```
//////////////////////////////////////////////////////////////////////////////
//////////////////////////
//   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 << "要癮删?除 y 的?顶Ⅴ点?不?存?在ú" << endl;
        return 0;
    }

    //统?计?删?除 y 的?边?数篛
    if (weightGraph == 1)
    {
        for (k = 0; k < m; k++) //处鋦理え?行 D
            if (Arcs[i][k] == 1)
                CurrentNumArcs--;
        if (graphType == 2) //有瓟向ò图?, ?处鋦理え?列
            for (k = 0; k < m; k++)
                if (Arcs[k][i] == 1)
                    CurrentNumArcs--;
    }
    else
    {
        for (k = 0; k < m; k++) //处鋦理え?行 D
            if (Arcs[i][k] !=0 && Arcs[i][k] != edgeMaxValue)
                CurrentNumArcs--;
        if (graphType == 2) //有瓟向ò图?, ?处鋦理え?列
            for (k = 0; k < m; k++)
                if (Arcs[k][i] != 0 && Arcs[k][i] != edgeMaxValue)
                    CurrentNumArcs--;
    }

    if (i == m-1) //删?除 y 的?顶Ⅴ点?序ò号?是?最?后ó一?个?
    {
```

```
        Vertexes.Delete(i+1, true);
        //统?计?删?除 y 的?边?数篛
        return 1;
    }

    //i 后ó的?每?一?行 D 上?移?
    for (j = i; j < m; j++)
        for (k = 0; k < m; k++)
            Arcs[j][k] = Arcs[j+1][k];
    //i 后ó的?每?一?列  左哩?移?
    for (j = i; j < m; j++)
        for (k = 0; k < m; k++)
            Arcs[k][j] = Arcs[k][j+1];
    //删?除 y 顶Ⅴ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)
{

}



////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////
//深度优先遍历
////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////
template<class vertexType, class arcType> void AdjacencyMatrixGraph<vertexType,
arcType>::DFTraverse()
{

}

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

```
//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)
{


}

////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////
//广度优先遍历
////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////
template<class vertexType, class arcType> void AdjacencyMatrixGraph<vertexType, arcType>::BFTraverse()
{

}

////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////
//判断图是否连通
////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////
template<class vertexType, class arcType> bool AdjacencyMatrixGraph<vertexType, arcType>::IsConnected()
{

}

////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////
//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)
{
}
```

```
//////////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////
//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)
{

}

//////////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////
//弗洛伊德(Floyd)算法
```

```
//////////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////
template<class vertexType, class arcType> void AdjacencyMatrixGraph<vertexType,
arcType>::Floyd()
{

}

//////////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////
//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;
    }
}

//////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////
//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;
    }

}

//////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////
//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;

    //不是有向图，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 (begin == -1)
            cout << "不存在，没有汇点的图无法求关键路径，";
        else
            if (begin == -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;
    }


}

//////////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////
//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;

}

//////////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////
//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)
```

{

}