Pj2说明文档

1. 环境，语言
   1. Clion, c++14 ; 交互式python(read\_xlsx.ipynb)
2. excel文件处理，输出txt
   1. 考虑到用c++处理excel和很多的sheet比较复杂，考虑用python处理文件。clion虽然支持程序内使用python，但是实际操作不太友好。所以在外部编写了python程序，把Timetable.xlsx转换成需要的数据文件subway.txt。
   2. python代码说明
   3. 图形用户界面, 文本, 应用程序, 电子邮件

      描述已自动生成
   4. 读timetable文件，按照sheet一个一个讲每条线上的站点存入到node\_list，且考虑到10号线11号线的岔路情况，这两条线的存储单元要多一个元素。
   5. 文本

      描述已自动生成
   6. timetable文件里有一些格式不规范的值，比如'\xa0'多了空格, 去掉不规范的格式。
   7. 这一块的output是输出存入node\_list的站点的名字，和检查是否格式都规范
   8. 图片包含 文本

      描述已自动生成
   9. 处理时间，统一生成某个站点到这一条线上的首发站之间的时间差；且考虑啊到10号线11号线的特殊情况，把两列的时间整合在一起。
   10. 这一块的输出是，station，每一个元素的值有[地铁线，车站名，到首发站经过的时间]
   11. 图形用户界面, 文本, 应用程序

       描述已自动生成
   12. 写文件，GBK编码
3. 读取txt文件生成地图数据graph
   1. 对应到graph.h文件
   2. 存储数据的格式：
      1. 首先考虑到用pair。但后来发现仅仅两个数据不够，选择了c++11以上的tuple，存储的格式有tuple<string, uint, uint>，分别是一个字符串，两个无符号整数
      2. Node
         1. name: 站点名
         2. line： 在几号线上
         3. time2first；到这条线上首发站所经过的时间，是从subway.txt直接读取的
         4. Node \*prev：前一个node
         5. time2prev：距离前一站所经过的时间，从time2first减去前一个站的time2first的来
      3. lineNum：地铁线数
      4. station：所有地铁站的集合，保存的是地铁站名
      5. GRAPH：地图数据，是最关键的数据，是后面两个算法中数据来源。存的是*<站名，与相邻站的权重，与相邻站在哪条地铁线上>*
      6. list：存放地铁线的数据，有14行，对应14条线，里面存的是每个地铁站的node。在遍历subway.txt文件元素时存进去
   3. initGraph() 读txt文件存数据
      1. 文本

         描述已自动生成
      2. node存好后放里list里。
      3. 特殊情况
         1. 10号线分叉，要把分叉的后面一截的prev指到前面来
            1. ***if***(node[i].name=="龙柏新村") {  
                node[i].time2prev = node[i].time2first - node[i-5].time2first;  
                node[i].prev = &node[i - 5];  
               }
         2. 11号线分叉
            1. ***else if***(node[i].name =="上海赛车场") {  
                node[i].time2prev = node[i].time2first - node[i-4].time2first;  
                node[i].prev = &node[i - 4];  
               }
         3. 4号线环路，第一个的prev要指向最后一个文本

            描述已自动生成
         4. *浦电路的四号线和6号线是两个站*
         5. ***if***(node[i].line == 4)  
             station\_name = node[i].name = "浦电路4";  
            ***else if***(node[i].line == 6)  
             station\_name = node[i].name = "浦电路6";  
            cout<< node[i].name<<endl;
   4. generate\_graph() 生成地图数据，大小和序号与station存的一样
      1. 初始化
         1. 按照station顺序，自己到自己距离为0，其他的为无穷大***for***(***int*** i = 0; i < GRAPH.size(); i++) {  
             ***for***(***int*** j = 0; j < GRAPH[i].size(); j++) {  
             get<0>(GRAPH[i][j]) = station[j];  
             get<1>(GRAPH[i][j]) = (i == j) ? 0:**UINT\_MAX**; *//全部初始化为无符号的无穷大, 对角线上为0* get<2>(GRAPH[i][j]) = 0; *//初始化不在一条线上，则第三个变量为0* }  
            }
      2. 按照地铁线（list），把前后相邻两站的距离添加进去，同时考虑到环线的处理。仅有环线的第一个数据有prev，其他的第一个首发站都无prev  ***for***(***auto*** & line\_i : list) {  
          ***for***(***int*** line\_node = (***int***)line\_i.size()-1; line\_node >= 0; line\_node--) { *//line\_node>0是因为是首发站，没有prev  
          // 遍历station找站对应的序号* ***int*** prev\_order = 0, order = 0;  
          ***if***(line\_node ==0 && line\_i[0].line != 4)  
          ***continue***;  
          ***for***(***int*** station\_i = 0; station\_i < station.size(); station\_i++) {  
          ***if***(station[station\_i] == line\_i[line\_node].name)  
          order = station\_i;  
          ***else if***(station[station\_i] == line\_i[line\_node].prev->name)  
          prev\_order = station\_i;  
          }  
          get<1>(GRAPH[prev\_order][order]) = line\_i[line\_node].time2prev;  
          get<1>(GRAPH[order][prev\_order]) = line\_i[line\_node].time2prev;  
          get<2>(GRAPH[order][prev\_order]) = line\_i[line\_node].line;  
          get<2>(GRAPH[prev\_order][order]) = line\_i[line\_node].line;  
          }  
         }
   5. isExist，输入的首发站和终点站是否存在
4. Dijkstra算法实现 Dijkstra.cpp
   1. 数据
      1. Matrix链接矩阵，对应于graph里的GRAPH数据
      2. SVex，已经获得最短路的二维集合，包含数据有：自己的名字，到source距离，这条路径的上一个点，与上一个点在哪一条地铁线上
      3. UVex，未获得最短路的点，包含数据同上
      4. EVex，包含数据同上，就是上面数组里的元素
   2. 函数
      1. EVex findShortest(const UVex &uVex) 从未遍历的顶点中找下一个距离最短的，返回Evex
         1. uint minDistance = **UINT\_MAX**;  
            EVex shortest;  
            ***int*** count = 0;  
            ***for*** (***const auto*** &node: uVex) {  
             *//cout<<count++<<" "<<get<1>(node)<<endl;* ***if*** (get<1>(node) != **UINT\_MAX** && get<1>(node) <= minDistance) { *// 未遍历点中，tuple的第一个元素 小于最小距离* minDistance = get<1>(node);  
             shortest = node;  
             *//cout<<get<0>(node)<<endl;* }  
            }  
            ***return*** shortest;
      2. SVex **Dijkstra**(***const*** Matrix& G, ***int*** source\_number)
         1. 初始化
         2. 把GRAPH的数据传进来，到source的上一个顶点初始设为source。权重按照GRAPH直接赋值
            1. ***for***(***int*** i = 0; i < G[0].size(); i++) {  
                ***if*** (get<0>(G[0][i]) == get<0>(G[0][source\_number]))  
                sVex.emplace\_back(get<0>(G[0][i]), 0, get<0>(G[0][source\_number]),0); *//如果i就是起始点，距离source为0。上一个顶点也为source* ***else*** uVex.emplace\_back(get<0>(G[0][i]), get<1>(G[source\_number][i]), get<0>(G[0][source\_number]),get<2>(G[source\_number][i]));  
               }
         3. 遍历UVEX找距离最短的点加入到SVex中，同时同UVex删除和更新UVex
            1. ***while*** (!uVex.empty()) {  
                ***auto*** nextVex = findShortest(uVex); *//找到位遍历点中距离起始点距离最小的点  
                //cout<<get<0>(nextVex)<<" next"<<endl;* uVex.erase(find(uVex.begin(), uVex.end(), nextVex)); *//从U中删除* sVex.emplace\_back(nextVex);  
                ***int*** next\_order = find\_order\_in\_graph(G,get<0>(nextVex));  
                *// 更新U* ***int*** k = 0;  
                ***for*** (***auto*** &node: uVex) {  
                *// distance[source->nextVex] + distance[nextVex->node] <? distance[source->node]  
                // get<1>node = distance[source->node]* ***int*** node\_order = find\_order\_in\_graph(G,get<0>(node));  
                ***if*** (get<1>(G[next\_order][node\_order]) != **UINT\_MAX** &&  
                get<1>(nextVex) + get<1>(G[next\_order][node\_order]) < get<1>(node))  
                {  
                get<1>(node) = get<1>(nextVex) + get<1>(G[next\_order][node\_order]);  
                get<2>(node) = get<0>(nextVex); *//update the node's last node to nextVex* get<3>(node) = get<2>(G[next\_order][node\_order]);  
                *//cout<<get<0>(node)<<" -> "<<get<2>(node)<<get<1>(node)<<endl;* }  
                k++;  
                }  
               }
      3. **print**(***const*** SVex& sv, string destination)
         1. 按照存入的每个点的prev去倒着退出从起点到终点经过了哪些线，存入answer中，并打印
      4. ***void* graph\_dijkstra**(***const*** Graph& g,***const*** string & source\_name, ***const*** string & destination)
         1. 运行dijkstra算法
5. Bellman\_ford算法实现 头文件+cpp文件
   1. 数据结构
      1. Matrix，地图数据，与graph的GRAPH一致
      2. N，点数据，tuple有当点起点的序号（站名换成序号），前一点序号，权重，与前一点在哪条线上
      3. EDGE获得最短路径点集合
   2. 函数
      1. 构造函数：给vertex，所有站点个数赋值
      2. insert\_edge()，根据传进来的地图数据，整理成bellman算法的数据Edge
         1. ***for*** (***int*** g\_i = 0; g\_i < G.size(); g\_i++) {  
             ***for*** (***int*** g\_j = g\_i + 1; g\_j < G[g\_i].size(); g\_j++) { *// 只需要取对角线上半部分* ***if*** (get<1>(G[g\_i][g\_j]) != **UINT\_MAX**) {  
             *//相连  
             // (from, to, weight, line)* Edge.emplace\_back(g\_i, g\_j, get<1>(G[g\_i][g\_j]), get<2>(G[g\_i][g\_j]));  
             Edge.emplace\_back(g\_j, g\_i, get<1>(G[g\_i][g\_j]), get<2>(G[g\_i][g\_j]));  
             *// 无权图，from，to各要insert一次* }  
             }  
            }
      3. bellman\_for()算法实现
         1. 处理数据，存放进nn。nn里面是起点到每个点所经历的路径中，前一个点与当前点的数据，下标是自己，存的数据第一个string是自己的名字，第二个string是前面那个点名字，第三个int是它俩之间间隔时间，第四个int是这段路在哪条地铁线上
         2. 遍历点，检查是否有负环
            1. ***for*** (***int*** i = 0; i < vertex; i++) {  
                updated = ***false***;  
                ***for*** (***const auto*** &e :Edge) {  
                ***int*** me = get<0>(e), prev = get<1>(e), e\_weight = get<2>(e), e\_line = get<3>(e);  
                ***if*** (distance[prev] != **INT\_MAX** && me!= source\_order &&  
                (distance[prev] + e\_weight < distance[me])) { *// e will be counted, and e.to will be updated* distance[me] = distance[prev] + e\_weight;  
                updated = ***true***;  
                ***if***(me == find\_order(G,destination))  
                node\_list[me].emplace\_back(make\_tuple(station[me], station[prev], e\_weight, e\_line));  
                ***else*** node\_list[me].emplace\_back(make\_tuple(station[me], station[prev], e\_weight, e\_line));  
                nn[me] = make\_tuple(station[me], station[prev], e\_weight, e\_line);  
                }  
                }  
                ***if*** (!updated) { *//Until no more updates, out of the loop* ***break***;  
                }  
               }  
               ***if*** (updated) {  
                cout << "There is a negative cycle." << endl;  
               } ***else*** {  
                *// 有最短路径*
         3. 没有负环，按照nn中，找终点的下标，按照nn[..]里存的前一个点，去找nn前一个点的数据，再根据这个再找前一个点，知道最终前一个点是起点为止。最终把这条路存进answer。打印的时候倒叙输出即可
            1. ***int*** des\_order = find\_order(G, destination);  
               vector<tuple<string,string,***int***,***int***>> answer;  
               ***auto*** pointer = nn[des\_order];  
               answer.emplace\_back(pointer);  
               ***while***(get<1>(pointer) != get<0>(nn[source\_order])) {  
                ***auto*** condition = [pointer](tuple<string, string, ***int***, ***int***> x) { ***return*** get<0>(x) == get<1>(pointer); };  
                pointer = \*find\_if(nn.begin(), nn.end(), condition);  
                answer.emplace\_back(pointer);  
               }  
               ***while***(1) {  
                cout << "选择输出格式： 1： 全部输入； 2：不输出中间站点； 0：退出" << endl;  
                ***int*** cd;  
                cin >> cd;  
                ***if*** (cd == 1) {  
                cout<<source\_name<<" -> "<<destination<<" , 预计通行时间: "<<distance[des\_order]<<"分钟"<<endl;  
                print\_all(answer);  
                cout << destination << endl;  
                }  
                ***else if*** (cd == 2) {  
                cout<<source\_name<<" -> "<<destination<<" , 预计通行时间: "<<distance[des\_order]<<"分钟"<<endl;  
                print\_simple(answer);  
                cout << destination << endl;  
                }  
                ***else if***(cd == 0)  
                ***break***;  
                ***else*** cout << "重输"<<endl;  
               }
      4. graph\_bellman()， 运行bellman算法
6. 最终控制台输出
   1. 因为debug的编码不一样，在debug时要把txt的编码格式转换成utf8，源文件中包含中文的也都要将编码格式转换成utf8
   2. 控制台输出，所有文件编码格式为gbk

文本

描述已自动生成