#pragma once

#include "CNode.h"

class CLink

{

public:

int ID; // 路段的编号，从零开始编号

CNode\* pInNode; // 路段的起节点

CNode\* pOutNode; // 路段的终节点

double FreeFlowTravelTime; // 自由流走行时间

double TravelTime = 0; // 走行时间

double Capacity; // 路段通行能力

double Alpha = 0.15; // BPR函数参数，一般取0.15

double Power = 4.0; // BPR函数参数，一般取4.0

};

#pragma once

#include"CNode.h"

#include"CLink.h"

#include"COrigin.h"

#include<vector>

#include<string>

#include <fstream>

using namespace std;

class CNetwork

{

public:

vector<CNode\*> m\_Node; //网络节点集合

vector<CLink\*> m\_Link; //网络路段集合

vector<COrigin\*> m\_Origin; //网络的起点集合

int m\_nNode; //节点数

int m\_nLink; //路段数量

int m\_nOrigin; //起点数量

double\* LinkFlow; //路段流量 free

vector<double> LinkTravelTime; //路段走行时间

double MaxUEGap = 1.0e-5; //UE的最大误差

double UEGap; //UE误差

double\* ShortestPathCost;//临时变量，所有节点到起点的最短路 free

int\* ShortestPathParent;//最短路上，所有节点到起点的在最短路上的前继路段 free

public:

void ReadNode(string DataPath); // 函数：读取节点文件

void ReadLink(string DataPath); // 函数：读取路段文件

void ReadODpairs(string DataPath); // 函数：读取OD对文件

vector<int> GetShortestPath(int Start, int End); // 计算最短路：返回Start到End最短路径上的 \*路段集合\*

double GetShortestCost(int Start, int End); // 计算最短距离：返回Start到End最短路径上的 \*最短距离\*

double\* AllorNothingAssignment(double\* LinkFlow); // 全有全无交通分配：输入为初始路段流量（索引对应CLink中的ID）

void FrankWolfeAlgorithm(); // Frank-Wolfe算法

double GetUEGap( double\* LinkFlow); // 间隙函数

double OptimalStepFunction(double\* FeasibleDescent, double\* LinkFlow, double paraLamuda); // 计算最优步长的求根函数的左边

private:

inline bool exists(const std::string name)

{

ifstream f(name.c\_str());

return f.good();

}

vector<string> split(const string& str, const string& delim)

{

vector<string> res;

if ("" == str) return res;

//先将要切割的字符串从string类型转换为char\*类型

char\* strs = new char[str.length() + 1]; //不要忘了

strcpy(strs, str.c\_str());

char\* d = new char[delim.length() + 1];

strcpy(d, delim.c\_str());

char\* p = strtok(strs, d);

while (p)

{

string s = p; //分割得到的字符串转换为string类型

res.push\_back(s); //存入结果数组

p = strtok(NULL, d);

}

return res;

}

};

#pragma once

#include<vector>

using namespace std;

class CNode

{

public:

int ID; // 节点的编号，从零开始编号

double PositionX; // 节点的X坐标

double PositionY; // 节点的Y坐标

int Origin\_ID = -1; // 节点对应的起点编号，-1表示不是起点

vector<int> IncomingLink; // 进入节点的路段编号集合

vector<int> OutgoingLink ; // 离开节点的路段编号集合

};

#pragma once

#include "CNode.h"

#include<vector>

using namespace std;

class COrigin

{

public:

int ID; // 起点的编号，从零开始编号

CNode\* pOriginNode; // 起点对应的节点

vector<int> DestinationNode; // OD对，只记录有需求的OD

vector<double> ODDemand; // OD需求，只记录有需求的OD

};

#include "CNetwork.h"

#include<iostream>

#include <fstream>

#include<math.h>

using namespace std;

void CNetwork::ReadNode(string DataPath)

{

ifstream in(DataPath); //ifstream 读文件 in() 读文件

string row;

vector<string> Data;

m\_Node.clear(); //node集合, vector<CNode\*> m\_Node

m\_nNode = 0;

if (in)

{

while (getline(in, row))

{

if (row.empty()) continue;

Data = split(row, "\t");

CNode \*pNode = new CNode;

pNode->ID = m\_nNode;

pNode->PositionX = atof(Data[1].c\_str());

pNode->PositionY = atof(Data[2].c\_str());

m\_nNode++;

m\_Node.push\_back(pNode);

}

in.close();

}

else

{

cout << DataPath << " does not exist!";

}

}

void CNetwork::ReadLink(string DataPath) {

ifstream in(DataPath);

string row;

vector<string> Data;

m\_Link.clear();

m\_nLink = 0;

if (in)

{

while (getline(in, row))

{

if (row.empty()) continue;

Data = split(row, "\t");

CLink\* pLink = new CLink();

pLink->ID = m\_nLink;

auto inNodeIndex = atoi(Data[0].c\_str()) - 1;

pLink->pInNode = m\_Node[inNodeIndex];

auto outNodeIndex = atoi(Data[1].c\_str()) - 1;

pLink->pOutNode = m\_Node[outNodeIndex];

pLink->FreeFlowTravelTime = atof(Data[2].c\_str());

pLink->Capacity = atof(Data[3].c\_str());

pLink->pInNode->OutgoingLink.push\_back(pLink->ID);

pLink->pOutNode->IncomingLink.push\_back(pLink->ID);

m\_nLink++;

m\_Link.push\_back(pLink);

}

in.close();

}

else

{

cout << DataPath << " does not exist!";

}

}

void CNetwork::ReadODpairs(string DataPath) {

ifstream in(DataPath);

string row;

vector<string> Data;

m\_Origin.clear();

m\_nOrigin = 0;

COrigin\* pOrigin;

CNode\* pNode;

if (in)

{

while (getline(in, row))

{

if (row.empty()) continue;

Data = split(row, "\t");

pNode = m\_Node[atoi(Data[0].c\_str()) - 1];

if (pNode->Origin\_ID == -1)

{

pOrigin = new COrigin();

m\_nOrigin++;

pOrigin->ID = m\_Origin.size();

pOrigin->pOriginNode = m\_Node[atoi(Data[0].c\_str()) - 1];

pNode->Origin\_ID = pOrigin->ID;

m\_Origin.push\_back(pOrigin);

}

else

pOrigin = m\_Origin[m\_Node[atoi(Data[0].c\_str()) - 1]->Origin\_ID];

pNode = m\_Node[atoi(Data[1].c\_str()) - 1];

pOrigin->DestinationNode.push\_back(pNode->ID);

pOrigin->ODDemand.push\_back(atof(Data[2].c\_str()));

}

in.close();

}

else

{

cout << DataPath << " does not exist!";

}

}

vector<int> CNetwork::GetShortestPath(int Start, int End)

{

CNode\* pNode;

CLink\* pLink;

// 初始化

int startposition = 0, endposition = 1;

ShortestPathCost = new double[m\_nNode]; //当前最短路

ShortestPathParent = new int[m\_nNode]; //前驱节点

auto checkList = new int[m\_nNode];//队列，循环使用

auto binCheckList = new bool[m\_nNode]; //是否在队列中

auto bscanStatus = new bool[m\_nNode];

for (int node = 0; node < m\_nNode; node++)

{

ShortestPathParent[node] = -1;

ShortestPathCost[node] = numeric\_limits<double>::max();

binCheckList[node] = false;

}

ShortestPathCost[Start] = 0;

checkList[0] = Start;

//开始while循环

while (startposition != endposition)

{

if (startposition >= m\_nNode) startposition = 0;

int i = checkList[startposition];

startposition++;

pNode = m\_Node[i];

for (int index = 0; index < pNode->OutgoingLink.size(); index++)

{

pLink = m\_Link[pNode->OutgoingLink[index]];

int j = pLink->pOutNode->ID;

double value = pLink->TravelTime;

if (ShortestPathCost[j] > ShortestPathCost[i] + value)

{

ShortestPathCost[j] = ShortestPathCost[i] + value;

ShortestPathParent[j] = i;

// 添加到队列尾部

if (endposition >= m\_nNode) endposition = 0;

checkList[endposition] = j;

endposition++;

bscanStatus[j] = true;

}

}

}

// 返回最短路径集合

vector<int> ShortestPath\_Link;

int point\_out = End;

for (; ; )

{

int i = 0;

int point\_in = ShortestPathParent[point\_out];

for (int link = 0; link < m\_nLink; link++)

{

pLink = m\_Link[link];

if ((point\_in == pLink->pInNode->ID) && (point\_out == pLink->pOutNode->ID))

{

ShortestPath\_Link.insert(ShortestPath\_Link.begin(), 1, pLink->ID);

point\_out = point\_in;

}

}

i++;

if (point\_in == Start) break;

}

//delete[] checkList;

//delete[] binCheckList;

//delete[] bscanStatus;

return ShortestPath\_Link;

}

double CNetwork::GetShortestCost(int Start, int End)

{

CNode\* pNode;

CLink\* pLink;

// 初始化

int startposition = 0, endposition = 1;

ShortestPathCost = new double[m\_nNode]; //当前最短路

ShortestPathParent = new int[m\_nNode]; //前驱节点

auto checkList = new int[m\_nNode];//队列，循环使用

auto binCheckList = new bool[m\_nNode]; //是否在队列中

auto bscanStatus = new bool[m\_nNode];

for (int node = 0; node < m\_nNode; node++)

{

ShortestPathParent[node] = -1;

ShortestPathCost[node] = numeric\_limits<double>::max();

binCheckList[node] = false;

}

ShortestPathCost[Start] = 0;

checkList[0] = Start;

//开始while循环

while (startposition != endposition)

{

if (startposition >= m\_nNode) startposition = 0;

int i = checkList[startposition];

startposition++;

pNode = m\_Node[i];

for (int index = 0; index < pNode->OutgoingLink.size(); index++)

{

pLink = m\_Link[pNode->OutgoingLink[index]];

int j = pLink->pOutNode->ID;

double value = pLink->TravelTime;

if (ShortestPathCost[j] > ShortestPathCost[i] + value)

{

ShortestPathCost[j] = ShortestPathCost[i] + value;

ShortestPathParent[j] = i;

// 添加到队列尾部

if (endposition >= m\_nNode) endposition = 0;

checkList[endposition] = j;

endposition++;

bscanStatus[j] = true;

}

}

}

// 返回最短路距离

return ShortestPathCost[End];

}

double\* CNetwork::AllorNothingAssignment(double\* LinkFlow) {

auto ANLinkFlow = new double[m\_Link.size()]; // 全有全无路段流量

// CLink pLink;

// COrigin pOrinin;

for (int link = 0; link < m\_nLink; link++)

{

// 计算路段走行时间

m\_Link[link]->TravelTime = m\_Link[link]->FreeFlowTravelTime \* (1 + m\_Link[link]->Alpha \* pow(LinkFlow[link] / m\_Link[link]->Capacity, m\_Link[link]->Power));

// 初始化全有全无路段流量

ANLinkFlow[link] = 0;

}

for (int origin = 0; origin < m\_nOrigin; origin++) // 遍历O的循环

{

int OriginNode = m\_Origin[origin]->pOriginNode->ID; // 任意一个O的编号

for (int i = 0; i < m\_Origin[origin]->DestinationNode.size(); i++) // 遍历D的循环

{

int DestinationNode = m\_Origin[origin]->DestinationNode[i]; // 对应D的编号

double Demand = m\_Origin[origin]->ODDemand[i];

// 计算最短路上的路径集合

vector<int> ShortestPath\_Link = GetShortestPath(OriginNode, DestinationNode);

for (auto it = ShortestPath\_Link.begin(); it != ShortestPath\_Link.end(); it++)

{

ANLinkFlow[\*it] += Demand;

}

}

}

// 输出全有全无结果

// foreach (int index in ANLinkFlow) Console.WriteLine(index);

return ANLinkFlow;

}

double CNetwork::GetUEGap(double\* LinkFlow) {

COrigin\* pOrigin;

CLink\* pLink;

// 计算被减项分母 + 更新路段走行时间（出行费用：TravelTime）

double num1 = 0;

for (int link = 0; link < m\_nLink; link++)

{

pLink = m\_Link[link];

// 更新TravelTime

pLink->TravelTime = pLink->FreeFlowTravelTime \* (1 + pLink->Alpha \* pow(LinkFlow[link] / pLink->Capacity, pLink->Power));

// 单一求和项的计算

num1 += (pLink->TravelTime \* LinkFlow[link]);

}

// 计算被减项分子

double num2 = 0;

for (int origin = 0; origin < m\_nOrigin; origin++) // 遍历O的循环

{

pOrigin = m\_Origin[origin];

int OriginNode = pOrigin->pOriginNode->ID; // 任意一个O的编号

for (int i = 0; i < pOrigin->DestinationNode.size(); i++) // 遍历D的循环

{

int DestinationNode = pOrigin->DestinationNode[i]; // 对应D的编号

// 单一求和项的计算

double Demand = pOrigin->ODDemand[i];

double ODCost = GetShortestCost(OriginNode, DestinationNode);

num2 += (Demand \* ODCost);

}

}

// 计算间隙函数

UEGap = 1 - num2 / num1;

return UEGap;

}

double CNetwork::OptimalStepFunction(double\* FeasibleDescent, double\* LinkFlow, double paraLamuda) {

CLink\* pLink;

double Value = 0;

for (int link = 0; link < m\_nLink; link++)

{

pLink = m\_Link[link];

double para = LinkFlow[link] + paraLamuda \* FeasibleDescent[link];

Value += FeasibleDescent[link] \* (pLink->FreeFlowTravelTime \* (1 + pLink->Alpha \* pow(para / pLink->Capacity, pLink->Power)));

}

return Value;

}

void CNetwork::FrankWolfeAlgorithm()

{

// 初始化：

int k = 0; // 迭代次数

LinkFlow = new double[m\_nLink];

for (int link = 0; link < m\_nLink; link++) LinkFlow[link] = 0; // 所有路段流量为0

LinkFlow = AllorNothingAssignment(LinkFlow); // 做全有全无分配，得到初始路段流量

UEGap = GetUEGap(LinkFlow);

cout << "第" << k << "次迭代" << endl;

cout << "Gap函数值：" << UEGap << endl;

cout << "LinkFlow：";

for (int i = 0; i < m\_nLink; ++i) {

cout << LinkFlow[i] << ",";

}

cout << endl;

cout << endl;

ofstream output;

output.open("Gap.txt", ios::out);

// 开始while循环

{

while (UEGap > MaxUEGap)

{

auto newLinkFlow = AllorNothingAssignment(LinkFlow);

// 计算可行下降方向

auto FeasibleDescentDirection = new double[m\_nLink];

for (int link = 0; link < m\_nLink; link++)

{

FeasibleDescentDirection[link] = newLinkFlow[link] - LinkFlow[link];

}

// 计算最优步长（二分法）

double paraLamuda; // 最优步长

double left = 0;

double right = 1;

double mid = 0;

double f\_left = OptimalStepFunction(FeasibleDescentDirection, LinkFlow, left); // left的求根函数值

double f\_right = OptimalStepFunction(FeasibleDescentDirection, LinkFlow, right); // right的求根函数值

double f\_mid;

if ((f\_left \* f\_right) > 0) // f\_left和f\_right同号，说明根不在[0,1]区间

{

if (abs(f\_left) > abs(f\_right)) paraLamuda = right;

else paraLamuda = left;

}

else // f\_left和f\_right异号，说明[0,1]区间存在方程的根

{

// 用二分法求根

while (right - left > MaxUEGap)

{

mid = (left + right) / 2;

f\_left = OptimalStepFunction(FeasibleDescentDirection, LinkFlow, left);

f\_right = OptimalStepFunction(FeasibleDescentDirection, LinkFlow, right);

f\_mid = OptimalStepFunction(FeasibleDescentDirection, LinkFlow, mid);

if ((f\_left \* f\_mid) > 0) left = mid;

else right = mid;

}

paraLamuda = (left + right) / 2;

}

// 更新路段流量

for (int link = 0; link < m\_nLink; link++)

{

LinkFlow[link] = LinkFlow[link] + paraLamuda \* FeasibleDescentDirection[link];

}

k += 1;

UEGap = GetUEGap(LinkFlow); // 计算Gap函数

// 将Gap函数值写入文档

output << UEGap << "\n";

// 输出结果

cout << "第" << k << "次迭代" << endl;

cout << "Gap函数值：" << UEGap << endl;

cout << "LinkFlow：";

for (int i = 0; i < m\_nLink; ++i) {

cout << LinkFlow[i] << ",";

}

cout << endl << endl;

}

}

cout << "UE平衡的路段流量：" << endl;

for (int i = 0; i < m\_nLink; ++i) {

cout << LinkFlow[i] << ",";

}

}

// FrankWolfe.cpp : This file contains the 'main' function. Program execution begins and ends there.

//

#include"CNetwork.h"

#include <iostream>

using namespace std;

int main()

{

CNetwork\* Network = new CNetwork();

Network->ReadNode("Node\_example.txt");

Network->ReadLink("Link\_example.txt");

Network->ReadODpairs("ODPairs\_example.txt");

// 运行FW算法

Network->FrankWolfeAlgorithm();

// 释放堆区

delete Network;

system("pause");

}