武汉大学实习报告

院系：遥感信息工程学院 专业：遥感科学与技术 实习时间：2022年三月到四月

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 实习名称 | 数据结构与算法集中实习 | | | | | 指导教师 | 蒋永华 |
| 姓名 | 周思瑞 | 年级 | 2021 | 学号 | 2021302131066 | 成绩 |  |
| **实习目的：**  （1）通过读取已给文件数据，运用合适的数据结构解决最短路径等实际问题  （2）通过图的创建，图的深度优先、广度优先遍历，寻找最短路径，输出最短路径等算法加深了对数据结构与算法原理层次的理解，实现对课堂中学到的知识的充分利用  （3）通过对读取csv文以及将最短路径的地图可视化的学习，提升学习能力，扩大知识面，适应数据结构应用中实践的问题；  **实习内容及要求：**  1）CSV格式数据文件的读写**（已完成）**  2）图的创建（邻接矩阵或邻接表）**（已完成）**  3）图的遍历（广度优先或深度优先）**（已完成）**  4）图的最短路径，并具体给出（A到B）的最短路径及其数值**（已完成）**  5）最短路径的地图可视化展示**（已完成）**  6） 算法的时间复杂度分析**（已完成）**  **1、每个人必须完成1）、2）、4）三 种算法；**  **2、3）、5）选一个**  **3、按照“数据结构与算法”课程要求，进行规范的数据结构、算法、以及ADT设计，并进行算法的时间复杂度分析和实际统计，算法、代码注释清晰易读**  **实习方法或技术路线：**  **算法原理**    这是我程序的主要架构，可以发现，完成了所有的任务（6个任务）并且做到了比较合适的程序逻辑。  基于这个程序逻辑，逐步实现相应的算法，下面仔细讲讲每一个任务点对应的算法相关原理。  **任务点一： CSV格式数据文件的读写**  CSV文件实际上就是文本文件，同时使用逗号来辅助分割。  因此读入文件的时候就只需要考虑csv文件的逗号分隔符，所以使用cin和string读取一行，读完了之后再分析这一句的单独结构。  这个算法结束之后，我们就获得了一个城市所有数据的动态数组和所有道路数据的动态数组，并为下一个算法做铺垫。  **任务点二：图的创建**  图的创建比较快速，但是这个是整个程序最为核心的部分，所有的后续操作都基于此运行。  因为已经获得了一个城市所有数据的动态数组和所有道路数据的动态数组，现在要做的就是将道路数据放在我动态分配的二维数组里面，这样就可以创建一个邻接矩阵了。  对每一条路线的起始点和终点，通过函数确定各自对应的矩阵下标，然后将路线信息存入邻接矩阵中，图便创建完成了。  **任务点三：图的遍历**  共有两个算法：  深度优先  从给定的起点出发，进行类似于树的先根遍历的过程。  当一条路径走不下去了再走相近节点的下一条路。对每个点要判断是否走过，否则会重复遍历。最后给出连通分量。  我使用了递归的方式进行深度遍历。当然，也可以利用栈对其进行操作，这样就可以避免大量递归对函数内存栈的大量消耗，进一步提高了函数的运算速度  广度优先  用队列来处理，共给定的起点出发，遍历一行的矩阵，如果距离不为最大值，就意味着这两个点之间存在道路，将读到的点放在队列里面。如此操作，可以将所有有关该点的所有点都存在里面，然后将队列的第一个点抛弃，表示此点已经被访问完毕，然后访问该队列的队头。  反复使用这一段循环，可以做到将每一个点都不重复的访问。  最后给出连通分量。  值得注意的是，用每一个地址开始检索，其对应的遍历结果是不一样的；还有其他因素，比如这些点甚至不能成为一张完整的图，而是两个及以上的图，所以会出现一些点无法访问到的情况。  **任务点四：图的最短路径，并给出具体数值**  最短路径使用经典的Dijstra算法，也即是一个不断更新周围点然后寻找最小距离的贪心操作。  考虑将图中的所有的点分为两个集合（起始点所在的集合及另一个集合），并以起始点为 中心扩展到其他所有的点的最短路径。每次从另一个集合中选取一个与起始点距离最近的 点加入起始点所在的集合中，并计算起点到这个点的最短路径。  这个算法主要原理有：  1）新加入点到 起始点的最短路径一定由起始点所在集合中的点构成；  2）假设A点到C点的最短路径经由 B点，那么这条路径上A点到C点的部分就是A到B的 最短路径。  **任务点五：最短路径的地图可视化显示**  地图可视化展示实际上就是按照一定的代码规则输出一个html文件。  有很多的方式来输出HTML文件。比如使用读入HTML然后在上面更改，或者是直接在函数内部写死HTML的所有格式。  这两种操作各有优劣：前者的可操作性很高，一旦百度提供了其他的接口，便可以给出一个样式文件然后进一步更改。当然缺点也很明显，代码量较大，与此同时我们这只是一个作业，没有用户的反复使用，所以对便捷性的要求并不高；后者的开发便捷性相当高，方便起见，选择后者是相对明智的。  **算法的模块化设计与实现**  本程序基于C++开发，所以算法的模块化设计较清晰，下面是整个程序的UML图    可以看出来，本程序的架构主要由几个类构成，我们先从最大的类，也就是左上角的Mutual类(交互类)上面开始讲起  **Mutual类（声明）**  下面是这个类的声明  //这个是交互类，所有的操作将会在这里完成  class Mutual  {  public:  Mutual();  };  可以看到这个类是一个完完全全的交互类，只有一个操作就是新建此类。但是它的构造函数内部却有很多东西。  **Mutual类（定义）**  #include "Mutual.h"  #include"City.h"  #include"Route.h"  #include"ShortestPath.h"  #include"Traverse.h"  #include"Timer.h"  #include<Windows.h>  Mutual::Mutual()  {  //读两个文件时间  Timer timer;  timer.Start();  //新建一个City类,会自动读取文件  City \_City;  //新建一个Route类，并自动读取文件  Route \_Route;  //新建一个GraphMatrix，自动就创建图  GraphMatrix \_GraphMatrix(\_City, \_Route);  //停止然后输出时间  timer.Stop();  SetConsoleTextAttribute(GetStdHandle(STD\_OUTPUT\_HANDLE), FOREGROUND\_INTENSITY | FOREGROUND\_BLUE);  cout << "读取两个文件并构建邻接矩阵的时间为 " << timer.ElapsedTime() << " ms" << endl;  SetConsoleTextAttribute(GetStdHandle(STD\_OUTPUT\_HANDLE), FOREGROUND\_INTENSITY | FOREGROUND\_RED | FOREGROUND\_GREEN | FOREGROUND\_BLUE);    //在图的遍历和最小路径的两个模式中选择  string mode;  cout << "请选择模式" << endl  << "--------------------------------------------" << endl  << "输入 “遍历” 进行图的遍历" << endl  << "输入 “最短” 进行两种最短路径的检索和输出" << endl  << "--------------------------------------------" << endl;  cin >> mode;  system("cls");  if (mode == "最短")  {  //新建一个shortestPath，函数内部会询问出发地和目的地，然后在选择条件（时间/距离），之后会在屏幕上打印路径  ShortestPath \_ShortestPath(\_GraphMatrix);  }  else if(mode=="遍历")  {  //新建一个Traverse类，在内部会询问遍历要求和遍历初始城市  Traverse \_Traverse(\_GraphMatrix);  }  }  这是全部操作逻辑，通过调用所有的相关类来进行面向对象设计里面的面向过程环节  **City类（声明）**  这个类是第一个有效的封装类，其作为一个整体，在交互类里第一个生成，通过其自身的构造函数完成这个类内部的所有相关操作。  下面是它的声明  //城市整体封装类  class City  {  public:  City();//初始化并读入文件  ~City();    private:  void ReadCities(void);  int getnumber(const char\*);//传入文件名计算城市大小    public:  int size;//城市数目  oneCity\* cities;//所有城市  };  可以发现City类是由一个oneCity动态数组和其大小指标组成的(相关关系可以看UML图)，下面我们来看看oneCity类的声明  **oneCity类（声明）**  //城市类（类型为点）  class oneCity  {  public:  oneCity();    public:  string m\_country; //国家  string m\_city; //城市    float m\_latitude; //纬度  float m\_longitude; //经度  };  发现它实际上就是一个存放城市信息的节点，将其组成一个数组之后就可以和它的长度指标size组成一个类，也就是City类  **oneCity类（定义）**  //平平无奇的单个元素初始化  oneCity::oneCity() :m\_city(""), m\_country(""), m\_latitude((double)0), m\_longitude((double)0) {}  **City类（定义）**  一个类的定义也就是这个类的实现核心，这里我使用了初始化构造函数来解决所有的问题，下面是相关的所有代码  这个是最核心的代码，即初始化时会自动向文件里读取内容  City::City() :cities(nullptr), size(0)  {  //读取文件  ReadCities();  }  这是一个普通的析构函数  City::~City()  {  delete[] cities;  }  这是操作时候的读取函数  void City::ReadCities(void)  {  if (freopen("./source/cities.csv", "rt", stdin) == nullptr)  {  cout << "无法打开该文件！"; //文件无法打开  exit(0);  }  else  {  //读文件里地点的数目,并new一个数组  size = getnumber("./source/cities.csv");  cities = new oneCity[size];      for (int i = 0; i < size; ++i) //fp没有读到文件结束并且城市没有读取完  {  //一行全部读取  char ctemp[100];  cin.getline(ctemp, 100, '**\n**');  string stemp(ctemp);    //行内容处理  //读取国家名  cities[i].m\_country = stemp.substr(0, (stemp.find\_first\_of(',')));  stemp.erase(0, stemp.find\_first\_of(',') + 1);    //读取城市名  cities[i].m\_city = stemp.substr(0, (stemp.find\_first\_of(',')));  stemp.erase(0, stemp.find\_first\_of(',') + 1);    //读取两个参数  cities[i].m\_latitude = stof(stemp.substr(0, (stemp.find\_first\_of(','))));  stemp.erase(0, stemp.find\_first\_of(',') + 1);    cities[i].m\_longitude = stof(stemp);  }  }  fclose(stdin); //文件关闭    //读取城市信息函数的空间复杂度  //cout << "ReadCities->Use of space:" << size << endl;  }  这个是读取文件之前要做的操作，也就是将文件的大小先读好，然后再分配合适的空间，以此来提高灵活性  int City::getnumber(const char\* filename)  {  FILE\* fp = nullptr;  fopen\_s(&fp, filename, "rt");    //计数  int count = 0;    //安全判断  if (fp == nullptr)  {  cout << "文件打开失败" << endl;  }  else  {  char buffer[1000];  for (; fgets(buffer, sizeof(buffer) / sizeof(buffer[0]), fp) != nullptr; ++count) {}  }    return count;  }  **Route类（声明）**  下面是该类的声明  class Route  {  public:  Route();  ~Route();  void ReadRoutes(void);  int getnumber(const char\*);//在数组里面找到下标，用于和矩阵交互    public:  oneRoute\* routes;  int size;  };  发现这个类和上面的City类没有很大的区别，有一个小类组成的动态数组和对应的长度指标，下面就是这个动态数组里面一个元oneRoute的声明。  **oneRoute类（声明）**  //路线类（类型为边）  class oneRoute  {  public:  oneRoute();    public:  string origin\_city; //路线起点  string destination\_city; //路线终点  string m\_transport; //交通工具  string other\_information; //其他信息    float m\_cost; //花费  float m\_time; //时间  };  和之前的oneCity一样，它也是一个节点类，只不过存放的是一段小路径。  **oneRoute类（定义）**  这就是一个和上面oneCity类一样的初始化  oneRoute::oneRoute() :m\_cost(0), m\_time(0), m\_transport(""), origin\_city(""), destination\_city(""), other\_information("") {}  **Route类（定义）**  和之前一样，使用初始化构造函数来做所有的操作  Route::Route() :routes(nullptr), size(0)  {  ReadRoutes();  }  析构函数来防止内存泄漏  Route::~Route()  {  delete[] routes;  }  寻找在矩阵里的下标  //输入城市名，返回矩阵下标（其实是自己的下标，但是也刚好是矩阵的）  int Route::getnumber(const char\* filename)  {  FILE\* fp = nullptr;  fopen\_s(&fp, filename, "rt");    //计数  int count = 0;    //安全判断  if (fp == nullptr)  {  cout << "文件打开失败" << endl;  }  else  {  char buffer[1000];  for (; fgets(buffer, sizeof(buffer) / sizeof(buffer[0]), fp) != nullptr; ++count) {}  }    return count;  }  将路径直接读入数组里面  窗体顶端   //就是字面意思  void Route::ReadRoutes(void)  {    if (freopen("./source/routes.csv", "rt", stdin) == nullptr)  {  cout << "无法打开该文件！" << endl;  exit(0);  }  else  {  //读文件里地点的数目,并new一个数组  this->size = getnumber("./source/routes.csv");  routes = new oneRoute[size];      for (int i = 0; i < size; ++i)  {  //一行全部读取  char ctemp[1000];  cin.getline(ctemp, sizeof(ctemp) / sizeof(ctemp[0]), '**\n**');  string stemp(ctemp);      //行内容处理  //第一个地区  routes[i].origin\_city = stemp.substr(0, (stemp.find\_first\_of(',')));  stemp.erase(0, stemp.find\_first\_of(',') + 1);    //第二个地区  routes[i].destination\_city = stemp.substr(0, (stemp.find\_first\_of(',')));  stemp.erase(0, stemp.find\_first\_of(',') + 1);    //运输方式  routes[i].m\_transport = stemp.substr(0, (stemp.find\_first\_of(',')));  stemp.erase(0, stemp.find\_first\_of(',') + 1);    //运输时间  routes[i].m\_time = stof(stemp.substr(0, (stemp.find\_first\_of(','))));  stemp.erase(0, stemp.find\_first\_of(',') + 1);    //运输费用  routes[i].m\_cost = stof(stemp.substr(0, (stemp.find\_first\_of(','))));  stemp.erase(0, stemp.find\_first\_of(',') + 1);      //其他信息  routes[i].other\_information = stemp;  }  }    fclose(stdin);  freopen("CON", "r", stdin);    //读取最短路径函数的空间复杂度  //cout << "ReadRoutes->Use of space:" << size << endl;  }  窗体底端  **GraphMatrix类（声明）**  class GraphMatrix  {  public:  GraphMatrix(City&, Route&);  ~GraphMatrix();    void CreateMatrixGraph();  int Locate(string, City&);//传入名字和城市大类,返回在矩阵中的位置    public:  oneGraph\*\* \_Graph;//二级指针  int size;  City& tempCity;  Route& tempRoute;  };  看得出来这个类的核心部分是一个二维指针，之后会对其进行二维数组的创建操作，所以下面来讲讲二维数组的基本元素：oneGraph  **oneGraph类（声明）**  class oneGraph  {  public:  oneGraph(); //构造函数  ~oneGraph(); //析构函数  public:  oneRoute\* routes; //路线(这就是一条路线的指针)  float m\_cost; //路线花费  float m\_time; //路线耗时  };  **oneGraph类（定义）**  一个初始化构造函数来将元素内部的东西初始化一边  oneGraph::oneGraph() :m\_cost(MAX), m\_time(MAX), routes(nullptr) {}  将分配的内存释放掉  oneGraph::~oneGraph()  {  if (routes)  {  delete routes;  }  }  **GraphMatrix类（定义）**  GraphMatrix::GraphMatrix(City& inputCity,Route& inputRoute) : tempCity(inputCity),tempRoute(inputRoute)  {  this->size = inputCity.size;  CreateMatrixGraph();  }  虽然是GraphMatrix的函数，但是实际调用的是City里面的内容，用来寻找对应城市的下标  int GraphMatrix::Locate(string city, City& \_city)  {  for (int i = 0; i < size; i++)  {  if (\_city.cities[i].m\_city == city)  {  return i; //找到了则返回找到该城市的的编号  }  }    //如果输入错误  cout << "-----------------" << endl  << "城市名字输入错误" << endl  << "-----------------" << endl;  exit(0);  }  这是二维数组开辟的核心函数  void GraphMatrix::CreateMatrixGraph()  {    //先开辟二维数组的纵列  \_Graph = new oneGraph \* [size];    for (int i = 0; i < this->size; ++i)  {  //新建二维数组，一边新建一边写入  \_Graph[i] = new oneGraph[size];    for (int j = 0; j < this->size; ++j)  {  if (i == j)  {  //初始化所有自身连接为0  \_Graph[i][j].m\_cost = \_Graph[i][j].m\_time = 0;  }  else  {  //初始化外界连接为最大值  \_Graph[i][j].m\_cost = MAX;  \_Graph[i][j].m\_time = MAX;  }  }  }    //矩阵写入(其实我感觉可以继续优化一下，把这个循环放在上面去，就可以实现时间复杂度的进一步降低)  for (int i = 0; i < this->tempRoute.size; i++)  {  int begin, end;  begin = Locate(tempRoute.routes[i].origin\_city, tempCity);  end = Locate(tempRoute.routes[i].destination\_city, tempCity); //将路线起始点和终点的序号定位  \_Graph[begin][end].m\_cost = tempRoute.routes[i].m\_cost; //将路程的花费和时间赋给图的cost和time变量  \_Graph[begin][end].m\_time = tempRoute.routes[i].m\_time;  \_Graph[begin][end].routes = &tempRoute.routes[i]; //将路程赋给路程指针储存  }      //图的邻接矩阵的空间复杂度  //int totalSize = tempRoute.size + this->size + tempCity.size;  //cout << "CreateMatrixGraph->Use of space:" << totalSize << endl;  }  将分配的内存析构掉可以让程序在长时间运行中效率不至于降低  窗体顶端  GraphMatrix::~GraphMatrix()  {  //只需要删掉数组的索引什么的就好  //因为我也写了其他地方的析构函数  delete[] \_Graph;  }  窗体底端  **ShortestPath类（声明）**  这个是本次实习的第一个下游操作类，用于寻找最短路径，下面是它的声明  class ShortestPath  {  public:  ShortestPath(GraphMatrix&);  ~ShortestPath();    private:  void Timecost(int begin, int end);  void Moneycost(int begin, int end);  void Dispath(int start, int end, int\* path\_ct, int& num);  void restrict(void);  void writeFile(int path\_ct[], int num);    private:  string city\_from;  string city\_to;  GraphMatrix& tempGraphMatrix;  int\* pre;//用数组记录之前的点  float\* time;//暂存的时间数组  float\* cost;//暂存的价格数组  };  **ShortestPath类（定义）**  这是该类的初始化构造函数，所有的操作都在这个函数里面调用或者询问  ShortestPath::ShortestPath(GraphMatrix& inputGM) :tempGraphMatrix(inputGM), time(nullptr), cost(nullptr), pre(nullptr)  {  //询问要去的两个地方  cout << "-------------------------------------" << endl  << "请依次输入两个城市" << " 始发地 和 终末地 " << endl  << "-------------------------------------" << endl;  cin >> city\_from >> city\_to;  system("cls");    //初始化两个数组  pre = new int[tempGraphMatrix.size];      //定位  int begin = tempGraphMatrix.Locate(city\_from, tempGraphMatrix.tempCity);  int end = tempGraphMatrix.Locate(city\_to, tempGraphMatrix.tempCity);    //询问关键词  string searchWay;  cout << "-----------------------------------------" << endl  << "选择优先方式 时间（time） 价格（money）" << endl  << "-----------------------------------------" << endl;  cin >> searchWay;  system("cls");    //开始计时  Timer timer;  timer.Start();    if (searchWay == "time" || searchWay == "时间")  {  //核心运算(时间)  time = new float[tempGraphMatrix.size];  Timecost(begin, end);  }  else if (searchWay == "money" || searchWay == "价格")  {  //核心运算(价格)  cost = new float[tempGraphMatrix.size];  Moneycost(begin, end);  }    //输出时间  timer.Stop();    SetConsoleTextAttribute(GetStdHandle(STD\_OUTPUT\_HANDLE), FOREGROUND\_INTENSITY | FOREGROUND\_BLUE);  cout << "此算法运行时间为" << timer.ElapsedTime() << " ms" << endl;  SetConsoleTextAttribute(GetStdHandle(STD\_OUTPUT\_HANDLE), FOREGROUND\_INTENSITY | FOREGROUND\_RED | FOREGROUND\_GREEN | FOREGROUND\_BLUE);    // 这两个是为百度地图做准备的  int\* path\_ct = new int[tempGraphMatrix.size];  int num = 0;    //将路径展示出来  Dispath(begin, end, path\_ct, num);    // 重新处理  restrict();    //写入百度地图  writeFile(path\_ct, num);  }  下面的是当以时间最短为指标的时候，要进行的函数操作  //最短时间查询  void ShortestPath::Timecost(int begin, int end)  {  int i, j, k;  float min;  float tem;    //选取数组  int\* get = new int[tempGraphMatrix.size];    for (i = 0; i < tempGraphMatrix.size; ++i)  {  get[i] = 0;  pre[i] = -1; //pre数组，记录上一个点  time[i] = tempGraphMatrix.\_Graph[begin][i].m\_time; //初始化和时间参数  }      get[begin] = 1;    //对起始点的初始化（起点到起点的时间肯定是 0 ）  time[begin] = 0;    //更新初始点位最近的点  for (i = 0; i < tempGraphMatrix.size; i++)  {  if (time[i] != 0 && time[i] != MAX) //如果该点不是当前点或无法到达点  {  pre[i] = begin; //前一个点，即路径的记录  }  }    //在所有已知的点里面寻找最小的点  for (i = 0; i < tempGraphMatrix.size - 1; ++i)  {  min = MAX;  for (j = 0; j < tempGraphMatrix.size; ++j)  {  if (get[j] == 0 && time[j] < min)  {  min = time[j];  k = j; //根据贪婪算法，遍历并且找到本回合中时间最短的路径  }  }    get[k] = 1; //取中该点  for (j = 0; j < tempGraphMatrix.size; j++) //对剩余的点进行更新操作  {  if (j != k && tempGraphMatrix.\_Graph[k][j].m\_time < 10000)  {  tem = (time[k] + tempGraphMatrix.\_Graph[k][j].m\_time); //该点的权值即到达时间改为原值加新值  if (get[j] == 0 && (tem < time[j]))  {  time[j] = tem;  pre[j] = k;  }  }  }  }    delete[]get;    //输出时间总值  cout << "总时间是 " << time[end] << endl;    //最短路径函数的空间复杂度  //int size = tempGraphMatrix.size + tempGraphMatrix.size + tempGraphMatrix.size + tempGraphMatrix.size \* tempGraphMatrix.size;  //cout << "ShortestPath->Use of space:" << size << endl;  }  下面的是当以距离最短为指标的时候，要进行的函数操作，和上面的函数照葫芦画瓢  //和上面一个照葫芦画瓢  void ShortestPath::Moneycost(int begin, int end)  {    int i, j, k;  float min;  float tem;    //标记有没有被访问过  int\* get = new int[tempGraphMatrix.size];    for (i = 0; i < tempGraphMatrix.size; ++i)  {  get[i] = 0;  pre[i] = -1; //pre数组，记录上一个点  cost[i] = tempGraphMatrix.\_Graph[begin][i].m\_cost; //初始化和时间参数  }      get[begin] = 1;    //对起始点的初始化（起点到起点的时间肯定是 0 ）  cost[begin] = 0;    //更新初始点位最近的点  for (i = 0; i < tempGraphMatrix.size; i++)  {  if (cost[i] != 0 && cost[i] != MAX) //如果该点不是当前点或无法到达点  {  pre[i] = begin; //前一个点，即路径的记录  }  }    //在所有已知的点里面寻找最小的点  for (i = 0; i < tempGraphMatrix.size - 1; ++i)  {  min = MAX;  for (j = 0; j < tempGraphMatrix.size; ++j)  {  if (get[j] == 0 && cost[j] < min)  {  min = cost[j];  k = j; //根据贪心算法，遍历并且找到本回合中时间最短的路径  }  }    get[k] = 1; //取中该点  for (j = 0; j < tempGraphMatrix.size; j++) //对剩余的点进行更新操作  {  if (j != k && tempGraphMatrix.\_Graph[k][j].m\_cost < MAX)  {  tem = (cost[k] + tempGraphMatrix.\_Graph[k][j].m\_cost); //该点的权值即到达时间改为原值加新值  if (get[j] == 0 && (tem < cost[j]))  {  cost[j] = tem;  pre[j] = k;  }  }  }  }    delete[] get;    //输出价格总值  cout << "总花费费用是 " << cost[end] << endl;    //最短路径函数的空间复杂度  //int size = tempGraphMatrix.size + tempGraphMatrix.size + tempGraphMatrix.size + tempGraphMatrix.size \* tempGraphMatrix.size;  //cout << "ShortestPath->Use of space:" << size << endl;  }  这是将最短路径的计算结果可视化输出的函数  void ShortestPath::Dispath(int start, int end, int\* path\_ct, int& num)  {  //显示路径函数  int i = 0, j = 0;    //结尾标志城市  int k = end;    //储存路径的数组  int\* pNew = new int[tempGraphMatrix.tempRoute.size];  int n = 1;    while (true)  {  if (pre[end] == start) { break; }    //循环将路径存入数组pNew  pNew[i] = pre[end];  end = pNew[i];  j++;  i++;  }    num = j + 2;    //初始化路径，让第一个成为起始城市  path\_ct[0] = start;    cout << "路线为：" << tempGraphMatrix.tempCity.cities[start].m\_city;    //输出路径，起点->循环输出路径->终点  for (i = j - 1; i >= 0; i--, n++)  {  cout << " " << tempGraphMatrix.tempCity.cities[pNew[i]].m\_city;  path\_ct[n] = pNew[i];  }    //在终点的前一个点结束并输出终点  path\_ct[num - 1] = k;  cout << " " << tempGraphMatrix.tempCity.cities[k].m\_city << "**\n**";  cout << "----------------------------------" << endl;    delete[]pNew;  }  下面这段代码用于让我们读入的信息进一步适配HTML的格式。  这么做的原因就在于有一些城市名字或者备注信息里面有“’”，也就是单引号，而HTML这种标记语言对单引号比较敏感，所以如果不在前面添加反斜杠的话会直接将单引号后面的所有东西无效化。  所以就要遍历所有的信息然后找到单引号，找到之后在其前面加上一个反斜杠即可  当然，我实际的代码排版和这上面的不一样，原因是一行过长的代码在添加的时候会让这个表格的右边界直接消失，所以为此特意改了排版  //将格式适配HTML的格式  void ShortestPath::restrict(void){  for (int i = 0; i < tempGraphMatrix.tempCity.size; ++i)  {  //一旦遇见单引号，就在前面放一个斜杠    //在名字里寻找  if (tempGraphMatrix.tempCity.cities[i].m\_country.find('**\'**') != -1)  {  tempGraphMatrix.tempCity.cities[i].m\_country.insert  (tempGraphMatrix.tempCity.cities[i].m\_country.find\_first\_of('**\'**'), "**\\**");  }    //在城市里寻找  if (tempGraphMatrix.tempCity.cities[i].m\_city.find('**\'**') != -1)  {  tempGraphMatrix.tempCity.cities[i].m\_city.insert  (tempGraphMatrix.tempCity.cities[i].m\_city.find\_first\_of('**\'**'), "**\\**");  }    //在备注里找  if (tempGraphMatrix.tempRoute.routes[i].other\_information.find('**\'**') != -1)  {  tempGraphMatrix.tempRoute.routes[i].other\_information.insert  (tempGraphMatrix.tempRoute.routes[i].other\_information.find\_first\_of('**\'**'), "**\\**");  }  }  }  下面这一段代码是百度地图的输出代码，因为一行实在是太长，重新排版也不现实，于是这里来就将其截一张图来表示大意    这是此类的析构函数，用于将申请的内存空间释放掉。  ShortestPath::~ShortestPath()  {  if (pre) { delete pre; }  if (time) { delete time; }  if (cost) { delete cost; }  }  下面是价格最小情况下北京到伦敦的可视化路径图    **Traverse类（声明）**  这是遍历类，对外仅仅提供了一个初始化构造函数的调用权限，外部只能显式新建此类，相关操作在内部展开  //这是一个遍历类，在函数内部询问之后，实现两种遍历，对外只提供初始化接口  class Traverse  {  public:  Traverse(GraphMatrix&);  ~Traverse();      private:  void BFSfunction(int i);//广度优先函数的遍历  void BFSTraverse(int pos);//广度优先遍历(BFS)的核心函数(遍历一行)(调用的时候就用这个)    void DFSfunction(int i);//广度优先函数递归  void DFSTraverse(void);//广度优先递归实现        private:  GraphMatrix& tempGraphMatrix;  int\* isTrav;//用来记录有没有被访问过  };  **Traverse类（定义）**  这是此类的构造函数，所有遍历的操作和询问都在此处进行逻辑判断  Traverse::Traverse(GraphMatrix& input) :isTrav(nullptr), tempGraphMatrix(input)  {  //初始化遍历数组  isTrav = new int[tempGraphMatrix.size];    //询问图的遍历方式  int flag = 0;  cout << "-----------------" << endl  << "深度遍历请输入 1 " << endl  << "广度遍历请输入 2" << endl  << "-----------------" << endl;  cin >> flag;  system("cls");    //询问开始的城市（作为节点使用）  string city;  cout << "-----------------" << endl  << "请输入遍历开始的城市" << endl  << "-----------------" << endl;  cin >> city;  system("cls");    //处理城市  int beginCity = tempGraphMatrix.Locate(city, tempGraphMatrix.tempCity);      //开始计算时间  Timer timer;  timer.Start();      //选择遍历方式  switch (flag)  {  case 1:  DFSTraverse();  break;    case 2:  BFSTraverse(beginCity);  break;  default:  cout << "-------------------------------" << endl  << "你输入了奇奇怪怪的字符，请检查一下" << endl  << "-------------------------------" << endl;  break;  }    timer.Stop();  SetConsoleTextAttribute(GetStdHandle(STD\_OUTPUT\_HANDLE), FOREGROUND\_INTENSITY | FOREGROUND\_BLUE);  cout << endl << "遍历时间是 " << timer.ElapsedTime() << " ms" << endl;  SetConsoleTextAttribute(GetStdHandle(STD\_OUTPUT\_HANDLE), FOREGROUND\_INTENSITY | FOREGROUND\_RED | FOREGROUND\_GREEN | FOREGROUND\_BLUE);  }  这是广度优先的核心函数，用队列解决问题  //广度优先遍历(BFS)的核心函数(遍历一行）  void Traverse::BFSfunction(int i)  {  int j;    //使用int型队列  queue<int>Q;    //遍历记录数组的初始化  isTrav[i] = 1;    cout << setw(30) << tempGraphMatrix.tempCity.cities[i].m\_city  << setw(20) << tempGraphMatrix.tempCity.cities[i].m\_latitude  << setw(20) << tempGraphMatrix.tempCity.cities[i].m\_longitude << endl;    //在队列末尾加上i  Q.push(i);    while (!Q.empty())  {  i = Q.front();    //删除掉队头元素  Q.pop();    for (j = 0; j < tempGraphMatrix.size; ++j)  {  //如果之间有联系并且没有被遍历过  if (tempGraphMatrix.\_Graph[i][j].m\_cost != MAX && !isTrav[j])  {  //直接输出  cout << setw(30) << tempGraphMatrix.tempCity.cities[j].m\_city  << setw(20) << tempGraphMatrix.tempCity.cities[j].m\_latitude  << setw(20) << tempGraphMatrix.tempCity.cities[j].m\_longitude << endl;    //可以看到，矩阵是可以压扁的，这个点访问过了就不用再访问了  isTrav[j] = 1;    //把这个点加入队列  Q.push(j);  }  }  }    //图的BFS遍历的空间复杂度  //int size = sizeof(isTrav) + sizeof(Q) + sizeof(MatrixGraph);  //cout << endl << "BFSfunction->Use of space:" << size << endl;  }  下面是广度优先函数的封装前置调用，就是封装好了之后再拿来进一步使用，提高代码的可读性  void Traverse::DFSTraverse()  {  //初始化，将所有顶点都设定为未访问过  for (int i = 0; i < tempGraphMatrix.tempCity.size; i++)  {  isTrav[i] = 0;  }    //连通分量计算  int count = 0;    for (int i = 0; i < tempGraphMatrix.tempCity.size; i++)  {  if (!isTrav[i])  {  //若未被访问则进行DFS遍历，防止未连通  DFSfunction(i);  ++count;  }  }    cout << "连通分量为 " << count << endl;  }  深度优先遍历的递归调用方式  //DFS深度优先遍历，用第二种种方式实现遍历  void Traverse::DFSfunction(int i)  {  //初始点设置为1（访问过）  isTrav[i] = 1;  cout << setw(30) << tempGraphMatrix.tempCity.cities[i].m\_city  << setw(20) << tempGraphMatrix.tempCity.cities[i].m\_latitude  << setw(20) << tempGraphMatrix.tempCity.cities[i].m\_longitude << endl;    //从第0个顶点开始判断到最后一个  for (int j = 0; j < tempGraphMatrix.tempCity.size; j++)  {  //如果城市i到城市j有路径，且j未访问，则进行递归  if (tempGraphMatrix.\_Graph[i][j].m\_cost != MAX && !isTrav[j])  {  DFSfunction(j);  }  }  }  深度优先的封装函数，直接拿来调用就可以了  //外面包着的函数  void Traverse::BFSTraverse(int pos)  {  int i;    //先将全部初始化为0，然后逐步遍历  for (i = 0; i < tempGraphMatrix.size; ++i)  {  isTrav[i] = 0;  }    //连通分量计算  int count = 0;    for (i = 0; i < tempGraphMatrix.size; ++i)  {  if (!isTrav[(i + pos) % tempGraphMatrix.size])  {  //递归算法完成路径的遍历  BFSfunction((i + pos) % tempGraphMatrix.size);    ++count;  }  }    cout << "连通分量为 " << count << endl;  }  一个普通的析构函数  //简简单单析构函数  Traverse::~Traverse()  {  delete[] isTrav;  }  至此，算法的模块化设计叙述结束，通过对这几个类的有序调用，可以在较好的逻辑下完成应有的功能  **算法复杂度分析与实测结果**  **要求1：读取csv文件**  分析：  **时间复杂度**：文件读取与数据元素个数正相关，时间复杂度为  **空间复杂度**：申请的个数也与元素个数相关，为  **要求2：图的创建**  分析：  **时间复杂度**：如果顶点数是 个，，那么时间复杂度为  **空间复杂度**：如果边数是 个， ，那么空间复杂度为  下面这张图是上面两个操作都完成之后的总花费时间    **要求3：图的遍历**  分析：  **时间复杂度**：  对矩阵的每一个部分都要遍历，所以时间复杂度为 。在广度优先的遍历中每个顶点都要进（出）一次列队且仅仅一下（类似于深度优先遍历），对于每一个顶点u出列队后,要访问的所有邻接点，时间为,因此我们可知广度优先遍历和深度优先遍历总的时间复杂度是一样的为或。  但是时间上为什么要比其他人慢这么多呢，这主要是控制台输出的缘故，要输出的东西越多，其时间就会越长  **空间复杂度**：我的算法需要一个长度为矩阵行数或列数的动态数组来存储其本身是否被遍历过，所以为  这是深度优先遍历所花的时间    这是广度优先遍历所花的时间    **要求4：最短路径算法**  分析：  **时间复杂度**：寻找到起点路径最短的点共需 ， 加入点时，更改最短路径共需 ，数组初始化 需要 次循环，从而算法的时间复杂度为 。更简单的，直接看我代码里面的最大循环级数，发现最多就两层循环，所以就是  **空间复杂度**：,那么空间复杂度为  这是以时间为指标最短路径的算法时间    **要求5：百度地图可视化**  分析：  时间复杂度：如果经过的城市是 个，那么  空间复杂度：可以看到，我的算法执行不需要申请新的临时空间，所以  这是输出成为百度地图所要花费的时间    **实习结论：**  这次实习既要求我们对已学习过的知识进行实践的一个整合与运用，还要求我们自行学习新知识，运用新知识，算是一个不太简单的挑战。  通过这次实习，通过与同学的探讨，通过不懈的去学习的经历，更在一次又一次bug的调试 中，也在最后结果出来的喜悦里，我看到了编程的另一面， 也对数据结构这门课有了更深的理解与应用，果然实践出真知。 | | | | | | | |
| 教师评语  指导教师\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ 2020年\_\_\_\_\_月\_\_\_\_\_日 | | | | | | | |