目录

[1. 课题分析 1](#_Toc155444308)

[1.1 课题内容 1](#_Toc155444309)

[1.2 实现目标 1](#_Toc155444310)

[1.2.1 Dijkstra算法 1](#_Toc155444311)

[1.2.2 Floyd算法 2](#_Toc155444312)

[2. 算法设计与程序流程 2](#_Toc155444313)

[2.1 Dijkstra算法的动态演示 2](#_Toc155444314)

[2.1.1 程序思路 2](#_Toc155444315)

[2.1.2 流程图 11](#_Toc155444316)

[2.2 Floyd算法的动态演示 11](#_Toc155444317)

[2.2.1 程序思路 11](#_Toc155444318)

[2.2.2 流程图 14](#_Toc155444319)

[3. 关键的难点问题以及解决方案 15](#_Toc155444320)

[3.1 Dijkstra数据的组织形式 15](#_Toc155444321)

[3.2 Dijkstra中间过程的保存与演示 15](#_Toc155444322)

[3.3 SFML图形绘制库的安装与使用 16](#_Toc155444323)

[3.4 保存Floyd算法更新节点的位置 17](#_Toc155444324)

[4. 程序的演示与使用 17](#_Toc155444325)

[4.1 Dijkstra算法演示 17](#_Toc155444326)

[4.2 Floyd算法演示 19](#_Toc155444327)

# 课题分析

## 1.1 课题内容

本课题为最短路径算法的动态演示，具体包括Dijkstra算法和Floyd算法（多源最短路径算法）的演示，需要自主设计测试数据，并利用图形绘制库，将算法的过程以可视化的方式实现。

Dijkstra算法基本原理为：

1、先选定一个根结点，并选定一个数组，先确定未遍历前的初始距离，把距离最短的邻接结点选定为中间结点，并标记访问过，开始往下遍历，挨个访问那个中间结点的邻接结点。计算出根结点到中间结点+中间结点到新邻接结点的距离，作为新距离，对比新距离和旧距离，如果新距离大，则把新距离替换掉旧距离，否则不变。

2、一轮访问结束后，从未标记的结点中选定距离最短的，把它作为中间结点，继续往下访问。若都标记过，则算法结束。

其时间复杂度为O(n^2)，且为单源最短路径，只能求一个顶点到其他顶点的最短路径。

Floyd算法基本原理为：

Floyd算法是求解多源最短路（多对多）的算法，即确定每个节点（起点）到其他节点（终点）的最短路径；

算法适用于有向图、无向图，允许边的权重为负，但是负边构成的回路（环）的权重之和不能为负（负环）；

算法通过不断“插点”的方式，更新节点之间的最短路，当所有节点都考虑一遍（“插点”）之后，图中所有节点之间的最短路得以确定。

Floyd 算法是一个基于「贪心」、「动态规划」求一个图中所有点到所有点最短路径的算法，时间复杂度 O(n3)。

## 1.2 实现目标

### 1.2.1 Dijkstra算法

Dijkstra算法的主要目标是找到图中两个节点之间的最短路径。从起始节点开始，逐步选择最短的未访问节点，直到到达目标节点。在每个步骤中，算法都会更新当前节点到其邻居节点的最短距离，我们需要存储这些更新步骤，并采用合适的方法将其可视化，以演示算法的执行过程。

### 1.2.2 Floyd算法

Floyd算法，也被称为Floyd-Warshall算法，是一种用于找到给定加权图中所有顶点对之间的最短路径的算法。这是一个动态规划算法，其基本思想是逐步改进路径长度的估计值，直到得到准确的结果。

# 算法设计与程序流程

## 2.1 Dijkstra算法的动态演示

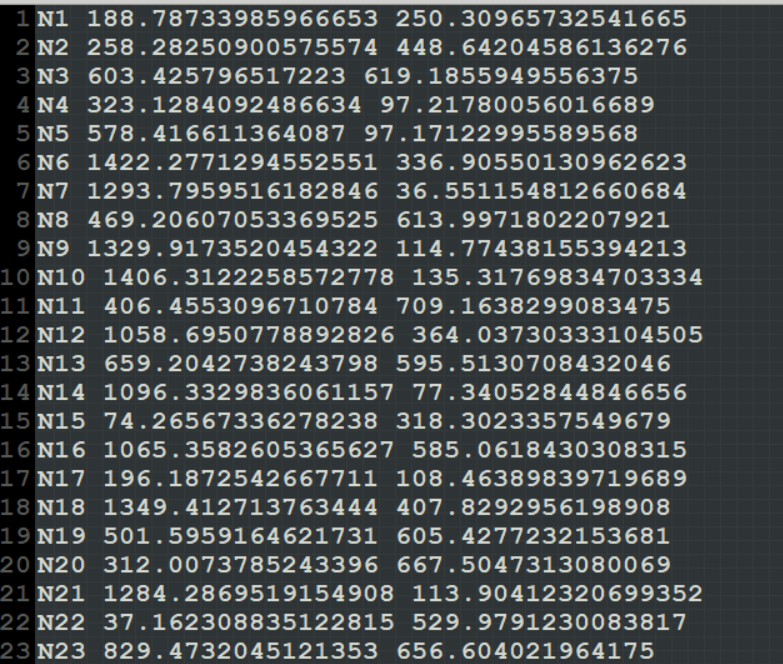
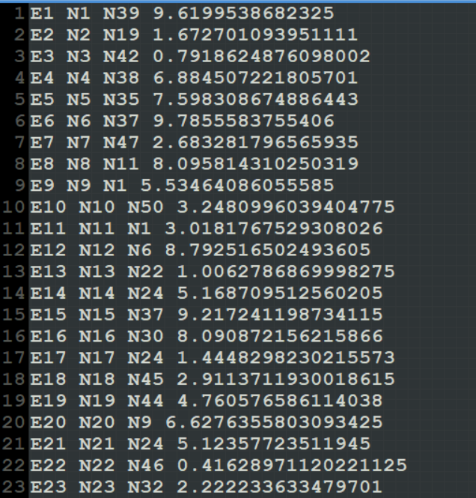
### 2.1.1 程序思路

**1）生成数据：**

考虑到算法的实际应用意义，使用python生成一定数量的节点与边，每个节点带有随机的xy坐标，根据随机坐标计算两点间的距离作为边的权重，这样在窗体上绘制时可直观表现边的权重关系。

左图为节点数据，分别为节点名、节点的x与y坐标；右图为边数据，依次为边名、两端节点名，边权重（根据坐标使用两点之间距离公式计算得到）。

为保证算法的复用性，可以直接在python代码中修改节点与边的数量，cpp代码会直接调用修改后的nodes与edges文件。



**2）组织数据使其适用于算法：**

首先我们定义两个结构体Edge与Node，分别用于表示图的边和节点，Edge结构体包含了四个成员变量：edgeName为边的名称，startnode和endnode为边的起始节点和结束节点，length表示边的长度；Node结构体包含三个成员变量：nodeName为节点的名称，x和y表示节点在二维空间中的坐标。

1. **struct** Edge
2. {
3. string edgeName;
4. string startNode;
5. string endNode;
6. **double** length;
7. };
8. **struct** Node
9. {
10. string nodeName;
11. **double** x;
12. **double** y;
13. }

接下来，使用两个函数readNodes和readEdges，从文件中读取节点信息和边信息，readNodes 函数接收一个文件路径作为参数，然后打开该文件并读取节点信息。每行文件包含节点的名称、x坐标和y坐标，这些信息被读取并存储在 Node 结构体的对象中，最后将这些节点对象存储在一个 vector<Node> 中，并返回该向量；readEdges 函数与 readNodes 函数类似，不同之处在于它读取包含边信息的文件。每行文件包含边的名称、起始节点、结束节点和长度，这些信息被读取并存储在 Edge 结构体的对象中，最后将这些边对象存储在一个 vector<Edge> 中，并返回该向量。

1. //read nodes from file
2. vector<Node> readNodes(string filepath)
3. {
4. ifstream file(filepath);
5. string nodeName;
6. **double** x, y;
7. vector<Node> nodes;
8. **while** (file >> nodeName >> x >> y)
9. {
10. nodes.push\_back({nodeName, x, y });
11. }
12. **return** nodes;
13. }
14. //read edges from file
15. vector<Edge> readEdges(string filepath)
16. {
17. ifstream file(filepath);
18. string edgeName, startNode, endNode;
19. **double** length;
20. vector<Edge> edges;
21. **while** (file >> edgeName >> startNode >> endNode >> length)
22. {
23. edges.push\_back({ edgeName, startNode, endNode, length });
24. }
25. **return** edges;
26. }

之后定义一个图数据结构Graph，使用unordered\_map表示图的邻接表，unordered\_map，其中键是节点的名称（字符串），值是一个vector，其中包含了与该节点相邻的节点和对应的边的长度，用 pair<string, double> 表示。创建图结构后在主函数中遍历边的集合edges，对于每条边，将其起始节点和结束节点加入到图的邻接表中。这里使用 graph[edge.startNode] 获取起始节点对应的邻接节点列表，然后通过 emplace\_back 将结束节点和边的长度添加到该列表中。同样，对于结束节点，也将起始节点和边的长度添加到其邻接节点列表中。这是因为无向图中的边是双向的，所以需要在两个节点的邻接表中都添加对方。

1. //define the graph
2. typedef unordered\_map<string,vector<pair<string,**double**>>> Graph;
3. //create the graph
4. Graph graph;
5. **for** (**const** auto& edge : edges)
6. {
7. graph[edge.startNode].emplace\_back(edge.endNode, edge.length);
8. graph[edge.endNode].emplace\_back(edge.startNode, edge.length);
9. }

**3）使用dijkstra算法求两点之间的在最短路径**

首先创建优先队列，完成距离映射初始化。pq: 优先队列，用于按照节点到起始节点的距离的升序访问节点；distances: 保存从起始节点到各个节点的最短距离；previous: 保存从起始节点到达某节点的最短路径上，该节点的前一个节点；pathUpdates: 用于保存算法执行过程中每一步的信息，包括当前节点、下一个节点和一个表示是否更新了最短路径的标志。

1. priority\_queue<pair<**double**, string>, vector<pair<**double**, string>>, greater<pair<**double**, string>>> pq;
2. unordered\_map<string, **double**> distances; // distance from start to each node
3. unordered\_map<string, string> previous;   // previous node in optimal path from start
4. vector<tuple<string, string, string>> pathUpdates;  // save the progress

接着规定初始化节点到起始节点的距离为无穷大，除了起始节点的距离为0。将起始节点和距离0入队列。

1. **for** (**const** auto& entry : graph)
2. {
3. distances[entry.first] = numeric\_limits<**double**>::infinity();
4. previous[entry.first] = "";
5. }
6. distances[start] = 0;
7. pq.push({0, start});

然后为Dijkstra算法主体部分，使用优先队列不断从队列中取出当前距离起始节点最近的节点，然后更新其邻居节点的距离信息。如果通过当前节点到达邻居节点的距离比已知的距离短，则更新距离和前一个节点，并将邻居节点加入队列。

pathUpdates 是一个记录算法执行过程中的详细信息的容器。具体来说，pathUpdates 中的每个元素都是一个三元组，表示一次算法迭代的结果。这三个元素分别是当前节点 current、下一个节点 next，以及一个标志位（字符串） "True" 或 "False"，用于指示是否更新了最短路径。这容器的目的主要是为了记录 Dijkstra 算法的执行过程中每一步的情况，包括节点的访问顺序以及是否有新的最短路径被找到。

1. **while** (!pq.empty())
2. {
3. string current = pq.top().second;
4. pq.pop();
6. **for** (**const** auto& neighbor : graph.at(current))
7. {
8. string next = neighbor.first;
9. **double** weight = neighbor.second;
10. **double** new\_distance = distances[current] + weight;
12. **if** (new\_distance < distances[next])
13. {
14. distances[next] = new\_distance;
15. previous[next] = current;
16. pq.push({new\_distance, next});
17. pathUpdates.push\_back({current, next, "True"});
18. }
19. **else**
20. {
21. pathUpdates.push\_back({current, next, "False"});
22. }
23. }
24. }

最后重构最短路径并返回结果，主要根据最短路径的前一个节点信息，逆序构建最短路径。

1. vector<string> path;
2. string current = end;
3. **while** (!current.empty())
4. {
5. path.insert(path.begin(), current);
6. current = previous[current];
7. }

整体而言，这段代码实现了 Dijkstra 算法，并在执行过程中记录了算法的每一步信息，包括节点的访问顺序和是否更新了最短路径。该Dijkstra算法中使用的pathUpdates变量，保存了算法执行过程中每一步的信息，包括当前节点、下一个节点和一个表示是否更新了最短路径的标志。

**4）算法的可视化演示**

首先创建一个从节点名称到二维坐标（使SFML库中Vector2f 类型表示）的映射nodeCoordinates，进而为图中的每个节点创建一个映射，将节点名称与其在二维空间中的坐标关联起来。具体为定义了一个 map 数据结构，将节点的名称（字符串）映射到其对应的二维坐标 (Vector2f 类型)；通过 for 循环遍历存储节点信息的 nodes 向量。对于每个节点，将其名称作为键，而使用 Vector2f(node.x, node.y) 创建的二维坐标作为值，将它们添加到 nodeCoordinates 映射中。

1. //create the coordinates mapping
2. map<string, Vector2f> nodeCoordinates;
3. **for** (**const** Node& node : nodes)
4. {
5. nodeCoordinates[node.nodeName] = Vector2f(node.x, node.y);
6. }

接着创建一个SFML窗口并加载字体，其大小为1500x900像素，标题为 "Dijkstra Algorithm Dynamic Demonstration"，尝试从文件 "arial.ttf" 中加载字体。如果加载失败，输出错误消息。这个字体可能会在后续的 SFML 渲染中用于显示文本。值得注意的是字体文件需保存在项目文件夹中并使用相对路径，若缺少字体文件则会导致控件无法加载。

1. //create SFML window
2. RenderWindow window(VideoMode(1500, 900), "Dijkstra Algorithm Dynamic Demonstration");
3. //create the font
4. Font font;
5. **if** (!font.loadFromFile("arial.ttf"))
6. {
7. cout<<"Error loading font"<<endl;
8. }

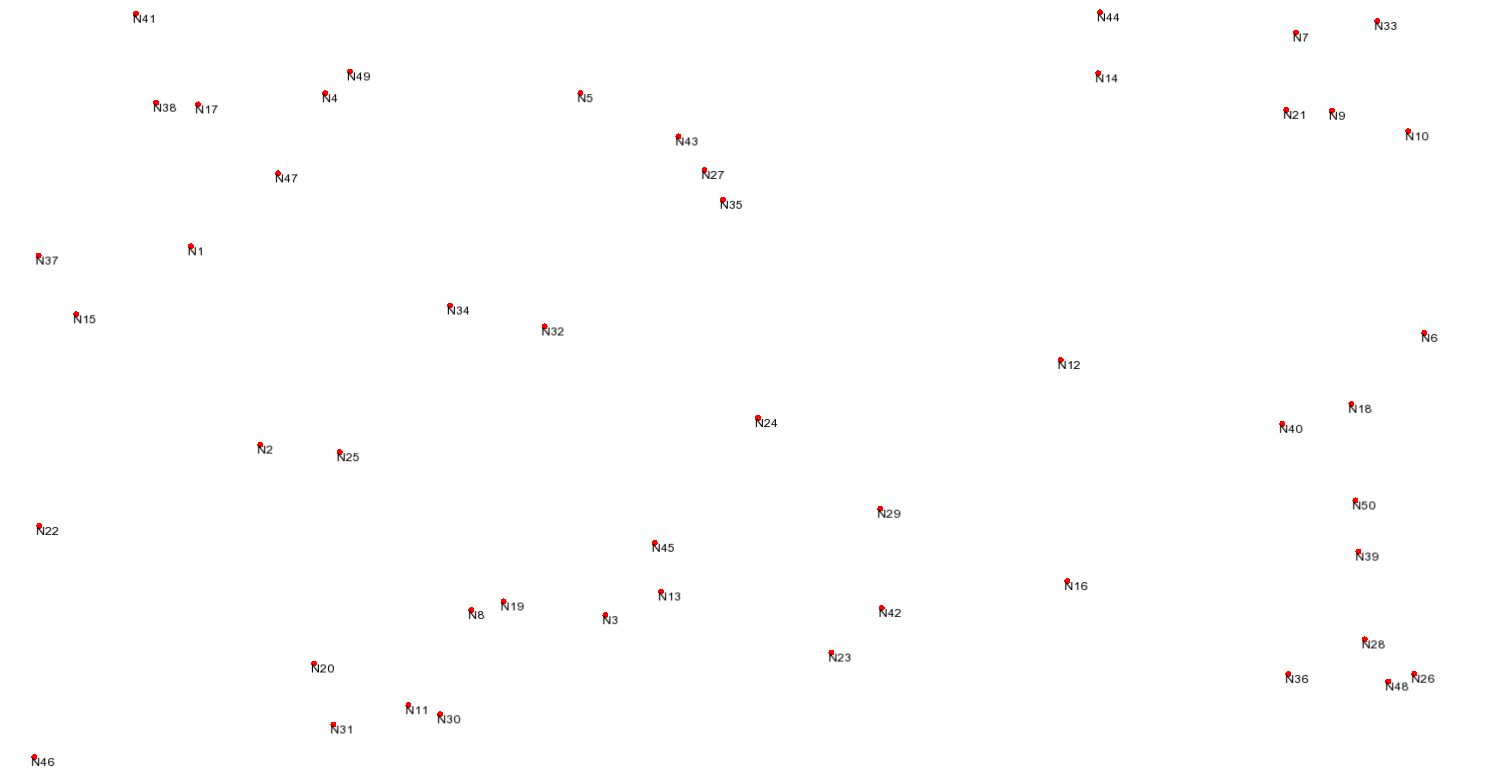
然后在窗口中绘制一组圆形节点（用红色圆圈表示）以及每个节点对应的文字标签（用黑色字体显示）

1. //add the nodes to the window
2. vector<CircleShape> shapes; //nodes
3. **for** (**const** auto& node : nodes)
4. {
5. CircleShape shape(3);
6. shape.setFillColor(Color::Red);
7. shape.setPosition(node.x, node.y);
8. shapes.push\_back(shape);
9. }

之后开启主循环，它处理窗口事件并在每次循环中绘制节点和节点标签。首先，通过window.clear(Color(255, 255, 255));清空窗口并设置白色背景。

1. //main loop
2. **while** (window.isOpen())
3. {
4. Event event;
5. **while** (window.pollEvent(event))
6. {
7. **if** (event.type == Event::Closed)
8. {
9. window.close();
10. }
11. }
13. //white background
14. window.clear(Color(255, 255, 255));

再通过遍历shapes vector，将每个圆形形状绘制到窗口中，然后通过遍历节点，创建包含节点名称的文字标签，并将其绘制到窗口中。



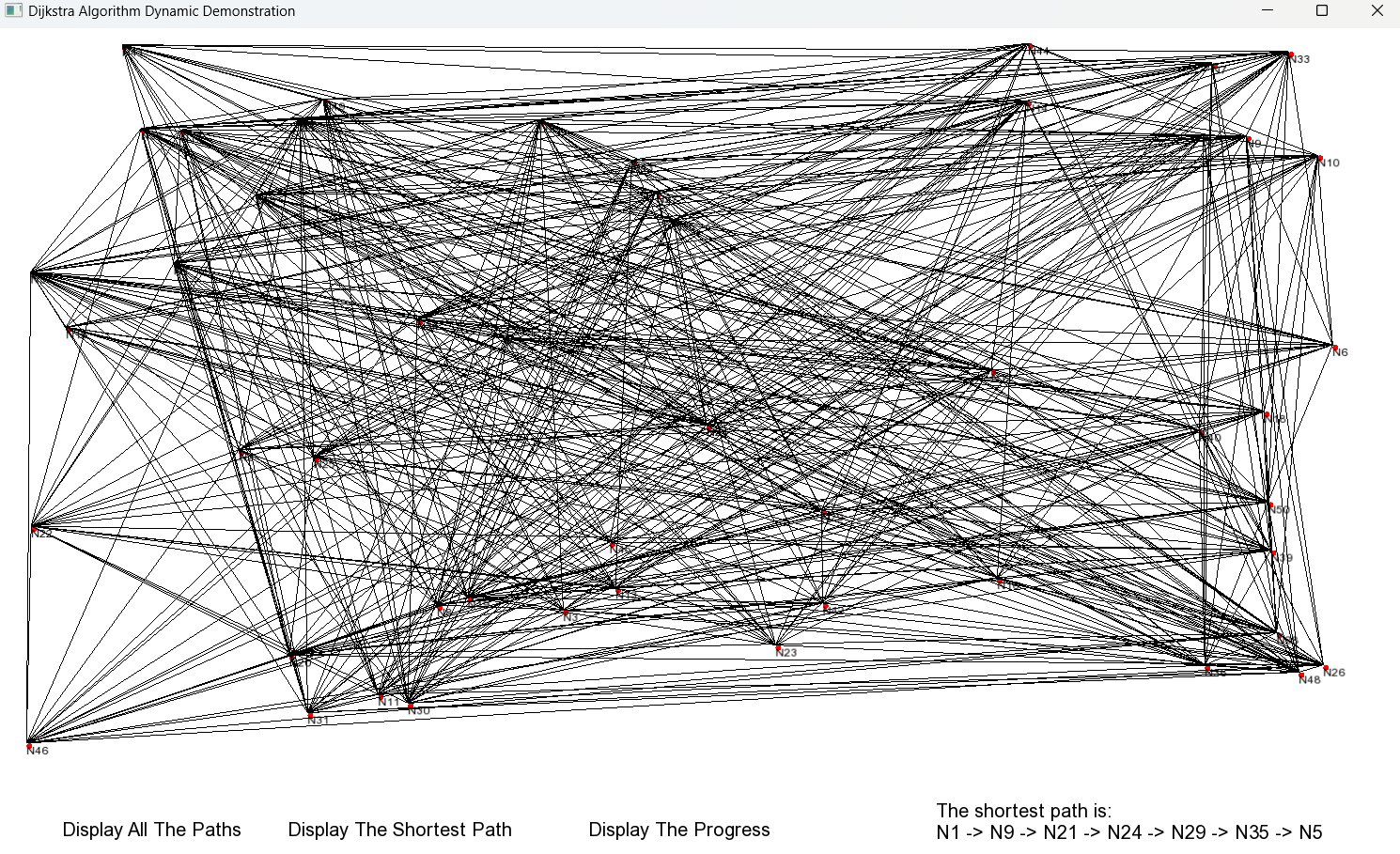
1. //draw the nodes
2. **for** (**const** auto& shape : shapes)
3. {
4. window.draw(shape);
5. }
7. //draw the font
8. **for** (**const** auto& node : nodes)
9. {
10. Text text;
11. text.setFont(font);
12. text.setString(node.nodeName);
13. text.setCharacterSize(12);
14. text.setFillColor(Color::Black);
15. text.setPosition(node.x ,node.y );
16. window.draw(text);
17. }

然后创建按钮来和SFML图形窗口进行交互，我们以显示全部路径为例：首先创建了一个白色矩形形状（button\_path），用于表示按钮。该矩形的大小是(215, 50)，位置为(60, 830)，颜色设置为白色，透明度为300。接着，创建了显示在按钮上的文本（buttonText），内容为"Display All The Paths"，字体大小为20，颜色为黑色，位置为(70, 840)。最后，声明了一个布尔变量isVisible\_path，用于表示路径是否可见，初始值为false。这段代码的目的是创建一个按钮，用户可以点击以显示或隐藏路径。按钮的状态由isVisible\_path变量控制。

1. //create the button for display the path
2. RectangleShape button\_path(Vector2f(215, 50));
3. button\_path.setPosition(60, 830);
4. button\_path.setFillColor(Color(255, 255, 255, 300));    //the last parameter is the transparency of the color
5. //create the text for the button
6. Text buttonText("Display All The Paths", font, 20);
7. buttonText.setFillColor(Color::Black);
8. buttonText.setPosition(70, 840);
9. //create the bool variable for the visibility of the path
10. **bool** isVisiable\_path = **false**;

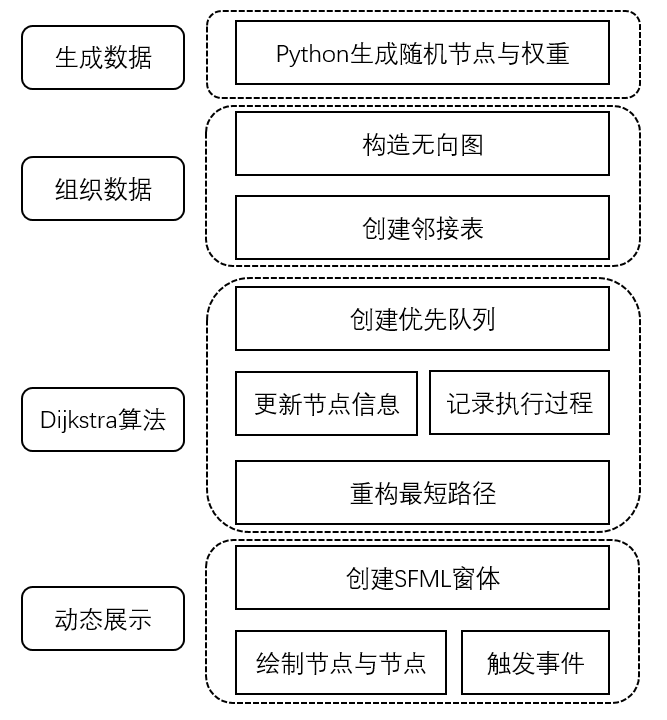
然后为处理鼠标点击事件，并在点击按钮区域时切换路径的可见性。如果路径可见 (isVisiable\_path 为真)，则通过绘制直线来显示所有路径。具体解释。首先检查鼠标事件是否是左键按下，并且点击位置是否在按钮区域内。如果是在按钮区域内点击，则切换路径的可见性状态 (isVisiable\_path)。如果路径可见，遍历所有边 (Edge) 并绘制路径。使用 VertexArray 来创建包含两个点的线段，每个点表示路径的起点和终点。绘制的线段颜色为黑色。

1. //mouse click event (draw all the paths)
2. **if** (event.type == Event::MouseButtonPressed && event.mouseButton.button == Mouse::Left)
3. {
4. //check if the mouse click is within the button's area
5. **if** (button\_path.getGlobalBounds().contains(window.mapPixelToCoords(Mouse::getPosition(window))))
6. {
7. isVisiable\_path = !isVisiable\_path;
8. }
9. }
10. **if** (isVisiable\_path)
11. {
12. //Toggle the visibility of the path
13. **for** (**const** Edge& edge : edges)
14. {
15. VertexArray line(Lines, 2);
16. line[0].position = nodeCoordinates[edge.startNode];
17. line[0].color = Color::Black;
18. line[1].position = nodeCoordinates[edge.endNode];
19. line[1].color = Color::Black;
20. window.draw(line);
21. }
22. }



其他绘制方法基本同上。

### 2.1.2 流程图



## 2.2 Floyd算法的动态演示

### 2.2.1 程序思路

**1）定义带权重无向图的邻接矩阵**

vector<vector<int>> graph = {...};: 这一行定义了一个二维 vector，名为 graph，用于表示图的邻接矩阵。这个邻接矩阵是一个正方形矩阵，其中 graph[i][j] 表示顶点 i 和顶点 j 之间的边的权重。如果 graph[i][j] 的值为 INF，则表示顶点 i 和顶点 j 之间没有直接的边。

1. vector<vector<**int**>> graph = {
2. {0,2,1,INF,INF},
3. {2,0,5,4,INF},
4. {1,5,0,5,INF},
5. {INF,4,5,0,6},
6. {INF,INF,INF,6,0}
7. };

**2）绘制无向图与表格**

showWindow() 函数可能用于显示窗口。setPlayingSpeed(200) 设置播放速度为 200（这个函数的具体实现不在提供的代码中）。通过 drawLine 函数绘制了图的边，通过 drawPoint 函数绘制了节点。使用 drawText 函数添加了节点的编号、权重等信息。通过 drawTable 函数绘制了两个矩阵，分别是距离矩阵和下一个节点矩阵。最后，通过 for 循环绘制了每个矩阵的行和列的标号。

drawTable这个函数通过两个嵌套的循环，在给定的起始坐标 (startX, startY) 处开始，绘制水平和垂直的线条，形成一个表格的结构。每个单元格的宽度为 cellWidth，高度为 cellHeight。水平线和垂直线的数量由 rows 和 columns 决定。绘制线条的宽度为 1，颜色由参数 lineColor 指定。

1. **void** drawTable(**int** rows, **int** columns, **float** startX, **float** startY, **float** cellWidth, **float** cellHeight, Gdiplus::Color lineColor)
2. {
3. **float** endX = startX + cellWidth \* columns; **float** endY = startY + cellHeight \* rows;
4. // Draw horizontal lines
5. **for** (**int** i = 0; i <= rows; ++i)
6. {
7. **float** y = startY + i \* cellHeight;
8. drawLine(startX, y, endX, y, 1, lineColor);
9. }
10. // Draw vertical lines
11. **for** (**int** i = 0; i <= columns; ++i)
12. {
13. **float** x = startX + i \* cellWidth;
14. drawLine(x, startY, x, endY, 1, lineColor);
15. }
16. }

**3）Floyd算法**

首先初始化数据结构，dist 保存了两点之间的最短距离，next 保存了最短路径中下一个点的信息，p\_Distance 和 p\_Next 保存了需要修改的图形显示的位置。

1. vector<vector<**int**>> dist(V, vector<**int**>(V));
2. vector<vector<**int**>> next(V, vector<**int**>(V, 0));
3. vector<vector<**void**\*>>p\_Distance(V, vector<**void**\*>(V, 0));
4. vector<vector<**void**\*>>p\_Next(V, vector<**void**\*>(V, 0));

接着初始化距离矩阵和下一个节点矩阵，距离矩阵 dist 被初始化为输入图的权重，下一个节点矩阵 next 被初始化为直接相邻的节点。同时，通过 drawText 函数在图形界面上显示相应的文本。

1. **for** (**int** i = 0; i < V; i++)
2. {
3. **for** (**int** j = 0; j < V; j++)
4. {
5. dist[i][j] = graph[i][j];  //Initialize the distance between two points
6. **if** (dist[i][j] != INF)
7. {
8. p\_Distance[i][j]=drawText(dist[i][j], startX\_Distance + 20+ i \* 50, startY\_Distance + 20+ j \* 50, 24, Gdiplus::Color::Green);
9. }
10. **else**
11. {
12. p\_Distance[i][j]=drawText("INF", startX\_Distance  + i \* 50, startY\_Distance + 20 + j \* 50, 24, Gdiplus::Color::Green);
13. }
14. **if** (graph[i][j] != INF && i != j)
15. {
16. next[i][j] = j;  //Initialize the next point of the shortest path
17. p\_Next[i][j] = drawText(j, startX\_Next + 20 + i \* 50, startY\_Next + 20 + j \* 50, 24, Gdiplus::Color::Green);
18. }
19. }
20. }

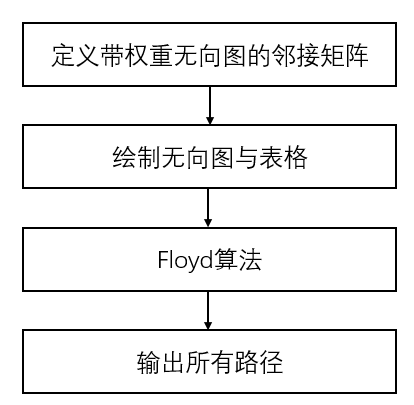
最后是Folyd算法的主循环，通过中间点 k 更新两点之间的最短路径信息。如果通过中间点的路径更短，就更新距离矩阵 dist 和下一个节点矩阵 next，同时更新图形显示的文本位置。

1. **for** (**int** k = 0; k < V; k++)
2. {
3. **for** (**int** i = 0; i < V; i++)
4. {
5. **for** (**int** j = 0; j < V; j++)
6. {
7. **if** (dist[i][k] + dist[k][j] < dist[i][j])
8. {
9. dist[i][j] = dist[i][k] + dist[k][j];
10. clearObject(p\_Distance[i][j]);
11. p\_Distance[i][j] = drawText(dist[i][j], startX\_Distance + 10 + i \* 50, startY\_Distance + 20 + j \* 50, 24, Gdiplus::Color::Red);
13. next[i][j] = next[i][k];  //Update the next point of the shortest path
14. clearObject(p\_Next[i][j]);
15. p\_Next[i][j] = drawText(next[i][j], startX\_Next + 20 + i \* 50, startY\_Next + 20 + j \* 50, 24, Gdiplus::Color::Red);
16. }
17. }
18. }
19. }

**4）输出所有路径**

输出图中所有节点对之间的最短路径。

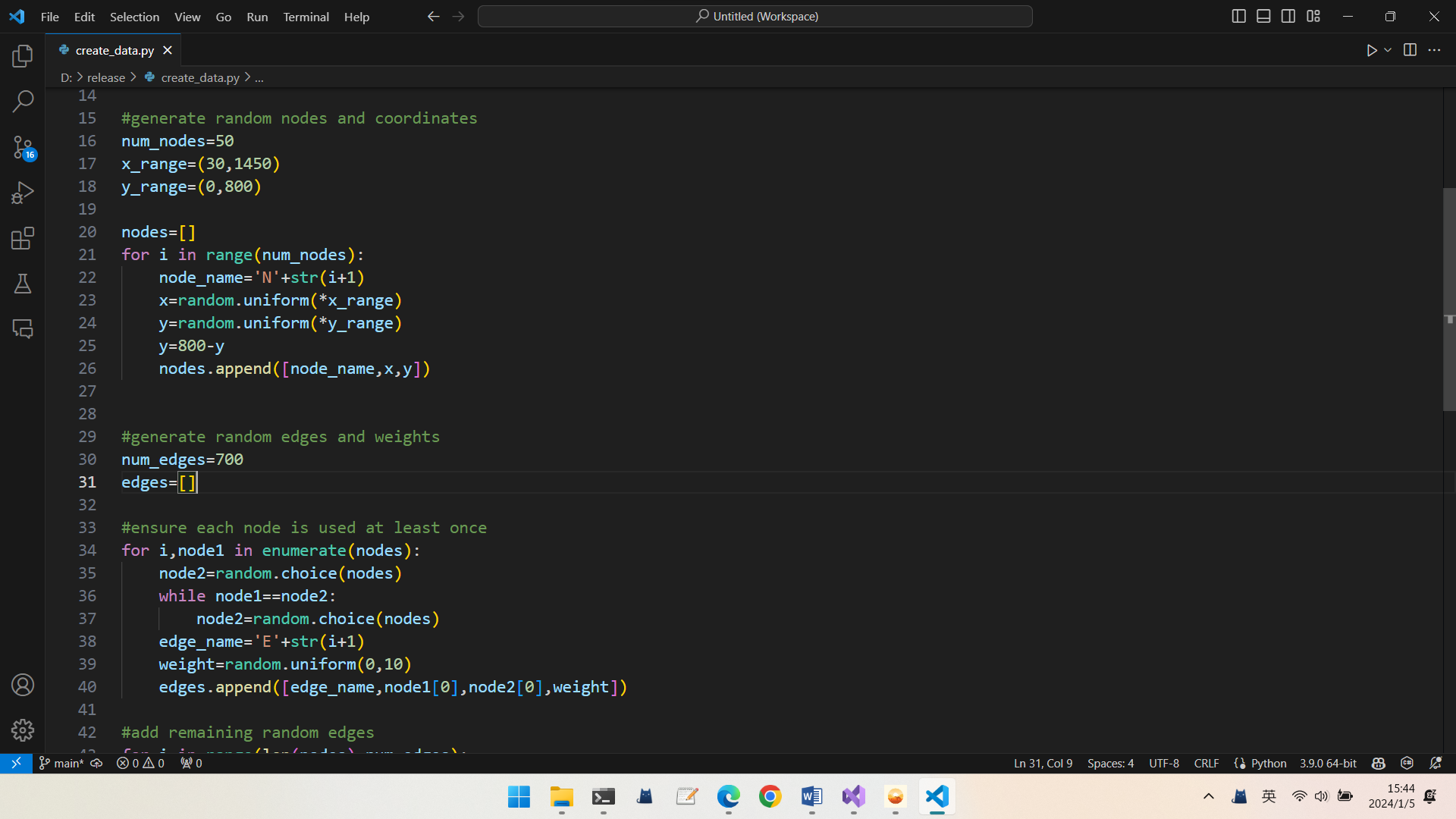
### 2.2.2 流程图



# 关键的难点问题以及解决方案

## 3.1 Dijkstra数据的组织形式

通过指定范围生成一定数量的随机节点（num\_nodes=50），每个节点包括名称、x坐标和y坐标。保证每个节点至少被使用一次，生成对应的随机边（num\_edges=700），每条边包括名称、起始节点、目标节点和权重。将剩余的边以随机方式连接节点，计算两节点间的距离作为权重。将节点和边的信息分别写入两个txt文件。



## 3.2 Dijkstra中间过程的保存与演示

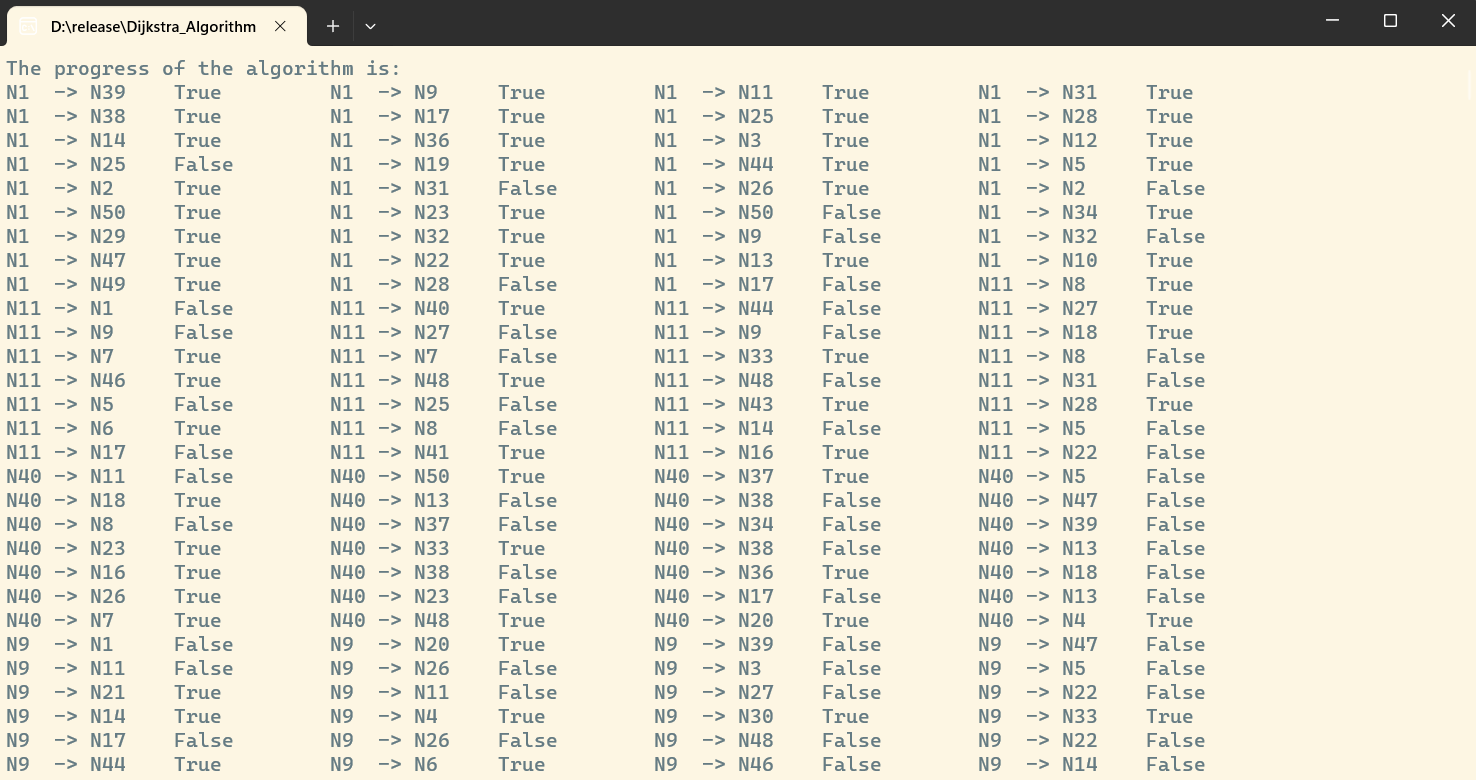
使用pathUpdates 变量记录了 Dijkstra 算法执行过程中每一步的路径更新情况。每次在算法的主循环中，当发现可以通过更短的路径到达某个节点时，会更新该节点的距离信息，并将路径的更新情况记录下来。这个记录包含了以下信息：

current：当前节点。

next：当前节点的邻居节点。

"True" 或 "False"：表示该更新是否实际修改了从起始节点到 next 节点的最短路径。

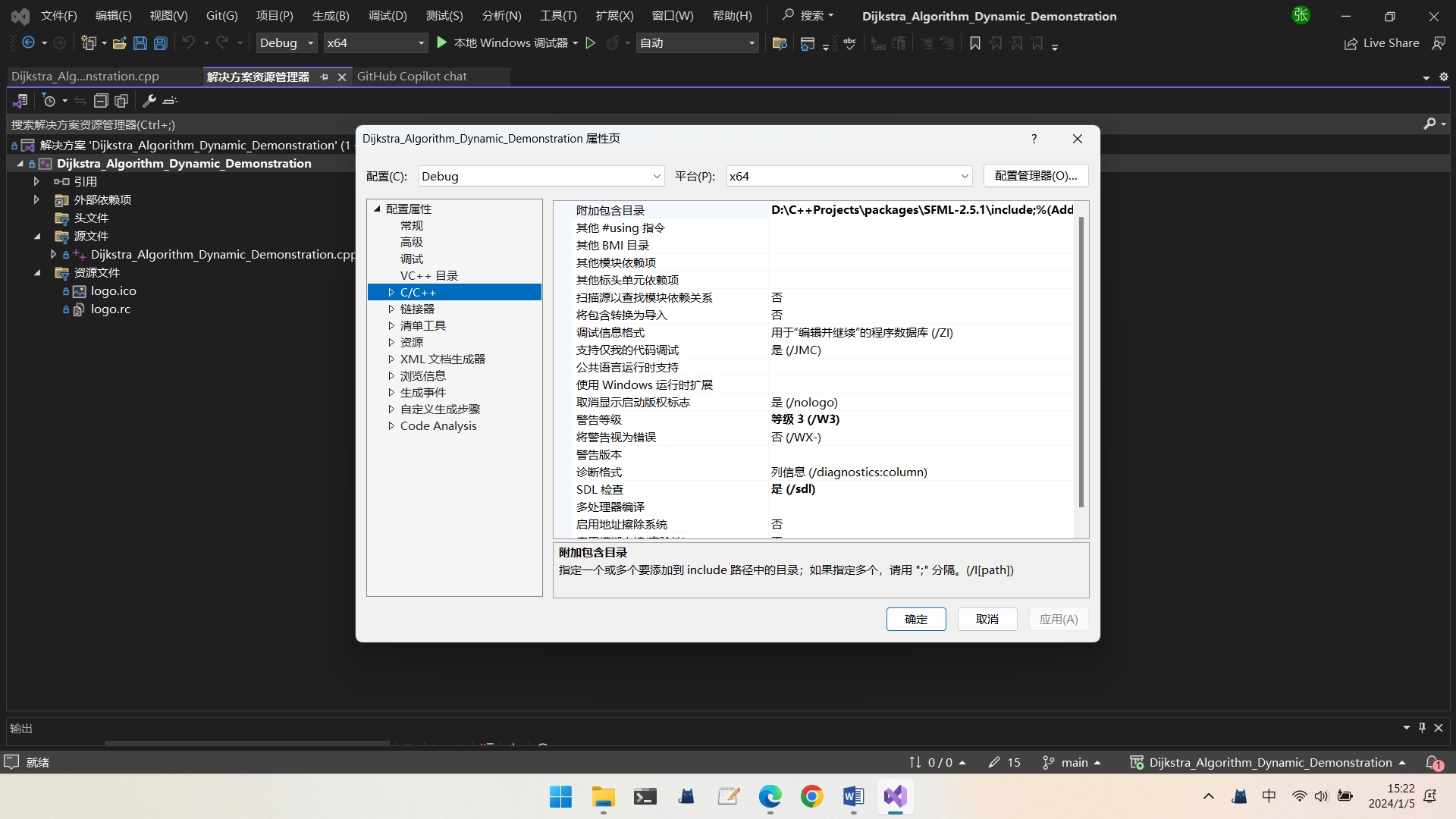
算法执行过程中遍历的节点以及是否更新会首先被输出到控制台中。



## 3.3 SFML图形绘制库的安装与使用

SFML（Simple and Fast Multimedia Library）是一个跨平台的开源C++图形库，用于实现2D游戏和多媒体应用程序。它提供了一系列易于使用的模块，包括图形渲染、窗口管理、音频处理、网络功能等，使开发者能够轻松地创建交互式和多媒体丰富的应用程序。

主要难点是将SFML图形库配置到项目中，具体是在对象资源管理器的属性中添加文件路径。由于采用动态编译而非静态编译，所以最后的可执行文件必须和SFML的dll文件在同一文件夹下才能运行。



## 3.4 保存Floyd算法更新节点的位置

创建了一个包含V行和列的二维向量p\_Next，其中每个元素是一个指向void（void\*）的指针。这个向量用于存储需要修改的数据的位置。保存每次更新节点在表格中的位置后如需再次更新，使用clearObject先清空此位置中的节点内容，再使用drawText绘制。

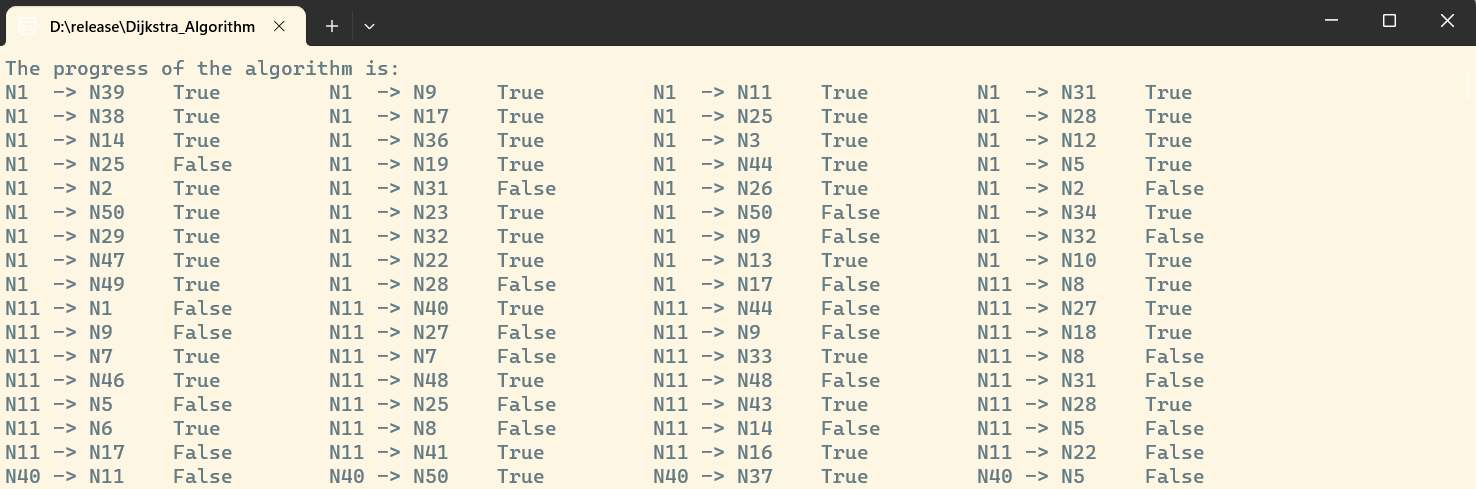
# 程序的演示与使用

## 4.1 Dijkstra算法演示

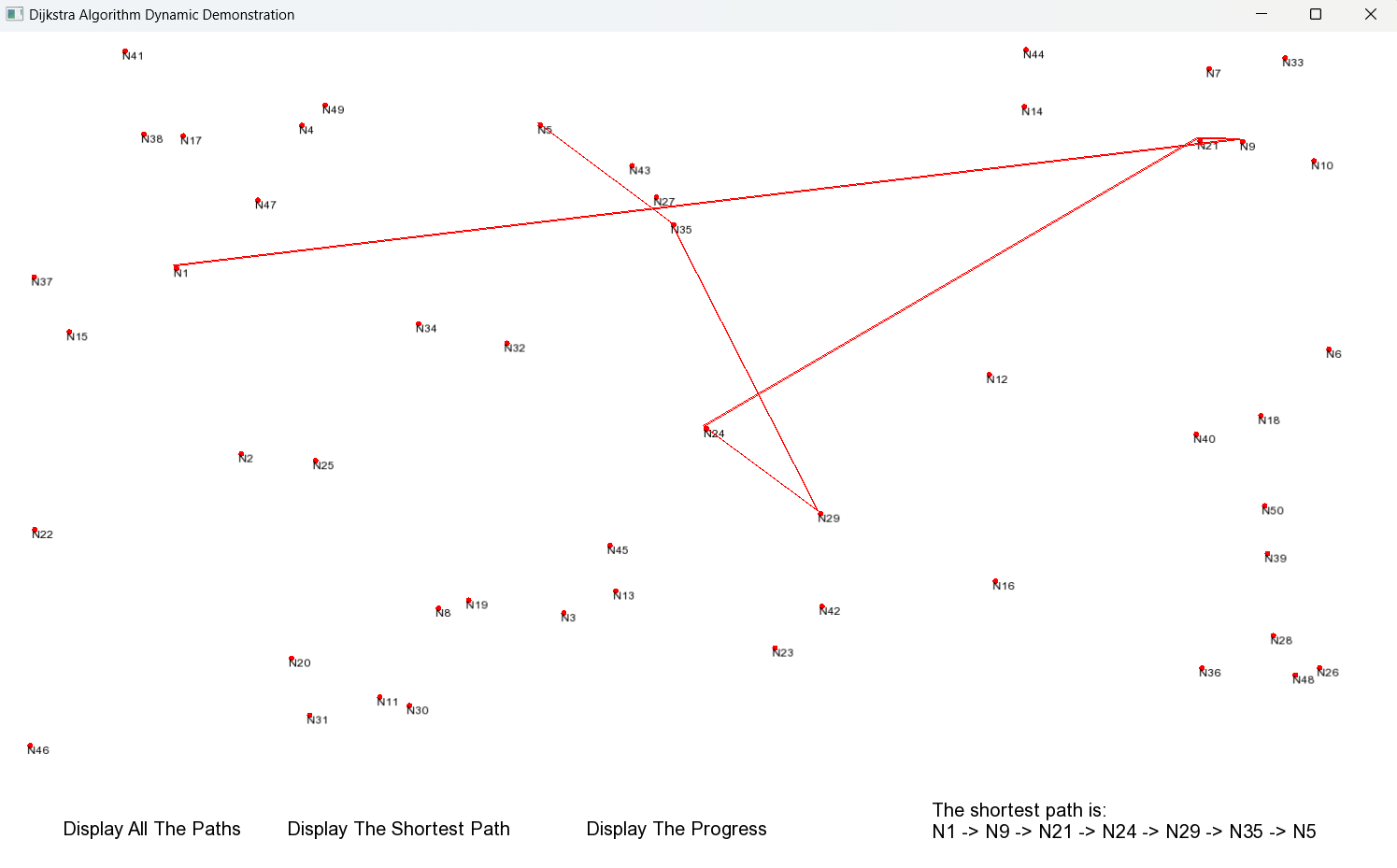
首先点击红线处的可执行文件启动程序；

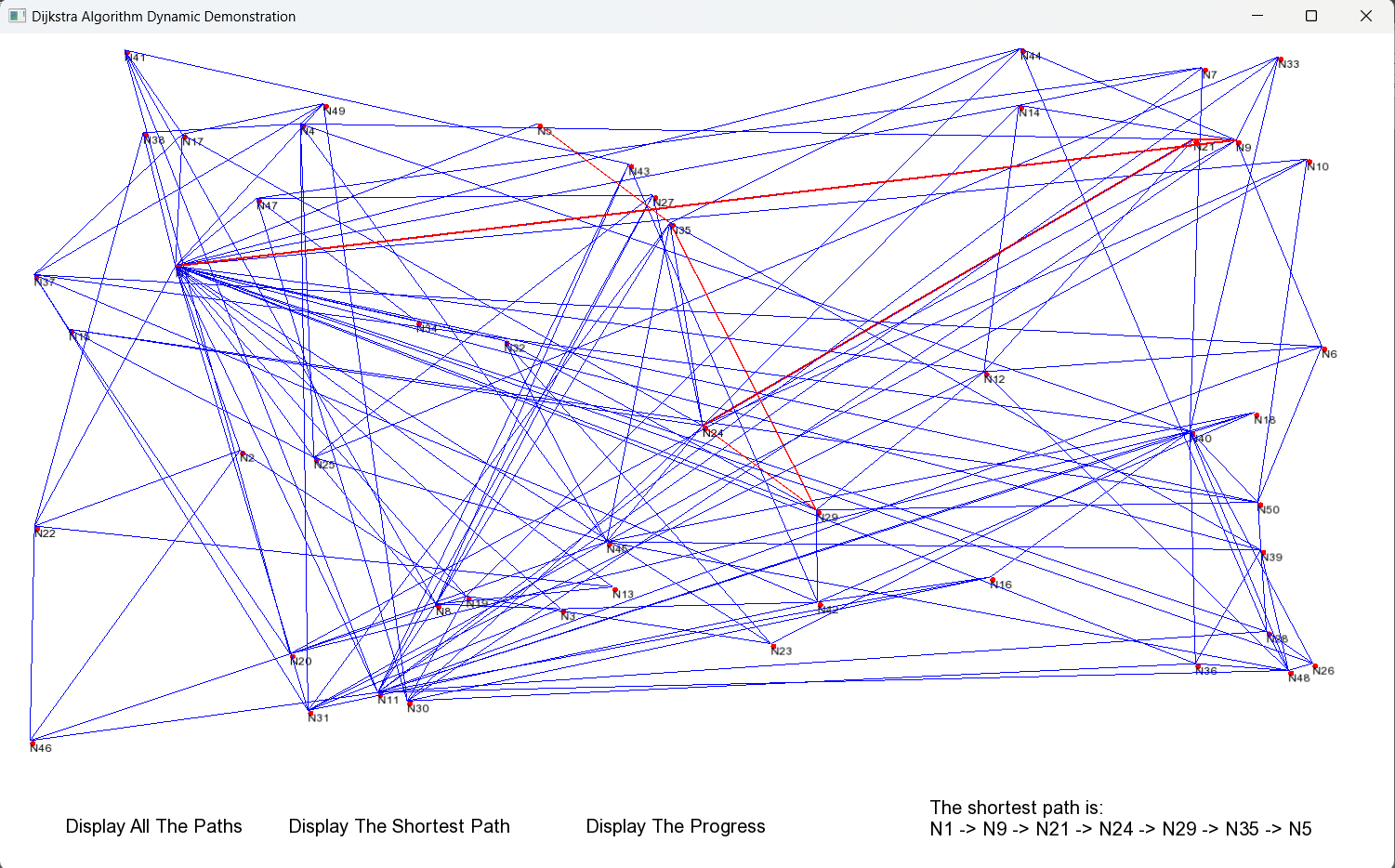


弹出的控制台分别输出了算法的执行过程，即从起始节点开始遍历，True代表更新节点，False代表未更新的节点；

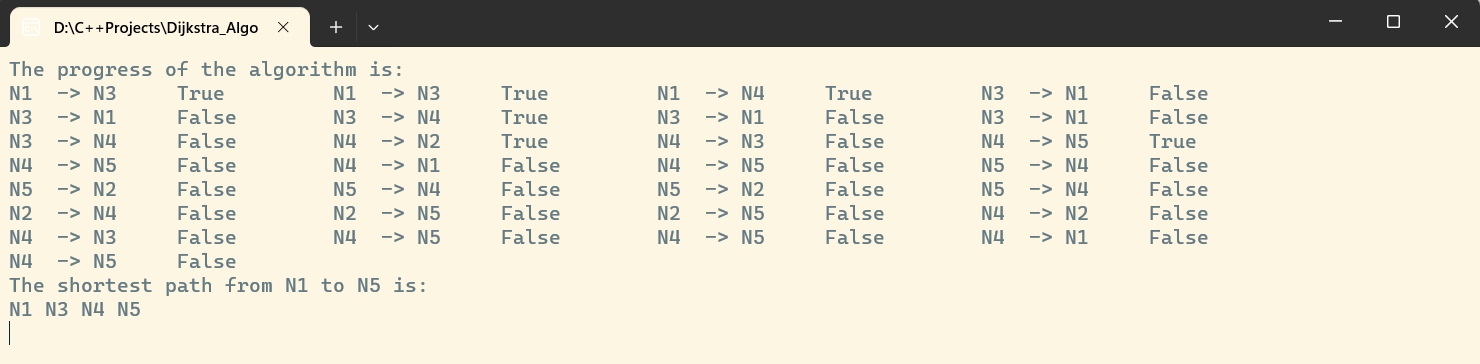
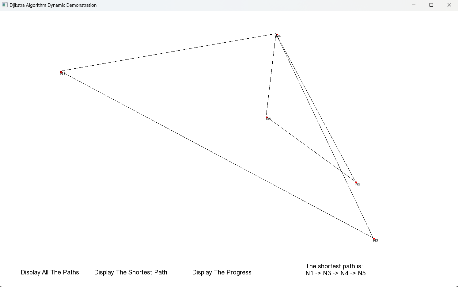
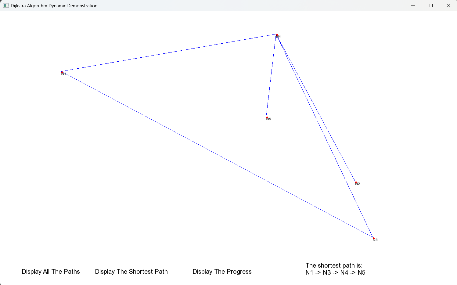
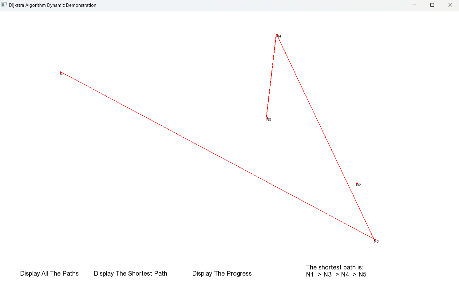


在窗体中可以点击下方的Display按钮用以分别显示所有路径（黑色）、最短路径（红色）和算法遍历过的路径（蓝色），同时窗体中还显示了最短路径所经过的节点。





还可在python文件中修改节点数目与边的数目，我们以较小的节点数为例再次展示Dijkstra算法的过程：



## 4.2 Floyd算法演示

控制台中输出所有的最短路径，窗体中的Distance Matrix表示某节点到其他节点的最短距离，Next Matrix存储了路径。

绿色数字代表初始化时存储的数据，红色数字代表了算法运行过程中更新的数据。

