第5章 图论算法

图状结构是一种比树结构更复杂的非线性数据结构。在树结构中，结点间具有分支层次关系，每一层上的结点只能和上一层中的至多一个结点相关，但可能和下一层的多个结点相关。而在图状结构中，任意两个结点之间都可能相关，即结点之间的邻接关系可以是任意的。因此，图状结构被用于描述各种复杂的数据对象，在自然科学、社会科学和人文科学等许多领域有着非常广泛的应用，与之相关的实现算法会影响到许多实际应用问题的算法效率。

随着互联网和物联网的快速发展，未来的世界将会是一个更加智能化、生态化、自动化和便捷化的“万物互联”世界。数字城市、智慧交通、智能医疗等，将现实世界不同领域抽象成一个个的图状结构，利用图的最小生成树算法，可以建设低成本的通信网络和交通网络，进行最佳旅游景点路线规划，优化城市天然气管道铺设布局等；图的最短路径算法，可以实现最优的物流运输的路径，降低物流成本，还可以应用于自然灾害、矿井、航空等突发事件的应急救援中，以减少生命及财产损失。

观看视频

5.1 图

观看视频

5.1.1 图的定义和术语

1. 图的定义

图(Graph)由非空的顶点集合和一个描述顶点之间关系——边(或者弧)的集合组成，其形式化定义为

$$G=(V,E)$$
$$V=\{v_i \mid v_i \in \text{dataobject}\}$$
$$E=\{(v_i,v_j) \mid v_i, v_j \in V \wedge P(v_i, v_j)\}$$

其中，G 表示一幅图；V 是图 G 中顶点的集合；E 是图 G 中边的集合；集合 E 中 $P(v_i,v_j)$ 表示顶点 v_i 和顶点 v_j 之间有一条直接连线，即偶对 (v_i,v_j) 表示一条边。图 5.1 给出了一幅图的示例，在该图中：

集合 $V=\{v_1,v_2,v_3,v_4,v_5\}$

集合 $E=\{(v_1,v_2),(v_1,v_4),(v_2,v_3),(v_3,v_4),(v_3,v_5),(v_2,v_5)\}$

2. 图的相关术语

(1) 无向图。在一幅图中，如果任意两个顶点构成的偶对$(v_i, v_j) \in E$是无序的，即顶点之间的连线是没有方向的，则称该图为无向图。如图 5.1 所示是一幅无向图G_1。

(2) 有向图。在一幅图中，如果任意两个顶点构成的偶对$(v_i, v_j) \in E$是有序的，即顶点之间的连线是有方向的，则称该图为有向图。如图 5.2 所示是一幅有向图G_2。

$$G_2 = (V_2, E_2)$$

$$V_2 = \{v_1, v_2, v_3, v_4\}$$

$$E_2 = \{<v_1, v_2>, <v_1, v_3>, <v_3, v_4>, <v_4, v_1>\}$$

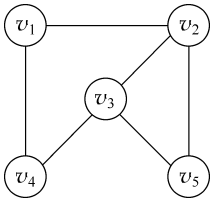


图 5.1　无向图G_1

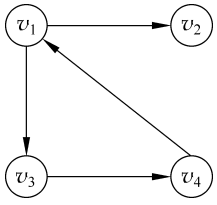


图 5.2　有向图G_2

(3) 顶点、边、弧、弧头、弧尾。图中，数据元素v_i称为顶点(Vertex)；$P(v_i, v_j)$表示在顶点v_i和顶点v_j之间有一条直接连线。如果是在无向图中，则称这条连线为边；如果是在有向图中，一般称这条连线为弧。边用顶点的无序偶对(v_i, v_j)来表示，称顶点v_i和顶点v_j互为邻接点，边(v_i, v_j)依附于顶点v_i与顶点v_j；弧用顶点的有序偶对$<v_i, v_j>$来表示，有序偶对的第一个结点v_i被称为始点(或弧尾)，在图中就是不带箭头的一端；有序偶对的第二个结点v_j被称为终点(或弧头)，在图中就是带箭头的一端。

(4) 无向完全图。在一幅无向图中，如果任意两个顶点都有一条直接边相连接，则称该图为无向完全图。可以证明，在一幅含有n个顶点的无向完全图中，有$n(n-1)/2$条边。

(5) 有向完全图。在一幅有向图中，如果任意两个顶点之间都有方向互为相反的两条弧相连接，则称该图为有向完全图。在一幅含有n个顶点的有向完全图中，有$n(n-1)$条边。

(6) 稠密图、稀疏图。若一幅图接近完全图，称为稠密图；称边数很少的图为稀疏图。

(7) 顶点的度、入度、出度。顶点的度(Degree)指依附于某顶点v的边数，通常记为$TD(v)$。在有向图中，要区别顶点的入度与出度的概念。顶点v的入度指以顶点为终点的弧的数目，记为$ID(v)$；顶点v的出度指以顶点v为始点的弧的数目，记为$OD(v)$。有$TD(v) = ID(v) + OD(v)$。

例如，在G_1中有

$$TD(v_1)=2 \quad TD(v_2)=3 \quad TD(v_3)=3 \quad TD(v_4)=2 \quad TD(v_5)=2$$

在G_2中有

$$ID(v_1)=1 \quad OD(v_1)=2 \quad TD(v_1)=3$$

$$ID(v_2)=1 \quad OD(v_2)=0 \quad TD(v_2)=1$$

$$ID(v_3)=1 \quad OD(v_3)=1 \quad TD(v_3)=2$$

$$ID(v_4)=1 \quad OD(v_4)=1 \quad TD(v_4)=2$$

可以证明，对于具有 n 个顶点、e 条边的图，顶点 v_i 的度 $\mathrm{TD}(v_i)$ 与顶点的个数以及边的数目满足关系：

$$e=\left(\sum_{i=1}^{n}\mathrm{TD}(v_i)\right)\Big/2$$


观看视频

（8）边的权、网图。与边有关的数据信息称为权(Weight)。在实际应用中，权值可以有某种含义。例如，在一幅反映城市交通线路的图中，边上的权值可以表示该条线路的长度或者等级；对于一幅电子线路图，边上的权值可以表示两个端点之间的电阻、电流或电压值；对于反映工程进度的图而言，边上的权值可以表示从前一个工程到后一个工程所需要的时间等。边上带权的图称为网图或网络(Network)。如图 5.3 所示就是一幅无向网图。如果边是有方向的带权图，则就是一幅有向网图。

（9）路径、路径长度。顶点 v_p 到顶点 v_q 之间的路径(Path)指顶点序列 v_p，v_{i1}，v_{i2}，…，v_{im}，v_q。其中，(v_p,v_{i1})，(v_{i1},v_{i2})，…，(v_{im},v_q)分别为图中的边。路径上边的数目称为路径长度。在图 5.1 所示的无向图 G_1 中，$v_1\to v_4\to v_3\to v_5$ 与 $v_1\to v_2\to v_5$ 是从顶点 v_1 到顶点 v_5 的两条路径，路径长度分别为 3 和 2。

（10）回路、简单路径、简单回路。第一个顶点和最后一个顶点相同的路径称为回路或者环(Cycle)。序列中顶点不重复出现的路径称为简单路径。在图 5.1 中，前面提到的 v_1 到 v_5 的两条路径都为简单路径。除第一个顶点与最后一个顶点之外，其他顶点不重复出现的回路称为简单回路，或者简单环。如图 5.2 中所示的 $v_1\to v_3\to v_4\to v_1$。

（11）子图。对于图 $G=(V,E)$，$G'=(V',E')$，若存在 V' 是 V 的子集，E' 是 E 的子集，则称图 G' 是 G 的一幅子图。图 5.4 分别给出了 G_2 和 G_1 的两个子图 G' 和 G''。

（12）连通的、连通图、连通分量。在无向图中，如果从一个顶点 v_i 到另一个顶点 v_j $(i\neq j)$ 有路径，则称顶点 v_i 和 v_j 是连通的。如果图中任意两个顶点都是连通的，则称该图是连通图。无向图的极大连通子图称为连通分量。图 5.5(a)中有两个连通分量，如图 5.5(b)所示。

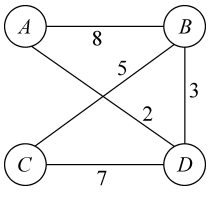
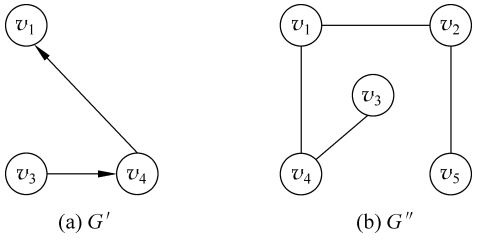


图 5.3　一幅无向网图示意

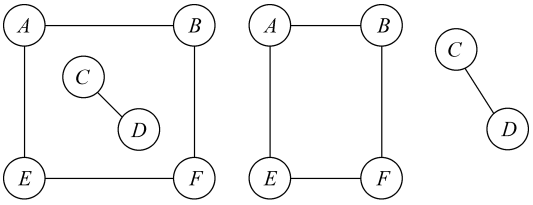


(a) G'　　(b) G''

图 5.4　图 G_2 和 G_1 的两幅子图示意

（13）强连通图、强连通分量。对于有向图来说，若图中任意一对顶点 v_i 和 v_j $(i\neq j)$ 均有从一个顶点 v_i 到另一个顶点 v_j 的路径，也有从 v_j 到 v_i 的路径，则称该有向图是强连通图。有向图的极大强连通子图称为强连通分量。图 5.2 中有两个强连通分量，分别是$\{v_1,v_3,v_4\}$和$\{v_2\}$，如图 5.6 所示。

（14）生成树。所谓连通图 G 的生成树，指包含 G 的全部 n 个顶点的一幅极小连通子图。它必定包含且仅包含 G 的 $n-1$ 条边。图 5.4(b)给出了图 5.1 中 G_1 的一棵生成树。在生成树中添加任意一条属于原图中的边必定会产生回路，因为新添加的边使其所依附的两个顶点之间有了第二条路径。若生成树中减少任意一条边，则必然成为非连通的。



(a) 无向图G_3　　(b) G_3的两个连通分量

图 5.5　无向图及连通分量示意

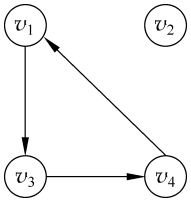


图 5.6　有向图 G_2 的两个强连通分量示意

(15) 生成森林。在非连通图中,由每个连通分量都可得到一幅极小连通子图,即一棵生成树,这些连通分量的生成树就组成了一幅非连通图的生成森林。

5.1.2　图的抽象数据类型


观看视频

图的抽象数据类型定义如下。

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}。
基本操作如下。
(1) CreateGraph(&G,V,VR): 按 V 和 VR 的定义构造图 G。
(2) DestroyGraph(&G): 销毁图 G。
(3) Locatevex(G,u): 若 G 中存在顶点 u,则返回该顶点在图中的位置; 否则返回其他信息。
(4) Getvex(G,v): 返回顶点 v 的值。
(5) PutVex(&G,v,value): 对顶点 v 赋值 value。
(6) FirstAdjVex(G,v): 返回顶点 v 的第一个邻接顶点。若顶点在 G 中没有邻接顶点,则返回"空"。
(7) NextAdjVex(G,v,w): 返回顶点 v 的(相对于 w 的)下一个邻接顶点.若 w 是 v 的最后一个邻接点,则返回"空"。
(8) InsertVex(&G,v): 在图 G 中增添新顶点 v。
(9) DeleteVex(&G,v,w): 删除 G 中顶点 v 及其相关的弧。
(10) InsertArc(&G,v,w): 在 G 中增添弧< v,w >,若 G 是有向的,则还增添对称弧< w,v >。
(11) DeleteArc(&G,v,w): 在 G 中删除弧< v,w >,若 G 是有向的,则还删除对称弧< w,v >。
(12) DFSTraverse(G,Visit()): 对图进行深度优先遍历。在遍历过程中对每个顶点调用函数 Visit()一次且仅一次。一旦 Visit()失败,则操作失败。
(13) BFSTraverse(G,Visit()): 对图进行广度优先遍历。在遍历过程中对每个顶点调用函数 Visit()一次且仅一次。一旦 Visit()失败,则操作失败。
}ADT Graph

5.1.3　图的存储结构


观看视频

图是一种结构复杂的数据结构,表现在不仅各顶点的度可以千差万别,而且顶点之间的逻辑关系也错综复杂。从图的定义可知,一幅图的信息包括两部分,即图中顶点的信息以及描述顶点之间的关系(边或者弧的信息)。因此无论采用什么方法建立图的存储结构,都要完整、准确地反映这两方面的信息。

1. 邻接矩阵

邻接矩阵(Adjacency Matrix)的存储结构就是用一维数组存储图中顶点的信息，用矩阵表示图中各顶点之间的邻接关系。如图 5.7 所示，假设图 $G=(V,E)$有 n 个确定的顶点，即 $V=\{v_0,v_1,\cdots,v_{n-1}\}$，则表示 G 中各顶点相邻关系为一个 $n\times n$ 的矩阵，矩阵的元素为

$$\boldsymbol{A}[i][j]=\begin{cases}1, & (v_i,v_j)\text{或}<v_i,v_j>\text{是}E(G)\text{中的边}\\ 0, & (v_i,v_j)\text{或}<v_i,v_j>\text{不是}E(G)\text{中的边}\end{cases}$$

如图 5.8 所示，若 G 是网图，则邻接矩阵可定义为

$$\boldsymbol{A}[i][j]=\begin{cases}w_{ij}, & (v_i,v_j)\text{或}<v_i,v_j>\text{是}E(G)\text{中的边}\\ 0\text{或}\infty, & (v_i,v_j)\text{或}<v_i,v_j>\text{不是}E(G)\text{中的边}\end{cases}$$

其中，w_{ij} 表示边(v_i,v_j)或$<v_i,v_j>$上的权值；∞表示一个计算机允许的、大于所有边上权值的数。

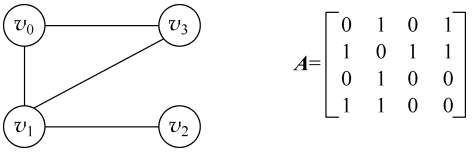


图 5.7 一幅无向图的邻接矩阵表示

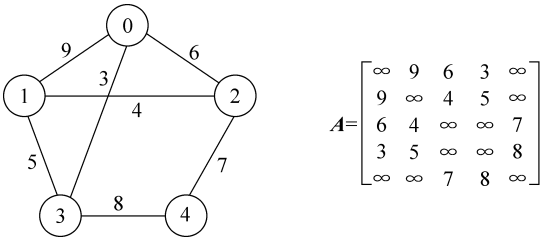


图 5.8 一幅网图的邻接矩阵表示

从图的邻接矩阵存储方法容易看出这种表示具有以下特点。

(1) 无向图的邻接矩阵一定是一个对称矩阵。因此，在具体存放邻接矩阵时只需存放上三角(或下三角)矩阵的元素即可。

(2) 对于无向图，邻接矩阵的第 i 行(或第 i 列)非零元素(或非∞元素)的个数正好是第 i 个顶点的度 $\mathrm{TD}(v_i)$。

(3) 对于有向图，邻接矩阵的第 i 行(或第 i 列)非零元素(或非∞元素)的个数正好是第 i 个顶点的出度 $\mathrm{OD}(v_i)$(或入度 $\mathrm{ID}(v_i)$)。

(4) 用邻接矩阵方法存储图，很容易确定图中任意两个顶点之间是否有边相连；但是，要确定图中有多少条边，则必须按行、按列对每个元素进行检测，所花费的时间代价很大。这是用邻接矩阵存储图的局限性。

在用邻接矩阵存储图时，除用一个二维数组存储用于表示顶点间相邻关系的邻接矩阵外，还需用一个一维数组来存储顶点信息，另外还有图的顶点数和边数。故可将其形式描述如下。

```
#define MaxVertexNum 100                    /* 最大顶点数设为 100 */
typedef char VertexType;                    /* 顶点类型设为字符型 */
typedef int EdgeType;                       /* 边的权值设为整型 */
typedef struct {
   VertexType vexs[MaxVertexNum];           /* 顶点表 */
```

```
   EdgeType edges[MaxVertexNum][MaxVertexNum];  /* 邻接矩阵,即边表 */
   int n,e;                                     /* 顶点数和边数 */
  }MGragh;                                      /* MGragh 是以邻接矩阵存储的图类型 */
```

建立一幅图的邻接矩阵存储的算法如下。

算法 5.1 建立有向图的邻接矩阵存储

```
void CreateMGraph(MGraph *G)
{                                               /* 建立有向图 G 的邻接矩阵存储 */
   int i,j,k,w;
   char ch;
   printf("请输入顶点数和边数(输入格式为:顶点数,边数):\n");
   scanf("%d,%d",&(G->n),&(G->e));              /* 输入顶点数和边数 */
   printf("请输入顶点信息(输入格式为:顶点号<CR>):\n");
   for (i=0;i<G->n;i++) scanf("\n%c",&(G->vexs[i]));
                                                /* 输入顶点信息,建立顶点表 */
   for (i=0;i<G->n;i++)
       for (j=0;j<G->n;j++) G->edges[i][j]=0;          /* 初始化邻接矩阵 */
   printf("请输入每条边对应的两个顶点的序号(输入格式为:i,j):\n");
   for (k=0;k<G->e;k++)
   { scanf("\n%d,%d",&i,&j);                    /* 输入 e 条边,建立邻接矩阵 */
       G->edges[i][j]=1;                        /* 若加入 G->edges[j][i]=1;, */
                                                /* 则建立完整的无向图邻接矩阵 */
    }
 }/* CreateMGraph */
```

2. 邻接表

邻接表(Adjacency List)是图的一种顺序存储与链式存储结合的存储方法。邻接表表示法类似于树的孩子链表表示法。就是对于图 G 中的每个顶点 v_i,将所有邻接于 v_i 的顶点 v_j 链成一个单链表,这个单链表就称为顶点 v_i 的邻接表,再将所有顶点的邻接表表头放到数组中,就构成了图的邻接表。在邻接表表示中有两种结点结构,如图 5.9 所示。

一种是顶点表的结点结构,它由顶点域(vertex)和指向第一条邻接边的指针域(firstedge)构成,另一种是边表(即邻接表)结点,它由邻接点域(adjvex)和指向下一条邻接边的指针域(next)构成。对于网图的边表需再增设一个存储边上权值信息的域(info)。网图的边表结构如图 5.10 所示。

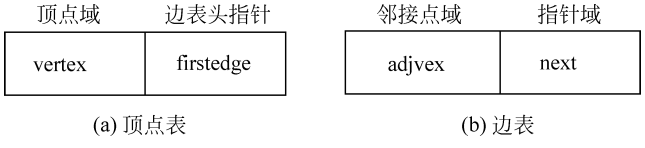


图 5.9 邻接矩阵表示的结点结构

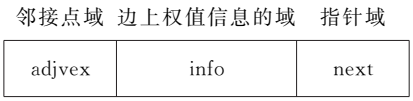


图 5.10 网图的边表结构

图 5.11 给出了无向图 5.7 对应的邻接表表示。

邻接表表示的形式描述如下。

```
#define MaxVerNum 100                   /* 最大顶点数为 100 */
typedef struct node{                    /* 边表结点 */
       int adjvex;                      /* 邻接点域 */
```

```
    struct node * next;                    /* 指向下一个邻接点的指针域 */
                                           /* 若要表示边上信息,则应增加一个数据域 info */
   }EdgeNode;
typedef struct vnode{                      /* 顶点表结点 */
    VertexType vertex;                     /* 顶点域 */
    EdgeNode * firstedge;                  /* 边表头指针 */
   }VertexNode;
typedef VertexNode AdjList[MaxVertexNum];  /* AdjList 是邻接表类型 */
typedef struct{
   AdjList adjlist;                        /* 邻接表 */
   int n,e;                                /* 顶点数和边数 */
  }ALGraph;                                /* ALGraph 是以邻接表方式存储的图类型 */
```

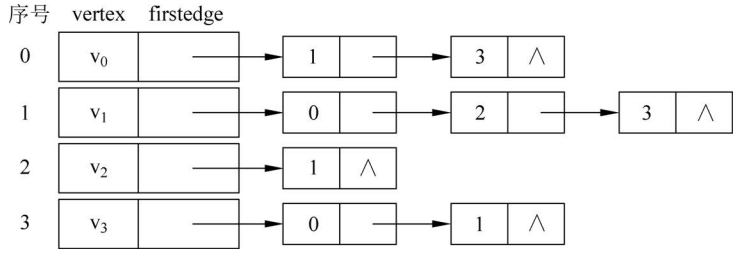


图 5.11　图的邻接表表示

建立一个有向图的邻接表存储的算法如下。

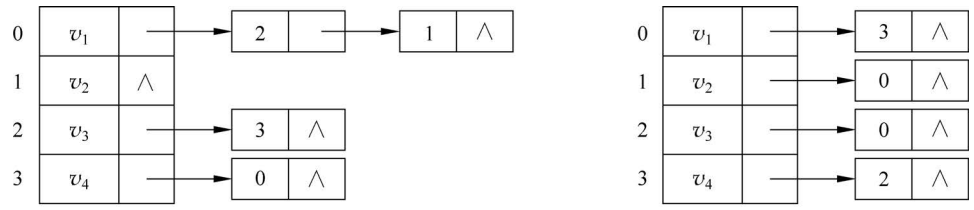
算法 5.2　建立有向图的邻接表存储

```
void CreateALGraph(ALGraph * G)
{                                          /* 建立有向图的邻接表存储 */
   int i,j,k;
   EdgeNode * s;
   printf("请输入顶点数和边数(输入格式为:顶点数,边数): \n");
   scanf("%d,%d",&(G->n),&(G->e)); /* 读入顶点数和边数 */
   printf("请输入顶点信息(输入格式为:顶点号<CR>): \n");
   for (i=0;i<G->n;i++)                    /* 建立有 n 个顶点的顶点表 */
   { scanf("\n%c",&(G->adjlist[i].vertex)); /* 读入顶点信息 */
       G->adjlist[i].firstedge=NULL;       /* 顶点的边表头指针设为空 */
    }
   printf("请输入边的信息(输入格式为:i,j): \n");
   for (k=0;k<G->e;k++)                    /* 建立边表 */
   { scanf("\n%d,%d",&i,&j);               /* 读入边<vi,vj>的顶点对应序号 */
       s=(EdgeNode *)malloc(sizeof(EdgeNode));    /* 生成新边表结点 s */
       s->adjvex=j;                        /* 邻接点序号为 j */
       s->next=G->adjlist[i].firstedge; /* 将新边表结点 s 插入顶点 vi 的边表头部 */
       G->adjlist[i].firstedge=s;
   }
}                                          /* CreateALGraph */
```

若无向图中有 n 个顶点、e 条边，则它的邻接表需 n 个头结点和 $2e$ 个表结点。显然，在边稀疏($e<<n(n-1)/2$)的情况下，用邻接表表示图比邻接矩阵节省存储空间，当和边相关的信息较多时更是如此。

在无向图的邻接表中，顶点 v_i 的度恰为第 i 个链表中的结点数；而在有向图中，第 i 个

链表中的结点个数只是顶点 v_i 的出度，为求入度，必须遍历整个邻接表。在所有链表中其邻接点域的值为 i 的结点的个数是顶点 v_i 的入度。有时，为了便于确定顶点的入度或以顶点 v_i 为头的弧，可以建立一个有向图的逆邻接表，即对每个顶点 v_i 建立一个链接以 v_i 为头的弧的链表。例如，图 5.12 所示为有向图 G_2（图 5.2）的邻接表和逆邻接表。



(a) 邻接表　　(b) 逆邻接表

图 5.12　图 5.2 的邻接表和逆邻接表

在建立邻接表或逆邻接表时，若输入的顶点信息为顶点的编号，则建立邻接表的时间复杂度为 $O(n+e)$；否则，需要通过查找才能得到顶点在图中的位置，时间复杂度为 $O(n*e)$。

在邻接表上容易找到任一顶点的第一个邻接点和下一个邻接点，但要判定任意两个顶点（v_i 和 v_j）之间是否有边或弧相连，则需搜索第 i 个或第 j 个链表，因此，不及邻接矩阵方便。

3. 十字链表


观看视频

十字链表（Orthogonal List）是有向图的一种存储方法，它实际上是邻接表与逆邻接表的结合，即把每一条边的边结点分别组织到以弧尾顶点为头结点的链表和以弧头顶点为头顶点的链表中。在十字链表表示中，顶点表和边表的结点结构分别如图 5.13(a)和图 5.13(b)所示。

顶点值域	指针域	指针域
vertex	firstin	firstout

(a) 十字链表顶点表的结点结构

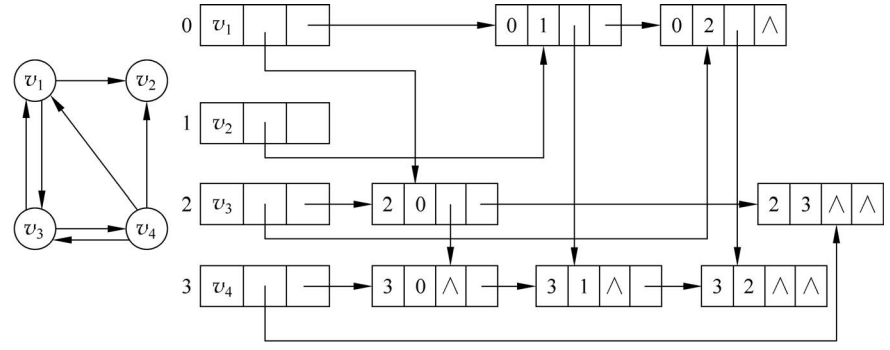
弧尾结点	弧头结点	弧上信息	指针域	指针域
tailvex	headvex	info	hlink	tlink

(b) 十字链表边表的结点结构

图 5.13　十字链表顶点表、边表的结点结构示意

在弧结点中有 5 个域，其中弧尾结点（tailvex）和弧头结点（headvex）分别指示弧尾和弧头这两个顶点在图中的位置，指针域 hlink 指向弧头相同的下一条弧，指针域 tlink 指向弧尾相同的下一条弧，info 域指向该弧的相关信息。弧头相同的弧在同一链表上，弧尾相同的弧也在同一链表上。它们的头结点即为顶点结点，它由 3 个域组成：vertex 域存储和顶点相关的信息，如顶点的名称等；firstin 和 firstout 为两个链域，分别指向以该顶点为弧头或弧尾的第一个弧结点。例如，图 5.14(a)中所示有向图的十字链表如图 5.14(b)所示。若将有向图的邻接矩阵看成稀疏矩阵，则十字链表也可以看成邻接矩阵的链表存储结构。在图

的十字链表中，弧结点所在的链表为非循环链表，结点之间相对位置自然形成，不一定按顶点序号有序，表头结点即顶点结点，它们之间是顺序存储的。



(a) 一幅有向图G_4　　(b) 有向图的十字链表

图 5.14　有向图及其十字链表表示示意

有向图的十字链表存储表示的形式描述如下。

```
#define MAX_VERTEX_NUM 20
typedef struct ArcBox {
    int tailvex,headvex;                    /* 该弧的尾和头顶点的位置 */
    struct ArcBox * hlink, tlink;           /* 分别为弧头相同和弧尾相同的弧的链域 */
    InfoType info;                          /* 该弧相关信息的指针 */
}ArcBox;
typedef struct VexNode {
    VertexType vertex:
    ArcBox fisrin, firstout;                /* 分别指向该顶点的第一条入弧和出弧 */
}VexNode;
typedef struct {
    VexNode xlist[MAX_VERTEX_NUM];          /* 表头向量 */
    int vexnum,arcnum;                      /* 有向图的顶点数和弧数 */
}OLGraph;
```

下面给出建立一个有向图的十字链表存储的算法。通过该算法，只要输入 n 个顶点的信息和 e 条弧的信息，便可建立该有向图的十字链表，其算法如下。

算法 5.3　建立有向图的十字链表

```
void CreateDG(LOGraph * *G)
/* 采用十字链表表示，构造有向图 G(G.kind = DG) */
{   scanf(&( *G->brcnum),&( *G->arcnum),&IncInfo);
                                            /* IncInfo 为 0 则各弧不含信息 */
    for (i = 0;i<*G->vexnum;++i)            /* 构造表头向量 */
     { scanf(&(G->xlist[i].vertex));        /* 输入顶点值 */
       *G->xlist[i].firstin = NulL; *G->xlist[i].firstout = NULL;  /* 初始化指针 */
     }
    for(k = 0;k<G.arcnum;++k)               /* 输入各弧并构造十字链表 */
    { scanf(&v1,&v2);                       /* 输入一条弧的始点和终点 */
       i = LocateVex( *G,v1); j = LocateVex( *G,v2);      /* 确定 v1 和 v2 在 G 中的位置 */
       p = (ArcBox * ) malloc(sizeof(ArcBox));     /* 假定有足够的空间 */
        *p = {i,j, *G->xlist[j].fistin, *G->xlist[i].firstout,NULL}
                                            /* 对弧结点赋值 */
```

```
                                                    /*{tailvex,headvex,hlink,tlink,info}*/
        *G->xlist[j].fisrtin = *G->xlist[i].firstout = p;
                                                    /*完成在入弧和出弧链头的插入*/
      if (IncInfo) Input( p->info);                 /*若弧含有相关信息,则输入*/
    }
  }/*CreateDG*/
```

在十字链表中既容易找到以 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(或视需要在建立十字链表的同时求出)。同时，由算法 5.3 可知，建立十字链表的时间复杂度和建立邻接表是相同的。在某些有向图的应用中，十字链表是很有用的工具。

4. 邻接多重表

邻接多重表(Adjacency Multilist)主要用于存储无向图。因为，如果用邻接表存储无向图，那么每条边的两个边结点分别在以该边所依附的两个顶点为头结点的链表中，这会给图的某些操作带来不便。例如，对已访问过的边进行标记，或者要删除图中某一条边等，都需要找到表示同一条边的两个结点。因此，在进行这一类操作的无向图的问题中采用邻接多重表作存储结构更为适宜。

邻接多重表的存储结构和十字链表类似，也是由顶点表和边表组成的，每条边用一个结点表示，其顶点表结点结构和边表结点结构如图 5.15 所示。

顶点值域	指针域
vertex	firstedge

(a) 邻接多重表的顶点表结点结构

标记域	顶点位置	指针域	顶点位置	指针域	边上信息
mark	ivex	ilink	jvex	jlink	info

(b) 邻接多重表的边表结点结构

图 5.15　邻接多重表顶点表、边表的结点结构示意

其中，顶点表由两个域组成，vertex 域存储和该顶点相关的信息，firstedge 域指示第一条依附于该顶点的边。边表结点由 6 个域组成，mark 为标记域，可用于标记该条边是否被搜索过；ivex 和 jvex 为该边依附的两个顶点在图中的位置；ilink 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；info 为指向和边相关的各种信息的指针域。

例如，图 5.16 所示为无向图 5.1 的邻接多重表。在邻接多重表中，所有依附于同一顶点的边串联在同一链表中，由于每条边依附于两个顶点，因此每个边结点同时链接在两个链表中。可见，对无向图而言，其邻接多重表和邻接表的差别仅仅在于同一条边在邻接表中用两个结点表示，而在邻接多重表中只有一个结点。因此，除在边结点中增加一个标志域外，邻接多重表所需的存储量和邻接表相同。在邻接多重表上，各种基本操作的实现也和邻接表相似。邻接多重表存储表示的形式描述如下。

```
#define MAX_VERTEX_NUM 20
```

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

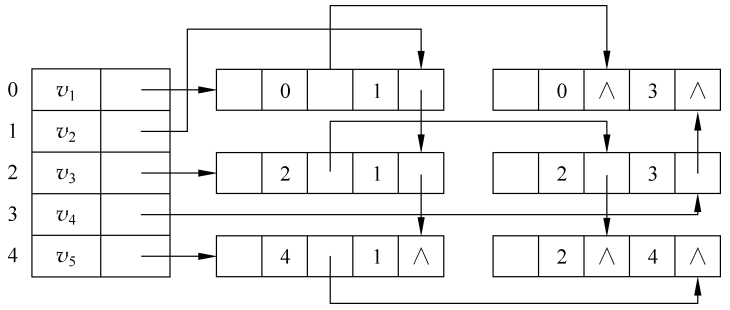


图 5.16　无向图 G_1 的邻接多重表

5.2　图的遍历算法

图的遍历指从图中的任一顶点出发，对图中的所有顶点访问一次且只访问一次。我国有 23 个省份、4 个直辖市、5 个自治区，以及 2 个特别行政区。34 个地区的风土人情各不相同。假如制订一份旅游计划，34 个地区全部游玩一遍而且只游玩一次，这就是遍历。图的遍历操作和树的遍历操作功能相似。图的遍历是图的一种基本操作，图的许多其他操作都是建立在遍历操作的基础之上的。

由于图结构本身的复杂性，图的遍历操作也较复杂，主要表现在以下 4 方面。

(1) 在图结构中，没有一个“自然”的首结点，图中任意一个顶点都可作为第一个被访问的结点。

(2) 在非连通图中，从一个顶点出发，只能访问它所在的连通分量上的所有顶点，因此，还需考虑如何选取下一个出发点以访问图中其余的连通分量。

(3) 在图结构中，如果有回路存在，那么一个顶点被访问之后，有可能沿回路又回到该顶点。

(4) 在图结构中，一个顶点可以和其他多个顶点相连，当这样的顶点访问过后，存在如何选取下一个要访问的顶点的问题。

图的遍历通常有深度优先搜索和广度优先搜索两种方式，下面分别介绍。


观看视频

5.2.1　深度优先搜索

深度优先搜索(Depth First Search,DFS)遍历类似于树的先根遍历,是树的先根遍历的推广。DFS算法最早是由John E. Hopcroft和他的学生Robert E. Tarjan一起提出来的,两位科学家凭借他们在数据结构与图论算法中的贡献共同获得了图灵奖。图灵奖被称为计算机领域的诺贝尔奖,获奖难度很高,但是数据结构领域的很多科学家都获得了图灵奖,因为数据结构领域的这些算法都非常的底层和经典,为后续很多算法和应用奠定了基础,甚至推动了学科领域的发展。

假设初始状态是图中的所有顶点未曾被访问,则深度优先搜索可从图中某个顶点 v 出发,访问此顶点,然后依次从 v 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 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为起始点,重复上述过程,直至图中所有顶点都被访问到为止。

以图5.17所示的无向图 G_5 为例,进行图的深度优先搜索。假设从顶点 v_1 出发进行搜索,在访问了顶点 v_1 之后,选择邻接点 v_2。因为 v_2 未曾访问,所以从 v_2 出发进行搜索,以此类推,接着从 v_4、v_8、v_5 出发进行搜索。在访问了 v_5 之后,由于 v_5 的邻接点都已被访问,因此搜索回到 v_8。因为同样的理由,搜索继续回到 v_4、v_2 直至 v_1,此时由于 v_1 的另一个邻接点未被访问,因此搜索又从 v_1 到 v_3,再继续进行下去,由此得到的顶点访问序列为

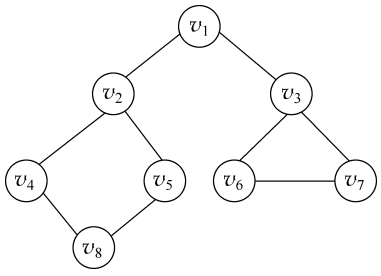


图5.17　一个无向图 G_5

$$v_1 \rightarrow v_2 \rightarrow v_4 \rightarrow v_8 \rightarrow v_5 \rightarrow v_3 \rightarrow v_6 \rightarrow v_7$$

显然,这是一个递归的过程。为了在遍历过程中便于区分顶点是否已被访问,需附设访问标志数组 visited[0:n-1],其初值为FALSE,一旦某个顶点被访问,则其相应的分量置为TRUE。

从图的某一点 v 出发,递归地进行深度优先遍历的过程如算法5.4所示。

算法5.4

```
void DFS(Graph G,int v)
{                                          /*从第 v 个顶点出发递归地深度优先遍历图 G*/
    visited[v] = TRUE;Visit(v);            /*访问第 v 个顶点*/
    for(w = FirstAdjVex(G,v);w; 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 算法*/
}
```

算法5.5和算法5.6给出了对以邻接表为存储结构的整幅图 G 进行深度优先遍历实现的C语言描述。

算法5.5

```
void DFSTraverseAL(ALGraph *G)
{                                          /*深度优先遍历以邻接表存储的图 G*/
   int i;
   for (i = 0;i < G->n;i++)
        visited[i] = FALSE;                /*标志向量初始化*/
```

```
    for (i=0;i<G->n;i++)
        if (!visited[i]) DFSAL(G,i);    /* v_i 未访问过,从 v_i 开始深度优先搜索 */
}                                       /* DFSTraverseAL */
```

算法 5.6

```
void DFSAL(ALGraph *G,int i)
{                                   /* 以 v_i 为出发点对邻接表存储的图 G 进行深度优先搜索 */
    EdgeNode *p;
    printf("visit vertex:V%c\n",G->adjlist[i].vertex);          /* 访问顶点 v_i */
    visited[i]=TRUE;                /* 标记 v_i 已访问 */
    p=G->adjlist[i].firstedge;      /* 取 v_i 边表的头指针 */
    while(p)                        /* 依次搜索 v_i 的邻接点 v_j,j=p->adjva */
    {if (!visited[p->adjvex])       /* 若 v_j 尚未访问,则以 v_j 为出发点向纵深搜索 */
       DFSAL(G,p->adjvex);
    p=p->next;                      /* 找 v_i 的下一个邻接点 */
    }
}                                   /* DFSAL */
```

分析上述算法，在遍历时，对图中每个顶点至多调用一次 DFS 函数，因为一旦某个顶点被标记成已被访问，就不再从它出发进行搜索。因此，遍历图的过程实质上是对每个顶点查找其邻接点的过程。其耗费的时间则取决于所采用的存储结构。当用二维数组表示邻接矩阵图的存储结构时，查找每个顶点的邻接点所需时间复杂度为 $O(n^2)$，其中 n 为图中顶点数。而当以邻接表作图的存储结构时，找邻接点所需时间复杂度为 $O(e)$，其中 e 为无向图中的边数或有向图中的弧数。由此，当以邻接表作存储结构时，深度优先搜索遍历图的时间复杂度为 $O(n+e)$。


观看视频

5.2.2 广度优先搜索

广度优先搜索（Breadth First Search，BFS）遍历类似于树的按层次遍历的过程。BFS 算法在如今看来并不复杂，但在其刚提出时被大众所接受的过程有一点曲折。它最早是康拉德教授于 1945 年在他的博士论文里面提出来的，但是这篇博士论文并没有第一时间被奥格斯堡大学发表，直到 1972 年，也就是又经过了 27 年，这篇论文手稿才被发表，才有了我们今天看到的 BFS 算法。

假设从图中某顶点 v 出发，在访问了 v 之后依次访问 v 的各未曾访问过的邻接点，然后分别从这些邻接点出发依次访问它们的邻接点，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，直至图中所有已被访问的顶点的邻接点都被访问到。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。换句话说，广度优先搜索遍历图的过程中以 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。

例如，对图 5.17 所示无向图 G_5 进行广度优先搜索遍历，首先访问 v_1 和 v_1 的邻接点 v_2 和 v_3，然后依次访问 v_2 的邻接点 v_4、v_5 及 v_3 的邻接点 v_6 和 v_7，最后访问 v_4 的邻接点 v_8。由于这些顶点的邻接点均已被访问，并且图中所有顶点都被访问，因此完成了图的遍历。得到的顶点访问序列为

$$v_1 \rightarrow v_2 \rightarrow v_3 \rightarrow v_4 \rightarrow v_5 \rightarrow v_6 \rightarrow v_7 \rightarrow v_8$$

与深度优先搜索类似，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,需附设队列以存储已被访问的路径长度为 1,2,…的顶点。

从图的某一顶点 v 出发,非递归地进行广度优先遍历的过程如算法 5.7 所示。

算法 5.7

```
Void BFSTraverse(Graph G, Status( * Visit)(int v))
{                                  /* 按广度优先非递归遍历图 G,使用辅助队列 Q 和访问标志数
                                      组 visited */
   for (v = 0;v < G.vexnum;++v)
       visited[v] = FALSE;
   InitQueue(Q);                               /* 初始化队列 Q */
   if (!visited[v])                            /* v 尚未被访问 */
   {  EnQueue(Q,v);                            /* v 入队列 */
      while (!QueueEmpty(Q))
      {  DeQueue(Q,u);                         /* 队头元素出队并置为 u */
         visited[u] = TRUE; visit(u);          /* 访问 u,并设置访问标志 */
         for(w = FirstAdjVex(G,u); w; w = NextAdjVex(G,u,w))
           if (!visited[w]) EnQueue(Q,w);      /* u 的尚未访问的邻接顶点 w 入队列 Q */
       }
    }
}                                              /* BFSTraverse */
```

算法 5.8 和算法 5.9 给出了对以邻接矩阵为存储结构的整幅图 G 进行广度优先遍历实现的 C 语言描述。

算法 5.8

```
void BFSTraverseAL(MGraph * G)
{                                              /* 广度优先遍历以邻接矩阵存储的图 G */
   int i;
   for (i = 0;i < G->n;i++)
       visited[i] = FALSE;                     /* 标志向量初始化 */
   for (i = 0;i < G->n;i++)
       if (!visited[i]) BFSM(G,i);             /* v_i 未访问过,从 v_i 开始 BFS 搜索 */
}                                              /* BFSTraverseAL */
```

算法 5.9

```
void BFSM(MGraph * G,int k)
{                                  /* 以 v_i 为出发点,对邻接矩阵存储的图 G 进行广度优先搜索 */
   int i,j;
   InitQueue(&Q);
   printf("visit vertex:V%c\n",G->vexs[k]);      /* 访问原点 v_k */
   visited[k] = TRUE;
   EnQueue(&Q,k);                      /* 原点 v_k 入队列 */
   while (!QueueEmpty(&Q))
   {  i = DeQueue(&Q);                 /* v_i 出队列 */
      for (j = 0;j < G->n;j++)         /* 依次搜索 v_i 的邻接点 v_j */
          if (G->edges[i][j] == 1 && !visited[j])     /* 若 v_j 未访问 */
          {  printf("visit vertex:V%c\n",G->vexs[j]); /* 访问 v_j */
             visited[j] = TRUE;
             EnQueue(&Q,j);           /* 访问过的 v_j 入队列 */
          }
   }
}                                      /* BFSM */
```

分析上述算法,每个顶点至多进一次队列。遍历图的过程实质是通过边或弧找邻接点的过程,因此广度优先搜索遍历图的时间复杂度和深度优先搜索遍历图的时间复杂度相同,

两者不同之处仅仅在于对顶点访问的顺序不同。


观看视频

5.2.3 深度优先搜索与广度优先搜索的应用

例 5.1 火力网：炮台的排放问题，图 5.18 表示了一个 4×4 的方形城市，其中黑色块是障碍物，白色块是路(空地)，黑色圆圈表示炮台安放的位置。布防规则是：炮台可排放在路上，但任意两个炮台若中间没有障碍物分隔就不能在同一行或同一列中，反之，合法。图 5.18 中前两种排放合法，后两种则不合法。

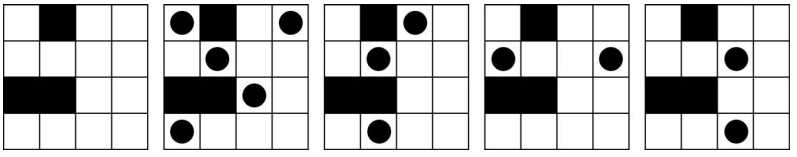


图 5.18 城市炮台的排放

输入文件包含一幅或多幅图的描述，0 表示输入结束。每幅图的描述都开始于一个整数 n($n \leqslant 4$)，表示城市大小 $n \times n$，接下来的 n 行逐行描述图的信息，“.”表示开放空间，“X”表示墙。请问每次输入一幅城市图之后，最多可以排放几个炮台。[ZOJ 1002]

输入示例

```
4
.X..
....
XX..
....
2
XX
.X
3
.X.
X.X
.X.
3
...
.XX
.XX
4
....
....
0
```

输出示例

```
5
1
5
2
4
```

解题思路如下。

由于地图的大小最大为 4×4，可将地图用一个 char 数组存起来，即 map[4][4]。如果 map[i][j]='X'则表示地图此处存放的为墙，map[i][j]='.'则表示此处存放的为空地，而 map[i][j]='o'则表示此处存放的为炮台。关键是炮台不能同时在水平和垂直线上，除非有墙作为间隔。定义 k 为位置，$k=0$ 即为地图左上方第一个格子(见图 5.19)。

依次往其中放炮台，需判断两个条件。

(1) 放的位置是否为空地。

(2) 同行同列不能有炮台，除非有墙间隔(见 canput 函数)。

如果到了 $k=n*n$，即终止条件时，看目前的最大炮台数是否大于 bestn 最优炮台数。

如此搜索，即可得到最好的结果。算法如下。

```
0  1  2  3
4  5  6  7
8  9  10 11
12 13 14 15
```

图 5.19 位置地图

算法 5.10

```
#include<stdio.h>
int n;                                  //城市的尺寸
char map[4][4];                         //城市的地图,最多是 4×4
int bestn;                              //最多放的炮台数

int canput(int row,int col)             //看炮台是否能够放置
{   int i;
    for(i=row-1;i>=0;i--)               //扫描行
    {   if(map[i][col]=='X')
        {   break;
        }
        if(map[i][col]=='o')
        {   return 0;
        }
    }
    for(i=col-1;i>=0;i--)               //扫描同一列
    {   if(map[row][i]=='X')
        {   break;
        }
        if(map[row][i]=='o')
        {   return 0;
        }
    }
    return 1;
}

void backtrack(int k,int current)       //current 为放的数目,k 为放置炮台的位置 0,1,2,3,…,n×n
{   int x,y;
    if(k>=n*n)                          //到达最后一个
    {   if(current>bestn)
        {   bestn=current;
        }
        return;
    }
    else
    {   x=k/n;                                  //计算 x 坐标
        y=k%n;                                  //计算 y 坐标
        if(map[x][y]=='.'&&canput(x,y))
         {   map[x][y]='o';                     //安放炮台
             backtrack(k+1,current+1);          //进入下一个坐标,数目加 1
             map[x][y]='.';                     //还原
         }
         backtrack(k+1,current);
    }
}
void initial()
{   int i,j;
    for(i=0;i<4;i++)
    {   for(j=0;j<4;j++)
        {   map[i][j]='.';
        }
    }
}
int main()
```

```
{   scanf("%d",&n);
    while(n)
    {   int i,j;
        bestn=0;
        initial();
        for(i=0;i<n;i++)
        {   for(j=0;j<n;j++)
            {   char ch;
                ch=getchar();
                if(ch=='\n')
                {   j--;
                    continue;
                }
                else
                {   map[i][j]=ch;
                }
            }
        }
        backtrack(0,0);                          //不要忘了初始化
        printf("%d\n",bestn);
        scanf("%d",&n);
    }
    return 0;
}
```

例 5.2 拼图游戏：有一个游戏，给出一幅图，该图由 $n\times n$ 个小正方形组成，每个小正方形又由 4 个三角形组成，且每个三角形上都有一个 0～9 的数字，要求用这 $n\times n$ 个小正方形拼成一幅图，该图的每个小正方形相邻的三角形中间的数是相同的，如图 5.20 所示。

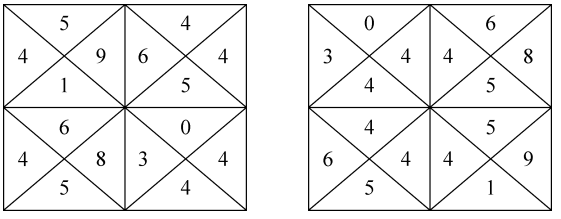


图 5.20 一个 2×2 的小正方形拼图前后的对比图

输入文件包含多组游戏情况，每组开始用一个整数 $n(0\leqslant n\leqslant 5)$ 表示游戏的规模，之后 $n\times n$ 行标识这些三角形的数字，每行包含 4 个数字，顺序是顶三角、右三角、底三角和左三角。0 表示输入结束。输出格式见输出样例，每个游戏之间输出一个空白行。[ZOJ 1008]

输入示例

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

输出示例

```
Game 1: Possible

Game 2: Impossible
```

```
2 2 2 2
3 3 3 3
4 4 4 4
0
```

解题思路如下。

本题属于 DFS +剪枝的题目，希望读者通过本题的学习，了解经典的剪枝思想。题目输入是一个个正方形的图，通过移动方块，使得每两个方块连接处的数字相同。

为了减少搜索时间，将相同类型的方块只保存一次，并且保存相同方块出现的次数。一旦一个方块不匹配了，那么相同方块都可以直接剪枝。

由于 $n \leqslant 5$，故用 element[25][4]，方块最多 25 块，每块有 4 个三角形，即顶三角、右三角、底三角和左三角，对应的坐标分别为 0、1、2、3。用 state 来保存每种类型方块出现的次数。backtrack 这个回溯函数，从 ipos = 0 处进行搜索，直到 ipos = $n * n$，这样就到了终止条件。在每个 ipos 位置，分别放入各状态的方块，用完一个状态的方块，那个状态的方块个数减 1；用 result 记录当前放入的方块的类型，并与正上方的方块、正左方的方块进行比对，看可否放入，不能放入返回 0，否则返回 1。此外，方块可移动但不能旋转。

算法 5.11

```
#include<stdio.h>
int n;                                  //n 表示游戏的大小,n 小于或等于 5
int element[25][4];                     //存放每个格子
int state[25];                          //每个状态
int result[25];                         //存放的结果
int q;                                  //状态的个数

void initial()                          //初始化
{   int i,j;
    for(i=0;i<25;i++)
    {   for(j=0;j<4;j++)
        {   element[i][j]=0;
        }
        state[i]=0;
        result[i]=0;
    }
    q=0;
}

int backtrack(int ipos)                 //搜索到 ipos 位
{   int i;
    if(ipos==n*n)                       //成功放完 n*n 个 square
    {   return 1;
    }
    else
    {  for(i=0;i<q;i++)                 //在 ipos 位把每个状态放一次
       {   if(state[i]==0)              //该种类型方块已经用完
            { continue;
            }
            else
            {   if(ipos>=n)             //判断能否符合要求: 不是最上边的
                {   if(element[result[ipos-n]][2]!=element[i][0])
                    {   continue;
                    }
```

```
                }
                if(ipos % n!= 0)                    //不是最左边的
                {   if(element[result[ipos - 1]][1]!= element[i][3])
                    {   continue;
                    }
                }
                state[i] -- ;
                result[ipos] = i;
                if(backtrack(ipos + 1) == 1)        //DFS 的精髓
                    return 1;
                state[i]++;                         //恢复该方块的类型数 1 个,便于下一次搜索
            }
        }
    }
    return 0;
}
int main()
{   int i,j,index;
    index = 0;
    int top,right,bottom,left;
    scanf(" % d",&n);
    while(n)
    {   initial();
        index++;
        for(i = 0;i < n * n;i++)                    //判断是否有同一种类型,是则进行 state[j]++
                                                    //同种类型加 1
        {   scanf(" % d % d % d % d",&top,&right,&bottom,&left);
            for(j = 0;j < q;j++)
            {  if(element[j][0] == top&&element[j][1] == right&&
                    element[j][2] == bottom&&element[j][3] == left)
                {   state[j]++;
                    break;
                }
            }
           if(j == q)                               //没有同种类型,将新的类型存入 iSquare 中
            {   element[q][0] = top;
                element[q][1] = right;
                element[q][2] = bottom;
                element[q][3] = left;
                state[q] = 1;                       //该种类型的数目现在是一种
                q++;
            }
        }
        if(index > 1)
        {   printf("\n");                           //陷阱!就是每个结果之间要有空白行
        }
        printf("Game % d: ",index);
        if(backtrack(0))
        {   printf("Possible\n");
        }
        else
        {   printf("Impossible\n");
        }
            scanf(" % d",&n);
```

```
    }
    return 0;
}
```

题目要求样例的解之间要用空行分隔，但最后一个样例的解之后就不应有多余的空行。

例 5.3　数独游戏：数独游戏是一种非常流行的填数游戏。要求用 1～9 的数字去填充如图 5.21 所示的 9×9 表格，具体要求如下。

（1）每行的 9 个格子中 1～9 各出现一次。

（2）每列的 9 个格子中 1～9 各出现一次。

（3）用粗线隔开的 3×3 的小块的 9 个格子中 1～9 个数字也各出现一次。

输入数据首先是给出测试用例数，然后是表相关的 9 行数据，每行 9 个十进制数字，0 表示该位置是空的。对每个测试用例按照输入数据的格式输出解，并在空白处填入符合规则的数字。如果解不唯一，只要输出其中一种即可。[POJ 2676]

1		3				5		9
		2	1		9	4		
			7		4			
3			5		2			6
	6						5	
7			8		3			4
			4		1			
		9	2		5	8		
8		4				1		7

图 5.21　数独表

输入示例	输出示例
1	143628579
103000509	572139468
002109400	986754231
000704000	391542786
300502006	468917352
060000050	725863914
700803004	237481695
000401000	619275843
009205800	854396127
804000107	

解题思路如下。

本题求解需要使用回溯法、DFS 算法，并使用标记法剪枝。下面说说剪枝。

（1）如果有一个格子，9 个数都不能填进去，剪枝。如果只能填一个，不用说，直接填。

（2）如果有这么一行，有一个数放到 9 个格子里面的任一个都不可，剪枝。如果只有一个格子可填该数，直接填。

（3）如果有这么一列，有一个数放到 9 个格子里面的任一个都不可，剪枝。如果只有一个格子可填该数，直接填。

（4）如果有这么一个九宫格，有一个数放到 9 个格子里面的任一个都不可，剪枝。如果只有一个格子可填该数，直接填。

参考算法如下。

算法 5.12

```
#include<stdio.h>
```

```
#include<stdlib.h>
bool rUsed[9][10],cUsed[9][10],sUsed[9][10];
//用于标记某行、某列、某个3×3小方格上哪些数字已经被使用过了
int pos[100];                                   //还没有填充数字的方格位置
int nullNum;                                    //空白的个数,即需填数字个数
int table[9][9];
bool DFS_SUDO;

void print()                                    //输出结果表
{   for(int i=0;i<9;i++)
{   for(int j=0;j<9;j++)
        printf("%d",table[i][j]);
        printf("\n");
}
}

void DFS(int n)
{   if (n>=nullNum)                             //已填写数字个数大于或等于空白数
{   DFS_SUDO=true;                              //递归结束
        print();                                //调用print函数输出结果
        return;
}
    int r=pos[n]/9;                             //在第r行
    int c=pos[n]%9;                             //在第c列
    int k=(r/3)*3+(c/3);                        //在第k个小方格
for(int i=1; i<=9 && !DFS_SUDO;i++)
{   if (cUsed[c][i]) continue;                  //判别列中是否用过该数字
        if (rUsed[r][i]) continue;              //判别行中是否用过该数字
        if (sUsed[k][i]) continue;              //判别方格中是否用过该数字
        cUsed[c][i]=rUsed[r][i]=sUsed[k][i]=true;       //置已用数字标志
        table[r][c]=i;                          //当前位置填上数字i
        DFS(n+1);                               //递归找下一个要填数的位置
        table[r][c]=0;                          //如果DFS失败就回溯,并还原原来的值
        cUsed[c][i]=rUsed[r][i]=sUsed[k][i]=false;      //还原标志位的值
    }
    return;
}

int main()
{   FILE *fp;
    int testCase;
    char line[10];
    fp=fopen("test.txt","r");                   //以文件形式打开测试文件
    //testCase = fgetc(fp)-'0';
    fscanf(fp,"%d",&testCase);                  //输入测试的数据组数
    fgetc(fp);                                  //输入一行结束符
    while(testCase--)
    {   nullNum=0;
            for(int i=0;i<9;i++)                //初始化标志位
                for(int j=0;j<10;j++)
                    rUsed[i][j]=cUsed[i][j]=sUsed[i][j]=false;
            for(int i=0;i<9;i++)
            {   fgets(line,11,fp);              //读入一行
                for(int j=0;j<9;j++)
```

```
                {   table[i][j] = line[j] - '0';
                    if(table[i][j]){
                        rUsed[i][table[i][j]] = true;       //第 i 行用过这个数
                        cUsed[j][table[i][j]] = true;       //第 j 列用过这个数
                        int k = (i/3) * 3 + (j/3);          //第 k 个 3×3 方格
                        sUsed[k][table[i][j]] = true;       //第 k 个方格用过这个数
                    }
                    else
                        pos[nullNum++] = 9 * i + j;       //使用数组,记录没有填数的小方格的位置
                }
            }
            DFS_SUDO = false;
            DFS(0);
        }
        fclose(fp);
    return 0;
    }
```

使用二维数组直接把已经存在的数字和没有填数字的位置用数组记录下来，这样在DFS时就非常方便，所以，选择合适和便于处理的数据结构有事半功倍的效果。建议读者再重新去思考和理解一下第2章中介绍的迷宫问题算法。

5.3　图的连通性

利用图的遍历算法可以判定一幅图的连通性。本节将重点讨论无向图的连通性、有向图的连通性、由图得到其生成树或生成森林以及连通图中是否有关结点等几个有关图的连通性的问题。


观看视频

5.3.1　无向图的连通性

在对无向图进行遍历时，对于连通图，仅需从图中任一顶点出发，进行深度优先搜索或广度优先搜索，便可访问到图中所有顶点。对于非连通图，则需从多个顶点出发进行搜索，而在每一次从一个新的起始点出发进行搜索的过程中得到的顶点访问序列恰为其各连通分量中的顶点集。例如，图5.5(a)是一幅非连通图G_3，对其邻接表(见图5.22)进行深度优先搜索遍历，并调用两次深度优先搜索(即分别从顶点A和C出发)，得到的顶点访问序列分别为$A\ B\ F\ E$和$C\ D$，这两个顶点集分别加上遍历时所依附于这些顶点的边，便构成了非连通图G_3的两个连通分量，如图5.5(b)所示。

因此，要想判定一幅无向图是否为连通图，或有几个连通分量，就可设一个计数变量count，初始时取值为0，在算法5.5的第二个for循环中，每调用一次深度优先搜索，就给count增1。这样，当整个算法结束时，依据count的值，就可确定图的连通性了。

5.3.2　有向图的连通性

深度优先搜索是求有向图的强连通分量的一个有效方法。假设以十字链表作有向图的存储结构，则求强连通分量的步骤如下。

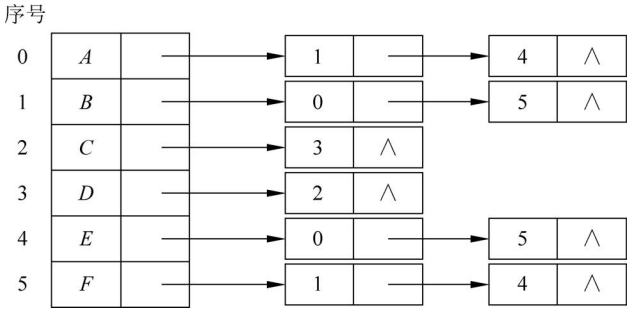


图 5.22 G_3 的邻接表

(1) 在有向图 G 上，从某个顶点出发沿以该顶点为尾的弧进行深度优先搜索遍历，并按其所有邻接点的搜索都完成(即退出 DFS 函数)的顺序将顶点排列起来。此时需对 5.2.1 节中的算法进行如下两点修改：①在进入 DFSTraverseAL 函数时首先进行计数变量的初始化，即在入口处加上 count = 0 的语句；②在退出函数之前将完成搜索的顶点号记录在另一个辅助数组 finished[vexnum]中，即在 DFSAL 函数结束之前加上 finished[++ count]= v 的语句。

(2) 在有向图 G 上，从最后完成搜索的顶点(即 finished[vexnum − 1]中的顶点)出发，沿着以该顶点为头的弧进行逆向的深度搜索遍历，若此次遍历不能访问到有向图中的所有顶点，则从余下的顶点中最后完成搜索的那个顶点出发，继续进行逆向的深度优先搜索遍历，以此类推，直至有向图中所有顶点都被访问到为止。此时调用 DFSTraverseAL 函数需进行如下修改：函数中第二条循环语句的边界条件应改为 v 从 finished[vexnum − 1]至 finished[0]。

由此，每次调用 DFSAL 函数进行逆向深度优先遍历所访问到的顶点集便是有向图 G 中一个强连通分量的顶点集。

例如图 5.14(a)所示的有向图，假设从顶点 v_1 出发进行深度优先搜索遍历，得到 finished 数组中的顶点号为(1,3,2,0)，则再从顶点 v_1 出发进行逆向的深度优先搜索遍历，得到两个顶点集$\{v_1,v_3,v_4\}$和$\{v_2\}$，这就是该有向图的两个强连通分量的顶点集。

上述求强连通分量的第(2)步，其实质如下。

(1) 构造一幅有向图 G_r，设 $G=(V,\{A\})$，则 $G_r=(V_r,\{A_r\})$对于所有$<v_i,v_j>\in A$，必有$<v_j,v_i>\in A_r$。即 G_r 中拥有和 G 方向相反的弧。

(2) 在有向图 G_r 上，从顶点 finished[vexnum − 1]出发进行深度优先遍历。可以证明，在 G_r 上所得深度优先生成森林中每棵树的顶点集即为 G 的强连通分量的顶点集。

显然，利用遍历求强连通分量的时间复杂度亦和遍历相同。

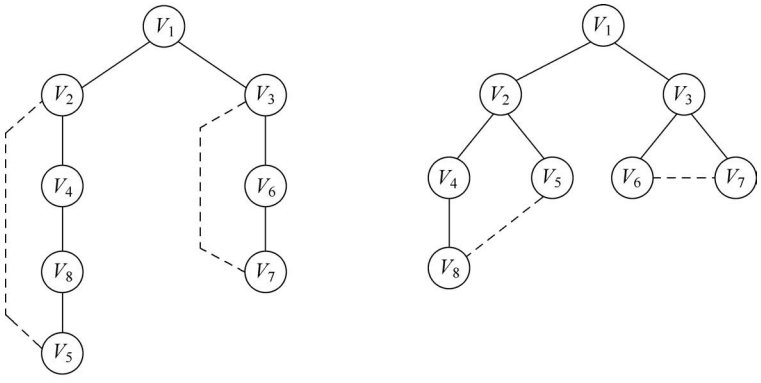

观看视频

5.3.3 生成树和生成森林

本节将给出通过对图的遍历，得到图的生成树或生成森林的算法。

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，必定将 $E(G)$分成两个集合 $T(G)$和 $B(G)$，其中 $T(G)$是遍历图过程中历经的边的集合；$B(G)$是剩余边的集合。显然，$T(G)$和图 G 中所有顶点一起构成连通图 G 的极小连通子图。按照

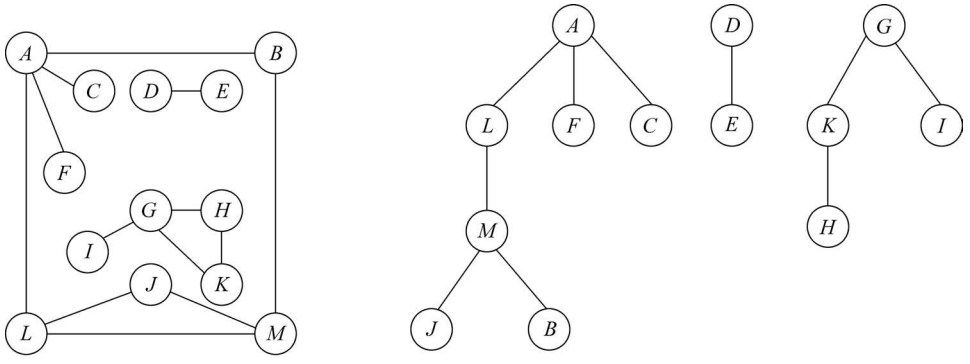
5.1.2 节的定义，它是连通图的一棵生成树，并且由深度优先搜索得到的为深度优先生成树；由广度优先搜索得到的为广度优先生成树。例如，图 5.23(a)和图 5.23(b)所示分别为连通图 G_5 的深度优先生成树和广度优先生成树，图中虚线为集合 $B(G)$中的边，实线为集合 $T(G)$中的边。



(a) G_5的深度优先生成树　　(b) G_5的广度优先生成树

图 5.23　由图 5.17 无向图 G_5 得到的生成树

对于非连通图，通过这样的遍历，将得到的是生成森林。例如，图 5.24(b)所示为图 5.24(a)的深度优先生成森林，它由 3 棵深度优先生成树组成。



(a) 一幅非连通图无向图G_6　　(b) G_6的深度优先生成森林

图 5.24　非连通图 G_6 及其生成森林

假设以孩子兄弟链表作生成森林的存储结构，则算法 5.13 生成非连通图的深度优先生成森林，其中 DFSTree 函数如算法 5.14 所示。显然，算法 5.13 的时间复杂度和遍历相同。

算法 5.13

```
void DFSForest(Graph G, CSTree *T)
{                                        /*建立无向图 G 的深度优先生成森林的孩子兄弟链表 T*/
  T = NULL;
  for (v = 0;v < G.vexnum;++v)
    if (!visited[v] = FALSE;
  for(v = 0;v < G.vexnum;++v)
    if (!visited[v])                     /*顶点 v 为新的生成树的根结点*/
      { p = (CSTree)malloc(sizeof(CSNode));   /*分配根结点*/
```

```
        p={GetVex(G,v).NULL,NULL};          /*给根结点赋值*/
         if (!T)
               (*T)=p;                      /*T是第一棵生成树的根*/
         else q->nextsibling=p;             /*前一棵的根的兄弟是其他生成树的根*/
         q=p;                               /*q指示当前生成树的根*/
         DFSTree(G,v,&p);                   /*建立以p为根的生成树*/
        }
 }
```

算法 5.14

```
void DFSTree(Graph G,int v,CSTree *T)
{                              /*从第v个顶点出发深度优先遍历图G,建立以*T为根的生成树*/
    visited[v]=TRUE;
    first=TRUE;
    for(w=FirstAdjVex(G,v); w; w=NextAdjVex(G,v,w))
    if(!visited[w])
    {   p=(CSTree)malloc(sizeof)CSNode));   /*分配孩子结点*/
        *p={GetVex(G,w),NULL,NULL};
        if (first)              /*w是v的第一个未被访问的邻接顶点,作为根的左孩子结点*/
        {   T->lchild=p;
            first=FALSE;
        }
        else                  /*w是v的其他未被访问的邻接顶点,作为上一邻接顶点的右兄弟*/
        {   q->nextsibling=p;
           }
        q=p;
        DFSTree(G,w,&q);          /*从第w个顶点出发深度优先遍历图G,建立生成子树*q*/
        }
}
```

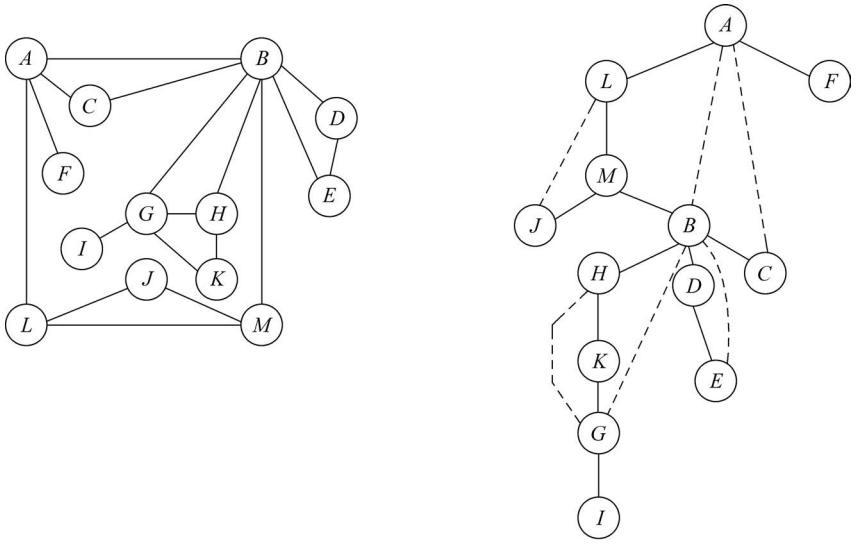

观看视频

5.3.4 关节点和重连通分量

假若在删去顶点 v 以及和 v 相关联的各边之后，将图的一个连通分量分割成两个或两个以上的连通分量，则称顶点 v 为该图的一个关节点(Articulation Point)。一幅没有关节点的连通图称为重连通图(Biconnected Graph)。在重连通图上，任意一对顶点之间至少存在两条路径，因此在删去某个顶点以及依附于该顶点的各边时也不破坏图的连通性。若在连通图上至少删去 k 个顶点才能破坏图的连通性，则称此图的连通度为 k。关节点和重连通图在实际中有较多应用。显然，一幅表示通信网络的图的连通度越高，其系统越可靠，无论是哪一个站点出现故障或遭到外界破坏，都不影响系统的正常工作；又如，一个航空网若是重连通的，则当某条航线因天气等某种原因关闭时，旅客仍可从其他航线绕道而行；再如，若将大规模的集成电路的关键线路设计成重连通图，则在某些元件失效的情况下，整个片子的功能不受影响，反之，在战争中，若要摧毁敌方的运输线，仅需破坏其运输网中的关节点即可。

例如，图 5.25(a)中图 G_7 是连通图，但不是重连通图。图中有 3 个关节点 A、B 和 G。若删去顶点 B 以及所有依附顶点 B 的边，G_7 就被分割成 3 个连通分量{A、C、F、L、M、J}、{G、H、I、K}和{D、E}。类似地，若删去顶点 A 或 G 以及所依附于它们的边，则 G_7 被分割成两个连通分量，由此，关节点亦称为割点。

利用深度优先搜索便可求得图的关节点，并由此可判别图是否是重连通的。



(a) 一幅连通图无向图G_7　　(b) G_7的深度优先生成树

图 5.25　无向连通图 G_7 及其生成树

图 5.25(b)所示为从顶点 A 出发深度优先生成树，图中实线表示树边，虚线表示回边(即不在生成树上的边)。对树中任一顶点 v 而言，其孩子结点为在它之后搜索到的邻接点，而其双亲结点和由回边连接的祖先结点是在它之前搜索到的邻接点。由深度优先生成树可得出两类关节点的特性。

(1) 若生成树的根有两棵或两棵以上的子树，则此根顶点必为关节点。因为图中不存在连接不同子树中顶点的边，因此，若删去根顶点，生成树便变成生成森林，如图 5.25(b)中所示的顶点 A。

(2) 若生成树中某个非叶结点 v，其某棵子树的根和子树中的其他结点均没有指向 v 的祖先的回边，则 v 为关节点。因为，若删去 v，则其子树和图的其他部分被分割开来，如图 5.25(b)所示的顶点 B 和 G。

若对图 Graph =(V,{Edge})重新定义遍历时的访问函数 visited，并引入一个新的函数 low，则由一次深度优先遍历便可求得连通图中存在的所有关节点。

定义 visited[v]为深度优先搜索遍历连通图时访问顶点 v 的次序号；定义：

$$\text{low(v)} = \text{Min}\left\{\text{visited[v]},\text{low[w]},\text{visited[k]} \;\middle|\; \begin{array}{l}\text{w 是 v 在 DFS 生成树上的孩子结点；}\\ \text{k 是 v 在 DFS 生成树上由回边联结的祖先结点；}\\ \text{(v,w)}\in\text{Edge；}\\ \text{(v,k)}\in\text{Edge。}\end{array}\right\}$$

若对于某个顶点 v，存在孩子结点 w 且 low[w]≥visited[v]，则该顶点 v 必为关节点。因为当 w 是 v 的孩子结点时，low[w]≥visited[v]，表明 w 及其子孙均无指向 v 的祖先的回边。

由定义可知，visited[v]值即为 v 在深度优先生成树的前序序列的序号，只需将 DFS 函数中头两条语句改为 visited[v0]=++ count(在 DFSTraverse 中设初值 count = 1)即可；low[v]可由后序遍历深度优先生成树求得，而 v 在后序序列中的次序和遍历时退出 DFS 函数的次序相同，由此修改深度优先搜索遍历的算法便可得到求关节点的算法(如算法 5.15 和算法 5.16 所示)。

算法 5.15

```
void FindArticul(ALGraph G)
{ /* 连通图 G 以邻接表作存储结构,查找并输出 G 上的全部关节点 */
  count = 1;                                  /* 全局变量 count 用于对访问计数 */
  visited[0] = 1;                             /* 设定邻接表上 0 号顶点为生成树的根 */
  for(i = 1;i < G.vexnum;++i)                 /* 其余顶点尚未访问 */
    visited[i] = 0;
  p = G.adjlist[0].first;
  v = p->adjvex;
  DFSArticul(g,v);                            /* 从顶点 v 出发深度优先查找关节点 */
  if(count < G.vexnum)                        /* 生成树的根至少有两棵子树 */
    {printf(0,G.adjlist[0].vertex);           /* 根是关节点,输出 */
     while(p->next)
        { p = p->next;
          v = p->adjvex;
          if(visited[v] == 0) DFSArticul(g,v);
        }
    }
} /* FindArticul */
```

算法 5.16

```
void DFSArticul(ALGraph G,int v0)
/* 从顶点 v0 出发深度优先遍历图 G,查找并输出关节点 */
{ visited[v0]= min = ++count;                 /* v0 是第 count 个访问的顶点 */
  for(p = G.adjlist[v0].firstedge; p; p = p->next;)        /* 对 v0 的每个邻接点检查 */
     { w = p->adjvex;                         /* w 为 v0 的邻接点 */
       if(visited[w] == 0)                    /* 若 w 未曾访问,则 w 为 v0 的孩子 */
         { DFSArticul(G,w);                   /* 返回前求得 low[w] */
           if(low[w]< min)min = low[w];
           if(low[w]>= visited[v0]) printf(v0,G.adjlist[v0].vertex);     /* 输出关节点 */
         }
       else if(visited[w]< min) min = visited[w];/* w 已访问,w 是 v0 在生成树上的祖先 */
     }
   low[v0] = min;
 }
```

例如,图 G_7 中各顶点计算所得 visited 和 low 的函数值如表 5.1 所示。

表 5.1 visited 与 low 的函数值表

i	0	1	2	3	4	5	6	7	8	9	10	11	12
G. adjlist[i]. vertex	A	B	C	D	E	F	G	H	I	J	K	L	M
visited[i]	1	5	12	10	11	13	8	6	9	4	7	2	3
low[i]	1	1	1	5	5	1	5	5	8	2	5	1	1
求得 low 值的顺序	13	9	8	7	6	12	3	5	2	1	4	11	10

表 5.1 中 J 是第一个求得 low 值的顶点,由于存在回边(J,L),因此 low[J]= Min{visited[J], visited[L]}= 2。顺便提一句,上述算法中将指向双亲的树边也看成回边,由于不影响关节点的判别,因此,为使算法简明,在算法中没有区别之。

由于上述算法的过程就是一个遍历的过程,因此,求关节点的时间复杂度仍为 $O(n+e)$。


观看视频

5.3.5 有向图的强连通分量

求强连通分量有 3 种算法——Kosaraju、Tarjan、Gabow。本节重点介绍高效的 Tarjan

算法。Tarjan 算法的应用非常广泛，几乎任何和图的遍历有关的问题都可以套用 Tarjan 算法的思想（如求割点、桥、块、强连通分量等）。提出此算法的普林斯顿大学的 Robert E. Tarjan 教授也是 1986 年的图灵奖获得者。

在有向图 G 中，如果两个顶点间至少存在一条路径，称两个顶点强连通（Strongly Connected）。如果有向图 G 的每两个顶点都强连通，称 G 是一幅强连通图。非强连通图有向图的极大强连通子图，称为强连通分量（Strongly Connected Component，SCC）。求强连通分量的意义是：由于强连通分量内部的结点性质相同，因此可以将一个强连通分量内的结点缩成一个点，即消除了环，这样，原图就变成了一幅有向无环图（Directed Acyclic Graph，DAG）。

图 5.26 中 G_8 的子图{1，2，3，4}为一个强连通分量，因为顶点 1、2、3、4 两两可达。{5}、{6}也分别是两个强连通分量。

直接根据定义，用双向遍历取交集的方法求强连通分量，时间复杂度为 $O(N^2+M)$。更好的方法是 Tarjan 算法或 Kosaraju 算法，两者的时间复杂度都是 $O(N+M)$。

Tarjan 算法是基于对图深度优先搜索的算法，每个强连通分量为搜索树中的一棵子树。搜索时，把当前搜索树中未处理的结点加入一个堆栈，回溯时可以判断栈顶到栈中的结点是否为一个强连通分量。

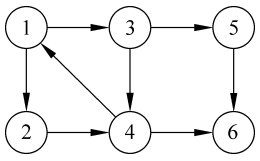


图 5.26 有向图 G_8

定义 DFN(u)为结点 u 搜索的次序编号（时间戳），Low(u)为 u 或 u 的子树能够追溯到的最早的栈中结点的次序号。由定义可以得出：

```
Low(u) = Min
{   DFN(u),
    Low(v),(u,v)为树枝边,u 为 v 的双亲结点
    DFN(v),(u,v)为指向栈中结点的后向边(非横叉边)
}
```

当 DFN(u)＝Low(u)时，以 u 为根的搜索子树上所有结点是一个强连通分量。伪代码如下。

算法 5.17

```
tarjan(u)
{   DFN[u] = Low[u] = ++Index                  //为结点 u 设定次序编号和 Low 初值
    Stack.push(u)                              //将结点 u 压入栈中
    for each (u, v) in E                       //枚举每条边
        if (v is not visted)                   //如果结点 v 未被访问过
            tarjan(v)                          //继续向下找
            Low[u] = min(Low[u], Low[v])
        else if (v in S)                       //如果结点 v 还在栈内
            Low[u] = min(Low[u], DFN[v])
    if (DFN[u] == Low[u])                      //如果结点 u 是强连通分量的根
        repeat
            v = S.pop                          //将 v 退栈,为该强连通分量中一个顶点
            print v
        until (u== v)
}
```

算法演示：从结点 1 开始进行深度优先搜索，把遍历到的结点加入栈中。搜索到结点 u

=6 时，DFN[6]=Low[6]，找到一个强连通分量。退栈到 u=v 为止，{6}为一个强连通分量（见图 5.27(a)）。

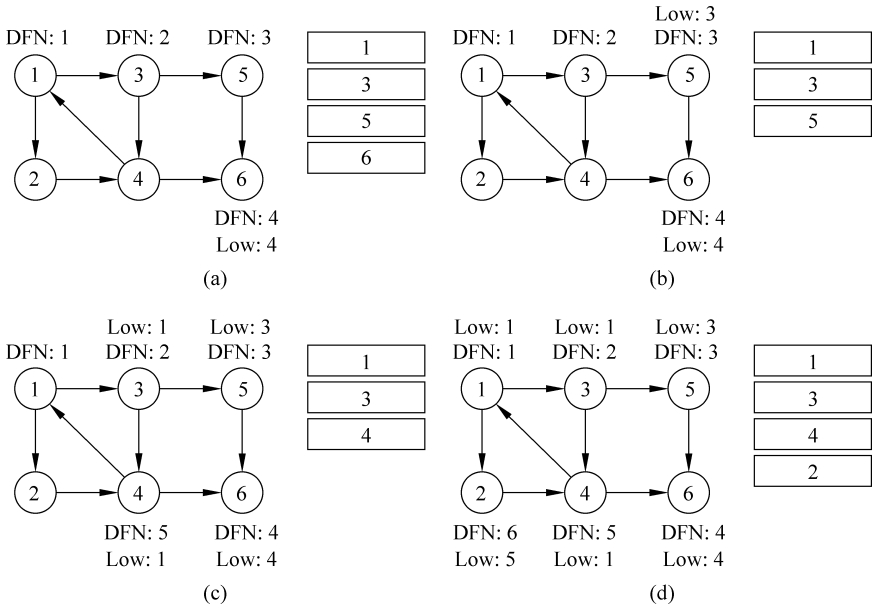


图 5.27　针对图 5.25 有向图 G_8 的 Tarjan 算法演示

返回结点 5，发现 DFN[5]=Low[5]，退栈后{5}为一个强连通分量（见图 5.27(b)）。

返回结点 3，继续搜索到结点 4，把 4 加入堆栈。发现结点 4 向结点 1 有后向边，结点 1 还在栈中，所以 Low[4]=1。结点 6 已经出栈，(4,6)是横叉边，返回 3，(3,4)为树枝边，所以 Low[3]=Low[4]=1（见图 5.26(c)）。

继续回到结点 1，最后访问结点 2。访问边(2,4)，4 还在栈中，所以 Low[2]=DFN[4]=5。返回 1 后，发现 DFN[1]=Low[1]，把栈中结点全部取出，组成一个连通分量{1,3,4,2}（见图 5.27(d)）。

至此，算法结束。通过该算法求出了图中全部的 3 个强连通分量，分别为{1,3,4,2}、{5}、{6}。运行 Tarjan 算法的过程中，每个顶点都被访问了一次，且只进出了一次堆栈，每条边也只被访问了一次，所以该算法的时间复杂度为 $O(N+M)$。算法模板如下。

算法 5.18

```
int top;                                          //用作栈顶的指针
int Stack[MAX];                                   //维护的一个栈
bool instack[MAX];                                //instack[i]为真表示 i 在栈中
int DFN[MAX],Low[MAX];
int Belong[MAX];                                  //Belong[i] = a; 表示 i 这个点属于第 a 个连通分量
int Bcnt,Dindex;                                  //Bcnt 用来记录连通分量的个数,Dindex 表示到达某个点的时间
void tarjan(int u)
{   int v;
    DFN[u] = Low[u] = ++Dindex;                   //这里要注意 Dindex 是初始化为 0,这里就不能
                                                  //Dindex++; 不然第一个点的 DFN 和 Low 就为 0
    Stack[++top] = u;
    instack[u] = true;
    for (edge *e = V[u]; e; e = e->next)          //对所有可达边进行搜索
```

```
    {   v = e->t;
        if (!DFN[v])                          //用来更新 Low[u]
        {   tarjan(v);
            if (Low[v] < Low[u])
                Low[u] = Low[v];
        }
        else if (instack[v] && DFN[v] < Low[u])
            Low[u] = DFN[v];
    }
    if (DFN[u] == Low[u])                     //已找完一个强连通
    {   Bcnt ++;                              //强连通个数加 1
        do
        {   v = Stack[top --];
            instack[v] = false;
            Belong[v] = Bcnt;
        }
        while (u != v);                       //一直到 v = u 都属于第 Bcnt 个强连通分量
    }
}
void solve()
{   int i;
    Stop = Bcnt = Dindex = 0;
    memset(DFN,0,sizeof(DFN));
    for (i = 1; i <= N; i ++)                 //一定要对所有点应用 Tarjan 算法才能求出所有
                                              //点的强连通分量
        if (!DFN[i])
            tarjan(i);
}
```

该 Tarjan 算法与求无向图的双连通分量(割点、桥)的 Tarjan 算法有着很深的联系。学习该 Tarjan 算法,有助于深入理解求双连通分量的 Tarjan 算法,两者可以类比、组合理解。

例 5.4 道路建设(Road Construction)。题目描述:给你一幅无向图,然后问你至少需要添加几条边,可以使整幅图变成边双连通分量,也就是说任意两点至少有两条路可以互相连通。


观看视频

第一行输入两个整数 n、r,n(3≤n≤1000)表示岛上的旅游景点数,r(2≤r≤1000)是道路数,旅游景点被标识为 1～n。接着的 r 行是表示景点的两个整数 v 和 w。游客可以沿着道路的任何一个方向旅行,每对景点之间至多一条直接的道路。在当前道路设置中,任何的两个景点间都可以旅行。输出需要添加的最少道路数。[POJ 3352]

输入示例 1

```
10 12
1 2
1 3
1 4
2 5
2 6
5 6
3 7
3 8
```

输入示例 2

```
3 3
1 2
2 3
1 3
```

输出示例 1

```
2
```

输出示例 2

```
0
```

```
7 8
4 9
4 10
9 10
```

问题分析如下。

对于属于同一个边双连通分量的任意点至少有两条通路是可以互相可达的，因此可以将一个边双连通分量缩成一个点。考虑不在边双连通分量中的点，通过缩点后可形成一棵树。对于一幅树形的无向图，需要添加（度为 1 的点的个数+1）/2 条边使得图成为双连通的。这样问题就变成缩点之后求图中度为 1 的点的个数了。

这个题目的条件给得很强，任意两个点之间不会有重边，因此可以直接经过 Tarjan 算法的 low 值进行边双连通分量的划分，最后求出度为 1 的点数即可。如果有重边，则不同的 low 值是可能属于同一个边双连通分量的，这时就要通过将图中的桥去掉然后求解边双连通分量。

Tarjan 算法在求解强连通分量时，通过引入深度优先搜索过程中对一个点访问的顺序 dfsNum（也就是在访问该点之前已经访问的点的个数）和一个点可以到达的最小的 dfsNum 的 low 数组，当遇到一个顶点的 dfsNum 值等于 low 值时，那么该点就是一个强连通分量的根。因为在深度优先搜索的过程中已经将点入栈，因此只需要将栈中的元素出栈直到遇到根，那么这些点就组成一个强连通分量。

对于边双连通分量，还需要先了解一些概念。

（1）边连通度：使一幅子图不连通所需要删除的最小的边数就是该图的边连通度。

（2）桥（割边）：当删除一条边就使得图不连通的那条边称为桥或者是割边。

（3）边双连通分量：边连通度大于或等于 2 的子图称为边双连通分量。

理解了这些概念之后再来看看 Tarjan 算法是如何求解边双连通分量的，不过在此之前还得先说说 Tarjan 算法是怎样求桥的。

引入 dfsNum 表示一个点在深度优先搜索过程中所被访问的时间，然后就是 low 数组表示该点最小的可以到达的 dfsNum。分析一下桥的特点，删除一条边之后，如果深度优先搜索过程中的子树没有任何一个点可以到达双亲结点及双亲结点以上的结点，那么这个时候子树就被封死了，这条边就是桥。有了这个性质，也就是说当深度优先搜索过程中遇到一条树边 a→b，并且此时 low[b]> dfsNum[a]，那么 a→b 就是一座桥。把所有的桥去掉之后那些独立的分量就是不同的边双连通分量，此时就可以按照需要灵活地求出边双连通分量了。参考代码如下。

算法 5.19

```
#include<iostream>
#include<cstring>
#include<cstdlib>
#include<cstdio>
#include<vector>
using namespace std;
const int Max=1010;
int top[Max],edge[Max][Max];                    //memset(top,0,sizeof(top));
```

```
int dfsNum[Max],dfsnum;                          //memset(dfsNum,0,sizeof(dfsNum)),dfsNum = 1;
int low[Max],degree[Max], cc[Max];
int ccCnt, ans;
bool exist[Max][Max];
void tarjan(int a,int fa)
{   dfsNum[a] = low[a] = ++dfsnum;
    for(int i = 0;i < top[a];i++)
    {   if(edge[a][i]!= fa)
        {   if(dfsNum[edge[a][i]] == 0)
            {   tarjan(edge[a][i],a);
                if(low[a]> low[edge[a][i]])
                    low[a] = low[edge[a][i]];
                if(dfsNum[a] < low[edge[a][i]])
                    exist[a][edge[a][i]] = exist[edge[a][i]][a] = true;
            }
            else
                  if(low[a]> dfsNum[edge[a][i]])
                    low[a] = dfsNum[edge[a][i]];
        }
    }
}

void dfs(int fa, int u)
{   cc[u] = ccCnt;
    for(int i = 0; i < top[u]; i++)
    { int v = edge[u][i];
        if(v != fa && !exist[u][v] && !cc[v])
              dfs(u, v);
    }
}

int solve(int n)
{   int i,j;
    int a,b;
    memset(cc, 0, sizeof(cc));
    ccCnt = 1;
    for(i = 1; i <= n; i++)
    {   if(!cc[i])
        {   dfs(-1, i);
            ccCnt++;
        }
    }
    for(i = 1;i <= n;i++)
    {   a = i;
        for(j = 0;j < top[i];j++)
        {   b = edge[a][j];
            if(cc[a] != cc[b])
            {   degree[cc[a]]++;
                degree[cc[b]]++;
            }
        }
    }
    int leaves = 0;
    for(i = 1;i < ccCnt;i++)
```

```
        if(degree[i] == 2)
            leaves++;
    return (leaves + 1)/2;
}

int main()
{   int n,m;
    int i,a,b;
    while(scanf("%d %d",&n,&m)!= EOF)
    {   memset(top,0,sizeof(top));
        memset(degree,0,sizeof(degree));
        for(i = 0;i < m;i++)
        {   scanf("%d %d",&a,&b);
            edge[a][top[a]++] = b;
            edge[b][top[b]++] = a;
        }
        memset(dfsNum,0,sizeof(dfsNum));
        dfsnum = 0;
        memset(exist, false, sizeof(exist));
        tarjan(1, -1);
        ans = solve(n);
        printf("%d\n",ans);
    }
    return 0;
}
```

Robert Tarjan 还发明了求双连通分量的 Tarjan 算法，以及求最近公共祖先的离线 Tarjan 算法，建议基础好的读者去设计实现这些算法[POJ 1236，1470]。

5.4 有向无环图及其应用


观看视频

5.4.1 有向无环图的概念

一幅无环的有向图称作有向无环图(Directed Acycline Graph，DAG)。DAG 是一类较有向树更一般的特殊有向图，图 5.28 给出了有向树、有向无环图和有向图的例子。

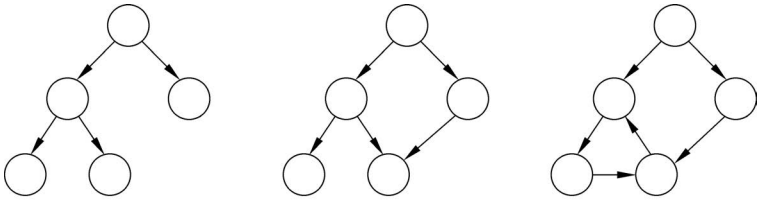


图 5.28 有向树、有向无环图和有向图示意

有向无环图是描述含有公共子式的表达式的有效工具。例如下述表达式：

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

可以用第 6 章讨论的二叉树来表示，如图 5.29 所示。仔细观察该表达式，可发现有一些相同的子表达式，如(c + d)和(c + d) * e 等，在二叉树中，它们也重复出现。若利用有向无环

图，则可实现对相同子式的共享，从而节省存储空间。例如，图 5.30 所示为表示同一表达式的有向无环图。

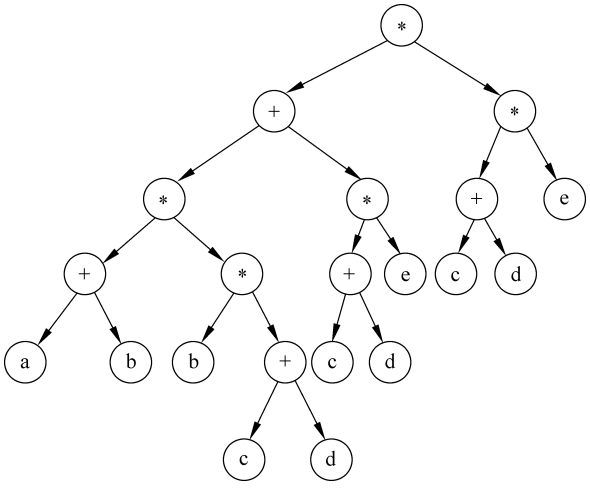


图 5.29　用二叉树描述表达式

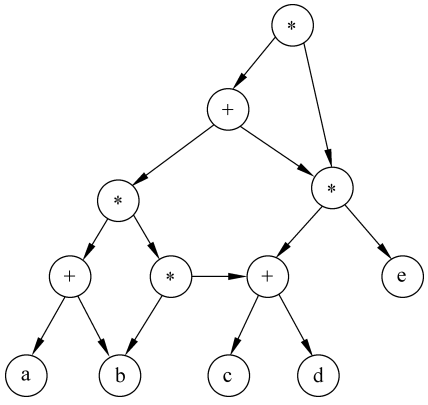


图 5.30　描述表达式的有向无环图

检查一幅有向图是否存在环要比无向图复杂。对于无向图来说，若在深度优先搜索遍历过程中遇到回边(即指向已访问过的顶点的边)，则必定存在环；而对于有向图来说，这条回边有可能是指向深度优先生成森林中另一棵生成树上顶点的弧。但是，如果从有向图上某个顶点 v 出发的遍历，在 dfs(v)结束之前出现一条从顶点 u 到顶点 v 的回边，由于 u 在生成树上是 v 的子孙，因此有向图必定存在包含顶点 v 和 u 的环。

有向无环图是描述一项工程或系统的进行过程的有效工具。除最简单的情况之外，几乎所有的工程(Project)都可分为若干称作活动(Activity)的子工程，而这些子工程之间通常受着一定条件的约束，如其中某些子工程的开始必须在另一些子工程完成之后。对整个工程和系统，人们关心的是两方面的问题：一是工程能否顺利进行；二是估算整个工程完成所必需的最短时间。5.4.2 节和 5.4.3 节将详细介绍这样两个问题是如何通过对有向图进行拓扑排序和关键路径操作来解决的。

5.4.2　AOV 网与拓扑排序


观看视频

1. AOV(Activity On Vertex)网

一个工程或某种流程可以分解为若干小工程或阶段，这些小工程或阶段就称为活动。若以图中的顶点来表示活动，有向边表示活动之间的优先关系，则这样的活动在顶点上的有向图称为 AOV 网。在 AOV 网中，若从顶点 i 到顶点 j 之间存在一条有向路径，称顶点 i 是顶点 j 的前驱，或者称顶点 j 是顶点 i 的后继。若$<i,j>$是图中的弧，则称顶点 i 是顶点 j 的直接前驱，顶点 j 是顶点 i 的直接后驱。

AOV 网中的弧表示了活动之间存在的制约关系。例如，计算机专业的学生必须完成一系列规定的基础课和专业课才能毕业。学生按照怎样的顺序来学习这些课程呢？这个问题可以被看作一个大的工程，其活动就是学习每一门课程。这些课程的名称与相应代号如表 5.2 所示。

表 5.2 计算机专业的课程设置及其关系

课程代号	课程名	先行课程代号	课程代号	课程名	先行课程代号
C_1	程序设计导论	无	C_8	算法分析	C_3
C_2	数值分析	C_1,C_{13}	C_9	高级语言	C_3,C_4
C_3	数据结构	C_1,C_{13}	C_{10}	编译系统	C_9
C_4	汇编语言	C_{12}	C_{11}	操作系统	C_{10}
C_5	自动机理论	C_{13}	C_{12}	解析几何	无
C_6	人工智能	C_3	C_{13}	微积分	C_{12}
C_7	机器原理	C_{13}			

表中，C_1、C_{12} 是独立于其他课程的基础课，而有的课却需要有先行课程，例如，学完程序设计导论和数值分析后才能学数据结构等，先行条件规定了课程之间的优先关系。这种优先关系可以用图 5.31 所示的有向图来表示。其中，顶点表示课程，有向边表示前提条件。若课程 i 为课程 j 的先行课，则必然存在有向边$<i,j>$。在安排学习顺序时，必须保证在学习某门课之前已经学习了其先行课程。

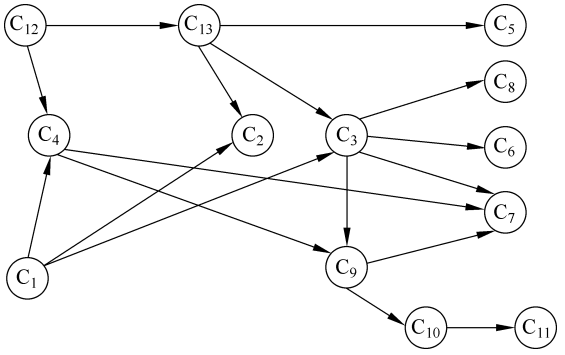


图 5.31 一个 AOV 网实例

类似的 AOV 网的例子还有很多，如大家熟悉的计算机程序，任何一个可执行程序也可以划分为若干程序段（或若干语句），由这些程序段组成的流程图也是一个 AOV 网。

2. 拓扑排序

首先介绍一下离散数学中的偏序集合与全序集合两个概念。

若集合 A 中的二元关系 R 是自反的、非对称的和传递的，则 R 是 A 上的偏序关系。集合 A 与关系 R 一起称为一个偏序集合。

若 R 是集合 A 上的一个偏序关系，如果对每个 $a,b\in A$ 必有 aRb 或 bRa，则 R 是 A 上的全序关系。集合 A 与关系 R 一起称为一个全序集合。

偏序关系经常出现在日常生活中。例如，若把 A 看成一项大的工程必须完成的一批活动，则 aRb 意味着活动 a 必须在活动 b 之前完成。例如，对于前面提到的计算机专业的学生必修的基础课与专业课，由于课程之间的先后依赖关系，某些课程必须在其他课程以前讲授，这里的 aRb 就意味着课程 a 必须在课程 b 之前学完。

AOV 网所代表的一项工程中活动的集合显然是一个偏序集合。为了保证该项工程得以顺利完成，必须保证 AOV 网中不出现回路；否则，意味着某项活动应以自身作为能否开

展的先决条件，这是不合理的。

测试 AOV 网是否具有回路（即是否是一幅有向无环图）的方法，就是在 AOV 网的偏序集合下构造一个线性序列，该线性序列具有以下性质。

（1）在 AOV 网中，若顶点 i 优先于顶点 j，则在线性序列中顶点 i 仍然优先于顶点 j。

（2）对于网中原来没有优先关系的顶点 i 与顶点 j，如图 5.31 中所示的 C_1 与 C_{13}，在线性序列中也建立了一个先后关系，或者顶点 i 优先于顶点 j，或者顶点 j 优先于顶点 i。

满足这样性质的线性序列称为拓扑有序序列。构造拓扑序列的过程称为拓扑排序。也可以说拓扑排序就是由某个集合上的一个偏序得到该集合上的一个全序的操作。

若某个 AOV 网中所有顶点都在它的拓扑序列中，则说明该 AOV 网不会存在回路，这时的拓扑序列集合是 AOV 网中所有活动的一个全序集合。以图 5.31 中的 AOV 网为例，可以得到不止一个拓扑序列，C_1、C_{12}、C_4、C_{13}、C_5、C_2、C_3、C_9、C_7、C_{10}、C_{11}、C_6、C_8 就是其中之一。显然，对于任何一项工程中各活动的安排，必须按拓扑有序序列中的顺序进行才是可行的。

3. 拓扑排序算法

对 AOV 网进行拓扑排序的方法和步骤如下。

（1）从 AOV 网中选择一个没有前驱的顶点（该顶点的入度为 0）并且输出它。

（2）从网中删去该顶点，并且删去从该顶点发出的全部有向边。

（3）重复上述两步，直到剩余的网中不再存在没有前驱的顶点为止。

这样操作的结果有两种：一种是网中全部顶点都被输出，这说明网中不存在有向回路；另一种就是网中顶点未被全部输出，剩余的顶点均有前驱顶点，这说明网中存在有向回路。

图 5.32 给出了在一个 AOV 网上实施上述步骤的例子。

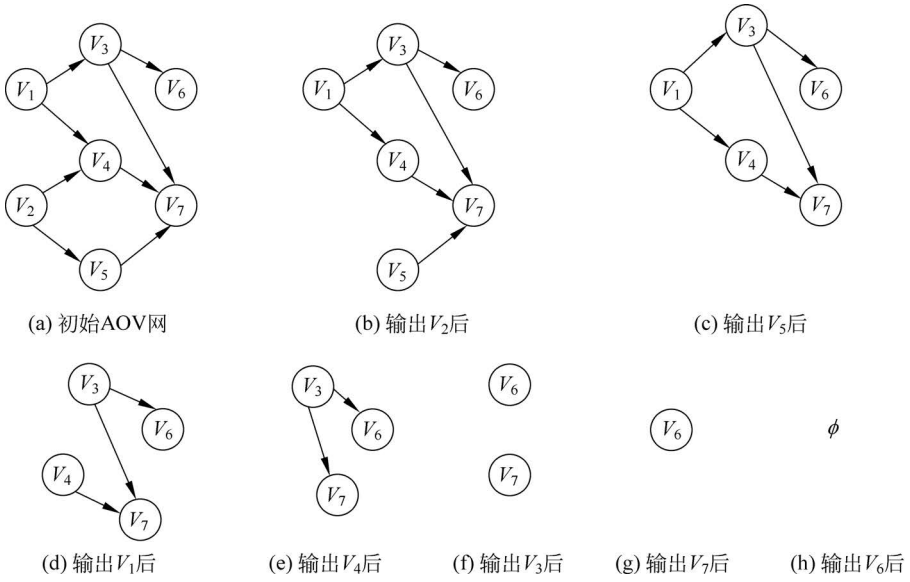


图 5.32　求一拓扑序列的过程

这样得到一个拓扑序列：V_2，V_5，V_1，V_4，V_3，V_7，V_6。

为了实现上述算法，对 AOV 网采用邻接表存储方式，并且在邻接表中的顶点结点中增加一个记录顶点入度的数据域，即顶点结构设为 count，vertex，firstedge。其中，vertex、firstedge 的含义如前所述；count 为记录顶点入度的数据域。边结点的结构同 5.2.2 节所述。图 5.32(a)中的 AOV 网的邻接表如图 5.33 所示。

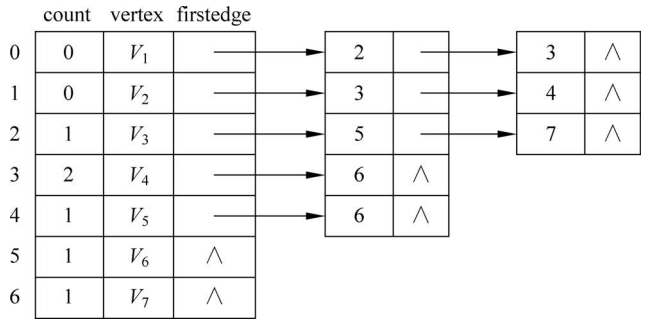


图 5.33　图 5.32(a)所示的 AOV 网的邻接表

顶点表结点结构的描述改为

```
typedef struct vnode{                          /* 顶点表结点 */
    int count                                  /* 存放顶点入度 */
    VertexType vertex;                         /* 顶点域 */
    EdgeNode * firstedge;                      /* 边表头指针 */
    }VertexNode;
```

当然也可以不增设入度域，而另外设一个一维数组来存放每个结点的入度。

算法中可设置一个堆栈，凡是网中入度为 0 的顶点都将其入栈。为此，拓扑排序的算法步骤如下。

(1) 将没有前驱的顶点(count 域为 0)压入栈。

(2) 从栈中退出栈顶元素输出，并把该顶点引出的所有有向边删去，即把它的各邻接顶点的入度减 1。

(3) 将新的入度为 0 的顶点再入堆栈。

(4) 重复步骤(2)～(4)，直到栈为空为止。此时或者是已经输出全部顶点，或者剩下的顶点中没有入度为 0 的顶点。

从上面的步骤可以看出，栈在这里只是起到一个保存当前入度为零的顶点，并使之处理有序的作用。这种有序可以是后进先出，也可以是先进先出，故此也可用队列来辅助实现。在下面给出用 C 语言描述的拓扑排序的算法实现中，采用栈来存放当前未处理过的入度为 0 的结点，但并不需要额外增设栈的空间，而是设一个栈顶位置的指针将当前所有未处理过的入度为 0 的结点联结起来，形成一个链式栈。

算法 5.20

```
void Topo_Sort(AlGraph * G)
{                              /* 对以代入度的邻接链表为存储结构的图 G,输出其一种拓扑序列 */
    int top = -1;                              /* 栈顶指针初始化 */
    for (i = 0; i < n; i++)                    /* 依次将入度为 0 的顶点压入链式栈 */
    {   if (G->adjlist[i].count == 0)
        {   G->adjlist[i].count = top;
            top = i;
```

```
        }
    }
    for (i = 0;i < n;i++)
    {   if (top = - 1)
        {   printf("The network has a cycle");
            return;
        }
        j = top;
        top = G - > adjlist[top].count;          /* 从栈中退出一个顶点并输出 */
        printf(" % c",G - > adjlist[j].vertex);
        ptr = G - > adjlist[j].firstedge;
        while (ptr!= null)
        {   k = ptr - > adjvex;
            G - > adjlist[k].count -- ;          /* 当前输出顶点邻接点的入度减 1 */
            if(G - > adjlist[k].count == 0)      /* 新的入度为 0 的顶点进栈 */
            {   G - > adjlist[k].count = top;
                top = k;
            }
            ptr = ptr - > next;                  /* 找到下一个邻接点 */
        }
    }
}
```

对一个具有 n 个顶点、e 条边的网来说，整个算法的时间复杂度为 $O(e+n)$。


观看视频

5.4.3 AOE 网与关键路径

1. AOE(Activity On Edge)网

若在带权的有向图中，以顶点表示事件，以有向边表示活动，边上的权值表示活动的开销(如该活动持续的时间)，则此带权的有向图称为 AOE 网。

如果用 AOE 网来表示一项工程，那么，仅仅考虑各子工程之间的优先关系还不够，更多的是关心整个工程完成的最短时间是多少；哪些活动的延期将会影响整个工程的进度，而加速这些活动是否会提高整个工程的效率。因此，通常在 AOE 网中列出完成预定工程计划所需要进行的活动，每个活动计划完成的时间，要发生哪些事件以及这些事件与活动之间的关系，从而可以确定该项工程是否可行，估算工程完成的时间以及确定哪些活动是影响工程进度的关键。

AOE 网具有以下两个性质。

(1) 只有在某顶点所代表的事件发生后，从该顶点出发的各有向边所代表的活动才能开始。

(2) 只有在进入一某顶点的各有向边所代表的活动都已经结束时，该顶点所代表的事件才能发生。

图 5.34 给出了一个具有 15 个活动、11 个事件的假想工程的 AOE 网。$v_1,v_2,\cdots,v_{11}$ 分别表示一个事件；$<v_1,v_2>,<v_1,v_3>,\cdots,<v_{10},v_{11}>$分别表示一个活动；用 $a_1,a_2,\cdots,a_{15}$ 代表这些活动。其中，v_1 称为源点，是整个工程的开始点，其入度为 0；v_{11} 为终点，是

整个工程的结束点，其出度为 0。

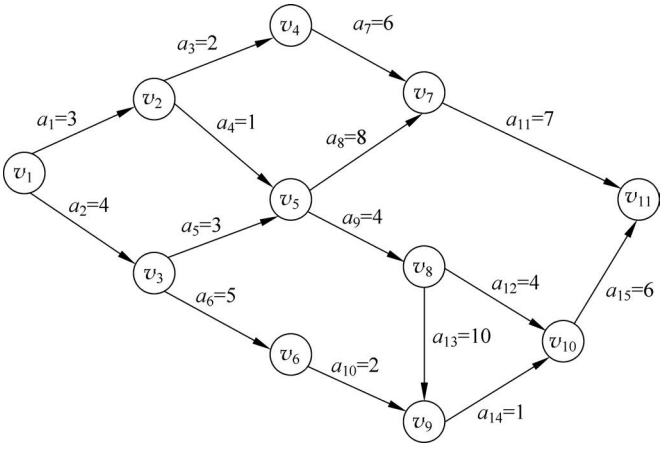


图 5.34 一个 AOE 网实例

对于 AOE 网，可采用与 AOV 网一样的邻接表存储方式。其中，邻接表中边结点的域为该边的权值，即该有向边代表的活动所持续的时间。

2. 关键路径

由于 AOE 网中的某些活动能够同时进行，因此完成整个工程所必须花费的时间应该为源点到终点的最大路径长度（这里的路径长度指该路径上的各活动所需时间之和）。具有最大路径长度的路径称为关键路径。关键路径上的活动称为关键活动。关键路径长度是整个工程所需的最短工期。这就是说，要缩短整个工期，必须加快关键活动的进度。

利用 AOE 网进行工程管理时需要解决的主要问题如下。

（1）计算完成整个工程的最短路径。

（2）确定关键路径，以找出哪些活动是影响工程进度的关键。

3. 关键路径的确定

为了在 AOE 网中找出关键路径，需要定义几个参量，并且说明其计算方法。

1）事件的最早发生时间 ve[k]

ve[k]指从源点到顶点的最大路径长度代表的时间。这个时间决定了所有从顶点发出的有向边所代表的活动能够开工的最早时间。根据 AOE 网的性质，只有进入 v_k 所有活动 $<v_j,v_k>$都结束时，v_k 代表的事件才能发生；而活动$<v_j,v_k>$的最早结束时间为 ve[j]+dut($<v_j,v_k>$)。所以计算 v_k 发生的最早时间的方法如下：

$$\begin{cases} \text{ve}[l]=0 \\ \text{ve}[k]=\text{Max}\{\text{ve}[j]+\text{dut}(<v_j,v_k>)\},<v_j,v_k>\in p[k] \end{cases} \tag{5.1}$$

其中，$p[k]$表示所有到达 v_k 的有向边的集合；dut($<v_j,v_k>$)为有向边$<v_j,v_k>$上的权值。


观看视频

2）事件的最迟发生时间 vl[k]

vl[k]指在不推迟整个工期的前提下，事件 v_k 允许的最晚发生时间。设有向边$<v_k,v_j>$

代表从 v_k 出发的活动，为了不拖延整个工期，v_k 发生的最迟时间必须保证不推迟从事件 v_k 出发的所有活动$<v_k, v_j>$的终点 v_j 的最迟时间 vl[j]。vl[k]的计算方法如下：

$$\begin{cases} \text{vl}[n] = \text{ve}[n] \\ \text{vl}[k] = \text{Min}\{\text{vl}[j] - \text{dut}(<v_k, v_j>)\}, <v_k, v_j> \in s[k] \end{cases} \tag{5.2}$$

其中，$s[k]$为所有从 v_k 发出的有向边的集合。

3) 活动 a_i 的最早开始时间 $e[i]$

若活动 a_i 由弧$<v_k, v_j>$表示，根据 AOE 网的性质，只有事件 v_k 发生了，活动 a_i 才能开始。也就是说，活动 a_i 的最早开始时间应等于事件 v_k 的最早发生时间。因此，有

$$e[i] = \text{ve}[k] \tag{5.3}$$

4) 活动 a_i 的最晚开始时间 $l[i]$

活动 a_i 的最晚开始时间指在不推迟整个工程完成日期的前提下，必须开始的最晚时间。若由弧$<v_k, v_j>$表示，则 a_i 的最晚开始时间要保证事件 v_j 的最迟发生时间不拖后。因此，应该有

$$l[i] = \text{vl}[j] - \text{dut}(<v_k, v_j>) \tag{5.4}$$

根据每个活动的最早开始时间 $e[i]$和最晚开始时间 $l[i]$就可判定该活动是否为关键活动，也就是那些 $l[i]=e[i]$的活动就是关键活动，而那些 $l[i]>e[i]$的活动则不是关键活动，$l[i]-e[i]$的值为活动的时间余量。关键活动确定之后，关键活动所在的路径就是关键路径。

以图 5.35 所示的 AOE 网为例，求出上述参量，确定该网的关键活动和关键路径。

首先，按照式(5.1)求事件的最早发生时间 ve[k]。

ve(1) = 0
ve(2) = 3
ve(3) = 4
ve(4) = ve(2) + 2 = 5
ve(5) = max{ve(2) + 1, ve(3) + 3} = 7
ve(6) = ve(3) + 5 = 9
ve(7) = max{ve(4) + 6, ve(5) + 8} = 15
ve(8) = ve(5) + 4 = 11
ve(9) = max{ve(8) + 10, ve(6) + 2} = 21
ve(10) = max{ve(8) + 4, ve(9) + 1} = 22
ve(11) = max{ve(7) + 7, ve(10) + 6} = 28

其次，按照式(5.2)求事件的最迟发生时间 vl[k]。

vl(11) = ve(11) = 28
vl(10) = vl(11) − 6 = 22
vl(9) = vl(10) − 1 = 21
vl(8) = min{vl(10) − 4, vl(9) − 10} = 11
vl(7) = vl(11) − 7 = 21
vl(6) = vl(9) − 2 = 19
vl(5) = min{vl(7) − 8, vl(8) − 4} = 7
vl(4) = vl(7) − 6 = 15
vl(3) = min{vl(5) − 3, vl(6) − 5} = 4
vl(2) = min{vl(4) − 2, vl(5) − 1} = 6
vl(1) = min{vl(2) − 3, vl(3) − 4} = 0

再按照式(5.3)和式(5.4)求活动 a_i 的最早开始时间 $e[i]$和最晚开始时间 $l[i]$。

a_1	$e(1) = \text{ve}(1) = 0$	$l(1) = \text{vl}(2) - 3 = 3$
a_2	$e(2) = \text{ve}(1) = 0$	$l(2) = \text{vl}(3) - 4 = 0$
a_3	$e(3) = \text{ve}(2) = 3$	$l(3) = \text{vl}(4) - 2 = 13$

a_4	$e(4)=\mathrm{ve}(2)=3$	$l(4)=\mathrm{vl}(5)-1=6$
a_5	$e(5)=\mathrm{ve}(3)=4$	$l(5)=\mathrm{vl}(5)-3=4$
a_6	$e(6)=\mathrm{ve}(3)=4$	$l(6)=\mathrm{vl}(6)-5=14$
a_7	$e(7)=\mathrm{ve}(4)=5$	$l(7)=\mathrm{vl}(7)-6=15$
a_8	$e(8)=\mathrm{ve}(5)=7$	$l(8)=\mathrm{vl}(7)-8=13$
a_9	$e(9)=\mathrm{ve}(5)=7$	$l(9)=\mathrm{vl}(8)-4=7$
a_{10}	$e(10)=\mathrm{ve}(6)=9$	$l(10)=\mathrm{vl}(9)-2=19$
a_{11}	$e(11)=\mathrm{ve}(7)=15$	$l(11)=\mathrm{vl}(11)-7=21$
a_{12}	$e(12)=\mathrm{ve}(8)=11$	$l(12)=\mathrm{vl}(10)-4=18$
a_{13}	$e(13)=\mathrm{ve}(8)=11$	$l(13)=\mathrm{vl}(9)-10=11$
a_{14}	$e(14)=\mathrm{ve}(9)=21$	$l(14)=\mathrm{vl}(10)-1=21$
a_{15}	$e(15)=\mathrm{ve}(10)=22$	$l(15)=\mathrm{vl}(11)-6=22$

最后，比较 $e[i]$和 $l[i]$的值可判断出 a_2，a_5，a_9，a_{13}，a_{14}，a_{15} 是关键活动，关键路径如图 5.35 所示。

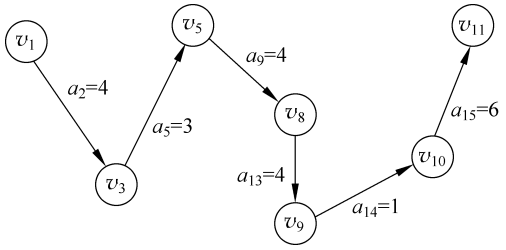


图 5.35 一个 AOE 网实例

由上述方法得到求关键路径的算法步骤如下。

(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] ($1\leqslant i\leqslant n-1$)。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]($n-2\geqslant i\geqslant 2$)。

(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。

由该步骤得到的算法如算法 5.21 和算法 5.22 所示。在算法 5.21 中，Stack 为栈的存储类型；引用的函数 FindInDegree(G,indegree)用来求图 G 中各顶点的入度，并将所求的入度存放于一维数组 indegree 中。

算法 5.21

```
int topologicalOrder(ALGraph G,Stack T)
{                    /*有向网 G 采用邻接表存储结构,求各顶点事件的最早发生时间 ve(全局变量)*/
   /*T 为拓扑序列顶点栈,S 为零入度顶点栈*/
   /*若 G 无回路,则用栈 T 返回 G 的一个拓扑序列,且函数值为 OK,否则为 ERROR*/
   FindInDegree(G, indegree);                  /*对各顶点求入度 indegree[0..vernum-1]*/
   InitStack(S);                               /*建立入度顶点栈 S*/
   count = 0; ve[0..G.vexnum-1] = 0;           /*初始化 ve[ ]*/
   for (i=0; i<G.vexnum; i++)                  /*将初始时入度为 0 的顶点入栈*/
   {   if (indegree[i]==0) push(S,i); }
   while (!StackEmpty(S)) {
```

```
        Pop(S,j); Push(T,j); ++count;          /* j 号顶点入 T 栈并计数 */
        for (p = G.adjlist[j].firstedge; p; p = p->next)
        {   k = p->adjvex;                     /* 对 j 号顶点的每个邻接点的入度减 1 */
            if(--indegree[k] == 0) Push(S,k);          /* 若入度减为 0, 则入栈 */
            if (ve[j] + *(p->info)> ve[k])
                ve[k] = ve[j] + *(p->info);
        }
      }
if (count < G. vexnum) return 0;               /* 该有向网有回路返回 0,否则返回 1 */
else return 1;
}                                              /* TopologicalOrder */
```

算法 5.22

```
int Criticalpath(ALGraph G)
{                                              /* G 为有向网,输出 G 的各项关键活动 */
      InitStack(T);                            /* 建立用于产生拓扑逆序的栈 T */
      if (!TopologicalOrder(G,T)) return 0;    /* 该有向网有回路返回 0 */
      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的 vl 值 */
        for (Pop(T,j), p = G.adjlist[j].firstedge; p; p = p->next)
          {   k = p->adjvex; dut = *(p->info);
              if (vl[k]-dut < vl[j]) vl[j] = vl[k] - dut;
          }
      for (j = 0; j < G.vexnum; ++j)           /* 求 e、l 和关键活动 */
          for (p = G.adjlist[j].firstedge; p; p = p->next)
          {   k = p->adjvex; dut = *(p->indo);
              e = ve[j]; l = vl[k] - dut;
              tag = (e == l) ? '*':' ';
              printf(j,k,dut,e,l,tag);         /* 输出关键活动 */
          }
    return 1;                                  /* 求出关键活动后返回 1 */
}                                              /* Criticalpath */
```

5.5 最短路径算法

观看视频

最短路径问题是图的又一个比较典型的应用问题。例如,某一地区的一个公路网,给定了该网内的 n 个城市以及这些城市之间的相通公路的距离,能否找到城市 A 到城市 B 之间一条距离最近的通路呢? 如果将城市用点表示,城市间的公路用边表示,公路的长度作为边的权值,那么,这个问题就可归结为在网图中,求点 A 到点 B 的所有路径中,边的权值之和最短的那一条路径。这条路径就是两点之间的最短路径,并称路径上的第一个顶点为源点(Sourse),最后一个顶点为终点(Destination)。在非网图中,最短路径指两点之间经历的边数最少的路径。

输入一幅赋权图:与每条边(v_i, v_j)相联系的是穿越该弧的代价(或称为值)$c_{i,j}$,一条路径 $v_1, v_2, \cdots, v_n$,的值是$\sum_{i=1}^{n-1} c_{i,i+1}$,叫作赋权路径长度(Weighted Path Length)。而无权路径长度(Unweighted Path Length)只是路径上的边数,即 $n-1$。

单源最短路径问题:给定一幅带权图 $G=(V, E)$和一个特定顶点 s 作为输入,找到 s 到G 中每个其他顶点的最短带权路径。

例如图 5.36(a)中，从 v_1 到 v_6 的最短带权路径长度为 6，它是从 v_1 到 v_4 到 v_7 再到 v_6 的路径。这两个顶点间的最短无权路径长度为 2。图 5.36(b)给出了一条权值为负数的边，从 v_5 到 v_4 的路径长度为 1，但通过循环 v_5, v_4, v_2, v_5, v_4 存在一条最短路径，其值为 −5，其实这条路径仍然不是最短的，因为循环可以进行多次，因此，这两个顶点间的最短路径问题是不确定的。类似地，v_1 到 v_6 的最短路径也是不确定的，因为它可以进入同样的循环，这个循环叫负值环(Negative Cost Cycle)；当它出现在图中时，最短路径问题就是不确定的。有带负值的边未必就是坏事，但它的出现似乎使问题增加了难度。

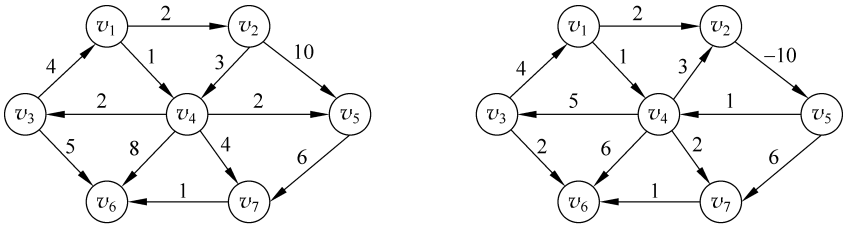


图 5.36　带权有向图和带负值权的有向图

本节重点介绍单源最短路径问题的相关算法。首先，考虑无权最短路径问题，并指出如何以 $O(|E|+|V|)$时间解决它。其次，假设边无负值，如何求解带权最短路径问题，期望在使用合理数据结构实现时的运行时间为 $O(|E| \cdot \log_2|V|)$。如果图有负边，介绍一个时间界为 $O(|E| \cdot |V|)$的简单解法。

5.5.1　无权最短路径

显然无权图可以视为权值都为 1 的带权图的特殊情形，如可将图 5.36(a)视为一幅权值均为 1 的无权图 G。使用某个顶点 s 作为输入参数，要找出从 s 到所有其他顶点的最短路径。假设选择 s 为 v_3，则 s 到 v_3 的最短路径为 0，下一步可以通过 v_3 找到路径长度为 1 的顶点 v_1 和 v_6，再通过 v_1 和 v_6 找出路径长度为 2 的顶点 v_2 和 v_4，最后通过 v_2、v_4 找出其余顶点的路径长度均为 3。显然，这个方法就是 BFS，处理过程类似于树的层次遍历，其时间复杂度为 $O(|E|+|V|)$。下面简要说明算法的实现。

对于每个顶点，关注顶点是否被处理(未处理为 F，处理过为 T，初始为 F)，s 到此顶点的路径长 dv(s 初始为 0，其他为 INFINITY)。在任意时刻，只存在两种类型的未知顶点，一些顶点的 dv = currDist，另一些顶点的 dv = currDist + 1。一种抽象是保留两个盒子，1 号盒子装有 dv = currDist 的那些未知顶点，而 2 号盒子装有 dv = currDist + 1 的那些顶点。找出一个合适顶点的测试可以用查找 1 号盒内的任意顶点 v 代替。在更新 v 的临界顶点 w 后，将 w 放入 2 号盒中。

可以使用一个队列进一步简化上述模型。迭代开始时，队列只含有距离为 currDist 的顶点。当添加距离为 currDist + 1 的那些邻接顶点时，由于它们自队尾入队，因此保证它们直到所有距离为 currDist 的顶点都被处理之后才处理。下面给出无权最短路径问题的伪代码。

算法 5.23

```
void unweighted(Vertex s)
{    Queue < Vertex > q = new Queue < Vertex >();                    //一个队列
```

```
    for each Vertex v                          //每个顶点初始距离为 INFINITY
        v.dist = INFINITY;
        s.dist = 0;                            //s 初始距离为 0
        q.enqueue(s);
        while(!q.isEmpty())
        {   Vertex v = q.dequeue();
            for each Vertex w adjacent to v    //遍历 v 的邻接顶点
                if(w.dist == INFINITY)         //如果 dist 是 INFINITY 说明没有处理过
                {   w.dist = v.dist + 1;
                    w.path = v;
                    q.enqueue(w);
                }
        }
}
```

5.5.2　Dijkstra 算法


观看视频

Dijkstra 算法是由迪杰斯特拉(Dijkstra)提出的一个按路径长度递增的次序产生最短路径的算法。这个算法是迪杰斯特拉教授在他 26 岁陪未婚妻逛街时想出来的,数学家在逛街疲惫休息时想出了如何提高逛街效率的方法,查找逛街的最短路径。对于权值全为正的图,Dijkstra 算法是解决单源最短路径的常用算法。凭借这一算法,迪杰斯特拉教授获得了图灵奖。

1. Dijkstra 算法的思想

设 $G=(V,E)$是一幅带权有向图(无向可以转换为双向有向),设置两个顶点的集合 S 和 $T=V-S$,集合 S 中存放已找到最短路径的顶点,集合 T 存放当前还未找到最短路径的顶点。初始状态时,集合 S 中只包含源点 v_0,然后不断从集合 T 中选取到顶点 v_0 路径长度最短的顶点 u 加入集合 S 中,集合 S 每加入一个新的顶点 u,都要修改顶点 v_0 到集合 T 中剩余顶点的最短路径长度值,集合 T 中各顶点新的最短路径长度值为原来的最短路径长度值与顶点 u 的最短路径长度值加上 u 到该顶点的路径长度值中的较小值。不断重复此过程,直到集合 T 的顶点全部加入 S 中为止。

Dijkstra 算法的正确性可以用反证法加以证明。假设下一条最短路径的终点为 x,那么,该路径必然或者是弧(v_0,x),或者是中间只经过集合 S 中的顶点而到达顶点 x 的路径。因为假若此路径上除 x 之外有一个或一个以上的顶点不在集合 S 中,那么必然存在另外的终点不在 S 中而路径长度比此路径还短的路径,这与按路径长度递增的顺序产生最短路径的前提相矛盾,所以此假设不成立。

2. Dijkstra 算法的具体步骤


观看视频

(1) 初始时,S 只包含源点,即 $S=\{v\}$,v 的距离 dist[v]为 0。T 包含除 v 外的其他顶点,T 中顶点 u 距离 dist[u]为边上的权值(有边$<v,u>$)或为∞(没有边$<v,u>$)。

(2) 从 T 中选取一个距离 v(dist[k])最小的顶点 k,把 k 加入 S 中(该选定的距离就是 v 到 k 的最短路径长度)。

(3) 以 k 为新考虑的中间点,修改 T 中各顶点的距离;若从源点 v 到顶点 $u(u\in T)$的距离(经过顶点 k)比原来距离(不经过顶点 k)短,则修改顶点 u 的距离值,修改后的距离值

为顶点 k 的距离加上边上的权(即如果 dist[k]+w[k,u]<dist[u],那么把 dist[u]更新成更短的距离 dist[k]+w[k,u])。

(4) 重复步骤(2)和(3),直到所有顶点都包含在 S 中(要循环 $n-1$ 次)。

由此求得从 v 到图上其余各顶点的最短路径是依路径长度递增的序列。

3. Dijkstra 算法的实现

Dijkstra 算法最简单的实现方法就是,在每次循环中,再用一个循环找距离最短的点,然后用任意的方法更新与其相邻的边,时间复杂度显然为 $O(n^2)$。

对于空间复杂度,如果只要求出距离,只要 n 的附加空间保存距离就可以了(距离小于当前距离的是已访问的结点,对于距离相等的情况可以比较编号或是特殊处理一下)。如果要求出路径则需要另外 V 的空间保存前一个结点,共需要 $2n$ 的空间。

首先,引进一个辅助向量 ***D***,它的每个分量 $D[i]$表示当前所找到的从始点 v 到每个终点 v_i 的最短路径的长度。其次,假设用带权的邻接矩阵 **edges** 来表示带权有向图,**edges**[i][j]表示弧<v_i,v_j>上的权值。若<v_i,v_j>不存在,则置 **edges**[i][j]为∞。

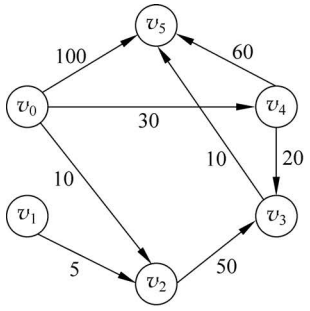
算法 5.24　用 C 语言描述的 Dijkstra 算法

```
void Dijkstra(Mgraph G,int v0,PathMatrix * p, ShortPathTable * D)
{ /* 用 Dijkstra 算法求有向网 G 的 v0 顶点到其余各顶点 v 的最短路径 P[v]及其路径长度 D[v] */
  /* 若 P[v][w]为 TRUE,则 w 是从 v0 到 v 当前求得最短路径上的顶点 */
  /* final[v]为 TRUE 当且仅当 v∈S,即已经求得从 v0 到 v 的最短路径 */
  /* 常量 INFINITY 为边上权值可能的最大值 */
     for (v = 0;v < G.vexnum;++v)
     {    fianl[v] = FALSE; D[v] = G.edges[v0][v];
          for (w = 0; w < G.vexnum; ++w) P[v][w] = FALSE;        /* 设空路径 */
          if (D[v]< INFINITY) {P[v][v0] = TRUE; P[v][w] = TRUE;}
     }
     D[v0] = 0; final[v0] = TRUE;                    /* 初始化,v0 顶点属于 S 集 */
  /* 开始主循环,每次求得 v0 到某个 v 顶点的最短路径,并加 v 到 S 集 */
     for(i = 1; i < G.vexnum; ++i)                   /* 其余 G.vexnum - 1 个顶点 */
     {    min = INFINITY;                            /* min 为当前所知离 v0 顶点的最近距离 */
          for (w = 0;w < G.vexnum;++w)
          if (!final[w])                             /* w 顶点在 V-S 中 */
              if (D[w]< min) {v = w; min = D[w];}
          final[v] = TRUE                            /* 离 v0 顶点最近的 v 加入 S 集合 */
          for(w = 0;w > G.vexnum;++w)                /* 更新当前最短路径 */
              if (!final[w]&&(min + G.edges[v][w]< D[w]))     /* 修改 D[w]和 P[w],w∈V - S */
              {    D[w] = min + G.edges[v][w];
                   P[w] = P[v]; P[w][v] = TRUE; /* P[w] = P[v] + P[w] */
              }
     }
}                                                    /* Dijkstra */
```

4. Dijkstra 算法过程演示

例如,图 5.37(a)所示有向网图 G_9 的带权邻接矩阵如图 5.37(b)所示。

若对 G_9 施行 Dijkstra 算法,则所得从 v_0 到其余各顶点的最短路径,以及运算过程中 ***D*** 向量的变化状况如表 5.3 所示。



(a) 有向图G_9

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

(b) G_9的邻接矩阵

图 5.37　有向网图 G_9 及其邻接矩阵

表 5.3　用 Dijkstra 算法构造单源点最短路径过程中各参数的变化示意

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	∞	∞	∞	∞	∞
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	$\{v_0,v_2\}$	$\{v_0,v_2,v_4\}$	$\{v_0,v_2,v_3,v_4\}$	$\{v_0,v_2,v_3,v_4,v_5\}$	

下面分析一下这个算法的运行时间。第一个 for 循环的时间复杂度是 $O(n)$，第二个 for 循环共进行 $n-1$ 次，每次执行的时间是 $O(n)$。所以总的时间复杂度是 $O(n^2)$。如果用带权的邻接表作为有向图的存储结构，则虽然修改 $\boldsymbol{D}$ 的时间可以减少，但由于在 $\boldsymbol{D}$ 向量中选择最小分量的时间不变，所以总的时间复杂度仍为 $O(n^2)$。

如果只希望找到从源点到某一个特定的终点的最短路径，从上面求最短路径的原理来看，这个问题和求源点到其他所有顶点的最短路径一样复杂，其时间复杂度也是 $O(n^2)$。

5. Dijkstra 算法实战练习

例 5.5　直到奶牛回家(Till the Cows Come Home)。问题：共有 N 个结点和 T 条边组成的无向图，现在求源点 N 到结点 1(Home)的最短路径($2\leqslant N\leqslant 1000, 1\leqslant T\leqslant 2000$)。

输入第一行是两个整数 T 和 N，第二行至第 $T+1$ 行，每行三个用空格分开的整数，分别表示顶点对和权值，即(V_i, V_j, W)。输出回家$(N\sim 1)$的最短距离。[POJ 2387]

输入示例

```
5 5
1 2 20
2 3 30
3 4 20
4 5 20
1 5 100
```

输出示例

```
90
```

解题思路如下。

本题属于简单的模板题。但用 Dijkstra 算法找最短路要注意此题为无向图，所以需要

考虑可能会存在重复的边，用邻接矩阵表示时有 $a[i][j]=a[j][i]$。

算法 5.25

```
#include<iostream>
using namespace std;
#define inf 1<<29
#define MAXV 1005
int map[MAXV][MAXV];
int n,m;

void dijkstra(){
    int i,j,min,v;
    int d[MAXV];
    bool vis[MAXV];
    for(i=1;i<=n;i++){
        vis[i]=0;
        d[i]=map[1][i];
    }
    for(i=1;i<=n;i++){
        min=inf;
        for(j=1;j<=n;j++)
            if(!vis[j] && d[j]<min){
                v=j;
                min=d[j];
            }
        vis[v]=1;
        for(j=1;j<=n;j++)
            if(!vis[j] && d[j]>map[v][j]+d[v])
                d[j]=map[v][j]+d[v];
    }
    printf("%d\n",d[n]);
}

int main(){
    int i,j,a,b,c;
    while(~scanf("%d%d",&m,&n)){
        for(i=1;i<=n;i++)
        for(j=1;j<=n;j++)
            if(i==j)
                map[i][i]=0;
            else map[i][j]=map[j][i]=inf;

        for(i=1;i<=m;i++){
            scanf("%d%d%d",&a,&b,&c);
            if(map[a][b]>c) map[a][b]=map[b][a]=c;
        }
        dijkstra();
    }
    return 0;
}
```

Dijkstra 算法的核心是以起始点为中心向外层层扩展，直到扩展到终点为止。对于单源最短路径问题，一般有以下两种经典解法。

（1）对于有权值为负的图，采用 Bellman-Ford 算法。

（2）对于权值全为正的图，常采用 Dijkstra 算法。

Bellman-Ford 算法将在 5.5.3 节介绍。


观看视频

5.5.3 具有负值边的图

如果图具有负值边，那么 Dijkstra 算法是行不通的。问题在于一旦一个顶点 u 被声明是已知的，就可能从某个另外的未知顶点 v 有一条回到 u 的负的路径。

一个诱人的解决方案是将一个常数 Δ 加到每条边上，从而除去负值边，再计算新图的最短路径，然后把结果用到原来的图上。这种方案不可能直接实现，因为那些须有许多条边的路径变得比那些具有很少边的路径权重更重了。另一个思路是把带权和无权的算法结合起来将会解决这个问题，但是要付出运行时间激烈增长的代价。下面主要介绍一个常用的能解决该问题的 Bellman-Ford 算法。

Bellman-Ford 算法是由美国数学家理查德·贝尔曼(Richard Bellman，动态规划的提出者)和小莱斯特·福特(Lester Ford)发明的。

1. Bellman-Ford 算法思想

Bellman-Ford 算法能在更普遍的情况下(存在负权边)解决单源点最短路径问题。对于给定的带权(有向或无向)图 $G=(V,E)$，其源点为 s，加权函数 w 是边集 E 的映射。对图 G 运行 Bellman-Ford 算法的结果是一个布尔值，表明图中是否存在着一个从源点 s 可达的负权回路。若不存在这样的回路，算法将给出从源点 s 到图 G 的任意顶点 v 的最短路径 Distant[v]，否则无解。

2. Bellman-Ford 算法流程

(1) 初始化：数组 Distant[i]记录从源点 s 到顶点 i 的路径长度，初始化数组 Distant[i]，源点 s 的 Distant[s]为 0，除源点外其他顶点 i 的 Distant[i]为∞。

(2) 迭代求解：反复对边集 E 中的每条边进行松弛操作，使得顶点集 V 中的每个顶点 v 的最短距离估计值逐步逼近其最短距离(运行 $|v|-1$ 次)，即对于每条边 $e(u,v)$，如果 Distant[u]+$w(u,v)$ < Distant[v]，则令 Distant[v] = Distant[u]+$w(u,v)$。$w(u,v)$为边 $e(u,v)$的权值。

若上述操作没有对 Distant 进行更新，说明最短路径已经查找完毕，或者部分点不可达，跳出循环；否则执行下次循环。

(3) 检验负权回路：判断边集 E 中的每条边的两个端点是否收敛，即对于每条边 $e(u,v)$，如果存在 Distant[u] + $w(u,v)$ < Distant[v]的边，且权值之和小于 0，则图中存在负环路，该图无法求出单源最短路径。如果存在不收敛的顶点，则算法返回 false，表明问题无解；否则算法返回 true，并且从源点可达的顶点 v 的最短距离保存在 Distant[v]中。

算法描述如下。

```
Bellman-Ford(G,w,s): boolean                    //图 G,边集函数 w,s 为源点
    for each vertex v ∈ V(G) do                 //初始化,1 阶段
        d[v] ←+ ∞
        d[s] ←0;                                //1 阶段结束
        for i=1 to |v| - 1 do                   //2 阶段开始,双重循环
            for each edge(u,v)∈E(G) do          //边集数组要用到,穷举每条边
                If d[v]>d[u] + w(u,v) then      //松弛判断
                    d[v] = d[u] + w(u,v)        //松弛操作,2 阶段结束
```

```
        for each edge(u,v)∈E(G) do
              If d[v]> d[u] + w(u,v) then
                    return false
    return true
```

Bellman-Ford 算法寻找单源最短路径的时间复杂度为 $O(V \cdot E)$。

3. Bellman-Ford 算法描述性证明

首先，图的任意一条最短路径既不能包含负权回路，也不会包含正权回路，因此它最多包含 $|v|-1$ 条边。

其次，从源点 s 可达的所有顶点如果存在最短路径，则这些最短路径构成一个以 s 为根的最短路径树。Bellman-Ford 算法的迭代松弛操作，实际上就是按顶点距离 s 的层次，逐层生成这棵最短路径树的过程。

在对每条边进行第 1 遍松弛时，生成了从 s 出发，层次至多为 1 的那些树枝。也就是说，找到了与 s 至多有 1 条边相连的那些顶点的最短路径；在对每条边进行第 2 遍松弛时，生成了第 2 层次的树枝，就是说找到了经过 2 条边相连的那些顶点的最短路径……因为最短路径最多只包含 $|v|-1$ 条边，所以，只需要循环 $|v|-1$ 次。

每实施一次松弛操作，最短路径树上就会有一层顶点达到其最短距离，此后这层顶点的最短距离值就会一直保持不变，不再受后续松弛操作的影响。

如果没有负权回路，由于最短路径树的高度最多只能是 $|v|-1$，因此最多经过 $|v|-1$ 遍松弛操作后，所有从 s 可达的顶点必将求出最短距离。如果 $d[v]$仍保持∞，则表明从 s 到 v 不可达。如果有负权回路，第 $|v|-1$ 遍松弛操作仍会成功，但负权回路上的顶点不会收敛。

4. Bellman-Ford 算法过程演示

Bellman-Ford 算法是最简单的算法，就是从开始结点开始循环每一条边，对它进行松弛操作，最后得到的路径就是最短路径。执行过程如图 5.38 所示。

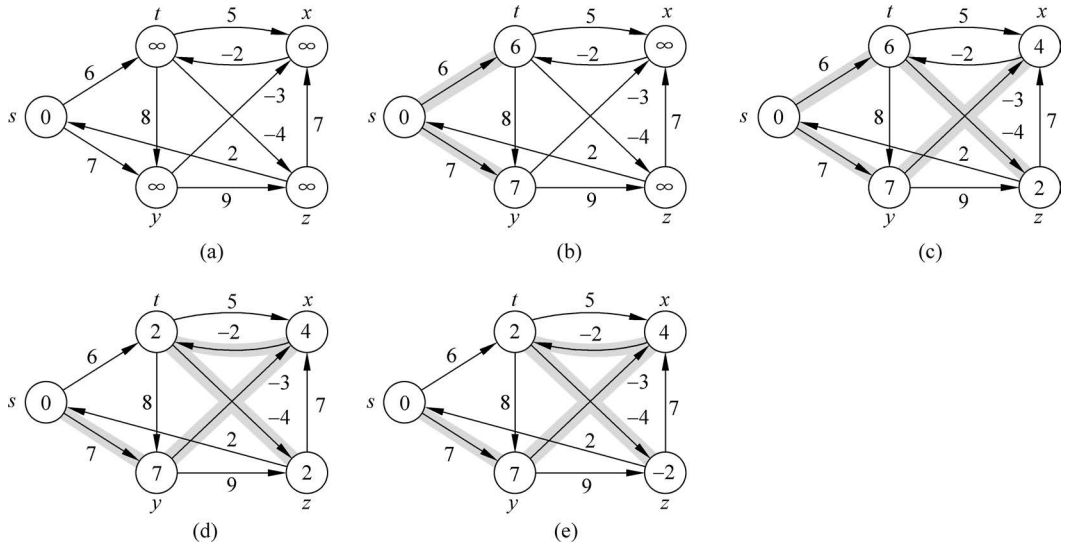


图 5.38　Bellman-Ford 算法的执行过程

在图 5.38 中，源点是顶点 s。d 值被标记在顶点内，阴影覆盖的边指示了前驱值。图 5.38(a)示出了对边进行第一趟操作前的情况。图 5.38(b)～图 5.38(e)示出了每一趟连续对边操作后的情况。图 5.38(e)中 d 的值是最终结果。Bellman-Ford 算法在本例中返回的是 True。

5. Bellman-Ford 算法参考代码

算法 5.26

```
#include<iostream>
#include<cstdio>
using namespace std;
#define MAX 0x3f3f3f3f
#define N 1010
int nodenum, edgenum, original;                    //点、边、起点
typedef struct Edge                                //边
{   int u, v;
    int cost;
}Edge;
Edge edge[N];
int dis[N], pre[N];
bool Bellman_Ford()
{   for(int i = 1; i <= nodenum; ++i)              //初始化
        dis[i] = (i == original ? 0 : MAX);
    for(int i = 1; i <= nodenum - 1; ++i)
        for(int j = 1; j <= edgenum; ++j)
            if(dis[edge[j].v] > dis[edge[j].u] + edge[j].cost)//松弛(顺序一定不能反)
            {   dis[edge[j].v] = dis[edge[j].u] + edge[j].cost;
                pre[edge[j].v] = edge[j].u;
            }
            bool flag = 1;                         //判断是否含有负权回路
            for(int i = 1; i <= edgenum; ++i)
                if(dis[edge[i].v] > dis[edge[i].u] + edge[i].cost)
                {   flag = 0;
                    break;
                }
                return flag;
}

void print_path(int root)                          //打印最短路的路径(反向)
{   while(root != pre[root])                       //前驱
    {   printf("%d-->", root);
        root = pre[root];
    }
    if(root == pre[root])
        printf("%d\n", root);
}

int main()
{   scanf("%d%d%d", &nodenum, &edgenum, &original);
    pre[original] = original;
    for(int i = 1; i <= edgenum; ++i)
    {   scanf("%d%d%d", &edge[i].u, &edge[i].v, &edge[i].cost);
    }
    if(Bellman_Ford())
```

```
        for(int i = 1; i <= nodenum; ++i)   //每个点的最短路
        { printf("%d\n", dis[i]);
            printf("Path:");
            print_path(i);
        }
    else
        printf("have negative circle\n");
    return 0;
}
```

建议读者利用 Bellman-Ford 算法重解例 5.5。


观看视频

5.5.4 所有点对的最短路径

Dijkstra 算法是求单源最短路径的，如果求图中所有点对的最短路径，则有以下两种解法。

(1) 以图中的每个顶点作为源点，分别调用 Dijkstra 算法，时间复杂度为 $O(n^3)$。

(2) Floyd 算法更简洁，但算法时间复杂度仍为 $O(n^3)$。

本节主要介绍 Floyd 提出的一个算法。

Floyd 算法是另一种经典的最短路径算法，不同的是 Dijkstra 算法仅计算了从一个起点出发的最短路径，而 Floyd 算法可以计算全部结点到其他结点的最短路径。Floyd 算法的基本思想也是松弛。这是一个动态规划的经典例子，在求解各点到其他点的最短路径的过程中往往会有很多的重叠问题，通过表 $D[][]$将这些问题保存下来，避免了重复的计算。

1. Floyd 算法基本思想

Floyd 算法仍从图的带权邻接矩阵 **cost** 出发，假设求从顶点 v_i 到 v_j 的最短路径。如果从 v_i 到 v_j 有弧，则从 v_i 到 v_j 存在一条长度为 edges$[i][j]$的路径，该路径不一定是最短路径，尚需进行 n 次试探。首先考虑路径(v_i,v_0,v_j)是否存在(即判别弧(v_i,v_0)和(v_0,v_j)是否存在)。如果存在，则比较(v_i,v_j)和(v_i,v_0,v_j)的路径长度，取长度较短者为从 v_i 到 v_j 的中间顶点的序号不大于 0 的最短路径。假如在路径上再增加一个顶点 v_1，也就是说，如果$(v_i,\cdots,v_1)$和$(v_1,\cdots,v_j)$分别是当前找到的中间顶点的序号不大于 1 的最短路径，那么$(v_i,\cdots,v_1,\cdots,v_j)$就有可能是从 v_i 到 v_j 的中间顶点序号不大于 1 的最短路径。将它和已经得到的从 v_i 到 v_j 中间顶点序号不大于 0 的最短路径相比较，从中选出中间顶点序号不大于 1 的最短路径，再增加一个顶点 v_2，继续进行试探，以此类推。在一般情况下，若$(v_i,\cdots,v_k)$和$(v_k,\cdots,v_j)$分别是从 v_i 到 v_k 和从 v_k 到 v_j 的中间顶点的序号不大于 k 的最短路径，则将$(v_i,\cdots,v_k,\cdots,v_j)$和已经得到的从 v_i 到 v_j 且中间顶点序号不大于 $k-1$ 的最短路径相比较，其长度较短者便是从 v_i 到 v_j 的中间顶点的序号不大于 k 的最短路径。这样，在经过 n 次比较后，最后求得的必是从 v_i 到 v_j 的最短路径。按此方法，可以同时求得各对顶点间的最短路径。

2. Floyd 算法的基本步骤

现定义一个 n 阶方阵序列：$\boldsymbol{D}^{(-1)},\boldsymbol{D}^{(0)},\boldsymbol{D}^{(1)},\cdots,\boldsymbol{D}^{(k)},\boldsymbol{D}^{(n-1)}$。

初始化：$\boldsymbol{D}^{(-1)}=\mathbf{cost}$，$D^{(-1)}[i][j]=$ edges$[i][j]$，表示初始的从 i 到 j 的中间不经过其

他中间点的最短路径。

迭代：设 $\boldsymbol{D}^{(k-1)}$ 已求出，如何得到 $\boldsymbol{D}^{(k)}$（$0 \leqslant k \leqslant n-1$）是该算法的关键，也是该算法中动态规划的主要思想，由 Floyd 算法基本思想可得：

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

从上述计算公式可见，$D^{(1)}[i][j]$ 是从 v_i 到 v_j 的中间顶点的序号不大于 1 的最短路径的长度；$D^{(k)}[i][j]$ 是从 v_i 到 v_j 的中间顶点的个数不大于 k 的最短路径的长度；$D^{(n-1)}[i][j]$ 就是从 v_i 到 v_j 的最短路径的长度。

3. Floyd 算法实现

由上述动态规划方程可知，可以用 3 个 for 循环来实现 Floyd 算法，需要注意的是 for 循环的嵌套顺序：

```
for(int k = 0; k < n; k++)
    for(int i = 0; i < n; i++)
        for(int j = 0; j < n; j++)
```

如果嵌套的顺序是习惯上的 i、j、k，而不是现在的 k、i、j，则所得的结果就会出现问题。

为了保存最短路径所行经的路径，这里要用到另一个矩阵 $\boldsymbol{P}$，它的定义是：$P[i][j]$ 的值如果为 p，就表示 i 到 j 的最短行经为 $i \rightarrow \cdots p \rightarrow j$，即 p 是 i 到 j 的最短行径中 j 之前的最后一个顶点。$\boldsymbol{P}$ 矩阵的初值为 $P[i][j]=i$。因此，采用逆序的方法即可输出实际的行径。

当 $D[i][j]>D[i][k]+D[k][j]$ 时，就把 $P[k][j]$ 存入 $P[i][j]$。

由此得到求任意两顶点间的最短路径的算法。

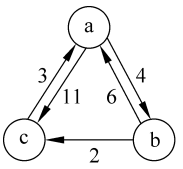
算法 5.27

```
void Floyd(Mgraph G, PathMatrix *P[],DistancMatrix *D)
{ /*用 Floyd 算法求有向网 G 中各对顶点 v 和 w 之间的最短路径 P[v][w]及其带权长度 D[v][w]*/
  /*若 P[v][w][u]为 TRUE,则 u 是从 v 到 w 当前求得的最短路径上的顶点*/
  for(v = 0;v < G.vexnum;++v)                    /*各对顶点之间初始已知路径及距离*/
   for(w = 0;w < G,vexnum;++w)
    { D[v][w] = G.arcs[v][w];
      for(u = 0;u < G,vexnum;++u) P[v][w][u] = FALSE;
      if (D[v][w]< INFINITY)                     /*从 v 到 w 有直接路径*/
       { P[v][w][v] = TRUE;
       }
    }
  for(u = 0; u < G.vexnum; ++u)
    for(v = 0; v < G.vexnum; ++v)
      for(w = 0;w < G.vexnum;++w)
        if (D[v][u] + D[u][w]< D[v][w])          /*从 v 经 u 到 w 的一条路径更短*/
         {D[v][w] = D[v][u] + D[u][w];
          for(i = 0;i < G.vexnum;++i)
          P[v][w][i] = P[v][u][i]||P[u][w][i];
         }
}                                                /*Floyd*/
```

4. Floyd 算法过程演示

图 5.39 给出了一个简单的有向网及其邻接矩阵。图 5.40 给出了用 Floyd 算法求该有

向网中每对顶点之间的最短路径过程中，数组 $\boldsymbol{D}$ 和数组 $\boldsymbol{P}$ 的变化情况。



(a) 有向网G_9

$$\begin{bmatrix} 0 & 4 & 11 \\ 6 & 0 & 2 \\ 3 & \infty & 0 \end{bmatrix}$$

(b) G_9的邻接矩阵

图 5.39 有向网图 G_9 及其邻接矩阵

$$\boldsymbol{D}^{(-1)}=\begin{bmatrix} 0 & 4 & 11 \\ 6 & 0 & 2 \\ 3 & \infty & 0 \end{bmatrix} \quad \boldsymbol{D}^{(0)}=\begin{bmatrix} 0 & 4 & 11 \\ 6 & 0 & 2 \\ 3 & 7 & 0 \end{bmatrix} \quad \boldsymbol{D}^{(1)}=\begin{bmatrix} 0 & 4 & 6 \\ 6 & 0 & 2 \\ 3 & 7 & 0 \end{bmatrix} \quad \boldsymbol{D}^{(2)}=\begin{bmatrix} 0 & 4 & 6 \\ 5 & 0 & 2 \\ 3 & 7 & 0 \end{bmatrix}$$

$$\boldsymbol{P}^{(-1)}=\begin{bmatrix} & ab & ac \\ ba & & bc \\ ca & & \end{bmatrix} \quad \boldsymbol{P}^{(0)}=\begin{bmatrix} & ab & ac \\ ba & & bc \\ ca & cab & \end{bmatrix} \quad \boldsymbol{P}^{(1)}=\begin{bmatrix} & ab & abc \\ ba & & bc \\ ca & cab & \end{bmatrix} \quad \boldsymbol{P}^{(2)}=\begin{bmatrix} & ab & abc \\ bca & & bc \\ ca & cab & \end{bmatrix}$$

图 5.40 Floyd 算法执行时数组 $\boldsymbol{D}$ 和数组 $\boldsymbol{P}$ 取值的变化示意

5. Floyd 算法实战练习

例 5.6 股票经纪小道消息(Stockbroker Grapevine)：股票经纪人要在一群人中散布一个传言，传言只能在认识的人中传递，题目将给出人与人的关系(是否认识)，以及传言在某两个认识的人中传递所需的时间，要求程序给出以哪个人为起点，可以在耗时最短的情况下，让所有人收到消息。

输入首行是股票经纪人数 n，接下来每一行表示某个经纪人的联系信息(联系数，联系人，传递耗时)，没有特殊的标点符号或间距规则。股票经纪人数按 1～100 编号，传递信息耗时 1～10 分钟，联系数为 0～$n-1$，n 为 0 时输入结束。输出股票经纪人最快的传输时间(保留整数)。

如果图中某个点是不可达的，则输出 disjoint，如果 A、B 间可互传信息，则 A 到 B 的传输时间不一定等于 B 到 A 传输时间。[POJ 1125]

解题思路如下。

题目是要求从某一结点开始，能让消耗的总时间最短。实际上这是一个在有向图中求最短路径问题，先求出每个人向其他人发信息所用的最短时间(当然不是每个人都能向所有人发信息)，然后在所有能向每个人发信息的人中比较他们所用最大时间，找出所用最大时间最少的那一个即为所求。

输入示例	输出示例
2 2 4 3 5	3 2
2 1 2 3 6	3 10
2 1 2 2 2	
5	
3 4 4 2 8 5 3	
1 5 8	

```
4 1 6 4 10 2 7 5 2
3
0
2 2 5 1 5
0
```

参考代码如下。

算法 5.28

```
#include<iostream>
#include<string>
#include<cstring>
#include<algorithm>
#include<cstdio>
using namespace std;
const int maxn=1000;
const int inf=10000000;
int map[maxn][maxn];
void floyd(int n){
    for(int k=1;k<=n;++k)
        for(int i=1;i<=n;++i)
            for(int j=1;j<=n;++j)
                map[i][j]=min(map[i][j],map[i][k]+map[k][j]);
}
int main(){
    int n;
    while(~scanf("%d",&n),n){
        for(int i=1;i<=n;++i){
            for(int j=1;j<=n;++j)
                if(i==j) map[i][j]=0;
                else map[i][j]=inf;          //初始化
            int m; scanf("%d",&m);
            while(m--){
                int x,c; scanf("%d%d",&x,&c);
                if(c<map[i][x]) map[i][x]=c;
            }
        }
        floyd(n);
        int ans=inf,mj=-1;
        for(int i=1;i<=n;++i){
            int maxt=0;
            for(int j=1;j<=n;++j)
                maxt=max(maxt,map[i][j]);
            if(maxt<ans) ans=maxt,mj=i;
        }
        if(mj==-1) puts("disjoint");
        else printf("%d %d\n",mj,ans);
    }
    return 0;
}
```

比较 Dijkstra 和 Floyd 算法，不难得出以下的结论：对于稀疏图，采用 n 次 Dijkstra 比较出色；对于稠密图，可以使用 Floyd 算法；此外，Floyd 算法还可以处理带负边的图。

观看视频

5.6 最小支撑树

由生成树的定义可知，无向连通图的生成树不是唯一的。连通图的一次遍历所经过的边的集合及图中所有顶点的集合就构成了该图的一棵生成树，对连通图的不同遍历，就可能得到不同的生成树。图 5.41(a)～图 5.41(c)所示均为图 5.17 所示的无向连通图 G_5 的生成树。

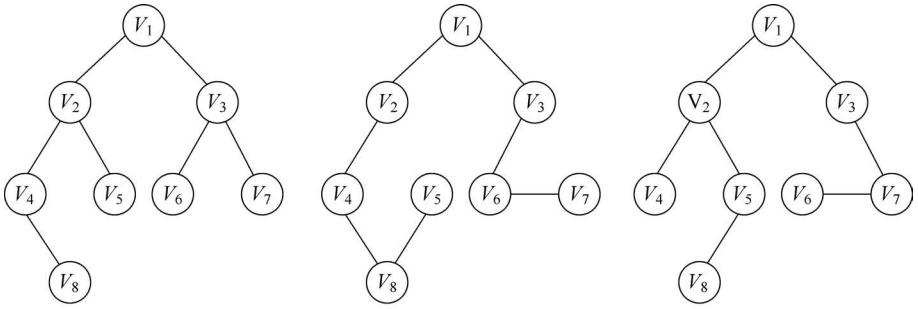


图 5.41 无向连通图 G_5 的 3 棵生成树

可以证明，对于有 n 个顶点的无向连通图，无论其生成树的形态如何，所有生成树中都有且仅有 $n-1$ 条边。如果无向连通图是一个网，那么它的所有生成树中必有一棵边的权值总和最小的生成树，称这棵生成树为最小生成树。

最小生成树的概念可以应用到许多实际问题中，如铁路进藏工程，如何以最低成本在世界屋脊修建“幸福天路”，此外，最小生成树算法在城市规划、电网、通信等领域都有广泛应用。例如以尽可能低的总造价建造城市间的通信网络，把 10 个城市联系在一起。在这 10 个城市中，任意两个城市之间都可以建造通信线路，通信线路的造价依据城市间的距离不同而有不同的造价，可以构造一个通信线路造价网络，在网络中，每个顶点表示城市，顶点之间的边表示城市之间可构造通信线路，每条边的权值表示该条通信线路的造价，要想使总的造价最低，实际上就是寻找该网络的最小生成树。

下面介绍两种常用的构造最小生成树的方法。

5.6.1 Prim 算法

假设 $G=(V,E)$为一网图，其中 V 为网图中所有顶点的集合，E 为网图中所有带权边的集合。设置两个新的集合 U 和 T，其中集合 U 用于存放 G 的最小生成树中的顶点，集合 T 存放 G 的最小生成树中的边。令集合 U 的初值为 $U=\{u_1\}$(假设构造最小生成树时，从顶点 u_1 出发)，集合 T 的初值为 $T=\{\}$。Prim 算法的思想是：从所有 $u\in U, v\in V-U$ 的边中，选取具有最小权值的边(u,v)，将顶点 v 加入集合 U 中，将边(u,v)加入集合 T 中，如此不断重复，直到 $U=V$ 时，最小生成树构造完毕，这时集合 T 中包含了最小生成树的所有边。

Prim 算法可用下述过程描述，其中用 w_{uv} 表示顶点 u 与顶点 v 边上的权值。

(1) U ={u}，T ={}；

(2) while (U ≠ V)do

(u,v)= min{w_{uv}; u∈U,v∈V − U}

T = T +{(u,v)}

U = U +{v};

(3) 结束。

如图 5.42(a)所示网图,按照 Prim 方法,从顶点 V_1 出发,该网的最小生成树的产生过程如图 5.42(b)～图 5.42(g)所示。

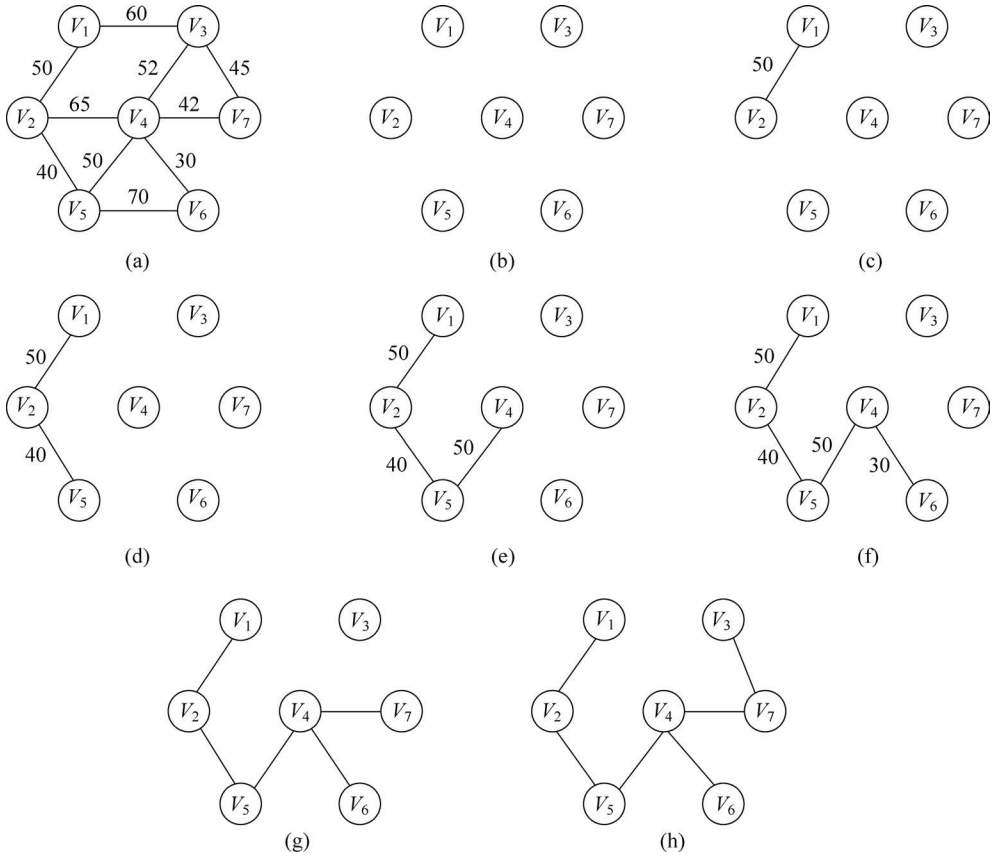


图 5.42　Prim 算法构造最小生成树的过程示意

为实现 Prim 算法,需设置两个辅助一维数组:lowcost 和 closevertex。其中,lowcost 用来保存集合 $V-U$ 中各顶点与集合 U 中各顶点构成的边中具有最小权值的边的权值;数组 closevertex 用来保存依附于该边的在集合 U 中的顶点。假设初始状态时,$U=\{u_1\}$(u_1 为出发的顶点),这时有 lowcost[0]=0,它表示顶点 u_1 已加入集合 U 中,数组 lowcost 的其他各分量的值是顶点 u_1 到其余各顶点所构成的直接边的权值。然后不断选取权值最小的边(u_i,u_k)($u_i \in U$,$u_k \in V-U$),每选取一条边,就将 lowcost(k)置为 0,表示顶点 u_k 已加入集合 U 中。由于顶点 u_k 从集合 $V-U$ 进入集合 U 后,这两个集合的内容发生了变化,因此需依据具体情况更新数组 lowcost 和 closevertex 中部分分量的内容。最后 closevertex 中即为所建立的最小生成树。

当无向网采用二维数组存储的邻接矩阵存储时,Prim 算法的 C 语言实现如下。

算法 5.29

```
void Prim(int gm[][MAXNODE],int n,int closevertex[])
{ /* 用 Prim 方法建立有 n 个顶点的邻接矩阵存储结构的网图 gm 的最小生成树 */
/* 从序号为 0 的顶点出发; 建立的最小生成树存于数组 closevertex 中 */
    int lowcost[100],mincost;
    int i,j,k;
    for (i = 1;i < n;i++)                          /* 初始化 */
    {   lowcost[i] = gm[0][i];
        closevertex[i] = 0;
    }
    lowcost[0] = 0;                                /* 从序号为 0 的顶点出发生成最小生成树 */
    closevertex[0] = 0;
    for (i = 1;i < n;i++)                          /* 寻找当前最小权值的边的顶点 */
       {   mincost = MAXCOST;                      /* MAXCOST 为一个极大的常量值 */
           j = 1;k = 1;
           while (j < n)
           {   if (lowcost[j]< mincost && lowcost[j]!= 0)
               {   mincost = lowcost[j];
                   k = j;
               }
               j++;
           }
           printf("顶点的序号 = %d 边的权值 = %d\n",k,mincost);
           lowcost[k] = 0;
           for (j = 1;j < n;j++)                   /* 修改其他顶点的边的权值和最小生成树顶点序号 */
           if (gm[k][j]< lowcost[j])
           {   lowcost[j] = gm[k][j];
               closevertex[j] = k;
           }
       }
}
```

表 5.4 给出了在用上述算法构造网图 5.42（a）的最小生成树的过程中，数组 closevertex、lowcost 及集合 U、$V-U$ 的变化情况，读者可进一步加深对 Prim 算法的了解。

在 Prim 算法中，第一个 for 循环的执行次数为 $n-1$，第二个 for 循环中又包括了一个 while 循环和一个 for 循环，执行次数为 $2(n-1)^2$，所以 Prim 算法的时间复杂度为 $O(n^2)$。

表 5.4 用 Prim 算法构造最小生成树过程中各参数的变化示意

顶点	(1)	(2)	(3)	(4)	(5)	(6)	(7)
	lowcost closevertex	lowcost closevertex	lowcost closevertex	lowcost closevertex	lowcost closevertex	lowcost closevertex	lowcost closevertex
v_1	0 1	0 1	0 1	0 1	0 1	0 1	0 1
v_2	50 1	0 1	0 1	0 1	0 1	0 1	0 1
v_3	60 1	60 1	60 1	52 4	52 4	45 7	0 7
v_4	∞ 1	65 2	50 5	0 5	0 5	0 5	0 5
v_5	∞ 1	40 2	0 2	0 2	0 2	0 2	0 2
v_6	∞ 1	∞ 1	70 5	30 4	0 4	0 4	0 4
v_7	∞ 1	∞ 1	∞ 1	42 4	42 4	0 4	0 4
U	$\{v_1\}$	$\{v_1,v_2\}$	$\{v_1,v_2,v_5\}$	$\{v_1,v_2,v_5,v_4\}$	$\{v_1,v_2,v_5,v_4,v_6\}$	$\{v_1,v_2,v_5,v_4,v_6,v_7\}$	$\{v_1,v_2,v_5,v_4,v_6,v_7,v_3\}$

续表

顶点	(1)	(2)	(3)	(4)	(5)	(6)	(7)
	lowcost closevertex	lowcost closevertex	lowcost closevertex	lowcost closevertex	lowcost closevertex	lowcost closevertex	lowcost closevertex
T	$\{\}$	$\{(v_1,v_2)\}$	$\{(v_1,v_2),(v_2,v_5)\}$	$\{(v_1,v_2),(v_2,v_5),(v_4,v_5)\}$	$\{(v_1,v_2),(v_2,v_5),(v_4,v_5),(v_4,v_6)\}$	$\{(v_1,v_2),(v_2,v_5),(v_4,v_5),(v_4,v_6),(v_4,v_7)\}$	$\{(v_1,v_2),(v_2,v_5),(v_4,v_5),(v_4,v_6),(v_4,v_7),(v_3,v_7)\}$

5.6.2　Kruskal 算法

观看视频

Kruskal 算法是一种按照网中边的权值递增的顺序构造最小生成树的方法。其基本思想是：设无向连通网为 $G=(V,E)$，令 G 的最小生成树为 T，其初态为 $T=(V,\{\})$，即开始时，最小生成树 T 由图 G 中的 n 个顶点构成，顶点之间没有一条边，这样 T 中各顶点各自构成一个连通分量。然后，按照边的权值由小到大的顺序，考察 G 的边集 E 中的各条边。若被考察的边的两个顶点属于 T 的两个不同的连通分量，则将此边作为最小生成树的边加入 T 中，同时把两个连通分量连接为一个连通分量；若被考察边的两个顶点属于同一个连通分量，则舍去此边，以免造成回路，如此下去，当 T 中的连通分量个数为 1 时，此连通分量便为 G 的一棵最小生成树。

对于图 5.42(a)所示的网，按照 Kruskal 方法构造最小生成树的过程如图 5.43 所示。在构造过程中，按照网中边的权值由小到大的顺序，不断选取当前未被选取的边集中权值最小的边。依据生成树的概念，n 个结点的生成树有 $n-1$ 条边，故重复上述过程，直到选取了 $n-1$ 条边为止，就构成了一棵最小生成树。

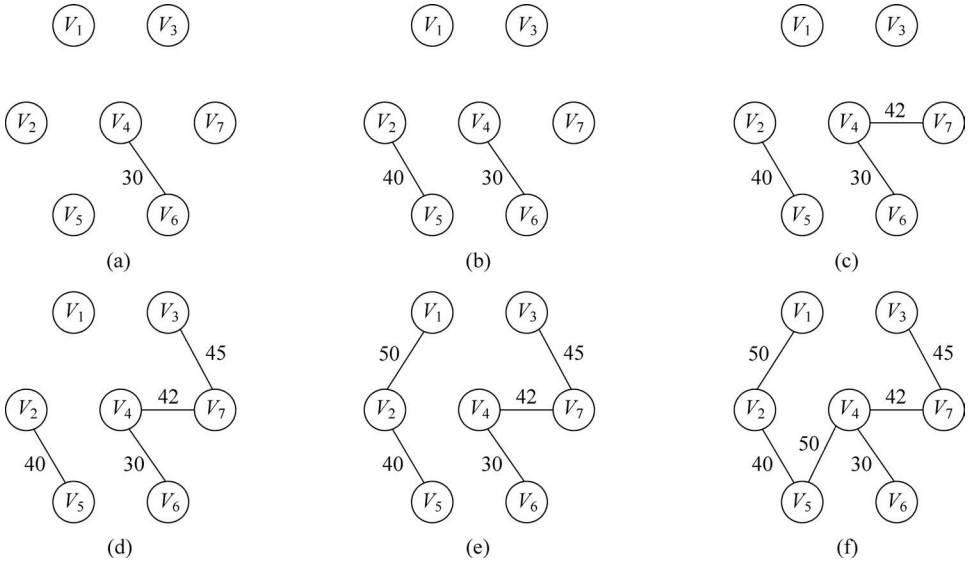


图 5.43　Kruskal 算法构造最小生成树的过程示意

下面介绍 Kruskal 算法的实现。

设置一个结构数组 edges 存储网中所有的边，边的结构类型包括构成的顶点信息和边权值，定义如下。

```
#define MAXEDGE  <图中的最大边数>
typedef struct {
    elemtype v1;
    elemtype v2;
    int cost;
    } EdgeType;
EdgeType edges[MAXEDGE];
```

在结构数组 edges 中，每个分量 edges[i]代表网中的一条边，其中 edges[i].v_1 和 edges[i].v_2 表示该边的两个顶点，edges[i].cost 表示这条边的权值。为了方便选取当前权值最小的边，事先把数组 edges 中的各元素按照其 cost 域值由小到大的顺序排列。在对连通分量合并时，采用集合的合并方法。对于有 n 个顶点的网，设置一个数组 father[n]，其初值为 father[i]=-1($i=0,1,\cdots,n-1$)，表示各顶点在不同的连通分量上，然后，依次取出 edges 数组中的每条边的两个顶点，查找它们所属的连通分量，假设 vf1 和 vf2 为两顶点所在的树的根结点在 father 数组中的序号，若 vf1 不等于 vf2，表明这条边的两个顶点不属于同一分量，则将这条边作为最小生成树的边输出，并合并它们所属的两个连通分量。

下面用 C 语言实现 Kruskal 算法，其中函数 Find 的作用是寻找图中顶点所在树的根结点在数组 father 中的序号。需说明的是，在程序中将顶点的数据类型定义成整型，而在实际应用中，可依据实际需要来设定。

算法 5.30

```
typedef int elemtype;
typedef struct {
    elemtype v1;
    elemtype v2;
    int cost;
   }EdgeType;
void Kruskal(EdgeType edges[ ],int n)
/*用 Kruskal 算法构造有 n 个顶点的图 edges 的最小生成树*/
{   int father[MAXEDGE];
    int i,j,vf1,vf2;
    for (i=0;i<n;i++) father[i]=-1;
    i=0;j=0;
    while(i<MAXEDGE && j<n-1)
    {   vf1=Find(father,edges[i].v1);
        vf2=Find(father,edges[i].v2);
        if (vf1!=vf2)
        {   father[vf2]=vf1;
            j++;
            printf("%3d%3d\n",edges[i].v1,edges[i].v2);
        }
        i++;
    }
}

    int Find(int father[ ],int v)
/*寻找顶点 v 所在树的根结点*/
```

```
{ int t;
  t = v;
  while(father[t]>=0)
    t = father[t];
  return(t);
}
```

在 Kruskal 算法中，第二个 while 循环是影响时间效率的主要操作，其循环次数最多为 MAXEDGE，其内部调用的 Find 函数的内部循环次数最多为 n，所以 Kruskal 算法的时间复杂度为 $O(n \cdot \text{MAXEDGE})$。

5.6.3 最小生成树算法应用

例 5.7 农业网(Agri-Net)：给出 N 个顶点及 N 个顶点间的距离，然后求一棵最小生成树。先输入顶点数 N，然后一个 $N \times N$ 的数组，用来描述各个顶点之间的距离。

输入包含若干种情况，每种情况，第一行是农场数 N ($3 \leqslant N \leqslant 100$)，接着是 $N \times N$ 的邻接距离矩阵，逻辑上说有 N 行以空格分开的 N 个整数，物理上说，每行长度限制为 80 个字符，所以有些行会延续到其他行。要求对每个用例以整数形式输出连接整个农场需要的光纤最小长度。[POJ 1258]

输入示例

```
4
0 4 9 21
4 0 8 17
9 8 0 16
21 17 16 0
```

输出示例

```
28
```

参考代码如下。

算法 5.31

```
#include<iostream>
using namespace std;

const int INFINITY = 9999999;
const int MAXVEX = 102;
int edge[MAXVEX][MAXVEX], lowcost[MAXVEX];
int vexNum;

int myPrim(int start);
int main()
{
    while(scanf("%d",&vexNum)!=EOF)
     {
        for(int i = 1; i <= vexNum; i++)
            for(int j = 1; j <= vexNum; j++)
                scanf("%d", &edge[i][j]);
      printf("%d\n",myprim(1));
     }
    return 0;
}
```

```
int myPrim(int start)
{
    int nextVex, minEdge, sumPath = 0;
    for(int i = 1; i <= vexNum; i++)
         lowcost[i] = edge[start][i];
    for(int i = 1; i < vexNum; i++){
         minEdge = INFINITY;
         nextVex = 1;
        for(int j = 2; j <= vexNum; j++){
            if((lowcost[j] > 0) && (lowcost[j] < minEdge)){
                 minEdge = lowcost[j];
                 nextVex = j;
             }
         }
         sumPath += minEdge;
         lowcost[nextVex] = 0;
        for(int j = 1; j <= vexNum; j++){
            if((edge[nextVex][j] < lowcost[j]) && (lowcost[j] > 0)){
                 lowcost[j] = edge[nextVex][j];
             }
         }
     }
    return sumPath;
}
```


观看视频

5.7 网络流问题

设给定边容量为 $C_{v,w}$ 的有向图 $G=(V,E)$。这些容量可以代表通过一个管道的水的容量或在两个交叉路口之间马路上的交通流量。有两个顶点：一个是 s，称为源点(Source)；另一个是 t，称为汇点(Sink)。对于任意一条边(v,w)，最多有“流”$C_{v,w}$ 个单位可以通过。在既不是源点 s 又不是汇点 t 的任一顶点 v，总的进入的流必须等于总的发出的流。最大流问题就是确定从 s 到 t 可以通过的最大流量。例如，对于图 5.44(a)，最大流是5，如图 5.44(b)所示。

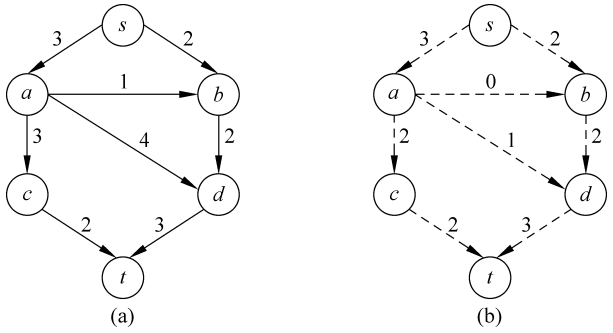


图 5.44 一幅图和它的最大流

正如问题叙述中所要求的，没有边负载超过它的容量的流。源点 s 将 5 个单位的流分给 a 和 b，顶点 a 有 3 个单位的流进入，它将这 3 个流分转给 c 和 d。顶点 d 从 a 和 b 得到3 个单位的流，并把它们结合起来发送到 t。一个顶点在不违反边的容量以及保持流守恒

(进入必须流出)的前提下,可以按任何方式结合和发送流。

5.7.1　网络流的最大流问题

本节开始讨论解决最大流问题的 Ford-Fulkerson 方法,该方法也称作“扩充路径方法”,该方法是大量算法的基础,有多种实现方法(如 Edmonds-Karp 算法、Dinic 算法等)。Ford-Fulkerson 算法是一种迭代算法,首先对图中所有顶点对的流清零,此时的网络流大小也为 0。在每次迭代中,通过寻找一条“增广路径”(Augument Path)来增加流的值。增广路径可以看作源点 s 到汇点 t 的一条路径,并且沿着这条路径可以增加更多的流。迭代直至无法再找到增广路径为止,此时必然从源点到汇点的所有路径中都至少有一条满边。

1. 一个简单的最大流问题

从图 G 开始并构造一幅流图 f,f 表示在算法的任意阶段已经达到的流。开始时 f 的所有边都没有流,希望当算法终止时 f 包含最大流。再构造一幅图 G_f,称为残余图(Residual Graph),它表示对于每条边还能再添加上多少流。对于每一条边,可以从容量中减去当前的流而计算出残余的流。G_f 的边叫作残余边(Residual Edge)。

所谓增广通路(Augmenting Path),指图 G_f 中从 s 到 t 的一条路径,而且在每个阶段,都需要找到这条路径。这条路径上的最小值边就是可以添加到路径每条边上的流量,这可以通过调整 f 和重新计算 G_f 来实现。当发现在 G_f 中没有从 s 到 t 的路径时算法终止。这个算法是不确定的,因为从 s 到 t 的路径是任意选择的。显然,有些选择会比另外一些选择好,后面再处理这个问题。针对例子运行这个算法。要记着这个算法有一个小欠缺。G、f 和 G_f 的初始配置如图 5.45 所示。

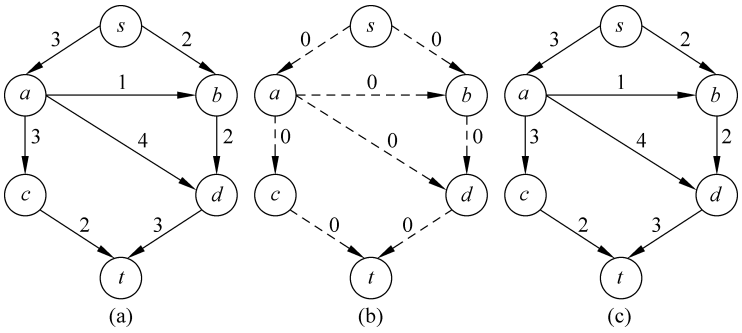


图 5.45　图、流图以及残余图的初始阶段

在残余图中有许多从 s 到 t 的路径。假设选择 s,b,d,t,此时可以发送 2 个单位的流通过这条路径的每一边。采用如下约定:一旦注满(使饱和)一条边,则这条边就要从残余图中除去。这样就得到图 5.46。

若选择路径 s,a,c,t,该路径也容许 2 个单位的流通量。进行必要的调整后,得到图 5.47 中的图。

唯一剩下可选择的路径是 s,a,d,t,这条路径能够容纳一个单位的流通过。结果得到如图 5.48 所示的图。

由于 t 从 s 出发是不可达到的,因此算法到此终止。结果正好 5 个单位的流是最大值。

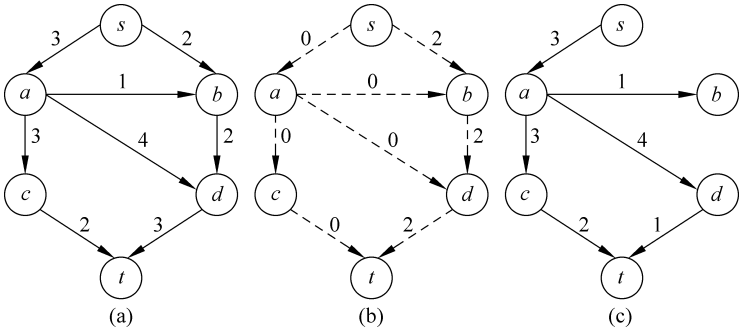


图 5.46 沿 s,b,d,t 加入 2 个单位的流后的 G、f、G_f

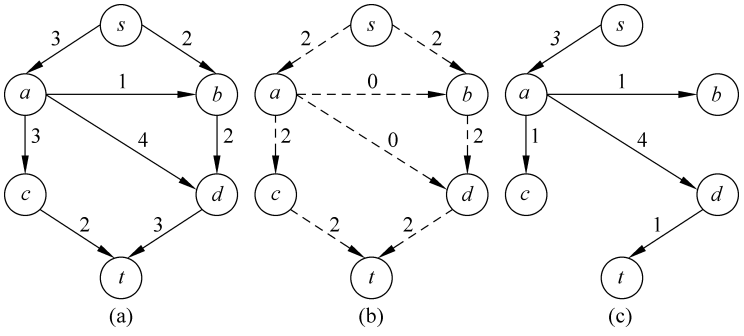


图 5.47 沿 s,a,c,t 加入 2 个单位的流后的 G、f、G_f

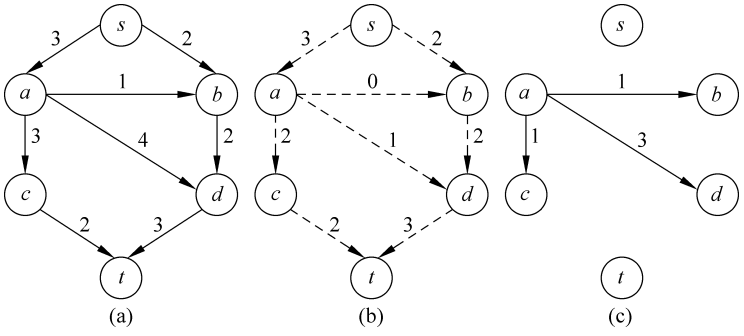


图 5.48 沿 s,a,d,t 加入 1 个单位的流后的 G、f、G_f——算法终止

为了看清问题的所在，设从初始图开始选择路径 s,a,d,t，这条路径容纳 3 个单位的流，从表面上看这是个好选择。然而选择的结果却使得在残余图中不再有从 s 到 t 的任何路径，因此，该算法不能找到最优解。这是贪婪算法行不通的一个例子。图 5.49 指出了为什么算法会失败。

为了使得算法有效，就需要让算法改变它的意向。为此，对于流图中具有流 $f_{v,w}$ 的每一边(v,w)将在残余图中添加一条容量为 $f_{v,w}$ 的边(w,v)。事实上，可以通过以相反的方向发回一个流而使算法改变它的意向。通过例子最能看清楚这个问题。从原始的图开始并选择增长通路 s,a,d,t 得到图 5.50 中的图。

注意，在残余图中有些边在 a 和 d 之间有两个方向。或者还有一个单位的流可以从 a 导向 d，或者有高达 3 个单位的流导向相反的方向——可以撤销流。现在算法找到流为 2

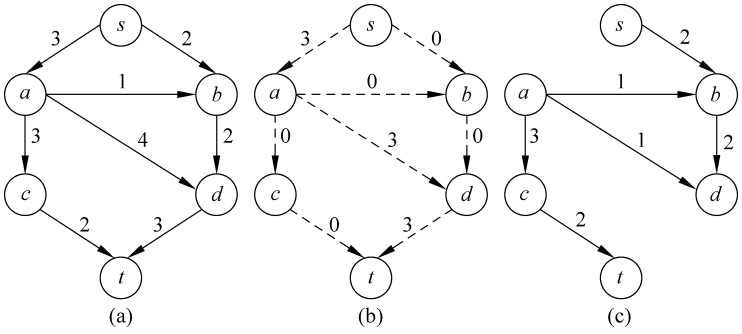


图 5.49　如果初始动作是沿 s,a,d,t 加入 3 个单位的流得到 G、f、G_f ——算法终止但解不是最优的

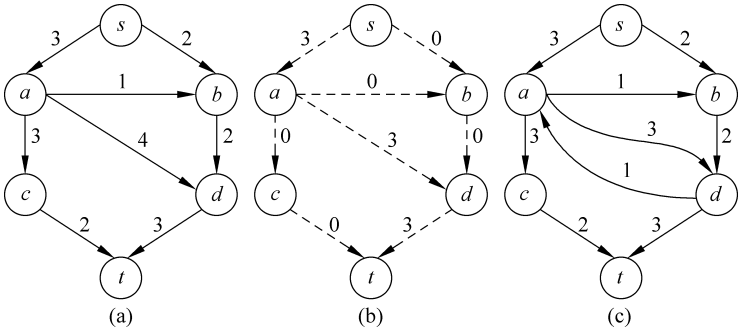


图 5.50　使用正确的算法沿 s,a,d,t 加入 3 个单位的流后的图

的增长通路 s,b,d,a,c,t。通过从 d 到 a 导入 2 个单位的流，算法从边(a,d)取走 2 个单位的流，因此本质上改变了它的意向。图 5.51 显示出新的图。

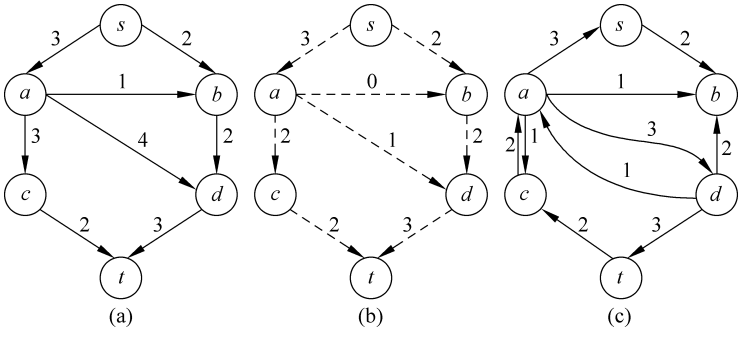


图 5.51　使用正确算法沿 s,b,d,a,c,t 加入 2 个单位的流后的图

在图 5.51 中没有增广通路，因此，算法终止。奇怪的是，可以证明，如果边的容量都是有理数，那么该算法总以最大流终止。证明多少有些困难，也超出了本书的范围。虽然例子正好是无环的，但这并不是算法有效工作所必需的。此处使用无环图只是为了简明。

2. Ford-Fulkerson 算法的正确性证明

利用最大流最小割定理可以证明 Ford-Fulkerson 算法的正确性。

最大流最小割定理：一个网中所有流中的最大值等于所有割中的最小容量。并且可以

证明以下3个条件等价。

(1) f 是流网络 G 的一个最大流。

(2) 残留网 G_f 不包含增广路径。

(3) G 的某个割(S, T),满足 $f(S, T) = c(S, T)$。

证明如下。

(1)(反证法)假设 f 是 G 的最大流,但是 G_f 中包含增广路径 p。显然此时沿着增广路径可以继续增大网络的流,则 f 不是 G 的最大流,与条件矛盾。

(2) 假设 G_f 中不包含增广路径,即 G_f 中不包含从 s 到 t 的路径。定义:

$S = \{v \in V:$ G_f 中包含 s 到 v 的路径$\}$

令 $T = V - S$,由于 G_f 中不存在从 s 到 t 的路径,因此 $t \notin S$,所以得到 G 的一个割(S, T)。对每对顶点 $u \in S$,$v \in T$,必须满足 $f(u, v) = c(u, v)$,否则边(u, v)就会存在于 G_f 的边集合中,那么 v 就应当属于S(而事实上是 $v \in T$)。所以,$f(S, T) = c(S, T)$。

(3) 网络的任何流的值都不大于任何一个割的容量,如果 G 的某个割(S, T),满足 $f(S, T) = c(S, T)$,则说明割(S, T)的流达到了网络流的上确界,它必然是最大流。

Ford-Fulkerson 算法的迭代终止条件是残留网中不包含增广路径,根据上面的等价条件,此时得到的流就是网络的最大流。

3. Ford-Fulkerson 算法的实现

依据上面的讨论,下面给出 Ford-Fulkerson 算法的伪代码。

算法 5.32

```
Ford - Fulkerson(G, s, t)
for each edge (u, v)∈E[G]                    //初始化每条边的流量为 0
{   f[u, v] = 0;
    f[v, u] = 0;
}
//Gf←G                                       //初始化剩余网络 Gf 为原网络 G,这里不需要代码
while there exists a path p from s to t in the network Gf   //网络中还存在增广路径,仍然进行迭代
{search a path p from network Gf                    //Edmonds-Karp 算法采用广度优先搜索算法,Dinic
                                                    //算法采用深度优先搜索算法
    cf(p) = Min{cf(u, v) | (u, v) is in p}          //确定增广路径上的流增量 Δf(p) = cf(p)
       for each edge (u, v) in p
       {   f[u, v] = f[u, v] + cf(p)                //增加剩余网络中增广路径上每条边的流量
           f[v, u] = - f[u, v]                      //显然该路径上反方向上的容量为负
           cf[u, v] = c[u, v] - f[u, v]             //计算剩余网络 Gf 中的每条边的容量
           cf[v, u] = c[v, u] - f[v, u]
       }
  }
```

Edmonds-Karp 算法与 Ford-Fulkerson 算法的主要区别在于:Karp 算法采用广度优先搜索算法寻找一条从 s 到 t 最短增广路径;Dinic 算法则在层次概念的基础上采用深度优先搜索算法寻找增广路径。

4. Edmonds-Karp 算法参考模板

为便于读者尽快掌握网络流算法,下面给出一个 Edmonds-Karp 算法的参考模板。

设有 n 个顶点、m 条有向边的网图 G,源点为 1,汇点为 n。每条有向边上的容量和流

量分别用 $c[I,j]$和 $f[I,j]$表示，则 Edmonds-Karp 参考代码如下。

算法 5.33

```
#include<iostream>
#include<queue>
using namespace std;
const int maxn=205;
const int inf=0x7fffffff;
int r[maxn][maxn];                          //残留网络,初始化为原图
bool visit[maxn];
int pre[maxn];
int m,n;
bool bfs(int s,int t)                       //寻找一条从 s 到 t 的增广路径,若找到返回 true
{   int p;
    queue<int> q;
    memset(pre,-1,sizeof(pre));
    memset(visit,false,sizeof(visit));
    pre[s]=s;
    visit[s]=true;
    q.push(s);
    while(!q.empty())
    {   p=q.front();
        q.pop();
        for(int i=1;i<=n;i++)
        {   if(r[p][i]>0&&!visit[i])
            {
                pre[i]=p;
                visit[i]=true;
                if(i==t) return true;
                q.push(i);
            }
        }
    }
    return false;
}
int EdmondsKarp(int s,int t)
{ int flow=0,d,i;
  while(bfs(s,t))
  {  d=inf;
     for(i=t;i!=s;i=pre[i])
         d=d<r[pre[i]][i]? d:r[pre[i]][i];
     for(i=t;i!=s;i=pre[i])
     {  r[pre[i]][i]-=d;
        r[i][pre[i]]+=d;
     }
     flow+=d;
  }
  return flow;
}

int main()
{   while(scanf("%d%d",&m,&n)!=EOF)
    {   int u,v,w;
        memset(r,0,sizeof(r));
        for(int i=0;i<m;i++)
        {   scanf("%d%d%d",&u,&v,&w);
```

```
            r[u][v] += w;
        }
        printf("%d\n",EdmondsKarp(1,n));
    }
    return 0;
}
```


观看视频

5.7.2 网络流应用

例 5.8 草地排水(Drainage Ditches)：有一个排水系统，有 N 条排水沟，M 个水渠交叉点，每一条排水道都有单位时间水量上限。农夫的池塘在交叉点 1，小溪在交叉点 m。问单位时间内最多有多少水可以从池塘排到小溪。

有多组测试数据，每组的首行是用空格分隔的两个整数 N(0≤N≤200)和 M(2≤M≤200)，N 为排水沟数，M 是交叉点数，后续 N 行包括 3 个整数 Si、Ei 和 Ci，水流由 Si 流向 Ei(1≤Si，Ei≤M)，Ci(0≤Ci≤10 000 000)表示水流最大速率。每组测试数据输出一个排水最大速率整数。[POJ 1273]

输入示例

```
1 2 40
1 4 20
2 4 20
2 3 30
5 4
3 4 10
```

输出示例

```
50
```

参考代码如下。

算法 5.34

```
#define VMAX 201
#include<iostream>
using namespace std;
int c[VMAX][VMAX];                                  //容量
int n, m;                                           //分别表示图的边数和顶点数
int Edmonds_Karp(int s, int t)                      //输入源点和汇点
{   int p, q, queue[VMAX], u, v, pre[VMAX], flow=0, aug;
    while(true)
    {   memset(pre, -1,sizeof(pre));                //记录双亲结点
        for(queue[p=q=0]=s; p<=q; p++)              //广度优先搜索
        {   u= queue[p];
            for(v=0; v<m&&pre[t]<0; v++)
                if(c[u][v]>0 && pre[v]<0)
                    pre[v]=u, queue[++q]=v;
            if(pre[t]>=0) break;
        }
        if(pre[t]<0)    break;                      //不存在增广路径
        aug=0x7fffffff;                             //记录最小残留容量
        for(u=pre[v=t]; v!=s; v=u,u=pre[u])
            if(c[u][v]<aug)    aug=c[u][v];
        for(u=pre[v=t]; v!=s; v=u,u=pre[u])
            c[u][v]-=aug, c[v][u]+=aug;
        flow+=aug;
    }
```

```
    return flow;
}

int main()
{   int i,a,b;
    int p[201][201];
    while(scanf("%d %d",&n,&m)!=EOF&&(n||m))
    {   memset(c,0,sizeof(c));
        for(i=0;i<n;i++)
        {   scanf("%d%d",&a,&b);
            Sacnf("%d",&p[a-1][b-1]);
            c[a-1][b-1]+=p[a-1][b-1];      //两点间可能有多条路径,把权值相加
        }
        Printf("%d\n",Edmonds_Karp(0,m-1));   //源点为0,汇点为顶点数-1 (m-1)
    }
return 0;
}
```

例 5.9　GSM 手机(GSM phone)。[POJ 3549]

描述

Mr. X wants to travel from a point $A(X_a, Y_a)$ to a point $B(X_b, Y_b)$, $A \neq B$. He has a GSM mobile phone and wants to stay available during whole the trip. A local GSM operator has installed K sets of GSM equipment in points $P_i(X_i, Y_i)$, $1 \leqslant i \leqslant K$. Each set of the equipment provides circular zone Z_i. Point P_i is the center of the zone Z_i and R_i is its radius. Mobile phones can operate inside such a zone and on its border. Zones can intersect, but no zone completely includes another one.

Your task is to find the length of the shortest way from A to B which is completely covered by GSM zones. You may assume that such a way always exists. Precision of calculations has to be 0.000 01.

输入

The first line contains four floating point numbers X_a Y_a X_b Y_b, separated by one or more spaces. The next line contains single integer number K ($K \leqslant 200$). Each of the rest K lines of the file contains three floating point numbers X_i, Y_i, R_i separated by one or more spaces. $R_i > 0$.

输出

The output has to contain a single floating point number.

输入示例	输出示例
0 0 8 0 2 0 4 5 8 4 5	8.24621

参考代码如下。

算法 5.35

```
#include<stdio.h>
#include<string.h>
#include<stdlib.h>
```

```
#include<deque>
#include<algorithm>
using namespace std;
int cap[1000][1000];
int flow[1000][1000];
int a[1000];
int f;
int p[1000];
const int inf = 0x7fffffff;

void Edmonds_karp(int N, int M)
{   deque<int>q;
    int t, u, v, x;
    memset(flow, 0, sizeof(flow));
    memset(p, 0, sizeof(p));
    f = 0;
    for (; ; )
    {   memset(a, 0, sizeof(a));
        a[1] = inf;
        q.push_back(1);
        while (!q.empty())
        {   u = q.front();
            q.pop_front();
            for(v = 1; v <= M; v++)
                if (!a[v] && cap[u][v] > flow[u][v])
                {   p[v] = u;
                    q.push_back(v);
                    a[v] = min(a[u], cap[u][v] - flow[u][v]);
                }
        }
        if (a[M] == 0)
            break;
        for (u = M; u != 1; u = p[u])
        {   flow[u][p[u]] -= a[M];
            flow[p[u]][u] += a[M];
        }
        f += a[M];
    }
}

int main()
{   int N, M, i, j, a, b, c;
    while (scanf("%d%d", &N, &M) != EOF) {
        memset(cap, 0, sizeof(cap));
        f = 0;
        for (i = 0; i < N; i++) {
            scanf("%d%d%d", &a, &b, &c);
            cap[a][b] += c;
        }
    Edmonds_karp(1, M);
    printf("%d\n",f);
    }
    return 0;
}
```

习题

(1) 已知如图 5.52 所示的有向图,请给出该图的:

① 每个顶点的入/出度;

② 邻接矩阵;

③ 邻接表;

④ 逆邻接表;

⑤ 强连通分量。

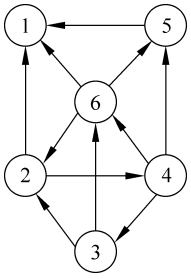


图 5.52　题(1)图

(2) 找出图 5.53 的一个拓扑排序。

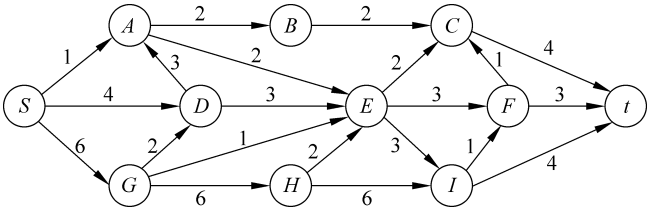


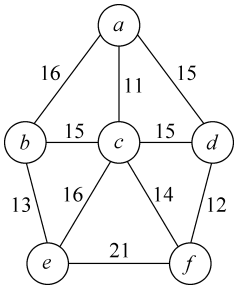
图 5.53　题(2)图

(3) 如果用一个栈代替拓扑排序中的队列,是否会得到不同的排序? 哪一种会给出"更好"的答案?

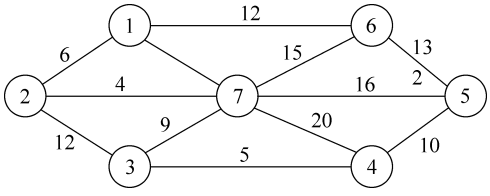
(4) 编写一个程序实现对一幅图的拓扑排序。

(5) 使用标准的二重循环,一个邻接矩阵仅初始化就需要 $O(|V|^2)$。试提出一种方法将一幅图存储在一个邻接矩阵中(使得测试一条边是否存在花费 $O(1)$),但避免二次的运行时间。

(6) 请用 Kruskal 和 Prim 两种算法分别为图 5.54(a)和图 5.54(b)构造最小生成树。



(a)



(b)

图 5.54　题(6)图

(7) 编写一个程序实现 Kruskal 算法。

(8) 编写一个程序实现 Prim 算法。

(9) 如果存在一些权值为负的边,那么 Prim 算法或 Kruskal 算法还能行得通吗?

(10) 证明 V 个顶点的图可以有 V^{V-2} 棵最小生成树。

(11) 如果一幅图的所有边权都为 1 和 $|E|$ 之间,那么能有多快算出最小生成树?

(12) 给出一个算法求解最大生成树,这比求解最小生成树更难吗?

（13）设一幅图的所有边的权都为 1 和$|E|$之间的整数，Dijkstra 算法可以多快实现？

（14）写出一个算法求解单源最短路径问题。

（15）① 解释如何修改 Dijkstra 算法以得到从 v 到 w 的不同的最小路径的个数计数。

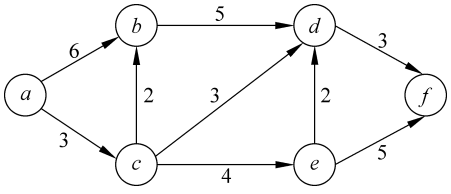


图 5.55　题(16)图

② 解释如何修改 Dijkstra 算法使得如果存在多于一条从 v 到 w 的最小路径，那么具有最少边数的路径将被选中。

（16）请用图示说明图 5.55 从顶点 a 到其余各顶点之间的最短路径。

（17）找出图 5.53 中的网络最大流。

（18）设 $G=(V,E)$是一棵树，s 是它的根，并且添加一个顶点 t 以及所有树叶到 t 的无穷容量的边。给出一个线性时间算法以找出从 s 到 t 的最大流。

（19）给出一个算法找出允许最大流通过的增长通路。

（20）写出一个求类似图 5.53 的有向图的网络最大流算法。

（21）已知 AOE 网有 9 个结点：V_1、V_2、V_3、V_4、V_5、V_6、V_7、V_8、V_9，其邻接矩阵如图 5.56 所示。

① 请画出该 AOE 图。

② 计算完成整个计划需要的时间。

③ 求出该 AOE 网的关键路径。

∞	6	4	5	∞	∞	∞	∞	∞
∞	∞	∞	∞	1	∞	∞	∞	∞
∞	∞	∞	∞	1	∞	∞	∞	∞
∞	∞	∞	∞	∞	2	∞	∞	∞
∞	∞	∞	∞	∞	∞	9	7	∞
∞	∞	∞	∞	∞	∞	∞	4	∞
∞	∞	∞	∞	∞	∞	∞	∞	2
∞	∞	∞	∞	∞	∞	∞	∞	4
∞	∞	∞	∞	∞	∞	∞	∞	∞

图 5.56　题(21)图

（22）写出一个用邻接矩阵存储图表示的关键路径算法。

（23）写出将一个无向图邻接矩阵转换成邻接表的算法。

（24）写出将一个无向图邻接表转换成邻接矩阵的算法。

（25）试以邻接矩阵为存储结构，分别写出连通图的深度优先搜索和广度优先搜索算法。

（26）写出建立一幅有向图的逆邻接表的算法。

（27）G 为一 n 个顶点的有向图，其存储结构分别为：

① 邻接矩阵。

② 邻接表。

请写出相应存储结构上的计算有向图 G 出度为 0 的顶点个数的算法。

（28）二分图 $G=(V,E)$是把 V 划分成两个子集 V_1 和 V_2 并且其边的两个顶点都不在同一个子集中的图。

① 给出一个线性算法以确定一幅图是否是二分图。

② 二分问题是找出 E 的最大子集 E'使得没有顶点含在多于一条的边中。图 5.57 中所示的是 4 条边的一个匹配(由虚线表示)。存在一个 5 条边的匹配，它是最大的匹配。

指出二分匹配问题如何能够用于解决下列问题：现有一组教师、一组课程，以及每位教师

有资格教授的课程表。如果没有教师需要教授多于一门课程，而且只有一位教师可以教授一门给定的课程，那么可以提供开设的课程的最多门数是多少？

③ 证明网络流问题可以用来解决二分匹配问题。

④ 对问题②的解法的时间复杂度如何？

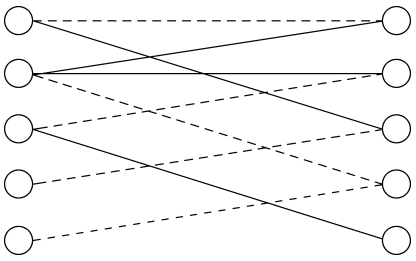


图 5.57　二分图

(29) ① 使用 Prim 和 Kruskal 两种算法求图 5.58 中图的最小生成树。

② 这棵最小生成树是唯一的吗？为什么？

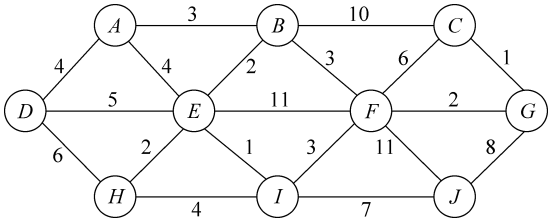


图 5.58　题(29)图

(30) 求出图 5.59 中图的所有割点。指出深度优先生成树和每个顶点的 Num 和 Low 的值。并证明寻找割点的算法的正确性。

(31) 给出一个算法以决定在一幅有向图的深度优先生成森林中的一条边(v,w)是否是树、背向边、交叉边或前向边。

(32) 找出图 5.60 中的强连通分支。

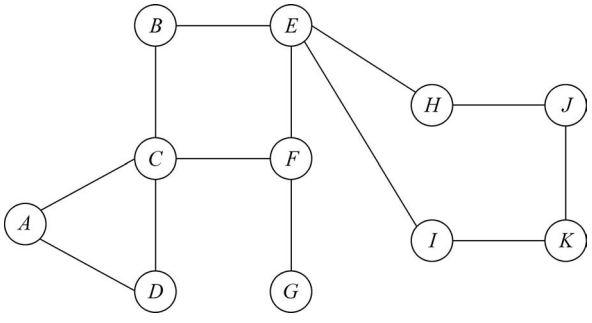


图 5.59　题(30)图

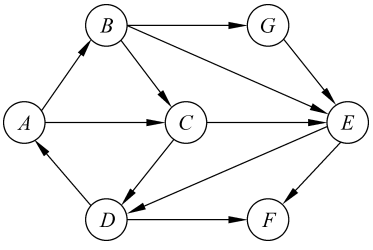


图 5.60　题(32)图

(33) 编写一个程序以找出一幅有向图的强连通分支。

ACM/ICPC 实战练习

(1) POJ 3083，ZOJ 2787，Children of the Candy Corn

(2) POJ 2251，ZOJ 1940，Dungeon Master

(3) POJ 1426，ZOJ 1530，Find The Multiple

(4) POJ 3087，ZOJ 2774，Shuffle'm Up

(5) POJ 1860,ZOJ 1544,Currency Exchange
(6) POJ 2253,ZOJ 1942,Frogger
(7) POJ 1125,ZOJ 1082,Stockbroker Grapevine
(8) POJ 2240,ZOJ 1092,Arbitrage
(9) POJ 1789,ZOJ 2158,Truck History
(10) POJ 2485,ZOJ 2048,Highways
(11) POJ 1094,ZOJ 1060,Sorting It All Out
(12) POJ 1459,ZOJ 1734,Power Network
(13) POJ 3436,ACM Computer Factory
(14) POJ 3041,ZOJ 1438,Asteroids
(15) POJ 3020,Antenna Placement
(16) POJ 1470,ZOJ 1141,Closest Common Ancestors