

“数据结构”

课程设计报告

**设计题目**  模拟sensornet work的工作

**姓 名**

**学 号**

**专 业**  计算机科学与技术

**班 级**

**完成日期**  2024年7月3日

课程设计成绩评定表

|  |  |  |  |
| --- | --- | --- | --- |
| **设计题目** | 模拟sensornetwork的工作 | **成绩** |  |
| **内容简介** | Sensornetwork是一种新型的网络，其基本结构如下图所示：该网络由两部分组成，Sensornode集和DataCollector。Sensornode（可简称为Sensor）能够完成感知环境数据并将其发往DataCollector的功能。DataCollector完成Sensor采集数据的收集，它就是一台带有无线接收功能的计算机。  本课程设计以Qt为框架实现了对sensornetwork的工作模拟，包括传感器节点的随机生成，传感器的数据模拟产生，节点网络的构建与图形化展示，节点最短路径的查找等，采用迪杰斯特拉算法作为路由算法，基本完整模拟了sensornetwork的工作流程，拥有良好的交互界面，包括查找定位，数据修改，规则制定等。 | | |
| **评语** | **教师签名：** | | |

1. **需求和规格说明**

**（1）问题描述**

Sensornetwork是一种新型的网络，其基本结构如下图所示：该网络由两部分组成，Sensornode集和DataCollector。Sensornode（可简称为Sensor）能够完成感知环境数据并将其发往DataCollector的功能。DataCollector完成Sensor采集数据的收集，它就是一台带有无线接收功能的计算机。



图1 Sensornetwork图示

Sensornetwork可应用到很多实际领域中，如在战争中将Sensor散播在防线的前沿，可以收集敌人的一些情报（如大规模的部队转移等）。Sensor散播的地方称为Interestedarea，Sensor在这个区域内采集各自所在位置的数据，然后将采集到的数据传送到DataCollector。各个Sensor之间通过无线广播通讯，由于Sensor广播能力的限制，它只能和位于自身的一定广播半径内的Sensor进行通讯，所以有些Sensor就需通过其它Sensor，经过多次路由后才能到达DataCollector（如上图）。如何路由由Sensor内保存的简单路由表来决定。DataCollector的位置就在InterstedArea的边缘，且固定不动。

**（2）课程设计目的**

应用数据结构知识模拟一个新型网络系统。

**（3）基本要求**

①应用数据结构在单机上模拟Sensornetwork的工作。

②用VC++（C也可以）实现模拟系统，N个Sensor和1个DataCollector，其具体位置随机确定，InterestArea就是屏幕。N可配置，缺省为100。

③Sensor进行周期性采集，其采集周期可配置。

④Sensor的广播半径固定，也是可配置的参数。

⑤路由算法自行选择或设计。

**（4）实现提示**

Sensor集可组织成数组，它采集及收到的数据包用队列存储。具体细节也可参阅有关资料。

**（二） 设计**

**（1）框架设计**

由上述需求可知，我们需要设计的是一个模拟传感器的网络系统，这个网路系统的主题由图构成，因此应该使用图形界面将图显示，直观的描绘出传感器节点（SensorNode），边（Edge），数据收集器（DataCollector）之间的相互关系，可以用图形化的界面表达出算法的运行。故选用Qt这种跨平台应用程序开发框架作为主体框架，在Qt框架的基础上进行进一步的设计。

DataCollector（数据收集类）：

* 作为传感器网络的核心节点，负责收集和处理传感器节点的数据。
* 实现与传感器节点之间的通信和数据交换。
* 具备管理传感器节点的功能，如初始化、数据请求等。

Graph（图类）：

* 负责存储传感器网络的拓扑结构，包括节点和边。
* 支持基本的图操作，如添加节点、添加边、删除节点、删除边等。
* 实现图算法，如Dijkstra算法，用于计算从DataCollector到各个传感器节点的最短路径。

SensorNode（传感器节点）：

* 表示传感器网络中的单个传感器节点。
* 具备基本属性，如位置（x, y坐标）、ID、数据等。
* 支持与DataCollector和其他传感器节点的通信。

SensorNetwork（网络管理）：

* 负责管理整个传感器网络，包括传感器节点和数据收集器。
* 维护传感器网络的整体状态，并提供对外接口用于操作网络。
* 包含数据收集周期、广播半径等可配置参数。

Widget（图形界面）：

* 使用Qt进行界面设计，负责绘制传感器网络的图形表示。
* 显示传感器节点、数据收集器和边的图形元素。
* 提供用户交互功能，如添加节点、删除节点、执行算法并显示结果等。

**（2）界面设计**

主窗口 (QWidget: Widget) **尺寸**是宽1237，高842。设计有上方的图形绘制区，长宽分别为1190与530，下方是功能区，分别有最左边的节点收集信息展示部分，收集信息展示部分，配置操作部分三部分，为算法功能的实现提供基础



图2 窗口设计图示

**（3）类与属性说明**

|  |  |  |  |
| --- | --- | --- | --- |
| 类名/结构名 | 属性 | 类型/返回值 | 名称 |
| DataCollector | 变量 | double | x |
| double | y |
| QStandardItemModel \* | model |
| 方法 | void | addItem(TData data) |
| void | removeItem(TData data) |
| Edge | 变量 | int | to |
| double | weight |
| int | type |
| double | desister |
| 方法 | void | Set(int a) |
| Graph | 变量 | unordered\_map<int, std::vector<Edge>> | adjList |
| 方法 | void | addEdge(int from, int to, double weight) |
| Edge | findEdge(int from, int to) |
| Data | 变量 | bool | final |
| double | dist |
| int | path |
| 方法 | void | Set(bool f, double d, int p) |
| SensorNetwork | 变量 | QList<SensorNode\*> | sensors |
| DataCollector \* | dataCollector |
| int | sensorCount |
| double | broadcastRadius |
| int | T |
| Graph | graph |
| int | multiple |
| unordered\_map<int, Data> | PathMap |
| 方法 | QPair<double, QList<int>> | getShortestPath(int targetNode) |
| void | printPathMap() |
| void | initializeNetwork() |
| void | Dijkstra() |
| void | initEdgeType() |
| void | StartT() |
| void | GetNew() |
| void | GetNewD() |
| TData | 变量 | int | fromNode |
| int | data |
| QString | type |
| QDateTime | currentDateTime |
| 方法 | QString | toString() |
| SensorNode | 变量 | int | a |
| double | x |
| double | y |
| int | type |
| double | broadcastRadius |
| QQueue<TData> | dataQueue |
| QStandardItemModel \* | model |
| 方法 | void | receiveData(const TData &data) |
| void | collectData() |
| void | SetType(int a) |
| Widget | 变量 | SensorNetwork\* | sensorNetwork |
| QTimer\* | timer |
| int | sensorCount |
| double | broadcastRadius |
| int | T |
| int | T2 |
| bool | Judge |
| QTimer \* | timer1 |
| QTimer \* | timer2 |
| int | lastClickedSensorIndex |
| int | nowNode |
| 方法 | void | paintEvent(QPaintEvent \*event) |
| void | collect() |
| void | Send() |
| void | mousePressEvent(QMouseEvent \*event) |
| void | on\_pushButton\_clicked() |
| void | on\_pushButton\_2\_clicked() |

**（4）重点函数说明：**

4.1 最优路径规划—迪杰斯特拉算法实现

|  |
| --- |
| void SensorNetwork::Dijkstra() {  // 初始化路径表  for (int i = 0; i <= sensorCount; i++) {  PathMap[i].Set(false, std::numeric\_limits<double>::infinity(), -1); // 将距离设为无穷大  }  // 放入第一个节点  PathMap[sensorCount].Set(true, 0, -1);  // 初始化第一个节点的邻接节点的距离  for (const Edge& edge : graph.adjList.at(sensorCount)) {  PathMap[edge.to].Set(false, edge.weight, sensorCount);  }  for (int i = 0; i < sensorCount; i++) {  // 找到未处理节点中距离最短的节点  int minNode = -1;  double minDist = std::numeric\_limits<double>::infinity();  for (const auto& entry : PathMap) {  if (!entry.second.final && entry.second.dist < minDist) {  minDist = entry.second.dist;  minNode = entry.first;  }  }  //if (minNode == -1) {  // break; // 所有节点已处理完毕  //}  PathMap[minNode].final = true; // 将该节点标记为已处理  // 更新该节点邻接节点的距离  for (const Edge& edge : graph.adjList[minNode]) {  if (!PathMap[edge.to].final && PathMap[minNode].dist + edge.weight < PathMap[edge.to].dist) {  PathMap[edge.to].Set(false, PathMap[minNode].dist + edge.weight, minNode);  }  }  }  } |

 **初始化路径表**：

* 为每个节点创建一个记录，包含三个部分：是否已处理 (final)、距离 (dist)、和前驱节点 (prev)。
* 使用一个特殊的值（无穷大）初始化所有节点的距离，表示这些节点当前还不可达。
* 对数据收集器节点（假设为 sensorCount）初始化距离为0，前驱节点为-1，标记为已处理。

 **初始化第一个节点的邻接节点**：

* 遍历数据收集器节点的邻接节点，并初始化这些节点的距离。
* 将这些邻接节点的前驱节点设置为数据收集器节点。

 **主循环（处理所有节点）**：

* 找到未处理节点中距离最短的节点 minNode。这是Dijkstra算法的核心部分，确保每次选择的节点是当前已知最短路径的节点。
* 将该节点标记为已处理。
* 遍历该节点的所有邻接节点，更新这些邻接节点的距离（如果通过当前节点找到的路径更短）。
* 更新操作：如果通过 minNode 达到某个邻接节点的距离小于之前记录的距离，则更新该邻接节点的距离，并将其前驱节点更新为 minNode。

4.2 最短路径查找

|  |
| --- |
| QPair<double, QList<int>> SensorNetwork::getShortestPath(int targetNode) {  QList<int> path;  double totalDistance = PathMap[targetNode].dist;  // 检查目标节点是否可达  if (totalDistance == std::numeric\_limits<double>::infinity()) {  return qMakePair(totalDistance, path); // 返回空路径和无穷距离  }  // 使用堆栈逆序构建路径  QStack<int> stack;  int currentNode = targetNode;  while (currentNode != -1) {  stack.push(currentNode);  currentNode = PathMap[currentNode].path;  }  // 将堆栈中的路径顺序放入 QList 中  while (!stack.isEmpty()) {  path.append(stack.pop());  }  return qMakePair(totalDistance, path);  } |

**初始化**：

* 创建一个 QList<int> 来存储路径节点。
* 获取目标节点的总距离 totalDistance。

 **检查目标节点可达性**：

* 如果目标节点的总距离为无穷大，表示目标节点不可达。此时，返回总距离为无穷大和空路径。

 **逆序构建路径**：

* 使用 QStack<int> 堆栈存储路径节点，方便之后顺序输出路径。
* 从目标节点开始，逐步追踪其前驱节点 path，将节点压入堆栈，直到到达数据收集器节点（即 path 为 -1）。

 **构建有序路径**：

* 从堆栈中弹出所有节点，并按顺序放入 QList<int> 中，形成正确顺序的路径。

 **返回结果**：

* 返回一个 QPair<double, QList<int>>，包含总距离和节点路径。

**（三） 用户手册**

开始运行时会按照设定的默认值初始化传感器节点数字为100，广播半径为150，采集周期为5，发送间隔为2。可以在自己设置这些参数后点击配置按钮，及可按照输入的配置信息配置图像，在查找前的输入框中输入查找内容，可以查找到节点并在图形展示区将节点该节点变为红色并显示该节点到收集站的路径，将路径展示为红色，在节点信息栏显示节点的信息并实时更新，显示出当前节点上储存的信息与节点距离与信息多少。在最左边为信息总览，包含所有节点传输到收集站的信息，并可以修改。最左边的滑轮可以设置阻塞比例，调整距离与阻塞对边权重的影响。可以直接点击图中的节点，及定位到此节点。

**初始化设置**

启动程序时，系统会根据以下默认值进行初始化：

传感器节点数量：100

广播半径：150

采集周期：5

发送间隔：2

**参数配置**

用户可以在设置这些参数后点击“配置”按钮，系统将根据输入的配置信息重新配置图像。

**查找节点**

在查找前的输入框中输入节点信息，可以查找到对应节点，并在图形展示区将该节点标记为红色。系统还会显示该节点到数据收集站的路径，路径以红色展示。

**节点信息**

在节点信息栏中显示节点的详细信息并实时更新，包括：

当前节点上存储的信息

节点与数据收集站的距离

信息的数量

**信息总览**

界面最左侧显示信息总览，包含所有节点传输到数据收集站的信息。用户可以在此修改信息。

**阻塞比例调整**

界面左侧的滑轮可以设置阻塞比例，调整距离与阻塞对边权重的影响。

**节点定位**

用户可以直接点击图中的节点，系统将自动定位到该节点。

**（四） 调试及测试**

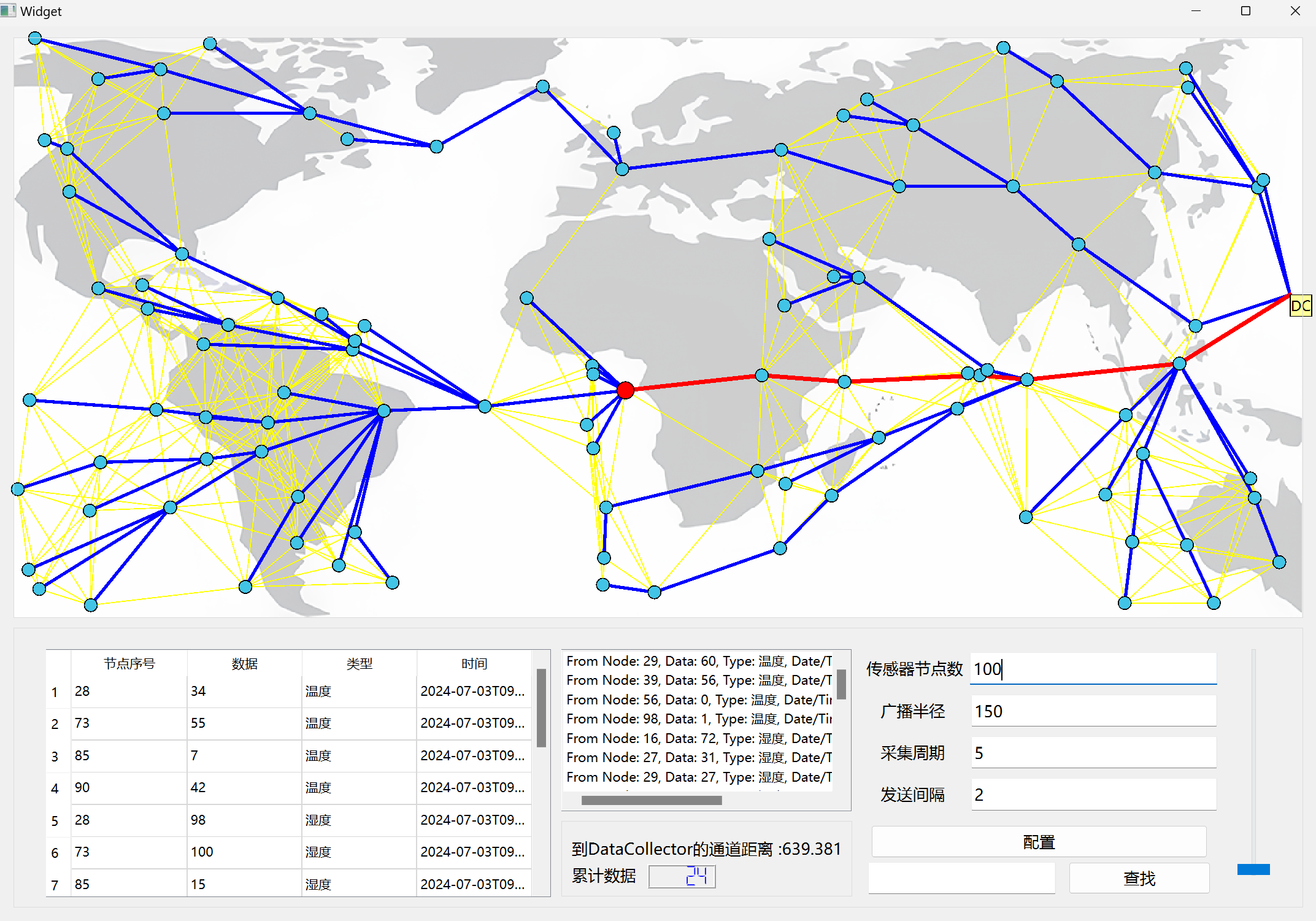
****

图3 运行结果

**（五） 运行实例：**

初始化后，将节点数改为150，采集周期改为10，发送间隔改为1

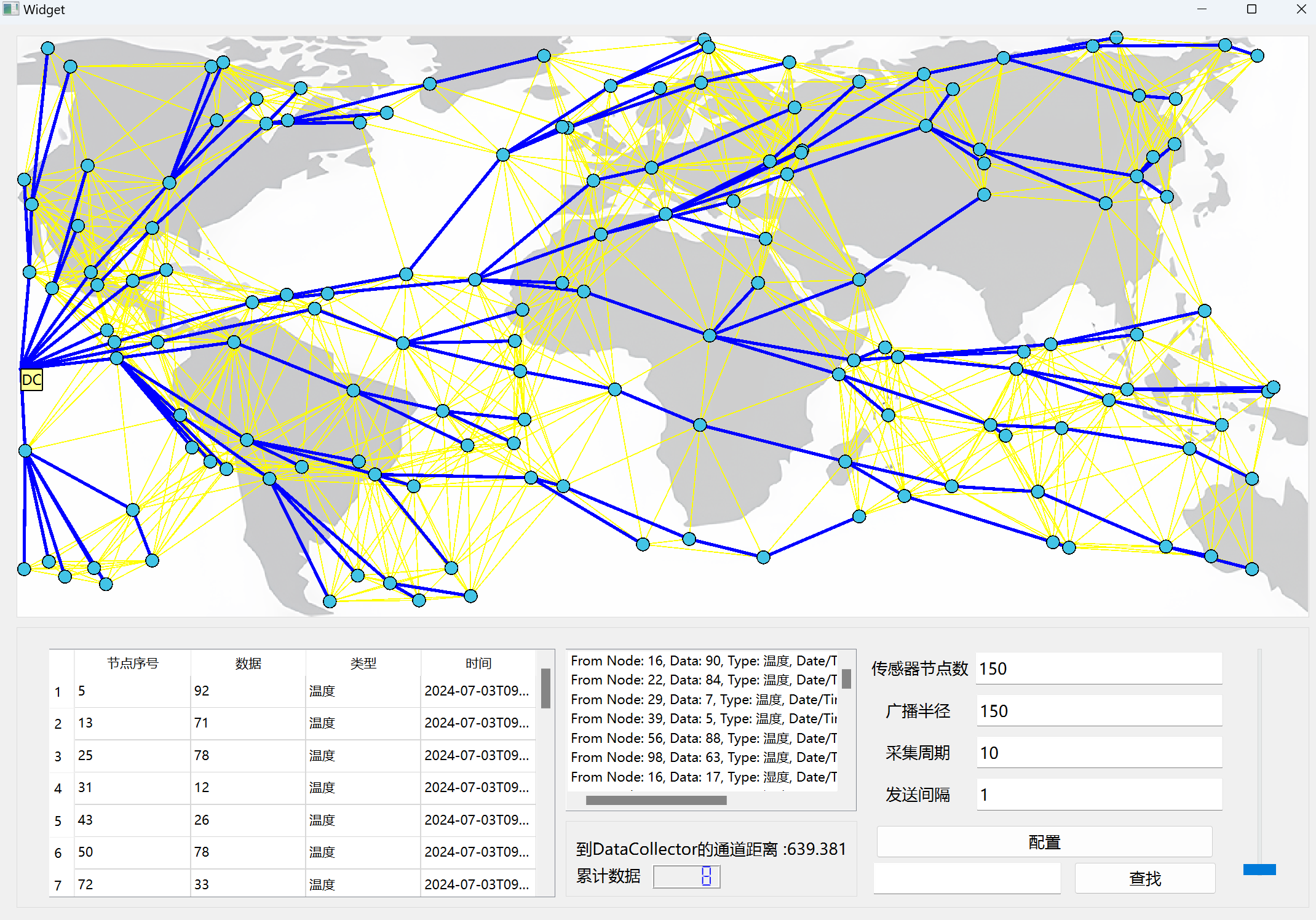


图4 手动配置

输入节点106，查找节点位置与最短路径

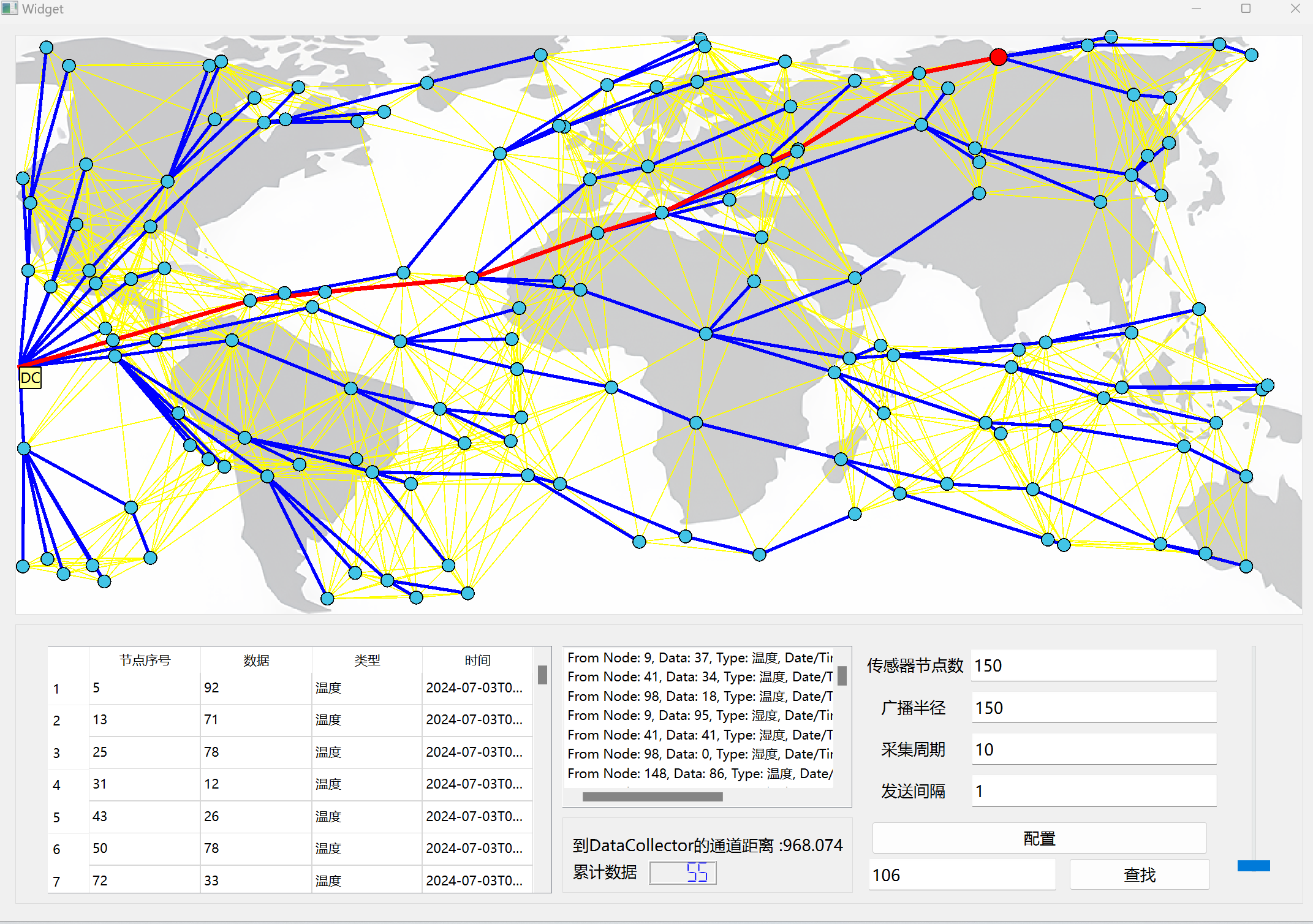


图5 查找节点

修改总数据中的条数据信息

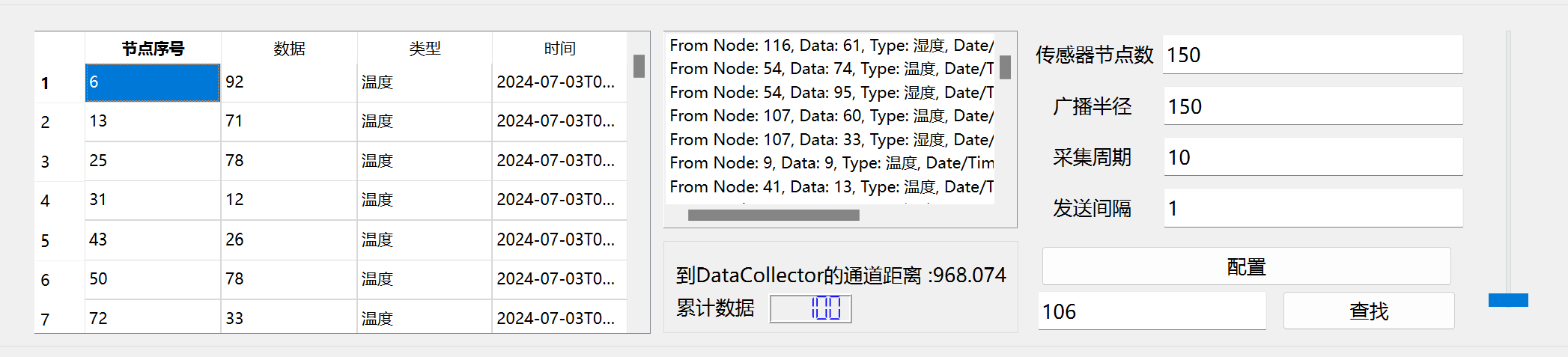


图7 数据修改

点击图中节点，自动定位节点与路径

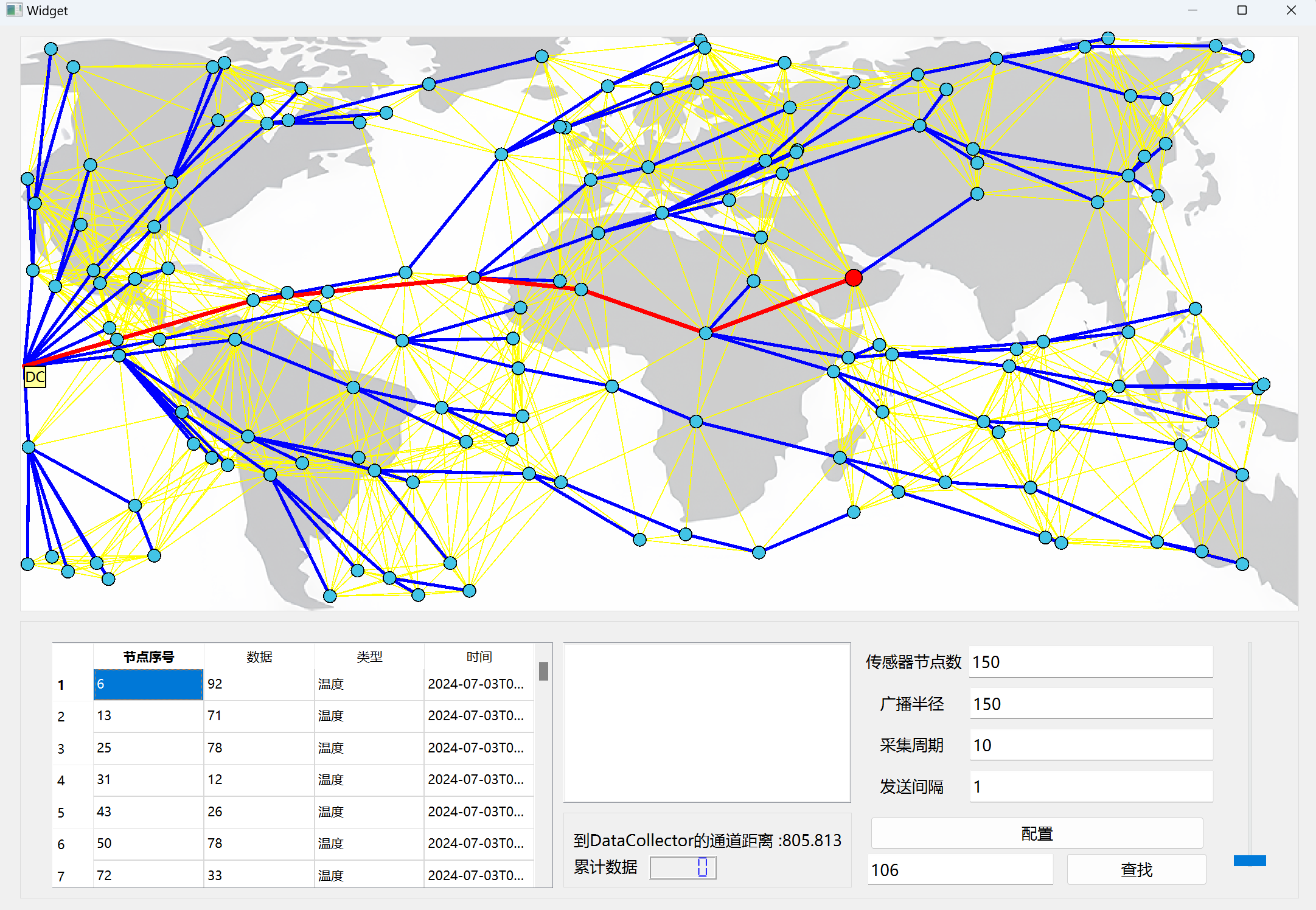


图8点击选择节点

**（六）进一步改进**

现在的实现有一个问题，当随机产生的节点有节点无法加入网络时，在遍历寻找时会报错导致程序奔溃，在后期优化要将不能加入网络的节点异常处理。

现在的界面有些单调，确实点美感，在后期需要对显示中的图点与边进行优化。

当前缺少直接在总数据中查找数据的功能，对收集到的数据没有处理，在后期可以加入对收集到的数据进行数据分析的功能。

**（七）心得体会**

在本次精心设计的课程项目中，我不仅深化了对图这一核心数据结构的认知，更在图的遍历与路径搜索的实践中锤炼了技能，使之从理论层面跃升至能够灵活应用于解决实际问题的层次。这一转变，让我深刻体会到计算机科学中“理论联系实际”的精髓所在。通过亲手构建图的模型，我学会了如何将错综复杂的现实世界问题抽象化，转化为计算机可处理的逻辑结构与算法，这一过程不仅锻炼了我的抽象思维能力，也强化了我的编程实践能力。

尤为值得一提的是，在将现实世界中的系统或网络映射为计算机中的类与对象时，我体会到了面向对象编程的强大与魅力。我学会了如何通过类的封装、继承与多态等特性，构建出既符合逻辑又易于维护的代码结构。这种从具体到抽象，再由抽象回归具体的过程，不仅加深了我对软件设计原则的理解，也为我未来在软件开发领域的探索奠定了坚实的基础。

面对项目过程中接踵而至的挑战，如图形界面的精细实现、算法效率的优化、界面交互的流畅性提升以及数据传输过程中的同步与异步处理等难题，我展现出了前所未有的韧性和坚持。每一个问题都是对我知识储备与解决问题能力的一次考验，而我则将这些挑战视为成长的契机，通过查阅资料、反复试验、向老师和同学请教等多种方式，逐一攻克难关。这一过程不仅提升了我的专业技能，更重要的是，它极大地增强了我的耐心、毅力和面对未知挑战时的勇气与决心。

**（八）附录⎯⎯源程序**

|  |
| --- |
| #ifndef DATACOLLECTOR\_H  #define DATACOLLECTOR\_H  #include <QStandardItemModel>  #include "sensornode.h"  // 数据收集的类  class DataCollector  {  public:  double x;  double y;  QStandardItemModel \*model;  public:  DataCollector();  void addItem(TData data);  void removeItem(TData data);  ~DataCollector();  };  #endif // DATACOLLECTOR\_H |

|  |
| --- |
| #include "datacollector.h"  DataCollector::DataCollector() : model(new QStandardItemModel()) {  // 设置表头  model->setHorizontalHeaderLabels({"节点序号", "数据", "类型","时间"});  }  // 添加数据项  void DataCollector::addItem(TData data) {  QList<QStandardItem \*> rowItems;  rowItems.append(new QStandardItem(QString::number(data.fromNode)));  rowItems.append(new QStandardItem(QString::number(data.data)));  rowItems.append(new QStandardItem(data.type));  rowItems.append(new QStandardItem(data.currentDateTime.toString(Qt::ISODate)));  model->appendRow(rowItems);  }  // 删除数据项  void DataCollector::removeItem(TData data) {  for (int row = 0; row < model->rowCount(); ++row) {  if (model->item(row, 0)->text().toInt() == data.fromNode &&  model->item(row, 1)->text().toInt() == data.data &&  model->item(row, 2)->text() == data.type) {  model->removeRow(row);  break;  }  }  }  // 析构函数中删除 model  DataCollector::~DataCollector() {  delete model;  } |

|  |
| --- |
| #ifndef SENSORNETWORK\_H  #define SENSORNETWORK\_H  #include <QList>  #include "SensorNode.h"  #include "DataCollector.h"  #include <QStack>  // 结构定义图中的边  struct Edge {  int to;  double weight;  int type;  double desister;  Edge() : to(-1), weight(0),type(1){}  Edge(int t, double w){  to = t;  weight = w;  type = 1;  desister = weight;  }  void Set(int a){  type = a;  }  };  // 图类  class Graph {  public:  std::unordered\_map<int, std::vector<Edge>> adjList;  // 添加边的方法  void addEdge(int from, int to, double weight);  // 查找边的方法  Edge findEdge(int from, int to);  };  // 结构定义迪杰斯特拉算法的储存  struct Data {  bool final; //是否找到最短路径  double dist; //最短路径长度  int path; //路径上的前驱  void Set(bool f, double d, int p){  final = f;  dist = d;  path = p;  }  };  // 管理整个传感器网络，包含传感器和数据收集器  class SensorNetwork  {  public:  QList<SensorNode\*> sensors; // 节点数组  DataCollector \*dataCollector; // 收集器  int sensorCount;  double broadcastRadius;  int T;  Graph graph;  int multiple;  std::unordered\_map<int, Data> PathMap; //路径地图  SensorNetwork(int sensorCount, double broadcastRadius, int T);  QPair<double, QList<int>> getShortestPath(int targetNode);  void printPathMap();  void initializeNetwork();  void Dijkstra();  void initEdgeType();  void StartT();  void GetNew();  void GetNewD(int A,int B);  };  #endif // SENSORNETWORK\_H |

|  |
| --- |
| #include "sensornetwork.h"  #include <QDebug>  #include <random>  #include <cmath>  // 计算欧几里得距离的函数  double euclideanDistance(double x1, double y1, double x2, double y2) {  return std::sqrt((x1 - x2) \* (x1 - x2) + (y1 - y2) \* (y1 - y2));  }  SensorNetwork::SensorNetwork(int sensorCount, double broadcastRadius, int T) {  this->sensorCount = sensorCount;  this->broadcastRadius = broadcastRadius;  this->T = T;  multiple = 1;  dataCollector = new DataCollector(); // 初始化DataCollector  }  void SensorNetwork::initializeNetwork()  {  int maxX = 1210;  int minX = 20;  int maxY = 540;  int minY = 10;  int d = 30;  std::random\_device rd; // 用于生成随机种子  std::mt19937 gen(rd()); // Mersenne Twister 随机数生成器  sensors.clear(); // 清空当前的传感器列表（如果有）  std::uniform\_int\_distribution<> dis1(minX, maxX);  std::uniform\_int\_distribution<> dis2(minY, maxY);  for (int i = 0; i < sensorCount; ++i) {  int x = dis1(gen);  int y = dis2(gen);  SensorNode \*sensor = new SensorNode(i+1 ,x, y, broadcastRadius);  qDebug() << i << ";" << x << ";" << y ;  sensors.append(sensor);  }  int D\_x = 20;  int D\_y = 20;  // 创建在 1 到 4 之间生成均匀分布的整数的分布对象  std::uniform\_int\_distribution<> dis3(1, 4);  int judgeD = dis3(gen);  if(judgeD == 1){  std::uniform\_int\_distribution<> dis11(minX, maxX);  std::uniform\_int\_distribution<> dis22(minY, minY + d);  D\_x = dis11(gen);  D\_y = dis22(gen);  }  else if(judgeD == 2){  std::uniform\_int\_distribution<> dis11(minX, minX + 30);  std::uniform\_int\_distribution<> dis22(minY + d, maxY - d);  D\_x = dis11(gen);  D\_y = dis22(gen);  }  else if(judgeD == 3){  std::uniform\_int\_distribution<> dis11(maxX - d, maxX);  std::uniform\_int\_distribution<> dis22(minY + d, maxY - d);  D\_x = dis11(gen);  D\_y = dis22(gen);  }  else if(judgeD == 4){  std::uniform\_int\_distribution<> dis11(minX, maxX);  std::uniform\_int\_distribution<> dis22(maxY - d, maxY);  D\_x = dis11(gen);  D\_y = dis22(gen);  }  qDebug() << "DXY" << ";" << D\_x << ";" << D\_y ;  dataCollector->x = D\_x;  dataCollector->y = D\_y;  //将小于半径的两个节点相连接  for (int i = 0; i < sensorCount; ++i) {  for (int j = i + 1; j < sensorCount; ++j) {  double distance = euclideanDistance(sensors[i]->x, sensors[i]->y, sensors[j]->x, sensors[j]->y);  if (distance <= broadcastRadius) {  graph.addEdge(i, j, distance);  //for (Edge& edge : graph.adjList.at(i)) {  // edge.weight = edge.desister + multiple \* sensors[j]->dataQueue.size();  // qDebug() << "距离" << edge.desister << "边" << edge.weight;  //}  }  }  double distanceToDC = euclideanDistance(sensors[i]->x, sensors[i]->y, dataCollector->x, dataCollector->y);  if (distanceToDC <= broadcastRadius) {  graph.addEdge(i, sensorCount, distanceToDC); // 连接传感器和收集节点  //Edge a = graph.findEdge(sensorCount,i);  //qDebug() << sensorCount << " " << i << ";" << a.to << " " << a.weight << "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*" ;  }  }  }  void SensorNetwork::Dijkstra() {  // 初始化路径表  for (int i = 0; i <= sensorCount; i++) {  PathMap[i].Set(false, std::numeric\_limits<double>::infinity(), -1); // 将距离设为无穷大  }  // 放入第一个节点  PathMap[sensorCount].Set(true, 0, -1);  // 初始化第一个节点的邻接节点的距离  for (const Edge& edge : graph.adjList.at(sensorCount)) {  PathMap[edge.to].Set(false, edge.weight, sensorCount);  }  for (int i = 0; i < sensorCount; i++) {  // 找到未处理节点中距离最短的节点  int minNode = -1;  double minDist = std::numeric\_limits<double>::infinity();  for (const auto& entry : PathMap) {  if (!entry.second.final && entry.second.dist < minDist) {  minDist = entry.second.dist;  minNode = entry.first;  }  }  //if (minNode == -1) {  // break; // 所有节点已处理完毕  //}  PathMap[minNode].final = true; // 将该节点标记为已处理  // 更新该节点邻接节点的距离  for (const Edge& edge : graph.adjList[minNode]) {  if (!PathMap[edge.to].final && PathMap[minNode].dist + edge.weight < PathMap[edge.to].dist) {  PathMap[edge.to].Set(false, PathMap[minNode].dist + edge.weight, minNode);  }  }  }  }  void Graph::addEdge(int from, int to, double weight) {  adjList[from].push\_back({to, weight});  adjList[to].push\_back({from, weight}); // 无向图需要双向边  }  Edge Graph::findEdge(int from, int to) {  if (adjList.find(from) != adjList.end()) {  for (const Edge& edge : adjList.at(from)) {  if (edge.to == to) {  return edge;  }  }  }  return Edge();  }  void SensorNetwork::printPathMap() {  for (const auto& entry : PathMap) {  int node = entry.first;  const Data& data = entry.second;  qDebug() << "Node:" << node  << "Final:" << data.final  << "Distance:" << data.dist  << "Previous:" << data.path;  }  }  void SensorNetwork::initEdgeType(){  for (const auto& entry : PathMap) {  int node = entry.first; //节点序号  const Data& data = entry.second; //节点数据  auto path = data.path;  auto D = data.dist;  if(D != std::numeric\_limits<double>::infinity()){  // graph.findEdge(node,path).Set(2);  for (Edge& edge : graph.adjList.at(node)) {  if (edge.to == path) {  edge.Set(2);  }  else{  edge.Set(1);  }  }  }  if(D == 0){  qDebug() << "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*";  for (Edge& edge : graph.adjList.at(sensorCount)) {  edge.Set(2);  }  }  }  for(int i = 0;i < sensorCount;i++){  sensors[i]->SetType(1);  }  }  // 修改 SensorNetwork 类中的 getShortestPath 函数  QPair<double, QList<int>> SensorNetwork::getShortestPath(int targetNode) {  QList<int> path;  double totalDistance = PathMap[targetNode].dist;  // 检查目标节点是否可达  if (totalDistance == std::numeric\_limits<double>::infinity()) {  return qMakePair(totalDistance, path); // 返回空路径和无穷距离  }  // 使用堆栈逆序构建路径  QStack<int> stack;  int currentNode = targetNode;  while (currentNode != -1) {  stack.push(currentNode);  currentNode = PathMap[currentNode].path;  }  // 将堆栈中的路径顺序放入 QList 中  while (!stack.isEmpty()) {  path.append(stack.pop());  }  return qMakePair(totalDistance, path);  }  void SensorNetwork::StartT(){  for(int i = 0; i < sensorCount; i++){  int per = PathMap[i].path; // 路径中前一个节点的位置  if (!sensors[i]->dataQueue.isEmpty()) {  auto Data = sensors[i]->dataQueue.first();  sensors[i]->dataQueue.dequeue();  sensors[i]->model->removeRow(0);  if(per == sensorCount){  dataCollector->addItem(Data);  //GetNewD(i,per);  // qDebug() << "向数据收集站";  } else {  //GetNewD(i,per);  sensors[per]->receiveData(Data);  // qDebug() << "普通节点传送消息";  }  } else {  // 数据队列为空时的处理  }  }  }  // 控制节点生成数据  void SensorNetwork::GetNew(){  for(int i = 0;i < sensorCount;i++){  sensors[i]->collectData();  }  }  void SensorNetwork::GetNewD(int from,int to){  for (Edge& edge : graph.adjList.at(from)) {  if (edge.to == to) {  edge.weight = edge.desister + multiple \* sensors[to]->dataQueue.size();  qDebug() << "距离" << edge.desister << "边" << edge.weight;  }  }  } |

|  |
| --- |
| #ifndef SENSORNODE\_H  #define SENSORNODE\_H  #include <QQueue>  #include <QPoint>  #include <random>  #include <QDateTime>  #include <QStandardItemModel>  struct TData{  int fromNode;  int data;  QString type;  QDateTime currentDateTime;  TData(int a,int D,QString Type){  fromNode = a;  data = D;  type = Type;  }  QString toString() const {  return QString("From Node: %1, Data: %2, Type: %3, Date/Time: %4")  .arg(fromNode)  .arg(data)  .arg(type)  .arg(currentDateTime.toString(Qt::ISODate));  }  };  // 传感器节点的类  class SensorNode  {  public:  int a; //编号  double x; // 传感器节点的位置  double y;  int type;  double broadcastRadius; // 广播半径  QQueue<TData> dataQueue; //收集数据  QStandardItemModel \*model;  public:  SensorNode(int a ,double x, double y, double broadcastRadius);  void receiveData(const TData &data); //接收数据  void collectData(); //收集数据  void SetType(int a);  };  #endif // SENSORNODE\_H |

|  |
| --- |
| #include "sensornode.h"  SensorNode::SensorNode(int \_a ,double \_x, double \_y, double \_broadcastRadius) {  x = \_x;  y = \_y;  a = \_a;  type = 1;  broadcastRadius = \_broadcastRadius;  model = new QStandardItemModel();  }  void SensorNode::SetType(int a){  type = a;  }  void SensorNode::collectData(){  QDateTime tDateTime = QDateTime::currentDateTime();  std::random\_device rd; // 用于生成随机种子  std::mt19937 gen(rd()); // Mersenne Twister 随机数生成器  //产生温度的随机数  std::uniform\_int\_distribution<> dis1(0, 100);  int T = dis1(gen);  TData A = TData(a,T,"温度");  A.currentDateTime = tDateTime;  dataQueue.enqueue(A); //收集到一个温度数据  std::uniform\_int\_distribution<> dis2(0, 100);  int S = dis2(gen);  TData B = TData(a,S,"湿度");  B.currentDateTime = tDateTime;  dataQueue.enqueue(B); //收集到一个湿度数据  model->appendRow(new QStandardItem(A.toString()));  model->appendRow(new QStandardItem(B.toString()));  }  void SensorNode::receiveData(const TData &data){  dataQueue.enqueue(data);  model->appendRow(new QStandardItem(data.toString()));  } |

|  |
| --- |
| #ifndef WIDGET\_H  #define WIDGET\_H  #include "sensornetwork.h"  #include <QWidget>  #include <QTimer>  #include <QMessageBox>  #include <QMouseEvent>  QT\_BEGIN\_NAMESPACE  namespace Ui {  class Widget;  }  QT\_END\_NAMESPACE  class Widget : public QWidget  {  Q\_OBJECT  public:  SensorNetwork\* sensorNetwork;  QTimer\* timer;  int sensorCount;  double broadcastRadius;  int T ;  int T2;  bool Judge;  QTimer \*timer1;  QTimer \*timer2;  int lastClickedSensorIndex = -1;  int nowNode;  Widget(QWidget \*parent = nullptr);  ~Widget();  void paintEvent(QPaintEvent \*event) override;  void collect();  void Send();  void mousePressEvent(QMouseEvent \*event) override;  private slots:  void on\_pushButton\_clicked();  void on\_pushButton\_2\_clicked();  private:  Ui::Widget \*ui;  };  #endif // WIDGET\_H |

|  |
| --- |
| #include "widget.h"  #include "ui\_widget.h"  #include "sensornetwork.h"  #include <QTimer>  #include <QPainter>  #include <cmath>  // 计算欧几里得距离的函数  double euclideanDistance1(double x1, double y1, double x2, double y2) {  return std::sqrt((x1 - x2) \* (x1 - x2) + (y1 - y2) \* (y1 - y2));  }  Widget::Widget(QWidget \*parent)  : QWidget(parent)  , ui(new Ui::Widget)  {  ui->setupUi(this);  ui->label\_I->setText("输入节点序号或点击节点");  sensorCount = 100;  broadcastRadius = 150;  T = 5;  nowNode = -1;  T2 = 2;  sensorNetwork = new SensorNetwork(sensorCount,broadcastRadius,T);  sensorNetwork->initializeNetwork();  ui->lineEdit\_SNUM->setText(QString::number(sensorCount));  ui->lineEdit\_R->setText(QString::number(broadcastRadius));  ui->lineEdit\_T->setText(QString::number(T));  ui->lineEdit\_SNUM\_2->setText(QString::number(T2));  ui->lcdNumber->setSegmentStyle(QLCDNumber::Flat); // 设置显示样式  // 设置样式表使数字显示为蓝色  ui->lcdNumber->setStyleSheet("color: blue;");  ui->tableView->setModel(sensorNetwork->dataCollector->model);  ui->tableView->horizontalHeader()->setSectionResizeMode(QHeaderView::Stretch);  ui->tableView->resizeRowsToContents();  // 设置定时器更新绘图  timer = new QTimer(this);  connect(timer, &QTimer::timeout, this, QOverload<>::of(&Widget::update));  //connect(timer, &QTimer::timeout, this, &Widget::Send); //每秒发送一次数据  timer->start(1000); // 每秒更新一次  //设置定时器收集数据  timer1 = new QTimer(this);  connect(timer1, &QTimer::timeout, this, &Widget::collect);  timer1->start(T\*1000); // 每T秒更新一次  timer2 = new QTimer(this);  connect(timer2, &QTimer::timeout, this, &Widget::Send);  timer2->start(T2\*1000); // 每T2秒发送一次  connect(ui->pushButton\_2, &QPushButton::clicked, this, &Widget::on\_pushButton\_2\_clicked);  sensorNetwork->Dijkstra();  sensorNetwork->printPathMap();  sensorNetwork->initEdgeType();  }  Widget::~Widget()  {  delete ui;  }  void Widget::paintEvent(QPaintEvent \*event)  {  QPainter painter(this);  // 设置画笔颜色  // 定义要绘制图片的区域和位置  QRect targetRect(20, 10, 1200, 540); // 目标区域的位置和大小  QPixmap pixmap(":/new/prefix1/pic/DiTu.png"); // 加载图片资源  // 在指定区域内绘制图片  painter.drawPixmap(targetRect, pixmap);  // 设置画笔颜色和粗细  QPen pen(Qt::yellow);  pen.setWidth(1);  painter.setPen(pen);  QPen Tpen(Qt::blue);  Tpen.setWidth(3);  QPen Tpen1(Qt::red);  Tpen1.setWidth(4);  //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  // 绘制符合条件的边  for (int i = 0; i < sensorNetwork->sensors.size(); ++i) {  for (const Edge& edge : sensorNetwork->graph.adjList[i]) {  if (edge.to != sensorCount) {  QPointF p1(sensorNetwork->sensors[i]->x, sensorNetwork->sensors[i]->y);  QPointF p2(sensorNetwork->sensors[edge.to]->x, sensorNetwork->sensors[edge.to]->y);  if(edge.type == 1){  painter.drawLine(p1, p2);  }  }  }  }  for (const Edge& edge : sensorNetwork->graph.adjList.at(sensorCount)) {  QPointF p1(sensorNetwork->dataCollector->x, sensorNetwork->dataCollector->y);  QPointF p2(sensorNetwork->sensors[edge.to]->x, sensorNetwork->sensors[edge.to]->y);  // qDebug() << sensorNetwork->dataCollector->x << " " << sensorNetwork->dataCollector->y << " ";  // qDebug() << sensorNetwork->sensors[edge.to]->x << " " << sensorNetwork->sensors[edge.to]->y << " ";  if(edge.type == 1){  painter.drawLine(p1, p2);  }  }  //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  for (const Edge& edge : sensorNetwork->graph.adjList.at(sensorCount)) {  QPointF p1(sensorNetwork->dataCollector->x, sensorNetwork->dataCollector->y);  QPointF p2(sensorNetwork->sensors[edge.to]->x, sensorNetwork->sensors[edge.to]->y);  // qDebug() << sensorNetwork->dataCollector->x << " " << sensorNetwork->dataCollector->y << " ";  // qDebug() << sensorNetwork->sensors[edge.to]->x << " " << sensorNetwork->sensors[edge.to]->y << " ";  if(edge.type == 2){  painter.setPen(Tpen);  painter.drawLine(p1, p2);  painter.setPen(pen);  }  }  for (int i = 0; i < sensorNetwork->sensors.size(); ++i) {  for (const Edge& edge : sensorNetwork->graph.adjList[i]) {  if (edge.to != sensorCount) { // Avoid drawing to DataCollector here  QPointF p1(sensorNetwork->sensors[i]->x, sensorNetwork->sensors[i]->y);  QPointF p2(sensorNetwork->sensors[edge.to]->x, sensorNetwork->sensors[edge.to]->y);  if(edge.type == 2){  painter.setPen(Tpen);  painter.drawLine(p1, p2);  painter.setPen(pen);  }  }  }  }  //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  for (const Edge& edge : sensorNetwork->graph.adjList.at(sensorCount)) {  QPointF p1(sensorNetwork->dataCollector->x, sensorNetwork->dataCollector->y);  QPointF p2(sensorNetwork->sensors[edge.to]->x, sensorNetwork->sensors[edge.to]->y);  // qDebug() << sensorNetwork->dataCollector->x << " " << sensorNetwork->dataCollector->y << " ";  // qDebug() << sensorNetwork->sensors[edge.to]->x << " " << sensorNetwork->sensors[edge.to]->y << " ";  if(edge.type == 3){  painter.setPen(Tpen1);  painter.drawLine(p1, p2);  painter.setPen(pen);  }  }  for (int i = 0; i < sensorNetwork->sensors.size(); ++i) {  for (const Edge& edge : sensorNetwork->graph.adjList[i]) {  if (edge.to != sensorCount) { // Avoid drawing to DataCollector here  QPointF p1(sensorNetwork->sensors[i]->x, sensorNetwork->sensors[i]->y);  QPointF p2(sensorNetwork->sensors[edge.to]->x, sensorNetwork->sensors[edge.to]->y);  if(edge.type == 3){  painter.setPen(Tpen1);  painter.drawLine(p1, p2);  painter.setPen(pen);  }  }  }  }  painter.setPen(Qt::black);  painter.setBrush(QColor(66, 196, 230)); // 设置刷子颜色  //painter.drawRect(QRectF(1210, 540, 10, 10));  // 绘制传感器节点  for (SensorNode \*sensor : sensorNetwork->sensors) {  if(sensor->type == 1){  painter.drawEllipse(QPointF(sensor->x, sensor->y), 6, 6); // 画一个半径为6的圆形  }  else if(sensor->type == 2){  painter.setBrush(Qt::red); // 设置刷子颜色  painter.drawEllipse(QPointF(sensor->x, sensor->y), 8, 8); // 画一个半径为6的圆形  painter.setBrush(QColor(66, 196, 230)); // 设置刷子颜色  }  }  // 设置刷子颜色  painter.setBrush(QColor(255, 255, 153));  //绘制DataCollector  painter.drawRect(QRectF(sensorNetwork->dataCollector->x, sensorNetwork->dataCollector->y, 20, 20)); // 画一个矩形  // 设置文本颜色和字体  painter.setPen(Qt::black);  QFont font = painter.font();  font.setPointSize(10); // 设置字体大小  painter.setFont(font);  // 在矩形内绘制文本  QRect rect(sensorNetwork->dataCollector->x, sensorNetwork->dataCollector->y, 20, 20); // 矩形的位置和大小  painter.drawText(rect, Qt::AlignCenter, "DC");  ui->tableView->setModel(sensorNetwork->dataCollector->model);  if(nowNode != -1){  int value = sensorNetwork->sensors[nowNode]->dataQueue.size();  ui->lcdNumber->display(value);  }  }  void Widget::on\_pushButton\_clicked()  {  QString DNUM = ui->lineEdit\_SNUM->text();  bool ok;  int DNUMnumber = DNUM.toInt(&ok);  if (ok) {  qDebug() << "更改节点数: " << DNUMnumber;  this->sensorCount = DNUMnumber;  } else {  QMessageBox::warning(this, "Error", "请确保传感器数量输入的是数字");  }  QString R = ui->lineEdit\_R->text();  bool ok1;  int Rnumber = R.toDouble(&ok1);  if (ok1) {  qDebug() << "更改半径: " << Rnumber;  this->broadcastRadius = Rnumber;  } else {  QMessageBox::warning(this, "Error", "请确保半径输入的是数字");  }  QString T321= ui->lineEdit\_T->text();  bool ok2;  int Tnumber = T321.toInt(&ok2);  if (ok2) {  qDebug() << "更改周期: " << Tnumber;  this->T = Tnumber;  } else {  QMessageBox::warning(this, "Error", "请确保周期输入的是数字");  }  QString T123= ui->lineEdit\_SNUM\_2->text();  bool ok3;  int Tnumber2 = T123.toInt(&ok3);  if (ok3) {  qDebug() << "更改发送周期: " << Tnumber2;  this->T2 = Tnumber2;  } else {  QMessageBox::warning(this, "Error", "请确保周期输入的是数字");  }  delete this->sensorNetwork;  SensorNetwork\* NewS = new SensorNetwork(this->sensorCount,this->broadcastRadius,this->T);  this->sensorNetwork = NewS;  sensorNetwork->initializeNetwork();  sensorNetwork->Dijkstra();  sensorNetwork->printPathMap();  sensorNetwork->initEdgeType();  //设置定时器收集数据  timer1->deleteLater();  timer2->deleteLater();  timer1 = new QTimer(this);  connect(timer1, &QTimer::timeout, this, &Widget::collect);  timer1->start(T\*1000); // 每T秒更新一次  timer2 = new QTimer(this);  connect(timer2, &QTimer::timeout, this, &Widget::Send);  timer2->start(T2\*1000); // 每T2秒发送一次  }  void Widget::on\_pushButton\_2\_clicked()  {  sensorNetwork->initEdgeType();  // 在这里添加你的处理代码  // qDebug() << "PushButton 2 clicked!";  QString DNUM = ui->lineEdit\_4->text();  bool ok;  int DNUMnumber = DNUM.toInt(&ok);  if (ok) {  qDebug() << "查找节点: " << DNUMnumber;  if(DNUMnumber >= sensorCount){  QMessageBox::warning(this, "Error", "请确保输入正确的节点查找数");  return;  }  //查找最短路径  auto result = sensorNetwork->getShortestPath(DNUMnumber);  double distance = result.first;  QString message = "到DataCollector的通道距离 :" + QString::number(distance);  nowNode = DNUMnumber;  ui->label\_I->setText(message);  QList<int> path = result.second;  // qDebug() << "Shortest distance to node" << DNUMnumber << ":" << distance;  // qDebug() << "Path:";  int last = -1;  int now = path.at(0);  //int value = sensorNetwork->sensors[DNUMnumber]->dataQueue.size();  //ui->lcdNumber->display(value);  for (int i = 1; i < path.size(); ++i) {  // qDebug() << node;  last = now;  now = path.at(i);  for (Edge& edge : sensorNetwork->graph.adjList.at(now)) {  if (edge.to == last) {  edge.Set(3);  }  }  for (Edge& edge : sensorNetwork->graph.adjList.at(last)) {  if (edge.to == now) {  edge.Set(3);  }  }  }  sensorNetwork->sensors[now]->SetType(2); // 将传感器节点设置为第二类显示点  ui->listView->setModel(sensorNetwork->sensors[now]->model);  } else {  QMessageBox::warning(this, "Error", "请确保输入正确的节点查找数");  }  }  void Widget::collect(){  sensorNetwork->GetNew();  }  void Widget::Send(){  sensorNetwork->StartT();  ui->verticalSlider->setRange(0, 100);  int sliderValue = ui->verticalSlider->value();  qDebug() << "Path:" << sliderValue;  sensorNetwork->multiple = sliderValue;  sensorNetwork->Dijkstra();  }  void Widget::mousePressEvent(QMouseEvent \*event)  {  if (event->button() == Qt::LeftButton) {  // 获取鼠标点击的位置  QPointF clickPos = event->position();  // 遍历传感器节点，检查是否点击到了某个传感器节点  for (int i = 0; i < sensorNetwork->sensors.size(); ++i) {  SensorNode \*sensor = sensorNetwork->sensors[i];  QPointF sensorPos(sensor->x, sensor->y);  // 计算点击位置与传感器节点位置的距离  double distance = euclideanDistance1(clickPos.x(), clickPos.y(), sensorPos.x(), sensorPos.y());  // 如果距离小于传感器节点半径（这里假设传感器节点半径为固定值  if (distance < 7) {  ui->lineEdit\_4->setText(QString::number(i));  sensorNetwork->initEdgeType();  // 将点击的传感器节点设置为第二类（type为2）  //查找最短路径  auto result = sensorNetwork->getShortestPath(i);  double distance = result.first;  QString message = "到DataCollector的通道距离 :" + QString::number(distance);  ui->label\_I->setText(message);  nowNode = i;  // int value = sensorNetwork->sensors[i]->dataQueue.size();  // ui->lcdNumber->display(value);  // QMessageBox::information(this,"查找成功", message);  QList<int> path = result.second;  // qDebug() << "Shortest distance to node" << DNUMnumber << ":" << distance;  // qDebug() << "Path:";  int last = -1;  int now = path.at(0);  for (int i = 1; i < path.size(); ++i) {  // qDebug() << node;  last = now;  now = path.at(i);  for (Edge& edge : sensorNetwork->graph.adjList.at(now)) {  if (edge.to == last) {  edge.Set(3);  }  }  for (Edge& edge : sensorNetwork->graph.adjList.at(last)) {  if (edge.to == now) {  edge.Set(3);  }  }  }  sensorNetwork->sensors[now]->SetType(2); // 将传感器节点设置为第二类显示点  ui->listView->setModel(sensorNetwork->sensors[now]->model);  // 将最后一次点击的传感器节点索引记录下来  lastClickedSensorIndex = i;  break; // 找到了点击的传感器节点，退出循环  }  }  }  } |

|  |
| --- |
| #include "widget.h"  #include <QApplication>  int main(int argc, char \*argv[])  {  QApplication a(argc, argv);  Widget w;  w.show();  return a.exec();  } |