstedge = s; } //将新结点*s插入顶点 $v_j$ 的边表头部
  //对有向图仅需生成一个邻接序号为j的边表结点, 所以上述后四行去掉即可
} //该算法时间复杂性:  $O(n+e)$ 
```

■ Computer Representation --- 图有2种常用的表示方法即存储结构

3. 图的存储结构的比较

邻接矩阵和邻接表是图的两种最常用(**most commonly used**)的存储结构.

The choice of the graph representation is situation specific. 例如:

邻接矩阵表示法易于实现和理解; 方便查询或删除图中的某一条边(**$O(1)$**).

邻接表表示法节省空间; 便于向图中增加一个新顶点.

从空间复杂性角度考虑而言:

若图G有 $e \ll n^2$ (边数 e 远小于顶点数 n^2) [此类图称作**稀疏图(sparse graph)**], 则显然用邻接表表示比用邻接矩阵表示节省存储空间;

若 $e \approx n^2$ (边数 e 接近于顶点数 n^2 : 即对无向图 $e \approx n(n-1)/2$, 对有向图 $e \approx n(n-1)$) [此类图称作**稠密图(dense graph)**], 考虑到邻接表中的附加指针域, 则邻接矩阵表示更合适.

■ Graph Traversal

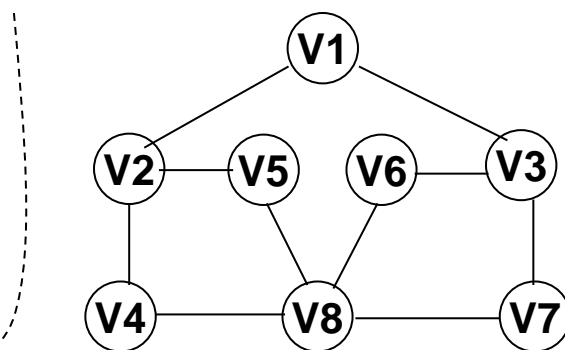
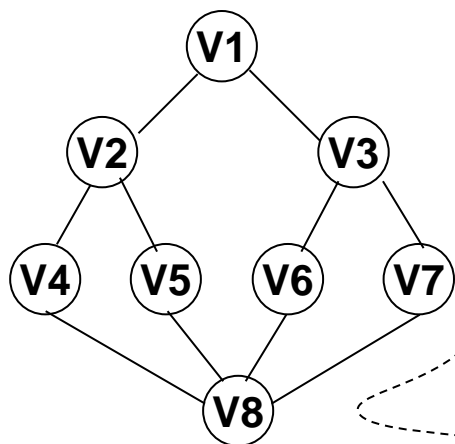
1. Depth-first traversal/search: DFT/DFS

思想 (be roughly analogous to preorder traversal of a tree)

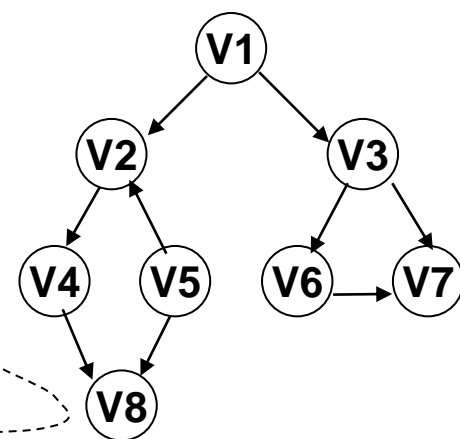
从图的某一顶点 V_i 出发, 访问此顶点; 然后依次从 V_i 的未被访问的一个邻接点出发, **深度优先**遍历图, 直至图中所有和 V_i 相通的顶点都被访问到; 若此时图中尚有顶点未被访问, 则另选图中一个未被访问的顶点作起点, 重复上述过程, 直至图中所有顶点都被访问为止. //

Suppose that the traversal has just visited a vertex v , and let w_1, w_2, \dots, w_k be the vertices adjacent to v . Then we shall next visit w_1 and keep w_2, \dots, w_k waiting. After visiting w_1 , we **traverse all the vertices to which it is adjacent before returning** to traverse w_2, \dots, w_k .

例子



$V1 \Rightarrow V2 \Rightarrow V4 \Rightarrow V8 \Rightarrow V5 \Rightarrow V6 \Rightarrow V3 \Rightarrow V7$



$V1 \Rightarrow V2 \Rightarrow V4 \Rightarrow V8 \Rightarrow V3 \Rightarrow V6 \Rightarrow V7 \Rightarrow \mathbf{V5}$

深度优先遍历次序:

$V1 \Rightarrow V2 \Rightarrow V4 \Rightarrow V8 \Rightarrow V5 \Rightarrow V6 \Rightarrow V3 \Rightarrow V7$

■ Graph Traversal

1. Depth-first traversal/search: DFT/DFS

算法描述

[outline]

P169/

A.7.4&

A.7.5

//DepthFirst: depth-first(DF) traversal of a graph

//Pre: The graph G has been created.

//Post: The function **Visit** has been performed at each vertex of G in DF

//Uses: Function **Traverse** produces the recursive DF order

void **DepthFirstSearch**(Graph G, void (*Visit)(Vertex)) {

 Boolean visited[MaxVertex];

 for (all v in G) visited[v] = FALSE;

 for (all v in G) if (!visited[v]) **Traverse**(v, Visit); }

//Traverse: recursive traversal of a graph

//Pre: v is the vertex of the graph G.

//Post: The **DepthFirst** traversal, using function **Visit** has been

//completed for v and for all vertices adjacent to v.

void **Traverse**(Vertex v, void (*Visit)(Vertex)) {

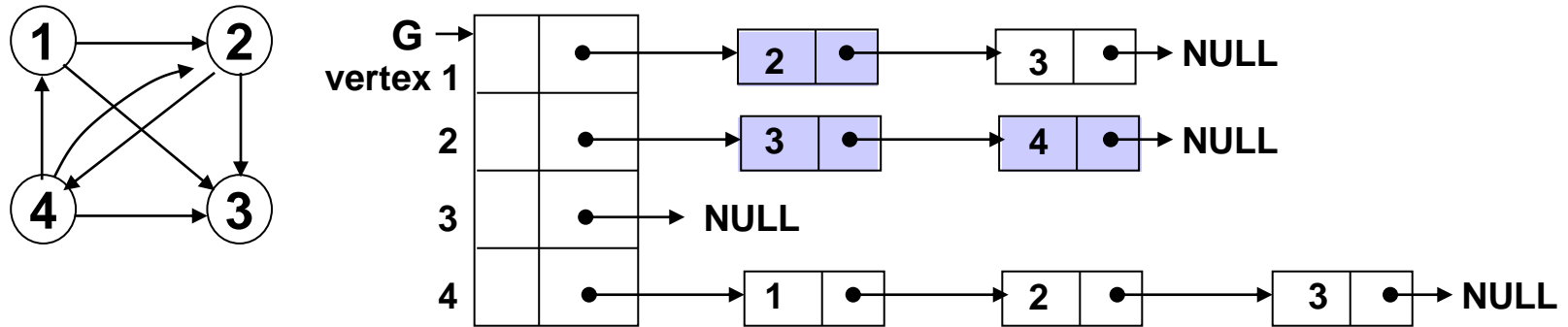
 visited[v] = TRUE; **Visit**(v);

 for (all w adjacent to v) if (!visited[w]) **Traverse**(w, Visit); }

■ Graph Traversal

1. Depth-first traversal/search: DFT/DFS

根据算法描述再看一例(假设图G的存储结构为邻接表)



深度优先遍历次序: $V1 \Rightarrow V2 \Rightarrow V3 \Rightarrow V4$

深度优先遍历的应用:

利用深度优先遍历, 还可以①查找路径②判断图中是否有回路③拓扑排序 等.

Path Finding -- to find a path between two given vertex u and z

- Call DFS(G, u) with u as the start vertex
- Use a stack S to keep track of the path between the start vertex and the current vertex
- As soon as destination vertex z is encountered, return the path as the contents of the stack

Detecting cycle in a graph – 参见exercise

Topological Sorting: 见后”图的应用”

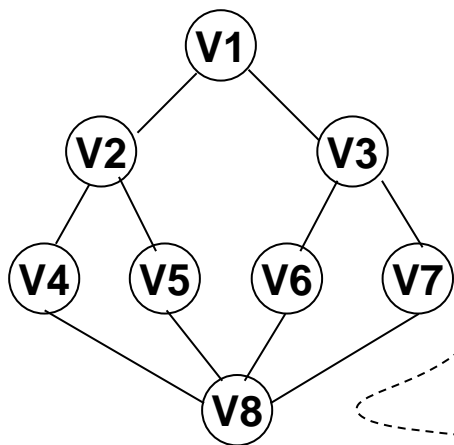
■ Graph Traversal

2. Breadth-first traversal/search: BFT/BFS

思想 (be roughly analogous to level-by-level traversal of a tree)

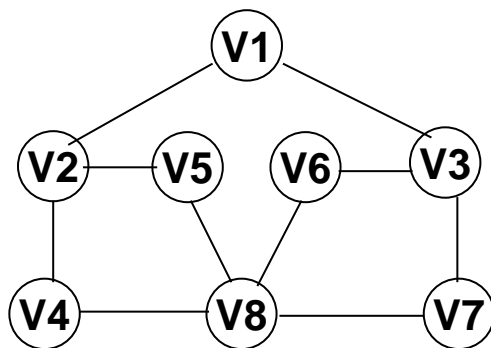
从图的某一顶点 V_0 出发, 访问此顶点后, **依次**访问 V_0 的各个未曾访问过的邻接点; 然后分别从这些邻接点出发, **同上依次**遍历图, 直至图中所有已被访问的顶点的邻接点都被访问到; 若此时图中尚有顶点未被访问, 则另选图中一个未被访问的顶点作起点, 重复上述过程, 直至图中所有顶点都被访问为止. // Suppose that the traversal has just visited a vertex v , then it **next visits** all the vertices adjacent to v , putting the vertices adjacent to these **in a waiting list** to be traversed after all vertices adjacent to v have been visited.

例子

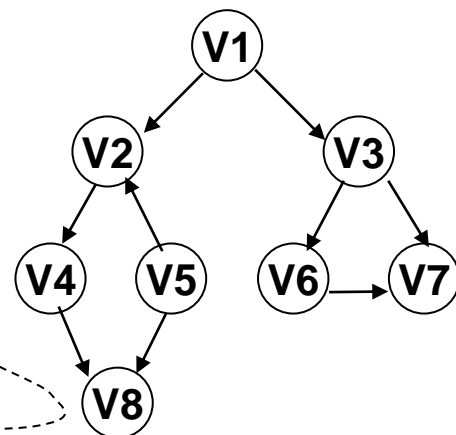


广度优先遍历次序:

$V1 \Rightarrow V2 \Rightarrow V3 \Rightarrow V4 \Rightarrow V5 \Rightarrow V6 \Rightarrow V7 \Rightarrow V8$



$V1 \Rightarrow V2 \Rightarrow V3 \Rightarrow V4 \Rightarrow V5 \Rightarrow V6 \Rightarrow V7 \Rightarrow V8$



$V1 \Rightarrow V2 \Rightarrow V3 \Rightarrow V4 \Rightarrow V6 \Rightarrow V7 \Rightarrow V8 \Rightarrow V5$

▪ Graph Traversal

2. Breadth-first traversal/search: BFT/BFS

算法描述

[outline]

P170/

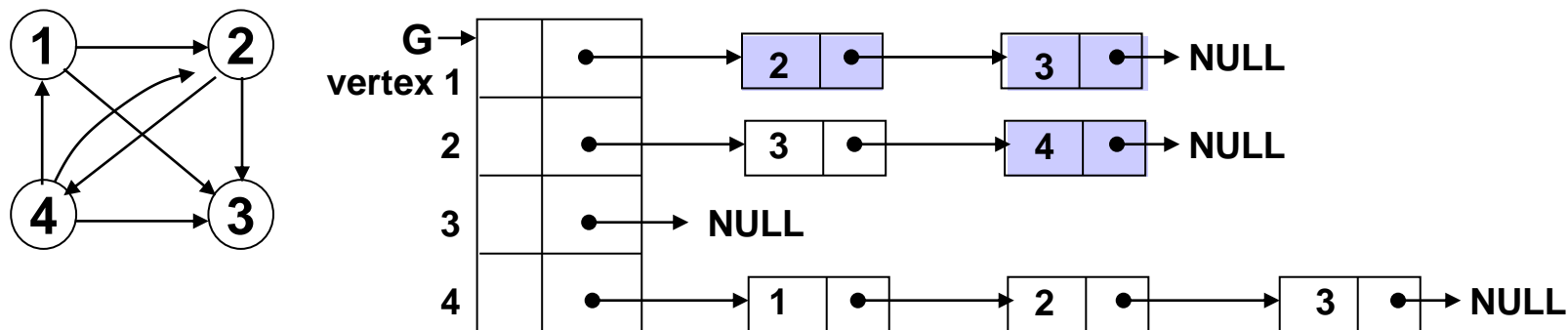
A.7.6

```
//BreadthFirst: breadth-first(BF) traversal of a graph
//Pre: The graph G has been created.
//Post: The function Visit has been performed at each vertex of G,
//where the vertices are chosen in breadth-first order.
void BreadthFirstSearch(Graph G, void (*Visit)(Vertex)) {
    Queue Q; //QueueEntry defined to be Vertex
    Boolean visited[MaxVertex];
    for (all v in G) visited[v] = FALSE;
    InitQueue(Q);
    for (all v in G)
        if (!visited[v]) {
            visited[v] = TRUE; Visit(v);
            EnQueue(Q, v);
            while(!QueueEmpty(Q)) {
                u = DeQueue(Q); //队首元素出队列并赋值给u
                for (all w adjacent to u)
                    if (!visited[w]) {
                        visited[w] = TRUE; Visit(w); EnQueue(Q, w); } } } }
```


▪ Graph Traversal

2. Breadth-first traversal/search: BFT/BFS

根据算法描述再看一例(假设图G的存储结构为邻接表)



广度优先遍历次序: **V1 \Rightarrow V2 \Rightarrow V3 \Rightarrow V4**

广度优先遍历的应用:

①搜索引擎的爬虫 / **Crawlers in Search Engines**

The idea is to start from source page and follow all links from source and keep doing same.

②搜索社交网络 / **Social Networking**

To find people within a given distance 'k' from a person using BFS in social networks.

③对等网络 / **Peer to Peer Networks**

BFS is used to find all neighbor nodes, like BitTorrent in P2P Networks.

④**GPS**导航系统(Find all neighboring locations) & 网络广播协议(a packet to reach all nodes)

⑤最小生成树 或 最短路径 (见后"图的应用")

... ..

▪ Application using graphs themselves as data structures

1. Minimum Spanning Tree / 最小生成树(MST)

生成树的定义 P159, P170

在图论中通常将一个**连通的且无回路的无向图**定义为**树**. 因为没有指定哪一个顶点作根, 所以这种图又称为**自由树(free tree)**. 有一些图本身不是树, 但其子图是树. 一个图可能有許多子图是树, 其中很重要的一类是**生成树**.

Given a connected and undirected graph, **a spanning tree** of that graph is a subgraph that is a tree and connects all the vertices together.

生成树(spanning tree)是一个**连通图的包含该图中的所有顶点的极小连通子图**. 所谓极小是指边数最少, 即若在生成树中去掉任何一条边则会使之变为非连通图, 若在生成树上再任意添加一条边则必定出现回路.

- ⇒ • 生成树的顶点个数和相应的连通图的顶点个数相同
- 一个连通图至少有一颗生成树
- 一棵有 n 个顶点的生成树中有且仅有 $n-1$ 条边

▪ Application using graphs themselves as data structures

1. Minimum Spanning Tree / 最小生成树(MST)

生成树的生成过程

设 $E(G)$ 为连通图 G 中所有边的集合, 从图 G 中任一顶点出发对其进行遍历, 其结果会把边集 $E(G)$ 分成两个集合 $T(G)$ 和 $B(G)$, 其中 $T(G)$ 是遍历过程中历经的边的集合, $B(G)$ 是剩余的边的集合. 则 $T(G)$ 和连通图 G 中所有顶点一起构成 G 的极小连通子图, 按生成树的定义, 它是连通图 G 的一棵生成树, 并且称由DFS得到的生成树为深度优先生成树, 由BFS得到的生成树为广度优先生成树.

生成森林是指一个非连通图的每个连通分量的生成树组成的非连通图.

对于非连通图, 则需从多个顶点出发进行遍历. 每次从一个新的起始点出发, 遍历过程中所得到的顶点访问序列恰为其相应连通分量中的顶点集. 这些连通分量的生成树组成非连通图的生成森林, 同样可分别称为深度优先生成森林和广度优先生成森林.

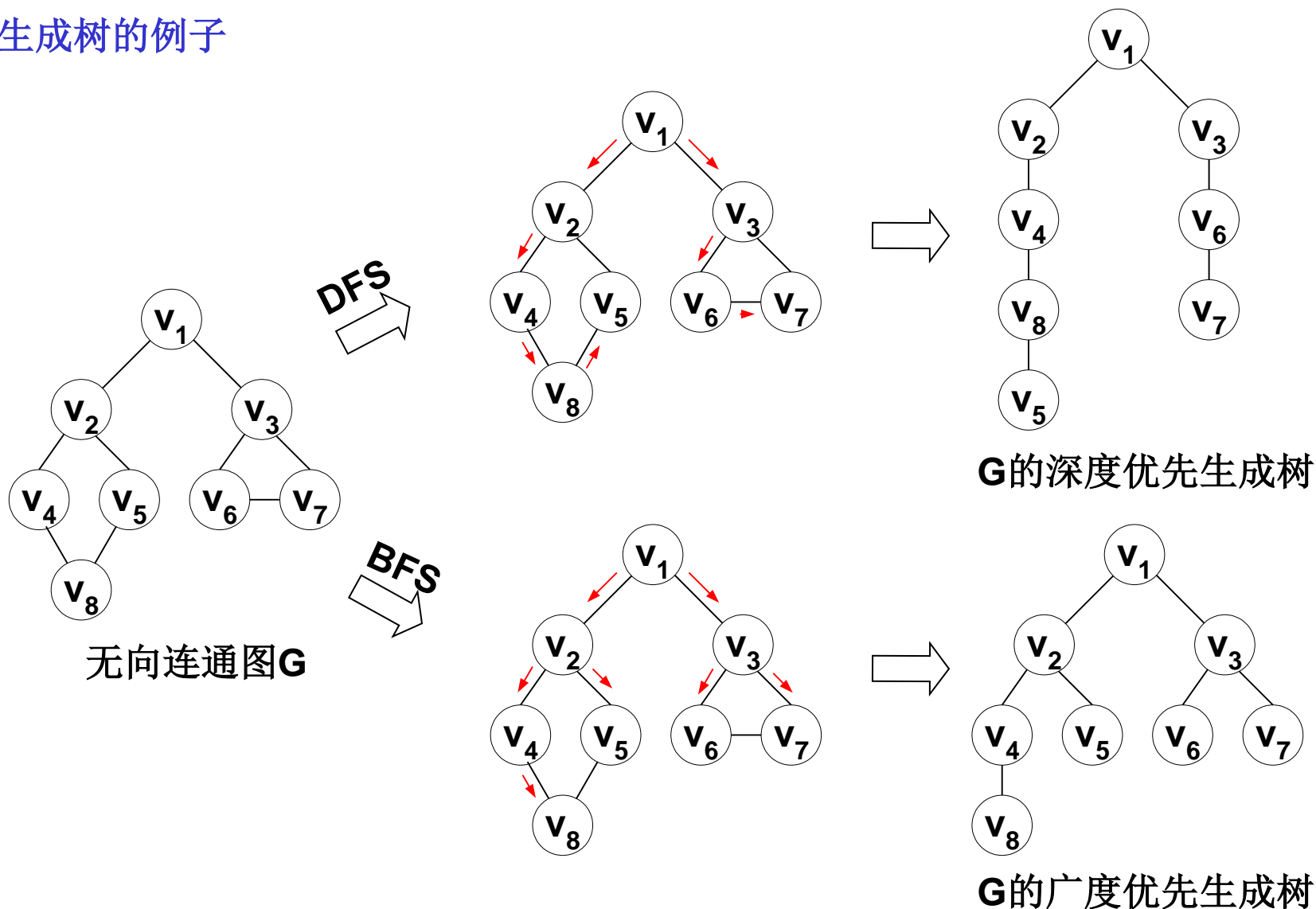
生成树(森林)的例子

可以看出: 连通图的生成树不唯一(从不同顶点出发经过不同可得到不同的生成树)

Application using graphs themselves as data structures

1. Minimum Spanning Tree / 最小生成树(MST)

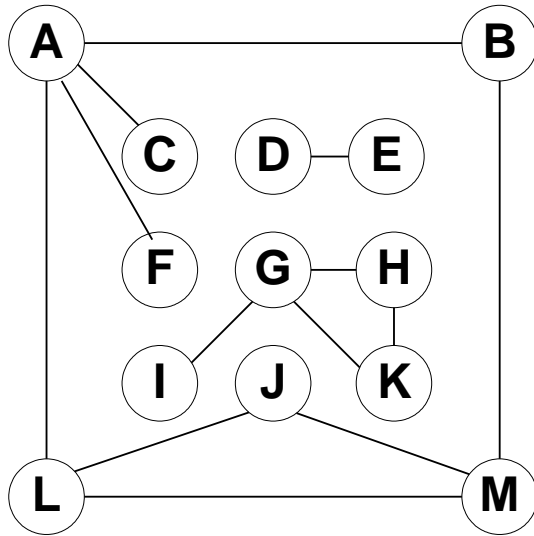
生成树的例子



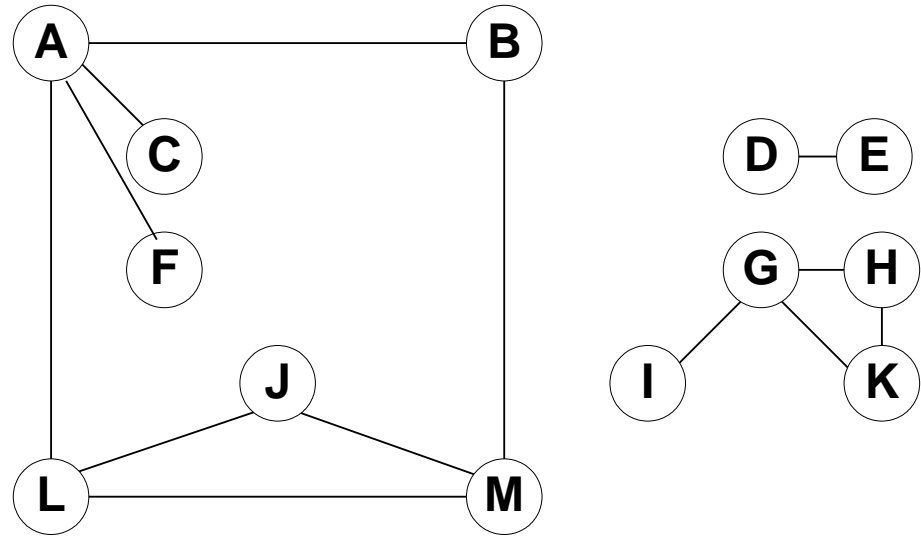
Application using graphs themselves as data structures

1. Minimum Spanning Tree / 最小生成树(MST)

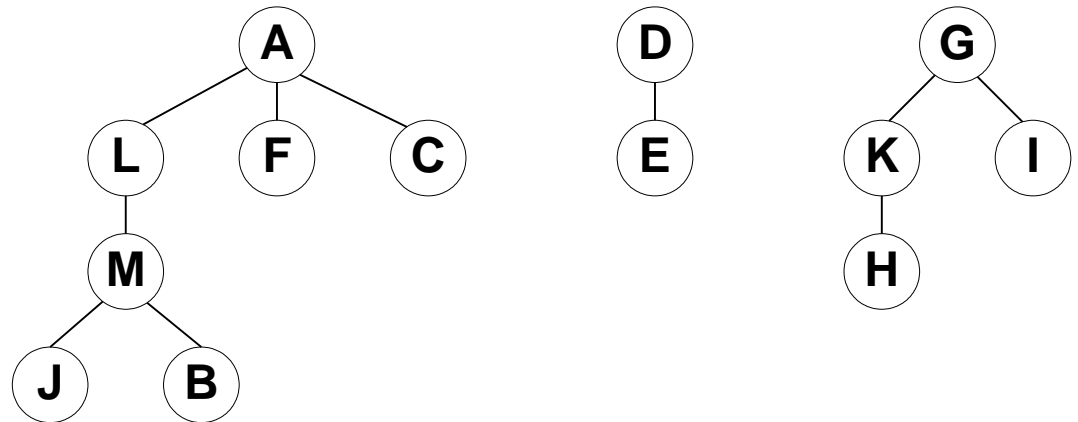
生成森林的例子



非连通图G



G的三个连通分量



G的深度优先生成森林

▪ Application using graphs themselves as data structures

1. Minimum Spanning Tree / 最小生成树(MST)

最小生成树问题

问题的提出: 要在 n 个城市间建立通信联络网的问题(and other diverse applications)

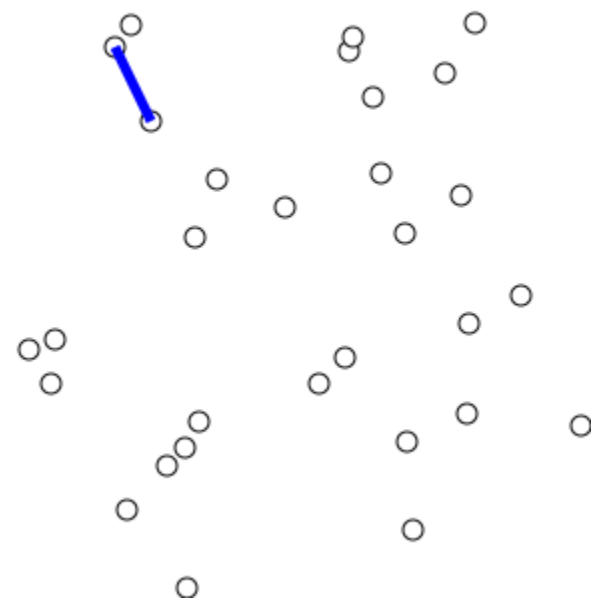
- 顶点 -- 表示城市
- 边 & 权值 -- 城市间的通信线路 & 建立通信线路所需花费代价(cost)
- 希望找到一棵生成树, 它的每条边上的权值之和(即建立该通信网所需花费的总代价)最小 -- 最小(代价)生成树问题

问题分析:

- n 个城市间, 最多可设置 $n(n-1)/2$ 条线路
 - n 个城市间建立通信网, 只需 $n-1$ 条线路
- ⇒ 如何在可能的线路中选择 $n-1$ 条, 能把所有城市(顶点)均连起来, 且总耗费(各边权值之和)最小

问题的解决办法: 贪心策略

- 每一步都找出当前局部的最优解
- 且 这个局部的最优解是最终全局最优解的一部分



最小生成树的基本性质

设 $G = (V, E)$ 是连通图, U 是顶点集 V 的一个真子集. 若 (u,v) 是一条具有最小权值(代价)的边, 其中 $u \in U, v \in V-U$, 则必存在一棵包含边 (u,v) 的最小生成树.

▪ Application using graphs themselves as data structures

1. Minimum Spanning Tree / 最小生成树(MST)

最小生成树的生成方法 -- **Prim/普里姆算法 & Kruskal/克鲁斯卡尔算法**

①**Prim算法**: 基本思路如下:

设 $G=(V, E)$ 是连通图, $T=(U, TE)$ 是MST, U 和 TE 分别是 G 的MST的顶点集和边集

- 初始令 $U = \{ u_0 \}$, ($u_0 \in V$), $TE = \phi$
 - 在所有 $u \in U$, $v \in V-U$ 的边 $(u,v) \in E$ 中, 找一条代价最小的边 (u_0, v_0)
- U : the Set of the vertices already included in MST
 $V-U$: the Set of the vertices not yet included
- Greedy criterion
↑
↓
connects the two sets

注: 设 U 中已有 k 个顶点, 则 $V-U$ 中有 $n-k$ 个顶点, 所以这时可能的边数是 $k \times (n-k)$, 从这么多边中选择一条代价最小的边显然低效. 事实上, 对于**每个**从 $v \in V-U$ 顶点到**各个** $u \in U$ 顶点的多条边中, 只有最短的那一条才**有可能是**代价最小的边. 因此**只需保留 $V-U$ 中 $n-k$ 个顶点所关联的最短边作为选择一条代价最小的边的候选集.**

- 将 (u_0, v_0) 并入集合 TE , 同时 v_0 并入 U
- 重复 $n-1$ 步上述操作直至 $U = V$ 为止, 则 $T = (V, TE)$ 为 G 的最小生成树

Prim方法生成MST的一个例子/算法过程

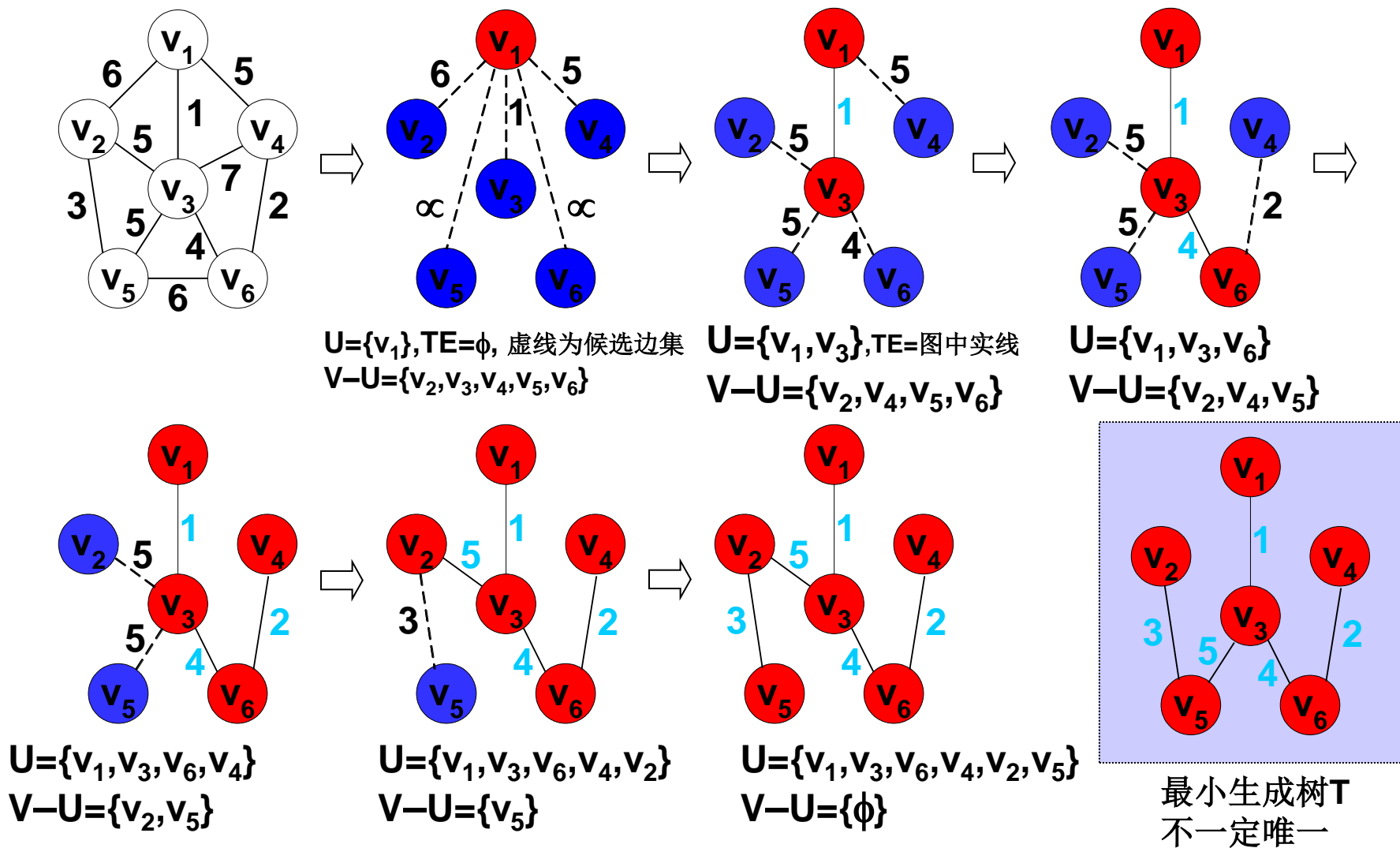
连通网的最小生成树不一定是唯一的, 但它们的权值和相等
或者说, 连通网中有相同权值边时(这时可任选其中一条)能生成不同的MST

Application using graphs themselves as data structures

1. Minimum Spanning Tree / 最小生成树(MST)

最小生成树的生成方法 -- **Prim/普里姆算法** & **Kruskal/克鲁斯卡尔算法**

Prim方法生成MST的一个例子



▪ Application using graphs themselves as data structures

1. Minimum Spanning Tree / 最小生成树(MST)

最小生成树的生成方法 -- **Prim/普里姆算法 & Kruskal/克鲁斯卡尔算法**

Prim算法描述:

```
struct { int adjvex; int lowcost; } closeedge[MaxVertex]; //构造MST所需的辅助数组(边集).  
//其中, adjvex表示与closeedge某个顶点i关联的在T之外(=V-U)的一个顶点(候选边);  
//lowcost表示边(i, adjvex)上的权值, 若lowcost=0, 则表示顶点adjvex已在T中(=U)  
void PrimMST( Graph *G, int u ) { //图G为邻接矩阵, 从顶点u出发, 构造图G的MST  
    closeedge[u].lowcost = 0; //初始化, U = {u}  
    for ( i = 0; i < G->n; i++ ) {  
        if ( i != u ) { //对V-U的顶点i, 初始化closeedge[i]  
            closeedge[i].adjvex = u;  
            closeedge[i].lowcost = G->edge[u][i]; } }  
    for ( e = 1; e < G->n-1; e++ ) { //找出n-1条最短边  
        min = INFINITY; //准备从候选边中找出当前最短边  
        for ( i = 0; i < G->n; i++ ) {  
            if ( closeedge[i].lowcost != 0 && closeedge[i].lowcost < min)  
                { min = closeedge[i].lowcost; v = i; }  
        printf(closeedge[v].adjvex, v); //输出生成树的当前最短边(u, v)  
        closeedge[v].lowcost = 0; //将顶点v并入U集合  
        for ( i = 0; i < G->n; i++ ) { //顶点v并入U集合后, 更新closeedge[i]  
            if ( (lowcost[i] != 0) && (G->edge[v][i] < closeedge[i].lowcost) ) {  
                closeedge[i].lowcost = G->edge[v][i];  
                closeedge[i].adjvex = v; } } } } //该算法的时间复杂度为O(n^2).
```

▪ Application using graphs themselves as data structures

1. Minimum Spanning Tree / 最小生成树(MST)

最小生成树的生成方法 -- **Prim/普里姆算法 & Kruskal/克鲁斯卡尔算法**

② **Kruskal算法**: 基本思路如下:

设 $G=(V, E)$ 是连通图, $T=(U, TE)$ 是MST

- 令最小生成树 T 的初始状态为只有 n 个顶点而无边的非连通图 $T=(V, \{\phi\})$, 图中每个顶点自成一个连通分量
- 在 E 中选取代价最小(Greedy criterion)的边, 若该边依附的2个顶点落在 T 中**不同的**连通分量上(即判断此时的生成树中是否会形成回路/环), 则将此边加入到 T 中; 否则, 舍去此边, 选取下一条代价最小的边
- 依此类推, 直至 T 中有 $n-1$ 条边为止(或 T 中所有顶点都在同一连通分量上为止)

Kruskal方法生成MST的一个例子

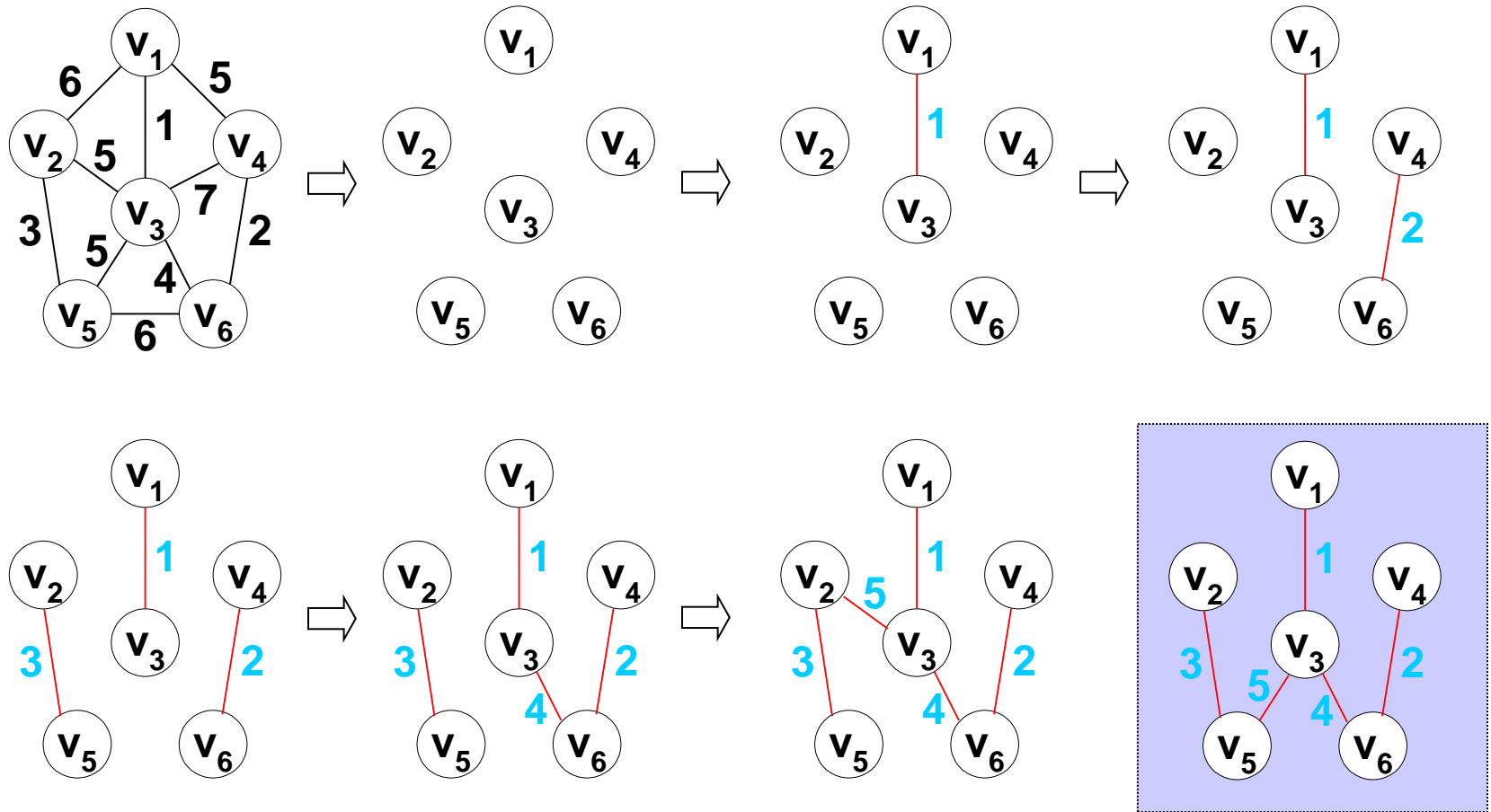
连通网的最小生成树不一定是唯一的, 但它们的权值和相等

Application using graphs themselves as data structures

1. Minimum Spanning Tree / 最小生成树(MST)

最小生成树的生成方法 -- **Prim/普里姆算法 & Kruskal/克鲁斯卡尔算法**

Kruskal方法生成**MST**的一个例子



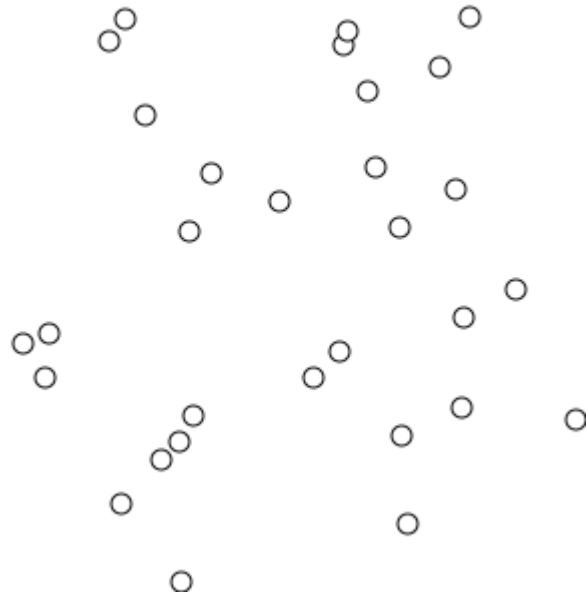
最小生成树
不一定唯一

▪ Application using graphs themselves as data structures

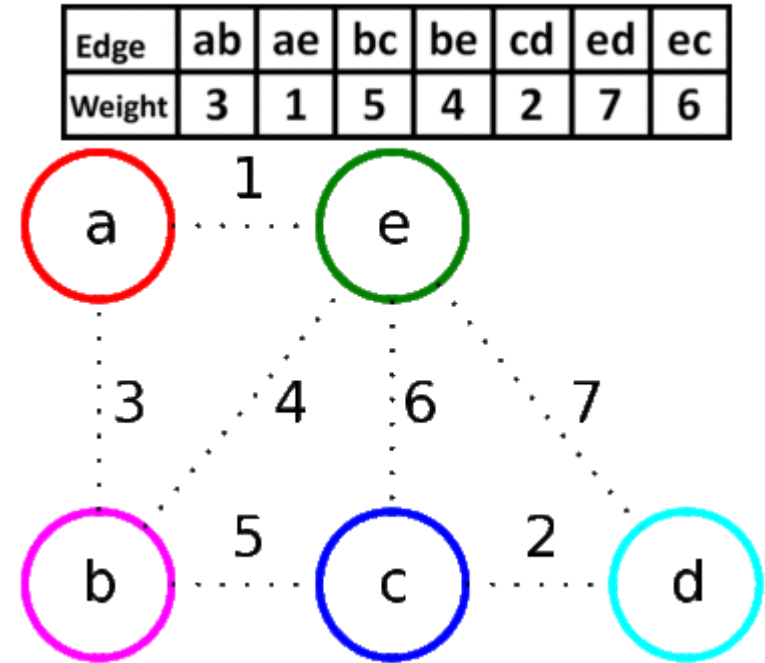
1. Minimum Spanning Tree / 最小生成树(MST)

最小生成树的生成方法 -- **Prim/普里姆算法 & Kruskal/克鲁斯卡尔算法**

Kruskal方法生成**MST**的一个例子 -- from Wikipedia



Kruskal Demo



MST of Kruskal

▪ Application using graphs themselves as data structures

1. Minimum Spanning Tree / 最小生成树(MST)

最小生成树的生成方法 -- **Prim/普里姆算法 & Kruskal/克鲁斯卡尔算法**

Kruskal算法描述:

```
typedef struct node { int u, v, w; } Edge; //边的类型定义, uvw表示边的起点终点和权值
void KruskalMST(Graph *G) { //假设G采用邻接矩阵作为数据结构, 其中顶点从0开始进行编号标识
    int vset[VertexNum]; //用于记录不同顶点集的辅助数组. 若顶点的数组值相同, 则它们属于同一个集合
    Edge e[EdgeNum]; //用于存储T所有边的辅助数组
    k = 0; //用于控制数组e的下标, 下标从0开始计算
    for ( i = 0; i < G->n; i++ ) { for ( j = 0; j < G->n; j++ ) {
        if ( G->edges[i][j] != 0 && G->edges[i][j] != INFINITY ) {
            e[k].u = i; e[k].v = j; e[k].w = G->edges[i][j]; k++; } } }
    HeapSort(Edge e); //使用效率高的排序方法按权值升序排序
    for ( i=0; i < G->n; i++) vset[i] = i; //初始化, 即G的n个顶点分别属于n个不同的集合
    j = 1; //用于控制T中的边数, 直至j=n-1
    k = 0; //用于控制数组e的下标
    while ( j < G->n ) {
        if ( vset[e[k].u] != vset[e[k].v] ) { //查集. 若当前最短边所依附的两个顶点分别属于两个不同的
            //顶点集合, 则把这条最短边加入到MST; 否则会形成回路
            j++;
            for ( i = 0; i < G->n; i++ ) if ( vset[i] == vset[e[k].v] ) vset[i] = vset[e[k].u]; //并集
            k++; } //准备加入下一条最短边
    }
}
```

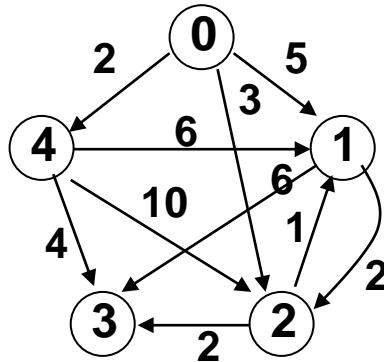
该算法的语句频度为: $n^2 + e \log_2 e + n$.

Application using graphs themselves as data structures

2. Shortest Path / 最短路径 P186

The Problem: 用带权的有向图(an edge-weighted digraph)表示一个交通运输网.

- 顶点 -- 表示城市
- 弧 -- 表示城市间的交通联系, 如 **airline routes**
- 权 -- 表示此线路的长度或沿此线路运输所花的时间或费用等
- 从任一个顶点 **s**(supposed v_0)(源点, **source**)出发, 沿图的边到达其它任意另一顶点 **t**(汇点, **sink**) 所经过的所有可能路径中, 各边上权值之和最小的一条路径 -- **最短路径问题**
(Goal: Find the shortest path from s to every other vertex in the given graph. \Leftarrow **SPT**)



A directed graph with weights



Shortest Path from v_0 to v_3

The **Brute force** method finding all possible paths between Source and Destination and then finding the minimum.



the **WORST** strategy

Simple Greedy Method:

At each node, choose the shortest outgoing path.



also **FAILS**



- x** v_0 goes via v_1 :11
- x** v_0 goes via v_1 and v_2 :7
- ✓** v_0 goes via v_2 :5
- x** v_0 goes via v_2 and v_1 :10
- x** v_0 goes via v_4 :6
- x** v_0 goes via v_4 and v_2 :14



- x** v_0 goes via v_4 :6

Application using graphs themselves as data structures

2. A Greedy Algorithm: Shortest Path / 最短路径

Method: 迪杰斯特拉(Dijkstra)算法(Only for single source and nonnegative weights)

基本思路

先把V分成两个集合: • **S** -- 已求出最短路径的顶点的集合

(Initially, V_0 is the only vertex in S)

• **T**(= $V-S$) -- 尚未确定最短路径的顶点集合

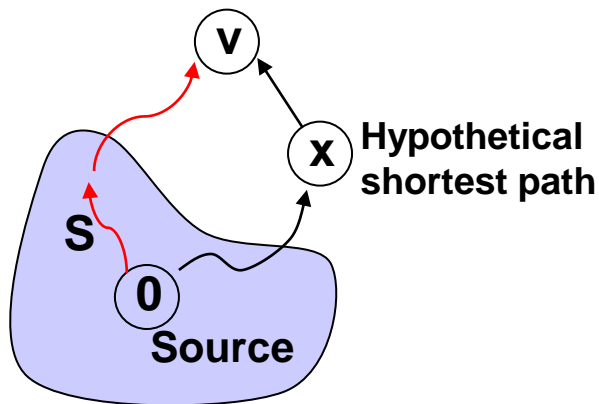
然后将T中顶点按最短路径递增的次序加入到S中, 其中需满足:

- 从源点 V_0 到S中各顶点的最短路径长度都不大于从 V_0 到T中任何顶点的最短路径长度(**greedy criterion**)

- 每个顶点对应一个距离值(**Distance table**, 用数组D来表示, 记作 $D[V_i]$), 且该值最小:
S中顶点 -- 从 V_0 到此顶点的最短路径长度;

T中顶点 -- 从 V_0 到此顶点 V_i 的最短路径长度, 可能会是以下2种情况之一:

①从 V_0 到 V_i 的直接路径权值 ②从 V_0 经S中顶点到 V_i 的路径权值之和(反证法可证之).



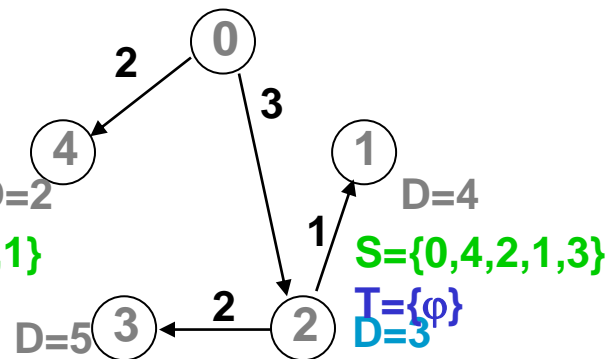
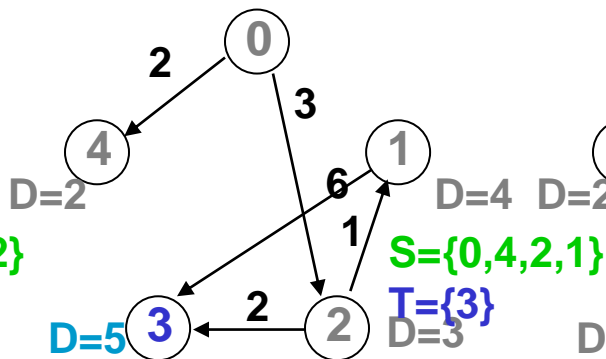
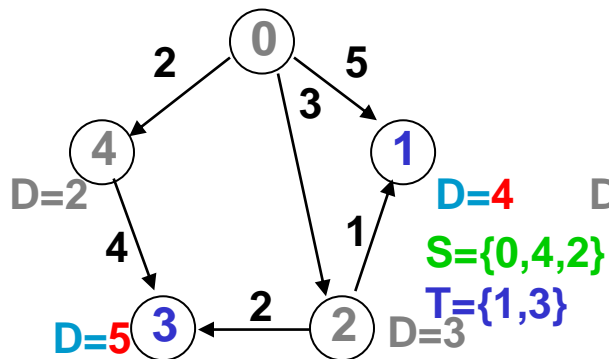
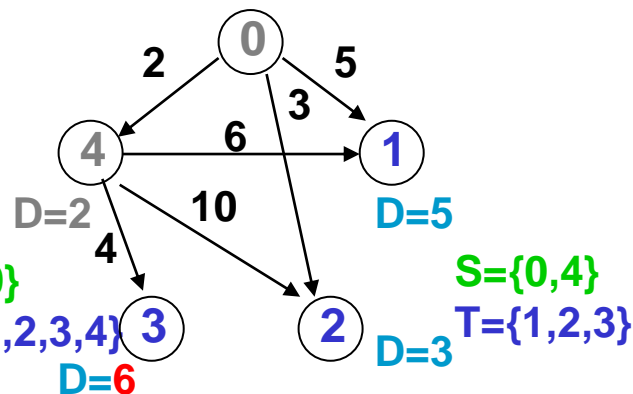
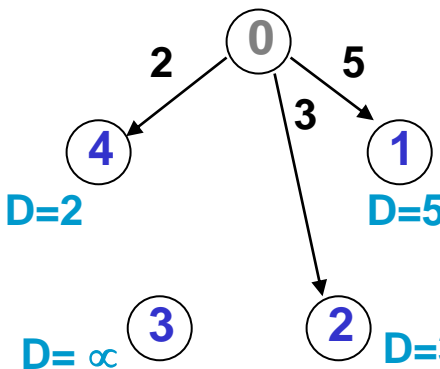
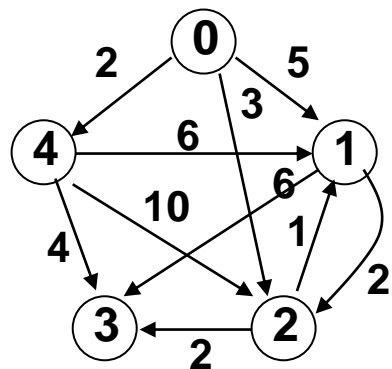
For suppose that there were a shorter path from 0 to v. This path first leaves S to go to some vertex x, then goes on to v. But if this path is shorter than the colored path to v, then its initial segment from 0 to x is also shorter, so that the greedy criterion would have chosen x rather than v as the next vertex to add to S, for we would have had $D[x] < D[v]$.

Application using graphs themselves as data structures

2. A Greedy Algorithm: Shortest Path / 最短路径

Algorithm: 实现的步骤(Only for single source and nonnegative weights)

- 初始时令 $S = \{V_0\}$, $T = \{\text{其余顶点}\}$, T 中顶点对应的距离值 $D[i]=$
若存在 $\langle V_0, V_i \rangle$, 则为 $\langle V_0, V_i \rangle$ 弧上的权值; 若不存在 $\langle V_0, V_i \rangle$, 则为 ∞
- 从 T 中选取一个其距离值为最小的顶点 W , 加入 $S \Leftarrow D[w] = \text{Min}\{ D[k] \mid V_k \in T \}$
- 对 T 中顶点的距离值 D 进行修改: 从 V_0 到 $V_i (V_i \in T)$ 的距离值 $D[i]$, 若加进 W 作中间顶点
比不加 W 的路径还要短, 则修改此距离值 $D[i] \Leftarrow$ 若 $D[i] > D[w] + \omega(w, i)$, 则 $D[i] = D[w] + \omega(w, i)$
- 重复上述的两个步骤, 直到 S 中包含所有顶点, 即 $S = V$ 为止



- Application using graphs themselves as data structures

2. A Greedy Algorithm: Shortest Path / 最短路径

Description for Dijkstra algorithm: 参见P189/A.7.15

```
#include <limits.h>
```

```
#define INFINITY INT_MAX // value for  $\infty$ 
```

```
typedef int AdjacencyTable[MaxVertex][MaxVertex];
```

```
typedef int DistanceTable[MaxVertex];
```

```
int n; // number of vertices in the graph
```

```
AdjacencyTable cost; //describes the graph
```

```
DistanceTable D; // shortest distances from vertex 0
```

```
// Pre: A directed graph is given which has n vertices by the weights given  
// in the table cost.
```

```
// Post: The function finds the shortest path from vertex 0 to each vertex  
// of the graph and returns the path that it finds in the array D.
```

```
void DijkstraSPT(int n, AdjacencyTable cost, DistanceTable D) {  
    Boolean final[MaxVertex]; // Has the distance from 0 to v been found?  
                                // final[v] is true iff v is in the set S.
```

```
    int i; // repetition count for the main loop
```

```
        // one distance is finalized on each pass through the loop
```

```
//... to be continued
```

- Application using graphs themselves as data structures

2. A Greedy Algorithm: Shortest Path / 最短路径

Description for Dijkstra algorithm: 参见P189/A.7.15

...

```
int w; // a vertex not yet added to the set S
int v; // vertex with minimum tentative distance in D[ ]
int min; // distance of v, equals D[v]
final[0] = TRUE; // Initialize with vertex 0 alone in the set S
D[0] = 0;
for (v = 1; v < n; v++) { final[v] = FALSE; D[v] = cost[0][v]; }
// start the main loop; add one vertex to S on each pass
for (i = 1; i < n; i++) {
    min = INFINITY; // Find the closest vertex to vertex 0
    for (w = 1; w < n; w++) if ( !final[w] )
        if ( D[w] < min ) { v = w; min = D[w]; }
    final[v] = TRUE; // Add v to the set S
    for (w = 1; w < n; w++) // Update the remaining distance in D
        if ( !final[w] )
            if ( min + cost[v][w] < D[w] ) D[w] = min + cost[v][w];
} // ends for i loop
}
```

算法分析: 主循环执行 $n-1$ 次, 内循环每趟也执行 $n-1$ 次 $\Rightarrow O(n^2)$

▪ Application using graphs themselves as data structures

3. Topological Sorting / 拓扑排序 P180

The Problem: 学生选修课程问题 或者 生产工序问题

- 顶点 -- 表示课程
- 弧 -- 表示先决条件, 若课程 i 是课程 j 的先决条件, 则图中有弧 $\langle i, j \rangle$
- 学生应按怎样的顺序学习这些课程, 才能合理地完成学业 -- 拓扑排序问题 →

两个名词

AOV网 用顶点表示活动、用弧表示活动间优先关系的有向图称为顶点表示活动的网(Activity On Vertex network). AOV网中不允许有回路(**DAG: Directed Acyclic Graph / 有向无环图**), 即某项活动不能以自己为先决条件.

拓扑排序 (Topological sorting) 把AOV网络中各顶点按照它们相互之间的优先关系排列成一个线性序列的过程叫~. 若 $\langle v_i, v_j \rangle$ 是DAG中一条边, 则 v_i 是 v_j 的前趋、 v_j 是 v_i 的后继(v_i 优于 v_j).

Fact: A DAG G has at least one vertex with in-degree 0 and one vertex with out-degree 0.

拓扑排序的思想 / 步骤 --无前趋的顶点优先 (vs. 无后继的顶点优先 -- 逆拓扑序列)

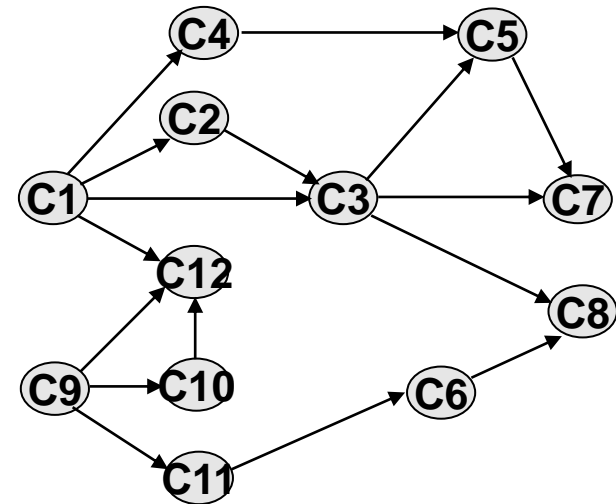
- 在有向图中选择一个没有前趋的顶点且输出之
- 从图中删除该顶点和所有以它为尾的弧
- 重复上述两步, 直至全部顶点均已输出; 或者当图中不存在无前趋的顶点为止

▪ Application using graphs themselves as data structures

3. Topological Sorting / 拓扑排序

An example: 学生选修课程

课程代号	课程名称	先修课
C1	程序设计基础	无
C2	离散数学	C1
C3	数据结构	C1, C2
C4	汇编语言	C1
C5	语言的设计和分析	C3, C4
C6	计算机原理	C11
C7	编译原理	C3, C5
C8	操作系统	C3, C6
C9	高等数学	无
C10	线性代数	C9
C11	普通物理	C9
C12	数值分析	C1, C9, C10



拓扑序列: C1 - C2 - C3 - C4 - C5 - C7 - C9 - C10 - C11 - C6 - C12 - C8

或: C9 - C10 - C11 - C6 - C1 - C12 - C4 - C2 - C3 - C5 - C7 - C8 或: ...

注: 1. 一个DAG的拓扑排序序列可能会有多个(若边带权重, 则不同序列可能会有不同的解释).

2. 拓扑排序 与 DFS 不同: 前者需在其访问邻接顶点之前先访问, 如上图, 对拓扑排序序列, C1和C9一定是在C12之前访问, 而对DFS则不然. 原因是两者的定义不同.

▪ Application using graphs themselves as data structures

3. Topological Sorting / 拓扑排序

实现算法: 拓扑排序结果序列为 **Vertex T[MaxVertex];**

//Pre:有向图G采用邻接表存储结构, 即:

```
typedef struct { VertexNode adjlist[MaxVertexNum];  
                int n,e; //图的当前顶点数和弧数  
            } Graph; //邻接表类型
```

其中顶点表结点VertexNode结构为:

vertex	firstedge
--------	-----------

边表结点EdgeNode结构为:

adjvex	nexttedge
--------	-----------

Post: Generates the resulting topological order of graph G in the array T.

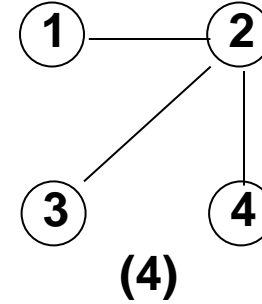
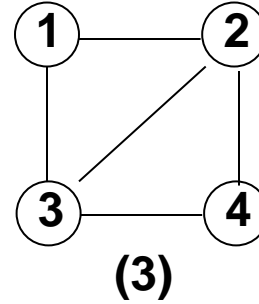
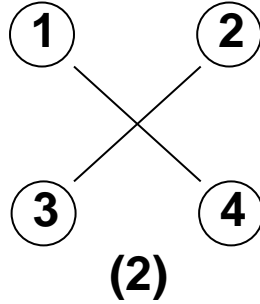
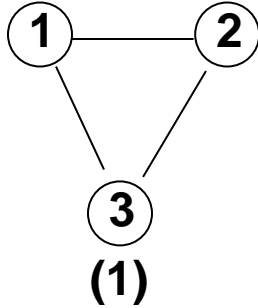
void TopologicalSort (Graph *G, Toporder T)

```
{    FindInDegree(G, indegree); //求各顶点的入度, 并置于入度向量indegree  
    InitStack(S);  
    for ( i = 0; i < G->n; ++i ) if ( !indegree[i] ) Push(S, i); //入度为0的顶点进栈  
    count = 0; //对输出顶点计数  
    while ( !StackEmpty(S) ) {  
        T[++count] = Pop(S, i); //输出i号顶点并计数即置于拓扑序列T中  
        for ( p = G->adjlist[i].firstedge; p; p = p->nextedge ) {  
            k = p->adjvex; //对i号顶点的每个邻接点的入度减1  
            if ( !(--indegree[k]) ) Push(S, k); //若入度减为0, 则入栈 } }  
    if ( count < G->n ) ERROR("该有向图有回路!"); }
```

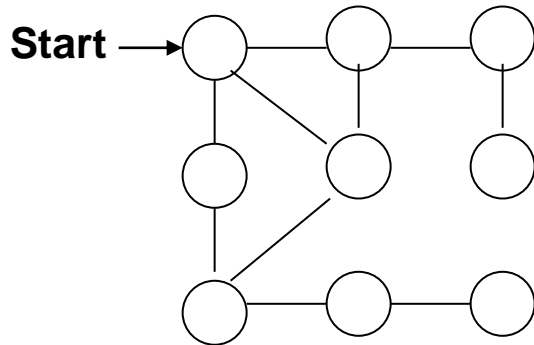
算法分析: 时间复杂性为 $O(n+e)$. // 上述算法也可用队列作为辅助存储结构

■ **Exercise (After class)**

1. 在下列各图中, **(a)**找出所有的简单回路; **(b)**哪些图是连通图? 对非连通图给出其连通分量; **(c)**哪些图是自由树?



2. Give the order of visiting the vertices of the following graph under both depth-first and breadth-first traversals.



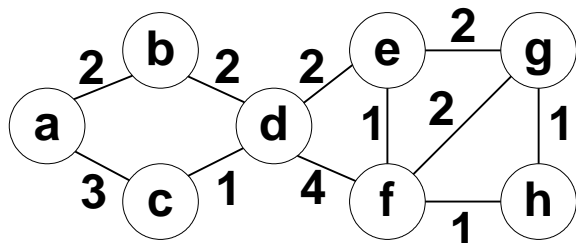
3. 假设有一棵完全二叉树包含A, B, ..., G等7个结点, 分别给出其邻接矩阵和邻接表.

▪ Exercise (After class)

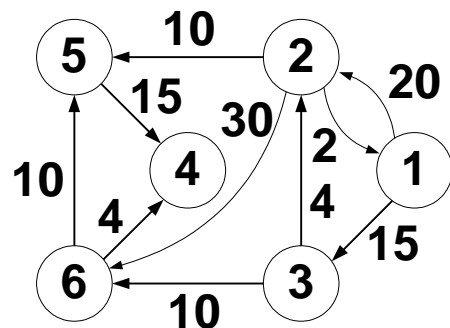
4. 按顺序输入顶点对: (1, 2), (1, 6), (2, 6), (1, 4), (6, 4), (1, 3), (3, 4), (6, 5), (4, 5), (1, 5), (3, 5), 根据算法CreateGraph画出相应的邻接表, 并写出在该邻接表上, 从顶点4开始搜索所得的DFS和BFS序列.
5. 在无向图的邻接矩阵或邻接表上分别实现如下算法:
 - ①向图中插入一个顶点 ②向图中插入一条边
 - ③删去图中某顶点 ④删去图中某条边[注: 在图中删除顶点v, 就是把v以及与v关联的边都删除; 在图中删除边, 仅需把该边删除]
6. 编写算法: 计算一个图中连通分量的个数.
7. 编写算法: 判断一个图中是否存在回路.
8. 编写算法: 查找一个图中从源点s到目标点的最短路径(即路径的边数最少).

▪ **Exercise (After class)**

9. 画出分别用**Prim**和**Kruskal**算法构造下列连通图的最小生成树的过程.



10. 利用**Dijkstra**算法求出如下有向网的从源点1到其它各顶点的最短路径.



11. 给出用**DFS**搜索下图所得到的逆拓扑序列. [注: 逆拓扑序列的形成是指每一步均是输出图中当前无后继(即出度为0)的顶点]

