```
//////////////////////////////////////////////////////////////////////////////
//////////////////////////
//   用邻接矩阵表示图的类实现
//   Author：Melissa M.CAO
//   Belong：Section of software theory，School of Computer Engineering & Science，
Shanghai University
//   Version： 1.0
//////////////////////////////////////////////////////////////////////////////
//////////////////////////

#include "StdAfx.h"
#include "AdjacencyMatrixGraph.h"
#include "LinkQueue.h"  //广度优先搜索中用到
#include "SeqStack.h"   //关键路径算法中用到
#include "MinSpanTree.h" //最小生成树算法中用到
#include "UFSet.h"      //最小生成树算法中用到

//////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////
//  构造函数
//////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////
template<class vertexType, class arcType> AdjacencyMatrixGraph<vertexType,
arcType>::AdjacencyMatrixGraph(int num, int type1, int type2)
{
    int i, j;

    graphType = type1;
    weightGraph = type2;

    for (i = 0; i < num; i++)
        for (j = 0; j < num; j++)
        {
            switch (weightGraph)
            {
                case 1:
                    if (i == j)
                        Arcs[i][j] = 0;
                    else
                        Arcs[i][j] = 0;
                break;

                case 2:
                    if (i == j)
```

```
                        Arcs[i][j] = 0;
                    else
                        Arcs[i][j] = edgeMaxValue;
                break;

                default:
                    cout<<"既不是带权图又不是无权图，无所适从，不干了！"<<endl;
            }

        }
    CurrentNumArcs = 0;
}

//////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////
//  构造函数
//////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////
template<class vertexType, class arcType> AdjacencyMatrixGraph<vertexType,
arcType>::AdjacencyMatrixGraph(int num, int type1, int type2, arcType max)
{
    int i, j;

    graphType = type1;
    weightGraph = type2;
    edgeMaxValue = max;

    for (i = 0; i < num; i++)
        for (j = 0; j < num; j++)
        {
            switch (weightGraph)
            {
                case 1:
                    Arcs[i][j] = 0;
                    /*if (i == j)
                        Arcs[i][j] = 0;
                    else
                        Arcs[i][j] = -1;*/
                break;

                case 2:
                    if (i == j)
                        Arcs[i][j] = 0;
                    else
```

```
                    Arcs[i][j] = edgeMaxValue;
                break;

                default:
                    cout <<"既不是带权图又不是无权图，无所适从，不干了！" << endl;
            }

        }
    CurrentNumArcs = 0;
}
////////////////////////////////////////////////////////////////////////////
/////////////////////////////////////////////
//无权值的图插入边
////////////////////////////////////////////////////////////////////////////
/////////////////////////////////////////////
template<class vertexType, class arcType> int AdjacencyMatrixGraph<vertexType,
arcType>::InsertArc(vertexType v1, vertexType v2)
{
    int x = GetVertexPos(v1);
    int y = GetVertexPos(v2);
    if (x < 0 || x >= NumberOfVertexes() || y < 0|| y >= NumberOfVertexes())
    {
        cout << "要插入的边的始点或终点不存在！" << endl;
        return 0;
    }

    if (x == y)
    {
        cout << "要插入的边的始点或终点相同，自身间的边不考虑！" << endl;
        return 0;
    }

    Arcs[x][y] = 1;
    if (graphType == 1)
        Arcs[y][x] = 1;
    CurrentNumArcs++;
    return 1;
}


////////////////////////////////////////////////////////////////////////////
/////////////////////////////////////////////
//图插入边
////////////////////////////////////////////////////////////////////////////
```

```
//////////////////////////////////////////////
template<class vertexType, class arcType> int AdjacencyMatrixGraph<vertexType,
arcType>::InsertArc(vertexType v1, vertexType v2, arcType weight)
{
    int x = GetVertexPos(v1);
    int y = GetVertexPos(v2);
    if (x < 0 || x >= Vertexes.Length() || y < 0|| y >= Vertexes.Length())
    {
        cout << "要插入的边的始点或终点不存在！" << endl;
        return 0;
    }

        if (x == y)
    {
        cout << "要插入的边的始点或终点相同，自身间的边不考虑！" << endl;
        return 0;
    }

    Arcs[x][y] = weight;
    if (graphType == 1)
        Arcs[y][x] = weight;
    CurrentNumArcs++;
    return 1;
}

//////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////
//显示图信息
//////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////
template<class vertexType, class arcType> void AdjacencyMatrixGraph<vertexType,
arcType>::Display()
{
    int i, j, num;
    if (graphType ==1)
        cout << "您所建立的图是无向图，";
    else
        if (graphType ==2)
            cout << "您所建立的图是有向图，";
        else {
            cout <<"您所建立的图既不是无向图也不是有向图，不知道哪里出错了！" <<
endl;
            return;
        }
```

```
    if (weightGraph ==1)
        cout << "该图无权值，";
    else
        if (weightGraph ==2)
            cout << "该图有权值，";
        else {
            cout << "该图既不是无权值也不是有权值，不知道哪里出错了！" << endl;
            return;
        }
    num = Vertexes.Length();
    cout << "该图具有" << num << "个顶点，有" << CurrentNumArcs << "条边" << endl;
    cout << "该图的" << num << "个顶点值为：";
    Vertexes.Display();
    cout << "该图的邻接矩阵为：" << endl;
    for (i = 0; i < num; i++)
    {
        for (j = 0;j < num; j++)
            cout << Arcs[i][j] << "     ";
        cout << endl;
    }
}

//////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////
//设置顶点值
//////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////
template<class vertexType, class arcType> void AdjacencyMatrixGraph<vertexType,
arcType>::SetVertex(vertexType *a, int num)
{
    int i;
    for (i = 0; i < num; i++)
        Vertexes.AppendItem(a[i]);
}

//////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////
//给出始点和终点的边的权值
//////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////
template<class vertexType, class arcType> arcType AdjacencyMatrixGraph<vertexType,
arcType>::GetWeight(vertexType v1,vertexType v2)
{
```

```
    if (weightGraph == 1)
    {
        cout << "该图不带权值，您查找的边无法给出权值！" << endl;
        return NULL;
    }
    int begin, end;
    begin = GetVertexPos(v1);
    end = GetVertexPos(v2);
    if (begin < 0 || begin >= Vertexes.Length() || end < 0 || end >= Vertexes.Length())
    {
        cout << "您查找的边的起点或终点不存在，无法给出权值！" << end;
        return NULL;
    }

    if (begin == end)
    {
        cout << "要查找的边的始点或终点相同，自身间的边不考虑！" << endl;
        return NULL;
    }


    if (Arcs[begin][end] == edgeMaxValue)
    {
        cout <<"您查找的起点和终点之间的边不存在,给出的权值是您输入的最大值!"<<
end;
        return Arcs[begin][end];
    }

    return Arcs[begin][end];
}

////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////
//删除始点和终点的边
////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////
template<class vertexType, class arcType> int AdjacencyMatrixGraph<vertexType,
arcType>::DeleteArc(vertexType v1,vertexType v2)
{
    int begin, end;
    begin = GetVertexPos(v1);
    end = GetVertexPos(v2);
    if (begin < 0 || begin >= Vertexes.Length() || end < 0 || end >= Vertexes.Length())
    {
```

```
        cout << "您要删除的边的起点或终点不存在，无法给出权值！" << end;
        return 0;
    }

    if (begin == end)
    {
        cout << "您要删除的边的始点或终点相同，自身间的边不考虑！" << endl;
        return 0;
    }



    if (weightGraph == 2 && Arcs[begin][end] == edgeMaxValue)
    {
        cout << "您要删除的起点和终点之间的边不存在，给出的权值是您输入的最大值！
" << end;
        return 0;
    }

    if (weightGraph == 2)
    {
        Arcs[begin][end] = edgeMaxValue;
        if (graphType == 1)
            Arcs[end][begin] = edgeMaxValue;
    }
    else
    {
        Arcs[begin][end] = 0;
        if (graphType == 1)
            Arcs[end[begin]] = 0;
    }
    return 1;
}

//////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////
//查找顶点 v 的第一条边，返回找到的边的终点下标号
//////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////
template<class vertexType, class arcType> int AdjacencyMatrixGraph<vertexType,
arcType>::GetFirstNeighbor(vertexType v)
{
    int i, x = GetVertexPos(v), m = Vertexes.Length();
    if (x >= 0 && x < m)
    {
```

```
        for (i = 0; i < m; i++)
        {
            if (weightGraph == 1) //无权图
            {
                if (Arcs[x][i] != 0)
                    return i;
            }
            else
                if (weightGraph == 2)// 有权图
                {
                    if (Arcs[x][i] != edgeMaxValue && x != i)//排除自身到自身的权
制
                        return i;
                }
                else
                    return -1;
        } // end of for
    }
    return -1;
}

//////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////
//查找顶点 v1 与 v2 构成边的下一条边，返回找到的边的终点下标号。
//
//////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////
template<class vertexType, class arcType> int AdjacencyMatrixGraph<vertexType,
arcType>::GetNextNeighbor(vertexType v1,vertexType v2)
{
    int x = GetVertexPos(v1);
    int y = GetVertexPos(v2);
    int m = Vertexes.Length();

    if (x < 0 || y < 0 || x >= m || y >= m)
    {
        cout << "所要查找的起点或终点不存在！" << endl;
        return -1;
    }
    int i;
    for (i = y+1; i < m; i++)
    {
        if (weightGraph == 1) //无权图
        {
```

```
            if (Arcs[x][i] != 0)
                return i;
        }
        else
            if (weightGraph == 2)// 有权图
            {
                if (Arcs[x][i] != edgeMaxValue && x != i) //排除自身到自身的权制
                    return i;
            }
            else
                return -1;
        } // end of for
    return -1;
}

////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////
//插入顶点 v，置于顺序表的最后
////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////
template<class vertexType, class arcType> int AdjacencyMatrixGraph<vertexType,
arcType>::InsertVertex(vertexType &v)
{
    int i, m;
    if (Vertexes.AppendItem(v) == 1)    //插入结点成功
    {
        //增加邻接矩阵的一行和一列
        m = Vertexes.Length();
        Arcs[m-1][m-1] = 0;
        for (i = 0; i < m-1; i++)
            if (weightGraph == 1)
            {
                Arcs[m-1][i] = 0;
                Arcs[i][m-1] = 0;
                return 1;
            }
            else
                if (weightGraph == 2)
                {
                    Arcs[m-1][i] = edgeMaxValue;
                    Arcs[i][m-1] = edgeMaxValue;
                    return 1;
                }
                else
```

```
                return 0;
    }
    return 0;
}

//////////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////
//删除顶点 v
//////////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////
template<class vertexType, class arcType> int AdjacencyMatrixGraph<vertexType,
arcType>::DeleteVertex(vertexType v)
{
    int i = GetVertexPos(v);
    int m = NumberOfVertexes();
    int j, k;
    if (i < 0 || i >= m)
    {
        cout << "要删除的顶点不存在" << endl;
        return 0;
    }

    //统计删除的边数
    if (weightGraph == 1)
    {
        for (k = 0; k < m; k++) //处理行
            if (Arcs[i][k] == 1)
                CurrentNumArcs--;
        if (graphType == 2) //有向图，处理列
            for (k = 0; k < m; k++)
                if (Arcs[k][i] == 1)
                    CurrentNumArcs--;
    }
    else
    {
        for (k = 0; k < m; k++) //处理行
            if (Arcs[i][k] !=0 && Arcs[i][k] != edgeMaxValue)
                CurrentNumArcs--;
        if (graphType == 2) //有向图，处理列
            for (k = 0; k < m; k++)
                if (Arcs[k][i] != 0 && Arcs[k][i] != edgeMaxValue)
                    CurrentNumArcs--;
    }
```

```
    if (i == m-1) //删除的顶点序号是最后一个
    {
        Vertexes.Delete(i+1, true);
        //统计删除的边数
        return 1;
    }

    //i 后的每一行上移
    for (j = i; j < m; j++)
        for (k = 0; k < m; k++)
            Arcs[j][k] = Arcs[j+1][k];
    //i 后的每一列左移
    for (j = i; j < m; j++)
        for (k = 0; k < m; k++)
            Arcs[k][j] = Arcs[k][j+1];
    //删除顶点 v
    Vertexes.Delete(i+1, true);

    return 1;
}

////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////
//-----------------------------------------------------------------扩展操作
------------------------------------------------
////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////
//定义的抽象的“访问/遍历”函数
////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////
template<class vertexType, class arcType> void AdjacencyMatrixGraph<vertexType,
arcType>::Visit( vertexType v )
{
    int i = GetVertexPos(v);

    if (i < 0 || i >= NumberOfVertexes())
    {
        cout << "要访问/遍历的顶点不存在" << endl;
        return;
    }
    else
        cout << "顶点值“" << v << "”是图中的第" << i+1 << "个顶点！" << endl;
}
```

```
//////////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////
//判断图中边的权值是否含有负数
//////////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////
template<class vertexType, class arcType> bool AdjacencyMatrixGraph<vertexType,
arcType>::HavePostiveEdge()
{
    int n = NumberOfVertexes();
    int i,j;

    for (i = 0; i < n; i++)
        for (j = 0; j < n; j++)
            if (Arcs[i][j] < 0)
                return false;
    return true;
}

//////////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////
//深度优先搜索
//   参数： v，指定顶点的序号（不是下标，下标+1）；*visited，记录顶点访问记录，即
每个顶点是否被访问过；
//          count：用于记录在遍历中已经访问了多少个顶点。
//////////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////
template<class vertexType, class arcType> void AdjacencyMatrixGraph<vertexType,
arcType>::DFS(const int v, int *visited, int *count)
{
    *count = *count + 1;
    vertexType temp = GetValue (v);
    Visit(temp);  //访问顶点 v, 例如 cout, modify
    visited[v] = 1;          //顶点 v 作访问标记
    int w = GetFirstNeighbor (temp);    //w 是返回的顶点的下标
    while ( w != -1 )         //若顶点 w 存在
    {
        if ( !visited[w+1] )
            DFS (w+1, visited, count);
        w = GetNextNeighbor (temp, GetValue (w+1));
    }  //重复检测 v 的所有邻接顶点
}


//////////////////////////////////////////////////////////////////////////////
```

```
////////////////////////////////////////////////
//深度优先遍历
//////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////
template<class vertexType, class arcType> void AdjacencyMatrixGraph<vertexType,
arcType>::DFTraverse()
{
    int i, n = NumberOfVertexes() ;   //取图的顶点个数
    int * visited = new int [n+1]; //定义访问标记数组 visited
    int count = 0, block = 0;
    for ( i = 0; i <= n; i++ )
        visited [i] = 0;  //访问标记数组 visited 初始化
    //对图中的每一个顶点进行判断
    for ( i = 1; i <= n; i++ ) {
        if (!visited [i])
        {
            DFS (i, visited, &count);
            block ++;
            cout << "||||以上为第" << block << "个连通分量！" << endl;
        }
    }
    cout << "深度优先遍历结果如上，该图有" << block << "个连通分量！" << endl;
    delete [ ] visited;                    //释放 visited
}

//////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////
//广度优先搜索，返回搜索到的顶点个数
//   参数： v，指定顶点的序号（不是下标，下标+1）；*visited，记录顶点访问记录，即
每个顶点是否被访问过；
//          count：用于记录在遍历中已经访问了多少个顶点。
//////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////
template<class vertexType, class arcType> void AdjacencyMatrixGraph<vertexType,
arcType>::BFS(int v, int *visited, int *count)
{
    int w;
    LinkQueue <int> q;  //定义队列 q
    vertexType valueofw,valueofv = GetValue (v);
    Visit(valueofv);  //访问顶点 v
    visited[v] = 1;           //顶点 v 作已访问标记
    *count += 1;
    q.EnQueue (v);  //顶点 v 进队列 q
    while ( !q.IsEmpty ( ) )
```

```
    {
        v = q.DeQueue ( );        //否则, 队头元素出队列
        valueofv = GetValue (v);
        w = GetFirstNeighbor (valueofv);
        while ( w != -1 )       //若邻接顶点 w 存在
        {
            valueofw = GetValue (w+1);
           if ( !visited[w+1] )   //若该邻接顶点未访问过
           {
                 Visit(valueofw);  //访问顶点 w
                 visited[w+1] = 1;        //顶点 w 作已访问标记
                 *count += 1;
                 q.EnQueue (w+1);  // w 进队列 q
          }
          w = GetNextNeighbor (valueofv, valueofw);
       }    //重复检测 v 的所有邻接顶点
    }    //外层循环，判队列空否

}

////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////
//广度优先遍历
////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////
template<class vertexType, class arcType> void AdjacencyMatrixGraph<vertexType,
arcType>::BFTraverse()
{
    int i, n = NumberOfVertexes() ;   //取图的顶点个数
    int * visited = new int [n+1]; //定义访问标记数组 visited
    int count = 0, block = 0;
    for ( i = 0; i <= n; i++ )
        visited [i] = 0;  //访问标记数组 visited 初始化
    //对图中的每一个顶点进行判断
    for ( i = 1; i <= n; i++ ) {
        if (!visited [i])
        {
            BFS (i, visited, &count);
            block ++;
            cout << "||||以上为第" << block << "个连通分量！" << endl;
        }
    }
    cout << "广度优先遍历结果如上，该图有" << block << "个连通分量！" << endl;
    delete [ ] visited;                    //释放 visited
```

```
}

//////////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////
//判断图是否连通
//////////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////
template<class vertexType, class arcType> bool AdjacencyMatrixGraph<vertexType,
arcType>::IsConnected()
{
    bool r1, r2;
    int i, n = NumberOfVertexes() ;   //取图的顶点个数
    int * visited = new int [n+1]; //定义访问标记数组 visited
    int count;

    for ( i = 0; i <= n; i++ )
        visited [i] = 0;  //访问标记数组 visited 初始化
    count = 0;
    DFS (1, visited, &count);
    if (count == n)
        r1 = true;
    else
        r1 = false;

    for ( i = 0; i <= n; i++ )
        visited [i] = 0;  //访问标记数组 visited 初始化
    count = 0;
    BFS (1, visited, &count);
    if (count == n)
        r2 = true;
    else
        r2 = false;

    cout << "按深度优先从第一个顶点开始搜索，判断该图连通性结果：" << r1 << endl;
    cout << "按广度优先从第一个顶点开始搜索，判断该图连通性结果：" << r2 << endl;
    if (r1 != r2)
    {
        cout <<"非常遗憾，采用不同的搜索方法，居然得出连通性判断的不同结果，出问
题了，请检查代码！" << endl;
        exit(1);
    }
    return r1;
}
```

```
//////////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////
//迪杰斯特拉算法，参数为顶点下标，即存储位置，不是逻辑位置（逻辑位置减 1）
//////////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////
template<class vertexType, class arcType> void AdjacencyMatrixGraph<vertexType,
arcType>::Dijkstra(int v)
{
    arcType min;
    int n = NumberOfVertexes();     //顶点数
    if (n < 1)
    {
        cout << "没有顶点，为空图，无须求最短路径！";
        return;
    }
    if (v < 0  || v >= n)
    {
        cout << "您选择的单源点起点不存在，无须求最短路径！";
        return;
    }

    int u, i, j, w;
    dist = new arcType[n];
    s = new int[n];
    path = new int[n];

    for (i = 0; i < n; i++)
    {
        dist[i] = Arcs[v][i]; //dist 数组初始化
        s[i] = 0;  //i 是否已经在 s 中
        if ( i != v && dist[i] < edgeMaxValue)
            path[i] = v;
        else
            path[i] = -1;        //path 数组初始化
    }
    s[v] = 1; //顶点 v 加入顶点集合 s
    cout << "您选择的源点为：" << GetValue(v+1) << endl;
    for ( i = 0; i < n-1; i++ )  //按递增序列球最短路径，共 n-1 步
    {
        cout << "第" << i+1 << "步：";
        min = edgeMaxValue;  //找当前最小值--开始
        u = v;
        for (j = 0; j < n; j++ )
            if ( !s[j] && dist[j] < min )
```

```
            {
                u = j;
                min = dist[j];
            }       //找当前最小值--结束
        cout << "确定的最小值为顶点" << GetValue(v+1) << "到顶点" << GetValue(u+1)
<< "的最短距离，值为：" << min << endl;
        cout << "修改后的当前距离值为：" << endl;
        s[u] = 1;                   //将顶点 u 加入集合 S
        for (w = 0; w < n; w++ )    //修改 dist 和 path
        {
              if ( !s[w] && dist[u] + Arcs[u][w] < dist[w] )
              {
                   dist[w] = dist[u] + Arcs [u][w];
                   path[w] = u;
               }
              if (!s[w])
              {
                  cout <<"源点到顶点（" <<GetValue(w+1) <<"）的当前最短距离为：
" << dist[w];
                  cout << "，经顶点" << GetValue(path[w]+1) << "到达该顶点！" <<
endl;
              }
        }
        cout << endl;
  }
}

//////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////
//利用迪杰斯特拉算法或贝尔曼—福特算法求单源点最短距离
//////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////
template<class vertexType, class arcType> void AdjacencyMatrixGraph<vertexType,
arcType>::ShortestPath(int type, int begin, vertexType &v)
{
    int indexV = -1;;
    bool r = HavePostiveEdge();
    if (type == 2)
    {
        indexV = GetVertexPos(v);
        if (r)
            Dijkstra(indexV);
        else
            BellmanFord(indexV); //该算法还没有最后验证其运行结果---2010-july-1
```

```
    }
    else
    {
        if (r)
            Dijkstra(begin);
        else
            BellmanFord(begin);
    }
    delete dist;
    delete path;
}

//////////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////
//贝尔曼—福特算法，参数为顶点下标，即存储位置，不是逻辑位置（逻辑位置减 1）
//////////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////
template<class vertexType, class arcType> void AdjacencyMatrixGraph<vertexType,
arcType>::BellmanFord(int v)
{
    int n = NumberOfVertexes();     //顶点数
    vertexType temp = GetValue(v+1);
    if (n < 1)
    {
        cout << "没有顶点，为空图，无须求最短路径！";
        return;
    }
    if (v < 0  || v >= n)
    {
        cout << "您选择的单源点起点不存在，无须求最短路径！";
        return;
    }

    int u, i, k;
    dist = new arcType[n];
    arcType *distTemp = new arcType[n]; //不能在原数组上进行计算
    //相当于递归排序中的两个表的数据来回交换。因为 dist[k]是在 dist[k-1]的基础上求
得的，因此在更新前必须保持 dist[k-1]的完整性
    path = new int[n];

    for (i = 0; i < n; i++)
    {
        dist[i] = Arcs[v][i]; //dist 数组初始化
        distTemp[i] = Arcs[v][i]; //distTemp 数组初始化
```

```
        if ( i != v && dist[i] < edgeMaxValue)
            path[i] = v;
        else
            path[i] = -1;        //path 数组初始化
    }

    for (k = 2; k < n; k++)        //循环次数 n-2
    {
        cout << "第" << k-1 << "次计算中被修改过的数据如下：" << endl;
        for (u = 0; u < n; u++)    //每个顶点 ，n 个
        {
           if (u != v)
           {
                for ( i = 0 ; i < n ; i++)     //根据 distk-1[u]计算 distk[u]
                    //if ( i != v && dist[u] > dist[i] + Arcs[i][u])
                    if ( i != v && Arcs[i][u] < edgeMaxValue && distTemp[u] > dist[i]
+ Arcs[i][u] )
                    {
                        distTemp[u] = dist[i] + Arcs[i][u];
                        path[u] = i;
                        //cout << "源点" << temp << "到顶点" << GetValue(u+1) << "
的当前最短距离由（" << dist[u] << "）修改为（";
                        //cout << distTemp[u] << "，新的最短距离经过顶点" <<
GetValue(path[u]+1) << "到达!" << endl;
                    }
           } //end of if (u!=v)
        }   // end of for u
        for (i = 0; i < n; i++)
        {
            dist[i] = distTemp[i];
            cout << dist[i] << ",              ";
        }
        cout << endl;
    } //end of for k
    delete distTemp;
}

////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////
//弗洛伊德(Floyd)算法
////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////
template<class vertexType, class arcType> void AdjacencyMatrixGraph<vertexType,
arcType>::Floyd()
```

```
{
    int n = NumberOfVertexes();     //顶点数
    arcType a[MaxVertexes][MaxVertexes];
    int i, j, k;
    int path[MaxVertexes][MaxVertexes];

    //矩阵a(-1)与path(-1)初始化
    for (i = 0; i < n; i++ )
        for (j = 0; j < n; j++ )
        {
            a[i][j] = Arcs[i][j];
            if ( i != j && a[i][j] < edgeMaxValue )
                  path[i][j] = i;   //从 i 到 j 有直接的路径（弧）
            else
                path[i][j] = -1;    //从 i 到 j 没直接的路径（弧）
        }   // end of for

    //产生A(k)及path(k)
    for (k = 0; k < n; k++ )
    {
        cout << "第" << k+1 << "次计算：" << endl;
        for ( i = 0; i < n; i++ )
        {
            for ( j = 0; j < n; j++ )
            {
             /*   if ( a[i][k] + a[k][j] < a[i][j] ) //缩短路径长度, 经过 k 到 j,
权值为正数，没有问题。权值为负数，会出现
                {                                       //表示无穷的最大值+一个负数
< 原来表示无穷的最大值*/
                if (a[i][k] != edgeMaxValue && a[k][j] != edgeMaxValue && a[i][k]
+ a[k][j] < a[i][j] )
                {
                    a[i][j] = a[i][k] + a[k][j];
                    path[i][j] = path[k][j];
                } // end of if
                cout << a[i][j] << ",            ";
            }
            cout << endl;
        }
    }
}

//////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////
```

```
//为拓扑排序算法增加入度数组，构造时未初始化入度数组，该函数进行专门处理
//为关键路径算法增加源点个数 begin 和汇点个数 end 的统计。若只有一个源点和汇点，则 begin 和 end 中分别存放源点和汇点的序号
// 若没有源点和汇点，begin 和 end 的值为-1，若多于一个源点或汇点，begin 和 end 的值为-2。
//////////////////////////////////////////////////////////////////////////////////////////////
template<class vertexType, class arcType> void AdjacencyMatrixGraph<vertexType, arcType>::InitialInDegree(int *begin, int *end)
{
    if (graphType==1)
    {
        cout <<"提示：该图是无向图，没有出度入度的概念，无须处理入度数组，故而不处理返回！" << endl;
        return;
    }

    int n = NumberOfVertexes();
    int i, j;
    int *OutDegree = new int[n];
    InDegree = new int[n];

    for (i = 0; i < n; i++)
    {
        InDegree[i] = 0;
        OutDegree[i] = 0;
    }

    //同时记录入读数组和出度数组（局部变量）
    for (i = 0; i < n; i++)
        for (j = 0; j < n; j++)
        {
            if (i != j && edgeMaxValue -Arcs[i][j] > 0.00000001) //存在边
            {
                InDegree[j]++;
                OutDegree[i]++;
            }
        }

    //统计源点个数和汇点个数
    *begin = -1;
    *end = -1;
    for (i = 0; i < n; i++)
    {
```

```
        if (InDegree[i] == 0)
            if (*begin == -1)
                *begin = i;
            else
                *begin = -2;
        if (OutDegree[i] == 0)
            if (*end == -1)
                *end = i;
            else
                *end = -2;
    }
}

//////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////
//拓扑排序算法
//////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////
template<class vertexType, class arcType> void AdjacencyMatrixGraph<vertexType,
arcType>::TopologicalSort()
{
    if (graphType == 1)
    {
        cout <<"提示：该图是无向图，没有出度入度的概念，无须处理入度数组，故而不
处理返回！" << endl;
        return;
    }

    int top = -1;                   //入度为零的顶点栈初始化
    int n = 0, j = 0, t;            //n 为输出的顶点数，初始值为 0
    InitialInDegree(&n, &j);        //初始化入度数组
    n = 0;
    int num = NumberOfVertexes();   //取得顶点的个数
    vertexType temp, temp1;

    for ( j = 0; j < num; j++ )    //入度为零的顶点进栈--采用静态链栈，目的是利用
数组 InDegree 同时作为栈，不需要为栈另辟空间
        if ( InDegree[j] == 0 )
        {
            InDegree[j] = top;
            top = j;
        }

    // 拓扑排序部分----A
```

```
    cout << "拓扑排序开始，逐个输出排序序列：";
    while (top !=  -1)        //继续拓扑排序
    {
        j = top;
        top = InDegree [top];
        n++; //  输出的顶点数加一
        temp = GetValue(j+1);
        cout << temp << ", ";
        t = GetFirstNeighbor(temp); //确定弧尾
        while (t != -1)//扫描以顶点 j 为弧尾的所有弧
        {
            if (--InDegree[t] == 0)//顶点 k 的入度减一，若为 0，进栈
            {
                InDegree[t] = top;
                top = t;
            }
            temp1 = GetValue(t+1);
            t = GetNextNeighbor(temp, temp1);
        }// end of while
    } // end of while top != -1

    delete InDegree; //入度数组在该过程中被申请，为避免后面的其他算法出错，退出该算法时同时释放申请的这些空间

    if ( n < num )
         cout << "AOV 网络中有回路(有向环)！" << endl;
    else
        cout << endl;
}

////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////
//有问题的关键路径算法
// 问题：求事件的最早开始时间应该按照拓扑排序的次序来求，否则可能出错。
// 该算法是教材中的一个错误的典型，其求事件的最早开始时间没有按照拓扑排序的次序
// 测试例子：
//           a-----8----->b---3------>c
//           | \         /|\        /|\ \ 6
//           |   \         |       /  |     \
//           |4    \5      |2    /7   |3      g
//           |       \     |   /      |      /
//           \|/         \ | /        |    / 9
//            d-----1----->e----2---->f  /
// 测试时的输入数据：顶点 abcdefg#---#为结束符
```

```
//                    边：ab8，ad4，ae5，bc3，cg6，de1，eb2，ec7，ef2，fc3，fg9，
$------$为结束符
//测试结果应该为：a 最早 0，最晚 0；b 最早 8，最晚 9；c 最早 12，最晚 12；d 最早 4，最晚 4；e 最早 5，最晚 5；f 最早 7，最晚 9；g 最早 18，最晚 18；
//运行结果为：      a 最早 0，最晚；b 最早 8，最晚；c 最早 12，最晚；d 最早 4，最晚；e 最早 5，最晚；f 最早 9，最晚；g 最早 17，最晚；
//错误原因：      求最早开始时间是按照序号进行的，因此，由 a0 求出 b8，d4，和 e5；由 b8 求出 c11；由 c11 求出 g17；
//                                                    由 d4 未修改;由 e5 求出 c12，和 f7；----此时虽然 Ve[c]由 11 更改为 12，但由于 c 已经处理完毕，因此，未能更改 g 的最早开始时间
//
          错误由此产生（尽管 fg 之间也有边，但<f,g>不是关键活动，不能正确求出 g 的最早开始时间）
//--------------------------------
// 该算法中求 Vl 的过程应该有同样问题
//原教材 P245 中有原话：“为了简化算法，假定在求关键路径之前已经对各顶点实现了拓扑排序，并按拓扑有序的顺序
//对各顶点重新进行了编号”。在该前提下，此算法能够运行得出正确结果，但不提倡这种做法，原因有两个：
//（1）按顶点存储自然序求最早开始时间和按顶点拓扑排序最早开始时间的实现思路和具体实现方法、手段并不统一；
//（2）在邻接矩阵方式存储图的情况下，按拓扑有序的顺序对各顶点重新进行编号时，不仅要重新生成顶点序号，同时要根据新旧序号的异同
//      交换矩阵的行和列，该工作并不容易实现，通常需要另外开辟空间并两次倒腾数据，不合理。
//为避免这两个问题，将算法 CriticalPathQuestion 修改为 CriticalPath。
////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////
template<class vertexType, class arcType> void AdjacencyMatrixGraph<vertexType, arcType>::CriticalPathQuestion()
{
    if (graphType == 1)
    {
        cout << "提示：该图是无向图，没有出度入度的概念，即没有源点、汇点的概念，无法求关键活动和关键路径，故而不处理返回！" << endl;
        return;
    }

    int i, j, e, l;
    int num = NumberOfVertexes();   //取得顶点的个数
    int * Ve = new int[num];        //事件（即顶点）的最早开始时间
    int * Vl = new int[num];        //事件（即顶点）的最晚开始时间
    vertexType temp, temp1;
```

```
    for ( i = 0; i < num; i++ )     //初始化
        Ve[i] = 0;
    //求每一个 Ve[i]
    for ( i = 0; i < num; i++ )
    {
        temp = GetValue(i+1);
        j = GetFirstNeighbor(temp);
        while ( j != -1 )
        {
            temp1 = GetValue(j+1);
            if ( Ve[i] +Arcs[i][j] > Ve[j] )
                Ve[j] = Ve[i] +Arcs[i][j];
            j = GetNextNeighbor(temp, temp1);
        } // end of while
    } // end of for

    for ( i = 0; i < num; i++ )
        Vl[i] = Ve[num-1];  //初始化
    //求每一个 Vl[i]
    for ( i = num - 2; i >= 0; i-- )
    {
        temp = GetValue(i+1);
        j = GetFirstNeighbor(temp);
        while (j != -1 )
        {
            temp1 = GetValue(j+1);
            if ( Vl[j] - Arcs[i][j] < Vl[i])
                Vl[i] = Vl[j] - Arcs[i][j];
            j = GetNextNeighbor(temp, temp1);
         }
    }

    //输出事件的最早、最晚开始时间
    cout << "事件（顶点）      " << "最早开始时间      " << "最晚开始时间" << endl;
    for (i = 0; i < num; i++)
    {
        temp = GetValue(i+1);
        cout << temp << "                " << Ve[i] << "                  " << Vl[i]
<< endl;

    }

    //求关键活动
```

```
    cout << "活动（边）      " << "最早开始时间      " << "最晚开始时间      " << "
松弛时间      " << "判断结果" << endl;
    for ( i = 0; i < num; i++ ) //对每一个顶点为弧头进行判断
    {
        temp = GetValue(i+1);
        j = GetFirstNeighbor(temp);
        while ( j != -1 ) //对选定顶点 i 的每一条边<i,j>进行判断
        {
            temp1 = GetValue(j+1);
            e = Ve[i];                  //活动（边）的最早开始时间
            l = Vl[j] - Arcs[i][j];     //活动（边）的最早开始时间
            cout << "<" << temp << ", " << temp1 << ">            " << e << "
";
            cout << l << "                    " << l-e << "              ";
            if ( l == e )               //活动的时间余量为 0
                cout << "是关键活动" << endl;
            else
                cout << "不是关键活动" << endl;
            j = GetNextNeighbor(temp, temp1);
        } // end of while
    }  // end of for
}

////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////
//关键路径算法之一
//测试一个源点一个汇点但有环的例子：
//顶点：abcde
//边：   ab12，ac14，cb7，ce6，bd8，dc5，de3，
////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////
template<class vertexType, class arcType> void AdjacencyMatrixGraph<vertexType,
arcType>::CriticalPath()
{
    int begin = 0, end = 0;

    //不是无项图，不处理
    if (graphType == 1)
    {
        cout << "提示：该图是无向图，没有出度入度的概念，即没有源点、汇点的概念，
无法求关键活动和关键路径，故而不处理返回！" << endl;
        return;
    }
```

```
    //源点汇点不存在或多于一个，不处理
    InitialInDegree(&begin, &end);
    if (begin < 0)
    {
        cout << "该图的源点";
        if (begin == -1)
            cout << "不存在，没有源点的图无法求关键路径，";
        else
            if (begin == -2)
                cout << "不是一个，多个源点的图不考虑，";
            else
                cout << "求源点时出错了，请检查程序代码，";
        cout << "因此，没有处理，直接返回了！" << endl;
        return;
    }
    if (end < 0)
    {
        cout << "该图的汇点";
        if (end == -1)
            cout << "不存在，没有汇点的图无法求关键路径，";
        else
            if (end == -2)
                cout << "不是一个，多个汇点的图不考虑，";
            else
                cout << "求汇点时出错了，请检查程序代码，";
        cout << "因此，没有处理，直接返回了！" << endl;
        return;
    }

    int i, j;
    arcType e, l;
    int num = NumberOfVertexes();   //取得顶点的个数
    int currentVertexNum = 0;   //当前已处理的顶点数目，用于判定能否形成环
    SeqStack<int> s(num);       //拓扑排序时顶点序号入栈，用于按逆拓扑排序次序求事件（顶点）的最晚开始时间
    int * Ve = new int[num];        //事件（即顶点）的最早开始时间
    int * Vl = new int[num];        //事件（即顶点）的最晚开始时间
    vertexType temp, temp1;
    int top = -1;   //利用数组 InDegree 同时作为保存入度为零的顶点的栈，不需要为该栈另辟空间

    for ( i = 0; i < num; i++ )     //初始化
        Ve[i] = 0;
    //求每一个 Ve[i]--按拓扑排序次序递推，将拓扑排序思想融合在此处
```

```
//（1）源点入栈，同时入 InDegree 这个静态链栈和栈 s
top = begin;    //因为只有一个源点，故，不需要象拓扑排序算法中那样用 for 循环来检查入栈
//思考：如何没有 begin 记录，如何确定源点？检查行为零
InDegree[begin] = -1;
s.Push(begin);
while (top !=  -1)        //继续拓扑排序
{
    j = top;
    top = InDegree [top];
    currentVertexNum++;  //  已处理（相当于拓扑排序）的顶点数加一
    temp = GetValue(j+1);//得到相应顶点的值
    i = GetFirstNeighbor(temp); //确定弧尾
    while (i != -1)//扫描以顶点 j 为弧尾的所有弧，即考虑所有出边
    {
        if (--InDegree[i] == 0)     //顶点 k 的入度减一，若为 0，进栈。此部分为的是拓扑排序
        {
            InDegree[i] = top;      //入 InDegree 这个静态链栈
            top = i;
            s.Push(i);              //入栈 s，用于按逆拓扑排序次序求事件（顶点）的最晚开始时间
        }
        if ( Ve[j] +Arcs[j][i] > Ve[i] )  //弧尾的最早开始时间是否需要修改。此部分处理的是关键路径中的事件最早开始时间
            Ve[i] = Ve[j] +Arcs[j][i];
        temp1 = GetValue(i+1);          //----先得到弧尾的值，再根据弧头和弧尾求弧头对于弧尾的下一条弧
        i = GetNextNeighbor(temp, temp1);
    } //end of while e != -1
}// end of while top != -1
//思考题：此处如何记录汇点？即栈中最后一个元素？currentVertexNum == num 时出栈的顶点？否则，如何倒推 Vl【】？
// 其他方法？检查列为零的点。邻接表表示的图，思想一致，实现不同。
if ( currentVertexNum < num )
{
     cout <<"AOV 网络中有回路(有向环),无法求关键路径!未处理,返回了!" << endl;
     return;
}

//汇点的最晚开始时间就是其最早开始时间，然后初始化 Vl
for ( i = 0; i < num; i++ )
    Vl[i] = Ve[end];  //初始化
//求每一个 Vl[i]
```

```
    i = s.Pop();    //---汇点的最晚开始时间就是其最早开始时间，因此，汇点先出栈
    if (i != end)
    {
        cout<<"栈顶不是汇点，与该图只有一个汇点矛盾，请检查代码!未处理，返回了!
";
        return;
    }
    while (!s.IsEmpty())    //栈不空，逐顶点处理
    {
        i = s.Pop();
        temp = GetValue(i+1);
        j = GetFirstNeighbor(temp);
        while (j != -1 )  //处理每一条出边
        {
            temp1 = GetValue(j+1);
            if ( Vl[j] - Arcs[i][j] < Vl[i])
                Vl[i] = Vl[j] - Arcs[i][j];
            j = GetNextNeighbor(temp, temp1);
         }
    }

    //输出事件的最早、最晚开始时间
    cout << "事件（顶点）      " << "最早开始时间      " << "最晚开始时间" << endl;
    for (i = 0; i < num; i++)
    {
        temp = GetValue(i+1);
        cout << temp << "                " << Ve[i] << "                  " << Vl[i]
<< endl;
    }

    //求关键活动
    cout << "活动（边）      " << "最早开始时间      " << "最晚开始时间      " << "
松弛时间      " << "判断结果" << endl;
    for ( i = 0; i < num; i++ ) //对每一个顶点为弧头进行判断
    {
        temp = GetValue(i+1);
        j = GetFirstNeighbor(temp);
        while ( j != -1 ) //对选定顶点 i 的每一条边<i,j>进行判断
        {
            temp1 = GetValue(j+1);
            e = Ve[i];                  //活动（边）的最早开始时间
            l = Vl[j] - Arcs[i][j];     //活动（边）的最早开始时间
            cout << "<" << temp << ", " << temp1 << ">            " << e << "
";
```

```
            cout << l << "                    " << l-e << "              ";
            if ( l == e )                //活动的时间余量为0
                cout << "是关键活动" << endl;
            else
                cout << "不是关键活动" << endl;
            j = GetNextNeighbor(temp, temp1);
        } // end of while
    }  // end of for

    //思考：关键活动已经得出，如何输出关键路径？

    delete InDegree; //入度数组在该过程中被申请，为避免后面的其他算法出错，退出该算法时同时释放申请的这些空间
    delete Ve;      //删除最早开始时间数组
    delete Vl;      //删除最晚开始时间数组
}

//////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////
//最小生成树的克鲁斯卡尔算法
//和教材上的区别：无须传递参数，在该算法中定义局部变量 result 用于存储最小生成树
//（除非生成树的结果需要继续使用，否则，定义为局部变量演示算法即可）
//测试数据：顶点--abcdef#；边--ab6，ac1，ad4，bc5，be3，cd5，ce6，cf4，df2，ef6；
//////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////
template<class vertexType, class arcType> void AdjacencyMatrixGraph<vertexType, arcType>::Kruskal()
{
    int numV = NumberOfVertexes();
    int i, j, u, v;
    MSTArcNodeForHeap<arcType>    **a    =    new    MSTArcNodeForHeap<arcType>*[CurrentNumArcs];
    MSTArcNode<vertexType,    arcType>    *aa    =    new    MSTArcNode<vertexType, arcType>[CurrentNumArcs];
    MinSpanTree<vertexType, arcType> result(numV);
    int count  = 0;
    MSTArcNode<vertexType, arcType> e;
    MSTArcNodeForHeap<arcType> *e1;
    vertexType *b = new vertexType[numV];

    if (numV == 0)
    {
        cout << "空图，无须处理！" << endl;
        return;
```

```
    }
    if (NumberOfArcs() == 0)
    {
        cout << "零图，无须处理！" << endl;
        return;
    }
    if (weightGraph == 1)
    {
        cout << "非带权图，没有最小代价的概念，无须处理！" << endl;
        return;
    }

    for (i = 0; i < CurrentNumArcs; i++)
        a[i] = new MSTArcNodeForHeap<arcType>;

    MinSpanTreeHeap<MSTArcNodeForHeap<arcType> *> h(CurrentNumArcs, 1);//第三版本
---1
    //将图中所有的边的完整信息置于数组中，依此构造最小堆，用于选择权值最小的边
    for (i = 0; i < numV; i++)
    {
        for (j = 0; j < numV; j++)
        {
            //if (i != j && Arcs[i][j] < edgeMaxValue) //无论有向无向，只加一条边。
有向图忽略其方向。
            if ((graphType == 1 && i < j && Arcs[i][j] < edgeMaxValue) || (graphType
== 2 && i != j && Arcs[i][j] < edgeMaxValue))
            {
                a[count]->SetWeight(Arcs[i][j]);
                a[count]->SetID(count);
                h.Insert(a[count]);                  //第三版本---2

                aa[count].SetAdiverFirst(GetValue(i+1));
                aa[count].SetAdiverSecond(GetValue(j+1));
                aa[count].SetWeight(Arcs[i][j]);

                count++;
            }
        }
        b[i] = GetValue(i+1);
    }
    /*
    //第二版本，能够编译运行，但构造的初始堆是按照输入顺序（即指针地址）
    //调用的是父类的 FiltDown 函数，而不是子类的，但插入、删除却正确
    //关于派生类与父类间的构造函数的细节语法，需要再详细查看。改第三版本
```

```
    //MinSpanTreeHeap<MSTArcNodeForHeap<arcType> *> h(a, count, 1);
    */
    cout << "所建立的最小堆为：";
    h.out();
    /*
    //第一版本：两个类型抽象，与最小堆的定义不太吻合，为了避免重新定义堆，修改为第二版本
//  MinSpanTreeHeap<MSTArcNode<vertexType, arcType> > h(a, count, 1);
*/

    //根据所有顶点构造并查集
    UFSets<vertexType> uf(b, NumberOfVertexes());

    //生成最小生成树，共 numV-1 条边，故循环 numV 次
    i = 1;

    while (i < numV)
    {
        e1 = h.DeleteTop();
        cout << "Check:" << e1->GetID() << "," << e1->GetWeight() << "; ";
        e = aa[e1->GetID()];
        u = uf.Find(e.GetFirst());
        v = uf.Find(e.GetSecond());
        if (u != v) //u,v 不在同一个并查集中，即加入(u, v)不构成环
        {
            uf.Union(u, v);
            result.Insert(e);
            i++;
        }
    }

    //输出结果
    cout << "按照克鲁斯卡尔算法生成结果---------->";
    result.Display();
}

//////////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////
//最小生成树的普里姆算法
//和教材上的区别：(1)无须传递参数，在该算法中定义局部变量 result 用于存储最小生成树
//（除非生成树的结果需要继续使用，否则，定义为局部变量演示算法即可）
//(2)和克鲁斯卡尔算法共享最小生成树及最小生成树结点的类，不另外定义，结点中的 weight 表示 lowweight，adjvex1 表示 nearvertex，adjvex2 空闲
```

```
//测试数据：顶点--abcdef#；边--ab6，ac1，ad4，bc5，be3，cd5，ce6，cf4，df2，ef6；
////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////
template<class vertexType, class arcType> void AdjacencyMatrixGraph<vertexType, arcType>::Prime(vertexType temp)
{
    int numV = NumberOfVertexes();
    int i, j, v;
    MSTArcNode<vertexType, arcType> *closearc = new MSTArcNode<vertexType, arcType>[numV];
    MinSpanTree<vertexType, arcType> result(numV);
    MSTArcNode<vertexType, arcType> e;
    arcType min;

    if (numV == 0)
    {
        cout << "空图，无须处理！" << endl;
        return;
    }
    if (NumberOfArcs() == 0)
    {
        cout << "零图，无须处理！" << endl;
        return;
    }
    if (weightGraph == 1)
    {
        cout << "非带权图，没有最小代价的概念，无须处理！" << endl;
        return;
    }

    //初始化
    j = GetVertexPos(temp); //得到起始结点的序号
    closearc[j].SetWeight(NULL);
    closearc[j].SetAdiverFirst(NULL);
    for (i = 0; i < numV; i++)
    {
        if (j != i)
        {
            closearc[i].SetAdiverFirst(GetValue(j+1));
            closearc[i].SetWeight(Arcs[j][i]);
        }
    }

    for ( i = 1; i < numV; i++ )  //循环 n-1 次, 加入 n-1 条边
```

```
    {
        //选取两个邻接顶点分别在U-V和U且具有最小权值的边---begin
        min = edgeMaxValue;
        v = -1;
        for ( j = 0; j < numV; j++ )
            if (closearc[j].GetWeight() != NULL && closearc[j].GetWeight() < min )
            {
                v = j;
                min = closearc[j].GetWeight();
            }
        //选取两个邻接顶点分别在U-V和U且具有最小权值的边--end

        if ( v  != -1)    //v == -1表示再也找不到所求的边
        {
            e.SetAdiverFirst(closearc[v].GetFirst());
            e.SetAdiverSecond(GetValue(v+1));
            e.SetWeight(closearc[v].GetWeight());
            result.Insert (e);     //把选出的边加入到生成树中
            closearc[v].SetWeight(NULL); //把顶点v加入U中
            for ( j = 0; j < numV; j++ )    //教材此处从1开始，是指定0号为开始节点。而实际上，可从任意顶点开始算法
                if   (closearc[j].GetWeight()   !=   NULL   &&   Arcs[v][j]   <  closearc[j].GetWeight() )
                {
                    // 对U-V中的每一个顶点考察是否要修改它在辅助数组中的值
                   closearc[j].SetWeight(Arcs[v][j]);
                   closearc[j].SetAdiverFirst(GetValue(v+1));
                } // end of if
        } // end of if
    }  // end of for

    //输出结果
    cout << "按照普里姆算法从顶点" << temp << "开始，生成结果---------->";
    result.Display();
}
////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////
```

///
///
///
///
///
///
///
///
///
///
///
///
///
///
///