第七章

图

※ 教学内容：

图的基本概念；图的存储结构；图的遍历；最小生成树；最短路径；拓扑排序和关键路径

※ 教学重点：

图的各种存储结构;遍历图的递归和非递归算法;应用图的遍历算法求各种简单路径问题，比如，最小生成树、最短路径、拓扑排序、关键路径等。

※ 教学难点：

图的遍历算法，构造最小生成树的算法

第七章　图

7.1 图的定义

7.2 图的存储表示

7.3 图的遍历

7.4 最小生成树

7.5 两点之间的最短路径问题

7.6 拓扑排序

7.7 关键路径

7.1 图的定义

一、图的结构定义:

图是由一个顶点集 V 和一个弧集 VR构成的数据结构

$$G = (V , VR)$$

其中，

VR = {<v,w>| v,w∈V 且 P(v,w)}

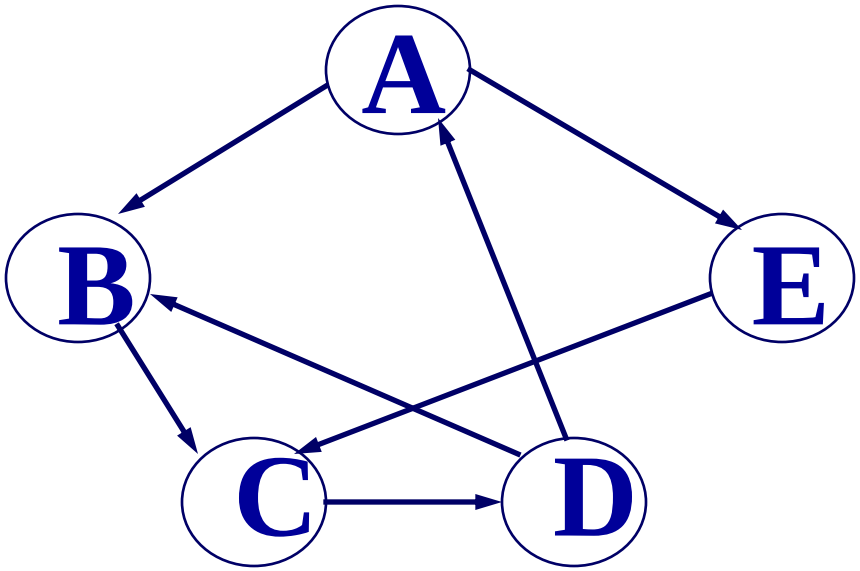
<v,w>表示从 v 到 w 的一条弧，并称 v 为弧尾，w 为弧头。

谓词 P(v,w) 定义了弧 <v,w>的意义或信息。

有向图

由于“弧”是有方向的，因此称由顶点集和弧集构成的图为有向图。

例如： $G_1 = (V_1, VR_1)$



其中

V_1={A, B, C, D, E}

VR_1={<A,B>, <A,E>,
<B,C>, <C,D>,
<D,B>, <D,A>,
<E,C> }

无向图:

若<v, w>∈VR 必有<w, v>∈VR, 即VR是对称的，则以无序对(v,w) 代替这两个有序对，表示顶点v 和顶点w 之间的一条**边**。

由顶点集和边集构成的图称作无向图。

例如: $G_2=(V_2,VR_2)$

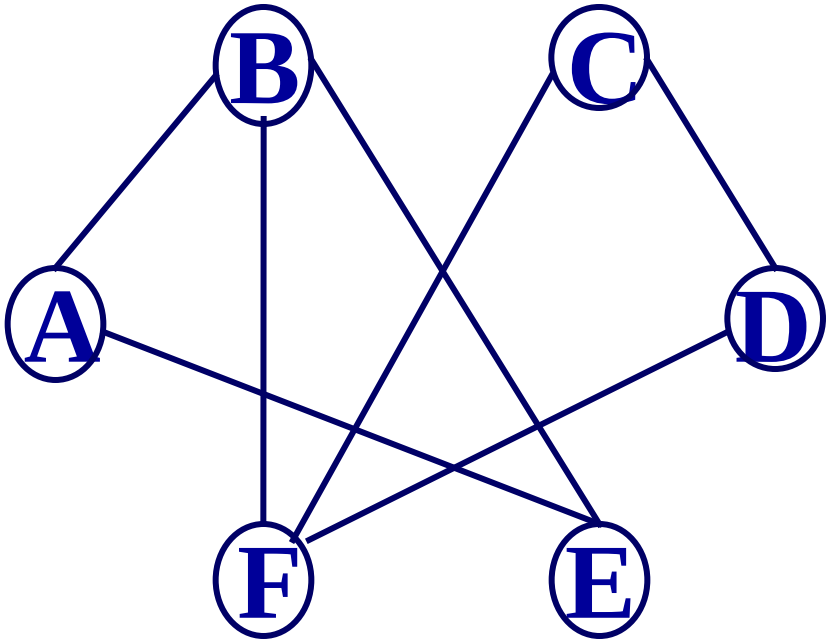
$V_2=\{A, B, C, D, E, F\}$

$VR_2=\{(A,B), (A,E),$

$(B,E), (C,D),$

$(D,F), (B,F),$

$(C,F)\}$



名词和术语

网、子图

完全图、稀疏图、稠密图

邻接点、度、入度、出度

路径、路径长度、简单路径、简单回路

连通图、连通分量、

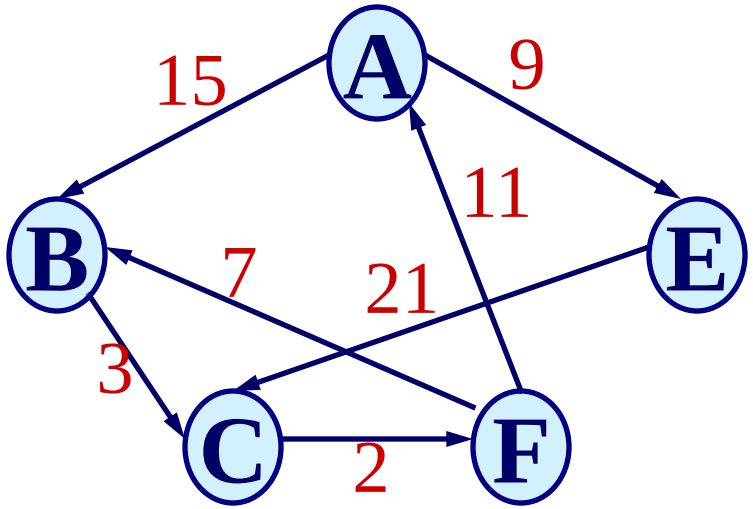
强连通图、强连通分量

生成树、生成森林

网：

有时图的边或弧具有与它相关的数，这种与图的边或弧相关的数叫做**权**。这些权可以表示从一个顶点到另一个顶点的距离或耗费。

弧或边带权的图分别称为**网(有向网、无向网)**。



子图：

假设有两个图

$G=(V,\{VR\})$ 和 $G'=(V',\{VR'\})$

如果 $V'\subseteq V$, $VR'\subseteq VR$，则称 G' 为 G 的**子图**。

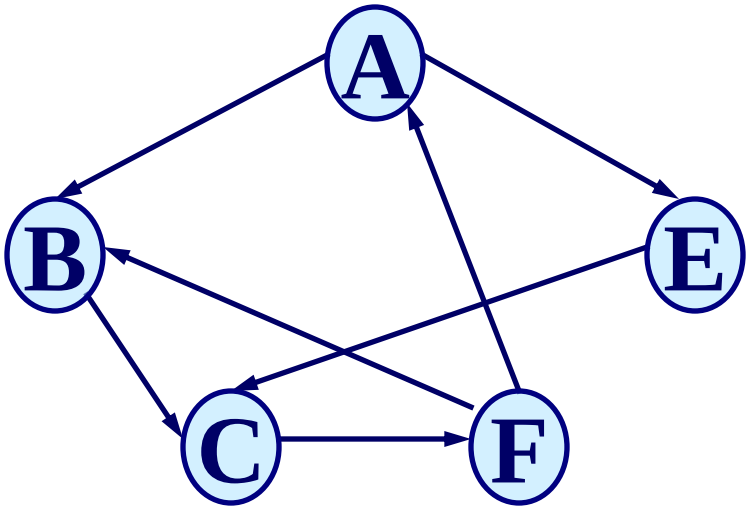


图 G

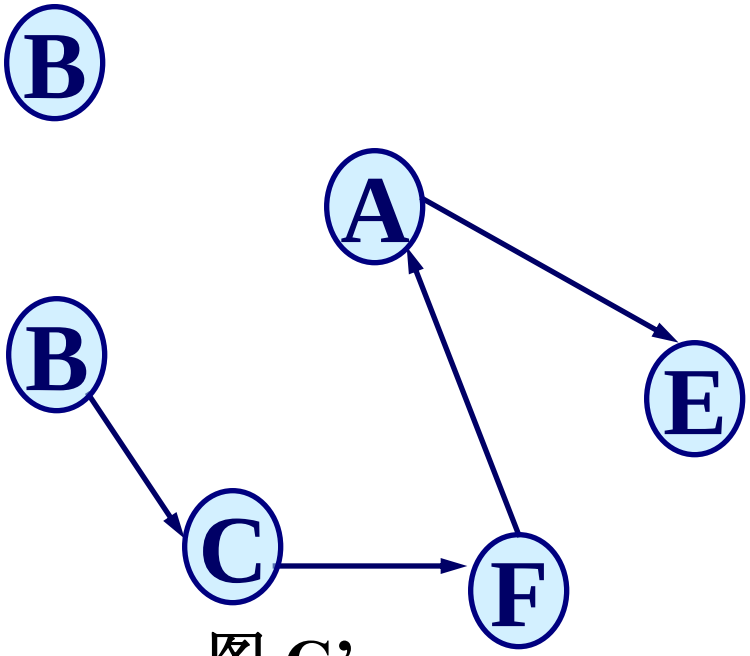


图 G’

完全图、稀疏图和稠密图:

假设图中有 **n** 个顶点，**e** 条边，则:

含有 **e=n(n-1)/2** 条边的无向图称作**完全图**;

含有 **e=n(n-1)** 条弧的有向图称作 **有向完全图**;

若边或弧的个数 **e<nlogn**，则称作**稀疏图**，

否则称作**稠密图**。

邻接点和度

假若顶点**v**和顶点**w**之间存在一条边，则称顶点**v** 和**w** 互为**邻接点**；

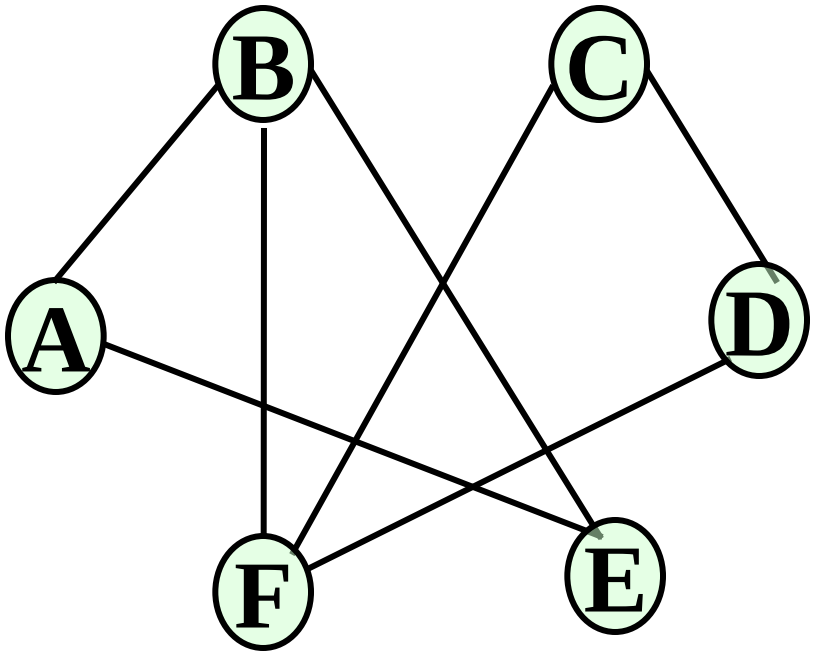
边(v, w) 和顶点v 和w 相**关联**；

和顶点v 关联的边的数目定义为顶点v的**度**。

例如:

TD(B) =3

TD(A) = 2



入度和出度

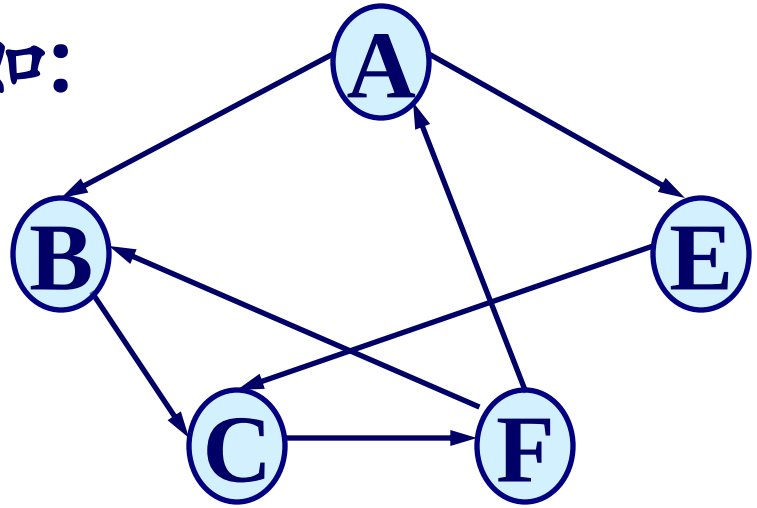
对有向图来说，

顶点v的**出度**：以顶点v为弧尾的弧的数目；

顶点v的**入度**：以顶点v为弧头的弧的数目。

顶点的**度(TD)=出度(OD)+入度(ID)**

例如：



OD(B) =1

ID(B) =2

TD(B) =3

路径

设图G=(V,{VR})中的一个顶点序列

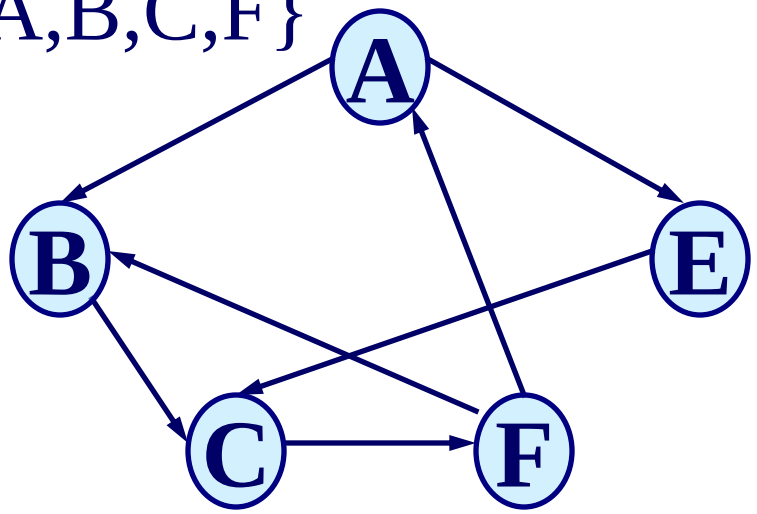
{ $u=v_{i,0},v_{i,1},\ldots,v_{i,m}=w$}中，$(v_{i,j-1},v_{i,j})\in VR$ $1\le j\le m$,

则称从顶点u 到顶点w 之间存在一条**路径**。

路径上边的数目称作**路径长度**。

如: 长度为3的路径

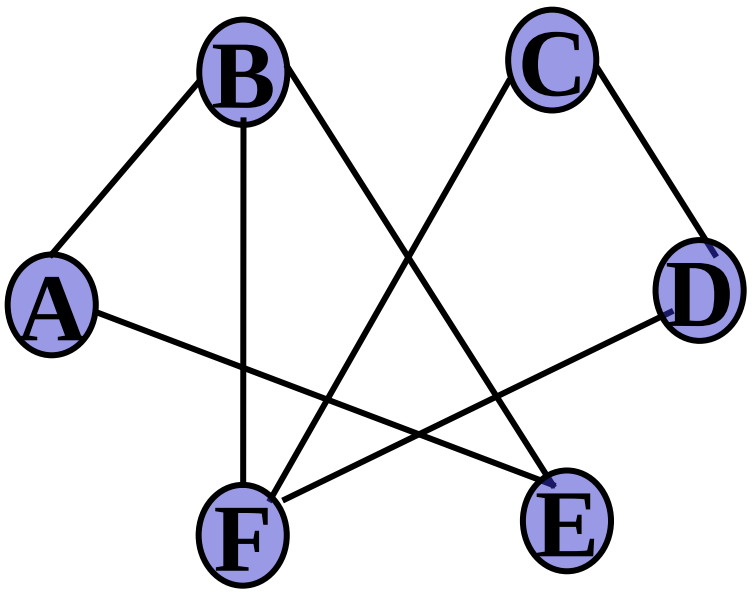
{A,B,C,F}



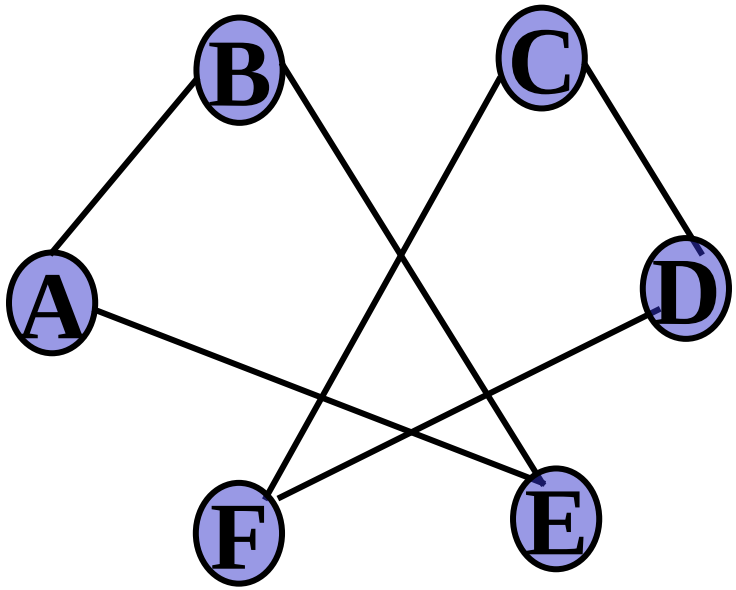
简单路径: 序列中顶点不重复出现的路径。

回路: 序列中第一个顶点和最后一个顶点相同的路径。

若图G中任意两个顶点之间都有路径相通，则称此图为**连通图**；否则，称之为**非连通图**。

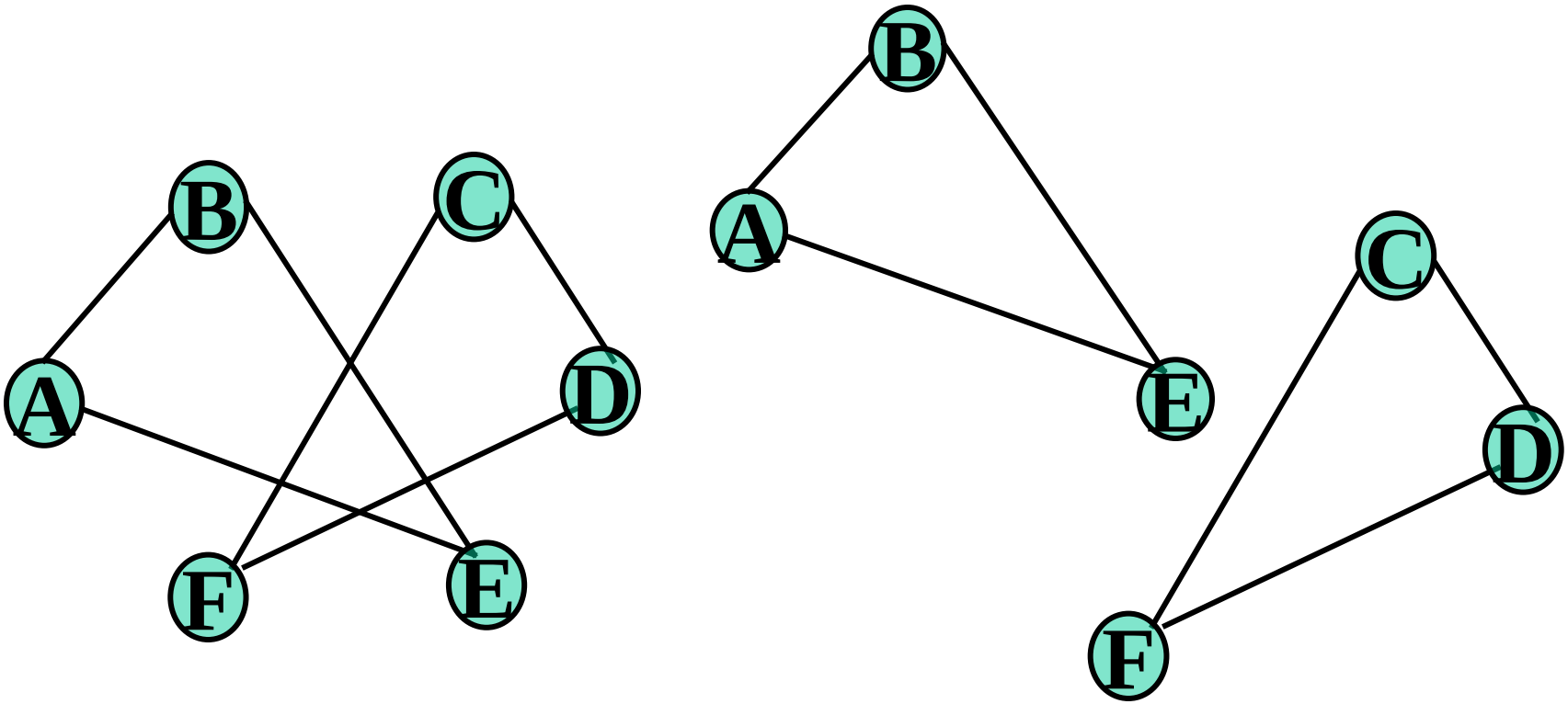


连通图

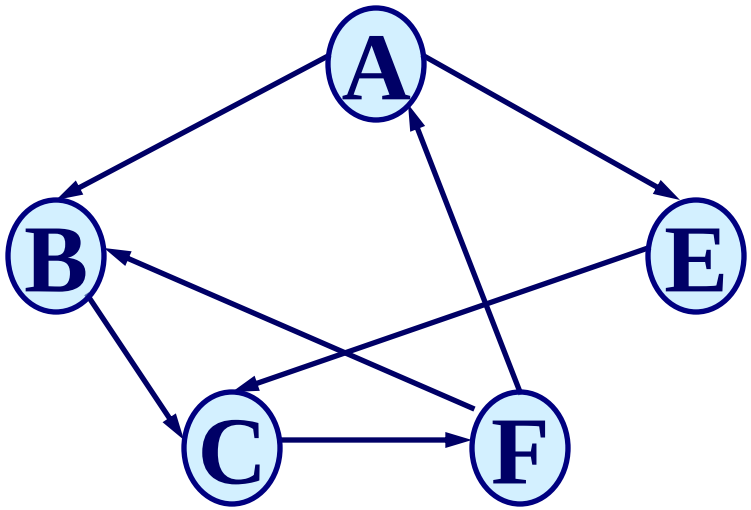


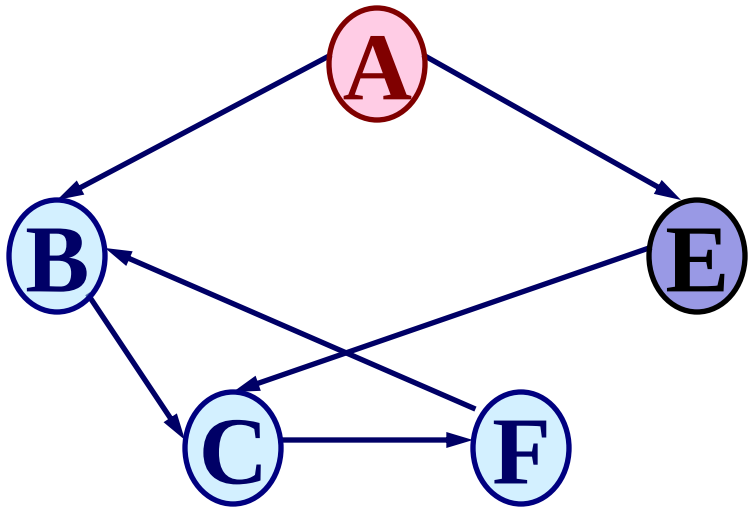
非连通图

若无向图为非连通图，则图中各个极大连通子图称作此图的**连通分量**。



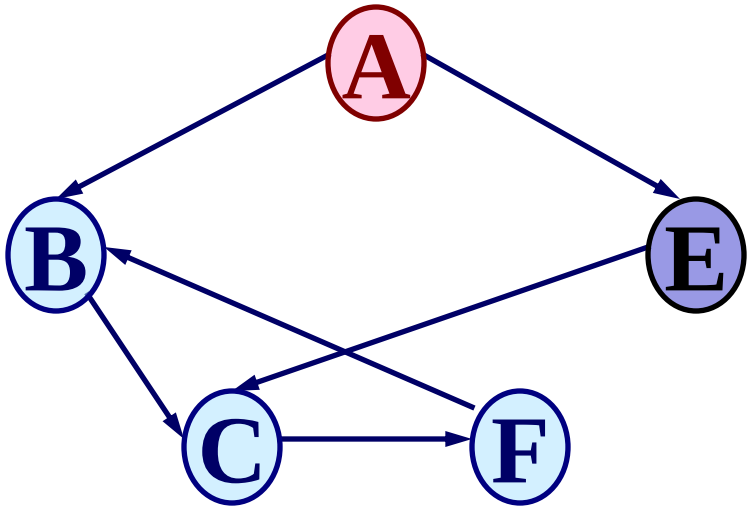
对有向图来说，若任意两个顶点之间都存在一条有向路径，则称此有向图为**强连通图**。

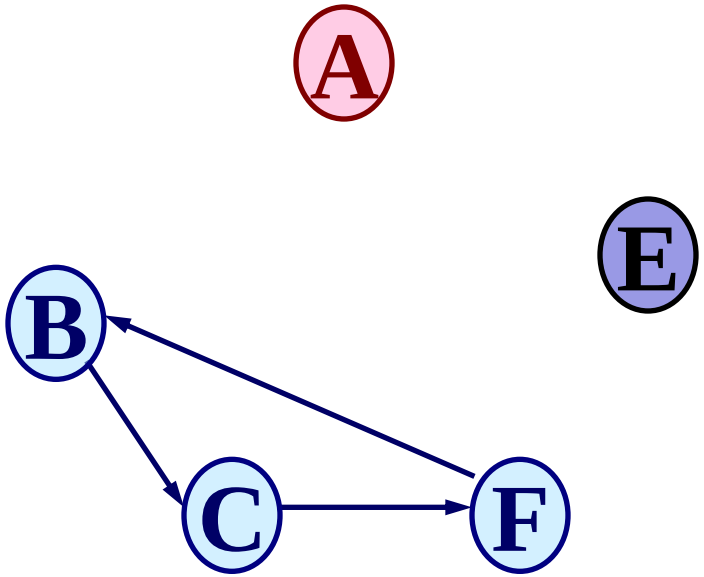




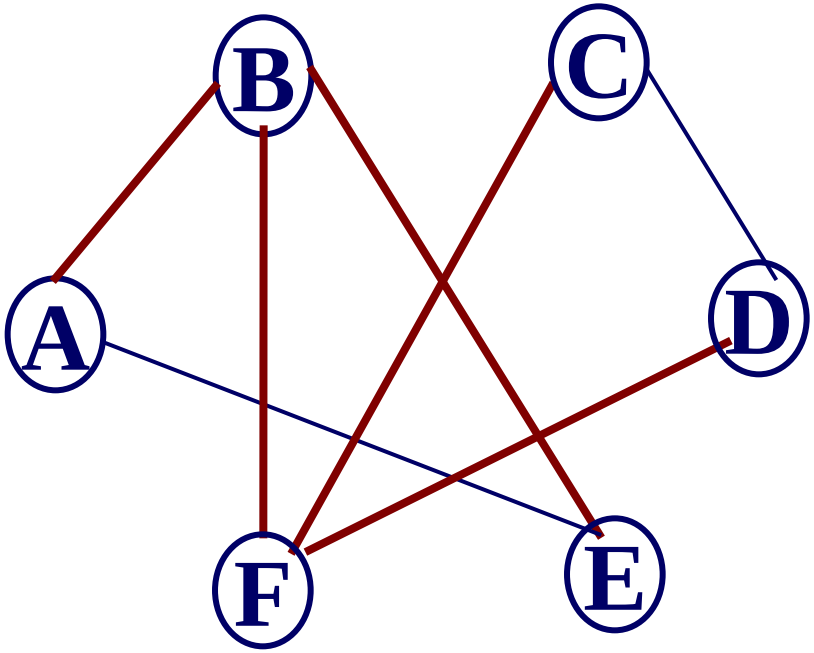
强连通图

否则，其各个强连通子图称作它的**强连通分量**。

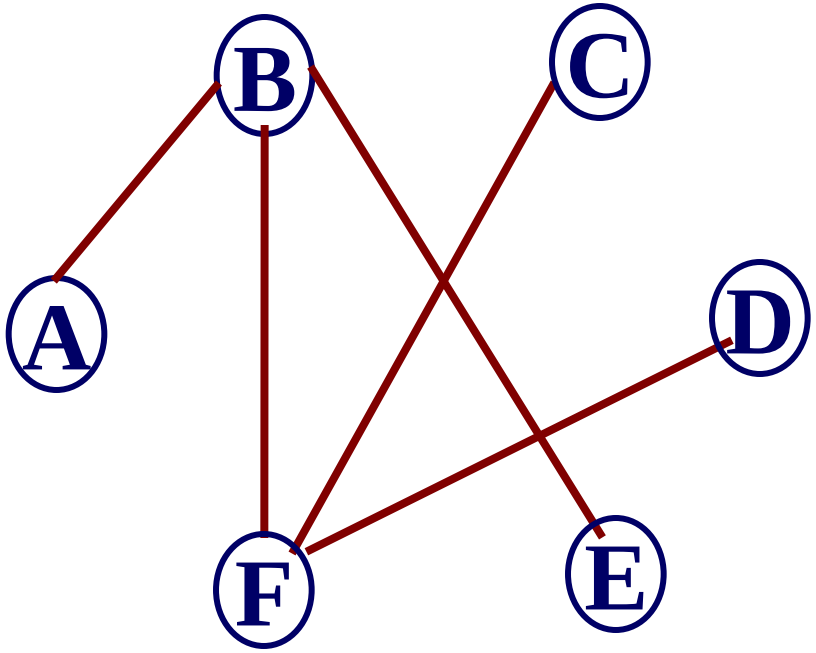




假设一个连通图有 n 个顶点和 e 条边，其中 n-1 条边和 n 个顶点构成一个极小连通子图，称该极小连通子图为此连通图的**生成树**。



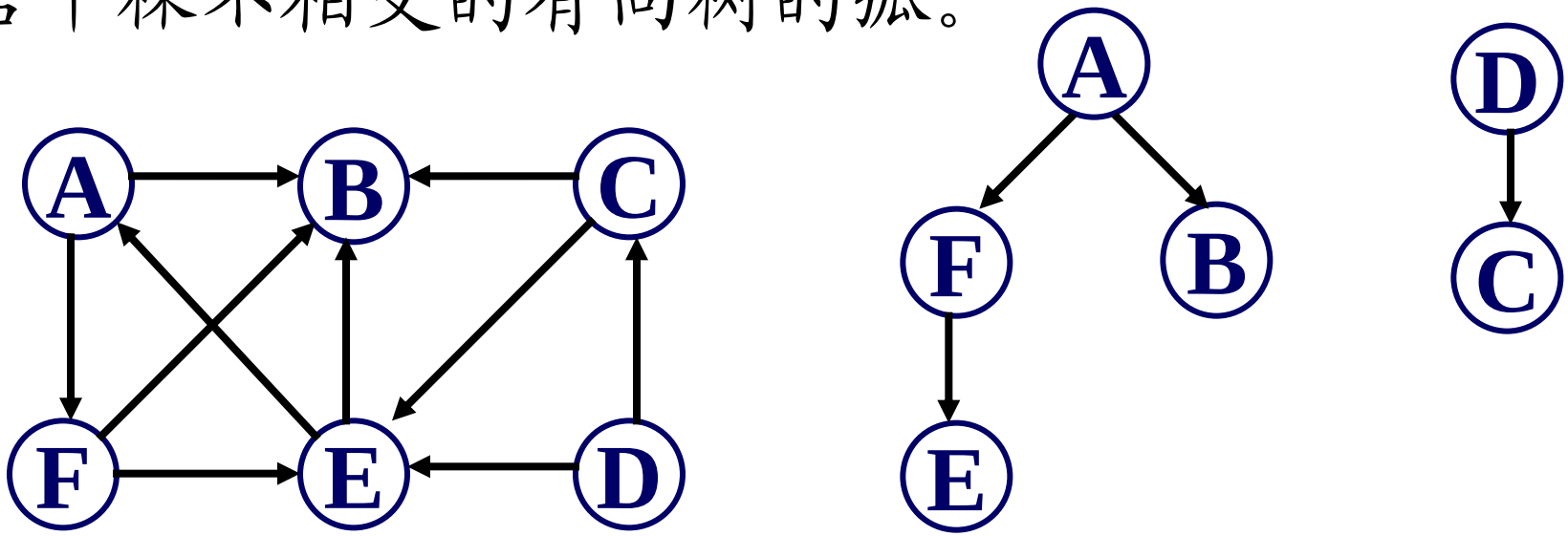
连通图



连通图的**生成树**

对非连通图，则称由各个连通分量的生成树的集合为此非连通图的**生成森林**

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。



一个有向图及其**生成森林**

二、抽象数据类型图的定义

ADT Graph{

数据对象V： **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： **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P:

}ADT Graph

基本操作

- 结构的建立和销毁
- 对顶点的访问操作
- 插入或删除顶点
- 插入和删除弧
- 对邻接点的操作
- 遍历

结构的建立和销毁

CreateGraph(&G, V, VR):

// 按定义(V, VR) 构造图

DestroyGraph(&G):

// 销毁图

对顶点的访问操作

LocateVex(G, u);

// 若G中存在顶点u，则返回该顶点在
// 图中“位置”；否则返回其它信息。

GetVex(G, v); // 返回 v 的值。

PutVex(&G, v, value); // 对 v 赋值value。

对邻接点的操作

FirstAdjVex(G, v);

// 返回 v 的“第一个邻接点”。若该顶点
// 在 G 中没有邻接点，则返回“空”。

NextAdjVex(G, v, w);

// 返回 v 的(相对于 w 的)“下一个邻接点”。
// 若w是v的最后一个邻接点，则返回“空”。

插入或删除顶点

InsertVex(&G, v);

//在图G中增添新顶点v。

DeleteVex(&G, v);

// 删除G中顶点v及其相关的弧。

插入和删除弧

InsertArc(&G, v, w);

// 在G中增添弧<v,w>，若G是无向的，
// 则还增添对称弧<w,v>。

DeleteArc(&G, v, w);

//在G中删除弧<v,w>，若G是无向的，
//则还删除对称弧<w,v>。

遍　历

DFSTraverse(G, v, Visit());

//从顶点v起深度优先遍历图G，并对每
//个顶点调用函数Visit一次且仅一次。

BFSTraverse(G, v, Visit());

//从顶点v起广度优先遍历图G，并对每
//个顶点调用函数Visit一次且仅一次。

7.2 图的存储表示

一、图的数组(邻接矩阵)存储表示

二、图的邻接表存储表示

三、有向图的十字链表存储表示

一、图的数组(邻接矩阵)存储表示

邻接矩阵:

表示顶点间相邻关系的矩阵，用一个二维数组来表示集合VR。此二维数组称为邻接矩阵。

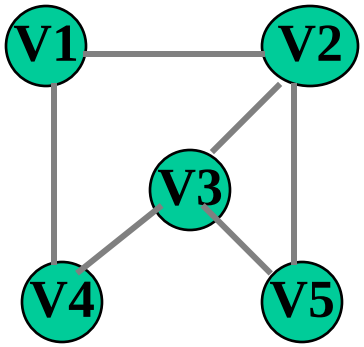
若G是一个具有n个顶点的图，其邻接矩阵是一个n阶方阵。

(1) 无向图的邻接矩阵

- 定义

设G=(V,VR)是有n(n ≥ 1)个顶点的无向图，则G的邻接矩阵是一个n × n 矩阵:

$$A[i,j]=A[j,i]=\begin{cases}1 & (v_i,v_j)\in VR\\ 0 & (v_i,v_j)\notin VR\end{cases}$$



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

特点：

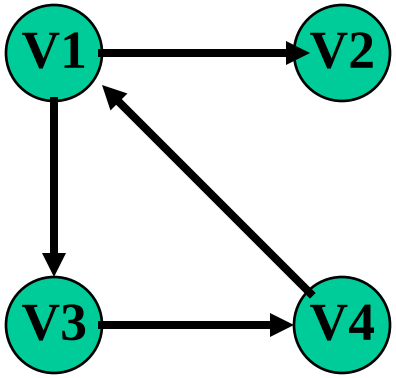
- 无向图的邻接矩阵为对称方阵；
- 邻接矩阵第i行（或第i列）的元素之和则是顶点Vi的度。

(2) 有向图的邻接矩阵

- 定义

设G=(V,VR)是有n(n≥1)个顶点的有向图，G的邻接矩阵是具有如下性质的n × n方阵:

$$A[i,j]=\begin{cases}1 & \text{当 } \langle v_i,v_j\rangle \in E \\ 0 & \text{当 } \langle v_i,v_j\rangle \notin E\end{cases}$$



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

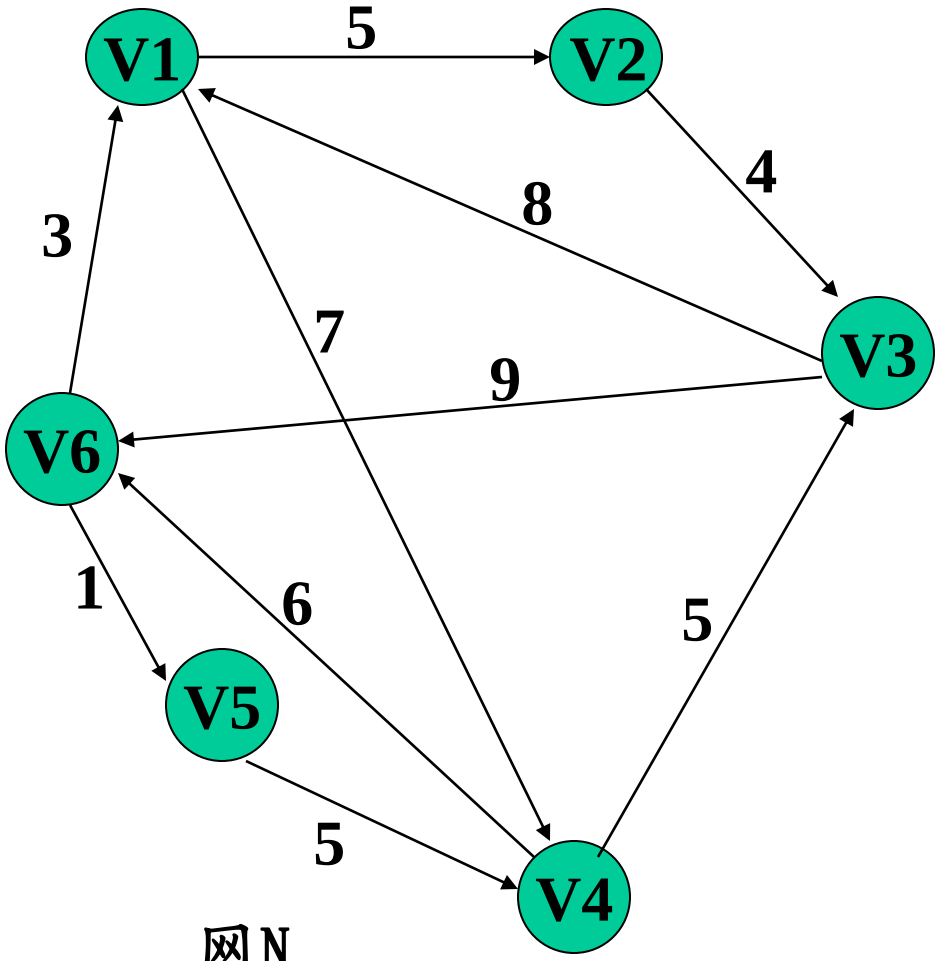
- 有向图的邻接矩阵为非对称矩阵;
- 邻接矩阵第i行的元素之和为顶点Vi的
- 邻接矩阵第j列的元素之和为顶点Vj的入度。

(3) 网的邻接矩阵

- 定义

设G=(V,VR)是有n(n≥1)个顶点的有向网，G的邻接矩阵是具有如下性质的n × n方阵:

$$A[i,j]=\begin{cases} w_{ij} & \text{当 } \langle v_i,v_j\rangle \in E \\ \infty & \text{当 } \langle v_i,v_j\rangle \notin E \end{cases}$$



网N

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

网N的邻接矩阵

图的邻接矩阵存储表示：

```
#define INFINITY    INF_MAX   // 最大值∞
#define MAX_VERTEX_NUM  20
                                  // 最大顶点个数
Typedef enum {DG,DN,UDG,UDN}
                                  GraphKind;
                // {有向图，有向网，无向图，无向网}
```

```
typedef struct ArcCell {  // 弧的定义
    VRType  adj;   // VRType是顶点关系类型，
                   // 对无权图，用1或0表示相邻否，
                   // 对带权网，则为权值类型
    InfoType  *info;   // 该弧相关信息的指针
} ArcCell, AdjMatrix [MAX_VERTEX_NUM]
                     [MAX_VERTEX_NUM];
```

```
typedef struct {                                    // 图的定义
    VertexType  vexs[MAX_VERTEX_NUM];
                                                    // 顶点向量
    AdjMatrix  arcs;                                // 邻接矩阵
     int  vexnum, arcnum;                           // 图的当前顶点数和弧数
     GraphKind  kind;                               // 图的种类标志
} MGraph;
```

- **优点：**

借助于邻接矩阵，容易判定任意两个顶点之间是否有边（或弧）相连，并容易求得各个顶点的度。

对于无向图，顶点Vi 的度是邻接矩阵第i 行（或第i列）的元素之和。

对于有向图，第i 行的元素之和为顶点Vi 的出度；第j 列的元素之和为顶点Vj的入度。

- **缺点：**

占用的存储单元个数只与图中顶点个数有关，而与边的数目无关。

二、图的邻接表存储表示

邻接表是图的一种链式存贮结构。在邻接表中，对图的每个顶点建立一个单链表，第i个单链表中的结点表示顶点i的所有邻接顶点。链表中每个结点由三个域组成:

表结点

–adivex — 顶点vi的邻接点

–info —存贮与边和弧相关的权值

–nextarc—指向vi下一个邻接点的指针

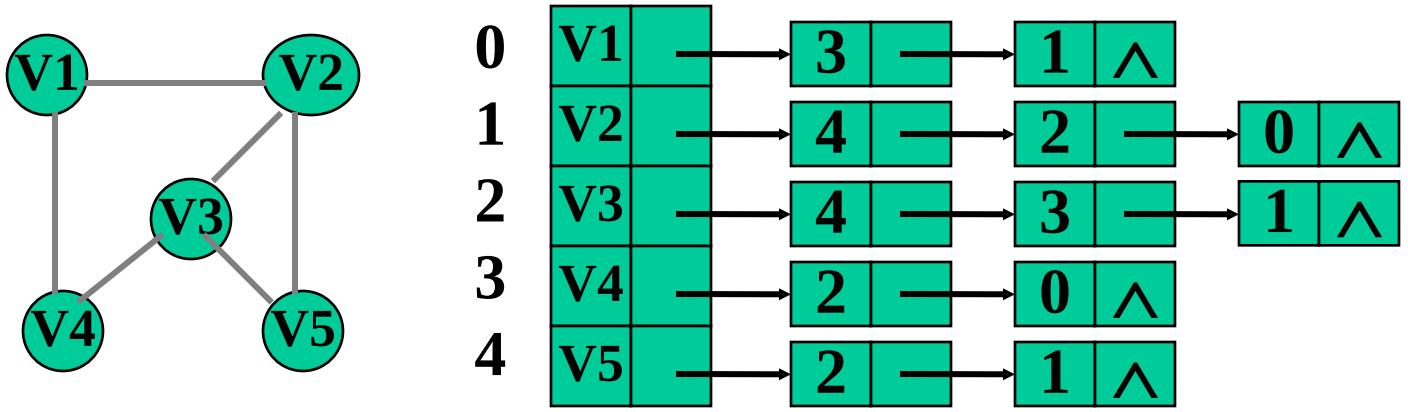
在每个链表上附设一个表头结点，由两个域组成：

–data　— 存贮与顶点vi有关信息的数据；

–fristarc — 指向vi的第一个邻接点的指针。

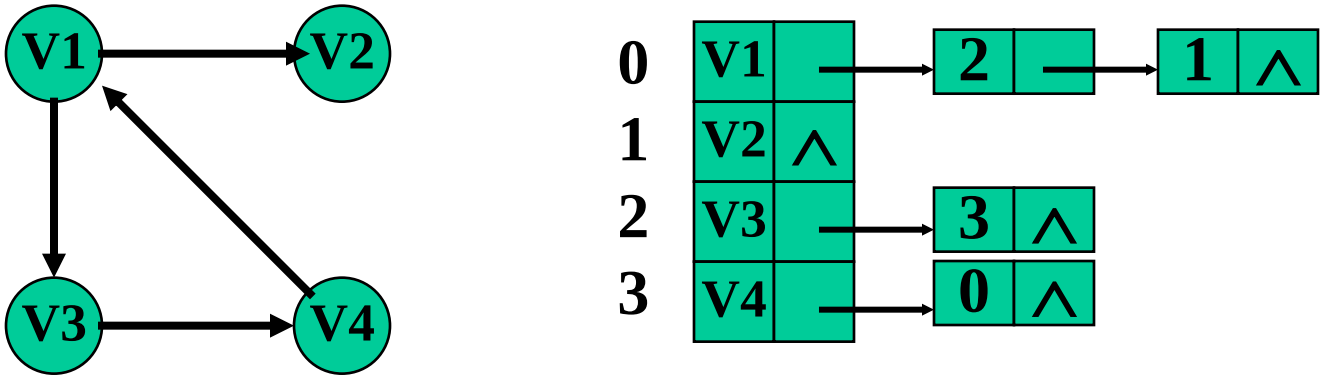
表头结点通常以顺序结构的形式存储，以便随机访问任一顶点的邻接点链表。

无向图的邻接表



在无向图的邻接表中，顶点Vi的度恰为第i个链表中的表结点数.

有向图的邻接表

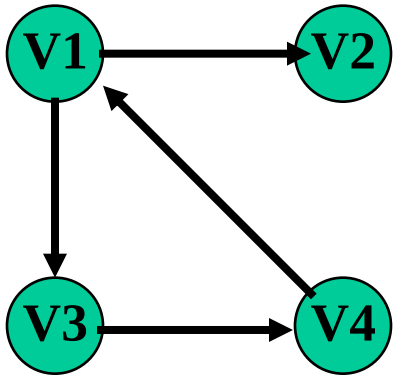


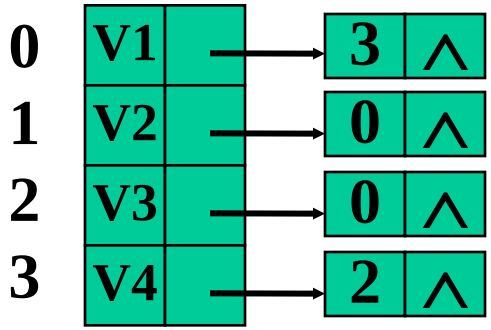
有向图的邻接表中，第i个链表中的表结点个数是顶点Vi的出度。

在有向图的邻接表中不易找到指向该顶点的弧。

有向图的逆邻接表

在有向图的逆邻接表中，对每个顶点，链接的是指向该顶点的弧。





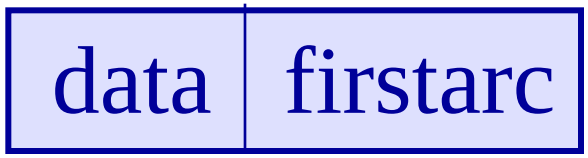
有向图的逆邻接表中，第i个链表中的表结点个数是顶点Vi的入度。

图的邻接表存储表示：

弧的结点结构
(表结点)


```
typedef struct ArcNode {
    int  adjvex;      // 该弧所指向的顶点的位置
    struct ArcNode  *nextarc;
                      // 指向下一条弧的指针
    InfoType  *info;  // 该弧相关信息的指针
} ArcNode;
```

顶点的结点结构(头结点)




```
typedef struct VNode {
    VertexType  data;   // 顶点信息
    ArcNode  *firstarc;
                        // 指向第一条依附该顶点的弧
} VNode, AdjList[MAX_VERTEX_NUM];
```

图的结构定义

```
typedef struct {
    AdjList  vertices;   // 表头结点向量
    int      vexnum, arcnum;
                         // 图的当前顶点数和弧数
    int      kind;       // 图的种类标志
} ALGraph;
```

- **优点：**

  当边数<<顶点个数的平方($m<<n^2$)时，节省存储空间；

  运算方便，容易找到任意顶点的第一个邻接点和下一个邻接点。

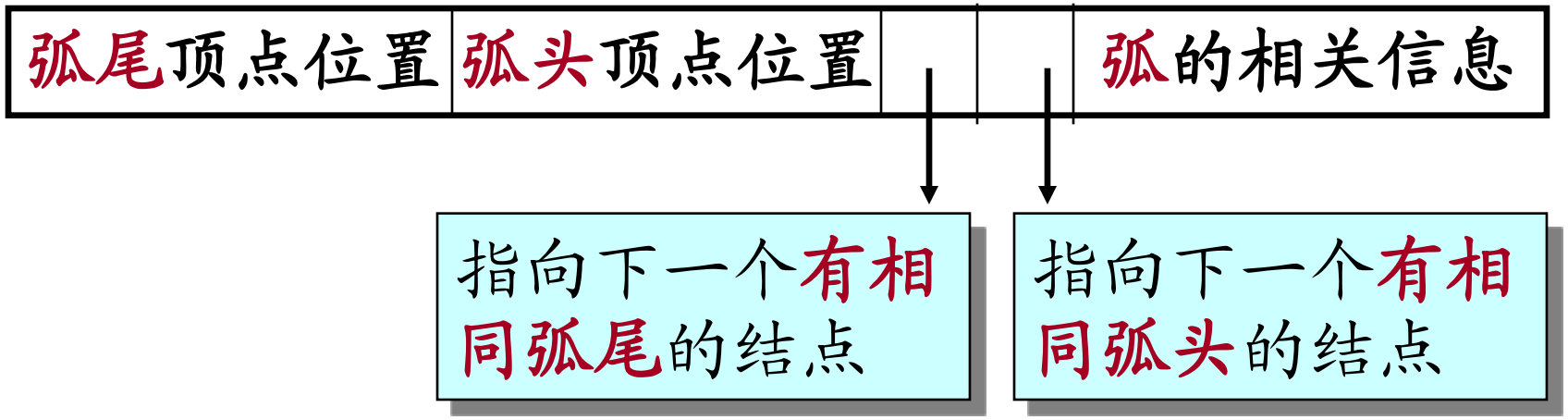
- **缺点：**

  要判定任意两个顶点之间是否有边或弧相连时，则需扫描第i个或第j个链表，不及邻接矩阵方便。

三、有向图的十字链表存储表示

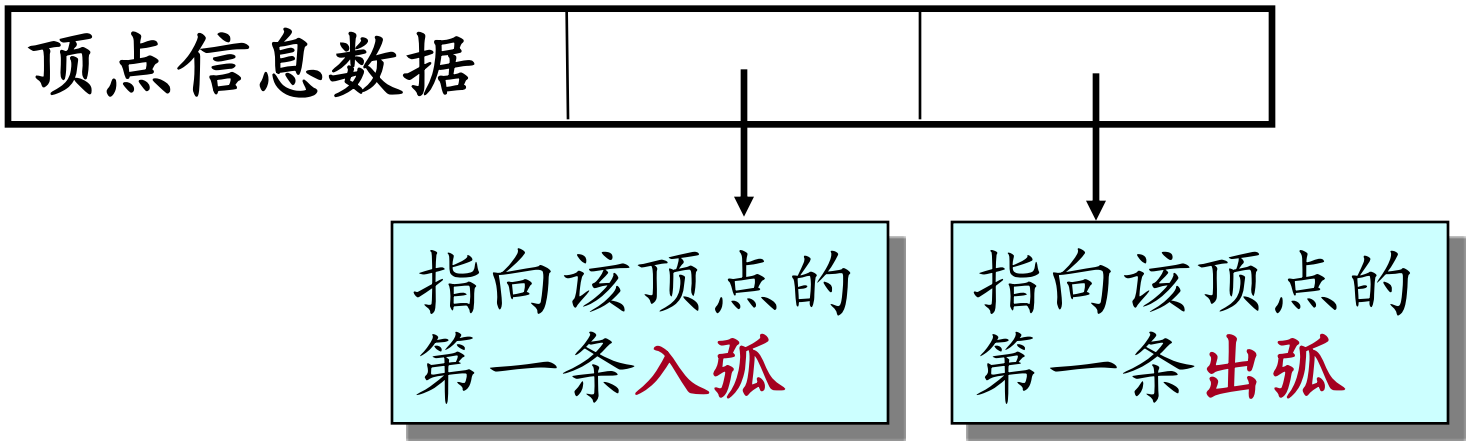
十字链表是图的另一种链式存贮结构，可以看成是将有向图的邻接表和逆邻接表结合起来得到的一种链表。在十字链表中，对应于有向图的每个顶点和每一条弧都有一个结点。

弧的结点结构(弧结点)




```
typedef struct ArcBox {        // 弧的结构表示
    int tailvex, headvex;
    InfoType  *info;
    struct ArcBox *tlink, *hlink;
} ArcBox;
```

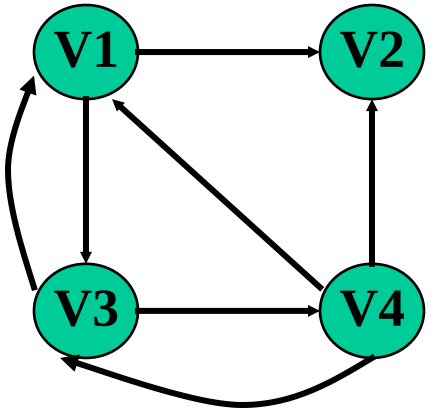
顶点的结点结构(顶点结点)

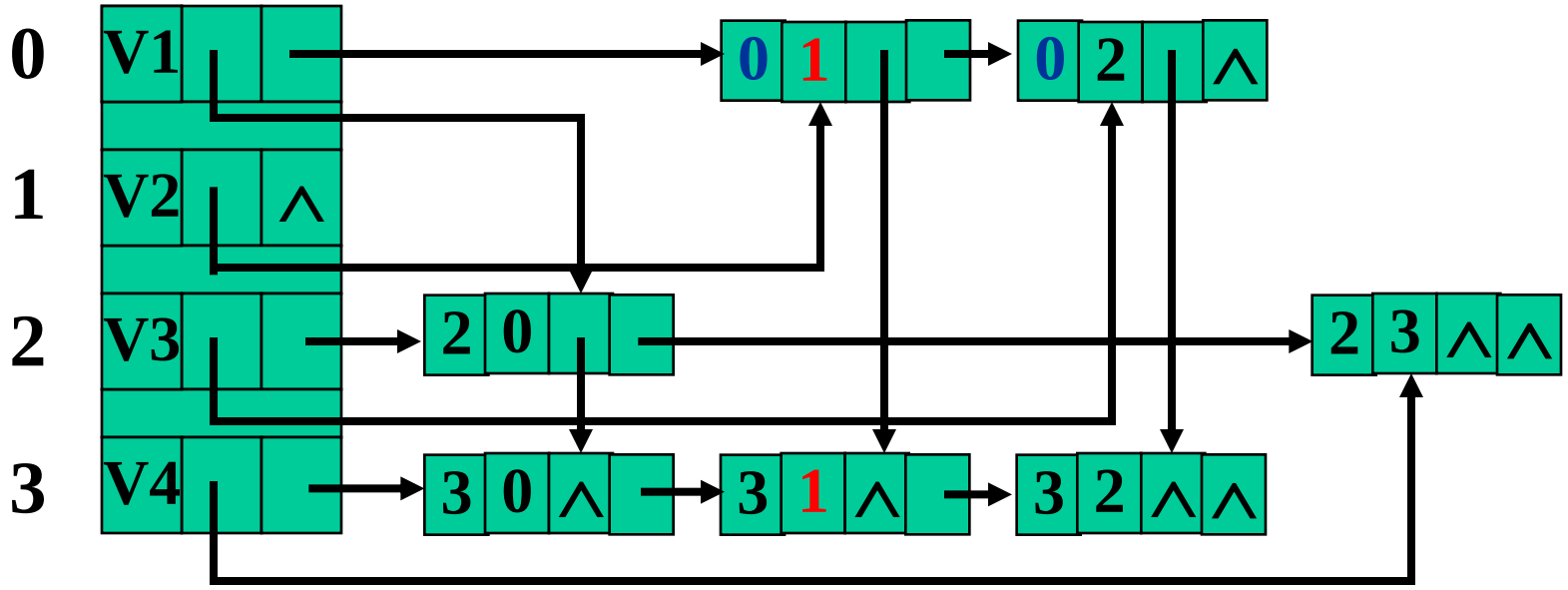



```
typedef struct VexNode { // 顶点的结构表示
    VertexType  data;
    ArcBox  *firstin, *firstout;
} VexNode;
```

有向图的结构表示(十字链表)

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


V1
V2
V3
V4

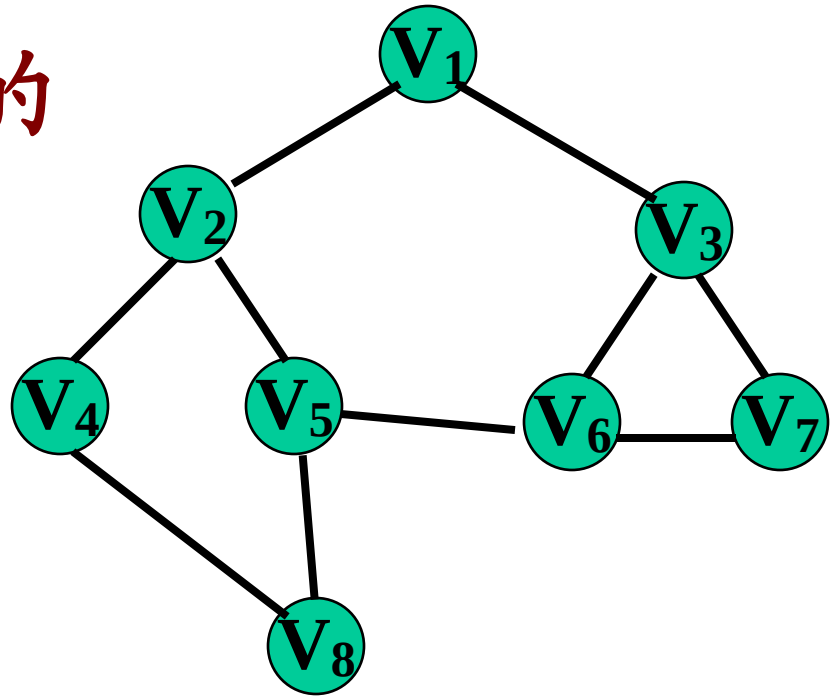

0
1
2
3
V1
V2
V3
V4
0 1
0 2 ∧
2 0
2 3 ∧ ∧
3 0 ∧
3 1 ∧
3 2 ∧ ∧

- **优点：**

  容易找到以V_i为尾的弧，也容易找到以V_i为头的弧，因而容易求得顶点的出度和入度。

7.3 图的遍历

从图中某个顶点出发，访遍图中其余顶点，并且使图中的每个顶点仅被访问一次的过程，叫做**图的遍历**。



图的遍历要比树的遍历复杂得多。因为图中任一顶点都可能和其余的顶点相邻接。所以，在访问了某个顶点之后，可能沿着某条搜索路径又回到该顶点上。为了避免同一顶点被多次访问，在遍历图的过程中，必须记下每个已访问过的顶点。

为此，我们可设一个**辅助数组** **visited[0..n-1]**，它的初始值置为“假”或者“0”，一旦访问了顶点Vi，便置visited[i]为“真”或者被访问时的次序号。

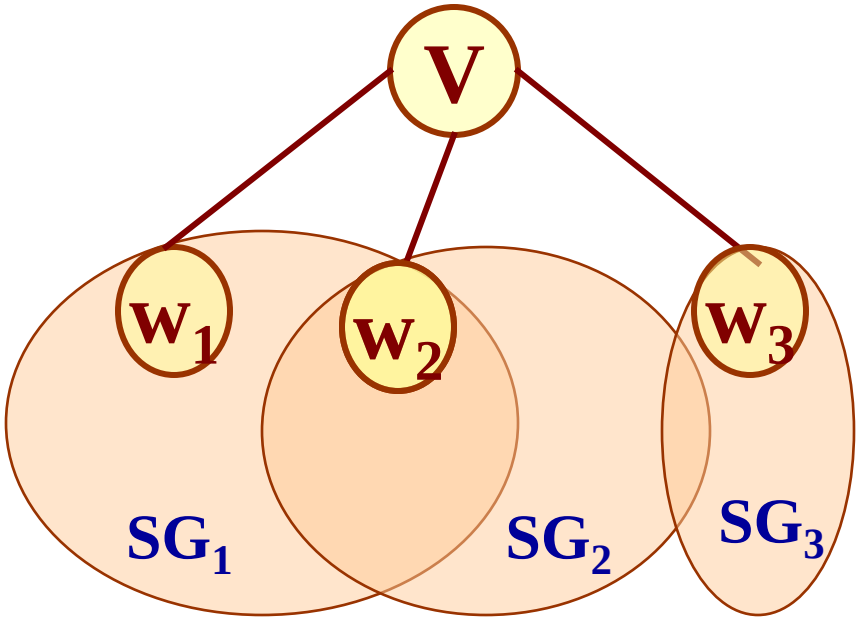
通常有两条遍历图的路径:

- 深度优先搜索（纵向优先搜索）
- 广度优先搜索（横向优先搜索）

一、深度优先搜索遍历图

连通图的深度优先搜索遍历

从图中某个顶点V_0出发，访问此顶点，然后**依次从V_0的各个未被访问的邻接点出发深度优先搜索遍历图**，直至图中所有和V_0有路径相通的顶点都被访问到。



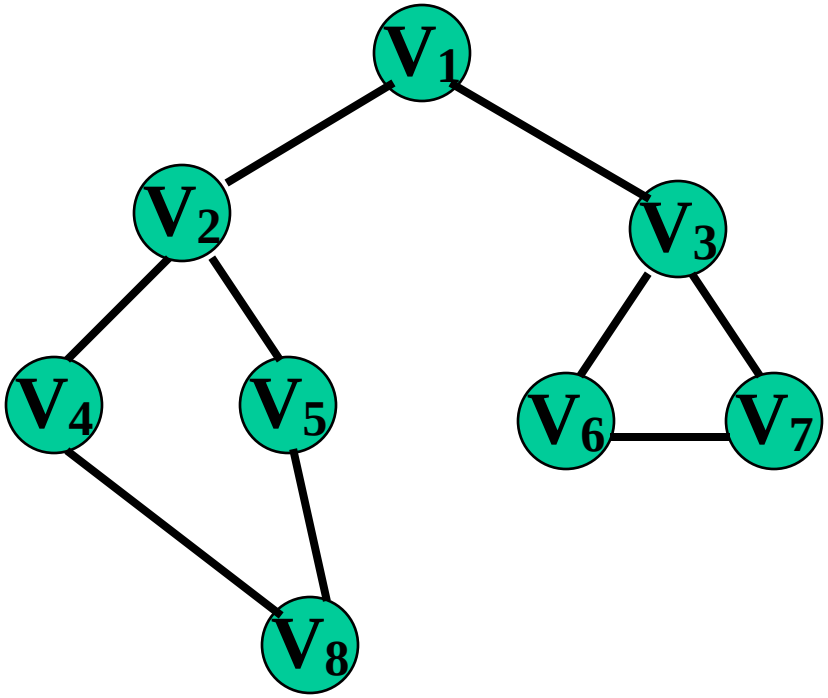
W_1、W_2和W_3 均为 V 的邻接点，SG_1、SG_2 和 SG_3 分别为含顶点W_1、W_2和W_3 的子图。

访问顶点 V;

for (W_1、W_2、W_3)

若该邻接点W未被访问，

则从它出发进行深度优先搜索遍历。



深度优先搜索结果:

$V_1 \rightarrow V_2 \rightarrow V_4 \rightarrow V_8 \rightarrow V_5 \rightarrow V_3 \rightarrow V_6 \rightarrow V_7$

从上页的图解可见：

1. 深度优先搜索遍历连通图的过程类似于树的先根遍历；

2. 如何判别V的邻接点是否被访问？

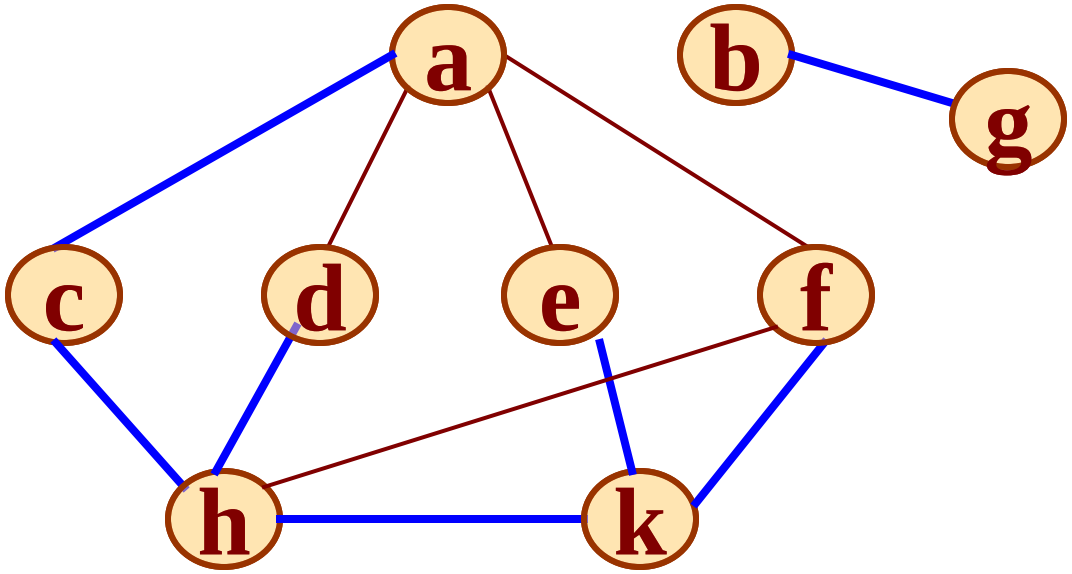
解决的办法是：为每个顶点设立一个“访问标志 visited[w]”。

```
void DFS(Graph G, int v) {
        // 从顶点v出发，深度优先搜索遍历连通图 G

    VisitFunc(v);       // 访问第v 个结点
    visited[v] = TRUE;
    for(w=FirstAdjVex(G, v);          // 调用基本操作
          w!=0; w=NextAdjVex(G,v,w))
       if (!visited[w])  DFS(G, w);
                // 对v的尚未访问的邻接顶点w,递归调用DFS

} // DFS
```

非连通图的深度优先搜索遍历

首先将图中每个顶点的访问标志设为FALSE，之后，从图中某个顶点v_0出发，访问此顶点，然后依次从v_0的未被访问的邻接点出发深度优先遍历图，直至图中所有和v_0有路径相通的顶点都被访问到为止；若此时图中尚有顶点未被访问，则另选图中一个未曾访问的顶点作起始点，重复上述过程，直至图中所有顶点都被访问到为止。

例如：



	0	1	2	3	4	5	6	7	8
	(a)	(b)	(c)	(d)	(e)	(f)	(g)	(h)	(k)
访问标志：	T	T	T	T	T	T	T	T	T

访问次序：	a	c	h	d	k	f	e	b	g

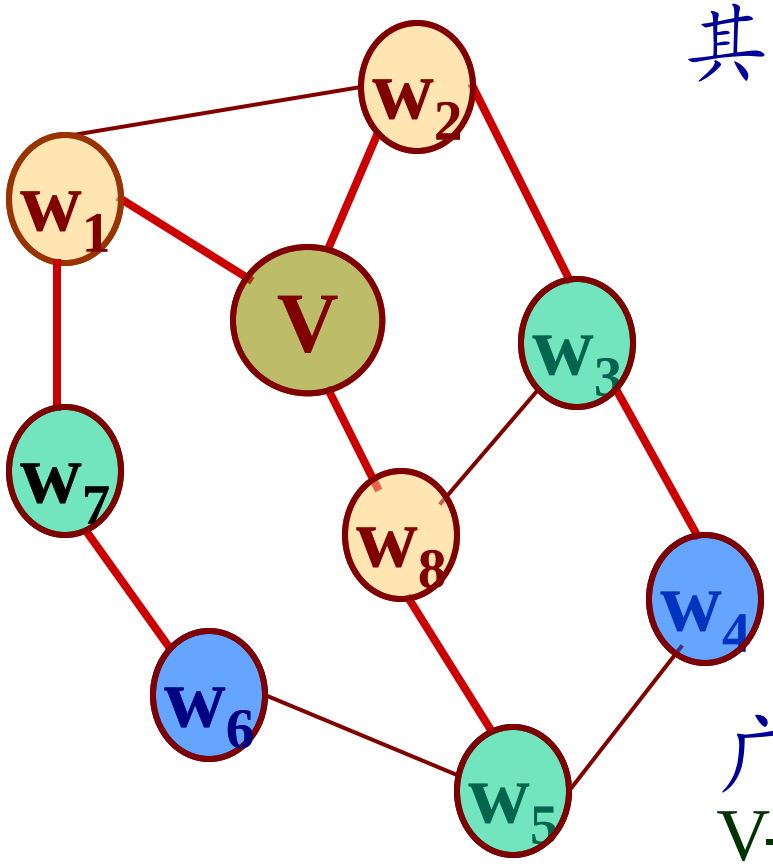
```
void DFSTraverse(Graph G,
                 Status (*Visit)(int v)) {
                                  // 对图 G 作深度优先遍历。
    VisitFunc = Visit;
                  // 使用全局变量，使DFS不必设函数指针参数
    for (v=0; v<G.vexnum; ++v)
        visited[v] = FALSE;      // 访问标志数组初始化
    for (v=0; v<G.vexnum; ++v)
        if (!visited[v])  DFS(G, v);
                                  // 对尚未访问的顶点调用DFS
}
```

二、广度优先搜索遍历图

从图中的某个顶点V_0出发，并在访问此顶点之后依次访问V_0的所有未被访问过的邻接点，之后按这些顶点被访问的先后次序依次访问它们的邻接点，直至图中所有和V_0有路径相通的顶点都被访问到。

若此时图中尚有顶点未被访问，则另选图中一个未曾被访问的顶点作起始点，重复上述过程，直至图中所有顶点都被访问到为止。

对连通图，从起始点V到其余各顶点必定存在路径。



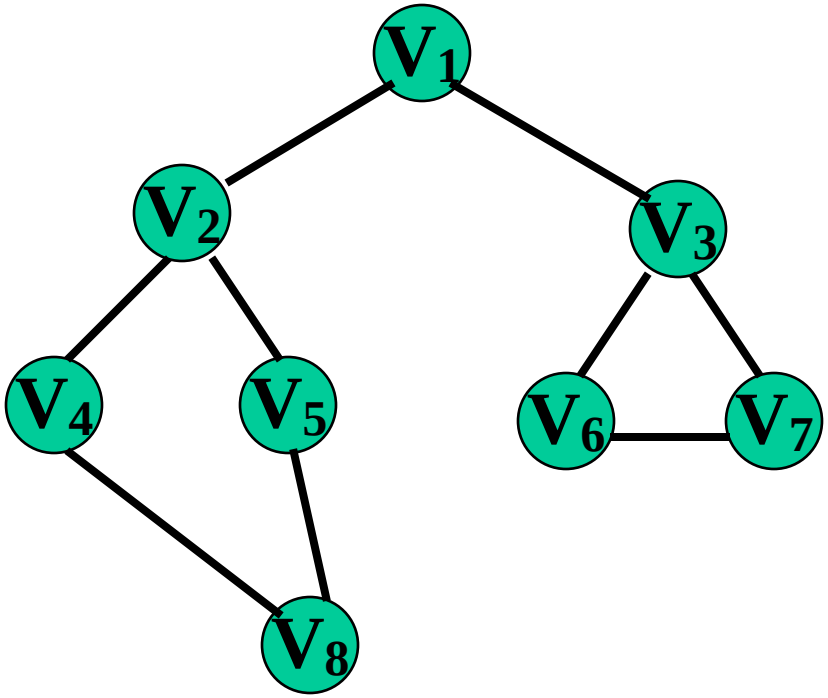
其中，$V->w_1$, $V->w_2$, $V->w_8$ 的路径长度为1;

$V->w_7$, $V->w_3$, $V->w_5$ 的路径长度为2;

$V->w_6$, $V->w_4$ 的路径长度为3。

广度优先搜索结果:

V->w1->w2->w8 ->w7->w3->w5 - >w6->w4



广度优先搜索结果:

$V_1 \rightarrow V_2 \rightarrow V_3 \rightarrow V_4 \rightarrow V_5 \rightarrow V_6 \rightarrow V_7 \rightarrow V_8$

广度优先搜索图的过程是以v为起始点，由近至远，依次访问和v有路径相通且路径长度为1，2，……的顶点。**类似于树的按层次遍历过程。**

和深度优先搜索类似，在遍历的过程中也需要一个访问**标志数组**。并且，为了顺次访问路径长度为2、3、…的顶点，需附设**队列**以存储以被访问的路径长度为1，2，…的顶点。

```
void BFSTraverse(Graph G,
     Status (*Visit)(int v)) // 对图 G 作广度优先遍历
{    for (v=0; v<G.vexnum; ++v)
        visited[v] = FALSE;   // 初始化访问标志
     InitQueue(Q);                  // 置空的辅助队列Q
     for ( v=0;  v<G.vexnum;  ++v )
          if ( !visited[v]) {          // v 尚未访问
            … …
        }
} // BFSTraverse
```

```
visited[v] = TRUE;  Visit(v);    // 访问v
EnQueue(Q, v);                   // v入队列
while (!QueueEmpty(Q))  {//循环结束时访问整个连通子图
   DeQueue(Q, u);                // 队头元素出队并置为u
   for(w=FirstAdjVex(G, u); w!=0;
                    w=NextAdjVex(G,u,w))
      if ( ! visited[w])  {
         visited[w]=TRUE;  Visit(w);
         EnQueue(Q, w);          // 访问的顶点w入队列
      } // if
} // while
```

三、遍历应用举例

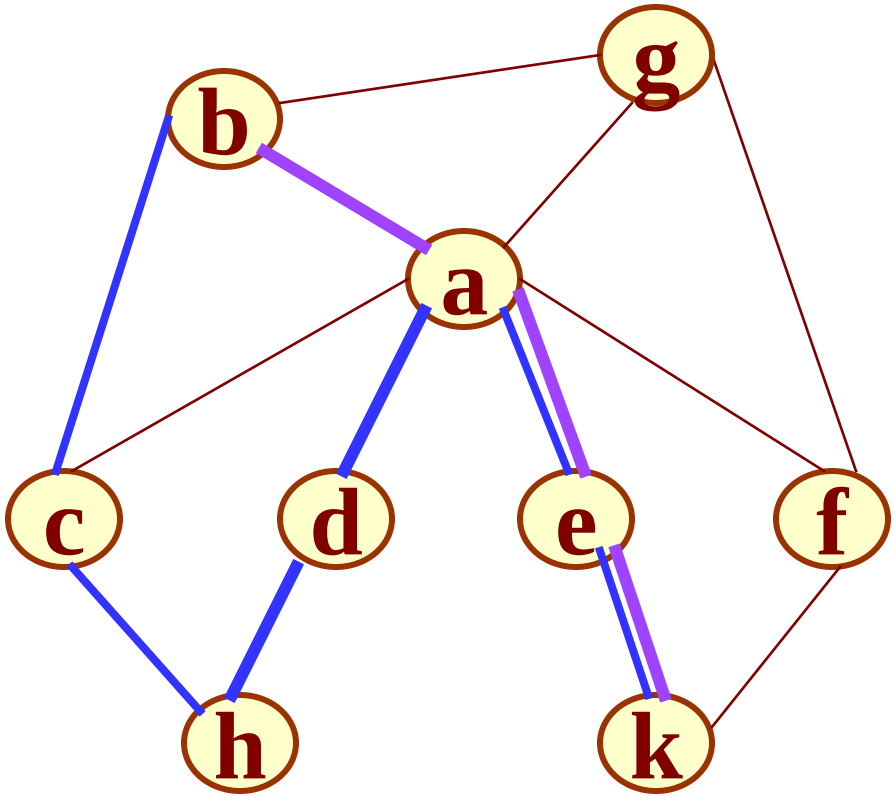
1. 求一条从顶点 i 到顶点 s 的简单路径

2. 求两个顶点之间的一条路径长度最短的路径

1. 求一条从顶点 i 到顶点 s 的简单路径

例如：求从顶点 b 到顶点 k 的一条简单路径。

从顶点 b 出发进行深度优先搜索遍历。



假设找到的第一个邻接点是c，则得到的结点访问序列为：

b c h d a e **k** f g,

假设找到的第一个邻接点是a，则得到的结点访问序列为：**b** a d h c e **k** f g。

结论:

(1). 从顶点 i 到顶点 s，若存在路径，则从顶点 i 出发进行深度优先搜索，必能搜索到顶点 s。

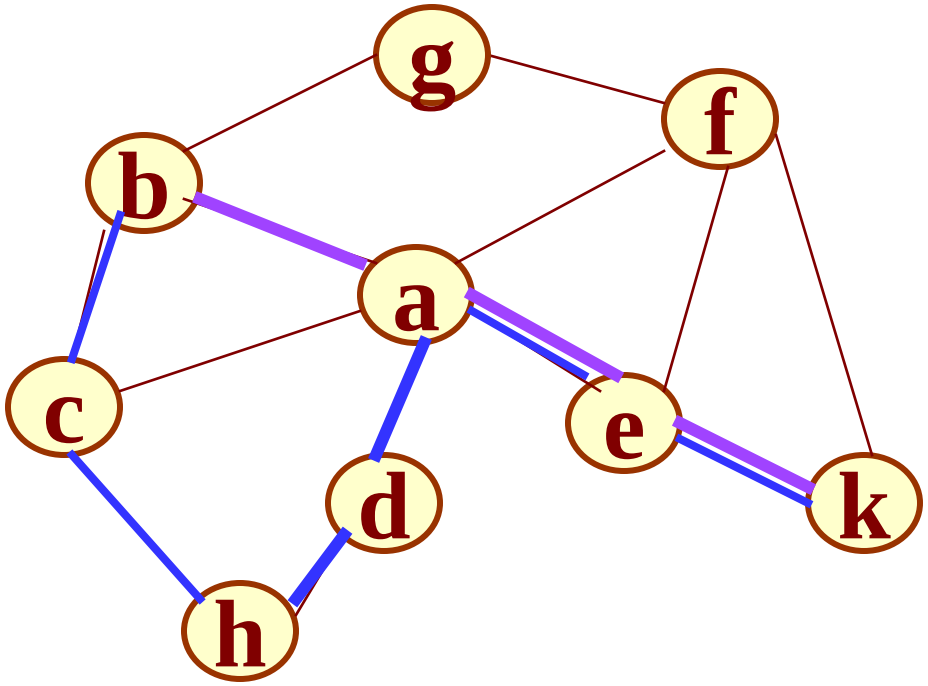
(2). 遍历过程中搜索到的顶点不一定是路径上的顶点。

(3). 由它出发进行的深度优先遍历已经完成的顶点不是路径上的顶点。

```
void DFSearch( int v, int s, char *PATH) {
 // 从第v个顶点出发递归地深度优先遍历图G，求得一条从v到s的简单路径，并记录在PATH中
  visited[v] = TRUE;                    // 访问第 v 个顶点
    Append(PATH, getVertex(v));   // 第v个顶点加入路径
  for (w=FirstAdjVex(v);  w!=0 && !found;
                          w=NextAdjVex(v,w) )
     if (w=s) { found = TRUE;  Append(PATH, w); }
     else  if (!visited[w])  DFSearch(w, s, PATH);
  if (!found)  Delete (PATH); // 没有到S的路径，删除
}
```

2. 求两个顶点之间的一条路径长度最短的路径

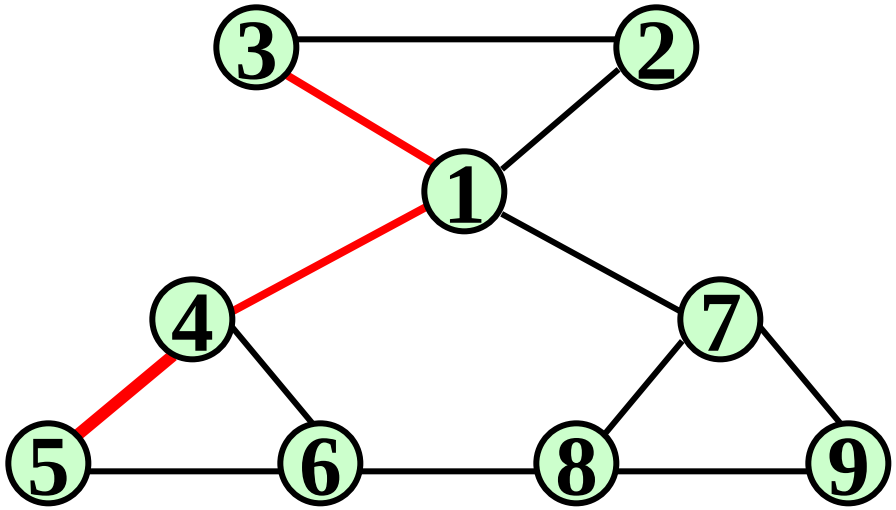
若两个顶点之间存在多条路径，则其中必有一条路径长度最短的路径。如何求得这条路径？（这儿只考虑两顶点之间所含边的数目最少）



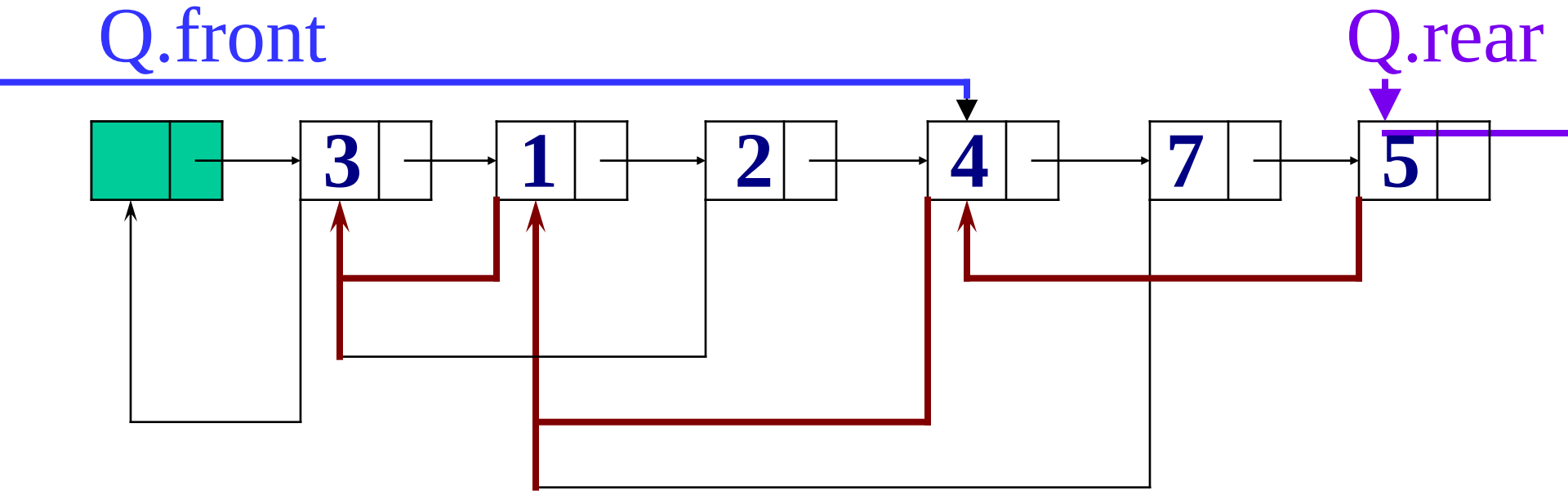
深度优先搜索访问顶点的**次序**取决于图的**存储结构**，而广度优先搜索访问顶点的次序是按“路径长度”渐增的次序。

因此，**求路径长度最短的路径可以基于广度优先搜索遍历进行。**

例如:求下图中顶点 3 至顶点 5 的一条最短路径。



假设使用链队列，需要修改链队列的结点结构及其入队列和出队列的算法



1) **将链队列的结点改为“双链”结点。**即结点中包含next 和prior两个指针;

2) **修改入队列的操作。**插入新的队尾结点时，令其prior域的指针指向刚刚出队列的结点，即当前的队头指针所指结点;

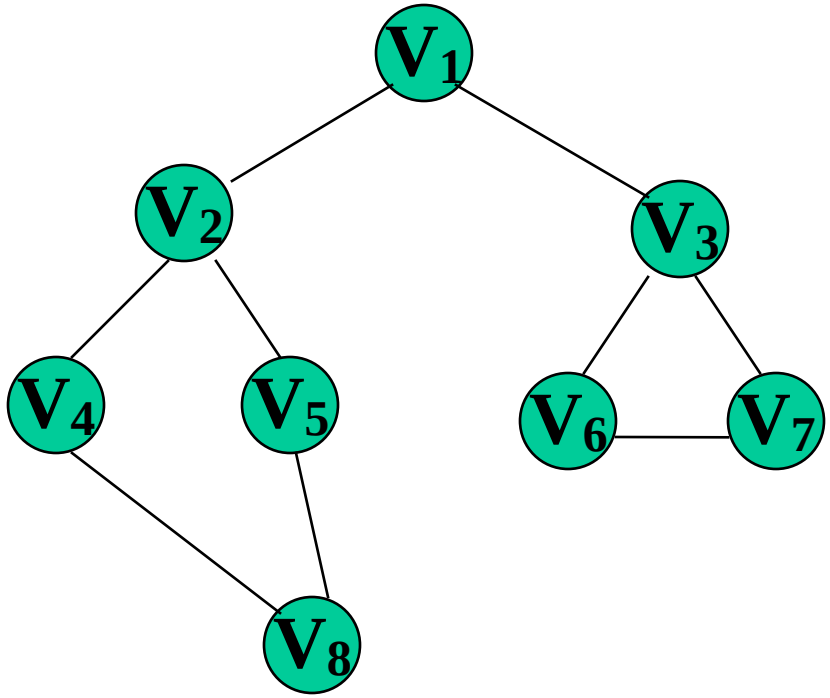
3) **修改出队列的操作。**出队列时，仅移动队头指针，而不将队头结点从链表中删除。

```
typedef  DuLinkList QueuePtr;
void InitQueue(LinkQueue& Q) {
   Q.front=Q.rear=(QueuePtr)malloc(sizeof(QNode));
   Q.front->next = Q.rear->next = NULL
}
void EnQueue( LinkQueue& Q, QelemType e ) {
    p = (QueuePtr) malloc (sizeof(QNode));
    p->data = e;  p->next = NULL;
    p->prior = Q.front
    Q.rear->next = p;  Q.rear = p;
}
void DeQueue( LinkQueue& Q, QelemType& e ) {
   Q.front = Q.front->next;  e = Q.front->data
}
```

7.4 图的连通性和生成树

1、连通图

从图中任一顶点出发，进行深度优先搜索或广度优先搜索，便可遍历完图中所有顶点，这个图就是连通图。



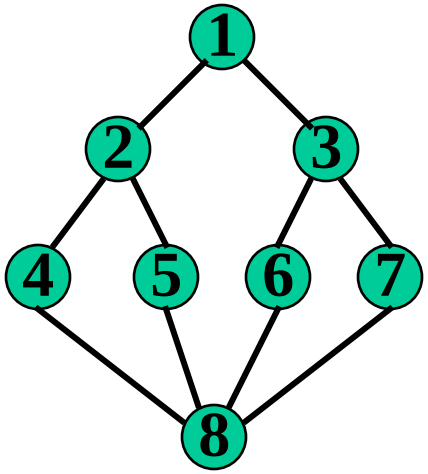
2、图的生成树

设G=(V,E)是个连通图，而E(G)为连通图中所有边的集合，若从图中任一顶点开始作一次搜索过程，则将E(G)分成两个集合：T(G)和B(G)。T(G)是遍历图时走过的边的集合，B(G)是剩余边的集合，T(G)和图中所有顶点一起构成连通图G的极小连通子树，称为图G的生成树。

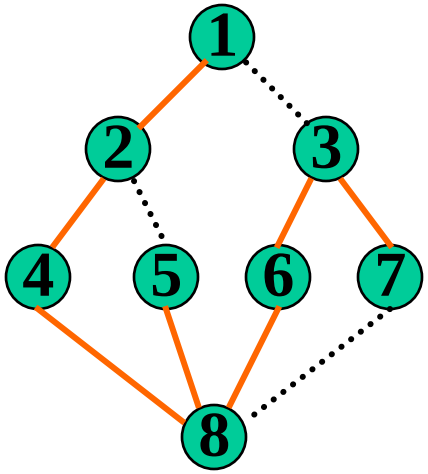
图的生成树不唯一，从不同结点出发进行搜索，可以得到不同的生成树。

深度优先生成树：由深度优先搜索得到的生成树。

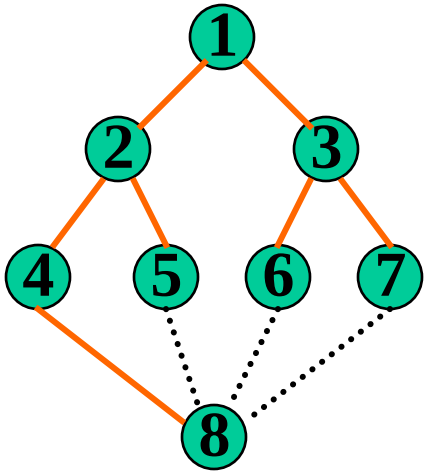
广度优先生成树：由广度优先搜索得到的生成树。



无向图



深度优先生成树



广度优先生成树

3、最小代价生成树

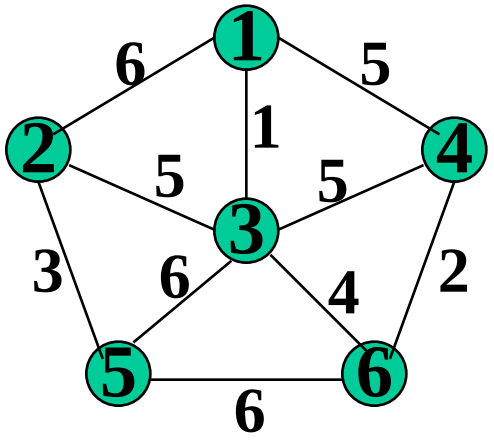
任何具有n个顶点的连通图，至少有n-1条边，而且所有具有n-1条边的连通图都是树。若在连通图的各条边上赋以权值即网络表示相应的代价，那么，则从连通图中选择一棵总代价最小的树，即为最小代价生成树。

4、最小代价生成树的应用

问题的提出：

假设要在 n 个城市之间建立通讯联络网，则连通 n 个城市只需要修建 $n-1$ 条线路，**如何在最节省经费的前提下建立**这个**通讯网**？

在每两个城市之间都可以设置一条线路，相应地都要付出一定的经济代价。n个城市之间，最多可能设置 n(n-1)/2 条线路，那么，如何在这些可能的线路中选择 n-1 条，以使总的耗费最少呢？



可以用连通网来表示n个城市以及n个城市之间可能设置的通信线路，其中，网的顶点表示城市，边表示两城市之间的线路，赋予边的权值表示相应的代价。对于n个顶点的连通网可以建立许多不同的生成树，每一棵生成树都可以是一个通讯网。现在，我们要选择这样一棵生成树，也就是总的耗费最少。

该问题等价于:

构造网的一棵最小生成树，即:

在 e 条带权的边中选取 n-1 条边（不构成回路），使“**权值之和**”为最小。

算法一：普里姆 (Prim) 法

算法二：克鲁斯卡尔 (Kruskal) 法

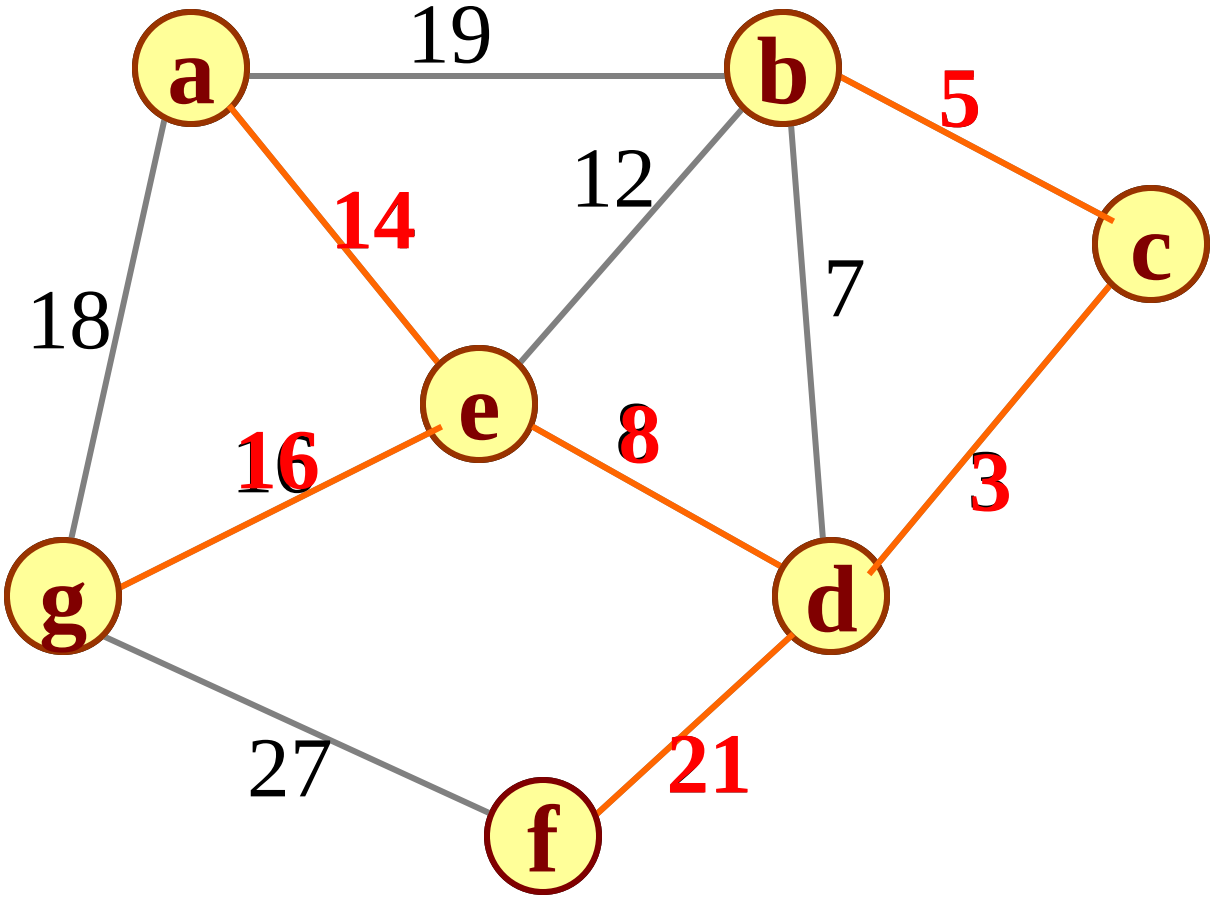
基本思想:

按权值非递减次序构造最小代价生成树。

（1）普里姆算法的基本思想

取图中任意一个顶点 v 作为生成树的根，之后往生成树上添加新的顶点 w。**在添加的顶点 w 和已经在生成树上的顶点v 之间必定存在一条边，并且该边的权值在所有连通顶点 v 和 w 之间的边中取值最小**。之后继续往生成树上添加顶点，直至生成树上含有 n 个顶点为止。

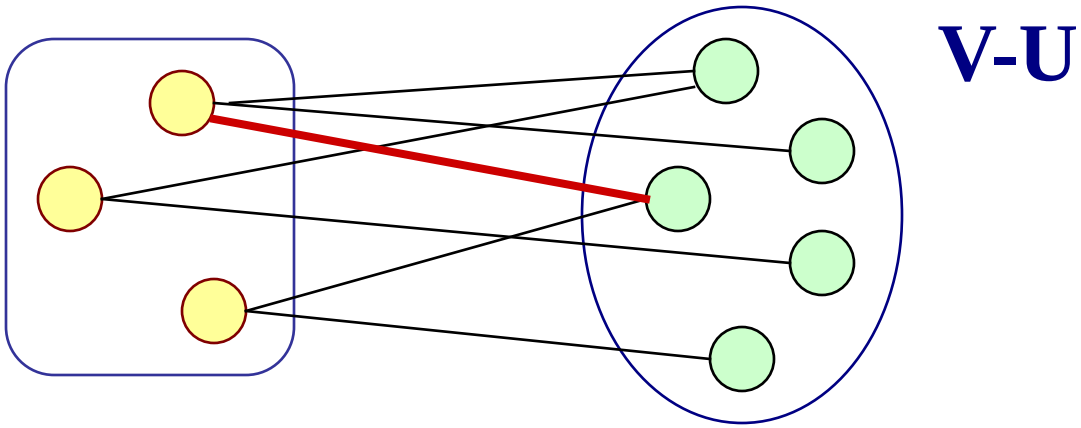
例如:



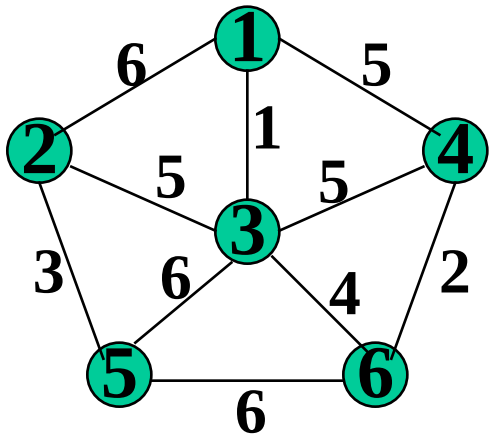
所得生成树权值和 = 14+8+3+5+16+21 = 67

一般情况下所添加的顶点应满足下列条件：

在生成树的构造过程中，图中 n 个顶点分属两个集合：已落在生成树上的顶点集 **U** 和尚未落在生成树上的顶点集**V-U**，则应**在所有连通U中顶点和V-U中顶点的边中选取权值最小的边**。

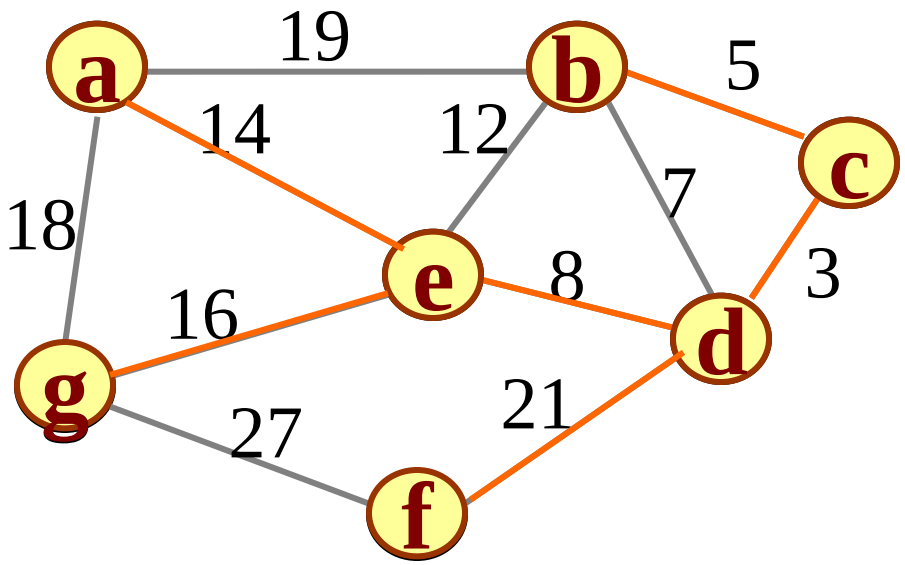


设置一个辅助数组，对当前V－U集中的每个顶点，记录和顶点集U中顶点相连接的代价最小的边:

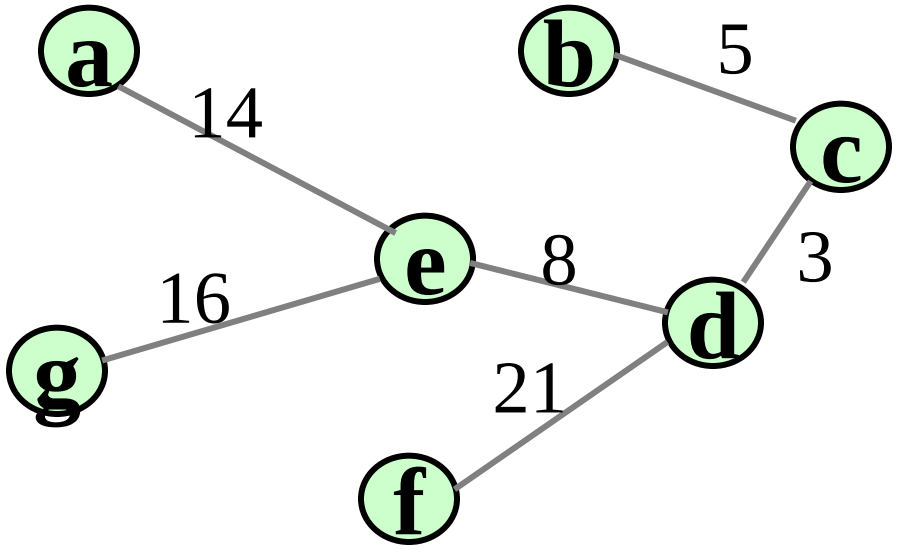



```
struct {
    VertexType  adjvex;  // U集中的顶点序号
    VRType      lowcost; // 边的权值
} closedge [MAX_VERTEX_NUM];
```

例如：

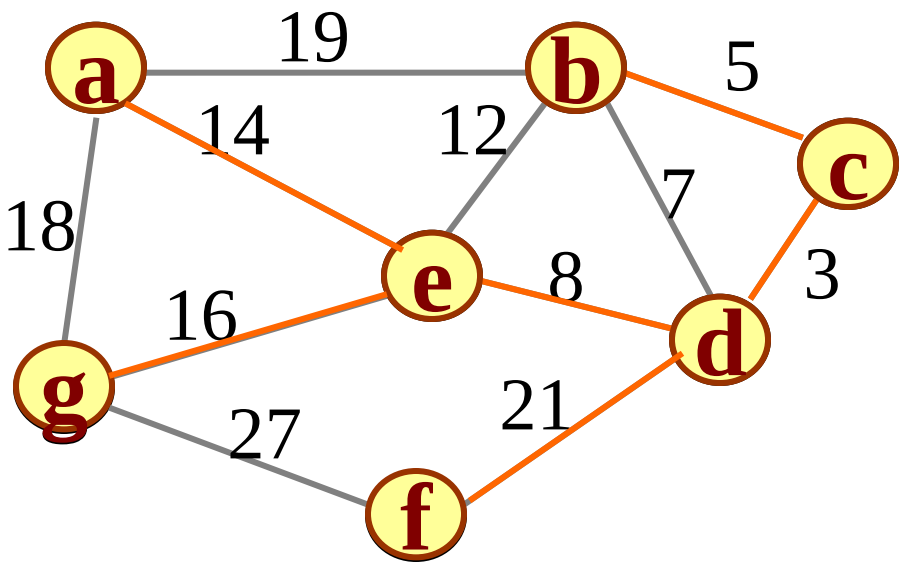


closedge	0 (a)	1 (b)	2 (c)	3 (d)	4 (e)	5 (f)	6 (g)
Adjvex		c	d	e	a	d	e
Lowcost		**5**	**3**	**8**	**14**	21	16


a
b
5
14
c
e
8
3
16
d
g
21
f

```
void MiniSpanTree_P(MGraph G, VertexType u) {
        //用普里姆算法从顶点u出发构造网G的最小生成树
  k = LocateVex ( G, u );
  for ( j=0; j<G.vexnum; ++j )  // 辅助数组初始化
    if (j!=k)
      closedge[j] = { u, G.arcs[k][j].adj };
                                  // {adjvex, lowcost}
  closedge[k].lowcost = 0;          // 初始，U = {u}
  for (i=1; i<G.vexnum; ++i) {//选择其余N-1个顶点
    ……继续向生成树上添加顶点; }
}
```

```
k = minimum(closedge);
                                    // 求出加入生成树的下一个顶点(k)
    // 此时closedge[k].lowcost=
    // MIN{closedge[v_i ].lowcost|closedge[v_i].lowcost>0, V_i∈V-U}
printf(closedge[k].adjvex, G.vexs[k]);
                                    // 输出生成树上一条边的两个顶点
closedge[k].lowcost = 0;     // 第k顶点并入U集
for (j=0; j<G.vexnum; ++j)
                                    // 修改其它顶点的最小边
   if (G.arcs[k][j].adj < closedge[j].lowcost)
                                    // 新顶点并入U后重新选择最小边
      closedge[j] = { G.vexs[k], G.arcs[k][j].adj };
```



	a	b	c	d	e	f	g
a	∞	19	∞	∞	**14**	∞	**18**
b	19	∞	**5**	**7**	12	∞	∞
c	∞	5	∞	**3**	∞	∞	∞
d	∞	7	3	∞	8	**21**	∞
e	14	12	∞	**8**	∞	∞	**16**
f	∞	∞	∞	21	∞	∞	**27**
g	18	∞	∞	∞	**16**	27	∞

closedge	0 (a)	1 (b)	2 (c)	3 (d)	4 (e)	5 (f)	6 (g)
Adjvex		c	d	e	a	d	e
Lowcost	0	**0**	**0**	**0**	**0**	21	16

（2）克鲁斯卡尔算法的基本思想

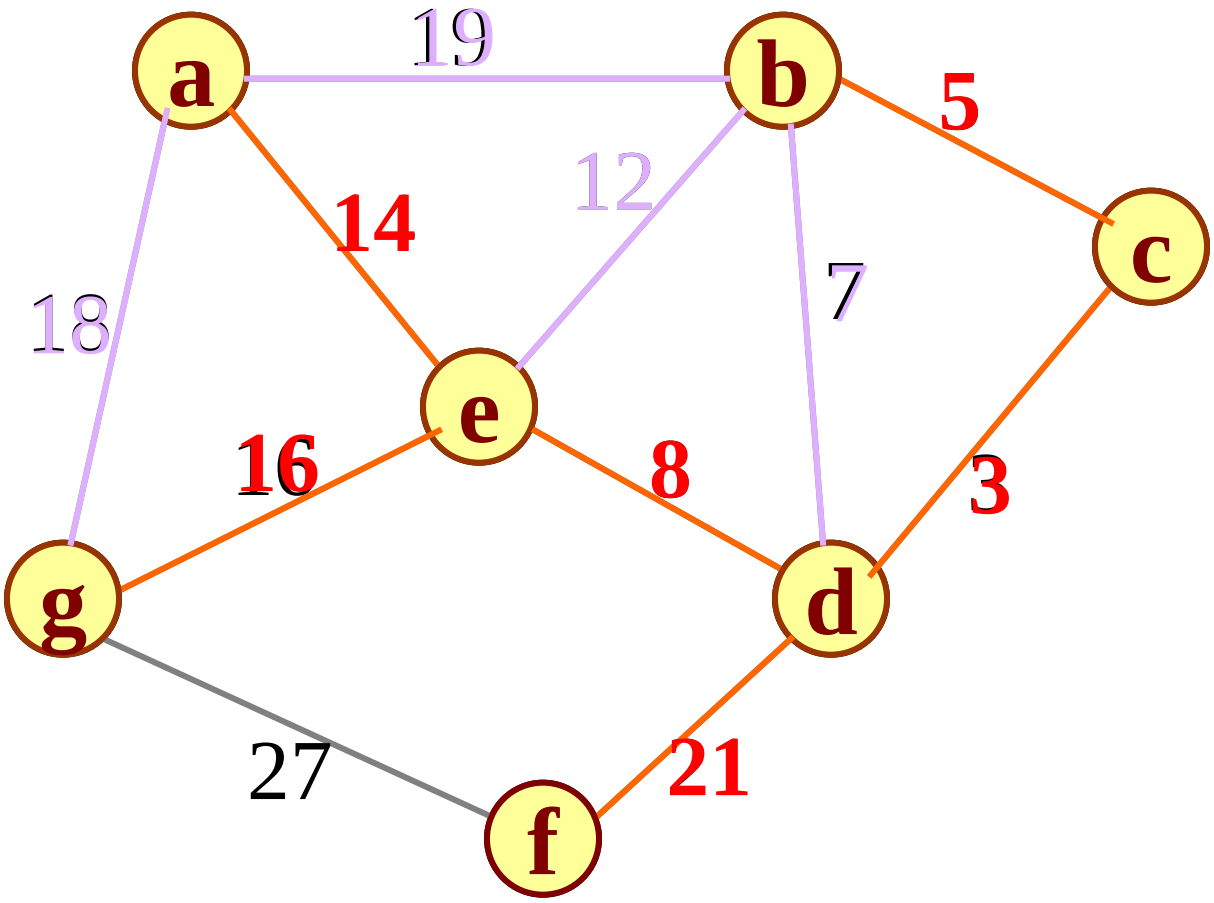
- **考虑问题的出发点：**

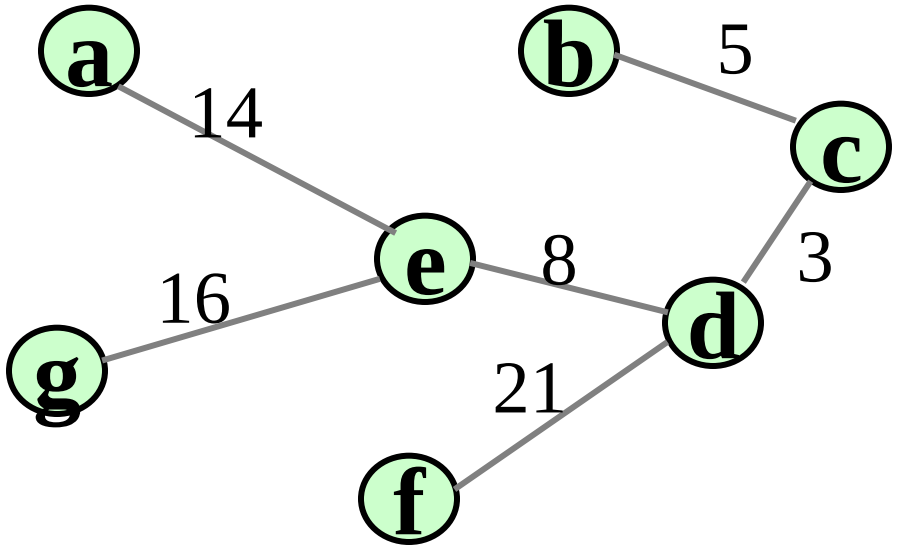
  为使生成树上边的权值之和达到最小，则应使生成树中每一条边的权值尽可能地小。

- **具体做法：**

  先构造一个只含 n 个顶点的子图 SG，然后从权值最小的边开始，若它的添加不使SG中产生回路，则在 SG 上加上这条边，如此重复，直至加上 n-1 条边为止。

例如:




a
b
5
14
c
e
8
3
16
d
g
21
f

算法描述:

构造非连通图 ST=(V,{ }); //v表示顶点的集合，

//{ }表示边的集合，首先是空

```
k = i = 0;                    // k 计选中的边数
while (k<n-1) {
    ++i;
    检查边集 E 中第 i 条权值最小的边(u,v);
    若(u,v)加入ST后不使ST中产生回路，
    则 输出边(u,v); 且 k++;
}
```

(3) 比较两种算法

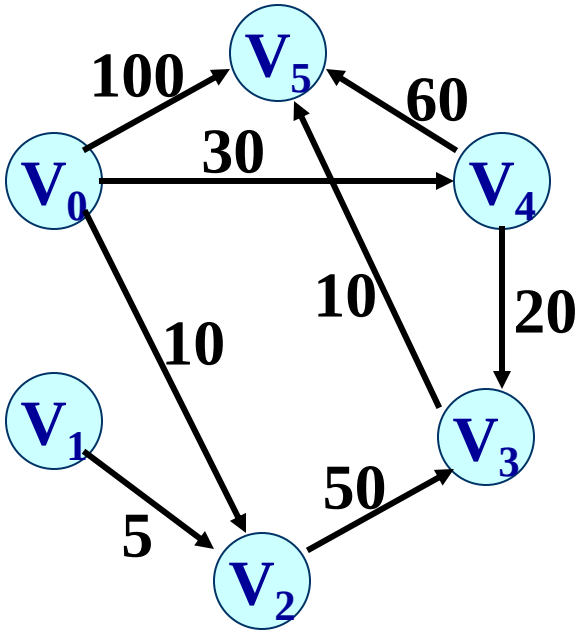
算法名	普里姆算法	克鲁斯卡尔算法
时间复杂度	$O(n^2)$	O(eloge)
适应范围	稠密图	稀疏图

7.5 两点之间的最短路径问题

- 求从某个源点到其余各点的最短路径
- 每一对顶点之间的最短路径

1、从某个源点到其余各点的最短路径

带权有向图中从V_0到其余各顶点之间的最短路径:

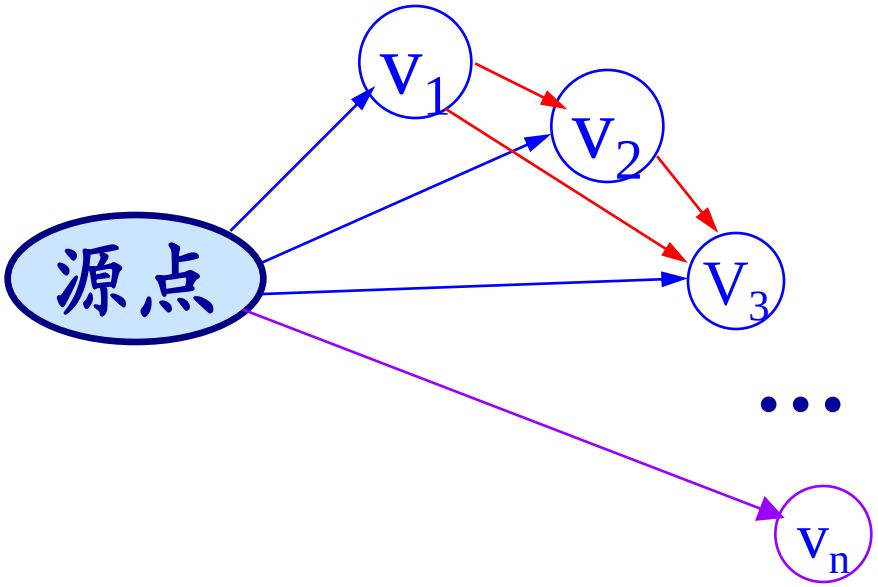


始点	终点	最短路径	路径长度
V_0	V_1	无	
	V_2	(V_0, V_2)	10
	V_3	(V_0, V_4, V_3)	50
	V_4	(V_0, V_4)	30
	V_5	(V_0, V_4, V_3, V_5)	60

如何求从源点到其余各点的最短路径？

算法的基本思想：

按路径长度递增的次序产生最短路径



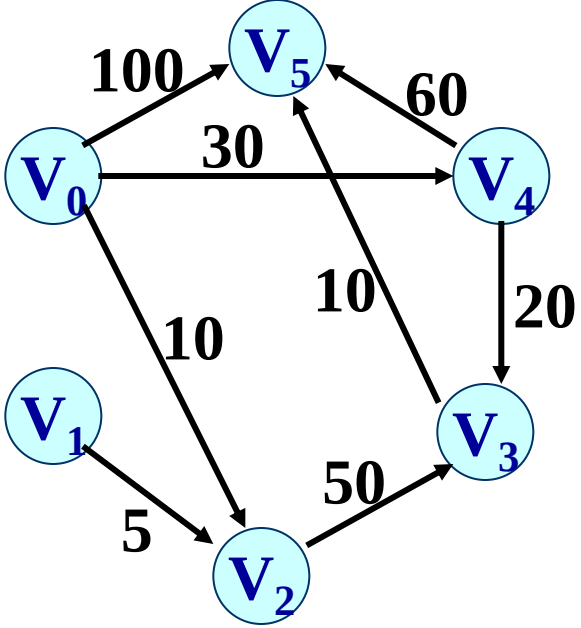
其中，从源点到顶点v_1的最短路径是所有最短路径中长度最短者。

下一条最短路径呢？

再下一条最短路径呢？

•路径长度最短的**最短路径**的特点：

在这条路径上，必定只含一条弧，并且这条弧的权值最小，不妨设(V_0,V_i)。

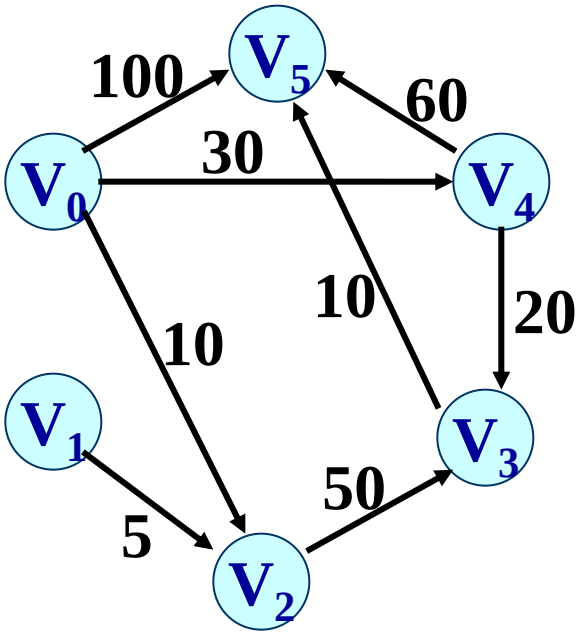


•下一条路径长度次短的**最短路径**的特点：

它只可能有两种情况：或者是直接从源点到该点(只含一条弧)，不妨设(V_0,V_j)；或者是从源点经过顶点v_i，再到达该顶点(由两条弧组成)(V_0,V_i,V_j)。

- 再下一条路径长度次短的**最短路径**的特点:

它可能有三种情况：或者是直接从源点到该点(只含一条弧)，不妨设(V_0,V_k)；或者是从源点经过顶点v_i，再到达该顶点(由两条弧组成)(V_0,V_i,V_k)；或者是从源点经过顶点v_j，再到达该顶点(V_0,V_j,V_k)。



- 其余最短路径的特点:

它或者是直接从源点到该点(只含一条弧);或者是从源点经过已求得最短路径的顶点，再到达该顶点。

一般情况下，假设S为已求得最短路径的终点的集合，则可证明：下一条最短路径（设其终点为x）或者是弧(v,x)，或者是中间只经过S中的顶点而最后到达顶点x的路径。

反证法来证明：

假设此路径上有一个顶点不在S中，则说明存在一条终点不在S而长度比此路径短的路径。但是，这是不可能的。因为我们是按路径长度递增的次序来产生各最短路径的，故长度比此路径短的所有路径均已产生，它们的终点必定在S中，即假设不成立。

求最短路径的迪杰斯特拉算法:

设置辅助数组Dist，其中每个分量Dist[k]表示当前所求得的从源点到其余各顶点 k 的最短路径。

一般情况下，

Dist[k] = <源点到顶点 k 的弧上的权值>

或者 = <源点到其它顶点的路径长度>

+ <其它顶点到顶点 k 的弧上的权值>。

1) 在所有从源点v_0（$v_0 \in S$) 出发的弧中选取一条权值最小的弧(v_0,u)，即为第一条最短路径。

$$Dist[k]=\begin{cases} G.arcs[v0][k] & \text{V0和k之间存在弧} \\ INFINITY & \text{V0和k之间不存在弧} \end{cases}$$

其中的最小值即为最短路径的长度。将该最小值所对应的顶点j加入到S集中（**S=S∪{j}**）。**S集是已求得最短路径的顶点的集合.**

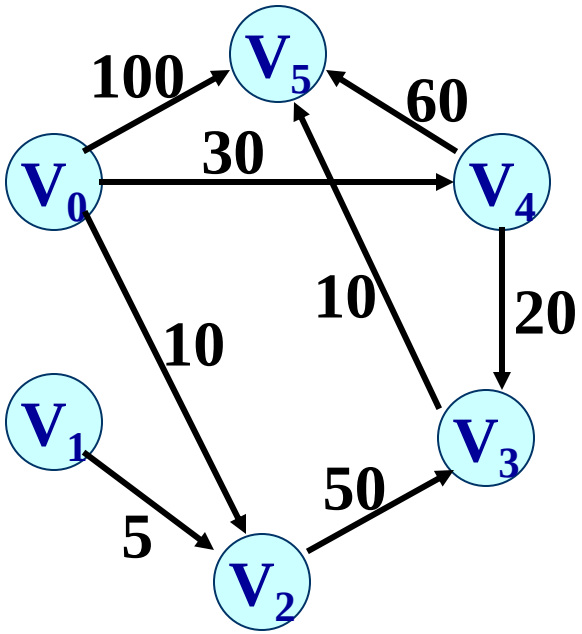
2) 修改其它各顶点的Dist[k]值。

假设求得最短路径的顶点为j，若

Dist[j]+G.arcs[j][k]<Dist[k]

则将 *Dist*[*k*] 改为 *Dist*[*j*]+*G.arcs*[*j*][*k*]。

3) 重复操作1）2) 共n-1次。由此求得其余各顶点的最短路径是依路径长度递增的序列。



$$\begin{pmatrix} \infty & \infty & 10 & \infty & 30 & 100 \\ \infty & \infty & 5 & \infty & \infty & \infty \\ \infty & \infty & \infty & 50 & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty & 10 \\ \infty & \infty & \infty & 20 & \infty & 60 \\ \infty & \infty & \infty & \infty & \infty & \infty \end{pmatrix}$$

邻接矩阵

终点	从 V_0 到各终点的 D 值和最短路径的求解过程				
	i=1	i=2	i=3	i=4	i=5
V_1	∞	∞	∞	∞	∞ 无
V_2	10 (V_0, V_2)				
V_3	∞	60 (V_0, V_2, V_3)	50 (V_0, V_4, V_3)		
V_4	30 (V_0, V_4)	30 (V_0, V_4)			
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)	
V_j	V_2	V_4	V_3	V_5	
S	$\{V_0, V_2\}$	$\{V_0, V_2, V_4\}$	$\{V_0, V_2, V_3, V_4\}$	$\{V_0, V_2, V_3, V_4, V_5\}$	

2、求每一对顶点之间的最短路径

方法1:

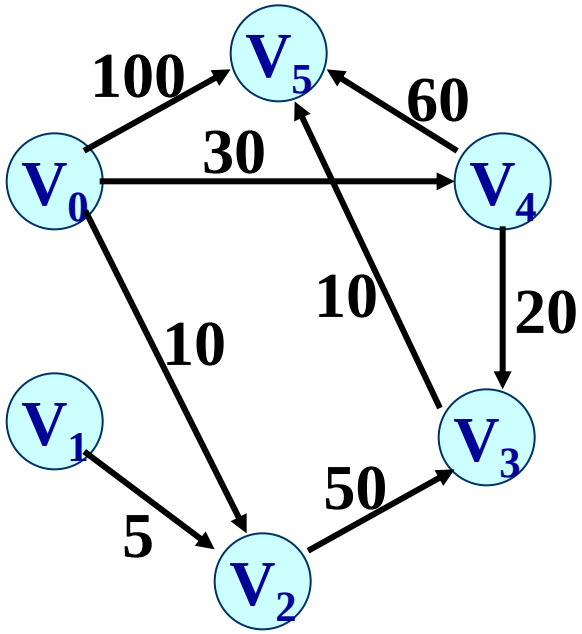
每次以一个顶点为源点，重复执行迪杰斯特拉算法n次。这样，便可求得每一对顶点之间的最短路径。

方法2:

弗洛伊德算法

弗洛伊德算法的基本思想是:

从 v_i 到 v_j 的所有可能存在的路径中，选出一条长度最短的路径。



若<v_i,v_j>存在，则存在路径{v_i,v_j}

// 路径中不含其它顶点

若<v_i,v_1>,<v_1,v_j>存在，则存在路径{v_i,v_1,v_j}

// 路径中所含顶点序号不大于1

若{v_i,…,v_2}, {v_2,…,v_j}存在，

则存在一条路径{v_i, …, v_2, …v_j}

// 路径中所含顶点序号不大于2

…

依次类推，则 vi 至 vj 的最短路径应是上述这些路径中，路径长度最小者。

7.6 拓扑排序

1、什么是拓扑排序

给出有向图G=(V,E)，V里的顶点的线性序列 (V_{i1}, V_{i2} ,..., V_{in})若满足如下条件:

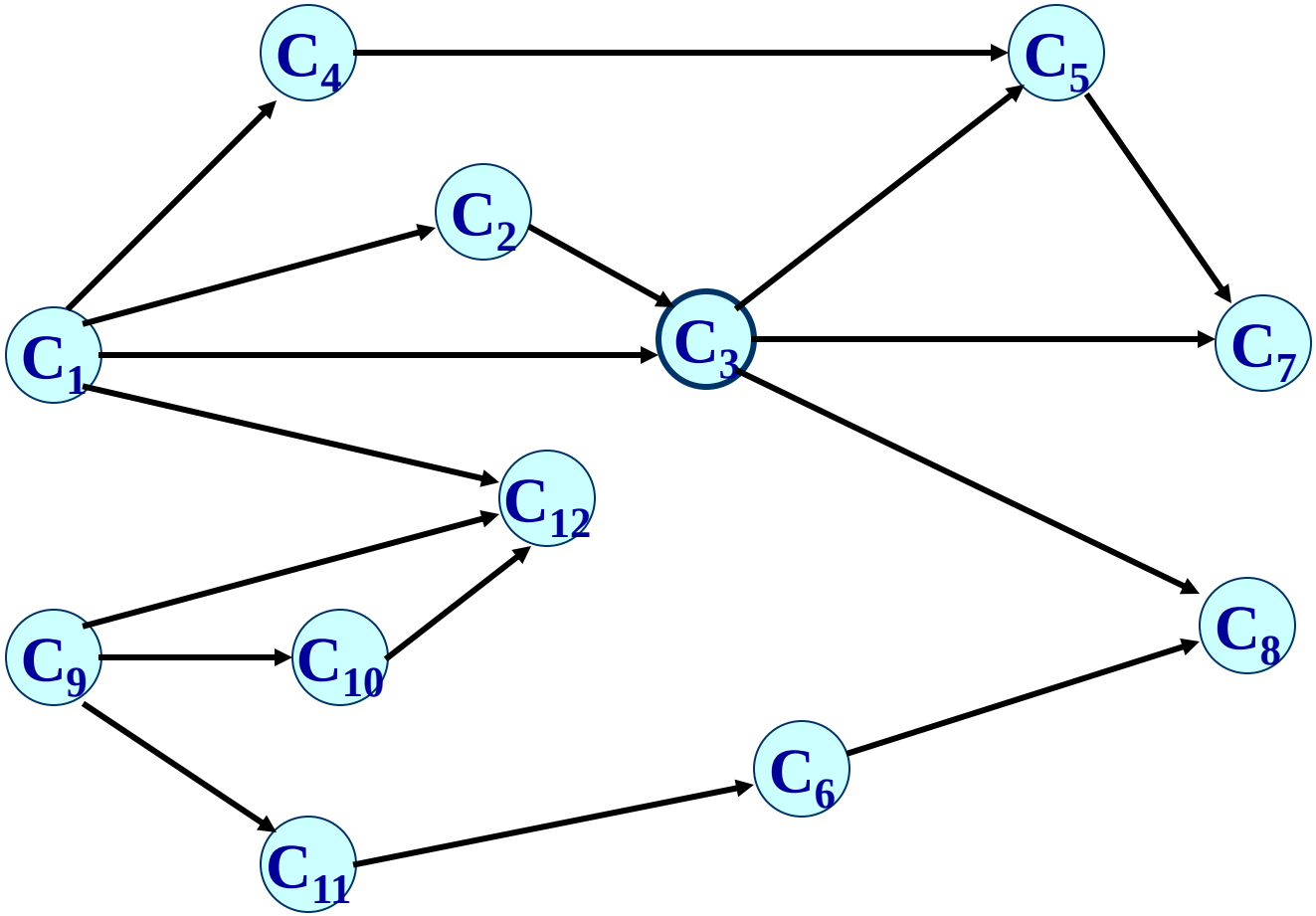
若在有向图G中从顶点V_i到顶点V_j有一条路径，则在序列中顶点V_i必在顶点V_j之前，则顶点的线性序列(V_{i1}, V_{i2} ,..., V_{in})称作一个拓扑序列。

求有向图中顶点的拓扑序列的过程，称作拓扑排序。

例如：软件专业学生必须学习一系列基本课程

课程编号	课程名称	先决条件
C_1	程序设计基础	无
C_2	离散数学	C_1
C_3	数据结构	C_1，C_2
C_4	汇编语言	C_1
C_5	语言的设计与分析	C_3，C_4
C_6	计算机原理	C_{11}
C_7	编译原理	C_5，C_3
C_8	操作系统	C_3，C_6
C_9	高等数学	无
C_{10}	线性代数	C_9
C_{11}	普通物理	C_9
C_{12}	数值分析	C_9，C_{10}，C_1

课程之间的优先关系可以用有向图表示:



一个有向图可用来表示工程的施工图，产品的生产流程图，学生的课程安排图等。有向图的顶点对应一项子工程或一门课程等；有向边<Vi,Vj>表示子工程或课程Vi必须在Vj之前完成。对有向图的顶点进行拓扑排序，对应于这些实际问题就是对各项子工程或各门课程排出一个线性的顺序关系。如果条件限制这些工作必须串行的话，那就应该按拓扑排序安排执行的先后顺序。

设有一项工程，有六项子工程

第一项：无条件；

第二项：在第一、三项完成之后；

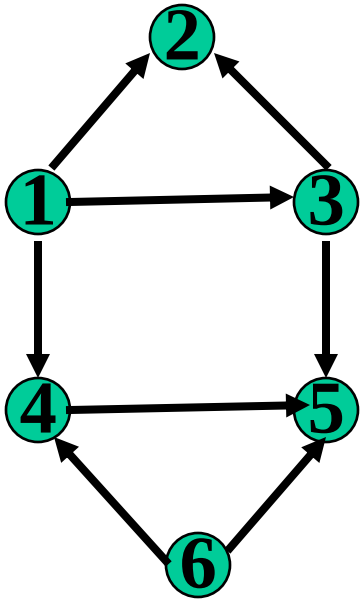
第三项：在第一项之后；

第四项：在第一、六项之后；

第五项：在第三、四、六项之后；

第六项：无条件；

该工程的先后关系可用有向图表示。



拓扑序列：

6，1，4，3，2，5

1，3，2，6，4，5

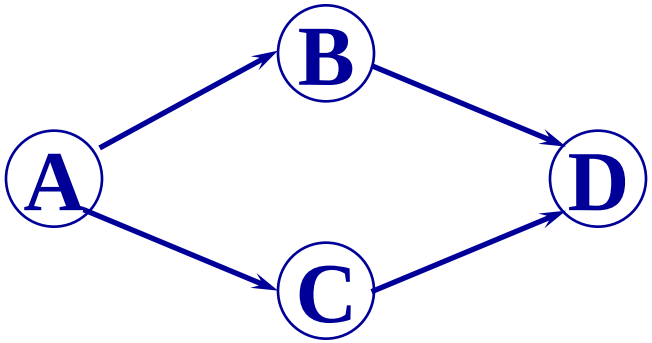
用顶点表示活动，用弧表示活动间优先关系的有向图称为AOV-网。

在AOV-网中，不应该出现有向环，因为存在环意味着某项活动应以自己为先决条件。显然，这是荒谬的。

因此，对给定的AOV-网，应首先判定网中是否存在环。检测的办法是

对有向图构造其顶点的拓扑有序序列，若网中所有顶点都在它的拓扑有序序列中，则该AOV-网中必定不存在环。

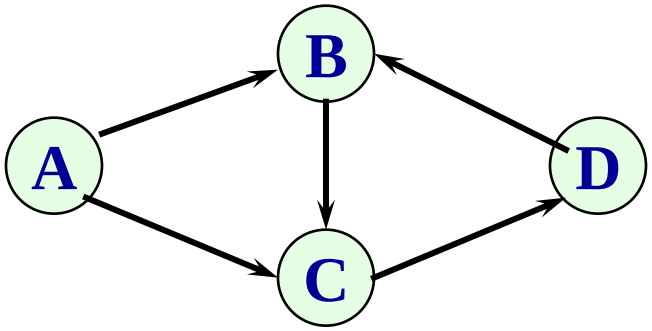
例如：对于下列有向图



可求得拓扑有序序列：

A B C D　或　A C B D

反之，对于下列有向图



不能求得它的拓扑有序序列。

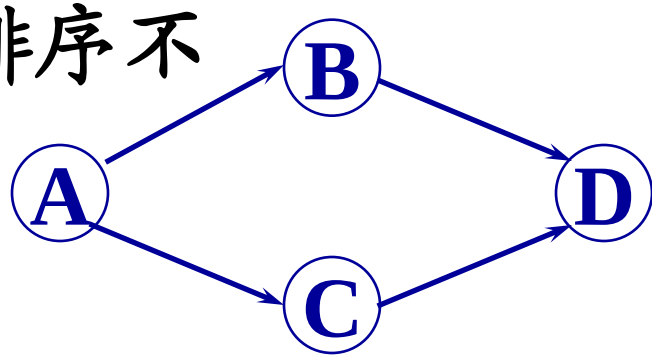
因为图中存在一个回路 {B，C，D}

如何进行拓扑排序？

(1) 从图中选择一个没有前驱的顶点，（即入度为零的点）且输出之;

(2) 从图中删除此顶点及其所有由它发出的边;

(3) 重复(1)、 (2)两步，直到所有的顶点均已输出，整个拓扑排序完成；或者直到剩下的图中再也没有入度为零的顶点，说明此图是有环图，拓扑排序不能再进行下去了。



a b h c d g f e

在算法中需要用定量的描述替代定性的概念

没有前驱的顶点 ≡≡ 入度为零的顶点

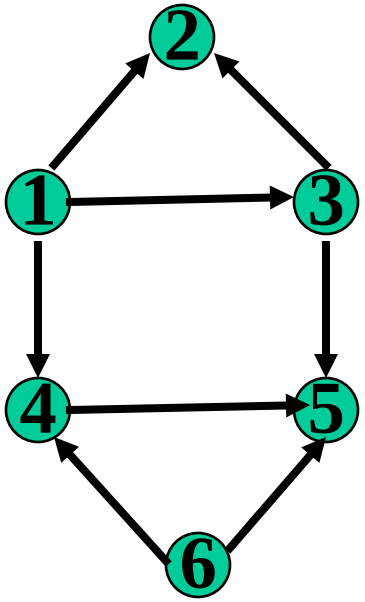
删除顶点及其发出的边 ≡≡ 弧头顶点的入度减1

拓扑排序的实现

(1) 在邻接矩阵上的实现

- 选入度为零的点——找全零的列;
- 删除某顶点的所有出边——对应行置为全零;

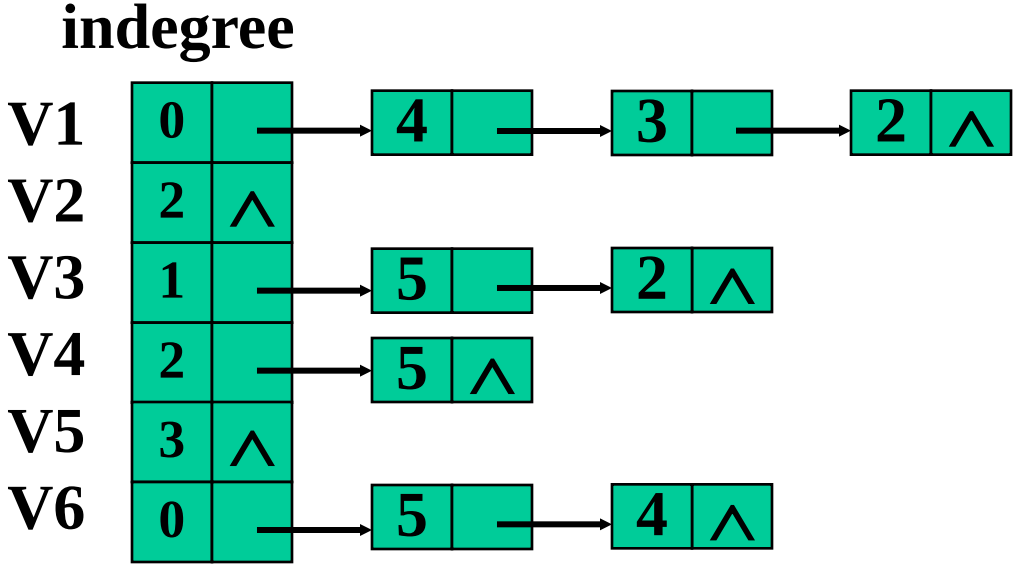
$$\begin{pmatrix} 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 \end{pmatrix}$$

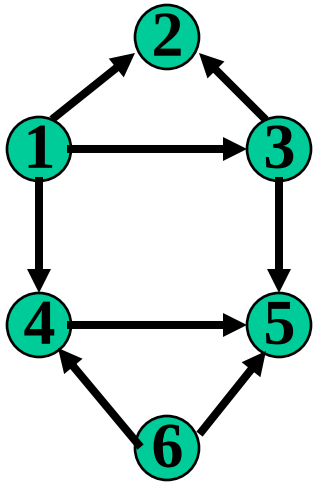


(2)在邻接表上的实现

在表头结点中增加一个存放顶点入度的域。

在拓扑排序过程中，当某顶点的入度为零时，输出此顶点，并将由此顶点发出的有向边所指的顶点的入度减1。为了避免重复检测入度为零的顶点，需另设一个栈，用以存放入度为零的顶点。





在邻接表上实现的拓扑排序算法：

(1) 输入有向边序列，建立邻接表；

(2) 查找邻接表中入度为零的顶点，把入度为零的顶点压入栈；

(3) 当栈不空时，

a. 使用退栈操作，取得栈顶元素v，并输出v；

b. 在邻接表中查找v 的所有邻接点w，将w 的入度减1，若w 的入度变成零，则w 进栈，再转(3)；

(4) 当栈空时，若有向图的所有顶点都已输出，则拓扑排序过程正常结束；否则说明有向图中存在有向回路。

```
TopologicalSort(ALGraph G) {
    //有向图采用邻接表存储结构。若G无回路，则输出G的顶点的一个拓扑序列并返回OK，否则ERROR
    FindInDegree(G,indegree);
                        // 对各顶点求入度,放到indegree数组
    InitStack(S);
    for ( i=0; i<G.vexnum; ++i)
        if (!indegree[i])  Push(S, i);
                                    // 入度为零的顶点入栈
```

```
count=0;                          // 对输出顶点计数
while (!EmptyStack(S)) {
    Pop(S, v);  ++count;  printf(v);
    for (w=FirstAdj(G,v); w!=0; w=NextAdj(G,v,w))
      { --indegree[w];          // 弧头顶点的入度减1
        if (!indegree[w])  Push(S, w);
                        // 新产生的入度为零的顶点入栈
      }
   }
if (count<G.vexnum) printf("图中有回路ERROR")
else return OK;
}// TopologicalSort
```

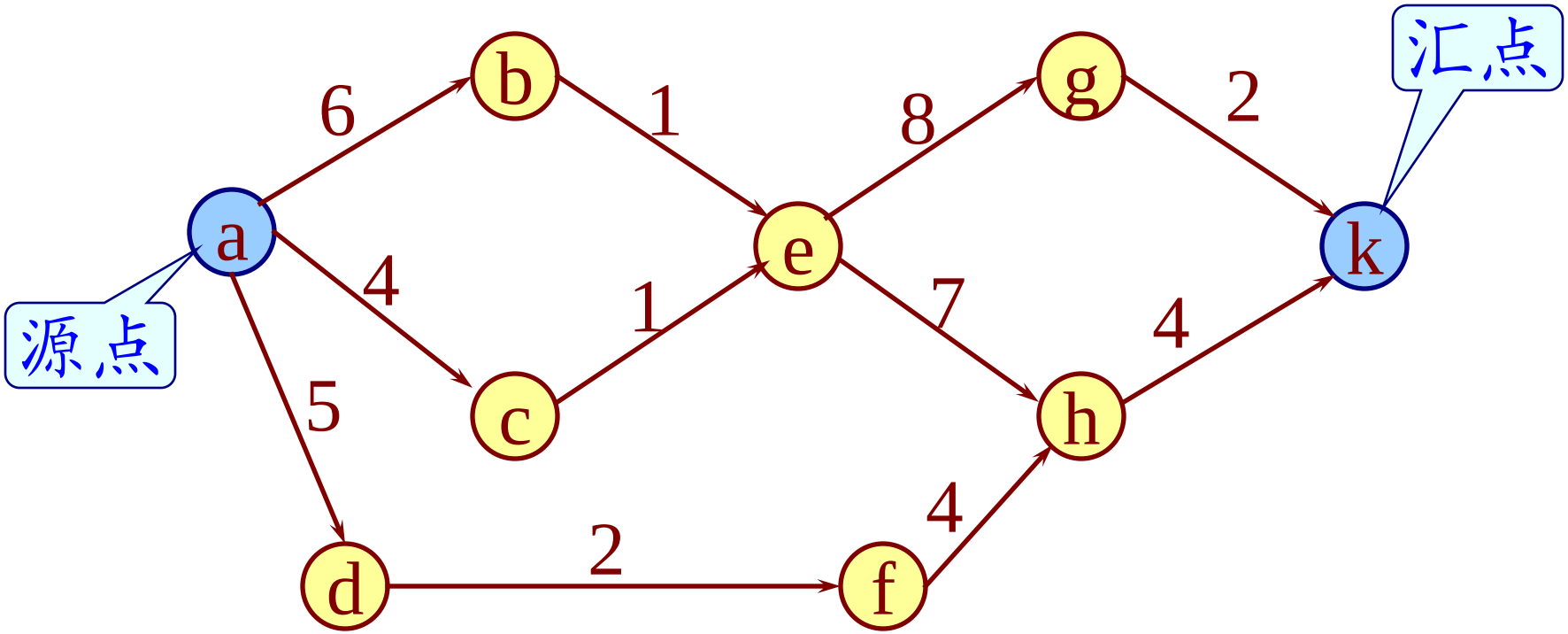
7.7 关键路径

AOE-网 (Active On Edge)：

用顶点表示事件，弧表示活动，弧上权值表示活动持续时间的带权有向无环图称为AOE-网。

AOE-网可用来估算工程的完成时间。

例如:



有11项活动（弧），弧上权值表示完成此项活动所需时间。 9 个事件（顶点）。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。

由于整个工程只有一个开始点和一个完成点，故在正常情况（无环）下，网中只有一个入度为零的点（称作源点）和一个出度为零的点（称作汇点）。

问题：

① 完成整项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？

即：哪些子工程项将影响整个工程的完成期限。

完成工程的最短时间:

从有向图的**源点**到**汇点**的**最长路径**的**长度**。（路径上各活动持续时间之和，不是路径上弧的数目。）

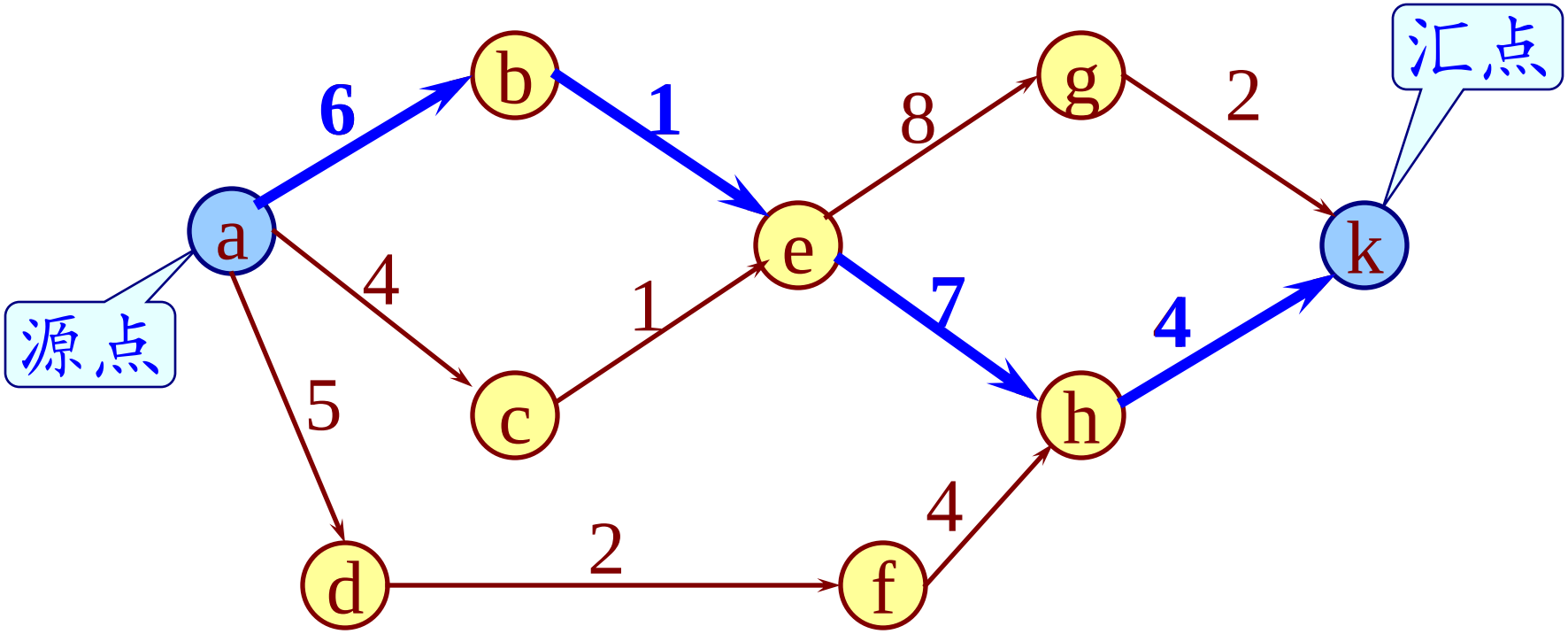
关键路径:

路径长度最长的路径，叫做关键路径。

关键活动:

关键路径上的活动为关键活动。（该弧上的**权值增加** 将使有向图上的**最长路径的长度增加。**）

例如：



如何求关键活动？

- “事件(顶点)” 的 最早发生时间 *ve*(j)

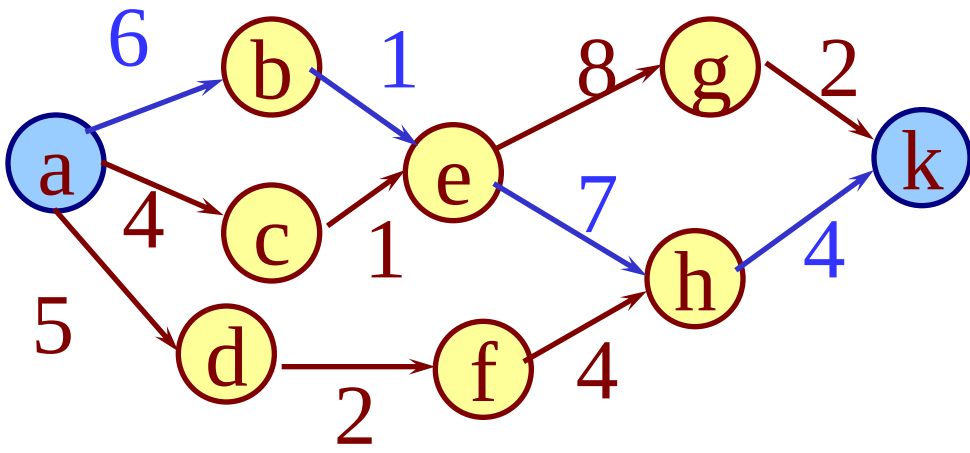
  ve(j) = 从源点到顶点j 的最长路径长度;

  这个时间决定了所有以 j 为尾的弧所表示的活动的最早开始时间。

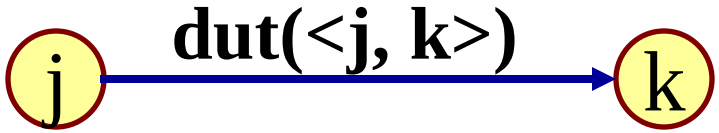
- “事件(顶点)” 的 最迟发生时间 *vl*(k)

  vl(k) = 从顶点k到汇点的最短路径长度。

即在不推迟整个工程完成的前提下，所有以k为尾的弧所表示的活动的最迟开始时间。



假设第 i 条弧为 <j, k>



事件发生时间的计算公式:

“事件(顶点)” 的 最早发生时间 *ve*(k):

ve(源点) = 0;

ve(k) = Max{ve(j) + dut(<j, k>)}

“事件(顶点)” 的 最迟发生时间 *vl*(j):

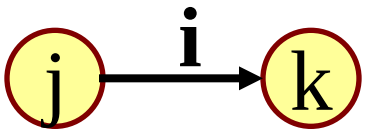
vl(汇点) = ve(汇点);

vl(j) = Min{vl(k) – dut(<j, k>)}

假设第 i 条弧为 <j, k>，则 对第 i 项活动而言，

“活动(弧)”的 最早开始时间 $e(i)$

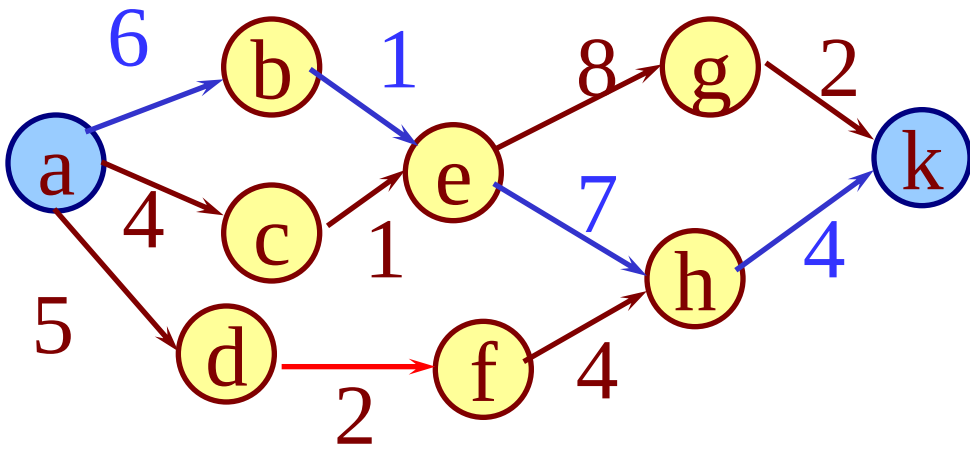
e(i) = ve(j);



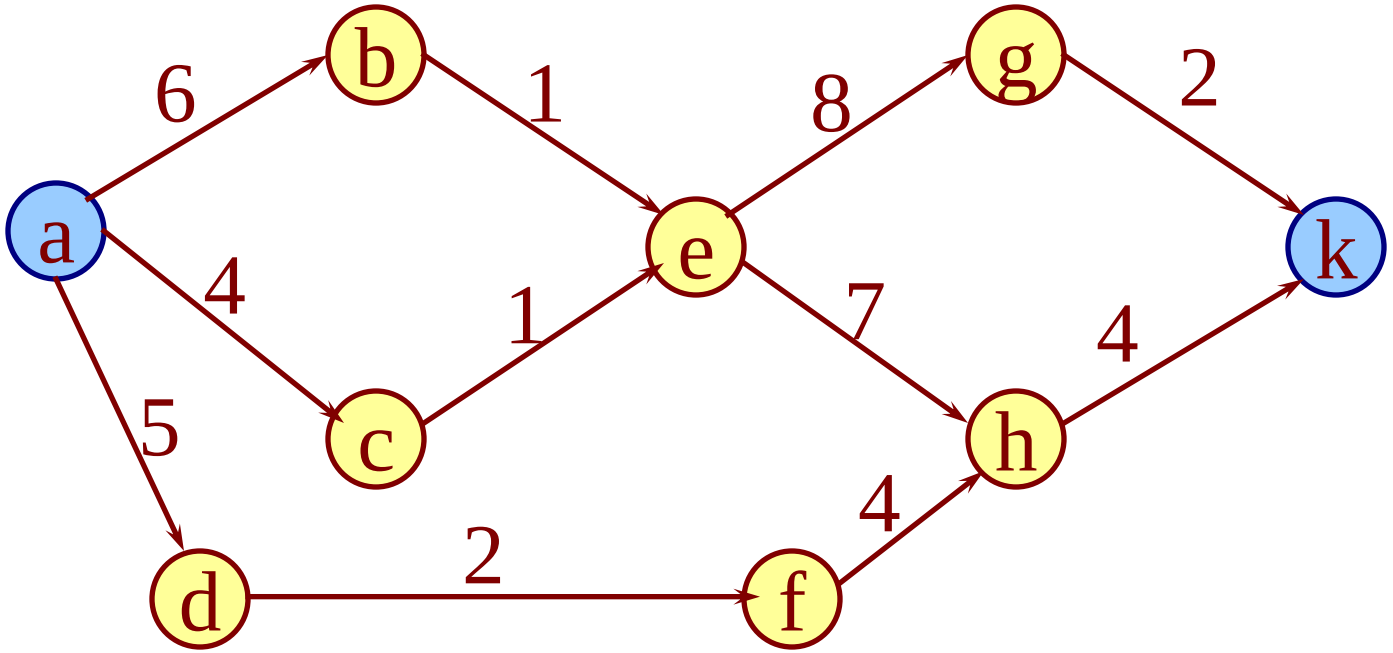
“活动(弧)”的 最迟开始时间 $l(i)$

即: 在不推迟整个工程完成的前提下,活动i最迟必须开始进行的时间

l(i) = vl(k) – dut(<j,k>);



两者之差 $l(i)-e(i)$ 意味着完成活动i 的时间余量，我们把 $l(i)=e(i)$ 的活动叫做关键活动

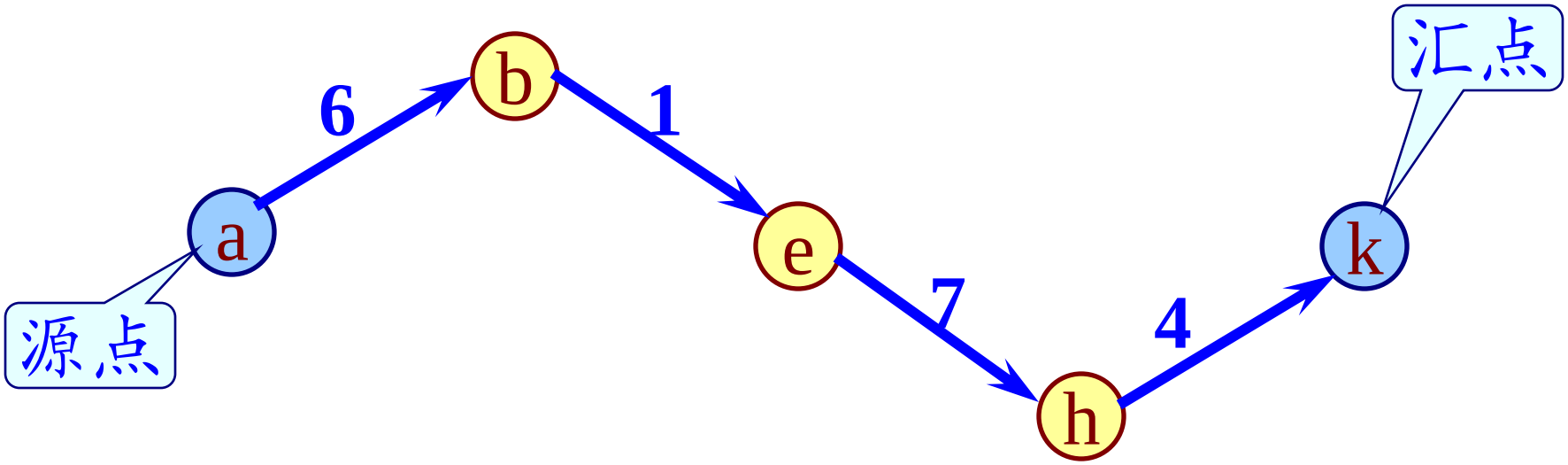


	a	b	c	d	e	f	g	h	k
ve	0	6	4	5	7	7	15	14	18
vl	0	6	6	8	7	10	16	14	18

拓扑有序序列: **a - d - f - c - b - e - h - g - k**

	a	b	c	d	e	f	g	h	k
ve	0	6	4	5	7	7	15	14	18
vl	0	6	6	8	7	10	16	14	18

弧	ab	ac	ad	be	ce	df	eg	eh	fh	gk	hk
权	6	4	5	1	1	2	8	7	4	2	4
e	0	0	0	6	4	5	7	7	7	15	14
l	0	2	3	6	6	8	8	7	10	16	14
	✓			✓				✓			✓



关键活动为<a,b> <b,e> <e,h> <h,k>

关键路径为(a,b,e,h,k)

用AOE网来估算工程完成的时间是非常有用的

提高非关键活动的速度一般不会加快工程进度。提高关键活动的速度有助于加快工程进度。

但关键活动的速度提高是有限度的，只有在不改变网的关键路径的前提下，提高关键活动的速度才有效。

若网中有几条关键路径，那么，单提高一条关键路径上的关键活动的速度，不能导致整个工程缩短工期，而必须提高同时在几条关键路径上的活动的速度。

求关键路径的算法:

(1) 输入 e 条弧<j,k>, 建立AOE-网的存储结构;

(2) 从源点v_0出发,令ve[0]=0,按拓扑有序求其余各顶点的最早发生时间ve[i](1≤i ≤n-1).如果得到的拓扑有序序列中顶点个数小于网中顶点数n,则说明网中存在环,不能求关键路径,算法终止;否则执行步骤(3);

(3) 从汇点v_n出发,令vl[n-1]=ve[n-1],按逆拓扑有序求其余各顶点的最迟发生时间vl(i)(n-2≥i ≥2);

(4) 根据各顶点的 ve 和 vl 值,求每条弧 s 的最早开始时间 e(s) 和最迟开始时间 l(s) . 若某条弧满足条件 e(s)= l(s), 则为关键活动.

算法的实现要点：

显然　求**ve**的顺序应该是按**拓扑有序**的次序；

而　　求**vl**的顺序应该是按**拓扑逆序**的次序；

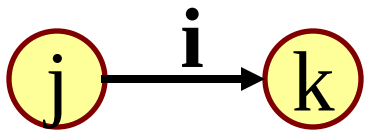
因为　拓扑逆序序列即为拓扑有序序列的**逆序列**，

因此　应该在拓扑排序的过程中，另设一个“**栈**”记下拓扑有序序列。

先将前面的拓扑排序算法改造。在算法中设置一个“栈”S，以保存“入度为零”的顶点。设置另一个“栈”T，保存G的拓扑序列。

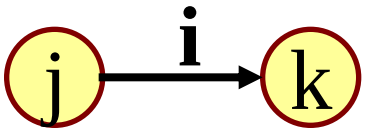
```
StatusTopologicalOrder(ALGraph  G, Stack  &T) {

  //有向图采用邻接表存储结构。若G无回路，则输出G的顶点的一个拓扑序列并返回OK，否则ERROR
    FindInDegree(G,indegree);
                                  // 对各顶点求入度,放到indegree数组
    InitStack(S);          InitStack(T);
    for ( i=0; i<G.vexnum; ++i)
       if (!indegree[i])  Push(S, i);
                                  // 入度为零的顶点入栈
```

```
count=0;      ve[0..G.vexnum-1]=0;    // 对输出顶点计数
while (!EmptyStack(S)) {
    Pop(S, j);     Push(T, j);    ++count;  //j顶点入T栈
    for (p=G.vertices[j].firstarc; p; p=p->nextarc)
       {k=p->adjvex;  --indegree[k]; // 弧头顶点的入度减1
         if (!indegree[k])  Push(S, k);
                          // 新产生的入度为零的顶点入栈
        if ((ve[j]+ *(p->info))>ve[k])     ve[k]=ve[j]+ *(p->info);
       }
   }
if (count<G.vexnum) printf("图中有回路ERROR")

}// TopologicalOrder
```

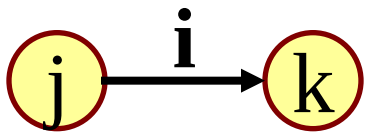


求关键路径的算法

```
Status CriticalPath(ALGraph  G)
                        //G为有向网，输出G的关键活动
{   if (!TopologicalOrder(G,T))  return  ERROR;
    vl [0..G.vexnum-1]=ve[G.vexnum-1];
                                  //初始化顶点事件的最迟发生时间
    while (!StackEmpty(T)) //按拓扑逆序求各顶点vl的值
    for ( pop(T, j),  p=G.vertices[j].firstarc;  p; p=p->nextarc)
      { k=p->adjvex;     dut=*(p->info);    //dut<j,k>
        if (vl[k]-dut<vl[j])      vl[j]=vl[k]-dut;
      }
      ……求e,l和关键活动
} //  CriticalPath
```



```
for (j=0;  j<G.vexnum;  ++j)          //求ee,el和关键活动
     for(p=G.vertices[j].firstarc;  p;  p=p->nextarc)  {
        k=p->adjvex;          dut=*(p->info);
        ee=ve[j];                  el=vl[k]-dut;
        tag=(ee= =el) ? '*': ' ';
        printf(j,k,dut,ee,el,tag);     //输出关键活动
  }
```



图的邻接表存储表示：

```
typedef struct ArcNode {          // 弧结点
    int  adjvex;                  // 该弧所指向的顶点的位置
    struct ArcNode  *nextarc;     // 指向下一条弧的指针
    InfoType  *info;              // 该弧相关信息的指针
} ArcNode;

typedef struct VNode {            // 头结点
    VertexType  data;             // 顶点信息
    ArcNode  *firstarc;           // 指向第一条依附该顶点的弧
  } AdjList[MAX_VERTEX_NUM];

typedef struct {                  // 图结构
    AdjList  vertices;            // 表头结点向量
    int      vexnum, arcnum;      // 图的当前顶点数和弧数
} ALGraph;
```

本章学习要点

1. 熟悉图的各种存储结构及其构造算法，了解实际问题的求解效率与采用何种存储结构和算法有密切联系。

2. 熟练掌握图的两种搜索路径的遍历：遍历的逻辑定义、深度优先搜索和广度优先搜索的算法。在学习中应注意图的遍历算法与树的遍历算法之间的类似和差异。

3. 应用图的遍历算法求解各种简单路径问题。

作业：7.22 7.23 7.27

7.22 试基于图的深度优先搜索策略写一算法，判别以邻接表方式存储的有向图中是否存在由顶点v_i到顶点v_j的路径(i<>j)。注意：算法中涉及的图的基本操作必须在此存储结构上实现。

7.23 同7. 22 题要求，试基于广度优先搜索策略写一算法。

7.27 采用邻接表存储结构，编写一个判别无向图中任意给定的两个顶点之间是否存在一条长度为k的简单路径的算法。