第 4 章　图

图（Graph）是一种比线性表和树更为复杂的非线性数据结构。在线性表中，数据元素之间仅存在线性关系，每个数据元素（除了首元素）只有一个直接前驱，每个元素（除了尾元素）只有一个直接后继。在树形结构中，数据元素间存在着明显的层次关系，每一层上的数据元素可以和下一层的一个或多个元素相关，但只能和上一层中的一个元素相关。在图结构中，数据元素间的关系可以是任意的，图中任意两个元素之间都可能相关。图结构反映的是一种网状关系。

图在数据挖掘、信息论、博弈论、运筹学等领域都有着广泛的应用。如何将实际应用中图（如互联网）的数据及关系存储到计算机中？如何解决实际应用的问题（如确定数据包在给定的两台计算机之间传送的最短路径）？本章主要介绍图的逻辑结构、存储结构以及图的常用操作。

4.1　图的基本概念

4.1.1　图的定义和概念

图是由顶点集合 V 和顶点之间的关系集合 E 组成的一种数据结构：

$$G=(V,E)$$

其中，V 是一个非空有限集合，代表顶点，E 代表关系的非空有限集合。在将图 G 可视化时，通常将两个顶点之间的关系用两个顶点之间的边来表示。

1．常用图的基本类型

无向图：在图 G 中，如果代表关系的边没有方向，则称 G 为无向图（undirected graph）。无向图中的边通过顶点的无序对来表示，如(v_i, v_j)表示 v_i 和 v_j 之间的无向边。一个公司的局域网就是一个无向图，其中把计算机看作结点，计算机之间的网络连接看作边。图 4-1 中 G_1 就是一个无向图，其中 $V(G_1)=\{v_1,v_2,v_3,v_4,v_5\}$，$E(G_1)=\{(v_1,v_2),(v_1,v_4),(v_2,v_3),(v_2,v_5),(v_3,v_4),(v_3,v_5)\}$。

有向图：如果图 G 中的边是有方向的，则称 G 为有向图（directed graph）。有向图中的边通过顶点的有序对来表示，如$<v_i, v_j>$表示 v_i 指向 v_j 的边。有向图中边又称为“弧”。在互联网中，如果把网页看作顶点，超级链接看作是源页面到目标页面间的有向边，则网页链接结构就是一个有向图。图 4-1 中 G_2 就是一个有向图，其中 $V(G_2)=\{v_1,v_2,v_3,v_4\}$，$E(G_2)=\{<v_1,v_2>,<v_1,v_3>,<v_3,v_4>,<v_4,v_1>,<v_4,v_2>\}$。

带权图：图的边（或弧）可以被赋予相关的数值，表达一定的实际意义（如表示两点之间的路径长度、通讯费用等）。边（或弧）上的数值称为“权”，边上具有权的图称

为带权图；根据边是否有方向又可分为带权有向图和带权无向图。图 4-1 中 G_3 就是一个带权无向图。

子图：对于图 $G=(V,E)$ 和 $G'=(V',E')$，如果 V' 是 V 的子集，即 $V'\subseteq V$，且 E' 是 E 的子集，即 $E'\subseteq E$，则称 G' 为 G 的子图。如图 4-1 中，G_4 是 G_1 的子图，G_5 是 G_2 的子图。

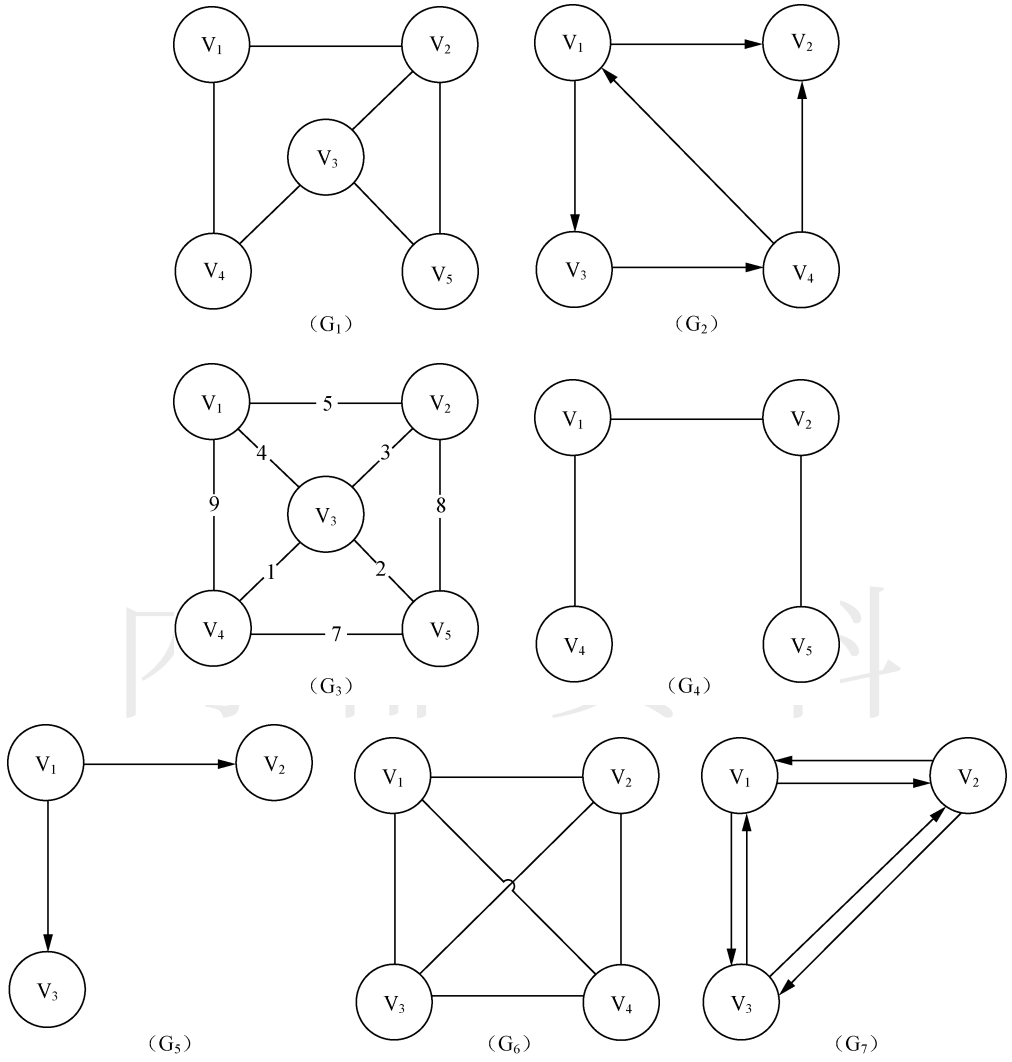

图 4-1　图的基本类型

2．关联边和邻接点

如果图中两点 v_i 与 v_j 之间存在一条边（v_i，v_j）(或< v_j，v_i,>)，则称 v_i 与 v_j 相邻接，（v_i，v_j）（或< v_j，v_i>）与顶点 v_i、v_j 相关联。如图 4-1 的无向图 G_1 中，v_3 和 v_2 是邻接

点，v_3 和 v_4 是邻接点，v_3 和 v_5 是邻接点；在图 4-1 的有向图 G_2 中，存在有向边 $<v_1,v_2>$，则称顶点 v_1 邻接到顶点 v_2，顶点 v_2 邻接于顶点 v_1；v_1、v_2 分别是有向边 $<v_1,v_2>$ 的始点和终点。如图 4-1 的无向图 G_1 中，与顶点 v_1 相关联的边有 $(v_1,v_2),(v_1,v_4)$。在图 4-1 的有向图 G_2 中，与顶点 v_1 相关联的边有 $<v_1,v_2>$、$<v_1,v_3>$ 和 $<v_4,v_1>$。

如果图中每条边关联不同的两个顶点（即不存在点到点自身的边）且不存在相同的边，则该图称为简单图。本书仅限于讨论简单图的结构、算法及应用。

性质 1 在具有 n 个顶点的无向图中，设边数为 e，则有：

$$0 \leqslant e \leqslant \frac{n(n-1)}{2}$$

性质 2 在具有 n 个顶点的有向图中，设边数为 e，则有：

$$0 \leqslant e \leqslant n(n-1)$$

包括所有可能边的图称为完全图；边数相对较少的图称为稀疏图，反之称为稠密图。图 4-1 中，G_6 是一个完全无向图，G_7 则是一个完全有向图。

3．顶点的度

在无向图中，顶点的度是与该顶点相关联的边的数目。在有向图中，顶点的度分为入度和出度，以顶点 v_i 为终点的边的数目，称为顶点 v_i 的入度；以顶点 v_i 为起点的边的数目，称为顶点 v_i 的出度。如图 4-1 的无向图 G_1 中，顶点 v_1 的度为 2；有向图 G_2 中，顶点 v_1 的入度为 1，出度为 2，度为 3。若一个图中有 n 个顶点和 e 条边，每个顶点的度为 $d_i(1 \leqslant i \leqslant n)$，则有 $e = \frac{1}{2}\sum_{i=1}^{n} d_i$。

4．路径及路径长度

在图 G(V,E) 中，如果从顶点 v_i 出发，经过一些顶点和边到达顶点 v_j，则称顶点序列 $\{v_i = w_0, w_1, w_2, ..., w_{m-1}, w_m = v_j\}$，其中 $(w_i, w_{i+1}) \in E$（或 $<w_i, w_{i+1}> \in E$）为顶点 v_i 到顶点 v_j 的路径。例如，图 4-1 的 G_1 中顶点序列 $\{v_1, v_2, v_3, v_5\}$ 是 v_1 到 v_5 的路径，G_2 中顶点序列 $\{v_1, v_3, v_4\}$ 是 v_1 到 v_4 路径。路径上的边数定义为路径长度。若一条路径上除起点和终点可以相同以外，其余顶点均不相同，则称此路径为简单路径。若一条路径的起点与终点相同，则此路径叫做回路或环。如果构成回路的路径是简单路径，则称此回路为简单回路。不带回路的图称为无环图。

5．图的连通性

在无向图 G 中，若从顶点 v_i 到 v_j 存在路径，则称 v_i 和 v_j 是连通的。若无向图 G 中任意两个顶点都连通，则称无向图 G 为连通图，反之为非连通图，例如，图 4-1G_1、G_3、G_4、G_6 都是连通的，图 4-2 是不连通的。无向图的最大连通子图称为连通分量，例如，图 4-2 中有两个连通分量 C_1 和 C_2，如图 4-3 所示。显然，任何连通图的连通分量只有

一个，即本身，而非连通图有多个连通分量。

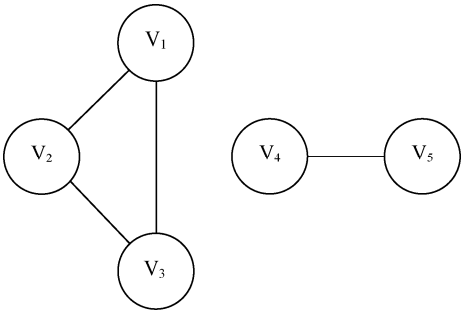

图 4-2　不连通的无向图 G_8

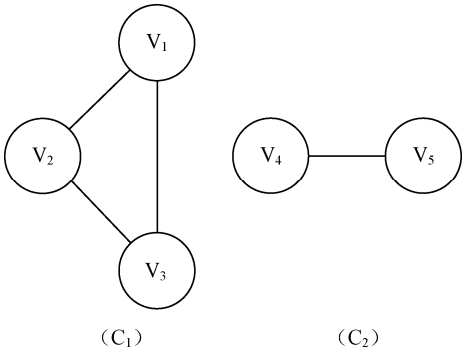

图 4-3　G_8 连通分量

在有向图 G 中，如果从 v_i 到 v_j 或从 v_j 到 v_i 都存在路径，则称顶点 v_i 和 v_j 连通。若图中任意两个顶点 v_i 和 v_j，既存在从 v_i 到 v_j 的路径，也存在从 v_j 到 v_i 的路径，则称图 G 是强连通图，例如，图 4-1 中的 G_7 是强连通的，有向图 G_2 不是强连通的，因为不存在从 v_2 到 v_1 的路径。有向图的最大强连通子图称为该有向图的强连通分量，例如，图 4-4 所示的是图 4-1 中 G_2 的两个强连通分量。显然，强连通图只有一个强连通分量，即本身，而非强连通图有多个强连通分量。

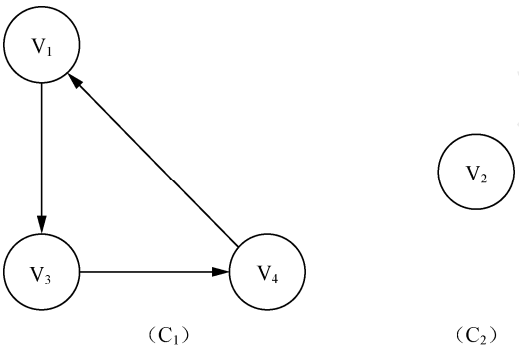

图 4-4　G_2 的两个强连通分量

6．生成树

对于具有 n 个顶点的连通图 G，如果存在连通子图 G'包含 G 中所有顶点和一部分边，且不形成回路，则称 G'为图 G 的生成树。显然，连通图 G 的生成树就是它的极小连通子图，具有如下性质：

1）包含 n 个顶点；

2）包含 n−1 条边；

3）是图 G 的连通子图。

如图 4-5 所示的连通图都是图 4-1 中 G_1 的生成树。显然生成树不唯一。

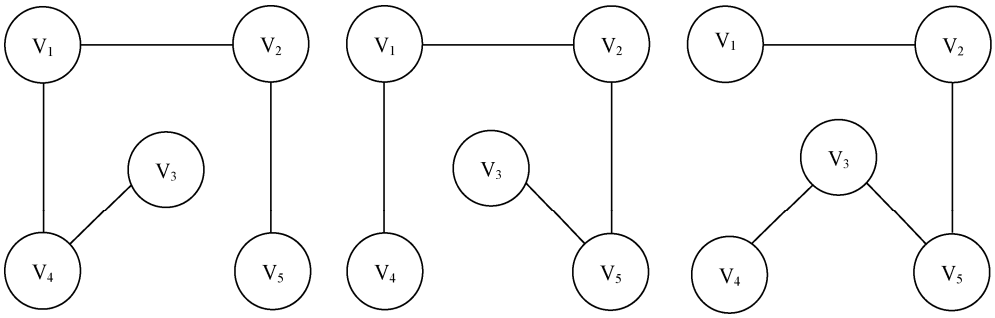

图 4-5　G_1 的生成树

4.1.2　图的抽象数据类型

4.1.1 节给出了图的基本概念，现在讨论图的抽象数据类型。虽然实际应用问题中的顶点包括很多信息，但是在处理过程中，通常先将顶点存储到静态的顶点信息表中并进行编号，然后通过编号来表示某个顶点。为了简便起见，假设 n 个顶点的编号为 0 到 n−1。

例 4.1 给出了图的抽象数据类型定义。

【例 4.1】图的抽象数据类型。

```
//G=（V,E）代表一个图，V 即是顶点的非空有限集合，E 是边（或弧）的集合
Template<class EdgeType>
class Graph
{
    public:
        Graph(int verticesNum);     //构造函数
        ~Graph();                   //析构函数

        virtual EdgeType FirstEdge(int oneVertex)=0;
                    //返回与顶点 oneVertex 相关联的第一条边
        virtual EdgeType NextEdge(Edge<EdgeTYpe> oneEdge)=0;
                    //返回与边 oneEdge 有相同起点的下一条边
        virtual void setEdge(int start,int end,int weight)=0;
                    //设置边（start,end）,权重为 weight
        virtual void delEdge(int start,int end)=0;
                    //删除边（start,end）

        int VerticesNum();          //返回图的顶点个数
        int EdgesNum();             //返回图的边数
        bool IsEdge(Edge<EdgeType> oneEdge);
                    //如果 oneEdge 是边则返回 TRUE，否则返回 FALSE
        int StartVertex(Edge<EdgeType> oneEdge);//返回边 oneEdge 的始点
```

```
        int EndVertex(Edge<EdgeType> oneEdge);//返回边 oneEdge 的终点
        EdgeType Weight(Edge<EdgeType> oneEdge);    //返回边 oneEdge
                                                    的权
};
```

由于图的大多数操作都需要顶点数目、边数目以及判断一个顶点是否已经处理等信息，所以通常将图描述成如下的类型。本章以此类作为图的基类。

【例 4.2】图的类定义。

```
template <class EdgeType>
class Edge{            //边类型
    public:
        int start,end;
        EdgeType weight;//start 是边的始点,end 为边的终点,weight 是边的权
        Edge();                        //构造一条边

        Edge(int st,int en,int w); //构造边(st,en),权重为 w
        bool operator >(Edge oneEdge);
                                       //重载运算符>,通过边权重比较边的大小
        bool operator <(Edge oneEdge);
                                       //重载运算符<,通过边权重比较边的大小
};

template <class EdgeType>
class Graph{                           //图类型
public:
    int vertexNum;                     //图的顶点数目
    int edgeNum;                       //图的边数目
    int *Mark;                         //标记某顶点是否被访问过
    Graph(int verticesNum){            //构造函数
        vertexNum=verticesNum;         //初始化图的顶点的个数
        edgeNum=0;                     //初始化图的边的个数
        Mark=new int[vertexNum];       //申请数组,Mark 为数组指针
        for(int i=0;i<vertexNum;i++)
      { //标志位初始化为未被访问过,入度初始化为 0
            Mark[i]=UNVISITED;
        }
    }
    ~Graph(){                          //析构函数
        delete [] Mark;                //释放 Mark 数组
        delete [] Indegree;            //释放 Indegree 数组
    }
    virtual Edge<EdgeType> FirstEdge(int oneVertex)=0;
    virtual Edge<EdgeType> NextEdge(Edge<EdgeType> oneEdge)=0;
    int VerticesNum()                  //返回图的顶点个数
    { return vertexNum; }
    int EdgesNum()                     //返回图的边数
```

```
    { return edgeNum; }
    bool IsEdge(Edge<EdgeType> oneEdge)
    {  //如果 oneEdge 是边则返回 TRUE，否则返回 FALSE
        if(oneEdge.weight>0&&oneEdge.weight<INFINITY&&oneEdge.end>=0)
            return true;
        else
            return false;
    }
    int StartVertex(Edge<EdgeType> oneEdge) //返回边 oneEdge 的始点
    { return oneEdge.start; }
    int EndVertex(Edge<EdgeType> oneEdge)   //返回边 oneEdge 的终点
    { return oneEdge.end; }
    EdgeType Weight(Edge<EdgeType> oneEdge)//返回边 oneEdge 的权重
    { return oneEdge.weight; }
    virtual void setEdge(int start,int end,int weight)=0;
    virtual void delEdge(int start,int end)=0;
};
```

下面将根据图的不同的存储结构，以不同的方式来实现图类的具体定义。

4.2 图的存储及基本操作

图的存储方法分为顺序存储（邻接矩阵）和链式存储（邻接表、邻接多重表以及十字链表）两大类型，应用中往往根据不同的需求使用不同的存储方法。

4.2.1 图的邻接矩阵表示法

在图的邻接矩阵表示中，顶点信息记录在一个顶点表中，顶点之间的邻接关系用一个二维数组表示。其中，数组的每一个元素表示一条边，元素的两个下标分别代表相邻接的两个顶点编号。若 $G=<V,E>$ 是一个具有 n 个顶点的图，则该图的邻接矩阵是如下定义的 n×n 矩阵：

$$A[i][j]=\begin{cases}1, & 若(v_i,v_j)\in E或<v_i,v_j>\in E\\ 0, & 若(v_i,v_j)\notin E或<v_i,v_j>\notin E\end{cases}$$

图 4-1 中的无向图 G_1 和有向图 G_2 的邻接矩阵如下所示：

$$A_1=\begin{bmatrix}0&1&0&1&0\\1&0&1&0&1\\0&1&0&1&1\\1&0&1&0&0\\0&1&1&0&0\end{bmatrix}\qquad A_2=\begin{bmatrix}0&1&1&0\\0&0&0&0\\0&0&0&1\\1&1&0&0\end{bmatrix}$$

由于无向图中的边 (v_i,v_j) 等价于 (v_j,v_i)，因此无向图的邻接矩阵是对称的，如 A_1 所示。有向图的邻接矩阵则不一定是对称的。利用邻接矩阵很容易获得顶点的度，如无向图中顶点 v_i 的度是第 i 行的元素或第 i 列的元素之和，而在有向图中，顶点 v_i 的出度是第 i

行的元素之和，顶点 v_i 的入度是第 i 列的元素之和。

对于带权图，设 $w_{i,j}$ 是边 (v_i, v_j)（或 $< v_i, v_j >$）的权，则邻接矩阵定义如下：

$$A[i][j] = \begin{cases} w_{ij}, & \text{若 } i \neq j \text{且} (v_i, v_j) \in E \text{或} < v_i, v_j > \in E \\ \infty, & \text{若} i \neq j \text{且} (v_i, v_j) \notin E \text{或} < v_i, v_j > \notin E \\ 0, & \text{若} i = j \end{cases}$$

图 4-1 中的带权图 G_3 的邻接矩阵如下所示：

$$A_3 = \begin{bmatrix} 0 & 5 & 4 & 9 & \infty \\ 5 & 0 & 3 & \infty & 8 \\ 4 & 3 & 0 & 1 & 2 \\ 9 & \infty & 1 & 0 & 7 \\ \infty & 8 & 2 & 7 & 0 \end{bmatrix}$$

下面给出采用邻接矩阵存储图的类型定义和操作实现。

【例 4.3】图的邻接矩阵的实现。

```
templete <class EdgeType〉
class AdjGraph: public Graph
{
    private:
        int **matrix;                    //指向邻接矩阵的指针
public:
    AdjGraph(int verticesNum):Graph(verticesNum) {
        int i,j;                         //i,j 为 for 循环中的计数器
        //申请 vertexNum 大小的 matrix 数组
        matrix=(int**)new int*[vertexNum];
        for(i=0;i<vertexNum;i++)         //申请 matrix 数组行的存储空间
            matrix[i] = new int[vertexNum];
        for(i=0;i<vertexNum;i++)         //初始化邻接矩阵的元素
          for(j=0;j<vertexNum;j++)
                matrix[i][j]=0;
    }
    ~AdjGraph()                          //析构函数
    {
        for(int i=0;i<vertexNum;i++)//释放每个 matrix[i] 申请的空间
            delete [] matrix[i];
        delete [] matrix;                //释放 matrix 指针指向的空间
    }
    Edge<EdgeType> FirstEdge(int oneVertex)
    {//返回顶点 oneVertex 的第一条边
        Edge<EdgeType> tmpEdge;          //边 tmpEdge 将作为函数的返回值
        //将顶点 oneVertex 作为边 tmpEdge 的始点
     tmpEdge.start=oneVertex;
     for(int i=0;i<vertexNum;i++)
     {  //找第一个 matrix[oneVertex][i] 不为 0 的 i 值
```

```
                if(matrix[oneVertex][i]!=0){
                    tmpEdge.end=i;      //将顶点 i 作为边 tmpEdge 的终点
                    tmpEdge.weight=matrix[oneVertex][i];
                    break;
                }
            }
            return tmpEdge;
        }
        Edge<EdgeType> NextEdge(Edge<EdgeType> oneEdge)
{     //返回与边 oneEdge 有相同始点的下一条边
            Edge<EdgeType> tmpEdge;     //边 tmpEdge 将作为函数的返回值
            //边 tmpEdge 的始点置为上一条边的始点
          tmpEdge.start=oneEdge.start;
          for(int i=oneEdge.end+1;i<vertexNum;i++){
                if(matrix[oneEdge.start][i]!=0)
                {
                    tmpEdge.end=i;
                    tmpEdge.weight=matrix[oneEdge.start][i];
                    break;
                }
            }
            return tmpEdge;
        }

        void setEdge(int start,int end, EdgeType weight){  //为图设置边
            if(matrix[start][end]==0) {                    //新增一条边
                edgeNum++;
            }
            matrix[start][end]=weight;                     //设置边
        }
        void delEdge(int start,int end){                   //删除边
            if(matrix[start][end]!=0)   {                  //该边存在
                edgeNum--;
            }
            matrix[start][end]=0;
        }
};
```

使用邻接矩阵存储图信息，可以很容易地判定任意两个顶点间是否有边相连。如果图中含有 n 个顶点，则邻接矩阵存储需要占用的存储单元为 n×n 个，与边的数目无关，因此邻接矩阵适用于稠密图的存储。

4.2.2 图的邻接表表示法

对于稀疏图（具有很少条边），采用邻接矩阵存储会造成存储空间的浪费。图的邻接表表示则是一种适用于稀疏图存储的表示方法。

邻接表表示法是对图中每一个顶点 v_i 建立一个单链表，将所有与 v_i 关联的边存储到该链表中。链表中的结点称为边结点，包含三个域：邻接点的编号（adjvex），边的信息（arcinfo），指示下一条关联边的边结点指针（nextarc）。其中，边的信息域是针对带权图而设计的，如果是非加权图，该域可以省略。每条链表设一个头结点，存储与该链表中所有边都关联的顶点信息。头结点通常采用顺序结构进行存储，以便进行随机访问。

为了不失一般性，可以将图 G_1 和 G_2 附上边信息，如图 4-6 中的 G_9 和 G_{10} 所示，图 4-7 和图 4-8 分别给出了无向图 G_9 和有向图 G_{10} 的邻接表表示。

由图 4-7 可以看出，使用邻接表存储无向图，顶点 v_i 的度就是第 i 条链表中的边结点数目。另外每条边在它关联的两个顶点的链表里各存储一次，因此，存储 n 个顶点 e 条边的无向图需要占用 n+2e 个单元的存储空间。

对于有向图的邻接表，可以方便的计算顶点 v_i 的出度，即第 i 条链表中的边结点数目。如果要知道顶点 v_i 的入度，必须遍历整个邻接表，查看有多少个边结点指向顶点为 v_i。使用邻接表存储有向图时，每条边的信息只在发出该边的顶点链表里存储一次，因此所需要的存储空间为 n+e。

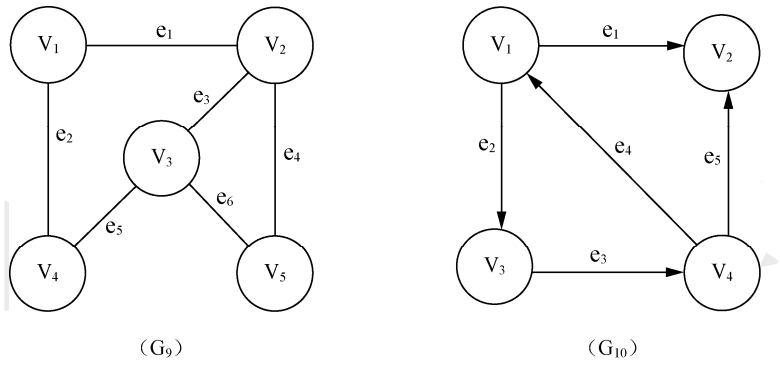

图 4-6　带权图

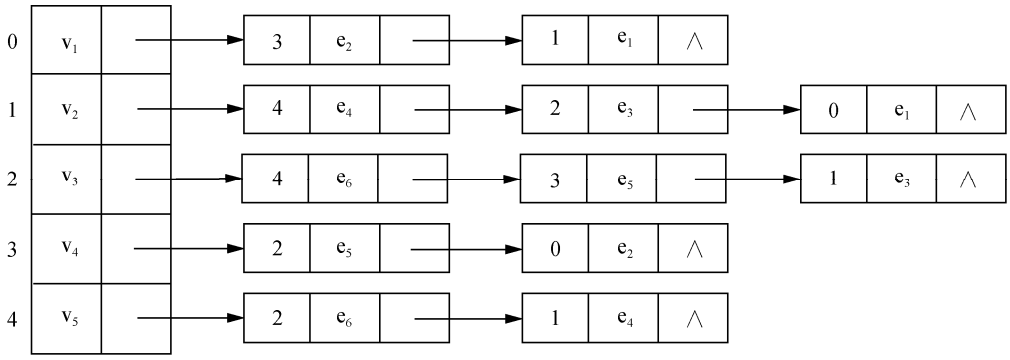

图 4-7　无向图 G_9 的邻接表表示

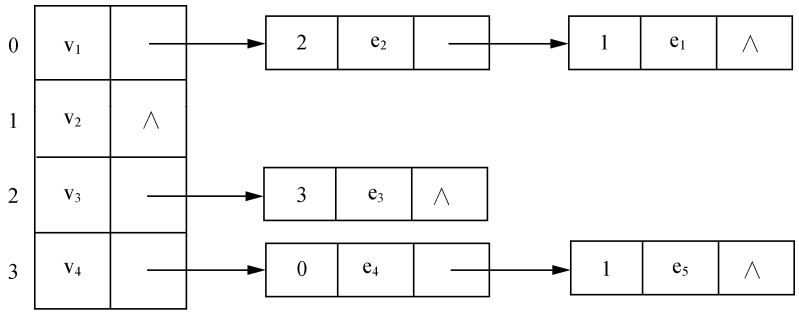

图 4-8　有向图 G_{10} 的邻接表表示

在计算图的关键路径等有向图的实际应用中，不仅需要知道由每个顶点出发的边信息，还需要使用指向每个顶点的边信息，而采用邻接表存储显然不便于找到指向每个顶点的边。因此对于有向图，还可以使用逆邻接表来存储。

和邻接表的存储类似，逆邻接表也包括头结点和边结点，头结点包含两个域：顶点的信息（vexinfo），指向该顶点第一条关联边的边结点指针（firstarc）。边结点包含三个域：邻接顶点序号（adjvex），边的信息（arcinfo），指向该顶点的下一条关联边的边结点指针（nextarc）。

如有向图 G_{10} 的逆邻接表表示如图 4-9 所示。

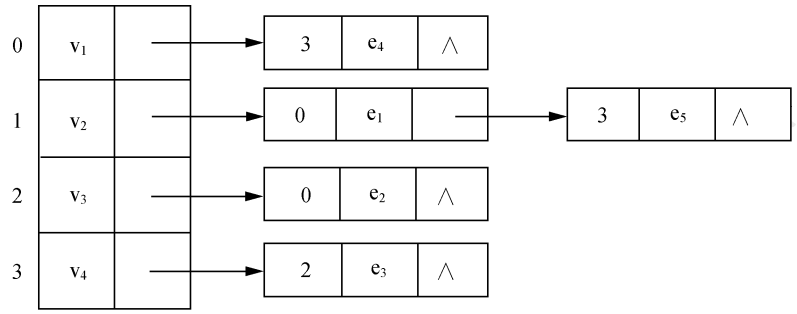

图 4-9　有向图 G_{10} 的逆邻接表表示

例 4.4 给出了图的邻接表表示的实现。

【例 4.4】图的邻接表的实现。

```
template<class EdgeType>
class listData                          //邻接表边结点的数据部分定义
{
public:
    int vertex;                         //边的终点
    EdgeType weight;                    //边的权
};
template<class Elem>                    //定义边结点
class ListNode {
```

```
        public:
            Elem element;                    //边结点的数据
            ListNode<Elem> *next;            //边结点指针，指向下一个边结点
            ListNode(const  Elem&  elemval,ListNode<Elem>  *nextval=NULL)
{ element=elemval; next=nextval; }
            ListNode(ListNode<Elem> *nextval=NULL) { next=nextval; }
        };
        template<class Elem>
        class EdgeList {                     //每个顶点所关联的边表
        public:
            ListNode<Elem>* head;            //head 指针
            EdgeList(){                      //构造函数
                head=new ListNode<Elem>();
            }
            void removeall()                 //释放边表所有边结点占据的空间
            {
                ListNode<Elem> *tmp;
                while(head!=NULL)            //逐步释放每个边结点占据的空间
                {
                    tmp=head;
                    head=head->next;
                    delete tmp;
                }
            }

            ~EdgeList() { removeall(); }     //析构函数
        };
        template <class EdgeType>
        class ListGraph: public Graph        //图的邻接表表示
        {
        private:
            EdgeList<listData<EdgeType>> *graList;
                                             //graList 是保存所有边表的数组
        public:
            ListGraph(int verticesNum):Graph(verticesNum) //构造函数
           {//申请空间，有 vertexNum 个顶点则有 vertexNum 个边表
                graList=new EdgeList<listData<EdgeType>>[vertexNum];
            }
            ~ListGraph()                     //析构函数
            { delete [] graList; }
            Edge FirstEdge(int oneVertex)    //返回顶点 oneVertex 的第一条边
            {
                Edge<EdgeType> tmpEdge;      //将边 tmpEdge 作为函数的返回值
                tmpEdge.start=oneVertex;
               ListNode<listData<EdgeType>> *temp=graList[oneVertex].head;
               //graList[oneVertex].head 保存的是顶点 oneVertex 的边表，
               //temp->next 指向顶点 oneVertex 的第一条边(如果 temp->next 不为 null)
```

```
            if(temp->next!=NULL)        //顶点 oneVertex 的第一条边存在
            {
                tmpEdge.end=temp->next->element.vertex;
                tmpEdge.weight=temp->next->element.weight;
            }
            return tmpEdge;
        }
        Edge<EdgeType> NextEdge(Edge<EdgeType> oneEdge)
    {//返回与边 OneEdge 有相同关联顶点的下一条边
            Edge<EdgeType> tmpEdge;
            tmpEdge.start=oneEdge.start;
            ListNode<listData<EdgeType>>*temp=graList[oneEdge.start].
head;
        //确定边 oneEdge 在边表中的位置,如果边 oneEdge 的下一条边确实存在,
        //则 temp->next 指针指向下一条边的表目
        while(temp->next!=NULL&&temp->next->element.vertex<=oneEdge.end)
                temp=temp->next;
            if(temp->next!=NULL)        //边 oneEdge 的下一条边存在
            {
                tmpEdge.end=temp->next->element.vertex;
                tmpEdge.weight=temp->next->element.weight;
            }
            return tmpEdge;
        }
        void setEdge(int start,int end, EdgeType weight)//为图设定一条边
        {
            ListNode<listData<EdgeType>> *temp=graList[start].head;
            while(temp->next!=NULL&&temp->next->element.vertex<end)
            //确定边(start,end)或<start,end>在边表中的位置,如果不存在,则边
            //(start,end)或<start,end>为新加的一条边
                temp=temp->next;
            if(temp->next==NULL)
        {//边在边表中不存在且在边表中其后已无其他边,
         //则在边表中加入这条边
                temp->next=new ListNode<listData>;
                temp->next->element.vertex=end;
                temp->next->element.weight=weight;
                edgeNum++;
                return;
            }
            if(temp->next->element.vertex==end)     //边在边表中已存在
            {
              temp->next->element.weight=weight;
                return;
            }
            if(temp->next->element.vertex>end)
            {//边在边表中不存在,但在边表中其后存在其他边,
```

```
            //则在边表中插入这条边
                ListNode<listData> *other=temp->next;
                temp->next=new ListNode<listData>;
                temp->next->element.vertex=end;
                temp->next->element.weight=weight;
                temp->next->next=other;
                edgeNum++;

            }
        }
        void delEdge(int start,int end)         //删掉图的一条边
        {
            ListNode<listData<EdgeType>> *temp=graList[start].head;
            while(temp->next!=NULL&&temp->next->element.vertex<end)
                temp=temp->next;
            if(temp->next==NULL) return;        //边不存在,不需任何操作
            if(temp->next->element.vertex==end)//边存在,将其删掉
            {
                ListNode<listData<EdgeType>> *other=temp->next->next;
                delete temp->next;
                temp->next=other;
                edgeNum--;
            }
        }
    };
```


4.2.3　十字链表和邻接多重表

十字链表是有向图的另一种链式存储结构，可以看成是邻接表和逆邻接表的结合。在十字链表中，也有两种结点类型即：边结点和头结点。边结点描述边的信息，共有五个域：始点编号（startvex）、终点编号（endvex）、指针域 startnextarc、指针域 endnextarc 和 arcinfo 域。指针域 startnextarc 指向始点相同的下一个边结点；指针域 endnextarc 指向终点相同的下一个边结点；此外还有一个表示边权值等信息的 arcinfo 域。头结点由三个域组成：vexinfo 域、finstinarc 域和 firstoutarc 域。vexinfo 域存放顶点的相关信息；指针域 firstinarc 指向以该顶点为终点的第一个边结点；指针域 firstoutarc 指向以该顶点为始点的第一个边结点。所有的头顶点通常存放在顺序存储结构中。

如有向图 G_{10} 的十字链表存储如图 4-10 所示。

在十字链表中，很容易找到以 v_i 为始点和终点的边。从 v_i 所对应的头结点的 firstoutarc 出发，沿着边结点的 startnextarc 域链接起来的链表，正好是原来的邻接表结构，统计这个链表中的边结点个数，可以得到顶点 v_i 的出度。如果从 v_i 所对应的头结点的 firstinarc 出发，沿着 endnextarc 域链接起来的链表，恰好是原来的逆邻接表结构，统计这个链表中的边结点个数，可以求出顶点 v_i 的入度。

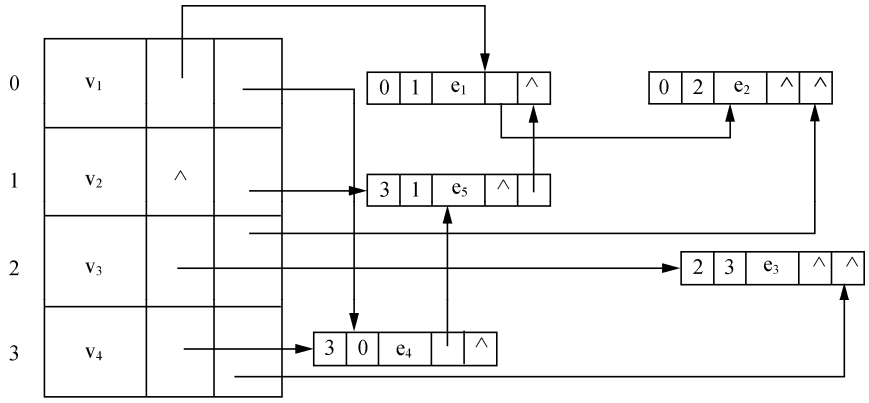

图 4-10　有向图 G_{10} 的十字链表存储示意图

邻接多重表是无向图的另一种链式存储结构。虽然邻接表是无向图的一种很有效的存储结构，但在邻接表中，每一条边都被存储两次，导致在某些对边进行的操作（例如对搜索过的边做标记）中就需要对每一条边处理两遍。采用邻接多重表存储无向图更加便于实现这类操作。

邻接多重表的结构与其他的链式存储结构类似，也具有头结点和边结点。边结点包括五个域：数据域 ivex 和 jvex 描述一条边所关联的两个顶点编号；指针域 inext 指向与该边具有相同的关联顶点 ivex 的下一个边结点；指针域 jnext 指向与该边具有相同的关联顶点 jvex 的下一个边结点；数据域 info 描述该边的权重等信息。头结点包含两个域：描述顶点信息的 vexinfo 域，以及指向与该顶点关联的第一个边结点的指针 firstedge。

例如图 4-1 中无向图 G_9 的邻接多重表的存储结构如图 4-11 所示。

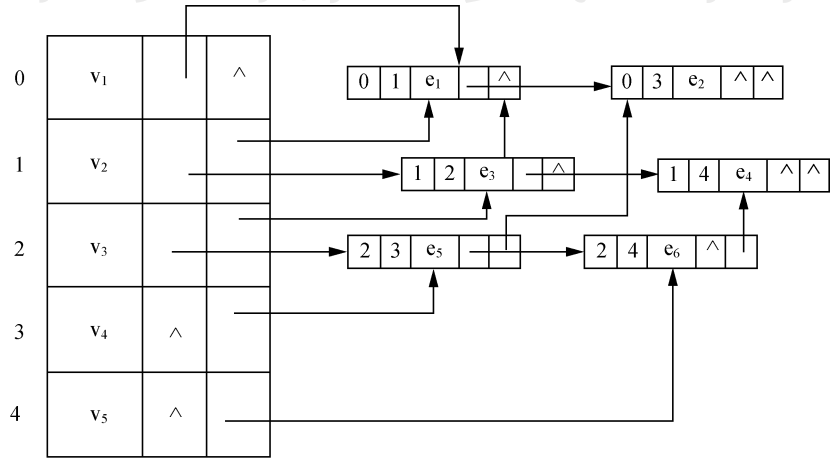

图 4-11　无向图 G_9 的邻接多重表存储结构示意图

采用邻接多重表存储无向图，也可以方便地找出一个顶点 v_i 关联的所有边，方便地

计算顶点的度，并且每条边只被存储一次，节省了一定的存储空间，但是存储规模仍然是 O(n＋e) 。

4.3　图 的 遍 历

与树的遍历相似，图的遍历是指从某一顶点出发访遍图中所有顶点，且使每一个顶点仅被访问一次的过程。图的遍历是许多图算法及图的应用基础，如网络爬虫就是按照图的遍历策略来爬取网络中页面。图的遍历是建立在“记忆化访问”的基础上，即在遍历的过程中为每一个顶点设置一个标志位，标记该顶点是否被访问过，这是由图的结构特点所决定的。例如，非连通图中从某一顶点出发可能不会到达其他所有顶点；再如图中存在回路则有可能导致算法陷入死循环，通过“记忆化访问”便可以解决这种问题。

图的遍历算法有深度优先搜索（Depth First Search，DFS）和广度优先搜索（Breadth First Search，BFS）两种。这两种遍历算法对于无向图和有向图都适用，这里以无向图为例来介绍。

4.3.1　深度优先搜索（DFS）

深度优先搜索类似树的前序遍历，基本思想是从图中某个顶点 v 出发，访问此顶点并标记为“已访问”，然后依次从与 v 相邻接且未被访问的邻接点 u 出发进行深度优先搜索，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，再选择图中一个未被访问的顶点做起始点，重复上述过程，直至图中所有顶点都被访问到。例如，按照深度优先搜索的方式遍历图 4-12 中的无向图 G_{11}，可以得到如下的顶点序列：v_1，v_2，v_4，v_8，v_5，v_3，v_6，v_7。深度优先搜索得到的遍历序列可能不唯一。

在搜索过程中,由某个顶点 v 访问与其相邻且未被访问的顶点 u 时经过的边（v,u）称为前向边。对于一个连通图 G，深度优先搜索过程中的所有前向边和顶点组成的子图 G'是原图 G 的一个生成树，也称为深度优先搜索生成树。对于图 4-12 的连通图 G_{11}，从顶点 v_1 开始进行深度优先搜索得到的深度优先搜索生成树如图 4-13 所示。

例 4.5 给出了深度优先搜索的算法实现。

【例 4.5】图的深度优先搜索（DFS）算法。

从某个顶点 v 开始进行的深度优先搜索的代码实现如下：

```
template <class EdgeType>
void Graph<EdgeType>::DFS(int v)
{
    Mark[V]= VISITED;                    //标记该顶点已访问
    visit(v);                            //访问该顶点
    for(Edge e=FirstEdge(V);IsEdge(e);e=NextEdge(e))
  { //由该点所关联的边进行深度优先搜索
          //访问 V 邻接到的未被访问过的顶点，并递归地进行深度优先搜索
       if(Mark[G.EndVertex(e)]==UNVISITED)
```

```
                DFS(EndVertex(e));
        }
    }
```

对于图的深度优先搜索的算法实现如下：

```
template <class EdgeType>
void Graph<EdgeType>::DFSTraverse()
{
    for(int i=0;i<VerticesNum();i++)    //对所有顶点的标志位初始化
        Mark[i]=UNVISITED;
    for(i=0;i<VerticesNum();i++)
  { //检查图是否有未访问的顶点，如果有则从该顶点开始深度优先搜索
        if(Mark[i]== UNVISITED)
                DFS(i);                 //对未访问的顶点调用 DFS
    }
}
```

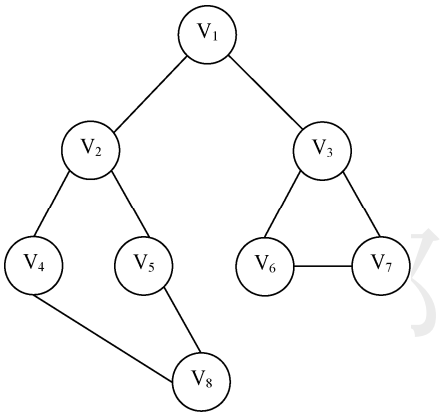

图 4-12　无向图 G_{11}

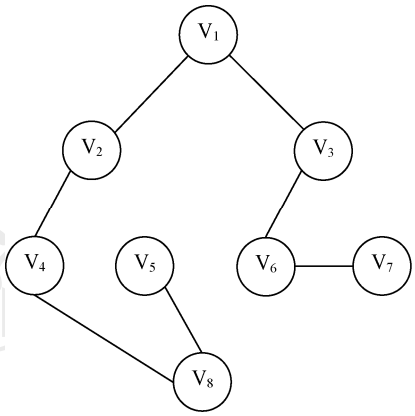

图 4-13　无向连通图 G_{11} 的深度优先搜索生成树

深度优先搜索也可以采用非递归的方法实现，这时需要使用栈结构。

【例 4.6】深度优先搜索的非递归实现。

```
template <class EdgeType>
void Graph<EdgeType>::DFSNoReverse()
{
    int i,v,u;
    ArrayStack<int> s;
    for( i=0;i<VerticesNum();i++)    //对所有顶点的标志位初始化
        Mark[i]=UNVISITED;
    for(i=0;i<VerticesNum();i++)
  { //检查图是否有未访问的顶点，如果有则从该顶点开始深度优先搜索
        if(Mark[i]== UNVISITED)
        {
```

```
                    s.push(i);
                    visit(i);
                    Mark[i]=VISITED;
                    while(!s.isEmpty())
                    {
                        v = s.pop();
                        for(Edge<EdgeType> e = FirstEdge(v); isEdge(e); e =
NextEdge(e))
                        {
                            u = EndVertex(e);
                            if(Mark[u] ==UNVISITED)
                            {
                                s.push(u);
                                visit(i);
                                Mark[u]=VISITED;
                            }
                        }
                    }
                }
            }
        }
```

深度优先搜索过程中，对图中每个顶点至多调用一次 DFS 函数。搜索过程实质上是对每个顶点查找其邻接点的过程，其耗费的时间取决于所采用的存储结构。用邻接矩阵表示图时，共需检查 n^2 个矩阵元素，所需时间为 $O(n^2)$；而使用邻接表时，找邻接点需将邻接表中所有边结点检查一遍，耗时为 $O(e)$，对应的深度优先搜索算法的时间复杂度为 $O(n+e)$。

4.3.2　广度优先搜索（BFS）

图的广度优先搜索类似于树的层次遍历，基本思想是从图中某个顶点 v 出发，访问并标记此顶点，然后依次访问 v 的各个未被访问的邻接点，并对这些邻接点进行以上相同的操作，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，则另选图中一个未被访问的顶点做起始点，重复上述过程，直至图中所有顶点都被访问到。例如，按照广度优先搜索的方式遍历图 4-12 中所示的无向图 G_{11}，得到的顶点序列是 v_1，v_2，v_3，v_4，v_5，v_6，v_7，v_8。对于一个连通图 G，

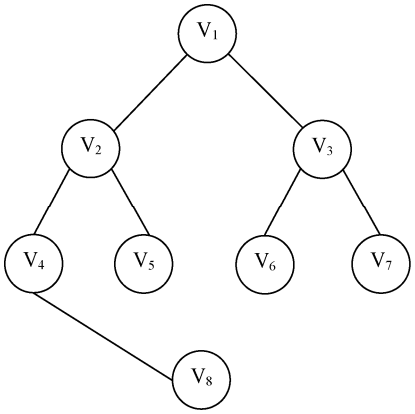

图 4-14　无向连通图 G_{11} 的广度优先搜索生成树

广度优先搜索过程中的所有前向边和顶点组成的子图 G'也是原图 G 的一个生成树，称为广度优先搜索生成树。例如对于连通图 G_{11}，从顶点 v_1 开始进行广度优先搜索得到的广度优先搜索生成树如图 4-14 所示。

例 4.7 给出了从顶点 v 开始的广度优先搜索的实现以及图的广度优先搜索算法实现。

【例 4.7】图的广度优先搜索(BFS)算法。

从一个顶点 v 开始的广度优先搜索算法如下：

```
template <class EdgeType>
void Graph<EdgeType>::BFS(int v)
{
    ArrayQueue<int> Q;                  //广度优先搜索要用到的队列
   Mark[v]= VISITED;                    //访问顶点 V，并将标志位置为已访问
   visit(v);                            //访问顶点 v
    Q.push(v);                          //v 入队
    while(!Q.empty())                   //如果队列仍然有元素
    {
        u=Q.front();
        Q.pop();                        //出队
        for(Edge<EdgeType> e = FirstEdge(v);IsEdge(e);e = NextEdge (e))
                                        //与该点相邻的每一个未访问点都入队
        {
            int u = Mark[EndVertex(e)];
            if(u == UNVISITED)
            {
                visit(u);               //访问顶点 u
                Mark[u]= VISITED;       //标记 u 的标志位
                Q.push(u);              //入队
            }
        }
    }
}
```

图的广度优先搜索算法如下：

```
template <class EdgeType>
void Graph<EdgeType>::BFSTraverse()
{
    int v;
   for(v=0;i<VerticesNum();v++)     //对所有顶点的标志位初始化
        Mark[v]=UNVISITED;
    for(v=0;i<VerticesNum();v++)
    {//检查图中是否有未访问的顶点，如果有则从该顶点开始广度优先搜索
        if(Mark[v]==UNVISITED)
            BFS(v);
    }
}
```

图的广度优先搜索中，每个顶点至多入队一次，搜索过程实质上也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相同，两者的不同之处仅仅在于对顶点访问的顺序不同。

4.4　最小生成树

生成树的概念已在 4.1 节给出。具有 n 个顶点的连通图 G 的生成树是其包含 n 个顶点、n−1 条边的极小连通子图。利用深度优先搜索和广度优先搜索可以得到连通图的生成树。

对于带权无向图，生成树上各条边的权重之和称为生成树的代价。代价最小的生成树称为最小生成树（Minimum-Cost Spanning Tree，MST）。许多应用问题都是求无向连通图的最小生成树问题。如：如何在 n 个城市之间铺设公路，使得任意两个城市之间都可以到达并且总费用最小？如何在 n 个教学楼之间铺设网线，使得任何两个教学楼之间都可以网络通信并且使用的线缆最少？

图 4-15 中（b）是（a）的一棵最小生成树。

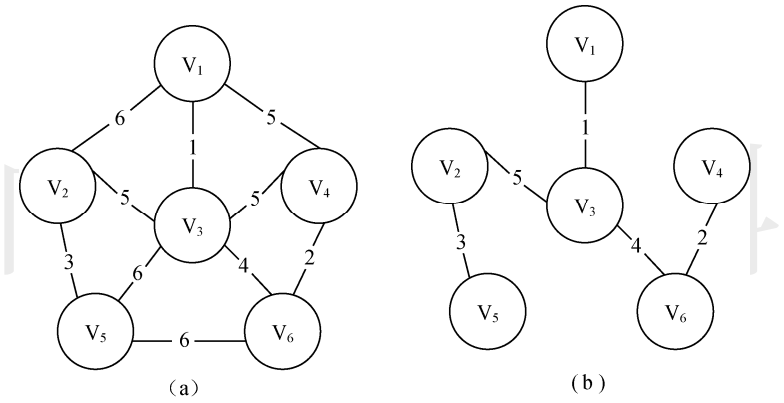

图 4-15 （a）为带权图 G_{12}，（b）为 G_{12} 的一棵最小生成树

在实际应用中，通常利用构造方法来构造最小生成树。典型的构造方法有：普里姆（Prim）算法和克鲁斯卡尔（Kruskal）算法。

4.4.1　普里姆（Prim）算法

设 $G=<V,E>$ 是一个连通的带权无向图。Prim 算法通过不断地增加生成树的顶点来得到最小生成树。在算法的任一时刻，一部分顶点已经添加到生成树的顶点集合中，而其余顶点尚未加到生成树中。此时，Prim 算法通过选择边 (u,v)，使得 (u,v) 的权值是所有 u 在生成树中但 v 不在生成树中的边的权值最小者，从而找到新的顶点 v 并把它添加到生成树中。图 4-16 指出该算法如何从顶点 v_1 开始构建图 G_{12} 的最小生成树。初始

时，只有顶点 v_1 在构造的生成树中，之后每一步向生成树中添加一条边和一个顶点。

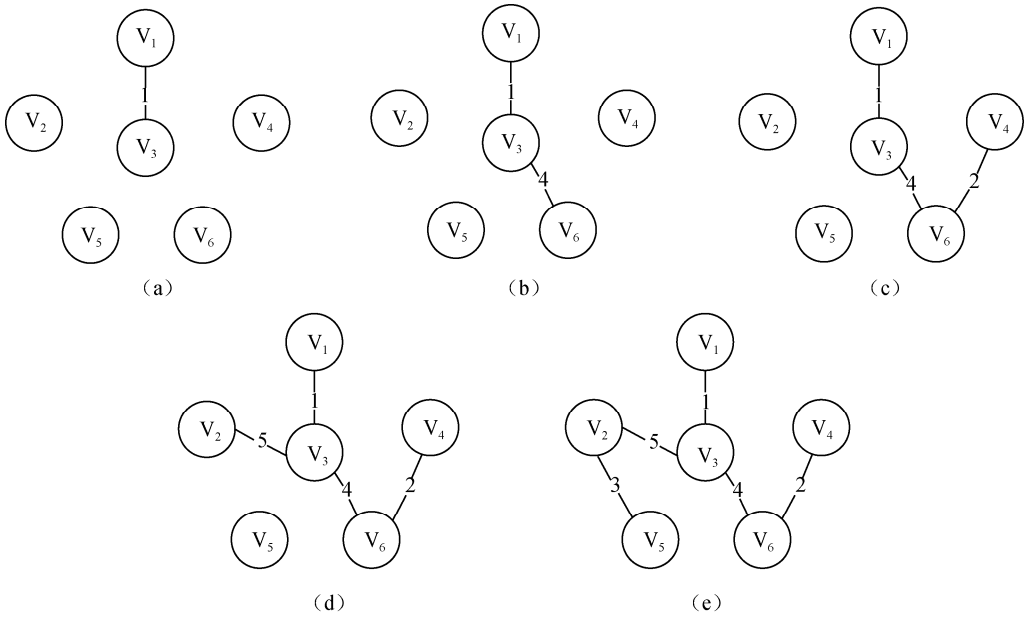

图 4-16　Prim 算法构造带权图 G_{12} 的最小生成树的步骤

设 U 是算法执行过程中最小生成树的顶点集合，TE 是最小生成树中边的集合。Prim 算法的详细步骤如下：

1）初始状态，$U=\{u_1\}$，$TE=\{\}$。其中，u_1 是图的顶点集合中某一顶点。

2）在所有 $u\in U$，$v\in V-U$ 的边 (u,v) 中寻找代价最小的边(u',v')，并纳入集合 TE 中；同时将 v'纳入集合 U 中。这一过程不会产生回路。

3）如果 $U=V$，则算法结束；否则重复步骤 2。

Prim 算法在执行过程中，需要标注各个点是否已经在生成树集合中；另外当向生成树中添加一个新顶点时，就要更新那些一个顶点已经在生成树集合而另一个顶点不在生成树集合中的边信息，Prim 算法是通过这样的边来发现新的顶点。在算法的具体实现时，可以引入 neighbor 和 nearest 数组，neighbor[j]的值如果为−1 则表明顶点 j 已经在生成树集合中；否则 neighbor[j]的值为已经在生成树中的某个顶点编号，该顶点与顶点 j 的边权值是所有已经在生成树中的顶点和 j 的边的权值中最小者，nearest[j]即为相应的最小权值。Prim 算法每次遍历 neighbor 和 nearest 数组，对于所有满足 neighbor[j]不为-1 的顶点 j，找到一个最小的 nearest[v]，则 v 即为新找到的顶点，（neighbor[v],v）为相应的边。这个确定新顶点的过程也可以通过最小堆来实现。在向生成树中添加顶点 v 后，要再次更新 neighbor 数组和 nearest 数组，对于尚未在生成树集合中顶点 j，检查一下（j，v）的边权值是否小于 nearest[j],如果是，则更新 nearest[j]以及 neighbor[j]。

例 4.8 给出了 Prim 算法的实现。

【例 4.8】Prim 算法的实现。

```
//最小生成树的 Prim 算法
template <class EdgeType>
Edge<EdgeType> * Prim(Graph< EdgeType >& G, int s)
{   //应用 Prim 算法从 s 顶点出发得到的最小生成树
    int i,j;
    Edge<EdgeType> *MST;    //存储最小生成树的边
    //各个顶点到生成树中的各个顶点的最短的边
    EdgeType *nearest;      //nearest[i]表示生成树中点到 i 点的最小边权值
    int *neighbor;          //neighbor[i]表示生成树中与 i 点最近的点编号,
                            //-1 表示 i 点已经在生成树集合中
   int n = G.VerticesNum();//图的顶点个数

    nearest = new EdgeType[n];
    neighbor = new int[n];
    MST = new EdgeType[n-1];
   for(i=0;i<n;i++)         //初始化 neighor 数组和 nearest 数组
    {
        neighbor[i]=s;
        nearest[i]=AFFINITY;
    }
   //与 s 相邻接的顶点的边权值作为这些点距离生成树集合的最短边长
   for( e=G.FirstEdge(s);G.IsEdge(e);e=G.NextEdge(e))
   {
        nearest[i]=e.weight;
   }

   neighbor[s]=-1;//将已加入到生成树的点的最近邻设置为-1
    for(i=1; i < n; i++)
    { //i 标记已经加入到生成树中的点个数
        EdgeType min=Affinity;
        int v=-1;
        for(j=0;j<n;j++)
        {//确定一个顶点在生成树集合一个顶点不在生成树集合且权值最小的
         //边所关联的顶点
            if(nearest[j]<min&&neighor[j]>-1)
            {
                min = nearest[j];
                v = j;
            }
        }//for(j)
        if(v>=0)
        {//将 v 加入到生成树集合中,更新到生成树外的各个点最小权值的边信息
```

```
                neighbor[v]=-1;
                Edge<EdgeType> tempEdge(neigbor[v],v,nearest[v]);
                MST[i]=tempEdge;//将边加入到生成树集合中
                for( e=G.FirstEdge(v);G.IsEdge(e);e=G.NextEdge(e))
                {
                    int u = e.end;
                    if(neighor[u]!=-1&&nearest[u]>e.weight)
                    {//用与 v 关联的边更新生成树之外顶点到生成树集合的最小权值边
                        neibhbor[u]=v;
                        nearest[u]=e.weight.
                    }
                }//for(e)
            }//if(v>=0)
        }//for(i)
        delete []neighbor; //释放空间
        delete []nearest;
       return MST;
    }
```

Prim 算法的实现主要是两个过程的重复：寻找一个满足条件的未插入到生成树集合中的顶点；利用该顶点更新其余顶点的信息。所以 Prim 算法的时间复杂度为 $O(n^2)$，其中 n 为图的顶点数。Prim 算法的复杂度与图中的边数无关，因此适用于边数比较稠密的图。

4.4.2 克鲁斯卡尔（Kruskal）算法

设 $G=<V,E>$ 是一个连通的带权图,令最小生成树的初始状态为只有 n 个孤立顶点的非连通图 $T=(V,\{\})$，图中每个顶点自成一个连通分量。Kruskal 算法的基本思想是基于贪心准则，首先在 E 中选择权重最小的边，若该边所关联的两个顶点属于两个不同的连通分量，则将此边加入到 T 中，否则选择下一条权重最小的边。重复上述过程，直至 T 中所有顶点都在一个连通分量中为止。

例如，对于图 4-15 中所示的带权图 G_{12}，按照 Kruskal 算法选取边的过程如图 4-17 所示。

Kruskal 算法的三个核心操作是：确定权值最小的边；判定一条边所关联的两个顶点是否在一个连通分量中；如果不是则合并两个顶点所属的连通分量。第一个操作可以按边的权值组成优先队列，该优先队列通过最小堆来实现。第二、三个操作涉及到合并两个集合操作，合并两个线性表复杂度为$O(n^2)$。可以通过“等价类”（也称之为“并查集”）来实现集合的合并操作，合并两个集合的复杂度为$O(nlgn)$。把连通分量中的顶点作为等价类的元素，采用等价类的查找（Find）算法确定边的两个连结点所属的连通分

量从而判定是否相同，采用等价类的合并（Union）方法合并两个连通分量。

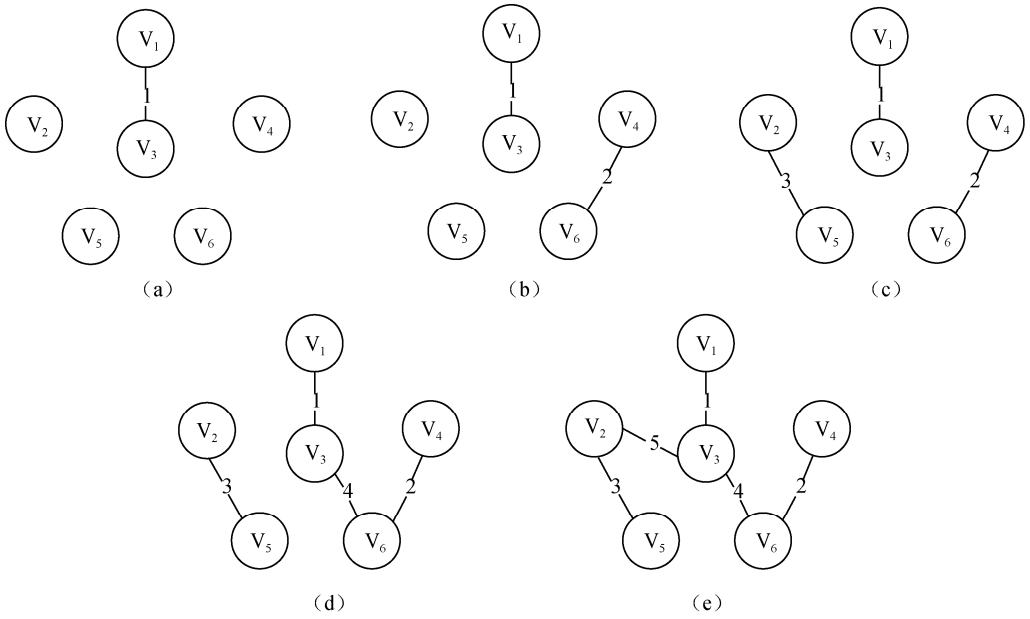

图 4-17　Kruskal 算法构造带权图 G_{12} 的最小生成树的步骤

【例 4.9】等价类。

```
class UFSets
{
    private:
        int n;          //等价类中元素个数
        int* root;      //root[i]表示元素 i 所在的等价类的代表元素编号
        int* next;      //next[i]表示在等价类中，i 的后面元素编号
        int*length;     //length[i]表示 i 所代表的等价类的元素个数
    public:
        UFSets(int size){           //初始 size 个元素的等价类
            n = size;
            for(int i=0;i<n;i++)
            {
                root[i]= next[i]=i; //各个元素独自成为一个等价类
                length[i]= 1;
            }
        }
        int Find(int v){return root[v];}
        void Union(int v,int u);    //合并 v 和 u 所在的等价类，将元素多的
                                    //合并到元素少的等价类中
    };
    void UFSets::Union( int v, int u)
```

```
        {
            if(root[u]==root[v])          //如果 v 和 u 在一个等价类中则返回
                return;
            else if(length[root[v]]<length[root[u]])
            {//将 v 所在等价类元素合并到 u 所在等价类,
            //并将 u 所在等价类的代表元素作为合并后的等价类的代表元

                rt =root[v];              //获取 v 所在等价类的代表元素
                length[root[u]]+= length[rt];//修改 u 所在等价类的元素个数

                root[rt]=root[u];//修改 v 所在等价类的各个元素的代表元素信息
                for(j=next[rt];j!=rt;j=next[j])
                    root[j]=root[u];

                swap(next[rt],next[root[u]]);
            //将两个等价类的元素连接起来
            }
            else // length[root[v]]>= length[root[u]]
            { //与上面类似  }
    }
```

基于上述等价类的定义，可以给出如下的 Kruskal 算法的实现。

【例 4.10】Kruskal 算法的实现。

```
    //最小生成树的 Kruskal 算法
    template <class EdgeType>
    Edge<EdgeType> * Kruskal(Graph<EdgeType>& G)
    {   //求含有 n 个顶点、e 条边的连通图 G 的最小生成树
        int n = G.VerticesNum();
        UFSets set(n);                //定义 n 个结点的等价类
        Edge<EdgeType> *MST=new Edge<EdgeType>[n-1];//记录最小生成树的边
        //定义含有 e 个元素的最小堆
       MinHeap<Edge<EdgeType>> H(G.EdgesNum());
        Edge<EdgeType> edge;
       for(int i=0; i<G.VerticesNum(); i++)//将图的所有边记录在数组 E 中
        {
            for(  edge=  G.FirstEdge(i);G.IsEdge(edge);edge=G.NextEdge
(edge))
            {
                if(G.StartVertex(edge)< G.EndVertex(edge))
              {//避免无向图中的边被重复考查
                    H.insert(edge);
              }
            }
        }
       int edgeNum = 0;          //生成树的边个数
        while(edgeNum <n-1)      //n 个结点的连通图的生成树有 n-1 条边
```

```
        {
            if(!H.isEmpty())
            {
                edge = H.removemin();       //找到权重最小的未处理的边
                int v = edge.start;
                int u = edge.end;
                if(set.Find(v)!=set.Find(u))
                {//判断该边关联的顶点是否在一个连通分量
                    set.Union(v,u);         //合并两个顶点所在的等价类
                    //将符合条件的边添加到生成树的边集合中
                    MST[edgeNum++]= edge;
                }
            }
            else
            {
                assert("不存在最小生成树.");
                return;
            }
        }
        return MST;
    }
```

Kruskal 算法实现过程中，向堆中插入了 e 条边，时间复杂度是 O(elog e)；从堆中最多删除了 e 条边，时间复杂度是 O(elog e)；最多执行 2e 次等价类的查找操作，时间复杂度是 O(e)；最多执行 e 次等价类的合并操作，时间复杂度是 O(eloge)。所以总的时间复杂度是 O(elog e)。Kruskal 算法的时间复杂度主要取决于边数，适用于构造稀疏图的最小生成树。

4.5 最 短 路 径

在实际生活中,经常会遇到如何选择最佳出行路线（费用最低或时间最短）的问题。这种最佳路线也称为最短路径。求带权有向图中的最短路径在许多领域里都有实际应用意义。所谓最短路径指的是，从图中某顶点出发（该点称为源点），经过图上的一些边到达另一顶点（称为终点）的所有路径中路径长度（路径上各条边的权重之和）最小的路径。本节介绍两种常见的最短路径——单源最短路径和所有顶点对之间的最短路径及其计算方法。

4.5.1 单源最短路径

对于图 G = (V,E)，给定源点 s∈ V，单源最短路径指的是从 s 到图中其他各顶点的最短路径。例如，对图 4-18 所示的带权有向图，从 v_0 到其余各个顶点的最短路径如表 4-1 所示。求解单源最短路径的一个常用算法是迪杰斯特拉（Dijkstra）算法。Dijkstra 算法

是一种按照路径长度递增的次序产生到各顶点最短路径的贪心算法。

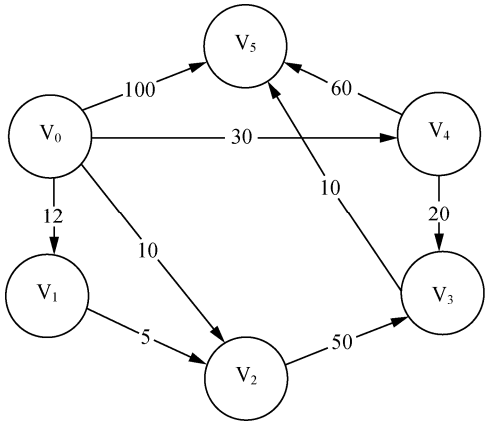

图 4-18　单源最短路径的示例 G_{13}

表 4-1　图 G_{13} 中 v_0 到其余各顶点的最短路径及长度

源点	终点	最短路径	路径长度
v_0	v_1	v_0->v_1	12
	v_2	v_0->v_2	10
	v_3	v_0->v_4->v_3	50
	v_4	v_0->v_4	30
	v_5	v_0->v_4->v_3->v_5	60

设图的邻接矩阵为 W。Dijkstra 算法首先将图的顶点集合划分成两个集合 S 和 V-S。集合 S 表示最短路径已经确定的顶点集合，其余的顶点则存放在另一个集合 V-S 中。初始状态时，集合 S 只包含源点，即 $S=\{s\}$，表示此时只有源点到自己的最短路径是已知的。设 v 是 V 中的某个顶点，把从源点 s 到顶点 v 且中间只经过集合 S 中顶点的路径称为从源点到顶点 v 的特殊路径，并用数组 D 来记录当前所找到的从源点 s 到每个顶点的最短特殊路径长度，用数组 Path 来记录到达各个顶点的前驱顶点。其中，如果从源点 s 到顶点 c 有弧，则以弧的权值作为 D[v]的初始值；否则将 D[v]初始为无穷大。Path 数组初始化为 s。

Dijkstra 算法每次从尚未确定最短路径长度的集合 V-S 中取出一个最短特殊路径长度最小的顶点 u，将 u 加入集合 S，同时更新数组 D、Path 中由 s 可达的各个顶点的最短特殊路径长度。更新 D 的策略是，若加进 u 做中间顶点，使得 v_i 的最短特殊路径长度变短，则修改 v_i 的最短特殊路径长度及前驱顶点编号，即当 $D[u]+W[u,v_i]<D[v_i]$ 时，令 $D[v_i]=D[u]+W[u,v_i]$，$path[v_i]=u$。重复上述操作，一旦 S 包含了 V 中所有的顶点，D 记录了从源点 s 到各个顶点的最短路径长度，Path 记录了相应最短路径的终点的

前驱顶点编号。

例如，对于图 4-18 所示的有向图 G_{13}，为方便计算，可以先得到其带权邻接矩阵，如图 4-19 所示，应用 Dijkstra 算法计算从源点 v_0 到其他顶点的最短路径的过程如表 4-2 所示。

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

图 4-19　图 G_{13} 的邻接矩阵

在表 4-2 中，初始时集合 S 只包含源点 v_0，通过邻接矩阵可以很容易地列出源点到每个顶点的最短特殊路径长度，Dijkstra 算法首先从集合 V-S 中取出一个最短特殊路径长度最小的顶点 v_2 加入集合 S，$D[v_2]=10$（表中用下划线标记），然后更新源点到其他顶点的最短特殊路径长度，例如，v_0 到 v_3 的最短路径(即 $D[v_3]$)原来为无穷大，当加入新顶点 v_2 时，通过邻接矩阵得到 $W[v_2,v_3]=50$，并且 $D[v_3]>D[v_2]+W[v_2,v_3]$，于是按照 Dijkstra 的思想更新 $D[v_3]$ 为 $D[v_3]=D[v_2]+W[v_2,v_3]=10+50=60$，同时更新 $Path[v_3]=v_2$。依此类推，直至确定了源点 s 到所有其余顶点的最短路径，即所有点都在集合 S 中。

表 4-2　Dijkstra 算法的处理过程，源点为 v_0

S / 顶点		$\{v_2\}$	$\{v_2,v_1\}$	$\{v_2,v_1,v_4\}$	$\{v_2,v_1,v_4,v_3\}$	$\{v_2,v_1,v_4,v_3,v_5\}$
v_1	12	<u>12</u>				
v_2	<u>10</u>					
v_3	∞	60	60	<u>50</u>		
v_4	30	30	30			
v_5	100	100	100	90	<u>60</u>	
最短路径	v_0v_2	v_0v_1	v_0v_4	$v_0v_4v_3$	$v_0v_4v_3v_5$	
新顶点	v_2	v_1	v_4	v_3	v_5	
路径长度	10	12	30	50	60	

经过表 4-2 的处理，可以很直观地看出顶点 v_0 到图中其他顶点的最短路径及最短路径的长度。

例 4.11 给出了 Dijkstra 算法的具体实现。

【例 4.11】Dijkstra 算法的实现。

```
    /*Dijkstra 算法，其中参数 s 是源点，D 是最短路径长度，Path[i]为路径上 i 的前驱顶点编号*/
    template <class EdgeType>
    void Dijkstra(Graph<EdgeType> &G,int s,EdgeType D[],int Path[])
    {
        int n=G.VerticesNum();
        int i,j;
        for(i=0;i<n;i++)
        {
            G.Mark[i] = UNVISITED;
            D[i]=INFINITY;
```

```
                Path[i]=-1;      //标记此时不存在 s 到 i 的路径
            }
            G.Mark[s]=VISITED;
            D[s] =0;
            Path[s]=s;

           for(i=0;i<n;i++)
            {
             //找到一条最短特殊路径,即 min{D[j]|G.Mark[j]==UNVISITED, 0<=j<n}
               EdgeTyep min = D[0];
               int k = 0;
               for(j=1;j<n;j++)
              {
                if(G.Mark[j]==UNVISITED && min>D[j])
                 {
                     min = D[j];
                     k = j;
                 }
               }
               //已确定 s 到 k 的最短路径
               G.Mark[k]=VISITED;
             //利用 k 更新到其余未访问顶点的最短特殊路径
              for(Edge<EdgeType> e = G.FirstEdge(k); G.IsEdge(e); e =
G.NextEdge(e))
               {
                   int endVertex = e.end;
                   if(G.Mark[endVertex] ==UNVISITED && D[endVertex]>(D[k]+
e.weight))
                   {//更新到 endVertex 的最短特殊路径
                       D[endVertex]=D[k]+ e.weight;
                       Path[endVertex] =k;
                   }
               }
            }
        }
```

分析迪杰斯特拉（Dijkstra）算法，主要包括两个操作：通过简单选择法来确定最短的特殊路径，时间复杂度为O(n)（如果用最小堆，则时间复杂度为$O(\log_2 n)$）；利用新确定的最短路径终点来确定到其余顶点的最短特殊路径，这个操作要遍历该终点相关联的各条边，最多检查 n−1 次，时间复杂度为 O(n)。因此迪杰斯特拉（Dijkstra）算法的时间复杂度为$O(n^2)$（如果用最小堆，则总的时间复杂度为$O(n\log_2 n)$）。Dijkstra 算法是在假定边权为非负的情况下设计的，如果存在负边权，那么 Dijkstra 算法不能正确运行。

4.5.2　顶点对之间的最短路径

顶点对之间的最短路径问题指的是图 $G=(V,E)$ 中任意的顶点对 $<v_i,v_j>$ 之间的最短路径。解决这个问题的一种方法是，分别以图上的各个顶点为源点，重复执行 n 次 Dijkstra 算法，从而计算出任意两点之间的最短路径，时间复杂度为 $O(n^3)$。另外一种计算顶点对之间最短路径的算法是下面介绍的弗洛伊德（Floyd）算法。

弗洛伊德（Floyd）算法是一种动态规划算法，即先自底向上分别求解子问题的解，然后由子问题的解得到原问题的解。弗洛伊德（Floyd）算法形式比较简单，时间复杂度仍然是 $O(n^3)$。

设图 G 的顶点集 $V=\{v_1,v_2,...,v_n\}$。Floyd 算法的基本思想是，如果 v_i 与 v_j 两点之间的最短路径经过一个或多个中间点，则可以认为这条最短路径由两条最短路径 $A_k=\{v_i=v_{i0},v_{i1},...,v_{im}=v_k\}$，$B_k=\{v_k=v_{j0},v_{j1},...,v_{jp}=v_j\}$ 连接而成，并且 i1，i2，…，im，j1，…，jp 均不大于 k（$1\leqslant k\leqslant n$）。Floyd 算法就要先分别确定 v_i 到 v_k、v_k 到 v_j 且中间经过的顶点编号小于 k 的最短路径，再考查路径 $\{A_k,B_k\}$ 的长度是否是 v_i 到 v_j 且中间经过的顶点编号小于等于 k 的最短路径。

Floyd 算法具体的求解过程是，定义 $adj^{(k)}$ 矩阵，元素 $adj^{(k)}[i,j]$ 描述从 v_i 到 v_j 且中间顶点编号不大于 k 的最短路径长度；定义 $path^{(k)}$ 矩阵，元素 $path^{(k)}[i,j]$ 描述从 v_i 到 v_j 且中间顶点编号不大于 k 的最短路径中 v_j 的前驱顶点编号。初始时，定义 $adj^{(0)}$ 为相邻矩阵，即任意两点之间不经过任何其他顶点。在矩阵 $adj^{(0)}$ 上做 n 次迭代，循环地产生一个矩阵序列 $adj^{(1)}$，…，$adj^{(k)}$，…，$adj^{(n)}$。这个循环地产生 $adj^{(1)}$，…，$adj^{(k)}$，…，$adj^{(n)}$ 的过程就是逐步允许越来越多的顶点作为路径的中间顶点，直到所有顶点都允许作为中间顶点，最短路径也就出来了。

上述求解过程中，假设已求得矩阵 $adj^{(k-1)}$，那么从顶点 v_i 到顶点 v_j 中间顶点的编号不大于 k 的最短路径有两种情况，一是中间不经过顶点 v_k，那么就有 $adj^{(k)}[i,j]=adj^{(k-1)}[i,j]$；另一种是中间经过顶点 v_k，那么 $adj^{(k)}[i,j]<adj^{(k-1)}[i,j]$，所以由顶点 v_i 经过 v_k 到顶点 v_j 的中间顶点编号不大于 k 的最短路径就分解成两个子路径问题：一段是从顶点 v_i 到 v_k 的中间顶点编号不大于 k−1 的最短路径；一段是从顶点 v_k 到 v_j 的中间顶点编号不大于 k−1 的最短路径，路径长度应为这两段最短路径长度之和，即：

$$adj^{(k-1)}[i,k]+adj^{(k-1)}[k,j]$$

综合这两种情况有：

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

例如，用 Floyd 算法求解图 4-20 中每对顶点间的最短路径的过程如图 4-21 所示。

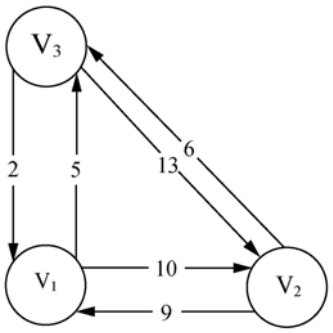

图 4-20　顶点对间的最短路径示例

$$adj^{(0)}=\begin{bmatrix}0&10&5\\9&0&6\\2&13&0\end{bmatrix}\quad path^{(0)}=\begin{bmatrix}1&1&1\\2&2&2\\3&3&3\end{bmatrix}$$

$$adj^{(1)}=\begin{bmatrix}0&10&5\\9&0&6\\2&12&0\end{bmatrix}\quad path^{(1)}=\begin{bmatrix}1&1&1\\2&2&2\\3&1&3\end{bmatrix}$$

$$adj^{(2)}=\begin{bmatrix}0&10&5\\9&0&6\\2&12&0\end{bmatrix}\quad path^{(2)}=\begin{bmatrix}1&1&1\\2&2&2\\3&1&3\end{bmatrix}$$

$$adj^{(3)}=\begin{bmatrix}0&10&5\\8&0&6\\2&12&0\end{bmatrix}\quad path^{(3)}=\begin{bmatrix}1&1&1\\3&2&2\\3&1&3\end{bmatrix}$$

图 4-21　Floyd 算法的求解过程

例 4.12 给出了求顶点对之间最短路径的 Floyd 算法的实现。

【例 4.12】Floyd 算法的实现。

```
template <class EdgeType>
void Floyd(Graph<EdgeType>& G,EdgeType **Adj,int **Path)
{
    int i,j,v;                    //i,j 是计数器，v 记录相应顶点
   int n = G.VerticesNum();
    for(i=0;i<n;i++)              //初始化 D 数组,Path 数组
    {
        for(j=0;j<n;j++)
        {
            if(i==j)
            {Adj[i][j]=0;Path[i][j]=i;}
            else
            { Adj[i][j]=INFINITY; Path[i][j]=-1;}
        }
    }
    for(v=0;v<n;v++)
    {//检查各条边,将边(v,u)的值作为 Adj[v,u],v 作为 Path[u]的值
        for(Edge<EdgeType> e=G.FirstEdge(v);G.IsEdge(e);e=G.NextEdge
(e))
        {
            Adj[v][e.end]=G.Weight(e);
        }
    }
  //如果两个顶点 i,j 间的最短路径经过顶点 v，且有 Adj[i][j]>(Adj[i][v]+
  //Adj[v][j])，则更新 Adj[i][j],Path[i][j]
    for(v=0;v<n;v++)
        for(i=0;i<n;i++)
            for(j=0;j<n;j++)
```

```
                if(Adj[i][j]>(Adj[i][v]+Adj[v][j]))
                {
                    Adj[i][j]=Adj[i][v]+Adj[v][j];
                //更新 i，j 之间经过的点编号不超过 v 的最短路径长度
                    Path[i][j]=v;
                //更新 i，j 之间经过的点编号不超过 v 的最短路径上 j 的前驱
                }
    }
```

弗洛伊德（Floyd）算法利用三层 for 循环来得到任意两点之间的最短路径，显然时间复杂度为$O(n^3)$。

4.6　拓 扑 排 序

在实际问题中，经常会遇到判断一个有向图中是否存在环的问题。例如，在计算机专业学生的部分课程结构（表 4-3）中，有些课程是基础课程不需要其他课程作为基础，而有些课程必须在学完先修课程后才可以开始。先修课程描述了课程之间的优先关系。可以用有向图表示这种优先关系，如图 4-22 所示。图中的顶点表示课程，有向边表示课程之间先修关系。如课程 C1 是 C2 的先修课程，则在图 4-22 中，存在 C1 到 C2 的一条有向边。显然在这样的有向图中不能存在环。

表 4-3　某大学计算机专业部分课程结构

课程编号	课程名称	先修课程
C1	C 语言程序设计	无
C2	离散数学	C1
C3	数据结构	C1，C2
C4	计算机组成与结构	无
C5	编译原理	C3
C6	操作系统	C3,C4

对于无向图，深度优先搜索过程中如果在当前访问的顶点和已经访问过的顶点之间存在一条边，且这条边不是前向边，则可以判断图中一定存在环。如图 4-23 所示，从顶点 v_1 开始深度优先搜索时，首先从 v_1 访问到 v_2，接着从 v_2 访问到 v_3 时，存在 v_3 到已经访问的顶点 v_1 之间的一条非前向边（v_3 到已经访问的顶点 v_2 之间的边就是前向边），从而可以判定该图存在环。而对于有向图，利用深度优先搜索有时不能得到正确的判定结果。如对图 4-20，首先从顶点 C1 开始进行深度优先搜索，则依次访问 C1，C2，C3，C5，C6，接着将从 C4 开始访问，而在 C4 与已经访问的顶点 C6 之间存在有向边（非前向边），但是显然图中并不存在环。

拓扑排序是对有向图中顶点进行的一种排序，根据排序结果可以判断有向图中是否存在环。

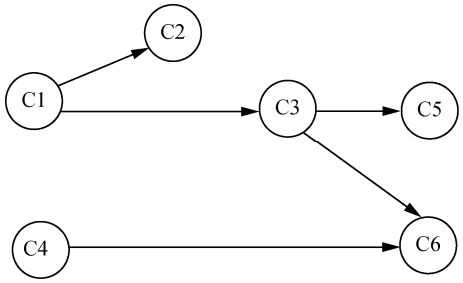

图 4-22　边表示优先关系的有向图

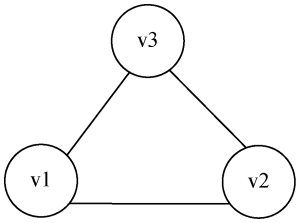

图 4-23　利用深度优先搜索判断无向图中是否存在环

在有向图 G 中，若<v_i,v_j>是一条弧，则称 v_i 为 v_j 的前驱顶点，v_j 为 v_i 的后继顶点。对有向无环图 G 中的所有顶点进行一个排序，如果满足：若 v_i 是 v_j 的前驱顶点，在序列中 v_i 必在 v_j 之前，则称这样的排序为拓扑排序（Topological Sorting），对应的顶点序列为拓扑序列（Topological Order）。序列{C1，C2，C3，C5，C4，C6}、{C4，C1，C2，C3，C5，C6}都是图 4-22 的拓扑序列，显然拓扑序列可能不唯一。

一个简单的拓扑排序算法是先找出任意一个没有入边的顶点（即入度为 0），然后输出该顶点，并从图中删除该顶点和由它指出的边，修正其余顶点的入度信息。然后对图的剩余部分应用同样的方法处理。当图中找不到没有入边的顶点时，如果所有的顶点已经访问完，则图中不存在环，否则存在环。

如图 4-24（a）所示，首先选择没有入边的顶点 A，输出 A 并删除 A 以及由 A 指出的边后，只有 B 的入度为 0，输出 B 后删除 B 以及由 B 指出的边；此时只有 C 没有入边，输出 C 后删除 C 以及由 C 指出的边，最后输出 D。而对图 4-24（b），首先选择没有入边的顶点 A，输出 A 并删除 A 以及由 A 指出的边后，找不到入度为 0 的顶点，而此时还有顶点没有输出，从而判定存在环。拓扑排序过程中选择没有入边的顶点时如果有多种选择，则意味着可以产生多种拓扑排序结果。

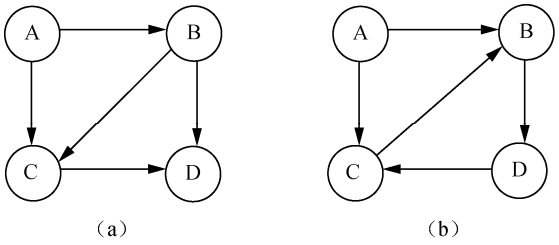

图 4-24　利用拓扑排序判断有向图中是否存在环

上述拓扑排序算法的核心操作是不断修正各个顶点的入度信息，为此在算法实现时，可以设计一个数组来表示各个顶点的入度信息，数组元素的下标表示相应的顶点序号。当删除一个顶点时，就需要检查由该顶点发出的边，将边的终点的入度值减 1。具

体的算法实现如例 4.13 所示。

【例 4.13】拓扑排序算法。

```
template <class EdgeType>
bool TopologySort (Graph<EdgeType>& G,int* SortArray)
{//对图 G 进行拓扑排序,将序列存放在数组 SortArray 中
    int n = G.VerticesNum(); //记录图中的顶点个数
    int* indegree = new int[n];  //创建一个数组记录各个顶点的入度值
    int v;
    Edge<EdgeType> e;
    for(v=0;v<n;v++)
    {//各个顶点的入度初始化 0,访问状态标记为未访问
        indegree[v]=0; G.Mark[v]= UNVISITED;
    }
    for(v=0;v<n;v++)
    { //统计各个顶点的入边信息
        for( e=G.FirstEdge(v);G.IsEdge(e);e=G.NextEdge(e))
        {
            indegree[v.end]++;
        }
    }
    for(int i=0;i<n;i++)
    {//依次确定拓扑序列 SortArray 中的第 i 个元素
        //找到入度为 0 且未被访问的顶点
        for(v=0;v<n;v++)
        {
        if(indegree[v]==0&&G.Mark[v]==UNVISITED)
        {
            break; //退出 for(v)循环
        }
        }
        if(v==n) {return false;}   //找不到入度为 0 的顶点,退出拓扑排序
        //将顶点 v 放到排序序列中,并将其状态设置为 VISITED
       G.Mark[v] = VISITED;
        SortArray[i]=v;
        for(e=G.FirstEdge(v);G.IsEdge(e);e=G.NextEdge(e))
        {//修改 v 指向的顶点的入度
            indegree[v.end]--;
        }
    }
  delete []indegree;
    return true;
}
```

分析拓扑排序算法，对于具有 n 个顶点、e 条边的有向图而言，初始化各个顶点的入度信息的时间复杂度是 O(e)，循环找到入度为 0 的顶点并修正其余顶点的入度信息的

时间复杂度是 $O(n^2+e)$，因此总的时间复杂度是 $O(n^2+e)$。

另外一种拓扑排序算法是先获得逆拓扑序列，然后再形成拓扑序列。要获得逆拓扑序列，可以先找出任意一个没有出边的顶点（即出度为 0），从图中删除该顶点和指向它的边，修正其余顶点的出度信息。然后对图的剩余部分应用同样的方法处理。这个过程也成为逆拓扑排序。

4.7 关 键 路 径

在有向无环图中，可以用顶点表示事件（Event），边表示活动（Activity），边的权值表示活动所需要的时间，边的方向表示活动可以在起点事件之后开始，在终点事件之前完成。这样的有向无环图也成为 AOE 网。在 AOE 网中只有一个入度为 0 的顶点（源点），一个出度为 0 的顶点（汇点）。实际应用中，通常用 AOE 网来估算项目的完成时间。如图 4-25 为一个包含 8 项活动、7 个事件的 AOE 网。在 AOE 网中，完成工程的最短时间是从源点到汇点的最长路径的长度，该路径称为关键路径。关键路径上的活动称为关键活动。在项目管理中，关键路径决定了整个项目的最短完成时间。如图 4-25 所示，边上数值表示相应的活动需要多少天完成，图中的路径 $v_1->v_2->v_4->v_5->v_7$ 是 v_1 到 v_7 的最长的路径，即关键路径，表明从 v_1 事件开始到 v_7 事件最快需要 18 天。

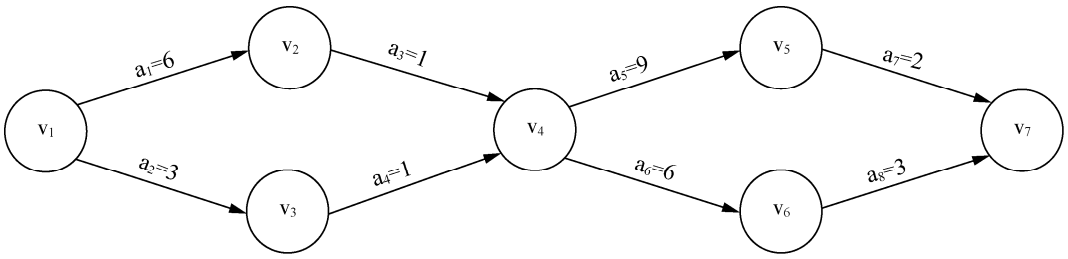

图 4-25　关键路径示例

为了计算 AOE 网中关键路径，首先引入以下几个概念：

$ve(v_i)$，表示事件 v_i 的最早发生时间，例如在图 4-25 中，$ve(v_1)=0$，$ve(v_3)=3$。

$vl(v_i)$，表示事件 v_i 的最迟发生时间，例如在图 4-25 中，$vl(v_1)=0$，$vl(v_3)=6$。

$e(a_i)$，表示活动 a_i 的最早开始时间，例如在图 4-25 中，$e(a_1)=0$，$e(a_2)=0$。

$l(a_i)$，表示活动 a_i 的最迟开始时间，例如在图 4-25 中，$l(a_1)=0$，$l(a_2)=3$。

$dut(j,k)$，表示活动 a_i 的持续时间。

$l(a_i)-e(a_i)$，表示完成活动 a_i 的时间余量。

$l(a_i)=e(a_i)$ 的活动叫做关键活动；显然，关键路径上的活动都是关键活动。

在计算关键路径的过程中，事件与活动有着密切的关系，设活动 a_i 关联的前后事件分别是 v_j 和 v_k，如图 4-26 所示，则有

$e(a_i)=ve(v_j)$ 且 $l(a_i)=vl(v_k)-dut(j,k)$。

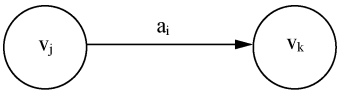

图 4-26　活动 a_i 与其关联的事件

要计算关键路径，就要先确定关键活动。而确定哪些活动是关键活动，则需要清楚各个活动的最早开始和最晚开始时间，即该活动所关联的起点事件的最早发生时间和终点事件的最迟发生时间。

求解事件的最早发生时间需要考虑该事件的前驱事件，例如图 4-25 中，有 $ve(v_2)=ve(v_1)+6$，$ve(v_3)=ve(v_1)+3$，但在求解 $ve(v_4)$ 时，有两种情况，一种是 $ve(v_4)=ve(v_2)+1$，另一种是 $ve(v_4)=ve(v_3)+1$，该如何取舍呢？根据实际工程经验可知，事件 v_4 发生的前提是事件 v_2 与 v_3 全部结束之后，所以事件 v_j 的最早是在所有前驱事件及活动都完成后发生，即

$ve(v_j)=Max\{ve(v_i)+dut(i,j)\}$，其中 v_i 是 v_j 的前驱事件。

类似的道理，在求解事件的最迟发生时间需要考虑它的后继事件，如图 4-25，可以得到 $vl(v_5)=vl(v_7)-2$，$vl(v_6)=vl(v_7)-3$，计算 $vl(v_4)$ 时依然有两种不同的选择，但是事件 v_4 最迟也不能耽误其后继事件，因此

$vl(v_i)=Min\{vl(v_j)-dut(i,j)\}$，其中 v_j 是 v_i 的后继事件。

计算出各个事件的最早发生和最迟发生时间后，就可以计算出各个活动的最早开始和最晚开始时间，从而确定关键路径。

求解关键路径的具体算法如下：

1）从源点 v_0 出发，令 $ve(v_0)=0$，利用拓扑排序过程顺序计算各事件的最早发生时间 $ve(v_i)$。

2）从汇点 v_n 出发，令 $vl(v_n)=ve(v_n)$，利用逆拓扑排序过程计算各事件的最迟发生时间 $vl(v_i)$。

3）根据各事件的 $ve(v_i)$ 和 $vl(v_i)$，计算各活动的最早开始时间 $e(a_j)$ 和最迟开始时间 $l(a_j)$。

4）$l(a_j)=e(a_j)$ 的活动叫做关键活动。关键活动组成的路径即为关键路径。

如计算工程图 4-25 中的关键路径，首先计算各个事件的最早发生和最迟发生时间如表 4-4 所示。然后计算出各个活动的最早开始和最迟开始时间如表 4-5 所示。

表 4-4　事件的最早最迟发生时间

事件	v_1	v_2	v_3	v_4	v_5	v_6	v_7
ve	0	6	3	7	16	13	18
vl	0	6	6	7	16	15	18

表 4-5　活动的最早最迟发生时间

活动	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8
e	0	0	6	3	7	7	16	13
l	0	3	6	6	7	9	16	15

由表 4-5 中可以确定 a_1，a_3，a_5，a_7 是关键活动，所以该工程的关键路径为 a_1，a_3，a_5，a_7。

根据上面的讨论分析可知，计算关键路径问题实质上就是要计算出各个活动的最早开始和最迟开始时间。假定图采用邻接矩阵存储，并且图中边的权值为整数。修改拓扑排序后可以用来计算各个顶点事件的最早发生时间，从而计算出各个活动的最早开始和最迟开始时间。具体如例 4.14、例 4.15 所示。

【例 4.14】修改拓扑排序算法，计算出各个顶点事件的最早发生时间。

```
bool ModifyTopSort(Graph<int>& G,int* VE,int* SortArray)
{//对图 G 利用拓扑排序,获得各个顶点事件的最早发生时间,
  //拓扑序列存放在数组 SortArray 中
    int n = G.VerticesNum();          //记录图中的顶点个数
    int* indegree = new int[n];       //创建一个数组记录各个顶点的入度值
    int v;
    Edge<int> e;
    for(v=0;v<n;v++)
    {//各个顶点的入度初始化 0,访问状态标记为未访问
        indegree[v]=0; G.Mark[v]= UNVISITED;
        VE[v]=0;
    }
    for(v=0;v<n;v++)
    { //统计各个顶点的入边信息
        for( e=G.FirstEdge(v);G.IsEdge(e);e=G.NextEdge(e))
        {
            indegree[v.end]++;
        }
    }
    for(int i=0;i<n;i++)
    {//依次确定拓扑序列 SortArray 中的第 i 个元素
        //找到入度为 0 且未被访问的顶点
        for(v=0;v<n;v++)
        {
            if(indegree[v]==0&&G.Mark[v]==UNVISITED)
            {
                break; //退出 for(v)循环
            }
        }
        //找不到入度为 0 的顶点,退出拓扑排序
        if(v==n) {return false;}
        //将顶点 v 放到排序序列中,并将其状态设置为 VISITED
```

```
            G.Mark[v] = VISITED;
            SortArray[i]=v;
            for(e=G.FirstEdge(v);G.IsEdge(e);e=G. NextEdge(e))
            {//修改 v 指向的顶点的入度
                indegree[e.end]--;
                //修改 v 的后继事件的最早发生时间
              if(VE[e.end]<VE[v]+e.weight)
                    VE[e.end] = VE[v]+e.weight;
            }
        }
       delete []indegree;
        return true;
    }
```

在例 4.14 基础上，确定关键活动的算法实现如代码例 4.15 所示。

【例 4.15】确定关键活动。

```
bool CriticalPath(Graph<int>& G)
{//计算图 G 中关键活动并返回真，如果不是 AOE 网，则返回 false
    int n = G.VerticesNum(); //记录图中的顶点个数
    int v;
    int* VE = new int[n];    //记录各个顶点事件的最早发生时间
    int* VL = new int[n];    //记录各个顶点事件的最迟发生时间
    int* TopOrder = new int[n];  //记录拓扑序列
    if(!ModifyTopSort(G,VE,TopOrder))
    {//获得图的拓扑排序及各个顶点事件的最早发生时间
     return false;  //存在环,不能计算关键活动
    }
    //各个顶点事件的最迟发生时间初始成各个顶点事件的最早发生时间
    for(i=0;i<n;i++)
        VL[i]=VE[i];

    for(i=n-1;i>=0;i--)
    { //利用逆拓扑序列来计算各个顶点的最迟发生时间
        for(int j=0;j<n;j++)
       {//如果存在边<j,i>,且 VL[j]>VL[i]-dut[j,i],则更新 VL[j]
            if(G.Matrix[j][i]!=0&&G.Matrix[j][i]<INFINITY)
            {
                dut = G.Matrix][j][i];
                if(VL[j]>Vl[i]-dut)
                    VL[j]=VL[i]-dut;
            }
       }
    }
    for(v=0;v<n;v++)
    {//确定关键活动
        for(Edge<int> e=G.FirstEdge(v);G.IsEdge(e);e=G.NextEdge(e))
```

```
            {
                int u = e.end;
                int dut = e.weight;
                if(VE[v]==VL[u]-dut)//确定关键活动并输出
                    cout<<'<'<<v<<','<<u<<')'<<VE[v]<<' '<<VL-dut<<endl;
            }
        }
        delete []VE;
      delete []VL;
        return true;
    }
```

确定关键活动的算法需要计算各个顶点事件的最早发生和最迟发生时间。如果图采用邻接矩阵存储，则算法的时间复杂度为$O(n^2)$,如果采用邻接表来存储，则时间复杂度为$O(n+e)$。

实际应用中，在不改变网络结构的前提下（不改变关键路径），通常采用提高关键活动的速度来缩短源点到汇点的工期；但如果 AOE 网中有多条关键路径，则必须提高各条关键路径共有的关键活动的速度才能缩短工期。

4.8 图 的 应 用

4.8.1 图的存储和遍历：地图染色应用的实现

1．需求描述

地图四色定理（Four color theorem）最先是由一位叫古德里（Francis Guthrie）的英国大学生提出来的。四色问题的内容是：“任何一张地图只用四种颜色就能使具有共同边界的国家着上不同的颜色。”如果用数学语言表示，就是指“将平面任意地细分为不相重叠的区域，每一个区域总可以用 1，2，3，4 这四个数字之一来标记，而不会使相邻的两个区域得到相同的数字。”这里所指的相邻区域，是指有一整段边界是公共的。如果两个区域只相遇于一点或有限多点，是不相邻的，用相同的颜色给它们着色不会引起混淆。该问题经过反反复复近 100 年的证明，在 1976 年，美国数学家阿佩尔（K.Appel）与哈肯（W.Haken）宣告借助电子计算机获得了四色定理的证明。

基于四色定理，本应用要在给定地图和颜色序号的前提下，计算出一种地图染色方案。具体要求如下：

1）用户输入地图中区域的个数 N，限定区域数目 N<=50。

2）用户输入邻接区域的代码，区域代码为 1～N，邻接关系只需输入一次，例如 1 3，输入“End”作为结束标志。

3）输出时，采用一一对应的方法，一个区域对应一种颜色形式：区域代码=>颜色

代码=>颜色。

4）本程序可为任意一张的地图染色，并且至多染四种颜色。

2．问题分析

本应用是图的深度优先遍历的一种变形，通过相邻结点不能有相同颜色的约束条件，来对遍历路径进行约束，如果一条路径上出现了走不通的状况，则进行回溯操作，程序结构如图 4-27 所示。

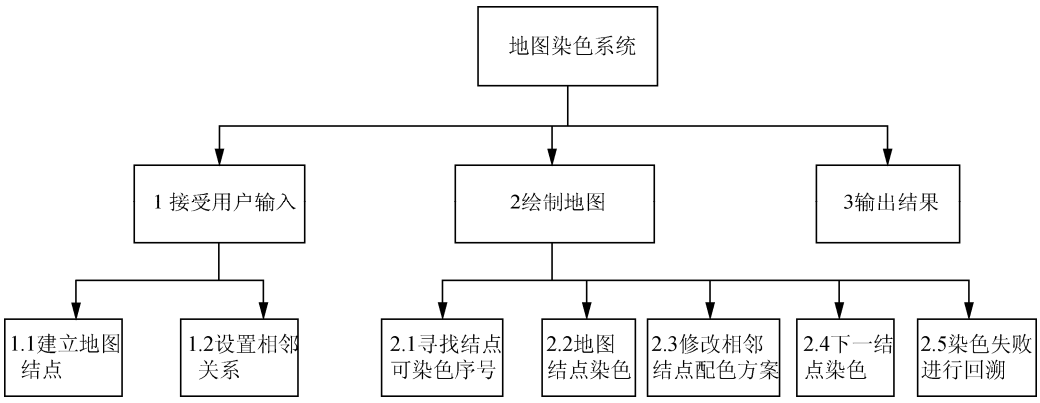

图 4-27　地图染色系统功能结构

绘制地图时，从初始结点开始随机选取一种颜色，然后向后遍历，下一个结点选色的时候，不能选取已经确定颜色的邻居结点的颜色。所以可以考虑在某一结点选取颜色后，立刻更新其邻居结点的可选颜色的集合，然后继续向下遍历，如果出现某一个结点中的所有可选颜色为空的时候，说明前面的配色方案不正确，进行回溯操作，如果回溯一步之后发现所有的配色方案仍不能满足，则继续回溯。直到找到一种能够让最后一个遍历的结点颜色可选方案不为空的时候，随机选取其中一种颜色作为最后的输出结果即可。本应用只要求输出一种可行的染色方法，实际上也可以输出所有的染色方案。

3．概要设计

采用邻接矩阵存储图的结构，即以二维数组 Neighbours[N][N]表示地图区块之间的相邻关系，N 表示区域数目，数组中以元素值为 0 表示不邻接，1 表示邻接，限定区域数目 N≤50。

类的关系如图 4-28 所示。地图区块类 Block，描述地图区块对象，该类的主要成员有用于表征可选的颜色方案数组 Colors，用于记录回溯数组的成员 previousColors，以及区块 ID 和该区块的最终着色方案 colorId。该类主要实现的程序逻辑是修改邻居结点的配色方案（TellNeighbours），检测当前结点配色方案是否为空（NoRemainingColour），以及结点配色方案的回溯操作（RecoverNeighbours）。地图类 Map，其成员为一个以

Block 类为元素的链表类，用于表征地图上的所有区块。该类主要用于实现程序逻辑，即初始化（Initialize），地图着色（DrawMap）以及结果的输出（showResult）。

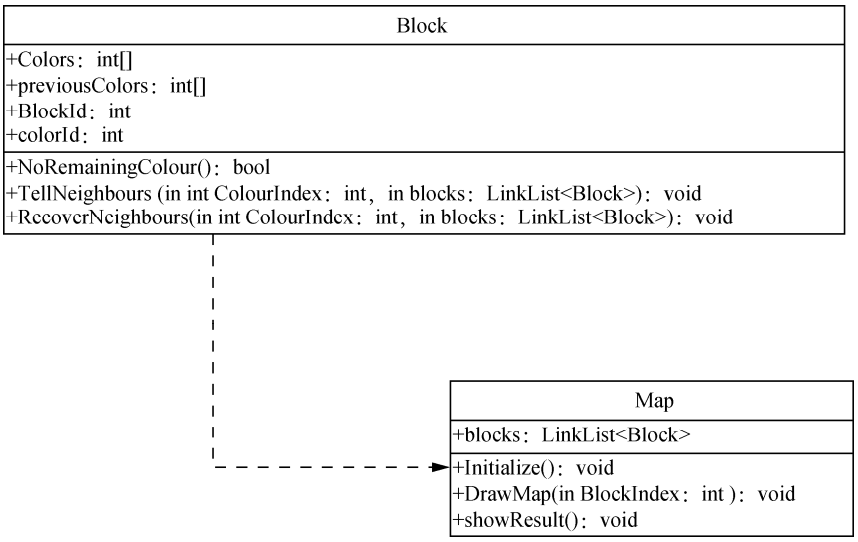

图 4-28　地图染色系统类图

地图染色过程实质上就是对图的深度优先遍历过程。在遍历过程中，为每一个访问到的区块选择一种可选的染色方案，同时更新其相邻的未处理的区块的染色可选方案；如果对一个区块进行方案选择的时候，发现没有可选方案，也就是该区块 Colors 中的所有数组元素的值均为 0，则表示之前的染色方案出了问题，进行回溯还原，采用其他颜色进行染色之后，继续遍历，直到最后一个结点完成选色。

4．详细设计

按照上述的概要设计，程序整体的流程如图 4-29 所示。在具体实现时，首先设置一些基本的参数常量，包括地图染色的颜色种类，区块的数量，以及用于存储相邻关系的矩阵。程序根据用户输入的地图区块数，以及区块之间的相邻关系来初始化地图。

图 4-29　地图染色系统流程

地图的染色过程是一个递归调用的过程，将区块存储于一个线性表中，从第一个区块开始，进行选色遍历，遍历结束的条件是所有区块元素遍历完成，并且选色方案中，至少有一种方案可以使用。当染色过程结束之后，每一个地图区块都会被赋予一个特定的颜色 ID 值，按照本应用的要求，输出地图 ID 和配色 ID 以及最后输出所搭配的颜色。

地图染色方法的具体流程如图 4-30 所示。该方法需要输入初始区块 ID，如果当前处理的区块有可选的配色方案，则选择一个颜色，并更新邻接区块的可选颜色方案，并继续从未处理的邻接区块继续染色。如果当前处理的区块没有可选的配色方案，则该方法回溯到上一个处理的区块重新选色。直到确定最后一块的颜色或者初始区块没有可选颜色时，结束该方法。

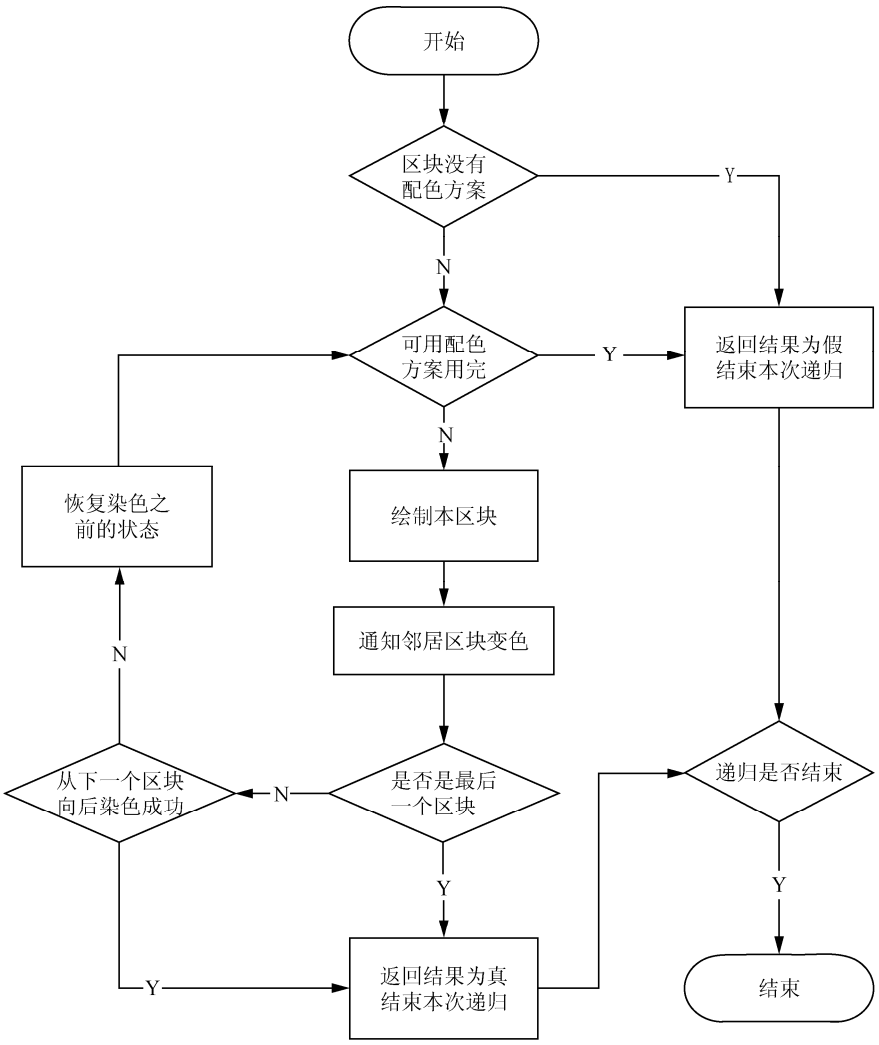

图 4-30　地图染色方法流程

绘制本区块操作包括三个步骤：为该区块选择配色方案，将上一个状态的配色方案保留，将其他染色方案置 0 并备份。

通知邻居区块变色是指一旦一个区块选定配色之后，要遍历所有的相邻区块，将相应的配色方案置 0。需要注意的是，即使上一个区块中的配色方案本身是 0，也要执行

将配色方案置 0 的操作，这样可以更新备份状态，防止其在配色恢复的时候出现错误。

如果当前区块是最后一个区块的话，选定配色方案之后，就代表着染色过程终止了。如果不是最后一个区块，则递归调用染色方法。如果染色失败，那么就要进行回溯操作，将本区块的染色方案，以及邻居区块的被修改的染色方案进行恢复。

地图染色的方案不是唯一的，而且解答方法也不唯一。与地图染色问题类似的应用问题还有八皇后问题，感兴趣的同学可以自己练习。

4.8.2 最小生成树：通信线路铺设问题

1．需求描述

随着网络技术的发展，信息化网络正在逐渐普及，各个城市都会铺设通信线路，使得城市中的区、县都存在通信路径。本应用程序将设计一个铺设费用最优的铺设方案，至少支持以下功能：

1）获取城市的下辖区/县。

2）修改区/县之间的花销。

3）计算最佳铺设方法。

2．问题分析

本应用的目标是给一个城市下辖的区/县铺设通信线路，可以利用本章中学习的图来存储城市中的区、县的距离信息，这里选择使用邻接矩阵来存储。首先，从文件中读取城市下辖的区、县信息（每个区、县包括对应的名称，X 坐标和 Y 坐标），利用获取的区、县信息来构建一个图。

实际生活中，由于区、县之间的地理环境因素，区、县之间的铺设代价不仅与区、县之间的距离有关，还与区、县之间每公里所需的代价有关，两者的乘积对应于图中边的权重。默认情况下区、县之间每公里所需的代价都为 1。应用程序应该提供修改每公里所需代价的接口函数。

本应用最重要的一个方面就是已知地理信息之后自动进行计算，并将结果输出。最佳通信线路铺设问题正好对应于本章中的最小生成树问题，可以利用 Kruskal 算法或者 Prim 算法解决。

3．概要设计

需求描述和问题分析中主要涉及区、县、城市、开销等概念。开销可以用 double 类型的数据来保存，因此本应用中需要有三个类：区、县 County 类、城市 City 类和用于表示通信线路铺设问题的 CabelAllocator 类。

根据上文的分析，区、县类需要包括的成员变量包括区、县名称、区、县所处位置的 X 坐标和 Y 坐标。区、县类需要包括的成员函数至少要有计算两个区、县之间距离

的函数。

城市类继承邻接图类，包括的数据成员是城市中所有的区、县；需要提供完成以下功能的成员函数：根据区、县名称查找区、县对应的 ID，根据区、县的 ID 查找区、县对应的名称，设置两个区、县之间边的权重等。

通信线路铺设类应该包括的数据成员有城市名、城市类的一个指针；包括根据城市的区、县信息来计算最佳线路铺设方案的成员函数。

相关类定义及关系如图 4-31 所示。

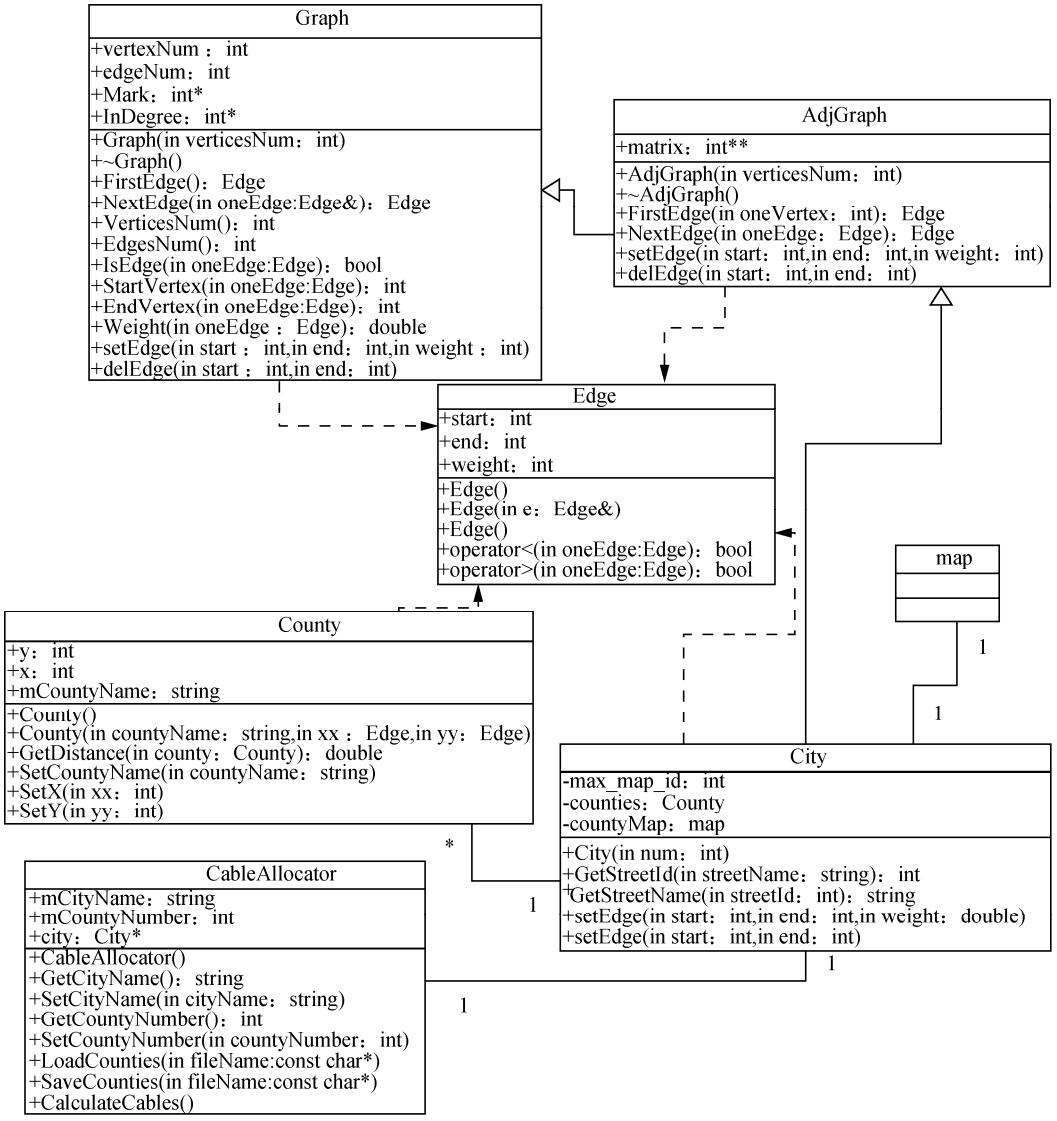

图 4-31　最优通信线路问题的类设计

4．详细设计

本应用中最重要的两个类是 City 类和 CabelAllocator 类。其中 City 继承自邻接图类 AdjGraph；AdjGraph 类继承自抽象类 Graph。

（1）City 类

每个城市都包含若干个区、县。在图中区、县只能以 ID 的形式保存，为此，在城市类中为区、县名称和区、县 ID 之间建立了一一映射，用 STL 的 MAP 来实现。同时提供了两个接口用于实现名称和 ID 之间的相互转换，即根据区、县名称获取区、县 ID，根据区、县 ID 获取区、县名称。

城市类中需要提供设置区、县之间线路铺设代价的函数 setEdge(int start,int end)。默认情况下，两个区、县之间的代价即为两个区、县之间的距离，区、县的 X 坐标和 Y 坐标已经给出。setEdge(int start,int end)的实现流程如图 4-32 所示。

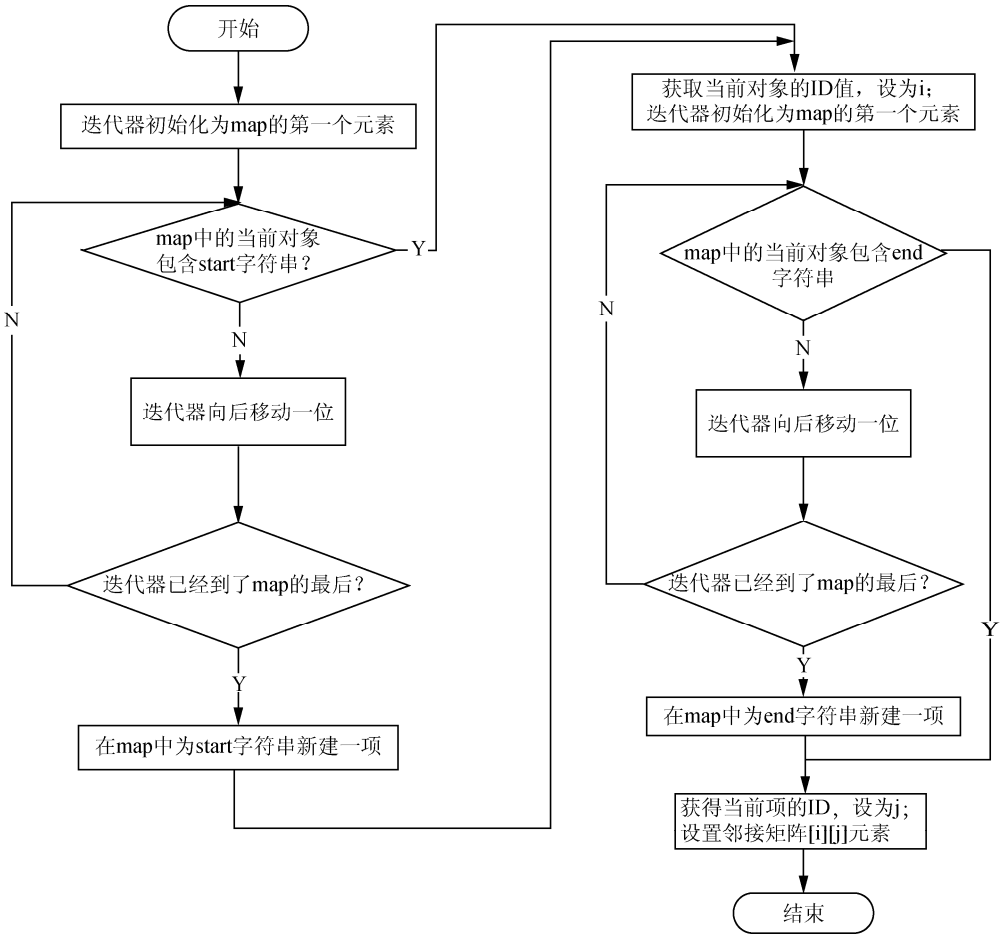

图 4-32　setEdge 方法的流程图

现实生活中线路的铺设代价并不仅仅与距离有关，有时还与地理环境有关系，应用中也需要提供给用户直接设置区、县之间线路铺设代价的接口。

（2）CabelAllocator 类

在 CabelAllocator 类中，最重要的成员函数是加载区、县信息的 LoadCounties()方法和计算最优铺设路线的 CalculateCables()方法。

首先讨论一下加载区、县信息的方法。在文件中，第一行记录代表城市中区、县的个数 N。接下来的 N 行，每行表示一个区、县。包括区、县名称、X 坐标和 Y 坐标。LoadCounties()方法要读取文件中的信息来构建图结构，并且初始化图中任意两个结点的距离，具体流程如图 4-33 所示。

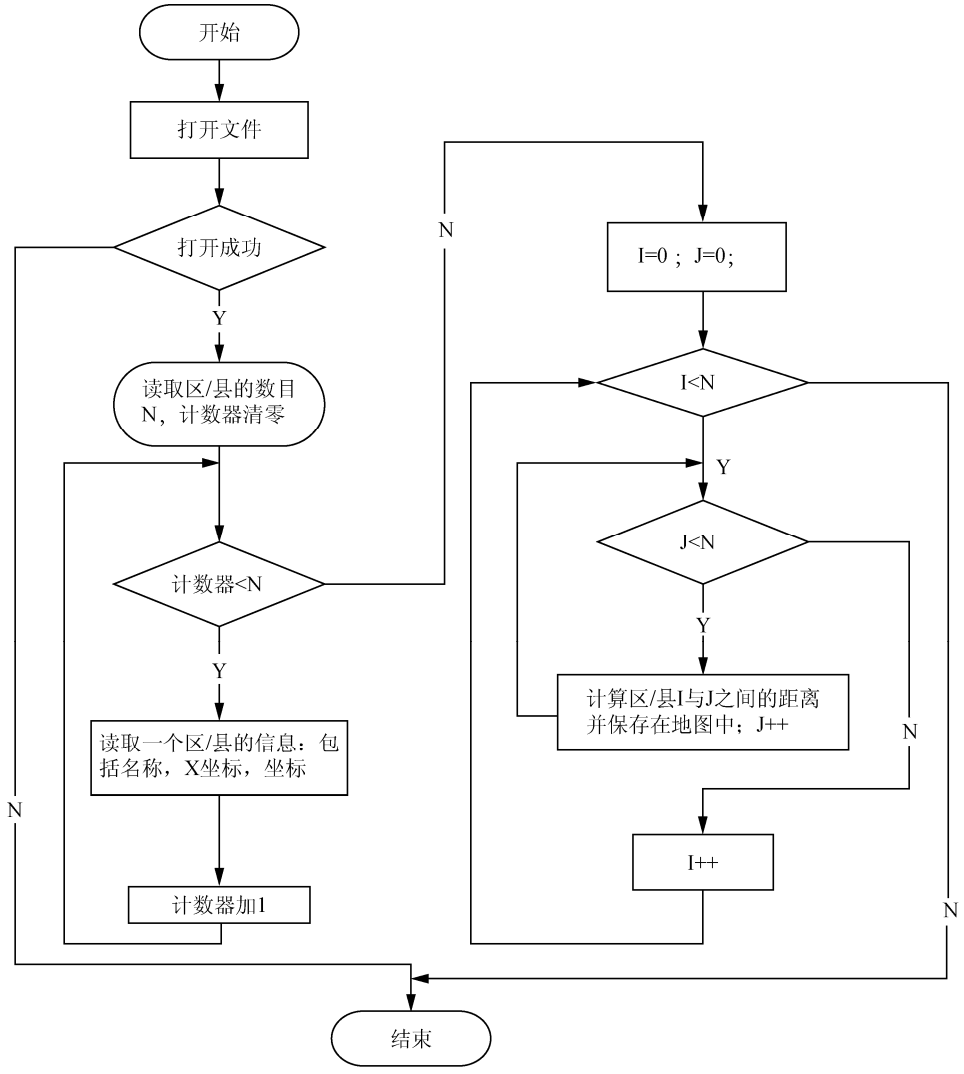

图 4-33　LoadCounties 方法的流程图

计算区/县之间的距离时调用的是 County::GetDistance()方法，函数的实现如下：

```
double Country<double>::GetDistance(County<double> county)
{
    return sqrt((county.x - x)*(county.x - x)  +  (county.y - y)*(county.y
- y));
}
```

计算最佳通信线路铺设问题就是利用 Prim 算法或者 Kruskal 算法生成最小生成树，最小生成树中被选中边就是要进行铺设的通信线路。关于 Prim 算法和 Kruskal 算法参考 4.4 节的内容。

4.8.3 最短路径：指定时间内路口拦截犯罪分子问题

1. 需求描述

如图 4-34 所示，在许多城市的犯罪分子抓捕中，往往在警察接到报警时犯罪分子已经逃离现场。由于城市中十字路口较多，犯罪分子的逃离路线就有很多种。只有在恰当路口进行及时有效堵截才能成功抓捕逃犯。

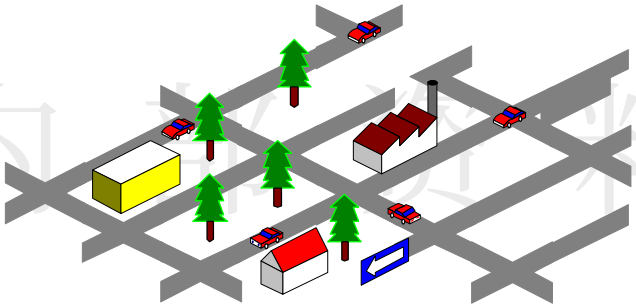

图 4-34 城市街道示意图

本应用就将解决如何对犯罪分子定点拦截问题。这个应用程序应当至少支持以下功能：

1）获取城市的街道信息，主要包括十字路口之间花费的时间；

2）更改城市的街道信息，包括更改十字路口之间的时间开销等；

3）获取犯罪所在位置；

4）获取犯罪分子作案时间；

5）计算警察应当拦截的路口（这里假设犯罪分子在作案之后一直在逃跑，不会在街道上停留，也不考虑在十字路口所花费的时间）。

2．问题分析

本应用的目标是在一个城市的街道中抓捕犯罪分子，可以利用本章中学习到的图来存储城市中的街道信息。获取城市的街道信息即对应于从文件中读取数据，并利用读取得到的数据构建一个图。删除一条街道即相当于在图中删除一条边，增加一条街道即相当于在图中增加一条边。实际情况中会遇到这样的情况：由于交通拥挤等问题会导致一条街道上花费的时间改变。所以还需要实现修改街道之间所花费的时间的功能，这对应于修改图中结点之间的权重。

犯罪分子在作案后，警察一般会过一段时间才接到报警，因此需要获取犯罪分子所在位置以及犯罪分子作案时间。假定犯罪分子在作案之后一直在逃跑并且不会在街道上停留，犯罪分子用于逃跑的时间等于作案的时刻与警察局接到报案之间的时间差。可以利用本章学习到的最短路径算法计算出犯罪分子在案发地点到其他街道所需的最短时间。如果犯罪分子从案发地点到某条街道所需的最短时间超过了作案时刻与警察接到报案之间的时间差，那么这条街道犯罪分子目前就不可能到达，警察就需要布置警力来拦截。反之，警察不需要布置警力去拦截了。

3．概要设计

根据需求描述和问题分析，本应用涉及街道、城市、犯罪分子作案地点、犯罪分子作案时间等概念。考虑到犯罪分子作案地点是街道的一个实例，因此犯罪分子作案地点类并不需要；街道只涉及街道名字，可以用 string 类型来表示；犯罪分子作案时间使用 int 类型来表示。本应用只需要犯罪分子抓捕助手 CriminalHunter 类和城市 City 类两个类型。

根据上文的分析，CriminalHunter 的成员变量至少包含：城市的名字（mCityName），城市的街道数目（mStreetNumber），街道之间时间代价组成的图（mStreetMap），犯罪分子作案的街道（mCriminalStreet），犯罪分子作案时间与报案时间之间的时间差，即逃逸时间（mRunTime）；而成员函数则包括获取城市街道数目 GetStreetNumber，设置城市街道数目 SetStreetNumber，加载街道之间代价图 LoadStreetMap，保存街道之间代价图 SaveStreetMap，读取案发街道 GetCriminalStreet，设置案发街道 SetCriminalStreet，读取逃逸时间 GetRunTime，设置逃逸时间 SetRunTime。

城市 City 类的定义与 4.8.2 节中的 City 类型定义相似。

相关类的结构及关系如图 4-35 所示。

4．详细设计

下面讨论 City 类和 CriminalHunter 类中主要成员函数的实现。

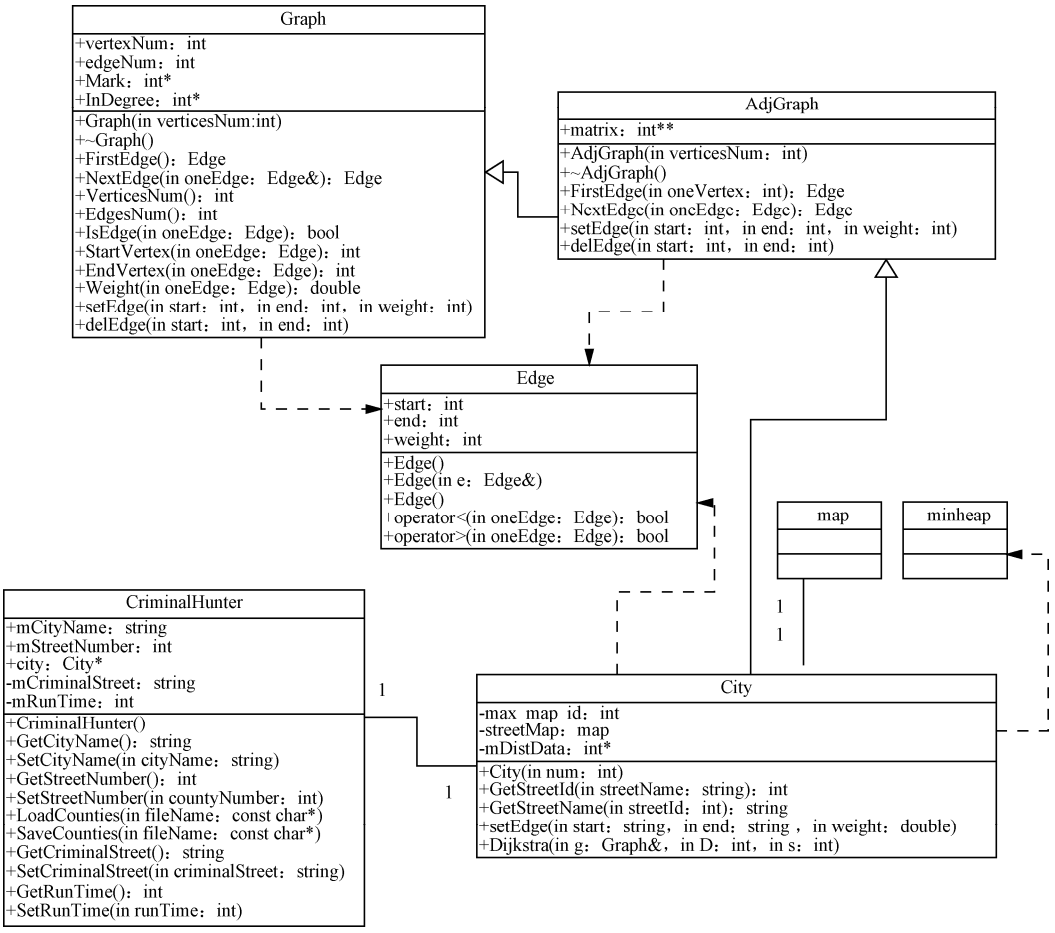

图 4-35　拦截犯罪分子问题的类图

（1）City 类

城市类 City 由邻接表表示的图 AdjGraph 派生。城市中的街道用 string 类型来表示，每条街道对应于一个唯一的 id，街道名与街道 ID 的映射关系保存在 map 类型中。为此，需要提供根据街道名来查找街道 ID 的接口、根据街道 ID 来查找街道名的接口。

另外，还需要提供根据街道名称直接设置图中边的权重的接口。其流程结构与 4.8.2 节的图 4-32 相同。

（2）CriminalHunter 类：

CriminalHunter 类中提供了主函数可以调用的接口函数。最重要的就是加载城市的街道信息函数 LoadStreets()和拦截犯罪分子的接口函数 Hunt()。

城市的街道信息保存在文件中，文件的第一行是城市的街道数 N；接下来的若干行中，每一行都是 str1 str2 time：表示街道 str1 与 str2 之间的时间代价为 time。

LoadStreets()函数的流程如图 4-36 所示。

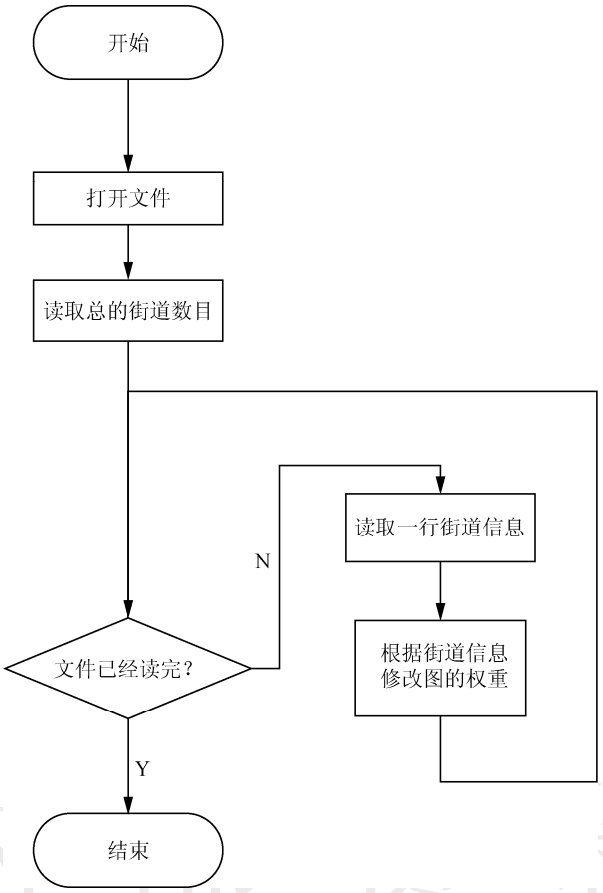

图 4-36　LoadStreets 方法的流程图

Hunt()函数首先找到案发街道对应的街道 ID 号，并以此 ID 为源点执行 Dijkstra 算法，找出其他街道离案发街道的最短时间代价，如果时间代价超出犯罪分子的逃逸时间，说明在对应街道上应该设置警力进行拦截。

为了实现本应用，还需要使用本章的 Graph 类、AdjGraph 类以及 Dijkstra 算法。

4.8.4　关键路径：软件项目管理的流程控制问题

1．需求描述

通常软件项目的开发过程称为一个工程，典型的软件工程过程包括需求分析、设计、实施、测试、维护等过程。其中，一些子过程可以同时进行，例如，编码与单元测试，以及每模块之间的编码过程等。另一些子过程则必须序列化完成，如测试完成后才能发布。称这些子过程为活动。仅当所有活动均完成时，整个工程才能完成。工程上通常使用甘特图来控制项目流程，如图 4-37 所示。

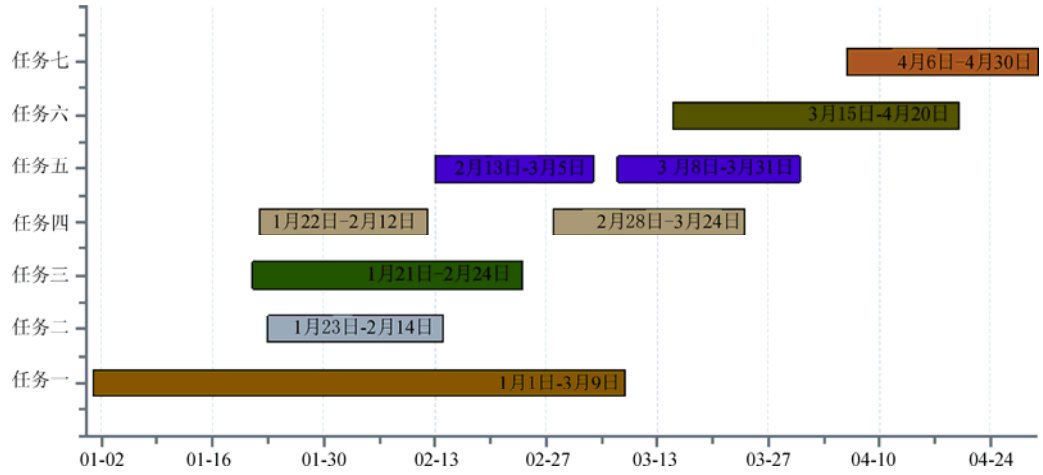

图 4-37　甘特图示例

为了简化描述，本应用中采用带权的有向无环图（AOE 网）来模拟流程，AOE 网中的边表示活动，权表示活动的持续时间，结点表示活动的完成事件。其中 AOE 网上有些活动可以并行，故从开始到结束结点的路径不止一条，并且各条路径上的时间也相差很大。但是只有各条路径上的活动全部完成后，该工程才能全部完成。因此完成工程的最短时间是从开始结点到结束结点的最长路径（即关键路径）的长度，即这条路径上所有活动所需时间的总和。关键路径在项目进度管理上有重大意义，可以估算工程的开销，评估项目中影响进度的关键活动，让工程管理者可以统筹全局进度，做出优化安排。

本应用就是要在给定一些工程活动和事件后对工程进行建模，求出该工程的关键路径。

2．问题分析

本应用要用图的数据结构进行建模，考虑到项目中活动的特性，此 AOE 网是稀疏的，采用邻接表存储图比较合理。

对于建立好的 AOE 网络，利用 4.7 节的算法可以寻找到关键路径，并且可以观察到一些关键路径的有趣的性质：仅当缩短关键路径上的活动工期时，才有可能缩短整个项目的工期；调整关键路径上的工期时，整个项目的关键路径可能会发生改变。

3．概要设计

根据上面的分析需要用 AOENetwork 类用来存储图，ProjectAct 类用来存储活动。

ProjectAct 类，需要存储活动 ID，活动名称 Name，持续时间 Duration 及开始事件 From 和结束事件 To。

对于 AOENetwork 类，由 ListGraph 派生而来。TopologySortEx 成员函数用来对网络进行拓扑排序并计算每个事件的最早开始时间，FindCriticalPath 成员函数用来计算关键路径。

本应用的涉及的类定义及类之间的关系如图 4-38 所示。

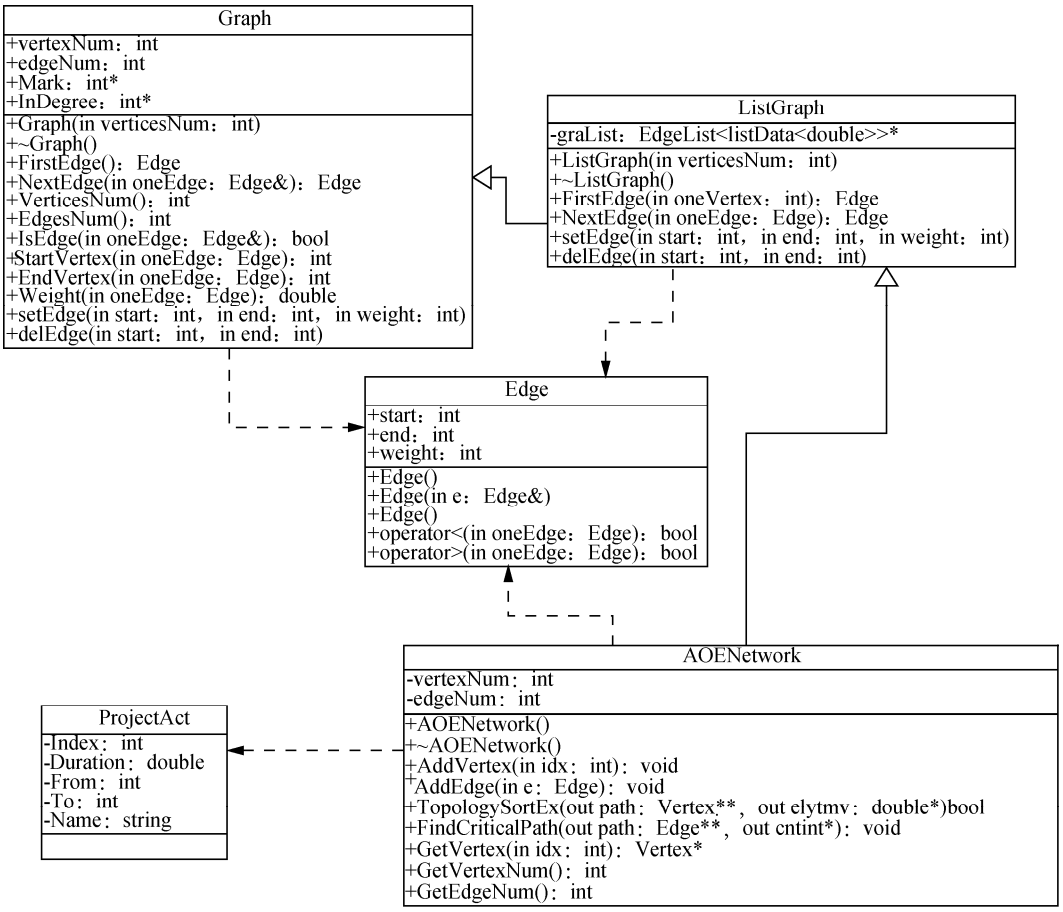

图 4-38　软件项目管理应用的类图设计

4．详细设计

下面说明 AOENetwork 类的关键成员函数 TopologySortEx 的具体实现。该函数的实现流程如图 4-39 所示。

AOENetwork 类中 FindCriticalPath 方法的实现可以参照 4.7 节中关键路径的算法。

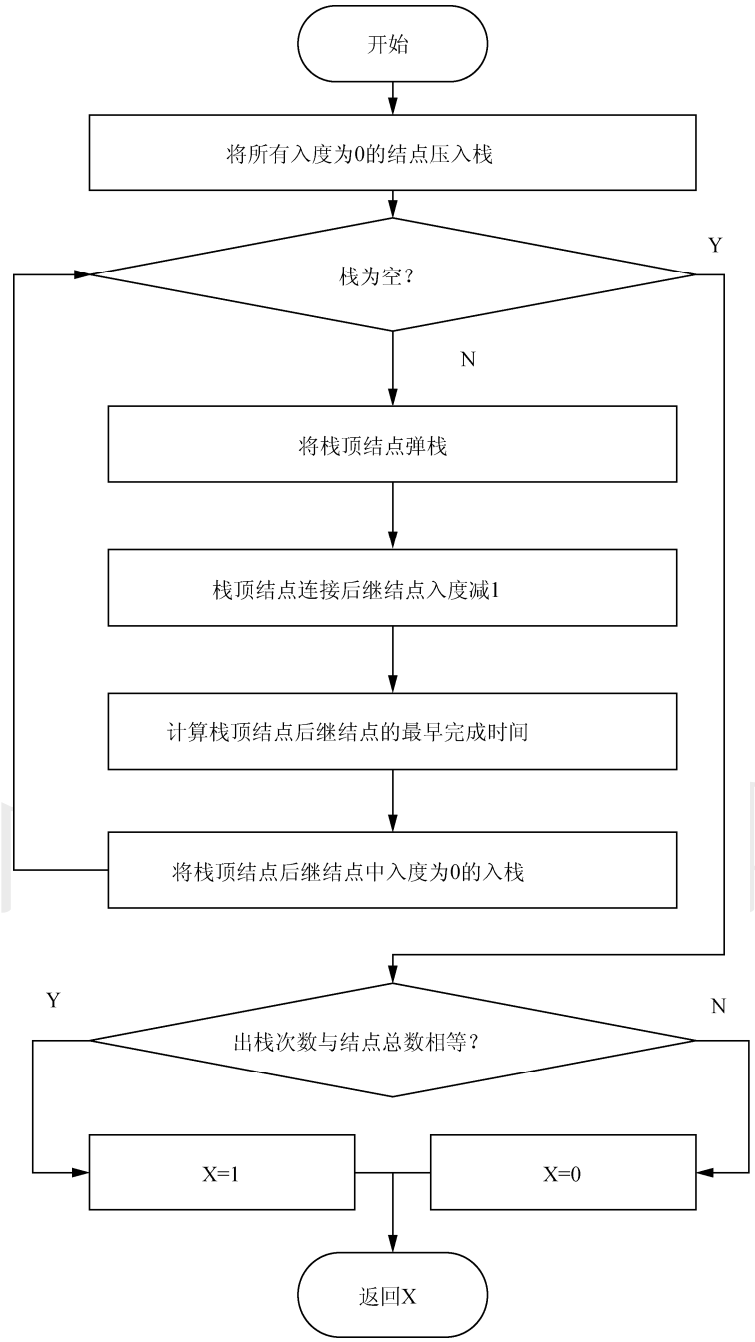

图 4-39　AOENetwork::TopologySortEx 方法流程图

习　　题

1．如图 4-40 所示为一有 5 个顶点$\{v_0, v_1, v_2, v_3, v_4\}$的有向图的邻接表。根据此邻接表:

（1）画出相应的有向图；

（2）由 v_0 出发，画出相应的深度优先搜索生成树和广度优先搜索生成树；

（3）该图是否存在拓扑排序序列？若存在给出所有可能的拓扑排序序列。

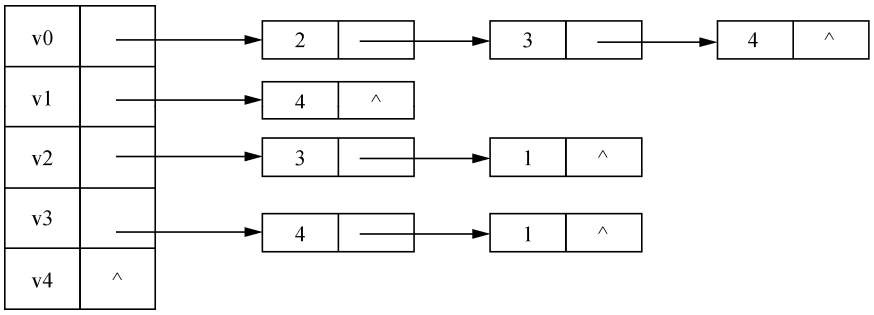

图 4-40　习题 1 中有向图的邻接表

2．如图 4-41 所示为一个 5 个顶点的带权无向图，

（1）从顶点 a 出发，画出相应的广度优先搜索生成树和深度优先搜索生成树（当从某顶点出发搜索它的邻接点时，请按邻接点序号递增序搜索）；

（2）从顶点 a 出发，画出按照普里姆算法构造的最小生成树，并给出构造过程中的加边顺序。

3．如图 4-42 所示为一有 6 个顶点$\{u_1, u_2, u_3, u_4, u_5, u_6\}$的带权有向图的邻接矩阵。

（1）根据此邻接矩阵画出相应的带权有向图;

（2）利用迪杰斯特拉算法求第一个顶点 u_1 到其余各顶点的最短路径，并给出计算过程。

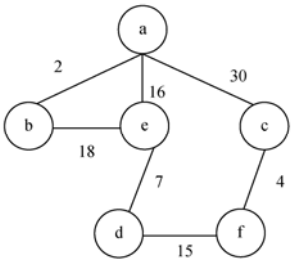

图 4-41　习题 2 的无向图

$$\begin{bmatrix} 0 & 7 & \infty & 2 & \infty & 60 \\ \infty & 0 & \infty & \infty & 6 & \infty \\ \infty & \infty & 0 & \infty & \infty & \infty \\ \infty & \infty & \infty & 0 & 12 & 15 \\ \infty & \infty & 25 & \infty & 0 & \infty \\ \infty & \infty & 10 & \infty & 1 & 0 \end{bmatrix}$$

图 4-42　习题 3 的邻接矩阵

4．证明在图中边权为负时，Dijkstra 算法不能正确运行。

5．如果图中存在负的边权，那么 Prim 算法或 Kruskal 算法还能正确运行吗？举例说明。

6．Dijkstra 算法如何应用到无向图？

7．已知一有向图的邻接矩阵如图 4-43 所示，如果需在其中一个结点建立娱乐中心，要求该结点距其他各结点的最长往返路程最短，相同条件下总的往返路程越短越好，问娱乐中心应选址何处？给出解题过程。

8．对图 4-44 所示的 AOE 网络，计算各活动的最早开始和最迟开始时间，以及各事件（顶点）的最早发生和最迟发生时间，列出所有关键路径。

$$\begin{array}{c} \\ v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \\ v_6 \end{array}\begin{bmatrix} 0 & 2 & \infty & \infty & \infty & 3 \\ \infty & 0 & 3 & 2 & \infty & \infty \\ 4 & \infty & 0 & \infty & 4 & \infty \\ 1 & \infty & \infty & 0 & 1 & \infty \\ \infty & 1 & \infty & \infty & 0 & 3 \\ \infty & \infty & 2 & 5 & \infty & 0 \end{bmatrix}$$

图 4-43　习题 7 的邻接矩阵

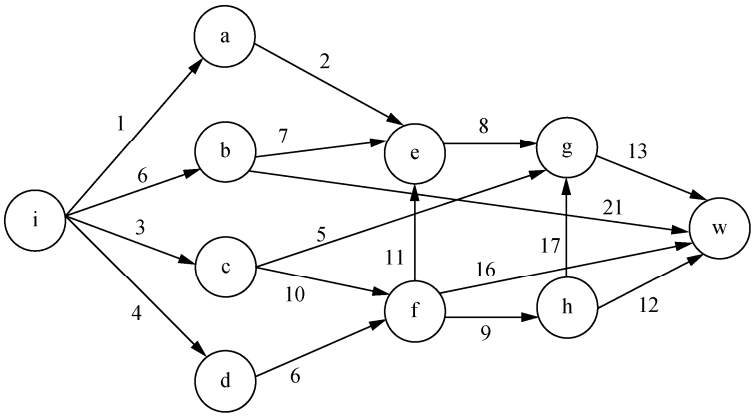

图 4-44　习题 8 的有向图

9．假设以邻接矩阵作为图的存储结构，编写算法判别在给定的有向图中是否存在一个简单有向回路，若存在，则以顶点序列的方式输出该回路（找到一条即可）。（注：图中不存在顶点到自己的弧）

10．可用“破圈法”求解带权连通无向图的一棵最小代价生成树。所谓“破圈法”就是“任取一圈，去掉圈上权最大的边”，反复执行这一步骤，直到没有圈为止。请给出用“破圈法”求解给定的带权连通无向图的一棵最小代价生成树的详细算法，并用程序实现你所给出的算法。注：圈就是回路。