离散数学

项目说明文档

**最小生成树(Kruskal算法)**



同济大学

Tongji University

姓名： 林觉凯

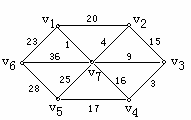
学号： 2253744

指导老师： 李冰

学院专业： 软件学院 软件工程

**1.实验背景介绍**

如下图所示的赋权图表示某七个城市，预先计算出它们之间的一些直接通信道路造价（单位：万元），试给出一个设计方案，使得各城市之间既能够保持通信，又使得总造价最小，并计算其最小值。



七个城市赋权图

**2.实验要求**

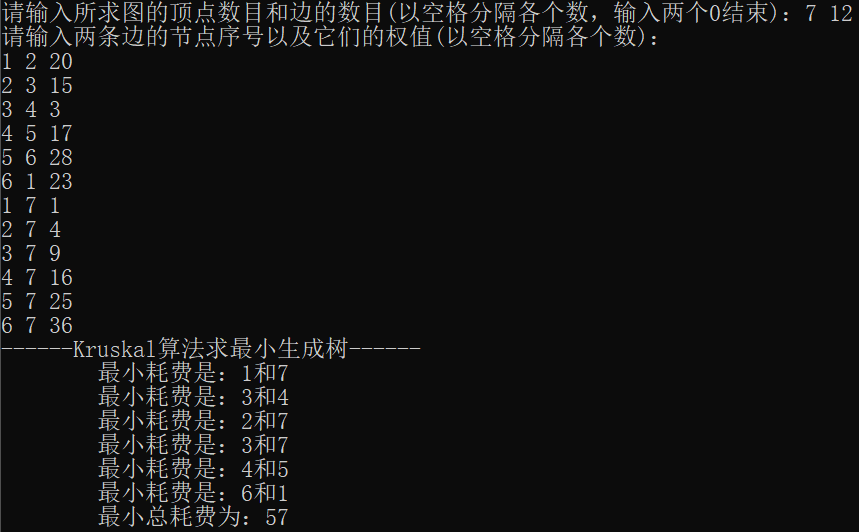
**2.1 输入要求**

要求用户输入城市个数（即图中的节点个数）和通信道路条数（即图中的边数），接着按照用户输入的具体节点个数和边数，输入正确的每一条边的起始点、结束点和相应的权值（道路造价）。

**2.2 输出要求**

构造出最小生成树，并显示要按照哪些条边来构建和最小耗费总值。

**2.3 项目示例**



**3.数据结构设计**

本题我重新全部写了一遍。要实现图的各种函数操作，首先要选择一种数据结构来表示图，本程序选择了用邻接矩阵来表示。我设计了两个类：边界点类（Edgenode）和图类（Graph），图类中使用动态内存分配给邻接矩阵分配空间（int\*\* Matrix），使用动态内存分配给边节点数组分配空间（Edgenode\* Edges），使用动态内存分配给顶点点序号数组分配空间（int\* Vertices）；同时我使用了并查集（UFSets）这一数据结构来判断两个节点是否在同一集合内。

**4.项目实现**

**4.1输入和初始化部分的实现**

在输入节点数和边数的时候我考虑到需要考虑多种输入错误情况，使用我写了应该Get\_node\_edge\_numbers函数来获得正确的节点数和边数，其中考虑了多种输入错误：比如有数据类型输入错误、节点数小于2、边数小于1、边数大于节点数\*（节点数-1）

void Get\_node\_edge\_numbers(int& Vertices\_num, int& Edges\_num)

{

while (1)

{

cin >> Vertices\_num >> Edges\_num;

//输入错误处理：（数据类型输入错误、节点数小于2、边数小于1、边数大于节点数\*（节点数-1））

if (cin.good() == 0 || Vertices\_num < 2 || Edges\_num < 1 || Edges\_num > Vertices\_num \* (Vertices\_num - 1) / 2)

{

cin.clear();

cin.ignore(1024, '\n');

cout << "输入错误！请重新输入！" << endl;

continue;

}

break;

}

}

初始化部分主要包含两个方面：输入正确的边权重和初始化数组。在初始化数组的时候，将顶点数组下标初始化为其下标值，初始化邻接矩阵时对角线的权值为0，其他都为最大权值；在输入正确的边权重时同时做了相应的代码健壮性检测使得输入的数据正确可信。这里需要注意一个点，在最后将Startnode和Endnode（即顶点序号）的时候不要和下标混淆，下标数值应该=顶点序号数值-1，不然在后续的操作时会导致越界访问。

Graph::Graph()

{

//输入顶点数目和边数

cout << "请输入所求图的顶点数目和边的数目(以空格分隔各个数，输入两个0结束)：";

Get\_node\_edge\_numbers(Vertices\_num, Edges\_num);

//初始化顶点数组

Vertices = new int[Vertices\_num];

for (int i = 0; i < Vertices\_num; i++)

Vertices[i] = i;

//初始化边数组

Edges = new Edgenode [Edges\_num];

//初始化邻接矩阵（对角线的权值为0，其他都为最大权值）

Matrix = new int\*[Vertices\_num];

for (int i = 0; i < Vertices\_num; i++)

{

Matrix[i] = new int[Vertices\_num];

for (int j = 0; j < Vertices\_num; j++)

{

if (i == j)

Matrix[i][j] = 0;

else

Matrix[i][j] = MAXWEIGHT;

}

}

//输入边的起点、终点和权重

cout << "请输入两条边的节点序号以及它们的权值(以空格分隔各个数)：" << endl;

for (int i = 0; i < Edges\_num; i++)

{

int Startnode, Endnode, Edgeweight;

cin >> Startnode >> Endnode >> Edgeweight;

cin.ignore(1024, '\n');

//输入的代码健壮性检验

if (cin.good() == 0 || Startnode < 1 || Startnode > Vertices\_num || Endnode < 1 || Endnode>Vertices\_num)

{

cin.clear();

cin.ignore(1024, '\n');

cout << "输入错误！请重新输入！" << endl;

i--;

continue;

}

Edges[i].start = Startnode;

Edges[i].end = Endnode;

Edges[i].weight = Edgeweight;

//这里要注意矩阵位置和序号的关系，矩阵位置等于节点序号-1

Matrix[Startnode - 1][Endnode - 1] = Matrix[Endnode - 1][Startnode - 1] = Edgeweight;

}

}

**4.2 Kruskal算法的实现**

Kruskal算法的的思想主要为“找边”，首先我们将每一条边根据其权重从小到大排序，接着遍历每一条边，同时设置应该并查集，（这里相对于把最小生成树中的每一条边都放在同一个集合内，每一次找当前树外权值最小边时都要判断该边是否在并查集内），每次找到了相对路径都要加到总耗费中。遍历完之后，打印最终结果。

void Graph::Create\_tree\_by\_Kruskal()

{

cout << "------Kruskal算法求最小生成树------" << endl;

//将边的权重按小到大排序

for (int i = 0; i < Edges\_num; i++)

{

for (int j = i + 1; j < Edges\_num; j++)

{

if (Edges[i].weight > Edges[j].weight)

{

int temp\_start = Edges[i].start;

Edges[i].start = Edges[j].start;

Edges[j].start = temp\_start;

int temp\_end = Edges[i].end;

Edges[i].end = Edges[j].end;

Edges[j].end = temp\_end;

int temp\_weight = Edges[i].weight;

Edges[i].weight = Edges[j].weight;

Edges[j].weight = temp\_weight;

}

}

}

//初始化并查集

int\* UFSets = new int[Vertices\_num];

for (int i = 0; i < Vertices\_num; i++)

UFSets[i] = i;

int Mincost = 0;

for (int i = 0; i < Edges\_num; i++)

{

int Startnode = Edges[i].start - 1;

int Endnode = Edges[i].end - 1;

int Edgeweight = Edges[i].weight;

//如果两个节点不属于同一个集合

if (FindUFSets(UFSets, Startnode) != FindUFSets(UFSets, Endnode))

{

//将其规划入最小生成树内，同时加上该路径的耗费

Union(UFSets, Startnode, Endnode);

Mincost += Edges[i].weight;

cout << "\t" << "最小耗费是：" << Startnode + 1 << "和" << Endnode + 1 << endl;

}

}

cout << "\t" << "最小总耗费为：" << Mincost << endl;

//动态释放空间，以免造成内存的浪费

delete[] UFSets;

}

**4.3并查集相关函数的实现**

并查集操作主要有两个函数，第一是寻找某一个节点的父节点，即一直向上循环查找，知道找到父节点为止；第二是合并两个集合，即找到两个元素节点所在其集合的父节点，将其中应该父节点指向来歪歪一个父节点即可完成该操作。

int Graph::FindUFSets(int\* UFSets, int node)

{

//循环向上查找父节点（祖先）

while (UFSets[node] != node)

node = UFSets[node];

return node;

}

void Graph::Union(int\* UFSets, int node1, int node2)

{

int Parent1 = FindUFSets(UFSets, node1);

int Parent2 = FindUFSets(UFSets, node2);

//node1的祖先指向node2的祖先，从而完成两个集合的合并

UFSets[Parent1] = Parent2;

}

**5.实验心得**

这次项目相对于前几个实验比较复杂，所涉及的Kruskal算法构建最小生成树用语言描述比较简单，但是通过C++代码来表示就比较复杂。我们需要设计多种数据结构来表示图、表示并查集、表示最小生成树，这些在代码中都是需要仔细打磨，反复debug的。在这次项目中，我对最小生成树的理论知识有了更好的认识：（最小生成树构建所用的Kruskal算法实际上是“找边”的过程，将树外符合条件的最小边连成最小生成树）；同时增强了用该算法编写代码的能力。

#include <iostream>

#define MAXWEIGHT INT\_MAX

using namespace std;

//获得正确的结点数和边数（有代码的健壮性检验）

void Get\_node\_edge\_numbers(int& Vertices\_num, int& Edges\_num)

{

while (1)

{

cin >> Vertices\_num >> Edges\_num;

//输入错误处理：（数据类型输入错误、节点数小于2、边数小于1、边数大于节点数\*（节点数-1））

if (cin.good() == 0 || Vertices\_num < 2 || Edges\_num < 1 || Edges\_num > Vertices\_num \* (Vertices\_num - 1) / 2)

{

cin.clear();

cin.ignore(1024, '\n');

cout << "输入错误！请重新输入！" << endl;

continue;

}

break;

}

}

//边结点类

class Edgenode

{

public:

char start; //边的起始顶点

char end; //边的终止顶点

int weight; //边的权重

Edgenode() { start = end = ' '; weight = 0; }

};

//图类

class Graph

{

public:

int Vertices\_num = 0; //图的顶点数

int Edges\_num = 0; //图的边数

int\* Vertices = NULL; //顶点数组

Edgenode\* Edges = NULL; //边结点数组

int\*\* Matrix = NULL; //邻接矩阵数组

Graph(); //构造函数

~Graph() {}; //默认析构函数

void Create\_tree\_by\_Kruskal(); //Kruskal算法求最小生成树

int FindUFSets(int\* UFSets, int node); //查找节点所在的集合

void Union(int\* UFSets, int node1, int node2); //合并两个集合

};

//构造函数：建立关系矩阵

Graph::Graph()

{

//输入顶点数目和边数

cout << "请输入所求图的顶点数目和边的数目(以空格分隔各个数，输入两个0结束)：";

Get\_node\_edge\_numbers(Vertices\_num, Edges\_num);

//初始化顶点数组

Vertices = new int[Vertices\_num];

for (int i = 0; i < Vertices\_num; i++)

Vertices[i] = i;

//初始化边数组

Edges = new Edgenode [Edges\_num];

//初始化邻接矩阵（对角线的权值为0，其他都为最大权值）

Matrix = new int\*[Vertices\_num];

for (int i = 0; i < Vertices\_num; i++)

{

Matrix[i] = new int[Vertices\_num];

for (int j = 0; j < Vertices\_num; j++)

{

if (i == j)

Matrix[i][j] = 0;

else

Matrix[i][j] = MAXWEIGHT;

}

}

//输入边的起点、终点和权重

cout << "请输入两条边的节点序号以及它们的权值(以空格分隔各个数)：" << endl;

for (int i = 0; i < Edges\_num; i++)

{

int Startnode, Endnode, Edgeweight;

cin >> Startnode >> Endnode >> Edgeweight;

cin.ignore(1024, '\n');

//输入的代码健壮性检验

if (cin.good() == 0 || Startnode < 1 || Startnode > Vertices\_num || Endnode < 1 || Endnode>Vertices\_num)

{

cin.clear();

cin.ignore(1024, '\n');

cout << "输入错误！请重新输入！" << endl;

i--;

continue;

}

Edges[i].start = Startnode;

Edges[i].end = Endnode;

Edges[i].weight = Edgeweight;

//这里要注意矩阵位置和序号的关系，矩阵位置等于节点序号-1

Matrix[Startnode - 1][Endnode - 1] = Matrix[Endnode - 1][Startnode - 1] = Edgeweight;

}

}

//查找某个节点属于的集合

int Graph::FindUFSets(int\* UFSets, int node)

{

//循环向上查找父节点（祖先）

while (UFSets[node] != node)

node = UFSets[node];

return node;

}

//合并两个结点所在的集合

void Graph::Union(int\* UFSets, int node1, int node2)

{

int Parent1 = FindUFSets(UFSets, node1);

int Parent2 = FindUFSets(UFSets, node2);

//node1的祖先指向node2的祖先，从而完成两个集合的合并

UFSets[Parent1] = Parent2;

}

//按照Kruskal求出最小生成树并打印

void Graph::Create\_tree\_by\_Kruskal()

{

cout << "------Kruskal算法求最小生成树------" << endl;

//将边的权重按小到大排序

for (int i = 0; i < Edges\_num; i++)

{

for (int j = i + 1; j < Edges\_num; j++)

{

if (Edges[i].weight > Edges[j].weight)

{

int temp\_start = Edges[i].start;

Edges[i].start = Edges[j].start;

Edges[j].start = temp\_start;

int temp\_end = Edges[i].end;

Edges[i].end = Edges[j].end;

Edges[j].end = temp\_end;

int temp\_weight = Edges[i].weight;

Edges[i].weight = Edges[j].weight;

Edges[j].weight = temp\_weight;

}

}

}

//初始化并查集

int\* UFSets = new int[Vertices\_num];

for (int i = 0; i < Vertices\_num; i++)

UFSets[i] = i;

int Mincost = 0;

for (int i = 0; i < Edges\_num; i++)

{

int Startnode = Edges[i].start - 1;

int Endnode = Edges[i].end - 1;

int Edgeweight = Edges[i].weight;

//如果两个节点不属于同一个集合

if (FindUFSets(UFSets, Startnode) != FindUFSets(UFSets, Endnode))

{

//将其规划入最小生成树内，同时加上该路径的耗费

Union(UFSets, Startnode, Endnode);

Mincost += Edges[i].weight;

cout << "\t" << "最小耗费是：" << Startnode + 1 << "和" << Endnode + 1 << endl;

}

}

cout << "\t" << "最小总耗费为：" << Mincost << endl;

//动态释放空间，以免造成内存的浪费

delete[] UFSets;

}

//主函数的实现

int main()

{

Graph City\_road; //实例化Graph City\_road

City\_road.Create\_tree\_by\_Kruskal();

return 0;

}