数据结构与算法

主讲教师：刘峤

第7章 图

第7章 内容提要

7.1 图的定义

7.2 图的储存方式

7.3 图的遍历

7.4 图的连通性（自学）

7.5 最小生成树

7.6 最短路径

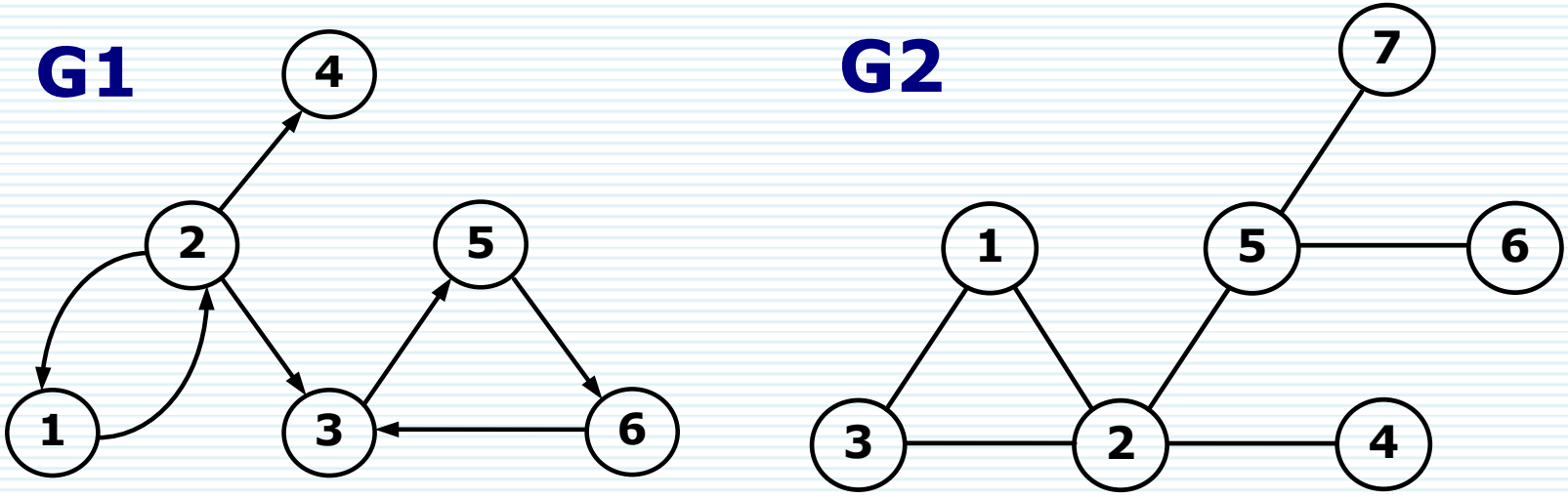
7.7 有向无环图的应用

本章导读

- 图是常用的一类重要的数据结构
  - 树可以看成是图的特例
    - 树中每个数据元素至多允许一个前驱
    - 只能反映数据元素之间一对多的关系
  - 图取消了该限制
    - 图中允许数据元素可以拥有多个前驱
    - 因此可以反映数据元素之间多对多的关系

1. 图的定义

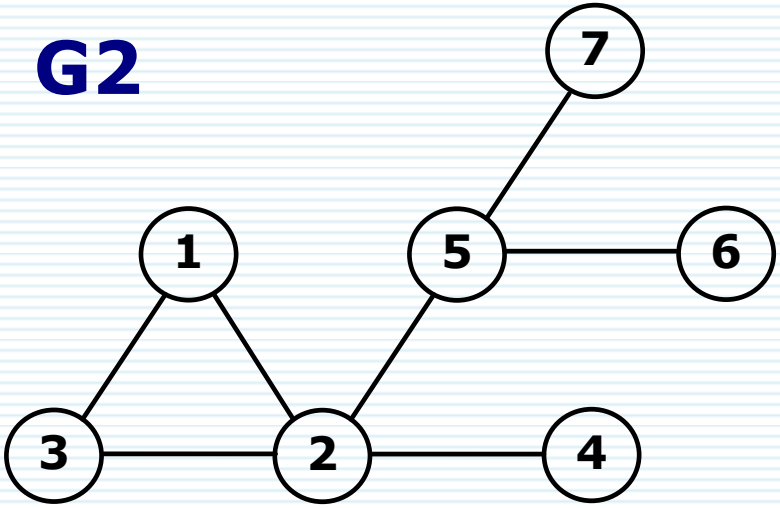
图（Graph）的定义



有向图：边是顶点的有序对　　**无向图：边是顶点的无序对**

- 图(G)是由两个集合V(G)和E(G)组成的，记为G=(V,E)
  - 其中：V(G)是顶点的非空有限集合
  - E(G)是边的有限集合（边是顶点的无序对或有序对）
    - 若E(G)为空集，则图G中只有顶点而没有边

图的分类



无向图：边是顶点的无序对

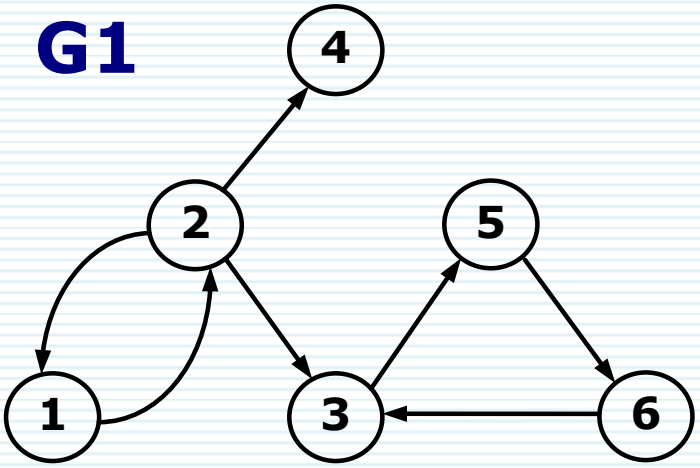
无向图G2中：

V(G2)={1, 2, 3, 4, 5, 6, 7}

E(G2)={ (1,2), (1,3), (2,3),
(2,4), (2,5), (5,6),
(5,7) }

- 无向图（undirected gragh）：由两个集合V(G)和E(G)组成
  - 其中：V(G)是顶点的非空有限集合
  - E(G)是无向边的有限集合，边是顶点的无序对
    - 记为：（v,w）或（w,v），并且（v,w）==(w,v)

图的分类



有向图：边是顶点的有序对

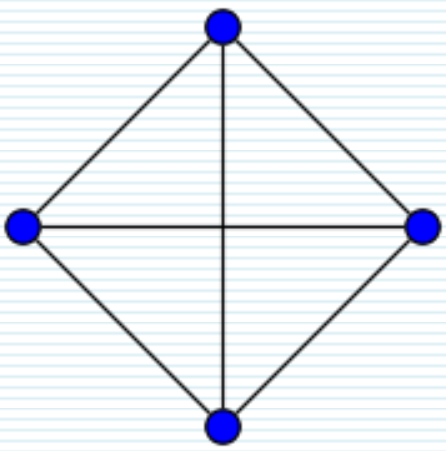
有向图G1中：

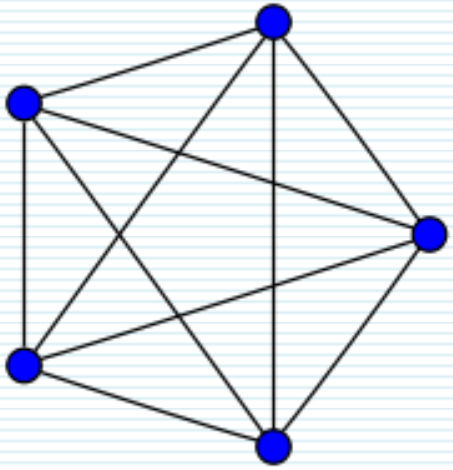
V(G1)={1, 2, 3, 4, 5, 6}

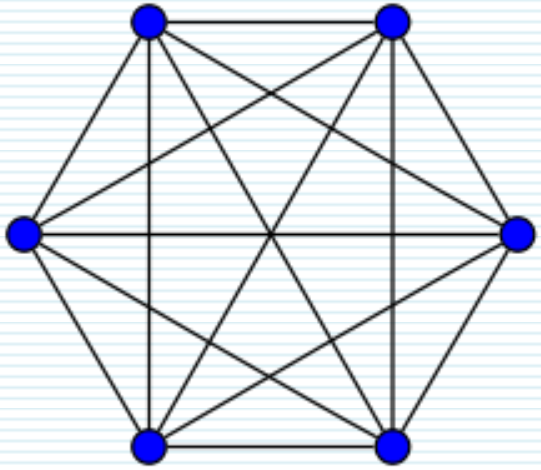
E(G1)={ <1,2>, <2,1>,
<2,3>, <2,4>, <3,5>,
<5,6>, <6,3> }

- 有向图（directed gragh）：由两个集合V(G)和E(G)组成
  - 其中：V(G)是顶点的非空有限集合
  - E(G)是有向边（也称弧）的有限集合，弧是顶点的有序对
    - 记为<v, w> （v和w是顶点，v为弧尾，w为弧头）

图的定义：完全图（Complete Graph）







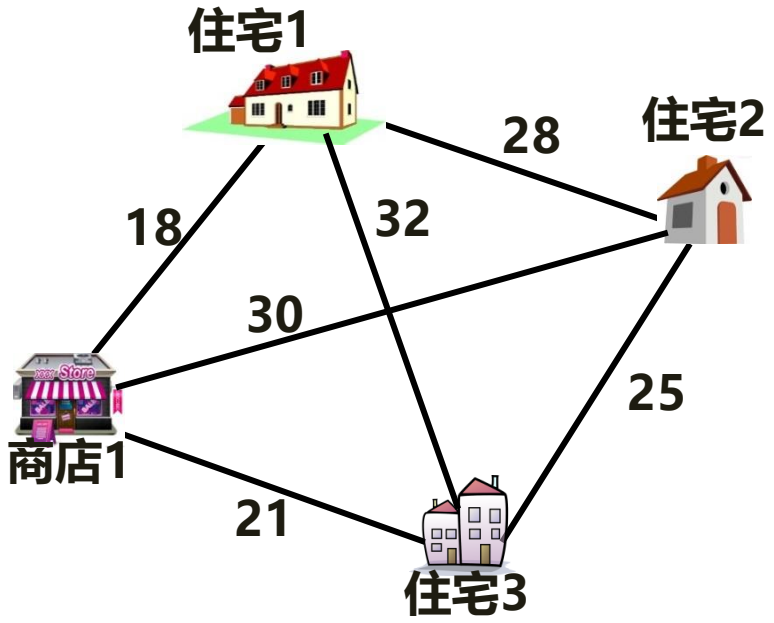
边数？ E(4)=6　　E(5)=10　　E(6)=15

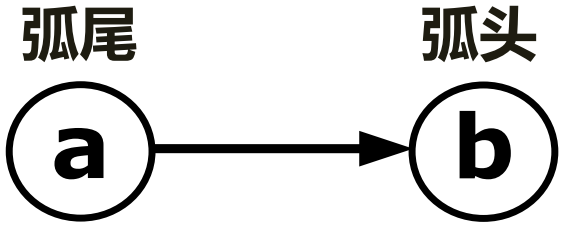
- 无向完全图（Undirected Complete Graph）
  - n个顶点的无向图最大边数：**#edges = n(n-1)/2**
- 有向完全图（Directed Complete Graph）
  - n个顶点的有向图最大边数：**#edges = n(n-1)**

图的定义

- 稀疏图（Spare Graph）
  - 图(G)中的边或者弧很少（edges < nlogn）
- 稠密图（Dense Graph）
  - 图(G)中的边或者弧很少（edges > nlogn）
- 网（Net）
  - 边或弧上带有权值的图
  - 权（weights）：与图的边或弧相关的数值称为权

图的定义：边和顶点的关系





顶点b是顶点a的邻接顶点

顶点a不是顶点b的邻接顶点

- 邻接关系
  - 无向图：若(v_i,v_j)是一条无向边，则称顶点v_i和v_j互为邻接点，并称(v_i, v_j)依附或关联于顶点v_i和v_j
  - 有向图：若$<v_i,v_j>$是一条有向边（弧），则称顶点v_i邻接于顶点v_j，并称顶点v_j是顶点v_i的邻接顶点

图的定义：顶点的度（Degree）

无向图：$$\text{edges} = \frac{1}{2}\sum_{i=1}^{n}\text{Degree}(v_i)$$

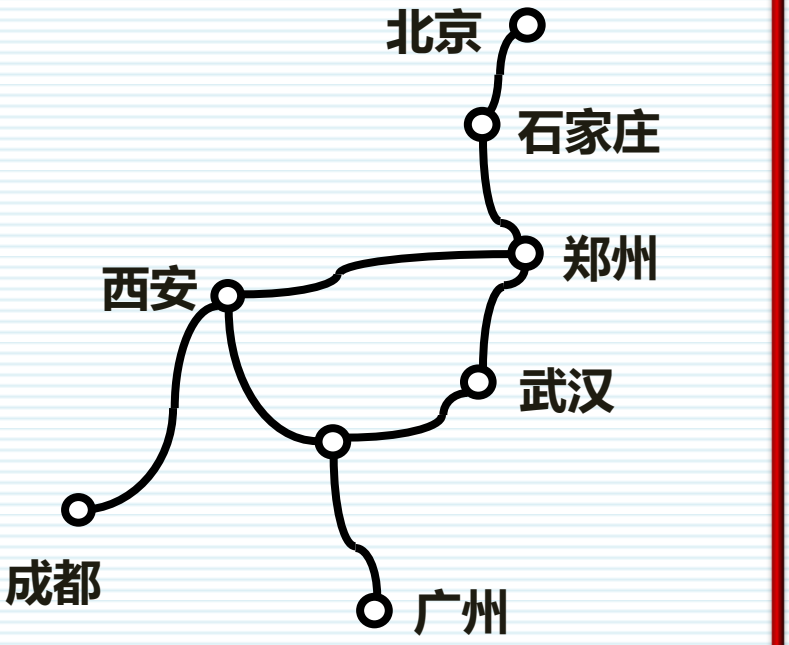
有向图：$$\text{edges} = \sum_{i=1}^{n}\text{in-degree}(v_i) = \sum_{i=1}^{n}\text{out-degree}(v_i)$$

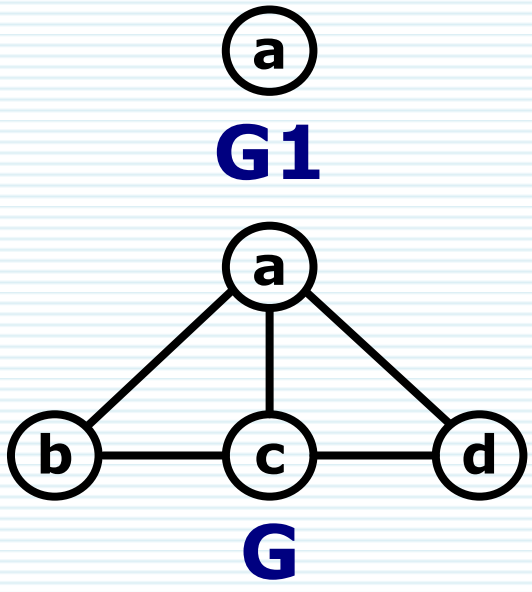
- 无向图中顶点v的度为：依附于顶点v的边数
  - 依附：边(a, b) 依附于顶点a和b
- 有向图中顶点v的度为：顶点v的入度与出度之和
  - 入度（in-degree）：以顶点V为弧头的弧数
  - 出度（out-degree）：以顶点V为弧尾的弧数
  - degree(v) = in-degree(v) + out-degree(v)

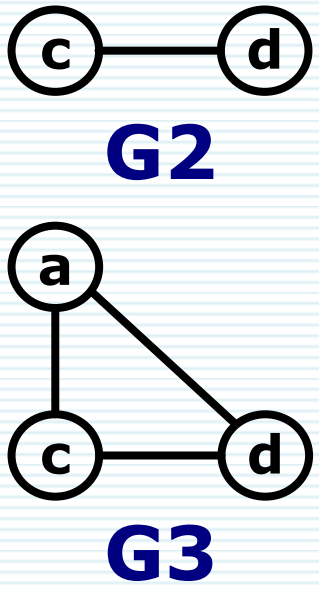
图的定义：路径（Path）

- 顶点 v_a 到 v_b 的路径定义为如下的顶点序列
  - $\{ v_a = v_{i0}, v_{i1}, \cdots, v_{in} = v_b \}$
  - 其中：$(v_{i(k-1)}, v_{ik}) \in E$ $(1<=k<=n)$
- 路径长度：路径上边或弧的数目
- 回路（或环）：首尾顶点相同的路径称为回路或环，即：
  - $\{ v = v_{i0}, v_{i1}, \cdots, v_{in} = v \}$
- 简单路径：路径中不含相同顶点的路径
- 简单回路：除首尾顶点外，路径中不含相同顶点的回路
  - 若通路或回路不重复地包含相同的边，则它是简单的

图的定义：子图

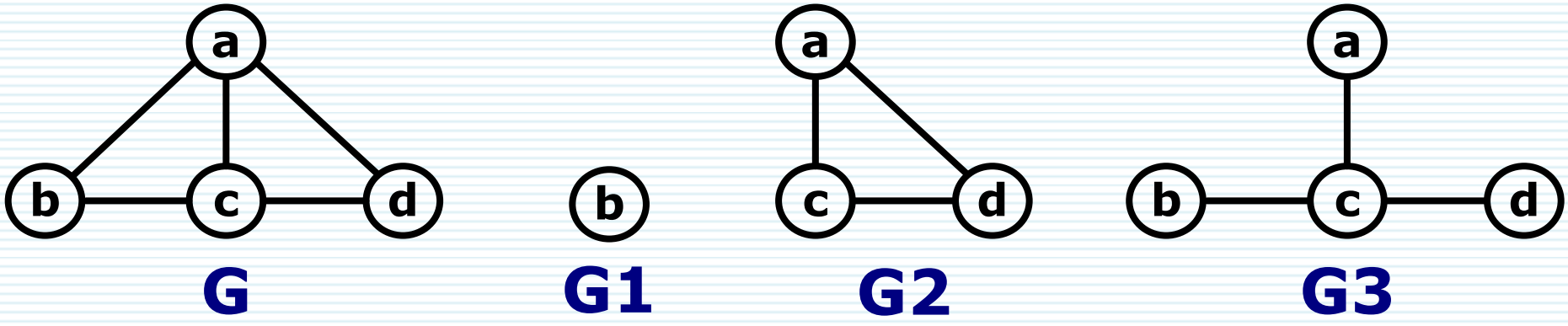






- 子图（Subgraph）
  - 子图是图的一部分，它本身也是一个图
  - 定义：如果图G(V,E)和图G′(V′, E′) 满足：
    - V′ ⊆ V，且 E′⊆ E，则称 G′ 为G的子图
  - 例如：中国铁路交通图的一个子图如图所示

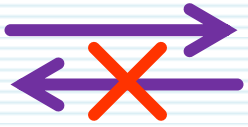
图的定义：连通图



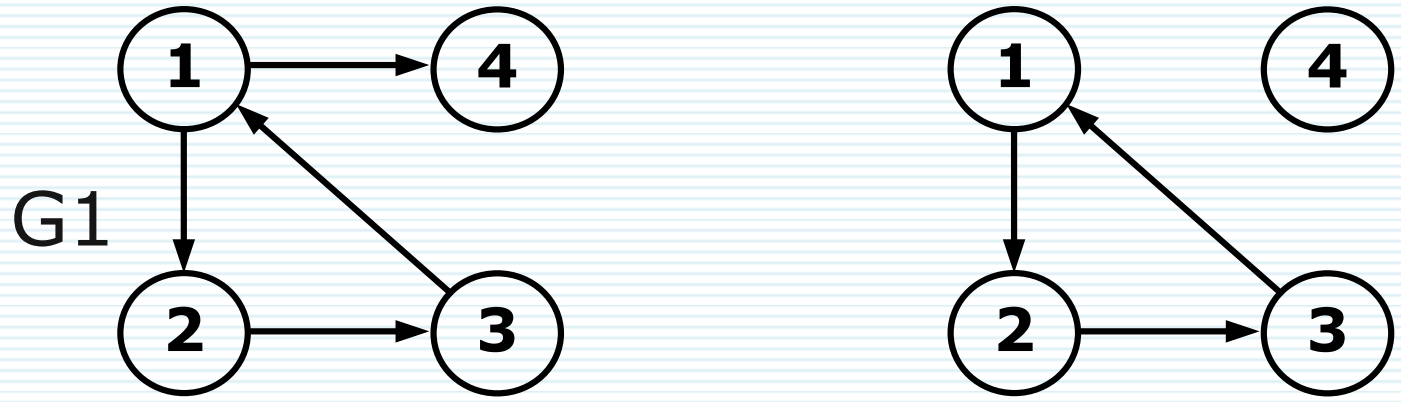
- 顶点连通：
  - 若顶点v到顶点 v′ 有路径，则称顶点v与 v′ 是连通的
- 连通图 （Connected Graph）
  - 若无向图中任意两个顶点$v_i \neq v_j$ 都是连通的
  - 则称该无向图是连通图
- 连通分量：无向图中极大连通子图，称为连通分量

图的定义：生成树（Spanning Tree）

- 生成树的定义
  - 设：**无向图G**是含有n个顶点的连通图
  - 则：G的生成树是含有**n**个顶点且只有**n-1**条边的**连通子图**
- 生成树的定义解读
  - 三要素：n个顶点，n-1条边，连通子图
  - 连通图的生成树是一个极小连通子图
    - 它含有图中的全部顶点
    - 但只有足以构成一棵树的n-1条边
  - 极小连通子图：若再加一条边，必构成环

生成树  **n-1条边**

图的定义：强连通图



- 强连通图
  - 若对于有向图中任意两个顶点$v_i \neq v_j$
  - 都存在从 v_i 到 v_j 和从 v_j 到 v_i 的路径
  - 则称该有向图为强连通图
- 强连通分量
  - 有向图中的极大强连通子图，称为强连通分量

思考题

- 下列叙述中正确的是

  A. 连通分量是无向图中的极小连通子图

  B. 生成树是连通图的一个极大连通子图

  C. 若一个含有n个顶点的有向图是强连通图，则该图中至少有n条弧

  D. 若一个含有n个顶点的无向图是连通图，则该图中至少有n条边

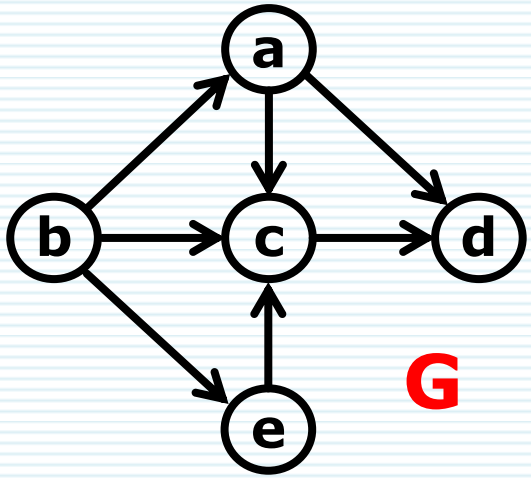
- 正确答案：C

  A. 连通分量是无向图中的极大连通子图

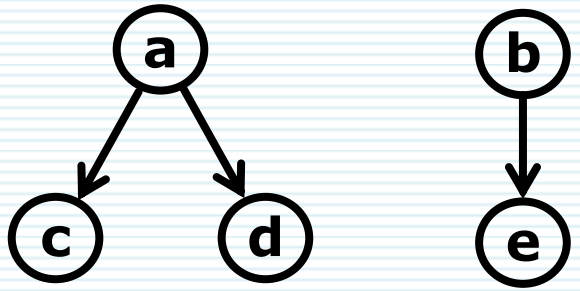
  B. 生成树是连通图的一个极小连通子图

  D. 若一个含有n个顶点的无向图是连通图，则该图中至少有n-1条边

图的定义：生成森林



G



T

- 有向树的定义
  - 若一个**有向图G**恰有一个顶点的入度为0
  - 其余顶点的入度均为1，则它是一棵有向树
- 生成森林的定义
  - 一个有向图的生成森林由若干棵有向树组成
    - 含有图中全部顶点
    - 但只有足以构成若干棵不相交的有向树的弧

图的抽象数据类型

- 图的抽象数据类型定义如下：

ADT Graph{

数据对象：

V是具有相同特性的数据元素的集合，称为顶点集

数据关系：

R={VR}

VR={<v, w>|v, w∈V且P(V, W), <v, w>表示从v到w的弧，
谓词P(V,W)定义了弧<v, w>的意义或信息}

基本操作：

GraphCreate(G, V, VR) **// 以顶点集合V和弧集合VR构造图G**

GraphDestroy(G) **// 删除图G，如果G存在，删除G**

GraphLocateVertex(G, V) **// 定位，返回顶点V在G中的位置**

GraphGetVertex(G, V) **// 取值操作，返回V的值**

图的抽象数据类型

- 图的抽象数据类型定义如下：

ADT Graph{

基本操作(续)：

GraphFirstAdj(G, V)　　**// 查询G中V的第一个邻接顶点**

GraphNextAdj(G, V, W)　**// 求V相对于W的下一个邻接点**

GraphInsertVertex(G, V)　**// 插入顶点**

GraphDeleteVertex(G, V)　**// 删除顶点**

GraphInsertArc(G, V, W)　**// 插入弧**

GraphDeleteArc(G, V)　　**// 删除弧**

DFSTtraverse(G, V, Visit())　**// 深度遍历图G**

BFSTtraverse(G, V, Visit())　**// 广度遍历图G**

}ADT Graph

2. 图的储存方式

图的存储方式

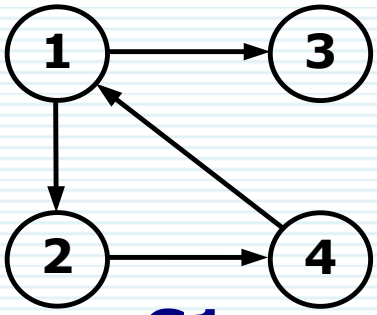
- 图有多种存储方式
  - 顺序存储结构：
    - 邻接矩阵法（二维数组）
  - 链式存储结构：
    - 邻接表法
    - 十字链表法
    - 邻接多重表法

图的顺序存储方式

- 图的数组表示法
  - 使用两个数组分别存储数据元素的信息和数据元素间的关系

```
#define  M 100                    // 顶点的最大个数
typedef struct {
     ElemType vertex;             // 顶点信息
}TVex;
typedef struct {
     int adj;                     // 弧的信息
}TArc;
typedef struct {
     TVex    vexs[M];
     TArc    arcs[M][M];
}TGraph;
```

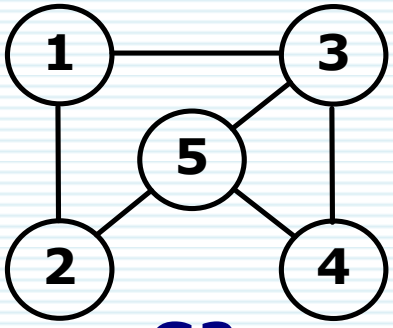
图的顺序存储方式



G1

邻接矩阵 A_1 =

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



G2

邻接矩阵 A_2 =

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

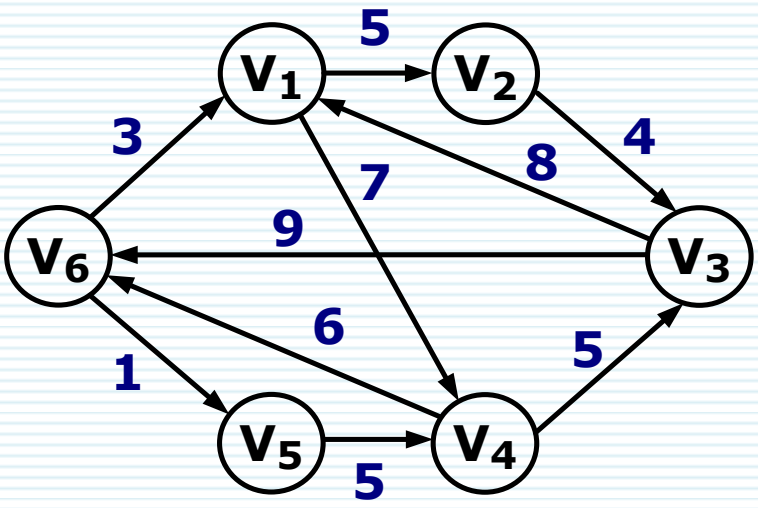
- 图的邻接矩阵表示法
  - int AdjMatrix[M][M]; **// 图的邻接矩阵**

$$\text{AdjMatrix}[i][j] = \begin{cases} 1 & \text{if } (v_i, v_j) \text{ or } \langle v_i, v_j \rangle \in VR \\ 0 & \text{else} \end{cases}$$

邻接矩阵存储方式的特点

- 判定两个顶点 v_i 与 v_j 是否邻接，只需判 A[i,j] 是否为1
- 便于求顶点的度：
  - 无向图中：$Degree(V_i)=\sum_{j=1}^{n}A[i,j]=\sum_{j=1}^{n}A[j,i]$
    - 顶点 v_i 的度等于邻接矩阵中第i行（i列）的元素之和
  - 有向图中：$Degree(V_i)=\sum_{j=1}^{n}A[i,j]+\sum_{j=1}^{n}A[j,i]$
    - 顶点 v_i 的度等于：v_i 的出度 + v_i 的入度
    - 顶点 v_i 的出度为邻接矩阵中第 i 行元素之和
    - 顶点 v_i 的入度为邻接矩阵中第 i 列元素之和

图的顺序存储结构



邻接矩阵 =

$$\begin{bmatrix} 0 & 5 & \infty & 7 & \infty & \infty \\ \infty & 0 & 4 & \infty & \infty & \infty \\ 8 & \infty & 0 & \infty & \infty & 9 \\ \infty & \infty & 5 & 0 & \infty & 6 \\ \infty & \infty & \infty & 5 & 0 & \infty \\ 3 & \infty & \infty & \infty & 1 & 0 \end{bmatrix}$$

- 如果G是带权图：w_{ij} 是边（v_i, v_j）或弧<v_i, v_j>的权
- 则其邻接矩阵定义为：

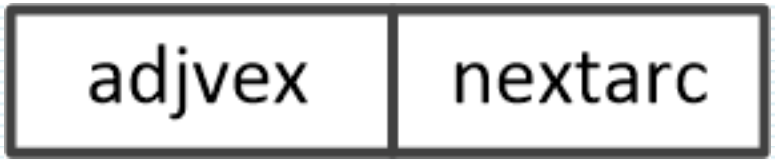
$$A[i,j]=\begin{cases} W_{ij} & \text{若}(v_i,v_j)\text{或}<v_i,v_j>\text{是图G的边}(i\neq j) \\ \infty & \text{若}(v_i,v_j)\text{或}<v_i,v_j>\text{不是图G的边}(i\neq i) \\ 0 & \text{所有对角线元素}(i=j) \end{cases}$$

示例：利用数组表示法创建无向图

```
void createGraph(TGraph  *G){
    int i, j, k, w;
    scanf("%d %d", &G->nv, G->ne)
    for(i=0; i<G->nv; i++){   // 建立顶点列表
        G->vexs[i] = getchar(); fflush(stdin);}
    for(i=0; i<G->nv; i++)    // 邻接矩阵初始化
        for(j=0; j<G->nv; j++)
            G->edges[i][j]=
    for(k = 0; k < G->ne;
        scanf("%d %d %d
        G->edges[i][j] = w
    }
}
```

```
typedef struct{
    char *vexs        // 顶点表
    int *edges        // 邻接矩阵
    int nv, ne;       // 顶点数和边数
}TGraph;
```

图的链式存储结构：邻接表

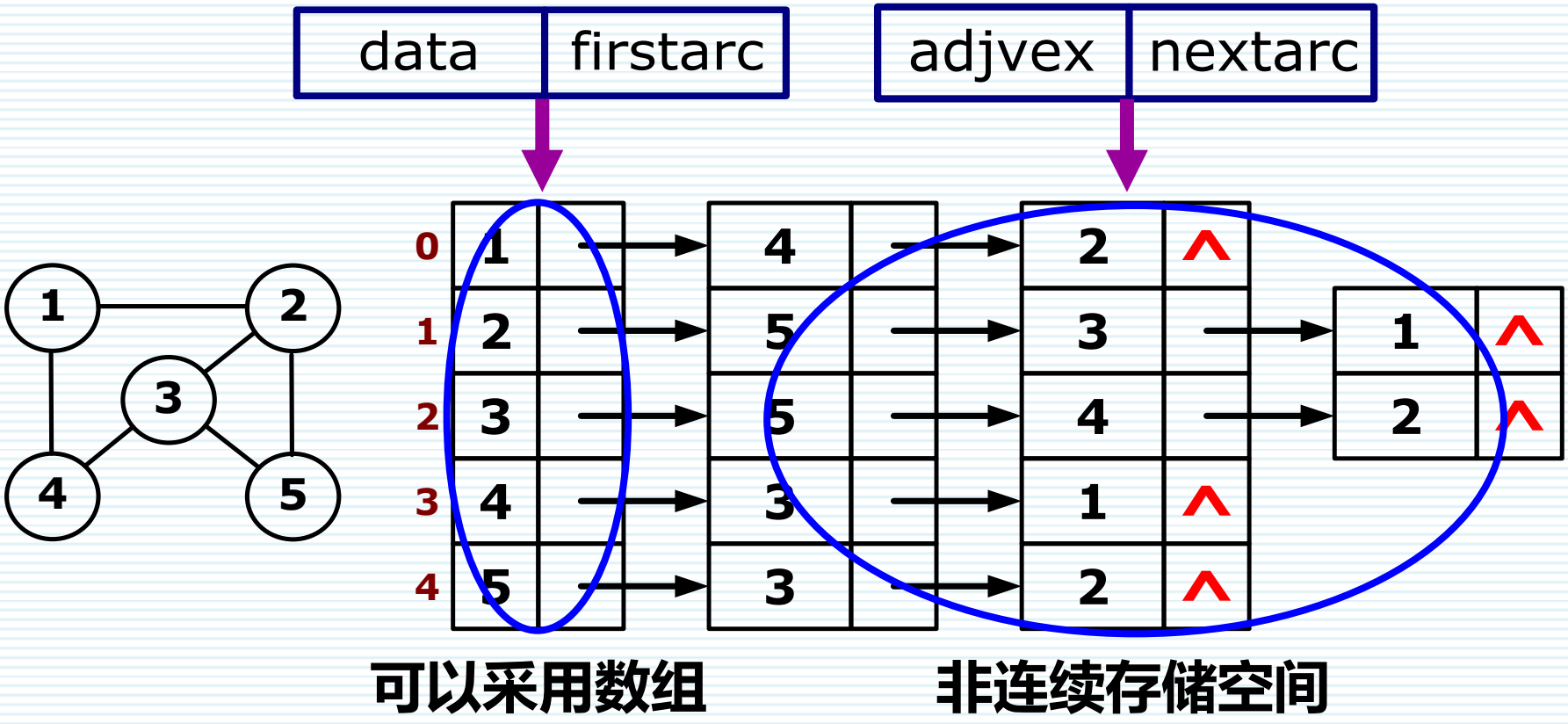


链表结点

头结点

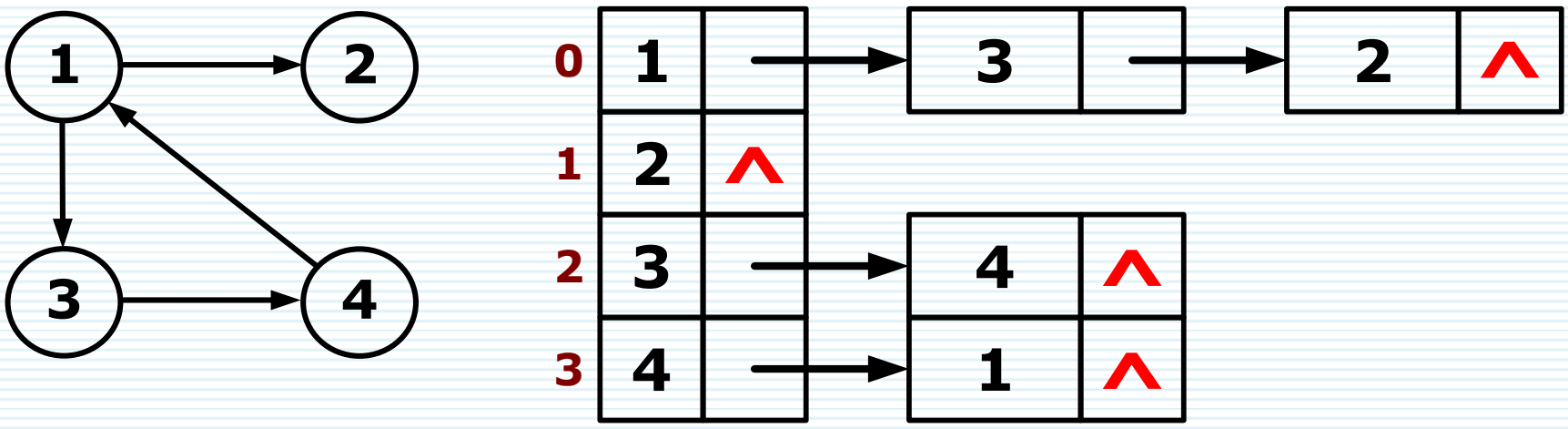
- 邻接表是图的一种链式存储结构
  - 对图中的每个顶点建立一个单链表
  - 单链表中的第i个结点表示依附于顶点vi的顶点
- 每个**链表结点**包含两个域：
  - adjvex：记载与顶点v_i 邻接的顶点信息
  - nextarc：指向下一个与顶点v_i 邻接的结点
- 每个链表附设一个**头结点**：
  - data：存放顶点信息（如：姓名、编号等）
  - fristarc：指向链表的第一个结点

无向图的邻接表表示法



- 无向图邻接表特点
  - n个顶点 e条边的无向图：需n个头结点和2e个链表结点
  - 顶点 v_i 的度： degree（v_i）= 链表 i 中的链表结点数

有向图的邻接表表示法



第 i 个链表上的结点是以 v_i 为弧尾的各个弧头顶点

- 有向图邻接表特点：
  - n个顶点 e条边的有向图：需n个头结点和e个链表结点
  - 第 i 条链表上的链表结点数：为 v_i 的出度
  - 因此：求顶点的出度易，求入度难

有向图邻接表的数据结构

```
typedef struct node{          // 链表结点（边结点）
    int adjvex;               // 邻接点
    struct node *nextarc;     // 下一邻接点指针
    int nv, ne;               // 数据域
}ENode;

typedef struct vnode{         // 表头结点
    ElemType data;            // 顶点域
    ENode *firstarc;          // 边的表头指针
}VNode;
```

有向图邻接表的数据结构

```
#define M 100      // 最大结点数
typedef struct vnode{       // 表头结点
    ElemType data;      // 顶点域
    ENode *firstarc;          // 边的表头指针
}VNode;
VNode adjlist[M];             // 邻接表
```

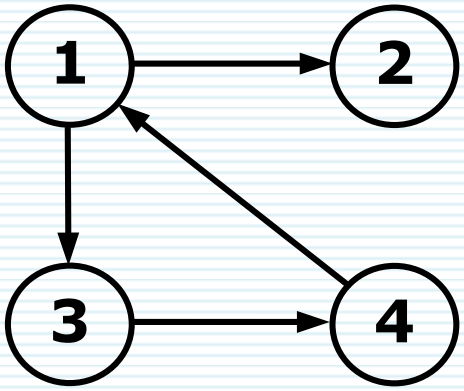
```
typedef struct{
    VNode adjlist[M];  // 邻接表
    int nv, ne;                  // 图中顶点数和边数
}TGraph;
```

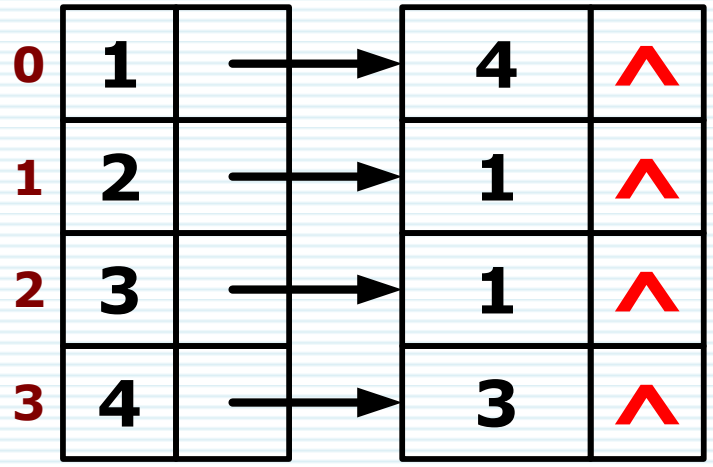
建立有向图的邻接表

```
void createGraph(TGraph *G){
    int  i, j, k;   ENode *s;
    for(i = 0; i < G->nv; i++){  // 建立顶点表
        G->adjlist[i].data = getchar(); // fflush(stdin)
        G->adjlist[i].firstarc = NULL;     // 边表置为空表
    }
    for(k = 0; k < G->ne; k++){   // 建立边表
        scanf("%d %d", &i, &j); fflush(stdin);
        s = (ENode *)malloc( sizeof(ENode)); //边表结点
        s->adjvex = j;   //邻接点的序号为j
        s->nextarc = G->adjlist[i].firstarc;
        G->adjlist[i].firstarc = s;  //将新结点插入边表头
    }
}
```

思考：如果是无向图?

有向图的逆邻接表表示法

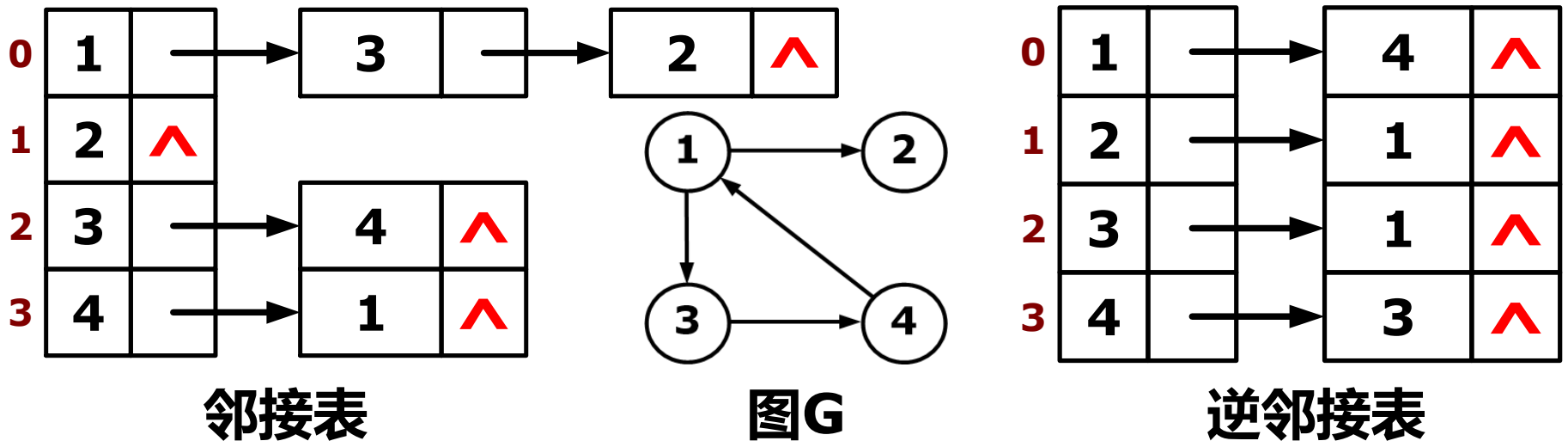




第 i 个链表上的结点是以 v_i 为弧头的各个弧尾顶点

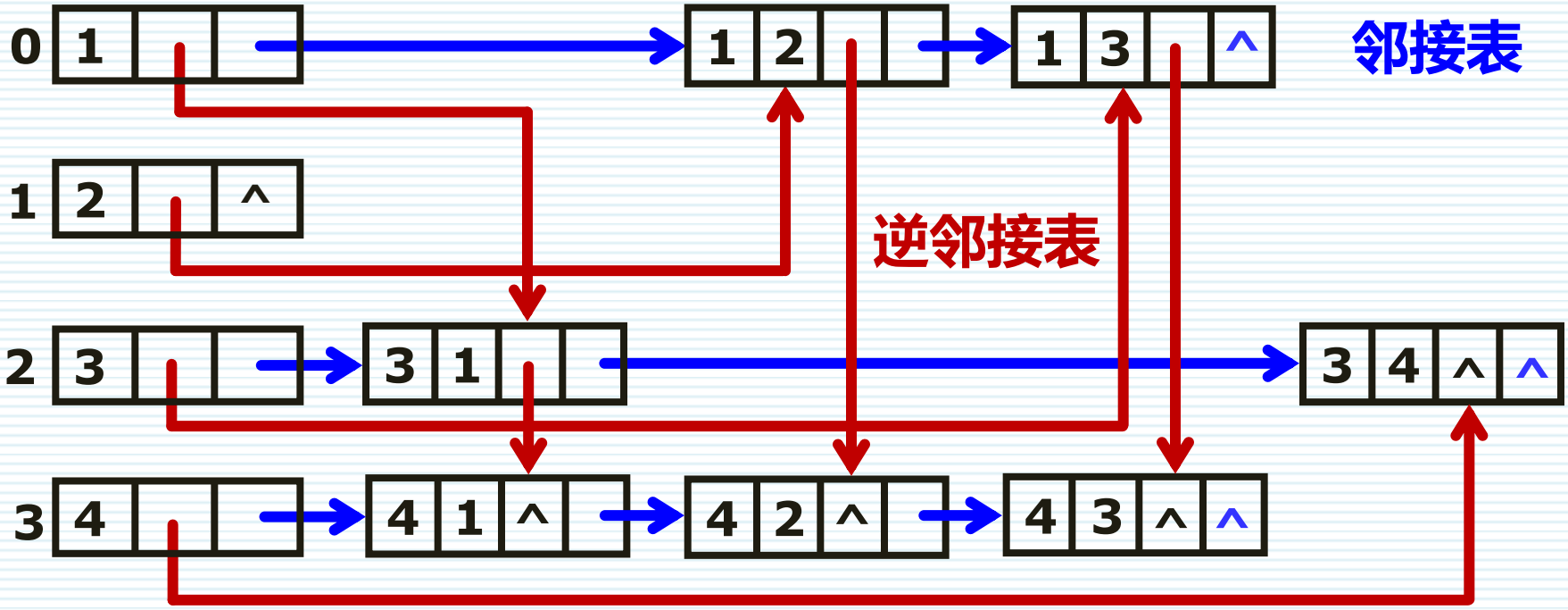
- 有向图逆邻接表特点：
  - n个顶点 e条边的有向图：需n个头结点和e个链表结点
  - 第 i 条链表上的链表结点数：为 v_i 的入度
  - 因此：求顶点的入度易，求出度难

图的链式存储结构：十字链表

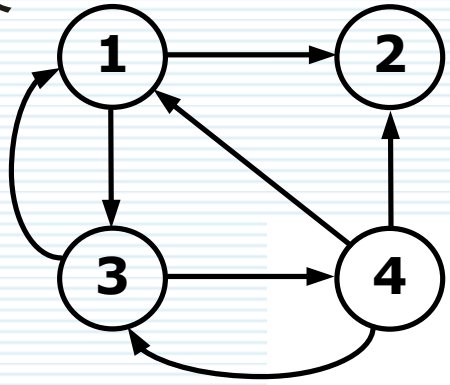


- 邻接表：弧尾相同的结点组成链表
  - 便于求结点出度
- 逆邻接表：弧头相同的结点组成链表
  - 便于求结点入度
- 思考：怎样结合二者的优点?
- 十字链表：将有向图的邻接表和逆邻接表相结合

图的链式存储结构：十字链表



- 思路：扩展头结点指针域，分别指向两种邻接表
- 相应地扩展链表结点
  - 保存弧的弧头和弧尾信息
  - 设置两个指针域，分别跟踪入度和出度



图的链式存储结构：十字链表

顶点结点：

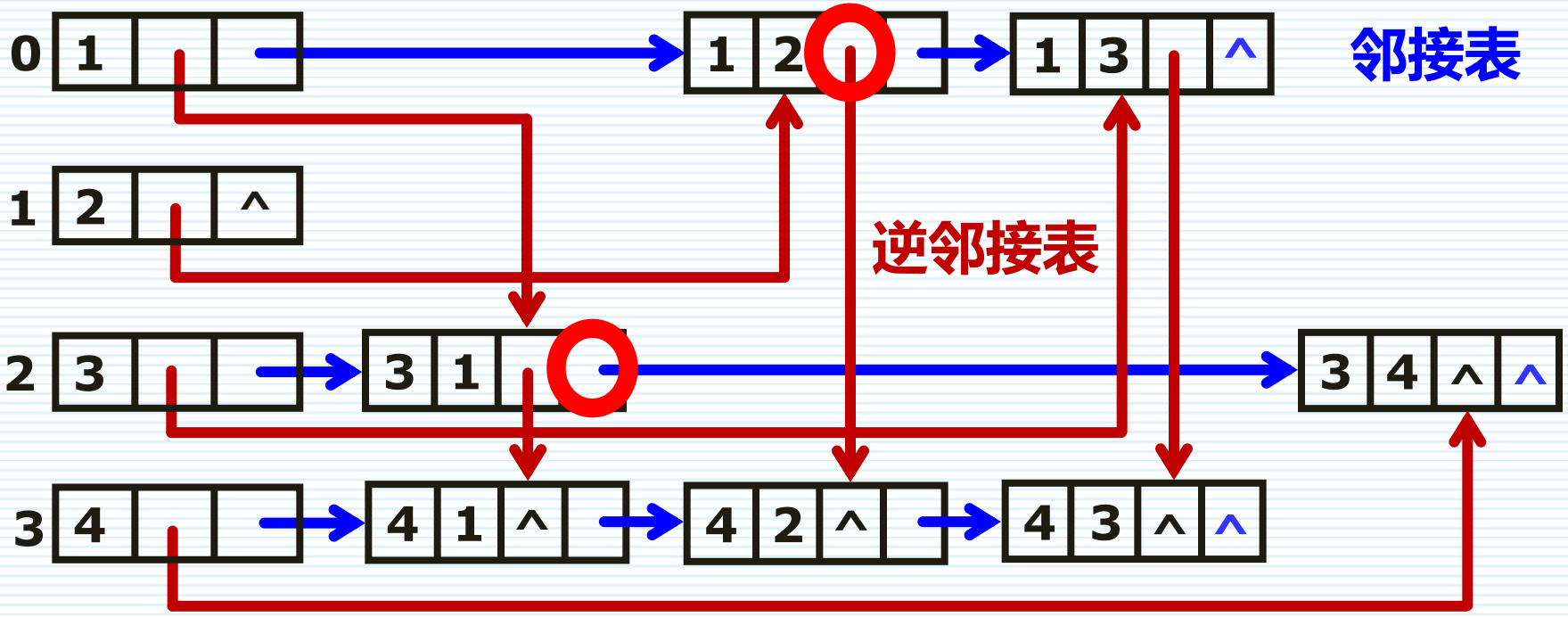
data	firstin	firstout

弧结点：

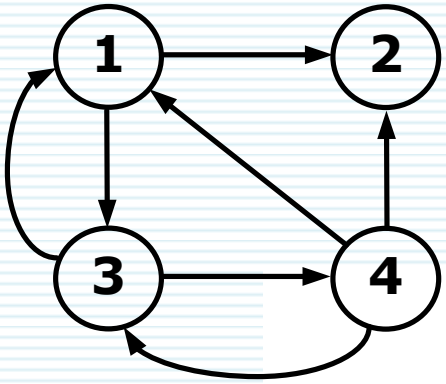
tailvex	headvex	hlink	tlink

- 十字链表：将有向图的邻接表和逆邻接表相结合
  - 顶点结点：
    - data：　存放顶点信息（如顶点的名称或位置）
    - firstin：　指向以该顶点为弧头的第一个弧结点
    - firstout：指向以该顶点为弧尾的第一个弧结点
  - 弧结点：
    - headvex，tailvex：　该弧的弧尾与弧头顶点编号
    - hlink：　指向弧头相同的下一条弧
    - tlink：　指向弧尾相同的下一条弧

图的链式存储结构：十字链表



- 通过hlink将弧头相同的弧连在一个链表上
- 通过tlink将弧尾相同的弧连在一个链表上
- hlink链和tlink链的头结点共用顶点结点



图的链式存储结构：十字链表

- 十字链表的特点
  - 顶点结点数 = 顶点数
  - 弧结点数 = 弧的条数
- 求顶点入度
  - 从顶点 v_i 的 firstin 指针出发
  - 沿着弧结点中的hlink遍历链表所经过的弧结点数
- 求顶点出度
  - 从顶点 v_i 的 firstout 指针出发
  - 沿着弧结点中的tlink遍历链表所经过的弧结点数

有向图十字链表的数据结构

```
typedef struct node{                    // 弧结点
    int headvex, tailvex;               // 头尾编号
    struct node *hlink, *tlink; // 链域
}ENode;
```

```
typedef struct {                        // 表头结点
    ElemType data;                  // 数据域
    ENode *firstin, *firstout;      // 边表头指针
}VNode;
```

有向图十字链表的数据结构

```
#define M 100      // 最大结点数
typedef struct {                        // 表头结点
    ElemType data;                  // 数据域
    ENode *firstin, *firstout;      // 边表头指针
}VNode;
VNode OrthList[M];                      // 十字链表
```

```
typedef struct{
    VNode OrthList[M];              // 十字链表
    int nv, ne;                     // 图中顶点数和边数
}OrthGraph;
```

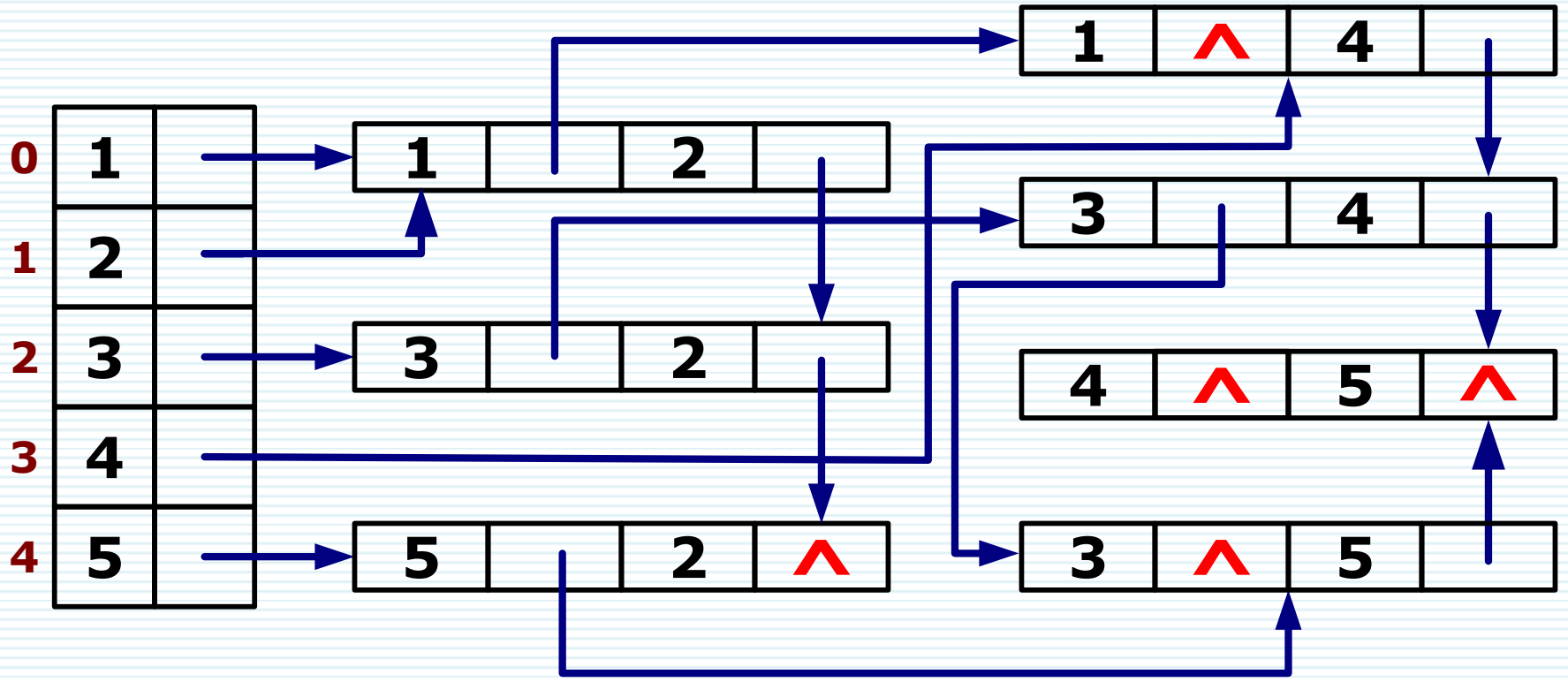
建立有向图的十字链表

```
void createGraph(OrthGraph *G){ int i, j, k;
    for(i = 0; i < G.nv; i++){
        printf("input the %dth data\n", i);
        scanf("%c", G->OrthList[i].data);
        G.OrthList[i].firstin = NULL;
        G.OrthList[i].firstout = NULL;
    }
    for(k = 0; k < G.ne; k++){
        char v1, v2; printf("input <v1, v2>\n");
        scanf("%c %c", &v1, &v2);   // fflush(stdin)
        i = locate(G, v1); j = locate(G, v2);
        ENode *p =(ENode *)malloc(sizeof(ENode));
        p->tailvex = i;  p->headvex = j;
        p->hlink = G.OrthList[j].firstin;
        p->tlink = G.OrthList[i].firstout;
        G.OrthList[j].firstin=p;  G.OrthList[i].firstout=p;
    }
}
```

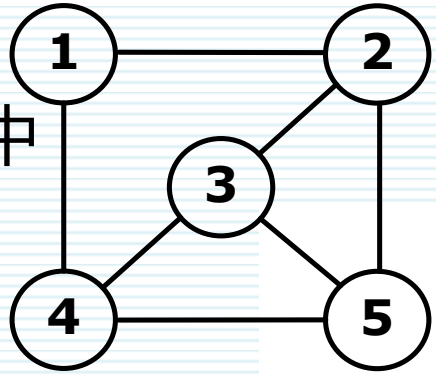
图的链式存储结构：邻接多重表

- 邻接多重表是无向图的另一种链式存储结构
  - 边结点：
    - ivex 和 jvex：为该边所依附的两个顶点的编号
    - ilink：指向下一条依附于顶点 ivex 的边
    - jlink：指向下一条依附于顶点 jvex 的边
  - 顶点结点：
    - data：存放顶点信息（如顶点的名称或位置）
    - firstedge：指向第一条依附于该顶点的边结点

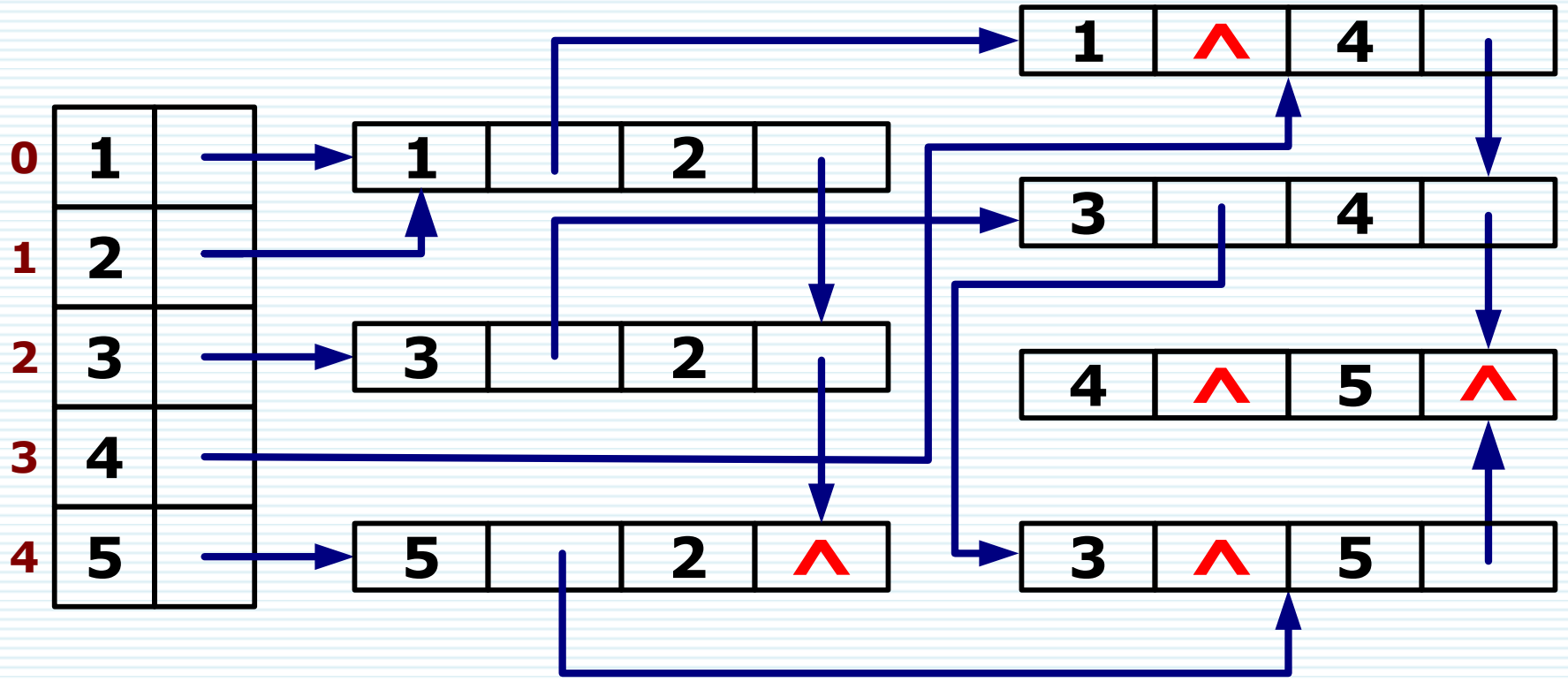
图的链式存储结构：邻接多重表



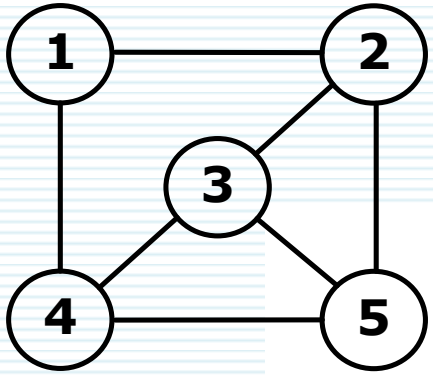
- 邻接多重表的特点
  - 所有依附于同一顶点的边被串联到同一链表中
  - 由于每个边有两个顶点
    - 因此每条边又被串联到两个链表中



图的链式存储结构：邻接多重表



- 邻接多重表的特点
  - 顶点结点数为n，边结点数为e
  - 便于得到边的两个顶点
  - 如：删除一条边，或判断边是否已访问



无向图邻接多重表的数据结构

```
typedef struct node{
    int ivex, jvex;                    // 头尾编号
    struct node *ilink, *jlink;  // 链域
}ENode;
```

```
typedef struct{              // 表头结点
    ElemType data;      // 数据域
    ENode *firstedge; // 边表头指针
}VNode;
```

无向图邻接多重表的数据结构

```
#define M 100      // 最大结点数
typedef struct {                    // 表头结点
    ElemType data;                  // 数据域
    ENode * firstedge;              // 边表头指针
}VNode;
VNode OrthList[M];                  // 邻接多重表
```

```
typedef struct{
    VNode OrthList[M];              // 邻接多重表
    int nv, ne;                     // 图中顶点数和边数
}OrthGraph;
```

3. 图的遍历

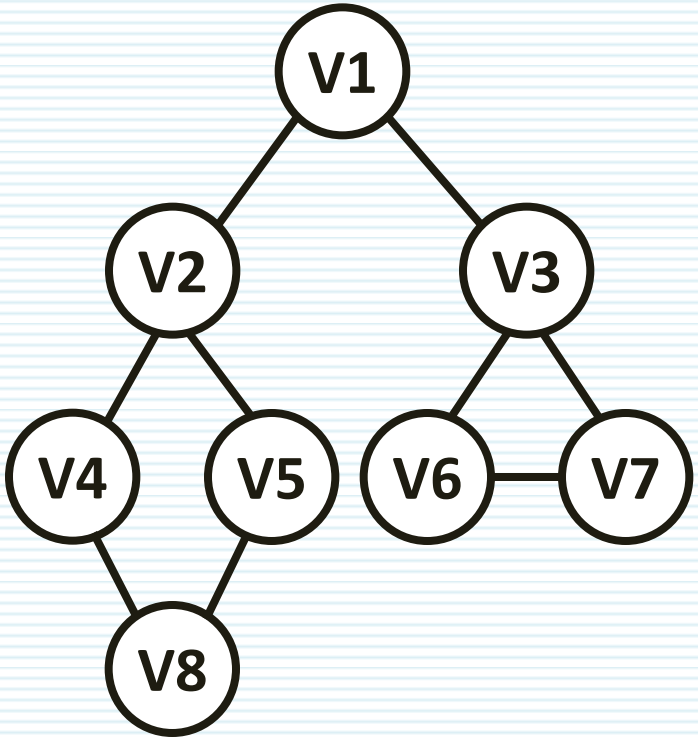
图的遍历

- 图的遍历
  - 定义：从图中某个顶点出发，沿图中的路径，使图中每个顶点被访问且仅被访问一次的过程，称为图的遍历
  - 回顾其他已经学过的数据结构的遍历方法
    - 顺序表的遍历
    - 单链表的遍历
    - 二叉树的遍历
  - 思考：对于图应该怎样遍历？ { **深度优先遍历** / **广度优先遍历** }
  - 注意：由于图中任一顶点可能与其余的顶点相邻接
    - 所以：遍历时可能沿着某条路径会回到该顶点
    - 因此：通常遍历时需考虑记录已访问过的顶点

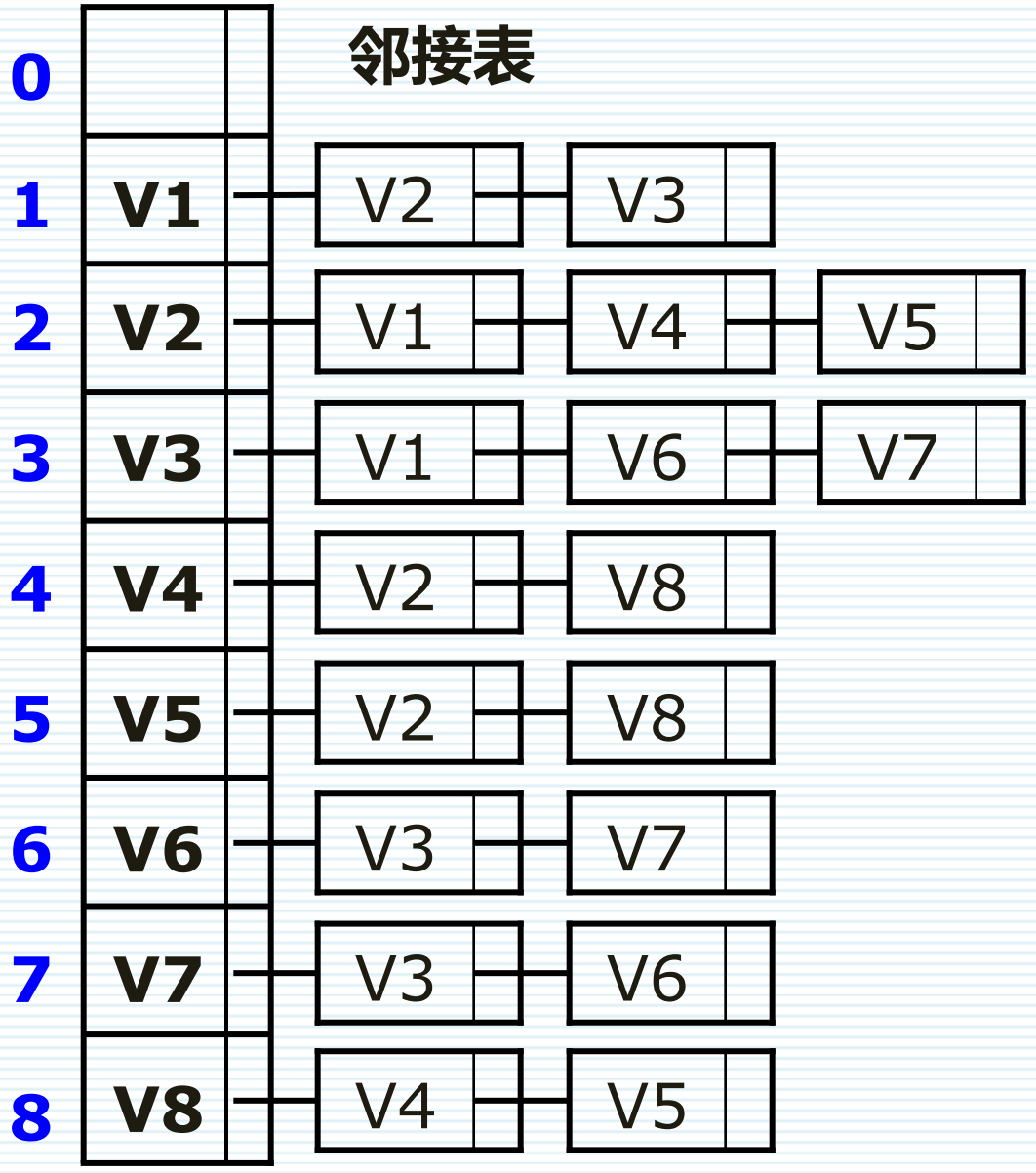
图的深度优先遍历

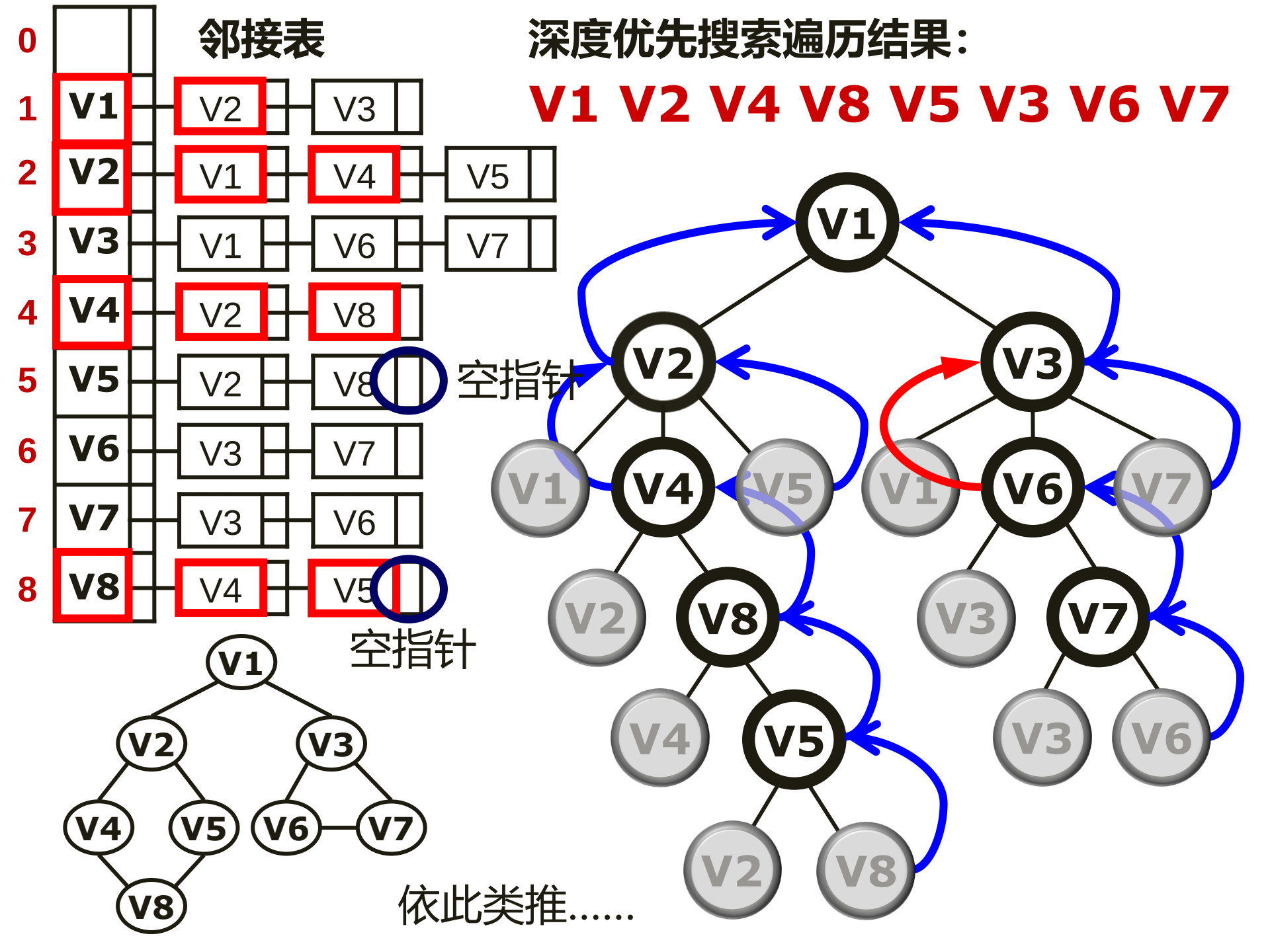
- 深度优先搜索算法（depth-first-search）
  - 访问指定的起始顶点 v_0，将 v_0 作为当前顶点
  - 依次从 v_0 的邻接结点出发深度优先对图进行遍历
    - 直到图中所有和 v_0 有路径相通的顶点都被访问到
    - 思考：此时是否当前顶点的所有邻接点都被访问过?
    - 思考：此时是否仍有顶点未能访问到?
  - 若此时尚有顶点没有被访问到?
    - 则从中选取一个顶点作为起始点
    - 重复上述步骤，直到图中所有顶点均被访问到为止
  - 要想遍历到图中全部结点需进行多趟遍历
    - 每趟遍历一个极大连通分量

深度优先搜索算法

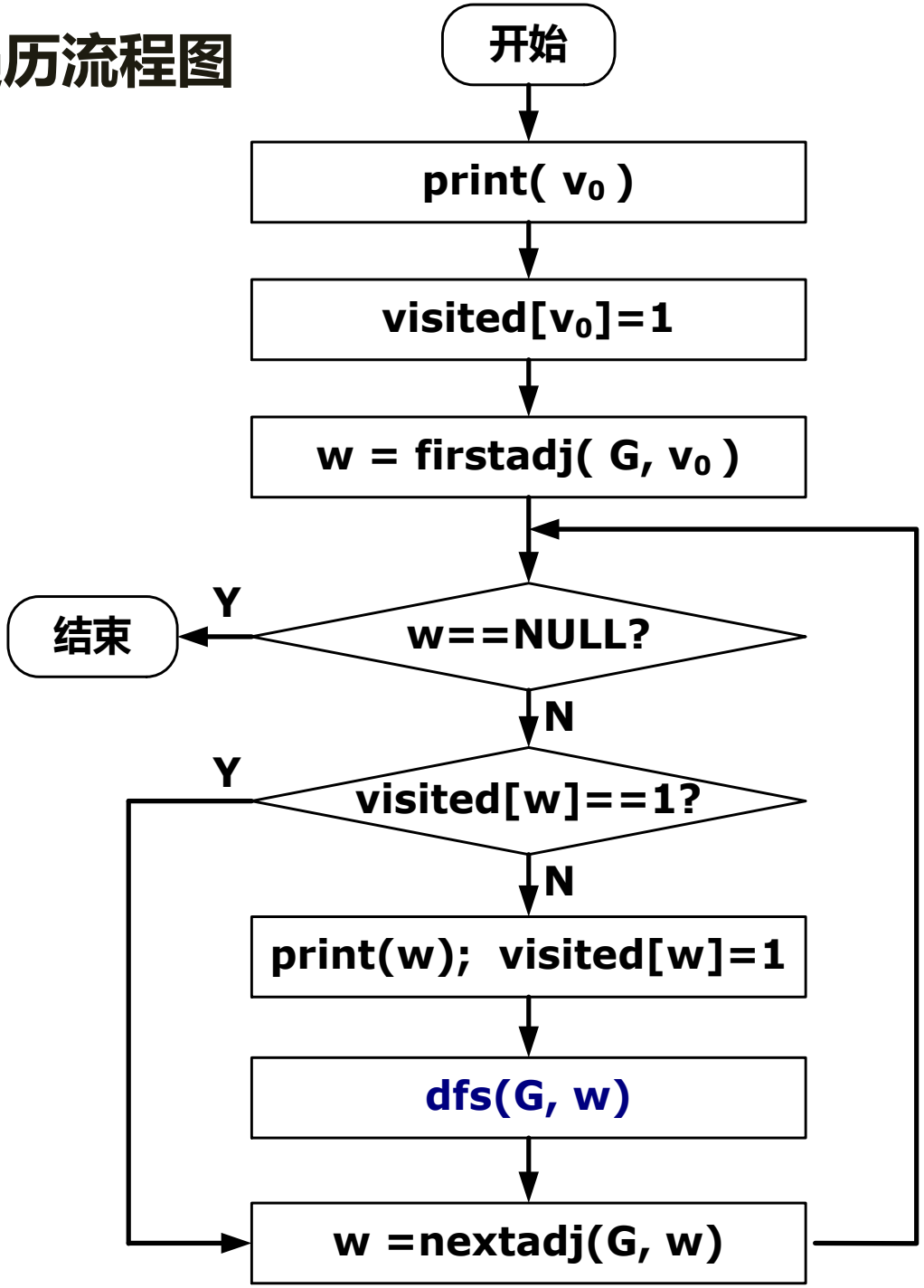

V1
V2
V3
V4
V5
V6
V7
V8

G6


邻接表
0
1
V1
V2
V3
2
V2
V1
V4
V5
3
V3
V1
V6
V7
4
V4
V2
V8
5
V5
V2
V8
6
V6
V3
V7
7
V7
V3
V6
8
V8
V4
V5


邻接表
深度优先搜索遍历结果：
V1 V2 V4 V8 V5 V3 V6 V7
0
1 V1 V2 V3
2 V2 V1 V4 V5
3 V3 V1 V6 V7
4 V4 V2 V8
5 V5 V2 V8
空指针
6 V6 V3 V7
7 V7 V3 V6
8 V8 V4 V5
空指针
V1
V2 V3
V4 V5 V6 V7
V8
依此类推……

深度优先遍历流程图

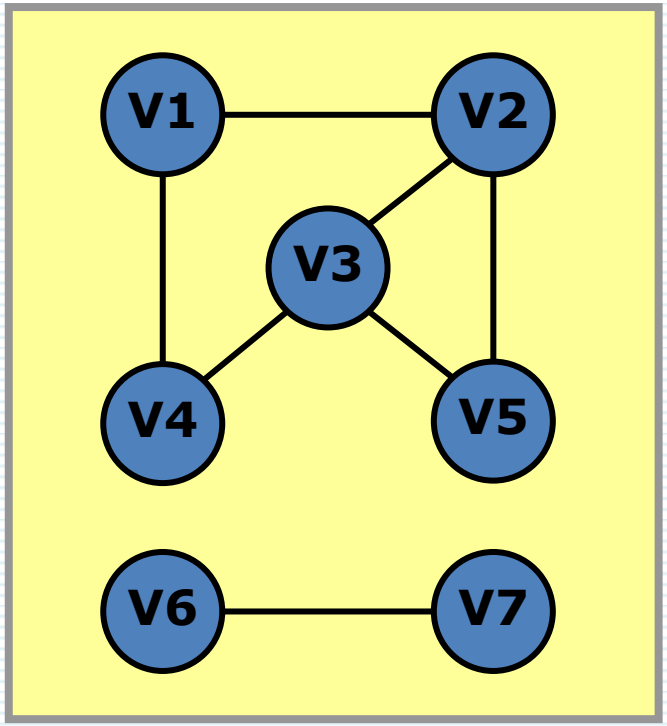

开始
print(v0)
visited[v0]=1
w = firstadj(G, v0)
结束
Y
w==NULL?
N
Y
visited[w]==1?
N
print(w); visited[w]=1
dfs(G, w)
w =nextadj(G, w)

图的深度优先搜索算法（**D**epth-**F**irst-**S**earch)

- 搜索过程中需要知道顶点是否已经被访问过
  - 实现：设置一个标志数组 visited[1..n]
  - 初始化：visited[i]＝0 （i∈[0, n-1])
  - 若顶点w被访问，则令 visited[w]=1
- 还需要能够求得当前结点的邻接点
  - 求初始结点 v_0 的第一个邻接点：firstadj(G, v_0)
  - 求当前结点 w 的下一个邻接点：nextadj(G, w)
  - 这两个函数的实现与图的具体存储结构有关

图的深度优先遍历算法

```
void traverse(TGraph *G)  { // 图G采用邻接表存储
    // 标志数组初始化
    for(int i=1; i<=G->nv; i++){
        visited[i] = 0;
    }
    // 对图G执行深度优先遍历
    for(int i=1; i<=G->nv; i++){
        if(visited[i]==0){
            DFS(G, i);
    }
}
```



思考： traverse函数调用DFS(G, i)的次数由什么决定?

有向图邻接表的数据结构

```
typedef struct node{
    int adjvex;                 // 邻接点
    struct node *nextarc;       // 下一邻接点指针
    int nv, ne;                 // 数据域
}ENode;

typedef struct vnode{           // 表头结点
    ElemType data;              // 顶点域
    EdgeNode *firstarc;         // 边的表头指针
}VNode;
```

从顶点 v_i 出发深度优先搜索图G

```
void DFS(TGraph *G, int i){   // 图G采用邻接表存储
    ENode *p;
    printf("%c", G->adjlist[i].vertex);  // 访问顶点vi
    visited[i] = 1;                      // 标记vi已访问
    p = G->adjlist[i].firstarc;    // 取vi边表的头指针
    while(p){   // 依次搜索vi的邻接点vj
        if(!visited[p->adjvex])  // 若vi尚未被访问
            DFS(G, p->adjvex);
        p = p->nextarc;              // 取vi的下一邻接点
    }
}
```

时间复杂度：O(n+e)

回顾：图的顺序存储方式

```
#define  M 100                          // 顶点的最大个数
typedef struct {
     ElemType vertex;                   // 顶点信息
}TVex;
typedef struct {
     int adj;                           // 弧的信息
}TArc;
typedef struct {
     TVex    vexs[M];   int nv, ne;
     TArc    arcs[M][M];
}TGraph;
```

从顶点 v_i 出发深度优先搜索图G

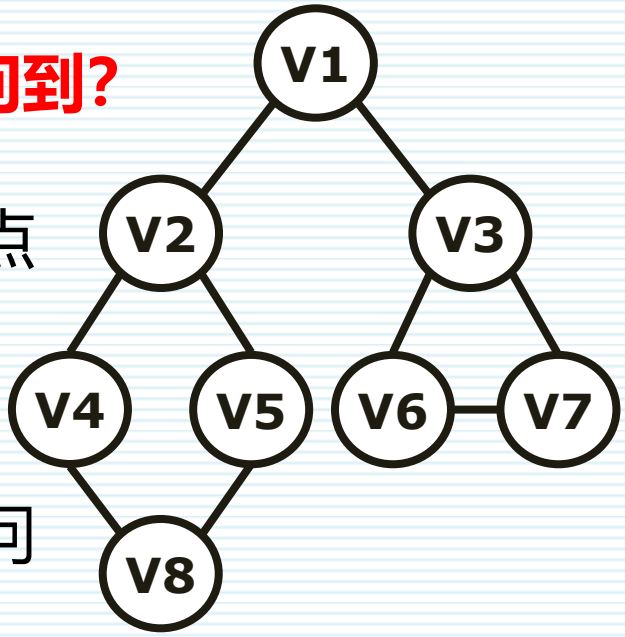
```
void DFS2(TGraph *G, int i){ // 图G采用邻接矩阵存储
    int j;
    printf("%c", G->vexs[i]);    // 访问顶点vi
    visited[i] = 1;
    for(j = 0, j < G->nv; j++){   // 依次遍历vi的邻接点
        if(G->arcs[i][j] == 1 && !visited[j])
            DFS2(G, j);
    }
}
```

时间复杂度：$O(n^2)$

图的广度优先遍历

思考：若仍有顶点未访问到?

- 则选取一个作为起始点
- 重复上述步骤
- 直至所有顶点均被访问



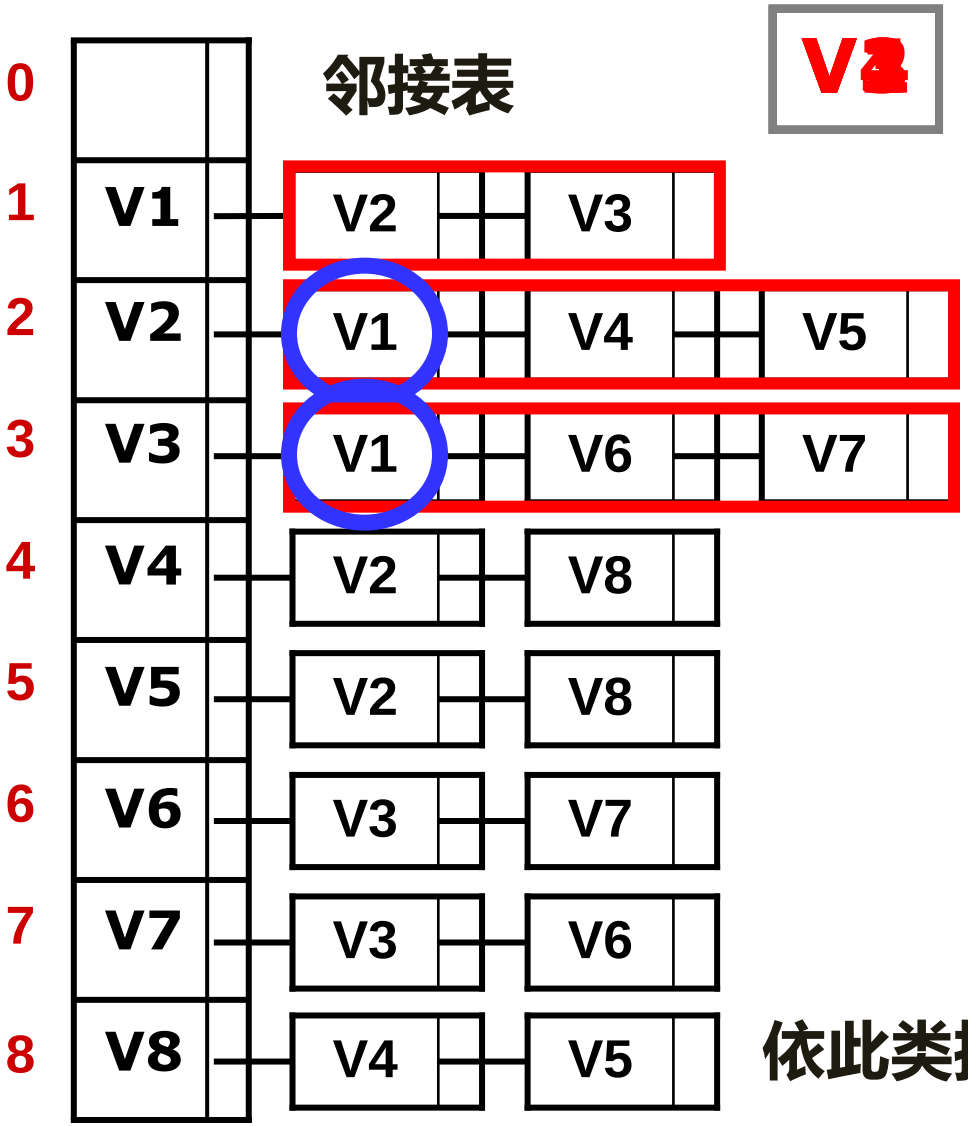
访问顶点序列为：

V1 V2 V3 V4
V5 V6 V7 V8

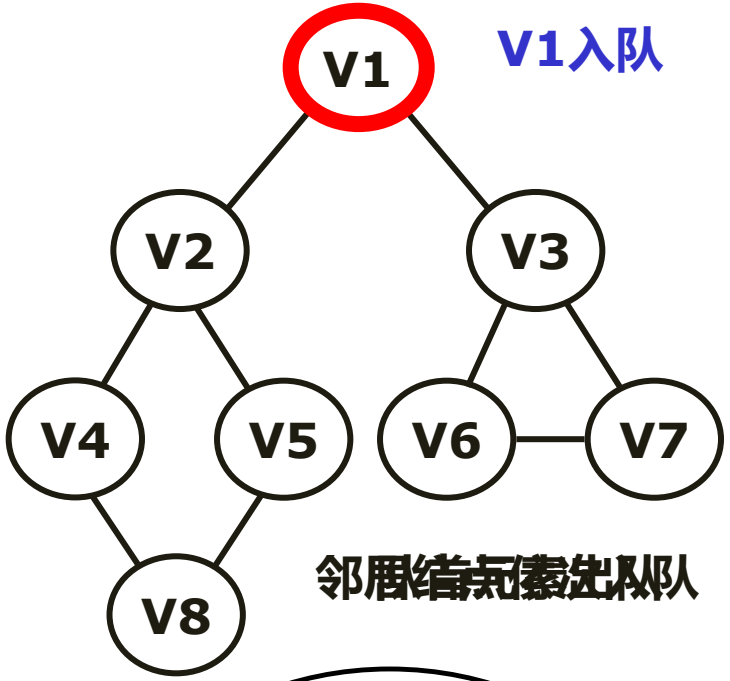
- 广度优先搜索算法（breadth-first-search）
  - 访问某个起始顶点 V_0，将 V_0 作为当前顶点
    - 依次访问当前顶点的所有未访问过的邻接点
  - 然后分别从这些邻接顶点出发广度优先遍历图
  - 直至图中所有已被访问过的顶点的邻接点都已被访问过为止

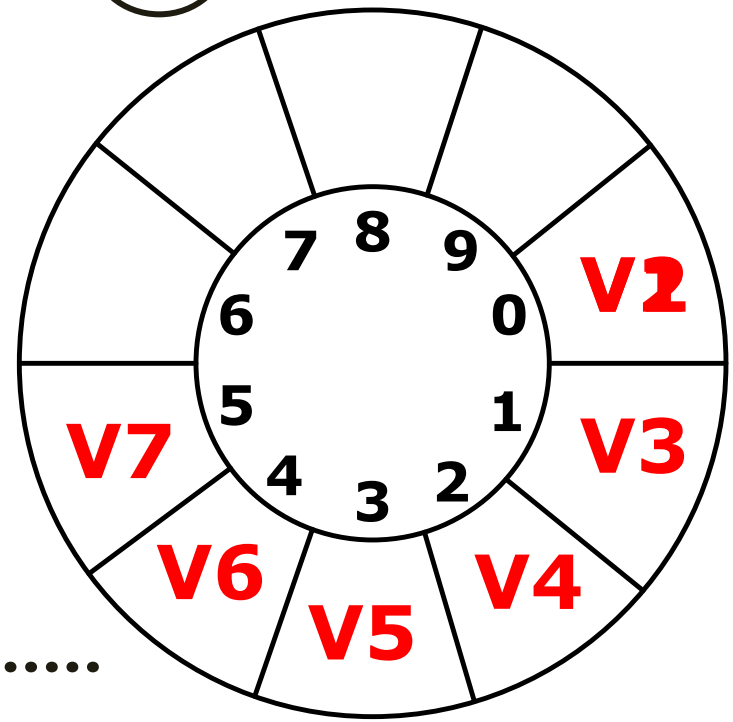
广度优先搜索遍历结果：

V1 V2 V3 V4 V5 V6 V7 V8



依此类推……





回顾：有向图邻接表的数据结构

```
typedef struct node{
    int adjvex;                 // 邻接点
    struct node *nextarc;       // 下一邻接点指针
    int nv, ne;                 // 数据域
}ENode;

typedef struct vnode{           // 表头结点
    ElemType data;              // 顶点域
    EdgeNode *firstarc;         // 边的表头指针
}VNode;
```

广度优先搜索算法（邻接表）

```
void BFS(TGraph *G,int *visited, int k){
   int i; ENode *p; PQue que = init_que(M);
   printf("visit: %c",G->adjlist[i].data);
   visited[k] = 1; enque(que, k);
   while(!is_empty(que)){
      i = deque(que); p = G->adjlist[i].firstarc;
      while(p) {    // 依次搜索vi的邻接点
         if(!visited[p->adjvex]) {
            printf("visit: %c", p->adjvex); // 访问vj
            visited[p->adivex] = 1;
            enque(que, p->adjvex);
         }
         p = p->nextarc; // 找vi的下一个邻接点
      }
   }
}
```

思考：若图不是连通图，如何进行广度优先遍历?

时间复杂度：O(n+e)

回顾：图的顺序存储方式

```
#define  M 100                          // 顶点的最大个数
typedef struct {
     ElemType vertex;                   // 顶点信息
}TVex;
typedef struct {
     int adj;                           // 弧的信息
}TArc;
typedef struct {
     TVex    vexs[M];   int nv, ne;
     TArc    arcs[M][M];
}TGraph;
```

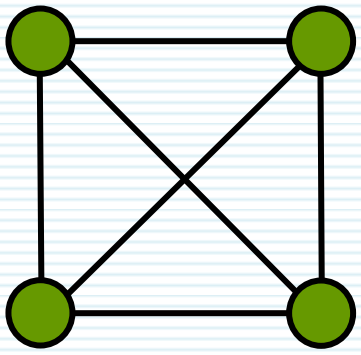
广度优先搜索算法（邻接矩阵）

```
void BFS2(TGraph *G,int *visited, int k){
   int i, j;  PQue que = init_que(M);
   printf("visit: %c", G->vexs[k]);
   visited[k] = 1; enque(que, k);
   while(!is_empty(que)){
      i = deque(que);
      for(j = 0, j < G->nv; j++){ // 依次搜索vi的邻接点
         if(G->arcs[i][j]==1 && !visited[j]) {
            printf("visit: %e", G->vexs[j]); // 访问vj
            visited[j] = 1;
            enque(que, j);
         }
      }
   }
}
```

时间复杂度：$O(n^2)$

4. 最小生成树

最小生成树



G

G的生成树

- 如果连通图**G**的一个子图**G'**是一棵包含**G**的所有顶点的树
  - 则该子图称为**G**的生成树（spanning tree）
- 若子图G'是图G的生成树，需满足如下三个条件：
  - V(G')=V(G)； G'是连通的； G'中无回路
- 对于具有n个顶点的无向连通图G而言
  - 其任一生成树（G'）恰好包含n-1条边
  - 生成树不一定唯一

最小生成树

- 对于连通的带权图G，其生成树也是带权的
  - 生成树T的边的权值总和称为该树的权，记作：

$$W(T)=\sum_{(u,v)\in E(T)} W(u,v)$$

    - 其中：E（T）表示 T 的边集
    - 权最小的生成树称为图G的最小生成树
- 最小生成树：一个无向图G的最小生成树就是
  - 由该图连接所有顶点的边构成的树，且总价值最低
  - 简称 MST：Minimum Spanning Tree
  - 最小生成树也不一定唯一

最小生成树性质

- 设：G=(V,E)是一个连通网络，U是顶点集V的一个真子集
  - 若(u,v)是一条具有最小权值的边（u∈U，v∈V-U)
  - 则一定存在一棵包含边(u,v)的G的最小生成树
- 证明：采用反证法
  - 假设：G中任何一棵最小生成树都不包括(u,v)
  - 若：T是G的一棵最小生成树
    - 则：T不包括(u,v)，且T中必有一条由u到v的路径P
  - 将边(u,v)加入到树T中
    - 边(u,v)与路径P必定构成一个回路
    - 删除路径P中与u或v相邻的边可得另一个生成树T’
  - 由于边(u,v)是最小权值的边，所以：W(T’) ≤ W(T)
  - 所以T’也是G的最小生成树，与假设矛盾，得证

利用MST性质求解最小生成树问题

- MST性质
  - 若：(u, v) = min{ cost(x,y) | x∈U, y∈V-U }
    - 则必定存在一棵包含边（u，v）的最小生成树
  - MST性质实际上揭示了MST问题的贪心选择性质
- 因此可以利用贪心算法设计策略求解
  - 第一种贪心选择策略：子树生长法
    - Prim算法
  - 第二种贪心选择策略：短边优先法（避圈法）
    - Kruskal算法

Prim算法

Prim算法流程

设：图G=(V,E)，生成树T的顶点集合为U

1. 任取图中一个顶点 v_0 加入集合 U

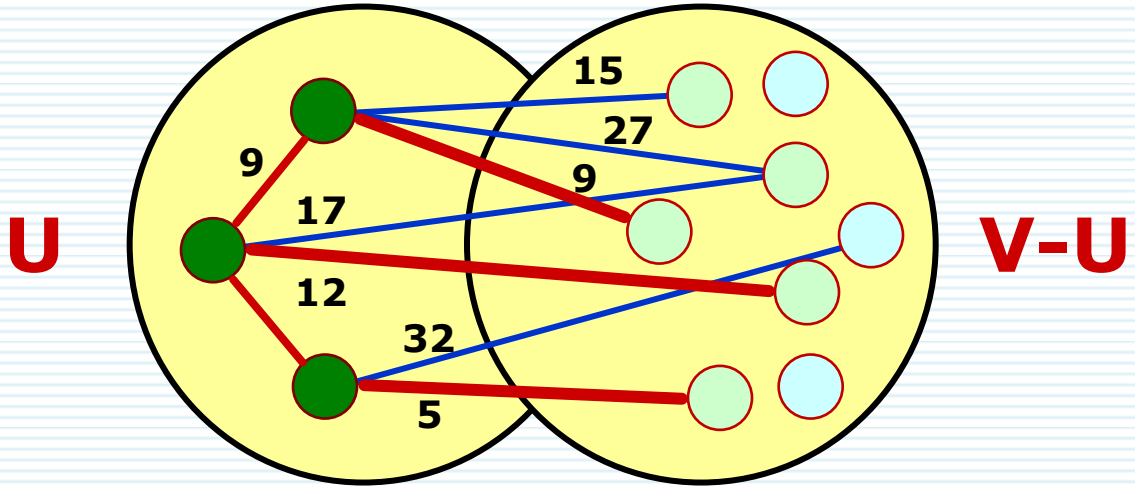
2. 在所有满足 u∈U，v∈V-U 的边 (u,v)∈E 中

   - 找到一个具有最小值的边加入生成树T中
   - 并将与之邻接的顶点 v 加入集合 U

3. 若全部n个顶点均已加入到U集合中，则算法结束

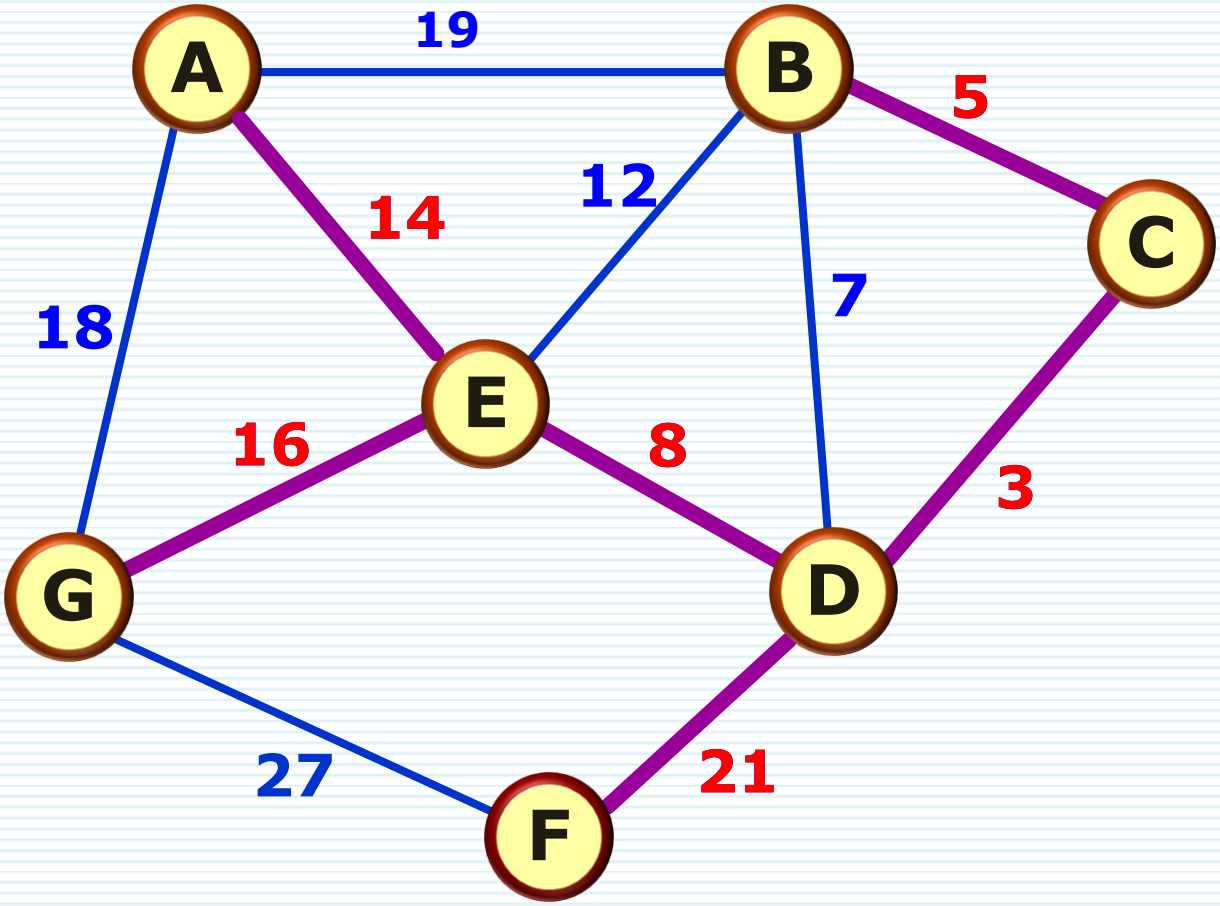
   - 否则继续执行第2步

Prim算法



- Prim算法设计思想
  - 在生成树的构造过程中，图中 n 个顶点分属两个集合：
    - 已加入到生成树中的顶点集：U
    - 尚未加入到生成树中的顶点集：V-U
  - 在所有连通U和V-U的边中选取权值最小的边加入MST中

Prim算法

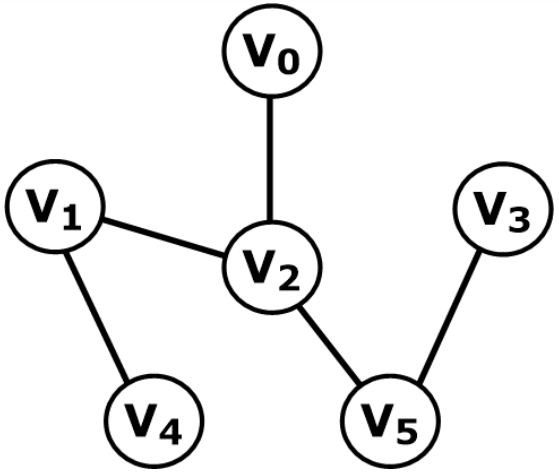


生成树代价 = 14+8+3+5+16+21 = 67

Prim算法的设计与实现

- 设： N =（ V, E ）是一个连通网
  - 设置：辅助集合V = { 1，2，…，n } 是 N 的顶点集合
  - 设置：辅助集合U，初值为{u_0}
    - 用来存放当前所得到的最小生成树的顶点集合
  - 设置：辅助集合 edges，初值为{}
    - 用来存放当前所得到的最小生成树的边
- 思考：应为edges设计怎样的数据结构?
  - 提示：edges中的元素与U和V-U有何关联?
    - 以U和V-U中顶点间的邻接关系为候选对象
    - 每次从中选出代价最小的边加入edges集合
    - 同时将与之关联的顶点从V-U中移到U中

Prim算法的设计与实现



cost	0	0	0	0	0	0
vex	0	2	0	5	1	2
下标	0	1	2	3	4	5

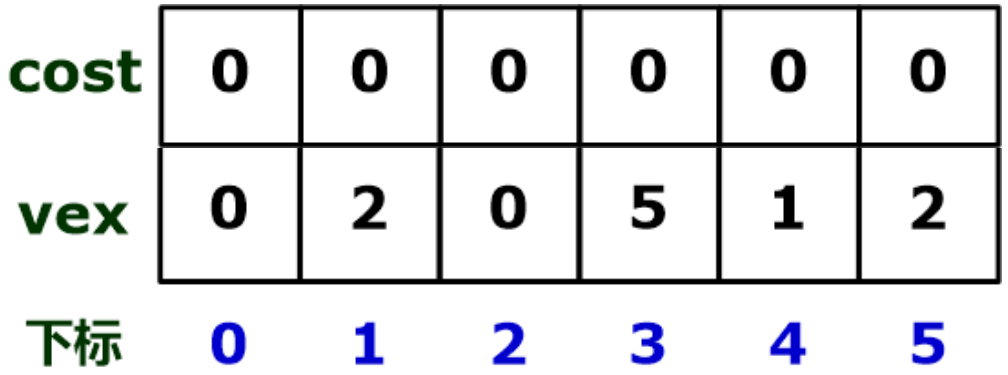
edges 数组

- 候选对象：连接集合U和V-U的所有边 **怎样表示?**
  - edges 数组元素成员 vex 表示集合 U 中的顶点
    - **(i, edges[i])** 表示图中的一条边
  - edges 数组元素成员 cost 表示候选边的权重
    - cost == 0 表示： i ∈ U **边(i, edges[i]) ∈MST**
    - cost != 0 表示： i ∈ V-U

      边(i, edges[i]) 是 i 到 U 中当前各顶点的权值最小边

Prim算法的设计与实现

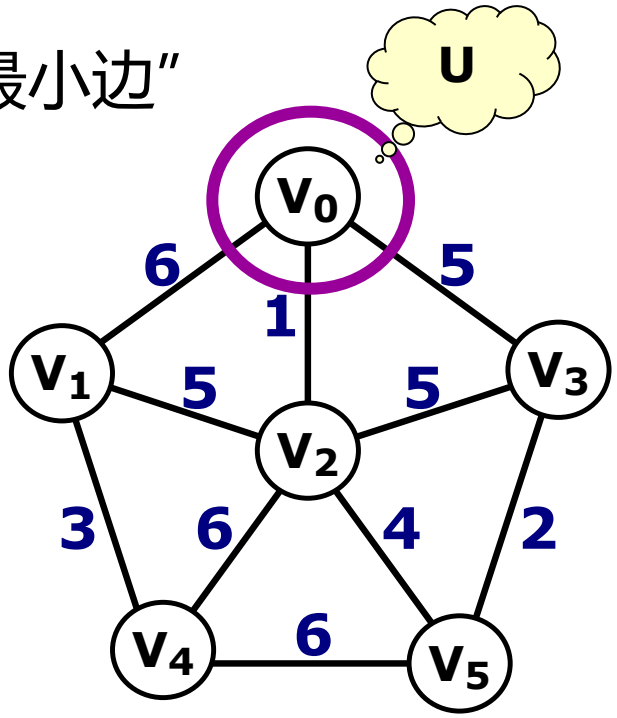
```
typedef struct {
    int vex;
    int cost;
}Edge;
```



- 针对候选对象设计数据结构
  - 设置一个edges数组
  - 数组元素的类型为结构体（Edge）
    - edges[i].vex： U中与i相邻的边权重最小的顶点编号
    - edges[i].cost：边（i，edges[i].vex）的权重
  - 数组长度为n (图中顶点总数)

对i∈V-U，edges数组中的元素含义

- edges[i].vex = k (k∈U)
  - 边 (i, k) 是 i 到 U 中各顶点的“权最小边”
- edges[i].cost
  - 存放 i 到 U 中当前各顶点的最小权重




```
for(v=1; v < n; ++v){  // V0加入U
        edges[v].vex=0;
        edges[v].cost = G[0][v];
}
```

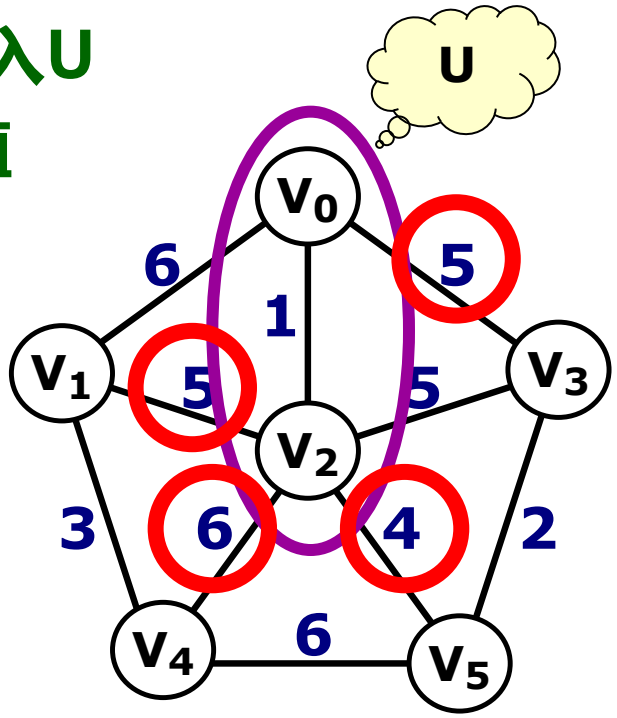
U={V_0} V-U={ V_1, V_2, V_3, V_4, V_5 }

i	0	1	2	3	4	5
vex		0	0	0	0	0
cost	0	6	1	5	∞	∞

$$\begin{bmatrix} 0 & 6 & 1 & 5 & \infty & \infty \\ 6 & 0 & 5 & \infty & 3 & \infty \\ 1 & 5 & 0 & 5 & 6 & 4 \\ 5 & \infty & 5 & 0 & \infty & 2 \\ \infty & 3 & 6 & \infty & 0 & 6 \\ \infty & \infty & 4 & 2 & 6 & 0 \end{bmatrix}$$

```
for( a=1; a< n; ++a ) {   // 逐一选择顶点加入U
    k = select(edges);    // 选出当前的最小权边
    edges[k].cost=0;      // 对应顶点加入U
    for(i=1; i<n; i++) {  // 更新cost值
        if( G[k][i]<edges[i].cost ) {
            edges[i].cost = G[k][i];
            edges[i].vex = k;
        }
    }
}
```

k = 2

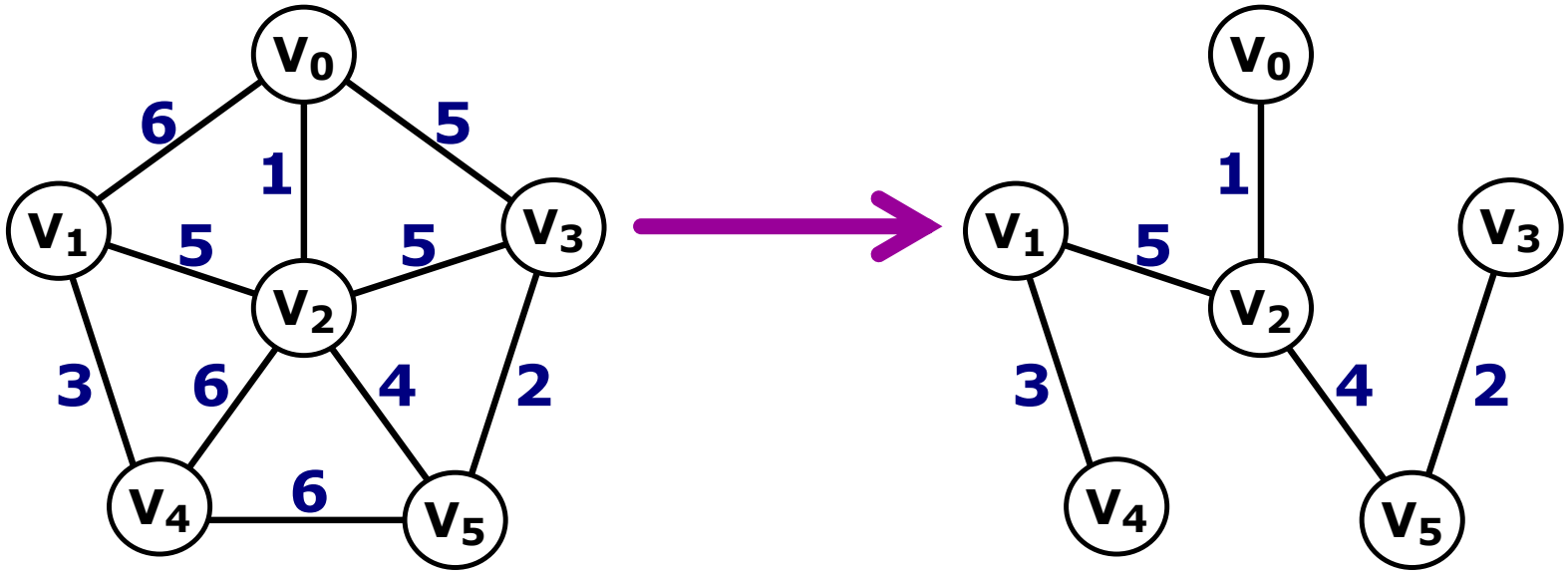


U={v_0, v_2} V-U={ v_1, v_3, v_4, v_5 }

i	0	1	2	3	4	5
vex		2	0	0	2	2
cost	0	5	0	5	6	4

$$\begin{bmatrix} 0 & 6 & 1 & 5 & \infty & \infty \\ 6 & 0 & 5 & \infty & 3 & \infty \\ 1 & 5 & 0 & 5 & 6 & 4 \\ 5 & \infty & 5 & 0 & \infty & 2 \\ \infty & 3 & 6 & \infty & 0 & 6 \\ \infty & \infty & 4 & 2 & 6 & 0 \end{bmatrix}$$

Prim算法：从序号为0的顶点出发，构造有n个顶点的网G的最小生成树MST，并输出MST的各条边，G采用邻接矩阵存储



i	0	1	2	3	4	5
vex	0	2	0	5	1	2
cost	0	0	0	0	0	0

边的加入顺序：(1,3) (3,6) (4,6) (2,3) (2,5)

```
void prim_mst(int G[][N], Edge *edges) {
   int a, i, j, k;  float min;
   for(i = 1; i < N; i++){ // 初始化
      edges[i].vex=0; edges[i].cost = G[0][i];
   }
   edges[0].cost=0; edges[0].vex=0;  // 将顶点0加入U
   for( a = 1; a < N; ++a ){
      k = select(edges); // 从候选边中选择权值最小的顶点
      printf("(%i, %i)\n", edges[k].vex, k );
      edges[k].cost=0;  // 将顶点k并入集合U
      for( i = 1; i < N; ++i)  {   // 更新候选边数组
         if( G[k][i]<edges[i].cost ) {
            edges[i].cost = G[k][i]; edges[i].vex = k;
         }
      }
   }
}
```

算法的时间复杂度？ $O(n^2)$

适用于边稠密的图

Prim算法：从候选边中选择权值最小的顶点

```
int select(Edge edges[]) {
    int min = INT_MAX,  idx = 1; int i;
    for( i = 1; i < N; ++i ){
        if ( edges[i].cost != 0 && edges[i].cost < min){
            min = edges[i].cost;
            idx = i;
        }
    }
    return idx;
}
```

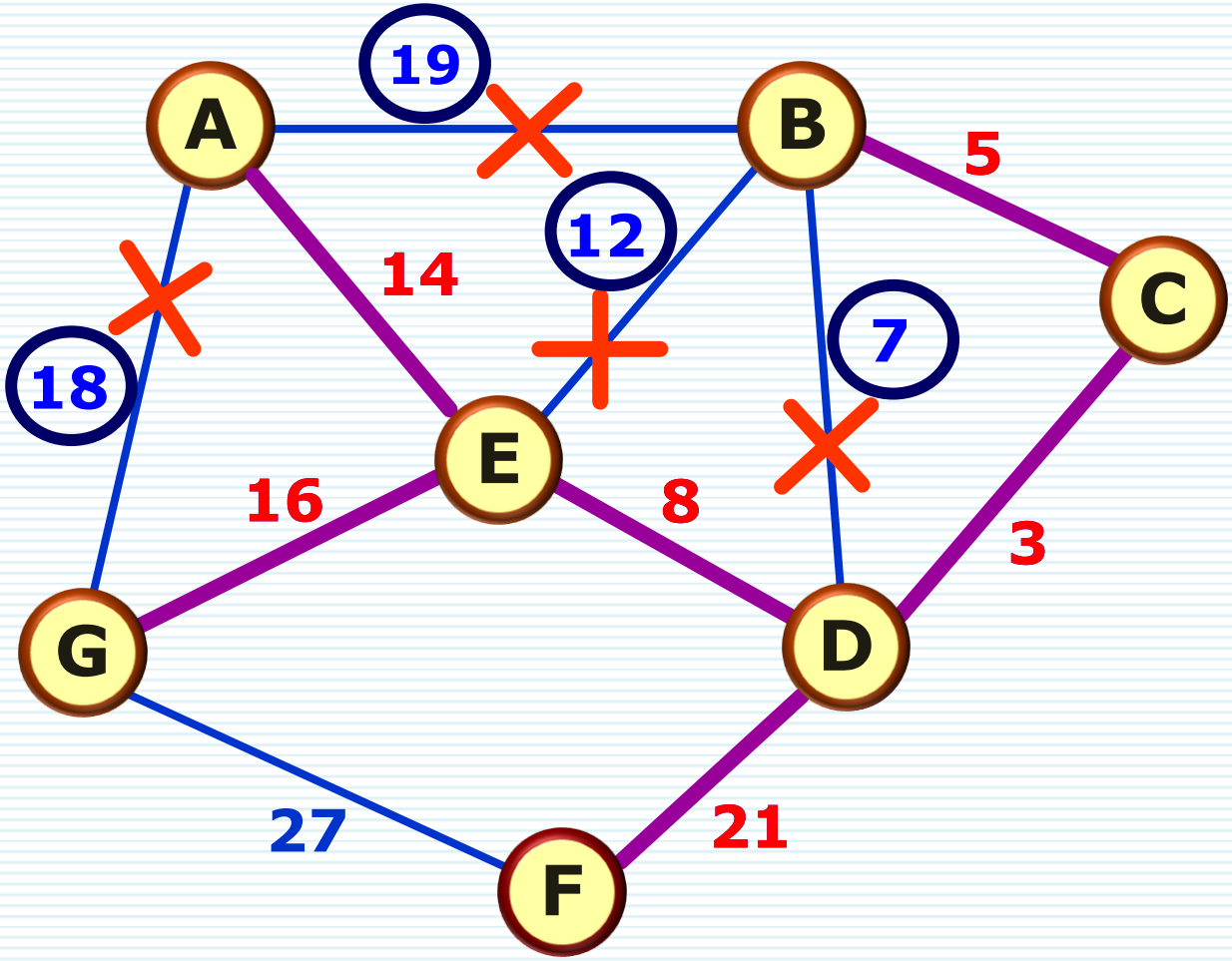
cost	0	6	1	5	∞	∞
vex	0	2	0	5	1	2
下标	0	1	2	3	4	5

Kruskal算法

Kruskal算法

- Kruskal算法设计思想
  - 逐步向森林T中添加不与T中的边构成回路的当前最小代价边
  - 算法特点：以最小代价边主
- Kruskal算法思路
  - 先构造一个只含 n 个顶点的子图 T
  - 然后从权值最小的边 e_i 开始考察
    - 若添加 e_i 不使T中产生回路，则在T中加上这条边
  - 如此重复，直至加上 n-1 条边为止

Kruskal算法



提示：顶点u和顶点v来自不同的连通分量　　思考：怎样判断环路？

方案：为每个连通分量设置一个编号　　生成树代价 = 67

Kruskal算法

- Kruskal算法流程
  - 设 G ={V，{E}}为给定的连通网
  - 将生成树T的初始状态置为 T={V，{Φ}}
  - 当T中边数小于n-1时，重复下列步骤：
    - 从E中选取代价最小的边（v, u）
    - 若顶点v和u落在T中不同的连同分量上
      - 则：将其加入生成树T中，并从E中将其删除
    - 否则：从E中将其删除，选择下一条代价最小的边

Kruskal算法的数据结构设计

```
typedef   struct {     // 顶点结点
    int  vex;          // 顶点信息
    int  gno;          // 顶点所在的连通分量编号
}TVex;
```

```
typedef   struct {      // 边结点
    int  vh, vt;        // 边依附的两顶点
    int  cost;          // 边的权值
    int  flag;      // 0：未加入MST；1：已入选；  -1：已删除
}TEdge;
```

Kruskal算法的详细设计

- 用顶点数组和边数组存放顶点和边信息
- 初始化：令每个顶点的gno互不相同，每个边的flag为0
- 选出权值最小且未加入MST的边（u, v）
  - 若（u, v）依附的两个顶点的gno值相同，则：
    - 令（u, v）的flag=-1 （表示舍去该边）
  - 若该边依附的两个顶点的gno值不同，则：
    - 令（u, v）的flag=1（表示选中该边加入MST）
    - 令 v.gno = u.gno
      - 并修改v所在集合中所有顶点的gno为u.gno
- 重复上述步骤，直到选出n-1条边为止

Kruskal算法的数据结构设计

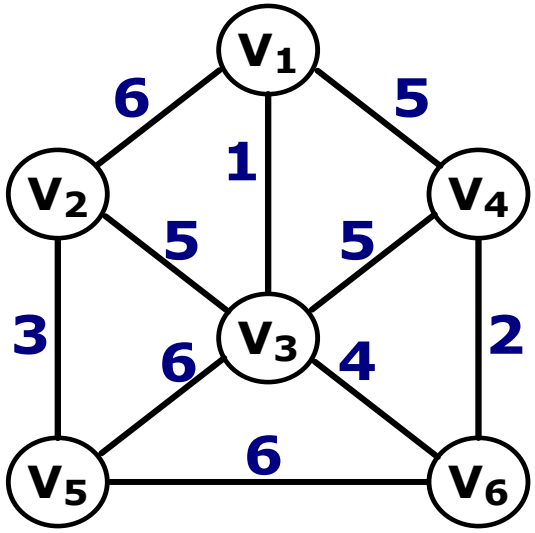
```
typedef   struct {
      int  vex;
      int  gno;
}TVex;
```

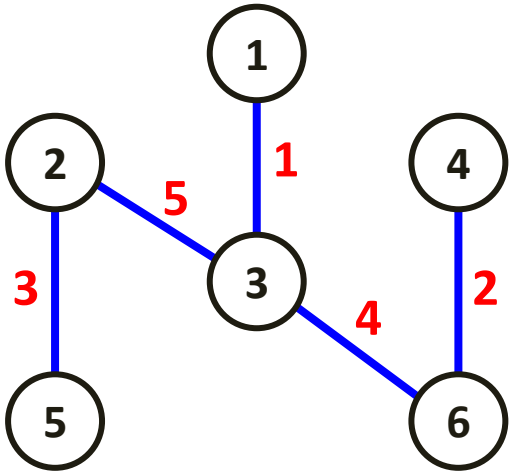
```
typedef   struct {
      int  vh, vt;
      int  cost;
      int  flag;
}TEdge;
```

- 图的存储方式

```
typedef struct{        // 图结构定义
      TVex * pv;       // 顶点数组
      TEdge *pe;       // 边数组
      int nv, ne;
}TGraph, * PGraph;
```

Kruskal算法的实现





顶点数组

	vex	gno
0	1	2
1	2	2
2	3	3
3	4	4
4	5	2
5	6	4

边数组

	vhead	vtail	cost	flag
0	1	2	6	0
1	1	3	1	1
2	1	4	5	-1
3	2	3	5	1
4	2	5	3	1
5	3	4	5	0
6	3	5	6	0
7	3	6	4	1
8	4	6	2	1
9	5	6	6	0

Kruskal算法

```
void kruskal_mst(TGraph *G) {
   int i, min, idx, m, n, g, count = 1;
   while( count < G->nv ) {  // 逐一加入 n-1 条边
      min = INT_MAX;
      for(i = 0; i < G->ne; ++i){   // 选取未访问过的最小权边
         if( G->pe[i].cost < min && G->pe[i].flag == 0){
            min = G->pe[i].cost; idx = i; } }
      m = G->pe[idx].vh; n = G->pe[idx].vt;
      if ( G->pv[m].gno != G->pv[n].gno){
            G->pe[idx].flag = 1;  count++; g = G->pv[n].gno;
            for(i = 1; i <= G->nv; ++i){
               if(G->pv[i].gno == g)
                  G->pv[i].gno = G->pv[m].gno;
            } }
       else G->pe[idx].flag = -1;
   }}
```

改进：对边按权重进行排序 O(eloge)

算法的时间复杂度? O(nv×ne)

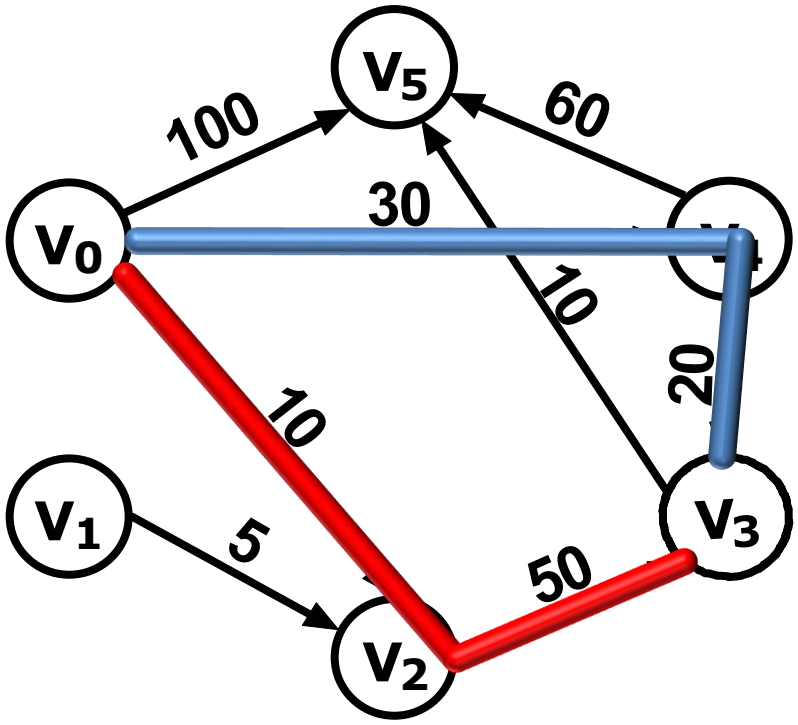
普里姆和克鲁斯卡尔最小生成树算法比较

普里姆最小生成树算法	克鲁斯卡尔最小生成树算法
以连通为主	以最小代价边为主
选保证连通的代价最小的邻接边	选不形成回路的当前最小代价边
算法时间复杂度：$O(n^2)$	算法时间复杂度：$O(e\log_2 e)$
算法时间复杂度与边无关	算法时间复杂度与边相关
适合于求边稠密网的最小生成树	适合于求边稀疏网的最小生成树

5. 最短路径

最短路径（Shortest Paths）

- 在有向图中，寻找从某个源点到其余各个顶点或者每一对顶点之间的最短带权路径的运算，称为最短路径问题
- 单源点最短路径问题
  - 给定：带权有向图G和源点v
  - 求解：从源点v到G中其余各顶点之间的最短路径
- 求所有顶点对之间的最短路径
  - 给定：带权有向图G
  - 求解：G中各顶点对之间的最短路



v_0 到各顶点的最短路径

源点	终点	最短路径	路径长度
v_0	v_1	----	—
	v_2	(v_0,v_2)	10
	v_3	(v_0,v_4,v_3)	50
	v_4	(v_0,v_4)	30
	v_5	(v_0,v_4,v_3,v_5)	60

例如：在带权有向图G中求出v_0到其余各顶点之间的最短路径

- 从图中可见：从v_0到v_1没有路径
- 从v_0到v_3有两条不同的路径：(v_0, v_2, v_3)和(v_0, v_4, v_3)
  - 前者长度为60，而后者长度为50
- 因此后者是从v_0到v_3的最短路径

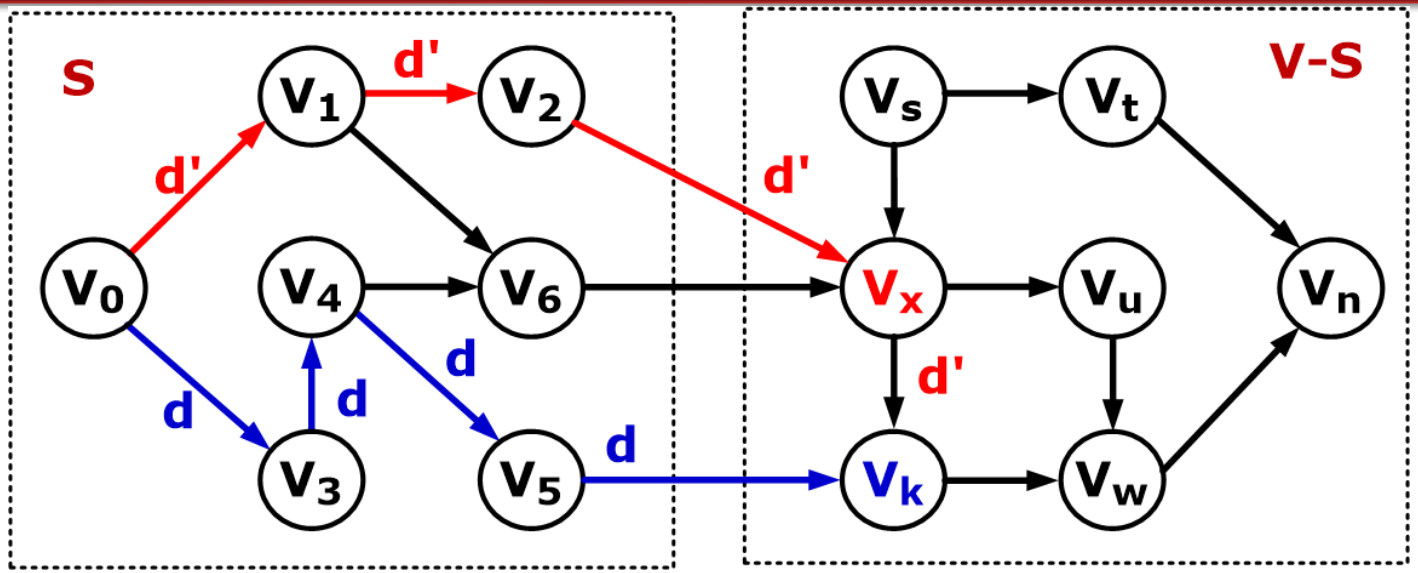
迪杰斯特拉（Dijkstra）算法

- Dijkstra算法是求解**单源最短路径问题**的一种有效算法
- 算法基本思路
  - 按路径长度递增的次序产生到各顶点的最短路径
  - 设置顶点集合**S**={v_0}并不断地做贪心选择来扩充S
- 贪心选择策略
  - 顶点v_k 属于**S**当且仅当从源到v_k 的最短路径长度已知
  - 可以证明：v_0 到 T=V-S 中顶点v_k的最短路径
    - 或者是从v_0 到v_k 的直接路径的权值
    - 或者是从v_0经**S**中顶点到v_k的路径权值之和

迪杰斯特拉（Dijkstra）算法

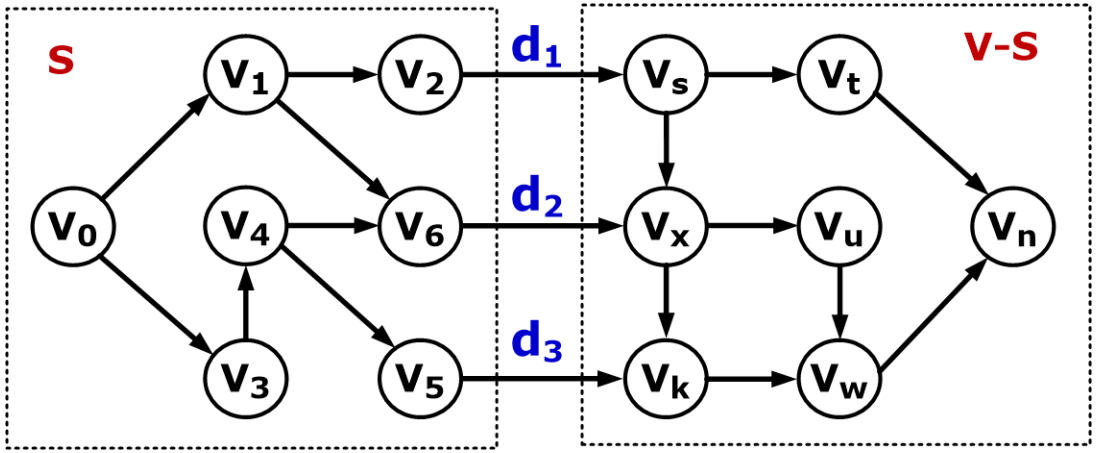
- 把图G的顶点V分成两组
  - **S**：已求出最短路径(SP)的顶点的集合
  - **T**=V-S：尚未确定最短路径的顶点集合
- 将T中顶点按最短路径递增的次序加入到S中**并确保**
  - 从v_0到S中任意顶点的SP ≤ 从V_0到T中任意顶点的SP
- 定义：从源v_0到顶点v_k的**特殊路径**
  - 从源v_0到v_k并且中间只经过S中顶点的路径
  - 由此：每个顶点对应一个距离值
    - S中顶点：从v_0到此顶点的最短路径长度
    - T中顶点：从v_0到此顶点的最短特殊路径长度

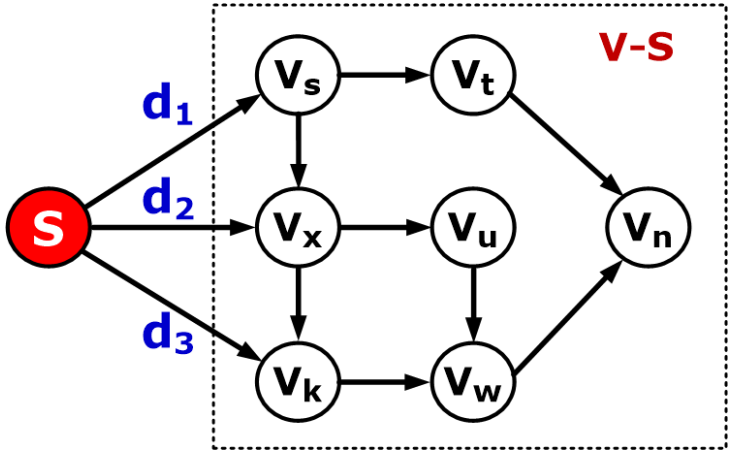
Dijkstra算法的贪心选择策略的有效性



- 贪心选择依据：v_0 到 T=V-S 中顶点v_k的最短路径（**d**）
  - 或者是从v_0 到v_k 的直接路径的权值
  - 或者是从v_0 经S中顶点到v_k的路径权值之和
- 问题：贪心选择的依据是否正确？
  - 反证：假设从v_0 到v_k 的最短路径d′经过T中的顶点v_x

Dijkstra算法的贪心选择策略的有效性



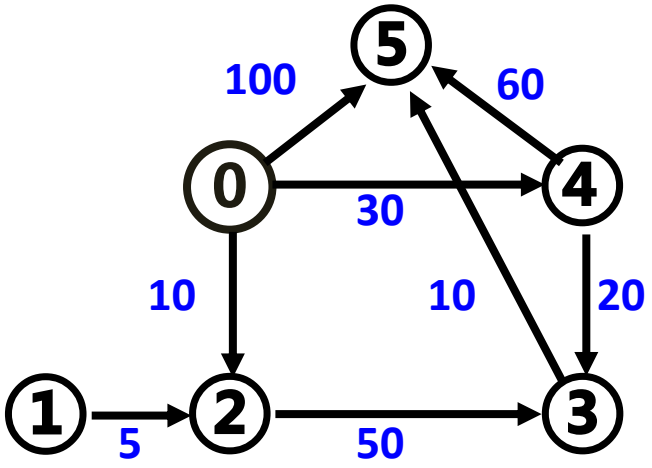


- 贪心选择策略：
  - 按路径长度递增的次序产生到各顶点的最短路径
- 问题：这种策略能否保证得到全局最优解？
  - 设图中标出的路径长度为从v_0到相应顶点的特殊路径长度
  - 不妨设：$d_1 \le d_2 \le d_3$
  - 则：从v_0出发到V-S的任意顶点的最短路径长度 $\ge d_1$

Dijkstra算法流程

1. 初始化
   - 令： S={v_0}，T={其余顶点}
   - T中顶点 v_i 与v_0的距离值 D_i 定义为
     - 若存在<v_0, v_i>： D_i 为弧<v_0, v_i>上的权值
     - 若不存在<v_0, v_i>： D_i 为∞
2. 从T=V-S中选取一个与v_0的距离值最小的顶点v_w加入S
   - 同时更新T中顶点的距离值：若增加v_w作中间顶点之后
     - 从v_0 经 v_w 到 v_i 的距离值比之前的特殊路径短
     - 则更新 v_i 距离值（为较小的值）
3. 重复上述步骤，直到S中包含所有顶点（即S=V）为止

例子



	0	1	2	3	4	5
0	∞	∞	10	∞	30	100
1	∞	∞	5	∞	∞	∞
2	∞	∞	∞	50	∞	∞
3	∞	∞	∞	∞	∞	10
4	∞	∞	∞	20	∞	60
5	∞	∞	∞	∞	∞	∞

终点	从v_0到各终点的最短路径和路径长度值（dist）				
v_1	∞	∞	∞	∞	∞
v_2	10 (v_0,v_2)	X	X	X	X
v_3	∞	60(v_0,v_2,v_3)	50(v_0,v_4,v_3)	X	X
v_4	30 (v_0,v_4)	30(v_0,v_4)	X	X	X
v_5	100(v_0,v_5)	100(v_0,v_5)	90(v_0,v_4,v_5)	60(v_0,v_4,v_3,v_5)	X
v_i	v_2	v_4	v_3	v_5	

Dijkstra算法流程

- Dijkstra算法的数据结构设计
  - 使用带权邻接矩阵表示有向图G
  - 数组**S[n]**：顶点集合 (n 为图中顶点数)
    - 记录已找到从v_0出发的最短路径的顶点
  - 数组**dist[n]**
    - 存放各顶点距离v_0的当前最短路径长度
  - 辅助数组：**path[n]**(存储最短路径)
    - path[i]表示从v_0到v_i的SP上，v_i 的前序顶点的序号
    - 若从 v_0 到某顶点 v_i 无路径，则path[i]=-1

Dijkstra算法的存储结构

```
#define N 6                    // 图中顶点总数

typedef  struct {

     int  vex[N];              // 顶点数组

     int  arc[N][N];           // 邻接矩阵

}TGraph;
```

Dijkstra算法

```
// 求有向网G的v0顶点到其余顶点的最短路径

void dijkstra(TGraph G, int v0, int path

    int S[N] = {0};   S[v0] = 1;    // 将v

    for(int i = 0; i < NV; i++) {

        dist[i] = G.arc[v0][i]; // v0到其他顶点的当前最短距离

        if( dist[i] < INT_MAX ) path[i] = v0;  // 记录前驱

        else path[i] = -1;

    }
```

```
typedef  struct {
    int  vex[N];
    int  arc[N][N];
}TGraph;
```

Dijkstra算法

```
for(int  i = 0; i < NV; ++i) {
    if( i != v0 ){
        int min = INT_MAX,  v = -1;     // 临时变量（记录当前最小）
        for( int k = 0; k < N; k++){    // 找出最小的dist[k]
            if( S[k]==0 && dist[k] < min) {
                v = k;  min = dist[k];
            }
        }
        if(v == -1) break;    // 已无顶点可加入S中
        S[v] = 1;             // 将顶点v并入集合S
        for(int k = 0; k < N; k++){
            if(S[k]==0 && (min + G.arc[v][k]) < dist[k] {
                dist[k] = min + G.arc[v][k];   path[k] = v;
            }
        }
    }
}}
```

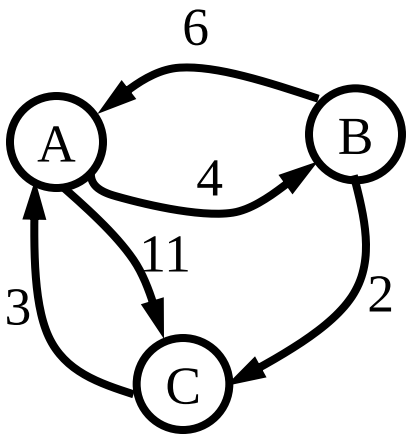
单源最短路径

- 算法复杂性分析
  - 对于有n个顶点和e条边的带权有向图G
    - 采用带权邻接矩阵表示图G
    - 在dist[]数组中查找最小值需时：O(n)
    - 需对n-1的顶点执行上述操作
    - 算法的其余部分需时不超过$O(n^2)$
  - Dijkstra算法的复杂度为：$O(n^2)$
- 算法改进?
  - 采用基于堆的优先队列　请自己实现

求每一对顶点之间的最短路径：Floyd算法

- 方法一：每次以一个顶点为源点
  - 重复执行Dijkstra算法n次：$T(n)=O(n^3)$
- 方法二：Floyd算法（逐个顶点试探法）
  - 从任意顶点A到任意顶点B的最短路径只有两种可能：
    - SP(A,B)为：从A到B的直接路径
    - SP(A,B)为：从A开始经过若干个中间节点 K_i 到B
  - 对于后者可采用逐一试探的方法
    - 对于每一个顶点 K （K≠A 且 K≠B）
    - 若有：Dist(A,K) + Dist(K,B) < Dist(A,B)
    - 则令：Dist(A,B) = Dist(A,K) + Dist(K,B)

例



初始状态：$\begin{pmatrix} 0 & 4 & 11 \\ 6 & 0 & 2 \\ 3 & \infty & 0 \end{pmatrix}$ 路径：

	AB	AC
BA		BC
CA		

在B、C之间加入A：$\begin{pmatrix} 0 & 4 & 11 \\ 6 & 0 & 2 \\ 3 & 7 & 0 \end{pmatrix}$ 路径：

	AB	AC
BA		BC
CA	CAB	

在A、C之间加入B：$\begin{pmatrix} 0 & 4 & 6 \\ 6 & 0 & 2 \\ 3 & 7 & 0 \end{pmatrix}$ 路径：

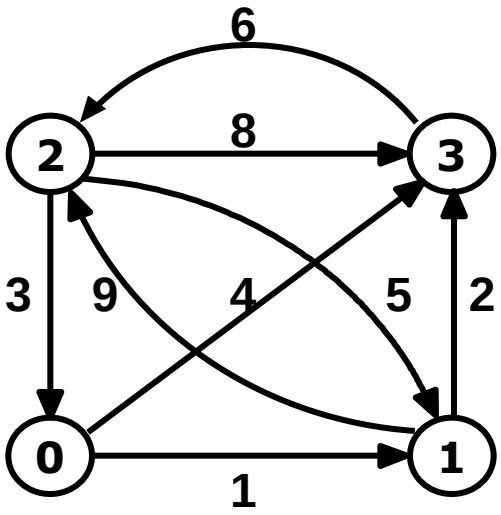
	AB	ABC
BA		BC
CA	CAB	

在A、B之间加入C：$\begin{pmatrix} 0 & 4 & 6 \\ 5 & 0 & 2 \\ 3 & 7 & 0 \end{pmatrix}$ 路径：

	AB	ABC
BCA		BC
CA	CAB	

Floyd算法流程

- 初始时设置一个n阶方阵Dist，令其对角线元素为0
  - 若存在弧<v_i, v_j>，则对应元素为权值；否则为∞
- 逐步尝试向顶点对（A,B）的当前最短路径中增加中间顶点
  - 若有：Dist(A,K) + Dist(K,B) < Dist(A,B)
    - 则令：Dist(A,B) = Dist(A,K) + Dist(K,B)
    - 记录：Path(A,B) = K（表示SP="A->K->B"）
  - 否则：维持Dist(A,K) 和Path(A,B) 的值不变
- 当所有顶点均试探完毕，则算法结束，此时：
  - Dist(AB)记录顶点A到B的最短路径的长度
  - Path(AB)保存顶点A到B的最短路径经过的顶点信息



	0	1	2	3
0	0	1	∞	4
1	∞	0	9	2
2	3	5	0	8
3	∞	∞	6	0

	$Dist^{(-1)}$				$Dist^{(0)}$				$Dist^{(1)}$				$Dist^{(2)}$				$Dist^{(3)}$			
	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3
0	0	1	∞	4	0	1	∞	4	0	1	10	3	0	1	10	3	0	1	9	3
1	∞	0	9	2	∞	0	9	2	∞	0	9	2	12	0	9	2	11	0	8	2
2	3	5	0	8	3	4	0	7	3	4	0	6	3	4	0	6	3	4	0	6
3	∞	∞	6	0	∞	∞	6	0	∞	∞	6	0	9	10	6	0	9	10	6	0

	$Path^{(-1)}$				$Path^{(0)}$				$Path^{(1)}$				$Path^{(2)}$				$Path^{(3)}$			
	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3
0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	1	1	0	0	3	1
1	1	1	1	1	1	1	1	1	1	1	1	1	2	1	1	1	3	1	3	1
2	2	2	2	2	2	0	2	0	2	0	2	1	2	0	2	1	2	0	2	1
3	3	3	3	3	3	3	3	3	3	3	3	3	2	2	3	3	2	2	3	3

6. 有向无环图的应用

有向无环图的应用

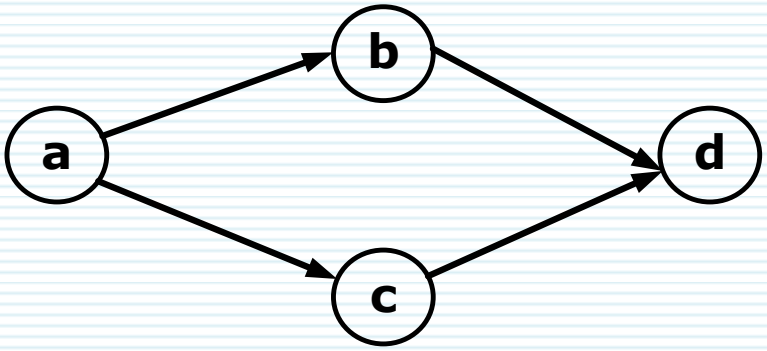
- 有向无环图(Directed Acycline Graph, DAG)
  - 是描述一项工程的进行过程的有效工具
  - 其应用主要有两方面
    - 进行拓扑排序
    - 求解关键路径

拓扑排序

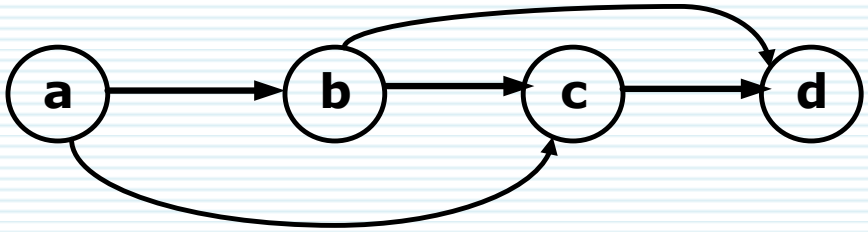
拓扑排序

- 拓扑排序
  - 根据某个集合上定义的一个偏序求出该集合上的全序
  - 简单地说就是将集合上的元素按某种关系进行排序
- 拓扑排序中关于偏序和全序的定义如下
  - 偏序：称**R**是集合X上的偏序
    - 若集合X上的关系**R**是自反的，反对称的和传递的
  - 全序：设R是集合X上的偏序关系
    - 如果对于每个 x，y∈X 必有 x**R**y 或 y**R**x
    - 则称**R**是集合X上的全序关系

有向无环图的应用



(a) 偏序

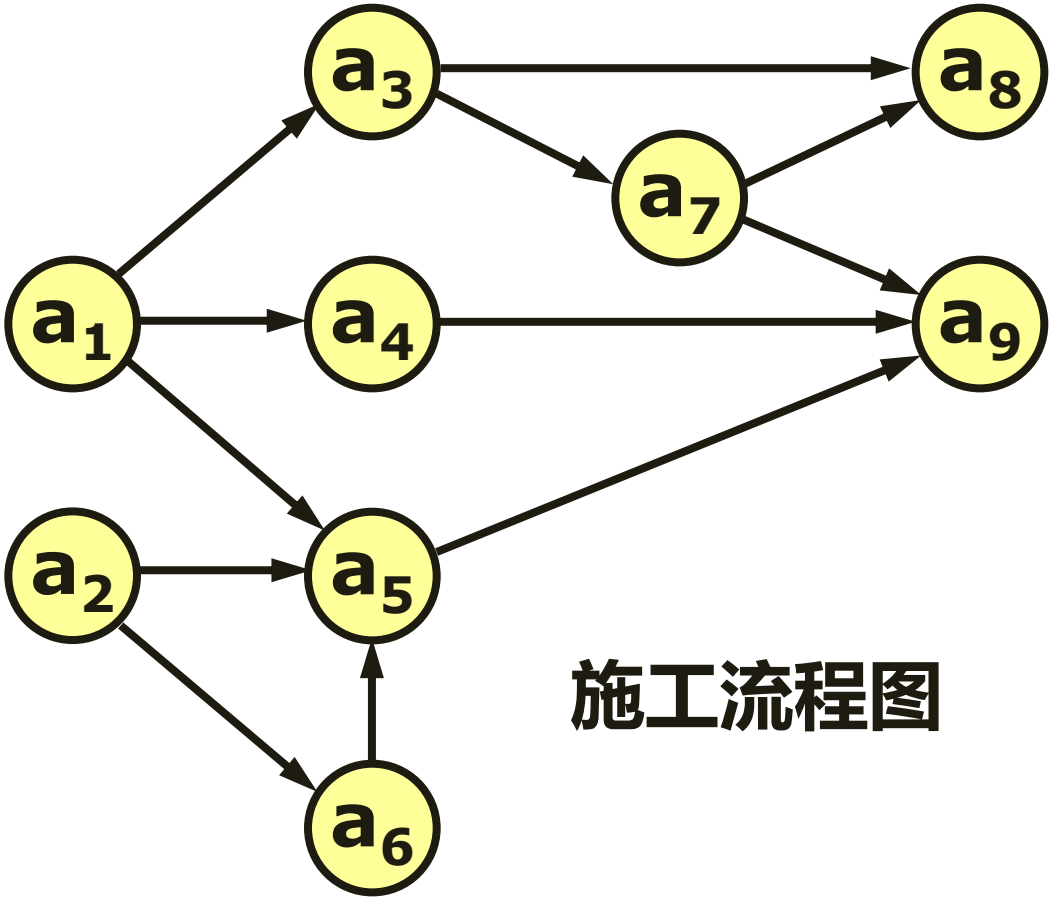


(b) 全序

- 拓扑有序
  - 若需对图（a）中的元素进行拓扑排序
  - 只需在b、c之间添加一条弧<b,c>
    - 集合{a,b,c,d}即变成全序关系
  - 则这个全序关系又称为拓扑有序

拓扑排序（topological sort）

- 拓扑排序
  - 是一种对非线性结构的有向图进行线性化的重要手段
- 问题的提出
  - 假设：以有向图表示一个工程的施工图
  - 要求：图中不允许出现回路
  - 问题：怎样检查有向图中是否存在回路?
    - 可以借助拓扑排序将其转化为线性问题



施工流程图

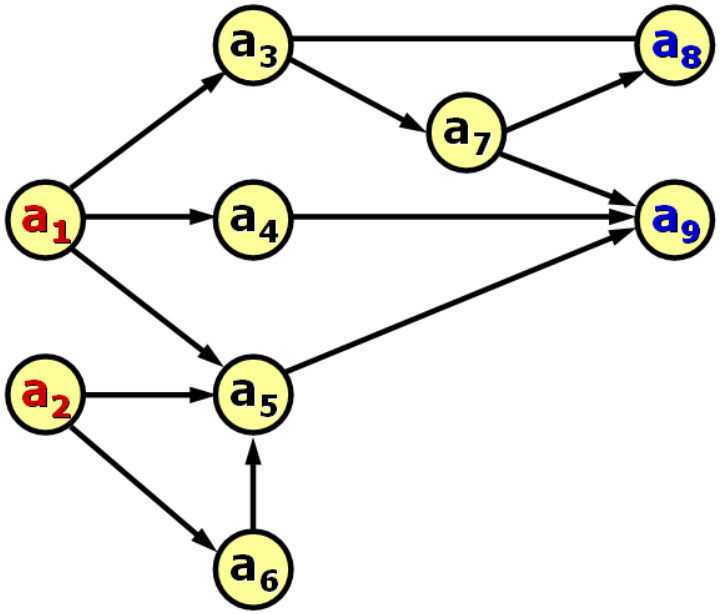
- 施工从活动 a_1、a_2开始，到达活动 a_8和 a_9时整个施工结束
- 有向图中：顶点表示活动，弧< a_i, a_j >表示活动 a_i优先于 a_j
- 称这类有向图为：顶点表示活动的网（AOV网）

AOV网（Activity On Vertex Network）

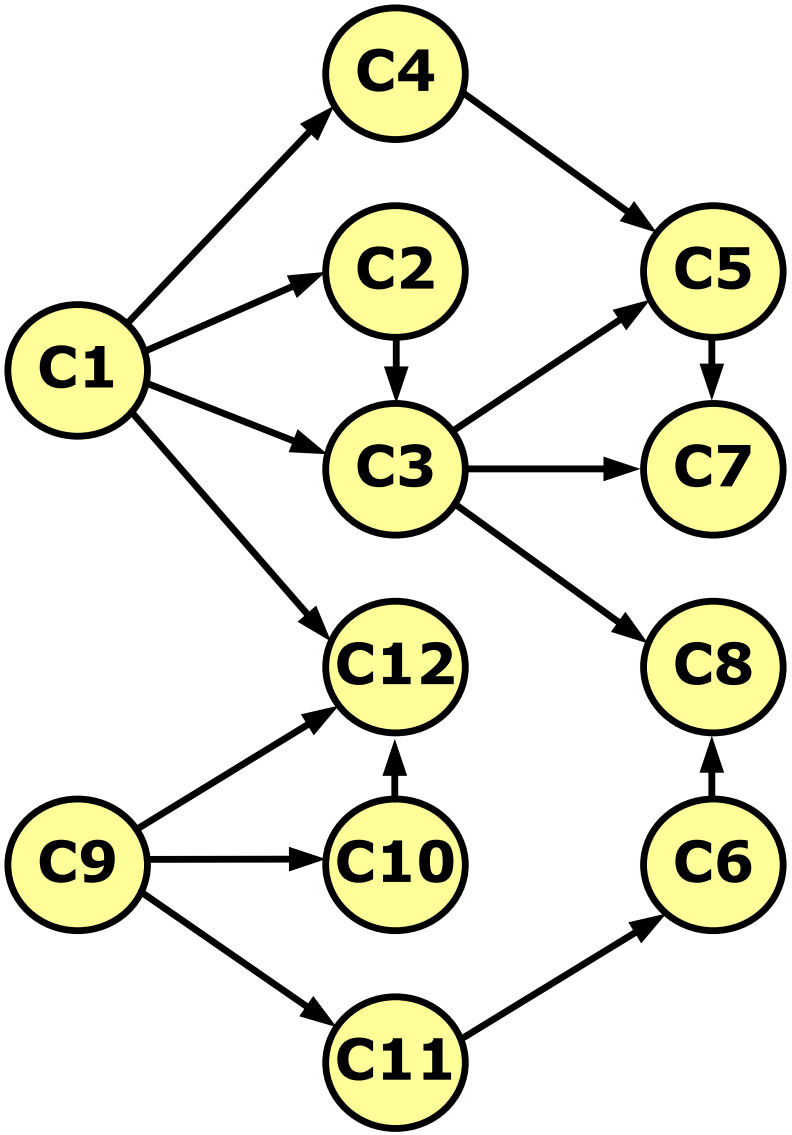
例如：右图中存在如下拓扑有序序列：

a1 a2 a3 a4 a6 a5 a7 a8 a9

a2 a6 a1 a3 a4 a5 a7 a9 a8

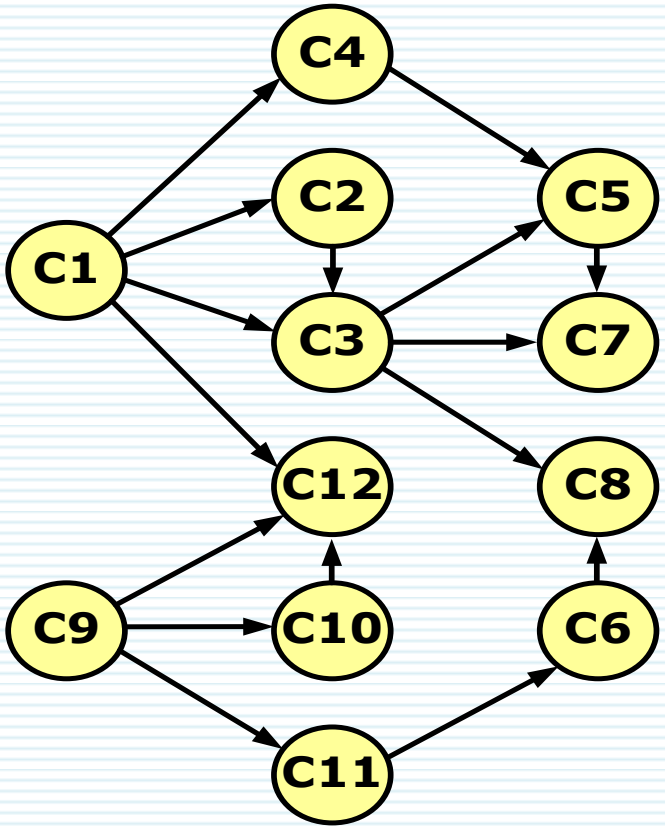


- AOV网：顶点表示活动的网
- AOV网可用于解决如下两类问题
  - 判定工程的可行性
    - 显然：若图中存在回路，则整个工程就无法结束
  - 确定各项活动在整个工程执行中的先后顺序
    - 称这种先后顺序为：拓扑有序序列



代码	课程名称	先修课程
C1	程序设计基础	无
C2	离散数学	C1
C3	数据结构与算法	C1，C2
C4	汇编语言	C1
C5	形式语言与自动机	C3，C4
C6	计算机原理	C11
C7	编译原理	C3，C5
C8	操作系统	C3，C6
C9	高等数学	无
C10	线性代数	C9
C11	普通物理	C9
C12	数值分析	C1,C9,C10

AOV网的拓扑序列不唯一



拓扑序列:

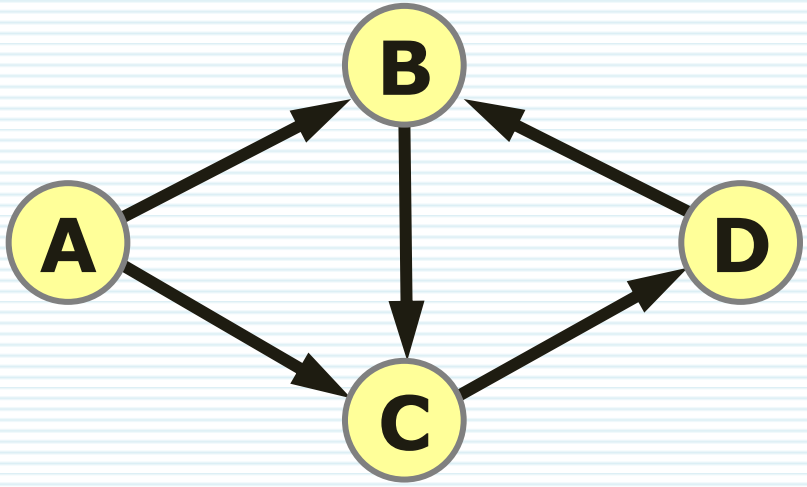
C1→C2→C3→C4→C5→C7→C9→C10→C11→C6→C12→C8

C9→C10→C11→C6→C1→C12→C4→C2→C3→C5→C7→C8

AOV网（Activity On Vertex Network）

问题：是否对所有的AOV网都能求得拓扑有序序列?

反例：对下述AOV网，无法求得它的拓扑有序序列



因为图中存在一个回路 {B, C, D}

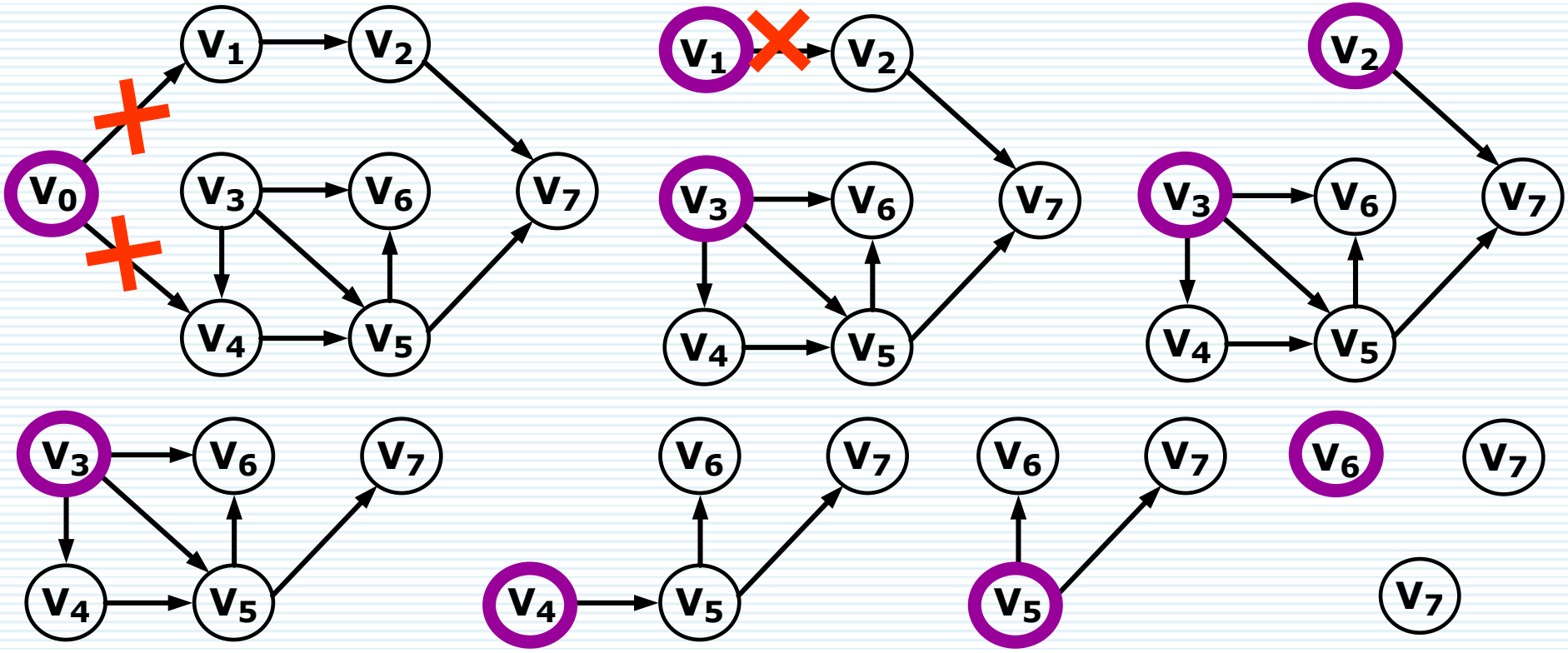
问题：怎样根据AOV网的特点对其进行拓扑排序?

AOV网的拓扑排序算法流程

- AOV网的拓扑排序算法流程

1. 在AOV网中选取一个没有前驱的顶点v开始遍历
2. 输出顶点v
3. 删除顶点v和所有以v为弧尾的弧
4. 重复1～3步，直到
   - AOV网中全部顶点都已输出（得到拓扑有序序列）
   - 或者图中再无没有前驱的顶点（AOV网中有环）

AOV网的拓扑排序算法示例



- 顶点 v_0 入度为0，选择输出v_0，并删除 v_0 及其所有出度边
- 入度为0的顶点是v_1 和v_3，输出v_1，并删除v1和它的所有出度边
- 依次选择v_2，v_3，v_4，v_5，v_6，v_7 进行输出
- 最后得到的拓扑序列为：v_0，v_1，v_2，v_3，v_4，v_5，v_6，v_7

AOV网的拓扑排序算法：数据结构设计

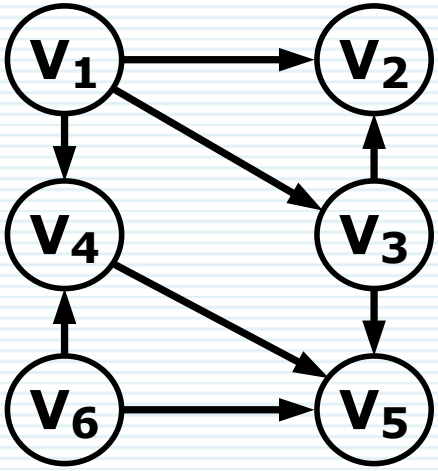
- 如何实现在AOV网中选取一个没有前驱的顶点进行输出？
  - 没有前驱的顶点即入度为0的顶点
- 如何删除该顶点和所有以它为弧尾的弧
  - 即：让该顶点的所有直接后继的入度减1
- 可见：拓扑排序算法的实现与顶点的入度有密切关系
  - 需求1：能比较方便地得到各顶点的入度
  - 需求2：易于寻找任一顶点的所有直接后继
- 因此：采用邻接表作为AOV网的存储结构
  - 在头结点中增加一个存放顶点入度的域

AOV网的拓扑排序算法：数据结构设计

邻接表结点：

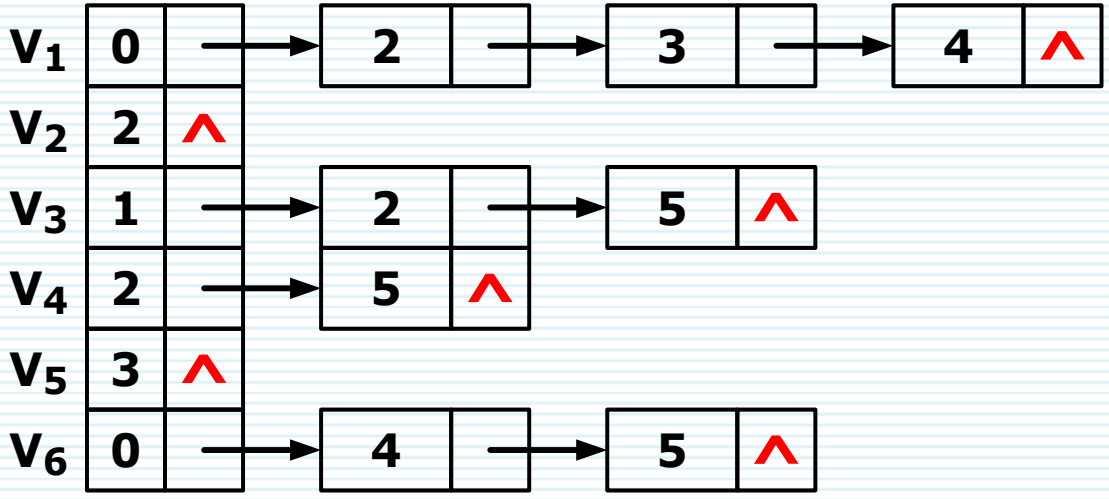
```
typedef  struct  node {
     int  adjvex;       // 邻接顶点域
     struct  node  *next;  // 链域
}ENode;
```

怎样求顶点入度?

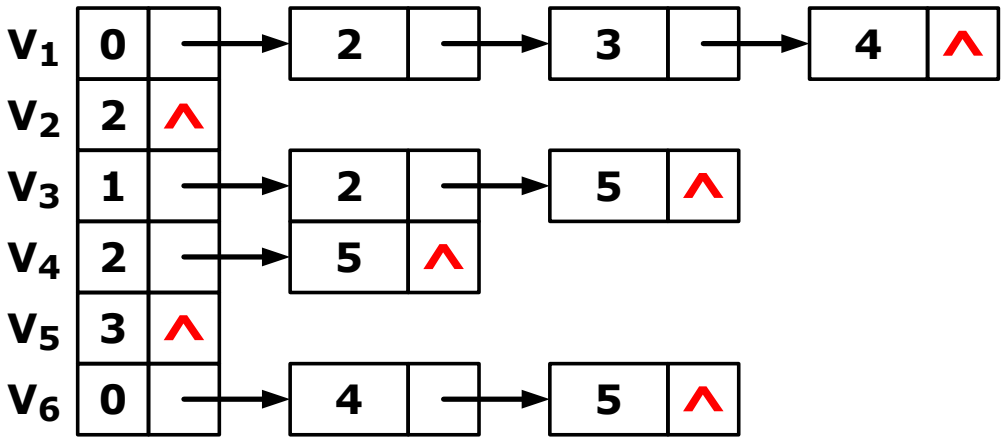


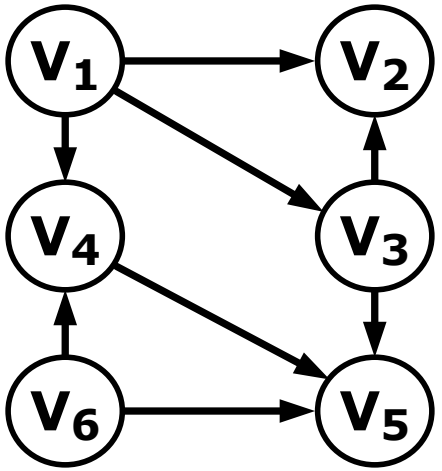
表头结点：

```
typedef  struct {
     int  indeg; // 入度
     TNode  *firstarc;
}VNode, *PGraph;
```



```
// 求出图中所有顶点的入度
void get_indeg (PGraph G, int *indegree, int n){
    int  i;   ENode  * p;
    for( i=0; i<n; ++i){
      indegree [i]=0;
    }
    for( i=0; i<n; ++i){
        p = G[i].firstarc;
        while(p){
            indegree [p->adjvex]++;
            p = p->next;
         }
    }
}
```





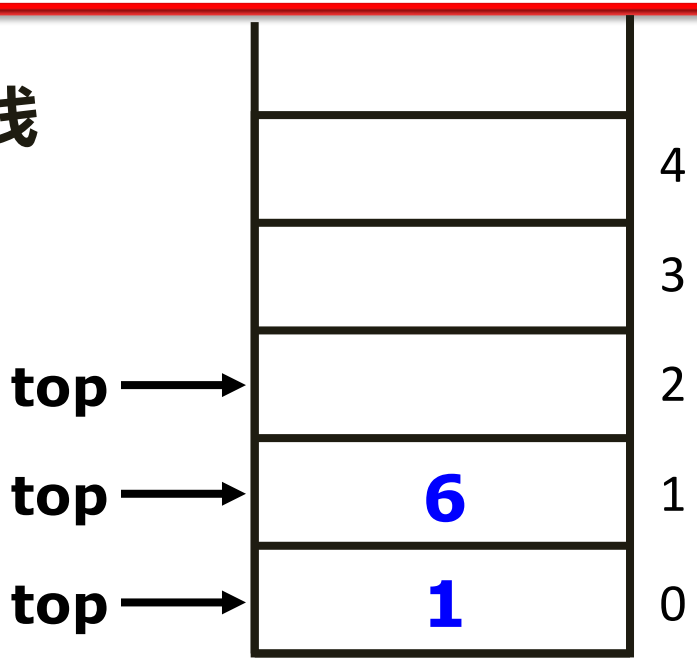
AOV网的拓扑排序算法流程

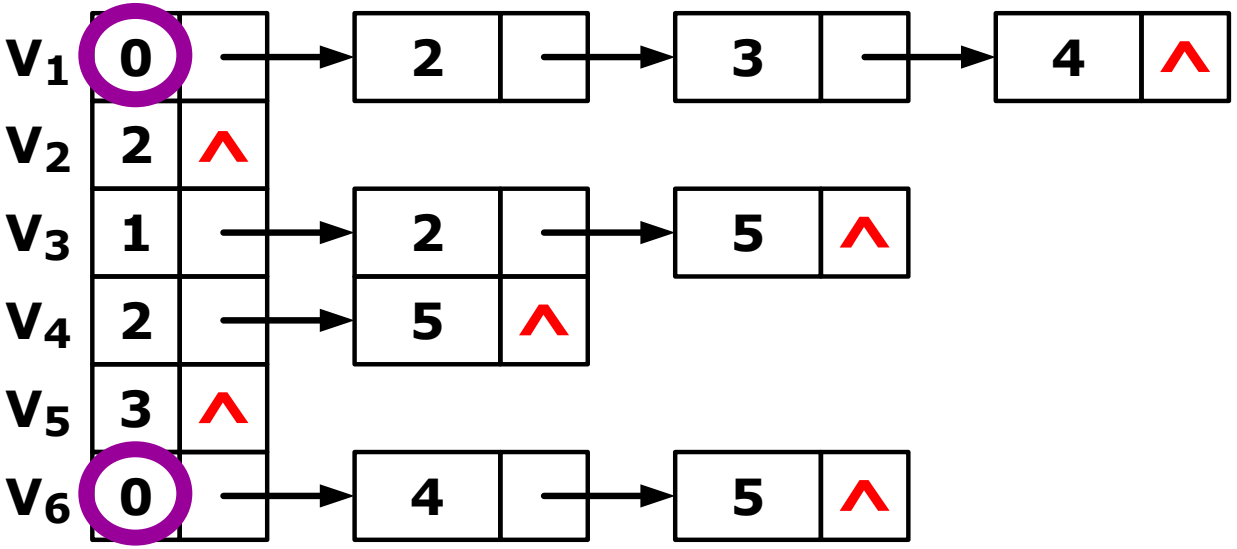
- 采用邻接表作为图的存储结构
- 使用栈记录入度为0的顶点（没有前驱或前驱已经输出）

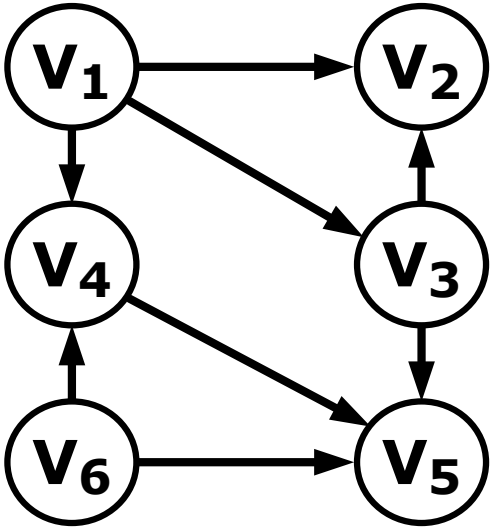
1. 将邻接表中所有入度为0的顶点入栈
2. 用栈进行流程控制：当栈非空时执行
   - 栈顶元素v_i 出栈，并将其输出
   - 在邻接表中查找v_i 的直接后继vk
   - 把v_k 的入度减1
   - 若v_k 的入度为0则对其进行入栈操作
3. 重复上述操作直至栈空为止
   - 若栈空时输出的顶点个数不是n，则该有向图有环
   - 否则拓扑排序完毕：得到一个拓扑有序序列

拓扑排序算法描述

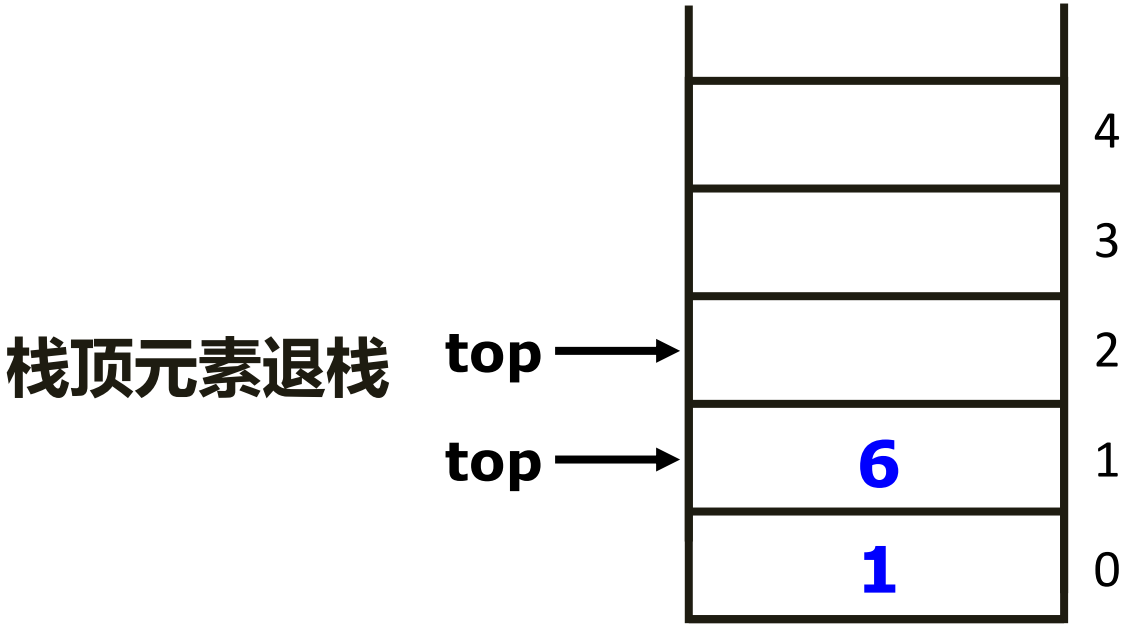
入度为0的顶点入栈



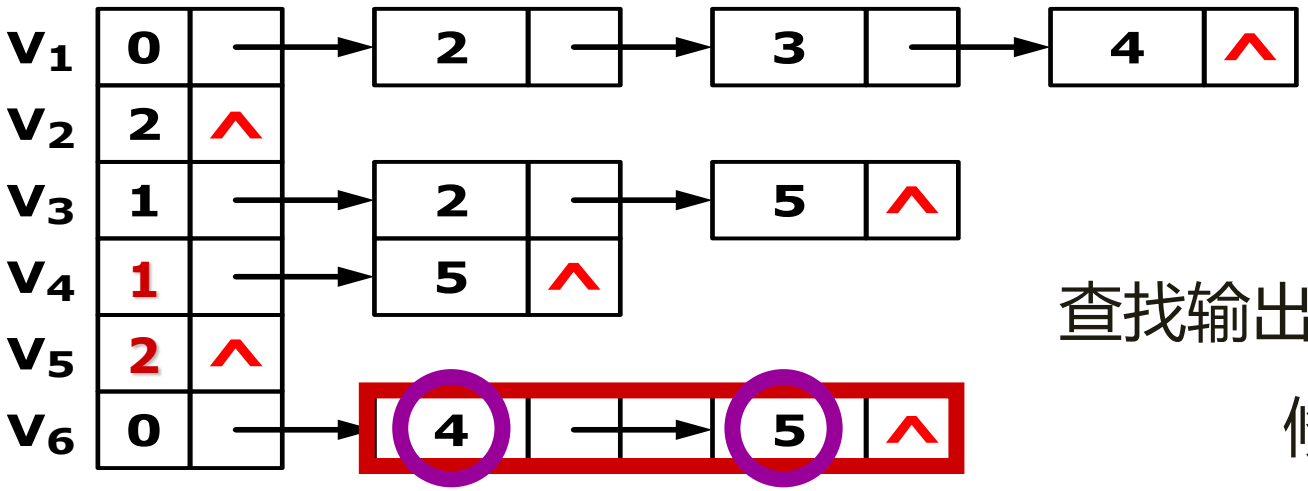




拓扑排序算法描述



输出序列： 6



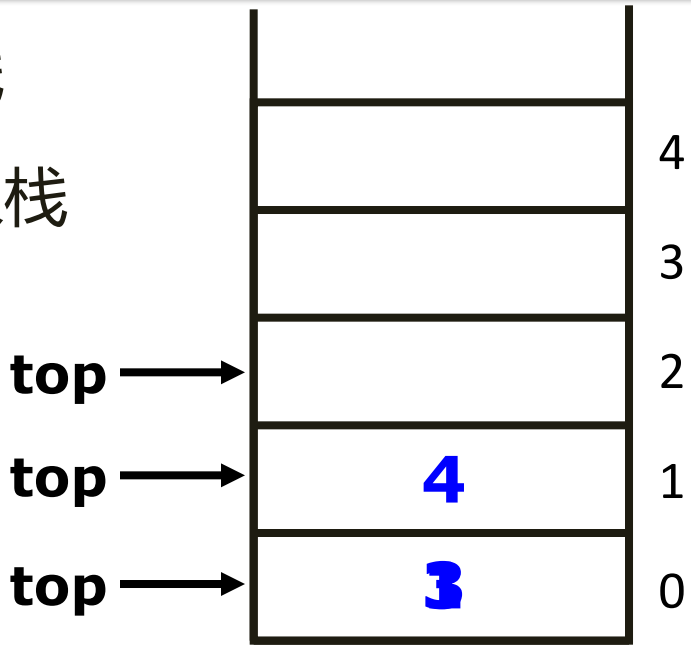
查找输出顶点的直接后继

修改入度

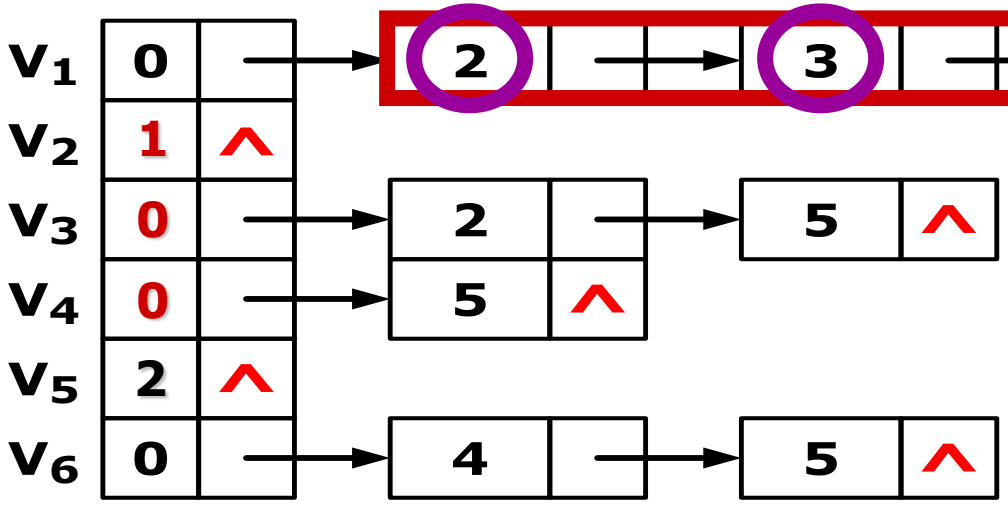
拓扑排序算法描述

栈顶元素退栈

入度为零元素入栈



输出序列：6 , 1



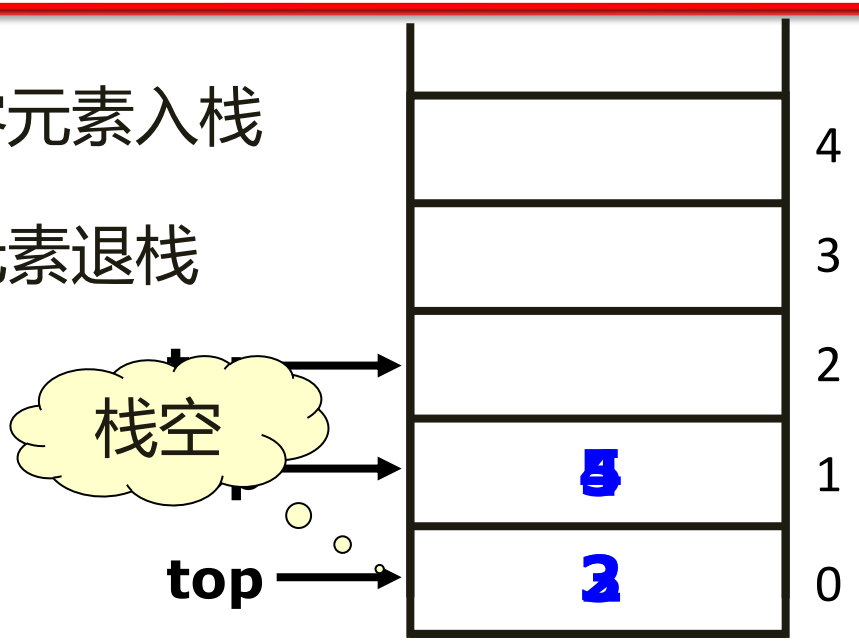
查找输出顶点的直接后继

修改入度

拓扑排序算法描述

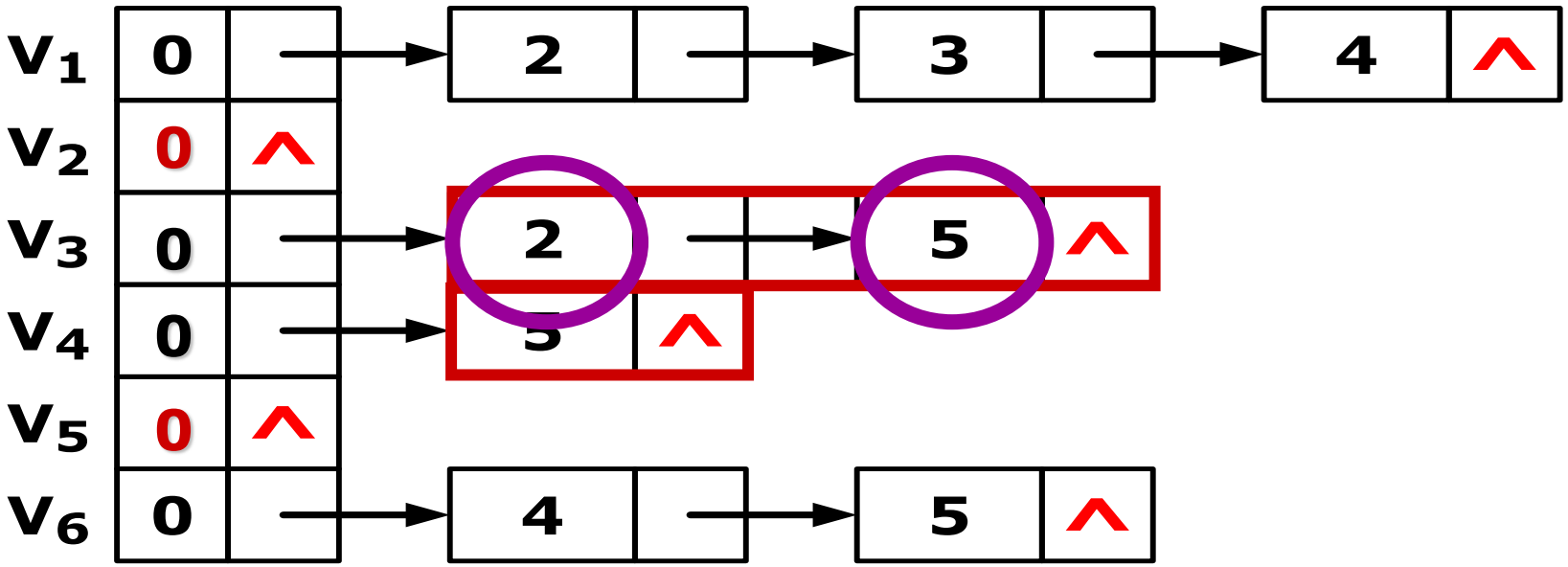
入度为零元素入栈

栈顶元素退栈



输出序列:

6, 1 , 4 , 3 , 5 , 2



```
// 对n个顶点的有向图g进行拓扑排序
void toposort(PGraph G, int n){
    int top, m, k, v, s[M];          // s保存入度为0的顶点的栈
    ENode *p;  top=0;  m=0;          // m记录输出的顶点个数
    for(v=1; v<=n; ++v)              // 入度为0的顶点入栈
        if(G[v].indeg==0)  s[top++]=v;
    while(top>0){
        v = s[--top];  printf("%d  ", v); m++;
        p = G[v].firstarc;
        while(p!=NULL){
            k = p->adjvex;  G[k].indeg--;
            if(G[k].indeg == 0){
                s[top++]=k;  p=p->next;
            }
        }
    }
    if(m<n)  printf("The network has a cycle\n");
}
```

修改顶点入度的运算：T(n)=O(e)

顶点出入栈：T(n)=O(n)

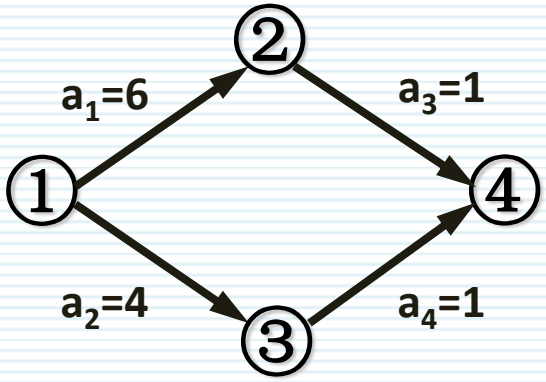
算法复杂度？ T(n)=O(n+e)

关键路径

关键路径

- AOE网（Activity On Edge Network）
- 满足如下条件的有向图称为AOE网
  - 顶点表示事件，弧表示活动
  - 弧上的权表示完成该活动所需的时间
- 源点（表示工程的开始）
  - 入度为0的事件称为源点，AOE网只能有一个源点
- 汇点（表示工程的结束）
  - 出度为0的事件称为汇点， AOE网只能有一个汇点
- 每一事件v表示：以它为弧头的所有活动已经完成
- 每一事件v表示：以它为弧尾的所有活动可以开始

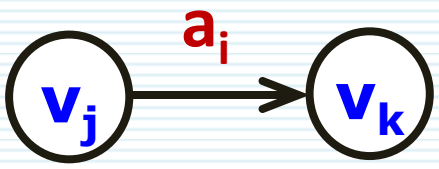
关键路径



- 上图为一个AOE网
  - 有4个事件：v_1，v_2，v_3，v_4（v_1为源点，v_4为汇点）
  - 有4个活动：a_1，a_2，a_3，a_4
    - 其中：事件v_3表示：a_2已完成，a_4可以开始
- AOE网可解决如下问题：
  - 估算最短工期：从源点到汇点至少需要多少时间
  - 找出哪些活动是影响整个工程进展的关键

关键路径的相关术语

- 路径长度：路径上各活动持续时间的总和
  - 即：路径上所有弧的权值之和
- 关键路径：从源点到汇点之间路径长度最长的路径
  - 注意：关键路径不一定唯一
- 事件（v_j）的最早发生时间：$E_v(j)$
  - 从源点到 v_j 的最长路径长度
- 活动（a_i）的最早开始时间：$E_a(i)$
  - 等于该活动的弧尾事件（v_j）的最早发生时间
  - 若以符号<j, k>表示活动 a_i，则有：$E_a(i) = E_v(j)$



关键路径

- 事件（v_k）的最迟发生时间：$L_v(k)$
  - 在不推迟整个工期的前提下，该事件最迟必须发生的时间
- 活动（a_i）的最迟开始时间：$L_a(i)$
  - 该活动的弧头事件最迟发生时间与该活动的持续时间之差
  - 即：$L_a(i) = L_v(k)$ - （a_i 的持续时间）
- 关键活动：$L_a(i) = E_a(i)$ 的活动
- 由此：在AOE网中找关键活动问题可转化为
  - 求满足如下条件的活动： $L_a(i) = E_a(i)$
  - 其中：$E_a(i) = E_v(j)$， $L_a(i) = L_v(k)$ - （a_i 的持续时间）
  - 因此需首先求出事件的最早、最迟发生时间

关键路径算法设计

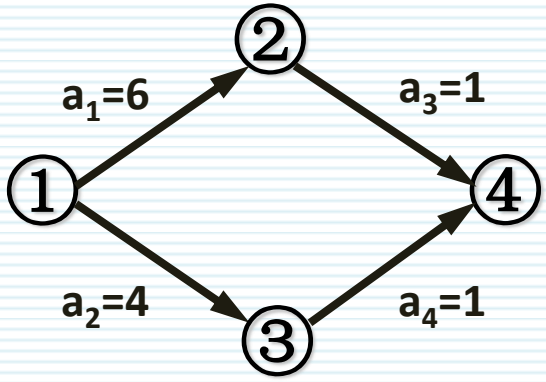
1. 从$E_v(1)=0$ 开始，递推计算出各事件的最早发生时间
   - 事件（v_i）的最早发生时间：从源点到 v_i 的最长路径长度
2. 从$L_v(n) = E_v(n)$开始，计算出各事件的最迟发生时间
   - $L_v(i)$：事件（v_i）最迟必须发生的时间
   - $E_v(i)$：事件（v_i）的最早发生时间
2. 计算出AOE网络中各活动的最早、最迟开始时间
3. 找出 $E_a(i) = L_a(i)$ 的活动，即为关键活动
4. 由关键活动组成的从源点到汇点的路径即为关键路径

关键路径算法设计

1. 从$E_V(1)=0$ 开始，递推计算出各事件的最早发生时间

$$ev(k)=\max_{<j,k>\in p}(ev(j)+w(j,k))\quad(1<k\le n)$$

- 其中：p是所有以k为弧头的弧集合
- w(j,k) 表示活动<j,k>的持续时间
- 以右图为例：

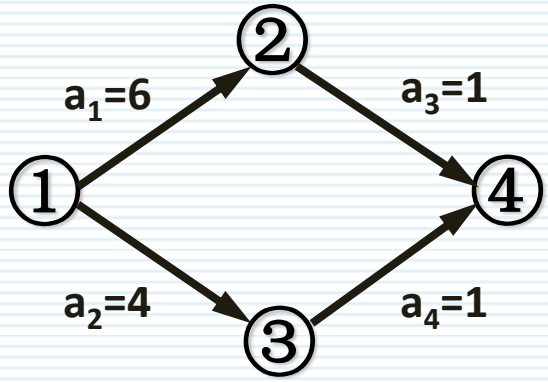


  - $E_v(4)= \max\{E_v(2)+w(2,4),\ E_v(3)+w(3,4)\}$
  - $E_v(2) = 6,\ w(2,4) = 1$
  - $E_v(3) = 4,\ w(3,4) = 1$
  - $E_v(4) = \max\{6+1,\ 4+1\} = 7$

关键路径算法设计

2. 从$L_V(i) = E_V(i)$开始，计算出各事件的最迟发生时间

$$Lv(j) = \min_{<j,k>\in S}(Lv(k) - w(j,k)) \quad (1 < j \le n)$$



- 其中：S是所有以 j 为弧尾的弧集合
- 以右图为例：
  - $L_v(4) = E_v(4) = 7$
  - $L_v(2) = L_v(4)-1 = 6$，$L_v(3) = L_v(4)-1 = 6$
  - $L_v(1) = \min\{L_v(2)-w(1,2),\ L_v(3)-w(1,3)\}$
  - $w(1,2) = 6,\ w(1,3) = 4$
  - $L_v(1) = \min\{6-6,\ 6-4\}=0$

关键路径算法设计

3. 设活动(a_i)由弧<j, k>表示，其持续时间为w(j,k)

   - 则利用下面公式计算出各活动的最早、最迟开始时间：
   - $E_a(i) = E_v(j)$
   - $L_a(i) = L_v(k) - w(j, k)$

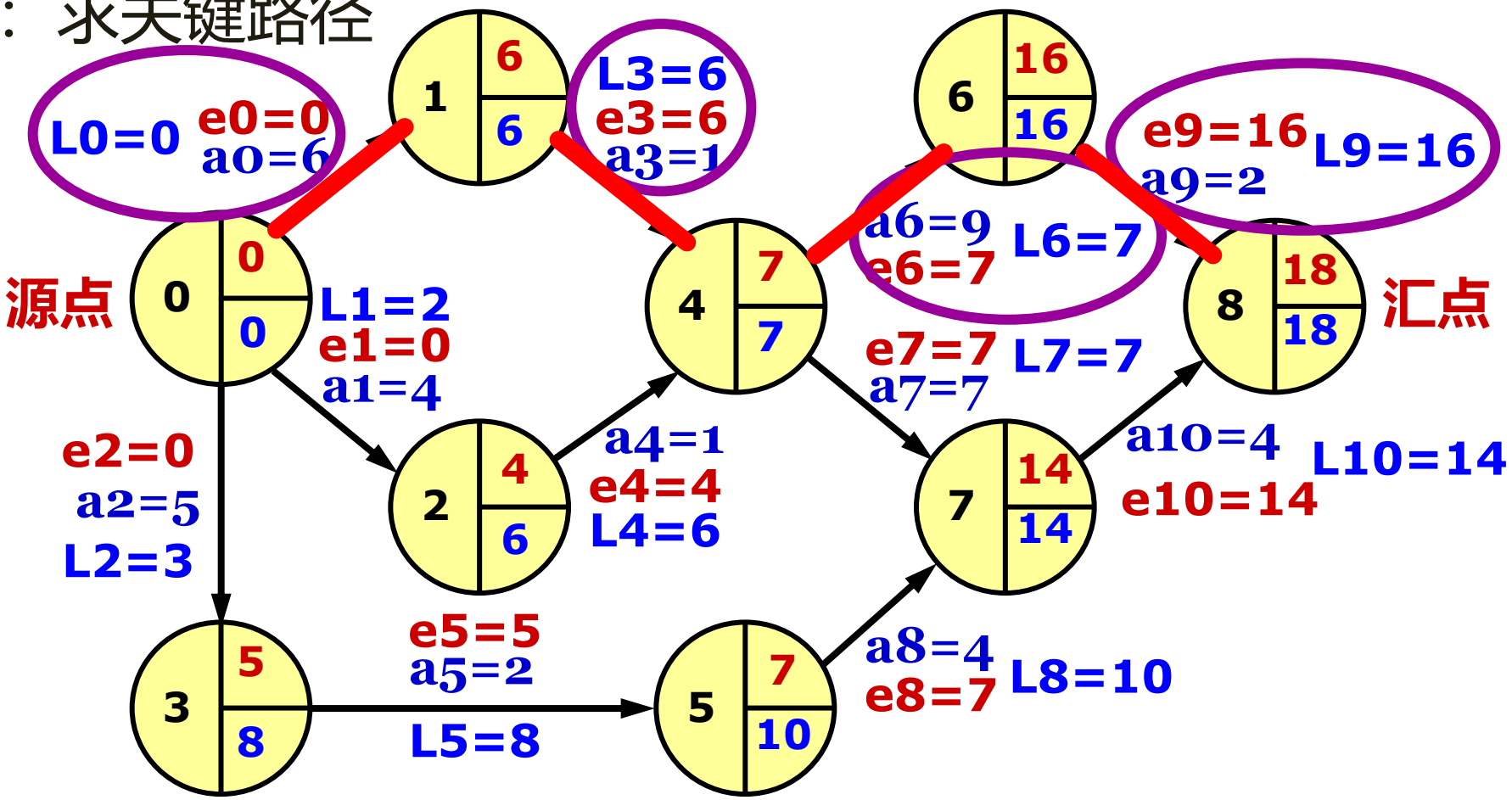
4. 找出 $E_a(i) = L_a(i)$ 的活动，即为关键活动

5. 由关键活动组成的从源点到汇点的路径即为关键路径

关键路径算法流程说明

- 输入顶点和弧信息，建立图的邻接表，计算每个顶点的入度
- 从源点 v_1 出发，对图中顶点进行拓扑排序
  - 排序过程中求顶点的$E_v[i]$
  - 将得到的拓扑序列进栈
- 从汇点 v_n 出发，按逆拓扑序列求顶点的$L_v[i]$
- 根据各顶点的E_v和L_v值计算每条弧的$E_a[i]$和$L_a[i]$
- 找出$E_a[i] = L_a[i]$的关键活动

例：求关键路径



1. 计算 $E_v(i)$：从v_0开始
2. 计算 $L_v(i)$：从v_8开始
3. 计算 $E_a(i)$：从a_0开始
4. 计算 $L_a(i)$：从a_{10}开始

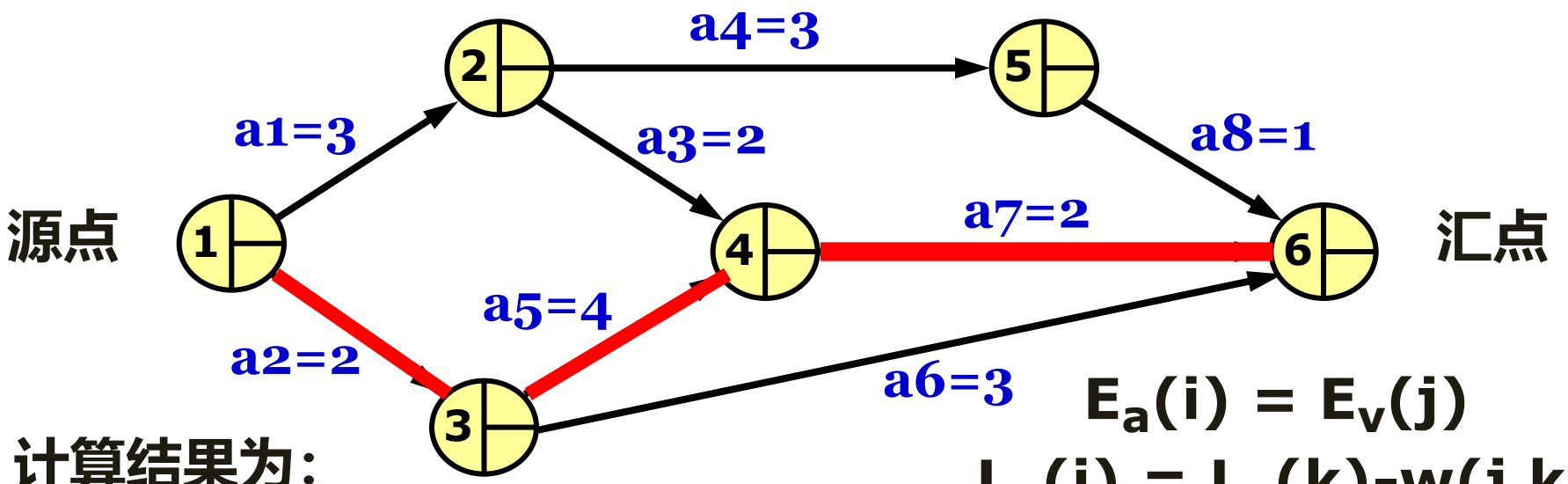
- 关键活动：

  $\{a_i \mid E_a(i) = L_a(i)\}$

- 关键路径：

  $v_0 \rightarrow v_1 \rightarrow v_4 \rightarrow v_6 \rightarrow v_8$

例：求解如下AOE网的关键路径



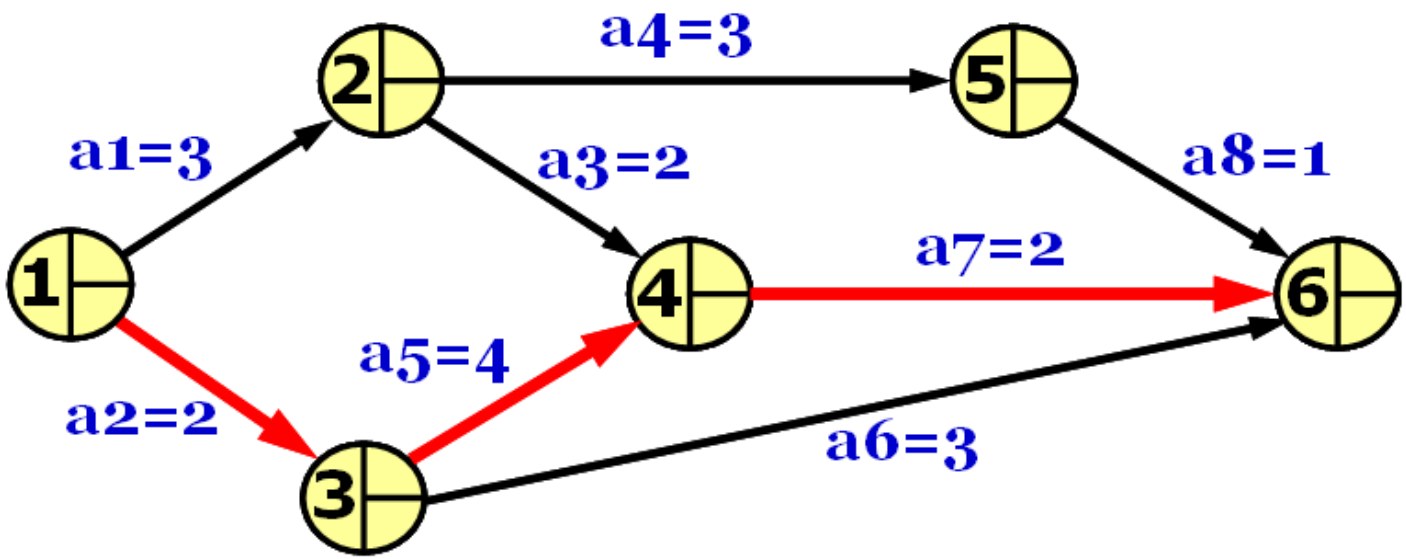
$E_a(i) = E_v(j)$

$L_a(i) = L_v(k)-w(j,k)$

计算结果为：

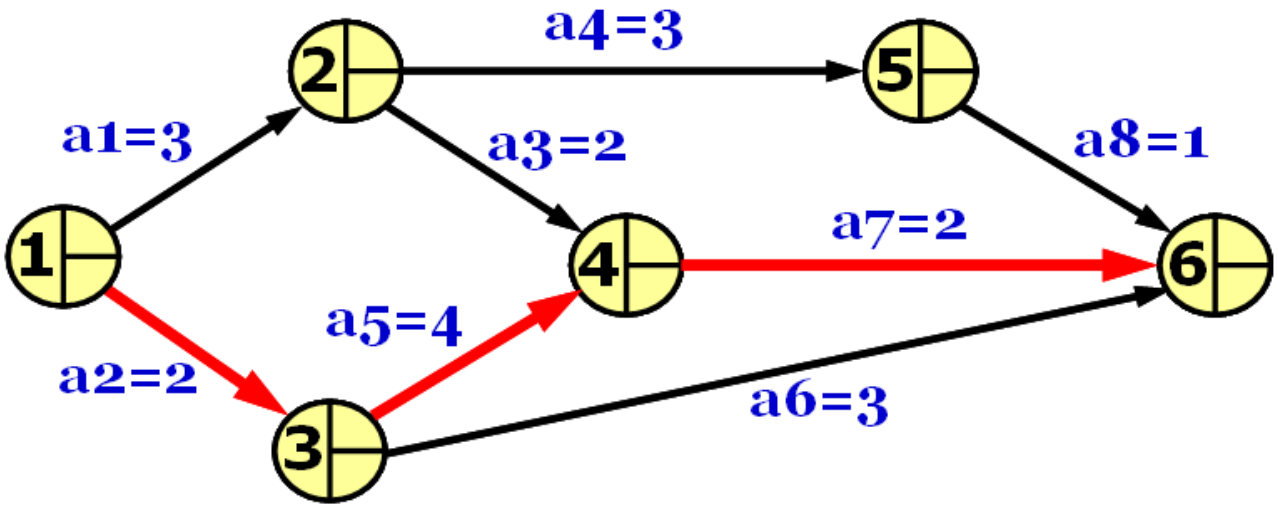
顶点	Ev	Lv	活动	弧	w	Ea	La	La-Ea	关键活动
v1	0	0	a1	<1,2>	3	0	1	1	
v2	3	4	a2	<1,3>	2	0	0	0	a2
v3	2	2	a3	<2,4>	2	3	4	1	
v4	6	6	a4	<2,5>	3	3	4	1	
v5	6	7	a5	<3,4>	4	2	2	0	a5
v6	8	8	a6	<3,6>	3	2	5	3	
	Max	Min	a7	<4,6>	2	6	6	0	a7
	+	-	a8	<5,6>	1	6	7	1	

关键路径小结



- 关键路径上的活动都是关键活动
  - 缩短非关键活动都不能缩短整个工期
    - 例如将a_6 缩短为1天，则整个工期仍为8天
    - 将a_6 推迟3天开始，或将a_6 拖延3天完成 ……
    - 均不影响整个工期

关键路径小结



- 分析关键路径的目的是找出影响整个工期的关键活动序列
  - 缩短关键活动的持续时间，通常可以缩短整个工期
    - 例如：将a_7 缩短为1天，则整个工期为7天
    - 然而：此时再缩短任一关键活动均不能缩短工期
    - 结论：在有多条关键路径时，缩短那些存在于所有关键路径上的关键活动，才能缩短整个工期

关键路径算法小结

- 求关键路径必须在拓扑排序的前提下进行
  - 有环图不能求关键路径
- 只有缩短关键活动的时间才有可能缩短工期
  - 但如果一个关键活动不在所有的关键路径上
    - 减少它并不能减少工期
  - 只有在不改变关键路径的前提下
    - 缩短关键活动才能缩短整个工期


UESTC
1956