



数据结构与算法 图

刘永壮

<http://homepage.hit.edu.cn/liuyongzhuang>

[*yongzhuang.liu@hit.edu.cn*](mailto:yongzhuang.liu@hit.edu.cn)

哈尔滨工业大学计算学部



学习目标

- 图结构是一种**非线性结构**，反映了数据对象之间的**任意关系**，应用非常广泛。
- **掌握**图的**定义及相关术语**、图的**逻辑结构及其特点**；
- **了解**图的**存储方法**，**重点掌握**图的**邻接矩阵**和**邻接表**存储结构；
- **掌握**图的**遍历方法**，**重点掌握**图的**遍历算法实现**；
- **了解**图的**应用**，**重点掌握****最小生成树**、**双连通性**、**强连通性**、**最短路径**、**拓扑排序**和**关键路径算法**的**基本思想**、**算法原理**和**实现过程**。



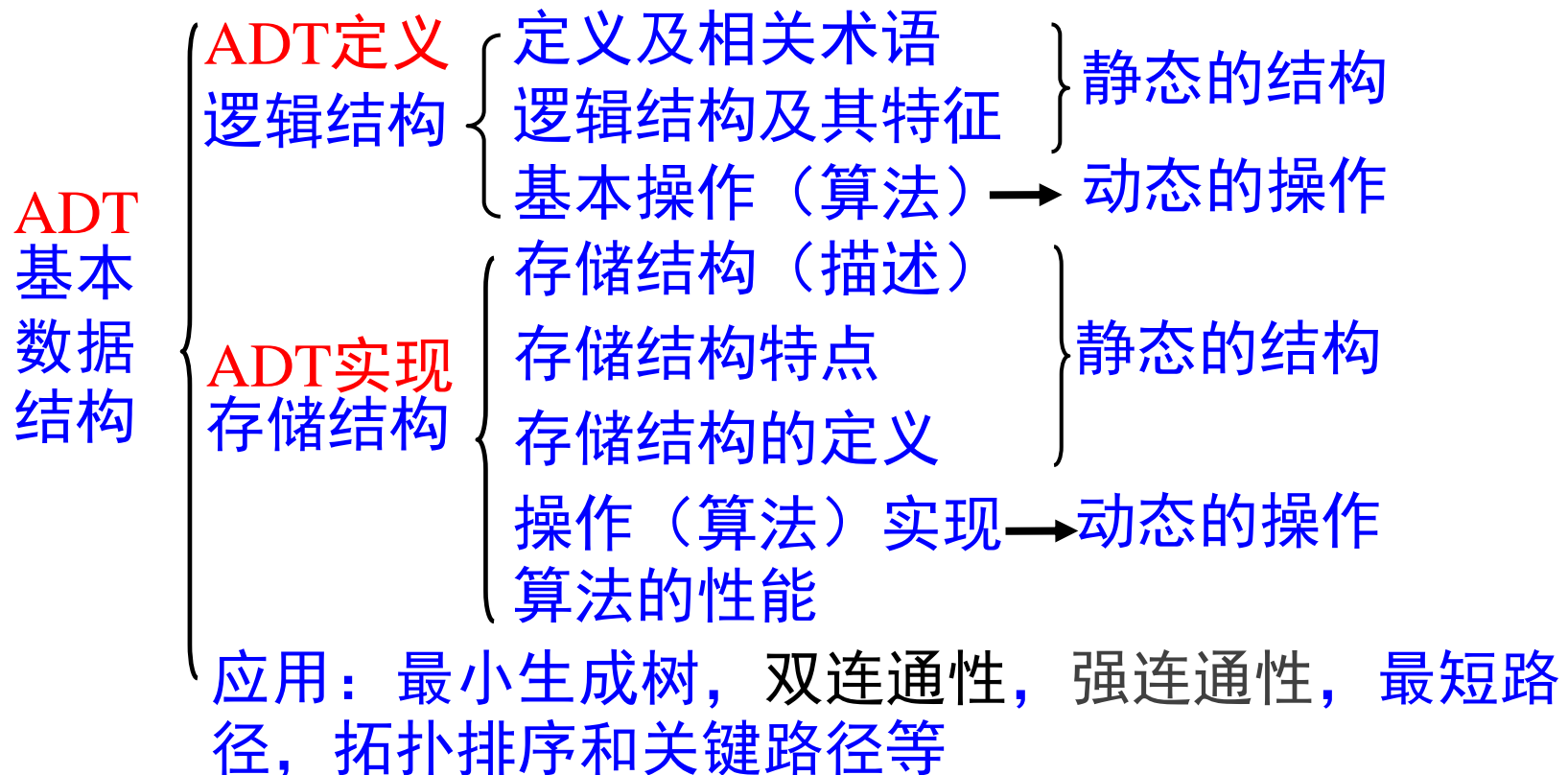
本章主要内容

- 4.1 图的基本概念
- 4.2 图的存储结构
- 4.3 图的搜索(遍历)
- 4.4 最小生成树算法
- 4.5 双连通性算法
- 4.6 强连通性算法
- 4.7 最短路径算法
- 4.8 拓扑排序算法
- 4.9 关键路径算法
- 本章小结



本章的知识点结构

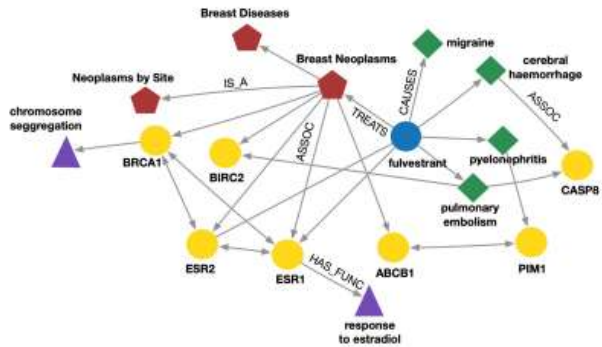
- 基本的数据结构（ADT）
 - 图（无向图、有向图；加权图----网络）
- 知识点结构



图的搜索(遍历)算法是有关图问题的重要基本核心算法



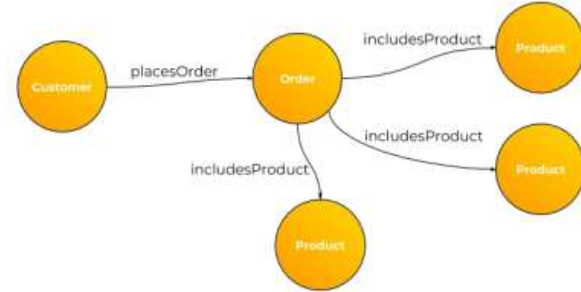
大多重要数据为图数据



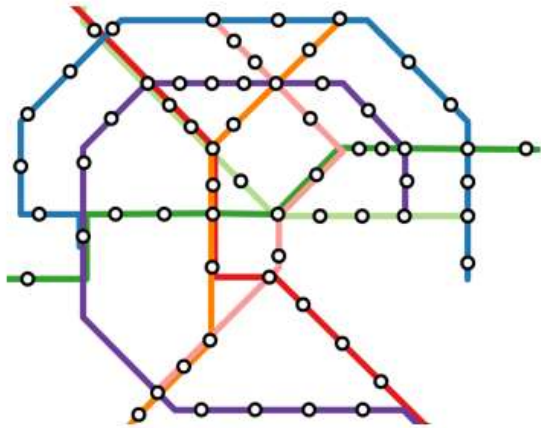
医疗健康



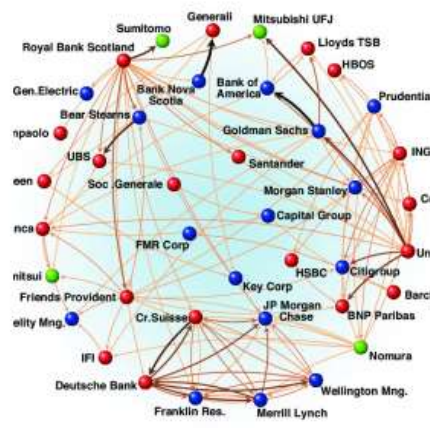
信息产业



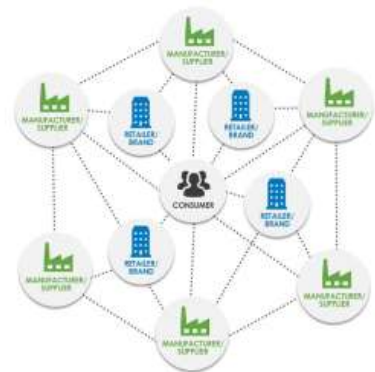
商业零售



交通



金融保险



工业生产



4.1 基本定义

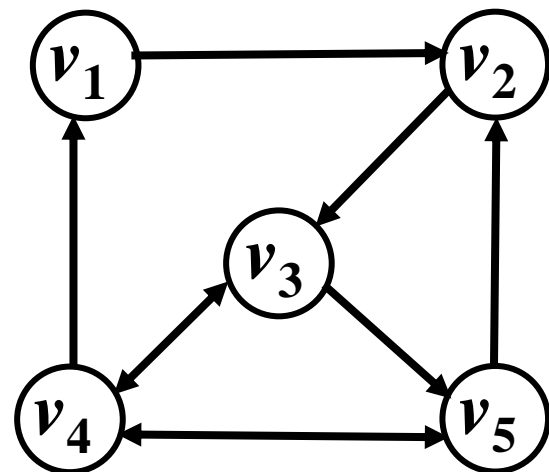
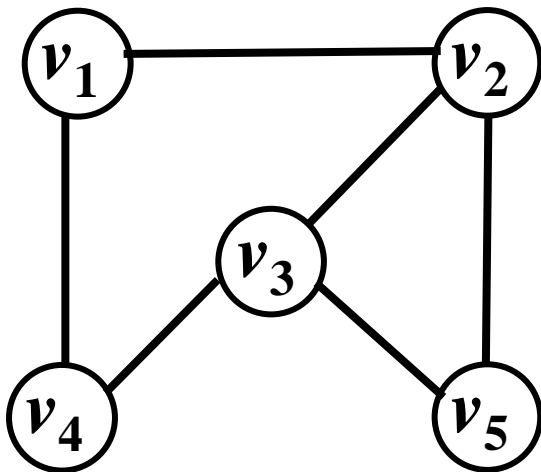
定义1：图(Graph)

- 图是由**顶点(vertex)**的**有穷非空**集合和顶点之间**边(edge)**的集合组成的一种数据结构，通常表示为：

$$G = (V, E)$$

其中： G 表示一个图， V 是图 G 中顶点的集合， E 是图 G 中顶点之间边的集合。

- 顶点**表示**数据对象**；**边**表示**数据对象之间的关系**。



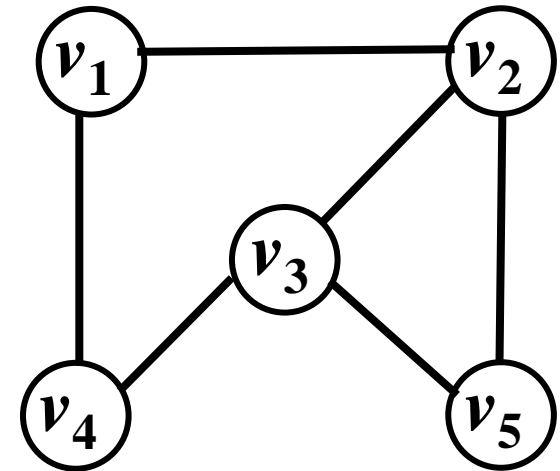


4.1 基本定义(cont.)

定义1：图

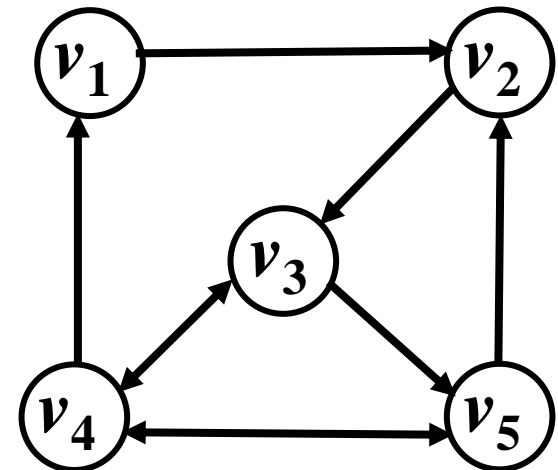
• 无向图：

- 若顶点 v_i 和 v_j 之间的边没有方向，则称这条边为**无向边**，表示为 (v_i, v_j) 。
- 如果图的任意两个顶点之间的边都是无向边，则称该图为**无向图**。



• 有向图：

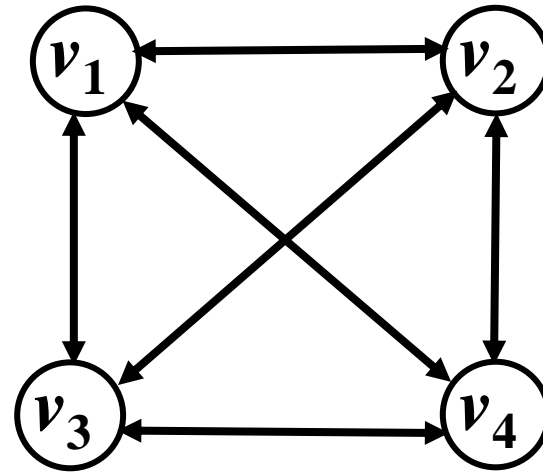
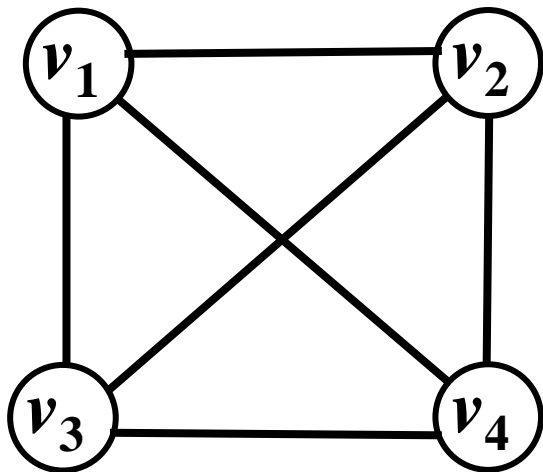
- 若顶点 v_i 和 v_j 之间的边有方向，则称这条边为**有向边(弧)**，表示为 $\langle v_i, v_j \rangle$ ：
- **<弧尾, 弧首(头)>**
- 如果图的任意两个顶点之间的边都是有向边，则称该图为**有向图**。





4.1 基本定义(cont.)

- **无向完全图**：在无向图中，如果任意两个顶点之间都存在边，则称该图为无向完全图。
- **有向完全图**：在有向图中，如果任意两个顶点之间都存在方向相反的两条弧，则称该图为有向完全图。
 - 含有 n 个顶点的无向完全图有多少条边？
 - 含有 n 个顶点的有向完全图有多少条弧？



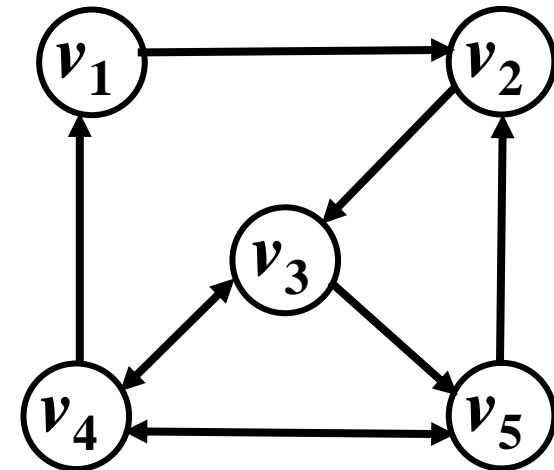
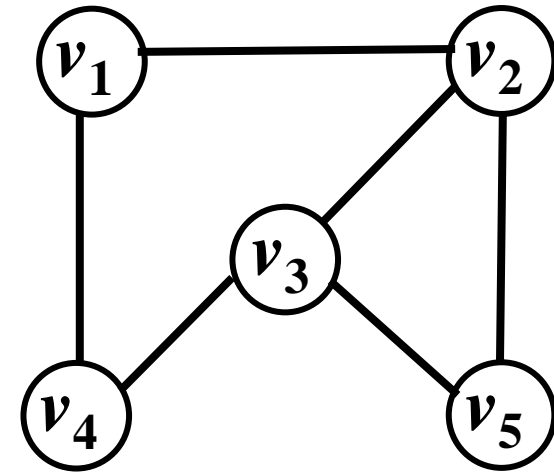


4.1 基本定义(cont.)

定义1：图

• 邻接、依附

- 在**无向图**中，对于任意两个顶点 v_i 和顶点 v_j ，若**存在边** (v_i, v_j) ，则称顶点 v_i 和顶点 v_j **相邻**，互为**邻接点**，同时称边 (v_i, v_j) **依附于**顶点 v_i 和顶点 v_j 。
- 例如： v_2 的邻接点： v_1, v_3, v_5
- 在**有向图**中，对于任意两个顶点 v_i 和顶点 v_j ，若存在有向边 $\langle v_i, v_j \rangle$ ，则称顶点 v_i **邻接到**顶点 v_j ，顶点 v_j **邻接于**顶点 v_i ，同时称弧 $\langle v_i, v_j \rangle$ **依附于**顶点 v_i 和顶点 v_j 。
- 例如： v_1 的邻接到 v_2 ， v_1 邻接于 v_4

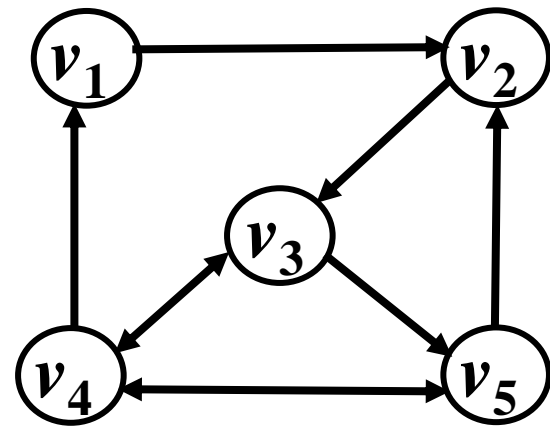
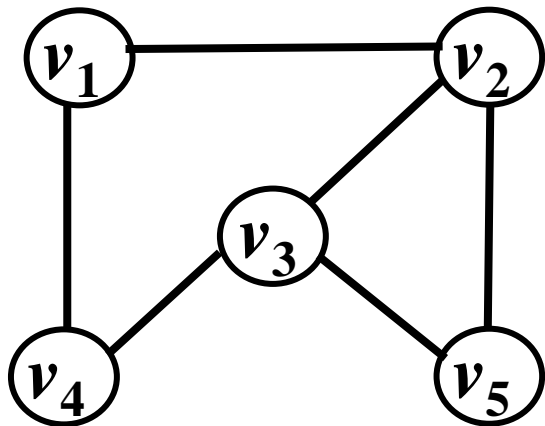




4.1 基本定义(cont.)

定义2：度(Degree)

- **顶点的度**：在**无向图**中，顶点 v 的**度**是指依附于该顶点的边数，通常记为 $D(v)$ 。
- **顶点的入度**：在**有向图**中，顶点 v 的**入度**是指以该顶点为**弧头**的弧的数目，记为 $ID(v)$ ；
- **顶点的出度**：在**有向图**中，顶点 v 的**出度**是指以该顶点为**弧尾**的弧的数目，记为 $OD(v)$ 。
- 在**有向图**中， $D(v) = ID(v) + OD(v)$





4.1 基本定义(cont.)

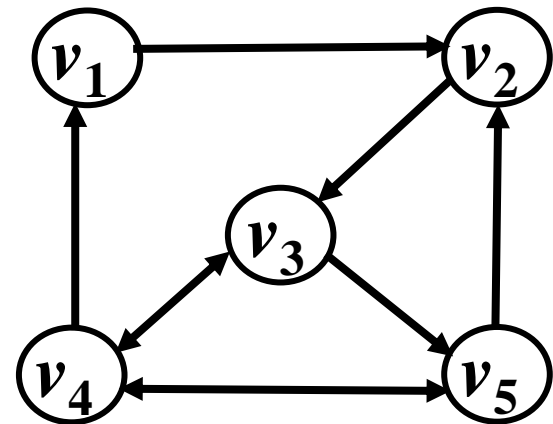
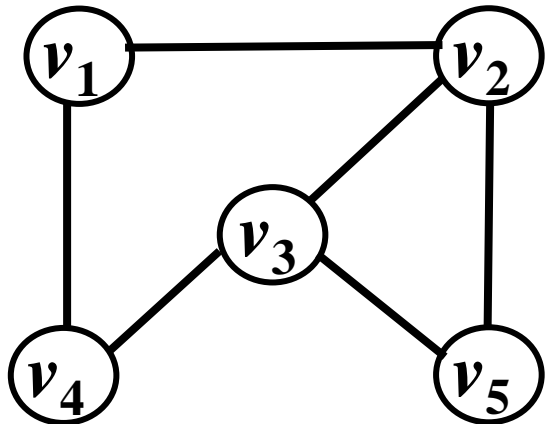
定义2：度(Dgree)

- 在具有 n 个顶点、 e 条边的**无向图** G 中，各顶点的度之和与边数之和的关系？

$$\sum_{i=1}^n D(v_i) = 2e$$

- 在具有 n 个顶点、 e 条边的**有向图** G 中，各顶点的入度之和与各顶点的出度之和的关系？与边数之和的关系？

$$\sum_{i=1}^n ID(v_i) = \sum_{i=1}^n OD(v_i) = e$$





4.1 基本定义(cont.)

定义3：路径（Path）和路径长度、简单路和简单回路

- 在**无向图** $G=(V, E)$ 中，若存在一个**顶点序列** $v_p, v_{i1}, v_{i2}, \dots, v_{im}, v_q$ ，使得 $(v_p, v_{i1}), (v_{i1}, v_{i2}), \dots, (v_{im}, v_q) \in E(G)$ ，则称顶点 v_p 路到 v_q 有一条**路径**。
- 在**有向图** $G=(V, E)$ 中，若存在一个**顶点序列** $v_p, v_{i1}, v_{i2}, \dots, v_{im}, v_q$ ，使得有向边 $\langle v_p, v_{i1} \rangle, \langle v_{i1}, v_{i2} \rangle, \dots, \langle v_{im}, v_q \rangle \in E(G)$ ，则称顶点 v_p 路到 v_q 有一条**有向路径**。
- **非带权图**的**路径长度**是指此路径上边的**条数**。
- **带权图**的**路径长度**是指路径上**各边的权之和**。
- **简单路径**：若路径上各顶点 v_1, v_2, \dots, v_m 均互不相同, 则称这样的路径为简单路径。
- **简单回路**：若路径上第一个顶点 v_1 与最后一个顶点 v_m 重合, 则称这样的简单路径为**简单回路或环**。

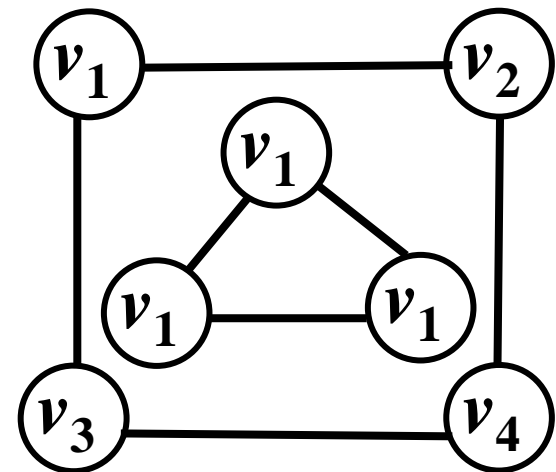


4.1 基本定义(cont.)

定义4：图的连通性

- 连通图与连通分量

- 顶点的连通性：在无向图中, 若从顶点 v_i 到顶点 v_j ($i \neq j$) 有路径, 则称顶点 v_i 与 v_j 是连通的。
- 连通图：如果一个无向图中任意一对顶点都是连通的, 则称此图是连通图。
- 连通分量：非连通图的极大连通子图叫做连通分量。
 - 含有极大顶点数
 - 依附于这些顶点的所有边



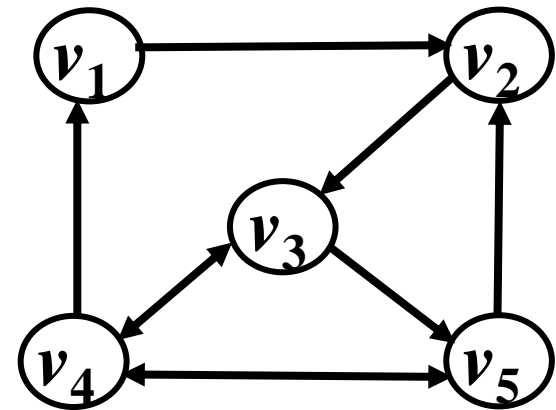


4.1 基本定义(cont.)

定义4 图的连通性

- 强连通图与强连通分量

- **顶点强连通性**：在**有向图**中, 若对于每一对顶点 v_i 和 v_j ($i \neq j$), **都存在**一条从 v_i 到 v_j 和从 v_j 到 v_i 的**有向**路径, 则称顶点 v_i 与 v_j 是**强连通的**。
- **强连通图**：如果一个**有向图**中任意一对顶点都是强连通的, 则称此有向图是**强连通图**。
- **强连通分量**：非强连通图的极大强连通子图叫做**强连通分量**。





4.1 基本定义(cont.)

图的操作

设图 $G=(V,E)$ ，图上的基本操作如下：

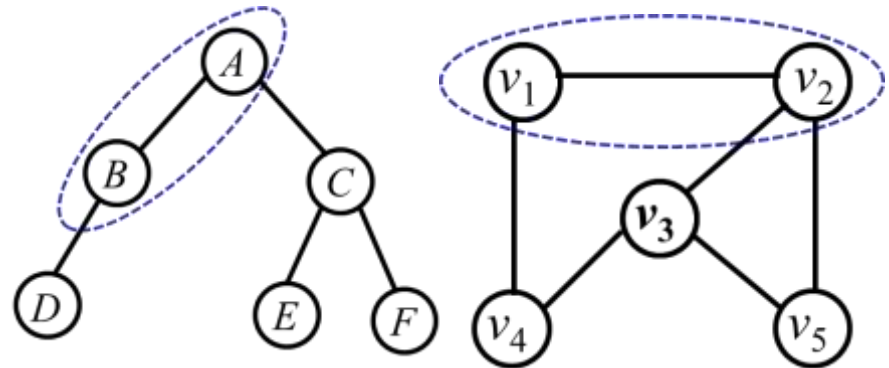
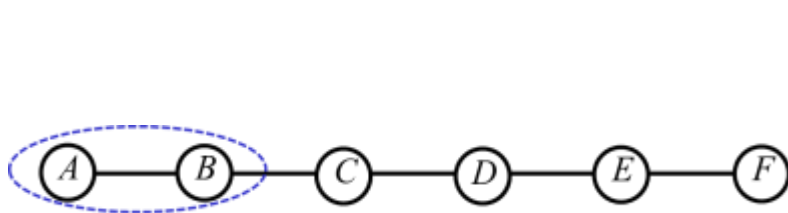
- $NEWNODE (G)$ ：建立一个新顶点， $V=V \cup \{v\}$
- $DELNODE (G, v)$ ：删除顶点 v 以及与之相关联的所有边
- $SETSUCC (G, v1, v2)$ ：增加一条边， $E = E \cup (v1,v2)$, $V=V$
- $DELSUCC (G, v1, v2)$ ：删除边 $(v1,v2)$, V 不变
- $SUCC (G, v)$ ：求出 v 的**所有**直接后继结点（结点序列）
- $PRED (G, v)$ ：求出 v 的**所有**直接前导结点（结点序列）
- $ISEDGE (G, v1, v2)$ ：**判断** $(v1,v2) \in E$
- $FirstAdjVex(G, v)$ ：顶点 v 的**第一个**邻接顶点
- $NextAdjVex(G, v, w)$ ：顶点 v 的某个邻接点 w 的**下一个**邻接顶点。



4.1 基本定义(cont.)

不同逻辑结构比较

- 在线性结构中，数据元素之间仅具有**线性关系(1:1)**；
 - 在树型结构中，结点之间具有**层次关系(1:m)**；
 - 在图型结构中，任意两个顶点之间**都可能有关系(m:n)**。
-
- 在线性结构中，元素之间的关系为**前驱**和**后继**；
 - 在树型结构中，结点之间的关系为**双亲**和**孩子**；
 - 在图型结构中，顶点之间的关系为**邻接**。





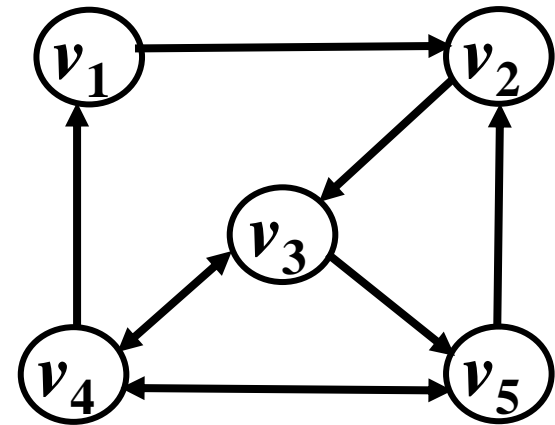
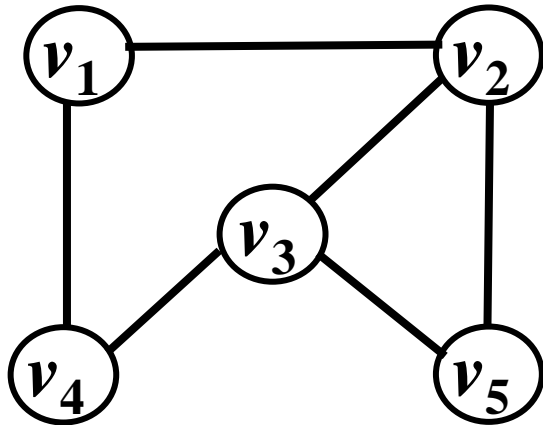
4.2 图的存储结构



4.2 图的存储结构

• 如何存储图？

- 图是由顶点和边组成的，顶点之间的关系是 $m:n$ ，即任何两个顶点之间都可能存在关系（边）；
- 分别考虑如何存储顶点、如何存储边（顶点之间的关系）。
- 邻接矩阵 (Adjacent Matrix)
- 邻接表 (Adjacent List)





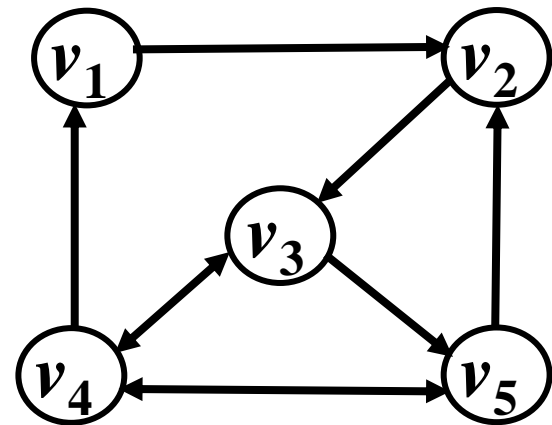
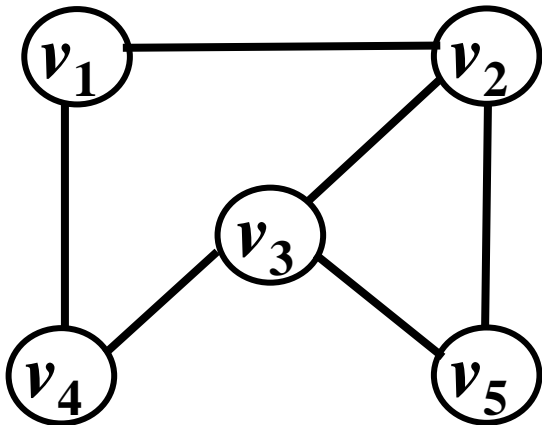
4.2 图的存储结构(cont.)

一、邻接矩阵 (Adjacency Matrix) 表示(数组表示法)

• 基本思想:

- 用一个一维数组存储图中顶点的信息
- 用一个二维数组（称为邻接矩阵）存储图中各顶点之间的邻接关系
- 假设图 $G=(V, E)$ 有 n 个顶点，则邻接矩阵是一个 $n \times n$ 方阵，定义为：

$$\text{edge}[i][j] = \begin{cases} 1 & \text{若 } (i, j) \in E \text{ 或 } \langle i, j \rangle \in E \\ 0 & \text{否则} \end{cases}$$

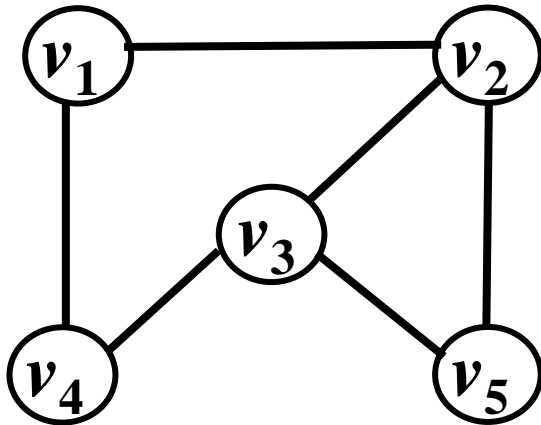




4.2 图的存储结构(cont.)

一、邻接矩阵 (Adjacency Matrix) 表示(数组表示法)

• 无向图的邻接矩阵:



vertex =

v_1	v_2	v_3	v_4	v_5
-------	-------	-------	-------	-------

edge =

	v_1	v_2	v_3	v_4	v_5
v_1	0	1	0	1	0
v_2	1	0	1	0	1
v_3	0	1	0	1	1
v_4	1	0	1	0	0
v_5	0	1	1	0	0

• 存储结构特点:

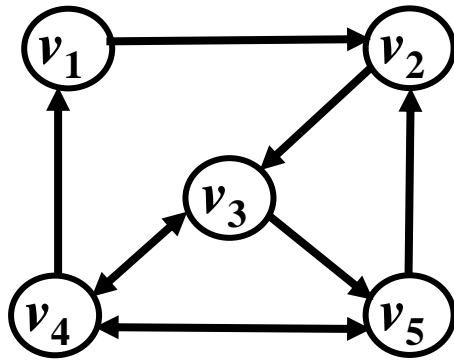
- 主对角线为 0 且一定是对称矩阵;
- 如何求顶点 v_i 的度?
- 如何判断顶点 v_i 和 v_j 之间是否存在边?
- 如何求顶点 v_i 的所有邻接点?



4.2 图的存储结构(cont.)

一、邻接矩阵 (Adjacency Matrix) 表示(数组表示法)

• 有向图的邻接矩阵：



vertex =

v_1	v_2	v_3	v_4	v_5
-------	-------	-------	-------	-------

edge =

v_1	v_2	v_3	v_4	v_5
0	1	0	0	0
0	0	1	0	0
0	0	0	1	1
1	0	1	0	1
0	1	0	1	0

v_1
 v_2
 v_3
 v_4
 v_5

• 存储结构特点：

- 有向图的邻接矩阵一定不对称吗？
- 如何求顶点 v_i 的出度？
- 如何求顶点 v_i 的入度？
- 如何判断顶点 v_i 和 v_j 之间是否存在有向边？
- 如何求邻接于顶点 v_i 的所有顶点？
- 如何求邻接到顶点 v_i 的所有顶点？

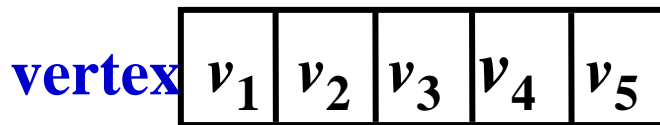


4.2 图的存储结构(cont.)

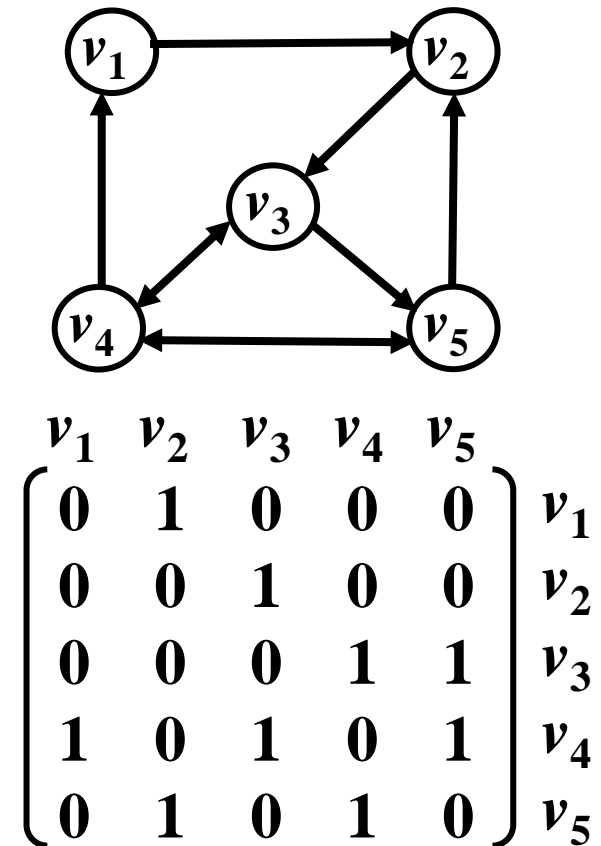
一、邻接矩阵 (Adjacency Matrix) 表示(数组表示法)

• 存储结构定义:

```
typedef struct {  
    VertexData vertex [NumVertices]; //顶点表  
    EdgeData edge[NumVertices][NumVertices];  
    //邻接矩阵: 边表, 可视为顶点之间关系  
    int n, e; //图的顶点数与边数  
} MTGraph;
```



edge=



• 空间复杂度:

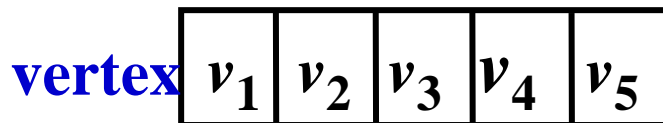
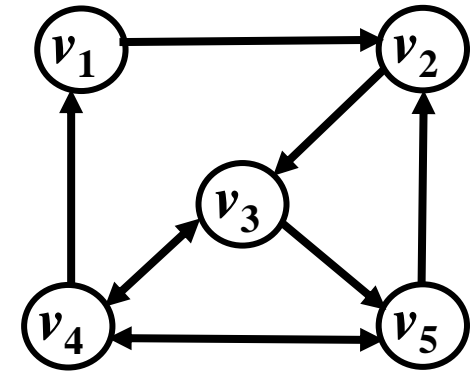
- 假设图G有n个顶点e条边, 则该图的存储需求为 $O(n+n^2) = O(n^2)$ 。
- 与边数量e无关。



4.2 图的存储结构(cont.)

存储结构建立算法：实现步骤

1. 确定图的顶点个数 n 和边数 e ;
2. 输入**顶点信息**存储在一维数组 $vertex$ 中;
3. 初始化邻接矩阵;
4. 依次输入**每条边**存储在邻接矩阵 $edge$ 中;
 - 4.1 输入边依附的两个顶点的序号 i, j ;
 - 4.2 将邻接矩阵的第 i 行第 j 列的元素值置为1;
 - 4.3 将邻接矩阵的第 j 行第 i 列的元素值置为1。



edge =

v_1	v_2	v_3	v_4	v_5	
0	1	0	0	0	v_1
0	0	1	0	0	v_2
0	0	0	1	1	v_3
1	0	1	0	1	v_4
0	1	0	1	0	v_5



4.2 图的存储结构(cont.)

存储结构建立算法：实现

```
void CreateMGraph (MTGraph *G) //建立图的邻接矩阵
{
    int i, j, k, w;
    cin>>G→n>>G→e;           //1.输入顶点数和边数
    for (i=0; i<G→n; i++)       //2.读入顶点信息，建立顶点表
        G→vexlist[i]=getchar( );
    for (i=0; i<G→n; i++)
        for (j=0; j<G→n; j++)
            G→edge[i][j]=0;     //3.邻接矩阵初始化
    for (k=0; k<G→e; k++) {     //4.读入e条边建立邻接矩阵
        cin>>i>>j>>w;          // 输入边(i, j)上的权值w
        G→ edge[i][j]=w; G→ edge[j][i]=w;
    }
} //时间复杂度：  $T = O(n + n^2 + 2e)$ 。
```

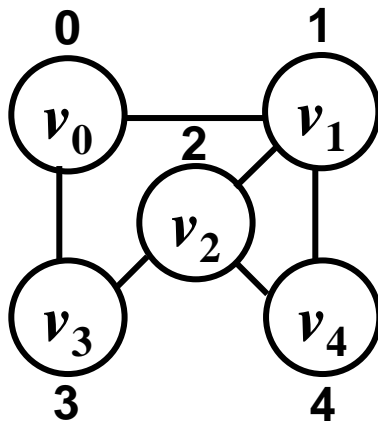



4.2 图的存储结构(cont.)

二、邻接表(Adjacency List)表示

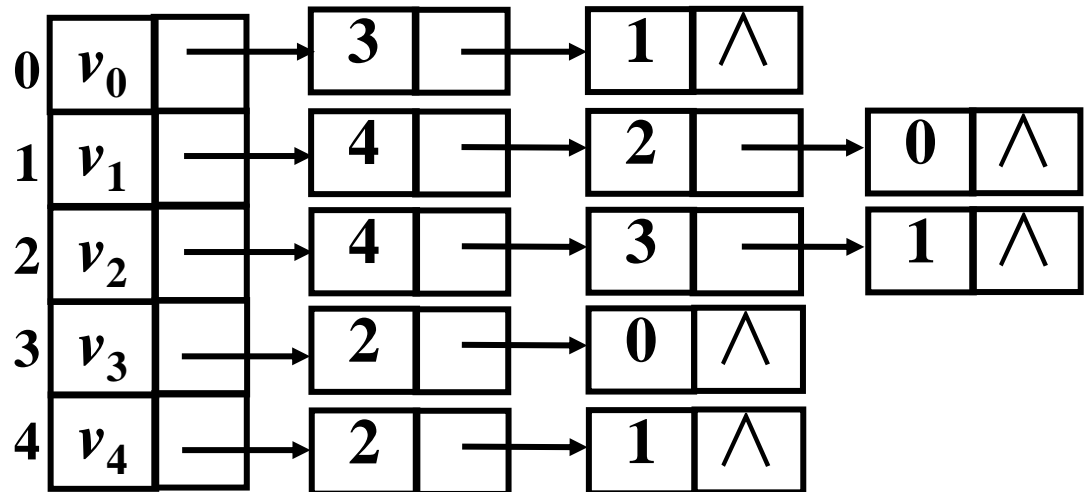
• 无向图的邻接表:

- 对于无向图的每个顶点 v_i , 将所有与 v_i 相邻的顶点链成一个单链表, 称为顶点 v_i 的边表 (顶点 v_i 的邻接表);
- 再将所有边表的指针和顶点信息构成一维数组的顶点表。



vertex firstedge

adjvex next



顶点表

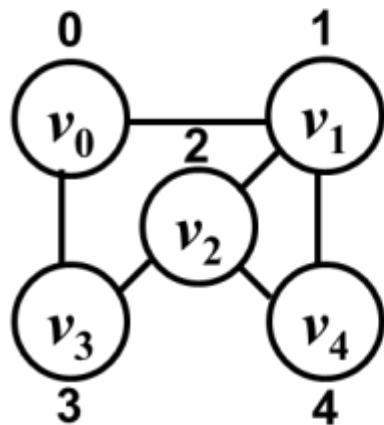
边表



4.2 图的存储结构(cont.)

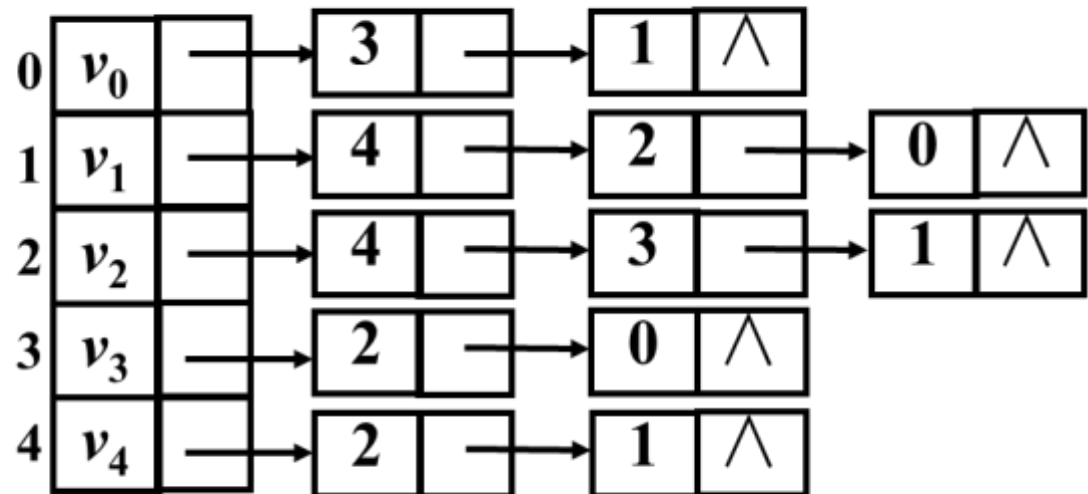
• 无向图的邻接表存储的特点:

- 边表中的结点表示什么?
- 如何求顶点 v_i 的度?
- 如何判断顶点 v_i 和顶点 v_j 之间是否存在边?
- 如何求顶点 v_i 的所有邻接点?
- 空间需求 $O(n+2e)$



vertex firstedge

adjvex next



顶点表

边表

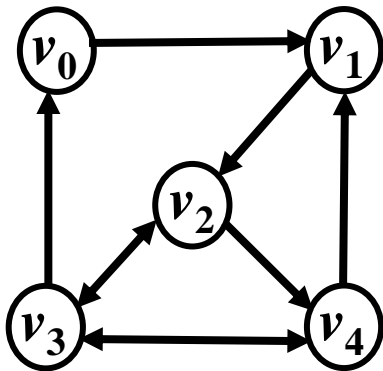


4.2 图的存储结构(cont.)

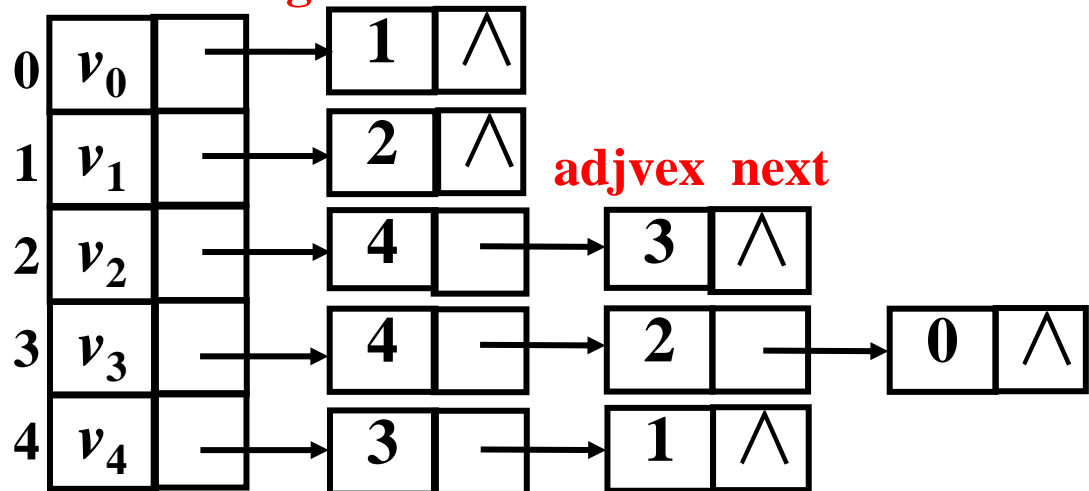
二、邻接表(Adjacency List)表示

• 有向图的邻接表：正邻接表

- 对于有向图的每个顶点 v_i ，将邻接于 v_i 的所有顶点链成一个单链表，称为顶点 v_i 的出边表；
- 再把所有出边表的指针和顶点信息的一维数组构成顶点表。



vertex firstedge



顶点表

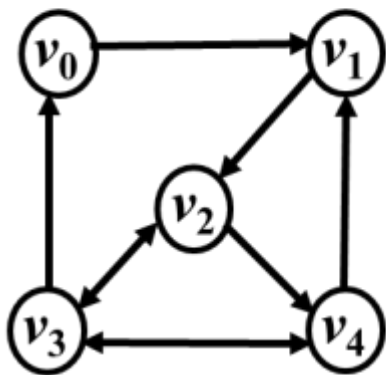
出边表



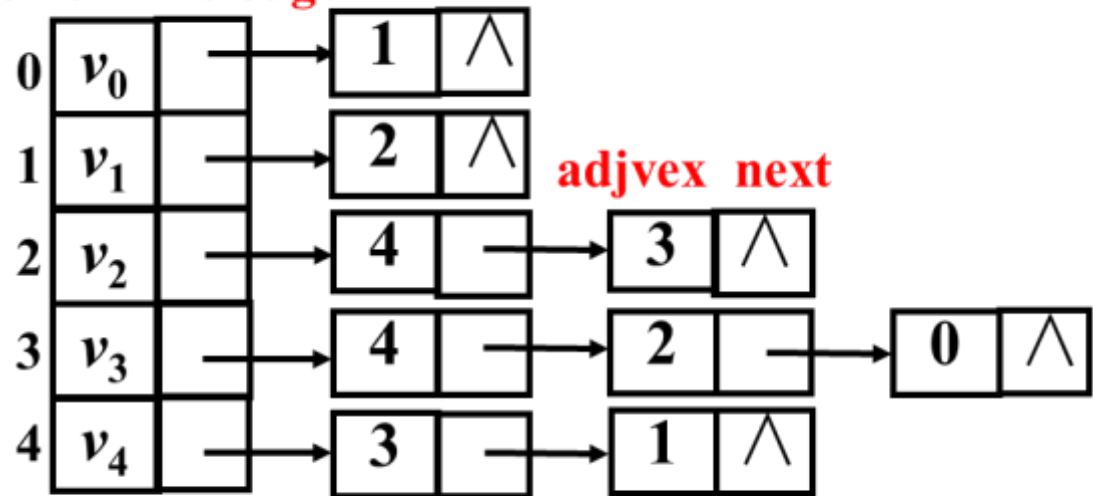
4.2 图的存储结构(cont.)

• 有向图的正邻接表的存储特点

- 出边表中的结点表示什么？
- 如何求顶点 v_i 的出度？如何求顶点 v_i 的入度？
- 如何判断顶点 v_i 和顶点 v_j 之间是否存在有向边？
- 如何求邻接于顶点 v_i 的所有顶点？
- 如何求邻接到顶点 v_i 的所有顶点？
- 空间需求: $O(n+e)$



vertex firstedge



顶点表

出边表

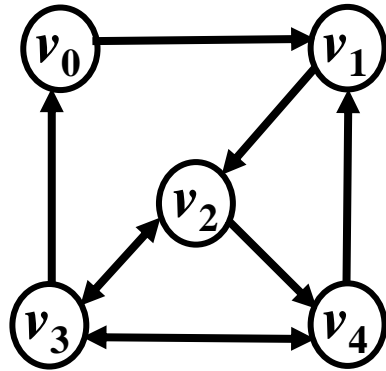


4.2 图的存储结构(cont.)

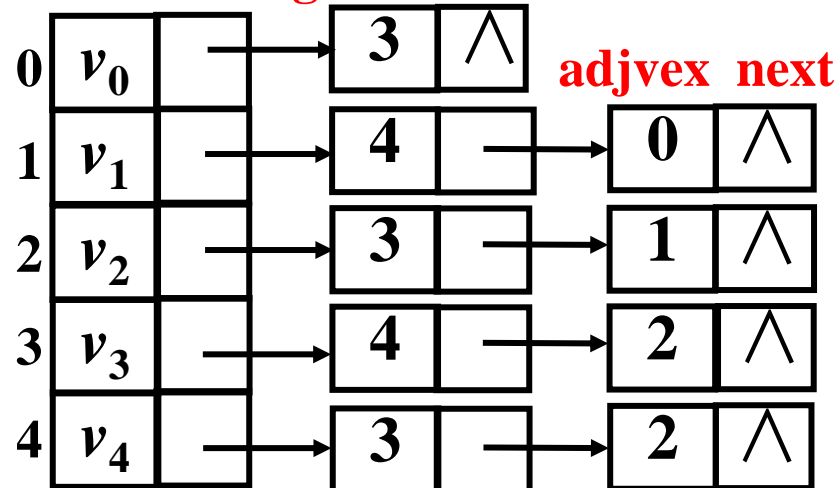
二、邻接表(*Adjacency List*)表示

• 有向图的邻接表：逆邻接表

- 对于有向图的每个顶点 v_i ，将邻接到 v_i 的所有顶点链成一个单链表，称为顶点 v_i 的入边表；
- 再把所有入边表的指针和存储顶点信息的一维数组构成顶点表。



vertex firstedge



顶点表

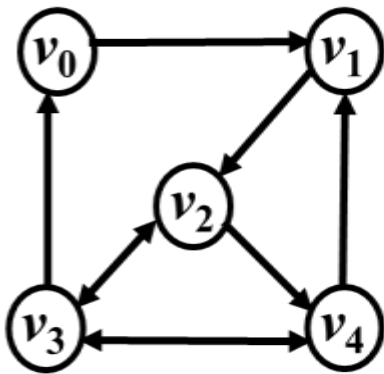
入边表



4.2 图的存储结构(cont.)

• 有向图的逆邻接表的存储特点

- 入边表中的结点表示什么？
- 如何求顶点 v_i 的入度？如何求顶点 v_i 的出度？
- 如何判断顶点 v_i 和顶点 v_j 之间是否存在有向边？
- 如何求邻接到顶点 v_i 的所有顶点？
- 如何求邻接于顶点 v_i 的所有顶点？
- 空间需求: $O(n+e)$



vertex firstedge

	vertex	firstedge	adjvex	next
0	v_0	3	\wedge	
1	v_1	4	0	\wedge
2	v_2	3	1	\wedge
3	v_3	4	2	\wedge
4	v_4	3	2	\wedge

顶点表

入边表

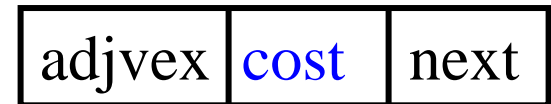


4.2 图的存储结构(cont.)

- 邻接表存储结构的定义：无向图

```
typedef struct node {           //边表结点
    int adjvex;                 //邻接点域（下标）
    EdgeData cost;              //边上的权值
    struct node *next;          //下一边链接指针
} EdgeNode;
```

边表结点



```
typedef struct {               //顶点表结点
    VertexData vertex;         //顶点数据域
    EdgeNode * firstedge;      //边链表头指针
} VertexNode;
```

顶点表结点



```
typedef struct {               //图的邻接表
    VertexNode vexlist [NumVertices]; //顶点表结点数组
    int n, e;                  //顶点个数与边数
} AdjGraph;
```



4.2 图的存储结构(cont.)

- 邻接表存储结构的定义：有向图

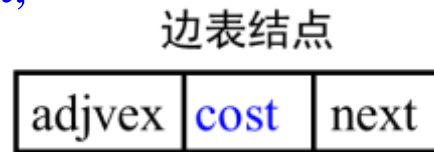
```
typedef struct node
```

```
int adjvex;
```

```
EdgeData cost;
```

```
node *next;
```

```
} EdgeNode;
```

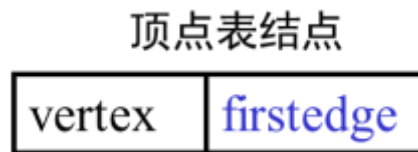


```
typedef struct {
```

```
VertexData vertex;
```

```
EdgeNode * firstedge;
```

```
} VertexNode;
```

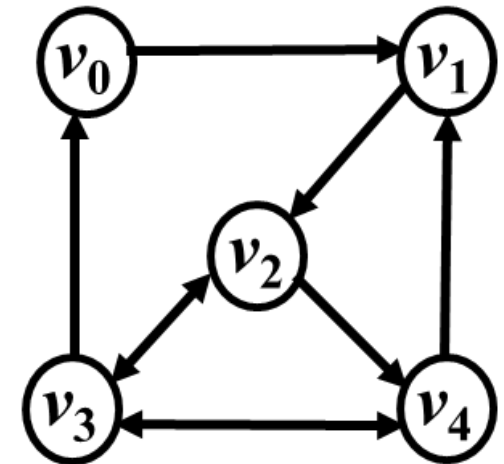


```
typedef struct {
```

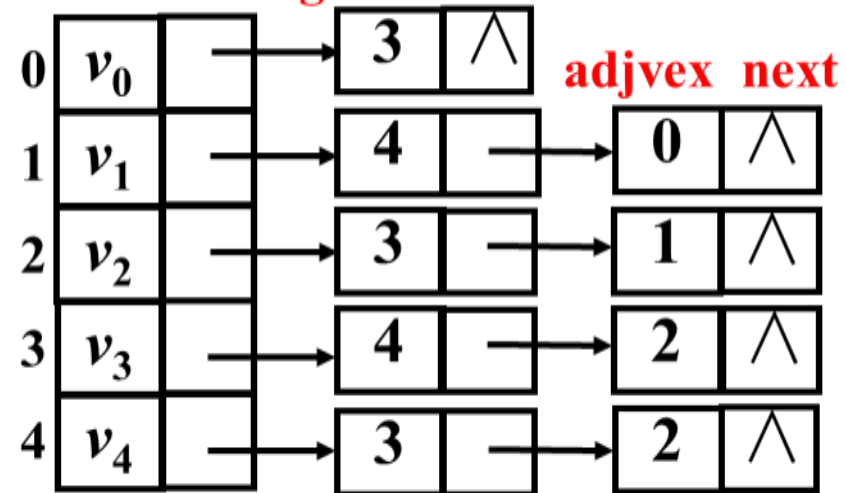
```
VertexNode vexlist [NumVertices];
```

```
int n, e;
```

```
} AdjGraph;
```



vertex firstedge



顶点表
vexlist

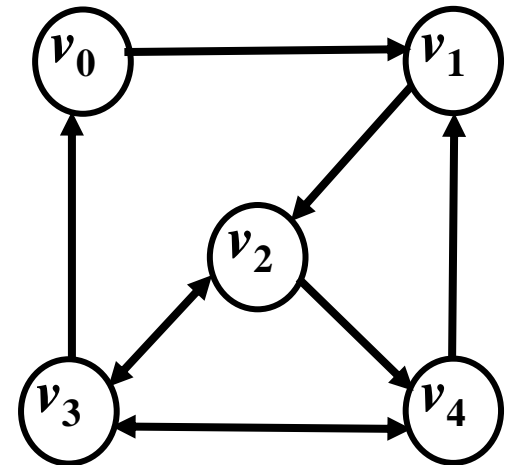
入边表



4.2 图的存储结构(cont.)

- 邻接表存储结构建立算法：实现步骤

1. 确定图的顶点个数和边个数；
2. 建立顶点表：
 - 2.1 输入顶点信息；
 - 2.2 初始化该顶点的边表；
3. 边表：依次输入边的信息并存储在边表中；
 - 3.1 输入边所依附的两个顶点的序号 $tail$ 和 $head$ 以及权值 w ；
 - 3.2 生成邻接点序号为 $head$ 的边表结点 p ；
 - 3.3 设置边表结点 p ；
 - 3.4 将结点 p 插入到第 $tail$ 个边表的头部；





4.2 图的存储结构(cont.)

邻接表存储结构建立算法：实现

void CreateGraph (AdjGraph G)

```
{ cin >> G.n >> G.e;
```

//1.输入顶点数和边数

```
for ( int i = 0; i < G.n; i++) {
```

//2.建立顶点表

```
    cin >> G.vexlist[i].vertex;
```

//2.1输入顶点信息

```
    G.vexlist[i].firstedge = NULL;} //2.2边表置为空表
```

```
for ( i = 0; i < G.e; i++) {
```

//3.逐条边输入,建立边表

```
    cin >> tail >> head >> weight;
```

//3.1输入

```
    EdgeNode * p = new EdgeNode;
```

//3.2建立新边结点

```
    p->adjvex = head; p->cost = weight;
```

//3.3设置新边结点

```
    p->next = G.vexlist[tail].firstedge;
```

//3.4新结点链入第 tail 号链表的前端

```
    G.vexlist[tail].firstedge = p;
```

```
    p = new EdgeNode;
```

```
    p->adjvex = tail; p->cost = weight;
```

```
    p->next = G.vexlist[head].firstedge;
```

//链入第 head 号链表的前端

```
G.vexlist[head].firstedge = p; }
```

//时间复杂度： O(n+e)

```
typedef struct {
```

```
    VertexNode vexlist [NumVertices];
```

```
    int n, e;
```

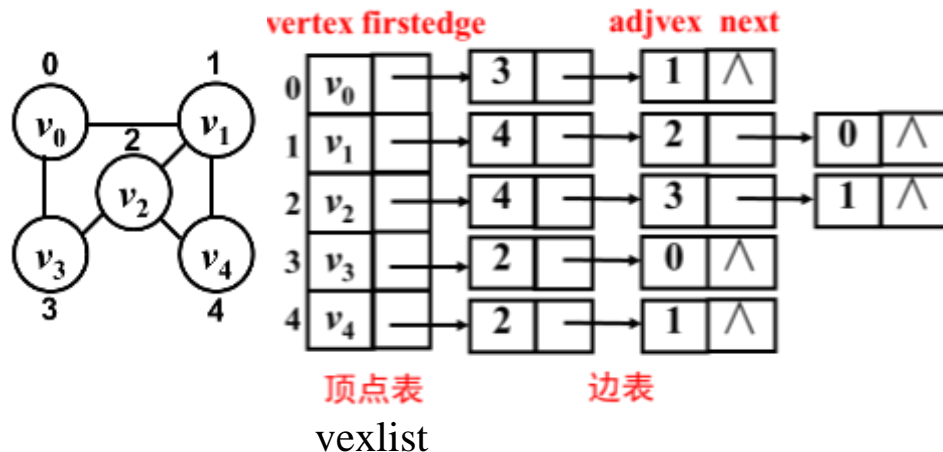
```
} AdjGraph;
```

顶点表结点

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

边表结点

adjvex	cost	next
--------	------	------

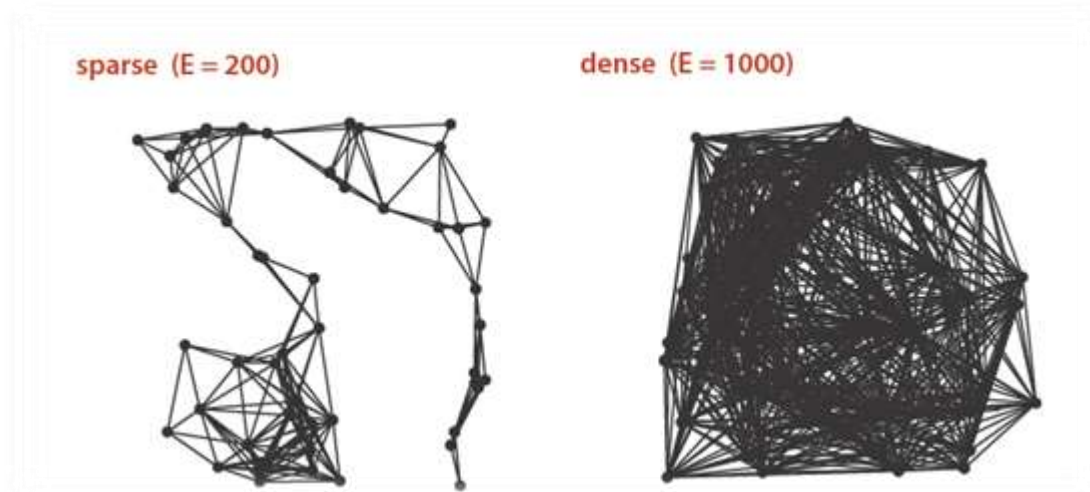




4.2 图的存储结构(cont.)

- 图存储结构比较：邻接矩阵与邻接表

	空间性能	时间性能	适用对象
邻接矩阵	$O(n^2)$	$O(n^2)$	稠密图
邻接表	$O(n+e)$	$O(n+e)$	稀疏图



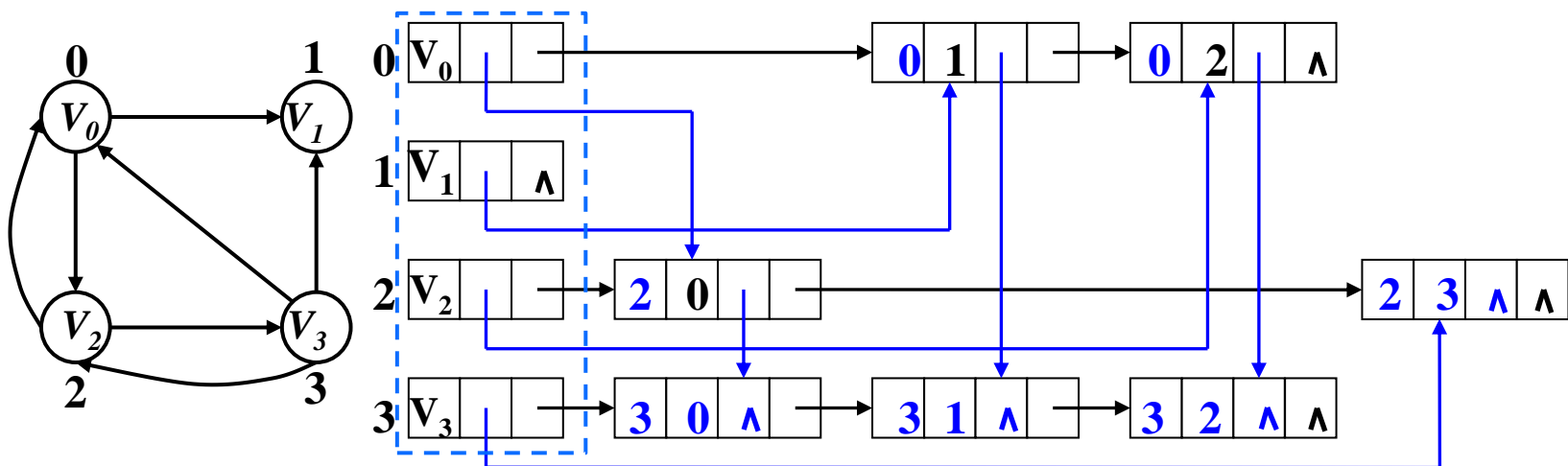
两个图结点数都是50



4.2 图的存储结构(cont.)

三、有向图的十字链表(*Orthogonal List*)表示

- 十字链表：有向图的另一种链式存储结构。
 - 可以看成是将有向图的正邻接表和逆邻接表**结合起来**得到的一种链式存储结构
 - 弧头相同的弧在同以一链表上，弧尾相同的弧也在同一链表上
 - 横向上是正邻接表(出边表)，纵向上是逆邻接表(入边表)
 - 方便有向图的顶点入度与出度计算





4.2 图的存储结构(cont.)

三、有向图的十字链表(*Orthogonal List*)表示

- 结点结构： 弧结点结构

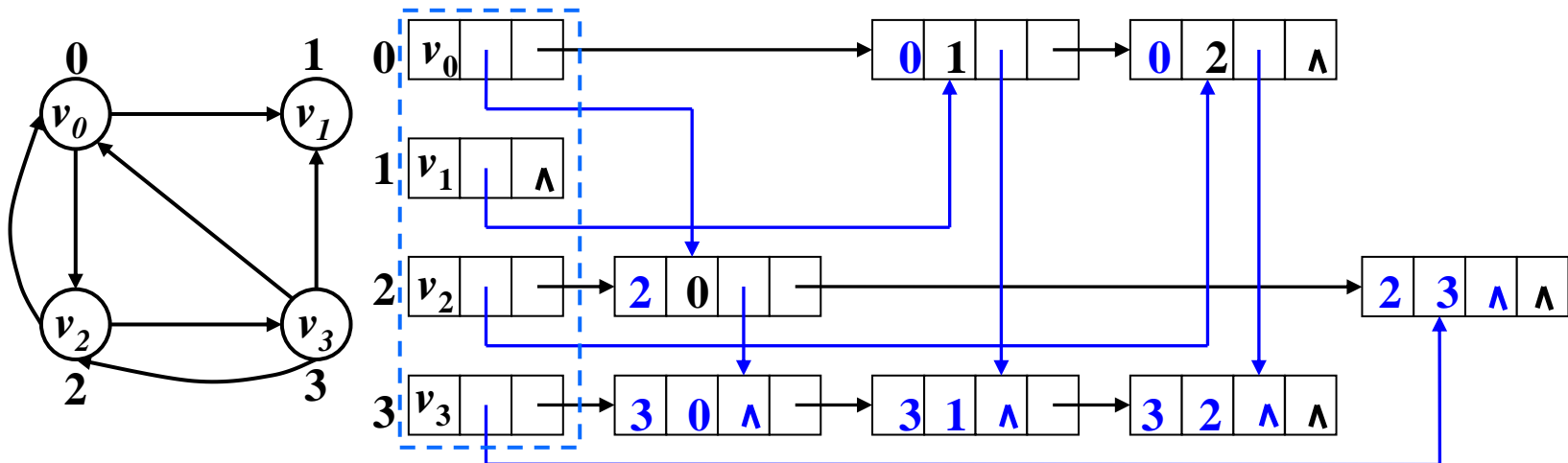
tailvex	headvex	hlink	tlink	info
---------	---------	-------	-------	------

- tailvex: 尾域, 指示弧尾顶点在图中的位置
- headvex: 头域, 指示弧头顶点在图中的位置
- hlink: 链域, 指向弧头相同的下一条弧
- tlink: 链域, 指向弧尾相同的下一条弧
- info: 数据域, 指向该弧的相关信息

- 头结点 (顶点结点) 结构

data	firstin	firstout
------	---------	----------

- data: 数据域, 存储和顶点相关的信息, 如顶点名称
- firstin: 链域, 指向以该顶点为弧头的第一个弧结点
- firstout: 链域, 指向以该顶点为弧尾的第一个弧结点





4.2 图的存储结构(cont.)

三、有向图的十字链表(*Orthogonal List*)表示

- 存储结构定义：

```
#define MAX_VERTEX_NUM 20

typedef struct ArcBox {
    int tailvex, headvex;           //该弧的尾和头顶点的位置
    struct ArcBox * hlink, * tlink; //分别为弧头相同和弧尾相同的弧的链域
    InfoType info;                 //该弧相关信息的指针
} ArcBox;

typedef struct VexNode {
    VertexType data;
    ArcBox * firstin, * firstout;   //分别指向该顶点第一条入弧和出弧
} VexNode;

typedef struct {
    VexNode xlist[MAX_VERTEX_NUM]; //表头向量
    int vexnum, arcnum;             //有向图的当前顶点数和弧数
} OLGraph;
```



4.2 图的存储结构(cont.)

三、有向图的十字链表(*Orthogonal List*)表示

• 存储结构定义：

```
#define MAX_VERTEX_NUM 20
```

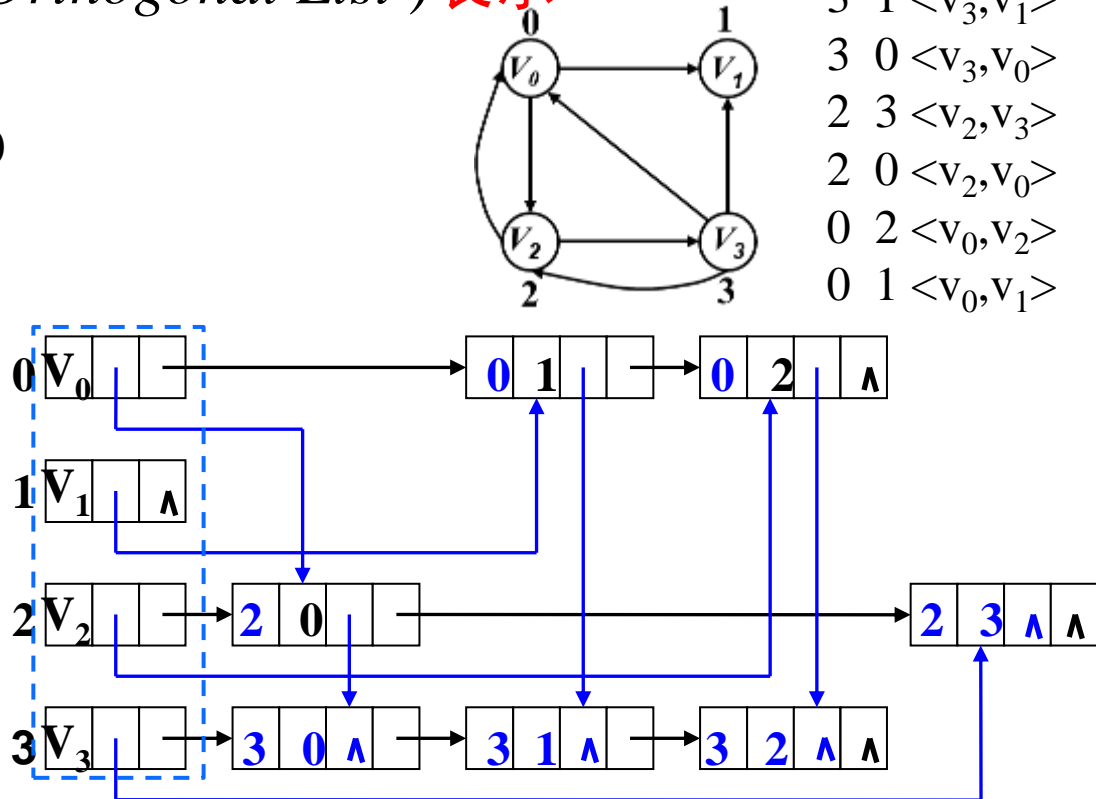
```
typedef struct ArcBox {  
    int tailvex, headvex;  
    struct ArcBox * hlink, * tlink;  
    InfoType info;  
} ArcBox;
```

```
typedef struct VexNode {  
    VertexType data;  
    ArcBox * firstin, * firstout;  
} VexNode;
```

```
typedef struct {  
    VexNode xlist[MAX_VERTEX_NUM];  
    int vexnum, arcnum;  
} OLGraph;
```

输入顺序：

3 2 $\langle v_3, v_2 \rangle$
3 1 $\langle v_3, v_1 \rangle$
3 0 $\langle v_3, v_0 \rangle$
2 3 $\langle v_2, v_3 \rangle$
2 0 $\langle v_2, v_0 \rangle$
0 2 $\langle v_0, v_2 \rangle$
0 1 $\langle v_0, v_1 \rangle$





4.2 图的存储结构(cont.)

三、有向图的十字链表(*Orthogonal List*)表示

- 构建算法:

```
void CreateDG (OLGraph &G) //采用十字链表存储表示, 构造有向图
{
    scanf (&G.vexnum, &G.arcnum, &IncInfo); //IncInfo为0则各弧不含其他信息
    for (i = 0; i < G.vexnum; ++ i) {
        scanf (&G.xlist[i].data); //构造顶点表
        G.xlist[i].firstin = NULL; G.xlist[i].firstout = NULL; //初始化指针
    }
    for (k = 0; k < G.arcnum; ++ k) {
        scanf (&v1, &v2); //输入各弧并构造十字链表(边表)
        i = LocateVex (G, v1); j = LocateVex (G, v2); //输入一条弧的始点和终点
        p = (ArcBox *) malloc (sizeof (ArcBox)); //确定v1和v2在G中位置
        *p = {i, j, G.xlist[j].firstin, G.xlist[i].firstout, NULL}; //假定有足够空间
        //对弧结点赋值 {tailvex, headvex, hlink, tlink, info}
        G.xlist[j].firstin = G.xlist[i].firstout = p; //在入弧和出弧链表头部的插入
        if (IncInfo) Input (*p->info); //若弧含有相关信息, 则输入
    } // for
} // CreateDG
```

示例的一种输入顺序: 3 2 ; 3 1; 3 0; 2 3; 2 0; 0 2; 0 1



4.3 图搜索(遍历)



4.3 图搜索(遍历)

- 图遍历（图搜索）

- 从图中**某**一顶点出发，对图中所有顶点访问一次且仅**访问一次**。
- **访问**：抽象操作，可以是对结点进行的各种处理。

- 图结构的复杂性

- 在**线性表**中，数据元素在表中的编号就是元素在序列中的位置，因而其**编号是唯一的**；
- 在**树结构**中，将结点按层序编号，由于树具有层次性，因而其**层序编号也是唯一的**；
- 在**图结构**中，任何两个顶点之间都可能存在边，顶点没有确定的先后次序，所以，**顶点的编号不唯一**。



4.3 图的搜索(遍历)(cont.)

- 图遍历要解决的关键问题

- 在图中如何选取遍历的起始顶点？
 - 解决办法：从编号小的顶点开始。
- 从某个起点开始可能到达不了所有其它顶点，怎么办？
 - 解决办法：多次调用从某顶点出发遍历图的算法。
- 图中可能存在回路，且图的任一顶点都可能与其它顶点相通，访问过程中，可能会又回到曾访问过的顶点。如何避免某些顶点可能会被重复访问？
 - 解决办法：附设访问标志数组visited[n]。
- 图中一个顶点可以和其它多个顶点相连，这样顶点访问后，如何选取下一个要访问的顶点？
 - 解决办法：深度优先搜索(Depth First Search)和广度优先搜索(Breadth First Search)。



深度优先搜索

(Depth FirstSearch, DFS)



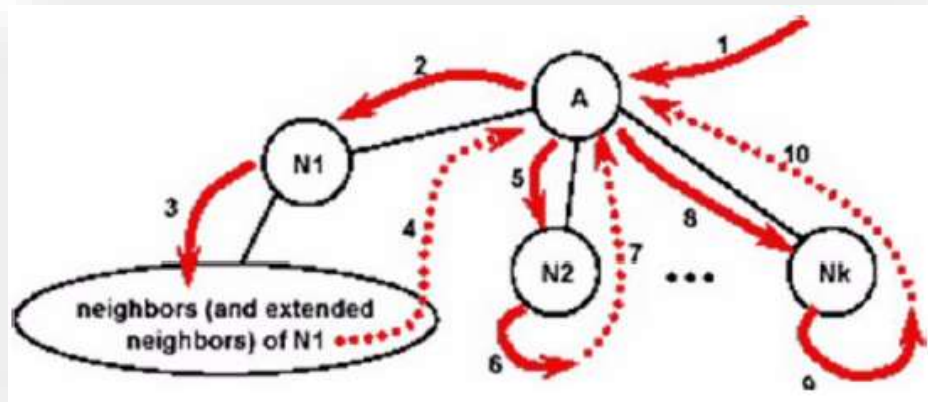
4.3 图的搜索(遍历)(cont.)

深度优先搜索(Depth First Search, DFS)：类似树结构先序遍历

- 设图G的初态是所有顶点都“未访问过(False)”，在G中任选一个顶点 v 为初始出发点(源点)，深度优先搜索定义为：
 - ① 首先访问出发点 v ，并将其标记为“访问过(True)”；
 - ② 然后，从 v 出发，依次考察与 v 相邻（邻接于或邻接到）顶点 w ；若 w “未访问过(False)”，则以 w 为新出发点递归地进行深度优先搜索，直到图中所有与源点 v 有路径相通的顶点（亦称从源点可到达的顶点）均被访问为止；（从源点出发的一次先深搜索）
 - ③ 若此时图中仍有未被访问过的顶点，则另选一个“未访问过”的顶点作为新搜索起点，重复上述过程，直到图中所有顶点都被访问过为止。

- **时间复杂度：**

- 邻接表： $O(n+e)$
- 邻接矩阵： $O(n^2)$

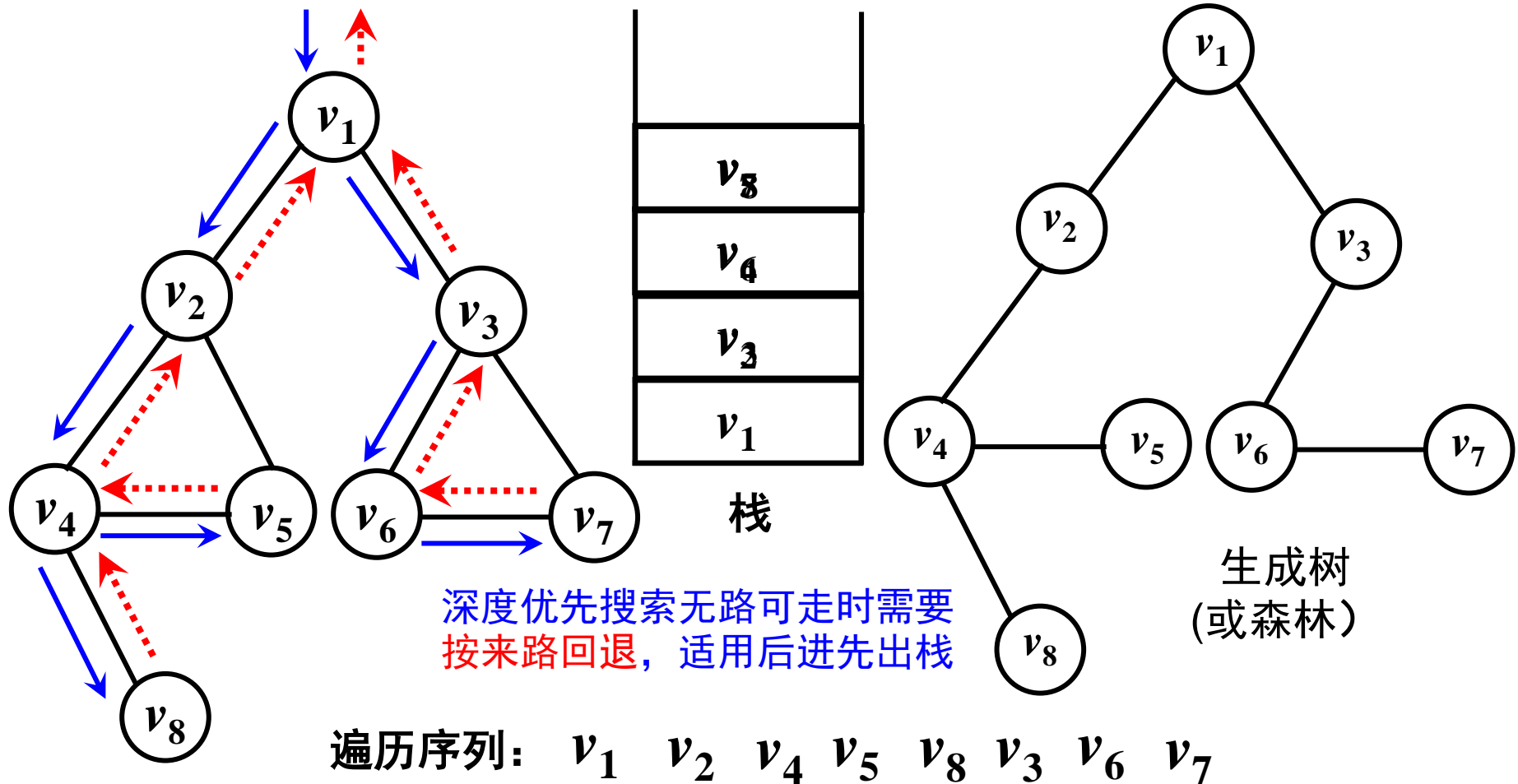




4.3 图的搜索(遍历)(cont.)

深度优先遍历：示例

– 深度优先遍历序列? 入栈序列? 出栈序列?





4.3 图的搜索(遍历)(cont.)

- 深度优先遍历**特点**
 - 为递归定义，尽可能对**纵深**方向上进行搜索，故称**先深**或**深度优先搜索**。
- **先深或深度优先编号**
 - 搜索过程中，根据访问顺序给顶点进行的编号，称为**先深**或**深度优先编号**。
- **先深序列或DFS序列：**
 - 先深搜索过程中，根据访问顺序得到的顶点序列，称为**先深序列**或**DFS序列**。
- **生成树（森林）：**
 - 由原图的**所有顶点**和搜索过程中**经过的边**构成的子图。
- **先深搜索结果不唯一**
 - 图的**DFS序列**、**先深编号**和**生成森林**不唯一。



4.3 图的搜索(遍历)(cont.)

- 深度优先遍历主算法：

bool visited[NumVertices]; //访问标记数组是全局变量

int dfn[NumVertices]; //顶点的先深编号

void **DFS**Traverse (AdjGraph G) //主算法

/* 邻接表表示的图G。以邻接矩阵表示G时，算法完全相同 *

{ int **count** = 1;

for (int i = 0; i < G.n; i++)

visited [i] =FALSE; //标志数组初始化

for (int i = 0; i < G.n; i++)

if (! visited[i])

DFSX (G, i); //从顶点 i 出发进行一次搜索， **DFSX**(G, i)

}



4.3 图的搜索(遍历)(cont.)

- 从一个顶点出发的一次深度优先遍历算法：

- 实现步骤：

0. 所有顶点标记为未访问 $\text{visited}[v] = \{0, \dots\}$;

1. 访问顶点 v ; $\text{visited}[v] = 1$;

2. $w =$ 顶点 v 的第一个邻接点;

3. while (w 存在)

- 3.1 if (w 未被访问)

- 从顶点 w 出发递归地执行此算法;

- 3.2 $w =$ 顶点 v 的下一个邻接点;



4.3 图的搜索(遍历)(cont.)

- 从一个顶点出发的一次深度优先遍历算法：邻接表实现

void **DFS1** (AdjGraph* G, int i) //递归实现

//以 v_i 为出发点，对邻接表表示的图G进行先深搜索

```
{   EdgeNode *p;
```

```
    cout<<G→vexlist[i].vertex;    //访问顶点 $v_i$ ;
```

```
    visited[i]=TRUE;                //标记 $v_i$ 已访问
```

```
    dfn[i]=count++;                 //对 $v_i$ 进行编号
```

```
    p=G→vexlist[i].firstedge;       //取 $v_i$ 边表的头指针
```

```
    while( p ) { //依次搜索 $v_i$ 的邻接点 $v_j$ , 这里 $j=p→adjvex$ 
```

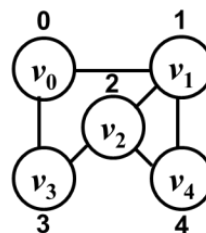
```
        if( !visited[ p→adjvex ] ) //若 $v_j$ 尚未访问
```

```
            DFS1(G, p→adjvex); //则以 $v_j$ 为出发点递归先深搜索
```

```
            p=p→next;
```

```
    }
```

```
} //DFS1
```





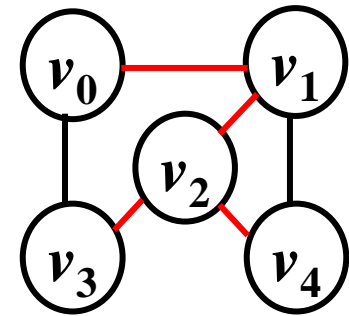
4.3 图的搜索(遍历)(cont.)

- 从一个顶点出发的一次深度优先遍历算法：邻接矩阵实现

```
void DFS2(MTGraph *G, int i)
```

```
//以 $v_i$ 为出发点对邻接矩阵表示的图G进行深度优先搜索
```

```
{  int j;  
    cout<<G→vexlist[i];  //访问定点 $v_i$   
    visited[i]=TRUE;      //标记 $v_i$ 已访问  
    dfn[i]=count;         //对 $v_i$ 进行编号  
    count ++;             //下一个顶点的编号  
    for( j=0; j<G→n; j++ ) //依次搜索 $v_i$ 的邻接点  
        if ( (G→edge[i][j] == 1)&&!visited[i] )  
            //若 $v_j$ 尚未访问，递归遍历  
                DFS2( G, j );  
}
```



	v_0	v_1	v_2	v_3	v_4	
v_0	0	1	0	1	0	v_0
v_1	1	0	1	0	1	v_1
v_2	0	1	0	1	1	v_2
v_3	1	0	1	0	0	v_3
v_4	0	1	1	0	0	v_4



广度优先搜索

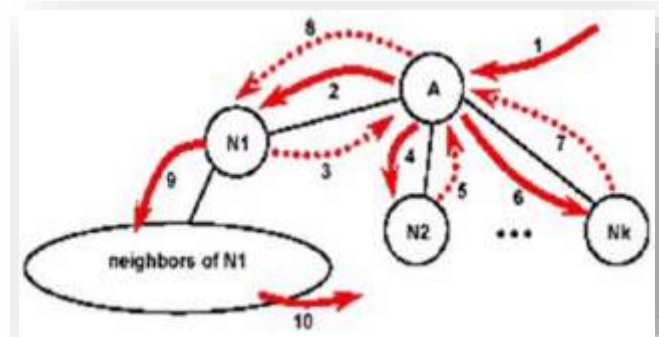
(Breadth First Search, BFS)



4.3 图的搜索(遍历)(cont.)

广度优先搜索(Breadth First Search, BFS): 类似于**树层序**遍历

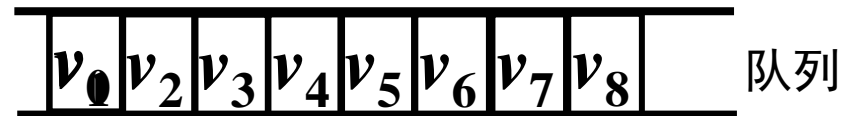
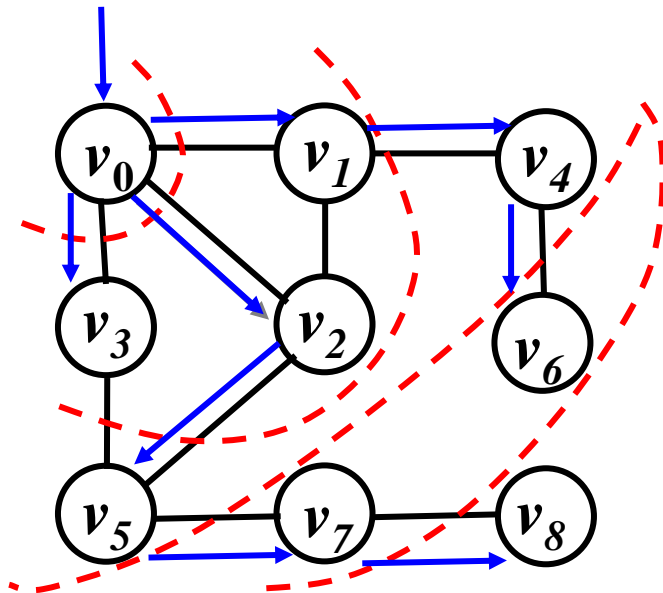
- 设**图G**的**初态**是所有顶点都“未访问过(False)”，在G中**任选**一个顶点 v 为**源点**，则广度优先搜索可定义为：
 - ① 首先访问出发点 v ，并将其标记为“访问过 (True)”；
 - ② 接着依次访问所有**与 v 相邻**的顶点 $w_1, w_2 \dots w_t$ ；
 - ③ 然后依次访问**与 $w_1, w_2 \dots w_t$ 相邻**的所有**未访问**的顶点；
 - ④ 依次类推，直至图中**所有与源点 v 有路相通的顶点**都已访问过为止；(从源点出发的一次**先广搜索**)
 - ⑤ 此时，从 v 开始的搜索结束，若G是连通的，则遍历完成；否则在G中**另选一个尚未访问**的顶点作为新源点，继续上述搜索过程，直到G中的**所有顶点**均已访问为止。
- **时间复杂度**:
 - 邻接表: $O(n+e)$; 邻接矩阵: $O(n^2)$





4.3 图的搜索(遍历)(cont.)

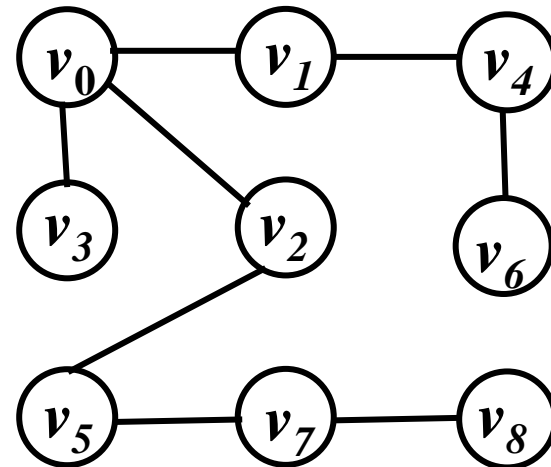
- 广度优先遍历示例
- 广度优先遍历序列?入队序列?出队序列?



广度优先需要保证先访问顶点的未访问邻接点先访问, 适用先进先出队列

遍历序列: $v_0 \ v_1 \ v_2 \ v_3 \ v_4 \ v_5 \ v_6 \ v_7 \ v_8$

生成树
(或森林)





4.3 图的搜索(遍历)(cont.)

- 广度优先遍历**特点**:
 - 尽可能**横向上**进行搜索，并使“先被访问的顶点的邻接点”先于“后被访问的顶点的邻接点”被访问，故称**先广搜索**或**广度优先搜索**。
- **先广或广度优先编号**:
 - 搜索过程中，根据访问顺序给顶点进行的编号，称为**先广或广度优先编号**
- **先广序列或BFS序列**:
 - 先广搜索过程中，根据访问顺序得到的顶点序列，称为**先广序列**或**BFS序列**。
- **生成树（森林）**:
 - 原图**所有顶点**和搜索过程中**经过的边**构成的子图。
- **先广搜索结果不唯一**:
 - 图的**BFS序列**、**先广编号**和**生成森林**不唯一。



4.3 图的搜索(遍历)(cont.)

- 广度优先遍历主算法：

```
bool visited[NumVertices];           //访问标记数组是全局变量
int dfn[NumVertices];                //顶点的先深编号
void BFSTraverse (AdjGraph G)        //主算法：递归实现
/* 邻接表表示图G；以邻接矩阵表示G时，算法完全相同 */
{  int count = 1;
   for ( int i = 0; i < G.n; i++ )
       visited [i] =FALSE; //标志数组初始化
   for ( int i = 0; i < G.n; i++ )
       if ( ! visited[i] )
           BFSX ( G, i ); //从顶点 i 出发的一次搜索， DFSX (G, i)
}
```




4.3 图的搜索(遍历)(cont.)

- 从一个顶点出发的一次广度优先遍历算法:

- 实现步骤:

1. 初始化队列Q;
2. 访问顶点v; $visited[v]=1$; 顶点v入队Q;
3. while (队列Q非空)
 - 3.1 v=队列Q的队头元素出队;
 - 3.2 w=顶点v的第一个邻接点;
 - 3.3 while (w存在) //访问v的所有邻接点
 - 3.3.1 如果w 未被访问, 则
访问顶点w; $visited[w]=1$;
顶点w入队列Q;
 - 3.3.2 w=顶点v的下一个邻接点;



4.3 图的搜索(遍历)(cont.)

从一个顶点出发的一次广度优先遍历算法：邻接表实现

```
void BFS1 (AdjGraph *G, int k)    //这里没有进行先广编号
{
    int i; EdgeNode *p; QUEUE Q;  MAKENULL(Q);
    cout << G→vexlist[ k ].vertex;  visited[ k ] = TRUE;
    ENQUEUE (k, Q);                //进队列
    while ( ! Empty (Q) ) {        //队空搜索结束
        i=DEQUEUE(Q);              //vi出队
        p =G→vexlist[ i ].firstedge; //取vi的边表头指针
        while ( p ) {              //若vi邻接点 vj (j= p→adjvex)存在，依次搜索
            if ( !visited[ p→adjvex ] ) { //若vj未访问过
                cout << G→vexlist[ p→adjvex ].vertex; //访问vj
                visited[ p→adjvex ]=TRUE;             //给vj作访问过标记
                ENQUEUE ( p→adjvex , Q );             //访问过的vj入队
            }
            p = p→next; //找vi的下一个邻接点
        }              // 重复检测 vi的所有邻接顶点
    }                  //外层循环，判队列空否
} //以vk为出发点，对用邻接表表示的图G进行先广搜索
```



4.3 图的搜索(遍历)(cont.)

从一个顶点出发的一次广度优先遍历算法：邻接矩阵实现

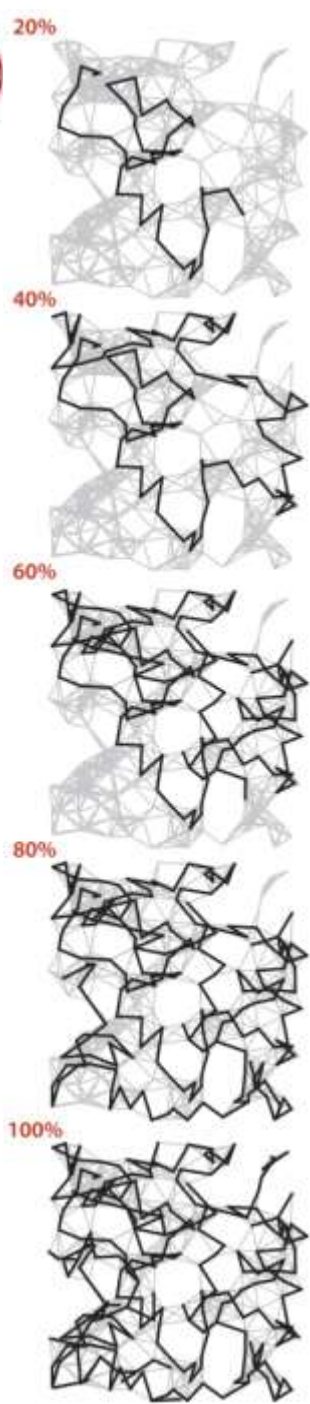
void **BFS2** (**MTGraph** *G, int k) //这里没有进行先广编号

```
{  int i, j; QUEUE Q;  MAKENULL(Q);
    cout << G→vexlist[ k ];      //访问 $v_k$ 
    visited[ k ] = TRUE;          //给 $v_k$ 作访问过标记
    ENQUEUE (k, Q);               //  $v_k$ 进队列
    while ( ! Empty (Q) ) {      //队空时搜索结束
        i=DEQUEUE(Q);            //  $v_i$ 出队
        for(j=0; j<G→n; j++) {    //依次搜索 $v_i$ 的邻接点  $v_j$ 
            if ( G→edge[ i ][ j ] ==1 && !visited[ j ] ) { //若 $v_j$ 未访问过
                cout << G→vexlist[ j ]; //访问 $v_j$ 
                visited[ j ]=TRUE;      //给 $v_j$ 作访问过标记
                ENQUEUE ( j , Q );      //访问过的 $v_j$ 入队
            }
        } //重复检测  $v_i$ 的所有邻接顶点
    } //外层循环，判队列空否
} // 以 $v_k$ 为出发点时对用邻接矩阵表示的图G进行先广搜索
```

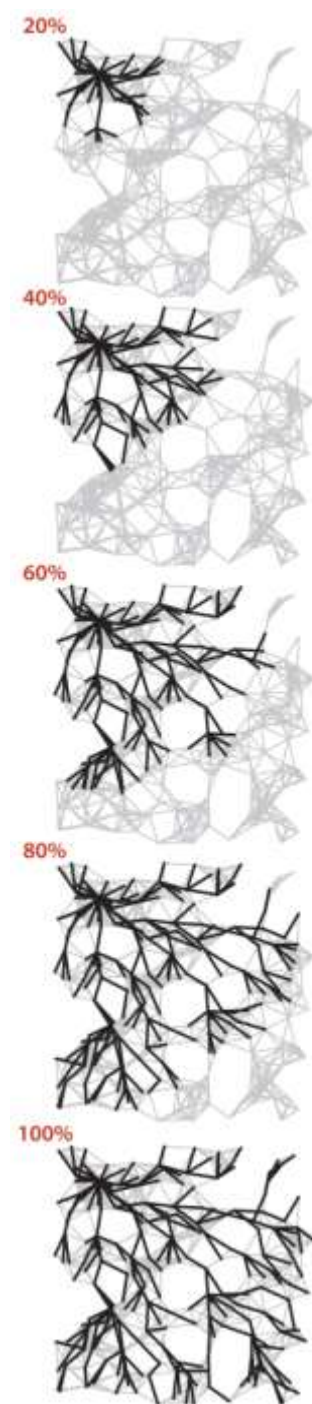
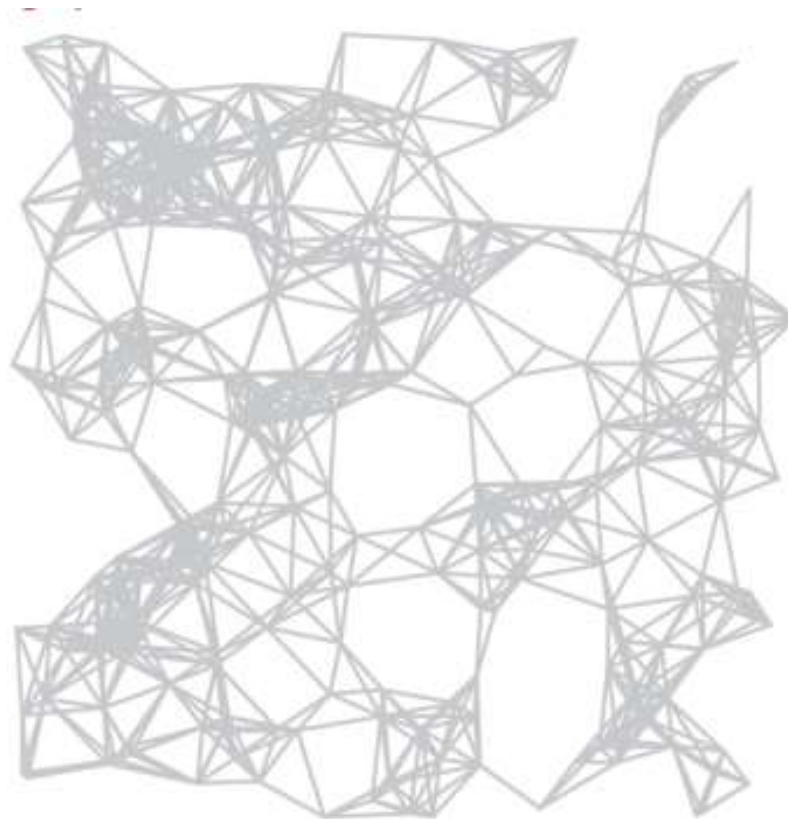


示例

图：250个结点，1273条边



先深遍历



先广遍历

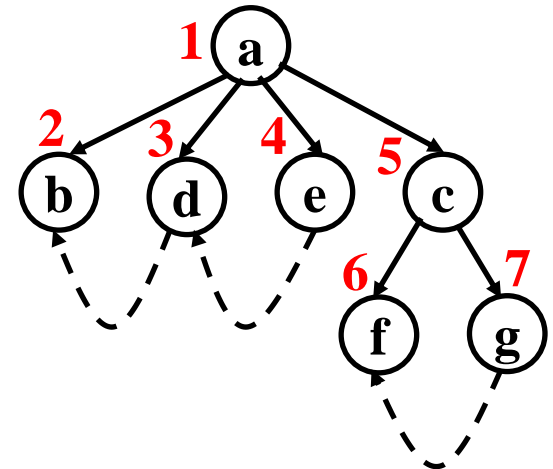
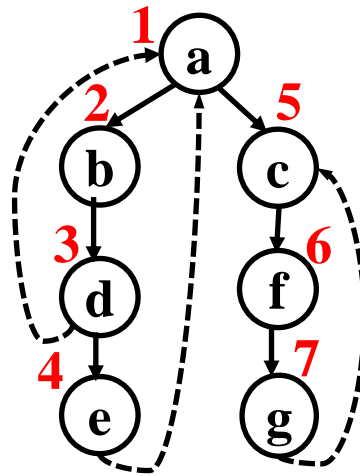
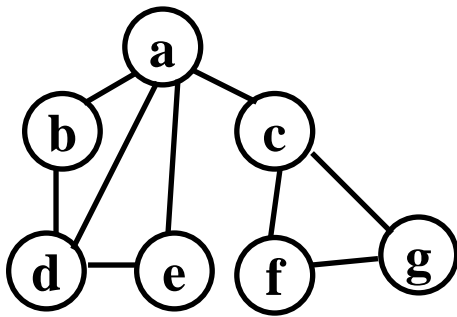


无向图(的搜索)及其应用

先深生成森林和先广生成森林

- 搜索产出结果

- 产生先深或先广生成森林、顶点线性序列、先深或先广编号
- 连通图：一个生成树
- 非连通图：生成森林，每棵树是原图的连通子图(连通分量)
- 树边与非树边





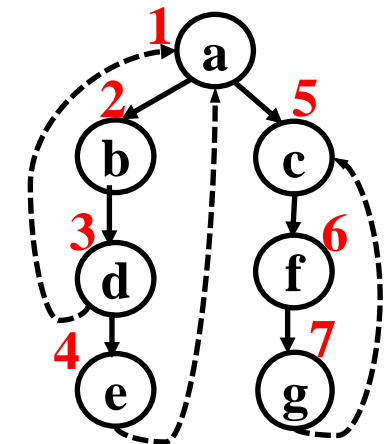
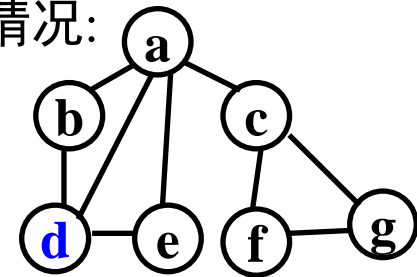
无向图(的搜索)及其应用

深度优先搜索过程对边分类：边分成两类

- **树边**：在搜索过程中所经过的边；
- **回退边**：图中的其它边
- **特点**：树边是从先深编号较小的指向较大的顶点，回退边相反。
- **如何在搜索过程中区分树边和回退边？**

– 设 v 是刚访问过的顶点True，下面搜索的 w 有**三种**情况：

- ① w 是False，则 (v,w) 是树边，将其加入T；(d,e)
- ② w 是True,且 w 是 v 的父亲，则 (w,v) 是树边，但是第二次遇到,不再加入T；(右图：d,b)
- ③ w 是True且 w 不是 v 的父亲，则 (v,w) 是回退边。
(右图：d,a)



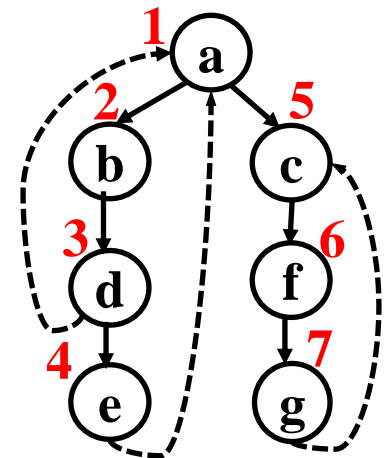
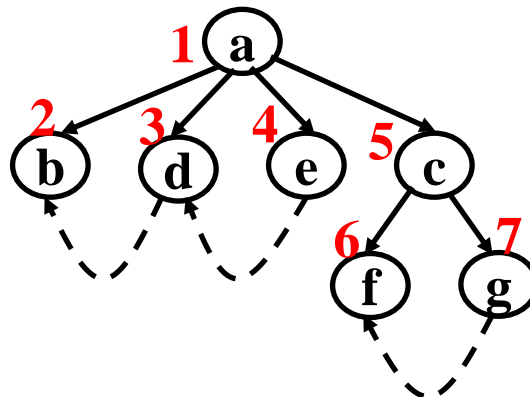
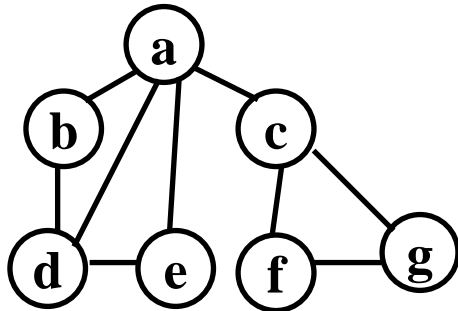
- **结论**：若G中存在**环路**，则在先深搜索过程中必遇到**回退边**；反之亦然。



无向图(的搜索)及其应用

广度优先搜索过程对边分类：边分为两类

- 两类边：
 - 树边：在搜索过程中经过的边
 - 横边：图中的其它边
- 特点：
 - 树边是从先深编号较小的指向较大的顶点；
 - 而横边不一定与之相反，但可规定：大→小。
- 结论：若G中存在环路，则在先广搜索过程中必遇到横边；反之亦然。





无向图（的搜索）及其应用

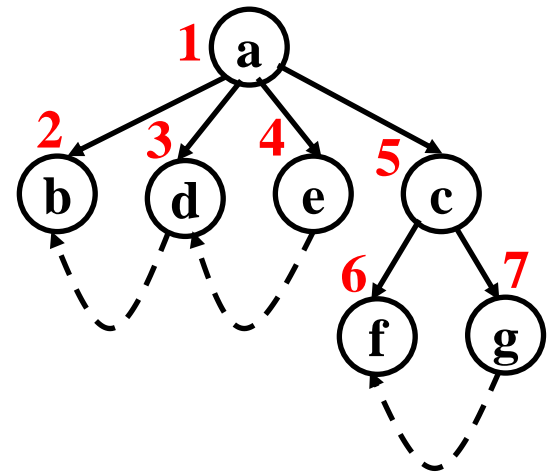
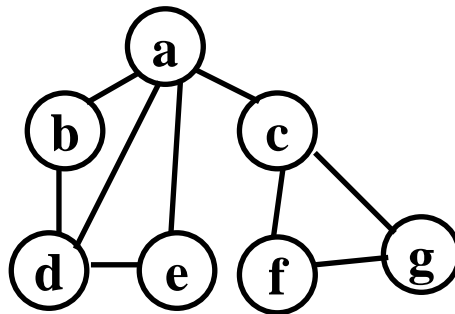
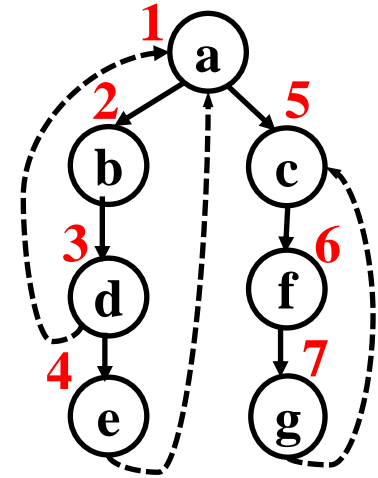
• 无向图连通性

– 不连通

- 求连通分量个数
- 求出每个连通分量

– 连通

- 判断是否有环路
- 求带权连通图的最小生成树
- 判断是否是双连通
- 求关节点和双连通分量





4.4 最小生成树算法

• 生成树及其代价

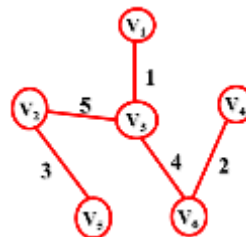
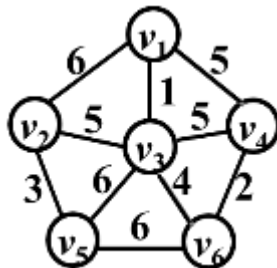
- 设 $G = (V, E)$ 是一个无向连通图， E 中每一条边 (u, v) 上的权值 $c(u, v)$ ，称为 (u, v) 的边长。
- 图 G 的生成树 (Spanning Tree)：连接 V 中所有顶点的一棵开放树 (Free Tree, 无环路的无向图)。
- 生成树的代价：图 G 生成树各边权值 (边长) 之和。

• 最小生成树 (Minimum-cost Spanning Tree, MST)

- 在图 G 所有生成树中，代价最小的生成树称为最小生成树

• 最小生成树应用广泛

- 例如，在 n 个教室之间建局域网络，至少需要架设 $n-1$ 条通信线路，而每两个教室之间的距离可能不同，从而架设通信线路的造价就是是不一样的，那么如何设计才能使得总造价最小？





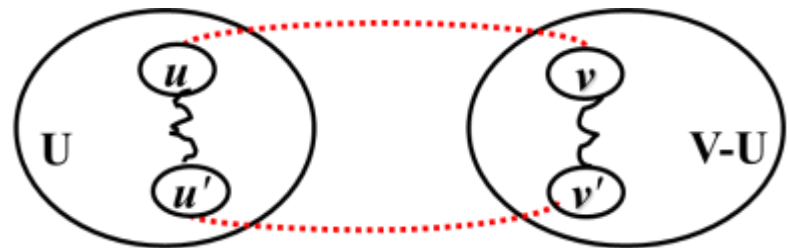
4.4 最小生成树算法(cont.)

- 最小生成树性质：贪心选择性

- 假设 $G = (V, E)$ 是一个连通图， U 是顶点 V 的一个非空真子集。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。
- 此性质保证了 *Prim* 和 *Kruskal* 贪心算法的正确性

- MST性质证明

- [反证]: 假设 G 的任何一棵最小生成树都不包含 (u, v) , 设 T 是连通图的一棵最小生成树, 当将边 (u, v) 加入到 T 中时, 由生成树的定义, T 必包含一条 (u, v) 的回路。
- 另一方面, 由于 T 是生成树, 则在 T 中必存在另一条边 (u', v') 且 u 和 u' 、 v 和 v' 之间均有路径相通。
- 删去边 (u', v') 便可消去上述回路, 同时得到另一棵最小生成树 T' 。
- 但因为 (u, v) 的代价不高于 (u', v') , 则 T' 的代价亦不高于 T , T' 是包含 (u, v) 的一棵最小生成树。





4.4 最小生成树算法

普里姆(Prim)算法



4.4 最小生成树算法(cont.)

- 普里姆(Prim)算法

- 基本思想

- ① 首先从集合 V 中任取一顶点(如顶点 v_1)放入集合 U 中。
这时 $U=\{v_1\}$, $TE=\{\}$
- ② 然后找出权值最小的边 (u, v) , 且 $u \in U$, $v \in (V-U)$, 将边加入 TE , 并将顶点 v 加入集合 U
- ③ 重复上述操作直到 $U=V$ 为止。这时 TE 中有 $n-1$ 条边,
 $T=(U, TE)$ 就是 G 的一棵最小生成树。

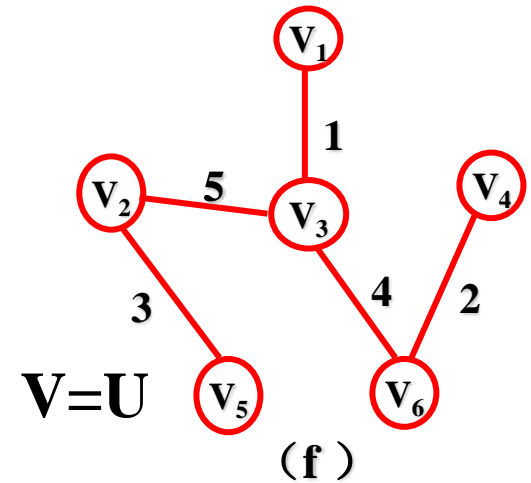
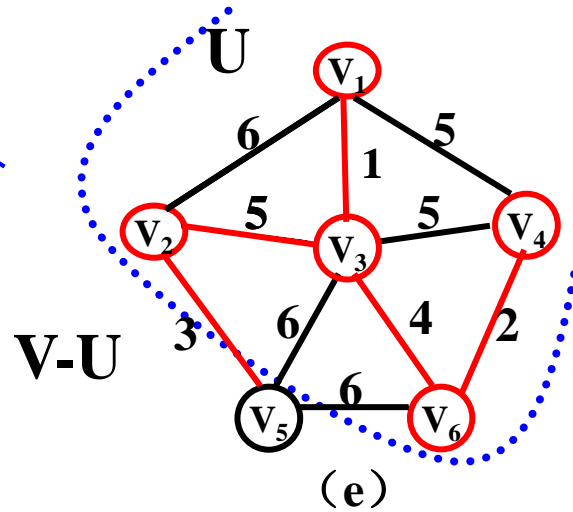
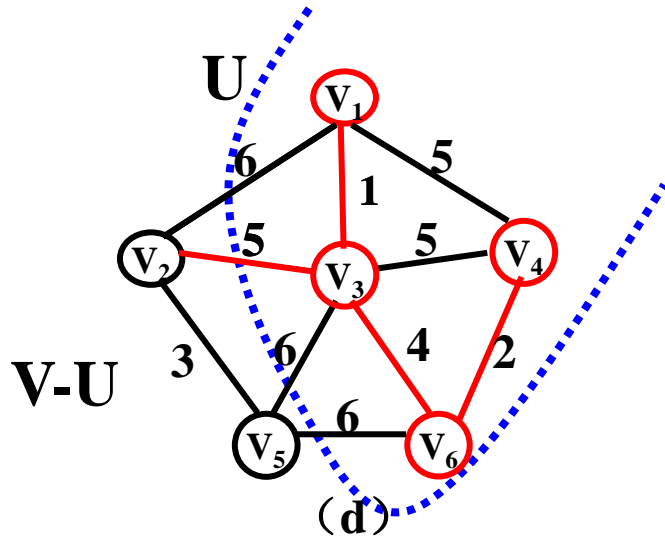
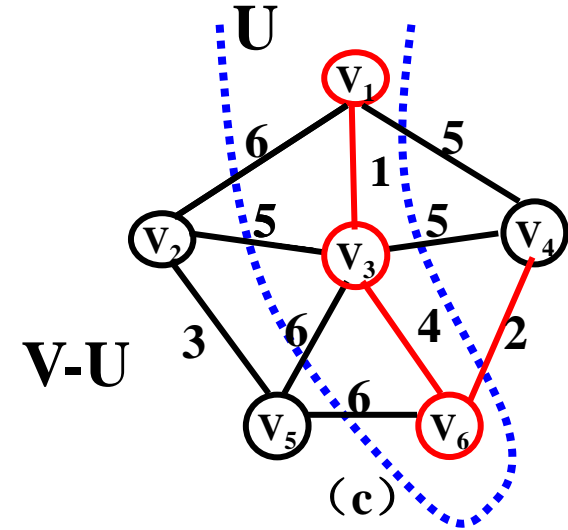
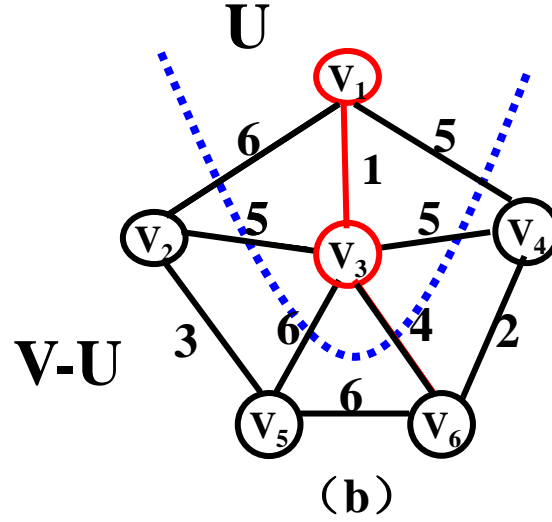
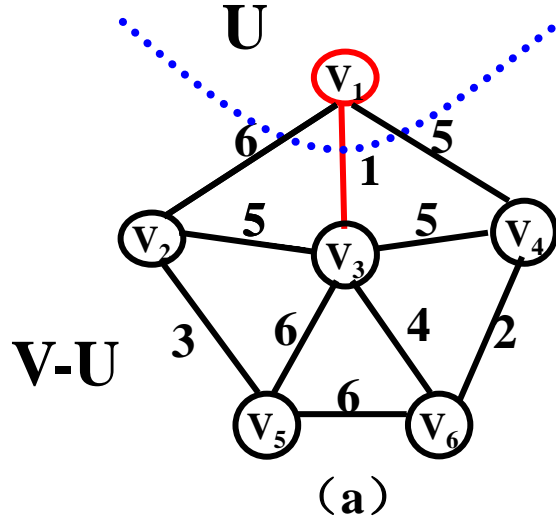
- 如何找到连接 U 和 $V-U$ 的最短边?

- 利用MST性质, 构造候选最短边集;
- 对于 $V-U$ 中的每个顶点, 保存该顶点到 U 中各顶点的最短边。



4.4 最小生成树算法(cont.)

• 普里姆(Prim)算法：示例





4.4 最小生成树算法(cont.)

普里姆(Prim)算法：数据结构及其更新

- 数据结构

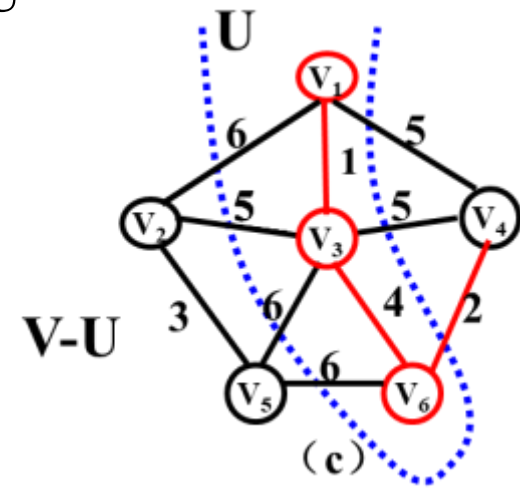
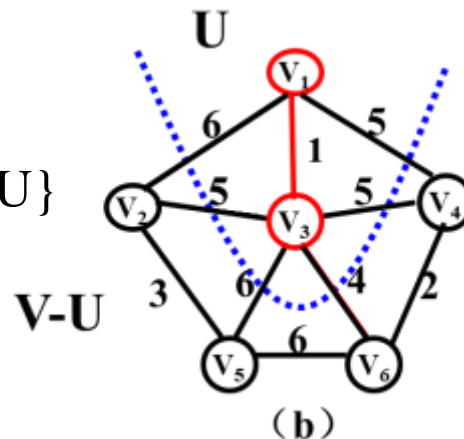
- 数组 **LOWCOST[n]**：用来保存**集合V-U中**各顶点与**集合U中**顶点**最短边的权值**，LOWCOST[v]=infinity表示顶点v**已加入**最小生成树中；
- 数组 **CLOSSET[n]**：用来保存**依附于该边的**（集合V-U中各顶点与集合U中顶点的最短边）**在集合U中的顶点**。

- 如何用数组 **LOWCOST[n]** 和 **CLOSSET[n]** 表示候选**最短边集**？

- LOWCOST[i] = w
 - CLOSSET[i] = k
- } 表示顶点 v_i 和顶点 v_k 之间的权值为 w ，
其中： $v_i \in V-U$ 且 $v_k \in U$

- 如何更新？

$$\text{LOWCOST}[j] = \min \{ \text{cost}(v_k, v_j) \mid v_j \in U \}$$
$$\text{CLOSSET}[j] = k$$



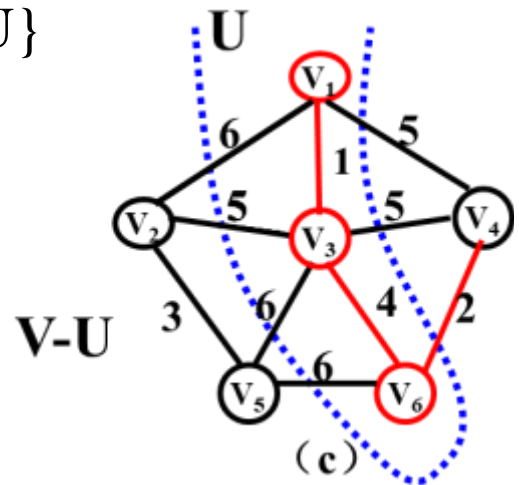
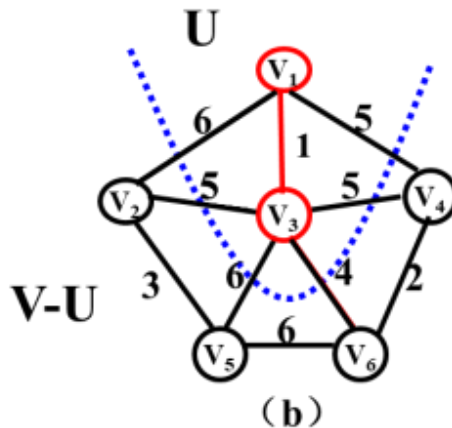


4.4 最小生成树算法(cont.)

普里姆(Prim)算法：实现步骤

1. 初始化两个辅助数组LOWCOST和CLOSSET;
2. 输出顶点 v_1 ，将顶点 v_1 加入集合U中;
3. 重复执行下列操作n-1次
 - 3.1 在LOWCOST中选取最短边，取CLOSSET中对应的顶点序号k;
 - 3.2 输出顶点k和对应的权值;
 - 3.3 将顶点k加入集合U中;
 - 3.4 调整数组LOWCOST和CLOSSET;

$$\text{LOWCOST}[j] = \min \{ \text{cost}(v_k, v_j) \mid v_j \in U \}$$
$$\text{CLOSSET}[j] = k$$

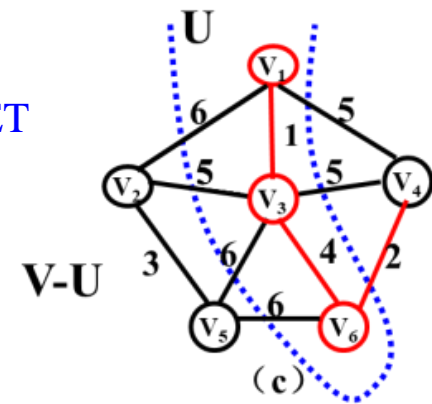
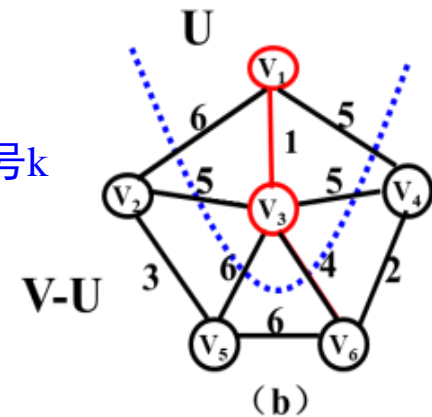




4.4 最小生成树算法(cont.)

普里姆(Prim)算法的：实现

```
void Prim(Costtype C[n+1][n+1] )
{
    costtype LOWCOST[n+1]; int CLOSSET[n+1]; int i, j, k; costtype min;
    for( i=2; i<=n; i++ ) //初始化数组LOWCOST和数组CLOSSET
    {
        LOWCOST[i] = C[1][i];    CLOSSET[i] = 1;    }
    for( i = 2; i <= n; i++ ) //n-1次
    {
        min = LOWCOST[i];
        k = i;
        //3.1在LOWCOST中选最短边，记CLOSSET中对应的顶点序号k
        for( j = 2; j <= n; j++ )
            if ( LOWCOST[j] < min )
                { min = LOWCOST[j];    k=j; }
        //3.2输出最小生成树的边信息
        cout << "(" << k << "," << CLOSSET[k] << ")" << endl;
        LOWCOST[k] = infinity; //3.3把顶点k加入最小生成树中
        for ( j = 2; j <= n; j++ ) //3.4调整数组LOWCOST和CLOSSET
            if ( C[k][j] < LOWCOST[j] && LOWCOST[j] < infinity )
                { LOWCOST[j]=C[k][j]; CLOSSET[j]=k; }
    }
} // 时间复杂度：  $O(|V|^2)$ 
```





4.4 最小生成树算法

克鲁斯卡尔(Kruskal)算法



4.4 最小生成树算法(cont.)

克鲁斯卡尔(Kruskal)算法

- 基本思想：

- 设无向连通网为 $G=(V, E)$ ，令 G 的最小生成树为 $T=(U, TE)$ ，其初值为 $U=V$ ， $TE=\{\}$ ，每个顶点看成一个连通分量。
- 按照边权值由小到大的顺序，依次考察 G 边集 E 中的各条边。
- 若被考察的边连接两个不同连通分量，则将此边作为最小生成树的边加入到 T 中，同时把两个连通分量连接为一个连通分量；
- 若被考察的边连接同一个连通分量，则舍去此边，避免产生回路；
- 如此下去，当 T 中的连通分量个数为1时，此连通分量便为 G 的一棵最小生成树。



4.4 最小生成树算法(cont.)

克鲁斯卡尔(Kruskal)算法

- 实现步骤:

1. 初始化: $U=V$; $TE=\{ \}$;
 2. 循环直到T中的连通分量个数为1
 - 2.1 在E中选择最短边 (u, v) ;
 - 2.2 如果顶点 u 与 v 位于T的两个不同连通分量, 则
 - 2.2.1 将边 (u, v) 并入TE;
 - 2.2.2 将这两个连通分量合为一个;
 - 2.3 在E中标记边 (u, v) , 使得 (u, v) 不参加后续最短边选取
- 时间复杂度: $O(|E| \cdot \log |E|)$



4.4 最小生成树算法(cont.)

克鲁斯卡尔(Kruskal)算法

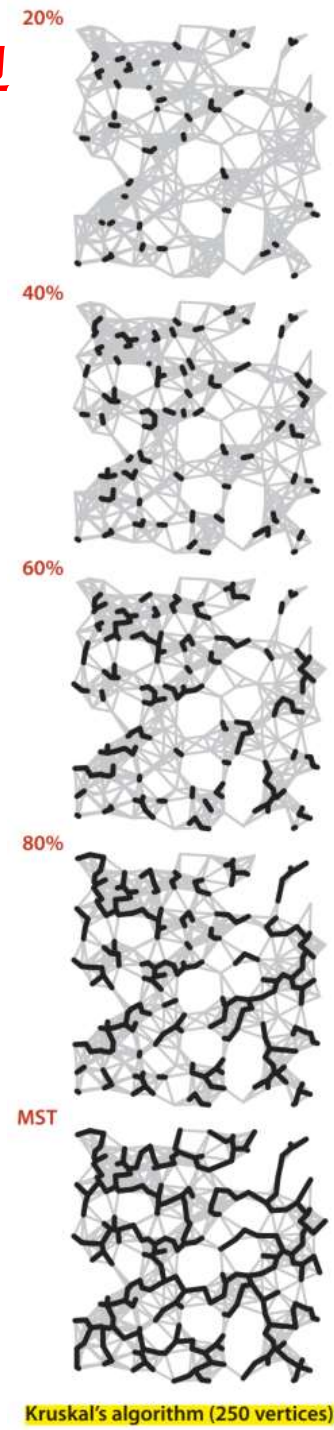
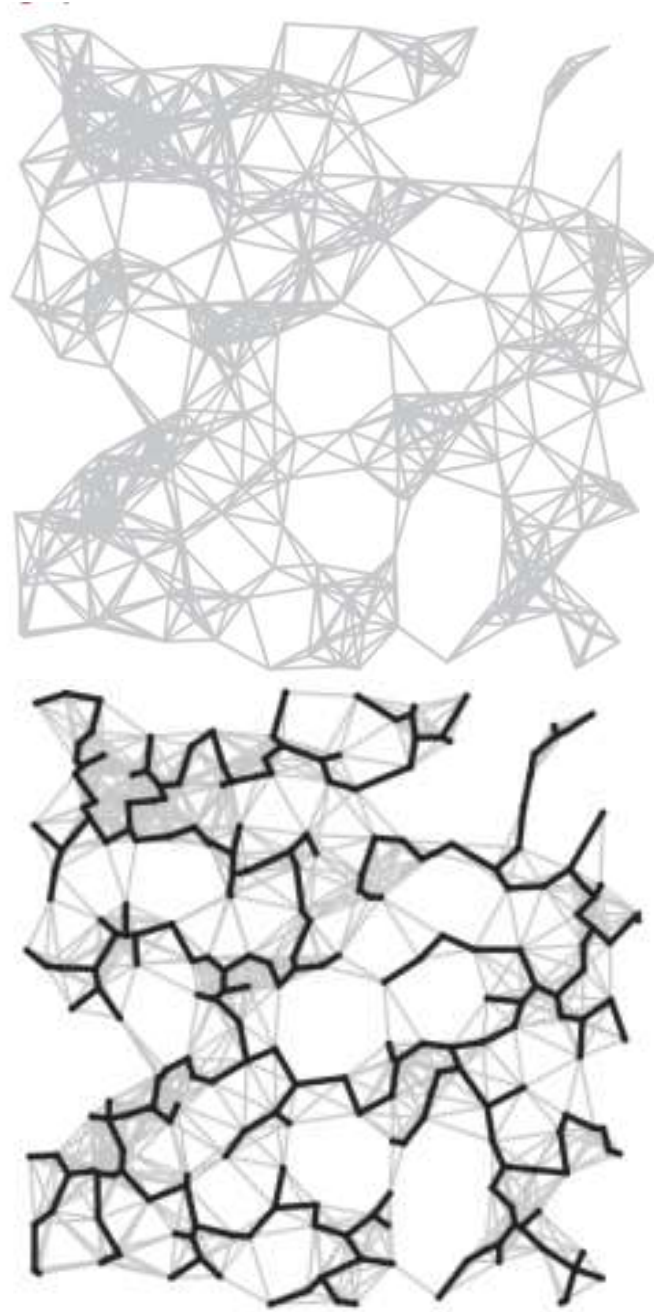
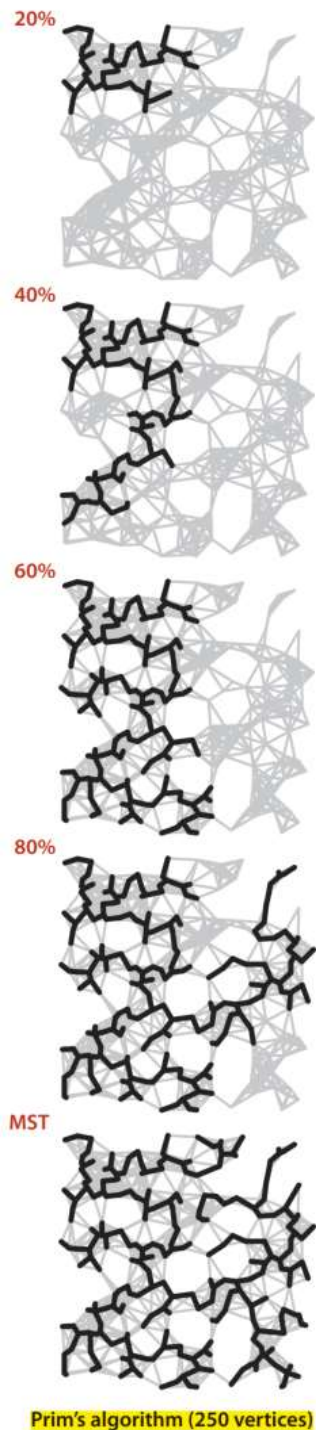
```
void Kruskal_Min_Tree(EdgeSet edges, int vexnum, int arcnum)
{
    int bnf, edf;  int parents[100];
    Sort(edges);           //按照权值大小排序,
    for(int i=0;i<vexnum;i++) //初始化parent[]数组 (连通分量)
        parents[i]=0;
    for(i=0;i<arcnum;i++) {
        bnf=Find(edges[i].begin,parents);
        edf=Find(edges[i].end,parents);
        if(bnf!=edf) {           // 非同 一连通分量
            parents[bnf]=edf;    // 连成一个连通子图
            cout<<('<<edges[i].begin<<','<<edges[i].end<<','<<edges[i].cost<<')<<endl;
        }
    }
} // 时间复杂度: O(|E|*log|E|)
```

	cost	begin	end
0	-	-	-
1			
2			

edges



示例图：250个结点，1273条边

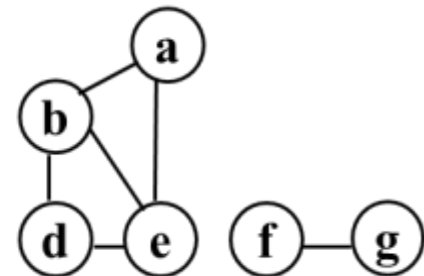
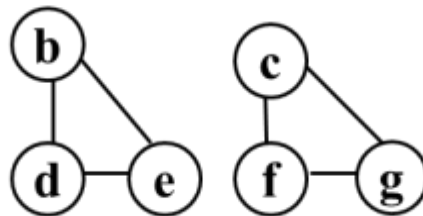
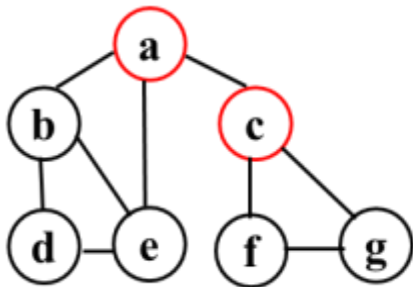




4.5 无向图双(重)连通性算法

相关概念： 设 $G=(V, E)$ 是一个连通图

- **关节点**(*Articulation Point*)： 一个顶点 a 称为**连通无向图**的**关节点**，若在删去顶点 a 以及与之相邻的边之后，图 G 被分割成两个或两个以上的**连通分量**，也称**割点**(*Cut-vertex*)。如图a, c。
- **双连通图**(*Biconnected Graph*)： 没有关节点的**连通图**称为**双连通图**。
- 在双连通图上，**任何一对**顶点之间**至少**存在有**两条路径**，在删去某个顶点及与该顶点相关联的边时，也**不破坏**图的连通性。
- 双连通的无向图是连通的，但连通的无向图**未必**双连通。





4.5 无向图双连通性算法(cont.)

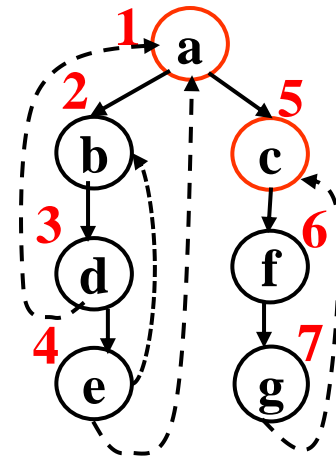
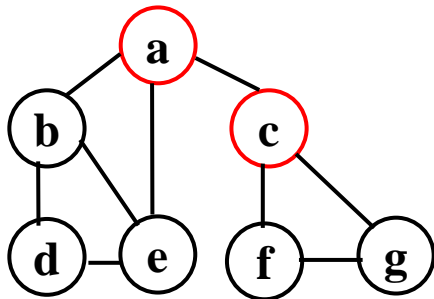
- **等价边**：称连通图 $G=(V, E)$ 的边 e_1 和 e_2 是等价的，若 $e_1=e_2$ 或者有一条**环路**包含 e_1 又包含 e_2 。
- 设 V_i 是**等价边集** E_i 中各边所连接的**顶点集** ($1 \leq i \leq k$)，每个图 $G_i = (V_i, E_i)$ 叫做 G 的一个**双连通分量** (*Biconnected Component*)。
- **双连通分量 G_i 的性质**：
 - 性质1 G_i 是双连通的 ($1 \leq i \leq k$)
 - 性质2 对所有的 $i \neq j$, $V_i \cap V_j$ **最多** 包含一个顶点
 - 性质3 v 是 G 的**关节点**，**当且仅当** $v \in V_i \cap V_j$ ，存在 $(i \neq j)$



4.5 无向图双连通性算法(cont.)

关节点性质：两类关节点

- 对图进行一次先深搜索便可求出所有关节点，由此可判别图是否重连通。由深度优先生成树可得出两类关节点的特性：
 - 第一类关节点：若生成树的根有两株或两株以上子树，则此根结点必为关节。因图中不存在连接不同子树中顶点的边，若删去根顶点，则生成树变成生成森林。
 - 第二类关节点：若生成树中非叶顶点 v ，其子树的根和子树中其它结点，均没有指向 v 的祖先的回退边，则 v 是关节点（如图中结点 c ）。因为删去 v ，则其子树和图其它部分被分割开来。





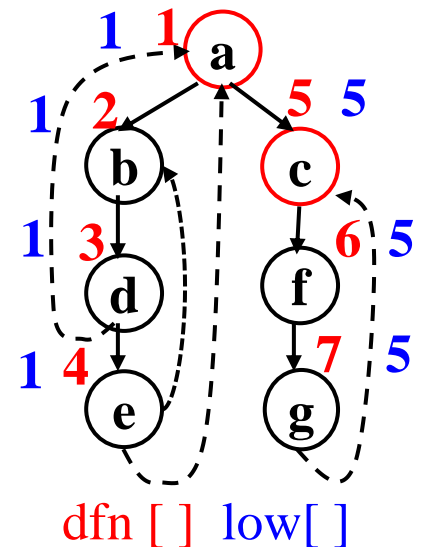
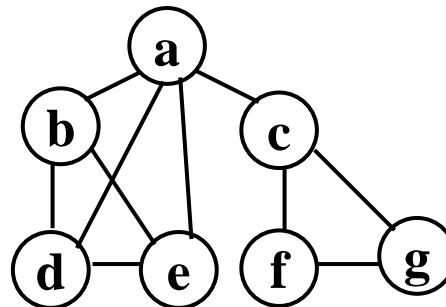
4.5 无向图双连通性算法(cont.)

low[v]编号：顶点的可达最小先深编号

- 设对连通图 $G=(V,E)$ 进行先深搜索的先深编号为 $dfn[v]$ ，先深生成树为 $S=(V,T)$ ， B 是回退边之集。对每个顶点 v ， $low[v]$ 定义如下：

$$low[v] = \min \left\{ \begin{array}{l} dfn[v], \quad (v, w) \in B, w \text{ 是顶点 } v \text{ 在先深生成树上} \\ \quad \quad \quad \text{由回退边连接的祖先结点;} \\ dfn[w], \quad (v, y) \in T, y \text{ 是顶点 } v \text{ 在先深生成树上的} \\ low[y] \quad \quad \quad \text{孩子顶点。} \end{array} \right.$$

- 若某个顶点 v ，存在孩子结点 y ，且 $low[y] \geq dfn[v]$ ，（不小于）则 v 必为关节点，例如，顶点 c 。
- 因为， y 及其子孙均无指向 v 的祖先的回退边。





Robert Endre Tarjan

Robert Endre Tarjan

- A.M. Turing Award Winner, 1986
 - With John E Hopcroft,
 - for fundamental achievements in the design and analysis of algorithms and data structures.
- BIRTH:
 - April 30, 1948, Pomona, California
- EDUCATION:
 - B.S., California Institute of Technology (1969, Mathematics);
 - MS Stanford University (1971, Computer Science);
 - Ph.D., Stanford University (1972, Computer Science with minor in Mathematics)
- 发现了解决最近公共祖先（LCA）问题、强连通分量问题、双连通分量问题的高效算法，
- 参与了开发斐波那契堆、伸展树，分析并查集等工作。

“Make everything as simple as possible, but not simpler” - Einstein

Sometimes, simplicity is **critical**, not just desirable, notably in **concurrent** algorithms.

Making such an algorithm correct is **notoriously hard**. The simpler the underlying idea, the greater the chance of success.





4.5双连通性算法(cont.)

R.Tarjan算法：求关节点算法算步骤：

1. 计算先深编号：对图进行先深搜索，计算每个结点 v 的先深编号 $dfn[v]$ ，形成先深生成树 $S=(V,T)$ 。
2. 计算 $low[v]$ ：在先深生成树上按**后根遍历**顺序进行计算每个顶点 v 的 $low[v]$ ， $low[v]$ 取下述三个结点中的**最小者**：
 - (1) $dfn[v]$;
 - (2) $dfn[w]$ ，凡是有回退边 (v,w) 的**任何**结点 w ;
 - (3) $low[y]$ ，对 v 的**任何**儿子 y 。
3. 求关节点：
 - 3.1 树根是关节点，当且仅当它有**两个或两个以上的儿子**(**第一类关节点**);
 - 3.2 非树根结点 v 是关节点，当且仅当 v 有某儿子 y ，使 $low[y] \geq dfn[v]$ (**第二类关节点**)。

R.Tarjan算法是先深搜索，因此，**时间复杂度为**： $O(n+e)$ 或 $O(n^2)$

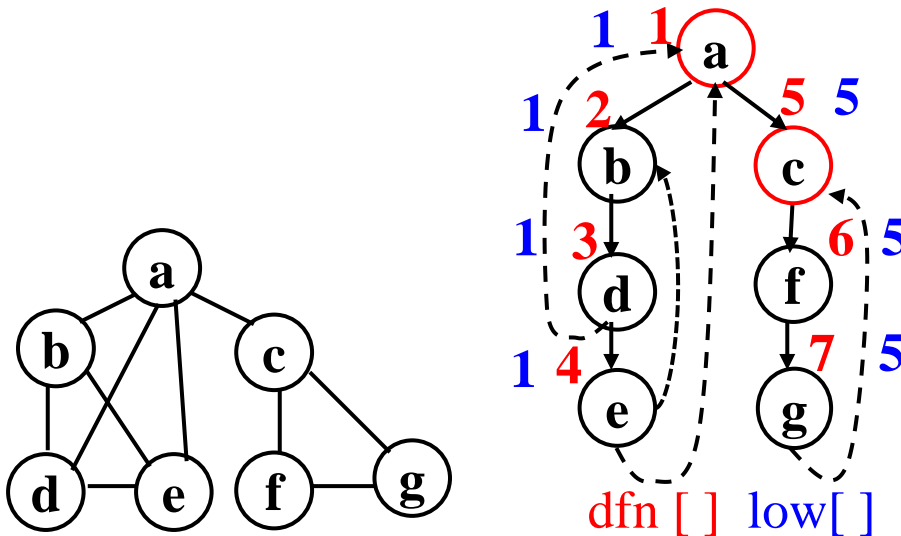


4.5双连通性算法(cont.)

示例：按后根遍历顺序计算 $low[v]$ 编号并求关节点

- 根结点a有两个孩子，是关节点；
- (c, f)是树边，即f是c的孩子且 $low[f] \geq dnf[c]$ ，所以c是关节点。

$$low[v] = \min \left\{ \begin{array}{l} dfn[v], \\ dfn[w], \\ low[y] \end{array} \right. \left. \begin{array}{l} (v, w) \in B, w \text{ 是顶点 } v \text{ 在先深生成树上} \\ \text{由回退边连接的祖先结点;} \\ (v, y) \in T, y \text{ 是顶点 } v \text{ 在先深生成树上的} \\ \text{孩子顶点.} \end{array} \right\}$$



序	结点	dfn[v]	dfn[w]	low[y]	min{ }
1	e	4	1,2		1
2	d	3	1	1	1
3	b	2		1	1
4	g	7	5		5
5	f	6		5	5
6	c	5		5	5
7	a	1		1,5	1



4.5无向图双连通性算法(cont.)

求关节点：R.Tarjan算法实现（同先深搜索算法）

```
void FindArticul(AdjGraph G)
```

```
{ /*连通图G以邻接表作存储结构，查找并输出G上全部关节点*/  
  count=1; /*变量count用于访问计数*/  
  dfn[0]=1; /*设定邻接表上0号顶点为生成树的根*/  
  for(i=1;i<G.n;++i) dfn[i]=0; /*其余顶点尚未访问，dfn[]兼职visited[]*/  
  p=G.vexlist[0].firstedge; v=p->adjvex;  
  DFSArticul(v); /*从顶点v出发深度优先查找关节点*/  
  if(count<G.n) { /*生成树的根至少有两棵子树，继续遍历*/  
    cout<<G.vexlist[0].vertex); /*根是关节点，输出*/  
    while(p->next) {  
      p=p->next;  
      v=p->adjvex;  
      if(dfn[v]==0) DFSArticul(v); /*顶点尚未访问，初始化dfn[i]=0*/  
    }//while  
  }//if  
} //FindArticul
```

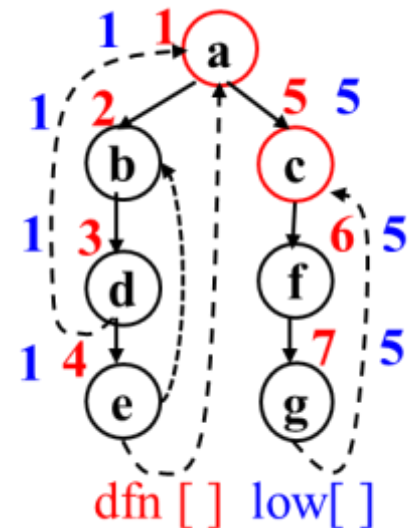
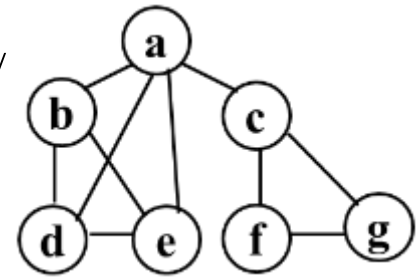


4.5 双连通性算法(cont.)

/*从顶点v0 出发深度优先遍历图G，计算low[]，查找并输出关节点*/

void DFSArticul(int v0)

```
{  dfn[v0]=min=count++; /*v0 是第count 个访问的顶点*/
  for(p=G.vexlist[v0].firstedge; p; p=p->next) /*对v0 的每个邻接点检查*/
  {    w=p->adjvex; /*w 为v0 的邻接点*/
      if(dfn[w]==0) /*若w 未曾访问，则w 为v0 的孩子*/
      {    DFSArticul(w); /*返回前求得low[w]*/
          if(low[w]<min)  min=low[w];
          if(low[w]>=dfn[v0])
              cout<<G.vexlist[v0].vertex); /*输出关节点*/
      }
      else if(dfn[w]<min)  min=dfn[w];
      /*w 已访问，w 是v0 在生成树上的祖先*/
  } //for
  low[v0]=min;
} //DFSArticul
```





4.6 强连通性：有向图

有向图强连通性的概念和性质

- 称有向图 $G=(V, E)$ 顶点 $v, w \in V$ 是等价的，要么 $v = w$ ；要么从 v 到 w 有一条有向路，并且从 w 到 v 也有一条有向路。
- 设 V_i 为等价类（顶点双向可达）， $E_i(1 \leq i \leq r)$ 是头、尾均在 V_i 中的边集，则 $G_i=(V_i, E_i)$ 称为 G 的一个强连通分量，简称强分量、强支。
- 对于有向图，在其每一个强连通分量中，任何两个顶点都是可达的。
 $\forall v \in G$ ，与 v 相互可到达的所有顶点，就是包含 v 的强连通分量的所有顶点。
- 设从 v 可到达（以 v 为起点的所有有向路径的终点）的顶点集合为 $T_1(G)$ ，而到达 v （以 v 为终点的所有有向路径的起点）的顶点集合为 $T_2(G)$ ，则包含 v 的强连通分量的顶点集合是： $T_1(G) \cap T_2(G)$ 。



4.6 强连通性(cont.)

有向图强连通性的概念和性质

- 强连通图性质定理

一个有向图是强连通的，当且仅当G中有一个回路，它至少包含每个顶点一次。

证明：

- 充分性

- 如果G中有一个回路，它至少包含每个顶点一次，则G中任两个顶点都是互相可达的，故G是强连通图。

- 必要性

- 如果有向图是强连通的，则任两个顶点都是相互可达。故必可做一回路经过图中所有顶点。
- 若不然则必有一回路不包含某一顶点 v ，并且 v 与回路上的各顶点就不是相互可达，与强连通条件矛盾。



4.6 强连通性(cont.)

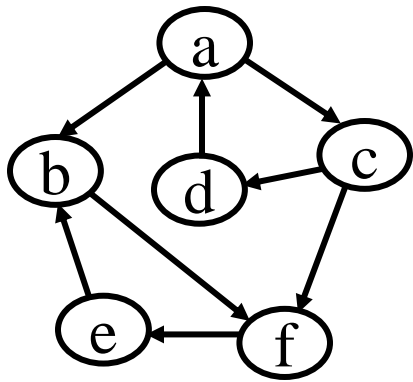
求有向图强连通分支算法：Kosaraju算法

- 输入：有向图G（例如，十字链表表示）
- 输出：有向图G的强连通分量（例如：森林的孩子-兄弟表示）
- 算法步骤：
 - 1.深度优先遍历有向图G（起点如何选择无所谓），并计算出每个顶点u的结束时间 $f[u]$ (按出栈的顺序编号)；
 - 2.深度优先遍历G的转置（反向）图 G^T ，按照顶点结束时间从大到小，选择遍历起点。遍历过程中，一边遍历，一边给顶点做分类标记，每找到一个新起点，分类标记值则加1。
 - 3.第2步中产生的标记值相同的顶点，构成深度优先森林中一棵树，也即一个强连通分量。

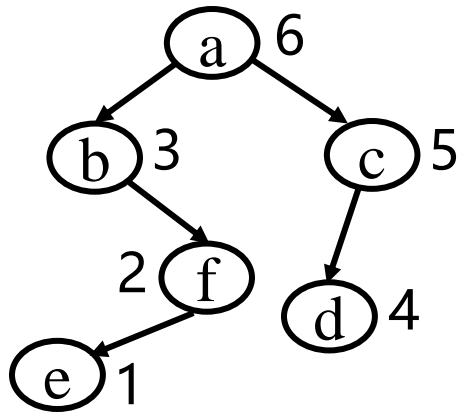


4.6 强连通性(cont.)

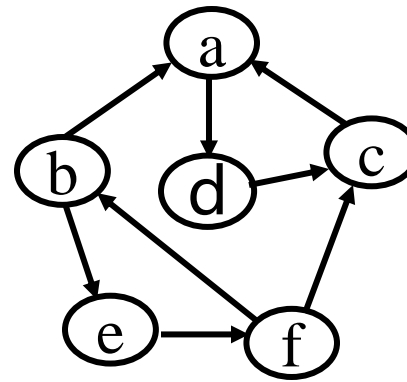
- 求有向图强连通分支示例



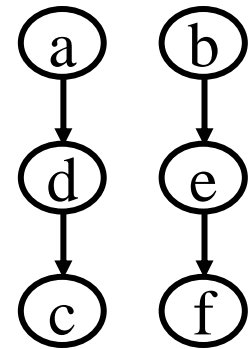
(a) 有向图G



(b) 执行步骤(1)



(c) G^T 执行步骤(2)



(d) 执行步骤(3)

- 利用深度优先搜索求有向图的强连通分量
 - 第一次DFS源图G，从顶点a开始；
 - 第二次DFS转置图 G^T ，由顶点f[n]大到小。



4.6 强连通性(cont.)

求有向图强连通分支：Kosaraju算法实现

/* 按弧的正向搜索，起点v如何选择无所谓 */

int in_order[MAX_VEX] ; /*全局 */

void DFS(OLGraph *G , int v)

{ ArcNode *p ;

Count=0 ;

visited[v]=TRUE ;

for (p=G->xlist[v].firstout ; p!=NULL ; p=p->tlink)

if (!visited[p->headvex])

DFS(G , p->headvex) ;

in_order[count++]=v ; /* 顶点计数，按出栈的顺序编号*/

}



4.6 强连通性(cont.)

求有向图强连通分支：Kosaraju算法实现

/* 对图G按弧的逆向进行搜索，起点为v*/

```
void Rev_DFS(OLGraph *G, int v)
```

```
{  ArcNode *p ;
```

```
    visited[v]=TRUE ;
```

```
    printf(“%d”, v) ;  /* 输出顶点 */
```

```
    for (p=G->xlist[v].firstin ; p!=NULL ; p=p->hlink)
```

```
        if (!visited[p->tailvex])
```

```
            Rev_DFS(G, p->tailvex) ;
```

```
}
```



4.6 强连通性(cont.)

求有向图强连通分支：Kosaraju算法实现

```
void Strongly_Connected_Component(OLGraph *G)
{
    int k=1, v, j ;
    for (v=0; v<G->vexnum; v++) visited[v]=FALSE ;
    for (v=0; v<G->vexnum; v++) /* 对图G正向遍历 */
        if (!visited[v]) DFS(G, v) ;
    for (v=0; v<G->vexnum; v++) visited[v]=FALSE ;
    for (j=G->vexnum-1; j>=0; j--) { /* 对图G逆向遍历 */
        v=in_order[j] ;
        if (!visited[v]) {
            printf("\n第%d个连通分量顶点:", k++);
            Rev_DFS(G, v) ;
        }
    }
}
```

Kosaraju算法复杂度:

- 深度优先搜索的复杂度: $\Theta(|V|+|E|)$
- 需两次深搜, 总的复杂度为: $\Theta(|V|+|E|)$ 。



4.7 最短路径算法



4.7 最短路径算法

- 最短路径(Shortest Path)问题

- 如果图中从一个顶点可以到达另一个顶点，则称这两个顶点间存在一条**路径**。
- 从一个顶点到另一个顶点间可能存在多条路径，而每条路径上经过的**边数**并不一定相同。
- 对于带权图，**路径长度**为路径上各边权值的总和。
- 两个顶点间路径**长度**最短的那条路径称为两个顶点间的**最短路径**，其路径长度称为**最短路径长度**。
- **如何找到**一条路径使得沿此路径上各边权值总和最小？
- **应用广泛**：GPS导航、路由选择、铺设管线等



4.7 最短路径算法(cont.)

- 问题解法

- 边上权值非负情形的单源最短路径问题
 - Dijkstra算法
- 边上权值为任意值的单源最短路径问题
 - Bellman-Ford算法
- 所有顶点之间的最短路径问题
 - Floyd-Warshall算法

- 边上权值非负情形的单源最短路径问题:

- 问题描述: 给定一个带权有向图 $G=(V, E)$ 和源点 $v \in V$, 求从 v 到 G 中其它顶点的最短路径。
- 限定各边上的权值大于或等于0。



4.7 最短路径算法(cont.)



Edsger Wybe Dijkstra

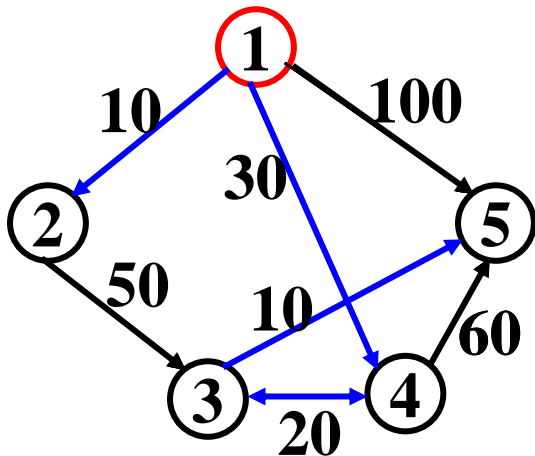
- 1930年5月11日~2002年8月6日
- [A.M. Turing Award Winner](#), 1972
- EDUCATION:
 - Undergraduate degree, physics, University of Leyden, 1956;
 - PhD computer science, University of Amsterdam, 1959.
- **结构程序设计之父**
- 提出“goto有害论”；
- 最短路径算法（SPF）的创造者；
- 解决了有趣的“哲学家聚餐”问题；
- 第一个Algol 60编译器的设计者和实现者；
- 提出信号量和PV原语；
- 提出银行家算法，解决了操作系统中资源分配问题等等



4.7 最短路径算法(cont.)

• Dijkstra算法的基本思想

- 按路径长度的递增次序, 逐步产生最短路径的贪心算法。
- 亦称SPF算法(Shortest Path First, 最短路径优先算法), 是OSPF路由协议的基础。
- 首先求出长度最短的一条最短路径, 再参照它求出长度次短的一条最短路径, 依次类推, 直到从顶点 v 到其它各顶点的最短路径全部求出为止。



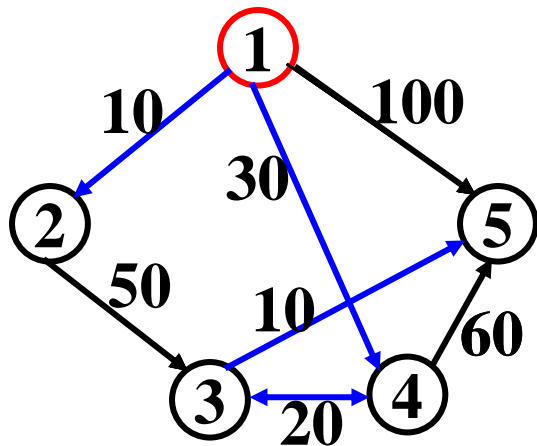
源点S	中间结点	终点	路径长度
1		2	1 0
1		4	3 0
1	4	3	5 0
1	4 3	5	6 0



4.7 最短路径算法(cont.)

• Dijkstra算法的数据结构

- 设带权有向图 $G=(V, E)$, 其中 $V=\{1, 2, \dots, n\}$, 顶点1为源点。
- 图 G 的存储结构: 采用带权的邻接矩阵 C 表示。
- 一维数组 $D[n]$: $D[i]$ 表示源点1到顶点 i 的当前最短路径长度, 初始时, $D[i]=C[1][i]$;
- 一维数组 $P[n]$: $P[i]$ 表示源点1到顶点 i 的当前最短路径上, 最后经过的顶点, 初始时, $P[i]=1$ (源点);
- $S[n]$: 存放源点和已生成的终点, 其初态为只有一个源点 v 。



$$C = \begin{bmatrix} \infty & 10 & \infty & 30 & 100 \\ \infty & \infty & 50 & \infty & \infty \\ \infty & \infty & \infty & 20 & 10 \\ \infty & \infty & 20 & \infty & 60 \\ \infty & \infty & \infty & \infty & \infty \end{bmatrix}$$

D		10	∞	30	100
---	--	----	----------	----	-----

P		1	1	1	1
---	--	---	---	---	---

S	T	F	F	F	F
---	---	---	---	---	---



4.7 最短路径算法(cont.)

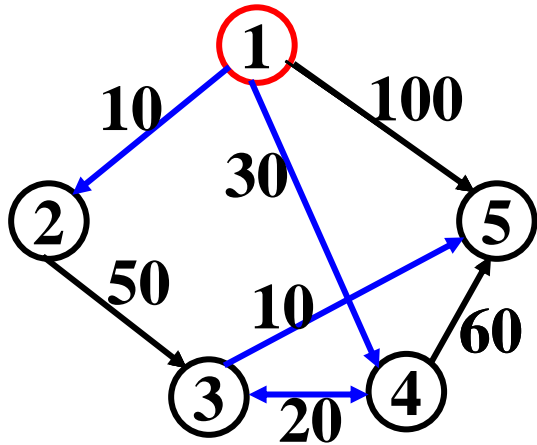
- Dijkstra算法实现步骤:

1. 将 V 分为两个集合 S （最短路径已经确定的顶点集合）和 $V-S$ （最短路径尚未确定的顶点集合）。初始时， $S=\{1\}$ ， $D[i]=C[1][i]$ （ $i=2,3,\dots,n$ ）， $P[i]=1$ （源点， $i \neq 1$ ）。
2. 从 S 之外即 $V-S$ 中，选取一个顶点 w ，使 $D[w]$ 最小（ $D[w]=\min\{D[i] \mid i \in V-S\}$ ），即从源点到达 w 只通过 S 中的顶点，且是一条最短路径（选定路径），并把 w 加入集合 S 。
3. 调整 $D[]$ 中记录的从源点到 $V-S$ 中每个顶点的最短距离，即从原来的 $D[v]$ 和 $D[w]+C[w][v]$ 中选择最小值作为 $D[v]$ 的新值，且 $P[v]=w$ 。
4. 重复2和3，直到 S 中包含 V 的所有顶点为止。此时，数组 $D[]$ 记录了从源到 V 中各顶点的最短距离，数组 $P[]$ 记录最短路径。

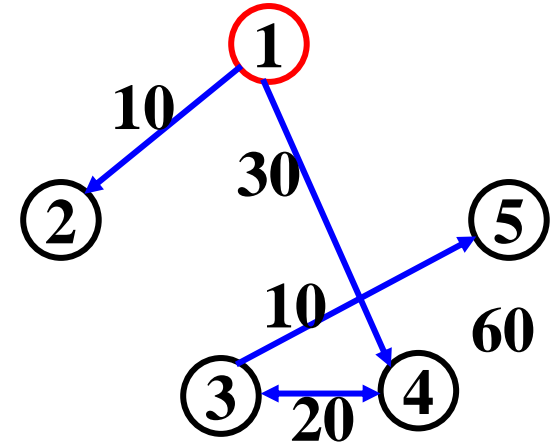


4.7 最短路径算法(cont.)

Dijkstra算法示例



$$C = \begin{bmatrix} \infty & 10 & \infty & 30 & 100 \\ \infty & \infty & 50 & \infty & \infty \\ \infty & \infty & \infty & 20 & 10 \\ \infty & \infty & 20 & \infty & 60 \\ \infty & \infty & \infty & \infty & \infty \end{bmatrix}$$



循环	S	w	D[2]	D[3]	D[4]	D[5]	P[2]	P[3]	P[4]	P[5]
初态	{1}	-	10	∞	30	100	1	1	1	1
1	{1,2}	2	10	60	30	100	1	2	1	1
2	{1,2,4}	4	10	50	30	90	1	4	1	4
3	{1,2,4,3}	3	10	50	30	60	1	4	1	3
4	{1,2,4,3,5}	5	10	50	30	60	1	4	1	3



4.7 最短路径算法(cont.)

Dijkstra算法的实现

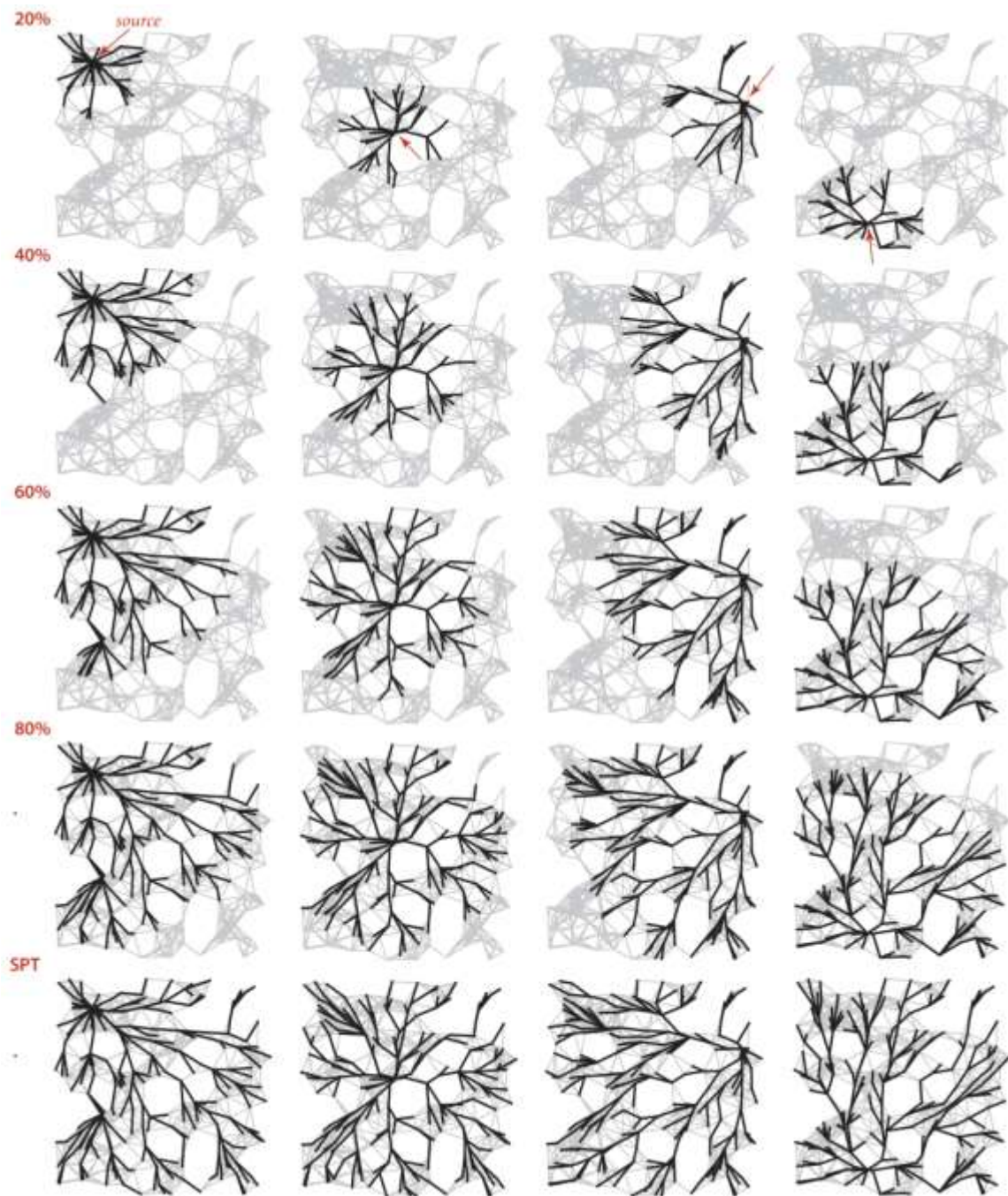
```
void Dijkstra(GRAPH C, costtype D[n+1], int P[n+1], bool S[n+1])
```

```
{ for ( i=1 ; i<=n; i++ )
    { D[i]=C[1][i] ; S[i]=FALSE ; }
  S[1]= TRUE ; // 已处理
  for( i=1; i<n; i++)
  { w=MinCost ( D, S ) ;
    S[w]=TRUE ; //w已完成
    for ( v=2 ; v<= n ; v++ )
      if ( S[v]!=TRUE ) //更新所有待处理的
      { sum=D[w] + C[w][v] ;
        if (sum < D[v] ) { D[v] = sum ; P[v]=w; } }
  }
} // 时间复杂度： O (n2)
```

```
costtype MinCost (D, S)
{
  temp = INFINITY ;
  w = 2 ;
  for ( i=2 ; i<=n ; i++ )
    if (!S[i]&&D[i]<temp)
      { temp = D[i] ;
        w = i ;
      }
  return w ;
}
```



示例:



Dijkstra's algorithm (250 vertices, various sources)



4.7 最短路径算法

Floyd-Warshall算法



4.7 最短路径算法(cont.)

- 其它最短路径问题及解法

- ✓ 单目标最短路径问题:

- 找出图中每个顶点 v 到某个指定结点 c 的最短路径

- 单顶点对间最短路径问题:

- 对于某对顶点 u 和 v , 找出 u 到 v 的一条最短路径
 - 以 u 为源点

- 所有顶点间的最短路径问题:

- 对图中每对顶点 u 和 v , 找出 u 到 v 的最短路径
 - 以每个顶点为源点
 - 直接用Floyd算法



Robert W. Floyd

- 1978年12月4日，**荣获图灵奖**，奖励他为编译理论奠定深远基础。
- 1936年6月8日生于纽约
- 1953年获得芝加哥大学**文学学士**学位(17岁)
- 1958年又获得了**理科学士**学位
- 1962年公司分析员
- 1964年与William共同发明了著名的**堆排序算法**
- 1965年卡内基—梅隆大学副教授
- 1968年斯坦福聘任为教授。在此期间完成了ALGOL 60编译器开发。





4.7 最短路径算法(cont.)

任意两个顶点之间的**最短路径(Floyd-Warshall算法)**

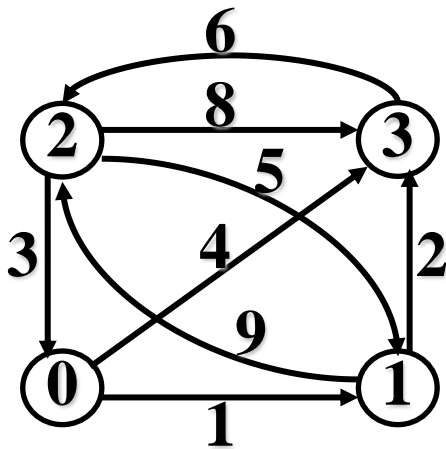
- **问题描述**：已知一个带权的有向图 $G=(V, E)$ ，对每一对顶点 $v_i, v_j \in V, (i \neq j)$ ，要求：求出 v_i 与 v_j 之间的**最短路径**和**最短路径长度**。限制条件：**不允许有负长度的环路**。
- **Floyd算法基本想法**：**动态规划算法**
 - 如果 v_i 与 v_j 之间有有向边，则 v_i 与 v_j 之间有一条路径，但不一定是最短的；也许经过**某些中间点**会使路径长度更短。
 - **经过哪些中间点**会使路径长度缩短呢？经过哪些中间点会使路径长度最短呢？
 - 只需尝试在原路径中间**加入其它顶点**作为中间顶点。
 - 如何尝试？
 - **系统地**在原路径中间加入每个顶点，然后不断地调整当前路径(和路径长度)即可。



4.7 最短路径算法(cont.)

最短路径示例：

- $\langle 2, 1 \rangle 5$: $\langle 2, 0 \rangle \langle 0, 1 \rangle 4$, $a[2][1] = a[2][0] + a[0][1]$ 调整
- 注意: 考虑 v_0 做中间点可能还会改变其它顶点间的距离:
 $\langle 2, 3 \rangle 8$: $\langle 2, 0, 3 \rangle 7$, $a[2][3] = a[2][0] + a[0][3]$
- $\langle 2, 3 \rangle$: $\langle 2, 0 \rangle \langle 0, 3 \rangle$: $\langle 2, 0 \rangle \langle 0, 1 \rangle \langle 1, 3 \rangle = \langle 2, 0, 1, 3 \rangle$ $a[2][3] = 6$
调整
- 注意: 有时加入中间顶点后的路径比原路径长 保持



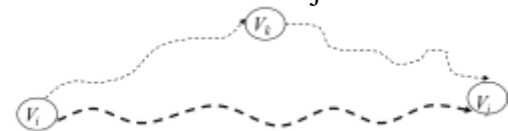
$$A = \begin{matrix} & \begin{matrix} 0 & 1 & 2 & 3 \end{matrix} \\ \begin{pmatrix} \infty & 1 & \infty & 4 \\ \infty & \infty & 9 & 2 \\ 3 & 5 & \infty & 8 \\ \infty & \infty & 6 & \infty \end{pmatrix} & \begin{matrix} 0 \\ 1 \\ 2 \\ 3 \end{matrix} \end{matrix}$$



4.7 最短路径算法(cont.)

Floyd算法基本思想：

- 假设求顶点 v_i 到顶点 v_j 的最短路径。如果从 v_i 到 v_j 存在一条长度为 $C[i][j]$ 的路径，该路径不一定是最短路径，尚需进行 n 次试探。
- 首先考虑路径 (v_i, v_0, v_j) 是否存在。如果存在，则**比较** (v_i, v_j) 和 (v_i, v_0, v_j) 的路径长度，取长度**较短者**，为从 v_i 到 v_j 的**中间顶点的序号不大于0**的最短路径。
- 假设在路径上再增加一个顶点 v_1 ，也就是说，如果 (v_i, \dots, v_1) 和 (v_1, \dots, v_j) 分别是当前找到的，中间顶点序号**不大于0**的最短路径，那么 $(v_i, \dots, v_1, \dots, v_j)$ 就是**有可能**是从 v_i 到 v_j 的**中间顶点序号不大于1**的最短路径。将它与已经得到的从 v_i 到 v_j **中间顶点序号不大于0**的最短路径相**比较**，从中选出**中间顶点序号不大于1**的最短路径，再增加一个顶点 v_2 ，继续进行**试探**。
- 若 (v_i, \dots, v_k) 和 (v_k, \dots, v_j) 分别是从小于 v_i 到 v_k 和从 v_k 到 v_j 的**中间顶点序号不大于 $k-1$** 的最短路径，则将 $(v_i, \dots, v_k, \dots, v_j)$ 和已经得到的从 v_i 到 v_j 且**中间顶点序号不大于 $k-1$** 的最短路径**比较**，其长度较短者便是从 v_i 到 v_j 的**中间顶点序号不大于 k** 的最短路径。





4.7 最短路径算法(cont.)

- Floyd算法的数据结构

- 图的存储结构:

- 带权的有向图采用邻接矩阵 $C[n][n]$ 存储

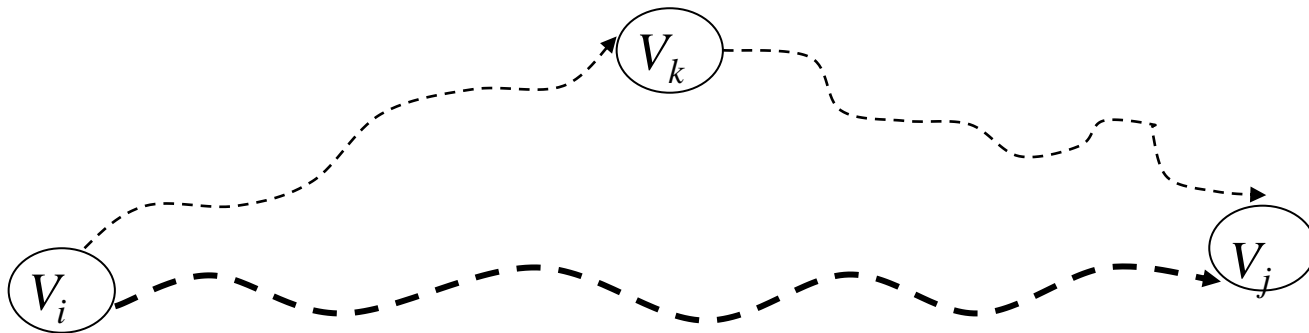
- 数组 $D[n][n]$:

- 存放在迭代过程中求得的最短路径长度。迭代公式:

$$\begin{cases} D_{-1}[i][j] = C[i][j] \\ D_k[i][j] = \min\{ D_{k-1}[i][j], D_{k-1}[i][k] + D_{k-1}[k][j] \} \quad 0 \leq k \leq n-1 \end{cases}$$

- 数组 $P[n][n]$:

- 存放从 v_i 到 v_j 求得的最短路径。初始时 $P[i][j] = -1$





4.7 最短路径算法(cont.)

• Floyd算法实现

```
void Floyd( costtype D[][], costtype C[][], int P[][], int n)
{
    for ( i = 0; i < n; i++ )
        for ( j = 0; j < n; j++ ) {
            D[i][j] = C[i][j];
            P[i][j] = -1;
        }
    for ( k = 0; k < n; k++ )
        for ( i = 0; i < n; i++ )
            for ( j = 0; j < n; j++ )
                if ( D[i][j] > D[i][k] + D[k][j] ) {
                    D[i][j] = D[i][k] + D[k][j];
                    P[i][j] = k;
                }
} // 时间复杂度: O(n3)
```

Warshall算法:

- 求有向图邻接矩阵C的传递闭包D
- 判定有向图任意两点间是否存在有向路

$$D[i][j] = D[i][j] \cup (D[i][k] \cap D[k][j])$$



4.7 最短路径算法(cont.)



- 要输出从i到j的最短路径上除i和j之外的那些顶点，可借助于如下函数：

```
Void Path (i, j)
{
    k=P[i][j];
    if(k!=0)
    {
        Path(i, k);
        cout<<k<<endl;
        Path(k, j);
    }
}
```



4.7 最短路径算法(cont.)

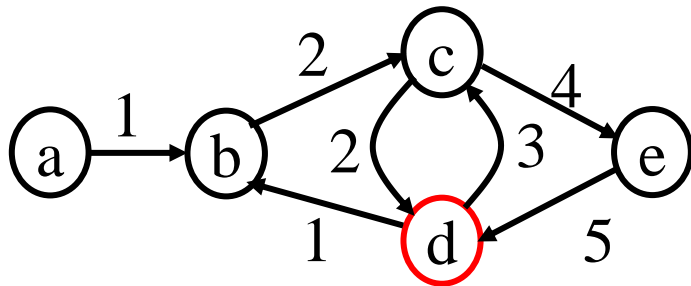
Floyd算法的应用：求有向图的中心点

- 顶点的偏心度：

- 设 $G=(V, E)$ 是一个带权有向图， $D[i][j]$ 表示从 i 到 j 的最短距离。对任意一个顶点 k ， $E(k) = \max\{d[i][k] \mid i \in V\}$ 称作顶点 k 的偏心度（其他顶点到顶点 k 的最大最短距离）。
- 矩阵 D 中每列 j 的最大值

- 图 G 的中心点：

- 称具有最小偏心度的顶点为图 G 的中心点。



	a	b	c	d	e
a	0	1	3	5	7
b	∞	0	2	4	6
c	∞	3	0	2	4
d	∞	1	3	0	7
e	∞	6	8	5	0

最短路径矩阵 D

顶点	偏心度
a	∞
b	6
c	8
d	5
e	7

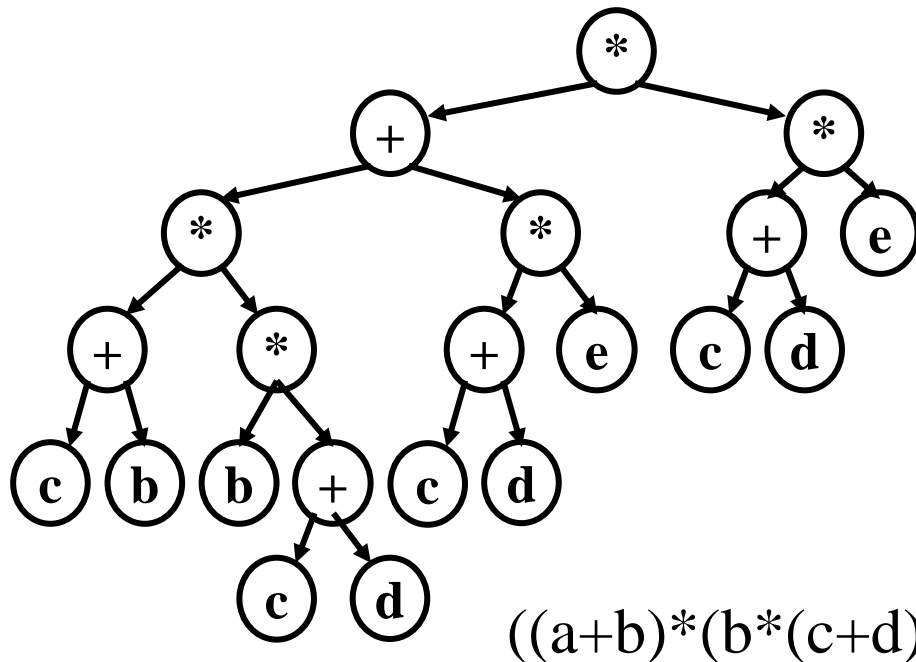


4.8 拓扑排序算法



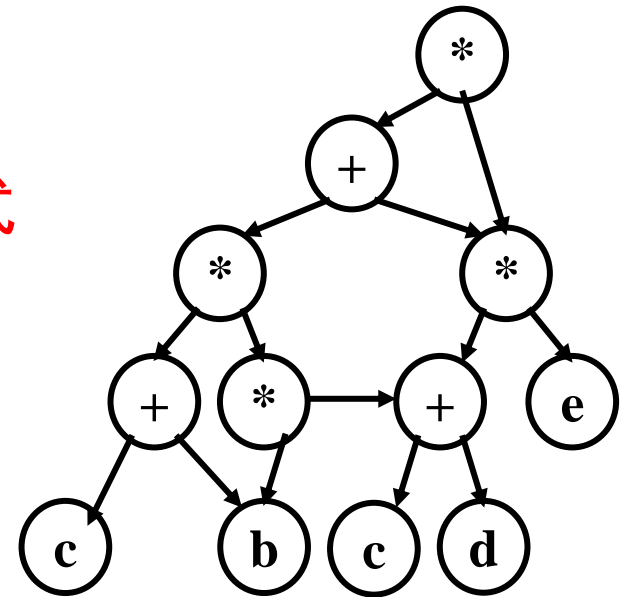
4.8 拓扑排序算法

- **无环路有向图**：不存在环路的有向图的简称。
- **注意**：无环路有向图对应的**无向图**可能存在环路。
- 无环路的有向图可用于表示**偏序集**。
- 无环路有向图可以描述**含有公共子式的表达式**（节省空间）



公共子式

- b
- $(c+d)$
- $(c+d)*e$





4.8 拓扑排序算法(cont.)

- **偏序关系**：若集合 X 上的关系 R 是**自反的**、**反对称的**和**传递的**，则称 R 是集合 X 上的**偏序关系**。
 - **自反性**：任意 $x \in X$ ， $(x, x) \in R$
 - **反对称性**：任意 $x, y \in X$ ，若 $(x, y) \in R$ 且 $(y, x) \in R$ ，则 $x = y$
 - **传递性**：任意 $x, y, z \in X$ ， $(x, y) \in R$ 且 $(y, z) \in R$ ，则 $(x, z) \in R$
- **全序关系**：
 - 设 R 是集合 X 上的偏序关系，如果对**每个** $x, y \in X$ ，必有 $(x, y) \in R$ 或 $(y, x) \in R$ ，则称 R 是集合 X 上的**全序关系**



4.8 拓扑排序算法(cont.)

- 如何用无环路有向图表示偏序关系？
 - 设 R 是有穷集合 X 上的偏序关系，对 X 中每个 v ，用一个以 v 为标号表示顶点，由此构成顶点集 V ；
 - 对任意 $(u, v) \in R, (u \neq v)$ ，由对应两个顶点建立一条有向边，由此构成边集 E ，则 $G = (V, E)$ 是无环路有向图。
- 拓扑排序：由某个集合的一个偏序得到该集合的一个线性序列的过程。所得的线性序列称为拓扑序列。
- AOV网：在一个表示工程的有向图中，用顶点表示活动，用弧表示活动之间的优先关系，称这样的有向图为顶点表示活动的网，简称AOV网。
 - AOV网中的弧表示活动之间存在的某种制约关系
 - AOV网中不能出现回路
 - 在AOV网中，若顶点 i 到 j 有一条有向路，则称 i 为 j 的前驱， j 为 i 的后继。
 - 若 $(i, j) \in E$ ，则 i 称为 j 的直接前驱， j 称为 i 的直接后继。

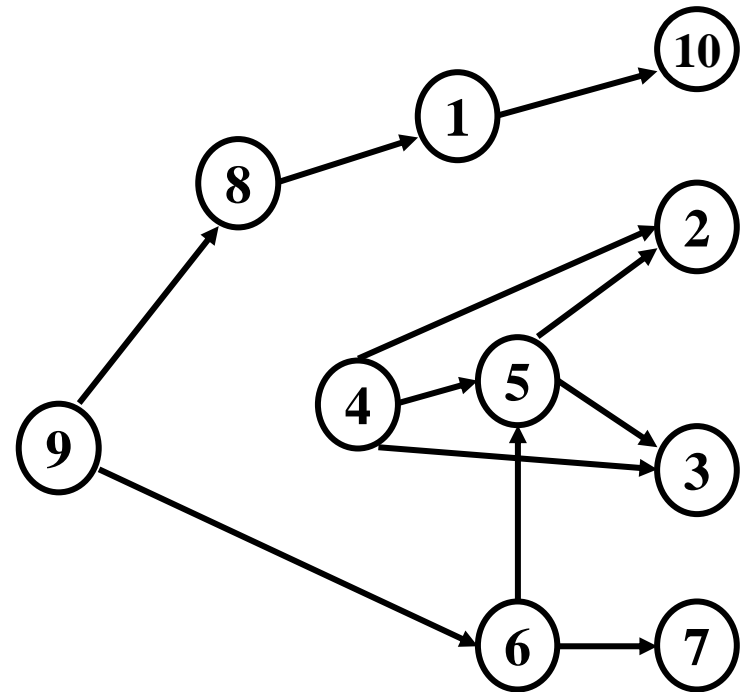


4.8 拓扑排序算法(cont.)

AOV网示例:

- 课程及课程间的先修关系是偏序关系，可以用AOV网表示。

课程代号	课程名称	先修课代号
1	计算机原理	8
2	编译原理	4,5
3	操作系统	4,5
4	程序设计	无
5	数据结构	4,6
6	离散数学	9
7	形式语言	6
8	电路基础	9
9	高等数学	无
10	计算机网络	1



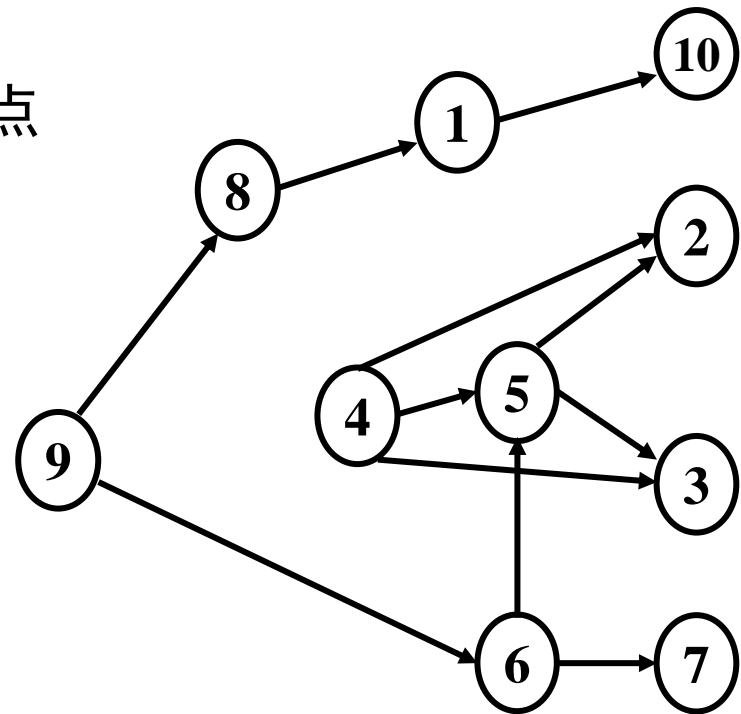


4.8 拓扑排序算法(cont.)

- 利用AOV网进行拓扑排序的基本思想：

- ① 从AOV网中选择一个没有前驱的顶点并且输出它；
- ② 从AOV网中删去该顶点和所有以该顶点为尾的弧；
- ③ 重复上述两步，直到全部顶点都被输出，或AOV网中不存在没有前驱的顶点。

- 任何无环路的AOV网，其顶点都可以排成一个拓扑序列。
- 其拓扑序列不一定是唯一的





4.8 拓扑排序算法(cont.)

- **拓扑排序算法**：实质是**广度优先**搜索算法

- **输入**：有向图的邻接表
- **输出**：所有顶点组成的拓扑序列
- **算法实现步骤**：（使用队列）

1. **建立**入度为零的顶点排队
2. 扫描顶点表，将**入度为0**的顶点**入队**；
3. while（**排队不空**）{
 输出队头结点；
 记下输出结点的数目；
 删去与之关联的**出边**；
 若有入度为0的结点，则入队；
}

注：图中还有**未输出**的顶点，
但已**跳出**循环处理，表明图中
还**剩**有顶点，且**都有直接前驱**，
这时网络中必存在**有向环**。

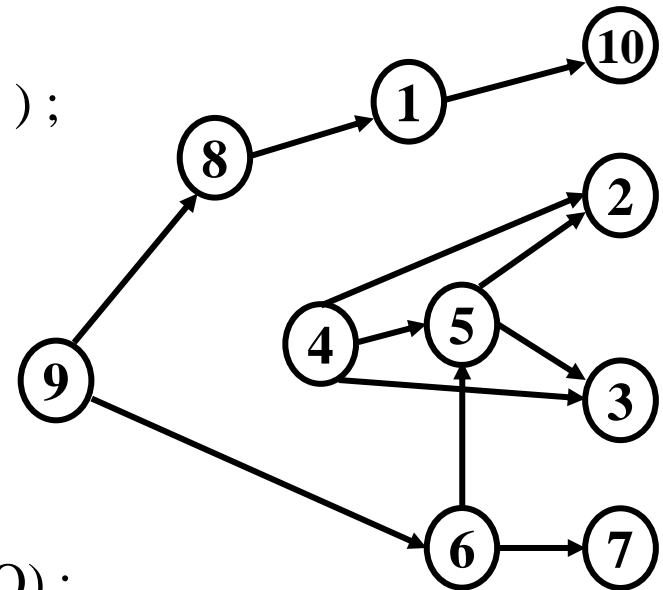
4. 若输出**结点个数小于n**，则**有环路**；否则，拓扑排序正常完成结束。



4.8 拓扑排序算法(cont.)

- **拓扑排序算法**：实质是**广度优先搜索**算法

```
void Topologicalsort( AdjGraph G )  
{  QUEUE Q ; nodes = 0 ;  
  MAKENUILL( Q ) ;  
  for( v=1; v<=G.n ; ++v )  
    if ( indegree[v]==0 ) ENQUEUE( v, Q ) ;  
  while ( !EMPTY( Q ) ) {  
    v = FRONT(Q) ;  
    DEQUEUE( Q ) ;  
    cout << v ;  nodes ++ ;  
    for( 邻接于 v 的每个顶点 w )  
      if( !(--indegree[w])) ENQUEUE(w,Q) ;  
  }  
  if ( nodes < n ) cout<<“图中有环路” ;  
}
```





4.8 拓扑排序算法(cont.)

广度优先拓扑排序：分析

- 与先广搜索的差别：
 - 搜索起点是入度为0的顶点；
 - 需判断是否有环路；
 - 需对访问并输出的顶点计数（引入计数器nodes）。
 - 需删除邻接于 v 的边（采用**入度数组** `indegree[]`或在顶点表中增加一个**属性域** `indegree`）。
- 其他实现方法
 - 也可采用**栈**数据结构进行广度优先拓扑排序
 - 也可采用**无后继顶点优先**的拓扑排序算法
 - 也可利用**DFS遍历**进行拓扑排序



4.8 拓扑排序算法(cont.)

利用栈结构进行拓扑排序：基本思想

输入：有向图的邻接表

输出：所有顶点组成的拓扑序列

算法实现步骤：（使用栈）

1. 建立入度为零的顶点**栈**
2. 扫描顶点表，将入度为0的顶点**入栈**；
3. while（**栈**不空）{
 输出栈顶结点；
 记下输出结点的数目；
 删去与之关联的出边；
 若有入度为0的结点，**入栈**
}
4. 若输出结点个数小于 n ，则有环路；否则，拓扑排序正常完成。



4.8 拓扑排序算法(cont.)

利用栈结构进行拓扑排序：实现

```
void Topologicalsort( AdjGraph G )
{   MAKENUILL( S ) ; count = 0 ;
    for( v=0; v<n ; ++v )
        if ( !indegree[v] ) push( v, S ) ;
    while ( !EMPTY( S ) ) {
        v = pop ( S ) ; printf( v ); ++count ;
        for( 邻接于 v 的每个顶点 w ) {
            if( !(--indegree[w]))
                push(S, w) ;
        }
    }
    if ( count < n ) cout<<“图中有环路” ;
}
```



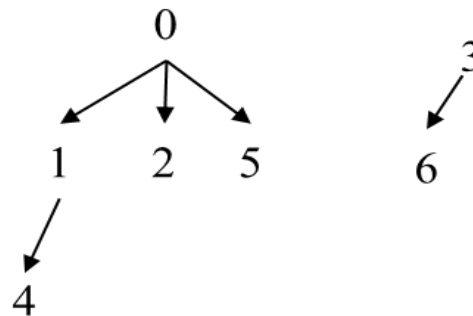
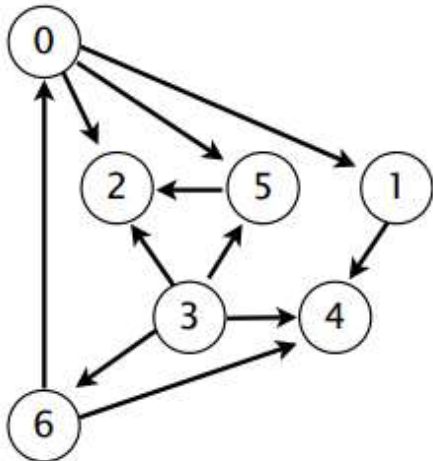
4.8 拓扑排序算法(cont.)

基于DFS的拓扑排序算法：基本思想

- ① 执行深度优先搜索
- ② 针对生成森林，输出后序遍历序列的逆序列

分析：

- 后序遍历序列最后项入度为0，恰好为起始点。
- 次后项为最后项的直接后继。

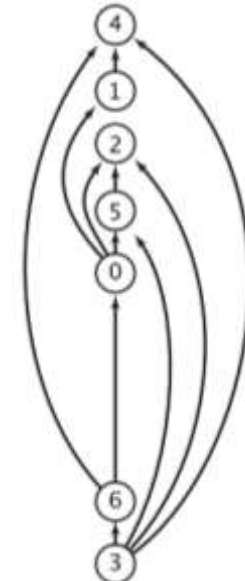


后续遍历序列：

4 1 2 5 0 6 3

拓扑序列（逆序）：

3 6 0 5 2 1 4



拓扑关系

- 并非唯一
- 结点1可下移



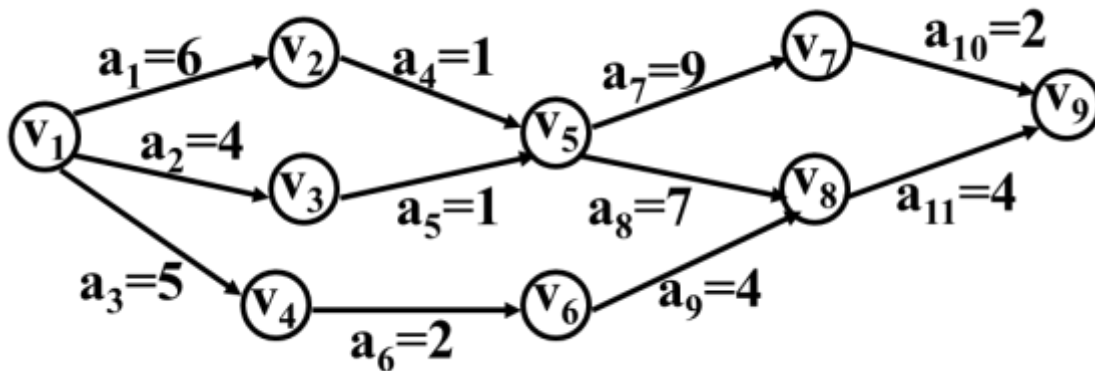
4.9 关键路径算法



4.9 关键路径算法

应用案例

- 一个软件研发项目，假设已确认了11项基本活动，所有这些活动的名称、每项活动完成所需时间，及其与其他活动的约束关系。
- 问题1：如何描述项目进度？
- 问题2：完成整个项目至少需要多少时间？
- 问题3：哪些任务是影响项目进度的关键？
 - 若任务 a_5 推迟了3天，对项目进度有何影响？



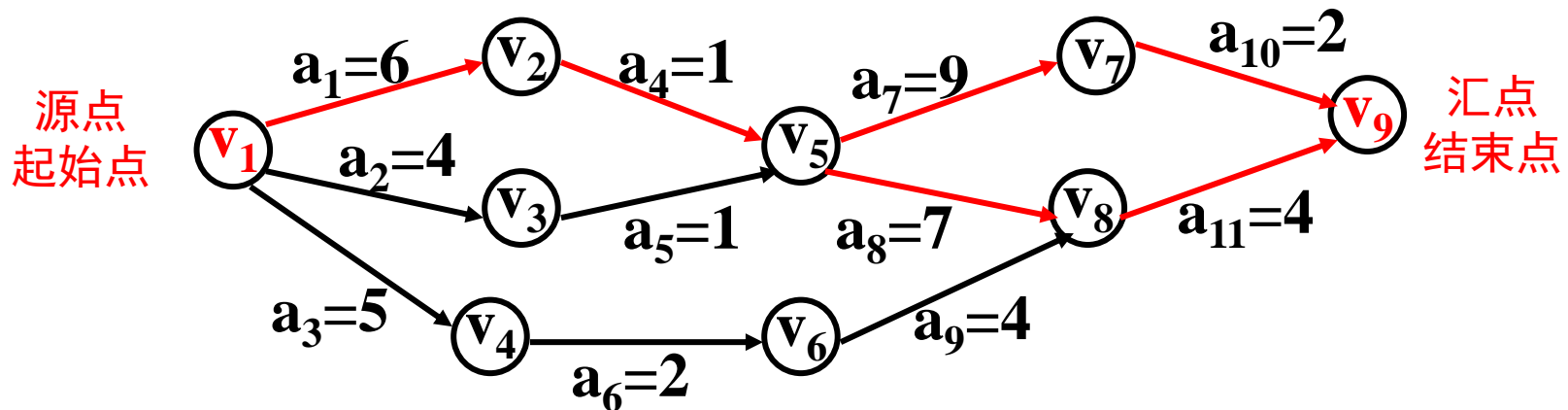
活动名称	必需时间(天)	前置任务
a₁	6	
a₂	4	
a₃	5	
a₄	1	a₁
a₅	1	a₂
a₆	2	a₃
a₇	9	a₄,a₅
a₈	7	a₄,a₅
a₉	4	a₆
a₁₀	2	a₇
a₁₁	4	a₈,a₉



4.9 关键路径算法(cont.)

AOE网 (Activity On Edge Network)

- 带权有向图中，顶点表示**事件**，边表示**活动**，边上权表示活动的**开销**（如**持续时间**），则称此有向图为**边表示活动的网络**，即**AOE网**。
- 示例如下图：具有11项活动、9个事件的AOE网。每个事件表示在它**之前**的活动**已经完成**，在它**之后**的活动**可以开始**。

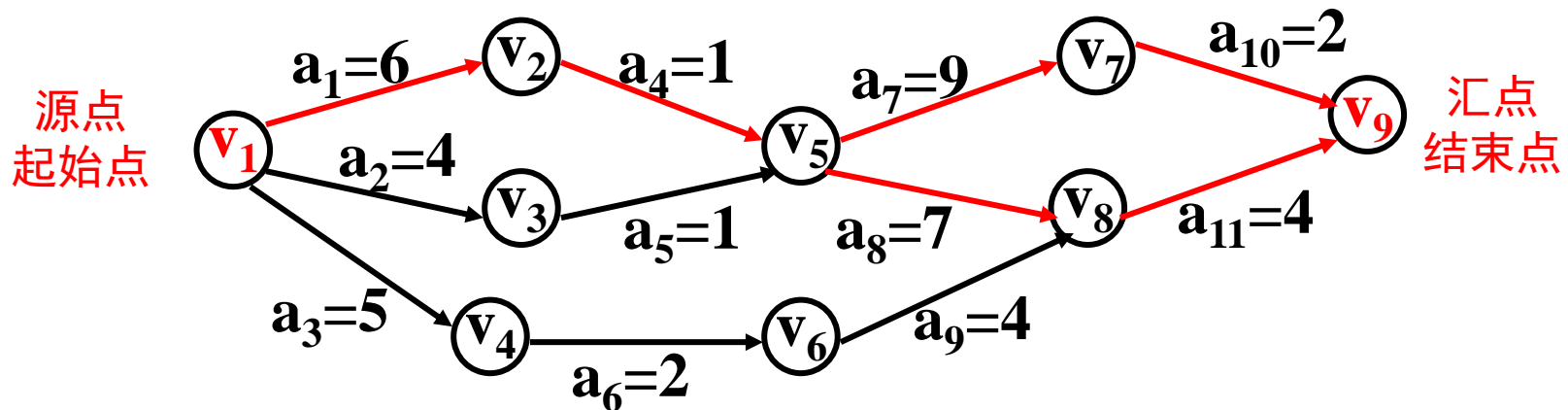




4.9 关键路径算法(cont.)

• AOE网的性质

- 只有在某个顶点所代表的**事件**发生后，从**该顶点出发**的各有向边代表的**活动才能开始**；
- 只有在**进入某一顶点**的各有向边代表的**活动已经结束**，该顶点所代表的**事件才能发生**；
- 表示实际工程计划的AOE网应是**无环图**，且存在唯一入度为0的开始顶点（**源点**）和唯一出度为0的结束点（**汇点**）。



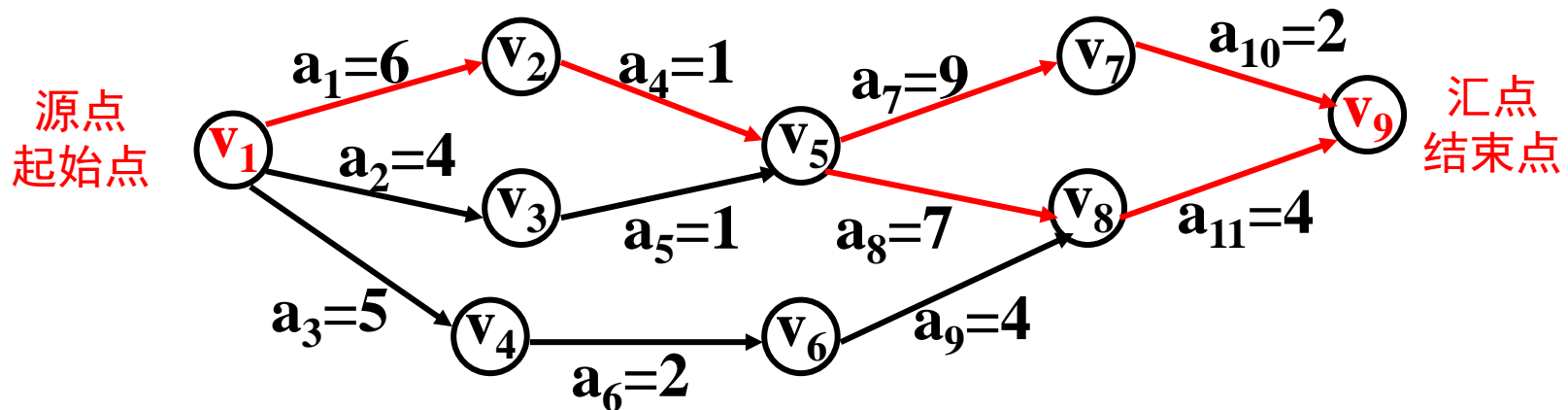


4.9 关键路径算法(cont.)

- AOE网主要研究问题：

- 如果AOE网表示一项工程，仅考虑各子工程之间的优先关系还不够，更应关注如下问题：

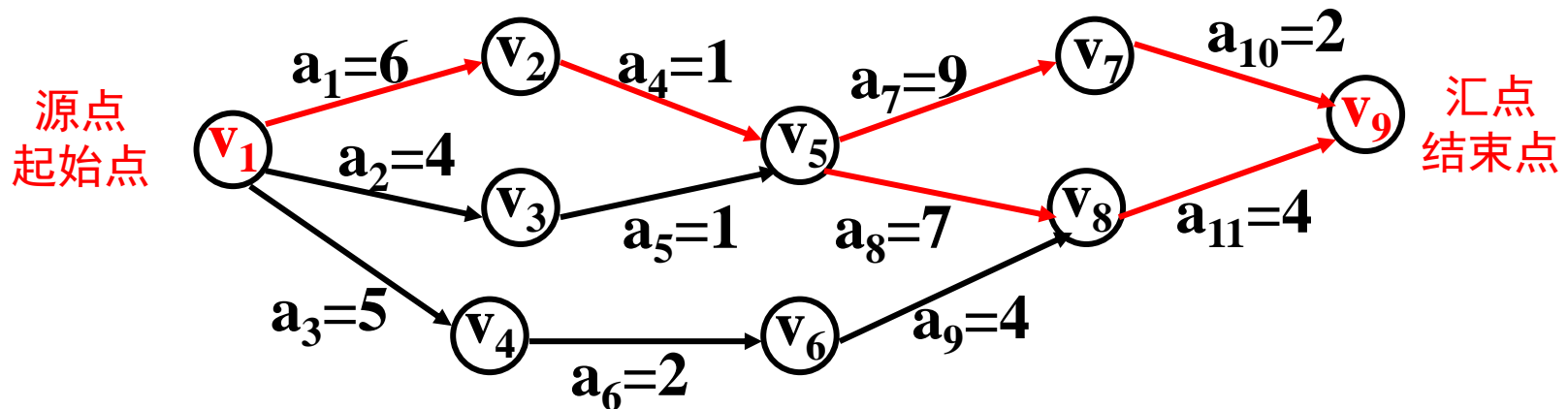
- ① 完成整个工程至少需要多少时间？
- ② 哪些活动是影响工程进度的关键活动？
- ③ 加速这些活动能否提高整个工程的效率？





4.9 关键路径算法(cont.)

- 路径长度、关键路径、关键活动：
 - 路径长度：从源点到汇点路径上所有活动的持续时间之和。
 - 关键路径：在AOE网中，完成工程的最短时间是源点到汇点的最大路径长度。把从源点到汇点具有最大长度的路径称为关键路径。有些活动可以并行。
 - 一个AOE中，关键路径可能不只一条。
 - 关键活动：关键路径上的活动称为关键活动。



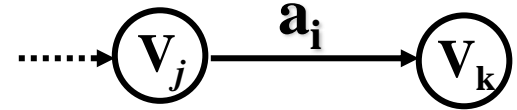


4.9 关键路径算法(cont.)

关键路径与关键活动性质分析：关键活动相关量

① 事件 V_j 的最早发生时间 $VE(j)$ ：

- 是从源点 V_1 到顶点 V_j 的最长路径长度。



② 活动 a_i 的最早开始时间 $E(i)$ ：

- 设活动 a_i 在边 $\langle V_j, V_k \rangle$ 上, 则 $E(i)$ 也是从源点 V_1 到顶点 V_j 的最长路径长度。
- 因为事件 V_j 发生, 表明以 V_j 为起点的所有活动 a_i 可立即开始。因此, $E(i) = VE(j)$ 。

③ 事件 V_k 的最迟发生时间 $VL(k)$ ：

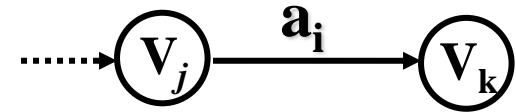
- 在保证汇点 V_n 在 $VE(n)$ 时刻发生的前提下, 事件 V_k 允许的最迟发生时间。
- 在不推迟工期的情况下, 一个事件最迟发生时间 $VL(k)$ 应该等于汇点最早发生时间 $VE(n)$ 减去从 V_k 到 V_n 的最大路径长度。



4.9 关键路径算法(cont.)

关键路径与关键活动性质分析：关键活动相关量

④ 活动 a_i 的最迟开始时间 $L(i)$:



- 是在不会引起工期延误前提下，允许活动 a_i 的最迟开始时间。
- 事件 V_k 发生表明，以 V_k 为终点的入边所表示的**所有活动**均已完成，所以事件 V_k 的**最迟发生**时间 $VL(k)$ 也是所有以 V_k 为终点的入边 $\langle V_j, V_k \rangle$ 所表示的活动 a_i **可以最迟完成时间**。
- 显然，为不推迟工期，活动 a_i 的**最迟开始**时间 $L(i)$ 应该是 a_i 的最迟完成时间 $VL(k)$ **减去** a_i 的持续时间，即

$$L(i) = VL(k) - ACT[j][k]$$

其中， $ACT[j][k]$ 是活动 a_i 的持续时间($\langle V_j, V_k \rangle$ 上的权值)



4.9 关键路径算法(cont.)

关键路径与关键活动性质分析：关键活动相关量

⑤ 时间余量 $L(i) - E(i)$

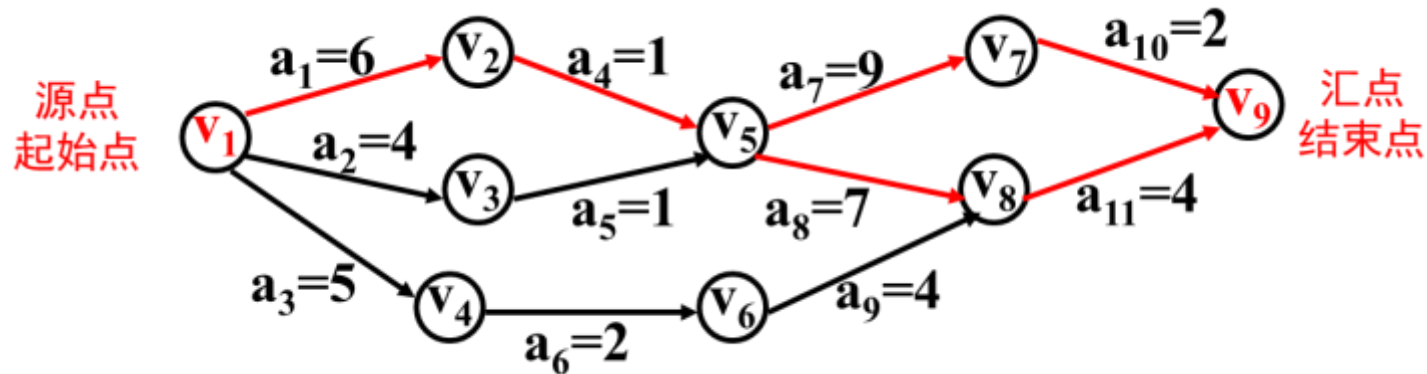
$L(i) - E(i)$ 表示活动 a_i 最早开始时间和最迟开始时间的的时间余量。

- 关键路径上的活动都满足： $L(i) = E(i)$
- $L(i) = E(i)$ 表示的活动是没有时间余量的关键活动。
- 为找出关键活动，需求出各个活动的 $E(i)$ 与 $L(i)$ ，以判别一个活动 a_i 是否满足 $L(i) = E(i)$ 。
- $E(i)$ 和 $L(i)$ 可由前面公式计算得到
- $VE(k)$ 和 $VL(k)$ 可由拓扑排序算法得到



4.9 关键路径算法(cont.)

关键路径和关键活动分析计算示例：



事件	VE[j]	VL[k]	活动	E[i]	L[i]	L[i] - E[i]
V_1	0	0	a_1	0	0	0
V_2	6	6	a_2	0	2	2
V_3	4	6	a_3	0	3	3
V_4	5	8	a_4	6	6	0
V_5	7	7	a_5	4	6	2
V_6	7	10	a_6	5	8	3
V_7	16	16	a_7	7	7	0
V_8	14	14	a_8	7	7	0
V_9	18	18	a_9	7	10	3
			a_{10}	16	16	0
			a_{11}	14	14	0

事件的
最早
发生
时间

事件的
最迟
发生
时间

活动的
最早
开始
时间

活动的
最晚
开始
时间

时间
余量



4.9 关键路径算法(cont.)

利用拓扑排序算法求关键路径和关键活动

(1) 前进阶段：从源点 V_1 出发，令 $VE(1) = 0$ ，按**拓扑序列**求出其余各顶点事件的**最早发生**时间：

$$VE(k) = \max\{ VE(j) + ACT[j][k] \} \quad j \in T$$

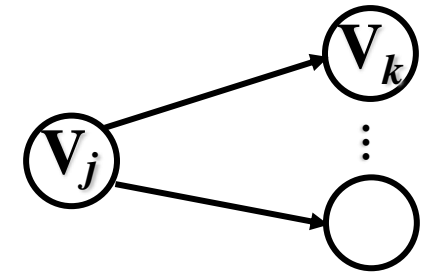
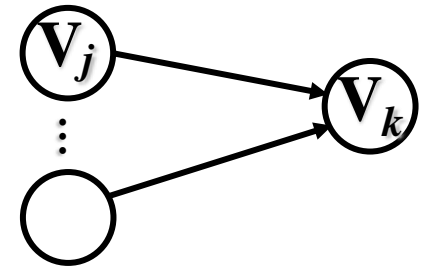
其中， T 是顶点 V_k 的**直接前驱**顶点的集合($2 \leq k \leq n$)

若网中**有回路**，则不能求出关键路径，算法中止；否则转(2)

(2) 回退阶段：从汇点 V_n 出发，令 $VL(n) = VE(n)$ ，按**逆拓扑**有序求其余各顶点**事件的最晚发生**时间：

$$VL(j) = \min\{ VL(k) - ACT[j][k] \} \quad k \in S$$

其中， S 是顶点 V_j 的**直接后继**顶点的集合($2 \leq j \leq n-1$)





4.9 关键路径算法(cont.)

利用拓扑排序算法求关键路径和关键活动

(3) 计算 $E(i)$ 和 $L(i)$ ：

- 求每一项活动 a_i 的最早开始时间： $E(i) = VE(j)$
- 求每一项活动 a_i 的最晚开始时间： $L(i) = VL(k) - ACT[j][k]$

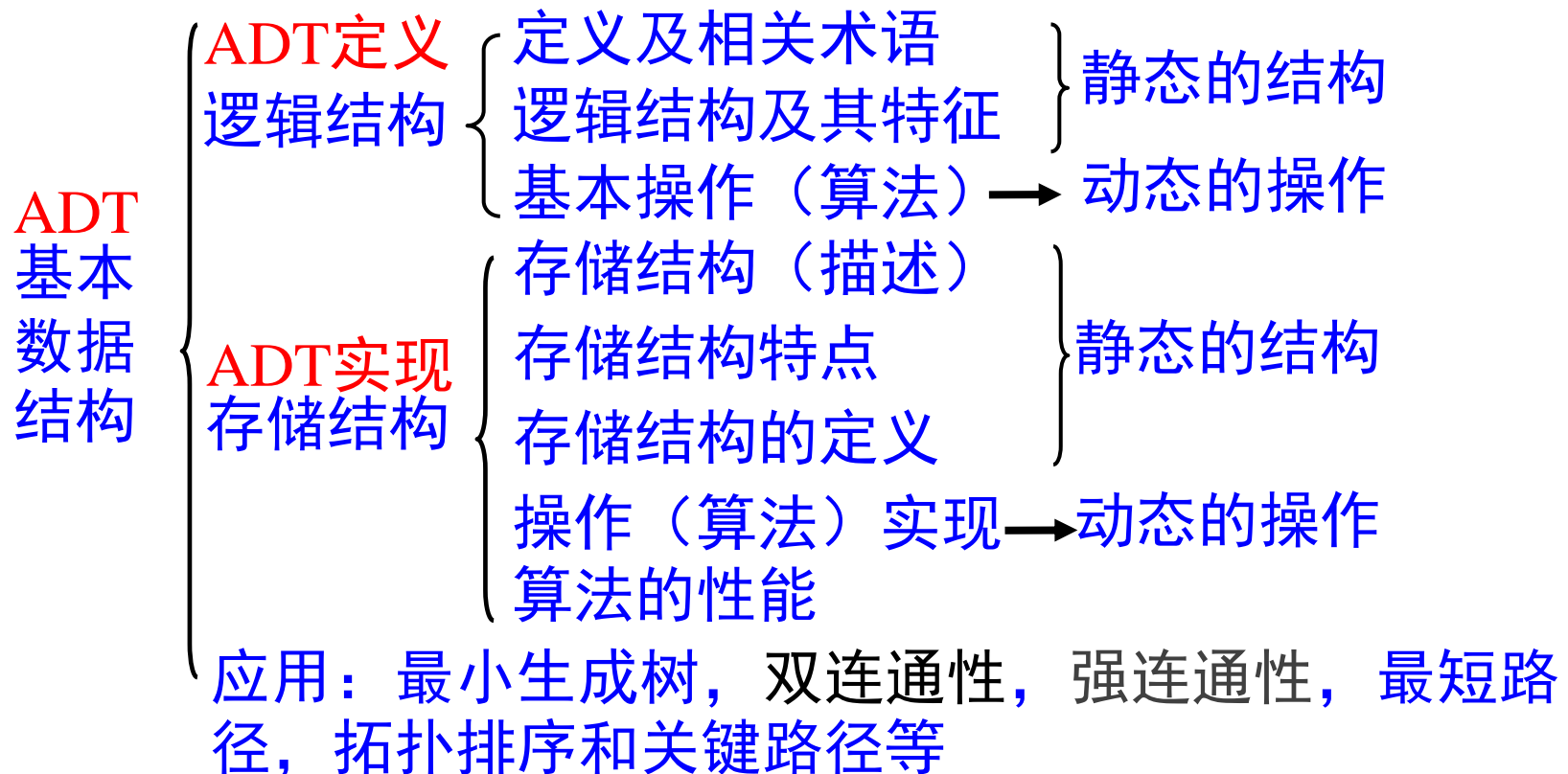
(4) 若某条边满足 $E(i) = L(i)$ ，则它是关键活动。

- 为简化算法，可以在求关键路径**之前已经**对各顶点实现**拓扑排序**，并按拓扑顺序对各顶点重新进行了编号。
- 不是任意一个关键活动的加速一定能使整个工程提前。
- 想整个工程提前，要考虑各关键路径上**所有**关键活动。



本章的知识点结构

- 基本的数据结构（ADT）
 - 图（无向图、有向图；加权图；网络）
- 知识点结构



图的搜索(遍历)算法是有关图问题的重要基本核心算法



图算法时间复杂度汇总(next)

- 深度优先搜索(Depth FirstSearch, DFS)
 - 时间复杂度:
 - 邻接表: $O(n+e)$, 邻接矩阵: $O(n^2)$
- 广度优先搜索(Breadth First Search, BFS)
 - 时间复杂度:
 - 邻接表: $O(n+e)$; 邻接矩阵: $O(n^2)$
- 普里姆(Prim)算法
 - 时间复杂度: $O(|V|^2)$
- 克鲁斯卡尔(Kruskal)
 - 时间复杂度: $O(|E| \log |E|)$
- Dijkstra算法的实现
 - 时间复杂度: $O(n^2)$
- Floyd算法实现
 - 时间复杂度: $O(n^3)$
- R.Tarjan算法(双连通性算法)
 - R.Tarjan算法是先深搜索, 因此, 时间复杂度为: $O(n+e)$ 或 $O(n^2)$
- Kosaraju算法复杂度(求有向图强连通分支):
 - 深度优先搜索的复杂度: $\Theta(|V|+|E|)$
 - 需两次深搜, 总的复杂度为: $\Theta(|V|+|E|)$ 。



本章小结

知识点

