第7章　图

图(Graph)是一种比线性表和树更为复杂的数据结构。

线性结构：是研究数据元素之间的一对一关系。在这种结构中，除第一个和最后一个元素外，任何一个元素都有唯一的一个直接前驱和直接后继。

树结构：是研究数据元素之间的一对多的关系。在这种结构中，每个元素对下(层)可以有0个或多个元素相联系，对上(层)只有唯一的一个元素相关，数据元素之间有明显的层次关系。

图结构：是研究数据元素之间的多对多的关系。在这种结构中，任意两个元素之间可能存在关系。即结点之间的关系可以是任意的，图中任意元素之间都可能相关。

图的应用极为广泛，已渗入到诸如语言学、逻辑学、物理、化学、电讯、计算机科学以及数学的其它分支。

7.1 图的基本概念

7.1.1 图的定义和术语

一个图(G)定义为一个偶对(V,E)，记为G=(V,E)。其中：V是顶点(Vertex)的非空有限集合，记为V(G)；E是无序集V&V的一个子集，记为E(G)，其元素是图的弧(Arc)。

将顶点集合为空的图称为空图。其形式化定义为：

G=(V，E)

V={v|v∈data object}

E={<v,w>| v,w∈V∧p(v,w)}

P(v,w)表示从顶点v到顶点w有一条直接通路。

弧(Arc)：表示两个顶点v和w之间存在一个关系，用顶点偶对<v,w>表示。通常根据图的顶点偶对将图分为有向图和无向图。

有向图(Digraph)：若图G的关系集合E(G)中，顶点偶对<v,w>的v和w之间是有序的，称图G是有向图。

在有向图中，若 <v,w>∈E(G)，表示从顶点v到顶点w有一条弧。其中：v称为弧尾(tail)或始点(initial node)，w称为弧头(head)或终点(terminal node)。

无向图(Undigraph)：若图G的关系集合E(G)中，顶点偶对<v,w>的v和w之间是无序的，称图G是无向图。

在无向图中，若∀<v,w>∈E(G)，有<w,v>∈E(G)，即E(G)是对称，则用无序对(v,w) 表示v和w之间的一条边(Edge)，因此(v,w) 和(w,v)代表的是同一条边。

例1：设有有向图G1和无向图G2，形式化定义分别是：

G1=(V1，E1)

V1={a,b,c,d,e}

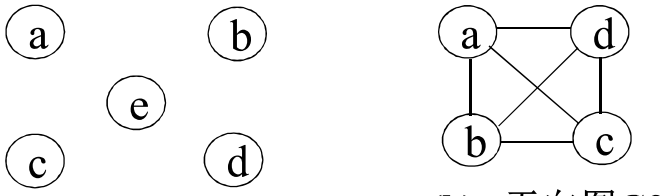
E1={<a,b>,<a,c>,
<a,e>,<c,d>,<c,e> ,<d,a>,<d,b>,<e,d>}

G2=(V2，E2)

V2={a,b,c,d}

E2={(a,b), (a,c), (a,d), (b,d), (b,c), (c,d)}

它们所对应的图如图7-1所示。



(a) 有向图**G1**　　**(b)** 无向图**G2**

图**7-1** 图的示例

完全无向图：对于无向图，若图中顶点数为**n**，用**e**表示边的数目，则**e** ∈**[0**，**n(n-1)/2]** 。具有**n(n-1)/2**条边的无向图称为完全无向图。

完全无向图另外的定义是：

对于无向图**G=(V**，**E)**，若∀v_i，v_j ∈**V**，当$v_i \neq v_j$时，有$(v_i, v_j) \in E$，即图中任意两个不同的顶点间都有一条无向边，这样的无向图称为完全无向图。

完全有向图：对于有向图，若图中顶点数为**n**，用**e**表示弧的数目，则**e∈[0，n(n-1)]**。具有**n(n-1)**条边的有向图称为完全有向图。

完全有向图另外的定义是：

对于有向图**G=(V，E)**，若$\forall v_i$，$v_j \in V$，当$v_i \neq v_j$时，有$<v_i, v_j> \in E \wedge <v_j, v_i> \in E$，即图中任意两个不同的顶点间都有一条弧，这样的有向图称为完全有向图。

有很少边或弧的图（$e<n\log n$）的图称为稀疏图，反之称为稠密图。

权(Weight)：与图的边和弧相关的数。权可以表示从一个顶点到另一个顶点的距离或耗费。

子图和生成子图：设有图G=(V，E)和G’=(V’，E’)，若V’⊂V且E’⊂E，则称图G’是G的子图；若V’=V且E’⊂E，则称图G’是G的一个生成子图。

顶点的邻接(Adjacent)：对于无向图G=(V，E)，若边(v,w)∈E，则称顶点v和w 互为邻接点，即v和w相邻接。边(v,w)依附(incident)与顶点v和w 。

对于有向图G=(V ，E)，若有向弧<v,w>∈E，则称顶点v “邻接到”顶点w，顶点w “邻接自”顶点v ，弧<v,w> 与顶点v和w “相关联” 。

顶点的度、入度、出度：对于无向图G=(V，E)，$\forall v_i \in V$，图G中依附于v_i的边的数目称为顶点v_i的度(degree)，记为$TD(v_i)$。

显然，在无向图中，所有顶点度的和是图中边的2倍。 即 $\sum TD(v_i)=2e$ $i=1, 2, \ldots, n$ ， e为图的边数。

对有向图G=(V，E)，若$\forall v_i \in V$ ，图G中以v_i作为起点的有向边(弧)的数目称为顶点v_i的出度(Outdegree)，记为$OD(v_i)$ ；以v_i作为终点的有向边(弧)的数目称为顶点v_i的入度(Indegree)，记为$ID(v_i)$ 。顶点v_i的出度与入度之和称为v_i的度，记为$TD(v_i)$ 。即

$$TD(v_i)=OD(v_i)+ID(v_i)$$

路径(Path)、路径长度、回路(Cycle) ：对无向图G=(V，E)，若从顶点v_i经过若干条边能到达v_j，称顶点v_i和v_j是连通的，又称顶点v_i到v_j有路径。

对有向图G=(V，E)，从顶点v_i到v_j有有向路径，指的是从顶点v_i经过若干条有向边(弧)能到达v_j。

或路径是图G中连接两顶点之间所经过的顶点序列。即

$Path=v_{i0}v_{i1}\ldots v_{im}$，$v_{ij}\in V$且$(v_{ij-1}, v_{ij})\in E$ $j=1,2,\ldots,m$

或

$Path=v_{i0}v_{i1}\ldots v_{im}$，$v_{ij}\in V$且$<v_{ij-1}, v_{ij}>\in E$ $j=1,2,\ldots,m$

路径上边或有向边(弧)的数目称为该路径的长度。

在一条路径中，若没有重复相同的顶点，该路径称为简单路径；第一个顶点和最后一个顶点相同的路径称为回路(环)；在一个回路中，若除第一个与最后一个顶点外，其余顶点不重复出现的回路称为简单回路(简单环)。

连通图、图的连通分量：对无向图G=(V，E)，若∀v_i，v_j∈V，v_i和v_j都是连通的，则称图G是连通图，否则称为非连通图。若G是非连通图，则极大的连通子图称为G的连通分量。

对有向图G=(V，E)，若∀v_i，v_j∈V，都有以v_i为起点，v_j为终点以及以v_j为起点，v_i为终点的有向路径，称图G是强连通图，否则称为非强连通图。若G是非强连通图，则极大的强连通子图称为G的强连通分量。

“极大”的含义：指的是对子图再增加图G中的其它顶点，子图就不再连通。

生成树、生成森林：一个连通图(无向图)的生成树是一个极小连通子图，它含有图中全部n个顶点和只有足以构成一棵树的n-1条边，称为图的生成树，如图7-2所示。

关于无向图的生成树的几个结论：

- 一棵有n个顶点的生成树有且仅有n-1条边；
- 如果一个图有n个顶点和小于n-1条边，则是非连通图；
- 如果多于n-1条边，则一定有环；
- 有n-1条边的图不一定是生成树。

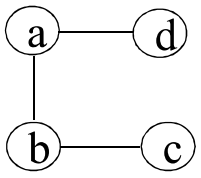
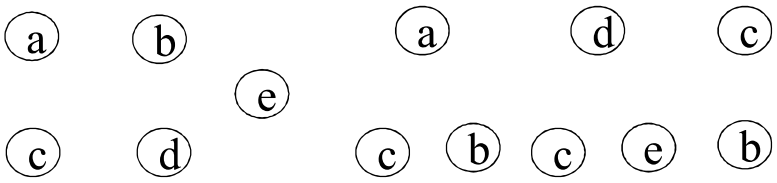


图7-2 图G2的一棵生成树

有向图的生成森林是这样一个子图，由若干棵有向树组成，含有图中全部顶点。

有向树是只有一个顶点的入度为0，其余顶点的入度均为1的有向图，如图7-3所示。

网：每个边(或弧)都附加一个权值的图，称为带权图。带权的连通图(包括弱连通的有向图)称为网或网络。网络是工程上常用的一个概念，用来表示一个工程或某种流程，如图7-4所示。



(a) 有向图　　(b) 生成森林

图7-3 有向图及其生成森林

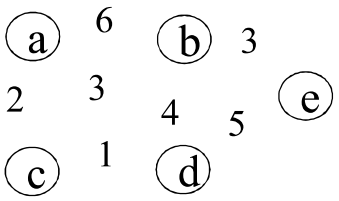


图7-4 带权有向图

7.1.2 图的抽象数据类型定义

图是一种数据结构，加上一组基本操作就构成了图的抽象数据类型。

图的抽象数据类型定义如下：

ADT Graph{

数据对象**V**：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,w∈V∧p(v,w)，**<v,w>**表示从**v**到**w**的弧，**P(v,w)**定义了弧**<v,w>**的信息 **}**

基本操作**P**：

Create_Graph()： 图的创建操作。

初始条件：无。

操作结果：生成一个没有顶点的空图**G**。

GetVex(G, v)： 求图中的顶点**v**的值。

初始条件：图**G**存在，**v**是图中的一个顶点。

操作结果：生成一个没有顶点的空图**G**。

… …

DFStraver(G,V)：从**v**出发对图**G**深度优先遍历。

初始条件：图**G**存在。

操作结果：对图**G**深度优先遍历，每个顶点访问且只访问一次。

……

BFStraver(G,V)：从**v**出发对图**G**广度优先遍历。

初始条件：图**G**存在。

操作结果：对图**G**广度优先遍历，每个顶点访问且只访问一次。

} ADT Graph

详见**p**$_{156\sim157}$。

7.2 图的存储结构

图的存储结构比较复杂，其复杂性主要表现在：

◆ 任意顶点之间可能存在联系，无法以数据元素在存储区中的物理位置来表示元素之间的关系。

◆ 图中顶点的度不一样，有的可能相差很大，若按度数最大的顶点设计结构，则会浪费很多存储单元，反之按每个顶点自己的度设计不同的结构，又会影响操作。

图的常用的存储结构有：邻接矩阵、邻接链表、十字链表、邻接多重表和边表。

7.2.1 邻接矩阵(数组)表示法

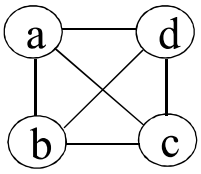
基本思想：对于有**n**个顶点的图，用一维数组**vexs[n]**存储顶点信息，用二维数组**A[n][n]**存储顶点之间关系的信息。该二维数组称为邻接矩阵。在邻接矩阵中，以顶点在**vexs**数组中的下标代表顶点，邻接矩阵中的元素**A[i][j]**存放的是顶点**i**到顶点**j**之间关系的信息。

1 无向图的数组表示

(1) 无权图的邻接矩阵

无向无权图G=(V，E)有n(n≧1)个顶点，其邻接矩阵是n阶对称方阵，如图7-5所示。其元素的定义如下：

$$A[i][j]=\begin{cases}1 & \text{若}(v_i, v_j)\in E\text{，即}v_i, v_j\text{邻接}\\ 0 & \text{若}(v_i, v_j)\notin E\text{，即}v_i, v_j\text{不邻接}\end{cases}$$



(a) 无向图

vexs
a
b
c
d

(b) 顶点矩阵

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

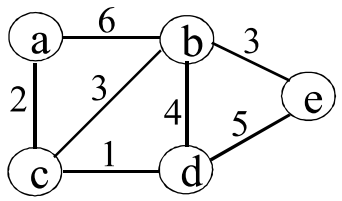
(c) 邻接矩阵

图7-5 无向无权图的数组存储

(2) 带权图的邻接矩阵

无向带权图G=(V，E) 的邻接矩阵如图7-6所示。其元素的定义如下：

$$A[i][j]=\begin{cases} W_{ij} & \text{若}(v_i, v_j)\in E\text{，即}v_i, v_j\text{邻接，权值为}w_{ij} \\ \infty & \text{若}(v_i, v_j)\notin E\text{，即}v_i, v_j\text{不邻接时} \end{cases}$$



(a) 带权无向图

vexs

a
b
c
d
e

(b) 顶点矩阵

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

(c) 邻接矩阵

图7-6 无向带权图的数组存储

(3) 无向图邻接矩阵的特性

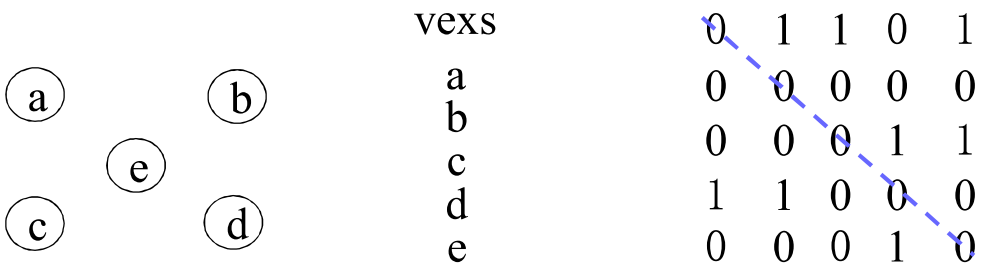
- 邻接矩阵是对称方阵；
- 对于顶点v_i，其度数是第i行的非0元素的个数；
- 无向图的边数是上(或下)三角形矩阵中非0元素个数。

2 有向图的数组表示

(1) 无权图的邻接矩阵

若有向无权图G=(V，E)有n(n≧1)个顶点，则其邻接矩阵是n阶对称方阵，如图7-7所示。元素定义如下：

$$A[i][j]=\begin{cases}1 & \text{若}<v_i, v_j>\in E\text{，从}v_i\text{到}v_j\text{有弧}\\ 0 & \text{若}<v_i, v_j>\notin E\text{ 从}v_i\text{到}v_j\text{没有弧}\end{cases}$$



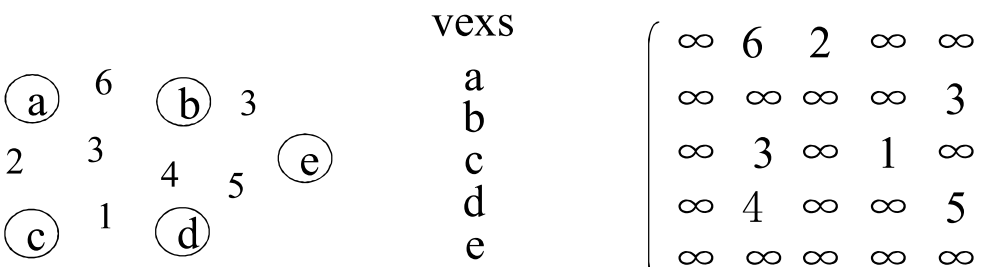
(a) 有向图　　**(b)** 顶点矩阵　　**(c)** 邻接矩阵

图7-7　有向无权图的数组存储

(2) 带权图的邻接矩阵

有向带权图G=(V，E)的邻接矩阵如图7-8所示。其元素的定义如下：

$$A[i][j]=\begin{cases} w_{ij} & \text{若}<v_i,v_j>\in E\text{，即}v_i\text{，}v_j\text{邻接，权值为}w_{ij} \\ \infty & \text{若}<v_i,v_j>\notin E\text{，即}v_i\text{，}v_j\text{不邻接时} \end{cases}$$



(a) 带权有向图　　**(b)** 顶点矩阵　　**(c)** 邻接矩阵

图**7-8** 带权有向图的数组存储

(3) 有向图邻接矩阵的特性

- 对于顶点v_i，第**i**行的非**0**元素的个数是其出度**OD(v_i)**；第**i**列的非**0**元素的个数是其入度**ID(v_i)**。
- 邻接矩阵中非**0**元素的个数就是图的弧的数目。

3 图的邻接矩阵的操作

图的邻接矩阵的实现比较容易，定义两个数组分别存储顶点信息(数据元素)和边或弧的信息(数据元素之间的关系)。其存储结构形式定义如下：

```
#define INFINITY  MAX_VAL     /* 最大值∞ */
  /* 根据图的权值类型，分别定义为最大整数或实数 */
#define MAX_VEX  30     /*  最大顶点数目 */
typedef enum {DG, AG, WDG,WAG} GraphKind ;
    /*  {有向图，无向图，带权有向图，带权无向图}  */
```

```
typedef struct ArcType
   {  VexType  vex1, vex2 ;   /*  弧或边所依附的两个顶点 */
      ArcValType  ArcVal ;    /*  弧或边的权值  */
      ArcInfoType  ArcInfo ;  /*  弧或边的其它信息   */
   }ArcType ;   /*  弧或边的结构定义  */
typedef struct
   {  GraphKind  kind ;   /*  图的种类标志   */
      int  vexnum , arcnum ;   /*  图的当前顶点数和弧数  */
      VexType   vexs[MAX_VEX] ;  /*  顶点向量  */
      AdjType   adj[MAX_VEX][MAX_VEX];
      }MGraph ;    /*  图的结构定义   */
```

利用上述定义的数据结构，可以方便地实现图的各种操作。

(1) 图的创建

```
AdjGraph  *Create_Graph(MGraph * G)
  {  printf(“请输入图的种类标志：”) ;
     scanf(“%d”, &G->kind) ;
     G->vexnum=0 ;      /*  初始化顶点个数  */
     return(G) ;
  }
```

(2) 图的顶点定位

图的顶点定位操作实际上是确定一个顶点在vexs数组中的位置(下标)，其过程完全等同于在顺序存储的线性表中查找一个数据元素。

算法实现：

```
int  LocateVex(MGraph *G , VexType *vp)
  {  int  k ;
     for (k=0 ; k<G->vexnum ; k++)
         if (G->vexs[k]==*vp)  return(k) ;
     return(-1) ;     /*  图中无此顶点  */
  }
```

(3) 向图中增加顶点

向图中增加一个顶点的操作，类似在顺序存储的线性表的末尾增加一个数据元素。

算法实现：

```
int  AddVertex(MGraph *G , VexType *vp)
  {  int  k , j ;
     if  (G->vexnum>=MAX_VEX)
        {  printf("Vertex Overflow !\n") ; return(-1) ;  }
     if  (LocateVex(G , vp)!=-1)
        {  printf("Vertex has existed !\n") ; return(-1) ; }
     k=G->vexnum ; G->vexs[G->vexnum++]=*vp ;
```

```
    if (G->kind==DG||G->kind==AG)
       for (j=0 ; j<G->vexnum ; j++)
          G->adj[j][k].ArcVal=G->adj[k][j].ArcVal=0 ;
                /*  是不带权的有向图或无向图  */
    else
       for (j=0 ; j<G->vexnum ; j++)
          {   G->adj[j][k].ArcVal=INFINITY ;
              G->adj[k][j].ArcVal=INFINITY ;
                  /*  是带权的有向图或无向图  */
          }
    return(k) ;
}
```

(4) 向图中增加一条弧

根据给定的弧或边所依附的顶点，修改邻接矩阵中所对应的数组元素。

算法实现：

```
int  AddArc(MGraph *G , ArcType *arc)
   {   int  k , j ;
       k=LocateVex(G , &arc->vex1)  ;
       j=LocateVex(G , &arc->vex1)  ;
       if (k==-1||j==-1)
          {  printf("Arc's Vertex do not existed !\n") ;
             return(-1) ;
          }
```

```
    if (G->kind==DG||G->kind==WDG)
       {  G->adj[k][j].ArcVal=arc->ArcVal;
          G->adj[k][j].ArcInfo=arc->ArcInfo ;
                /*  是有向图或带权的有向图*/
       }
    else
       {  G->adj[k][j].ArcVal=arc->ArcVal ;
          G->adj[j][k].ArcVal=arc->ArcVal ;
          G->adj[k][j].ArcInfo=arc->ArcInfo ;
          G->adj[j][k].ArcInfo=arc->ArcInfo ;
              /*  是无向图或带权的无向图,需对称赋值  */
       }
    return(1) ;
}
```

7.2.2 邻接链表法

基本思想：对图的每个顶点建立一个单链表，存储该顶点所有邻接顶点及其相关信息。每一个单链表设一个表头结点。

第i个单链表表示依附于顶点V_i的边(对有向图是以顶点V_i为头或尾的弧)。

1 结点结构与邻接链表示例

链表中的结点称为表结点，每个结点由三个域组成，如图**7-9(a)**所示。其中邻接点域**(adjvex)**指示与顶点V_i邻接的顶点在图中的位置(顶点编号)，链域**(nextarc)**指向下一个与顶点V_i邻接的表结点，数据域**(info)**存储和边或弧相关的信息，如权值等。对于无权图，如果没有与边相关的其他信息，可省略此域。

每个链表设一个表头结点(称为顶点结点)，由两个域组成，如图**7-9(b)**所示。链域**(firstarc)**指向链表中的第一个结点，数据域**(data)** 存储顶点名或其他信息。

图7-9　邻接链表结点结构

在图的邻接链表表示中，所有顶点结点用一个向量以顺序结构形式存储，可以随机访问任意顶点的链表，该向量称为表头向量，向量的下标指示顶点的序号。

用邻接链表存储图时，对无向图，其邻接链表是唯一的，如图**7-10**所示；对有向图，其邻接链表有两种形式，如图**7-11**所示。

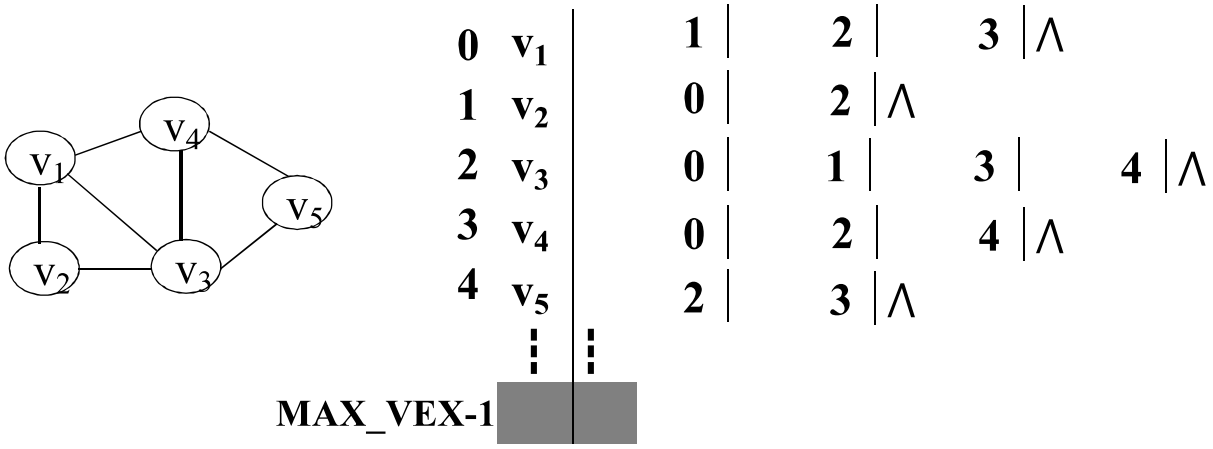
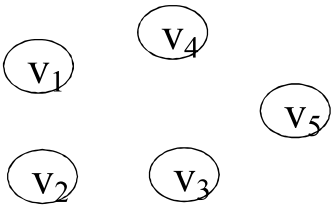


图7-10 无向图及其邻接链表



(a) 有向图

(b) 正邻接链表，出度直观

(c) 逆邻接链表，入度直观

图7-11 有向图及其邻接链表

2 邻接表法的特点

◆ 表头向量中每个分量就是一个单链表的头结点，分量个数就是图中的顶点数目；

◆ 在边或弧稀疏的条件下，用邻接表表示比用邻接矩阵表示节省存储空间；

◆ 在无向图，顶点V_i的度是第i个链表的结点数；

◆ 对有向图可以建立正邻接表或逆邻接表。正邻接表是以顶点V_i为出度(即为弧的起点)而建立的邻接表；逆邻接表是以顶点V_i为入度(即为弧的终点)而建立的邻接表；

◆ 在有向图中，第i个链表中的结点数是顶点V_i的出 (或入)度；求入 (或出)度，须遍历整个邻接表；

◆ 在邻接表上容易找出任一顶点的第一个邻接点和下一个邻接点；

3 结点及其类型定义

```
#define MAX_VEX  30     /*  最大顶点数  */
typedef int  InfoType;
typedef enum {DG, AG, WDG,WAG} GraphKind ;
typedef struct LinkNode
   {  int  adjvex ;        // 邻接点在头结点数组中的位置(下标)
      InfoType    info  ;       // 与边或弧相关的信息, 如权值
      struct LinkNode  *nextarc ;     // 指向下一个表结点
   }LinkNode ;    /*  表结点类型定义   */
```

```
typedef struct VexNode
  { VexType  data;     // 顶点信息
    int  indegree ;   // 顶点的度, 有向图是入度或出度或没有
    LinkNode  *firstarc ;    // 指向第一个表结点
  }VexNode ;     /*  顶点结点类型定义   */
typedef struct ArcType
  { VexType  vex1, vex2 ;    /*  弧或边所依附的两个顶点 */
    InfoType    info  ;       // 与边或弧相关的信息, 如权值
  }ArcType ;     /*  弧或边的结构定义  */
```

```
typedef struct
  {  GraphKind  kind ;      /*  图的种类标志  */
     int vexnum ;
     VexNode   AdjList[MAX_VEX] ;
  }ALGraph ;    /*  图的结构定义  */
```

利用上述的存储结构描述，可方便地实现图的基本操作。

(1) 图的创建

```
ALGraph *Create_Graph(ALGraph * G)
  {  printf(“请输入图的种类标志：”) ;
     scanf(“%d”, &G->kind) ;
     G->vexnum=0 ;       /*  初始化顶点个数  */
     return(G) ;
  }
```

(2) 图的顶点定位

图的顶点定位实际上是确定一个顶点在**AdjList**数组中的某个元素的**data**域内容。

算法实现：

```
int  LocateVex(ALGraph *G , VexType *vp)
   {  int  k ;
      for (k=0 ; k<G->vexnum ; k++)
         if (G->AdjList[k].data==*vp)  return(k) ;
      return(-1) ;     /*  图中无此顶点  */
   }
```

(3) 向图中增加顶点

向图中增加一个顶点的操作，在AdjList数组的末尾增加一个数据元素。

算法实现：

```
int  AddVertex(ALGraph *G , VexType *vp)
  {  int  k , j ;
     if  (G->vexnum>=MAX_VEX)
        {  printf("Vertex Overflow !\n") ;  return(-1) ; }
     if  (LocateVex(G , vp)!=-1)
        {  printf("Vertex has existed !\n") ; return(-1) ; }
     G->AdjList[G->vexnum].data=*vp ;
```

```
    G->AdjList[G->vexnum].degree=0 ;
    G->AdjList[G->vexnum].firstarc=NULL ;
    k=++G->vexnum ;
    return(k) ;
  }
```

(4) 向图中增加一条弧

根据给定的弧或边所依附的顶点，修改单链表：无向图修改两个单链表；有向图修改一个单链表。

算法实现：

```
int  AddArc(ALGraph *G , ArcType *arc)
  {  int  k , j ;
     LinkNode *p ,*q ;
```

```
k=LocateVex(G , &arc->vex1)  ;
j=LocateVex(G , &arc->vex2)  ;
if (k==-1||j==-1)
   {  printf("Arc's Vertex do not existed !\n") ;
      return(-1) ;
   }
p=(LinkNode *)malloc(sizeof(LinkNode)) ;
p->adjvex=arc->vex1 ; p->info=arc->info ;
p->nextarc=NULL ;  /*  边的起始表结点赋值  */
q=(LinkNode *)malloc(sizeof(LinkNode)) ;
q->adjvex=arc->vex2 ; q->info=arc->info ;
q->nextarc=NULL ;  /*  边的末尾表结点赋值  */
```

```
    if (G->kind==AG||G->kind==WAG)
       {  q->nextarc=G->adjlist[k].firstarc ;
          G->adjlist[k].firstarc=q ;
          p->nextarc=G->adjlist[j].firstarc ;
          G->adjlist[j].firstarc=p ;
       }  /*  是无向图,用头插入法插入到两个单链表  */
    else       /*  建立有向图的邻接链表,用头插入法  */
       {  q->nextarc=G->adjlist[k].firstarc ;
          G->adjlist[k].firstarc=q ;  /*  建立正邻接链表用 */
          //q->nextarc=G->adjlist[j].firstarc ;
          //G->adjlist[j].firstarc=q ;  /* 建立逆邻接链表用 */
       }
    return(1);
}
```

7.2.3 十字链表法

十字链表(Orthogonal List)是有向图的另一种链式存储结构，是将有向图的正邻接表和逆邻接表结合起来得到的一种链表。

在这种结构中，每条弧的弧头结点和弧尾结点都存放在链表中，并将弧结点分别组织到以弧尾结点为头(顶点)结点和以弧头结点为头(顶点)结点的链表中。这种结构的结点逻辑结构如图7-12所示。

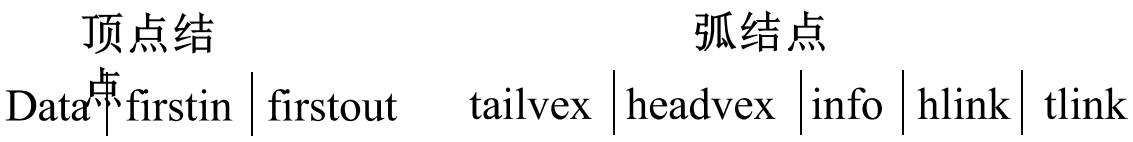


图7-12 十字链表结点结构

- **data域**：存储和顶点相关的信息；
- 指针域**firstin**：指向以该顶点为弧头的第一条弧所对应的弧结点；
- 指针域**firstout**：指向以该顶点为弧尾的第一条弧所对应的弧结点；
- 尾域**tailvex**：指示弧尾顶点在图中的位置；
- 头域**headvex**：指示弧头顶点在图中的位置；
- 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；

结点类型定义

```
#define INFINITY  MAX_VAL     /* 最大值∞ */
#define MAX_VEX  30     //  最大顶点数
typedef struct ArcNode
   {   int  tailvex , headvex ;  //  尾结点和头结点在图中的位置
      InfoType    info  ;      // 与弧相关的信息, 如权值
      struct ArcNode  *hlink , *tlink ;
   }ArcNode ;   /*  弧结点类型定义   */
typedef struct VexNode
   {  VexType  data;     // 顶点信息
      ArcNode  *firstin , *firstout ;
   }VexNode ;   /*  顶点结点类型定义   */
```

```
typedef struct
  {  int vexnum ;
     VexNode  xlist[MAX_VEX] ;
  }OLGraph ;   /*  图的类型定义   */
```

图7-13所示是一个有向图及其十字链表(略去了表结点的info域)。

从这种存储结构图可以看出，从一个顶点结点的firstout出发，沿表结点的tlink指针构成了正邻接表的链表结构，而从一个顶点结点的firstin出发，沿表结点的hlink指针构成了逆邻接表的链表结构。

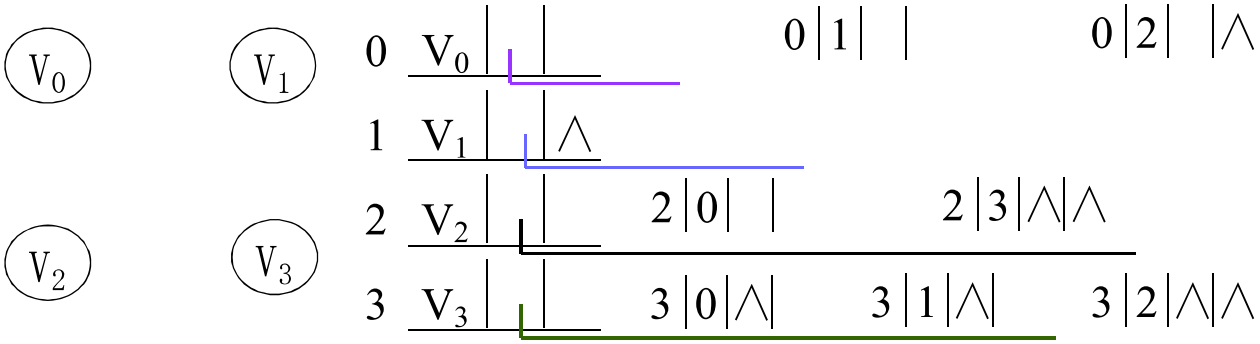


图7-13　有向图的十字链表结构

7.2.4 邻接多重表

邻接多重表(Adjacency Multilist)是无向图的另一种链式存储结构。

邻接表是无向图的一种有效的存储结构，在无向图的邻接表中，一条边(v,w)的两个表结点分别初选在以v和w为头结点的链表中，很容易求得顶点和边的信息，但在涉及到边的操作会带来不便。

邻接多重表的结构和十字链表类似，每条边用一个结点表示；邻接多重表中的顶点结点结构与邻接表中的完全相同，而表结点包括六个域如图7-14所示。

顶点结点		表结点					
data	firstedge	mark	ivex	jvex	info	ilink	jlink

图7-14 邻接多重表的结点结构

- **Data域**：存储和顶点相关的信息；
- 指针域**firstedge**：指向依附于该顶点的第一条边所对应的表结点；
- 标志域**mark**：用以标识该条边是否被访问过；
- **ivex**和**jvex**域：分别保存该边所依附的两个顶点在图中的位置；
- **info**域：保存该边的相关信息；
- 指针域**ilink**：指向下一条依附于顶点**ivex**的边；
- 指针域**jlink**：指向下一条依附于顶点**jvex**的边；

结点类型定义

```
#define INFINITY  MAX_VAL     /* 最大值∞ */
#define MAX_VEX  30     /*  最大顶点数  */
typedef  emnu {unvisited , visited}  Visitting ;
typedef struct EdgeNode
   {  Visitting  mark ;   // 访问标记
      int  ivex , jvex  ;  // 该边依附的两个结点在图中的位置
      InfoType    info  ;      // 与边相关的信息, 如权值
      struct EdgeNode  *ilink , *jlink ;
         // 分别指向依附于这两个顶点的下一条边
   }EdgeNode ;   /*  弧边结点类型定义   */
```

```
typedef struct VexNode
  { VexType  data;     // 顶点信息
    ArcNode  *firsedge ;  // 指向依附于该顶点的第一条边
  }VexNode ;   /* 顶点结点类型定义 */
typedef struct
  { int vexnum ;
    VexNode mullist[MAX_VEX] ;
}AMGraph ;
```

图7-15所示是一个无向图及其邻接多重表。

邻接多重表与邻接表的区别：

后者的同一条边用两个表结点表示，而前者只用一个表结点表示；除标志域外，邻接多重表与邻接表表达的信息是相同的，因此，操作的实现也基本相似。

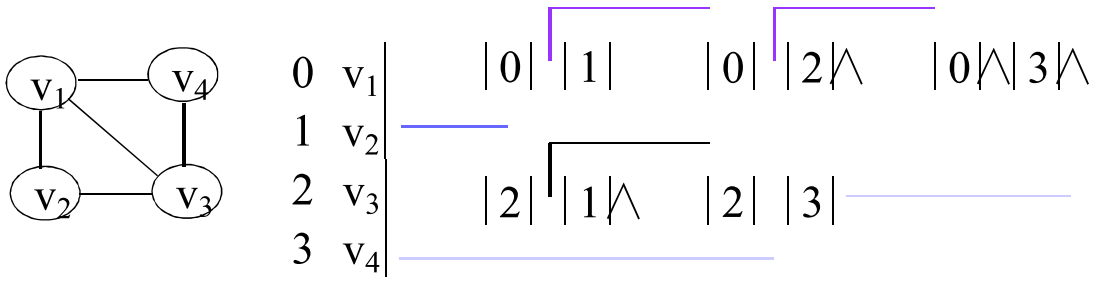


图7-15 无向图及其多重邻接链表

7.2.5 图的边表存储结构

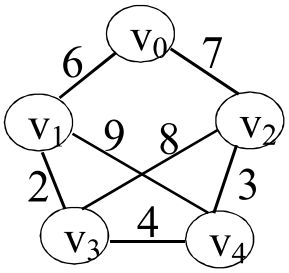
在某些应用中，有时主要考察图中各个边的权值以及所依附的两个顶点，即图的结构主要由边来表示，称为边表存储结构。

在边表结构中，边采用顺序存储，每个边元素由三部分组成：边所依附的两个顶点和边的权值；图的顶点用另一个顺序结构的顶点表存储。如图**7-16**所示。

边表存储结构的形式描述如下：

```
#define INFINITY  MAX_VAL     /* 最大值∞ */
#define MAX_VEX  30     /*  最大顶点数  */
#define MAX_EDGE  100     /*  最大边数  */
```

```
typedef struct ENode
  { int ivex , jvex ;  /*  边所依附的两个顶点 */
    WeightType  weight ;     /*  边的权值  */
  }ENode ;   /* 边表元素类型定义  */
typedef struct
  { int vexnum , edgenum ;    /*  顶点数和边数   */
    VexType vexlist[MAX_VEX] ;   /*  顶点表   */
    ENode edgelist[MAX_EDGE] ;    /*  边表   */
  }ELGraph ;
```



顶点表		边 表		
0	v_0	0	1	6
1	v_1	0	2	7
2	v_2	1	3	2
3	v_3	1	4	9
4	v_4	2	3	8
		2	4	3
		3	4	4

图**7-16** 无向图的边表表示

7.3 图的遍历

图的遍历(Travering Graph)：从图的某一顶点出发，访遍图中的其余顶点，且每个顶点仅被访问一次。图的遍历算法是各种图的操作的基础。

- 复杂性：图的任意顶点可能和其余的顶点相邻接，可能在访问了某个顶点后，沿某条路径搜索后又回到原顶点。
- 解决办法：在遍历过程中记下已被访问过的顶点。设置一个辅助向量Visited[1…n](n为顶点数)，其初值为0，一旦访问了顶点v_i后，使Visited[i]为1或为访问的次序号。

图的遍历算法有深度优先搜索算法和广度优先搜索算法。采用的数据结构是(正)邻接链表。

7.3.1 深度优先搜索算法

深度优先搜索(Depth First Search--DFS)遍历类似树的先序遍历，是树的先序遍历的推广。

1 算法思想

设初始状态时图中的所有顶点未被访问，则：

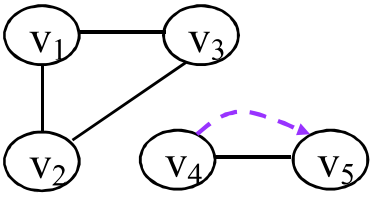
⑴：从图中某个顶点v_i出发，访问v_i；然后找到v_i的一个邻接顶点v_{i1}；

⑵：从v_{i1}出发，深度优先搜索访问和v_{i1}相邻接且未被访问的所有顶点；

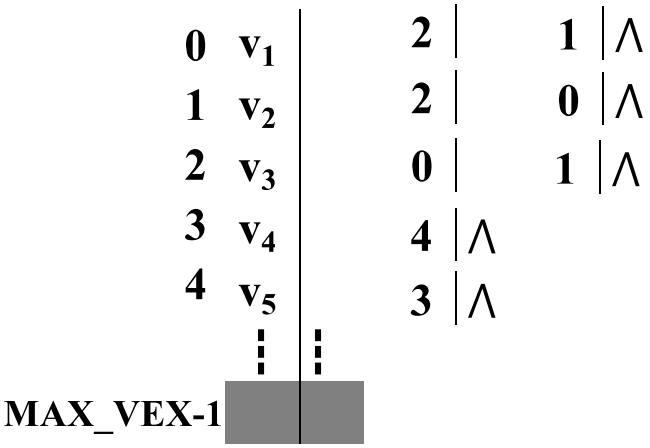
⑶：转⑴，直到和v_i相邻接的所有顶点都被访问为止

⑷：继续选取图中未被访问顶点vj作为起始顶点，转(1)，直到图中所有顶点都被访问为止。

图7-17是无向图的深度优先搜索遍历示例(红色箭头)。某种DFS次序是：$v_1 \rightarrow v_3 \rightarrow v_2 \rightarrow v_4 \rightarrow v_5$



(a) 无向图G



(b) G的邻接链表

图7-17 无向图深度优先搜索遍历

2 算法实现

由算法思想知，这是一个递归过程。因此，先设计一个从某个顶点(编号)为v_0开始深度优先搜索的函数，便于调用。

在遍历整个图时，可以对图中的每一个未访问的顶点执行所定义的函数。

```
typedef  emnu {FALSE , TRUE} BOOLEAN ;
BOOLEAN  Visited[MAX_VEX] ;
```

```
void  DFS(ALGraph *G , int v)
   {  LinkNode  *p ;
      Visited[v]=TRUE ;
      Visit[v] ;       /*  置访问标志，访问顶点v  */
      p=G->AdjList[v].firstarc;   /*  链表的第一个结点  */
      while (p!=NULL)
         {  if  (!Visited[p->adjvex]) DFS(G, p->adjvex) ;
               /*  从v的未访问过的邻接顶点出发深度优先搜索   */
            p=p->nextarc ;
         }
   }
```

```
void DFS_traverse_Grapg(ALGraph *G)
  {  int v ;
     for (v=0 ; v<G->vexnum ; v++)
        Visited[v]=FALSE ;    /*  访问标志初始化  */
     p=G->AdjList[v].firstarc ;
     for (v=0 ; v<G->vexnum ; v++)
        if (!Visited[v])   DFS(G , v);
  }
```

3 算法分析

遍历时，对图的每个顶点至多调用一次**DFS**函数。其实质就是对每个顶点查找邻接顶点的过程，取决于存储结构。当图有**e**条边，其时间复杂度为**O(e)**，总时间复杂度为**O(n+e)** 。

7.3.2 广度优先搜索算法

广度优先搜索(Breadth First Search--BFS)遍历类似树的按层次遍历的过程。

1 算法思想

设初始状态时图中的所有顶点未被访问，则：

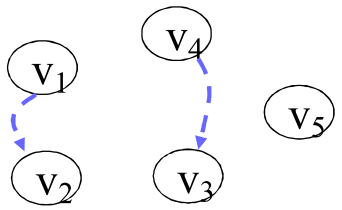
⑴：从图中某个顶点v_i出发，访问v_i；

⑵：访问v_i的所有相邻接且未被访问的所有顶点v_{i1}，v_{i2}，…，v_{im}；

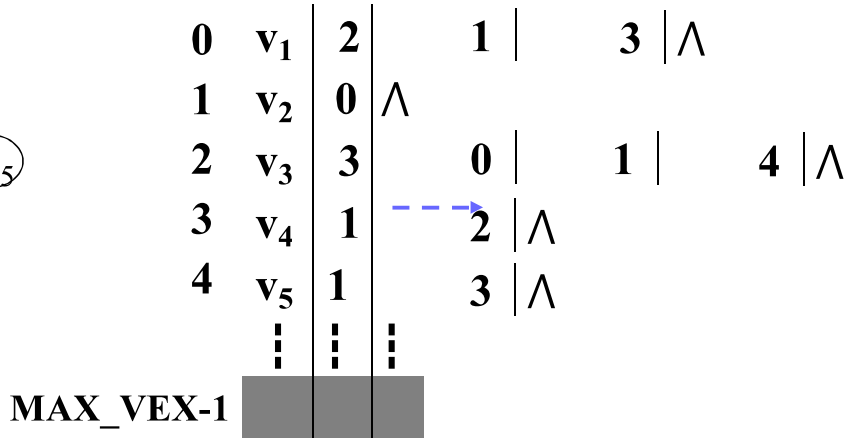
⑶：以v_{i1}，v_{i2}，…，v_{im}的次序，以$v_{ij}(1 \leqq j \leqq m)$依此作为$v_i$，转⑴；

(4)：继续选取图中未被访问顶点v_k作为起始顶点，转(1)，直到图中所有顶点都被访问为止。

图7-18是有向图的广度优先搜索遍历示例(红色箭头)。上述图的BFS次序是：$v_1 \rightarrow v_2 \rightarrow v_4 \rightarrow v_3 \rightarrow v_5$



(a) 有向图G'



(b) G'的正邻接链表

图7-18 有向图广度优先搜索遍历

2 算法实现

为了标记图中顶点是否被访问过，同样需要一个访问标记数组；其次，为了依此访问与v_i相邻接的各个顶点，需要附加一个队列来保存访问v_i的相邻接的顶点。

```
typedef  emnu {FALSE , TRUE} BOOLEAN ;
BOOLEAN  Visited[MAX_VEX] ;
typedef struct Queue
   {  int   elem[MAX_VEX] ;
      int  front , rear ;
   }Queue ;    /*  定义一个队列保存将要访问顶点  */
```

```
void BFS_traverse_Grapg(ALGraph *G)
   {  int k ,v , w ;
      LinkNode  *p ; Queue  *Q ;
      Q=(Queue *)malloc(sizeof(Queue)) ;
      Q->front=Q->rear=0 ;   /*  建立空队列并初始化  */
      for (k=0 ; k<G->vexnum ; k++)
         Visited[k]=FALSE ;     /*  访问标志初始化  */
      for (k=0 ; k<G->vexnum ; k++)
         {  v=G->AdjList[k].data ;    /*  单链表的头顶点  */
            if (!Visited[v])    /*   v尚未访问   */
               {  Q->elem[++Q->rear]=v ;   /*   v入对   */
                  while (Q->front!=Q->rear)
```

```
            {  w=Q->elem[++Q->front] ;
               Visited[w]=TRUE ;    /* 置访问标志 */
               Visit(w) ;    /* 访问队首元素 */
               p=G->AdjList[w].firstarc ;
               while (p!=NULL)
                  {  if (!Visited[p->adjvex])
                        Q->elem[++Q->rear]=p->adjvex ;
                     p=p->nextarc ;
                     }
               }  /* end while */
         }   /* end if */
      } /* end for */
}
```

用广度优先搜索算法遍历图与深度优先搜索算法遍历图的唯一区别是邻接点搜索次序不同，因此，广度优先搜索算法遍历图的总时间复杂度为**O(n+e)**。

图的遍历可以系统地访问图中的每个顶点，因此，图的遍历算法是图的最基本、最重要的算法，许多有关图的操作都是在图的遍历基础之上加以变化来实现的。

7.4 图的连通性问题

本节所讨论的内容是图的遍历算法的具体应用。

7.4.1 无向图的连通分量与生成树

1 无向图的连通分量和生成树

对于无向图，对其进行遍历时：

- 若是连通图：仅需从图中任一顶点出发，就能访问图中的所有顶点；
- 若是非连通图：需从图中多个顶点出发。每次从一个新顶点出发所访问的顶点集序列恰好是各个连通分量的顶点集；

如图**7-19**所示的无向图是非连通图，按图中给定的邻接表进行深度优先搜索遍历，**2**次调用**DFS**所得到的顶点访问序列集是：**{ v1 ,v3 ,v2}**和**{ v4 ,v5 }**

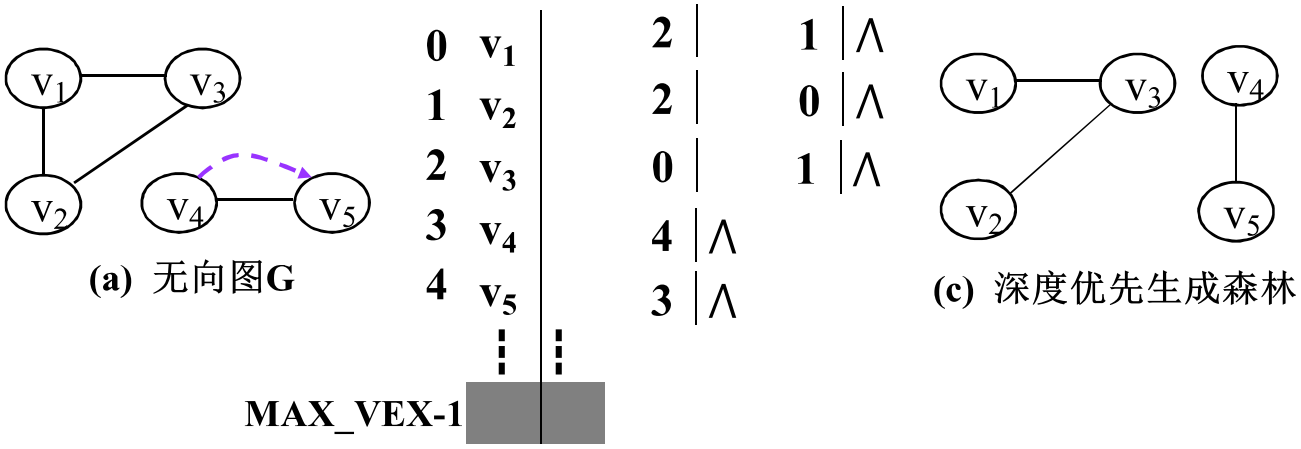


图**7-19** 无向图及深度优先生成森林

⑴ 若**G=(V,E)**是无向连通图， 顶点集和边集分别是**V(G)**，**E(G)**。若从**G**中任意点出发遍历时， **E(G)**被分成两个互不相交的集合：

T(G)：遍历过程中所经过的边的集合；

B(G)：遍历过程中未经过的边的集合；

显然： **E(G)=T(G)∪B(G)**，**T(G)∩B(G)=Ø**

显然，图**G'=(V, T(G))**是**G**的极小连通子图，且**G'**是一棵树。**G'**称为图**G**的一棵生成树。

从任意点出发按**DFS**算法得到生成树**G'**称为深度优先生成树；按**BFS**算法得到的**G'**称为广度优先生成树。

⑵ 若**G=(V,E)**是无向非连通图，对图进行遍历时得到若干个连通分量的顶点集：$V_1(G)$,$V_2(G)$,…,$V_n(G)$和相应所经过的边集：$T_1(G)$,$T_2(G)$, …,$T_n(G)$ 。

则对应的顶点集和边集的二元组：$G_i=(V_i(G),T_i(G))$ **(1≦i≦n)**是对应分量的生成树，所有这些生成树构成了原来非连通图的生成森林。

说明：当给定无向图要求画出其对应的生成树或生成森林时，必须先给出相应的邻接表，然后才能根据邻接表画出其对应的生成树或生成森林。

2 图的生成树和生成森林算法

对图的深度优先搜索遍历DFS(或BFS)算法稍作修改，就可得到构造图的DFS生成树算法。

在算法中，树的存储结构采用孩子—兄弟表示法。首先建立从某个顶点V出发，建立一个树结点，然后再分别以V的邻接点为起始点，建立相应的子生成树，并将其作为V 结点的子树链接到V结点上。显然，算法是一个递归算法。

算法实现：

(1) DFStree算法

```
typedef  struct  CSNode
   {  ElemType  data ;
      struct  CSNode *firstchild , *nextsibling ;
   }CSNode ;
CSNode  *DFStree(ALGraph *G , int v)
   {  CSNode  *T , *ptr , *q ;
      LinkNode  *p ;  int w ;
      Visited[v]=TRUE ;
      T=(CSNode *)malloc(sizeof(CSNode)) ;
      T->data=G->AdjList[v].data ;
      T->firstchild=T->nextsibling=NULL ; //   建立根结点
```

```
    q=NULL ; p=G->AdjList[v].firstarc ;
    while (p!=NULL)
       {  w=p->adjvex ;
          if  (!Visited[w])
             {  ptr=DFStree(G,w) ;      /*  子树根结点  */
                if  (q==NULL)  T->firstchild=ptr ;
                else  q->nextsibling=ptr ;
               q=ptr ;
             }
          p=p->nextarc ;
       }
    return(T) ;
}
```

(2) BFStree算法

```
typedef struct Queue
   {  int  elem[MAX_VEX] ;
      int front , rear ;
   }Queue ;    /*  定义一个队列保存将要访问顶点  */
CSNode  *BFStree(ALGraph *G ,int v)
  {  CSNode  *T , *ptr , *q ;
     LinkNode  *p ; Queue  *Q ;
     int w , k ;
     Q=(Queue *)malloc(sizeof(Queue)) ;
     Q->front=Q->rear=0 ;  /*建立空队列并初始化*/
     Visited[v]=TRUE ;
```

```
T=(CSNode *)malloc(sizeof(CSNode)) ;
T->data=G->AdjList[v].data ;
T->firstchild=T->nextsibling=NULL ; // 建立根结点
Q->elem[++Q->rear]=v ;   /*   v入队   */
while (Q->front!=Q->rear)
   {  w=Q->elem[++Q->front] ;   q=NULL ;
      p=G->AdjList[w].firstarc ;
      while (p!=NULL)
         {   k=p->adjvex ;
            if  (!Visited[k])
               {  Visited[k]=TRUE ;
```

```
                ptr=(CSNode *)malloc(sizeof(CSNode)) ;
                ptr->data=G->AdjList[k].data ;
                ptr->firstchild=T->nextsibling=NULL ;
                if  (q==NULL)  T->firstchild=ptr ;
                else  q->nextsibling=ptr ;
                q=ptr ;
                Q->elem[++Q->rear]=k ;  /*   k入对   */
            }    /*   end  if   */
          p=p->nextarc ;
        }     /*  end  while  p  */
      }    /*  end whil  Q  */
    return(T) ;
} /*求图G广度优先生成树算法BFStree*/
```

(3) 图的生成森林算法

```
CSNode  *DFSForest(ALGraph *G)
   {  CSNode  *T , *ptr , *q ;  int w ;
      for (w=0; w<G->vexnum; w++) Visited[w]=FALSE;
      T=NULL ;
      for (w=0 ; w<G->vexnum ; w++)
         if (!Visited[w])
            {  ptr=DFStree(G, w) ;
               if  (T==NULL)  T=ptr ;
               else  q->nextsibling=ptr ;
               q=ptr ;   }
      return(T) ;
   }
```

7.4.2 有向图的强连通分量

对于有向图，在其每一个强连通分量中，任何两个顶点都是可达的。 $\forall V \in G$，与V可相互到达的所有顶点就是包含V的强连通分量的所有顶点。

设从V可到达 (以V为起点的所有有向路径的终点) 的顶点集合为$T_1(G)$，而到达V (以V为终点的所有有向路径的起点)的顶点集合为$T_2(G)$，则包含V的强连通分量的顶点集合是： $T_1(G) \cap T_2(G)$ 。

求有向图G的强连通分量的基本步骤是：

⑴ 对G进行深度优先遍历，生成G的深度优先生成森林T。

⑵ 对森林T的顶点按中序遍历顺序进行编号。

⑶ 改变**G**中每一条弧的方向，构成一个新的有向图**G’**。

⑷ 按⑵中标出的顶点编号，从编号最大的顶点开始对**G’**进行深度优先搜索，得到一棵深度优先生成树。若一次完整的搜索过程没有遍历**G’**的所有顶点，则从未访问的顶点中选择一个编号最大的顶点，由它开始再进行深度优先搜索，并得到另一棵深度优先生成树。在该步骤中，每一次深度优先搜索所得到的生成树中的顶点就是**G**的一个强连通分量的所有顶点。

⑸ 重复步骤⑷，直到**G’**中的所有顶点都被访问。

如图**7-20(a)**是求一棵有向树的强连通分量过程。

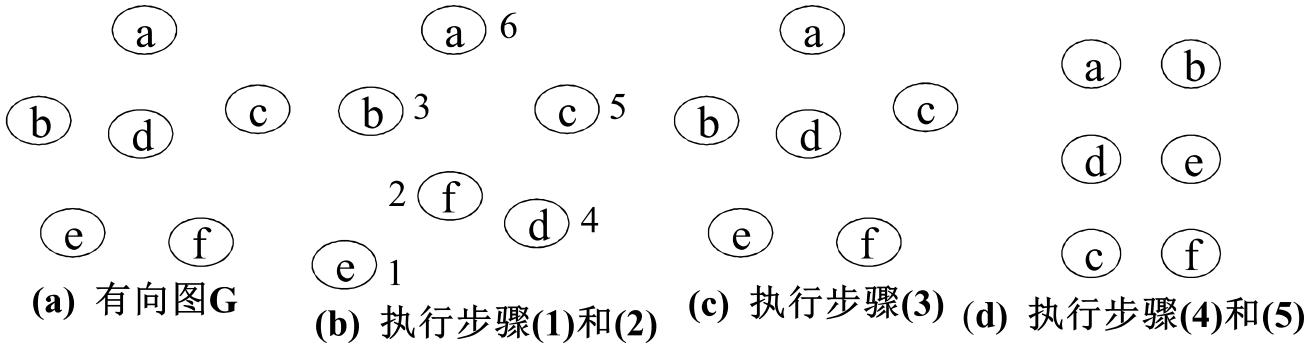


图7-20 利用深度优先搜索求有向图的强连通分量

在算法实现时，建立一个数组**in_order[n]**存放深度优先生成森林的中序遍历序列。对每个顶点**v**，在调用**DFS**函数结束时，将顶点依次存放在数组**in_order[n]**中。图采用十字链表作为存储结构最合适。

算法实现：

```
int in_order[MAX_VEX] ;
```

```
void  DFS(OLGraph *G , int v)  // 按弧的正向搜索
   { ArcNode  *p ;
      Count=0 ;
      Visited[v]=TRUE ;
      for  (p=G->xlist[v].firstout ; p!=NULL ; p=p->tlink)
         if  (!Visited[p->headvex])
            DFS(G , p->headvex) ;
      in_order[count++]=v ;
   }
```

```
void  Rev_DFS(OLGraph *G , int v)
   { ArcNode  *p ;
      Visited[v]=TRUE ;
      printf("%d" , v) ;     /* 输出顶点 */
      for  (p=G->xlist[v].firstin ; p!=NULL ; p=p->hlink)
         if  (!Visited[p->tailvex])
            Rev_DFS(G , p->tailvex) ;
   }   /* 对图G按弧的逆向进行搜索 */

void  Connected_DG(OLGraph *G)
   {  int  k=1, v, j ;
      for (v=0; v<G->vexnum; v++)
         Visited[v]=FALSE ;
```

```
    for (v=0; v<G->vexnum; v++)    /* 对图G正向遍历 */
        if (!Visited[v])  DFS(G,v) ;
    for (v=0; v<G->vexnum; v++)
        Visited[v]=FALSE ;
    for (j=G->vexnum-1; j>=0; j--)    /* 对图G逆向遍历 */
        {  v=in_order[j] ;
           if (!Visited[v])
              {  printf("\n第%d个连通分量顶点: ", k++) ;
                 Rev_DFS(G, v) ;
              }
        }
}
```

7.5 最小生成树

如果连通图是一个带权图，则其生成树中的边也带权，生成树中所有边的权值之和称为生成树的代价。

最小生成树(Minimum Spanning Tree)：带权连通图中代价最小的生成树称为最小生成树。

最小生成树在实际中具有重要用途，如设计通信网。设图的顶点表示城市，边表示两个城市之间的通信线路，边的权值表示建造通信线路的费用。n个城市之间最多可以建n×(n-1)/2条线路，如何选择其中的n-1条，使总的建造费用最低?

构造最小生成树的算法有许多，基本原则是：

构造最小生成树的算法有许多，基本原则是：

- 尽可能选取权值最小的边，但不能构成回路；
- 选择n-1条边构成最小生成树。

以上的基本原则是基于MST的如下性质：

设G=(V，E)是一个带权连通图，U是顶点集V的一个非空子集。若u∈U，v∈V-U，且(u, v)是U中顶点到V-U中顶点之间权值最小的边，则必存在一棵包含边(u, v)的最小生成树。

证明：用反证法证明。

设图**G**的任何一棵最小生成树都不包含边**(u,v)**。设**T**是**G**的一棵生成树，则**T**是连通的，从**u**到**v**必有一条路径**(u,…,v)**，当将边**(u,v)**加入到**T**中时就构成了回路。则路径**(u, …,v)**中必有一条边**(u',v')**，满足**u'∈U**，**v'∈V-U**。删去边**(u',v')** 便可消除回路，同时得到另一棵生成树**T'**。

由于**(u,v)**是**U**中顶点到**V-U**中顶点之间权值最小的边，故**(u,v)**的权值不会高于**(u',v')**的权值，**T'**的代价也不会高于**T**， **T'**是包含**(u,v)** 的一棵最小生成树，与假设矛盾。

7.5.1 普里姆(Prim)算法

从连通网N=(U，E)中找最小生成树T=(U，TE) 。

1 算法思想

(1) 若从顶点v_0出发构造，U={v_0}，TE={}；

(2) 先找权值最小的边(u，v)，其中u∈U且v∈V-U，并且子图不构成环，则U= U∪{v}，TE=TE∪{(u，v)} ；

(3) 重复(2)，直到U=V为止。则TE中必有n-1条边，T=(U，TE)就是最小生成树。

如图7-21所提示。

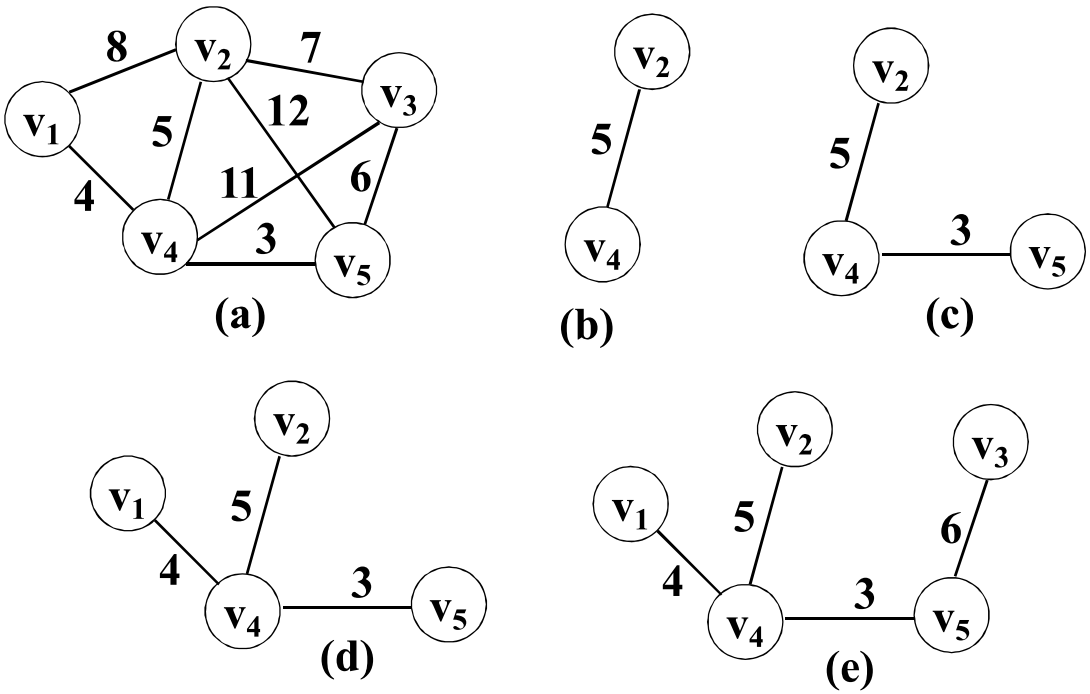


图7-21 按prime算法从v2出发构造最小生成树的过程

2 算法实现说明

设用邻接矩阵(二维数组)表示图，两个顶点之间不存在边的权值为机内允许的最大值。

为便于算法实现，设置一个一维数组**closedge[n]**，用来保存**V- U**中各顶点到**U**中顶点具有权值最小的边。数组元素的类型定义是：

```
struct
  {  int  adjvex ;     /*  边所依附于U中的顶点  */
     int  lowcost ;    /*  该边的权值  */
  }closedge[MAX_EDGE] ;
```

例如： **closedge[j].adjvex=k**，表明边(v_j, v_k)是**V-U**中顶点v_j到**U**中权值最小的边，而顶点v_k是该边所依附的**U**中的顶点。 **closedge[j].lowcost**存放该边的权值。

假设从顶点v_s开始构造最小生成树。初始时令：

Closedge[s].lowcost=0：表明顶点v_s首先加入到**U**中；

Closedge[k].adjvex=s，**Closedge[k].lowcost=cost(k, s)**

表示**V-U**中的各顶点到**U**中权值最小的边**(k≠s)**，**cost(k, s)**表示边(v_k, v_s) 权值。

3 算法步骤

⑴ 从closedge中选择一条权值(不为0)最小的边(v_k, v_j)，然后做：

① 置closedge[k].lowcost为0，表示v_k已加入到U中。

② 根据新加入v_k的更新closedge中每个元素：

$\forall v_i \in V-U$，若cost(i, k)≦colsedge[i].lowcost，表明在U中新加入顶点v_k后，(v_i, v_k)成为v_i到U中权值最小的边，置：

Closedge[i].lowcost=cost(i, k)

Closedge[i].adjvex=k

⑵ 重复⑴n-1次就得到最小生成树。

如表7-1所提示。

在Prime算法中，图采用邻接矩阵存储，所构造的最小生成树用一维数组存储其n-1条边，每条边的存储结构描述：

```
typedef struct MSTEdge
  { int  vex1, vex2 ;   /* 边所依附的图中两个顶点 */
    WeightType  weight ;    /* 边的权值 */
  }MSTEdge ;
```

算法实现

```
#define INFINITY  MAX_VAL    /* 最大值 */
MSTEdge *Prim_MST(AdjGraph *G , int u)
   /*  从第u个顶点开始构造图G的最小生成树  */
  { MSTEdge TE[] ;  // 存放最小生成树n-1条边的数组指针
```

```
int j , k , v , min ;
for (j=0; j<G->vexnum; j++)
   {  closedge[j].adjvex=u  ;
      closedge[j].lowcost=G->adj[j][u]  ;
   }   /*  初始化数组closedge[n]  */
closedge[u].lowcost=0 ;     /*  初始时置U={u}  */
TE=(MSTEdge *)malloc((G->vexnum-
1)*sizeof(MSTEdge)) ;
for (j=0; j<G->vexnum-1; j++)
   { min= INFINITY ;
      for (v=0; v<G->vexnum; v++)
         if  (closedge[v].lowcost!=0&&
      closedge[v].Lowcost<min)
```

```
            {  min=closedge[v].lowcost ; k=v ;  }
        TE[j].vex1=closedge[k].adjvex ;
        TE[j].vex2=k ;
        TE[j].weight=closedge[k].lowcost ;
        closedge[k].lowcost=0 ;     /*  将顶点k并入U中  */
        for (v=0; v<G->vexnum; v++)
          if (G->adj[v][k]<closedge[v]. lowcost)
            {  closedge[v].lowcost= G->adj[v][k] ;
               closedge[v].adjvex=k ;
            } /*  修改数组closedge[n]的各个元素的值  */
      }
    return(TE) ;
} /*  求最小生成树的Prime算法  */
```

表7-1 构造过程中辅组数组closedge中各分量的值的变化情况

i / closedge	0	1	2	3	4	U	V-U	K
adjvex	v_2		v_2	v_2	v_2	$\{v_2\}$	$\{v_1, v_3, v_4, v_5\}$	3
lwcost	8		7	5	12			
adjvex	v_4		v_2	v_2	v_4	$\{v_2, v_4\}$	$\{v_1, v_3, v_5\}$	4
lwcost	4		7	0	3			
adjvex	v_4		v_5	v_2	v_4	$\{v_2, v_4, v_5\}$	$\{v_1, v_3\}$	0
lwcost	4		6	0	0			
adjvex	v_4		v_5	v_2	v_4	$\{v_2, v_4, v_5, v_1\}$	$\{v_3\}$	2
lwcost	0		6	0	0			
adjvex	v_4		v_5	v_2	v_4	$\{v_2, v_4, v_5, v_1, v_3\}$	$\{\}$	
lwcost	0		0	0	0			

算法分析：设带权连通图有**n**个顶点，则算法的主要执行是二重循环： 求**closedge**中权值最小的边，频度为**n-1**； 修改**closedge**数组，频度为**n** 。因此，整个算法的时间复杂度是$O(n^2)$，与边的数目无关。

7.5.2 克鲁斯卡尔(Kruskal)算法

1 算法思想

设G=(V, E)是具有n个顶点的连通网，T=(U, TE)是其最小生成树。初值：U=V，TE={}。

对G中的边按权值大小从小到大依次选取。

(1) 选取权值最小的边(v_i，v_j)，若边(v_i，v_j)加入到TE后形成回路，则舍弃该边(边(v_i，v_j)；否则，将该边并入到TE中，即TE=TE∪{(v_i，v_j)}。

(2) 重复(1)，直到TE中包含有n-1条边为止。

如图7-22所提示。

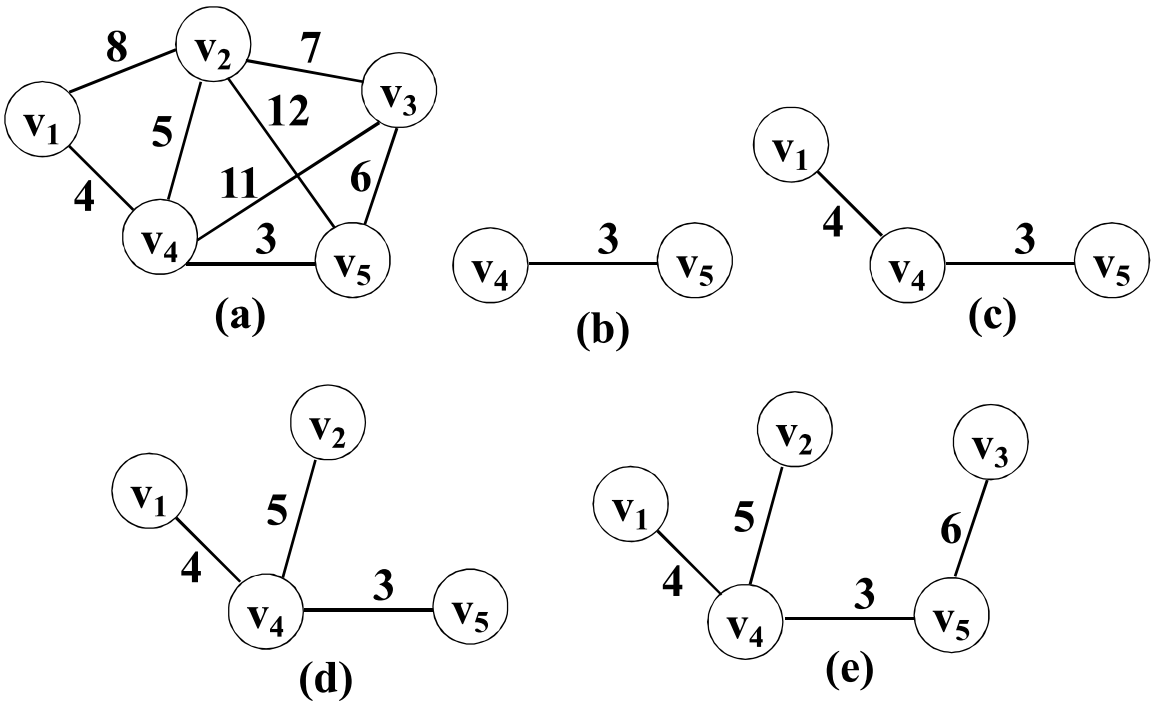


图7-22　按kruskal算法构造最小生成树的过程

2 算法实现说明

Kruskal算法实现的关键是：当一条边加入到**TE**的集合后，如何判断是否构成回路?

简单的解决方法是：定义一个一维数组**Vset[n]**，存放图**T**中每个顶点所在的连通分量的编号。

- 初值：**Vset[i]=i**，表示每个顶点各自组成一个连通分量，连通分量的编号简单地使用顶点在图中的位置（编号）。
- 当往**T**中增加一条边**(v_i，v_j)** 时，先检查**Vset[i]**和**Vset[j]**值：
  - ☆ 若**Vset[i]=Vset[j]**：表明v_i和v_j处在同一个连通分量中，加入此边会形成回路；

☆ 若Vset[i]≠Vset[j]，则加入此边不会形成回路，将此边加入到生成树的边集中。

◆ 加入一条新边后，将两个不同的连通分量合并：将一个连通分量的编号换成另一个连通分量的编号。

算法实现

```
MSTEdge *Kruskal_MST(ELGraph *G)
    /*  用Kruskal算法构造图G的最小生成树  */
  {  MSTEdge TE[] ;
     int  j, k, v, s1, s2, Vset[] ;
     WeightType  w ;
     Vset=(int  *)malloc(G->vexnum*sizeof(int)) ;
```

```
    for (j=0; j<G->vexnum; j++)
       Vset[j]=j  ;     /*  初始化数组Vset[n]  */
    sort(G->edgelist) ;  /*  对表按权值从小到大排序  */
    j=0 ; k=0 ;
    while (k<G->vexnum-1&&j< G->edgenum)
       {  s1=Vset[G->edgelist[j].vex1] ;
          s2=Vset[G->edgelist[j].vex2] ;
/*  若边的两个顶点的连通分量编号不同, 边加入到TE中  */
          if  (s1!=s2)
             {  TE[k].vex1=G->edgelist[j].vex1 ;
                TE[k].vex2=G->edgelist[j].vex2 ;
                TE[k].weight=G->edgelist[j].weight ;
```

```
                k++ ;
                for (v=0; v<G->vexnum; v++)
                    if  (Vset[v]==s2)  Vset[v]=s1 ;
              }
           j++ ;
       }
    free(Vset) ;
    return(TE) ;
}    /*  求最小生成树的Kruskal算法  */
```

算法分析：设带权连通图有**n**个顶点，**e**条边，则算法的主要执行是：

- **Vset**数组初始化：时间复杂度是**O(n)**；
- 边表按权值排序：若采用堆排序或快速排序，时间复杂度是**O(e**log**e)**；
- **while**循环：最大执行频度是**O(n)**，其中包含修改**Vset**数组，共执行**n-1**次，时间复杂度是$O(n^2)$；

整个算法的时间复杂度是$O(e\log e+n^2)$。

7.6 有向无环图及其应用

有向无环图(Directed Acycling Graph)：是图中没有回路(环)的有向图。是一类具有代表性的图，主要用于研究工程项目的工序问题、工程时间进度问题等。

一个工程(project)都可分为若干个称为活动(active)的子工程(或工序)，各个子工程受到一定的条件约束：某个子工程必须开始于另一个子工程完成之后；整个工程有一个开始点(起点)和一个终点。人们关心：

- 工程能否顺利完成?影响工程的关键活动是什么?
- 估算整个工程完成所必须的最短时间是多少?

对工程的活动加以抽象：图中顶点表示活动，有向边表示活动之间的优先关系，这样的有向图称为顶点表示活动的网(Activity On Vertex Network，AOV网)。

7.6.1 拓扑排序

1 定义

拓扑排序(Topological Sort)：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。

- 集合上的关系：集合A上的关系是从A到A的关系(A×A)。
- 关系的自反性：若∀a∈A有(a，a)∈R，称集合A上的关系R是自反的。
- 关系的对称性：如果对于a，b∈A，只要有(a，b)∈R就有(b，a)∈R，称集合A上的关系R是对称的。

◆ 关系的对称性与反对称性：如果对于**a**，b∈**A**，只要有**(a，b)∈R**就有**(b，a)∈R**，称集合**A**上的关系**R**是对称的。如果对于**a**，b∈**A**，仅当a=b时有**(a，b)∈R**和**(b，a)∈R**，称集合**A**上的关系**R**是反对称的。

◆ 关系的传递性：若**a，b，c∈A**，若**(a，b)∈R**，并且**(b，c)∈R**，则**(a，c)∈R**，称集合**A**上的关系**R**是传递的。

◆ 偏序：若集合**A**上的关系**R**是自反的，反对称的和传递的，则称**R**是集合**A**上的偏序关系。

◆ 全序：设**R**是集合**A**上的偏序关系，∀a，**b∈A**，必有**aRb**或**bRa**，则称**R**是集合**A**上的全序关系。

即偏序是指集合中仅有部分元素之间可以比较，而全序是指集合中任意两个元素之间都可以比较。

在**AOV**网中，若有有向边**<i, j>**，则**i**是**j**的直接前驱，**j**是**i**的直接后继；推而广之，若从顶点**i**到顶点**j**有有向路径，则**i**是**j**的前驱，**j**是**i**的后继。

在**AOV**网中，不能有环，否则，某项活动能否进行是以自身的完成作为前提条件。

检查方法：对有向图的顶点进行拓扑排序，若所有顶点都在其拓扑有序序列中，则无环。

有向图的拓扑排序：构造**AOV**网中顶点的一个拓扑线性序列$(v'_1, v'_2, \cdots, v'_n)$，使得该线性序列不仅保持原来有向图中顶点之间的优先关系，而且对原图中没有优先关系的顶点之间也建立一种(人为的)优先关系。

手工实现

如图7-23是一个有向图的拓扑排序过程，其拓扑序列是：　$(v_1,v_6,v_4,v_3,v_2,v_5)$

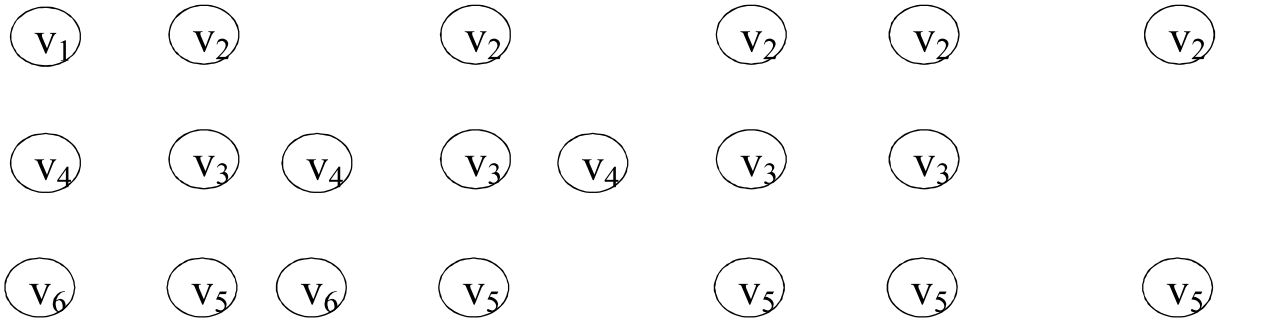
2 拓扑排序算法

算法思想

① 在AOV网中选择一个没有前驱的顶点且输出；

② 在AOV网中删除该顶点以及从该顶点出发的(以该顶点为尾的弧)所有有向弧(边)；

③ 重复①、②，直到图中全部顶点都已输出(图中无环)或图中不存在无前驱的顶点(图中必有环)。



(a) 有向图　(b) 输出v_1后　(c) 输出v_6后　(d) 输出v_4后　(e) 输出v_3后

图7-23 有向图的拓扑排序过程

3 算法实现说明

- 采用正邻接链作为AOV网的存储结构；
- 设立堆栈，用来暂存入度为0的顶点；
- 删除顶点以它为尾的弧：弧头顶点的入度减1。

算法实现

(1) 统计各顶点入度的函数

```
void count_indegree(ALGraph *G)
   {  int k ; LinkNode *p ;
      for (k=0; k<G->vexnum; k++)
         G->adjlist[k].indegree=0 ;     /*  顶点入度初始化  */
      for (k=0; k<G->vexnum; k++)
         {  p=G->adjlist[k].firstarc ;
            while (p!=NULL)     /*  顶点入度统计  */
               {  G->adjlist[p->adjvex].indegree++ ;
                  p=p->nextarc ;
               }
         }
   }
```

(2) 拓扑排序算法

```
int  Topologic_Sort(ALGraph *G, int topol[])
    /*  顶点的拓扑序列保存在一维数组topol中  */
   {  int  k, no, vex_no, top=0, count=0, boolean=1 ;
      int  stack[MAX_VEX] ;       /*  用作堆栈  */
      LinkNode *p ;
      count_indegree(G) ;   /*  统计各顶点的入度  */
      for (k=0; k<G->vexnum; k++)
         if  (G->adjlist[k].indegree==0)
            stack[++top]=G->adjlist[k].data ;
      do
         {  if (top==0)  boolean=0 ;
```

```
    else
       {   no=stack[top--] ;     /* 栈顶元素出栈 */
           topl[++count]=no ;    /* 记录顶点序列 */
           p=G->adjlist[no].firstarc ;
           while (p!=NULL)     /*删除以顶点为尾的弧*/
               {   vex_no=p->adjvex ;
                   G->adjlist[vex_no].indegree-- ;
                   if  (G->adjlist[vex_no].indegree==0)
                       stack[++top]=vex_no  ;
                   p=p->nextarc ;
               }
       }
}while(boolean==0) ;
```

```
    if (count<G->vexnum)  return(-1) ;
    else  return(1) ;
}
```

算法分析：设AOV网有n个顶点，e条边，则算法的主要执行是：

- 统计各顶点的入度：时间复杂度是O(n+e)；
- 入度为0的顶点入栈：时间复杂度是O(n)；
- 排序过程：顶点入栈和出栈操作执行n次，入度减1的操作共执行e次，时间复杂度是O(n+e)；

因此，整个算法的时间复杂度是O(n+e)。

7.6.2 关键路径(Critical Path)

与AOV网相对应的是AOE(Activity On Edge)，是边表示活动的有向无环图，如图7-24所示。图中顶点表示事件(Event)，每个事件表示在其前的所有活动已经完成，其后的活动可以开始；弧表示活动，弧上的权值表示相应活动所需的时间或费用。

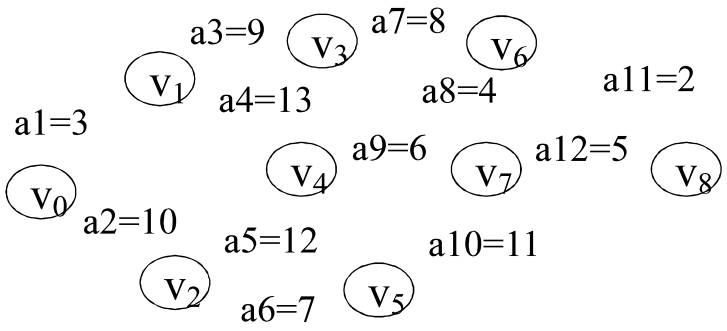


图7-24 一个AOE网

1 与AOE有关的研究问题

- 完成整个工程至少需要多少时间?
- 哪些活动是影响工程进度(费用)的关键?

工程完成最短时间：从起点到终点的最长路径长度(路径上各活动持续时间之和)。长度最长的路径称为关键路径，关键路径上的活动称为关键活动。关键活动是影响整个工程的关键。

设v_0是起点，从v_0到v_i的最长路径长度称为事件v_i的最早发生时间，即是以v_i为尾的所有活动的最早发生时间。

若活动a_i是弧<j, k>，持续时间是dut(<j, k>)，设：

- e(i)：表示活动a_i的最早开始时间；

◆ ***l*(i)**：在不影响进度的前提下，表示活动a_i的最晚开始时间； 则***l*(i)-e(i)**表示活动a_i的时间余量，若***l*(i)-e(i)=0**，表示活动a_i是关键活动。

◆ **ve(i)**：表示事件v_i的最早发生时间，即从起点到顶点v_i的最长路径长度；

◆ **v*l*(i)**：表示事件v_i的最晚发生时间。则有以下关系：

$$e(i)=ve(j)$$
$$l(i)= vl(k)-dut(<j, k>) \qquad \text{7-1}$$

$$ve(j)=\begin{cases} 0 & j=0，表示v_j是起点 \\ Max\{ve(i)+dut(<i, j>)|<v_i, v_j>是网中的弧\} & \end{cases} \qquad \text{7-2}$$

含义是：源点事件的最早发生时间设为0；除源点外，只有进入顶点v_j的所有弧所代表的活动全部结束后，事件v_j才能发生。即只有v_j的所有前驱事件v_i的最早发生时间ve(i)计算出来后，才能计算ve(j)。

方法是：对所有事件进行拓扑排序，然后依次按拓扑顺序计算每个事件的最早发生时间。

$$vl(j)=\begin{cases} ve(n-1) & j=n-1，表示v_j是终点 \\ Min\{vl(k)-dut(<j,k>)|<v_j,v_k>是网中的弧\} \end{cases} \quad 7\text{-}3$$

含义是：只有v_j的所有后继事件v_k的最晚发生时间vl(k)计算出来后，才能计算vl(j)。

方法是：按拓扑排序的逆顺序，依次计算每个事件的最晚发生时间。

2 求AOE中关键路径和关键活动

(1) 算法思想

① 利用拓扑排序求出AOE网的一个拓扑序列；

② 从拓扑排序的序列的第一个顶点(源点)开始，按拓扑顺序依次计算每个事件的最早发生时间ve(i)；

③ 从拓扑排序的序列的最后一个顶点(汇点)开始，按逆拓扑顺序依次计算每个事件的最晚发生时间vl(i)；

对于图7-24的AOE网，处理过程如下：

◆ 拓扑排序的序列是：(v_0, v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8)

◆ 根据计算ve(i)的公式(7-2)和计算v*l*(i)的公式(7-3)，计算各个事件的ve(i)和v*l*(i)值，如表7-2所示。

◆ 根据关键路径的定义，知该AOE网的关键路径是：$(v_0, v_2, v_4, v_7, v_8)$ 和$(v_0, v_2, v_5, v_7, v_8)$。

◆ 关键路径活动是：$<v_0, v_2>$，$<v_2, v_4>$，$<v_2, v_5>$，$<v_4, v_7>$，$<v_5, v_7>$，$<v_5, v_8>$。

表7-2 图7-24的ve(i)和v*l*(i)的值

顶点	v_0	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8
ve(i)	0	3	10	12	22	17	20	28	33
v*l*(i)	0	9	10	23	22	17	31	28	33

(2) 算法实现

```
void critical_path(ALGraph *G)
  {  int j, k, m ; LinkNode *p ;
     if  (Topologic_Sort(G)==-1)
        printf("\nAOE网中存在回路，错误!!\n\n") ;
     else
        {  for ( j=0; j<G->vexnum; j++)
              ve[j]=0 ;     /*  事件最早发生时间初始化  */
           for (m=0 ; m<G->vexnum; m++)
              {   j=topol[m] ;
                  p=G->adjlist[j].firstarc ;
                  for (; p!=NULL; p=p->nextarc )
```

```
            {  k=p->adjvex ;
               if  (ve[j]+p->weight>ve[k])
                    ve[k]=ve[j]+p->weight ;
            }
    }     /* 计算每个事件的最早发生时间ve值  */
for ( j=0; j<G->vexnum; j++)
    vl[j]=ve[j] ;    /*  事件最晚发生时间初始化  */
for (m=G->vexnum-1; m>=0; m--)
    {   j=topol[m] ; p=G->adjlist[j].firstarc ;
        for (; p!=NULL; p=p->nextarc )
            {  k=p->adjvex ;
               if  (vl[k]-p->weight<vl[j])
                    vl[j]=vl[k]-p->weight ;
```

```
                }
            }   /* 计算每个事件的最晚发生时间vl值 */
        for (m=0 ; m<G->vexnum; m++)
            {   p=G->adjlist[m].firstarc ;
                for (; p!=NULL; p=p->nextarc )
                    {  k=p->adjvex ;
                       if  ( (ve[m]+p->weight)==vl[k])
                           printf("<%d, %d>, m, j") ;
                    }
            }    /* 输出所有的关键活动 */
        }    /*  end of else  */
}
```

(3) 算法分析

设AOE网有n个事件，e个活动，则算法的主要执行是：

- 进行拓扑排序：时间复杂度是O(n+e)；
- 求每个事件的ve值和vl值：时间复杂度是O(n+e)；
- 根据ve值和vl值找关键活动：时间复杂度是O(n+e)；

因此，整个算法的时间复杂度是O(n+e)。

7.7 最短路径

若用带权图表示交通网，图中顶点表示地点，边代表两地之间有直接道路，边上的权值表示路程(或所花费用或时间)。从一个地方到另一个地方的路径长度表示该路径上各边的权值之和。问题：

- 两地之间是否有通路?
- 在有多条通路的情况下，哪条最短?

考虑到交通网的有向性，直接讨论的是带权有向图的最短路径问题，但解决问题的算法也适用于无向图。

将一个路径的起始顶点称为源点，最后一个顶点称为终点。

7.7.1 单源点最短路径

对于给定的有向图**G=(V，E)**及单个源点V_s，求V_s到**G**的其余各顶点的最短路径。

针对单源点的最短路径问题，**Dijkstra**提出了一种按路径长度递增次序产生最短路径的算法，即迪杰斯特拉**(Dijkstra)**算法。

1 基本思想

从图的给定源点到其它各个顶点之间客观上应存在一条最短路径，在这组最短路径中，按其长度的递增次序，依次求出到不同顶点的最短路径和路径长度。

即按长度递增的次序生成各顶点的最短路径，即先求出长度最小的一条最短路径，然后求出长度第二小的最短路径，依此类推，直到求出长度最长的最短路径。

2 算法思想说明

设给定源点为V_s，S为已求得最短路径的终点集，开始时令$S=\{V_s\}$。当求得第一条最短路径(V_s, V_i)后，S为$\{V_s, V_i\}$。根据以下结论可求下一条最短路径。

设下一条最短路径终点为V_j，则V_j只有：

- 源点到终点有直接的弧$<V_s, V_j>$；
- 从V_s出发到V_j的这条最短路径所经过的所有中间顶点必定在S中。即只有这条最短路径的最后一条弧才是从S内某个顶点连接到S外的顶点V_j。

若定义一个数组**dist[n]**，其每个**dist[i]**分量保存从V_s出发中间只经过集合**S**中的顶点而到达V_i的所有路径中长度最小的路径长度值，则下一条最短路径的终点V_j必定是不在**S**中且值最小的顶点，即：

$$dist[i]=Min\{\ dist[k]|\ V_k \in V-S\ \}$$

利用上述公式就可以依次找出下一条最短路径。

3 算法步骤

① 令**S={Vs}**，用带权的邻接矩阵表示有向图，对图中每个顶点**Vi**按以下原则置初值：

$$dist[i]=\begin{cases} 0 & i=s \\ W_{si} & i\neq s\text{且}<v_s,v_i>\in E,\ w_{si}\text{为弧上的权值} \\ \infty & i\neq s\text{且}<v_s,v_i>\text{不属于}E \end{cases}$$

② 选择一个顶点V_j，使得：

$dist[j]=Min\{ dist[k]| V_k \in V-S \}$

V_j就是求得的下一条最短路径终点，将V_j并入到S中，即$S=S \cup \{V_j\}$ 。

③ 对V-S中的每个顶点V_k，修改dist[k]，方法是：

若$dist[j]+W_{jk}<dist[k]$，则修改为：

$dist[k]=dist[j]+W_{jk}$ $(\forall V_k \in V-S)$

④ 重复②，③，直到S=V为止。

4 算法实现

用带权的邻接矩阵表示有向图， 对**Prim**算法略加改动就成了**Dijkstra**算法，将**Prim**算法中求每个顶点V_k的**lowcost**值用**dist[k]**代替即可。

- 设数组**pre[n]**保存从V_s到其它顶点的最短路径。若**pre[i]=k**，表示从V_s到V_i的最短路径中，V_i的前一个顶点是V_k，即最短路径序列是$(V_s, \ldots, V_k, V_i)$。
- 设数组**final[n]**，标识一个顶点是否已加入**S**中。

算法实现的关键

待求点的最短路径长度本身就是待求的，又如何找出其中的最短呢？

```
BOOLEAN  final[MAX_VEX] ;
int  pre[MAX_VEX] , dist[MAX_VEX] ;
void Dijkstra_path (AdjGraph *G, int v)
        /*  从图G中的顶点v出发到其余各顶点的最短路径   */
   {  int j, k, m, min ;
      for ( j=0; j<G->vexnum; j++)
         {   pre[j]=v ;  final[j]=FALSE ;
             dist[j]=G->adj[v][j] ;
         }   /* 各数组的初始化 */
      dist[v]=0 ; final[v]=TRUE ;     /* 设置S={v} */
      for ( j=0; j<G->vexnum-1; j++) /* 其余n-1个顶点 */
         {   m=0 ;
```

```
while (final[m])  m++;   /*  找不在S中的顶点v_k */
min=INFINITY ;
for ( k=0; k<G->vexnum; k++)
   {  if  (!final[k]&&dist[m]<min)
          {   min=dist[k] ;  m=k ; }
   }     /* 求出当前最小的dist[k]值  */
final[m]=TRUE ;      /*  将第k个顶点并入S中  */
for ( j=0; j<G->vexnum; j++)
   { if (!final[j]&&(dist[m]+G-
>adj[m][j]<dist[j]))
        {   dist[j]=dist[m]+G->adj[m][j] ;
            pre[j]=m ;
        }
    }     /* 修改dist和pre数组的值  */
```

```
        }      /* 找到最短路径  */
    }
```

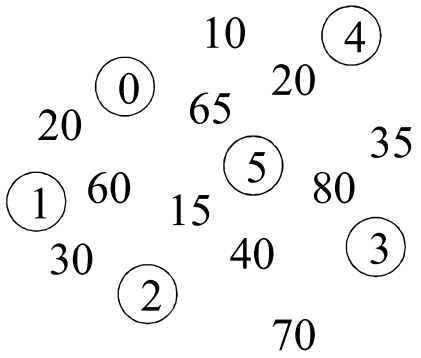
5 算法分析

Dijkstra算法的主要执行是：

- 数组变量的初始化：时间复杂度是**O(n)**；
- 求最短路径的二重循环：时间复杂度是**O(n2)**；

因此，整个算法的时间复杂度是**O(n2)**。

对图**7-25**的带权有向图，用**Dijkstra**算法求从顶点**0**到其余各顶点的最短路径，数组**dist**和**pre**的各分量的变化如表**7-3**所示。



∞	20	60	∞	10	65
∞	∞	30	70	∞	∞
∞	∞	∞	40	∞	∞
∞	∞	∞	∞	35	∞
∞	∞	∞	∞	∞	20
∞	∞	15	80	∞	∞

图**7-25** 带权有向图及其邻接矩阵

表7-3　求最短路径时数组dist和pre的各分量的变化情况

步骤＼顶点		1	2	3	4	5	S
初态	Dist pre	20 0	60 0	∞ 0	10 0	65 0	{0}
1	Dist pre	20 0	60 0	∞ 0	10 0	30 4	{0, 4}
2	Dist pre	20 0	50 1	90 1	10 0	30 4	{0, 4, 1}
3	Dist pre	20 0	45 5	90 1	10 0	30 4	{0, 4, 1, 5}
4	Dist pre	20 0	45 5	85 2	10 0	30 4	{0, 4, 1, 5, 2}
5	Dist pre	20 0	45 5	85 2	10 0	30 4	{0, 4, 1, 5, 2, 3}

7.7.2 每一对顶点间的最短路径

用**Dijkstra**算法也可以求得有向图**G=(V，E)**中每一对顶点间的最短路径。方法是：每次以一个不同的顶点为源点重复**Dijkstra**算法便可求得每一对顶点间的最短路径，时间复杂度是$O(n^3)$。

弗罗伊德**(Floyd)**提出了另一个算法，其时间复杂度仍是$O(n^3)$，但算法形式更为简明，步骤更为简单，数据结构仍然是基于图的邻接矩阵。

1 算法思想

设顶点集S(初值为空)，用数组A的每个元素A[i][j]保存从V_i只经过S中的顶点到达V_j的最短路径长度，其思想是：

① 初始时令S={ }， A[i][j]的赋初值方式是：

$$A[i][j]=\begin{cases}0 & i=j\text{时}\\ W_{ij} & i\neq j\text{且}<v_i,v_j>\in E,\ w_{ij}\text{为弧上的权值}\\ \infty & i\neq j\text{且}<v_i,v_j>\text{不属于}E\end{cases}$$

② 将图中一个顶点V_k加入到S中，修改A[i][j]的值，修改方法是：

A[i][j]=Min{A[i][j] , (A[i][k]+A[k][j]) }

原因： 从V_j只经过S中的顶点(V_k)到达V_j的路径长度可能比原来不经过V_k的路径更短。

③ 重复②，直到G的所有顶点都加入到S中为止。

2 算法实现

◆ 定义二维数组Path[n][n](n为图的顶点数)，元素Path[i][j]保存从V_i到V_j的最短路径所经过的顶点。

◆ 若Path[i][j]=k：从V_i到V_j经过V_k，最短路径序列是(V_i, …, V_k, …, V_j)，则路径子序列：(V_i, …, V_k)和(V_k, …, V_j)一定是从V_i到V_k和从V_k到V_j的最短路径。从而可以根据Path[i][k]和Path[k][j]的值再找到该路径上所经过的其它顶点，…依此类推。

◆ 初始化为**Path[i][j]=-1**，表示从V_i到V_j不经过任何(**S**中的中间)顶点。当某个顶点V_k加入到**S**中后使**A[i][j]**变小时，令**Path[i][j]=k**。

表**7-4**给出了利用**Floyd**算法求图**7-26**的带权有向图的任意一对顶点间最短路径的过程。

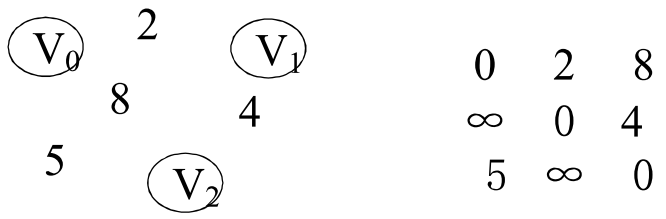


图**7-26** 带权有向图及其邻接矩阵

表7-4 用Floyd算法求任意一对顶点间最短路径

步骤	初态	k=0	K=1	K=2
A	$\begin{pmatrix} 0 & 2 & 8 \\ \infty & 0 & 4 \\ 5 & \infty & 0 \end{pmatrix}$	$\begin{pmatrix} 0 & 2 & 8 \\ \infty & 0 & 4 \\ 5 & 7 & 0 \end{pmatrix}$	$\begin{pmatrix} 0 & 2 & 6 \\ \infty & 0 & 4 \\ 5 & 7 & 0 \end{pmatrix}$	$\begin{pmatrix} 0 & 2 & 6 \\ 9 & 0 & 4 \\ 5 & 7 & 0 \end{pmatrix}$
Path	$\begin{pmatrix} -1 & -1 & -1 \\ -1 & -1 & -1 \\ -1 & -1 & -1 \end{pmatrix}$	$\begin{pmatrix} -1 & -1 & -1 \\ -1 & -1 & -1 \\ -1 & 0 & -1 \end{pmatrix}$	$\begin{pmatrix} -1 & -1 & 1 \\ -1 & -1 & -1 \\ -1 & 0 & -1 \end{pmatrix}$	$\begin{pmatrix} -1 & -1 & 1 \\ 2 & -1 & -1 \\ -1 & 0 & -1 \end{pmatrix}$
S	{ }	{ 0 }	{ 0, 1 }	{ 0, 1, 2 }

根据上述过程中Path[i][j]数组，得出：

V_0到V_1： 最短路径是{ 0, 1 }， 路径长度是2；

V_0到V_2： 最短路径是{ 0, 1, 2 }， 路径长度是6；

V_1到V_0： 最短路径是{ 1, 2, 0 }， 路径长度是9；

V_1到V_2：最短路径是{ 1, 2 }，路径长度是4；

V_2到V_0：最短路径是{ 2, 0 }，路径长度是5；

V_2到V_1：最短路径是{ 2, 0, 1 }，路径长度是7；

算法实现

```
int  A[MAX_VEX][MAX_VEX] ;
int  Path[MAX_VEX][MAX_VEX] ;
void Floyd_path (AdjGraph *G)
   {  int j, k, m ;
      for ( j=0; j<G->vexnum; j++)
         for ( k=0; k<G->vexnum; k++)
            {  A[j][k]=G->adj[j][k] ; Path[j][k]=-1 ;  }
               /*  各数组的初始化  */
```

```
for ( m=0; m<G->vexnum; m++)
   for ( j=0; j<G->vexnum; j++)
      for ( k=0; k<G->vexnum; k++)
        if ((A[j][m]+A[m][k])<A[j][k])
           {  A[j][k]=A[j][m]+A[m][k] ;
              Path[j][k]=k ;
           }    /*  修改数组A和Path的元素值  */
for ( j=0; j<G->vexnum; j++)
   for ( k=0; k<G->vexnum; k++)
      if (j!=k)
        {  printf("%d到%d的最短路径为:\n", j, k) ;
           printf("%d  ",j) ; prn_pass(j, k) ;
           printf("%d  ", k) ;
```

```
                printf("最短路径长度为: %d\n",A[j][k]) ;
            }
    }     /*  end of Floyd  */

void  prn_pass(int j , int k)
    {  if (Path[j][k]!=-1)
        {  prn_pass(j, Path[j][k]) ;
           printf(", %d" , Path[j][k]) ;
           prn_pass(Path[j][k],  k) ;
        }
    }
```