SHANGHAI JIAO TONG UNIVERSITY

最小生成树问题

吕艺

517021910745

December 27, 2018

1 需求分析

1. 本演示文件的主要需求为几个城市之间的通信网络架构问题，主要背景是在 n 个城市之间以最低的经济代价建设通信网络，是一个最小生成树的问题。

2. 演示文件需用户输入，分为用户自行输入数据和自动运行默认数据两种选择。文件运行结束后即在计算机终端上打印路径和最小代价，显示在屏幕上。

3. 程序执行的命令包括:

a) 用户输入命令（默认参数或手动输入）

b) 根据数据建立边表

c) 利用 Prim 算法生成最小生成树

d) 将最小生成树以边的文本形式打印在屏幕上，并打印最小代价

4. 测试数据

顶点集：V={a,b,c,d,e,f,g,h}

边集:E={(a,b,4),(a,c,3),(b,c,5),(b,d,5),(b,e,9),(c,d,5)
(c,h,5),(d,e,7),(d,f,6),(d,g,5),(d,h,4),(e,f,3),(f,g,2),(g,h,6)}

2 概要设计

本演示文件的主要目标是实现一个最小生成树问题，通过给定的节点以及边的代价，找到遍历节点且代价最低的路径。为了实现这一功能，本演示文件中设计了父类 *graph* 和派生类 *adjListGraph*。

1. 父类 *graph*

数据对象：int Ver;　int Edges;

基本操作：virtual bool insert(int s,int e,TypeOfEdge w) = 0;
virtual bool remove(int s, int e) = 0;
virtual bool exist(int s, int e) const = 0;

2. 派生类 *adjListGraph*

数据对象：verNode *verList;　int Vers;　int Edges;

私有类：struct edgeNode {
int end;
TypeOfEdge weight;
edgeNode(int e, TypeOfEdge w, edgeNode *n = NULL)
end = e; weight = w; next = n;
};
struct verNode {
TypeOfVer ver;
edgeNode *head;
verNode(edgeNode *h = NULL) head = h ;
};

基本操作：adjListGraph(int vSize, const TypeOfVer d[]);
操作条件：边表还未被初始化
操作结果：利用传入参数初始化边表
bool insert(int u, int v, TypeOfEdge w);
初始条件：边表已被初始化
操作结果：插入边

bool remove(int u, int v);
初始条件：边表已存在这条边
操作条件：删除中的指定边
bool exist(int u, int v) const;
操作条件：边表已被初始化
操作结果：寻找边表中是否存在特定边
adjListGraph() ;
操作条件：边表已被初始化
操作结果：析构边表，释放内存
void prim(TypeOfEdge noEdge) const;
操作条件：边表已被初始化
操作结果：输出边和最小代价

2. 本程序包括两个模块：

1) int main() {
input order
switch(order){
order 1: process default data and print result
order 2: input data and print result }

2) 边表单元模块--实现边表的创建以及生成最小生成树

3 详细设计

1）边表单元模块

```
template<class TypeOfEdge>
class graph{
protected:
    int Ver,Edges;                          //size and num of edges
public:
    virtual bool insert(int s,int e,TypeOfEdge w) = 0;      //插入边
    virtual bool remove(int s, int e) = 0;                  //删除边
    virtual bool exist(int s, int e) const = 0;             //判断指定边是否存在
};


template <class TypeOfVer, class TypeOfEdge>
class adjListGraph:public graph<TypeOfEdge> {
public:
    adjListGraph(int vSize, const TypeOfVer d[]);           //构造空表
    bool insert(int u, int v, TypeOfEdge w);                //插入边
    bool remove(int u, int v);                              //删除边
```

```
    bool exist(int u, int v) const;                          //判断指定边是否存在
    ~adjListGraph() ;                                        //析构函数，释放内存
    void prim(TypeOfEdge noEdge) const;                      //生成最小生成树

private:
    struct edgeNode {                //邻接表中存储边的结点类
        int end;                     //终点编号
        TypeOfEdge weight;           //边的权值
        edgeNode *next;
        edgeNode(int e, TypeOfEdge w, edgeNode *n = NULL)
        { end = e; weight = w; next = n;}
    };
    struct verNode{                  //保存顶点的数据元素类型
        TypeOfVer ver;               //顶点值
        edgeNode *head;              //对应的单链表的头指针
        verNode( edgeNode *h = NULL)  { head = h ;}
    };
    verNode *verList;
    int Vers;
    int Edges;
};


template <class TypeOfVer, class TypeOfEdge>
adjListGraph<TypeOfVer, TypeOfEdge>::adjListGraph(int vSize,  const TypeOfVer d[])
{   Vers = vSize;
    Edges = 0;

    verList = new verNode[vSize];
    for (int i = 0; i < Vers; ++i) verList[i].ver = d[i];    //初始化边表
}


template <class TypeOfVer, class TypeOfEdge>
adjListGraph<TypeOfVer, TypeOfEdge>::~adjListGraph()
{ int i;
    edgeNode *p;
    for (i = 0; i < Vers; ++i)
        while ((p = verList[i].head) != NULL){               //释放边表内存
            verList[i].head = p->next;
            delete p;
        }
    delete [] verList;
}


template <class TypeOfVer, class TypeOfEdge>
bool adjListGraph<TypeOfVer, TypeOfEdge>::
insert(int u, int v, TypeOfEdge w)                          //插入指定边
{  verList[u].head =
```

```
            new edgeNode(v, w, verList[u].head );
    ++Edges;
    return true;
}


template <class TypeOfVer, class TypeOfEdge>
bool adjListGraph<TypeOfVer, TypeOfEdge>::remove(int u, int v)
{edgeNode *p = verList[u].head, *q;
    if (p == NULL) return false;            //结点u没有相连的边
    if (p->end == v)                        //单链表中的第一个结点就是被删除的边
    { verList[u].head = p->next;
        delete p; --Edges;
        return true;    }
    while (p->next !=NULL && p->next->end != v) p = p->next;
    if (p->next == NULL) return false;      //没有找到被删除的边
    q = p->next; p->next = q->next; delete q;
    --Edges;
    return true;
}


template <class TypeOfVer, class TypeOfEdge>
bool adjListGraph<TypeOfVer, TypeOfEdge>::exist(int u, int v) const
{ edgeNode *p = verList[u].head;                        //判断指定边是否存在
    while (p !=NULL && p->end != v) p = p->next;
    if (p == NULL) return false; else  return true;
}


template <class TypeOfVer, class TypeOfEdge>
void adjListGraph<TypeOfVer, TypeOfEdge>::prim
        (TypeOfEdge noEdge) const               //生成最小生成树
{   int total_weight = 0;
    bool *flag = new bool[Vers];
    TypeOfEdge *lowCost = new TypeOfEdge[Vers];
    int *startNode = new int[Vers];
    edgeNode *p;
    TypeOfEdge min;
    int start, i, j;

    for (i = 0; i < Vers; ++i) {                //  初始化点和边
        flag[i] = false;
        lowCost[i] = noEdge;  }
    start = 0;
    for ( i = 1; i < Vers; ++i) {
        for (p = verList[start].head; p != NULL; p = p->next)
            if (!flag[p->end] && lowCost[p->end] > p->weight) {
                lowCost[p->end] = p->weight;
                startNode[p->end] = start;  }
```

```
        flag[start] = true;                 // 选取最小代价的点
        min = noEdge;
        for (j = 0; j < Vers; ++j)
        {if (lowCost[j] < min) {min = lowCost[j]; start = j;}}

        total_weight += min;
        cout << '(' << verList[startNode[start]].ver << ","
             << verList[start].ver << ")\t";     // 选取新的节点
        lowCost[start] = noEdge;
    }
    cout<<endl;
    cout<<"Total cost:"<<endl;
    cout<<total_weight<<endl;
    delete [] flag;
    delete [] startNode;
    delete [] lowCost;
}
```

2）主函数模块

```
int main() {
    int order;
    cout<<"Please input order: (1: default data 2: input data) "<<endl;              //用户输入命令
    cin>>order;
    switch(order)
    {
        case 1 :                                    //使用测试数据
        {char value[9] = {'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'};
            adjListGraph<char, int> A(8, value);
            A.insert(0, 1, 4);    //a b 4           //因为Prim算法利用的是无向图
            A.insert(1, 0, 4);                      //所以同一条边插入两次
            A.insert(0, 2, 3);    //a c 3
            A.insert(2, 0, 3);
            A.insert(1, 2, 5);    //b c 5
            A.insert(2, 1, 5);
            A.insert(1, 3, 5);    //b d 5
            A.insert(3, 1, 5);
            A.insert(1, 4, 9);    //b e 9
            A.insert(4, 1, 9);
            A.insert(2, 3, 5);    //c d 5
            A.insert(3, 2, 5);
            A.insert(2, 7, 5);    //c h 5
            A.insert(7, 2, 5);
            A.insert(3, 4, 7);    //d e 7
            A.insert(4, 3, 7);
            A.insert(3, 5, 6);    //d f 6
            A.insert(5, 3, 6);
            A.insert(3, 6, 5);    //d g 5
            A.insert(6, 3, 5);
            A.insert(3, 7, 4);    //d h 4
            A.insert(7, 3, 4);
```

```
            A.insert(4, 5, 3);     //e f 3
            A.insert(5, 4, 3);
            A.insert(5, 6, 2);     //f g 2
            A.insert(6, 5, 2);
            A.insert(6, 7, 6);     //g h 6
            A.insert(7, 6, 6);
            A.prim(100); break;}            //调用prim算法生成最小生成树并打印路径和最小代价

        case 2:
        {int num;
            cout<<"Please input num:"<<endl;
            cin>>num;                                   //输入元素数
            char * d;
            cout<<"Please input characters:"<<endl;
            d = new char[num];                          //输入元素
            cin>>d;
            adjListGraph<char, int> B(num+1, d);
            int start,end,weight;
            cout<<"Please input edges:"<<endl;
            while(true)
            {   cin>>start;                             //输入边的起点，终点和权值
                if(start == -1) break;
                cin>>end>>weight;
                B.insert(start,end,weight);
                B.insert(end,start,weight);
            }
            B.prim(100);            //调用prim算法生成最短路径和最小代价
            break;}
    }
    return 0;
}
```

4 调试分析

一开始由于对输入情况考虑的欠缺，本演示文件一开始把边表的节点值设为了字符。为了加强类的泛化能力，本演示文件采用模板类以及模板函数，从而使文件的适用范围更广。

3. 算法的复杂度分析

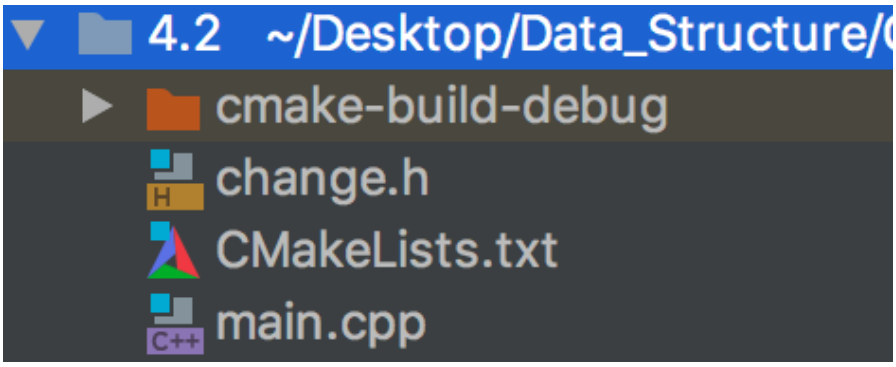
1)时间复杂度由于采用边表的形式存储,各种操作的算法复杂度比较合理。*adjListGraph* 函数的时间复杂度为O(v)，主要取决于新建边表时需要赋值的元素个数。*adjListGraph* 函数的时间复杂度取决于需要释放的元素个数和边条数，故为O（V+E)。*insert* 函数的时间复杂度为O（1），将插入的节点作为链表的头结点。*remove* 函数的时间复杂度取决于删除的边在边表中的位置，所以为O（E），*exist* 函数的时间复杂度取决于指定查找元素在边表中的位置，所以为O（E）*prim* 函数的时间复杂度主要取决于O（E*V），取决于遍历的次数。

2）空间复杂度

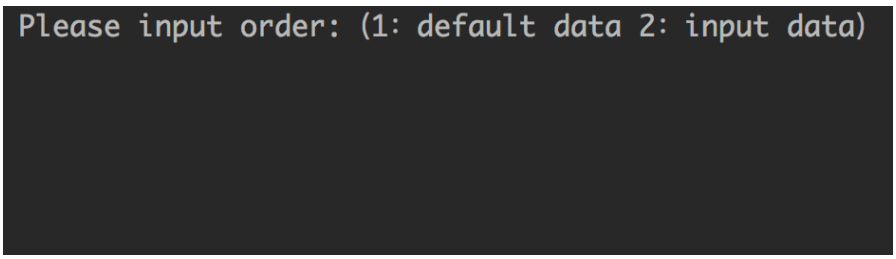
边表模块的空间复杂度与节点的个数与边的条数成正比，即 $O(V*E)$。主函数模块的复杂度取决于定义主函数作用域中的边表，故空间复杂度也为 $O(V*E)$。

5 用户手册

1. 本程序以 Jetbrains Clion 2018.2.5, 采用 C++ 11 标准，程序以项目方式组织 (project)，如图 1 所示：



2. 依次点击菜单"Run"->build, 再点击"Run", 显示文本方式的用户界面, 如图 2 所示:

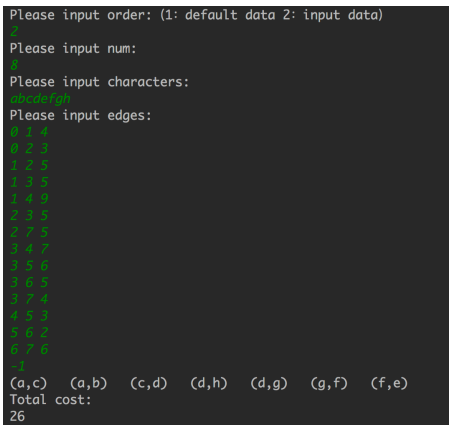


3. 键入操作命令符，1 代表使用测试数据，2 代表用户自定义参数，之后按“回车键”。程序就执行相应命令。

6 测试结果

键入命令 1 后自动打印默认参数结果

键入命令 2 后依次输入元素个数，元素值，边（起点终点权值），并以空格分开，边输入结束后输入-1 以结束输入。



7 附录

源程序文件名清单

main.cpp //主函数

adjgraph.h //边表单元模块