数据结构与算法

Data Structures and Algorithms

张岩

海量数据计算研究中心

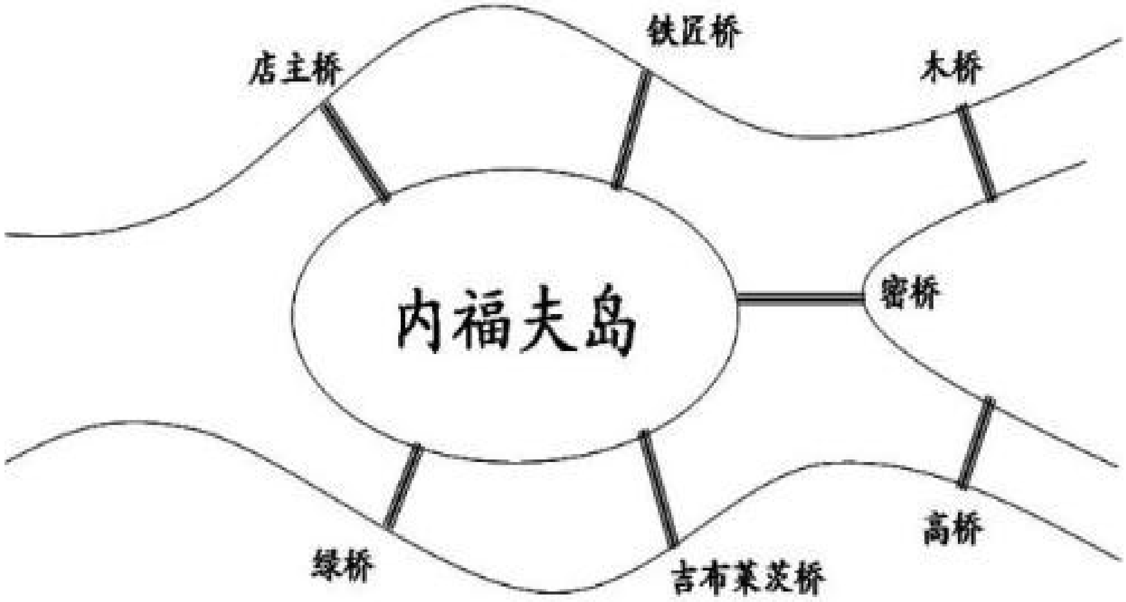
哈工大计算机科学与技术学院

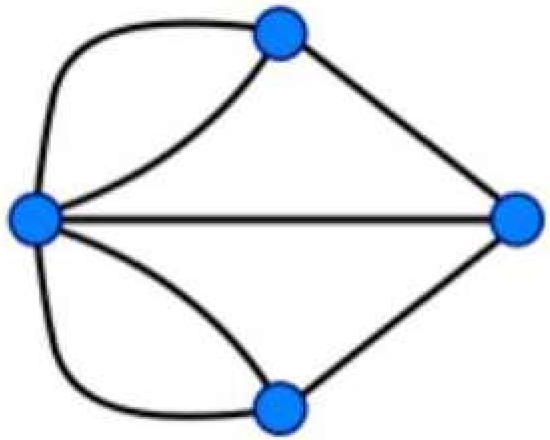
第4章 图结构及其应用算法

欧拉—图论创始人

欧拉1707年出生在瑞士的巴塞尔城，19岁开始发表论文，直到76岁。几乎每一个数学领域都可以看到欧拉的名字，从初等几何的欧拉线，多面体的欧拉定理，立体解析几何的欧拉变换公式，四次方程的欧拉解法到数论中的欧拉函数，微分方程的欧拉方程，级数论的欧拉常数，变分学的欧拉方程，复变函数的欧拉公式等等。据统计他那不倦的一生，共写下了886本书籍和论文，其中分析、代数、数论占40%，几何占18%，物理和力学占28%，天文学占11%，弹道学、航海学、建筑学等占3%。1733年，年仅26岁的欧拉担任了彼得堡科学院数学教授。1741年到柏林担任科学院物理数学所所长，直到1766年，重回彼得堡，没有多久，完全失明。欧拉在数学上的建树很多，对著名的哥尼斯堡七桥问题的解答开创了图论的研究。

哥尼斯堡七桥问题





- 历史名城哥尼斯堡有一条河叫普莱格尔河，河上有七座古桥，将河中两个岛与河岸连接。。。。
- 居民中流传着一道难题：一个人怎样才能一次走遍七座桥，每座桥只走过一次，最后回到出发点？
- 欧拉把他转化为连通图的一笔画问题（边搜索问题）。

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，在计算机科学、数学和工程中有着非常广泛的应用。
- 了解图的定义及相关的术语，掌握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的实现；
- 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

ADT基本数据结构
- ADT定义 逻辑结构
  - 定义及相关术语
  - 逻辑结构及其特征 } 静态的结构
  - 基本操作（算法）→ 动态的操作
- ADT实现 存储结构
  - 存储结构（描述）
  - 存储结构特点
  - 存储结构的定义 } 静态的结构
  - 操作（算法）实现→动态的操作
  - 算法的性能
- 应用:最小生成树,最短路径,拓扑排序和关键路径

图的搜索(遍历)算法是有关图问题的重要核心算法！

4.1 基本定义

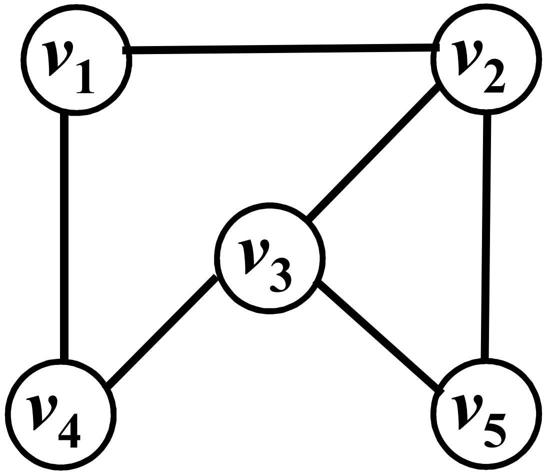
定义1 图(Graph)

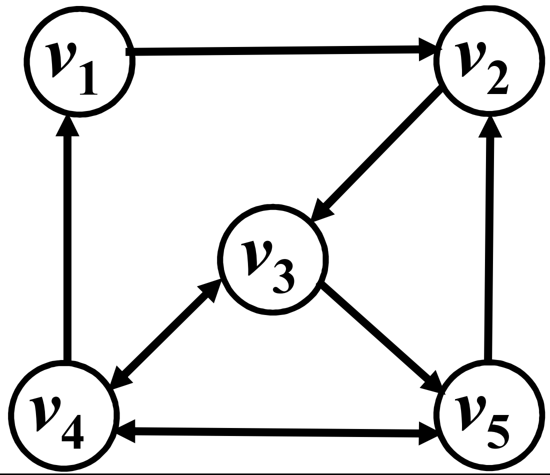
- 图是由顶点(vertex)的有穷非空集合和顶点之间边(edge)的集合组成的一种数据结构，通常表示为：

$$G = (V, E)$$

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。

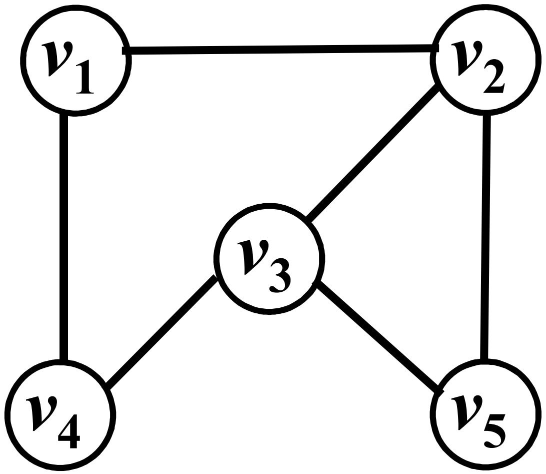
顶点表示数据对象；边表示数据对象之间的关系。

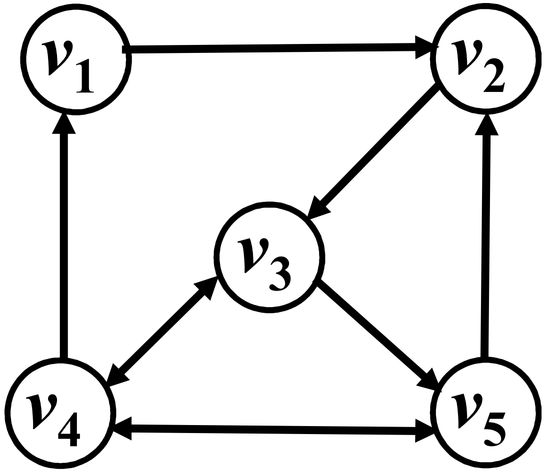




4.1 基本定义(cont.)

定义1 图

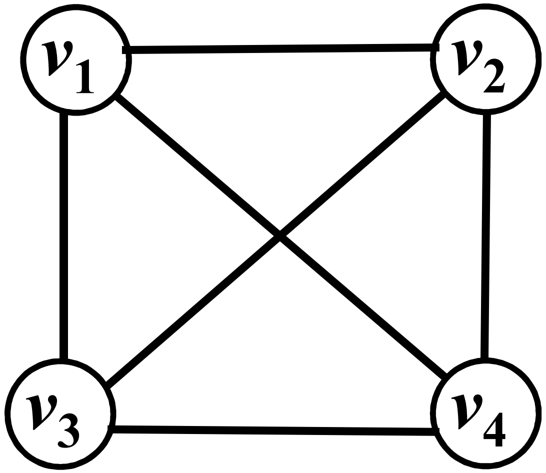


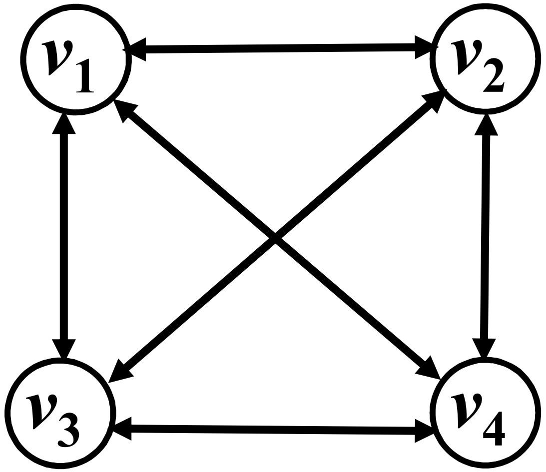


- 无向图：
  - 若顶点v_i和v_j之间的边没有方向，则称这条边为无向边，表示为(v_i , v_j)。
  - 如果图的任意两个顶点之间的边都是无向边，则称该图为无向图。
- 有向图：
  - 若顶点v_i和v_j之间的边有方向，则称这条边为有向边(弧)，表示为$<v_i , v_j>$：
  - <弧尾，弧首(头)>
  - 如果图的任意两个顶点之间的边都是有向边，则称该图为有向图。

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。

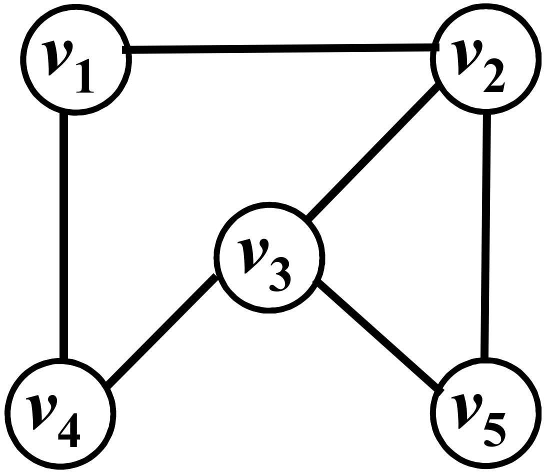




- 含有n个顶点的无向完全图有多少条边？
- 含有n个顶点的有向完全图有多少条弧？

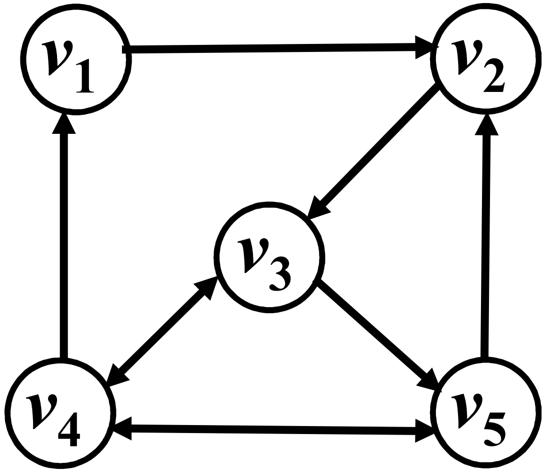
4.1 基本定义(cont.)

定义1 图



邻接、依附

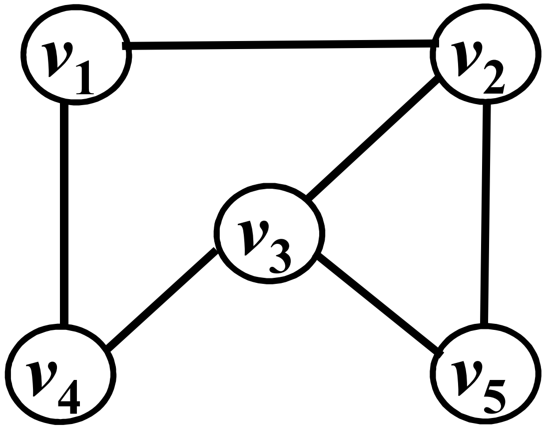
- 在无向图中，对于任意两个顶点v_i和顶点v_j，若存在边(v_i，v_j)，则称顶点v_i和顶点v_j相邻，互为邻接点，同时称边(v_i，v_j)依附于顶点v_i和顶点v_j。
- 如：v_2的邻接点：v_1，v_3，v_5

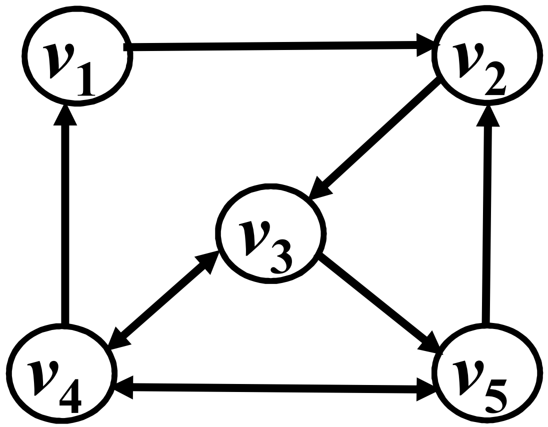


- 在有向图中，对于任意两个顶点v_i和顶点v_j，若存在有向边$<v_i, v_j>$，则称顶点v_i邻接到顶点v_j，顶点v_j邻接于顶点v_i，同时称弧$<v_i, v_j>$依附于顶点v_i和顶点v_j。
- 如：v_1的邻接到v_2，v_1邻接于v_4

4.1 基本定义(cont.)

定义2 度(Dgree)

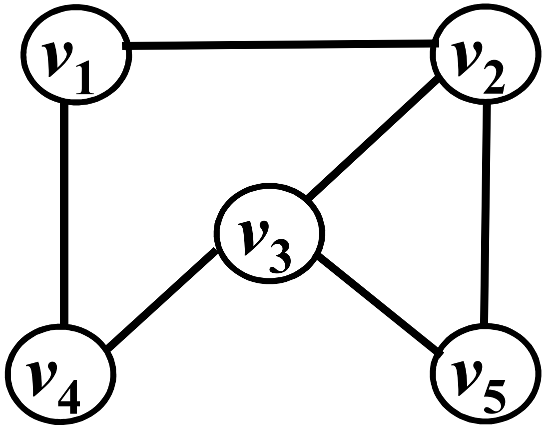


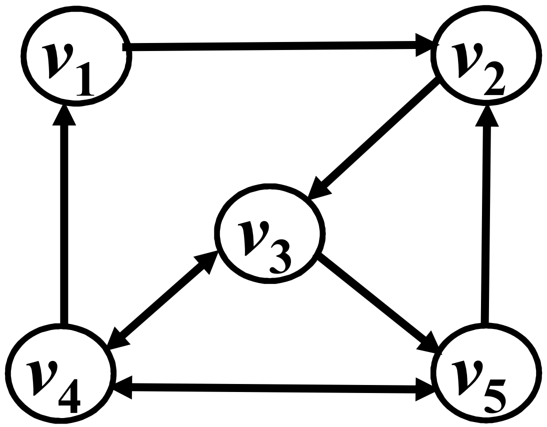


- 顶点的度：在无向图中，顶点v的度是指依附于该顶点的边数，通常记为D (v)。
- 顶点的入度：在有向图中，顶点v的入度是指以该顶点为弧头的弧的数目，记为ID (v)；
- 顶点的出度：在有向图中，顶点v的出度是指以该顶点为弧尾的弧的数目，记为OD (v)。
- 在有向图中， D (v)= ID (v) + OD (v)

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.)

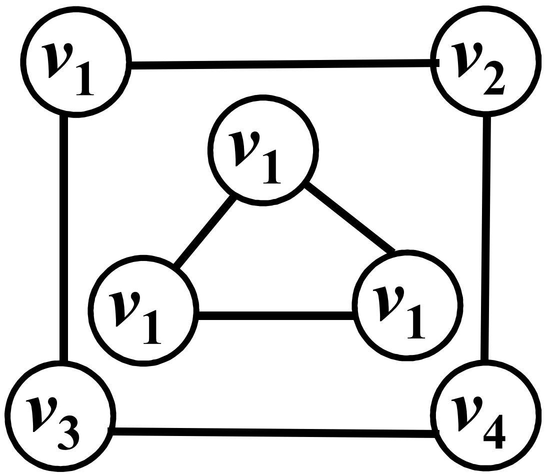
定义3 路径（Path）和路径长度、简单路和简单回路

- 在无向图G=(V, E) 中，若存在一个顶点序列v_p ,v_{i1} , v_{i2} , … v_{im} ,v_q ， 使得（v_p ,v_{i1}），（v_{i1} ,v_{i2}）, … ,（v_{im} ,v_q）∈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} ,… v_{im} ,v_q ， 使得有向边$<v_p ,v_{i1}>,<v_{i1} ,v_{i2}>, \ldots ,<v_{im} ,v_q>$∈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 $v_1,v_2,\ldots,v_m$ 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。

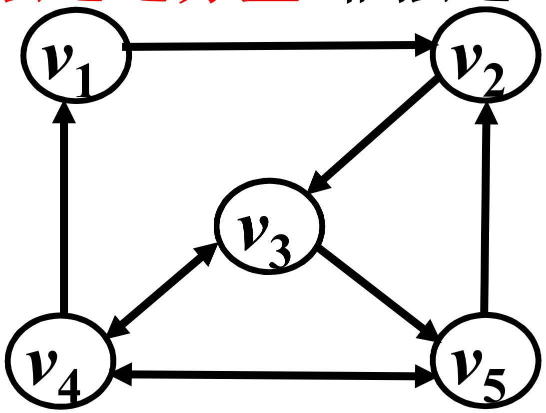


1.含有极大顶点数；
2.依附于这些顶点的所有边.

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
  - 顶点的强连通性：在有向图中, 若对于每一对顶点v_i和v_j $(i≠j)$, 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.)

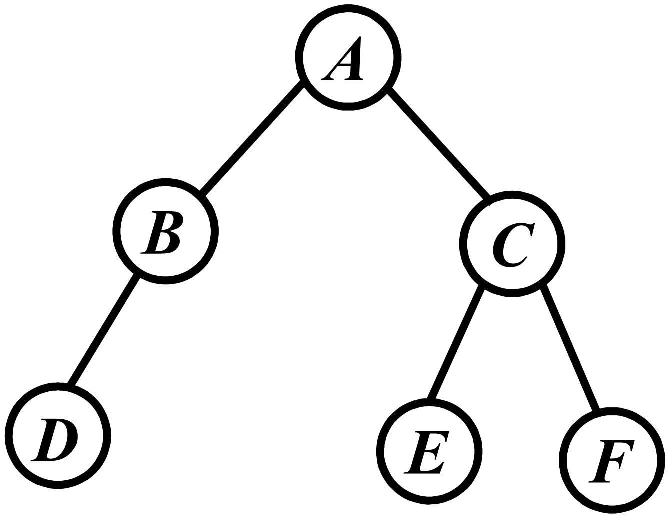
图的操作

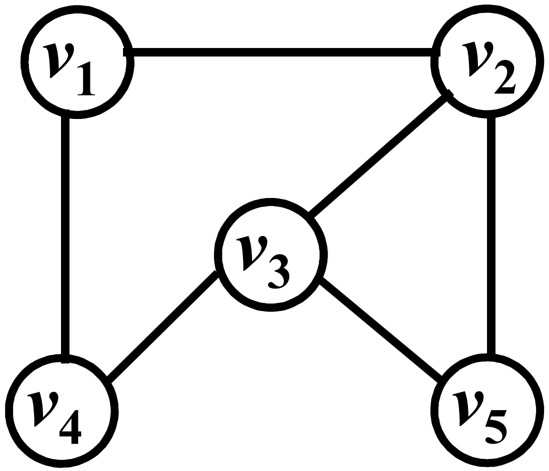
设图G=(V,E)，图上定义的基本操作如下：

- **NEWNODE (G)**：建立一个新顶点，**V=V∪{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∪(v1,v2) ,V=V**
- **DELSUCC (G, v1, v2)**：删除边（v1,v2）,V不变
- **SUCC (G, v1, v2)**：求出v的所有直接后继结点
- **PRED (G, v)**：求出v的所有直接前导结点
- **ISEDGE (G, v1, v2)**：判断（v1,v2）∈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。
- 等等

4.1 基本定义(cont.)

不同逻辑结构之间的比较

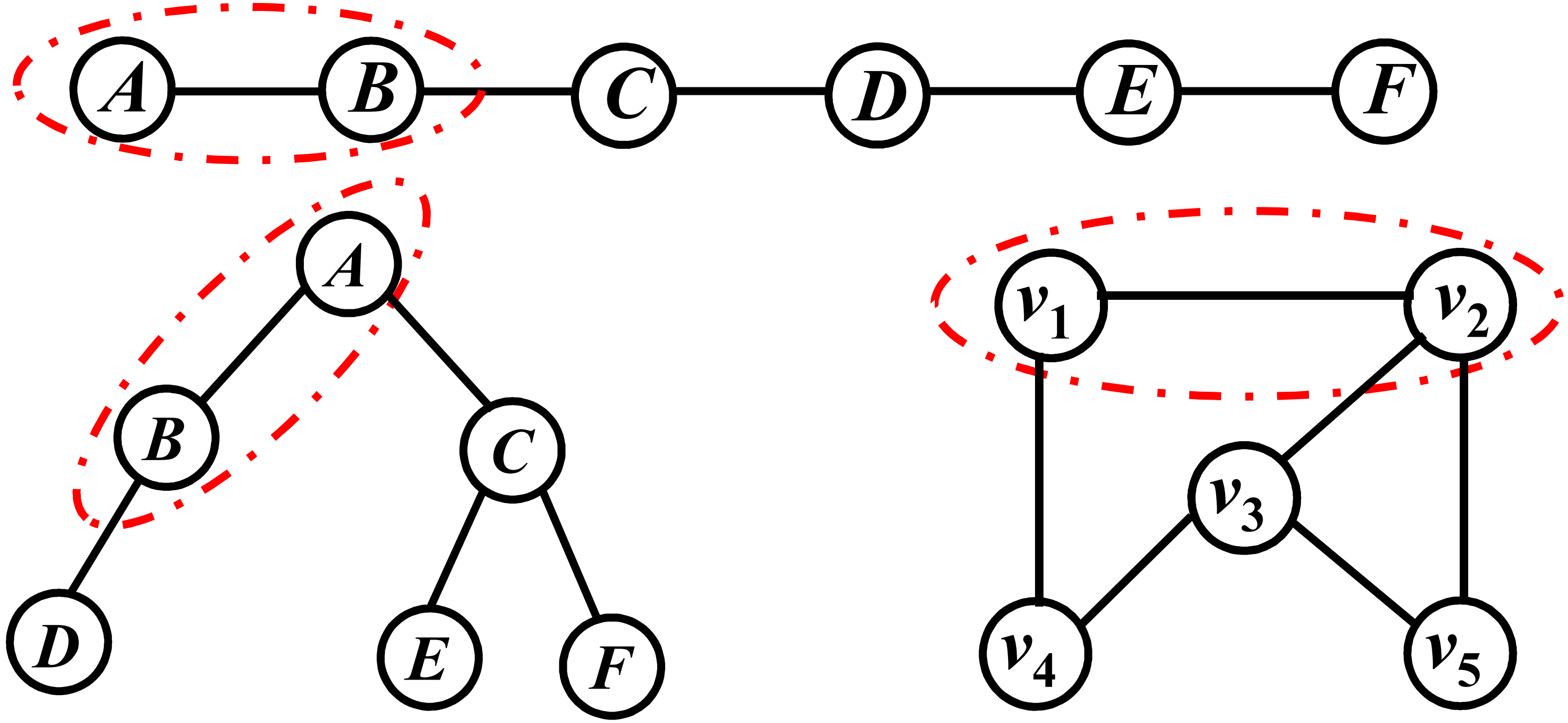




- 在线性结构中，数据元素之间仅具有**线性关系(1:1)**；
- 在树型结构中，结点之间具有**层次关系(1:*m*)**；
- 在图型结构中，任意两个顶点之间**都可能有关系(*m*:*n*)**。

4.1 基本定义(cont.)

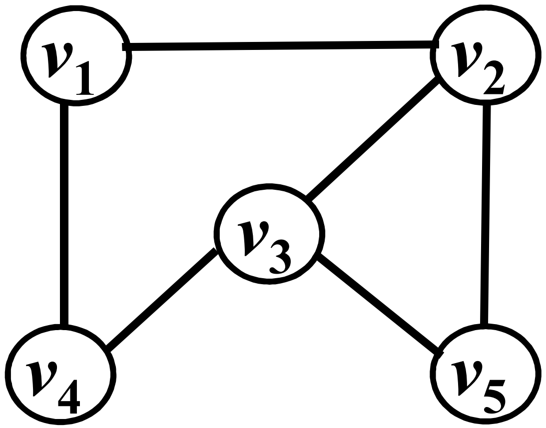
不同逻辑结构之间的比较

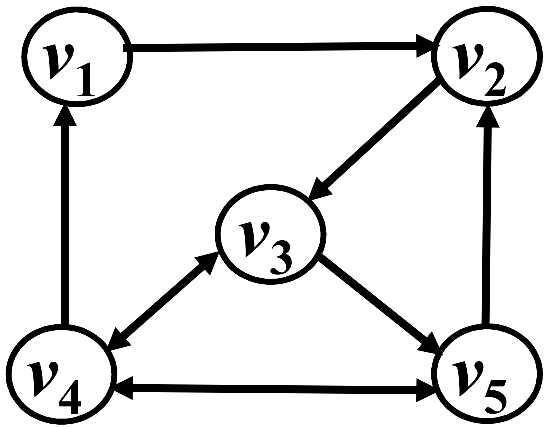


- 在线性结构中，元素之间的关系为前驱和后继；
- 在树型结构中，结点之间的关系为双亲和孩子；
- 在图型结构中，顶点之间的关系为邻接。

4.2 图的存储结构

- 是否可以采用顺序存储结构存储图?
  - 图的特点：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），无法通过存储位置表示这种任意的逻辑关系，所以，图无法采用顺序存储结构。
- 如何存储图？
  - 考虑图的定义，图是由顶点和边组成的；
  - 分别考虑如何存储顶点、如何存储边----顶点之间的关系。



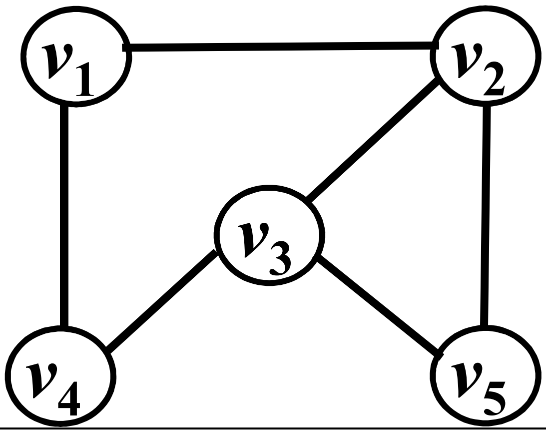


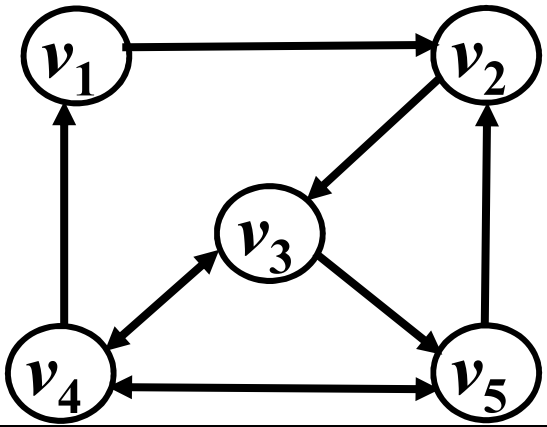
4.2 图的存储结构(cont.)

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$n \times n$的方阵，定义为：

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

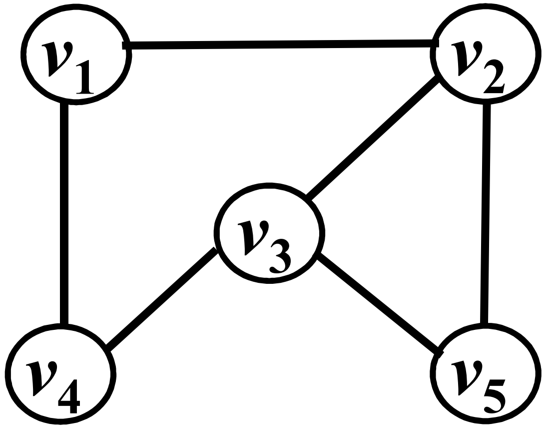




4.2 图的存储结构(cont.)

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

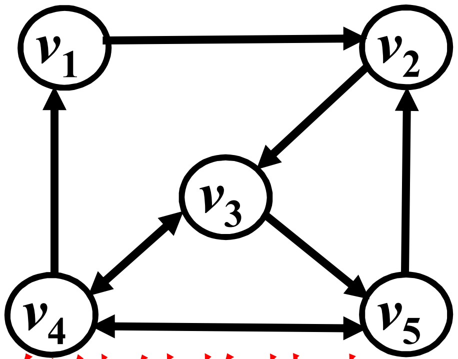
$$\text{edge}=\begin{array}{c}\begin{array}{ccccc} v_1 & v_2 & v_3 & v_4 & v_5 \end{array}\\ \begin{pmatrix} 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 \end{pmatrix}\end{array}\begin{array}{c} v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \end{array}$$

- 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 |

$$
\text{edge} = \begin{array}{c}
\begin{array}{ccccc} v_1 & v_2 & v_3 & v_4 & v_5 \end{array} \\
\left(\begin{array}{ccccc}
0 & 1 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 1 & 1 \\
1 & 0 & 1 & 0 & 1 \\
0 & 1 & 0 & 1 & 0
\end{array}\right)
\end{array}
\begin{array}{c} \\ v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \end{array}
$$

- 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？

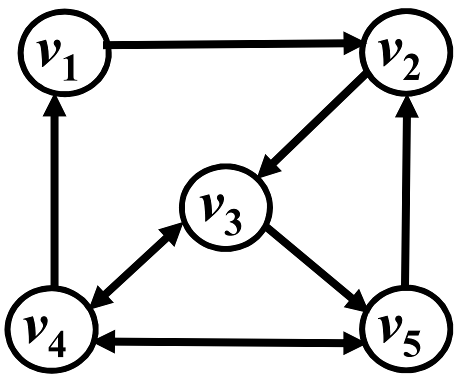
4.2 图的存储结构(cont.)

一、邻接矩阵 (*Adjacency Matrix*)表示(数组表示法)

- 存储结构定义：

> 假设图*G*有*n*个顶点*e*条边，则该图的存储需求为$O(n+n^2) = O(n^2)$，与边的条数e无关。

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



vertex

v_1	v_2	v_3	v_4	v_5

$$
\text{edge}=\begin{array}{c}
\begin{array}{ccccc} v_1 & v_2 & v_3 & v_4 & v_5 \end{array} \\
\begin{pmatrix}
0 & 1 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 1 & 1 \\
1 & 0 & 1 & 0 & 1 \\
0 & 1 & 0 & 1 & 0
\end{pmatrix}
\end{array}
\begin{array}{c} v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \end{array}
$$

4.2 图的存储结构(cont.)

- 存储结构的建立----算法实现的步骤：

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；

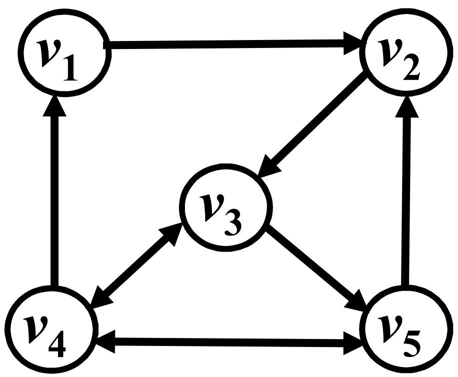
3.初始化邻接矩阵；

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。



vertex

v_1	v_2	v_3	v_4	v_5

$$\text{edge}=\begin{array}{c} \begin{array}{ccccc} v_1 & v_2 & v_3 & v_4 & v_5 \end{array} \\ \begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 \end{pmatrix} \end{array} \begin{array}{c} \\ v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \end{array}$$

4.2 图的存储结构(cont.)

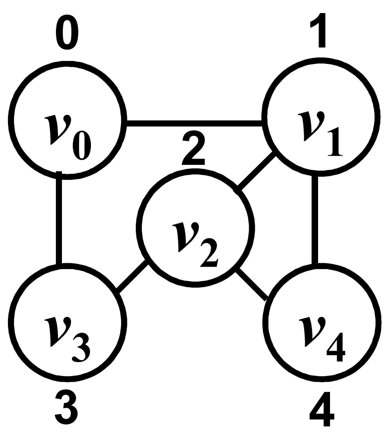
- 存储结构的建立算法的实现：

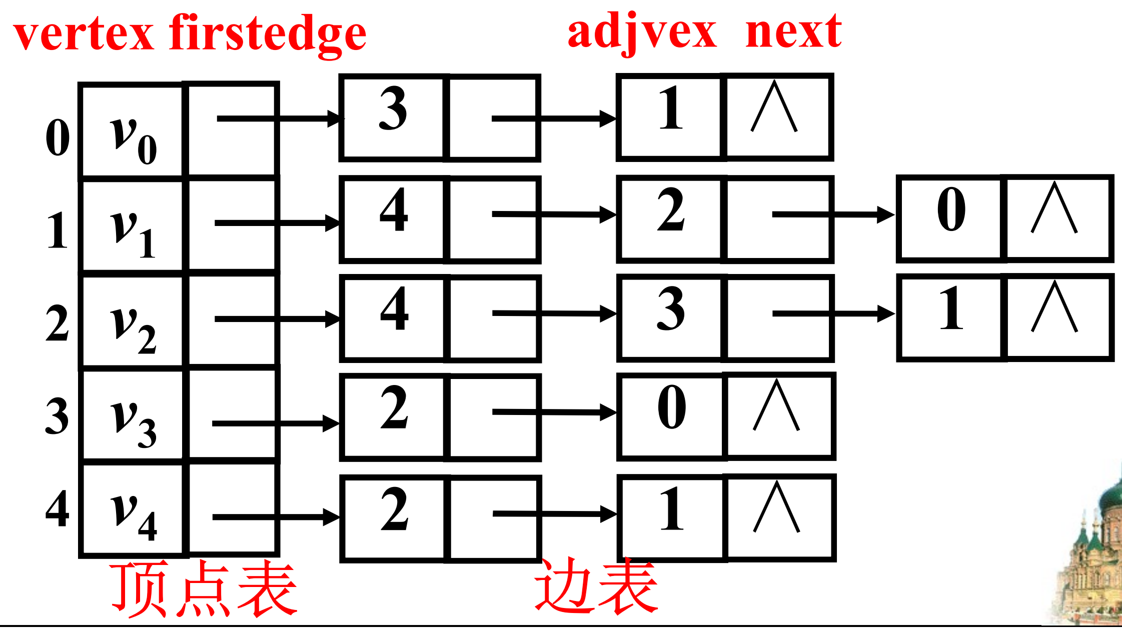
```
void  CreateMGragh (MTGragh *G) //建立图的邻接矩阵
{     int  i, j, k, w;
      cin>>G→n>>G→e;              //1.输入顶点数和边数
      for (i=0; i<G→n; i++)        //2.读入顶点信息，建立顶点表
           G→vexlist[i]=getchar( );
      for (i=0; i<G→n; i++)
           for (j=0;j<G→n;j++)
                G→edge[i][j]=0;       //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+ n² +2e) 。当e < <n, T = O( n² ) ?
```

4.2 图的存储结构(cont.)

二、邻接表(*Adjacency List*)表示

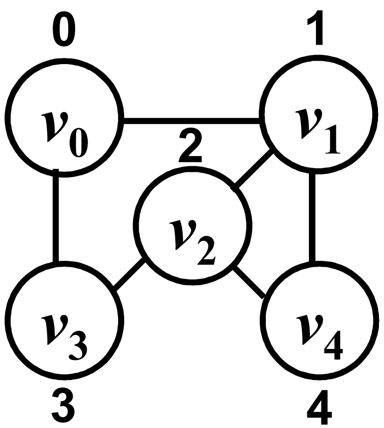
- 无向图的邻接表:
  - 对于无向图的每个顶点v_i，将所有与v_i相邻的顶点链成一个单链表，称为顶点v_i的边表（顶点v_i的邻接表）；
  - 再把所有边表的指针和顶点信息的一维数组构成顶点表。

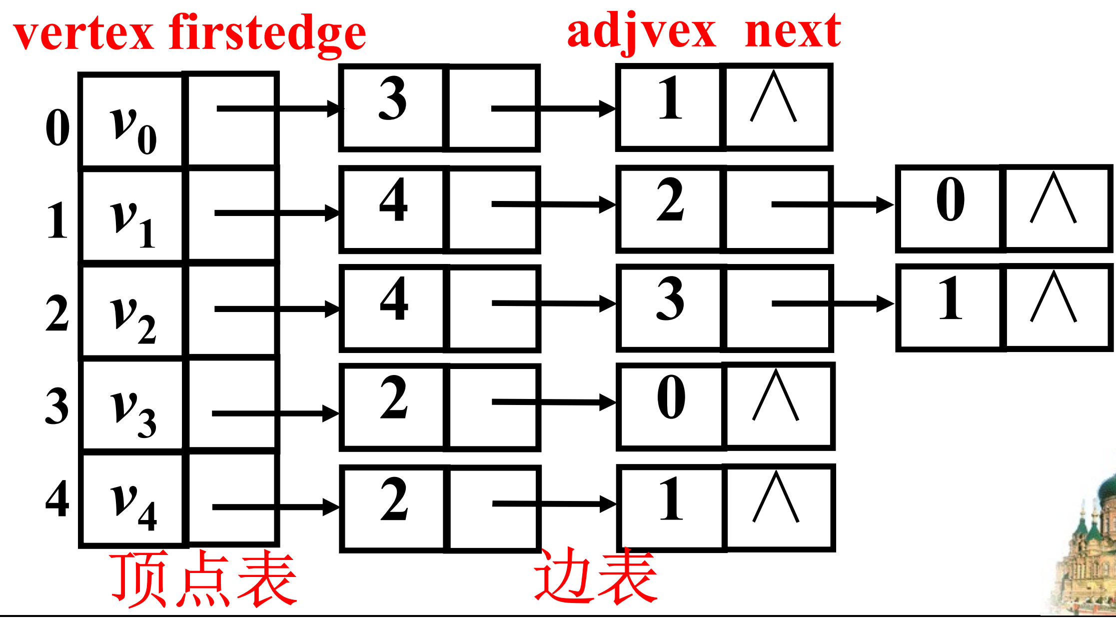




4.2 图的存储结构(cont.)

- 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)

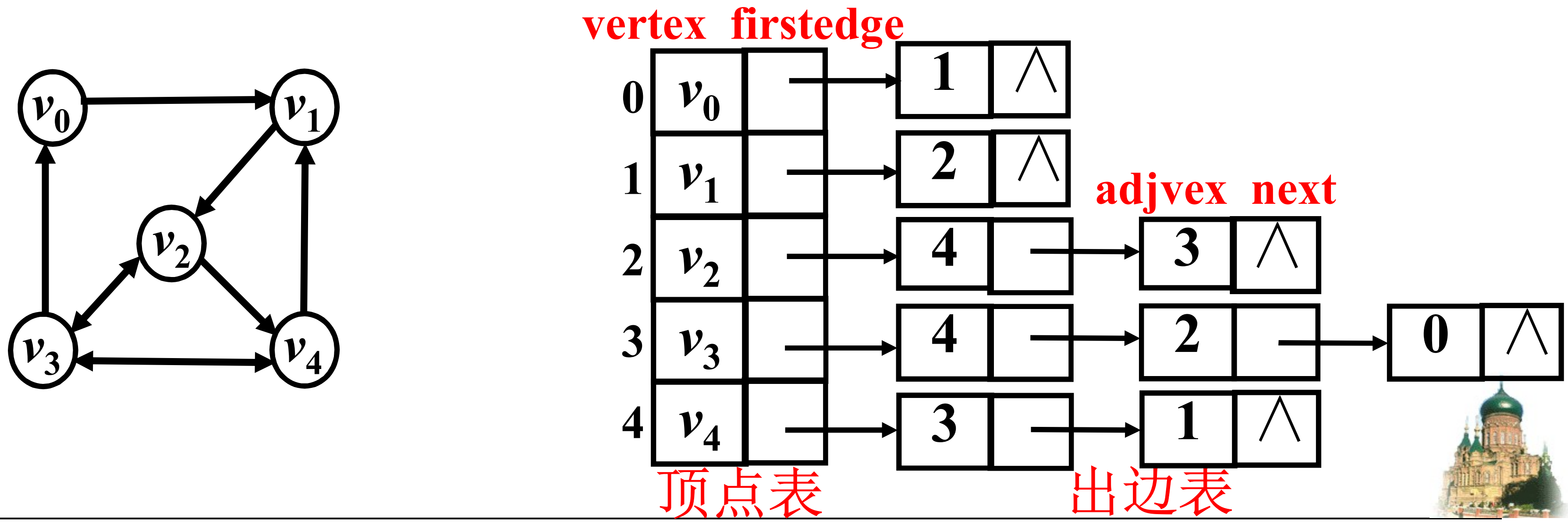




4.2 图的存储结构(cont.)

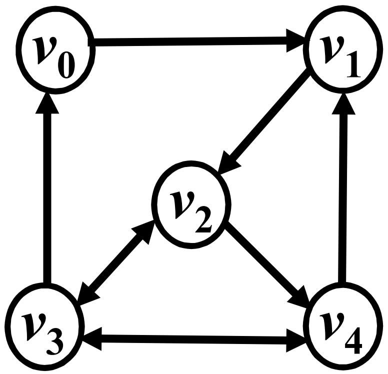
二、邻接表(*Adjacency List*)表示

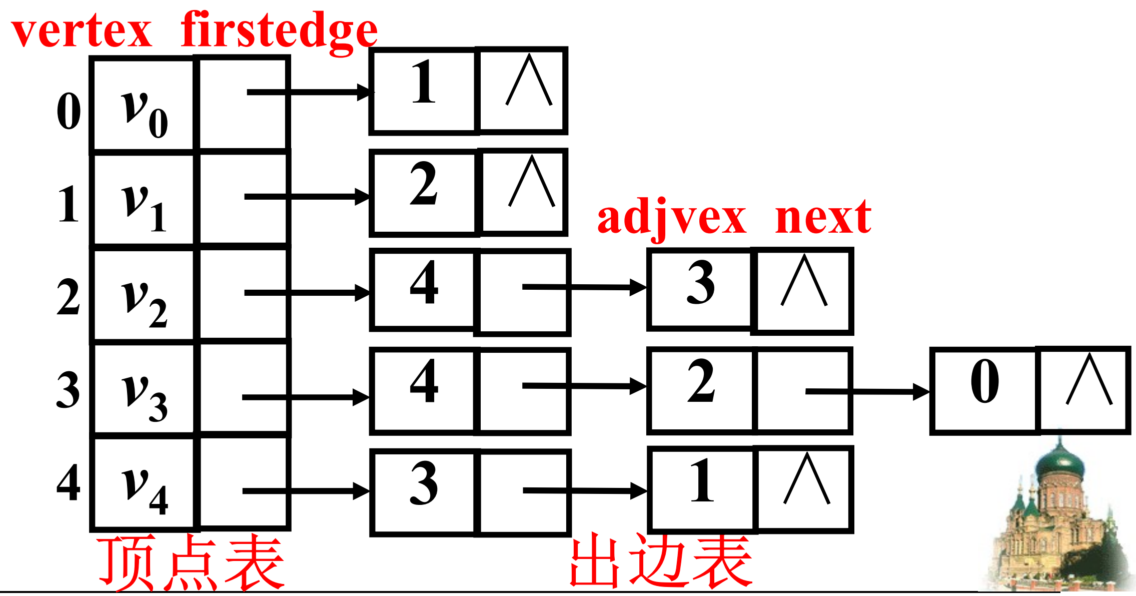
- 有向图的邻接表---正邻接表
  - 对于有向图的每个顶点v_i，将邻接于v_i的所有顶点链成一个单链表，称为顶点v_i的出边表；
  - 再把所有出边表的指针和顶点信息的一维数组构成顶点表.



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)

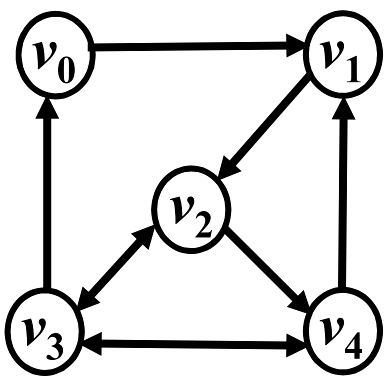


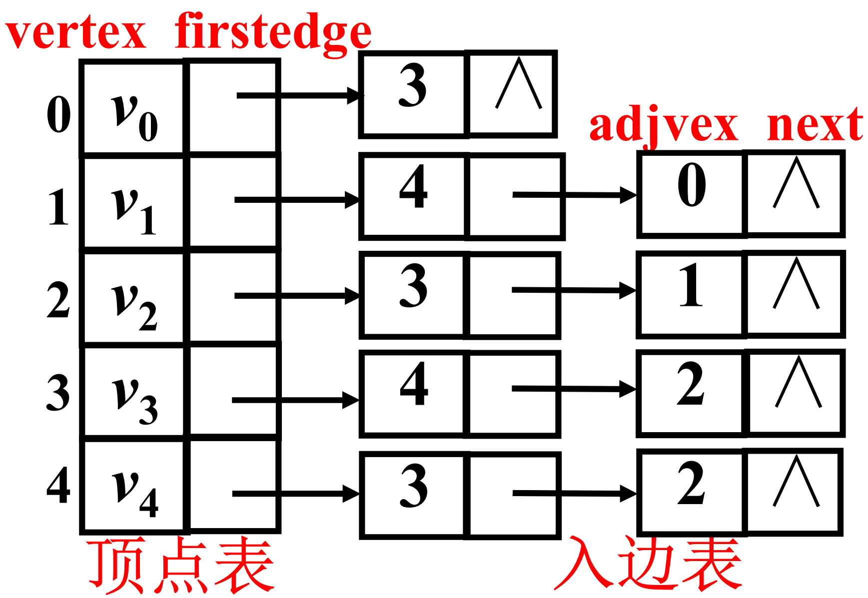


4.2 图的存储结构(cont.)

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。

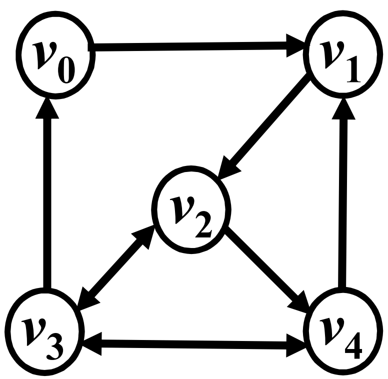


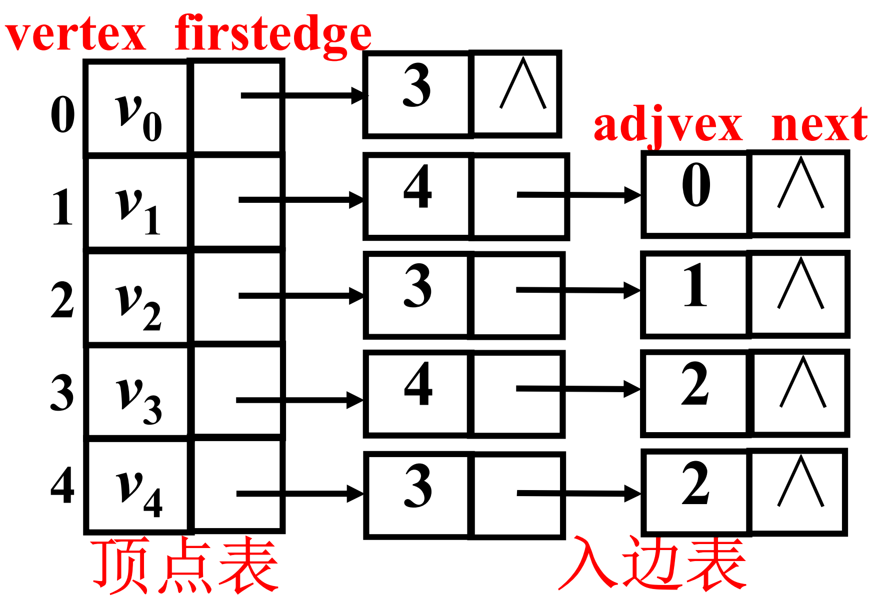


4.2 图的存储结构(cont.)

有向图的逆邻接表的存储特点

- 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)





4.2 图的存储结构(cont.)

- 邻接表存储结构的定义

```
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;
typedef struct {            //顶点表结点
    VertexData  vertex; //顶点数据域
    EdgeNode * firstedge;//边链表头指针
} VertexNode;
typedef struct {        //图的邻接表
   VertexNode vexlist [NumVertices];
    int n, e;            //顶点个数与边数
} AdjGraph;
```

边表结点

adjvex	cost	next

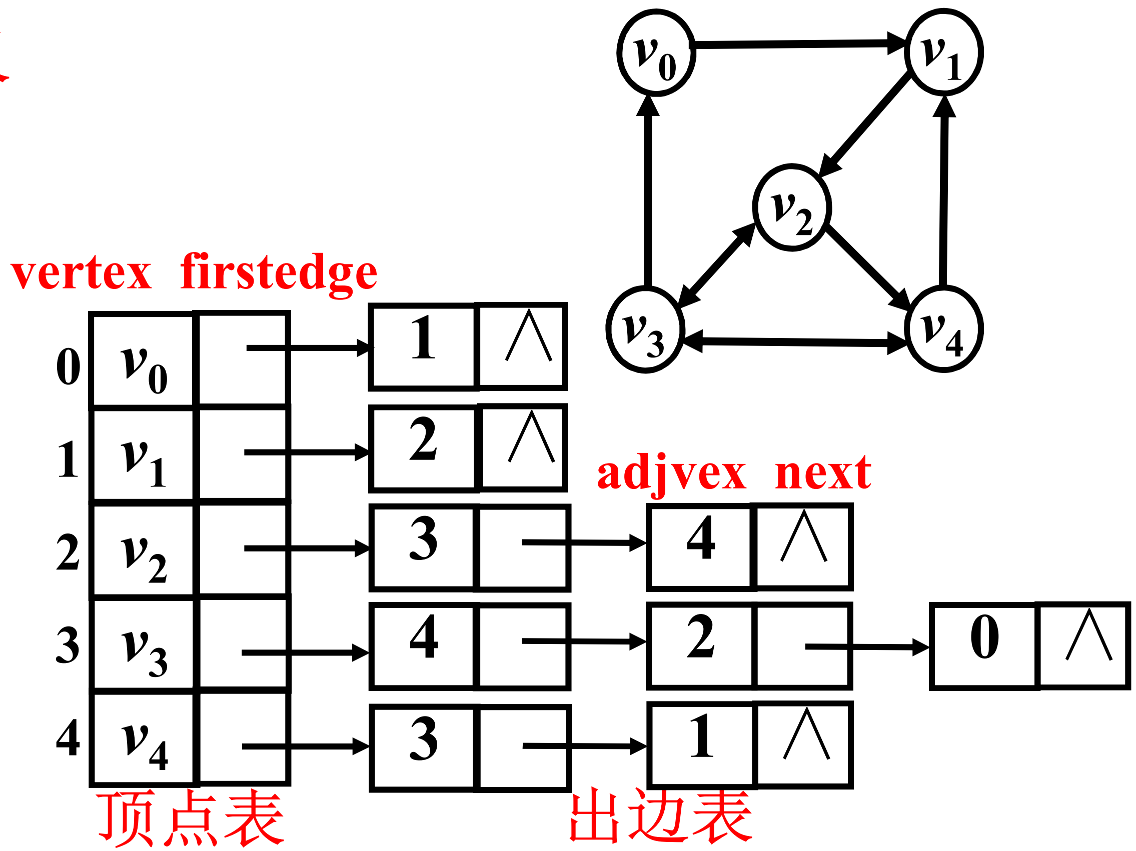
顶点表结点

vertex	firstedge

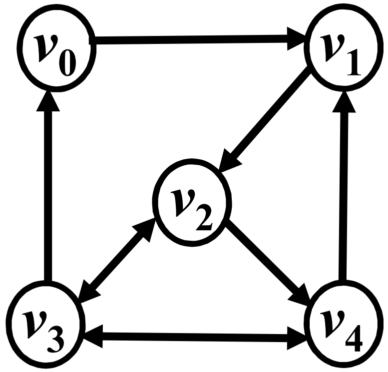
4.2 图的存储结构(cont.)

- 邻接表存储结构的定义

```
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;
```



4.2 图的存储结构(cont.)



- 邻接表存储结构建立算法实现的步骤：

1. 确定图的顶点个数和边的个数；

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. 依次输入边的信息并存储在边表中；

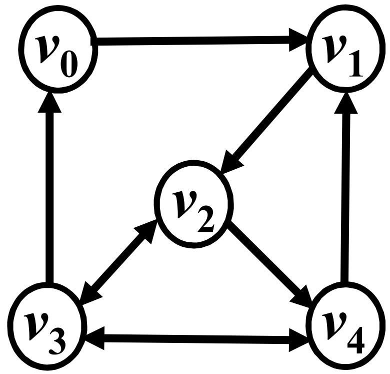
   3.1 输入边所依附的两个顶点的序号tail和head和权值w；

   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的实现：

```
void CreateGraph (AdjGraph G)
{  cin >> G.n >> G.e;                        //1.输入顶点个数和边数
   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建立边结点
      p→adjvex = head;   p→cost = weight;    //3.3设置边结点
      p→next = G.vexlist[tail].firstedge;    //3.4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+2e)
```

4.2 图的存储结构(cont.)

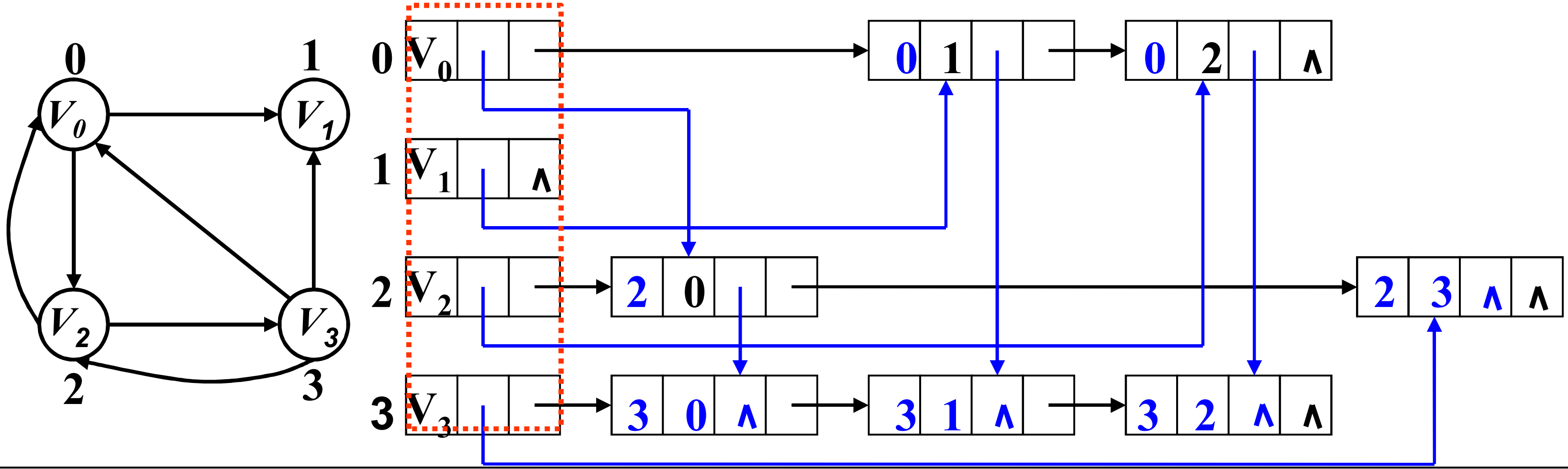
- 图的存储结构的比较——邻接矩阵和邻接表

	空间性能	时间性能	适用范围	唯一性
邻接矩阵	O（n^2）	O（n^2）	稠密图	唯一？
邻接表	O（n+e）	O（n+e）	稀疏图	不唯一？

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，是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
  - 即，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
  - 横向上看是正邻接表(出边表)，纵向上看是逆邻接表(入边表)
  - 方便有向图的顶点的入度与出度的计算



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

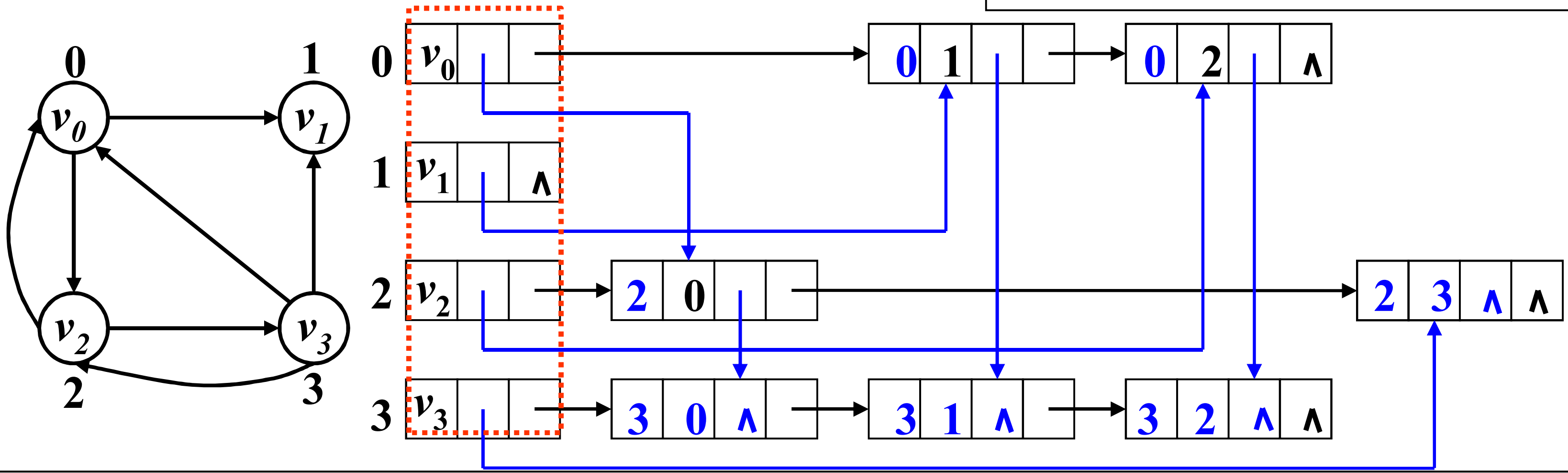
头结点（顶点结点）结构

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.)

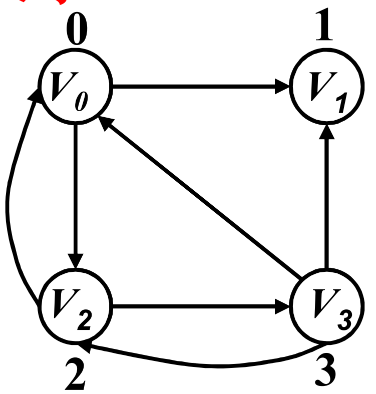
三、有向图的十字链表(*Orthogonal List*)表示

- 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.)

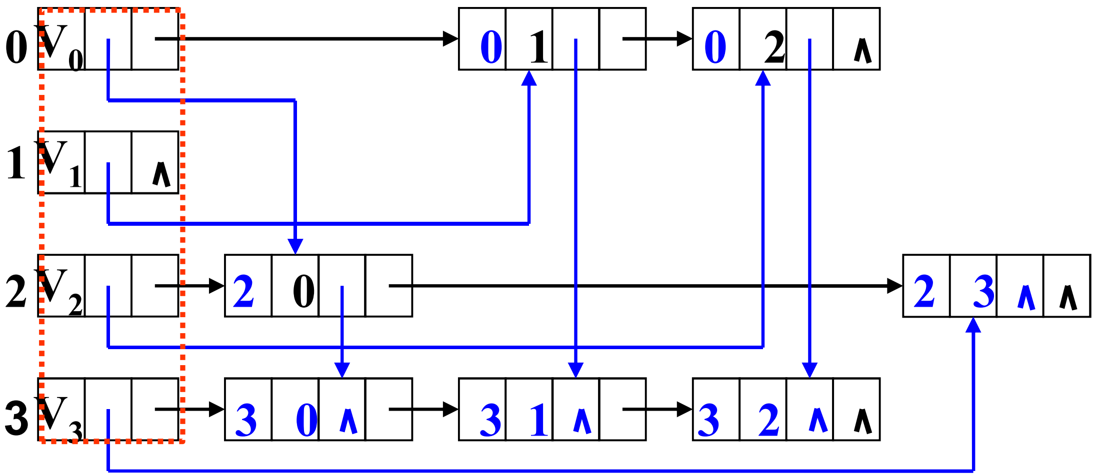
三、有向图的十字链表(*Orthogonal List*)表示



输入顺序:
3 2 <v3,v2>
3 1 <v3,v1>
3 0 <v3,v0>
2 3 <v2,v3>
2 0 <v2,v0>
0 2 <v0,v2>
0 1 <v0,v1>

- 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;
```



4.2 图的存储结构(cont.)

三、有向图的十字链表(*Orthogonal List*)表示

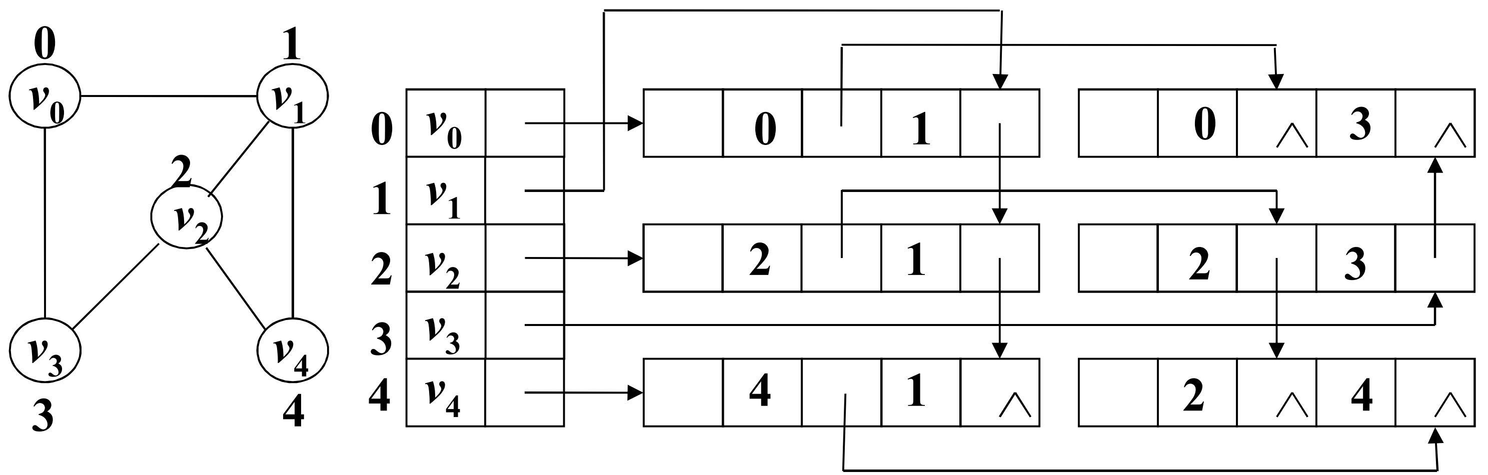
- 构建算法：

```
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) {                //构造顶点表
         scanf (&G.xlist[i].data);                               //输入顶点值
         G.xlist[i].firstin = NULL; G.xlist[i].firstout = NULL;   //初始化指针
     }
     for (k = 0; k < G.arcnum; + + k) {              //输入各弧并构造十字链表(边表)
          scanf (&v1, &v2);                                //输入一条弧的始点和终点
          i = LocateVex (G, v1);  j = LocateVex (G, v2);  //确定v1和v2在G中位置
          p = (ArcBox *) malloc (sizeof (ArcBox));         //假定有足够空间
         *p = {i, j, G.xlist[j].firstin, G.xlist[i].firstout, NULL};
                //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
```

4.2 图的存储结构(cont.)

四、无向图的邻接多重表(*Adjacency Multilist*)表示

- 邻接多重表，是对无向图的邻接矩阵的一种压缩表示
  - 这种结构在边的操作上会方便，如对已访问的边做标记，或要删除图中某条边，都需找到表示同一条边的两个结点
  - 邻接多重表的结构与十字链表类似。在邻接多重表中，所有依附于同一顶点的边串联在同一链表中，由于每条边依附两个顶点，则每个边结点同时链接在两个链表中。



4.2 图的存储结构(cont.)

四、无向图的邻接多重表(*Adjacency Multilist*)表示

- 结点结构：

边表的结点结构

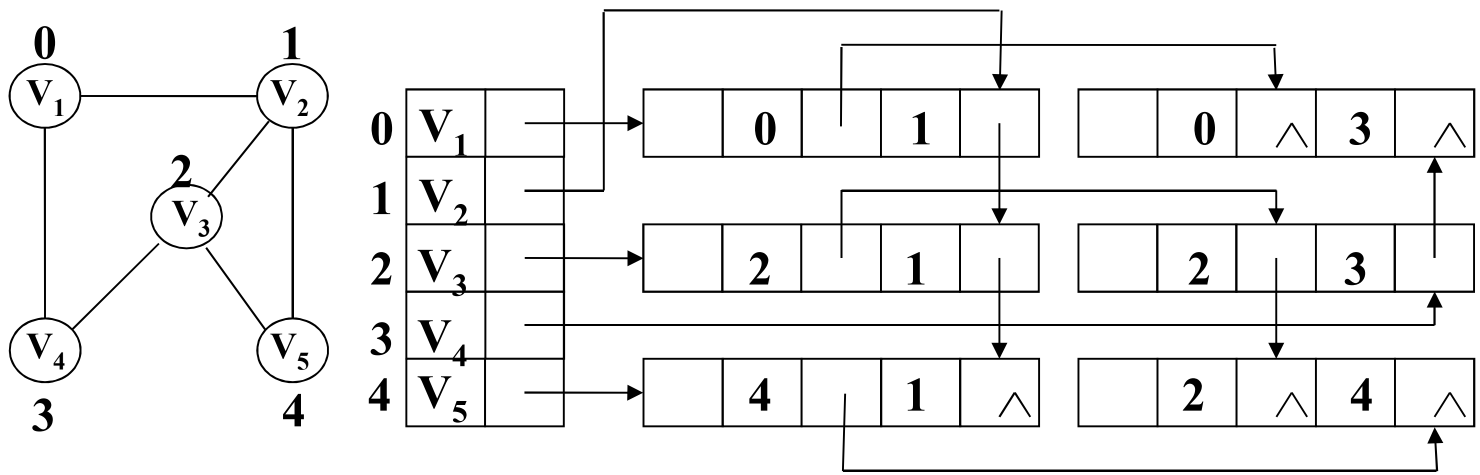
mark	ivex	ilink	jvex	jlink	info

顶点表的结点结构

data	firstedge

mark：标志域，用以标记该条边是否被搜索过
ivex和**jvex**：为该边依附的两个顶点在图中的位置
ilink：链域，指向下一条依附于顶点**ivex**的边
jlink：链域，指向下一条依附于顶点**jvex**的边
info：数据域，指向和边相关的各种信息的指针域

Data：数据域，存储和该顶点相关的信息
firstedge：链域，指示第一条依附于该顶点的边



4.2 图的存储结构(cont.)

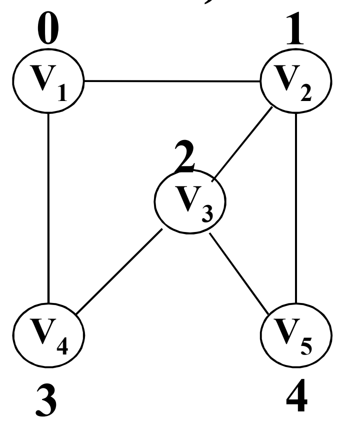
四、无向图的邻接多重表(*Adjacency Multilist*)表示

- 存储结构定义：

```
#define MAX_VERTEX_NUM   20
typedef emnu {unvisited, visited} VisitIf;
typedef struct EBox {
    VisitIf   mark;                         //边访问标记
    int  ivex, jvex;                        //该边依附的两个顶点的位置
    struct EBox  * ilink, * jlink;          //分别指向依附这两个顶点的下一条边
    InfoType   *info;                       //该边信息指针
} EBox;
typedef struct VexBox {
    VertexType   data;
    EBox   * firstedge;          //指向第一条依附于该顶点的边
} VexBox;
typedef struct {
    VexBox  adjmulist[MAX_VERTEX_NUM];
    int  vexnum, edgenum;                   //无向图的当前顶点数和边数
} AMLGraph;
```

4.2 图的存储结构(cont.)

四、无向图的邻接多重表(*Adjacency Multilist*)表示

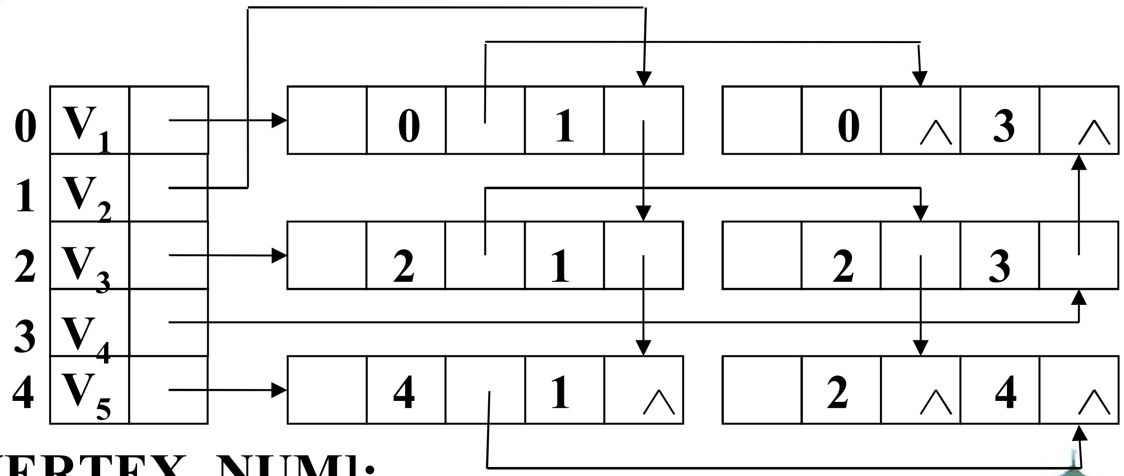


输入顺序：

0 3 (v_1,v_4)
2 4 (v_3,v_5)
2 3 (v_3,v_4)
4 1 (v_5,v_2)
2 1 (v_3,v_2)
0 1 (v_1,v_2)

- 存储结构定义：

```
#define MAX_VERTEX_NUM   20
typedef emnu {unvisited, visited} VisitIf;
typedef struct EBox {
    VisitIf   mark;
    int  ivex, jvex;
    struct EBox  * ilink, * jlink;
    InfoType   *info;
} EBox;
typedef struct VexBox {
    VertexType   data;
    EBox    * firstedge;
} VexBox;
typedef struct {
    VexBox  adjmulist[MAX_VERTEX_NUM];
     int  vexnum, edgenum;
} AMLGraph;
```



4.2 图的存储结构(cont.)

四、无向图的邻接多重表(*Adjacency Multilist*)表示

- 构建算法:

```
void CreateUDG_AML(AMLGraph &G)   //用邻接多重表存储，构造无向图G
{   string v1, v2;   int i, j, k;
    cin>>G.vexnum>>G.arcnum;
    for(i=0;i<G.vexnum;i++) {      //建立顶点表
        cin>>G.adjmulist[i].data; G.adjmulist[i].firstedge=NULL;
    }
    for(k=0;k<G.arcnum;k++)  {  // 建立边表
        cin>>v1>>v2;
        i=LocateVex(G,v1);  j=LocateVex(G,v2);
        while(i<0|| i>G.vexnum-1 || j<0 || j>G.vexnum-1) {
            cout<<"结点位置输入错误,重新输入: ";
            cin>>v1>>v2;
            i=LocateVex(G,v1);  j=LocateVex(G,v2);
        }
        EBox *p=new EBox;
        p->ivex=i;   p->jvex=j;  p->mark=0;
        p->ilink=G.adjmulist[i].firstedge;     p->jlink=G.adjmulist[j].firstedge;
        G.adjmulist[i].firstedge = G.adjmulist[j].firstedge = p;
    }
} // CreateUDG
```

示例中边的一种输入顺序是：0 3； 2 4； 2 3； 4 1； 2 1； 0 1

4.3 图的搜索(遍历)

John Edward Hopcroft

Robert Endre Tarjan

1986年图灵奖获得者

约翰·霍普克洛夫特1939年生于西雅图。美国国家科学院和工程院院士、康奈尔大学智能机器人实验室主任。1962和1964年获斯坦福大学硕士和博士学位。先后在普林斯顿大学、斯坦福大学等工作，也曾任职于一些科学研究机构如NSF和NRC。著作很多如《算法设计与分析基础》《 数据结构与算法》《自动机理论、语言和计算导论》《形式语言及其与自动机的关系》

罗伯特·塔扬普林斯顿大学计算机科学系教授，1948年4月30日生于加利福尼亚州。1969年本科毕业，进入斯坦福大学研究生院，1972年获得博士学位。平面图测试的高效算法；合并-搜索问题；“分摊”算法的概念；八字形树；持久性数据结构

4.3 图的搜索(遍历)(cont.)

- 图的遍历（图的搜索）
  - 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，顶点是没有确定的先后次序的，所以，顶点的编号不唯一。

4.3 图的搜索(遍历)(cont.)

- 图的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”,在访问完某个顶点之后可能会沿着某些边又回到了曾访问过的顶点。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]** 。
  - 在图中，一个顶点可以和其它多个顶点相连，当这样的顶点访问过后，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**)。

4.3 图的搜索(遍历)(cont.)

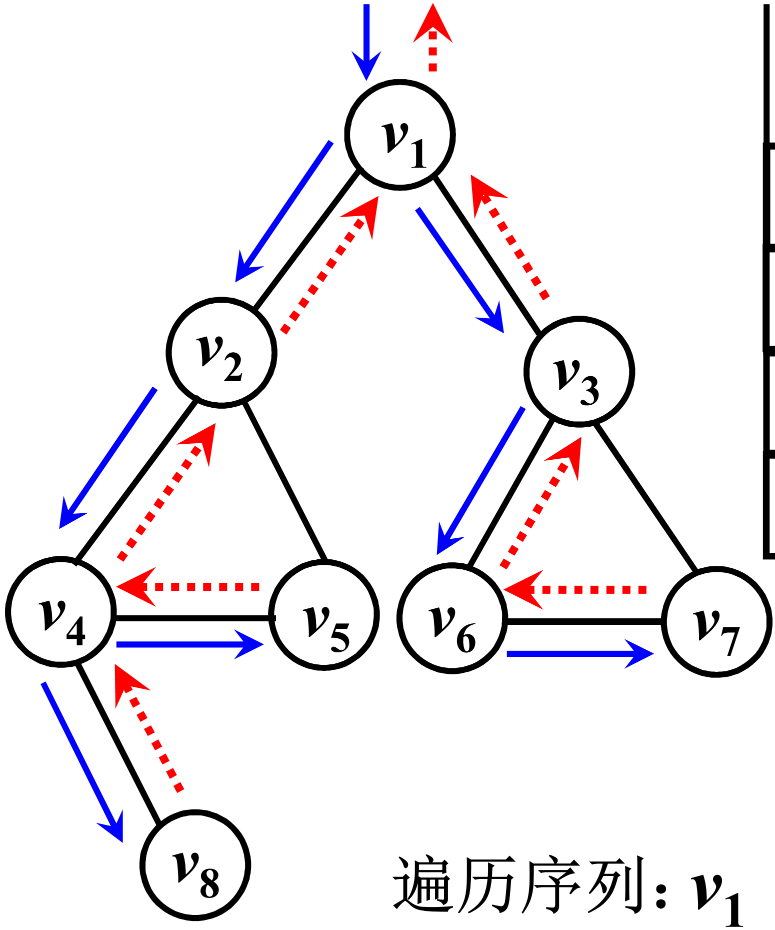
- 深度优先搜索(Depth-First-Search) ----类似于树结构的先序遍历
  - 设图G的初态是所有顶点都“未访问过(False)”，在G中任选一个顶点 v 为初始出发点(源点)，则深度优先搜索可定义为:
  - ①首先访问出发点 v，并将其标记为“访问过 (True) ”；
  - ②然后，从 v 出发，依次考察与 v 相邻(邻接于或邻接到v)的顶点 w；若 w “未访问过(False)”，则以 w 为新的出发点递归地进行深度优先搜索，直到图中所有与源点 v 有路径相通的顶点（亦称从源点可到达的顶点）均被访问为止；(从源点出发的一次先深搜索)
  - ③若此时图中仍有未被访问过的顶点，则另选一个“未访问过”的顶点作为新的搜索起点，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(n²)

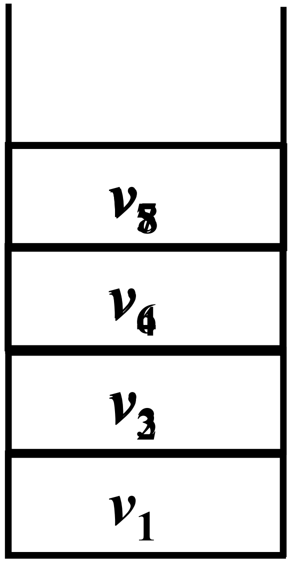
4.3 图的搜索(遍历)(cont.)

- 深度优先遍历示例

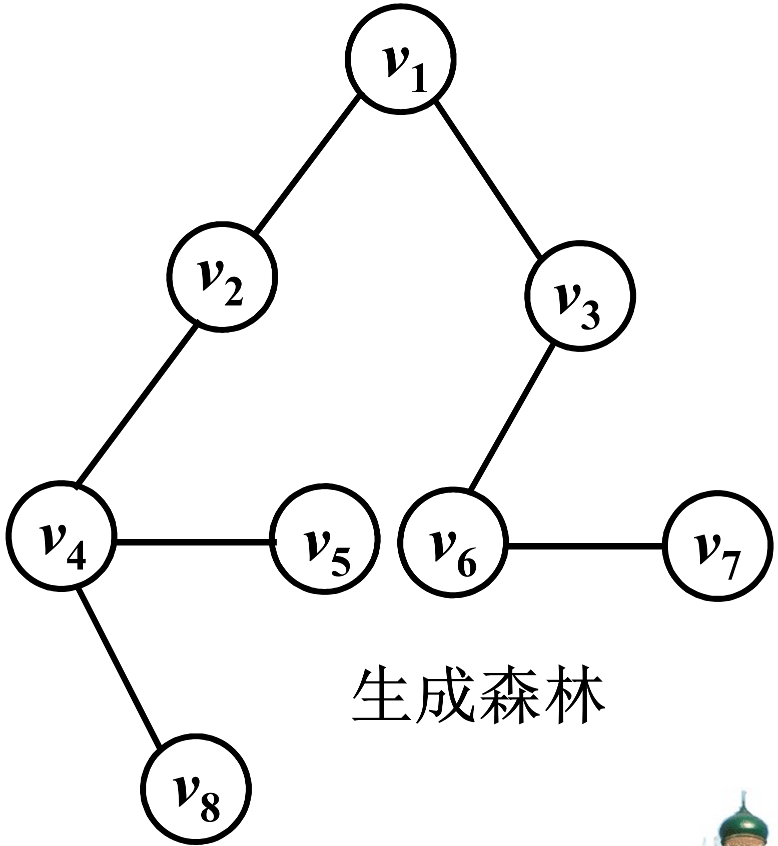
深度优先需要无路可走时按照来路往回退，正好是后进先出

- 深度优先遍历序列?入栈序列?出栈序列?





栈



生成森林

遍历序列：v_1 v_2 v_4 v_5 v_8 v_3 v_6 v_7

4.3 图的搜索(遍历)(cont.)

- 深度优先遍历特点：
  - 是递归的定义，是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。
- 生成森林（树）：
  - 由原图的所有顶点和搜索过程中所经过的边构成的子图。
- 先深搜索结果不唯一
  - 即图的**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 DFSTraverse (AdjGraph G)     //主算法
/* 先深搜索一邻接表表示的图G；而以邻接矩阵表示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 出发的一次搜索，BFSX(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**visited[v]={0,…};**

1. 访问顶点**v; 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**出发递归地执行该算法;

  3.2 w=顶点**v**的下一个邻接点;

4.3 图的搜索(遍历)(cont.)

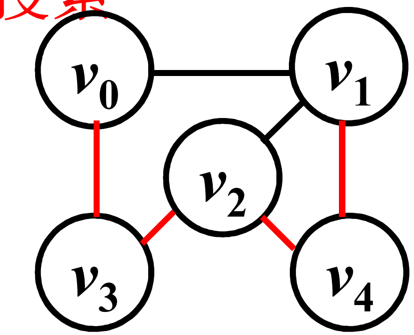
- 从一个顶点出发的一次深度优先遍历算法：

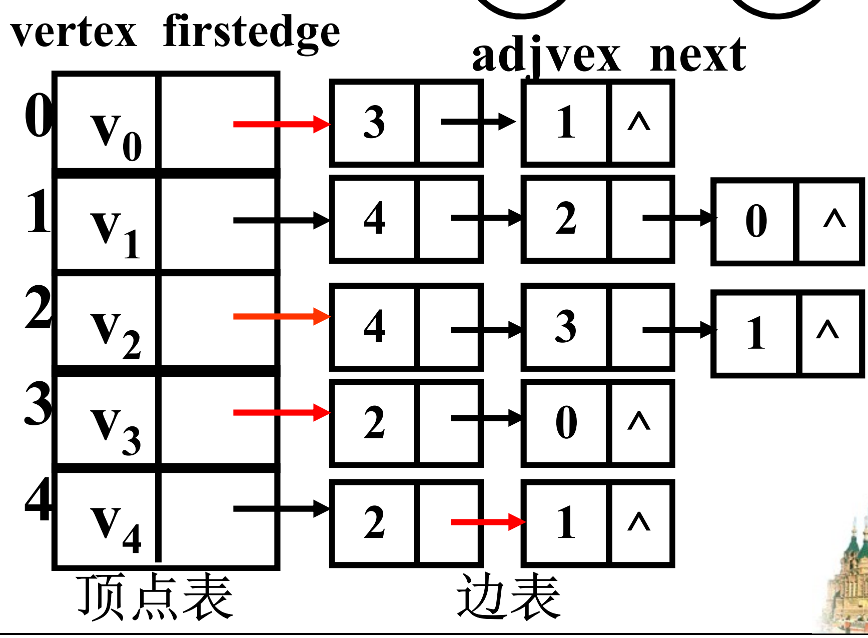
```
void DFS1 (AdjGraph* G, int i)
//以vi为出发点时对邻接表表示的图G进行先深搜索
{     EdgeNode *p;
      cout<<G→vexlist[i].vertex;     //访问顶点vi;
      visited[i]=TRUE;               //标记vi已访问
      dfn[i]=count++;                //对vi进行编号
      p=G→vexlist[i].firstedge;      //取vi边表的头指针
      while( p ) {   //依次搜索vi的邻接点vj, 这里j=p->adjvex
          if( !visited[ p→adjvex ] )  //若vj尚未访问
              DFS1(G, p→adjvex);      //则以vj为出发点先深搜索
          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:

```
void DFS1 (AdjGraph* G, int i)
//以vi为出发点时对邻接表表示的图G进行先深搜索
{     EdgeNode *p;
      cout<<G→vexlist[i].vertex;
      visited[i]=TRUE;
      dfn[i]=count++;
      p=G→vexlist[i].firstedge;
      while( p ) {
         if( !visited[ p→adjvex ] )
            DFS1(G, p→adjvex);
         p=p→next;
      }
} //DFS1
```



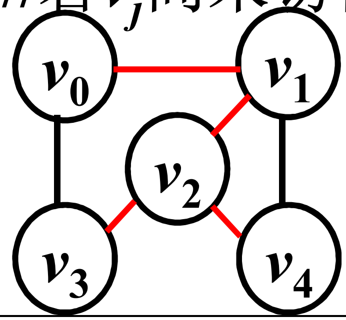


4.3 图的搜索(遍历)(cont.)

- 从一个顶点出发的一次深度优先遍历算法：

```
void DFS2(MTGraph *G, int i)
//以vi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vi
    visited[i]=TRUE;           //标记vi已访问
    dfn[i]=count;              //对vi进行编号
    count ++;                  //下一个顶点的编号
    for( j=0; j<G→n; j++ ) //依次搜索vi的邻接点
        if ( (G→edge[i][j] == 1)&&!visited[i] ) //若vj尚未访问
            DFS2( G, j );
}//DFS2
```

$$\begin{array}{c c}
\begin{array}{ccccc} v_0 & v_1 & v_2 & v_3 & v_4 \end{array} & \\
\begin{bmatrix} 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 \end{bmatrix} & \begin{array}{c} v_0 \\ v_1 \\ v_2 \\ v_3 \\ v_4 \end{array}
\end{array}$$



4.3 图的搜索(遍历)(cont.)

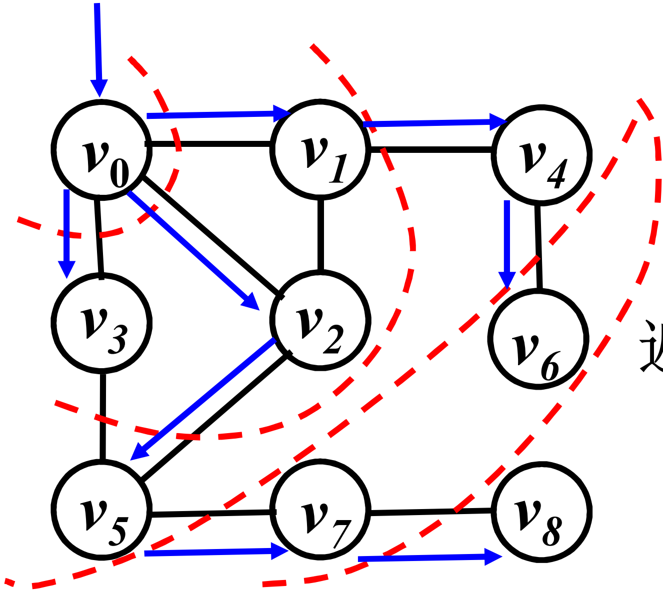
- 广度优先搜索(Breadth-First-Search)---类似于树的层序遍历
  - 设图G的初态是所有顶点都“未访问过(False)”，在G中任选一个顶点 v 为源点，则广度优先搜索可定义为:
  - ①首先访问出发点 v，并将其标记为“访问过 (True)”；
  - ②接着依次访问所有与 v 相邻的顶点w_1，w_2…w_t;
  - ③然后依次访问与w_1，w_2… w_t相邻的所有未访问的顶点;
  - ④依次类推，直至图中所有与源点v有路相通的顶点都已访问过为止；(从源点出发的一次先广搜索)
  - ⑤此时，从 v 开始的搜索结束，若G是连通的，则遍历完成；否则在G中另选一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：$O(n^2)$

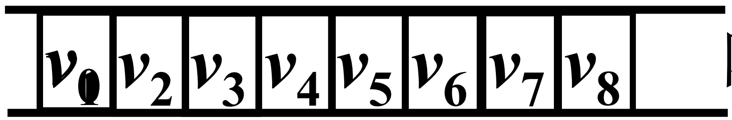
4.3 图的搜索(遍历)(cont.)

- 广度优先遍历示例

广度优先需要保证先访问顶点的未访问邻接点先访问，恰好就是先进先出

- 广度优先遍历序列?入队序列?出队序列?

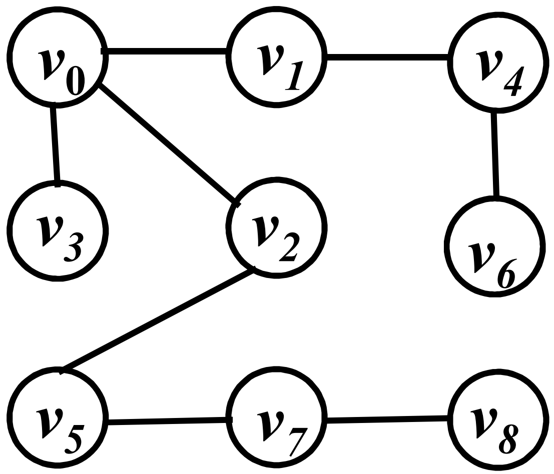




队列

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。
- 生成森林（树）：
  - 有原图的所有顶点和搜索过程中所经过的边构成的子图。
- 先广搜索结果不唯一：
  - 即图的**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
/* 先广搜索一邻接表表示的图G；而以邻接矩阵表示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.)

```
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的邻接点 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为出发点时对用邻接表表示的图G进行先广搜索
```

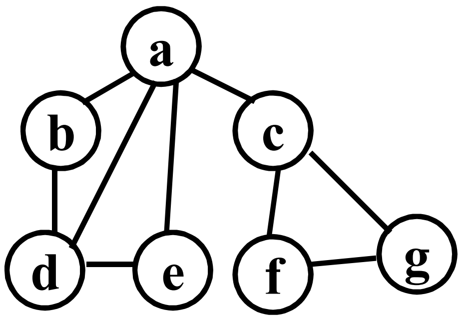
4.3 图的搜索(遍历)(cont.)

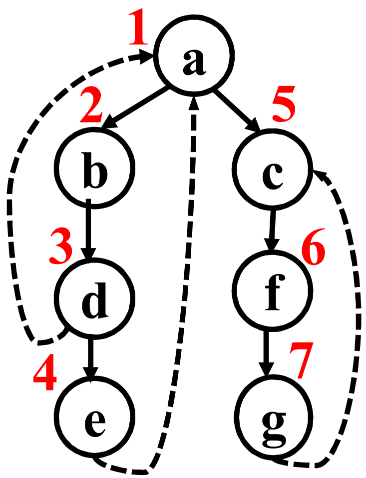
```
void BFS2 (MTGraph *G, int k) //这里没有进行先广编号
{    int i , j; QUEUE Q;   MAKENULL(Q);
     cout << G→vexlist[ k ];   //访问vk
     visited[ k ] = TRUE; //给vk作访问过标记
     ENQUEUE (k, Q);   // vk进队列
     while ( ! Empty (Q) ) {  //队空时搜索结束
          i=DEQUEUE(Q); //vi出队
          for(j=0; j<G→n; j++) {   //依次搜索vi的邻接点 vj
               if ( G→edge[ i ][ j ] ==1 && !visited[ j ]) { //若vj未访问过
                    cout << G→vexlist[ j ];//访问vj
                    visited[ j ]=TRUE;  //给vj作访问过标记
                    ENQUEUE ( j , Q ); //访问过的vj入队
               }
          } //重复检测 vi的所有邻接顶点
     } //外层循环，判队列空否
} // 以vk为出发点时对用邻接矩阵表示的图G进行先广搜索
```

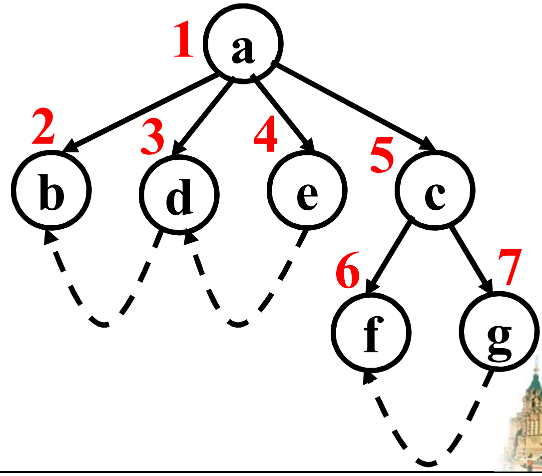
无向图(的搜索)及其应用

先深生成森林和先广生成森林

- 搜索的结果
  - 先深或先广生成森林、顶点的线性序列(和先深或先广编号)
  - 树边与非树边
  - 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)







无向图(的搜索)及其应用

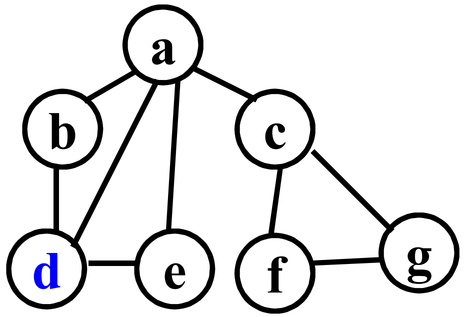
深度优先搜索过程中对边的分类----分成两类

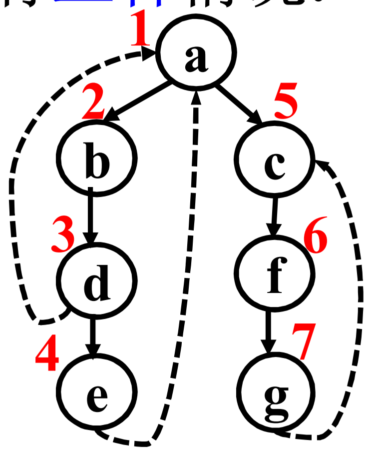
- 树边—在搜索过程中所经过的边；回退边—图中的其它边
- 特点: 树边是从先深编号较小的指向较大的顶点;回退边相反;
- 如何在搜索过程中区分树边和回退边？
  - 设v是刚访问过的顶点True，下面搜索到 w, w 有三种情况:

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；(d,b)

3.w是True且w不是v的父亲,则(v,w)是回退边。(d,a)



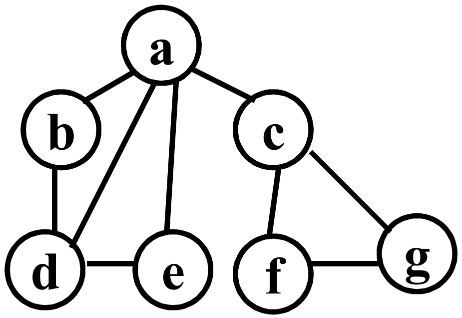


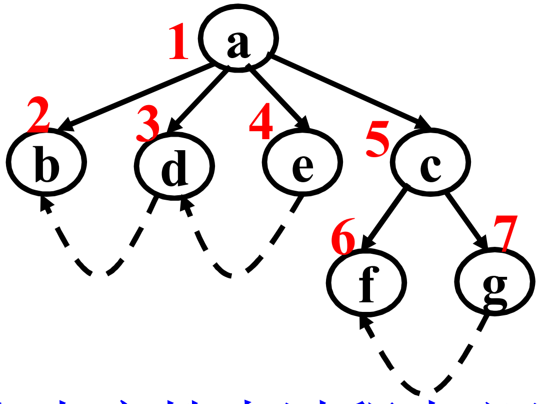
- 结论:若G中存在环路,则在先深搜索过程中必遇到回退边;反之亦然

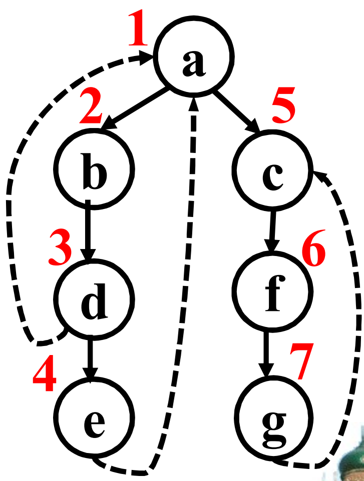
无向图(的搜索)及其应用

广度优先搜索过程中对边的分类

- 两类:
  - 树边—在搜索过程中所经过的边;
  - 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.



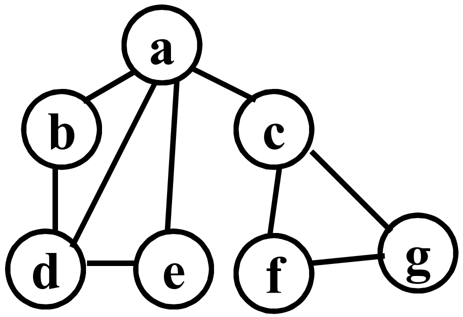


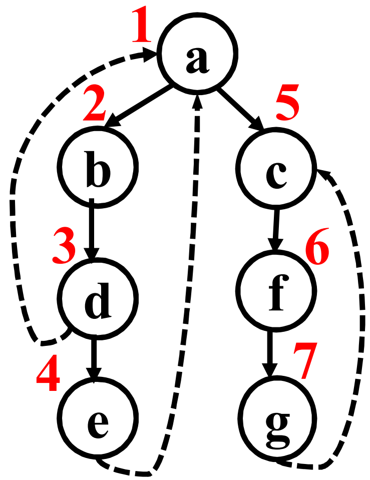


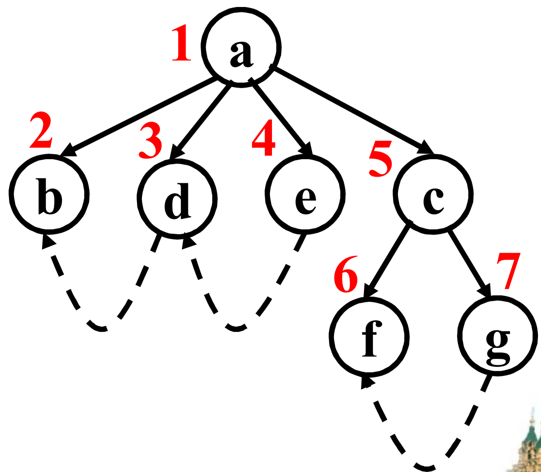
- 结论:若G中存在环路,则在先广搜索过程中必遇到横边;反之亦然

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
    - 判断是否是双连通的
    - 求关节点和双连通分量







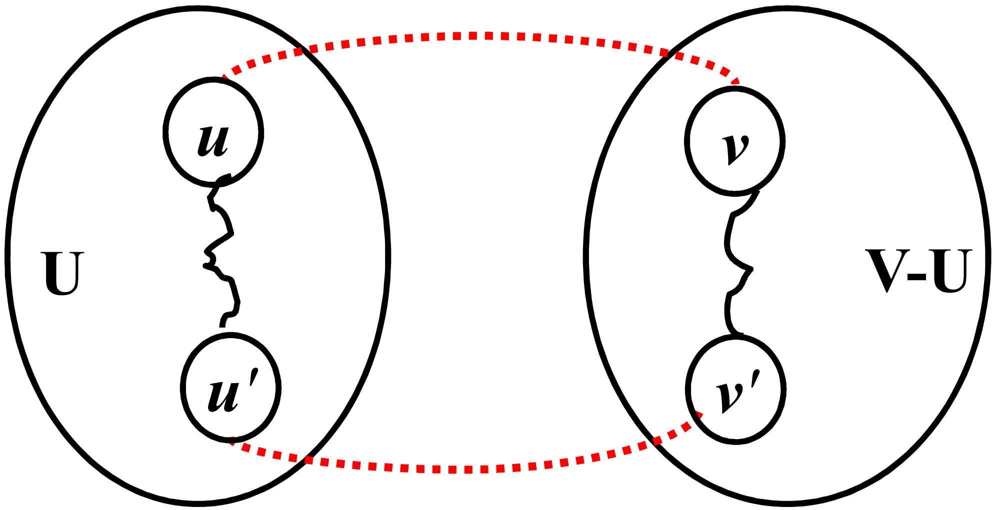
4.4 最小生成树算法

- 生成树的代价
  - 设G = (V, E) 是一个无向连通网， E中每一条边(*u* , *v*)上的权值*c*(*u* , *v*)， 称为(*u* , *v*)的边长。
  - 图G的生成树上各边的权值(边长)之和称为该生成树的代价
- 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
- 最小生成树可以应用到许多实际问题
  - 例如，在*n*个教室之间建局域网络，至少要架设*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.)

最小生成树的性质----贪心选择性

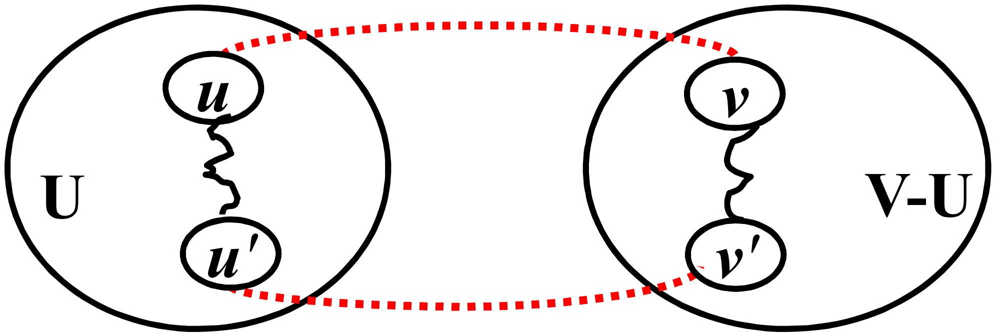
- 假设G =（V, E）是一个连通网，U是顶点V的一个非空真子集。若(*u* , *v*)是一条具有最小权值（代价）的边，其中 $u \in$ U, $v \in$ V-U, 则必存在一棵包含边(*u* , *v*)的最小生成树。
- 此性质保证了*Prim*和Kruskal贪心算法的正确性



4.4 最小生成树算法(cont.)

MST性质的证明

- [反证]假设G的任何一棵最小生成树都不包含(*u* , *v*)，设 T是连通网的一棵最小生成树，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(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={ }

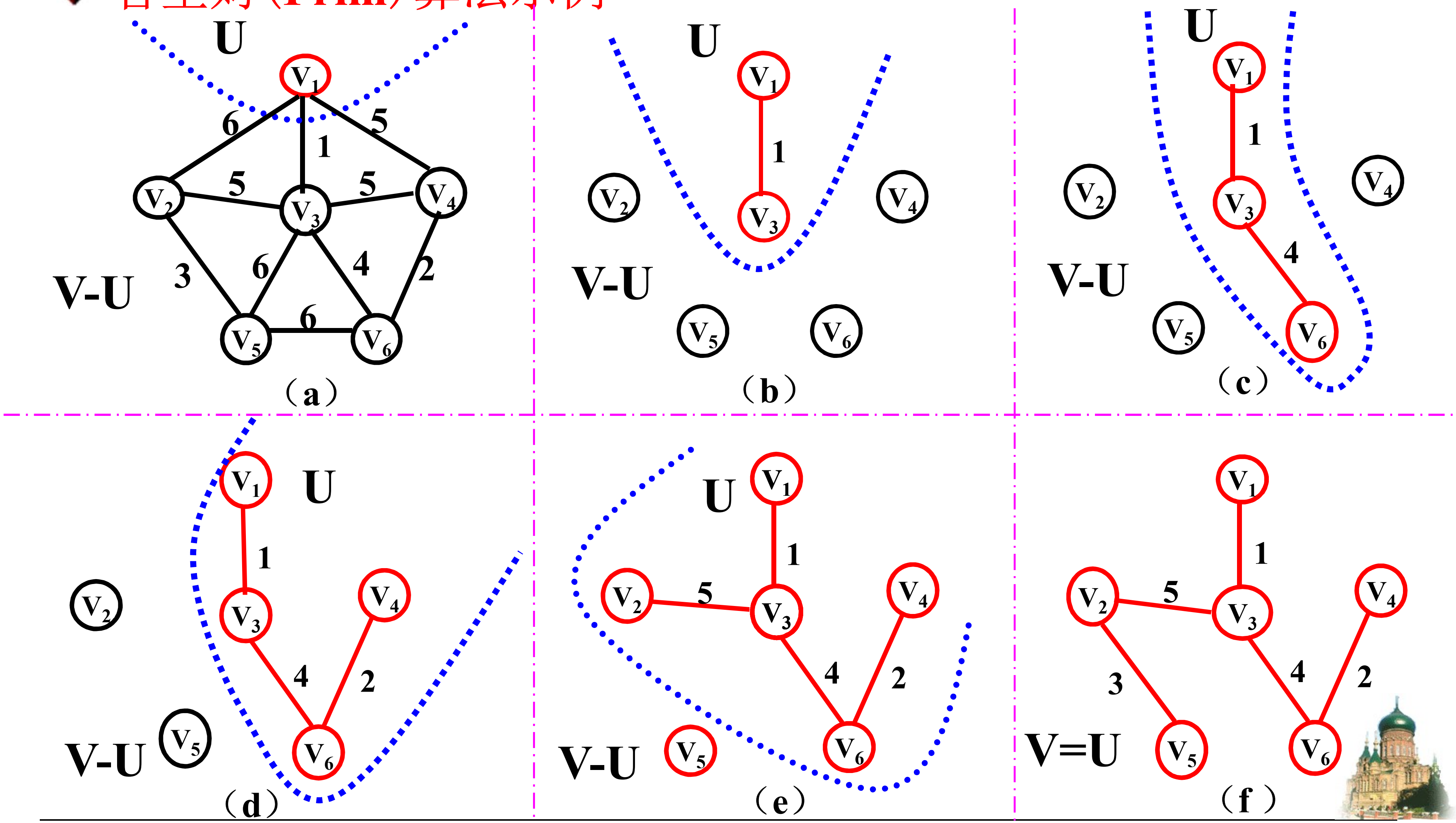
② 然后找出权值最小的边(u，v)，且$u \in U$，$v \in (V-U)$，将边加入TE，并将顶点v加入集合U

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，可以用下述方法构造候选最短边集：对于V-U中的每个顶点，保存从该顶点到U中的各顶点的最短边。

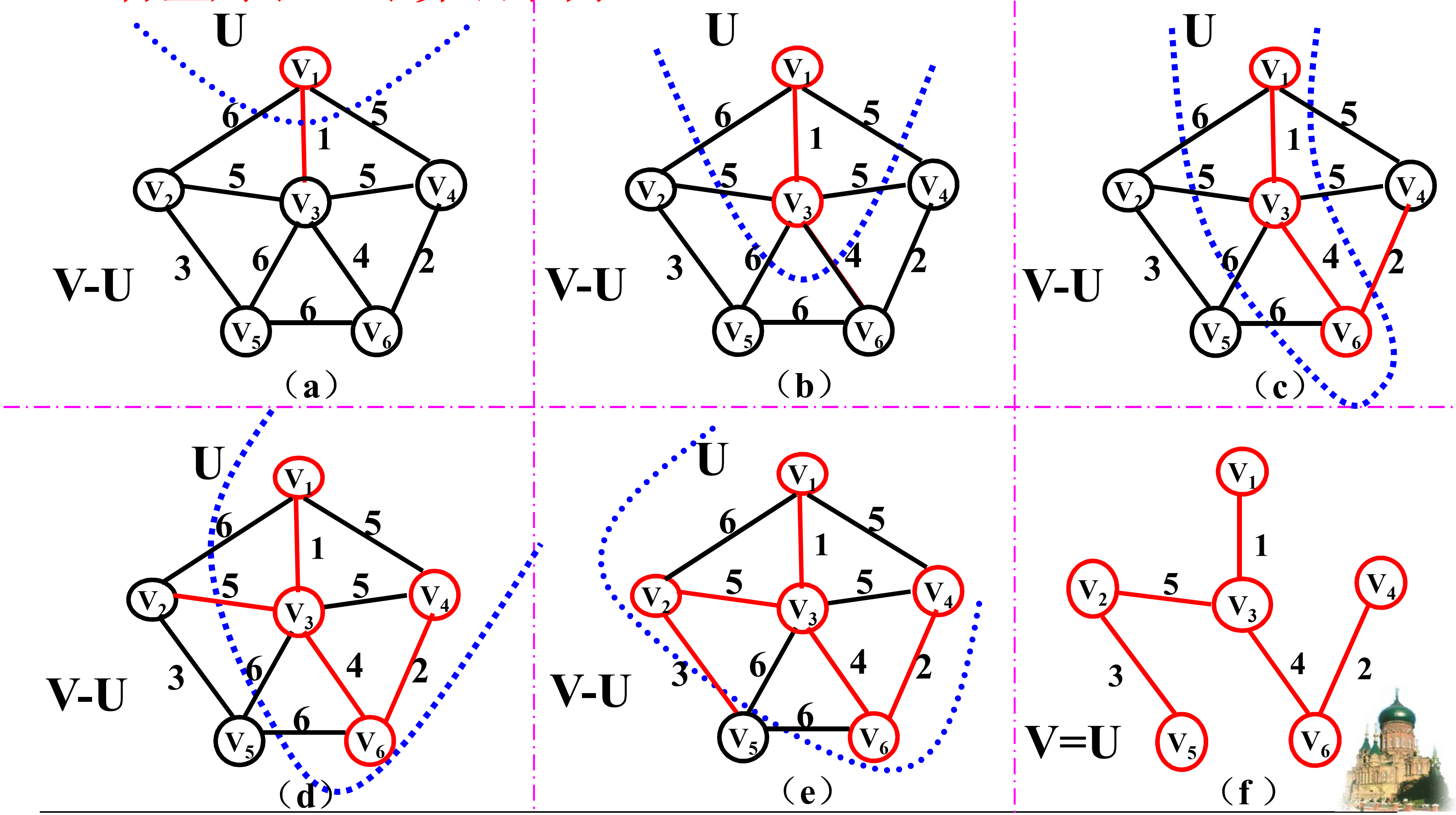
4.4 最小生成树算法(cont.)

- 普里姆(Prim)算法示例



4.4 最小生成树算法(cont.)

- 普里姆(Prim)算法示例



4.4 最小生成树算法(cont.)

- 普里姆(Prim)算法的实现
  - 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？

$$\left.\begin{array}{l}\text{LOWCOST}[i]=w\\ \text{CLOSSET}[i]=k\end{array}\right\}\begin{array}{l}\text{表示顶点}v_i\text{和顶点}v_k\text{之间的权值}\\ \text{为}w\text{，其中：}v_i\in \text{V-U 且}v_k\in \text{U}\end{array}$$

  - 如何更新？

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

4.4 最小生成树算法(cont.)

- 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;

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

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++ )
     {     min = LOWCOST[i];
           k = i;
           for(  j = 2;   j <= n;  j++ ) //3.1在LOWCOST中选最短边,记CLOSSET中对应的顶点序号k
                if ( LOWCOST[j] < min )
                {     min = LOWCOST[j] ;     k=j;  }
           cout << "("  << k << "," << CLOSSET[k] <<  ")"  <<  end1;//3.2输出最小生成树的边信息
           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(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)，其初态为U＝V，TE＝{ }，即把每个顶点看成一个连通分量。
    - 然后，按照边的权值由小到大的顺序，依次考察G的边集E中的各条边。
    - 若被考察的边连接的是两个不同连通分量，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；
    - 若被考察的边连接的是同一个连通分量，则舍去此边，以免造成回路，
    - 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*、*v*位于T的两个不同连通分量，则

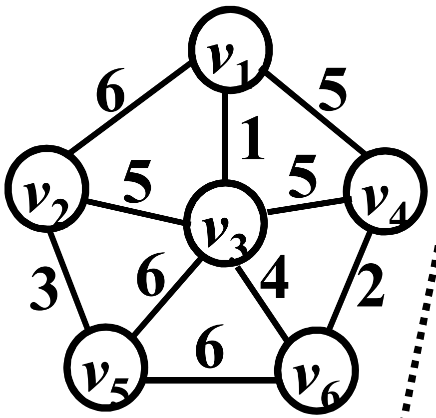
      2.2.1 将边(*u*，*v*)并入TE；

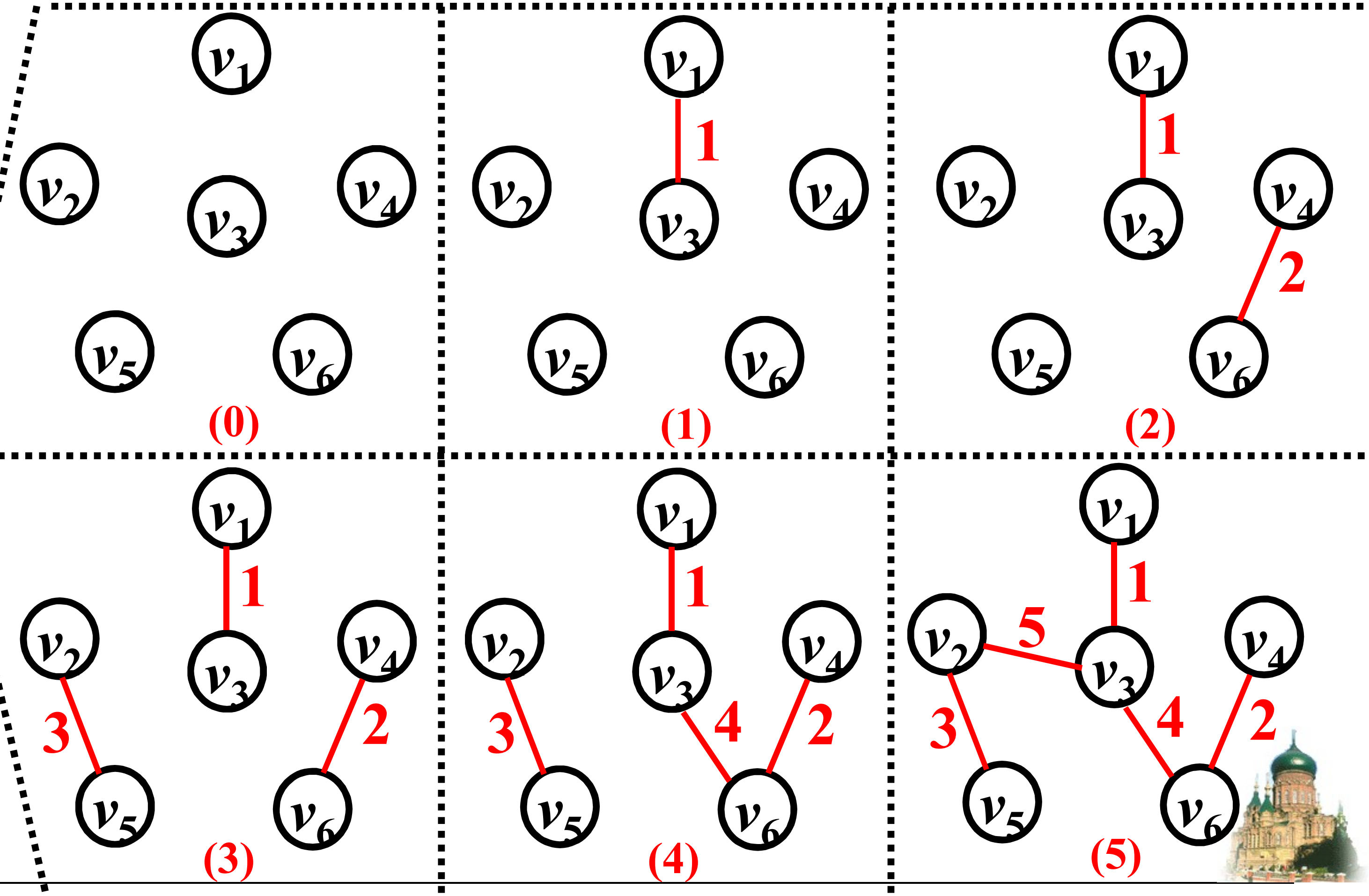
      2.2.2 将这两个连通分量合为一个；

   2.3 在E中标记边(*u*，*v*)，使得(*u*，*v*)不参加后续最短边的选取

4.4 最小生成树算法(cont.)

(v_1,v_3)	(v_4,v_6)	(v_2,v_5)	(v_3,v_6)	(v_1,v_4)	(v_3,v_4)	(v_2,v_3)	(v_1,v_2)	(v_3,v_5)	(v_5,v_6)
1	2	3	4	5	5	5	6	6	6





时间复杂度：

O (|E|*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
       parents[i]=0;
     for(i=0;i<arcnum;i++)  {
        bnf=Find(edges[i].begin,parents);
        edf=Find(edges[i].end,parents);
        if(bnf!=edf) {
           parents[bnf]=edf;
           cout<<'('<<edges[i].begin]<<', ' ;
           cout<<edges[i].end]<<','<<edges[i].cost<<') ';
           cout<<endl;
        }
     }
}   /* 时间复杂度：O (|E|*log|E|)
```

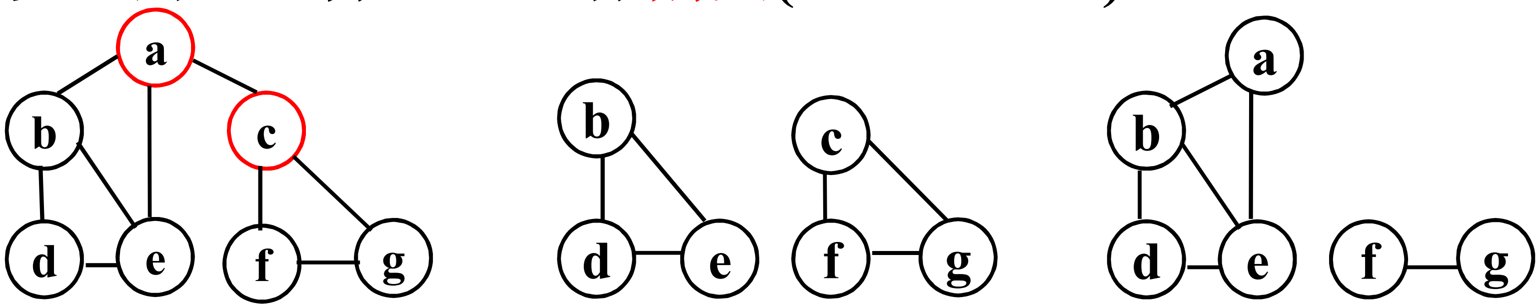
	cost	begin	end
0	–	–	–
1			
2			

edges

4.5 双(重)连通性算法

相关概念：设G=(V, E)是一个连通图，

- 一个顶点*a*称为连通无向图的关节点(*Articulation Point*)，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*) 。



- 没有关节点的连通图称为双连通图（*Biconnected Graph*）。
- 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。
- 一个连通图*G*如果不是双连通图，那么它可以包括几个双连通分量（*Biconnected Component*）

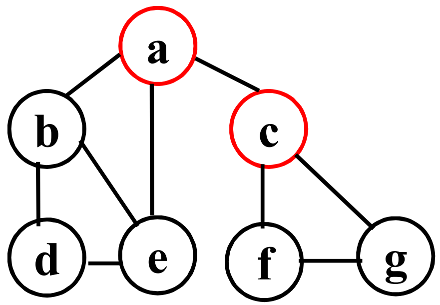
4.5 双连通性算法(cont.)

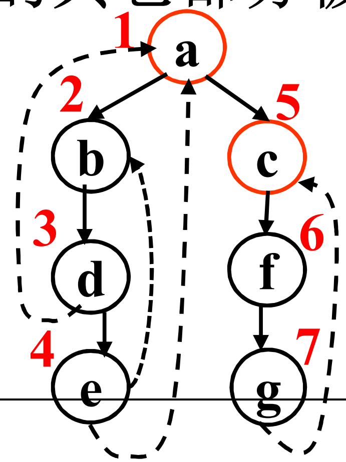
- 一个连通图*G*如果不是双连通图，那么它可以包括几个双连通分量（***Biconnected Component***）。
- 称连通图**G=(V, E)**的边e_1 和 e_2 是等价的，若 $e_1=e_2$ 或者有一条环路包含 e_1又包含 e_2 。
- 设 V_i是等价边集 E_i 中各边所连接的顶点集（**1≤i≤k**），每个图 $G_i = (V_i, E_i)$ 叫做 **G** 的一个双连通分量。
- 双连通图的性质：
  - 性质**1**　G_i 是双连通的（**1≤i≤k**）
  - 性质**2**　对所有的 **i≠j**, $V_i \cap V_j$ 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≠j)**

4.5 双连通性算法(cont.)

关节点性质

- 对图进行一次先深搜索便可求出所有的关节点，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:
  - 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（第一类关节点）。因为图中不存在连接不同子树中顶点的边，因此，若删去根顶点，生成树变成生成森林。
  - 若生成树中非叶顶点*v*，其某株子树的根和子树中的其它结点均没有指向*v* 的祖先的回退边，则*v* 是关节点（第二类关节点）。 因为删去*v*，则其子树和图的其它部分被分割开来





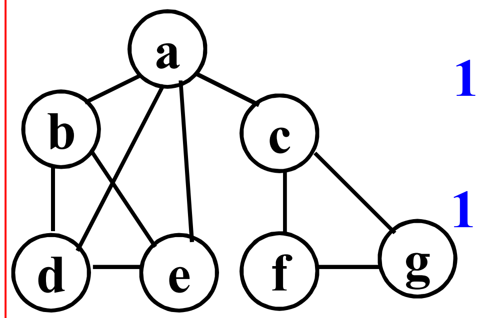
4.5 双连通性算法(cont.)

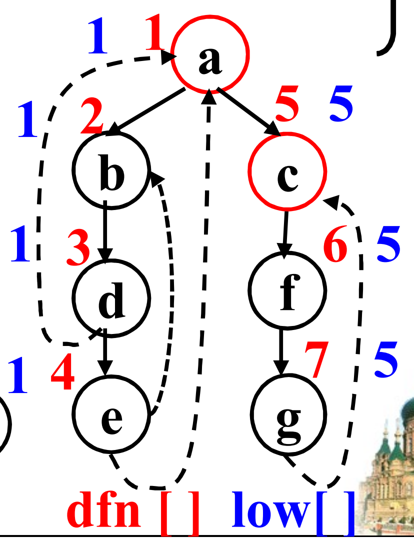
low[v]编号—顶点的最小深度优先数编号

- low[v]-是v及其子树结点能到达的最小 (dfn[])编号结点的编号
- 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：

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

若某个顶点v,存在孩子结点y,且low[y]≥dfn[v],则v必为关节点. 因为这表明，y及其子孙均无指向v的祖先的回退边。





4.5 双连通性算法(cont.)

R.Tarjan算法—求关节点算法算步骤

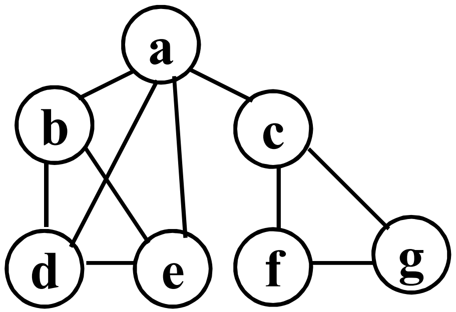
- 1. 计算先深编号：对图进行先深搜索，计算每个结点v的先深编号dfn[v]，形成先深生成树S=(V,T)。
- 2. 计算low[v]：在先深生成树上按后根遍历顺序进行计算每个顶点v的 low[v] ，low[v]取下述三个结点中的最小者：
  - (1) dfn[v]；
  - (2) dfn[w]，凡是有回退边(v,w)的任何结点w；
  - (3) low[y]，对v的任何儿子y。
- 3. 求关节点：
  - 3.1 树根是关节点，当且仅当它有两个或两个以上的儿子(第一类关节点)；
  - 3.2 非树根结点v是关节点当且仅当v有某个儿子y，使 low[y]≥dfn[v] (第二类关节点)。

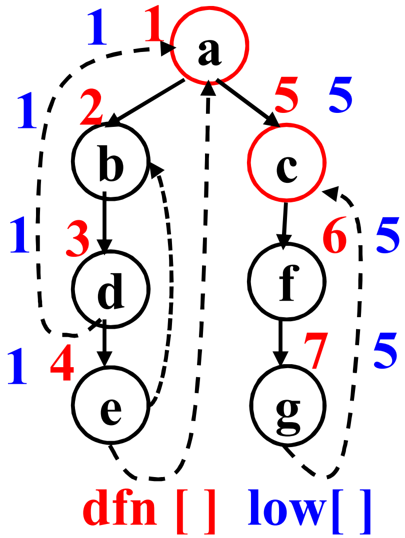
R.Tarjan算法是先深搜索，因此，时间复杂度为O(n+e)或O(n^2)

4.5 双连通性算法(cont.)

示例：按后根遍历顺序计算low[v]编号和求关节

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





序	结点	dnf[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

- 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] ≥ dnf[c]，所以c是关节点。

4.5 双连通性算法(cont.)

求关节点的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; /*全局变量count 用于对访问计数*/
     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*/
          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);
          }//while
     }//if
} //FindArticul
```

4.5 双连通性算法(cont.)

```
void DFSArticul(int v0)
/*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 */
{    dnf[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(dnf[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

- 最受欢迎的牛！
  - 有一群牛，总数为N(N<=10000)。牛之间的粉丝关系为，如A是C粉丝，C是B粉丝等等，设这种粉丝关系是可以传递的，若A是C的粉丝，那么A同时也是C的粉丝的粉丝。(粉丝关系数e<=50000)。如果所有的牛都是一头牛的粉丝，那么它将是最受欢迎的牛。
  - 是否有最受欢迎的牛？
  - 有多少牛是最受欢迎的？
  - 强连通性的判定和强连通分量求解算法的应用

4.6 强连通性(cont.)

有向图强连通性的概念和性质

- 称有向图G =(V, E) 顶点v ,w∈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 $E_i(1\le i\le r)$ 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。∀v∈G，与v可相互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

- 强连通图的性质定理
  - 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。

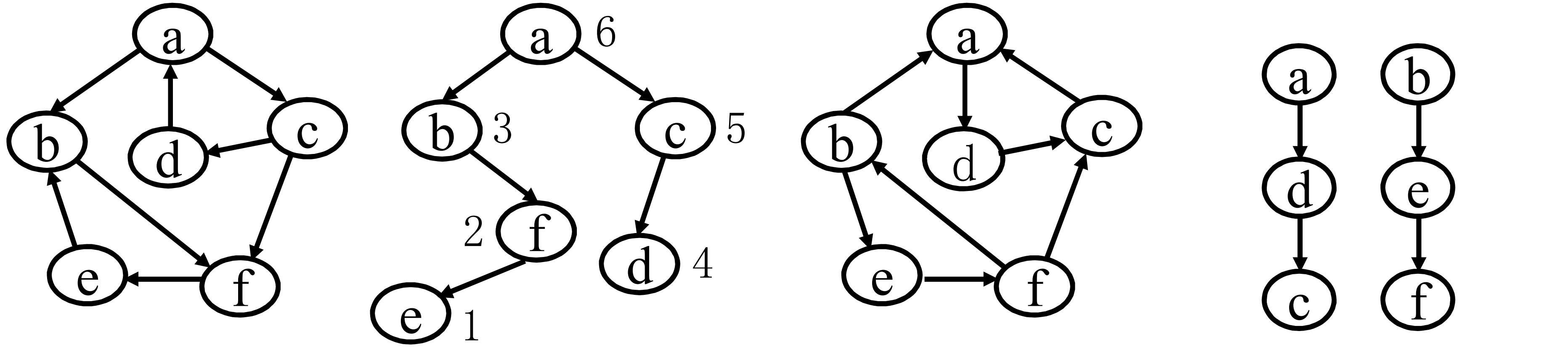
4.6 强连通性(cont.)

求有向图强连通分支的算法—Korasaju算法

- 输入：有向图G（如，十字链表表示）
- 输出：有向图G的强连通分量（森林的孩子-兄弟表示）
- 算法步骤：
  - 1.深度优先遍历G（起点如何选择无所谓），并计算出每个顶点u的结束时间dfn[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，选择遍历的起点时，按照顶点的结束时间从大到小进行。遍历的过程中，一边遍历，一边给顶点做分类标记，每找到一个新的起点，分类标记值就加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的一棵树，也即一个强连通分量

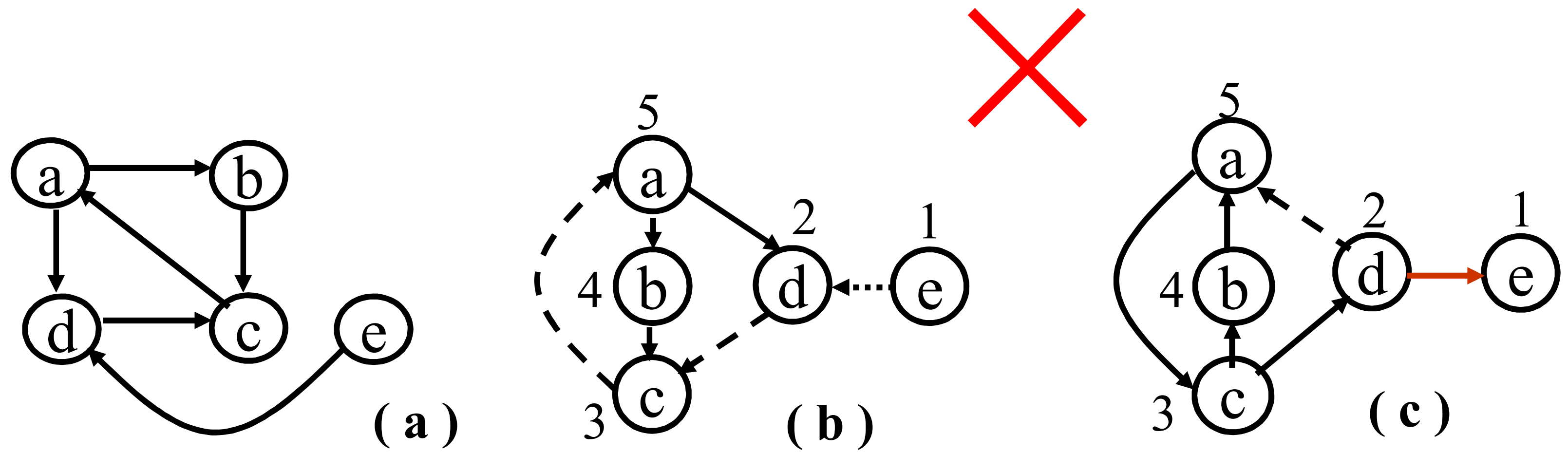
4.6 强连通性(cont.)

求有向图强连通分支的示例



(a) 有向图G　(b) 执行步骤(1)　(c) 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



4.6 强连通性(cont.)

- 求有向图强连通分支的Korasaju算法实现

```
/* 按弧的正向搜索，起点如何选择无所谓 */
int in_order[MAX_VEX] ;
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 ;
 }
```

4.6 强连通性(cont.)

- 求有向图强连通分支的Korasaju算法实现

```
/*  对图G按弧的逆向进行搜索  */
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) ;
}
```

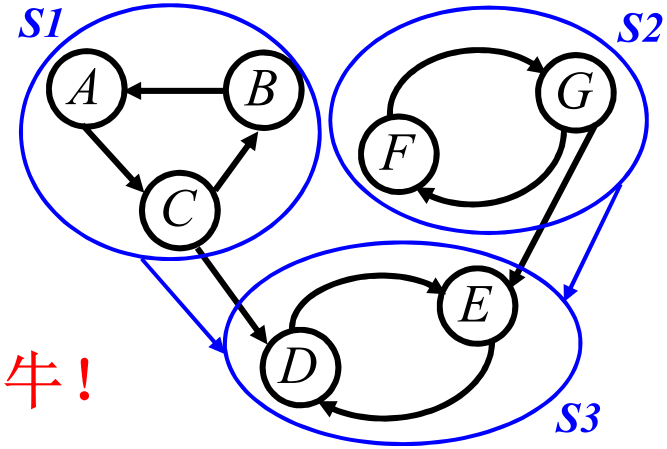
4.6 强连通性(cont.)

- 求有向图强连通分支的Korasaju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) ;
         }
      }
}
```

4.6 强连通性(cont.)

- **Korasaju算法复杂度分析**
  - 深度优先搜索的复杂度：Θ(|V|+|E|)
  - 需两次深搜，总的复杂度为: Θ (|V|+|E|)，非常好的算法!
- 其他求有向图强连通分量的算法
  - **Tarjan**算法(与关节点算法类似)
  - **Gabow** 算法
    - 都只需一次深度优先搜索



- 求强连通分量算法的应用—最受欢迎的牛！
  - 有一群牛，总数为N(N<=10000)。牛之间的粉丝关系为，如A是C粉丝，C是B粉丝等等，设这种粉丝关系是可以传递的，若A是C的粉丝，那么A同时也是C的粉丝的粉丝。(粉丝关系数e<=50000)。如果所有的牛都是一头牛的粉丝，那么它将是最受欢迎的牛。问有多少牛是“最受欢迎的”。

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。
- 如果图是一个带权图，则路径长度为路径上各边的权值的总和，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上的权值总和达到最小？
- 集成电路设计、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）与源点 v ∈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。

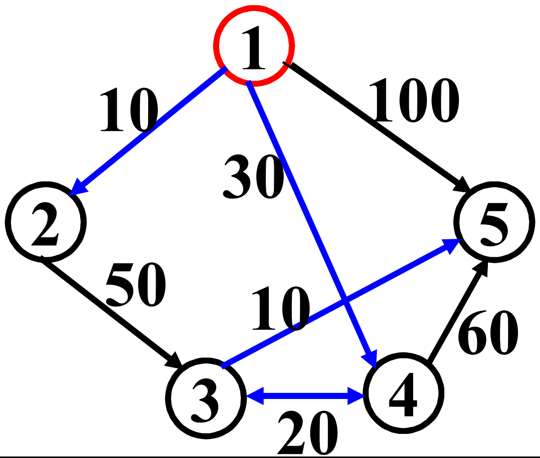
4.7 最短路径算法(cont.)

- Edsger Wybe Dijkstra(1930年5月11日~2002年8月6日)荷兰计算机科学家，毕业就职于荷兰Leiden大学，早年钻研物理及数学，而后转为计算学。1972年获得图灵奖，1974年获得AFIPS Harry Goode Memorial Award、1989年ACM SIGCSE计算机科学教育教学杰出贡献奖、以及2002年ACM PODC最具影响力论文奖。
- 与D. E. Knuth并称为这个时代最伟大的计算机科学家。
  - 提出“goto有害论”;结构程序设计之父
  - 提出信号量和PV原语;
  - 解决了有趣的“哲学家聚餐”问题;
  - 最短路径算法（SPF）的创造者;
  - 第一个Algol 60编译器的设计者和实现者
  - THE操作系统的设计者和开发者;
  - 提出银行家算法，解决了操作系统中资源分配问题

4.7 最短路径算法(cont.)

Dijkstra算法的基本思想

- Dijkstra提出按路径长度的递增次序， 逐步产生最短路径的贪心算法—Dijkstra算法。
- 亦称SPF算法(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。

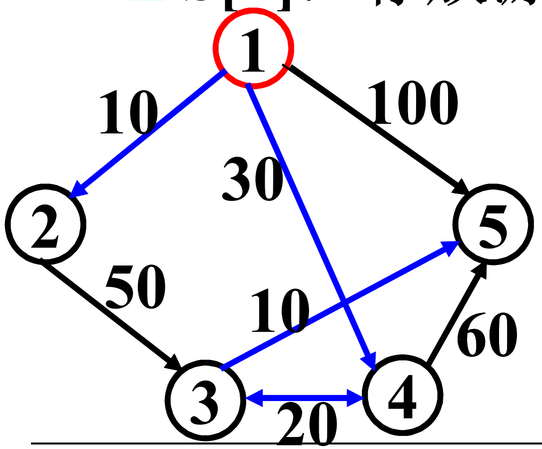


源点S	中间结点	终点	路径长度
1		2	1 0
1		4	3 0
1	4	3	5 0
1	4 3	5	6 0

4.7 最短路径算法(cont.)

Dijkstra算法的数据结构

- 假设带权有向图G=(V, E)，其中V={ 1， 2， …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



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

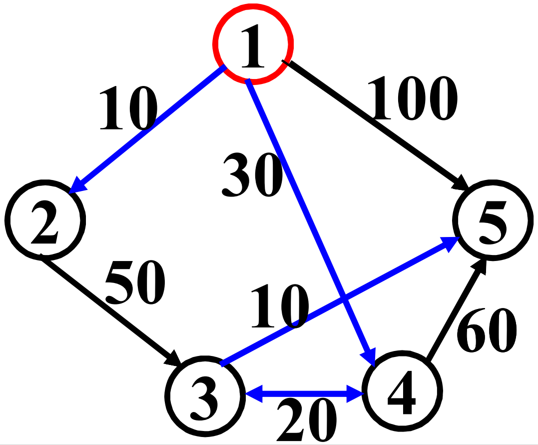
D		10	∞	30	100
P		1	1	1	1
S	T	F	F	F	F

4.7 最短路径算法(cont.)

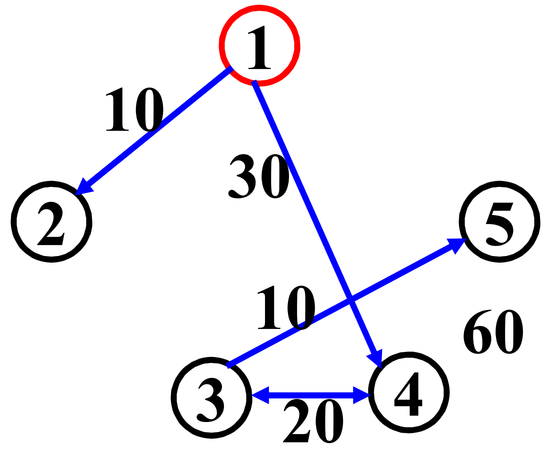
- **Dijkstra算法实现步骤：**
  - **1.** 将V分为两个集合S（最短路径已经确定的顶点集合）和V-S（最短路径尚未确定的顶点集合。初始时，S={ 1 }，D[i]=C[1][i]（i=2,3,…n），P[i]=1(源点，i≠1)。
  - **2.** 从S之外即V-S中选取一个顶点w，使D[w]最小（即选这样的w，D[w]=min{ D[i]| i∈V-S }），于是从源点到达w只通过S中的顶点，且是一条最短路径（选定路径），并把w加入集合S。
  - **3.** 调整D中记录的从源点到V-S中每个顶点的最短距离，即从原来的D[v]和D[w]＋C[w][v]中选择最小值作为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就记录了从源到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



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



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][j] ;  S[i]=FALSE ;}
   S [1]= TRUE ;
   for( i=1; i<n; i++)
   {  w=MinCost ( D, S ) ;
      S[w]=TRUE ;
      for ( v=2 ; v<= n ; n++ )
         if ( S[v]!=TRUE )
         {   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 ;
}
```

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* 最短路径
    - 只需每边取反?
  - 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

4.7 最短路径算法(cont.)

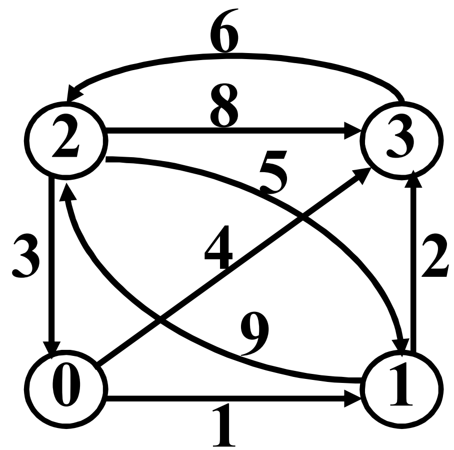
任意两个顶点之间的最短路径(Floyd- Warshall算法)

- 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算法的基本想法：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.)

- 示例:
  - <2,1>5 <2,0><0,1> 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:

    <2,0,3>7 <2,3>8 a[2][3]=a[2][0]+a[0][3]
  - <2,3>: <2,0><0,3>: <2,0><0,1><1,3>=<2,0,1,3> 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



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

4.7 最短路径算法(cont.)

Floyd算法的基本思想：

- 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 $(v_i ,\ldots, v_1)$和 $(v_1,\ldots, v_j)$ 分别是当前找到的中间顶点的序号不大于0的最短路径,那么 $(v_i,\ldots, v_1 ,\ldots, v_j)$就是有可能是从$v_i$到$v_j$的中间顶点的序号不大于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，从中选出中间顶点的序号不大于1的最短路径，再增加一个顶点v_2, 继续进行试探。
- 一般情况下，若 $(v_i ,\ldots, v_k)$ 和 $(v_k, \ldots, v_j)$ 分别是从 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 $(v_i ,\ldots, v_k ,\ldots, v_j)$ 和已经得到的从 v_i到 v_j且中间顶点序号不大于k-1的最短路径相比较，其长度较短者便是从v_i到 v_j 的中间顶点的序号不大于 ***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：

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

  - 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算法的实现

```
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][k] + D[k][j] < D[i][j] ) {
                         D[i][j] = D[i][k] + D[k][j] ;
                         P[i][j] = k;}
}
/* 时间复杂度：O(n^3) */
```

Warshall算法

求有向图邻接矩阵C的传递闭包D

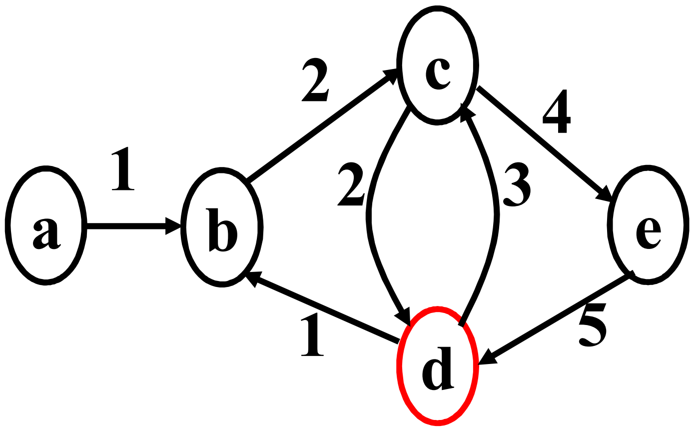
D[i][j]=D[i][j]∪(D[i][k]∩D[k][j]) ;

可以判定有向图任意两点间是否存在有向路。

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] | i ∈V }称作顶点 k 的偏心度(其他顶点到顶点k的距离中的最大者)。
- 图G的中心点：
  - 称具有最小偏心度的顶点为图G的中心点。



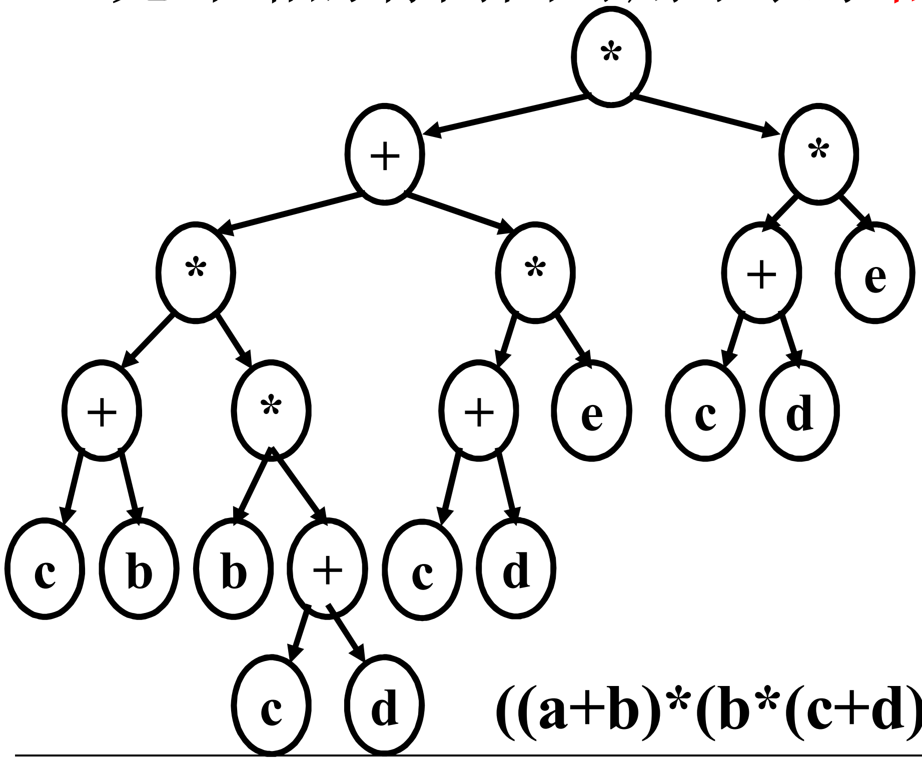
最短路径矩阵D

$$
\begin{array}{c}
\\ a \\ b \\ c \\ d \\ e
\end{array}
\begin{array}{c}
\begin{array}{ccccc} a & b & c & d & e \end{array} \\
\begin{pmatrix}
0 & 1 & 3 & 5 & 7 \\
\infty & 0 & 2 & 4 & 6 \\
\infty & 3 & 0 & 2 & 4 \\
\infty & 1 & 3 & 0 & 7 \\
\infty & 6 & 8 & 5 & 0
\end{pmatrix}
\end{array}
$$

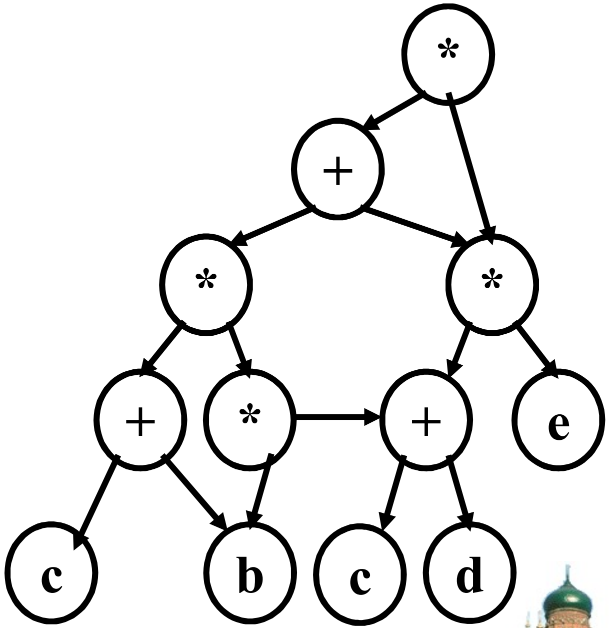
顶点	偏心度
a	∞
b	6
c	8
d	5
e	7

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



公共子式
b
(c+d)
(c+d)*e



((a+b)*(b*(c+d))+(c+d)*e)*((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
  - 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$(x, z) \in R$

  则称R是集合X上的偏序关系。
- 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
- 直观上，偏序指集合上只有部分元素之间可比较，而全序是指全体元素均可比较。

4.8 拓扑排序算法(cont.)

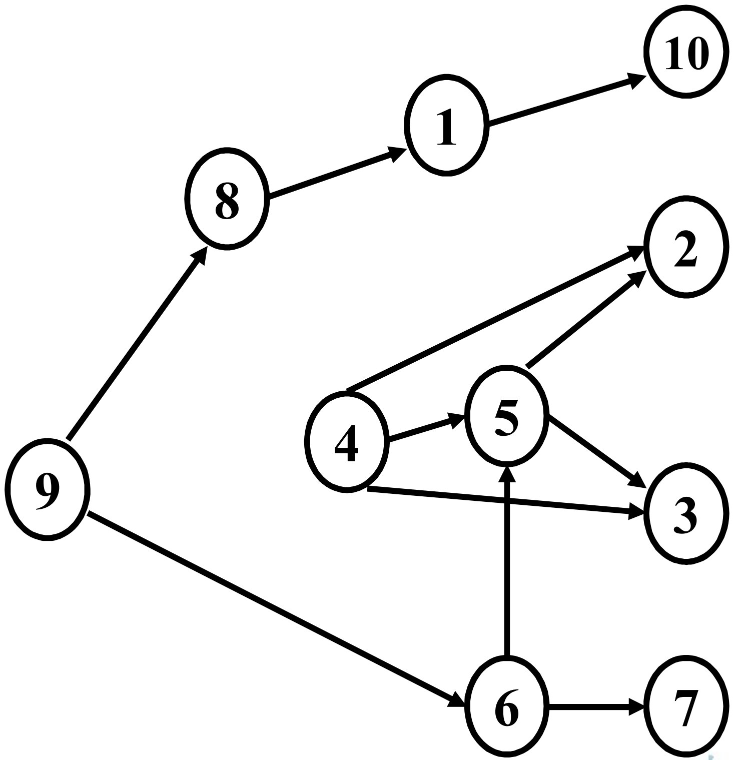
- 如何用无环路的有向图表示偏序关系？
  - 设R是有穷集合X上的偏序关系，对X中每个 v，用一个以 v 为标号的顶点表示，由此构成顶点集V；对任意$(u, v) \in$ R, $(u \neq v)$，由对应两个顶点建立一条有向边，由此构成边集E，则G =(V , E)是无环路有向图。
- 拓扑排序：是由某个集合上的一个偏序得到该集合上的一个全序的过程。所得到的线性序列称为拓扑序列。
- AOV网：在一个表示工程的有向图中，用顶点表示活动，用弧表示活动之间的优先关系，称这样的有向图为顶点表示活动的网，简称AOV网。
  - AOV网中的弧表示活动之间存在的某种制约关系。
  - AOV网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

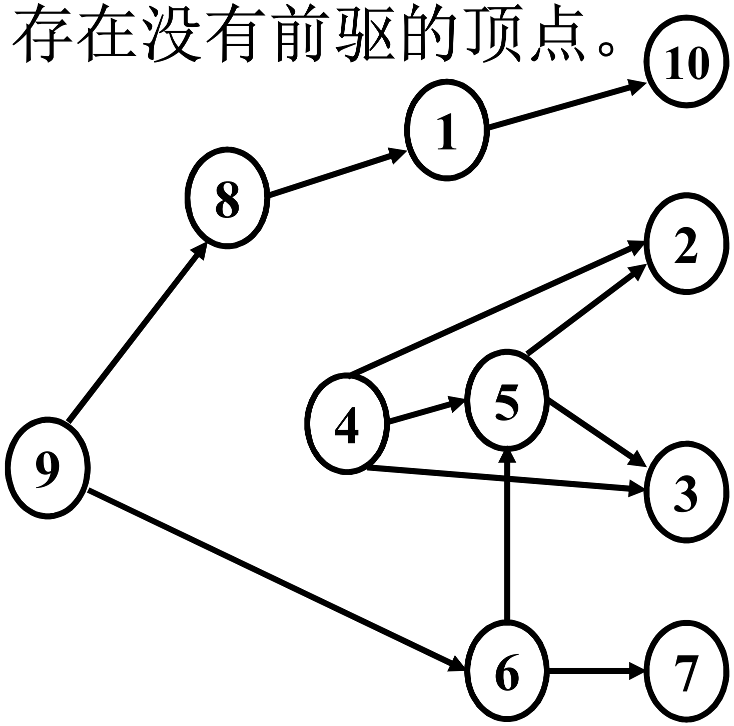
- 课程及课程间的先修关系是偏序关系，可以用AOV网表示

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，并且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，入队
}
4. 若输出结点个数小于n，则输出有环路
  ；否则拓扑排序正常结束。
```

☞图中还有未输出的顶点，但已跳出循环处理。说明图中还剩下一些顶点，它们都有直接前驱。这时网络中必存在有向环；或

☞全部顶点均已输出，拓扑有序序列形成，拓扑排序完成。

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 ;
    MAKENUlLL( 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.)

- 关于广度优先拓扑排序的几点说明
  - 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（引入数组**indegree[ ]**或在顶点表中增加一个属性域**indegree**）。
  - 也可以采用栈数据结构进行广度优先拓扑排序。
  - 亦可采用无后继顶点优先的拓扑排序算法
  - 也可以利用**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
  - 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的顶点栈；
3.while（栈不空）{
        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，入栈
}
4.若输出结点个数小于n，则输出有环路
  ；否则拓扑排序正常结束。
```

4.8 拓扑排序算法(cont.)

- 利用栈结构进行拓扑排序

```
void  Topologicalsort( AdjGraph  G )
{   MAKENUlLL( 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

```
void topodfs ( v )
{   PUSH( v ,S ) ;
    mark[v]=TRUE;
    for ( L[v] 中的每一个顶点w)
        if ( mark[w] = FALSE )
            topodfs ( w ) ;
    printf ( top( S ) ) ;
    POP ( S ) ;
}
```

思想：借助栈，在DFS中，把第一次遇到的顶点入栈，到达某一顶点递归返回时，从栈中弹出顶点并输出。

```
void dfs-topo ( GRAPH L )
{ MAKENULL( S );
   for( u=1;u<=n;u++)
       mark[u]=FALSE;
   for( u=1;u<=n;u++)
       if ( !mark[u] )
           topodfs( u ) ;
}
```

4.9 关键路径算法

案例场景

- 某软件公司承接一家企业的信息系统集成业务。经过公司董事会的讨论，决定任命你作为该系统集成项目的项目经理，你接到任务后，应该制定进度表，这样项目才可以依照进度表继续下去。
- 在与项目团队成员探讨后，假设已经确认了11项基本活动。所有这些活动的名称、完成每项活动所需的时间，以及与其他活动之间的约束关系如右侧表：

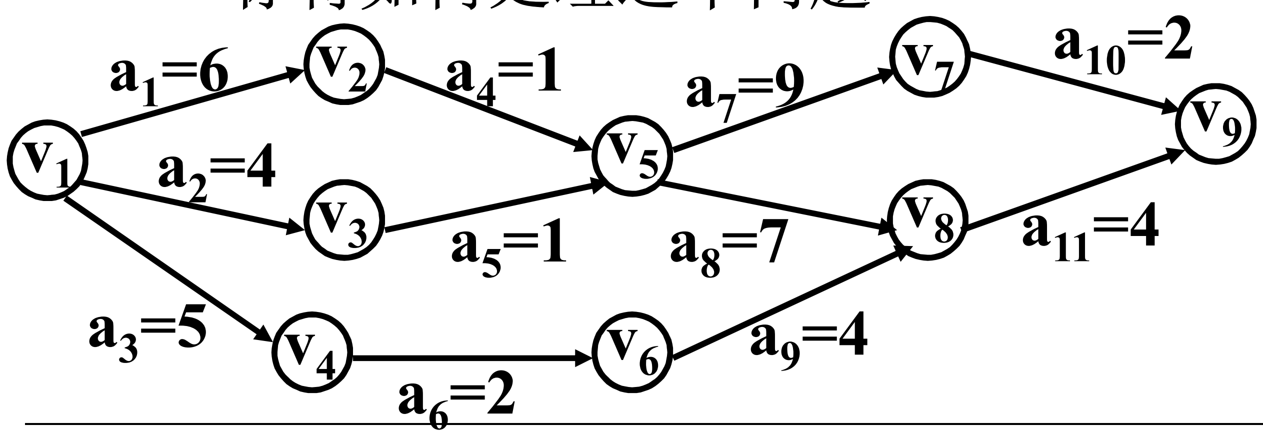
活动名称	必需的时间(天)	前置任务
a_1	6	
a_2	4	
a_3	5	
a_4	1	a_1
a_5	1	a_2
a_6	2	a_3
a_7	9	a_4,a_5
a_8	7	a_4,a_5
a_9	4	a_6
a_{10}	2	a_7
a_{11}	4	a_8,a_9

4.9 关键路径算法(cont.)

案例场景

- 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？
  - 例如，若任务a_5推迟了3天，对项目进度有何影响？作为项目经理，你将如何处理这个问题？

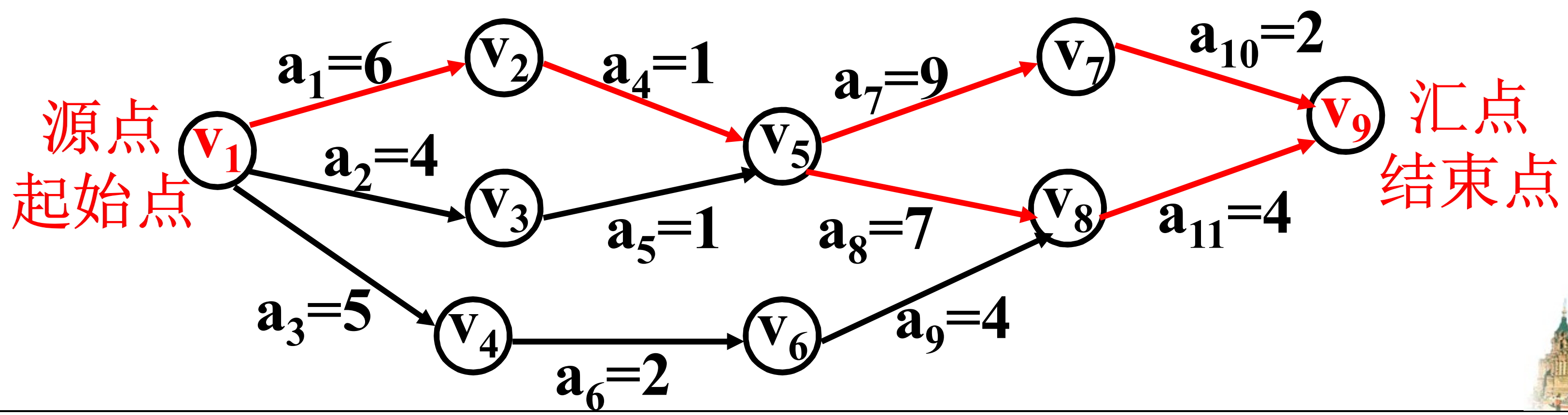
活动名称	必需的时间(天)	前置任务
a_1	6	
a_2	4	
a_3	5	
a_4	1	a_1
a_5	1	a_2
a_6	2	a_3
a_7	9	a_4,a_5
a_8	7	a_4,a_5
a_9	4	a_6
a_{10}	2	a_7
a_{11}	4	a_8,a_9



4.9 关键路径算法(cont.)

AOE网（Activity On Edge Network）

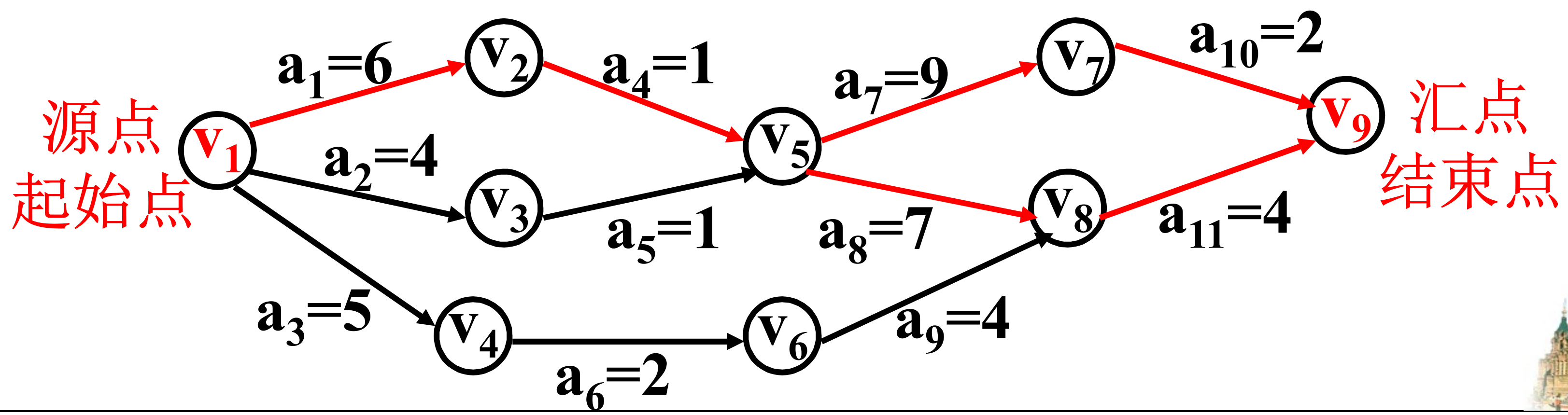
- 在带权的有向图中，用顶点表示事件，用边表示活动，边上权表示活动的开销（如持续时间），则称此有向图为边表示活动的网络，简称**AOE**网。
- 下图是有**11**项 活动，**9**个事件的**AOE**网，每个事件表示在它之前的活动已经完成，在它之后的活动可以开始。



4.9 关键路径算法(cont.)

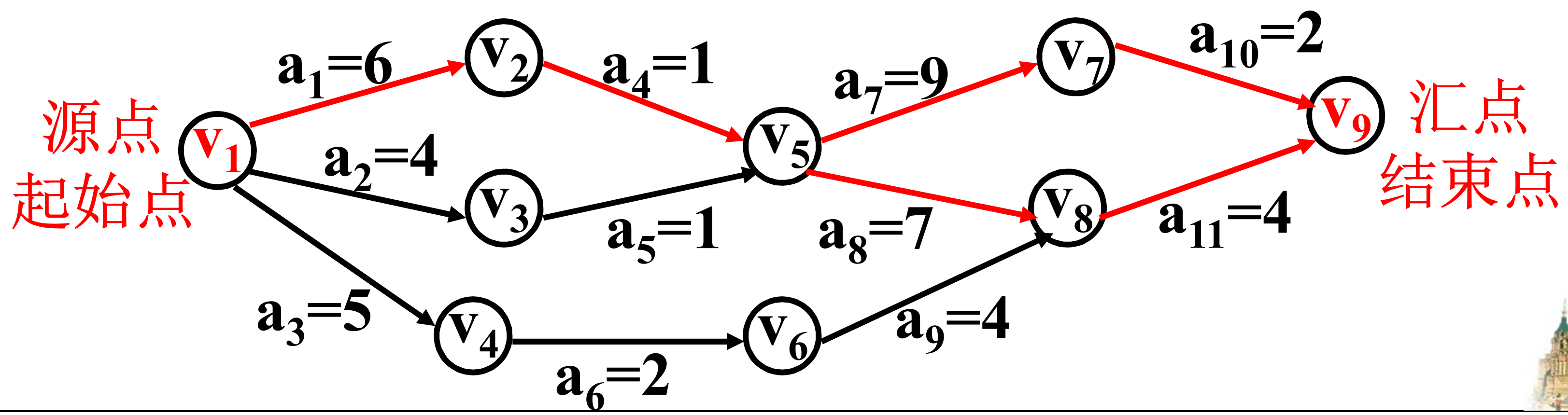
AOE网的性质

- 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网应该是无环的，并且存在唯一的入度为0的开始顶点（源点）和唯一的出度为0的结束点（汇点）。



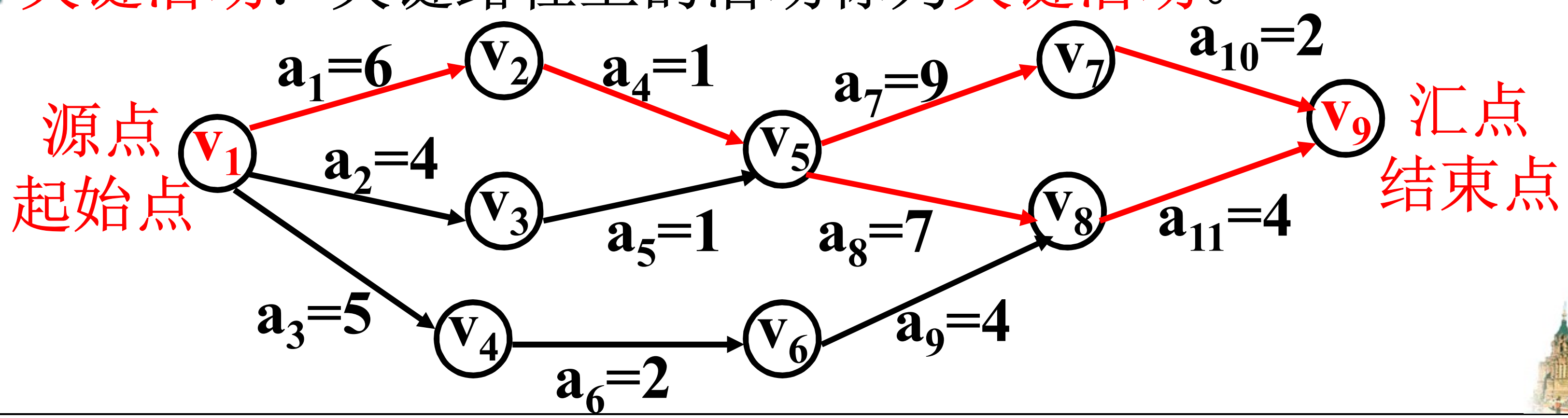
4.9 关键路径算法(cont.)

- **AOE网研究的主要问题:**
  - 如果用AOE 网表示一项工程，那么仅仅考虑各个子工程之间的优先关系还不够，更多地是关心整个工程完成的最短时间是多少，哪些活动的延迟将影响整个工程进度，而加速这些活动能否提高整个工程的效率，因此AOE网有待研究的问题是:
    - (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?



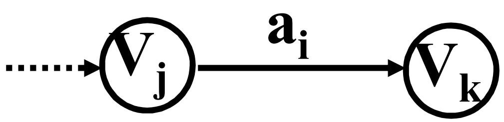
4.9 关键路径算法(cont.)

- 路径长度、关键路径、关键活动：
  - 路径长度：是指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，由于有些活动可以并行，所以完成工程的最短时间是从源点到汇点的最大路径长度。因此，把从源点到汇点具有最大长度的路径称为关键路径。
  - 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.)

- 关键路径和关键活动性质分析------与计算关键活动有关的量
  - 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< V_j , V_k>上, 则E(i)也是从源点V_1到顶点V_j 的最长路径长度。这是因为事件V_j发生表明以V_j为起点的所有活动a_i可以立即开始。因此,
      - E(i) = VE(j) …………..(1)
  - ③事件V_k的最迟发生时间VL(k)
    - 是在保证汇点V_n在VE(n)时刻发生的前提下，事件V_k的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)应该等于汇点的最早发生时间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.)

- 关键路径和关键活动性质分析--与计算关键活动有关的量
  - ④ 活动a_i 的最迟开始时间 L(i)

$$\cdots\cdots \to V_j \xrightarrow{a_i} V_k$$

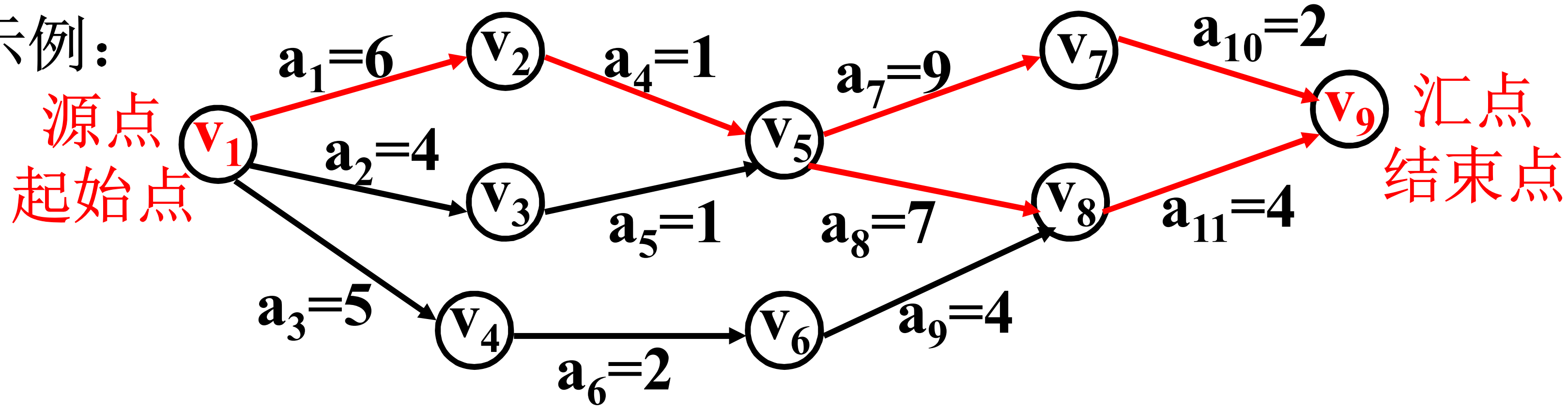
- 是指在不会引起工期延误的前提下，活动a_i允许的最迟开始时间。
- 因为事件V_k发生表明以V_k为终点的入边所表示的所有活动均已完成，所以事件V_k的最迟发生时间VL(k)也是所有以V_k为终点的入边$< V_j , V_k>$所表示的活动a_i可以最迟完成时间。
- 显然，为不推迟工期，活动a_i的最迟开始时间L(i)应该是<u>a_i的最迟完成时间VL(k)减去a_i的持续时间</u>，即
  - L(i) = VL(k) - ACT[j][k] ....(2)
  - 其中, ACT[j][k]是活动a_i的持续时间($< V_j , V_k>$上的权)

4.9 关键路径算法(cont.)

- 关键路径和关键活动性质分析------与计算关键活动有关的量
  - ⑤时间余量 L(i) - E(i)
    - L(i) - E(i)表示活动 a_i 的最早开始时间和最迟开始时间的时间余量。
  - 关键路径上的活动都满足：L(i) = E(i) ……………(3)
  - L(i) = E(i)表示活动是没有时间余量的关键活动。
  - 由上述分析知，为找出关键活动，需要求各个活动的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)可有公式 (1)和(2)计算得到。而VE(k) 和VL(k)可由拓扑排序算法得到。
- 关键路径和关键活动分析计算示例：

4.9 关键路径算法(cont.)

- 示例：



事件	VE[j]	VL[k]
V_1	0	0
V_2	6	6
V_3	4	6
V_4	5	8
V_5	7	7
V_6	7	10
V_7	16	16
V_8	14	14
V_9	18	18

事件的最早发生时间

事件的最迟发生时间

活动	E[i]	L[i]	L[i] - E[i]
a_1	0	0	0
a_2	0	2	2
a_3	0	3	3
a_4	6	6	0
a_5	4	6	2
a_6	5	8	3
a_7	7	7	0
a_8	7	7	0
a_9	7	10	3
a_{10}	16	16	0
a_{11}	14	14	0

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，按拓扑序列次序求出其余各顶点事件的最早发生时间：

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

  - 其中T是以顶点V_k为尾的所有边的头顶点的集合(2≤k≤n)
  - 若网中有回路，不能求出关键路径则算法中止；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的最晚发生时间：

$$VL(j) = \min_{k \in S}\{ VL(k)+ACT[j][k] \}$$

  - 其中S是以顶点V_j为头的所有边的尾顶点的集合(2≤j≤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。
- 为了简化算法，可以在求关键路径之前已经对各顶点实现拓扑排序，并按拓扑有序的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。
- 想使整个工程提前，要考虑各个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

ADT基本数据结构
- ADT定义 逻辑结构
  - 定义及相关术语
  - 逻辑结构及其特征
  - 基本操作（算法）→ 动态的操作

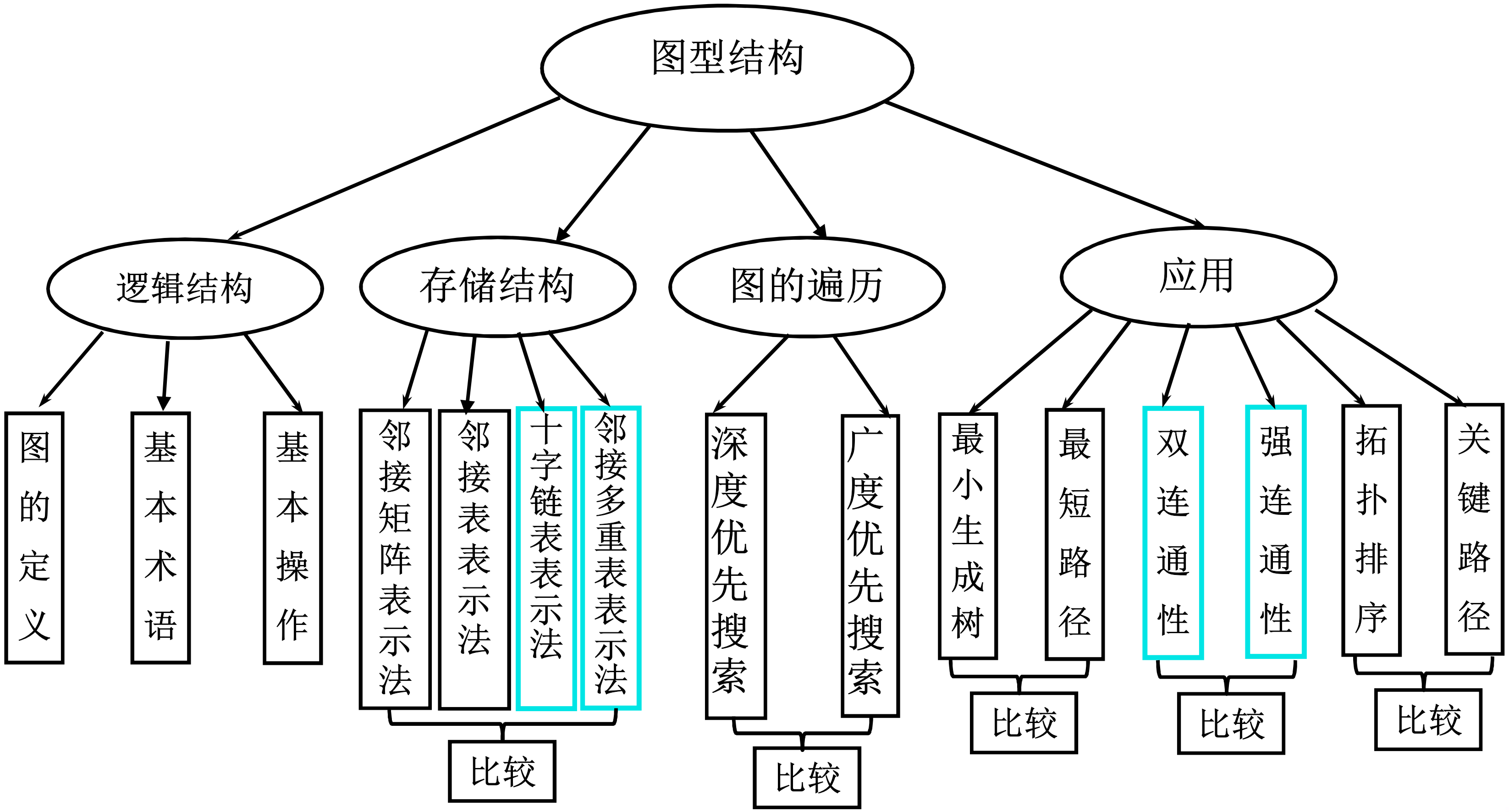
  （定义及相关术语、逻辑结构及其特征：静态的结构）
- ADT实现 存储结构
  - 存储结构（描述）
  - 存储结构特点
  - 存储结构的定义
  - 操作（算法）实现→动态的操作
  - 算法的性能

  （存储结构（描述）、存储结构特点、存储结构的定义：静态的结构）
- 应用: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等

图的遍历(搜索)算法是有关图问题的重要核心算法！

本章小结

知识点总结


图型结构
逻辑结构
存储结构
图的遍历
应用
图的定义
基本术语
基本操作
邻接矩阵表示法
邻接表表示法
十字链表表示法
邻接多重表表示法
比较
深度优先搜索
广度优先搜索
比较
最小生成树
最短路径
比较
双连通性
强连通性
比较
拓扑排序
关键路径
比较