

第七章 图

- 图的定义和术语
- 图的存储结构
- 图的遍历与连通性
- 最小生成树
- 活动网络
- 最短路径

7.1 图的定义和术语

- 图形结构的形式定义 图是由顶点集合(vertex)及顶点间的关系集合组成的一种数据结构:

$$\text{Graph} = (V, R)$$

其中:

$V = \{x \mid x \in \text{某个数据对象}\}$, 是顶点的有穷非空集合;

R ——边的有限集合

$R = \{(x, y) \mid x, y \in V\}$ 无向图 或

$R = \{\langle x, y \rangle \mid x, y \in V \ \&\& \text{Path}(x, y)\}$ 有向图

是顶点之间关系的有穷集合, 也叫做边(edge)集合。

$\text{Path}(x, y)$ 表示从 x 到 y 的一条单向通路, 它是有方向的。 x 弧尾, y 弧头

■ 有向图与无向图

❖ 有向图中：边用 $\langle x, y \rangle$ 表示，且 x 与 y 是有序的。

a. 有向图中的边称为“弧”

b. x ——弧尾或初始点 y ——弧头或终端点

❖ 无向图：边用 (x, y) 表示，且顶 x 与 y 是无序的。

■ 完全图

❖ 在具有 n 个顶点的有向图中，最大弧数为 $n(n-1)$

❖ 在具有 n 个顶点的无向图中，最大边数为 $n(n-1)/2$

■ 顶点的度

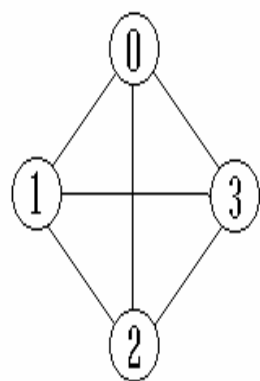
❖ 无向图：与该顶点相关的边的数目

❖ 有向图：

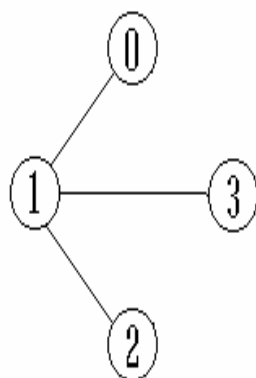
入度 $ID(v)$ ：以该顶点为头的弧的数目

出度 $OD(v)$ ：以该顶点为尾头的弧的数目

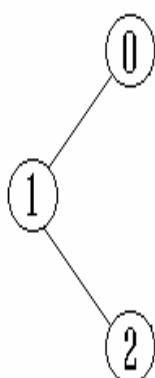
- **权** 某些图的边具有与它相关的数，称之为权。
这种带权图叫做网络。
- **子图** 设有两个图 $G = (V, E)$ 和 $G' = (V', E')$ 。
若 $V' \subseteq V$ 且 $E' \subseteq E$ ，则称图 G' 是图 G 的子图。



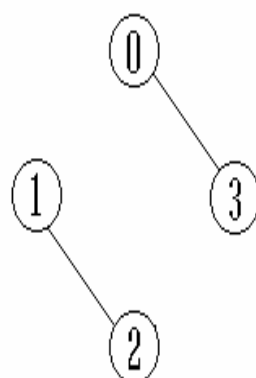
(a) G_1



子图



子图



子图



(b) G_3



子图



子图



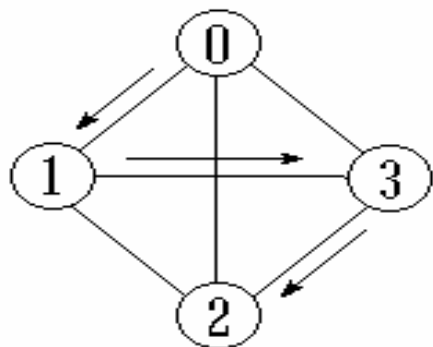
子图

- **路径** 在图 $G = (V, E)$ 中, 若从顶点 v_i 出发, 沿一些边经过一些顶点 $v_{p1}, v_{p2}, \dots, v_{pm}$, 到达顶点 v_j 。则称顶点序列 $(v_i, v_{p1}, v_{p2}, \dots, v_{pm}, v_j)$ 为从顶点 v_i 到顶点 v_j 的路径。它经过的边 $(v_i, v_{p1}), (v_{p1}, v_{p2}), \dots, (v_{pm}, v_j)$ 应是属于 E 的边。

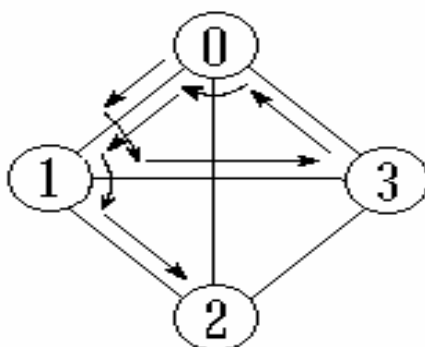
- **路径长度**

- ◆ 非带权图的路径长度是指此路径上边/弧的条数。
- ◆ 带权图的路径长度是指路径上各边/弧的权之和。

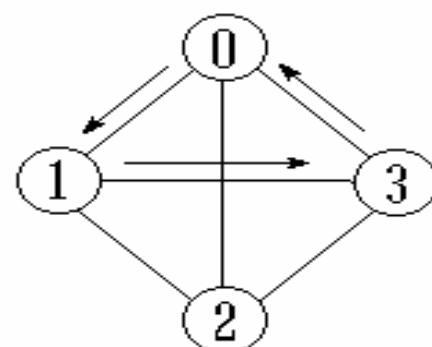
- **简单路径** 若路径上各顶点 v_1, v_2, \dots, v_m 均不互相重复, 则称这样的路径为简单路径。
- **回路** 若路径上第一个顶点 v_1 与最后一个顶点 v_m 重合, 则称这样的路径为回路或环。



(a) 简单路径



(b) 非简单路径

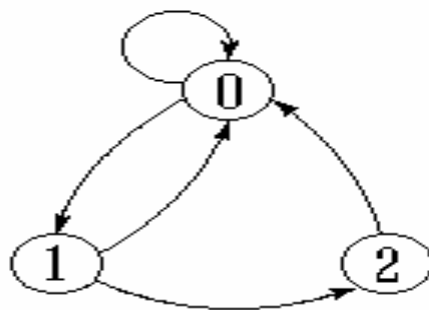


(c) 回路

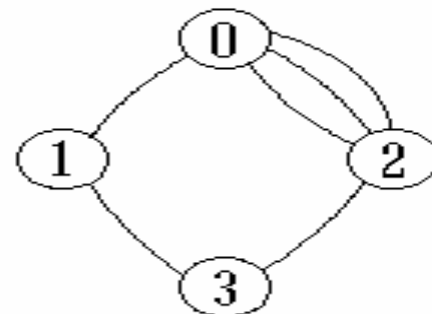
- **连通图与连通分量** 在无向图中, 若从顶点 v_1 到顶点 v_2 有路径, 则称顶点 v_1 与 v_2 是连通的。如果图中任意一对顶点都是连通的, 则称此图是连通图。非连通图的极大连通子图叫做连通分量。

- **强连通图与强连通分量** 在有向图中, 若对于每一对顶点 v_i 和 v_j , 都存在一条从 v_i 到 v_j 和从 v_j 到 v_i 的路径, 则称此图是强连通图。非强连通图的极大强连通子图叫做强连通分量。
- **生成树** 一个连通图的生成树是它的极小连通子图, 在 n 个顶点的情形下, 有 $n-1$ 条边。
 - ❖ 生成树是对指连通图来而言的
 - ❖ 是连同图的极小连同子图
 - ❖ 包含图中的所有顶点
 - ❖ 有且仅有 $n-1$ 条边

- 本章不予讨论的图



(a) 带自身环的图



(b) 多重图

7.2 图的存储表示

1. 邻接矩阵 (Adjacency Matrix)表示法 (数组表示法)

- **顶点表**: 一个记录各个顶点信息的一维数组,
- **邻接矩阵**: 一个表示各个顶点之间的关系 (边或弧) 的二维数组。
- 设图 $G = (V, E)$ 是一个有 n 个顶点的图, 则图的邻接矩阵 $G.arcs[n][n]$ 定义为:
- $G.arcs[i][j] = \begin{cases} 1 & \text{若 } \langle V_i, V_j \rangle \text{ 或 } (V_i, V_j) \in E \\ 0 & \text{反之} \end{cases}$

1) 类型定义

```
#define MAX_VERTEX_NUM 20 //最大顶点数
//typedef enum{DG,DN,UDG,UDN} Grapfkind
typedef int
    AdjMatrix[MAX_VERTEX_NUM][MAX_VERTEX_NUM];

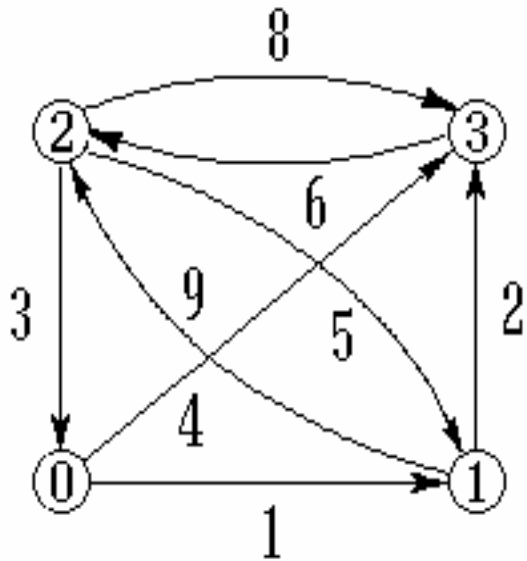
//邻接矩阵类型

typedef struct {
    VertexType vexs[MAX_VERTEX_NUM]; //顶点表
    AdjMatrix arcs; //邻接矩阵
    int vexnum,arcnum; //图的顶点数和弧数
    //Graphkind kind;
} MGraph;
```

- 无向图的邻接矩阵是以主对角线对称的，有向图的邻接矩阵可能是不对称的。
- 在有向图中：
第 i 行 1 的个数就是顶点 i 的出度，
第 j 列 1 的个数就是顶点 j 的入度。
- 在无向图中，第 i 行 (列) 1 的个数就是顶点 i 的度。

网的邻接矩阵

$$G.\text{arcs}[i][j] = \begin{cases} W_{i,j} & \text{若 } \langle V_i, V_j \rangle \text{ 或 } (V_i, V_j) \in E \\ \infty & \text{反之} \end{cases}$$



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

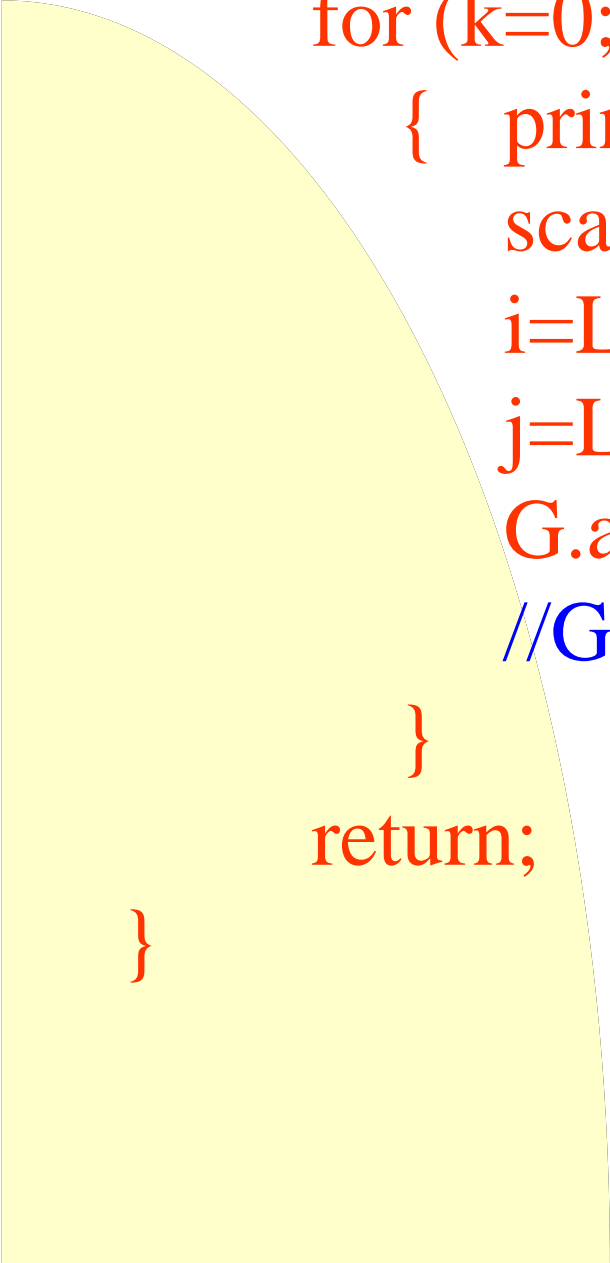
```
int LocateVex(MGraph G,char u)
{   int i;
    for (i=0;i<G.vexnum;i++)
        {   if (u == G.vexs[i]) return i;}
    if (i == G.vexnum) {   printf("Error u!\n");
                          exit(1);
                      }

    return 0;
}
```

2) 建立邻接矩阵

算法7.1,7.2,p162~163

```
void CreateMGraph(MGraph &G)
{   int i,j,k,w; char v1,v2;
    printf("Input vexnum & arcnum:");
    scanf("%d,%d",&G.vexnum,&G.arcnum);
    printf("Input Vertices:");
    scanf("%s",G.vexs);
    // for (i=0;i<G.vexnum;i++)
    //     scanf("%c",&G.vexs[i]);
    for (i=0;i<G.vexnum;i++)
        for (j=0;j<G.vexnum;j++)
            G.arcs[i][j]=0;
```



```
for (k=0;k<G.arcnum;k++)
{   printf("Input Arcs(v1,v2 & w):\n");
    scanf("%c%c,%d",&v1,&v2,&w);
    i=LocateVex(G,v1);
    j=LocateVex(G,v2);
    G.arcs[i][j]=w;
    //G.arcs[j][i]=w;
}
return;
}
```

```
void PrintMGraph(MGraph G)
{   int i,j;
    printf("Output Vertices:");
    printf("%s",G.vexs); printf("\n");
    printf("Output AdjMatrix:\n");
    for (i=0;i<G.vexnum;i++)
        {   for (j=0;j<G.vexnum;j++)
                printf("%4d",G.arcs[i][j]);
            printf("\n");
        }
    return;
}
```


邻接表 (Adjacency List)—— 一种链式存储结构

表结点

adjvex	nextarc	info
--------	---------	------

头结点

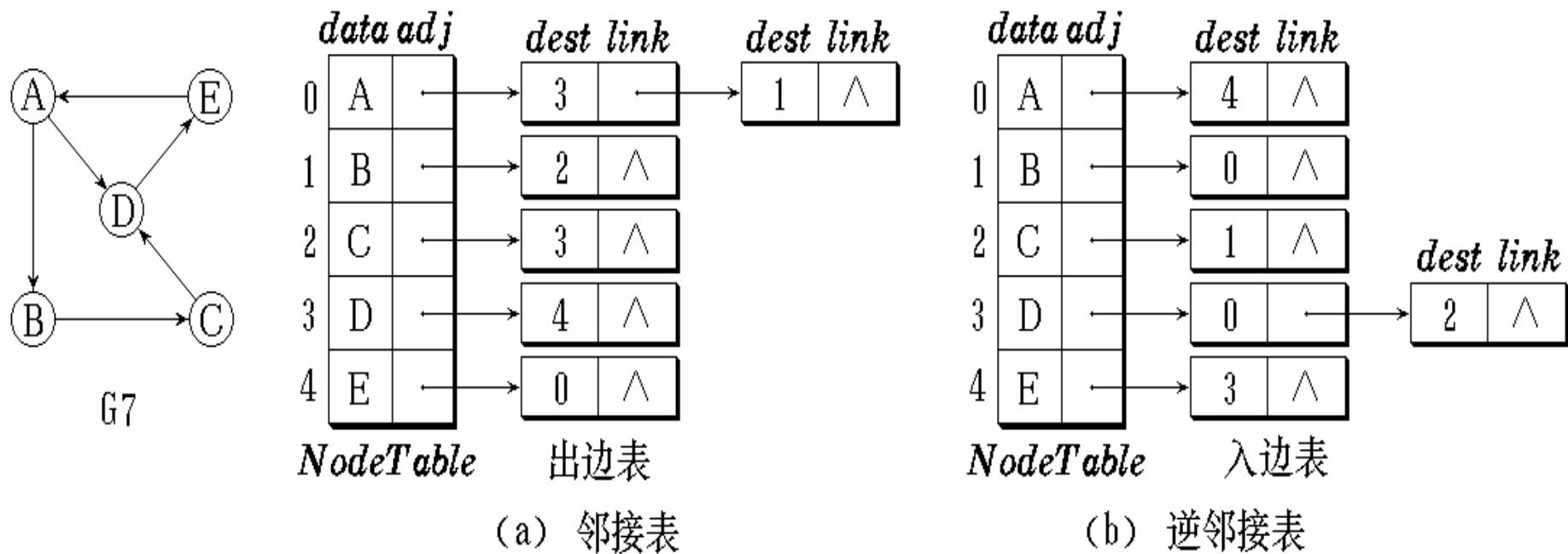
data	firstarc
------	----------

- 把同一个顶点发出的边链接在同一个边链表中，链表的每一个结点代表一条边，叫做表结点（边结点），邻接点域`adjvex`保存与该边相关联的另一顶点的顶点下标，链域`nextarc`存放指向同一链表中下一个表结点的指针，数据域`info`存放边的权。边链表的表头指针存放在头结点中。头结点以顺序结构存储，其数据域`data`存放顶点信息，链域`firstarc`指向链表中第一个顶点。

存储表示

```
typedef struct ArcNode{  
    int adjvex;  
    struct ArcNode *nextarc;  
    int info;  
}ArcNode; //边结点类型  
typedef struct VNode{  
    VertexType data;  
    ArcNode *firstarc;  
}VNode,AdjList[MAX_VERTEX_NUM];  
typedef struct{  
    AdjList vertices; //邻接表  
    int vexnum,arcnum;  
}ALGraph;
```

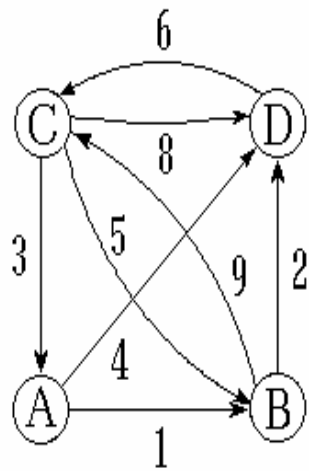
■ 有向图的邻接表和逆邻接表



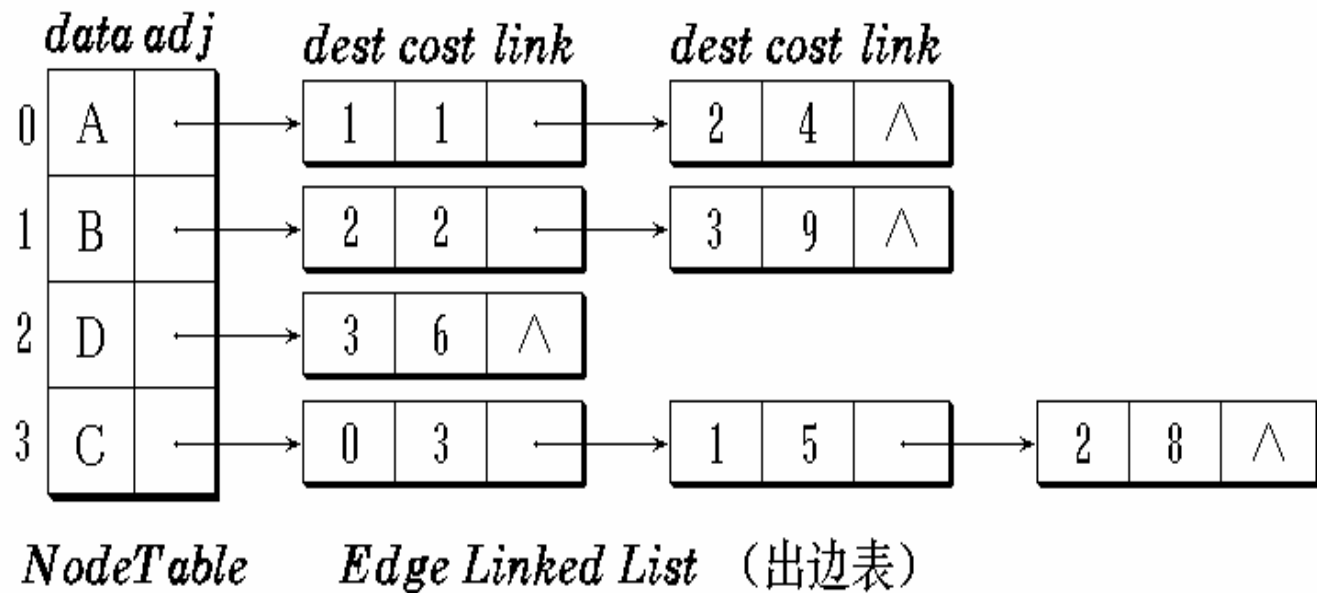
- 在有向图的邻接表中，第 i 个链表中结点的个数是**顶点 V_i 的出度**。
- 在有向图的逆邻接表中，第 i 个链表中结点的个数是**顶点 V_i 的入度**。

- 带权图的边结点中 *info* 保存该边上的权值。
- 顶点 V_i 的边链表的头结点存放在下标为 i 的顶点数组中。
- 在邻接表的边链表中，各个边结点的链入顺序任意，视边结点输入次序而定。
- 设图中有 n 个顶点， e 条边，则用邻接表表示无向图时，需要 n 个顶点结点， $2e$ 个边结点；用邻接表表示有向图时，若不考虑逆邻接表，只需 n 个顶点结点， e 个边结点。
- 建立邻接表的时间复杂度为 $O(n*e)$ 。若顶点信息即为顶点的下标，则时间复杂度为 $O(n+e)$ 。

网络 (带权图) 的邻接表



G9



```
int LocateVex(ALGraph G,char u)
{
    int i;
    for (i=0;i<G.vexnum;i++)
        { if(u==G.vertices[i].data) return i; }
    if (i==G.vexnum) {printf("Error u!\n");exit(1);}
    return 0;
}
```

```
void CreateALGraph_adjlist(ALGraph &G)
{ int i,j,k,w; char v1,v2;
  ArcNode *p;
  printf("Input vexnum & arcnum:");
  scanf("%d,%d",&G.vexnum,&G.arcnum);
  printf("Input Vertices:");
  for (i=0;i<G.vexnum;i++)
  {   scanf("%c",&G.vertices[i].data);
      G.vertices[i].firstarc=NULL;
  }
```



```
printf('Input Arcs(v1,v2 & w):\n');
for (k=0;k<G.arcnum;k++)
{
    scanf('%c,%c,%d",&v1,&v2,&w);
    i=LocateVex(G,v1);
    j=LocateVex(G,v2);
    p=(ArcNode*)malloc(sizeof(ArcNode));
    p->adjvex=j;
    p->info = w;
    p->nextarc=G.vertices[i].firstarc;
    G.vertices[i].firstarc=p;
}
return;
}
```

十字链表 (Orthogonal List)

——有向图的另一种链式存储结构

- 可看作是将有向图的邻接表和逆邻接表结合的一种链表

弧结点

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

顶点结点

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

data顶点信息，firstin以该顶点为头的第一个弧结点，firstout以该结点为尾的第一个弧结点

tailvex弧尾，headvex弧头，hlink指向弧头相同的下一条弧，tlink指向弧尾相同的下一条弧

```
typedef struct ArcBox{
    int headvex, tailvex;
    struct ArcBox *hlink, *tlink;
    int info;
}ArcBox;
typedef struct VexNode{
    VertexType data;
    ArcBox *firstin, *firstout;
}VexNode;
typedef struct{
    VexNode xlist[MAX_VERTEX_NUM];
    int vexnum, arcnum;
}OLGraph; //图的十字链表表示
```

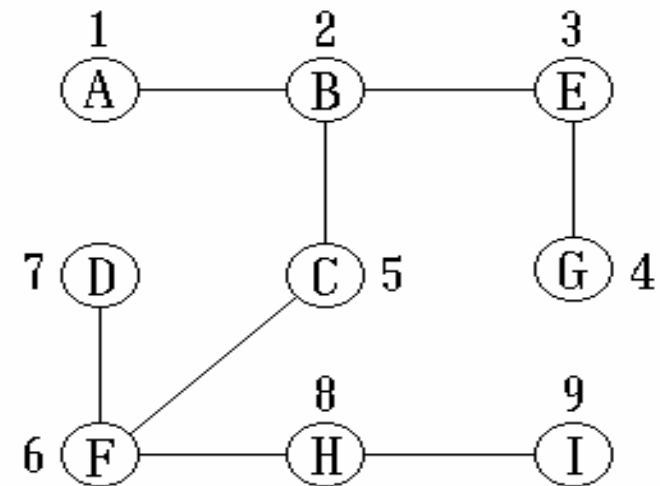
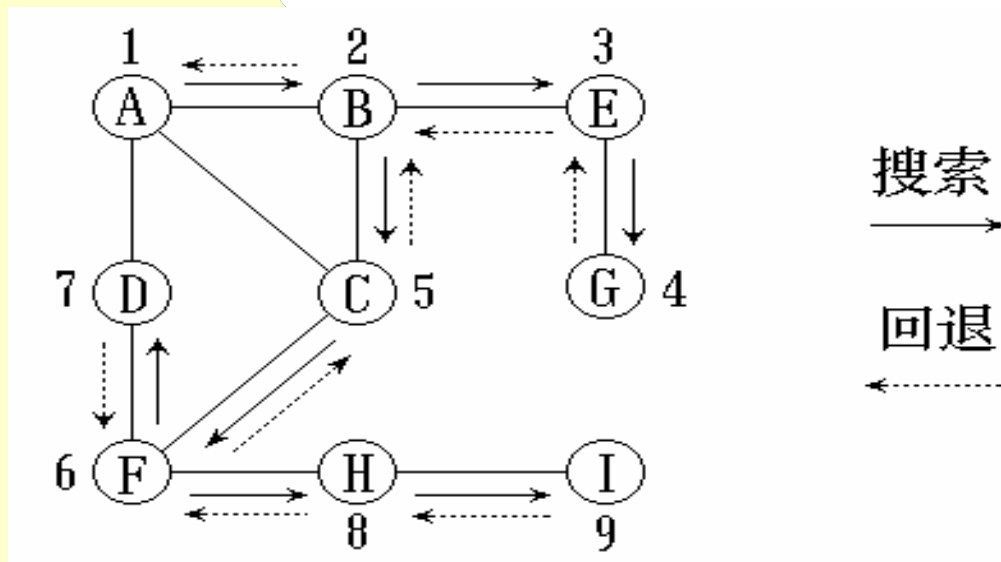
图的遍历

- 从图中某一顶点出发访遍图中其余顶点，且使每个顶点仅被访问一次，就叫做**图的遍历** (Traversing Graph)。
- 图的遍历算法是求解图的**连通性问题**、**拓扑排序**和求 **关键路径**等算法的基础。
- 两条遍历图的路径：**深度优先搜索**、**广度优先搜索**

- 图中可能存在回路，且图的任一顶点都可能与其它顶点相通，在访问完某个顶点之后可能会沿着某些边又回到了曾经访问过的顶点。
- 为了避免重复访问，可设置一个标志顶点是否被访问过的辅助数组 *visited* []，它的初始状态为 0，在图的遍历过程中，一旦某一个顶点 *i* 被访问，就立即让 *visited* [*i*] 为 1，防止它被多次访问。

深度优先搜索 *DFS* (Depth First Search)

■ 深度优先搜索的示例



- **DFS** 在访问图中某一起始顶点 v 后，由 v 出发，访问它的任一邻接顶点 w_1 ；再从 w_1 出发，访问与 w_1 邻接但还没有访问过的顶点 w_2 ；然后再从 w_2 出发，进行类似的访问，... 如此进行下去，直至到达所有的邻接顶点都被访问过的顶点 u 为止。接着，退回一步，退到前一次刚访问过的顶点，看是否还有其它没有被访问的邻接顶点。如果有，则访问此顶点，之后再从此顶点出发，进行与前述类似的访问；如果没有，就再退回一步进行搜索。重复上述过程，直到连通图中所有顶点都被访问过为止。

■ DFS 的思路

- (1) 从图中的某个顶点 V 出发，访问之；
- (2) 依次从顶点 V 的未被访问过的邻接点出发，

深度优先遍历图，直到图中所有和顶点 V 有路径相通的顶点都被访问到；

- (3) 若此时图中尚有顶点未被访问到，则另选一个未被访问过的顶点作起始点，重复上述(1) (2)的操作，直到图中所有的顶点都被访问到为止。

图的深度优先搜索算法

```
int visited[MAX_VERTEX_NUM];  
void DFS(ALGraph G, int v)  
{ ArcNode *p;  
  printf("%c",G.vertices[v].data);  
  visited[v]=1;  
  p=G.vertices[v].firstarc;  
  while (p)  
  { if (!visited[p->adjvex]) DFS(G,p->adjvex);  
    p=p->nextarc;  
  }  
} //从第v个顶点出发DFS
```

整个图的DFS遍历

```
void DFSTraverse(ALGraph G){  
    for (int v=0;v<G.vexnum;++v)  
        visited[v]=0;  
    for (v=0;v<G.vexnum;++v)  
        if (!visited[v]) DFS(G,v);  
}
```

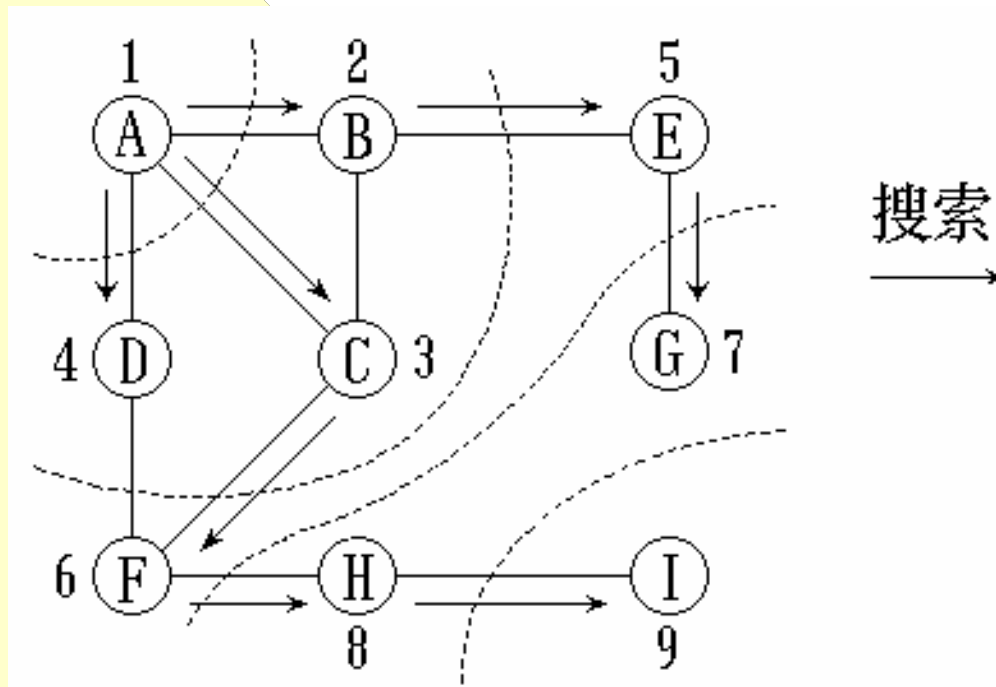
对于连通图，从一个顶点出发，调用DFS函数即可将所有顶点都遍历到。

算法分析

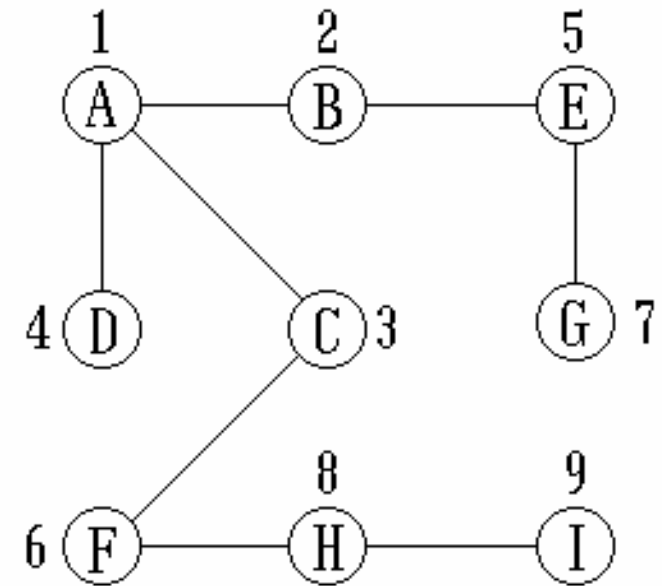
- 图中有 n 个顶点， e 条边。
- 如果用邻接表表示图，沿 $Firstarc \rightarrow nextarc$ 链可以找到某个顶点 v 的所有邻接顶点 w 。由于总共有 $2e$ 个边结点，所以扫描边的时间为 $O(e)$ 。而且对所有顶点递归访问1次，所以遍历图的时间复杂性为 $O(n+e)$ 。
- 如果用邻接矩阵表示图，则查找每一个顶点的所有边，所需时间为 $O(n)$ ，则遍历图中所有的顶点所需的时间为 $O(n^2)$ 。

广度优先搜索 *BFS* (Breadth First Search)

■ 广度优先搜索的示例



广度优先搜索过程



广度优先生成树

- 使用广度优先搜索在访问了起始顶点 v 之后，由 v 出发，依次访问 v 的各个未曾被访问过的邻接顶点 w_1, w_2, \dots, w_t ，然后再顺序访问 w_1, w_2, \dots, w_t 的所有还未被访问过的邻接顶点。再从这些访问过的顶点出发，再访问它们的所有还未被访问过的邻接顶点，... 如此做下去，直到图中所有顶点都被访问到为止。
- 广度优先搜索是一种分层的搜索过程，每向前走一步可能访问一批顶点，不像深度优先搜索那样有往回退的情况。因此，广度优先搜索不是一个递归的过程，其算法也不是递归的。

■ BFS 的思路

- (1) 从图中的某个顶点 V 出发，访问之；
- (2) 依次访问顶点 V 的各个未被访问过的邻接点，将 V 的全部邻接点都访问到；
- (3) 分别从这些邻接点出发，依次访问它们的未被访问过的邻接点，并使“先被访问的顶点的邻接点”先于“后被访问的顶点的邻接点”被访问，直到图中所有已被访问过的顶点的邻接点都被访问到。

- 为了实现逐层访问，算法中使用了一个队列，以记忆正在访问的这一层和上一层的顶点，以便于向下一层访问。
- 与深度优先搜索过程一样，为避免重复访问，需要一个辅助数组 *visited* []，给被访问过的顶点加标记。

图的广度优先搜索算法

```
void BFSTraverse(ALGraph G){  
    for (int v=0;v<G.vexnum;++v)  
        visited[v]=0;  
    for (v=0;v<G.vexnum;++v)  
        if (!visited[v]) BFS(G,v);  
}
```

```
void BFS(ALGraph G,int v)  
{ ArcNode *p; SqQueue Q;  
  InitQueue(Q);  
  printf("%c",G.vertices[v].data);  
  visited[v]=1;  
  Q.base[Q.rear]=v;  
  Q.rear=(Q.rear+1)%MAXQSIZE;  
  while (Q.front!=Q.rear)  
  { v=Q.base[Q.front];  
    Q.front=(Q.front+1)%MAXQSIZE;  
    p=G.vertices[v].firstarc;
```

while (p)

{ if (!visited[p->adjvex])

{ printf("%c",G.vertices[p->adjvex].data);

visited[p->adjvex]=1;

Q.base[Q.rear]=p->adjvex;

Q.rear=(Q.rear+1)%MAXQSIZE;

}

p=p->nextarc;

}

}

}

算法分析

- 如果使用邻接表表示图，则循环的总时间代价为 $d_0 + d_1 + \dots + d_{n-1} = O(e)$ ，其中的 d_i 是顶点 i 的度。
- 如果使用邻接矩阵，则对于每一个被访问过的顶点，循环要检测矩阵中的 n 个元素，总的时间代价为 $O(n^2)$ 。

最小生成树(minimum cost spanning tree)

——连通网的最小代价生成树

1. 生成树

- 假设 $E(G)$ 为连通图 G 中所有边的集合，则从图中任一顶点出发遍历图时，必将 $E(G)$ 分成两个集合 $T(G)$ 和 $B(G)$ ，其中 $T(G)$ 是遍历过程中历经的边的集合； $B(G)$ 是剩余的边的集合。则边集 $T(G)$ 和图 G 中所有顶点一起构成连通图 G 的一棵生成树。
- 按照生成树的定义， n 个顶点的连通网络的生成树有 n 个顶点、 $n-1$ 条边。

- 使用不同的遍历图的方法，可以得到不同的生成树；从不同的顶点出发，也可能得到不同的生成树。如深度优先生成树、广度优先生成树

2. 最小生成树（最小代价生成树） p173

- 构造准则：
 - ◆ 尽可能用网络中权值最小的边；
 - ◆ 必须使用且仅使用 $n-1$ 条边来联结网络中的 n 个顶点；
 - ◆ 不能使用产生回路的边。

3. 算法：采用MST性质 p173

1) 普里姆算法

2) 克鲁斯卡尔算法

普里姆(Prim)算法

- 普里姆算法的基本思想:

从连通网络 $N = \{ V, E \}$ 中的某一顶点 u_0 出发, 选择与它关联的具有最小权值的边 (u_0, v) , 将其顶点加入到生成树的顶点集合 U 中。

以后每一步从一个顶点在 U 中, 而另一个顶点不在 U 中的各条边中选择权值最小的边 (u, v) , 把该边加入到生成树的边集 TE 中, 把它的顶点加入到集合 U 中。如此重复执行, 直到网络中的所有顶点都加入到生成树顶点集合 U 中为止。

- 采用邻接矩阵作为图的存储表示。

- 若选择从顶点0出发, 即 $u_0 = 0$, 则辅助数组的两个域的初始状态为:
- 然后反复做以下工作:
 - ◆ 在 *closedge* [i] 中选择 *adjvex* $\neq 0$ && *lowcost* 最小的边 *i*
用 *k* 标记它。则选中的权值最小的边为 (*closedge*[*k*], *G.vexs*[*k*]), 相应的权值为 *closedge*[*k*].*lowcost*。
 - ◆ 将 *closedge*[*k*].*adjvex* 改为 0, 表示它已加入生成树顶点集合。将边 (*closedge*[*k*], *G.vexs*[*k*]) 加入生成树的边集合。

- ◆ 取 $lowcost[i] = \min\{ lowcost[i], G.arcs[k][i] \}$, 即用生成树顶点集合外各顶点 i 到刚加入该集合的新顶点 k 的距离 $G.arcs[k][i]$ 与原来它们到生成树顶点集合中顶点的最短距离 $lowcost[i]$ 做比较, 取距离近的作为这些集合外顶点到生成树顶点集合内顶点的最短距离。
- ◆ 如果生成树顶点集合外顶点 i 到刚加入该集合的新顶点 k 的距离比原来它到生成树顶点集合中顶点的最短距离还要近, 则修改 adj : $adjvex = v$ 。表示生成树外顶点 i 到生成树内顶点 k 当前距离最近。

```
typedef int VRType;
struct{
    VertexType adjvex;
    VRType lowcost;
}closedge[MAX_VERTEX_NUM];
void MiniSpanTree_PRIM(MGraph G,VertexType u)
{
    int k,j,i,minCost;
    k=LocateVex(G,u);
    for (j=0;j<G.vexnum;++j)
        if (j!=k)
        {
            closedge[j].adjvex=u;
            closedge[j].lowcost=G.arcs[k][j];
        }
```

```
closedge[k].lowcost=0;
for (i=1;i<G.vexnum;++i)
{ k=minimum(closedge);
  minCost=INFINITY;
  for (j=0;j<G.vexnum;++j)
  { if (closedge[j].lowcost <minCost &&
      closedge[j].lowcost!=0)
    { minCost=closedge[j].lowcost;
      k=j;}
  }
}
```

```
printf("(%c,%c)\n",closededge[k].adjvex,G.vexs[k]);
closededge[k].lowcost=0;
for (j=0;j<G.vexnum;++j)
    if (G.arcs[k][j]<closededge[j].lowcost)
        { closededge[j].adjvex=G.vexs[k];
          closededge[j].lowcost=G.arcs[k][j];}
    }
}
```

算法分析:

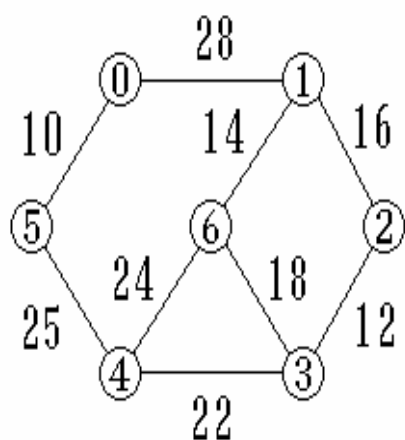
若连通网络有 n 个顶点，则该算法的时间复杂度为 $O(n^2)$ ，它适用于边稠密的网络。

克鲁斯卡尔 (Kruskal) 算法

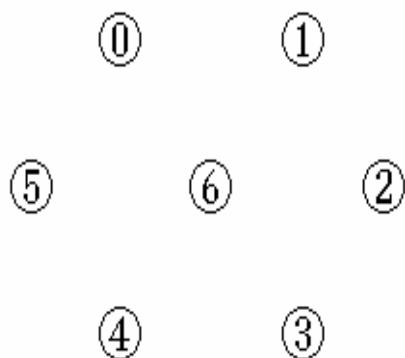
- 克鲁斯卡尔算法的基本思想:

设有一个有 n 个顶点的连通网络 $N = \{V, E\}$, 最初先构造一个只有 n 个顶点, 没有边的非连通图 $T = \{V, \emptyset\}$, 图中每个顶点自成一个连通分量。当在 E 中选到一条具有最小权值的边时, 若该边的两个顶点落在不同的连通分量上, 则将此边加入到 T 中; 否则将此边舍去, 重新选择一条权值最小的边。如此重复下去, 直到所有顶点在同一个连通分量上为止。

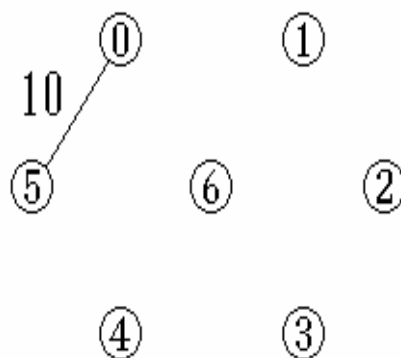
应用克鲁斯卡尔算法构造最小生成树的过程



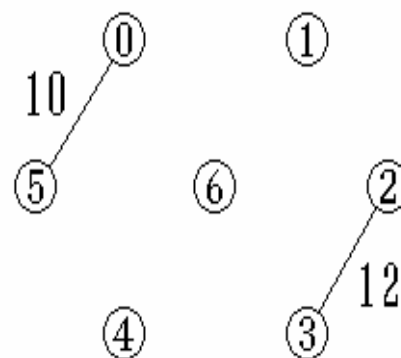
(a)



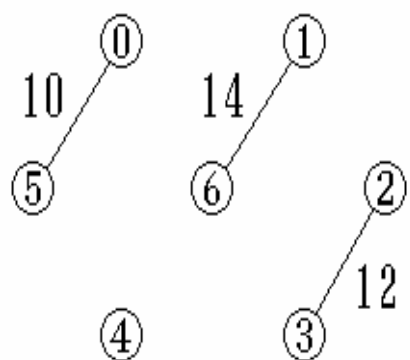
(b)



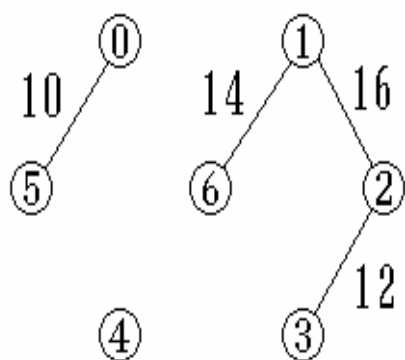
(c)



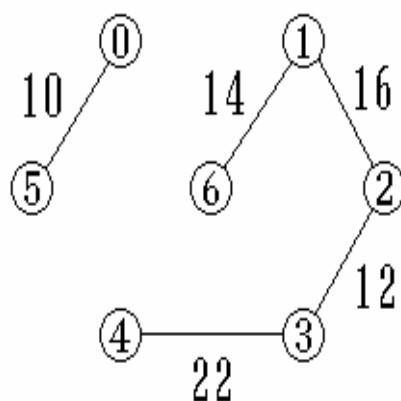
(d)



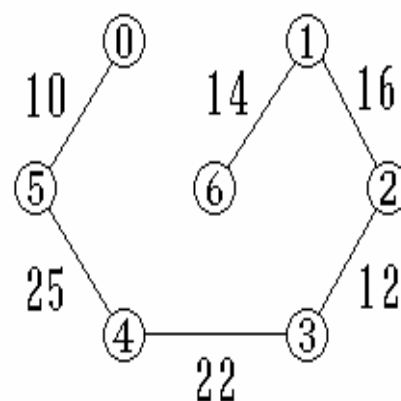
(e)



(f)



(g)



(h)

最短路径 (Shortest Path)

- **最短路径问题**: 如果从图中某一顶点(称为源点)到达另一顶点(称为终点)的路径可能不止一条, 如何找到一条路径使得沿此路径上各边上的权值总和达到最小。
- **问题解法**
 - ◆ 边上权值非负情形的单源最短路径问题
 - **Dijkstra算法**
 - ◆ 所有顶点之间的最短路径
 - **Floyd算法**

边上权值非负情形的单源最短路径问题

- **问题的提法：** 给定一个带权有向图 D 与源点 v ，求从 v 到 D 中其它顶点的最短路径。限定各边上的权值大于或等于0。
- 为求得这些最短路径，Dijkstra提出按路径长度的递增次序，逐步产生最短路径的算法。首先求出长度最短的一条最短路径，再参照它求出长度次短的一条最短路径，依次类推，直到从顶点 v 到其它各顶点的最短路径全部求出为止。
- 举例说明(p188图7.34,7.35)

- 引入一个辅助数组 D 。它的每一个分量 $D[i]$ 表示当前找到的从源点 v_0 到终点 v_i 的最短路径的长度。初始状态：
 - ◆ 若从源点 v_0 到顶点 v_i 有边，则 $D[i]$ 为该边上的权值；
 - ◆ 若从源点 v_0 到顶点 v_i 没有边，则 $D[i]$ 为 $+\infty$ 。
- 一般情况下，假设 S 是已求得的最短路径的终点的集合，则可证明：下一条最短路径必然是从 v_0 出发，中间只经过 S 中的顶点便可到达的那些顶点 v_x ($v_x \in V-S$)的路径中的一条。
- 每次求得一条最短路径之后，其终点 v_k 加入集合 S ，然后对所有的 $v_i \in V-S$ ，修改其 $dist[i]$ 值。

Dijkstra算法可描述如下:

☆ 初始化: $S \leftarrow \{v_0\};$

$D[j] \leftarrow arcs[0][j], j = 1, 2, \dots, n-1;$

// n 为图中顶点个数

🕒 求出最短路径的长度:

$D[k] \leftarrow \min\{D[i]\}, i \in V - S;$

$S \leftarrow S \cup \{k\};$

🕒 修改:

$D[i] \leftarrow \min\{D[i], D[k] + arcs[k][i]\},$

对于每一个 $i \in V - S;$

🕒 判断: 若 $S = V$, 则算法结束, 否则转🕒。

用于计算最短路径的图邻接矩阵的定义

```
#define NumVertices 6           //图中最大顶点个数
typedef int PathMatrix[NumVertices][NumVertices];
//最短路径数组
typedef int ShortPathTable[NumVertices]; //最短路径长度
typedef int AdjMatrix[NumVertices][NumVertices];
typedef char VertexType;
typedef struct{
    VertexType vexs[NumVertices];
    AdjMatrix arcs; //邻接矩阵
    int vexnum, arcnum;
}MGraph;
```

迪杰斯特拉算法(p189 算法7.15)

```
void ShortestPath_DIJ(MGraph G,int v0,PathMatrix
&P,ShortPathTable &D){
    int final[NumVertices]; int w;
    for (int v=0;v<G.vexnum;++v){
        final[v]=FALSE;
        D[v]=G.arcs[v0][v];
        for ( w=0;w<G.vexnum;++w)
            P[v][w]=FALSE;
        if (D[v]<INFINITY){
            P[v][v0]=TRUE; P[v][v]=TRUE;}
    }
}
```

```
for (int i=1;i<G.vexnum;++i){
    int min = INFINITY;
    for (w=0;w<G.vexnum ;++w)
        if (!final[w])
            if (D[w]<min)
                { v=w;min=D[w];}
    for (w=0;w<G.vexnum;++w)
        if
        (!final[w]&&(min+G.arcs[v][w]<D[w])){
            D[w]=min+G.arcs[v][w];
            for (int j=0;j<G.vexnum;++j)
                P[w][j]=P[v][j];
            P[w][w]=TRUE;
        }
```

}

}

算法的时间复杂度: $O(n^2)$

所有顶点之间的最短路径

- 问题的提法: 已知一个各边权值均大于0的带权有向图, 对每一对顶点 $v_i \neq v_j$, 要求求出 v_i 与 v_j 之间的最短路径和最短路径长度。

- Floyd算法的基本思想:

定义一个 n 阶方阵序列:

$$D^{(-1)}, D^{(0)}, \dots, D^{(n-1)}.$$

其中 $D^{(-1)}[i][j] = G.arcs[i][j];$

$$D^{(k)}[i][j] = \min \{ D^{(k-1)}[i][j],$$

$$D^{(k-1)}[i][k] + D^{(k-1)}[k][j] \}, k = 0, 1, \dots, n-1$$

$D^{(0)}[i][j]$ 是从顶点 v_i 到 v_j ,中间顶点是 v_0 的最短路径的长度, $D^{(k)}[i][j]$ 是从顶点 v_i 到 v_j ,中间顶点的序号不大于 k 的最短路径的长度, $D^{(n-1)}[i][j]$ 是从顶点 v_i 到 v_j 的最短路径长度。

Floyd算法允许图中有带负权值的边,但不许有包含带负权值的边组成的回路。

本章给出的求解最短路径的算法不仅适用于带权有向图,对带权无向图也可以适用。因为带权无向图可以看作是有往返二重边的有向图,只要在顶点 v_i 与 v_j 之间存在无向边 (v_i, v_j) ,就可以看成是在这两个顶点之间存在权值相同的两条有向边 $\langle v_i, v_j \rangle$ 和 $\langle v_j, v_i \rangle$ 。



活动网络 (Activity Network)

用顶点表示活动的网络 (AOV网络)

- 计划、施工过程、生产流程、程序流程等都是“**工程**”。除了很小的工程外，一般都把工程分为若干个叫做“**活动**”的子工程。完成了这些活动，这个工程就可以完成了。
- 例如，计算机专业学生的学习就是一个工程，每一门课程的学习就是整个工程的一些活动。其中有些课程要求先修课程，有些则不要求。这样在有的课程之间有领先关系，有的课程可以并行地学习。

课程代号

课程名称

先修课程

C₁

高等数学

C₂

程序设计基础

C₃

离散数学

C₁, C₂

C₄

数据结构

C₃, C₂

C₅

高级语言程序设计

C₂

C₆

编译方法

C₅, C₄

C₇

操作系统

C₄, C₉

C₈

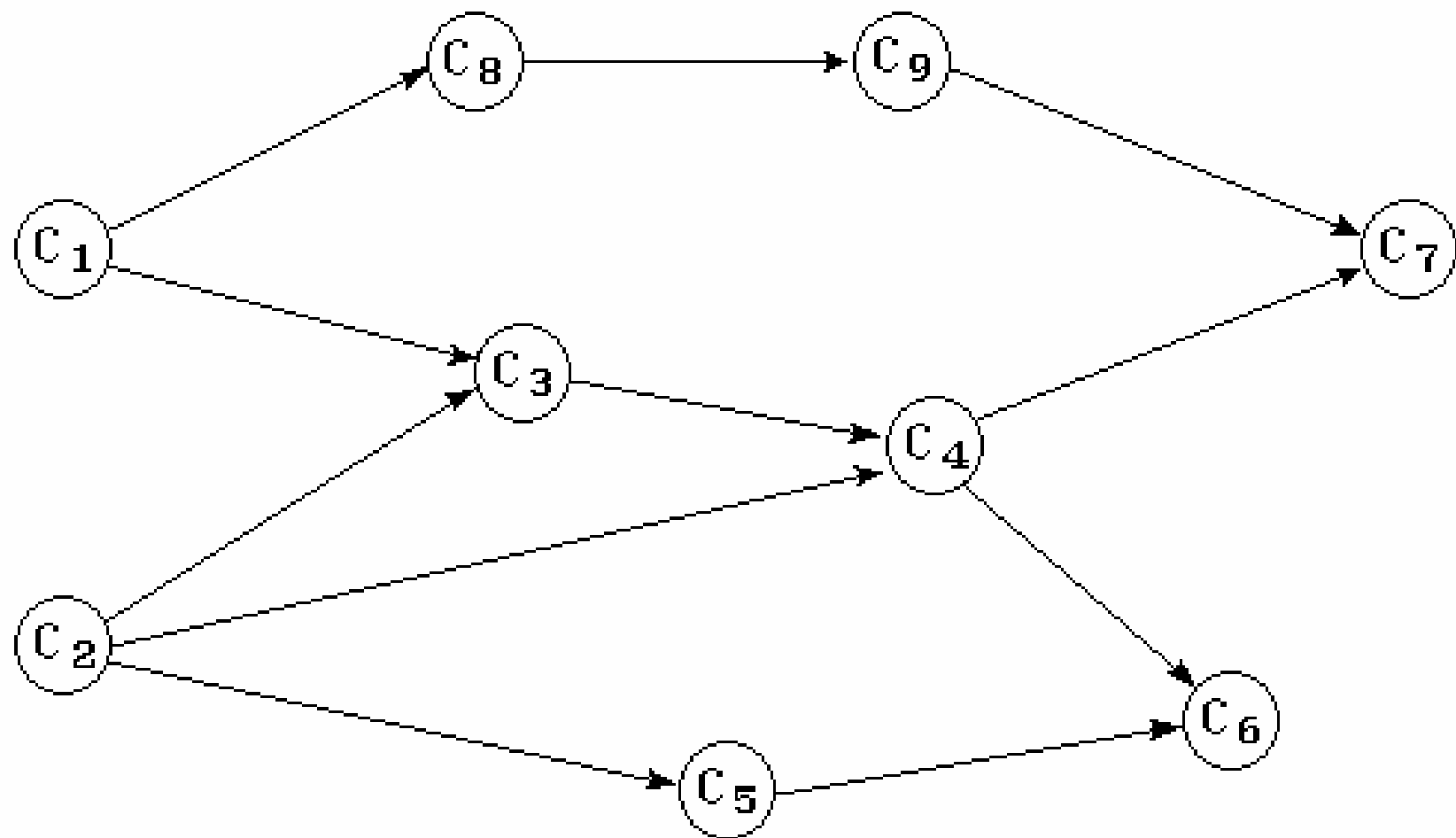
普通物理

C₁

C₉

计算机原理

C₈



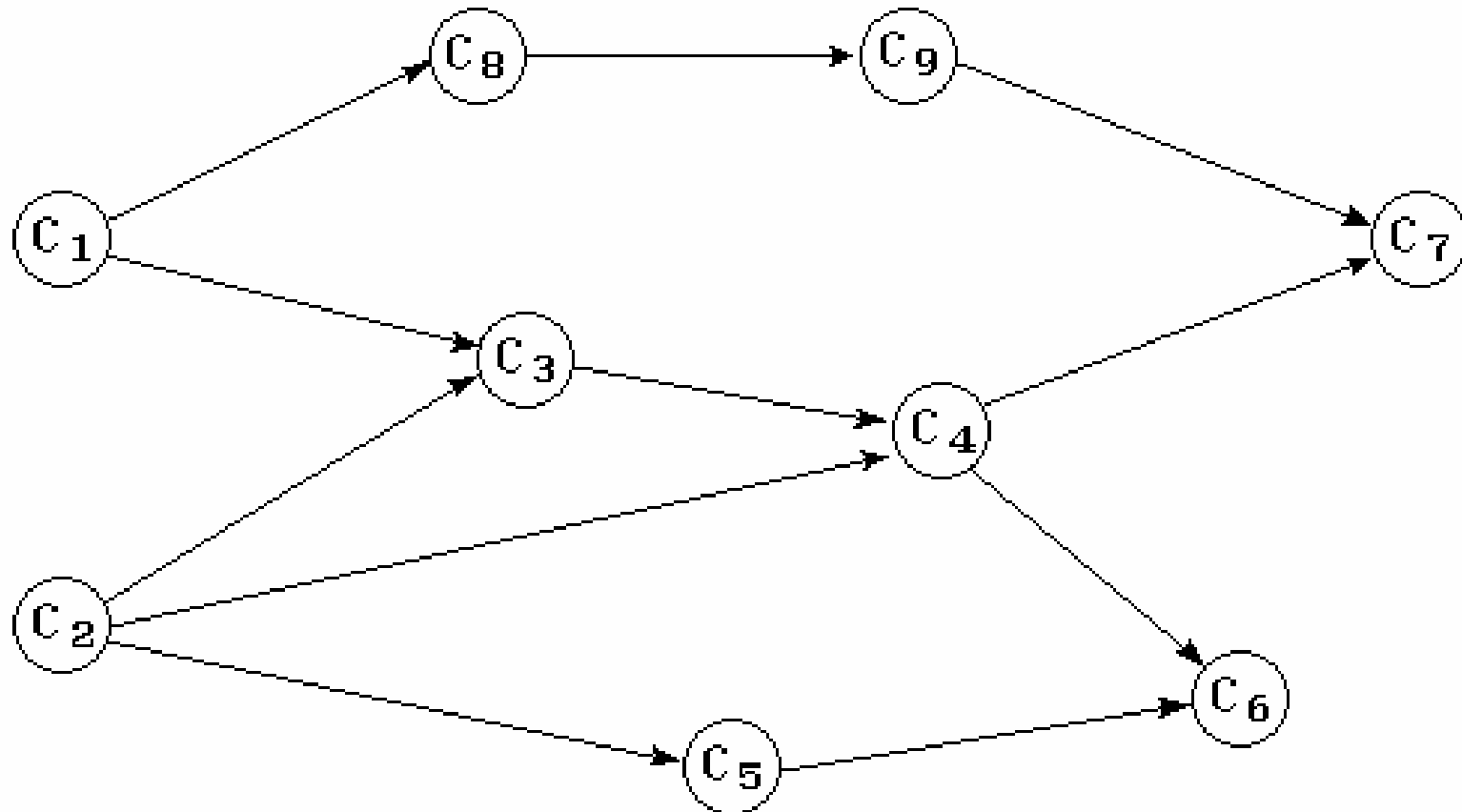
学生课程学习工程图

- 可以用有向图表示一个工程。在这种有向图中，用顶点表示活动，用有向边 $\langle V_i, V_j \rangle$ 表示活动间的优先关系。 V_i 必须先于活动 V_j 进行。这种有向图叫做顶点表示活动的AOV网络(Activity On Vertices)。
- 在AOV网络中，如果活动 V_i 必须在活动 V_j 之前进行，则存在有向边 $\langle V_i, V_j \rangle$ ，AOV网络中不能出现有向回路，即有向环。在AOV网络中如果出现了有向环，则意味着某项活动应以自己作为先决条件。
- 因此，对给定的AOV网络，必须先判断它是否存在有向环。

- 检测有向环的一种方法是对AOV网络构造它的**拓扑有序序列**。即将各个顶点(代表各个活动)排列成一个线性有序的序列,使得AOV网络中所有应存在的前驱和后继关系都能得到满足。
- 这种构造AOV网络全部顶点的拓扑有序序列的运算就叫做**拓扑排序**。
- 如果通过拓扑排序能将AOV网络的所有顶点都排入一个拓扑有序的序列中,则该AOV网络中必定不会出现有向环;相反,如果得不到满足要求的拓扑有序序列,则说明AOV网络中存在有向环,此AOV网络所代表的工程是不可行的。

- 例如，对学生选课工程图进行拓扑排序，得到的拓扑有序序列为

- $C_1, C_2, C_3, C_4, C_5, C_6, C_8, C_9, C_7$
或 $C_1, C_8, C_9, C_2, C_5, C_3, C_4, C_7, C_6$



1.拓扑排序

1) 定义：由某个集合上的一个偏序得到该集合上的一个全序，这个操作称之为拓扑排序——P180

- 偏序：指集合中仅有部分成员之间可比较

全序：指集合中全体成员之间均可比较

- 由偏序定义得到拓扑有序的操作便是拓扑排序

1.拓扑排序

2) AOV网——Activity On Vertex Network

——用顶点表示活动，用弧表示活动间的优先关系的有向图，称为顶点表示活动的网。

- 例如P181图 7.26 7.27

- AOV-网中不能有回路

3) 拓扑排序

(1) 结果是产生一个拓扑有序序列

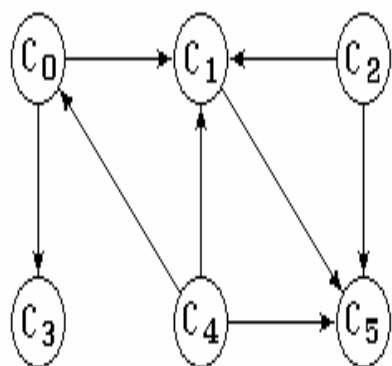
(2) 在拓扑排序的结果中：

- 序列的顶点数 = 图中的顶点数 无回路
- 序列的顶点数 < 图中的顶点数 有回路

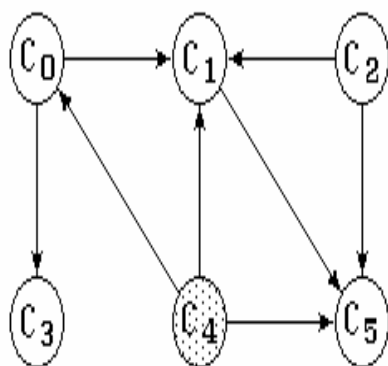
进行拓扑排序的方法

- ★ 输入AOV网络。令 n 为顶点个数。
- ⌚ 在AOV网络中选一个没有直接前驱的顶点，并输出之；
- ⌚ 从图中删去该顶点，同时删去所有它发出的有向边；
- ⌚ 重复以上 ⌚、⌚ 步，直到
 - ◆ 全部顶点均已输出，拓扑有序序列形成，拓扑排序完成；或
 - ◆ 图中还有未输出的顶点，但已跳出处理循环。这说明图中还剩下一一些顶点，它们都有直接前驱，再也找不到没有前驱的顶点了。这时AOV网络中必定存在有向环。

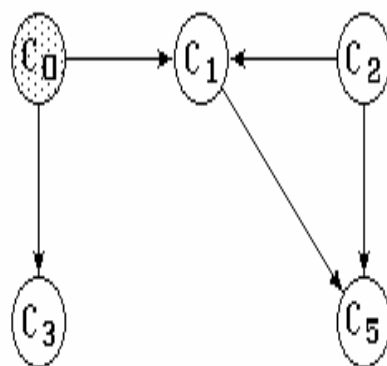
拓扑排序的过程



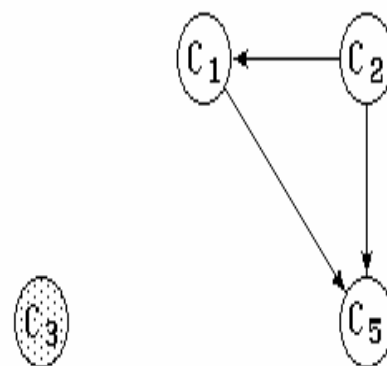
(a) 有向无环图



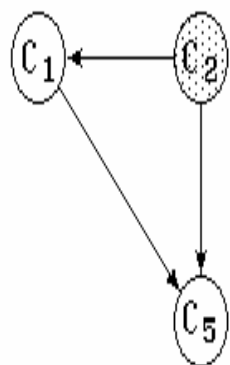
(b) 输出顶点 c_4



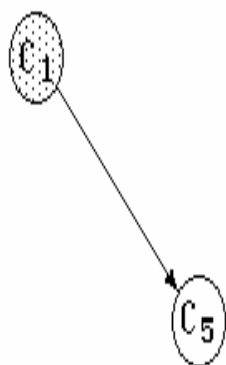
(c) 输出顶点 c_0



(d) 输出顶点 c_3



(e) 输出顶点 c_2

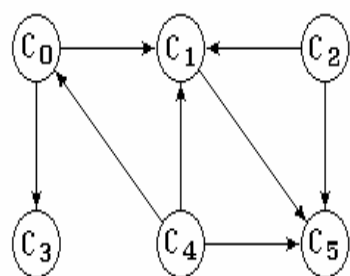


(f) 输出顶点 c_1

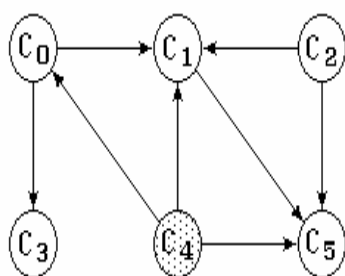


(g) 输出顶点 c_5

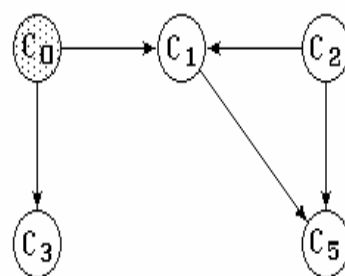
(h) 拓扑排序完成



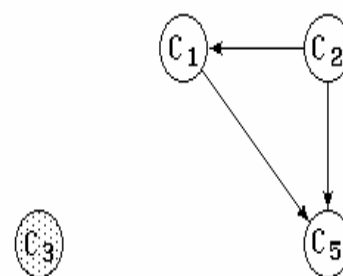
(a) 有向无环图



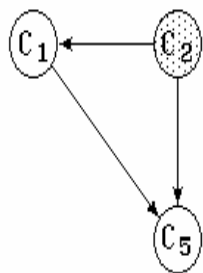
(b) 输出顶点 C_4



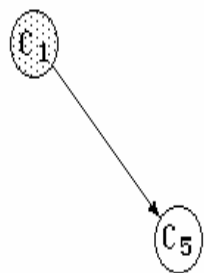
(c) 输出顶点 C_0



(d) 输出顶点 C_3



(e) 输出顶点 C_2



(f) 输出顶点 C_1

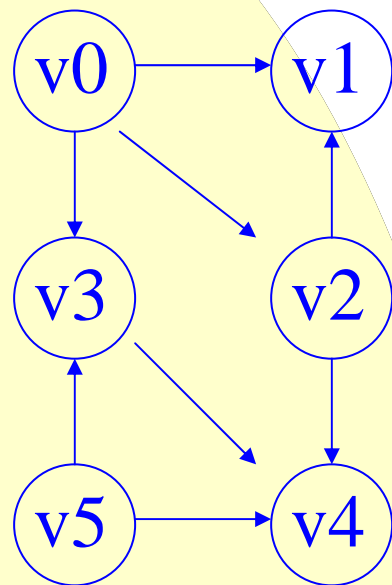


(g) 输出顶点 C_5

(h) 拓扑排序完成

最后得到的拓扑有序序列为 $C_4, C_0, C_3, C_2, C_1, C_5$ 。它满足图中给出的所有前驱和后继关系，对于本来没有这种关系的顶点，如 C_4 和 C_2 ，也排出了先后次序关系。

AOV网络及其邻接表表示



indegree

0	v0	0		→	1	→	2	→	3	^
1	v1	2	^							
2	v2	1		→	1	→	4	^		
3	v3	2		→	4	^				
4	v4	3	^							
5	v5	0		→	3	→	4	^		

顶点信息

入度

Firstarc ——以vexdata
为尾的第一条弧的邻接点

- 在邻接表的头结点中增设一个域indegree，记录各个顶点入度。入度为零的顶点即无前驱的顶点。
- 在输入数据前，邻接表的顶点表*G.vertices[]*.indegree全部初始化。在输入数据时，每输入一条边<j, k>，就需要建立一个边结点，并将它链入相应边链表中，统计入度信息：

//建立边结点, *data*域赋为 *k*

p → *nextarc* = *G.vertices[j].firstarc*;

G.vertices[j].firstarc = *p*;

//链入顶点 *j* 的边链表的前端

G.vertices[k].indegree++; //顶点 *k* 入度加一

- 在拓扑排序算法中，使用一个存放入度为零的顶点的链式栈，供选择和输出无前驱的顶点。只要出现入度为零的顶点，就将它加入栈中。
- 使用这种栈的拓扑排序算法可以描述如下：
 - (1) 建立入度为零的顶点栈；
 - (2) 当入度为零的顶点栈不空时，重复执行
 - ◆ 从顶点栈中退出一个顶点，并输出之；
 - ◆ 从AOV网络中删去这个顶点和它发出的边，边的终顶点入度减一；
 - ◆ 如果边的终顶点入度减至0，则该顶点进入度为零的顶点栈；
 - (3) 如果输出顶点个数少于AOV网络的顶点个数，则报告网络中存在有向环。

- 在算法实现时，为了建立入度为零的顶点栈，可以不另外分配存储空间，直接利用入度为零的顶点的 $indegree[]$ 数组元素。我们设立了一个栈顶指针 top ，指示当前栈顶的位置，即某一个入度为零的顶点位置。栈初始化时置 $top = -1$ ，表示空栈。

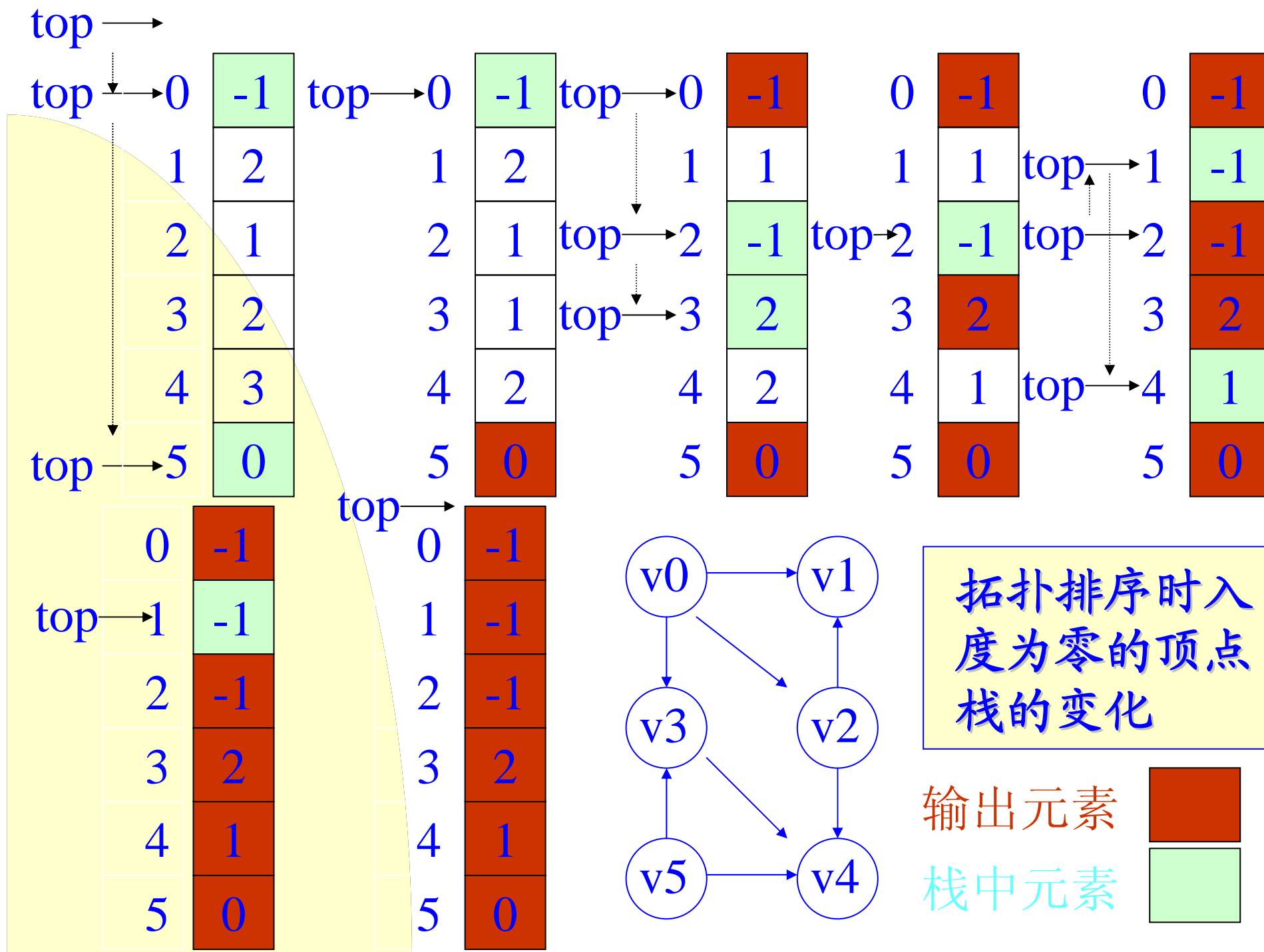
- 将顶点 i 进栈时，执行以下指针的修改：

$indegree[i] = top; top = i; //$ top 指向新栈顶 i ,
原栈顶元素放在 $indegree[i]$ 中

- 退栈操作可以写成：

$j = top; top = indegree[top];$

//位于栈顶的顶点的位置记于 j , top 退到次栈顶



typedef struct ArcNode{//图的邻接表的定义

int adjvex;

struct ArcNode *nextarc;

int info;

}ArcNode;

typedef struct VNode{

VertexType data;

int indegree;

ArcNode *firstarc;

}VNode,AdjList[MAX_VERTEX_NUM];

typedef struct{

AdjList vertices;

int vexnum,arcnum;

int kind; }ALGraph;

拓扑排序的算法(p182算法7.12的改进程序)

```
Status TopologicalSort(ALGraph G){
    ArcNode *p;
    int k;
    // FindIndegree(G,indegree);
    int top=-1; //入度为零的顶点栈初始化
    for (int i=0;i<G.vexnum;++i)
        if (G.vertices[i].indegree==0){
            G.vertices[i].indegree=top;
            top=i; //入度为零顶点进栈
        }
    int count=0;
```

```
while (top+1){ //top=-1表示没有入度为0顶点
    i=top;      top=G.vertices[top].indegree;
    printf("%c",G.vertices[i].data);
    ++count;
    for (p=G.vertices[i].firstarc;p;p=p-
>nextarc){ //扫描该顶点的出边表
        k=p->adjvex; //边的另一顶点
        G.vertices[k].indegree--; //顶点入度减1
        if (G.vertices[k].indegree==0){
            G.vertices[k].indegree=top;
            top=k;} //入度为0入栈
    }
}
```

```
if (count<G.vexnum) return ERROR;//有向环  
else return OK;
```

```
}
```

逆拓扑排序:

- ◆在有向图中选一个没有后继（出度为零）的顶点，并将它输出；
- ◆从有向图中删除该顶点及所有以它为头的弧；
- ◆重复以上操作，直至图中全部顶点均已输出，或者当前图中不存在无后继的顶点。

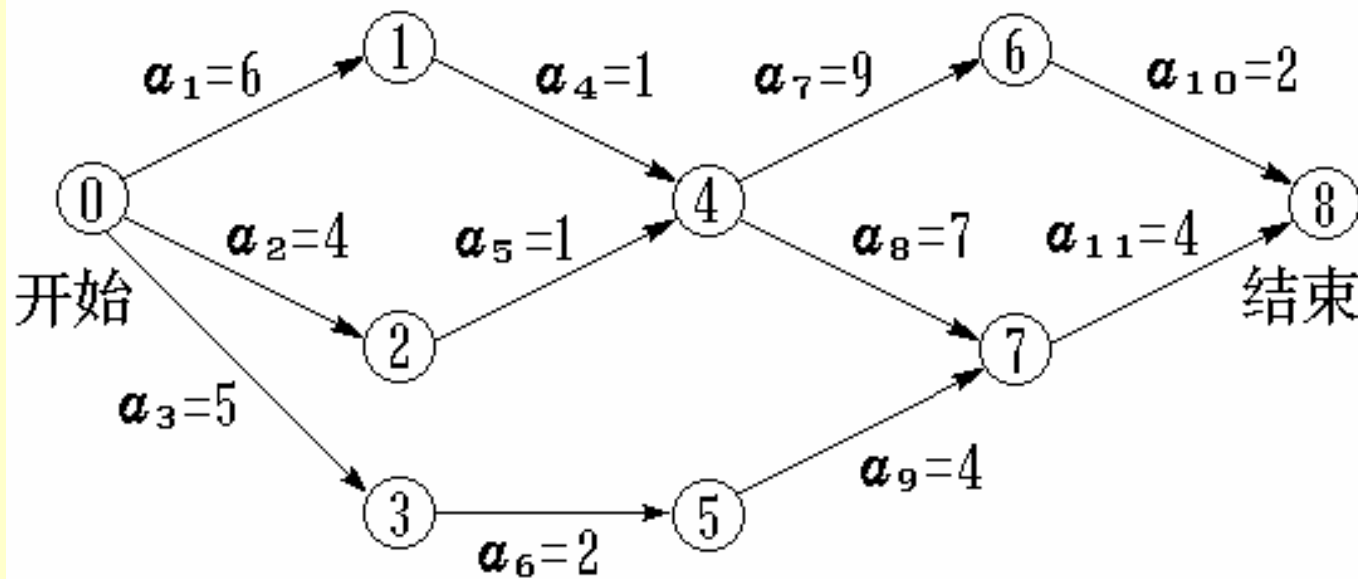
- 分析此拓扑排序算法可知，如果AOV网络有 n 个顶点， e 条边，在拓扑排序的过程中，搜索入度为零的顶点，建立链式栈所需要的时间是 $O(n)$ 。在正常的情况下，有向图有 n 个顶点，每个顶点进一次栈，出一次栈，共输出 n 次。顶点入度减一的运算共执行了 e 次。所以总的时间复杂度为 $O(n+e)$ 。

用边表示活动的网络(AOE网络)

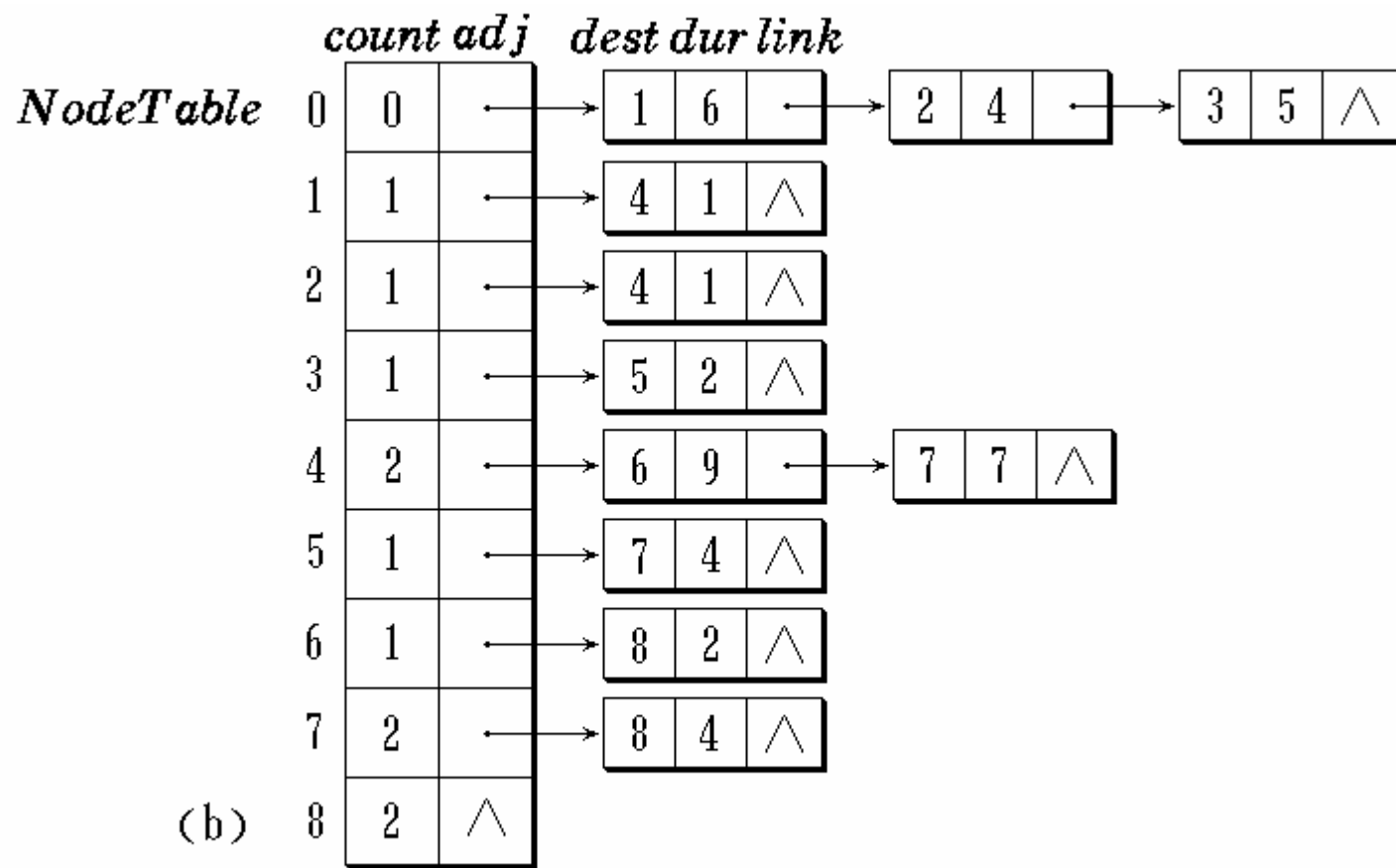
- 如果在无有向环的带权有向图中
 - ◆ 用有向边表示一个工程中的各项活动(Activity)
 - ◆ 用边上的权值表示活动的持续时间(Duration)
 - ◆ 用顶点表示事件(Event)
- 则这样的有向图叫做用边表示活动的网络，简称AOE (Activity On Edges)网络。
- AOE网络在某些工程估算方面非常有用。例如，可以使人们了解：
 - (1) 完成整个工程至少需要多少时间(假设网络中没有环)?
 - (2) 为缩短完成工程所需的时间,应当加快哪些活动?

- 在AOE网络中,有些活动顺序进行,有些活动并行进行。
- 从源点到各个顶点,以至从源点到汇点的有向路径可能不止一条。这些路径的长度也可能不同。完成不同路径的活动所需的时间虽然不同,但只有各条路径上所有活动都完成了,整个工程才算完成。
- 因此,完成整个工程所需的时间取决于从源点到汇点的最长路径长度,即在这条路径上所有活动的持续时间之和。这条路径长度最长的路径就叫做关键路径(Critical Path)。

- 要找出关键路径，必须找出**关键活动**，即不按期完成就会影响整个工程完成的活动。
- 关键路径上的所有活动都是关键活动。因此，只要找到了关键活动，就可以找到 **关键路径**



(a)



定义几个与计算关键活动有关的量:

☆ 事件 V_i 的最早可能开始时间 $Ve(i)$

是从源点 V_0 到顶点 V_i 的最长路径长度。

⌚ 事件 V_i 的最迟允许开始时间 $VL[i]$

是在保证汇点 V_{n-1} 在 $Ve[n-1]$ 时刻完成的前提下，事件 V_i 的允许的最迟开始时间。

⌚ 活动 a_k 的最早可能开始时间 $e[k]$

设活动 a_k 在边 $\langle V_i, V_j \rangle$ 上，则 $e[k]$ 是从源点 V_0 到顶点 V_i 的最长路径长度。因此， $e[k] = Ve[i]$ 。

⌚ 活动 a_k 的最迟允许开始时间 $l[k]$

$l[k]$ 是在不会引起时间延误的前提下，该活动允许的最迟开始时间。

$$l[k] = Vl[j] - dur(<i, j>).$$

其中, $dur(<i, j>)$ 是完成 a_k 所需的时间。



时间余量 $l[k] - e[k]$

表示活动 a_k 的最早可能开始时间和最迟允许开始时间的时间余量。 $l[k] == e[k]$ 表示活动 a_k 是没有时间余量的关键活动。

- 为找出关键活动, 需要求各个活动的 $e[k]$ 与 $l[k]$, 以判别是否 $l[k] == e[k]$.
- 为求得 $e[k]$ 与 $l[k]$, 需要先求得从源点 V_0 到各个顶点 V_i 的 $Ve[i]$ 和 $Vl[i]$.
- 求 $Ve[i]$ 的递推公式

- ◆ 从 $Ve[0] = 0$ 开始, 向前递推

$$Ve[i] = \max_j \{ Ve[j] + dur(< V_j, V_i >) \},$$
$$< V_j, V_i > \in S2, \quad i = 1, 2, \dots, n-1$$

其中, $S2$ 是所有指向顶点 V_i 的有向边 $< V_j, V_i >$ 的集合。

- ◆ 从 $Vl[n-1] = Ve[n-1]$ 开始, 反向递推

$$Vl[i] = \min_j \{ Vl[j] - dur(< V_i, V_j >) \},$$
$$< V_i, V_j > \in S1, \quad i = n-2, n-3, \dots, 0$$

其中, $S1$ 是所有从顶点 V_i 发出的有向边 $< V_i, V_j >$ 的集合。

- 这两个递推公式的计算必须分别在拓扑有序及逆拓扑有序的前提下进行。

- 设活动 a_k ($k = 1, 2, \dots, e$) 在带权有向边 $\langle V_i, V_j \rangle$ 上, 它的持续时间用 $dur(\langle V_i, V_j \rangle)$ 表示, 则有

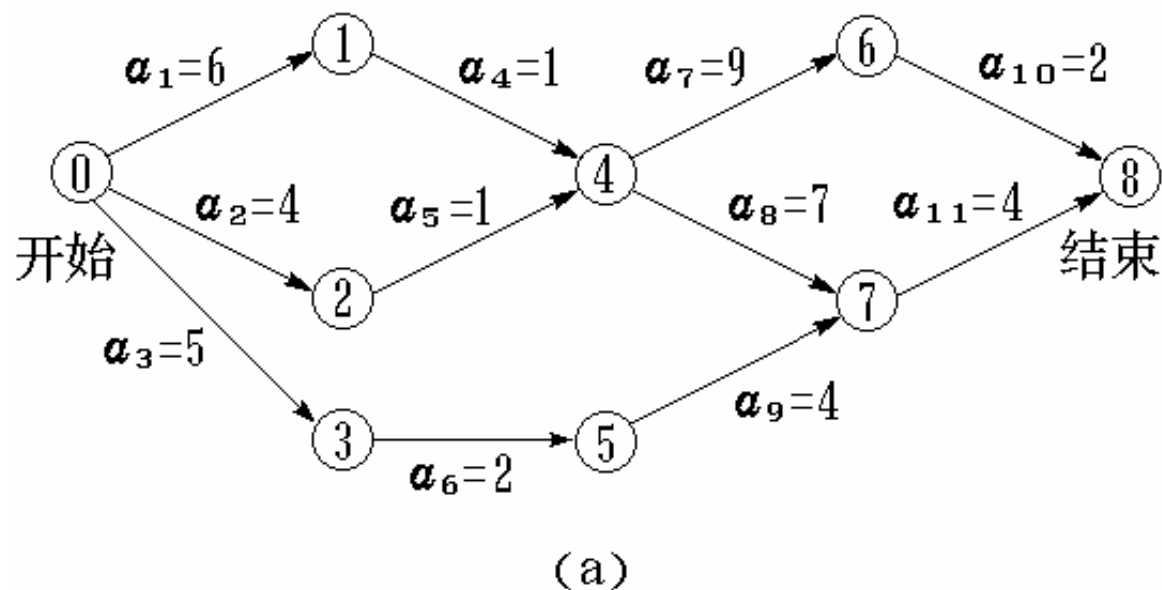
$$e[k] = Ve[i];$$

$$l[k] = Vl[j] - dur(\langle V_i, V_j \rangle); \quad k = 1, 2, \dots, e。$$

这样就得到计算关键路径的算法。

- 计算关键路径时, 可以一边进行拓扑排序一边计算各顶点的 $Ve[i]$ 。为了简化算法, 假定在求关键路径之前已经对各顶点实现了拓扑排序, 并按拓扑有序的顺序对各顶点重新进行了编号。算法在求 $Ve[i], i=0, 1, \dots, n-1$ 时按拓扑有序的顺序计算, 在求 $Vl[i], i=n-1, n-2, \dots, 0$ 时按逆拓扑有序的顺序计算。

事件	$Ve[i]$	$VL[i]$
V_0	0	0
V_1	6	6
V_2	4	6
V_3	5	8
V_4	7	7
V_5	7	10
V_6	16	16
V_7	14	14
V_8	18	18



边	$\langle 0,1 \rangle$	$\langle 0,2 \rangle$	$\langle 0,3 \rangle$	$\langle 1,4 \rangle$	$\langle 2,4 \rangle$	$\langle 3,5 \rangle$	$\langle 4,6 \rangle$	$\langle 4,7 \rangle$	$\langle 5,7 \rangle$	$\langle 6,8 \rangle$	$\langle 7,8 \rangle$
活动	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	a_9	a_{10}	a_{11}
e	0	0	0	6	4	5	7	7	7	16	14
l	0	2	3	6	6	8	7	7	10	16	14
$l - e$	0	2	3	0	2	3	0	0	3	0	0
关键	是			是			是	是		是	是

利用关键路径法求AOE网的各关键活动

```
void CriticalPath (ALGraph G) {  
    int i, j, k, e, l;      int *Ve,*Vl;      ArcNode *p;  
    Ve = new int[G.vexnum]; Vl = new int[G.vexnum];  
    for ( i = 0; i < G.vexnum; i++ ) Ve[i] = 0;  
    for ( i = 0; i < G.vexnum; i++ ) {  
        ArcNode *p = G.vertices[i].firstarc;  
        while ( p != NULL ) {  
            k = p->adjvex;  
            if (Ve[i] + p->info > Ve[k])  
                Ve[k] = Ve[i] + p->info;  
            p = p->nextarc;  
        }  
    }  
}
```



```
for ( i = 0; i < G.vexnum; i++ )
    V1[i] = Ve[G.vexnum-1];
for ( i = G.vexnum-2; i; i-- ) {
    p = G.vertices[i].firstarc;
    while ( p != NULL ) {
        k = p->adjvex;
        if ( V1[k] - p->info < V1[i])
            V1[i] = V1[k] - p->info;
        p = p->nextarc;
    }
}
```

```
for ( i = 0; i < G.vexnum; i++ ) {  
    p = G.vertices[i].firstarc;  
    while ( p != NULL ) {  
        k = p->adjvex;  
        e = Ve[i]; l = Vl[k] - p->info;  
        char tag= (e == l) ? '*' : ' ';  
        printf("(%c,%c),e=%d,l=%d,%c\n",G.vertices[i]  
].data,G.vertices[k].data,e,l,tag);  
        p = p->nextarc;    }  
    }  
}
```

在拓扑排序求 $Ve[i]$ 和逆拓扑有序求 $Vl[i]$ 时, 所需时间为 $O(n+e)$, 求各个活动的 $e[k]$ 和 $l[k]$ 时所需时间为 $O(e)$, 总共花费时间仍然是 $O(n+e)$ 。

作业:

第47页, 7.1: 1),2),3),4);

7.4;

7.7:1);

7.9只列出拓扑序列即可。