

# 数据结构课程设计报告

设计题目：网络布线

学生姓名：敬成超

专 业：物联网

班 级：物联网一班

学 号：2023212388

指导教师：胡学钢

完成日期：2023年6月27日

### （二）设计题 6：网络布线

**问题描述：**计算机网络要求网络中的计算机被连接起来，本问题考虑一个“线性”的网络，在这一网络中计算机被连接到一起，并且除了首尾的两台计算机只分别连接着一台计算机外，其它任意一台计算机恰连接着两台计算机。图 3 中用圆点表示计算机，它们的位置用直角坐标表示。网络连接的计算机之间的距离单位为英尺。

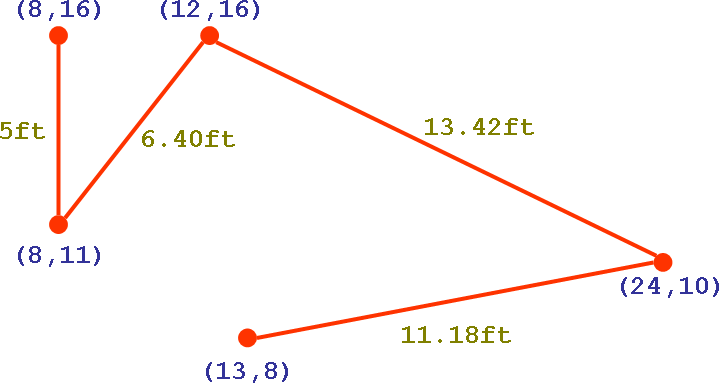


图 3 网络布线示意图

由于很多原因，我们希望使用的电缆长度应可能地短。你的问题是去决定计算机应如何被连接以使你所使用的电缆长度最短。在设计方案施工时，电缆将埋在地下，因此连接两台计算机所要用的电缆总长度等于计算机之间的距离加上额外的 16 英尺电缆，以从地下连接到计算机，并为施工留一些余量。

图 4 是计算机的最优连接方案，这样一个方案用电缆的总长度是

(4 + 16) + (5 + 16) + (5.38 + 16) + (11.18 + 16) = 90.01 英尺

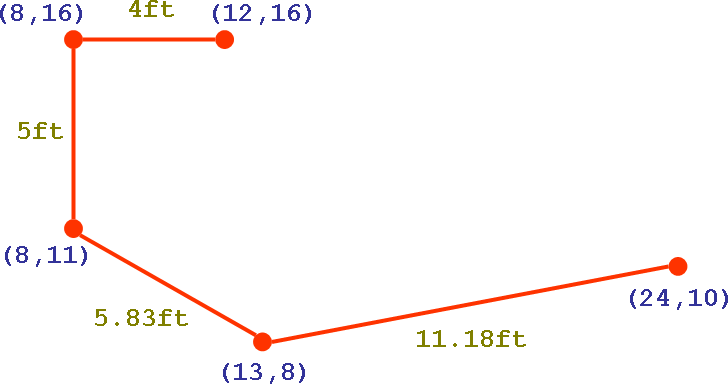


图 4 最优连接方案示意图

**实现要求：**

1. 输入网络中的计算机总数和每台计算机的坐标。
2. 输出使电缆长度最短的连接方案。给出最优连接方案中每两台相邻计算机之间的距离，以及总的电缆长度。
3. 参考图 3、图 4，用图形化的方式显示结果，包括点的坐标、最优路径

、相邻计算机之间的距离，支持界面设置参数与演示。

## 1．问题分析

这个问题是一个典型的优化问题，涉及到计算机之间的最优连接方式以最小化总电缆长度。下面是对这个问题的分析：

1. 问题定义：

- 目标是找到一种连接计算机的方式，使得总电缆长度最短。

- 每台计算机除了首尾两台外，都连接两台计算机。

- 电缆长度包括计算机之间的直线距离加上额外的16英尺。

2. 输入要求：

- 需要输入计算机的总数。

- 需要输入每台计算机的坐标，通常以(x, y)的形式表示。

3. 输出要求：

- 输出最优连接方案中每两台相邻计算机之间的距离。

- 输出总的电缆长度。

4. 额外要求：

- 需要图形化显示结果，包括点的坐标、最优路径、相邻计算机之间的距离。

- 需要支持界面设置参数与演示。

5. 问题难点：

- 确定最优连接方案可能需要考虑所有可能的连接组合，这在计算机数量较多时会非常复杂。

- 需要一种有效的方法来计算和比较不同连接方案的电缆长度。

6. 算法选择：

- 可以考虑使用图论中的最小生成树算法，如Prim算法或Kruskal算法，来找到连接所有计算机的最小成本树。

- 由于每台计算机除了首尾外都连接两台计算机，可能需要对算法进行一些调整以满足这一条件。

7. 实现步骤：

- 首先，将每台计算机视为图中的一个节点。

- 计算每对计算机节点之间的距离，并将其作为边的权重。

- 应用最小生成树算法来找到连接所有节点的最小成本树。

- 调整算法以确保除了首尾节点外，每个节点都恰好连接两台其他计算机。

- 计算并输出最优连接方案的电缆长度和每对相邻计算机之间的距离。

8. 图形化展示：

- 使用图形库来绘制计算机的坐标点。

- 根据最优连接方案绘制连接线，并标注每段电缆的长度。

9. 用户交互：

- 设计一个用户界面，允许用户输入计算机的总数和坐标。

- 提供参数设置选项，如选择不同的算法或调整额外电缆长度。

- 展示最优连接方案的图形化结果，并允许用户进行交互式探索。

通过上述分析，我们可以设计一个算法来解决这个问题，并提供一个用户友好的界面来展示结果。

## 2．设计思想

这段代码是一个使用SFML库实现的图形界面程序，用于计算并显示最小生成树（MST）的解决方案。设计思想主要体现在以下几个方面：

1. 图形用户界面（GUI）：

使用SFML库创建一个窗口，提供图形化的用户界面，允许用户输入数据和查看结果。这里涉及一些基本的函数，例如draw,setingposition,setfillcolor.同时还得注意循环。下面举其中一个例子

for (const auto& edge : edges) {

totalCableLength += edge.dist + 16.0f;

}

std::stringstream totalCableStream;

totalCableStream << "Total Cable Length: " << std::fixed << std::setprecision(2) << totalCableLength << " ft";

totalCableText.setString(totalCableStream.str());

totalCableText.setFillColor(sf::Color::White);

// 确保 totalCableText 的位置设置正确

totalCableText.setPosition(window.getSize().x - totalCableText.getLocalBounds().width - 10, 10);

// 在窗口中绘制 totalCableText

window.draw(totalCableText);

2. 数据输入：

程序通过文本输入接收用户输入的计算机坐标点，使用户能够方便地输入网络中的节点信息。将输入的数据作为输入流，用恰当函数来给变量赋初值。

3. 动态数据处理：

程序在用户输入后动态地创建点（Point）的集合，并根据输入的数量和坐标初始化这些点。使用了vector，让数据动态化，更加简单。

4. 最小生成树算法：

采用Prim算法来计算给定点集的最小生成树。这是一种贪心算法，逐步构建最小成本的树。这是经典算法，通过这个课程设计我掌握了他，实践了它再具体问题的应用。Prim算法是一种用于求解最小生成树（Minimum Spanning Tree, MST）的经典算法。最小生成树是一个连通无向图中包含所有顶点的树，并且其边的权重之和最小。Prim算法的基本思想是从一个顶点开始，逐步扩展生成树，每次选择与当前生成树连接的最小权重边，直到包含所有顶点为止。

以下是Prim算法的基本步骤：

1. **初始化**：选择一个起始顶点，将其加入生成树中。初始化一个集合 T 表示当前生成树的顶点集合，初始时 T 只包含起始顶点。同时，初始化一个集合 E 表示当前生成树的边集合，初始时 E 为空。
2. **选择最小权重边**：在所有连接生成树顶点集合 T 和非生成树顶点集合 V-T 的边中，选择权重最小的边 (u, v)，其中 u 属于 T，v 属于 V-T。
3. **扩展生成树**：将选中的边 (u, v) 加入生成树的边集合 E，并将顶点 v 加入生成树的顶点集合 T。
4. **重复步骤2和3**：重复选择最小权重边并扩展生成树的过程，直到生成树包含图中的所有顶点。
5. **结束**：当生成树包含所有顶点时，算法结束，此时 E 中的边构成最小生成树。

每一次循环都得进行动态数据的更新，如min\_e，sel\_e，used。

5. 结果展示：

将最小生成树的边以红色线条在界面上展示，同时在每条边上显示边的长度。还有再右上角显示实际用的电缆长度。

6. 额外电缆长度：

在计算总电缆长度时，将每条边的长度加上额外的16英尺，以满足题目要求。

7. 实时反馈：

用户输入数据后，程序能够即时计算并显示最小生成树和总电缆长度。

10. 中心视图调整：

根据所有点的中心位置调整视图，以便更好地展示整个网络布局。让视图更加符合人们习惯。

12. 错误处理：

程序在无法加载字体文件时会给出提示，并退出。

13. 代码的模块化：

将不同的功能（如点的定义、距离计算、最小生成树算法等）封装在不同的结构或函数中，提高了代码的可读性和可维护性。

## 3. 设计表示

这是个小问题，故所有代码实现都在一起，这虽然有点简陋，但节省了目录结构，让我更专注于代码的实现

电缆布线

。

GUI

Core

也有对应的头文件，和显示点线代码。

里面有头文件，prim代码，还有好几个类

代码目录结构图如下所示：

1. core 包中的相关类与结构说明：

在core包的中有很多类，，它们的类图分别如下所示（这里简要说明一下举几个例子）：

point类：

|  |  |  |  |
| --- | --- | --- | --- |
| **类名** | **成员类别** | **成员名** | **描述** |
| point | 函数 | distance | 计算两点间距离 |
| 数据 | Int x | 横坐标 |
| Int y | 纵坐标 |
|  |  |
|  |  |

**Edge** 类：

|  |  |  |  |
| --- | --- | --- | --- |
| **类名** | **成员类别** | **成员名** | **描述** |
| edge | 函数 | operator < | 为重载函数，比较距离大小，返回布尔类型 |
|  |  |
|  |  |

|  |  |  |  |
| --- | --- | --- | --- |
| **类名** | **成员类别** | **成员名** | **描述** |
| primMST | 函数 |  | 核心算法实现最小生成树 |
| vector<Point>& points: | 顶点集合 |
| vector<Edge>& result | 用于存储最小生成树的边集合。 |
| vector<float> min\_e: | 记录每个顶点到当前生成树的最小距离 |
| vector<int> sel\_e | 记录每个顶点在生成树中的连接顶点 |
| :vector<bool> used | 记录每个顶点是否已经在生成树中 |
|  |  |

1. 图形相关类与结构说明：

结构 Point

* **概述**: 表示图中的一个顶点，包含两个浮点数成员变量 x 和 y，分别表示顶点的横坐标和纵坐标。

结构 Edge

* **概述**: 表示图中的一条边，包含两个整数成员变量 u 和 v，分别表示边的两个顶点的索引，以及一个浮点数成员变量 dist，表示边的权重（即两个顶点之间的距离）。

函数 distance

* **概述**: 计算两个顶点之间的欧几里得距离，接受两个 Point 对象作为参数，返回它们之间的距离。

函数 primMST

* **概述**: 使用Prim算法计算最小生成树（MST），接受一个顶点集合和一个用于存储结果的边集合作为参数。该函数通过逐步选择最小权重的边来构建MST。

SFML 相关类

* **sf::RenderWindow**: 表示一个窗口，用于显示图形。提供了创建窗口、清除窗口内容、绘制图形对象和显示绘制内容的功能。
* **sf::Font**: 表示字体，用于在窗口中显示文本。提供了从文件加载字体的功能。
* **sf::Text**: 表示文本对象，用于在窗口中绘制文本。提供了设置文本内容、位置和颜色的功能。
* **sf::Vertex**: 表示顶点对象，用于绘制线条。提供了创建顶点对象的功能。
* **sf::CircleShape**: 表示圆形对象，用于绘制顶点。提供了创建圆形对象、设置位置和颜色的功能。

SFML类：

|  |  |  |  |
| --- | --- | --- | --- |
| **类名** | **成员类别** | **成员名** | **函数** |
| SFML 相关类 | 函数 | RenderWindow | 构造函数，创建窗口。 |
| clear(const sf::Color& color = sf::Color(0, 0, 0, 255)) | 清除窗口内容。 |
| draw(const sf::Drawable& drawable, const sf::RenderStates& states = sf::RenderStates::Default) | 绘制图形对象。 |
| display() | 显示绘制的内容。 |

Text类

|  |  |  |  |
| --- | --- | --- | --- |
| 类名 | 成员类别 | 成员名 | 函数 |
| **Text** |  | Text | 构造函数，创建文本对象。 |
| setString | 设置文本内容 |
| setPosition | 设置文本位置 |
| setFillColor | 设置文本颜色 |
|  |  |

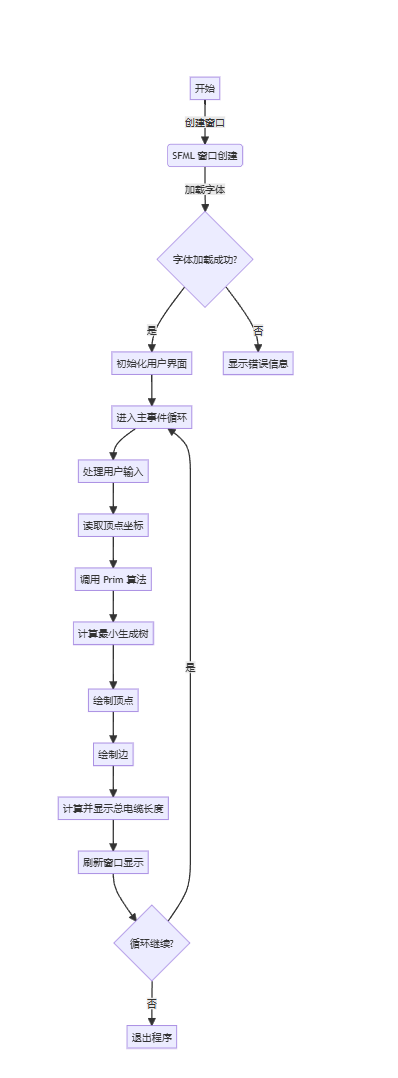
## 4. 核心算法

Prim算法是一种用于求解加权无向图的最小生成树的经典算法。最小生成树是一个包含图中所有顶点的树，并且其边的权重之和最小。Prim算法的基本思想是从一个顶点开始，逐步扩展生成树，每次选择与当前生成树连接的最小权重边，直到包含所有顶点为止。

Prim算法步骤

1. **初始化**：
   * 选择一个起始顶点，将其加入生成树中。
   * 初始化一个集合 T 表示当前生成树的顶点集合，初始时 T 只包含起始顶点。
   * 初始化一个集合 E 表示当前生成树的边集合，初始时 E 为空。
   * 初始化一个数组 min\_e，用于记录每个顶点到当前生成树的最小距离，初始值为无穷大（std::numeric\_limits<float>::max()），起始顶点的 min\_e 值为0。
   * 初始化一个数组 sel\_e，用于记录每个顶点在生成树中的连接顶点，初始值为-1。
   * 初始化一个数组 used，用于记录每个顶点是否已经在生成树中，初始值为 false。
2. **选择最小权重边**：
   * 在所有连接生成树顶点集合 T 和非生成树顶点集合 V-T 的边中，选择权重最小的边 (u, v)，其中 u 属于 T，v 属于 V-T。
3. **扩展生成树**：
   * 将选中的边 (u, v) 加入生成树的边集合 E，并将顶点 v 加入生成树的顶点集合 T。
   * 更新 used[v] 为 true，表示顶点 v 已经在生成树中。
   * 更新 sel\_e[v] 为 u，表示顶点 v 是通过顶点 u 加入生成树的。
   * 更新 min\_e[v] 为边的权重 dist，表示顶点 v 到生成树的最小距离。
4. **重复步骤2和3**：
   * 重复选择最小权重边并扩展生成树的过程，直到生成树包含图中的所有顶点。
5. **结束**：
   * 当生成树包含所有顶点时，算法结束，此时 E 中的边构成最小生成树。

④ 程序的相关流程图如下：



（2）用户GUI界面的设计思路

用户GUI界面的设计思路主要包括以下几个方面：

1. 界面布局

* **窗口大小和位置**：确定窗口的大小和位置，使其适合显示图形和文本信息。在本例中，窗口大小设置为800x600像素。
* **视图设置**：根据用户输入的顶点坐标，动态调整视图以确保所有顶点和边都能在窗口中显示。

2. 用户输入处理

* **文本输入**：允许用户通过键盘输入顶点坐标。使用 sf::Event::TextEntered 事件处理用户输入，支持退格键删除输入，并将输入存储在 inputBuffer 中。
* **回车键确认**：当用户按下回车键时，解析 inputBuffer 中的输入，提取顶点坐标，并计算最小生成树。

3. 图形绘制

* **顶点绘制**：使用 sf::CircleShape 绘制顶点，每个顶点用一个小圆圈表示，并显示其坐标。
* **边绘制**：使用 sf::Vertex 绘制边，每条边用一条直线表示，并在边的中点显示其权重（距离）。
* **文本显示**：使用 sf::Text 显示提示信息、顶点坐标、边的权重以及总的电缆长度。

4. 交互反馈

* **提示信息**：在用户输入过程中，显示提示信息，指导用户输入顶点坐标。
* **结果显示**：在计算完成后，显示最小生成树的边和顶点，以及总的电缆长度。

5. 性能优化

* **动态调整视图**：根据顶点的分布动态调整视图，确保所有顶点和边都能在窗口中显示。
* **文本位置调整**：根据文本内容动态调整文本的位置，确保文本不会重叠或超出窗口边界。

6. 错误处理

* **字体加载**：在加载字体时进行错误处理，如果字体加载失败，输出错误信息并退出程序。
* **输入解析**：在解析用户输入时进行错误处理，确保输入的格式正确。

7. 用户友好性

* **简洁明了**：界面设计简洁明了，避免过多的复杂元素，使用户能够快速理解和使用。
* **一致性**：保持界面元素的一致性，例如使用相同的颜色和字体风格

（3）软件整体的异常检测及成功/失败反馈功能的实现

1. 输入验证

* **格式检查**：在用户输入顶点坐标时，检查输入格式是否正确。例如，确保输入的是数字，并且每个顶点有对应的x和y坐标。
* **范围检查**：检查输入的坐标值是否在合理的范围内。例如，确保坐标值不会过大或过小，避免图形显示异常。

2. 错误处理

* **字体加载错误**：在加载字体文件时，使用 sf::Font::loadFromFile 方法，并检查返回值。如果加载失败，输出错误信息并退出程序。
* **输入解析错误**：在解析用户输入时，使用 std::istringstream 进行解析，并检查解析结果。如果解析失败，输出错误信息并提示用户重新输入。

3. 异常检测

* **空输入检测**：在用户按下回车键时，检查 inputBuffer 是否为空。如果为空，提示用户输入顶点坐标。
* **顶点数量检测**：在解析顶点坐标时，检查输入的顶点数量是否符合要求。例如，确保至少有两个顶点才能计算最小生成树。

（三）用户手册

# 系统要求

# 操作系统：Windows 7 或更高版本

# 处理器：1 GHz 或更快

# 内存：1 GB RAM

# 图形：支持 OpenGL 2.1 或更高版本的显卡

# 存储空间：100 MB 可用空间

# 安装与启动

# 下载软件安装包并解压到任意目录。

# 双击运行 NetworkOptimization.exe 文件。

# 界面介绍

# 主窗口：显示图形和文本信息。

# 输入框：用于输入顶点坐标。

# 提示信息：指导用户输入顶点坐标。

# 状态栏：显示当前操作的状态。

# 使用步骤

# 启动软件：双击 NetworkOptimization.exe 文件启动软件。

# 输入顶点坐标：

# 在输入框中输入顶点数量和坐标。例如：

# 3

# 100 100

# 200 200

# 300 100

# 按下回车键确认输入。

# 查看结果：

# 软件将显示最小生成树的边和顶点。

# 显示总的电缆长度。

# 输入格式

# 输入的第一行是顶点的数量。

# 接下来的每一行是每个顶点的x和y坐标，用空格分隔。

# 错误处理

# 如果输入格式不正确，软件将显示错误信息并提示重新输入。

# 如果字体加载失败，软件将显示错误信息并退出。

# 常见问题

# 为什么输入后没有反应？

# 请检查输入格式是否正确，确保每个顶点有对应的x和y坐标。

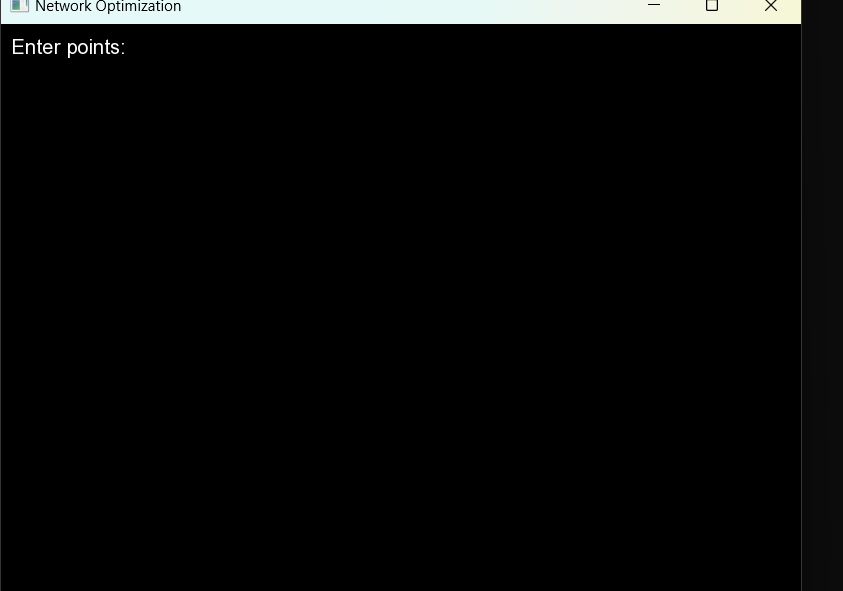
# 如何重新输入？

# 可以直接在输入框中修改输入内容，然后按下回车键确

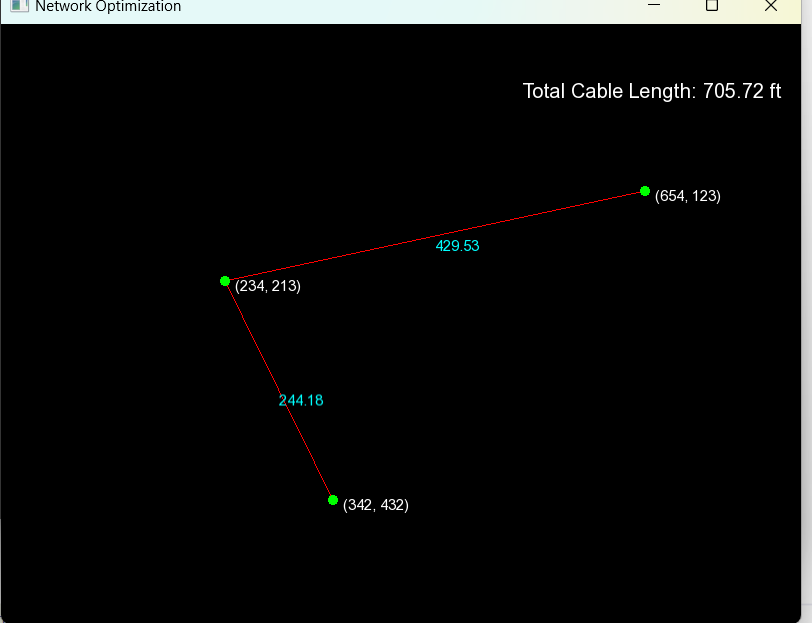
# （四）调试及测试

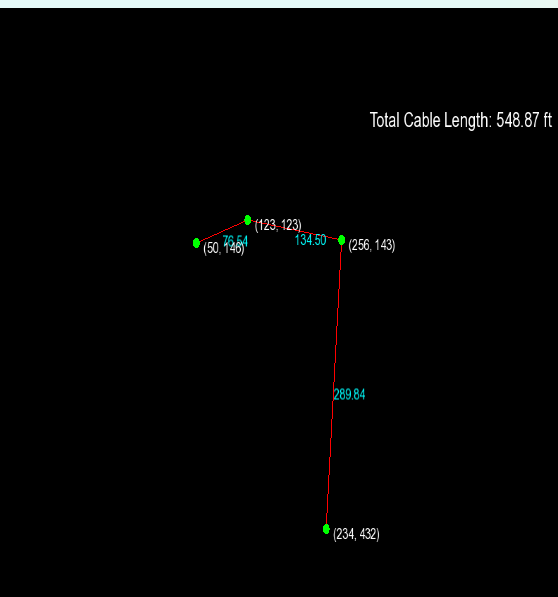
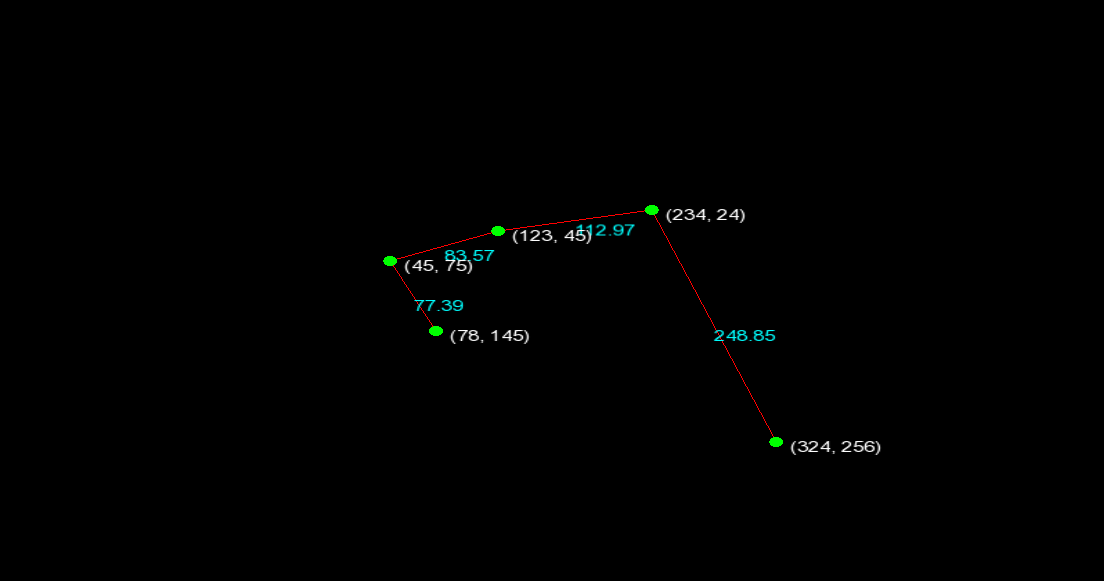
## 1. 部分程序运行截图：

主界面截图



结果截图





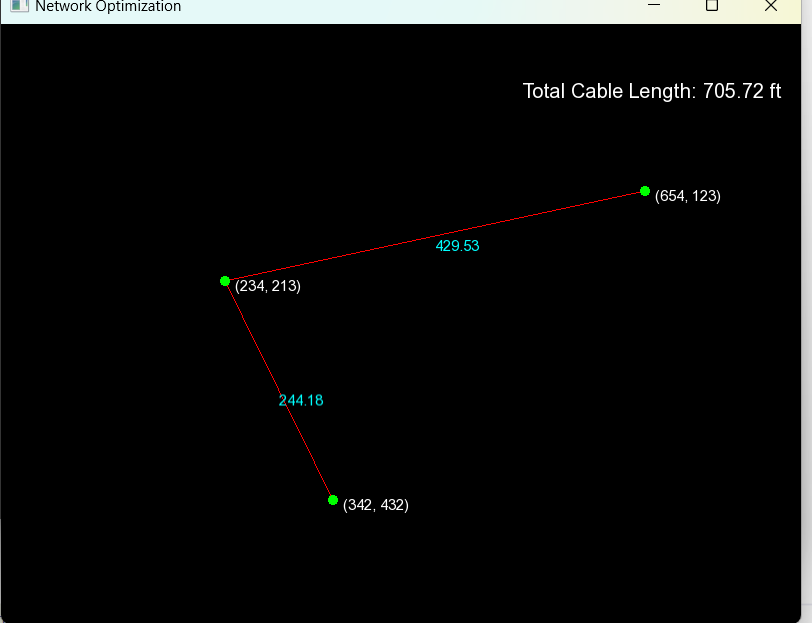
## 测试数据

（342，432）（234，213）（654，123）

(78,145)(45,75)(123,45)(234,24)(324,256)

(50,146)(123,123)(256,143)(234,432)

## 测试结果及运行截图演示：

1. 

## 进一步改进：

1. 1. 增加图形界面元素
2. **按钮**：添加“计算”按钮，用户输入完成后点击按钮进行计算，而不是依赖回车键。
3. **清除输入**：添加“清除”按钮，方便用户清除当前输入，重新开始。
4. **保存结果**：添加“保存”按钮，允许用户将计算结果保存为文件。
5. 2. 增强输入验证
6. **实时验证**：在用户输入过程中实时验证输入格式，并在输入框下方显示错误提示。
7. **坐标范围限制**：限制输入的坐标值在合理范围内，避免图形显示异常。
8. 3. 优化错误处理
9. **详细错误信息**：在显示错误信息时，提供更详细的错误描述，帮助用户快速定位问题。
10. **自动恢复**：在检测到错误后，提供自动恢复选项，例如清除错误输入并提示用户重新输入。
11. 4. 增加功能
12. **导入导出**：允许用户从文件导入顶点坐标，或将结果导出到文件。
13. **多语言支持**：增加多语言支持，满足不同用户的需求。
14. 5. 提升性能
15. **优化算法**：优化Prim算法，提高计算速度，特别是在处理大量顶点时。
16. **动态调整窗口大小**：根据顶点数量和屏幕分辨率动态调整窗口大小，确保所有内容都能显示。
17. 6. 用户反馈
18. **满意度调查**：在软件中添加满意度调查链接，收集用户反馈，持续改进软件。
19. **更新提示**：在软件启动时检查更新，并提示用户下载最新版本。
20. 7. 用户手册更新
21. **详细操作指南**：在用户手册中增加详细的操作指南，包括每个按钮的功能和使用方法。
22. **常见问题解答**：更新常见问题解答，涵盖更多用户可能遇到的问题。

# (五) 感想

1. **用户为中心的设计**：在整个设计过程中，我始终将用户的需求和体验放在首位。通过不断优化界面和功能，确保软件易于使用且功能强大，从而提升用户满意度。
2. **细节决定成败**：在设计用户界面和交互流程时，我意识到细节的重要性。一个小小的按钮位置或提示信息的措辞都可能影响用户的使用体验。因此，我花费了大量时间来调整和优化这些细节。
3. **持续学习和改进**：软件设计是一个不断学习和改进的过程。在设计过程中，我不断查阅资料，学习新的设计理念和技术，以确保软件的先进性和实用性。
4. **团队合作的重要性**：虽然这个设计是由我个人完成的，但我深刻理解到在实际项目中团队合作的重要性。与团队成员的有效沟通和协作能够大大提高工作效率和设计质量。
5. **测试和反馈的必要性**：在设计完成后，我意识到测试和用户反馈的重要性。通过测试可以发现潜在的问题，而用户反馈则可以帮助我了解用户的需求和期望，从而进一步优化设计。
6. **创新和创造力**：在设计过程中，我尝试引入一些创新的想法和功能，以提升软件的独特性和竞争力。这不仅需要技术知识，还需要创造力和对市场的敏锐洞察。

总的来说，这个设计过程是一次宝贵的学习和成长经历。它不仅提升了我的技术能力，也锻炼了我的设计思维和解决问题的能力。我期待将这些经验应用到未来的项目中，不断创造出更好的软件产品。

# 附录：（代码单独打包与文档一起提交）

代码

Main.cpp

#include <SFML/Graphics.hpp>

#include <iostream>

#include <vector>

#include <cmath>

#include <limits>

#include <sstream>

#include <iomanip>

struct Point {

float x, y;

Point(float xCoord, float yCoord) : x(xCoord), y(yCoord) {}

};

float distance(const Point& a, const Point& b) {

return std::sqrt((a.x - b.x) \* (a.x - b.x) + (a.y - b.y) \* (a.y - b.y));

}

struct Edge {

int u, v;

float dist;

Edge(int u, int v, float dist) : u(u), v(v), dist(dist) {}

bool operator < (const Edge& e) const {

return dist < e.dist;

}

};

void primMST(std::vector<Point>& points, std::vector<Edge>& result) {

int n = points.size();

std::vector<float> min\_e(n, std::numeric\_limits<float>::max());

std::vector<int> sel\_e(n, -1);

std::vector<bool> used(n, false);

min\_e[0] = 0;

for (int i = 0; i < n; ++i) {

int v = -1;

for (int j = 0; j < n; ++j) {

if (!used[j] && (v == -1 || min\_e[j] < min\_e[v]))

v = j;

}

used[v] = true;

if (sel\_e[v] != -1)

result.push\_back(Edge(sel\_e[v], v, min\_e[v]));

for (int to = 0; to < n; ++to) {

float dist = distance(points[v], points[to]);

if (dist < min\_e[to] && !used[to]) {

min\_e[to] = dist;

sel\_e[to] = v;

}

}

}

}

int main() {

sf::RenderWindow window(sf::VideoMode(800, 600), "Network Optimization");

sf::Font font;

if (!font.loadFromFile("C:\\Windows\\Fonts\\arial.ttf")) {

std::cout << "Could not load font\n";

return 1;

}

std::vector<Point> points;

std::vector<Edge> edges;

std::string inputBuffer;

sf::Text text("", font, 20);

text.setFillColor(sf::Color::White);

bool inputComplete = false;

float totalCableLength = 0.0f;

sf::Text totalCableText("", font, 20);

totalCableText.setFillColor(sf::Color::White);

while (window.isOpen()) {

sf::Event event;

while (window.pollEvent(event)) {

if (event.type == sf::Event::Closed)

window.close();

else if (event.type == sf::Event::TextEntered) {

if (event.text.unicode == '\b') {

if (!inputBuffer.empty()) {

inputBuffer.pop\_back();

}

}

else if (event.text.unicode < 128) {

inputBuffer += static\_cast<char>(event.text.unicode);

}

}

else if (event.type == sf::Event::KeyPressed && event.key.code == sf::Keyboard::Enter) {

std::istringstream ss(inputBuffer);

int count;

ss >> count;

float x, y;

points.clear();

for (int i = 0; i < count; i++) {

ss >> x >> y;

points.emplace\_back(x, y);

}

inputComplete = true;

edges.clear();

primMST(points, edges);

totalCableLength = 0.0f;

for (const auto& edge : edges) {

totalCableLength += edge.dist + 16.0f;

}

std::stringstream totalCableStream;

totalCableStream << "Total Cable Length: " << std::fixed << std::setprecision(2) << totalCableLength << " ft";

totalCableText.setString(totalCableStream.str());

float centerX = 0, centerY = 0;

for (const auto& point : points) {

centerX += point.x;

centerY += point.y;

}

centerX /= points.size();

centerY /= points.size();

sf::View view(sf::Vector2f(centerX, centerY), sf::Vector2f(800, 600));

window.setView(view);

inputBuffer.clear();

text.setString("");

}

}

window.clear(sf::Color::Black);

for (const auto& edge : edges) {

sf::Vertex line[] = {

sf::Vertex(sf::Vector2f(points[edge.u].x, points[edge.u].y), sf::Color::Red),

sf::Vertex(sf::Vector2f(points[edge.v].x, points[edge.v].y), sf::Color::Red)

};

window.draw(line, 2, sf::Lines);

std::stringstream ss;

ss << std::fixed << std::setprecision(2) << edge.dist;

sf::Text distText(ss.str(), font, 15);

distText.setPosition((points[edge.u].x + points[edge.v].x) / 2, (points[edge.u].y + points[edge.v].y) / 2);

distText.setFillColor(sf::Color::Cyan);

window.draw(distText);

}

for (const auto& point : points) {

sf::CircleShape circle(5);

circle.setPosition(point.x - 5, point.y - 5);

circle.setFillColor(sf::Color::Green);

window.draw(circle);

std::stringstream ss;

ss << "(" << point.x << ", " << point.y << ")";

sf::Text coordText(ss.str(), font, 15);

coordText.setPosition(point.x + 10, point.y - 5);

coordText.setFillColor(sf::Color::White);

window.draw(coordText);

}

text.setString(inputComplete ? "" : "Enter points: " + inputBuffer);

text.setPosition(10, 10);

if (!inputComplete) {

window.draw(text);

}

totalCableText.setPosition(window.getSize().x - totalCableText.getLocalBounds().width - 10, 10);

window.draw(totalCableText);

window.display();

}

return 0;

}