项目说明文档

数据结构课程设计

——电网建设造价模拟系统

作 者 姓 名： 杨煜

学 号： 1850217

指 导 教 师： 张颖

学院、 专业： 软件学院 软件工程

同济大学

Tongji University

目 录

[1 分析 1](#_Toc26873087)

[1.1 项目简介 1](#_Toc26873088)

[1.2 项目要求 1](#_Toc26873089)

[2 设计 1](#_Toc26873090)

[2.1 数据结构设计 1](#_Toc26873091)

[2.2 图结构设计 1](#_Toc26873092)

[2.3 成员与操作设计 1](#_Toc26873093)

[3 主要功能实现 5](#_Toc26873094)

[3.1 构造函数的实现 5](#_Toc26873095)

[3.1.1 构造函数 5](#_Toc26873096)

[3.1.2 构造函数核心代码 6](#_Toc26873097)

[3.2 析构函数的实现 6](#_Toc26873098)

[3.2.1 析构函数 6](#_Toc26873099)

[3.2.2 析构函数核心代码 6](#_Toc26873100)

[3.3 获取首个邻接顶点的实现 7](#_Toc26873101)

[3.3.1 获取首个邻接顶点 7](#_Toc26873102)

[3.3.2 获取首个邻接顶点核心代码 7](#_Toc26873103)

[3.4 获取下一个顶点的实现 7](#_Toc26873104)

[3.4.1 获取下一个顶点 7](#_Toc26873105)

[3.4.2 获取下一个顶点核心代码 7](#_Toc26873106)

[3.5 插入顶点函数的实现 8](#_Toc26873107)

[3.5.1 插入顶点 8](#_Toc26873108)

[3.5.2 插入顶点核心代码 8](#_Toc26873109)

[3.6 插入边的实现 9](#_Toc26873110)

[3.6.1 插入边 9](#_Toc26873111)

[3.6.2 插入边核心代码 9](#_Toc26873112)

[3.7 Prim算法的实现 11](#_Toc26873113)

[3.7.1 Prim算法 11](#_Toc26873114)

[3.7.2 Prim算法核心代码 11](#_Toc26873115)

[3.8 总体的实现 12](#_Toc26873116)

[3.8.1 总体流程 12](#_Toc26873117)

[3.8.2 总体核心代码 12](#_Toc26873118)

[3.8.2 总体核心流程图 16](#_Toc26873119)

[4 测试 16](#_Toc26873120)

[4.1 功能测试 16](#_Toc26873121)

[4.1.1 一般情况 16](#_Toc26873122)

[4.2 出错测试 17](#_Toc26873123)

[4.2.1 操作码不正确 17](#_Toc26873124)

# 1 分析

## 1.1 项目简介

假设一个城市有n个小区，要实现n个小区之间的电网都能够相互接通，构造这个城市n个小区之间的电网，使总工程造价最低。请设计一个能够满足要求的造价方案。

## 项目要求

在每个小区之间都可以设置一条电网线路，都要付出相应的经济代价。n个小区之间最多可以有n（n-1）/2条线路，选择其中的n-1条使总的耗费最少。

# 2 设计

## 2.1 数据结构设计

如上述项目简介所述，图这一数据结构是解决问题的合理的方法，因为项目要求中有顶点，有邻边。这种顶点与邻边的关系，图无疑是最为合适的。在建立图结构之后，我们需要更进一步的建立最小生成树。为此，我们需要使用Prim算法来构成最小生成树。在构成最小生成树时，为了效率的考量需要使用最小堆来存储最小权值的边。

## 2.2 图结构设计

图的存储方式有两种形式，一为邻接矩阵，一位邻接表。如果使用邻接矩阵来存储的话，可能造成空间上的浪费，搜索效率上也较差，需要O（n^2）的时间复杂度。故而使用邻接表是一个来存储图结构，是一个较优的选择。

## 2.3 成员与操作设计

**边节点类**

template<class T, class E>

struct Edge {

int dest;//指向的顶点

E cost;//权值

Edge<T, E>\* link;

Edge() {}

Edge(T num, E weight)

{

dest = num;

cost = weight;

link = NULL;

}

bool operator!= (Edge<T,E>& R) const

{

return (dest != R.dest) ? true : false;

}

};

**顶点节点类**

template<class T, class E>

struct Vertex

{

T data;//数据

Edge<T, E>\* adj;//边

};

**图的邻接表类**

template<class T, class E>

class Graphlnk

{

public:

Graphlnk(int sz=50);//构造函数

~Graphlnk();//析构函数

T getValue(int i) //获取权值

{

return (i >= 0 && i < numVertices) ? NodeTable[i].data : '0';

}

E getWeight(int v1, int v2);//两个节点间的权值

int getFirstNeighbor(int v);//第一个相邻的顶点

int getNextNeighbor(int v, int w);//下一个相邻的顶点

bool addVertexes(int& numCount);//加入节点

void addEdges();//加入边

int getnumVertices()//获取节点个数

{

return numVertices;

}

int getnumEdges()//获取边个数

{

return numEdges;

}

int getVertexPos(const T vertex)//查找节点位置

{

for (int i = 0; i < numVertices; i++)

{

if (NodeTable[i].data == vertex)

{

return i;

}

}

return -1;

}

private:

bool insertEdge(int v1, int v2, E cost);//插入边

Vertex<T, E>\* NodeTable;//图的链表

int numVertices;//节点个数

int maxVertices;//最大节点个数

int numEdges; //边个数

};

**最小生成树类**

template<class T, class E>

struct MSTEdgeNode //节点

{

int tail, head;

E key;

MSTEdgeNode() : tail(-1), head(-1), key(0) {}

bool operator<= (MSTEdgeNode<T, E>& R) { return key <= R.key; }

bool operator> (MSTEdgeNode<T, E>& R) { return key > R.key; }

};

template<class T, class E>

class MinSpanTree//最小生成树类

{

public:

MSTEdgeNode<T, E>\* edgevalue;

MinSpanTree(int sz = 50 - 1) : maxSize(sz), n(0)

{

edgevalue = new MSTEdgeNode<T, E>[sz];

}

bool InsertNode(MSTEdgeNode<T, E>& node)

{

if (n == maxSize)

{

cerr << "The MinSpanTree is full" << endl;

return false;

}

else

{

edgevalue[n] = node;

n++;

return true;

}

}

protected:

int maxSize, n;

};

**最小堆类**

template<class T, class E>

class Heap

{

public:

Heap(int size = DefaultVerticesNum);

~Heap() { delete[]heap; }

bool insert(const MSTEdgeNode<T, E>& x);

bool remove(MSTEdgeNode<T, E>& item);

void siftDown(int start, int end);

void siftUp(int start);

int length() { return currentSize; }

bool isEmpty() { return currentSize == 0; }

private:

MSTEdgeNode<T, E>\* heap;

int currentSize;

int maxSize;

};

# 3 主要功能实现

## 3.1 构造函数的实现

### 3.1.1 构造函数

构造函数构造函数接受一个参数，依次开辟节点空间，将每个节点的邻接的指针置为NULL。

### 3.1.2 构造函数核心代码

template<class T, class E>

Graphlnk<T, E>::Graphlnk(int sz)

{

maxVertices = sz;

numEdges = 0;

numVertices = 0;

NodeTable = new Vertex<T, E>[maxVertices];//开辟空间

for (int i = 0; i < maxVertices; i++)

{

NodeTable[i].adj = NULL;//置空

}

}

## 3.2 析构函数的实现

### 3.2.1 析构函数

析构函数将节点的所用邻接关系一一清除，最后删去这个存放顶点的数组空间

### 3.2.2 析构函数核心代码

template<class T, class E>

Graphlnk<T, E>::~Graphlnk()

{

for (int i = 0; i < numVertices; i++)

{

Edge<T, E>\* p = NodeTable[i].adj;

while (p != NULL)//清除链表

{

NodeTable[i].adj = p->link;

delete p;

p = NodeTable[i].adj;

}

}

delete[]NodeTable;//删去节点数组

}

## 3.3 获取首个邻接顶点的实现

### 3.3.1 获取首个邻接顶点

获取输入顶点的首个邻接顶点。如果没有则返回-1。

### 3.3.2 获取首个邻接顶点核心代码

template<class T, class E>

int Graphlnk<T, E>::getFirstNeighbor(int v)

{

if (v != -1) {

Edge<T, E>\* p = NodeTable[v].adj;

if (p != NULL)

return p->dest;

}

return -1;

}

## 3.4 获取下一个顶点的实现

### 3.4.1 获取下一个顶点

接受两个参数，找到当前顶点v的，在w之后的一个顶点。

### 3.4.2 获取下一个顶点核心代码

template<class T, class E>

int Graphlnk<T, E>::getNextNeighbor(int v, int w)

{

if (v != -1)

{

Edge<T, E>\* p = NodeTable[v].adj;

while (p != NULL && p->dest != w)

{

p = p->link;

}

if (p != NULL && p->link != NULL)

{

return p->link->dest;

}

}

return -1;

}

## 3.5 插入顶点函数的实现

### 3.5.1 插入顶点

插入一个顶点

### 3.5.2 插入顶点核心代码

template<class T, class E>

bool Graphlnk<T, E>::addVertexes(int& numCount)

{

cout << "请输入顶点的个数：";

int n;

cin >> n;

numCount = n;

cout << "请输入各个顶点的名称：";

while (n > 0) {

char name;

cin >> name;

if (numVertices == maxVertices)

{

return false;

}

else

{

NodeTable[numVertices].data = name;

numVertices++;

}

n--;

}

}

## 3.6 插入边的实现

### 3.6.1 插入边

读入所需要插入的边，直到接收到末尾标识符。找到要插入边的节点，随后进行插入边的操作。

### 3.6.2 插入边核心代码

template<class T, class E>

void Graphlnk<T, E>::addEdges()

{

int loop = 1;

while (loop) {

cout << "请输入两个顶点及边：";

char v1, v2;

int weight;

cin >> v1 >> v2 >> weight;

if (v1 == '?' || v2 == '?') {

loop = 0;

break;

}

int v1\_pos = getVertexPos(v1);

int v2\_pos = getVertexPos(v2);

if (!insertEdge(v1\_pos, v2\_pos, weight)) {

cerr << "顶点创建失败，请检查后重新输入" << endl;

}

}

}

template<class T, class E>

bool Graphlnk<T, E>::insertEdge(int v1, int v2, E weight)

{

if (v1 >= 0 && v1 < numVertices && v2 >= 0 && v2 < numVertices)

{

Edge<T, E>\* q;

Edge<T, E>\* p = NodeTable[v1].adj;

while (p != NULL && p->dest != v2)

{

p = p->link;

}

if (p != NULL)

{

return false;

}

p = new Edge<T, E>;

q = new Edge<T, E>;

p->dest = v2;

p->cost = weight;

p->link = NodeTable[v1].adj;

NodeTable[v1].adj = p;

q->dest = v1;

q->cost = weight;

q->link = NodeTable[v2].adj;

NodeTable[v2].adj = q;

numEdges++;

return true;

}

return 0;

}

## 3.7 Prim算法的实现

### 3.7.1 Prim算法

将顶点分为两个不重叠的部分。每一轮的迭代中，找到当前割的桥，挑选出其中的最小值。随后将顶点加入到生成树的顶点集合之中，不断重复即可得到最小生成树。

### 3.7.2 Prim算法核心代码

template<class T, class E>

void Prim(const T u0, Graphlnk<T, E>& G, MinSpanTree<T, E>& MST)

{

MSTEdgeNode<T, E> ed;

int i, u, v, count;

int n = G.getnumVertices();

int m = G.getnumEdges();

u = G.getVertexPos(u0);

Heap<T, E> H(m);

bool\* Vmst = new bool[n];

for (int i = 0; i < n; i++)

{

Vmst[i] = false;

}

Vmst[u] = true;

count = 1;

while (count < n)

{

v = G.getFirstNeighbor(u);

while (v != -1)

{

if (Vmst[v] == false)

{

ed.head = v;

ed.tail = u;

ed.key = G.getWeight(u, v);

H.insert(ed);

}

v = G.getNextNeighbor(u, v);

}

while (!(H.isEmpty()) && count < n)

{

H.remove(ed);

if (Vmst[ed.head] == false)

{

MST.InsertNode(ed);

u = ed.head;

Vmst[u] = true;

count++;

break;

}

}

}

}

## 3.8 总体的实现

### 3.8.1 总体流程

首先建立表格，随后通过操作码，switch分支，进行相应的操作。

### 3.8.2 总体核心代码

int main() {

cout << "\*\*" << setw(10) << ' ' << "电网造价模拟系统" << setw(10) << ' ' << "\*\*" << endl;

cout << "========================================" << endl;

cout << "\*\*" << setw(9) << ' ' << "A --- 创建电网顶点" << setw(9) << ' ' << "\*\*" << endl;

cout << "\*\*" << setw(9) << ' ' << "B --- 添加电网的边" << setw(9) << ' ' << "\*\*" << endl;

cout << "\*\*" << setw(9) << ' ' << "C --- 构造最小生成树" << setw(7) << ' ' << "\*\*" << endl;

cout << "\*\*" << setw(9) << ' ' << "D --- 显示最小生成树" << setw(7) << ' ' << "\*\*" << endl;

cout << "\*\*" << setw(9) << ' ' << "E --- 退出程序" << setw(13) << ' ' << "\*\*" << endl;

cout << "========================================" << endl;

Graphlnk<char, int> G;

MinSpanTree<char, int> MST;

int numVertices = 0;

string option;

int loop = 1;

int Init = 0;

while (loop) {

cout << endl << "请选择操作：";

cin >> option;

switch (option[0]) {

case 'A':

if (Init == 1) {

cerr << "已有电网存在，不要重复创建" << endl;

}

else {

G.addVertexes(numVertices);

Init = 1;

}

break;

case 'B':

if (Init != 1) {

cout << "请先创建电网" << endl;

}

else {

G.addEdges();

}

break;

case 'C':

if (Init != 1) {

cout << "请先创建电网" << endl;

break;

}

cout << "请输入起始顶点：";

char begin;

cin >> begin;

Prim(begin, G, MST);

cout << "生成Prim最小生成树!" << endl;

break;

case 'D': {

if (Init != 1) {

cout << "请先创建电网" << endl;

break;

}

cout << "最小生成树的顶点及边分别为：" << endl;

for (int i = 0; i < numVertices - 1; i++) {

char head = G.getValue(MST.edgevalue[i].tail);

char tail = G.getValue(MST.edgevalue[i].head);

cout << head << "-<" << MST.edgevalue[i].key << ">->" << tail;

if (i != numVertices - 2) {

cout << "\t";

}

}

cout << endl;

break;

}

case 'E':

loop = 0;

break;

default:

cout << "选项不存在，请重新输入" << endl;

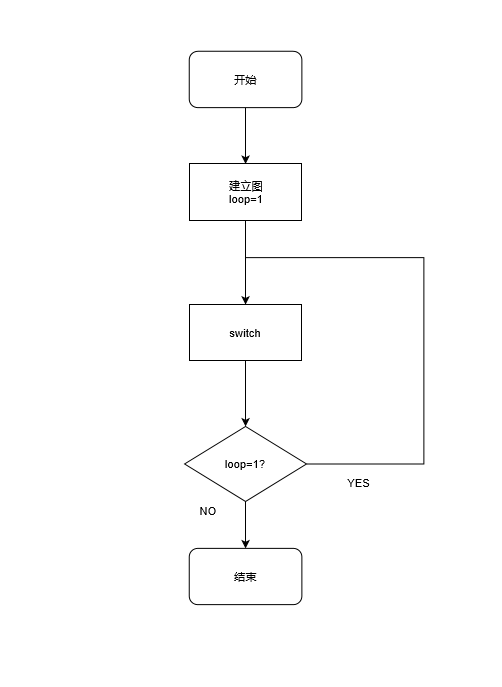
break;

}

}

}

### 3.8.2 总体核心流程图

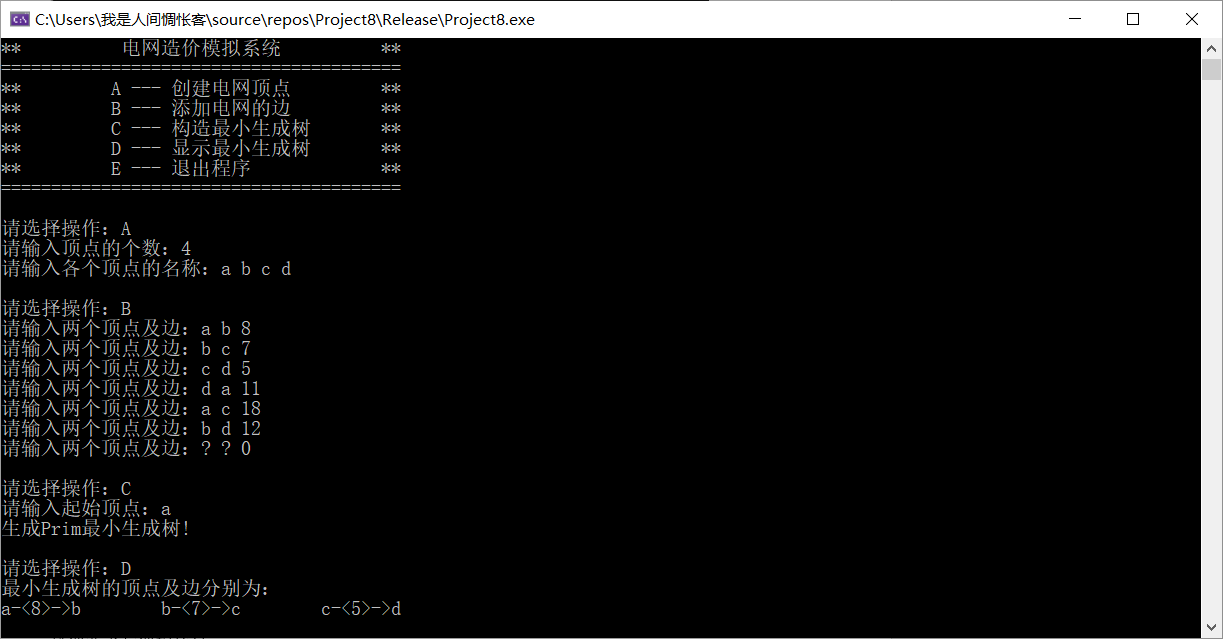


# 4 测试

## 4.1 功能测试

### 4.1.1 一般情况

**实验结果:**



## 4.2 出错测试

### 4.2.1 操作码不正确

**测试用例：**

F

**预期结果：**程序给出提示信息，程序正常运行不崩溃。

**实验结果：**

